

# Rechnersysteme und -netze

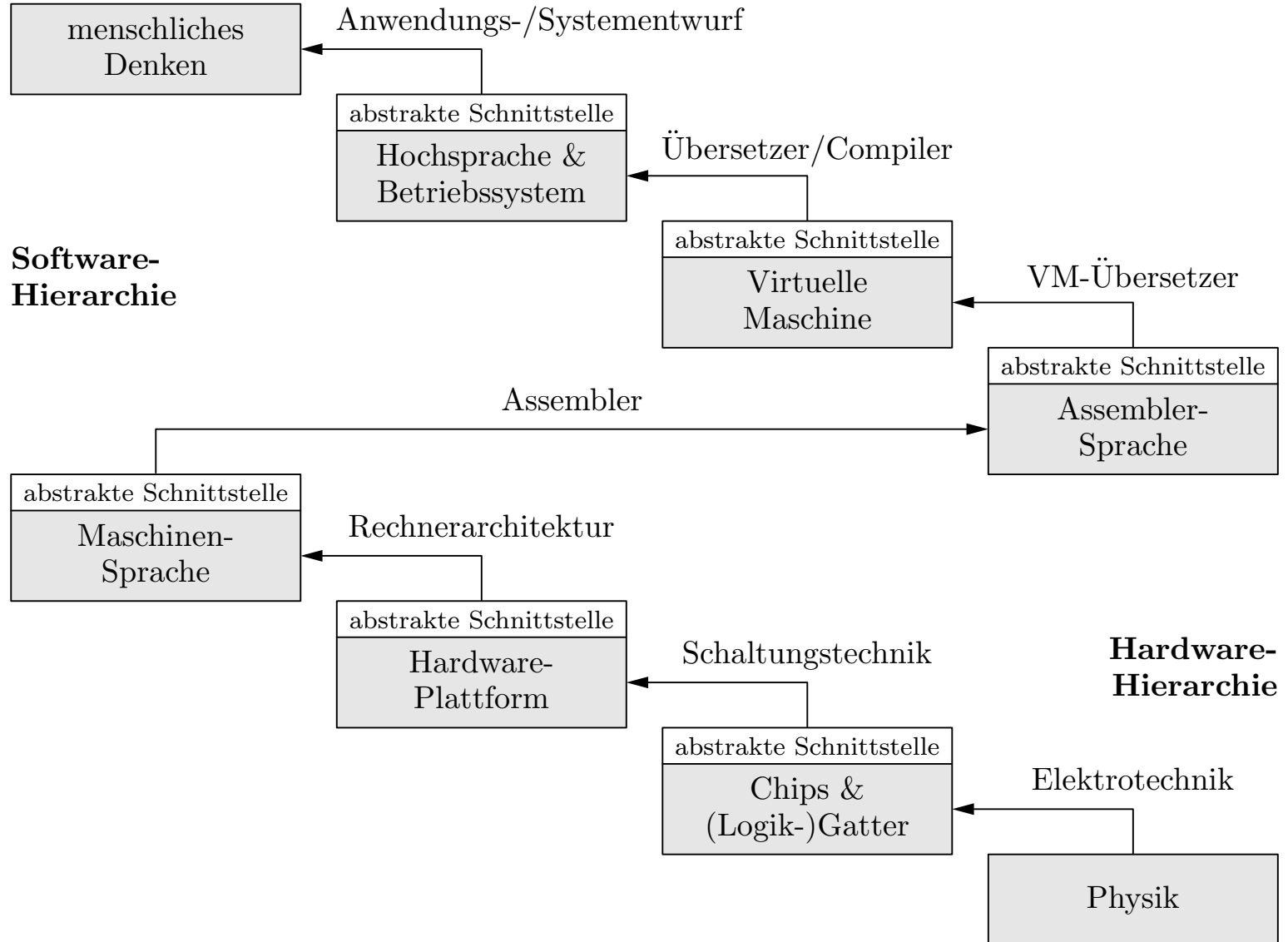
## Kapitel 2 - Schaltungstechnik Teil II

**Bastian Goldlücke**

Universität Konstanz  
WS 2020/21

---

# Rechnersysteme: Plan der Vorlesung



# Erinnerung: Schaltungstechnik I

- **Boolesche Algebra / Schaltalgebra**

- Definitionen, Operatoren, Schreibweisen etc.
- Wahrheitstafeln und Boolesche Funktionen
- Disjunktive und konjunktive Normalform
- Vollständige Operationenmengen

- **Gatterlogik**

- Elementare (Logik-)Gatter, Schaltzeichen für Gatter
- Zusammengesetzte Gatter: Schnittstelle und Implementierung
- Bitmustererkennung, Datenflußsteuerung, Multiplexer und Dekodierer

- **Implementierung durch Schaltkreise**

- Elektrotechnische Grundlagen, Schalter, Transistoren etc.
- Integrierte Schaltungen und ihre Herstellung

# Erinnerung: Boolesche Algebra

Algebraische Einkleidung der Aussagenlogik

[George Boole 1847]

Eine **Boolesche Algebra** ist eine Menge  $B$  mit dem Nullelement  $0$  und dem Einselement  $1$  (d.h.,  $0, 1 \in B$ ), auf der die zweistelligen Operationen **Konjunktion**  $\wedge$  und **Disjunktion**  $\vee$  sowie die einstellige Operation **Negation**  $\neg$  definiert sind.

(Redundantes) Axiomensystem (es sind  $a, b, c \in B$ ):

[Guiseppe Peano 1888]

Kommutativität  $a \wedge b = b \wedge a$

$a \vee b = b \vee a$

Assoziativität  $(a \wedge b) \wedge c = a \wedge (b \wedge c)$

$(a \vee b) \vee c = a \vee (b \vee c)$

Idempotenz  $a \wedge a = a$

$a \vee a = a$

Distributivität  $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$

$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$

Neutralität  $a \wedge 1 = a$

$a \vee 0 = a$

Extremalität  $a \wedge 0 = 0$

$a \vee 1 = 1$

Involution  $\neg\neg a = a$

[Augustus De Morgan 1860]

De Morgan  $\neg(a \wedge b) = \neg a \vee \neg b$

$\neg(a \vee b) = \neg a \wedge \neg b$

Komplementarität  $a \wedge \neg a = 0$

$a \vee \neg a = 1$

Dualität  $\neg 0 = 1$

$\neg 1 = 0$

Absorption  $a \vee (a \wedge b) = a$

$a \wedge (a \vee b) = a$

# Inhalt

## **1 Minimierung Boolescher Formeln**

- 1.1 Äquivalenzumformungen
- 1.2 Karnaugh–Veitch-Diagramme
- 1.3 Quine–McCluskey-Algorithmus
- 1.4 Petricks Algorithmus

## **2 Programmierbare Logikarrays**

- 2.1 Beispiel: Originalfunktion vs. disjunktive Normalform
- 2.2 Allgemeine Struktur
- 2.3 Hardware-Implementation

## **3 Hardware-Beschreibungssprache (HDL)**

- 3.1 Motivation: Hardware-Simulation
- 3.2 Beispiel: Aufbau eines AND-Gatters
- 3.3 Schnittstelle und Implementierung
- 3.4 Simulationsumgebung für diese Vorlesung

# Inhalt

## **1 Minimierung Boolescher Formeln**

- 1.1 Äquivalenzumformungen
- 1.2 Karnaugh–Veitch-Diagramme
- 1.3 Quine–McCluskey-Algorithmus
- 1.4 Petricks Algorithmus

## **2 Programmierbare Logikarrays**

- 2.1 Beispiel: Originalfunktion vs. disjunktive Normalform
- 2.2 Allgemeine Struktur
- 2.3 Hardware-Implementation

## **3 Hardware-Beschreibungssprache (HDL)**

- 3.1 Motivation: Hardware-Simulation
- 3.2 Beispiel: Aufbau eines AND-Gatters
- 3.3 Schnittstelle und Implementierung
- 3.4 Simulationsumgebung für diese Vorlesung

# Semantische und syntaktische Äquivalenz

Seien  $\varphi$  und  $\psi$  zwei Boolesche Formeln.

## Semantische Äquivalenz

Es gilt  $\varphi \equiv \psi$ ,

wenn für alle Belegungen der in  $\varphi$  und  $\psi$  auftretenden Variablen die beiden Formeln den gleichen (Wahrheits-)Wert haben.

## Syntaktische Äquivalenz

Es gilt  $\varphi = \psi$ ,

wenn  $\varphi$  durch Äquivalenzumformungen (mit Hilfe der Booleschen Axiome) in  $\psi$  umgeformt werden kann.

### Beispiel: Kontraposition

Sei  $\varphi \hat{=} a \rightarrow b$  und  $\psi \hat{=} \bar{b} \rightarrow \bar{a}$ .

## Semantische Äquivalenz

Var.		$\varphi$	$\psi$
$a$	$b$	$a \rightarrow b$	$\bar{b} \rightarrow \bar{a}$
0	0	1	1
0	1	1	1
1	0	0	0
1	1	1	1

## Syntaktische Äquivalenz

$$a \rightarrow b = \bar{a} \vee b \quad (\text{Def. Implikation})$$

$$= b \vee \bar{a} \quad (\text{Kommutativität})$$

$$= \bar{\bar{b}} \vee \bar{a} \quad (\text{Involution})$$

$$= \bar{b} \rightarrow \bar{a} \quad (\text{Def. Implikation})$$

# Semantische und syntaktische Äquivalenz

## Semantische Äquivalenz

- Vorteile:
  - (Fast) “Ohne Nachdenken” zu prüfen.
  - Abschluß der Äquivalenzprüfung garantiert.
  - Kann die Booleschen Axiome selbst prüfen.  
(Die Booleschen Axiome sind semantische Äquivalenzen.)
- Nachteile:
  - Aufwand exponentiell in der Zahl der Variablen,  
Teil des Aufwandes kann redundant sein.

## Syntaktische Äquivalenz

- Vorteile:
  - Nachweis oft wesentlich kürzer:  
wenige Schritte können das Ergebnis liefern.
- Nachteile:
  - „Richtige“ Umformungen sind oft schwer zu finden.
  - Abschluß der Äquivalenzprüfung nicht garantiert.  
(kein Weg gefunden  $\nRightarrow$  Formeln sind nicht äquivalent)



# Semantische und syntaktische Äquivalenz

- Idee der Booleschen Algebra/Schaltalgebra als mathematische Struktur zur Einkleidung der Aussagenlogik:

**Die Axiome der Booleschen Algebra/Schaltalgebra erlauben es, alle (in der Aussagenlogik) semantisch geltenden Äquivalenzen auf syntaktischem Wege abzuleiten/zu prüfen.**

- Denn es gilt:

**Zwei Boolesche Formeln sind genau dann semantisch äquivalent, wenn sie syntaktisch äquivalent sind.**

(Die Boolesche Algebra ist bzgl. Äquivalenzen **korrekt** und **vollständig**.)

- Daher: Die natürlichere, aber aufwendiger zu prüfende semantische Äquivalenz kann durch die syntaktische Äquivalenz ersetzt werden.
- Wegen des exponentiellen Aufwandes der Prüfung der semantischen Äquivalenz (Durchlaufen aller Variablenbelegungen) sind bestimmte Äquivalenzen (speziell mit vielen Variablen) praktisch nur auf syntaktischem Wege zu überprüfen.

# Minimierung Boolescher Formeln: Äquivalenzen

## Syntaktische Vereinfachung einer Booleschen Formel

$$(\bar{x} \wedge y \wedge \bar{z}) \vee (\bar{x} \wedge y \wedge z) \vee (x \wedge y \wedge \bar{z})$$

$$\begin{array}{l} 3 \times \text{Kommutativität} \\ \underline{=} \end{array} (y \wedge \bar{x} \wedge \bar{z}) \vee (y \wedge \bar{x} \wedge z) \vee (y \wedge x \wedge \bar{z})$$

$$\begin{array}{l} \text{Distributivität} \\ \underline{=} \end{array} (y \wedge \bar{x} \wedge \bar{z}) \vee (y \wedge ((\bar{x} \wedge z) \vee (x \wedge \bar{z})))$$

$$\begin{array}{l} \text{Distributivität} \\ \underline{=} \end{array} y \wedge ((\bar{x} \wedge \bar{z}) \vee (\bar{x} \wedge z) \vee (x \wedge \bar{z}))$$

$$\begin{array}{l} \text{Distributivität} \\ \underline{=} \end{array} y \wedge ((\bar{x} \wedge (\bar{z} \vee z)) \vee (x \wedge \bar{z}))$$

$$\begin{array}{l} \text{Kommutativität} \\ \text{Komplementarität} \\ \underline{=} \end{array} y \wedge ((\bar{x} \wedge 1) \vee (x \wedge \bar{z}))$$

$$\begin{array}{l} \text{Neutralität} \\ \underline{=} \end{array} y \wedge (\bar{x} \vee (x \wedge \bar{z}))$$

$$\begin{array}{l} \text{Distributivität} \\ \underline{=} \end{array} y \wedge ((\bar{x} \vee x) \wedge (\bar{x} \vee \bar{z}))$$

$$\begin{array}{l} \text{Kommutativität} \\ \text{Komplementarität} \\ \underline{=} \end{array} y \wedge (1 \wedge (\bar{x} \vee \bar{z}))$$

$$\begin{array}{l} \text{Kommutativität} \\ \underline{=} \end{array} y \wedge ((\bar{x} \vee \bar{z}) \wedge 1)$$

$$\begin{array}{l} \text{Neutralität} \\ \underline{=} \end{array} y \wedge (\bar{x} \vee \bar{z})$$

Bemerkung: Eigentlich ist hier an mehreren Stellen auch die Nutzung der Assoziativität nötig, was aber aus Platzgründen nicht notiert wurde.

# Minimierung Boolescher Formeln: Transformationen

- Offensichtliche **Probleme dieses Ansatzes**:
  - Es fehlt eine klare, nachvollziehbare Strategie zur Vereinfachung, d.h., eine Strategie, wann und wie die verschiedenen Gesetze der Booleschen Algebra angewandt werden sollten.
  - Es kann sein, daß man zu Zeiten sogenannte „Heureka“-Schritte benötigt (griech.: εὕρηκα – „Ich habe es [die Lösung] gefunden“; bekannt durch Archimedes von Syrakus, Entdeckung des Archimedischen Prinzips).
- Besser wäre ein **systematisches Vereinfachungsverfahren**.
- Wir werden im folgenden zwei solcher Verfahren betrachten:
  - **Karnaugh–Veitch-Diagramme**  
[Edward W. Veitch 1952, Maurice Karnaugh 1953]
  - **Quine–McCluskey-Algorithmus**  
[Willard V.O. Quine 1955, Edward J. McCluskey 1956]

# Minimierung: Karnaugh–Veitch-Diagramme

- **Karnaugh–Veitch-Diagramme** sind tabellarische Darstellungen Boolescher Funktionen (wie Wahrheitstafeln, nur andere Auflistung der Funktionswerte).
- Ein Karnaugh–Veitch-Diagramm einer Booleschen Funktion mit  $n$  Argumenten hat  $2^n$  Felder: ein Feld für jede Wertekombination der Argumentvariablen. In diese Felder wird der zugehörige Funktionswert eingetragen.
- Die Felder/Argumentwertekombinationen werden so angeordnet, daß ein Übergang zu einem Nachbarfeld, egal in welcher Richtung, den Wert nur genau eines Argumentes/einer Variablen ändert.
- Eine Anordnung von Binärzahlen in einer solchen Reihenfolge heißt **Gray-Code** [Frank Gray 1953].
- Beispiel für zwei Variablen:      00      01      11      10

Bemerkung: Diese Anordnung ist bis auf zyklische Vertauschungen und Spiegelungen eindeutig. Hier alle möglichen Varianten:

00	01	11	10	10	11	01	00
10	00	01	11	11	01	00	10
11	10	00	01	01	00	10	11
01	11	10	00	00	10	11	01

# Minimierung: Karnaugh–Veitch-Diagramme

- Karnaugh–Veitch-Diagramm einer Funktion mit **zwei Variablen**:

$$O = A \oplus B = (A \wedge \bar{B}) \vee (\bar{A} \wedge B)$$

		A	
		0	1
B	0	0	1
	1	1	0

oder

				AB			
				00	01	11	10
				0	1	0	1

- Karnaugh–Veitch-Diagramm einer Funktion mit **drei Variablen**:

$$O = (A \wedge B \wedge \bar{C}) \vee (A \wedge \bar{B} \wedge \bar{C}) \vee (A \wedge \bar{B} \wedge C)$$

		AB			
		00	01	11	10
C	0	0	0	1	1
	1	0	0	0	1

- Man beachte: Benachbarte Felder unterscheiden sich nur im Wert einer Variable.

# Minimierung: Karnaugh–Veitch-Diagramme

- Karnaugh–Veitch-Diagramm einer Funktion mit **vier Variablen**:

$$O = (\bar{A} \wedge \bar{B} \wedge C \wedge D) \vee (\bar{A} \wedge B \wedge C \wedge \bar{D}) \vee (A \wedge B \wedge \bar{C} \wedge \bar{D})$$

		AB			
		00	01	11	10
CD	00	0	0	1	0
	01	0	0	0	0
	11	1	0	0	0
	10	0	1	0	0

- Man beachte: Benachbarte Felder unterscheiden sich nur im Wert einer Variable, da auf beiden Achsen ein Gray-Code verwendet wird.
- Dies gilt sogar über den Rand des Diagramms hinaus:  
Felder in der ersten und letzten Spalte der gleichen Zeile sowie  
Felder in der ersten und letzten Zeile der gleichen Spalte  
unterscheiden sich ebenfalls nur im Wert einer Variable.

# Minimierung: Karnaugh–Veitch-Diagramme

- Sind zwei benachbarte Felder eines Karnaugh–Veitch-Diagramms beide 1, so zeigt dies an, daß eine bestimmte **Variable** (nämlich die, deren Wert sich von dem einen Feld zum anderen ändert) in der ursprünglichen Funktion **irrelevant** ist.

- **Beispiel:**  $O = (\bar{A} \wedge B \wedge \bar{C} \wedge D) \vee (A \wedge B \wedge \bar{C} \wedge D)$

		AB			
		00	01	11	10
CD	00	0	0	0	0
	01	0	1	1	0
	11	0	0	0	0
	10	0	0	0	0

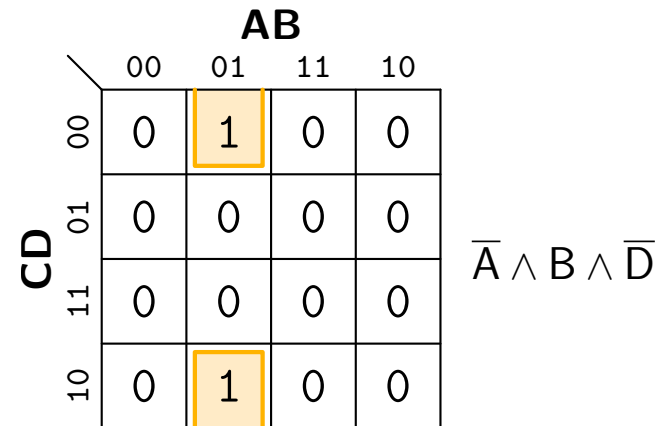
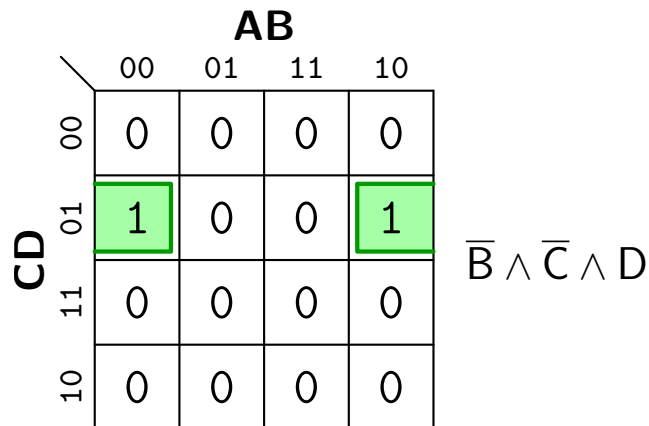
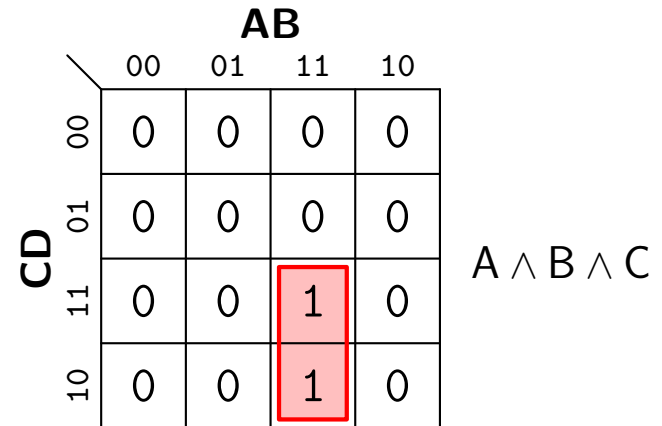
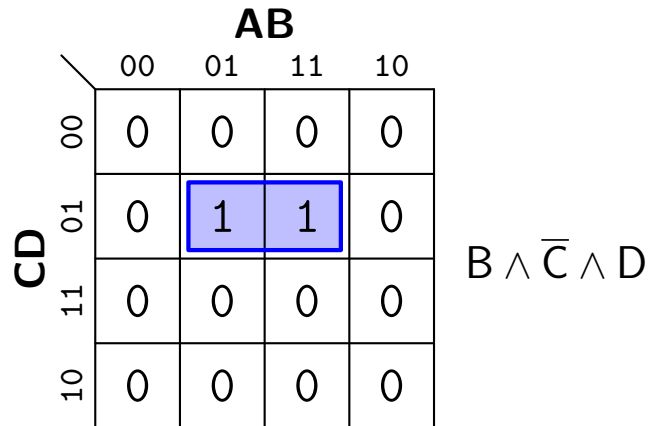
Die Variable  $A$  ist in  $O$  irrelevant:

$$\begin{aligned} O &= (\bar{A} \wedge B \wedge \bar{C} \wedge D) \vee (A \wedge B \wedge \bar{C} \wedge D) \\ &= (B \wedge \bar{C} \wedge D) \wedge (\bar{A} \vee A) \\ &= B \wedge \bar{C} \wedge D \wedge 1 \\ &= B \wedge \bar{C} \wedge D \end{aligned} \quad (\text{Achtung: verkürzte Ableitung!})$$

- Der minimierte Ausdruck  $B \wedge \bar{C} \wedge D$  gibt die **Lage** der vereinigten bzw. zusammengefaßten Felder des Diagramms an.
- Er kann aus dem eingezeichneten blauen Rechteck abgelesen werden, denn er besteht aus den Variablen, die nur einen Wert annehmen.

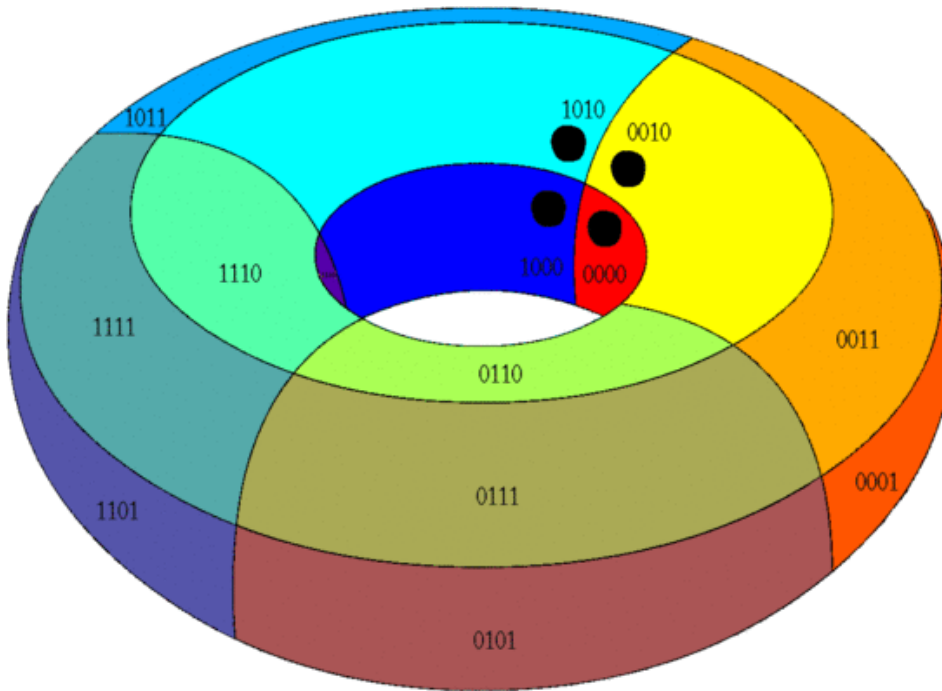
# Minimierung: Karnaugh–Veitch-Diagramme

- Beim Zusammenfassen von Feldern darf auch der Rand überschritten werden, da sich auch in diesem Fall der Wert nur einer Variable ändert.





# Minimierung: Karnaugh–Veitch-Diagramme



●	0000	0100	1100	●
0001	0101	1101	1001	
0011	0111	1111	1011	
●	0010	0110	1110	●

[www.wikipedia.org](http://www.wikipedia.org)

- Da von der ersten zur letzten Zeile/Spalte nur eine Variable ihren Wert ändert, kann ein Karnaugh–Veitch-Diagramm als Oberfläche eines Torus aufgefaßt werden.

# Minimierung: Karnaugh–Veitch-Diagramme

- Ein Zusammenfassen von  $2 = 2^1$  Feldern ist nur ein Spezialfall. Allgemein kann jedes Rechteck mit  $2^k$  Feldern,  $k \geq 1$ , zusammengefaßt werden. Ein solches Zusammenfassen eliminiert  $k$  Variablen.

		AB			
		00	01	11	10
CD	00	0	0	0	0
	01	0	0	0	0
	11	1	1	1	1
	10	0	0	0	0

$C \wedge D$

		AB			
		00	01	11	10
CD	00	0	0	0	0
	01	1	1	0	0
	11	1	1	0	0
	10	0	0	0	0

$\bar{A} \wedge D$

		AB			
		00	01	11	10
CD	00	0	0	0	0
	01	0	0	0	0
	11	1	0	0	1
	10	1	0	0	1

$\bar{B} \wedge C$

		AB			
		00	01	11	10
CD	00	0	0	1	1
	01	0	0	0	0
	11	0	0	0	0
	10	0	0	1	1

$A \wedge \bar{D}$

# Minimierung: Karnaugh–Veitch-Diagramme

- Ein Zusammenfassen von  $2 = 2^1$  Feldern ist nur ein Spezialfall.  
Allgemein kann jedes Rechteck mit  $2^k$  Feldern,  $k \geq 1$ , zusammengefaßt werden.  
Ein solches Zusammenfassen eliminiert  $k$  Variablen.

		AB				
		00	01	11	10	
CD	00	0	0	0	0	C
	01	0	0	0	0	
	11	1	1	1	1	
	10	1	1	1	1	

		AB				
		00	01	11	10	
CD	00	0	0	1	1	A
	01	0	0	1	1	
	11	0	0	1	1	
	10	0	0	1	1	


		AB				
		00	01	11	10	
CD	00	1	0	0	1	B
	01	1	0	0	1	
	11	1	0	0	1	
	10	1	0	0	1	

		AB				
		00	01	11	10	
CD	00	1	1	1	1	D
	01	0	0	0	0	
	11	0	0	0	0	
	10	1	1	1	1	

# Minimierung: Karnaugh–Veitch-Diagramme


- **Achtung:** Nicht alle Rechtecke können zu einem Term zusammengefaßt werden!

		AB			
		00	01	11	10
CD	00	0	0	0	0
	01	0	1	1	1
	11	0	1	1	1
	10	0	0	0	0



- Die Seitenlängen/Felderzahlen der Rechtecke müssen Zweierpotenzen sein, also etwa 1, 2, 4 oder 8.
- Die Breite 3 bzw. die Anzahl 6 sind keine Zweierpotenzen, das Rechteck folglich unzulässig.

		AB			
		00	01	11	10
CD	00	0	0	0	0
	01	0	1	1	1
	11	0	1	1	1
	10	0	0	0	0



		AB			
		00	01	11	10
CD	00	0	0	0	0
	01	0	1	1	1
	11	0	1	1	1
	10	0	0	0	0

Vereinigung ist nicht maximal!  
Ebenfalls zu vermeiden!

- Rechtecke  $3 \times 2$  und  $3 \times 3$  müssen in Quadrate  $2 \times 2$  zerlegt werden.

# Minimierung: Karnaugh–Veitch-Diagramme

- Zu minimierende Funktion und ihr KV-Diagramm:

$$\begin{aligned}
 O &= (A \wedge B \wedge \bar{C} \wedge \bar{D}) \vee (A \wedge \bar{B} \wedge \bar{C} \wedge \bar{D}) \\
 &\vee (A \wedge B \wedge \bar{C} \wedge D) \vee (A \wedge \bar{B} \wedge \bar{C} \wedge D) \\
 &\vee (A \wedge \bar{B} \wedge C \wedge D) \vee (\bar{A} \wedge B \wedge C \wedge \bar{D}) \\
 &\vee (A \wedge B \wedge C \wedge \bar{D}) \vee (A \wedge \bar{B} \wedge C \wedge \bar{D})
 \end{aligned}$$

		AB			
		00	01	11	10
CD	00	0	0	1	1
	01	0	0	1	1
	11	0	0	0	1
	10	0	1	1	1

- **1. Schritt:** Finde alle möglichen maximalen Zusammenfassungen von Feldern.  
(maximal: nicht Teil einer größeren Zusammenfassung mit mehr Feldern)  
(aber: Zusammenfassungen dürfen sich überlappen!)

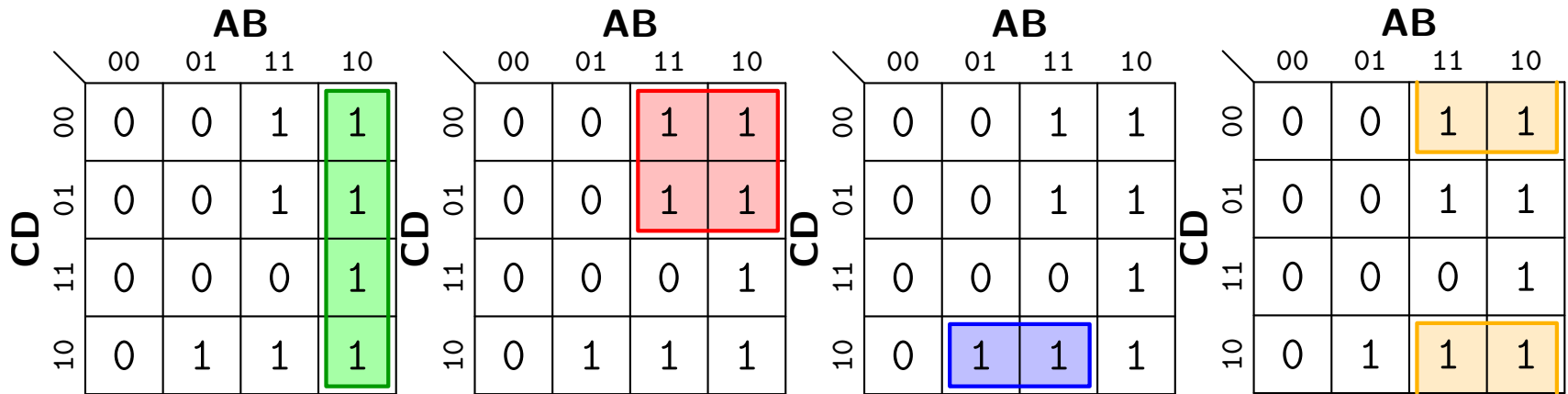
		AB			
		00	01	11	10
CD	00	0	0	1	1
	01	0	0	1	1
	11	0	0	0	1
	10	0	1	1	1

		AB			
		00	01	11	10
CD	00	0	0	1	1
	01	0	0	1	1
	11	0	0	0	1
	10	0	1	1	1

		AB			
		00	01	11	10
CD	00	0	0	1	1
	01	0	0	1	1
	11	0	0	0	1
	10	0	1	1	1

		AB			
		00	01	11	10
CD	00	0	0	1	1
	01	0	0	1	1
	11	0	0	0	1
	10	0	1	1	1

# Minimierung: Karnaugh–Veitch-Diagramme

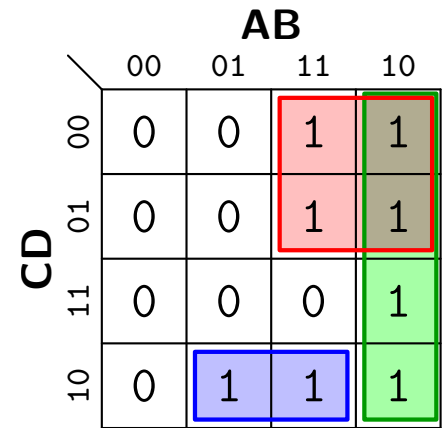


## - 2. Schritt:

Wähle möglichst wenige Zusammenfassungen, die alle Einsen abdecken.  
(Achtung: Hier kann es nötig sein, Einzelfelder einzubeziehen!)

- **3. Schritt:** Sammle die benötigten Ausdrücke  
(werden aus der Lage der gewählten Zusammenfassungen  
und ggf. noch benötigten Einzelfeldern abgelesen).

$$\begin{aligned}
 O &= (A \wedge \overline{C}) && \text{(rot)} \\
 &\vee (A \wedge \overline{B}) && \text{(grün)} \\
 &\vee (B \wedge C \wedge \overline{D}) && \text{(blau)}
 \end{aligned}$$



# Minimierung: Karnaugh–Veitch-Diagramme

- Zu minimierende Funktion und ihr KV-Diagramm:

$$\begin{aligned}
 O &= (A \wedge B \wedge \bar{C} \wedge \bar{D}) \vee (A \wedge \bar{B} \wedge \bar{C} \wedge \bar{D}) \\
 &\vee (A \wedge B \wedge \bar{C} \wedge D) \vee (A \wedge \bar{B} \wedge \bar{C} \wedge D) \\
 &\vee (A \wedge \bar{B} \wedge C \wedge D) \vee (\bar{A} \wedge B \wedge C \wedge \bar{D}) \\
 &\vee (A \wedge B \wedge C \wedge \bar{D}) \vee (A \wedge \bar{B} \wedge C \wedge \bar{D})
 \end{aligned}$$

		AB			
		00	01	11	10
CD	00	0	0	1	1
	01	0	0	1	1
	11	0	0	0	1
	10	0	1	1	1

- Natürlich kann die Minimierung nicht nur durch Zusammenfassen von Einsen, sondern, unter Rückgriff auf die Dualität der Booleschen Gesetze bzw. die Dualität der disjunktiven und der konjunktiven Normalform, auch durch **Zusammenfassen von Nullen** durchgeführt werden.

		AB			
		00	01	11	10
CD	00	0	0	1	1
	01	0	0	1	1
	11	0	0	0	1
	10	0	1	1	1

		AB			
		00	01	11	10
CD	00	0	0	1	1
	01	0	0	1	1
	11	0	0	0	1
	10	0	1	1	1

		AB			
		00	01	11	10
CD	00	0	0	1	1
	01	0	0	1	1
	11	0	0	0	1
	10	0	1	1	1

		AB			
		00	01	11	10
CD	00	0	0	1	1
	01	0	0	1	1
	11	0	0	0	1
	10	0	1	1	1

# Minimierung: Karnaugh–Veitch-Diagramme

		AB			
		00	01	11	10
CD	00	0	0	1	1
	01	0	0	1	1
	11	0	0	0	1
	10	0	1	1	1

		AB			
		00	01	11	10
CD	00	0	0	1	1
	01	0	0	1	1
	11	0	0	0	1
	10	0	1	1	1

		AB			
		00	01	11	10
CD	00	0	0	1	1
	01	0	0	1	1
	11	0	0	0	1
	10	0	1	1	1

		AB			
		00	01	11	10
CD	00	0	0	1	1
	01	0	0	1	1
	11	0	0	0	1
	10	0	1	1	1

- 3. Schritt:** Sammeln der benötigten Ausdrücke (werden aus der Lage der gewählten Zusammenfassungen und ggf. noch benötigten Einzelfeldern abgelesen).

$$\begin{aligned}
 O &= (A \vee C) && \text{(rot)} \\
 \wedge &(A \vee B) && \text{(grün)} \\
 \wedge &(\overline{B} \vee \overline{C} \vee \overline{D}) && \text{(blau)}
 \end{aligned}$$

		AB			
		00	01	11	10
CD	00	0	0	1	1
	01	0	0	1	1
	11	0	0	0	1
	10	0	1	1	1

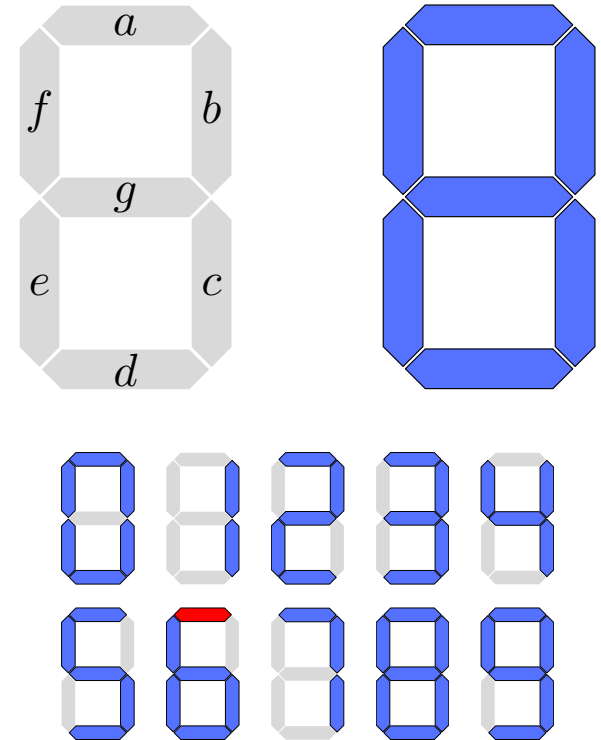
- Man beachte, daß das Ergebnis eine **Konjunktion von Disjunktionen** ist (analog zur konjunktiven Normalform) und folglich die Variablenwerte, die die Zusammenfassungen beschreiben, negiert werden müssen.



# Minimierung: 7-Segment-Anzeige

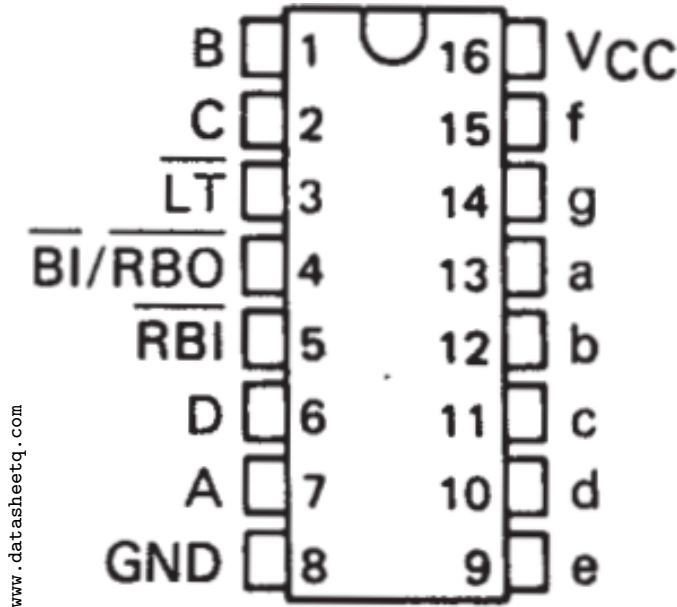
Eingabe				Segmente						
D	C	B	A	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	0	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

7-Segment-Digitalanzeige

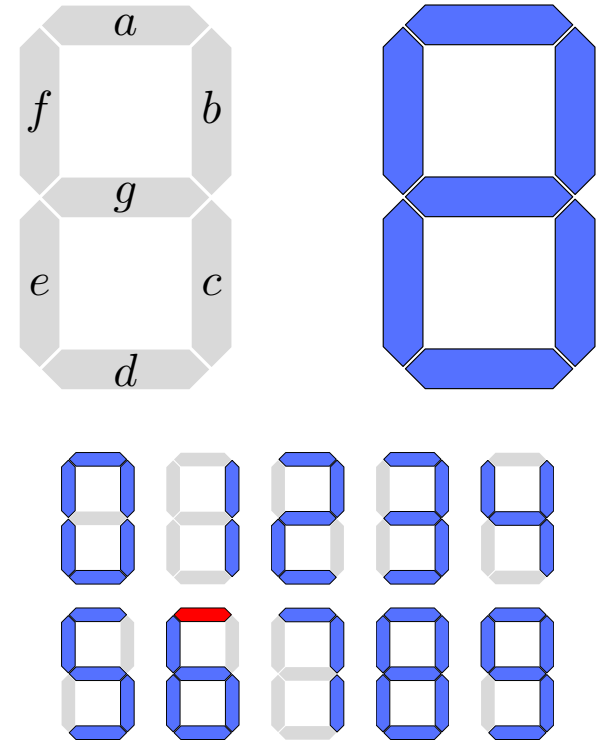


- Beachte: Die sechs Eingabekombinationen 1010, 1011, 1100, 1110, 1111 haben für diese 7-Segment-Anzeige keine Bedeutung. (Warum?)
- Wir fordern, daß der ansteuernde Schaltkreis diese Kombinationen nicht erzeugt; im Karnaugh–Veitch-Diagramm markieren wir sie mit „×“ („don't care“).

# Minimierung: 7-Segment-Anzeige



7-Segment-Digitalanzeige



- Die Darstellung der 6 (mit ausgeschaltetem Segment *a*) wird hier betrachtet, weil sie so durch den integrierten Schaltkreis Ti 7448 implementiert wird.
- Da diese Darstellung aber auch für den Kleinbuchstaben „b“ verwendet wird, findet man die 6 oft mit eingeschaltetem Segment *a* dargestellt (z.B. Ti 74248).

# Minimierung: 7-Segment-Anzeige

- Funktion für das Einschalten des Segments  $a$ :

$$\begin{aligned}
 a = & (\bar{A} \wedge \bar{B} \wedge \bar{C} \wedge \bar{D}) \vee (\bar{A} \wedge B \wedge \bar{C} \wedge \bar{D}) \\
 & \vee (A \wedge B \wedge \bar{C} \wedge \bar{D}) \vee (A \wedge \bar{B} \wedge C \wedge \bar{D}) \\
 & \vee (A \wedge B \wedge C \wedge \bar{D}) \vee (\bar{A} \wedge \bar{B} \wedge \bar{C} \wedge D) \\
 & \vee (A \wedge \bar{B} \wedge \bar{C} \wedge D)
 \end{aligned}$$

		AB			
		00	01	11	10
CD	00	1	1	1	0
	01	1	×	×	1
	11	×	×	×	×
	10	0	0	1	1

- **1. Schritt:** Finde alle möglichen maximalen Zusammenfassungen von Feldern.  
(Zusammenfassungen dürfen „×“ enthalten und sich überlappen!)  
(aber: kein Zusammenfassen von nur „×“ — mindestens eine 1!)

		AB			
		00	01	11	10
CD	00	1	1	1	0
	01	1	×	×	1
	11	×	×	×	×
	10	0	0	1	1

		AB			
		00	01	11	10
CD	00	1	1	1	0
	01	1	×	×	1
	11	×	×	×	×
	10	0	0	1	1

		AB			
		00	01	11	10
CD	00	1	1	1	0
	01	1	×	×	1
	11	×	×	×	×
	10	0	0	1	1

		AB			
		00	01	11	10
CD	00	1	1	1	0
	01	1	×	×	1
	11	×	×	×	×
	10	0	0	1	1

# Minimierung: 7-Segment-Anzeige

		AB			
		00	01	11	10
CD	00	1	1	1	0
	01	1	×	×	1
	11	×	×	×	×
	10	0	0	1	1

		AB			
		00	01	11	10
CD	00	1	1	1	0
	01	1	×	×	1
	11	×	×	×	×
	10	0	0	1	1

		AB			
		00	01	11	10
CD	00	1	1	1	0
	01	1	×	×	1
	11	×	×	×	×
	10	0	0	1	1

		AB			
		00	01	11	10
CD	00	1	1	1	0
	01	1	×	×	1
	11	×	×	×	×
	10	0	0	1	1

## - 2. Schritt:

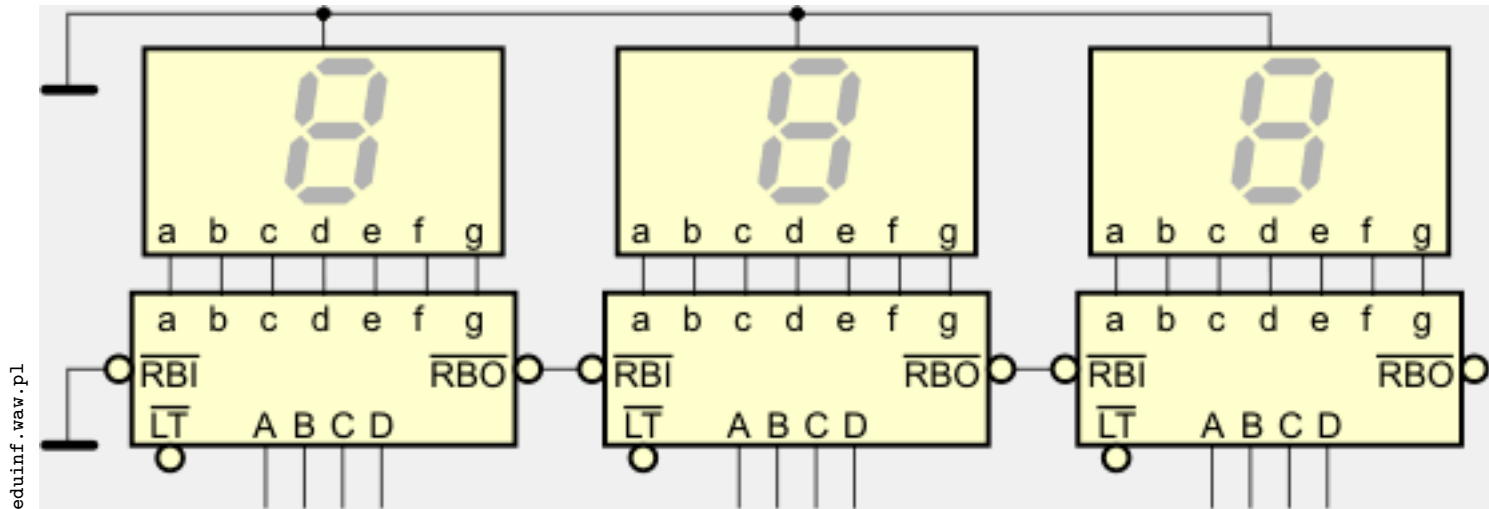
Wähle möglichst wenige Zusammenfassungen, die alle Einsen abdecken.  
(Abgedeckte Felder mit „×“ werden so zu 1, aber für diese “we don’t care”).

- **3. Schritt:** Sammle die benötigten Ausdrücke.  
(werden aus den gewählten Rechtecken abgelesen).

$$\begin{aligned}
 a &= D && \text{(blau)} \\
 \vee &(\bar{A} \wedge \bar{C}) && \text{(rot)} \\
 \vee &(B \wedge \bar{C}) && \text{(grün)} \\
 \vee &(A \wedge C) && \text{(gelb)}
 \end{aligned}$$

		AB			
		00	01	11	10
CD	00	1	1	1	0
	01	1	×	×	1
	11	×	×	×	×
	10	0	0	1	1

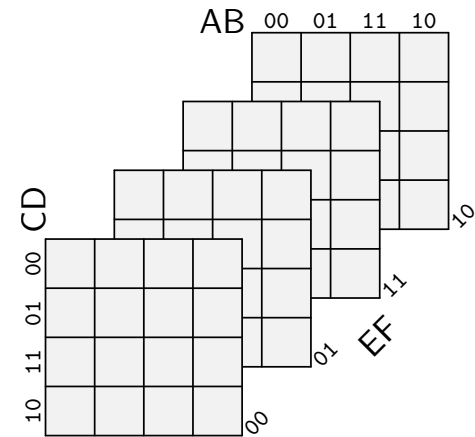
## 7-Segment-Anzeige: Mehrstellige Anzeige



- Der Eingang Ripple Blanking Input (RBI) und der Ausgang Ripple Blanking Output (RBO) des Ti 7448 dienen zur Unterdrückung führender Nullen.
- Eine Null (d.h.  $ABCD = 0000$ ) wird auf zwei Weisen dargestellt:
  - Ist der Eingang  $RBI = 0$  (also  $\overline{RBI} = 1$ ),  
so wird bei Eingabe einer Null tatsächlich eine Null angezeigt.
  - Ist der Eingang  $RBI = 1$  (also  $\overline{RBI} = 0$ ),  
so wird bei Eingabe einer Null kein Segment eingeschaltet (blank).

# Minimierung: Quine–McCluskey-Algorithmus

- Wird der Eingang Ripple Blanking Input (RBI) berücksichtigt, werden alle Segmentfunktionen fünfstellig (Argumente  $A$ ,  $B$ ,  $C$ ,  $D$ , und RBI).
- Eine Minimierung logischer Funktionen mit bis zu sechs Argumenten ist zwar mit erweiterten Karnaugh–Veitch-Diagrammen im Prinzip möglich, aber unhandlich.
- Ein besserer Weg besteht in der Verwendung eines anderen Verfahrens, des **Quine–McCluskey-Algorithmus**.



[Willard V.O. Quine 1955, Edward J. McCluskey 1956]

- Dieser Algorithmus arbeitet direkt auf den Termen der Normalformen, um sogenannte **Primimplikanten** zu finden und auszuwählen.

Vorteile:

- Er ist leicht auch auf mehr als vier/sechs Variablen anwendbar.
- Er ist noch systematischer als Karnaugh–Veitch-Diagramme und dadurch besser für eine Implementierung als Programm geeignet.

# Minimierung: Quine–McCluskey-Algorithmus

- **0. Schritt:** Bilde die disjunktive Normalform der zu minimierenden Funktion.

(Der Algorithmus kann analog auch mit der konjunktiven Normalform ausgeführt werden.)

- **1. Schritt:** Finden der **Primimplikanten**.

- Vereinige Terme, die sich nur dadurch unterscheiden, daß eine einzelne Variable in dem einen Term als positives, im anderen als negatives Literal auftritt.  
(Der gleiche Term darf für mehrere Vereinigungen verwendet werden.)
- Wiederhole dies rekursiv mit den Vereinigungsergebnissen (sog. Implikanten), bis keine weiteren Vereinigungen mehr möglich sind.
- Vernachlässige alle Terme/Implikanten, die mit anderen vereinigt wurden.  
Was übrig bleibt, sind die sogenannten **Primimplikanten**.

- Idee des ersten Schrittes: (analog zu Karnaugh–Veitch)

$$\begin{aligned}(A_1 \dots A_i \dots A_n) \vee (A_1 \dots \bar{A}_i \dots A_n) &= A_1 \dots A_{i-1} A_{i+1} \dots A_n \wedge (A_i \vee \bar{A}_i) \\ &= A_1 \dots A_{i-1} A_{i+1} \dots A_n \wedge 1 \\ &= A_1 \dots A_{i-1} A_{i+1} \dots A_n\end{aligned}$$

# Minimierung: Quine–McCluskey-Algorithmus

- **Beispiel:** Minimiere die Boolesche Funktion (disjunktive Normalform)

$$f(A, B, C, D) = \overline{A}B\overline{C}\overline{D} \vee \overline{A}B\overline{C}D \vee \overline{A}B\overline{C}D \vee \overline{A}B\overline{C}\overline{D} \\ \vee \overline{A}BCD \vee \overline{A}BC\overline{D} \vee \overline{A}BCD \vee \overline{A}BCD.$$

- **1. Schritt:** Finden der **Primimplikanten**.

1er Implikanten			
1	$\overline{A}\overline{B}\overline{C}\overline{D}$	0100	×
2	$\overline{A}\overline{B}\overline{C}D$	1000	×
3	$\overline{A}\overline{B}\overline{C}D$	1001	×
4	$\overline{A}\overline{B}\overline{C}D$	1010	×
5	$\overline{A}\overline{B}CD$	1011	×
6	$\overline{A}B\overline{C}\overline{D}$	1100	×
7	$\overline{A}B\overline{C}\overline{D}$	1110	×
8	$\overline{A}BCD$	1111	×

Die 1er Implikanten  
sind die Minterme.

2er Implikanten				
1+6 → 9	$B\overline{C}\overline{D}$	-100	$P_1$	
2+3 → 10	$\overline{A}B\overline{C}$	100-	×	
2+4 → 11	$\overline{A}B\overline{D}$	10-0	×	
2+6 → 12	$\overline{A}C\overline{D}$	1-00	×	
3+5 → 13	$\overline{A}B\overline{D}$	10-1	×	
4+5 → 14	$\overline{A}B\overline{C}$	101-	×	
4+7 → 15	$\overline{A}C\overline{D}$	1-10	×	
5+8 → 16	$\overline{A}CD$	1-11	×	
6+7 → 17	$\overline{A}B\overline{D}$	11-0	×	
7+8 → 18	$\overline{A}BC$	111-	×	

4er Implikanten				
10+14,				
11+13 → 19	$\overline{A}\overline{B}$	10--	$P_2$	
11+17,				
12+15 → 20	$\overline{A}\overline{D}$	1--0	$P_3$	
14+18,				
15+16 → 21	$\overline{A}C$	1-1-	$P_4$	

Primimplikanten  
(unvereinigte Implikanten)  
sind mit  $P_i$  bezeichnet.

„×“: Minterm bzw. Implikant wurde mit einem anderen vereinigt.



# Minimierung: Quine–McCluskey-Algorithmus

- **2. Schritt:** Aufstellen und Auswerten der **Primimplikantentabelle**.
  - In einer Tabelle, deren Spalten die **Minterme** (Konjunktionen der disjunktiven Normalform) und deren Zeilen die Primimplikanten zugeordnet sind, wird markiert, welche Primimplikanten welche Minterme abdecken.
  - Finde die **wesentlichen Primimplikanten** durch Aufsuchen der Spalten (Minterme) mit nur einer Markierung.  
(Idee: Minterm kann nur durch einen Primimplikanten abgedeckt werden.  
⇒ Dieser Primimplikant ist wesentlich zur Darstellung der Funktion.)
  - Entferne aus der Tabelle alle Spalten (Minterme), die durch die wesentlichen Primimplikanten abgedeckt werden.
  - Wähle zur Abdeckung der verbleibenden Spalten (Minterme) eine möglichst kleine Menge von Primimplikanten.
  - Eine systematische Methode für diese Auswahl von Primimplikanten ist **Petricks Algorithmus** (später). [Stanley R. Petrick 1956]

# Minimierung: Quine–McCluskey-Algorithmus

- **Beispiel:** Minimiere die Boolesche Funktion (disjunktive Normalform)

$$f(A, B, C, D) = \overline{A}\overline{B}\overline{C}\overline{D} \vee \overline{A}\overline{B}\overline{C}D \vee \overline{A}\overline{B}C\overline{D} \vee \overline{A}\overline{B}CD \\ \vee \overline{A}B\overline{C}\overline{D} \vee \overline{A}B\overline{C}D \vee \overline{A}BC\overline{D} \vee \overline{A}BCD.$$

- **2. Schritt:** Aufstellen und Auswerten der **Primimplikantentabelle**.

			Minterme (Konjunktionen der disjunktiven NF)							
Primimplikanten			0100	1000	1001	1010	1011	1100	1110	1111
$P_1$	-100	*	●					●		
$P_2$	10--	*		●	●	●	●			
$P_3$	1--0			●		●		●	●	
$P_4$	1-1-	*				●	●		●	●

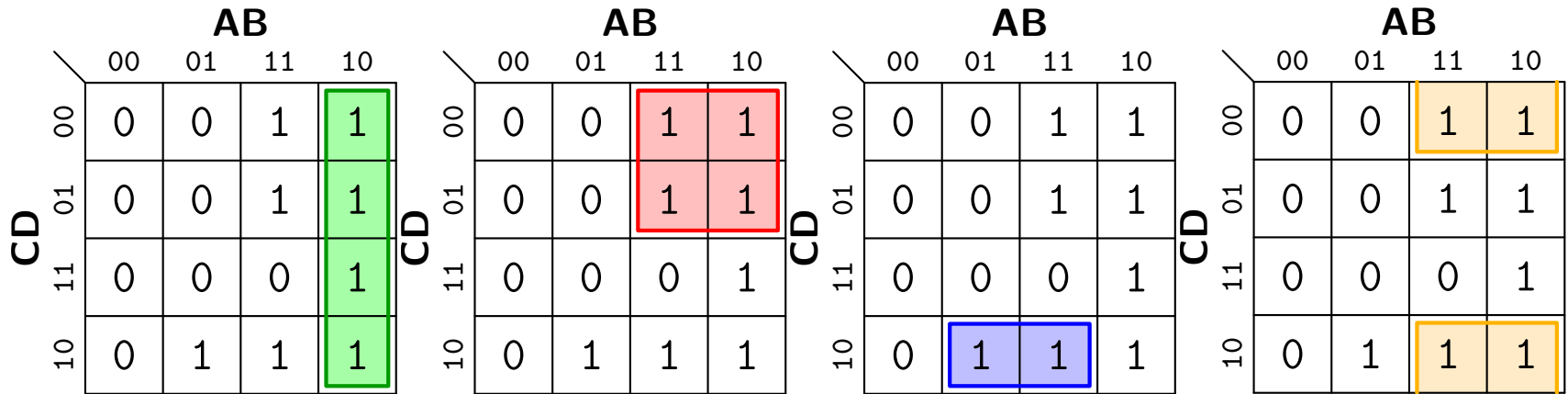
Wesentliche Primimplikanten sind durch „\*“ markiert.

**Ergebnis:**

- Abdeckung **nur durch einen** Primimplikanten, dieser ist daher wesentlich.
- Abdeckung auch durch wesentliche Primimplikanten.
- Nicht benötigte Abdeckungen.

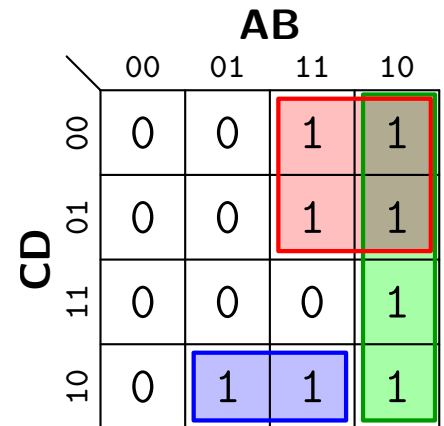
$$f = \overline{B}\overline{C}\overline{D} \\ \vee \overline{A}\overline{B} \\ \vee \overline{A}C$$

# Karnaugh–Veitch-Diagramme: Zusammenfassungen

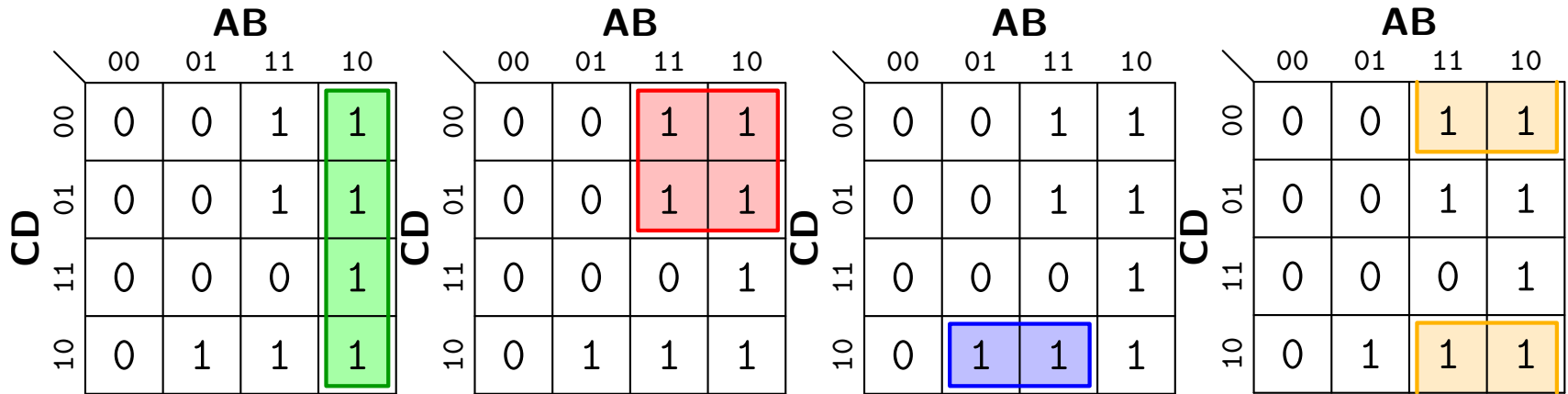


- **Primimplikanten des Quine–McCluskey-Algorithmus entsprechen Feld-Zusammenfassungen in Karnaugh–Veitch-Diagrammen.**
- $2^k$ -Primimplikanten sind Zusammenfassungen von  $2^k$  Feldern.  
( $k$  = Anzahl der Striche im Primimplikanten)

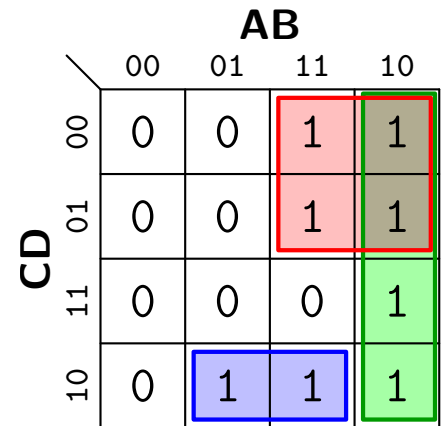
- Grün: entspricht dem 4er-Primimplikanten  $10--$ .
- Rot: entspricht dem 4er-Primimplikanten  $1-0-$ .
- Blau: entspricht dem 2er-Primimplikanten  $-110$ .
- Gelb: entspricht dem 4er-Primimplikanten  $1--0$ .



# KV-Diagramme: Wesentliche Zusammenfassungen



- **Primimplikanten des Quine–McCluskey-Algorithmus entsprechen Feld-Zusammenfassungen in Karnaugh–Veitch-Diagrammen.**
- Entsprechend gibt es auch **wesentliche Zusammenfassungen** in Karnaugh–Veitch-Diagrammen.
  - Grün: wesentlich, da nur sie 1011 abdeckt.
  - Rot: wesentlich, da nur sie 1101 abdeckt.
  - Blau: wesentlich, da nur sie 0110 abdeckt.
  - Gelb: nicht wesentlich.



# Minimierung: Petricks Algorithmus

- Decken die wesentlichen Primimplikanten nicht alle Minterme ab, müssen aus den restlichen Primimplikanten geeignete ausgewählt werden, um die verbleibenden Minterme abzudecken.
  
- Auswahl mit Hilfe von **Petricks Algorithmus**: [Stanley R. Petrick 1956]
  1. Bilde eine reduzierte Primimplikantentabelle, die nur noch die noch nicht abgedeckten Minterme und die nicht wesentlichen Primimplikanten enthält, die einige davon abdecken.
  2. Ordne jedem Primimplikanten eine Auswahlvariable  $P_i$  zu:  
 $P_i = 1$ : Primimplikant ausgewählt,  $P_i = 0$ : Primimplikant nicht ausgewählt.  
Bilde für jede Spalte (Minterm) die Disjunktion der Auswahlvariablen der sie abdeckenden Primimplikanten und verknüpfe diese Disjunktionen konjunktiv.  
Die sich ergebende Konjunktion  $C$  von Disjunktionen von Auswahlvariablen ist genau dann wahr, wenn alle Spalten (Minterme) abgedeckt sind.  
**Idee:** Die Konjunktion  $C$  beschreibt alle möglichen Wahlen von Mengen von Primimplikanten, die alle (verbleibenden) Minterme abdecken.

# Minimierung: Petricks Algorithmus

- Auswahl mit Hilfe von **Petricks Algorithmus**: [Stanley R. Petrick 1956]
  3. Wandle die Konjunktion  $C$  durch Anwenden der Distributivgesetze um in eine Disjunktion  $D$  von Konjunktionen von Auswahlvariablen.  
Dies liefert, in den Konjunktionen der Disjunktion  $D$ , Mengen von Primimplikanten, die alle Minterme abdecken.
  4. Vereinfache  $D$  mit Hilfe des Absorptionsgesetzes  $a \vee (a \wedge b) = a$  zu  $D'$ .  
Dies liefert, als Konjunktionen der Disjunktion  $D'$ , alle Mengen von Primimplikanten, die alle Minterme irredundant abdecken.  
Die Konjunktionen von  $D'$  enthalten keine „überflüssigen“ Primimplikanten. Jede der Konjunktionen (Primimplikantenwahlen) in  $D'$  kann daher im Prinzip zur sparsamen Abdeckung der (verbleibenden) Minterme verwendet werden.
  5. Wähle die Konjunktionen in  $D'$  mit den wenigsten Primimplikanten; wähle aus diesen eine derjenigen mit der kleinsten Zahl an Literalen.  
Idee: Konjunktionen mit wenigen Primimplikanten sind meist „einfacher“ (weniger Literale) als solche mit vielen. (Aber: Dies ist eine Heuristik!)

# Minimierung: Petricks Algorithmus

- **Beispiel:** Minimiere die Boolesche Funktion (disjunktive Normalform)

$$f(A, B, C) = \overline{A}\overline{B}\overline{C} \vee \overline{A}\overline{B}C \vee \overline{A}B\overline{C} \\ \vee A\overline{B}\overline{C} \vee A\overline{B}C \vee ABC.$$

- **Primimplikantentabelle** dieser Funktion (ohne Herleitung):

Primimplikanten		Minterme (Konjunktionen)					
		000	001	010	101	110	111
$P_1$	00-	•	•				
$P_2$	0-0	•		•			
$P_3$	-01		•		•		
$P_4$	-10			•		•	
$P_5$	1-1				•		•
$P_6$	11-					•	•

- Es gibt keine wesentlichen Primimplikanten,  
da keine Spalte (Minterm) nur eine Markierung enthält.  
⇒ direkte Anwendung von **Petricks Algorithmus**.

# Minimierung: Petricks Algorithmus

Primimp.		Minterme (Konjunktionen)					
		000	001	010	101	110	111
$P_1$	00-	●	●				
$P_2$	0-0	●		●			
$P_3$	-01		●		●		
$P_4$	-10			●		●	
$P_5$	1-1				●		●
$P_6$	11-					●	●

Konjunktion von Disjunktionen  
der Primimplikanten je Spalte:

$$\begin{aligned}
 & (P_1 \vee P_2) & (000) \\
 \wedge & (P_1 \vee P_3) & (001) \\
 \wedge & (P_2 \vee P_4) & (010) \\
 \wedge & (P_3 \vee P_5) & (101) \\
 \wedge & (P_4 \vee P_6) & (110) \\
 \wedge & (P_5 \vee P_6) & (111)
 \end{aligned}$$

## - Umformung in Disjunktion von Konjunktionen (**Distributivgesetze**):

$$\begin{aligned}
 & (P_1 \vee P_2) \wedge (P_1 \vee P_3) \wedge (P_4 \vee P_2) \wedge (P_4 \vee P_6) \wedge (P_3 \vee P_5) \wedge (P_5 \vee P_6) \\
 &= (P_1 \vee (P_2 \wedge P_3)) \wedge (P_4 \vee (P_2 \wedge P_6)) \wedge (P_5 \vee (P_3 \wedge P_6)) \\
 &= ((P_1 \wedge P_4) \vee (P_1 \wedge P_2 \wedge P_6) \vee (P_2 \wedge P_3 \wedge P_4) \vee (P_2 \wedge P_2 \wedge P_3 \wedge P_6)) \\
 &\quad \wedge (P_5 \vee (P_3 \wedge P_6)) \\
 &= P_1 P_4 P_5 \vee P_1 P_2 P_5 P_6 \vee P_2 P_3 P_4 P_5 \vee P_2 P_3 P_5 P_6 \\
 &\quad \vee P_1 P_3 P_4 P_6 \vee P_1 P_2 P_3 P_6 \vee P_2 P_3 P_4 P_6 \vee P_2 P_3 P_6
 \end{aligned}$$



# Minimierung: Petricks Algorithmus

- Vereinfachung mit Hilfe des **Absorptionsgesetzes**  $a \vee (a \wedge b) = a$ :

$$\begin{aligned} & P_1P_4P_5 \vee P_1P_2P_5P_6 \vee P_2P_3P_4P_5 \vee P_2P_3P_5P_6 \\ & P_1P_3P_4P_6 \vee P_1P_2P_3P_6 \vee P_2P_3P_4P_6 \vee P_2P_3P_6 \\ & = P_1P_4P_5 \vee P_1P_2P_5P_6 \vee P_2P_3P_4P_5 \vee P_1P_3P_4P_6 \vee P_2P_3P_6. \end{aligned}$$

(Bemerkung: dieser Schritt sieht im Hinblick auf den nächsten redundant aus, aber siehe vorletzter Punkt).

- Wahl der Konjunktionen mit der kleinsten Zahl an Primimplikanten:

$$\begin{array}{lll} P_1 \wedge P_4 \wedge P_5 & \text{entspricht} & f(A, B, C) = \overline{A}\overline{B} \vee B\overline{C} \vee AC. \\ P_2 \wedge P_3 \wedge P_6 & \text{entspricht} & f(A, B, C) = \overline{A}\overline{C} \vee \overline{B}C \vee AB. \end{array}$$

- Da beide Ausdrücke 6 Literale enthalten, kann jeder der beiden gewählt werden.
- Die Auswahl der Konjunktionen mit der kleinsten Zahl an Primimplikanten ist eine **Heuristik**. Im Prinzip müßten alle Konjunktionen geprüft werden.
- Das Problem der Wahl der besten Menge von Primimplikanten ist im allgemeinen Fall **NP-vollständig** ( $\Rightarrow$  Komplexitätstheorie).

(Intuitiv: Es geht nicht fundamental besser, als alle Möglichkeiten zu probieren.)

# Inhalt

## **1 Minimierung Boolescher Formeln**

- 1.1 Äquivalenzumformungen
- 1.2 Karnaugh–Veitch-Diagramme
- 1.3 Quine–McCluskey-Algorithmus
- 1.4 Petricks Algorithmus

## **2 Programmierbare Logikarrays**

- 2.1 Beispiel: Originalfunktion vs. disjunktive Normalform
- 2.2 Allgemeine Struktur
- 2.3 Hardware-Implementation

## **3 Hardware-Beschreibungssprache (HDL)**

- 3.1 Motivation: Hardware-Simulation
- 3.2 Beispiel: Aufbau eines AND-Gatters
- 3.3 Schnittstelle und Implementierung
- 3.4 Simulationsumgebung für diese Vorlesung

# Programmierbare Logikarrays

- Man beachte, daß alle bisher betrachteten Minimierungsergebnisse die folgende allgemeine Form haben  
(die  $i_k$ ,  $k = 1, 2, 3, \dots$ , bezeichnen die Eingaben,  $o$  die Ausgabe):

$$o = (i_1 \wedge \bar{i}_2 \wedge \bar{i}_3 \wedge \dots) \vee (\bar{i}_1 \wedge i_2 \wedge \bar{i}_3 \wedge \dots) \vee (\bar{i}_1 \wedge \bar{i}_2 \wedge i_3 \wedge \dots) \vee \dots$$

- Alle Booleschen Formeln können in diese Form gebracht werden, denn es handelt sich um eine Disjunktion von Konjunktionen von Literalen (sum of products, SOP); Spezialfall: **disjunktive Normalform**.
- **Programmierbare Logikarrays** (programmable logic array, PLAs) sind solche Formeln “realisiert in Hardware”.
  - Alle Eingaben  $i_k$  und ihre Negationen  $\bar{i}_k$  sind verfügbar.
  - Die Eingaben werden über AND-Gatter verknüpft.
  - Die Ausgaben der AND-Gatter werden durch OR-Gatter verknüpft.
  - Ein programmierbarer Logikarray wird durch Entfernen von Verbindungen konfiguriert (“programmiert”).

# Programmierbare Logikarrays: Beispiel

- Gegeben sei die Funktion  $s(a, m, w) = \bar{a} \wedge (m \vee w)$ .
- **Wahrheitstafel** dieser Funktion:

$a$	$m$	$w$	Minterm	$s(a, m, w) = \bar{a} \wedge (m \vee w)$
0	0	0	$m_0 = \bar{a}\bar{m}\bar{w}$	0
0	0	1	$m_1 = \bar{a}\bar{m}w$	1
0	1	0	$m_2 = \bar{a}m\bar{w}$	1
0	1	1	$m_3 = \bar{a}mw$	1
1	0	0	$m_4 = a\bar{m}\bar{w}$	0
1	0	1	$m_5 = a\bar{m}w$	0
1	1	0	$m_6 = am\bar{w}$	0
1	1	1	$m_7 = amw$	0

- **Disjunktive Normalform:** (Disjunktion von Konjunktionen von Literalen)

$$s(a, m, w) = m_1 \vee m_2 \vee m_3 = \bar{a}\bar{m}w \vee \bar{a}m\bar{w} \vee \bar{a}mw.$$

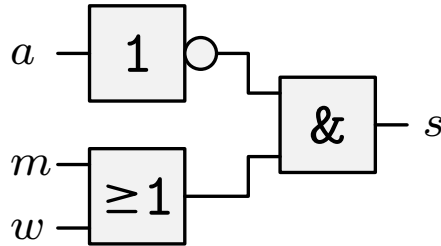
- **Minimierte Form:** (auch Disjunktion von Konjunktionen von Literalen)

$$s(a, m, w) = \bar{a}m \vee \bar{a}w.$$

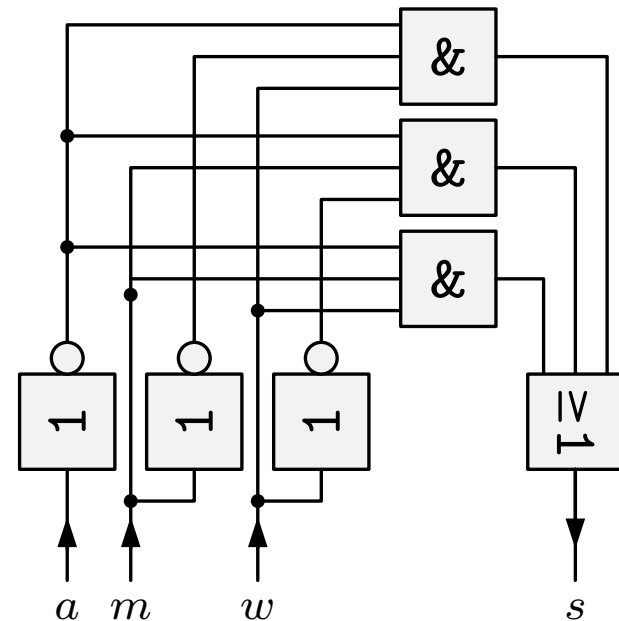
# Programmierbare Logikarrays: Beispiel

- Gatterimplementierung der Originalfunktion und der disjunktiven Normalform:

$$s(a, m, w) = \bar{a} \wedge (m \vee w)$$



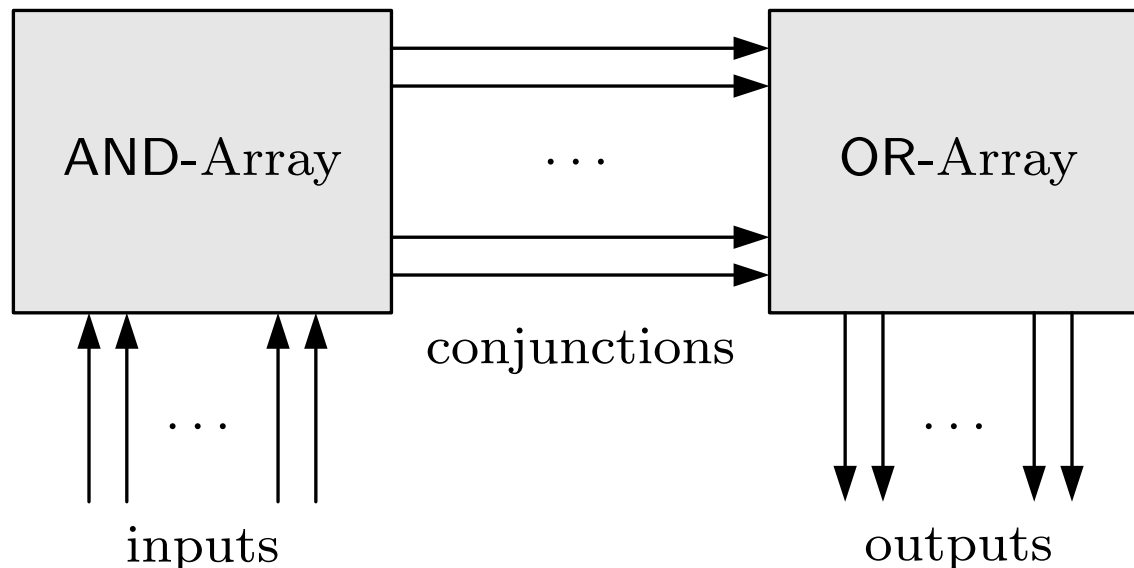
$$s(a, m, w) = \bar{a}\bar{m}w \vee \bar{a}m\bar{w} \vee \bar{a}mw.$$



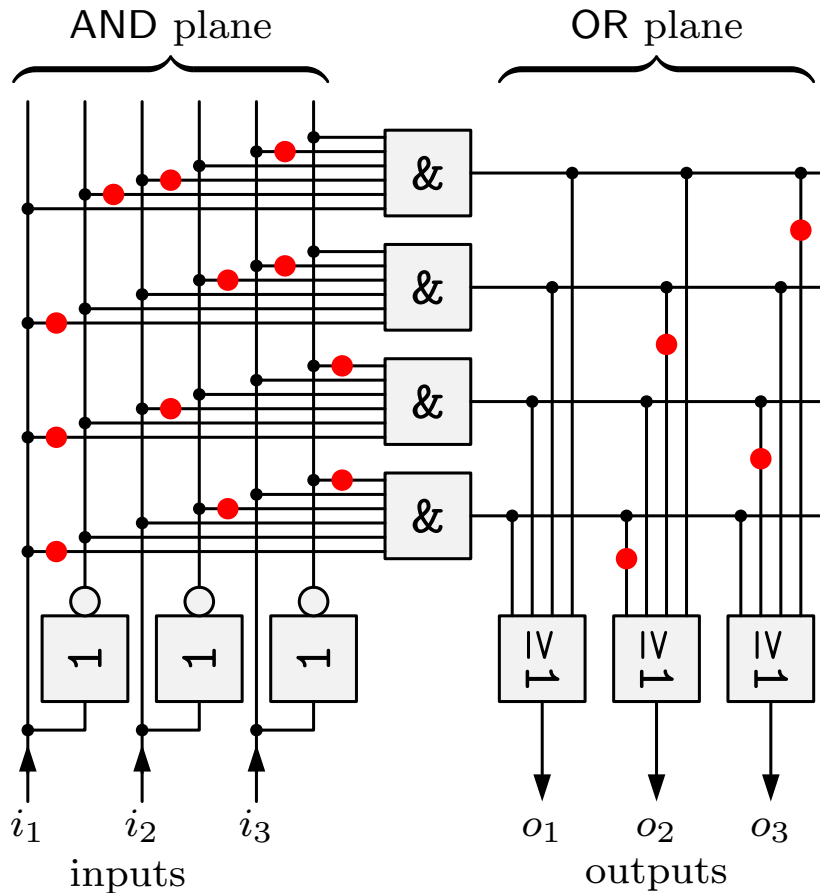
- Vorteil der Originalfunktion: weniger Gatter.
- Nachteil der Originalfunktion: Struktur hängt von der jeweiligen Funktion ab.
- Vorteil der disjunktiven Normalform: einheitliche Standardstruktur mit regelmäßigen Termen, durch Standard-Gatterstruktur darstellbar.

# Programmierbare Logikarrays: Allgemeine Struktur

- Jede Funktion in disjunktiver Normalform (oder allgemein jede Disjunktion von Konjunktionen von Literalen) kann durch eine Standard-Gatterstruktur dargestellt werden, die aus den folgenden Elementen besteht:
  - einem NOT-Gatter (Inverter) für jede Eingabe, so daß jede Eingabe negiert und unnegiert zur Verfügung steht (Literale),
  - einem AND-Array zur Berechnung der Konjunktionen und
  - einem OR-Array zur disjunktiven Verknüpfung der Konjunktionen.



# Programmierbare Logikarrays



Die rot markierten Verbindungen werden durch die Programmierung **entfernt**.

## Beispiel:

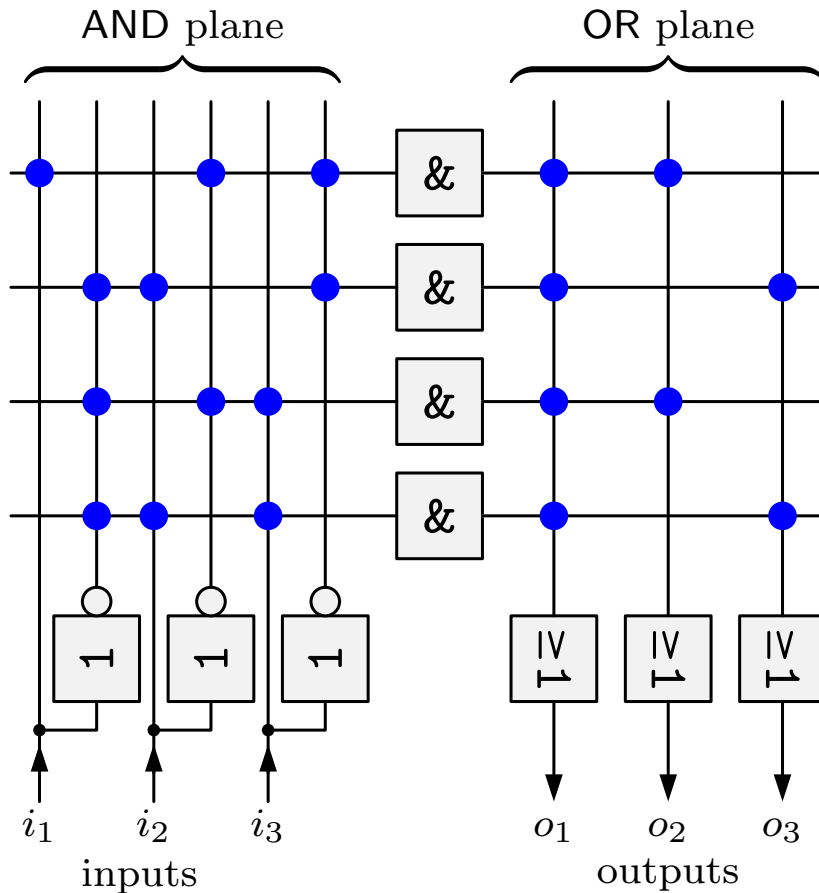
ternäre Boolesche Funktion

$$o_1 = f(i_1, i_2, i_3):$$

$j$	$i_1$	$i_2$	$i_3$	$o_1$	$C_j$
1	0	0	0	0	
2	1	0	0	1	$i_1 \wedge \bar{i}_2 \wedge \bar{i}_3$
3	0	1	0	1	$\bar{i}_1 \wedge i_2 \wedge \bar{i}_3$
4	1	1	0	0	
5	0	0	1	1	$\bar{i}_1 \wedge \bar{i}_2 \wedge i_3$
6	1	0	1	0	
7	0	1	1	1	$\bar{i}_1 \wedge i_2 \wedge i_3$
8	1	1	1	0	

$$\begin{aligned}
 D &= C_2 \vee C_3 \vee C_5 \vee C_7 \\
 &= (i_1 \wedge \bar{i}_2 \wedge \bar{i}_3) \vee (\bar{i}_1 \wedge i_2 \wedge \bar{i}_3) \\
 &\quad \vee (\bar{i}_1 \wedge \bar{i}_2 \wedge i_3) \vee (\bar{i}_1 \wedge i_2 \wedge i_3)
 \end{aligned}$$

# Programmierbare Logikarrays



Vereinfachte Darstellung:  
nur eine Linie je Gatter und  
die **Verbindungen** werden markiert.

## Beispiel:

ternäre Boolesche Funktion

$$o_1 = f(i_1, i_2, i_3):$$

$j$	$i_1$	$i_2$	$i_3$	$o_1$	$C_j$
1	0	0	0	0	
2	1	0	0	1	$i_1 \wedge \bar{i}_2 \wedge \bar{i}_3$
3	0	1	0	1	$\bar{i}_1 \wedge i_2 \wedge \bar{i}_3$
4	1	1	0	0	
5	0	0	1	1	$\bar{i}_1 \wedge \bar{i}_2 \wedge i_3$
6	1	0	1	0	
7	0	1	1	1	$\bar{i}_1 \wedge i_2 \wedge i_3$
8	1	1	1	0	

$$\begin{aligned}
 D &= C_2 \vee C_3 \vee C_5 \vee C_7 \\
 &= (i_1 \wedge \bar{i}_2 \wedge \bar{i}_3) \vee (\bar{i}_1 \wedge i_2 \wedge \bar{i}_3) \\
 &\quad \vee (\bar{i}_1 \wedge \bar{i}_2 \wedge i_3) \vee (\bar{i}_1 \wedge i_2 \wedge i_3)
 \end{aligned}$$



## (Programmierbare) Logikarrays: Implementierung

- Für eine elektronische Implementierung (durch Transistoren) sind AND- und OR-Gatter nicht besonders gut geeignet.
- Besser geeignet sind NAND- und NOR-Gatter (und ggf. NOT-Gatter), da sich diese direkt durch einfache Transistorschaltungen implementieren lassen.
- Implementierung nur durch NOR-/NOT-Gatter: ( $l_{jk}$  sind Literale)

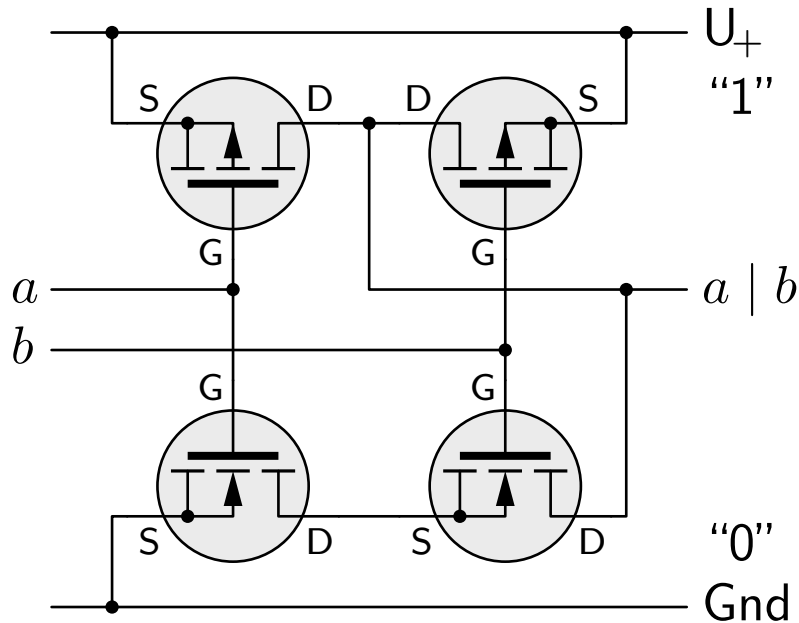
$$\begin{aligned} o &= \bigvee_{j=1}^n \left( \bigwedge_{k=1}^{m_j} l_{jk} \right) = \bigvee_{j=1}^n \neg \neg \left( \bigwedge_{k=1}^{m_j} l_{jk} \right) \\ &= \bigvee_{j=1}^n \neg \left( \bigvee_{k=1}^{m_j} \neg l_{jk} \right) = \bigvee_{j=1}^n \downarrow (\neg l_{j1}, \dots, \neg l_{j,m_j}) \\ &= \neg \downarrow (\downarrow (\neg l_{11}, \dots, \neg l_{1,m_1}), \dots, \downarrow (\neg l_{n1}, \dots, \neg l_{n,m_n})) \end{aligned}$$

- Implementierung nur durch NAND-/NOT-Gatter: ( $l_{jk}$  sind Literale)

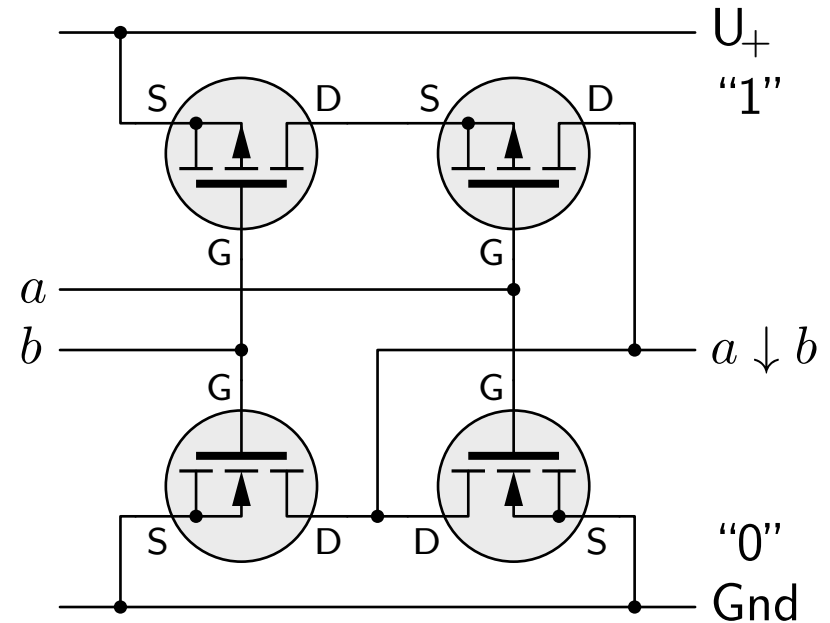
$$o = \bigvee_{j=1}^n \left( \bigwedge_{k=1}^{m_j} l_{jk} \right) = \dots$$

(Übungsaufgabe, Blatt 5)

## Erinnerung: Gatter-Transistorschaltungen



NAND-Gatter in CMOS-Technik

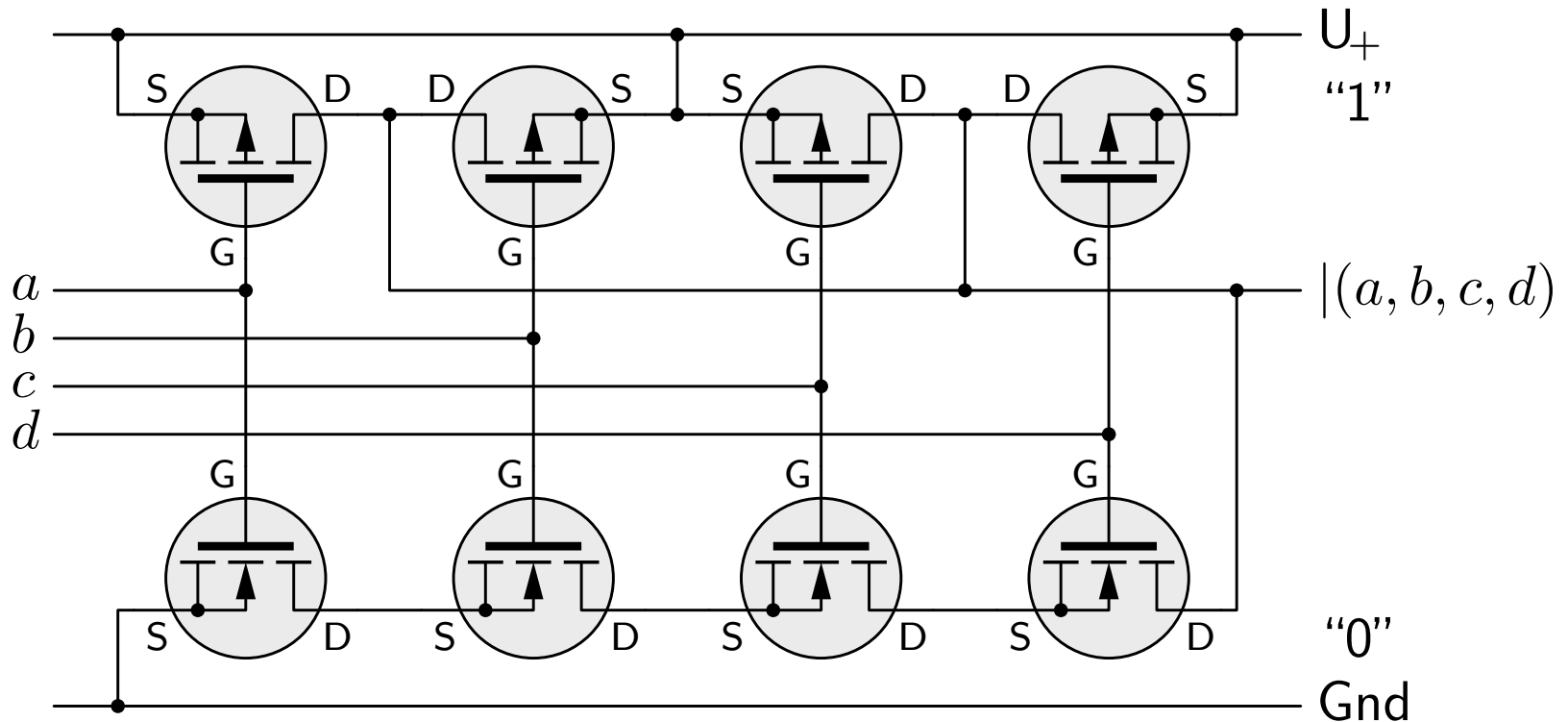


NOR-Gatter in CMOS-Technik

Man beachte die Dualität der Schaltkreise:

Durch Vertauschen der "1" (Versorgungsspannung) und der "0" (Masse) sowie Vertauschen der Transistortypen (N-Typ, unten, gegen P-Typ, oben) und umgekehrt gehen die beiden Schaltungen ineinander über.

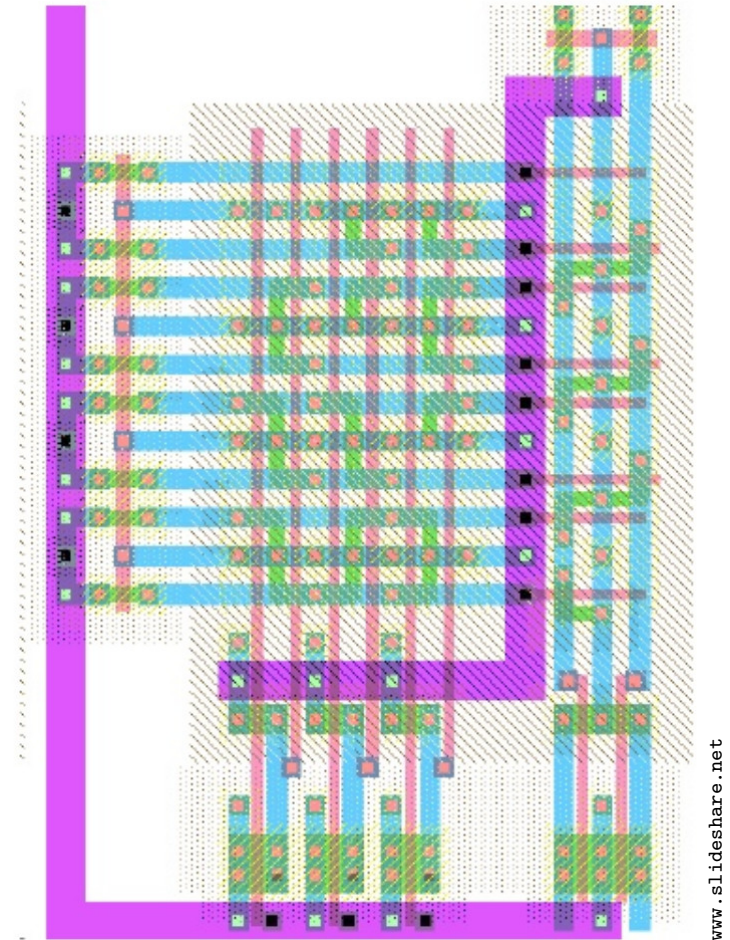
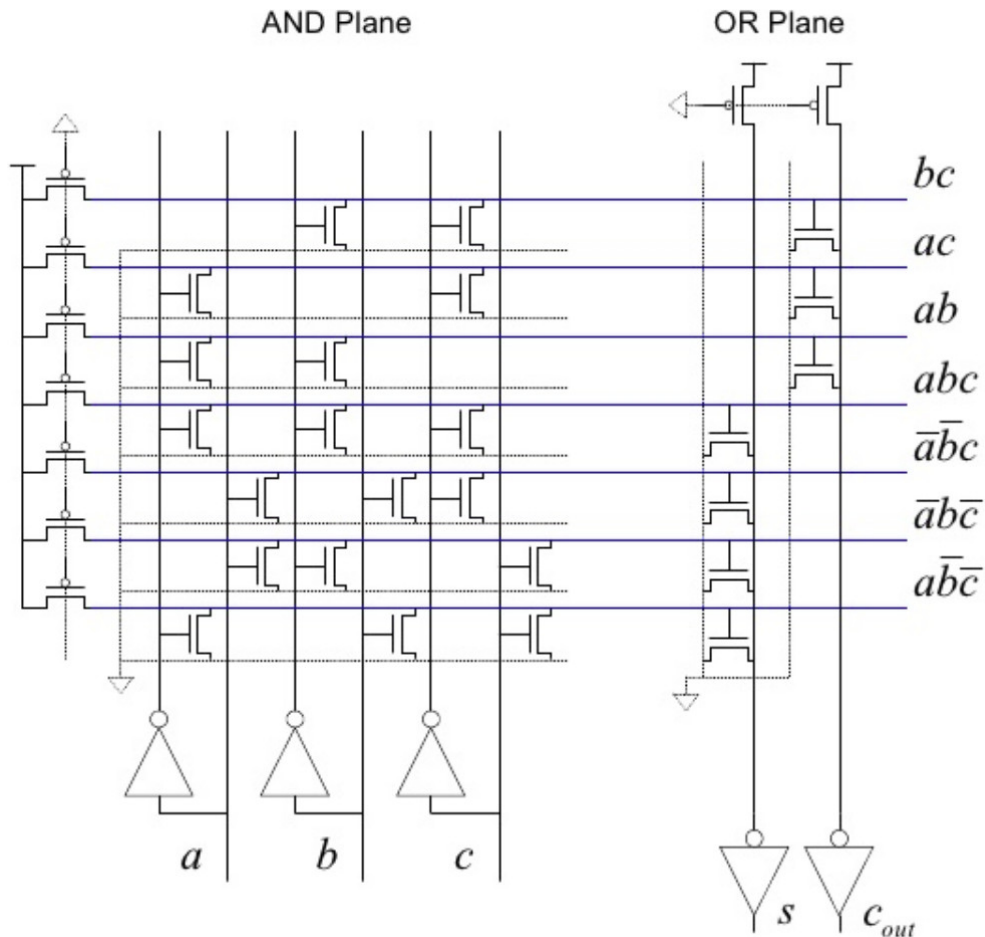
## Erinnerung: Gatter-Transistorschaltungen



### NAND-Gatter mit 4 Eingängen in CMOS-Technik

Derartige Transistorschaltungen für NAND- und NOR-Gatter können nicht nur zwei, sondern im Prinzip beliebig viele Eingänge haben. Von dieser Möglichkeit wird z.B. in Logikarrays Gebrauch gemacht.

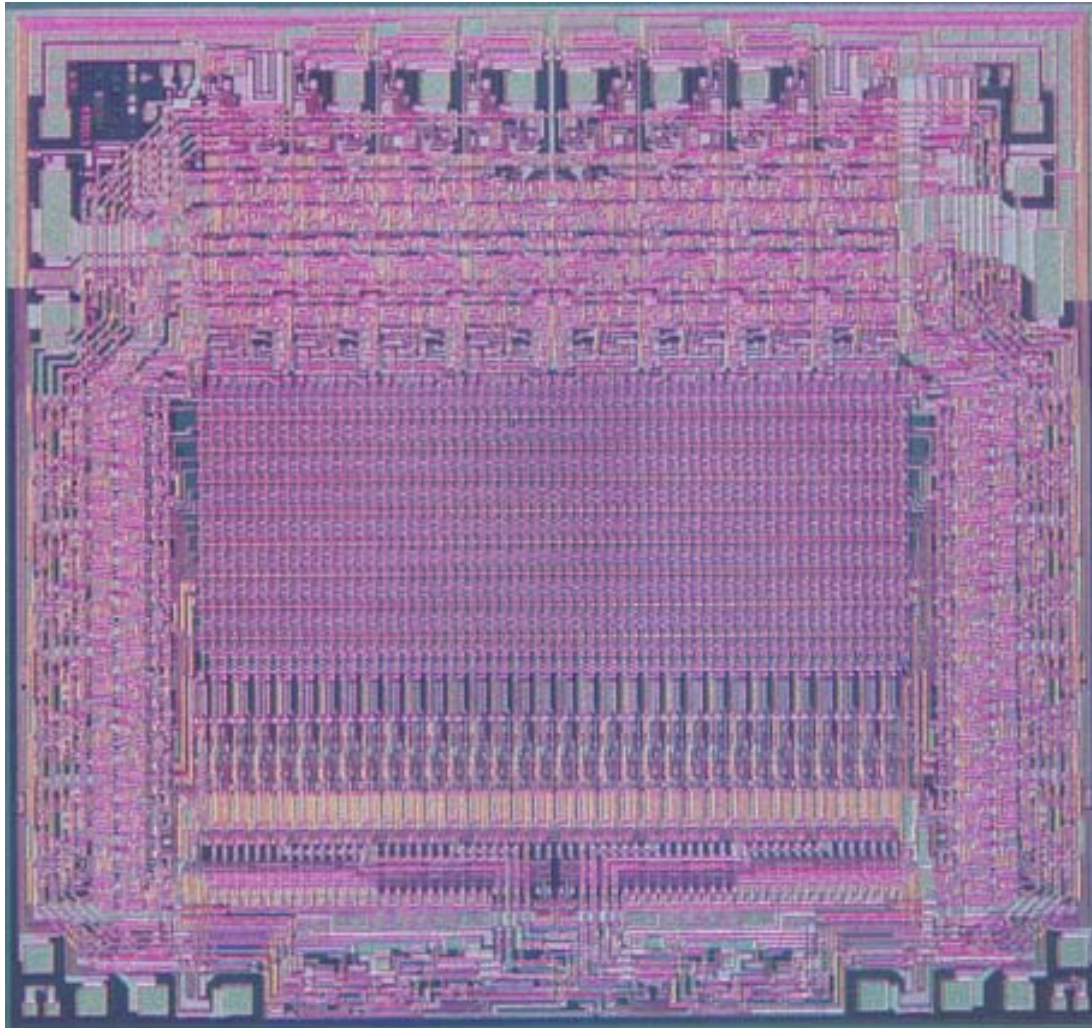
# (Programmierbare) Logikarrays: Layout



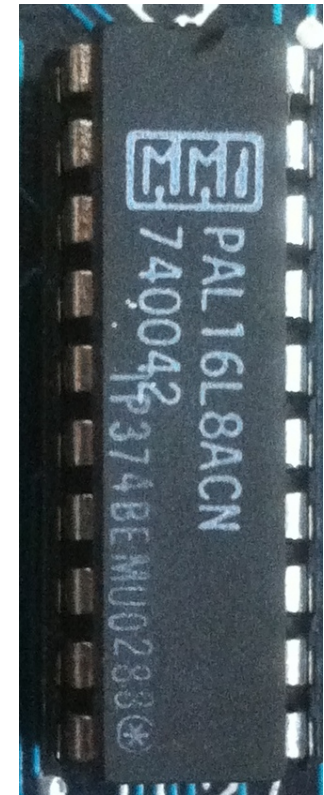
Verwendet NOR-NOT-Logik, aber:  
P-Typ-Transistor (links/oben) wird als ein „Widerstand“ benutzt (Pseudo-RTL).



# Integrierte Schaltungen: Programmierbare Logikarrays



smithsonianchips.si.edu



www.mikrocontroller.net

MMI PAL 16L8 [1977]  
(Monolithic Memories Inc.)  
(Programmable  
Array Logic)

# Programmierbare Logikarrays: Heute

- Anfangs waren programmierbare Logikarrays “write once” (einmal schreibbar), da mögliche Verbindungen durch “fuses” (Sicherungen) dargestellt waren, die (elektrisch) durchtrennt wurden (ähnlich einer Schmelzsicherung).
- Später wurden durch Ultraviolettlicht (vgl. EPROM — Erasable Programmable Read Only Memory), dann auch elektrisch löschbare Gattermatrizen möglich (vgl. EEPROM — Electrically Erasable Programmable Read Only Memory)
- Heute bieten “Complex Programmable Logic Devices” (CLPDs) u.a.
  - programmierbare AND/OR-Matrizen,
  - programmierbare Rückkopplungen (feedback lines),
  - programmierbare Eingabemodule mit Speicherzellen (Register),
  - programmierbare Ausgabemodule mit Speicherzellen (Register).
- CLPDs sind im wesentlichen komplexe Zusammenstellungen von programmierbaren Bauelement-Matrizen (Makrozellen).  
⇒ verwandt zu VLSI-Layout mit Standardzellen.

# Inhalt

## 1 Minimierung Boolescher Formeln

- 1.1 Äquivalenzumformungen
- 1.2 Karnaugh–Veitch-Diagramme
- 1.3 Quine–McCluskey-Algorithmus
- 1.4 Petricks Algorithmus

## 2 Programmierbare Logikarrays

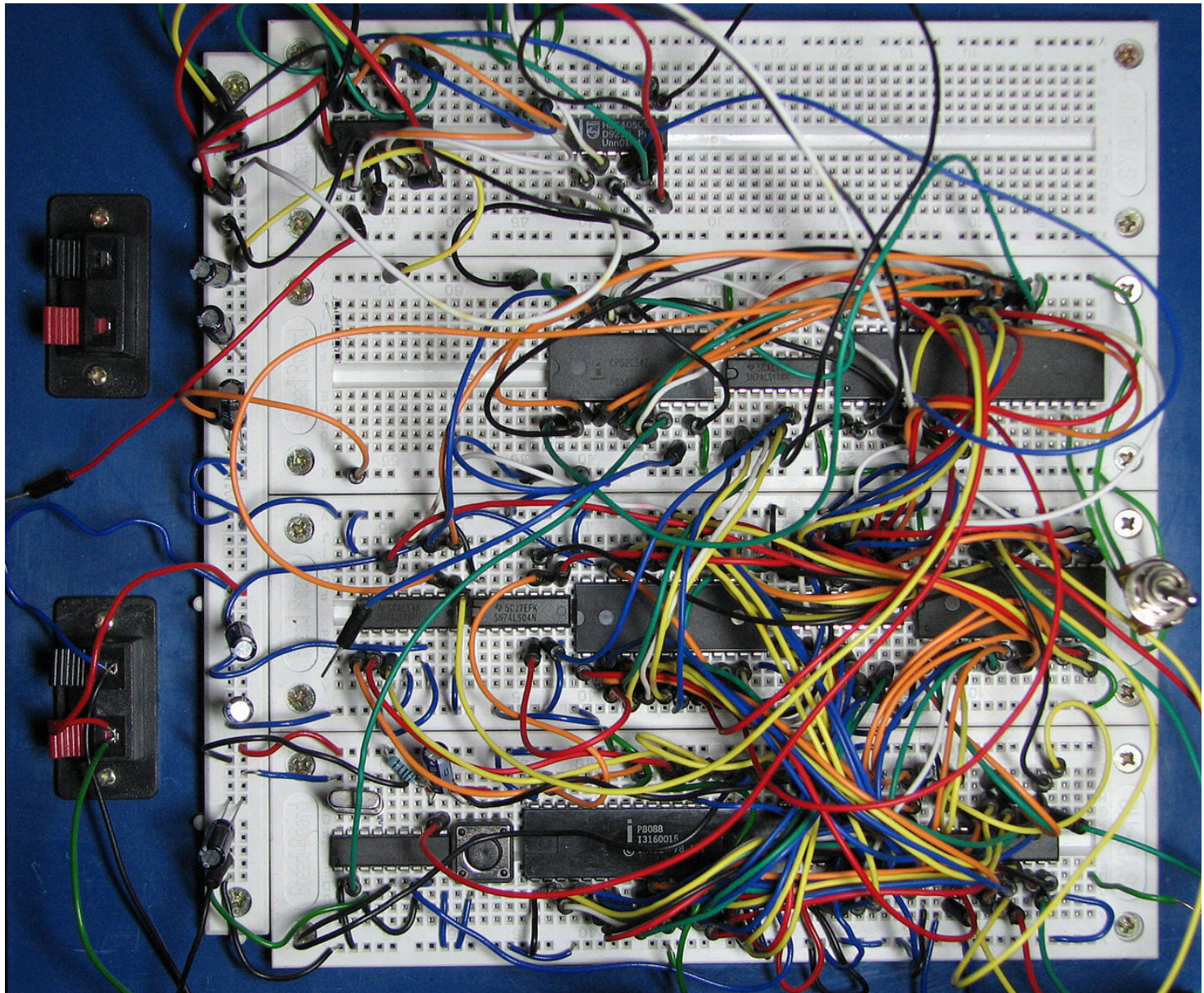
- 2.1 Beispiel: Originalfunktion vs. disjunktive Normalform
- 2.2 Allgemeine Struktur
- 2.3 Hardware-Implementation

## 3 Hardware-Beschreibungssprache (HDL)

- 3.1 Motivation: Hardware-Simulation
- 3.2 Beispiel: Aufbau eines AND-Gatters
- 3.3 Schnittstelle und Implementierung
- 3.4 Simulationsumgebung für diese Vorlesung



# Steckplatinen-Experimente



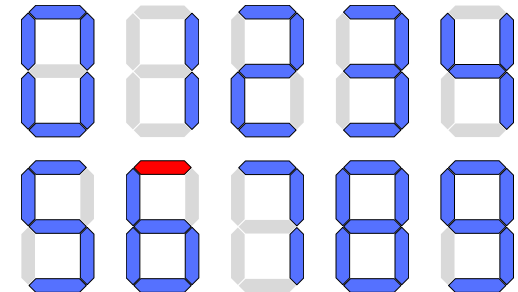
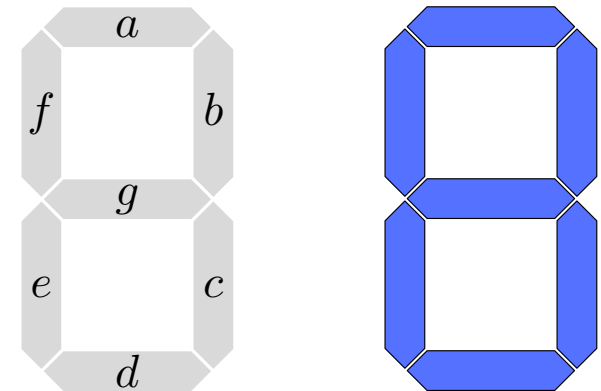
www.gameover.com.mx



# Steckplatten-Experimente: 7-Segment-Anzeige

Eingabe				Segmente						
A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

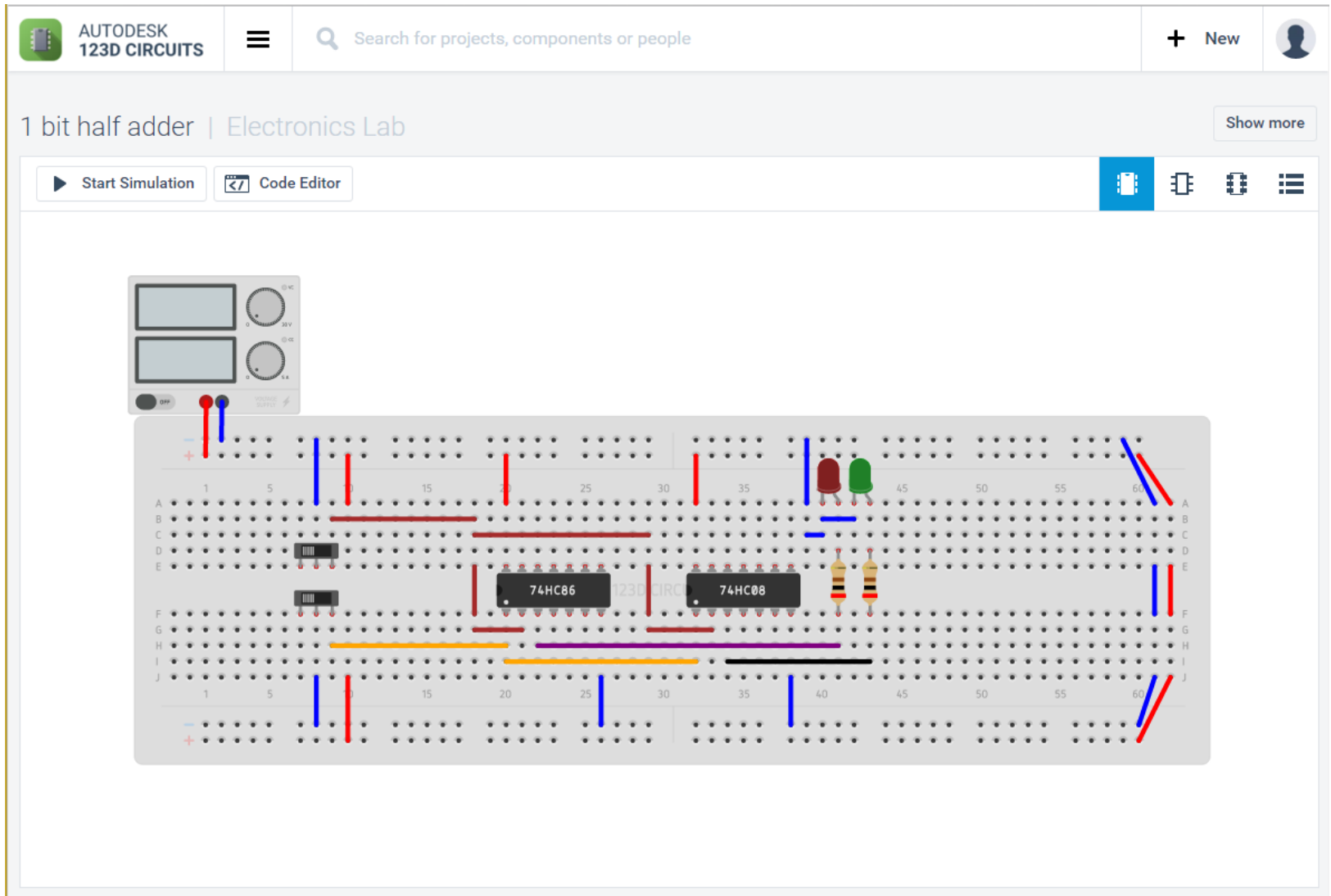
7-Segment-Digitalanzeige



- Der integrierte Schaltkreis Ti 7448 (und verwandte Typen) vereinfacht die Ansteuerung einer 7-Segment-Anzeige.
- Durch Ripple Blanking Input/Output Signal können bei mehreren Stellen führende Nullen unterdrückt werden.

# Steckplatten-Simulator

<http://123d.circuits.io>

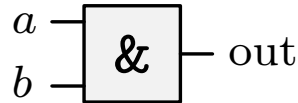


# Hardware-Beschreibungssprache

- **Steckplatinen** skalieren nicht besonders gut:  
Größere Schaltungen sind damit aufwendig zu bauen  
und noch wesentlich aufwendiger gründlich zu testen.
- Bessere Lösung: **Hardware-Simulatoren in Software.**
  - Idee: Beschreibe Gatterschaltungen in einer Sprache, der **Hardware-Beschreibungssprache** (hardware description language, HDL).
  - In dieser Sprache spezifiziert man das Verhalten von Bausteinen (Schnittstelle), sowie ihre Implementierung (Aufbau aus Elementargattern oder bereits erstellten anderen Bausteinen).
  - Ein **Hardware-Simulator** kann die Elementargatter und damit (indirekt) alle aus ihnen zusammengesetzten Bausteine simulieren, indem er die Implementierungsbeschreibungen interpretiert und ausführt.
  - Das simulierte Verhalten kann automatisch mit der spezifizierten Schnittstelle verglichen werden: **automatische Fehlerprüfung.**

# HDL: Aufbau eines AND-Gatters

And.cmp		
<i>a</i>	<i>b</i>	<i>out</i>
0	0	0
0	1	0
1	0	0
1	1	1



Ziel: Aufbau eines AND-Gatters  
aus elementarerer Gattern;  
Test dieses Aufbaus.

```
And.hdl
CHIP And
{
  IN a, b;
  OUT out;
  // implementation missing
}
```

```
And.tst
load And.hdl,
output-file And.out,
compare-to And.cmp,
output-list a b out;
set a 0, set b 0, eval, output;
set a 0, set b 1, eval, output;
set a 1, set b 0, eval, output;
set a 1, set b 1, eval, output;
```

Hardware-Beschreibung eines Bauteils durch drei Dateien:

- \*.cmp Beschreibung der Schnittstelle durch Ein-/Ausgabetupel.
- \*.hdl Beschreibung der Implementierung durch Gatterzusammenschaltung.
- \*.tst Befehle zum Durchführen eines Funktionstests.

# HDL: Aufbau eines AND-Gatters

**Schnittstelle:**  $\text{And}(a, b) = 1$  genau dann, wenn  $a = b = 1$ .



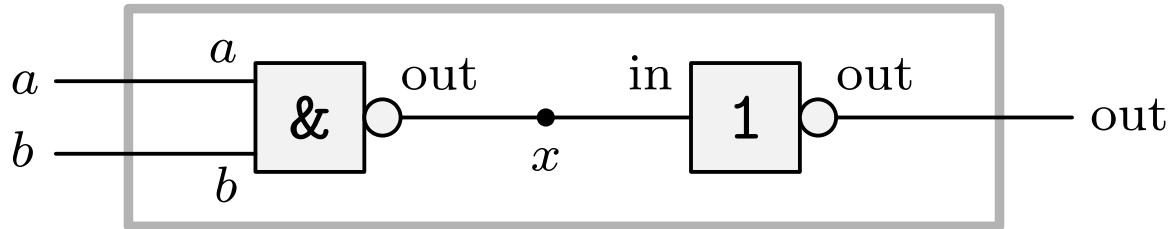
And.hdl

```
CHIP And
{
    IN a, b;
    OUT out;
    // implementation missing
}
```

- Die Schnittstelle bestimmt die Ein- und Ausgänge des Bausteins.
- Die Implementierung beschreibt, wie die Ein- und Ausgänge mit Gattern (und diese miteinander) verbunden sind.

# HDL: Aufbau eines AND-Gatters

**Implementierung:**  $\text{And}(a, b) = \text{Not}(\text{Nand}(a, b))$ .



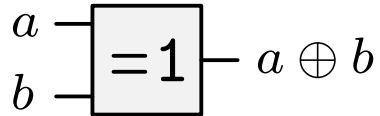
And.hdl

```
CHIP And
{
  IN a, b;
  OUT out;
  PARTS:
    Nand(a = a, b = b, out = x);
    Not(in = x, out = out);
}
```

- Verbindungsbeschreibung erfordert u.U. zusätzliche Variablen (pins).

# Gatter-/Schaltungslogik

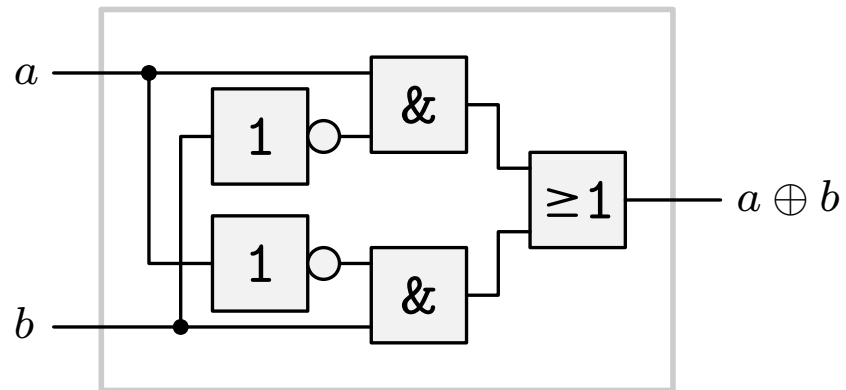
## Schnittstelle (interface)



$a$	$b$	$a \oplus b$
0	0	0
1	0	1
0	1	1
1	1	0

## Implementierung der Schnittstelle

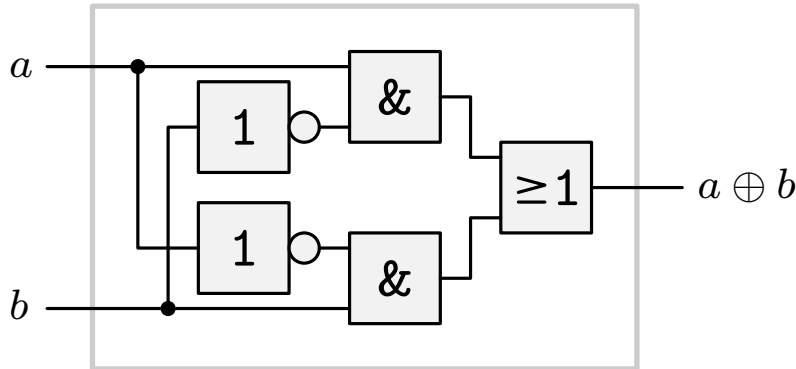
$$a \oplus b \equiv (a \wedge \neg b) \vee (\neg a \wedge b)$$



- Weiteres Beispiel: Schnittstelle eines XOR-Gatters (Wahrheitstafel) und seine Implementierung durch zusammengeschaltete andere Gatter.
- Die Implementierung beruht auf einer möglichen logischen Äquivalenz, durch die das Exklusive Oder dargestellt werden kann.

# Hardware-Beschreibungssprache

## Implementierung



## HDL-Beschreibung

```
CHIP Xor {  
  IN a, b;  
  OUT out;  
  PARTS:  
    Not(in=a,out=Nota);  
    Not(in=b,out=Notb);  
    And(a=a,b=Notb,out=w1);  
    And(a=Nota,b=b,out=w2);  
    Or(a=w1,b=w2,out=out);  
}
```

- Die Implementierung des XOR-Gatters wird nun wieder in der **Hardware-Beschreibungssprache** (hardware description language, HDL) angegeben. Dies ermöglicht u.a. eine automatische Prüfung der Implementierung.
- Bestimmte Gatter (hier: NAND, NOR und NOT) sind vorgegeben.



# Hardware-Simulator

Hardware Simulator - D:\hack\Chips\Project 1\Xor.hdl

File View Run Help

Animate: Program flow Format: Decimal View: Script

Chip Name: Time: 0

Input pins		Output pins	
Name	Value	Name	Value
a	0	out	0
b	0		

Internal pins	
Name	Value
nota	1
notb	1
x	0
y	0

```
// Xor (exclusive or) gate
// if a<>b out=1 else out=0
CHIP Xor {
  IN a,b;
  OUT out;
  PARTS:
    Not (in=a,out=nota);
    Not (in=b,out=notb);
    And (a=a,b=notb,out=x);
    And (a=nota,b=b,out=y);
    Or (a=x,b=y,out=out);
}
```

```
load Xor,
output-file Xor.out,
compare-to Xor.cmp,
output-list a%B3.1.3 b%B3.1.3 out%B3.1.3;

set a 0,
set b 0,
eval,
output;

set a 0,
set b 1,
eval,
output;

set a 1,
set b 0,
eval,
output;

set a 1,
set b 1,
eval,
output;
```

Script restarted

# Hardware-Simulator

Hardware Simulator - D:\hack\Chips\Project 1\Xor.hdl

File View Run Help

Animate: Program flow Format: Decimal View: Script

Chip Name: Xor Time: 0

Input pins

Name	Value
a	1
b	1

Output pins

Name	Value
out	0

HDL

```
// Xor (exclusive or) gate
// if a<>b out=1 else out=0
CHIP Xor {
  IN a,b;
  OUT out;
  PARTS:
    Not (in=a,out=nota);
    Not (in=b,out=notb);
    And (a=a,b=notb,out=x);
    And (a=nota,b=b,out=y);
    Or (a=x,b=y,out=out);
}
```

Internal pins

Name	Value
nota	0
notb	0
x	0
y	0

```
load Xor,
output-file Xor.out,
compare-to Xor.cmp,
output-list a%B3.1.3 b%B3.1.3 out%B3.1.3;

set a 0,
set b 0,
eval,
output;

set a 0,
set b 1,
eval,
output;

set a 1,
set b 0,
eval,
output;

set a 1,
set b 1,
eval,
output;
```

Xor.out

a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

End of script - Comparison ended successfully

# HDL-Projekthinweise

- **The Elements of Computing Systems:  
Building a Modern Computer from First Principles**

Noam Nisan & Shimon Schocken

MIT Press, Cambridge, MA, USA 2008

Die Übungen zu HDL (und auch anderen Lehrinhalten)  
orientieren sich sehr stark an diesem Buch.

- Lesen Sie die Kapitel 1 und 2 dieses Buches!  
(Universitätsbibliothek, Kapitel auch im WWW als PDF-Dateien verfügbar)
- Für die Übungen zu HDL laden Sie die Software zu diesem Buch herunter:

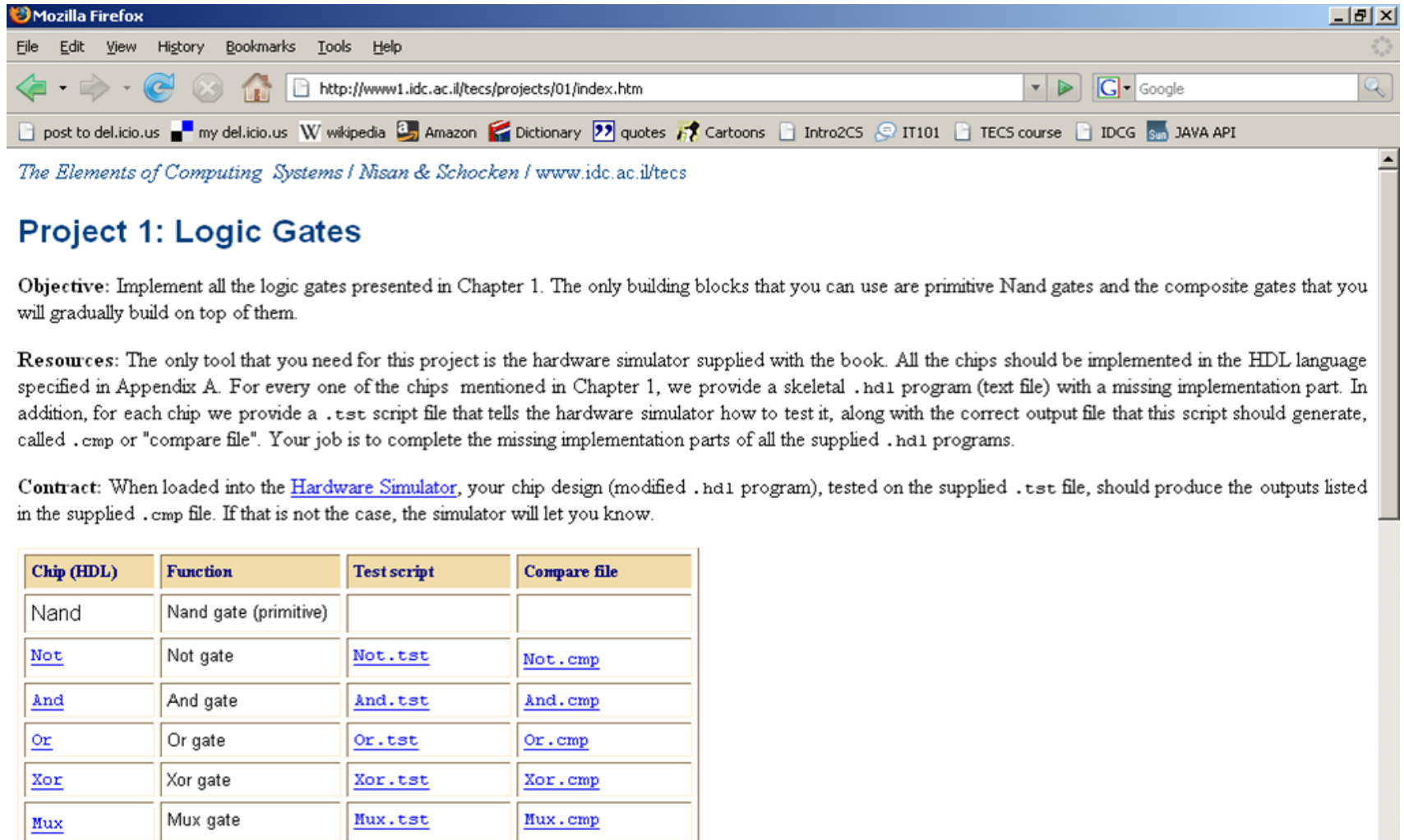
`http://nand2tetris.org/software.php`

- Arbeiten Sie das Tutorial zu dem Hardware-Simulator durch!
- Bearbeiten Sie die Projekte auf

`http://www1.idc.ac.il/tecs/projects/01/index.htm`

# HDL-Projektunterlagen

<http://www1.idc.ac.il/tecs/projects/01/index.htm>



The screenshot shows a Mozilla Firefox browser window. The address bar displays the URL <http://www1.idc.ac.il/tecs/projects/01/index.htm>. The page title is "The Elements of Computing Systems / Nisan & Schocken / www.idc.ac.il/tecs". The main heading is "Project 1: Logic Gates". Below this, there are three paragraphs: "Objective", "Resources", and "Contract". At the bottom, there is a table with four columns: "Chip (HDL)", "Function", "Test script", and "Compare file". The table lists six logic gates: Nand, Not, And, Or, Xor, and Mux, each with its function, test script, and compare file.

**Objective:** Implement all the logic gates presented in Chapter 1. The only building blocks that you can use are primitive Nand gates and the composite gates that you will gradually build on top of them.

**Resources:** The only tool that you need for this project is the hardware simulator supplied with the book. All the chips should be implemented in the HDL language specified in Appendix A. For every one of the chips mentioned in Chapter 1, we provide a skeletal .hdl program (text file) with a missing implementation part. In addition, for each chip we provide a .tst script file that tells the hardware simulator how to test it, along with the correct output file that this script should generate, called .cmp or "compare file". Your job is to complete the missing implementation parts of all the supplied .hdl programs.

**Contract:** When loaded into the [Hardware Simulator](#), your chip design (modified .hdl program), tested on the supplied .tst file, should produce the outputs listed in the supplied .cmp file. If that is not the case, the simulator will let you know.

Chip (HDL)	Function	Test script	Compare file
Nand	Nand gate (primitive)		
<a href="#">Not</a>	Not gate	<a href="#">Not.tst</a>	<a href="#">Not.cmp</a>
<a href="#">And</a>	And gate	<a href="#">And.tst</a>	<a href="#">And.cmp</a>
<a href="#">Or</a>	Or gate	<a href="#">Or.tst</a>	<a href="#">Or.cmp</a>
<a href="#">Xor</a>	Xor gate	<a href="#">Xor.tst</a>	<a href="#">Xor.cmp</a>
<a href="#">Mux</a>	Mux gate	<a href="#">Mux.tst</a>	<a href="#">Mux.cmp</a>

# Zusammenfassung

- **Minimierung Boolescher Formeln**
  - Äquivalenzumformungen mit Hilfe der Booleschen Gesetze
  - Das Minimierungsverfahren von Karnaugh–Veitch (Karnaugh–Veitch-Diagramme)
  - Das Minimierungsverfahren von Quine–McCluskey (Methode der Primimplikanten, Petricks Algorithmus)
- **Programmierbare Logikarrays**  
(programmable logic arrays, PLAs)
  - Implementierung von Disjunktionen von Konjunktionen von Literalen
- **Hardware-Beschreibungssprache**  
(hardware description language, HDL)
  - Beschreibung der Zusammenschaltung von Gattern (Schnittstelle und Implementierung)
  - Simulationsumgebung für Gatterschaltungen (Schaltungstest auf Grundlage einer Spezifikation)