

Digitaltechnik

1 Kombinatorische Schaltungen



Revision 1.01

Boole'sche Algebra

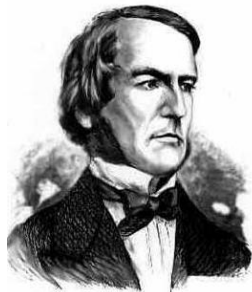
Gatter

Rechenregeln

Minimierung kombinatorischer Schaltungen

Kombinatorische Schaltungen in Verilog

- ▶ Kombinatorische Schaltungen haben *Eingänge* und *Ausgänge*, und sind **zyklusfrei**
- ▶ Abstraktion: Die Werte der Ausgänge sind durch eine **Boole'sche Funktion** der Eingänge gegeben
- ✗ Wirklichkeit: Schaltverzögerung! (später)



George Boole
(1815 – 1864)

- ▶ Zwei Wahrheitswerte: $\{0, 1\}$
(Konstanten)
- ▶ Variablen: $V = \{x, y, z, \dots\}$
- ▶ Boolesche Operatoren:
 $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$
(absteigende Priorität)
- ▶ Rechenregeln (später)

Semantik der Operatoren:

x	y	$\neg x$	$x \wedge y$	$x \vee y$	$x \rightarrow y$	$x \leftrightarrow y$
0	0	1	0	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	0	0
1	1	0	1	1	1	1

- ▶ eine *Belegung* β ist eine Abbildung $\beta: V \rightarrow \{0, 1\}$
- ▶ eine Belegung wird rekursiv über die Formel-Struktur ausgewertet
- ▶ eine Belegung β *erfüllt* eine Formel f gdw. $\beta(f) = 1$

- ▶ eine *Belegung* β ist eine Abbildung $\beta: V \rightarrow \{0, 1\}$
- ▶ eine Belegung wird rekursiv über die Formel-Struktur ausgewertet
- ▶ eine Belegung β *erfüllt* eine Formel f gdw. $\beta(f) = 1$

- ▶ eine Formel heißt **erfüllbar** gdw. es eine erfüllende Belegung gibt (engl. satisfiable $\Rightarrow$ SAT)
- ▶ eine Formel heißt **Tautologie** gdw. alle Belegungen erfüllend sind
- ▶ Formel g wird von Formel f **impliziert** ($f \Rightarrow g$) gdw. $f \rightarrow g$ Tautologie
- ▶ Formeln f und g sind **äquivalent** ($f \equiv g$) gdw. $f \leftrightarrow g$ Tautologie

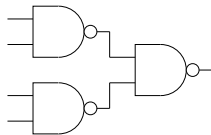
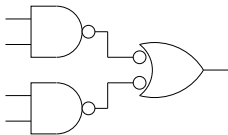
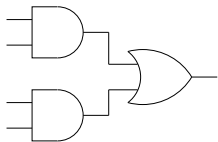
- ▶ nicht alle Operatoren sind notwendig
- ▶ eine *Basis* B ist eine funktionale minimale Anzahl von Operatoren
- ▶ funktional, heißt alle anderen lassen sich damit darstellen

- ▶ nicht alle Operatoren sind notwendig
- ▶ eine *Basis* B ist eine funktionale minimale Anzahl von Operatoren
- ▶ funktional, heißt alle anderen lassen sich damit darstellen
- ▶ Beispiel: $x \vee y \equiv \overline{\overline{(x \wedge x)} \wedge \overline{(y \wedge y)}}$ wenn nur NAND vorhanden
- ▶ technologisch bedingt ist das NAND sehr beliebt
- ▶ Software Repräsentation von Schaltkreisen mit $B = \{\neg, \wedge\}$
 $\Rightarrow$ *Signed-And-Graphs*

Bezeichnung (Verilog)	Boolesche Logik	Algebraische Schreibweise	Traditionelles Schaltbild	IEEE Schaltbild
Disjunktion ()	$x \vee y$	$x + y$		
Negierte Disjunktion (nicht in Verilog)	$\neg(x \vee y)$	$\overline{x + y}$		
Konjunktion (& &)	$x \wedge y$	$x \cdot y$ (oder $x.y, xy, \dots$)		
Negierte Konjunktion (nicht in Verilog)	$\neg(x \wedge y)$	$\overline{x \cdot y}$		

Bezeichnung (Verilog)	Boolesche Logik	Algebraische Schreibweise	Traditionelles Schaltbild	IEEE Schaltbild
Exklusiv-Oder ($\wedge$)	$x \nleftrightarrow y$ (oder $x \neq y$)	$x \oplus y$		
Äquivalenz ($=$)	$x \leftrightarrow y$ (oder $x = y$)	$\overline{x \oplus y}$		
Implikation (nicht in Verilog)	$x \rightarrow y$			
Negation ($!$)	$\neg x$	$\overline{x}$ (oder x')		

	Konjunktive Formulierung	Disjunktive Formulierung
Kommutativität	$x \wedge y \equiv y \wedge x$	$x \vee y \equiv y \vee x$
Assoziativität	$x \wedge (y \wedge z) \equiv (x \wedge y) \wedge z$	$x \vee (y \vee z) \equiv (x \vee y) \vee z$
Distributivität	$x \wedge (y \vee z) \equiv (x \wedge y) \vee (x \wedge z)$	$x \vee (y \wedge z) \equiv (x \vee y) \wedge (x \vee z)$
De'Morgan	$\neg(x \wedge y) \equiv \neg x \vee \neg y$	$\neg(x \vee y) \equiv \neg x \wedge \neg y$
Idempotenz	$x \wedge x \equiv x$	$x \vee x \equiv x$
Controlling Value	$x \wedge 0 \equiv 0$	$x \vee 1 \equiv 1$
Neutraler Wert	$x \wedge 1 \equiv x$	$x \vee 0 \equiv x$
Doppelte Negation		$\neg(\neg x) \equiv x$



Und-Oder Schaltung lässt sich einfach als NAND-Schaltung darstellen!

Shannonsches Expansionstheorem (Fallunterscheidung)

$$e \equiv x \wedge e[1/x] \vee \neg x \wedge e[0/x]$$

mit

- ▶ e, f Boolesche Ausdrücke,
- ▶ x Variable,
- ▶ $e[f/x]$ Substitution von x durch f in e .

Beispiel: (zwei Darstellungen für Exklusiv-Oder)

$$(x \vee y) \wedge (\neg x \vee \neg y)$$

$$\equiv x \wedge (1 \vee y) \wedge (\neg 1 \vee \neg y) \quad \vee \quad \neg x \wedge (0 \vee y) \wedge (\neg 0 \vee \neg y)$$

$$\equiv x \wedge 1 \wedge (0 \vee \neg y) \quad \vee \quad \neg x \wedge y \wedge (1 \vee \neg y)$$

$$\equiv x \wedge 1 \wedge \neg y \quad \vee \quad \neg x \wedge y \wedge 1$$

$$\equiv x \wedge \neg y \vee \neg x \wedge y$$

Algebraische Schreibweise: (= statt $\equiv$ ist auch erlaubt)

$$(x + y) \cdot (\bar{x} + \bar{y}) \equiv x \cdot \bar{y} + \bar{x} \cdot y$$

Abschwächung $x \cdot y \equiv x$, falls $x \Rightarrow y$ (y schwächer als x).

Abschwächung $x \cdot y \equiv x$, falls $x \Rightarrow y$ (y schwächer als x).

Verstärkung $x + y \equiv x$, falls $y \Rightarrow x$ (y stärker als x).

Abschwächung $x \cdot y \equiv x$, falls $x \Rightarrow y$ (y schwächer als x).

Verstärkung $x + y \equiv x$, falls $y \Rightarrow x$ (y stärker als x).

Konsensus (Fallunterscheidung und Abschwächung)

$$\boxed{x \cdot y + \bar{x} \cdot z} + y \cdot z \equiv x \cdot y + \bar{x} \cdot z$$

Intuition: **If-Then-Else** schwächer als **Konjunktion** von
Then und Else.

x	y	z	$x \cdot y + \bar{x} \cdot z$	$x \cdot y + \bar{x} \cdot z + y \cdot z$
0	0	0	0	0
0	0	1	1	1
0	1	0	0	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

✗ Mühsam für viele Variablen (exponentielle Laufzeit)!

$$\begin{aligned}(x \cdot y + \bar{x} \cdot z + y \cdot z)[0/x] &\equiv 0 \cdot y + \bar{0} \cdot z + y \cdot z && \text{Substitution} \\ &\equiv 0 + 1 \cdot z + y \cdot z && \text{Controlling Value} \cdot \\ &\equiv 0 + z + y \cdot z && \text{Non-Controlling Value} \cdot \\ &\equiv z + y \cdot z && \text{Non-Controlling Value} + \\ &\equiv 1 \cdot z + y \cdot z && \text{Non-Controlling Value} \cdot \\ &\equiv (1 + y) \cdot z && \text{Distributivität} \cdot \\ &\equiv z && \text{Controlling Value} +\end{aligned}$$

$$\begin{aligned}(x \cdot y + \bar{x} \cdot z + y \cdot z)[0/x] &\equiv 0 \cdot y + \bar{0} \cdot z + y \cdot z && \text{Substitution} \\ &\equiv 0 + 1 \cdot z + y \cdot z && \text{Controlling Value} \cdot \\ &\equiv 0 + z + y \cdot z && \text{Non-Controlling Value} \cdot \\ &\equiv z + y \cdot z && \text{Non-Controlling Value} + \\ &\equiv 1 \cdot z + y \cdot z && \text{Non-Controlling Value} \cdot \\ &\equiv (1 + y) \cdot z && \text{Distributivität} \cdot \\ &\equiv z && \text{Controlling Value} +\end{aligned}$$

Analog erhält man $(x \cdot y + \bar{x} \cdot z + y \cdot z)[1/x] \equiv y$

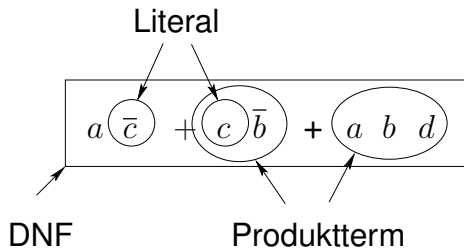
$$\begin{aligned}(x \cdot y + \bar{x} \cdot z + y \cdot z)[0/x] &\equiv 0 \cdot y + \bar{0} \cdot z + y \cdot z && \text{Substitution} \\ &\equiv 0 + 1 \cdot z + y \cdot z && \text{Controlling Value} \cdot \\ &\equiv 0 + z + y \cdot z && \text{Non-Controlling Value} \cdot \\ &\equiv z + y \cdot z && \text{Non-Controlling Value} + \\ &\equiv 1 \cdot z + y \cdot z && \text{Non-Controlling Value} \cdot \\ &\equiv (1 + y) \cdot z && \text{Distributivität} \cdot \\ &\equiv z && \text{Controlling Value} +\end{aligned}$$

Analog erhält man $(x \cdot y + \bar{x} \cdot z + y \cdot z)[1/x] \equiv y$

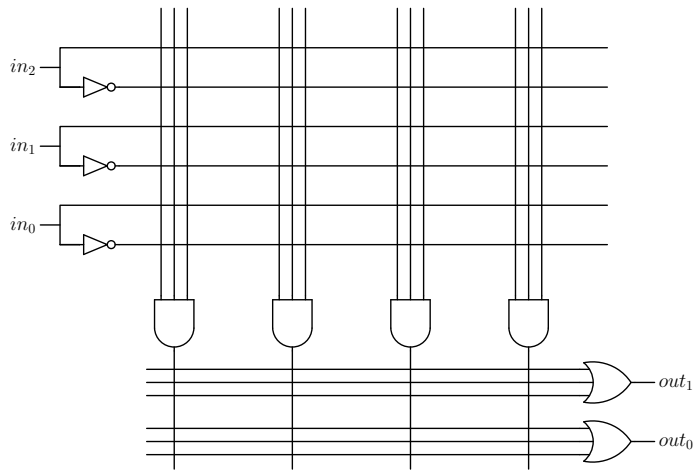
Mit Shannonschem Expansionstheorem:

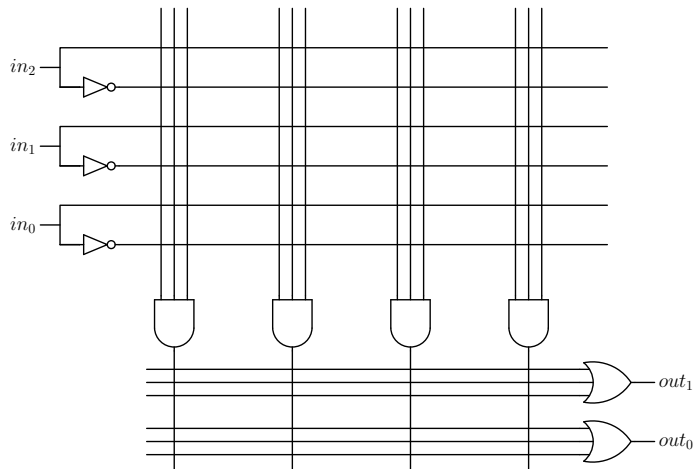
$$\begin{aligned}x \cdot y + \bar{x} \cdot z + y \cdot z &\equiv x \cdot (x \cdot y + \bar{x} \cdot z + y \cdot z)[1/x] + \\ &\quad \bar{x} \cdot (x \cdot y + \bar{x} \cdot z + y \cdot z)[0/x] \\ &\equiv x \cdot y + \bar{x} \cdot z\end{aligned}$$

- ▶ gegeben eine feste Menge von Variablen
- ▶ ein *Literal* ist eine negierte oder unnegierte Variable, z.B. x , $\bar{x}$, a , $\bar{z}$,
...
- ▶ ein *Produktterm* ist eine Konjunktion von Literalen, z.B. $x \cdot y$, $\bar{b} \cdot c$, ...
- ▶ ein *Minterm* ist ein maximaler Produktterm (enthält alle Variablen)
- ▶ eine *DNF* ist eine Disjunktion von Produkttermen (wie im Beispiel)



man bezeichnet eine DNF auch als ein *Polynom*
die Produktterme heißen dann Monome





PLAs implementieren DNF

Annahme: Chipgröße in etwa linear in der Größe der Formel

Gegeben: Zu implementierende Funktion f dargestellt als Formel

Gesucht: (möglichst) minimal große Formel g mit $f \equiv g$

Annahme: Chipgröße in etwa linear in der Größe der Formel

Gegeben: Zu implementierende Funktion f dargestellt als Formel

Gesucht: (möglichst) minimal große Formel g mit $f \equiv g$

Beispiel:

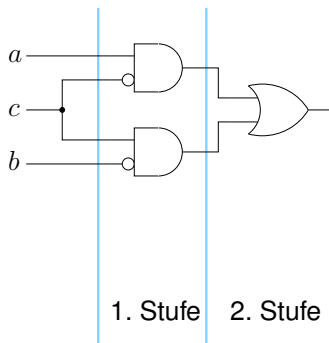
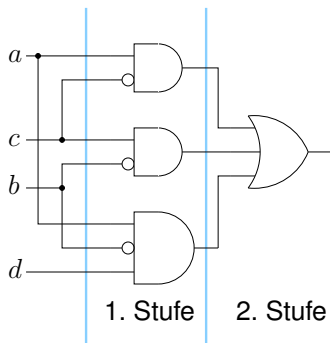
$$a\bar{c} \vee c\bar{b} \vee a\bar{b}d$$

Minimierung:

$$a\bar{c} \vee c\bar{b} \vee a\bar{b}d \equiv a\bar{c} \vee c\bar{b} \vee a\bar{b} \vee a\bar{b}d \quad \text{Konsensus über } c$$

$$\equiv a\bar{c} \vee c\bar{b} \vee a\bar{b} \quad \text{Verstärkung}$$

$$\equiv a\bar{c} \vee c\bar{b} \quad \text{Konsensus über } c$$



(gleiches Beispiel wie bei der Minimierung)

- ▶ auch Karnaugh-Veitch-Diagramme, KV-Diagramme, Karnaugh-Maps, etc.
- ▶ praktikabel bis zu 5 Variablen
- ▶ 2-Dimensionale Anordnung einer Funktionstabelle
- ▶ benachbarte Zellen unterscheiden sich in einem Bit

		d	
c		0 ₀	0 ₁
		0 ₂	1 ₃

2 Variablen

MinTerm Index	c	d	$c \wedge d$
0	0	0	0
1	0	1	0
2	1	0	0
3	1	1	1

Erzeugung des Gitters von Karnaugh-Diagrammen

d	
0	1

1 Variable

d	
0	1
2	3

c

2 Variablen

		d			
c		0	1	5	4
		2	3	7	6
				b	

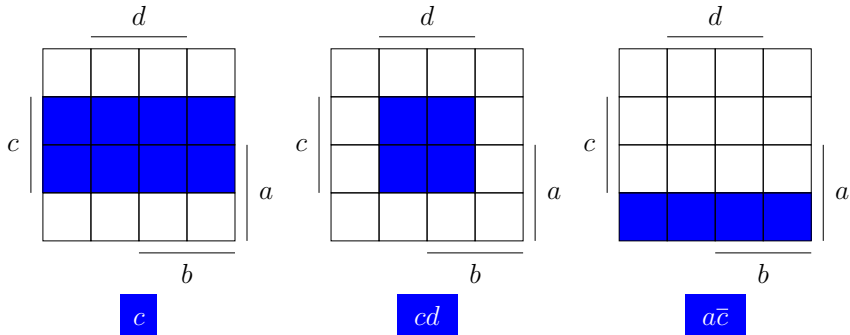
3 Variablen

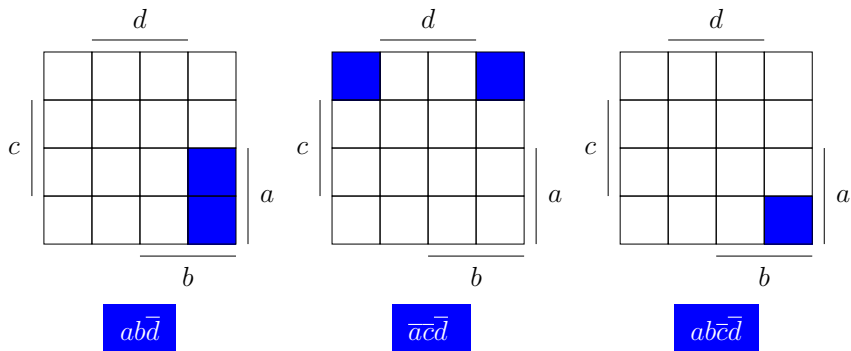
d			
0	1	5	4
2	3	7	6
10	11	15	14
8	9	13	12
b			

c

a

4 Variablen





- ① Hinreichend großes Gitternetz erstellen
- ② Positive Literale markieren
- ③ Zu maximalen Blöcken zusammenfassen
- ④ Minimale Überdeckung ablesen

Zur Übung:

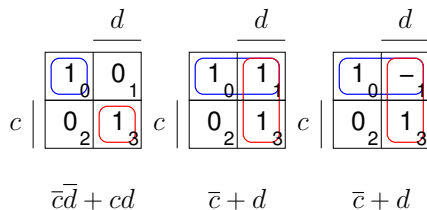
`http:`

`//tech-www.informatik.uni-hamburg.de/applets/kvd/`

Karnaugh-Diagramme – Beispiel

d				
0	0	0	0	
1	1	0	0	
2	1	0	0	
3	1	0	0	
4	1	1	0	
5	1	1	0	
6	1	1	1	
7	1	1	1	
8	1	1	1	
9	1	1	1	
10	1	1	1	
11	1	1	1	
12	1	1	1	
13	1	1	1	
14	1	1	1	
15	1	1	1	

Minterm Index	a	b	c	d	$a\bar{c} + \bar{c}\bar{b} + \bar{a}bd$
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	1
3	0	0	1	1	1
4	0	1	0	0	0
5	0	1	0	1	0
6	0	1	1	0	0
7	0	1	1	1	0
8	1	0	0	0	1
9	1	0	0	1	1
10	1	0	1	0	1
11	1	0	1	1	1
12	1	1	0	0	1
13	1	1	0	1	1
14	1	1	1	0	0
15	1	1	1	1	0



MinTerm Index	c	d	f
0	0	0	1
1	0	1	–
2	1	0	0
3	1	1	1

Don't Cares “–” können zur Vergrößerung von Blöcken verwendet werden
(müssen aber nicht abgedeckt werden)

- ▶ ein **Implikant** einer DNF ist ein implizierender Produktterm
z.B. ein Monom oder der Konsensus zweier Monome
Block im Karnaugh-Diagramm
- ▶ ein **Primimplikant** ist ein maximaler Implikant
entspricht maximalen Block, der in keine Richtung erweitert werden kann
- ▶ ein **Kernimplikant** überdeckt einen sonst nicht überdeckten Minterm
- ▶ ein **redundanter** Primimplikant wird von Kernimplikanten überdeckt
z.B. der Konsensus von zwei Primimplikanten

Einfache Konsensusregel

$$x \cdot y + \bar{x} \cdot y \equiv y$$

(Beweis im Buch)

Auch für mehr als zwei Variablen:

$$\bar{a} \cdot b \cdot \bar{c} \cdot d + a \cdot b \cdot \bar{c} \cdot d \equiv b \cdot \bar{c} \cdot d$$

1. Generiere alle Primimplikanten durch einfache Konsensusregel
2. Gib Kernimplikanten aus, da in jeder minimalen DNF enthalten
3. Entferne redundante Primimplikanten
4. Wähle minimale Überdeckung aus restlichen Primimplikanten

Gegeben Funktionstabelle

Minterm Index	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	Funktions- wert	Minterm Index	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	Funktions- wert
0	0	0	0	0	1	8	1	0	0	0	1
1	0	0	0	1	0	9	1	0	0	1	0
2	0	0	1	0	1	10	1	0	1	0	1
3	0	0	1	1	0	11	1	0	1	1	0
4	0	1	0	0	0	12	1	1	0	0	1
5	0	1	0	1	1	13	1	1	0	1	1
6	0	1	1	0	0	14	1	1	1	0	0
7	0	1	1	1	0	15	1	1	1	1	1

(Achtung: anderes Beispiel als beim Karnaugh-Diagramm)

Generierung aller Primimplikanten Schritt 1

Minterm Indizes	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	Anzahl Einsen	Primimplikant
0	0	0	0	0	0	
2	0	0	1	0	1	
8	1	0	0	0	1	
5	0	1	0	1	2	
10	1	0	1	0	2	
12	1	1	0	0	2	
13	1	1	0	1	3	
15	1	1	1	1	4	

(Gruppeneinteilung nach Anzahl Einsen reduziert Konsensusversuche)

Generierung aller Primimplikanten Schritt 1

Minterm Indizes	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	Anzahl Einsen	Primimplikant
0	0	0	0	0	0	nein
2	0	0	1	0	1	nein
8	1	0	0	0	1	nein
5	0	1	0	1	2	nein
10	1	0	1	0	2	nein
12	1	1	0	0	2	nein
13	1	1	0	1	3	nein
15	1	1	1	1	4	nein

(Gruppeneinteilung nach Anzahl Einsen reduziert Konsensusversuche)

Generierung aller Primimplikanten Schritt 2

Minterm Indizes	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	Anzahl Einsen	Primimplikant
0, 2	0	0	–	0	0	
0, 8	–	0	0	0	0	
2, 10	–	0	1	0	1	
8, 10	1	0	–	0	1	
8, 12	1	–	0	0	1	
5, 13	–	1	0	1	2	
12, 13	1	1	0	–	2	
13, 15	1	1	–	1	3	

Generierung aller Primimplikanten Schritt 2

Minterm Indizes	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	Anzahl Einsen	Primimplikant
0, 2	0	0	–	0	0	nein
0, 8	–	0	0	0	0	nein
2,10	–	0	1	0	1	nein
8,10	1	0	–	0	1	nein
8,12	1	–	0	0	1	ja
5,13	–	1	0	1	2	ja
12,13	1	1	0	–	2	ja
13,15	1	1	–	1	3	ja

Generierung aller Primimplikanten Schritt 3

Minterm Indizes	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	Anzahl Einsen	Primimplikant
0,2,8,10	–	0	–	0	0	
0,8,2,10	–	0	–	0	0	

Generierung aller Primimplikanten Schritt 3

Minterm Indizes	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	Anzahl Einsen	Primimplikant
0,2,8,10	–	0	–	0	0	ja
0,8,2,10	–	0	–	0	0	redundant

Generierung aller Primimplikanten Schritt 3

Minterm Indizes	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	Anzahl Einsen	Primimplikant
0,2,8,10	–	0	–	0	0	ja
0,8,2,10	–	0	–	0	0	redundant

Alle nicht weiter vereinfachbaren Implikanten sind Primimplikanten:

8,12	5,13	12,13	13,15	0,2,8,10
$a \cdot \bar{c} \cdot \bar{d}$	$b \cdot \bar{c} \cdot d$	$a \cdot b \cdot \bar{c}$	$a \cdot b \cdot d$	$\bar{b} \cdot \bar{d}$

Finde minimale Überdeckung mit Primimplikantentafel

	0	2	5	8	10	12	13	15
8,12				x		x		
5,13			x				x	
12,13						x	x	
13,15							x	x
0,2,8,10	x	x		x	x			

Finde minimale Überdeckung mit Primimplikantentafel

	0	2	5	8	10	12	13	15
8,12				×		×		
5,13			×				×	
12,13						×	×	
13,15							×	×
0,2,8,10	×	×		×	×			

× = Minterme in Primimplikanten

⊗ = Minterme in Kernimplikanten

⊠ = Minterme überdeckt von Kernimplikanten

Finde minimale Überdeckung mit Primimplikantentafel

	0	2	5	8	10	12	13	15
8,12				⊗		×		
5,13			⊗				⊗	
12,13						×	⊗	
13,15							⊗	⊗
0,2,8,10	⊗	⊗		⊗	⊗			

× = Minterme in Primimplikanten

⊗ = Minterme in Kernimplikanten

⊗ = Minterme überdeckt von Kernimplikanten

Minimalpolynom enthält alle Kernprimimplikanten,
aber nur entweder 8,12 oder 12,13

- ① Alle Wahrheitsbelegungen mit Wahrheitswert 1 aus der Funktionstabelle auswählen
- ② Mit der einfachen Konsensusregel *alle* Primimplikanten generieren
- ③ Kernimplikanten suchen, redundante Primimplikanten entfernen und Überdeckung aller Primimplikanten bestimmen
- ④ Die Disjunktion der Primimplikanten der Überdeckung ergibt die gewünschte minimierte Formel

Generierung Primimplikanten Schritt 1

MinTerm Index	c	d	f
0	0	0	1
1	0	1	–
2	1	0	0
3	1	1	1

Minterm Indizes	c	d	Anzahl Einsen	Primimplikant
0	0	0	0	
1	0	1	1	
3	1	1	2	

Der Don't Care Minterm 1 wird zunächst normal als Minterm behandelt

Generierung Primimplikanten Schritt 1

MinTerm Index	c	d	f
0	0	0	1
1	0	1	–
2	1	0	0
3	1	1	1

Minterm Indizes	c	d	Anzahl Einsen	Primimplikant
0	0	0	0	nein
1	0	1	1	nein
3	1	1	2	nein

Der Don't Care Minterm 1 wird zunächst normal als Minterm behandelt

Generierung Primimplikanten Schritt 2

Minterm Indizes	c	d	Anzahl Einsen	Primimplikant
0,1	0	–	0	ja
1,3	–	1	1	ja

Primimplikanten werden wie gewohnt bestimmt

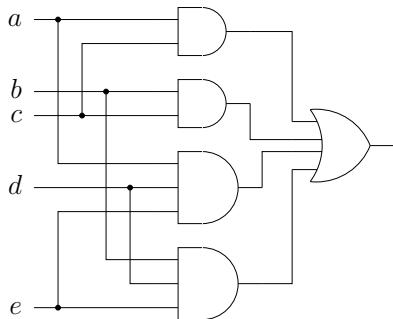
Minimale Überdeckung mit Primimplikantentafel

	0	1	3
0,1	⊗		
1,3			⊗

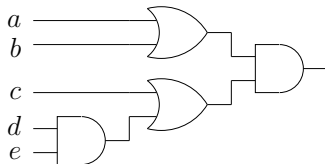
Ignoriere alle Don't Care Spalten!

- ▶ Quine-McCluskey ist exponentiell in der Anzahl Minterme da exponentiell viele Primimplikanten und Überdeckungsproblem NP-vollständig
- ▶ **Heuristiken!**
- ▶ Espresso von Berkeley als Standardverfahren:
 - Komplementierung: Berechnung der Negation einer DNF als DNF
 - Expansion: Erweitern von Implikanten zu Primimplikanten
 - Reduktion: Entfernen von redundanten Primimplikanten
 - Wiederholung: solange Kosten (z.B. Anzahl Gatter) sich verringern

Kürzere Schaltzeit



Weniger Gatter



Beide Schaltkreise sind äquivalent!

- ▶ Faktorisieren:

$a \cdot c + a \cdot d \cdot e + b \cdot c + b \cdot d \cdot e$ hat mehr Gatter als $(a + b) \cdot (c + d \cdot e)$

- ▶ führt zu mehrstufiger Logik (multi-level logic)

- ▶ Tradeoff zwischen Laufzeit und Platz (siehe später Addierer)

- ▶ keine exakten Verfahren vorhanden!

- ▶ SIS Synthese Werkzeug

<http://www-cad.eecs.berkeley.edu>

- ▶ Sprache zur Beschreibung von Schaltungen
- ▶ Übersetzung in Schaltung: **Synthese**

- ▶ Sprache zur Beschreibung von Schaltungen
- ▶ Übersetzung in Schaltung: **Synthese**
- ▶ Mehrere Stufen:
 1. Netzliste (Graph aus Gattern)
 2. Größendimensionierung der Gatter
 3. Gatter werden platziert und verdrahtet (*place and route*)
- ▶ Vor nicht allzu langer Zeit:
CAD Zeichnungen einzelner Transistoren!

- ▶ Verilog Modelle bestehen aus mehreren *Modulen*
- ▶ jedes Modul hat ein *Interface*,
das die Inputs und Outputs definiert

```
module main(input clk, schalter, output lampe);  
  
    wire x;  
    assign x=!schalter;  
5    assign lampe=x;  
  
endmodule
```

- ▶ Verbindungsmöglichkeiten werden in der Klammer beschrieben (Semikolon nach der Klammer nicht vergessen)
- ▶ Rumpf des Moduls enthält Implementierung

- ▶ **input** Eingang,
output Ausgang,
inout bidirektionale Ports
- ▶ **wire** generiert ein "Kabel"
- ▶ mit `assign x=!schalter;` wird `x` das Resultat von
`!schalter` zugewiesen (assigned)
- ▶ **!** ist $\neg$ (wie C/C++, Java, ...)

- ▶ Verilog unterscheidet Groß- und Kleinschreibung (wie C, JAVA)
- ▶ Bezeichner bestehen aus Buchstaben, Zahlen und _ (underscore)
- ▶ Escape-Mechanismus: `\$ein+fancy%Bezeichner`
- ▶ zeilenweise Kommentare beginnen mit Prefix `//`
ähnlich den `//` Kommentaren in C++
- ▶ Mehrfachzeilen-Kommentare mit `/* ... */` wie in C
- ▶ Spaces, Zeilenumschaltung, Tabulator trennen Tokens
- ▶ Semikolon definiert das Ende von Befehlen

a	b	c	z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

c			
0	0	1	0
1	1	1	0
a			

b

```
module combinational_function(input a, b, c, output z);
```

```
    assign z = (!a && b) || (a && c);
```

```
5 endmodule
```



```
module combinational_function(input a, b, c, output z);
```

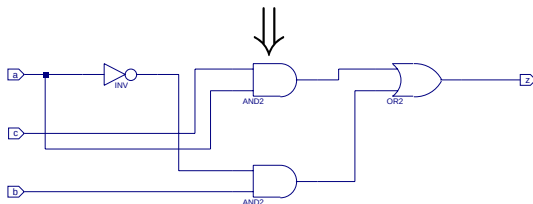
```
    assign z = (!a && b) || (a && c);
```

```
5 endmodule
```

```
module combinational_function(input a, b, c, output z);
```

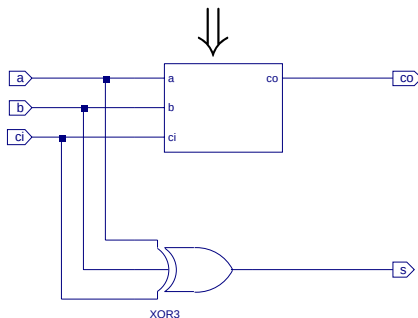
```
    assign z = (!a && b) || (a && c);
```

```
5 endmodule
```

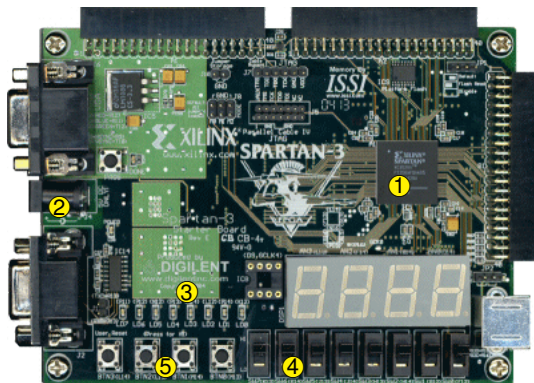


- *Synthese* des Verilog Codes mit Xilinx ISE generiert
strukturelle Netzliste
- Noch nicht technologieabhängig

```
module add (input a, b, ci, output s, co);  
  assign s = a ^ b ^ ci;  
  assign co = (a && b) || (b && ci) || (a && ci);  
endmodule
```



Kombinatorisches mit dem FPGA-Board (I)



① FPGA von Xilinx

③ Leuchtdioden (LEDs)

⑤ Taster

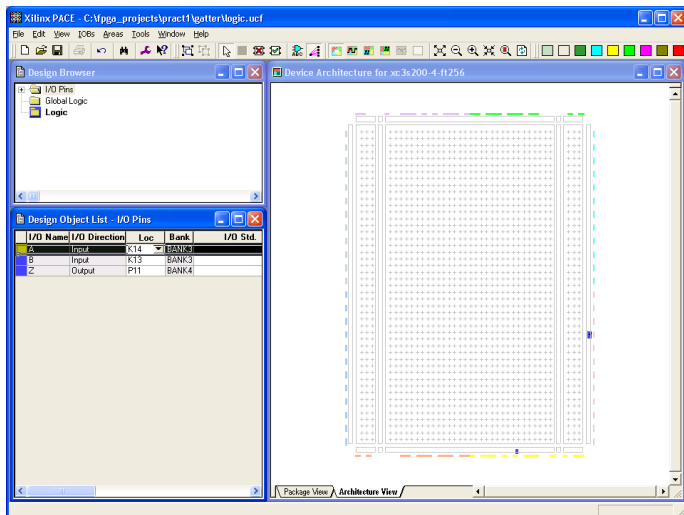
② Stromversorgung

④ Schiebeschalter

- ▶ Der Prozess 'Assign Package Pins' weist einem Input/Output ein 'Beinchen' des FPGAs zu
- ▶ Die 'Beinchen' sind fest mit den LEDs, Tastern/Schaltern verbunden

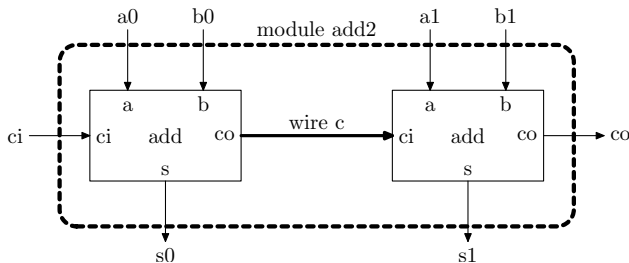
→ siehe erste Rechnerübung

Kombinatorisches mit dem FPGA-Board (III)



Module können zu größeren Modulen kombiniert werden:

```
module add2 (input a0, b0, a1, b1, ci,  
             output s0, s1, co);  
    wire c;  
    add A1(a0, b0, ci, s0, c);  
    add A2(a1, b1, c, s1, co);  
endmodule
```



Es gibt eine Alternative zu **wire** und **assign**:

```
module also_comb(input a, b, ci, output s, co);  
  
    reg s, co;  
  
5    always @(a, b, ci) begin  
        s = a ^ b ^ ci;  
        co = (a && b) || (b && ci) || (a && ci);  
    end  
  
10 endmodule
```

Ählich wie sequentielle Programmierspache, aber **kombinatorisch**!


```
module ifthen_example(input a, b, c, output x);
```

```
    reg x;
```

```
5    always @(a, b, c) begin
```

```
        x=0;
```

```
        if (a || b || c) x=1;
```

```
    end
```

```
10 endmodule
```



- ▶ Mehrfachzuweisungen möglich
- ▶ Jedes Ausgangssignal muss auf *allen Pfaden* zugewiesen werden!
- ▶ Die *Sensitivitätsliste* @ (. . . .) muss vollständig sein!

```
module thing(input [4:0] in1, in2, input s,  
             output reg [4:0] out1);  
  
    integer i;  
5    reg [4:0] t2, t1;  
  
    always @(in1, in2, s) begin  
        i=4;  
        while (i>=0) begin  
10         t1[i] = s && in1[i];  
           t2[i] = !s && in2[i];  
           i = i - 1;  
        end  
        out1 = t1 | t2;  
15    end  
  
endmodule
```

Welches Bauteil stellt `thing` dar?