**Informatik 12**
**September 2021**

# Boolesche Terme und Schaltalgebra

## 1. Datentyp boolean

Eine Boolesche Variable oder ein Boolescher Ausdruck (Term) nimmt nur zwei Werte an:
**`True`** oder **`False`**
(abkürzend: 1 oder 0; in Python sind **`True`** oder **`False`** zu verwenden)

Insbesondere sind folgende Terme Boolesche Ausdrücke, deren Wert sich auch einer Variablen zuweisen läßt:

**`8 > 5`** hat den Wert **`True`**
**`7 == 8`** hat den Wert **`False`**
**`7 != 8`** hat den Wert **`True`**
**`x`** hat den Wert **`True`** nach der Wertzuweisung **`x = 7 < 12`**
**`x`** hat den Wert **`False`** nach der Wertzuweisung **`x = (0 == 6)`**
**`a or b`** hat den Wert **`True`** genau dann, wenn mindestens eine der Variablen **a**, **b** den Wert **`True`** hat; andernfalls hat **`a or b`** den Wert **`False`**.

Mit **`a = 7 != 8`** oder **`a = (7 != 8)`** wird in Python der Booleschen Variablen **a** der Wert des Booleschen Terms **`7 != 8`** (hier: **`True`**) zugewiesen.

Wir definieren die Verknüpfungen **`and`** und **`or`** sowie die Operation **`not`** jeweils über eine Wahrheitstafel:

| a | b | a or b |
|---|---|---|
| False | False | False |
| False | True | True |
| True | False | True |
| True | True | True |

| a | b | a and b |
|---|---|---|
| False | False | False |
| False | True | False |
| True | False | False |
| True | True | True |

| a | not a |
|---|---|
| False | True |
| True | False |

Abkürzende Schreibweisen (a, b, c sind Boolesche Variable oder Boolesche Terme):

$$\text{a and b} = a \wedge b = a \cdot b = a\,b$$
$$\text{a or b} = a \vee b = a + b$$
$$\text{not a} = \neg\, a = \overline{a}$$

Dabei gelte auch die aus der Algebra bekannte Vereinbarung “Punkt vor Strich”, d. h.

$$a + (b \cdot c) = a + b \cdot c = a + b\,c$$

Die **AND**-Verknüpfung nennen wir auch **Konjunktion**,
die **OR**-Verknüpfung **Disjunktion**.

## 2. Rechenregeln für Boolesche Variable

**Kommutativgesetz**

(1) $a + b = b + a$ (1′) $a \cdot b = b \cdot a$

**Assoziativgesetz**

(2) $a + (b + c) = (a + b) + c$ (2′) $a \cdot (b \cdot c) = (a \cdot b) \cdot c$

**Distributivgesetz**

(3) $a \cdot (b + c) = a \cdot b + a \cdot c$ (3′) $a + b \cdot c = (a + b)\cdot(a + c)$

**Absorptionsgesetz**

**(4)** $a(a + b) = a$ **(4')** $a + ab = a$

**Tautologie**

**(5)** $a \cdot a = a$ **(5')** $a + a = a$

**Gesetz über die Negation**

**(6)** $\overline{a} \cdot a = 0$ **(6')** $\overline{a} + a = 1$

**Doppelte Negation**

**(7)** $\overline{\overline{a}} = a$

**Gesetz von De Morgan**

**(8)** $\overline{a \cdot b} = \overline{a} + \overline{b}$ **(8')** $\overline{a + b} = \overline{a} \cdot \overline{b}$

**Operationen mit 0 und 1**

**(9.1)** $a \cdot 1 = a$ **(9.1')** $a + 0 = a$

**(9.2)** $a \cdot 0 = 0$ **(9.2')** $a + 1 = 1$

**(9.3)** **not 0 = 1** **(9.3')** **not 1 = 0**

**Bemerkung:** Die jeweils in einer Zeile stehenden Gesetze sind duale Gesetze voneinander; Beispiel: (3') ist das duale Gesetz von (3), (3) das duale Gesetz von (3').

**Beweis** von Rechengesetz (3):

| a | b | c | b + c | a(b + c) | ab | ac | ab + ac |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Da die Spalten zu **a(b + c)** und **ab + ac** übereinstimmen, gilt: **a(b + c) = ab + ac** .

**Aufgaben:**

1. Beweise das Distributivgesetz (3').
2. Beweise die Gesetze von De Morgan.
   Hinweis: Wahrheitstafel; außer den Spalten für a und b (4 Zeilen) erstelle Spalten für $a \cdot b$, $\overline{a \cdot b}$, $\overline{a}$, $\overline{b}$, $\overline{a} + \overline{b}$ für Regel (8).
3. Unter der Disjunktion **a or b** versteht man das nichtausschließende **oder** („non-exclusive or"), d. h., **a or b** ist genau dann **True**, falls **a** oder **b** oder sowohl **a** als auch **b** **True** sind („oder" im Sinne von lat. vel).
   Unter der Verknüpfung **a xor b** (andere Schreibweise: $a \oplus b$ ) versteht man das ausschließende **oder** (e**x**clusive **or**), d. h., $a \oplus b$ ist genau dann **True**, falls entweder **a** oder **b** den Wert **True** hat.
   Zeige: $a \oplus b = \overline{a} \cdot b + a \cdot \overline{b}$

### BEISPIEL 1

Die Boolesche Funktion
**z = f(a,b,c)**
ist durch nebenstehende Wahrheitstafel gegeben:

| a | b | c | z |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

a) Ermittle die disjunktive Normalform (**DNF**; Disjunktion von Konjunktionen) für **z**.

b) Vereinfache den Funktionsterm unter Anwendung der Booleschen Rechengesetze.

c) Zeichne den Schaltplan für die optimierte Funktion **z**.

**Lösung:**

a) $z = \bar{a}\cdot\bar{b}\cdot\bar{c} + \bar{a}\cdot\bar{b}\cdot c + \bar{a}\cdot b\cdot c + a\cdot b\cdot c$

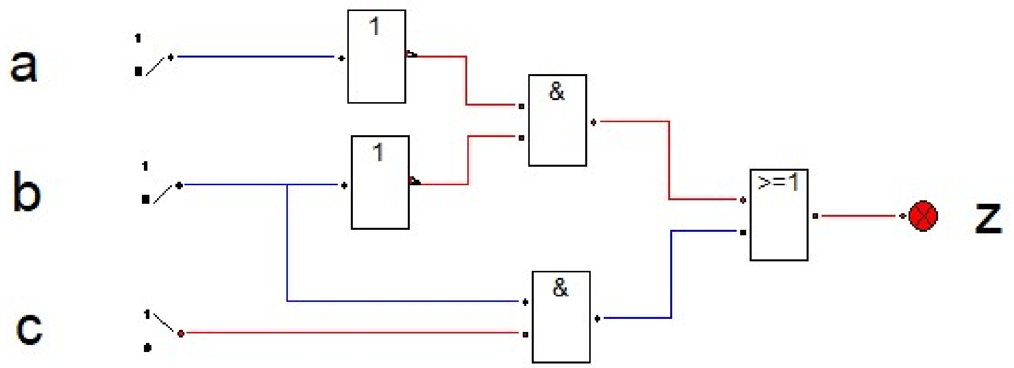
b) $z = \bar{a}\cdot\bar{b}\cdot(\bar{c} + c) + b\cdot c\cdot(\bar{a} + a)$ *Kommutativ- und Distributivgesetz*

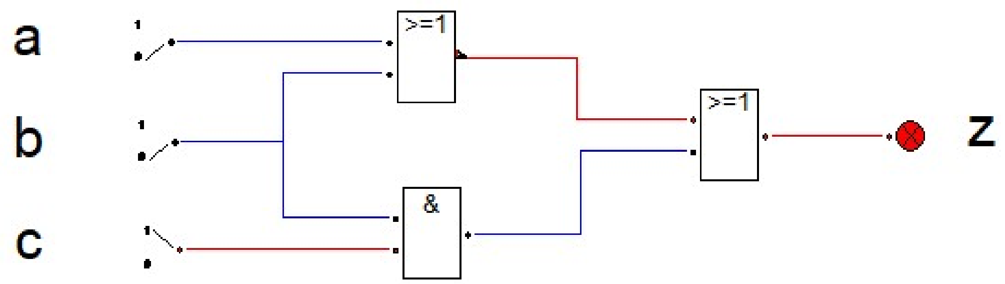
$= \bar{a}\cdot\bar{b}\cdot 1 + b\cdot c\cdot 1$

$= \bar{a}\cdot\bar{b} + b\cdot c$

$= \overline{a+b} + b\cdot c$ *de Morgan's Gesetz*

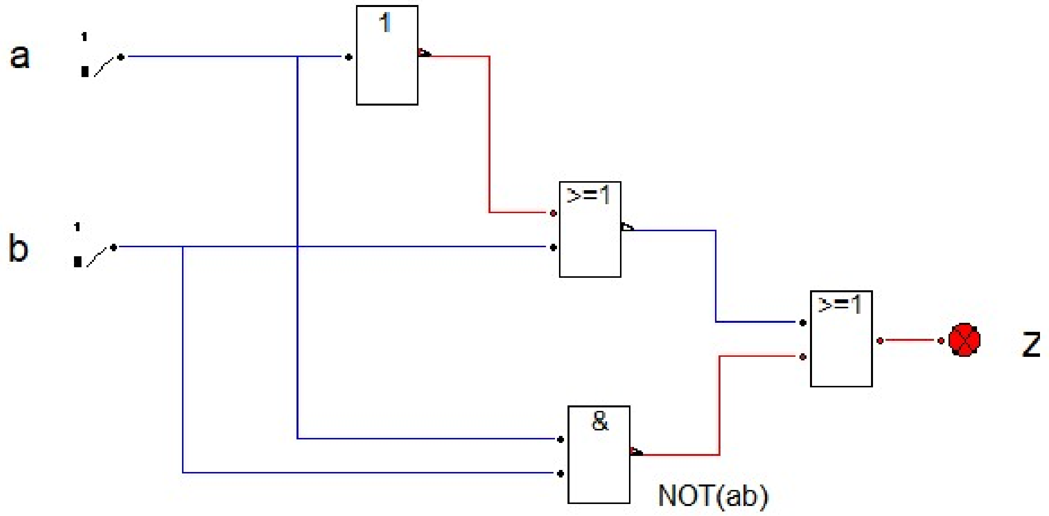
c) $z = \bar{a}\cdot\bar{b} + b\cdot c$ (oben) $z = \overline{a+b} + b\cdot c$ (unten)



## Beispiel 2

Gegeben ist folgende digitale Schaltung mit den Eingangsvariablen **a**, **b** und der Ausgangsvariablen **z**:



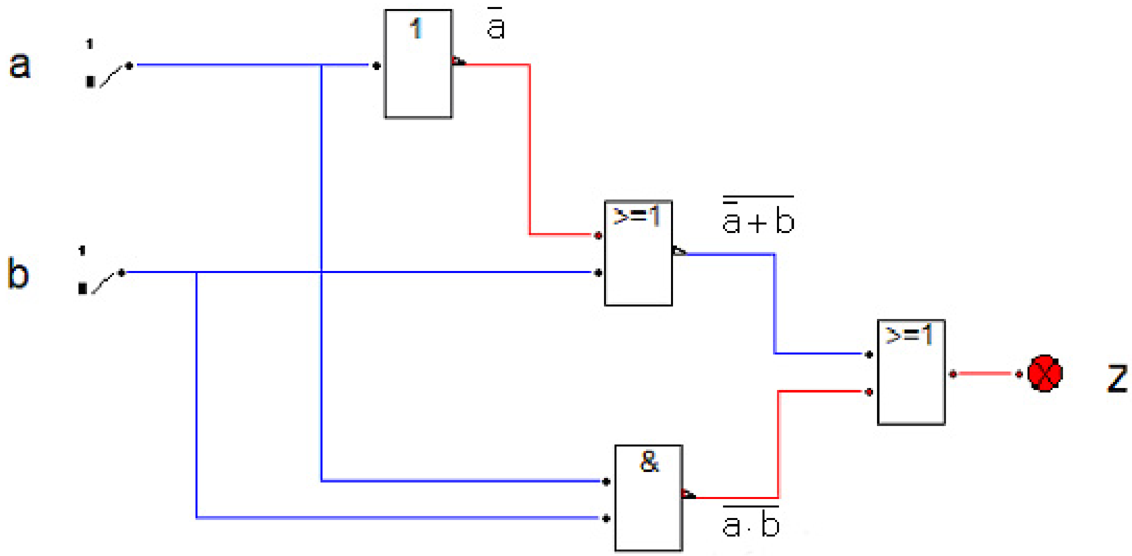

a) Ermittle den Booleschen Term für die Boolesche Funktion **z = f(a,b,c)**. Hinweis: Notiere am Ausgang jedes Gatters jeweils den Booleschen Term (Beispiel: $\overline{a \cdot b}$ am Ausgang des NAND-Gatters).

b) Vereinfache den in a) erhaltenen Term unter Verwendung der Rechenregeln für Boolesche Ausdrücke;

c) Erstelle die Wahrheitstafel und zeichne das Schaltbild für den vereinfachten Funktionsterm; teste beide Schaltungsvarianten mit einem Digitalsimulationsprogramm.

**Lösung:**

zu a):

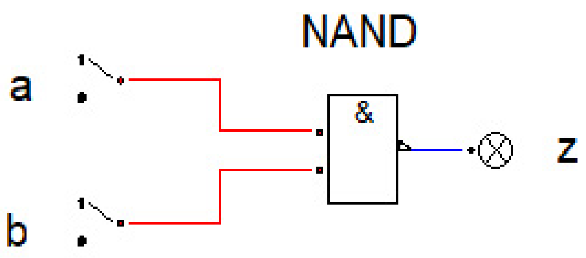


zu b):

$$
\begin{aligned}
z &= \overline{\overline{a}+b}+\overline{a\cdot b} \\
&= \overline{\overline{a}}\cdot\overline{b}+(\overline{a}+\overline{b}) && \textit{(2-mal de Morgan)} \\
&= a\cdot\overline{b}+\overline{a}+\overline{b} && \textit{(wegen } \overline{\overline{a}}=a\textit{)} \\
&= \overline{a}+\overline{b}\cdot a+\overline{b} && \textit{(Kommutativgesetze)} \\
&= \overline{a}+\overline{b}\cdot a+\overline{b}\cdot 1 && \textit{(wegen } a=a\cdot 1\textit{)} \\
&= \overline{a}+\overline{b}\cdot(a+1) && \textit{(Distributivgesetz)} \\
&= \overline{a}+\overline{b} && \textit{(wegen } a+1=1\textit{)} \\
&= \overline{a\cdot b} && \textit{(de Morgan)}
\end{aligned}
$$

zu c):

*optimierte Schaltung:*

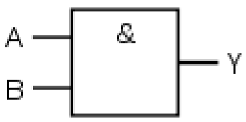
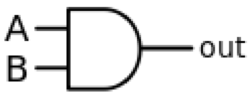
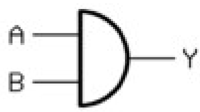
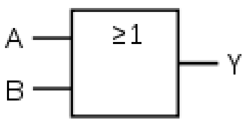
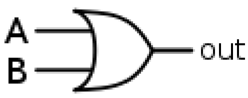
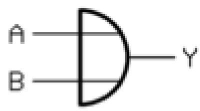
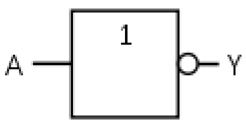
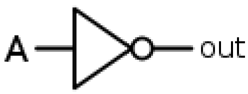
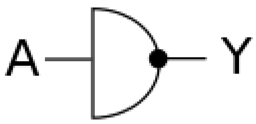
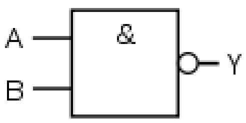
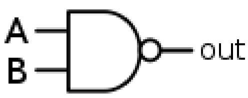
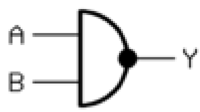
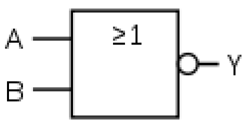
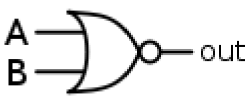
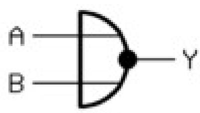
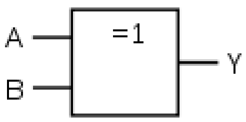
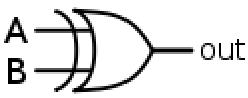
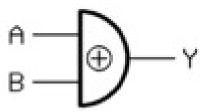



*Wertetabelle:*

| a | b | z |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

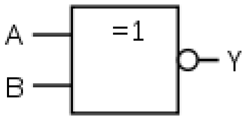
# Typen von Logikgattern und Symbolik

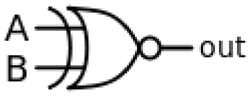
Logikgatter werden mit Schaltsymbolen bezeichnet, die nach unterschiedlichen, mehr oder weniger parallel existierenden Standards definiert sind.

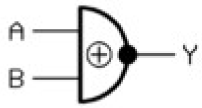
<table>
<tr><th rowspan="2">Name</th><th rowspan="2">Funktion</th><th colspan="3">Symbol in Schaltplan</th><th rowspan="2">Wahrheits-<br>tabelle</th></tr>
<tr><th><u>IEC</u> 60617-12 : 1997 & <u>ANSI</u>/IEEE Std 91/91a-1991</th><th><u>ANSI</u>/IEEE Std 91/91a-1991</th><th><u>DIN</u> 40700 (vor 1976)</th></tr>
<tr><td><u>Und-Gatter</u><br>(AND)</td><td>$Y=A\cdot B$</td><td></td><td></td><td></td><td>A B Y<br>0 0 0<br>0 1 0<br>1 0 0<br>1 1 1</td></tr>
<tr><td><u>Oder-Gatter</u><br>(OR)</td><td>$Y=A+B$</td><td></td><td></td><td></td><td>A B Y<br>0 0 0<br>0 1 1<br>1 0 1<br>1 1 1</td></tr>
<tr><td><u>Nicht-Gatter</u><br>(NOT)</td><td>$Y=\overline{A}$</td><td></td><td></td><td></td><td>A Y<br>0 1<br>1 0</td></tr>
<tr><td><u>NAND-Gatter</u><br>(NICHT UND)<br>(NOT AND)</td><td>$Y=\overline{A\cdot B}$</td><td></td><td></td><td></td><td>A B Y<br>0 0 1<br>0 1 1<br>1 0 1<br>1 1 0</td></tr>
<tr><td><u>NOR-Gatter</u><br>(NICHT ODER)<br>(NOT OR)</td><td>$Y=\overline{A+B}$</td><td></td><td></td><td></td><td>A B Y<br>0 0 1<br>0 1 0<br>1 0 0<br>1 1 0</td></tr>
<tr><td><u>XOR-Gatter</u><br>(Exklusiv-ODER, Antivalenz)<br>(eXclusiveOR)</td><td>$Y=A\oplus B$</td><td></td><td></td><td></td><td>A B Y<br>0 0 0<br>0 1 1<br>1 0 1<br>1 1 0</td></tr>
</table>

**XNOR-Gatter (Exklusiv-Nicht-ODER, Äquivalenz)** (eXclusive Not OR)

$Y = \overline{A \oplus B}$




| A | B | Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Früher waren auf dem europäischen Kontinent die deutschen Symbole (rechte Spalte) verbreitet; im englischen Sprachraum waren und sind die amerikanischen Symbole (mittlere Spalte) üblich. Die IEC-Symbole sind international auf beschränkte Akzeptanz gestoßen und werden in der amerikanischen Literatur (fast) durchgängig ignoriert.

# JK-Flipflop

Ein Flip-Flop (bistabile Kippstufe oder bistabiler Multivibrator) hat zwei stabile Zustände am Ausgang Q; die Zustände heißen „gesetzt“ (set) oder „zurückgesetzt“ (reset). Ein 1-Bit-Speicher läßt sich somit als FlipFlop realisieren.

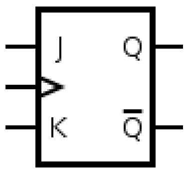
Ein JK-FlipFlop ist ein taktgesteuertes FlipFlop: die an den Eingängen J und K liegende Information wird mit einer Flanke (hier: der steigenden Flanke) des an C liegenden Taktsignals auf die Ausgänge Q und $\overline{Q}$ übernommen.

Mit dem Taktsignal (clock, C) und der Eingangsbelegung J = 1 und K = 0 wird am Ausgang Q eine 1 erzeugt und gespeichert, alternativ eine 0 bei J = 0 und K = 1.

Bei der Realisierung des JK-Flipflops als taktflankengesteuertes Flipflop kann der Eingang C für steigende Flanken (Wechsel von 0 auf 1) oder für fallende Flanken (Wechsel von 1 auf 0) ausgelegt sein.

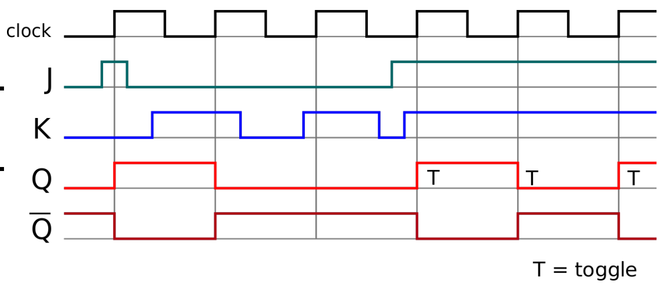
**Name und Schaltzeichen**

Flanken-gesteuertes JK-Flipflop




**Signal-Zeit-Diagramm**

Übernahme der Eingangsinformation durch steigende Flanke an C (clock)




**Funktionstabelle**

**bis zur … n-ten Taktflanke** / **nach der … n-ten Taktflanke**

| **J** | **K** | $\mathbf{Q_n}$ |
|---|---|---|
| 0 | 0 | $Q_{n-1}$ (unverändert) |
| 0 | 1 | 0 (zurückgesetzt) |
| 1 | 0 | 1 (gesetzt) |
| 1 | 1 | NOT $Q_{n-1}$ (gewechselt) |

*(Wikipedia)*

# Halbaddierer und Volladdierer

Die Ziffern einer im Dezimalsystem geschriebenen Zahl a ergeben sich als Aneinanderreihung der Koeffizienten aus der Dezimalzerlegung (Summe von Zehnerpotenzen) von a; entsprechend erhalten wir die Darstellung von a im Dualsystem als Aneinanderreihung der Koeffizienten aus der Dualzerlegung (Summe von Zweierpotenzen).

$87_{dezimal} = 8 \cdot 10^1 + 7 \cdot 10^0$

$87_{dezimal} = 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 1010111_{dual}$

Addition der Dualzahlen
$a = a_3 \cdot 2^3 + a_2 \cdot 2^2 + a_1 \cdot 2^1 + a_0 \cdot 2^0$ und $b = b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0$ :

| | | | | | |
|---|---|---|---|---|---|
| | | $a_3$ | $a_2$ | $a_1$ | $a_0$ |
| + | | $b_3$ | $b_2$ | $b_1$ | $b_0$ |
| | $s_4$ | $s_3$ | $s_2$ | $s_1$ | $s_0$ |

| | | | | | |
|---|---|---|---|---|---|
| | | 1 | 1 | 0 | 1 |
| + | | 1 | 0 | 1 | 1 |
| | 1 | 1 | 0 | 0 | 0 |

Den Übertrag („carry"), der sich aus der i-ten Stelle ergibt und der bei der Addition in der (i + 1)-ten Stelle zu berücksichtigen ist, bezeichnen wir mit $c_{i+1}$; $i \geq 0$.

Für die 0-te Stelle genügt ein Halbaddierer mit den Eingängen $a_0$ und $b_0$ und den Ergebnissen $s_0$ und $c_1$; die Addition in der i-ten Stelle, $i \geq 1$, erfordert einen Volladdierer mit den Eingängen $a_i$, $b_i$, $c_i$ und den Ergebnissen $s_i$ und $c_{i+1}$.

### Halbaddierer HA

Wahrheitstafel:

| $a_0$ | $b_0$ | $s_0$ | $c_1$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

Wir ermitteln für $\mathbf{s_0}$ und $\mathbf{c_1}$ jeweils die disjunktive Normalform („Disjunktion der Konjunktionen"):

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

$$c_1 = a_0 \cdot b_0$$

### Volladdierer VA

Wahrheitstafel:

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

Wir ermitteln für $\mathbf{s_i}$ und $\mathbf{c_{i+1}}$ jeweils die disjunktive Normalform („Disjunktion der Konjunktionen") und vereinfachen ggf. die booleschen Funktionsterme:

$$s_i = \bar{a}_i \cdot \bar{b}_i \cdot c_i + \bar{a}_i \cdot b_i \cdot \bar{c}_i + a_i \cdot \bar{b}_i \cdot \bar{c}_i + a_i \cdot b_i \cdot c_i$$

ohne Index i geschrieben:

$$s = \bar{a} \cdot \bar{b} \cdot c + \bar{a} \cdot b \cdot \bar{c} + a \cdot \bar{b} \cdot \bar{c} + a \cdot b \cdot c$$

$$s = (\bar{a} \cdot b + a \cdot \bar{b}) \cdot \bar{c} + \bar{a} \cdot \bar{b} \cdot c + a \cdot b \cdot c$$

$$s = (a \oplus b) \cdot \bar{c} + (\bar{a} \cdot \bar{b} + a \cdot b) \cdot c$$

$$s = (a \oplus b) \cdot \bar{c} + (0 + \bar{a} \cdot \bar{b} + a \cdot b + 0) \cdot c$$

$$s = (a \oplus b) \cdot \bar{c} + (\bar{a} \cdot a + \bar{a} \cdot \bar{b} + a \cdot b + b \cdot \bar{b}) \cdot c$$

$$s = (a \oplus b) \cdot \bar{c} + [(\bar{a} + b) \cdot (a + \bar{b})] \cdot c$$

$$s = (a \oplus b) \cdot \bar{c} + [(\bar{a} + \bar{\bar{b}}) \cdot (\bar{\bar{a}} + \bar{b})] \cdot c$$

$$s = (a \oplus b) \cdot \bar{c} + [(\overline{a \cdot \bar{b}}) \cdot (\overline{\bar{a} \cdot b})] \cdot c$$

$$s = (a \oplus b) \cdot \bar{c} + [\overline{a \cdot \bar{b} + \bar{a} \cdot b}] \cdot c$$

$$s = (a \oplus b) \cdot \bar{c} + (\overline{a \oplus b}) \cdot c$$

$$s = (a \oplus b) \oplus c$$

mit Index i erhält man:

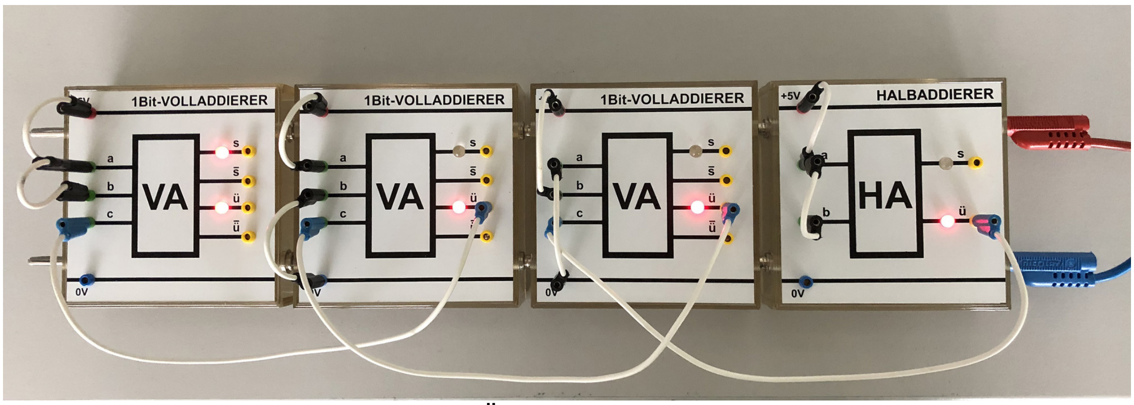
$$s_i = (a_i \oplus b_i) \oplus c_i$$

$$c_{i+1} = \bar{a}_i \cdot b_i \cdot c_i + a_i \cdot \bar{b}_i \cdot c_i + a_i \cdot b_i \cdot \bar{c}_i + a_i \cdot b_i \cdot c_i$$

$$c_{i+1} = (\bar{a}_i \cdot b_i + a_i \cdot \bar{b}_i) \cdot c_i + a_i \cdot b_i \cdot (\bar{c}_i + c_i)$$

$$c_{i+1} = (\bar{a}_i \cdot b_i + a_i \cdot \bar{b}_i) \cdot c_i + a_i \cdot b_i \cdot 1$$

$$c_{i+1} = (a_i \oplus b_i) \cdot c_i + a_i \cdot b_i$$

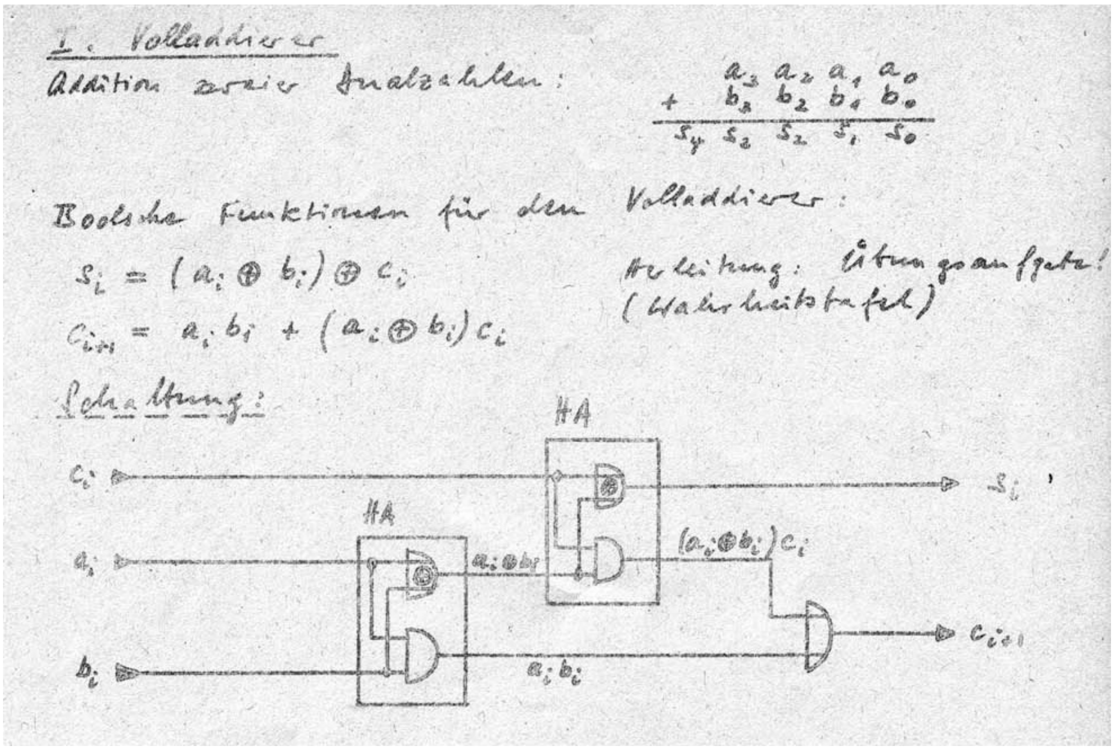


4-Bit-Paralleladdierer mit seriellem Übertrag

**Merke:**
Bei der Addition zweier Dualzahlen benötigt man
für das LSB (least significant bit) einen Halbaddierer (Eingänge: $a_0$, $b_0$; Ausgänge: $s_0$, $c_1$),
für die höherwertigen Bits jeweils einen Volladdierer (Eingänge: $a_i$, $b_i$, $c_i$; Ausgänge: $s_i$, $c_{i+1}$).

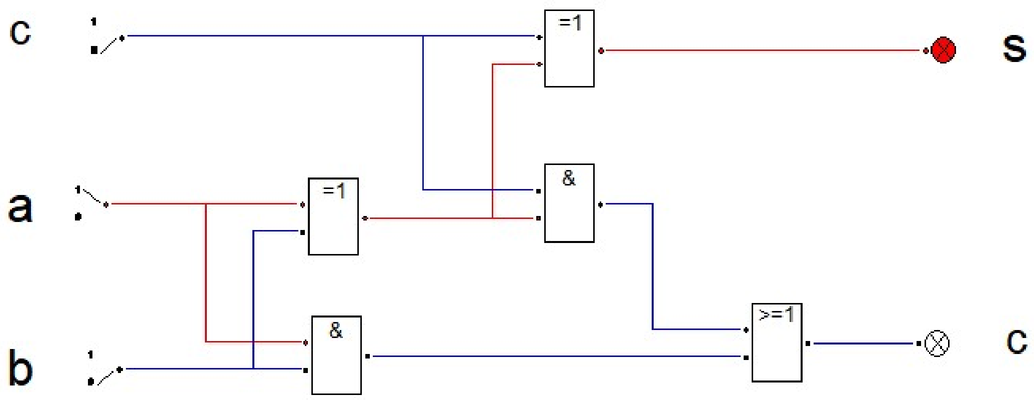
## Schaltungen

Halbaddierer (HA) und Volladdierer (VA)




**Merke:**
Die Schaltung des Volladdierers (VA) besteht aus zwei Halbaddierern (HA) und einem oder-Gatter.

Volladdierer

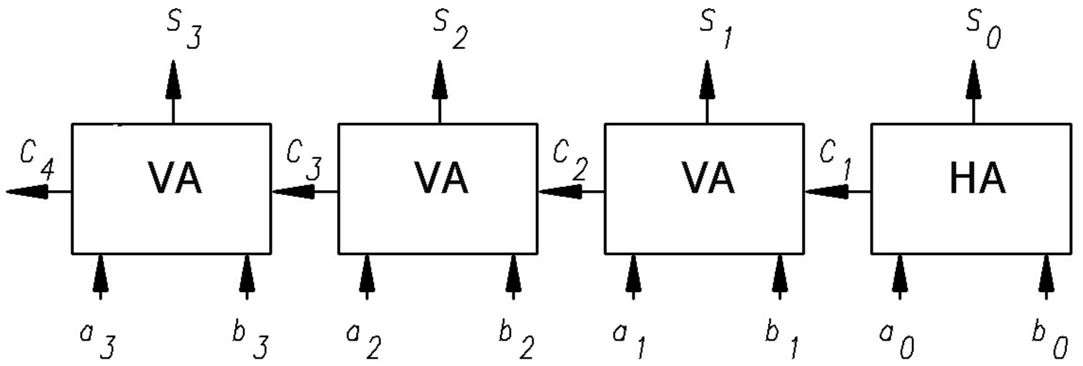


# Addier-Schaltungen für Dualzahlen (hier: 4-Bit-Addierer)

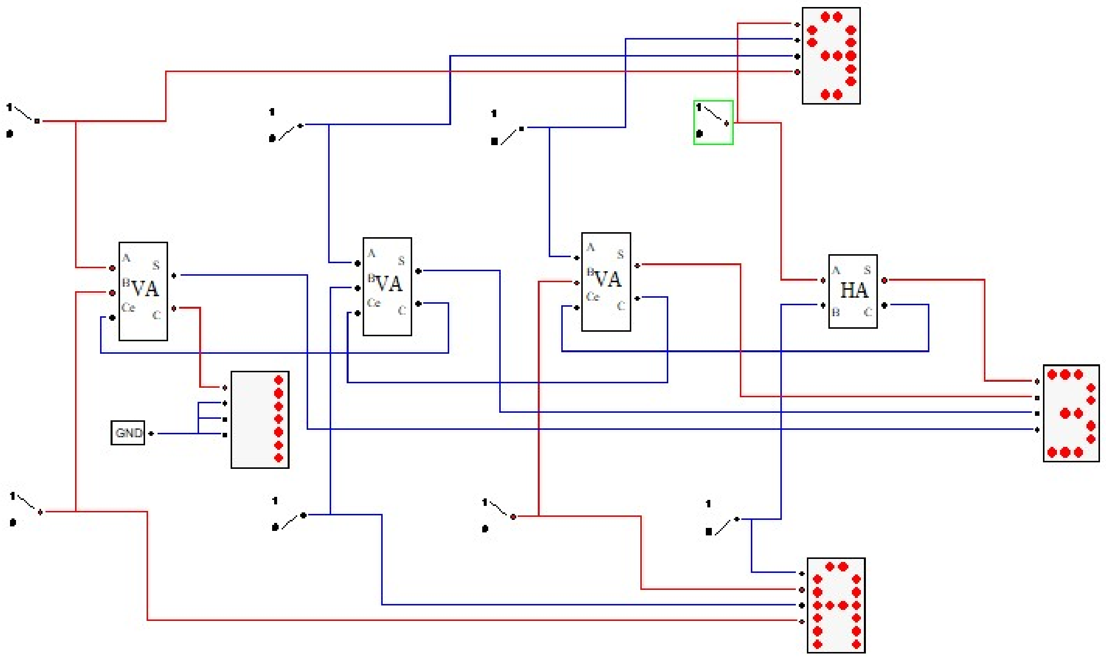
$$\begin{array}{ccccc} & a_3 & a_2 & a_1 & a_0 \\ + & b_3 & b_2 & b_1 & b_0 \\ \hline s_4 & s_3 & s_2 & s_1 & s_0 \end{array}$$

## 1. Paralleladdierer mit seriellem Übertrag

Für das Least Significant Bit (LSB) genügt ein Halbaddierer (HA); die höherwertigen Bits erfordern jeweils einen Volladdierer, da hier der Übertrag aus der vorherigen Stelle zu berücksichtigen ist.




Beachte:
Das Most Significant Bit $s_4$ des Ergebnisses (hier: der aus den Ziffern $s_4, \ldots, s_0$ bestehenden Summe) erhalten wir als den Übertrag (carry) $c_4$, der auch als Überlauf bezeichnet wird.

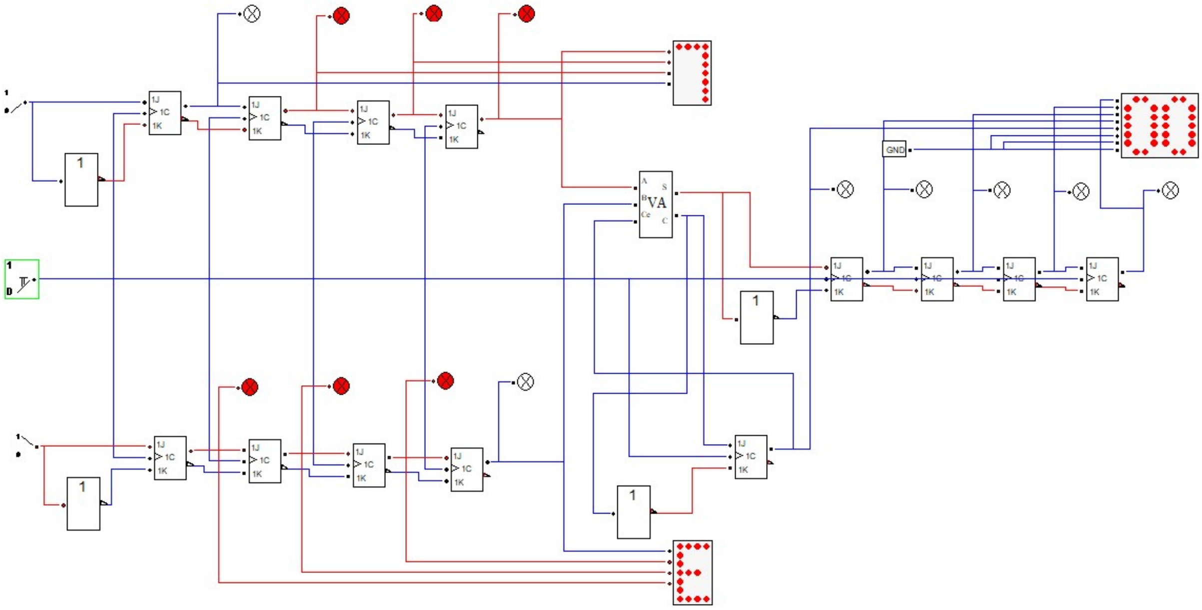



```
Dezimal:   09   Hexadezimal:   09    Dual: 0000 1001
         + 10                + 0A        + 0000 1010
           19                  13          0001 0011
```

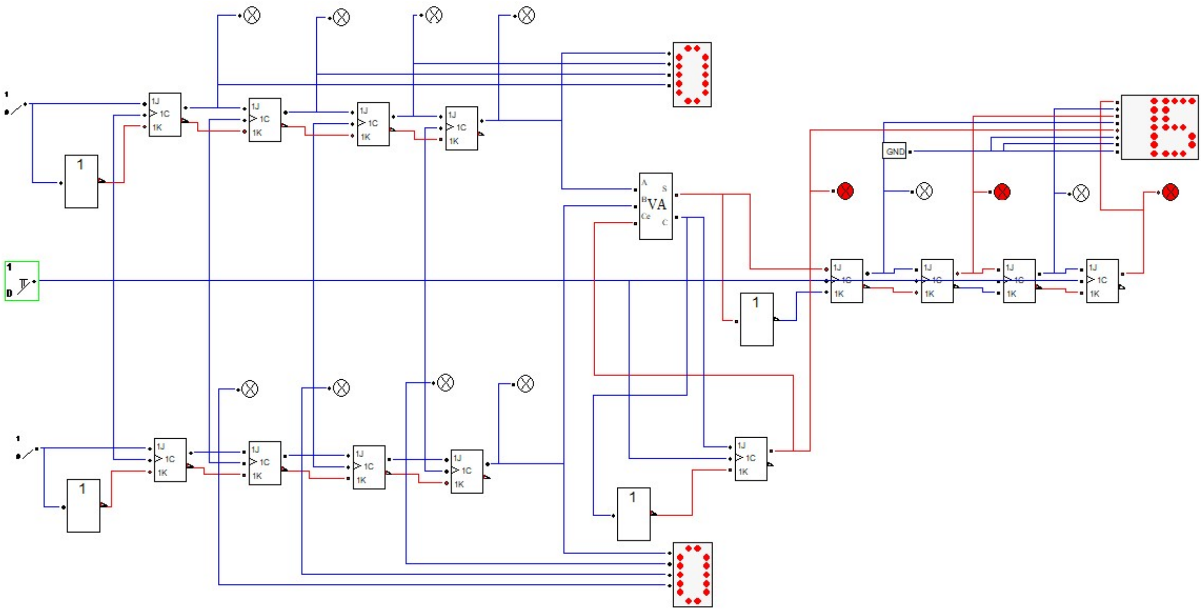
## 2. Serieller 1-Bit-Addierer für 4-stellige Dualzahlen

Die Operanden (hier: die Summanden **a** und **b**) werden jeweils in einem 4-Bit-Schieberegister abgelegt; nach 4 Taktimpulsen finden wir das 5-Bit-breite Ergebnis (hier: die Summe **s**) in einem weiteren 4-Bit-Schieberegister in Verbindung mit einem Flip-Flop für das MSB.

Da der Übertrag aus der vorherigen Stelle für die Addition in der jeweils aktuellen Stelle zu berücksichtigen ist, wird er in einem Flip-Flop zwischengespeichert. Dieses Flip-Flop liefert auch das Most Significant Bit (MSB) des Ergebnisses.



Nach 4 Taktimpulsen (hier: Triggerung der Flip-Flops auf der steigenden Taktflanke) sind die Schieberegister für die Operanden geleert, das Schieberegister für das Ergebnis enthält zusammen mit dem im Flip-Flop gespeicherten MSB das Ergebnis:



| | | | | | |
|---|---|---|---|---|---|
| Dezimal: | 07 | Hexadezimal: | 07 | Dual: | 0000 0111 |
| | + 14 | | + 0E | | + 0000 1110 |
| | 21 | | 15 | | 0001 0101 |

Die wesentlichen Komponenten einer **CENTRAL PROCESSING UNIT** (**CPU**) bestehen aus der **CONTROL UNIT** (**CU**) und der **ARITHMETIC LOGIC UNIT** (**ALU**).

Die **ALU** berechnet arithmetische und logische Funktionen, die **CU** decodiert die im Arbeitsspeicher abgelegten Befehle und führt sie aus.

In der Minimalkonfiguration beherrscht die **ALU** die arithmetische Funktion **„Addition"** sowie die logischen Operationen **„Negation"** (NOT) und **„Konjunktion"** (AND). Zu Lasten der Rechenzeit lassen sich die übrigen arithmetischen und logischen Funktionen auf die genannten, minimal verfügbaren Operationen zurückführen.

## 1. Subtraktion

Die duale Subtraktion

$$\begin{array}{rcccc} & a_3 & a_2 & a_1 & a_0 \\ - & b_3 & b_2 & b_1 & b_0 \\ \hline & d_3 & d_2 & d_1 & d_0 \end{array}$$

läßt sich auf eine duale Addition nach folgendem Verfahren zurückführen:

- Bilde das Einerkomplement des Subtrahenden $b_3\ b_2\ b_1\ b_0$ , indem man alle Ziffern negiert (invertiert; aus 0 wird 1 und aus 1 wird 0).
- Addiere das Einerkomplement und die Zahl 1 zum Minuenden.
- Das Ergebnis ist die gesuchte Differenz; dabei bleibt der Überlauf unberücksichtigt.

a) Verdeutliche das genannte Verfahren anhand einiger selbst gewählter Beispiele (ein Beweis des Verfahrens ist nicht erforderlich.).

b) Ergänze die Schaltung „4-bit-Paralleladdierer.dsim" so, daß man nach entsprechender Umschaltung wahlweise eine duale Addition oder eine duale Subtraktion durchführen kann.
Hinweise:
   - Ersetze den HA für das least significant bit (LSB) durch einen VA, um erforderlichenfalls eine „1" als Summand einspeisen zu können (wie?).
   - Die Invertierung der Ziffern des Subtrahenden gelingt z. B. durch den geeigneten Einsatz von XOR-Gattern.

## 2. Weitere Rechenoperationen

Gegeben sind die (im einfachsten Fall positiven ganzzahligen) Operanden a und b. Um zu verdeutlichen, wie man die „höheren" Rechenoperationen mittels geeigneter Iteration auf die Grundoperationen „Addition" und „Subtraktion" zurückführen kann, schreibe und teste ein Python-Programm, welches die Operationen „Multiplikation" (a*b), „Division" (a/b, ganzzahlige Division) und „Potenzierung" (a**b) realisiert.

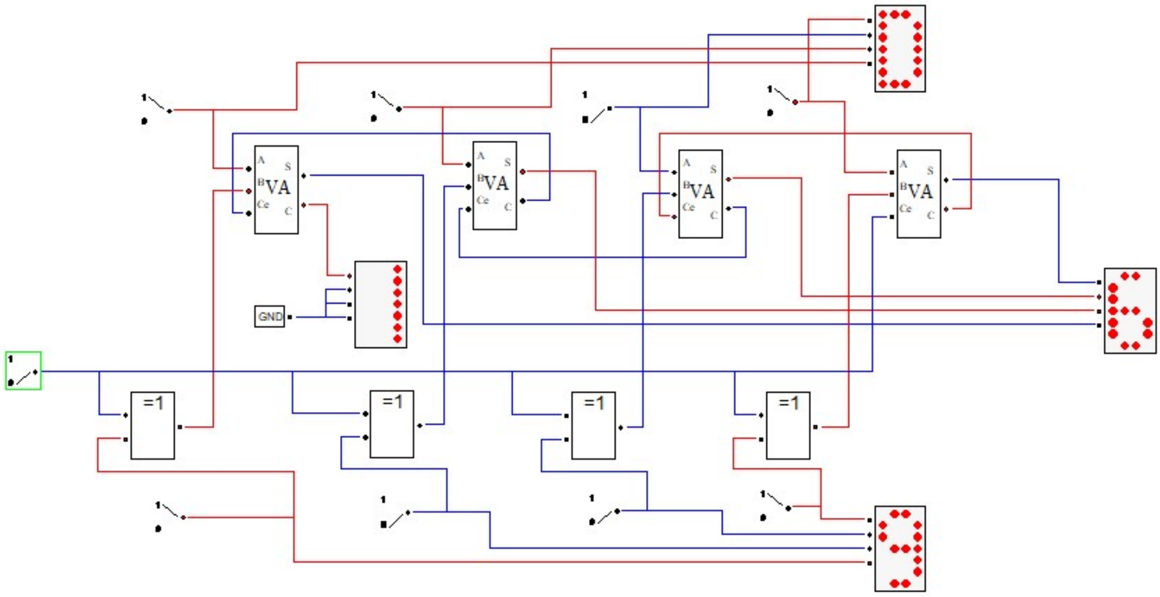
## 3. Logische Operationen

Zeige examplarisch, daß sich die logischen Verknüpfungen

a) $a + b$
b) $a \oplus b$
c) $a \cdot (b + \overline{c})$

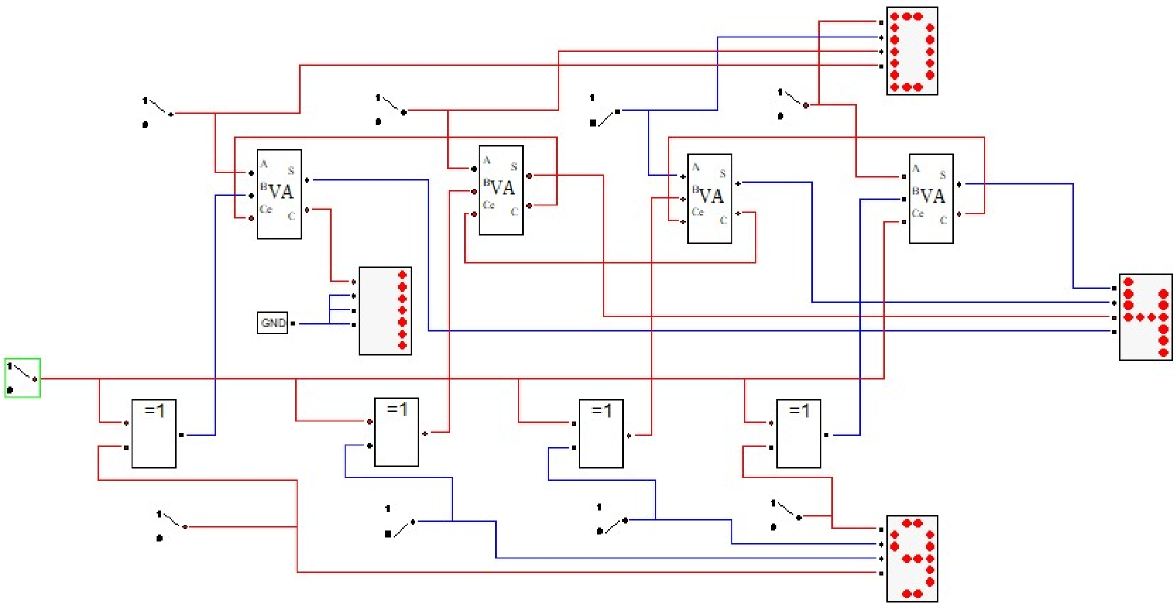
auf die Operationen NOT und AND zurückführen lassen.

**Paralleladdierer mit seriellem Übertrag (4-Bit-Addierer)**




| Dezimal: | | Hexadezimal: | | Dual: | |
|---|---|---|---|---|---|
| | 13 | | 0D | | 0000 1101 |
| | + 09 | | + 09 | | + 0000 1001 |
| | 22 | | 16 | | 0001 0110 |

**Parallelsubtrahierer mit seriellem Übertrag (4-Bit-Subtrahierer)**




| Dezimal: | | Hexadezimal: | | Dual: | |
|---|---|---|---|---|---|
| | 13 | | 0D | | 0000 1101 |
| | − 09 | | − 09 | | − 0000 1001 |
| | 04 | | 04 | | 0000 0100 |

Das Carry-Bit $c_4$ (Überlauf) bleibt beim Ergebnis unberücksichtigt.

```
# Grundrechenarten
# Die "höheren" Rechenoperationen Multiplizieren, Potenzieren, Dividieren
# werden durch geeignete Iteration auf die Grundoperationen
# Addieren und Subtrahieren zurückgeführt.

def summe(a,b):
      return a + b

def differenz(a,b):
      return a - b

def produkt(a,b):
      ergebnis = 0
      i = 0
      while i <= b - 1:
          ergebnis = summe(ergebnis,a)
          i +=1
      return ergebnis

def potenz(a,b):
      if b == 0: return 1
      else:
        ergebnis = a
        i = 0
        while i <= b - 2:
          ergebnis = produkt(ergebnis,a)
          i = i + 1
        return ergebnis

def quotient(a,b):
      rest = a
      ergebnis = 0
      while rest >= b:
          rest = differenz(rest,b)
          ergebnis += 1
      return ergebnis

print ('Operanden:')
x = int(input('x = '))
y = int(input('y = '))
print()

print('Operation:')
print('  Addition < + >')
print('  Subtraktion < - >')
print('  Multiplikation < * >')
print('  Division < / >')
print('  Potenz < ** > ')
op = input()
print()

if op == '+':    print (x, ' + ', y, '=',summe(x,y))
elif op == '-':  print (x, ' - ', y, '=',differenz(x,y))
elif op == '*':  print (x, ' * ', y, '=',produkt(x,y))
elif op == '/':  print (x, ' // ', y, '=',quotient(x,y))
elif op == '**': print (x, ' ^ ', y, '=',potenz(x,y))
else: print('falsche Eingabe')
```