

# Kapitel 1 – Grundlagen

## **1. Mathematische Grundlagen**

### 2. Beispielrechner ReTi

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur  
WS 2016/17

# Mathematische Grundlagen

---

- Verständigung auf gemeinsame Basis
- Die meisten Begriffe sollten bekannt sein, bzw. werden in anderen Vorlesungen noch formal und im Detail eingeführt.
- Hier: Informale, möglichst intuitive Einführung
  - Mengen, Funktionen, Relationen
  - Boolesche Algebra ( $\{0, 1\}$ ,  $\wedge$ ,  $\vee$ ,  $\neg$ )
  - Graphen, O-Notation
  - Beweistechniken

# „Philosophie“ der Mathematik

---

*die gelten*

- Gegeben gewisse Aussagen (Axiome), welche andere Aussagen lassen sich aus ihnen herleiten?
- Sind die Axiome wahr und existiert eine solche Herleitung (Beweis), so sind die Folgerungen unumstößlich und indiskutabel wahr!
- Beschreiben die Axiome etwa ein physikalisches System, so gelten die hergeleiteten Folgerungen für dieses System.
- Die Frage, ob Axiome Realitätsbezug haben, ist aber außerhalb der (reinen) Mathematik!

# Menge (Naive Definition)

## Definition

Eine **Menge** ist eine Zusammenfassung von wohldefinierten, paarweise verschiedenen Objekten zu einem Ganzen.

- Die Objekte nennt man Elemente der Menge.  
(Für eine formal vollständige Definition der Menge bräuchte man mehrere Vorlesungsstunden.)
  
- Notation: Sind  $a_1, a_2, \dots, a_n$  paarweise verschieden, so schreibt man die Menge  $M$ , die aus ihnen besteht, als  
$$M = \{a_1, a_2, \dots, a_n\}.$$
  - $a_i \in M$  bezeichnet, dass  $a_i$  Element von  $M$  ist.

# Beispiele für Mengen

---

- Leere Menge: ~~∅~~ (es gibt kein  $a \in \emptyset$ ).  $\emptyset$
- Menge der natürlichen Zahlen:  $\mathbb{N} = \{0, 1, 2, \dots\}$ .
- Menge der booleschen Werte:  $\mathbb{B} = \{0, 1\}$ .  
*falsch      wahr*
- Achtung: Die Anordnung von Elementen der Menge und gegebenenfalls Wiederholungen sind belanglos:  
 $\{a, b, c\} = \{c, a, b\} = \{a, a, b, c, a, b\}$ .
- Eine Menge kann Elemente enthalten, die selber Mengen sind, z.B.  $\{a, b, \{a\}, \{a, b\}\}$  enthält 4 Elemente, wobei  
*| | T T*      *die letzten beiden wieder Mengen sind.*

# Spezifikation von Mengen

- Man kann eine Menge durch Angabe von **Zusatzbedingungen** spezifizieren.

Beispiele:

- Menge der ganzen Zahlen:

$$\mathbb{Z} = \{z, -z \mid z \in \mathbb{N}\}.$$

- Menge der rationalen Zahlen:

$$\mathbb{Q} = \{p/q \mid p \in \mathbb{N}, q \in \mathbb{Z}, q \neq 0, p, q \text{ teilerfremd}\}.$$

- Menge der endlichen Zeichenketten:

$$\text{STRINGS} = \{s_1 s_2 \dots s_n \mid n \in \mathbb{N}, s_i \text{ ein Buchstabe}\}.$$

$$\{0, 1, 2, 3, 4, \dots\}$$

Menge d. Brüche  
bedeutet, dass sich  
p, q nur durch  
1 teilen lassen.

$$\frac{4}{6} = \frac{2}{3}$$

nicht tf

# Untermengen, Potenzmenge, Mächtigkeit

"C"

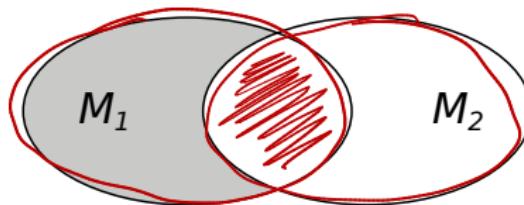
Menge  $\{a\}$  ist  
Untermenge

- Menge  $U$  ist Untermenge von  $M$ , wenn jedes Element von  $U$  auch Element von  $M$  ist.
  - Notation:  $U \subset M$  bzw.  $M \supset U$
  - Achtung:  $\underline{\{a\}} \subset \underline{\{a,b,c\}}$ , aber  $a \in \underline{\{a,b,c\}}$
- Potenzmenge von  $M$ :  $Pot(M) = \{m \mid m \subset M\}$ .
  - $(Pot(\{a,b,c\})) \rightarrow 8$  $= \{\underline{\emptyset}, \underline{\{a\}}, \underline{\{b\}}, \underline{\{c\}}, \underline{\{a,b\}}, \underline{\{a,c\}}, \underline{\{b,c\}}, \underline{\{a,b,c\}}\}$  $\quad \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{matrix}$
- Die Anzahl  $|M|$  der Elemente einer Menge  $M$  heißt Mächtigkeit oder Kardinalität von  $M$ .

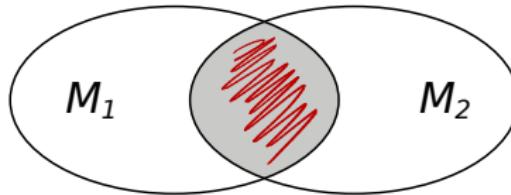
# Operationen auf Mengen 1/2

---

- Mengendifferenz:  $M_1 \setminus M_2 = \{m \mid m \in M_1 \text{ und } m \notin M_2\}$

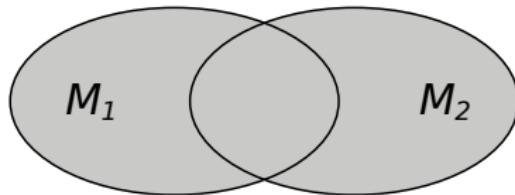


- Mengenschnitt:  $M_1 \cap M_2 = \{m \mid m \in M_1 \text{ und } m \in M_2\}$



# Operationen auf Mengen 2/2

- Mengenvereinigung:  $M_1 \cup M_2 = \{m \mid m \in M_1 \text{ oder } m \in M_2\}$



- Kartesisches Produkt:

$$\underline{M_1 \times M_2} = \{(m_1, m_2) \mid \underline{m_1} \in M_1 \text{ und } \underline{m_2} \in M_2\}$$

- $(m_1, m_2)$  ist ein Tupel, bei dem es, im Gegensatz zu einer Menge  $\{m_1, m_2\}$ , auf die Reihenfolge ankommt!

$$\overleftarrow{\hookrightarrow} = \{m_2, m_1\}$$

- Notation:  $\underline{M^n} = M \times \cdots \times M$  ( $n$  mal).

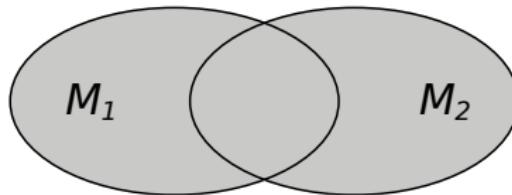
$$= \{ (m_1, \dots, m_n) \mid \begin{array}{l} m_1 \in M \\ m_2 \in M \\ \vdots \\ m_n \in M \end{array} \}$$

$$M^2 = M \times M$$

# Operationen auf Mengen 2/2



- Mengenvereinigung:  $M_1 \cup M_2 = \{m \mid m \in M_1 \text{ oder } m \in M_2\}$



- Kartesisches Produkt:

$$M_1 \times M_2 = \{(m_1, m_2) \mid m_1 \in M_1 \text{ und } m_2 \in M_2\}$$

- $(m_1, m_2)$  ist ein Tupel, bei dem es, im Gegensatz zu einer Menge  $\{m_1, m_2\}$ , auf die Reihenfolge ankommt!
- Notation:  $M^n = M \times \cdots \times M$  ( $n$  mal).

# SMILE – Mengen

$\{ \}$	$\{\emptyset\}$	$\{\{1\}\}$	$\{\{1, 2\}\}$	$\{\{1, 2, 3\}\}$	$\{\{1, 2, 3, 4\}\}$	$\{\{1, 2, 3, 4, 5\}\}$

E1.1  $\notin \cup M_1$

$\cup M_1$

$\cup M_2$

Frage: Welche der folgenden Aussagen sind wahr?

a. Pot({la, le, lu}) = 3 Elemente  $\rightarrow 2^3=8$  Gilt

{ $\emptyset$ , {la}, {le}, {lu}, {la, le}, {la, lu}, {le, lu}, {la, le, lu}}

b. |Pot({1, 2, 3, 4, 5})| = 32 unknown

c.  $\{a, a, b, c\} \setminus \{a, b\}$  = {a, c} gilt nicht!

d.  $|\{a, a, b, c\}|$  = 3 gilt

e.  $(\mathbb{Z} \setminus \mathbb{Q}) \cup \mathbb{Q}$  =  $\mathbb{Z}$  gilt nicht!

$\Rightarrow |\{a, b, c\}|$  = 3

$\{a, a, b, c\} = \{a, b, c\}$   
 $\{a, b, c\} \setminus \{a, b\} = \{c\}$

$2^5$   
||  
32

$\begin{matrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{matrix}$

# Relationen

## Definition

Eine **Relation**  $R$  zwischen den Mengen  $X$  und  $Y$  ist eine Teilmenge von  $\underline{X \times Y}$ .

- Notation: Statt  $(x,y) \in R$  schreibt man  $xRy$ .

- Beispiele:

- Relation  $<$  zwischen  $\mathbb{N}$  und  $\mathbb{N}$ .

$$\leq = \{(0,1), (0,2), \dots, (1,2), (1,3), \dots\}$$

- $R = \{(a,b) \mid a, b \in \mathbb{N}, a+b \text{ ungerade}\}$

$$(2,5) \in R$$

$$R := <$$

$$0 < 1$$

$$< : \mathbb{N} \times \mathbb{N}$$

$$\begin{array}{l} y=1 \\ y=2 \\ y=3 \\ \vdots \end{array}$$

$$x=0$$

# Funktionen

## Definition

Seien  $X$  und  $Y$  Mengen. Eine **Funktion**  $f : X \rightarrow Y$  ist eine Relation zwischen den Mengen  $X$  und  $Y$ , wobei für jedes  $x \in X$  genau ein  $y \in Y$  existiert, so dass  $(x, y) \in f$ .

- $X$  heißt Definitionsbereich,  $Y$  Wertebereich von  $f$ .

- Notation: Statt  $(x, y) \in f$  schreibt man  $y = f(x)$ .

- Beispiele:  $x$      $y$

- **Quadratfunktion**  $f : \mathbb{N} \rightarrow \mathbb{N}, f(x) = x^2$ .

$$f = \{(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), \dots\}$$

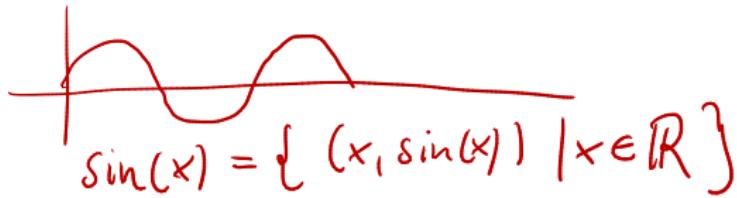
- **Kardinalitätsfunktion**  $f : \text{Pot}(\{a, b, c\}) \rightarrow \mathbb{N}$ .

$$f = \{(\emptyset, 0), (\{a\}, 1), (\{b\}, 1), (\{c\}, 1), (\{a, b\}, 2), (\{a, c\}, 2), (\{b, c\}, 2), (\{a, b, c\}, 3)\}$$

*da {b} aus 2 Elementen besteht  
aus 2 Elementen*

# Beispiele: Relationen, Funktionen

- Jede Funktion ist auch eine Relation. ✓
- Aber es gibt natürlich Relationen, die keine Funktionen sind. (*liegt an dem „genau“ in Def. Folie 12*)
- Beispiel:
  - $\sin^{-1}(x) = \{(\underline{\sin(x)}, x) \mid x \in \mathbb{R}\}$  ist eine Relation, aber keine Funktion!



# Summen und Produkte (Notation)

- Wir schreiben für  $f : \mathbb{N} \rightarrow \mathbb{R}$

$$\sum_{i=m}^n f(i) = \underline{f(m)} + \underline{f(m+1)} + \cdots + \underline{f(n-1)} + \underline{f(n)}$$

$$\prod_{i=m}^n f(i) = f(m) \cdot f(m+1) \cdot \dots \cdot f(n-1) \cdot f(n)$$

- Beispiel:

$$\sum_{i=0}^5 i^2 = \underline{0^2} + \underline{1^2} + \underline{2^2} + \underline{3^2} + \underline{4^2} + \underline{5^2} = 55$$

- Schreibweise mit beliebigen Bedingungen:

$$\sum_{\substack{i,j > 0, i+2j \leq 5 \\ j \in \{1,2,3\}}} (i^2/j) = \begin{matrix} \downarrow & \downarrow & \downarrow & \checkmark & \downarrow & \checkmark & \downarrow & \downarrow \\ (1^2/1) + (1^2/2) + (2^2/1) + (3^2/1) = 14,5 \end{matrix}$$

$\{j \in \{1,2,3\}\}$   
 $j \in \{1,2,3\}$

# Boolesche Algebra ( $\{0, 1\}$ , $\wedge$ , $\vee$ , $\neg$ ) 1/4

## Definition

$x, y, z \in \mathbb{B}$

■  $\mathbb{B} := \{0, 1\}$

■ Konjunktion (UND-Verknüpfung)  $\wedge : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

$$\underline{0 \wedge 0 = 0}, \quad \underline{0 \wedge 1 = 0}, \quad \underline{1 \wedge 0 = 0}, \quad \underline{1 \wedge 1 = 1}$$

$$\begin{array}{c} x \mapsto 0/1 \\ \nearrow + \\ y \mapsto 0/1 \\ z \mapsto 0/1 \end{array}$$

■ Disjunktion (ODER-Verknüpfung)  $\vee : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

$$\underline{0 \vee 0 = 0}, \quad \underline{0 \vee 1 = 1}, \quad \underline{1 \vee 0 = 1}, \quad \underline{1 \vee 1 = 1}$$

■ Negation  $\neg : \mathbb{B} \rightarrow \mathbb{B}$

$$\neg 0 = 1, \quad \neg 1 = 0$$

■ Boolescher Ausdruck

■ Die Elemente aus  $\mathbb{B}$  sind boolesche Ausdrücke.

■ Seien  $A$  und  $B$  boolesche Ausdrücke, dann sind  $(A \wedge B)$ ,  $(A \vee B)$ ,  $(\neg A)$  wieder boolesche Ausdrücke.

$$\begin{array}{c} x \wedge (y + z) \\ \hline x=1 \\ y=0 \\ z=0 \end{array}$$
$$1 \wedge (0 \vee 0) = 1 \wedge 0 = 0$$

## Konventionen

- Man schreibt auch  $\cdot$  statt  $\wedge$  und  $+$  statt  $\vee$ .
- Für  $\neg x$  sind viele Notationen üblich:  $\sim x$ ,  $x'$  oder  $\bar{x}$ .
- Zur Vereinfachung der Notation bei booleschen Ausdrücken vereinbaren wir:  
Negation  $\sim$  bindet stärker als Konjunktion  $\cdot$ , Konjunktion  $\cdot$  /  $\wedge$   
bindet stärker als Disjunktion  $+$  /  $\vee$

# Boolesche Algebra $(\{0,1\}, \underline{\wedge}, \underline{\vee}, \neg)$ 3/4

## Axiome der booleschen Algebra

Kommutativität:  $x + y = y + x$

$$x \cdot y = y \cdot x$$

Assoziativität:  $x + (y + z) = (x + y) + z$

$$x \cdot (y \cdot z) = (x \cdot y) \cdot z$$

Absorption:

*Auslöschung*

$$x + (x \cdot y) = x$$

$$x \cdot (x + y) = x$$

Distributivität:  $x + (y \cdot z) = (x + y) \cdot (x + z)$

$$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$$

Komplement:

$$x + (y \cdot \neg y) = x$$

$$x \cdot (y + \neg y) = x$$

$$\begin{aligned} &x = 1 \rightarrow 1 \vee (1 \cdot y) = 1 \\ &x = 0 \rightarrow 0 \vee (0 \cdot y) = 0 \\ &= 0 \end{aligned}$$

X

immer gleich

$$(0 \wedge \neg 0) = 0 \wedge 1 = 0$$

$$(1 \wedge \neg 1) = 1 \wedge 0 = 0$$



## Axiome der booleschen Algebra

Kommutativitt:  $x + y = y + x$

$$x \cdot y = y \cdot x$$

Assoziativitt:  $x + (y + z) = (x + y) + z$

$$x \cdot (y \cdot z) = (x \cdot y) \cdot z$$

Absorption:  $x + (x \cdot y) = x$

$$x \cdot (x + y) = x$$

Distributivitt:  $x + (y \cdot z) = (x + y) \cdot (x + z)$

$$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$$

Komplement:  $x + (y \cdot \neg y) = x$

$$x \cdot (y + \neg y) = x$$

# SMILE – Boolesche Ausdrücke

$$13 = 42 \not\rightarrow \text{gleichbed. mit } 13 \neq 42$$

Frage: Welche dieser Umformungen Boolescher Ausdrücke sind richtig? Das heißt, die Gleichung ist immer erfüllt für alle möglichen Werte von  $x, y \in \mathbb{B}$ .

- a.  $x + 1 = 1 + x \cdot y \Leftrightarrow 1 = 1$
- b.  $x + \neg x = x \quad x=0 \rightarrow 0+1 \neq 0 \not\rightarrow$
- c.  $(x \cdot (\neg x + (y \cdot \neg y))) = x$

$$\begin{aligned} & (x \cdot (\neg x + (y \cdot \neg y))) \\ & \quad \underbrace{\neg x + y \cdot \neg y}_{=0} \\ & \quad \underbrace{x \cdot 0}_{=0} \\ & \quad x \cdot \neg x = 0 \end{aligned}$$

$$0 = x$$

$x$  kann auch 1 sein  $\Rightarrow 0 = 1$

$\circ$   $+$

- Neben der vorgestellten gibt es weitere boolesche Algebren, in denen diese Axiome gelten.
- Die folgenden Regeln sind aus den Axiomen ableitbar:

## Weitere Regeln für boolesche Algebren

Doppeltes Komplement:

$$\neg(\neg x) = x$$

Idempotenz:

$$x + x = x \cdot x = x$$

De-Morgan-Regel:

$$\neg(x + y) = (\neg x) \cdot (\neg y)$$

$$\neg(x \cdot y) = (\neg x) + (\neg y)$$

Consensus-Regel:

\*  $(x \cdot y) + ((\neg x) \cdot z)$

$$= (x \cdot y) + ((\neg x) \cdot z) + (y \cdot z)$$

$$\rightarrow (x + y) \cdot ((\neg x) + z)$$

$$= (x + y) \cdot ((\neg x) + z) \cdot (y + z)$$

"mit  
wobei  
eine Kette  
aus  
Ketten  
ist"

# Boolesche Funktion

## Definition

Eine **boolesche Funktion**  $f$  in  $n$  Variablen und mit  $m$  Ausgängen ist eine Funktion

$$f : \mathbb{B}^n \rightarrow \mathbb{B}^m (n, m \in \mathbb{N}).$$

*"ausgäbe"*

- Die Menge aller booleschen Funktionen in  $n$  Variablen mit  $m$  Ausgängen ist

$$\mathbb{B}_{n,m} := \{f \mid f : \mathbb{B}^n \rightarrow \mathbb{B}^m\}.$$



- Wir schreiben abkürzend  $\mathbb{B}_n$  statt  $\mathbb{B}_{n,1}$ .
- Ein digitaler Schaltkreis ohne Speicherelemente, mit  $n$  Eingängen und  $m$  Ausgängen realisiert eine solche Funktion! (Details später)

$$f : \mathbb{B}^n \rightarrow \mathbb{B}$$

$$f_{SK} : \mathbb{B}^4 \rightarrow \mathbb{B}$$

# Gerichteter Graph

## Definition

$G = (V, E)$  ist ein **gerichteter Graph**, wenn folgendes gilt:

- $V$  endliche, nichtleere Menge (**Knoten**)  
*↳ vertex*

- $E$  endliche Menge (**Kanten**)  
*↳ edge*

- Abbildungen  $Q : E \rightarrow V$  und  $Z : E \rightarrow V$   
 $Q(e)$  ist Quelle,  $Z(e)$  Ziel einer Kante  $e$

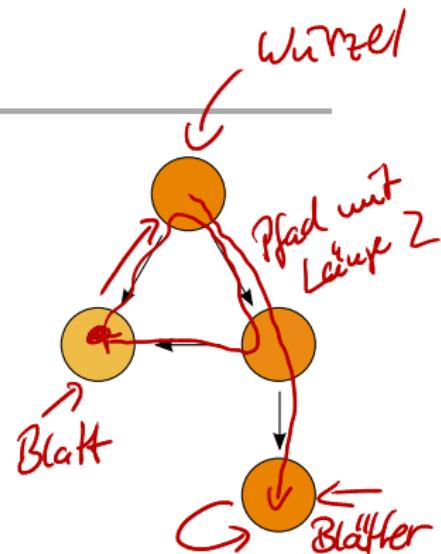


- Abbildungen  $\text{indeg} : V \rightarrow \mathbb{N}$  und  $\text{outdeg} : V \rightarrow \mathbb{N}$   
 $\text{indeg}(v) = |\{e \mid Z(e) = v\}|$  ist der **Eingangsgrad**,  
 $\text{outdeg}(v) = |\{e \mid Q(e) = v\}|$  der **Ausgangsgrad** von  $v$ .

*outdeg = out degree*

# Pfade in gerichteten Graphen

- Ein Knoten mit
  - $\text{indeg}(v) = 0$  heißt Wurzel.
  - $\text{outdeg}(v) = 0$  heißt Blatt.
  - $\text{outdeg}(v) > 0$  heißt innerer Knoten.
- Ein Pfad (der Länge  $k$ ) in  $G$  ist eine Folge von  $k$  Kanten  $e_1, e_2, \dots, e_k$  ( $k \geq 0$ ) mit  $Z(e_i) = Q(e_{i+1})$  für alle  $i$  ( $k - 1 \geq i \geq 1$ )
- Ein Zyklus in  $G$  ist ein Pfad der Länge  $\geq 1$  in  $G$ , bei dem Ziel und Quelle identisch sind ( $G$  heißt azyklisch, falls kein Zyklus in  $G$  existiert).
- Die Graph-Tiefe eines azyklischen Graphen ist definiert als die Länge des längsten Pades in  $G$ .



# Bäume, Binäre Bäume

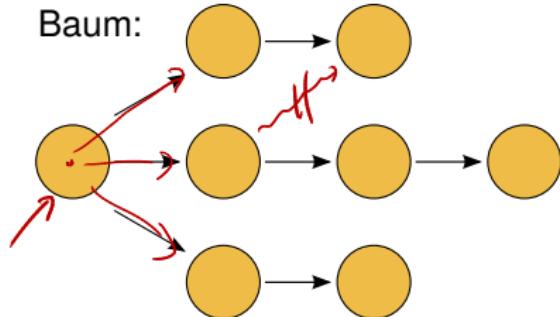
Blau über

## Definition

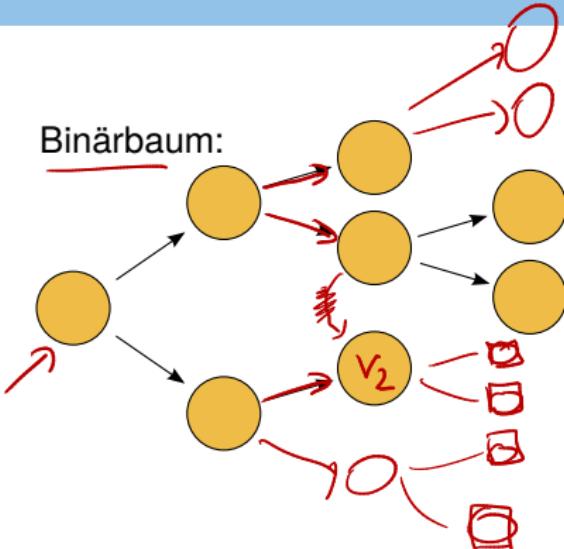
Ein **Baum** ist ein gerichteter, azyklischer Graph mit genau einer Wurzel  $w$  ( $\text{indeg}(w) = 0$ ) und  $\text{indeg}(v) = 1$  für alle andere Knoten  $v$ . Ein Baum heißt **binär** (bzw. Binärbaum), wenn für seine innere Knoten  $v$   $\text{outdeg}(v) \leq 2$  gilt.

## Beispiele:

Baum:



Binärbaum:



# Groß-O-Notation (1/2)

positive, reelle Zahlen (inklusive 0)

- Seien  $f, g : \mathbb{R}_0^+ \rightarrow \mathbb{R}_0^+$ .

Man schreibt  $f(x) \in O(g(x))$ , wenn es  $c \in \mathbb{R}_0^+, x_0 \in \mathbb{R}_0^+$  gibt, so dass  $f(x) \leq c \cdot g(x)$  für alle  $x > x_0$  gilt.

- Beispiel:  $5x + 2 \in O(x^2)$   $\leftarrow$  besser:  $O(x)$

Beweis: Setze  $c = 5, x_0 = 1$

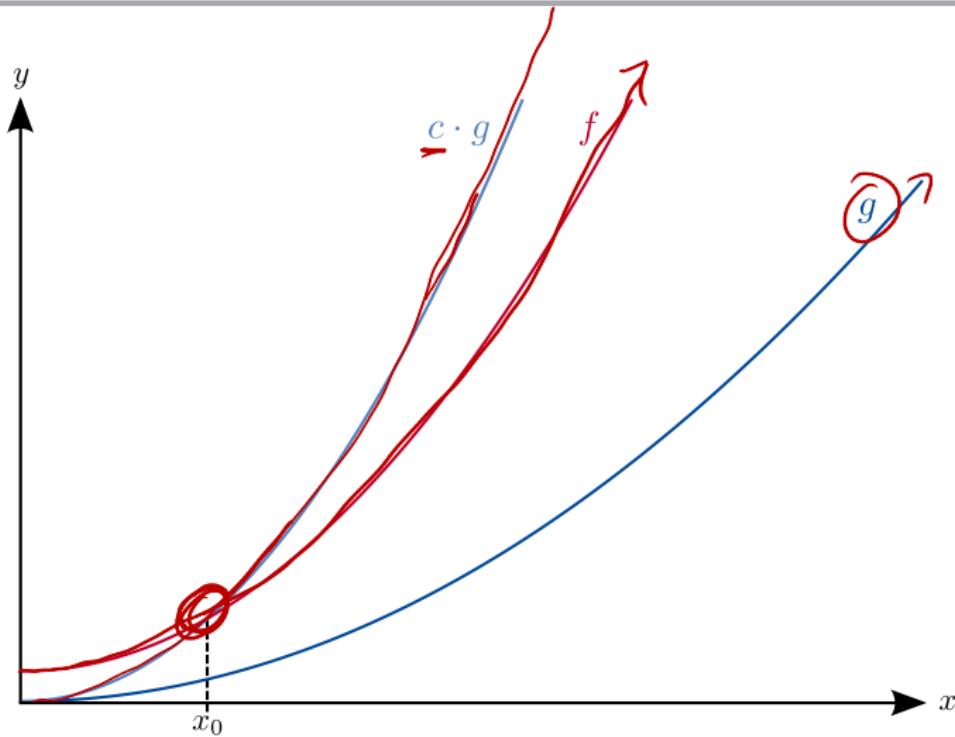
$$5x + 2 \leq 5 \cdot x^2, \text{ für } x > 1.$$

$x=1 : 5 \cdot 1 + 2 = 7 \not\leq 5 \cdot 1^2 = 5$   
 $x=2 : 5 \cdot 2 + 2 = 12 \leq 5 \cdot 2^2 = 20$

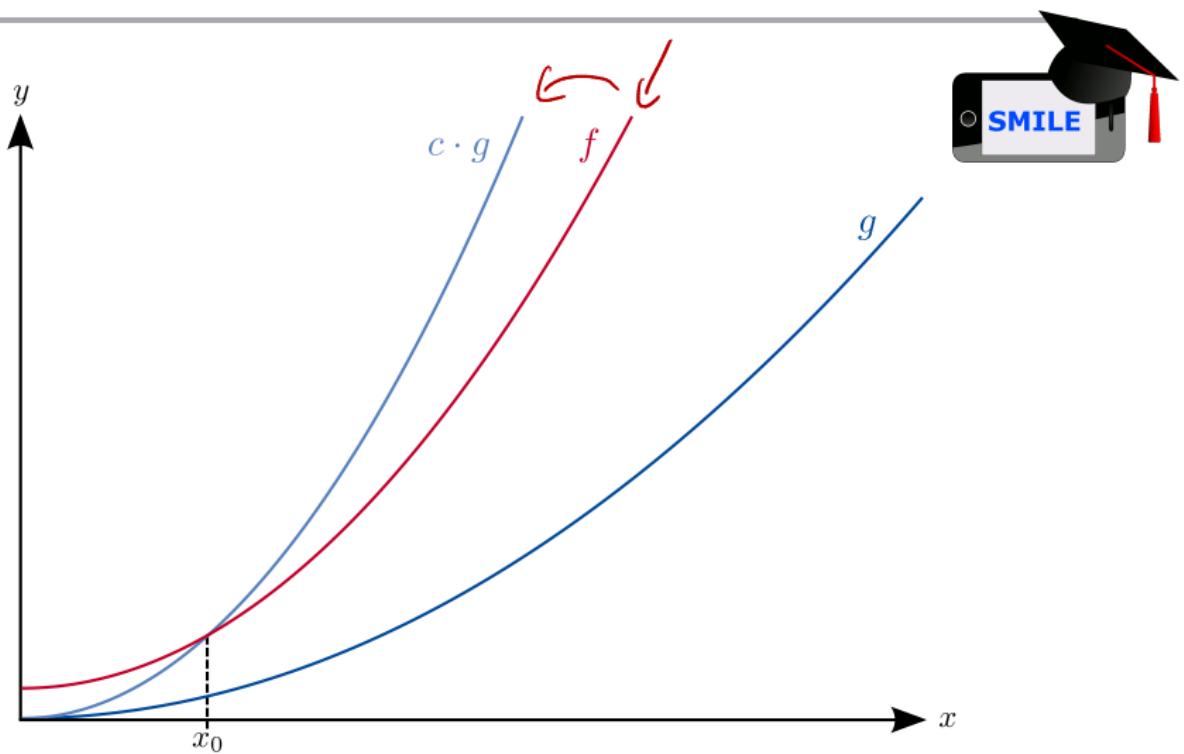
- Groß-O-Notation wird verwendet, um Größe von parametrisierten Objekten (z.B. Graphen), Laufzeit von Algorithmen (Anzahl von Rechenschritten in Abhängigkeit von der Eingabe) usw. **asymptotisch**, d.h. bis auf eine multiplikative Konstante, abzuschätzen.

- Die Notation  $f(x) = O(g(x))$  ist weit verbreitet, aber eigentlich falsch, da  $O(g(x))$  eine Menge ist. So folgt aus  $f(x) = O(g(x))$  und  $h(x) = O(g(x))$  keinesfalls  $f(x) = h(x)!$

# Groß-O-Notation (2/2)

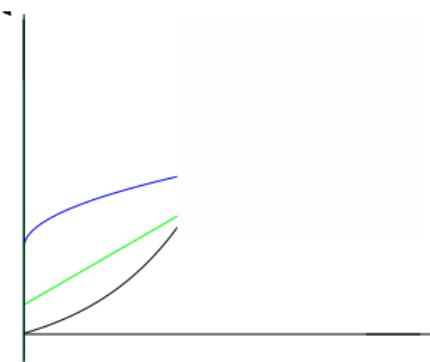


# Groß-O-Notation (2/2)



# SMILE – O-Notation

Gegeben: *blau*      *schwarz*      *grün*  
 $f(x) = \sqrt{x} + 2$ ,     $g(x) = 0,5e^x$ ,     $h(x) = x + 1$ ,  
Welche Aussagen sind dann wahr?



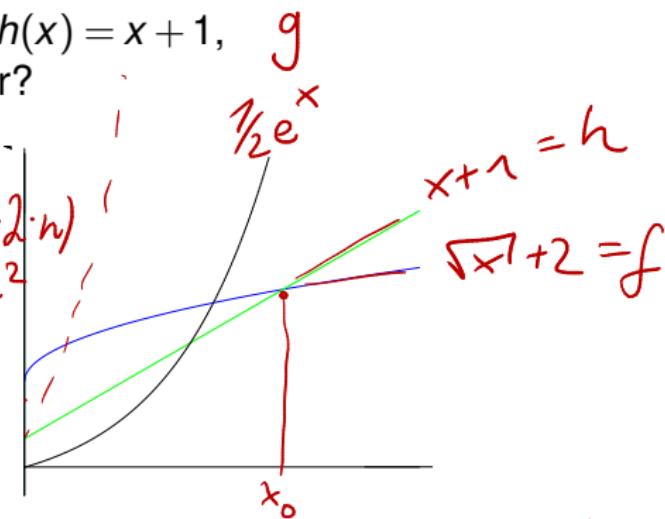
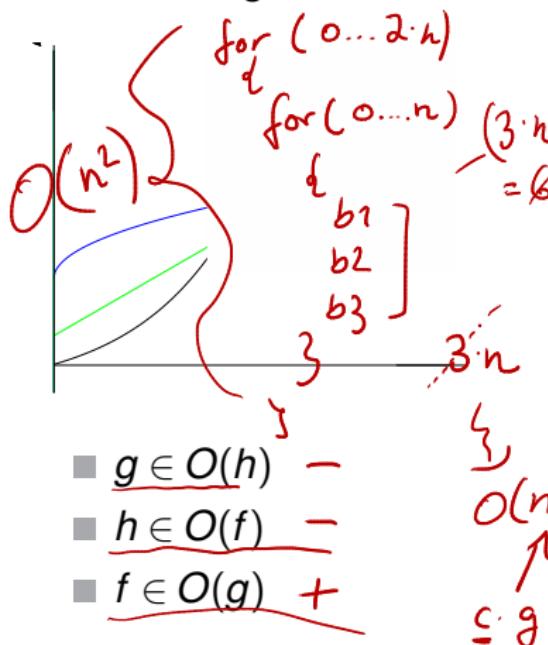
- $g \in O(h)$       NEIN
- $h \in O(f)$       NEIN
- $f \in O(g)$       JA

# SMILE – O-Notation

Gegeben:

$$f(x) = \sqrt{x} + 2, \quad g(x) = 0,5e^x, \quad h(x) = x + 1,$$

Welche Aussagen sind dann wahr?



$\forall x > x_0 : f(x) < h(x)$

- $g \in O(h)$  -
- $h \in O(f)$  -
- $f \in O(g)$  +

$O(n)$

$\subseteq g$

- **Sukzessive Folgerungen bzw. Direkter Beweis**
- **Indirekter Beweis bzw. Beweis durch Widerspruch**
- **Vollständige Induktion**

# Sukzessive Folgerungen

---

Gegeben Aussage  $A$ , es soll Aussage  $B$  bewiesen werden.

- **Sukzessive Folgerungen:**

Aus  $A$  folgt  $C$ , aus  $C$  folgt  $D$ , aus  $D$  folgt  $B$ , also gilt  $B$ .

$$A \rightarrow C \rightarrow D \rightarrow \underline{B}$$

$\overline{A}$   $\square$

q.e.d.

# Beispiel: Sukzessive Folgerungen

„ $f \sim g \sim h$ “

- Gegeben  $f, g, h$ ,  $f(x) \in O(g(x))$ ,  $g(x) \in O(h(x))$ .  
Dann gilt  $f(x) \in O(h(x))$ .



## Beweis:

- 1 Aus  $f(x) \in O(g(x))$  folgt die Existenz von  $c_f, x_{0f} : f(x) \leq c_f \cdot g(x)$  für  $x > x_{0f}$ . Aus  $g(x) \in O(h(x))$  folgt die Existenz von  $c_g, x_{0g} : g(x) \leq c_g \cdot h(x)$  für  $x > x_{0g}$ .
- 2 Man setze  $x_0 = \max\{x_{0f}, x_{0g}\}$ . Dann gilt für  $x > x_0$  sowohl  $f(x) \leq c_f \cdot g(x)$  als auch  $g(x) \leq c_g \cdot h(x)$ .
- 3 Man setze  $c := c_f \cdot c_g$ . Dann gilt für  $x > x_0$ :  
$$f(x) \leq c_f \cdot g(x) \leq c_f(c_g \cdot h(x)) = c \cdot h(x).$$
Dies bedeutet aber gerade  $f(x) \in O(h(x))$

# Indirekter Beweis 1/2

Annahme:  $\neg S$

$$\begin{array}{c} \{ \\ 31 = 42 \\ \hline \end{array} \quad \begin{array}{l} \swarrow \\ \curvearrowleft \\ \curvearrowright \\ \curvearrowleft \end{array} \quad S$$

Es soll Aussage  $S$  bewiesen werden.

- **Indirekter Beweis:** Man nimmt an,  $\neg S$  (also die Umkehrung von  $S$ ) würde gelten. Daraus leitet man einen Widerspruch her (z.B. "es gilt  $C$  und  $\neg C$ ", " $\underline{31 = 42}$ ", ...).
- Da der Widerspruch schrittweise aus  $\neg S$  logisch hergeleitet wurde, kann  $\neg S$  nicht gelten und somit muss  $S$  gelten.

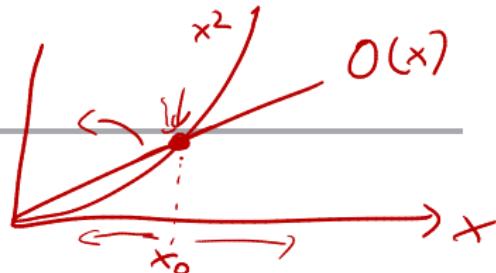
## Indirekter Beweis 2/2

aus  $A$  folgt  $B$

$$\underline{A=1} \rightarrow \underline{B=1}$$

- Betrachte den Spezialfall  $\underline{\underline{S}} \approx \underline{\underline{A \Rightarrow B}}$ .
- Dann ist  $\neg \underline{\underline{S}} \approx \underline{\underline{A \wedge \neg B}}$ . Man nimmt also an, dass  $\underline{A}$  gilt, aber  $\neg \underline{B}$ .
- Ergibt sich aus der Annahme ein Widerspruch, dann muss aus der Gültigkeit von  $\underline{A}$  die Gültigkeit von  $\underline{B}$  folgen.
- Ergibt sich der Widerspruch speziell durch Herleitung von  $\neg \underline{A}$  aus  $\neg \underline{B}$ , dann reduziert sich der Widerspruchsbeweis auf den Spezialfall **Beweis der "Kontraposition"**  $\neg \underline{B} \Rightarrow \neg \underline{A}$ .
- $\underline{A \Rightarrow B}$  und  $\neg \underline{B} \Rightarrow \neg \underline{A}$  sind logisch äquivalent.  $\underline{B=0} \rightarrow \underline{A=0}$
- Implizit setzt man immer die Gültigkeit sämtlicher Axiome voraus. Sei  $\underline{Ax}$  die Aussage "Sämtliche Axiome gelten".
- Dann ist  $\underline{S'} = (\underline{A \wedge Ax}) \Rightarrow \underline{B}$  zu beweisen.
- Annahme ist dann also:  $\neg \underline{S'} = \underline{A \wedge Ax \wedge \neg B}$  gilt.

# Beispiel: Indirekter Beweis



- Zu zeigen:  $x^2 \notin O(x)$

Beweis:

- Wir nehmen an, dass  $x^2 \in O(x)$  wäre. Dann gibt es  $c$  und  $x_0$ , so dass für  $x > x_0$  gilt:

$$x^2 \leq c \cdot x \quad (1)$$

- Nun suchen wir ein  $x_1$ , für das  $x_1^2 = c \cdot x_1$ . Dies ist für  $x_1 = c$  der Fall.

- Für alle  $x > x_1 = c$  ist  $x^2 > c \cdot x$ . Man wähle ein  $x_2 > \max\{x_0, x_1\}$ . Dann gilt auch für  $x_2$ :

$$x_2^2 > c \cdot x_2 \quad (2)$$

- Andererseits muss für  $x_2$  auch (1) gelten. Widerspruch! Somit kann die Annahme nicht stimmen.

# Vollständige Induktion

- Die vollständige Induktion ist eine Beweismethode für Aussagen, die für alle natürlichen Zahlen  $n$  gelten sollen.
- Zuerst wird die Aussage für den Basisfall  $n = 0$  beweisen (manchmal auch  $n = 1$  oder höher).
- Dann wird der Induktionsschritt durchgeführt:  
Unter der Annahme, dass die Aussage für  $n$  gilt (Induktionsvoraussetzung) wird bewiesen, dass die Aussage auch für  $n + 1$  gilt.
- Daraus folgt die Gültigkeit der Aussage für alle natürlichen Zahlen.

$n = 0$ ,  $n$   
 $1$  -  
 $2$  -  
 $3$  -  $(n+1)$

# Vollständige Induktion: Beispiel (1/2)

## ■ Behauptung:

$$\sum_{k=1}^n \frac{1}{k(k+1)} = \frac{n}{n+1}$$
 gilt für alle  $n \in \mathbb{N}$ .

## ■ Induktionsanfang:

Zeige die Behauptung für  $n = 1$ .

$$\sum_{k=1}^1 \frac{1}{k(k+1)} = \frac{1}{1(1+1)} = \frac{1}{2} = \frac{1}{1+1}$$

# Vollständige Induktion: Beispiel (2/2)

## ■ **Induktionsvoraussetzung (IV):**

Nehme an, die Behauptung gilt für *ein*  $n \in \mathbb{N}$ .

Also: Es gibt ein  $n$  für das gilt:  $\sum_{k=1}^n \frac{1}{k(k+1)} = \frac{n}{n+1}$

## ■ **Induktionsschritt:**

Zeige die Behauptung für  $n+1$ .

$$\begin{aligned}\sum_{k=1}^{n+1} \frac{1}{k(k+1)} &= \left( \sum_{k=1}^n \frac{1}{k(k+1)} \right) + \frac{1}{(n+1)(n+2)} \stackrel{\text{IV}}{=} \frac{n}{n+1} + \frac{1}{(n+1)(n+2)} \\ &= \frac{n(n+2)+1}{(n+1)(n+2)} = \frac{n^2+2n+1}{(n+1)(n+2)} = \frac{(n+1)^2}{(n+1)(n+2)} = \frac{(n+1)}{(n+2)} = \frac{(n+1)}{(n+1)+1} \quad \square\end{aligned}$$

*(n+1)-Summenglied*

qed.

# Kapitel 1 – Grundlagen

1. Mathematische Grundlagen
2. **Beispielrechner ReTI**

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer  
Professur für Rechnerarchitektur  
WS 2016/17

- Ursprünglich eingeführt in [Keller, Paul] unter dem Namen ReSa.
- Hier wird ReTI zunächst abstrakt eingeführt.
  - Alle Speicher bestehen aus unendlich vielen Speicherzellen, die beliebig große ganze Zahlen aufnehmen können.
- Später wird die tatsächliche Implementierung von ReTI unter realistischen Annahmen thematisiert.

### ■ Zwei unendlich große Speicher

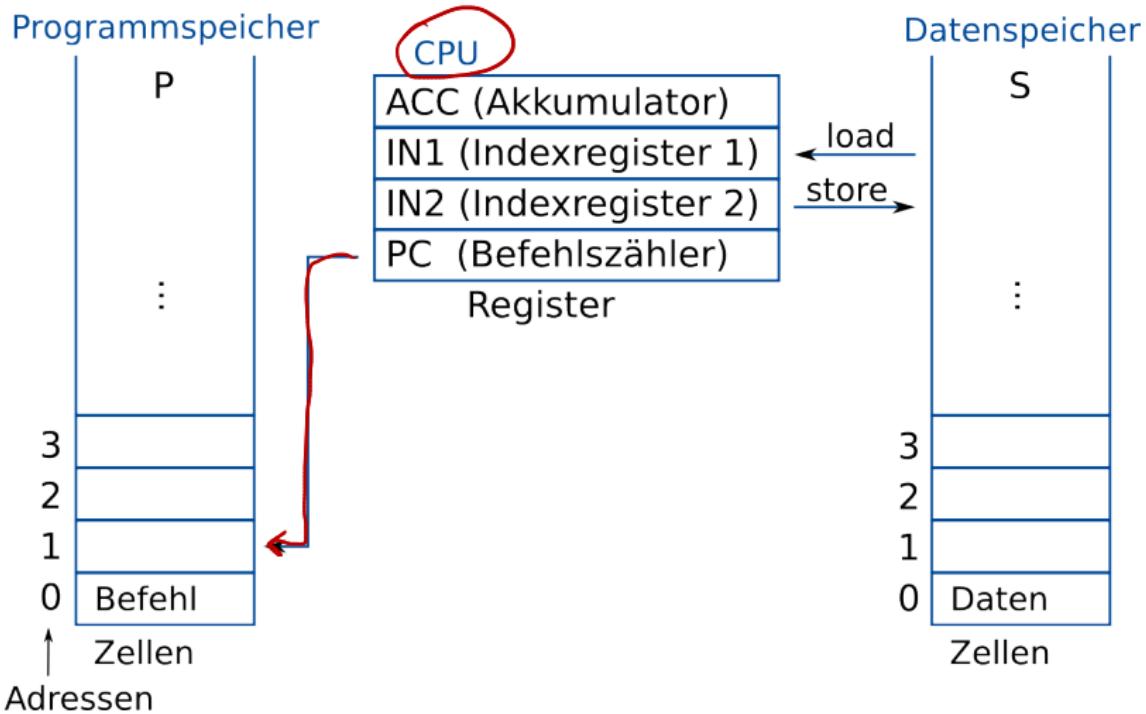
- Datenspeicher  $S$  für Daten (beliebig große Zahlen).
- $S(i)$  = Inhalt von Zelle  $i$  des Datenspeichers,  $i \in N$  Adresse.
- Programmspeicher  $P$  für Maschinenbefehle.
- Lade-/Speicher-, Rechen-, Sprungbefehle - siehe später.
- $P(i)$  = Inhalt von Zelle  $i$  des Programmspeichers.

von Neumann:  
gemeinsamer  
Prg.-/Datensp.

### ■ Zentraleinheit CPU (Central Processing Unit)

- Vier für Benutzer sichtbare Register.
- PC = Befehlszähler (Program Counter).
- ACC = Akkumulator.
- IN1, IN2 = Indexregister 1 und 2.

# Aufbau von ReTI



# Programmablauf

---

- Programme bzw. Daten stehen beim Start der Maschine in  $P$  bzw.  $S$ .
- Programm beginnt bei Zelle 0 von  $P$ .
- Inhalt von  $P$  wird nicht geändert.
- Maschine arbeitet in Schritten  $t = \underline{1, 2, \dots}$ 
  - In jedem Schritt  $t$ :
    - Ausführung eines Befehls:  $P(PC)$  wird als Befehl interpretiert und in Schritt  $t$  ausgeführt.
    - $PC$  erhält neuen Wert (abhängig von Befehl).
- Bei Programmstart ist  $PC = 0$ .

*festes Programm daher  
brauchen auch die  
Indexregister*

*aus der Zelle mit Index  
"PC"*

# ReTI-Befehle und ihre Wirkung

---

- Load/Store: Laden von Werten aus dem Datenspeicher S bzw. Schreiben von Werten in S.

- Compute: Berechnungen (hier zunächst Addition und Subtraktion).

- Mit Werten im Datenspeicher S.
  - Mit Absolutwerten (Immediate).

$$\begin{aligned} \text{ACC} &:= \text{ACC} + S(i) \\ \text{ACC} &:= \text{ACC} + i \end{aligned}$$

- Indexregister: Indirekte Speicheradressierung (siehe unten).

- Sprungbefehle: Bedingte und unbedingte Sprünge.

*- if - Anweisungen  
- Schleifen realisieren zu können.*

# Load/Store

---

*S*

Transport von Daten zwischen *ACC* und *Datenspeicher*.

- **LOAD *i*:**

Lädt *Inhalt S(i)* von Speicherzelle *i* in Akkumulator *ACC* und erhöht *PC* um 1.

- **STORE *i*:**

Speichert den *Inhalt von ACC* in *S(i)* und erhöht *PC* um 1.

# Load/Store: Übersicht

---

Befehl	Wirkung	
<u>LOAD <math>i</math></u>	$ACC := S(i)$	$PC := PC + 1$
STORE $i$	$S(i) := ACC$	$PC := PC + 1$

# Beispielprogramm

Ein Programm, das Inhalte von Speicherzelle  $S(0)$  ( $= \underline{x}$ ) und  $S(1)$  ( $= \underline{y}$ ) vertauscht.

0	LOAD 0;	$\underline{ACC := S(0) = x}$
1	STORE 2;	$S(2) := \underline{ACC} = x$
2	<u>LOAD 1;</u>	$ACC := S(1) = y$
3	<u>STORE 0;</u>	$S(0) := ACC = y$
4	LOAD 2;	$ACC := \underline{S(2)} = x$
5	STORE 1;	$S(1) := \underline{ACC} = x$

$$\begin{aligned}S(0) &= \cancel{x} \quad y^{(3)} \\S(1) &= y \\S(2) &= x^{(1)} \quad \cancel{y^{(2)}} \\ACC &= x^{(0)} \quad y^{(2)}\end{aligned}$$

# Compute-Befehle

Verknüpfe den Inhalt von  $ACC$  mit  $S(i)$  oder mit einer Konstante und speichere das Ergebnis in  $ACC$  ab.

- $ADD, SUB$  = Compute memory-Befehle
- $ADDI, SUBI$  = Compute immediate-Befehle
- Beides zusammen ergibt die **Compute-Befehle**.

Bei Compute memory: Interpretiere Parameter  $i$  direkt als Speicheradresse.

Befehl	Wirkung
$ADD \ i$	$ACC := ACC + S(i) \quad PC := PC + 1$
$SUB \ i$	$ACC := ACC - S(i) \quad PC := PC + 1$

# Immediate-Befehle

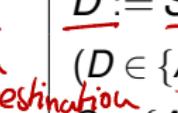
---

Interpretiere Parameter  $i$  direkt als Konstante.

Befehl	Wirkung	
<u>LOADI</u> $i$	$ACC := i$	$PC := PC + 1$
<u>ADDI</u> $i$	$ACC := ACC + i$	$PC := PC + 1$
<u>SUBI</u> $i$	$ACC := ACC - i$	$PC := PC + 1$

- Anmerkung: ADDI und SUBI sind Compute Befehle.  
LOADI ist den Load-/Store-Befehlen zuzuordnen.

# Indexregister-Befehle

Befehl	Wirkung	
LOADINj <i>i</i>	$ACC := \underline{S}(\underline{INj} + i)$ $(j \in \{1, 2\})$	$PC := PC + 1$
STOREINj <i>i</i>	$S(\underline{INj} + i) := \underline{ACC}$ $(j \in \{1, 2\})$	$PC := PC + 1$
MOVE S D <i>Source</i>  <i>Destination</i> 	$\underline{D} := \underline{S}$ $(D \in \{\underline{ACC}, \underline{IN1}, \underline{IN2}\},$ $S \in \{\underline{ACC}, \underline{IN1}, \underline{IN2}, \underline{PC}\})$	$\underline{PC} := PC + 1$
MOVE S PC	$PC := S$ $(S \in \{ACC, IN1, IN2\})$	

→ Unterscheidung, ob  $D = PC$  ist oder nicht!

# Beispielprogramm für Indexregister-Befehle

$S(0) = x, S(1) = y$   
Kopiere  $y$  in Zelle  $S(x)$ :

0	LOAD 0;	$\underline{ACC} := \underline{S(0)} = x$
1	MOVE ACC IN1;	$\underline{IN1} := \underline{ACC} = x$
2	LOAD <u>1</u> ;	$\underline{ACC} := \underline{S(1)} = y$
3	STOREIN1 0;	$S(x) = S(\underline{IN1} + 0) := \underline{ACC} = y$

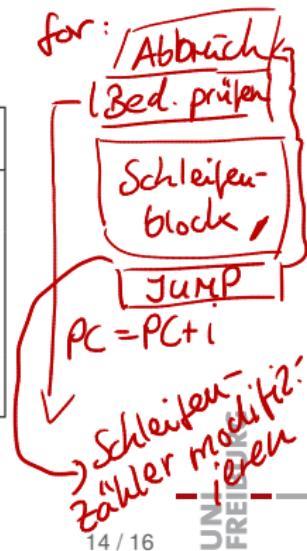
$\downarrow$   
 $S(x) = y$

# Sprung-Befehle

Manipulation des Befehlszählers.

- JUMP für *unbedingte Sprünge*,
- JUMP<sub>c</sub> mit  $c \in \{<, =, >, \leq, \neq, \geq\}$  für *bedingte Sprünge*.
- Mit bedingten Sprüngen kann man *Programmschleifen* und *bedingte Anweisungen* realisieren!

Befehl	Wirkung
<i>unbedingte</i> JUMP $i$	$PC := PC + i \quad (i \in \mathbb{Z})$
<i>bedingte</i> JUMP <sub>c</sub> $i$	$PC := \begin{cases} PC + i, & \text{falls } ACC \underset{c}{=} 0 \\ PC + 1, & \text{sonst} \end{cases}$ $(i \in \mathbb{Z}, c \in \{<, =, >, \leq, \neq, \geq\})$



# Beispielprogramm

$$S(0) = \underline{x}; S(1) = \underline{y}, y \geq 0$$

$y = 3$   
 $\downarrow$   
 $y = 2$   
 $\downarrow$   
 $x \times x$

$y = 1$   
 $\downarrow$   
 $x \times x \times x$

$y = 0$   
 $\downarrow$   
 $x \times x \times x \times x$

SMI ←  $y = -1$  analog  
 $\downarrow$   
 DONE

$\Rightarrow$  Multiplikation von  
 $x \cdot y = S(2)$

0	LOAD I 0;	ACC := 0
1	STORE 2;	$S(2) := 0 \rightarrow$ Initialisierung
2	LOAD 1;	$ACC := S(1) \rightsquigarrow$ Schleifenzähler
(3)	SUBI 1;	$ACC := ACC - 1$ ( $y = y - 1$ ) $S(2) = x + \dots + x$
4	STORE 1;	$S(1) := ACC$
5	JUMP $\leq 5$ ;	$PC := PC + 5$ , falls $ACC < 0$
6	LOAD 2;	$ACC := S(2)$
7	ADD 0;	$ACC := ACC + S(0)$ $ACC = ACC + x$
8	STORE 2;	$S(2) := ACC$
9	JUMP -7;	$PC := PC - 7$

# Zusammenfassung

---

- Mathematik erlaubt es uns, reale Zusammenhänge formal zu fassen und allgemeingültige Folgerungen aus ihnen herzuleiten.
- Rechner ReTI wird uns im weiteren Verlauf der Vorlesung als Illustrator und Anwendungsbeispiel für die vorgestellten Konzepte dienen.

# Motivation

TTL :



- Ein Rechner speichert, verarbeitet und produziert Informationen.
- Alle Ergebnisse müssen als Funktion der Anfangswerte exakt reproduzierbar sein.
  - Informationsspeicherung und Verarbeitung müssen exakt sein.  
*schön*      *noise*
- Probleme: Noise, Crosstalk, Abschwächung
- Es gibt keine exakte Datenübertragung oder Datenspeicherung.
- Ziel: Quantisierung der Informationsspeicherung mit Signal groß gegenüber maximaler Störung
- Binär-Codierung (nur zwei Zustände) ist die einfachste (und sicherste) Signal-Quantisierung.
- **BIT** (0, 1) als grundlegende Informationseinheit



# Motivation

---

- Ein Rechner kann üblicherweise
    - Zeichen verarbeiten (Textverarbeitung)
    - mit Zahlen rechnen
    - Bilder, Audio- und Videoinformationen verarbeiten und darstellen ...
  - Ein Algorithmus kann zwar prinzipiell mit abstrakten Objekten verschiedener Art operieren, aber diese müssen im Rechner letztendlich als Folgen von Bits repräsentiert werden.
- **Kodierung!**

# Kapitel 2.1 - Kodierung von Zeichen

---

- Wie werden im Rechner Zeichen dargestellt ?
- Codes fester Länge
- “Längenoptimale Kodierungen” von Zeichen:  
Häufigkeitscodes (Bsp.: Huffman-Code)

# Kapitel 2 – Kodierung

1. **Kodierung von Zeichen**
2. Kodierung von Zahlen
3. Anwendung: ReTI ↗ *Kodierung von Befehlen*

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur  
WS 2016/17

# Alphabete und Wörter

" $a$ ", " $b$ ", ...

## Definition

Eine nichtleere Menge  $A = \{a_1, \dots, a_m\}$  heißt (endliches) **Alphabet der Größe  $m$** .

$a_1, \dots, a_m$  heißen **Zeichen** des Alphabets.

- $A^* = \{w \mid w = b_1 \dots b_n \text{ mit } n \in \mathbb{N}, \forall i \text{ mit } 1 \leq i \leq n : b_i \in A\}$  ist die **Menge aller Wörter** über dem Alphabet  $A$ .
- $|b_1 \dots b_n| := n$  heißt **Länge** des Wortes  $b_1 \dots b_n$ .
- Das Wort der Länge 0 wird mit  $\varepsilon$  bezeichnet.

Beispiel:

Sei  $A = \{a, b, c, d\}$ .

Dann ist  $bcada$  ein Wort der Länge 5 über  $A$ .

$\overline{\varepsilon} A^*$

$$a_1 \in A \Rightarrow c(a_1) = 01001$$

Sei  $A = \{a_1, \dots, a_m\}$  ein endliches Alphabet der Größe  $m$ .

*Codewörter können unterschiedliche Länge haben,  $c: A \rightarrow \{0,1\}^n$  mit  $n! = 1, 2, 3, \dots$*

- Eine Abbildung  $c: A \rightarrow \{0,1\}^*$  oder  $c: A \rightarrow \{0,1\}^n$  heißt Code, falls  $c$  injektiv ist.
- Die Menge  $c(A) := \{w \in \{0,1\}^* \mid \exists a \in A : c(a) = w\}$  heißt Menge der Codewörter.
- Ein Code  $c: A \rightarrow \{0,1\}^n$  heißt Code fester Länge.
- Für einen Code  $c: A \rightarrow \{0,1\}^n$  fester Länge gilt:  $n \geq \lceil \log_2 m \rceil$ .
  - Ist  $n = \lceil \log_2 m \rceil + r$  mit  $r > 0$ , so können die r zusätzlichen Bits zum Test auf Übertragungsfehler verwendet werden (siehe Kap. 6).

*) Links eindeutig, d.h. für jedes Codewort gibt es genau ein  $a_i \in A$ .*

# Codes fester Länge

---

- Die Kodierung eines jeden Zeichens besteht aus *n Bits*.
  - ASCII (American Standard Code for Information Interchange): 7 Bits (es gibt Erweiterungen mit 8 Bits)
  - EBCDIC: 8 Bits
  - Unicode: 16 Bits
  - Vgl. „Rechnerarchitektur“

Diese Kodierungen sind recht einfach zu behandeln. Unter Umständen wird für sie aber mehr Speicherplatz gebraucht als unbedingt nötig.

# Beispiel: ASCII-Tabelle

erste 3 Bits							
0	0	0	0	1	1	1	1
0	0	1	1	0	0	1	1
0	1	0	1	0	1	0	1
0000	nul	dle	!	@	P	-	p
0001	soh	dc1	"	A	Q	a	q
0010	sfx	dc2	#	B	R	b	r
0011	etx	dc3	\$	C	S	c	s
0100	eot	dc4	%	D	T	d	t
0101	enq	nak	&	E	U	e	u
0110	ack	syn	'	F	V	f	v
0111	bel	etb	(	G	W	g	w
1000	bs	can	)	H	X	h	x
1001	ht	em	*	I	Y	i	y
1010	lf	sub	:	J	Z	j	z
1011	vt	esc	,	K	[	k	{
1100	ff	fs	<	L	\	l	
1101	cr	qs	=	M	]	m	}
1110	so	rs	.	N	^	n	*
1111	si	us	/	O	-	o	del

**letzte 4 Bits**

**Carriage return**

**Steuerzeichen**

**Schriftzeichen**

*Codierung dieses 10 Zeichen je einmal zu 3 Bit*

# Häufigkeitsabhängige Codes

---

- **Ziel:** Reduktion der Länge einer Nachricht durch Wahl verschieden langer Codewörter für die verschiedenen Zeichen eines Alphabets (also kein Code fester Länge!)

- **Idee:** Häufiges Zeichen → kurzer Code  
Seltenes Zeichen → langer Code

- **Voraussetzungen:**

- Häufigkeitsverteilung ist bekannt → statische Kompression

- Häufigkeitsverteilung ist nicht bekannt → dynamische Kompression
  - 1) Stream alle Zeichen, zähle die Häufigkeiten

- 2) kodieren

- 3) v) Verteilung dem "Dekodierer" mitschicken.

# Huffman-Code

---

- Der **Huffman-Code** ist der bekannteste Häufigkeitsabhängige Code.
- Kommt als Teilschritt z.B. in MP3 oder JPEG vor.

- Beispiel:

Zeichen	a	b	c	d	e	f	g	h	i	j
Häufigkeit [%]	20	25	15	8	7	6	5	5	5	4

Baue binären Baum, indem die beiden kleinsten Häufigkeiten jeweils zu einem neuen Knoten addiert werden.

# Huffman-Code

- Der **Huffman-Code** ist der bekannteste Häufigkeitsabhängige Code.
- Kommt als Teilschritt z.B. in MP3 oder JPEG vor.

- Beispiel:

Zeichen	a	b	c	d	e	f	g	h	i	j
Häufigkeit [%]	20	25	15	8	7	6	5	5	5	4



Baue binären Baum, indem die beiden kleinsten Häufigkeiten jeweils zu einem neuen Knoten addiert werden.

# Huffman-Code

- Der **Huffman-Code** ist der bekannteste Häufigkeitsabhängige Code.
- Kommt als Teilschritt z.B. in MP3 oder JPEG vor.

- Beispiel:

Zeichen	a	b	c	d	e	f	g	h	i	j
Häufigkeit [%]	20	25	15	8	7	6	5	5	5	4

The diagram shows a step in the Huffman coding process. Two nodes with frequency 5 (highlighted with red circles) are merged into a new node with frequency 10 (also highlighted with a red circle). Red lines connect the frequencies 5 and 10.

Baue binären Baum, indem die beiden kleinsten Häufigkeiten jeweils zu einem neuen Knoten addiert werden.

# Huffman-Code

- Der **Huffman-Code** ist der bekannteste häufigkeitsabhängige Code.
- Kommt als Teilschritt z.B. in MP3 oder JPEG vor.

- Beispiel:

Zeichen	a	b	c	d	e	f	g	h	i	j
Häufigkeit [%]	20	25	15	8	7	6	5	5	5	4

```
graph TD; Root --- Node1[13]; Root --- Node2[10]; Node1 --- Node3[9]; Node2 --- Node4[9];
```

Baue binären Baum, indem die beiden kleinsten Häufigkeiten jeweils zu einem neuen Knoten addiert werden.

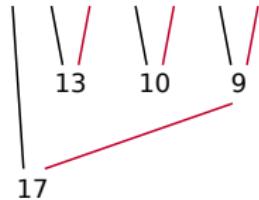
# Huffman-Code

- Der **Huffman-Code** ist der bekannteste häufigkeitsabhängige Code.
- Kommt als Teilschritt z.B. in MP3 oder JPEG vor.

- Beispiel:

Zeichen	a	b	c	d	e	f	g	h	i	j
Häufigkeit [%]	20	25	15	8	7	6	5	5	5	4

Baue binären Baum, indem die beiden kleinsten Häufigkeiten jeweils zu einem neuen Knoten addiert werden.

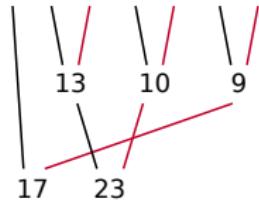


# Huffman-Code

- Der **Huffman-Code** ist der bekannteste häufigkeitsabhängige Code.
- Kommt als Teilschritt z.B. in MP3 oder JPEG vor.
- Beispiel:

Zeichen	a	b	c	d	e	f	g	h	i	j
Häufigkeit [%]	20	25	15	8	7	6	5	5	5	4

Baue binären Baum, indem die beiden kleinsten Häufigkeiten jeweils zu einem neuen Knoten addiert werden.



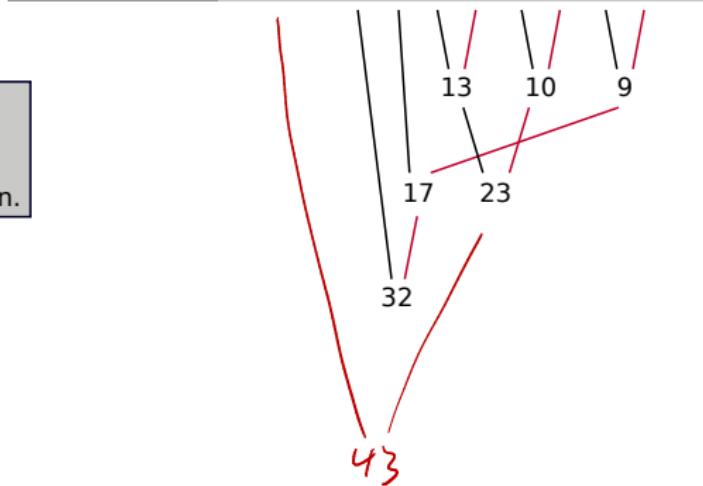
# Huffman-Code

- Der **Huffman-Code** ist der bekannteste häufigkeitsabhängige Code.
- Kommt als Teilschritt z.B. in MP3 oder JPEG vor.

- Beispiel:

Zeichen	a	b	c	d	e	f	g	h	i	j
Häufigkeit [%]	20	25	15	8	7	6	5	5	5	4

Baue binären Baum, indem die beiden kleinsten Häufigkeiten jeweils zu einem neuen Knoten addiert werden.



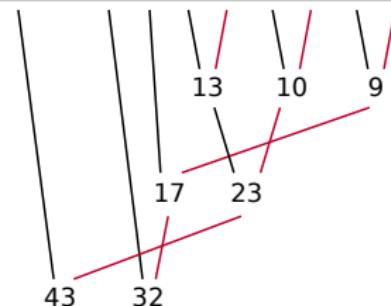
# Huffman-Code

- Der **Huffman-Code** ist der bekannteste häufigkeitsabhängige Code.
- Kommt als Teilschritt z.B. in MP3 oder JPEG vor.

- Beispiel:

Zeichen	a	b	c	d	e	f	g	h	i	j
Häufigkeit [%]	20	25	15	8	7	6	5	5	5	4

Baue binären Baum, indem die beiden kleinsten Häufigkeiten jeweils zu einem neuen Knoten addiert werden.

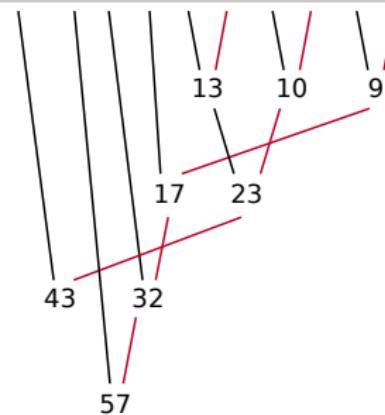


# Huffman-Code

- Der **Huffman-Code** ist der bekannteste häufigkeitsabhängige Code.
- Kommt als Teilschritt z.B. in MP3 oder JPEG vor.
- Beispiel:

Zeichen	a	b	c	d	e	f	g	h	i	j
Häufigkeit [%]	20	25	15	8	7	6	5	5	5	4

Baue binären Baum, indem die beiden kleinsten Häufigkeiten jeweils zu einem neuen Knoten addiert werden.

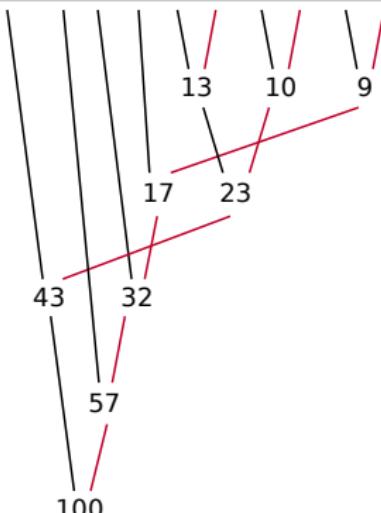


# Huffman-Code

- Der **Huffman-Code** ist der bekannteste häufigkeitsabhängige Code.
- Kommt als Teilschritt z.B. in MP3 oder JPEG vor.
- Beispiel:

Zeichen	a	b	c	d	e	f	g	h	i	j
Häufigkeit [%]	20	25	15	8	7	6	5	5	5	4

Baue binären Baum, indem die beiden kleinsten Häufigkeiten jeweils zu einem neuen Knoten addiert werden.



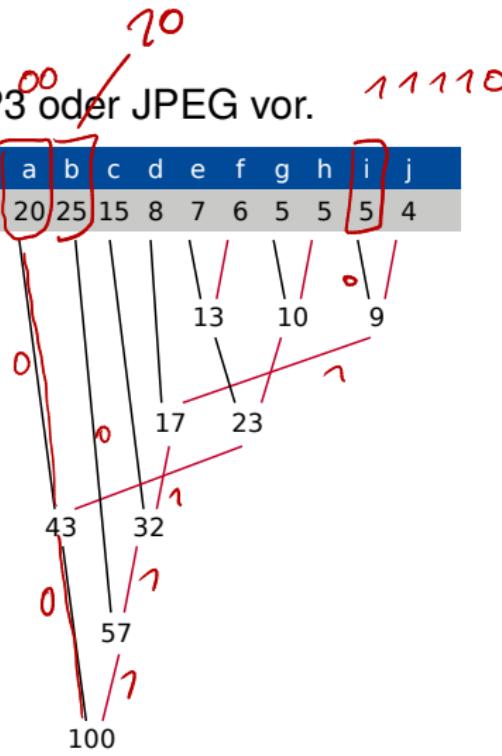
# Huffman-Code

- Der **Huffman-Code** ist der bekannteste häufigkeitsabhängige Code.
- Kommt als Teilschritt z.B. in MP3 oder JPEG vor.

- Beispiel:

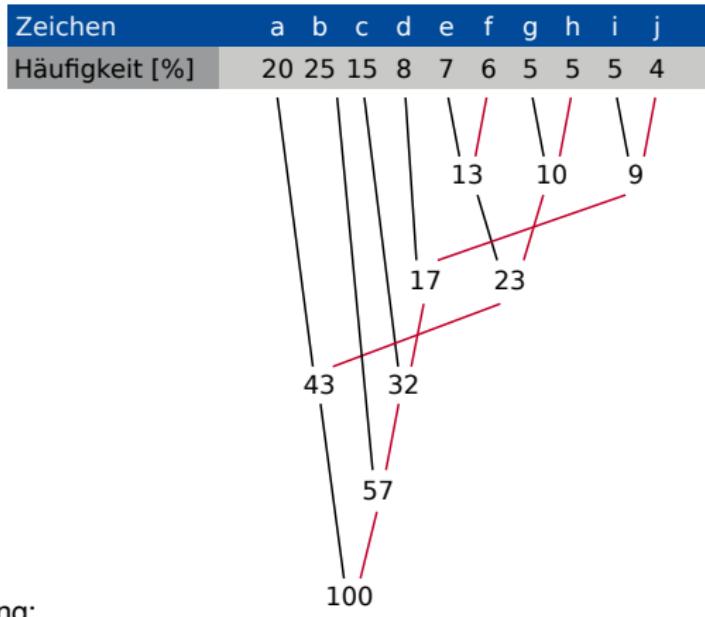
Zeichen	a	b	c	d	e	f	g	h	i	j
Häufigkeit [%]	20	25	15	8	7	6	5	5	5	4

Baue binären Baum, indem die beiden kleinsten Häufigkeiten jeweils zu einem neuen Knoten addiert werden.



Markiere nun die linken Kanten mit 0 und die rechten Kanten mit 1, fertig ist der Huffman-Code!

# Erzeugte Huffman-Kodierung



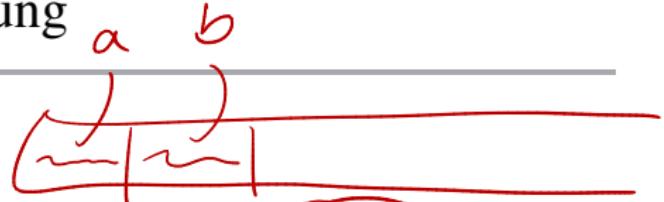
Erzeugte Kodierung:

a	b	c	d	e	f	g	h	i	j
00	10	110	1110	0100	0101	0110	0111	11110	11111

# Huffman-Code: Dekodierung

Erzeugte Kodierung:

a	b	c	d	e	f	g	h	i	j
00	10	110	1110	0100	0101	0110	0111	1110	11111



Problem :

$$\begin{aligned}a &= 01 \\b &= 0111\end{aligned}$$

$\alpha$   
0  
100

Bitstrom :

011.....

ist  
das 'a' mit 1  
oder ist das 'b'?

1  
Schnitt

# Präfixcodes

## Definition

Sei  $A$  ein Alphabet der Größe  $m$ .

"01" ist Präfix von "011"

- $a_1 \dots a_p \in A^*$  heißt **Präfix** von  $b_1 \dots b_l \in A^*$ , falls  $p \leq l$  und  $a_i = b_i \forall i, 1 \leq i \leq p$ . *variabler Länge*
- Ein Code  $c : A \rightarrow \{0, 1\}^*$  heißt **Präfixcode**, falls es kein Paar  $i, j \in \{1, \dots, m\}$  gibt, so dass  $c(a_i)$  Präfix von  $c(a_j)$ .

- Der Huffman-Code ist ein Präfixcode.
- Bei Präfixcodes können Wörter über  $\{0, 1\}$  eindeutig dekodiert werden. (Sie entsprechen Binärbäumen mit Codewörtern an den Blättern.)
- Huffman-Code ist ein bzgl. mittlerer Codelänge optimaler Präfixcode (unter Voraussetzung einer bekannten Häufigkeitsverteilung) - ohne Beweis.

## Definition

Sei  $A$  ein Alphabet der Größe  $m$ .



- $a_1 \dots a_p \in A^*$  heißt **Präfix** von  $b_1 \dots b_l \in A^*$ , falls  $p \leq l$  und  $a_i = b_i \forall i, 1 \leq i \leq p$ .
- Ein Code  $c : A \rightarrow \{0, 1\}^*$  heißt **Präfixcode**, falls es kein Paar  $i, j \in \{1, \dots, m\}$  gibt, so dass  $c(a_i)$  Präfix von  $c(a_j)$ .
  - Der Huffman-Code ist ein Präfixcode.
  - Bei Präfixcodes können Wörter über  $\{0, 1\}$  eindeutig dekodiert werden. (Sie entsprechen Binärbäumen mit Codewörtern an den Blättern.)
  - Huffman-Code ist ein bzgl. mittlerer Codelänge optimaler Präfixcode (unter Voraussetzung einer bekannten Häufigkeitsverteilung) - ohne Beweis.

# SMILE – Präfixcodes

---

Frage: Welche dieser Codes sind Präfixcodes

- a.  $c('A') = \underline{01}$ ,  $c('B') = \underline{110}$ ,  $c('C') = \underline{011} \rightarrow \text{Kein!}$
- b.  $c('A') = \underline{01}$ ,  $c('B') = \underline{110}$ ,  $c('C') = \underline{111} \rightarrow \text{JA}$
- c.  $c('1') = xz$ ,  $c('2') = xy$ ,  $c('3') = yz \rightarrow \text{JA}$
- d. Keiner der Obigen.  $\curvearrowright$  Aussage kann nicht stimmen!

# Weitere Verfahren

---

- Es gibt zahlreiche Ansätze zur **Datenkompression**.  
(Beispiel: Lempel-Ziv-Welch.)
  - In Programmtexten gibt es häufig viele Leerzeichen, gleiche Schlüsselwörter und so weiter.
- Kodiere Folgen von Leerzeichen bzw. Schlüsselwörter durch kurze Codes.
- Das wird z.B. bei GIF und TIFF genutzt.
  - Das soll auch funktionieren, wenn man noch nicht weiß, welche Zeichenketten häufig vorkommen.



# Kapitel 2 – Kodierung

1. Kodierung von Zeichen
- 2. Kodierung von Zahlen**
3. Anwendung: ReTI

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer  
Professur für Rechnerarchitektur  
WS 2016/17

## Definition

Ein **Zahlensystem** ist ein Tripel  $S = (\underline{b}, \underline{Z}, \delta)$  mit den folgenden Eigenschaften:

- $b \geq 2$  ist eine natürliche Zahl, die **Basis** des Stellenwertsystems.
- $Z$  ist eine  $b$ -elementige Menge von Symbolen, den Ziffern.
- $\delta : Z \rightarrow \{0, 1, \dots, b - 1\}$  ist eine **Abbildung**, die jeder Ziffer umkehrbar eindeutig eine natürliche Zahl zwischen 0 und  $b - 1$  zuordnet.

# Beispiele für Zahlensysteme

---

- Dualsystem:

$$b = \underline{2} \quad Z = \{0, 1\}$$

$$\delta(0) \rightarrow 0$$
$$\delta(1) \rightarrow 1$$

- Oktalsystem:

$$b = 8 \quad Z = \{0, 1, 2, 3, 4, 5, 6, 7\}$$

- Dezimalsystem:

$$b = 10 \quad Z = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$\delta(A) = 10$$

- Hexadezimalsystem:

$$b = 16 \quad Z = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$$

$$\begin{matrix} 1 & 1 & 1 & 1 & 1 \\ 10 & 11 & 12 & 13 & 14 \end{matrix} \curvearrowright 15$$

$$\delta: \mathbb{Z} \rightarrow \mathbb{N}$$

## Definition

Eine **Festkommazahl** ist eine endliche Folge von Ziffern aus einem Zahlensystem zur Basis  $b$  mit Ziffernmenge  $Z$ .

- Sie besteht aus  $n+1$  Vorkommastellen ( $n \geq 0$ ) und  $k \geq 0$  Nachkommastellen.  
*nicht-negative Zahl*
- Der Wert  $\langle d \rangle$  einer nicht-negativen Festkommazahl  
 $d = d_nd_{n-1}\dots d_1d_0d_{-1}\dots d_{-k}$  mit  $d_i \in Z$  ist gegeben durch

$$\langle d \rangle = \sum_{i=-k}^n b^i \cdot \delta(d_i)$$

# Festkommazahlen: Schreibweise

Bsp.  $n=1, k=2$

$d_2 \ d_1 \ d_0 \ . \ d_{-1} \ d_{-2}$   $\approx 1, \frac{1}{2}^{\prime\prime}$

$d_0 \ d_{-1} \ d_{-2}$   $= 1,5_{10}$

$2^{-2} = \frac{1}{4}$

$2^{-1} = \frac{1}{2}$

- Vorkomma- und Nachkommastellen werden zur Verdeutlichung durch ein **Komma** oder einen **Punkt** getrennt:

$$d = d_n d_{n-1} \dots d_1 d_0 \underline{.} d_{-1} \dots d_{-k}$$

- Um anzudeuten, welches Zahlensystem zu Grunde liegt, wird gelegentlich die **Basis als Index** an die Ziffernfolge angehängt.

$d_3 \ d_2 \ d_1 \ d_0 \ . \ d_{-1} \ d_{-2} \dots$   $\rightsquigarrow$  Keine Nachkommastelle

- Beispiel ( $n=3, k=0$ ):

$$\begin{array}{rcl} 0110_2 & = & 6 = 2^2 + 2^1 = 4 + 2 = 6 \\ 0110_8 & = & 72 = 8^2 + 8^1 = 64 + 8 = 72 \\ 0110_{10} & = & 110 \checkmark \\ 0110_{16} & = & 272 = 16^2 + 16 = 256 + 16 = 272 \end{array}$$

# Negative Festkommazahlen

(Im Folgenden wird Basis 2 angenommen.)

- Bei der Darstellung negativer Festkommazahlen nimmt die höchstwertigste Stelle  $d_n$  eine Sonderrolle ein:  
■ Ist  $d_n = 0$ , so handelt es sich um eine nichtnegative Zahl.
- Bei der Darstellung negativer Zahlen gibt es folgende Alternativen:
  - Darstellung durch Betrag und Vorzeichen:  
 $[d_n, d_{n-1}, \dots, d_0, d_{-1}, \dots, d_{-k}]_{BV} := (-1)^{d_n} \sum_{i=-k}^{n-1} d_i 2^i$  ← Spiegelung "an der 0"
  - Einer-Komplement-Darstellung:  
 $[d_n, d_{n-1}, \dots, d_0, d_{-1}, \dots, d_{-k}]_1 := \sum_{i=-k}^{n-1} d_i 2^i - d_n (2^n - 2^{-k})$
  - Zweier-Komplement-Darstellung:  
 $[d_n, d_{n-1}, \dots, d_0, d_{-1}, \dots, d_{-k}]_2 := \sum_{i=-k}^{n-1} d_i 2^i - d_n 2^n$

# Betrag und Vorzeichen

Vz Bsp.:  $n=3, k=2$   
 $d_3 d_2 d_1 d_0, d_{-1} d_{-2}$

$$[d_n, d_{n-1}, \dots, d_0, d_{-1}, \dots, d_{-k}]_{BV} := (-1)^{d_n} \sum_{i=-k}^{n-1} d_i 2^i$$

$\left. \begin{matrix} \{ & \} \\ \frac{1}{2} & \frac{1}{4} \end{matrix} \right\}$

**Beispiel:**  $n = 2, k = 0$   $\rightarrow$  keine Nachkommastellen

$a$	000	001	010	011	100	101	110	111
	0	1	2	3	0	-1	-2	-3
$[a]_{BV}$	0	1	2	3	0	-1	-2	-3

$2^n - 2^{-k} = 2^3 - 2^{-2} = 8 - \frac{1}{4} = 7\frac{3}{4}$   
 Wert =

Bsp. mit  $n=2, k=0$ :  $2^n - 2^{-k} = 2^2 - 2^0 = 4 - 1 = 3$

- Der Zahlenbereich ist **symmetrisch**:
  - Kleinste Zahl:  $-(2^n - 2^{-k})$ , größte Zahl:  $2^n - 2^{-k}$
- Man erhält zu  $a$  die **inverse** Zahl, indem man das erste Bit komplementiert.  
 $\underline{\underline{= -a}}$
- Zwei Darstellungen für die **Null** (000 und 100 im Beispiel).
- „Benachbarte Zahlen“ haben gleichen **Abstand  $2^{-k}$** .

## Einer-Komplement

$$[a]_1 + [a']_1 = \sum_{i=0}^n a_i'' + \sum_{i=0}^n \bar{a}_i''' - \left( \begin{array}{c} a_n \\ a_n \end{array} \right) \dots = 0$$

größte darst.  
Wert

$$[d_n, d_{n-1}, \dots, d_0, d_{-1}, \dots, d_{-k}]_1 := \sum_{i=-k}^{n-1} d_i 2^i - d_n (2^n - 2^{-k})$$

**Beispiel:**  $n = 2, k = 0$

$a$	000	001	010	011	100	101	110	111
	0	1	2	3	-3	-2	-1	0
$[a]_1$	0	1	2	3	-3	-2	-1	0

$\xrightarrow{\text{komp. aller Bits}}$

- Der Zahlenbereich ist **symmetrisch**:  $-(2^n - 2^{-k}) \dots 2^n - 2^{-k}$
- Zwei Darstellungen für die **Null** (000 und 111 im Beispiel).
- „Benachbarte Zahlen“ haben gleichen **Abstand**  $2^{-k}$ .
- Man erhält zu  $a$  die **inverse Zahl**, indem man alle Bits komplementiert (siehe Lemma nächste Folie).

# Einer-Komplement: Inversion

## Lemma

Sei  $a$  eine Festkommazahl,  $a'$  die Festkommazahl, die aus  $a$  durch Komplementieren aller Bits ( $0 \rightarrow 1, 1 \rightarrow 0$ ) hervorgeht.

Dann gilt  $\underline{[a']_1 = -[a]_1}$ .

*t.z.:*  $\boxed{[a]_1 + [a']_1 = 0}$   
 $= [a_1] - [a'_1]$

**Beweisidee:** Addiert man Bits  $n-1 \dots -k$  von  $a$  und  $a'$ , erhält man  $111\dots11$ . Das ist aber gerade

$$(2^n - 2^{-k}) = (a_n + a'_n) \cdot (2^n - 2^{-k}).$$

$$\begin{array}{r} (a_{n-1}, \dots, a_{-k}) & 011011 \\ (a'_{n-1}, \dots, a'_{-k}) & 100100 \\ \hline 111111 \end{array}$$

mit  $\underline{<111111>} = <(a_{n-1}, \dots, a_{-k})> + <(a'_{n-1}, \dots, a'_{-k})>$

größte Zahl / Betrag  
darstellt b.

## Zweier-Komplement

$$\begin{array}{ccccccccc} \overbrace{100 \quad 101 \quad 110 \quad 111}^{\rightarrow} & & \overbrace{000 \quad 001 \quad 010 \quad 011}^{\rightarrow} \\ \hline 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -4 & -3 & -2 & -1 & 0 & 1 & 2 & 3 & \\ [d_n, d_{n-1}, \dots, d_0, d_{-1}, \dots, d_{-k}]_2 := \sum_{i=-k}^{n-1} d_i 2^i - \underline{d_n 2^n} \end{array}$$

Beispiel:  $n = 2, k = 0$

$a$	000	001	010	011	100	101	110	111
$[a]_2$	0	1	2	3	-4	-3	-2	-1

$$= 2^2 - 1 = 3$$

- Der Zahlenbereich ist asymmetrisch:  $-2^n \dots 2^n - 2^{-k}$ .
- Die Zahlendarstellung ist eindeutig, auch für die Null.
- „Benachbarte Zahlen“ haben gleichen Abstand  $2^{-k}$ .
- Man erhält zu  $a$  die inverse Zahl, indem man alle Bits komplementiert und an der niederwertigsten Stelle 1 addiert (siehe Lemma nächste Folie).

# Zweier-Komplement: Inversion

$$n=5, k=0 \quad d_5 \ d_4 \ d_3 \ d_2 \ d_1 \ d_0$$

$$\begin{array}{r} 17_{10} = 0 \ 1 \ 0 \ 0 \ 0 \ 1 \\ \hline \end{array}$$

## Lemma

Sei  $a$  eine Festkommazahl,  $a'$  die Festkommazahl, die aus  $a$  durch Komplementieren aller Bits ( $0 \rightarrow 1, 1 \rightarrow 0$ ) hervorgeht. Dann gilt  $[a']_2 + 2^{-k} = -[a]_2$ .

*Kleinste Beleg*

**Beweisidee:** Addiert man Bits  $n-1 \dots 0$  von  $a$  und  $a'$ , erhält man  $111 \dots 11$ . Addiert man noch  $000 \dots 01$  hinzu, so erhält man gerade  $2^n = (a_n + a'_n) \cdot 2^n$ ,

$$\begin{array}{r} (a_{n-1}, \dots, a_{-k}) \\ (a'_{n-1}, \dots, a'_{-k}) \\ "2^{-k}" \\ \hline 011011 \\ 100100 \\ 000001 \\ \hline 1000000 \end{array}$$

mit

$$2^n = (2^{n-2^{-k}})^2 = 2^n$$

$$<1000000> = <(a_{n-1}, \dots, a_{-k})> + <(a'_{n-1}, \dots, a'_{-k})> + 2^{-k}$$

## Vorteil von Zweier-Komplement

Addition nach  
Schulmethode

$$\begin{array}{r} \text{--} 17 \\ + 23 \\ \hline 40 \end{array}$$

falsch

- Wir werden später Schaltungen betrachten, die zwei Zahlen als Eingaben nehmen und ihre Summe oder Differenz an den Ausgängen bereit stellen (**Addierer**, **Subtrahierer**).
- Es stellt sich heraus, dass diese Schaltungen besonders einfach sind, wenn Negativzahlen im Zweier-Komplement dargestellt werden.
- Daher wird in der Praxis oft die Zweier-Komplement-Darstellung verwendet.

# Festkommazahlen - Übersicht

## Betrag mit Vorzeichen      Einerkomplement      Zweierkomplement

$[d_n, d_{n-1}, \dots, d_0, d_{-1}, \dots, d_{-k}]_{BV}$

$$:= (-1)^{d_n} \sum_{i=-k}^{n-1} d_i 2^i$$

$[d_n, d_{n-1}, \dots, d_0, d_{-1}, \dots, d_{-k}]_1$

$$:= \sum_{i=-k}^{n-1} d_i 2^i - d_n (2^n - 2^{-k})$$

$[d_n, d_{n-1}, \dots, d_0, d_{-1}, \dots, d_{-k}]_2$

$$:= \sum_{i=-k}^{n-1} d_i 2^i - d_n 2^n$$

$$n = 2, k = 0$$

$a$	000	001	010	011	100	101	110	111
$[a]_{BV}$	0	1	2	3	0	-1	-2	-3
$[a]_1$	0	1	2	3	-3	-2	-1	0
$[a]_2$	0	1	2	3	-4	-3	-2	-1

symmetrisch

symmetrisch

asymmetrisch

kleinste Zahl

$$-(2^n - 2^{-k})$$

$$-(2^n - 2^{-k})$$

$$\underline{-2^n}$$

größte Zahl

$$2^n - 2^{-k}$$

$$2^n - 2^{-k}$$

$$\underline{2^n - 2^{-k}}$$

Inverses durch

kompl. 1. Bit

kompl. alle Bits

kompl. alle Bits, add. 1

Null

2 Darstellungen

2 Darstellungen

1 Darstellung

Abstand

$$2^{-k}$$

$$2^{-k}$$

$$2^{-k}$$

# Festkommazahlen - Übersicht

## Betrag mit Vorzeichen

$[d_n, d_{n-1}, \dots, d_0, d_{-1}, \dots, d_{-k}]_{BV}$

$$:= (-1)^{d_n} \sum_{i=-k}^{n-1} d_i 2^i$$

## Einerkomplement

$[d_n, d_{n-1}, \dots, d_0, d_{-1}, \dots, d_{-k}]_1$

$$:= \sum_{i=-k}^{n-1} d_i 2^i - d_n (2^n - 2^{-k})$$

## Zweierkomplement

$[d_n, d_{n-1}, \dots, d_0, d_{-1}, \dots]$

$$:= \sum_{i=-k}^{n-1} d_i 2^i - d_n$$



$$n = 2, k = 0$$

$a$	000	001	010	011	100	101	110	111
$[a]_{BV}$	0	1	2	3	0	-1	-2	-3
$[a]_1$	0	1	2	3	-3	-2	-1	0
$[a]_2$	0	1	2	3	-4	-3	-2	-1

symmetrisch

symmetrisch

asymmetrisch

kleinste Zahl

$$-(2^n - 2^{-k})$$

$$-(2^n - 2^{-k})$$

$$-2^n$$

größte Zahl

$$2^n - 2^{-k}$$

$$2^n - 2^{-k}$$

$$2^n - 2^{-k}$$

Inverses durch

kompl. 1. Bit

kompl. alle Bits

kompl. alle Bits, add. 1

Null

2 Darstellungen

2 Darstellungen

1 Darstellung

Abstand

$$2^{-k}$$

$$2^{-k}$$

$$2^{-k}$$

# SMILE – Festkommazahlen

$$b) -7 - (-7) = \underline{-7+7=0}$$

-7  
)

Frage: Welche der Aussagen sind wahr für die Zahl  $[1001]_2$ ?

a.  $[1001]_2 = [0111]_{BV}$  nein

b.  $[1001]_2 - [1001]_2 = [10010]_2$  -14 nein

c.  $[1001]_2$  ist das (additive) Inverse zu  $[0111]_2$  ✓

d.  $[1001]_2 = [1000]_1$  ✓

-7

$$\begin{array}{r} d_3 \quad d_2 \quad d_1 \quad d_0 \\ \hline 1 \quad 0 \quad 0 \quad 1 \end{array}$$

$n=3$   
 $k=0$

$$1001 \xrightarrow{\text{INV.}} 0110 \xrightarrow{+1} 0111$$

$$(-2^n + 2^k) + 0 = \underline{-7}$$

# Probleme von Festkommazahlen

- Betrachte die Menge aller Zahlen, die eine Zweier-Komplement-Darstellung mit n Vor- und k Nachkommastellen haben.
  - Keine ganz großen bzw. kleinen Zahlen darstellbar!
    - Zahlen mit größtem Absolutbetrag:  $-2^n$  und  $2^n - 2^{-k}$
    - Zahlen mit kleinstem Absolutbetrag:  $-2^{-k}$  und  $2^{-k}$
- Operationen sind nicht abgeschlossen!
  - $2^{n-1} + 2^{n-1}$  ist nicht darstellbar, obwohl die Operanden darstellbar sind.
- Assoziativ- und Distributivgesetz gelten nicht, da bei ihrer Anwendung evtl. der darstellbare Zahlenbereich verlassen wird!

■ Beispiel:  $(2^{n-1} + 2^{n-1}) - 2^{n-1} \neq 2^{n-1} + (2^{n-1} - 2^{n-1})$

$\underbrace{2^{n-1} + 2^{n-1}}_{\text{überlappend}} - 2^{n-1} \quad \underbrace{2^{n-1}}_{= 0}$

# Gleitkomma-Zahlen

Vor + Nach Kommasstellen  
Position des Komma einzustellen

- Die verfügbaren Bits werden in Vorzeichen S, Exponent E und Mantisse M unterteilt.

$$a = (-1)^S \cdot M \cdot 2^E$$

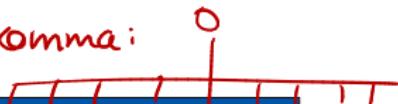
- Einfache Genauigkeit (insg. 32 Bit)

31	30 29 28 27 26 25 24 23	22 21 20 19 ... 3 2 1 0
S	Exponent E	Mantisse M

Gleitkomma:



Festkomma:



- Doppelte Genauigkeit (insg. 64 Bit)

63	62 61 60 59 ... 54 53 52	51 50 49 48 ... 3 2 1 0
S	Exponent E	Mantisse M

- Implementierungsdetails: siehe z.B. IEEE754-Standard



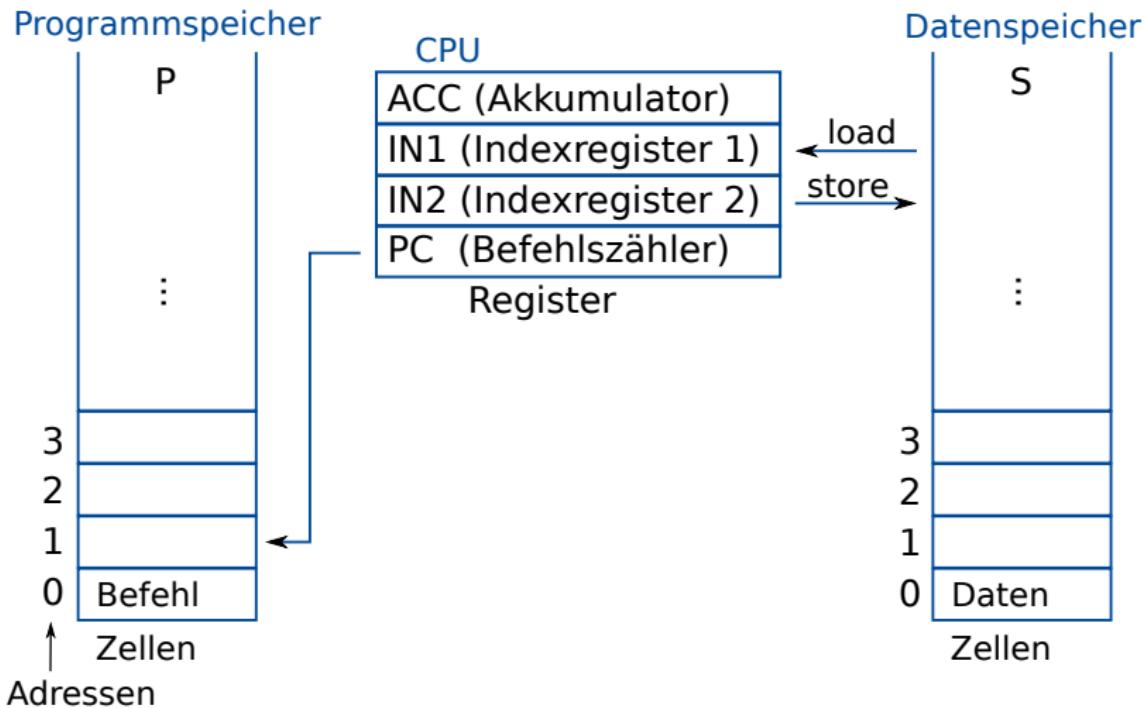
# Kapitel 2 – Kodierung

1. Kodierung von Zeichen
2. Kodierung von Zahlen
- 3. Anwendung: ReTI**

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer  
Professur für Rechnerarchitektur  
WS 2016/17

# Realisierung von ReTI



# Unterschiede abstrakter/realer Rechner

*Adressbereiche / Raum  
32 Bit*

$$2^{32} \cdot 4 \text{ Byte} = 16 \text{ GB}$$

- Bei realer Maschine nur ein Speicher  $M$  für Daten und Befehle.
  - $M$  ist endlich (Größe  $2^{32}$ ). Für  $i \in \{0, \dots, 2^{32} - 1\}$  ist  $\underline{M(i)}$  Inhalt der  $i$ -ten Speicherzelle.
  - Speicherzellen können Elemente aus  $\underline{\mathbb{B}^{32}}$  aufnehmen.
- CPU-Register  $PC$ ,  $ACC$ ,  $IN1$  und  $IN2$  können nur Elemente  $w \in \underline{\mathbb{B}^{32}}$  aufnehmen.  $w$  heißt Wort.
  - Ein Wort kann als Binärzahl (z.B. Adresse im  $M$ ), Zweierkomplementzahl oder Bitstring interpretiert werden.
- Befehle sind ebenfalls Wörter aus  $\mathbb{B}^{32}$ .

# Notation

---

- $\underline{b}^j = \underbrace{(b, \dots, b)}_{j \text{ mal}}$  für  $b \in \{0, 1\}$

- $\langle A \rangle := B$

(A Register oder Speicherzelle,  $B \in \{0, \dots, 2^{32} - 1\}$ )  
bedeutet  $A := bin_{32}(B)$

- Beispiel:  $\underline{\langle PC \rangle} := \underline{\langle PC \rangle + 1}$

- $\underline{[A]} := B$

(A Register oder Speicherzelle,  $B \in \{-2^{31}, \dots, 2^{31} - 1\}$ )  
bedeutet  $A := twoc_{32}(B)$   
 $B$  wird als Zweierkomplement-Zahl interpretiert.

# Befehlsformate

ADDI i:

$$\text{``ACC} = \text{ACC} + \underline{i}$$

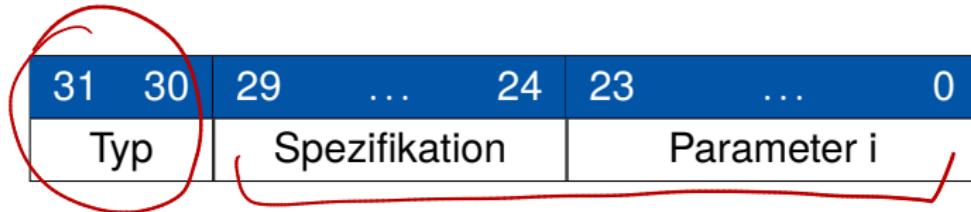
- Zur Erinnerung: Der Befehlssatz von ReTI besteht aus Load-/Store-, Compute-, Indexregister- und Sprungbefehlen.
- Sie werden als Wörter aus  $\mathbb{B}^{32}$  kodiert. Etwaige Parameter sind in der Kodierung enthalten.
  - Notation: Sei  $I = i_{31}, \dots, i_0 \in \mathbb{B}^{32}$ .  
 $I[y, x] := i_y, i_{y-1}, \dots, i_x$  für  $0 \leq x \leq y \leq 31$ .
- Allgemeines Instruktionsformat:  
I

31	30	29	...	24	23	...	0
Typ	Spezifikation			Parameter i			

$I[31:30] \approx \text{Typ des Befehls!}$

# Typ einer Instruktion

I[31, 30]	Typ
0 0	Compute
0 1	Load ←
1 0	Store, Move
1 1	Jump



# Load-Befehle: Kodierungsprinzip

31	30	29	28	27	26	25	24	23	...	0
0	1	<u>M</u>	*		<u>D</u>			i		

*nicht genügt*

- M: Modus
- D: Vorerst irrelevant
  - ↳ Destination

# Load-Befehle: Kodierung

Typ	Modus	Befehl	Wirkung	
0 1	0 0	LOAD $i$	$\underline{ACC} := M(\langle i \rangle)$ $\langle PC \rangle := \langle PC \rangle + 1$	
0 1	0 1	LOADIN1 $i$	$\underline{ACC} := M(\langle IN1 \rangle + [i])$ $\langle PC \rangle := \langle PC \rangle + 1$	
0 1	1 0	LOADIN2 $i$	$\underline{ACC} := M(\langle IN2 \rangle + [i])$ $\langle PC \rangle := \langle PC \rangle + 1$	
0 1	1 1	LOADI $i$	$\underline{ACC} := 0^8 i$ 32 Bit $\langle PC \rangle := \langle PC \rangle + 1$	

Durchführung von Rechnungen  $\langle x \rangle + [y]$  später.

$0^8 i$

$00000000 i_3 \dots i_0$   
32 Bit

// d.h. wir füllen mit 0 auf

# Store-, Move-Befehle: Prinzip

31	30	29	28	27	26	25	24	23	...	0
1	0	M		S		D			i	

- M: Modus
- S: Source
- D: Destination

Kodierung S, D		Register
S, D		
0 0	PC	]
0 1	IN1	
1 0	IN2	
1 1	ACC	]

# Store-, Move-Befehle: Kodierung

Typ	Modus	Befehl	Wirkung	
1 0	0 0	STORE $i$	$M(\langle i \rangle) := ACC$	$\langle PC \rangle := \langle PC \rangle + 1$
1 0	0 1	STOREIN1 $i$	$M(\langle IN1 \rangle + [i]) := ACC$	$\langle PC \rangle := \langle PC \rangle + 1$
1 0	1 0	STOREIN2 $i$	$M(\langle IN2 \rangle + [i]) := ACC$	$\langle PC \rangle := \langle PC \rangle + 1$
1 0	1 1	MOVE $S D$	$D := S$	$\langle PC \rangle := \langle PC \rangle + 1$

↑      ↗  
außer bei  $D = 00 (PC)$

# Compute-Befehle: Prinzip

---

31	30	29	28	27	26	25	24	23	...	0
<u>0</u>	<u>0</u>	<u>MI</u>		<u>F</u>		<u>D</u>			i	

- MI: „compute memory“/„compute immediate“
- F: Funktionsfeld
- D: Vorerst irrelevant  
*Destination*

# Compute-Befehle: Kodierung



**EXOR**  
„Exklusiv-ODer“

Typ	MI	F	Befehl	Wirkung
0 0	0	0 1 0	SUBI $i$	$[ACC] := [ACC] - [i]$ $\langle PC \rangle := \langle PC \rangle + 1$
		0 1 1	ADDI $i$	$[ACC] := [ACC] + [i]$ $\langle PC \rangle := \langle PC \rangle + 1$
		1 0 0	OPLUSI $i$	$ACC := ACC \oplus 0^8 i$ $\langle PC \rangle := \langle PC \rangle + 1$
		1 0 1	ORI $i$	$ACC := ACC \vee 0^8 i$ $\langle PC \rangle := \langle PC \rangle + 1$
		1 1 0	ANDI $i$	$ACC := ACC \wedge 0^8 i$ $\langle PC \rangle := \langle PC \rangle + 1$
0 0	1	0 1 0	SUB $i$	$[ACC] := [ACC] - [M(\langle i \rangle)]$ $\langle PC \rangle := \langle PC \rangle + 1$
		0 1 1	ADD $i$	$[ACC] := [ACC] + [M(\langle i \rangle)]$ $\langle PC \rangle := \langle PC \rangle + 1$
		1 0 0	OPLUS $i$	$ACC := ACC \oplus M(\langle i \rangle)$ $\langle PC \rangle := \langle PC \rangle + 1$
		1 0 1	OR $i$	$ACC := ACC \vee M(\langle i \rangle)$ $\langle PC \rangle := \langle PC \rangle + 1$
		1 1 0	AND $i$	$ACC := ACC \wedge M(\langle i \rangle)$ $\langle PC \rangle := \langle PC \rangle + 1$

# Bitstring-Operationen am Beispiel von OPLUS

$a$	$b$	$\text{EXOR}(a, b)$
0	0	0
0	1	1
1	0	1
1	1	0

■  $ACC := ACC \oplus 0^8 i_{23} \dots i_0$

$$\cong (\underbrace{ACC_{31} \oplus 0, \dots, ACC_{24} \oplus 0}_{\uparrow}, \underbrace{ACC_{23} \oplus i_{23}, \dots, ACC_0 \oplus i_0}_{\uparrow})$$

# Sprungbefehle: Prinzip

nicht benutzt

31	30	29	28	27	26	25	24	23	...	0
1 1		C		*				i		

- C: Condition

gilt „ACC c 0“ dann Sprung!

C	Bedingung c
0 0 0	nie
0 0 1	>
0 1 0	=
0 1 1	$\geq$
1 0 0	<
1 0 1	$\neq$
1 1 0	$\leq$
1 1 1	immer

NOP  $\rightarrow$  No operation,  
nur  $PC = PC + 1$

unbedingter  
Sprung

# Bedingungskodierung nach Schema

C	Bedingung c
0 0 0	nie
0 0 1	>
0 1 0	=
0 1 1	$\geq$
1 0 0	<
1 0 1	$\neq$
1 1 0	$\leq$
1 1 1	immer

nur  $I[29] = 1 \Leftrightarrow <$  wird abgefragt  
nur  $I[28] = 1 \Leftrightarrow =$  wird abgefragt  
nur  $I[27] = 1 \Leftrightarrow \geq$  wird abgefragt

Andere Abfragen durch Kombinationen,  
z.B.  $C = 101 :< \text{ oder } >, \text{ also } \neq.$

# Sprungbefehle: Kodierung

Typ	Befehl	Wirkung
1 1	JUMP <sub>c</sub> i	$\langle PC \rangle := \begin{cases} \langle PC \rangle + [i], & \text{falls } [ACC] c 0 \\ \langle PC \rangle + 1, & \text{sonst} \end{cases}$

- „immer“
- Unbedingte Sprünge werden durch  $C = 111$  ausgedrückt.
  - Bei  $C = \underline{000}$ : Keine Wirkung des Befehls außer Inkrementieren des Befehlszählers  
⇒ NOP - Befehl (No Operation)

(In Kap. 8 werden wir kurz auf die Notwendigkeit von NOP - Befehlen bei Pipelining eingehen.)

# Zusätzliche Befehle

---

- Zusätzliche Befehle sind durchaus sinnvoll und bei anderen Architekturen evtl. schon als Grundbefehl vorhanden.
- Nicht vorhandene Befehle müssen hier durch **Befehlsfolgen** "simuliert" werden.
- Beispiel: Multiplikation, siehe Kapitel 1.2..

# Addition und Sign Extension

## ■ Probleme bei **Additionen**:

- 1 Addition verschieden langer Zahlen (z.B. [ACC] + [i]).  
*32 Bit / 24 Bit*
- 2 Addition von Binärdarstellungen und  
Zweierkomplementzahlen (z.B. M(<IN1> + [i]) := ACC).  
*Bin.      2er-kompl.*

## ■ Zu 1: Lösung durch Sign Extension

- Sei  $y \in \mathbb{B}^{24}$ .  $\text{sext}(y) := \underbrace{y^8}_{\text{sign}} \underbrace{y_{23}}_{\text{ext}}$  heißt **sign extension** von  $y$ .
- Es gilt:  $[y] = [\text{sext}(y)]$ . (Beweis: Übung)
- Dann wird  $[ACC] + [i]$  zurückgeführt auf  $[ACC] + [\text{sext}(i)]$ .  
*32 Bit    32 Bit*

## ■ Zu 2: Siehe nächste Folie.

# Addition von Binär- und Zweierkomplementzahlen

*Binärzahl* *carry out / Übertrag* *Summe*

**Lemma**

Sei  $x \in \mathbb{B}^{32}$ ,  $y \in \mathbb{B}^{24}$ ,  $0 \leq \langle x \rangle + [y] < 2^{32}$  und sei  
 $\langle x \rangle + \langle \text{sext}(y) \rangle = \langle c, s \rangle$  mit  $c \in \mathbb{B}$ ,  $s \in \mathbb{B}^{32}$ .  
Dann gilt:  $\langle x \rangle + [y] = \langle s \rangle$

- Bedeutung: Kommt es beim Addieren nicht zum Überlauf ( $0 \leq \langle x \rangle + [y] < 2^{32}$ ), so kann man  $x$  und  $y$  als Binärzahlen interpretieren, addieren und Übertrag ignorieren!
- Zunächst ohne Beweis.
- Wir können somit Parameter  $i$  bei den Befehlen ohne Fallunterscheidung nach  $i$  positiv / negativ verwenden!

# Zusammenfassung

---

- Kodierungen von Zeichen: Codes fester Länge (z.B. ASCII) sind einfacher aber weniger effizient als Codes variabler Länge (z.B. Huffman).
- Zweier-Komplement-Kodierung von Festkomma-Zahlen erlaubt in Verbindung mit Sign Extension effiziente Umsetzung arithmetischer Operationen in Hardware eines Rechners (tatsächliche Implementierung siehe Kapitel 3.5).
- Der Befehlssatz von ReTI ist auf der nächsten Folie zusammengefasst.

Load-Befehle		$I[25 : 24] = D$
$I[31 : 28]$	Befehl	Wirkung
0100	LOAD $D i$	$D := M(\langle i \rangle)$
0101	LOADIN1 $D i$	$D := M(\langle IN1 \rangle + [i])$
0110	LOADIN2 $D i$	$D := M(\langle IN2 \rangle + [i])$
0111	LOADI $D i$	$D := 0^8 i$

Store-Befehle		$MOVE: I[27 : 24] = SD$
$I[31 : 28]$	Befehl	Wirkung
1000	STORE $i$	$M(\langle i \rangle) := ACC$
1001	STOREIN1 $i$	$M(\langle IN1 \rangle + [i]) := ACC$
1010	STOREIN2 $i$	$M(\langle IN2 \rangle + [i]) := ACC$
1011	MOVE $S D$	$D := S$ $\langle PC \rangle := \langle PC \rangle + 1$ falls $D \neq PC$

Compute-Befehle		$I[25 : 24] = D$
$I[31 : 26]$	Befehl	Wirkung
000010	SUBI $D i$	$[D] := [D] - [i]$
000011	ADDI $D i$	$[D] := [D] + [i]$
000100	OPLUSI $D i$	$D := D \oplus 0^8 i$ $\langle PC \rangle := \langle PC \rangle + 1$ falls $D \neq PC$
000101	ORI $D i$	$D := D \vee 0^8 i$
000110	ANDI $D i$	$D := D \wedge 0^8 i$
001010	SUB $D i$	$[D] := [D] - [M(\langle i \rangle)]$
001011	ADD $D i$	$[D] := [D] + [M(\langle i \rangle)]$
001100	OPLUS $D i$	$D := D \oplus M(\langle i \rangle)$ $\langle PC \rangle := \langle PC \rangle + 1$ falls $D \neq PC$
001101	OR $D i$	$D := D \vee M(\langle i \rangle)$
001110	AND $D i$	$D := D \wedge M(\langle i \rangle)$

Jump-Befehle		
$I[31 : 27]$	Befehl	Wirkung
11000	NOP	$\langle PC \rangle := \langle PC \rangle + 1$
11001	JUMP $> i$	
11010	JUMP $= i$	
11010	JUMP $> i$	
11011	JUMP $< i$	$\langle PC \rangle := \begin{cases} \langle PC \rangle + [i], & \text{falls } [ACC] c 0 \\ \langle PC \rangle + 1, & \text{sonst} \end{cases}$
11100	JUMP $\neq i$	
11110	JUMP $< i$	
11111	JUMP $i$	$\langle PC \rangle := \langle PC \rangle + [i]$



Kodierung S,D

S, D	Register
0 0	PC
0 1	IN1
1 0	IN2
1 1	ACC

# SMILE – Programme

Ein ReTI-Programm  $P$  sieht folgendermaßen aus:  $P =$

```
011111000000000000000000000000001110000000000000000000000000000000000  
000011110000000000000000000000011000000000000000000000000000000000000011
```

Frage: Welche der Aussagen über das Programm  $P$  sind wahr?

ADDI      ACC = ACC + 1     $\sim ACC = 2$

- a. Der  $PC$  ist nach Ausführung des Programms um 3 erhöht. Nein, um 4 Store
- b. Das Programm besteht aus LOADI, NOP, ADDI, STORE. ✓
- c. Nach Ausführung des Programms steht in Speicherzelle 3 eine 2. ✓
- d. Das Programm verwendet alle möglichen Register der ReTI-Maschine. IN1, IN2 nicht
- e. Das Programm enthält einen Befehl, den man immer weglassen kann, da er nie Auswirkungen hat.

# Kapitel 3 – Kombinatorische Logik

## 1. Kombinatorische Schaltkreise

2. Normalformen, zweistufige Synthese
3. Berechnung eines Minimalpolynoms
4. Arithmetische Schaltungen
5. Anwendung: ALU von ReTI

Albert-Ludwigs-Universität Freiburg

UNI  
FREIBURG

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur  
WS 2016/17

# Kombinatorische Schaltkreise

## Definition

**Kombinatorische Logik** ist ein Modell von Hardware, die eine boolesche Funktion  $f : \mathbb{B}^n \rightarrow \mathbb{B}^m$  ( $n, m \in \mathbb{N}$ ) implementiert.

- Ein **kombinatorischer Schaltkreis** (Schaltnetz) hat  $n$  Eingänge und  $m$  Ausgänge. Legt man an den Eingängen den Vektor  $i \in \mathbb{B}^n$  an, berechnet der Schaltkreis den Vektor  $f(i) \in \mathbb{B}^m$  und stellt ihn an den Ausgängen bereit.
- Es gibt weitere Arten von Hardware:
  - Sequentielle Logik mit speichernden Elementen (später).
  - Analog- und Mixed-Signal-Blöcke (nicht in TI).

# Kombinatorische Logiksynthese

---

- Kombinatorische Logiksynthese ist das Problem, zu einer gegebenen Booleschen Funktion einen möglichst effizienten kombinatorischen Schaltkreis, d. h. einen mit möglichst geringen Kosten, zu finden.
- Die Definition von Kosten hängen von der verwendeten Technologie ab und können sich auf die Größe, Verzögerung, Energieverbrauch des Schaltkreises beziehen und eventuell weitere Parameter (Zuverlässigkeit, Testbarkeit, ...) berücksichtigen.

# Technologien

---

- Wir konzentrieren uns hier auf zwei Arten von Technologien:
  - 1 **Programmierbare Logikfelder** (Programmable Logic Arrays, PLAs).
    - Implementieren sogenannter zweistufigen Realisierungen.
  - 2 Mehrstufige Realisierungen mit allgemeinen Bibliothekszellen (**Logik-Gattern**).

# Logikgatter

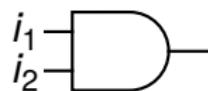
---

Gatter : AND, OR, XOR

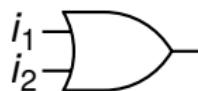
- Gatter sind kleine kombinatorische Blöcke, in der Regel mit bis zu 4 Eingängen und einem Ausgang.
- Gatter werden mit Transistoren realisiert.
- Gatter können zu größeren Schaltungen verbunden werden.
- Die Menge der verfügbaren Gattern ergibt eine Standardzellen-Bibliothek.

# Einige wichtige Gatter

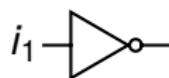
$i_1$	$i_2$	$AND_2$
0	0	0
0	1	0
1	0	0
1	1	1



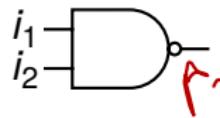
$i_1$	$i_2$	$OR_2$
0	0	0
0	1	1
1	0	1
1	1	1



$i_1$	$NOT_2$
0	1
1	0



$i_1$	$i_2$	$NAND_2$
0	0	1
0	1	1
1	0	1
1	1	0



$i_1$	$i_2$	$NOR_2$
0	0	1
0	1	0
1	0	0
1	1	0

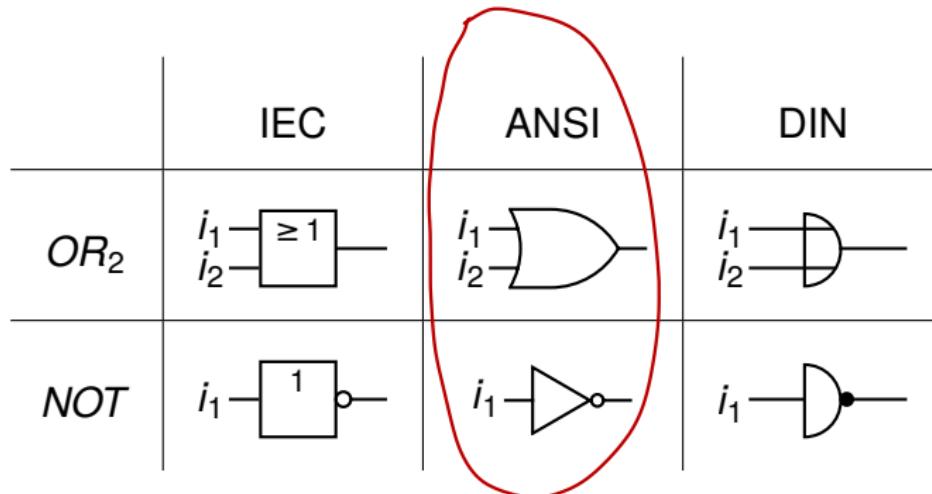


$i_1$	$i_2$	$XOR_2$
0	0	0
0	1	1
1	0	1
1	1	0



# Logikgatter - verschiedene Notationen

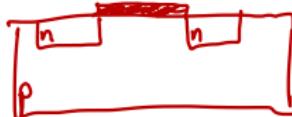
Es gibt verschiedene Notationen für Logikgatter.



Wir werden in dieser Vorlesung die **ANSI-Notation** verwenden.

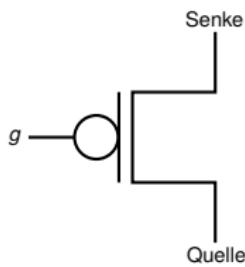
# Transistoren

MOSFET

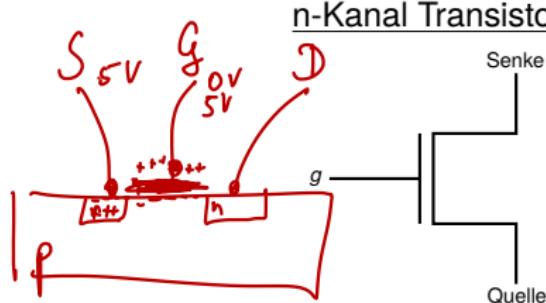


- Einen Transistor kann man vereinfacht als spannungsgesteuerten Schalter sehen:
  - Leitung  $g$  (gate) regelt Leitfähigkeit zwischen Quelle und Senke.

p-Kanal Transistor



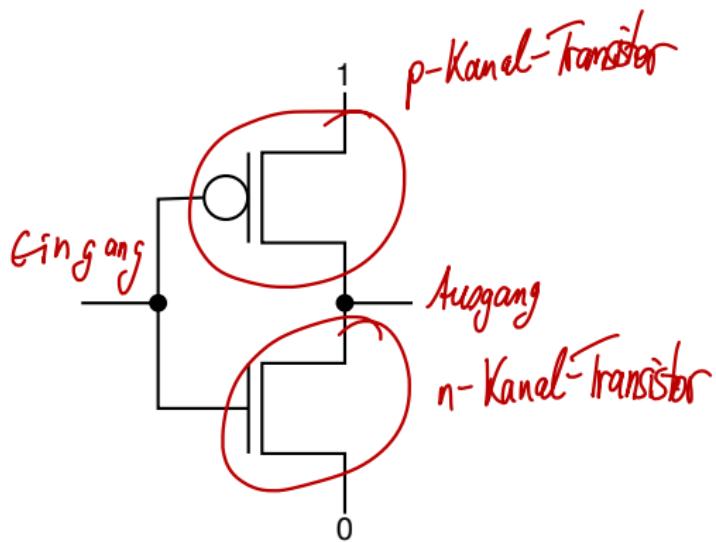
n-Kanal Transistor



- Leitet, wenn an  $g$  eine 0 anliegt.
- Sperrt, wenn an  $g$  eine 1 anliegt.
- Leitet, wenn an  $g$  eine 1 anliegt.
- Sperrt, wenn an  $g$  eine 0 anliegt.

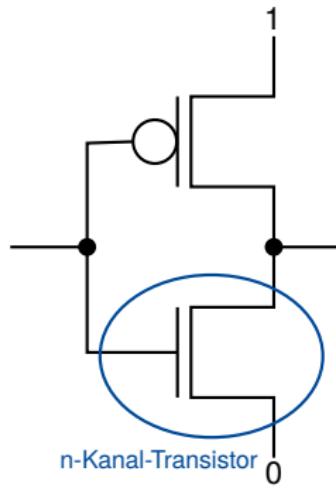
# Realisierung von Gattern in CMOS-Technologie

- Complementary Metal Oxide Semiconductor.
- Es werden p- und n-Kanal-Transistoren verwendet.
- Beispiel: CMOS-Inverter.



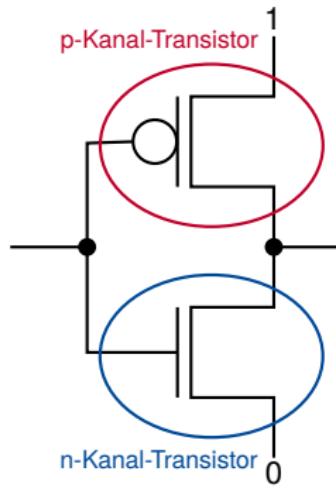
# Realisierung von Gattern in CMOS-Technologie

- Complementary Metal Oxide Semiconductor.
- Es werden p- und n-Kanal-Transistoren verwendet.
- Beispiel: CMOS-Inverter.



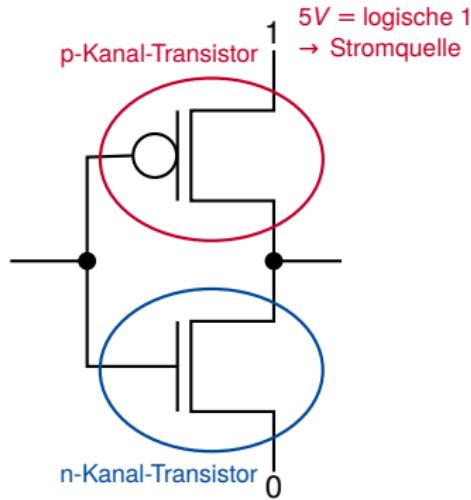
# Realisierung von Gattern in CMOS-Technologie

- Complementary Metal Oxide Semiconductor.
- Es werden p- und n-Kanal-Transistoren verwendet.
- Beispiel: CMOS-Inverter.



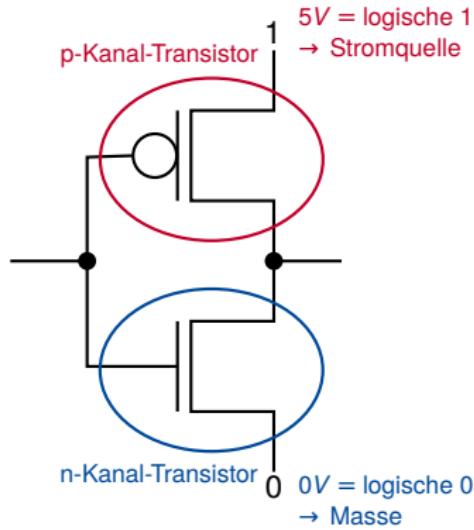
# Realisierung von Gattern in CMOS-Technologie

- Complementary Metal Oxide Semiconductor.
- Es werden p- und n-Kanal-Transistoren verwendet.
- Beispiel: CMOS-Inverter.



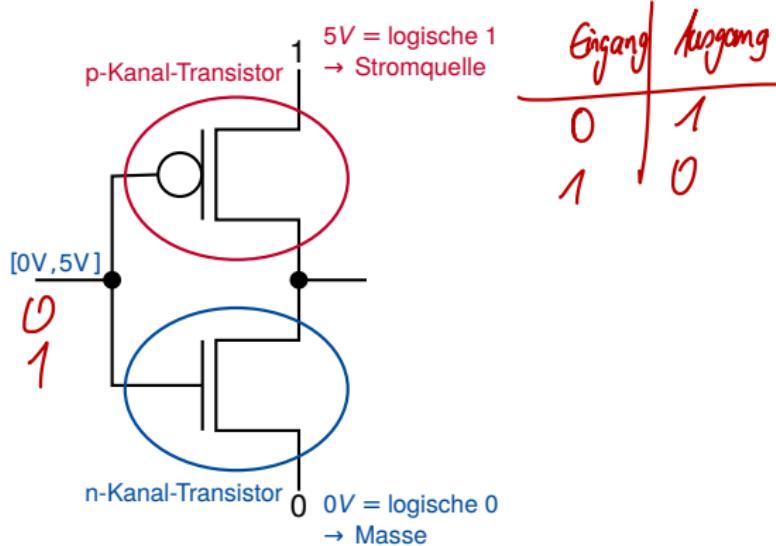
# Realisierung von Gattern in CMOS-Technologie

- Complementary Metal Oxide Semiconductor.
- Es werden p- und n-Kanal-Transistoren verwendet.
- Beispiel: CMOS-Inverter.



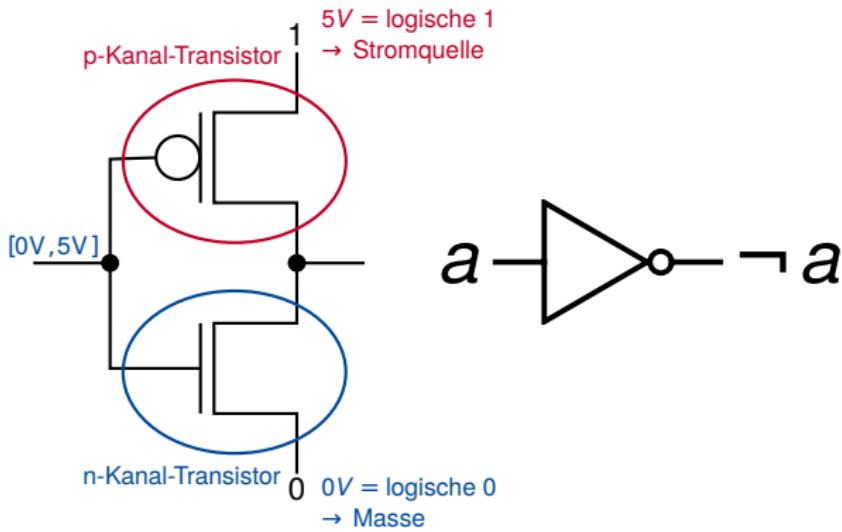
# Realisierung von Gattern in CMOS-Technologie

- Complementary Metal Oxide Semiconductor.
- Es werden p- und n-Kanal-Transistoren verwendet.
- Beispiel: CMOS-Inverter.



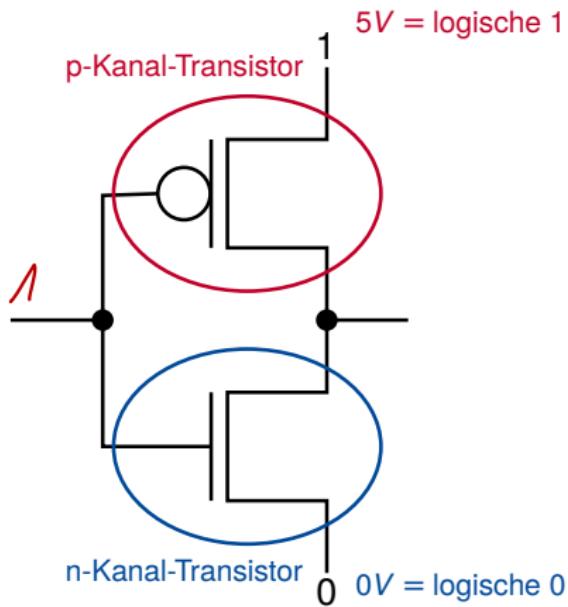
# Realisierung von Gattern in CMOS-Technologie

- Complementary Metal Oxide Semiconductor.
- Es werden p- und n-Kanal-Transistoren verwendet.
- Beispiel: CMOS-Inverter.



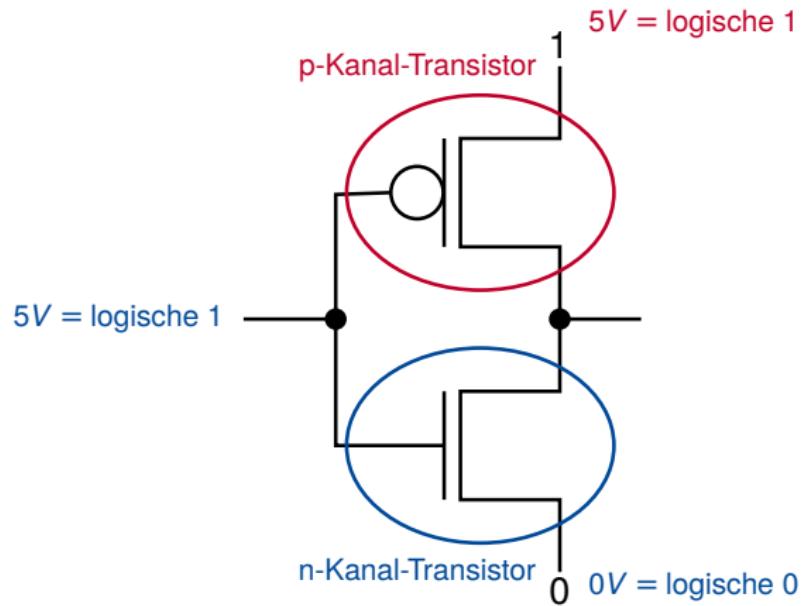
# CMOS-Inverter mit 1 am Gate

- Leitender Pfad zwischen Ausgang und Masse (logische 0).



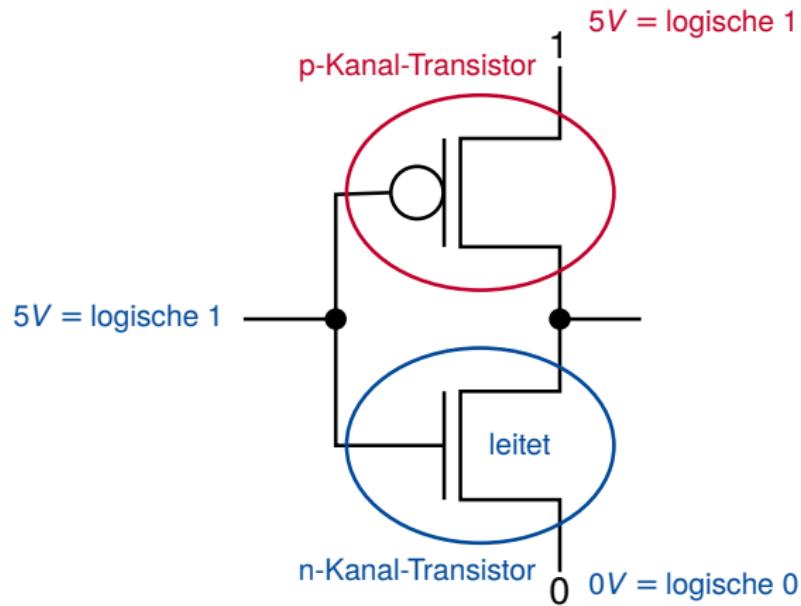
# CMOS-Inverter mit 1 am Gate

- Leitender Pfad zwischen Ausgang und Masse (logische 0).



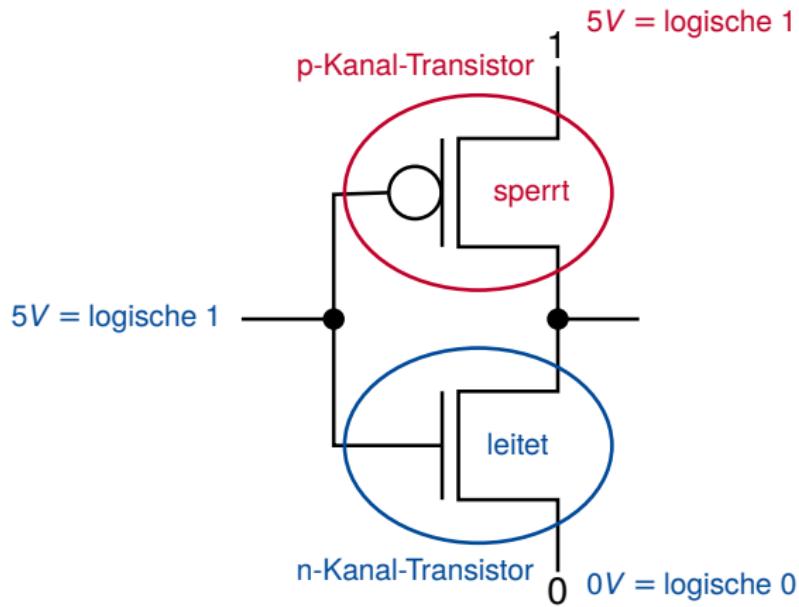
# CMOS-Inverter mit 1 am Gate

- Leitender Pfad zwischen Ausgang und Masse (logische 0).



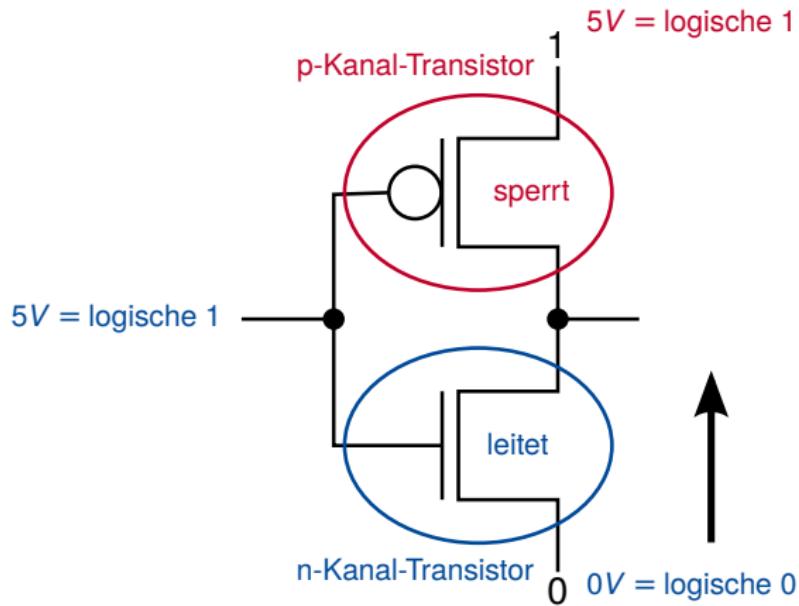
# CMOS-Inverter mit 1 am Gate

- Leitender Pfad zwischen Ausgang und Masse (logische 0).



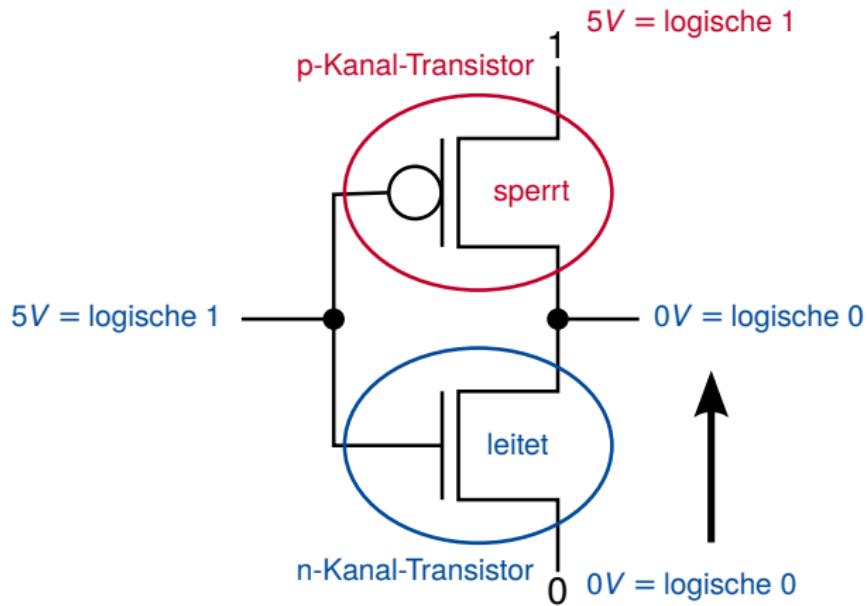
# CMOS-Inverter mit 1 am Gate

- Leitender Pfad zwischen Ausgang und Masse (logische 0).



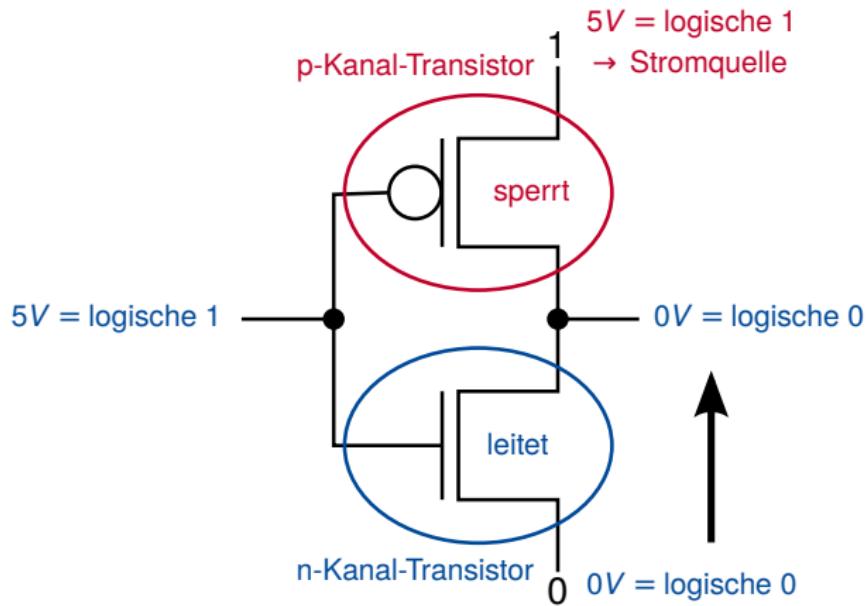
# CMOS-Inverter mit 1 am Gate

- Leitender Pfad zwischen Ausgang und Masse (logische 0).



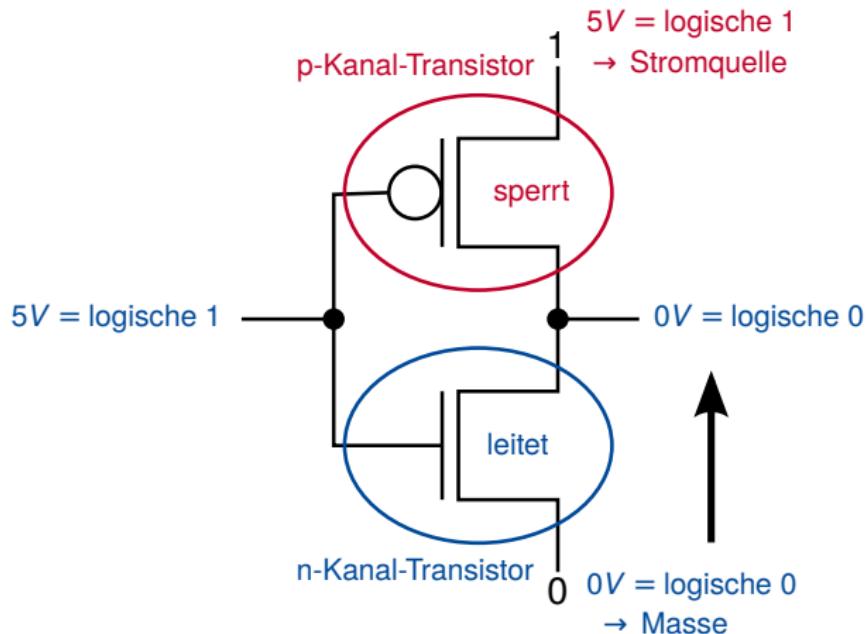
# CMOS-Inverter mit 1 am Gate

- Leitender Pfad zwischen Ausgang und Masse (logische 0).



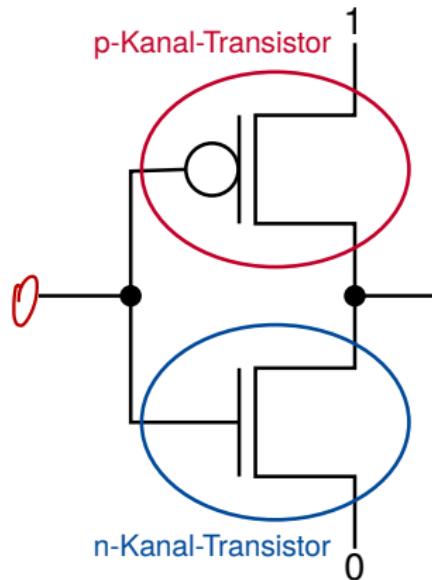
# CMOS-Inverter mit 1 am Gate

- Leitender Pfad zwischen Ausgang und Masse (logische 0).



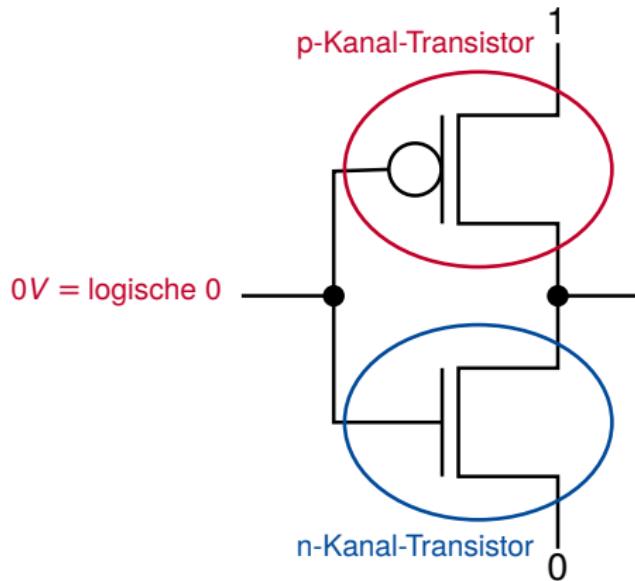
# CMOS-Inverter mit 0 am Gate

- Leitender Pfad zwischen Ausgang und Spannungsversorgung (logische 1).



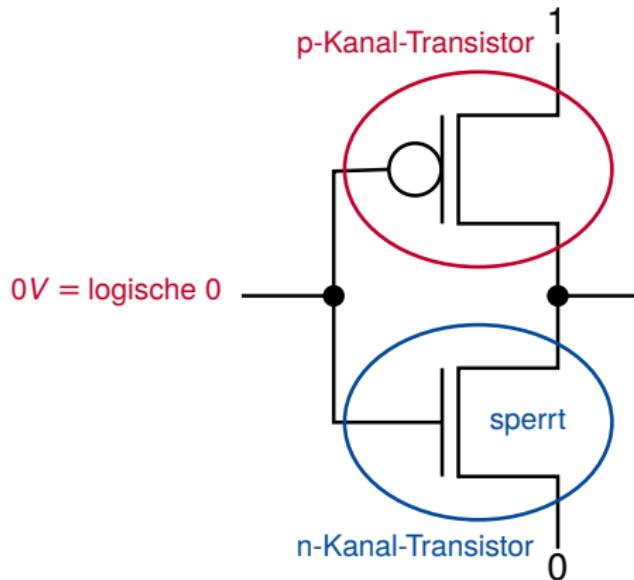
# CMOS-Inverter mit 0 am Gate

- Leitender Pfad zwischen Ausgang und Spannungsversorgung (logische 1).



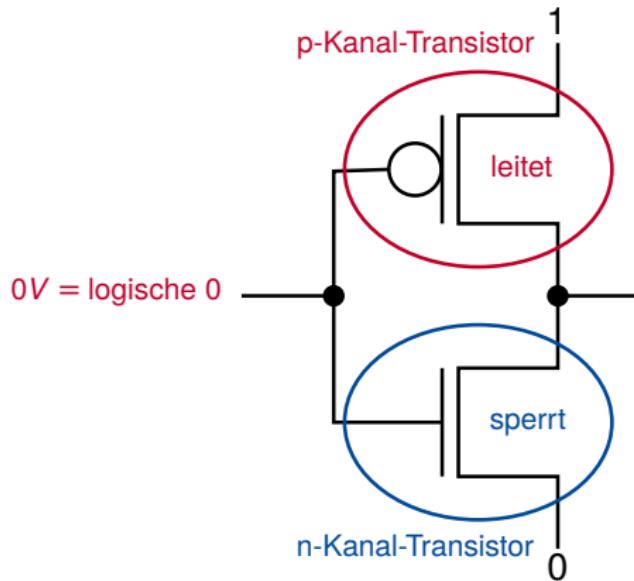
# CMOS-Inverter mit 0 am Gate

- Leitender Pfad zwischen Ausgang und Spannungsversorgung (logische 1).



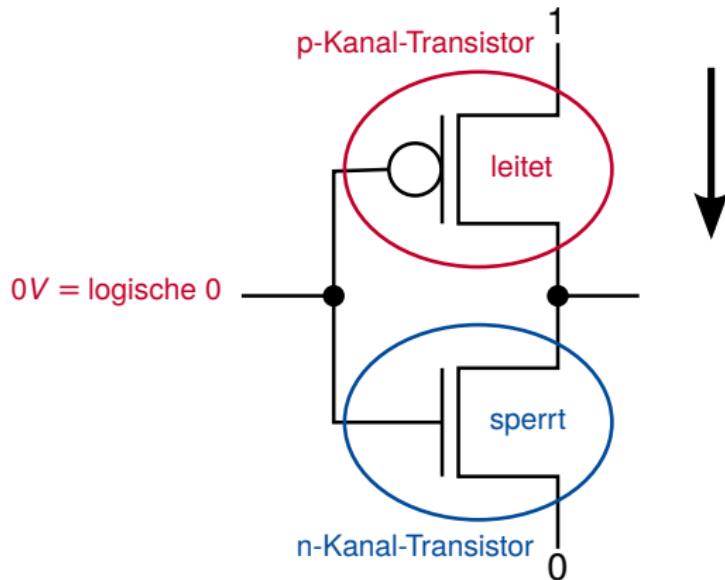
# CMOS-Inverter mit 0 am Gate

- Leitender Pfad zwischen Ausgang und Spannungsversorgung (logische 1).



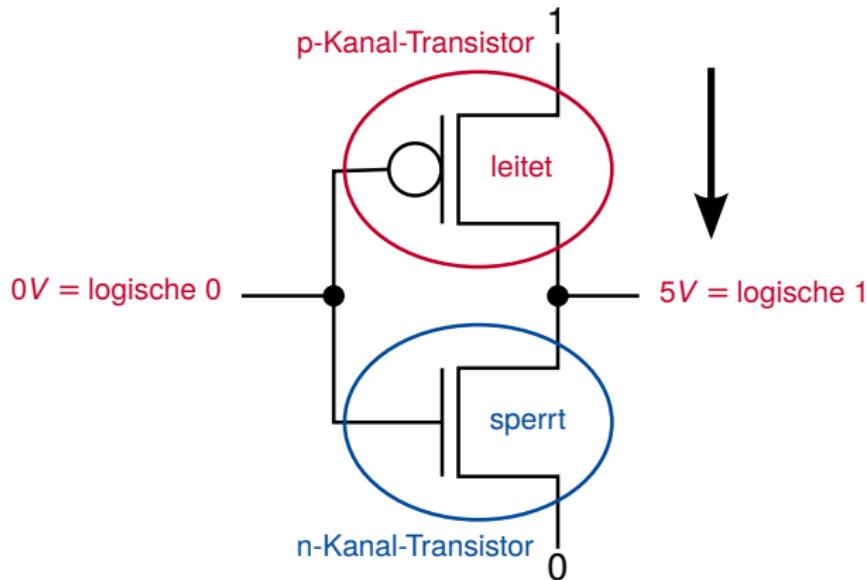
# CMOS-Inverter mit 0 am Gate

- Leitender Pfad zwischen Ausgang und Spannungsversorgung (logische 1).



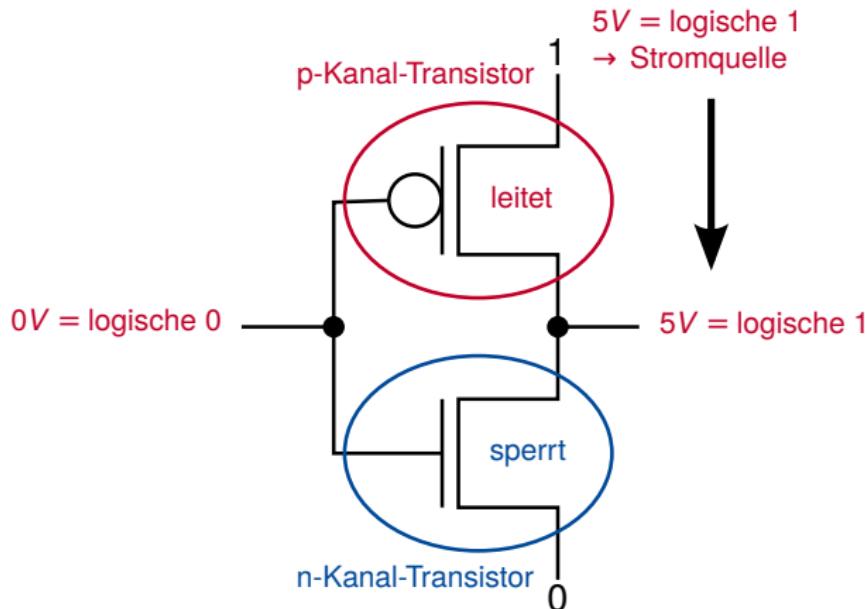
# CMOS-Inverter mit 0 am Gate

- Leitender Pfad zwischen Ausgang und Spannungsversorgung (logische 1).



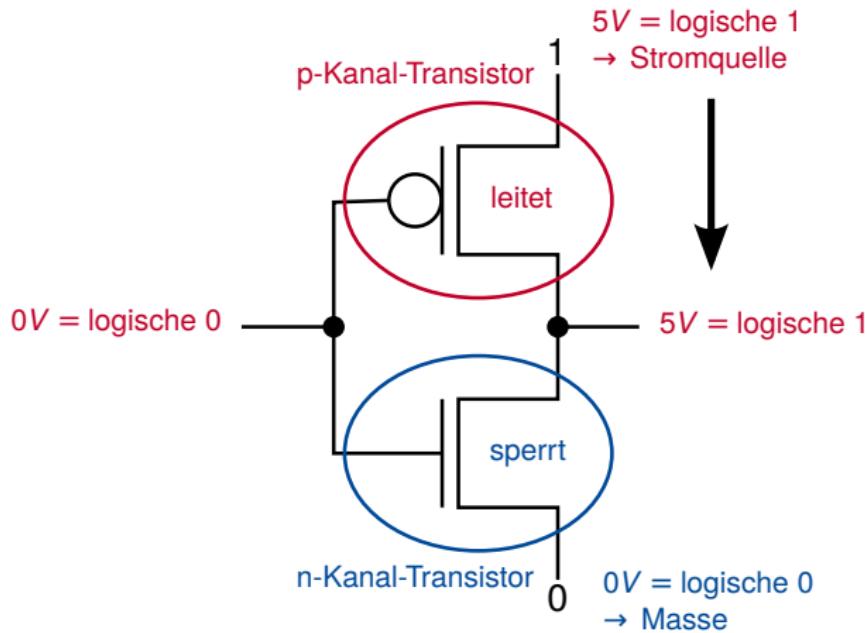
# CMOS-Inverter mit 0 am Gate

- Leitender Pfad zwischen Ausgang und Spannungsversorgung (logische 1).



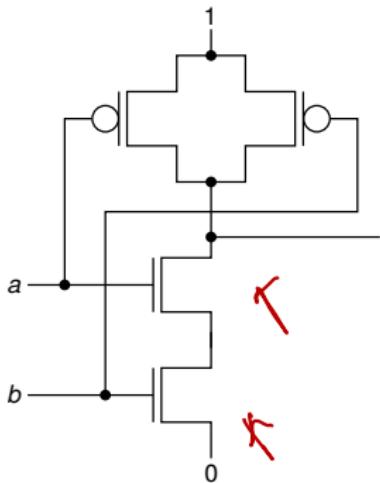
# CMOS-Inverter mit 0 am Gate

- Leitender Pfad zwischen Ausgang und Spannungsversorgung (logische 1).



# SMILE - CMOS

Welche Funktion wird mit Hilfe der p-Kanal und n-Kanal Transistoren realisiert?



Was muss passieren, damit Ausgang = 0 ist?

$$?(a, b)$$

Sonst?

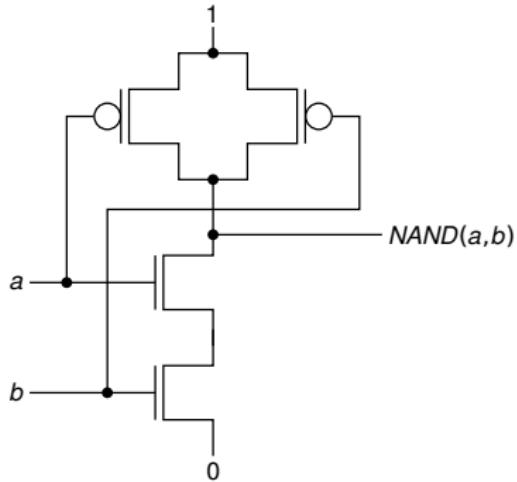
$$a=1$$

$$b=1$$

$a$	$b$	$\overline{ab}$	$\overline{a+b}$
0	0	1	1
0	1	0	0
1	0	0	0
1	1	0	0



# CMOS-NAND-Gatter



## Ausgang ist 0

- ⇒ Es existiert ein leitender Pfad von 0 zum Ausgang
- ⇒ beide n-Kanal-Transistoren leiten
- ⇒  $a = b = 1, a \wedge b = 1$
- ⇒  $\text{NAND}(a,b) = 0$

## Ausgang ist 1

- ⇒ Es existiert ein leitender Pfad von 1 zum Ausgang
- ⇒ einer der p-Kanal-Transistoren leitet
- ⇒  $a = 0$  oder  $b = 0, \neg a \vee \neg b = 1$
- ⇒  $\text{NAND}(a,b) = 1$

# Weitere CMOS-Gatter

---

- Es gibt **keine „direkte“ Implementierung** von AND- und OR-Gattern. Sie werden aus NAND-/NOR-Gattern plus Invertern zusammengesetzt.
- Zu jedem p-Kanal Transistor gibt es stets einen **komplementären n-Kanal-Transistor**, der genau dann sperrt, wenn der erste Transistor leitet und umgekehrt. Dadurch gibt es niemals einen leitenden Pfad von der Stromversorgung zur Masse. Dies reduziert Leistungsverluste.

# Schaltkreise

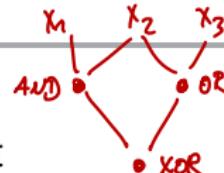
---

- Idee:  
„gerichteter Graph mit einigen zusätzlichen Eigenschaften“

# Gerichtete Graphen (Wiederholung)

- $G = (V, E)$  ist ein **gerichteter Graph**, genau dann, wenn
  - $V$  endliche nichtleere Menge („Knoten“)
  - $E$  endliche Menge („Kanten“)
  - Auf  $E$  sind Abbildungen  $Q, Z : E \rightarrow V$  definiert  
( $Q(e)$  heißt **Quelle**,  $Z(e)$  **Ziel** einer Kante  $e$ )
- Die Abbildung  $\text{indeg} : V \rightarrow \mathbb{N}$ ,  $\text{indeg}(v) = |\{e \mid Z(e) = v\}|$  gibt den **Eingangsgrad** eines Knotens  $v \in V$  an.  
 $\text{indeg}(v_2) = 3$
- Die Abbildung  $\text{outdeg} : V \rightarrow \mathbb{N}$ ,  $\text{outdeg}(v) = |\{e \mid Q(e) = v\}|$  gibt den **Ausgangsgrad** eines Knotens  $v \in V$  an.  
 $\text{outdeg}(v_2) = 2$
- Ein **Pfad** (der Länge  $k$ ) in  $G$  ist eine Folge von  $k$  Kanten  $e_1, e_2, \dots, e_k$  ( $k \geq 0$ ) mit  $Z(e_i) = Q(e_{i+1}) \forall i$  mit  $k - 1 \geq i \geq 1$ .  
 $Q(e_1)$  heißt **Quelle**,  $Z(e_k)$  **Ziel** des Pfades.  
Zyklus :  $v_2 v_2$   
 $v_2 v_3 v_4 v_2$
- Ein **Zyklus** in  $G$  ist ein Pfad der Länge  $\geq 1$  in  $G$ , bei dem Ziel und Quelle identisch sind.
- $G$  heißt **azyklisch**, falls kein Zyklus in  $G$  existiert.
- Die **Graph-Tiefe** eines azyklischen Graphen ist definiert als die Länge des längsten Pfades in  $G$ .

# Modellierung durch Schaltkreise (1/2)



- Eine Zellenbibliothek  $BIB \subset \bigcup_{n \in \mathbb{N}} \mathbb{B}_n$  enthält Basisoperatoren, die den Grundgattern entsprechen.
- Ein 5-Tupel  $SK = (\vec{X}_n, G, typ, IN, \vec{Y}_m)$  heißt Schaltkreis mit  $n$  Eingängen und  $m$  Ausgängen über der Zellenbibliothek  $BIB$  genau dann, wenn
  - $\vec{X}_n = (x_1, \dots, x_n)$  ist eine endliche Folge von Eingängen.
  - $G = (V, E)$  ist ein azyklischer, gerichteter Graph mit  $\underline{\{0, 1\}} \cup \underline{\{x_1, \dots, x_n\}} \subseteq V$ .
  - Die Menge  $I = V \setminus (\{0, 1\} \cup \{x_1, \dots, x_n\})$  heißt Menge der Gatter. Die Abbildung  $typ : I \rightarrow BIB$  ordnet jedem Gatter  $v \in I$  einen Zellentyp  $typ(v) \in BIB$  zu.
  - ...

## Modellierung durch Schaltkreise (2/2)

$E^*$  = beliebig (endlich) viele Elemente von E  
hinter einander

$E^G$  = unendlich viele Elemente

■ ...

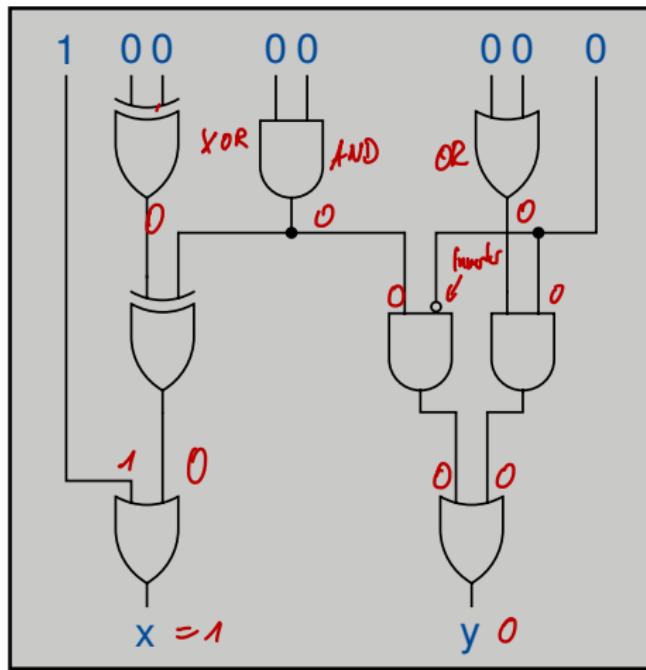
- Für jedes Gatter  $v \in I$  mit  $\text{typ}(v) \in \mathbb{B}_k$  gilt  $\text{indeg}(v) = k$ .
- $\text{indeg}(v) = 0$  für  $v \in \{0, 1\} \cup \{x_1, \dots, x_n\}$ .
- Die Abbildung  $IN : I \rightarrow E^*$  legt für jedes Gatter  $v \in I$  eine Reihenfolge der eingehenden Kanten fest, d.h. falls  $\text{indeg}(v) = k$ , dann ist  $IN(v) = (e_1, \dots, e_k)$  mit  $Z(e_i) = v \forall 1 \leq i \leq k$ .
- Die Folge  $\vec{Y}_m = (y_1, \dots, y_m)$  zeichnet Knoten  $y_i \in V$  als Ausgänge aus.



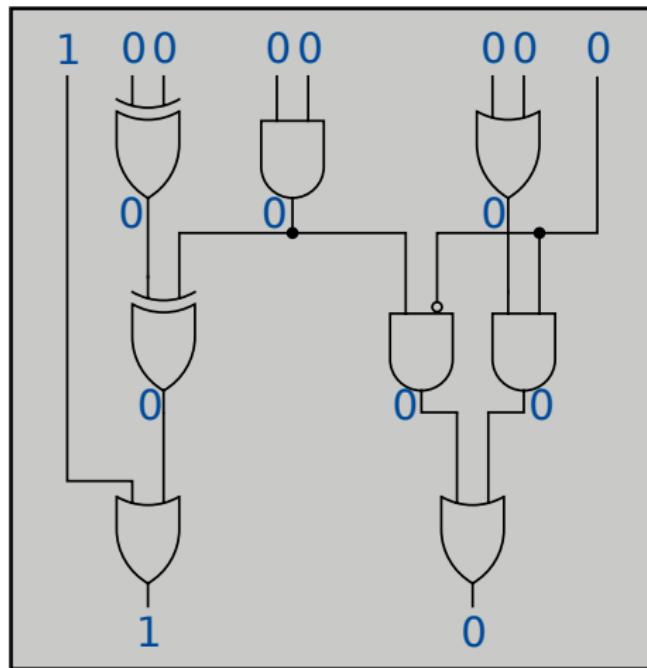
- ...
- Für jedes Gatter  $v \in I$  mit  $typ(v) \in \mathbb{B}_k$  gilt  $indeg(v) = k$ .
- $indeg(v) = 0$  für  $v \in \{0, 1\} \cup \{x_1, \dots, x_n\}$ .
- Die Abbildung  $\text{IN} : I \rightarrow E^*$  legt für jedes Gatter  $v \in I$  eine Reihenfolge der eingehenden Kanten fest, d.h. falls  $indeg(v) = k$ , dann ist  $\text{IN}(v) = (e_1, \dots, e_k)$  mit  $Z(e_i) = v \forall 1 \leq i \leq k$ .
- Die Folge  $\vec{Y}_m = (y_1, \dots, y_m)$  zeichnet Knoten  $y_i \in V$  als Ausgänge aus.

# SMILE - Beispiel für einen mehrstufigen komb. Schaltkreis ( $f \in \mathbb{B}_{8,2}$ )

Welche Werte nehmen x und y gegeben der derzeitigen  
Eingabe an?



# SMILE - Beispiel für einen mehrstufigen komb. Schaltkreis ( $f \in \mathbb{B}_{8,2}$ )



# Formale Semantikdefinition für Schaltkreise (1/2)



- Sei  $SK = (\vec{X}_n, G, typ, IN, \vec{Y}_m)$  ein Schaltkreis über einer Zellenbibliothek  $BIB$ .
- Sei eine Eingangsbelegung  $\underline{\alpha} = (\alpha_1, \dots, \alpha_n) \in \mathbb{B}^n$  gegeben.
- Eine Belegung  $\Phi_{SK, \underline{\alpha}} : V \rightarrow \mathbb{B}$  für alle Knoten  $v \in V$  ist dann gegeben durch die folgenden Definitionen:
  - $\Phi_{SK, \underline{\alpha}}(x_i) = \alpha_i \quad \forall 1 \leq i \leq n$ .
  - $\Phi_{SK, \underline{\alpha}}(0) = 0, \Phi_{SK, \underline{\alpha}}(1) = 1$ .
  - falls  $v \in I$  mit  $typ(v) = g \in \mathbb{B}_k$ ,  $IN(v) = (e_1, \dots, e_k)$ , dann ist  $\Phi_{SK, \underline{\alpha}}(v) = g(\Phi_{SK, \underline{\alpha}}(Q(e_1)), \dots, \Phi_{SK, \underline{\alpha}}(Q(e_k)))$ .
- Warum ist das wohldefiniert?
- Weil  $G$  azyklisch!

wende Gatefunktion auf die Werte der Eingänge an!

## Formale Semantikdefinition für Schaltkreise (2/2)

- $(\Phi_{SK,\alpha}(y_1), \dots, \Phi_{SK,\alpha}(y_m))$  ist dann die unter Eingangsbelegung  $\alpha = (\alpha_1, \dots, \alpha_n)$  berechnete Ausgangsbelegung des Schaltkreises  $SK$ .
- Die Berechnung von  $\Phi_{SK,\alpha}$  bei Eingangsbelegung  $\alpha$  heißt auch **Simulation** von  $SK$  für Belegung  $\alpha$ .
- Die an einem Knoten  $v$  berechnete Boolesche Funktion  $\Psi(v) : \mathbb{B}^n \rightarrow \mathbb{B}$  ist definiert durch

$$\Psi(v)(\alpha) := \Phi_{SK,\alpha}(v)$$

für ein beliebiges  $\alpha \in \mathbb{B}^n$ .

- Die durch den Schaltkreis berechnete Funktion ist

$$f_{SK} := (\Psi(y_1), \dots, \Psi(y_m)).$$

# Standardzellen-Bibliothek

---

- Eine Standardzellen-Bibliothek enthält eine Menge von Gattern und kleinen kombinatorischen Schaltungen (Standardzellen).
  - Z. B. einen 8-Bit-Addierer
- Für jedes Element der Bibliothek werden Parameter wie **Fläche** auf dem Chip, **Schaltgeschwindigkeit**, **Leistungsaufnahme** des Gatters bzw. der Standardzelle abgespeichert.
- Es sind oft z. B. mehrere Inverter unterschiedlicher Größe und Geschwindigkeit vorhanden.

# Kombinatorische Logiksynthese

---

- Allgemeine **kombinatorische Logiksynthese** optimiert mehrere Parameter gleichzeitig.
- Exakte Verfahren existieren, stoßen aber schon für kleinste Schaltkreise an ihre Grenzen.
- In der Praxis werden Heuristiken eingesetzt, die auf Ausschnitten eines großen Schaltkreises **lokale Optimierungen** durchführen.
- Hier beschränken wir uns auf eine wichtige Unterklasse von kombinatorischen Schaltkreisen:  
Die **zweistufige Logik**.  $\leftarrow$  *optimale Schaltungen einer bestimmten Struktur*
- Allgemeinere kombinatorische Schaltkreise betrachten wir später bei der Einführung arithmetischer Schaltkreise.



# Kapitel 3 – Kombinatorische Logik

1. Kombinatorische Schaltkreise
- 2. Normalformen, zweistufige Synthese**
3. Berechnung eines Minimalpolynoms
4. Arithmetische Schaltungen
5. Anwendung: ALU von ReTI

•

Albert-Ludwigs-Universität Freiburg

UNI  
FREIBURG

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur  
WS 2016/17

# Kombinatorische Schaltkreise

$$f(x) = 3x^2 + 4x + 2 \quad \text{reell}$$
$$f(x,y,z) = \underbrace{x'y}_{\text{Monom}} + xy' \quad \text{Boolesche Funktion}$$

## Ziel:

- Wir werden zeigen, dass sich jede boolesche Funktion als ein **Polynom**, also eine **Disjunktion** (ODER-Verknüpfung) von **Monomen**, die ihrerseits **Konjunktionen** (UND-Verknüpfungen) von Eingangsvariablen und negierten Eingangsvariablen sind, darstellen lässt.
- Wir werden für solche Darstellungen **Kostenkriterien** aufstellen und diese optimieren.
- **Monome** und **Polynome** sind spezielle **boolesche Ausdrücke** (s. auch Kap. 7)

# Beispielfunktion

---

- 3 Eingangsvariablen  $x_1, x_2, x_3$
- Wird im Folgenden zur Illustration verwendet.

$x_1$	$x_2$	$x_3$	$f_1$
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	1
1	1	1	1

## Definition

Als **Literal** einer booleschen Funktion  $f \in \mathbb{B}_n$  wird der Ausdruck  $x_i$  oder  $x'_i$  bezeichnet, wobei  $x_i, i \in 1, \dots, n$ , eine Eingangsvariable von  $f$ .

- $x_i$  (auch  $x_i^1$  geschrieben) wird **positives Literal**,  
 $x'_i$  (auch  $x_i^0$  geschrieben) wird **negatives Literal** genannt.
- Literale werden zur kompakten Beschreibung von booleschen Funktionen verwendet. So bezeichnet das Literal  $x_i$  die boolesche Funktion  $g \in \mathbb{B}_n$  mit  
 $g(\alpha_1, \dots, \alpha_n) = 1$  genau dann, wenn  $\alpha_i = 1$ .  
Analog gilt:  $x'_i$  bezeichnet die boolesche Funktion  $h \in \mathbb{B}_n$  mit  
 $h(\alpha_1, \dots, \alpha_n) = 1$  genau dann, wenn  $\alpha_i = 0$ .  
Wir schreiben vereinfachend auch  $g = x_i$  bzw.  $h = x'_i$ .

# Monome

## Definition

*„Ver undung“*

*„leeres“ Monom*

- Ein **Monom** ist eine Konjunktion von Literalen, in der kein Literal mehr als einmal vorkommt und zu keiner Variable sowohl das positive als auch das negative Literal vorkommt. Außerdem ist „1“ ein Monom.
  - $x_1x_2x'_3$  und  $x'_1x_3$  sind Monome,  $x_2x_3x'_3$  ist kein Monom.
- Ein Monom heißt **vollständig** oder **Minterm**, wenn jede Variable entweder als positives oder als negatives Literal vorkommt.
  - Wenn drei Variablen  $x_1, x_2, x_3$  betrachtet werden, ist  $x_1x_2x'_3$  ein Minterm,  $x'_1x_3$  ist kein Minterm.
- Für eine Eingabebelegung  $\alpha \in \mathbb{B}^n$  heißt  $m(010) \cancel{x_1 x_2 x_3}$

$$m(\alpha) = \bigwedge_{i=1}^n x_i^{\alpha_i} \quad (\text{Notation: } x_i^1 := x_i, x_i^0 := x'_i)$$

der zu  $\alpha$  gehörende Minterm.

# Monome als Beschreibung boolescher Funktionen

---

- Beispiel: Das Monom  $m = x_i x'_j$  bezeichnet die boolesche Funktion  $f \in \mathbb{B}_n$  mit  $f(\alpha_1, \dots, \alpha_n) = 1$  genau dann, wenn  $\alpha_i = 1$  und  $\alpha_j = 0$ . Wir schreiben vereinfachend auch  $f = x_i x'_j$ .

# Polynome

---

- Eine Disjunktion von paarweise verschiedenen Monomen heißt **Polynom**. Sind alle Monome des Polynoms vollständig, so heißt das Polynom **vollständig**.

Beispiel: Bei einer booleschen Funktion mit drei Variablen  $x_1, x_2, x_3$  wäre:

- $x'_1x_2 + x'_2x_3$  ein Polynom.
- $x'_1x'_2x_3 + x_1x'_2x_3$  ein vollständiges Polynom.
- Das Polynom  $x'_1x_2 + x'_2x_3$  beschreibt die boolesche Funktion  $f \in \mathbb{B}_3$  mit  $f(\alpha_1, \alpha_2, \alpha_3) = 1$  genau dann, wenn  $\alpha_1 = 0, \alpha_2 = 1$  oder  $\alpha_2 = 0, \alpha_3 = 1$ . Wir schreiben vereinfachend auch  $f = x'_1x_2 + x'_2x_3$ .

# Disjunktive Normalform

$$\text{DNF} = \bar{x}_1 x_2 + \bar{x}_2 x_3$$

$$KNF = (\bar{x}_1 \cdot x_2) \circ (\bar{x}_2 + x_3)$$

- Eine disjunktive Normalform (DNF) von  $f$  ist ein Polynom, das  $f$  beschreibt. Eine kanonische disjunktive Normalform (KDNF) von  $f$  ist ein vollständiges Polynom von  $f$ .
  - $f_1 = x'_1 x'_2 + x'_2 x_3 + x_1 x_2$  ist in DNF, aber nicht in KDNF.

$$f_1 = \bar{x}_1 x_2 + \bar{x}_2$$

$$f_2 = \bar{x}_1 x_2 + \frac{x_1 \bar{x}_2 + \bar{x}_1 \bar{x}_2}{(x_1 + \bar{x}_1) \bar{x}_2}$$
$$\underline{\bar{x}_2}$$

# Bestimmung der kanonischen disjunktiven Normalform

- Für  $f : \mathbb{B}^n \rightarrow \mathbb{B}$  heißt  $ON(f) := \{\alpha \in \mathbb{B}^n \mid f(\alpha) = 1\}$  die Erfüllbarkeitsmenge von  $f$ .

- Die KDNF ist gegeben durch  $f = \sum_{\alpha \in ON(f)} m(\alpha)$

- Die KDNF ist (bis auf Anordnung von Literalen in Mintermen und von Termen im Polynom) eindeutig.

- Beispiel: KDNF für  $f_1 = x'_1 x'_2 + x'_2 x_3 + x_1 x_2$

$$\begin{aligned}f_1 &= m(000) + m(001) \\&\quad + m(101) + m(110) + m(111) \\&= x'_1 x'_2 x'_3 + x'_1 x'_2 x_3 \\&\quad + x_1 x'_2 x_3 + x_1 x_2 x'_3 + x_1 x_2 x_3\end{aligned}$$

$x_1$	$x_2$	$x_3$	$f_1$
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	1
1	1	1	1

- Anmerkung: Analog zur Erfüllbarkeitsmenge ist  $OFF(f) := \{\alpha \in \mathbb{B}^n \mid f(\alpha) = 0\}$  als die Unerfüllbarkeitsmenge definiert.

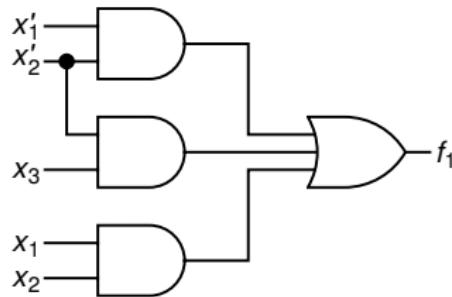
# Realisierungen von DNF

---

- Erste Möglichkeit: Benutze „gewöhnliche“ UND- und ODER-Gatter.
- Zweite Möglichkeit: PLAs
  - Programmierbare logische Felder können nur Funktionen in DNF implementieren.
  - Sie benötigen dafür weniger Transistoren als eine Realisierung mit UND- und ODER-Gattern.

# Realisierung durch Logikgatter

- Bilde erst alle Monome durch UND-Gatter.
- Verbinde dann alle Monome mit ODER-Gattern.
  - Notation: Man verzichtet in der Regel auf die Abbildung von Invertern.



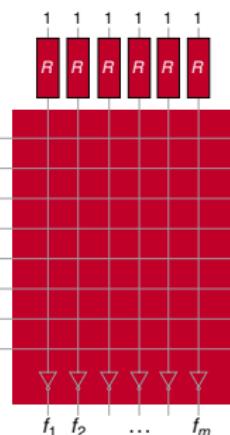
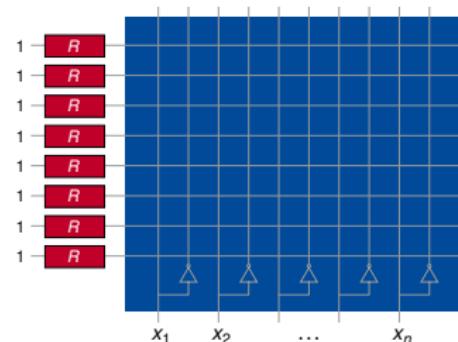
- Die Kosten ergeben sich dann aus allen benötigten UND- und ODER-Gattern.

# Programmierbare logische Felder (PLA)

Zweistufige Darstellung zur Realisierung von booleschen Polynomen.

$$f_i = m_{i1} + m_{i2} + \dots + m_{ik} \text{ mit } m_{iq} \text{ aus } \{m_1, \dots, m_s\}$$

UND-Feld



ODER-Feld

Enthält Monom  $m_j$   $k$  Literale, so werden  $k$  Transistoren in der entsprechenden Zeile des **UND-Feldes** benötigt.

Besteht die Beschreibung von Funktion  $f_t$  aus  $p$  Monomen, so benötigt man  $p$  Transistoren in der entsprechenden Spalte des **ODER-Feldes**.

Fläche:  $\sim (m + 2n) \times (\text{Anzahl der benötigten Monome})$

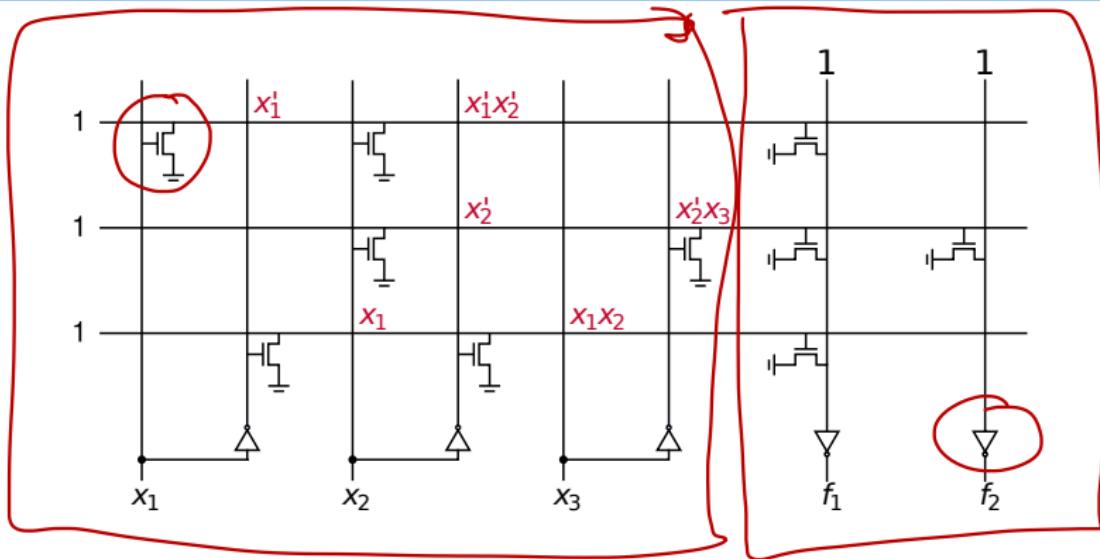
# PLAs: Realisierungsdetails

## Beispiel

$$f_1 = x'_1 x'_2 + x'_2 x_3 + x_1 x_2$$

$$f_2 = x'_2 x_3$$

1-Leitung Ø  $\Leftrightarrow x_1 = 1 \vee x_2 = 1$



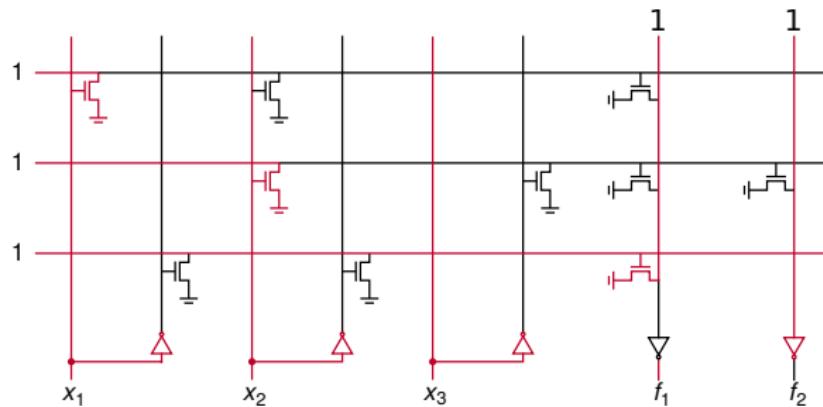
# Konkrete Belegung

## Beispiel

$$f_1 = x'_1 x'_2 + x'_2 x_3 + x_1 x_2$$

$$f_2 = x'_2 x_3$$

Bei Belegung von  $x_1 = 1, x_2 = 1$  und  $x_3 = 1$  liegt folgende Situation vor:



- Sei  $q = q_1 \cdot \dots \cdot q_r$  ein Monom, dann sind die Kosten  $|q|$  von  $q$  gleich der Anzahl der zur Realisierung von  $q$  benötigten Transistoren im PLA, also  $|q| := r$ .
- Seien  $p_1, \dots, p_m$  Polynome, dann bezeichne  $M(p_1, \dots, p_m)$  die Menge der in diesen Polynomen verwendeten Monome.
  - Die primären Kosten  $cost_1(p_1, \dots, p_m)$  einer Menge  $\{p_1, \dots, p_m\}$  von Polynomen sind gleich der Anzahl der benötigten Zeilen im PLA, um  $p_1, \dots, p_m$  zu realisieren, also  $cost_1(p_1, \dots, p_m) = |M(p_1, \dots, p_m)|$ .
  - Die sekundären Kosten  $cost_2(p_1, \dots, p_m)$  einer Menge  $\{p_1, \dots, p_m\}$  von Polynomen sind gleich der Anzahl der benötigten Transistoren im PLA, also
$$cost_2(p_1, \dots, p_m) = \underbrace{\sum_{q \in M(p_1, \dots, p_m)} |q|}_{\text{und-Feld}} + \underbrace{\sum_{i=1, \dots, m} |M(p_i)|}_{\text{oder-Feld}}.$$

# Kombiniertes Kostenmaß

---

- Sei im Folgenden  $\text{cost} = (\text{cost}_1, \text{cost}_2)$  die Kostenfunktion mit der Eigenschaft, dass für zwei Polynommengen  $\{p_1, \dots, p_m\}$  und  $\{p'_1, \dots, p'_m\}$  die Ungleichung

$$\text{cost}(p_1, \dots, p_m) \leq \text{cost}(p'_1, \dots, p'_m)$$

gilt, wenn

- entweder  $\text{cost}_1(p_1, \dots, p_m) < \text{cost}_1(p'_1, \dots, p'_m)$
- oder  $\text{cost}_1(p_1, \dots, p_m) = \text{cost}_1(p'_1, \dots, p'_m)$   
und  $\text{cost}_2(p_1, \dots, p_m) \leq \text{cost}_2(p'_1, \dots, p'_m)$



# SMILE – Kosten der Realisierung

---

Welche der folgenden PLAs ist die kostengünstigste Lösung?

- a. PLA 1 mit 10 Zeilen und 19 Transistoren.
- b. PLA 2 mit 15 Zeilen und 30 Transistoren.
- c. PLA 3 mit 12 Zeilen und 15 Transistoren.
- d. PLA 4 mit 12 Zeilen und 19 Transistoren.

# Das Problem der zweistufigen Logikminimierung

---

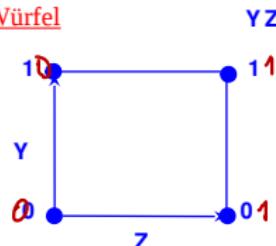
- **Gegeben:** Eine boolesche Funktion  $f = (f_1, \dots, f_m)$  in  $n$  Variablen und  $m$  Ausgängen in Form einer Tabelle der Dimension  $(n+m) \cdot 2^n$  oder einer Menge von  $m$  Polynomen  $\{q_1, \dots, q_m\}$  mit  $f_i = q_i$ .
- **Gesucht:**  $m$  Polynome  $p_1, \dots, p_m$ , so dass  $p_i$  für alle  $i$  der Funktion  $f_i$  entspricht und die Kosten  $cost(p_1, \dots, p_m)$  minimal sind.
- Ab sofort werden nur noch Funktionen mit einem Ausgang betrachtet.

# Veranschaulichung von Monomen und Polynomen

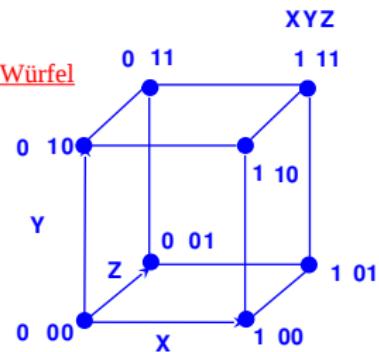
1-dim Würfel



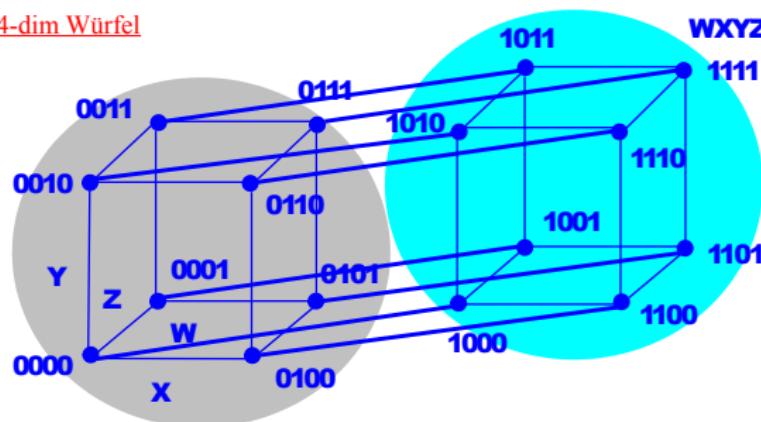
2-dim Würfel



3-dim Würfel



4-dim Würfel

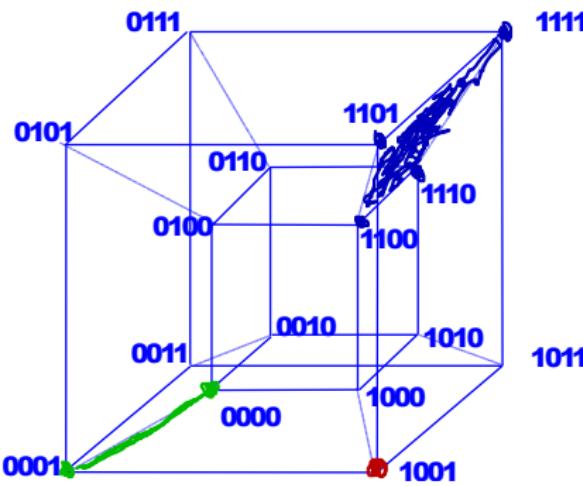


# Veranschaulichung durch Würfel

- Jede boolesche Funktion  $f$  in  $n$  Variablen und einem Ausgang kann über einen  $n$ -dimensionalen Würfel durch Markierung der  $ON(f)$ -Menge spezifiziert werden.

## Beispiel:

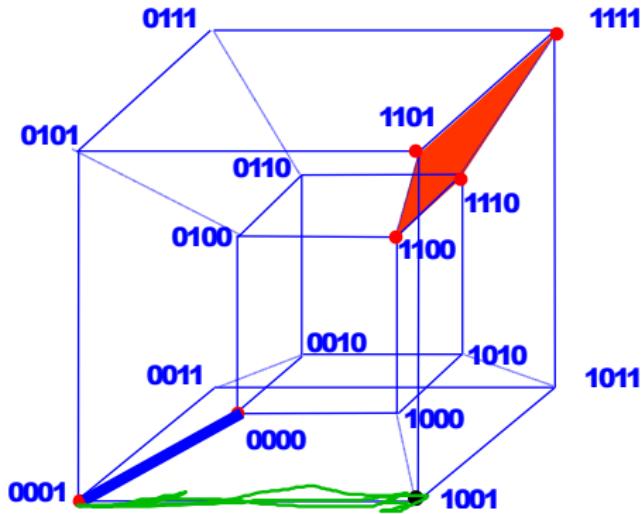
$$f(x_1, x_2, x_3, x_4) = \underline{x_1 x_2} + \underline{x'_1 x'_2 x'_3} + \underline{x_1 x'_2 x'_3 x_4}$$



# Monome und Polynome als Teilwürfel

- Monome der Länge  $k$  entsprechen  $(n - k)$ -dimensionalen Teilwürfeln!
- Ein Polynom entspricht einer Vereinigung von Teilwürfeln.
- **Beispiel:**

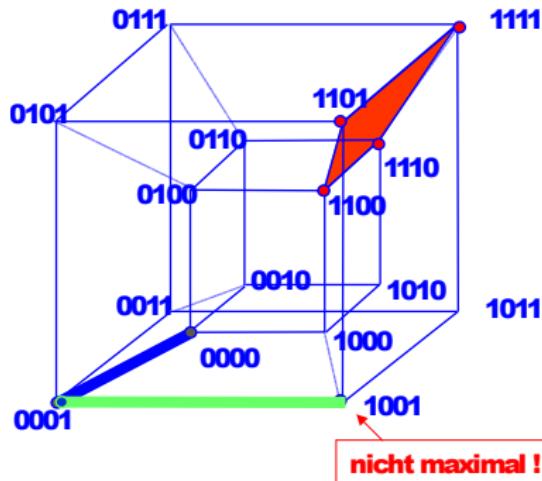
$$\begin{aligned} f(x_1, x_2, x_3, x_4) \\ = & \quad x_1 x_2 + x'_1 x'_2 x'_3 \\ & + x_1 x'_2 x'_3 x_4 \end{aligned}$$



- **Gegeben:** Boolesche Funktion  $f$  in  $n$  Variablen und **einem** Ausgang, in Form eines markierten  $n$ -dimensionalen Würfels.
- **Gesucht:** Eine **minimale Überdeckung** der markierten Knoten durch **maximale Teilwürfel** im  $n$ -dimensionalen Würfel.

Entspricht einer Minimallösung:

$$x_1 x_2 + x'_1 x'_2 x'_3 + x'_2 x'_3 x_4$$



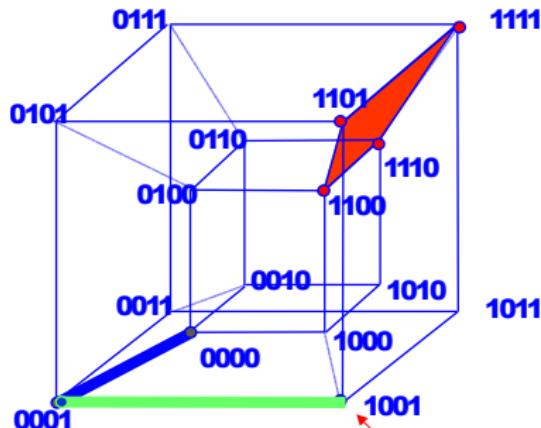
## Zweistufige Logikminimierung als Überdeckungsproblem auf dem Würfel

- **Gegeben:** Boolesche Funktion  $f$  in  $n$  Variablen und **einem** Ausgang, in Form eines markierten  $n$ -dimensionalen Würfels.
- **Gesucht:** Eine **minimale Überdeckung** der markierten Knoten durch **maximale Teilwürfel** im  $n$ -dimensionalen Würfel.



Entspricht einer Minimallösung:

$$x_1 x_2 + x'_1 x'_2 x'_3 + x'_2 x'_3 x_4$$



# SMILE – Minimale Kosten

---

Gegeben ein PLA mit  $m > 1$  Ausgängen und jede Funktion ist durch ein Minimalpolynom, also eine Minimallösung, die die • Funktion überdeckt wie auf der vorherigen Folie, realisiert.  
Sind die gesamten Kosten des PLA folglich auch minimal?

- a. Ja.
- b. Nein.

# Implikanten und Primimplikanten

## Definition

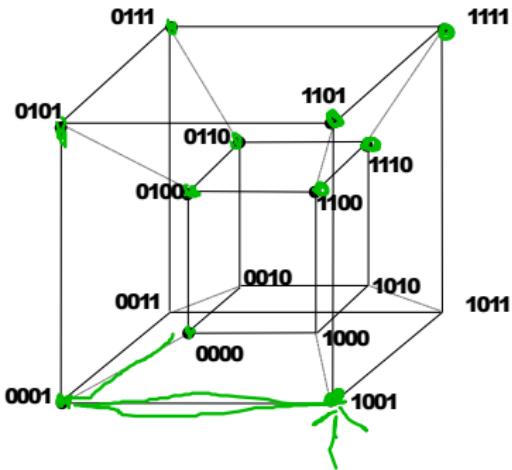
Eine boolesche Funktion  $f \in \mathbb{B}_n$  ist **kleiner gleich** einer anderen booleschen Funktion  $g \in \mathbb{B}_n$  ( $f \leq g$ ), wenn  $\forall \alpha \in \mathbb{B}_n : f(\alpha) \leq g(\alpha)$ . (Das heißt, wenn  $f$  an einer Stelle 1 ist, dann auch  $g$ .)

## Definition

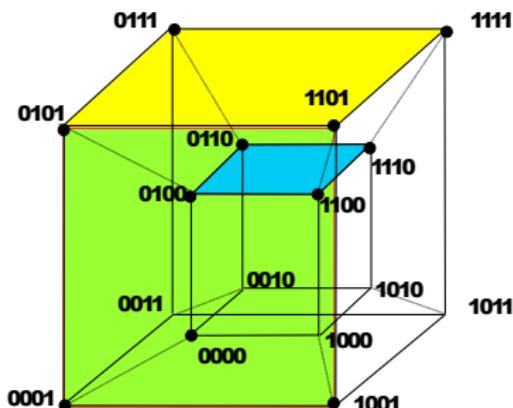
Sei  $f$  eine boolesche Funktion mit einem Ausgang. Ein **Implikant** von  $f$  ist ein Monom  $q$  mit  $q \leq f$ . Ein **Primimplikant** von  $f$  ist ein maximaler Implikant  $q$  von  $f$ , das heißt es gibt keinen Implikanten  $s$  ( $s \neq q$ ) von  $f$  mit  $q \leq s$ .  
Implikanten und Primimplikanten können durch  $n$ -dimensionale Würfel veranschaulicht werden.

# Veranschaulichung durch Würfel

- Ein **Implikant** von  $f$  ist ein Teilwürfel, der nur markierte Knoten enthält.
- Ein **Primimplikant** von  $f$  ist ein maximaler Teilwürfel mit dieser Eigenschaft.



# Illustration für konkrete Funktion



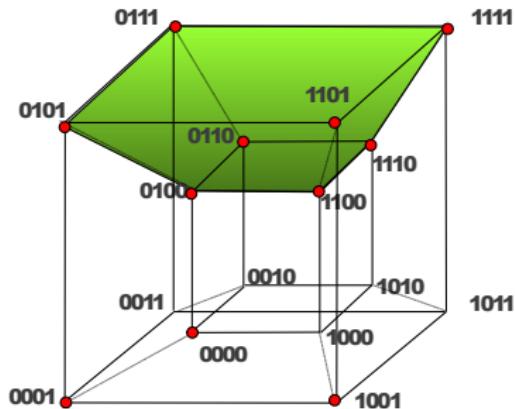
## Implikanten

- alle markierten Knoten
- alle Kanten, deren Ecken alle markiert sind
- alle Flächen, deren Ecken alle markiert sind
- alle 3-dimensionalen Würfel, deren Ecken alle markiert sind

## Allgemein

- Die Implikanten sind die Teilwürfel, deren Ecken alle markiert sind.

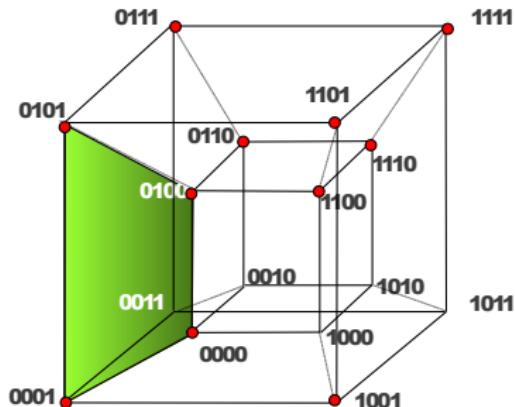
# Bestimmung von Primimplikanten



Es gibt 3 Primimplikanten, der durch unseren Würfel definierten Funktion:

■  $X_2$

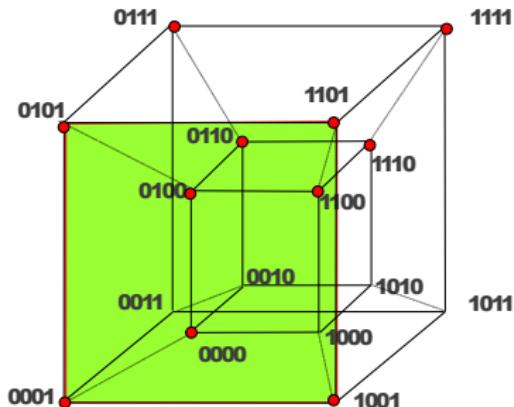
# Bestimmung von Primimplikanten



Es gibt 3 Primimplikanten, die durch unseren Würfel definierten Funktion:

- $x_2$
- $x'_1 x'_3$

# Bestimmung von Primimplikanten



Es gibt 3 Primimplikanten, der durch unseren Würfel definierten Funktion:

- $x_2$
- $x'_1 x'_3$
- $x'_3 x_4$

# Polynome und Implikanten einer Funktion $f$

---

## Lemma

Die Monome eines Polynoms  $p$  von  $f$  sind alle Implikanten von  $f$ .

### Beweis:

- Enthält ein Polynom  $p$  von  $f$  ein Monom  $m$ , welches nicht Implikant von  $f$  ist, so gilt nicht:  $m \leq f$
- Das heißt es gibt eine Belegung  $(\alpha_1, \dots, \alpha_n)$  der Variablen  $(x_1, \dots, x_n)$  mit
  - $f(\alpha_1, \dots, \alpha_n) = 0$
  - $m(\alpha_1, \dots, \alpha_n) = 1$ , also auch  $p(\alpha_1, \dots, \alpha_n) = 1$
- Das heißt Monom  $m$  ist ein Teilwürfel, bei dem eine Ecke nicht markiert ist.

Demnach ist Polynom  $p$  kein Polynom von  $f$ .

⇒ Widerspruch!



# Kapitel 3 – Kombinatorische Logik

1. Kombinatorische Schaltkreise
2. Normalformen, zweistufige Synthese
- 3. Berechnung eines Minimalpolynoms**
4. Arithmetische Schaltungen
5. Anwendung: ALU von ReTI

# Billigste Überdeckung der markierten Ecken

---

Wir suchen ein sogenanntes Minimalpolynom, das heißt ein Polynom mit minimalen Kosten.

## Definition

Ein Minimalpolynom  $p$  einer booleschen Funktion  $f$  ist ein Polynom von  $f$  mit minimalen Kosten, das heißt mit der Eigenschaft  $\text{cost}(p) \leq \text{cost}(p')$  für jedes andere Polynom  $p'$  von  $f$ .

# Quine's Primimplikantensatz

## Satz

Jedes Minimalpolynom  $p$  einer booleschen Funktion  $f$  besteht ausschließlich aus Primimplikanten von  $f$ .

## Beweis:

- Nehme an, dass  $\underline{p}$  einen nicht primen Implikanten  $\underline{m}$  von  $\underline{f}$  enthält.
- $\underline{m}$  wird durch einen Primimplikanten  $\underline{m'}$  von  $\underline{f}$  überdeckt, ist also in  $\underline{m'}$  enthalten.
  - Es gilt demnach  $\underline{\underline{cost(m') < cost(m)}}.$
  - Ersetzt man in  $\underline{p}$  den Implikanten  $\underline{m}$  durch den Primimplikanten  $\underline{m'}$ , so erhält man ein Polynom  $\underline{p'}$ , das ein Polynom von  $\underline{f}$  ist mit  $\underline{\underline{cost(p') < cost(p)}}.$
  - **Widerspruch** dazu, dass  $\underline{p}$  ein Minimalpolynom ist.

# Berechnung von Implikanten

## Lemma 1

Ist  $m$  ein Implikant von  $f$ , so auch  $m \cdot x$  und  $m \cdot x'$  für jede Variable  $x$ , die in  $m$  weder als positives, noch als negatives Literal vorkommt.

### Beweis:

- $m \cdot x$  und  $m \cdot x'$  sind Teilwürfel des Würfels  $m$ .
- Sind also alle Ecken von  $m$  markiert, so auch alle Ecken von  $m \cdot x$  und  $m \cdot x'$ .

## Lemma 2

Sind  $m \cdot x$  und  $m \cdot x'$  Implikanten von  $f$ , so auch  $m$ .

### Beweis:

- $f \geq m \cdot x + m \cdot x' = m \cdot (x + x') = m$

$$\begin{array}{l} f \geq mx \\ f \geq mx' \end{array} \quad \left. \begin{array}{l} f \geq mx + mx' \end{array} \right\}$$

# Charakterisierung von Implikanten

## Satz

Ein Monom  $m$  ist genau dann ein Implikant von  $f$ , wenn entweder

- $m$  ein Minterm von  $f$  ist, oder
- $m \cdot x$  und  $m \cdot x'$  Implikanten von  $f$  sind für eine Variable  $x$ , die nicht in  $m$  vorkommt.

- Äquivalente Schreibweise:

$$\boxed{m \in \text{Implikant}(f) \Leftrightarrow (m \in \text{Minterm}(f)) \vee (m \cdot x, m \cdot x' \in \text{Implikant}(f))}$$

- Beweis folgt unmittelbar aus Lemma 1 und Lemma 2.

# Berechnung eines Minimalpolynoms

- Verfahren von Quine-McCluskey zur Berechnung aller Primimplikanten.
  - Idee: Berechne sogar alle Implikanten. Dann ist klar, welche Primimplikanten sind.
- Verfahren zur Lösung des „Überdeckungsproblems“.
  - Treffe unter den Primimplikanten eine geeignete Auswahl, so dass die Disjunktion der ausgewählten Primimplikanten ein Polynom für  $f$  ist und minimale Kosten hat.



# Verfahren von Quine: Der Algorithmus

$\overline{Y_1} \overline{X_2}$      $X_1 X_2$

Prime implicants function **Quine** ( $f : \mathbb{B}^n \rightarrow \mathbb{B}$ )

**begin**

$L_0 := \underline{\text{Minterm}}(f);$

$i := 0;$

    //  $L_i$  enthält alle Implikanten von  $f$  der Länge  $n - i$ .

$\underline{\text{Prim}}(f) := \emptyset$

**while** ( $L_i \neq \emptyset$ ) **and** ( $i < n$ )

**loop**  $L_{i+1} := \{m \mid m \cdot x \text{ und } m \cdot x' \text{ sind in } L_i \text{ für ein } x\};$  |

$\underline{\text{Prim}}(f) := \text{Prim}(f) \cup$

$\{m' \mid m' \in L_i \text{ und } m' \text{ wird von keinem } q \in L_{i+1} \text{ überdeckt}\};$

$i := i + 1$

**end loop;**

**return**  $\underline{\text{Prim}}(f) \cup L_i;$

**end;**

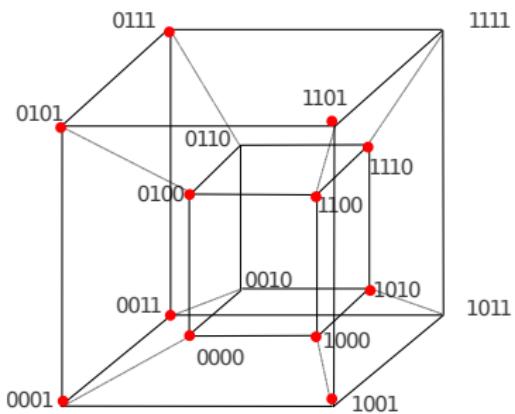
$$\begin{aligned} mx + m\bar{x} &\rightsquigarrow m \\ = m(x + \bar{x}) &= m \end{aligned}$$

# Verbesserung durch Mccluskey

---

- Vergleiche nur **Monome** untereinander
  - die die gleichen Variablen enthalten und
  - bei denen sich die Anzahl der positiven Literale nur um 1 unterscheidet.
- Dies wird erreicht durch:
  - Partitionierung von  $L_i$  in Klassen  $\underline{L_i^M}$ , mit  $M \subseteq \{x_1, \dots, x_n\}$  und  $|M| = n - i$ .
  - $\underline{L_i^M}$  enthält die Implikanten aus  $L_i$ , deren Literale alle aus  $M$  sind.
  - Anordnung der Monome in  $\underline{L_i^M}$  gemäß der Anzahl der positiven Literale.

# Beispiel Quine-McCluskey



$ON(f)$

$L_0^{\{X_1, X_2, X_3, X_4\}}:$

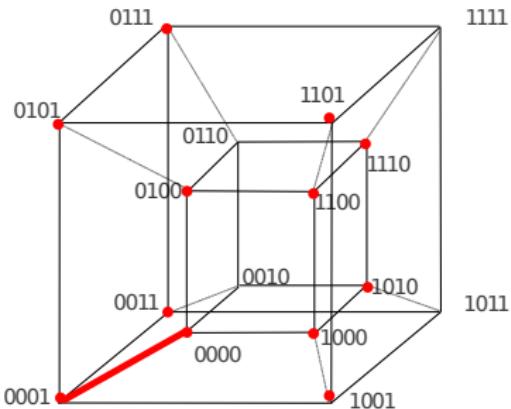
0 0 0 0	$\bar{Y}_1 \bar{Y}_2 \bar{Y}_3 \bar{Y}_4$
0 0 0 1	$\bar{Y}_1 \bar{Y}_2 Y_3 Y_4$
0 1 0 0	$\bar{Y}_1 Y_2 \bar{Y}_3 Y_4$
1 0 0 0	$Y_1 \bar{Y}_2 \bar{Y}_3 \bar{Y}_4$
0 0 1 1	$\bar{Y}_1 Y_2 Y_3 \bar{Y}_4$
0 1 0 1	$\bar{Y}_1 Y_2 Y_3 Y_4$
1 0 0 1	$Y_1 \bar{Y}_2 Y_3 Y_4$
1 0 1 0	$Y_1 Y_2 \bar{Y}_3 Y_4$
1 1 0 0	$Y_1 Y_2 Y_3 \bar{Y}_4$
0 1 1 1	$\bar{Y}_1 \bar{Y}_2 \bar{Y}_3 \bar{Y}_4$
1 1 0 1	$\bar{Y}_1 \bar{Y}_2 Y_3 \bar{Y}_4$
1 1 1 0	$\bar{Y}_1 Y_2 \bar{Y}_3 \bar{Y}_4$

steht für  $X'_1 X'_2 X'_3 X'_4$

steht für  $X_1 X'_2 X'_3 X_4$

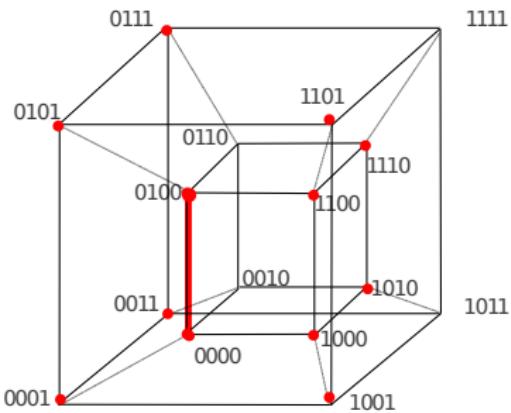
Vergleiche im Folgenden nur Monome aus benachbarten Blöcken!

# Beispiel Quine-McCluskey: Bestimmung von $L_1$ (1/4)



$L_0^{\{x_1, x_2, x_3, x_4\}}:$	$L_1^{\{x_1, x_2, x_3\}}:$
$\begin{array}{r} 0000 \\ \hline 0001 \\ \text{---} \\ 0100 \\ 1000 \\ 0011 \\ 0101 \\ 1001 \\ 0010 \\ 1010 \\ 1100 \\ 1101 \\ 1111 \end{array}$	$\begin{array}{r} 000 - \\ \text{---} \\ 0000 \\ 0001 \\ \text{---} \\ 0100 \\ 1000 \\ 0011 \\ 0101 \\ 1001 \\ 0010 \\ 1010 \\ 1100 \\ 1101 \\ 1110 \end{array}$

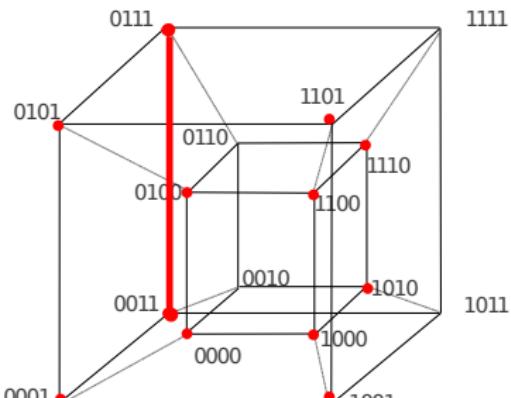
## Beispiel Quine-McCluskey: Bestimmung von $L_1$ (2/4)



$$\begin{array}{r} L_0^{\{x_1, x_2, x_3, x_4\}}: \\ \xrightarrow{\quad} \begin{array}{r} 0000 \\ \hline 0001 \end{array} \\ \xrightarrow{\quad} \begin{array}{r} 0100 \\ \hline 1000 \\ 0011 \\ 0101 \\ 1001 \\ 1010 \\ 1100 \\ 0111 \\ 1101 \\ 1111 \end{array} \end{array}$$

$$\begin{array}{r} L_1^{\{x_1, x_2, x_3\}}: \\ 000 - \\ L_1^{\{x_1, x_3, x_4\}}: \\ 0 - 00 \end{array}$$

# Beispiel Quine-McCluskey: Bestimmung von $L_1$ (3/4)



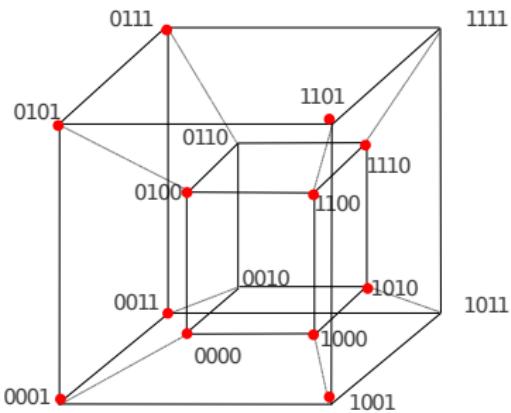
$$L_0^{\{x_1, x_2, x_3, x_4\}}:$$

0 0 0 0	
0 0 0 1	
0 1 0 0	↖
1 0 0 0	
<hr/>	
0 0 1 1	↖
0 1 0 1	
1 0 0 1	
1 0 1 0	
1 1 0 0	
<hr/>	
0 1 1 1	
1 1 0 1	
1 1 1 0	

$$L_1^{\{x_1, x_2, x_3\}}:$$

0 0 0 -	
<hr/>	
0 - 0 0	
<hr/>	
0 - 1 1	

## Beispiel Quine-McCluskey: Bestimmung von $L_1$ (4/4)



$$\begin{array}{r} L_0^{\{x_1, x_2, x_3, x_4\}}: \\ \hline 0000 \\ 0001 \\ 0100 \\ 1000 \\ \hline 0011 \\ 0101 \\ 1001 \\ 1010 \\ \hline 1100 \\ 0111 \\ 1110 \\ 1111 \end{array}$$

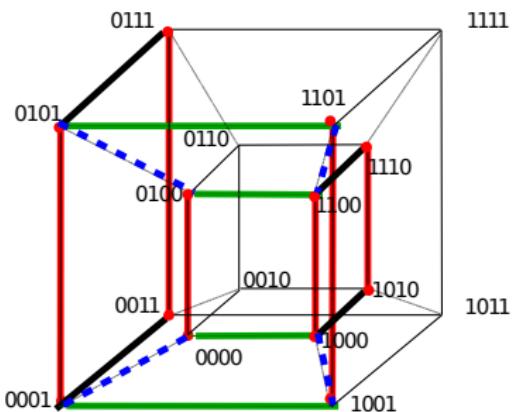
→

$$\begin{array}{r} L_1^{\{x_1, x_2, x_3\}}: \\ \hline 000 - \\ \hline \end{array}$$

$$\begin{array}{r} L_1^{\{x_1, x_3, x_4\}}: \\ \hline 0 - 00 \\ \hline 0 - 11 \end{array}$$

Nicht kürzbar, da nicht  
Ecken der gleichen Kante.  
(Consensus existiert nicht!)

# Beispiel Quine-McCluskey: Alle bestimmten Mengen $L_1$



$$\begin{array}{l} 1+a \rightarrow 0--1 \\ 1+b \text{ geht nicht} \end{array}$$

$$\begin{array}{l} 2+a \rightarrow \text{geht nicht} \\ 2+b \rightarrow 1--0 \end{array}$$

$$L_1^{\{x_1, x_2, x_4\}}:$$

0 0 - 1 · 1
1 0 - 0 2
0 1 - 1 9
1 1 - 0 b

$$L_1^{\{x_1, x_2, x_3\}}:$$

0 0 0 -
0 1 0 -
1 0 0 -
1 1 0 -

$$L_1^{\{x_2, x_3, x_4\}}:$$

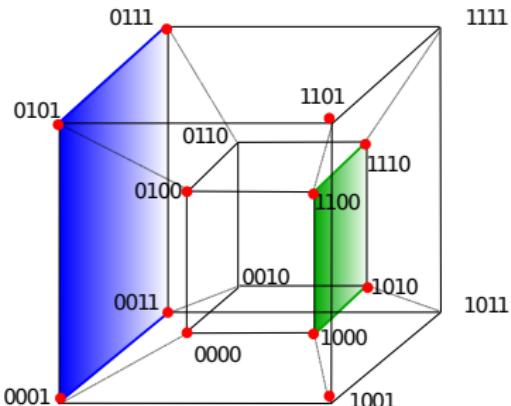
- 0 0 0
- 0 0 1
- 1 0 0
- 1 0 1

$$L_1^{\{x_1, x_3, x_4\}}:$$

0 - 0 0
0 - 0 1
1 - 0 0
0 - 1 1
1 - 0 1
1 - 1 0

Alle Minterme von  $f$  sind Eckpunkte von Kanten,  
die Implikanten sind:  $\text{Prim}(f) = \emptyset$

# Beispiel Quine-McCluskey: Bestimmung von $L_2$ (1/2)



$$L_1^{\{x_1, x_2, x_4\}}:$$

$$\begin{array}{r} \textcolor{blue}{\rightarrow} 00 - 1 \\ \textcolor{green}{\rightarrow} 10 - 0 \\ \hline \textcolor{blue}{\rightarrow} 01 - 1 \\ \textcolor{green}{\rightarrow} 11 - 0 \end{array}$$

$$L_1^{\{x_1, x_2, x_3\}}:$$

$$\begin{array}{r} 000 - \\ 010 - \\ 100 - \\ \hline 110 - \end{array}$$

$$L_1^{\{x_2, x_3, x_4\}}:$$

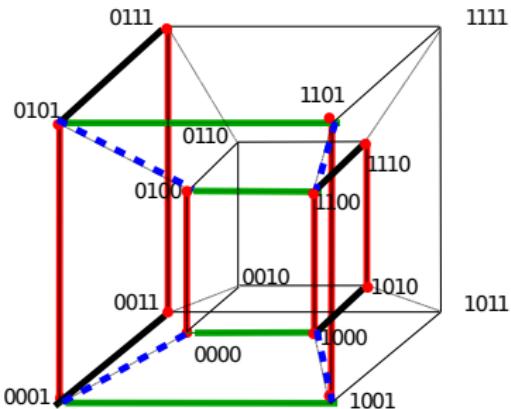
$$\begin{array}{r} -000 \\ -001 \\ -100 \\ \hline -101 \end{array}$$

$$L_1^{\{x_1, x_3, x_4\}}:$$

$$\begin{array}{r} 0-00 \\ 0-01 \\ 1-00 \\ \hline 0-11 \\ 1-01 \\ 1-10 \end{array}$$

Alle Implikanten aus  $L_1^{\{x_1, x_2, x_4\}}$  sind Kanten von Flächen, die Implikanten sind:  $\text{Prim}(f) = \emptyset$

## Beispiel Quine-McCluskey: Bestimmung von $L_2$ (2/2)



$$L_1^{\{x_1, x_2, x_4\}}:$$

$$\begin{array}{r} 00 - 1 \\ 10 - 0 \\ \hline 01 - 1 \\ 11 - 0 \end{array}$$

$$L_1^{\{x_1, x_2, x_3\}}:$$

$$\begin{array}{r} 000 - \\ 010 - \\ \hline 100 - \\ 110 - \end{array}$$

$$L_1^{\{x_2, x_3, x_4\}}:$$

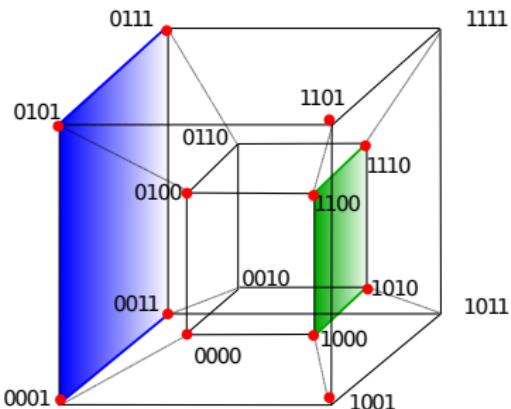
$$\begin{array}{r} -000 \\ -001 \\ -100 \\ -101 \end{array}$$

$$L_1^{\{x_1, x_3, x_4\}}:$$

$$\begin{array}{r} 0-00 \\ 0-01 \\ 1-00 \\ 0-11 \\ 1-01 \\ 1-10 \end{array}$$

Alle Implikanten aus  $L_1 M$  sind Kanten von Flächen, die Implikanten sind:  $Prim(f) = \emptyset$

# Beispiel Quine-McCluskey: Bestimmung von $L_3$ (1/2)



$$L_2^{\{x_1, x_2\}}:$$

$$L_2^{\{x_1, x_3\}}:$$

$$L_2^{\{x_1, x_4\}}:$$

$$L_2^{\{x_2, x_3\}}:$$

$$L_2^{\{x_2, x_4\}}:$$

$$L_2^{\{x_3, x_4\}}:$$

$$\begin{array}{r} \textcolor{blue}{0} \text{ - } 1 \\ \textcolor{green}{1} \text{ - } 0 \end{array} \quad \begin{array}{c} \overline{x_1} \quad x_4 \\ x_1 \quad \overline{x_4} \end{array}$$

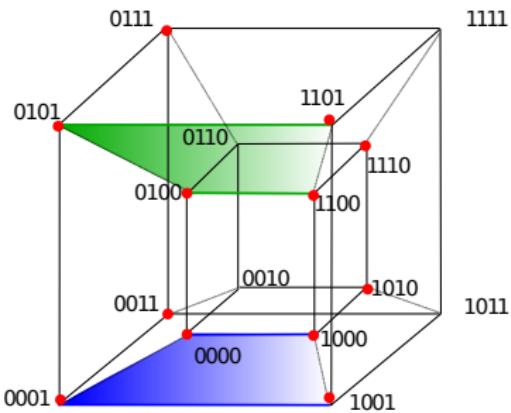
$$\begin{array}{r} \text{ - } 0 \\ \text{ - } 1 \end{array} \quad \begin{array}{c} \text{ - } 0 \\ \text{ - } 1 \end{array}$$

$$\begin{array}{r} \text{ - } 0 \\ \text{ - } 0 \end{array} \quad \begin{array}{c} \text{ - } 0 \\ \text{ - } 1 \end{array}$$

$$\begin{array}{c} \text{ - } 0 \\ \text{ - } 1 \end{array} \quad \begin{array}{c} \text{ - } 0 \\ \overline{x_3} \end{array}$$

Die markierten Implikanten-Flächen sind nicht Rand eines 3-dim. Implikanten. Sie sind also prim!  $\Rightarrow \text{Prim}(f) = \{x'_1x_4, x_1x'_4\}$

## Beispiel Quine-McCluskey: Bestimmung von $L_3$ (2/2)



$L_2^{\{x_1, x_2\}}:$

$$\begin{array}{r} 0 - 0 - \\ 1 - 0 - \end{array}$$

$L_2^{\{x_1, x_3\}}:$

$$\begin{array}{r} 0 - 0 - \\ 1 - 0 - \end{array} \quad \text{--- } O \text{ ---}$$

$L_2^{\{x_1, x_4\}}:$

$$\begin{array}{r} 0 - - 1 \\ 1 - - 0 \end{array}$$

*keine Vereinfachung möglich  $\rightarrow$  Prim impl.*

$L_2^{\{x_2, x_3\}}:$

$$\begin{array}{r} - 0 0 - \\ - 1 0 - \end{array}$$

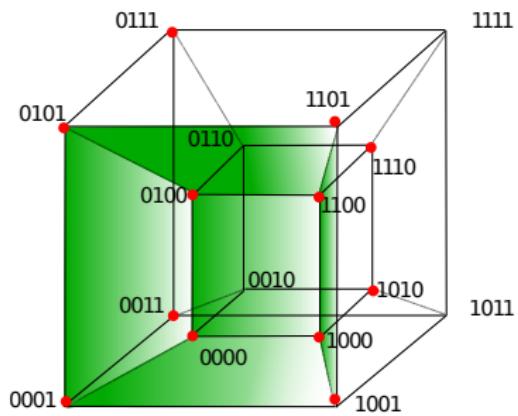
$L_2^{\{x_2, x_4\}}:$

$$\begin{array}{r} - - 0 0 \\ - - 0 1 \end{array} \quad \text{--- } O \text{ ---}$$

$\overline{x_1}x_4, \overline{x_1}x_4$

Die markierten Implikanten-Flächen sind Rand eines 3-dimensionalen Implikanten. Sie sind also nicht prim!  $\Rightarrow \text{Prim}(f) = \{x'_1x_4, x_1x'_4\}$

# Beispiel Quine-McCluskey: Ende



$$L_3^{\{x_1\}}:$$

$$L_3^{\{x_2\}}:$$

$$L_3^{\{x_3\}}:$$

$$L_3^{\{x_4\}}.$$

*-- 0 -- Primimplikant  $\overrightarrow{x_3}$*

$$\text{Prim}(f) = \{x'_1 x_4, x_1 x'_4\}$$

$$\Rightarrow \text{Prim}(f) \in \{x'_1 x_4, x_1 x'_4, x'_3\}$$

$$p_{complete}(f) = x'_1 x_4 + x_1 x'_4 + x'_3$$

# Korrektheit von Quine-McCluskey (1/2)

---

Prime implicants function **Quine** ( $f : \mathbb{B}^n \rightarrow \mathbb{B}$ )

**begin**

$L_0 := \text{Minterm}(f);$

$i := 0;$

  //  $L_i$  enthält alle Implikanten von  $f$  der Länge  $n - i$ .

$\text{Prim}(f) := \emptyset$

**while** ( $L_i \neq \emptyset$ ) **and** ( $i < n$ )

**loop**  $L_{i+1} := \{m \mid m \cdot x$  und  $m \cdot x'$  sind in  $L_i$  für ein  $x\}$ ;

$\text{Prim}(f) := \text{Prim}(f) \cup$

$\{m' \mid m' \in L_i$  und  $m'$  wird von keinem  $q \in L_{i+1}$  überdeckt};

$i := i + 1$

**end loop;**

**return**  $\text{Prim}(f) \cup L_i;$

**end;**

# Korrektheit von Quine-McCluskey (2/2)

## Satz

Für alle  $i = 0, 1, \dots, n$  gilt:

- $L_i$  enthält nur Monome mit  $n - i$  Literalen.
- $L_i$  enthält genau die Implikanten von  $f$  mit  $n - i$  Literalen.
- Nach Iteration  $i$  enthält  $\text{Prim}(f)$  genau die Primimplikanten von  $f$  mit mindestens  $n - i$  Literalen.

## Beweis:

Induktion über  $i$ :

- Abbruchbedingung ( $L_i = \emptyset$ ) oder ( $i = n$ ):
- $L_i = \emptyset$  bedeutet, dass keine Implikanten bei der "Partnersuche" entstanden sind, d.h.  $L_{i-1}$  ist vollständig in  $\text{Prim}(f)$  aufgegangen.
- $i = n$  bedeutet, dass  $L_n$  berechnet wurde, es gilt dann  $L_n = \emptyset$  oder  $L_n = \{1\}$ , letzteres bedeutet  $f$  ist die Eins-Funktion und  $\text{Prim}(f) = \{1\}$ .

# Kosten des Verfahrens

---

## Lemma

Es gibt  $\underline{3^n}$  verschiedene Monome in  $n$  Variablen.

### Beweis:

Für jedes Monom  $m$  und jede der  $n$  Variablen  $x$  liegt genau eine der drei folgenden Situationen vor:

- $m$  enthält weder das positive noch das negative Literal von  $x$ .
- $m$  enthält das positive Literal  $x$ .
- $m$  enthält das negative Literal  $x'$ .

Jedes Monom ist durch diese Beschreibung auch eindeutig bestimmt.

# Komplexität des Verfahrens von Quine-McCluskey

## Satz

Die Laufzeit des Verfahrens liegt in  $O(n^2 \cdot 3^n)$  beziehungsweise in  $O(\log^2(N) \cdot N^{\log(3)})$ , wobei  $N = 2^n$  die Größe der Funktionstabelle ist.

$$n = \log_2 N$$

$$3^{\log N}$$

### Beweisidee:

Jedes der  $3^n$  Monome wird im Verlauf des Verfahrens mit höchstens  $n$  anderen Monomen verglichen.

- Gegeben sei ein Monom  $\underline{mx}$ . Die Erzeugung von  $\underline{mx'}$  und die Suche nach  $\underline{mx'}$  in  $L_i$  ist bei Verwendung geeigneter Datenstrukturen in  $O(n)$  durchführbar.

$O(n^2 \cdot 3^n) = O(\log^2(N) \cdot N^{\log(3)})$  durch Nachrechnen:

$$3^n = (2^{\log(3)})^n = (2^n)^{\log(3)} = N^{\log(3)}$$

# Das Matrix-Überdeckungsproblem

---

- Wir haben nun durch das Verfahren von Quine-McCluskey alle Primimplikanten von  $f$  bestimmt.
- Die Disjunktion aller Primimplikanten ist ein Polynom, das  $f$  implementiert. Es ist aber im Allgemeinen kein Minimalpolynom von  $f$ .
- Für das Minimalpolynom benötigen wir eine kostenminimale Teilmenge  $M$  von  $\text{Prim}(f)$ , so dass die Monome von  $M f$  überdecken.
- Diese Art von Problemen wird **Matrix-Überdeckungsproblem** genannt.

# Das Matrix-Überdeckungsproblem



- Wir haben nun durch das Verfahren von Quine-McCluskey alle Primimplikanten von  $f$  bestimmt.
- Die Disjunktion aller Primimplikanten ist ein Polynom, das  $f$  implementiert. Es ist aber im Allgemeinen kein Minimalpolynom von  $f$ .
- Für das Minimalpolynom benötigen wir eine kostenminimale Teilmenge  $M$  von  $\text{Prim}(f)$ , so dass die Monome von  $M$   $f$  überdecken.
- Diese Art von Problemen wird **Matrix-Überdeckungsproblem** genannt.

# SMILE - Das Matrix-Überdeckungsproblem: Einfaches Beispiel

---

- Für eine Expedition wird ein Fahrer, ein Messtechniker und ein Kameramann benötigt. Es stehen fünf Kandidaten mit unterschiedlichen Fähigkeiten und Gehaltsvorstellungen zur Auswahl. Welches ist das kostengünstigste Team?

Kandidat	Fahrer?	Messtechniker?	Kameramann?	Gehalt
Alice	Ja	Nein	Ja	4000
Dilbert	Ja	Ja	Nein	2000
Dogbert	Ja	Ja	Ja	5000
Ted	Nein	Nein	Ja	1000
Wally	Nein	Ja	Ja	1500

# Primimplikantentafel

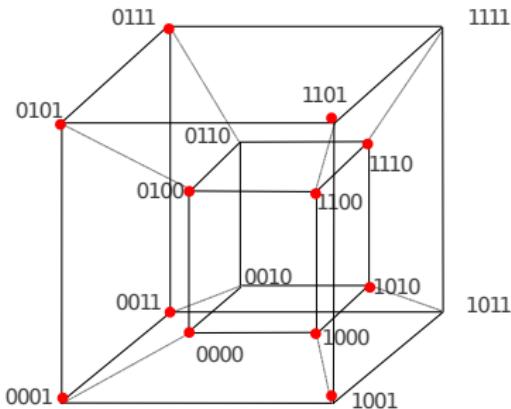
---

- Definiere eine boolesche Matrix  $PIT(f)$ , die Primimplikantentafel von  $f$ :
  - Die Zeilen entsprechen eindeutig den Primimplikanten von  $f$ .
  - Die Spalten entsprechen eindeutig den Mintermen von  $f$ .
  - Sei  $min(\alpha)$  ein beliebiger Minterm von  $f$ .  
Dann gilt für Primimplikant  $m$ :  $PIT(f)[m, min(\alpha)] = 1 \Leftrightarrow m(\alpha) = 1$ .
- Der Eintrag an der Stelle  $[m, min(\alpha)]$  ist also genau dann 1, wenn  $min(\alpha)$  eine Ecke des Würfels  $m$  beschreibt.

## Gesucht:

Eine kostenminimale Teilmenge  $M$  von  $Prim(f)$ , so dass jede Spalte von  $PIT(f)$  überdeckt ist,  
d.h.  $\forall \alpha \in ON(f)$     $\exists m \in M$  mit  $PIT(f)[m, min(\alpha)] = 1$ .

# Primimplikantentafel: Beispiel (1/2)



$$\text{Prim}(f) = \{x'_1 x_4, x_1 x'_4, x'_3\}$$

Primimplikantentafel PIT(f):

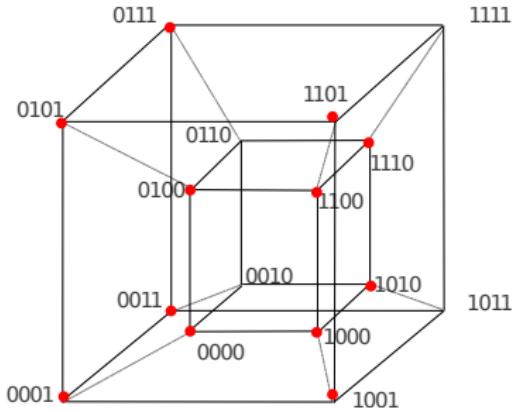
	0	1	3	4	5	7	8	9	10	12	13	14
→	0000	1	1	1	1	1	1	1	1	1	1	1
→	x'_1 x <sub>4</sub>		1									
→	x <sub>1</sub> x'_4			1	1							
→	x'_3	1	1			1	1	1	1	1	1	1

Annotations: A red circle highlights the cell at index 10 (row x'\_1 x<sub>4</sub>, column 1). Another red circle highlights the cell at index 14 (row x'\_3, column 14). A red arrow points to the row x'\_1 x<sub>4</sub>. A red arrow points to the row x<sub>1</sub> x'\_4. A red arrow points to the row x'\_3. A red circle highlights the cell at index 1 (row 0000, column 1).

## Primimplikantentafel: Beispiel (2/2)

### Gesucht:

Eine kostenminimale Teilmenge  $M$  von  $\text{Prim}(f)$ , so dass jede Spalte von  $\text{PIT}(f)$  überdeckt ist,  
d.h.  $\forall \alpha \in \text{ON}(f) \quad \exists m \in M \text{ mit } \text{PIT}(f)[m, \min(\alpha)] = 1$ .



$$\text{Prim}(f) = \{x'_1 x_4, x_1 x'_4, x'_3\}$$

Primimplikantentafel  $\text{PIT}(f)$ :

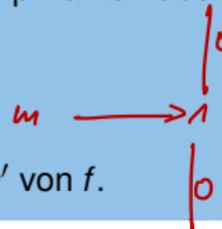
	0	1	3	4	5	7	8	9	10	12	13	14
$x'_1 x_4$	1		1		1							
$x_1 x'_4$							1		1	1		1
$x'_3$	1	1			1	1		1		1	1	

# Erste Reduktionsregel - Wesentlicher Implikant

## Definition

Ein Primimplikant  $m$  von  $f$  heißt **wesentlich**, wenn es einen Minterm  $\min(\alpha)$  von  $f$  gibt, der nur von diesem Primimplikanten überdeckt wird, also:

- $\underline{PIT(f)[m, \min(\alpha)] = 1}$
- $PIT(f)[m', \min(\alpha)] = 0$  für jeden anderen Primimplikanten  $m'$  von  $f$ .

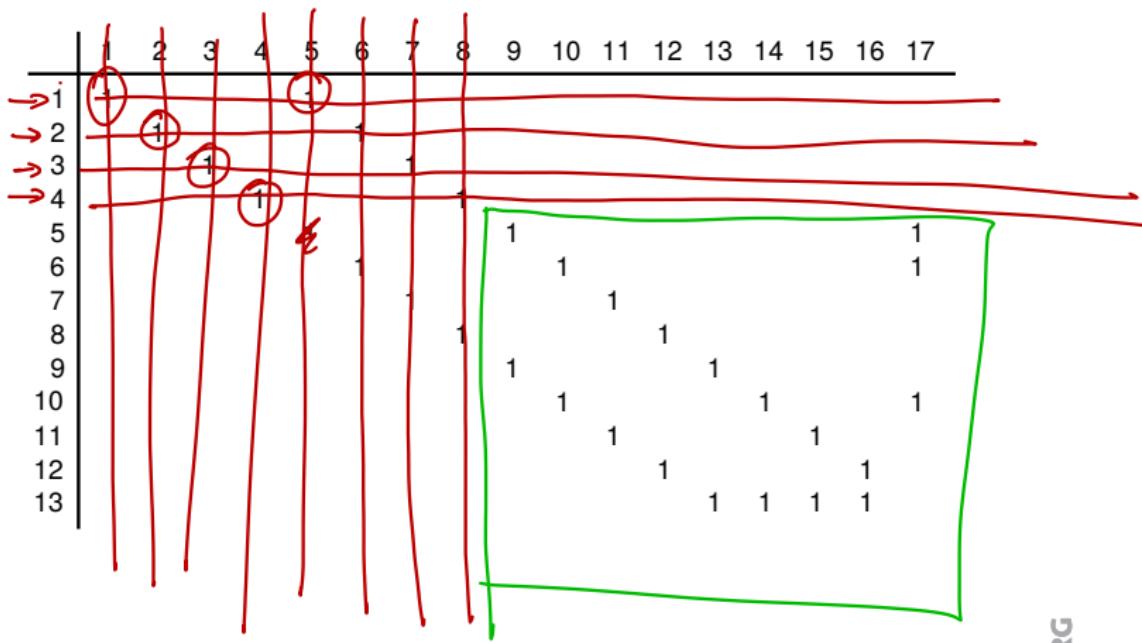


## Lemma

Jedes Minimalpolynom von  $f$  enthält alle wesentlichen Primimplikanten von  $f$ .

**1. Reduktionsregel:** Entferne aus der Primimplikantentafel  $PIT(f)$  alle wesentlichen Primimplikanten und alle Minterme, die von diesen überdeckt werden.

# Erste Reduktionsregel: Beispiel (1/2)



## Erste Reduktionsregel: Beispiel (2/2)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
wesentlich	1																
1		1															
2			1														
3				1													
4					1												
5						1											
6							1										
7								1									
8									1								
9										1							
10											1						
11												1					
12													1				
13														1			

# Nach Anwendung der 1. Reduktionsregel

---

	9	10	11	12	13	14	15	16	17
5	1								1
6		1							1
7			1						
8				1					
9	1				1				
10		1				1			1
11			1				1		
12				1				1	
13					1	1	1	1	

Die Matrix enthält keine wesentlichen Zeilen mehr!

# Zweite Reduktionsregel - Spaltendominanz

## Definition

Sei  $A$  eine boolesche Matrix. Spalte  $j$  von  $A$  **dominiert** Spalte  $i$  von  $A$ , wenn für jede Zeile  $k$  gilt:  $A[k, i] \leq A[k, j]$ .

- Nutzen für unser Problem: Dominiert ein Minterm  $w'$  von  $f$  einen anderen Minterm  $w$  von  $f$ , so braucht man  $w'$  nicht weiter zu betrachten, da  $w'$  auf jeden Fall überdeckt werden muss und hierdurch auch Minterm  $w'$  überdeckt wird.
- Jeder in  $PIT(f)$  vorhandene Primimplikant  $p$ , der  $w$  überdeckt, überdeckt auch  $w'$ .

**2. Reduktionsregel:** Entferne aus der Primimplikantentafel  $PIT(f)$  alle Minterme, die einen anderen Minterm in  $PIT(f)$  dominieren.

# Zweite Reduktionsregel: Beispiel

	9	10	11	12	13	14	15	16	17
5	1								
6		1							
7			1						
8				1					
9	1				1				
10		1				1			
11			1				1		
12				1				1	
13					1	1	1	1	

Spalte 17 dominiert Spalte 10  
⇒ Spalte 17 kann gelöscht werden!

## Dritte Reduktionsregel - Zeilendominanz

Kosten

3	$\alpha$	0	1	1	0
2	$\beta$	0	1	0	0
1	$\gamma$	0	0	1	0

### Definition

Sei  $A$  eine boolesche Matrix. Zeile  $i$  von  $A$  dominiert Zeile  $j$  von  $A$ , wenn für jede Spalte  $k$  gilt:  $A[i, k] \geq A[j, k]$ .

- Nutzen für unser Problem: Dominiert ein Primimplikant  $m$  einen Primimplikanten  $m'$ , so braucht man  $m'$  nicht weiter zu betrachten, wenn  $cost(m') \geq cost(m)$  gilt.
- Der Primimplikant  $m$  überdeckt jeden noch nicht überdeckten Minterm von  $f$ , der von  $m'$  überdeckt wird, obwohl er nicht teurer ist.

**3. Reduktionsregel:** Entferne aus der Primimplikantentafel  $PIT(f)$  alle Primimplikanten, die durch einen anderen, nicht teureren Primimplikanten dominiert werden.

# Dritte Reduktionsregel: Beispiel

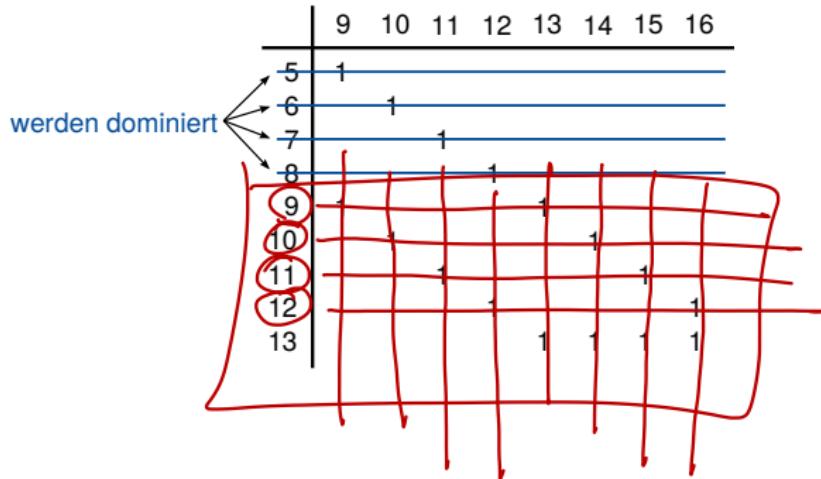
---

Nehme an, dass die Zeilen 5 bis 12 gleiche Kosten haben.

	9	10	11	12	13	14	15	16
5	1							
6		1						
7			1					
8				1				
9	1				1			
10		1				1		
11			1				1	
12				1				1
13					1	1	1	1

# Dritte Reduktionsregel: Beispiel

Nehme an, dass die Zeilen 5 bis 12 gleiche Kosten haben.



# Nach Anwendung der 3. Reduktionsregel

---

	9	10	11	12	13	14	15	16
9	1				1			
10		1				1		
11			1				1	
12				1				1
13					1	1	1	1

- Offensichtlich kann nun wieder die **erste Reduktionsregel** angewendet werden, da die Zeilen 9, 10, 11, 12 wesentlich sind.
  - Die resultierende Matrix ist leer.
  - Das gefundene Minimalpolynom ist:  
 $1 + 2 + 3 + 4 + 9 + 10 + 11 + 12$

# Nach Anwendung der 3. Reduktionsregel



	9	10	11	12	13	14	15	16
9	1				1			
10		1				1		
11			1				1	
12				1				1
13					1	1	1	1

- Offensichtlich kann nun wieder die **erste Reduktionsregel** angewendet werden, da die Zeilen 9, 10, 11, 12 wesentlich sind.
  - Die resultierende Matrix ist leer.
  - Das gefundene Minimalpolynom ist:  
 $1 + 2 + 3 + 4 + 9 + 10 + 11 + 12$

# SMILE - Ein weiteres Beispiel

Welche Reduktionsregel(n) können in dem Beispiel angewendet werden?

$$\text{Prim}(f) = \{\{7, 5\}, \{5, 13\}, \{13, 9\}, \{9, 11\}, \{11, 3\}, \{3, 7\}\}$$

Primimplikantentafel  $\text{PIT}(f)$ :

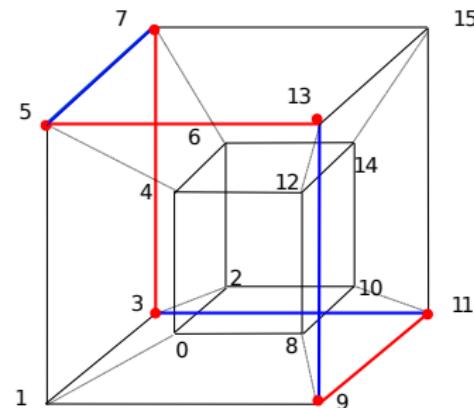
	3	5	7	9	11	13
$\{7, 5\}$		1	1			
$\{5, 13\}$		1				1
$\{13, 9\}$				1		1
$\{9, 11\}$				1	1	
$\{11, 3\}$						1
$\{3, 7\}$				1		

Lösungsmöglichkeit: Fallunterscheidung:  
branch & bound

# SMILE - Ein weiteres Beispiel

Welche Reduktionsregel(n) können in dem Beispiel angewendet werden?

$$\text{Prim}(f) = \{\{7, 5\}, \{5, 13\}, \{13, 9\}, \{9, 11\}, \{11, 3\}, \{3, 7\}\}$$



Primimplikantentafel  $\text{PIT}(f)$ :

	3	5	7	9	11	13
{7, 5}		1	1			
{5, 13}		1				1
{13, 9}				1		1
{9, 11}				1	1	
{11, 3}	1				1	
{3, 7}	1			1		

Kein Primimplikant ist wesentlich!

$$\text{Prim}(f) = \{ \{7, 5\}, \{5, 13\}, \{13, 9\}, \{9, 11\}, \{11, 3\}, \{3, 7\} \}$$

# Zyklische Überdeckungsprobleme

## Definition

Eine Primimplikantentafel heißt **reduziert**, wenn keine der drei Reduktionsregeln anwendbar ist.

- Ist eine reduzierte Tafel nicht-leer, spricht man von einem **zyklischen Überdeckungsproblem**.
- In der Praxis werden solche Probleme heuristisch gelöst. Es gibt auch exakte Methoden (Petrick, Branch-and-Bound).

Primimplikantentafel  $PIT(f)$ :

	3	5	7	9	11	13
{7, 5}		1	1			
{5, 13}		1				1
{13, 9}				1		1
{9, 11}				1	1	
{11, 3}	1				1	
{3, 7}	1			1		

# Petrick's Methode

## Verfahren:

- Übersetze die PIT in ein Produkt von Summen, d.h. in ein (OR, AND)-Polynom, das alle Möglichkeiten der Überdeckung enthält.

	3	5	7	9	11	13
a : {7, 5}		1	1			
b : {5, 13}		1				1
c : {13, 9}				1		1
d : {9, 11}				1	1	
e : {11, 3}			1			1
f : {3, 7}		1		1		

- Multipliziere das (OR, AND)-Polynom aus, so dass ein (AND-OR)-Polynom entsteht.

wird übersetzt in

$$\begin{aligned} & (e+f) \cdot (a+b) \cdot (a+f) \cdot (c+d) \cdot (d+e) \cdot (b+c) \\ & - (ea+eb+fa+fb) \cdot (ac+ad+fc+fd) \\ & \cdot (db+dc+eb+ec) \end{aligned}$$

⋮

$$= \underline{\underline{ace + acde + abcde + abcd + \dots + bdf}}$$

↑

Bei gleichen Kosten für alle Pls sind ace und bdf minimal.

1. Wende alle möglichen Reduktionsregeln an.
  2. Ist die Matrix  $A$  leer, ist man fertig.
  3. Sonst wähle die Zeile  $i$ , die die meisten Spalten überdeckt.  
Lösche diese Zeile und alle von ihr überdeckten Spalten  
und gehe zu 1.
- Dieser Algorithmus liefert nicht immer die optimale Lösung!
- Hinweis: Bei der Ausgangs-Matrix aus unserem Beispiel überdeckt Zeile 13 die meisten Spalten.  
Diese ist nicht Teil der gefundenen Lösung!

# Zusammenfassung Schaltkreise

$$f_n = r_1 \oplus x_2 \oplus x_3 \oplus \dots \oplus x_n$$

$\rightarrow 2^{n-1}$  Minterme

- Schaltkreise stellen boolesche Funktionen dar.
- Optimale boolesche Polynome können sehr viel größer sein, als entsprechende Schaltkreise.
  - exponentielle Unterschiede möglich
  - Rechtfertigung für Einsatz von Schaltkreisen statt PLAs
- Es gibt auch Algorithmen zur Berechnung optimaler (mehrstufiger) Schaltkreise.
  - anspruchsvoller als Optimierung von booleschen Polynomen
  - meist heuristisch (Näherungsverfahren)
  - nicht Gegenstand dieser Vorlesung
- Hier: Schaltkreise für spezielle Funktionen, insbesondere Arithmetik.

# Kapitel 3 – Kombinatorische Logik

1. Kombinatorische Schaltkreise
2. Normalformen, zweistufige Synthese
3. Berechnung eines Minimalpolynoms
- 4. Arithmetische Schaltungen**
5. Anwendung: ALU von ReTI

Albert-Ludwigs-Universität Freiburg

UNI  
FREIBURG

Dr. Tobias Schubert, Dr. Ralf Wimmer

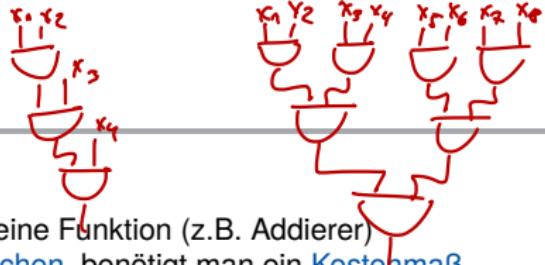
Professur für Rechnerarchitektur  
WS 2016/17

# Arithmetische Schaltungen

---

- Addieren nach der Schulmethode: [Carry-Ripple-Addierer](#).
- Effizienteres Addieren: [Conditional-Sum-Addierer](#).
- Addition von [Zweierkomplement-Zahlen](#).
- Subtrahierer.
- Exkurs: [Multiplizierer](#).

# Kosten von Schaltkreisen



- Um unterschiedliche Schaltkreise, die eine Funktion (z.B. Addierer) implementieren, miteinander zu vergleichen, benötigt man ein Kostenmaß.

## Definition

Die Kosten  $C(SK)$  eines Schaltkreises  $SK$  sind durch die Anzahl seiner Gatter gegeben.

- Deutet auf die Fläche und den Energieverbrauch der resultierenden Hardware-Blöcke hin.

*hier : nur der - Gatto  
keine negierten Gattereingänge*

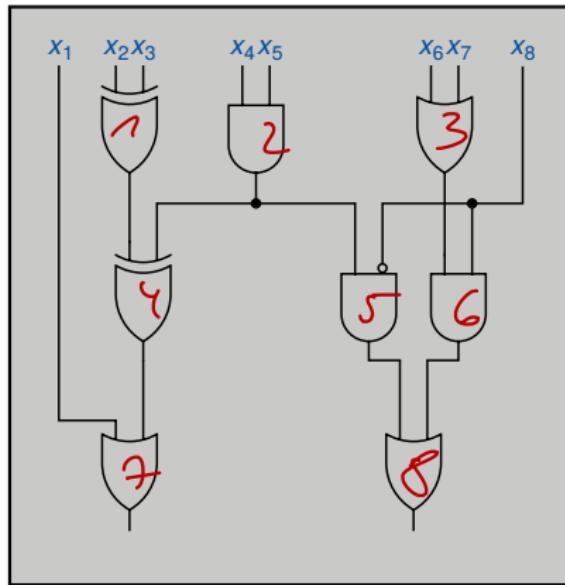
## Definition

Die Tiefe  $depth(SK)$  eines Schaltkreises ist die maximale Anzahl von Gattern auf einem Pfad von einem beliebigen Eingang zu einem beliebigen Ausgang von  $SK$ .

- Deutet auf die Signallaufzeit durch  $SK$  und somit die maximal mögliche Taktfrequenz (Geschwindigkeit) des Schaltkreises hin.

# Beispiel: Kosten und Tiefe

---

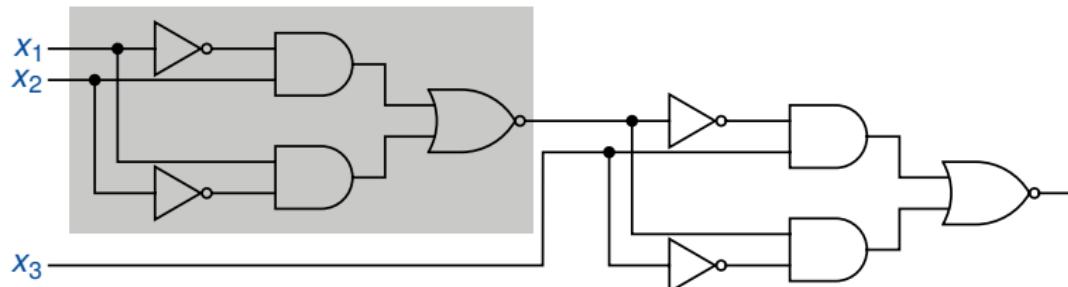


$$C(SK) = \underline{8}$$

$$\text{Depth}(SK) = 3$$

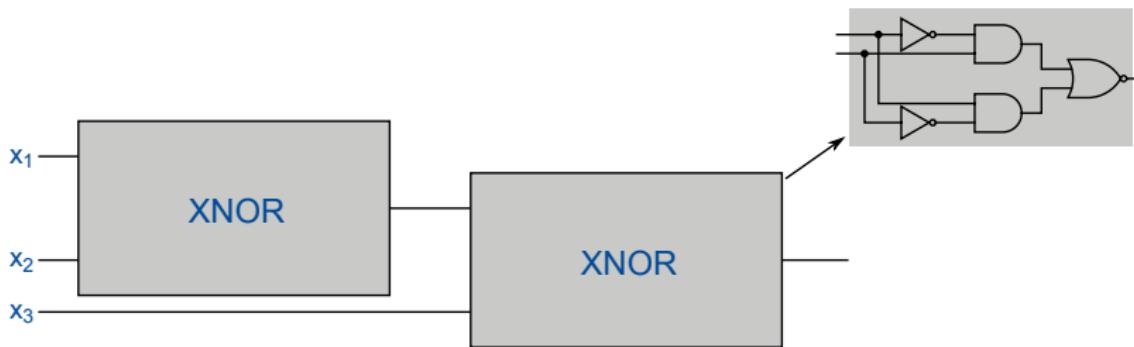
# Teilschaltkreise, hierarchischer Entwurf (informell)

- Illustration eines Teilschaltkreises.



# Hierarchische Schaltkreise

- In **hierarchischen Schaltkreisen** sind Teilschaltkreise durch Symbole ersetzt.
- Den zugehörigen („flachen“) Schaltkreis erhält man, indem man die Symbole durch **Einsetzen** der Teilschaltkreise wieder entfernt.



# Wiederholung Zahlendarstellung

$$\langle 1011 \rangle = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\ 8 + 0 + 2 + 1 = 11$$

Sei  $a = \underline{a_{n-1} \dots a_0}$  eine Folge von Ziffern,  $a_i \in \{0, 1\}$ .

- Binärdarstellung:  $\langle a \rangle = \sum_{i=0}^{n-1} a_i 2^i$
- Zweierkomplement:  $[a_n a_{n-1} \dots a_0] = \sum_{i=0}^{n-1} a_i 2^i - a_n 2^n$   
 $\langle a_{n-1} \dots a_0 \rangle - a_n 2^n$
- Rechenregel:  $\underline{-[a]} = [\bar{a}] + 1$

mit  $\bar{a} = \bar{a}_n \bar{a}_{n-1} \dots \bar{a}_0$ .

# Addierer für nichtnegative Zahlen

$$\begin{array}{r} 010011 \\ 001110 \\ \hline \overline{1\ 1\ 1\ 1\ 0} \end{array}$$

## ■ Gegeben:

2 positive Binärzahlen  $\langle a \rangle = \langle a_{n-1} \dots a_0 \rangle$ ,  $\langle b \rangle = \langle b_{n-1} \dots b_0 \rangle$   
mit Eingangsübertrag  $c \in \{0,1\}$ .

## ■ Gesucht:

Schaltkreis, der Binärdarstellung s von  $\langle a \rangle + \langle b \rangle + c$   
berechnet.

Eingänge:  $\begin{matrix} n & + & n & +1 \\ \xrightarrow{cas} & \xrightarrow{cbs} & \xrightarrow{c} \end{matrix}$

■ Wegen  $\langle a \rangle + \langle b \rangle + c \leq 2 \cdot (2^n - 1) + 1 = 2^{n+1} - 1$  genügen  
 $n + 1$  Stellen für die Darstellung von s, d.h. der Schaltkreis  
hat  $n + 1$  Ausgänge.

# Formale Definition $n$ -Bit-Addierer

---

- Ein  **$n$ -Bit-Addierer** ist ein Schaltkreis, der die folgende boolesche Funktion berechnet:

$$+_n : \mathbb{B}^{2n+1} \rightarrow \mathbb{B}^{n+1},$$

$$\underline{+_n} : (a_{n-1}, \dots, a_0, b_{n-1}, \dots, b_0, c) = (s_n, \dots, s_0)$$

$$\text{mit } \langle s \rangle = \langle s_n \dots s_0 \rangle = \langle a_{n-1} \dots a_0 \rangle + \langle b_{n-1} \dots b_0 \rangle + c$$

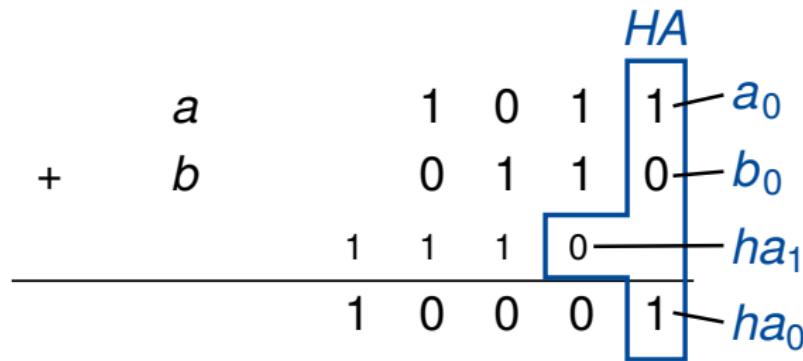
# Addieren nach der Schulmethode (1/4)

---

- Wir werden im Folgenden den einfachsten Addierertypen einführen, der die „**Schulmethode**“ umsetzt.
- Hierzu werden einige Grundschaltungen (Halb- und Volladdierer) notwendig sein.
- Beispiel für die Schulmethode:

$$\begin{array}{r} a & 1\cdot 0\ 1\ 1 \\ + b & \cdot 0\ 1\ 1\ 0 \\ \hline & 1\ 1\ 1\ 0 \\ & 1\ 0\ 0\ 0\ 1 \end{array}$$

## Addieren nach der Schulmethode (2/4)



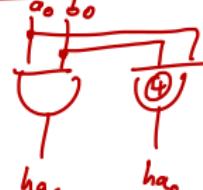
$a_0$	$b_0$	$ha_1$	$ha_0$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

# Halbaddierer (Half Adder, HA)

- Ein **Halbaddierer** dient zur Addition zweier 1-Bit-Zahlen ohne Eingangsübertrag.
- Er berechnet die Funktion  $ha : \mathbb{B}^2 \rightarrow \mathbb{B}^2$  mit  $ha(a_0, b_0) = (ha_1, ha_0)$ , wobei  $2ha_1 + ha_0 = a_0 + b_0$ .

$a_0$	$b_0$	$ha_1$	$ha_0$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$ha_0(a_0, b_0) = a_0 \oplus b_0$$
$$ha_1(a_0, b_0) = a_0 \wedge b_0$$



# Schaltkreis eines Halbaddierers

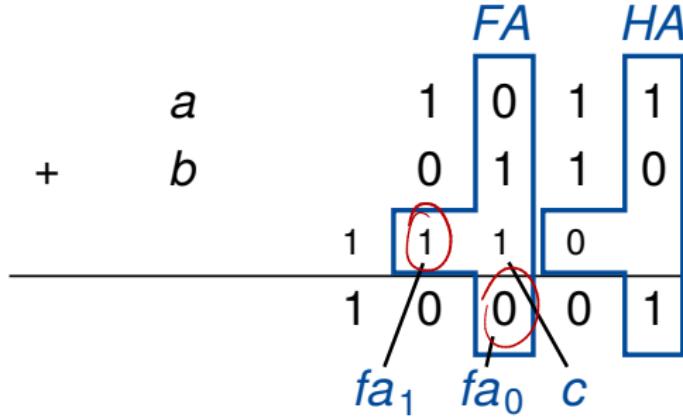
---



dabei gilt:

$$C(HA) = 2, \quad depth(HA) = 1$$

## Addieren nach der Schulmethode (3/4)



# Volladdierer (Full Adder, FA)

- Ein **Volladdierer** dient zur Addition zweier 1-Bit-Zahlen mit Eingangsübertrag.

- Er berechnet die Funktion  $fa : \mathbb{B}^3 \rightarrow \mathbb{B}^2$  mit  
 $fa(a_0, b_0, c) = (fa_1, fa_0)$   
wobei  $2fa_1 + fa_0 = a_0 + b_0 + c$

$a_0$	$b_0$	$c$	$fa_1$	$fa_0$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

# Volladdierer als Funktion von *HAs*

Aus der Tabelle folgt:

$$\begin{aligned}fa_0 &= a_0 \oplus b_0 \oplus c \\&= ha_0(c, ha_0(a_0, b_0))\end{aligned}$$

$$\begin{aligned}fa_1 &= (a_0 \wedge b_0) \vee (c \wedge (a_0 \oplus b_0)) \\&= ha_1(a_0, b_0) + ha_1(c, ha_0(a_0, b_0))\end{aligned}$$

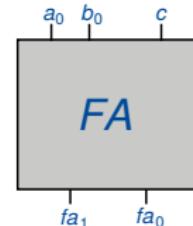
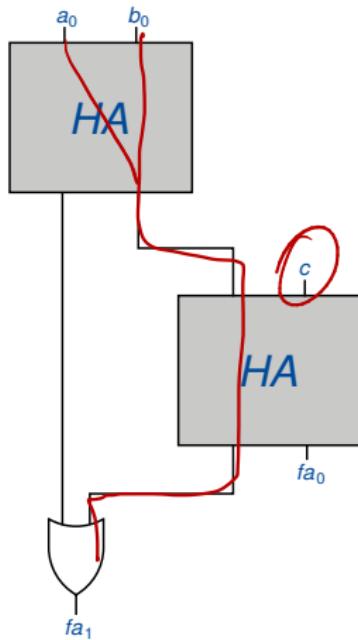
Kosten und Tiefe eines *FA*:

$$C(FA) = 5, \quad depth(FA) = 3$$

$a_0$	$b_0$	$c$	$fa_1$	$fa_0$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

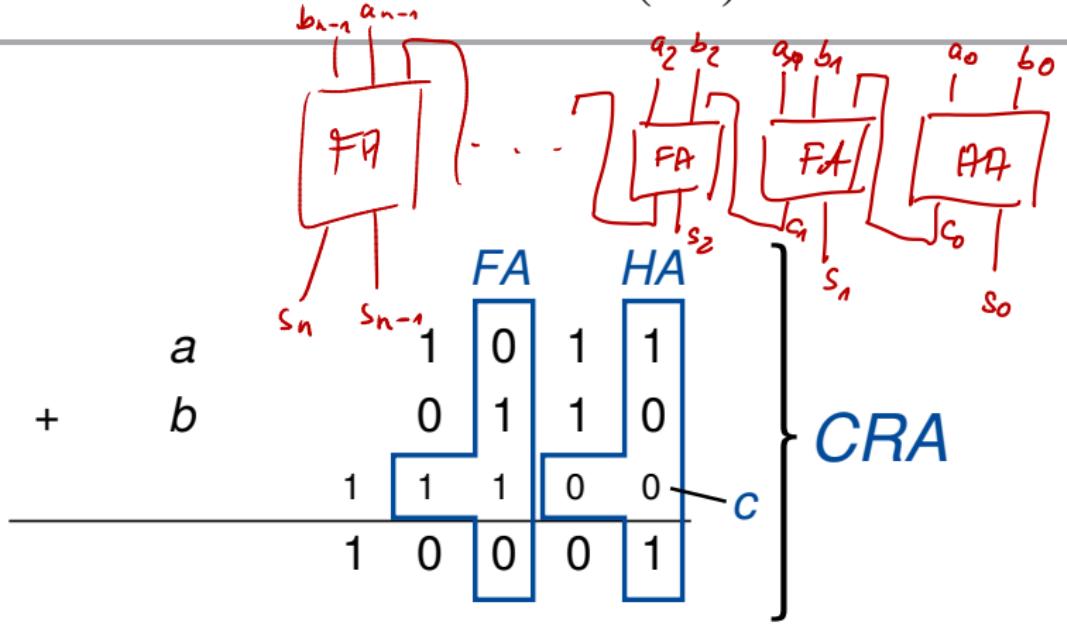
# Schaltkreis eines Volladdierers

$$\text{HA: } \begin{array}{l} C = 2 \\ d = 1 \end{array}$$

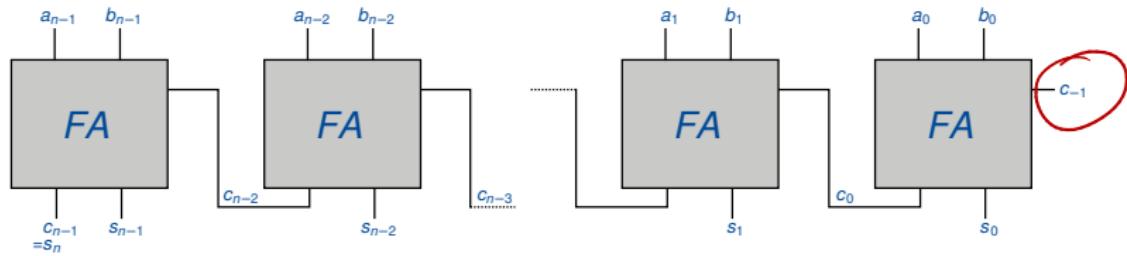


$$\begin{array}{l} C = 5 \\ d = 3 \end{array}$$

# Addieren nach der Schulmethode (4/4)



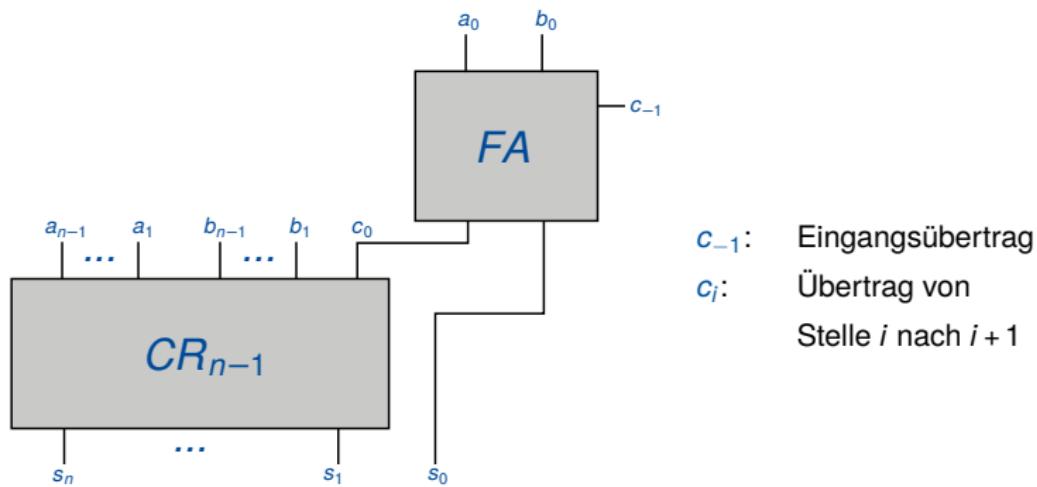
# Aufbau eines Carry-Ripple-Addierers



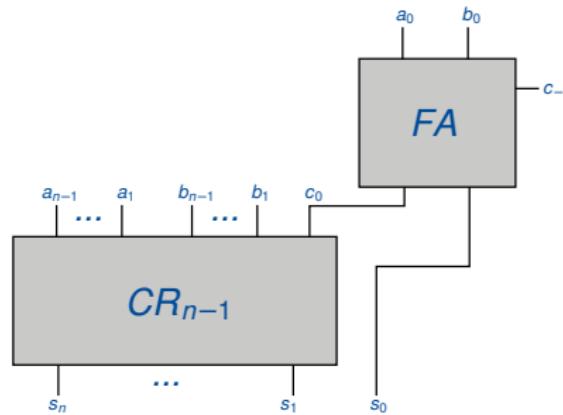
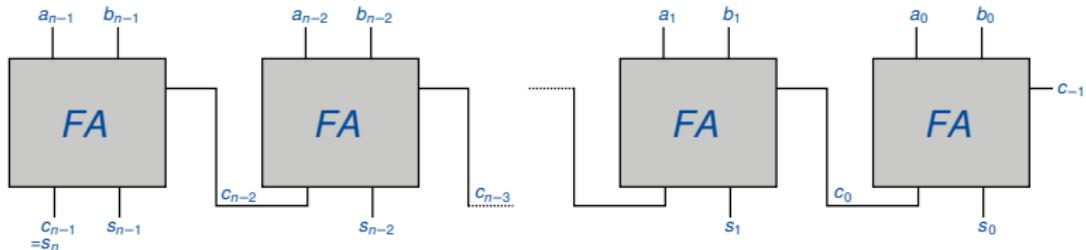
# Induktive Definition des Carry-Ripple-Addierers *CR*

---

- Für  $n = 1$  :  $CR_1 = FA$
- Für  $n > 1$  : Folgender Schaltkreis:



# Zwei (identische) Darstellungen von $CR$



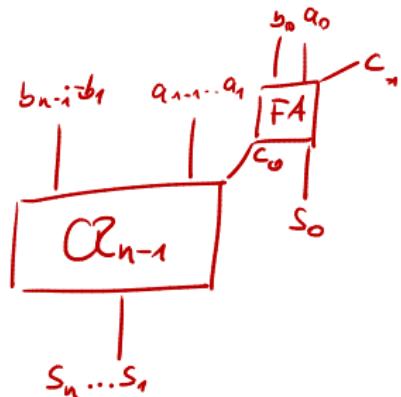
# Carry-Ripple-Addierer

## Satz

$CR_n$  ist ein  $n$ -Bit-Addierer.

**Beweis** (durch Induktion):

- $n = 1$  ( $CR_1 = FA$ ) ✓
- $n - 1 \rightarrow n$ : Eingabe an  $CR_n$ :  $(a_{n-1}, \dots, a_0, b_{n-1}, \dots, b_0, c_{-1})$   
Zeige für Ausgabe  $(s_n, \dots, s_0)$  von  $CR_n$ :  
 $\langle s \rangle = \langle s_n \dots s_0 \rangle = \langle a_{n-1} \dots a_0 \rangle + \langle b_{n-1} \dots b_0 \rangle + c_{-1}$ .
- Nach Induktionsvoraussetzung gilt für  $CR_{n-1}$ :  
 $\langle s_n \dots s_1 \rangle = \langle a_{n-1} \dots a_1 \rangle + \langle b_{n-1} \dots b_1 \rangle + c_0$ . (a)  
Wegen FA-Eigenschaft gilt  $\langle c_0, s_0 \rangle = a_0 + b_0 + c_{-1}$ . (b)
- Insgesamt:  $\langle s_n \dots s_0 \rangle = 2 \cdot \langle s_n \dots s_1 \rangle + s_0$   
(a)  $= 2 \cdot (\langle a_{n-1} \dots a_1 \rangle + \langle b_{n-1} \dots b_1 \rangle + c_0) + s_0$   
 $= 2 \cdot (\langle a_{n-1} \dots a_1 \rangle + \langle b_{n-1} \dots b_1 \rangle) + 2 \cdot c_0 + s_0 = a_0 + b_0 + c_{-1}$   
(b)  $= 2 \cdot \langle a_{n-1} \dots a_1 \rangle + a_0 + 2 \cdot \langle b_{n-1} \dots b_1 \rangle + b_0 + c_{-1}$   
 $= \langle a \rangle + \langle b \rangle + c_{-1}$



# Schaltbild und Komplexität von $CR$

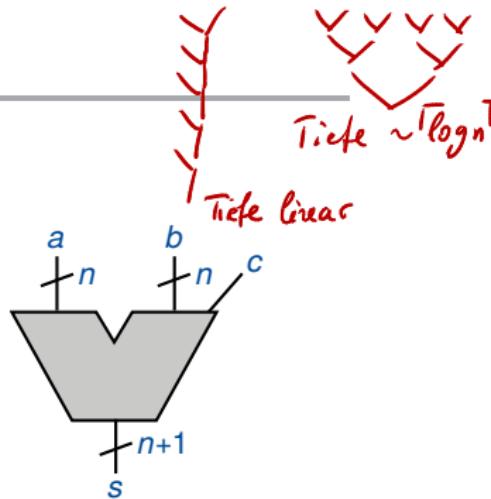
---

- $C(CR_n) = n \cdot C(FA) = 5n.$
- $depth(CR_n) = ?.$



# Schaltbild und Komplexität von $CR$

- $C(CR_n) = n \cdot C(FA) = \underline{5n}$ .
- $depth(CR_n) = 3 + \underline{2(n - 1)}$ .
- Sowohl die Kosten als auch die Tiefe von  $CR$  sind somit **linear** in  $n$ .
- Es gibt (asymptotisch) bessere Addierer. Wir werden hier den **Conditional-Sum-Addierer** kennen lernen, für den wir wieder eine Hilfsschaltung (**Multiplexer**) benötigen.
- Eine weitere wichtige Schaltung ist der **Inkrementer**.



# *n*-Bit-Inkrementer

## Definition

Ein *n*-Bit-Inkrementer  $INC_n$  berechnet die Funktion

$$inc_n : \mathbb{B}^{n+1} \rightarrow \mathbb{B}^{n+1}$$

$$inc_n(a_{n-1}, \dots, a_0, c) = (s_n, \dots, s_0) \text{ mit } \langle s_n \dots s_0 \rangle = \langle a \rangle + c$$

- Ein Inkrementer ist ein Addierer mit  $b_i = 0$  für alle *i*.  
⇒ Ersetze in  $CR_n$  die *FA* durch *HA*.
- Kosten und Tiefe:
  - $C(INC_n) = n \cdot C(HA) = \underline{2n}$
  - $\underline{depth(INC_n) = n \cdot depth(HA) = n}$

# $n$ -Bit-Multiplexer

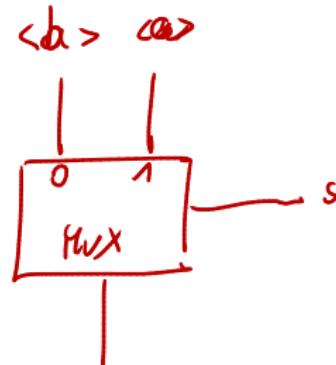
## Definition

Ein  $n$ -Bit-Multiplexer  $MUX_n$  berechnet die Funktion

$$\underline{sel}_n : \mathbb{B}^{2n+1} \rightarrow \mathbb{B}^n$$

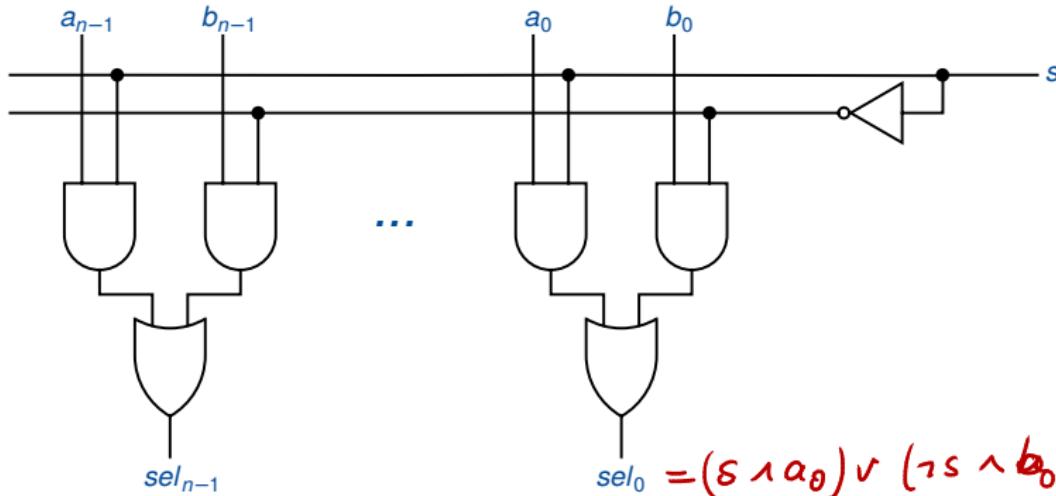
$$sel_n(a_{n-1}, \dots, a_0, b_{n-1}, \dots, b_0, s) = \begin{cases} (a_{n-1} \dots a_0), & \text{falls } s = 1 \\ (b_{n-1} \dots b_0), & \text{falls } s = 0 \end{cases}$$

- Es gilt:  $(sel_n)_i = s \cdot a_i + \bar{s} \cdot b_i$



# Aufbau von $MUX_n$

Tiefe : 3  
Kosten :  $3n+1$



$$sel_0 = (s \wedge a_0) \vee (\neg s \wedge b_0)$$

$$s=1 : (1 \wedge a_0) \vee (0 \wedge b_0) = a_0$$

$$s=0 : (0 \wedge a_0) \vee (1 \wedge b_0) = b_0$$

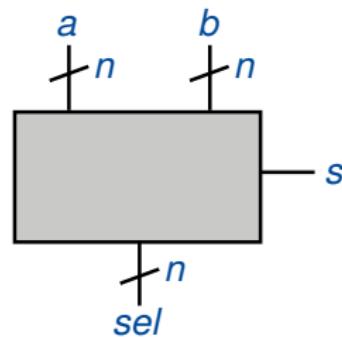
# Schaltbild und Kosten $MUX_n$

---

Kosten und Tiefe:

$$C(MUX_n) = 3n + 1.$$

$$\text{depth}(MUX_n) = 3.$$



# Rückkehr zum Addierer

---

Gibt es billigere Addierer als  $CR_n$ ?

**Untere Schranken:**

$$C(+_n) \geq \underline{2 \cdot n}, \quad \underline{\text{depth} (+_n)} \geq \underline{\log(n) + 1}$$

Sei  $f \in \mathbb{B}_n$ . Dann sind  $C(f)$  und  $\text{depth}(f)$  definiert durch

$$C(f) = \min\{C(SK) | f_{SK} = f\}$$
 und

$$\text{depth}(f) = \min\{\text{depth}(SK) | f_{SK} = f\}.$$

Binäre Bäume mit  $2n+1$  Blättern haben  $2n$  innere Knoten.

Binäre Bäume mit  $n$  Blättern haben mindestens Tiefe  $\lceil \log(n) \rceil$ .

Im Folgenden sei  $n = 2^k$ .

# Conditional-Sum-Addierer (CSA)

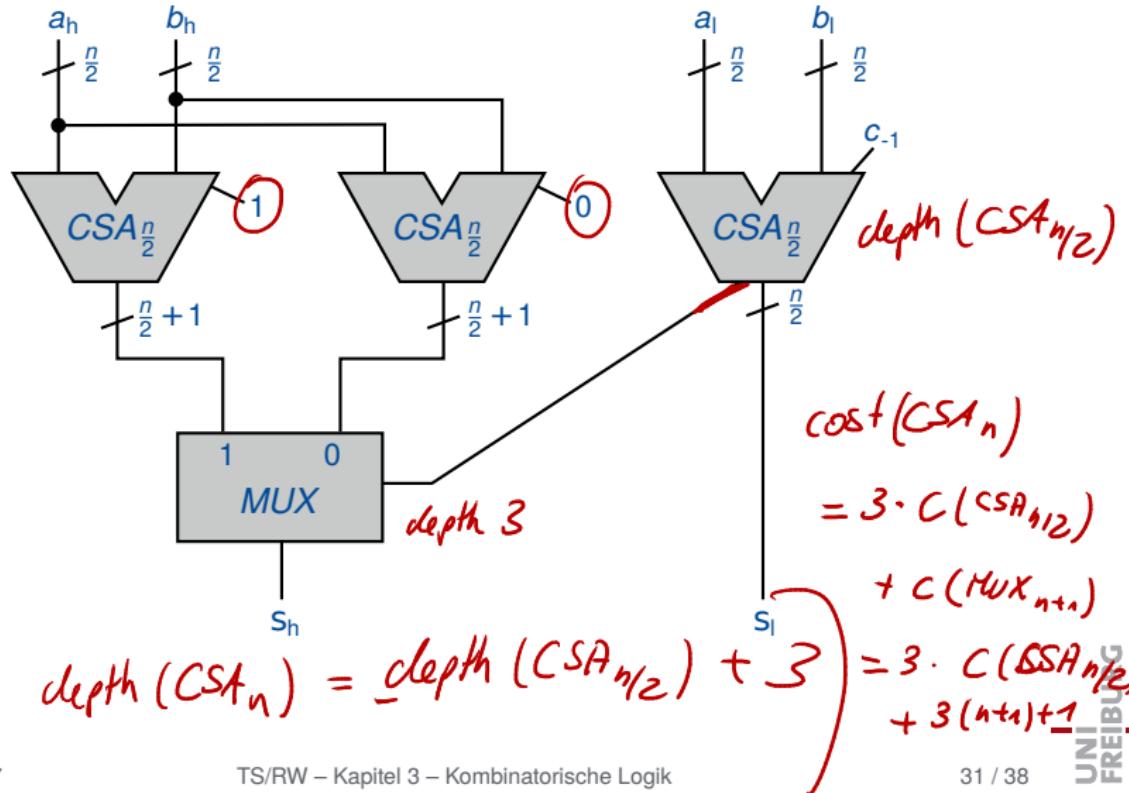
- Idee: Nutze Parallelverarbeitung, um Tiefe zu reduzieren!

$$\begin{array}{r} & a_{n-1} \dots a_{\frac{n}{2}} \\ + & b_{n-1} \dots b_{\frac{n}{2}} \\ + & c_{\frac{n}{2}-1} \\ \hline s_n s_{n-1} \dots s_{\frac{n}{2}} & \end{array} \quad \begin{array}{r} a_{\frac{n}{2}-1} \dots a_0 \\ b_{\frac{n}{2}-1} \dots b_0 \\ c_{-1} \\ \hline s_{\frac{n}{2}-1} \dots s_0 \end{array}$$

- $CSA_1 = FA$ .
- $CSA_n$ : Siehe nächste Folie.
- Im Folgenden sei  $n = 2^k$ .

# Aufbau von $CSA_n$

$$a_{n-1} \dots a_0 \quad \begin{cases} a_e \\ a_h \end{cases}$$



# Komplexität von $CSA_n$ : Tiefe

## Satz

$CSA_n$  hat Tiefe  $\leq 3 \log(n) + 3$ .

**Beweis:** Setze  $n = 2^k$  voraus.

■  $n = 1$ :  $\text{depth}(CSA_1) = \text{depth}(FA) = 3$ .

$$n = 2^k$$

$$\begin{aligned} ■ \quad n > 1: \text{depth}(CSA_n) &\leq \text{depth}(CSA_{\frac{n}{2}}) + \text{depth}(MUX_{\frac{n}{2}+1}) \\ &\leq \text{depth}(CSA_{\frac{n}{2}}) + 3 \\ &\leq \text{depth}(CSA_{\frac{n}{4}}) + 3 + 3 \\ &\leq \text{depth}(CSA_{\frac{n}{8}}) + 3 + 3 + 3 \\ &\dots \\ &\leq \text{depth}(CSA_{\frac{n}{2^k}}) + k \cdot 3 \\ &= \text{depth}(CSA_1) + k \cdot 3 \\ &\leq 3 \cdot (k + 1) = 3 \log(n) + 3. \end{aligned}$$

# Komplexität von $CSA_n$ : Kosten

## Satz (ohne Beweis)

$$C(CSA_n) = 10n^{\log(3)} - 3n - 2.$$

$$C(CSA_n) = 3 \cdot C(CSA_{1/2}) + 3n + 4$$

- Man kann den hier vorgestellten  $CSA$  in einfacher Weise modifizieren, so dass

- Tiefe =  $O(\log(n))$ ,
- Kosten =  $O(n \cdot \log(n))$ .



- Es gibt auch Addierer mit linearen Kosten und logarithmischer Tiefe.
  - Carry-Lookahead-Addierer (*CLA*).
  - $C(CLAn) \leq 11n$ ,
  - $\underline{depth(CLAn) \leq 4 \cdot \log(n) + 2}$ .

# Addition von Zweierkomplementzahlen

- Auszurechnen ist:

$$[a_n a_{n-1} \dots a_0] + [b_n b_{n-1} \dots b_0] = (-a_n 2^n) + (-b_n 2^n) + \sum_{i=0}^{n-1} a_i 2^i + \sum_{i=0}^{n-1} b_i 2^i$$

- Im Fall von  $(n+1)$ -Bit-Zweierkomplementzahlen können Ergebnisse im Bereich  $R_n = \{-2^n, \dots, 2^n - 1\}$  dargestellt werden; andernfalls kommt es zu einem Überlauf.
- Der Satz auf der nächsten Folie sagt aus:
  - Kommt es bei der Addition nicht zu einem Überlauf, so kann man den „gewöhnlichen“ Binäraddierer zur Addition von Zweierkomplementzahlen benutzen.
  - Ob es zu einem Überlauf kommt, lässt sich anhand von Werten  $a_n$ ,  $b_n$  und  $s_n$  im Binäraddierer entscheiden.

# Zweierkomplement-Addition formal

## Satz

Seien  $\underline{a}, \underline{b} \in \mathbb{B}^{n+1}$ ,  $\underline{c}_{-1} \in \{0, 1\}$  und  $\underline{s} \in \{0, 1\}^{n+1}$ ,  
so dass  $\langle \underline{c}_n, \underline{s} \rangle = \langle \underline{a} \rangle + \langle \underline{b} \rangle + \underline{c}_{-1}$ .

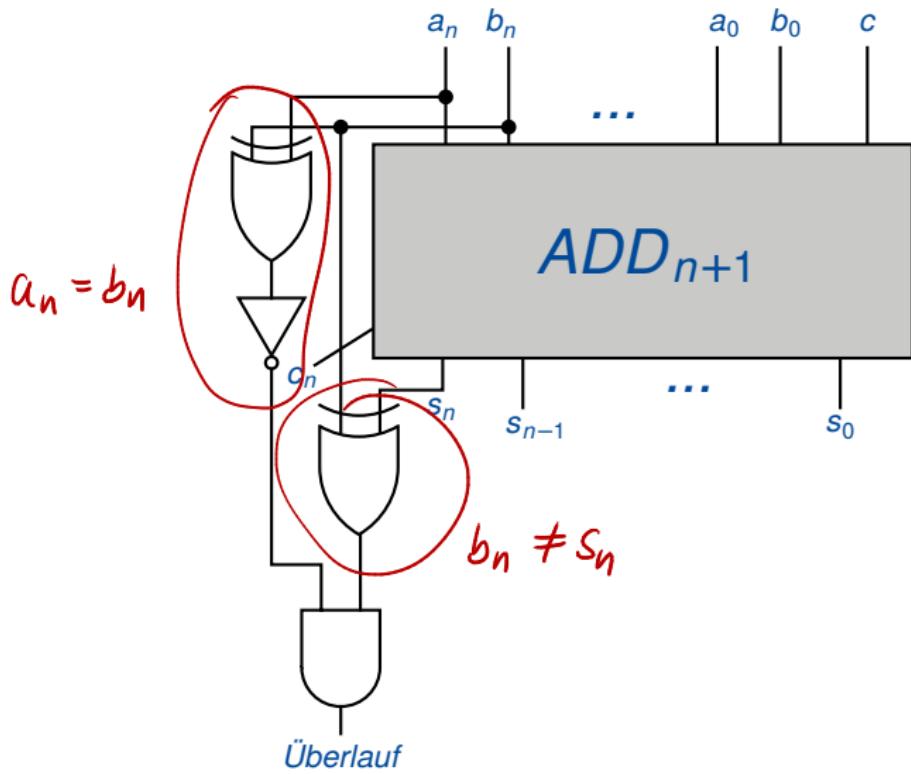
Dann gilt:

- $[\underline{a}] + [\underline{b}] + \underline{c}_{-1} \notin R_n \Leftrightarrow (a_n = b_n) \wedge (b_n \neq s_n)$  *Überlauferkennung*
- $[\underline{a}] + [\underline{b}] + \underline{c}_{-1} \in R_n \Rightarrow [\underline{a}] + [\underline{b}] + \underline{c}_{-1} = [\underline{s}]$

- Beweis durch Fallunterscheidung ( $[\underline{a}], [\underline{b}]$  beide positiv, beide negativ,  $[\underline{a}]$  positiv /  $[\underline{b}]$  negativ) und Nachrechnen (Induktion).
- Man kann einen alternativen Überlauftest zeigen:

$$[\underline{a}] + [\underline{b}] + \underline{c}_{-1} \notin R_n \Leftrightarrow c_n \neq c_{n-1}$$

# Addierer für $(n + 1)$ -Bit-Zweierkomplement-Zahlen



# Subtraktion

---

- Wegen  $-[b] = [\bar{b}] + 1$  kann  $[a] - [b]$  zurückgeführt werden auf  $[a] + [\bar{b}] + 1$ .

- **Beispiel:**

$\cancel{C}_-1$

$$[a] = [0110] = 6_{10}, \quad [b] = [0111] = 7_{10}, \quad [\bar{b}] = [1000]$$

$$\begin{array}{r} 0110 \\ + 1000 \\ + 1 \\ \hline 1111 \end{array} \qquad 1111 = (-1)_{10}$$

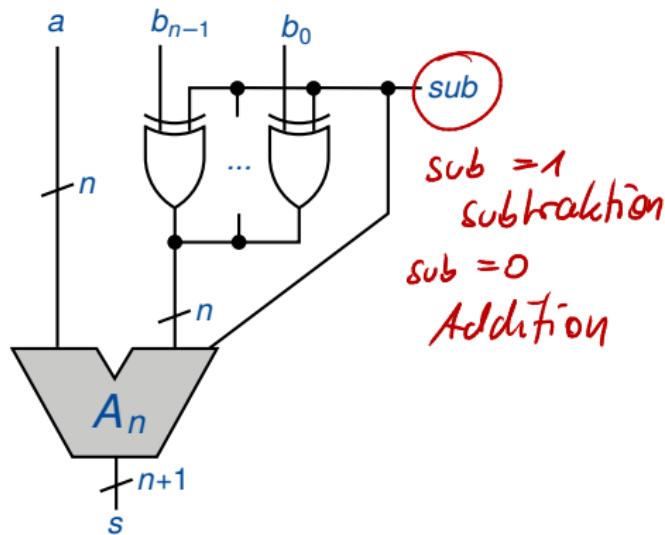
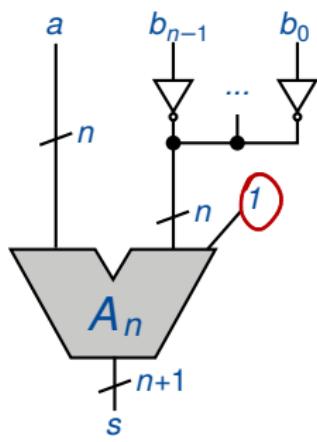
- Den Schaltkreis für Subtraktion gewinnt man aus einem Addierer.
- Kombinierter Addierer/Subtrahierer.

# Subtrahierer

$$b \oplus 0 = b$$

$$b \oplus 1 = -b$$

reiner Subtrahierer



$$b_i \oplus 0 = b_i$$

$$b_i \oplus 1 = \bar{b}_i$$

$$\text{sub} = 0 : [a] + [b] + 0$$

$$\text{sub} = 1 : [a] + [\bar{b}] + 1 = [a] - [b]$$



# Exkurs – Multiplizierer

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur  
WS 2016/17

UNI  
FREIBURG

# Multiplizierer

- **Gesucht:** Schaltkreis zur Multiplikation zweier Binärzahlen  
 $\langle a_{n-1}, \dots, a_0 \rangle, \langle b_{n-1}, \dots, b_0 \rangle$ .

- **Beispiel:**

$$\underbrace{(110)}_{6_{10}} \cdot \underbrace{(101)}_{5_{10}}$$

$$\begin{array}{r} 243 \cdot 12 \\ \hline 2430 \\ 486 \\ \hline 2916 \end{array}$$

$$\begin{array}{r} & 1 & 1 & 0 \\ & \underline{0} & 0 & \underline{0} \\ + & 0 & 0 & 0 \\ + & 1 & 1 & 0 \\ \hline & 1 & 1 & 1 & 0 \end{array} \quad \begin{array}{ll} i=0 & b_i=1 \\ i=1 & b_i=0 \\ i=2 & b_i=1 \end{array}$$

$$= 30_{10}$$

# Allgemeines zum Multiplizierer

---

- Wieviele Stellen werden für das Ergebnis benötigt?

$$\begin{aligned} < a > \cdot < b > &\leq (2^n - 1) \cdot (2^n - 1) \\ = \underline{2^{2n} - 2^{n+1} + 1} &\leq 2^{2n} - 1 \end{aligned}$$

- Also:

2n Stellen zur Multiplikation von Binärzahlen.

# Vorgehen bei der Multiplikation

---

- Multipliziere die Beträge der Zahlen.
- Bestimme das Vorzeichen des Produkts.
- Setze das Endergebnis zusammen.

# n-Bit-Multiplizierer

## Definition

Ein **n-Bit-Multiplizierer** ist ein Schaltkreis, der die folgende Funktion berechnet:

$$mul_n : \{0,1\}^{2n} \rightarrow \{0,1\}^{2n}$$

$$mul_n(\underline{a_{n-1}, \dots, a_0}, \underline{b_{n-1}, \dots, b_0}) = (p_{2n-1}, \dots, p_0) \text{ mit}$$

$$\langle p_{2n-1}, \dots, p_0 \rangle = \langle a \rangle \cdot \langle b \rangle$$

$\langle a \rangle + i \text{ Nullen}$   
oder  $\langle 0 \rangle + i \text{ Nullen}$

$$\underbrace{\langle a \rangle \cdot \langle b \rangle}_{=} = \underbrace{\langle a \rangle}_{\cdot} \sum_{i=0}^{n-1} b_i \cdot 2^i = \sum_{i=0}^{n-1} (\underbrace{\langle a \rangle \cdot b_i}_{\cdot} \cdot 2^i)$$

# Die Multiplikationsmatrix

$$\begin{pmatrix} pp_0 \\ pp_1 \\ \vdots \\ pp_{n-1} \end{pmatrix} =$$

*1 And-Gatter*

$$\begin{pmatrix} 0 & 0 & \dots & 0 & 0 & a_{n-1}b_0 & a_{n-2}b_0 & \dots & a_1b_0 & a_0b_0 \\ 0 & 0 & \dots & 0 & a_{n-1}b_1 & a_{n-2}b_1 & a_{n-3}b_1 & \dots & a_0b_1 & 0 \\ \vdots & \vdots & & \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & a_{n-1}b_{n-1} & \dots & a_2b_{n-1} & a_1b_{n-1} & a_0b_{n-1} & 0 & \dots & 0 & 0 \end{pmatrix}$$

*i = 0*  
*i = 1*  
*i = 2*  
*i = n - 1*

*n - 1 Nullen*

- Realisierung der Multiplikationsmatrix mit  $n^2$  AND-Gattern (und  $n^2$  Konstanten 0).

## Daraus entstehende Aufgabe:

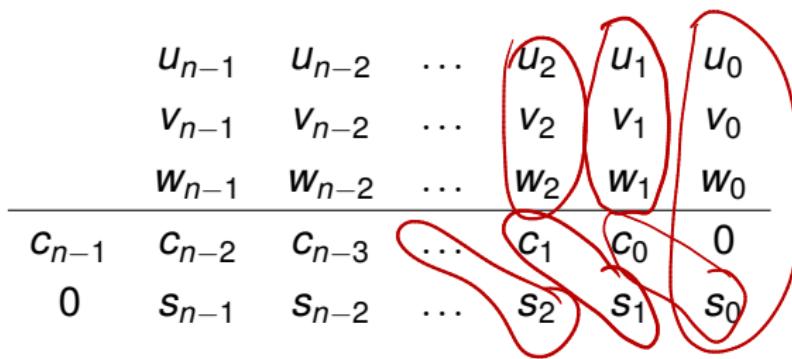
---

- Schnelle Addition von  $n$  Partialprodukten der Länge  $2n$ .
- Mit Carry-Lookahead-Addierern (CLAs) lösbar mit Kosten  $O(n^2)$ , Tiefe  $O(n \log(n))$  bei *linearem Aufsummieren* der Partialprodukte  $((((pp_0 + pp_1) + pp_2) + \dots) + pp_{n-1})$ , Tiefe  $O(\log^2(n))$  bei baumartigem Zusammenfassen der Partialprodukte.

bisher :  $C(Mul_n) = O(n^2)$   
 $depth(Mul_n) = O(\log^2 n)$  mit optimalem Addierer!

# Verbesserung

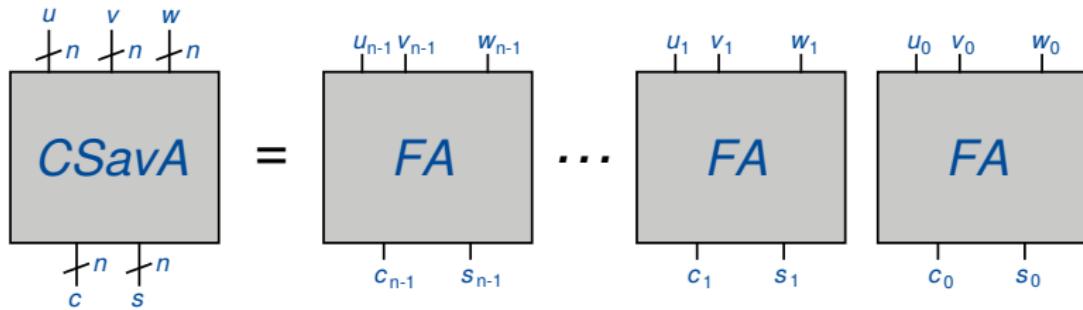
- Verwende Carry-Save-Addierer.
- Reduktion von 3 Eingabeworten  $u, v, w$  zu zwei Ausgabeworten  $s, c$  mit
$$\langle u \rangle + \langle v \rangle + \langle w \rangle = \langle s \rangle + \langle c \rangle.$$



- Gelöst durch Nebeneinandersetzen von Volladdierern (keine Carry-Chain!)

# Carry-Save-Addierer (CSavA)

$$\langle u \rangle + \langle v \rangle + \langle w \rangle = \langle c \rangle + \langle s \rangle$$



Volladdiere sind nicht verbunden !!

Tiefe = 3

Kosten =  $5n$

# Bemerkung zum Aufbau des CSavA

---

- Speziell bei **Partialprodukten**:  
Reduziere **3** 2n-Bit-Zahlen zu **2** 2n-Bit-Zahlen.

$$\begin{array}{cccc} pp_{0,2n-1} & \dots & pp_{0,1} & pp_{0,0} \\ pp_{1,2n-1} & \dots & pp_{1,1} & pp_{1,0} \\ \hline pp_{2,2n-1} & \dots & pp_{2,1} & pp_{2,0} \\ c_{2n-2} & \dots & c_0 & 0 \\ s_{2n-1} & \dots & s_1 & s_0 \end{array}$$

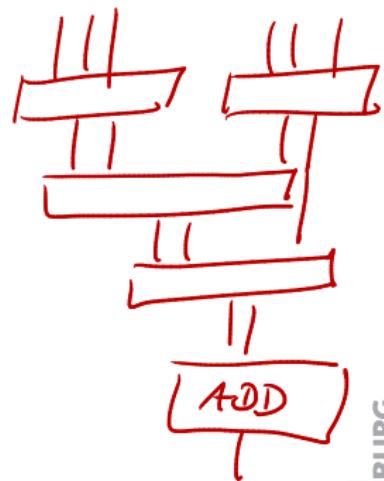
||

- ( $c_{2n-1} = 0$ : Carry-Ausgang des letzten FA nicht verwendet.)

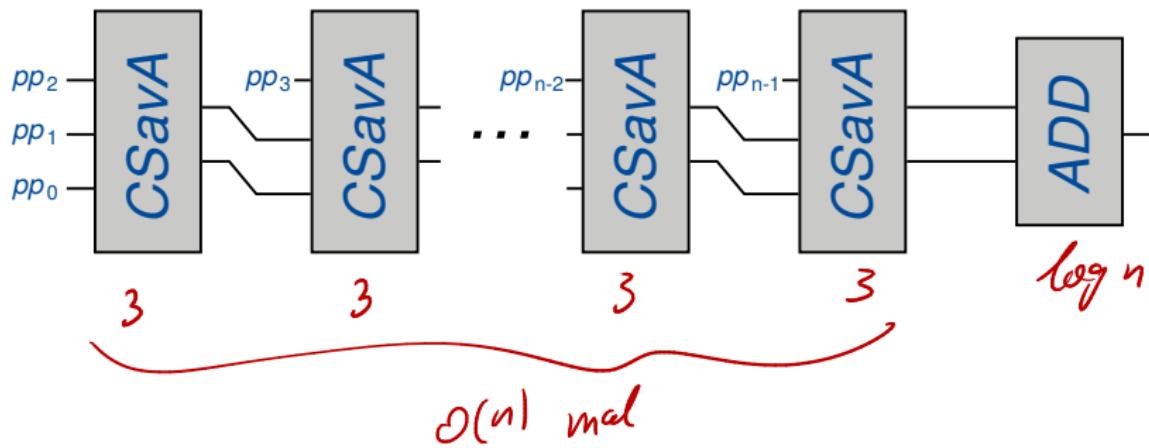
# 1. Serielle Lösung

---

- Hintereinanderschalten von  $n-2$  CSavA-Addierern der Länge  $2n$ .
  - Fasse  $n$  Partialprodukte zu 2  $2n$ -Bit-Worten zusammen.
- Addiere die  $2n$ -Bit-Worte mit CLA.
  - *siehe Abb. einer Addierstufe*
  - Kosten  $O(n^2)$ , Tiefe  $O(n)$

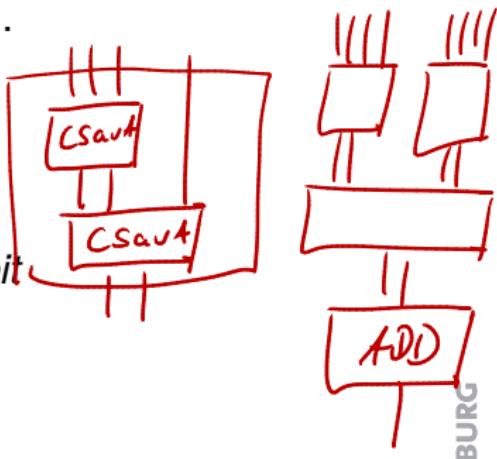


# Addierstufe im Multiplizierer



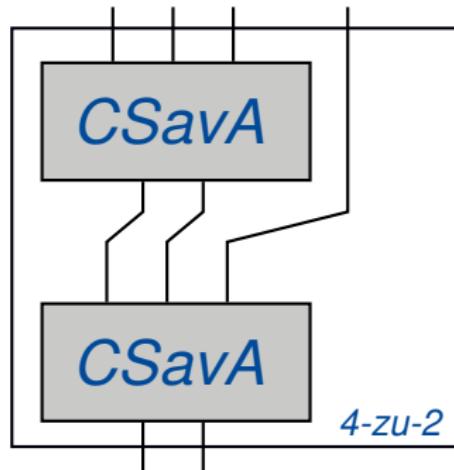
## 2. Baumartige Lösung

- Neue Grundzelle zur Reduktion von  $4 \times 2n$ -Bit-Eingabeworten zu zwei Ausgabeworten, bestehend aus 2 CSavA-Addierern (siehe Abb. zur Reduktionszelle).
- Baumartiges Zusammenfassen der Partialprodukte mit 4-zu-2-Bausteinen zu 2  $2n$ -Bit Worten.
- Addiere die  $2n$ -Bit-Worte mit CLA.
- siehe Abb. der Addierstufe mit log. Zeit
- Kosten  $O(n^2)$ , Tiefe  $O(\log n)$

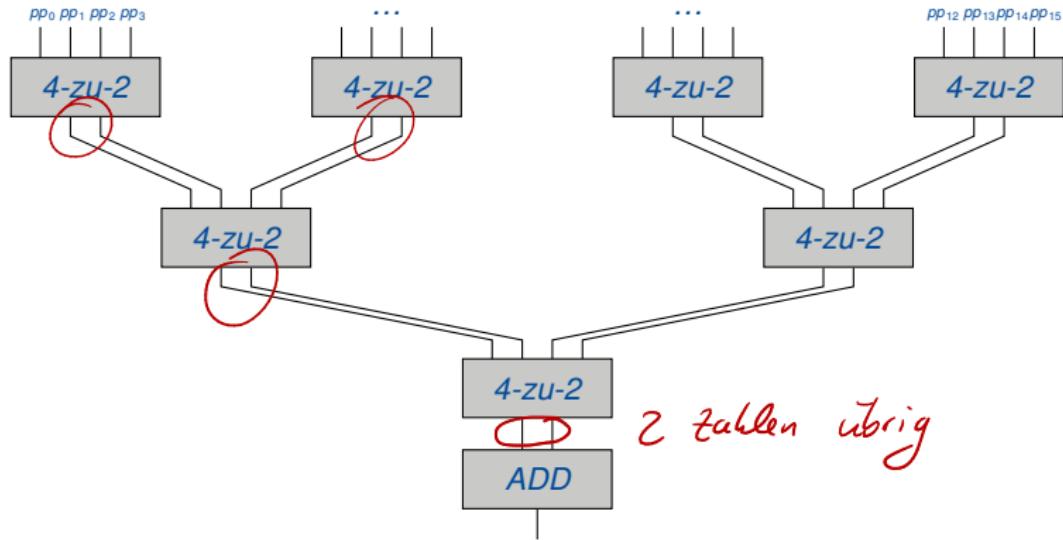


# 4-zu-2 Reduktions-Grundzelle

---



# Addierstufe des log-Zeit-Multiplizierers für 16 Bit





# Kapitel 3 – Kombinatorische Logik

1. Kombinatorische Schaltkreise
2. Normalformen, zweistufige Synthese
3. Berechnung eines Minimalpolynoms
4. Arithmetische Schaltungen
5. **Anwendung: ALU von ReTI**

# Anwendung: ALU von ReTI

---

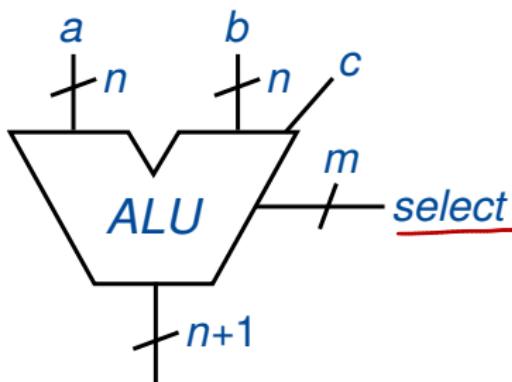
- Die **ALU** (Arithmetic Logic Unit, arithmetisch-logische Einheit) dient der Berechnung von **arithmetischen** und **logischen Operationen**.
- Sie wird von den **Compute-Befehlen** verwendet und übernimmt weitere Aufgaben, z.B. Berechnung von Speicheradressen oder Erhöhung des *PC*.
- Erinnerung: Der Befehlssatz von ReTI hat die folgenden Compute-Befehle (s. nächste Folie).

# Compute-Befehle: Kodierung

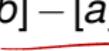
Typ	MI	F	Befehl	Wirkung	
<u>0 0</u>	<u>0</u>	0 1 0	SUBI <i>i</i>	$[ACC] := [ACC] - [\underline{i}]$	$\langle PC \rangle := \langle PC \rangle + 1$
		0 1 1	ADDI <i>i</i>	$[ACC] := [ACC] + [\underline{i}]$	$\langle PC \rangle := \langle PC \rangle + 1$
		1 0 0	OPLUSI <i>i</i>	$ACC := ACC \oplus \underline{0^8 i}$	$\langle PC \rangle := \langle PC \rangle + 1$
		1 0 1	ORI <i>i</i>	$ACC := ACC \vee \underline{0^8 i}$	$\langle PC \rangle := \langle PC \rangle + 1$
		1 1 0	ANDI <i>i</i>	$ACC := ACC \wedge \underline{0^8 i}$	$\langle PC \rangle := \langle PC \rangle + 1$
<u>0 0</u>	<u>1</u>	0 1 0	SUB <i>i</i>	$[ACC] := [ACC] - [M(\underline{\langle i \rangle})]$	$\langle PC \rangle := \langle PC \rangle + 1$
		0 1 1	ADD <i>i</i>	$[ACC] := [ACC] + [M(\underline{\langle i \rangle})]$	$\langle PC \rangle := \langle PC \rangle + 1$
		1 0 0	OPLUS <i>i</i>	$ACC := ACC \oplus M(\underline{\langle i \rangle})$	$\langle PC \rangle := \langle PC \rangle + 1$
		1 0 1	OR <i>i</i>	$ACC := ACC \vee M(\underline{\langle i \rangle})$	$\langle PC \rangle := \langle PC \rangle + 1$
		1 1 0	AND <i>i</i>	$ACC := ACC \wedge M(\underline{\langle i \rangle})$	$\langle PC \rangle := \langle PC \rangle + 1$

# Spezifikation der ALU für ReTI

- Eine  $n$ -Bit-ALU mit:
  - Zwei  $n$ -Bit-Operanden  $a, b$ , Eingangscarry  $c$ ,
    - ReTI:  $n = 32$ .
  - Einem  $m$ -Bit select-Eingang, der ausgewählt, welche Funktion ausgeführt wird,
    - Hier: 8 Funktionen (s. nächste Folie), daher  $m = 3$  Bits.
  - Einem  $(n + 1)$ -Bit-Ausgang.
    - $n = 32$ .
- Insgesamt 68 Ein- und 33 Ausgänge.



# Select-Eingang bei ReTI-ALU

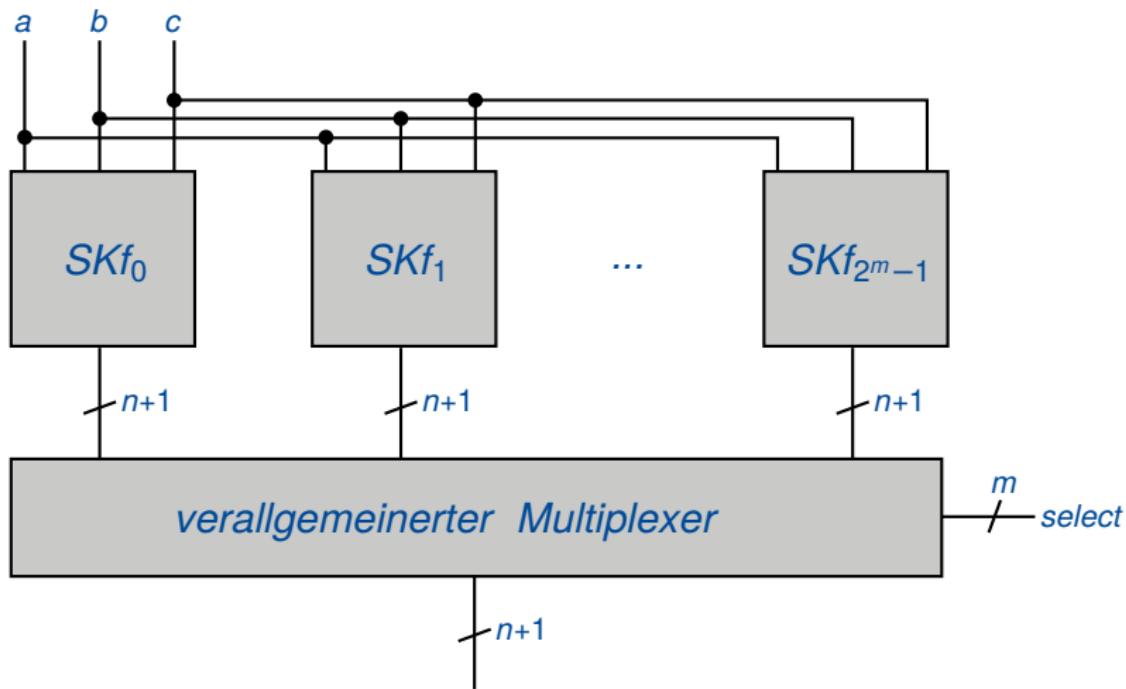
Funktionsnummer	ALU-Funktion
$s_2 \quad s_1 \quad s_0$  0 0 0	0...0
0 0 1	$[b] - [a]$ 
0 1 0	$[a] - [b]$
0 1 1	$[a] + [b] + c$
1 0 0	$a \oplus b = (a_{n-1} \oplus b_{n-1}, \dots, a_0 \oplus b_0)$
1 0 1	$a \vee b = (a_{n-1} \vee b_{n-1}, \dots, a_0 \vee b_0)$
1 1 0	$a \wedge b = (a_{n-1} \wedge b_{n-1}, \dots, a_0 \wedge b_0)$
1 1 1	1...1

# Mögliche Realisierungen der ALU (1/2)

---

- **Option 1:** Realisiere Funktionen  $f_0, \dots, f_{2^m-1}$  getrennt durch  $SK_{f_i}$  für  $f_i$ , dann Auswahl durch einen **verallgemeinerten Multiplexer**.

# Realisierung durch einen verallgemeinerten Multiplexer

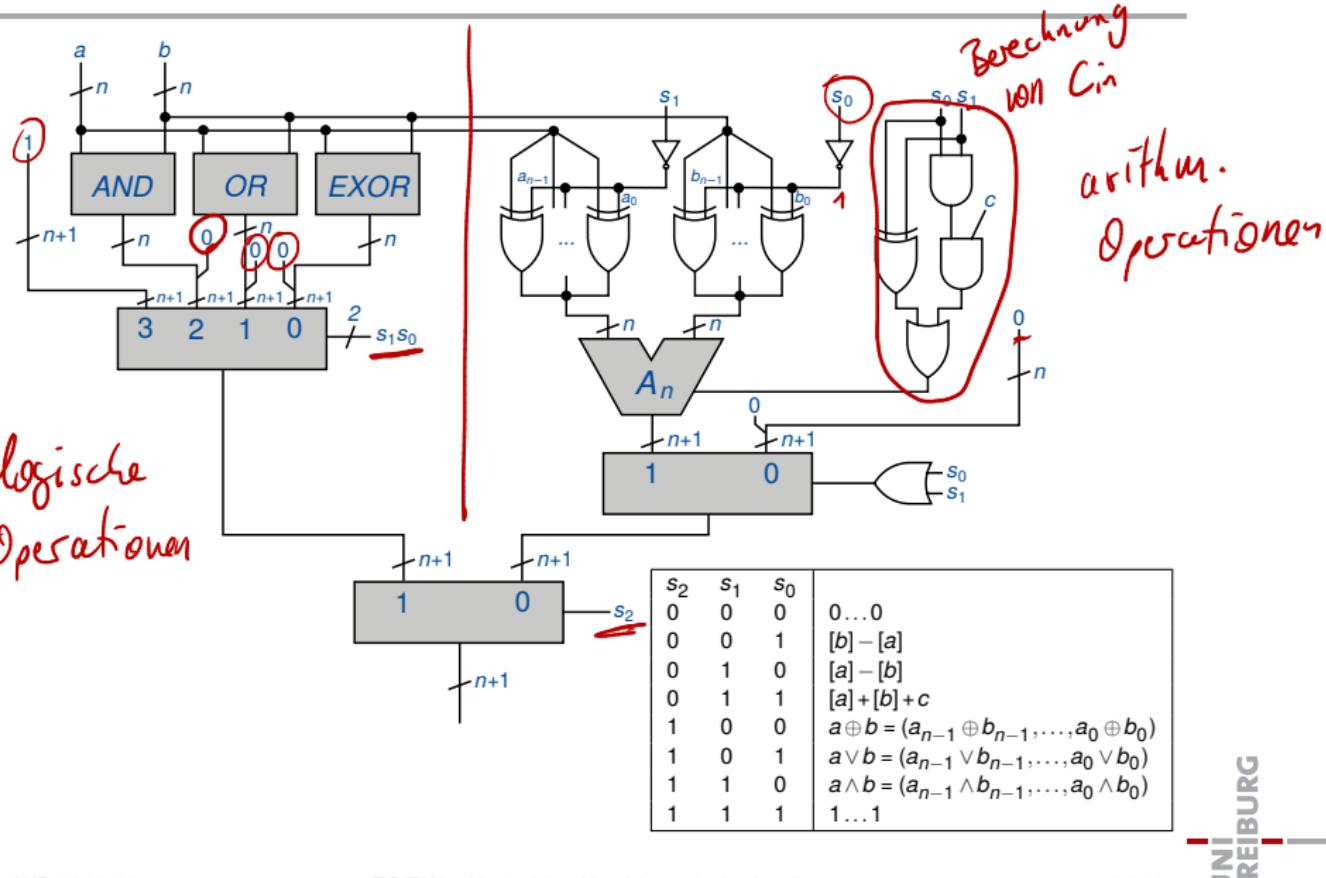


## Mögliche Realisierungen der ALU (2/2)

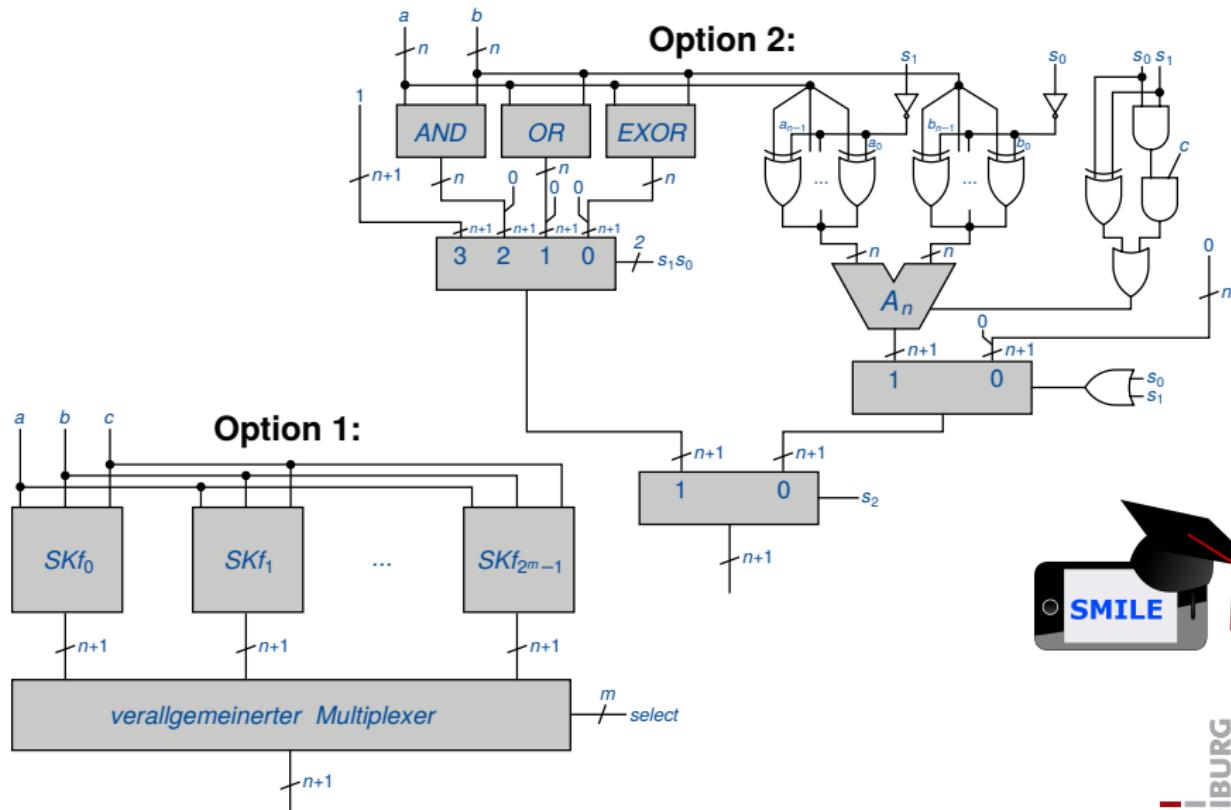
---

- **Option 1:** Realisiere Funktionen  $f_0, \dots, f_{2^m-1}$  getrennt durch  $SK_{f_i}$  für  $f_i$ , dann Auswahl durch einen **Verallgemeinerten Multiplexer**.
- **Option 2:** Gemeinsame Behandlung ähnlicher Funktionen.
  - Komplexer, aber effizienter.

# Schaltrealisierung der ALU (1/2)



# Schaltrealisierung der ALU (2/2)



# SMILE - Vergleich Option 1 mit Option 2

---

Wie verhalten sich Kosten und Tiefe der beiden Realisierungsoptionen zueinander?

- a.  $(Tiefe(Option1) \leq Tiefe(Option2))$  und  $(Kosten(Option1) \leq Kosten(Option2))$
- b.  $(Tiefe(Option1) \geq Tiefe(Option2))$  und  $(Kosten(Option1) \leq Kosten(Option2))$
- c.  $(Tiefe(Option1) \leq Tiefe(Option2))$  und  $(Kosten(Option1) \geq Kosten(Option2))$
- d. keine der obigen

# Zusammenfassung Kombinatorische Logik

---

- Kombinatorische Schaltkreise setzen boolesche Funktionen um.
- PLAs sind zweistufig, mehrstufige Schaltungen bestehen aus Gattern und diese aus Transistoren.
- Minimierung von PLAs mit Verfahren von Quine-McCluskey und Lösen des Überdeckungsproblems.
- Statt Minimierung allgemeiner mehrstufiger Schaltkreise wurde eine Klasse (Addierer für Binär- und Zweierkomplementzahlen) betrachtet und ihre Integration in der ALU von ReTI diskutiert.



# Kapitel 4 – Sequentielle Logik

1. Speichernde Elemente
2. Sequentielle Schaltkreise
3. Entwurf sequentieller Schaltkreise
4. SRAM
5. Anwendung: Datenpfade von ReTI



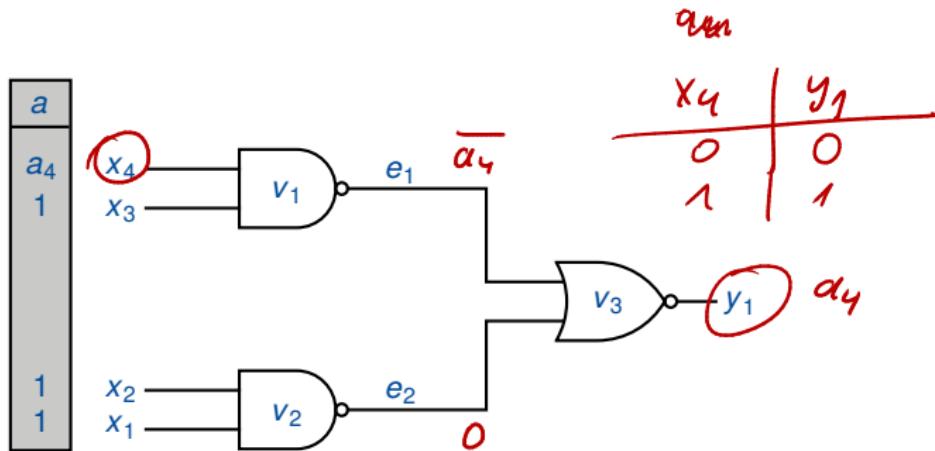
Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur  
WS 2016/17

# Schaltkreis (1/2)

$$\begin{aligned} \text{typ}(v_1) &= \text{typ}(v_2) = \text{NAND} \\ \text{typ}(v_3) &= \text{NOR} \end{aligned}$$

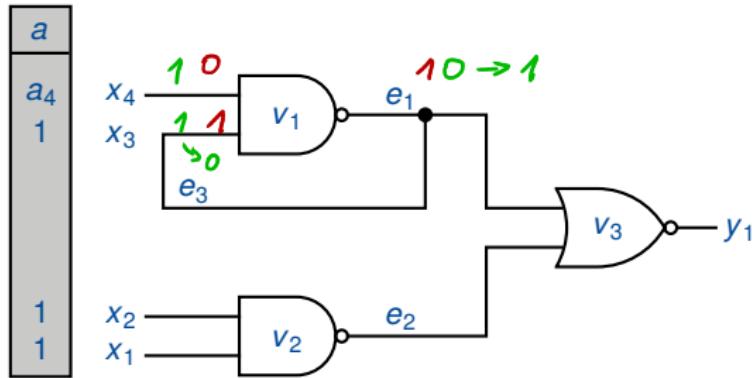


Schaltkreis mit Eingabebelegung  $a = (1, 1, 1, a_4)$ .



## Schaltkreis (2/2)

---



# Schaltpläne (1/3)

---

- Analyse von Schaltplänen  $SP = (\vec{X}_n, G, typ, IN, \vec{Y}_m)$  mit  $G$  nicht notwendigerweise azyklisch.

## Schaltpläne (2/3)

---

- Eine Zellenbibliothek  $BIB \subseteq \bigcup_{n \in \mathbb{N}} \mathbb{B}_n$  enthält Basisoperationen, die den Grundgattern entsprechen.
- Ein 5-Tupel  $SP = (\vec{X}_n, G, typ, IN, \vec{Y}_m)$  heißt Schaltplan mit  $n$  Eingängen und  $m$  Ausgängen über der Zellenbibliothek  $BIB$  genau dann, wenn
  - $\vec{X}_n = (x_1, \dots, x_n)$  ist eine endliche Folge von Eingängen.
  - $G = (V, E)$  ist ein gerichteter Graph mit  $\{0, 1\} \cup \{x_1, \dots, x_n\} \subseteq V$ .
  - Die Menge  $I = V \setminus (\{0, 1\} \cup \{x_1, \dots, x_n\})$  heißt Menge der Gatter.  
Die Abbildung  $typ : I \rightarrow BIB$  ordnet jedem Gatter  $v \in I$  einen Zellentyp  $typ(v) \in BIB$  zu.
  - ...

## Schaltpläne (3/3)

---

- ...
- Für jedes Gatter  $v \in I$  mit  $typ(v) \in B_k$  gilt  $indeg(v) = k$ .
- $indeg(v) = 0$  für  $v \in \{0, 1\} \cup \{x_1, \dots, x_n\}$ .
- Die Abbildung  $\underline{IN} : I \rightarrow E^*$  legt für jedes Gatter  $v \in I$  eine Reihenfolge der eingehenden Kanten fest, d.h. falls  $indeg(v) = k$ , dann ist  $IN(v) = (e_1, \dots, e_k)$  mit  $Z(e_i) = v \quad \forall 1 \leq i \leq k$ .
- Die Folge  $\vec{\underline{Y}}_m = (\underline{y}_1, \dots, \underline{y}_m)$  zeichnet Knoten  $y_i \in V$  als Ausgänge aus.

# Belegungen von Schaltplänen (1/2)

---

Sei nun ein Schaltplan  $SP = (\vec{X}_n, G, typ, IN, \vec{Y}_n)$  gegeben.

- Eine Abbildung  $\Phi_{SP,a} : V \rightarrow \{0, 1\}$  für  $a = (a_1, \dots, a_n) \in \mathbb{B}^n$  heißt **Belegung** für Eingabebelegung  $a$ , falls
  - $\Phi_{SP,a}(\underline{x_i}) = \underline{a_i} \quad \forall 1 \leq i \leq n,$
  - $\Phi_{SP,a}(0) = 0, \Phi_{SP,a}(1) = 1.$

## Belegung von Schaltplänen (2/2)

Interessant für uns sind **stabile** Belegungen.

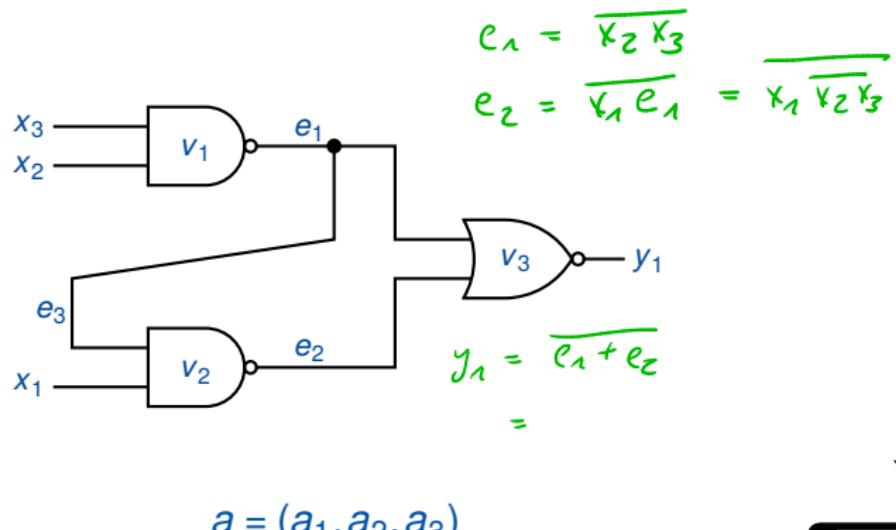
- Eine Belegung  $\Phi_{SP,a} : V \rightarrow \{0, 1\}$  heißt stabil, wenn gilt:
  - Für alle  $v \in I$  mit  $typ(v) = g \in B_k, IN(v) = (e_1, \dots, e_k)$  ist  $\Phi_{SP,a}(v) = g(\Phi_{SP,a}(Q(e_1)), \dots, \Phi_{SP,a}(Q(e_k)))$ .
  - $(\Phi_{SP,a}(y_1), \Phi_{SP,a}(y_2), \dots, \Phi_{SP,a}(y_n))$  heißt Ausgangsbelegung von  $SP$  bei Eingangsbelegung  $a$ .

Es ist möglich, dass es zu einer Eingangsbelegung  $a$

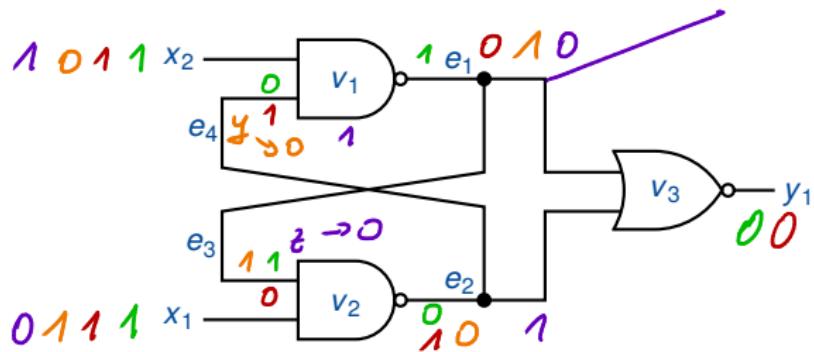
- **keine** stabile Signalbelegung  $\Phi_{SP,a}$  gibt,
- **mehrere** stabile Signalbelegungen  $\Phi_{SP,a}$  gibt.



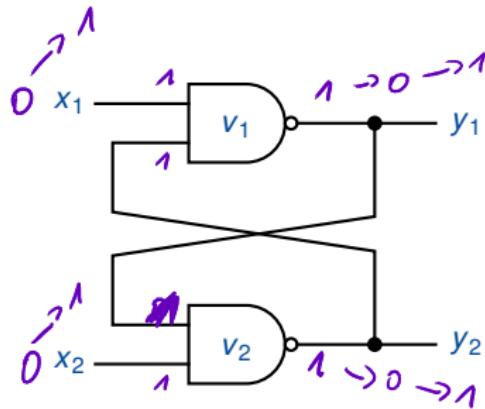
# Ein weiterer Schaltkreis (1/2)



## Ein weiterer Schaltkreis (2/2)



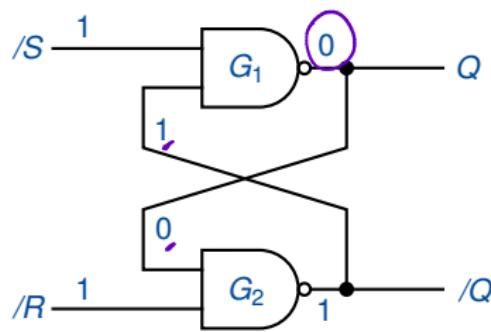
# Ausschnitt aus vorherigem Schaltplan



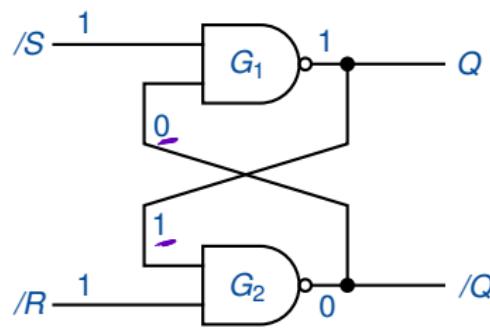
Hinweis: Gatter benötigen Zeit bis ihr “Ergebnis” am Ausgang bereit liegt. Was passiert wenn man die Eingabebelegung nach einer Zeit (mehrfach) ändert?

# RS-Flipflop (RS-FF)

- Die vorherige Schaltung heißt **RS-Flipflop**.
- Sie hat mindestens zwei stabile Zustände.



Zustand  $Q = 0$



Zustand  $Q = 1$

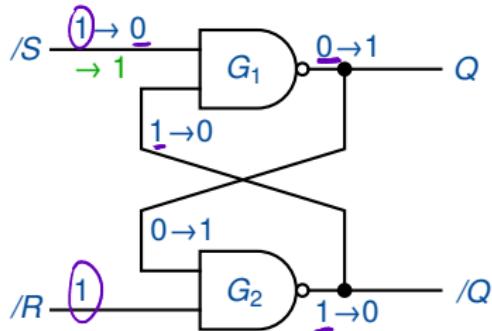
# Übergang (1/2)

---

- Für das Umschalten von einem Zustand zum anderen in einer realen Implementierung eines RS-FFs ist es von entscheidender Bedeutung, dass reale Gatter Verzögerungszeiten haben.
- D.h.: Wenn sich die Eingangsbelegung eines Gatters ändert, dann erfolgt die daraus resultierende Änderung des Ausgangswertes nicht direkt, sondern mit einer gewissen Verzögerung.
- (Detailliertere Betrachtung in Kapitel 5, Physikalische Eigenschaften von Gattern.)

# Übergang (2/2)

- Zustand  $Q = 0 \rightarrow$  Zustand  $Q = 1$ :



- Senke  $/S$  zur Zeit  $t_0$  ab und hebe zu  $t_0 + x$  wieder an (einen solchen Signalverlauf nennt man Puls).
- Nach Zeit  $t_{P/SQ}$  ist  $Q = 1$ .
- Nach Zeit  $t_{P/S/Q}$  ist  $/Q = 0$ .

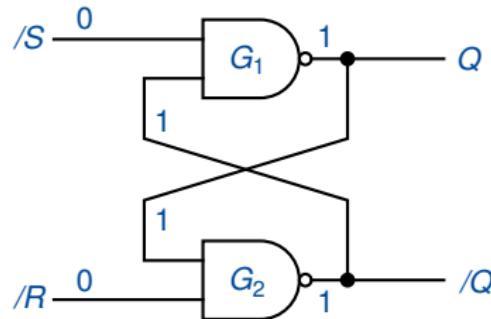
# Weitere Bezeichnungen

---

- Umschalten des FF in Zustand  $Q = 1$  heißt **Setzen** (set).
- Umschalten des FF in Zustand  $Q = 0$  heißt **Zurücksetzen** (reset).
- $/S$  heißt **Set-Signal**.
- $/R = /C$  heißt **Reset-** oder **Clear-Signal**.
- Weil  $/R, /S$  durch Absenken aktiviert werden, nennt man sie **active low**.
- Signalnamen von active-low-Signalen beginnen in der Regel mit /.

# “Zustand” $Q = 1, /Q = 1$

---

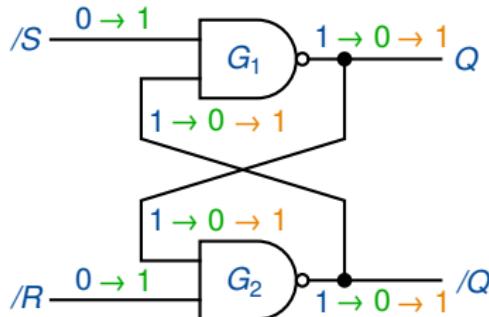


- Stabile Signalbelegung bei Eingangsbelegung  $/S = 0, /Q = 0$
- Aber warum ist es trotzdem problematisch,  $/S$  und  $/R$  gleichzeitig zu aktivieren ( $/S = 0, /R = 0$ )?

# Flackern

Annahme:

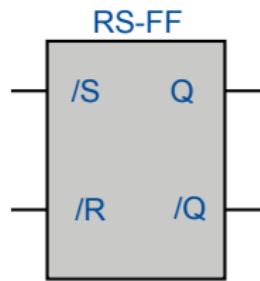
- $/S$  und  $/R$  werden nach ihrer Aktivierung beide gleichzeitig inaktiv ( $00 \rightarrow 11$ )
- $G_1$  und  $G_2$  schalten exakt gleich schnell, d.h. haben exakt die gleiche Verzögerungszeit



- $\Rightarrow$  Es kommt es zum Flackern ("metastabiler" Zustand).
- In der Praxis wird in der Regel nach einer gewissen Zeit einer der beiden stabilen Zustände angenommen (weil Gatterverzögerungen leicht variieren).

# Schaltsymbol eines RS-FF

---



# Nachteil von RS-FF

---

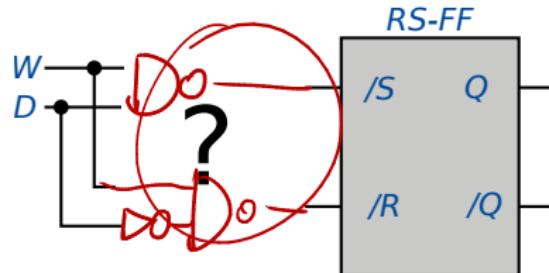
Beim Speichern eines Wertes **0** oder **1** muss man den Wert kennen:

- **0** → Aktiviere  $/R$
- **1** → Aktiviere  $/S$

## Ziel:

- Speichern unbekannter Werte.

# D-Latch (1/2)

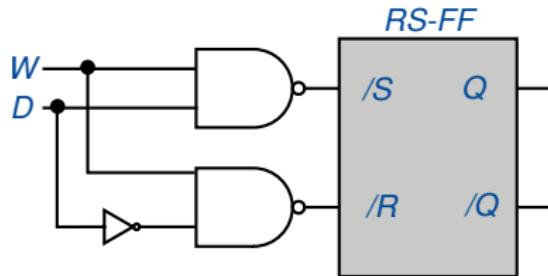


$W$	$D$	$/S$	$/R$
0	0		
0	1		
1	0		
1	1		

- $W$  ist active high.
  - $W = 0 \Rightarrow /S, /R$  inaktiv
  - $W = 1 \Rightarrow \begin{cases} /S \text{ aktiv, falls } D = 1 \\ /R \text{ aktiv, falls } D = 0 \end{cases}$

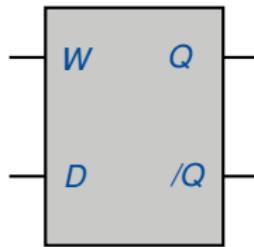


## D-Latch (2/2)



$W$	$D$	$/S$	$/R$
0	0	1	1
0	1	1	1
1	0	1	0
1	1	0	1

Symbol:

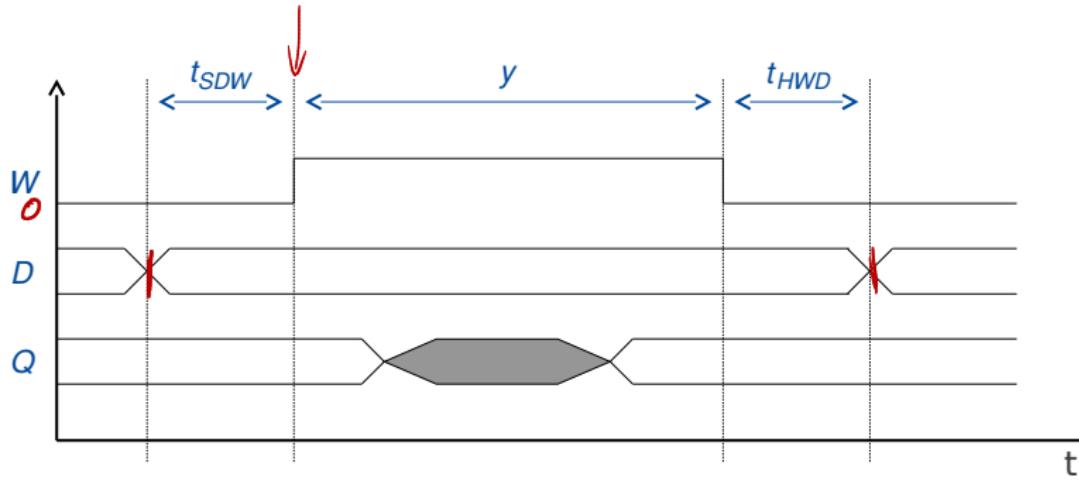


# Ansteuerung: Schreibimpuls

---

- Die Daten müssen für eine gewisse Zeit  $t_{SDW}$ , genannt **Setup-Zeit**, an  $D$  stabil anliegen.
- Dann geht  $W$  von 0 auf 1, bleibt für eine Zeit  $y$ , genannt **Pulsweite**, auf 1 und geht auf 0 zurück.
- Anschließend müssen die Daten für eine Zeit  $t_{HDW}$ , genannt **Hold-Zeit**, an  $D$  stabil gehalten werden.

# Timing-Diagramm



Wie lange müssen die einzelnen Signale aktiv sein, damit der Schreibvorgang reibungslos abläuft?

⇒ Siehe nächstes Kapitel ([Timing](#)).

# Weitere Eigenschaften eines D-Latches

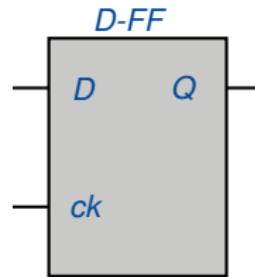
---

- Bisher: Keine Datenänderungen während des Schreibpulses auf  $W$ .
- Man kann das D-Latch aber auch im **transparenten Modus** betreiben:
  - Das D-Latch heißt transparent, wenn das Schreibsignal aktiv ist.
  - Hält man  $W$  lange aktiv und ändert  $D$  zur Zeit  $t$ , dann ändert sich  $Q$  zur Zeit  $t + t_{PDQ}$ .
  - Auch hier sind zeitliche Bedingungen zu beachten: Keine Datenänderungen kurz nach Beginn des Transparenzmodus bzw. kurz vor Ende des Transparenzmodus.

# Taktflankengesteuertes D-Flipflop (1/2)

- Taktflankengesteuerte Flipflops wie das D-Flipflop übernehmen Daten zu einem bestimmten Zeitpunkt (kein transparenter Modus!), nämlich bei der steigenden Flanke des Clocksignals.

D	ck	Q	/Q
0	¬	0	1
1	¬	1	0
X	0	Q	/Q
X	1	Q	/Q



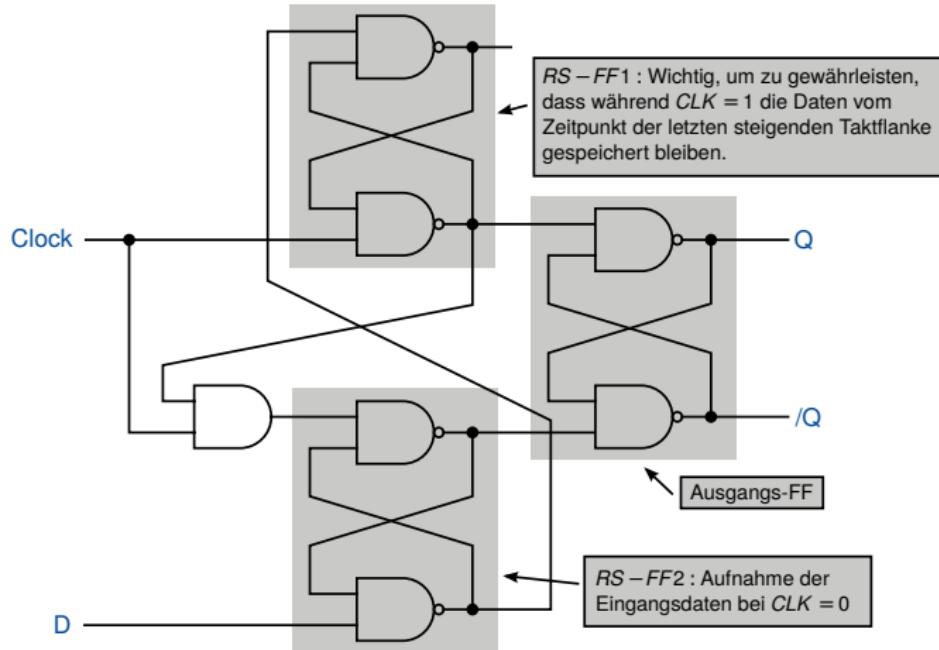
- Vorteil: Daten müssen lediglich bei der steigenden Taktflanke stabil sein (zzgl. Setup- und Holdzeit).

## Taktflankengesteuertes D-Flipflop (2/2)

---

- Realisierung: Wesentlich komplexer als bei taktzustandsgesteuerten D-Latches
- Analyse des Schaltplanes (und entsprechende Timing-Analyse) wesentlich komplizierter.

# D-FF: Realisierung mit RS-Flipflops



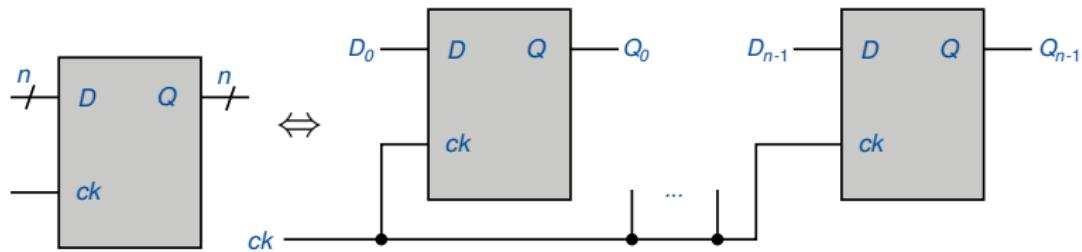
# Einfache Bausteine mit Flipflops

---

- Register
- Schieberegister
- Zähler

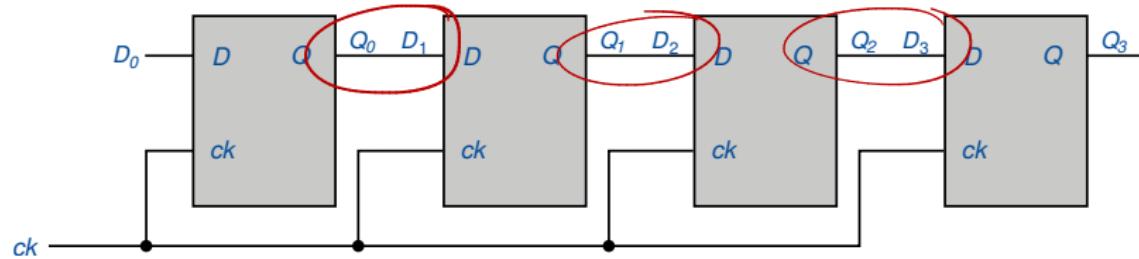
# $n$ -Bit Register

- $n$  D-Flipflops mit gemeinsamen Clocksignal.



- Entsprechend:  $n$ -Bit Latch =  $n$  D-Latches mit gemeinsamem Schreibsignal  $W$ .

# Schieberegister

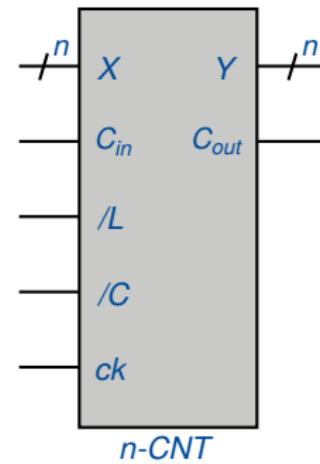


- In jedem Takt (bei jeder steigenden Flanke von  $ck$ ) werden die Werte im Register um eine Position nach rechts verschoben.

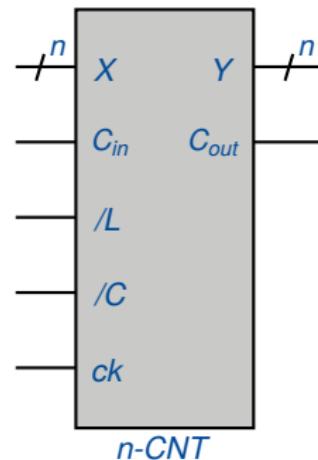
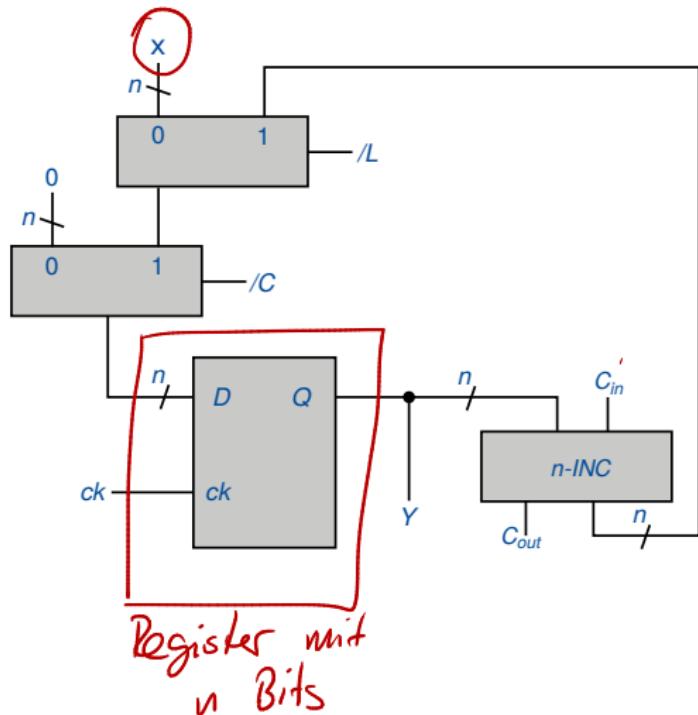
# Zähler

Ein ***n*-Bit-Zähler** ist eine Schaltung mit folgenden Ein- und Ausgängen:

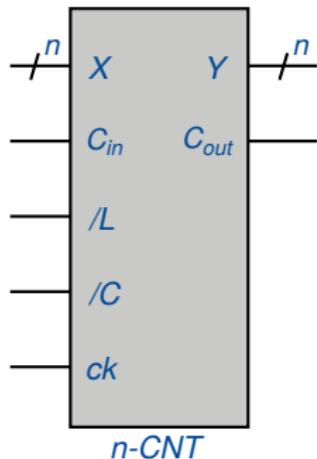
- Dateneingänge  $X = (X_{n-1}, \dots, X_0)$
- Datenausgänge  $Y = (Y_{n-1}, \dots, Y_0)$
- Dateneingang  $C_{in}$  für Eingangsübertrag
- Datenausgang  $C_{out}$  für Ausgangsübertrag
- Eingänge für Kontrollsignale:
  - $/C$  (Clear)
  - $/L$  (Load)
  - $ck$  (Clock)



# Aufbau eines Zählers



# $n$ -Bit Zähler: Funktionalität

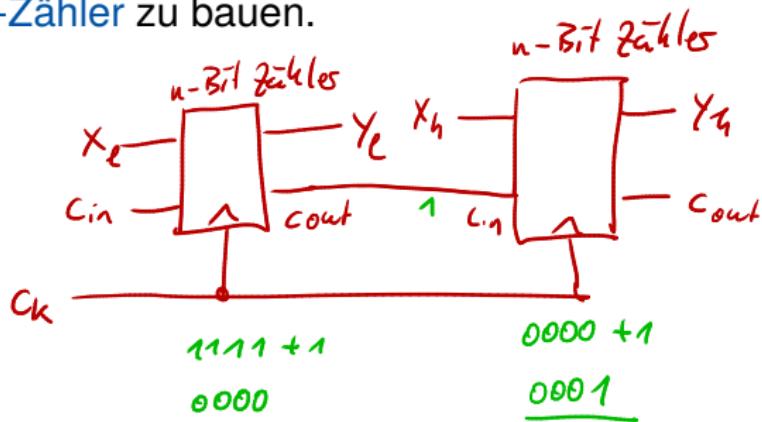


- Ein Zähler speichert ein  $n$ -Bit-Wort, das an den Ausgängen  $Y$  erscheint (**Zählerstand**).
- Bei jeder steigenden Flanke von  $ck$  wird ein neuer Zählerstand  $Y_{neu}$  gespeichert. Für  $Y_{neu}$  gilt :

$$Y_{neu} = \begin{cases} 0\dots0, & \text{falls } /C = 0 \\ X, & \text{falls } /C = 1, /L = 0 \\ \underline{\text{bin}_n((Y) + C_{in})mod2^n)}, & \text{falls } /C = 1, /L = 1 \end{cases}$$

# $n$ -Bit Zähler kaskadieren

- Ausgangsübertrag  $C_{out}$  ermöglicht es, den Zähler zu **kaskadieren**, zum Beispiel aus  $s$   $n$ -Bit-Zählern einen  $(s \cdot n)$ -Bit-Zähler zu bauen.





# Kapitel 4 – Sequentielle Logik

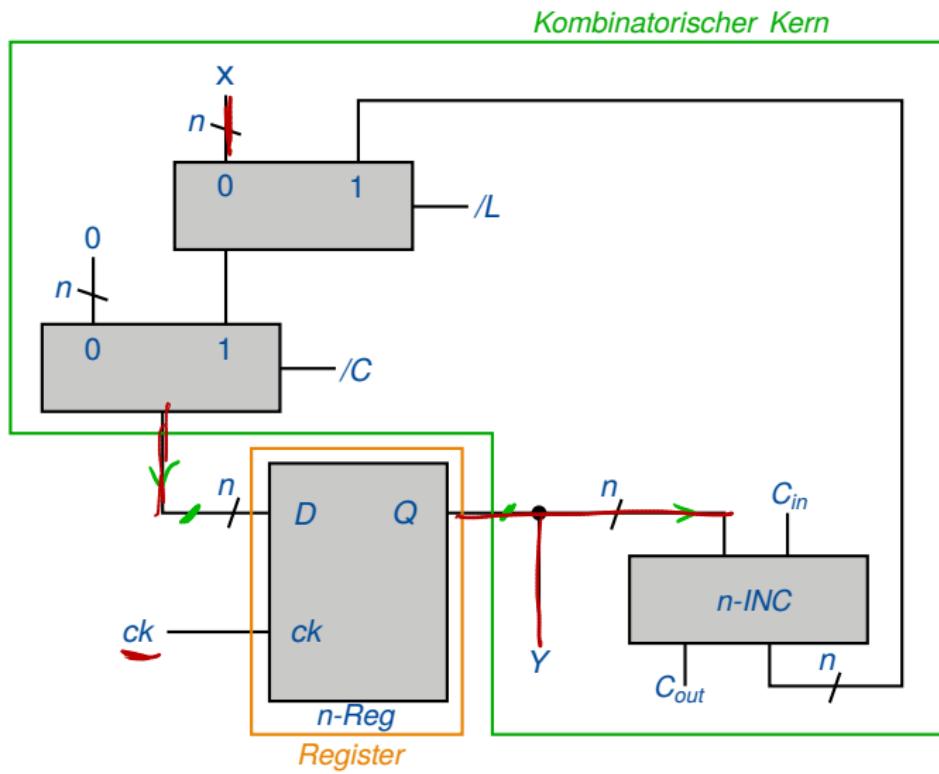
1. Speichernde Elemente
2. **Sequentielle Schaltkreise**
3. Entwurf sequentieller Schaltkreise
4. SRAM
5. Anwendung: Datenpfade von ReTI

# Sequentielle Schaltkreise

---

- Im Folgenden werden keine allgemeinen Schaltpläne mehr analysiert, sondern sogenannte **Schaltwerke** (auch (synchrone) **sequentielle Schaltkreise** genannt).
- Diese bestehen aus einem Register und einem (**kombinatorischen**) **Schaltkreis** (auch **kombinatorischer Kern** genannt).
- Im Gegensatz zu (**kombinatorischen**) Schaltkreisen können Schaltwerke (= sequentielle Schaltkreise) Zyklen enthalten. Die Zyklen müssen aber durch Flipflops des Registers gehen.
- Der Zustand eines Schaltwerkes ist gegeben durch die im Register gespeicherten Werte.
- Schaltwerke (= sequentielle Schaltkreise) entsprechen **endlichen Zustandsautomaten**.

# Beispiel: Zähler als sequentieller Schaltkreis



# Endliche Zustandsautomaten

---

- Endliche Zustandsautomaten (Finite State Machines, FSMs) sind ein Formalismus, um sequentielles (zeitabhängiges) Verhalten zu spezifizieren.
  - Mealy- und Moore-Automaten
  - In der theoretischen Informatik werden endliche Automaten mit akzeptierenden Zuständen betrachtet. Diese sind mit FSMs verwandt, aber nicht identisch.
- Aus einer FSM-Spezifikation kann der sequentielle Schaltkreis hergeleitet werden (Sequentielle Synthese).

## Definition

Das Quadrupel  $H = (I, S, S_0, \delta)$  heißt **deterministischer, endlicher Halbautomat**. Dabei bezeichnet:

- $I$  eine endliche Menge von erlaubten **Eingabesymbolen** ("Eingabealphabet"),
- $S$  eine endliche Menge von **Zuständen**,
- $S_0 \subseteq S$  ist eine endliche Menge von erlaubten **Anfangszuständen**,
- $\underline{\delta : S \times I \rightarrow S}$  eine **Übergangsfunktion**.

# Beispiel: Kaffeeautomat

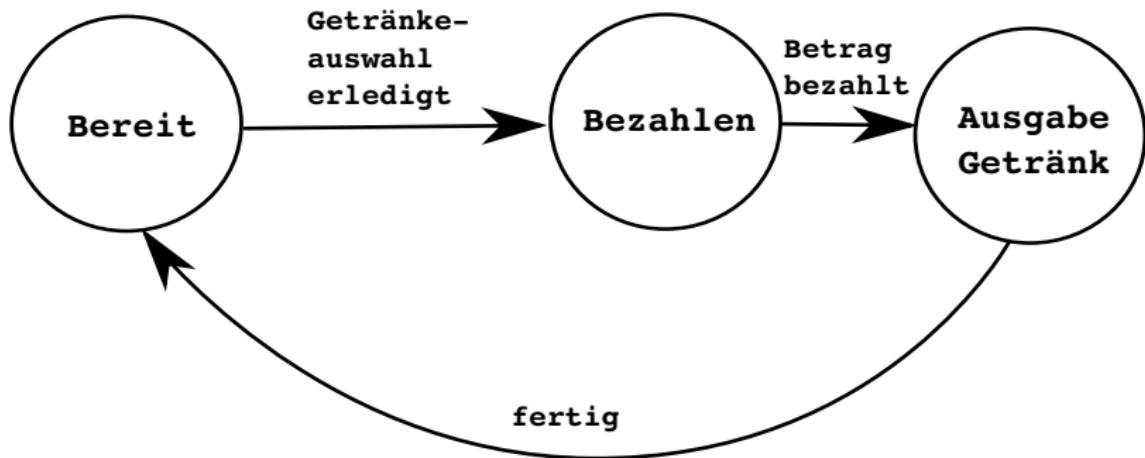
---



Auswahl:  
Kaffee  
Tee  
Zucker

korrekter  
Münzeinwurf  
oder Fertig

# Darstellung als Zustandsdiagramm



- **Knoten:** Zustände des Automaten.
- **Kanten:** Zustandsübergänge.
- **Kantenmarkierung:** Eingabe (bzw. Ereignis).

# Mealy- und Moore-Automat

## Definition

Ein **Mealy-Automat**  $M = (I, \underline{O}, S, S_0, \delta, \lambda)$  ist ein endlicher deterministischer Halbautomat  $H$  erweitert um:

- eine endliche Menge  $O$  von **Ausgabesymbolen** („Ausgabealphabet“),
- eine Ausgabefunktion  $\lambda : S \times I \rightarrow O$ .

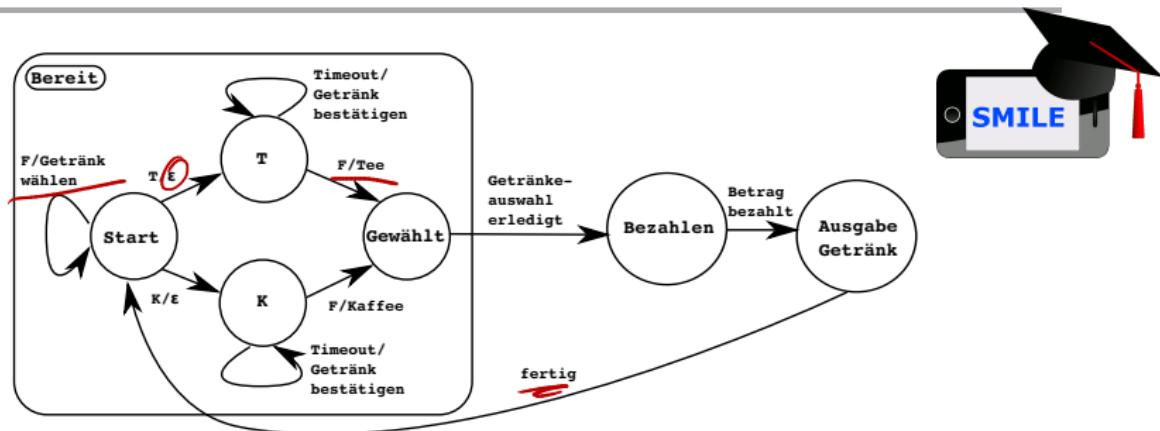
$$\delta : S \times I \rightarrow S$$

## Definition

Ein **Moore-Automat**  $M = (I, \underline{O}, S, S_0, \delta, \lambda)$  ist ein endlicher, deterministischer Halbautomat  $H$  erweitert um:

- eine endliche Menge  $O$  von **Ausgabesymbolen**,
- eine Ausgabefunktion  $\lambda : S \rightarrow O$ .

# Beispiel: Getränkeauswahl - Mealy-Automat



- **Knoten:** Zustände des Automaten.
- **Kanten:** Zustandsübergänge.
- **Kantenmarkierung:** Ein-/Ausgabe (Mealy-Automat).
- Beim Moore-Automaten werden die Zustände mit der **Ausgabe** beschriftet.
- **ε** bedeutet keine Eingabe/Ausgabe.

# Mealy- vs. Moore-Automat (1/2)

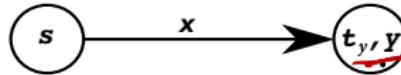
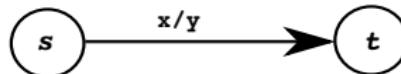
---

- Beim Mealy-Automaten ist:
  - die **Ausgabe** abhängig vom aktuellen Zustand **und** der aktuellen Eingabe,
  - der **Folgezustand** abhängig vom aktuellen Zustand und der aktuellen Eingabe.
- Ein **Moore-Automat** ist ein spezieller Mealy-Automat, bei dem die Ausgabe nur vom **aktuellen Zustand** und nicht von der Eingabe abhängt.
- Moore- und Mealy-Automaten kann man **ineinander überführen**.

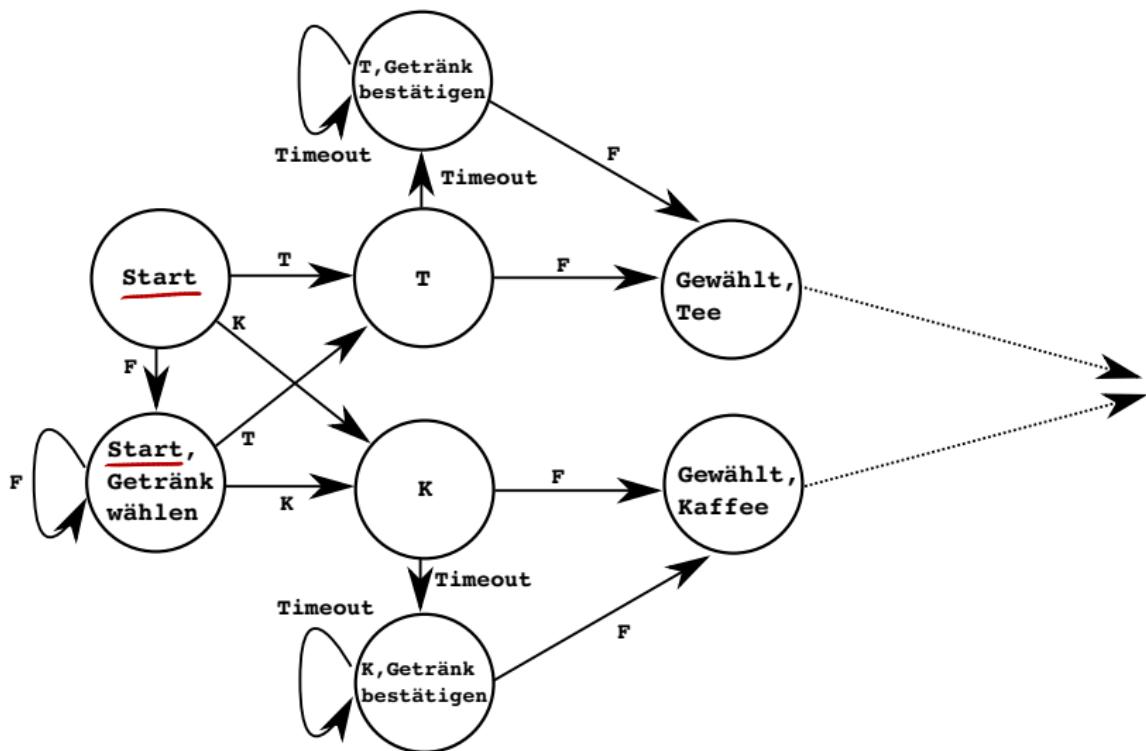
## Mealy- vs. Moore-Automat (2/2)

---

- Überführung Moore  $\rightarrow$  Mealy: trivial
- Überführung Mealy  $\rightarrow$  Moore:  
Grundidee: "Ziehe Ausgabe in den Zustand"



# Beispiel: Moore-Automat von Zustand “Bereit”



# Unterschiedliche Darstellungen von endlichen Zustandsautomaten

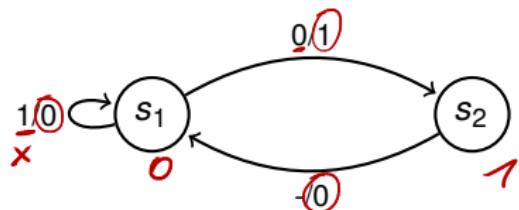
a) Zustands- und Ausgangstafel:

$x$	<i>state</i>	<i>next-state</i>	$y$
1	$s_1$	$s_1$	0
0	$s_1$	$s_2$	1
-	$s_2$	$s_1$	0

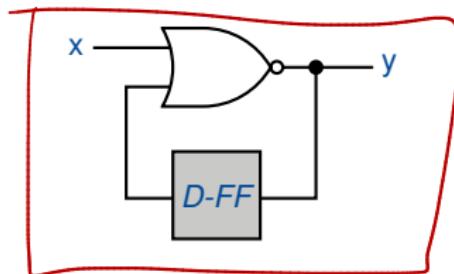
b) Flusstafel:

	$x = 0$	$x = 1$
$s_1$	$s_2, 1$	$s_1, 0$
$s_2$	$s_1, 0$	$s_1, 0$

c) Zustandsdiagramm:



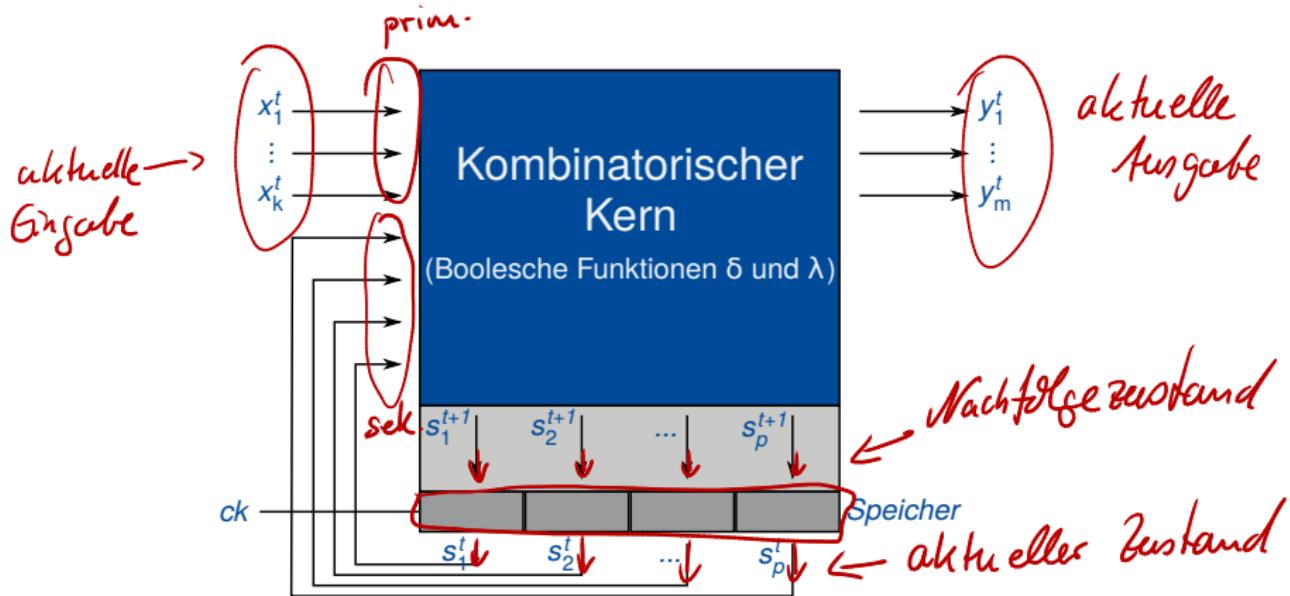
d) Sequentieller Schaltkreis:



■ Im Folgenden: Weg von c) zu d)

$$y = \overline{s+x}$$

# Sequentielle Schaltkreise allgemein



$$\begin{aligned} \underline{y_i^t} \\ s_i^{t+1} \end{aligned} = \lambda_i(x_1^t, x_2^t, \dots, x_k^t, s_1^t, s_2^t, \dots, s_p^t)$$
$$= \delta_i(x_1^t, x_2^t, \dots, x_k^t, s_1^t, s_2^t, \dots, s_p^t)$$

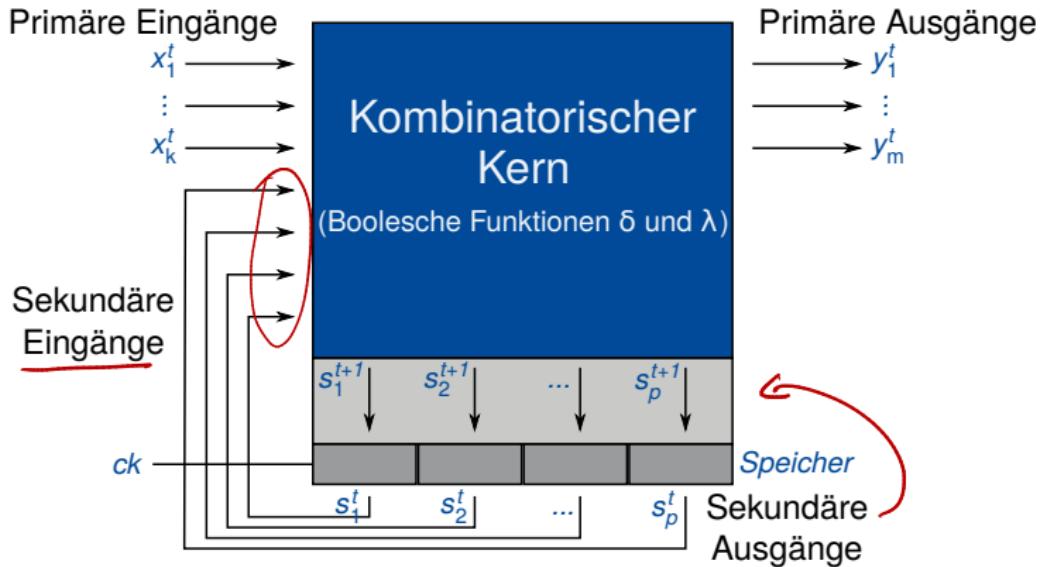
Die Belegung  $s_t$  der Flipflops im Register heißt **Zustand** des sequentiellen Schaltkreises zum **Zeitpunkt**  $t$ .

# Kombinatorischer Kern

---

- Der kombinatorische Kern hat vier Arten von Ein- und Ausgängen:
  - Primäre Eingänge bekommen Werte „von außen“.
  - Primäre Ausgänge liefern Werte „nach außen“.
  - Sekundäre Eingänge sind mit den Datenausgängen der Flipflops im Register verbunden. Auf diese Weise kann der aktuelle Zustand des Schaltkreises in Funktionen  $\delta$  und  $\lambda$  berücksichtigt werden.
  - Sekundäre Ausgänge sind mit den Dateneingängen der Flipflops verbunden. Durch sie wird der nächste Zustand des Schaltkreises spezifiziert.

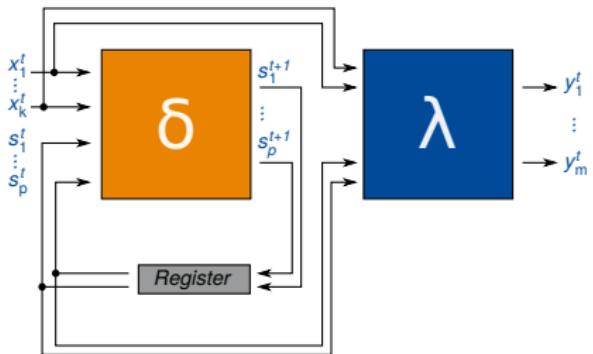
# Primäre und sekundäre Ein- und Ausgänge



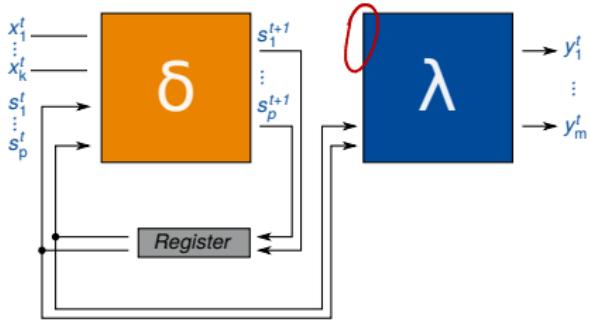
$$\begin{aligned}y_i^t &= \lambda_i(x_1^t, x_2^t, \dots, x_k^t, s_1^t, s_2^t, \dots, s_p^t) \\s_i^{t+1} &= \delta_i(x_1^t, x_2^t, \dots, x_k^t, s_1^t, s_2^t, \dots, s_p^t)\end{aligned}$$

# Sequentielle Schaltung für einen FSM

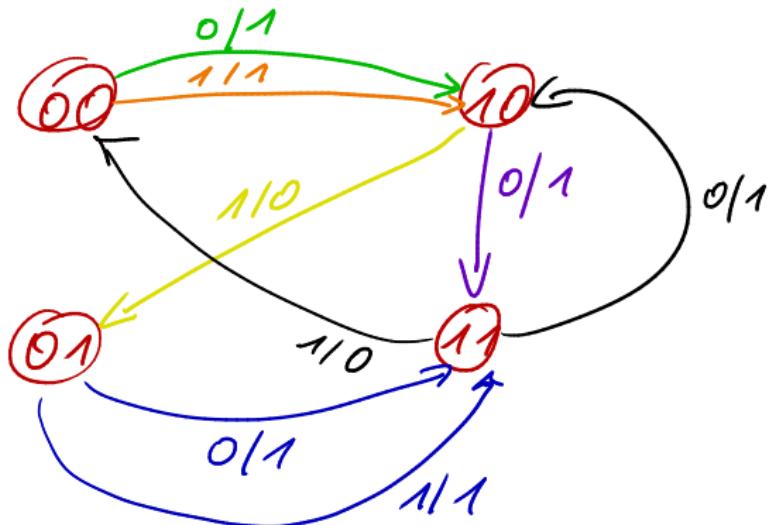
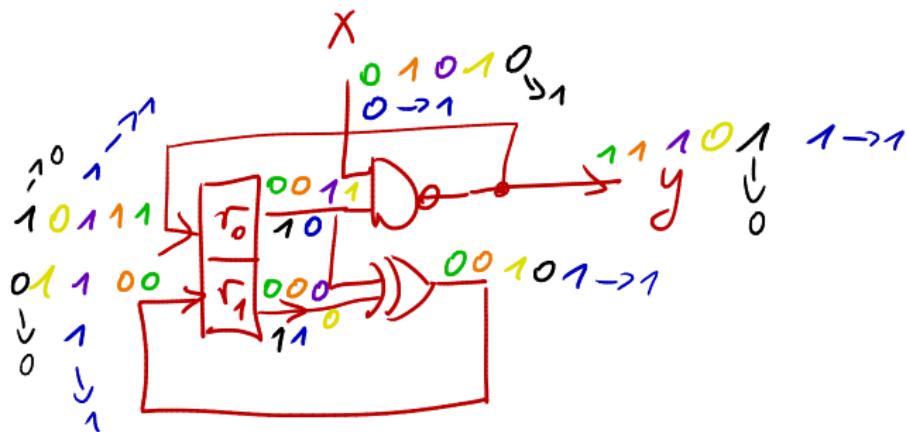
Mealy-Automat



Moore-Automat



- Eingabevektor:  $X = (x_1, x_2, \dots, x_k)$
- Ausgabevektor:  $Y = (y_1, y_2, \dots, y_m)$
- Zustandsvektor:  $S = (s_1, s_2, \dots, s_p)$
- Ausgabefunktion (Mealy):  $Y^t = \underline{\lambda(X^t, S^t)}$
- Übergangsfunktion:  $S^{t+1} = \delta(X^t, S^t)$
- Ausgabefunktion (Moore):  $Y^t = \underline{\lambda(S^t)}$





# Kapitel 4 – Sequentielle Logik

1. Speichernde Elemente
2. Sequentielle Schaltkreise
- 3. Entwurf sequentieller Schaltkreise**
4. SRAM
5. Anwendung: Datenpfade von ReTI

.

Albert-Ludwigs-Universität Freiburg

UNI  
FREIBURG

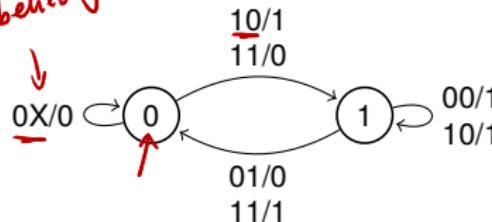
Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur  
WS 2016/17

# Entwurf sequentieller Schaltkreise

$x =$   
beliebig

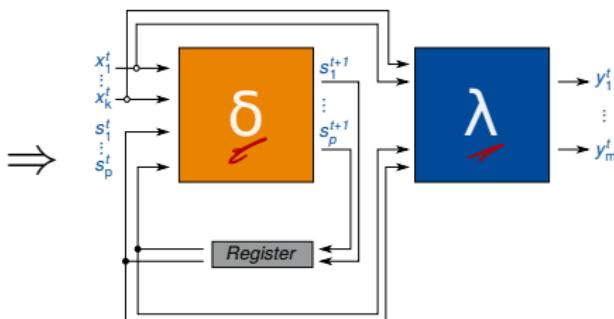
Zustandsdiagramm:



Zustands- und Ausgangstafel:

$s^t$	$x_1^t$	$x_2^t$	$s^{t+1}$	$y^t$
0	0	0	0	0
0	0	1	0	0
0	1	0	1	1
0	1	1	1	0
1	0	0	1	1
1	0	1	0	0
1	1	0	1	1
1	1	1	0	1

Sequentieller Schaltkreis:



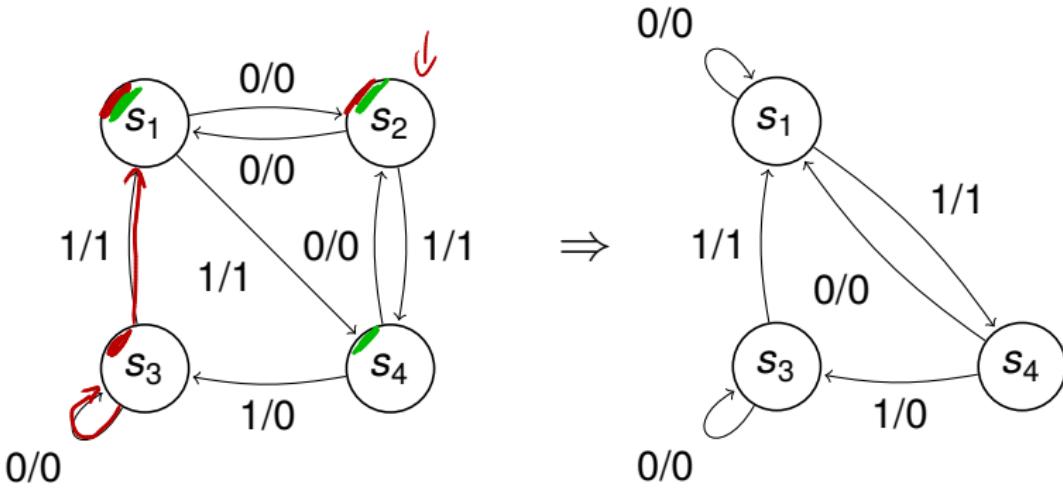
- Optimierung des Zustandsdiagramms:  
**Zustandsminimierung**
  - Identifikation der äquivalenten Zustände.
  - Ergebnis: Ein (evtl. kleineres) Zustandsdiagramm.
- Wahl der **Zustandskodierung**.
  - Ergebnis: Anzahl der Flipflops im Register,  
Funktionen  $\delta$  und  $\lambda$  (Zustands- und Ausgangstafel).
- Implementierung von  $\delta$  und  $\lambda$ .
  - Kombinatorische Logiksynthese, z.B. Quine-McCluskey.

# Zustandsminimierung

## Idee:

Bestimme und verschmelze äquivalente Zustände.

- Zwei Zustände sind **äquivalent**, wenn der Automat von ihnen aus bei gleichen Eingabefolgen stets die gleichen Ausgabefolgen produziert.

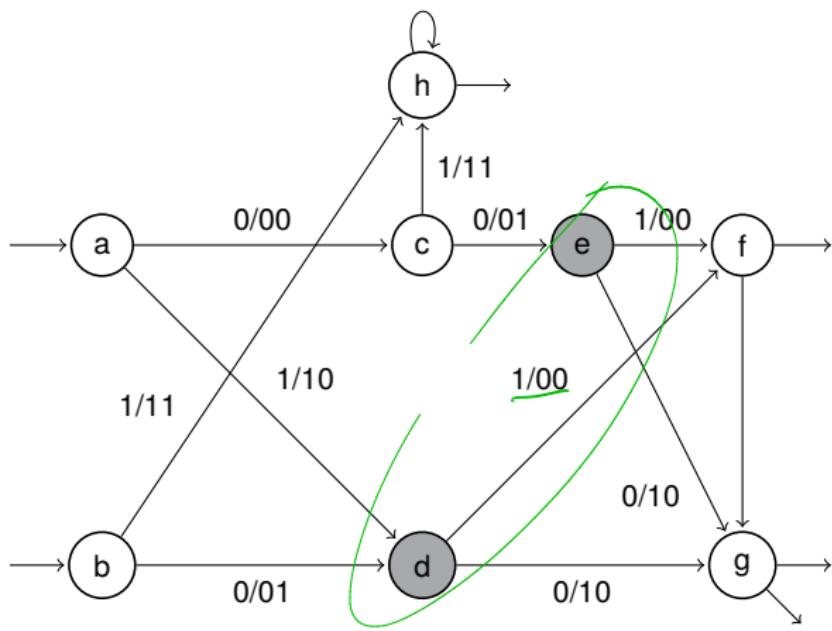


## Weiteres Beispiel (1/4)

---

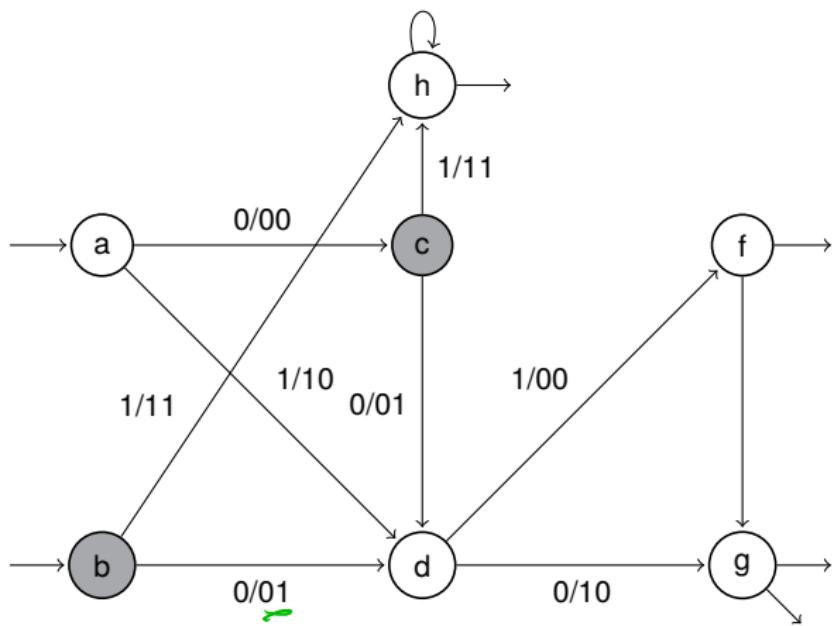
- **Hinreichende Bedingung:** Wenn bei zwei Zuständen bei gleicher Eingabe auch die gleiche Ausgabe erzeugt wird und der gleiche Folgezustand angenommen wird, dann sind die Zustände sicherlich äquivalent.
- Äquivalente Zustände können durch einen einzigen Zustand ersetzt werden (siehe nächste Folie).

## Weiteres Beispiel (2/4)



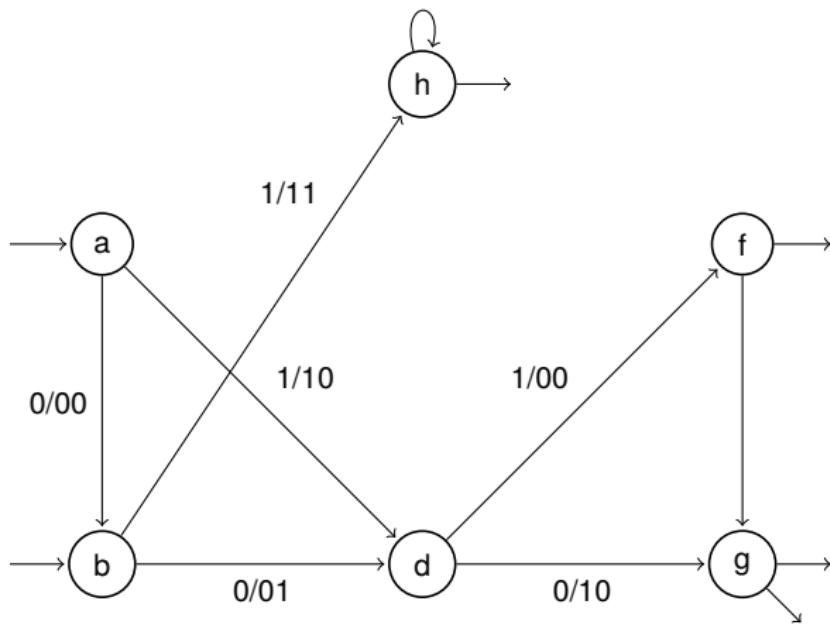
Zustand e und d sind äquivalent.

## Weiteres Beispiel (3/4)



Zustand  $e$  eliminiert.  
Zustand  $b$  und  $c$  sind äquivalent.

## Weiteres Beispiel (4/4)



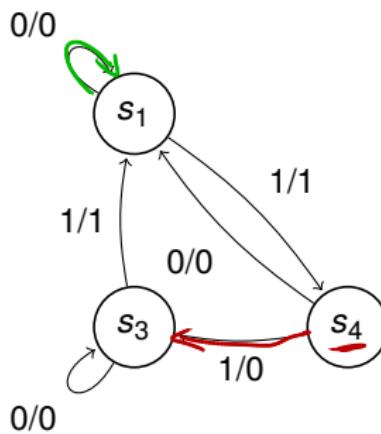
Zustand c eliminiert.

# Entwurfsschritte

---

- Optimierung des Zustandsdiagramms:  
**Zustandsminimierung**
  - Identifikation der äquivalenten Zustände.
  - Ergebnis: Ein (evtl. kleineres) Zustandsdiagramm.
- Wahl der **Zustandskodierung**.
  - Ergebnis: Anzahl der Flipflops im Register,  
Funktionen  $\delta$  und  $\lambda$  (Zustands- und Ausgangstafel).
- Implementierung von  $\delta$  und  $\lambda$ .
  - Kombinatorische Logiksynthese, z.B. Quine-McCluskey.

# Zustandskodierung



Kodierung  $S_1 \equiv 00, S_3 \equiv 10, S_4 \equiv 01 : 9$  Literale

$$\begin{aligned}\delta_1(s, i) &= s_2 i + s_1 \bar{i} \\ \delta_2(s, i) &= \bar{s}_1 \bar{s}_2 i \\ \lambda(s, i) &= \bar{s}_2 i\end{aligned}$$

$s$	$x$	$s'$	$y$
00	0	00	0
01	1	10	0

Kodierung  $S_1 \equiv 01, S_3 \equiv 11, S_4 \equiv 10 : 11$  Literale

$$\begin{aligned}\delta_1(s, i) &= s_1 s_2 \bar{i} + \bar{s}_1 i + \bar{s}_2 i \\ \delta_2(s, i) &= s_1 + i \\ \lambda(s, i) &= s_2 i\end{aligned}$$

$s$	$x$	$s'$	$y$
01	0	01	0

- **Ziel:** Wähle Zustandskodierung, die nachfolgende kombinatorische Synthese erleichtert.
- Dafür gibt es (heuristische) Verfahren.



- Aufgabenbeschreibung (**Textspezifikation**):  
Modulo-4 Vorwärts/Rückwärtzzähler
  - Der Zähler soll von **0** bis **3** zählen können.
  - Ist der Steuereingang **x** auf **1** gesetzt, so soll vorwärts gezählt werden, d.h. die Zahlenfolge **0,1,2,3** durchlaufen werden.
  - Ist **x** auf **0** gesetzt, so soll rückwärts gezählt werden, d.h. die Zahlenfolge **3,2,1,0** durchlaufen werden.
  - Am Ausgang ist der **Zählerstand** anzugeben (Ausgabevektor  **$y_0, y_1$** ). Der Zähler ist als Ringzähler zu realisieren.

# SMILE - Zustände im Zustandsdiagramm

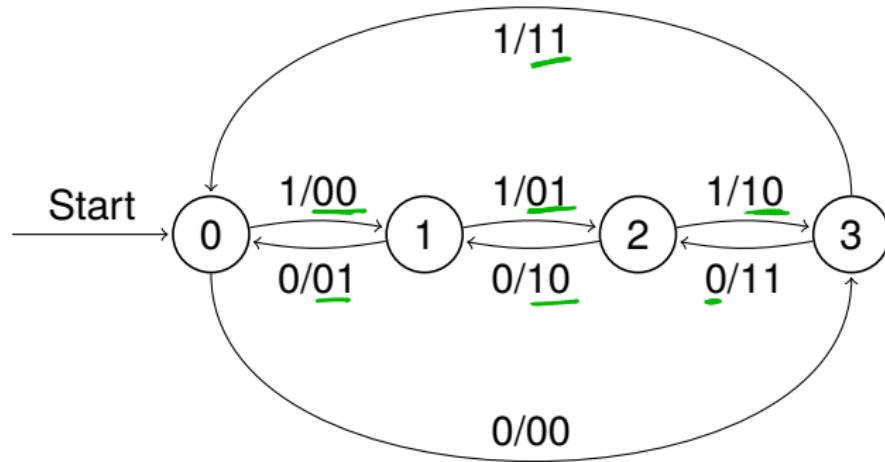
---

Wie viele Zustände benötigt das Zustandsdiagramm für den Zähler bei der gegebenen Spezifikation?

- a. 1
- b. 2
- c. 4
- d. 8
- e. 16

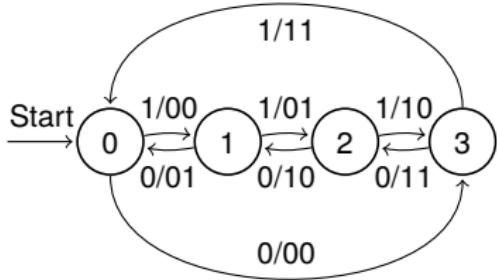
# Von der Textspezifikation zum Zustandsdiagramm

- 4 Zustände erforderlich.
- Startzustand 0.



# Vom Zustandsdiagramm zur Zustands- und Ausgangstafel

- Zustandsminimierung  $\Rightarrow$  Keine äquivalente Zustände.
- Zustandskodierung:  $0 \rightarrow \underline{00}, 1 \rightarrow \underline{01}, 2 \rightarrow \underline{10}, 3 \rightarrow \underline{11}$ .



	$x$	$z_1^t$	$z_0^t$	$z_1^{t+1}$	$z_0^{t+1}$	$y_1$	$y_0$
Vorwärts-zählen	1	0	0	0	1	0	0
	1	0	1	1	0	0	1
	1	1	0	1	1	1	0
	1	1	1	0	0	1	1
Rückwärts-zählen	0	1	1	1	0	1	1
	0	1	0	0	1	1	0
	0	0	1	0	0	0	1
	0	0	0	1	1	0	0

$$z_1^{t+1} = \overline{x} \overline{z}_1 \overline{z}_0 + \overline{x} z_1 \overline{z}_0 + x z_1 \overline{z}_0 + x \overline{z}_1 \overline{z}_0$$

# Implementierung des kombinatorischen Kerns

	x	$z_1^t$	$z_0^t$	$z_1^{t+1}$	$z_0^{t+1}$	$y_1$	$y_0$
Vorwärts-zählen	1	0	0	0	1	0	0
	1	0	1	1	0	0	1
	1	1	0	1	1	1	0
	1	1	1	0	0	1	1
Rückwärts-zählen	0	1	1	1	0	1	1
	0	1	0	0	1	1	0
	0	0	1	0	0	0	1
	0	0	0	1	1	0	0

$$\text{Übergangsfunktion: } z_0^{t+1} = x\bar{z}_1\bar{z}_0^t + xz_1^t\bar{z}_0 + \bar{x}z_1^t\bar{z}_0^t + \bar{x}\bar{z}_1^t\bar{z}_0$$

$$z_1^{t+1} = x\bar{z}_1z_0^t + xz_1^t\bar{z}_0 + \bar{x}z_1^tz_0^t + \bar{x}\bar{z}_1^t\bar{z}_0$$

# Implementierung des komb. Kerns: Logikminimierung

---

Ausgangsfunktion:  $\underline{y_0^t} = z_0^t, \quad y_1^t = z_1^t$

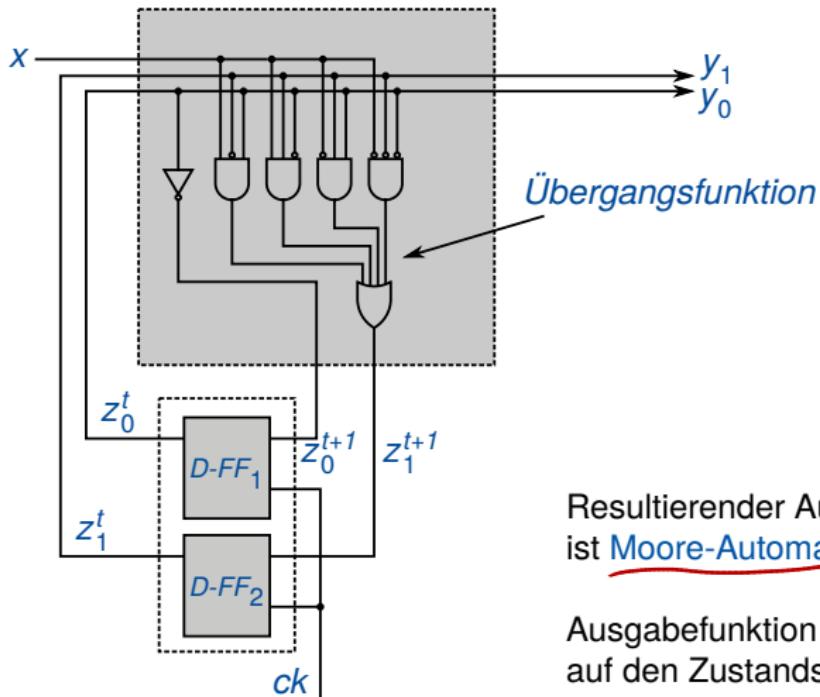
Übergangsfunktion:  $z_0^{t+1} = x\bar{z}_1^t z_0^t + xz_1^t \bar{z}_0^t + \bar{x}z_1^t \bar{z}_0^t + \bar{x}\bar{z}_1^t z_0^t$  ||  
 $z_1^{t+1} = \underline{x\bar{z}_1^t z_0^t} + \underline{xz_1^t \bar{z}_0^t} + \underline{\bar{x}z_1^t z_0^t} + \underline{\bar{x}\bar{z}_1^t z_0^t}$  |||

Minimierung:

$$z_0^{t+1} = \underline{\bar{z}_0^t}$$

$$\underline{z_1^{t+1}} = x\bar{z}_1^t z_0^t + xz_1^t \bar{z}_0^t + \bar{x}z_1^t z_0^t + \bar{x}\bar{z}_1^t z_0^t$$

# Beispiel: Ergebnis



Resultierender Automat  
ist Moore-Automat.

Ausgabefunktion ist Identität  
auf den Zustandsbits.

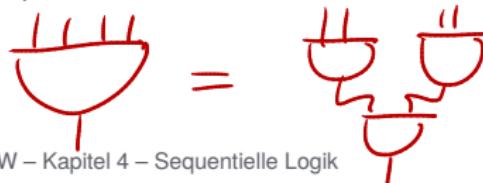


# Kapitel 4 – Sequentielle Logik

1. Speichernde Elemente
2. Sequentielle Schaltkreise
3. Entwurf sequentieller Schaltkreise
- 4. SRAM**
5. Anwendung: Datenpfade von ReTI

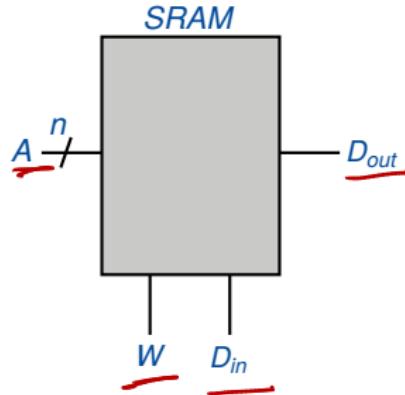
## ■ Static Random-Access Memory.

- Konzeptuell: Eine (sehr große) Anzahl  $N$  von Speicherzellen sowie ein (sehr großer) Multiplexer, um auf die einzelnen Zellen zuzugreifen.
- Durch die Größe spielen Beschränkungen eine Rolle, die bei einer Realisierung beachtet werden müssen.
  - **Fanout-Beschränkung:** Eine Leitung kann nicht beliebig verzweigen  $\Rightarrow$  Treiberbäume
  - **Fanin-Beschränkung:** Ein Gatter kann nicht beliebig viele Eingänge haben (Hier:  $N$ -faches ODER als Baum).



# SRAM: Ein-/Ausgänge und Zeichen

- Sei  $n \in \mathbb{N}, N = 2^n$ . Ein  $N$ -Bit statischer Speicher oder **SRAM** hat:
  - $n$  Eingänge  $A = (A_{n-1}, \dots, A_0)$  „Adresse“,
  - Dateneingang  $D_{in}$ ,
  - Datenausgang  $D_{out}$ ,
  - Kontrollsignal  $W$  „write“



# SRAM: Funktionalität

---

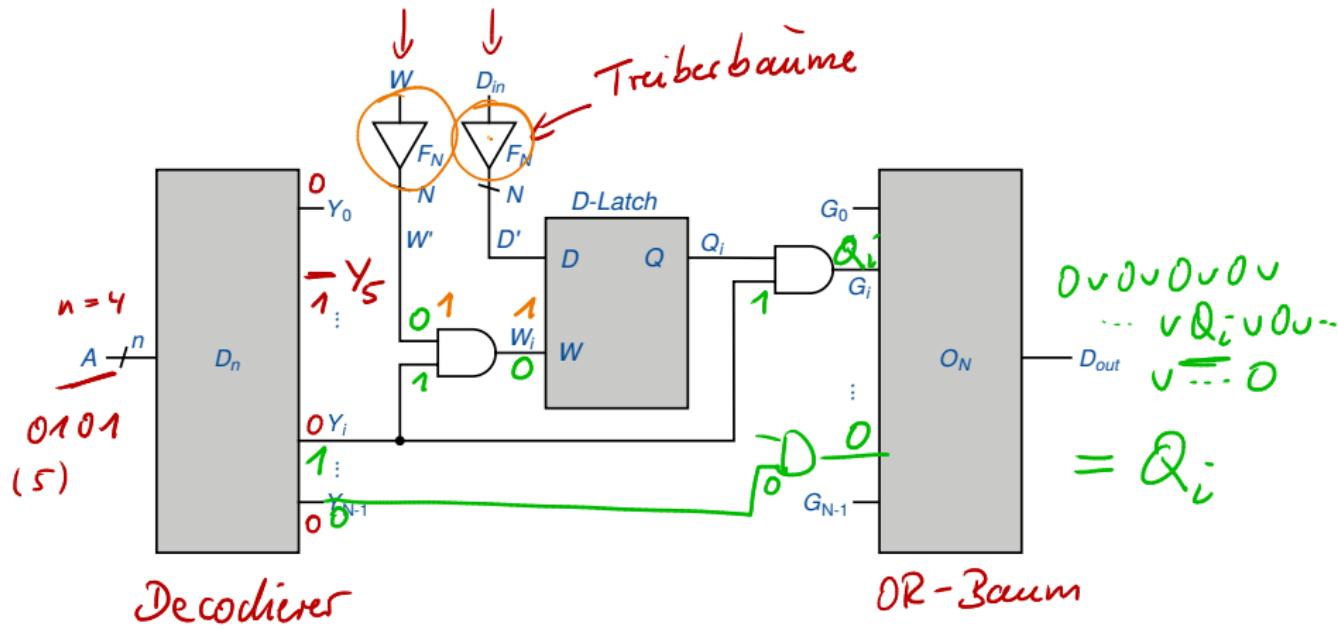
- Der Speicher enthält  $N$  Speicherzellen  $L_0, \dots, L_{N-1}$ , die je ein Bit speichern können.
- Zelle  $L_{\langle A \rangle}$  wird mit Hilfe der Adresse  $A$  ausgewählt.
  - An  $D_{out}$  erscheint der Inhalt von  $L_{\langle A \rangle}$ .
  - Durch Schreibpuls an  $W$  wird  $D_{in}$  nach  $L_{\langle A \rangle}$  übernommen.

# $N$ -Bit-SRAM, $N \times s$ -Bit-SRAM: Aufbau

---

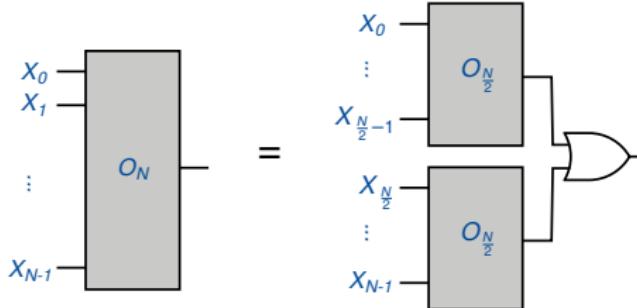
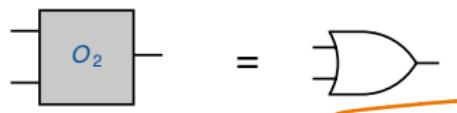
- Ein  $N \times s$ -Bit-SRAM besteht aus  $s$   $N$ -Bit SRAMs mit gemeinsamen Adress- und Schreibsignalen.
  - $s$  heißt Bitbreite des  $N \times s$ -Bit-SRAMs.
- Ein  $N$ -Bit-SRAM besteht im Prinzip aus 3 Hilfsschaltkreisen:
  - mehrfaches ODER
  - Treiberbäume
  - Dekodierer

# SRAM: Schaltbild

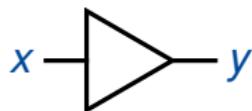


# $O_N$ : Mehrfaches ODER

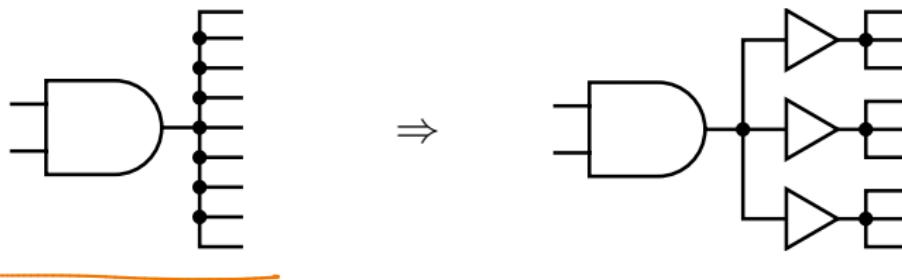
- Ein **N-faches ODER**  $O_N$  mit  $N = 2^n$  ist ein Schaltkreis, der **N-faches Oder** berechnet.
- **Balancierter Baum**, um Verzögerungszeit zu minimieren (Tiefe  $O(\log N) = \underline{O(n)}$ ).



# Treiberbäume



- Ein **Treiber** ist ein Gatter mit einem Eingang  $X$  und einem Ausgang  $Y$ , das die Identität  $Y = X$  berechnet.
  - Eingesetzt, um **Fanout-Beschränkung** zu überwinden.
  - Beispiel: Fanout-Beschränkung von 3.



# $F_N$ : Treiberbäume im SRAM

- Zur Erinnerung: Ein **Baum** ist ein azyklischer gerichteter Graph  $G = (V, E)$  mit:
  - Genau einer Quelle w,
  - $\text{indeg}(v) = 1$  für alle  $v \in V \setminus \{w\}$
  - Blätter = Knoten  $v \in V$  mit  $\text{outdeg}(v) = 0$
  - Innere Knoten = Knoten  $v \in V$  mit  $\text{outdeg}(v) \geq 1$ .
- Im SRAM für Realisierung von  $F_N$  eingesetzt.
- Fanout-Beschränkung von 10 → 10-äre Bäume.

# 10-äre Bäume (1/2)

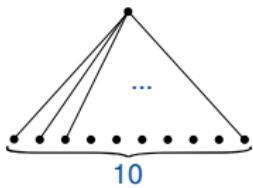
$B_0$ :

•

- Anzahl der Blätter von  $B_s$ :  $L(B_s)$ .

- Entspricht dem erreichten Verzweigungsgrad.

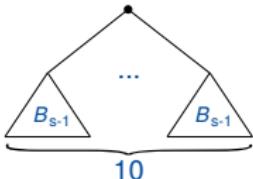
$B_1$ :



- Anzahl der inneren Knoten von  $B_s$ :  $I(B_s)$ .

- Entspricht der benötigten Anzahl von Treibern.

$B_s$ :



$$L(B_s) = 10^s$$

$$I(B_s) = \sum_{i=0}^{s-1} 10^i = \frac{10^s - 1}{10 - 1} < \frac{L(B_s)}{9}$$

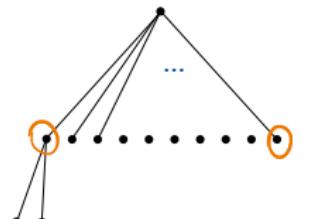
## 10-äre Bäume (2/2)

---

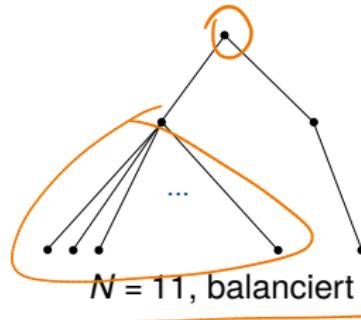
- Benutze also  $B_s$  zum  $10^s$ -fachen Vervielfältigen eines Signals
- Innere Knoten des Baumes werden durch Treiber ersetzt
- ⇒ **Treiberbaum** mit Fanoutbeschränkung 10

# Treiberäume: Allgemeiner Fall (1/2)

- Angenommen, ein Signal soll  $N$ -fach vervielfältigt werden mit  $10^{s-1} < N < 10^s$  keine Zehnerpotenz.
- Ziel: **Balancierte Treiberäume**, d.h. alle Pfade von der Wurzel zu einem Blatt haben gleiche Länge.



$N = 11$ , unbalanciert



$N = 11$ , balanciert

# Treiberäume: Allgemeiner Fall (2/2)

- **Idee:** "Fülle Bäume von links her 10-är auf" und sorge zusätzlich für gleiche Tiefe der Blätter!
- Beispiel: ...

## Lemma

$\forall s \in \mathbb{N}$  und  $N \in \{10^{s-1} + 1, \dots, 10^s\}$  gibt es einen Baum  $T(N)$  mit Ausgangsgrad  $\leq 10$  an jedem inneren Knoten und den folgenden Eigenschaften:

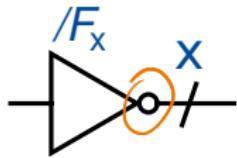
- 1  $T(N)$  hat  $N$  Blätter.
- 2  $T(N)$  hat  $\leq \frac{N}{9} + s$  innere Knoten.
- 3 Alle Pfade von der Wurzel zu einem Blatt haben genau die Länge  $s = \lceil \log_{10} N \rceil$  mit  $\lceil \log_{10} N \rceil < \frac{1}{3} \log_2 N + 1$ .

- Beweis: Induktion über  $s \Rightarrow$  Übung

# Notation: Invertierender Treiberbaum

---

- Invertierender Treiberbaum  $/F_x$ :  
Ersetze den Treiber an der Wurzel durch einen Inverter.



# $D_n$ : Dekodierer

---

- Sei  $n \in \mathbb{N}$ ,  $N = 2^n$ . Ein  $n$ -Bit-Dekodierer  $D_n$  ist ein Schaltkreis, der die Funktion  $d : \mathbb{B}^n \rightarrow \mathbb{B}^N$  berechnet, wobei gilt:

$$d_i(a) = \begin{cases} 1, & \text{falls } \langle a \rangle = i \\ 0, & \text{sonst} \end{cases} \quad \forall i = 0, \dots, N - 1$$

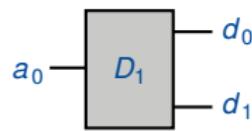
( $d_i(a)$  ist Bit  $i$  des  $N$ -Tupels  $d(a)$ .)

- Induktive Konstruktion von  $D_n$ : Siehe nächste Folie.

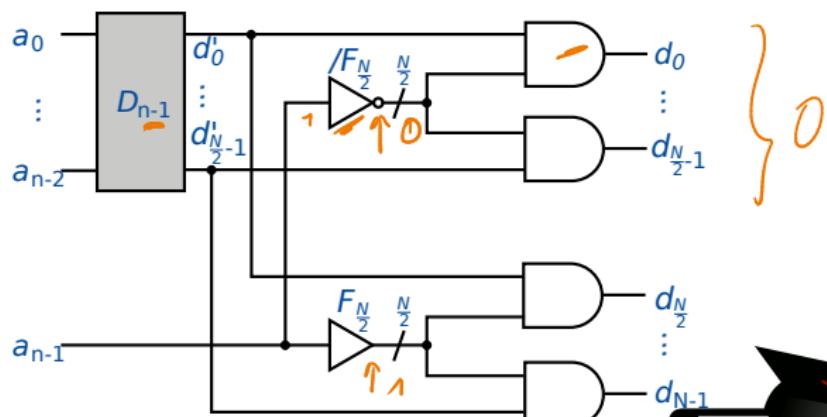
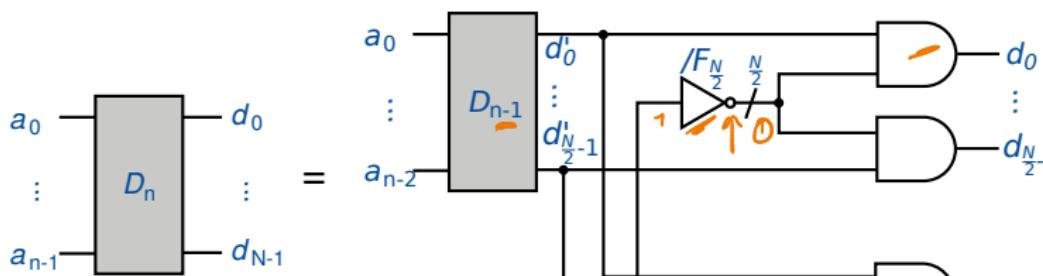
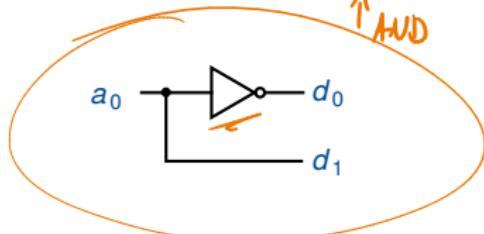
# Dekodierer: Rekursiver Aufbau

$$C(\mathcal{D}_1) = 1$$

$$C(\mathcal{D}_n) = C(\mathcal{D}_{n-1}) + 2^n + O(2^n)$$



=

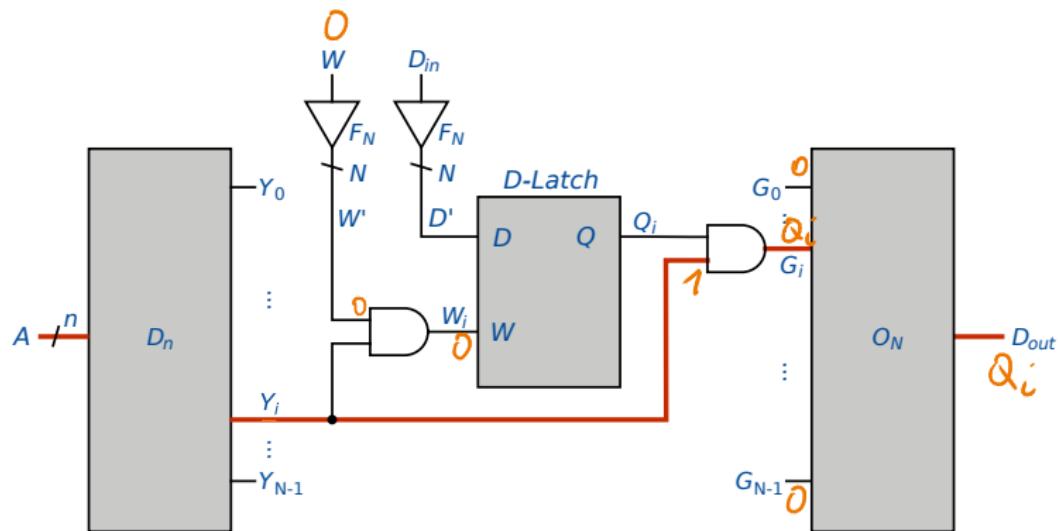


Welche Tiefe hat ein n-Bit Dekodierer, der - wie auf der vorhergehenden Folie dargestellt - rekursiv aufgebaut wird?

- a.  $O(1)$
- b.  $O(\log(n))$
- c.  $O(n)$
- d. Keine der obigen.

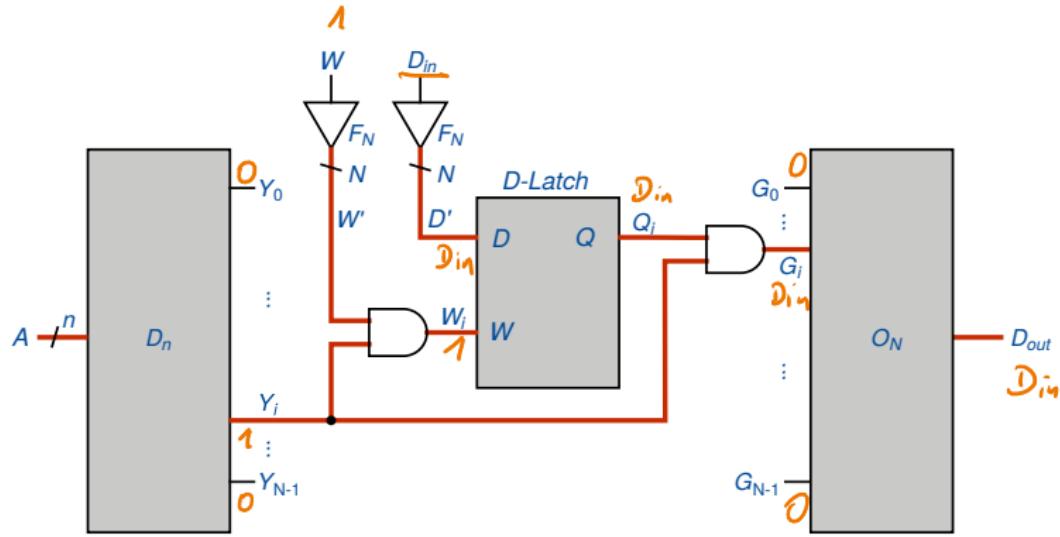
# SRAM: Lesevorgang ( $W = 0$ )

- $D_n$  setzt  $Y_i = 1$  für  $i = \langle A \rangle$ ,  $Y_j = 0$  für  $j \neq i$ .
- Der Inhalt der  $i$ -ten Zelle  $L_i$  steht an  $G_i$ , für alle  $j \neq i$  steht an  $G_j$  der Wert 0.



# SRAM: Schreibvorgang (Puls auf $W$ )

- $D_{in}$  an  $D$ -Eingänge sämtlicher Latches angelegt.
- Schreibpuls nur am  $W$ -Eingang von  $L_i$  (da  $Y_i = 1$ ).



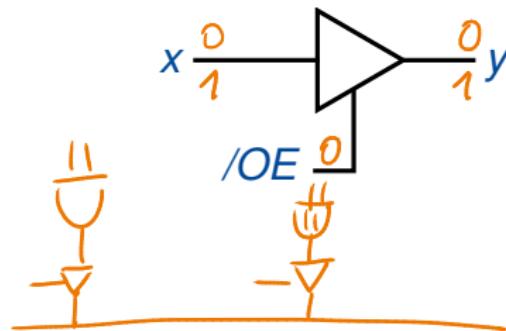
# Tristate-Treiber und Busse



- Tristate-Treiber sind Treiber mit Eingangssignal  $x$  und zusätzlichem Signal  $/OE$ , dem Output-Enable-Signal.

- Am Ausgang  $y$  erscheint

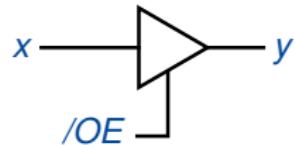
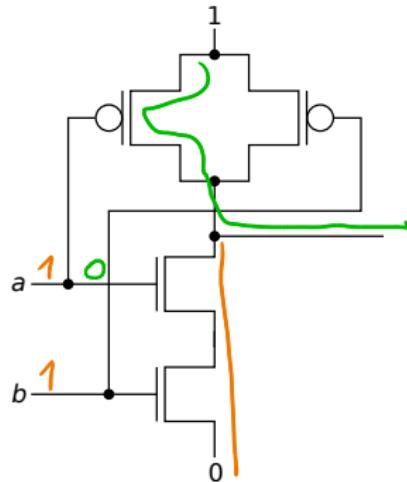
$$y = \begin{cases} x, & \text{falls } /OE = 0 \\ Z, & \text{sonst } /OE = 1 \end{cases}$$



- $Z$  bezeichnet den Zustand hoher Impedanz (high-Z).

# Zustand hoher Impedanz

- Wir haben bisher Schaltungen betrachtet, die aus CMOS-Gattern bestehen. Dort ist jede Leitung zu jedem Zeitpunkt entweder mit  $V_{DD}$  (logisch-1) oder Masse (logisch-0) verbunden.
- Eine Leitung im Zustand  $Z$ , also der Ausgang eines Treibers mit  $/OE = 1$ , ist weder mit  $V_{DD}$  noch mit Masse verbunden. Man sagt, der Treiber ist disabled ( $/OE = 0$ : enabled).



# $n$ -Bit-Treiber

- **$n$ -Bit-Treiber:**  $n$  Treiber mit gemeinsamen  $/OE$ .
- Im Gegensatz zu Ausgängen üblicher Gatter kann man Ausgänge von Tristate-Treibern zusammenschalten. Man muss dafür sorgen, dass zu jeder Zeit höchstens ein Treiber enabled ist.
- Ein  $n$  Bit breiter Bus ist ein Bündel aus  $n$  Leitungen, welche die Ausgänge von mehreren  $n$ -Bit-Treibern verbindet.



# Bus vs. Multiplexer

- k Tristate-Treiber, die durch einen Bus verbunden sind, wirken ähnlich wie ein *k-fach-Multiplexer*.



- Vorteile Bus gegenüber Multiplexer:

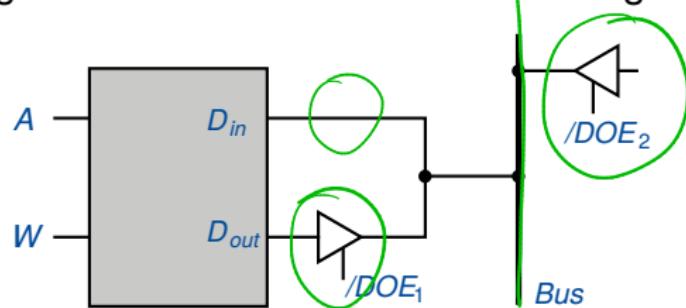
- Leicht erweiterbar.
- Datentransport in verschiedene Richtungen zu verschiedenen Zeiten.

- Nachteil von Bus:

- Man muss **Bus Contention** vermeiden, d.h. es darf nie mehr als ein Treiber auf einem Bus gleichzeitig enabled sein (sonst Folgen bis hin zur physikalischen Zerstörung der Schaltung)!

# Bus zur Kommunikation mit SRAM

- SRAM mit gemeinsamem Datenein- und -ausgang.



- **Lesezugriff** auf den Speicher:  $/DOE_1$  enabled, alle anderen Treiber, z.B.  $/DOE_2$ , disabled.
- **Schreibzugriff**:  $D_{in}$  nimmt den Wert vom Bus,  $/DOE_1$  disabled.



# Kapitel 4

Sequentielle Logik:

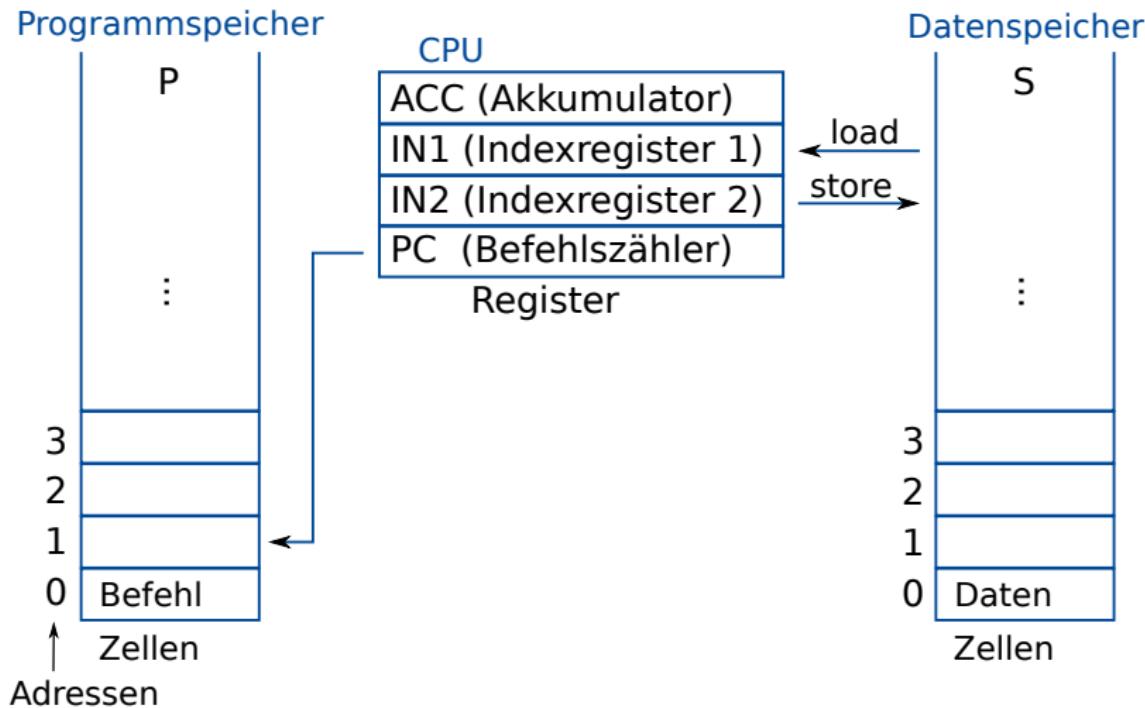
1. Speichernde Elemente
2. Sequentielle Schaltkreise
3. Entwurf sequentieller Schaltkreise
4. SRAM
5. **Anwendung: Datenpfade von ReTI**

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur  
WS 2016/17

# Zur Erinnerung: ReTI bisher



# Zur Erinnerung: Datenpfade von ReTI

---

- ReTI besteht aus
  - 4 benutzersichtbaren Registern *PC*, *ACC*, *IN1*, *IN2*  
→ Realisiert durch Zähler (*PC*) bzw. Register.
  - Einem  $2^{32}$ -Wort-Speicher (Wortbreite von 32 Bit), der Daten und Befehle enthält  
→ Realisiert durch SRAM.
- ReTI unterstützt Load-/Store-, Compute-, Indexregister- und Sprungbefehle.

31 .. 30	29 .. 24	23 .. 0
Typ	Spezifikation	Parameter <i>i</i>

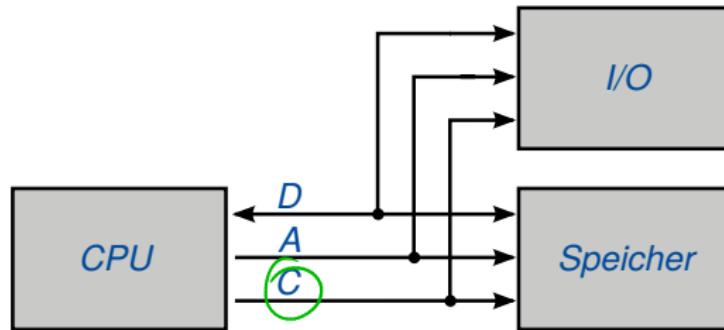
# Zur Erinnerung: ReTI-Befehle im Überblick

	31 30	29 28	27	26	25	24	23	...	0	
Load	0 1	M	*		D		i			
	31 30	29 28	27	26	25	24	23	...	0	
Store	1 0	M	S		D		i			
	31 30	29	28	27	26	25	24	23	...	0
Compute	0 0	MI	F		D		i			
	31 30	29 28	27	26	25	24	23	...	0	
Jump	1 1	C		*			i			

M - Modus ; S - Source ; D - Destination ; MI - memory/immediate ;  
F - Function ; C - Condition

# Umsetzung von ReTI: Externe Sicht

- 3-Bus-Architektur zur Ansteuerung von Speicher und I/O-Geräten:
  - 32 Bit breiter Datenbus  $D = D[31 : 0]$ ,
  - 32 Bit breiter Adressbus  $A = A[31 : 0]$ ,
  - Kontrollbus  $C$  (Breite später festgelegt).



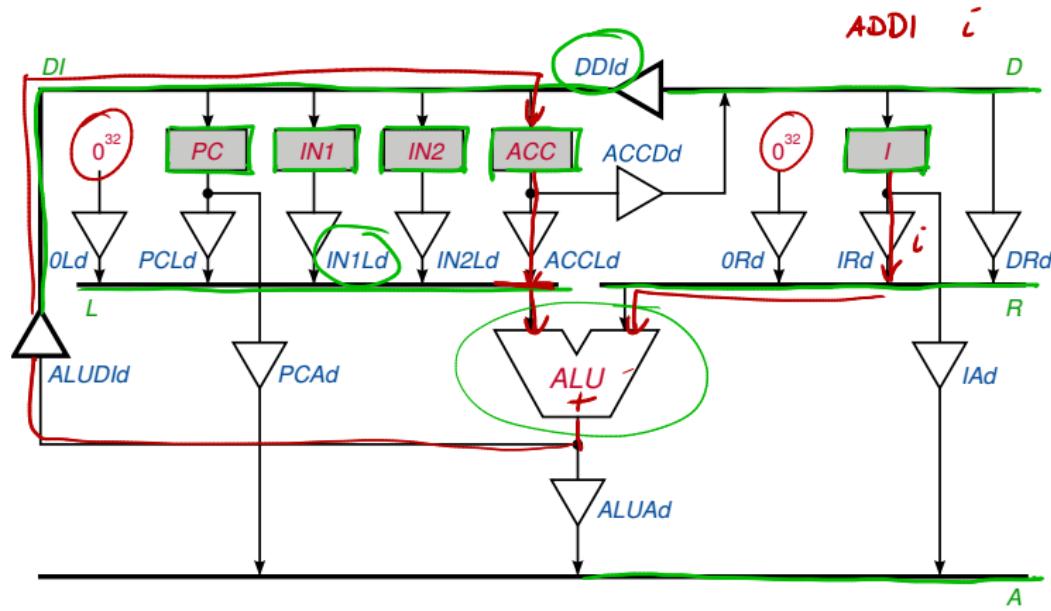
# Umsetzung von ReTI: Interne Sicht

---

- *CPU* besteht aus:
  - Zähler *PC*.
  - 3 für Benutzer sichtbaren Registern *ACC*, *IN1*, *IN2*,
  - Instruktionsregister *I*,
  - *ALU*,
  - *CPU*-internen Bussen:
    - *L*, *R* für linken bzw. rechten Operanden der *ALU*,
    - internem Datenbus *DI*.
- Register, *PC*, *ALU*, Busse und die zugehörigen Treiber sind 32 Bit breit.

# Interner Aufbau der ReTI-CPU

Load / Acc 15



# SMILE - ReTI, interner Aufbau/Befehlstypen

---

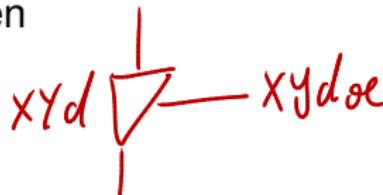
Bisher haben wir Compute-Befehle nur über den Akkumulator (ACC) definiert.

Ist es mit dem internen Aufbau der ReTI - wie auf der vorherigen Folie dargestellt - möglich, die Compute-Befehle auch auf die Register PC, IN1 und IN2 zu erweitern?

- a. Ja, aber nur für den PC.
- b. Ja, aber nur für IN1 und IN2.
- c. Ja, für alle 3.
- d. Nein.

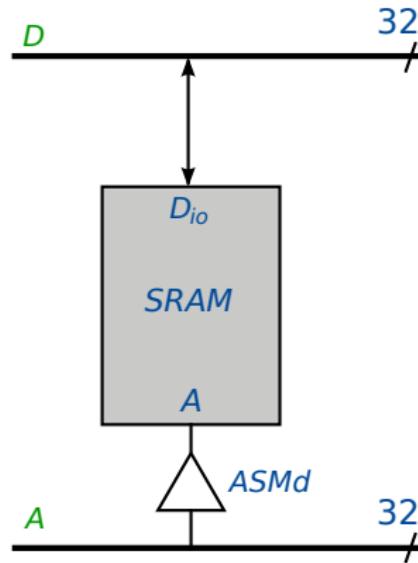
# Hinweise zum Schaltbild

- Namenskonvention für Treiber: Treiber zwischen Bus/Baustein X und Bus/Baustein Y:  $XYd$ ;
  - Output-Enable-Signal:  $XYdoe$ .
  - $0Ld$  und  $0Rd$  können  $0^{32}$  auf  $L$  bzw.  $R$  legen.
- Busse  $A$  und  $D$  sind an den Speicher (sowie I/O-Geräte) angeschlossen, s. nächste Folie.
- Das Bild enthält keine Steuerleitungen:
  - Output-Enable-Signale der Treiber,
  - Funktionsauswahl der ALU,
  - Clock-Signale und “Clock-Enable-Signale” der Register.

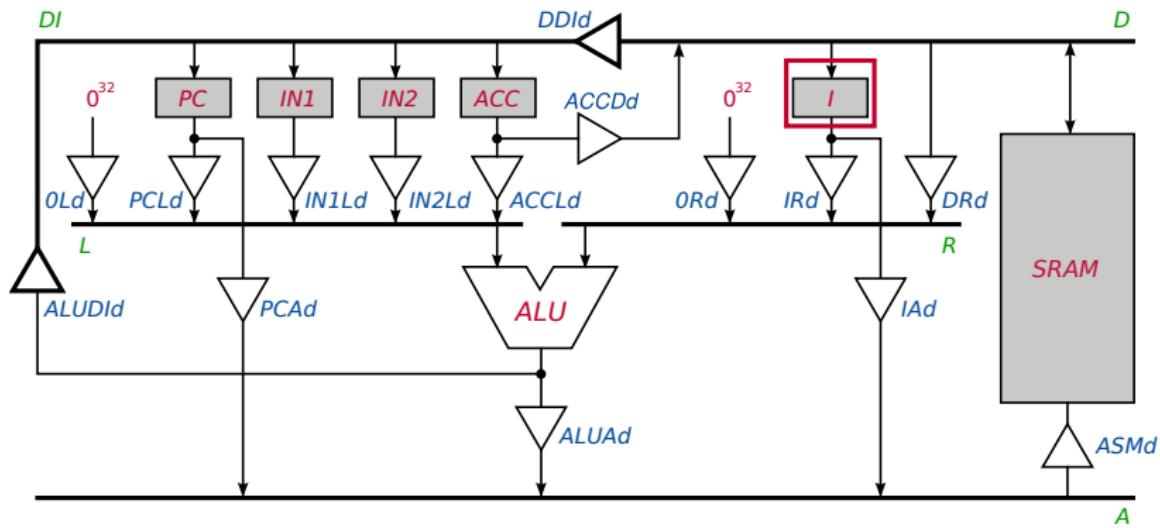


# Speicher SM von ReTI

- Datenein- und ausgänge mit Datenbus  $D$  der CPU verbunden.
- Adressleitungen mit Adressbus  $A$  der CPU verbunden.
- Treiber  $ASMd$  immer enabled.



# Verfeinerung des Schaltbilds

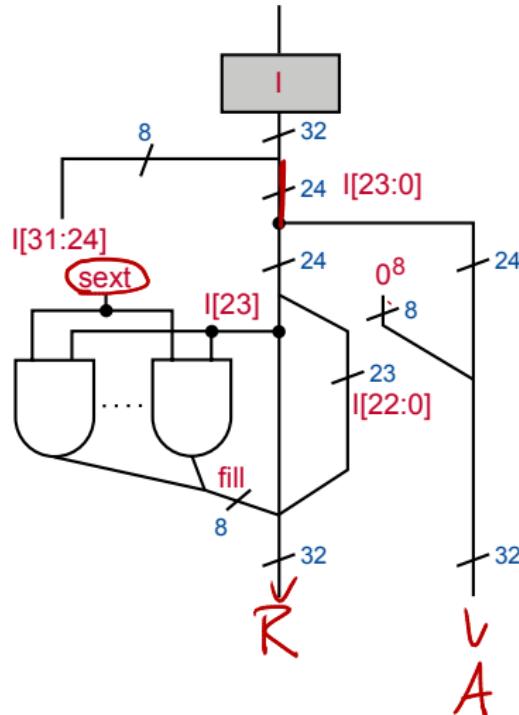


# Verarbeitung der Daten im Register /

---

- Im Instruktionsregister *I* steht der gerade verarbeitete Befehl.
  - *I[31 : 24]*: Befehlskodierung (für die im Schaltbild nicht dargestellten Steuersignale benötigt).
  - *I[23 : 0]*: Speicheradresse oder Konstante für die ALU.
    - Speicheradresse wird mit acht Nullen aufgefüllt.  
 $0^8 I[23 : 0]$  wird auf Bus *A* gelegt (*IAd* enabled).
    - Natürliche 24-Bit-Konstante wird mit acht Nullen aufgefüllt.  
 $0^8 I[23 : 0]$  wird auf *R* gelegt (*IRd* enabled).
    - Ganzzahlige 24-Bit-Konstante wird vorzeichenerweitert.  
 $\text{sext}(I[23 : 0])$  wird auf *R* gelegt (*IRd* enabled).

# Verfeinerung

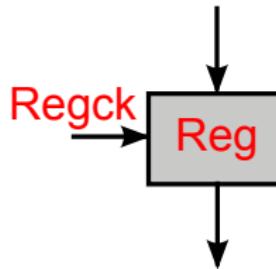


## ■ Neues Kontrollsiegel $\text{sext}$ :

- $\text{sext} = 0$ : Ersetze  $I[31 : 0]$  durch  $0^8/[23 : 0]$ .
- $\text{sext} = 1$ : Ersetze  $I[31 : 0]$  durch  $\text{sext}(I[23 : 0])$ .

# Weitere Verfeinerung für Register (1/3)

- Im Buch von Keller / Paul werden die Clockeingänge aller Register **Reg** mit einem Clocksignal **Regck** verbunden, das **durch die Kontrolllogik berechnet wird.**
- ⇒ Datenübernahme zu ausgewählten Zeitpunkten



- Vorgehen bei Realisierung der ReTI auf einer Platine mit diskreten Bausteinen ok!
- Nicht ok bei heutigen Designs, die Prozessoren auf einem einzigen Chip integrieren.

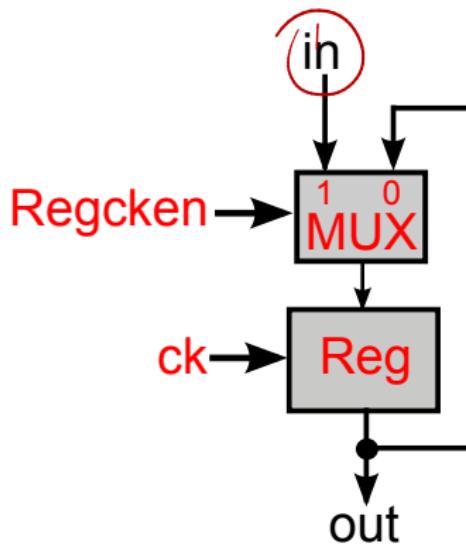
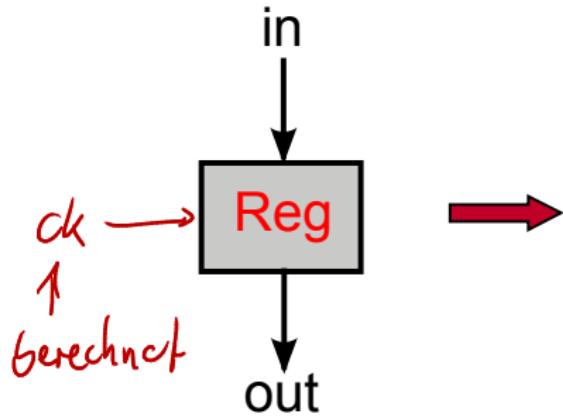
# Weitere Verfeinerung für Register (2/3)

---

- Gründe für “**Designrule**”, die es verbietet, Clockeingänge mit berechneten Datensignalen zu verbinden:
  - Spezielle Methoden (“**Clock–Tree–Synthese**”) gewährleisten, dass die steigende Flanke der globalen Clock an allen Clockeingängen zum **gleichen** Zeitpunkt ankommt (z.B. durch Ausgleich des Effekts unterschiedlicher Leitungsverzögerungen m.H. von Treibern). Datensignale auf Clockeingängen verhindern Clock–Tree–Synthese.
  - Heutige Werkzeuge zur automatischen **Timing–Analyse** (und Berechnung der maximalen Clockfrequenz) sind nicht in der Lage, mit berechneten Clocksignalen umzugehen.
  - Spezielle Anforderungen an “Flankensteilheit” der Clocksignale (siehe Kapitel über physikalische Eigenschaften)

## Weitere Verfeinerung für Register (3/3)

- Transformation bzw. Verfeinerung:

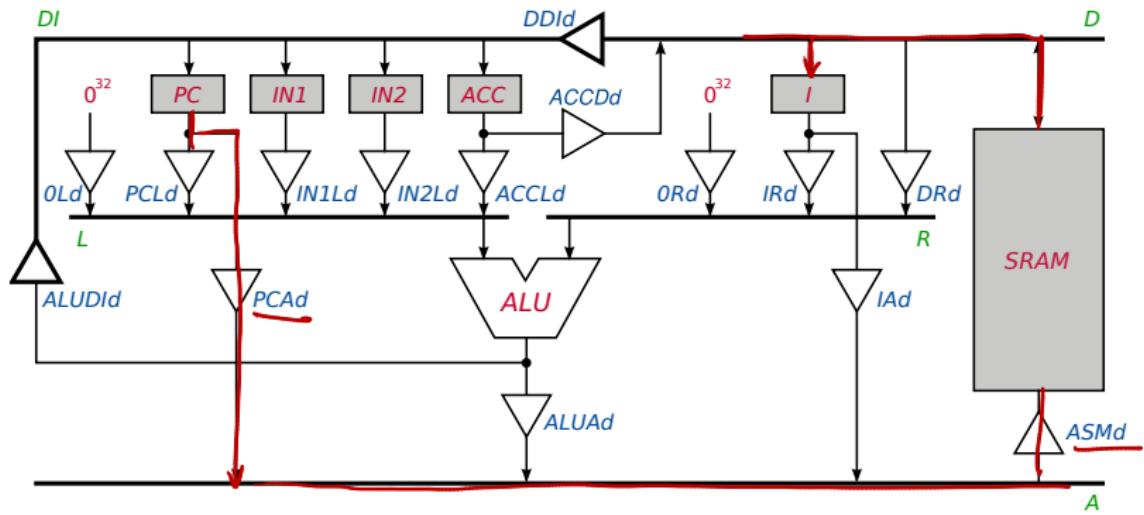


# Programmabarbeitung durch ReTI

---

- Zwei sich abwechselnde Phasen der *CPU*:
  - **Fetch-Phase**: Lädt nächsten auszuführenden Befehl aus Memory ins **Instruktionsregister /** der *CPU*.
  - **Execute-Phase**: Befehl, der in **/** steht, wird ausgeführt.
- Vorgehen:
  - 1 Definition der **Datenpfade**, d.h. der benötigten Datenverbindungen zwischen den Komponenten der *CPU*.
  - 2 Herleitung der **Kontrollsignale** zur Ansteuerung der im Punkt 1 hergeleiteten Datenpfade.
    - Treiber-OE, *ALU*-Funktionsselektion, Register-Clock.
  - 3 Sequentielle Synthese.

# Datenpfade: Fetch-Phase



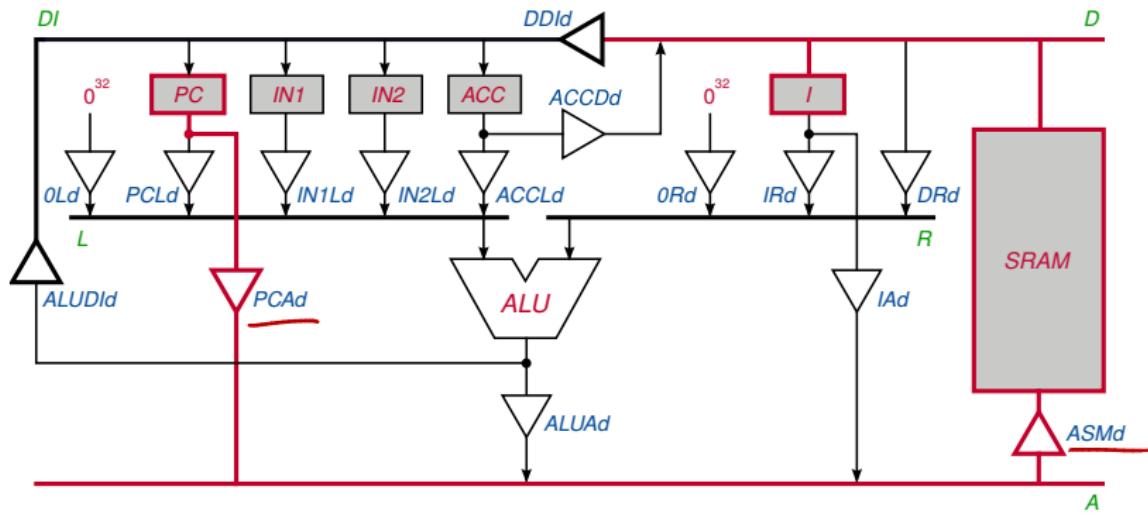
# SMILE - Fetch-Phase, Treiber

---

Welche Treiber müssen in der Fetch-Phase enabled sein?

- a. ALUAd
- b. ASMd
- c. DDId
- d. IAd
- e. IRd
- f. PCAd
- g. PCLd

# Datenpfade: Fetch-Phase

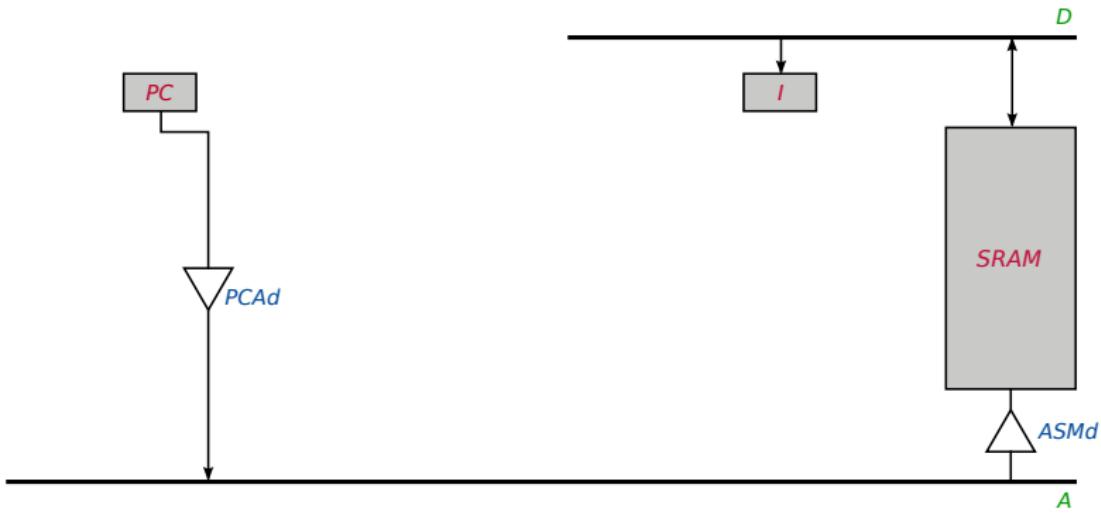


# Bedeutung des Diagramms

---

- In der Fetch-Phase muss:
  - Bus A mit PC verbunden sein, d.h. PCAd enabled.
  - Register I den Wert von Bus D übernehmen, d.h. Icken muss enabled werden.
  - Alle anderen Treiber (außer denen, die stets enabled sind) sind disabled, um Bus Contentions zu vermeiden.
- Die Steuersignale müssen in der Fetch-Phase entsprechend gesetzt sein.
  - Z.B. /PCAdoe = 0, /ALUAdoe = 1, usw. (active low!)

# Fetch: Die durchgeschalteten Pfade



# Datenpfade in der Execute-Phase

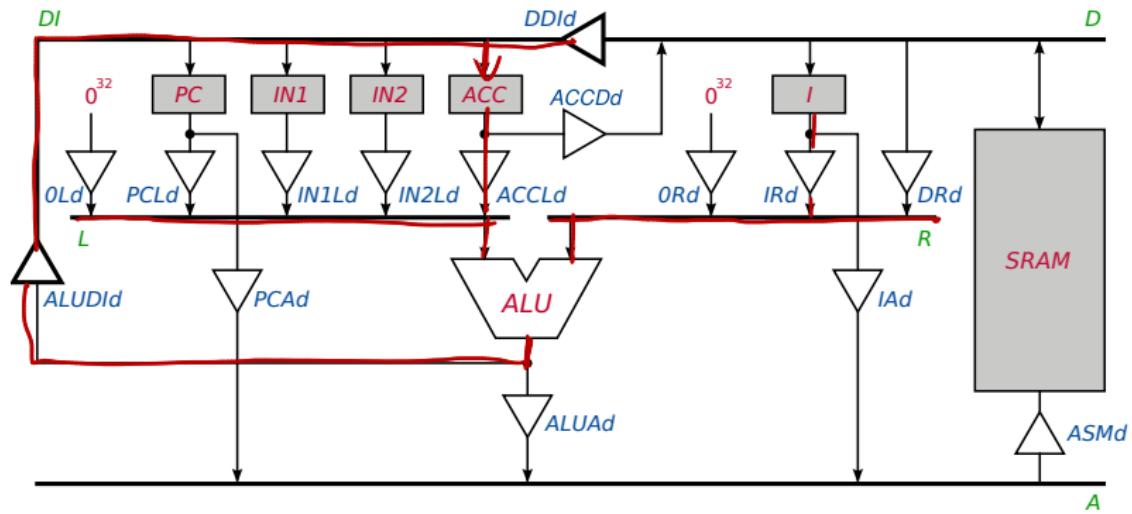
---

- Betrachte unterschiedliche Befehlstypen.

- *Compute Immediate*,
- *Compute Memory*,
- *JUMP*,
- *LOAD*,
- *LOADIN1* (*LOADIN2* analog),
- *LOADI*,
- *STORE*,
- *STOREIN1*, (*Store IN2*)
- *MOVE*.

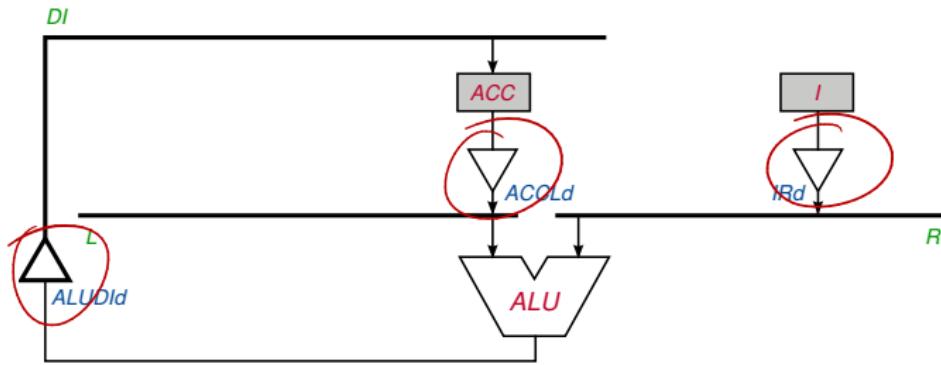
# Datenpfade: *Compute Immediate* (1/2)

*ADD I ACC <v>*

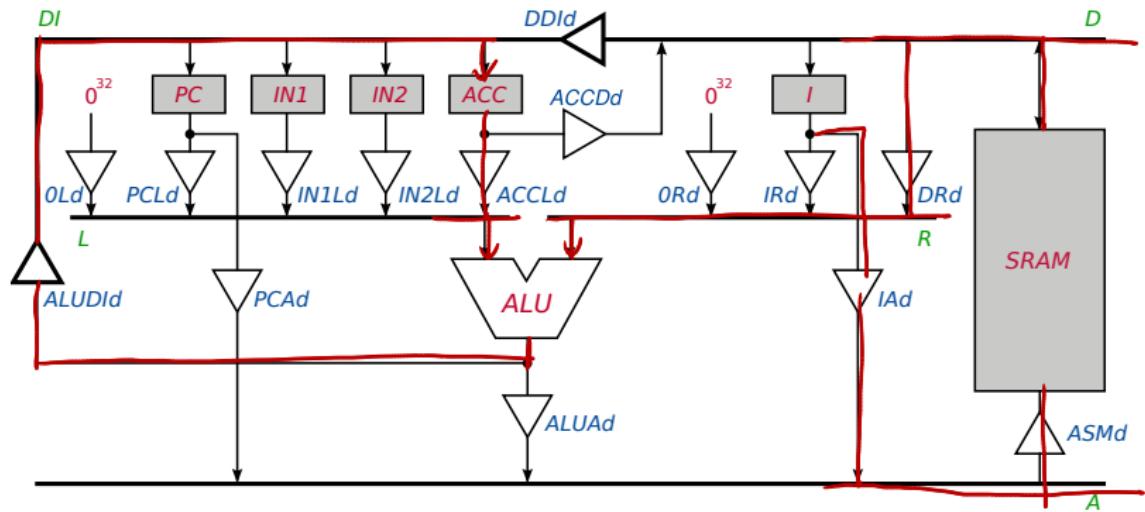


## Datenpfade: *Compute Immediate* (2/2)

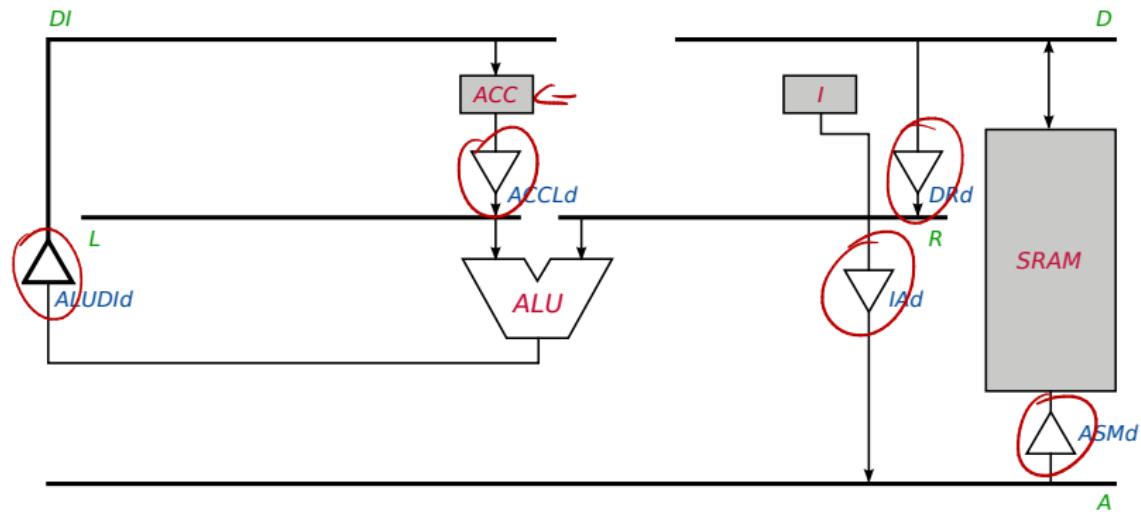
---



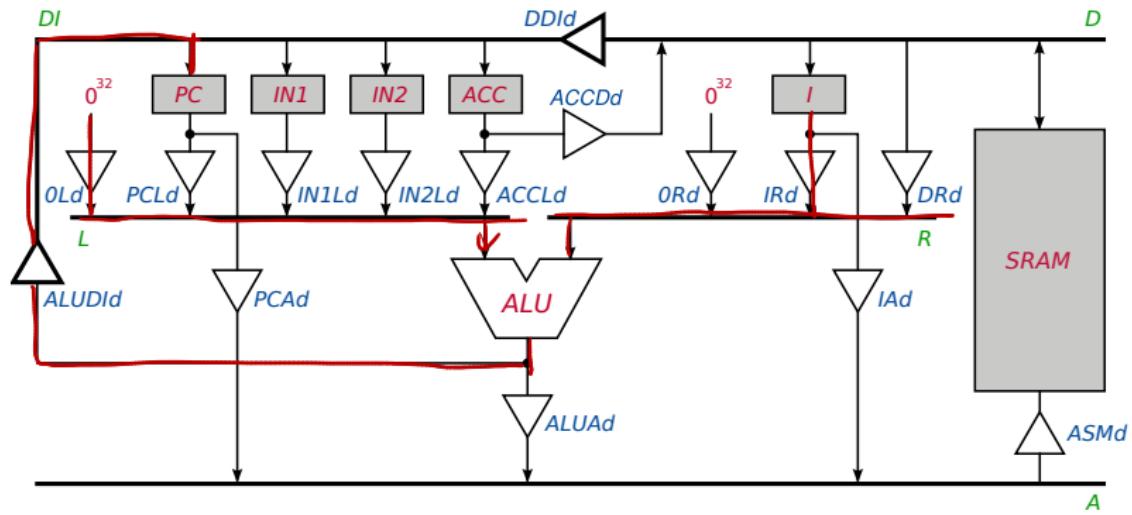
# Datenpfade: *Compute Memory* (1/2)



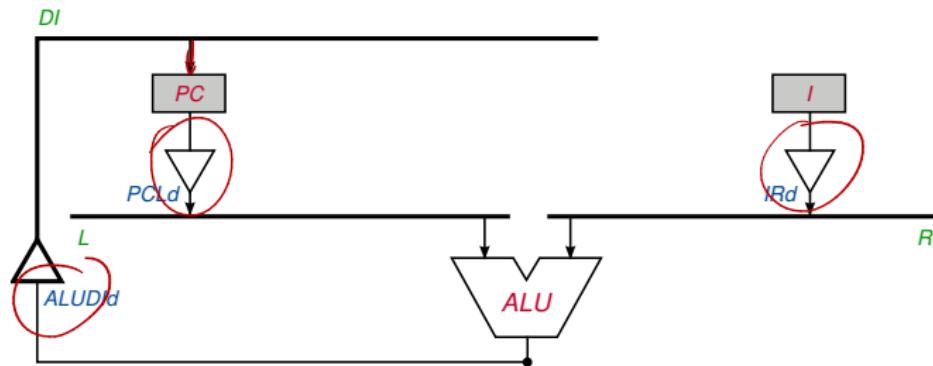
# Datenpfade: *Compute Memory* (2/2)



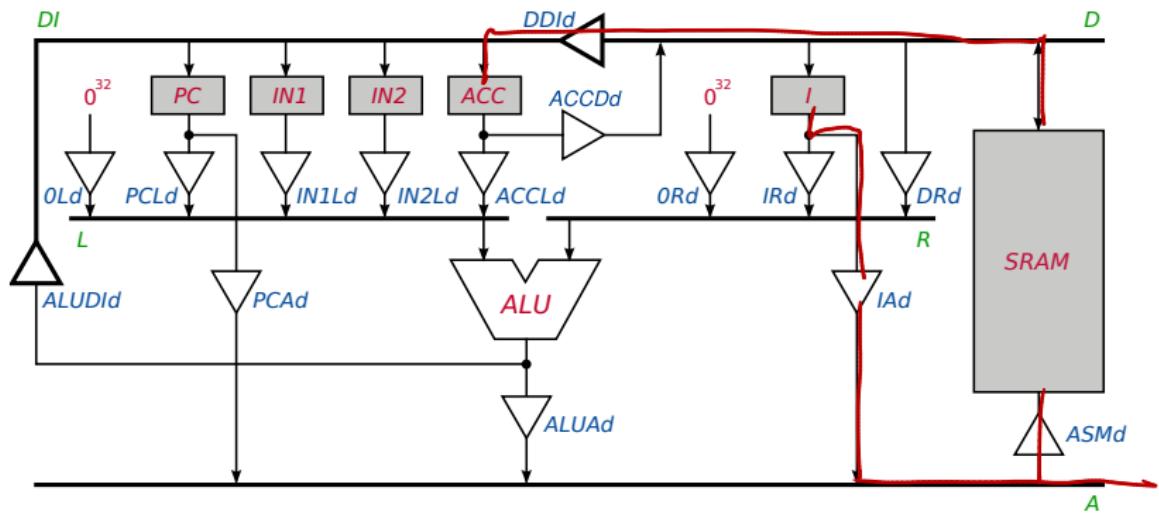
# Datenpfade: *JUMP* (1/2)



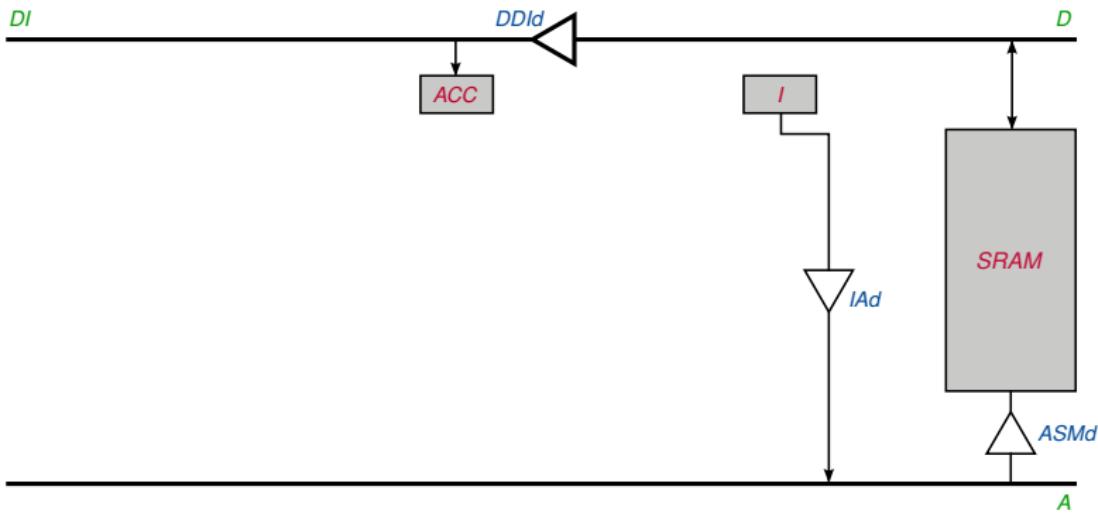
## Datenpfade: *JUMP* (2/2)



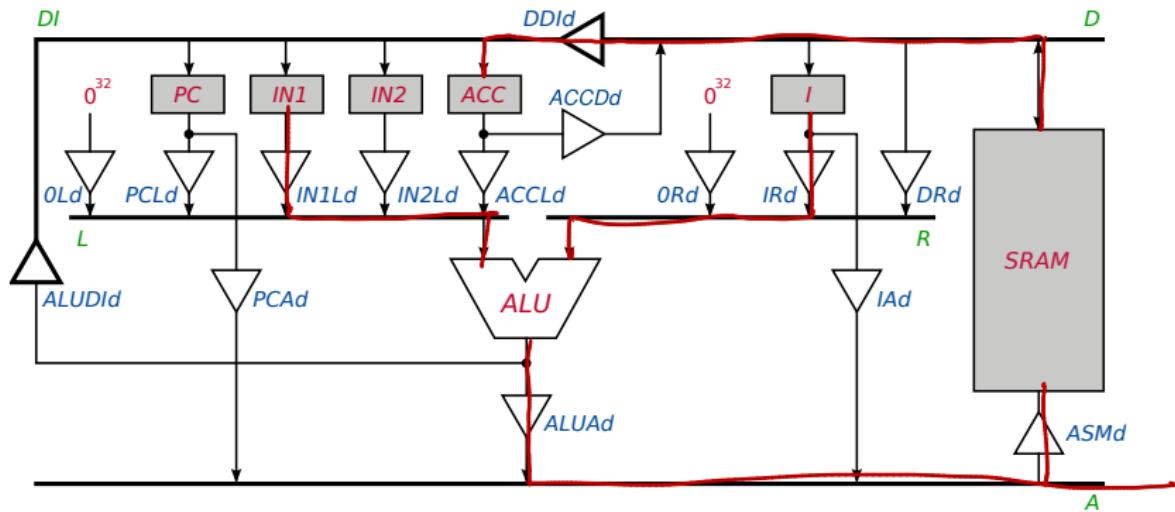
# Datenpfade: *LOAD i* (1/2)



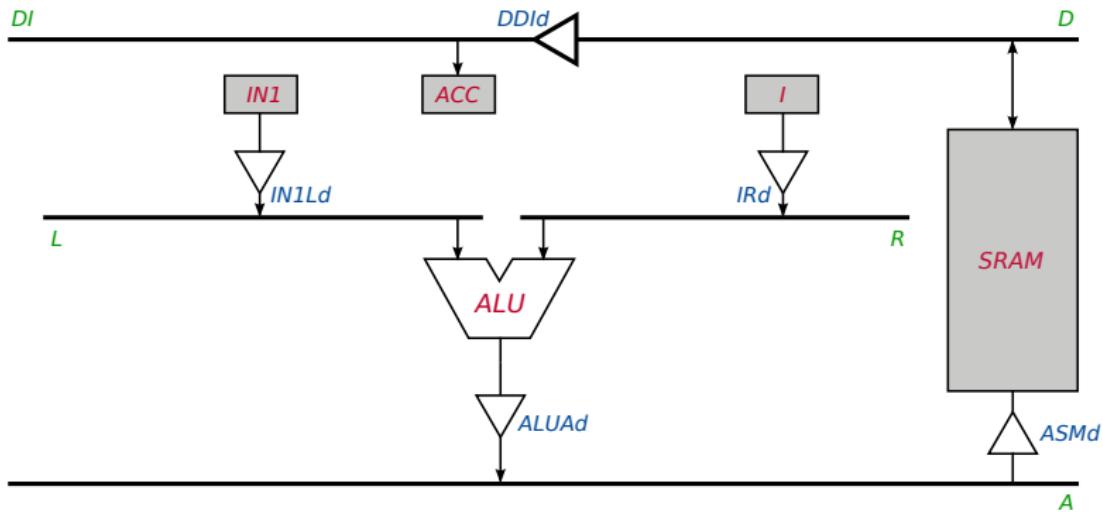
## Datenpfade: *LOAD i* (2/2)



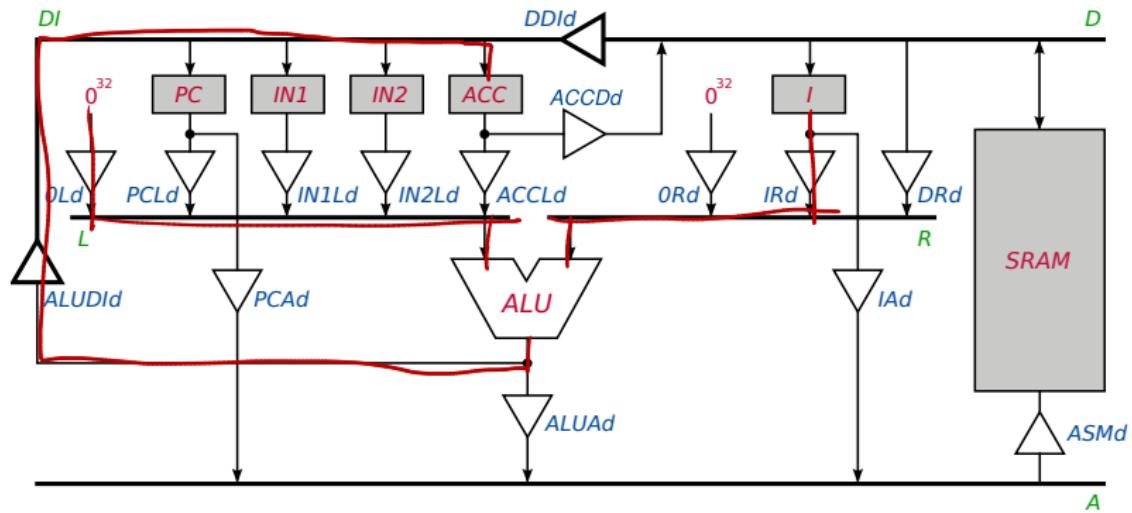
# Datenpfade: LOADIN1 i (1/2)



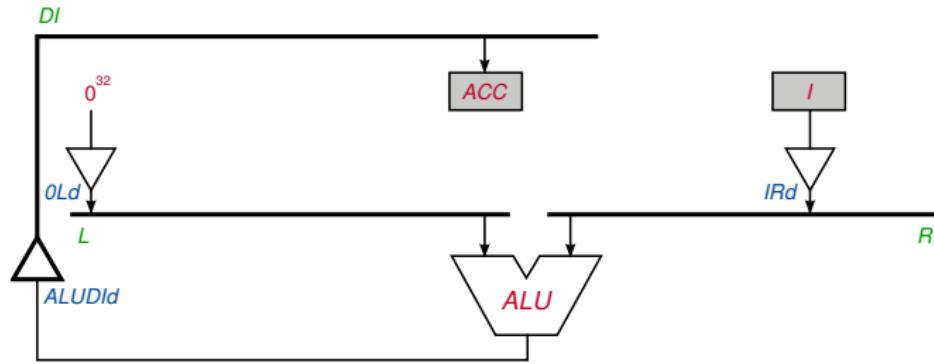
# Datenpfade: *LOADIN1 i* (2/2)



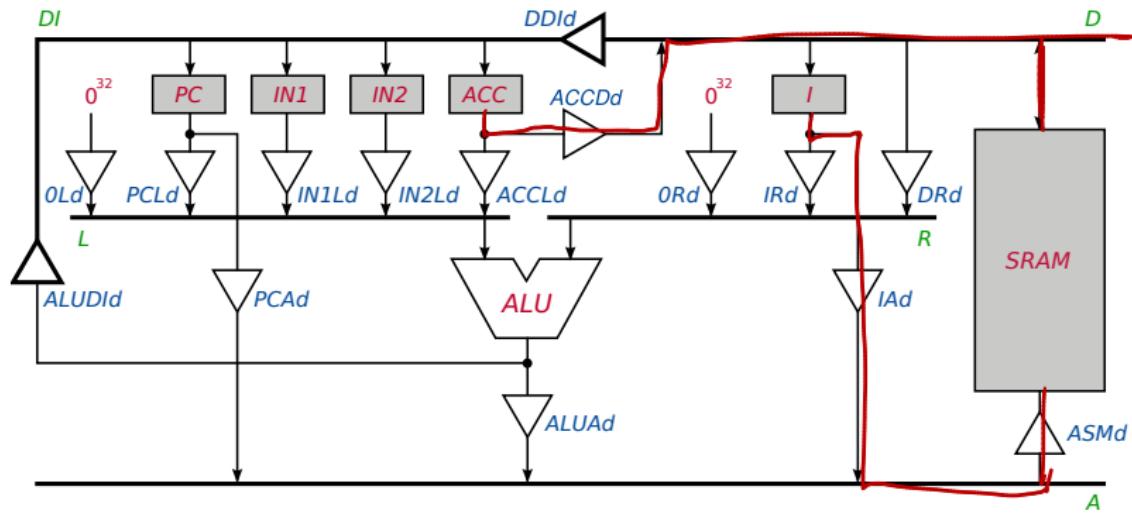
# Datenpfade: *LOADI i* (1/2)



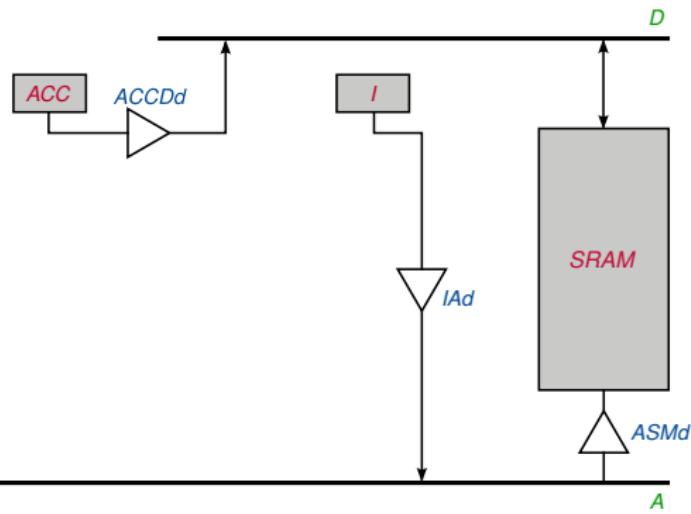
## Datenpfade: *LOAD I* (2/2)



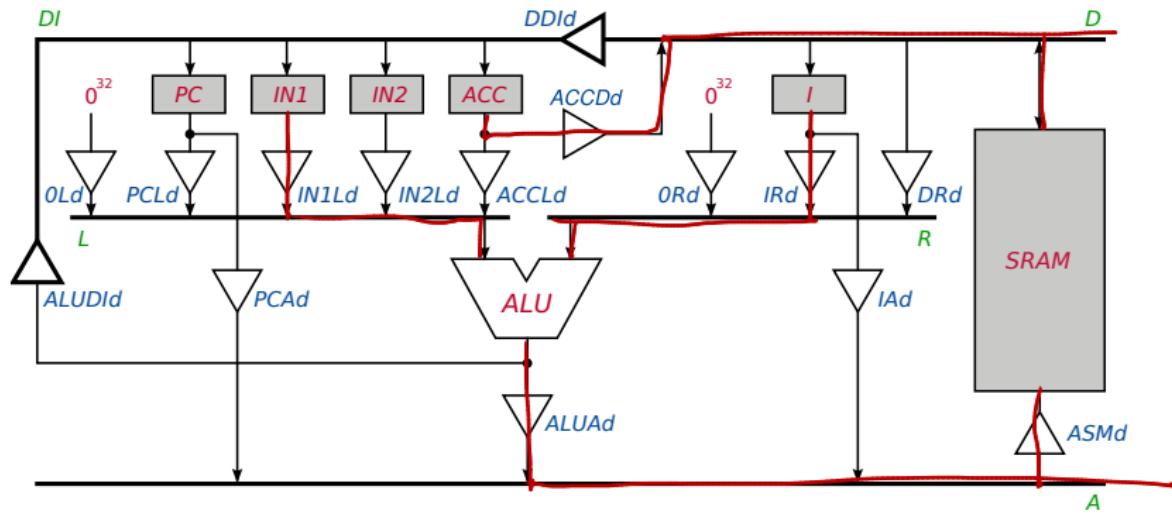
# Datenpfade: *STORE i* (1/2)



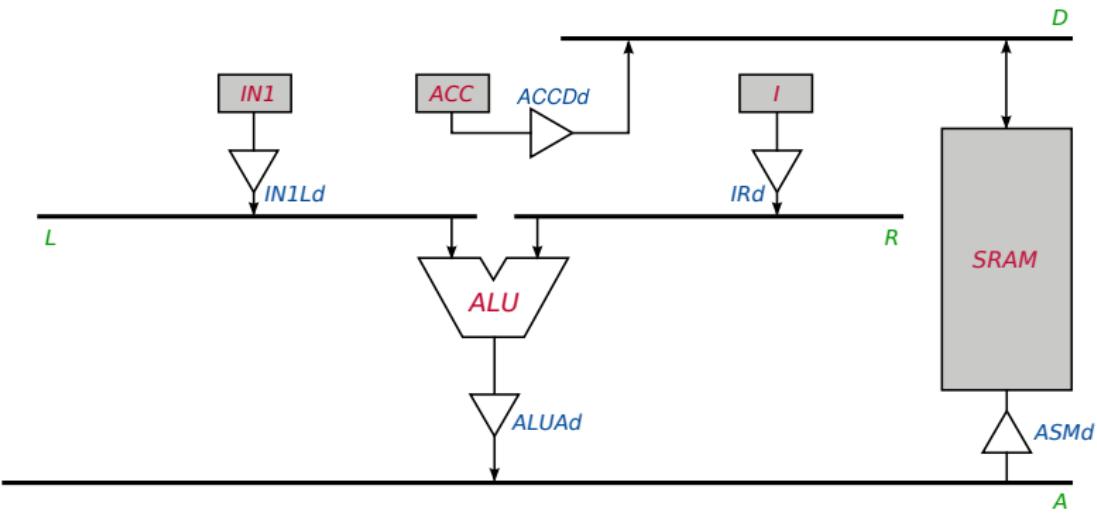
## Datenpfade: *STORE i* (2/2)



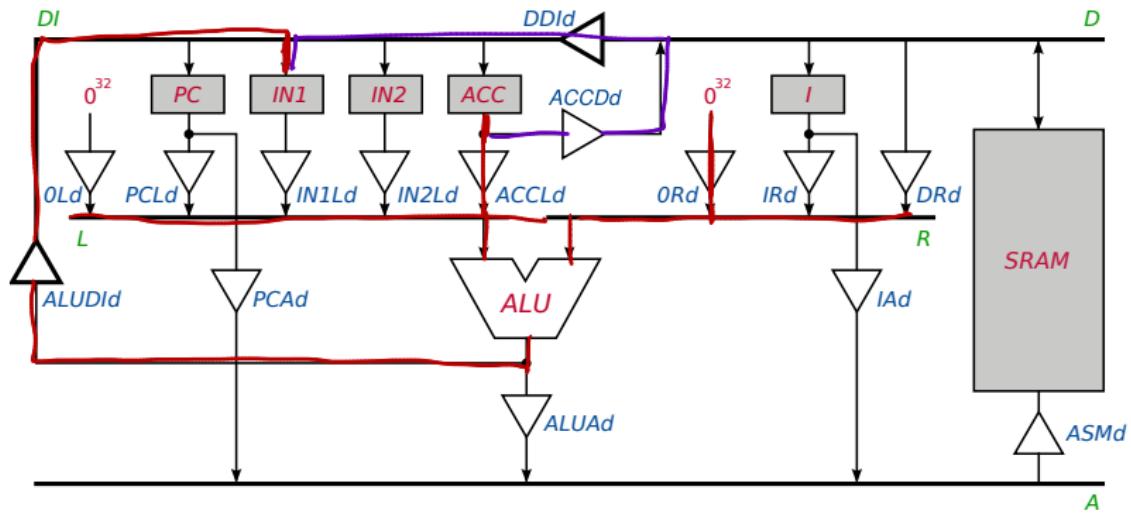
# Datenpfade: *STOREIN1 i* (1/2)



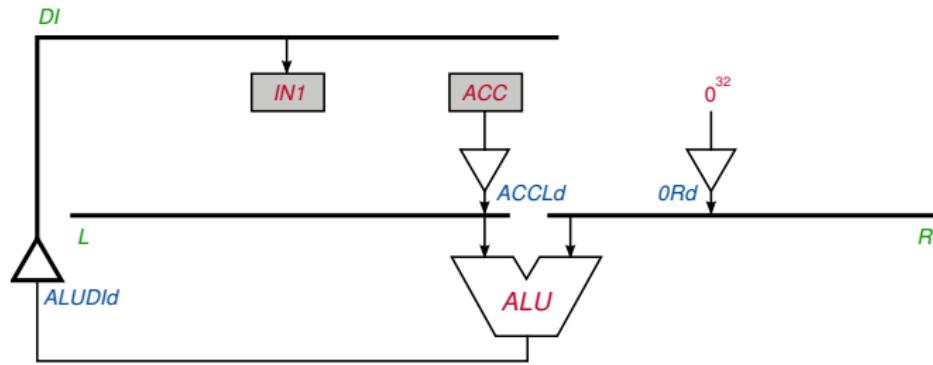
# Datenpfade: *STOREIN1 i* (2/2)



# Datenpfade: *MOVE ACC IN1* (1/2)



# Datenpfade: *MOVE ACC IN1* (2/2)



# Zusätzliche Befehle

- Man kann weitere Befehle ohne zusätzliche Hardware realisieren!
- Load- und Compute-Befehle mit beliebigem Zielregister  
 $r \in \{PC, IN1, IN2, ACC\}$ .
  - Kein  $\langle PC \rangle := \langle PC \rangle + 1$ , wenn  $r = PC$ .
- Befehlsformat:  $LOAD\ r\ i; ADD\ r\ i$ ; etc.
- Befehlskodierung:

	31	30	29	28	27	26	25	24	23	...	0
Load	0	1	M		*		D		i		

	31	30	29	28	27	26	25	24	23	...	0
Compute	0	0	MI	F			D		i		

# Zur Illustration: ReTI-Simulator „Neumi“

---

- Verfügbar in Ilias (Zusatzmaterial)
  - Simulator der ReTI-Maschine.
  - Anleitungen und zwei Beispielprogramme.
- Für die Ausführung wird Java benötigt.
  - Für Windows: <http://www.java.com/de/download/manual.jsp>
  - Für Linux: Nicht den GNU-, sondern den SUN-Compiler verwenden (Neumi funktioniert nicht mit GCJ).
  - Für Solaris und MacOS sollte die richtige Java-Version vorliegen.
- Syntax geringfügig gegenüber Vorlesung abgewandelt.
  - Kommas zwischen Operanden: „ADDI ACC, 1“ statt „ADDI ACC 1“.
  - Jump-Befehle anders geschrieben: „JUMP ge, 2“ statt „JUMP  $\geq$  2“.
  - Details: Anleitungen auf der Webseite.

# Implementierung von ReTI (1/2)

---

- Im Buch von Keller/Paul wird ReTI (bzw. ReSa) mit sogenannten **diskreten FAST-Bausteinen** realisiert.
  - Register, Zähler, ALU, Treiber, Speicher: Bausteine aus der FAST-Bibliothek, teilweise mehrere Bausteine, um **Bitbreite 32** zu erreichen.
  - Kontrollsignale werden durch sog. **PALs** realisiert
    - Programmable Array Logic (Bausteine von AMD).
    - Spezielle Beschreibungssprache PALASM.
    - PALs werden heute nur noch selten verwendet.

# Implementierung von ReTI (2/2)

---

- Im Gegensatz dazu verwenden wir State-of-the-Art-Bausteine der **NanGate-Bibliothek** (<http://www.si2.org/openeda.si2.org/projects/nangatelib>) im Hinblick auf eine VLSI–Implementierung auf einem einzigen Chip.
- Auf Basis einer solchen Implementierung behandeln wir später wesentliche Konzepte für eine „**Timing-Analyse**“. Wir beginnen zunächst mit der Realisierung der Kontrollsignale (Output–Enable, Clock–Enable ...).
- Kontrollsignale werden durch einen Endlichen Automaten generiert. Wir **skizzieren** hier das Vorgehen.

# Zu generierende Kontrollsignale

---

- Clock–Enable–Signale für alle Register  $r$ , Bez.:  $rcken$ .
- Output enable Signale (active low) für alle Treiber  $XYd$ , Bez.:  $/XYdoe$ .
- Funktions-Select-Signale  $f[2 : 0]$  zum Selektieren der Funktion, die von ALU ausgeführt wird.
- Signale  $/PCclear$ ,  $/PCload$  für  $PC$ .
- $sext$  zur Berechnung der Füllbits bei 24-Bit-Immediate-Konstanten.
- Für den Speicher benötigen wir die Kontrollsingale (active low)  $/SMDdoe$ ,  $/SMw$ .

# Idealisierte Timing-Diagramme

---

- Grobe Ablaufplanung mit idealisierten Timing-Diagrammen.
- Vereinfachende Annahme: Verzögerungszeit aller Bausteine = 0 (exakte Analyse mit Verzögerungszeiten später).
- Befehlsabarbeitung ist unterteilt in Takte (= Folge von Taktsignalen *high, low*).
- Fragen:
  - Wie sollen Kontrollsignale zusammenspielen? |
  - In welchem Takt sollen welche Treiber aktiviert, welche Registerclock enabled werden?

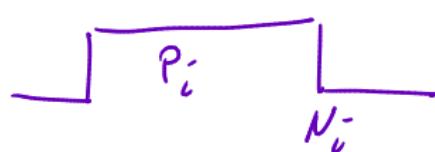
# Befehlsabarbeitung in Takten

- Sowohl Fetch- als auch Execute-Phase bestehen aus 4 Taktten gleicher Länge.

- Kontrollsignal:

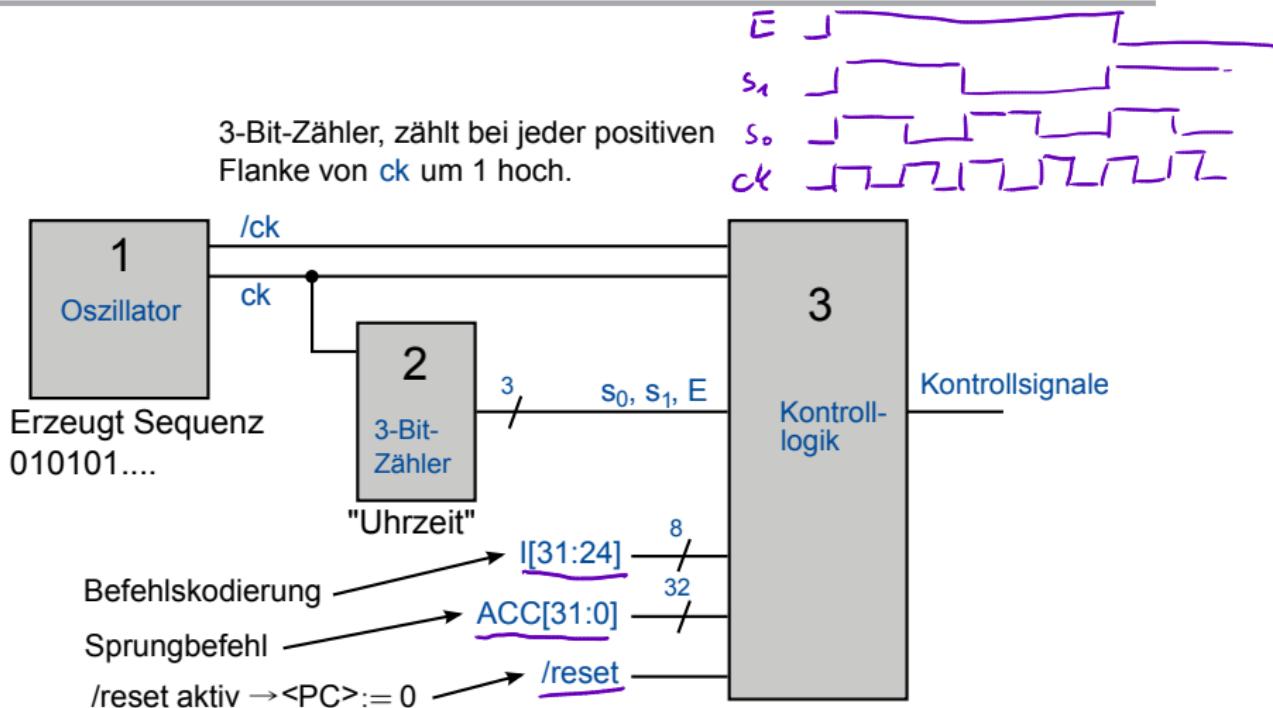
- $E = 0$ : Fetch-Phase,

- $E = 1$ : Execute-Phase.

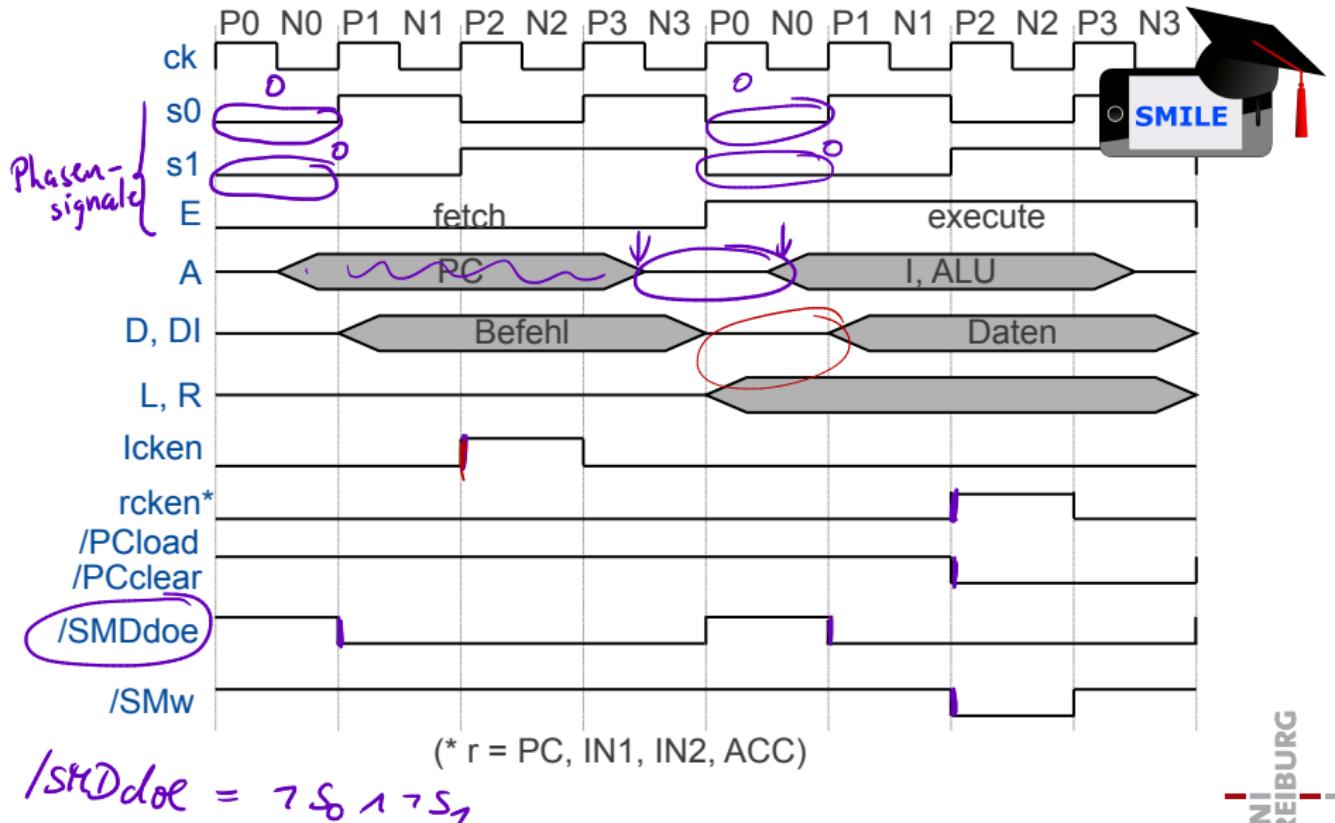


- Signale  $s_0, s_1$  = Binärkodierung der **Nummer des Taktes** (innerhalb Fetch, Execute), in dem ReTI sich befindet.
- Steigende Flanken (Anfang des Taktes) werden mit  $P_i$ , fallende (Mitte des Taktes) mit  $N_i$  bezeichnet ( $i = 0, \dots, 3$ ).
- Clock  $ck$ , Signale  $s_0, s_1$  und  $E$  werden Phasensignale genannt. Weitere (Kontroll-)Signale werden aus den Phasensignalen erzeugt.

# Erzeugung der Phasensignale

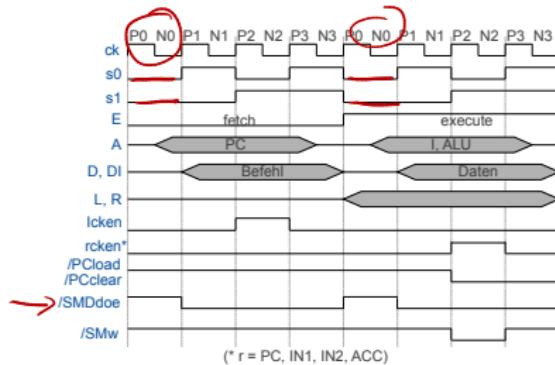


# Idealisiertes Timing-Diagramm



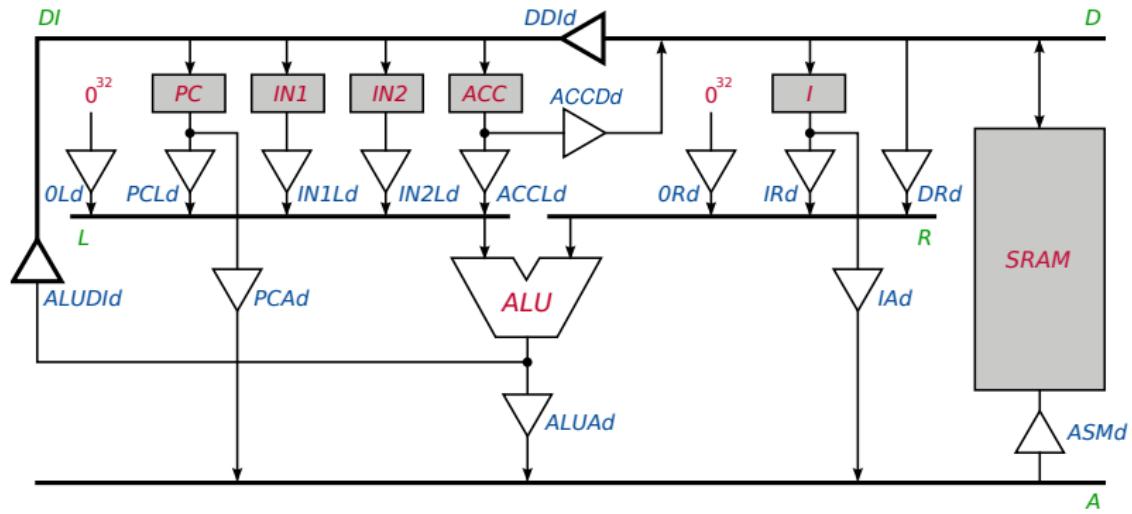
# SMILE - Kontrollsignale

Welche der folgenden Funktionen realisiert das Signal /SMDdoe, wie es bei der Ausführung eines entsprechenden Befehls gebraucht wird?



- a.  $(E \cdot (s_0 + s_1))'$
- b.  $s_0' \cdot s_1'$
- c.  $s_0 + s_1$
- d. Keine der obigen

# Zur Erinnerung: Datenpfade



# Entwurfsziele

---

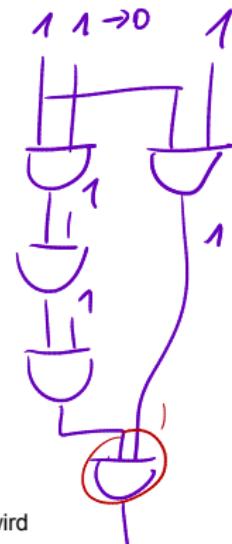
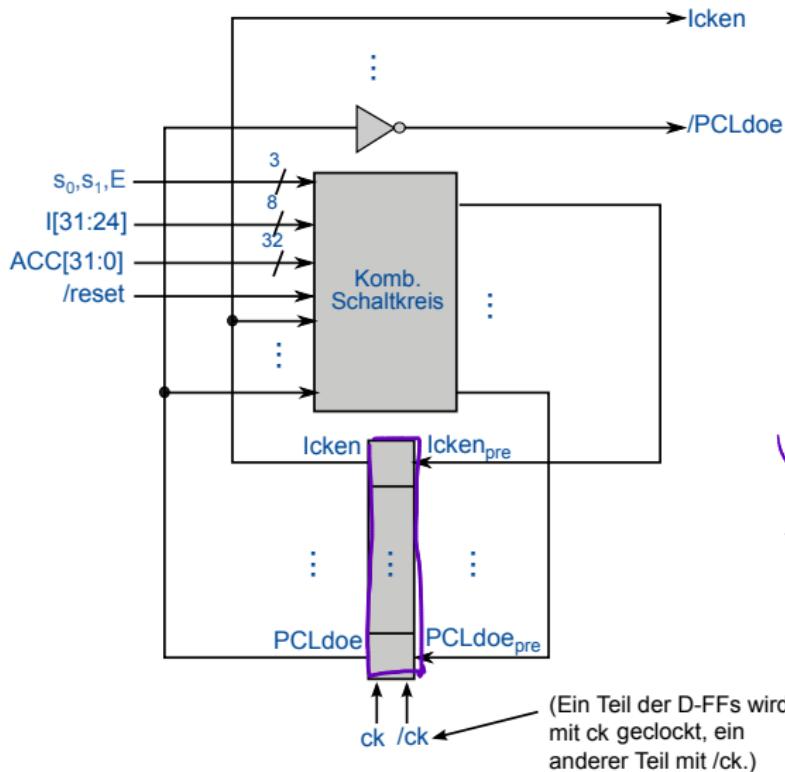
- Nutze Busse möglichst lange (unter Vermeidung von Bus Contention).
- Clock-Enable-Signale der Register möglichst spät  
→ viel Zeit für Berechnung neuer Daten.
- Nach dem Entwurfsende muss das Timing der CPU mit den konkreten Werten der eingesetzten Fertigungstechnologie überprüft werden. (“Wie schnell kann man maximal takten, um korrektes Funktionieren zu garantieren?”)
  - Siehe nächstes Kapitel.

# Aufbau der Kontrolllogik (1/2)

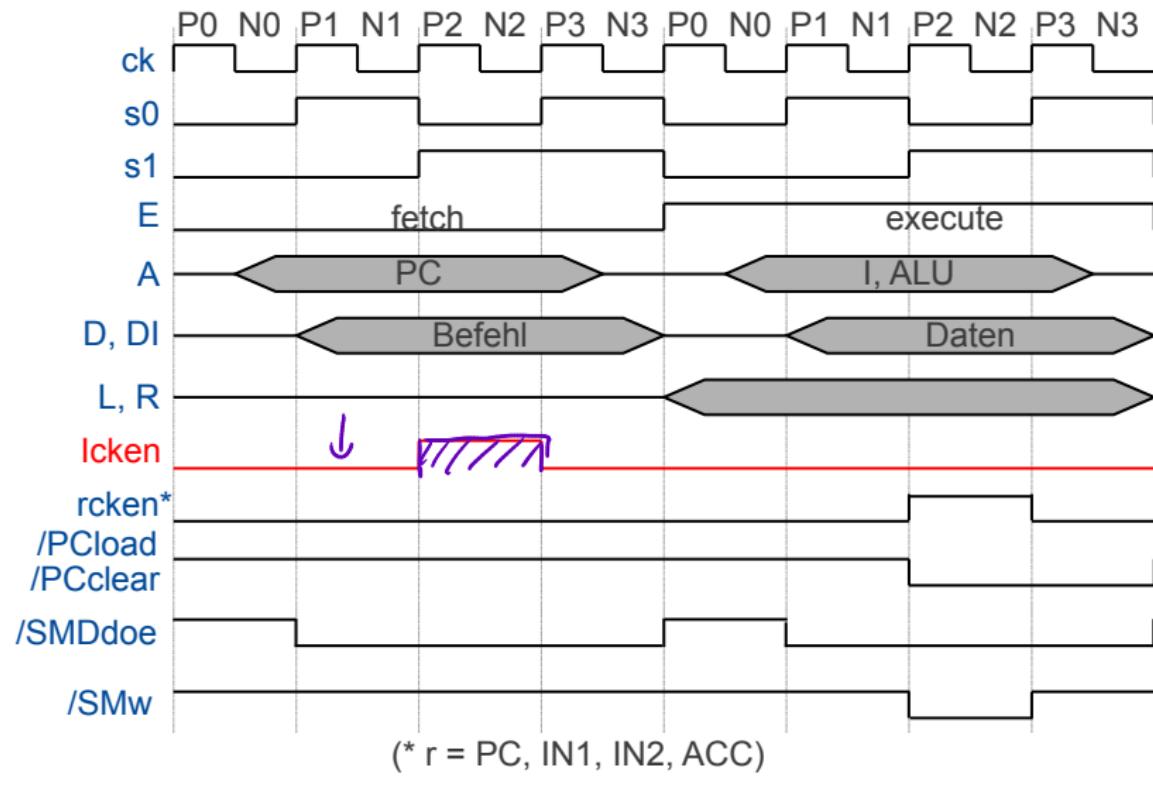
---

- Die eigentliche Kontrolllogik wird realisiert durch einen Endlichen Automaten.
- Die Kontrollsgrade sind als Ausgangssgrade des Endlichen Automaten implementiert.
- Ist ein Kontrollsgrad *active low*, dann bezeichnen wir es z.B. mit  $\underline{x}$ . Das Ausgangsgrad  $\underline{x}$  ergibt sich dann durch Negation des Ausgangsgrades  $x$  eines entsprechenden FFs mit Eingangsgrad  $x_{pre}$ .
- Ist ein Kontrollsgrad *active high*, dann bezeichnen wir es z.B. mit  $x$ . Das Ausgangsgrad  $x$  entspricht dem Ausgangsgrad eines FFs mit Eingangsgrad  $x_{pre}$ .

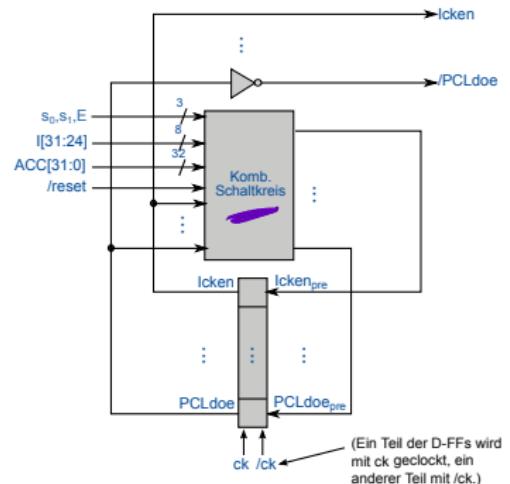
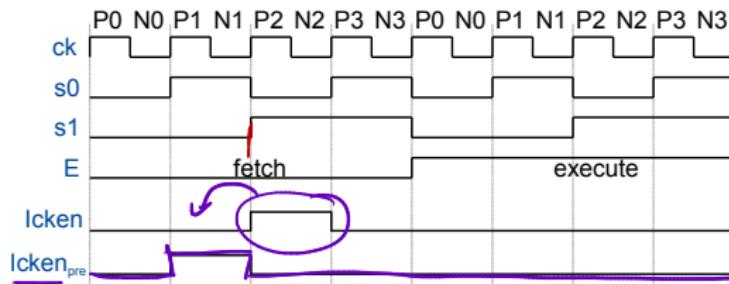
# Aufbau der Kontrolllogik (2/2)



# Berechnung von *Icken* als erstes Beispiel (1/2)



# Berechnung von Icken als erstes Beispiel (2/2)



- $Icken_{pre}$  hat steigende Flanke bei  $P1$ , fallende bei  $P2$  von Fetch.
- Realisierung:  $Icken_{pre} = \bar{E} \cdot \bar{s}_1 \cdot s_0$ .

$$S_0 = 1$$

$$S_1 = 0$$

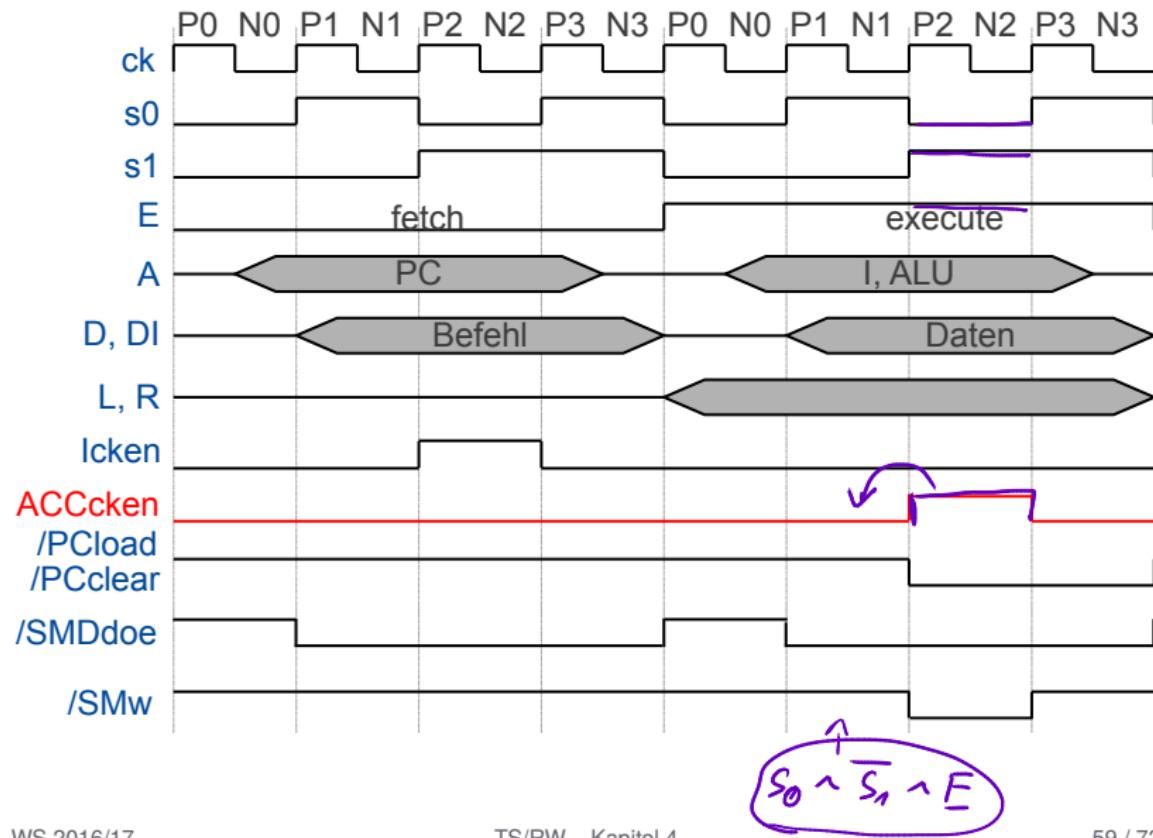
$$E = 0$$

# Berechnung von *ACCcken*, *IN1cken*, *IN2cken*, *PCcken*

---

- Analog, unter Berücksichtigung der Tatsache, dass Register nur bei bestimmten Befehlen neu beschrieben werden dürfen

# Berechnung von ACCcken als Beispiel (1/3)



## Berechnung von $ACCcken$ als Beispiel (2/3)

- $ACCcken_{pre}$  hat steigende Flanke bei  $P1$ , fallende bei  $P2$  von Execute.
- Aber nur bei folgenden Befehlen:
  - Compute mit  $D = ACC$
  - Load mit  $D = ACC$
  - Move mit  $D = ACC$

- Compute:  $\underline{I_{31}} \cdot \underline{I_{30}}$
- Load:  $\underline{I_{31}} \cdot \underline{I_{30}}$
- Move:  $\underline{I_{31}} \cdot \underline{I_{30}} \cdot I_{29} \cdot I_{28}$
- $D = ACC$ :  $\underline{I_{25}} \cdot \underline{I_{24}}$

Kodierung S, D	
S, D	Register
0 0	PC
0 1	IN1
1 0	IN2
1 1	ACC

## Berechnung von $ACC_{cken}$ als Beispiel (3/3)

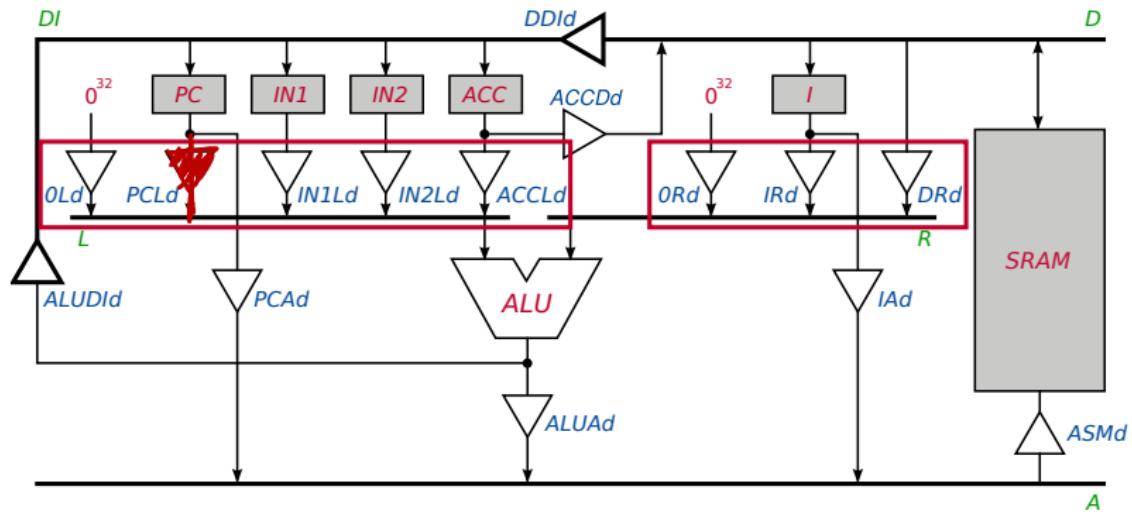
---

$$ACC_{cken}_{pre} = \frac{E \cdot \overline{s_1} \cdot s_0 \cdot}{I_{25} \cdot I_{24} \cdot} \left( \frac{\overline{I_{31}} \cdot \overline{I_{30}}}{Comp.} + \frac{\overline{I_{31}} \cdot I_{30}}{Load} + \frac{I_{31} \cdot \overline{I_{30}} \cdot I_{29} \cdot I_{28}}{Move} \right) \quad // P1 von execute$$

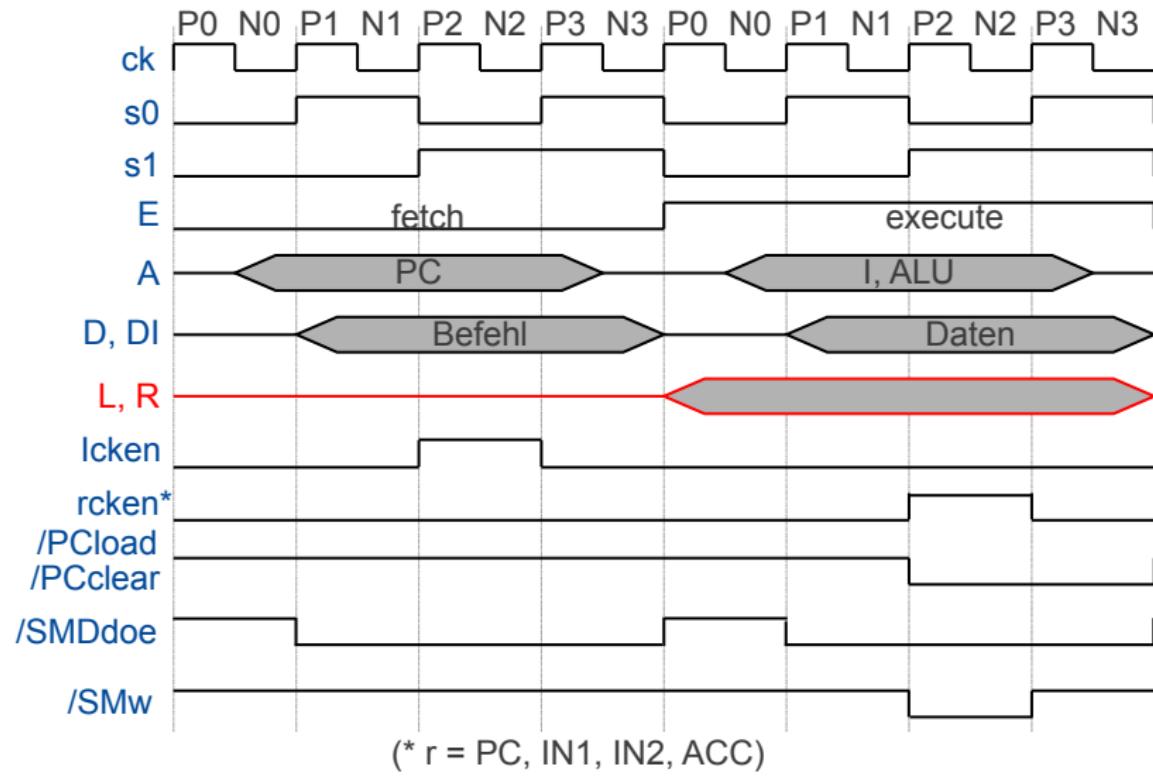
*Comp.*    *Load*    *Move*

$$\quad // D = ACC$$
$$\quad // Compute, Load oder Move$$

# Datenpfade und Treiber auf $L$ und $R$



# Idealisiertes Timingdiagramm

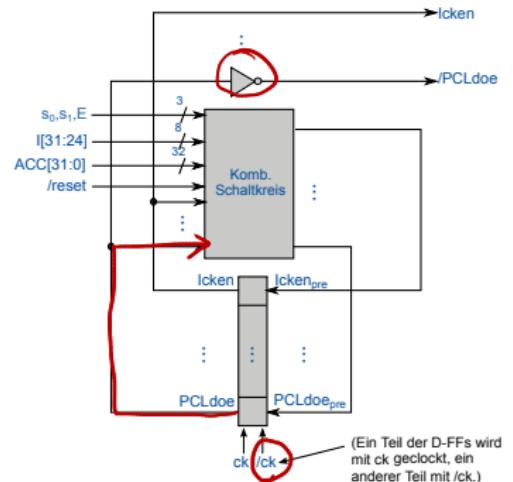
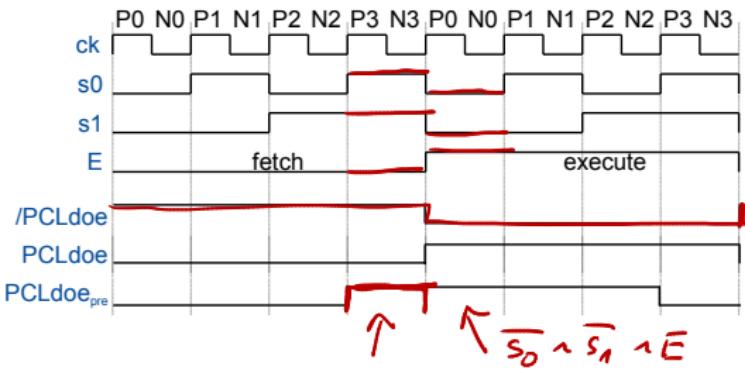


# Treiber auf Bussen $L$ und $R$

---

- $0Ld, PCLd, IN1Ld, IN2Ld, ACCLd, 0Rd, IRd, DRd.$
- Enabled in der ganzen Execute-Phase.
- Beispiel für Realisierung: /PCLdoe.
  - Enabled für
    - JUMP ( $I[31 : 30] = 11$ )
    - Compute-Befehle ( $I[31 : 30] = 00$ ) mit  $D = PC$  ( $I[25 : 24] = 00$ )
    - MOVE ( $I[31 : 28] = 1011$ ) mit  $S = PC$  ( $I[27 : 26] = 00$ ).

# Berechnung von $/PCLdoe$ (1/2)



- $PCLdoe_{pre}$  hat steigende Flanke bei  $P3$  von Fetch.

Zeitpunkt für ersten Takt, in dem  $PCLdoe_{pre} = 1$  ist:

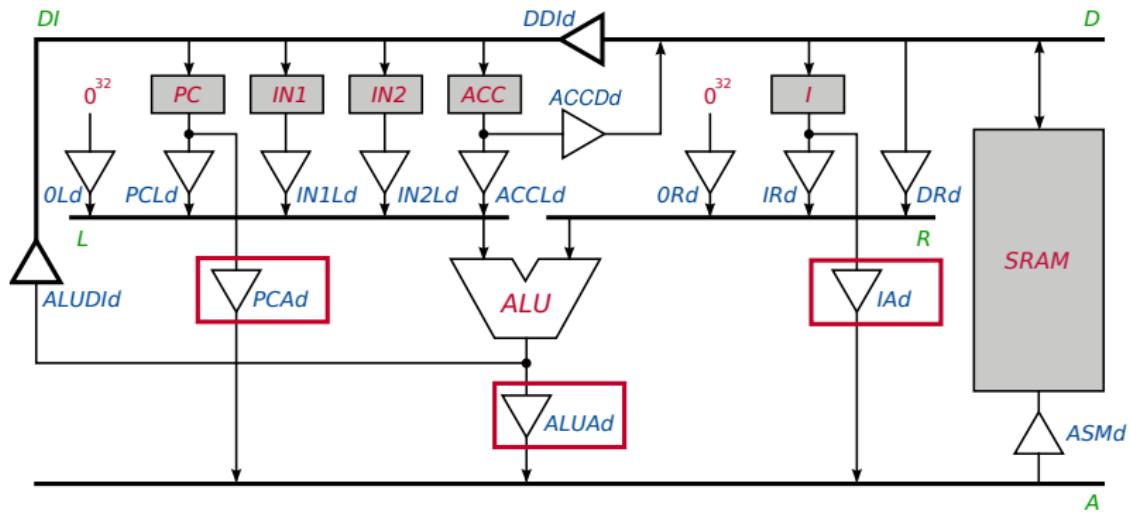
$$s_0 \wedge s_1 \wedge \neg E$$

## Berechnung von $PCLdoe$ (2/2)

$$PCLdoe_{pre} = \left| \begin{array}{l} (\bar{E} \cdot s_1 \cdot s_0 \cdot \\ [I_{31} \cdot I_{30} + \\ \underline{\overline{I_{31}} \cdot \overline{I_{30}} \cdot \overline{I_{25}} \cdot \overline{I_{24}}} \\ \underline{\overline{I_{31}} \cdot \overline{I_{30}} \cdot \overline{I_{29}} \cdot \overline{I_{28}} \cdot \overline{I_{27}} \cdot \overline{I_{26}}]) \\ + PCLdoe \cdot E \cdot \overline{s_1} \cdot \overline{s_0} \\ + PCLdoe \cdot E \cdot \overline{s_1} \cdot s_0 \\ + \underline{\overline{PCLdoe} \cdot E \cdot s_1 \cdot \overline{s_0}} \end{array} \right| \begin{array}{l} // P3 von fetch \\ // \text{JUMP} \\ // \text{Compute mit } D = PC \\ // MOVE mit \underline{S = PC} \\ // Halten in Takt 0 von execute \\ // Halten in Takt 1 von execute \\ // Halten in Takt 2 von execute \end{array}$$

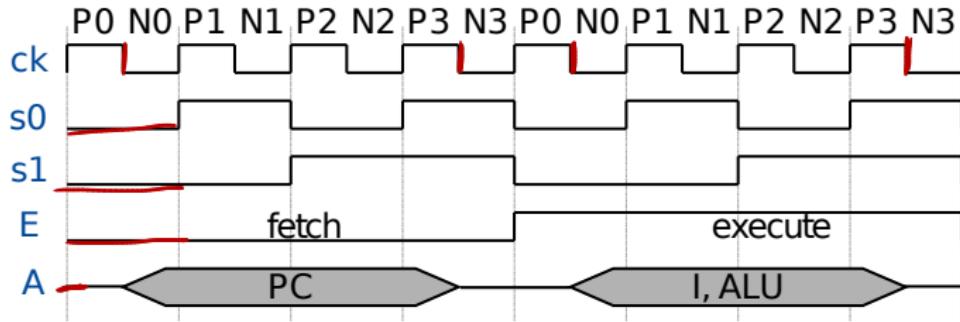
- Output-Enable-Signale für andere Treiber auf  $L$  und  $R$  analog.

# Datenpfade und Treiber auf Adressbus

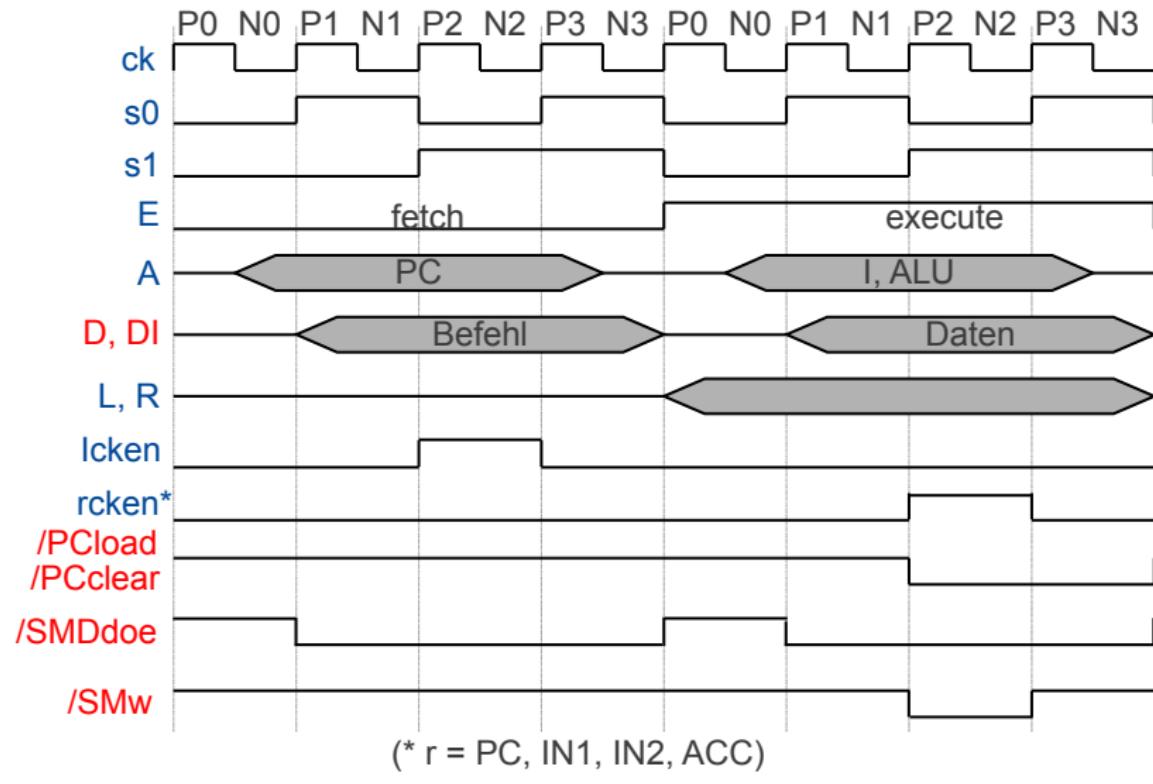


# Treiber auf Adressbus

- *PCAd*: enabled (unabhängig vom Befehl) bei  $N0$  der Fetch-Phase, disabled bei  $N3$  der Fetch-Phase.
- *IAd*, *ALUAd*: enabled bei  $N0$ , disabled bei  $N3$  von Execute (aber nicht bei allen Befehlen).
- Die D-FFs zu Output-Enable-Signalen auf dem Adressbus werden mit der invertierten Clock getaktet (Verschiebung um einen halben Takt!)



# Idealisiertes Timingdiagramm

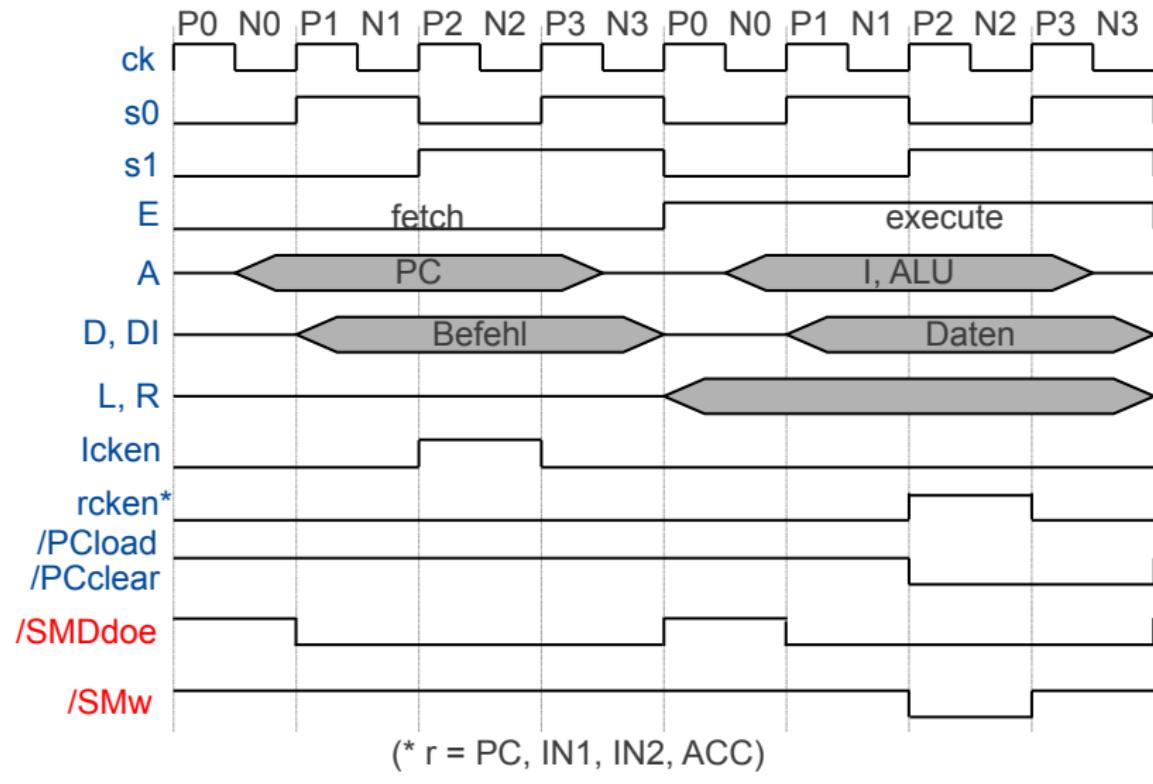


# Weitere Kontrollsignale

---

- Treiber auf  $D$ ,  $DI$
- Signale  $/PCload$ ,  $/PCclear$
- Speicheransteuerung: s. nächste Folie.
- Kontrolle der ALU und Sign-Extension:
  - Funktions-Select-Signale  $f[2:0]$  der ALU
  - Eingangsübertrag  $c_{in}$
  - $sext$  und  $fill$
  - All diese Signale werden durch den kombinatorischen Schaltkreis (ohne zusätzliche FFs) berechnet.

# Idealisiertes Timing-Diagramm



# Speicheransteuerung

---

- Output-Enable */SMDdoe* für SMDd (Treiber am Speicherausgang) aktiviert von P1 bis P0 bei Leseoperationen, d.h. bei
  - Fetch
  - Compute Memory
  - LOAD, LOADINj
- Schreibsignal für Speicher */SMw* (memory write) aktiviert von P2 bis P3 von execute bei Schreiboperationen, d.h. bei
  - STORE, STOREINj

# Zusammenfassung Sequentielle Logik

---

- Sequentielle Schaltkreise bestehen aus **speichernden Elementen** (Latches, Flipflops) und einem **kombinatorischen Kern**.
- Sie implementieren **endliche Zustandsautomaten**.
- Der Entwurf eines sequentiellen Schaltkreises besteht aus der Aufstellung des **Zustandsdiagramms**, der **Zustandsminimierung**, der **Zustandskodierung** und der **Synthese** der kombinatorischen Logik.
- Nun war es uns möglich, den **Entwurf von ReTI** zu vervollständigen (exakte Timing-Analyse folgt).

# Kapitel 5

Timing:

- 1. Physikalische Eigenschaften**
2. Timing wichtiger Komponenten
3. Exaktes Timing von ReTI

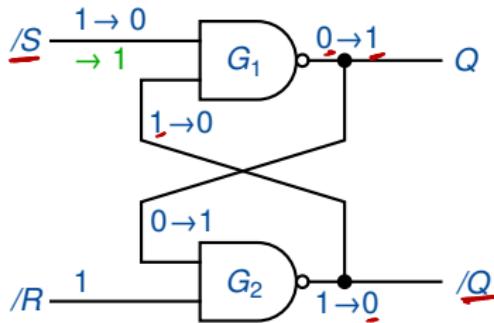
Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur  
WS 2016/17

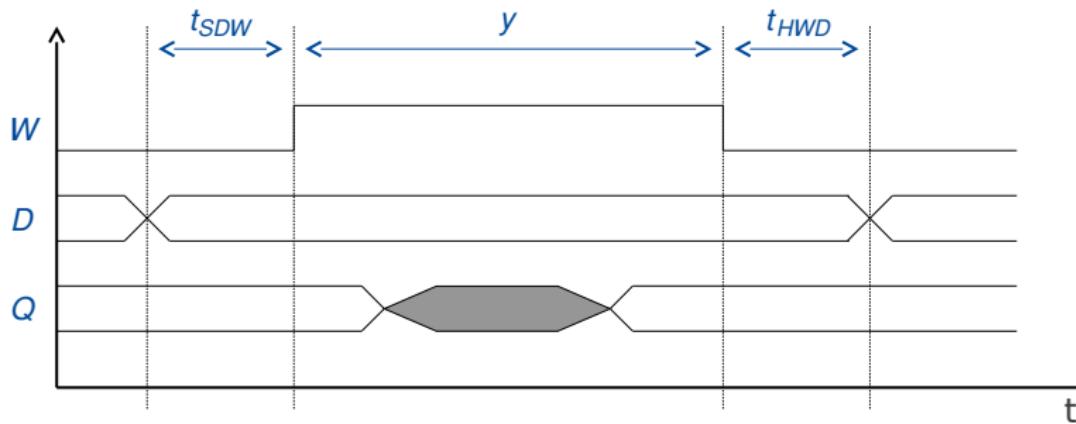
# Wiederholung: Übergang beim RS-Flipflop

- Zustand  $Q = 0 \rightarrow$  Zustand  $Q = 1$ :



- Senke  $/S$  zur Zeit  $t_0$  ab und hebe zu  $t_0 + x$  wieder an (einen solchen Signalverlauf nennt man **Puls**).
- Nach Zeit  $t_{P/SQ}$  ist  $Q = 1$ . Nach Zeit  $t_{P/S/Q}$  ist  $/Q = 0$ .
- „Gatter brauchen Zeit zum Schalten!“ Aber wie lange ist  $t_{P/SQ}$ ,  $t_{P/S/Q}$ ? Oder wie lange muss ein Puls mindestens dauern? (= Pulswelte).

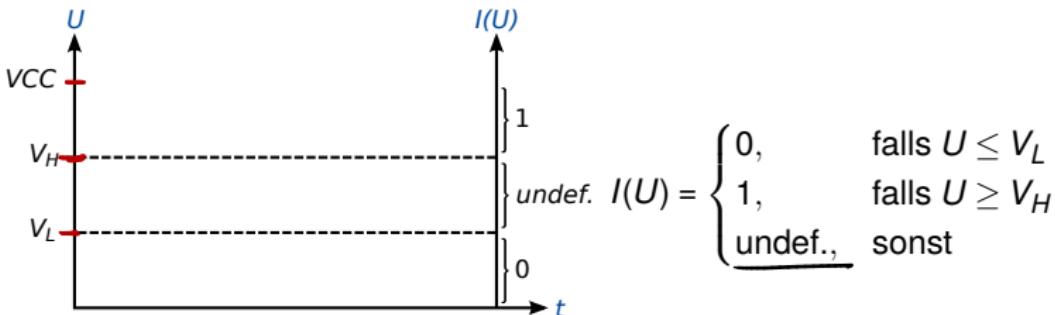
# Wiederholung: Timing-Diagramm D-LATCH



- Wie lange müssen die einzelnen Signale aktiv sein, damit der Schreibvorgang reibungslos abläuft?
- D. h. Wie lange ist Setup-Zeit  $t_{SDW}$ , Hold-Zeit  $t_{HWD}$ , Pulsweite  $y$ ?

$t_{HWD}$

# Physikalische Signale $\leftrightarrow$ Logische Signale

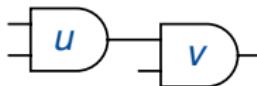


- In jeder Technologie gibt es eine Versorgungsspannung  $VCC$  (z.B. 1.1 V bei NanGate).
- Eine Spannung  $U \in [0, VCC]$  wird als logischer Wert  $I(U)$  interpretiert.
  - Am Eingang (Input) eines Gatters:  $V_{IL}, V_{IH}$ .
  - Am Ausgang (Output) eines Gatters:  $V_{OL}, V_{OH}$ .
- $V_{IL}, V_{IH}, V_{OL}, V_{OH}$  eines Bausteins sind gegeben.

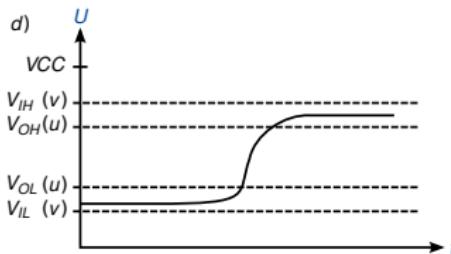
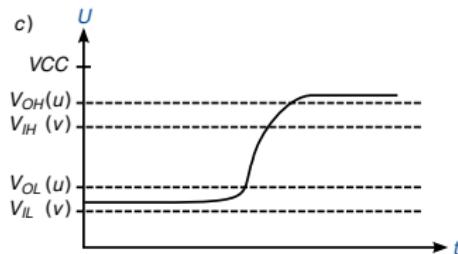
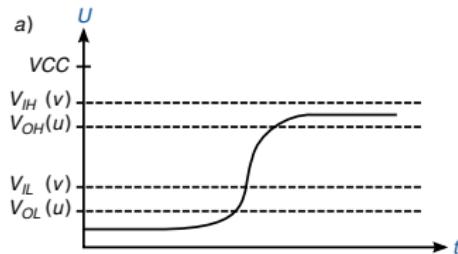
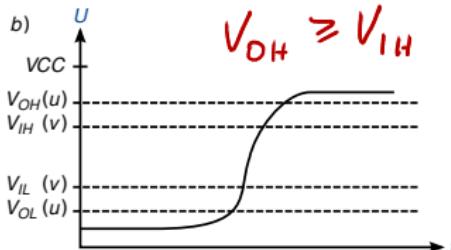
# SMILE – Physikalische Signale



Unter welchen Bedingungen laufen die Schaltvorgänge bei untenstehenden Schaltung reibungslos ab?

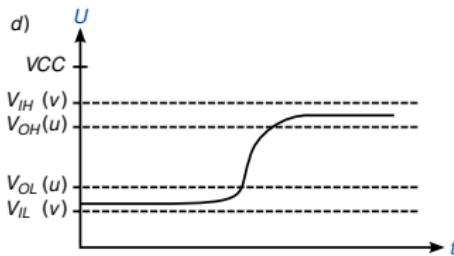
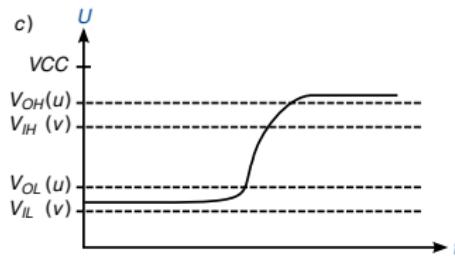
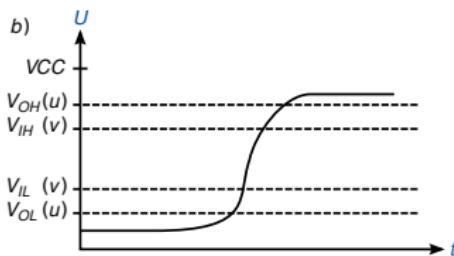
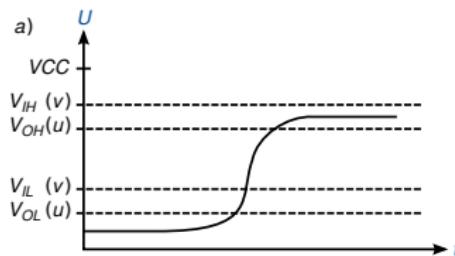
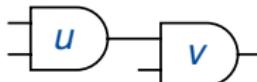


$$V_{OL} \leq V_{IL}$$



# SMILE – Physikalische Signale

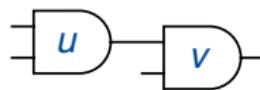
Unter welchen Bedingungen laufen die Schaltvorgänge bei der untenstehenden Schaltung reibungslos ab?



$\Rightarrow$  Fazit: Es sollte  $V_{OL}(u) \leq V_{IL}(v)$  und  $V_{OH}(u) \geq V_{IH}(v)$  gelten.

# Zusammenschalten von Gattern

---

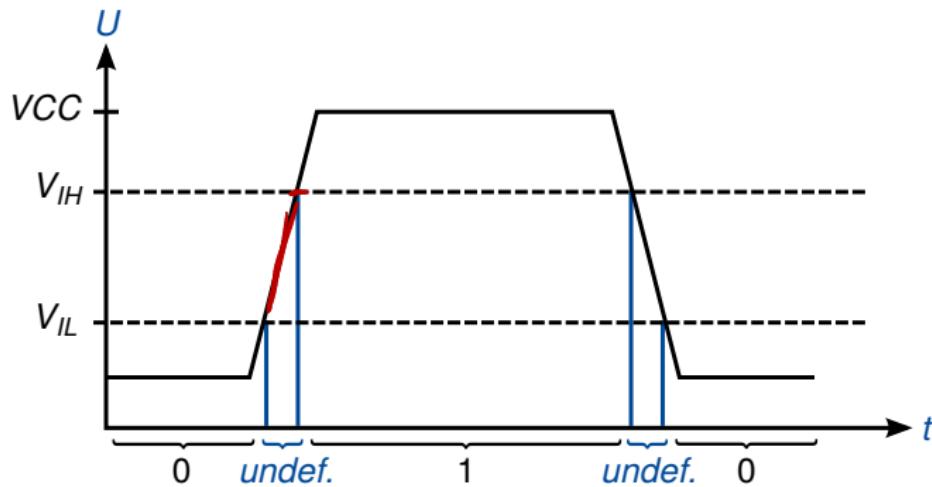


- Will man den Ausgang eines Gatters  $u$  mit dem Eingang eines Gatters  $v$  verbinden, dann sollte gelten:
  - $V_{OL}(u) \leq V_{IL}(v)$  und
  - $V_{OH}(u) \geq V_{IH}(v)$ .
- Sonst werden Signale falsch interpretiert.

# Beispiel: NanGate

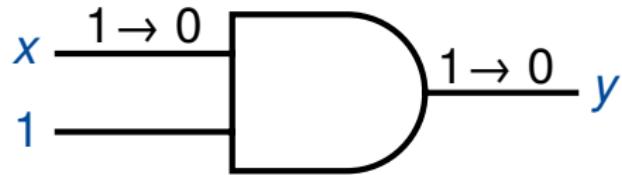
$$V_{IL} = 30\% \cdot VCC = 0.33 \text{ V}$$
$$V_{IH} = 70\% \cdot VCC = 0.77 \text{ V}$$

Entsprechend Output-Pegel  
 $V_{OL}, V_{OH}$ .

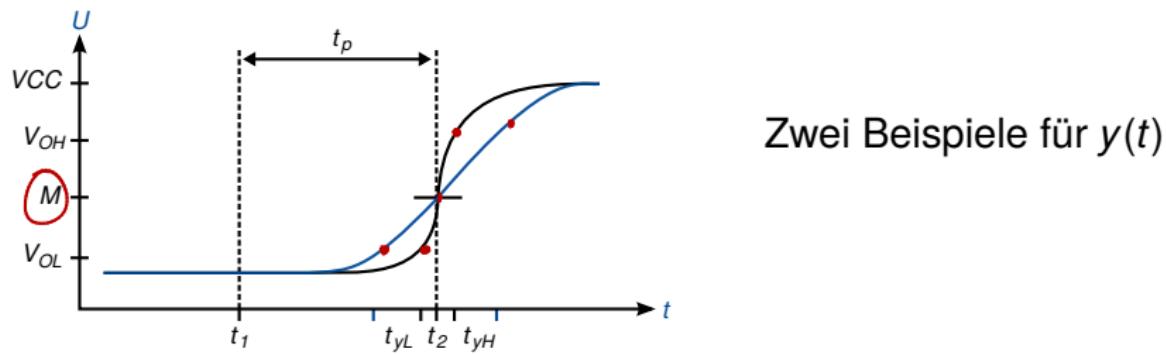
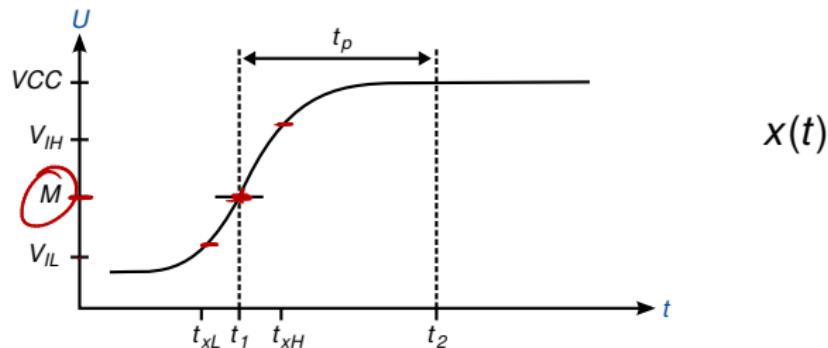


# Verzögerung

---



# Beispiel-Spannungsverlauf $x(t)$ , $y(t)$



Zwei Beispiele für  $y(t)$

# Allgemeine Bemerkung zu Verzögerungszeiten

---

- Im Allgemeinen gilt nicht  $y(t) = x(t - t_p)$ , so dass man nicht einfach  $t_p$  als Verzögerungszeit definieren kann.  
 $y(t)$  wird verformt.
- Die Verzögerungszeit (Propagation Delay) wird definiert als  $t_p := (t_2 - t_1)$  bezüglich einer festen „Referenzspannung“  $M$  mit  $V_L < M < V_H$   
(Bsp.:  $M = 0.5V_{CC} = 0.55\text{ V}$  bei NanGate).
- Bestimme  $t_1, t_2$  mit  $x(t_1) = y(t_2) = M$ .

# Angaben zur Verzögerungszeit

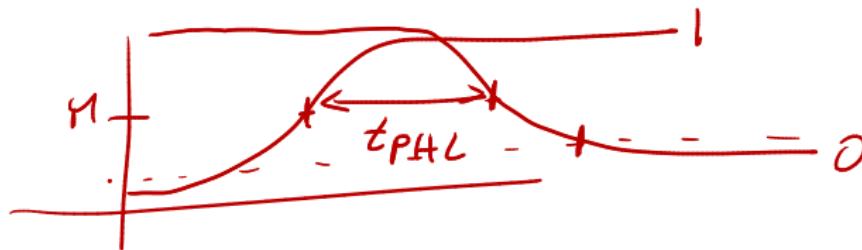
---

- In der Regel gibt es **verschiedene** Verzögerungszeiten für Übergänge am Ausgang:
  - $t_{PLH}$ : Verzögerungszeit bei  $0 \rightarrow 1$ .
  - $t_{PHL}$ : Verzögerungszeit bei  $1 \rightarrow 0$ .

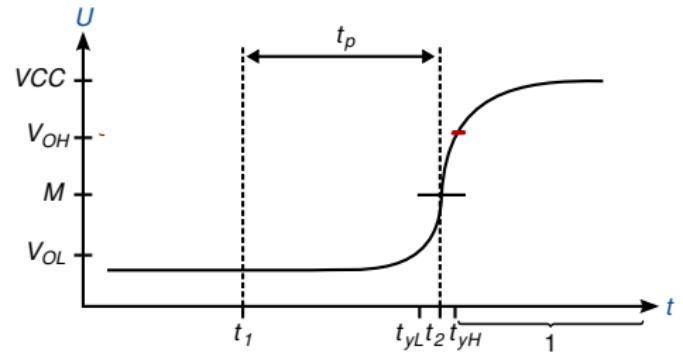
# Modellierung der Verzögerungszeit

- **Problem** bei der Modellierung der Verzögerungszeit bezüglich fester Spannung  $M$ :

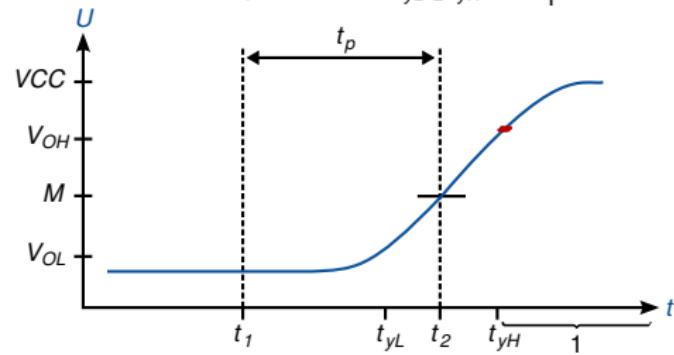
- Keine Aussage darüber, wann logische Signale 0 oder 1 sind, d. h. physikalische Signale unterhalb  $V_{OL}$  oder oberhalb  $V_{OH}$  sind.



# Illustration des Problems



→ Ähnliches Problem am Gattereingang.



# Anstiegs- und Abfallzeiten

---

- Für jedes Signal braucht man also zusätzliche Informationen über:
  - Anstiegszeit (Rise Time) =  
Zeit, in der Signal von  $V_L$  nach  $V_H$  steigt.
  - Abfallzeit (Fall Time) =  
Zeit, in der Signal von  $V_H$  nach  $V_L$  fällt.
  - Bzw. noch genauer würde man eigentlich benötigen:
    - Anstiegszeit von  $M$  nach  $V_H$
    - Abfallzeit von  $M$  nach  $V_L$

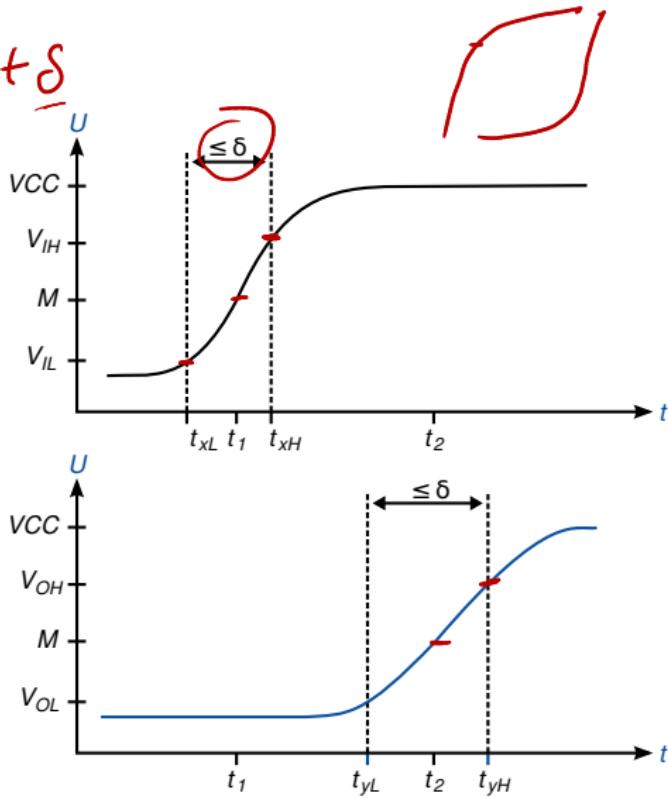
# Beschränkung dieser Zeiten

$$t_{yH} - t_{xL} \leq \underline{\delta} + t_{PLH} + \underline{\delta}$$

- Die in unseren Analysen verwendeten Gatter haben die folgende angenehme Eigenschaft:

- $\exists \delta$  mit folgender Eigenschaft:

Falls rise/fall time  $\leq \delta$  am Gattereingang, dann rise/fall time  $\leq \delta$  am Gatterausgang.



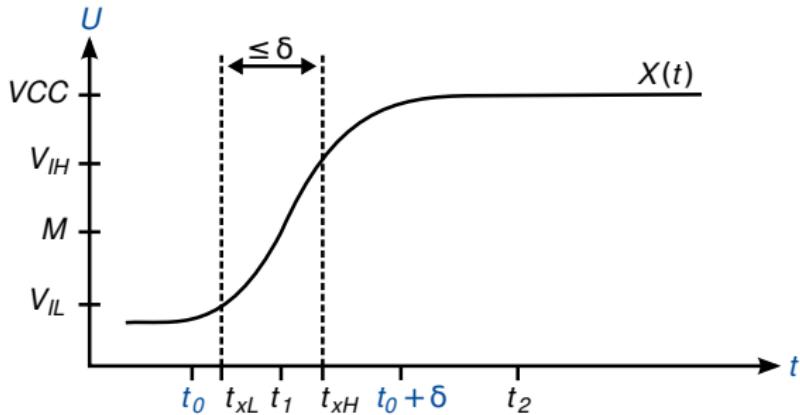
## Beispiel: NanGate

---

- $V_{IL} = 30\% \cdot VCC = \underline{0.33} \text{ V}$
- $V_{IH} = 70\% \cdot VCC = \underline{0.77} \text{ V}$
- NanGate für  $M = \underline{0.55} \text{ V}$  spezifiziert.  
Bausteine *NAND, NOT, AND, OR, EXOR*.
- $t_p$  zwischen 0.00 ns und 0.21 ns.
- $\delta = \underline{0.13} \text{ ns}$  ( $1 \text{ ns} = 10^{-9} \text{ s}$ )
- Die Zeiten, an denen die entsprechenden Signale  
*wohldefinierte logische Werte 0, 1* annehmen,  
unterscheiden sich von denen für  $M$  um *höchstens*  $\delta$ .

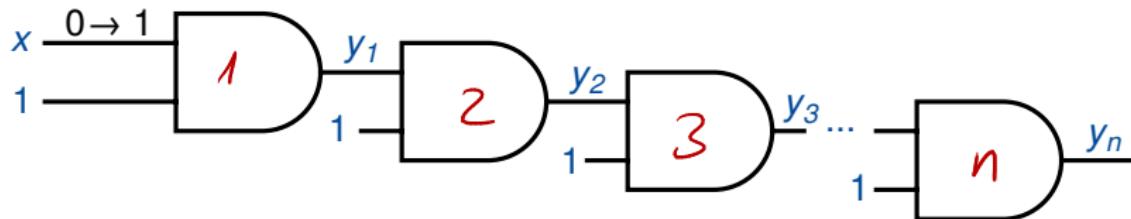
## Bemerkung

- Eine rise/fall time  $\leq \delta$  an den primären Eingängen einer Schaltung kann man garantieren, wenn man den Schaltvorgang zur Zeit  $t_0$  beginnt und spätestens zur Zeit  $t_0 + \delta$  abschließt.

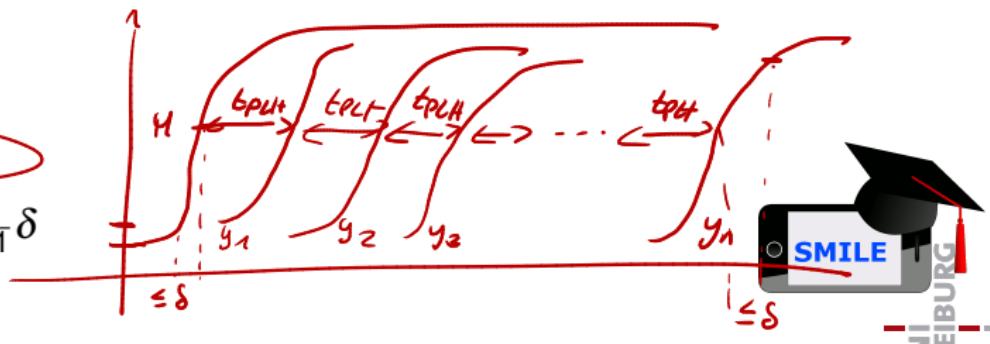


# SMILE – Verzögerungszeit

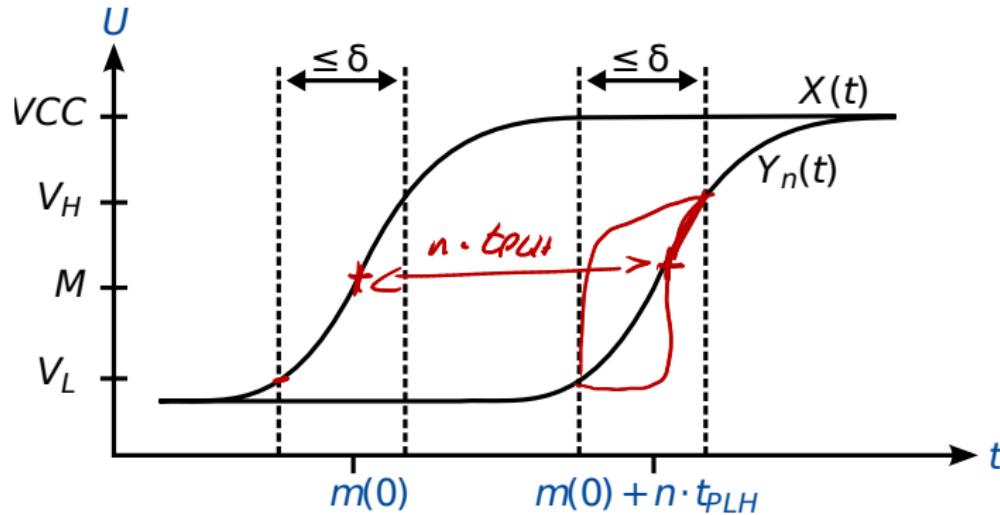
Nach welcher Zeit ist der Schaltvorgang bei  $Y_n$  von 0 nach 1 spätestens abgeschlossen, nachdem der Schaltvorgang von 0 nach 1 bei  $x$  angefangen hat?



- a.  $n \cdot (t_{PLH} + \delta)$
- b.  $t_{PLH} + n\delta$
- c.  $n \cdot t_{PLH} + 2\delta$
- d.  $n \cdot t_{PLH} + \frac{n}{n-1}\delta$



## Analyse der Verzögerungszeit einer Kette von $n$ Gattern (2/3)



## Analyse der Verzögerungszeit einer Kette von $n$ Gattern (3/3)

---

- Durchläuft  $X(t)$  nach Zeit  $m(0)$  die Spannung  $M$ , dann durchläuft  $Y_n(t)$  die Spannung  $M$  nach  $m(0) + n \cdot t_{PLH}$ .
- Falls  $X(t)$  mit Anstiegszeit  $\leq \delta$ , dann auch  $Y_1(t), \dots, Y_n(t)$ .
- Also ist  $Y_n$  auf jeden Fall zur Zeit  $m(0) + n \cdot t_{PLH} + \delta$  logisch 1.
- Beginnt man im Beispiel den Schaltvorgang bei  $t_0$  und beendet ihn bei  $t_0 + \delta$ , dann gilt  $m(0) \leq t_0 + \delta$  und  $Y_n$  ist spätestens nach  $t_0 + n \cdot t_{PLH} + 2\delta$  logisch 1.  
 $t_0 + n \cdot t_{PLH} + 2\delta$

# Vereinbarungen

---

- Im Folgenden soll

Signal  $X$  wird zum Zeitpunkt  $\underline{t_1}$  abgesenkt/angehoben  
bedeuten

$X$  wird abgesenkt/angehoben mit  $\underline{X(t_1) = M}$ .

- Des Weiteren sind alle Zeitangaben in  $\underline{ns}$ .

# Einfluss auf Verzögerungszeiten

---

- Verzögerungszeiten von Gattern sind **nicht konstant**, sondern werden beeinflusst durch:
  - Betriebstemperatur
  - Fertigungsprozess des Chips
  - kapazitive Last am Gatterausgang (Fanout)  
(Gattereingänge, die mit einem Gatterausgang verbunden sind, verhalten sich wie Kondensatoren, d. h. sie werden beim Schalten geladen bzw. entladen.)

# Worst-case Timing-Analyse

---

- Wegen Abhangigkeit der Verzogerungszeit von Temperatur, Fertigungsprozess und Fanout werden vom Hersteller **keine festen Zeiten  $t_{PLH}/t_{PHL}$**  angegeben, sondern 3 Werte:
  - $t_{\underline{}}^{min}$  = untere Schranke
  - $t^{max}$  = obere Schranke
  - $t^{typ}$  = *typischer Wert* (???)

## *min, max und typ* (1/2)

- Für die tatsächliche Verzögerungszeit  $t_p$  gilt:

$$t^{\min} \leq t_p \leq t^{\max}$$

---

- Wir nehmen in den folgenden Analysen an, dass  $t_p$  im Intervall  $[t^{\min}, t^{\max}]$  liegt, falls
  - die Temperatur im Bereich  $T$  liegt („kommerzieller Temperaturbereich“  $0 - 70^\circ\text{C}$ , „militärischer Temperaturbereich“  $-55 - 125^\circ\text{C}$ )
  - und eine bestimmte kapazitive Last  $C_0$  nicht überschritten wird.
- $C_0$  wird so gewählt, dass mit Einhalten einer Fanoutbeschränkung von 10  $C_0$  auf keinen Fall überschritten wird.

## *min, max und typ* (2/2)

- Für  $t^{typ}$  gilt ebenfalls  $t^{min} \leq t^{typ} \leq t^{max}$ .
  - Beim Rechnen mit  $t^{typ}$  macht man aber einen Fehler mit unbekannter Größe.
- Kein Rechnen mit  $t^{typ}$ , sondern mit Intervallen  $[t^{min}, t^{max}]$ .



# Exkurs: Rechnen mit Intervallarithmetik (1/2)

## Definition

Ein Intervall  $[a, b] := \{x \in \mathbb{R} \mid a \leq x \leq b\} \subset \mathbb{R}$  auf  $\mathbb{R}$  ist eine zusammenhängende und abgeschlossene Teilmenge von  $\mathbb{R}$ . Man bezeichnet es auch als das abgeschlossene Intervall von  $a$  bis  $b$ .

- Wir betrachten hier nur die Menge der abgeschlossenen Intervalle IR auf  $\mathbb{R}$ .
- Es gilt:
  - $\min[a, b] = a$
  - $\max[a, b] = b$
  - $a \in \mathbb{R} \simeq [a, a] \in \text{IR}$   
(eine reelle Zahl  $a$  kann aufgefasst werden als das Punktintervall von  $a$  bis  $a$ )

$$(a, b] \quad a < x \leq b$$

# Exkurs: Rechnen mit Intervallarithmetik (2/2)

## Definition

Gegeben ein Operator  $\text{\texttt{op}} \in \{+, -, \cdot\}$  in  $\mathbb{R}$ . Der dazugehörige Operator  $\text{\texttt{@op}}$  auf IR ist definiert als:

Für  $a, b, c, d \in \mathbb{R}$ :

$$[a, b] \text{\texttt{@op}} [c, d] := \{x \text{\texttt{ op }} y \mid x \in [a, b], y \in [c, d]\}$$

Beispiele:

$$[2, 3] \text{\texttt{@oplus}} [0, 5] = [-2, 8]$$

$$[2, 3] \text{\texttt{@ominus}} [0, 5] = [-3, 3]$$

- $[a, b] \text{\texttt{@oplus}} [c, d] = [\underline{a+c}, \underline{b+d}]$
- $[a, b] \text{\texttt{@ominus}} [c, d] = [a - d, b - c]$
- $[a, b] \text{\texttt{@otimes}} [c, d] = [\min(\underline{a \cdot c}, \underline{a \cdot d}, \underline{b \cdot c}, \underline{b \cdot d}), \max(\underline{a \cdot c}, \underline{a \cdot d}, \underline{b \cdot c}, \underline{b \cdot d})]$

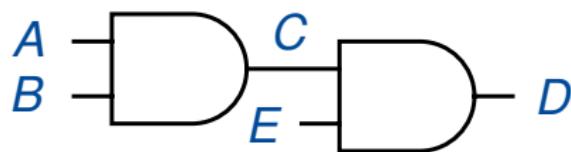
# Bemerkungen

---

- Wir schreiben vereinfachend nur  $\text{\textcircled{op}}$  statt  $\text{\textcircled{op}}$ .
- Für unsere Belange sind ausschließlich  $+$ ,  $-$  und  $\cdot$  Operator von Bedeutung. ( $\cdot$  mit Punktinvervallen)
- Ein Intervall bezeichnen wir mit  $\tau = [t^{\min}, t^{\max}]$ .

# Beispiel: AND-Gatter

---



AND

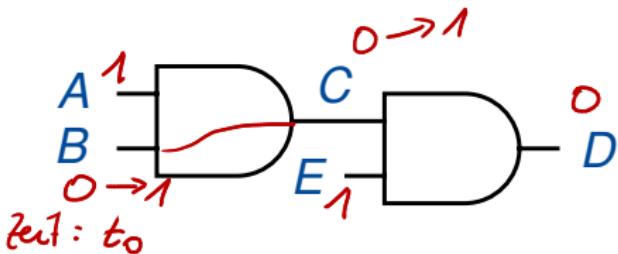
$$\tau_{PLH} = [0.02, 0.12]$$

$$\tau_{PHL} = [0.02, 0.12]$$

Bzw.:

AND	<u><math>t^{\min}</math></u>	<u><math>t^{\max}</math></u>
$\tau_{PLH}$	0.02	0.12
$\tau_{PHL}$	0.02	0.12

# Fall 1



AND	$t^{\min}$	$t^{\max}$
$\tau_{PLH}$	0.02	0.12
$\tau_{PHL}$	0.02	0.12

■  $A, E$  fest auf 1.

■  $B$  von 0 auf 1 zum Zeitpunkt  $t_0$ .

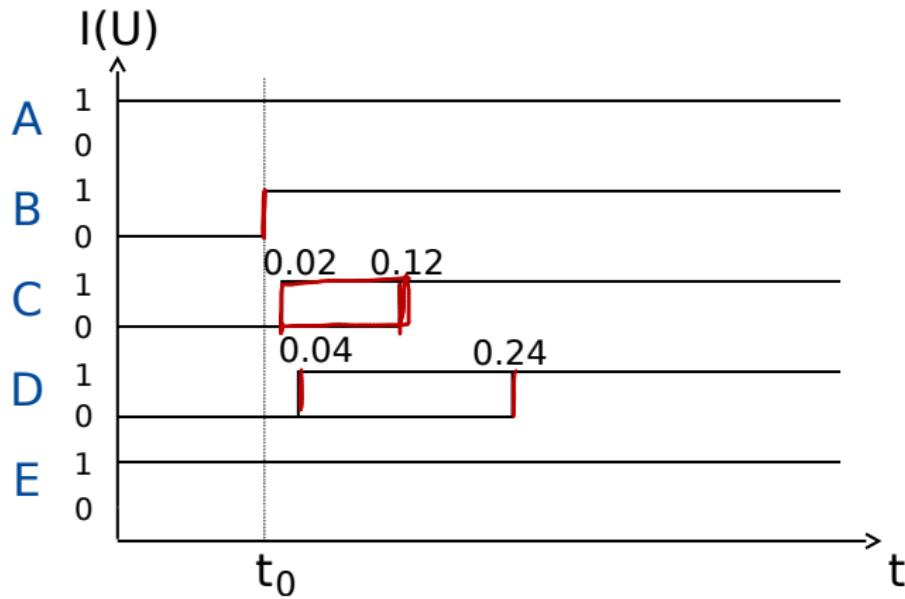
→ Änderung von  $C$  zur Zeit

$$\begin{aligned}\tau_1 &= t_0 + \tau_{PLH} (\text{AND}) \\ &= t_0 + [0.02, 0.12]\end{aligned}$$

→ Änderung von  $D$  zur Zeit

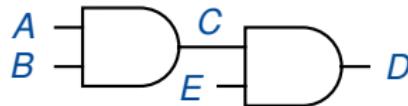
$$\begin{aligned}\underline{\tau_2} &= \tau_1 + \tau_{PLH} (\text{AND}) \\ &= t_0 + 2 \cdot \tau_{PLH} (\text{AND}) \\ &= t_0 + 2 \cdot [0.02, 0.12] \\ &= t_0 + [0.04, 0.24]\end{aligned}$$

# Fall 1 - Timing-Diagramm



## Fall 2

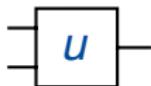
---



- $A, B, E$  können sich zum Zeitpunkt  $t_0$  ändern, sind vorher und nachher stabil.
- Es ist unbekannt, wieviele Signale sich ändern und wie sie sich ändern.  
→ Gröbere Abschätzungen

# Gröbere Abschätzung

- Bestimmung von Zeitintervallen, zu denen Gatter überhaupt schalten können.
- Beispiel:



$$\begin{aligned}\tau_{PLH} &= [t_{PLH}^{\min}, t_{PLH}^{\max}] \\ \tau_{PHL} &= [t_{PHL}^{\min}, t_{PHL}^{\max}]\end{aligned}$$

*U* ist beliebiges Gatter oder eine Schaltung.

- Gesucht ist ein Intervall  $\tau_p$  in dem die Verzögerungsintervalle aller möglichen Schaltvorgänge enthalten sind.

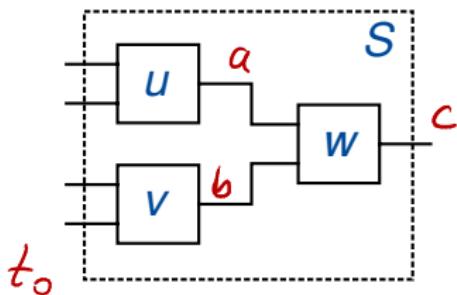
■ Definiere

$$\begin{aligned}t_p^{\min} &:= \underline{\min}(t_{PLH}^{\min}, t_{PHL}^{\min}) \\ t_p^{\max} &:= \underline{\max}(t_{PLH}^{\max}, t_{PHL}^{\max})\end{aligned}$$

- Dann ist  $\tau_p := [\underline{t_p^{\min}}, \underline{t_p^{\max}}]$  das gesuchte Intervall.

# SMILE – Timing Abschätzung

In welchem Intervall  $\tau_p(S)$  kann  $S$  schalten?



$$\tau_p(U) = [a_1, b_1]$$

$$\tau_p(V) = [a_2, b_2]$$

$$\tau_p(W) = [a_3, b_3]$$

$t_0$

$$a : t_0 + [\bar{a}_1, \bar{b}_1]$$

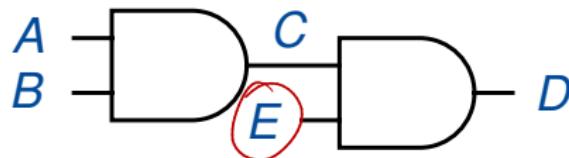
$$b : t_0 + [\bar{a}_2, \bar{b}_2]$$

$$\left[ \min(a_1, a_2), \max(b_1, b_2) \right] \\ + [a_3, b_3]$$



## Fall 2

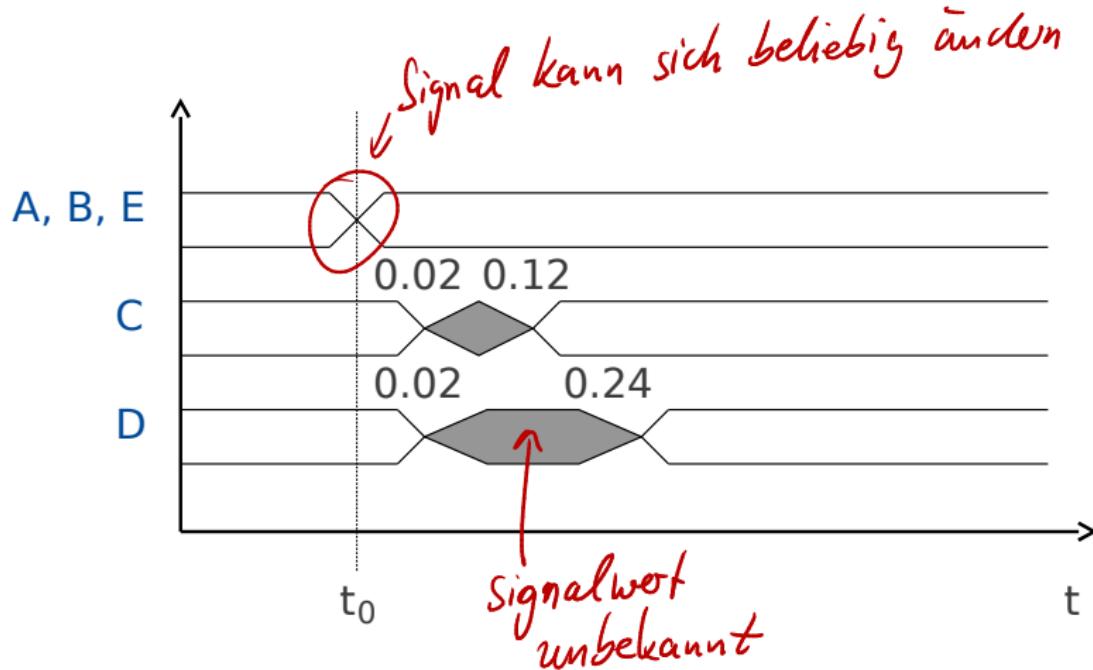
---



AND	$t^{\min}$	$t^{\max}$
$\tau_{PLH}$	0.02	0.12
$\tau_{PHL}$	0.02	0.12

- Wenn die Gatter schalten, dann in folgenden Intervallen:
  - A, B, E:  $t_0 + [0.0, 0.0]$
  - C:  $t_0 + [0.02, \underline{0.12}]$
  - D:  $t_0 + [\underline{0.0}, 0.12] + [\underline{0.02}, \underline{0.12}] = t_0 + [0.02, 0.24]$

## Fall 2 - Timing-Diagramm



# Interpretation des Timing-Diagramms

- Was kann im grauen Bereich passieren?

- Beispiel:**

$t_0: \underline{A, B, E} \quad 110 \rightarrow 101$



- Annahme:**

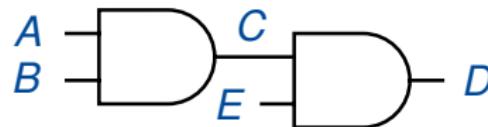
AND-Gatter haben folgende Verzögerungszeiten.

- 1. AND-Gatter:  $t_{PLH} = 0.12, \quad t_{PHL} = 0.12$
- 2. AND-Gatter:  $t_{PLH} = 0.02, \quad t_{PHL} = 0.02$

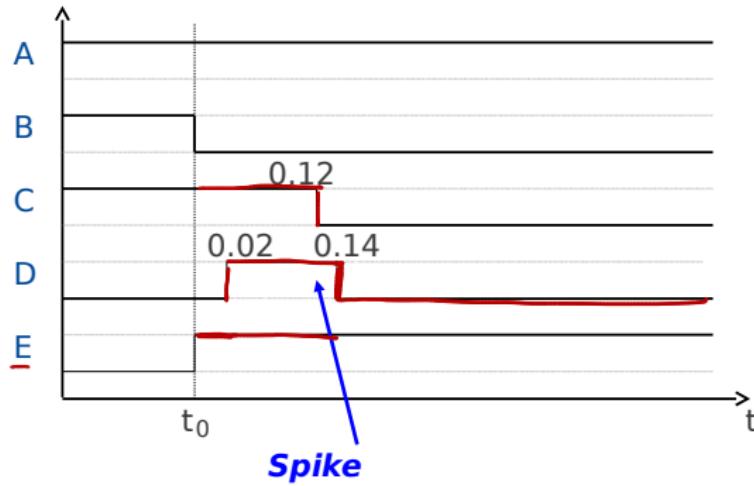
$$t_{PLH}^{\text{AND}} = [0.02, 0.12]$$

# Timing-Diagramm zum Beispiel

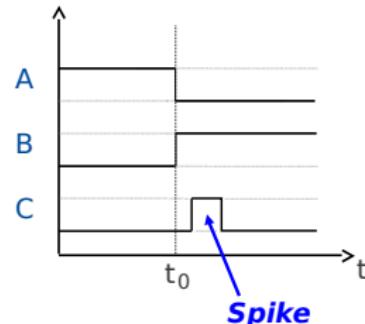
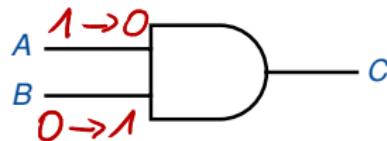
$m0 \rightarrow m1$



1. AND:  $t_{PLH} = 0.12$      $t_{PHL} = 0.12$   
2. AND:  $t_{PLH} = 0.02$      $t_{PHL} = 0.02$



# Spikefreies Umschalten von Gattern



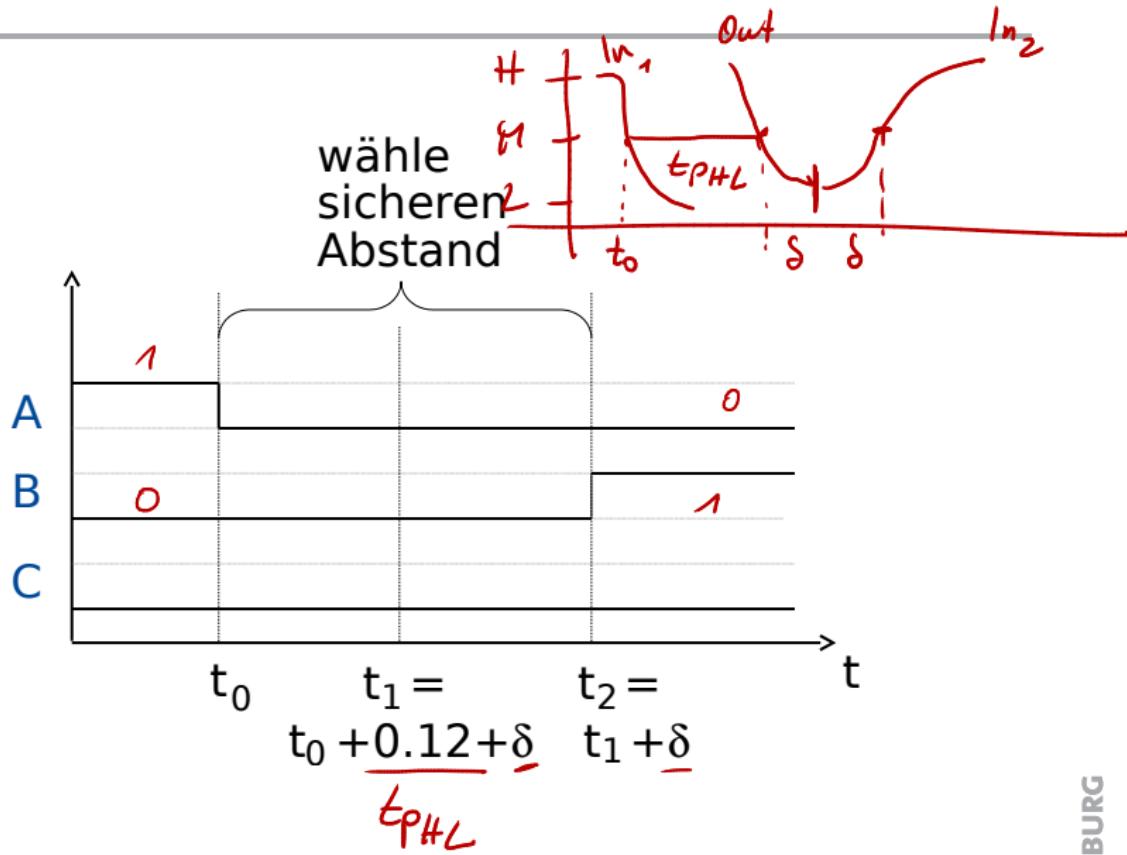
## Ziel:

Übergang von  $A = 1, B = 0$  zu  $A = 0, B = 1$ , ohne Spike am Ausgang.

## Bemerkung:

Der Übergang  $(0, 1) \rightarrow (1, 0)$  bzw. umgekehrt ist der einzige, bei dem an AND/NAND-Gattern ein Spike auftreten kann.

# AND-Gatter



## Lemma

Man kann zeigen, dass Übergänge für  $A$  und  $B$  mit

$$0.12\text{ ns} + 2\delta = 0.38\text{ ns}$$

sicher sind, d. h. keine Spikes am Ausgang entstehen können.

# Zum Beweis - Timing im Gatter

1 Senke  $A$  bei  $t_0 = 0$ .

→  $C = 0$  wegen  $A = 0$  spätestens bei  $t_1 = t_0 + 0.12 + \delta$

■ Grund:

- Bei tatsächlichem Schalten von  $C = 0$  wegen  $A = 0$  würde das Signal spätestens nach  $t_{PHL}^{max} = 0.12$  ns den Wert  $M$  durchlaufen und wäre 0 spätestens nach  $0.12 + \delta$  ns.
- Interner Umschaltvorgang „ $C = 0$  wegen  $A = 0$ “ muss also spätestens nach  $0.12 + \delta$  ns beendet sein.

2 Hebe  $B$  (bzgl.  $M$ !) zum Zeitpunkt  $t_2 = t_1 + \delta$ .

→ Zum Zeitpunkt  $t_1$  gilt auf jeden Fall noch  $B = 0$ .

■ Also:

Vor  $t_1$ :  $B = 0 \Rightarrow C = 0$

Nach  $t_1$ :  $A = 0 \Rightarrow C = 0$

→ Übergänge für  $A$  und  $B$  mit Abstand  
 $t_2 - t_0 = 0.12 + 2\delta = 0.38$  ( $\delta = 0.13$ ).

<u>AND</u>	$t^{\min}$	$t^{\max}$
$\tau_{PLH}$	0.02	0.12
$\tau_{PHL}$	0.02	0.12

# Regel für spikefreies Umschalten

- Wähle den Abstand für die Signaländerungen am Eingang eines Gastes so, dass
  - die maximale Verzögerung des ersten Schaltvorganges *und*
  - $2 \times$  die rise-/fall-time ( $\delta$ )zwischen den beiden Schaltvorgängen am Eingang liegt.

$$\delta = 0.13$$

- Beispiel: NAND



NAND	$t^{\min}$	$t^{\max}$
$\rightarrow \tau_{PLH}$	0.02	0.15
$\tau_{PHL}$	0.02	0.12

- Kritischer Übergang: Zuerst  $A : 1 \rightarrow 0$ , dann  $B : 0 \rightarrow 1$ .
- Daraus ergibt sich der Abstand  $\max(\tau_{PLH}) + 2\delta = 0.41$

$$0.15 + 2 \cdot 0.13 =$$



# Kapitel 5

Timing:

1. Physikalische Eigenschaften
2. **Timing wichtiger Komponenten**
3. Exaktes Timing von ReTi

Albert-Ludwigs-Universität Freiburg

UNI  
FREIBURG

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur  
WS 2016/17

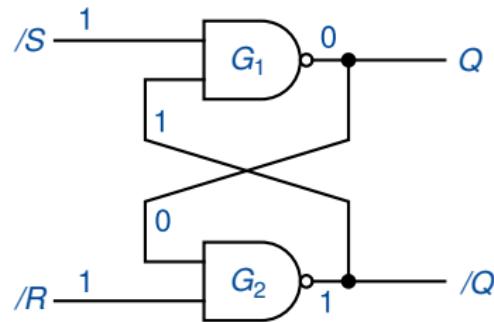
# Timing - Übersicht

---

- Timing für ein paar (bereits bekannte) Schaltpläne:
  - RS-Flipflop
  - D-Latch
  - D-Flipflop
- Timing weiterer Komponenten, die bei der Realisierung der ReTI genutzt werden:
  - Kontrolllogik
  - Register mit Clock-Enable
  - ALU
  - Speicher

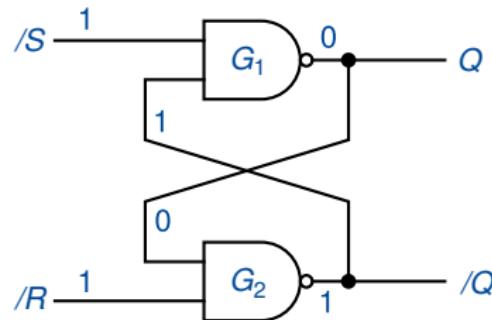
# RS-Flipflop

- Zustand  $Q = 0 \rightarrow$  Zustand  $Q = 1$ :



# RS-Flipflop

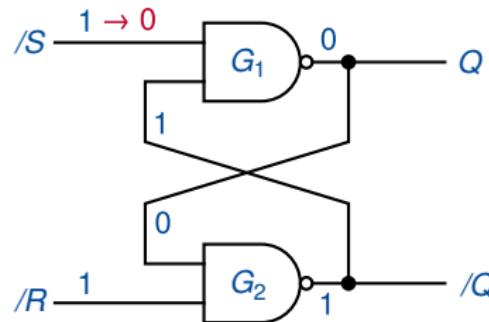
- Zustand  $Q = 0 \rightarrow$  Zustand  $Q = 1$ :



- Senke  $/S$  zur Zeit  $t_0$  ab und hebe zu  $t_0 + x$  wieder an (einen solchen Signalverlauf nennt man Puls).

# RS-Flipflop

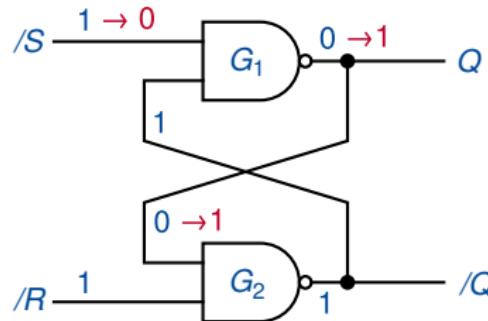
- Zustand  $Q = 0 \rightarrow$  Zustand  $Q = 1$ :



- Senke  $/S$  zur Zeit  $t_0$  ab und hebe zu  $t_0 + x$  wieder an (einen solchen Signalverlauf nennt man Puls).

# RS-Flipflop

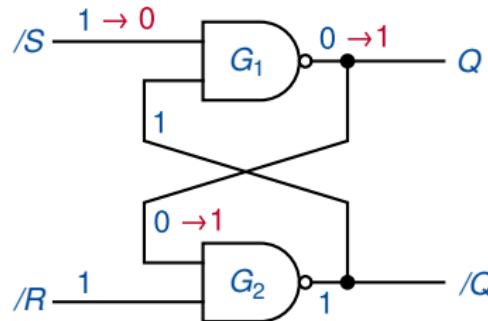
- Zustand  $Q = 0 \rightarrow$  Zustand  $Q = 1$ :



- Senke  $/S$  zur Zeit  $t_0$  ab und hebe zu  $t_0 + x$  wieder an (einen solchen Signalverlauf nennt man Puls).

# RS-Flipflop

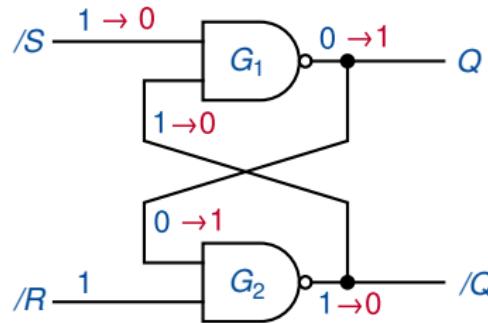
- Zustand  $Q = 0 \rightarrow$  Zustand  $Q = 1$ :



- Senke  $/S$  zur Zeit  $t_0$  ab und hebe zu  $t_0 + x$  wieder an (einen solchen Signalverlauf nennt man Puls).
- Nach Zeit  $t_{P/SQ}$  ist  $Q = 1$ .

# RS-Flipflop

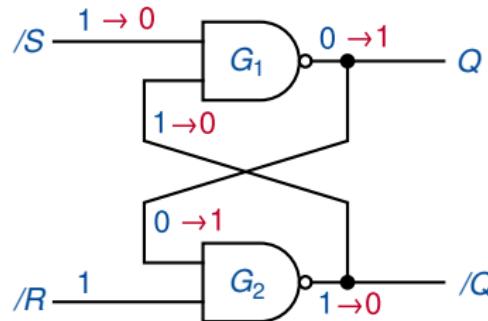
- Zustand  $Q = 0 \rightarrow$  Zustand  $Q = 1$ :



- Senke  $/S$  zur Zeit  $t_0$  ab und hebe zu  $t_0 + x$  wieder an (einen solchen Signalverlauf nennt man Puls).
- Nach Zeit  $t_{P/SQ}$  ist  $Q = 1$ .

# RS-Flipflop

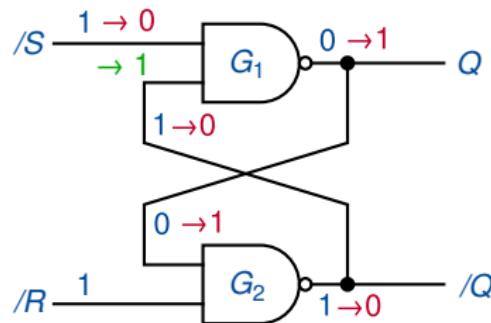
- Zustand  $Q = 0 \rightarrow$  Zustand  $Q = 1$ :



- Senke  $/S$  zur Zeit  $t_0$  ab und hebe zu  $t_0 + x$  wieder an (einen solchen Signalverlauf nennt man Puls).
- Nach Zeit  $t_{P/SQ}$  ist  $Q = 1$ .
- Nach Zeit  $t_{P/S/Q}$  ist  $/Q = 0$ .

# RS-Flipflop

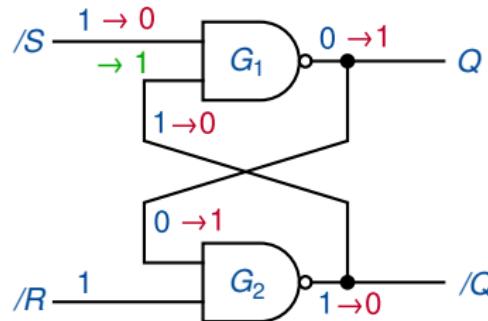
- Zustand  $Q = 0 \rightarrow$  Zustand  $Q = 1$ :



- Senke  $/S$  zur Zeit  $t_0$  ab und hebe zu  $t_0 + x$  wieder an (einen solchen Signalverlauf nennt man Puls).
- Nach Zeit  $t_{P/SQ}$  ist  $Q = 1$ .
- Nach Zeit  $t_{P/S/Q}$  ist  $/Q = 0$ .

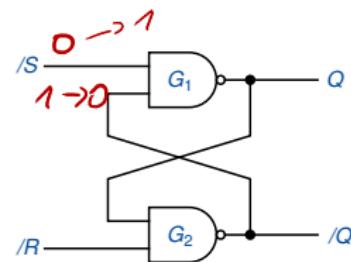
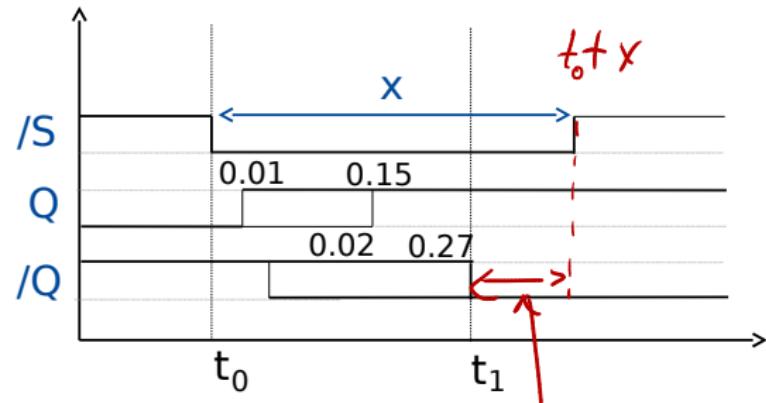
# RS-Flipflop

- Zustand  $Q = 0 \rightarrow$  Zustand  $Q = 1$ :



- Senke  $/S$  zur Zeit  $t_0$  ab und hebe zu  $t_0 + x$  wieder an (einen solchen Signalverlauf nennt man Puls).
- Nach Zeit  $t_{P/SQ}$  ist  $Q = 1$ .
- Nach Zeit  $t_{P/S/Q}$  ist  $/Q = 0$ .
- Wähle  $x$  so, dass kein Spike entsteht.

# Übergang - graphisch



	NAND	$t^{\min}$	$t^{\max}$
$G_1$	$\tau_{PLH}$	0.01	0.15
$G_2$	$\tau_{PHL}$	0.01	0.12

*muss  
spikefreies Umschalten  
von  $G_1$  gewährleisten !!!*

# Spikefreier Übergang

---

- Nach den Regeln des spikefreien Umschaltens von Gattern entsteht kein Spike, falls:

$$\cancel{(t_0 + x)} - \cancel{(t_0 + 0.27)} \geq 0.41 \Leftrightarrow \underline{x \geq 0.68ns}$$

- Wechsel von Zustand  $Q = 1$  zu Zustand  $Q = 0$  aus Symmetriegründen analog.

# Symbole und Bezeichnungen

---

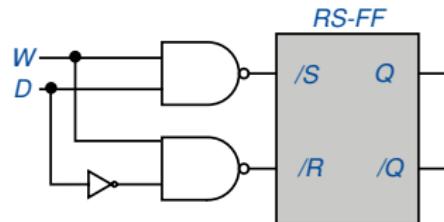
Symbol	Bezeichnung	$t^{\min}$	$t^{\max}$
$x$	Pulsweite	0.68	
$\tau_{P/SQ}$	Verzögerungszeit von /S bis Q	<u>0.01</u>	<u>0.15</u>
$\tau_{P/S/Q}$	Verzögerungszeit von /S bis /Q	<u>0.02</u>	<u>0.27</u>
$\tau_{P/RQ}$	Verzögerungszeit von /R bis Q	0.02	0.27
$\tau_{P/R/Q}$	Verzögerungszeit von /R bis /Q	0.01	0.15

# D-Latch

- $W$  ist *active high*.

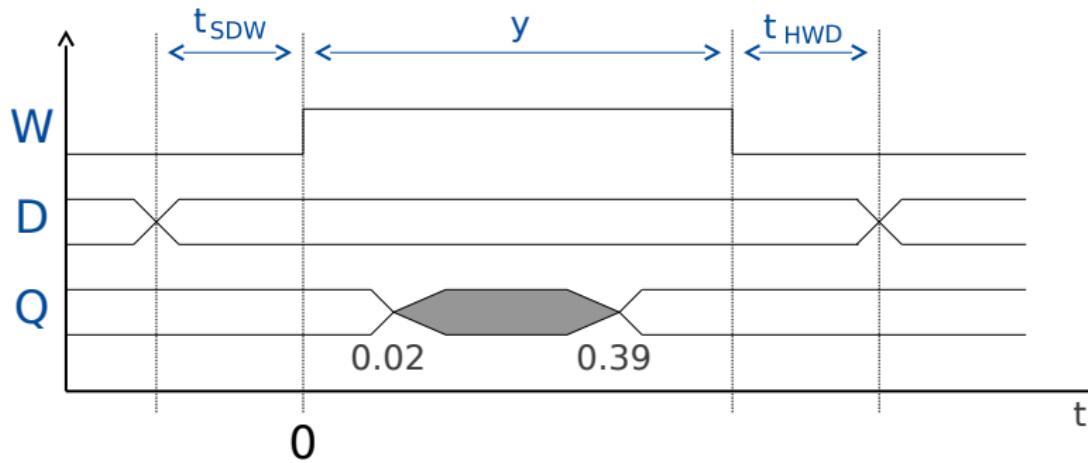
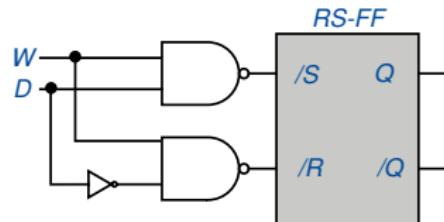
- $W = 0 \Rightarrow /S, /R$  inaktiv

- $W = 1 \Rightarrow \begin{cases} /S \text{ aktiv,} & \text{falls } D = 1 \\ /R \text{ aktiv,} & \text{falls } D = 0 \end{cases}$



- Wie beim RS-Flipflop (minimale Pulsweite!) muss man auch beim D-Latch bestimmte Forderungen an den zeitlichen Verlauf der Signale stellen, um Spikefreiheit zu garantieren.

# Timing-Diagramm



# Timing-Bedingungen für das D-Latch

---

- $W$  muss beim Schreiben lange genug 1 sein, um minimale Pulsweite  $x$  des RS-FFs zu garantieren.
- Vor  $W : 0 \rightarrow 1$  werden Daten für Zeit  $t_{SDW}$  stabil gehalten, um Spikes auf  $/S$ ,  $/R$  zu vermeiden (der kritischste Fall ist das Verhindern von Spikes auf  $/R$  bei Schreiben von 1).
- Nach  $W : 1 \rightarrow 0$  werden Daten für Zeit  $t_{HWD}$  stabil gehalten, um Spikes auf  $/S$ ,  $/R$  zu vermeiden (der kritischste Fall ist das Verhindern von Spikes auf  $/S$  beim Schreiben von 0).

# Man rechnet nach:

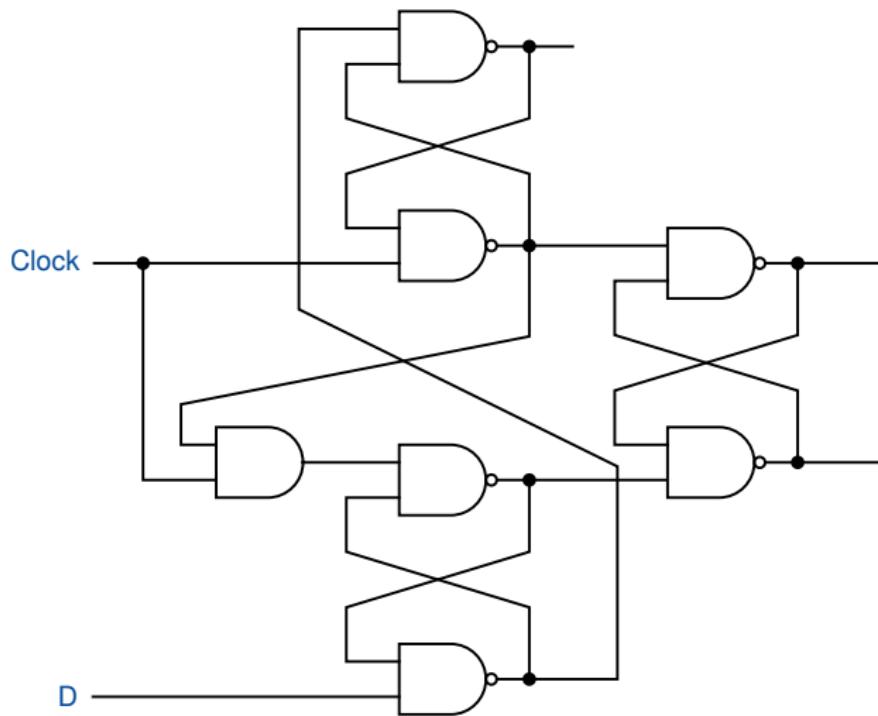
---

Der Schreibvorgang beim D-Latch funktioniert mit den Parameterwerten aus der Tabelle ([siehe Übung](#)).

Symbol	Bezeichnung	$t^{\min}$	$t^{\max}$
$y$	Pulsweite des Schreibimpulses	0.79	
$t_{SDW}$	Setupzeit von $D$ bis $W$	0.49	
$t_{HDW}$	Holdzeit von $W$ nach $D$	0.41	
$\tau_{PWQ}$	Verzögerungszeit von $W$ bis $Q$	0.02	0.39
$(\tau_{PDQ})$	Verzögerungszeit von $D$ bis $Q$	0.02	0.54)

# Mögliche Realisierung: D-Flipflop

---



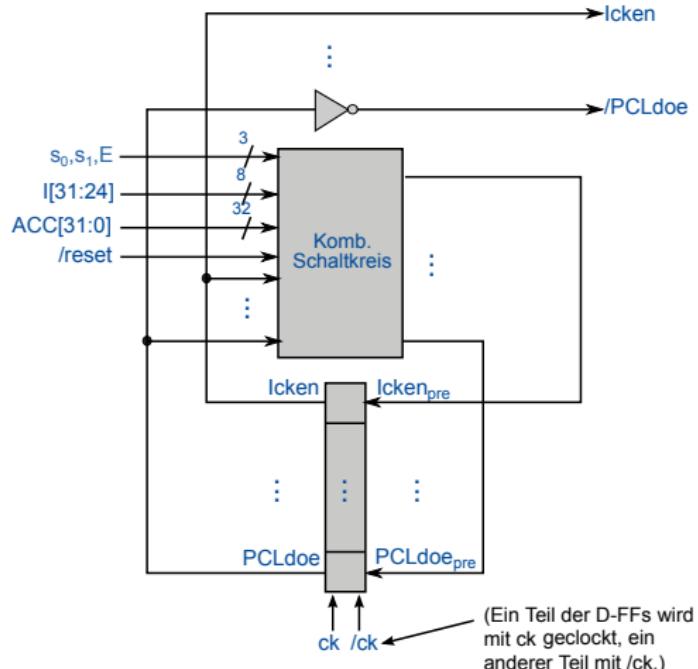
# Timing: D-Flipflop

---

- Vorgehen [analog](#) zu RS-Flipflop und D-Latch.
- Wir verzichten daher auf weitere Details.
- Die [NanGate-Bibliothek](#) enthält bereits ein D-FF mit folgenden charakteristischen Zeiten (in  $ns$ ):

Symbol	Bezeichnung	$t^{\min}$	$t^{\max}$
$t_{SDC}$	Setupzeit von $D$ bis $ck$	0.08	
$t_{HCD}$	Holdzeit von $D$ nach $ck$	0.14	
$\tau_{PCQ}$	Verzögerungszeit von $ck$ bis $Q$	0.12	0.26

# Aufbau der Kontrolllogik, zur Erinnerung



- Generierung der Kontrollsignale (OE von Treibern, ALU-Ansteuerung, ...).
- Ist ein Kontrollsignal *active low*, dann bezeichnen wir es z.B. mit  $/x$ . Das Ausgangssignal  $/x$  ergibt sich dann durch Negation des Ausgangssignals  $x$  eines entsprechenden FFs mit Eingangssignal  $x_{pre}$ .
- Ist ein Kontrollsignal *active high*, dann bezeichnen wir es z.B. mit  $x$ . Das Ausgangssignal  $x$  entspricht dem Ausgangssignal eines FFs mit Eingangssignal  $x_{pre}$ .

# Kontrolllogik

- Die Dauer eines Taktes bezeichnen wir als **Zykluszeit  $t_c$** .
- Active-High-Ausgangssignale der Kontrolllogik, bei denen das FF mit  $\text{ck}$  gesteuert ist, sind gegenüber der steigenden Flanke von  $\text{ck}$  um Zeit  $\tau_{p,ah}^+$  verzögert (resultiert aus D-FF-Verzögerung).
- Active-Low-Ausgangssignale der Kontrolllogik, bei denen das FF mit  $\text{ck}$  gesteuert ist, sind gegenüber der steigenden Flanke von  $\text{ck}$  um Zeit  $\tau_{p,al}^+$  verzögert (resultiert aus D-FF-Verzögerung + Inverterverzögerung).
- Active-High-Ausgangssignale der Kontrolllogik, bei denen das FF mit  $/\text{ck}$  gesteuert ist, sind gegenüber der letzten steigenden Flanke von  $\text{ck}$  um Zeit  $\tau_{p,ah}^- = \tau_{p,ah}^+ + t_c/2 + \tau_{PLH,Inv}$  verzögert.
- Active-Low-Ausgangssignale der Kontrolllogik, bei denen das FF mit  $/\text{ck}$  gesteuert ist, sind gegenüber der letzten steigenden Flanke von  $\text{ck}$  um Zeit  $\tau_{p,al}^- = \tau_{p,al}^+ + t_c/2 + \tau_{PLH,Inv}$  verzögert.

	INV	$t^{\min}$	$t^{\max}$
$\tau_{PLH}$	0.01	0.15	
$\tau_{PHL}$	0.00	0.08	

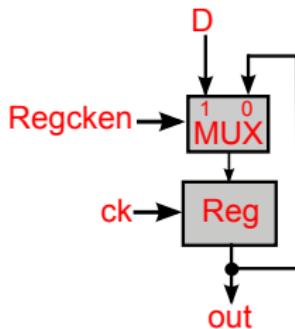
# Timing: Kontrolllogik

- Mit geeigneter Implementierung des kombinatorischen Teiles erhält man folgende charakteristische Zeiten.

Symbol	Bezeichnung	$t^{\min}$	$t^{\max}$
$\tau_{p,ah}^+$	Verzögerungszeit $ck$ bis $Q$ , active high	0.12	0.26
$\tau_{p,al}^+$	Verzögerungszeit $ck$ bis $Q$ , active low	0.12	0.41
$\tau_{p,ah}^-$	Verzögerungszeit $ck$ bis $Q$ (von $/ck$ angesteuert, active high)	$t_c/2 + 0.13$	$t_c/2 + 0.41$
$\tau_{p,al}^-$	Verzögerungszeit $ck$ bis $Q$ (von $/ck$ angesteuert, active high)	$t_c/2 + 0.13$	$t_c/2 + 0.56$
$t_{SDC}^+$	Setupzeit von $D$ bis $ck$	0.88	
$t_{SDC}^-$	Setupzeit von $D$ bis $/ck$	0.88	
$t_{HCD}^+$	Holdzeit von $D$ nach $ck$	0.06	
$t_{HCD}^-$	Holdzeit von $D$ nach $/ck$	0.06	

# Register mit Clock-Enable

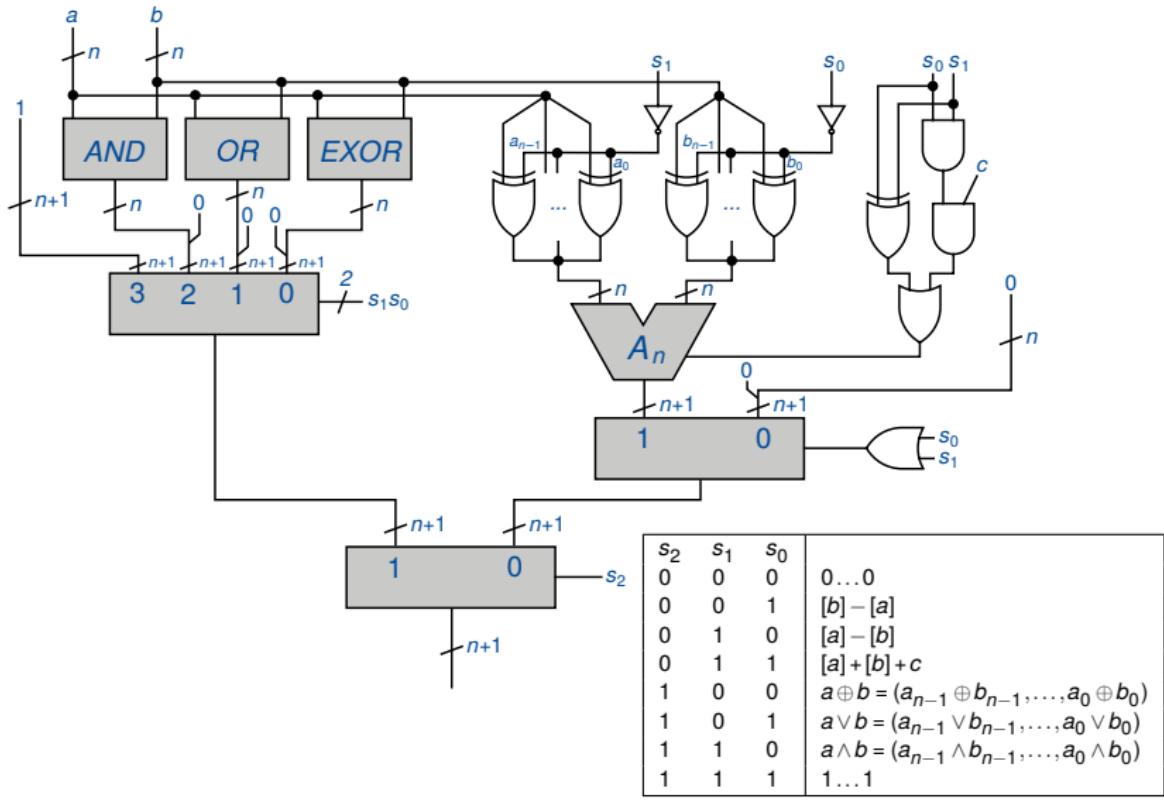
- Bei der Implementierung benötigen wir noch einen Treiberbaum der Tiefe 2, um *Regcken* auf 32 1-Bit-Multiplexer zu verteilen.



Symbol	Bezeichnung	$t^{\min}$
$t_{SDC}$	Setup-Zeit von <i>D</i> vor <i>ck</i>	0.23
$t_{HDC}$	Hold-Zeit von <i>D</i> nach <i>ck</i>	0.11
$t_{SEC}$	Setup-Zeit von <i>Regcken</i> vor <i>ck</i>	0.46
$t_{HEC}$	Hold-Zeit von <i>Regcken</i> nach <i>ck</i>	0.08

- $t_{SDC}$  ergibt sich aus Setupzeit D-FF + maximale Verzögerungszeit Multiplexer (Daten bis Ausgang) ( $0.08 + 0.15$ ).
- $t_{HDC}$  ergibt sich aus Holdzeit D-FF - minimale Verzögerungszeit Multiplexer (Daten bis Ausgang) ( $0.14 - 0.03$ ).
- $t_{SEC}$  ergibt sich aus Setupzeit D-FF + maximale Verzögerungszeit Multiplexer (Select bis Ausgang) + 2 x maximale Verzögerungszeit Treiber ( $0.08 + 0.16 + 2 \times 0.11$ ).
- $t_{HEC}$  ergibt sich aus Holdzeit D-FF - minimale Verzögerungszeit Multiplexer (Select bis Ausgang) - 2 x minimale Verzögerungszeit Treiber ( $0.14 - 0.02 - 2 \times 0.02$ ).

# Schaltrealisierung der ALU

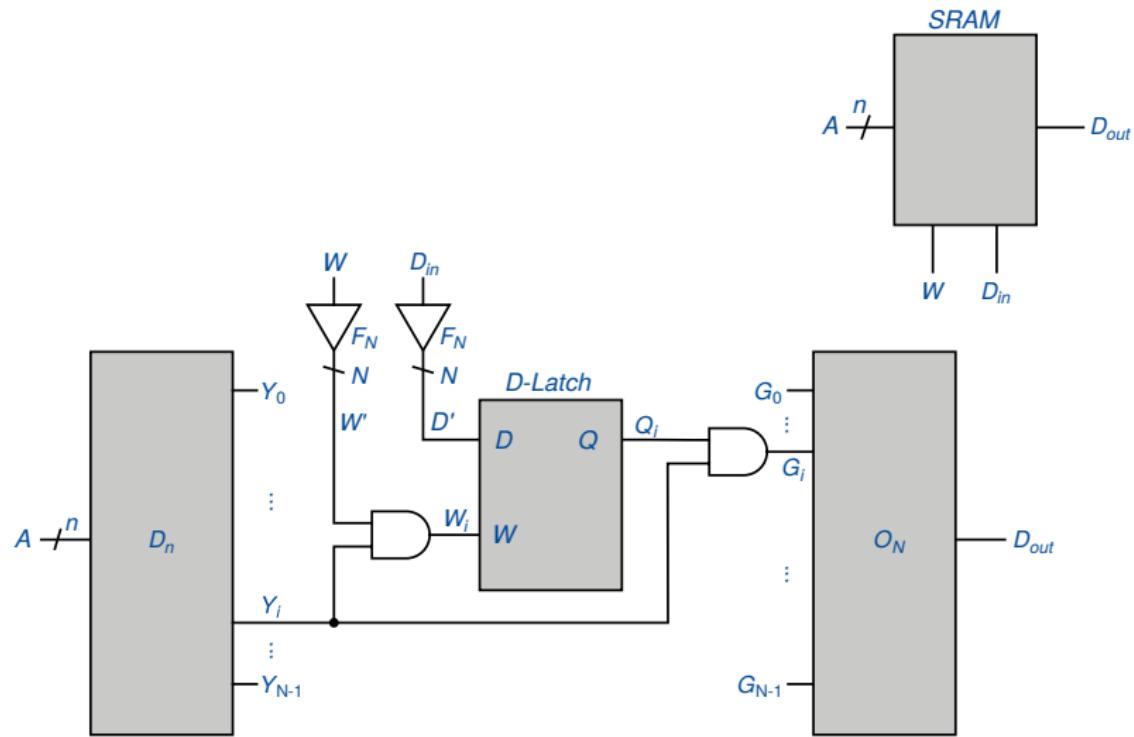


# Timing: ALU

- Annahme: ALU mit 32-Bit-Addierer (Conditional Sum).
- Man zeigt:
  - Längster Pfad über ALU läuft durch den Addierer.
  - Annahme:
    - Die Funktion-Select-Bits sind mindestens  $t_{select} = 0.28$  ns vor den Operanden gültig.
    - Dann ist garantiert, dass der kritische Pfad nicht durch die select-Eingänge bestimmt wird.
  - Zeitverhalten der ALU:

Symbol	Bezeichnung	$t^{\min}$	$t^{\max}$
$t_{select}$		0.28	
$t_{ALU}$	Verzögerungszeit von $a$ , $b$ bzw. $c_{in}$ bis Ausgang		3.25

# SRAM

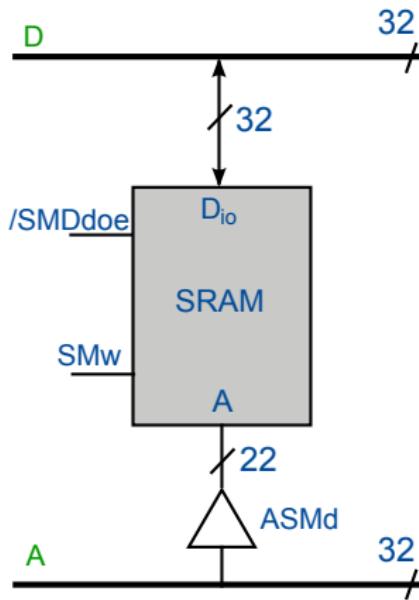


# Timing: SRAM

---

- Auch hier wäre das Vorgehen **analog** zu den bereits vorgestellten Analysen möglich. Zuvor muss man sich noch Gedanken machen um das Timing von:
  - Dekodierer
  - Treiberbäume
  - OR-Baum
- Eine detaillierte Timinganalyse ist **aufwändig**.
- Für die folgenden Timinganalysen orientieren wir uns an dem **kommerziell angebotenen** SRAM CY7C1079DV33 der Firma **Cypress Semiconductor** (siehe folgende Folien).

# Interface zu CY7C1079DV33



# Timing: CY7C1079DV33

---

- Aus dem Datenblatt entnimmt man:

Symbol	Bezeichnung	$t^{\min}$	$t^{\max}$
$t_{acc}$	Lesezugriffszeit		12.0
$t_{OED}$	Zeit von /SMD $d_{oe} = 0$ bis $D$		7.0
$t_{OEZ}$	Zeit von /SMD $d_{oe} = 1$ bis high-Z		7.0
$t_{wc}$	Schreibzykluszeit	12.0	
$t_{SAW}$	Setupzeit von $A$ bis $W$	0.0	
$t_{SAEW}$	Setupzeit von $A$ bis Ende $W$	9.0	
$t_{HWA}$	Holdzeit von $A$ nach $W$	0.0	
$w$	Schreibpulsweite	9.0	
$t_{SDEW}$	Setupzeit von $D$ bis Ende $W$	7.0	
$t_{HWD}$	Holdzeit von $D$ nach $W$	0.0	

# Kapitel 5

Timing:

1. Physikalische Eigenschaften
2. Timing wichtiger Komponenten
- 3. Exaktes Timing von ReTI**

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

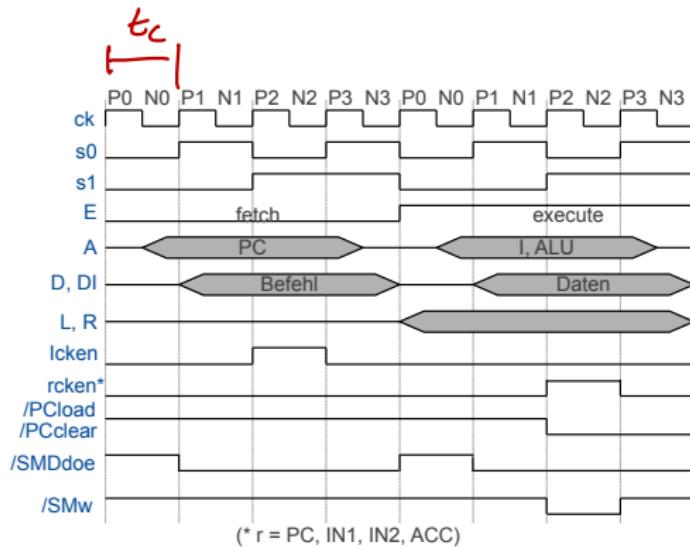
Professur für Rechnerarchitektur  
WS 2016/17

Es gilt:

- Bei hinreichend langsamem Takt funktioniert der Rechner.

## Frage:

- Wie schnell kann man den Rechner takten?  
Wie lange muss ein Takt mind. sein?
- Ersetze idealisiertes Timing durch exakte Timinganalyse
- Gesucht:  
Untere Grenze für Zykluszeit  $t_c$



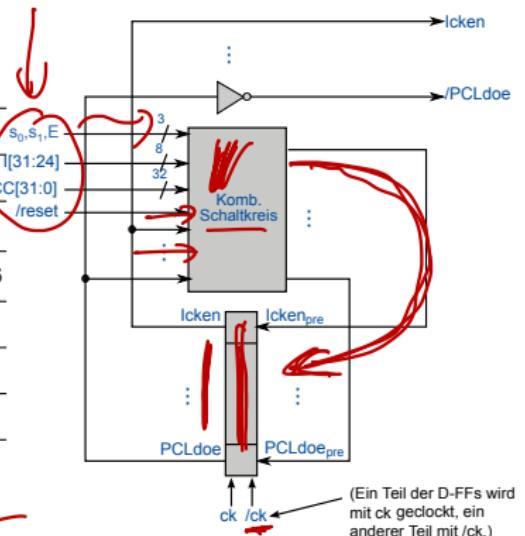
# Schritte der Analyse

---

- 1 Einhalten von Setup- und Hold-Zeiten der Kontrolllogik
  - 2 Vermeidung von Bus-Contention
  - 3 PC-Inkrementierung
  - 4 Compute-Befehle:  
OE: Compute Memory
  - 5 *Fetch, Load, Store, Jump*
- 
- Wir werden uns auf Compute-Befehle beschränken.

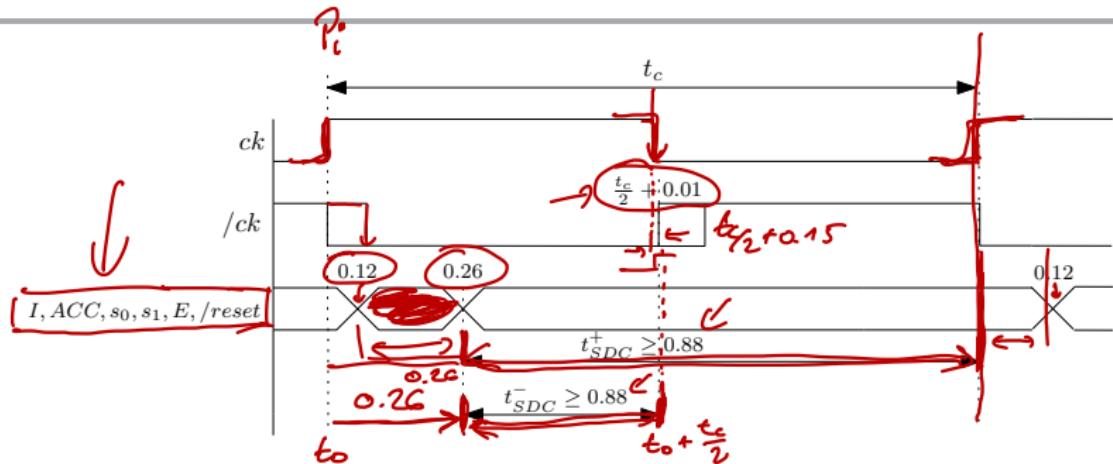
# Timing der Kontrolllogik (1/3)

Symbol	Bezeichnung	$t_{\min}$	$t_{\max}$
$\tau_{p,ah}^+$	Verzögerungszeit $ck$ bis $Q$ , active high	0.12	0.26
$\tau_{p,al}^+$	Verzögerungszeit $ck$ bis $Q$ , active low	0.12	0.41
$\tau_{p,ah}^-$	Verzögerungszeit $ck$ bis $Q$ (von $/ck$ angesteuert, active high)	$t_c/2 + 0.13$	$t_c/2 + 0.41$
$\tau_{p,al}^-$	Verzögerungszeit $ck$ bis $Q$ (von $/ck$ angesteuert, active high)	$t_c/2 + 0.13$	$t_c/2 + 0.56$
$t_{SDC}^+$	Setupzeit von $D$ bis $ck$	0.88	
$t_{SDC}^-$	Setupzeit von $D$ bis $/ck$	0.88	
$t_{HCD}^+$	Holdzeit von $D$ nach $ck$	0.06	
$t_{HCD}^-$	Holdzeit von $D$ nach $/ck$	0.06	



- Setupzeit der Dateneingänge bis  $ck$  ist  $t_{SDC}^+ \geq 0.88$ , Setupzeit der Dateneingänge bis  $/ck$  ist  $t_{SDC}^- \geq 0.88$ .
- Dateneingänge ( $s_0, s_1, E, I, ACC, \text{reset}$ ) müssen rechtzeitig bereit sein.
- Alle Dateneingänge sind Ausgangssignale von FFs, die mit  $ck$  getaktet werden.

## Timing der Kontrolllogik (2/3)



- Wähle eine beliebige steigende Taktflanke  $P_i$  als zeitlichen Bezugspunkt.
- Die Dateneingänge sind also bereit zur Zeit  $\tau_{PPCQ} = [0.12, 0.26]$  (Verzögerung eines D-FF).
- Die nächste steigende Taktflanke von  $/ck$  ist bei  $\frac{t_c}{2} + [0.01, 0.15]$ , die nächste steigende Taktflanke von  $ck$  bei  $t_c$ .

*INVERTER*

$$\Rightarrow \frac{t_c}{2} + 0.01 \geq 0.88 + t_{SDC}^- \quad \text{Inverterverzögerung}$$

$$\Rightarrow t_c \geq 0.88 + 0.26 = 1.14 \quad \tau_{PPCQ} \text{ für FFs}$$

$$\Rightarrow t_{SDC}^+ + \tau_{PPCQ} \text{ für FFs} = 1.14$$

## Timing der Kontrolllogik (3/3)

---

- Hold-Zeiten sind unkritisch:

- FFs, die mit  $ck$  getaktet sind:

$t_{HDC}^+ \geq 0.06$  und Eingangsdaten werden mindestens noch  
0.12 ns nach steigender Flanke von  $ck$  gehalten  
(Verzögerung D-FF).

- FFs, die mit  $/ck$  getaktet sind:

$t_{HDC}^- \geq 0.06$  und Eingangsdaten werden sowieso noch  
einen halben Takt nach steigender Flanke von  $/ck$  gehalten  
(+ D-FF-Verzögerung).

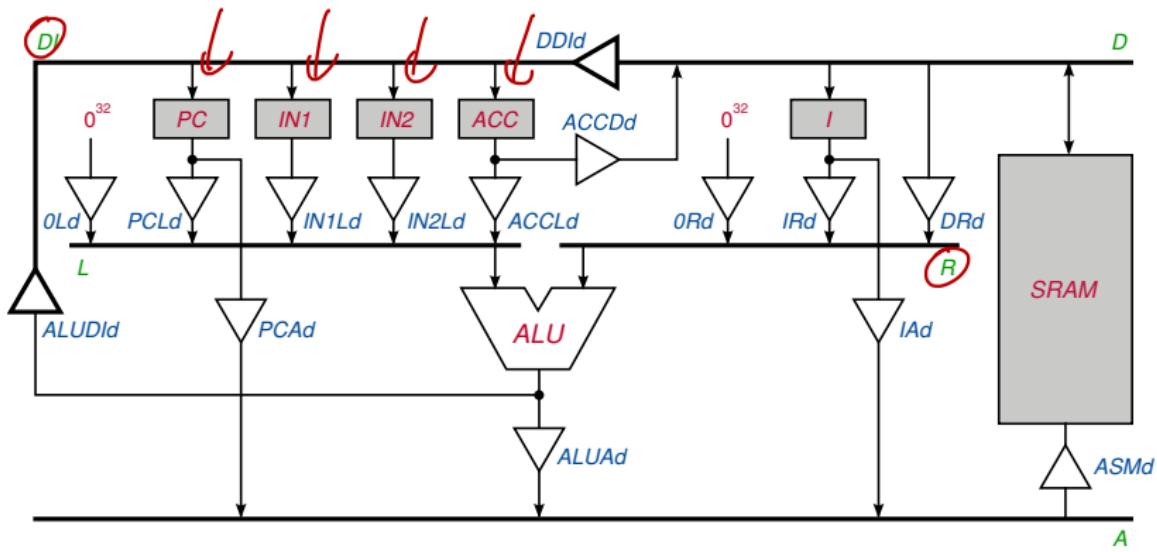
# Constraints

---

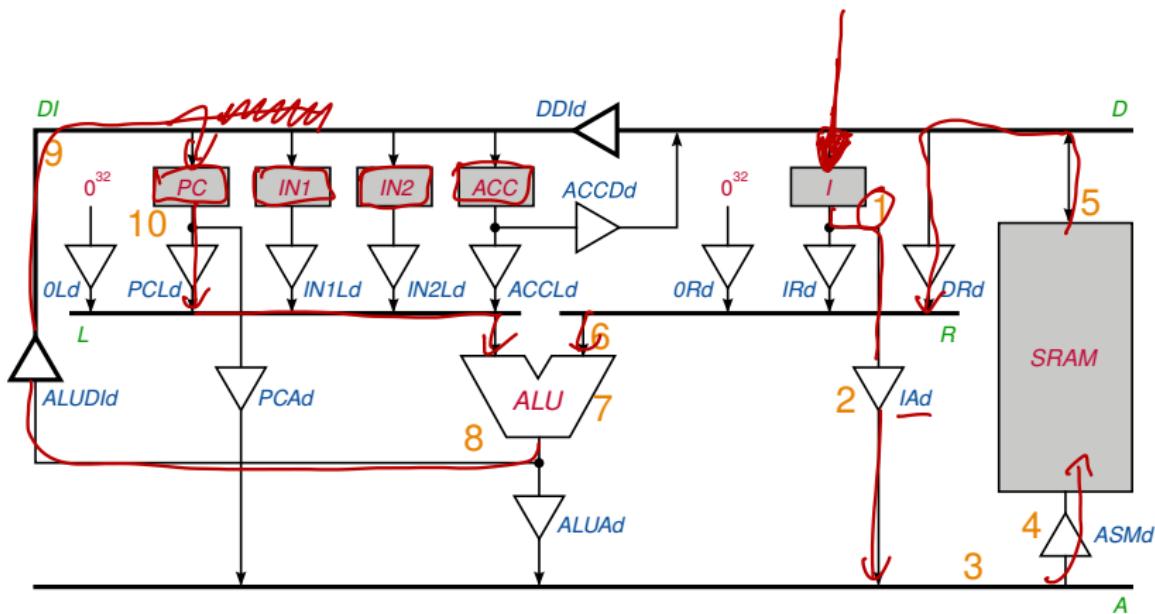
- $t_c \geq 2.26$

# Compute-Befehle

■ Am zeitkritischsten ist Compute memory!

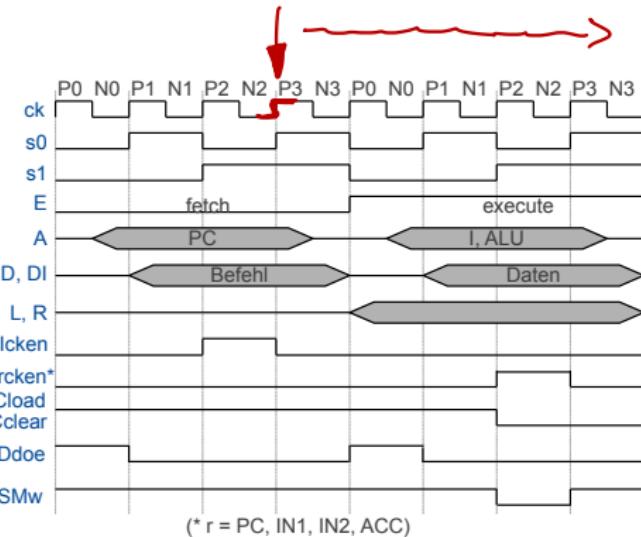


$[r] := [r] + [M(\langle i \rangle)]$ ; hier:  $\underline{[PC]} := \underline{[PC]} + \underline{[M(\langle i \rangle)]}$



# Analyse allgemein

- Beginn der Analyse bei *P3* von Fetch als zeitlicher Bezugspunkt.
- Bei *P3* von Fetch wird der Befehl ins Instruktionsregister übernommen.



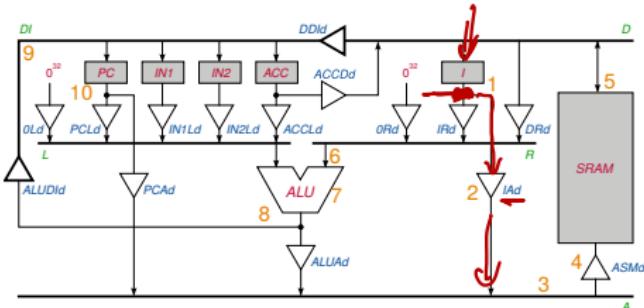
# I-Ausgänge (1/2)

I-Ausgänge gültig bei

$$\tau_1 = \underbrace{[0.12, 0.26]}_{\substack{\text{t}_{PCQ} \text{ von Register } I \\ \rightarrow D\text{-FF Verzögerung}}}$$

D-FF	Bezeichnung	$t^{\min}$	$t^{\max}$
$t_{SDC}$	Setupzeit von $D$ bis $ck$	0.08	
$t_{HCD}$	Holdzeit von $D$ nach $ck$	0.14	
$\tau_{PCQ}$	Verzögerungszeit von $ck$ bis $Q$	0.12	0.26
$\tau_{PDQ}$	Verzögerungszeit von $D$ bis $Q$	0.10	0.21

- $0^8 / 23 \dots I_0$  wird über Treiber  $IAd$  auf Adressbus gegeben.
- $0^8$  ist eine Konstante und steht daher ebenfalls zu  $\tau_1$  bereit.



# Constraints

---

L

- $\tau_1 = [0.12, 0.26]$

- $t_c \geq 2.26$

# I-Ausgänge (2/2)

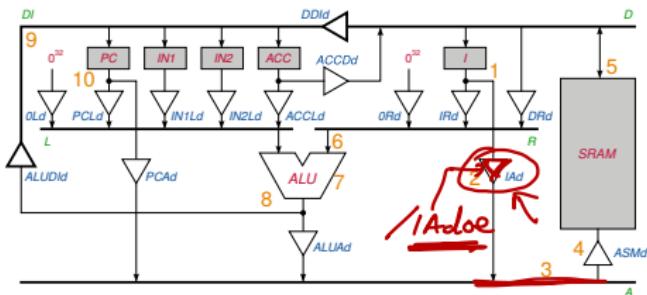
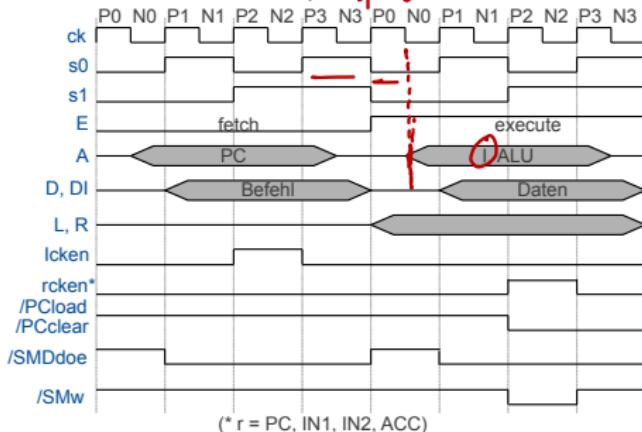
IAd enabled bei N0 von Execute, d.h. /IAdoe aktiv zur Zeit

$$\begin{aligned}\tau_2 &= t_c + \cancel{t_{p.al}} \\ &= t_c + \frac{t_c}{2} + [0.13, 0.56] \\ &= \frac{3}{2}t_c + [0.13, 0.56]\end{aligned}$$

I schon gültig vor Aktivierung von IAd bei Punkt 2, falls

$$\begin{aligned}\max(\tau_1) &\leq \min(\tau_2) \Leftrightarrow \\ 0.26 &\leq \frac{3}{2}t_c + 0.13 \Leftrightarrow \\ \frac{3}{2}t_c &\geq 0.13 \Leftrightarrow \\ t_c &\geq 0.09\end{aligned}$$

Referenz  
↓  
 $t_c$   
↓



# Constraints

---

- $\tau_1 = [0.12, 0.26]$
- $\tau_2 = t_c + \tau_{p,al}^- = \frac{3}{2}t_c + [0.13, 0.56]$

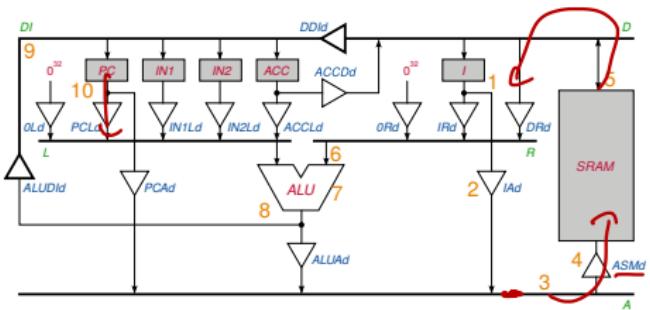
- $t_c \geq 2.26$  ↗
- $t_c \geq 0.09$

# Gültiges A (1/2)

→ A gültig zur Zeit

$$\begin{aligned}\tau_3 &= \tau_2 + [0.03, 0.11] \\ &\quad \text{Enable Zeit Treiber} \\ &= \frac{3}{2} t_c + [0.16, 0.67]\end{aligned}$$

	Tristate-Treiber	min	max
$\tau_{PZL}$	Enable-Zeiten	0.03	0.10
$\tau_{PZH}$	Enable-Zeiten	0.03	0.11
$\tau_{PLZ}$	Disable-Zeiten	0.03	0.11
$\tau_{PHZ}$	Disable-Zeiten	0.03	0.10
$\tau_{PLH}$	Umschaltverzögerung bei $/OE = 0$	0.02	0.07
$\tau_{PHL}$	Umschaltverzögerung bei $/OE = 0$	0.03	0.10



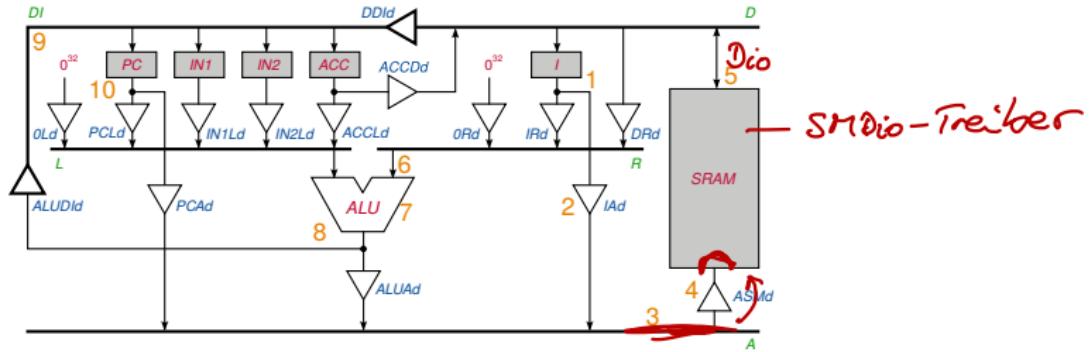
# Constraints

---

- $\tau_1 = [0.12, 0.26]$
- $\tau_2 = t_c + \tau_{p,al}^- = \frac{3}{2}t_c + [0.13, 0.56]$
- $\tau_3 = \tau_2 + [0.03, 0.11] = \frac{3}{2}t_c + [0.16, 0.67]$

- $t_c \geq 2.26$
- $t_c \geq 0.09$

# Gültiges A (2/2)



## ■ ASMd immer enabled

- nur Treiber-Verzögerung berücksichtigt
- A an SM bei

$$\tau_4 = \tau_3 + [0.02, 0.10] = \frac{3}{2}t_c + [0.18, 0.77]$$

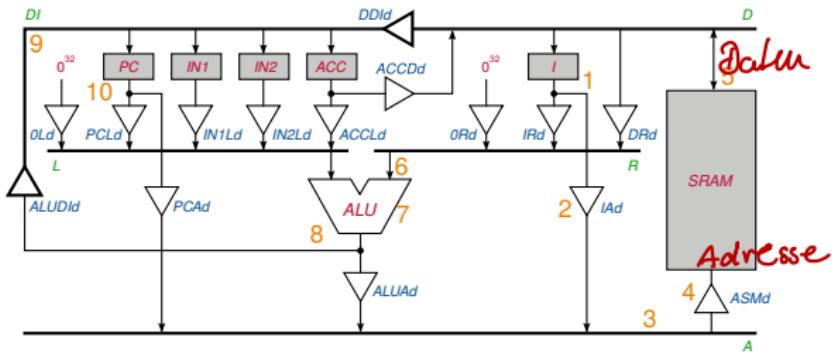
# Constraints

---

- $\tau_1 = [0.12, 0.26]$
- $\tau_2 = t_c + \tau_{p,al}^- = \frac{3}{2}t_c + [0.13, 0.56]$
- $\tau_3 = \tau_2 + [0.03, 0.11] = \frac{3}{2}t_c + [0.16, 0.67]$
- $\tau_4 = \tau_3 + [0.02, 0.10] = \underline{\frac{3}{2}t_c + [0.18, 0.77]}$

- $t_c \geq 2.26$
- $t_c \geq 0.09$

# Daten am Speicherausgang (1/3)

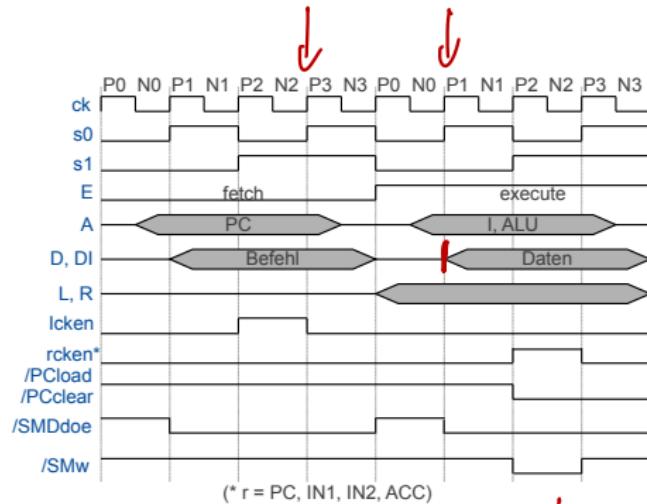


- Lesezugriffszeit von **SRAM**: [0.0, 12.0] (siehe Daten von CY7C1079DV33)  
→ Gültige Daten am Speicherausgang bei

$$\begin{aligned}\tau_5 &= \underline{\tau_4 + [0.0, 12.0]} \\ &= \frac{3}{2} t_c + [0.18, 0.77] + [0.0, 12.0] \\ &= \underline{\frac{3}{2} t_c + [0.18, 12.77]} \quad \leftarrow\end{aligned}$$

- Das ist aber nur korrekt, wenn der Ausgangstreiber durch /SMDoe rechtzeitig enabled ist!

# Daten am Speicherausgang (2/3)



SRAM CY7C1079DV33			
Symbol	Bezeichnung	$t^{\min}$	$t^{\max}$
$t_{acc}$	Lesezugriffszeit		12.0
$t_{OED}$	Zeit von <u>/SMDdoe = 0 bis D</u>	<u>7.0</u>	
...			

- /SMDdoe aktiviert zur Zeit  $\tau' = 2 \cdot t_c + \tau_{p.al}^+ = 2 \cdot t_c + [0.12, 0.41]$ .
- Daten am Speicherausgang aufgrund Treiber-Enable gültig zur Zeit  $\tau'' = \tau' + [0.0, 7.0] = 2 \cdot t_c + [0.12, 7.41]$ .
- ⇒ Daten am Speicherausgang gültig spätestens zur Zeit  $\tau''' = \max(\max(\tau_5), \max(\tau''))$ .

## Daten am Speicherausgang (3/3)

- Daten am Speicherausgang gültig zur Zeit  $t''' = \max(\max(\tau_5), \max(\tau''))$ .
- Bedingung für  $\underline{\max(\tau_5) \geq \max(\tau'')}$ :

$$\begin{aligned}\frac{3}{2}t_c + 12.77 &\geq 2 \cdot t_c + 7.41 \Leftrightarrow \\ \frac{1}{2}t_c &\leq 5.36 \Leftrightarrow \\ t_c &\leq 10.72\end{aligned}$$

- Wir nehmen ab jetzt an, dass die Taktperiode  $t_c \leq 10.72$  ist und rechnen mit  $\underline{\max(\tau_5)}$  weiter.
- (Es gilt auf jeden Fall  $\underline{\min(\tau_5)} (= \frac{3}{2}t_c + 0.18) < \min(\tau'') (= 2 \cdot t_c + 0.12)$ )
- Sollte sich später ergeben, dass die minimale Taktperiode  $t_c > 10.72$ , dann müssten wir die Rechnung nochmals korrigieren.

# Constraints

- $\tau_1 = [0.12, 0.26]$
- $\tau_2 = t_c + \tau_{p,al}^- = \frac{3}{2}t_c + [0.13, 0.56]$
- $\tau_3 = \tau_2 + [0.03, 0.11] = \frac{3}{2}t_c + [0.16, 0.67]$
- $\tau_4 = \tau_3 + [0.02, 0.10] = \frac{3}{2}t_c + [0.18, 0.77]$
- $\tau_5 = \tau_4 + [0.0, 12.0] = \frac{3}{2}t_c + [0.18, 12.77]$

Spätestens  
wir lieber  
SRAM-Daten  
auf

- $t_c \geq 2.26$
- $t_c \geq 0.09$
- $t_c \leq 10.72$

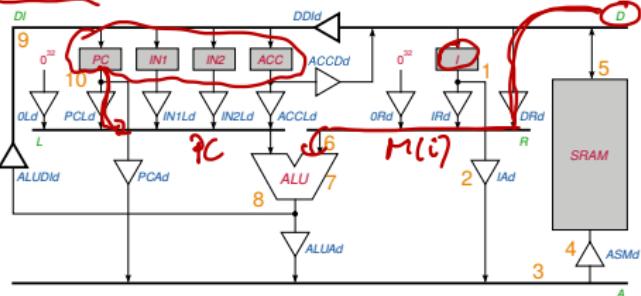
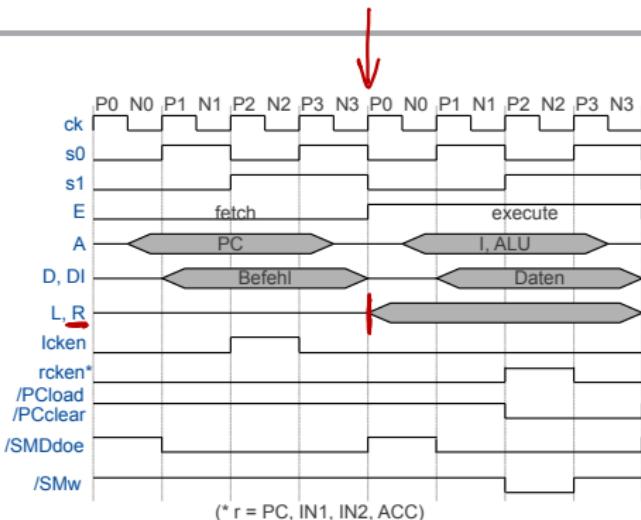
# Daten auf R

$DRd$  enabled bei  $P0$  von Execute, also einen Takt vor Ausgangstreiber von SM  
 → Enable nicht kritisch  
 → Daten auf R spätestens bei

$$\tau_6 = \tau_5 + [0.02, 0.10] \text{ (Treiber-Verzögerung)}$$

$$= \frac{3}{2} t_c + [0.20, 12.87]$$

$$PC = PC \times M(i)$$



# Constraints

---

- $\tau_1 = [0.12, 0.26]$
- $\tau_2 = t_c + \tau_{p,al}^- = \frac{3}{2}t_c + [0.13, 0.56]$
- $\tau_3 = \tau_2 + [0.03, 0.11] = \frac{3}{2}t_c + [0.16, 0.67]$
- $\tau_4 = \tau_3 + [0.02, 0.10] = \frac{3}{2}t_c + [0.18, 0.77]$
- $\tau_5 = \tau_4 + [0.0, 12.0] = \frac{3}{2}t_c + [0.18, 12.77]$
- $\tau_6 = \tau_5 + [0.2, 0.10] = \frac{3}{2}t_c + [0.20, 12.87]$
- $t_c \geq 2.26$
- $t_c \geq 0.09$
- $t_c \leq 10.72$

# Daten auf $L$

---

- Registerausgänge  $r \in \{PC, ACC, IN1, IN2\}$  schon seit letzter Execute-Phase gültig.
  - nicht kritisch
- Treiber  $rLd$  enabled bei  $P0$  von Execute, d.h. wie auch bei  $DRd$  ist Zeit zum Enablen unkritisch im Vergleich zu  $t_6$ .

- $f[2 : 0], c_{in}$  werden durch den kombinatorischen Schaltkreis der Kontrolllogik aus  $I_{31}, \dots, I_{24}$  berechnet.
  - $I$ -Ausgänge aber schon gültig bei  $\tau_1 = [0.12, 0.26]$ .  
 $\text{D-FF}$
  - Verzögerungszeit des kombinatorischen Schaltkreises  
 $< t_{SDC}^+ = 0.88$
  - $f[2 : 0], c_{in}$  gültig spätestens bei  $t_7 = 0.26 + 0.88 = 1.14$ .
- völlig unkritisch verglichen mit  $\max_i(\tau_6) = \frac{3}{2}t_c + 12.87$

# Constraints

---

- $\tau_1 = [0.12, 0.26]$
- $\tau_2 = t_c + \tau_{p,al}^- = \frac{3}{2}t_c + [0.13, 0.56]$
- $\tau_3 = \tau_2 + [0.03, 0.11] = \frac{3}{2}t_c + [0.16, 0.67]$
- $\tau_4 = \tau_3 + [0.02, 0.10] = \frac{3}{2}t_c + [0.18, 0.77]$
- $\tau_5 = \tau_4 + [0.0, 12.0] = \frac{3}{2}t_c + [0.18, 12.77]$
- $\tau_6 = \tau_5 + [0.2, 0.10] = \frac{3}{2}t_c + [0.20, 12.87]$
- $t_7 = 1.14$
- $t_c \geq 2.26$
- $t_c \geq 0.09$
- $t_c \leq 10.72$

# Voraussetzungen für die exakte Timinganalyse von *Compute memory*

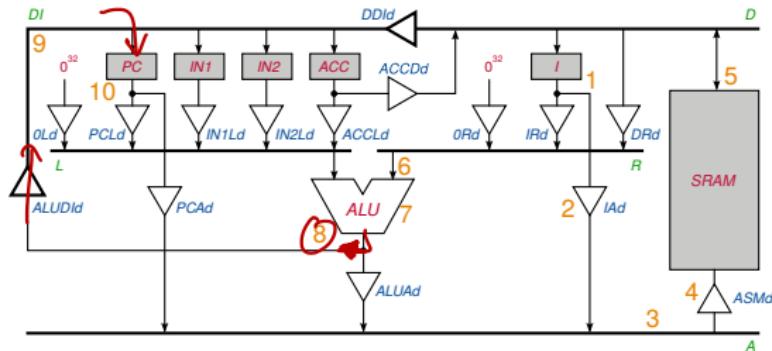
---

## ■ ALU

- Analyse der ALU (32-Bit mit Conditional Sum) unter folgender Annahme:
  - Funktionsselect-Signale liegen 0.28 ns vor den Daten an (unkritisch, da  $t_7 + 0.28 = 1.14 + 0.28 = 1.42 < \min(\tau_6) = \frac{3}{2}t_c + 0.20$ ).  $t_c \geq 2.26$
  - Resultatsausgänge gültig 3.25 ns nachdem die Daten anliegen.

Symbol	Bezeichnung	$t^{\min}$	$t^{\max}$
$t_{\text{select}}$		0.28	
$t_{\text{ALU}}$	Verzögerungszeit von $a$ , $b$ bzw. $c_{in}$ bis Ausgang		3.25

# ALU-Ausgänge



■ Spätestens gültig bei

$$t_8 = \max(\tau_6) + \underbrace{3.25}_{\text{Delay ALU}} = \frac{3}{2} t_c + 12.87 + 3.25$$
$$= \frac{3}{2} t_c + 16.12$$

# Constraints

---

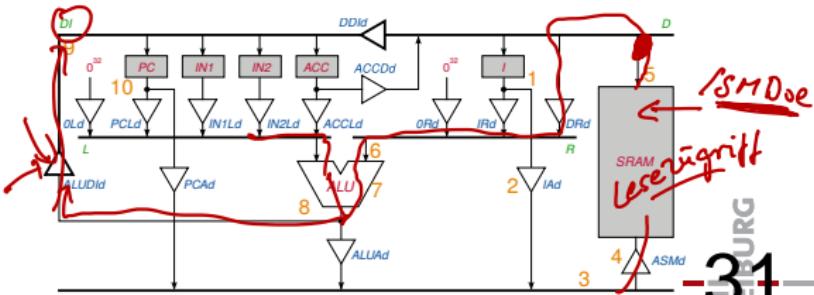
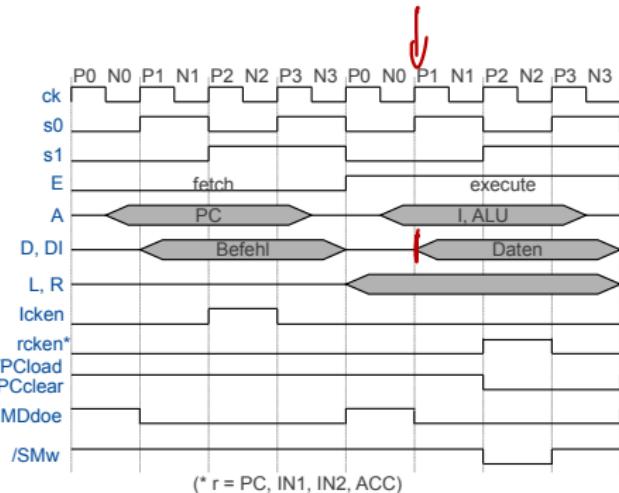
- $\tau_1 = [0.12, 0.26]$
- $\tau_2 = t_c + \tau_{p,al}^- = \frac{3}{2}t_c + [0.13, 0.56]$
- $\tau_3 = \tau_2 + [0.03, 0.11] = \frac{3}{2}t_c + [0.16, 0.67]$
- $\tau_4 = \tau_3 + [0.02, 0.10] = \frac{3}{2}t_c + [0.18, 0.77]$
- $\tau_5 = \tau_4 + [0.0, 12.0] = \frac{3}{2}t_c + [0.18, 12.77]$
- $\tau_6 = \tau_5 + [0.2, 0.10] = \frac{3}{2}t_c + [0.20, 12.87]$
- $t_7 = 1.14$
- $t_8 = \max(\tau_6) + 3.25 = \frac{3}{2}t_c + 16.12$
- $t_c \geq 2.26$
- $t_c \geq 0.09$
- $t_c \leq 10.72$

- /ALUDIdoe wird wie /SMDdoe aktiviert bei P1 von Execute
- Daten kommen an ALUDId später an als an als Daten am internen SRAM-Treiber
- Enable-Zeit von ALUDId jedoch kürzer als bei SRAM
- Mit  $t_c \leq 10.72$  ist auf jeden Fall auch für ALUDId gewährleistet, dass Treiber enabled, wenn Daten kommen.

→ Berücksichtige nur Treiberverzögerung.

→ Gültig spätestens bei

$$\begin{aligned}
 t_9 &= t_8 + 0.10 \\
 &= \frac{3}{2} t_c + 16.12 + 0.10 \\
 &= \frac{3}{2} t_c + 16.22
 \end{aligned}$$



# Constraints

---

- $\tau_1 = [0.12, 0.26]$
- $\tau_2 = t_c + \tau_{p,al}^- = \frac{3}{2}t_c + [0.13, 0.56]$
- $\tau_3 = \tau_2 + [0.03, 0.11] = \frac{3}{2}t_c + [0.16, 0.67]$
- $\tau_4 = \tau_3 + [0.02, 0.10] = \frac{3}{2}t_c + [0.18, 0.77]$
- $\tau_5 = \tau_4 + [0.0, 12.0] = \frac{3}{2}t_c + [0.18, 12.77]$
- $\tau_6 = \tau_5 + [0.2, 0.10] = \frac{3}{2}t_c + [0.20, 12.87]$
- $t_7 = 1.14$
- $t_8 = \max(\tau_6) + 3.25 = \frac{3}{2}t_c + 16.12$
- $t_9 = t_8 + 0.10 = \frac{3}{2}t_c + 16.22$
- $t_c \geq 2.26$
- $t_c \geq 0.09$
- $t_c \leq 10.72$

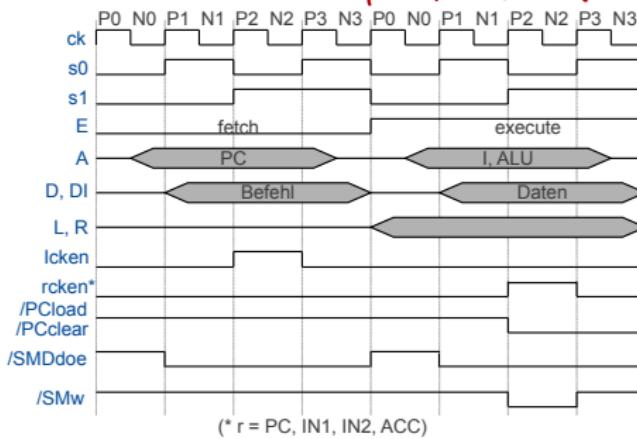
# Datenübernahme in Register $r$

## ■ Clocksignale bei P3 von Execute

→ steigende Flanke bei  $\tau_{10} = 4t_c$

## ■ Minimale Taktperiode wird aus Setup-Zeit von $r$ berechnet (Setup-Zeit am größten, wenn $r = PC$ )

$$\downarrow t_{cy} \quad t_c \quad t_c \quad t_c \quad t_c \quad \uparrow$$

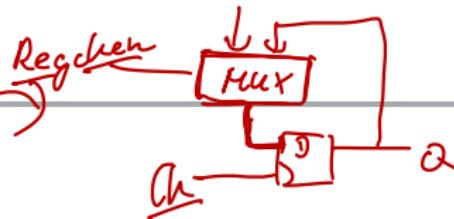


# Constraints

---

- $\tau_1 = [0.12, 0.26]$
- $\tau_2 = t_c + \tau_{p,al}^- = \frac{3}{2}t_c + [0.13, 0.56]$
- $\tau_3 = \tau_2 + [0.03, 0.11] = \frac{3}{2}t_c + [0.16, 0.67]$
- $\tau_4 = \tau_3 + [0.02, 0.10] = \frac{3}{2}t_c + [0.18, 0.77]$
- $\tau_5 = \tau_4 + [0.0, 12.0] = \frac{3}{2}t_c + [0.18, 12.77]$
- $\tau_6 = \tau_5 + [0.2, 0.10] = \frac{3}{2}t_c + [0.20, 12.87]$
- $t_7 = 1.14$
- $t_8 = \max(\tau_6) + 3.25 = \frac{3}{2}t_c + 16.12$
- $t_9 = t_8 + 0.10 = \frac{3}{2}t_c + 16.22$
- $\tau_{10} = 4 \cdot t_c$
- $t_c \geq 2.26$
- $t_c \geq 0.09$
- $t_c \leq 10.72$

# Timing: Zähler



- Aus einer Analyse des Zählers in einer Implementierung gemäß Kapitel 4.1 (aber mit zusätzlichem Clock-Enable für das Register!) ergeben sich folgende Zeiten:

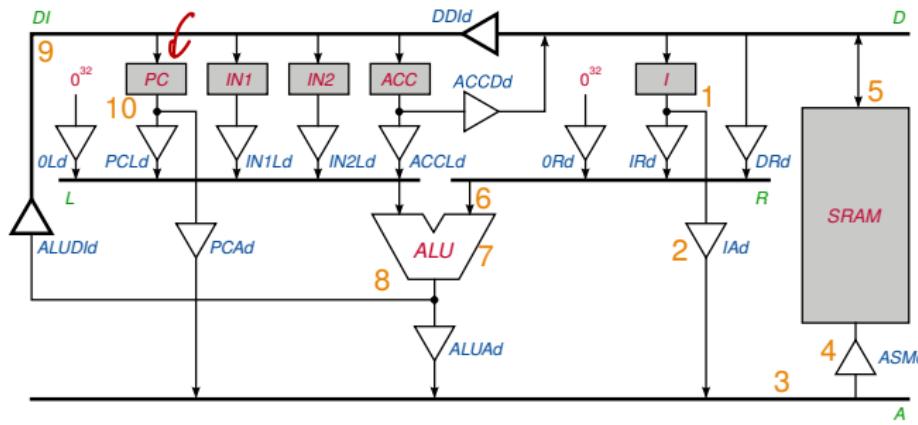
Symbol	Bezeichnung	$t^{\min}$	$t^{\max}$
$t_{SDC}$	Setup-Zeit von $D$ vor $ck$	0.53	
$t_{HDC}$	Hold-Zeit von $D$ nach $ck$	0.05	
$t_{SLC}$	Setup-Zeit von $/L$ vor $ck$	0.76	
$t_{HLC}$	Hold-Zeit von $/L$ nach $ck$	0.02	
$t_{SEC}$	Setup-Zeit von $/PCcken$ vor $ck$	0.46	
$t_{HEC}$	Hold-Zeit von $/PCcken$ nach $ck$	0.08	
...	...	...	...

# Setup-Zeit von Zähler

■ Setup-Zeit:  $t_{SDC} = 0.53 \text{ ns}$  (siehe Aufbau Zähler)

→ Bedingung:

$$\begin{aligned} & \boxed{t_9 + 0.53 \leq \min(\tau_{10})} \\ & \Leftrightarrow \frac{3}{2}t_c + 16.75 \leq \underline{4t_c} \\ & \Leftrightarrow \frac{5}{2}t_c \geq 16.75 \\ & \Leftrightarrow \underline{t_c \geq 6.70} \end{aligned}$$



# Constraints

---

- $\tau_1 = [0.12, 0.26]$
- $\tau_2 = t_c + \tau_{p,al}^- = \frac{3}{2}t_c + [0.13, 0.56]$
- $\tau_3 = \tau_2 + [0.03, 0.11] = \frac{3}{2}t_c + [0.16, 0.67]$
- $\tau_4 = \tau_3 + [0.02, 0.10] = \frac{3}{2}t_c + [0.18, 0.77]$
- $\tau_5 = \tau_4 + [0.0, 12.0] = \frac{3}{2}t_c + [0.18, 12.77]$
- $\tau_6 = \tau_5 + [0.2, 0.10] = \frac{3}{2}t_c + [0.20, 12.87]$
- $t_7 = 1.14$
- $t_8 = \max(\tau_6) + 3.25 = \frac{3}{2}t_c + 16.12$
- $t_9 = t_8 + 0.10 = \frac{3}{2}t_c + 16.22$
- $\tau_{10} = 4 \cdot t_c$

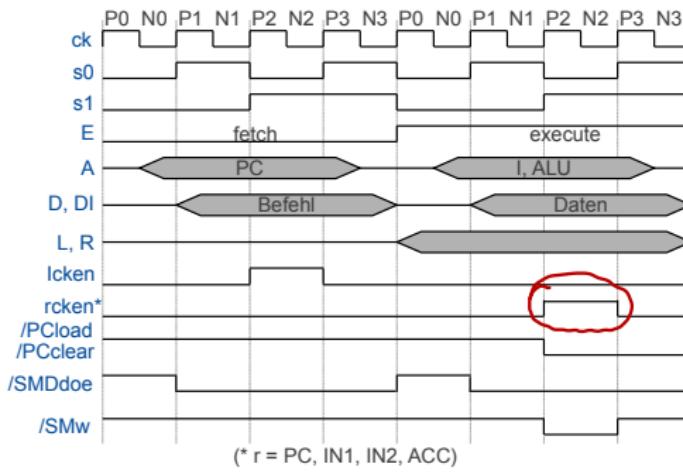
- $t_c \geq \underline{2.26}$
- $t_c \geq 0.09$
- $t_c \leq 10.72$
- $t_c \geq \underline{6.70}$

Es bleiben zu beachten:

$$\text{Holdzeit} = \frac{1}{2} - 0.03 = 0.14$$

- Hold-Zeit  $t_{HCD}$
- Maximal bei  $r \in \{ACC, IN1, IN2\}$ ,  $t_{HCD} = 0.11$

- Unproblematisch, da alle Treiber noch mindestens  $\frac{1}{2}$  Takt nach  $rck$  enabled sind.



# Setup- und Hold-Zeiten von *rcken*

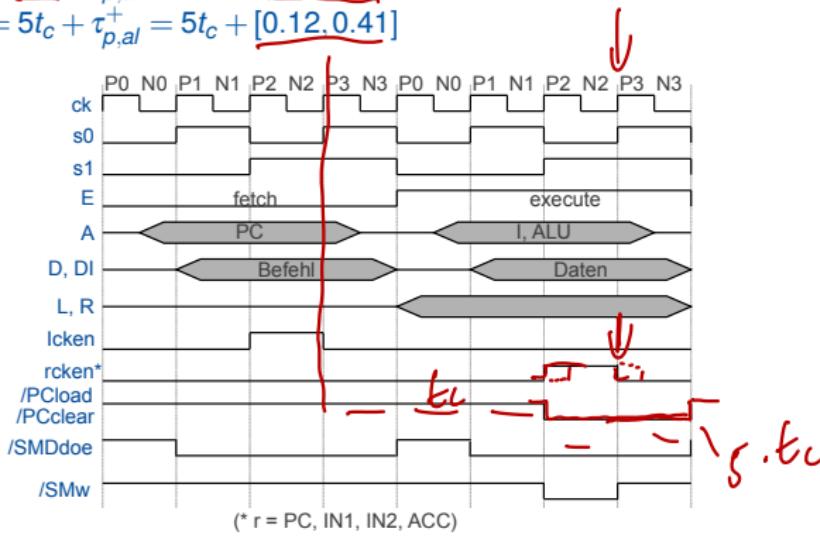
---

- *rcken* aktiv bei P2 von Execute, inaktiv bei P3 von Execute, d.h. aktiv von:
  - $\tau_{11} = 3t_c + \tau_{p,ah}^+ = 3t_c + [0.12, 0.26]$  bis
  - $\tau_{12} = 4t_c + \tau_{p,ah}^+ = 4t_c + [0.12, 0.26]$
- Setup-Zeit:
  - Für alle  $r \in \{PC, ACC, IN1, IN2\}$ :  $t_{SEC} = 0.46$
  - ⇒  $\max(\tau_{11}) + 0.46 \leq 4t_c$ , d.h.  $3t_c + 0.26 + 0.46 \leq 4t_c$  bzw.
  - ⇒  $t_c \geq 0.72$  (unkritisch im Vergleich zu bisherigen Constraints)
- Hold-Zeit:
  - Für alle  $r \in \{PC, ACC, IN1, IN2\}$ :  $t_{HEC} = 0.08$
  - ⇒  $\min(\tau_{12}) \geq 4t_c + 0.08$ , d.h.  $0.12 \not\leq 0.08$
- (Analog auch für alle anderen Takte.)

# Setup- und Hold-Zeiten /PCload beim Zähler

- Setup /L bis  $ck$ :  $t_{SLC} = 0.76$
- Hold /L nach  $ck$ :  $t_{HLC} = 0.02$
- /PCload (benötigt, wenn **neue** Werte in Zähler kommen) aktiv bei  $P2$  von Execute, inaktiv bei  $P0$  von Fetch, d.h. aktiv von:

- $\tau_{13} = \underline{3t_c + \tau_{p,al}^+} = 3t_c + \underline{[0.12, 0.41]}$  bis
- $\tau_{14} = 5t_c + \tau_{p,al}^+ = 5t_c + \underline{[0.12, 0.41]}$



# Bedingungen für Setup- und Hold-Zeiten / *PCload* beim Zähler

## ■ Setup:

$$\begin{aligned}\max(\tau_{13}) + 0.76 &\leq \min(\tau_{10}) \Leftrightarrow \\ 3t_c + 0.41 + 0.76 &\leq 4t_c \Leftrightarrow \\ t_c &\geq \underline{1.18}\end{aligned}$$

## ■ Hold:

$$\begin{aligned}\min(\tau_{14}) &\geq \max(\tau_{10} + 0.02) \Leftrightarrow \\ 5t_c + 0.12 &\geq 4t_c + 0.02 \Leftrightarrow \\ t_c &\geq \underline{-0.10}\end{aligned}$$

→ Beide unkritisch im Vergleich zu bisherigen Constraints.

$t_c \geq 1.18$

## Fazit: Zykluszeit und Befehlsrate

---

- Vorläufiges Ergebnis: Falls sich durch andere Befehle keine schärferen Bedingungen an die Zykluszeit ergeben, dann lautet sie:
  - $t_c \geq 6.70 \text{ ns}$
- Taktfrequenz:
  - $v = \frac{1}{6.70} \cdot 10^9 \text{ Hz} = 149.2 \text{ MHz}$
- 8 Takte pro Befehl  $\rightarrow$  18.6 Millionen Befehle pro Sekunde, d.h. Befehlsrate von **18.6 MIPS** (= Million Instructions per Second)

# Anmerkungen zur Timing–Analyse

---

- Eine „echte“ Timing–Analyse müsste noch Leitungslaufzeiten auf dem Chip berücksichtigen.
  - Dazu muss dann aber schon das Layout des Chips bekannt sein, um die Leitungslängen und -kapazitäten zu berechnen.
  - Leitungslaufzeiten waren früher bei einem Aufbau mit diskreten Bausteinen vernachlässigbar, sind es bei den heutigen Technologien aber nicht mehr.
- ⇒ Exakte Timing-Analysen sind heute daher kaum ohne maschinelle Unterstützung durchführbar.
- Synthesetools sind in der Lage, durch Optimierung der Treiberstärken von Grundgattern (verschiedene Versionen in der Bibliothek!) Laufzeiten zu minimieren.
  - Wird das SRAM nicht auf dem Chip integriert (d.h. stattdessen ein kommerzielles externes SRAM angeschlossen), dann muss man noch Verzögerungszeiten für I/O–Pads des Chips mit eventueller Anpassung von Spannungspegeln berücksichtigen.

## ■ Beschleunigung

- Schnellere Komponenten, z.B. ALU ([siehe Übung](#))
- Schnellere Adressberechnung (Treiber schon bei P0 öffnen)
- Verkürzung des Fetch-Zyklus um 1 Takt
- Pipelining
- Schnellerer Speicher
- ...

# Ausblick (2/2)

---

- **Fehlertoleranz** (Kapitel 6)

- Wie umgehen mit Übertragungsfehlern auf den Leitungen?
- Parity-Check, Hamming-Code
- Andere Fehler ...

- **Verifikation** (Kapitel 7)

- Ist der Entwurf der ReTI überhaupt korrekt?
- Automatische Methoden für den Beweis der Korrektheit

- **Architekturkonzepte** (Kapitel 8)

- Speicherhierarchie
- Pipelining
- Parallelität

# Kapitel 6

## **Fehlertoleranz**

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur  
WS 2016/17

# Motivation

---

- Bisher: Rechnerarchitektur am Beispiel von ReTI
- Fehler auf verschiedenen Ebenen **entdecken** und **beheben**
  - Fehler durch Fertigungsdefekte, Störungen: Kapitel 6
  - Konzeptuelle Fehler, Entwurfsfehler: Kapitel 7
- Allgemeine Rechnerarchitektur und Entwurfskonzepte

# Physikalische Fehler

---

- Bei der **Informationsverarbeitung** und -Übermittlung können **physikalische Störungen** auftreten.
  - Elektrisches Rauschen, Radiation, Defekte.
  - Absichtliche Manipulation durch Angreifer.
- Die Störungen äußern sich darin, dass Wert 0 statt Wert 1 berechnet/übertragen wird und umgekehrt.
  - „Kippen“ des Werts  $0 \rightarrow 1, 1 \rightarrow 0$ .
- Hier konzentrieren wir uns auf Fehler bei der **Datenübertragung**.
  - Manche Fehler betreffen direkt die **Hardware** oder **andere Systemteile**. Ihre Behandlung ist **komplexer**.

# Wiederholung

---

Sei  $A = \{a_1, \dots, a_m\}$  ein endliches Alphabet der Größe  $m$ .

- Eine Abbildung  $c : A \rightarrow \{0, 1\}^n$  heißt **Code fester Länge**, falls  $c$  injektiv ist.
- Die Menge  $c(A) := \{w \in \{0, 1\}^n \mid \exists a \in A : c(a) = w\}$  heißt Menge der **Codewörter**.
- Minimale Codelänge: Für einen Code  $c : A \rightarrow \{0, 1\}^n$  fester Länge gilt:  $\underline{n \geq \lceil \log_2 m \rceil}$ .

$$\begin{array}{ll} a \rightarrow \overset{\smile}{000} & e \rightarrow \overset{\smile}{100} \\ b \rightarrow 001 & f \rightarrow \underline{101} \\ c \rightarrow 010 & g \rightarrow 110 \\ d \rightarrow 011 & h \rightarrow 111 \end{array}$$

# Fehler bei Datenübertragung (1/2)

$$w \xrightarrow{w \neq \tilde{w}} \tilde{w}$$

## Annahme:

- Sei  $c$  ein Code minimaler fester Länge  $n$ .
- Ein Datum  $a$  (z.B. ein Buchstabe, eine Zahl) wird, durch ein Codewort  $w = c(a)$  repräsentiert übertragen.
- Sei  $\tilde{w} \in \{0, 1\}^n$  das empfangene Wort.
- Bei der Übertragung (z.B. über Internet) können einzelne Bits von  $c(a)$  kippen. Dann ist  $w \neq \tilde{w}$ .

# Fehler bei Datenübertragung (2/2)

---

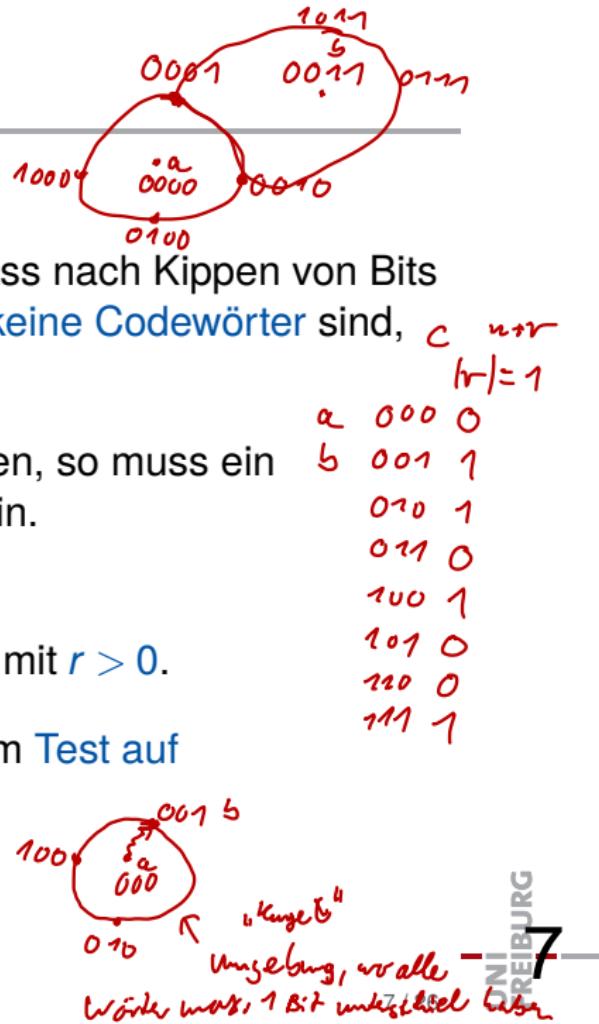
## Ziel:

- Durch Verändern des Codes  $c$  in einen Code  $C$  fester Länge  $\underline{n+r}$  sollen diese Bits
  - erkannt
    - Fehlererkennende Codes  
Bsp. für 1-fehlererkennenden Code: Parity-Code
  - korrigiert
    - Fehlerkorrigierende Codes  
Bsp. für 1-fehlerkorrigierenden Code: Hamming-Code
- werden.

# Fehlererkennende Codes

## Idee:

- Wähle Codewörter  $w \in c(A)$  so, dass nach Kippen von Bits Wörter  $\tilde{w} \in \{0,1\}^n$  entstehen, die **keine Codewörter** sind, d.h.  $\tilde{w} \notin c(A)$ .  
 $c$  nur  
 $|r|=1$   
 $a \quad 000 \quad 0$   
 $b \quad 001 \quad 1$   
 $010 \quad 1$   
 $011 \quad 0$   
 $100 \quad 1$   
 $101 \quad 0$   
 $110 \quad 0$   
 $111 \quad 1$
- Wird ein **Nicht-Codewort** empfangen, so muss ein Übertragungsfehler aufgetreten sein.
- Benutze Codes mit  $n = \lceil \log_2 m \rceil + r$  mit  $r > 0$ .
- Benutze die  $r$  zusätzlichen Bits zum Test auf Übertragungsfehler.
- Beispiel: Parity-Code



# Parity-Code

---

- Eine Bitfolge  $w \in \{0, 1\}^n$  besteht den **Paritätstest** (engl. **Parity-Check**), wenn die Anzahl der auf 1 gesetzten Bitstellen gerade ist.
- Sei  $c : A \rightarrow \{0, 1\}^n$  ein Code fester Länge von  $A$ . Betrachte den Code  $C : A \rightarrow \{0, 1\}^{n+1}$ , der aus Code  $c$  entsteht, in dem eine Bitstelle an jedes Codewort  $c(a)$  hinten angefügt wird und so gesetzt wird, dass der neue Code  $C(a)$  den Paritätstest besteht.

# Fehlererkennender Code

---

- Ein Code  $c : A \rightarrow \{0, 1\}^n$  fester Länge heißt  **$k$ -fehlererkennend**, wenn der Empfänger in jedem Fall entscheiden kann, ob ein gesendetes Codewort durch Kippen von bis zu  $\underline{k}$  Bits verfälscht wurde.
- Der Parity-Code  $C$  ist **1-fehlererkennend**.
  - **Beweis:** Kippt bei der Übertragung von  $C(a)$  genau eine Bitstelle, so kommt eine Bitfolge an, die den Paritätstest nicht besteht und somit kein Codewort von  $C$  darstellt. Überprüft der Empfänger die Parität der empfangenen Bitfolge, kann er auf einen Fehler schließen.



- Ein Code  $c : A \rightarrow \{0, 1\}^n$  fester Länge heißt  **$k$ -fehlererkennend**, wenn der Empfänger in jedem Fall entscheiden kann, ob ein gesendetes Codewort durch Kippen von bis zu  $k$  Bits verfälscht wurde.
- Der Parity-Code  $C$  ist **1-fehlererkennend**.
  - **Beweis:** Kippt bei der Übertragung von  $C(a)$  genau eine Bitstelle, so kommt eine Bitfolge an, die den Paritätstest nicht besteht und somit kein Codewort von  $C$  darstellt. Überprüft der Empfänger die Parität der empfangenen Bitfolge, kann er auf einen Fehler schließen.

# SMILE - Fehlererkennung bei Parity-Code

---

Welche Fehler bei der Übertragung werden durch den Paritätstest beim Empfänger erkannt? Das letzte Bit bei Sender und Empfänger sei dabei das Parity-Bit.

- a. Sender: 100010101, Empfänger: 100010101 ✗
- b. Sender: 100110100, Empfänger: 100010100 ✓
- c. Sender: 111010101, Empfänger: 101010101 ✓
- d. Sender: 000000000, Empfänger: 100000001 ✗  


# Hamming-Abstand (1/2)

## Definition

Der Hamming-Abstand  $\text{dist}(v, w)$  zweier  $n$ -Bitfolgen  $v$  und  $w$  ist die Anzahl der Stellen, an denen  $v$  und  $w$  sich unterscheiden.

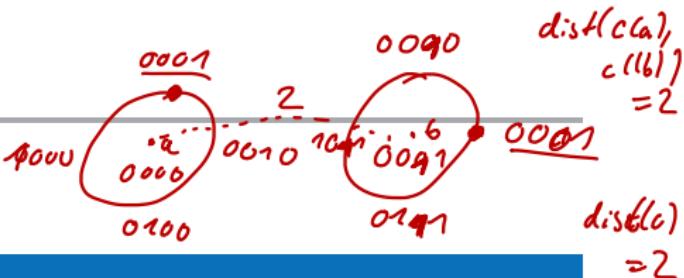
- $\text{dist}(\underline{00001101}, \underline{10001100}) = 2$
- $\text{dist}(00001101, 00001101) = 0$
- Ist  $v$  das übertragene und  $w$  das empfangene Codewort, so liegt ein Übertragungsfehler genau dann vor, wenn  $\text{dist}(v, w) \neq 0$ .
  - Ein Übertragungsfehler heißt einfach, wenn  $\text{dist}(v, w) = 1$ .
- Der Hamming-Abstand eines Codes  $c : A \rightarrow \{0, 1\}^n$  ist der kleinste Abstand zweier Codewörter von  $c$ :  
$$\text{dist}(c) := \min \{ \text{dist}(c(a_i), c(a_j)) ; a_i, a_j \in A \text{ mit } a_i \neq a_j \}.$$

$$\begin{array}{r} 000 \\ 010 \\ 101 \\ \hline \end{array}$$

$\text{dist}(\dots) = 1$

UNI  
FREIBURG

## Hamming-Abstand (2/2)



### Lemma

Ein Code  $c$  fester Länge ist genau dann  $k$ -fehlererkennend, wenn  $\underline{\text{dist}}(c) \geq k + 1$  gilt.

$$k+1 \Rightarrow k\text{-fehlererkennend}$$

- Beweisidee:** Durch das Kippen von bis zu  $\underline{l} \leq k$  Bits kann aus Codewort  $a \in c(A)$  kein anderes Codewort  $a^* \in c(A)$  entstehen, denn sonst wäre  $\text{dist}(c(a), c(a^*)) = \underline{l}$ , was kleiner als der Hamming-Abstand des Codes wäre.

# Fehlerkorrigierende Codes

---

## Idee:

- Benutze r zusätzliche Bits, so dass das gesendete Codewort aus dem empfangenen Wort rekonstruiert werden kann.
- Beispiel: Hamming-Code

# Hamming-Code

$$\begin{array}{r} 3 \ 2 \ 1 \\ \downarrow \downarrow \downarrow \\ 0 \ 0 \ 0 \\ 0 \ 0 \ 1 \end{array} \rightarrow \begin{array}{r} 6 \ 5 \ 4 \ 3 \ 2 \ 1 \\ \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \\ 0 \ 0 \ - \ 0 \ - \ - \\ 0 \ 0 \ - \ 1 \ - \ - \end{array}$$

010

011

100

101

110

111

- Benutze die Bitstellen  $\underline{2^0}, \underline{2^1}, \dots, \underline{2^{r-1}}$  als Überprüfungsbits, wobei die Bitstelle  $\underline{2^j}$  die Bitstellen überprüft, deren Binärdarstellungen an der  $\underline{j}$ -ten Stelle eine 1 haben.
- Die Bitstelle  $\underline{2^j}$  wird so belegt, dass gerade viele Bitstellen, deren Binärdarstellungen an der  $j$ -ten Stelle eine 1 haben, gesetzt sind. (vgl. Paritätstest)

$$\begin{array}{l} 2^0 = 1 \\ 2^1 = 2 \\ 2^2 = 4 \end{array}$$

# Hamming-Code an einem Beispiel

---

- Uncodiertes Wort: 0111 0101 0000 1111.

→  $m = 16$ .

↪ Länge des Code Wort

- Konstruktion des Hamming-codierten Codeworts:

- Das Wort wird unter Auslassung der „Zweierpotenz“-Bitstellen aufgeschrieben:

01110 1010000 111 1   .  
     $2^4$         $2^3$       $2^2$     $2^1$   $2^0$

- Dies ergibt insgesamt 21 Bitstellen ( $r = 5$ ).
- Die „Zweierpotenz“-Bitstellen werden als Überprüfungsbits benutzt (Nummerierung beginnt rechts mit der Stelle 1).
- Zur Erinnerung: Die Bitstelle  $2^j$  wird so belegt, dass gerade viele Bitstellen, deren Binärdarstellungen an der  $j$ -ten Stelle eine 1 haben, gesetzt sind.

# Hamming-Code-Beispiel (1/4)

0111010100001111  
01110\_1010000\_111\_1\_\_  
P  
z1  
... 111  
321

	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	zu codierende Bitfolge
→	3					1
→	5					1
→	6					1
→	7					1
→	9					0
→	10					0
→	11					0
→	12					0
→	13					1
→	14					0
→	15					1
→	17					0
→	18					1
→	19					1
→	20					1
→	21					0

11  
101  
110  
111  
1001  
1010

:

:

:

Das Überprüfungsbit  
 $2^j$  überprüft die Bitstel-  
len, die in ihrer Binär-  
darstellung an der  $j$ -ten  
Stelle eine 1 haben.

# Hamming-Code-Beispiel (2/4)

	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	zu codierende Bitfolge
3			—	—		1
5		—		—		1
6		—	—			1
7		—	—	—		1
9	—			—		0
10	—			—		0
11	—		—	—		0
12	—	—				0
13	—	—	—			1
14	<u>0</u>	<u>0</u>	<u>0</u>			0
15	<u>1</u>	<u>1</u>	<u>1</u>	<u>1</u>		1
17	—			—		0
18	—			—		1
19	—			—		1
20	—			—		1
21	—			—		0

Das Überprüfungsbit  
 $2^j$  überprüft die Bitstellen,  
die in ihrer Binär-  
darstellung an der  $j$ -ten  
Stelle eine 1 haben.

$$\begin{array}{r} 10101_2 = 21_{10} \\ \overline{11111} \\ \hline 11000 \end{array}$$

# Hamming-Code-Beispiel (3/4)

	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	zu codierende Bitfolge	
3				1 0	1 0	1	←
5			1		1	1	
6		1 0	1 0		1	1	← 0
7		1 0	1 0	1 0	1 0	1	← 0
9	0				0	0	
10	0 1		0 1		0	0	→ 1
11	0		0		0	0	
12	0	0			0	0	
13	1	1			1	1	
14	0	0	1		0	0	
15	1	1	1	1	1	1	
17	0				0	0	
18	1			1		1	
19	1			1	1	1	
20	1 0		1 0		1 0	1	←
21	0		0		0	0	

X   X   X   X   X   X   X   Parity Code

1   0   0   0   0   0 →

$\uparrow \quad 2^3 + 2^1 \uparrow = 10$

Der Prüfbitwert ergibt sich aus der Summe modulo 2 der jeweiligen Spalten.

Das Überprüfungsbit  $2^j$  überprüft die Bitstellen, die in ihrer Binärdarstellung an der  $j$ -ten Stelle eine 1 haben.

## Hamming-Code Beispiel (4/4)

---

- Die Bitfolge 0111 0101 0000 1111
- wird mit dem Hamming-Code zum Codewort  
0 1110 **1101** 0000 **0111** **0100**.

# Und wie findet man einen Fehler? (1/2)

---

- Nehme einen Übertragungsfehler an Position 13 des Codeworts an.
- Fehlerhaft empfangenes Wort:  
0 1110 110 0000 0111 0100.

## Und wie findet man einen Fehler? (2/2)

	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	empfangene Bitfolge
3				1	1	1
5			1		1	1
6		1	1			1
7		1	1	1		1
9	0				0	0
10	0		0			0
11	0		0			0
12	0	0				0
13	0	1	0	1	0	←
14	0	0	0			0
15	1	1	1	1		1
17	0			0		0
18	1			1		1
19	1		1	1		1
20	1		1			1
21	0		0	0		0

⇒ Fehler muss sich in  
Zeile  $8 + 4 + 1 = 13$   
befinden!

$$1101_2 = 13_{10}$$



Die Spalten 8,4 und 1 bestehen  
den Paritätstest nicht!

# Fehlerkorrigierende Codes

## Definition

Ein Code  $c : A \rightarrow \{0, 1\}^n$  fester Länge heißt  **$k$ -fehlerkorrigierend**, wenn der Empfänger in jedem Fall entscheiden kann, ob ein gesendetes Codewort  $w$  durch Kippen von bis zu  $k$  Bits verfälscht wurde und daraufhin  $w$  aus dem empfangenen Wort  $\tilde{w}$  rekonstruiert werden kann.

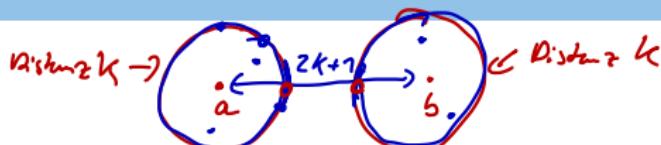
- Der Hamming-Code ist 1-fehlerkorrigierend. Die Anzahl der Zusatzbits  $r = 1 + \lfloor \log_2 m \rfloor$  ist minimal (Korrektetheit folgt aus noch folgendem Satz).

# Zusammenhang Hamming-Abstand und Fehlerkorrektur

## Lemma

Ein Code  $c$  fester Länge ist genau dann  $k$ -fehlerkorrigierend, wenn  $\underline{\text{dist}(c) \geq 2k + 1}$  gilt.

$$2 \cdot 1 + 1 = 3$$



Beweis:

- Sei  $M(c(a_i), k) := \{w \in \{0, 1\}^n \mid \text{dist}(c(a_i), w) \leq k\}$  die Kugel um  $c(a_i)$  mit Radius  $k$ .
- Dann gilt:  
 $c$  ist  $k$ -fehlerkorrigierend  $\Leftrightarrow \forall a_i, a_j, i \neq j$  gilt:  $M(c(a_i), k) \cap M(c(a_j), k) = \emptyset$ .
- Für den Beweis ist also zu zeigen:  
 $\underline{[\forall a_i, a_j, i \neq j : M(c(a_i), k) \cap M(c(a_j), k) = \emptyset]} \Leftrightarrow \underline{\text{dist}(c) \geq 2k + 1}$ .

# Beweis der Hilfs-Behauptung

$$A \Rightarrow B$$

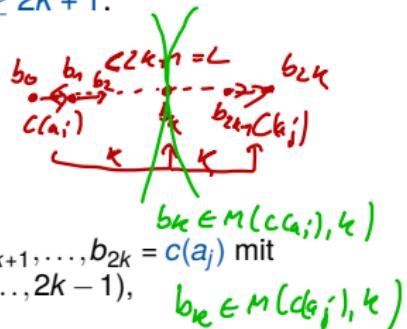
$$\neg B \Rightarrow \neg A$$

(A)

(B)

$$\rightarrow [\forall a_i, a_j \ i \neq j : \underline{M(c(a_i), k) \cap M(c(a_j), k)} = \emptyset] \Leftrightarrow \text{dist}(c) \geq 2k + 1.$$

$$c(a_i) = b_0 \\ c(a_j) = b_{2k}$$



## Beweis „ $\Rightarrow$ “:

Annahme:  $\text{dist}(c) < 2k + 1$

D.h.  $\exists a_i, a_j$  mit  $\text{dist}(c(a_i), c(a_j)) = l$  und  $l < 2k + 1$ ;

also gibt es eine Folge:  $c(a_i) = b_0, b_1, \dots, b_{k-1}, b_k, b_{k+1}, \dots, b_{2k} = c(a_j)$  mit  
 $\text{dist}(b_i, b_{i+1}) = 0$  oder  $\text{dist}(b_i, b_{i+1}) = 1$  (für alle  $i = 0, \dots, 2k - 1$ ),

also  $b_k \in M(c(a_i), k) \cap M(c(a_j), k)$ .  $\Downarrow \Rightarrow \neg A$

## Beweis „ $\Leftarrow$ “:

$\rightarrow$  Annahme:  $\underline{M(c(a_i), k) \cap M(c(a_j), k)} \neq \emptyset$ .

Es gibt also  $b$  im Durchschnitt mit:

$$\underline{\text{dist}(c) \leq \text{dist}(c(a_i), c(a_j))} \leq \text{dist}(c(a_i), b) + \text{dist}(b, c(a_j)) \leq \underline{k+k}.$$

$$\begin{matrix} \swarrow & \searrow \\ \leq k & \leq k \end{matrix}$$

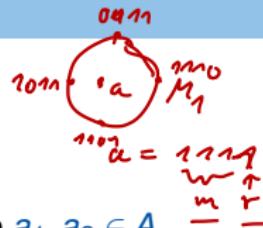
# Anzahl Zusatzbits für Fehlerkorrigierende Codes

## Satz

Für einen 1-fehlerkorrigierenden Code  $c : A \rightarrow \{0, 1\}^{m+r}$  fester Länge über  $A$  mit  $|A| = 2^m$  gilt:  $r \geq 1 + \lfloor \log_2 m \rfloor$ .

## Beweis:

- $M_1(a) := \{b \in \{0, 1\}^{m+r} : b \text{ entsteht aus } c(a) \text{ durch Kippen von bis zu 1 Bit}\}$ .
- Nach Lemma muss gelten:  $M_1(a_1) \cap M_1(a_2) = \emptyset$  für alle  $a_1, a_2 \in A$ ,
- es gilt  $|M_1(a)| = m+r+1$  für alle  $a \in A$ .
- Also müssen  $2^m$  überschneidungsfreie Kugeln, jede mit  $(m+r+1)$  Elementen, im Raum  $\mathbb{B}^{m+r}$  enthalten sein:  $2^m(m+r+1) \leq 2^{m+r}$ .
- Behauptung: Aus  $2^m(m+r+1) \leq 2^{m+r}$  folgt  $r \geq 1 + \lfloor \log_2 m \rfloor$ .
- Sei hierzu  $k := \lfloor \log_2 m \rfloor \Rightarrow 2^k \leq m$ .
- $2^m(m+r+1) \leq 2^{m+r} \Leftrightarrow m+r+1 \leq 2^r \Rightarrow 2^k + \cancel{X} + 1 \leq 2^r \Rightarrow 2^k + 1 \leq 2^r \Rightarrow k < r \Leftrightarrow k+1 \leq r \Leftrightarrow \lfloor \log_2 m \rfloor + 1 \leq r$ .



# Ausblick

---

- Wir haben bisher angenommen, dass Fehler auf Kommunikationskanälen auftreten.
- Es gibt auch Fehler in der Hardware selbst.
  - Permanente Fehler (Fertigungsdefekte)  
→ Testmethoden (Rechnerarchitektur, Spezialvorlesung „Testen“)
  - Latente Fehler („Beinahe-Defekte“)  
→ Stresstest (Spezialvorlesung „Testen“)
  - Transiente Fehler (Störungen während des Betriebs)  
→ Fehlertoleranz, Redundanz (Spezialvorlesung „Testen“)
  - Absichtlich herbeigeführte Fehler (Angriffe)
    - Aus Vergleich des Systemverhaltens mit und ohne Fehler auf geschützte Daten schließen (Fault-Based Cryptanalysis). (Seminar)
- Vor allem bei sicherheitskritischen Systemen in neuesten Fertigungstechnologien sind Fehler problematisch.

# Kapitel 7

Formale Spezifikation von Hardware:

1. **Boolesche Ausdrücke** *binary decision*
2. Binäre Entscheidungsdiagramme (BDDs) *decision diagram*
3. Anwendung: Formale Verifikation

Albert-Ludwigs-Universität Freiburg

Dr. Tobias Schubert, Dr. Ralf Wimmer

Professur für Rechnerarchitektur  
WS 2016/17

# Motivation

---

- Der Entwurf von ReTI hat (hoffentlich) gezeigt, dass Hardware-Synthese komplex und fehleranfällig ist.
- Es gibt automatische Methoden, um Fehler zu finden oder ihre Abwesenheit nachweisen zu können.
- Für ihre Anwendbarkeit muss ein Schaltkreis formal vollständig spezifiziert werden.
- Wir schauen uns daher boolesche Funktionen nochmals (und genauer) an und lernen effiziente Algorithmen und Datenstrukturen zu ihrer Handhabung.

# Motivation

---

- Der Entwurf von ReTI hat (hoffentlich) gezeigt, dass Hardware-Synthese **komplex und fehleranfällig** ist.
- Es gibt **automatische** Methoden, um **Fehler zu finden** oder ihre **Abwesenheit nachweisen** zu können.
- Für ihre Anwendbarkeit muss ein Schaltkreis **formal vollständig spezifiziert** werden.
- Wir schauen uns daher boolesche Funktionen nochmals (und genauer) an und lernen effiziente Algorithmen und Datenstrukturen zu ihrer Handhabung.

# Boolesche Algebren - allgemein

- Operation*
- Es sei  $M$  eine Menge auf der zwei binäre Operationen  $\cdot$  und  $+$  und eine unäre Operation  $\sim$  definiert sind.
  - Das Tupel  $(M, \cdot, +, \sim)$  heißt **boolesche Algebra**, falls  $M$  eine nichtleere Menge ist und für alle  $x, y, z \in M$  die folgenden Axiome gelten:

Kommutativität	$x + y = y + x$	$x \cdot y = y \cdot x$
Assoziativität	$x + (y + z) = (x + y) + z$	$x \cdot (y \cdot z) = (x \cdot y) \cdot z$
Absorption	<u><math>x + (x \cdot y) = x</math></u>	<u><math>x \cdot (x + y) = x</math></u>
Distributivität	$x + (y \cdot z) = (x + y) \cdot (x + z)$	$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$
Komplement	$x + (y \cdot \sim y) = x$	$x \cdot (y + \sim y) = x$

# Beispiele boolescher Algebren

$$\begin{array}{c} \{0,1\} \\ / \\ (\mathbb{B}, \underline{\wedge}, \underline{\vee}, \underline{\neg}) \end{array}$$

Potenzmenge  
 $S = \{a, b, c\}$   
 $\bigcup \text{Pot}(S) = \{ \emptyset, \{a\}, \{b\}, \{c\}, \{a, b, c\} \}$

- Boolesche Algebra der Teilmengen einer Menge  $S$ :  $(\text{Pot}(S), \cap, \cup, {}^c)$
- Boolesche Algebra der booleschen Funktionen in  $n$  Variablen:  
 $(\mathbb{B}_n, \cdot, +, \sim)$

⇒ Allgemein: Lässt sich eine Aussage direkt aus den Axiomen herleiten, dann gilt sie in allen booleschen Algebren!

- Man darf beim Beweis der Aussage aber auch wirklich nur die Axiome verwenden und keine Eigenschaften der konkreten booleschen Algebra.

# Boolesche Algebra der Teilmengen von $S$ ( $Pot(S), \cap, \cup, {}^C$ )

- Menge: Potenzmenge von  $S$
- $\cdot: Pot(S) \times Pot(S) \rightarrow Pot(S); (M_1, M_2) \mapsto M_1 \cap M_2$
- $+: Pot(S) \times Pot(S) \rightarrow Pot(S); (M_1, M_2) \mapsto M_1 \cup M_2$
- ${}^C: Pot(S) \rightarrow Pot(S); M \mapsto M^C := S \setminus M$

Komplement/  
Inverse/  
Negation

## Satz

$(Pot(S), \cap, \cup, {}^C)$  ist eine boolesche Algebra.

- **Beweis:** Nachrechnen, dass **alle** Axiome gelten.

**Beispiel:** Absorption

- Seien  $M_1, M_2 \in Pot(S)$ .
- Dann ist  $(M_1 + (M_1 \cdot M_2)) = (M_1 \cup (M_1 \cap M_2)) = M_1$   
und  $(M_1 \cdot (M_1 + M_2)) = (M_1 \cap (M_1 \cup M_2)) = M_1$ .

$$M_1 \subseteq M_1$$

# Boolesche Algebra der Funktionen in $n$ Variablen $(\mathbb{B}_n, \cdot, +, \sim)$

- Menge:  $\mathbb{B}_n$  (Menge der booleschen Funktionen in  $n$  Variablen)
- $\cdot: \mathbb{B}_n \times \mathbb{B}_n \rightarrow \mathbb{B}_n; (f \cdot g)(\alpha) = f(\alpha) \cdot g(\alpha)$  für alle  $\alpha \in \mathbb{B}^n$
- $+: \mathbb{B}_n \times \mathbb{B}_n \rightarrow \mathbb{B}_n; (f + g)(\alpha) = f(\alpha) + g(\alpha)$  für alle  $\alpha \in \mathbb{B}^n$
- $\sim: \mathbb{B}_n \rightarrow \mathbb{B}_n; (\sim f)(\alpha) = 1 \Leftrightarrow f(\alpha) = 0$  für alle  $\alpha \in \mathbb{B}^n$

## Satz

$(\mathbb{B}_n, \cdot, +, \sim)$  ist eine boolesche Algebra.

- **Beweis:** Nachrechnen, dass **alle** Axiome gelten.

**Beispiel:** Kommutativität

- Seien  $f, g \in \mathbb{B}_n$ .
- Für alle  $\alpha \in \mathbb{B}^n$  gilt:  $(f + g)(\alpha) = \underbrace{f(\alpha) + g(\alpha)}_{+: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}} = \underbrace{g(\alpha) + f(\alpha)}_{+: \mathbb{B}_n \times \mathbb{B}_n \rightarrow \mathbb{B}_n} = (g + f)(\alpha)$ .
- Also  $f + g = g + f$ .

# Weitere, aus den Axiomen ableitbare Regeln:

- Existenz neutraler Elemente:

$$\exists \mathbf{0} : \underline{x + 0 = x}, \underline{x \cdot 0 = 0} \quad \exists \mathbf{1} : \underline{x \cdot 1 = x}, \underline{x + 1 = 1}$$



- Doppeltes Komplement:

$$(\sim(\sim x)) = x$$

- Eindeutigkeit des Komplements:

$$(x \cdot y = \mathbf{0} \text{ und } x + y = \mathbf{1}) \Rightarrow y = (\sim x)$$

- Idempotenz:

$$x + x = x \quad x \cdot x = x$$

- de Morgan-Regel:

$$\underline{\sim(x+y) = (\sim x) \cdot (\sim y)} \quad \underline{\sim(x \cdot y) = (\sim x) + (\sim y)}$$

- Consensus-Regel: / Resolution

$$(x \cdot y) + ((\sim x) \cdot z) = (x \cdot y) + ((\sim x) \cdot z) + (y \cdot z)$$

$$(x + y) \cdot ((\sim x) + z) = (x + y) \cdot ((\sim x) + z) \cdot (y + z)$$

- Diese Regeln gelten in allen booleschen Algebren!

# Dualitätsprinzip bei booleschen Algebren

## Prinzip der Dualität

Gilt eine aus den Gesetzen der booleschen Algebra abgeleitete Gleichung  $p$ , so gilt auch die zu  $p$  duale Gleichung, die aus  $p$  hervorgeht durch gleichzeitiges Vertauschen von  $+$  und  $\cdot$ , sowie  $0$  und  $1$ .

### ■ Beispiel:

*Consensusregel*

- $(x \cdot y) + ((\sim x) \cdot z) + (y \cdot z) = (x \cdot y) + ((\sim x) \cdot z)$
- $(x + y) \cdot ((\sim x) + z) \cdot (y + z) = (x + y) \cdot ((\sim x) + z)$

# Boolesche Ausdrücke - allgemein

---

- Formal vollständige Definition boolescher Ausdrücke
  - Syntax (korrekte Schreibweise) → Def. boolescher Ausdrücke  $BE(X_n)$
  - Semantik (Bedeutung) → Interpretationsfunktion  $\Psi$  von  $BE(X_n)$
- Zweck: Einem Rechner unzweifelhaft „beibringen“, was und was nicht ein boolescher Ausdruck ist und was seine Funktion bezüglich einer booleschen Algebra ist.
- Zum Beispiel: Unterschied zwischen dem Ausdruck „ $(x_1 \cdot (\sim x_2))$ “ und der Funktion  $f = x_1 \wedge \neg x_2$ .

$BE$        $\Psi$

# Syntax boolescher Ausdrücke

- Sei  $X_n = \{x_1, \dots, x_n\}$  eine endliche Menge von Symbolen/Variablen.
- Sei  $A = \underline{X_n \cup \{0, 1, +, \cdot, \sim, (, )\}}$  ein Alphabet.

## Definition

Die Menge  $\underline{BE(X_n)}$  der **vollständig geklammerten booleschen Ausdrücke** über  $X_n$  ist eine Teilmenge von  $\underline{A^*}$ , die folgendermaßen induktiv definiert ist:

- 0, 1 und  $x_i \in X_n$   $i = 1, \dots, n$  sind boolesche Ausdrücke
- Sind  $g$  und  $h$  boolesche Ausdrücke, so auch
  - die Disjunktion  $(g + h)$ ,
  - die Konjunktion  $(g \cdot h)$ ,
  - die Negation  $(\sim g)$ .

# Schreibweise von $BE(X_n)$

---

- **Konvention:** Negation  $\sim$  bindet stärker als Konjunktion  $\cdot$ , Konjunktion  $\cdot$  bindet stärker als Disjunktion  $+$ .
  - Klammern können **wegelassen** werden, ohne dass Mehrdeutigkeiten entstehen.
- Je nach Kontext (betrachtete boolesche Algebra) schreibt man auch
  - statt **0, 1**: Die entsprechenden neutralen Elemente,
  - statt  $\cdot$ :  $\wedge, \cap$ ,
  - statt  $+$ :  $\vee, \cup$ ,
  - statt  $\sim x$ :  $\neg x, x^C, x', \bar{x}$ .
- So „vereinfachte“ Ausdrücke entsprechen zwar nicht genau der obigen Definition, es gibt aber für **jeden** solchen Ausdruck einen **äquivalenten vollständig geklammerten Ausdruck** im Sinne der Definition.
- **Beispiel:** Der äquivalente vollständige geklammerte Ausdruck für „ $x_1 \wedge \neg x_2$ “ wäre „ $(x_1 \cdot (\neg x_2))$ “.

# Semantik boolescher Ausdrücke

Typischerweise  $\mathcal{B}_n$

- Sei  $\tilde{\mathbf{M}} = (M, \cdot, +, \sim)$  eine beliebige boolesche Algebra.
- Seien  $0, 1 \in M$  die neutralen Elemente von  $\mathcal{B}$ .

## Definition

Jedem booleschen Ausdruck  $BE(X_n)$  kann durch eine Interpretationsfunktion  $\Psi : BE(X_n) \rightarrow M_n$  eine boolesche Funktion  $M_n : M^n \rightarrow M$  zugeordnet werden.

$\Psi$  wird folgendermaßen induktiv definiert:

$$\uparrow \quad \mathcal{B}^n \rightarrow \mathcal{B}$$

- $\Psi(0) = \underline{0}; \Psi(1) = \underline{1};$   
 $\Psi(\underline{x_i})(\alpha_1, \dots, \alpha_n) = \underline{\alpha_i}$  für alle  $\alpha \in M^n$  (Projektion)
- $\Psi(\underline{(g+h)}) = \underline{\Psi(g)} + \underline{\Psi(h)}$  (Disjunktion)
- $\Psi(\underline{(g \cdot h)}) = \underline{\Psi(g)} \cdot \underline{\Psi(h)}$  (Konjunktion)
- $\Psi(\underline{(\sim g)}) = \sim (\Psi(g))$  (Negation)

# Interpretation boolescher Ausdrücke

- Sei  $\underline{e}$  ein boolescher Ausdruck.

- $\Psi(e)(\alpha)$  für ein  $\alpha \in M^n$  ergibt sich durch Ersetzen von  $\underline{x_i}$  durch  $\underline{\alpha_i}$  in  $e$ , für alle  $i$  und Rechnen in der booleschen Algebra  $\tilde{M}$ .  
*n-elem. Tupel aus gl*
- Gilt  $\underline{\Psi(e) = f}$  für eine boolesche Funktion  $f \in M_n$ , so sagen wir, dass  $e$  ein boolescher Ausdruck für  $f$  ist, bzw. dass  $e$  die boolesche Funktion  $f$  beschreibt.
- Zwei boolesche Ausdrücke  $e_1$  und  $e_2$  heißen äquivalent ( $e_1 \equiv e_2$ ) genau dann, wenn  $\underline{\Psi(e_1)} = \underline{\Psi(e_2)}$ .  
Sie sind gleich, wenn  $\underline{e_1} = \underline{e_2}$ .

- Wir betrachten folgend nur noch die Interpretation in  $B = (\mathbb{B}, \wedge, \vee, \neg)$ .

# Boolesche Ausdrücke $\leftrightarrow$ boolesche Funktionen

## Lemma 1

Zu jedem booleschen Ausdruck  $e \in BE(X_n)$  existiert eine boolesche Funktion  $f$ , die durch  $e$  beschrieben wird.

■ **Beweis:**  $f := \underline{\Psi(e)}$

## Lemma 2

Zu jeder booleschen Funktion  $f$  existiert ein boolescher Ausdruck, der  $f$  beschreibt.

■ **Beweis:** Es gilt:  $f = \underline{\Psi}(\sum_{\alpha \in ON(f)} m(\alpha))$ .

m.a.W. Die DNF ist ein Boolescher Ausdruck.  
*mit anderen Worten*

# Zusammenhang mit Schaltkreisen

## Lemma 3

Zu jedem booleschen Ausdruck  $e \in BE(X_n)$  gibt es einen kombinatorischen Schaltkreis, der  $e$  implementiert.

### ■ Beweis: Übung

$$e \in \mathcal{B}E(x_n) \xrightarrow{\sim} f = \gamma(e)$$

$\uparrow \downarrow$   
 $Skf$

- Zu jeder booleschen Funktion gibt es einen kombinatorischen Schaltkreis, der sie implementiert (zum Beispiel zweistufige Umsetzung der DNF/KDNF).
- Zu jedem kombinatorischen Schaltkreis gibt es sowohl eine boolesche Funktion, als auch einen boolescher Ausdruck.