# Einführung in die Digitaltechnik

## Theorie (L)

Version vom 26. Februar 2022

# 1 Boolesche Algebra

### Motivation

Wer den Aufbau und die Funktionsweise digitaler Computer verstehen will, muss sich mit elektrischen Schaltkreisen befassen.

Da in einer solchen Schaltung meist nur die zwei Zustände

- ein Strom fliesst (eine Lampe brennt)
- kein Strom fliesst (eine Lampe brennt nicht)

von Bedeutung sind, muss ein Weg gefunden werden, wie mit diesen beiden Zuständen „gerechnet“ werden kann.

### Etwas Geschichte

Der englische Mathematiker George Boole entwickelte in der Mitte des 19. Jahrhunderts ein System von Regeln, die es erlauben, sinnvoll mit den Werten 0 und 1 zu rechnen.

Damit schuf Boole die formale Grundlage für die ersten Digitalrechner, die in der Mitte des 20. Jahrhunderts erfunden wurden.

### Der NOT-Operator

$\overline{x} := 1 - x$

„:=“ bedeutet, dass der linke Ausdruck durch den rechten definiert wird.

| $x$ | $\overline{x}$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

Anstelle von $\overline{x}$ schreibt man auch

- $\text{NOT}\, x$
- $\neg x$

### Der AND-Operator

$x \wedge y := x \cdot y$

| $x$ | $y$ | $x \wedge y$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Anstelle von $x \wedge y$ schreibt man auch

- $x \,\text{AND}\, y$
- $xy$

$x \wedge y$ ist genau dann 1, wenn beide Operanden 1 sind.

### Der OR-Operator

$x \vee y := x + y - x \cdot y$

| $x$ | $y$ | $x \vee y$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Anstelle von $x \vee y$ schreibt man auch

- $x \,\text{OR}\, y$
- $x + y$

$x \vee y$ ist genau dann 1, wenn mindestens ein Operand 1 ist.

### Boolesche Algebra

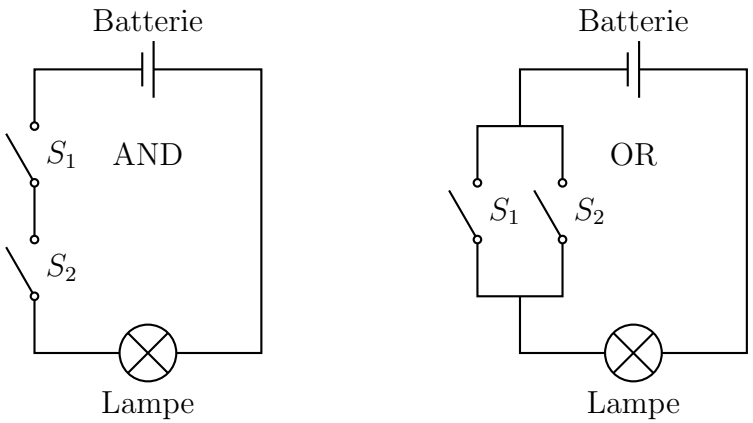
Die Menge $B = \{0, 1\}$ bildet zusammen mit den drei Operatoren

- NOT (Negation)
- AND (Konjunktion)
- OR (Disjunktion)

eine algebraische Struktur, die *boolesche Algebra* genannt wird.

### Boolesche Schaltungen

Die Konjunktion und die Disjunktion lassen sich durch eine Serie- bzw. Parallelschaltung veranschaulichen.




### Boolesche Terme

Ein *boolescher Ausdruck* besteht aus Variablen $a$, $b$, $c$, ..., den oben beschriebenen Operatoren NOT, AND, OR und Klammern.

*Beispiel:* $a \vee (\overline{b} \wedge c)$

Setzt man für jede Variable eines booleschen Terms einen Wert aus $B = \{0, 1\}$ ein, so erhält man eine *Belegung* der Variablen.

*Beispiel:* $a = 0$, $b = 1$, $c = 1$

Setzt man eine Belegung in einen booleschen Term ein und wertet ihn aus, erhält man den *Wert* des Terms.

*Beispiel:* $0 \vee (\overline{1} \wedge 1) = 0 \vee (0 \wedge 1) = 0 \vee 0 = 0$

Zwei boolesche Terme sind *äquivalent* (gleichwertig), wenn sie für jede mögliche Belegung den gleichen Wahrheitswert haben.

### Doppelte Negation

| $x$ | $\overline{x}$ | $\overline{\overline{x}}$ |
|---|---|---|
| 0 | 1 | 0 |
| 1 | 0 | 1 |

$\Rightarrow \quad \overline{\overline{x}} = x$

**Kommutativgesetze**

| $x$ | $y$ | $x \wedge y$ | $y \wedge x$ |
|---|---|---|---|
| 0 | 0 | $0 \wedge 0 = 0$ | $0 \wedge 0 = 0$ |
| 0 | 1 | $0 \wedge 1 = 0$ | $1 \wedge 0 = 0$ |
| 1 | 0 | $1 \wedge 0 = 0$ | $0 \wedge 1 = 0$ |
| 1 | 1 | $1 \wedge 1 = 1$ | $1 \wedge 1 = 1$ |

$\Rightarrow \quad x \wedge y = y \wedge x$

Analog beweist man: $x \vee y = y \vee x$

**Assoziativgesetze**

| $x$ | $y$ | $z$ | $(x \wedge y) \wedge z$ | $x \wedge (y \wedge z)$ |
|---|---|---|---|---|
| 0 | 0 | 0 | $0 \wedge 0 = 0$ | $0 \wedge 0 = 0$ |
| 0 | 0 | 1 | $0 \wedge 1 = 0$ | $0 \wedge 0 = 0$ |
| 0 | 1 | 0 | $0 \wedge 0 = 0$ | $0 \wedge 0 = 0$ |
| 0 | 1 | 1 | $0 \wedge 1 = 0$ | $0 \wedge 1 = 0$ |
| 1 | 0 | 0 | $0 \wedge 0 = 0$ | $1 \wedge 0 = 0$ |
| 1 | 0 | 1 | $0 \wedge 1 = 0$ | $1 \wedge 0 = 0$ |
| 1 | 1 | 0 | $1 \wedge 0 = 0$ | $1 \wedge 0 = 0$ |
| 1 | 1 | 1 | $1 \wedge 1 = 1$ | $1 \wedge 1 = 1$ |

$\Rightarrow \quad (x \wedge y) \wedge z = x \wedge (y \wedge z)$

Analog beweist man: $(x \vee y) \vee z = x \vee (y \vee z)$

**Distributivgesetze**

| $x$ | $y$ | $z$ | $x \wedge (y \vee z)$ | $(x \wedge y) \vee (x \wedge z)$ |
|---|---|---|---|---|
| 0 | 0 | 0 | $0 \wedge 0 = 0$ | $0 \vee 0 = 0$ |
| 0 | 0 | 1 | $0 \wedge 1 = 0$ | $0 \vee 0 = 0$ |
| 0 | 1 | 0 | $0 \wedge 1 = 0$ | $0 \vee 0 = 0$ |
| 0 | 1 | 1 | $0 \wedge 1 = 0$ | $0 \vee 0 = 0$ |
| 1 | 0 | 0 | $1 \wedge 0 = 0$ | $0 \vee 0 = 0$ |
| 1 | 0 | 1 | $1 \wedge 1 = 1$ | $0 \vee 1 = 1$ |
| 1 | 1 | 0 | $1 \wedge 1 = 1$ | $1 \vee 0 = 1$ |
| 1 | 1 | 1 | $1 \wedge 1 = 1$ | $1 \vee 1 = 1$ |

$\Rightarrow \quad x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$

Analog beweist man: $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$

### Gesetze von De Morgan

| $x$ | $y$ | $\overline{x \wedge y}$ | $\overline{y} \vee \overline{x}$ |
|---|---|---|---|
| 0 | 0 | $\overline{0} = 1$ | $1 \vee 1 = 1$ |
| 0 | 1 | $\overline{0} = 1$ | $1 \vee 0 = 1$ |
| 1 | 0 | $\overline{0} = 1$ | $0 \vee 1 = 1$ |
| 1 | 1 | $\overline{1} = 0$ | $0 \vee 0 = 0$ |

$\Rightarrow \quad \overline{x \wedge y} = \overline{x} \vee \overline{y}$

Analog beweist man: $\overline{x \wedge y} = \overline{x} \vee \overline{y}$

### Idempotenzgesetze

| $x$ | $x \wedge x$ |
|---|---|
| 0 | $0 \wedge 0 = 0$ |
| 1 | $1 \wedge 1 = 1$ |

$\Rightarrow x \wedge x = x$

| $x$ | $x \vee x$ |
|---|---|
| 0 | $0 \vee 0 = 0$ |
| 1 | $1 \vee 1 = 1$ |

$\Rightarrow x \vee x = x$

### Identitätsgesetze

| $x$ | $x \wedge 1$ |
|---|---|
| 0 | $0 \wedge 1 = 0$ |
| 1 | $1 \wedge 1 = 1$ |

$\Rightarrow x \wedge 1 = x$

| $x$ | $x \vee 0$ |
|---|---|
| 0 | $0 \vee 0 = 0$ |
| 1 | $1 \vee 0 = 1$ |

$\Rightarrow x \vee 0 = x$

### Komplementärgesetze

| $x$ | $x \wedge \overline{x}$ |
|---|---|
| 0 | $0 \wedge 1 = 0$ |
| 1 | $1 \wedge 0 = 0$ |

$\Rightarrow x \wedge \overline{x} = 0$

| $x$ | $x \vee \overline{x}$ |
|---|---|
| 0 | $0 \vee 1 = 1$ |
| 1 | $1 \vee 0 = 1$ |

$\Rightarrow x \vee \overline{x} = 1$

### Logische Funktionen

Wie wir oben gesehen haben, ordnet ein boolescher Term jeder Belegung seiner Variablen einen Wahrheitswert (0 oder 1) zu. Daher definieren boolsche Terme *boolesche Funktionen*.

Weil verschiedene boolesche Terme dieselbe boolesche Funktion definieren können, ist es praktisch, wenn für die Darstellung von boolschen Termen einheitliche Formen – sogenannte *Normalformen* – existieren.

### Normalformen

Um die konjunktive und disjunktive Normalform zu bilden, muss zuerst die Wahrheitstabelle mit allen möglichen Belegungen aufgestellt werden. Hier am Beispiel von $f(a, b, c) = a \vee (\overline{b} \wedge c)$.

| $a$ | $b$ | $c$ | $a \vee (\overline{b} \wedge c)$ |
|---|---|---|---|
| 0 | 0 | 0 | $0 \vee (\overline{0} \wedge 0) = 0$ |
| 0 | 0 | 1 | $0 \vee (\overline{0} \wedge 1) = 1$ |
| 0 | 1 | 0 | $0 \vee (\overline{1} \wedge 0) = 0$ |
| 0 | 1 | 1 | $0 \vee (\overline{1} \wedge 1) = 0$ |
| 1 | 0 | 0 | $1 \vee (\overline{0} \wedge 0) = 1$ |
| 1 | 0 | 1 | $1 \vee (\overline{0} \wedge 1) = 1$ |
| 1 | 1 | 0 | $1 \vee (\overline{1} \wedge 0) = 1$ |
| 1 | 1 | 1 | $1 \vee (\overline{1} \wedge 1) = 1$ |

### Die konjunktive Normalform

Die konjunktive Normalform besteht aus einer Konjunktion ($\wedge$) von Disjunktionen ($\vee$) verneinter oder unverneinter Variablen.

Da eine Konjunktion genau dann den Wert Null hat, wenn mindestens ein Operand null ist, müssen wir dafür sorgen, dass mindestens ein Operand (hier die Disjunktionen) null wird, wenn die Belegung den Wert

Da eine Konjunktion ($\wedge$) genau dann den Wert 0 hat, wenn mindestens ein Operand den

Wert Null hat, bilden wir eine Konjunktion aller Zeilen der Wahrheitstabelle, die den Wert Null haben und negieren die Variablen mit dem Wert 1.

$f(a, b, c) = (a \wedge b \wedge c) \vee (a \wedge \overline{b} \wedge c) \vee (a \wedge \overline{b} \wedge \overline{c})$

### Disjunktive Normalform

Da eine Disjunktion ($\vee$) genau dann den Wert 1 hat, wenn mindestens ein Operand den Wert Eins hat, bilden wir eine Disjunktion aller Zeilen der Wahrheitstabelle, die den Wert Eins haben und negieren die Variablen mit dem Wert 0.

$f(a, b, c) = (\overline{a} \vee \overline{b} \vee c) \wedge (a \wedge \overline{b} \wedge \overline{c}) \wedge (a \wedge \overline{b} \wedge c) \wedge (a \wedge b \wedge \overline{c}) \wedge (a \wedge b \wedge c)$

# 2 Transistoren, Gatter und Chips

### Transistoren

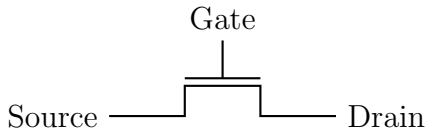
Ein *Transistor* (*Trans*fer Res*istor*) ist ein elektronisches Halbleiter-Bauteil, mit dem sich ein elektrischer Strom durch einen anderen Strom ein- und ausschalten lässt.

Die Grundlage für einen Transistor bildet meist ein Halbleiter. Ein Halbleiter ist ein festes Material, das unter bestimmten Bedingungen leitend (wie Kupfer) oder auch nicht leitend (wie Kunststoff) sein kann. Durch ein spezielles Verfahren lässt sich ein Halbleitermaterial so präparieren, dass seine Leitfähigkeit steuerbar wird.

### MOS-Transistoren

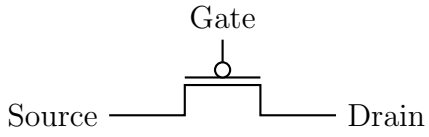
Ein *Metall-Oxid-Halbleiter-Transistor* (engl. *MOS Transistor*) besteht aus drei elektrischen Anschlüssen. Zwei dieser Anschlüsse (*Source* und *Drain*) sind durch ein speziell präpariertes Halbleitermaterial getrennt, das mit Hilfe einer Spannungsquelle am dritten Anschluss (*Gate*) leitend oder nicht leitend gemacht werden kann. Man unterscheidet zwei Bauformen:

### Der pMOS-Transistor




| | |
|---|---|
| Ladung am Gate: | Source und Drain haben Kontakt |
| keine Ladung am Gate: | Source und Drain sind getrennt |

### Der nMOS-Transistor



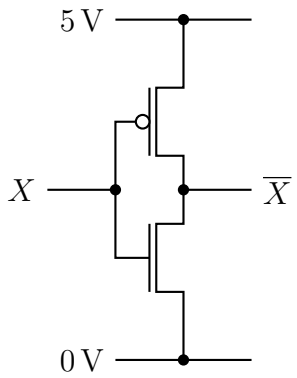

| | |
|---|---|
| Ladung am Gate: | Source und Drain sind getrennt |
| keine Ladung am Gate: | Source und Drain haben Kontakt |

### Gatter

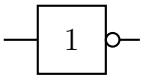
Wir werden jetzt die vom Programmierunterricht bekannten logischen Funktionen `NOT`, `AND`, `OR` (und noch ein paar mehr) durch Schaltungen realisieren, die aus Transistoren aufgebaut sind. Eine solche Schaltung wird auch *Gatter* (engl. *gate*) genannt.

### Das NOT-Gatter

Koppelt man einen nMOS- mit einem pMOS-Transistor, erhält man eine Schaltung, die den Eingangswert $x$ verneint.

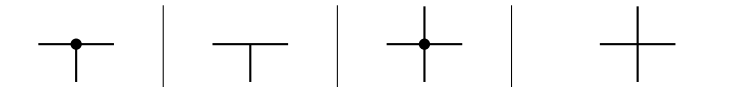



DIN-Symbol




In den Übungen findest du eine Aufgabe, in der du zeigen sollst, dass diese Schaltung auch tatsächlich das Gewünschte leistet.

### Zeichenerklärung




### Das logische NAND

Das logische NAND ist eine AND mit verneintem Ergebnis.

| $X$ | $Y$ | $X \wedge Y$ | $\overline{X \wedge Y}$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

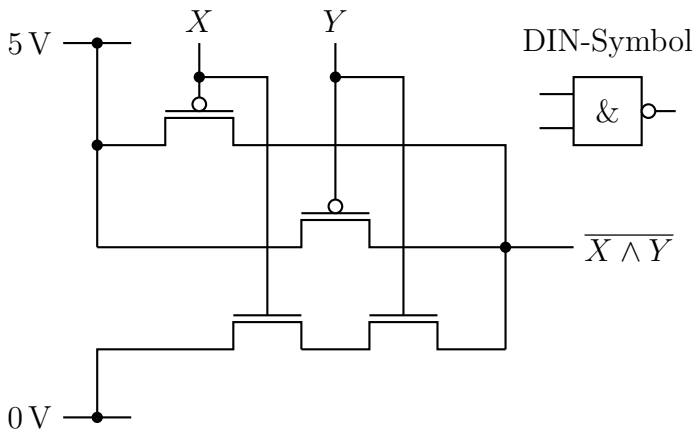
### Das logische NOR

Das logische NOR ist ein OR mit verneintem Ergebnis.

| $X$ | $Y$ | $X \vee Y$ | $\overline{X \vee Y}$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 |

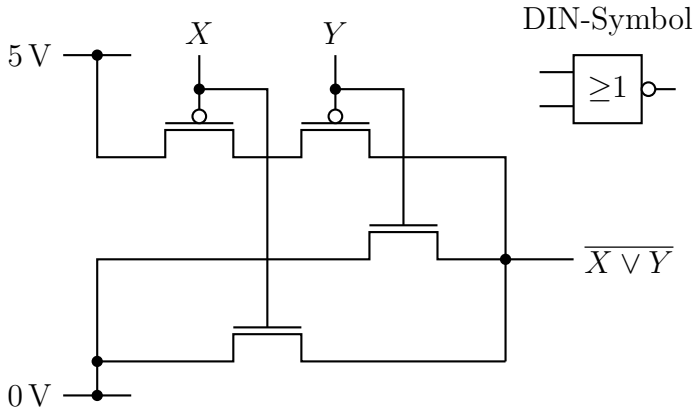
Wir führen NAND und NOR ein, da es die einfachsten Schaltungen sind, die sich aus jeweils zwei nMOS- und pMOS-Transistoren aufbauen lassen.

**Das NAND-Gatter**




In den Übungen findest du eine Aufgabe, in der du zeigen sollst, dass diese Schaltung auch tatsächlich das Gewünschte leistet.

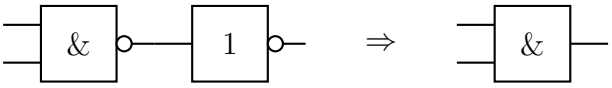
**Das NOR-Gatter**




In den Übungen findest du eine Aufgabe, in der du zeigen sollst, dass diese Schaltung auch tatsächlich das Gewünschte leistet.

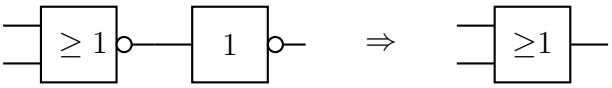
**Das AND-Gatter**

Das AND-Gatter lässt sich nun leicht aus den bereits bestehenden Gattern aufbauen, indem man die Verneinung eines NAND-Gatters durch ein nachgeschaltetes NOT-Gatter wieder aufhebt.



### Das OR-Gatter

Das OR-Gatter lässt sich nun leicht aus den bereits bestehenden Gattern aufbauen, indem man die Verneinung eines NOR-Gatters durch ein nachgeschaltetes NOT-Gatter wieder aufhebt.




### Die XOR-Funktion

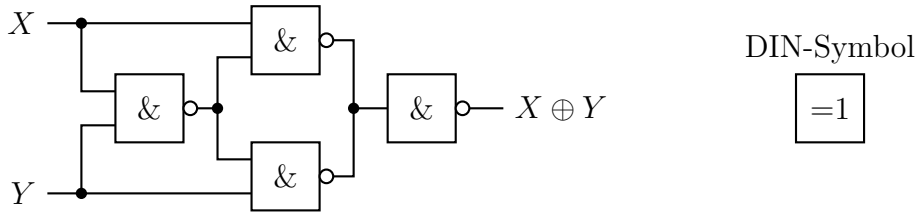
Das *auschliessende ODER* (*exclusive or*) ist die logische Funktion, die zwei Wahrheitswerten $X$ und $Y$ genau dann den Wert 1 zuordnet, wenn $X$ und $Y$ verschieden sind. Dies entpricht dem umgangssprachlichen „entweder oder“.

| $X$ | $Y$ | $X \oplus Y$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Die XOR-Funktion wird unter anderem für die Subtraktion von Zahlen oder zum Verschlüsseln von Daten verwendet.

### Das XOR-Gatter

Die XOR-Schaltung lässt sich aus vier NAND-Gattern aufbauen.




### Chips

Wenn eine elektronische Schaltung aus vielen Bauteilen (Transistoren, Wiederstände, Kondensatoren, . . . ) vollständig auf einer einzigen Halbleiterscheibe aufgebaut wird, spricht man von einer *integrierten Schaltung* (*integrated circuit, IC*) oder einem *Chip*.

Auf modernen CPU-Chips befinden sich heute (2021) mehrere Hundert Millionen Transistoren auf einer Fläche von einem Quadratzentimeter.

# 3 Addition von Binärzahlen

### Halbaddierer

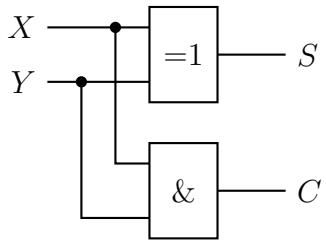
Ein *Halbaddierer* ist eine Schaltung, die zwei einstellige Binärzahlen ohne Berücksichtigung eines vorgängigen Übertrags addiert.

| $X$ | $Y$ | $X+Y$ | $C$ (Carry) |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

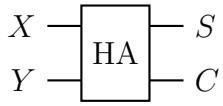
Carry steht für „Übertrag“. Beachte, dass ein Halbaddierer zwar einen Übertrag ausgibt aber keinen vorgängigen Übertrag in der Summe berücksichtigt.

### Ein Schaltnetz für einen Halbaddierer

Für einen Halbaddierer genügen ein XOR- und ein AND-Gatter.




Für einen 1-Bit-Halbaddierer (*half adder*) verwenden wir das folgende Symbol:



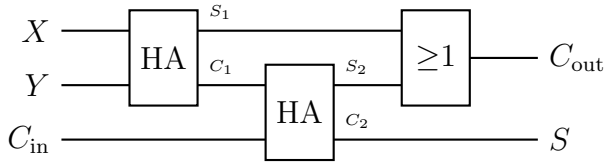

### Der Volladdierer

Für die Addition mehrstelliger Binärzahlen müssen wir einen allfälligen Übertrag aus der vorangehenden Stelle berücksichtigen.

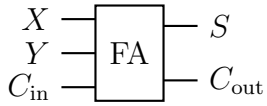
| $C_{\text{in}}$ | $X$ | $Y$ | $X+Y$ | $C_{\text{out}}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

### Ein Schaltnetz für einen Volladdierer

Für einen Volladdierer genügen zwei Halbaddierer und ein OR-Gatter.

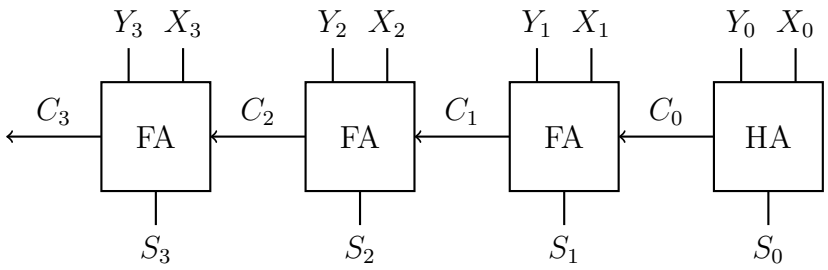



Für einen 1-Bit-Volladdierer (*full adder*) verwenden wir das folgende Symbol:




### Ein Paralleladdierer

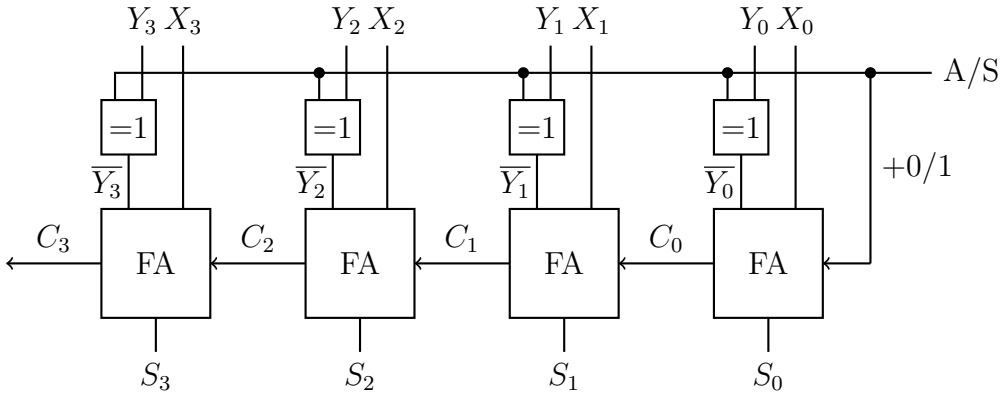
Sollen zwei vierstellige Binärzahlen addiert werden, benötigen wir einen Halb- und drei Volladdierer.




### Eine Schaltung zum Addieren und Subtrahieren

Digitalrechner subtrahieren eine Zahl $Y$, indem sie die Gegenzahl $-Y$ addieren, wobei die Gegenzahl im Zweierkomplement dargestellt wird: $-Y = \overline{Y} + 1$.

Wir signalisieren mit einer Steuerleitung (A/S), ob die Operanden addiert (Wert 0) oder subtrahiert (Wert 1) werden sollen.



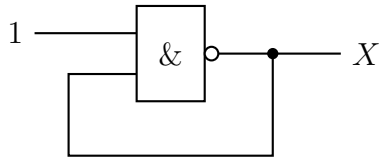
# 4 Flipflops

Im ersten Kapitel wurde erklärt, wie die elementaren logischen Funktionen durch Schaltungen (Gatter) realisiert werden können.

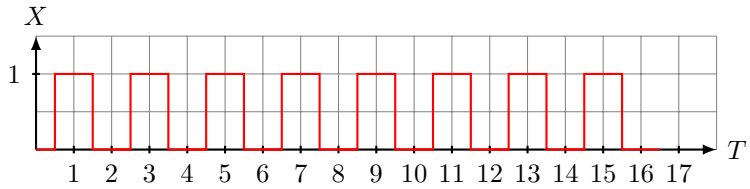
Im zweiten Kapitel haben wir aus diesen Bausteinen komplexere Schaltungen gebildet, mit denen sich binär addieren und subtrahieren (und damit auch multiplizieren und dividieren) lässt.

In diesem Kapitel konstruieren wir Schaltungen, die es uns erlauben, Daten zu speichern, solange die Schaltkreise mit Strom versorgt sind. Dazu führen wir das Konzept der *Rückkopplung* ein. Bei einer Rückkopplung wird der Ausgang eines Gatters mit dem Eingang eines Gatters verbunden.

### Eine Flimmerschaltung



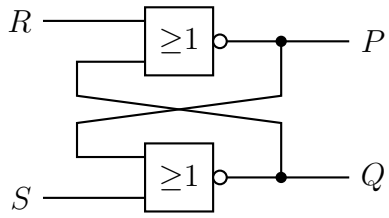

| Zeitschritt $T$ | $X$ |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 0 |
| 3 | 1 |
| ... | ... |




Die Schaltung ist instabil und der Zustand $X$ ändert sich mit hoher Geschwindigkeit.

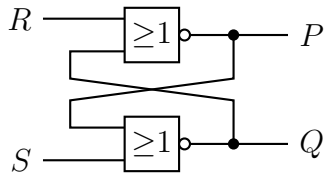
### Das RS-Flipflop

Die folgende Schaltung flimmert zwar nicht mehr aber dafür hängt das Ergebnis möglicherweise davon ab, welches Gatter schneller schaltet.



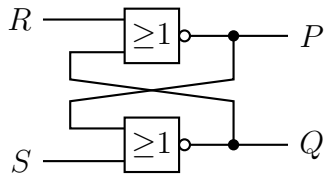

Wir werden gleich sehen, dass dies kein grosses Problem ist und dass die Schaltung dafür einige nützliche Eigenschaften hat.

**Fall 1: Das obere Gatter schaltet schneller**




| $R$ | $S$ | $Q_0$ | $P_1$ | $Q_1$ | $P_2$ | $Q_2$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |

**Fall 2: Das untere Gatter schaltet schneller**




| $R$ | $S$ | $P_0$ | $P_1$ | $Q_1$ | $P_2$ | $Q_2$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 |

**Beobachtungen**

- Nach spätestens zwei Schaltvorgängen sind die Werte in $P$ und $Q$ *stabil*.
- Mit Ausnahme von $R = 1$ und $S = 1$ sind die Werte in $P$ und $Q$ nach zwei Schaltvorgängen *komplementär*; d. h. $P = \overline{Q}$.
- $R = 0$ und $S = 0$: Die Schaltung merkt sich den letzten Zustand von $P$ (und $Q = \overline{P}$).
- $R = 0$ und $S = 1$: Die Schaltung setzt $P = 1$ (und $Q = 0$).
- $R = 1$ und $S = 0$: Die Schaltung setzt $P = 0$ (und $Q = 1$).

**Folgerungen**

- Aus $S = 1$ und $R = 0$ folgt $P = 1$ (*set*).
- Aus $S = 0$ und $R = 1$ folgt $P = 0$ (*reset*).
- Für $S = 0$ und $R = 0$ behält $P$ seinen vorherigen Wert bei.

Sofern man die Eingänge nicht gleichzeitig mit $R = 1$ und $S = 1$ belegt, realisiert ein $RS$-Flipflop einen ...

1 Bit-Speicher.