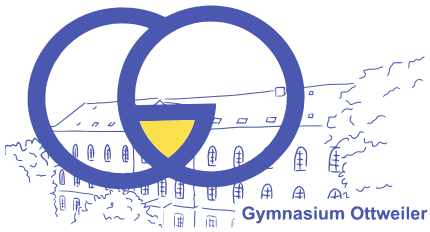
Kapitel 2

Logische Schaltungen und Aussagenlogik



Skript Informatik · Klasse 10 · MINT

Jeder Computer rechnet am Ende mit nur zwei Zuständen: Strom an oder Strom aus, 1 oder 0. In diesem Kapitel siehst du, wie man aus ganz einfachen Bausteinen, den **Gattern**, Schaltungen baut, die Zahlen addieren und sich sogar ein Bit merken können. Danach lernst du die **Aussagenlogik** kennen: eine Schreibweise, mit der man solche Schaltungen genau beschreiben und vereinfachen kann. Eine einfachere Schaltung braucht weniger Bauteile, ist billiger und schneller.

Zum Ausprobieren der Schaltungen benutzt du den Online-Simulator **CircuitVerse**. Python brauchst du in diesem Kapitel nur als Hilfsmittel, um Ergebnisse nachzuprüfen.

Das erwartet dich in diesem Kapitel:

1. **Grundgatter und Belegungstabellen** – die Bausteine UND, ODER, XOR und NICHT und wie man ihr Verhalten in Tabellen festhält.
2. **Schaltungen entwerfen** – aus einer Beschreibung in Worten wird Schritt für Schritt eine Schaltung.
3. **Addition im Binärsystem und Halbaddierer** – schriftlich rechnen mit 0 und 1 und die erste rechnende Schaltung.
4. **Volladdierer und Carry-Ripple-Addierer** – wie ein Computer ganze Binärzahlen addiert.
5. **Das Flipflop** – eine Schaltung, die sich durch Rückkopplung ein Bit merkt.
6. **Aussagenlogik** – Schaltungen als Terme schreiben und als Termbaum darstellen.
7. **Äquivalenz und disjunktive Normalform** – wann zwei Terme dasselbe leisten und wie man zu jeder Tabelle einen Term findet.
8. **KV-Diagramme** – Terme vereinfachen, damit Schaltungen mit weniger Gattern auskommen.

So kannst du es sagen

Mit diesen Satzanfängen beschreibst und begründest du deine Ergebnisse in diesem Kapitel:

- *„Der Term/die Tabelle repräsentiert die Schaltung, weil …“*
- *„Die Rückkopplung bzw. Gruppierung bewirkt …“*

Am Ende hilft dir ein **Rückblick** mit den wichtigsten Begriffen und einem Kompetenz-Check beim Wiederholen.

2.1 Grundgatter und Belegungstabellen

Leitidee

Ein Computer kennt in seinem Innersten nur zwei Zustände: Auf einer Leitung liegt Spannung an, oder es liegt keine Spannung an. Man schreibt dafür **1** und **0**. Trotzdem kann ein Computer rechnen, vergleichen und Entscheidungen treffen. Möglich machen das winzige elektronische Bausteine, die **Gatter**. Jedes Gatter bekommt einen oder zwei Werte 0 oder 1 und liefert daraus nach einer festen Regel wieder eine 0 oder eine 1. Der amerikanische Mathematiker und Ingenieur Claude Shannon hat gezeigt, dass man mit solchen Schaltungen logisch denken und sogar rechnen kann. In diesem Abschnitt lernst du die vier Grundgatter kennen. In den nächsten Abschnitten baust du daraus eine Alarmanlage und ein Rechenwerk.

Lernziele

Nach diesem Abschnitt kannst du

- die vier Grundgatter UND (AND), ODER (OR), XOR und NICHT (NOT) nennen und mit ihren Symbolen skizzieren,
- zu jedem Grundgatter die Belegungstabelle angeben,
- begründen, warum es bei n Eingängen 2^n Belegungen gibt,
- zu einer Schaltung aus mehreren Gattern die Belegungstabelle mit Hilfsspalten erstellen,
- Gatter im Simulator CircuitVerse aufbauen und testen,
- Belegungstabellen mit Python nachprüfen.

2.1.1 Nur zwei Zustände: 0 und 1

Denk an eine Taschenlampe. Der Schalter ist entweder aus oder an, die Lampe ist entweder dunkel oder hell. Einen dritten Zustand gibt es nicht. Genau so arbeitet die Elektronik in einem Computer. Jede Leitung hat zu jedem Zeitpunkt einen von zwei Werten: **0** (keine Spannung, „aus“, „falsch“) oder **1** (Spannung, „an“, „wahr“). Einen solchen Wert, der nur 0 oder 1 sein kann, nennt man ein **Bit**.

Eine **logische Schaltung** wandelt Eingangswerte in Ausgangswerte um. In den Zeichnungen gehen die Leitungen für die **Eingänge** links hinein, die Leitungen für die **Ausgänge** kommen rechts heraus. Die Eingänge bezeichnen wir mit Großbuchstaben wie A, B, C, den Ausgang oft mit Q.

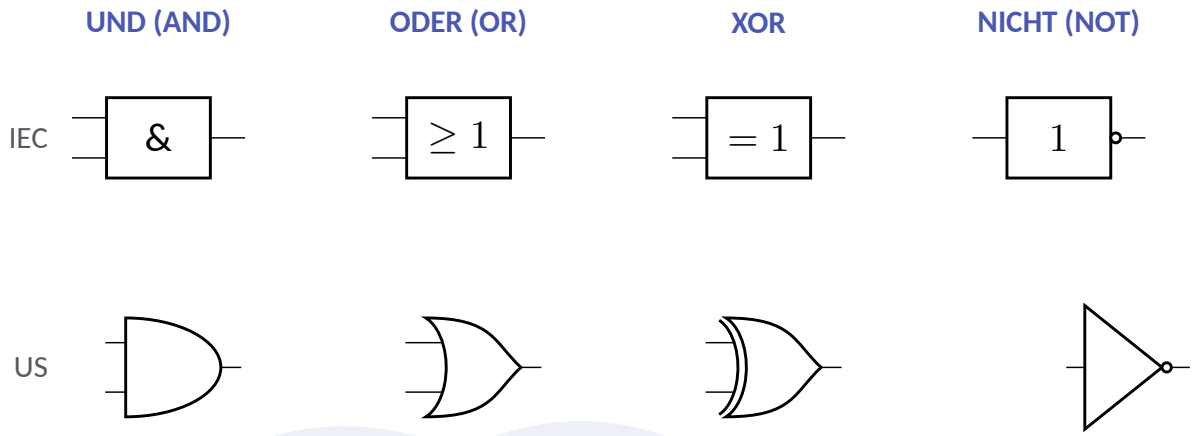
Die kleinsten Bausteine einer logischen Schaltung heißen **Gatter**. Ein Gatter ist ein elektronisches Bauteil mit einem oder mehreren Eingängen und einem Ausgang. Welcher Wert am Ausgang herauskommt, hängt nur davon ab, welche Werte an den Eingängen anliegen. Vier Gatter sind so wichtig, dass man sie **Grundgatter** nennt: UND, ODER, XOR und NICHT. Oft verwendet man die englischen Namen AND, OR, XOR und NOT.

Definition: Grundgatter

Ein **Gatter** ist ein Baustein einer logischen Schaltung, der die Werte 0 oder 1 an seinen Eingängen nach einer festen Regel in einen Ausgangswert 0 oder 1 umwandelt. Die **Grundgatter** sind UND (AND), ODER (OR), XOR und NICHT (NOT).

So sehen Gatter aus

In deutschen Schulbüchern zeichnet man Gatter nach der Norm **IEC 60617**: Jedes Gatter ist ein Rechteck. Im Rechteck steht ein Kennzeichen, das die Regel des Gatters verrät, zum Beispiel & für UND. In diesem Skript werden alle Schaltungen so gezeichnet. Die Abbildung zeigt in der oberen Reihe die vier Grundgatter in dieser Form. Was die Kennzeichen bedeuten, erfährst du gleich bei den einzelnen Gattern.



Die vier Grundgatter: oben die IEC-Symbole (so zeichnest du sie im Heft und in Klassenarbeiten), unten die US-Symbole, die der Simulator CircuitVerse zeigt

Tipp: CircuitVerse zeigt andere Symbole

Zum Ausprobieren benutzt du den kostenlosen Online-Simulator **CircuitVerse**. Du findest ihn unter der Adresse `circuitverse.org/simulator`. CircuitVerse zeichnet die Gatter nicht nach IEC, sondern mit den in den USA üblichen Symbolen (Norm ANSI/IEEE): UND hat die Form eines „D“, ODER ist vorn spitz, XOR sieht aus wie ODER mit einem zusätzlichen Bogen, NICHT ist ein Dreieck mit einem kleinen Kreis an der Spitze (untere Reihe in der Abbildung).

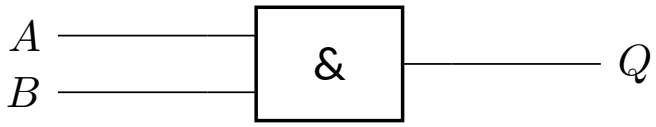
Beide Schreibweisen bedeuten **genau dasselbe**, nur die Zeichnung ist anders. In CircuitVerse geht es nur um das Verhalten der Schaltung. Im Heft und in Klassenarbeiten verwendest du die IEC-Symbole.

2.1.2 Das UND-Gatter (AND)

In einer Werkstatt steht eine Papierschneidemaschine. Damit niemand mit der Hand unter das Messer gerät, gibt es zwei Starttasten, eine links und eine rechts. Das Messer fährt nur herunter, wenn die linke Taste **und** die rechte Taste gedrückt sind. Ist auch nur eine Taste losgelassen, passiert nichts.

Genau diese Regel erfüllt das **UND-Gatter**: Sein Ausgang ist nur dann 1, wenn **alle** Eingänge 1 sind. Das Kennzeichen im Rechteck ist &, das Zeichen für „und“.

Mit zwei Eingängen gibt es genau vier Möglichkeiten, die Eingänge mit 0 und 1 zu **belegen**. Für jede dieser Belegungen schreibt man in einer Tabelle auf, welcher Wert am Ausgang herauskommt (rechts neben dem Symbol):



UND-Gatter: Die Eingänge A und B gehen links hinein, rechts kommt der Ausgang Q heraus

A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

Jede Zeile beschreibt einen Fall. Die dritte Zeile besagt zum Beispiel: Liegt an A eine 1 und an B eine 0 an, dann liefert das UND-Gatter eine 0. Nur in der letzten Zeile sind beide Eingänge 1, nur dort ist der Ausgang 1.

Über der Ausgangsspalte steht $A \wedge B$ (gelesen: „A und B“). Das Zeichen $\wedge$ ist die mathematische Schreibweise für UND. Ein solcher Ausdruck heißt **logischer Term**. Er beschreibt, was die Schaltung mit den Eingängen macht. Die UND-Verknüpfung heißt in der Logik auch **Konjunktion**.

Belegungstabellen

Eine Tabelle wie die oben heißt **Belegungstabelle** (manchmal auch Wahrheitstabelle). Sie ist das wichtigste Werkzeug in diesem Kapitel, denn sie beschreibt das Verhalten einer Schaltung vollständig.

Definition: Belegungstabelle

Eine **Belegungstabelle** (Wahrheitstabelle) listet für jede mögliche Kombination von Eingangswerten den zugehörigen Ausgangswert auf.

Wie viele Zeilen braucht man? Jeder Eingang kann 0 oder 1 sein, hat also 2 Möglichkeiten. Bei einem Eingang gibt es 2 Zeilen. Bei zwei Eingängen kann man jede der 2 Möglichkeiten von A mit jeder der 2 Möglichkeiten von B kombinieren: $2 \cdot 2 = 4$ Zeilen. Jeder weitere Eingang verdoppelt die Anzahl:

Anzahl der Eingänge n	**1**	**2**	**3**	**4**	**5**
Anzahl der Belegungen 2^n	2	4	8	16	32

Damit du keinen Fall vergisst, schreibst du die Zeilen immer in derselben Reihenfolge auf. Fange mit lauter Nullen an. In der **letzten** Eingangsspalte wechseln sich 0 und 1 in jeder Zeile ab. In der Spalte davor wechseln sie alle 2 Zeilen, davor alle 4 Zeilen und so weiter. Die erste Spalte besteht also aus einer Hälfte Nullen und einer Hälfte Einsen. Die letzte Zeile besteht aus lauter Einsen.

Merke

- Eine **Belegungstabelle** (Wahrheitstabelle) listet für jede mögliche Kombination von Eingangswerten den zugehörigen Ausgangswert auf.
- Bei n Eingängen gibt es 2^n mögliche Belegungen.

Arbeitsauftrag 1: Zeilen zählen

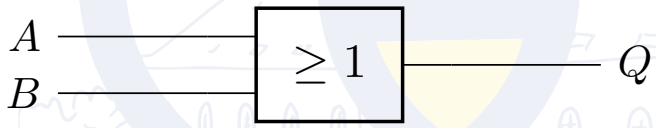
1. Wie viele Zeilen hat die Belegungstabelle einer Schaltung mit 3 Eingängen, wie viele bei 6 Eingängen?
2. Schreibe die Eingangsspalten A, B, C einer Belegungstabelle mit 3 Eingängen in der richtigen Reihenfolge auf.

Deine Antwort:

2.1.3 Das ODER-Gatter (OR)

Im Auto geht die Innenbeleuchtung an, wenn die Fahrertür geöffnet ist **oder** die Beifahrertür geöffnet ist. Sie leuchtet natürlich auch, wenn beide Türen offen sind. Nur wenn beide Türen geschlossen sind, bleibt es dunkel.

Das ist die Regel des **ODER-Gatters**: Sein Ausgang ist 1, wenn **mindestens ein** Eingang 1 ist. Daher kommt das Kennzeichen ≥1 („größer oder gleich 1“): Mindestens eine 1 muss an den Eingängen anliegen. Überlege zuerst selbst, wie die Belegungstabelle aussieht. Lies dann weiter.



ODER-Gatter: Das Kennzeichen ≥ 1 bedeutet „mindestens ein Eingang ist 1“

A	B	$A \vee B$
0	0	0
0	1	1
1	0	1
1	1	1

Das Zeichen $\vee$ steht für ODER; $A \vee B$ liest man „A oder B“. Die ODER-Verknüpfung heißt in der Logik auch **Disjunktion**.

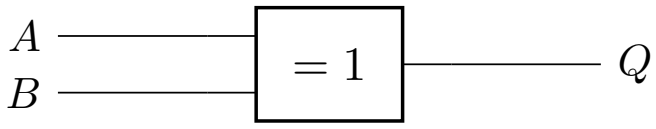
Stolperstein

Im Alltag bedeutet „oder“ oft „das eine oder das andere, aber nicht beides“: „Willst du Tee oder Kaffee?“ Das logische ODER ist anders. Es liefert auch dann eine 1, wenn **beide** Eingänge 1 sind (letzte Zeile der Tabelle). Wenn du in einer Aufgabe „oder“ liest, überlege genau, ob „beides“ erlaubt sein soll.

2.1.4 Das XOR-Gatter

Für das „entweder … oder“ gibt es ein eigenes Gatter: das **XOR-Gatter**. XOR ist die Abkürzung für *exclusive or*, auf Deutsch „ausschließendes Oder“ oder **Exklusiv-Oder**. Sein Ausgang ist 1, wenn die beiden Eingänge **unterschiedlich** sind, also wenn genau einer der beiden Eingänge 1 ist. Das Kennzeichen =1 bedeutet: Es muss **genau eine** 1 anliegen.

Ein Beispiel aus dem Alltag ist die Flurbeleuchtung mit zwei Schaltern, einem an jedem Ende des Flurs. Man stellt die Schaltung so ein, dass das Licht aus ist, wenn beide Schalter unten stehen. Kippt man einen der beiden Schalter um, geht das Licht an. Kippt man danach auch den anderen um, geht es wieder aus. Das Licht brennt also genau dann, wenn die beiden Schalter unterschiedlich stehen.



XOR-Gatter: Das Kennzeichen $= 1$ bedeutet „genau ein Eingang ist 1“

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Das Zeichen $\oplus$ steht für XOR. Vergleiche die Tabelle mit der Tabelle des ODER-Gatters: Nur die letzte Zeile ist anders. Wenn beide Eingänge 1 sind, liefert ODER eine 1, XOR aber eine 0.

2.1.5 Das NICHT-Gatter (NOT)

Das **NICHT-Gatter** hat nur **einen** Eingang. Es dreht den Wert einfach um: Aus 0 wird 1, aus 1 wird 0. Sein Ausgang ist also immer das **Gegenteil** des Eingangs. Man nennt das auch **Negation** (Verneinung). Ein Beispiel ist das Licht im Kühlschrank: Es ist an, wenn die Tür **nicht** geschlossen ist.

Im IEC-Symbol steht eine 1 im Rechteck, denn das Gatter gibt den einen Eingangswert weiter. Dass der Wert umgedreht wird, zeigt ein kleiner **Kreis** direkt am Ausgang. In manchen Büchern steht an seiner Stelle ein kleiner Schrägstrich. Beide Zeichen bedeuten dasselbe: Der Wert wird verneint.



NICHT-Gatter: Der kleine Kreis am Ausgang zeigt, dass der Eingangswert umgedreht wird

A	$\neg A$
0	1
1	0

Das Zeichen $\neg$ steht für NICHT; $\neg A$ liest man „nicht A“.

2.1.6 Die Grundgatter im Überblick

Die folgende Tabelle fasst die vier Grundgatter zusammen. Lerne sie auswendig. Du brauchst sie in allen weiteren Abschnitten dieses Kapitels.

Merke

Gatter	IEC-Symbol	Verhalten
UND (AND, Konjunktion)	&	Ausgang ist 1, wenn **alle** Eingänge 1 sind.
ODER (OR, Disjunktion)	≥1	Ausgang ist 1, wenn **mindestens ein** Eingang 1 ist.
XOR (Exklusiv-Oder)	=1	Ausgang ist 1, wenn die Eingänge **unterschiedlich** sind.
NICHT (NOT, Negation)	1 mit Kreis am Ausgang	Ausgang ist das **Gegenteil** des Eingangs.

So kannst du es sagen

So beschreibst du ein Gatter in Fachsprache – vom einfachen zum genauen Satz:

- *„Da kommt 1 raus, wenn beide 1 sind."* → *„Das UND-Gatter liefert genau dann eine 1, wenn an beiden Eingängen eine 1 anliegt."*
- *„Bei 1 und 0 gibt es eine 1."* → *„Zu der Belegung $A = 1$, $B = 0$ gehört beim ODER-Gatter der Ausgangswert 1."*
- *„Die Zeile 1, 1, 0 heißt …"* → *„Die letzte Zeile der Belegungstabelle besagt: Liegen an beiden Eingängen des XOR-Gatters Einsen an, ist der Ausgangswert 0."*

Arbeitsauftrag 2: Grundgatter skizzieren

Skizziere die vier Grundgatter UND, ODER, XOR und NICHT mit ihren **IEC-Symbolen**. Beschrifte jeweils die Eingänge mit A und B (beim NICHT-Gatter nur A) und den Ausgang mit Q. Schreibe unter jedes Gatter seinen Namen und in einem kurzen Satz, wann sein Ausgang 1 ist.

Tipp

Skizzieren heißt: einen Sachverhalt auf das Wesentliche reduzieren und grafisch oder sprachlich darstellen. Du brauchst also kein Lineal und keine schöne Zeichnung. Wichtig sind das Rechteck, das richtige Kennzeichen, der Kreis beim NICHT-Gatter und die beschrifteten Leitungen.

Deine Skizze:

2.1.7 Gatter in CircuitVerse ausprobieren

Im Simulator CircuitVerse kannst du Schaltungen am Bildschirm zusammenstecken und testen. Du brauchst nur einen Browser. Ein Login ist nicht nötig, aber praktisch: Dann kannst du deine Schaltungen speichern und später weiterbearbeiten. So baust du ein einzelnes Gatter auf:

1. Öffne die Seite `circuitverse.org/simulator`.
2. Im linken Bereich findest du unter **Input** den Baustein **Input**. Das ist ein Schalter, der 0 oder 1 liefert. Ziehe zwei davon ins Arbeitsfeld.
3. Unter **Output** findest du den Baustein **Output**. Er zeigt den Wert an, der bei ihm ankommt (bei einer Lampe: leuchtet = 1). Ziehe einen davon ins Arbeitsfeld.
4. Unter **Gates** findest du AND, OR, XOR und NOT. Ziehe ein Gatter zwischen die Schalter und die Anzeige.
5. Verbinde die Bausteine mit Leitungen: Klicke auf einen Anschluss, halte die Maustaste gedrückt und ziehe zum anderen Anschluss.
6. Klicke auf einen Input-Schalter. Er wechselt zwischen 0 und 1. Beobachte, was die Anzeige am Ausgang zeigt.

Arbeitsauftrag 3: Alle vier Grundgatter testen

Baue in CircuitVerse **nacheinander** je eine Schaltung mit einem AND-, einem OR-, einem XOR- und einem NOT-Gatter (beim NOT-Gatter reicht ein Eingang). Teste **alle** Eingangskombinationen und trage deine Beobachtungen in die Tabellen ein. Vergleiche danach mit den Tabellen weiter vorn in diesem Abschnitt.

A	B	Ausgang AND	Ausgang OR	Ausgang XOR
0	0			
0	1			
1	0			
1	1			

A	Ausgang NOT
0	
1	

2.1.8 Belegungstabellen mit Python nachprüfen

Belegungstabellen kannst du auch mit Python erzeugen und so deine Ergebnisse kontrollieren. Python kennt die logischen Operatoren `and`, `or` und `not` schon. Für XOR gibt es keinen eigenen Operator, aber der Vergleich `!=` („ungleich“) tut genau das Richtige: `a != b` ist wahr, wenn `a` und `b` unterschiedlich sind.

Werkzeug

Python behandelt die Zahl `0` wie `False` und die Zahl `1` wie `True`. Du kannst deshalb `and`, `or` und `not` direkt auf die Werte 0 und 1 anwenden. Das Ergebnis ist manchmal eine Zahl und manchmal ein Wahrheitswert. Damit am Ende immer 0 oder 1 herauskommt, wandeln wir es mit `int(...)` um: `int(True)` ergibt `1`, `int(False)` ergibt `0`.

Für jedes Gatter schreiben wir eine Funktion. Zwei ineinander geschachtelte `for`-Schleifen gehen alle vier Belegungen von `a` und `b` durch:

```
1  def und(a, b):
2      return int(a and b)
3
4  def oder(a, b):
5      return int(a or b)
6
7  def xor(a, b):
8      return int(a != b)
9
10 print("A B | UND ODER XOR")
11 for a in [0, 1]:
12     for b in [0, 1]:
13         print(f"{a} {b} |  {und(a, b)}   {oder(a, b)}   {xor(a, b)}")
```

AUSGABE

```
A B | UND ODER XOR
0 0 |  0   0   0
0 1 |  0   1   1
1 0 |  0   1   1
1 1 |  1   1   0
```

- **Zeilen 1–8:** Drei Funktionen, eine für jedes Gatter. Jede bekommt die beiden Eingangswerte als Parameter und gibt den Ausgangswert 0 oder 1 zurück.
- **Zeile 10:** Die Kopfzeile der Tabelle.
- **Zeilen 11–12:** Die äußere Schleife setzt `a` nacheinander auf 0 und 1. Für **jeden** dieser Werte läuft die innere Schleife einmal ganz durch und setzt `b` auf 0 und 1. So entstehen die vier Belegungen in der üblichen Reihenfolge: 0 0, 0 1, 1 0, 1 1.
- **Zeile 13:** Ein f-String gibt die Belegung und die drei Ausgangswerte in einer Zeile aus.

Das NICHT-Gatter hat nur einen Eingang. Hier genügt eine Schleife:

```
1 def nicht(a):
2     return int(not a)
3
4 for a in [0, 1]:
5     print(a, "->", nicht(a))
```

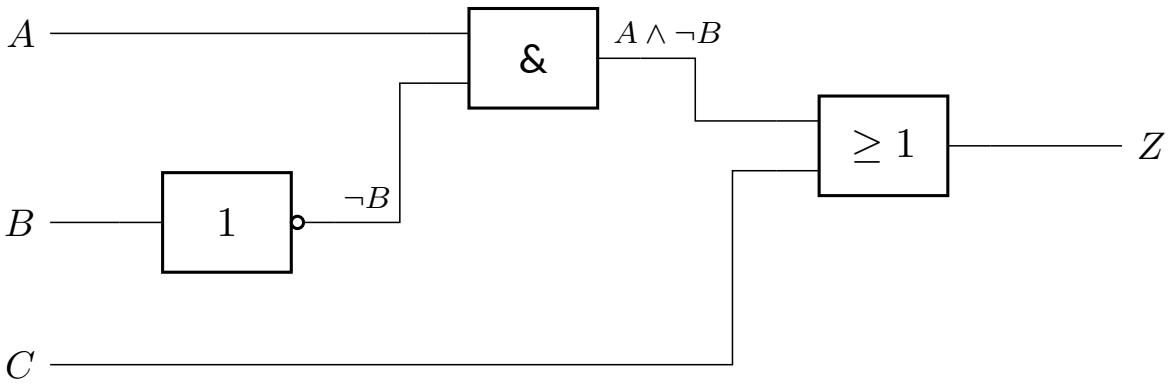
AUSGABE

```
0 -> 1
1 -> 0
```

- **Zeilen 1–2:** Die Funktion `nicht` bekommt nur einen Eingangswert. `not a` dreht ihn um, `int(...)` macht daraus wieder 0 oder 1.
- **Zeilen 4–5:** Die Schleife setzt `a` nacheinander auf 0 und 1 und gibt jeweils Eingang und Ausgang aus.

2.1.9 Schaltungen aus mehreren Gattern

Ein einzelnes Gatter kann nicht viel. Spannend wird es, wenn man Gatter hintereinanderschaltet: Der Ausgang eines Gatters wird zum Eingang des nächsten. Die folgende Schaltung hat drei Eingänge A, B, C und einen Ausgang Z.



Eine Schaltung aus drei Gattern. Sie berechnet den Term $(A \wedge \neg B) \vee C$

Um die Belegungstabelle aufzustellen, gehst du die Schaltung **von links nach rechts** durch, Gatter für Gatter. Für jede Zwischenleitung legst du eine eigene Spalte an. Solche Spalten heißen **Hilfsspalten**.

1. Das NICHT-Gatter dreht B um. Die erste Hilfsspalte ist $\neg B$.
2. Das UND-Gatter verknüpft A mit dem Ausgang des NICHT-Gatters. Die zweite Hilfsspalte ist $A \wedge \neg B$. Sie ist nur dort 1, wo A **und** $\neg B$ beide 1 sind.
3. Das ODER-Gatter verknüpft diese Leitung mit C. Das ergibt den Ausgang $Z = (A \wedge \neg B) \vee C$. Er ist 1, wo die zweite Hilfsspalte **oder** C eine 1 enthält.

A	B	C	$\neg B$	$A \wedge \neg B$	$(A \wedge \neg B) \vee C$
0	0	0	1	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	1	0	0	1
1	0	0	1	1	1
1	0	1	1	1	1
1	1	0	0	0	0
1	1	1	0	0	1

Lies zur Kontrolle eine Zeile in Worten: Die fünfte Zeile besagt, dass zu der Belegung $A = 1, B = 0, C = 0$ der Ausgangswert $Z = 1$ gehört. Das stimmt: $\neg B$ ist 1, also ist auch $A \wedge \neg B$ gleich 1, und damit ist das ODER-Gatter erfüllt.

Merke

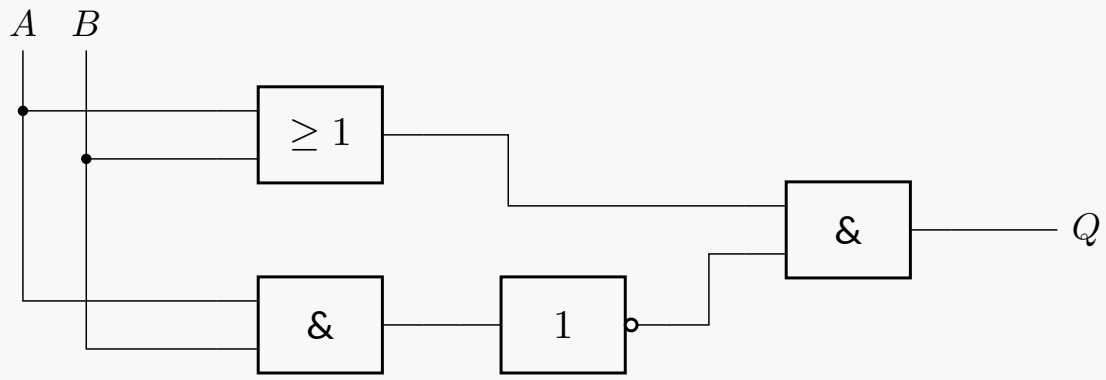
Zu einer Schaltung aus mehreren Gattern erstellt man die Belegungstabelle von links nach rechts: Für den Ausgang jedes Gatters legt man eine **Hilfsspalte** an und berechnet sie aus den Spalten, die schon da sind.

Stolperstein

Der häufigste Fehler bei Belegungstabellen ist, eine Zeile auszulassen oder doppelt zu schreiben. Halte dich deshalb immer an die feste Reihenfolge der Zeilen (letzte Eingangsspalte wechselt in jeder Zeile). Zähle am Ende nach: Bei 3 Eingängen müssen es genau 8 Zeilen sein.

Arbeitsauftrag 4: Belegungstabelle zu einer Schaltung

Erstelle die Belegungstabelle zur folgenden Schaltung. Fülle dazu die Hilfsspalten von links nach rechts aus. Mit welchem Grundgatter stimmt die letzte Spalte überein?



Schaltung mit zwei Eingängen und vier Gattern

A	B	$A \vee B$	$A \wedge B$	$\neg(A \wedge B)$	Q
0	0				
0	1				
1	0				
1	1				

Die letzte Spalte stimmt überein mit:

Arbeitsauftrag 5: Mehrheitsschaltung

Baue in CircuitVerse eine Schaltung mit **drei Eingängen** A, B, C, die genau dann 1 ausgibt, wenn **mindestens zwei** der drei Eingänge 1 sind (Mehrheitsentscheidung, englisch *majority*). Verwende dafür mehrere AND- und OR-Gatter.

1. Fülle die Belegungstabelle aus.

2. Skizziere die Schaltung mit IEC-Symbolen.
3. Teste deine Schaltung in CircuitVerse mit allen acht Belegungen.

Tipp: Die Schaltung lässt sich als $(A \wedge B) \vee (B \wedge C) \vee (A \wedge C)$ beschreiben: Entweder sind A und B beide 1, oder B und C, oder A und C. In CircuitVerse kannst du beim OR-Gatter die Anzahl der Eingänge auf 3 stellen oder zwei OR-Gatter hintereinanderschalten.

A	B	C	$A \wedge B$	$B \wedge C$	$A \wedge C$	Q
0	0	0				
0	0	1				
0	1	0				
0	1	1				
1	0	0				
1	0	1				
1	1	0				
1	1	1				

Deine Skizze:

Gymnasium Ottweiler

2.1.10 Übungen

Arbeitsauftrag 6: Welches Gatter ist das?

Zu vier unbekannten Gattern mit den Eingängen A und B wurden die Ausgangswerte Q_1 bis Q_4 gemessen. Gib jeweils an, um welches Gatter es sich handelt. Eine der vier Spalten passt zu keinem Grundgatter. Begründe, woran du das erkennst.

A	B	Q_1	Q_2	Q_3	Q_4
0	0	0	0	0	1
0	1	1	0	1	1
1	0	1	0	1	1
1	1	1	1	0	0

Deine Antwort:

2.1.11 Weitere Übungen

Hier kannst du das Gelernte weiter üben. Die Punkte im Titel zeigen, wie schwer eine Aufgabe ist: ●○○ leicht, ●●○ mittel, ●●● knifflig. Fang mit den leichten an.

Arbeitsauftrag 7: Gatter erkennen ●○○

Die Abbildung zeigt vier Gatter in IEC-Darstellung. Nenne zu jedem Gatter den Namen und beschreibe in einem kurzen Satz, wann sein Ausgang 1 ist.



Vier Gatter, durchnummeriert von (1) bis (4)

Deine Antwort:

Gymnasium Ottweiler

Arbeitsauftrag 8: Ausgang bestimmen ●○○

Bestimme für jede Zeile den Ausgangswert des Gatters. Beim NICHT-Gatter gibt es nur den Eingang A.

Gatter	A	B	Ausgang
UND	1	0	
ODER	1	0	
XOR	1	0	
UND	1	1	
ODER	1	1	
XOR	1	1	
ODER	0	0	
NICHT	1	–	

Arbeitsauftrag 9: Gatter im Alltag ●○○

Gib zu jeder Situation das passende Grundgatter an. Begründe kurz.

a) Ein Handy lässt sich entsperren, wenn der Fingerabdruck erkannt wird oder die richtige PIN eingegeben wird.
b) Ein Online-Spiel startet nur, wenn du eingeloggt bist und das Internet verbunden ist.
c) Bei einem Quiz hat jedes der beiden Teams einen Buzzer. Die Lampe „Antwort“ leuchtet, wenn genau ein Team gedrückt hat. Drücken beide gleichzeitig, bleibt sie aus.
d) Die Warnleuchte am Wasserkocher leuchtet, wenn **kein** Wasser im Kocher ist. (Eingang: 1 = Wasser ist im Kocher.)

Deine Antwort:

Gymnasium Ottweiler

Arbeitsauftrag 10: Welche Zeile fehlt? ●○○

Bei dieser Belegungstabelle für drei Eingänge fehlt eine Zeile.

1. Gib an, wie viele Zeilen die Tabelle haben muss.
2. Bestimme die fehlende Belegung und gib an, an welcher Stelle sie eingefügt werden muss.

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

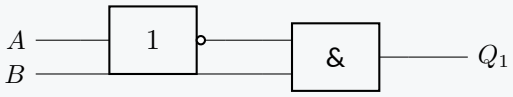
Deine Antwort:

Gymnasium Ottweiler

Arbeitsauftrag 11: Zwei kleine Schaltungen ●●○

Erstelle zu beiden Schaltungen die Belegungstabelle. Fülle die Hilfsspalten aus. Beschreibe danach in Worten, wann $Q_2 = 1$ ist.

a)



b)



Links Schaltung a) mit dem Ausgang Q_1, rechts Schaltung b) mit dem Ausgang Q_2

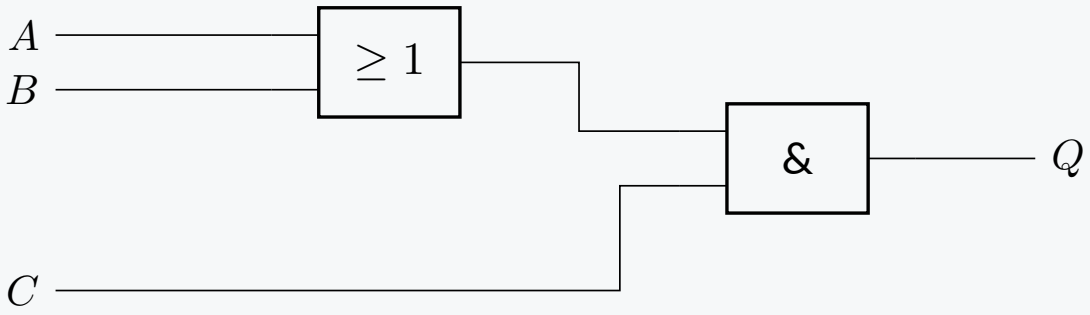
A	B	$\neg A$	Q_1	$A \oplus B$	Q_2
0	0				
0	1				
1	0				
1	1				

Deine Beschreibung:

Gymnasium Ottweiler

Arbeitsauftrag 12: Eine Schaltung mit drei Eingängen ●●○

Erstelle die Belegungstabelle zur folgenden Schaltung. Gib danach alle Belegungen an, für die $Q = 1$ ist.



Schaltung mit drei Eingängen: ein ODER-Gatter und ein UND-Gatter

A	B	C	$A \vee B$	Q
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Gymnasium Ottweiler

Deine Antwort:

Arbeitsauftrag 13: Kontrolle mit Python ●●○

Überprüfe deine Tabellen aus den Aufträgen *Zwei kleine Schaltungen* und *Eine Schaltung mit drei Eingängen* mit Python. Schreibe für jede der drei Schaltungen eine Funktion (`q1(a, b)`, `q2(a, b)` und `q3(a, b, c)`), die 0 oder 1 zurückgibt. Gib die Belegungstabellen mit geschachtelten `for`-Schleifen aus.

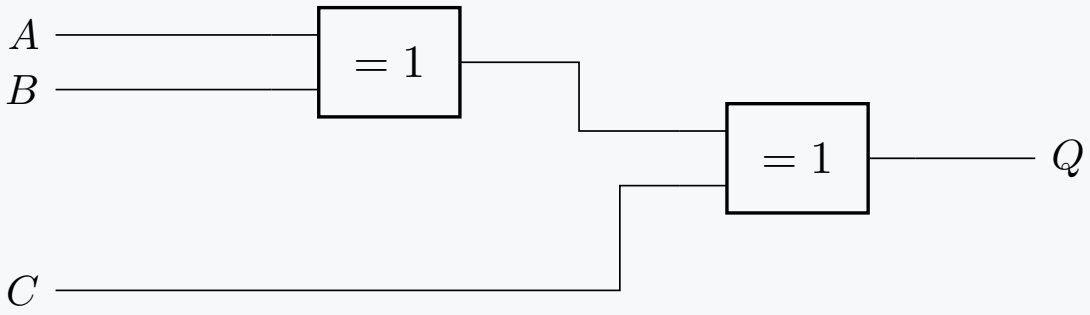
Tipp

XOR schreibst du in Python als `a != b`. Ein NICHT-Gatter hinter dem XOR-Gatter wird dann zu `not (a != b)`. Die Klammern sind wichtig, damit `not` sich auf das ganze Ergebnis bezieht.

Arbeitsauftrag 14: Zwei XOR-Gatter hintereinander ●●●

In der folgenden Schaltung ist der Ausgang eines XOR-Gatters mit einem zweiten XOR-Gatter verbunden.

1. Erstelle die Belegungstabelle mit der Hilfsspalte $A \oplus B$.
2. Zähle in jeder Zeile, wie viele der drei Eingänge 1 sind, und trage die Anzahl in die Spalte „Einsen“ ein. Beschreibe damit in einem Satz, wann $Q = 1$ ist.



Zwei XOR-Gatter hintereinander

A	B	C	$A \oplus B$	Q	**Einsen**
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

Deine Beschreibung:

2.1.12 Das Wichtigste in Kürze

- In logischen Schaltungen hat jede Leitung den Wert **0** oder **1** (ein **Bit**).
- Die vier **Grundgatter**: UND (&, $A \wedge B$, alle Eingänge 1), ODER (≥1, $A \vee B$, mindestens ein Eingang 1), XOR (=1, $A \oplus B$, Eingänge unterschiedlich), NICHT (1 mit Kreis am Ausgang, $\neg A$, Gegenteil des Eingangs).
- Eine **Belegungstabelle** listet zu jeder Kombination der Eingangswerte den Ausgangswert auf. Bei n Eingängen hat sie 2^n Zeilen.
- Bei Schaltungen aus mehreren Gattern rechnet man von links nach rechts und legt für jede Zwischenleitung eine **Hilfsspalte** an.
- In Zeichnungen verwendet man die IEC-Symbole. CircuitVerse zeigt die US-Symbole; sie bedeuten dasselbe.

Selbstcheck

- ☐ Ich kann die vier Grundgatter nennen und mit IEC-Symbolen skizzieren.
- ☐ Ich kann zu jedem Grundgatter die Belegungstabelle aus dem Kopf aufschreiben.
- ☐ Ich kann den Unterschied zwischen ODER und XOR erklären.
- ☐ Ich weiß, warum eine Belegungstabelle mit 3 Eingängen 8 Zeilen hat, und kann die Zeilen in der richtigen Reihenfolge aufschreiben.
- ☐ Ich kann zu einer Schaltung aus mehreren Gattern die Belegungstabelle mit Hilfsspalten erstellen.
- ☐ Ich kann ein Gatter in CircuitVerse aufbauen, testen und die US-Symbole lesen.

Gymnasium Ottweiler

2.2 Schaltungen entwerfen: vom Text zur Schaltung

Leitidee

Im letzten Abschnitt hast du Schaltungen untersucht, die schon fertig gezeichnet waren. Jetzt drehst du den Spieß um: Du bekommst eine Beschreibung in Worten, zum Beispiel „Die Alarmanlage soll losgehen, wenn …“, und entwirfst selbst die passende Schaltung. Genau so arbeiten Ingenieurinnen und Ingenieure, wenn sie die Steuerung für eine Alarmanlage, einen Fahrstuhl oder eine Waschmaschine bauen. Damit dabei nichts schiefgeht, gehst du immer in denselben Schritten vor: von der Beschreibung über die Belegungstabelle zum Term und zur Schaltung. Am Ende testest du die Schaltung im Simulator.

Lernziele

Nach diesem Abschnitt kannst du

- zu einer Beschreibung in Worten die Eingänge und Ausgänge einer Schaltung festlegen,
- die Belegungstabelle aufstellen, die die Beschreibung erfüllt,
- die Beschreibung in einen logischen Term übersetzen und daraus die Schaltung skizzieren,
- die Schaltung in CircuitVerse aufbauen und mit allen Belegungen testen,
- dein Ergebnis mit einem kurzen Python-Programm nachprüfen.

2.2.1 Das Vorgehen beim Schaltungsentwurf

Eine Anforderung in Worten ist oft ungenau. Bevor du Gatter zeichnest, übersetzt du sie deshalb Schritt für Schritt in eine genaue Form. Die Belegungstabelle ist dabei das Herzstück: Sie legt für **jeden** möglichen Fall fest, was die Schaltung tun soll.

Merke

Beim Schaltungsentwurf übersetzt man eine **Anforderung in Textform** zunächst in eine **Belegungstabelle**, danach in eine passende **Gatterkombination**. Vorgehen:

1. Eingangs- und Ausgangsgrößen benennen (und festlegen, was 0 und 1 jeweils bedeuten).
2. Belegungstabelle aufstellen: alle Fälle durchgehen und markieren, in welchen der Ausgang 1 sein soll.
3. Passende Gatter auswählen und verschalten. Dabei hilft ein logischer Term.
4. Schaltung testen: alle Belegungen durchprobieren und mit der Tabelle vergleichen.

Bei Schritt 3 hilft dir die Sprache der Aufgabe. Die Wörter „und“, „oder“ und „nicht“ in der Beschreibung passen oft direkt zu den Gattern UND, ODER und NICHT. Ob das wirklich stimmt, prüfst du immer an der Belegungstabelle.

2.2.2 Beispiel: eine Alarmanlage

Eine Alarmanlage soll auslösen, wenn die **Tür geöffnet** ist **und** die Anlage **scharfgeschaltet** ist, **oder** wenn der **Rauchmelder** anschlägt. Bei Rauch soll der Alarm also immer losgehen, egal ob die Anlage scharfgeschaltet ist oder nicht.

Schritt 1: Eingänge und Ausgang benennen

Lies den Text und frage dich: Welche Informationen bekommt die Anlage? Das sind die Eingänge. Was soll sie tun? Das ist der Ausgang. Für jede Größe legst du fest, was 0 und was 1 bedeutet.

Größe	Art	Bedeutung von 1	Bedeutung von 0
Tür	Eingang	Tür ist geöffnet	Tür ist geschlossen
Scharf	Eingang	Anlage ist scharfgeschaltet	Anlage ist aus
Rauch	Eingang	Rauchmelder schlägt an	kein Rauch
Alarm	Ausgang	Alarm wird ausgelöst	kein Alarm

Die Schaltung hat also drei Eingänge und einen Ausgang.

Schritt 2: Belegungstabelle aufstellen

Drei Eingänge bedeuten $2^3 = 8$ Zeilen. Schreibe die Eingangsspalten in der gewohnten Reihenfolge auf. Gehe dann jede Zeile durch und frage: Soll hier Alarm sein? Die Hilfsspalte für „Tür offen und scharf“ macht das leichter.

Tür	Scharf	Rauch	Tür $\wedge$ Scharf	Alarm
0	0	0	0	0
0	0	1	0	1
0	1	0	0	0
0	1	1	0	1
1	0	0	0	0
1	0	1	0	1
1	1	0	1	1
1	1	1	1	1

Lies einige Zeilen in Worten, um die Tabelle zu prüfen:

- **Zeile 2** (0, 0, 1): Die Tür ist zu, die Anlage ist aus, aber es brennt. Es muss Alarm geben.
- **Zeile 5** (1, 0, 0): Die Tür ist offen, aber die Anlage ist nicht scharf. Das ist der normale Fall, wenn jemand zu Hause ist und das Haus betritt. Kein Alarm.
- **Zeile 7** (1, 1, 0): Die Tür wird geöffnet, während die Anlage scharf ist. Das könnte ein Einbruch sein. Alarm.

Schritt 3: Term und Schaltung

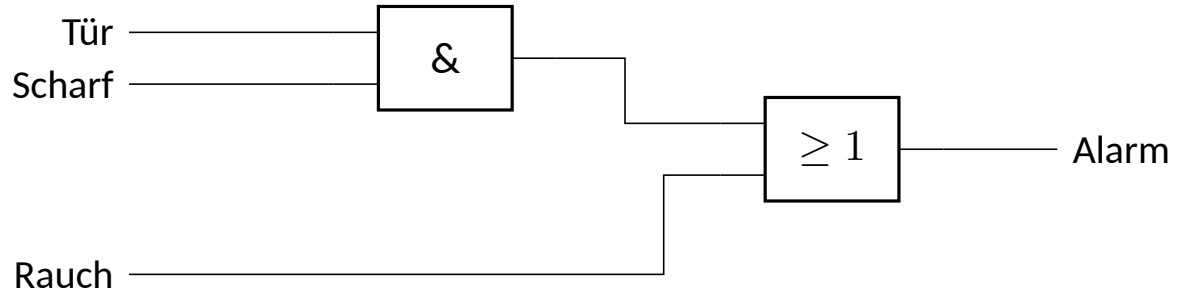
Jetzt übersetzt du die Beschreibung in einen logischen Term. Der Satz hat zwei Teile, die durch „oder“ verbunden sind:

- „die Tür ist geöffnet **und** die Anlage ist scharfgeschaltet“ wird zu Tür $\wedge$ Scharf,
- „der Rauchmelder schlägt an“ ist einfach Rauch.

Beide Teile werden mit ODER verknüpft. Weil das UND zuerst ausgewertet werden soll, kommt es in Klammern:

$$\text{Alarm} = (\text{Tür} \wedge \text{Scharf}) \vee \text{Rauch}$$

Der Term sagt dir direkt, welche Gatter du brauchst: ein UND-Gatter für die Klammer und ein ODER-Gatter, das den Ausgang des UND-Gatters mit dem Rauchmelder verknüpft. Was in der Klammer steht, liegt in der Schaltung weiter links, denn es wird zuerst berechnet.



Schaltung der Alarmanlage: Alarm $=$ (Tür $\wedge$ Scharf) $\vee$ Rauch

Vergleiche die Schaltung mit der Tabelle: Die Leitung zwischen den beiden Gattern führt genau die Werte der Hilfsspalte Tür $\wedge$ Scharf.

Schritt 4: Testen

Eine Schaltung ist erst fertig, wenn sie getestet ist. Dazu baust du sie in CircuitVerse nach und probierst **alle acht** Belegungen aus.

Arbeitsauftrag 15: Alarmanlage in CircuitVerse

Baue die Schaltung der Alarmanlage in CircuitVerse nach. Verwende drei Input-Schalter für Tür, Scharf und Rauch, ein AND-Gatter, ein OR-Gatter und einen Output für Alarm. Teste alle acht Belegungen und vergleiche mit der Belegungstabelle oben. Notiere, welche Zeilen du getestet hast.

Getestete Zeilen und Auffälligkeiten:

Gymnasium Ottweiler

Tipp

In CircuitVerse kannst du jedem Input und Output einen Namen geben (Feld *Label* in den Eigenschaften des Bausteins). Mit beschrifteten Schaltern verwechselst du beim Testen keine Eingänge.

Auch Python kann die Belegungstabelle zur Kontrolle ausgeben. Für drei Eingänge brauchst du drei ineinander geschachtelte Schleifen:

```
1 def alarm(tuer, scharf, rauch):
2     return int((tuer and scharf) or rauch)
3
4 print("Tür Scharf Rauch -> Alarm")
5 for tuer in [0, 1]:
6     for scharf in [0, 1]:
7         for rauch in [0, 1]:
8             ergebnis = alarm(tuer, scharf, rauch)
9             print(f" {tuer}    {scharf}      {rauch}    -> {ergebnis}")
```

AUSGABE

```
 Tür Scharf Rauch -> Alarm
  0    0      0   -> 0
  0    0      1   -> 1
  0    1      0   -> 0
  0    1      1   -> 1
  1    0      0   -> 0
  1    0      1   -> 1
  1    1      0   -> 1
  1    1      1   -> 1
```

- **Zeilen 1–2:** Die Funktion `alarm` enthält den Term der Schaltung. Die Klammern stehen genau wie im Term. `int(...)` sorgt dafür, dass das Ergebnis 0 oder 1 ist.
- **Zeile 4:** Die Kopfzeile der Tabelle.
- **Zeilen 5–7:** Drei Schleifen erzeugen alle acht Belegungen in der üblichen Reihenfolge. Die innerste Schleife (`rauch`) wechselt am schnellsten.
- **Zeilen 8–9:** Für jede Belegung wird der Ausgang berechnet und zusammen mit den Eingängen ausgegeben. Die Leerzeichen im f-String sorgen dafür, dass die Werte unter den Überschriften stehen.

Die Ausgabe stimmt Zeile für Zeile mit der Belegungstabelle überein. Die Schaltung erfüllt also die Anforderung.

Stolperstein

Achte auf die **Klammern**. Der Term Tür $\wedge$ (Scharf $\vee$ Rauch) sieht ähnlich aus, beschreibt aber eine andere Schaltung. Vergleiche die beiden Terme in der Belegungstabelle (hier abgekürzt: T = Tür, S = Scharf, R = Rauch):

T	S	R	$(T \wedge S) \vee R$	$T \wedge (S \vee R)$
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	1	0
1	0	0	0	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

In der zweiten und vierten Zeile unterscheiden sie sich: Es brennt, aber die Tür ist geschlossen. Mit dem falschen Term bliebe die Anlage stumm. Ein Test mit **allen** Belegungen deckt solche Fehler auf.

So kannst du es sagen

So beschreibst du eine Schaltung in Fachsprache – vom einfachen zum genauen Satz:

- *„Links gehen drei Leitungen rein, rechts eine raus."* → *„Die Schaltung hat drei Eingänge und einen Ausgang."*
- *„Bei 0, 0, 1 kommt 1 raus."* → *„Zu der Kombination der Eingangswerte Tür = 0, Scharf = 0, Rauch = 1 gehört der Ausgangswert Alarm = 1."*
- *„Die Formel ist die Schaltung."* → *„Der logische Term* $(\text{Tür} \wedge \text{Scharf}) \vee \text{Rauch}$ *repräsentiert die Schaltung der Alarmanlage."*

2.2.3 Typische Fehler beim Entwurf

- **Bedeutung von 0 und 1 nicht festgelegt.** Schreibe in Schritt 1 immer auf, was 1 bedeutet. Bei der Fahrstuhltür in den Übungen unten bedeutet 1 bei den Lichtschranken „frei", nicht „blockiert".
- **Alltags-Oder mit XOR verwechseln.** Prüfe beim Wort „oder", ob „beides gleichzeitig" erlaubt ist. Dann ist es ODER, sonst XOR.
- **Klammern vergessen.** Was in der Beschreibung zusammengehört, gehört im Term in eine Klammer und in der Schaltung zu einem eigenen Gatter.
- **Nicht alle Fälle getestet.** Eine Schaltung, die in drei Fällen funktioniert, kann im vierten falsch sein. Teste immer alle 2^n Belegungen.

2.2.4 Übungen

Arbeitsauftrag 16: Fahrstuhltür

Eine Fahrstuhltür darf sich nur schließen (Ausgang Schließen = 1), wenn **beide** Lichtschranken LS1 und LS2 frei sind **und** der Zeitgeber Zeit abgelaufen ist. Bei den Lichtschranken bedeutet 0 „blockiert" und 1 „frei". Beim Zeitgeber bedeutet 1 „abgelaufen".

1. Stelle die Belegungstabelle auf.
2. Gib einen logischen Term an und skizziere die Schaltung.
3. Baue die Schaltung in CircuitVerse und teste sie.

4. Kontrolliere dein Ergebnis mit Python. Ergänze dazu die Funktion `schliessen` im folgenden Programm:

```
def schliessen(ls1, ls2, zeit):
    return ...   # hier ergänzen

for ls1 in [0, 1]:
    for ls2 in [0, 1]:
        for zeit in [0, 1]:
            ergebnis = schliessen(ls1, ls2, zeit)
            print(ls1, ls2, zeit, "->", ergebnis)
```

LS1	LS2	Zeit	Schließen
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Term und Skizze:

Gymnasium Ottweiler

Arbeitsauftrag 17: Abstimmungsschaltung mit Vetorecht

Drei Personen A, B und C stimmen ab (0 = dagegen, 1 = dafür). Ein Antrag ist angenommen (Ausgang $Q = 1$), wenn die **Mehrheit** dafür ist. **Aber:** Stimmt Person A dagegen, ist der Antrag **in jedem Fall** abgelehnt (Vetorecht).

1. Überlege dir zunächst in Worten, wann der Ausgang 1 sein muss.
2. Stelle die Belegungstabelle auf. Die Hilfsspalte „Mehrheit“ gibt an, ob mindestens zwei Personen dafür sind.
3. Gib einen Term an, skizziere die Schaltung und baue sie in CircuitVerse.

4. Vergleiche mit einer Python-Kontrolle. Schreibe dazu eine Funktion `antrag_angenommen(a, b, c)`.

A	B	C	**Mehrheit**	Q
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Tipp

Die Mehrheitsschaltung kennst du schon aus dem letzten Abschnitt: $(A \wedge B) \vee (B \wedge C) \vee (A \wedge C)$. Überlege, wie das Vetorecht von A mit diesem Ergebnis verknüpft werden muss.

In Worten, Term, Skizze und Python-Funktion:

Arbeitsauftrag 18: Gartenbewässerung

Eine Pumpe bewässert einen Garten. Sie soll laufen (Pumpe = 1), wenn der Boden **trocken** ist und es **nicht regnet**. Außerdem gibt es einen **Handschalter**: Ist er eingeschaltet, läuft die Pumpe immer.

1. Benenne die Eingänge und den Ausgang und lege fest, was 0 und 1 bedeuten.
2. Stelle die Belegungstabelle auf (mit Hilfsspalten).

3. Gib einen Term an und skizziere die Schaltung.

Deine Lösung:

2.2.5 Weitere Übungen

Hier kannst du das Gelernte weiter üben. Die Punkte im Titel zeigen, wie schwer eine Aufgabe ist: ●○○ leicht, ●●○ mittel, ●●● knifflig. Fang mit den leichten an.

Arbeitsauftrag 19: Eingänge und Ausgang benennen ●○○

Ein Ventilator im Zimmer soll laufen, wenn es im Zimmer **warm** ist **und** eine **Person** im Zimmer ist. Ein Temperaturfühler und ein Bewegungsmelder liefern die Eingangswerte.

Ergänze die Tabelle: Ist die Größe ein Eingang oder ein Ausgang? Was bedeuten 1 und 0?

Größe	Art	Bedeutung von 1	Bedeutung von 0
Warm			
Person			
Ventilator			

Arbeitsauftrag 20: Schaltung für den Ventilator ●○○

Setze den Auftrag *Eingänge und Ausgang benennen* fort.

1. Stelle die Belegungstabelle auf.
2. Gib einen logischen Term an und skizziere die Schaltung mit IEC-Symbolen.

Warm	Person	Ventilator
0	0	
0	1	
1	0	
1	1	

Term und Skizze:

Arbeitsauftrag 21: Sätze in Terme übersetzen ●○○

Übersetze jeden Satz in einen logischen Term. Nenne die Gatter, die du für die Schaltung brauchst.

a) Das Tablet lädt (L), wenn das Kabel steckt (K) und die Steckdose Strom hat (S).
b) Die Klingel tönt (T), wenn an der Haustür (H) oder an der Gartentür (G) jemand drückt.
c) Der Bildschirm des Laptops ist an (D), wenn der Deckel nicht zugeklappt ist. ($Z = 1$ bedeutet: Der Deckel ist zugeklappt.)
d) Die Handykamera blitzt (B), wenn es dunkel ist (N) und der Blitz nicht ausgeschaltet ist. ($X = 1$ bedeutet: Der Blitz ist ausgeschaltet.)

Deine Terme:

Arbeitsauftrag 22: Welcher Term passt? ●○○

Das Licht auf dem Pausenhof geht an ($L = 1$), wenn es dunkel ist ($D = 1$) und der Bewegungsmelder etwas meldet ($B = 1$), oder wenn der Hausmeister den Schalter einschaltet ($H = 1$).

Welcher der drei Terme beschreibt die Anlage richtig? Begründe deine Wahl.

① $(D \wedge B) \vee H$ ② $D \wedge (B \vee H)$ ③ $(D \vee B) \wedge H$

Deine Antwort:

Gymnasium Ottweiler

Arbeitsauftrag 23: Einlass ins Kino ●●○

Am Eingang eines Kinosaals öffnet sich eine Schranke (Auf $= 1$), wenn ein gültiges Ticket gescannt wurde ($T = 1$) **und außerdem** die Person mindestens 16 Jahre alt ist ($S = 1$) **oder** von einer erwachsenen Person begleitet wird ($E = 1$).

1. Stelle die Belegungstabelle auf. Die Hilfsspalte gibt an, ob die Altersregel erfüllt ist.
2. Gib einen logischen Term an.
3. Skizziere die Schaltung mit IEC-Symbolen.

T	S	E	$S \vee E$	**Auf**
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Term und Skizze:

Arbeitsauftrag 24: Kinoeinlass mit Python prüfen ●●○

Schreibe eine Funktion `schranke(t, s, e)`, die den Term aus dem Auftrag *Einlass ins Kino* berechnet und 0 oder 1 zurückgibt. Gib mit drei geschachtelten `for`-Schleifen die Belegungstabelle aus und vergleiche sie mit deiner Tabelle.

Arbeitsauftrag 25: Wasserspiel mit Sturmabschaltung ●●●

Ein Wasserspiel im Freizeitpark spritzt ($W = 1$), wenn Musik läuft ($M = 1$) oder ein Kind auf die Bodenplatte tritt ($K = 1$). Bei Sturm ($S = 1$) bleibt es aber **in jedem Fall** aus.

1. Stelle die Belegungstabelle auf (mit Hilfsspalten).
2. Gib einen Term an und skizziere die Schaltung.
3. Jonas schlägt den Term $M \vee K \vee \neg S$ vor. Gib eine Belegung an, bei der sein Term das Falsche liefert, und erkläre seinen Fehler.

Tabelle, Term, Skizze und Antwort zu Jonas:

2.2.6 Das Wichtigste in Kürze

- Beim Schaltungsentwurf geht man in festen Schritten vor: Eingänge und Ausgänge benennen, Belegungstabelle aufstellen, Term und Schaltung entwickeln, Schaltung testen.
- Für jeden Eingang und Ausgang legt man fest, was 0 und 1 bedeuten.
- Die Wörter „und“, „oder“, „nicht“ einer Beschreibung führen oft zu den Gattern UND, ODER, NICHT. Klammern im Term zeigen, was zuerst berechnet wird.
- Das Beispiel Alarmanlage: Alarm $=$ (Tür $\wedge$ Scharf) $\vee$ Rauch.
- Verschiedene Schaltungen können dieselbe Belegungstabelle haben. Dann erfüllen beide die Anforderung; die mit weniger Gattern ist billiger.
- Getestet wird mit allen Belegungen, zum Beispiel in CircuitVerse oder mit einem Python-Programm mit geschachtelten Schleifen.

Selbstcheck

- ☐ Ich kann zu einer Beschreibung die Eingänge und Ausgänge benennen und festlegen, was 0 und 1 bedeuten.
- ☐ Ich kann zu einer Beschreibung die Belegungstabelle aufstellen.
- ☐ Ich kann eine Beschreibung in einen logischen Term mit Klammern übersetzen.
- ☐ Ich kann aus dem Term die Schaltung skizzieren.
- ☐ Ich kann eine Schaltung in CircuitVerse aufbauen und mit allen Belegungen testen.
- ☐ Ich kann die Alarmanlage als Beispiel erklären.

2.3 Addition im Binärsystem und Halbaddierer

Leitidee

Jeder Taschenrechner, jedes Handy und jeder Computer kann in Bruchteilen einer Sekunde Millionen von Zahlen addieren. Dabei kennt die Elektronik nur 0 und 1, und rechnen können ihre Gatter eigentlich gar nicht: Sie verknüpfen nur Bits nach festen Regeln. Wie kann daraus ein Rechenwerk werden? Der Trick ist, Zahlen im **Binärsystem** zu schreiben, also nur mit den Ziffern 0 und 1. Dann funktioniert die Addition genau wie die schriftliche Addition aus der Grundschule. In diesem Abschnitt rechnest du zuerst selbst im Binärsystem. Danach baust du aus zwei Gattern eine Schaltung, die zwei Bits addiert: den **Halbaddierer**.

Lernziele

Nach diesem Abschnitt kannst du

- natürliche Zahlen vom Dezimalsystem ins Binärsystem umrechnen und umgekehrt,
- zwei natürliche Zahlen im Binärsystem schriftlich addieren und dabei die Überträge richtig berücksichtigen,
- den Aufbau des Halbaddierers aus einem XOR- und einem UND-Gatter skizzieren,
- die Funktionsweise des Halbaddierers mit seiner Belegungstabelle erläutern,
- einen Halbaddierer in CircuitVerse bauen und als Python-Funktion nachbilden.

2.3.1 Zahlen im Binärsystem

Im Alltag schreiben wir Zahlen im **Dezimalsystem** (Zehnersystem). Es gibt zehn Ziffern, 0 bis 9. Welchen Wert eine Ziffer hat, hängt von ihrer Stelle ab. In der Zahl 347 steht die 3 an der Hunderterstelle, die 4 an der Zehnerstelle und die 7 an der Einerstelle: $347 = 3 \cdot 100 + 4 \cdot 10 + 7 \cdot 1$. Von Stelle zu Stelle wird der Stellenwert zehnmal so groß.

Das **Binärsystem** (Zweiersystem) funktioniert genauso, nur mit zwei Ziffern: 0 und 1. Von Stelle zu Stelle wird der Stellenwert **doppelt** so groß. Von rechts nach links sind die Stellenwerte also 1, 2, 4, 8, 16, 32 und so weiter. Jede Stelle einer Binärzahl ist ein Bit.

Definition: Binärsystem

Im **Binärsystem** (Zweiersystem) werden Zahlen nur mit den Ziffern 0 und 1 geschrieben. Die Stellenwerte sind von rechts nach links 1, 2, 4, 8, 16, ...; jeder ist doppelt so groß wie der vorige.

Von binär nach dezimal

Um eine Binärzahl in eine Dezimalzahl umzurechnen, schreibst du sie in eine **Stellenwerttabelle**. Dann addierst du die Stellenwerte, unter denen eine 1 steht. Beispiel: die Binärzahl 1101.

Stellenwert	**8**	**4**	**2**	**1**
Ziffer	1	1	0	1

Unter 8, 4 und 1 steht eine 1. Also ist die Binärzahl 1101 gleich $8+4+1 = 13$. Damit man die Zahlensysteme nicht verwechselt, schreibt man manchmal das System klein dahinter: $1101_2 = 13_{10}$.

Von dezimal nach binär

Für den umgekehrten Weg füllst du die Stellenwerttabelle von links nach rechts. Beginne mit dem größten Stellenwert, der in die Zahl passt. Beispiel: die Dezimalzahl 22.

1. Die Stellenwerte sind 1, 2, 4, 8, 16, 32, … Der größte, der in 22 passt, ist 16. Schreibe unter 16 eine 1. Es bleiben $22 - 16 = 6$.
2. Passt 8 in 6? Nein. Schreibe unter 8 eine 0.
3. Passt 4 in 6? Ja. Schreibe unter 4 eine 1. Es bleiben $6 - 4 = 2$.
4. Passt 2 in 2? Ja. Schreibe unter 2 eine 1. Es bleibt 0.
5. Unter 1 kommt eine 0, denn es ist nichts mehr übrig.

Stellenwert	**16**	**8**	**4**	**2**	**1**
Ziffer	1	0	1	1	0

Also ist $22_{10} = 10110_2$. Zur Probe rechnest du zurück: $16 + 4 + 2 = 22$.

Nullen am Anfang ändern den Wert einer Zahl nicht: 0101 und 101 sind beide gleich 5. Im Folgenden schreiben wir Binärzahlen oft mit vier Stellen, also 0101 statt 101. Dann stehen beim Rechnen alle Stellen schön untereinander.

Merke

Im Binärsystem gibt es nur die Ziffern 0 und 1. Die Stellenwerte sind von rechts nach links 1, 2, 4, 8, 16, … Den Dezimalwert einer Binärzahl erhält man, indem man die Stellenwerte addiert, unter denen eine 1 steht.

Arbeitsauftrag 26: Umrechnen

Rechne um. Mache jeweils die Probe.

dezimal	**6**	**9**	**12**	**15**
binär				

binär	**0111**	**1010**	**1011**	**10000**
dezimal				

Werkzeug

Python kann Zahlen zwischen Dezimal- und Binärsystem umrechnen. Diese beiden Befehle brauchst du nur zum Nachprüfen:

- `int("1101", 2)` wandelt die Zeichenkette `"1101"` in eine Zahl um. Die `2` sagt Python, dass die Ziffern im Binärsystem gemeint sind. Ergebnis: `13`.
- `bin(13)` liefert die Binärschreibweise von 13 als Zeichenkette: `'0b1101'`. Die Vorsilbe `0b` zeigt nur an, dass eine Binärzahl folgt.

```
print(int("1101", 2))
print(bin(22))
```

AUSGABE

```
13
0b10110
```

2.3.2 Schriftliche Addition im Binärsystem

Erinnere dich an die schriftliche Addition im Dezimalsystem. Du rechnest Stelle für Stelle von rechts nach links. Wird eine Stelle 10 oder größer, schreibst du nur die Einerziffer hin und nimmst eine 1 als **Übertrag** in die nächste Stelle mit:

```
    4 7
  + 3 8
Ü   1
  -----
    8 5
```

Rechts: $7 + 8 = 15$. Du schreibst 5 und überträgst 1. Links: $4 + 3 + 1 = 8$. Die Zeile mit dem Ü enthält die Überträge. Jeder Übertrag steht genau unter der Stelle, zu der er addiert wird.

Im Binärsystem ist es genauso, nur dass schon bei 2 ein Übertrag entsteht. Denn eine Ziffer 2 gibt es nicht; 2 schreibt man binär als 10. Für zwei Bits gibt es nur vier Fälle:

Rechnung	Ergebnis (binär)	schreibe	Übertrag
$0 + 0$	0	0	0
$0 + 1$	1	1	0
$1 + 0$	1	1	0
$1 + 1$	10	0	1

Kommt ein Übertrag aus der Stelle davor dazu, können auch drei Einsen zusammenkommen: $1 + 1 + 1 = 3$, binär 11. Dann schreibst du 1 und überträgst 1.

Merke

- Die **binäre Addition** funktioniert wie die schriftliche Addition im Dezimalsystem, nur mit den Ziffern 0 und 1: $0 + 0 = 0$, $0 + 1 = 1$, $1 + 0 = 1$, $1 + 1 = 10$ (also 0 schreiben, 1 **Übertrag**).
- Mit einem Übertrag aus der Stelle davor gilt außerdem $1 + 1 + 1 = 11$ (also 1 schreiben, 1 Übertrag).

Beispiel: 0101 + 0011

Wir addieren 0101 (das ist 5) und 0011 (das ist 3):

```
      0 1 0 1     (= 5)
    + 0 0 1 1     (= 3)
Ü     1 1 1
    ---------
      1 0 0 0     (= 8)
```

Von rechts nach links:

Stelle	Rechnung	schreibe	Übertrag in die nächste Stelle
1 (ganz rechts)	$1+1=10$	0	1
2	$0+1+1=10$	0	1
3	$1+0+1=10$	0	1
4	$0+0+1=1$	1	0

Ergebnis: 1000. Probe: $1000_2 = 8$ und $5+3=8$. ✓

In diesem Beispiel entsteht in fast jeder Stelle ein Übertrag. Ab der zweiten Stelle werden deshalb **drei** Bits addiert: die beiden Ziffern der Zahlen und der Übertrag. Das wird im nächsten Abschnitt wichtig.

Stolperstein

Typische Fehler bei der Binäraddition:

- **„1 + 1 = 2“ hinschreiben.** Die Ziffer 2 gibt es im Binärsystem nicht. Richtig: 0 schreiben, 1 übertragen.
- **Übertrag vergessen.** Schreibe jeden Übertrag sofort in die Ü-Zeile, und zwar unter die nächste Stelle links.
- **Stellen verrutscht.** Ergänze kürzere Zahlen vorn mit Nullen, damit die Stellen genau untereinanderstehen.
- **Letzten Übertrag vergessen.** Entsteht ganz links noch ein Übertrag, wird das Ergebnis eine Stelle länger: $1111 + 0001 = 10000$.

Mache immer die Probe im Dezimalsystem.

Arbeitsauftrag 27: Von Hand rechnen

Berechne **von Hand** Schritt für Schritt mit Überträgen die folgenden binären Additionen. Kontrolliere anschließend mit der Probe im Dezimalsystem und mit Python (siehe Werkzeug-Kasten oben).

1. $0110 + 0101$
2. $1011 + 0111$
3. $1111 + 0001$

Deine Rechnungen:

2.3.3 Eine Schaltung, die zwei Bits addiert

Jetzt soll eine Schaltung die Arbeit übernehmen. Wir beginnen mit der einfachsten Aufgabe: der Stelle ganz rechts. Dort werden nur **zwei** Bits addiert, nennen wir sie A und B. Das Ergebnis besteht aus zwei Bits: der Ziffer, die man hinschreibt, und dem Übertrag. Man nennt sie **Summe** S und **Übertrag** C (vom englischen *carry*, „tragen“).

A	B	$A+B$ **(dezimal)**	**Übertrag** C	**Summe** S
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	2	1	0

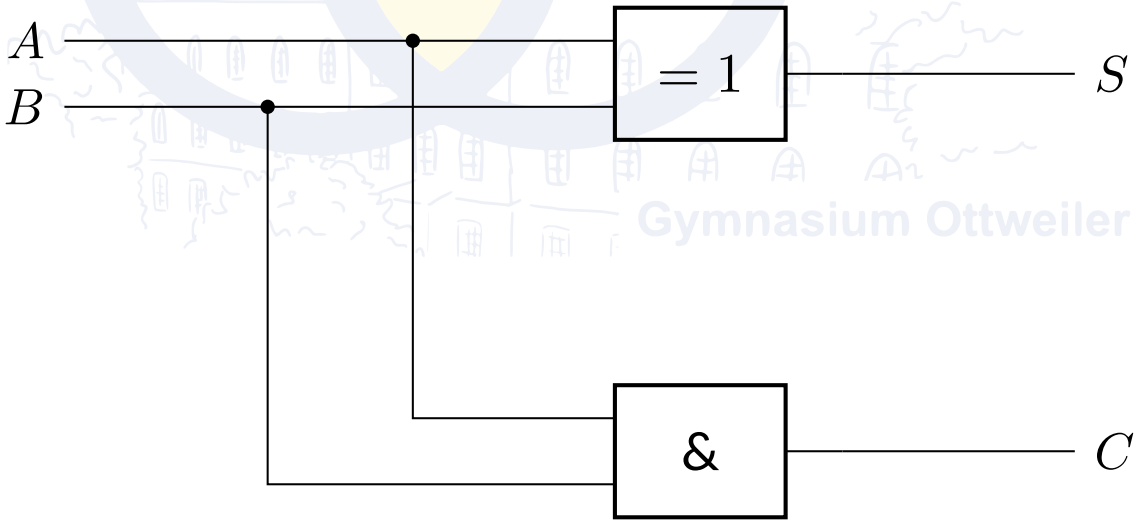
Liest man C und S zusammen als zweistellige Binärzahl, steht dort genau die Summe: 00, 01, 01, 10, also 0, 1, 1, 2.

Schau dir die beiden Ausgangsspalten genau an. Kommen sie dir bekannt vor?

- Die Spalte S ist 1, wenn A und B **unterschiedlich** sind. Das ist die Belegungstabelle des **XOR-Gatters**.
- Die Spalte C ist nur 1, wenn A **und** B beide 1 sind. Das ist die Belegungstabelle des **UND-Gatters**.

Mehr braucht man nicht. Eine Schaltung aus einem XOR-Gatter und einem UND-Gatter addiert zwei Bits. Sie heißt **Halbaddierer**.

2.3.4 Der Halbaddierer



Halbaddierer: Das XOR-Gatter liefert die Summe S, das UND-Gatter den Übertrag C

So ist der Halbaddierer aufgebaut:

- Die beiden Eingänge A und B werden **beiden** Gattern zugeführt. Deshalb verzweigen sich die Leitungen. Die Punkte markieren die Verzweigungen. Wo sich die Leitung von A und die Leitung von B nur kreuzen, ist kein Punkt.

- Das XOR-Gatter berechnet die Summe: $S = A \oplus B$.
- Das UND-Gatter berechnet den Übertrag: $C = A \wedge B$.

Die Belegungstabelle des Halbaddierers fasst das zusammen:

A	B	$S = A \oplus B$	$C = A \wedge B$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Funktionsweise an einem Beispiel: Liegt an A und an B je eine 1 an, dann sind die Eingänge gleich. Das XOR-Gatter liefert also $S = 0$. Das UND-Gatter liefert $C = 1$, weil beide Eingänge 1 sind. Zusammen ergibt das die Binärzahl 10, also 2. Das ist richtig, denn $1 + 1 = 2$.

Definition: Halbaddierer

Ein **Halbaddierer** ist eine Schaltung mit zwei Eingängen A, B und zwei Ausgängen. Er addiert die beiden Bits und liefert die Summe $S = A \oplus B$ (XOR-Gatter) und den Übertrag $C = A \wedge B$ (UND-Gatter).

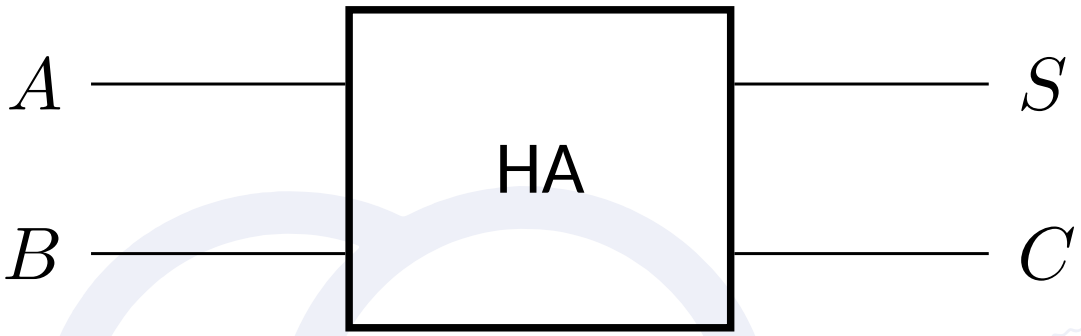
Warum heißt er „halb“? Schau noch einmal auf das Beispiel $0101 + 0011$. In der Stelle ganz rechts werden zwei Bits addiert; das schafft der Halbaddierer. Ab der zweiten Stelle muss aber zusätzlich der **Übertrag aus der Stelle davor** addiert werden. Dafür hat der Halbaddierer keinen Eingang. Er erledigt also nur die halbe Arbeit. Eine Schaltung, die auch den eingehenden Übertrag verarbeitet, lernst du im nächsten Abschnitt kennen: den Volladdierer.

Gymnasium Ottweiler

Merke

- Ein **Halbaddierer** addiert **zwei einzelne Bits** (A, B) und liefert zwei Ausgänge:
  - **Summe** $S = A \oplus B$ (XOR),
  - **Übertrag** $C = A \wedge B$ (UND; englisch *carry*).
- „Halb“-Addierer heißt er, weil er **keinen eingehenden Übertrag** aus einer vorherigen Stelle berücksichtigen kann. Das kann erst der Volladdierer.

Wenn man den Halbaddierer in größere Schaltungen einbaut, zeichnet man oft nicht jedes Gatter, sondern nur einen Kasten mit der Aufschrift HA. So ein vereinfachtes Bild heißt **Blockschaltbild**. Es zeigt nur, was hineingeht und was herauskommt.



Blockschaltbild des Halbaddierers: zwei Eingänge, zwei Ausgänge

So kannst du es sagen

So beschreibst du den Halbaddierer in Fachsprache – vom einfachen zum genauen Satz:

- *„Da gehen zwei Leitungen rein und zwei raus.“ → „Ein Halbaddierer hat zwei Eingänge und zwei Ausgänge.“*
- *„Der rechnet A plus B.“ → „Ein Halbaddierer addiert zwei Bits und liefert die Summe und den Übertrag.“*
- *„Bei 1 und 1 kommt 0 und 1 raus.“ → „Ein Halbaddierer addiert die an den beiden Eingängen anliegenden Bits und liefert deren Summe und den entstehenden Übertrag. Liegen zwei Einsen an, ist die Summe 0 und der Übertrag 1.“*

Arbeitsauftrag 28: Halbaddierer in CircuitVerse

1. Lege zwei Input-Schalter A und B an.
2. Füge ein **XOR-Gatter** hinzu (Eingänge: A, B). Sein Ausgang ist die **Summe** S.
3. Füge ein **AND-Gatter** hinzu (Eingänge: A, B). Sein Ausgang ist der **Übertrag** C.
4. Verbinde beide Gatterausgänge mit je einem Output.
5. Teste alle vier Belegungen, trage die Werte ein und vergleiche mit der Belegungstabelle oben.

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

Arbeitsauftrag 29: Halbaddierer in Python

Schreibe eine Funktion `halbaddierer(a, b)`, die die Summe und den Übertrag als Liste `[summe, uebertrag]` zurückgibt. Teste sie für alle vier Belegungen.

Tipp

Mit `return` gibt eine Funktion einen Wert zurück. Wenn du zwei Ergebnisse brauchst, packst du sie in eine Liste: `return [summe, uebertrag]`. An die einzelnen Werte kommst du später mit dem Index heran: `ergebnis[0]` ist die Summe, `ergebnis[1]` der Übertrag.

2.3.5 Übungen

Arbeitsauftrag 30: Halbaddierer erläutern

1. Erläutere den Aufbau und die Funktionsweise eines Halbaddierers. Gehe dabei auf die Belegung $A = 0, B = 1$ ein.
2. Begründe am Beispiel $0101 + 0011$, warum ein Halbaddierer für die zweite Stelle von rechts nicht ausreicht.

Gymnasium Ottweiler

Deine Antwort:

Arbeitsauftrag 31: Ergebnis vorhersagen

An einem Halbaddierer liegt $A = 1$ und $B = 0$ an. Gib ohne Tabelle an, welche Werte S und C haben, und begründe mit den beiden Gattern. Welche Belegung führt zu $C = 1$?

Deine Antwort:

Gymnasium Ottweiler

2.3.6 Weitere Übungen

Hier kannst du das Gelernte weiter üben. Die Punkte im Titel zeigen, wie schwer eine Aufgabe ist: ●○○ leicht, ●●○ mittel, ●●● knifflig. Fang mit den leichten an.

Arbeitsauftrag 32: Bits addieren ●○○

Bestimme für jede Rechnung, welche Ziffer man hinschreibt und welcher Übertrag entsteht.

Rechnung	Ergebnis (binär)	schreibe	Übertrag
$1 + 0$			
$1 + 1$			
$0 + 0 + 1$			
$1 + 0 + 1$			
$1 + 1 + 1$			

Arbeitsauftrag 33: Addieren ohne Übertrag ●○○

Berechne von Hand. Die Dezimalwerte stehen zur Kontrolle in Klammern.

a) $0100 + 0011$ (4 + 3)
b) $1010 + 0101$ (10 + 5)
c) $1000 + 0110$ (8 + 6)

Woran erkennst du schon vor dem Rechnen, dass kein Übertrag entsteht?

Deine Rechnungen und deine Antwort:

Gymnasium Ottweiler

Arbeitsauftrag 34: Addieren mit einem Übertrag ●○○

Berechne von Hand. Schreibe den Übertrag in die Ü-Zeile.

a) $0010 + 0010$ (2 + 2)
b) $0101 + 0001$ (5 + 1)
c) $0110 + 0100$ (6 + 4)

Deine Rechnungen:

Arbeitsauftrag 35: Lückentext zum Halbaddierer ●○○

Ergänze die Lücken (1) bis (7).

Ein Halbaddierer hat (1) Eingänge und (2) Ausgänge. Die Summe S liefert ein (3)-Gatter, den Übertrag C ein (4)-Gatter. Liegt an beiden Eingängen eine 1 an, ist $S =$ (5) und $C =$ (6). „Halb“ heißt er, weil er keinen (7) aus der Stelle davor verarbeiten kann.

Deine Lösungen:

Arbeitsauftrag 36: Addieren mit mehreren Überträgen ●●○

Berechne von Hand Schritt für Schritt. Schreibe alle Überträge in die Ü-Zeile. Prüfe dein Ergebnis anschließend mit Python.

a) $0111 + 0011$ (7 + 3)
b) $0110 + 0110$ (6 + 6)
c) $1110 + 0011$ (14 + 3)

Deine Rechnungen:

Arbeitsauftrag 37: Fehler finden ●●○

Lena und Tim haben gerechnet. Beide Ergebnisse sind falsch.

```
Lena:   0 1 0 1          Tim:   0 1 1 0
      + 0 0 0 1                + 0 1 1 1
        ---------                ---------
          0 1 0 2                  0 0 0 1
```

1. Beschreibe jeweils den Fehler.
2. Rechne beide Aufgaben richtig.
3. Tim hat in jeder Spalte dasselbe Gatter „nachgespielt“. Welches Gatter ist das?

Deine Antwort und die richtigen Rechnungen:

Arbeitsauftrag 38: Rechnungen mit Python prüfen ●●○

Schreibe eine Funktion `addiere(a_text, b_text)`, die zwei Binärzahlen als Zeichenketten bekommt. Sie soll die Summe als Binärzahl ausgeben und zur Kontrolle auch die Rechnung im Dezimalsystem. Prüfe damit deine Ergebnisse aus dem Auftrag *Addieren mit mehreren Überträgen*.

Für `addiere("0111", "0011")` soll die Ausgabe so aussehen:

```
0111 + 0011 = 0b1010 | dezimal: 7 + 3 = 10
```

Tipp

Die Befehle `int("0111", 2)` und `bin(...)` kennst du aus dem Werkzeug-Kasten im Abschnitt über das Binärsystem.

Arbeitsauftrag 39: Fehlende Ziffern ●●●

In jeder Rechnung ist eine Ziffer durch **x** ersetzt. Bestimme x und begründe, Spalte für Spalte von rechts.

a) $1\text{x}01 + 0110 = 10011$
b) $01\text{x}1 + 0101 = 1100$

Deine Begründung:

2.3.7 Das Wichtigste in Kürze

- Im **Binärsystem** gibt es nur die Ziffern 0 und 1. Die Stellenwerte sind 1, 2, 4, 8, 16, … Beispiel: $1101_2 = 8 + 4 + 1 = 13$.
- Binärzahlen werden schriftlich addiert wie Dezimalzahlen: Stelle für Stelle von rechts nach links. Es gilt $1 + 1 = 10$ (0 schreiben, 1 übertragen) und $1 + 1 + 1 = 11$.
- Die Überträge schreibt man in eine eigene Zeile. Die Probe im Dezimalsystem deckt Fehler auf.
- Ein **Halbaddierer** addiert zwei Bits: Summe $S = A \oplus B$ (XOR-Gatter), Übertrag $C = A \wedge B$ (UND-Gatter).
- Er heißt „halb“, weil er keinen Übertrag aus der Stelle davor verarbeiten kann.

Selbstcheck

- ☐ Ich kann eine Binärzahl mit der Stellenwerttabelle in eine Dezimalzahl umrechnen und umgekehrt.
- ☐ Ich kann zwei Binärzahlen schriftlich addieren und schreibe die Überträge in eine eigene Zeile.
- ☐ Ich weiß, was $1 + 1$ und $1 + 1 + 1$ im Binärsystem ergibt.
- ☐ Ich kann den Halbaddierer aus XOR- und UND-Gatter skizzieren.
- ☐ Ich kann die Belegungstabelle des Halbaddierers angeben und seine Funktionsweise erläutern.
- ☐ Ich kann erklären, warum der Halbaddierer „halb“ heißt.

Gymnasium Ottweiler

2.4 Volladdierer und Carry-Ripple-Addierer

Leitidee

Mit dem Halbaddierer kann eine Schaltung zwei Bits addieren. Das reicht aber nur für die Stelle ganz rechts. Schon in der zweiten Stelle kommt der Übertrag aus der ersten Stelle dazu, und dann müssen **drei** Bits addiert werden. Dafür gibt es den **Volladdierer**. Er lässt sich aus zwei Halbaddierern und einem ODER-Gatter zusammensetzen. Schaltet man mehrere Volladdierer hintereinander, erhält man ein Rechenwerk, das ganze Binärzahlen addiert. Es arbeitet genau so, wie du schriftlich addierst: Stelle für Stelle, und der Übertrag wird jeweils an die nächste Stelle weitergereicht.

Lernziele

Nach diesem Abschnitt kannst du

- erklären, warum beim Addieren mehrstelliger Binärzahlen in jeder Stelle drei Bits addiert werden,
- den Aufbau eines Volladdierers aus zwei Halbaddierern und einem ODER-Gatter skizzieren und seine Funktionsweise erläutern,
- die Belegungstabelle des Volladdierers angeben,
- einen Carry-Ripple-Addierer skizzieren und erläutern, wie der Übertrag von Stelle zu Stelle wandert,
- eine Addition Stufe für Stufe durch einen Carry-Ripple-Addierer verfolgen.

2.4.1 Drei Bits addieren

Erinnere dich an die Rechnung $0101 + 0011$ aus dem letzten Abschnitt:

```
      0 1 0 1     (= 5)
    + 0 0 1 1     (= 3)
Ü     1 1 1
    ---------
      1 0 0 0     (= 8)
```

In der zweiten Stelle von rechts rechnest du $0 + 1 + 1$: die Ziffer der ersten Zahl, die Ziffer der zweiten Zahl und den Übertrag aus der ersten Stelle. Eine Schaltung für diese Stelle braucht also **drei Eingänge**:

- A und B: die beiden Bits, die an dieser Stelle addiert werden,
- C_{ein}: den **eingehenden Übertrag** aus der Stelle rechts davon.

Die Summe von drei Bits ist höchstens $1 + 1 + 1 = 3$, binär 11. Das Ergebnis passt also wieder in zwei Bits: die **Summe** S, die an dieser Stelle hingeschrieben wird, und den **ausgehenden Übertrag** C_{aus}, der an die nächste Stelle links weitergegeben wird. Im Englischen heißen die Überträge *carry in* und *carry out*, kurz C_{in} und C_{out}.

A	B	C_{ein}	**Einsen**	C_{aus}	S
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	1	0	1
0	1	1	2	1	0
1	0	0	1	0	1
1	0	1	2	1	0
1	1	0	2	1	0
1	1	1	3	1	1

Die Spalte „Einsen“ gibt an, wie viele der drei Eingänge 1 sind. Liest man C_{aus} und S zusammen als Binärzahl, steht dort genau diese Anzahl: 00, 01, 10 oder 11. Zwei Beobachtungen helfen beim Merken:

- S ist 1, wenn die Anzahl der Einsen **ungerade** ist (1 oder 3).
- C_{aus} ist 1, wenn **mindestens zwei** Eingänge 1 sind. Das ist genau die Mehrheitsschaltung, die du schon kennst.

Definition: Volladdierer

Ein **Volladdierer** ist eine Schaltung mit drei Eingängen und zwei Ausgängen. Er addiert die beiden Bits A und B und den eingehenden Übertrag C_{ein} und liefert die Summe S und den ausgehenden Übertrag C_{aus}.

So kannst du es sagen

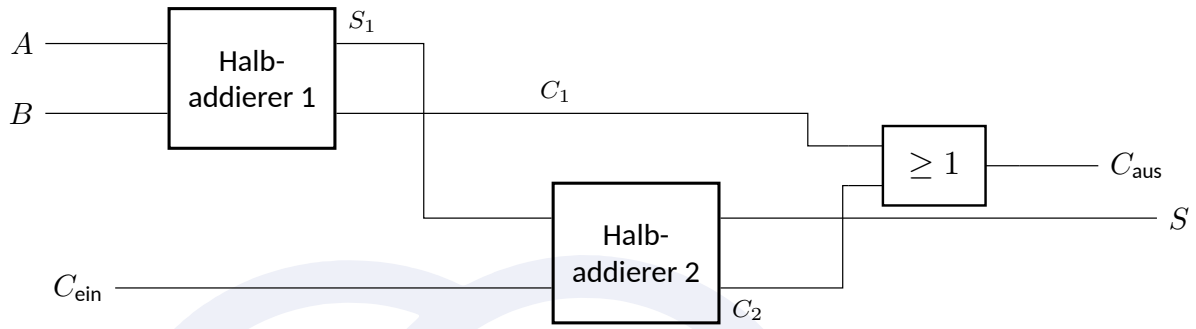
So beschreibst du den Volladdierer in Fachsprache – vom einfachen zum genauen Satz:

- *„Es gehen links drei Leitungen hinein und rechts zwei hinaus.“* → *„Ein Volladdierer hat drei Eingänge und zwei Ausgänge.“*
- *„Der rechnet drei Bits zusammen.“* → *„Ein Volladdierer addiert drei Bits und liefert die Summe und den Übertrag.“*
- *„Da kommt die Summe und der Übertrag raus.“* → *„Ein Volladdierer addiert die an den drei Eingängen anliegenden Bits und liefert deren Summe und den entstehenden Übertrag.“*

2.4.2 Der Volladdierer aus zwei Halbaddierern

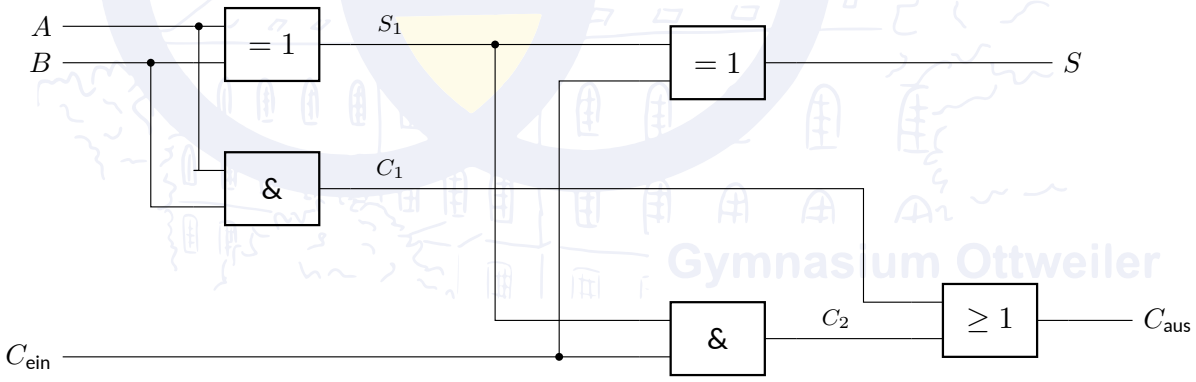
Wie baut man eine solche Schaltung? Du musst keine neuen Gatter erfinden. Denk daran, wie du drei Zahlen im Kopf addierst: Erst addierst du zwei davon, dann addierst du die dritte zum Zwischenergebnis. Genauso arbeitet der Volladdierer mit zwei Halbaddierern:

1. **Halbaddierer 1** addiert A und B. Er liefert eine Zwischensumme S_1 und einen Übertrag C_1.
2. **Halbaddierer 2** addiert die Zwischensumme S_1 und den eingehenden Übertrag C_{ein}. Er liefert die endgültige Summe S und einen zweiten Übertrag C_2.
3. Ist bei einem der beiden Schritte ein Übertrag entstanden, gibt es einen ausgehenden Übertrag. Ein **ODER-Gatter** verknüpft deshalb C_1 und C_2: $C_{\text{aus}} = C_1 \vee C_2$.



Volladdierer aus zwei Halbaddierern und einem ODER-Gatter

Jeder Halbaddierer besteht aus einem XOR- und einem UND-Gatter. Zeichnet man alle Gatter einzeln, sieht der Volladdierer so aus. Diese Zeichnung brauchst du, wenn du ihn in CircuitVerse aufbaust:



Volladdierer aus Gattern: oben links Halbaddierer 1, rechts Halbaddierer 2, ganz rechts das ODER-Gatter

Die Belegungstabelle mit den Zwischenergebnissen als Hilfsspalten zeigt, dass der Aufbau stimmt. Die beiden letzten Spalten sind genau die Spalten S und C_{aus} aus der Tabelle oben.

A	B	C_{ein}	S_1	C_1	C_2	S	C_{aus}
0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0
0	1	0	1	0	0	1	0
0	1	1	1	0	1	0	1
1	0	0	1	0	0	1	0
1	0	1	1	0	1	0	1
1	1	0	0	1	0	0	1
1	1	1	0	1	0	1	1

Dabei gilt $S_1 = A \oplus B$, $C_1 = A \wedge B$, $C_2 = S_1 \wedge C_{\text{ein}}$, $S = S_1 \oplus C_{\text{ein}}$ und $C_{\text{aus}} = C_1 \vee C_2$.

Funktionsweise an einem Beispiel (letzte Zeile): An allen drei Eingängen liegt eine 1 an. Halbaddierer 1 rechnet $1 + 1 = 10$, also $S_1 = 0$ und $C_1 = 1$. Halbaddierer 2 rechnet $S_1 + C_{\text{ein}} = 0 + 1 = 1$, also $S = 1$ und $C_2 = 0$. Das ODER-Gatter liefert $C_{\text{aus}} = 1 \vee 0 = 1$. Zusammen ergibt das 11, also 3. Das stimmt: $1 + 1 + 1 = 3$.

Tipp

Warum genügt am Ende ein ODER-Gatter? Könnten nicht beide Halbaddierer gleichzeitig einen Übertrag liefern? Nein: $C_1 = 1$ heißt, dass A und B beide 1 sind. Dann ist aber $S_1 = 0$, und der zweite Halbaddierer addiert höchstens $0 + 1$. Er liefert also keinen Übertrag. In der Tabelle sind C_1 und C_2 nie gleichzeitig 1.

Merke

- Ein **Volladdierer** addiert **drei Bits**: die beiden zu addierenden Bits A und B sowie einen **eingehenden Übertrag** C_{ein} aus der vorherigen Stelle. Er liefert die **Summe** S und einen **ausgehenden Übertrag** C_{aus}.
- Ein Volladdierer lässt sich aus **zwei Halbaddierern und einem ODER-Gatter** aufbauen:
  1. Halbaddierer 1: A und B ergeben die Zwischensumme S_1 und den Übertrag C_1.
  2. Halbaddierer 2: S_1 und C_{ein} ergeben die Summe S und den Übertrag C_2.
  3. $C_{\text{aus}} = C_1 \vee C_2$.

Arbeitsauftrag 40: Volladdierer in CircuitVerse

1. Baue **zwei Halbaddierer** wie im letzten Abschnitt (je ein XOR- und ein AND-Gatter).
2. Verbinde den Summenausgang des ersten Halbaddierers mit einem Eingang des zweiten Halbaddierers. Der zweite Eingang ist C_{ein}.
3. Verknüpfe die beiden Übertragsausgänge C_1 und C_2 mit einem **OR-Gatter**. Dessen Ausgang ist C_{aus}.
4. Teste alle **acht** möglichen Belegungen von A, B und C_{ein}, trage die Werte ein und vergleiche mit der Belegungstabelle oben.

A	B	C_{ein}	S	C_{aus}
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Tipp

In CircuitVerse gibt es auch einen fertigen Volladdierer-Baustein (englisch *Full Adder* bzw. *Adder*). Baue den Volladdierer aber **zuerst selbst aus Gattern**, damit du verstehst, wie er funktioniert. Den fertigen Baustein kannst du danach zum Vergleichen und für größere Schaltungen verwenden.

Kontrolle mit Python

Die Funktion `volladdierer` bildet die Schaltung Gatter für Gatter nach. Wie beim Halbaddierer gibt sie ihre beiden Ergebnisse als Liste zurück.

```
1  def volladdierer(a, b, c_ein):
2      s1 = int(a != b)          # Halbaddierer 1: Summe
3      c1 = int(a and b)         # Halbaddierer 1: Übertrag
4      s = int(s1 != c_ein)      # Halbaddierer 2: Summe
5      c2 = int(s1 and c_ein)    # Halbaddierer 2: Übertrag
6      c_aus = int(c1 or c2)     # ODER-Gatter
7      return [s, c_aus]
8
9  for a in [0, 1]:
10     for b in [0, 1]:
11         for c_ein in [0, 1]:
12             erg = volladdierer(a, b, c_ein)
13             print(f"A={a} B={b} Cein={c_ein} -> {erg}")
```

AUSGABE

```
A=0 B=0 Cein=0 -> [0, 0]
A=0 B=0 Cein=1 -> [1, 0]
A=0 B=1 Cein=0 -> [1, 0]
A=0 B=1 Cein=1 -> [0, 1]
A=1 B=0 Cein=0 -> [1, 0]
A=1 B=0 Cein=1 -> [0, 1]
A=1 B=1 Cein=0 -> [0, 1]
A=1 B=1 Cein=1 -> [1, 1]
```

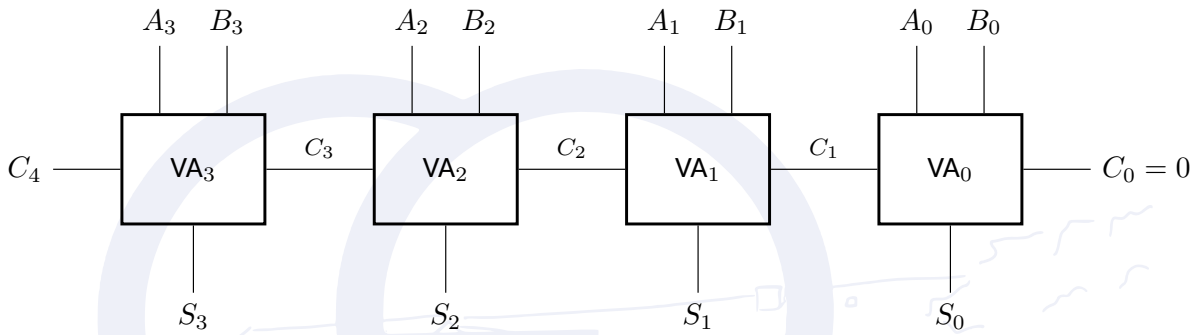
- **Zeilen 2–3:** der erste Halbaddierer (XOR für die Summe, UND für den Übertrag).

- **Zeilen 4–5:** der zweite Halbaddierer. Er bekommt die Zwischensumme `s1` und den eingehenden Übertrag `c_ein`.
- **Zeile 6:** das ODER-Gatter für den ausgehenden Übertrag.
- **Zeilen 9–13:** Drei geschachtelte Schleifen gehen alle acht Belegungen durch. Ausgegeben wird die Liste `[S, Caus]`: vorn die Summe, hinten der ausgehende Übertrag.

2.4.3 Der Carry-Ripple-Addierer

Ein Volladdierer kümmert sich um **eine** Stelle. Für vierstellige Binärzahlen nimmst du vier Volladdierer, einen für jede Stelle. Die Bits der beiden Zahlen nummeriert man von rechts nach links mit 0, 1, 2, 3: A_0 ist das Bit ganz rechts von A, A_3 das Bit ganz links. Der Volladdierer VA_0 ist für die Stelle 0 zuständig, VA_1 für die Stelle 1 und so weiter.

Die Volladdierer werden wie eine Kette verbunden: Der **ausgehende Übertrag** einer Stelle wird zum **eingehenden Übertrag** der nächsthöheren Stelle. Ganz rechts gibt es keinen Übertrag von vorher, deshalb ist dort $C_0 = 0$. Der Übertrag C_4, der ganz links herauskommt, ist die fünfte Stelle des Ergebnisses.



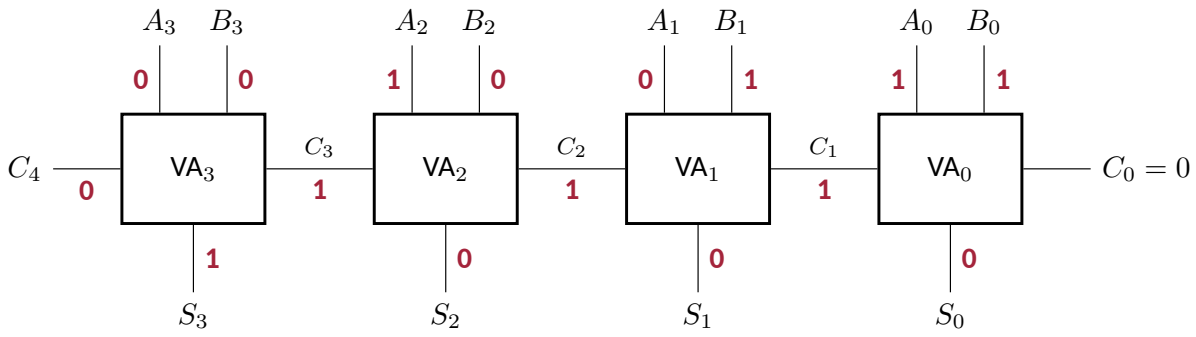
Carry-Ripple-Addierer für vier Bit: Der Übertrag wandert von rechts nach links durch die Volladdierer

Das ist nichts anderes als die schriftliche Addition, nur als Schaltung: Jeder Volladdierer rechnet eine Spalte, und die Leitungen zwischen ihnen sind die Ü-Zeile.

Eine Rechnung durch alle Stufen verfolgen

Wir lassen den Carry-Ripple-Addierer $0101 + 0011$ rechnen. Es ist $A_3A_2A_1A_0 = 0101$ und $B_3B_2B_1B_0 = 0011$. Die Stufen arbeiten von rechts nach links. (Achtung: S_1 bezeichnet hier das Summenbit der Stelle 1, nicht die Zwischensumme im Inneren eines Volladdierers.)

Stufe	A_i	B_i	Übertrag ein	Summe S_i	Übertrag aus
VA_0	1	1	$C_0 = 0$	$S_0 = 0$	$C_1 = 1$
VA_1	0	1	$C_1 = 1$	$S_1 = 0$	$C_2 = 1$
VA_2	1	0	$C_2 = 1$	$S_2 = 0$	$C_3 = 1$
VA_3	0	0	$C_3 = 1$	$S_3 = 1$	$C_4 = 0$



Der Carry-Ripple-Addierer rechnet $0101 + 0011$. Die roten Zahlen sind die Werte auf den Leitungen

Das Ergebnis liest man von links nach rechts ab: $C_4S_3S_2S_1S_0 = 01000$, also 8. Die Rechnung stimmt: $5+3 = 8$.

Achte auf die Spalte „Übertrag ein“: Jede Stufe braucht den Übertrag der Stufe rechts davon. VA$_1$ kann sein endgültiges Ergebnis erst liefern, wenn VA$_0$ fertig ist; VA$_2$ muss auf VA$_1$ warten, und so weiter. Der Übertrag **wandert** wie eine Welle von rechts nach links durch die Kette. Daher kommt der Name: Englisch *carry* heißt Übertrag, *ripple* heißt „rieseln“ oder „sich wellenförmig ausbreiten“.

Das hat einen Nachteil: Je mehr Stellen, desto länger dauert es, bis der Übertrag ganz links angekommen ist. Ein Computer, der 64-stellige Binärzahlen addiert, müsste im schlimmsten Fall warten, bis der Übertrag durch alle 64 Volladdierer gewandert ist. Echte Prozessoren verwenden deshalb geschicktere, aber auch aufwendigere Addierschaltungen.

Merke

Ein **Carry-Ripple-Addierer** verkettet mehrere Volladdierer: Der ausgehende Übertrag C_{aus} einer Stelle wird zum eingehenden Übertrag C_{ein} der nächsthöheren Stelle. So werden mehrstellige Binärzahlen addiert. Der Übertrag „rieselt“ (englisch *ripple*) von Stelle zu Stelle durch.

So kannst du es sagen

So erläuterst du den Carry-Ripple-Addierer:

- *„Da sind vier Volladdierer nebeneinander.“* → *„Ein 4-Bit-Carry-Ripple-Addierer besteht aus vier hintereinandergeschalteten Volladdierern, einer für jede Stelle.“*
- *„Der Übertrag geht weiter.“* → *„Der ausgehende Übertrag jedes Volladdierers liegt am Übertragseingang des Volladdierers der nächsthöheren Stelle an.“*

Arbeitsauftrag 41: Carry-Ripple-Addierer skizzieren und erläutern

Skizziere das **Blockschaltbild** eines 4-Bit-Carry-Ripple-Addierers: vier Volladdierer-Blöcke nebeneinander mit den Eingängen A_0 bis A_3 und B_0 bis B_3, den Summenausgängen S_0 bis S_3 sowie Pfeilen, die zeigen, wie der ausgehende Übertrag jeder Stelle zum eingehenden Übertrag der nächsten Stelle wird. Erläutere in ein bis zwei Sätzen, warum die Bezeichnung „Ripple“ (rieseln) passend ist.

Deine Skizze und Erläuterung:

Arbeitsauftrag 42: Eine Addition verfolgen

Verfolge die Addition $1011 + 0110$ durch alle Stufen eines 4-Bit-Carry-Ripple-Addierers. Fülle die Tabelle aus, lies das Ergebnis ab und mache die Probe im Dezimalsystem.

Stufe	A_i	B_i	Übertrag ein	Summe S_i	Übertrag aus
VA$_0$					
VA$_1$					
VA$_2$					
VA$_3$					

Ergebnis und Probe:

Arbeitsauftrag 43: 4-Bit-Carry-Ripple-Addierer in Python

Schreibe eine Funktion `carry_ripple_addierer(a_bits, b_bits)`. Sie bekommt zwei 4-Bit-Zahlen als Listen mit vier Bits, zum Beispiel `[0, 1, 0, 1]` für 0101. Sie soll die Zahlen mithilfe der Funktion `volladdierer` (siehe oben) **stellenweise von rechts nach links** addieren und das Ergebnis als Liste mit fünf Bits zurückgeben (einschließlich des Übertrags ganz links).

Test: `carry_ripple_addierer([0, 1, 0, 1], [0, 0, 1, 1])` soll `[0, 1, 0, 0, 0]` liefern, denn $5 + 3 = 8$.

Tipp

Der eingehende Übertrag der niedrigsten Stelle ist 0. Verarbeite die Listen von hinten nach vorn: Das Bit ganz rechts hat den Index 3, das Bit ganz links den Index 0. Eine Schleife, die rückwärts zählt, kennst du: `for i in range(3, -1, -1)`. Mit `liste.insert(0, wert)` fügst du einen Wert vorn in eine Liste ein.

Arbeitsauftrag 44 (Zusatz): 2-Bit-Carry-Ripple-Addierer in CircuitVerse

Baue in CircuitVerse einen **2-Bit-Carry-Ripple-Addierer** aus zwei Volladdierern (oder aus einem Halbaddierer für die niedrigste Stelle und einem Volladdierer). Teste ihn mit mehreren Zahlenpaaren und notiere die Ergebnisse.

Getestete Zahlenpaare und Ergebnisse:

2.4.4 Weitere Übungen

Hier kannst du das Gelernte weiter üben. Die Punkte im Titel zeigen, wie schwer eine Aufgabe ist: ●○○ leicht, ●●○ mittel, ●●● knifflig. Fang mit den leichten an.

Arbeitsauftrag 45: Was liefert der Volladdierer? ●○○

Bestimme für jede Belegung die Summe S und den ausgehenden Übertrag C_{aus}. Addiere dazu die drei Bits wie bei der schriftlichen Addition.

A	B	C_{ein}	S	C_{aus}
1	0	0		
0	1	1		
1	1	0		
0	0	1		
1	1	1		

Arbeitsauftrag 46: Richtig oder falsch? ●○○

Entscheide, ob die Aussage richtig oder falsch ist. Verbessere die falschen Aussagen.

a) Ein Volladdierer hat drei Eingänge und zwei Ausgänge.
b) Ein Volladdierer besteht aus zwei Halbaddierern und einem UND-Gatter.
c) Beim Carry-Ripple-Addierer wird der ausgehende Übertrag einer Stelle zum eingehenden Übertrag der nächsthöheren Stelle.
d) In der Stelle ganz rechts ist der eingehende Übertrag immer 1.
e) Die Summe zweier vierstelliger Binärzahlen kann fünf Stellen haben.

Deine Antwort:

Arbeitsauftrag 47: Blick ins Innere ●○○

An einem Volladdierer liegt $A = 1$, $B = 0$ und $C_{\text{ein}} = 1$ an. Bestimme alle Zwischenwerte. Die Abbildung des Volladdierers aus zwei Halbaddierern weiter vorn in diesem Abschnitt hilft dir.

Wert	so wird er berechnet	Ergebnis
S_1	$A \oplus B$ (Halbaddierer 1)	
C_1	$A \wedge B$ (Halbaddierer 1)	
S	$S_1 \oplus C_{\text{ein}}$ (Halbaddierer 2)	
C_2	$S_1 \wedge C_{\text{ein}}$ (Halbaddierer 2)	
C_{aus}	$C_1 \vee C_2$ (ODER-Gatter)	

Arbeitsauftrag 48: Bausteine zählen ●○○

Ein Carry-Ripple-Addierer soll zwei **achtstellige** Binärzahlen addieren. Alle Stellen werden mit Volladdierern gebaut.

1. Wie viele Volladdierer braucht man?
2. Wie viele Stellen kann das Ergebnis höchstens haben?
3. Wie viele Gatter stecken in einem Volladdierer, wenn man ihn aus zwei Halbaddierern und einem ODER-Gatter baut? Wie viele sind es dann im ganzen Addierer?

Deine Antwort:

Gymnasium Ottweiler

Arbeitsauftrag 49: Eine Addition verfolgen ●●○

Verfolge die Addition $0110 + 0111$ durch alle Stufen eines 4-Bit-Carry-Ripple-Addierers. Fülle die Tabelle aus und lies das Ergebnis ab. Zur Probe: $6 + 7 = 13$.

Stufe	A_i	B_i	Übertrag ein	Summe S_i	Übertrag aus
VA_0					
VA_1					
VA_2					
VA_3					

Ergebnis:

Arbeitsauftrag 50: Ein 3-Bit-Addierer ●●○

Skizziere das Blockschaltbild eines **3-Bit**-Carry-Ripple-Addierers. Verwende für die Stelle ganz rechts einen Halbaddierer und für die anderen Stellen Volladdierer. Beschrifte alle Ein- und Ausgänge. Gib an, welche Ausgänge zusammen das Ergebnis bilden.

Deine Skizze:

Gymnasium Ottweiler

Arbeitsauftrag 51: Acht Bit mit Python ●●○

Nimm die Funktionen `volladdierer` und `carry_ripple_addierer` aus der Lösung zum Auftrag *4-Bit-Carry-Ripple-Addierer in Python*.

1. Teste sie mit zwei **achtstelligen** Binärzahlen: $00010110 + 00001111$ (also $22 + 15$) und $11111111 + 00000001$ (also $255 + 1$).
2. Erkläre, warum die Funktion ohne Änderung auch für acht Stellen funktioniert.

Deine Erklärung zu 2.:

Arbeitsauftrag 52: Wie weit rieselt der Übertrag? ●●●

1. Bestimme für jede Addition die Überträge C_1 bis C_4 zwischen den Stufen und das Ergebnis.

Rechnung	C_1	C_2	C_3	C_4	Ergebnis
$0101 + 0010$					
$0111 + 0001$					
$1010 + 0110$					

2. Gib eine Addition zweier vierstelliger Binärzahlen an, bei der **alle** vier Überträge C_1 bis C_4 gleich 1 sind.
3. Erkläre, warum ein Carry-Ripple-Addierer bei deiner Addition aus 2. am längsten braucht.

Deine Antworten zu 2. und 3.:

2.4.5 Das Wichtigste in Kürze

- Beim Addieren mehrstelliger Binärzahlen werden ab der zweiten Stelle **drei** Bits addiert: A, B und der eingehende Übertrag.
- Ein **Volladdierer** hat drei Eingänge (A, B, C_{ein}) und zwei Ausgänge (Summe S, Übertrag C_{aus}).
- Aufbau: zwei Halbaddierer und ein ODER-Gatter. Halbaddierer 1 addiert A und B, Halbaddierer 2 addiert die Zwischensumme und C_{ein}, das ODER-Gatter verknüpft die beiden Überträge.
- Die Belegungstabelle des Volladdierers hat 8 Zeilen. S ist 1 bei einer ungeraden Anzahl von Einsen, C_{aus} ist 1, wenn mindestens zwei Eingänge 1 sind.
- Ein **Carry-Ripple-Addierer** verkettet Volladdierer: Der ausgehende Übertrag jeder Stelle ist der eingehende Übertrag der nächsthöheren Stelle. Ganz rechts ist $C_0 = 0$.
- Der Übertrag wandert Stufe für Stufe von rechts nach links. Bei vielen Stellen dauert die Addition deshalb länger.

Selbstcheck

- ☐ Ich kann erklären, warum man für mehrstellige Additionen einen Volladdierer braucht.
- ☐ Ich kann den Volladdierer aus zwei Halbaddierern und einem ODER-Gatter skizzieren.
- ☐ Ich kann die Belegungstabelle des Volladdierers angeben und an einer Zeile seine Funktionsweise erläutern.
- ☐ Ich kann einen 4-Bit-Carry-Ripple-Addierer skizzieren.
- ☐ Ich kann eine Addition Stufe für Stufe durch einen Carry-Ripple-Addierer verfolgen.
- ☐ Ich kann erklären, warum der Carry-Ripple-Addierer so heißt.

2.5 Rückkopplung: Das Flipflop speichert ein Bit

Leitidee

Stell dir die Alarmanlage aus dem Abschnitt über Schaltungsentwurf vor: Ein Einbrecher öffnet die Tür, und der Alarm geht los. Dann schließt er die Tür wieder – und der Alarm hört sofort auf. Das ist keine gute Alarmanlage! Die Schaltung hat kein **Gedächtnis**: Sie reagiert nur auf das, was gerade jetzt an den Eingängen anliegt. Ein Computer muss sich aber ständig etwas merken: Zahlen, Buchstaben, Zwischenergebnisse. In diesem Abschnitt lernst du eine Schaltung kennen, die sich ein Bit merken kann: das **Flipflop**. Der Trick dahinter heißt **Rückkopplung**.

Lernziele

Nach diesem Abschnitt kannst du

- erklären, worin sich eine Schaltung ohne Gedächtnis von einer speichernden Schaltung unterscheidet,
- das Prinzip der Rückkopplung erläutern,
- ein SR-Flipflop aus zwei NOR-Gattern skizzieren,
- die Zustände Setzen, Rücksetzen, Speichern und den verbotenen Zustand beschreiben,
- Schritt für Schritt nachvollziehen, wie die Rückkopplung einen Wert festhält,
- ein SR-Flipflop in CircuitVerse bauen und testen.

2.5.1 Schaltungen ohne Gedächtnis

Alle Schaltungen, die du bisher gebaut hast, funktionieren nach demselben Prinzip: Die Eingangswerte laufen von links nach rechts durch die Gatter, und am Ende kommt ein Ausgangswert heraus. Die Alarmanlage zum Beispiel berechnet

$$\text{Alarm} = (\text{Tür} \wedge \text{Scharf}) \vee \text{Rauch}.$$

Welcher Wert am Ausgang anliegt, hängt **nur** davon ab, welche Werte **jetzt gerade** an den Eingängen anliegen. Was vor einer Sekunde war, spielt keine Rolle. Genauso ist es beim Halbaddierer und beim Volladdierer: Sie rechnen mit den Bits, die gerade anliegen. Ändert sich ein Eingang, ändert sich sofort auch das Ergebnis.

Definition: Kombinatorische Schaltung

Eine **kombinatorische Schaltung** ist eine Schaltung, deren Ausgangswerte nur von den **aktuellen** Eingangswerten abhängen. Zu jeder Kombination von Eingangswerten gehört immer derselbe Ausgangswert.

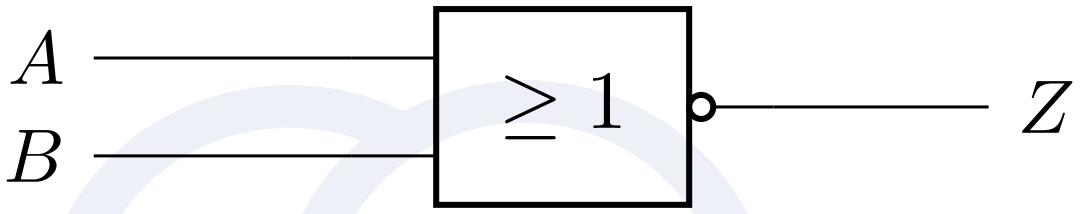
Für die Alarmanlage heißt das: Sobald die Tür wieder zu ist (Tür = 0) und kein Rauch da ist, ist auch der Alarm aus. Wir wollen aber eine Anlage, die **weiter heult**, bis jemand sie mit einer Taste ausschaltet. Der Ausgang soll also auch davon abhängen, was **vorher** passiert ist – von der **Vorgeschichte**. Dafür brauchen wir eine Schaltung, die sich etwas merken kann.

Merke

- Bisherige Schaltungen (Addierer usw.) sind **kombinatorisch**: Der Ausgang hängt nur vom **aktuellen** Eingang ab.
- Ein **Flipflop** kann sich dagegen ein Bit **merken**: Der Ausgang hängt auch von der **Vorgeschichte** ab. Flipflops sind die Grundlage für Speicher im Computer, zum Beispiel für die Register im Prozessor.

2.5.2 Ein neues Gatter: das NOR-Gatter

Für das Flipflop brauchst du ein Gatter, das du noch nicht kennst: das **NOR-Gatter**. Der Name ist eine Abkürzung für „NOT OR“, also NICHT-ODER: Das Gatter rechnet zuerst ein ODER und dreht das Ergebnis dann um. Im IEC-Symbol steht deshalb ≥ 1 wie beim ODER-Gatter, und am Ausgang sitzt ein kleiner Kreis. Dieser Kreis bedeutet „umdrehen“. In manchen Büchern steht an seiner Stelle ein kleiner Schrägstrich. Das bedeutet dasselbe.



Das NOR-Gatter (IEC-Symbol): ein ODER-Gatter mit dem Zeichen für „umdrehen“ am Ausgang

A	B	$A \vee B$	**NOR:** $\neg(A \vee B)$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

Für das Flipflop ist eine Eigenschaft besonders wichtig. Lies sie direkt aus der Tabelle ab:

- Liegt an **einem** Eingang eine 1, ist der Ausgang **sicher 0** – egal, was am anderen Eingang anliegt.
- Nur wenn **beide** Eingänge 0 sind, ist der Ausgang 1. Dann gibt das Gatter sozusagen die Umkehrung des anderen Eingangs weiter.

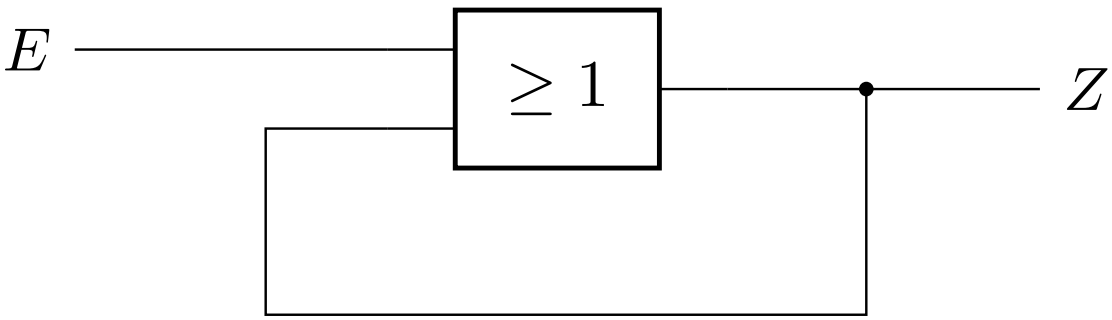
Tipp

Merke dir für die folgenden Abschnitte den kurzen Satz: **„Eine 1 am Eingang macht den NOR-Ausgang zu 0.“** Damit kannst du jedes Flipflop-Rätsel lösen.

2.5.3 Die Idee: Rückkopplung

Bisher liefen alle Leitungen nur in eine Richtung: von den Eingängen zum Ausgang. Jetzt machen wir etwas Neues: Wir führen den Ausgang eines Gatters **zurück** auf einen seiner eigenen Eingänge. Das nennt man **Rückkopplung**.

Ein ganz einfaches Beispiel ist ein ODER-Gatter, dessen Ausgang Z auf den zweiten Eingang zurückgeführt wird:



Ein ODER-Gatter mit Rückkopplung: Der Ausgang Z wird auf den unteren Eingang zurückgeführt

Wir nehmen an, dass am Anfang $E = 0$ und $Z = 0$ ist. Dann passiert Folgendes:

1. Solange $E = 0$ ist, rechnet das Gatter $0 \vee 0 = 0$. Der Ausgang bleibt 0.
2. Jetzt wird E kurz auf 1 gestellt. Das Gatter rechnet $1 \vee 0 = 1$, also wird $Z = 1$.
3. Diese 1 läuft über die Rückkopplung zurück an den unteren Eingang.
4. Nun wird E wieder auf 0 gestellt. Das Gatter rechnet $0 \vee 1 = 1$. Der Ausgang **bleibt 1**!

Die Schaltung hat sich gemerkt, dass E einmal 1 war. Der Ausgang hält sich über die Rückkopplung selbst fest. Leider hat diese Schaltung einen Haken: Die 1 lässt sich nie wieder löschen. Für einen brauchbaren Speicher brauchen wir also noch einen zweiten Eingang, mit dem man den Wert wieder auf 0 setzen kann. Genau das leistet das Flipflop, das du gleich kennenlernst.

Definition: Rückkopplung

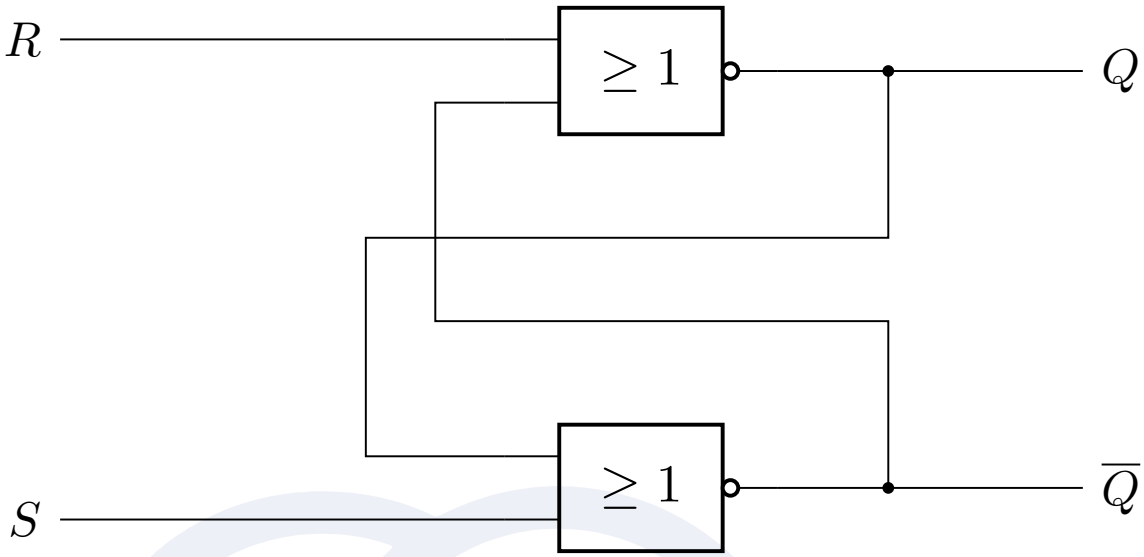
Bei einer **Rückkopplung** wird der Ausgang einer Schaltung auf einen ihrer eigenen Eingänge zurückgeführt. Dadurch hängt der Ausgang auch von seinem eigenen früheren Wert ab.

Merke

Möglich wird das Speichern durch **Rückkopplung**: Der Ausgang einer Schaltung wird auf einen ihrer eigenen Eingänge zurückgeführt.

2.5.4 Das SR-Flipflop aus zwei NOR-Gattern

Das einfachste Flipflop besteht aus nur zwei NOR-Gattern. Der Ausgang jedes Gatters wird an einen Eingang des **anderen** Gatters zurückgeführt. Man sagt: Die Gatter sind **über Kreuz** zurückgekoppelt.



SR-Flipflop aus zwei NOR-Gattern: Die Ausgänge sind über Kreuz zurückgekoppelt

Schau dir die Abbildung in Ruhe an und verfolge jede Leitung mit dem Finger:

- Das **obere** Gatter bekommt den Eingang R und den Ausgang $\overline{Q}$ des unteren Gatters. Sein eigener Ausgang heißt Q.
- Das **untere** Gatter bekommt den Eingang S und den Ausgang Q des oberen Gatters. Sein eigener Ausgang heißt $\overline{Q}$.
- Wo eine Leitung sich verzweigt, sitzt ein Punkt. Wo sich zwei Leitungen nur kreuzen, gibt es keinen Punkt.

Die Namen kommen aus dem Englischen:

Name	englisch	Bedeutung
S	*Set* (setzen)	setzt den gespeicherten Wert auf 1
R	*Reset* (zurücksetzen)	setzt den gespeicherten Wert auf 0
Q	–	der gespeicherte Wert, also das Bit, das sich das Flipflop merkt
$\overline{Q}$	–	gesprochen „Q quer“; im Normalfall das Gegenteil von Q

In CircuitVerse und vielen Büchern schreibt man statt $\overline{Q}$ auch `Q'`.

Definition: Flipflop

Ein **Flipflop** ist eine Schaltung mit Rückkopplung, die ein Bit speichern kann. Der gespeicherte Wert bleibt erhalten, bis er über einen Eingang geändert wird.

Die vier Fälle

Je nachdem, welche Werte an S und R anliegen, macht das Flipflop etwas anderes:

S	R	Q danach	Name des Zustands
1	0	1	Setzen
0	1	0	Rücksetzen
0	0	wie vorher	Speichern
1	1	–	verboten

Neu ist die Zeile mit $S = 0$ und $R = 0$. Dort steht kein fester Wert, sondern „wie vorher“. Das gibt es bei einer kombinatorischen Schaltung nie! Das Flipflop behält einfach den Wert, den es zuletzt hatte. Genau das ist der **Speicher-Effekt**.

Merke

Das einfachste Beispiel ist das **SR-Flipflop** (Set-Reset), gebaut aus zwei über Kreuz rückgekoppelten **NOR-Gattern**:

- $S = 1$ (Set) setzt den gespeicherten Wert auf 1.
- $R = 1$ (Reset) setzt den gespeicherten Wert auf 0.
- $S = 0$ und $R = 0$: Der zuletzt gespeicherte Wert **bleibt erhalten**. Das ist der Speicher-Effekt!
- $S = 1$ und $R = 1$ gleichzeitig ist ein **verbotener Zustand** und sollte vermieden werden.

Arbeitsauftrag 53: SR-Flipflop skizzieren

Decke die Abbildung des SR-Flipflops ab. Skizziere das SR-Flipflop dann aus dem Gedächtnis: zwei NOR-Gatter, deren Ausgang jeweils auf einen Eingang des **anderen** Gatters zurückgeführt wird. Beschrifte die freien Eingänge mit S und R und die beiden Ausgänge mit Q und $\overline{Q}$. Vergleiche zum Schluss mit der Abbildung.

Meine Skizze:

Gymnasium Ottweiler

2.5.5 Schritt für Schritt: Was in der Rückkopplung passiert

Warum merkt sich die Schaltung wirklich etwas? Das siehst du, wenn du die Gatter **nacheinander** ausrechnest. In echt geht das blitzschnell, aber eben nicht gleichzeitig: Jedes Gatter reagiert erst, wenn sich einer seiner Eingänge geändert hat.

Ausgangslage: $S = 0$, $R = 0$, $Q = 0$ und $\overline{Q} = 1$. Prüfe, dass das zusammenpasst: Das obere Gatter bekommt $R = 0$ und $\overline{Q} = 1$. Eine 1 am Eingang macht den Ausgang zu 0, also $Q = 0$. ✓ Das untere Gatter bekommt $S = 0$ und $Q = 0$. Beide Eingänge sind 0, also $\overline{Q} = 1$. ✓ Die Schaltung ist **stabil** – nichts ändert sich. Sie speichert die 0.

Jetzt wird S kurz auf 1 gestellt und danach wieder auf 0:

Schritt	S	R	Was passiert?	Q	$\overline{Q}$
0	0	0	Ausgangslage: Das Flipflop speichert 0.	0	1
1	1	0	S wird 1. Das untere Gatter hat eine 1 am Eingang, also wird $\overline{Q} = 0$.	0	0
2	1	0	Das obere Gatter bekommt jetzt $R = 0$ und $\overline{Q} = 0$. Beide sind 0, also wird $Q = 1$.	1	0
3	1	0	Das untere Gatter bekommt $S = 1$ und $Q = 1$: $\overline{Q}$ bleibt 0. Nichts ändert sich mehr.	1	0
4	0	0	S wird wieder 0. Das untere Gatter bekommt aber noch $Q = 1$ über die Rückkopplung: $\overline{Q}$ bleibt 0. Also bleibt auch $Q = 1$.	1	0

Schritt 4 ist der entscheidende: Der Eingang S ist wieder 0, trotzdem bleibt $Q = 1$. Die 1 an Q hält über die Rückkopplung das untere Gatter auf 0 fest, und diese 0 hält wiederum das obere Gatter auf 1. Die beiden Gatter **stützen sich gegenseitig**. Das Flipflop hat sich die 1 gemerkt.

So kannst du es sagen

So beschreibst du das Flipflop in Fachsprache – vom einfachen zum genauen Satz:

- *„Man kann sich damit eine 0 oder 1 merken."* → *„Die Schaltung speichert eine 0 oder eine 1."*
- *„Die Schaltung kann ein Bit speichern."* → *„Die Schaltung ist in der Lage, ein Bit zu speichern."*
- *„Das Ding merkt sich den Wert, weil der Ausgang zurückgeht."* → *„Das Flipflop nutzt die Technik der Rückkopplung und ist so in der Lage, ein Bit zu speichern. Die Rückkopplung bewirkt, dass der Wert erhalten bleibt."*

Arbeitsauftrag 54: Rücksetzen Schritt für Schritt

Das Flipflop speichert jetzt die 1: $S = 0$, $R = 0$, $Q = 1$, $\overline{Q} = 0$. Nun wird R kurz auf 1 gestellt und danach wieder auf 0. Fülle die Tabelle aus. Gehe vor wie im Beispiel oben: Rechne immer nur **ein** Gatter aus und beschreibe kurz, was passiert.

Schritt	S	R	Was passiert?	Q	$\overline{Q}$
0	0	0	Ausgangslage: Das Flipflop speichert 1.	1	0
1	0	1			
2	0	1			
3	0	1			
4	0	0			

Der verbotene Zustand

Was passiert, wenn $S = 1$ **und** $R = 1$ gleichzeitig anliegen? Beide Gatter haben dann eine 1 am Eingang, also sind **beide** Ausgänge 0: $Q = 0$ und $\overline{Q} = 0$. Das widerspricht der Idee, dass $\overline{Q}$ das Gegenteil von Q ist. Findest du also in einer Tabelle $Q = 0$ und $\overline{Q} = 0$, dann liegt (oder lag gerade) $S = R = 1$ an.

Schlimmer ist, was danach kommt: Werden S und R **gleichzeitig** wieder 0, sehen beide Gatter nur noch Nullen und wollen beide auf 1 springen. Welches Gatter „gewinnt“, hängt von winzigen Unterschieden in der Geschwindigkeit der Bauteile ab. Was danach gespeichert ist, lässt sich nicht vorhersagen. Deshalb nennt man $S = R = 1$ den **verbotenen Zustand**. Eine Schaltung, die ein SR-Flipflop benutzt, muss dafür sorgen, dass er nie vorkommt.

2.5.6 Das Flipflop in Python nachbilden

Eine Python-Funktion, wie du sie im Kapitel über Programmierung kennengelernt hast, hat kein Gedächtnis: Zu denselben Argumenten liefert sie immer dasselbe Ergebnis – genau wie eine kombinatorische Schaltung. Ein Flipflop lässt sich deshalb nicht einfach als Funktion von S und R schreiben. Man kann den Speicher-Effekt aber nachbilden, wenn man den alten Wert von Q in einer Variablen aufbewahrt und ihn der Funktion jedes Mal wieder mitgibt:

```python
def sr_flipflop(s, r, q):
    if s == 1 and r == 0:
        return 1
    elif s == 0 and r == 1:
        return 0
    elif s == 0 and r == 0:
        return q
    else:
        print("Achtung: S = 1 und R = 1 ist der verbotene Zustand!")
        return 0

s_folge = [1, 0, 0, 0]
r_folge = [0, 0, 1, 0]
q = 0
for i in range(4):
    q = sr_flipflop(s_folge[i], r_folge[i], q)
    print(f"S={s_folge[i]} R={r_folge[i]} -> Q={q}")
```

AUSGABE

```
S=1 R=0 -> Q=1
S=0 R=0 -> Q=1
S=0 R=1 -> Q=0
S=0 R=0 -> Q=0
```

- **Zeilen 1–10:** Die Funktion `sr_flipflop` bekommt die Eingänge `s` und `r` und den **bisher gespeicherten** Wert `q`. Sie gibt den neuen Wert von Q zurück.
- **Zeilen 2–5:** Setzen liefert 1, Rücksetzen liefert 0.
- **Zeilen 6–7:** Speichern: Der alte Wert `q` wird unverändert zurückgegeben. Das ist der Speicher-Effekt.
- **Zeilen 8–10:** Bleibt nur noch $S = R = 1$ übrig. Die Funktion gibt eine Warnung aus und liefert 0, denn im verbotenen Zustand ist $Q = 0$.
- **Zeilen 12–13:** Die beiden Listen enthalten die Testreihenfolge: Setzen, Speichern, Rücksetzen, Speichern. An derselben Stelle `i` stehen jeweils die Werte von S und R, die zusammengehören.
- **Zeile 14:** Am Anfang speichert das Flipflop 0.
- **Zeilen 15–17:** Für jeden Schritt wird der neue Wert berechnet und **in derselben Variablen `q` gespeichert**. Im nächsten Schritt wird er der Funktion wieder mitgegeben. Die Variable `q` übernimmt also die Rolle der Rückkopplung.

Wichtig: Das Programm simuliert nur das **Verhalten** des Flipflops, nicht die einzelnen Gatter. Das Gedächtnis steckt hier in der Variablen `q`, in der echten Schaltung dagegen in der Rückkopplung.

Versuch: Probiere es aus

Ändere die beiden Listen so, dass einmal $S = 1$ und $R = 1$ gleichzeitig vorkommt. Was gibt das Programm aus? Setze danach zweimal hintereinander $S = 1$, $R = 0$. Ändert sich beim zweiten Mal etwas?

Arbeitsauftrag 55: Zustandsfolge vorhersagen

Das Flipflop speichert am Anfang die 0. Dann liegen nacheinander diese Werte an:

Schritt	1	2	3	4	5	6	7	8
S	0	1	0	1	0	0	0	1
R	0	0	0	0	1	0	1	0
Q danach								

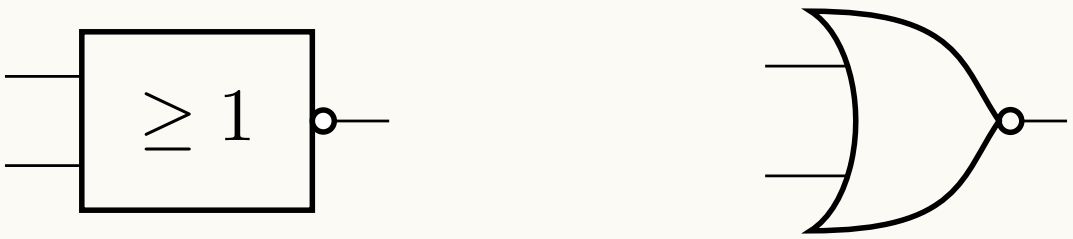
1. Trage ein, welchen Wert Q nach jedem Schritt hat.
2. Überprüfe deine Lösung mit dem Python-Programm oben. Ändere dazu die beiden Listen und die Anzahl der Schritte.

2.5.7 Das Flipflop in CircuitVerse bauen

Jetzt baust du das SR-Flipflop im Simulator CircuitVerse und testest den Speicher-Effekt selbst.

Tipp

CircuitVerse zeigt die US-Symbole für Gatter. Das NOR-Gatter sieht dort aus wie das gebogene ODER-Symbol mit einem kleinen Kreis an der Spitze:



Das NOR-Gatter: links das IEC-Symbol (Skript), rechts das US-Symbol (CircuitVerse)

Falls du kein NOR-Gatter findest, kannst du ein ODER-Gatter mit einem nachgeschalteten NICHT-Gatter nehmen. Das ergibt dasselbe.

So gehst du vor:

1. Lege zwei Eingänge (Schalter) S und R sowie zwei Ausgänge (LEDs) Q und `Q'` an.
2. Füge **zwei NOR-Gatter** hinzu.
3. Verschalte sie **über Kreuz**:
   - Das erste NOR-Gatter bekommt R und den Ausgang des **zweiten** NOR-Gatters.
   - Das zweite NOR-Gatter bekommt S und den Ausgang des **ersten** NOR-Gatters.
4. Der Ausgang des ersten Gatters ist Q, der Ausgang des zweiten ist `Q'`.
5. Teste die Speicherwirkung: Setze kurz $S = 1$, $R = 0$. Dann wird Q zu 1. Schalte danach S wieder auf 0, also $S = 0$, $R = 0$. Beobachte: Q bleibt 1, obwohl kein Eingang mehr aktiv ist!

Stolperstein

- **S und R vertauscht:** Wenn bei dir $S = 1$ den Ausgang Q auf 0 setzt, hast du die Eingänge vertauscht. R gehört an das Gatter, dessen Ausgang Q ist. (Merkhilfe: Das Gatter mit R liefert Q. $R = 1$ macht dieses Gatter zu 0, also setzt R das Q auf 0 zurück.)
- **Unklarer Anfang:** Direkt nach dem Aufbau ist manchmal nicht klar, was gespeichert ist. Vielleicht sind sogar beide Ausgänge gleich. Setze dann einmal S auf 1 und wieder auf 0. Danach ist alles eindeutig.

Arbeitsauftrag 56: Zustände protokollieren

Führe die Testreihenfolge in deiner CircuitVerse-Schaltung durch. Trage jeweils den beobachteten Wert von Q ein.

Schritt	S	R	Q (beobachtet)	Erklärung
1	1	0		Setzen
2	0	0		Speichern (Q bleibt wie zuvor)
3	0	1		Rücksetzen
4	0	0		Speichern (Q bleibt wie zuvor)

2.5.8 Wozu braucht man Flipflops?

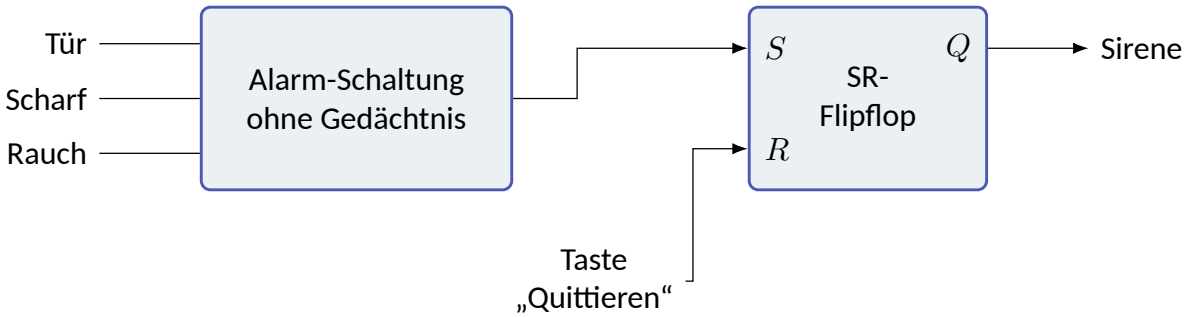
Ein Speicher für ein Bit

Ein Flipflop speichert genau **ein Bit**. Setzt man acht Flipflops nebeneinander, kann man ein ganzes Byte speichern, zum Beispiel eine Zahl von 0 bis 255 oder einen Buchstaben. So sind die **Register** im Prozessor aufgebaut: kleine, sehr schnelle Speicher, in denen der Prozessor die Zahlen ablegt, mit denen er gerade rechnet. Auch die schnellen Zwischenspeicher im Prozessor (der *Cache*) bestehen aus Speicherzellen, die wie Flipflops funktionieren.

Der große Arbeitsspeicher (RAM) eines PCs arbeitet meist mit einer anderen Technik: Dort wird jedes Bit als winzige elektrische Ladung gespeichert. Das Prinzip ist aber dasselbe: Eine Speicherzelle merkt sich eine 0 oder eine 1, bis sie überschrieben wird.

Eine Alarmanlage mit Gedächtnis

Jetzt kannst du das Problem vom Anfang lösen. Der Ausgang der bisherigen Alarm-Schaltung wird an den Eingang S eines Flipflops angeschlossen. An R kommt eine Taste „Quittieren“, mit der man den Alarm ausschaltet. Die Sirene hängt an Q.



Alarmanlage mit Gedächtnis: Die Alarm-Schaltung setzt das Flipflop, die Taste setzt es zurück

- Öffnet der Einbrecher die Tür, liefert die Alarm-Schaltung eine 1. Das Flipflop wird **gesetzt**, die Sirene heult.
- Schließt er die Tür, liegt an S wieder 0 an. Mit $R = 0$ wird **gespeichert**: Die Sirene heult weiter.
- Erst wenn jemand die Taste drückt ($R = 1$), wird das Flipflop **zurückgesetzt** und die Sirene verstummt.

Arbeitsauftrag 57: Warum Rückkopplung?

Beantworte in 2–3 Sätzen: Warum kann eine **rein kombinatorische** Schaltung (ohne Rückkopplung) kein Bit speichern? Was genau ermöglicht die Rückkopplung, das eine gewöhnliche Gatterkette nicht kann?

Meine Erklärung:

Arbeitsauftrag 58 (Zusatz): Andere Flipflops

Recherchiere (zum Beispiel im Internet) den Unterschied zwischen dem SR-Flipflop und einem **D-Flip-flop** oder einem **JK-Flipflop**. Welchen Vorteil haben sie gegenüber dem einfachen SR-Flipflop?

Meine Ergebnisse:

2.5.9 Weitere Übungen

Hier kannst du das Gelernte weiter üben. Die Punkte im Titel zeigen, wie schwer eine Aufgabe ist: ●○○ leicht, ●●○ mittel, ●●● knifflig. Fang mit den leichten an.

Arbeitsauftrag 59: Mit oder ohne Gedächtnis? ●○○

Entscheide bei jedem Beispiel, ob es sich wie eine kombinatorische Schaltung verhält (ohne Gedächtnis) oder ob es sich etwas merkt. Begründe kurz.

a) der Halbaddierer
b) die Alarmanlage $\text{Alarm} = (\text{Tür} \wedge \text{Scharf}) \vee \text{Rauch}$
c) das SR-Flipflop
d) die Haltewunsch-Lampe im Bus: Sie geht an, wenn jemand die Stopp-Taste drückt, und bleibt an, bis der Bus an der Haltestelle die Türen öffnet.
e) eine Türklingel: Sie tönt nur, solange jemand auf den Knopf drückt.

Deine Antwort:

Gymnasium Ottweiler

Arbeitsauftrag 60: Die vier Fälle ●○○

Das SR-Flipflop speichert gerade die 1. Ergänze für jede Zeile den Namen des Zustands und den Wert von Q danach. Wo es keinen sinnvollen Wert gibt, schreibe einen Strich.

S	R	Name des Zustands	Q danach
1	0		
0	1		
0	0		
1	1		

Arbeitsauftrag 61: Eine kurze Folge ●○○

Das Flipflop speichert am Anfang die 1. Dann liegen nacheinander diese Werte an. Trage ein, welchen Wert Q nach jedem Schritt hat.

Schritt	1	2	3	4	5
S	0	0	0	1	0
R	0	1	0	0	0
Q danach					

Arbeitsauftrag 62: Passende Eingänge finden ●●○

Jetzt umgekehrt: Das Flipflop speichert am Anfang die 0. Q soll nach den einzelnen Schritten die Werte in der unteren Zeile haben. Bestimme passende Werte für S und R. Der verbotene Zustand darf nicht vorkommen.

Schritt	1	2	3	4	5	6
S						
R						
Q danach	0	1	1	0	0	1

Arbeitsauftrag 63: Haltewunsch im Bus mit Python ●●○

Die Haltewunsch-Lampe im Bus verhält sich wie ein SR-Flipflop: Die Stopp-Taste ist S, das Öffnen der Türen ist R, die Lampe ist Q. Am Anfang ist die Lampe aus.

Schritt	1	2	3	4	5	6
Taste	0	1	0	1	0	0
Tür	0	0	0	0	1	0

1. Sage voraus, ob die Lampe nach jedem Schritt leuchtet.
2. Prüfe deine Vorhersage mit Python. Verwende die Funktion `sr_flipflop` aus dem Abschnitt *Das Flipflop in Python nachbilden* und zwei Listen `taste` und `tuer`. Gib in jedem Schritt eine Zeile wie `Schritt 2: Taste=1 Tür=0 -> Lampe=1` aus.

Deine Vorhersage:

Arbeitsauftrag 64: Ein rätselhafter Baustein ●●●

Ein Baustein mit den Eingängen X und Y und dem Ausgang Z wurde getestet. Das ist das Protokoll:

Schritt	1	2	3	4	5	6
X	1	0	0	0	1	0
Y	0	0	1	0	0	0
Z	1	1	0	0	1	1

1. Begründe mit dem Protokoll, dass der Baustein **keine** kombinatorische Schaltung sein kann.
2. Prüfe, ob sich der Baustein wie ein SR-Flipflop verhält, wenn X der Eingang S und Y der Eingang R ist.
3. In Schritt 7 liegt $X = 0, Y = 1$ an, in Schritt 8 $X = 0, Y = 0$. Sage Z für beide Schritte voraus.

Deine Antwort:

2.5.10 Das Wichtigste in Kürze

- **Kombinatorische Schaltungen** (z. B. Addierer, Alarmanlage) haben kein Gedächtnis: Der Ausgang hängt nur von den aktuellen Eingängen ab.
- Bei der **Rückkopplung** wird ein Ausgang auf einen Eingang derselben Schaltung zurückgeführt. Dadurch kann eine Schaltung einen Wert festhalten.
- Ein **Flipflop** speichert ein Bit; sein Ausgang hängt auch von der Vorgeschichte ab. Das **SR-Flipflop** besteht aus zwei über Kreuz rückgekoppelten NOR-Gattern.
- $S = 1$ setzt Q auf 1, $R = 1$ setzt Q auf 0, bei $S = R = 0$ wird gespeichert. $S = R = 1$ ist verboten.
- Flipflops sind die Grundlage von Speichern wie den Registern im Prozessor.

Selbstcheck

- ☐ Ich kann erklären, warum eine kombinatorische Schaltung kein Bit speichern kann.
- ☐ Ich kann erklären, was Rückkopplung bedeutet.
- ☐ Ich kann ein SR-Flipflop aus zwei NOR-Gattern skizzieren und richtig beschriften.
- ☐ Ich kenne die Zustände Setzen, Rücksetzen, Speichern und den verbotenen Zustand.
- ☐ Ich kann Schritt für Schritt erklären, warum Q nach dem Setzen erhalten bleibt.
- ☐ Ich kann zu einer Folge von Eingangswerten angeben, welchen Wert Q jeweils hat.

2.6 Aussagenlogik: Schaltungen als Terme beschreiben

Leitidee

Eine Schaltung zu zeichnen dauert lange, und bei großen Schaltungen verliert man schnell den Überblick. Viel kürzer ist eine Formel wie $(\text{Tür} \wedge \text{Scharf}) \vee \text{Rauch}$. Solche Formeln gibt es schon viel länger als Computer: Die **Aussagenlogik** untersucht seit über 2000 Jahren, wie man Aussagen mit „und“, „oder“ und „nicht“ verknüpft. Der Ingenieur Claude Shannon hat erkannt, dass man mit genau diesen Formeln auch Schaltungen beschreiben kann. In diesem Abschnitt lernst du, wie solche Formeln – man nennt sie **Terme** – genau aufgebaut sind, wie man sie als **Baum** zeichnet und wie man zwischen Term und Schaltung hin- und herübersetzt.

Lernziele

Nach diesem Abschnitt kannst du

- die Begriffe Konjunktion, Disjunktion und Negation sachgemäß verwenden und ihre Belegungstabellen angeben,
- den Begriff des aussagenlogischen Terms induktiv definieren,
- entscheiden, ob eine Zeichenfolge ein Term ist, und die Klammerregeln anwenden,
- aussagenlogische Terme als Termbäume darstellen,
- logische Schaltungen durch Terme beschreiben und Terme in Schaltungen übertragen.

2.6.1 Aussagen und Aussagenvariablen

Eine **Aussage** ist ein Satz, der entweder wahr oder falsch ist. „Die Tür ist offen.“ ist eine Aussage. „Mach die Tür zu!“ ist keine Aussage, denn ein Befehl ist weder wahr noch falsch.

In der Aussagenlogik interessiert uns nicht, was eine Aussage genau sagt, sondern nur, ob sie wahr oder falsch ist. Deshalb ersetzt man sie durch einen Buchstaben oder ein kurzes Wort, zum Beispiel Tür für „Die Tür ist offen.“ Eine solche **Aussagenvariable** hat einen von zwei Werten: **1** für wahr, **0** für falsch.

Das passt genau zu den Schaltungen: Eine Leitung führt Strom (1) oder keinen Strom (0). Jeder Eingang einer Schaltung entspricht einer Aussagenvariablen. Wenn wir einer Variablen einen Wert geben, sagt man: Die Variable wird mit einem Wert **belegt**.

Definition: Aussagenvariable

Eine **Aussagenvariable** ist ein Platzhalter für eine Aussage. Sie kann mit einem der beiden Wahrheitswerte 1 (wahr) oder 0 (falsch) belegt werden. Man schreibt sie meist als Großbuchstaben: A, B, C …

Merke

Die **Aussagenlogik** beschreibt dieselben Zusammenhänge wie logische Schaltungen – nur mit Symbolen statt mit Gattern und Leitungen.

2.6.2 Die drei Verknüpfungen: Negation, Konjunktion, Disjunktion

Aus einfachen Aussagen baut man zusammengesetzte Aussagen. Dafür gibt es in der Aussagenlogik drei Zeichen. Du kennst sie schon von den Gattern:

- Die **Negation** $\neg A$ (gesprochen „nicht A“) dreht den Wert um. Sie entspricht dem NICHT-Gatter.
- Die **Konjunktion** $A \wedge B$ (gesprochen „A und B“) ist nur dann 1, wenn **beide** Teile 1 sind. Sie entspricht dem UND-Gatter.
- Die **Disjunktion** $A \vee B$ (gesprochen „A oder B“) ist 1, wenn **mindestens einer** der Teile 1 ist. Sie entspricht dem ODER-Gatter.

A	B	$\neg A$	$A \wedge B$	$A \vee B$
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

Definition: Negation, Konjunktion, Disjunktion

- Die **Negation** $\neg A$ ist genau dann 1, wenn A den Wert 0 hat.
- Die **Konjunktion** $A \wedge B$ ist genau dann 1, wenn A und B beide den Wert 1 haben.
- Die **Disjunktion** $A \vee B$ ist genau dann 1, wenn mindestens eine der beiden Variablen den Wert 1 hat.

Tipp

Eine Merkhilfe für die Zeichen: $\vee$ sieht aus wie ein „v“ – das lateinische Wort *vel* heißt „oder“. Das andere Zeichen $\wedge$ ist dann das „und“. Für das Zeichen $\neg$ stell dir einen Haken vor, der den Wert umdreht.

Die folgende Tabelle fasst zusammen, wie die drei Verknüpfungen in der Aussagenlogik, bei den Gattern und in Python heißen:

Name	Zeichen	gesprochen	Gatter	Python
Negation	$\neg A$	nicht A	NICHT (NOT)	`not a`
Konjunktion	$A \wedge B$	A und B	UND (AND)	`a and b`
Disjunktion	$A \vee B$	A oder B	ODER (OR)	`a or b`

Merke

Entsprechung zu Schaltungen: $\wedge$ (Konjunktion) entspricht dem UND-Gatter, $\vee$ (Disjunktion) dem ODER-Gatter, $\neg$ (Negation) dem NICHT-Gatter.

Stolperstein

Das „oder“ der Aussagenlogik ist **kein** „entweder … oder“. $A \vee B$ ist auch dann 1, wenn **beide** Variablen 1 sind. Im Alltag meinen wir mit „oder“ manchmal „nur eins von beiden“ („Willst du Tee oder Kaffee?“). Das wäre XOR. Das Zeichen $\oplus$ für XOR gehört aber nicht zu den Zeichen der Aussagenlogik, die du gleich kennenlernst. Dass man XOR auch mit $\wedge$, $\vee$ und $\neg$ ausdrücken kann, hast du im Auftrag *Belegungstabelle zu einer Schaltung* gesehen. Ein Verfahren, das immer funktioniert, lernst du im nächsten Abschnitt.

2.6.3 Was ist ein Term? Die induktive Definition

Nicht jede Zeichenfolge ist eine sinnvolle Formel. Mit $A \wedge \vee B$ kann niemand etwas anfangen. Wir brauchen also eine genaue Regel, welche Zeichenfolgen erlaubt sind. Die erlaubten Formeln heißen **aussagenlogische Terme**.

Die Regel funktioniert wie ein Baukasten: Man fängt mit den kleinsten Bausteinen an, den Variablen. Dann gibt es Regeln, wie man aus fertigen Termen größere Terme baut. Jeder Term, den man so gebaut hat, darf wieder als Baustein verwendet werden. Eine solche Definition, die Schritt für Schritt vom Kleinen zum Großen aufbaut, nennt man **induktiv**.

Merke

Induktive Definition eines aussagenlogischen Terms:

1. Jede Aussagenvariable (z. B. A, B, C) ist ein Term.
2. Ist t ein Term, so ist auch $\neg t$ (**Negation**) ein Term.
3. Sind t_1 und t_2 Terme, so sind auch $(t_1 \wedge t_2)$ (**Konjunktion**, entspricht AND) und $(t_1 \vee t_2)$ (**Disjunktion**, entspricht OR) Terme.
4. Nichts anderes ist ein Term.

Die Buchstaben t, t_1 und t_2 stehen hier nicht für Variablen, sondern für **beliebige Terme** – kleine wie A oder große wie $(A \wedge \neg B)$. Regel 4 ist wichtig: Sie sagt, dass es keine anderen Terme gibt als die, die man mit den Regeln 1 bis 3 bauen kann.

Gymnasium Ottweiler

Beispiel

Wir bauen den Term $((A \wedge B) \vee \neg C)$ Schritt für Schritt auf:

Schritt	gebauter Term	Regel	gebaut aus
1	A, B, C	1	–
2	$(A \wedge B)$	3	$t_1 = A, t_2 = B$
3	$\neg C$	2	$t = C$
4	$((A \wedge B) \vee \neg C)$	3	$t_1 = (A \wedge B), t_2 = \neg C$

Weil jeder Schritt eine der Regeln benutzt, ist $((A \wedge B) \vee \neg C)$ ein Term.

Umgekehrt kann man bei einer Zeichenfolge prüfen, ob sie ein Term ist. Diese Zeichenfolgen sind **keine** Terme:

Zeichenfolge	Warum kein Term?
$A\wedge$	Für $\wedge$ braucht man zwei Terme, rechts fehlt einer.
$\vee AB$	Das Zeichen $\vee$ muss **zwischen** zwei Termen stehen.
$(A \vee B$	Die schließende Klammer fehlt.
$A\neg B$	$\neg$ steht vor einem Term, nicht zwischen zwei Termen.
$A + B$	Das Zeichen $+$ kommt in den Regeln nicht vor.

Klammern sparen

Nach der strengen Definition stehen um jede Konjunktion und jede Disjunktion Klammern. Bei großen Termen wird das unübersichtlich. Deshalb gibt es Vereinbarungen, wann man Klammern weglassen darf:

1. **Die äußersten Klammern** dürfen weg: $(A \wedge B) \vee \neg C$ statt $((A \wedge B) \vee \neg C)$.
2. **Das Zeichen $\neg$ bezieht sich nur auf das, was direkt dahinter steht.** $\neg A \wedge B$ bedeutet $(\neg A) \wedge B$. Soll $\neg$ für mehr gelten, braucht man Klammern: $\neg(A \wedge B)$.
3. **Bei lauter $\wedge$ (oder lauter $\vee$)** ist die Klammerung egal, das Ergebnis ist immer dasselbe. Man schreibt einfach $A \wedge B \wedge C$ statt $((A \wedge B) \wedge C)$.
4. **$\wedge$ bindet stärker als $\vee$** – so wie Punkt vor Strich. $A \wedge B \vee C$ bedeutet also $(A \wedge B) \vee C$.

Python hält sich übrigens an dieselbe Reihenfolge: zuerst `not`, dann `and`, dann `or`.

Tipp

Die letzte Vereinbarung ist leicht zu vergessen. Setze deshalb Klammern, sobald $\wedge$ und $\vee$ zusammen vorkommen. Schreibe also lieber $(A \wedge B) \vee C$ als $A \wedge B \vee C$. Zu viele Klammern schaden nie.

Stolperstein

$\neg(A \vee B)$ und $\neg A \vee B$ sind zwei verschiedene Terme! Beim ersten wird das Ergebnis von $A \vee B$ umgedreht. Beim zweiten wird nur A umgedreht. Setze für $A = 0$, $B = 1$ ein: $\neg(0 \vee 1) = \neg 1 = 0$, aber $\neg 0 \vee 1 = 1 \vee 1 = 1$.

Arbeitsauftrag 65: Term oder kein Term?

Entscheide bei jeder Zeichenfolge, ob sie nach der strengen induktiven Definition ein Term ist. Begründe kurz. Gib bei den Termen an, in welcher Reihenfolge sie mit den Regeln aufgebaut werden.

a) $(A \vee \neg B)$
b) $\neg\neg A$
c) $(A \wedge B\vee)$
d) $(\neg(A \vee B) \wedge C)$
e) $\wedge(A, B)$

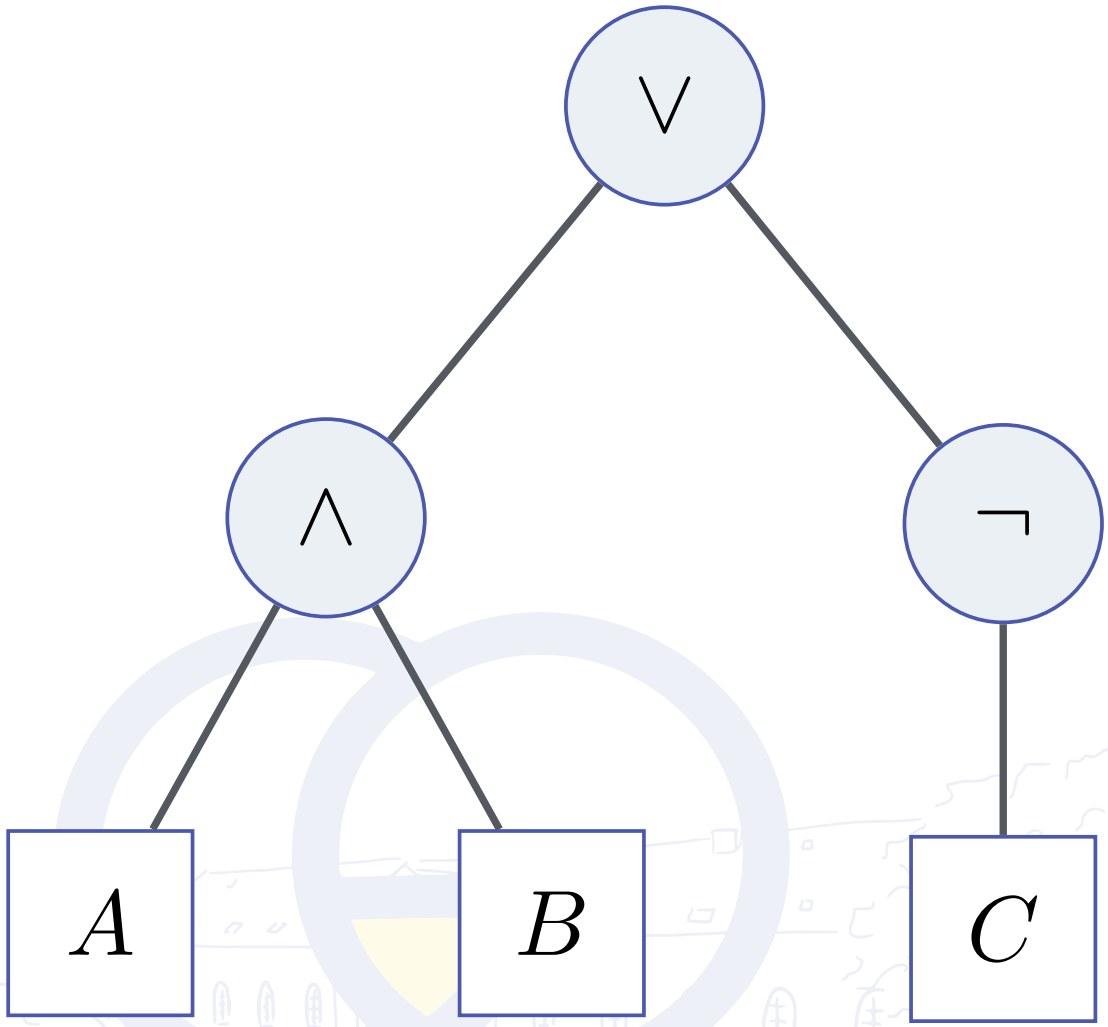
Meine Antworten:

2.6.4 Termbäume

Ein Term ist eine Zeichenkette. Seinen Aufbau sieht man aber besser in einem Bild. Dafür zeichnet man einen **Termbaum**. Er steht – anders als echte Bäume – auf dem Kopf:

- Ganz oben steht die **Wurzel**. Das ist der Operator, der **zuletzt** ausgerechnet wird.
- Unter jedem Operator hängen die Teilterme, mit denen er arbeitet: unter $\wedge$ und $\vee$ zwei, unter $\neg$ einer.
- Ganz unten stehen die **Blätter**. Das sind die Variablen.

Hier ist der Termbaum zum Term $(A \wedge B) \vee \neg C$:



Termbaum zu $(A \wedge B) \vee \neg C$: Der Operator, der zuletzt ausgewertet wird, steht ganz oben (Wurzel)

Vergleiche den Baum mit dem schrittweisen Aufbau im Beispiel oben: Jede Anwendung einer Regel ist ein Knoten im Baum. Die Variablen aus Schritt 1 sind die Blätter. Der letzte Schritt ($\vee$) ist die Wurzel.

Definition: Termbaum

Ein **Termbaum** stellt einen Term als Baumstruktur dar. Die **Blätter** sind die Aussagenvariablen, die **inneren Knoten** die Operatoren ($\neg$, $\wedge$, $\vee$). Der Operator, der zuletzt ausgewertet wird, steht an der Wurzel.

Merke

Ein **Termbaum** stellt einen Term als Baumstruktur dar: Die Blätter sind die Aussagenvariablen, die inneren Knoten die Operatoren ($\neg$, $\wedge$, $\vee$).

So zeichnest du einen Termbaum

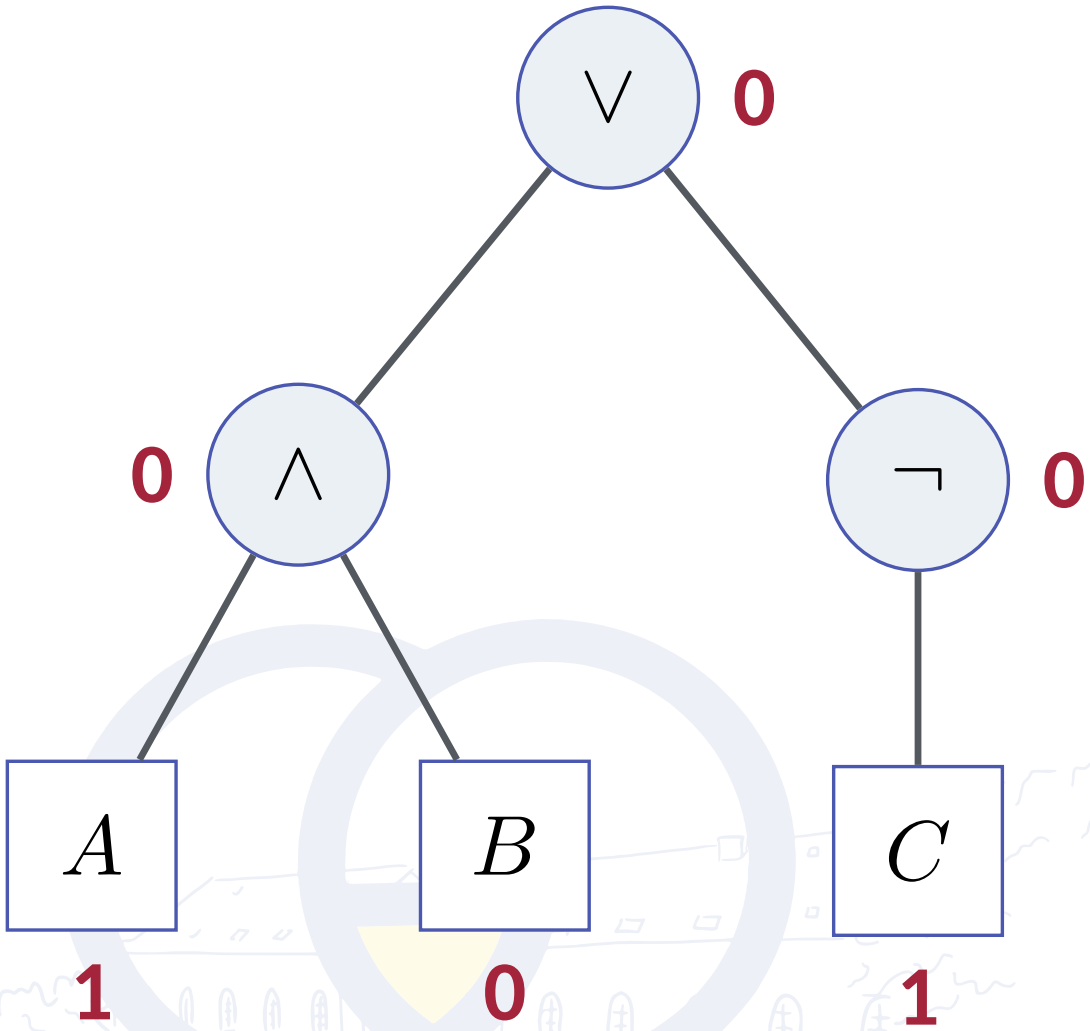
1. **Suche den Operator, der zuletzt ausgerechnet wird.** Das ist der Operator, der **außerhalb aller Klammern** steht. Stehen dort mehrere, nimm den schwächsten ($\vee$ vor $\wedge$ vor $\neg$). Er wird die Wurzel.
2. **Zeichne darunter seine Teilterme.** Bei $\wedge$ und $\vee$ ist das der Teil links und der Teil rechts vom Zeichen. Bei $\neg$ ist es das, was dahinter steht.
3. **Wiederhole** Schritt 1 und 2 für jeden Teilterm, bis nur noch Variablen übrig sind.

Für $(A \wedge B) \vee \neg C$: Außerhalb der Klammer stehen $\vee$ und $\neg$. Das schwächere Zeichen ist $\vee$, also ist $\vee$ die Wurzel. Links davon steht $(A \wedge B)$, rechts $\neg C$. Beide werden auf dieselbe Weise weiter zerlegt.

Einen Term am Baum ausrechnen

Am Termbaum kannst du auch gut ausrechnen, welchen Wert ein Term für eine bestimmte Belegung hat. Du rechnest **von unten nach oben**. Für $A = 1$, $B = 0$, $C = 1$ sieht das so aus (die Werte stehen rot neben den Knoten):

Gymnasium Ottweiler



Auswertung von $(A \wedge B) \vee \neg C$ für $A = 1, B = 0, C = 1$: Die Werte werden von den Blättern zur Wurzel weitergereicht

- $A \wedge B$: $1 \wedge 0 = 0$.
- $\neg C$: $\neg 1 = 0$.
- An der Wurzel: $0 \vee 0 = 0$. Der Term hat also für diese Belegung den Wert 0.

Arbeitsauftrag 66: Vom Term zum Termbaum

Zeichne den Termbaum zum Term

$$\neg(A \vee B) \wedge C.$$

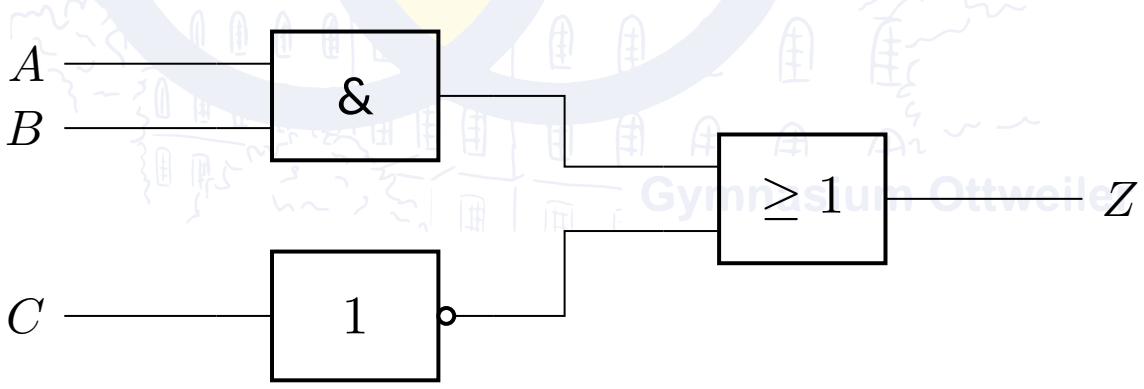
Mein Termbaum:

2.6.5 Vom Term zur Schaltung und zurück

Ein Termbaum und eine Schaltung haben denselben Aufbau. Dreh den Termbaum in Gedanken um 90 Grad, sodass die Wurzel rechts steht:

- Die **Blätter** (Variablen) werden zu den **Eingängen** links.
- Jeder **innere Knoten** wird zu einem **Gatter**: $\wedge$ zum UND-Gatter, $\vee$ zum ODER-Gatter, $\neg$ zum NICHT-Gatter.
- Die **Wurzel** wird zum letzten Gatter. Sein Ausgang ist der Ausgang der Schaltung.

So sieht die Schaltung zum Term $(A \wedge B) \vee \neg C$ aus:



Schaltung zum Term $(A \wedge B) \vee \neg C$: Jeder Knoten des Termbaums ist ein Gatter

Die Schaltung braucht so viele Gatter, wie der Termbaum innere Knoten hat – hier drei. Zum Bauen gehst du **von innen nach außen** vor: zuerst die Gatter für die innersten Teilterme ($A \wedge B$ und $\neg C$), zuletzt das Gatter der Wurzel.

Von der Schaltung zum Term

Der umgekehrte Weg geht genauso. Schreibe an jede Leitung, welchen Teilterm sie führt. Beginne an den Eingängen und arbeite dich nach rechts vor:

1. An den Eingängen stehen die Variablen.
2. Am Ausgang jedes Gatters steht der Teilterm, der aus seinen Eingängen entsteht. Ein UND-Gatter mit den Eingängen A und B liefert zum Beispiel $A \wedge B$.
3. Am Ausgang der Schaltung steht dann der gesamte Term.

Setze dabei um jeden Teilterm, der in ein weiteres Gatter läuft, Klammern. Überflüssige Klammern kannst du am Ende wieder weglassen.

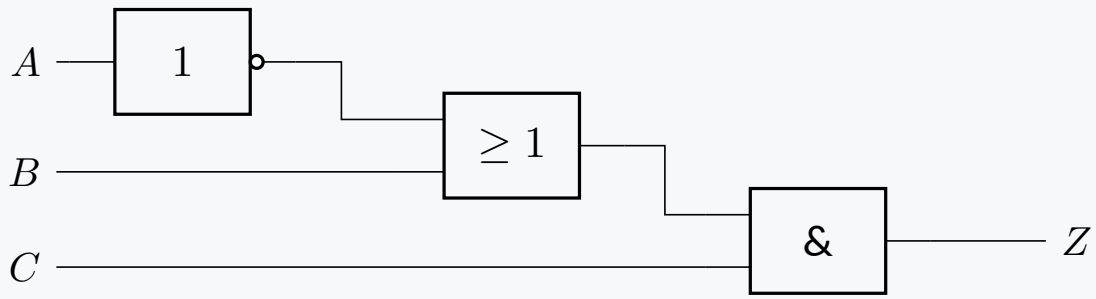
So kannst du es sagen

So sprichst du über Terme und Schaltungen – vom einfachen zum genauen Satz:

- *„Der Baum zeigt, wie der Term aufgebaut ist."* → *„Der Termbaum stellt den logischen Term dar und strukturiert ihn."*
- *„Die Formel ist die Schaltung."* → *„Der logische Term repräsentiert die Schaltung."*
- *„A ist 1 und B ist 0."* → *„Die logischen Variablen werden mit Werten belegt: A mit 1, B mit 0."*

Arbeitsauftrag 67: Schaltung beschreiben

Gib den Term an, den die folgende Schaltung darstellt. Zeichne anschließend den Termbaum.



Eine Schaltung mit drei Eingängen

Term und Termbaum:

Arbeitsauftrag 68: Termbaum der Alarmanlage

Im Abschnitt über Schaltungsentwurf hast du die Alarmanlage gebaut:

$$\text{Alarm} = (\text{Tür} \wedge \text{Scharf}) \vee \text{Rauch}$$

Zeichne den Termbaum zu diesem Term.

Mein Termbaum:

2.6.6 Terme mit Python kontrollieren

Terme lassen sich direkt als Python-Ausdrücke schreiben: `and` für $\wedge$, `or` für $\vee$, `not` für $\neg$. Damit kannst du prüfen, ob du einen Term richtig verstanden hast. Das folgende Programm gibt die Belegungstabelle zum Term $\neg(A \vee B) \wedge C$ aus:

```
1 def term1(a, b, c):
2     return int(not (a or b) and c)
3
4 print("Term 1: not (A or B) and C")
5 for a in [0, 1]:
6     for b in [0, 1]:
7         for c in [0, 1]:
8             print(f"A={a} B={b} C={c} -> {term1(a, b, c)}")
```

AUSGABE

```
Term 1: not (A or B) and C
A=0 B=0 C=0 -> 0
A=0 B=0 C=1 -> 1
A=0 B=1 C=0 -> 0
A=0 B=1 C=1 -> 0
A=1 B=0 C=0 -> 0
A=1 B=0 C=1 -> 0
A=1 B=1 C=0 -> 0
A=1 B=1 C=1 -> 0
```

- **Zeilen 1–2:** Die Funktion `term1` rechnet den Term für eine Belegung aus. Python wertet `not` vor `and` aus – genau wie in der Aussagenlogik. `int(...)` sorgt dafür, dass das Ergebnis als 0 oder 1 erscheint und nicht als `True` oder `False`.
- **Zeilen 5–7:** Drei ineinander geschachtelte Schleifen gehen alle Belegungen durch. Jede Variable läuft über die Liste `[0, 1]`. Die innerste Schleife (für `c`) läuft am schnellsten. So entstehen die $2^3 = 8$ Belegungen in der üblichen Reihenfolge von 000 bis 111.
- **Zeile 8:** Für jede Belegung wird eine Zeile der Tabelle ausgegeben.

Der Term ist nur bei $A = 0, B = 0, C = 1$ wahr. Das passt zur Bedeutung: „Weder A noch B, aber C.“

Arbeitsauftrag 69: Vom Term zur Schaltung

Jetzt geht es in die umgekehrte Richtung. Nimm den Term $\neg(A \vee B) \wedge C$ aus dem Auftrag *Vom Term zum Termbaum*.

1. Skizziere die zugehörige Schaltung mit IEC-Symbolen.
2. Baue die Schaltung in CircuitVerse: ein ODER-Gatter für $A \vee B$, ein NICHT-Gatter dahinter und ein UND-Gatter, das das Ergebnis mit C verknüpft.
3. Teste alle 8 Belegungen. Vergleiche mit deinem Termbaum und mit der Ausgabe des Python-Programms oben.

Meine Skizze:

Arbeitsauftrag 70: Genau eine von drei

Formuliere einen aussagenlogischen Term für folgende Situation: „Der Ausgang ist 1, wenn **genau eine** der drei Variablen A, B, C den Wert 1 hat.“ Notiere den Term symbolisch **und** als Python-Funktion `term3(a, b, c)`. Prüfe deine Funktion mit drei geschachtelten Schleifen.

Tipp: Überlege, in welchen Fällen der Ausgang 1 sein soll. Beschreibe **jeden Fall** mit einer Konjunktion, die genau auf diesen Fall passt. Verbinde dann alle Fälle mit $\vee$.

Mein Term:

Gymnasium Ottweiler

2.6.7 Weitere Übungen

Hier kannst du das Gelernte weiter üben. Die Punkte im Titel zeigen, wie schwer eine Aufgabe ist: ●○○ leicht, ●●○ mittel, ●●● knifflig. Fang mit den leichten an.

Arbeitsauftrag 71: Drei Schreibweisen ●○○

Ergänze die Tabelle.

Name	Zeichen	Gatter	Python
Negation			
	$\wedge$		
		ODER (OR)	

Arbeitsauftrag 72: Termbäume zeichnen ●○○

Zeichne zu jedem Term den Termbaum.

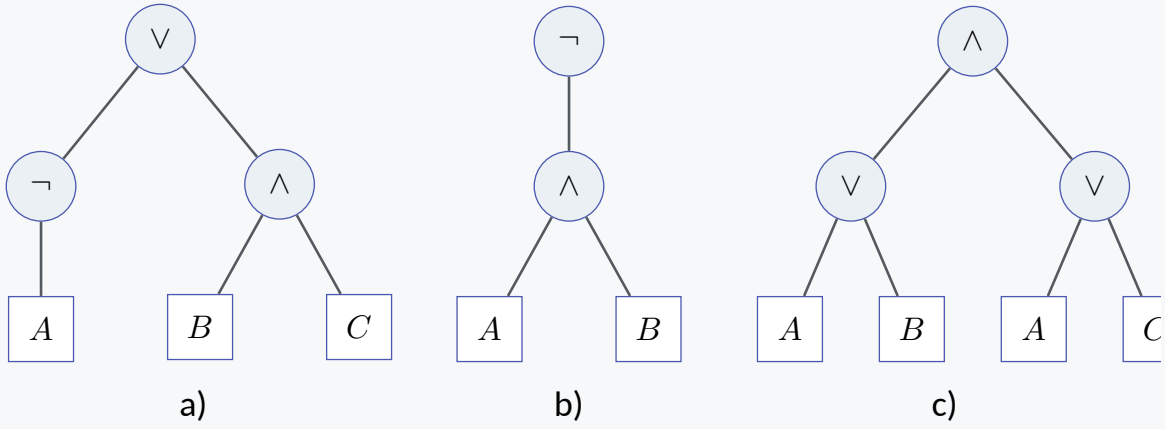
a) $\neg A \vee B$ b) $(A \vee B) \wedge C$ c) $\neg(A \vee \neg B)$

Deine Termbäume:

Gymnasium Ottweiler

Arbeitsauftrag 73: Vom Termbaum zum Term ●○○

Gib zu jedem Termbaum den Term an. Setze Klammern, wo sie nötig sind.



Drei Termbäume a), b) und c)

Deine Terme:

Arbeitsauftrag 74: Term oder kein Term? ●●○

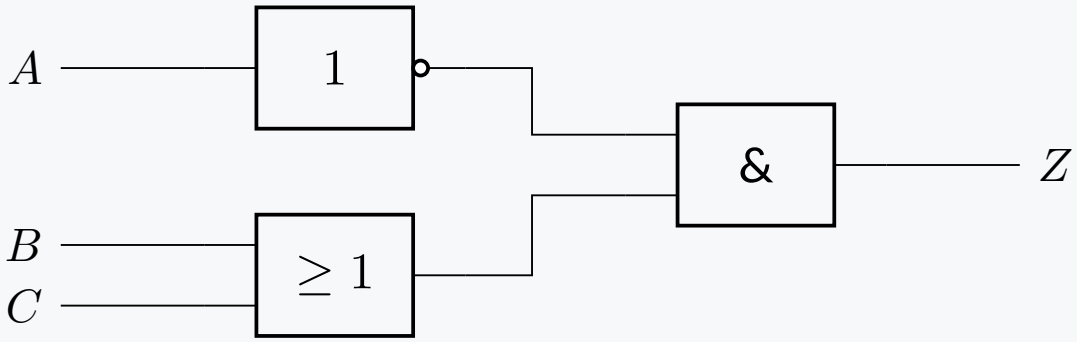
Entscheide mit der induktiven Definition, ob die Zeichenfolge ein aussagenlogischer Term ist. Begründe mit den Regeln.

a) A
b) $(A \wedge \neg C)$
c) $(A \vee B \vee)$
d) $((A \vee B) \wedge \neg C)$
e) $(A \oplus B)$
f) $\neg \wedge A$

Deine Antworten:

Arbeitsauftrag 75: Schaltung, Term und Termbaum ●●○

1. Gib den Term an, den die Schaltung darstellt.
2. Zeichne den Termbaum.
3. Berechne den Wert von Z für die Belegung $A = 0$, $B = 1$, $C = 0$. Schreibe die Zwischenwerte an die Knoten deines Termbaums.



Eine Schaltung mit drei Gattern

Term, Termbaum und Wert:

Arbeitsauftrag 76: Genau zwei von drei ●●●

Formuliere einen aussagenlogischen Term für die Situation: „Der Ausgang ist 1, wenn **genau zwei** der drei Variablen A, B, C den Wert 1 haben." Notiere den Term symbolisch **und** als Python-Funktion `genau_zwei(a, b, c)`. Prüfe die Funktion mit drei geschachtelten Schleifen.

Tipp

Geh vor wie im Auftrag *Genau eine von drei*: Überlege, welche Fälle es gibt. Beschreibe jeden Fall mit einer Konjunktion, in der alle drei Variablen vorkommen. Verbinde die Fälle mit $\vee$.

Dein Term:

2.6.8 Das Wichtigste in Kürze

- Eine **Aussagenvariable** steht für eine Aussage und hat den Wert 1 (wahr) oder 0 (falsch). Sie entspricht einem Eingang einer Schaltung.
- **Negation** $\neg$, **Konjunktion** $\wedge$ und **Disjunktion** $\vee$ entsprechen dem NICHT-, UND- und ODER-Gatter.
- Aussagenlogische Terme sind **induktiv** definiert: Variablen sind Terme; aus Termen entstehen mit $\neg$, $\wedge$, $\vee$ und Klammern neue Terme; sonst gibt es keine Terme.
- Klammern darf man sparen: äußerste Klammern weglassen, $\neg$ bindet am stärksten, dann $\wedge$, dann $\vee$.
- Ein **Termbaum** zeigt den Aufbau eines Terms: Blätter sind Variablen, innere Knoten Operatoren. Termbaum und Schaltung haben denselben Aufbau: Jeder innere Knoten ist ein Gatter.

Selbstcheck

- ☐ Ich kann Konjunktion, Disjunktion und Negation erklären und ihre Belegungstabellen angeben.
- ☐ Ich kann die induktive Definition eines aussagenlogischen Terms wiedergeben.
- ☐ Ich kann begründen, ob eine Zeichenfolge ein Term ist.
- ☐ Ich kann zu einem Term den Termbaum und die Schaltung zeichnen und zu einer Schaltung den Term angeben.
- ☐ Ich kann einen Term mit Python für alle Belegungen ausrechnen lassen.

Gymnasium Ottweiler

2.7 Belegungstabellen, Äquivalenz und disjunktive Normalform

Leitidee

Ein Term beschreibt eine Schaltung. Aber welche Werte liefert sie eigentlich? Das zeigt die **Belegungstabelle**. Spannend wird es, wenn man zwei Terme vergleicht: Oft sehen zwei Terme ganz verschieden aus und liefern trotzdem in jeder Zeile dasselbe. Dann beschreiben sie Schaltungen, die genau dasselbe tun – und man kann die billigere bauen. Umgekehrt hat man beim Entwurf oft nur eine Tabelle: „In diesen Fällen soll der Ausgang 1 sein." Mit der **disjunktiven Normalform** lernst du ein Verfahren, das zu jeder solchen Tabelle zuverlässig einen passenden Term liefert.

Lernziele

Nach diesem Abschnitt kannst du

- zu einem Term systematisch die Belegungstabelle erstellen,
- den Begriff der Äquivalenz zweier Terme erläutern,
- prüfen, ob zwei Terme äquivalent sind – von Hand und mit Python,
- aus einer Belegungstabelle die disjunktive Normalform (DNF) ablesen,
- die DNF zu einem gegebenen Term erstellen.

2.7.1 Belegungstabellen zu Termen

Eine Belegungstabelle enthält für jede mögliche Belegung der Variablen eine Zeile. Bei n Variablen gibt es 2^n Belegungen: bei zwei Variablen 4, bei drei Variablen 8, bei vier Variablen 16.

Damit keine Zeile fehlt, schreibst du die Belegungen immer in derselben Reihenfolge auf: Du **zählst im Binärsystem** von 000 bis 111. Dann wechselt die letzte Variable in jeder Zeile, die mittlere alle zwei Zeilen und die erste alle vier Zeilen. Die erste Spalte besteht also aus vier Nullen und vier Einsen, die zweite aus zwei Nullen, zwei Einsen, zwei Nullen, zwei Einsen.

Bei größeren Termen rechnet man nicht alles auf einmal aus. Man legt **Hilfsspalten** für die Teilterme an – und zwar in der Reihenfolge, in der man sie im Termbaum **von unten nach oben** findet. Für den Term $(A \wedge \neg B) \vee C$ sind das zuerst $\neg B$, dann $A \wedge \neg B$ und zuletzt der ganze Term:

A	B	C	$\neg B$	$A \wedge \neg B$	$(A \wedge \neg B) \vee C$
0	0	0	1	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	1	0	0	1
1	0	0	1	1	1
1	0	1	1	1	1
1	1	0	0	0	0
1	1	1	0	0	1

Jede Hilfsspalte entsteht aus Spalten weiter links. Die Spalte $A \wedge \neg B$ zum Beispiel ist nur dort 1, wo **sowohl** in der Spalte A **als auch** in der Spalte $\neg B$ eine 1 steht. Die letzte Spalte ist 1, wo in der Spalte $A \wedge \neg B$ **oder** in der Spalte C eine 1 steht.

Tipp

Lies eine Zeile der Belegungstabelle immer als ganzen Satz. Die fünfte Zeile besagt: „Ist A mit 1, B mit 0 und C mit 0 belegt, dann hat der Term den Wert 1."

Arbeitsauftrag 77: Vom Term zur Belegungstabelle

Erstelle **von Hand** die vollständige Belegungstabelle zum Term

$$(A \vee B) \wedge \neg C.$$

Überprüfe dein Ergebnis anschließend mit einem Python-Programm.

A	B	C	$A \vee B$	$\neg C$	$(A \vee B) \wedge \neg C$
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

2.7.2 Äquivalenz: verschiedene Terme, gleiche Wirkung

Vergleiche die beiden Terme $A \vee (A \wedge B)$ und A. Sie sehen ganz verschieden aus. Schau dir aber ihre Belegungstabellen an:

A	B	$A \wedge B$	$A \vee (A \wedge B)$
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

Die letzte Spalte stimmt Zeile für Zeile mit der Spalte A überein! Egal, wie man A und B belegt: Beide Terme liefern immer denselben Wert. Man sagt: Die beiden Terme sind **äquivalent** (gleichwertig).

Für Schaltungen ist das sehr nützlich. Die Schaltung zu $A \vee (A \wedge B)$ braucht ein UND- und ein ODER-Gatter. Die Schaltung zu A braucht gar kein Gatter – der Ausgang ist einfach mit dem Eingang A verbunden. Beide Schaltungen verhalten sich genau gleich. Natürlich baut man dann die einfachere.

Definition: Äquivalenz

Zwei aussagenlogische Terme heißen **äquivalent**, wenn sie für **jede** mögliche Belegung der Variablen denselben Wert liefern, wenn also ihre Belegungstabellen übereinstimmen.

Beachte: „äquivalent“ ist ein Wort, mit dem wir **über** zwei Terme sprechen – so wie man über zwei Wörter sagen kann, dass sie gleich lang sind. Es ist kein neues Zeichen, das man in einen Term hineinschreibt wie $\wedge$ oder $\vee$. Wir schreiben deshalb einfach: „Die Terme $A \vee (A \wedge B)$ und A sind äquivalent.“

Merke

Zwei aussagenlogische Terme heißen **äquivalent**, wenn sie für **jede** mögliche Belegung denselben Wahrheitswert liefern – also **dieselbe Belegungstabelle** besitzen. Äquivalenz ist ein Begriff **über** Terme, kein eigener Operator!

Nicht äquivalent: Ein Gegenbeispiel genügt

Um zu zeigen, dass zwei Terme äquivalent sind, musst du **alle** Zeilen prüfen. Um zu zeigen, dass sie **nicht** äquivalent sind, reicht **eine einzige** Zeile, in der sie sich unterscheiden. Eine solche Zeile heißt **Gegenbeispiel**.

A	B	$A \vee B$	$A \oplus B$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	0

ODER und XOR stimmen in drei Zeilen überein, aber in der letzten Zeile nicht. Für $A = 1, B = 1$ ist $A \vee B = 1$, aber $A \oplus B = 0$. Also sind die beiden nicht äquivalent.

Äquivalenz mit Python prüfen

Bei drei oder vier Variablen ist der Vergleich von Hand mühsam. Python kann das schnell erledigen. Die Idee: Man geht alle Belegungen durch und vergleicht die Werte beider Terme. Am Anfang nimmt man an, dass die Terme äquivalent sind. Findet man eine Belegung, in der sie sich unterscheiden, merkt man sich das.

```python
def term_links(a, b):
    return a or (a and b)

def term_rechts(a, b):
    return a

aequivalent = True
for a in [0, 1]:
    for b in [0, 1]:
        if term_links(a, b) != term_rechts(a, b):
            aequivalent = False
            print(f"Unterschied bei A={a} B={b}")

print("Äquivalent:", aequivalent)
```

AUSGABE

```
Äquivalent: True
```

- **Zeilen 1–5:** Jeder Term wird als eigene Funktion geschrieben.
- **Zeile 7:** Die Variable `aequivalent` startet mit `True`: Solange kein Unterschied gefunden ist, gehen wir von Äquivalenz aus.
- **Zeilen 8–9:** Zwei geschachtelte Schleifen erzeugen alle vier Belegungen.
- **Zeilen 10–12:** Liefern die beiden Funktionen verschiedene Werte, wird `aequivalent` auf `False` gesetzt und das Gegenbeispiel ausgegeben.
- **Zeile 14:** Am Ende steht fest, ob es irgendeine Belegung mit einem Unterschied gab.

Hier wurde kein Unterschied gefunden, also gibt das Programm nur `True` aus.

Versuch: Probiere es aus

Ersetze in `term_links` das `or` durch `and`. Welche Gegenbeispiele findet das Programm jetzt? Prüfe auch: Ist `a and (a or b)` äquivalent zu `a`?

Arbeitsauftrag 78: Äquivalenz prüfen

Prüfe **mit Python**, ob die beiden folgenden Terme äquivalent sind. Vergleiche dazu für **alle** Belegungen, ob beide Funktionen stets denselben Wert liefern.

Term A: $\neg(A \wedge B)$ Term B: $\neg A \vee \neg B$

Ergänze das Programm:

```python
def term_a(a, b):
    return not (a and b)

def term_b(a, b):
    return (not a) or (not b)

aequivalent = True
for a in [0, 1]:
    for b in [0, 1]:
        # Vergleich einbauen und ggf. aequivalent = False setzen
        ...

print("Äquivalent:", aequivalent)
```

Erstelle zusätzlich die Belegungstabelle beider Terme von Hand.

Meine Tabelle:

Die Regeln von De Morgan

Die Äquivalenz aus dem letzten Auftrag ist so wichtig, dass sie einen eigenen Namen hat. Der englische Mathematiker **Augustus De Morgan** hat im 19. Jahrhundert zwei Regeln aufgeschrieben, die zeigen, wie man eine verneinte Klammer auflöst:

Merke

Regeln von De Morgan: Für alle Belegungen von A und B gilt

- $\neg(A \wedge B)$ ist äquivalent zu $\neg A \vee \neg B$ („nicht beide“ = „mindestens einer nicht“),
- $\neg(A \vee B)$ ist äquivalent zu $\neg A \wedge \neg B$ („keiner von beiden“ = „der eine nicht und der andere auch nicht“).

Merkregel: Zieht man das $\neg$ in die Klammer, wird **jede** Variable verneint, und aus $\wedge$ wird $\vee$ bzw. aus $\vee$ wird $\wedge$.

Ein Beispiel aus dem Alltag: „Es ist **nicht** so, dass es regnet **oder** windet“ bedeutet dasselbe wie „Es regnet **nicht und** es windet **nicht**“. Das ist die zweite Regel.

Für Schaltungen heißt das: Ein NAND-Gatter (UND mit verneintem Ausgang) leistet dasselbe wie ein ODER-Gatter, dessen beide Eingänge vorher verneint werden. Ein NOR-Gatter leistet dasselbe wie ein UND-Gatter mit zwei verneinten Eingängen. Mit den Regeln von De Morgan kann man also Gatter gegeneinander austauschen, ohne dass sich das Verhalten der Schaltung ändert.

Arbeitsauftrag 79: Die zweite Regel von De Morgan

Zeige mit einer Belegungstabelle, dass $\neg(A \vee B)$ und $\neg A \wedge \neg B$ äquivalent sind. Fülle dazu die Tabelle aus und begründe dein Ergebnis in einem Satz.

A	B	$A \vee B$	$\neg(A \vee B)$	$\neg A$	$\neg B$	$\neg A \wedge \neg B$
0	0					
0	1					
1	0					
1	1					

Begründung:

Stolperstein

Ein sehr häufiger Fehler ist, das $\neg$ einfach in die Klammer „hineinzuziehen“: $\neg(A \wedge B)$ ist **nicht** äquivalent zu $\neg A \wedge \neg B$. Gegenbeispiel: Für $A = 1$, $B = 0$ ist $\neg(1 \wedge 0) = \neg 0 = 1$, aber $\neg 1 \wedge \neg 0 = 0 \wedge 1 = 0$. Wer das $\neg$ in die Klammer zieht, muss das $\wedge$ in ein $\vee$ umwandeln (und umgekehrt).

2.7.3 Die disjunktive Normalform (DNF)

Beim Schaltungsentwurf steht am Anfang oft eine Belegungstabelle: In diesen Fällen soll der Ausgang 1 sein, in den anderen 0. Wie kommt man von der Tabelle zu einem Term? Dafür gibt es ein Verfahren, das **immer** funktioniert.

Die Idee: ein Steckbrief für jede 1-Zeile

Schau dir diese Tabelle an:

A	B	**Ausgang**
0	0	0
0	1	1
1	0	1
1	1	0

Der Ausgang ist in zwei Zeilen 1. Für jede dieser Zeilen schreiben wir einen „Steckbrief“: eine Konjunktion, die **genau in dieser einen Zeile** den Wert 1 hat und sonst überall 0.

- Zeile $A = 0$, $B = 1$: Wir brauchen „A ist 0 und B ist 1“. „A ist 0“ heißt dasselbe wie „$\neg A$ ist 1“. Der Steckbrief lautet also $\neg A \wedge B$.
- Zeile $A = 1$, $B = 0$: Der Steckbrief lautet $A \wedge \neg B$.

Der Ausgang soll 1 sein, wenn wir in der **einen oder** in der **anderen** Zeile sind. Also verbinden wir die Steckbriefe mit $\vee$:

$$(\neg A \wedge B) \vee (A \wedge \neg B)$$

Das ist die **disjunktive Normalform** der Tabelle. Übrigens ist die Tabelle genau die von XOR. Wir haben also nebenbei herausgefunden, wie man XOR mit $\wedge$, $\vee$ und $\neg$ schreibt.

Definition: Minterm und disjunktive Normalform

Ein **Minterm** ist eine Konjunktion, in der **jede** Variable genau einmal vorkommt, entweder ohne oder mit $\neg$. Er hat genau in einer Zeile der Belegungstabelle den Wert 1.

Die **disjunktive Normalform (DNF)** einer Belegungstabelle ist die Disjunktion ($\vee$) der Minterme aller Zeilen, in denen das Ergebnis 1 ist.

Der Name erklärt sich so: **disjunktiv**, weil die Minterme mit $\vee$ (Disjunktion) verbunden werden; **Normalform**, weil jeder Term auf diese feste, „normale“ Form gebracht werden kann.

Merke

Die **disjunktive Normalform (DNF)** ist eine standardisierte Darstellung eines Terms:

- Für **jede** Zeile der Belegungstabelle, in der der Ausgang 1 ist, bildet man einen **Minterm**: die Konjunktion aller Variablen, wobei Variablen mit Wert 0 negiert werden.
- Alle Minterme werden anschließend **disjunktiv** (mit $\vee$) verknüpft.

Beispiel: Ist der Ausgang bei $A = 1, B = 0$ und bei $A = 0, B = 1$ gleich 1 (sonst 0), lautet die DNF $(A \wedge \neg B) \vee (\neg A \wedge B)$ – das ist genau XOR!

Ob man die Minterme in der Reihenfolge $(\neg A \wedge B) \vee (A \wedge \neg B)$ oder $(A \wedge \neg B) \vee (\neg A \wedge B)$ aufschreibt, ist egal. Bei $\vee$ kommt es auf die Reihenfolge nicht an.

Mit Python lässt sich prüfen, dass die DNF wirklich die Tabelle von oben liefert:

```python
def dnf_beispiel(a, b):
    return int((not a and b) or (a and not b))

for a in [0, 1]:
    for b in [0, 1]:
        print(f"A={a} B={b} -> {dnf_beispiel(a, b)}")
```

AUSGABE

```
A=0 B=0 -> 0
A=0 B=1 -> 1
A=1 B=0 -> 1
A=1 B=1 -> 0
```

Die Klammern um `not a and b` sind in Python eigentlich nicht nötig, weil `and` stärker bindet als `or`. Sie machen die Funktion aber leichter lesbar.

Warum funktioniert das?

Jeder Minterm ist in **genau einer** Zeile 1 – nämlich in der Zeile, aus der er gebaut wurde. In jeder anderen Zeile stimmt mindestens eine Variable nicht, und die Konjunktion wird 0. Die Disjunktion aller Minterme ist deshalb genau in den Zeilen 1, aus denen ein Minterm gebaut wurde – und das sind genau die 1-Zeilen

der Tabelle. Die DNF hat also dieselbe Belegungstabelle wie der Ausgang. Wenn die Tabelle zu einem Term gehört, ist die DNF zu diesem Term **äquivalent**.

Das Verfahren Schritt für Schritt

1. Suche in der Belegungstabelle alle Zeilen, in denen das Ergebnis 1 ist. Die Zeilen mit 0 brauchst du nicht.
2. Schreibe für jede dieser Zeilen den Minterm auf: alle Variablen mit $\wedge$ verbunden; eine Variable, die in der Zeile 0 ist, bekommt ein $\neg$.
3. Setze um jeden Minterm Klammern und verbinde alle Minterme mit $\vee$.

Beispiel

DNF der Alarmanlage. Wir kürzen ab: T für Tür, S für Scharf, R für Rauch. Der Term lautet $(T \wedge S) \vee R$. Zuerst die Belegungstabelle, dann für jede 1-Zeile der Minterm:

T	S	R	$(T \wedge S) \vee R$	**Minterm**
0	0	0	0	
0	0	1	1	$\neg T \wedge \neg S \wedge R$
0	1	0	0	
0	1	1	1	$\neg T \wedge S \wedge R$
1	0	0	0	
1	0	1	1	$T \wedge \neg S \wedge R$
1	1	0	1	$T \wedge S \wedge \neg R$
1	1	1	1	$T \wedge S \wedge R$

Die DNF lautet:

$$(\neg T \wedge \neg S \wedge R) \vee (\neg T \wedge S \wedge R) \vee (T \wedge \neg S \wedge R) \vee (T \wedge S \wedge \neg R) \vee (T \wedge S \wedge R)$$

Sie ist äquivalent zu $(T \wedge S) \vee R$, aber viel länger. Im nächsten Abschnitt lernst du, wie man eine DNF wieder vereinfacht.

Stolperstein

- **Negation vertauscht:** Eine Variable bekommt ein $\neg$, wenn sie in der Zeile **0** ist – nicht, wenn sie 1 ist.
- **Variable vergessen:** In jedem Minterm müssen **alle** Variablen vorkommen. $\neg T \wedge R$ ist kein Minterm, wenn es drei Variablen gibt.
- **Falsche Zeilen:** Nur die Zeilen mit Ergebnis **1** werden verwendet.
- **Falsche Zeichen:** Innerhalb eines Minterms steht $\wedge$, zwischen den Mintermen $\vee$ – nicht umgekehrt.

So kannst du es sagen

So sprichst du über Belegungstabellen und Terme – vom einfachen zum genauen Satz:

- *„Die Tabelle passt zum Term."* → *„Die Belegungstabelle gehört zum Term."*
- *„In Zeile 5 kommt 1 raus."* → *„Die fünfte Zeile der Belegungstabelle besagt: Ist A mit 1 und sind B und C mit 0 belegt, hat der Term den Wert 1."*
- *„Die beiden Terme sind gleich."* → *„Die beiden Terme sind äquivalent, denn ihre Belegungstabellen stimmen überein."*

Arbeitsauftrag 80: DNF ablesen

Gegeben ist folgende Belegungstabelle:

A	B	C	**Ausgang**
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

1. Bestimme die DNF auf Papier.
2. Schreibe die DNF als Python-Funktion `meine_dnf(a, b, c)` und überprüfe sie mit drei geschachtelten Schleifen gegen die Tabelle.

Meine DNF:

Arbeitsauftrag 81: DNF zu einem Term

Bestimme die DNF zum Term $(A \vee B) \wedge \neg C$ aus dem Auftrag *Vom Term zur Belegungstabelle*. Benutze deine Belegungstabelle von dort.

Meine DNF:

2.7.4 Weitere Übungen

Hier kannst du das Gelernte weiter üben. Die Punkte im Titel zeigen, wie schwer eine Aufgabe ist: ●○○ leicht, ●●○ mittel, ●●● knifflig. Fang mit den leichten an.

Arbeitsauftrag 82: Belegungstabelle mit zwei Variablen ●○○

Erstelle die Belegungstabelle zum Term $\neg A \vee B$. Benutze die Hilfsspalte.

A	B	$\neg A$	$\neg A \vee B$
0	0		
0	1		
1	0		
1	1		

Arbeitsauftrag 83: Minterme aufschreiben ●○○

1. Schreibe zu jeder Belegung den Minterm auf (drei Variablen A, B, C).

   a) $A = 1, B = 0, C = 1$ b) $A = 0, B = 0, C = 0$ c) $A = 0, B = 1, C = 1$

2. Jetzt umgekehrt: Bei welcher Belegung hat der Minterm $\neg A \wedge B \wedge \neg C$ den Wert 1?

Deine Antworten:

Arbeitsauftrag 84: DNF mit zwei Variablen ●○○

Bestimme die DNF zu dieser Belegungstabelle. Beschreibe in Worten, wann der Ausgang 1 ist.

A	B	**Ausgang**
0	0	1
0	1	0
1	0	0
1	1	1

DNF und Beschreibung:

Gymnasium Ottweiler

Arbeitsauftrag 85: DNF mit drei Variablen ●○○

Bestimme die DNF zu dieser Belegungstabelle.

A	B	C	**Ausgang**
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

DNF:

Arbeitsauftrag 86: Äquivalent oder nicht? ●●○

Prüfe mit Belegungstabellen, ob die beiden Terme äquivalent sind. Wenn nicht, gib eine Belegung an, in der sie sich unterscheiden.

a) $A \vee (\neg A \wedge B)$ und $A \vee B$
b) $\neg A \wedge \neg B$ und $\neg(A \wedge B)$

Deine Tabellen und deine Antwort:

Gymnasium Ottweiler

Arbeitsauftrag 87: Vom Term zur DNF ●●○

Gegeben ist der Term $(A \wedge \neg B) \vee (B \wedge C)$.

1. Erstelle die Belegungstabelle mit Hilfsspalten.
2. Bestimme die DNF.

A	B	C	$\neg B$	$A \wedge \neg B$	$B \wedge C$	**Term**
0	0	0				
0	0	1				
0	1	0				
0	1	1				
1	0	0				
1	0	1				
1	1	0				
1	1	1				

DNF:

Arbeitsauftrag 88: Äquivalenz mit Python prüfen ●●○

Prüfe mit Python, ob die Terme äquivalent sind. Wenn nicht, soll das Programm alle Belegungen ausgeben, in denen sie sich unterscheiden.

a) $A \wedge (B \vee C)$ und $(A \wedge B) \vee (A \wedge C)$
b) $A \vee (B \wedge C)$ und $(A \vee B) \wedge C$

Tipp

Schreibe jeden Term als eigene Funktion. Gehe dann vor wie im Abschnitt *Äquivalenz mit Python prüfen*, nur mit drei geschachtelten Schleifen.

Arbeitsauftrag 89: Warnleuchte im Auto ●●●

Eine Warnleuchte im Auto ($L = 1$) leuchtet, wenn der Motor läuft ($M = 1$) und der Gurt nicht angelegt ist ($G = 0$). Sie leuchtet außerdem, wenn der Motor läuft und eine Tür offen ist ($T = 1$).

1. Stelle die Belegungstabelle für M, G, T auf.
2. Bestimme die DNF.
3. Mia hat den kürzeren Term $M \wedge (\neg G \vee T)$ gefunden. Prüfe mit Python, ob er zu deiner DNF äquivalent ist.

Tabelle und DNF:

2.7.5 Das Wichtigste in Kürze

- Eine **Belegungstabelle** hat bei n Variablen 2^n Zeilen. Die Belegungen werden wie Binärzahlen von $0 \ldots 0$ bis $1 \ldots 1$ aufgeschrieben. Hilfsspalten für Teilterme erleichtern das Ausrechnen.
- Zwei Terme sind **äquivalent**, wenn ihre Belegungstabellen übereinstimmen. Äquivalenz ist ein Begriff über Terme, kein Operator.
- Für den Nachweis der Äquivalenz prüft man alle Zeilen. Für den Nachweis, dass Terme nicht äquivalent sind, genügt ein **Gegenbeispiel**.
- **Regeln von De Morgan:** $\neg(A \wedge B)$ ist äquivalent zu $\neg A \vee \neg B$, und $\neg(A \vee B)$ ist äquivalent zu $\neg A \wedge \neg B$.
- Ein **Minterm** ist eine Konjunktion aller Variablen (Variablen mit Wert 0 negiert). Er ist genau in einer Zeile 1.
- Die **DNF** ist die Disjunktion der Minterme aller 1-Zeilen. Sie ist zum ursprünglichen Term äquivalent, aber oft lang.

Selbstcheck

- ☐ Ich kann zu einem Term mit Hilfsspalten die Belegungstabelle erstellen.
- ☐ Ich kann erklären, wann zwei Terme äquivalent sind.
- ☐ Ich kann mit einem Gegenbeispiel zeigen, dass zwei Terme nicht äquivalent sind.
- ☐ Ich kann die Regeln von De Morgan nennen und mit einer Belegungstabelle begründen.
- ☐ Ich kann mit Python prüfen, ob zwei Terme äquivalent sind.
- ☐ Ich kann aus einer Belegungstabelle die DNF ablesen.
- ☐ Ich kann die DNF zu einem gegebenen Term bestimmen.

2.8 KV-Diagramme: Terme vereinfachen, Schaltungen sparen

Leitidee

Im letzten Abschnitt hast du die disjunktive Normalform kennengelernt. Sie liefert zu jeder Belegungstabelle einen Term – aber oft einen sehr langen. Für die Alarmanlage hatte die DNF fünf Minterme, obwohl der Term $(T \wedge S) \vee R$ dasselbe leistet. Jedes $\wedge$, $\vee$ und $\neg$ ist in der Schaltung ein Gatter, und jedes Gatter kostet Geld, Platz und Strom. Deshalb will man Terme so weit wie möglich **vereinfachen**. Ein praktisches Hilfsmittel dafür ist das **Karnaugh-Veitch-Diagramm**, kurz **KV-Diagramm**. Es wurde in den 1950er-Jahren von Edward Veitch und Maurice Karnaugh erfunden. Damit findest du einen kurzen Term, indem du in einer Tabelle Kästchen um Einsen zeichnest.

Lernziele

Nach diesem Abschnitt kannst du

- die Kosten einer Schaltung über die Anzahl ihrer Gatter vergleichen,
- ein KV-Diagramm für zwei, drei und vier Variablen aufstellen und erklären, warum die Spalten in der Reihenfolge 00, 01, 11, 10 stehen,
- Einsen zu Blöcken der Größe 1, 2, 4 und 8 zusammenfassen, auch über den Rand hinweg,
- aus den Blöcken den vereinfachten Term ablesen,
- die vereinfachte Form mit der DNF vergleichen und ihre Äquivalenz mit Python prüfen.

2.8.1 Warum vereinfachen? Die Kosten einer Schaltung

Als einfaches Maß für die Kosten einer Schaltung zählen wir ihre **Gatter**. Dabei gilt:

- Wir nehmen an, dass jedes UND- und jedes ODER-Gatter **zwei** Eingänge hat. Für $A \wedge B \wedge C$ braucht man deshalb **zwei** UND-Gatter: eins für $A \wedge B$, ein zweites, das das Ergebnis mit C verknüpft. (Ein UND-Gatter mit drei Eingängen, wie bei der Fahrstuhltür, zählt also wie zwei Gatter.)
- Jedes NICHT-Gatter zählt mit. Kommt dieselbe negierte Variable mehrmals vor, genügt **ein** NICHT-Gatter: Seine Ausgangsleitung wird einfach verzweigt.

Vergleiche damit die beiden äquivalenten Terme der Alarmanlage (T: Tür offen, S: Scharf, R: Rauch):

Term	UND	ODER	NICHT	Gatter gesamt
DNF mit 5 Mintermen	$5 \cdot 2 = 10$	4	3	**17**
$(T \wedge S) \vee R$	1	1	0	**2**

Die DNF braucht 17 Gatter: Jeder der fünf Minterme hat drei Variablen und braucht zwei UND-Gatter. Vier ODER-Gatter verbinden die fünf Minterme. Und für $\neg T$, $\neg S$ und $\neg R$ braucht man drei NICHT-Gatter. Die einfache Form kommt mit zwei Gattern aus. Beide Schaltungen tun genau dasselbe! Je weniger Gatter, desto billiger, kleiner und sparsamer ist die Schaltung, und desto schneller liegt das Ergebnis vor.

2.8.2 Die Idee: Nachbarn zusammenfassen

Fangen wir klein an. Eine Belegungstabelle mit zwei Variablen hat genau in den Zeilen $A = 0, B = 0$ und $A = 0, B = 1$ eine 1. Die DNF lautet:

$$(\neg A \wedge \neg B) \vee (\neg A \wedge B)$$

Schau dir die beiden Minterme genau an. Sie stimmen in $\neg A$ überein und unterscheiden sich nur bei B. Der erste verlangt $B = 0$, der zweite $B = 1$. Zusammen heißt das: Wenn $A = 0$ ist, ist das Ergebnis 1 – **egal**, welchen Wert B hat. B spielt also keine Rolle und **fällt weg**. Übrig bleibt:

$$\neg A$$

Diese Regel ist der Kern der ganzen Methode: Zwei Minterme, die sich nur in **einer** Variablen unterscheiden, kann man zusammenfassen. Die Variable, in der sie sich unterscheiden, fällt weg.

Das prüfen wir mit Python. Beide Terme müssen für alle Belegungen denselben Wert liefern:

```python
def unminimiert(a, b):
    return (not a and not b) or (not a and b)

def minimiert(a, b):
    return not a

for a in [0, 1]:
    for b in [0, 1]:
        print(a, b, unminimiert(a, b) == minimiert(a, b))
```

AUSGABE

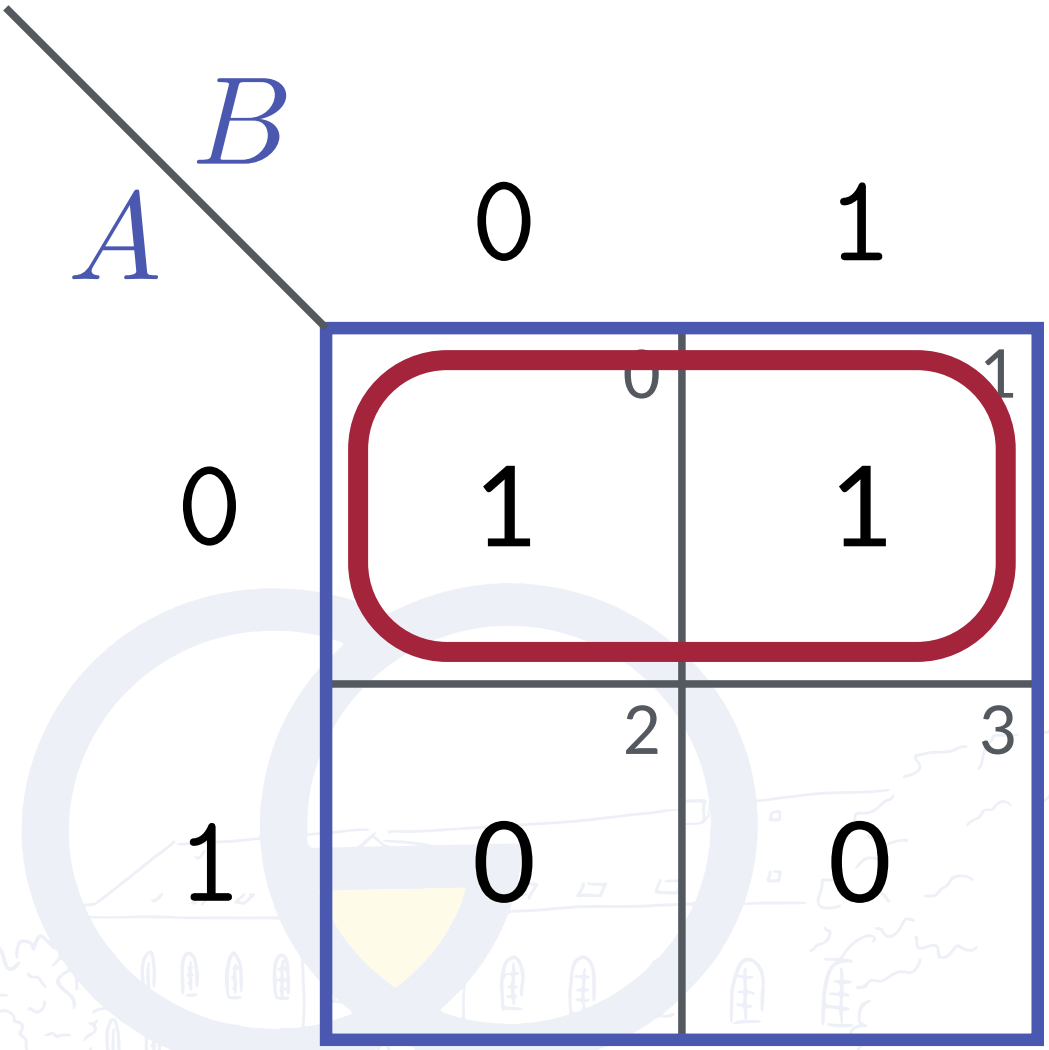
```
0 0 True
0 1 True
1 0 True
1 1 True
```

In jeder Zeile steht `True`: Die beiden Terme sind äquivalent.

Das KV-Diagramm mit zwei Variablen

Um solche Paare schnell zu finden, ordnet man die Zeilen der Belegungstabelle in einem Raster an, dem **KV-Diagramm**. Bei zwei Variablen hat es zwei Zeilen und zwei Spalten. Die Zeile gibt den Wert von A an, die Spalte den Wert von B. In jedem Feld steht der Ausgangswert für diese Belegung. Die kleinen grauen Zahlen sind die Nummern der Zeilen in der Belegungstabelle, gezählt ab 0. Man erhält sie, wenn man die Belegung als Binärzahl liest: Das Feld $A = 1, B = 0$ hat die Nummer $10_2 = 2$.

In unserem Beispiel stehen die beiden Einsen in der oberen Zeile ($A = 0$) nebeneinander. Wir umranden sie als **Block**. Im ganzen Block hat A den Wert 0, B dagegen wechselt. Also bleibt nur $\neg A$ übrig:



KV-Diagramm zu $(\neg A \wedge \neg B) \vee (\neg A \wedge B)$: Die beiden Einsen bilden einen Block mit $A = 0$. Die kleinen grauen Zahlen sind die Zeilennummern der Belegungstabelle

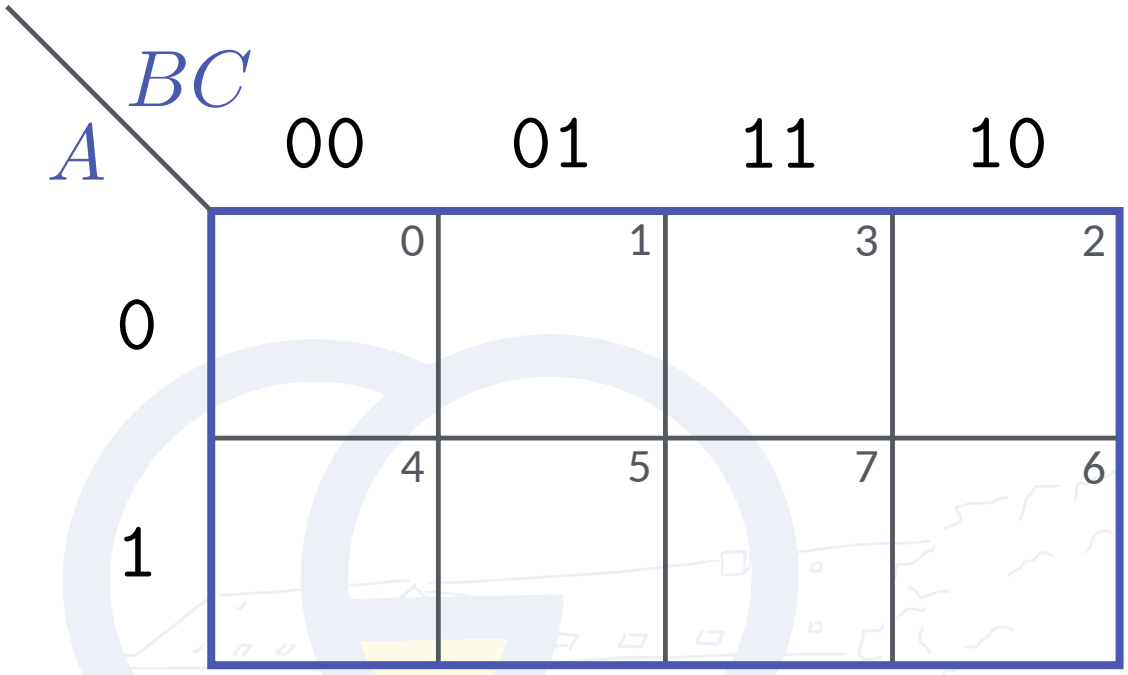
Definition: Karnaugh-Veitch-Diagramm

Ein **Karnaugh-Veitch-Diagramm (KV-Diagramm)** ist eine Anordnung aller Belegungen einer Belegungstabelle in einem Raster. Die Felder sind so angeordnet, dass sich benachbarte Felder in **genau einer** Variablen unterscheiden.

2.8.3 KV-Diagramme mit drei Variablen

Bei drei Variablen hat das KV-Diagramm zwei Zeilen und vier Spalten. Die Zeile gibt den Wert von A an. Die Spalte gibt die Werte von B und C zusammen an. Achtung: Die Spalten stehen **nicht** in der Reihenfolge 00, 01, 10, 11, sondern

$$00, \quad 01, \quad 11, \quad 10.$$



KV-Diagramm für drei Variablen. Die kleine graue Zahl in jedem Feld ist die Nummer der passenden Zeile in der Belegungstabelle

Warum diese seltsame Reihenfolge?

Damit man Nachbarn zusammenfassen kann, müssen sich benachbarte Felder in **genau einer** Variablen unterscheiden. Bei der gewohnten Reihenfolge 00, 01, 10, 11 stünden 01 und 10 nebeneinander. Die beiden unterscheiden sich aber in **beiden** Stellen. In der Reihenfolge 00, 01, 11, 10 ändert sich von Spalte zu Spalte immer nur eine Stelle:

```
00 -> 01    (C ändert sich)
01 -> 11    (B ändert sich)
11 -> 10    (C ändert sich)
10 -> 00    (B ändert sich)
```

Diese Reihenfolge heißt **Gray-Code** (nach dem Ingenieur Frank Gray). Beachte die letzte Zeile: Auch die **letzte und die erste Spalte** unterscheiden sich nur in einer Stelle. Sie sind also ebenfalls Nachbarn, obwohl sie am Rand weit auseinander stehen. Stell dir vor, das Diagramm wäre zu einer Rolle gebogen, sodass der linke und der rechte Rand aneinanderstoßen.

Tipp

Um eine Belegungstabelle ins KV-Diagramm zu übertragen, liest du in jeder Zeile die Belegung als Binärzahl. Die Zeile $A = 1, B = 0, C = 1$ hat die Nummer $101_2 = 5$. Ihr Wert gehört in das Feld in der Zeile $A = 1$ und der Spalte $BC = 01$. Die Feldnummern in der Vorlage oben helfen dir dabei.

Blöcke bilden

Jetzt fassen wir benachbarte Einsen zu **Blöcken** zusammen. Dafür gelten feste Regeln:

1. Ein Block enthält **nur Einsen**.
2. Ein Block ist ein **Rechteck** aus 1, 2, 4 oder 8 Feldern. Blöcke aus 3 oder 6 Feldern gibt es nicht.
3. Blöcke dürfen **über den Rand** gehen: Gegenüberliegende Ränder sind Nachbarn.
4. Blöcke dürfen sich **überlappen**.
5. **Jede** 1 muss in mindestens einem Block liegen.
6. Die Blöcke sollen **so groß wie möglich** sein, und es sollen **so wenige wie möglich** sein.

Zu jedem Block gehört ein Term. Du findest ihn so: Schau, welche Variablen in **allen** Feldern des Blocks **denselben** Wert haben. Nur diese bleiben übrig – mit $\neg$, wenn der Wert 0 ist. Alle anderen fallen weg. Die Terme aller Blöcke werden mit $\vee$ verbunden.

Je größer ein Block ist, desto mehr Variablen fallen weg. Bei drei Variablen bleiben in einem Einzelfeld drei Variablen übrig, in einem Zweierblock zwei, in einem Viererblock eine.

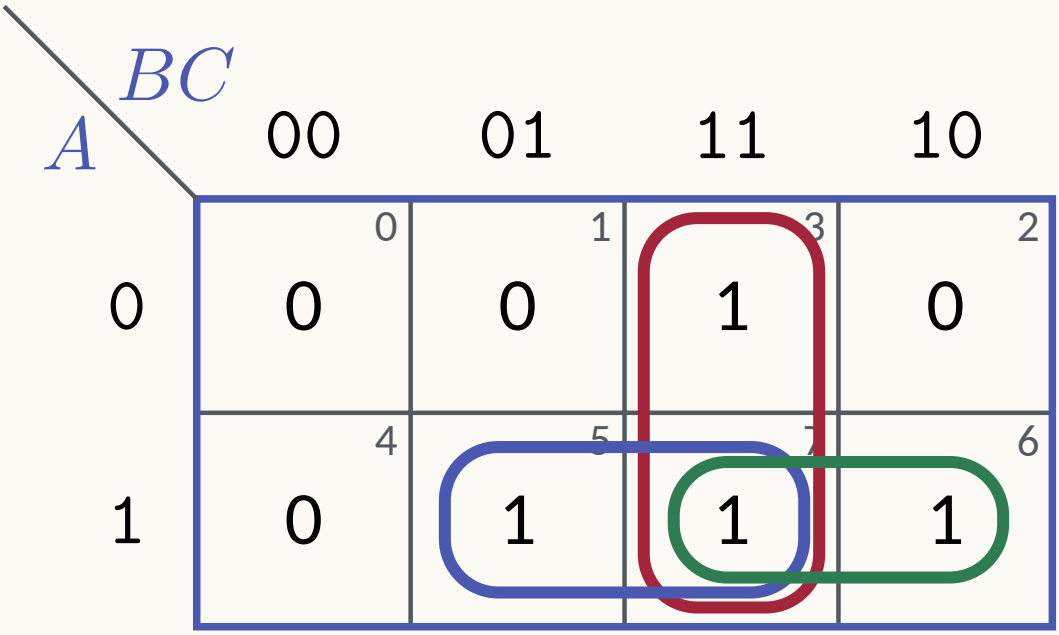
Merke

- Ein **Karnaugh-Veitch-Diagramm (KV-Diagramm)** ordnet alle Belegungen so in einem Raster an, dass sich **benachbarte Felder nur in genau einer Variablen** unterscheiden.
- Benachbarte Einsen werden zu **Blöcken** der Größe 1, 2, 4, 8 … zusammengefasst. Je größer der Block, desto **weniger Variablen** braucht der zugehörige Term.
- Innerhalb eines Blocks fällt jede Variable weg, die sowohl **negiert als auch nicht negiert** vorkommt. Übrig bleiben nur die Variablen, die im gesamten Block **konstant** sind.

Beispiel

Mehrheitsentscheid. Drei Personen A, B, C stimmen ab. Der Ausgang ist 1, wenn mindestens zwei dafür sind. Das ist in den Zeilen 3, 5, 6 und 7 der Belegungstabelle der Fall. Die DNF hat vier Minterme:

$$(\neg A \wedge B \wedge C) \vee (A \wedge \neg B \wedge C) \vee (A \wedge B \wedge \neg C) \vee (A \wedge B \wedge C)$$



KV-Diagramm zum Mehrheitsentscheid: drei Zweierblöcke, die sich im Feld 7 überlappen

Feld 7 hat drei Nachbarn mit einer 1: Feld 3 direkt darüber, Feld 5 links daneben und Feld 6 rechts daneben. Einen Viererblock gibt es nicht, weil die Felder 1, 2 und 4 den Wert 0 haben. Also bildet Feld 7 mit jedem seiner Nachbarn einen Zweierblock:

- **Rot** (Felder 3 und 7): $B = 1$ und $C = 1$ sind konstant, A wechselt. Term: $B \wedge C$.
- **Blau** (Felder 5 und 7): $A = 1$ und $C = 1$ sind konstant, B wechselt. Term: $A \wedge C$.
- **Grün** (Felder 6 und 7): $A = 1$ und $B = 1$ sind konstant, C wechselt. Term: $A \wedge B$.

Der vereinfachte Term lautet $(B \wedge C) \vee (A \wedge C) \vee (A \wedge B)$. Feld 7 liegt in allen drei Blöcken. Das ist erlaubt und sogar nötig, denn sonst könnte man die anderen Einsen nicht in Zweierblöcke packen. Die DNF braucht 14 Gatter (8 UND, 3 ODER, 3 NICHT), die vereinfachte Form nur 5 (3 UND, 2 ODER).

Stolperstein

- **Diagonal ist nicht benachbart.** Zwei Felder, die sich nur an einer Ecke berühren, unterscheiden sich in zwei Variablen. Sie dürfen nicht in einen Zweierblock.
- **Keine Dreierblöcke.** Drei Einsen in einer Reihe deckt man mit zwei überlappenden Zweierblöcken ab.
- **Nicht zu klein.** Wer zwei Zweierblöcke zeichnet, wo ein Viererblock möglich ist, bekommt einen richtigen, aber längeren Term.
- **Rand vergessen.** Viele übersehen Blöcke über den Rand. Prüfe immer: Stehen in der ersten und der letzten Spalte (bei vier Variablen auch in der ersten und der letzten Zeile) Einsen, die man zusammenfassen kann? Wie das aussieht, zeigt der Abschnitt *Über den Rand hinaus* weiter unten.

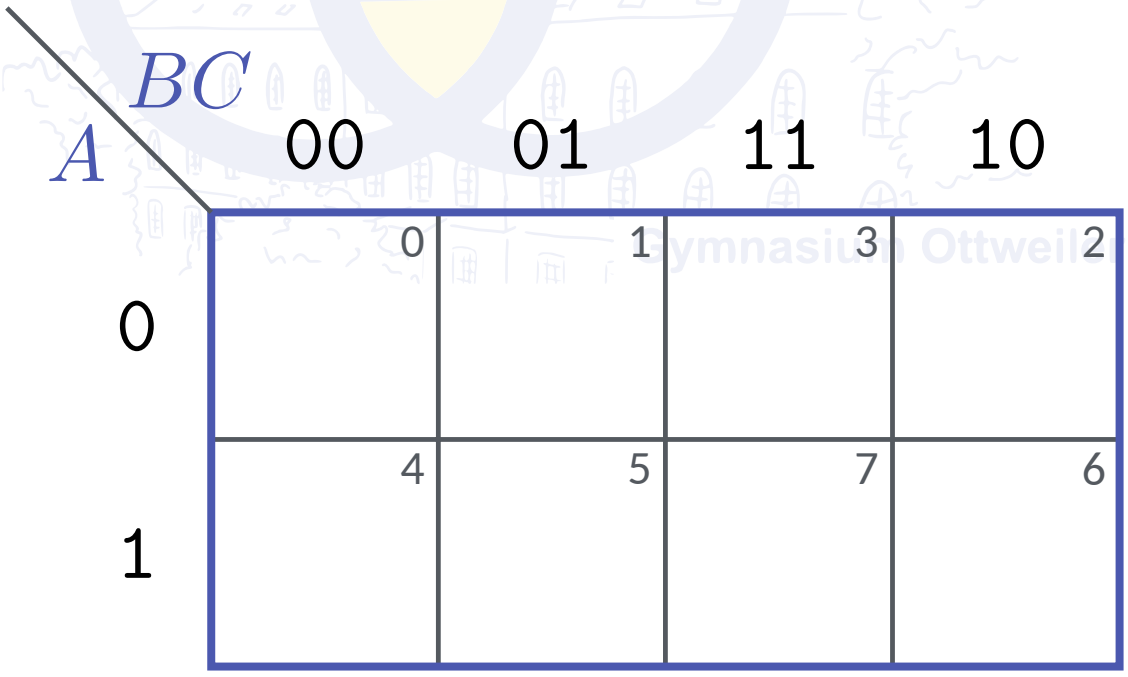
Arbeitsauftrag 90: KV-Diagramm zeichnen und minimieren

Gegeben ist folgende Belegungstabelle für drei Variablen:

A	B	C	**Ausgang**
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

1. Zeichne ein KV-Diagramm und finde die größtmöglichen Blöcke.
2. Notiere die minimierte Form.
3. Überprüfe deine minimierte Form mit Python gegen die unminimierte DNF. Verwende dazu diese Funktion:

```python
def unminimiert_a(a, b, c):
    return ((not a and not b and not c) or (not a and not b and c) or
            (a and not b and not c) or (a and not b and c))
```

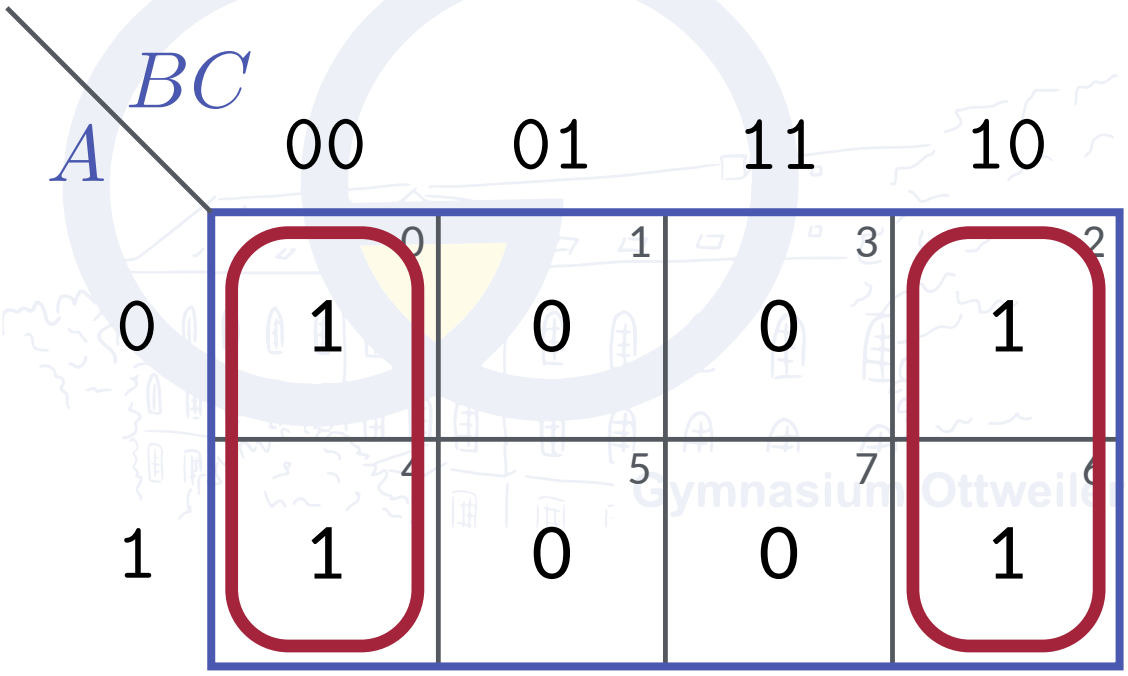


Vorlage zum Ausfüllen: KV-Diagramm für drei Variablen mit Feldnummern

Minimierte Form:

2.8.4 Über den Rand hinaus

Weil der linke und der rechte Rand Nachbarn sind, dürfen Blöcke über den Rand gehen. In diesem Beispiel ist der Ausgang in den Zeilen 0, 2, 4 und 6 gleich 1:



Randüberlappung: Die linke und die rechte Spalte sind Nachbarn. Die vier Einsen bilden einen Block mit $C = 0$

Die Einsen stehen in der ersten Spalte ($BC = 00$) und in der letzten Spalte ($BC = 10$). Diese beiden Spalten unterscheiden sich nur in B. Zusammen bilden die vier Felder einen Viererblock. In der Abbildung ist er als zwei getrennte Rechtecke in derselben Farbe gezeichnet – denk dir die Rolle zusammengebogen, dann stoßen die beiden Teile aneinander. Im Block ist überall $C = 0$, A und B wechseln. Der Term lautet $\neg C$.

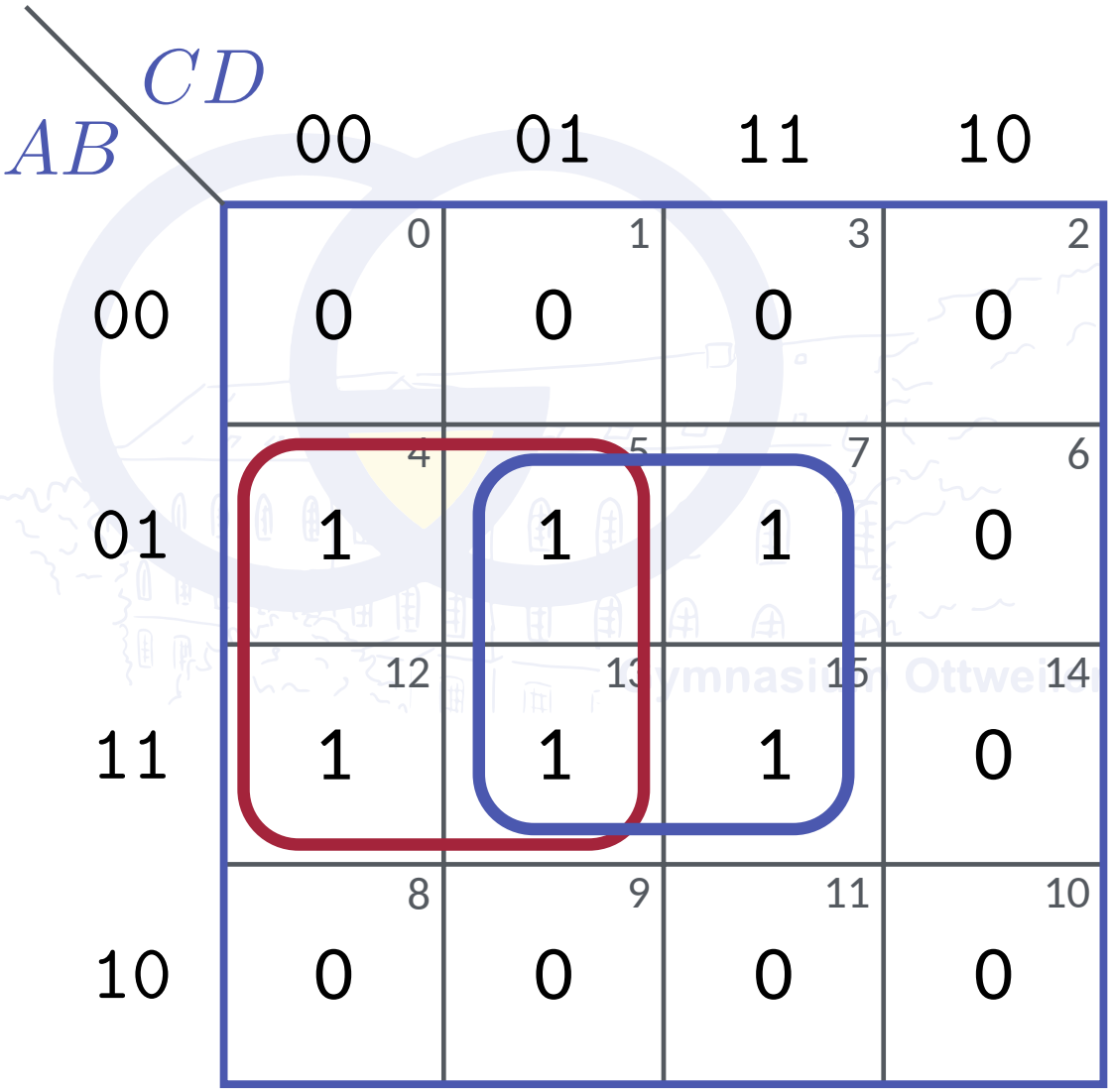
2.8.5 KV-Diagramme mit vier Variablen

Bei vier Variablen hat das KV-Diagramm vier Zeilen und vier Spalten. Die Zeilen geben die Werte von A und B an, die Spalten die Werte von C und D. **Beide** folgen dem Gray-Code 00, 01, 11, 10. Die Feldnummern erhältst du wieder, indem du $ABCD$ als Binärzahl liest. Im ersten Beispiel unten sind sie in jedem Feld eingetragen.

Jetzt sind sowohl der linke und der rechte Rand Nachbarn als auch der obere und der untere Rand. Blöcke dürfen also nach rechts **und** nach unten über den Rand gehen. Bei vier Variablen bleiben in einem Zweierblock drei Variablen übrig, in einem Viererblock zwei und in einem Achterblock nur noch eine.

Beispiel: zwei überlappende Viererblöcke

Der Ausgang ist in den Zeilen 4, 5, 7, 12, 13 und 15 gleich 1:



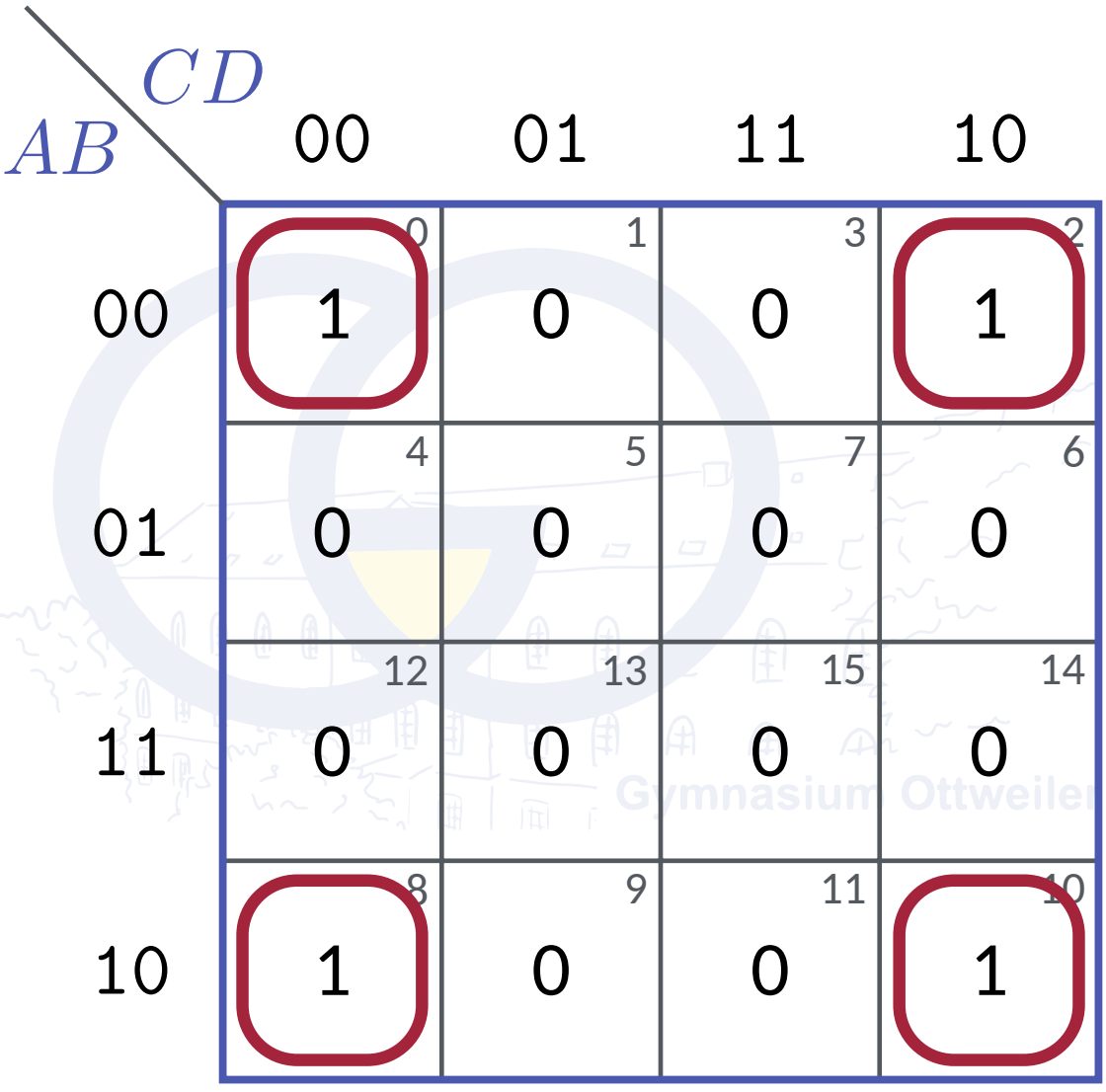
Zwei Viererblöcke, die sich überlappen: $B \wedge \neg C$ (rot) und $B \wedge D$ (blau)

- **Rot** (Felder 4, 5, 12, 13): In beiden Zeilen ist $B = 1$, in beiden Spalten ist $C = 0$. A und D wechseln. Term: $B \wedge \neg C$.
- **Blau** (Felder 5, 7, 13, 15): $B = 1$ und $D = 1$ sind konstant. Term: $B \wedge D$.

Der vereinfachte Term lautet $(B \wedge \neg C) \vee (B \wedge D)$. Die DNF hätte sechs Minterme mit je vier Variablen.

Beispiel: die vier Ecken

Die vier Eckfelder sind alle miteinander benachbart: Links und rechts sind Nachbarn, oben und unten auch. Sind genau die Ecken 1, bilden sie einen Viererblock:

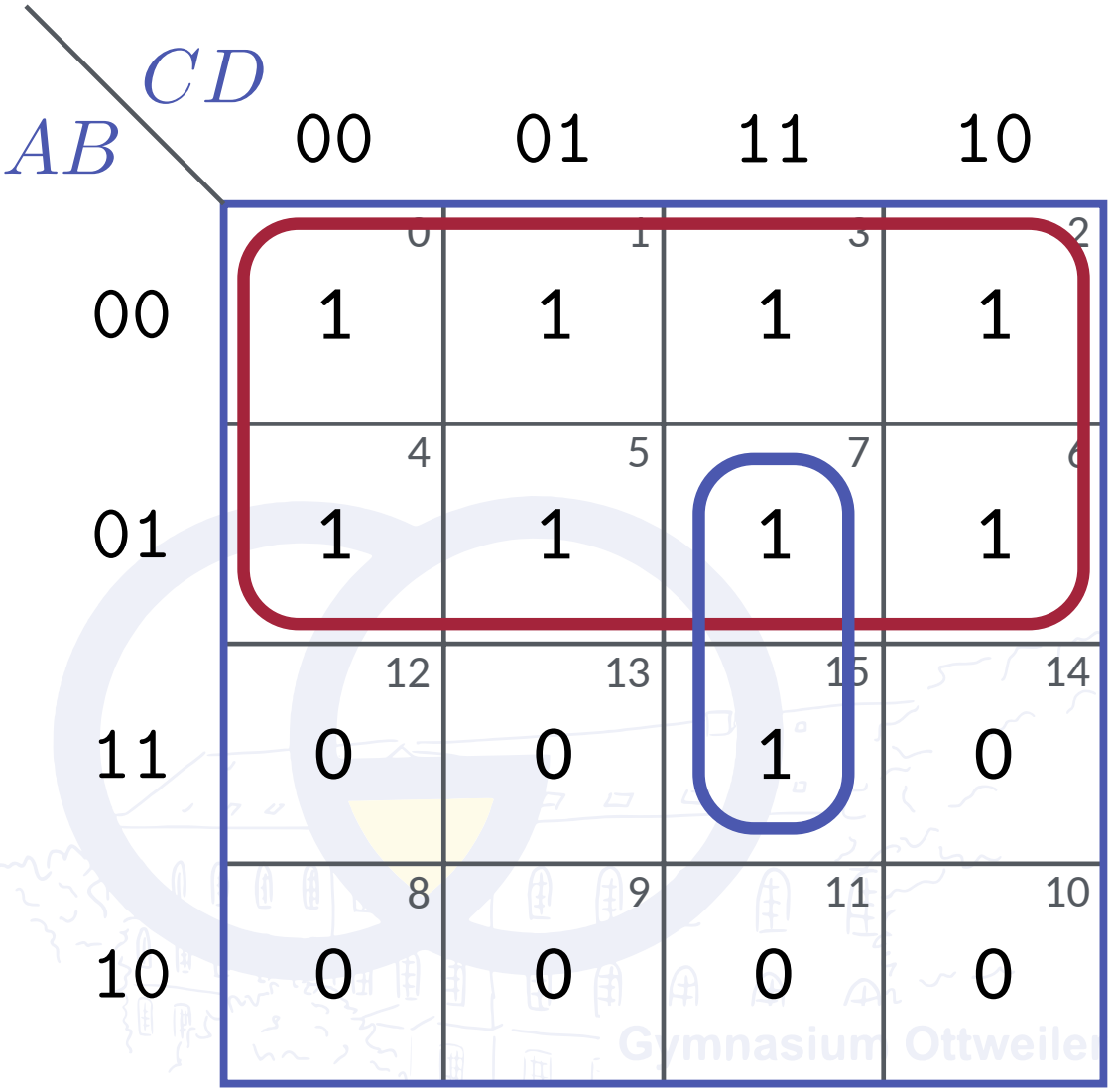


Die vier Ecken sind Nachbarn und bilden einen Viererblock mit $B = 0$ und $D = 0$

In allen vier Ecken ist $B = 0$ (Zeilen 00 und 10) und $D = 0$ (Spalten 00 und 10). Der Term lautet $\neg B \wedge \neg D$.

Beispiel: ein Achterblock

Hier ist der Ausgang in den Zeilen 0 bis 7 und in Zeile 15 gleich 1:



Ein Achterblock ($A = 0$) und ein Zweierblock ($B \wedge C \wedge D$)

- **Rot** (die beiden oberen Zeilen, acht Felder): Nur $A = 0$ ist konstant. Term: $\neg A$.
- **Blau** (Felder 7 und 15): Feld 15 hat keinen Viererblock. Sein bester Partner ist Feld 7 direkt darüber, denn die Zeilen 01 und 11 sind benachbart. $B = 1$, $C = 1$ und $D = 1$ sind konstant. Term: $B \wedge C \wedge D$.

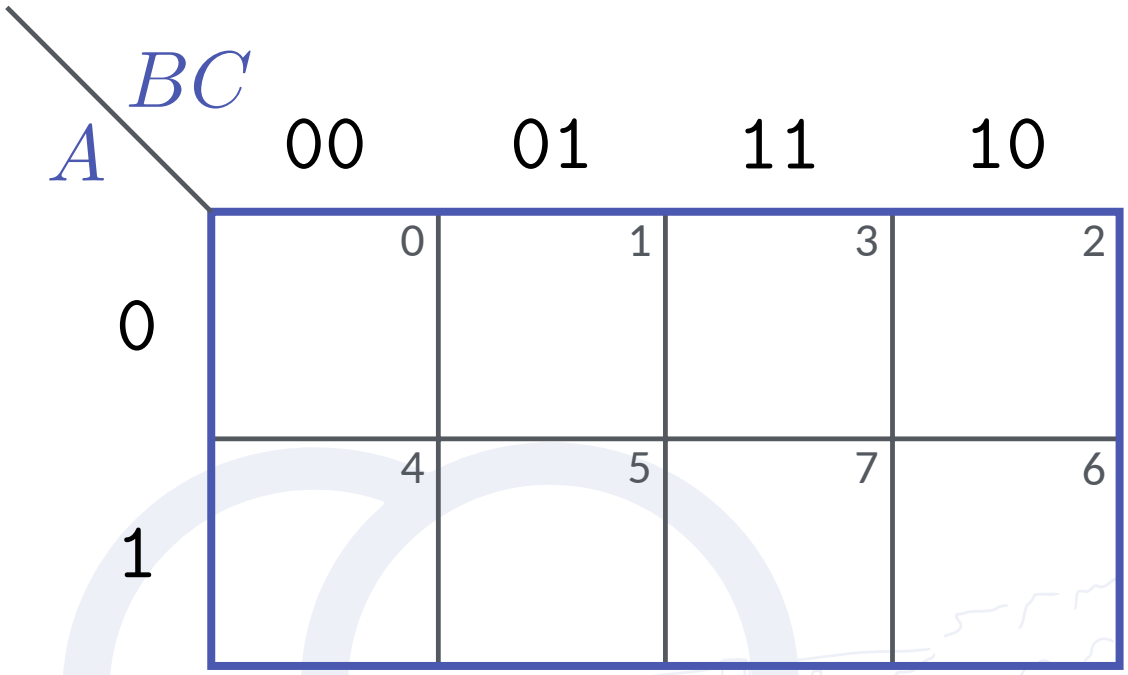
Der vereinfachte Term lautet $\neg A \vee (B \wedge C \wedge D)$. Die DNF hätte neun Minterme.

Arbeitsauftrag 91: KV-Training

Vereinfache mithilfe eines KV-Diagramms. Benutze die Vorlagen unten.

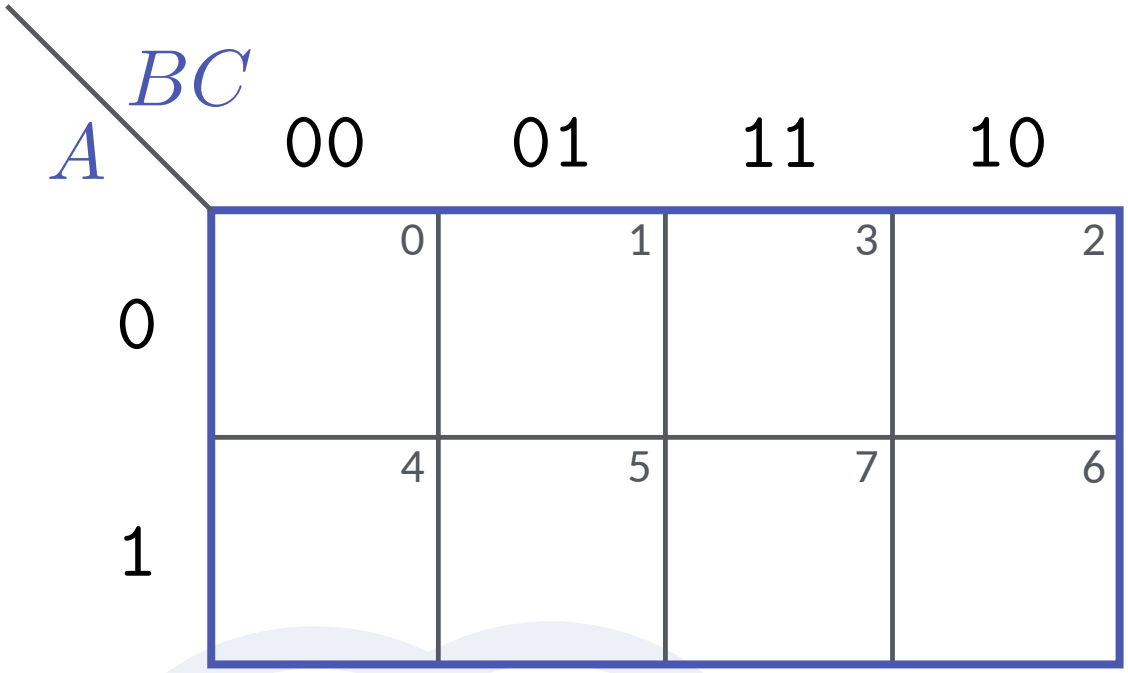
a) die DNF $(\neg A \wedge \neg B \wedge C) \vee (\neg A \wedge B \wedge C) \vee (A \wedge \neg B \wedge C) \vee (A \wedge B \wedge C)$

b) Drei Variablen A, B, C: Der Ausgang ist genau in den Zeilen 0, 2 und 5 der Belegungstabelle 1.
c) Vier Variablen A, B, C, D: Der Ausgang ist genau in den Zeilen 0, 1, 2, 3, 8, 9, 10 und 11 der Belegungstabelle 1.



Vorlage für a)

Gymnasium Ottweiler



Vorlage für b)

Gymnasium Ottweiler

AB \ CD	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

Vorlage für c)

Meine vereinfachten Terme:

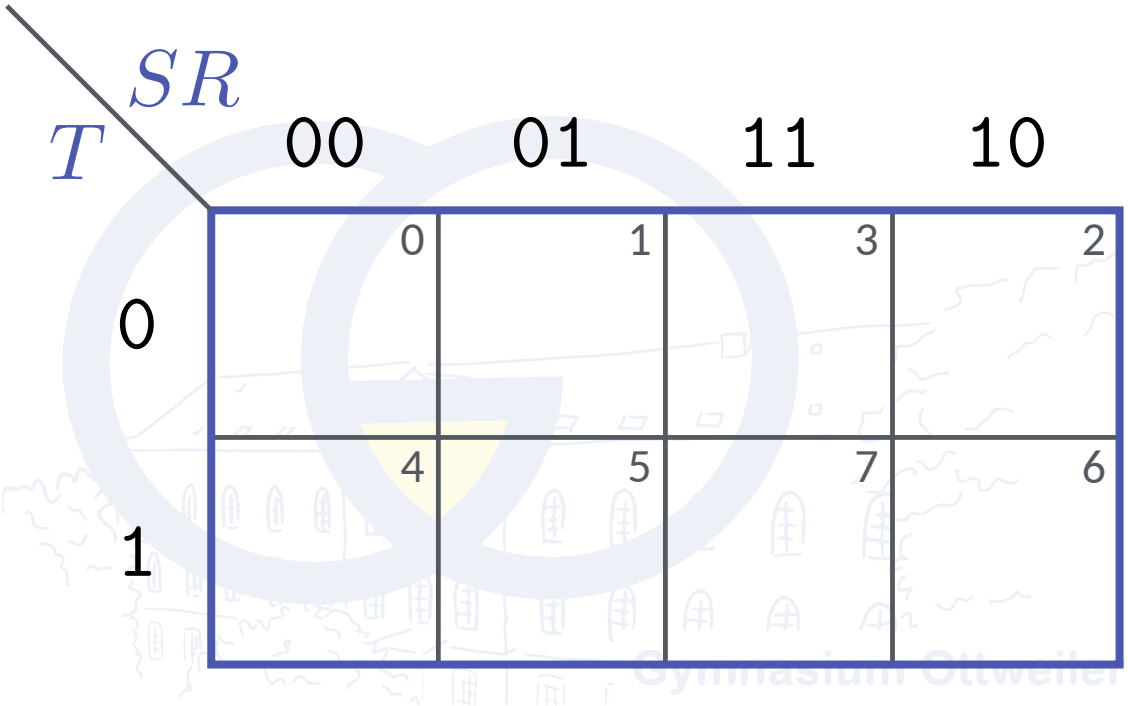
Merke

Das KV-Diagramm liefert eine **minimierte DNF**, die zur ursprünglichen DNF **äquivalent**, aber kürzer ist. Das ist wichtig für **kostengünstigere Schaltungen** (weniger Gatter).

2.8.6 Zurück zur Alarmanlage

Arbeitsauftrag 92: Ist die Alarmanlage minimal?

Erinnere dich an die Alarmanlage: Alarm $=$ (Tür $\wedge$ Scharf) $\vee$ Rauch. Ist dieser Term bereits minimal, oder lässt er sich mit einem KV-Diagramm weiter vereinfachen? Begründe deine Antwort in Worten. Ein Programm brauchst du nicht.



Vorlage: KV-Diagramm mit den Variablen T, S, R

Meine Begründung:

So kannst du es sagen

So sprichst du über KV-Diagramme – vom einfachen zum genauen Satz:

- *„Damit wird der Term kürzer."* → *„Das KV-Diagramm vereinfacht (minimiert) die DNF."*
- *„Die vier Einsen kreise ich ein."* → *„Die vier Felder werden zu einem Block zusammengefasst."*
- *„B ist hier egal."* → *„Im Block kommt B negiert und nicht negiert vor, deshalb fällt B weg."*

Arbeitsauftrag 93 (Zusatz): Vier Variablen, ein Block

Erstelle selbst eine Belegungstabelle mit **vier** Variablen A, B, C, D und mindestens vier 1-Zeilen, die sich zu einem **einzigen** Block zusammenfassen lassen. Gib die DNF und die minimierte Form an. Prüfe deine Minimierung mit Python. Tipp: Verwende vier geschachtelte Schleifen und eine Variable `aequivalent` wie im Abschnitt über Äquivalenz.

Meine Tabelle (als Liste der 1-Zeilen), DNF und minimierte Form:

Gymnasium Ottweiler

2.8.7 Weitere Übungen

Hier kannst du das Gelernte weiter üben. Die Punkte im Titel zeigen, wie schwer eine Aufgabe ist: ●○○ leicht, ●●○ mittel, ●●● knifflig. Fang mit den leichten an.

Arbeitsauftrag 94: Unterscheiden sich die Felder in genau einer Variablen? ●○○

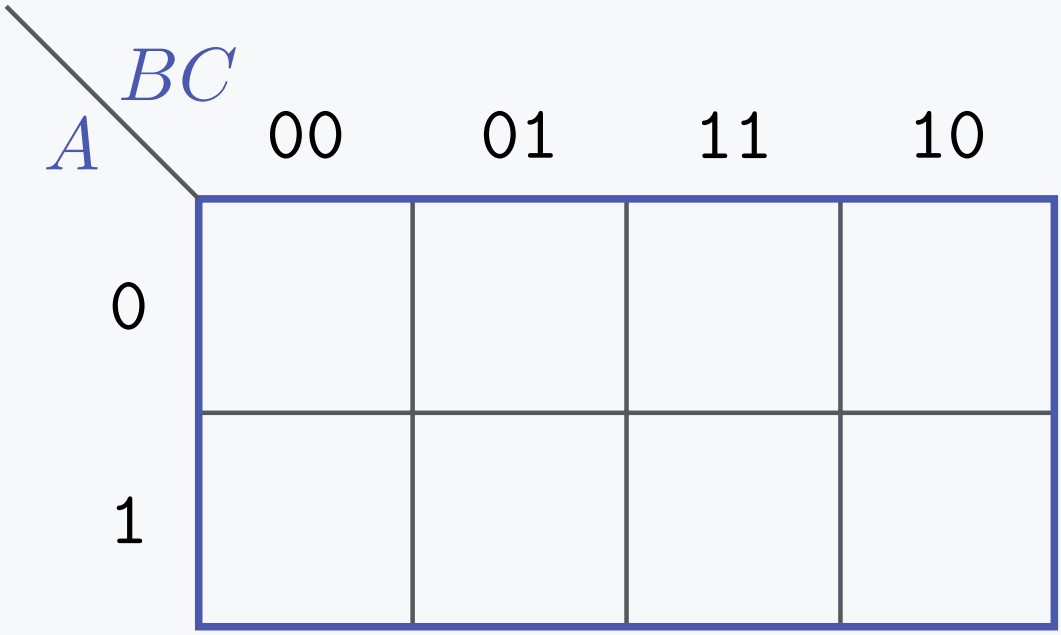
Im KV-Diagramm werden nur Felder zu einem Zweierblock zusammengefasst, die sich in **genau einer** Variablen unterscheiden. Gib für jedes Paar von Belegungen (A, B, C) an, in wie vielen Variablen sie sich unterscheiden und ob man sie zu einem Zweierblock zusammenfassen darf.

Belegung 1	Belegung 2	Anzahl verschiedener Variablen	Zweierblock erlaubt?
0 0 0	0 0 1		
0 1 1	1 1 1		
0 0 1	0 1 0		
1 1 0	1 1 1		
1 0 1	1 1 0		
0 1 0	1 1 0		

Arbeitsauftrag 95: Eine Tabelle ins KV-Diagramm übertragen ●○○

Übertrage die Belegungstabelle in das KV-Diagramm. Achte auf die Reihenfolge der Spalten: 00, 01, 11, 10.

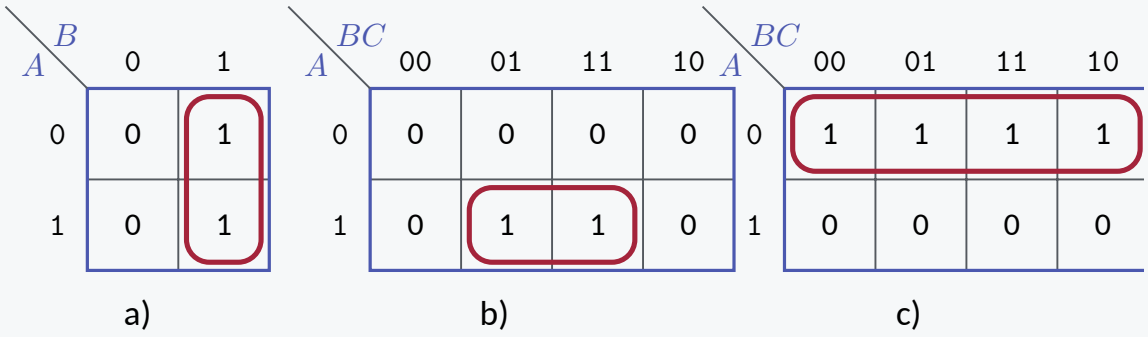
A	B	C	Ausgang
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



Vorlage zum Ausfüllen: KV-Diagramm für drei Variablen

Arbeitsauftrag 96: Blöcke ablesen ●○○

Gib zu jedem eingezeichneten Block den Term an.



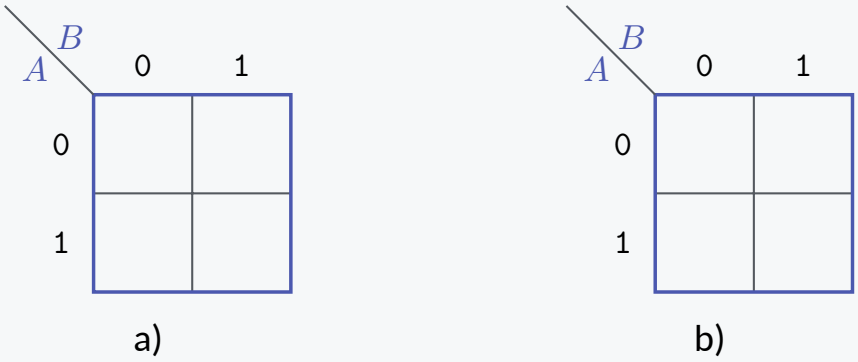
Drei KV-Diagramme mit eingezeichneten Blöcken

Deine Terme:

Arbeitsauftrag 97: Minimieren mit zwei Variablen ●○○

Vereinfache die DNF mithilfe eines KV-Diagramms.

a) $(A \wedge \neg B) \vee (A \wedge B)$
b) $(\neg A \wedge \neg B) \vee (\neg A \wedge B) \vee (A \wedge B)$



Vorlagen für a) und b)

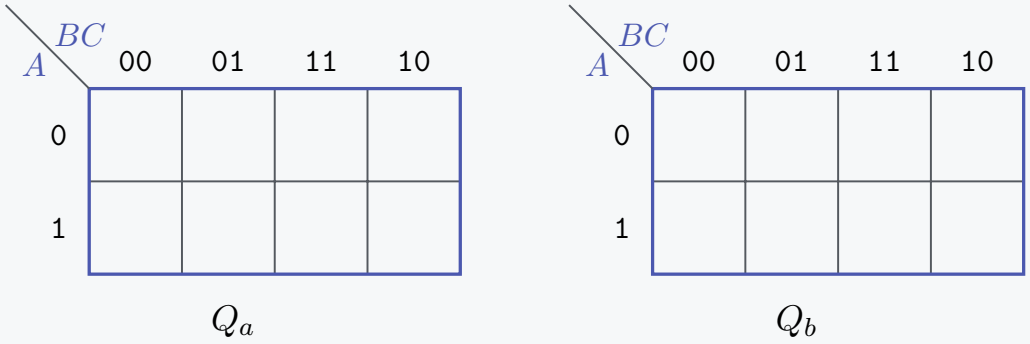
Deine vereinfachten Terme:

Gymnasium Ottweiler

Arbeitsauftrag 98: Minimieren mit drei Variablen ●●○

Die Tabelle enthält zwei Ausgänge Q_a und Q_b. Vereinfache beide mit je einem KV-Diagramm. Prüfe dein Ergebnis zu Q_b mit Python: Vergleiche für alle Belegungen deine vereinfachte Form mit der DNF.

A	B	C	Q_a	Q_b
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	0	1	1
1	1	1	1	1



Vorlagen für Q_a und Q_b

Deine vereinfachten Terme:

Arbeitsauftrag 99: Fehler im KV-Diagramm ●●○

Tom hat ein KV-Diagramm ausgefüllt und die drei Einsen mit einem Block umrandet. Daraus liest er den Term $\neg A$ ab.

A \ BC	00	01	11	10
0	0	1	1	1
1	0	0	0	0

Toms KV-Diagramm

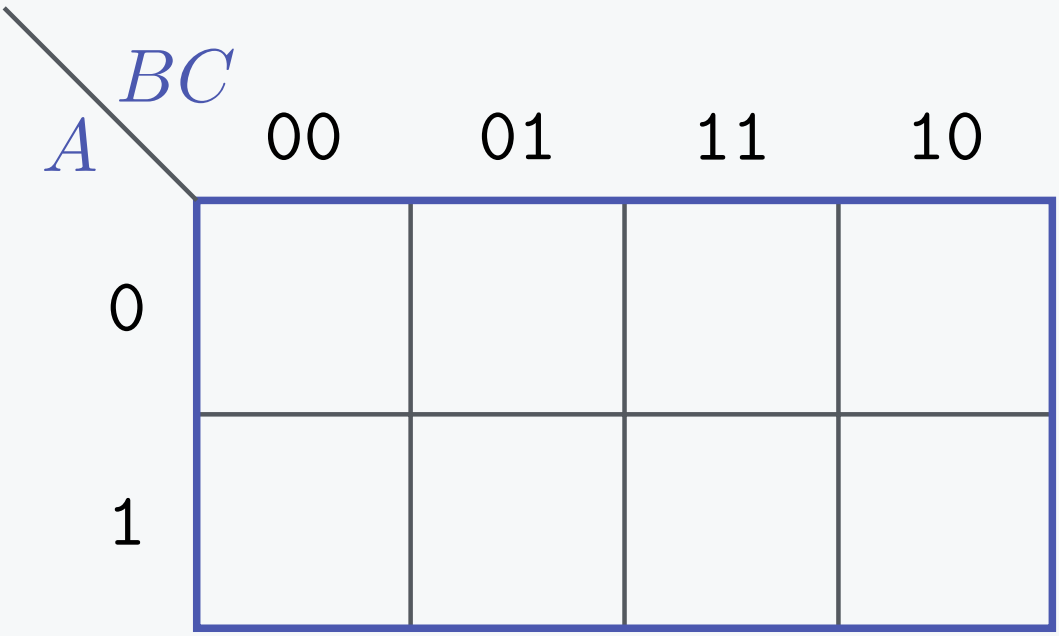
1. Erkläre, welche Regel Tom verletzt hat.
2. Zeige mit einer Belegung, dass sein Term $\neg A$ falsch ist.
3. Gib den richtigen vereinfachten Term an.

Deine Antwort:

Arbeitsauftrag 100: Eine sparsame Schaltung ●●○

Der Ausgang Q einer Schaltung soll genau bei diesen Belegungen 1 sein: $(0,1,1)$, $(1,0,0)$, $(1,0,1)$, $(1,1,0)$ und $(1,1,1)$. Die Werte stehen in der Reihenfolge A, B, C.

1. Wie viele Minterme hätte die DNF?
2. Vereinfache mit einem KV-Diagramm.
3. Skizziere die Schaltung zum vereinfachten Term und gib an, wie viele Gatter sie braucht.



Vorlage zum Ausfüllen: KV-Diagramm für drei Variablen

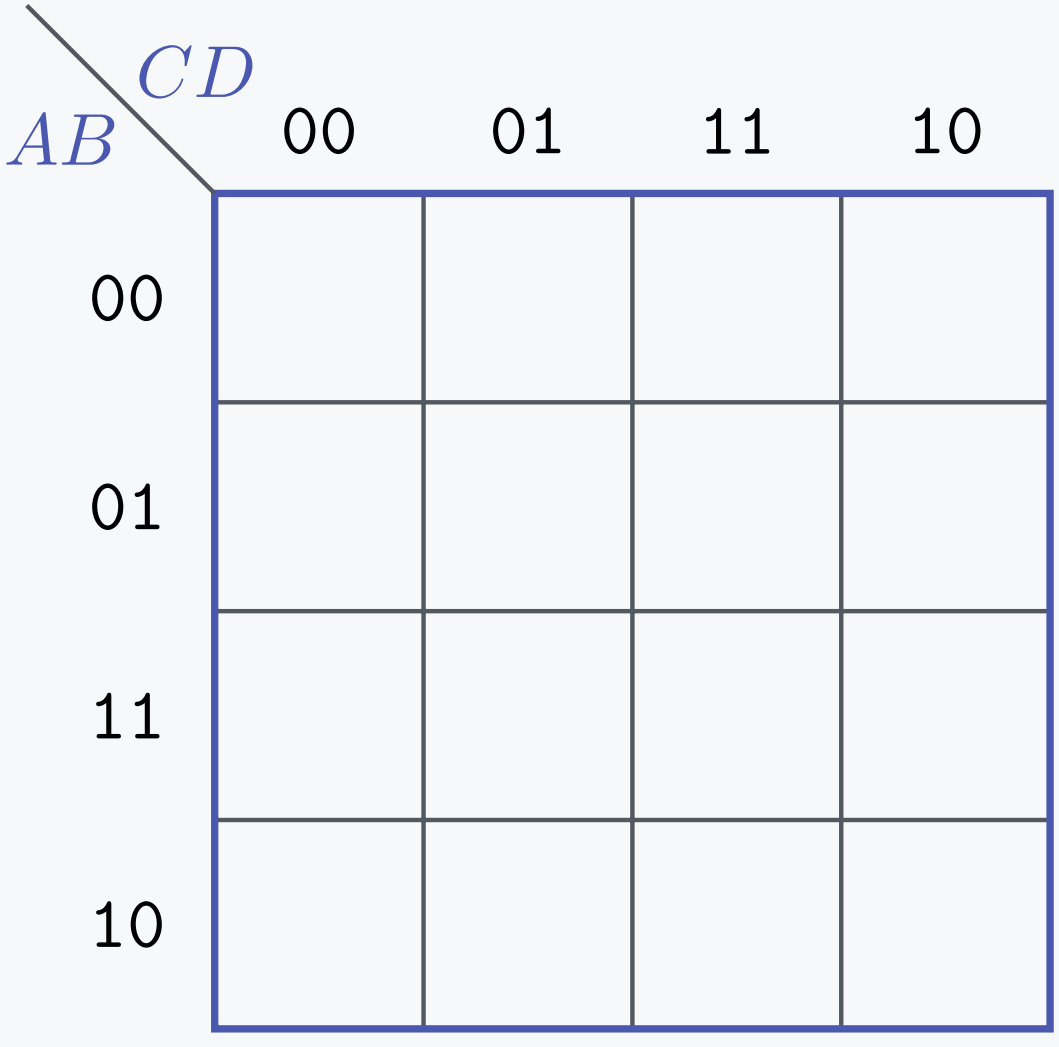
Dein Term und deine Skizze:

Gymnasium Ottweiler

Arbeitsauftrag 101: Vier Variablen ●●●

Vereinfache die DNF mit einem KV-Diagramm für vier Variablen. Prüfe dein Ergebnis mit Python.

$$(\neg A \wedge B \wedge \neg C \wedge \neg D) \vee (\neg A \wedge B \wedge \neg C \wedge D) \vee (A \wedge B \wedge \neg C \wedge \neg D)$$
$$\vee (A \wedge B \wedge \neg C \wedge D) \vee (A \wedge B \wedge C \wedge D)$$



Vorlage: KV-Diagramm für vier Variablen

Tipp

Die Zeilen des Diagramms geben A und B an, die Spalten C und D. Beide folgen der Reihenfolge 00, 01, 11, 10. Trage für jeden Minterm eine 1 ein: Für $\neg A \wedge B \wedge \neg C \wedge D$ zum Beispiel in die Zeile $AB = 01$ und die Spalte $CD = 01$.

Dein vereinfachter Term:

2.8.8 Das Wichtigste in Kürze

- Die **Kosten** einer Schaltung messen wir an der Anzahl der Gatter. Eine DNF ist oft viel teurer als nötig.
- Im **KV-Diagramm** sind die Belegungen im Gray-Code 00, 01, 11, 10 angeordnet. Benachbarte Felder unterscheiden sich in genau einer Variablen. Auch gegenüberliegende Ränder sind benachbart.
- Benachbarte Einsen werden zu rechteckigen **Blöcken** aus 1, 2, 4 oder 8 Feldern zusammengefasst – so groß und so wenige wie möglich. Blöcke dürfen sich überlappen und über den Rand gehen.
- Zu jedem Block gehört die Konjunktion der Variablen, die im Block **konstant** sind. Die Blockterme werden mit $\vee$ verbunden.
- Das Ergebnis ist eine **minimierte DNF**: äquivalent zur DNF, aber mit weniger Gattern.

Selbstcheck

- ☐ Ich kann die Kosten zweier äquivalenter Schaltungen über die Anzahl der Gatter vergleichen.
- ☐ Ich kann erklären, warum die Spalten im KV-Diagramm in der Reihenfolge 00, 01, 11, 10 stehen.
- ☐ Ich kann eine Belegungstabelle mit zwei, drei oder vier Variablen in ein KV-Diagramm übertragen.
- ☐ Ich kann möglichst große Blöcke bilden, auch über den Rand.
- ☐ Ich kann zu jedem Block den Term ablesen und den vereinfachten Term angeben.
- ☐ Ich kann mit Python prüfen, ob der vereinfachte Term zur DNF äquivalent ist.

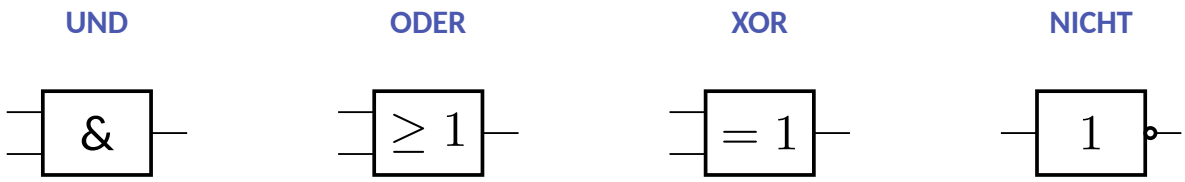
Gymnasium Ottweiler

2.9 Kapitel 2 kompakt: der Lernzettel

Leitidee

Dieser Lernzettel fasst das Kapitel auf wenigen Seiten zusammen. Nutze ihn zum Wiederholen vor einem Test. Ausführliche Erklärungen, Beispiele und Übungen findest du in den einzelnen Abschnitten.

2.9.1 Grundgatter, Belegungstabellen, Schaltungsentwurf (Abschnitte 2.1 und 2.2)



Die vier Grundgatter als IEC-Symbole. CircuitVerse zeigt sie als US-Symbole

Gatter	Zeichen	Term	Ausgang 1, wenn …
UND (AND)	`&`	$A \wedge B$	**alle** Eingänge 1 sind
ODER (OR)	`≥1`	$A \vee B$	**mindestens ein** Eingang 1 ist
XOR	`=1`	$A \oplus B$	die Eingänge **unterschiedlich** sind
NICHT (NOT)	`1` mit Kreis	$\neg A$	der Eingang 0 ist (Gegenteil)

A	B	$A \wedge B$	$A \vee B$	$A \oplus B$	$\neg A$
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

- **Belegungstabelle:** für jede Kombination der Eingangswerte der Ausgangswert. Bei n Eingängen 2^n Zeilen (3 Eingänge: 8 Zeilen). Reihenfolge: Die letzte Spalte wechselt in jeder Zeile, die davor alle 2 Zeilen, die davor alle 4 Zeilen.
- **Schaltung → Tabelle:** von links nach rechts, für jeden Gatterausgang eine **Hilfsspalte**.
- **Entwurf in vier Schritten:** Eingänge und Ausgang benennen (was bedeuten 1 und 0?), Belegungstabelle aufstellen, Gatter auswählen und verschalten (Term hilft), mit **allen** Belegungen testen.
- Beispiel Alarmanlage: Alarm $=$ (Tür $\wedge$ Scharf) $\vee$ Rauch. Was in der Klammer steht, rechnet ein eigenes Gatter zuerst.
- Kontrolle mit Python: `and`, `or`, `not`, XOR als `a != b`; alle Belegungen mit geschachtelten Schleifen `for a in [0, 1]:` usw. Ergebnis mit `int(...)` als 0 oder 1 ausgeben.

Typische Fehler: ODER und XOR verwechseln (Unterschied nur bei 1 1); eine Zeile der Tabelle vergessen; die Bedeutung von 0 und 1 nicht festlegen; Klammern vergessen ($(T \wedge S) \vee R$ ist nicht $T \wedge (S \vee R)$).

2.9.2 Binäre Addition, Halb- und Volladdierer (Abschnitte 2.3 und 2.4)

- **Binäre Addition** wie schriftlich im Dezimalsystem, von rechts nach links: $0 + 1 = 1$, $1 + 1 = 10$ (0 schreiben, 1 übertragen), $1 + 1 + 1 = 11$ (1 schreiben, 1 übertragen). Überträge in eine eigene Ü-Zeile.

```
      0 1 0 1     (= 5)
    + 0 0 1 1     (= 3)
Ü     1 1 1
    ---------
      1 0 0 0     (= 8)
```

- **Halbaddierer:** addiert **zwei** Bits. Summe $S = A \oplus B$ (XOR-Gatter), Übertrag $C = A \wedge B$ (UND-Gatter). „Halb“, weil er keinen eingehenden Übertrag verarbeiten kann.
- **Volladdierer:** addiert **drei** Bits A, B, C_{ein} und liefert S und C_{aus}. Aufbau: zwei Halbaddierer und ein ODER-Gatter: HA 1 rechnet $A + B$ (ergibt S_1, C_1), HA 2 rechnet $S_1 + C_{\text{ein}}$ (ergibt S, C_2), dann $C_{\text{aus}} = C_1 \vee C_2$. Kurz: $C_{\text{aus}}S$ ist die Summe der drei Bits als Binärzahl.
- **Carry-Ripple-Addierer:** ein Volladdierer pro Stelle; der ausgehende Übertrag einer Stelle ist der eingehende Übertrag der nächsthöheren Stelle. Ganz rechts ist der Übertrag 0. Der Übertrag „rieselt“ von rechts nach links; jede Stufe wartet auf die vorige. Vier Stellen ergeben bis zu fünf Ergebnisbits.
- Kontrolle mit Python: `int("0101", 2)` ergibt 5, `bin(8)` ergibt `'0b1000'`.

Typische Fehler: „1 + 1 = 2“ schreiben; Übertrag vergessen oder unter die falsche Stelle schreiben; den letzten Übertrag ganz links weglassen; beim Volladdierer ein UND statt des ODER-Gatters am Ende.

2.9.3 Rückkopplung und Flipflop (Abschnitt 2.5)

- **Kombinatorische Schaltung:** Ausgang hängt nur von den **aktuellen** Eingängen ab (Gatter, Addierer, Alarmanlage).
- **Flipflop:** merkt sich ein Bit; der Ausgang hängt auch von der **Vorgeschichte** ab. Grundlage schneller Speicher, z. B. Register.
- **Rückkopplung:** Ein Ausgang wird auf einen Eingang der eigenen Schaltung zurückgeführt.
- **SR-Flipflop:** zwei NOR-Gatter (ODER mit Kreis am Ausgang), **über Kreuz** zurückgekoppelt. Ausgänge Q und $\overline{Q}$ (auch `Q'`).
- $S = 1$, $R = 0$: **Setzen** ($Q = 1$). $S = 0$, $R = 1$: **Rücksetzen** ($Q = 0$). $S = 0$, $R = 0$: **Speichern** (Q bleibt wie vorher). $S = R = 1$: **verboten**.
- In Python ahmt eine Variable die Rückkopplung nach: `q = sr_flipflop(s, r, q)`.

Typische Fehler: bei $S = R = 0$ einfach 0 eintragen statt den alten Wert; S und R vertauschen.

2.9.4 Aussagenlogik und Termbäume (Abschnitt 2.6)

Name	Zeichen	Gatter	Python
Negation	$\neg A$	NICHT	`not a`
Konjunktion	$A \wedge B$	UND	`a and b`
Disjunktion	$A \vee B$	ODER	`a or b`

- **Induktive Definition:** (1) Jede Aussagenvariable ist ein Term. (2) Ist t ein Term, dann auch $\neg t$. (3) Sind t_1, t_2 Terme, dann auch $(t_1 \wedge t_2)$ und $(t_1 \vee t_2)$. (4) Nichts anderes ist ein Term.

- **Termbaum:** Blätter = Variablen, innere Knoten = Operatoren, Wurzel = Operator, der **zuletzt** ausgewertet wird. Beispiel $(A \wedge B) \vee \neg C$: Wurzel $\vee$, darunter $\wedge$ (mit A, B) und $\neg$ (mit C).
- **Term ↔ Schaltung:** Jeder innere Knoten ist ein Gatter ($\wedge$ UND, $\vee$ ODER, $\neg$ NICHT), die Blätter sind die Eingänge, die Wurzel ist das letzte Gatter. Was in einer Klammer steht, rechnet ein eigenes Gatter zuerst.

Typische Fehler: $\neg(A \vee B)$ und $\neg A \vee B$ verwechseln (das $\neg$ gilt nur für das, was direkt dahinter steht); die falsche Wurzel wählen; $\oplus$ in einen Term schreiben (gehört nicht zu den Regeln).

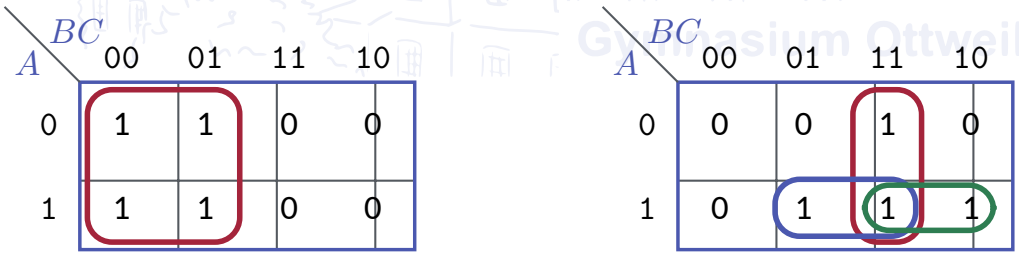
2.9.5 Äquivalenz und DNF (Abschnitt 2.7)

- **Äquivalent:** Zwei Terme liefern für **jede** Belegung denselben Wert, haben also dieselbe Belegungstabelle. Äquivalenz ist ein Begriff **über** Terme, kein Operator. Für „nicht äquivalent“ genügt **eine** Belegung, in der sie sich unterscheiden.
- Beispiel (De Morgan): $\neg(A \wedge B)$ ist äquivalent zu $\neg A \vee \neg B$, aber **nicht** zu $\neg A \wedge \neg B$.
- **Minterm:** Konjunktion **aller** Variablen; eine Variable mit dem Wert 0 bekommt ein $\neg$. Zur Zeile $A = 1$, $B = 0$, $C = 1$ gehört $A \wedge \neg B \wedge C$.
- **DNF:** Minterme aller Zeilen mit Ausgang 1, verbunden mit $\vee$. Beispiel XOR: Einsen bei $(0, 1)$ und $(1, 0)$, DNF $(\neg A \wedge B) \vee (A \wedge \neg B)$.
- Äquivalenz mit Python prüfen: beide Terme als Funktionen schreiben, `aequivalent = True` setzen, in geschachtelten Schleifen alle Belegungen vergleichen und bei `!=` auf `False` setzen.

Typische Fehler: $\neg$ an die Variablen mit Wert 1 setzen; eine Variable im Minterm vergessen; Zeilen mit Ausgang 0 verwenden; $\wedge$ und $\vee$ vertauschen.

2.9.6 KV-Diagramme (Abschnitt 2.8)

- **KV-Diagramm:** alle Belegungen in einem Raster, Spalten (und bei vier Variablen auch Zeilen) in der Reihenfolge **00, 01, 11, 10**. Benachbarte Felder unterscheiden sich in **genau einer** Variablen.
- Benachbarte Einsen zu **Blöcken** zusammenfassen: Rechtecke aus 1, 2, 4 oder 8 Feldern, nur Einsen, so **groß** und so **wenige** wie möglich; Überlappen ist erlaubt; jede 1 liegt in einem Block.
- **Term eines Blocks:** nur die Variablen, die im ganzen Block **konstant** sind (mit $\neg$ bei 0). Alle Blockterme mit $\vee$ verbinden. Ergebnis: **minimierte DNF**, äquivalent zur DNF, aber mit weniger Gattern.



Links: ein Viererblock mit $B = 0$, Ergebnis $\neg B$. Rechts: Mehrheit, drei Zweierblöcke, Ergebnis $(B \wedge C) \vee (A \wedge C) \vee (A \wedge B)$

Typische Fehler: Spalten in der Reihenfolge 00, 01, 10, 11 anlegen; diagonale Felder zusammenfassen; Dreierblöcke bilden; zu kleine Blöcke wählen (richtig, aber nicht minimal); eine 1 in keinem Block vergessen.

2.10 Rückblick auf Kapitel 2

Leitidee

Am Ende des Kapitels lohnt sich ein Blick zurück: Welche Fachbegriffe kennst du jetzt? Was kannst du schon sicher, was solltest du noch einmal üben? Die Tabellen helfen dir dabei, zum Beispiel bei der Vorbereitung auf eine Klassenarbeit.

2.10.1 Die Basisbegriffe

Begriff	Bedeutung
logische Schaltung	verarbeitet Eingangswerte 0 und 1 zu Ausgangswerten 0 und 1
Grundgatter	einfachste Bausteine: UND (AND), ODER (OR), XOR, NICHT (NOT)
Belegungstabelle	listet für jede Kombination der Eingangswerte die Ausgangswerte auf
Halbaddierer	addiert zwei Bits; liefert Summe S und Übertrag C
Volladdierer	addiert drei Bits (zwei Summanden und einen Übertrag); liefert Summe und Übertrag
Carry-Ripple-Addierer	hintereinandergeschaltete Volladdierer, die mehrstellige Binärzahlen addieren; der Übertrag wandert von Stelle zu Stelle
Rückkopplung	ein Ausgang wird wieder an einen Eingang zurückgeführt
Flipflop	Schaltung mit Rückkopplung, die ein Bit speichert
Konjunktion	UND-Verknüpfung $A \wedge B$: nur wahr, wenn beide wahr sind
Disjunktion	ODER-Verknüpfung $A \vee B$: wahr, wenn mindestens einer wahr ist
Negation	NICHT $\neg A$: kehrt den Wahrheitswert um
Termbaum	stellt den Aufbau eines Terms als Baum dar; die Wurzel ist der zuletzt ausgewertete Operator
Äquivalenz	Zwei Terme sind äquivalent, wenn sie für jede Belegung denselben Wert haben (gleiche Belegungstabelle).
Regeln von De Morgan	$\neg(A \wedge B)$ ist äquivalent zu $\neg A \vee \neg B$; $\neg(A \vee B)$ ist äquivalent zu $\neg A \wedge \neg B$
Disjunktive Normalform (DNF)	ODER-Verknüpfung der Minterme; jede 1-Zeile der Belegungstabelle ergibt einen Minterm (UND-Verknüpfung aller Variablen)
Karnaugh-Veitch-Diagramm	Anordnung der Belegungstabelle, in der benachbarte Felder sich in genau einer Variablen unterscheiden; dient zum Vereinfachen der DNF

2.10.2 Kompetenz-Check

Ich kann ...	sicher	teilweise	noch nicht
die Grundgatter nennen und skizzieren und ihre Belegungstabellen angeben.			
zu einer gegebenen Schaltung die Belegungstabelle erstellen.			
in einem Sachzusammenhang eine Schaltung entwerfen, auch mit einem Simulator.			
zwei natürliche Zahlen im Binärsystem addieren.			
Aufbau und Funktionsweise von Halb- und Volladdierer erläutern.			
einen Carry-Ripple-Addierer skizzieren und seine Funktionsweise erläutern.			
ein Flipflop skizzieren und erklären, wie es durch Rückkopplung ein Bit speichert.			
Schaltungen durch Terme beschreiben und Terme in Schaltungen übertragen.			
aussagenlogische Terme induktiv definieren und als Termbaum darstellen.			
die Begriffe Konjunktion, Disjunktion und Negation sachgemäß verwenden.			
zu einem Term die Belegungstabelle erstellen.			
erläutern, wann zwei Terme äquivalent sind.			
die Regeln von De Morgan nennen und anwenden.			
die DNF zu einem Term oder einer Belegungstabelle aufstellen.			
eine DNF mit bis zu vier Variablen mit einem KV-Diagramm vereinfachen.			

Gymnasium Ottweiler