

# Mathematische Grundlagen

Albert-Ludwigs-Universität Freiburg

Prof. Dr. Christoph Scholl

Institut für Informatik  
Sommersemester 2023

UNI  
FREIBURG

# Mathematische Grundlagen

---

- Verständigung auf gemeinsame Basis
- Die meisten Begriffe sollten bekannt sein, weil sie in anderen Vorlesungen bereits eingeführt wurden.
- Hier: Kurzer Überblick
  - Mengen, Funktionen, Relationen
  - Boolesche Algebra ( $\{0, 1\}$ ,  $\wedge$ ,  $\vee$ ,  $\neg$ )
  - O-Notation
  - Beweistechniken

# „Philosophie“ der Mathematik

---

- 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:  $\emptyset$  (es gibt kein  $a \in \emptyset$ ).
- Menge der natürlichen Zahlen:  $\mathbb{N} = \{0, 1, 2, \dots\}$ .
- Menge der booleschen Werte:  $\mathbb{B} = \{0, 1\}$ .
- 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.  $\{a, b, \{a\}, \{a, b\}\}$ .

# 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**:  
$$\mathbb{Q} = \{p/q \mid p \in \mathbb{Z}, q \in \mathbb{N}, q \neq 0, p, q \text{ teilerfremd}\}.$$
- Menge der **endlichen Zeichenketten**:  
$$STRINGS = \{s_1 s_2 \dots s_n \mid n \in \mathbb{N}, s_i \text{ ein Buchstabe}\}.$$

# Teilmengen, Potenzmenge, Mächtigkeit

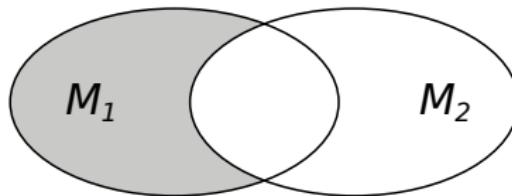
---

- Menge  $U$  ist **Teilmenge** von  $M$ , wenn jedes Element von  $U$  auch Element von  $M$  ist.
  - Notation:  $U \subset M$  bzw.  $M \supset U$
  - Achtung:  $\{a\} \subset \{a,b,c\}$ , aber  $a \in \{a,b,c\}$
- **Potenzmenge** von  $M$ :  $Pot(M) = \{m \mid m \subset M\}$ .
  - $Pot(\{a,b,c\})$   
 $= \{\emptyset, \{a\}, \{b\}, \{c\}, \{a,b\}, \{a,c\}, \{b,c\}, \{a,b,c\}\}$
- Die Anzahl  $|M|$  der Elemente einer Menge  $M$  heißt **Mächtigkeit** oder **Kardinalität** von  $M$ .

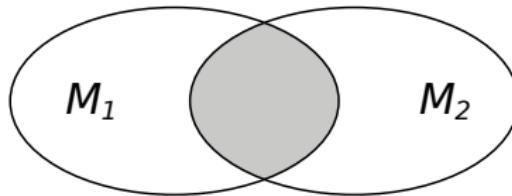
# Operationen auf Mengen 1/2

---

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



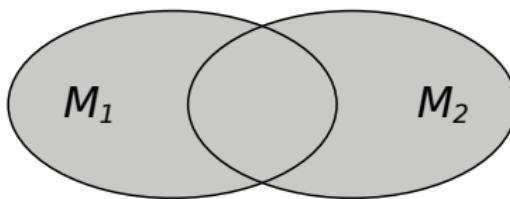
- **Mengenschnitt:**  $M_1 \cap M_2 = \{m \mid m \in M_1 \text{ und } m \in M_2\}$



# 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:

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

- $(m_1, m_2)$  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).

## Definition

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

- Notation: Statt  $(x,y) \in R$  schreibt man  $xRy$ .
- Beispiele:
  - Relation  $<$  zwischen  $\mathbb{N}$  und  $\mathbb{N}$ .  
 $<= \{(0,1), (0,2), \dots, (1,2), (1,3), \dots\}$
  - $R = \{(a,b) \mid a, b \in \mathbb{N}, a+b \text{ ungerade}\}$

# 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:

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

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

- **Sinusfunktion**  $\sin = \{(x, \sin(x)) \mid x \in \mathbb{R}\}$

# Beispiele: Relationen, Funktionen

---

- Jede Funktion ist auch eine Relation.
- Aber es gibt natürlich Relationen, die keine Funktionen sind.
- Beispiel:
  - $\sin^{-1} = \{(\sin(x), x) \mid x \in \mathbb{R}\}$  ist eine Relation, aber keine Funktion!

# Summen und Produkte (Notation)

---

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

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

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

- Beispiel:

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

- Schreibweise mit beliebigen Bedingungen:

$$\sum_{\substack{i,j > 0, \\ i+2j \leq 5}} (i^2/j) = (1^2/1) + (1^2/2) + (2^2/1) + (3^2/1) = 14,5$$

## Definition

■  $\mathbb{B} := \{0, 1\}$

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

$$0 \wedge 0 = 0, \quad 0 \wedge 1 = 0, \quad 1 \wedge 0 = 0, \quad 1 \wedge 1 = 1$$

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

$$0 \vee 0 = 0, \quad 0 \vee 1 = 1, \quad 1 \vee 0 = 1, \quad 1 \vee 1 = 1$$

■ **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.

## 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  $\cdot$  bindet stärker als Disjunktion  $+$ .

Eine Menge  $(M, \cdot, +, \neg)$  heißt Boolesche Algebra, wenn für alle  $x, y, z \in M$  folgende Axiome gelten:

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

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

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

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

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

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

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

- Man zeigt, dass ( $\{0, 1\}, \wedge, \vee, \neg$ ) eine Boolesche Algebra ist, indem man beweist, dass alle Axiome einer Booleschen Algebra für ( $\{0, 1\}, \wedge, \vee, \neg$ ) gelten.
- Die folgenden Regeln sind aus den Axiomen ableitbar:

Regeln für boolesche Algebren:  $\forall x, y, z \in M$  gilt:

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:

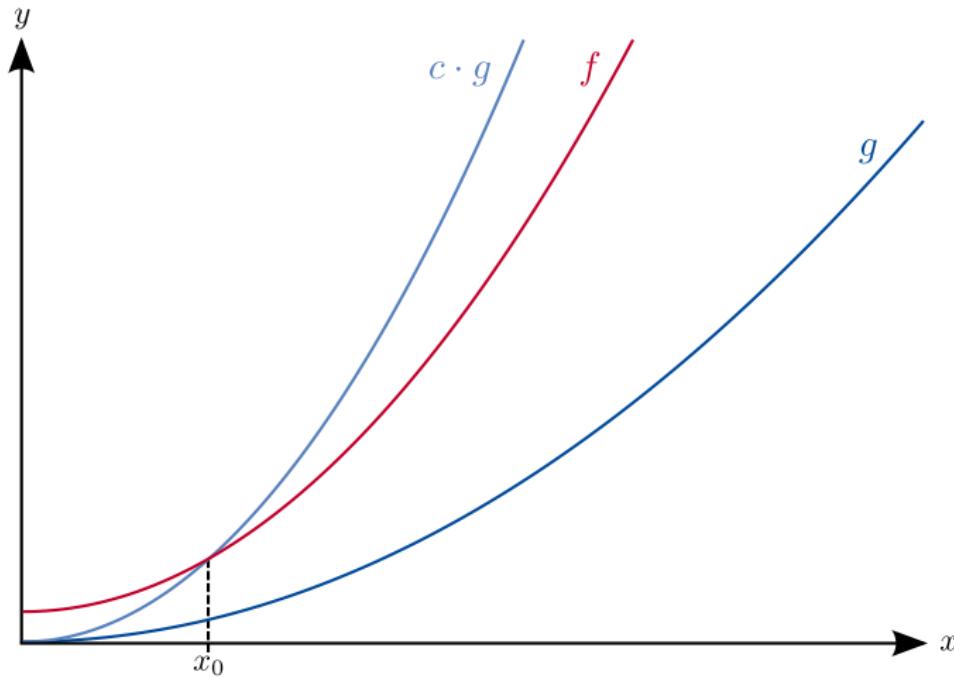
$$\begin{aligned} & (x \cdot y) + ((\neg x) \cdot z) \\ &= (x \cdot y) + ((\neg x) \cdot z) + (y \cdot z) \\ & (x + y) \cdot ((\neg x) + z) \\ &= (x + y) \cdot ((\neg x) + z) \cdot (y + z) \end{aligned}$$

# Groß-O-Notation (1/2)

---

- 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:  $5x + 2 \in O(x^2)$   
Beweis: Setze  $c = 6, x_0 = 2$   
$$5x + 2 \stackrel{2 < x}{<} 5x + x = 6x \stackrel{1 \leq x}{\leq} 6 \cdot x^2, \text{ für } x > 2.$$
- 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)



$$f(x) \in O(g(x))$$

# Beweistechniken

---

- **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  $B$ .

# Beispiel: Sukzessive Folgerungen

- 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  $c_f, x_{0f} \in \mathbb{R}_0^+$ :  $f(x) \leq c_f \cdot g(x)$  für alle  $x > x_{0f}$ . Aus  $g(x) \in O(h(x))$  folgt die Existenz von  $c_g, x_{0g} \in \mathbb{R}_0^+$ :  $g(x) \leq c_g \cdot h(x)$  für alle  $x > x_{0g}$ .
- 2 Man setze  $x_0 := \max\{x_{0f}, x_{0g}\}$ . Dann gilt für alle  $x > x_0$  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 alle  $x > x_0$ :  
$$f(x) \leq c_f \cdot g(x) \leq c_f \cdot (c_g \cdot h(x)) = c \cdot h(x).$$
Dies bedeutet aber gerade  $f(x) \in O(h(x))$

# Indirekter Beweis 1/2

---

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  $\neg C$ ”, “ $31 = 42$ ”, ...).
- Da der Widerspruch schrittweise aus  $\neg S$  logisch hergeleitet wurde, kann  $\neg S$  nicht gelten und somit muss  $S$  gelten.

## Indirekter Beweis 2/2

---

- Betrachte den Spezialfall  $S = A \Rightarrow B$ .
  - Dann ist  $\neg S = A \wedge \neg B$ . Man nimmt also an, dass  $A$  gilt, aber  $\neg B$ .
  - Ergibt sich aus der Annahme ein Widerspruch, dann muss aus der Gültigkeit von  $A$  die Gültigkeit von  $B$  folgen.
  - Ergibt sich der Widerspruch speziell durch Herleitung von  $\neg A$  aus  $\neg B$ , dann reduziert sich der Widerspruchsbeweis auf den Spezialfall **Beweis der “Kontraposition”**  $\neg B \Rightarrow \neg A$ .
  - $A \Rightarrow B$  und  $\neg B \Rightarrow \neg A$  sind logisch äquivalent.
- Implizit setzt man immer die Gültigkeit sämtlicher Axiome voraus. Sei  $Ax$  die Aussage “Sämtliche Axiome gelten”.
- Dann ist  $S' = (A \wedge Ax) \Rightarrow B$  zu beweisen.
- Annahme ist dann also:  $\neg S' = A \wedge Ax \wedge \neg B$  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  $x_0 \in \mathbb{R}_0^+$ , so dass für alle  $x > x_0$  gilt:

$$x^2 \leq c \cdot x \tag{1}$$

- Beweisstrategie: Versuche ein  $x_1 > x_0$  zu finden mit  $x_1^2 > c \cdot x_1$  – das wäre der gewünschte Widerspruch.
- Für alle  $x > c \in \mathbb{R}_0^+$  ist  $x^2 > c \cdot x$ . Ein beliebiges  $x_1 > c$  liefert also einen Widerspruch zu (1)!

# 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.

# Vollständige Induktion: Beispiel (1/2)

---

## ■ Behauptung:

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

## ■ Induktionsanfang:

Zeige die Behauptung für  $n = 0$ .

$$\sum_{k=1}^0 \frac{1}{k(k+1)} = 0 = \frac{0}{0+1}$$

# Vollständige Induktion: Beispiel (2/2)

---

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

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

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

## ■ **Induktionsschritt:**

Zeige die Behauptung für  $n + 1$ .

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

# Technische Informatik

## Organisatorisches

Albert-Ludwigs-Universität Freiburg

Prof. Dr. Christoph Scholl

Institut für Informatik  
Sommersemester 2023

UNI  
FREIBURG

# Organisatorisches - Personen

---

- Prof. Dr. Christoph Scholl  
Büro: Geb. 51, 2. Stock, Raum 02..033  
Sprechstunde: nach Vereinbarung  
[scholl@informatik.uni-freiburg.de](mailto:scholl@informatik.uni-freiburg.de)
- Betreuung der Übungen:  
Tobias Seufert, M.Sc.  
Büro: Geb. 51, 2. Stock, Raum 02..032  
[seufert@informatik.uni-freiburg.de](mailto:seufert@informatik.uni-freiburg.de)

- Vorlesung

- Montag, 14:15 - 16:00 h, Geb. 101, HS 00 026
  - Mittwoch, 12:15 - 13:00 h, Geb. 101, HS 00 026

- Übungen:

- 9 Übungsgruppen
  - Termine: Di 8-10 h, Mi 8-10 h, Mi 14-16 h, Do 8-10 h, ~~Fr~~ 16-18 h, Ort siehe HISinOne bzw. Lehrstuhlseite
  - Übungsausgabe und -abgabe jeweils mittwochs.
  - Übungsabgabe mittwochs bis 13:00 h.
  - Nähere Informationen zur Anmeldung etc. gleich ...

# Webseite zur Vorlesung (1/4)

■ <http://abs.informatik.uni-freiburg.de>  
⇒ Lehre ⇒ Technische Informatik

The screenshot shows a web browser window with the URL [https://abs.informatik.uni-freiburg.de/src/teach\\_main.php?id=168](https://abs.informatik.uni-freiburg.de/src/teach_main.php?id=168). The page is titled "Technische Informatik - Sommersemester 2023". The left sidebar has a red circle around the "Lehre" link. The main content area includes sections for "Beschreibung" and "Kommentar", both with red circles around them. The "Beschreibung" section contains text about the course content, mentioning Boolean functions, decision diagrams, and logic synthesis. The "Kommentar" section notes that the course is part of the Bachelor's program in Informatics, ESE, and Lehramt, with a credit value of 3-1 SWS (6 ECTS). At the bottom, there are links for "Link zu den Materialien" and "Link zum Übungspool", both with red circles around them. A note at the bottom states: "Wichtig: Die Einteilung der Übungsgruppen erfolgt im Rahmen der Veranstaltung (und nicht im HisInOne)." The Uni Freiburg logo is visible on the right side of the page.

# Webseite zur Vorlesung (2/4)

## ■ Link zu Iiaskurs

The screenshot shows the central learning platform of the University of Freiburg. The URL is https://lms.uni-freiburg.de/goto.php?target=crs\_304044&client\_id=uniwebzug. The page title is 'Zentrale Lernplattform der Universität Freiburg'. The navigation bar includes links for Magazin, Lehre/Veranstaltungen im SoSe 2023, Technische Fakultät, Studiengang Informatik/Bewegte Informatik, Bachelor - Grundlagen der Informatik, and Technische Informatik (SoSe 2023). A sidebar on the left contains links for Überblick, Benutzer und Gruppen, Mein Arbeitsraum, Heimwerker, Seminar, Kommunikation, and Support. The main content area displays course information for 'Technische Informatik (SoSe 2023)'. It includes sections for Vorlesung (lectures), Vorlesungs- und Übungsmaterialien (lecture and exercise materials), Links (links), and Kommunikation & Kooperation (communication and cooperation). The 'Vorlesung' section specifies lectures on Monday from 14:00 to 16:00 and Wednesday from 13:00 to 14:00 in HS 00 D06 in Geb. 101. The 'Vorlesungs- und Übungsmaterialien' section lists 'Alle Materialien (Passwort: ti2023lect)' and a link to the exercise portal. The 'Links' section points to the professor's website and course information. The 'Kommunikation & Kooperation' section features a forum with one post by Tobias Seufert.

- Zugang über Rechenzentrumsaccount
- Für erstmaligen Beitritt zum Kurs: Passwort ti2023lect
- Alle Vorlesungsmaterialien vom Iiaskurs aus ohne zusätzliches Passwort zugreifbar.

# Webseite zur Vorlesung (3/4)

---

- Link zu Vorlesungsmaterialien:
  - [Nicht-annotierte Folien](#) (vor Vorlesung zur Verfügung gestellt)
  - [Annotierte Folien](#) (nach Vorlesung) ↗
  - [Vorlesungsaufzeichnungen](#)
  - [Übungsblätter](#)
  - Alle Materialien sind passwortgeschützt.
    - Passwort: ti2023lect
  - Einmalige Passworteingabe, danach Zugriff auf alle nextcloud-Ordner der Vorlesung möglich

# Webseite zur Vorlesung (4/4)

## ■ Link zu Übungsportal



The screenshot shows a web browser window displaying the 'Übungsportal' login page for the 'Technische Informatik SoSe 2023'. The URL in the address bar is <https://ira.informatik.uni-freiburg.de/cgi-bin/teaching/ti-ss23/login.cgi>. The page features a vertical decorative sidebar on the left with horizontal stripes in orange, yellow, red, and green. On the sidebar, there are two buttons: 'Login' (orange) and 'Anmeldung' (yellow, underlined). The main content area has a large title 'Übungsportal' and the subtitle 'Technische Informatik SoSe 2023'. A red message states 'Das Portal ist derzeit wegen Wartungsarbeiten offline'. Below this, there is a link 'Link zu ILIAS-Kurs'. The 'Login:' section contains fields for 'Gruppe/Tutor:' and 'Passwort:', both with placeholder text (''). A 'Login' button is located below these fields. At the bottom, there is a link 'Zugangsdaten vergessen oder nicht erhalten?'.

# Übungsbetrieb (1/3)

---

- Ausgabe der Übungsblätter auf Vorlesungsseite / ILIAS mittwochs bis 17:00 h
- Erstellung der Lösungen zu Übungsaufgaben im pdf-Format (siehe Übungsblatt 0, schon verfügbar)
- Abgabe der Lösungen im pdf-Format per Upload im Übungsportal
  - Übungsabgabe in festen Zweiergruppen (oder einzeln)
  - Abgabe i.d.R. am darauffolgenden **Mittwoch bis 13:00 h.**
- Rückgabe der korrigierten Lösungen ebenfalls über Übungsportal
- Besprechung der Übungsblätter in darauffolgender Woche in Übungsgruppe

# Übungsbetrieb (2/3)

---

## ■ Übungsportal:

- Link zum Übungsportal auf Vorlesungs-Homepage und in ILIAS.
- **Wichtig:** Gruppenvergabe (9 Gruppen) erfolgt über die Anmeldung am Übungsportal und **nicht** über HisInOne!
- Anleitung zur Anmeldung auf Übungsblatt 0, schon verfügbar.
- Anmeldung Übungsportal: **Beginn: 17.04.23 (heute!), 16:30 Uhr, Ende: 24.04.23, 23:59 h.**
- Ausgabe des ersten regulären Übungsblatts Mittwoch, 19.04.23, Abgabe Mittwoch, 26.04.23, Besprechung in den Übungsgruppen der darauffolgenden Woche

# Demo: Anmeldung an Übungsportal (3/3)

- Examplarisch für Anmeldung des Studierenden Matthias Mustermann zur Veranstaltung “Betriebssysteme WS20/21”

The screenshot shows the login interface of the Übungsportal. On the left, there is a vertical decorative bar with horizontal stripes in orange, yellow, and red. Two pens (one red, one green) are positioned behind the bar. The main content area has a light gray background. At the top center, the text "Übungsportal" is displayed in a large, bold, black font. Below it, the text "Betriebssysteme WS20/21 Demo" is shown in a smaller, regular black font. A red oval highlights the "Anmeldung" button, which is located in the top-left corner of the login form. The login form itself has a white background and contains the following fields:

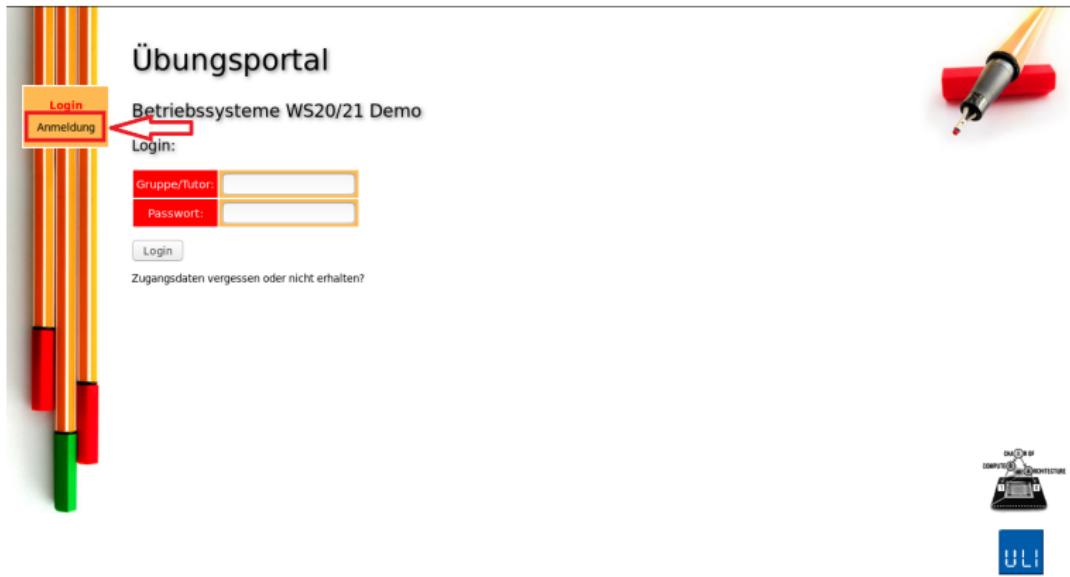
- "Login:" followed by two input fields: "Gruppe/Tutor:" and "Passwort:".
- A "Login" button below the input fields.
- A link "Zugangsdaten vergessen oder nicht erhalten?" at the bottom of the form.

On the right side of the page, there is a graphic of a red pen and a black pen lying next to each other. In the bottom right corner, there is a small logo for the University of Freiburg (UNI FREIBURG) and another logo for the Institute for Computer Science (Institut für Informatik). At the very bottom right, there is a blue square containing the letters "ULI".

Portal Revision 865, 2008-01-16 13:51:47 +0100 (Wed, 16 Jan 2008) by T. Nopper

# Demo: Anmeldung an Übungsportal (3/3)

- Examplarisch für Anmeldung des Studierenden Matthias Mustermann zur Veranstaltung "Betriebssysteme WS20/21"



Portal Revision 865, 2008-01-16 13:51:47 +0100 (Wed, 16 Jan 2008) by T. Nopper

# Demo: Anmeldung an Übungsportal (3/3)

- Examplarisch für Anmeldung des Studierenden Matthias Mustermann zur Veranstaltung "Betriebssysteme WS20/21"

**Übungsportal**

Betriebssysteme WS20/21 Demo

Anmeldung:

Schritt 1 von 5: Bitte geben Sie Ihre persönlichen Daten ein:

Vorname(n)	Matthias
Nachname	Mustermann
Matrikelnummer	1111111
Studiengang	Informatik
Semester	3
E-Mail	matthias.demo@mustermann.demo

Bitte geben Sie unbedingt eine gültige E-Mail-Adresse an, da Ihr Zugangspasswort an diese Adresse geschickt wird.

Zurücksetzen Weiter->

Portal Revision 865, 2008-01-16 13:51:47 +0100 (Wed, 16 Jan 2008) by T. Nopper

# Demo: Anmeldung an Übungsportal (3/3)

- Examplarisch für Anmeldung des Studierenden Matthias Mustermann zur Veranstaltung "Betriebssysteme WS20/21"

**Übungsportal**

Betriebssysteme WS20/21 Demo

Anmeldung:

Schritt 1 von 5: Bitte geben Sie Ihre persönlichen Daten ein:

Vorname(n)	Matthias
Nachname	Mustermann
Matrikelnummer	111111
Studienfach	Informatik
Semester	3
E-Mail	matthias.demo@mustermann.demo

Bitte geben Sie unbedingt eine gültige E-Mail-Adresse an, da Ihr Zugangspasswort an diese Adresse geschickt wird.

Zurücksetzen **Weiter->**


Portal Revision 865, 2008-01-16 13:51:47 +0100 (Wed, 16 Jan 2008) by T. Nopper

# Demo: Anmeldung an Übungsportal (3/3)

- Examplarisch für Anmeldung des Studierenden Matthias Mustermann zur Veranstaltung "Betriebssysteme WS20/21"

The screenshot shows a web-based application for course registration. On the left, there's a vertical sidebar with a yellow header containing 'Login' and 'Anmeldung'. The main area has a title 'Übungsportal' and a subtitle 'Betriebssysteme WS20/21 Demo'. Below this, a section titled 'Anmeldung:' contains the instruction 'Schritt 2 von 5: Bitte wählen Sie eine Gruppe:'. A table lists eight tutor sessions with their details:

Auswahl	Tutor	Zeit	Ort	Freie Plätze
<input checked="" type="radio"/>	Tutor1	Mi 08:00-10:00	online	10
<input type="radio"/>	Tutor2	Mi 08:00-10:00	online	12
<input type="radio"/>	Tutor3	Mi 12:00-14:00	online	12
<input type="radio"/>	Tutor4	Mi 12:00-14:00	online	12
<input type="radio"/>	Tutor5	Mi 12:00-14:00	online	12
<input type="radio"/>	Tutor6	Do 08:00-10:00	online	12
<input type="radio"/>	Tutor7	Do 08:00-10:00	online	12
<input type="radio"/>	Tutor8	Do 08:00-10:00	online	12

At the bottom of the page are three buttons: '<-Zurück', 'Zurücksetzen', and 'Weiter->'. To the right of the table, there's a small logo of a telephone handset with text above it. At the very bottom right, there's a blue square with the letters 'ULI'.

Portal Revision 865, 2008-01-16 13:51:47 +0100 (Wed, 16 Jan 2008) by T. Nopper

# Demo: Anmeldung an Übungsportal (3/3)

- Examplarisch für Anmeldung des Studierenden Matthias Mustermann zur Veranstaltung "Betriebssysteme WS20/21"

**Übungsportal**

Betriebssysteme WS20/21 Demo

Anmeldung:

Schritt 2 von 5: Bitte wählen Sie eine Gruppe:

Auswahl	Tutor	Zeit	Ort	Freie Plätze
<input checked="" type="radio"/>	Tutor1	Mi 08:00-10:00	online	10
<input type="radio"/>	Tutor2	Mi 08:00-10:00	online	12
<input type="radio"/>	Tutor3	Mi 12:00-14:00	online	12
<input type="radio"/>	Tutor4	Mi 12:00-14:00	online	12
<input type="radio"/>	Tutor5	Mi 12:00-14:00	online	12
<input type="radio"/>	Tutor6	Do 08:00-10:00	online	12
<input type="radio"/>	Tutor7	Do 08:00-10:00	online	12
<input type="radio"/>	Tutor8	Do 08:00-10:00	online	12

<-Zurück Zurücksetzen Weiter->

DAI-IPB ULI  
UNIVERSITÄT FREIBURG  
INFORMATIK

# Demo: Anmeldung an Übungsportal (3/3)

- Examplarisch für Anmeldung des Studierenden Matthias Mustermann zur Veranstaltung "Betriebssysteme WS20/21"

Übungsportal

Betriebssysteme WS20/21 Demo

Anmeldung:

Schritt 3 von 5: Bitte wählen Sie eine Subgruppe:

Einer bestehenden Subgruppe beitreten

Max Mustermann

Eine neue Subgruppe eröffnen

Neue Subgruppe

Einzelabgabe

Neue Subgruppe, zu der sich niemand zusätzlich anmelden kann

<-Zurück Zurücksetzen Weiter->

DAI-BE  
DAMPF-UNIVERSITÄT  
ULI

Portal Revision 865, 2008-01-16 13:51:47 +0100 (Wed, 16 Jan 2008) by T. Nopper

# Demo: Anmeldung an Übungsportal (3/3)

- Examplarisch für Anmeldung des Studierenden Matthias Mustermann zur Veranstaltung "Betriebssysteme WS20/21"

**Übungsportal**

Betriebssysteme WS20/21 Demo

Anmeldung:

Schritt 3 von 5: Bitte wählen Sie eine Subgruppe:

Einer bestehenden Subgruppe beitreten  
Max Mustermann

Eine neue Subgruppe eröffnen  
Neue Subgruppe

Einzelabgabe  
Neue Subgruppe, zu der sich niemand zusätzlich anmelden kann

<-Zurück Zurücksetzen Weiter->

DAI-BW  
DOPPELTE  
BETRIEBS  
SYSTEME

ULI

Portal Revision 865, 2008-01-16 13:51:47 +0100 (Wed, 16 Jan 2008) by T. Nopper

# Demo: Anmeldung an Übungsportal (3/3)

- Examplarisch für Anmeldung des Studierenden Matthias Mustermann zur Veranstaltung "Betriebssysteme WS20/21"

The screenshot shows a web-based registration form for the Übungsportal. The left side features a vertical decorative bar with horizontal stripes in orange, red, and yellow. The main content area has a light gray background.

**Übungsportal**

Betriebssysteme WS20/21 Demo

Anmeldung:

Schritt 4 von 5: Bitte bestätigen Sie Ihre Angaben:

Vorname(n)	Matthias
Nachname	Mustermann
Matrikelnummer	1111111
Studiengang	Informatik
Semester	3
E-Mail	matthias.demo@mustermann.demo
Gruppe 1	Tutor1 Mi 08:00-10:00 <a href="https://bbblink.demo.de">https://bbblink.demo.de</a>
Subgruppe 1	Max Mustermann

<-Zurück Fertigstellen->

On the right side of the form, there is a graphic of a red pen and a yellow pencil. In the bottom right corner, there is a small logo for the University of Freiburg (ULI) and another logo for the Institute for Computer Science (Informatik).

Portal Revision 865, 2008-01-16 13:51:47 +0100 (Wed, 16 Jan 2008) by T. Nopper

# Demo: Anmeldung an Übungsportal (3/3)

- Examplarisch für Anmeldung des Studierenden Matthias Mustermann zur Veranstaltung "Betriebssysteme WS20/21"

The screenshot shows a web-based registration form for the Übungsportal. On the left, there's a vertical decorative bar with horizontal stripes in orange, yellow, and red. A red arrow points from the 'Fertigstellen->' button to the right side of the screen.

**Übungsportal**

Betriebssysteme WS20/21 Demo

Anmeldung:

Schritt 4 von 5: Bitte bestätigen Sie Ihre Angaben:

Vorname(n)	Matthias
Nachname	Mustermann
Matrikelnummer	1111111
Studiengang	Informatik
Semester	3
E-Mail	matthias.demo@mustermann.demo
Gruppe 1	Tutor1 Mi 08:00-10:00 <a href="https://bbblink.demo.de">https://bbblink.demo.de</a>
Subgruppe 1	Max Mustermann

<-Zurück Fertigstellen->

Portal Revision 865, 2008-01-16 13:51:47 +0100 (Wed, 16 Jan 2008) by T. Nopper

# Demo: Anmeldung an Übungsportal (3/3)

- Examplarisch für Anmeldung des Studierenden Matthias Mustermann zur Veranstaltung "Betriebssysteme WS20/21"

The screenshot shows a web-based login interface for a course titled "Betriebssysteme WS20/21 Demo". On the left, there is a vertical sidebar with a yellow header containing "Login" and "Anmeldung" (the latter being bolded). The main content area has a light gray background with orange and red vertical stripes on the left side. It displays the course title "Betriebssysteme WS20/21 Demo" and the message "Anmeldung:". Below this, it says "Schritt 5 von 5: Anmeldung abgeschlossen." followed by "Sie haben sich erfolgreich zu den Übungen zur Veranstaltung Betriebssysteme WS20/21 Demo angemeldet." and "Ihre Zugangsdaten werden Ihnen per Mail zugeschickt." A red underline is placed under the last sentence. In the top right corner, there is a graphic of a red pen and a black marker. At the bottom right, there is a logo for "DAI UNI STUTTGART" featuring a small computer monitor icon above the text "DAI UNI STUTTGART". Below that is a blue square with the letters "ULI".

Portal Revision 865, 2008-01-16 13:51:47 +0100 (Wed, 16 Jan 2008) by T. Nopper

# Demo: Anmeldung an Übungsportal (3/3)

- Examplarisch für Anmeldung des Studierenden Matthias Mustermann zur Veranstaltung "Betriebssysteme WS20/21"

The screenshot shows the login interface of the Übungsportal. On the left, there's a vertical decorative bar with horizontal stripes in orange, red, and yellow. At the top right is a red and black pen icon. The main area has a light gray background. At the top center, it says "Übungsportal". Below that, "Betriebssysteme WS20/21 Demo" is displayed. A "Login" button is highlighted with a red oval. To its right, the word "Anmeldung" is shown in a smaller box. The login form consists of two input fields: "Gruppe/Tutor:" and "Passwort:", both with red ovals around them. Below the fields is a "Login" button. At the bottom of the form, there's a link "Zugangsdaten vergessen oder nicht erhalten?". In the bottom right corner, there's a small logo for "DAI-BW LSF" featuring a computer monitor and the text "DABW LSF", "DÖPFLER", and "SCHOTTSTEIGE". Below that is a blue square containing the letters "ULI".

Portal Revision 865, 2008-01-16 13:51:47 +0100 (Wed, 16 Jan 2008) by T. Nopper

# Demo: Anmeldung an Übungsportal (3/3)

- Examplarisch für Anmeldung des Studierenden Matthias Mustermann zur Veranstaltung "Betriebssysteme WS20/21"

## Übungsportal

Betriebssysteme WS20/21 Demo  
Gruppe 1.1 - Max Mustermann **Matthias Mustermann**  
Tutor: Tutor1 - Raum: Meetingraum Tutorat - Zeit: Mi 08:00-10:00

Blatt	Abgabetermin	Aufgaben	Ihre Abgaben	Korrekturen	Punkte		Neue Abgabe
					M.M.	M.M.	
1	Freitag, 13. NOV. 2020 17:00	Aufgabe 1 (pdf, 208 KB) Do 22.Okt.20 16:11	Keine Abgabe	Keine Korrektur			<input type="button" value="Browse..."/> No file selected.
Summe			Keine Abgabe	Keine Korrektur			

Bitte nur einmal auf [Upload] klicken, der Vorgang kann je nach Internetzugang einige Zeit dauern!



<mailto:matthias.demo@mustermann.demo>

Portal Revision 865, 2008-01-16 13:51:47 +0100 (Wed, 16 Jan 2008) by T. Nopper

# Demo: Anmeldung an Übungsportal (3/3)

- Examplarisch für Anmeldung des Studierenden Matthias Mustermann zur Veranstaltung "Betriebssysteme WS20/21"

## Übungsportal

Betriebssysteme WS20/21 Demo  
Gruppe 1.1 - Max Mustermann **Matthias Mustermann**  
Tutor: Tutor1 - Raum: Meetingraum Tutorat - Zeit: Mi 08:00-10:00

Blatt	Abgabetermin	Aufgaben	Ihre Abgaben	Korrekturen	Punkte		Neue Abgabe
					M.M.	M.M.	
1	Freitag, 13. NOV. 2020 17:00	Aufgabe 1 (pdf, 208 KB) Do 22.Okt.20 16:11	Keine Abgabe	Keine Korrektur			<input type="button" value="Browse..."/> No file selected. <input type="button" value="Upload"/>
Summe		Keine Aufgabe	Keine Abgabe	Keine Korrektur			

Bitte nur einmal auf [Upload] klicken, der Vorgang kann je nach Internetzugang einige Zeit dauern!

Zurücksetzen

mailto: matthias.demo@mustermann.demo

Portal Revision: 865, 2008-01-16 13:51:47 +0100 (Wed, 16 Jan 2008) by T. Nopper

# Prüfungsleistung / Studienleistung (1/2)

---

- Siehe auch “Merkliste und Fristen auf ilias” ...
- **Prüfungsleistung:**
  - Wird erbracht durch Bestehen der Klausur nach dem Semester.
    - Zeit und Ort werden noch bekanntgegeben, voraussichtlich 23.08.23, 14:00 h
  - Klausurvoraussetzung ist Anmeldung zur Prüfung in HISinOne
    - Anmeldezeitraum: 29.05. - 09.07.2023
    - Voraussetzung zur Prüfungsanmeldung ist Anmeldung zur Veranstaltung – jetzt! (bis 09.07.2023)
  - Sonst: Formal keine Voraussetzungen

# Prüfungsleistung / Studienleistung (2/2)

---

## ■ Studienleistung:

- Wird erbracht durch
  - Sinnvolles Bearbeiten von mindestens 75% der Aufgaben in den Übungsblättern
  - Regelmäßige, aktive Teilnahme an den Übungen
  - Vorrechnen mindestens einer Übungsaufgabe
- Voraussetzung ist Anmeldung zur Studienleistung in HISinOne
  - Anmeldezeitraum: 17.04. - 22.07.2023 (gesamter Vorlesungszeitraum)
  - Voraussetzung zur Anmeldung zur Studienleistung ist Anmeldung zur Übung in HISinOne – jetzt! (bis ~~09.07.2023~~, <sup>30.</sup> Anmeldung zu der globalen “Dummy-Gruppe”!
- Voraussetzung ist Anmeldung zur Übung im Übungsportal bis zum **24.04.23, 23:59 h.**
- Übungen sind wichtig zum Bestehen der Klausur!  
⇒ Studienleistung sollte **vor** der Prüfungsleistung erbracht werden, alles andere ist Unsinn!

# TODO Liste

---

## ■ Diese Woche:

- Anmeldung zur Vorlesung in HisInOne (jetzt!).
- Anmeldung zur Übung in HisInOne (zur globalen "Dummy-Gruppe", jetzt!).
- Anmeldung zur Übungsgruppe in Übungsportal (ab 16:30 h)
- Beitritt zum ILIAS-Kurs mit RZ-Account (Passwort: ti2023lect)



## ■ Während des Semesters:

- Erbringen der Studienleistung (Übung)
- Anmeldung zu
  - Prüfungsleistung (Klausur) **und**
  - Studienleistung

## ■ Ende des Semesters:

- Bestehen der Klausur.



# Kapitel 0 – Einführung

## **1. Erste Begriffsklärung, Vorlesungsplan**

2. Historie bis heute

Albert-Ludwigs-Universität Freiburg

Prof. Dr. Christoph Scholl

Institut für Informatik  
Sommersemester 2023

UNI  
FREIBURG

- Informatik beschäftigt sich mit Information, ihrer Analyse und Verarbeitung mittels Algorithmen.
- Algorithmen werden in der Regel als Software implementiert, die auf (programmierbarer) Hardware läuft.

- ... beschäftigt sich allgemein mit **Hardware**:
  - programmierbare Universalrechner
  - programmierbare eingebettete Rechner
  - Spezialhardware
- Dabei sind wichtig
  - theoretische Grundlagen zum Entwurf von Hardware,
  - die Methodik zum Entwurf von Hardware,
  - der Aufbau von Rechnern,
  - ...

# Beispiele

---

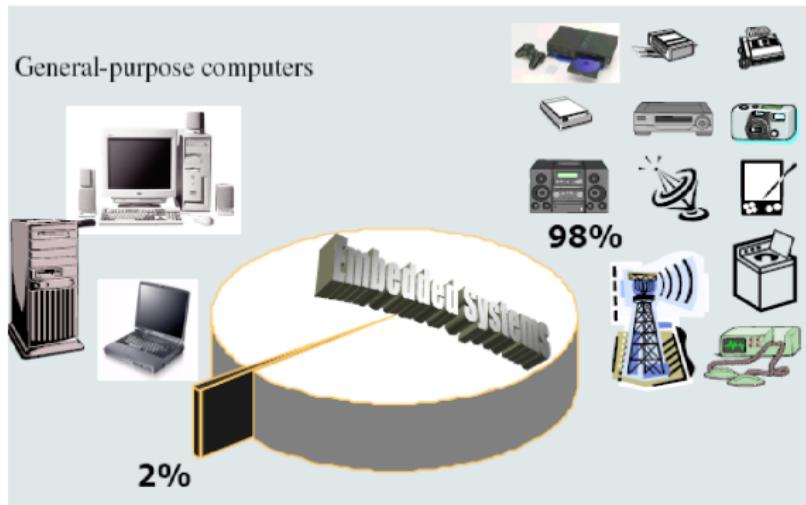
- Universalrechner (general purpose computer), PCs, Arbeitsplatzrechner, Server
- eingebettete Rechner in Autos, Flugzeugen, Waschmaschinen, Fernsehern, CD-Spielern, Spielkonsolen, Navigationsgeräte



# Marktanteile

Universalrechner  $\approx 2\%$

Eingebettete Rechner  $\approx 98\%$



# Verstehen / Entwerfen von (Rechen-)Systemen

---

- Wie versteht / entwirft man ein System, welches aus sehr vielen Komponenten besteht:
  - ... aus einem Rechner, der aus 3 Milliarden Transistoren besteht
  - ... aus darauf laufender Software (z.B. Betriebssystem, Übersetzer, Datenbanken, Kommunikationssystemen, Anwendungsprogrammen) mit Millionen Zeilen Programmcode
- Erkennen der **Struktur**:
  - Aufbau aus **Komponenten**
  - **Zusammenwirken** dieser Komponenten zur Realisierung ihrer Funktion

# Inhalt der Vorlesung

---

- Konzepte der Technischen Informatik und ...
- ihre Umsetzung mittels ...  
Entwurf und Analyse eines einfachen Rechners ReTI

# Überblick

---

Kap. 0 Einführung/Umfeld
5
4
3
2
1

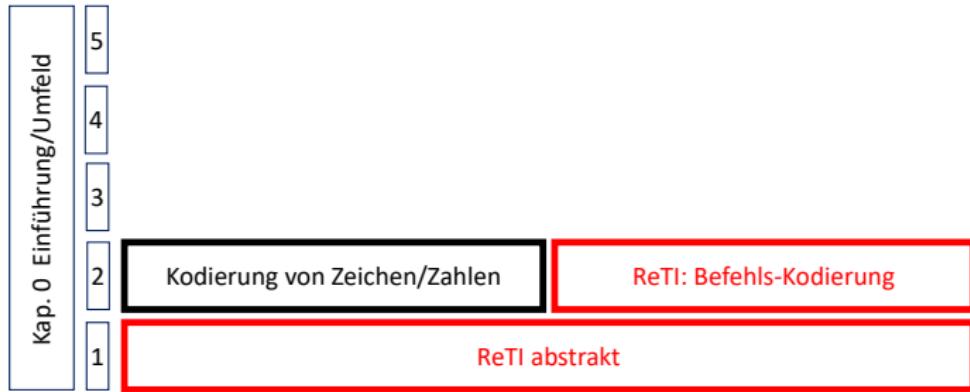
# Überblick

---



# Überblick

---



# Überblick

Kap. 0 Einführung/Umfeld	5	Physikalische Eigenschaften Timing	ReTi: Exaktes Timing
	4	<u>Sequentielle Logik</u>	ReTi: Datenpfade, Idealisiertes Timing
	3	<u>Komb.</u> Logik Minimalpolynome Arithmetik	ReTi: ALU
	2	Kodierung von Zeichen/Zahlen	ReTi: Befehls-Kodierung
	1	ReTi abstrakt	

# Literatur

---

- Jörg Keller, Wolfgang J. Paul  
„Hardware-Design:  
*Formaler Entwurf digitaler Schaltungen*“  
Teubner, 2005.
  
- Bernd Becker, Paul Molitor  
„Technische Informatik: Eine einführende Darstellung“  
Oldenbourg Wissenschaftsverlag München,  
2008, ISBN 978-3-486-58650-3, 435 Seiten.

# Ergänzende Literatur

---

- A. Tanenbaum  
*Structured Computer Organization.* Prentice Hall International, 2005.
- D. Patterson, J. Hennessy  
*Computer Organization and Design RISC-V Edition: The Hardware Software Interface.* (The Morgan Kaufmann Series in Computer Architecture and Design) 2017
- J. Hennessy, D. Patterson  
*Computer Architecture: A Quantitative Approach.* Morgan Kaufman Publishers, 2017.
- R. Drechsler, B. Becker  
~~*Graphenbasierte Funktionsdarstellungen.*~~ Teubner, 1998.
- P. Molitor, G. Scholl  
~~*Datenstrukturen und effiziente Algorithmen für die Logiksynthese kombinatorischer Schaltungen.*~~ Teubner, 1999.

*Rechner-  
architektur*

# Kapitel 0 – Einführung

1. Erste Begriffsklärung, Vorlesungsplan

## **2. Historie bis heute**

Albert-Ludwigs-Universität Freiburg

Prof. Dr. Christoph Scholl

Institut für Informatik

Sommersemester 2023

- Historische Entwicklung
  - von den Anfängen zum Multi-Core
  - Chips mit mehr als 100 Milliarden Transistoren
  - Probleme und Herausforderungen

# Wirtschaftliche Treiber der Computerentwicklung

---

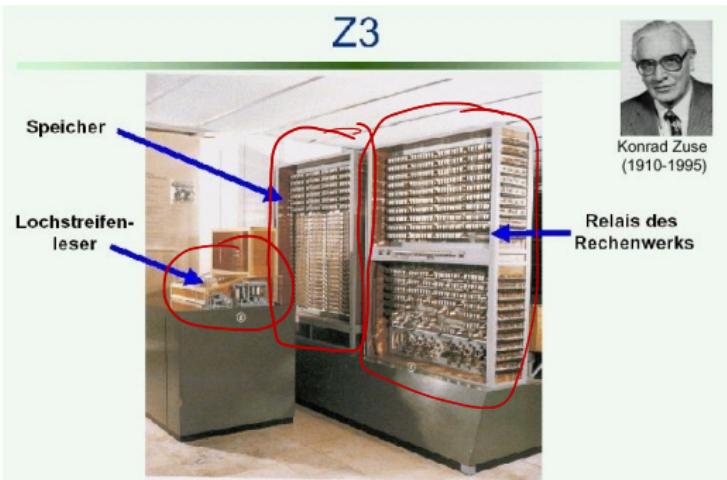
- **1930–60:** Wenige, sehr teure Rechner.
  - Wettervorhersagen, militärische Anwendungen
- **1960–80:** EDV für die Infrastruktur.
  - Banken, Verkehrs-, Energiesysteme
- **1980–2000:** Rechnerbasierte Konsumgüter.
  - PCs, Unterhaltungselektronik, Autos, Handys
- **Heute:** Ubiquitous Computing, Internet of Things (IoT), KI-Anwendungen, Maschinelles Lernen
  - Smartphones/Internet Devices, Autonome Fahrzeuge, Gebäudetechnik, Geräte zur medizinischen Überwachung, Assistenzsysteme, ...

# Technischer Fortschritt als Basis der Entwicklung

---

- Hardware-Fertigungstechnologie
  - Moore's Law: Verdoppelung der Anzahl Transistoren pro Flächeneinheit alle 18 Monate
- Neue Algorithmenklassen
  - Bilderkennung, schnelle Kommunikation, Data Mining / Suchmaschinen, Machine Learning
- Neue Entwurfsverfahren
  - Entwurf deutlich **komplexerer Systeme** in gleicher / unterproportional erhöhter **Zeit**

# Zuse Z1 / Z3 (1934–1941)



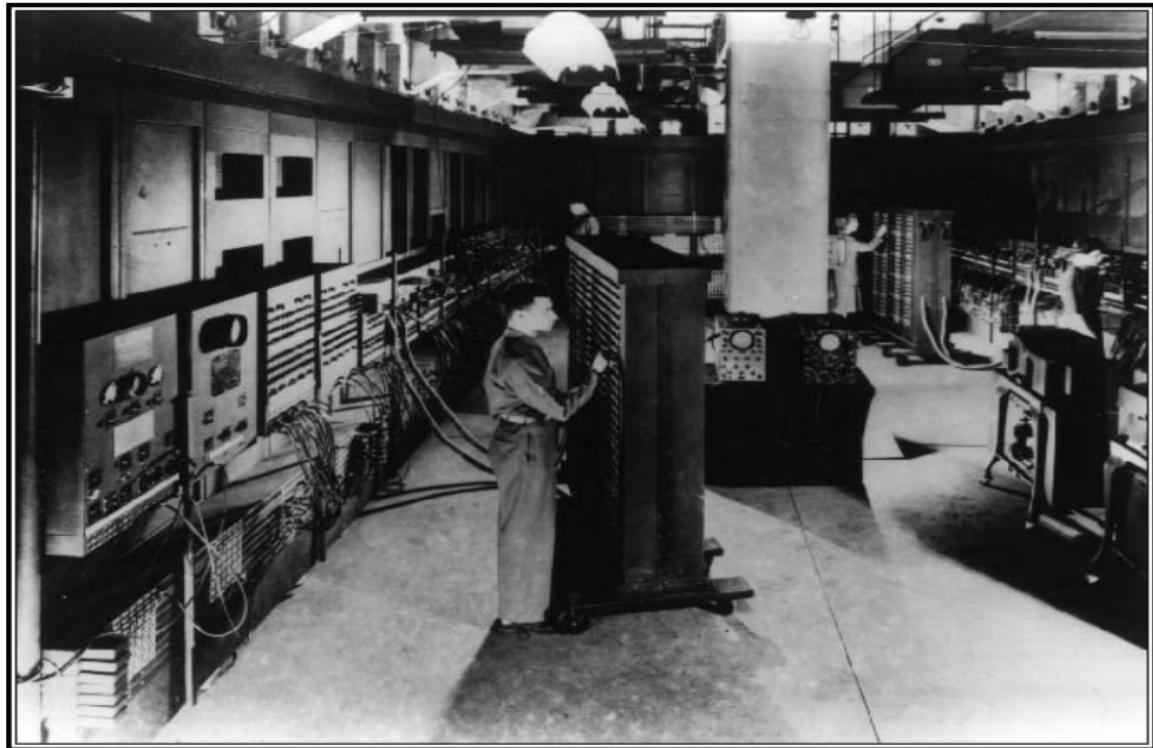
- Bis zu 10 Hz Taktfrequenz
- 64 Speicherzellen der Breite 22
- 4000 Watt Leistung
- Gewicht: 1000kg
- Programmierung über gelochte Filmstreifen

# Historische Entwicklung: ENIAC (1946)

---

- 30.000 kg schwer, 3 m hoch, 24 m breit
- 17.648 Vakuumröhren, 70.000 Widerstände, 1.500 Relais, 10.000 Kondensatoren
- 150.000 Watt Leistung
- Multiplikationszeit: 3 ms
- Programmierung über Schalttafeln

# Eniac



# Historische Entwicklung: Zitate

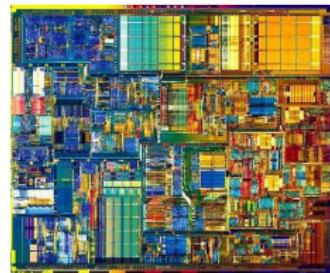
---

- Thomas Watson (IBM, 1943):  
*„I think there is a world market for maybe five computers.“*
- Popular Mechanics (1949):  
*„Computers in the future may weigh no more than 1.5 tons.“*
- Ken Olsen (DEC, 1977):  
*„There is no reason for any individual to have a computer in their home.“*

# Heutiger Stand: Mikroprozessoren

- 2000: Pentium 4

- 42 Millionen Transistoren
  - Taktfrequenz > 1,5 GHz



- Oktober 2022: Intel Core i9-13900K

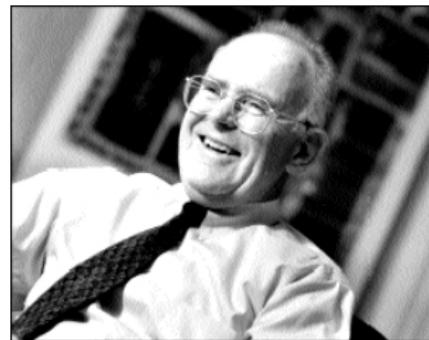
- $\approx 14,2$  (?) Milliarden Transistoren  
(vgl. Apple M1 Ultra: 114 Milliarden)
  - Taktfrequenz 3.0 GHz (5.8 GHz im Turbo-Mode)
  - „Multi-Core“ (24 Kerne)
    - Parallelausführung mehrerer Programme
    - Parallelausführung von Teilen desselben Programms



# Fortschritt der Halbleiterfertigung: Moore's Law



- Verdoppelung der Transistor-Dichte alle 18 Monate.  
(Gordon Moore, Mitbegründer von Intel, 1965)
- Kein Gesetz, sondern Voraussage, was technologisch möglich sein wird!

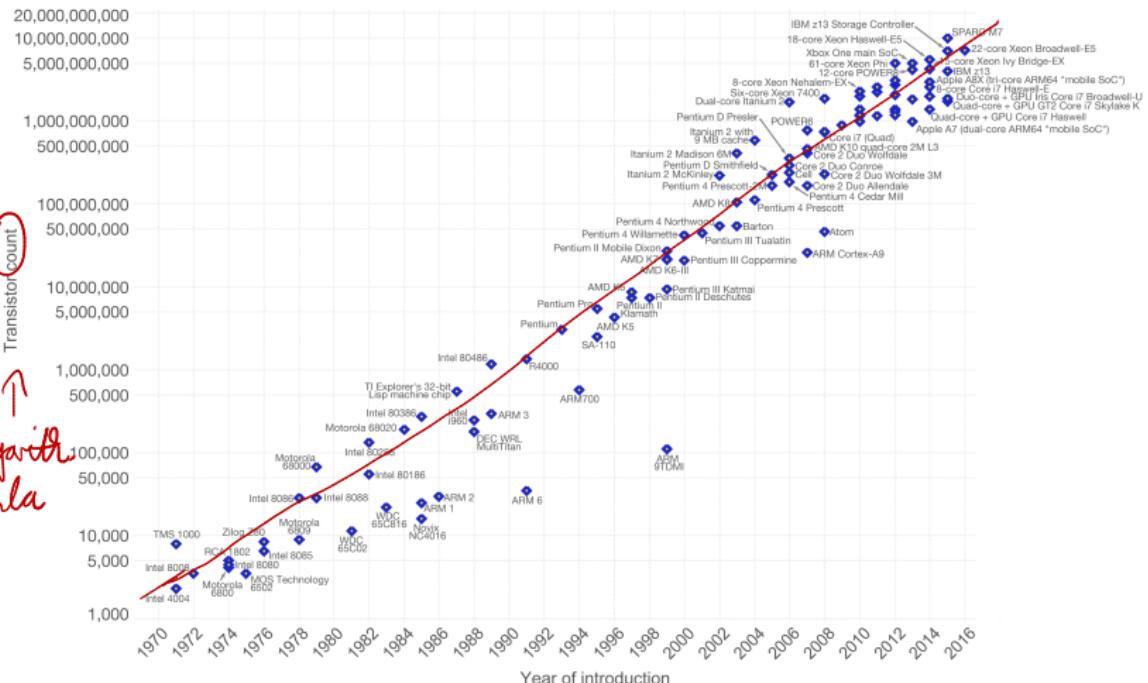


# Moore's Law

## Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

OurWorld  
in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.



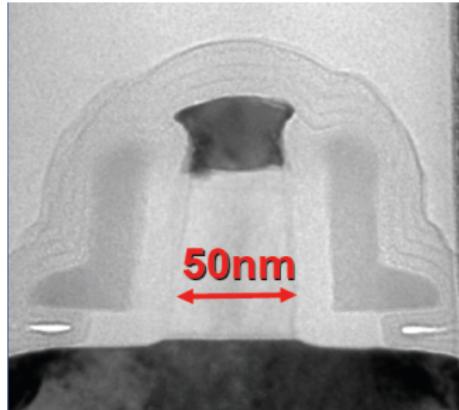
 ← Fläde A, Wahrscheinlichkeit für korrektes Funktionieren  $p$

 ← Fläde 2A, " " "  
"  $p^2$ "

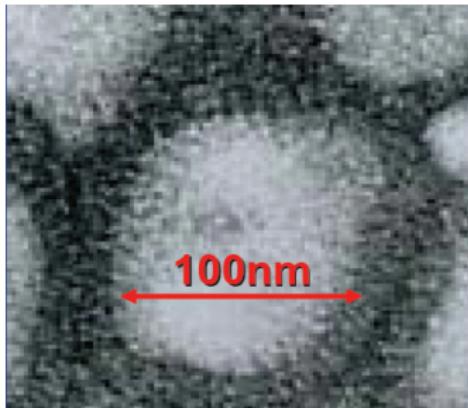


 ← Fläde  $n \cdot A$ , " " "  
"  $p^n$ "

# Transistor



Transistor



Grippevirus

- Inzwischen: Intel Core i9 mit 10-nm-Technologie, 7 nm in Arbeit.
- Aktuelle Apple-Prozessoren (M1, M2, A16 Bionic) mit 5-nm-Technologie
- 3-nm-Chips in Vorbereitung

# Halbleiterfertigung im Überblick

---

- Siliziumkristalle werden gezüchtet und in dünne runde Scheiben (Wafers) geschnitten.
- Transistoren werden auf einem Wafer gefertigt (Frontend).
  - Transistoren bilden Logikgatter
- Metallische Verbindungsleitungen (Interconnects) werden über den Transistoren gefertigt (Backend).
  - Interconnects verbinden Logikgatter miteinander und stellen Stromversorgung bereit.



# Einige Fakten über Halbleiterfertigung

---

- Eine Intel-Fab neuester Generation kostet mehr als 3 Milliarden US-Dollar.
- Über 500 Fertigungsschritte.



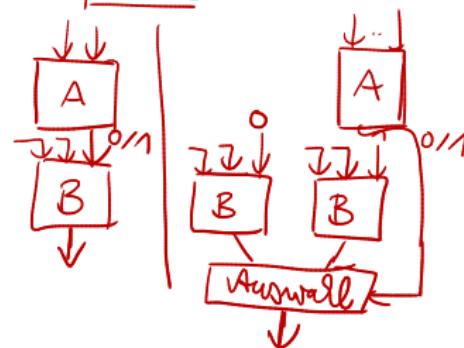
# Fortschritt bei Entwurfsverfahren

---

- Systeme bestehen aus **Hardware**, **Software** und gegebenenfalls **Zwischenschichten** (Betriebssystem und ähnliches).
- Für alle diese Bereiche gibt es effiziente **Entwurfsabläufe** („**Flows**“).
- Sie sind notwendig, um die Komplexität heutiger Systeme zu beherrschen und die Milliarden Transistoren auch zu nutzen.

# Vorgehen

- Es sind **Entwurfsziele** zu erreichen, die oft einander widersprechen und gegeneinander abgewogen werden müssen (Trade-Offs).
  - Systemabmessungen (physikalische Größe) Fläche
  - Geschwindigkeit
  - Energieverbrauch
  - Entwicklungszeit (Time to Market)
  - Kosten (Entwicklung / Fertigung)

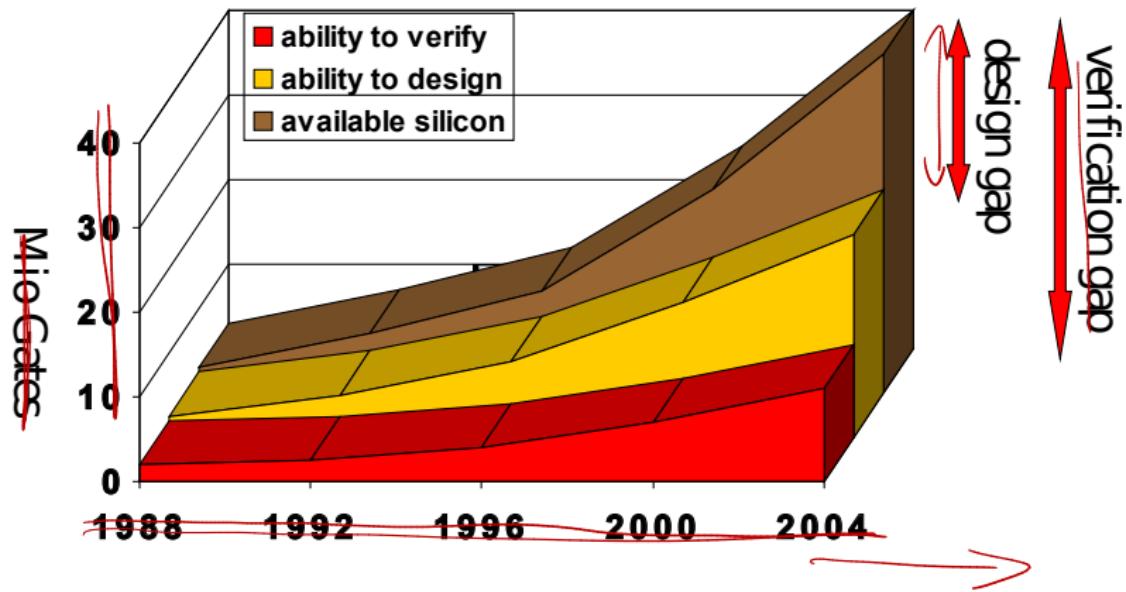


# Allgemeine Entwurfsprinzipien

---

- **Hierarchie:** Partitionierung des Systems in kleinere Blöcke; Entwurf der Blöcke und ihre spätere Zusammensetzung.
- **Entwurfsautomatisierung:** Verwende automatische Verfahren für möglichst viele Entwurfsschritte.
- **Wiederverwendung** von existierenden Designs (früher entworfen oder von Außen als „intellectual property core“ zugekauft).

# Design und Verification Gap



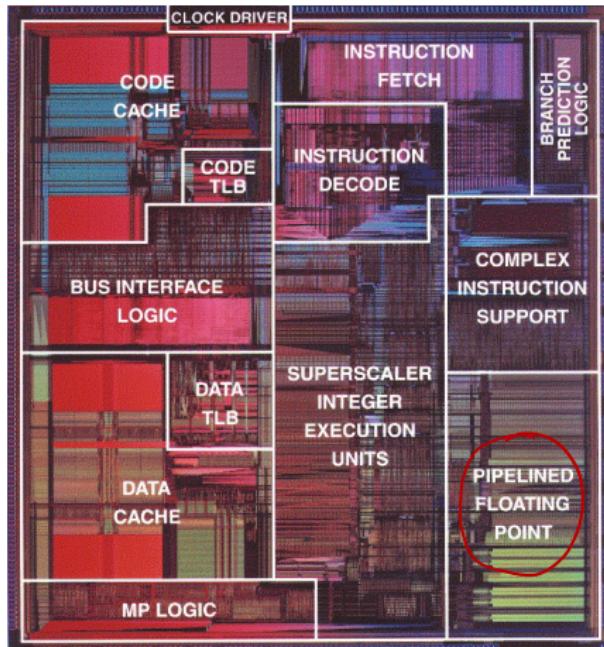
# Einige berühmte Beispiele von Entwurfsfehlern

---

- Pentium-Bug (*Fehler in der Hardware*).  
<http://www.intel.com/support/processors/pentium/fdiv/wp>
- Mars-Pathfinder-Mission (*Fehler auf Systemebene*).  
[http://research.microsoft.com/en-us/um/people/mbj/mars\\_pathfinder/authoritative\\_account.html](http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/authoritative_account.html)
- Ariane-Trägerraketen-Explosion (*Software der Vorgängerversion übernommen*).  
<http://www.ima.umn.edu/~arnold/disasters/ariane5rep.html>
- ...

# Pentium Bug (1/2)

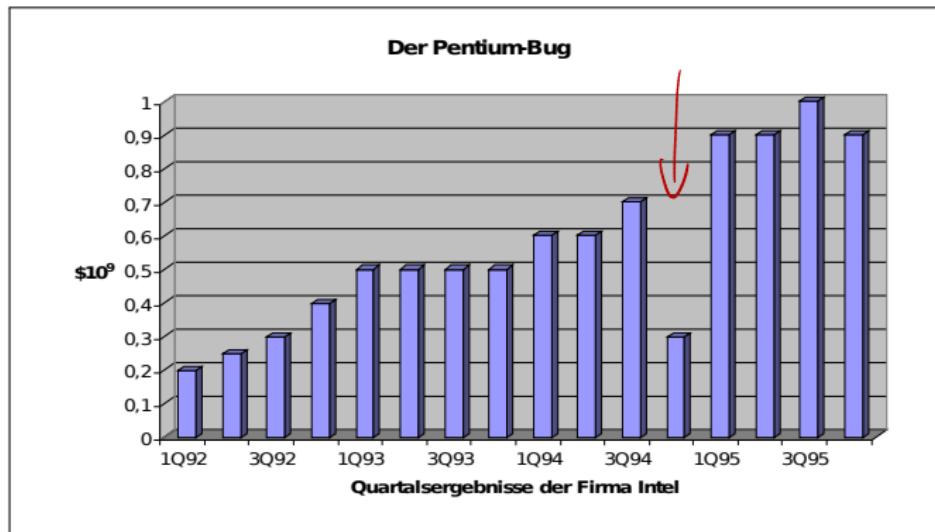
$$\begin{aligned}x &= \underline{4195835.0} \\y &= \underline{3145727.0} \\z &= x - \underline{(x/y)} \times y \\z &= 256.0!\end{aligned}$$



# Pentium Bug (2/2)

## ■ Fehler in Floating-Point-Einheit

- ⇒ Imageverlust für Intel
- ⇒ 480 Mio \$ zusätzliche Kosten für Intel im 4. Quartal 1994



# Herausforderungen (1/2)

---

## ■ Wie entwerfen? Entwurfsproblematik – „Design-Gap“

→ VLSI-Entwurf, Entwurfsmethodik

Anforderungen:

■ Entwurfszeit (Time to market)

■ Arbeitszeit der Designer

■ Laufzeit der Algorithmen

■ Energieverbrauch

■ Entwurfskorrektheit (siehe z.B. Pentium-Bug)

■ Effizienzanforderungen (Performanz, Energieverbrauch, Fläche)

## ■ Was tun mit all den Transistoren?

Sinnvolle Ausnutzung der verfügbaren Ressourcen

→ Architekturkonzepte, Rechnerarchitektur

# Herausforderungen (2/2)

---

## **VLSI-Entwurf, Entwurfsmethodik**

- Wird in dieser Vorlesung bei ReTI angedeutet
- Vorlesung Rechnerarchitektur, erster Teil
- Spezialvorlesungen
  - Testen
  - Verifikation
  - Eingebettete Systeme
  - ...
- Projekte, Praktika (z.B. FPGA-Design)

## **Architekturkonzepte, Rechnerarchitektur**

- Vorlesung Rechnerarchitektur, zweiter Teil

# Überblick TI

Kap. 0 Einführung/Umfeld		
5	Physikalische Eigenschaften Timing	ReTi: Exaktes Timing
4	Sequentielle Logik	ReTi: Datenpfade, Idealisiertes Timing
3	Komb. Logik Minimalpolynome Arithmetik	ReTi: ALU
2	Kodierung von Zeichen/Zahlen	ReTi: Befehls-Kodierung
1	<u>ReTi abstrakt</u>	

# Kapitel 1 – Grundlagen

## **Beispielrechner ReTI**

Albert-Ludwigs-Universität Freiburg

Prof. Dr. Christoph Scholl

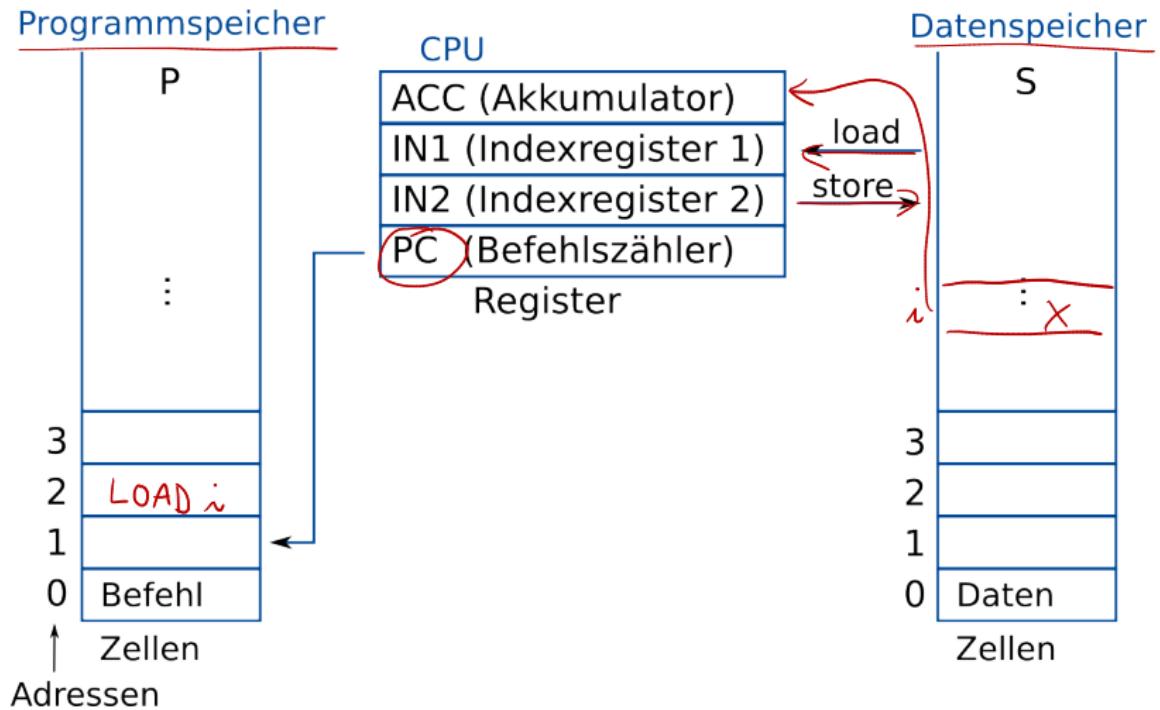
Institut für Informatik  
Sommersemester 2023

UNI  
FREIBURG

- Ursprünglich eingeführt in [Keller, Paul] unter dem Namen ReSa („Rechner Saarbrücken“).
- 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.
- 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  $t = 1, 2, \dots$   
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  $PC = 0$ .

# 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  $S$  und Werten in Registern
  - Mit Absolutwerten (Immediate).
- **Indexregister:** Indirekte Speicheradressierung (siehe unten).
- **Sprungbefehle:** Bedingte und unbedingte Sprünge.

# Load/Store

---

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

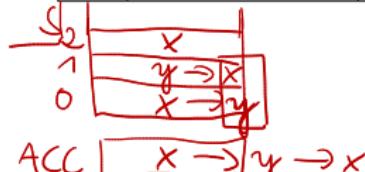
---

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

# Beispielprogramm

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

0	LOAD 0;	$ACC := S(0) = x$
1	STORE 2;	$S(2) := ACC = x$
2	LOAD 1;	$ACC := S(1) = y$ .
3	STORE 0;	$S(0) := ACC = y$
4	LOAD 2;	$ACC := S(2) = x$
5	STORE 1;	$S(1) := ACC = x$



# Compute-Befehle

Verknüpfe den Inhalt von ACC mit  $S(i)$  oder mit einer Konstante  $i$  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
<u>ADD i</u>	$ACC := ACC + S(i)$ $PC := PC + 1$
<u>SUB i</u>	$ACC := ACC - S(i)$ $PC := PC + 1$

# Immediate-Befehle

---

Interpretiere Parameter  $i$  direkt als Konstante.

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

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

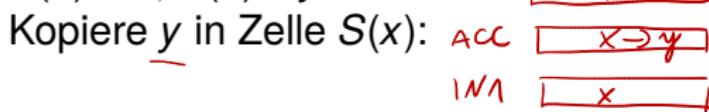
# Indexregister-Befehle

Befehl	Wirkung	
<u>LOADINj i</u> <i>LOADIN1</i> <i>LOADIN2</i>	$ACC := S(\underline{INj} + \underline{i})$ $(j \in \{1, 2\})$	$PC := PC + 1$
<u>STOREINj i</u>	$S(\underline{INj} + \underline{i}) := ACC$ $(j \in \{1, 2\})$	$PC := PC + 1$
<u>MOVE S D</u> <i>↑</i> <i>Copy</i>	$\underline{D} := \underline{S}$ $(D \in \{ACC, IN1, IN2\},$ $S \in \{ACC, IN1, IN2, PC\})$	<u><math>PC := PC + 1</math></u>
<u>MOVE S PC</u>	$\underline{PC} := \underline{S}$ $(S \in \{ACC, IN1, IN2\})$	<u><math>PC</math></u>

# Beispielprogramm für Indexregister-Befehle

$$S(0) = x, S(1) = y$$

Kopiere y in Zelle  $S(x)$ :



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

# Sprung-Befehle ← Relative Sprünge!

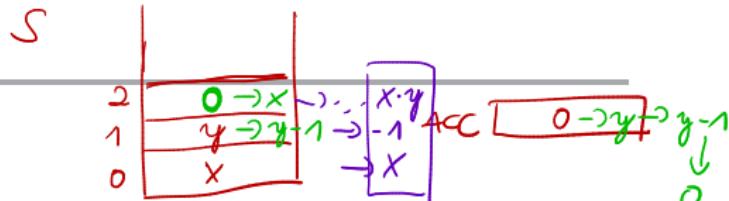
Manipulation des Befehlszählers.

- 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!  $\text{JUMP}_{\leq} i$  : es wird gesprungen, falls  $\text{ACC} \leq 0$ .

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

# Beispielprogramm

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



0	LOAD <u>I</u> 0;	$ACC := 0$	$\left. \begin{array}{l} S(2) = 0 \\ S(1) := S(1) - 1 \end{array} \right\}$
1	STORE 2;	$S(2) := 0$	
2	LOAD 1;	$ACC := S(1)$	
3	SUB <u>I</u> 1;	$ACC := ACC - 1$	$S(1) := S(1) - 1$
4	STORE 1;	$S(1) := ACC$	
5	JUMP < 5;	$PC := PC + 5$ , falls $ACC < 0$	$(ACC = S(1))$
6	LOAD 2;	$ACC := S(2)$	
7	ADD 0;	$ACC := ACC + S(0)$	$S(2) = S(2) + x$
8	STORE 2;	$S(2) := ACC$	
9	JUMP -7;	$PC := PC - 7$	
10	JUMP 0;	$PC := PC$	

# Zusammenfassung

---

- Rechner ReTI wird uns im weiteren Verlauf der Vorlesung als Illustrator und Anwendungsbeispiel für die vorgestellten Konzepte dienen.

# Kapitel 2 – Kodierung

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

Albert-Ludwigs-Universität Freiburg

Prof. Dr. Christoph Scholl

Institut für Informatik  
Sommersemester 2023

# Motivation

- Ein Rechner speichert, verarbeitet und produziert Informationen.
- Alle Ergebnisse müssen als Funktion der Anfangswerte exakt reproduzierbar sein.

→ Informationsspeicherung und Verarbeitung müssen exakt sein.



■ Probleme: Noise, Crosstalk, Abschwächung

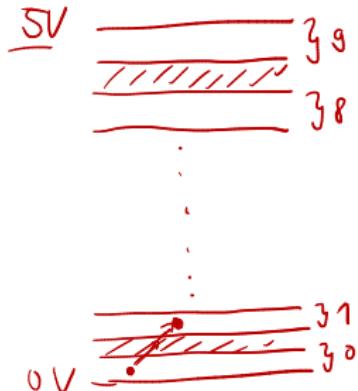
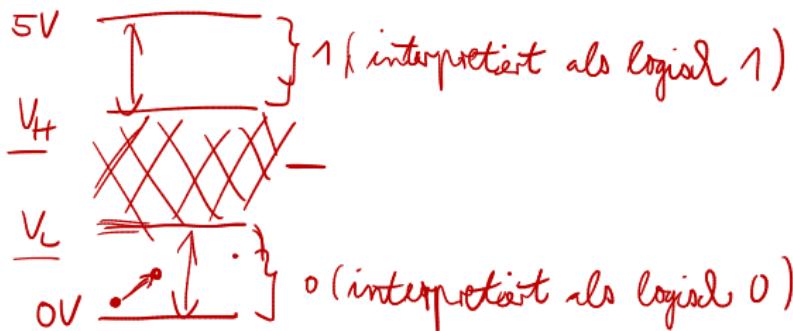
→ Es gibt keine exakte Datenübertragung oder Datenspeicherung.



→ 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)

# Alphabete und Wörter

## Definition

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

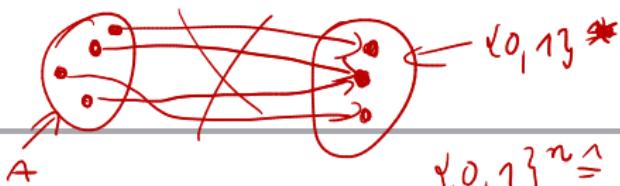
$a_1, \dots, a_m$  heißen **Zeichen** des Alphabets.

- $\underline{A^*} = \{w \mid w = b_1 \dots b_n \text{ mit } n \in \mathbb{N}, \forall i \text{ mit } 1 \leq i \leq n : b_i \in A\}$  ist die **Menge aller endlichen 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  $\underline{\varepsilon}$  bezeichnet.

Beispiel:

Sei  $A = \{a, b, c, d\}$ .  $A^* = \{\varepsilon, a, b, c, d, aa, ab, ac, ad, ba, \dots\}$   
Dann ist  $bcada$  ein Wort der Länge 5 über  $A$ .

# Code



$\{0,1\}^n \stackrel{\text{def}}{=} \text{Menge aller Wörter der Länge } n \text{ über Alphabet } \{0,1\}$

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

- Eine Abbildung  $c : A \rightarrow \{0,1\}^*$  oder  $c : A \rightarrow \{0,1\}^n$  heißt Code, falls  $c$  injektiv ist.  $\forall a_1, a_2 \in A : c(a_1) = c(a_2) \Rightarrow a_1 = a_2$
- 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.

$$\begin{aligned} n=3 \quad & \downarrow \{0,1\}^3 = \\ & \{000, 001, 010, \\ & 011, 100, \dots\} \end{aligned}$$

$2^n \geq |A| = m$  weil  $c$  injektiv sein muss.  
 $\Rightarrow n \geq \lceil \log_2 m \rceil \Rightarrow n \geq \lceil \log_2 m \rceil$ , da  $n$  eine natürliche Zahl ist.

# 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 (Extended Binary Coded Decimal Interchange Code): 8 Bits
  - Unicode: 16 Bits
- 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	0	0	0	1	1
0	1	0	1	0	1	0	1
<b>0000</b>	nul	dle	!	0	@	P	'
0001	soh	dc1	!	1	A	Q	a
0010	sfx	dc2	"	2	B	R	b
0011	etx	dc3	#	3	C	S	c
0100	<u>eot</u>	dc4	\$	4	D	T	d
0101	enq	nak	%	5	E	U	e
0110	<u>ack</u>	syn	&	6	F	V	f
<b>0111</b>	bel	etb	'	7	G	W	g
1000	bs	can	(	8	H	X	h
1001	ht	em	)	9	I	Y	i
1010	<u>if</u>	sub	*	:	J	Z	j
1011	vt	esc	+	;	K	[	k
1100	ff	fs	,	<	L	\	l
1101	<u>cr</u>	qs	-	=	M	]	m
1110	so	rs	.	>	N	^	n
1111	si	us	/	?	O	-	del

**letzte 4 Bits**

**Steuerzeichen**      **Schriftzeichen**

code('G') = 100 0111  
code(' ') = 010 0000

# 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

# 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	4	4

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

```
graph TD; a[20] --- b[25]; a --- c[15]; a --- d[8]; a --- e[7]; a --- f[6]; a --- g[5]; a --- h[5]; a --- i[5]; a --- j[4]; j --- k[9]; k --- l[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 --- a[a: 20]; Node1 --- b[b: 25]; Node2 --- c[c: 15]; Node2 --- d[d: 8]; Node1 --- Node3[10]; Node3 --- e[e: 7]; Node3 --- f[f: 6]; Node3 --- g[g: 5]; Node3 --- h[h: 5]; Node3 --- i[i: 5]; Node3 --- j[j: 4];
```

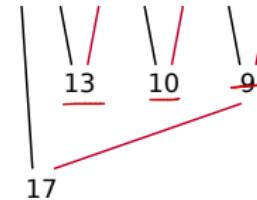
Baue binären Baum, indem die beiden kleinsten Häufigkeiten jeweils zu einem neuen Knoten addiert werden.

# 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



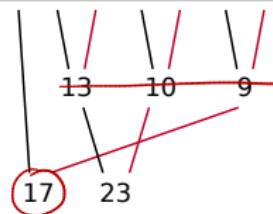
Baue binären Baum, indem die beiden kleinsten Häufigkeiten jeweils zu einem neuen Knoten addiert werden.

# 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.



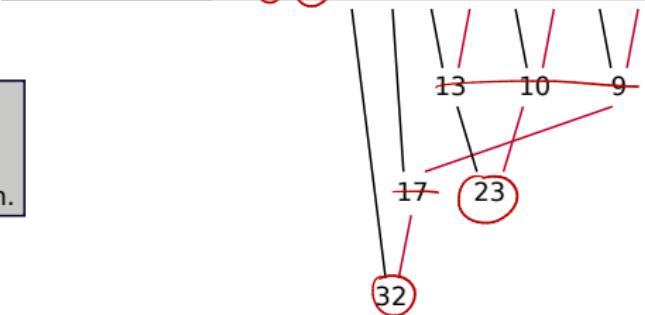
# Huffman-Code

- 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.

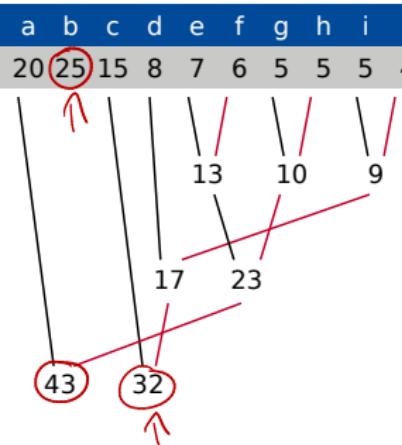


# Huffman-Code

- 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.

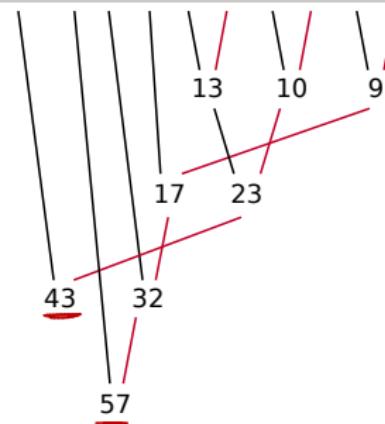


# Huffman-Code

- 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.

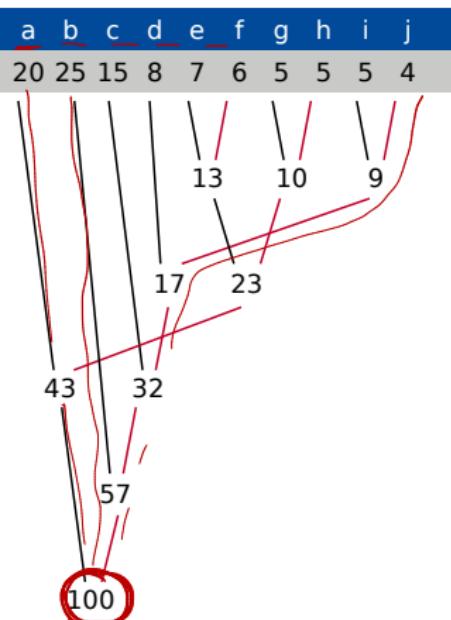


# Huffman-Code

- 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.



# Huffman-Code

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

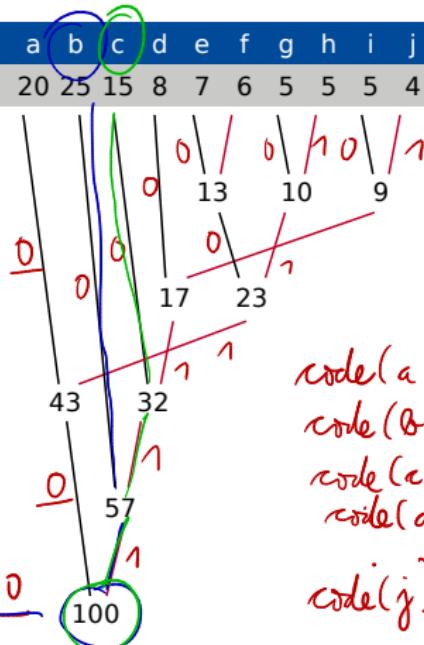
- Beispiel:

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!

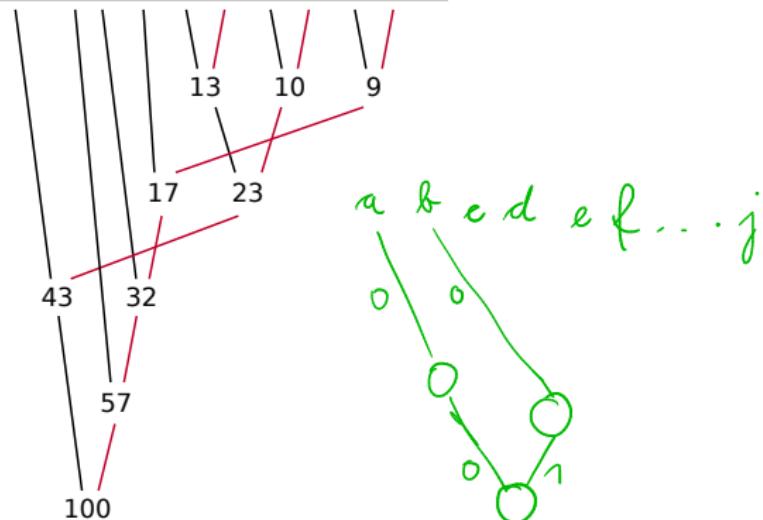
Kodierung zu b c d: 1011101110



$\text{code}(a) = \underline{\underline{00}}$   
 $\text{code}(b) = \underline{\underline{10}}$   
 $\text{code}(c) = \underline{\underline{110}}$   
 $\text{code}(d) = \underline{\underline{1110}}$   
 $\text{code}(j) = \underline{\underline{1111}}$

# Erzeugte Huffman-Kodierung

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



Erzeugte Kodierung:

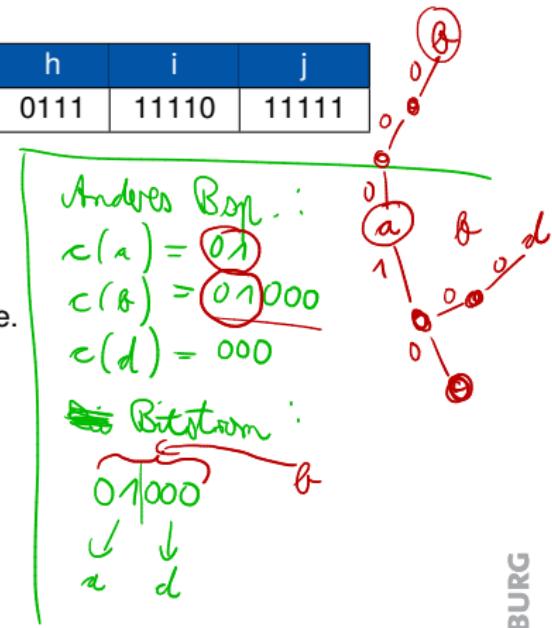
a	b	c	d	e	f	g	h	i	j
00	10	110	1110	0100	0101	0110	0111	11110	11111

# Huffman-Code: Dekodierung

Erzeugte Kodierung:

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

- 1 Lesen des Bitstromes bis ~~Symbol~~ erkannt wurde.  
*Ziehen*
- 2 Erkanntes ~~Symbol~~ ausgeben und weiter mit 1.  
*Ziehen*



# Präfixcodes

## Definition

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

- $\underline{a_1 \dots a_p} \in A^*$  heißt **Präfix** von  $\underline{b_1 \dots b_l} \in A^*$ , falls  $p \leq l$  und  $\underline{a_i = b_i} \forall i, 1 \leq i \leq p$ .  $A = \{a_1, \dots, a_m\}$
- 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 codierten Zeichen an den Blättern.)
  - Huffman-Code ist ein bzgl. mittlerer Codelänge optimaler Präfixcode (unter Voraussetzung einer bekannten Häufigkeitsverteilung) - ohne Beweis.

$$A = \{a_1, \dots, a_m\}$$

Käufigkeitsverteilung  $p$  mit  $\sum_{i=1}^m p(a_i) = 1$

Mittlere Codellänge von  $c: A \rightarrow \{0,1\}^*$ :

$$\sum_{i=1}^m p(a_i) \cdot |c(a_i)|$$

# Beispiel Präfixcodes

---

Frage: Welche dieser Codes sind Präfixcodes

- a.  $c('A') = \underline{01}$ ,  $c('B') = 110$ ,  $c('C') = \underline{011}$  *nun!*
- b.  $c('A') = \underline{01}$ ,  $c('B') = \underline{110}$ ,  $c('C') = \underline{111}$  *ja!*
- c.  $c('1') = xz$ ,  $c('2') = xy$ ,  $c('3') = yz$  *ist überhaupt kein Code!*
- d. ~~Keiner der Obigen.~~

# Weitere Verfahren

---

- Es gibt zahlreiche Ansätze zur **Datenkompression**.  
(Beispiel: Lempel-Ziv-Welch.) *Folgen von Zeichen mit Endewörtern kodiert!*
- 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

Prof. Dr. Christoph Scholl

Institut für Informatik  
Sommersemester 2023

## Definition

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

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

# Beispiele für Zahlensysteme

---

## Dualsystem bzw. Binärsystem:

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

## Oktalsystem:

$$\underline{b = 8} \quad Z = \{0, 1, 2, 3, 4, 5, 6, 7\} \quad d(0) = 0, \dots, \quad d(7) = 7$$

## Dezimalsystem:

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

## Hexadezimalsystem:

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

$$d(0) = 0, \dots, \quad d(9) = 9, \quad d(A) = 10, \quad d(B) = 11, \quad d(C) = 12, \\ d(D) = 13, \quad d(E) = 14, \quad d(F) = 15$$

# Festkommazahlen

## 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.
- Der Wert  $\langle d \rangle$  einer nicht-negativen Festkommazahl  $d = d_n d_{n-1} \dots d_1 d_0 d_{-1} \dots d_{-k}$  mit  $d_i \in Z$  ist gegeben durch

$$\underbrace{d_n d_{n-1} \dots d_1 d_0}_{\text{Vorkommastellen}} \underbrace{d_{-1} \dots d_{-k}}_{\text{Nachkommastellen}} \langle d \rangle = \sum_{i=-k}^n b^i \cdot \delta(d_i)$$

Bsp.: Dezimalsystem,  $2 = 2$ ,  $n = 2$

$$\langle 935.76 \rangle = 10^{-2} \cdot 6 + 10^{-1} \cdot 7 + 5 \cdot 10^0 + 3 \cdot 10^1 + 9 \cdot 10^2 = \\ 935,76$$

$d_2 \uparrow d_1 \uparrow d_0 \uparrow d_{-1} \uparrow d_{-2}$

# Festkommazahlen: Schreibweise

---

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

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

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

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

$$\langle 0110_2 \rangle = 6 = 0 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3$$

$$\langle 0110_8 \rangle = 72 = 0 \cdot 8^0 + 1 \cdot 8^1 + 1 \cdot 8^2 + 0 \cdot 8^3$$

$$\langle 0110_{10} \rangle = 110 = \dots$$

$$\langle 0110_{16} \rangle = 272 = 0 \cdot 16^0 + 1 \cdot 16^1 + 1 \cdot 16^2 + 0 \cdot 16^3$$

# Negative Festkommazahlen

Im Folgenden werden Basis 2 und 0 Nachkommastellen 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.

$d_n = 1 \rightsquigarrow$  negative Zahl oder 0

- Bei der Darstellung negativer Zahlen gibt es folgende Alternativen:

- Darstellung durch Betrag und Vorzeichen:

$$(d_n, d_{n-1}, \dots, d_0)_{BV} := (-1)^{d_n} \sum_{i=0}^{n-1} d_i 2^i \quad \begin{array}{l} 0 \triangleq + \\ 1 \triangleq - \end{array}$$

- Einer-Komplement-Darstellung:

$$[d_n, d_{n-1}, \dots, d_0]_1 := \sum_{i=0}^{n-1} d_i 2^i - d_n (2^n - 1)$$

- Zweier-Komplement-Darstellung:

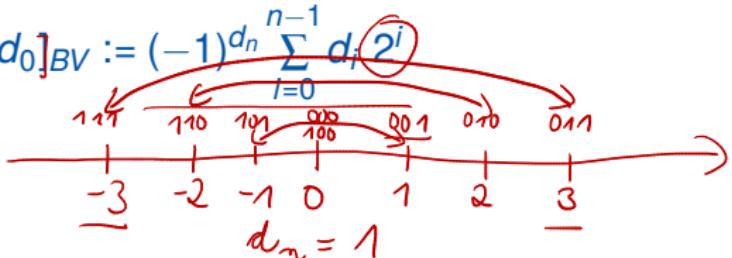
$$[d_n, d_{n-1}, \dots, d_0]_2 := \sum_{i=0}^{n-1} d_i 2^i - d_n 2^n$$

# Betrag und Vorzeichen

$$[d_n, d_{n-1}, \dots, d_0]_{BV} := (-1)^{d_n} \sum_{i=0}^{n-1} d_i 2^i$$

Beispiel:  $n = 2, \underline{z} = 0$

$$d_n = 0$$



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

- Der Zahlenbereich ist **symmetrisch**:
  - Kleinste Zahl:  $-(2^n - 1)$ , größte Zahl:  $2^n - 1$
- Man erhält zu  $d$  die **inverse** Zahl, indem man das erste Bit komplementiert.
- Zwei Darstellungen für die **Null** (000 und 100 im Beispiel).

größte Zahl:  $d_m = 0, d_{m-1} = \dots = d_0 = 1$

$$\langle d \rangle = (-1)^{d_m} \cdot \sum_{i=0}^{n-1} d_i \cdot 2^i = \sum_{i=0}^{n-1} 2^i = \frac{2^n - 1}{2 - 1} = \underline{\underline{2^n - 1}}$$

geom. Summenformel:

$$\text{Für } R \neq 1: \sum_{i=0}^{n-1} R^i = \frac{R^n - 1}{R - 1}$$

$$\begin{array}{c} n-1 & 0 \\ \downarrow & \downarrow \\ \langle 11\dots11 \rangle = \underline{\underline{2^n - 1}} \end{array}$$

$$\begin{array}{r} + \begin{array}{cccc} 1 & 1 & \dots & 1 \end{array} \\ \hline \langle 100\dots00 \rangle = 2^n \\ \uparrow \quad \uparrow \\ n \quad 0 \end{array}$$

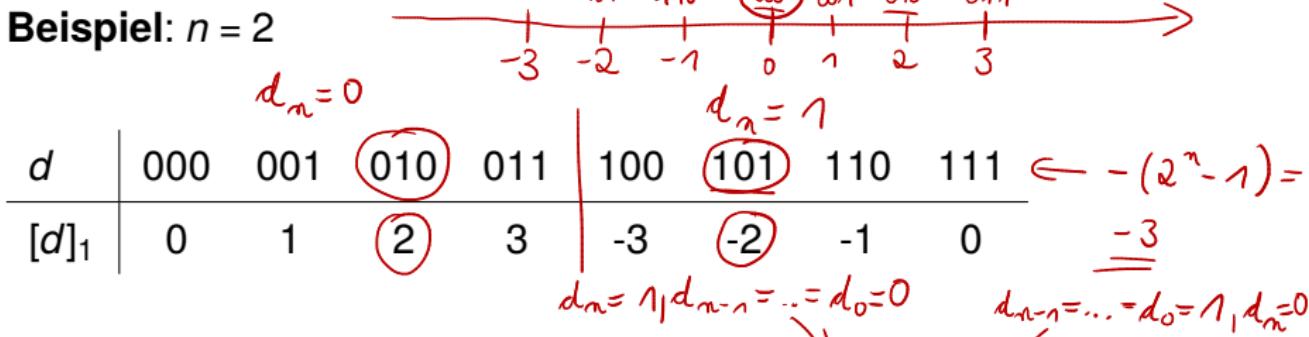
kleinste Zahl: entgegengesetzt:  $d_m = 1, d_{m-1} = 1, \dots, d_0 = 1$

$$\langle d \rangle = -(2^n - 1)$$

# Einer-Komplement

$$[d_n, d_{n-1}, \dots, d_0]_1 := \sum_{i=0}^{n-1} d_i 2^i - d_n (2^n - 1)$$

Beispiel:  $n = 2$



- Der Zahlenbereich ist **symmetrisch**:  $-(2^n - 1) \dots 2^n - 1$
- Zwei Darstellungen für die **Null** (000 und 111 im Beispiel).
- Man erhält zu  $d$  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  $[a']_1 = -[a]_1$ .  $\Leftrightarrow [a']_1 + [a]_1 = 0$

$$\text{B.s.: } \underline{[a']_1} + \underline{[a]_1} = 0$$

Beweis: ...

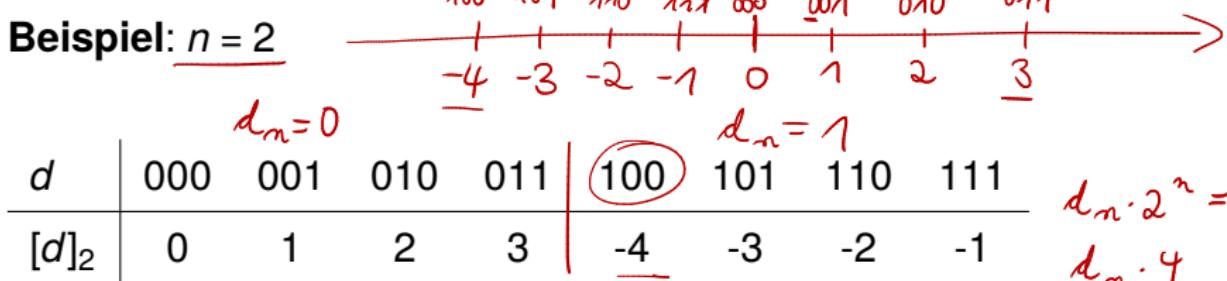
$$\begin{aligned} & \sum_{i=0}^{n-1} \underline{a'_i} \cdot 2^i - \underline{a'_m} (2^n - 1) + \sum_{i=0}^{n-1} \underline{a_i} \cdot 2^i - \underline{a_m} (2^n - 1) = \\ & > \sum_{i=0}^{n-1} (\underline{a'_i} + \underline{a_i}) \cdot 2^i - (\underline{a'_m} + \underline{a_m}) \cdot (2^n - 1) \\ & = \sum_{i=0}^{n-1} 2^i - (2^n - 1) = (2^n - 1) - (2^n - 1) = 0 \end{aligned}$$

$$\begin{array}{c|c} a_i & a'_i \\ \hline 0 & 1 \\ 1 & 0 \end{array} \Rightarrow \underline{a_i + a'_i = 1}$$

# Zweier-Komplement

$$[d_n, d_{n-1}, \dots, d_0]_2 := \sum_{i=0}^{n-1} d_i 2^i - d_n 2^n$$

Beispiel:  $n = 2$



$$d_n \cdot 2^n =$$

$$d_n \cdot 4$$

$$d_n = 1, d_{n-1} = \dots = d_0 = 0$$

$$d_n = 0, d_{n-1} = \dots = d_0 = 1$$

- Der Zahlenbereich ist **asymmetrisch**:  $\underline{-2^n \dots 2^n - 1}$
- Die Zahlendarstellung ist eindeutig, auch für die Null.
- Man erhält zu  $d$  die **inverse Zahl**, indem man alle Bits komplementiert und an der niedrigwertigsten Stelle 1 addiert (siehe Lemma nächste Folie).

# Zweier-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  $[a']_2 + 1 = -[a]_2$ .  $\Leftrightarrow [a]_2 + [a']_2 + 1 = 0$

Bsp.:  $n=0, n=3$

## Beweis: Übung

$$a = \underline{0011} \quad \begin{array}{r} 1100 \\ + 1 \\ \hline 1101 \end{array}$$

$$[a]_2 = \underline{1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2} - 0 \cdot 2^3 = 3 \quad \left. \begin{array}{l} \\ \\ \end{array} \right\} [a']_2 + [a]_2 + 1 =$$

$$a' = \underline{1100}$$

$$[a']_2 = 0 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 - 1 \cdot 2^3 = -4 \quad \left. \begin{array}{l} \\ \\ \end{array} \right\} \underline{\underline{0}}$$

$$[1101]_2 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 - 1 \cdot 2^3 = -3$$

# Vorteil von Zweier-Komplement

---

- 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

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

$$:= (-1)^{d_n} \sum_{i=0}^{n-1} d_i 2^i$$

$n = 2$

$a$	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
	<i>BV</i>		<i>E, Z,</i>		<i>Z, Z.</i>			
	<u>symmetrisch</u>		<u>symmetrisch</u>		<u>asymmetrisch</u>			
<u>kleinste Zahl</u>	<u><math>-(2^n - 1)</math></u>		<u><math>-(2^n - 1)</math></u>		<u><math>-2^n</math></u>			
<u>größte Zahl</u>	<u><math>2^n - 1</math></u>		<u><math>2^n - 1</math></u>		<u><math>2^n - 1</math></u>			
<u>Inverses durch</u>	<u>kompl. 1. Bit</u>		<u>kompl. alle Bits</u>		<u>kompl. alle Bits, add. 1</u>			
<u>Null</u>	<u>2 Darstellungen</u>		<u>2 Darstellungen</u>		<u>1 Darstellung</u>			

# Beispiel – Festkommazahlen

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

- a.  $\underline{[1001]}_2 = \underline{[0111]}_{BV}$  f
- b.  $\underline{[1001]}_2 - \underline{[1001]}_2 = \underline{[10010]}_2$  f
- c.  $[1001]_2$  ist das (additive) Inverse zu  $\underline{[0111]}_2$  ✓
- d.  $\underline{[1001]}_2 = \underline{[0000]}_1$  ✓

$$\begin{array}{r} 0110 \\ + 1 \\ \hline 0111 \end{array}$$

$-2^3 + 2^0 = -7$        $1(-2^3 + 1) = -7$

$-2^n$        $+1$

$(-2^n + 1)$        $+0$

# Probleme von Festkommazahlen

- Betrachte (positive) Festkommazahlen mit  $n$  Vor- und  $k$  Nachkommastellen, Vorzeichenbit  $d_n = 0$ .
    - Keine ganz großen bzw. kleinen Zahlen darstellbar!
      - Größte Zahl:  $2^n - 2^{-k}$
      - Kleinste Zahl:  $2^{-k}$
  - Operationen sind nicht abgeschlossen!
    - $2^{n-1} + 2^{n-1}$  ist nicht darstellbar, obwohl die Operanden darstellbar sind.
  - Rechengesetze wie Assoziativgesetz gelten nicht, da bei ihrer Anwendung evtl. der darstellbare Zahlenbereich verlassen wird!

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

# Gleitkomma-Zahlen

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

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

$$\frac{2^E}{2^{E-1}} \sim 2^{E+1}$$

$$\frac{2^E}{2^E} \sim 2^{E-1}$$

- 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								

- 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

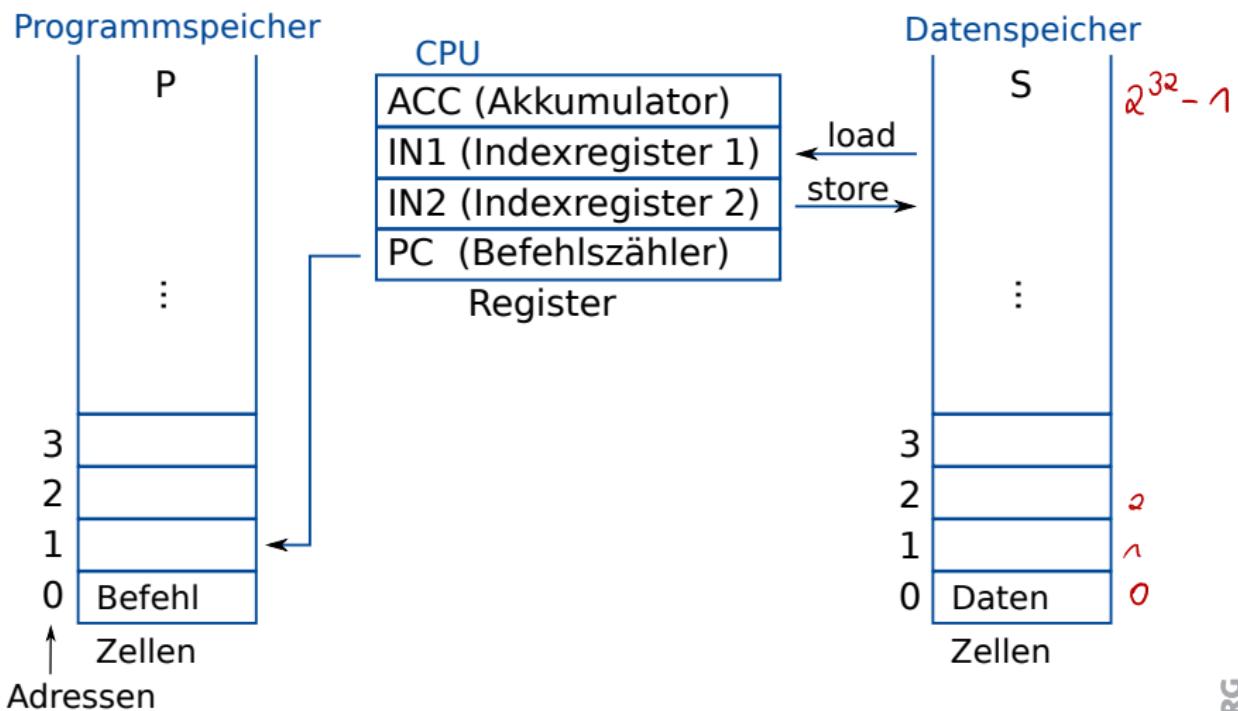
1. Kodierung von Zeichen
2. Kodierung von Zahlen
- 3. Anwendung: ReTI**

Albert-Ludwigs-Universität Freiburg

Prof. Dr. Christoph Scholl

Institut für Informatik  
Sommersemester 2023

# Realisierung von ReTI



# Unterschiede abstrakter/realer ReTI

---

- 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  $M(i)$  Inhalt der  $i$ -ten Speicherzelle.
  - Speicherzellen können Elemente aus  $\{0, 1\}^{32} = \mathbb{B}^{32}$  aufnehmen.
- CPU-Register  $PC, ACC, IN1$  und  $IN2$  können nur Elemente  $w \in \mathbb{B}^{32}$  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

---

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

- $\langle A \rangle := B$

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

bedeutet  $A := \text{bin}_{32}(B)$  ← Binärcodierung der Zahl B

- Beispiel:  $\langle PC \rangle := \langle PC \rangle + 1$

$$PC := \text{bin}_{32}(\langle PC \rangle + 1)$$

- $[A] := B$

(A Register oder Speicherzelle,  $B \in \{-2^{31}, \dots, 2^{31} - 1\}$ )

bedeutet  $A := \text{twoc}_{32}(B)$  ← Zweierkomplementdarstellung von B  
B wird als Zweierkomplement-Zahl interpretiert.

# Befehlsformate

- 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  $I = i_{31}, \dots, i_0 \in \mathbb{B}^{32}$ .  
 $\underline{I[y, x]} := \underline{i_y, i_{y-1}, \dots, i_x}$  für  $0 \leq x \leq y \leq 31$ .
- Allgemeines Instruktionsformat:

31	30	29	...	24	23	...	0			
Typ	Spezifikation				Parameter i					
2		6		24						

# Typ einer Instruktion

---

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

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

# Load-Befehle: Kodierungsprinzip

---

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

The diagram illustrates the memory layout for load instructions. It shows a sequence of addresses from 31 down to 0. Addresses 29 through 24 are circled in red, indicating they represent the memory location of the operand. Address 25 is also circled in red. Address 28 is marked with an asterisk (\*), which is typically used for immediate values or memory addresses. Address 27 is marked with 'D', likely referring to the destination register. Address 26 is marked with 'M', indicating it is the Modulus field. Address 23 is marked with 'i', likely representing the index or base register. Red wavy lines connect the circled addresses, grouping them together.

- M: Modus
- D: Vorerst irrelevant

# Load-Befehle: Kodierung

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

# Store-, Move-Befehle: Prinzip

I [31:0]

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

■ M: Modus I[28], I[28]  
■ S: Source I[27], I[26]  
■ D: Destination I[25], I[24]

Kodierung S, D

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

# Store-, Move-Befehle: Kodierung

Typ	Modus	Befehl	Wirkung	
1 0	<u>0 0</u>	<u>STORE</u> <u><i>i</i></u>	$M(\underline{i}) := \underline{ACC}$	$\langle PC \rangle := \langle PC \rangle + 1$
1 0	<u>0 1</u>	<u>STOREIN1</u> <u><i>i</i></u>	$M(\langle IN1 \rangle + [i]) := \underline{ACC}$	$\langle PC \rangle := \langle PC \rangle + 1$
1 0	<u>1 0</u>	<u>STOREIN2</u> <u><i>i</i></u>	$M(\underline{IN2} + [i]) := \underline{ACC}$	$\langle PC \rangle := \langle PC \rangle + 1$
1 0	<u>1 1</u>	<u>MOVE</u> <u><i>S D</i></u> 	<u><i>D</i> := <i>S</i></u>	$\langle PC \rangle := \langle PC \rangle + 1$ (außer bei $D = 00$ ( $PC$ ))

MOVE ACC IN1  
11    01

# Compute-Befehle: Prinzip

---

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

- MI: „compute **m**emory”/„compute **i**mmediate“
- F: Funktionsfeld
- D: Vorerst irrelevant

# Compute-Befehle: Kodierung

Typ	MI	F	Befehl	Wirkung	
0 0	0	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>	$ACC := ACC \wedge \underline{0^8 i}$	$\langle PC \rangle := \langle PC \rangle + 1$
0 0	1	0 1 0	SUB <i>i</i>	$[ACC] := [ACC] - [M(\langle i \rangle)]$	$\langle PC \rangle := \langle PC \rangle + 1$
		0 1 1	ADD <i>i</i>	$[ACC] := [ACC] + [M(\langle i \rangle)]$	$\langle PC \rangle := \langle PC \rangle + 1$
		1 0 0	OPLUS <i>i</i>	$ACC := ACC \oplus M(\langle i \rangle)$	$\langle PC \rangle := \langle PC \rangle + 1$
		1 0 1	OR <i>i</i>	$ACC := ACC \vee M(\langle i \rangle)$	$\langle PC \rangle := \langle PC \rangle + 1$
		1 1 0	AND <i>i</i>	$ACC := ACC \wedge M(\langle i \rangle)$	$\langle PC \rangle := \langle PC \rangle + 1$

# Bitstring-Operationen

$\wedge$ : Und-Operation

$a$	$b$	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

$\vee$ : Oder-Operation

$a$	$b$	$a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

$\oplus$ : Exklusiv-Oder-Operation  
*oder*

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

■ Beispiel OPLUS:

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

$$ACC_{31} \dots ACC_0 \quad 0 \dots 0 \quad i_{23} \dots i_0 \quad ACC_0 \oplus i_0$$

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

$$ACC_{31}$$

$$ACC_{24}$$

# Sprungbefehle: Prinzip

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

- C: Condition

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

"<" oder "=" oder ">"

JUMP<sub>c</sub>

Vergleichsvektion per Def.  
immer falsch  $\rightarrow$  kein Sprung

NOP

"no operation"

Vergleichsvektion per Def. immer  
wahr  $\sim$  Sprung  
auf Fall  $\rightarrow$  unbedingte 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  $I[27] = 1 \Leftrightarrow >$  wird abgefragt

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

# Sprungbefehle: Kodierung

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

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

# 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, vgl. Kapitel 1.

# Zusammenfassung

---

- Zusammenfassung des Befehlssatzes der ReTI auf der nächsten Folie ...

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 $D_i$	$D := 0^8i$

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	SUB $D_i$	$[D] := [D] - [i]$
000011	ADD $D_i$	$[D] := [D] + [i]$
000100	OPLUS $D_i$	$D := D \oplus 0^8i$ $\langle PC \rangle := \langle PC \rangle + 1$ falls $D \neq PC$
000101	ORI $D_i$	$D := D \vee 0^8i$
000110	ANDI $D_i$	$D := D \wedge 0^8i$
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 $\geq i$	
11011	JUMP $< i$	
11100	JUMP $\neq i$	
11110	JUMP $\leq i$	
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	$ACC$

## Hinweis zu den LOAD- und Compute-Befehlen

hier in verallgemeinerter Form:  
Zielregister steht in  $I[25 : 24]$

## Beispiel – Programme

Ein ReTI-Programm  $P$  sieht folgendermaßen aus:



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

- a. Der PC ist nach Ausführung des Programms um 3 erhöht. (f)
  - 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. f



# Kapitel 3 – Kombinatorische Logik

## 1. Kombinatorische Schaltkreise

1.1 Gatter, Transistoren

1.2 Definition

2. Boolesche Algebren

3. Boolesche Ausdrücke, Normalformen, zweistufige Synthese

4. Berechnung eines Minimalpolynoms

5. Arithmetische Schaltungen

6. Anwendung: ALU von ReTI

Albert-Ludwigs-Universität Freiburg

Prof. Dr. Christoph Scholl

Institut für Informatik

Sommersemester 2023

# Boolesche Funktionen

## Definition

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

$$\mathbb{B} = \{0, 1\}$$

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

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

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

- Wir schreiben abkürzend  $\underline{\mathbb{B}_n}$  statt  $\underline{\mathbb{B}_{n,1}}$ .

# Kombinatorische Schaltkreise

---

- Ein **kombinatorischer Schaltkreis** (Schaltnetz) ist ein Modell für Hardware, die eine boolesche Funktion  $f : \mathbb{B}^n \rightarrow \mathbb{B}^m$  ( $n, m \in \mathbb{N}$ ) implementiert.
- Ein kombinatorischer Schaltkreis für  $f$  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.
- 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 sogenannte zweistufigen Realisierungen, siehe später (Kapitel 3.3).
  - 2 Mehrstufige Realisierungen mit allgemeinen Bibliothekszellen (Logik-Gattern).

# Logikgatter

---

- 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 Gatter ergibt eine Standardzellen-Bibliothek bzw. Gatter-Bibliothek.

# Einige wichtige Gatter

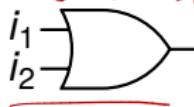
$i_1$	$i_2$	$AND_2$
0	0	0
0	1	0
1	0	0
1	1	1

$AND_2 : \mathbb{B}^2 \rightarrow \mathbb{B}$



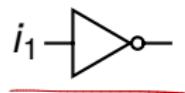
$i_1$	$i_2$	$OR_2$
0	0	0
0	1	1
1	0	1
1	1	1

$OR_2 : \mathbb{B}^2 \rightarrow \mathbb{B}$

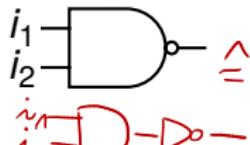


$i_1$	$NOT$
0	1
1	0

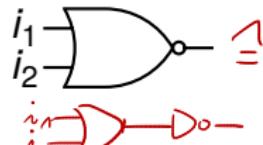
$NOT : \mathbb{B} \rightarrow \mathbb{B}$



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

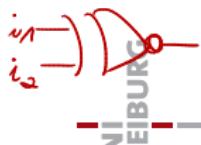


$i_1$	$i_2$	$NOR_2$
0	0	1
0	1	0
1	0	0
1	1	0



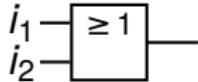
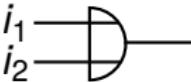
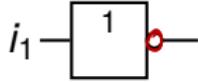
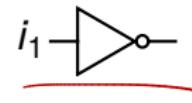
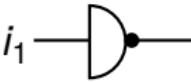
$i_1$	$i_2$	$XOR_2$
0	0	0
0	1	1
1	0	1
1	1	0

EQUIV		$XNOR_2$
in	$i_1$	0 0   1
	$i_2$	0 1   0
		1 0   0
		1 1   1



# Logikgatter - verschiedene Notationen

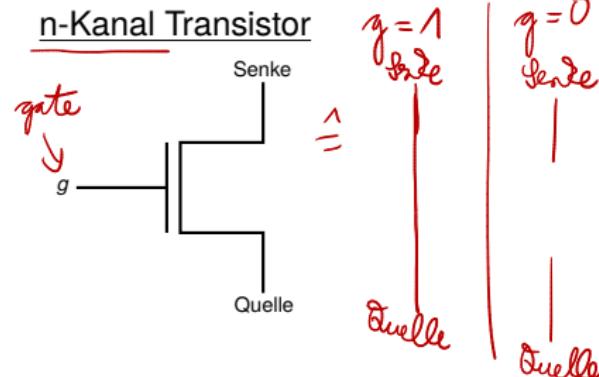
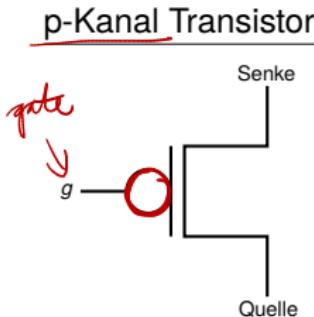
Es gibt verschiedene Notationen für Logikgatter.

	IEC	ANSI	DIN
$OR_2$			
$NOT$			

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

# Transistoren

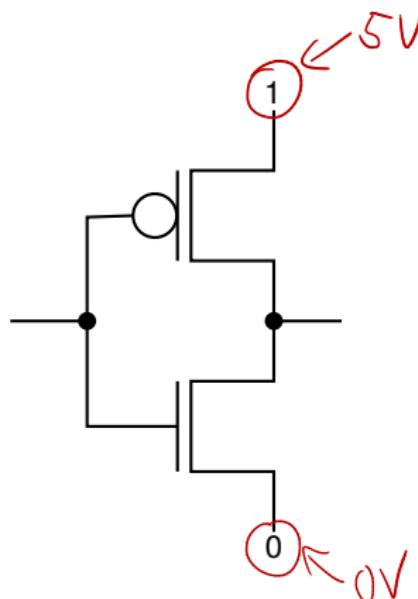
- Einen Transistor kann man vereinfacht als spannungsgesteuerten Schalter sehen:
  - Leitung  $g$  (gate) regelt Leitfähigkeit zwischen Quelle und Senke.



- 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.

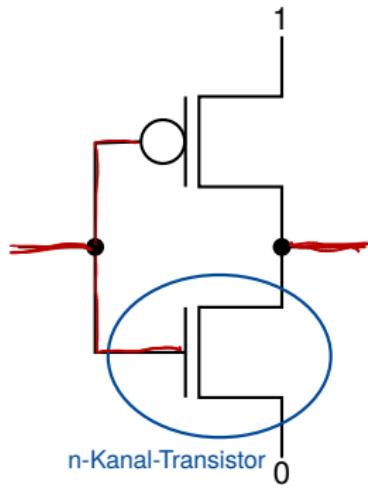
# Realisierung von Gattern in CMOS-Technologie

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



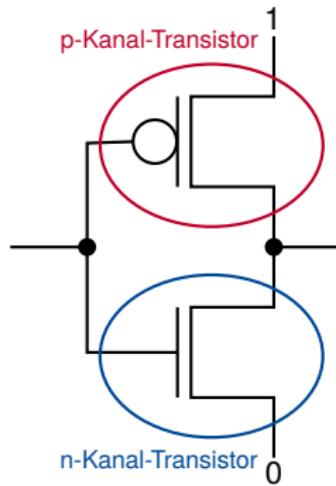
# Realisierung von Gattern in CMOS-Technologie

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



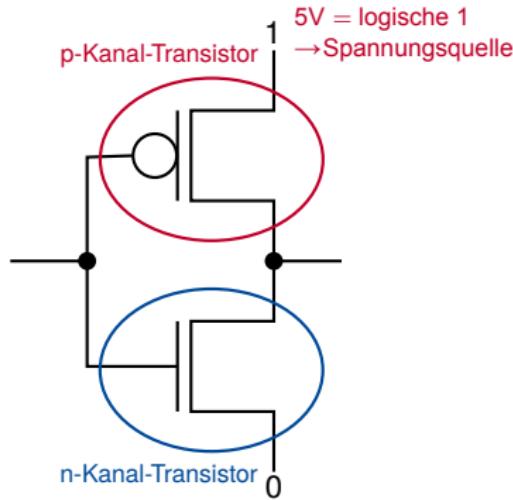
# Realisierung von Gattern in CMOS-Technologie

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



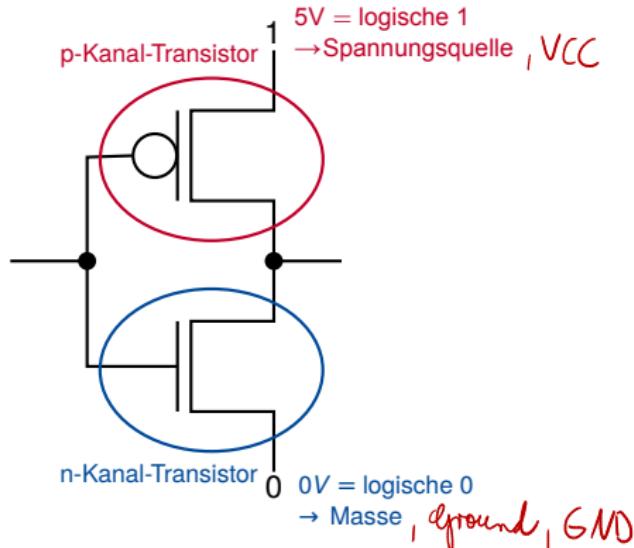
# Realisierung von Gattern in CMOS-Technologie

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



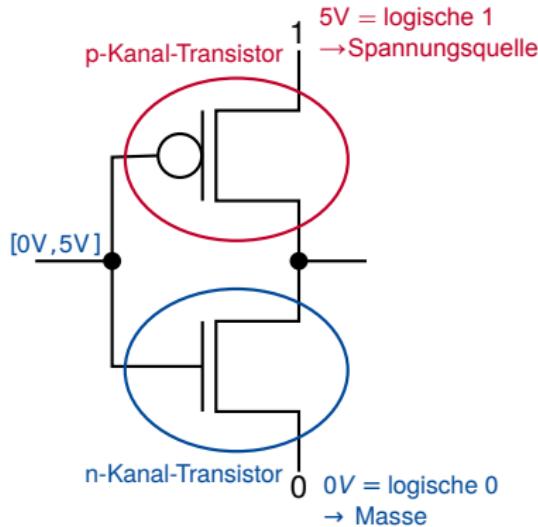
# Realisierung von Gattern in CMOS-Technologie

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



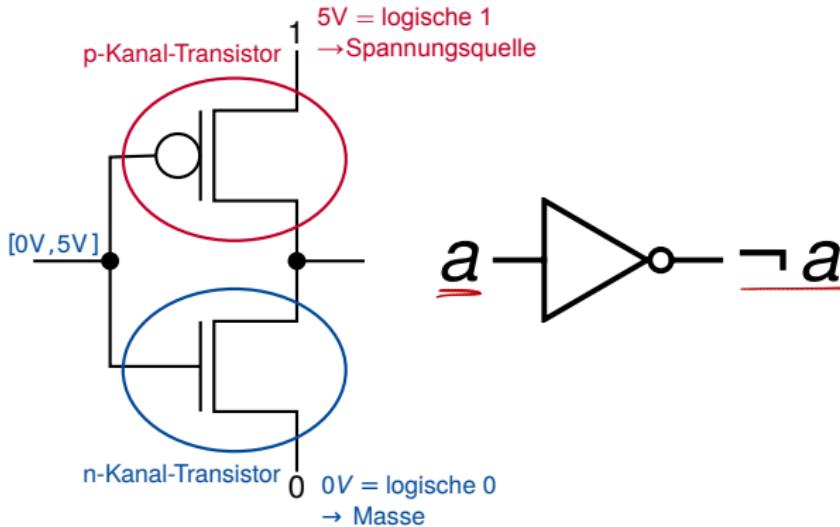
# Realisierung von Gattern in CMOS-Technologie

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



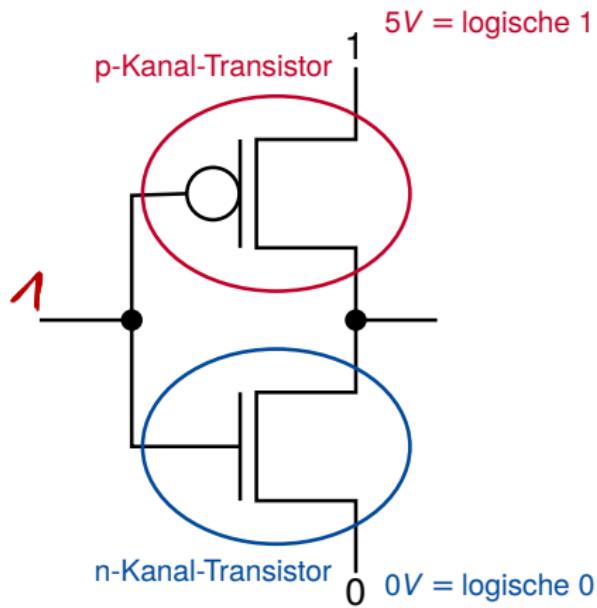
# Realisierung von Gattern in CMOS-Technologie

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



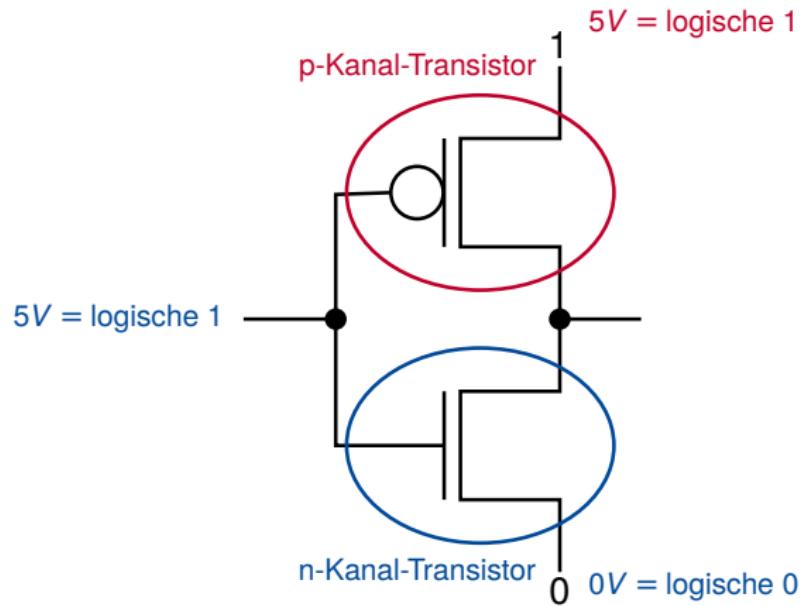
# CMOS-Inverter mit 1 am Gate

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



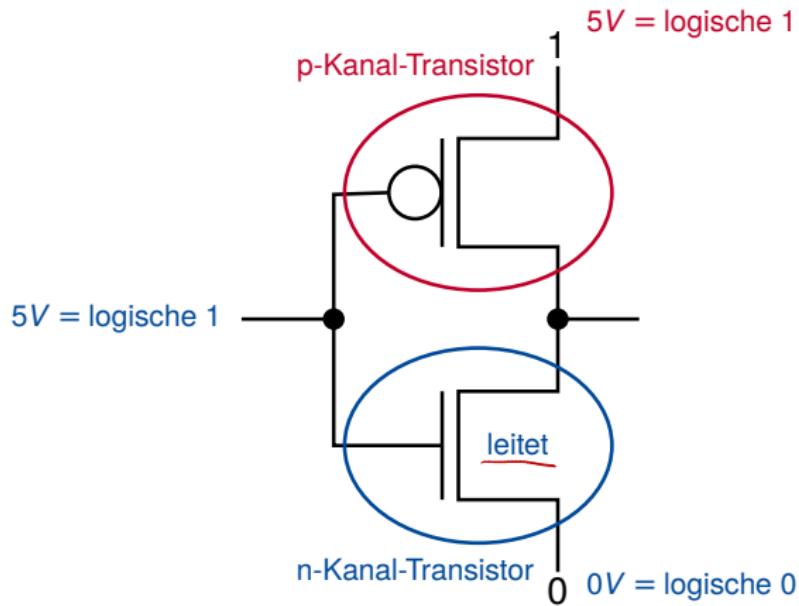
# CMOS-Inverter mit 1 am Gate

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



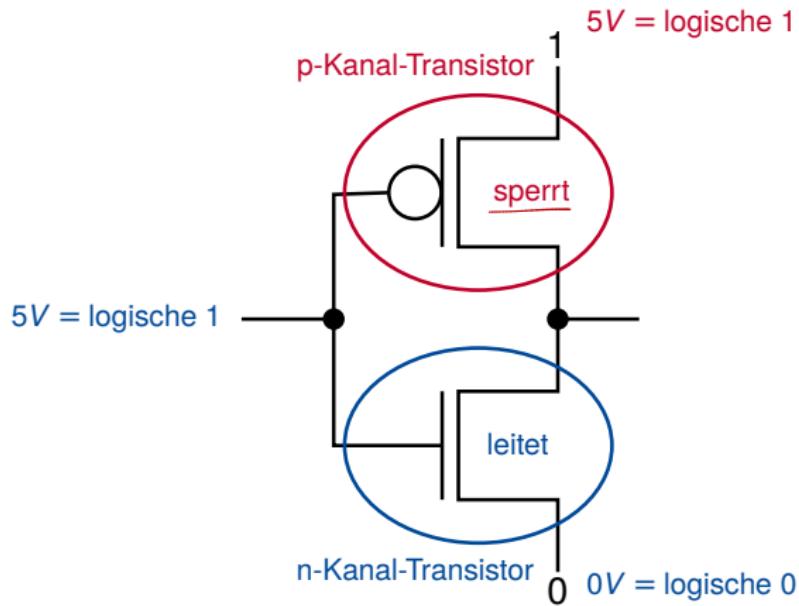
# CMOS-Inverter mit 1 am Gate

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



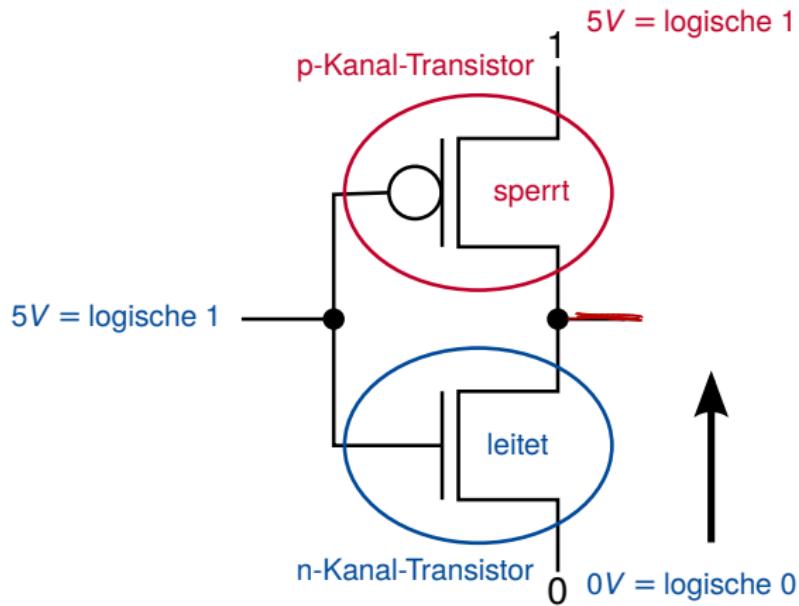
# CMOS-Inverter mit 1 am Gate

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



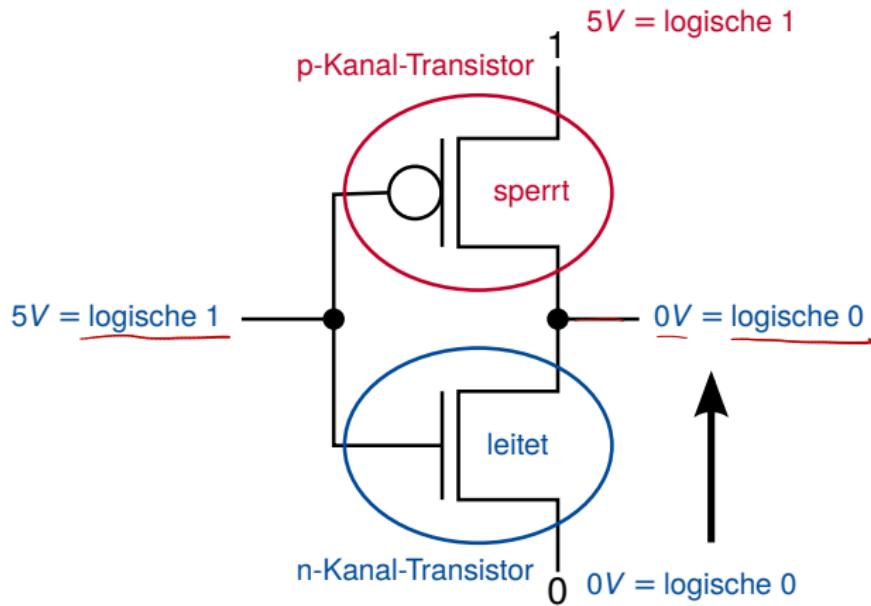
# CMOS-Inverter mit 1 am Gate

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



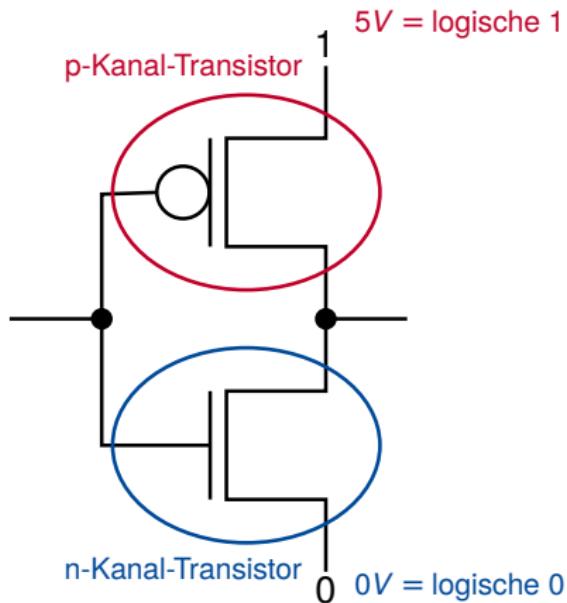
# CMOS-Inverter mit 1 am Gate

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



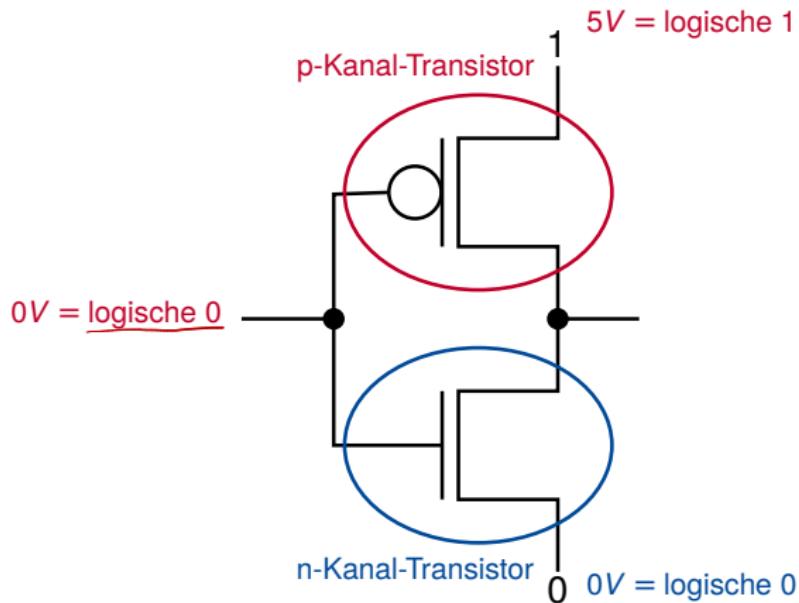
# CMOS-Inverter mit 0 am Gate

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



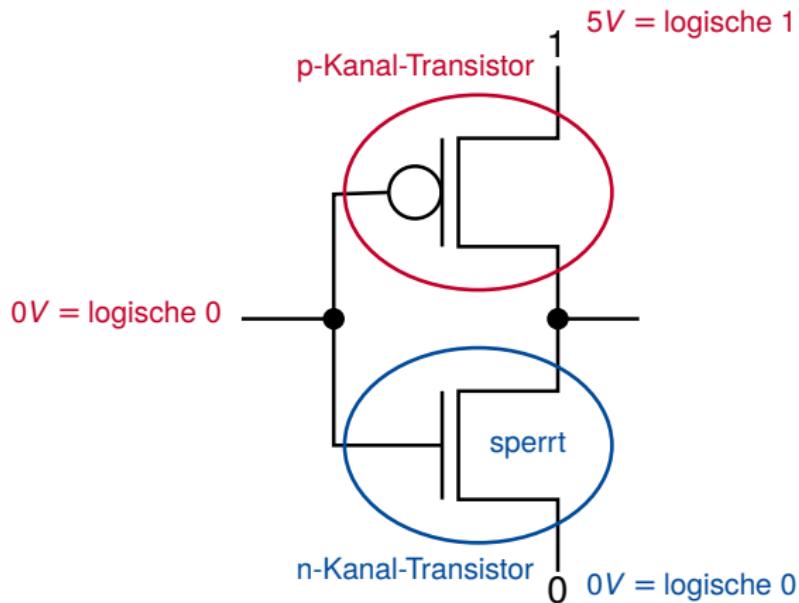
# CMOS-Inverter mit 0 am Gate

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



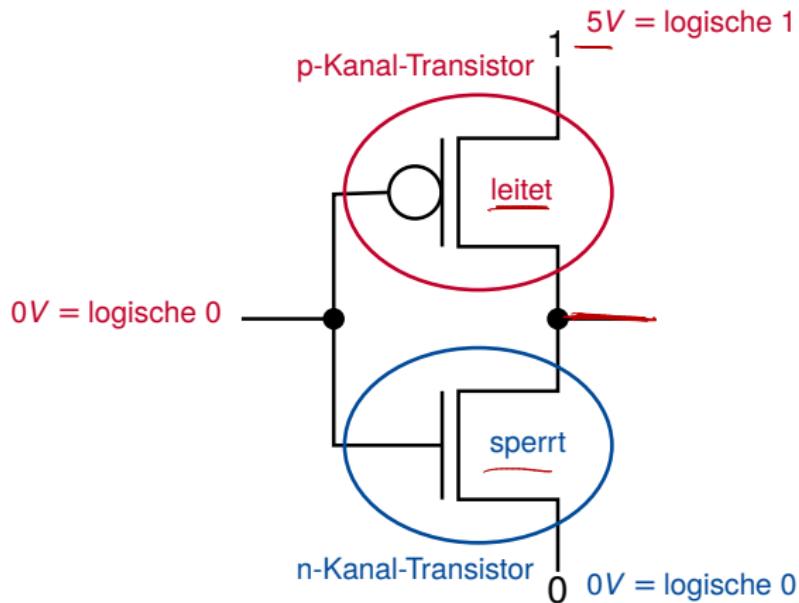
# CMOS-Inverter mit 0 am Gate

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



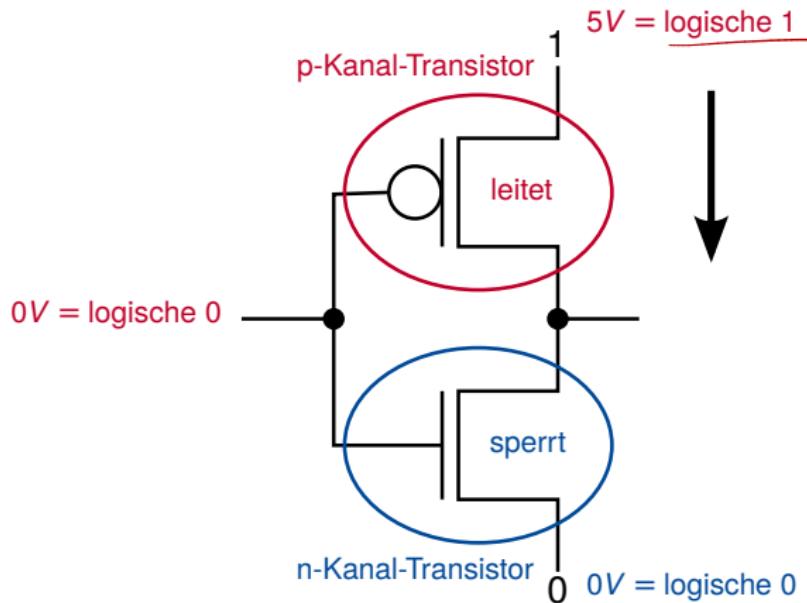
# CMOS-Inverter mit 0 am Gate

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



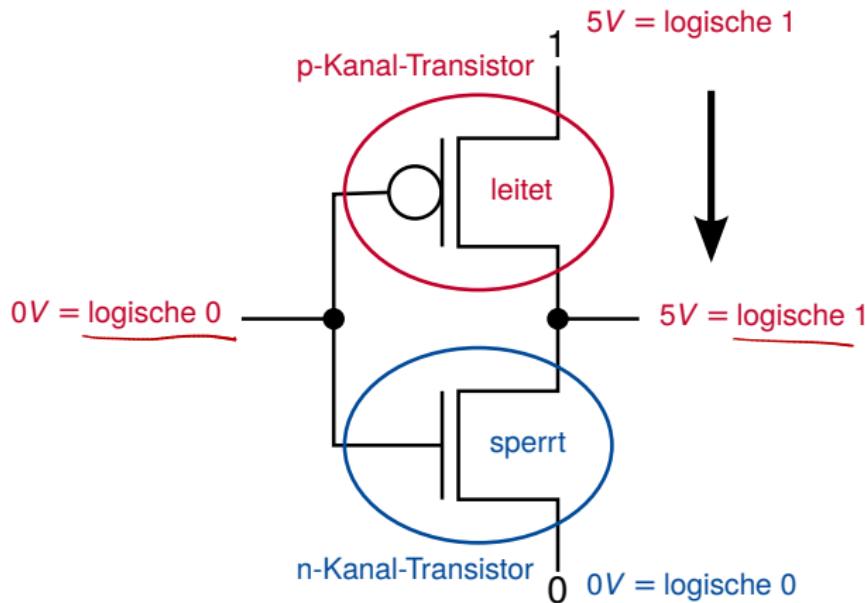
# CMOS-Inverter mit 0 am Gate

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

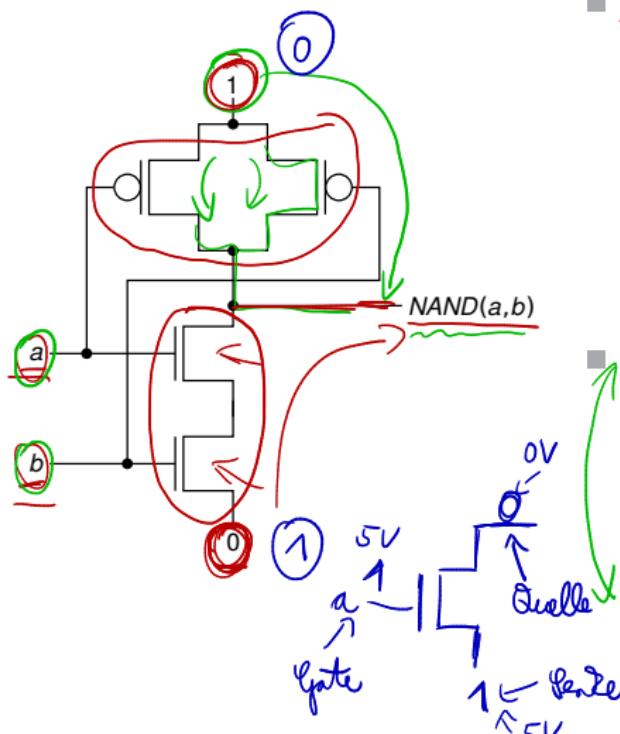


# CMOS-Inverter mit 0 am Gate

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



## CMOS-NAND-Gatter



■ Ausgang ist 0

$\Leftrightarrow$  Es existiert ein leitender Pfad von 0 zum Ausgang

$\Leftrightarrow$  beide n-Kanal-Transistoren leiten

$$\Leftrightarrow a = b = 1, a \wedge b = 1 \Leftrightarrow \overline{a \wedge b} = 0$$

$\Leftrightarrow \overline{NAND}(a, b) =$

## Ausgang ist 1

$\Leftrightarrow$  Es existiert ein leitender Pfad von 1 zum Ausgang

↔ einer der

p-Kanal-Transistoren leitet

$$\Leftrightarrow a = 0 \text{ oder } b = 0, \neg a \vee \neg b = 1$$

$$\Leftrightarrow \overline{NAND}(a,b) = 1$$

and

# 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.

# Kapitel 3 – Kombinatorische Logik

## 1. Kombinatorische Schaltkreise

1.1 Gatter, Transistoren

1.2 Definition

2. Boolesche Algebren

3. Boolesche Ausdrücke, Normalformen, zweistufige Synthese

4. Berechnung eines Minimalpolynoms

5. Arithmetische Schaltungen

6. Anwendung: ALU von ReTI

Albert-Ludwigs-Universität Freiburg

Prof. Dr. Christoph Scholl

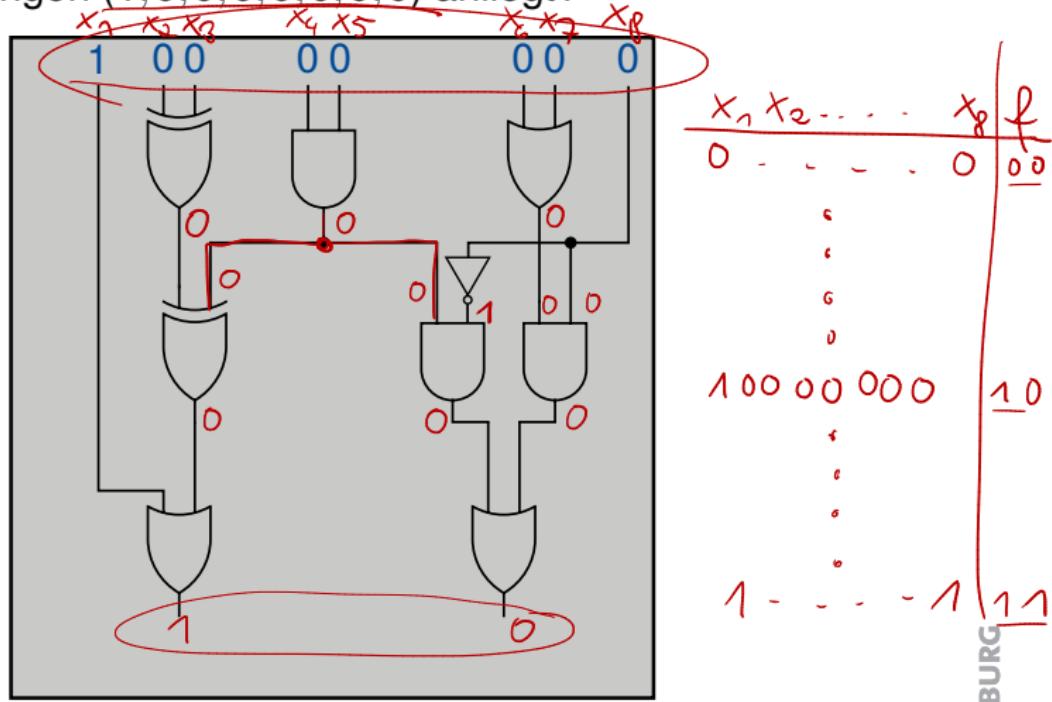
Institut für Informatik

Sommersemester 2023

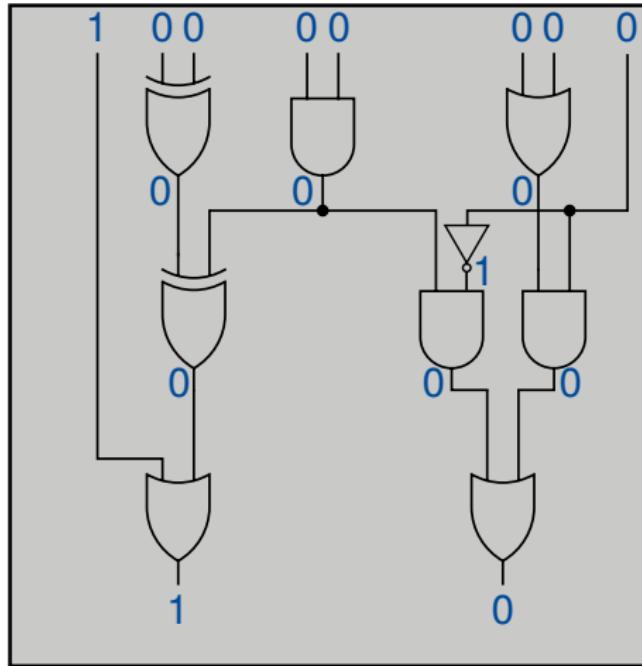
# Schaltkreis: Zunächst informal durch Beispiel

$f \in \mathbb{B}_{8,2}$

Welche Werte an den Ausgängen werden "berechnet", wenn an den Eingängen  $(1, 0, 0, 0, 0, 0, 0, 0)$  anliegt?



## Beispiel für einen Schaltkreis ( $f \in \mathbb{B}_{8,2}$ )



# Schaltkreise

---

- Idee:  
„gerichteter Graph mit einigen zusätzlichen Eigenschaften“

# Exkurs: Gerichteter Graph

## Definition

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

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

Bsp.:  $V = \{v_1, v_2, v_3, v_4\}$

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

Bsp.:  $E = \{e_1, e_2, e_3, e_4\}$

- Abbildungen  $Q : E \rightarrow V$  und  $Z : E \rightarrow V$

$Q(e)$  ist Quelle,  $Z(e)$  Ziel einer Kante  $e$

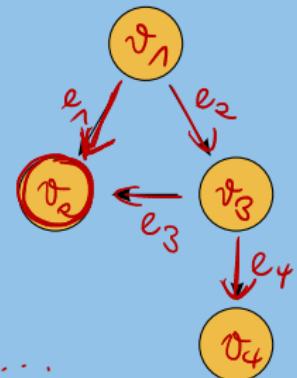
Bsp.:  $Q(e_1) = v_1, Z(e_1) = v_2, Q(e_2) = v_1, \dots$

- Abbildungen indeg :  $V \rightarrow \mathbb{N}$  und outdeg :  $V \rightarrow \mathbb{N}$

indeg( $v$ ) =  $|\{e \mid Z(e) = v\}|$  ist der **Eingangsgrad**,

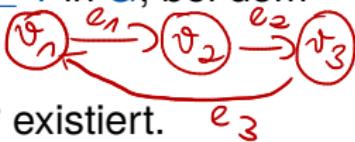
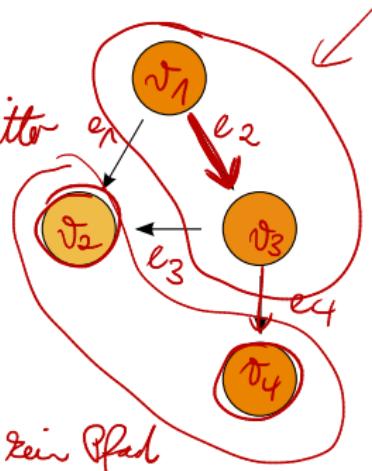
outdeg( $v$ ) =  $|\{e \mid Q(e) = v\}|$  der **Ausgangsgrad** von  $v$ .

Bsp.:  $\text{indeg}(v_3) = 1, \text{outdeg}(v_3) = 2$



# Exkurs: Pfade in gerichteten Graphen

- Ein Knoten mit
  - $\text{indeg}(v) = 0$  heißt Wurzel.  $v_1$  Wurzel
  - $\text{outdeg}(v) = 0$  heißt Blatt.  $v_2, v_4$  sind Blatt
  - $\text{outdeg}(v) > 0$  heißt innerer Knoten.  
 $v_1, v_3$  sind innere 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  $i$  ( $k-1 \geq i \geq 1$ )  
 $Q(e_1)$  heißt Quelle,  $Z(e_k)$  Ziel des Pfades.  
Bsp.:  $e_2, e_4$  Pfad,  $e_2, e_3$  Pfad,  $e_1, e_4$  kein Pfad
- Ein Zyklus in  $G$  ist ein Pfad der Länge  $\geq 1$  in  $G$ , bei dem Ziel und Quelle identisch sind  
 $e_1, e_2, e_3$   $Q(e_1) = v_1 = Z(e_3)$
- $G$  heißt azyklisch, falls kein Zyklus in  $G$  existiert.
- Die Graph-Tiefe eines azyklischen Graphen ist definiert als die Länge des längsten Pfades in  $G$ .  
Bsp.: 2



# Exkurs: Bäume, Binäre Bäume

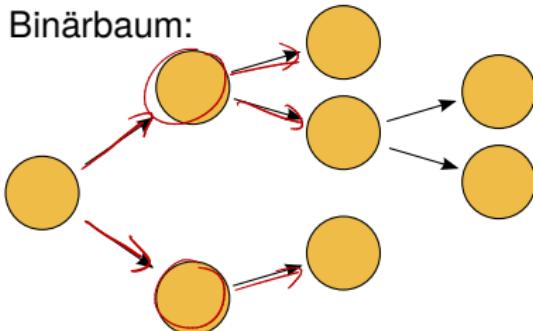
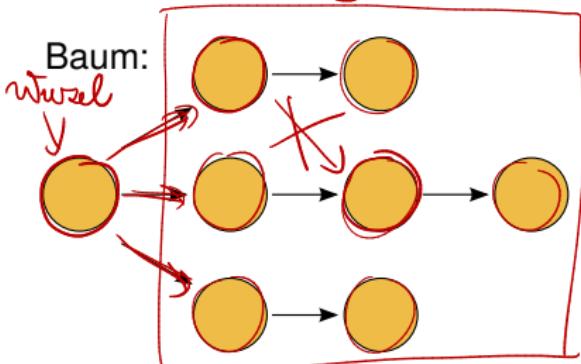
## Definition

Ein (Out-)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:

$\text{indeg} = 1$



Def.: Ein (In-)Baum ist ein ger. azykl. G. mit genau einem Blatt  $w$  ( $\text{outdeg}(w)=0$ ) und  $\text{outdeg}(v)=1$  f. a. anderen Knoten  $v$ .

# Modellierung durch Schaltkreise (1/2)

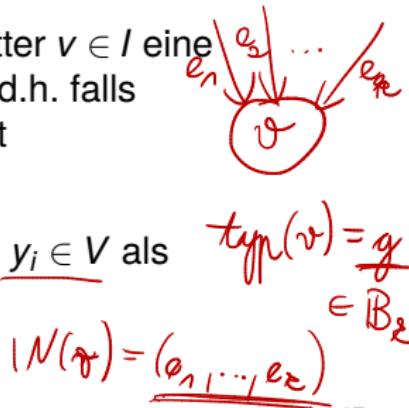
Bsp.:  $\text{STD} = \{\text{and}_2, \text{or}_2, \text{not}, \text{not}_2, \text{and}_2\}$

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

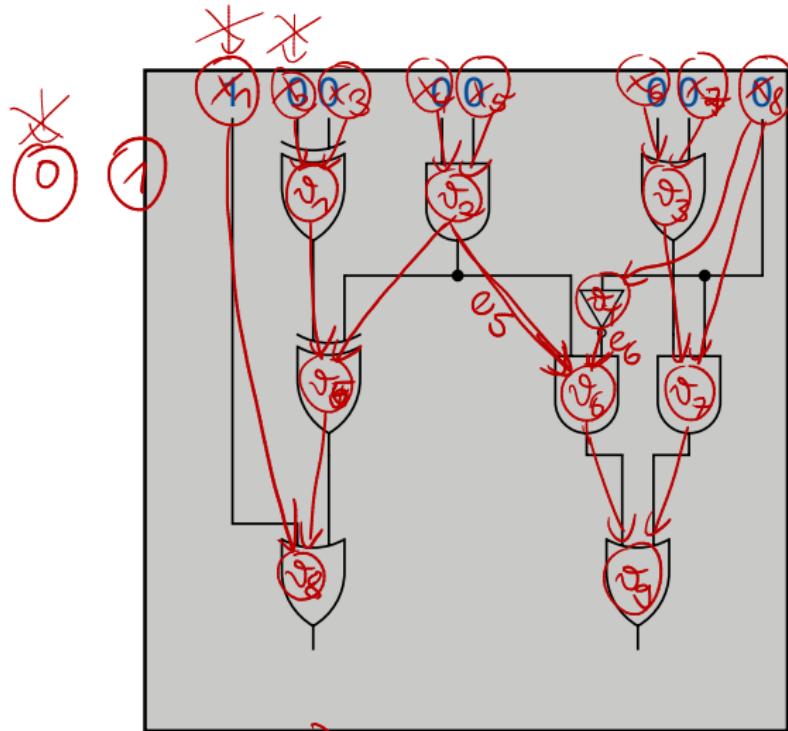
## Modellierung durch Schaltkreise (2/2)

■ ...

- 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  $\text{IN} : I \rightarrow E^*$  legt für jedes Gatter  $v \in I$  eine Reihenfolge der eingehenden Kanten fest, d.h. falls  $\text{indeg}(v) = k$ , dann ist  $\text{IN}(v) = (e_1, \dots, e_k)$  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.



# Schaltkreis für $f \in \mathbb{B}_{8,2}$



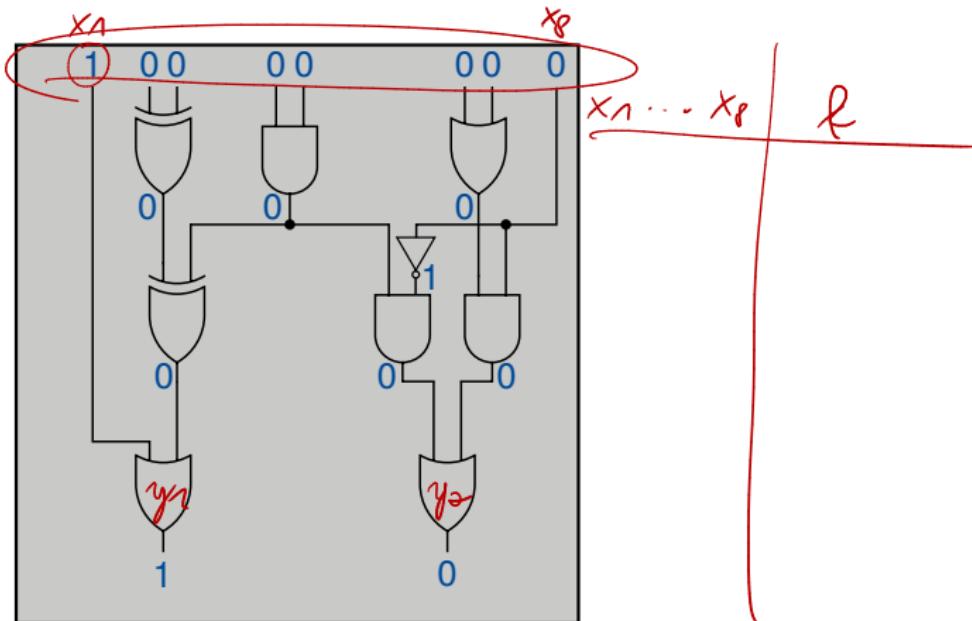
$$\text{typ}(v_1) = \text{sen}_2$$
$$\text{typ}(v_2) = \text{and}_2$$

$$\text{typ}(v_4) = \text{not}$$

$$IN(v_6) = (e_5, e_6)$$

$$\vec{Y}_2 = (v_8, v_9)$$

# Informale Semantikdefinition ( $f \in \mathbb{B}_{8,2}$ )



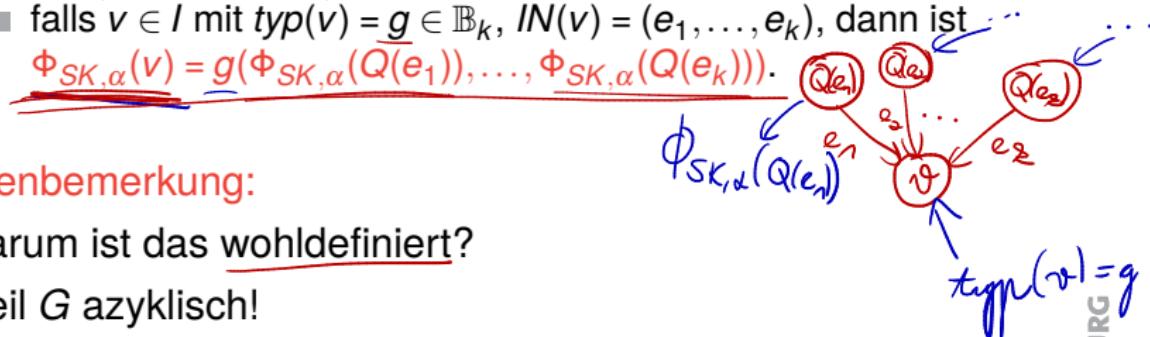
Die Boolesche Funktion  $f \in \mathbb{B}_{8,2}$  kann aus dem Schaltkreis hergeleitet werden, indem man für alle Werte aus  $\mathbb{B}^8$  den Schaltkreis auswertet ("simuliert").

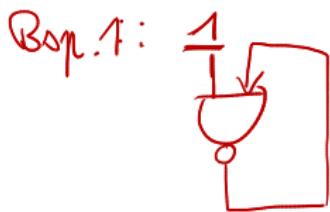
# Formale Semantikdefinition für Schaltkreise (1/2)

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

Zwischenbemerkung:

- Warum ist das wohldefiniert?
- Weil  $G$  azyklisch!





$$\phi_{SK,(1)}(x_1) = 1$$

$$\underline{\phi_{SK,(1)}(v)} = \text{nand}_2(\underline{\phi_{SK,(1)}(x_1)}, \underline{\phi_{SK,(1)}(v)})$$

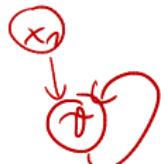
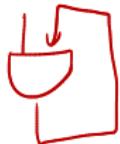
$\brace{=1}$

$\downarrow \quad \downarrow$

$\text{nand}_2(1, \underline{0}) = \underline{1}$

$\text{nand}_2(1, \underline{1}) = \underline{0}$

Bsp. 2:



$$\phi_{SK,(1)}(x_1) = 1$$

$$\underline{\phi_{SK,(1)}(v)} = \text{and}_2(\underline{\phi_{SK,(1)}(x_1)}, \underline{\phi_{SK,(1)}(v)})$$

$\brace{=1}$

$$\text{and}_2(1, 1) = 1$$

$$\text{and}_2(1, 0) = 0$$

## 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  $\Psi(v) : \underline{\mathbb{B}^n} \rightarrow \underline{\mathbb{B}}$  ist definiert durch

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

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

- Die durch den Schaltkreis berechnete Funktion ist

$$f_{SK} : \underline{\mathbb{B}^n} \rightarrow \underline{\mathbb{B}^m}, \underline{f_{SK}(\alpha)} = (\Psi(y_1)(\alpha), \dots, \Psi(y_m)(\alpha)).$$

- Eine Standardzellen-Bibliothek enthält eine Menge von Gattern (Standardzellen).
  - Z.B. AND-Gatter mit 4 Eingängen, 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

*e.g.: Bibliothek, Buch, Person  
ges.: SK, der f. bedient*

- 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**.
- Allgemeinere kombinatorische Schaltkreise betrachten wir später bei der Einführung arithmetischer Schaltkreise.

# Kapitel 3 – Kombinatorische Logik

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

# Überblick

---

- Boolesche Funktionen kann man durch Schaltkreise darstellen.
- Wir werden uns als nächstes mit Logiksynthese für **zweistufige** Schaltkreise beschäftigen.
- Bei der Logiksynthese für zweistufige Schaltkreise macht man häufig von einer alternativen Darstellungsform Gebrauch, den **Booleschen Ausdrücken**.
- Führe daher Boolesche Ausdrücke zunächst sauber ein, vorher aber noch eine etwas genauere Betrachtung von **Booleschen Algebren**.

# Boolesche Algebren - allgemein

- 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

$$\underline{x + y = y + x}$$

Assoziativität

$$\underline{x + (y + z) = (x + y) + z}$$

Absorption

$$\underline{x + (x \cdot y) = x}$$

Distributivität

$$\underline{x + (y \cdot z) = (x + y) \cdot (x + z)}$$

Komplement

$$\underline{x + (y \cdot \sim y) = x}$$

$$\underline{x \cdot y = y \cdot x}$$

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

$$\underline{x \cdot (x + y) = x}$$

$$\underline{x \cdot (y + z) = (x \cdot y) + (x \cdot z)}$$

$$\underline{x \cdot (y + \sim y) = x}$$

# Beispiele boolescher Algebren:

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

### Definition

- $\mathbb{B} := \{0, 1\}$
- Konjunktion (UND-Verknüpfung)  $\wedge : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$   
 $0 \wedge 0 = 0, \quad 0 \wedge 1 = 0, \quad 1 \wedge 0 = 0, \quad 1 \wedge 1 = 1$
- Disjunktion (ODER-Verknüpfung)  $\vee : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$   
 $0 \vee 0 = 0, \quad 0 \vee 1 = 1, \quad 1 \vee 0 = 1, \quad 1 \vee 1 = 1$
- Negation  $\neg : \mathbb{B} \rightarrow \mathbb{B}$   
 $\neg 0 = 1, \quad \neg 1 = 0$

$x$	$y$	$x+y$	$y+x$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	1

## Beispiele boolescher Algebren:

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

### Konventionen

- Man schreibt auch  $\cdot$  statt  $\wedge$  und  $+$  statt  $\vee$ .
- Für  $\neg x$  sind viele Notationen üblich:  $\sim x$ ,  $x'$  oder  $\bar{x}$ .

# Weitere Beispiele boolescher Algebren

---

- Boolesche Algebra der booleschen Funktionen in  $n$  Variablen:  $(\mathbb{B}_n, \cdot, +, \neg)$
  - Boolesche Algebra der Teilmengen einer Menge  $S$ :  
 $(\text{Pot}(S), \cap, \cup, \neg)$
- ⇒ **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 Funktionen in $n$ Variablen $(\mathbb{B}_n, \cdot, +, \neg)$

- Menge:  $\underline{\mathbb{B}_n}$  (Menge der booleschen Funktionen in  $n$  Variablen)
- $\cdot: \underline{\mathbb{B}_n} \times \underline{\mathbb{B}_n} \rightarrow \underline{\mathbb{B}_n}; \underline{(f \cdot g)(\alpha)} = f(\alpha) \cdot g(\alpha)$  für alle  $\underline{\alpha \in \mathbb{B}^n}$
- $+: \underline{\mathbb{B}_n} \times \underline{\mathbb{B}_n} \rightarrow \underline{\mathbb{B}_n}; \underline{(f + g)(\alpha)} = f(\alpha) + g(\alpha)$  für alle  $\underline{\alpha \in \mathbb{B}^n}$  ↪
- $\neg: \underline{\mathbb{B}_n} \rightarrow \underline{\mathbb{B}_n}; \underline{(\neg f)(\alpha)} = 1 \Leftrightarrow f(\alpha) = 0$  für alle  $\underline{\alpha \in \mathbb{B}^n}$

## Satz

$(\mathbb{B}_n, \cdot, +, \neg)$  ist eine boolesche Algebra.

- **Beweis:** Nachrechnen, dass **alle** Axiome gelten.

### Beispiel: Kommutativität

- Betrachte beliebige  $f, g \in \mathbb{B}_n$ . z.e.:  $f + g = g + f$
- Für alle  $\alpha \in \mathbb{B}^n$  gilt:  $(f + g)(\alpha) = \underbrace{f(\alpha) + g(\alpha)}_{+: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}} = \underbrace{g(\alpha) + f(\alpha)}_{+: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}} = (g + f)(\alpha)$ .
- Also  $f + g = g + f$ .

# Boolesche Algebra der Teilmengen von $S$ ( $Pot(S)$ , $\cap$ , $\cup$ , $\neg$ )

- Menge: Potenzmenge von  $S$
- $\cdot: \underline{Pot(S)} \times \underline{Pot(S)} \rightarrow \underline{Pot(S)}$ ;  $(M_1, M_2) \mapsto \underline{M_1 \cap M_2}$
- $+: \underline{Pot(S)} \times \underline{Pot(S)} \rightarrow \underline{Pot(S)}$ ;  $(M_1, M_2) \mapsto \underline{M_1 \cup M_2}$
- $\neg: \underline{Pot(S)} \rightarrow \underline{Pot(S)}$ ;  $M \mapsto \underline{\neg M} := \underline{S \setminus M}$



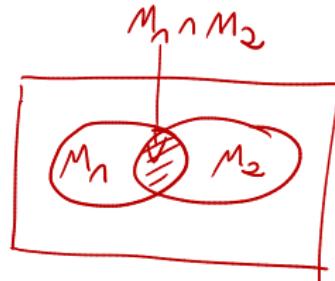
## Satz

$(Pot(S), \cap, \cup, \neg)$  ist eine boolesche Algebra.

- **Beweis:** Nachrechnen, dass alle Axiome gelten.

### Beispiel: Absorption

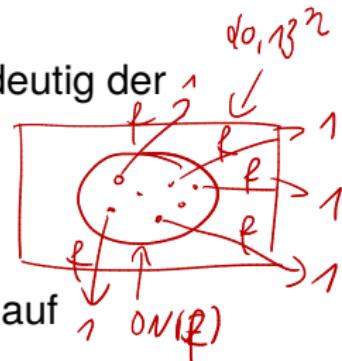
- Betrachte beliebige  $M_1, M_2 \in Pot(S)$ .  
 $\forall M_1, M_2 : M_1 + (M_1 \cdot M_2) = M_1$
- Dann ist  $(M_1 + (M_1 \cdot M_2)) = (M_1 \cup (M_1 \cap M_2)) = M_1$   
und  $(M_1 \cdot (M_1 + M_2)) = (M_1 \cap (M_1 \cup M_2)) = M_1$ .



# Zusammenhang zwischen $(\mathbb{B}_n, \cdot, +, \neg)$ und $(\text{Pot}(\mathbb{B}^n), \cap, \cup, \neg)$

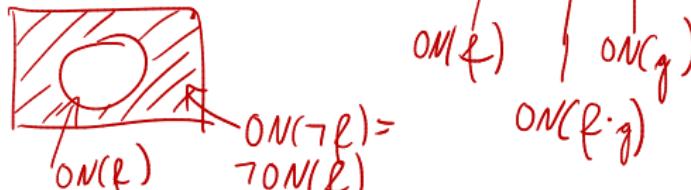
- Eine Funktion  $f \in \mathbb{B}_n$  entspricht umkehrbar eindeutig der folgenden Teilmenge von  $\mathbb{B}^n$ :

$$ON(f) = \{\alpha \in \mathbb{B}^n \mid f(\alpha) = 1\}$$



- Die Operationen in  $(\mathbb{B}_n, \cdot, +, \sim)$  übertragen sich auf  $(\text{Pot}(\mathbb{B}^n), \cap, \cup, \neg)$ :

- $ON(f \cdot g) = ON(f) \cap ON(g)$
- $ON(f + g) = ON(f) \cup ON(g)$
- $ON(\neg f) = \neg ON(f)$



# Weitere, aus den Axiomen ableitbare Regeln:

## ■ Existenz und Eindeutigkeit neutraler Elemente:

$\exists \mathbf{0} \in M : x + \mathbf{0} = x \forall x \in M$ ,  $\exists \mathbf{1} \in M : x \cdot \mathbf{1} = x \forall x \in M$  und außerdem sind die Elemente  $0$  und  $1 \in M$  mit der angegebenen Eigenschaft eindeutig.

→ ■  $\forall x \in M : x \cdot \neg x = \mathbf{0}$   $\forall x \in M : x + \neg x = \mathbf{1}$  *← entspricht Korollar*

■  $\forall x \in M : x \cdot \mathbf{0} = \mathbf{0}$   $\forall x \in M : x + \mathbf{1} = \mathbf{1}$

## ■ Doppeltes Komplement:

$$\forall x \in M : (\sim(\sim x)) = x$$

## ■ Eindeutigkeit des Komplements:

$$\forall x, y \in M : (x \cdot y = \mathbf{0} \text{ und } x + y = \mathbf{1}) \Rightarrow y = (\sim x)$$

## ■ Idempotenz:

$$\forall x \in M : x + x = x \quad x \cdot x = x$$

## ■ de Morgan-Regel:

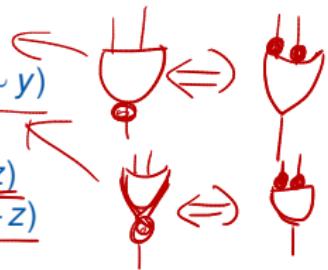
$$\forall x, y \in M : \sim(x + y) = (\sim x) \cdot (\sim y) \quad \sim(x \cdot y) = (\sim x) + (\sim y)$$

## ■ Consensus-Regel:

$$\forall x, y, z \in M : (x \cdot y) + ((\sim x) \cdot z) = (x \cdot y) + ((\sim x) \cdot z) + (y \cdot z)$$

$$\forall x, y, z \in M : (x + y) \cdot ((\sim x) + z) = (x + y) \cdot ((\sim x) + z) \cdot (y + z)$$

## ■ Diese Regeln gelten in allen booleschen Algebren!



1) Existenz:  $\exists 0 \in M$ , so dass  $\forall x \in M: x + 0 = x$

Folgt einfach aus der Komplementregel:

Für bel.  $y \in M$  gilt:  $x + (\underbrace{y \cdot \gamma_y}_{} = 0) = x$

2) Eindeutigkeit:

Falls es  $0, 0' \in M$  gibt mit  $\underbrace{x + 0 = x}_{(*)}$  und  $\underbrace{x + 0' = x}_{(**)}$  f. a.  $x \in M$ , dann  $0 = 0'$ .

$$\begin{array}{ccc} 0 = 0 + 0' & = 0' + 0 & = 0' \\ \uparrow & \uparrow & \uparrow \\ \text{wegen } (*) \text{ mit } x=0 & \text{Kommutat.} & \text{wegen } (**) \text{ mit } x=0' \end{array}$$

Korollar:  $\forall y \in M: y \cdot \gamma_y = 0$

## Prinzip der Dualität

Gilt eine aus den Axiomen 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:

- $(x \cdot y) + ((\sim x) \cdot z) + (y \cdot z) = (x \cdot y) + ((\sim x) \cdot z)$  ↙
- $(x + y) \cdot ((\sim x) + z) \cdot (y + z) = (x + y) \cdot ((\sim x) + z)$  ↙



# Kapitel 3 – Kombinatorische Logik

1. Kombinatorische Schaltkreise
2. Boolesche Algebren
- 3. Boolesche Ausdrücke, Normalformen, zweistufige Synthese**
  - 3.1 Boolesche Ausdrücke, Disjunktive Normalform
  - 3.2 zweistufige Logikminimierung
4. Berechnung eines Minimalpolynoms
5. Arithmetische Schaltungen
6. Anwendung: ALU von ReTI

# Kombinatorische Schaltkreise – zweistufig

---

## ■ Ziel:

Wir werden zeigen, dass sich jede boolesche Funktion als ein **Polynom**, d.h. als 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**
- Beginne daher mit einer exakten Definition, was wir unter einem booleschen Ausdruck verstehen.

# Boolesche Ausdrücke - allgemein

---

- Formal vollständige Definition boolescher Ausdrücke
  - Syntax (korrekte Schreibweise) → Def. boolescher Ausdrücke  $BE(X_n)$
  - Semantik (Bedeutung) → Interpretationsfunktion  $\Psi$  von  $BE(X_n)$

# Syntax boolescher Ausdrücke

- Sei  $X_n = \{x_1, \dots, x_n\}$  eine endliche Menge von Variablen.
- Sei  $A = \underline{X_n} \cup \{0, 1, +, \cdot, \sim, (, )\}$  ein Alphabet.

## Definition

Die Menge  $BE(X_n)$  der vollständig geklammerten booleschen Ausdrücke über  $X_n$  ist die kleinste 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  $(\sim g)$ .

**Bsp.:**  $((x_1 \cdot x_2)) + (\sim x_3) \in BE(\underline{X_3})$ .

# 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.

$$(x_1 + (x_2 \cdot x_3)) \stackrel{\Delta}{=} x_1 + x_2 x_3$$

- Man schreibt auch

- statt  $\cdot$ :  $\wedge$ ,
- statt  $+$ :  $\vee$ ,
- statt  $\sim x$ :  $\neg x, x'$ ,  $\bar{x}$ .

→ 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 „ $x_1 \wedge x_2 + \neg x_3$ “ wäre „ $((x_1 \cdot x_2) + (\sim x_3))$ “.

# Semantik boolescher Ausdrücke

## Definition

Jedem booleschen Ausdruck aus  $BE(X_n)$  kann durch eine **Interpretationsfunktion**  $\Psi : BE(X_n) \rightarrow \mathbb{B}_n$  eine boolesche Funktion zugeordnet werden.

$\Psi$  wird folgendermaßen induktiv definiert:

- $\Psi(0) = 0; \Psi(1) = 1;$
- $\Psi(x_i)(\alpha_1, \dots, \alpha_n) = \alpha_i$  für alle  $\alpha \in \mathbb{B}^n$  (Projektion)
- $\Psi((g + h)) = \Psi(g) + \Psi(h)$  (Disjunktion)
- $\Psi((g \cdot h)) = \Psi(g) \cdot \Psi(h)$  (Konjunktion)
- $\Psi((\sim g)) = \sim (\Psi(g))$  (Negation)

Nullfkt.  $\Theta : \{0, 1\}^n \rightarrow \{0, 1\}, \Theta(\alpha) = 0 \quad \forall \alpha \in \{0, 1\}^n$

# Alternative Betrachtung der Semantik boolescher Ausdrücke

- Sei  $e$  ein boolescher Ausdruck.

- $\Psi(e)(\alpha)$  für ein  $\alpha \in \mathbb{B}^n$  ergibt sich durch Ersetzen von  $x_i$  durch  $\alpha_i$  in  $e$ , für alle  $i$  und Rechnen in der booleschen Algebra  $\mathbb{B}$ .

- **Bsp.:**

$$\underline{\Psi(((x_1 \cdot x_2) + (\sim x_3)))}(0, 0, 1) = ((0 \cdot 0) + (\sim 1)) = (0 + 0) = \underline{0}$$

- Gilt  $\underline{\Psi(e)} = f$  für eine boolesche Funktion  $f \in \mathbb{B}_n$ , so sagen wir, dass  $e$  ein boolescher Ausdruck für  $f$  ist, bzw. dass  $e$  die boolesche Funktion  $f$  beschreibt.

In Zukunft schreiben wir auch  $\underline{f} = e$  statt  $\underline{\Psi(e)} = f$ .

- 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  $e_1 = e_2$ .

$$\underline{e_1} = x_n + x_1 x_2$$

$$\underline{e_2} = x_1$$

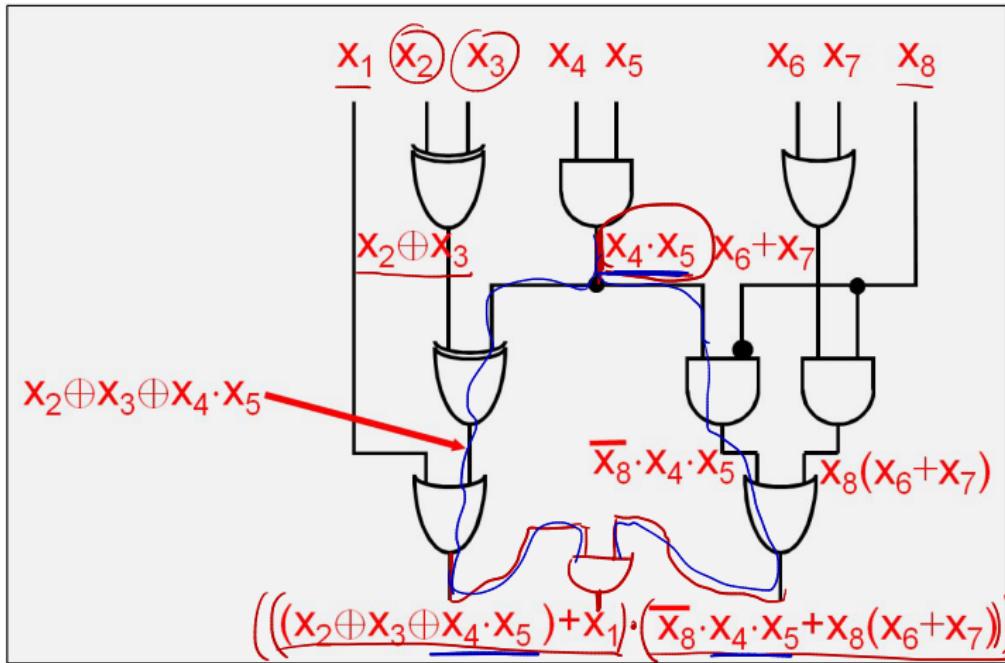
$$e_1 \equiv e_2, \text{ weil } \underline{\Psi(e_1)} = \underline{\Psi(e_2)}$$

# Beziehung zwischen Schaltkreisen und booleschen Ausdrücken (1/2)

---

- Zu jedem Ausgang eines Schaltkreises lässt sich durch „symbolische Simulation“ ein boolescher Ausdruck berechnen, der die entsprechende boolesche Funktion darstellt.
- Symbolische Simulation benutzt zur Simulation eines Schaltkreises keine festen booleschen Werte an den Inputs, sondern boolesche Variablen.
- Es wird dann für jeden Knoten ein boolescher Ausdruck zu der Funktion bestimmt, die der Knoten berechnet.

# Beziehung zwischen Schaltkreisen und booleschen Ausdrücken (2/2)



$(x_1 \oplus x_2)$  ist eine abkürzende Schreibweise für  $x_1\bar{x}_2 + \bar{x}_1x_2$ .

# Spezielle boolesche Ausdrücke: Literale

## 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  $f$ .

- $\underline{x}_i$  (auch  $\underline{x}_i^1$  geschrieben) wird positives Literal,  
 $\overline{x}_i = \underline{x}_i'$  (auch  $\underline{x}_i^0$  geschrieben) wird negatives Literal genannt.  
 $= \sim x_i = \overline{x}_i$

- Anmerkung: Eine **andere Notation** für ein negatives Literal  $x'$  ist  $\neg x$  oder  $\overline{x}$ .
- Wie schon erwähnt bezeichnet das Literal  $\underline{x}_i$  die boolesche Funktion  $g \in \mathbb{B}_n$  mit  $g(\alpha_1, \dots, \alpha_n) = 1$  genau dann, wenn  $\alpha_i = 1$ .  
 $\underline{x}'_i$  bezeichnet die boolesche Funktion  $h \in \mathbb{B}_n$  mit  $h(\alpha_1, \dots, \alpha_n) = 1$  genau dann, wenn  $\alpha_i = 0$ .

## Spezielle boolesche Ausdrücke: Monome

## Definition

- 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,  $\underline{x_2x_3x'_3}$  ist kein Monom,  $\underline{\underline{x_2x_3x_3}}$  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  $\underline{x_1x_2x'_3}$  ein Minterm,  $x'_1x_3$  ist kein Minterm.
    - Für eine Eingabebelegung  $\underline{\alpha \in \mathbb{B}^n}$  heißt
 
$$m(\alpha) = \underline{\underline{x_1 \cdot x_2 \cdot x_3}} \quad \begin{array}{l} \alpha = (1, 0, 1) \\ m(\alpha) = \underline{\underline{x_1 \cdot x_2 \cdot x_3}} \\ \uparrow \quad \uparrow \quad \uparrow \\ 1 \cdot 0 \cdot 1 = 1 \cdot 1 \cdot 1 = 1 \end{array}$$

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

# Monome als Beschreibung boolescher Funktionen

---

- Beispiel: Das Monom  $m = x_i x_j'$  bezeichnet die boolesche Funktion  $f \in \mathbb{B}_n$  mit  $\underline{f(\alpha_1, \dots, \alpha_n)} = 1$  genau dann, wenn  $\underline{\alpha_i} = 1$  und  $\underline{\alpha_j} = 0$ . Wir schreiben vereinfachend auch  $f = x_i x_j'$ .  
 $\cancel{f(x_i x_j')} = f$
- Beispiel: Der Minterm  $m(0, \underline{1}, 0, 1) = \underline{x_1} x_2 \underline{x_3} x_4$  bezeichnet die boolesche Funktion  $f \in \mathbb{B}_4$  mit  $\underline{f(\alpha)} = 1$  genau dann, wenn  $\underline{\alpha} = (0, 1, 0, 1)$ .

# Spezielle boolesche Ausdrücke: Polynome

---

- Eine Disjunktion von paarweise verschiedenen Monomen heißt **Polynom**. Außerdem ist „0“ ein Polynom. Sind alle Monome des Polynoms vollständig, so heißt das Polynom **vollständig**.

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

- $x'_1x_2 + x'_2x_3$  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

---

- Ein Polynom für  $f$  heißt auch disjunktive Normalform (DNF) von  $f$ . Ein vollständiges Polynom für  $f$  heißt auch kanonische disjunktive Normalform (KDNF) von  $f$ .
- $f_1 = \underline{x'_1} \underline{x'_2} + \underline{x'_2} \underline{x_3} + \underline{x_1} \underline{x_2}$  ist in DNF, aber nicht in KDNF.

# Bestimmung der kanonischen disjunktiven Normalform

- Für  $f : \mathbb{B}^n \rightarrow \mathbb{B}$  heißt  $\underline{\text{ON}}(f) := \{\alpha \in \mathbb{B}^n \mid f(\alpha) = 1\}$  die Erfüllbarkeitsmenge (ON-Menge) von  $f$ .
- Die KDNF ist gegeben durch  $f = \sum_{\alpha \in \text{ON}(f)} m(\alpha)$
- Die KDNF ist (bis auf Anordnung von Literalen in Mintermen und von Monomen im Polynom) eindeutig.
- Beispiel: KDNF für  $f_1 = x'_1x'_2 + x'_2x_3 + x_1x_2$

$$\begin{aligned}f_1 &= \underline{m(000)} + \underline{m(001)} \\&\quad + \underline{m(101)} + \underline{m(110)} + \underline{m(111)} \\&= \underline{x'_1x'_2x'_3} + \underline{x'_1x'_2x_3} \\&\quad + \underline{x_1x'_2x_3} + \underline{x_1x_2x'_3} + \underline{x_1x_2x_3}\end{aligned}$$

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

$f_1 \in \mathbb{B}_3$

$x_1$	$x_2$	$x_3$	$f_1$	
0	0	0	1	1 0
0	0	1	1	0 1
0	1	0	0	0 0
0	1	1	0	0 0
1	0	0	0	0 0
1	0	1	1	0 0
1	1	0	1	0 0
1	1	1	1	0 0

$m(000) \ m(001)$

# Realisierungen von DNF

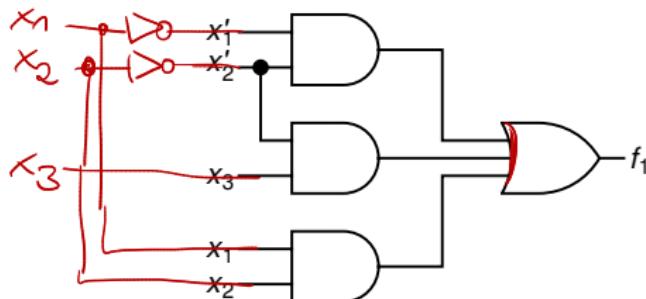
---

- Erste Möglichkeit: Benutze „gewöhnliche“ UND- und ODER-Gatter.
- Zweite Möglichkeit: PLAs (Programmable Logic Arrays)
  - 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.

$$\overline{x_1} \overline{x_2} + \overline{x_2} x_3 + x_1 x_2$$



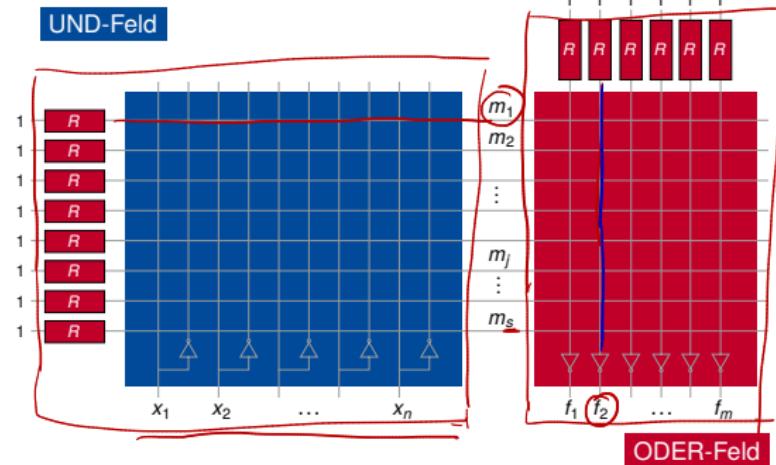
- Die Kosten ergeben sich dann aus allen benötigten UND- und ODER-Gattern.

# Programmierbare logische Felder (PLA)

Eingangsvariablen:  $x_1, \dots, x_n$

Zweistufige Darstellung zur Realisierung von booleschen Polynomen.  $f_1, f_2, \dots, f_m$

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



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

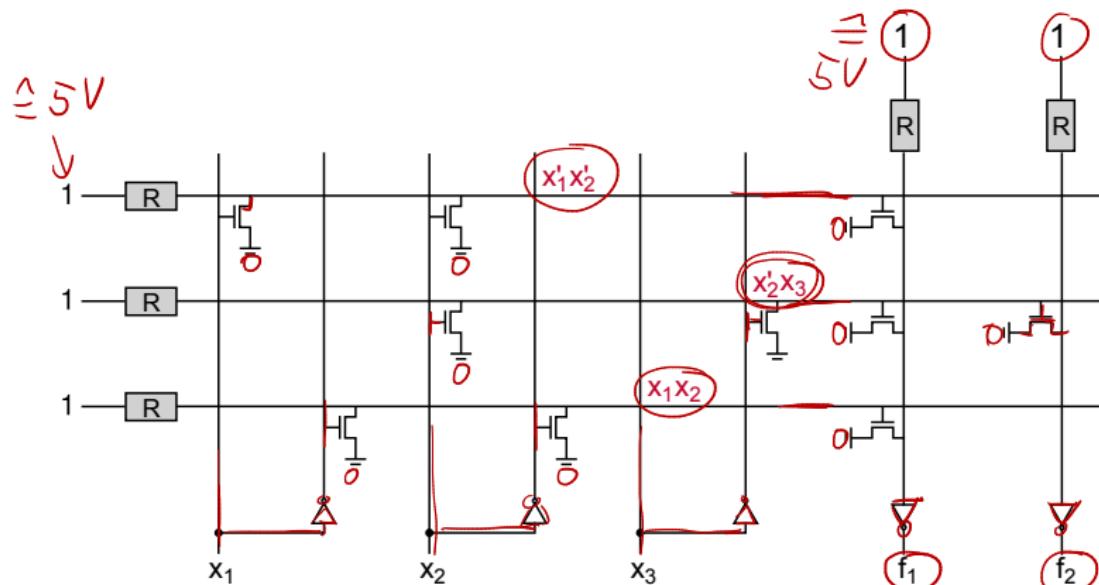
Besteht die Beschreibung von Funktion  $f_t$  aus  $p$  Monomen, so benötigt man  $p$  Transistoren in der entsprechenden Spalte des **ODER-Feldes**.

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

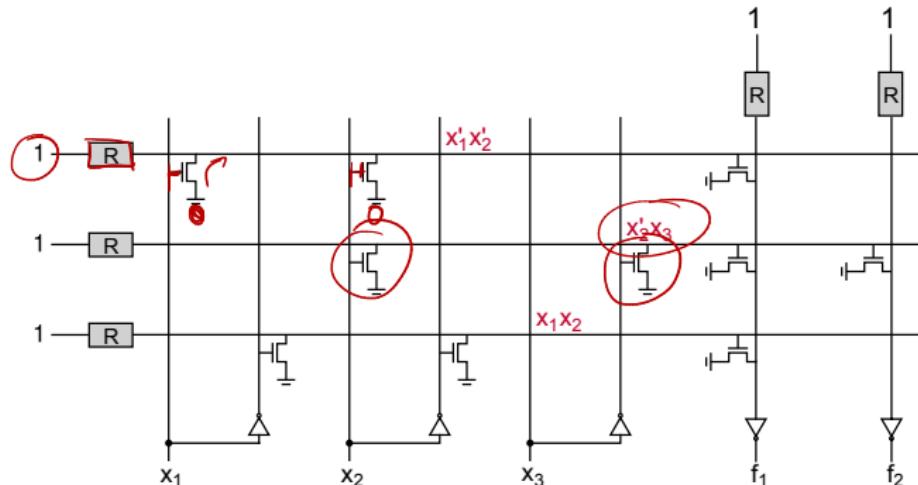
# PLAs: Realisierungsdetails

## Beispiel

$$f_1 = \underline{x'_1 x'_2} + \underline{x'_2 x_3} + \underline{x_1 x_2}$$
$$f_2 = \underline{x'_2 x_3}$$

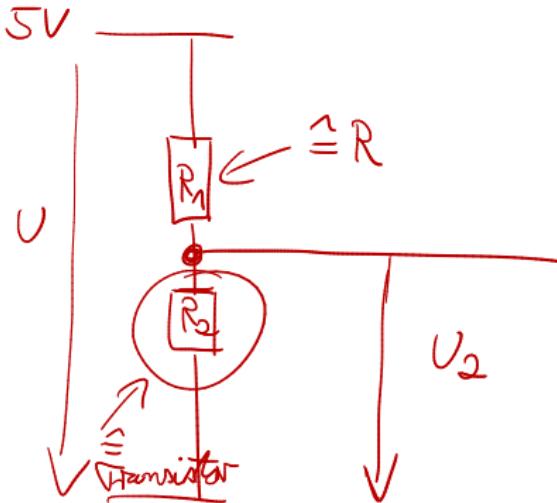


# Realisierungsdetails, waagerechte Monomleitungen



- Ein n-Kanal-Transistor leitet, wenn sein Gate mit 1 belegt ist.
- Wenn ein n-Kanal-Transistor leitet, dann zieht er die entsprechende Monomleitung auf 0.
- Bsp.: Die erste Monomleitung ist genau dann 1, wenn *beide* Transistoren der Reihe sperren, d.h. wenn  $x_1 = 0$  und  $x_2 = 0$ .  
⇒ Die Leitung realisiert die Funktion  $x'_1 x'_2$ .

$$\overline{(x_2 + x_3)} = \overline{x_2} \cdot \overline{x_3}$$



Transistor leitet:

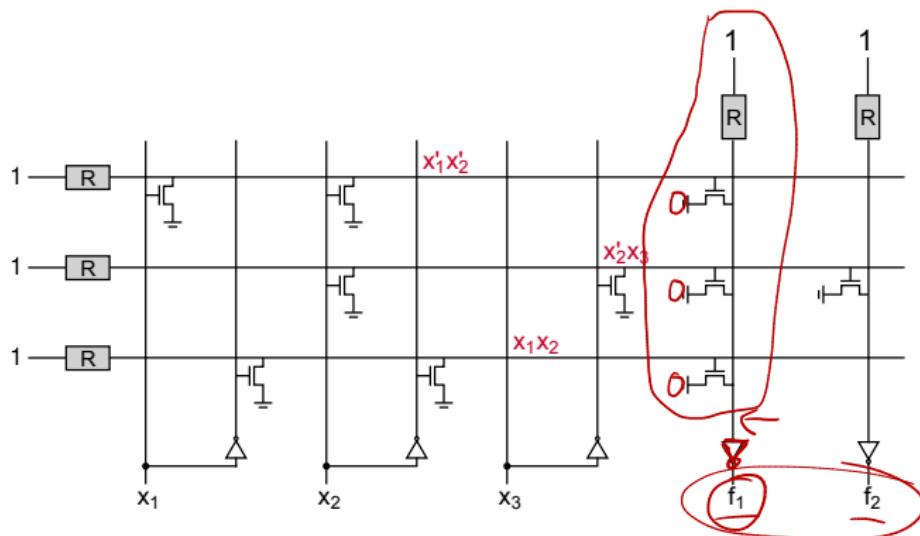
$$R_2 \approx 0 \Rightarrow U_2 \approx 0V$$

Transistor Sperrt:  $\stackrel{\approx 0}{\downarrow}$

$$\text{„}R_2 \approx \infty\text{“} \Rightarrow U_2 \approx U = 5V \quad U_2 = R_2 \cdot \frac{U}{R_h + R_2} = \frac{R_2}{R_h + R_2} \cdot U$$

$\stackrel{\approx 1}{\downarrow}$

# Realisierungsdetails, senkrechte Funktionsleitungen



- Ein n-Kanaltransistor leitet, wenn sein Gate mit 1 belegt ist.
- Wenn ein n-Kanal-Transistor leitet, dann zieht er die entsprechende Leitung auf 0.
- Bsp.: Der  $f_1$ -Ausgang ist genau dann 1, wenn der Eingang des zugehörigen Inverters 0 ist.  
Der Inverter-Eingang ist genau dann 0, wenn mindestens einer der drei Transistoren der Spalte leitet, d.h. wenn  $x'_1 x'_2 = 1$  oder  $x'_2 x_3 = 1$  oder  $x_1 x_2 = 1$ .  
⇒ Der  $f_1$ -Ausgang realisiert die Funktion  $x'_1 x'_2 + x'_2 x_3 + x_1 x_2$ .

- 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
$$cost_2(p_1, \dots, p_m) = \sum_{q \in M(p_1, \dots, p_m)} |q| + \sum_{i=1, \dots, m} |M(p_i)|.$$

# der Trans. im AND-Feld      # der Trans. im OR-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

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

genau dann 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  $\text{cost}_2(p_1, \dots, p_m) \leq \text{cost}_2(p'_1, \dots, p'_m)$

# Beispiel: 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.
- e. PLA 5 mit 10 Zeilen und 18 Transistoren ←



# Kapitel 3 – Kombinatorische Logik

1. Kombinatorische Schaltkreise
2. Boolesche Algebren
- 3. Boolesche Ausdrücke, Normalformen, zweistufige Synthese**
  - 3.1 Boolesche Ausdrücke, Disjunktive Normalform
  - 3.2 zweistufige Logikminimierung**
4. Berechnung eines Minimalpolynoms
5. Arithmetische Schaltungen
6. Anwendung: ALU von ReTI

Albert-Ludwigs-Universität Freiburg

Prof. Dr. Christoph Scholl

Institut für Informatik  
Sommersemester 2023

# Kombinatorische Schaltkreise – zweistufig

---

## ■ Ziel:

Wir werden zeigen, dass sich jede boolesche Funktion als ein **Polynom**, d.h. als eine **Disjunktion** (ODER-Verknüpfung) von **Monomen**, die ihrerseits **Konjunktionen** (UND-Verknüpfungen) von Eingangsvariablen und negierten Eingangsvariablen sind, darstellen lässt.

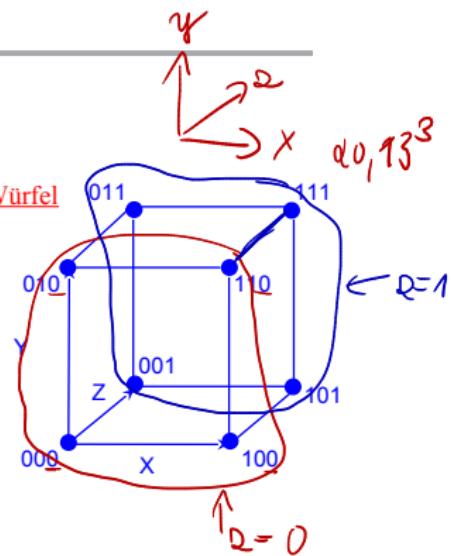
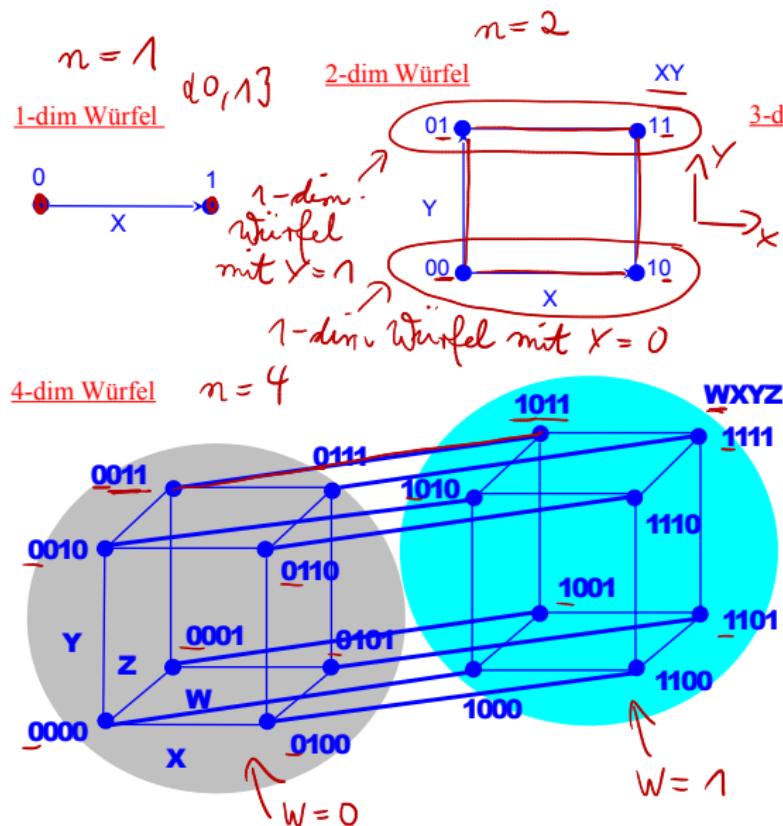
- Wir werden für solche Darstellungen **Kostenkriterien** aufstellen und diese optimieren.
- **Monome** und **Polynome** sind spezielle **boolesche Ausdrücke**.

# 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  $\text{cost}(p_1, \dots, p_m)$  minimal sind.
- Ab sofort werden nur noch Funktionen **mit einem Ausgang** betrachtet.  $m=1$

# Veranschaulichung von $\{0, 1\}^n$



# Veranschaulichung durch Würfel

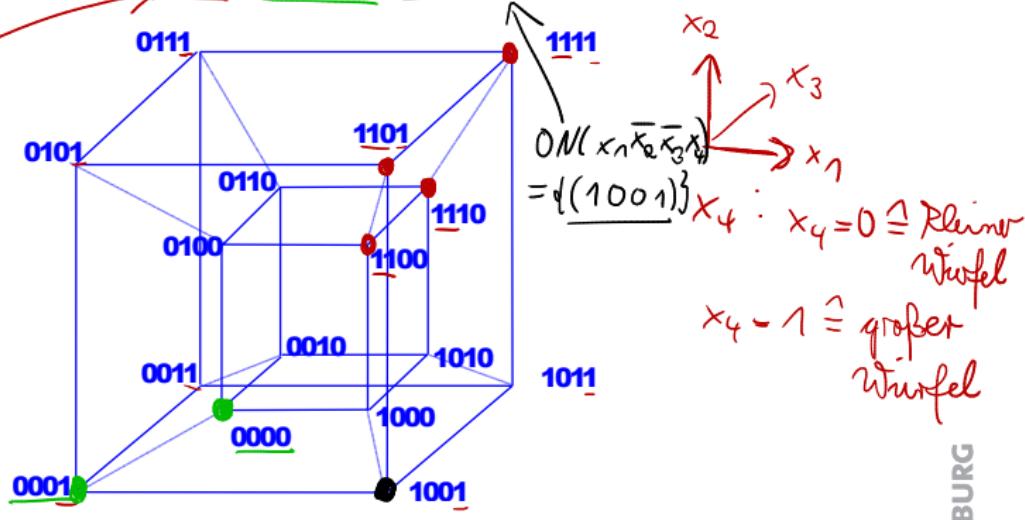
- Jede boolesche Funktion  $f$  in  $n$  Variablen und einem Ausgang kann über einen  $n$ -dimensionalen Würfel durch Markierung der  $ON(f)$ -Menge spezifiziert werden.

- Beispiel:**

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

$n = 4$

$$ON(x_1 x_2) = \{ (1\ 1\ 0\ 0), (1\ 1\ 0\ 1), (1\ 1\ 1\ 0), (1\ 1\ 1\ 1) \}$$



# Monome und Polynome als Teilwürfel

*n Variablen!*

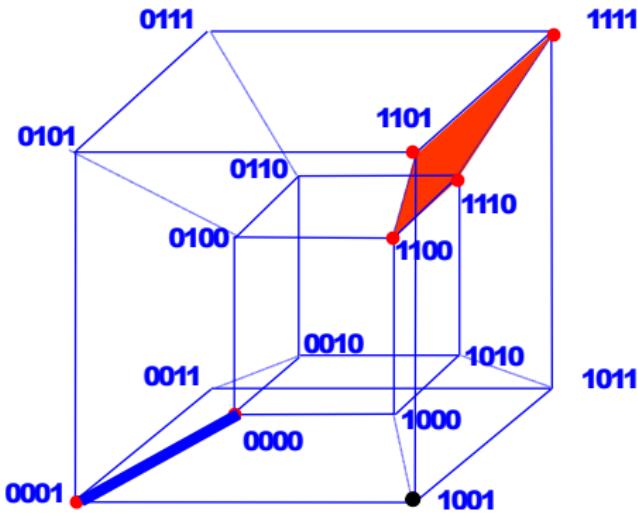


- Monome der Länge  $k$  entsprechen  $(n-k)$ -dimensionalen Teilwürfeln!  *$2^{n-k}$  Elemente von  $\{0, 1\}^n$  werden durch das Monom auf 1 abgebildet.*
- Ein Polynom entspricht einer Vereinigung von Teilwürfeln.
- Beispiel:

$$f(x_1, x_2, x_3, x_4)$$

$$= \boxed{x_1 x_2} + \boxed{x'_1 x'_2 x'_3}$$

$$+ \boxed{x_1 x'_2 x'_3 x_4}$$



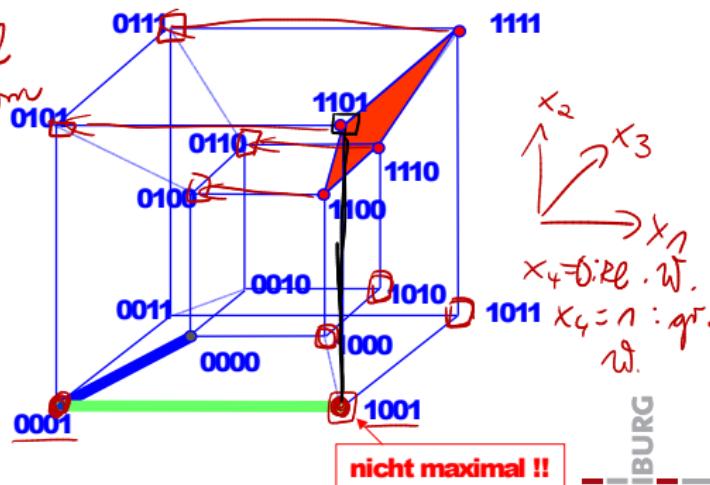
## Zweistufige Logikminimierung als Überdeckungsproblem auf dem Würfel

- **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.

minimale Anzahl  
von Literalen im Monom  
( $\rightarrow$  sekundäre Kosten!)

Entspricht einer Minimallösung:

$$\begin{aligned} & x_1 x_2 + x'_1 x'_2 x'_3 + x'_2 x'_3 x'_4 \\ & + x_1 \bar{x}_2 \bar{x}_3 x_4 \\ & + x_1 \bar{x}_3 x_4 \end{aligned}$$



## Beispiel: 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.

$$\begin{aligned}f_1 &= x_1 x_2 + \overline{x}_1 \overline{x}_2 \overline{x}_3 + \underline{\underline{x}_1 \overline{x}_2 \overline{x}_3 x_4}} \\f_2 &= \underline{\underline{x}_1 \overline{x}_2 \overline{x}_3 x_4}\end{aligned}$$

# Implikanten und Primimplikanten

Aber nicht:

$$\underbrace{x_1}_q \cdot t \leq \underbrace{x_1}_{\delta} \overline{x_2}$$

$$\overline{x_2} \cdot x_3 \leq \overline{x_1} \overline{x_2}$$

## 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  $g$ ).  $ON(f) \subseteq ON(g)$

## Definition

$$ON(g) \subseteq ON(f)$$

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  $q \leq s$ . bzw.  $ON(q) \subseteq ON(s)$ .  
 Implikanten und Primimplikanten können durch  $n$ -dimensionale Würfel veranschaulicht werden.

$$m=4 :$$

$$Bsp.:$$

$$x_1 \overline{x}_2 x_3 \leq x_1 \overline{x}_2$$

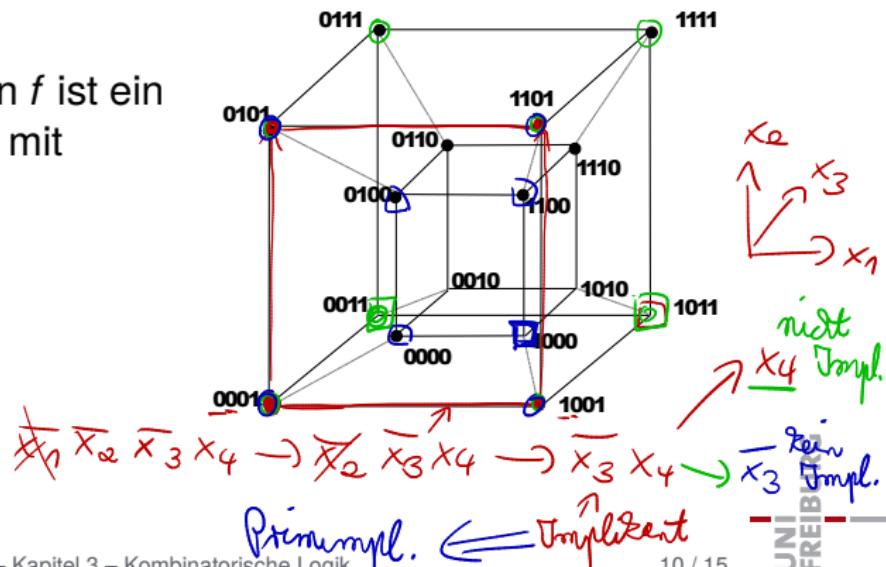
$$ON(x_1 \overline{x}_2 x_3) \subseteq$$

$$ON(x_1 \overline{x}_2)$$

Wann gilt für Monome  $q$  und  $s$ , dass  $q \leq s$  bzw.  $ON(q) \subseteq ON(s)$ ?  
Falls alle Literale aus  $s$ , auch in  $q$  enthalten sind.

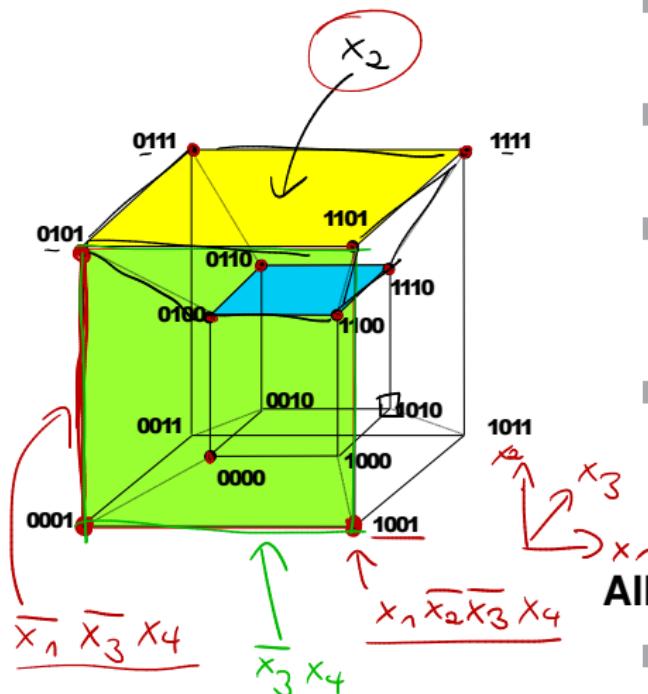
# Veranschaulichung durch Würfel

- 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

$$f: \{0,1\}^4 \rightarrow \{0,1\}$$



## Implikanten

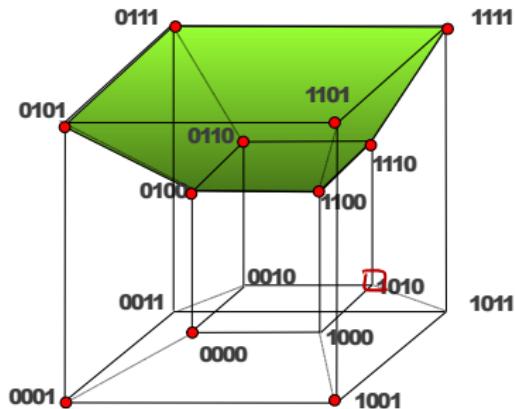
- alle markierten Knoten
  - alle Kanten, deren Ecken alle markiert sind
  - alle Flächen, deren Ecken alle markiert sind
  - alle 3-dimensionalen Würfel, deren Ecken alle markiert sind
  - der 4-dimensionale Würfel, wenn alle Ecken markiert sind

Teilwürfel, deren Ecken alle  
markiert sind

## Allgemein

- Die Implikanten sind die

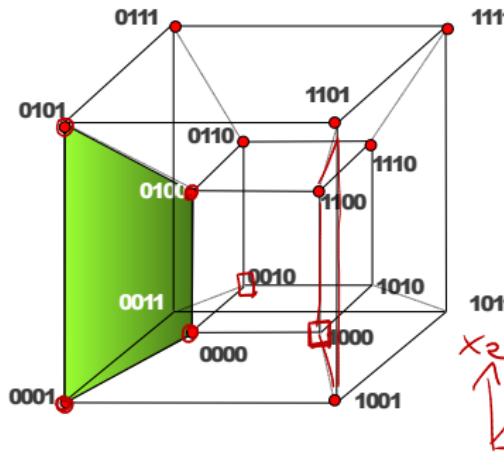
# Bestimmung von Primimplikanten



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

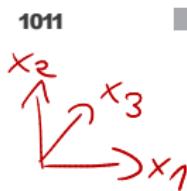
■  $X_2$

# Bestimmung von Primimplikanten

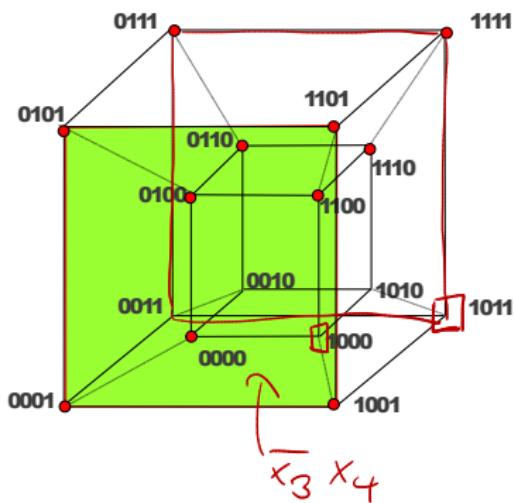


Es gibt 3 Primimplikanten, die durch unseren Würfel definierten Funktion:

- $x_2$
  - $\underline{x'_1} \underline{x'_3}$
- $\overline{x_1}$  kein Impl.  
 $\underline{\overline{x_3}}$  kein Impl.



# Bestimmung von Primimplikanten



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

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

$\bar{x}_3$  kein Impl.  
 $x_4$  kein Impl.

# Polynome und Implikanten einer Funktion $f$

## Lemma

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

**Beweis:** (durch Widerspruch)

- $p = m_1 + \dots + \textcolor{red}{(m_j)} + \dots + m_k$
- Ann.: Es gibt ein Polynom  $p$  von  $f$ , das ein Monom  $m_j$  enthält, welches nicht Implikant von  $f$  ist, d.h. es gilt nicht:  $\psi(m_j) \leq f$  ist  $\psi(m_j) \subseteq ON(\psi(m_j)) \subseteq ON(f)$
  - Das heißt es gibt eine Belegung  $(\alpha_1, \dots, \alpha_n)$  der Variablen  $(x_1, \dots, x_n)$  mit
    - $f(\alpha_1, \dots, \alpha_n) = 0$
    - $\psi(m_j)(\alpha_1, \dots, \alpha_n) = 1$ , also auch  $\psi(p)(\alpha_1, \dots, \alpha_n) = 1$

Demnach ist  $\psi(p) \neq f$ , also  $p$  kein Polynom von  $f$ .

⇒ **Widerspruch!**

# Kapitel 3 – Kombinatorische Logik

1. Kombinatorische Schaltkreise
2. Boolesche Algebren
3. Boolesche Ausdrücke, Normalformen, zweistufige Synthese
- 4. Berechnung eines Minimalpolynoms**
  - 4.1 Quine-McCluskey
  - 4.2 Überdeckungsproblem
5. Arithmetische Schaltungen
6. Anwendung: ALU von ReTI

Albert-Ludwigs-Universität Freiburg

Prof. Dr. Christoph Scholl

Institut für Informatik  
Sommersemester 2023

# 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:

$$\begin{aligned} p &= m_1 + m_2 + \dots + m_{i-1} + \boxed{m_i} + m_{i+1} + \dots + m_k \\ p' &= m_1 + m_2 + \dots + m_{i-1} + \boxed{m'_i} + m_{i+1} + \dots + m_k \end{aligned}$$

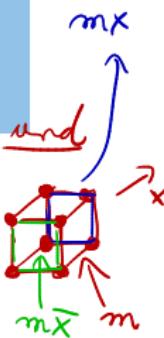
- Nehme an, dass  $p$  einen nicht primen Implikanten  $m$  von  $f$  enthält.
- $m$  wird durch einen Primimplikanten  $m'$  von  $f$  überdeckt, ist also in  $m'$  enthalten.
- Es gilt demnach  $\underline{\text{cost}}(m') < \underline{\text{cost}}(m)$ .
- Ersetzt man in  $p$  den Implikanten  $m$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

# Berechnung von Implikanten

## Lemma 1

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

**Beweis:**  $\text{Bsp.: } m = x_1 x_3 \bar{x}_4 \text{ Impl.} \Rightarrow x_1 \bar{x}_2 x_3 \bar{x}_4 \text{ Impl. und}$   
 $x_1 x_2 x_3 \bar{x}_4 \text{ Impl.}$



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

## Lemma 2

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

**Beweis:**  $\text{ON}(mx) \subseteq \text{ON}(f)$ ,  $\text{ON}(m\bar{x}) \subseteq \text{ON}(f)$   
 $\Rightarrow \text{ON}(mx + m\bar{x}) \subseteq \text{ON}(f)$

- $f \geq m \cdot x + m \cdot x' = m \cdot \underbrace{(x+x')}_{1} = m \quad (\text{bzw. } \text{ON}(f) \supseteq \text{ON}(m))$

# 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.

$\Leftrightarrow$ :  $m$  ist ein Minterm und  $m$  ist Implikant von  $f$  bel.

### ■ Äquivalente Schreibweise:

$$m \in \underline{\text{Implikant}}(f)$$

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

### ■ Beweis folgt unmittelbar aus Lemma 1 und Lemma 2.

" $\Rightarrow$ "      " $\Leftarrow$ "

# 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

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

**begin**

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

$i := 0;$

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

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

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

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

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

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

$i := i + 1;$  bzw.  $\{m' \mid m' \in L_i \text{ und } \exists \text{ kein } q \in L_{i+1} \text{ mit}$

**end while;**

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

**end;**

für den Spezialfall, dass die while-Schleife mit  $i = n$  beendet wird und  $L_n = \{1\}$  ist.

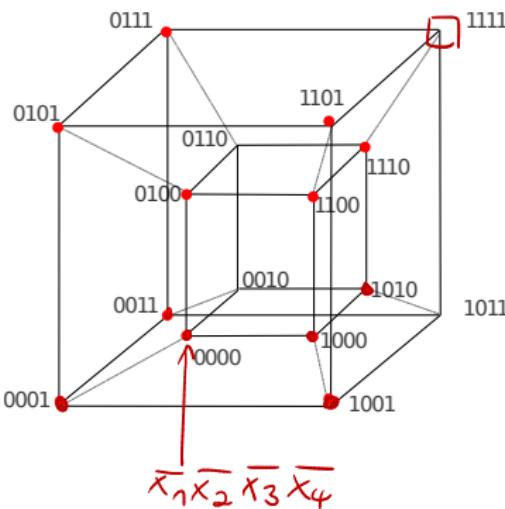
# Verbesserung durch Mccluskey

---

- Vergleiche nur **Monome** untereinander
  - welche 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  $L_i^M$ , mit  $M \subseteq \{x_1, \dots, x_n\}$  und  $|M| = n - i$ .
  - $L_i^M$  enthält die Implikanten aus  $L_i$ , deren Literale alle aus  $M$  sind.
  - Anordnung der Monome in  $L_i^M$  gemäß der Anzahl der positiven Literale.

# Beispiel Quine-McCluskey

$$f: \{0,1\}^4 \rightarrow \{0,1\}$$



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

$\overbrace{\phantom{0000}}$	$0000$	steht für $x'_1 x'_2 x'_3 x'_4$
$\overbrace{\phantom{0001}}$	$0001$	
$\overbrace{\phantom{0100}}$	$0100$	
$\overbrace{\phantom{1000}}$	$1000$	
$\overbrace{\phantom{0011}}$	$0011$	
$\overbrace{\phantom{0101}}$	$0101$	
$\overbrace{\phantom{1001}}$	$1001$	
$\overbrace{\phantom{0110}}$	$0110$	
$\overbrace{\phantom{1100}}$	$1100$	
$\overbrace{\phantom{0111}}$	$0111$	
$\overbrace{\phantom{1111}}$	$1111$	

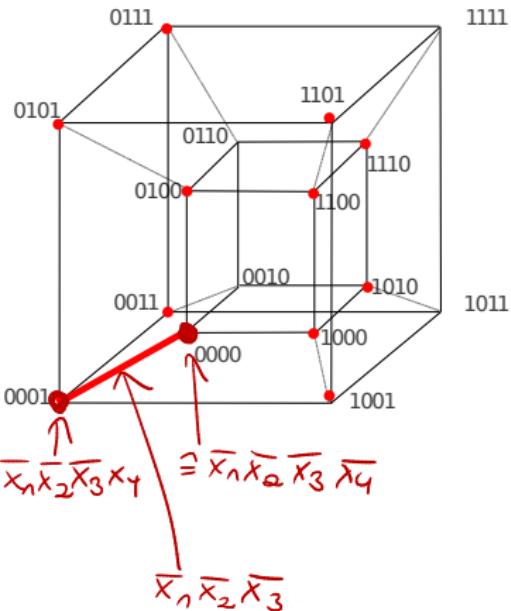
1 pos.  
fit.

2 pos.  
fit.

3 pos.  
fit.

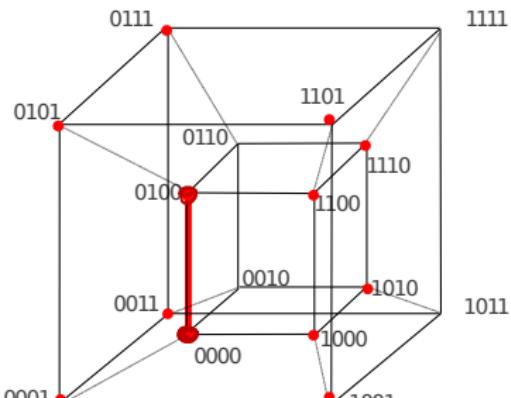
Vergleiche im Folgenden nur Monome aus benachbarten Blöcken!

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



	$L_0^{\{x_1, x_2, x_3, x_4\}}$ :	$L_1^{\{x_1, x_2, x_3\}}$ :
→	0 0 0 0	$\overbrace{\overline{x_1} \overline{x_2} \overline{x_3} x_4}$
→	0 0 0 1	$\overbrace{\overline{x_1} \overline{x_2} \overline{x_3} x_4}$
	0 1 0 0	$m$
	1 0 0 0	$x$
	0 0 1 1	
	0 1 0 1	
	1 0 0 1	
	1 0 1 0	
	1 1 0 0	
	0 1 1 1	
	1 1 0 1	
	1 1 1 0	

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



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

0 0 0 0	$\rightarrow \bar{x}_1 \bar{x}_2 \bar{x}_3 \bar{x}_4$
0 0 0 1	
0 1 0 0	$\rightarrow \bar{x}_1 \bar{x}_2 \bar{x}_3 \bar{x}_4$
1 0 0 0	
0 0 1 1	
0 1 0 1	
1 0 0 1	
1 0 1 0	
1 1 0 0	
0 1 1 1	
1 1 0 1	
1 1 1 0	

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

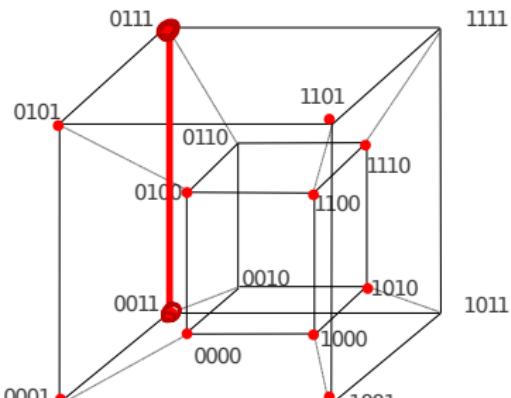
0 0 0 -	
0 0 1 1	
0 1 0 1	
1 0 0 1	
1 0 1 0	
1 1 0 0	
0 1 1 1	
1 1 0 1	
1 1 1 0	

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

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

Annotations in red highlight circled terms:  $\bar{x}_1 \bar{x}_2 \bar{x}_3 \bar{x}_4$ ,  $\bar{x}_1 \bar{x}_2 \bar{x}_3 \bar{x}_4$ , and  $\bar{x}_2 \bar{x}_3 \bar{x}_4$ . A blue arrow points from the first term to the second row of the  $L_0$  table. Another blue arrow points from the third term to the third row of the  $L_1$  table. A blue arrow also points from the third term to the first row of the  $L_1^{\{x_1, x_3, x_4\}}$  table.

# 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
<u>0 0 1 1</u>
0 1 0 1
1 0 0 1
1 0 1 0
1 1 0 0
<u>0 1 1 1</u>
1 1 0 1
1 1 1 0

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

0 0 0 -

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

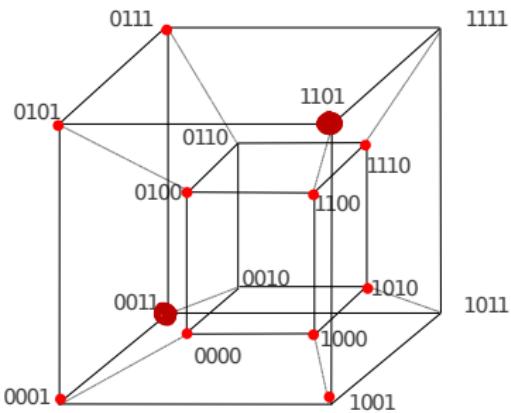
0 - 0 0

0 - 1 1

$\bar{x}_1 \bar{x}_2 x_3 x_4$

$\bar{x}_1 x_3 x_4$

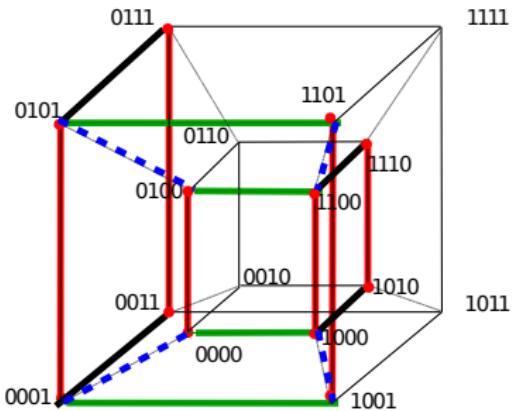
## Beispiel Quine-McCluskey: Bestimmung von $L_1$ (4/4)



$L_0^{\{x_1, x_2, x_3, x_4\}}:$	$L_1^{\{x_1, x_2, x_3\}}:$	$L_1^{\{x_1, x_3, x_4\}}:$
0 0 0 0	0 0 0 -	0 - 0 0
0 0 0 1		0 1 1
0 1 0 0		
1 0 0 0		
0 0 1 1 $\overline{x_1} \overline{x_2} \overline{x_3} x_4$	0 1 0 1 $\underline{x_1} \underline{x_2} \underline{x_3} x_4$	0 - 0 0
0 1 0 1	1 0 0 1	0 1 1
1 0 0 1	1 0 1 0	
1 0 1 0	1 1 0 0	
1 1 0 0	0 1 1 1	
0 1 1 1	1 1 0 1 $x_1 x_2 \overline{x_3} x_4$	
1 1 1 0		

Nicht kürzbar, da nicht  
Ecken der gleichen Kante.

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



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

→	0 0 - 1	{ 1 pos. lit.
1 0 - 0		
0 1 - 1		

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

0 0 0 -	↙
0 1 0 -	
1 0 0 -	

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

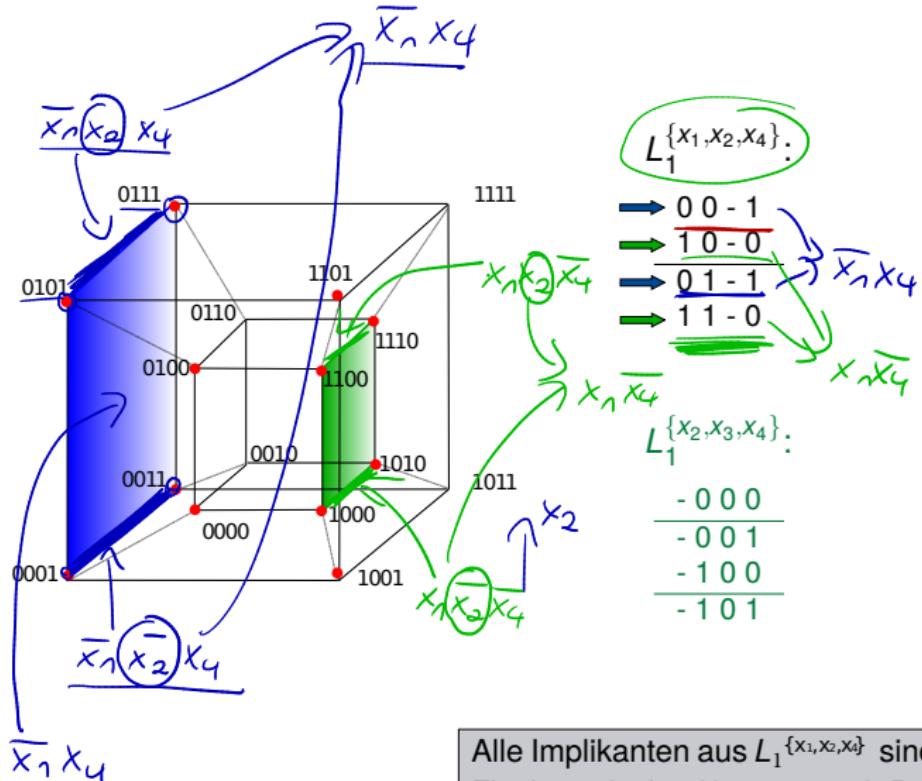
- 0 0 0	2 pos. lit.
- 0 0 1	
- 1 0 0	
- 1 0 1	

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

0 - 0 0	→
0 - 0 1	
1 - 0 0	
0 - 1 1	
1 - 0 1	

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

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



Alle Implikanten aus  $L_1^{\{x_1, x_2, x_3\}}$  sind Kanten von Flächen, die Implikanten sind:  $Prim(f) = \emptyset$

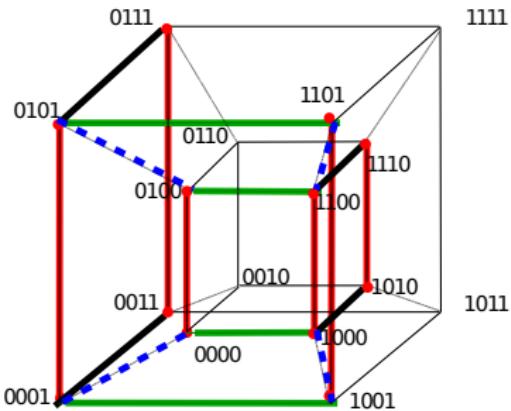
$$L_1^{\{x_1, x_2, x_3\}}.$$

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

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

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

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



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

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

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

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

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

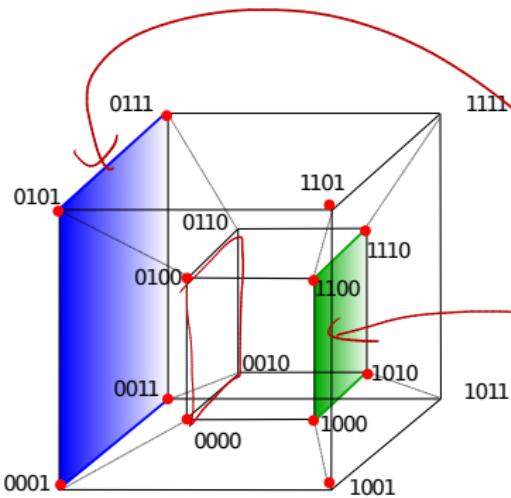
$$\begin{array}{r} -000 \\ -001 \\ -100 \\ -101 \end{array}$$

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

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

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)



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



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

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

$$\begin{array}{l} \cancel{x_1} \cancel{x_4} \\ x_1 \cancel{x_4} \end{array}$$

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

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

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

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

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

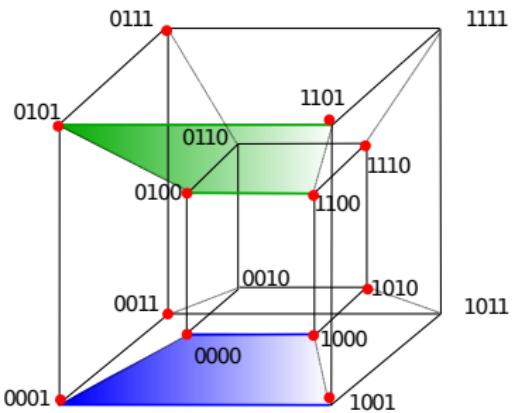


$$L_2^{\{x_3, x_4\}}:$$

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

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

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



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

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

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

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

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

$$\begin{array}{r} 0 \dots 0 \dots \\ 1 \dots 0 \dots \end{array} \Rightarrow \begin{array}{r} \dots \\ \dots \end{array}$$

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

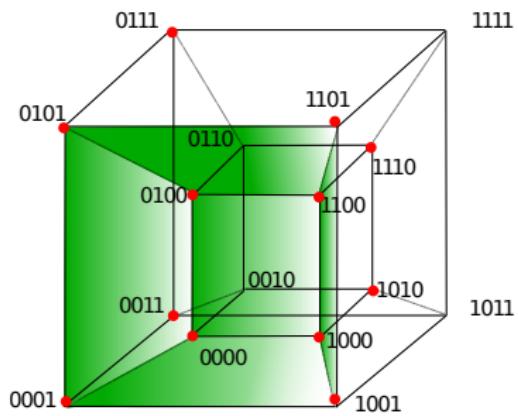
$$\begin{array}{r} -00\dots \\ -10\dots \end{array} \leftarrow \begin{array}{r} \overbrace{x_2}^{\textcolor{blue}{-}} \overbrace{x_3}^{\textcolor{red}{-}} \\ \overbrace{x_2}^{\textcolor{green}{-}} \overbrace{x_3}^{\textcolor{red}{-}} \end{array} \rightarrow \begin{array}{r} \overbrace{x_3}^{\textcolor{red}{-}} \end{array}$$

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

$$\begin{array}{r} -00\dots \\ -01\dots \end{array} \rightarrow \begin{array}{r} \overbrace{x_3}^{\textcolor{red}{-}} \end{array}$$

Die markierten Implikanten-Flächen sind Rand eines 3-dimensionalen Implikanten. Sie sind also nicht prim!  $\Rightarrow \text{Prim}(f) = \{x'_1 x_4, x_1 x'_4\}$

# Beispiel Quine-McCluskey: Ende



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

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

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

$$L_3^{\{x_4\}}:$$

-- 0 --  $\hookleftarrow \overline{x_3}$

$$\text{Prim}(f) = \underline{x_1'x_4}, \underline{x_1x_4'}$$

$$\Rightarrow \text{Prim}(f) = \underline{x_1'x_4}, \underline{x_1x_4'}, \underline{x_3'}$$

$$p_{complete}(f) = \underline{x_1'x_4} + \underline{x_1x_4'} + \underline{x_3'}$$

# Korrektheit von Quine-McCluskey (1/2)

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

**begin**

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

$i := 0;$

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

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

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

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

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

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

    |  $i := i + 1;$

**end while;**

**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  $3^n$  verschiedene Monome in  $n$  Variablen.

$$x_1 \ x_2 \ \dots \ x_n$$

$\uparrow$        $\uparrow$        $\uparrow$

**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  $O(\log^2(N) \cdot N^{\log_2(3)})$ , wobei  $N = 2^n$  die Größe der Funktionstabelle ist.

$$\log_2(3) \approx 1.58$$

$$m = \underline{x_1 \bar{x}_2 x_3 x_4}$$

~~$\begin{array}{cccc} \bar{x}_1 & \bar{x}_2 & x_3 & x_4 \\ x_1 & x_2 & x_3 & x_4 \\ x_1 x_2 & \bar{x}_3 & x_4 \\ \bar{x}_1 \bar{x}_2 x_3 & \bar{x}_4 \end{array}$~~

### Beweisidee:

Jedes der  $3^n$  Monome wird im Verlauf des Verfahrens mit höchstens  $n$  anderen Monomen verglichen.  $O(3^n \cdot n \cdot n)$

- Gegeben sei ein Monom  $mx$ . Die Erzeugung von  $mx'$  und die Suche nach  $mx'$  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:

$$\underline{N = 2^n} \Rightarrow n = \log_2 N$$

$$3^n = (2^{\log_2(3)})^n = (2^n)^{\log_2(3)} = \underline{N^{\log_2(3)}}$$

$$3 = 2^{\log_2(3)}$$

$$(x^a)^b = x^{a \cdot b} = (x^b)^a$$



# Kapitel 3 – Kombinatorische Logik

1. Kombinatorische Schaltkreise
2. Boolesche Algebren
3. Boolesche Ausdrücke, Normalformen, zweistufige Synthese
- 4. Berechnung eines Minimalpolynoms**
  - 4.1 Quine-McCluskey
  - 4.2 Überdeckungsproblem**
5. Arithmetische Schaltungen
6. Anwendung: ALU von ReTI

# 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: Ein 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

---

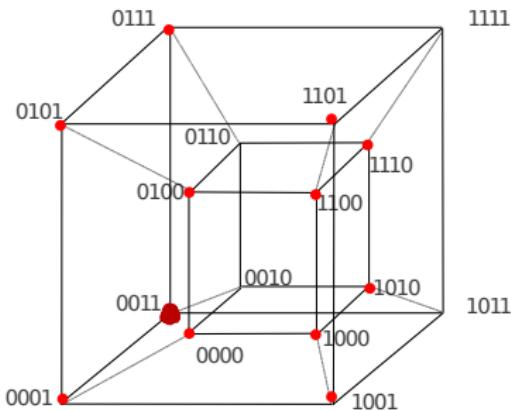
- Definiere eine boolesche Matrix  $PIT(f)$ , die Primimplikantentafel von  $f$ :
  - Die **Zeilen** entsprechen eindeutig den Primimplikanten von  $f$ .
  - Die **Spalten** entsprechen eindeutig den Mintermen von  $f$  bzw. den Elementen von  $ON(f)$ .
  - Sei  $min(\alpha)$  ein beliebiger Minterm von  $f$ . *f.kw. ein Bel.  $\alpha \in ON(f)$* : 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.  $\forall \alpha \in ON(f) \quad \exists m \in M$  mit  $PIT(f)[m, min(\alpha)] = 1$ .

# Primimplikantentafel: Beispiel (1/2)

$$f: \{0, 1\}^4 \rightarrow \{0, 1\}$$



$$\text{Prim}(f) = \{\underline{x'_1x_4}, \underline{x_1x'_4}, \underline{x'_3}\}$$

$$\overline{x_1} \overline{x_2} \overline{x_3} \overline{x_4}$$

/

0000      0011

$$\overline{x_1} x_2 x_3 x_4$$

/

0111

Primimplikantentafel PIT(f):

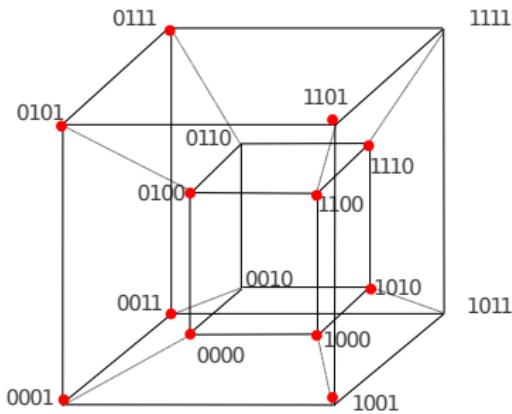
	0	1	3	4	5	7	8	9	10	12	13	14	$\text{DNF}(f)$
$x'_1x_4$	0	1	1	0	1	1	--	-	-	-	-	-	$(1, 2)$
$x_1x'_4$	0	0	0	0	0	0	1	-	1	1	1	1	$(1, 2)$
$x'_3$	1	1	0	1	1	0	1	1	1	1	1	1	$(1, 1)$

## 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.  $\forall \alpha \in \text{ON}(f) \quad \exists m \in M \text{ mit } \text{PIT}(f)[m, \underline{\min(\alpha)}] = 1$ .



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

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

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

# Erste Reduktionsregel - Wesentlicher Implikant

## Definition

Ein Primimplikant  $m$  von  $f$  heißt **wesentlich**, wenn es ein  $\alpha \in ON(f)$  gibt, das nur von diesem Primimplikanten überdeckt wird, also:

- $PIT(f)[\underline{m}, \underline{\min(\alpha)}] = 1$
- $PIT(f)[\underline{m'}, \underline{\min(\alpha)}] = 0$

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)

*Hinweise: Wir brauchen P. i. 1, 2, 3, 4.*

*Wesentliche P. i.*

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

## 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	

Spalte 17 dominiert Spalte 10

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								1
6		1							1
7			1						
8				1					
9	1				1				1
10		1				1			1
11			1				1		
12				1				1	
13					1	1	1	1	

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

Zeile 9 dominiert Zeile 5 ⇒ Streiche Zeile 5, falls diese nicht billiger ist als Zeile 9.

# Dritte Reduktionsregel - Zeilendominanz

## 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  $\text{cost}(m') \geq \text{cost}(m)$  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			
10						1		
11							1	
12								1
13								

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

# Nach Anwendung der 3. Reduktionsregel

*Vermerk: Nehme P. i. 9, 10, 11, 12*

	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:  
 $1 + 2 + 3 + 4 + 9 + 10 + 11 + 12$

# Ein weiteres Beispiel

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

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

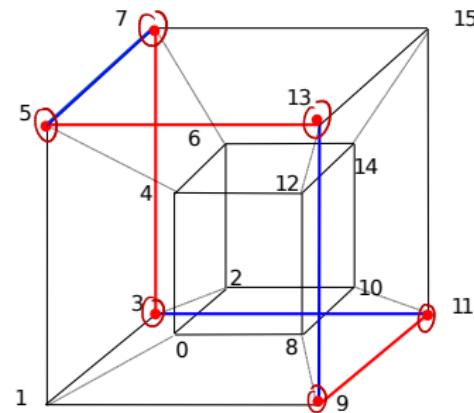
Primimplikantentafel  $\text{PIT}(f)$ :

	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			

# Ein weiteres Beispiel

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

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



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			

Kein Primimplikant ist wesentlich! und auch Reduktionsregeln 2 und 3 sind nicht anwendbar.

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

# Zyklische Überdeckungsprobleme

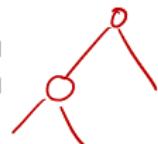
## Definition

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.

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

- Die gesuchte minimale Überdeckung ist gegeben durch das Monom, das einer PI-Auswahl mit minimalen Kosten entspricht.

„ $a=1$ “ bedeutet „ $\{7,5\}$  wird im Ergebnis genommen“  
„ $a=0$ “ bedeutet „ $\{7,5\}$  wird im Ergebnis nicht genommen“

	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		

1. Spalte 2. Spalte wird übersetzt in  
 $(e+f) \cdot (a+b) \cdot (a+f) \cdot (c+d) \cdot (d+e) \cdot (b+c)$   
 $= (ea + eb + fa + fb) \cdot (ac + ad + fc + fd)$   
 $\cdot (db + dc + eb + ec)$

$\vdots$

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

Jede 1-Stelle dieses OR-AND-Polynoms entspricht einer gültigen Überdeckung aller 1-Stellen von f denk P.i.

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!

# Vergleich Schaltkreise, boolesche Polynome

---

- Sowohl Schaltkreise als auch boolesche Polynome 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. Boolesche Algebren
3. Boolesche Ausdrücke, Normalformen, zweistufige Synthese
4. Berechnung eines Minimalpolynoms
- 5. Arithmetische Schaltungen**
  - 5.1 Carry-Ripple-Addierer
  - 5.2 CSA, 2er-Komplement, Subtraktion
6. Anwendung: ALU von ReTI

# Arithmetische Schaltungen

---

- Addieren nach der Schulmethode: Carry-Ripple-Addierer.
- Effizienteres Addieren: Conditional-Sum-Addierer.
- Addition von Zweierkomplement-Zahlen.
- Subtrahierer.

# 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.

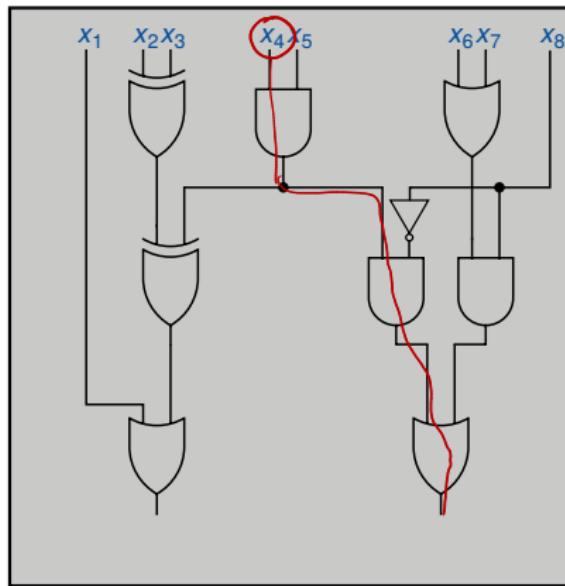
## Definition

Die Tiefe  $\text{depth}(SK)$  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

gatto aus Bibliothek  $BIB = B_1 \cup B_2$

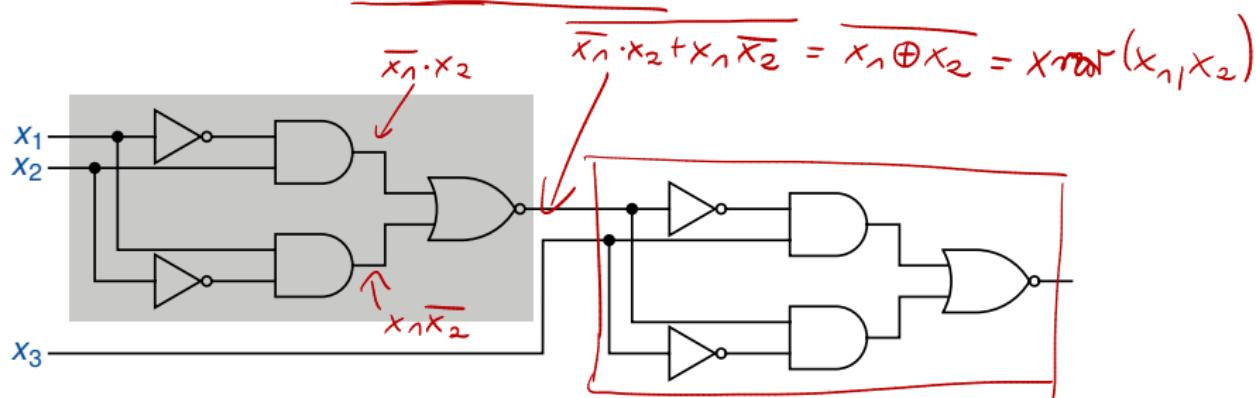


$$C(SK) = \underline{9}$$

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

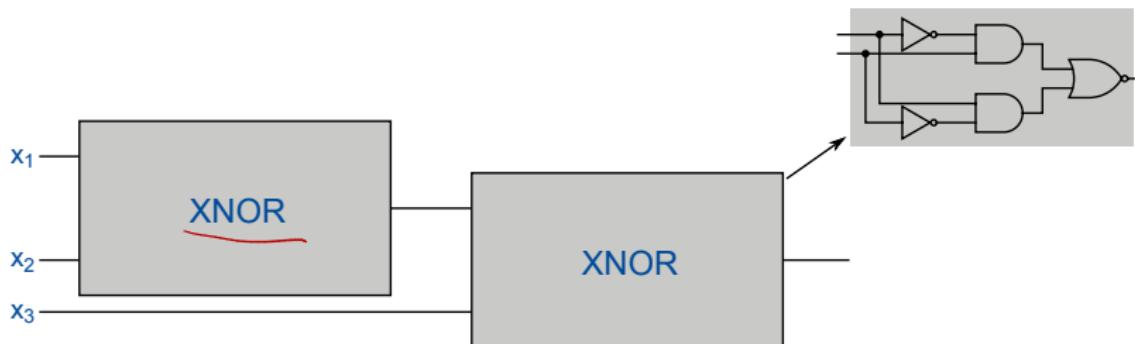
# Teilschaltkreise, hierarchischer Entwurf (informell)

## ■ 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

---

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

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

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

# Addierer für nichtnegative Zahlen

$$\sum_{i=0}^{n-1} a_i \cdot 2^i$$

## ■ 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  $\boxed{s}$  von  $\langle a \rangle + \langle b \rangle + c$  berechnet.

■ 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:

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

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

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

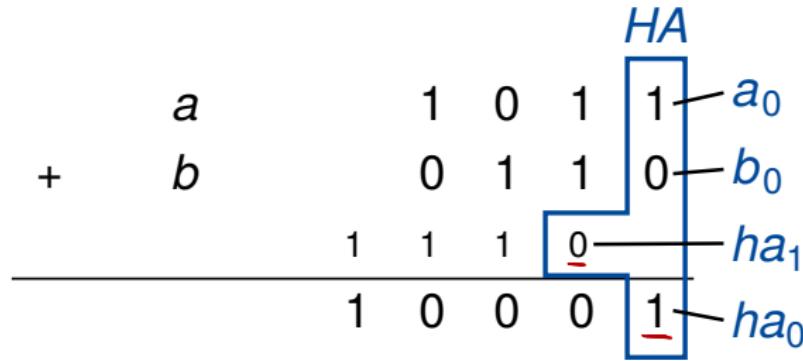
# Addieren nach der Schulmethode (1/4)

---

- 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:

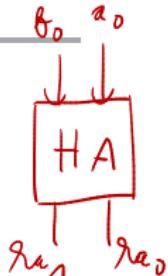
$$\begin{array}{r} 4378 \\ + 5613 \\ \hline 9991 \end{array}$$
  
$$\begin{array}{r} a & 1011 \\ + b & 0110 \\ \hline 1110 \\ \hline 10001 \end{array}$$

## Addieren nach der Schulmethode (2/4)



$a_0$	$b_0$	$ha_1$	$ha_0$	
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	2

# 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

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

# Schaltkreis eines Halbaddierers

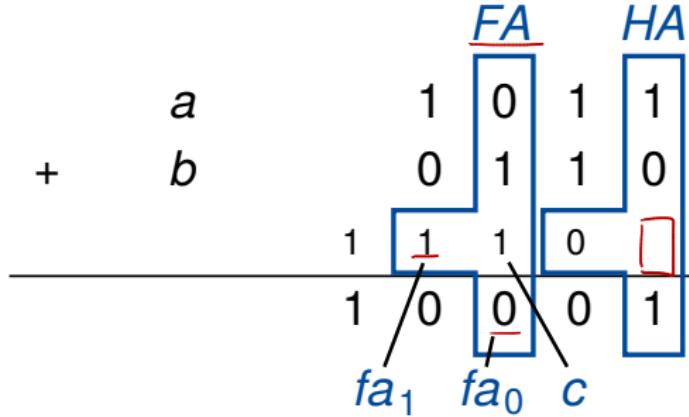
---



dabei gilt:

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

## Addieren nach der Schulmethode (3/4)



# Volladdierer (Full Adder, FA)



$$\begin{array}{c} a_0 \\ b_0 \\ \hline fa_1 & fa_0 \\ c \end{array}$$

- 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  $2fa_1 + fa_0 = a_0 + b_0 + c$

$$x_{n+1} \dots x_m = \begin{cases} 1, \text{ falls } \sum_{i=1}^n x_i = m \\ 0, \text{ sonst } \end{cases}$$

$a_0$	$b_0$	$c$	$fa_1$	$fa_0$	
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	1	1
0	1	1	1	0	2 ↲
1	0	0	0	1	1
1	0	1	1	0	2 ↲
1	1	0	1	0	2 ↲
1	1	1	1	1	3 ↲

$$\begin{aligned} fa_0(a_0, b_0, c) &= \underline{\text{exor}}_3(a_0, b_0, c) = \underline{a_0 \oplus b_0} \oplus c = \underline{(a_0 \oplus b_0)} \oplus c \\ fa_1(a_0, b_0, c) &= a_0 b_0 + a_0 c + b_0 c = \underline{a_0 b_0} + c \underline{(a_0 + b_0)} \\ &= \underline{a_0 b_0} + c \underline{(a_0 \oplus b_0)} \end{aligned}$$

# Volladdierer als Funktion von HAs

Aus der Tabelle folgt:

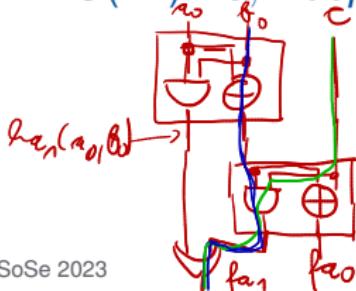
$$\begin{aligned}fa_0 &= \underline{a_0 \oplus b_0 \oplus c} = (\underline{a_0 \oplus b_0}) \oplus \underline{c} \\&= ha_0(c, ha_0(a_0, b_0))\end{aligned}$$

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

$a_0$	$b_0$	$c$	$fa_1$	$fa_0$
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

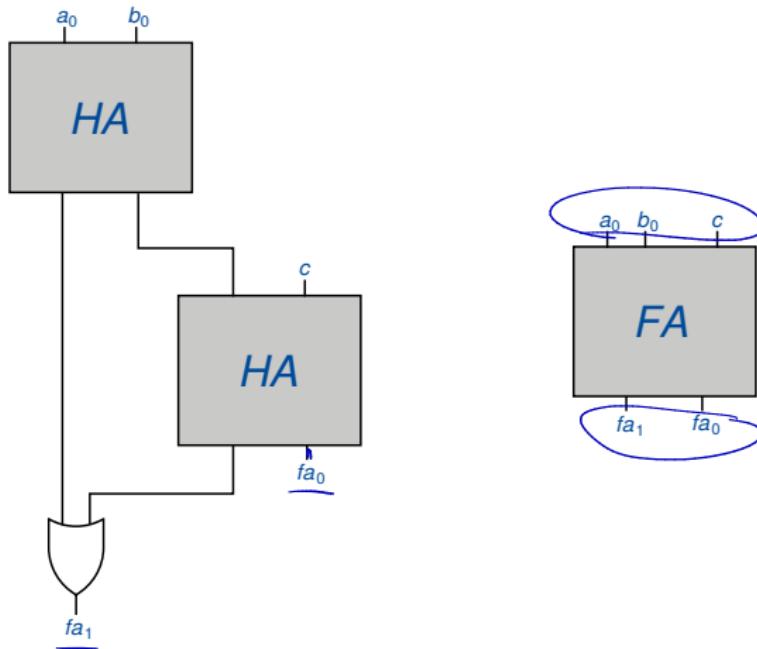
Kosten und Tiefe eines FA:

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

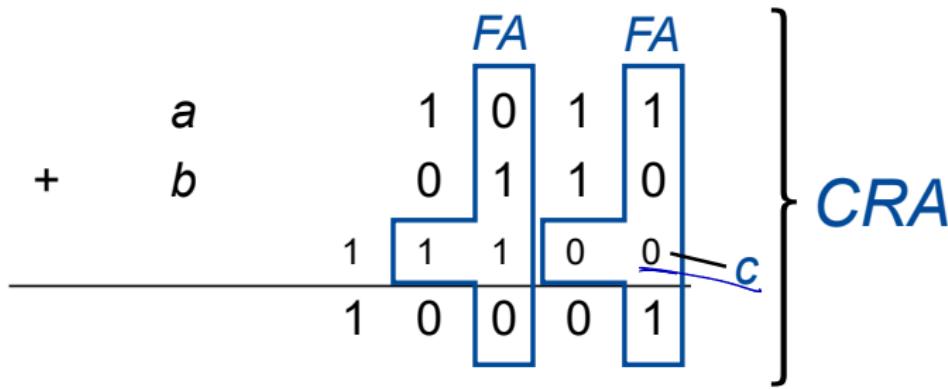


# Schaltkreis eines Volladdierers

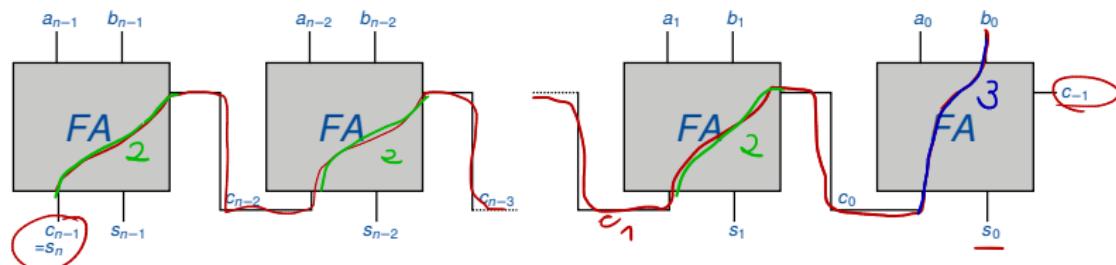
---



## Addieren nach der Schulmethode (4/4)



# Aufbau eines Carry-Ripple-Addierers



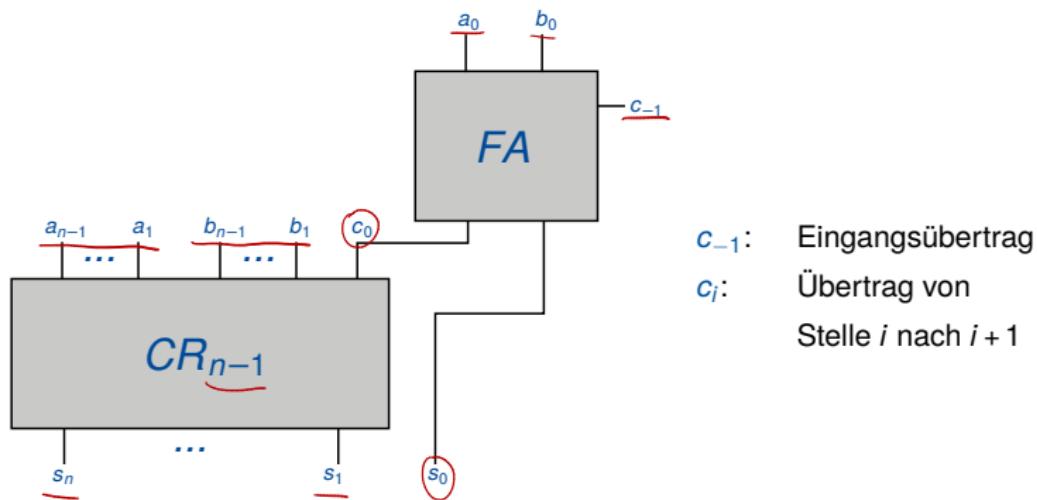
$$\begin{array}{r} a_{n-1} \ a_{n-2} \ \dots \ a_2 \ a_1 \ a_0 \\ b_{n-1} \ b_{n-2} \ \dots \ b_2 \ b_1 \ b_0 \\ \hline c_{n-1} \ c_{n-2} \ c_{n-3} \ \dots \ c_1 \ c_0 \ c_{-1} \end{array}$$

$$\begin{array}{r} a_{n-1} \ a_{n-2} \ a_{n-3} \ \dots \ a_1 \ a_0 \\ \parallel \\ a_{n-1} \ a_{n-2} \ a_{n-3} \ \dots \ a_1 \ a_0 \\ \hline c_{n-1} \end{array}$$

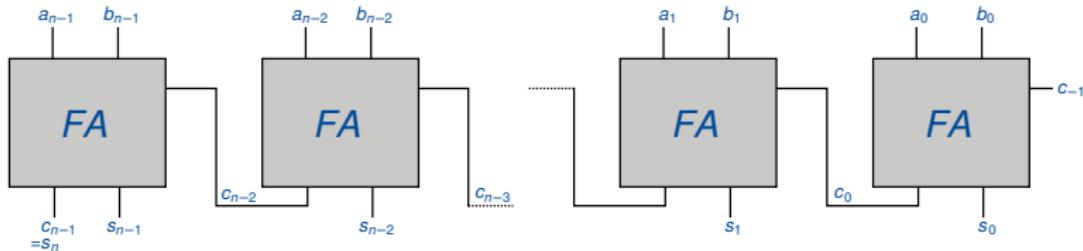
# Induktive Definition des Carry-Ripple-Addierers $CR_n$

---

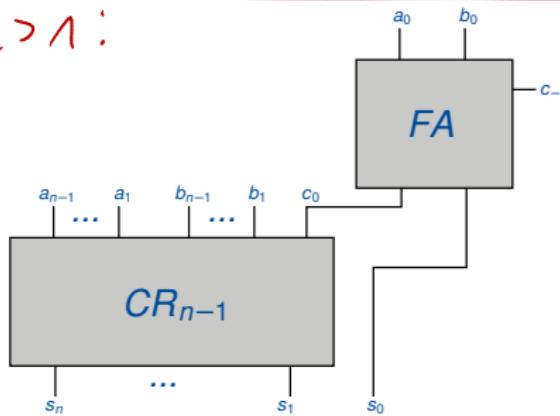
- Für  $n = 1$  :  $CR_1 = FA$
- Für  $n > 1$  : Folgender Schaltkreis:



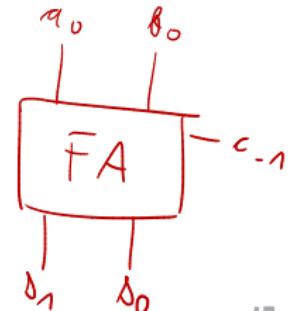
# Zwei (identische) Darstellungen von $CR_n$



Für  $n > 1$ :



Für  $n = 1$



# 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:  $\langle s_n \dots s_0 \rangle = 2 \cdot \langle s_n \dots s_1 \rangle + s_0$   
 $\stackrel{(a)}{=} 2 \cdot (\langle a_{n-1} \dots a_1 \rangle + \langle b_{n-1} \dots b_1 \rangle + c_0) + s_0$   
 $= 2 \cdot (\langle a_{n-1} \dots a_1 \rangle + \langle b_{n-1} \dots b_1 \rangle) + 2 \cdot c_0 + s_0$   
 $\stackrel{(b)}{=} 2 \cdot \langle a_{n-1} \dots a_1 \rangle + a_0 + 2 \cdot \langle b_{n-1} \dots b_1 \rangle + b_0 + c_{-1}$   
 $= \langle a \rangle + \langle b \rangle + c_{-1}$

# Schaltbild und Komplexität von $CR_n$

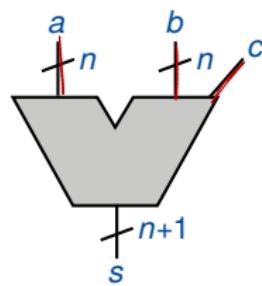
---

- $C(CR_n) = n \cdot \underbrace{C(FA)}_{5} = 5n.$
- $depth(CR_n) = ?.$

# Schaltbild und Komplexität von $CR_n$

---

- $C(CR_n) = n \cdot C(FA) = 5n.$
- $depth(CR_n) = \underline{3 + 2(n - 1)} = 2n + 1$
- Sowohl die Kosten als auch die Tiefe von  $CR_n$  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**.





# Kapitel 3 – Kombinatorische Logik

1. Kombinatorische Schaltkreise
2. Boolesche Algebren
3. Boolesche Ausdrücke, Normalformen, zweistufige Synthese
4. Berechnung eines Minimalpolynoms
- 5. Arithmetische Schaltungen**
  - 5.1 Carry-Ripple-Addierer
  - 5.2 CSA, 2er-Komplement, Subtraktion
6. Anwendung: ALU von ReTI

# $n$ -Bit-Inkrementer

## Definition

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

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

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

- Ein Inkrementer ist ein Addierer mit  $b_i = 0$  für alle  $i$ .  
⇒ Ersetze in  $CR_n$  die FA durch HA.

- Kosten und Tiefe:

- $C(INC_n) = n \cdot \underline{C(HA)} = \underline{2n}$

- $depth(INC_n) = n \cdot \underline{depth(HA)} = \underline{n}$

$\begin{array}{r} 11\dots 11 \\ \hline 0 \dots 00 \\ \hline \dots c_2 c_1 c_0 \end{array}$

$\begin{array}{r} a_{n-1} \dots a_1 a_0 \\ 0 \dots 00 \\ \hline \dots c_2 c_1 c_0 \end{array}$

$\begin{array}{r} a_{n-1} \dots a_1 a_0 \\ 0 \dots 00 \\ \hline \dots c_2 c_1 c_0 \end{array}$

$\begin{array}{r} a_{n-1} \dots a_1 a_0 \\ 0 \dots 00 \\ \hline \dots c_2 c_1 c_0 \end{array}$

$\begin{array}{r} a_{n-1} \dots a_1 a_0 \\ 0 \dots 00 \\ \hline \dots c_2 c_1 c_0 \end{array}$

# $n$ -Bit-Multiplexer

## Definition

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

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

*select-Signal*

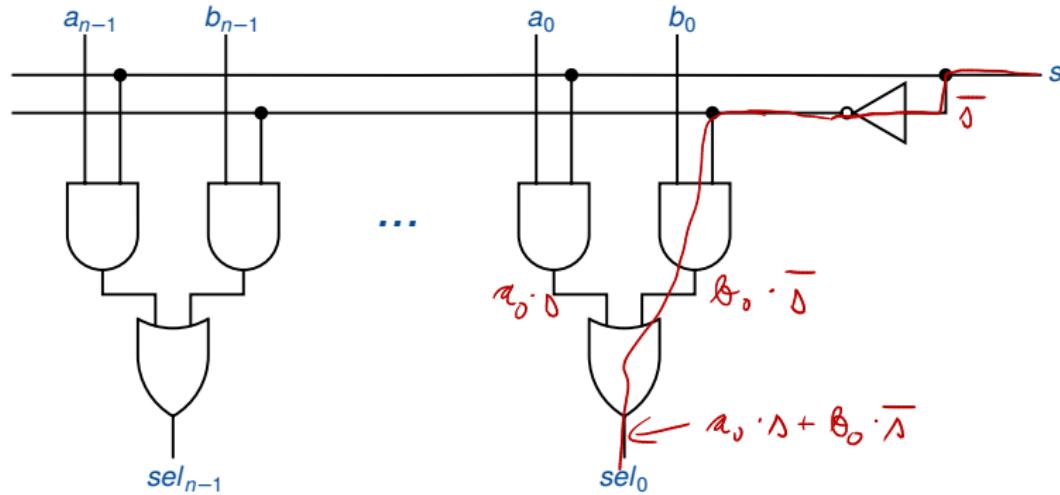
$$sel_n(a_{n-1}, \dots, a_0, b_{n-1}, \dots, b_0, s) = \begin{cases} (a_{n-1} \dots a_0), & \text{falls } s = 1 \\ (b_{n-1} \dots b_0), & \text{falls } s = 0 \end{cases}$$

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

$$s=1 : s \cdot a_i + \bar{s} \cdot b_i = \underbrace{1 \cdot a_i}_{a_i} + \underbrace{\bar{1} \cdot b_i}_{0} = a_i$$

$$s=0 : s \cdot a_i + \bar{s} \cdot b_i = \underbrace{0 \cdot a_i}_{0} + \underbrace{\bar{0} \cdot b_i}_{0} = b_i$$

# Aufbau von $MUX_n$

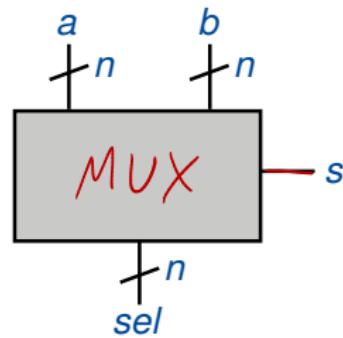


# Schaltbild und Kosten $MUX_n$

---

Kosten und Tiefe:

$$C(MUX_n) = \underline{3n + 1}.$$
$$\underline{depth(MUX_n) = 3.}$$



# Rückkehr zum Addierer

Gibt es billigere Addierer als  $CR_n$ ?

Notation: Sei  $f \in \mathbb{B}_n$ .

Dann sind  $C(f)$  und  $depth(f)$  wie folgt definiert:

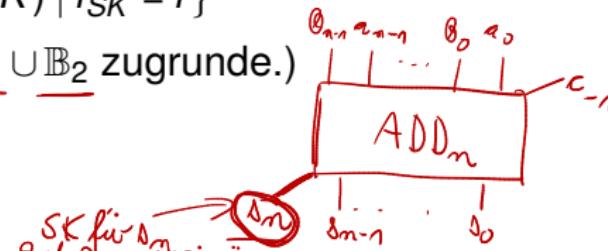
$$C(f) := \min\{C(SK) \mid f_{SK} = f\}$$

$$depth(f) := \min\{depth(SK) \mid f_{SK} = f\}$$

(Wir legen hier die Bibliothek  $BIB = \underline{\mathbb{B}_1} \cup \underline{\mathbb{B}_2}$  zugrunde.)

**Untere Schranken:**

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



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

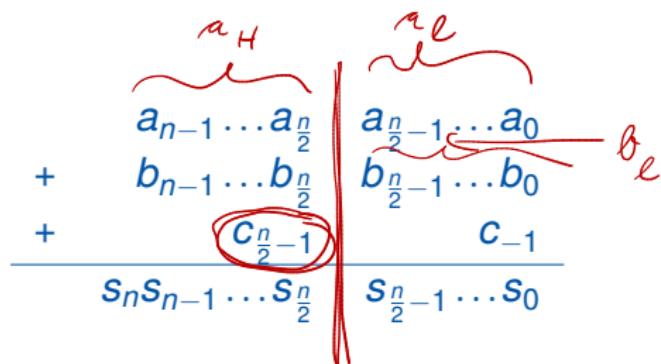
Binäre Bäume mit ~~X~~ Blättern haben mindestens Tiefe  $\lceil \log(n) \rceil$ .

$$\begin{aligned} & \text{Im Folgenden sei } n = 2^k. \\ & \log(2^{n+1}) \geq \log(2^{n+1}) \\ & > \log(2^n) = \log n + 1 \end{aligned}$$

=)

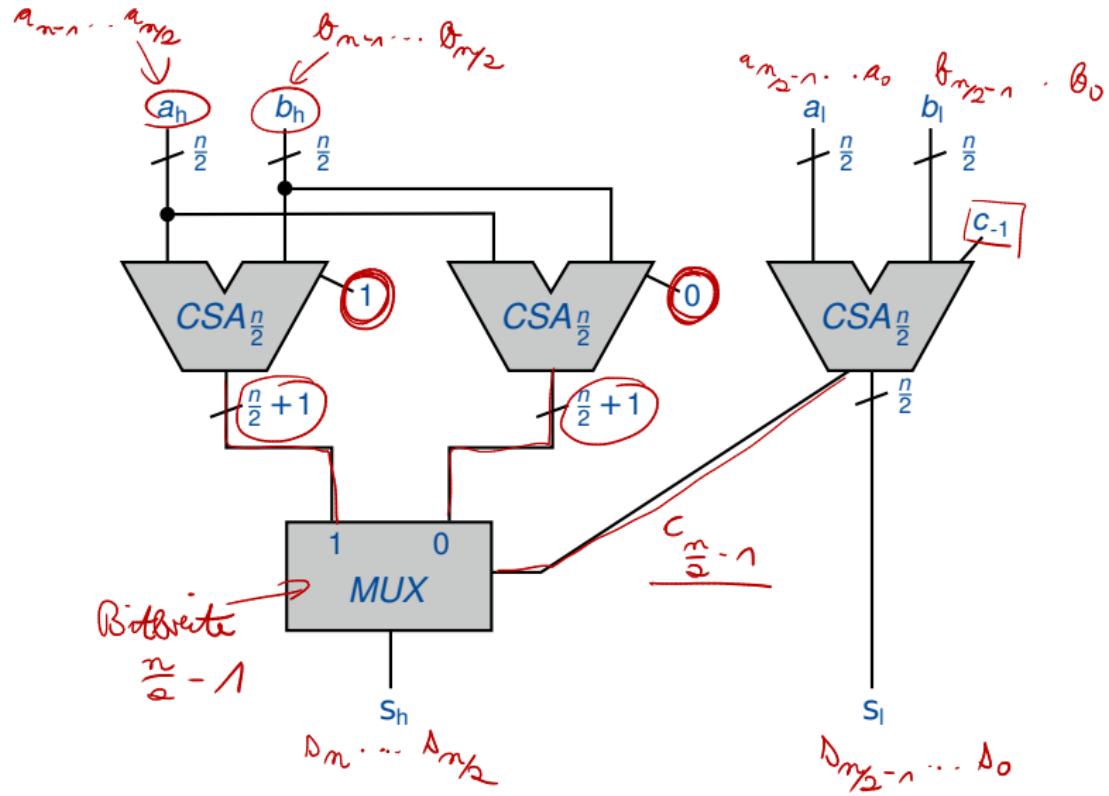
# Conditional-Sum-Addierer (CSA)

- Idee: Nutze **Parallelverarbeitung**, um Tiefe zu reduzieren!



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

# Aufbau von $CSA_n$



# Komplexität von $CSA_n$ : Tiefe

## Satz

$CSA_n$  hat Tiefe  $\leq 3 \log(n) + 3$ . für  $n = 2^k$ .

$$3k + 3$$

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

- $n=1$ :  $depth(CSA_1) = depth(FA) = 3$ .
- $n > 1$ :  $depth(CSA_n) \leq depth(CSA_{\frac{n}{2}}) + depth(MUX_{\frac{n}{2}+1})$   
 $\leq \underbrace{depth(CSA_{\frac{n}{2}})}_{2} + 3 \leq \underbrace{(3k+3)}_{3(k+1)} + 3 = \underline{3(k+1)+3}$   
 $\leq \underbrace{depth(CSA_{\frac{n}{4}})}_{4} + 3 + 3$   
 $\leq \underbrace{depth(CSA_{\frac{n}{8}})}_{8} + 3 + 3 + 3$   
...  
 $\leq depth(CSA_{\frac{n}{2^k}}) + k \cdot 3$   
 $= \underbrace{depth(CSA_1)}_{2^k} + k \cdot 3$   
 $\leq 3 \cdot (k+1) = \underline{3 \log(n) + 3}$ .

# Komplexität von $CSA_n$ : Kosten

## Satz

$$C(CSA_n) = \underline{10n^{\log(3)} - 3n - 2} \approx 10 \cdot n^{1.58} - 3n - 2$$

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

$$(2^k)^{\log_2 3} = 2^{k \cdot \log_2 3} = (2^{\log_2 3})^k = 3^k$$

■  $10(2^k)^{\log(3)} - 3 \cdot 2^k - 2 = 10 \cdot 3^k - 3 \cdot 2^k - 2$

■  $k = 0$ :  $C(CSA_1) = C(FA) = 5$ .

■ IV:  $C(CSA_{2^k}) = 10 \cdot 3^k - 3 \cdot 2^k - 2$

■ IS,  $2^k \rightarrow 2^{k+1}$ :  $C(CSA_{2^{k+1}}) = 3 \cdot \underline{C(CSA_{2^k})} + C(MUX_{\frac{2^{k+1}}{2} + 1})$

$$\stackrel{IV}{=} 3 \cdot [10 \cdot 3^k - 3 \cdot 2^k - 2] + 3 \cdot (2^k + 1) + 1$$

$$= 10 \cdot 3^{k+1} - 3 \cdot 3 \cdot 2^k - 6 + 3 \cdot 2^k + 3 + 1$$

$$= 10 \cdot 3^{k+1} - 2 \cdot 3 \cdot 2^k - 2$$

$$= 10 \cdot 3^{k+1} - 3 \cdot 2^{k+1} - 2, \text{ qed}$$

# Komplexität von $CSA_n$ : Kosten

## Satz

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

- 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 ( $CLA$ ) *← Radner-Fischer-Addierer*
  - $C(CLA_n) \leq \underline{11n}$ ,
  - $\underline{depth(CLA_n) \leq 4 \cdot \log(n) + 2}$ .

# Addition von Zweierkomplementzahlen

$$[a_n \dots a_0] = (-a_n \cdot 2^n) + \sum_{i=0}^{n-1} a_i \cdot 2^i$$

- Auszurechnen ist:

$$\underline{[a_n a_{n-1} \dots a_0]} + \underline{[b_n b_{n-1} \dots b_0]} = \underline{(-a_n 2^n)} + \underline{(-b_n 2^n)} + \sum_{i=0}^{n-1} a_i 2^i + \sum_{i=0}^{n-1} b_i 2^i$$

- Im Fall von  $(n+1)$ -Bit-Zweierkomplementzahlen können Ergebnisse im Bereich  $R_n = \{-\underline{2^n}, \dots, \underline{2^n - 1}\}$  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  $\underline{a_n}$ ,  $\underline{b_n}$  und  $\underline{s_n}$  im Binäraddierer entscheiden.

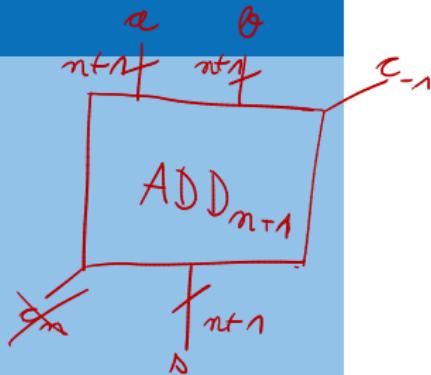
# Zweierkomplement-Addition

## Satz

Seien  $a, b \in \mathbb{B}^{n+1}$ ,  $c_{-1} \in \{0, 1\}$  und  $s \in \{0, 1\}^{n+1}$ , so dass  $\langle c_n, s \rangle = \langle a \rangle + \langle b \rangle + c_{-1}$ .

Dann gilt:

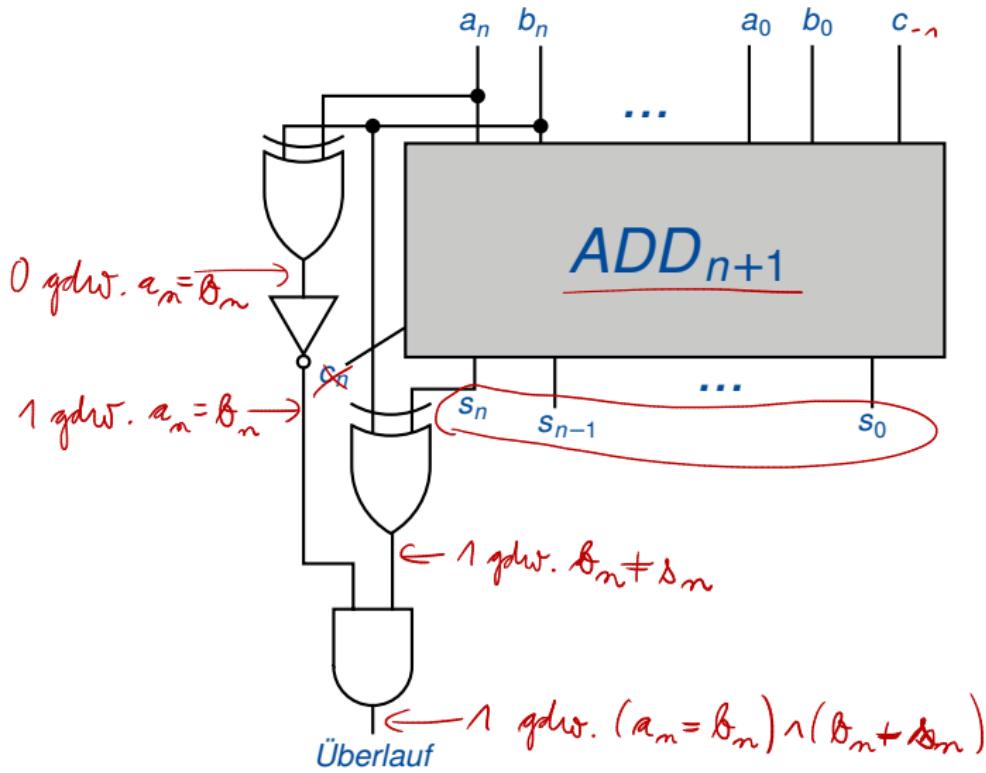
- $[a] + [b] + c_{-1} \notin R_n \Leftrightarrow (a_n = b_n) \wedge (b_n \neq s_n)$  ←
- $[a] + [b] + c_{-1} \in R_n \Rightarrow [a] + [b] + c_{-1} = [s]$



- Beweis durch Fallunterscheidung ( $[a], [b]$  beide positiv, beide negativ,  $[a]$  positiv /  $[b]$  negativ) und weitere Fallunterscheidung, ob Übertrag auf (Vorzeichen-)Stelle  $n$  kommt.
- Man kann einen alternativen Überlauftest zeigen:

$$[a] + [b] + c_{-1} \notin R_n \Leftrightarrow c_n \neq \underline{\underline{c_{n-1}}}$$

# Addierer für $(n + 1)$ -Bit-Zweierkomplement-Zahlen



# Zweierkomplement-Addition im Beispiel

$$[a] + [\bar{b}] + c_{-n} = (-a_n \cdot 2^n) + (-b_n \cdot 2^n) + \sum_{i=0}^{n-1} a_i \cdot 2^i + \sum_{i=0}^{n-1} b_i \cdot 2^i$$

**Beispiel:**  $n = 3, R_3 = \{-8, \dots, 7\}$

$$\begin{array}{r} 0001 \\ + 0011 \\ \hline (0)0100 \end{array} \quad \begin{array}{r} 1 \\ 3 \\ 4 \\ \hline 4 \end{array}$$

zur Überlauf

$$\begin{array}{r} 1001 \\ + 0011 \\ \hline (0)1100 \end{array} \quad \begin{array}{r} -7 \\ 3 \\ -4 \\ \hline -4 \end{array}$$

zur Überlauf

$$\begin{array}{r} 1001 \\ + 1011 \\ \hline (1)0100 \end{array} \quad \begin{array}{r} -7 \\ -5 \\ -12 \\ \hline -12 \end{array}$$

Überlauf

$$\begin{array}{r} 0100 \\ + 0101 \\ \hline (0)1001 \end{array} \quad \begin{array}{r} 4 \\ 5 \\ 9 \leftarrow -7 \\ \hline -7 \end{array}$$

Überlauf

$$\begin{array}{r} 1100 \\ + 0101 \\ \hline (1)0001 \end{array} \quad \begin{array}{r} -4 \\ 5 \\ 1 \\ \hline 1 \end{array}$$

zur Überlauf

$$\begin{array}{r} 1100 \\ + 1101 \\ \hline (1)1001 \end{array} \quad \begin{array}{r} -4 \\ -3 \\ -7 \\ \hline -7 \end{array}$$

zur Überlauf

# Subtraktion

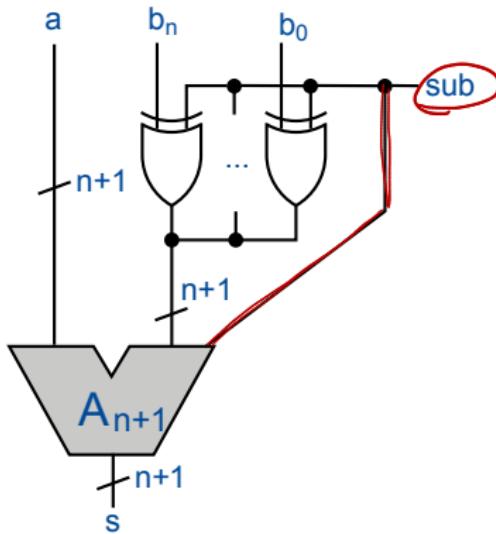
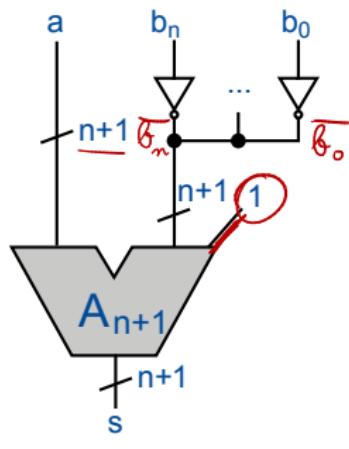
- Wegen  $-[b] = \underline{\bar{b}} + 1$  kann  $\underline{[a] - [b]}$  zurückgeführt werden auf  $\underline{[a]} + \underline{\bar{b}} + \underline{1}$ .
- Beispiel:  $\underline{+} \underline{\underline{[b]}} + 1$

$$[a] = [01110] = 6_{10}, \quad [b] = [01111] = 7_{10}, \quad [\bar{b}] = \underline{[1000]}$$

$$\begin{array}{r} 01110 \\ + 10000 \\ + 00001 \\ \hline 11111 \end{array} \qquad 1111 = \underline{(-1)_{10}}$$

- Den Schaltkreis für Subtraktion gewinnt man aus einem Addierer.
- Kombinierter Addierer/Subtrahierer.

# Subtrahierer



$$\begin{array}{l} \underline{b_j \oplus 0 = b_j} \\ \underline{b_j \oplus 1 = \underline{b_i}} \end{array}$$

$$\begin{aligned} sub &= 0 : [a] + [b] + 0 \\ sub &= 1 : [a] + [\bar{b}] + 1 = [a] - [b] \end{aligned}$$



# Kapitel 3 – Kombinatorische Logik

1. Kombinatorische Schaltkreise
2. Boolesche Algebren
3. Boolesche Ausdrücke, Normalformen, zweistufige Synthese
4. Berechnung eines Minimalpolynoms
5. Arithmetische Schaltungen
- 6. 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.
- Erinnerung: Der Befehlssatz von ReTI hat die folgenden Compute-Befehle (s. nächste Folie).

# Compute-Befehle: Kodierung

*immediate*

*memory*

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

# Addition und Sign Extension

## ■ Probleme bei **Additionen**:

- 1 Addition verschieden langer Zahlen (z. B.  $[ACC] + [i]$ ).
- 2 Addition von Binärdarstellungen und Zweierkomplementzahlen (braucht man bei Adressrechnungen für LOADINj, STOREINj, z. B.  $M(\langle IN1 \rangle + [i]) := ACC$ ).

32 Bit  
↓  
24 Bit

## ■ Zu 1: Lösung durch Sign Extension

- Sei  $y \in \mathbb{B}^{24}$ .  $\text{sext}(y) := y_{23}^8 y$  heißt **sign extension** von  $y$ .

- Es gilt:  $[y]_2 = [\text{sext}(y)]_2$ . (Beweis: Übung)

- Dann wird  $[ACC] + [i]$  zurückgeführt auf  $[ACC] + [\text{sext}(i)]$ .

## ■ Zu 2: Siehe nächste Folie.

$$\begin{aligned}[y_{23} \dots y_0] &= \sum_{i=0}^{22} y_i \cdot 2^i - y_{23} \cdot 2^{23} \\[y_{23} y_{23} \dots y_0] &= \sum_{i=0}^{23} y_i \cdot 2^i - y_{23} \cdot 2^{23} \\&= \sum_{i=0}^{22} y_i \cdot 2^i + y_{23} \cdot 2^{23} - y_{23} \cdot 2^{23} \\&= \sum_{i=0}^{22} y_i \cdot 2^i - y_{23} \cdot 2^{23}\end{aligned}$$

# Addition von Binär- und Zweierkomplementzahlen

## Lemma

*Ergebnis = gültige Adresse wird vorausgesetzt!*

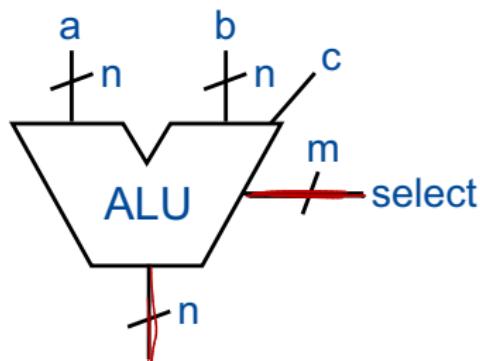
Sei  $x \in \mathbb{B}^{32}$ ,  $y \in \mathbb{B}^{24}$ ,  $0 \leq \langle x \rangle + [y] < 2^{32}$  und sei  
 $\langle x \rangle + \langle \text{sext}(y) \rangle = \langle c, s \rangle$  mit  $c \in \mathbb{B}$ ,  $s \in \mathbb{B}^{32}$ .  
Dann gilt:  $\langle x \rangle + [y] = \langle s \rangle$

**Beweis:** Erweitere auf 33-stellige Zweierkomplementzahlen  
und benutze einfach den Satz aus dem letzten Abschnitt.

- Der Satz aus dem letzten Abschnitt besagt:  
Falls  $\langle 0, x \rangle + \langle y_{23}, \text{sext}(y) \rangle = \langle c'', c', s \rangle$  und  
 $[0, x] + [y_{23}, \text{sext}(y)] \in \{-2^{32}, \dots, 2^{32} - 1\}$ , dann  
 $[0, x] + [y_{23}, \text{sext}(y)] = [c', s]$ .
- $[0, x] + [y_{23}, \text{sext}(y)] = \langle x \rangle + [y] \in \{-2^{32}, \dots, 2^{32} - 1\}$  ist klar  
wegen  $0 \leq \langle x \rangle + [y] < 2^{32}$ .
- Aus  $[0, x] + [y_{23}, \text{sext}(y)] = \langle x \rangle + [y] = [c', s]$  und  $0 \leq \langle x \rangle + [y]$   
folgt  $c' = 0$  und damit  $\langle x \rangle + [y] = [0, s] = \langle s \rangle$ .

# 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  $n$ -Bit-Ausgang.
    - $n = 32$ .
- Insgesamt 68 Ein- und 32 Ausgänge.



# Select-Eingang bei ReTI-ALU

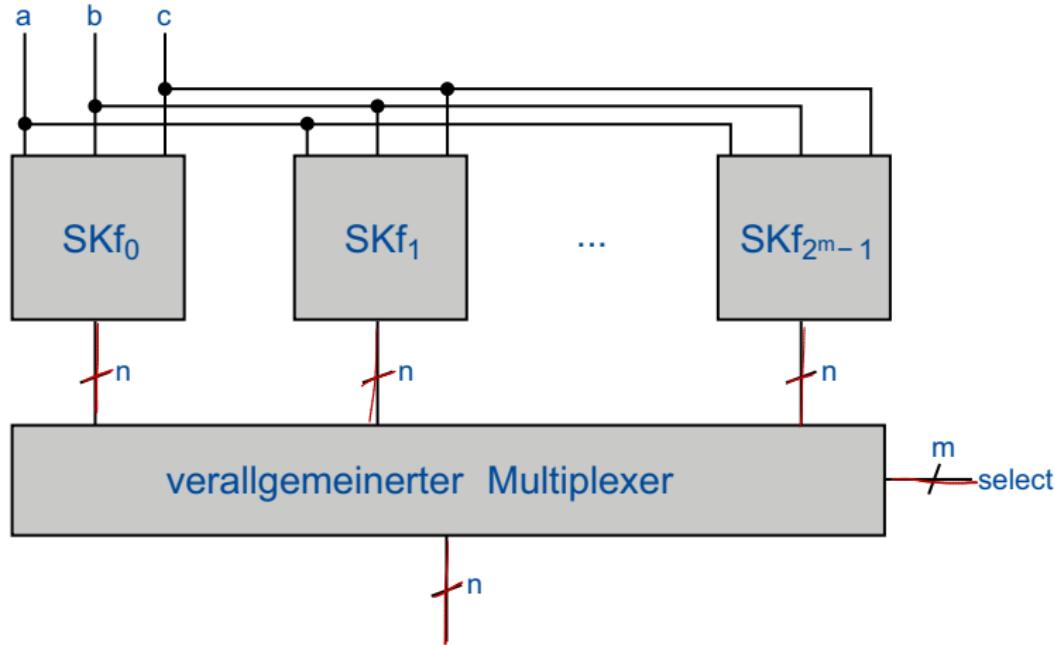
Funktionsnummer	ALU-Funktion
$s_2 \ s_1 \ s_0$	
0 0 0	<u>0...0</u>
0 0 1	<u><math>[b] - [a]</math></u>
0 1 0	<u><math>[a] - [b]</math></u>
0 1 1	<u><math>[a] + [b] + c</math></u>
1 0 0	<u><math>a \oplus b = (a_{n-1} \oplus b_{n-1}, \dots, a_0 \oplus b_0)</math></u>
1 0 1	<u><math>a \vee b = (a_{n-1} \vee b_{n-1}, \dots, a_0 \vee b_0)</math></u>
1 1 0	<u><math>a \wedge b = (a_{n-1} \wedge b_{n-1}, \dots, a_0 \wedge b_0)</math></u>
1 1 1	<u>1...1</u>

# Mögliche Realisierungen der ALU (1/2)

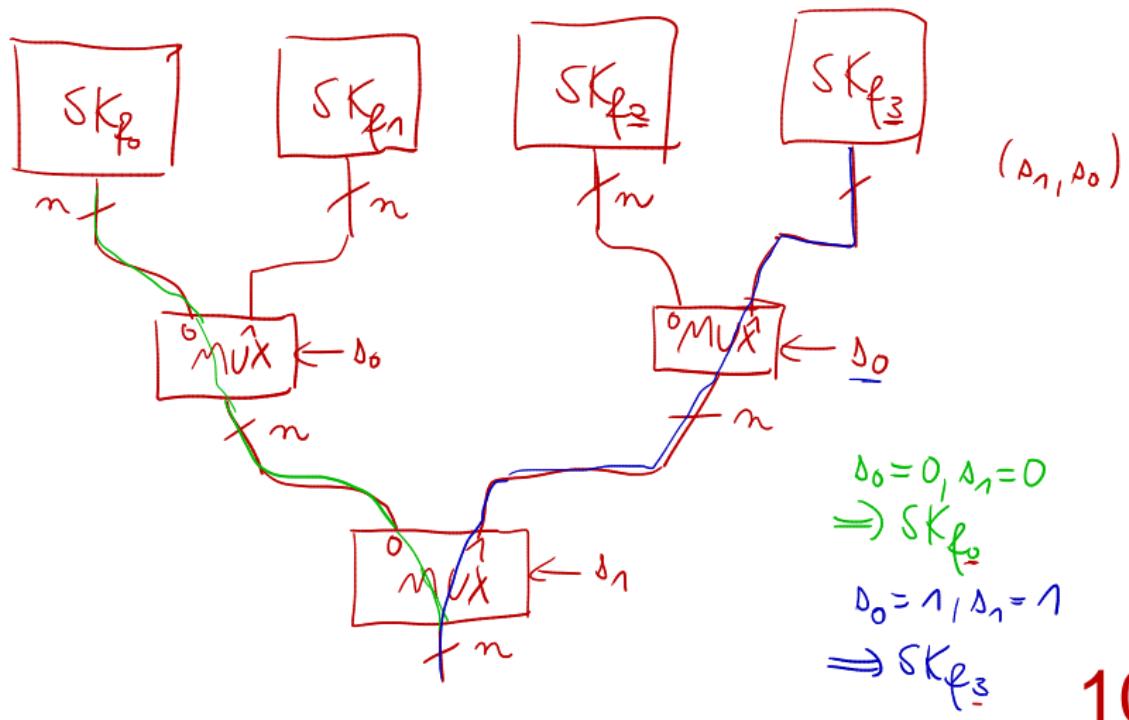
---

- **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



$m=2 \Rightarrow 2$  select-Bits ( $s_1, s_0$ )

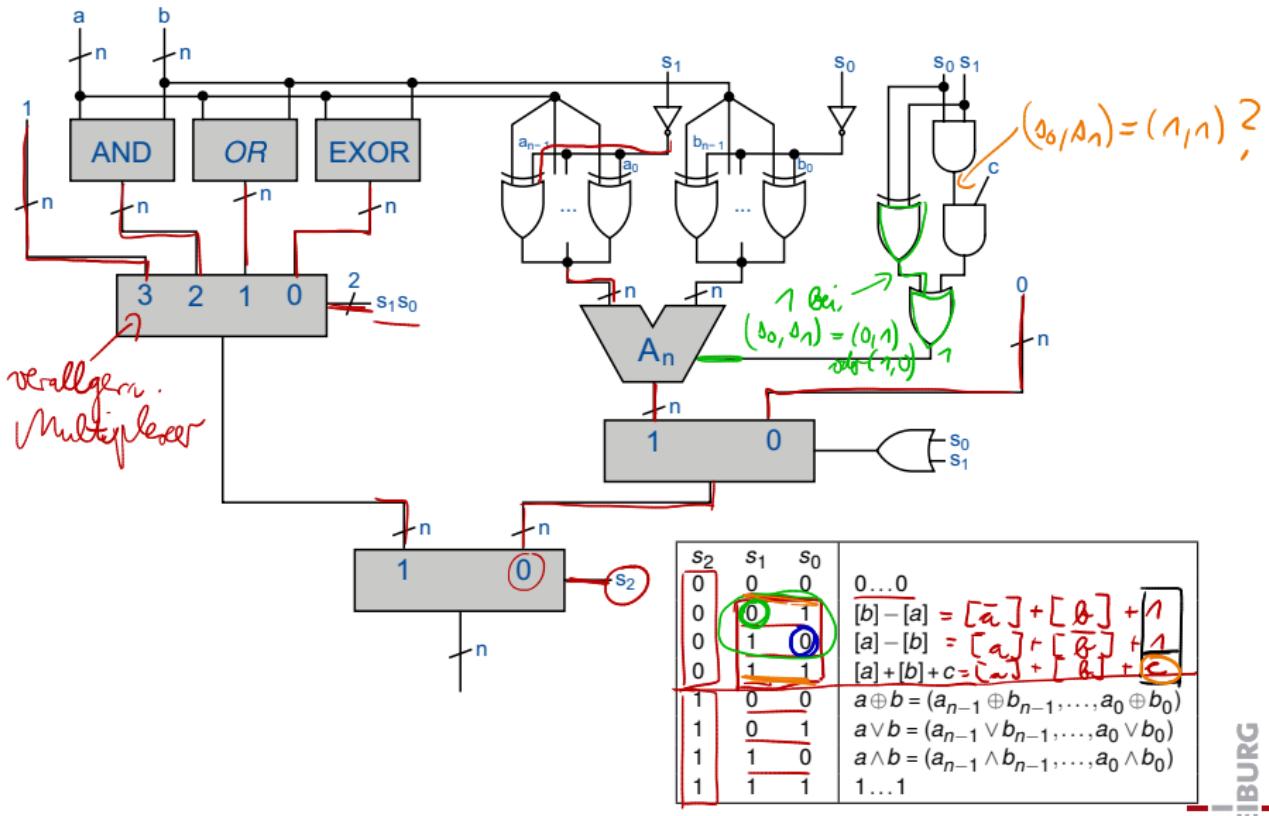


## Mögliche Realisierungen der ALU (2/2)

---

- **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.
  - Komplizierter zu realisieren, aber effizienter.

# Schaltungsrealisierung der ALU



# 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

1.1 Schaltpläne, RS-Flip Flop

1.2 D-Flipflops, weitere Bausteine

## 2. Sequentielle Schaltkreise

## 3. Entwurf sequentieller Schaltkreise

## 4. SRAM

## 5. Anwendung: Datenpfade von ReTI

Albert-Ludwigs-Universität Freiburg

Prof. Dr. Christoph Scholl

Institut für Informatik

Sommersemester 2023

# Schaltpläne (1/3)

---

- Analyse von Schaltplänen  $SP = (\vec{X}_n, \underline{G}, typ, IN, \vec{Y}_m)$  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  $SP = (\vec{X}_n, \underline{G}, \underline{typ}, \underline{IN}, \vec{Y}_m)$  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.  
~~ausgetauscht~~
  - $G = (V, E)$  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  $IN : I \rightarrow E^*$  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  $\vec{Y}_m = (y_1, \dots, y_m)$  zeichnet Knoten  $y_i \in V$  als Ausgänge aus.

# Belegungen von Schaltplänen (1/2)

Sei nun ein Schaltplan  $\underline{SP} = (\vec{X}_n, G, typ, IN, \vec{Y}_m)$  gegeben.

- Eine Abbildung  $\Phi_{SP,a}: V \rightarrow \{0, 1\}$  für  $a = \underline{(a_1, \dots, a_n)} \in \mathbb{B}^n$  heißt **Belegung** für Eingangsbelegung  $a$ , falls
  - $\Phi_{SP,a}(x_i) = \underline{a_i} \quad \forall 1 \leq i \leq n,$
  - $\Phi_{SP,a}(0) = \underline{0}, \Phi_{SP,a}(1) = \underline{1},$
  - für alle  $v \in I$  mit  $typ(v) = g \in B_k, IN(v) = (e_1, \dots, e_k)$  gilt  
 $\underline{\Phi_{SP,a}(v) = g(\Phi_{SP,a}(Q(e_1)), \dots, \Phi_{SP,a}(Q(e_k)))}.$



## Belegung von Schaltplänen (2/2)

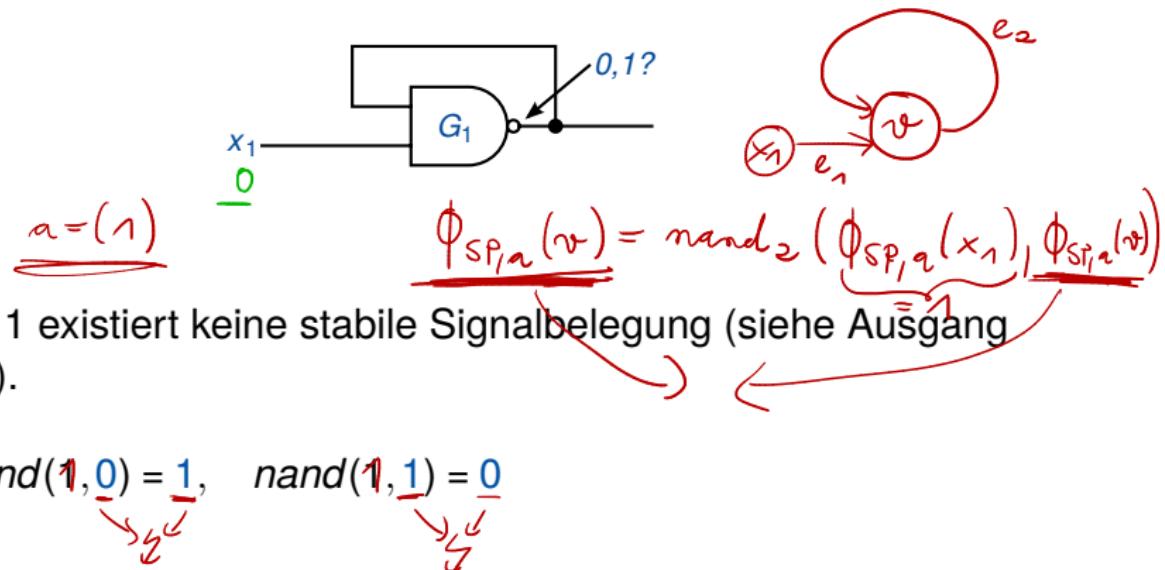
---

- Da ein Schaltplan nicht azyklisch ist, muss nicht zu jeder Eingangsbelegung eine Belegung definiert sein (im Gegensatz zu Schaltkreisen).
- Falls bei einem Schaltplan zu einer Eingangsbelegung eine Belegung definiert ist, dann nennen wir diese zur Verdeutlichung auch stabile Belegung.

Genauer: Es ist möglich, dass es zu einer Eingangsbelegung a

- keine stabile Signalbelegung  $\Phi_{SP,a}$  gibt,
- mehrere stabile Signalbelegungen  $\Phi_{SP,a}$  gibt.

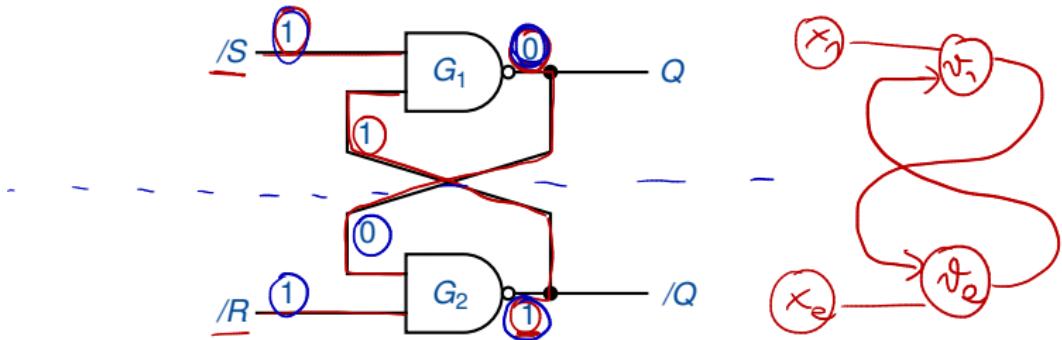
# Beispiel 1



Für  $a = 0$  existiert eine stabile Signalbelegung:

$$\phi_{SP,a}(v) = 1 \quad | \quad \text{nand}_2(0, 1) = 1$$

## Beispiel 2 (1/2)



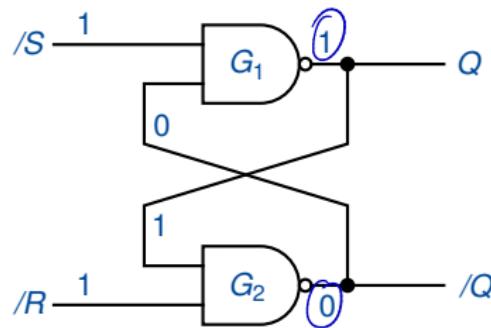
Betrachte Eingangsbelegung  $/S = 1$ ,  $/R = 1$

- $G_1 : \text{nand}(1, 1) = \underline{0}$

- $G_2 : \text{nand}(0, 1) = \underline{1}$

$\Rightarrow \Phi_{SP,(1,1)}(G_1) = \underline{0}$  und  $\Phi_{SP,(1,1)}(G_2) = \underline{1}$  stellt eine stabile Signalbelegung dar!

## Beispiel 2 (2/2)



Betrachte Eingangsbelegung  $/S = 1$ ,  $/R = 1$

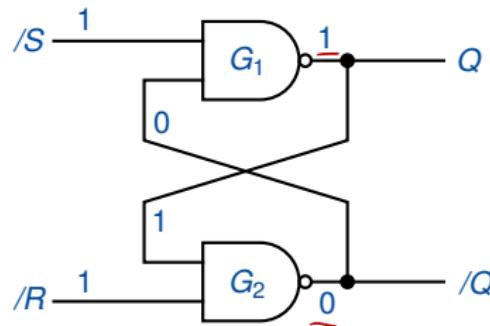
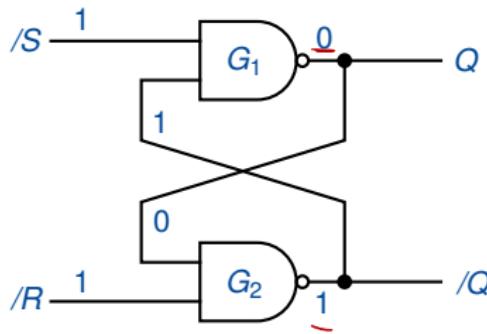
- $G_1 : \text{nand}(1, 0) = 1$

- $G_2 : \text{nand}(1, 1) = 0$

$\Rightarrow \Phi'_{SP,(1,1)}(G_1) = 1$  und  $\Phi'_{SP,(1,1)}(G_2) = 0$  stellt ebenfalls eine stabile Signalbelegung dar!

# RS-Flipflop

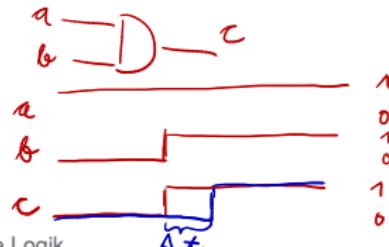
- Die vorherige Schaltung heißt **RS-Flipflop** (kurz **RS-FF**).
- Sie hat für die Eingangsbelegung  $/S = 1, /R = 1$  zwei stabile Zustände.



- Frage:** Wie kann man von einem Zustand zum anderen kommen?

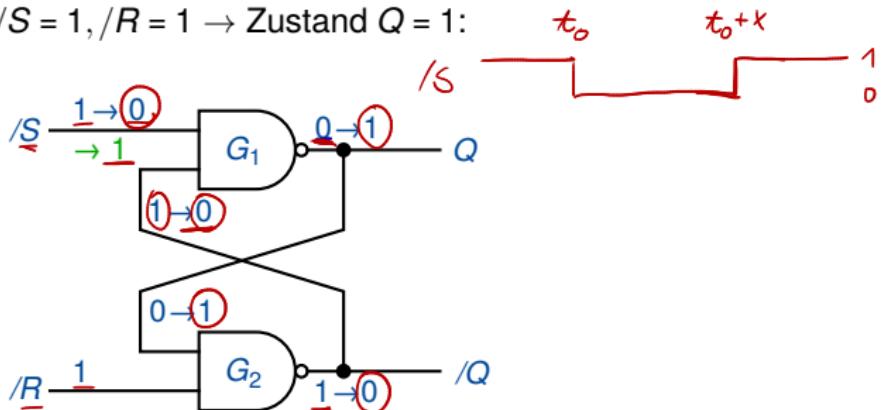
# Übergang (1/2)

- 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  $Q = 0$  mit  $/S = 1, /R = 1 \rightarrow$  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  $\underline{Q = 1}$ .
- Nach Zeit  $t_{P/S/Q}$  ist  $\underline{/Q = 0}$ .
- Wechsel von Zustand  $Q = 1$  zu Zustand  $Q = 0$  aus Symmetriegründen analog.

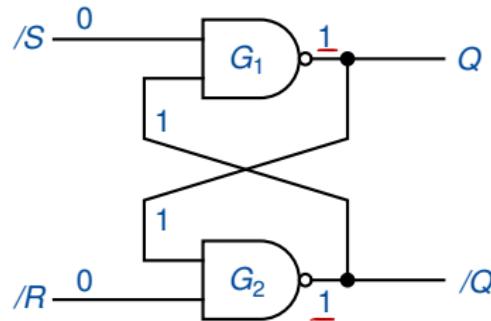
# Weitere Bezeichnungen

---

- 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” $Q = 1, /Q = 1$

---

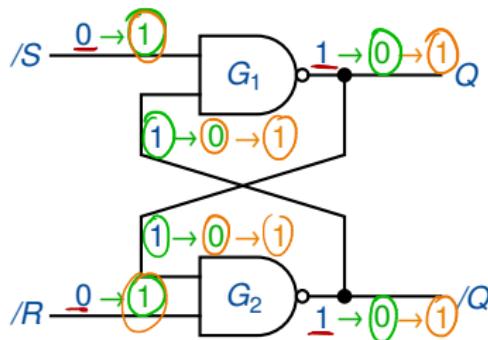


- Stabile Signalbelegung bei Eingangsbelegung  $\underline{/S = 0}, \underline{/R = 0}$
- Aber warum ist es trotzdem problematisch,  $/S$  und  $/R$  gleichzeitig zu aktivieren ( $/S = 0, /R = 0$ )?

# Flackern

Annahme:

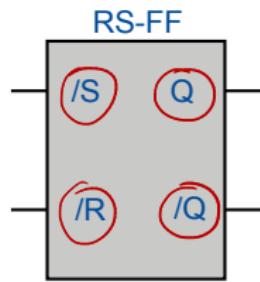
- $/S$  und  $/R$  werden nach ihrer Aktivierung beide gleichzeitig inaktiv ( $00 \rightarrow 11$ )
- $G_1$  und  $G_2$  schalten exakt gleich schnell, d.h. haben exakt die gleiche Verzögerungszeit



- $\Rightarrow$  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

---



# Kapitel 4 – Sequentielle Logik

## 1. Speichernde Elemente

1.1 Schaltpläne, RS-Flip Flop

1.2 D-Flipflops, weitere Bausteine

2. Sequentielle Schaltkreise

3. Entwurf sequentieller Schaltkreise

4. SRAM

5. Anwendung: Datenpfade von ReTI

Albert-Ludwigs-Universität Freiburg

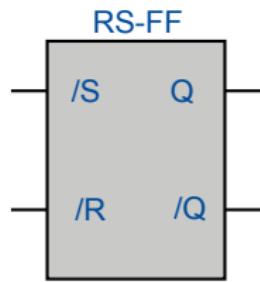
Prof. Dr. Christoph Scholl

Institut für Informatik

Sommersemester 2023

# Schaltsymbol eines RS-FF

---



# Nachteil von RS-FF

---

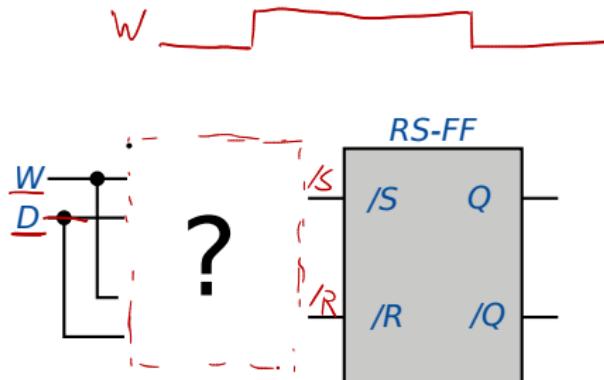
Beim Speichern eines Wertes **0** oder **1** muss man den Wert kennen:

- **0** → Aktiviere  $/R$
- **1** → Aktiviere  $/S$

## Ziel:

- Speichern unbekannter Werte.

# D-Latch (1/2)

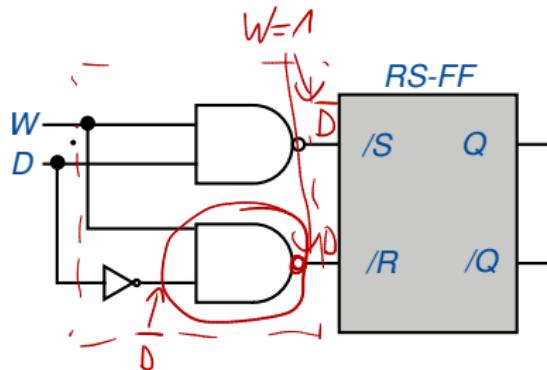


$W$	$D$	$/S$	$/R$
0	0	1	1
0	1	1	1
1	0	1	0
1	1	0	1

■  $W$  ist active high.

- $\underline{W = 0} \Rightarrow \underline{/S}, \underline{/R}$  inaktiv
- $\underline{W = 1} \Rightarrow \begin{cases} \underline{/S} \text{ aktiv, falls } \underline{D = 1} \\ \underline{/R} \text{ aktiv, falls } \underline{D = 0} \end{cases}$

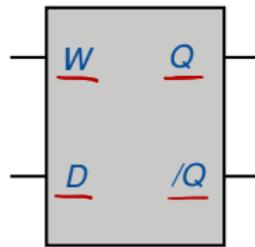
## D-Latch (2/2)



$W$	$D$	$/S$	$/R$
0	0	1	1
0	1	1	1
1	0	1	0
1	1	0	1

←

Symbol:

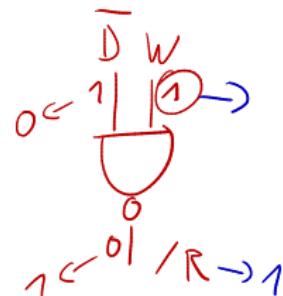
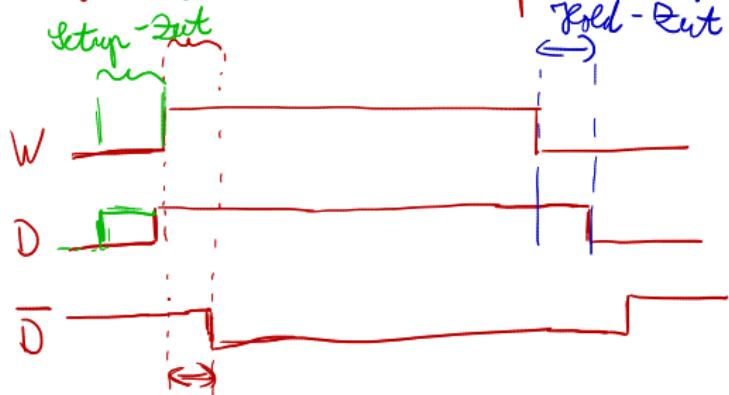


$D=1$

$W$

$/S$

Möglicher Problem beim Abspeisen von 1:



$/R$

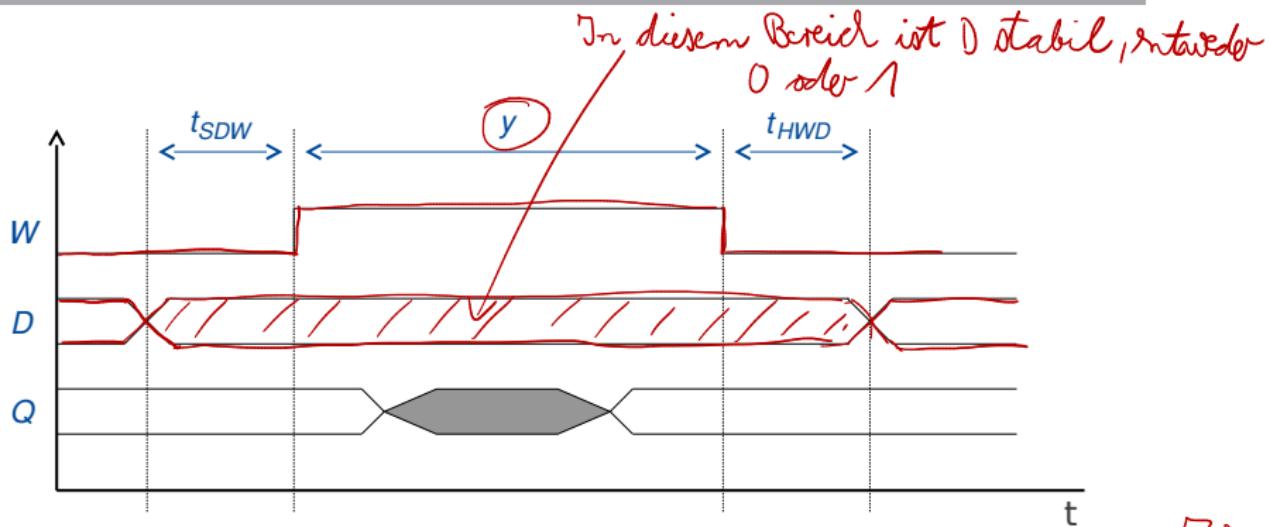
Puls auf  $/R$  überwölbt!  
W erst anheben, wenn  $/R = 1$ , wegen  $\bar{D} = 0$   $\Rightarrow$  „Setup-Zeit“  
 $D$  erst abstecken, wenn  $/R = 1$ , wegen  $W = 0$   $\Rightarrow$  „Hold-Zeit“

# Ansteuerung: Schreibimpuls

---

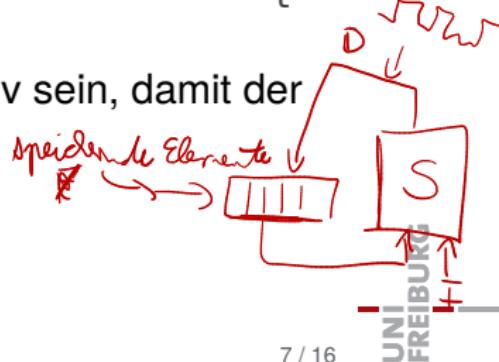
- Die Daten müssen für eine gewisse Zeit  $t_{SDW}$ , genannt **Setup-Zeit**, an  $D$  stabil anliegen.
- Dann geht  $W$  von 0 auf 1, bleibt für eine Zeit  $y$ , genannt **Pulsweite**, auf 1 und geht auf 0 zurück.
- Anschließend müssen die Daten für eine Zeit  $t_{HDW}^{WD}$ , genannt **Hold-Zeit**, an  $D$  stabil gehalten werden.

# Timing-Diagramm



Wie lange müssen die einzelnen Signale aktiv sein, damit der Schreibvorgang reibungslos abläuft?

⇒ Siehe nächstes Kapitel ([Timing](#)).

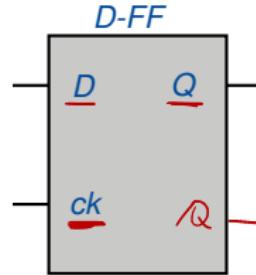


# Taktflankengesteuertes D-Flipflop (1/2)



- **Taktflankengesteuerte Flipflops** wie das D-Flipflop übernehmen Daten zu einem bestimmten Zeitpunkt, 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).

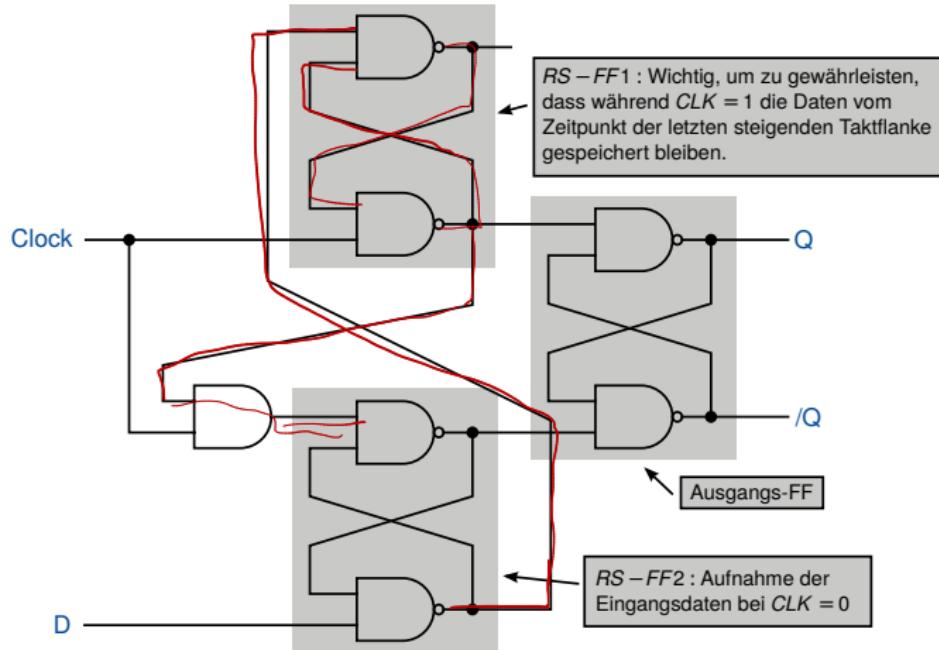


## Taktflankengesteuertes D-Flipflop (2/2)

---

- 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

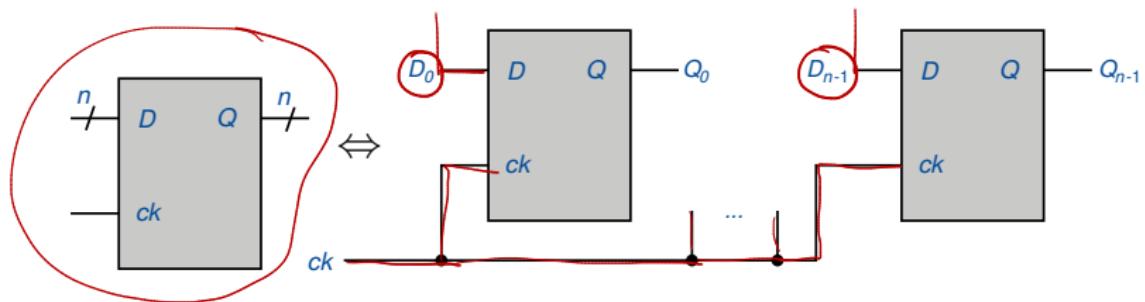
---

- Register
- Schieberegister
- Zähler

# $n$ -Bit Register

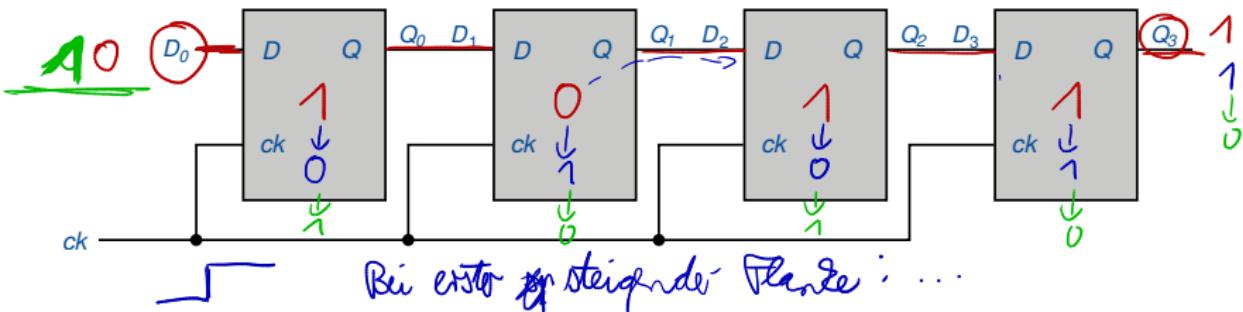
Bsp. RETI : ACC, W1, W2 sind 32-Bit-Register

- $n$  D-Flipflops mit gemeinsamen Clocksignal.



- Entsprechend:  $n$ -Bit Latch =  $n$  D-Latches mit gemeinsamem Schreibsignal  $W$ .

# Schieberegister

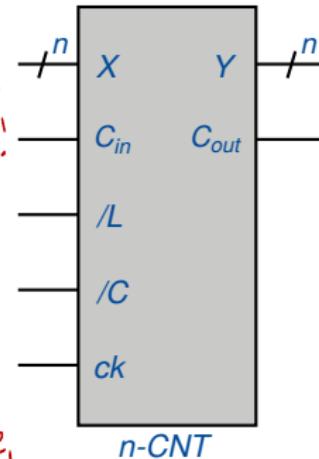


- In jedem Takt (bei jeder steigenden Flanke von  $ck$ ) werden die Werte im Register um eine Position nach rechts verschoben.

Zähler  $\rightarrow$  z. B. Programmzähler bei der ReTI!  
 $\Rightarrow$  32-Bit-Zähler!

Ein **n-Bit-Zähler** ist eine Schaltung mit folgenden Ein- und Ausgängen:

- Dateneingänge  $X = \underline{(X_{n-1}, \dots, X_0)}$
- Datenausgänge  $Y = \underline{(Y_{n-1}, \dots, Y_0)}$  *Zählerstand!*
- Dateneingang  $C_{in}$  für Eingangsübertrag
- Datenausgang  $C_{out}$  für Ausgangsübertrag
- Eingänge für Kontrollsignale:
  - /C (Clear)
  - /L (Load)
  - ck (Clock)

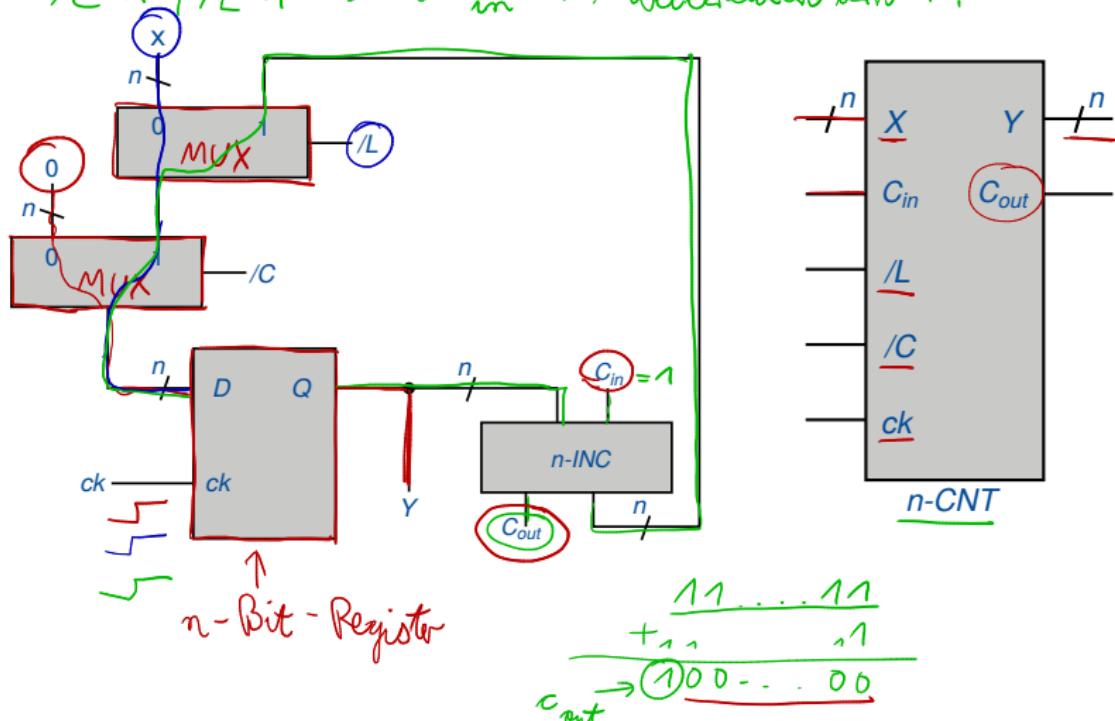


# Aufbau eines Zählers

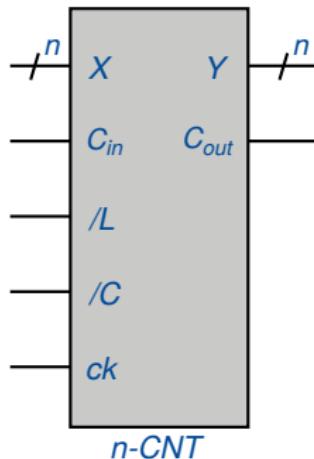
$/C = 0$

$/C = 1, /L = 0 \Rightarrow$  Laden der Eingangsdaten  $X$  in Zähler

$/C = 1, /L = 1 \Rightarrow$  bei  $c_{in} = 1$ : Weiterzählen um 1!



# $n$ -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 :

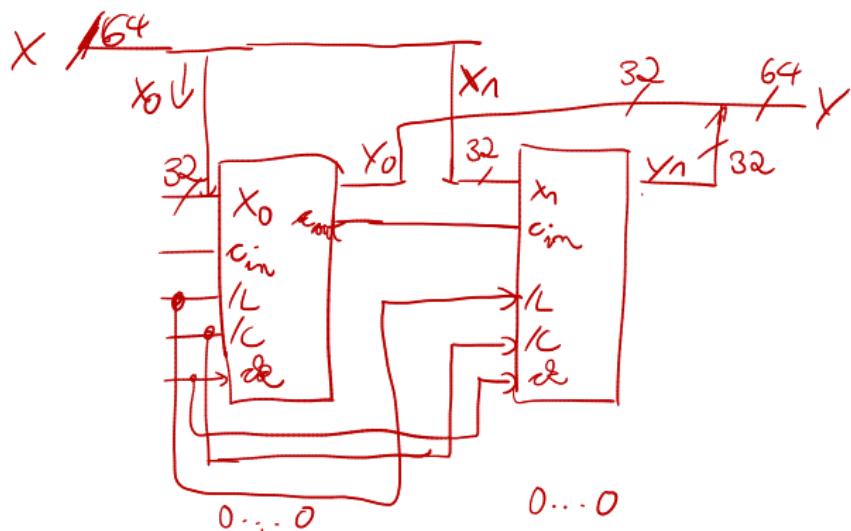
$$Y_{neu} = \begin{cases} \underline{\underline{0 \dots 0}}, & \text{falls } \underline{\underline{/C = 0}} \\ \underline{\underline{X}}, & \text{falls } \underline{\underline{/C = 1, /L = 0}} \\ \underline{\underline{bin_n((Y) + C_{in}) mod 2^n}}, & \text{falls } \underline{\underline{/C = 1, /L = 1}} \end{cases}$$

Wie  $\mod 2^n$ ?

$$\langle 1 \dots 1 \rangle = 2^n - 1 \quad \langle 1 \dots 1 \rangle + 1 = 2^n$$

$$2^n \mod 2^n = 0$$

$$A \mod 2^n = A, \text{ falls } 0 \leq A \leq 2^n - 1$$



Kaskadiert 2 32-Bit-Zähler zu einem  
64-Bit-Zähler

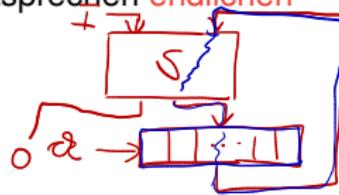


# Kapitel 4 – Sequentielle Logik

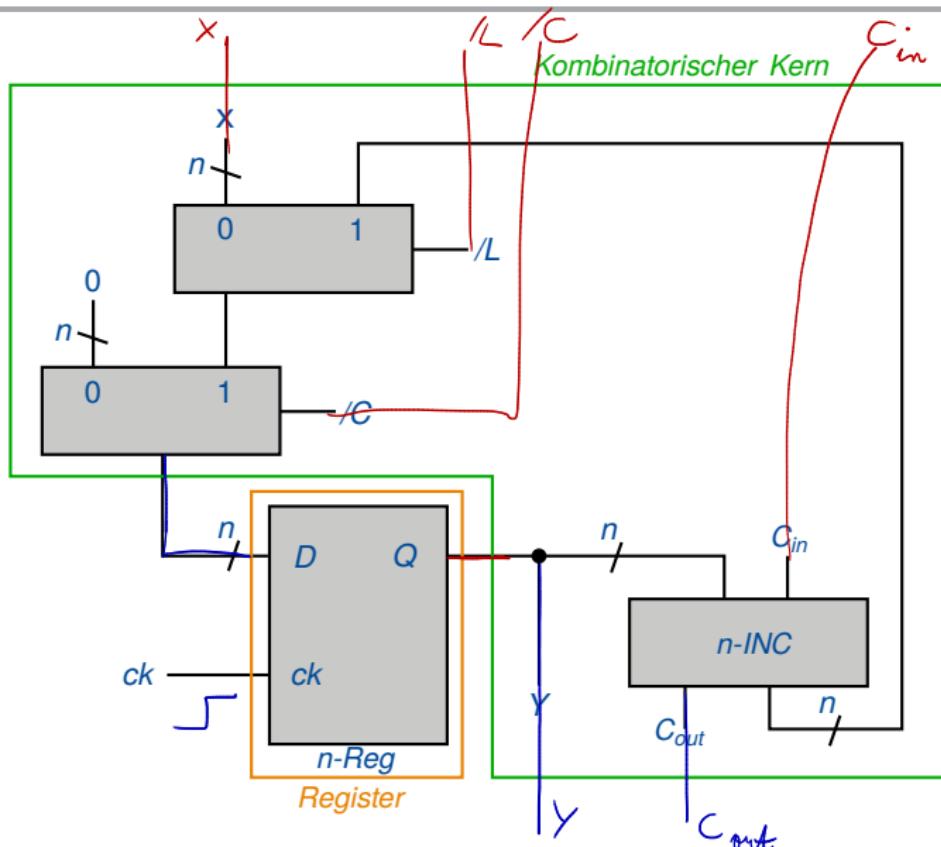
1. Speichernde Elemente
- 2. Sequentielle Schaltkreise**
3. Entwurf sequentieller Schaltkreise
4. SRAM
5. Anwendung: Datenpfade von ReTI

# 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 (unter anderem) 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).

# Halbautomat

## Definition

Das Quadrupel  $H = (\underline{I}, \underline{S}, \underline{S_0}, \underline{\delta})$  heißt **deterministischer, endlicher Halbautomat**. Dabei bezeichnet:

- $I$  eine endliche Menge von erlaubten **Eingabesymbolen** („Eingabealphabet“), (*bei uns üblicherweise  $I \subseteq \{0,1\}^*$* )
- $S$  eine endliche Menge von **Zuständen**, (*bei uns  $S \subseteq \{0,1\}^n$* )
- $S_0 \subseteq S$  ist eine endliche Menge von erlaubten **Anfangszuständen**,
- $\delta : \underline{S} \times \underline{I} \rightarrow \underline{S}$  eine **Übergangsfunktion**.

Bsp.:

$$\begin{aligned}\delta(s_1, i_1) &= s_2 \\ \delta(s_1, i_2) &= s_3\end{aligned}$$



# Mealy- und Moore-Automat

## Definition

Ein Mealy-Automat  $M = (I, O, S, S_0, \delta, \lambda)$  ist ein endlicher deterministischer Halbautomat  $H$  erweitert um:

- eine endliche Menge  $O$  von Ausgabesymbolen („Ausgabealphabet“),  
Bsp.:  $\lambda(s_1, i_1) = o_1$   
 $\lambda(s_1, i_2) = o_2$
- eine Ausgabefunktion  $\lambda : S \times I \rightarrow O$ .



## Definition

Ein Moore-Automat  $M = (I, O, S, S_0, \delta, \lambda)$  ist ein endlicher, deterministischer Halbautomat  $H$  erweitert um:

- eine endliche Menge  $O$  von Ausgabesymbolen,  
Bsp.:  $\lambda(s_1) = o_1$   
 $\lambda(s_2) = o_2$
- eine Ausgabefunktion  $\lambda : S \rightarrow O$ .



# Mealy- vs. Moore-Automat

---

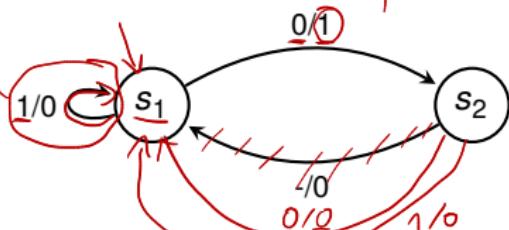
- Beim Mealy-Automaten ist:
  - die **Ausgabe** abhängig vom aktuellen Zustand **und** der aktuellen Eingabe,
  - der **Folgezustand** abhängig vom aktuellen Zustand und der aktuellen Eingabe.
- Ein Moore-Automat ist ein spezieller Mealy-Automat, bei dem die Ausgabe nur vom **aktuellen Zustand** und nicht von der Eingabe abhängt.
- Moore- und Mealy-Automaten kann man **ineinander überführen**.

# Unterschiedliche Darstellungen von endlichen Zustandsautomaten

a) Zustands- und Ausgangstafel:

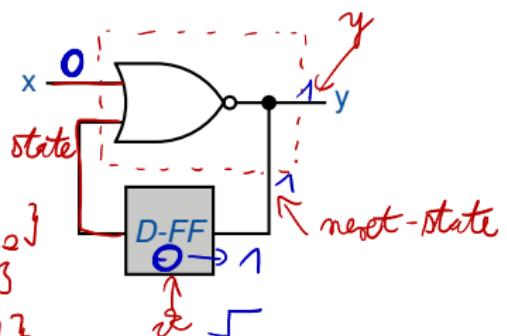
state	x	next-state	y
$s_1 \rightarrow 0$	1	$s_1 \rightarrow 0$	0
$s_1 \rightarrow 0$	0	$s_2 \rightarrow 1$	1
$s_2$		$s_1$	0
$s_2 \rightarrow 1$	0	$s_1 \rightarrow 0$	0
$s_2 \rightarrow 1$	1	$s_1 \rightarrow 0$	0

b) Zustandsdiagramm:



■ Im Folgenden: Weg von b) zu c)

c) Sequentieller Schaltkreis:



$$S = \{s_1, s_2\}$$

$$S_0 = \{s_1\}$$

$$I = \{0, 1\}$$

$$O = \{0, 1\}$$

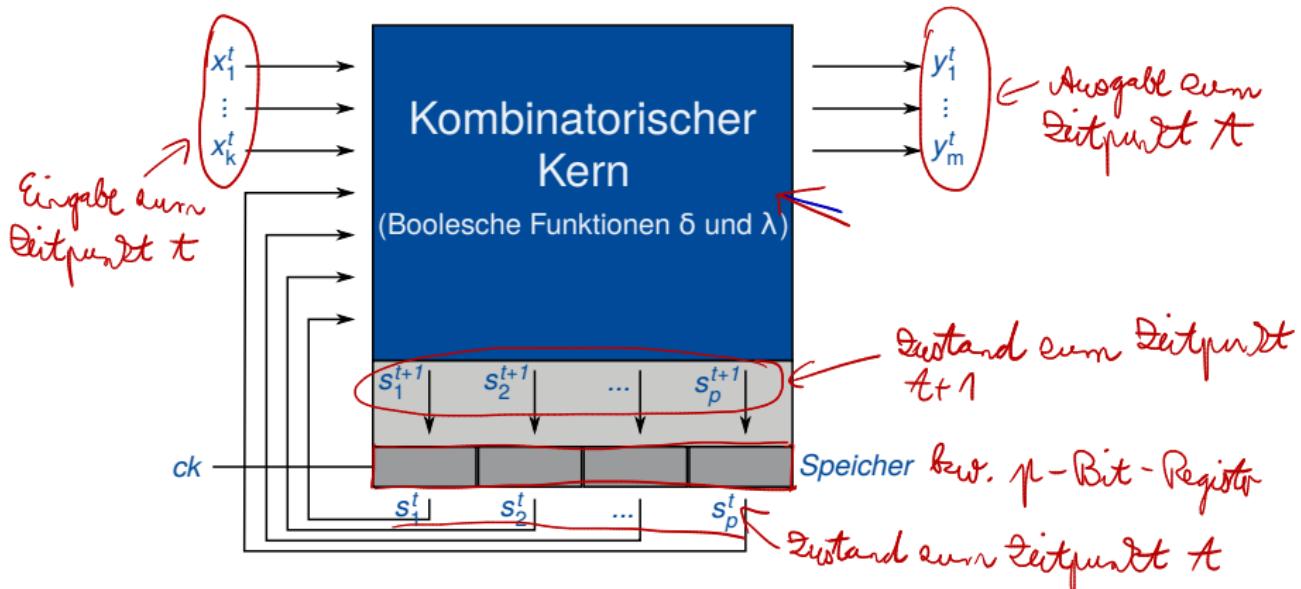
$$f(s_1, 0) = s_2 \quad \lambda(s_1, 0) = 1$$

$$f(s_1, 1) = s_1 \quad :$$

$$f(s_2, 0) = s_1 \quad :$$

$$f(s_2, 1) = s_2 \quad :$$

# Sequentielle Schaltkreise allgemein



$$\begin{aligned}y_i^t &= \underline{\lambda_i(x_1^t, x_2^t, \dots, x_k^t, s_1^t, s_2^t, \dots, s_p^t)} \\s_i^{t+1} &= \underline{\delta_i(x_1^t, x_2^t, \dots, x_k^t, s_1^t, s_2^t, \dots, s_p^t)}\end{aligned}$$

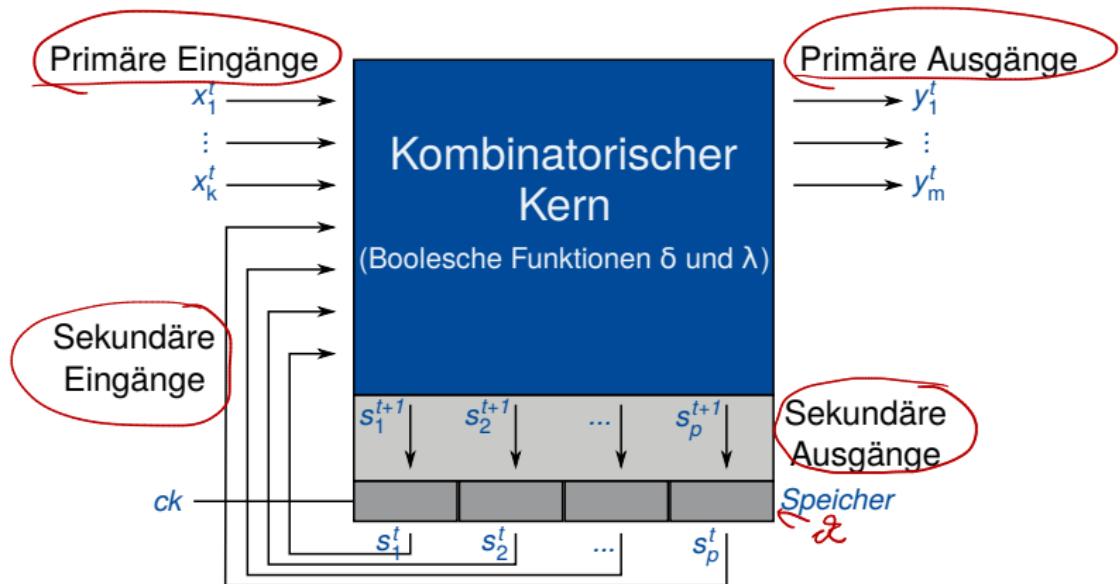
Die Belegung  $s^t$  der Flipflops im Register heißt **Zustand** des sequentiellen Schaltkreises zum **Zeitpunkt t**.

# Kombinatorischer Kern

---

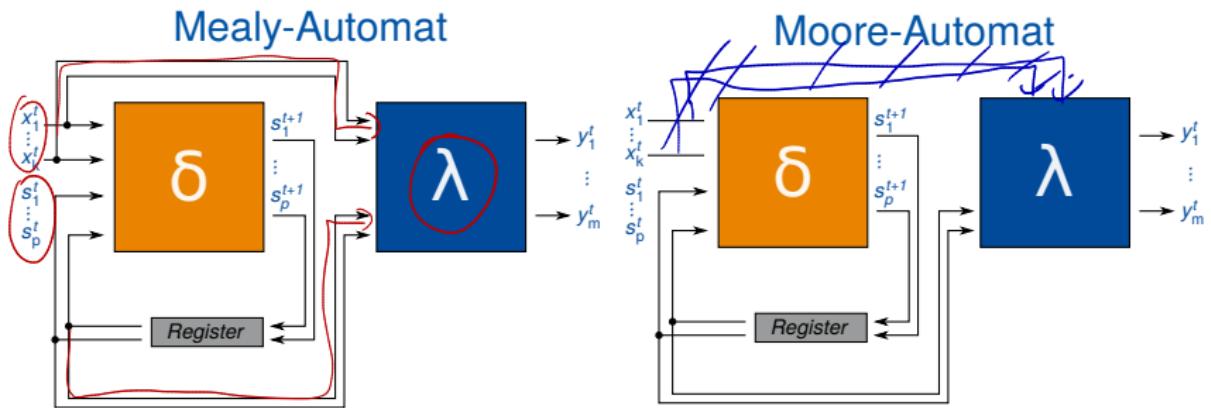
- Der kombinatorische Kern hat vier Arten von Ein- und Ausgängen:
  - Primäre Eingänge bekommen Werte „von außen“.
  - Primäre Ausgänge liefern Werte „nach außen“.
  - Sekundäre Eingänge sind mit den Datenausgängen der Flipflops im Register verbunden. Auf diese Weise kann der aktuelle Zustand des Schaltkreises 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



- Übergangsfunktion:  $S^{t+1} = \delta(X^t, S^t)$
- Ausgabefunktion (Mealy):  $Y^t = \lambda(X^t, S^t)$
- Ausgabefunktion (Moore):  $Y^t = \lambda(S^t)$

# 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

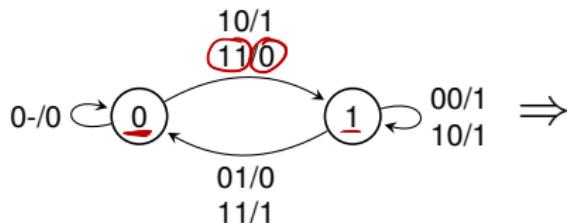
UNI  
FREIBURG

Prof. Dr. Christoph Scholl

Institut für Informatik  
Sommersemester 2023

# Entwurf sequentieller Schaltkreise

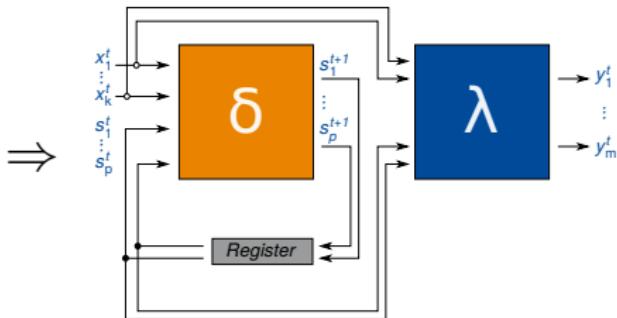
Zustandsdiagramm:



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:



# Entwurfsschritte

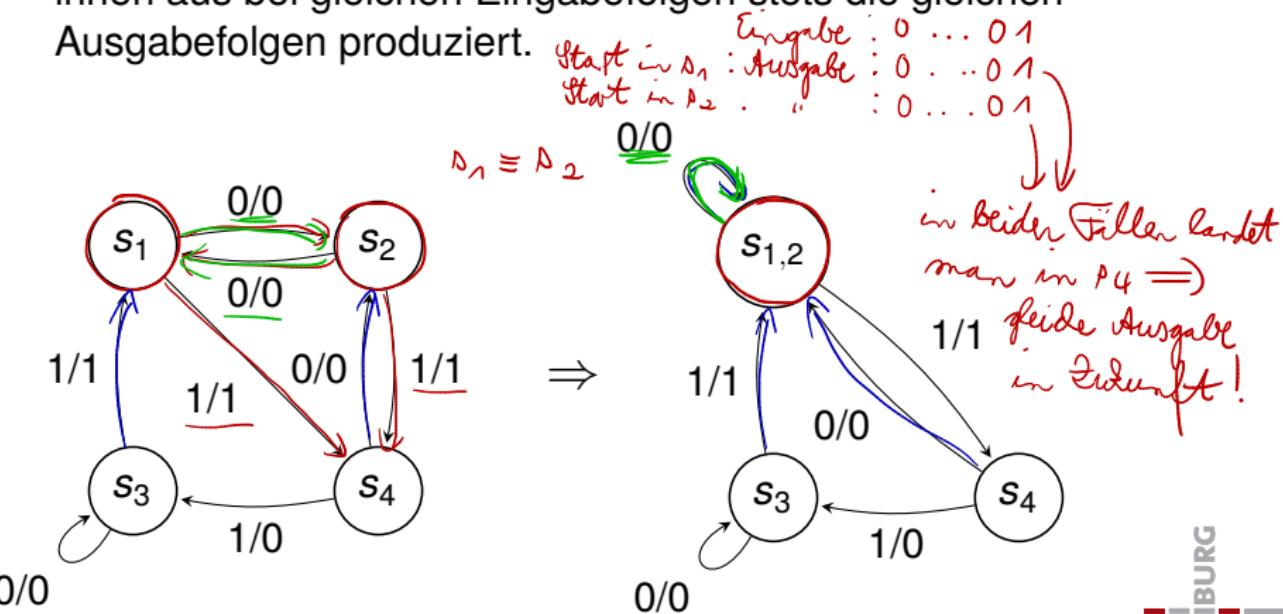
- Optimierung des Zustandsdiagramms:  
**Zustandsminimierung**
  - Identifikation der äquivalenten Zustände.
  - Ergebnis: Ein (evtl. kleineres) Zustandsdiagramm.
- Wahl der **Zustandskodierung**.
  - Ergebnis: Anzahl der Flipflops im Register,  
Funktionen  $\delta$  und  $\lambda$  (Zustands- und Ausgangstafel).
- Implementierung von  $\delta$  und  $\lambda$ .
  - Kombinatorische Logiksynthese, z.B. Quine-McCluskey.

# 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.

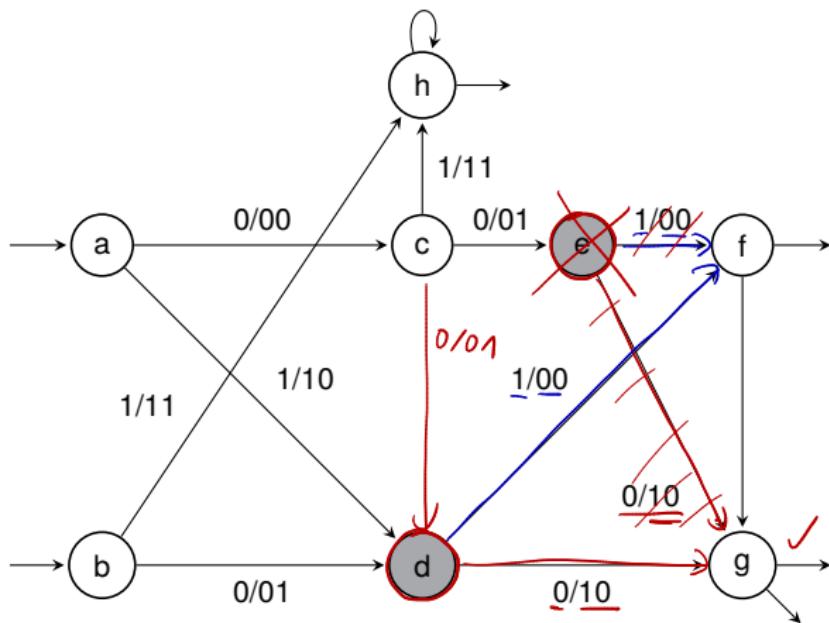


## Weiteres Beispiel (1/4)

---

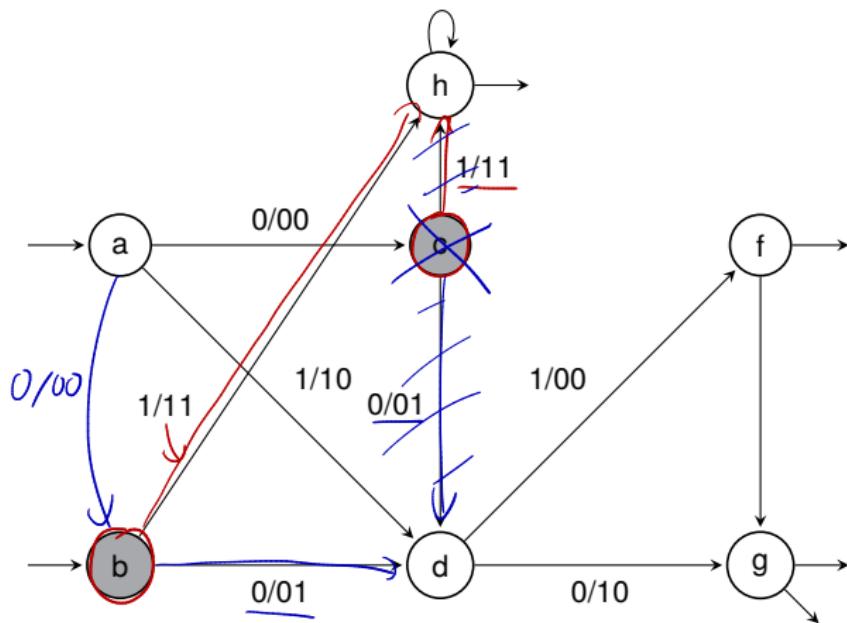
- **Hinreichende Bedingung** (unvollständiges Verfahren!): Wenn bei zwei Zuständen bei gleicher Eingabe auch die gleiche Ausgabe erzeugt wird und der gleiche Folgezustand angenommen wird, dann sind die Zustände sicherlich äquivalent.
- Äquivalente Zustände können durch einen einzigen Zustand ersetzt werden (siehe nächste Folie).

## Weiteres Beispiel (2/4)



Zustand  $e$  und  $d$  sind äquivalent, weil die hinreichende Bedingung erfüllt ist.

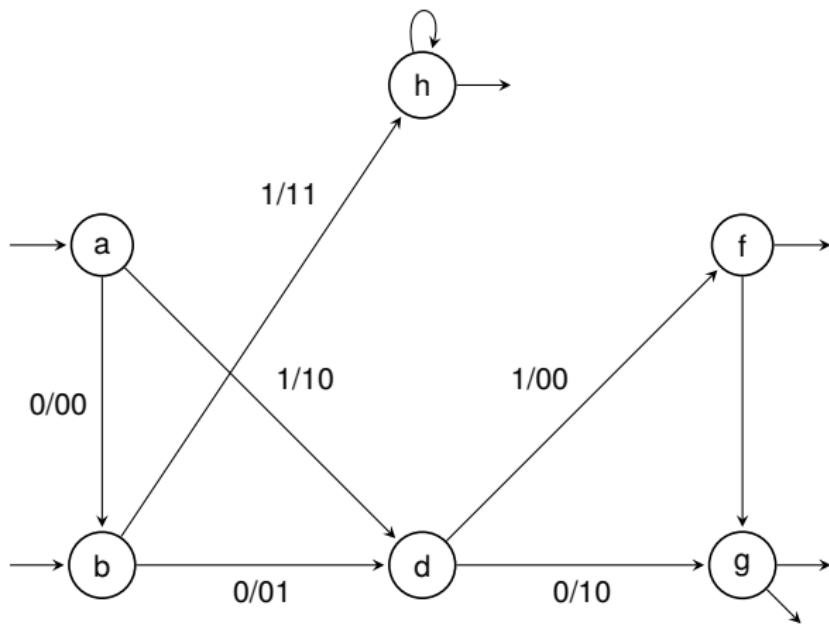
## Weiteres Beispiel (3/4)



Zustand **e** eliminiert.

Zustand **b** und **c** sind äquivalent gemäß hinreichender Bedingung.

## Weiteres Beispiel (4/4)



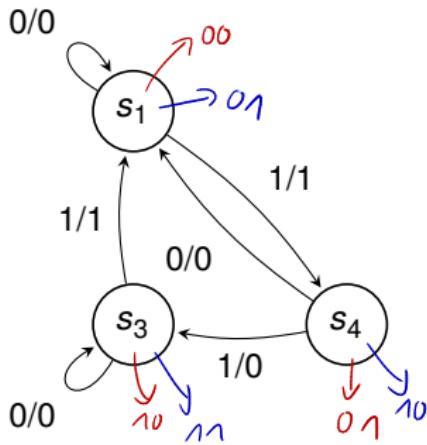
Zustand c eliminiert.

# 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  $S_1 \equiv 00, S_3 \equiv 10, S_4 \equiv 01 : \underline{4 \text{ Monome}}, \underline{9 \text{ Literale}}$

$$\begin{aligned}\delta_1(s, i) &= \underline{s_2}i + \underline{s_1}\bar{i} \\ \delta_2(s, i) &= \underline{\bar{s}_1}\underline{s_2}i \\ \lambda(s, i) &= \underline{\bar{s}_2}i\end{aligned}$$

Kodierung  $S_1 \equiv 01, S_3 \equiv 11, S_4 \equiv 10 : \underline{6 \text{ Monome}}, \underline{11 \text{ Literale}}$

$$\begin{aligned}\delta_1(s, i) &= \underline{s_1}s_2\bar{i} + \underline{\bar{s}_1}i + \underline{\bar{s}_2}i \\ \delta_2(s, i) &= \underline{s_1} + \underline{i} \\ \lambda(s, i) &= \underline{s_2}i\end{aligned}$$

- **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  $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$  durchlaufen werden.
- Ist  $x$  auf 0 gesetzt, so soll rückwärts gezählt werden, d.h. die Zahlenfolge  $3 \rightarrow 2 \rightarrow 1 \rightarrow 0$  durchlaufen werden.
- Am Ausgang ist der Zählerstand anzugeben.
- Start des Zählers mit Wert 0.

# Anzahl der Zustände im Zustandsdiagramm

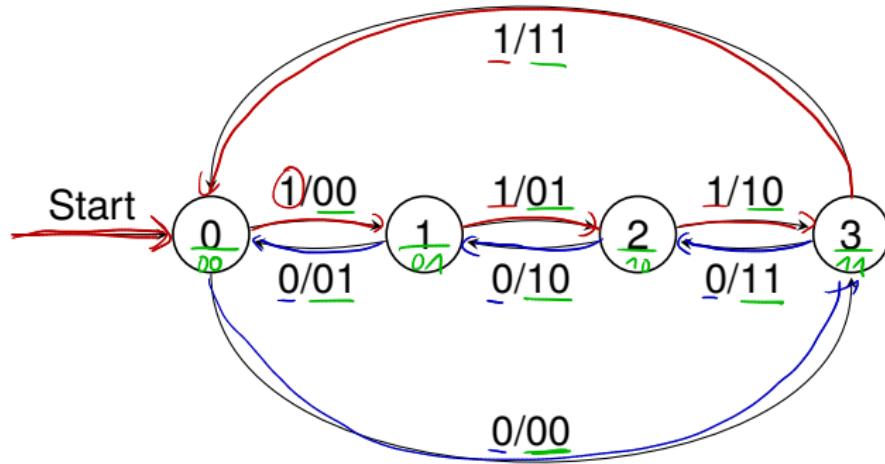
---

Wie viele Zustände benötigt das Zustandsdiagramm für den Zähler bei der gegebenen Spezifikation?

- a. 1
- b. 2
- c. 4
- d. 8
- e. 16

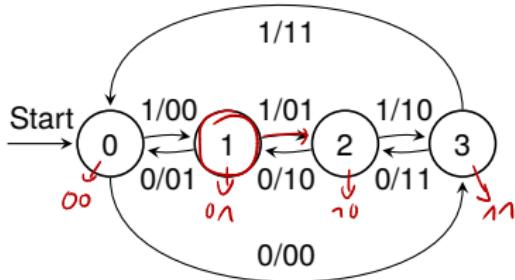
# Von der Textspezifikation zum Zustandsdiagramm

- 4 Zustände erforderlich.
- Startzustand 0.



# Vom Zustandsdiagramm zur Zustands- und Ausgangstafel

- Zustandsminimierung  $\Rightarrow$  Keine äquivalente Zustände.
- Zustandskodierung:  $0 \rightarrow 00, 1 \rightarrow 01, 2 \rightarrow 10, 3 \rightarrow 11$ .



	$x$	$z_1^t$	$z_0^t$	$z_1^{t+1}$	$z_0^{t+1}$	$y_1$	$y_0$
Vorwärts-zählen	1 1 1 1	0 0 1 1	0 1 0 1	0 1 1 0	1 0 1 0	0 0 1 1	0 1 0 0
Rückwärts-zählen	0 0 0 0	1 1 0 0	1 0 1 0	1 0 0 1	0 1 1 1	1 1 1 0	1 0 1 0

$\underbrace{\hspace{10em}}$  Eingänge       $\underbrace{\hspace{10em}}$  Ausgänge

# 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

Handwritten annotations:

- Red circles highlight specific bits:  $z_1^t$ ,  $z_0^t$ ,  $z_1^{t+1}$ ,  $z_0^{t+1}$ ,  $y_1$ ,  $y_0$ .
- Blue arrows point from the first row to the second.
- Red arrows point from the second row to the third.
- Red arrows point from the third row to the fourth.
- A blue bracket groups  $z_1^t$  and  $z_0^t$  under the heading "Rückwärts-zählen".
- A red bracket groups  $z_1^{t+1}$  and  $z_0^{t+1}$  under the heading "Vorwärts-zählen".
- Equations on the right side show:  
 $y_0 = z_0^t$   
 $y_1 = z_1^t$   
 $\bar{z}_0^t$  is shown in a circle with a red arrow pointing to it.
- Below the tables, two equations are shown:  
 $z_0^{t+1} = \underline{xz_1^t \bar{z}_0^t} + \underline{xz_1^t \bar{z}_0^t} + \underline{\bar{x}z_1^t \bar{z}_0^t} + \underline{\bar{x}z_1^t \bar{z}_0^t} = \underline{\bar{z}_0^t}$   
 $z_1^{t+1} = \underline{xz_1^t z_0^t} + \underline{xz_1^t \bar{z}_0^t} + \underline{\bar{x}z_1^t z_0^t} + \underline{\bar{x}z_1^t \bar{z}_0^t}$
- Below the second equation, labels indicate the number of products:  
 $\uparrow 2 \text{ p.m.d.}$   
 $\uparrow 2 \text{ p.m.d.}$   
 $\uparrow 2 \text{ p.m.d.}$   
 $\uparrow 0 \text{ p.m.d.}$

Übergangsfunktion:

$$z_0^{t+1} = \underline{xz_1^t \bar{z}_0^t} + \underline{xz_1^t \bar{z}_0^t} + \underline{\bar{x}z_1^t \bar{z}_0^t} + \underline{\bar{x}z_1^t \bar{z}_0^t} = \underline{\bar{z}_0^t}$$

$$z_1^{t+1} = \underline{xz_1^t z_0^t} + \underline{xz_1^t \bar{z}_0^t} + \underline{\bar{x}z_1^t z_0^t} + \underline{\bar{x}z_1^t \bar{z}_0^t}$$

$\uparrow 2 \text{ p.m.d.}$     $\uparrow 2 \text{ p.m.d.}$     $\uparrow 2 \text{ p.m.d.}$     $\uparrow 0 \text{ p.m.d.}$

# Implementierung des komb. Kerns: Logikminimierung

Ausgangsfunktion:

$$y_0^t = z_0^t, \quad y_1^t = z_1^t$$

Übergangsfunktion:

$$z_0^{t+1} = x\bar{z}_1^t z_0^t + xz_1^t \bar{z}_0^t + \bar{x}z_1^t \bar{z}_0^t + \bar{x}\bar{z}_1^t \bar{z}_0^t$$

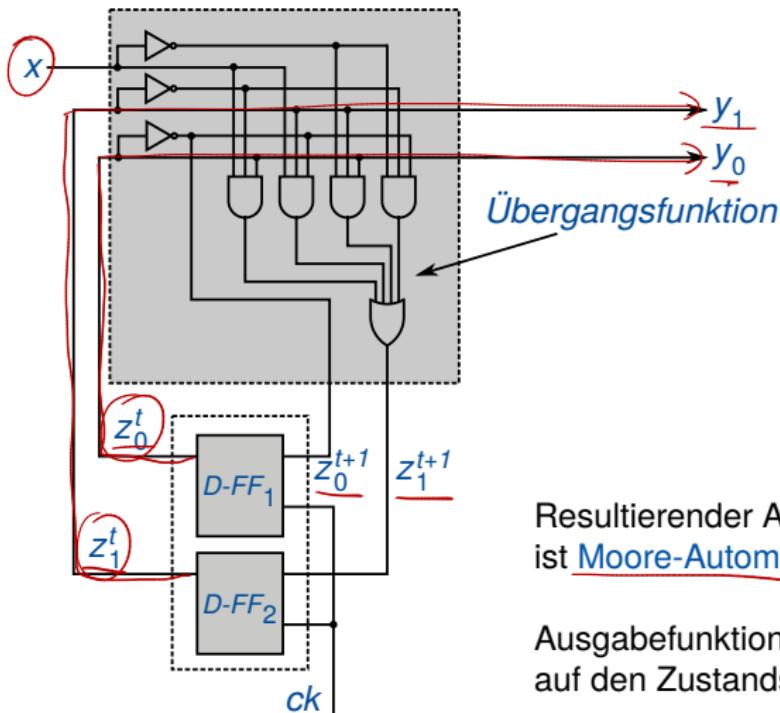
$$z_1^{t+1} = x\bar{z}_1^t z_0^t + xz_1^t \bar{z}_0^t + \bar{x}z_1^t z_0^t + \bar{x}\bar{z}_1^t \bar{z}_0^t$$

Minimierung:

$$\underline{z_0^{t+1} = \bar{z}_0^t}$$

$$\underline{z_1^{t+1} = x\bar{z}_1^t z_0^t + xz_1^t \bar{z}_0^t + \bar{x}z_1^t z_0^t + \bar{x}\bar{z}_1^t \bar{z}_0^t}$$

# Beispiel: Ergebnis





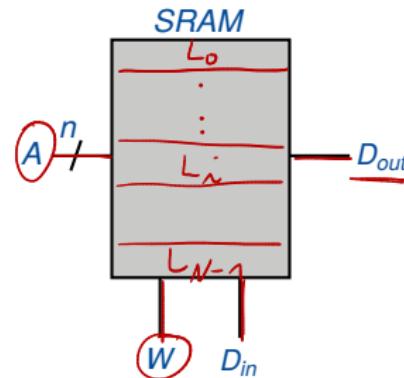
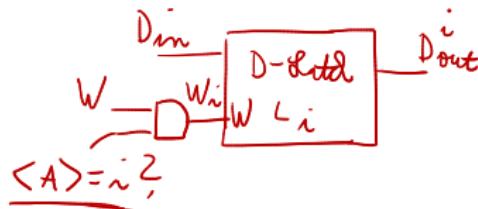
# Kapitel 4 – Sequentielle Logik

1. Speichernde Elemente
2. Sequentielle Schaltkreise
3. Entwurf sequentieller Schaltkreise
4. **SRAM, Treiber**
5. Anwendung: Datenpfade von ReTI

# Static Random-Access Memory – SRAM

- Sei  $n \in \mathbb{N}, N = 2^n$ . Ein N-Bit statischer Speicher oder **SRAM** hat:

- $n$  Eingänge  $A = (\underline{A_{n-1} \dots A_0})$ , „Adresse“,
- Dateneingang  $D_{in}$ ,
- Datenausgang  $D_{out}$ ,
- Kontrollsignal  $W$ , „write“



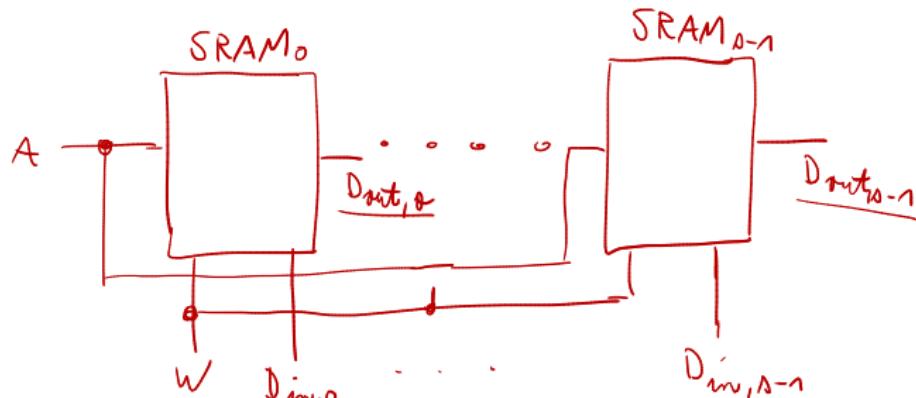
# SRAM: Funktionalität

---

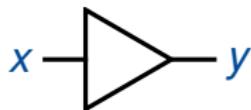
- Der Speicher enthält  $N = 2^n$  Speicherzellen  $L_0, \dots, L_{N-1}$  (z.B. D-Latches), die je ein Bit speichern können.
- Zelle  $L_{\langle A \rangle}$  wird mit Hilfe der Adresse  $\underline{A} = (\underline{A_{n-1}} \dots \underline{A_0})$  ausgewählt.
  - Lesen: An  $D_{out}$  erscheint der Inhalt von  $\underline{L_{\langle A \rangle}}$ .
  - Schreiben: Durch Schreibpuls an  $W$  wird  $D_{in}$  nach  $L_{\langle A \rangle}$  übernommen.  
Beim Schreiben wird nur lokales Schrebsignal von Zelle  $L_{\langle A \rangle}$  aktiviert.

# $N$ -Bit-SRAM, $N \times s$ -Bit-SRAM: Aufbau

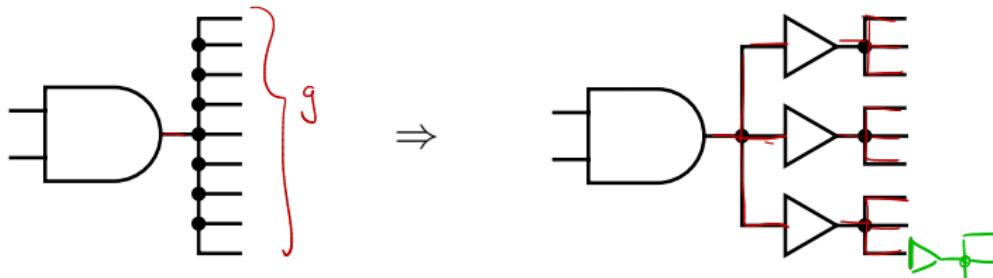
- Ein  $\textcolor{red}{N} \times \underline{s}$ -Bit-SRAM besteht aus  $\underline{s}$   $N$ -Bit SRAMs mit gemeinsamen Adress- und Schreibsignalen.
- $s$  heißt Bitbreite des  $N \times s$ -Bit-SRAMs.



# Treiberbäume



- Ein **Treiber** ist ein Gatter mit einem Eingang  $X$  und einem Ausgang  $Y$ , das die Identität  $Y = X$  berechnet.
- **Fanout-Beschränkung**: Aus elektrischen Gründen kann eine Leitung nicht auf beliebig viele Gattereingänge verzweigen.
- Eingesetzt, um Fanout-Beschränkung zu überwinden.
- Beispiel: Fanout-Beschränkung von 3.

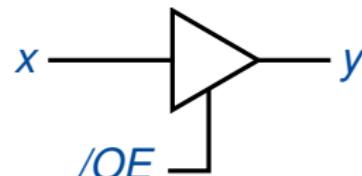


# Tristate-Treiber und Busse

- Tristate-Treiber sind Treiber mit Eingangssignal  $x$  und zusätzlichem Signal  $/OE$ , dem Output-Enable-Signal.

- Am Ausgang  $y$  erscheint

$$y = \begin{cases} x, & \text{falls } /OE = 0 \\ Z, & \text{sonst } /OE = 1 \end{cases}$$

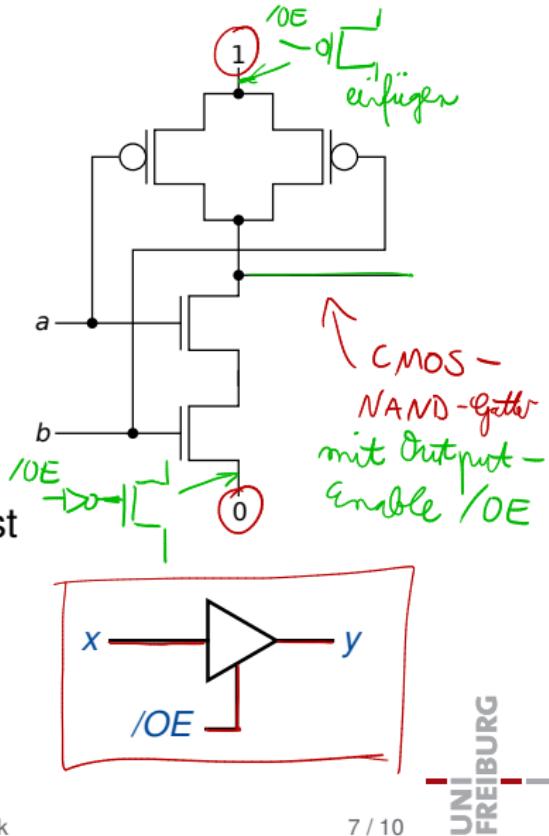


- $Z$  bezeichnet den Zustand hoher Impedanz (high-Z).

„isolierter Zustand  
„als ob x und y unverbunden wären“

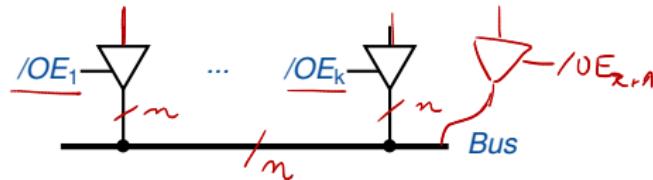
## Zustand hoher Impedanz

- 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.



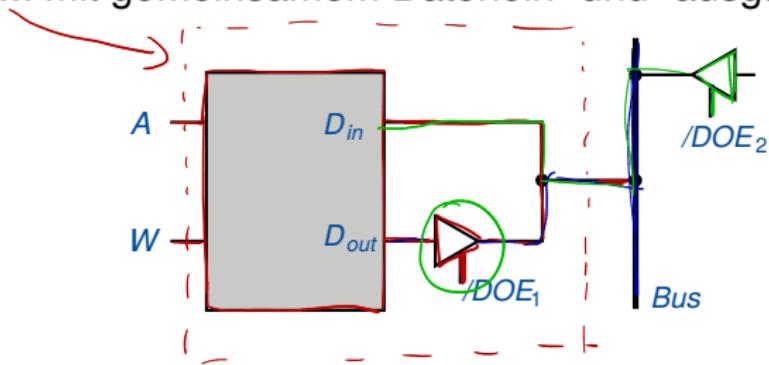
# Bus vs. Multiplexer

---

- *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,  $\underline{/DOE_1}$  disabled.



# Kapitel 4 – Sequentielle Logik

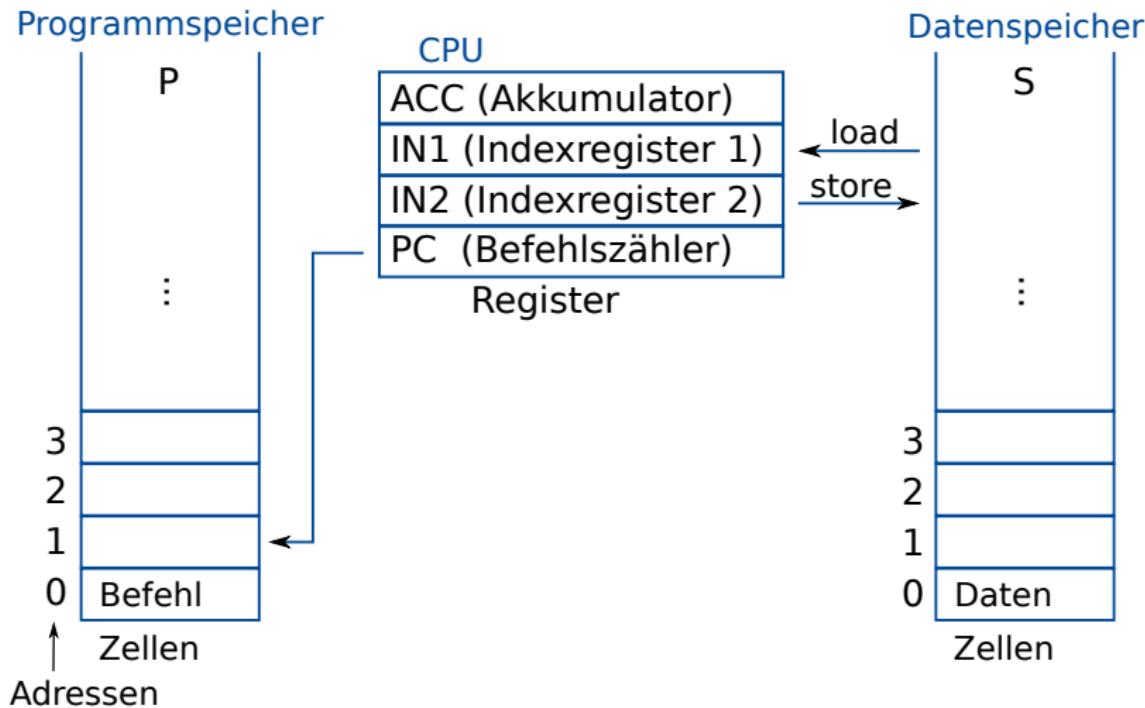
1. Speichernde Elemente
2. Sequentielle Schaltkreise
3. Entwurf sequentieller Schaltkreise
4. SRAM
5. **Anwendung: Datenpfade von ReTI**

Albert-Ludwigs-Universität Freiburg

Prof. Dr. Christoph Scholl

Institut für Informatik  
Sommersemester 2023

# 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>

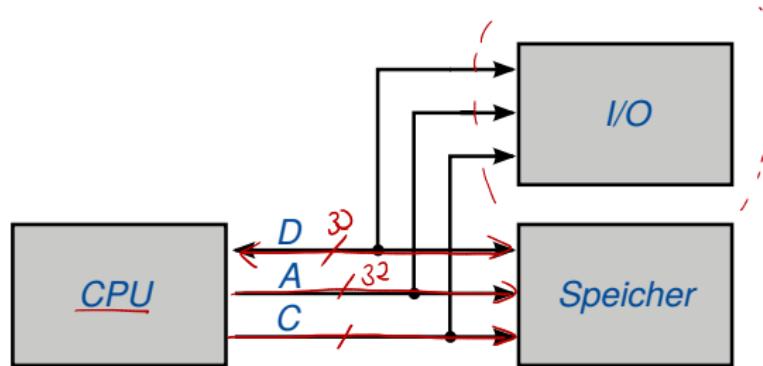
# Zur Erinnerung: ReTI-Befehle im Überblick

	31 30	29 28	27 26	25 24	23 ...	0
Load	<u>0 1</u>	M	*	D	i	
	31 30	29 28	27 26	25 24	23 ...	0
Store	<u>1 0</u>	M	S	D	i	
	31 30	29	28	27	26	25 24
Compute	0 0	MI	F		D	i
	31 30	29 28	27	26	25 24	23 ...
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).

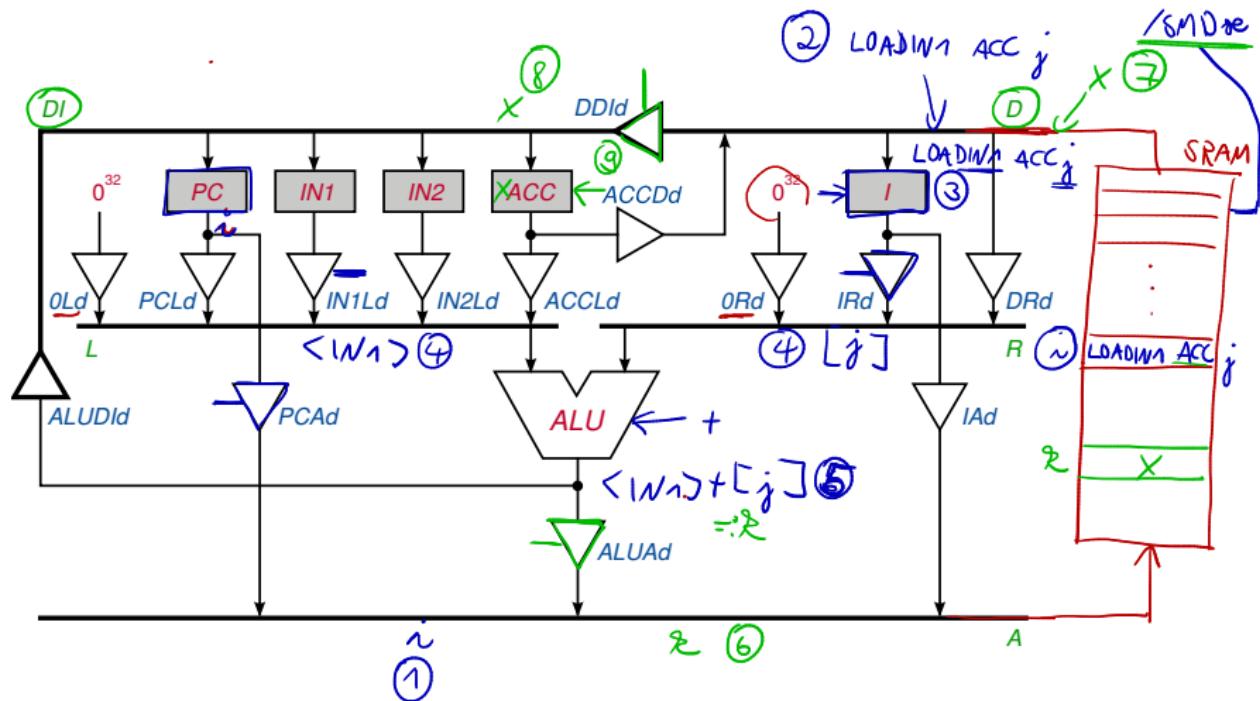


# Umsetzung von ReTI: Interne Sicht

---

- *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



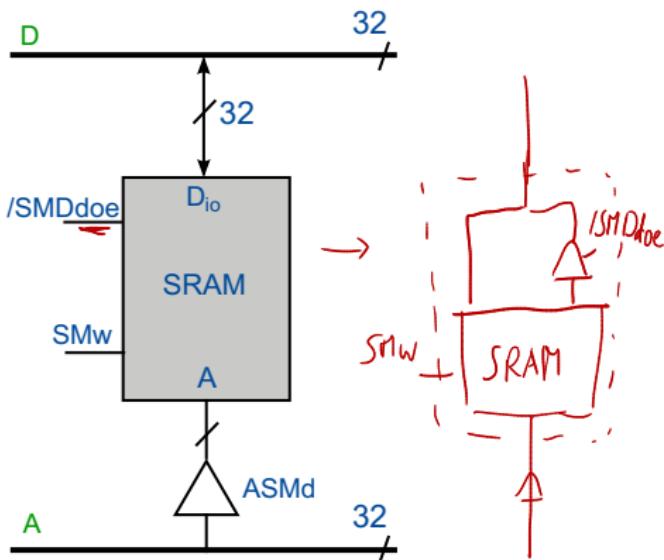
# Hinweise zum Schaltbild

---

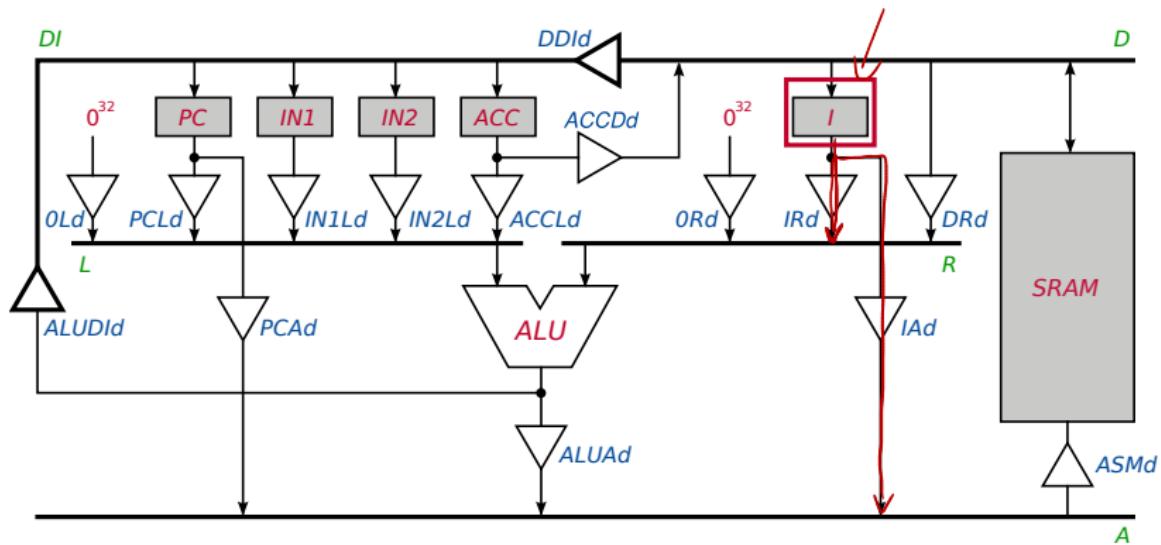
- Namenskonvention für Treiber: Treiber zwischen Bus/Baustein  $X$  und Bus/Baustein  $Y$ :  $XYd$ ;
  - Output-Enable-Signal:  $/XYdoe$ .
  - $0Ld$  und  $0Rd$  können  $0^{32}$  auf  $L$  bzw.  $R$  legen.
- Busse  $A$  und  $D$  sind an den Speicher (sowie I/O-Geräte) angeschlossen, s. nächste Folie.
- 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 Schaltbils



# 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).
- $\underline{I[23 : 0]}$ : Speicheradresse oder Konstante für die ALU.

- Speicheradresse wird mit acht Nullen aufgefüllt.

$0^8 I[23 : 0]$  wird auf Bus  $A$  gelegt ( $IAd$  enabled).

$LOAD, STORE, Compute\ memory$

- Natürliche 24-Bit-Konstante wird mit acht Nullen aufgefüllt.

$0^8 I[23 : 0]$  wird auf  $R$  gelegt ( $IRd$  enabled).

$ANDI, ORI, OPLUS, LOADI$

- Ganzzahlige 24-Bit-Konstante wird vorzeichenverweitert.

$sext(I[23 : 0])$  wird auf  $R$  gelegt ( $IRd$  enabled).

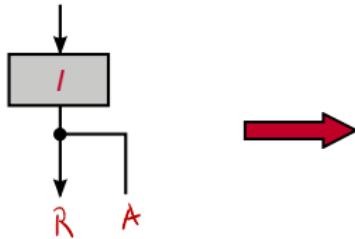
$$= I[23] 8 \quad I[23 : 0]$$

$(ADDI, SUBI, LOADINj, STOREINj, LOADR)$

$\} \quad sext = 0$

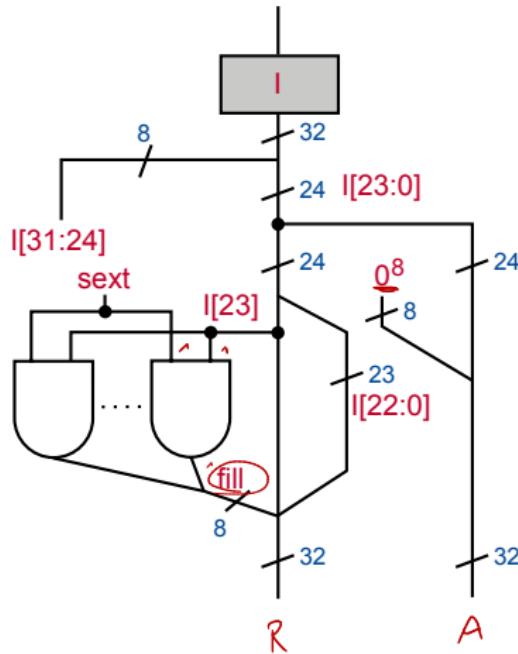
$\} \quad sext = 1$

# Verfeinerung



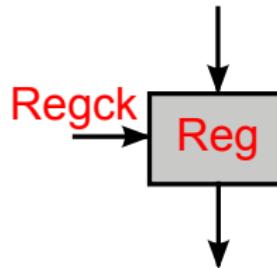
## ■ Neues Kontrollsignal sext:

- sext = 0: Ersetze  $I[23 : 0]$  durch  $0^8[23 : 0]$ .
- sext = 1: Ersetze  $I[23 : 0]$  durch  $\text{sext}(I[23 : 0])$ .



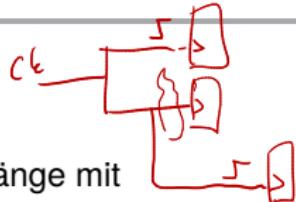
# Weitere Verfeinerung für Register (1/3)

- 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)

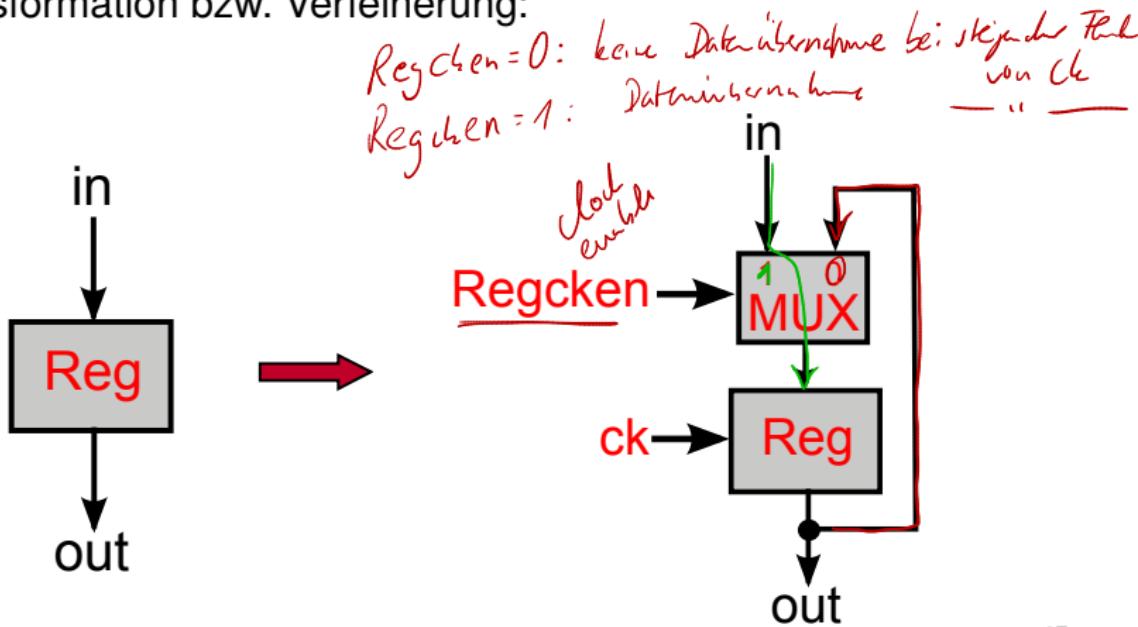


- Gründe für „**Designrule**“, die es verbietet, Clockeingänge mit berechneten Datensignalen zu verbinden:
  - Spezielle Methoden („**Clock-Tree-Synthese**“) gewährleisten, dass die steigende Flanke der globalen Clock an allen Clockeingängen zum **gleichen** Zeitpunkt ankommt (z. B. durch Ausgleich des 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)



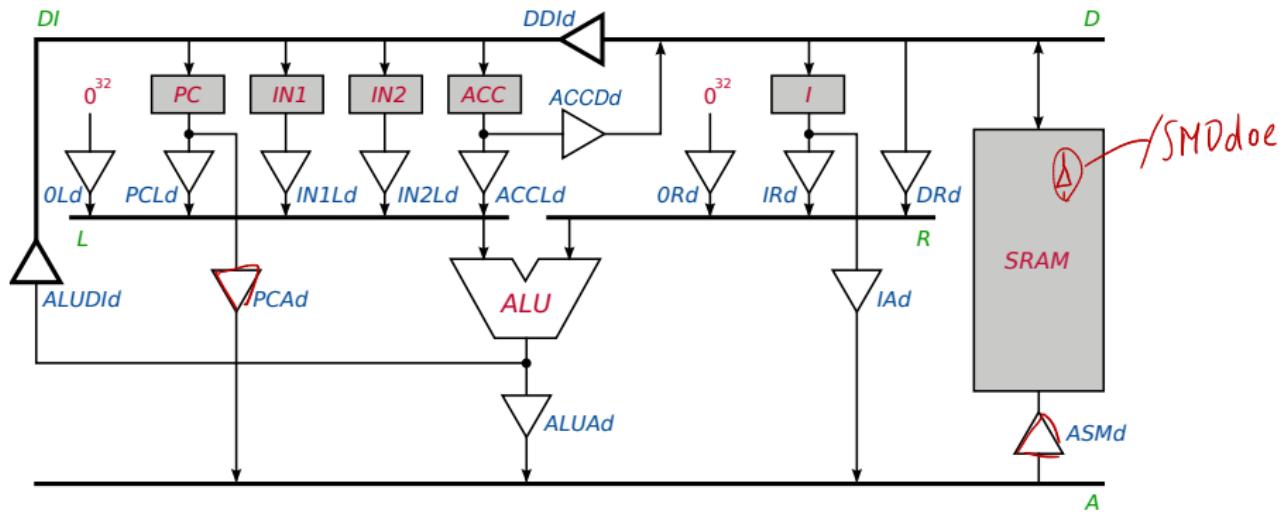
## Weitere Verfeinerung für Register (3/3)

### ■ Transformation bzw. Verfeinerung:



- 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, Enable für Register-Clocks.
  - 3 Sequentielle Synthese.

# Datenpfade: Fetch-Phase



# SMILE – Fetch-Phase, Treiber

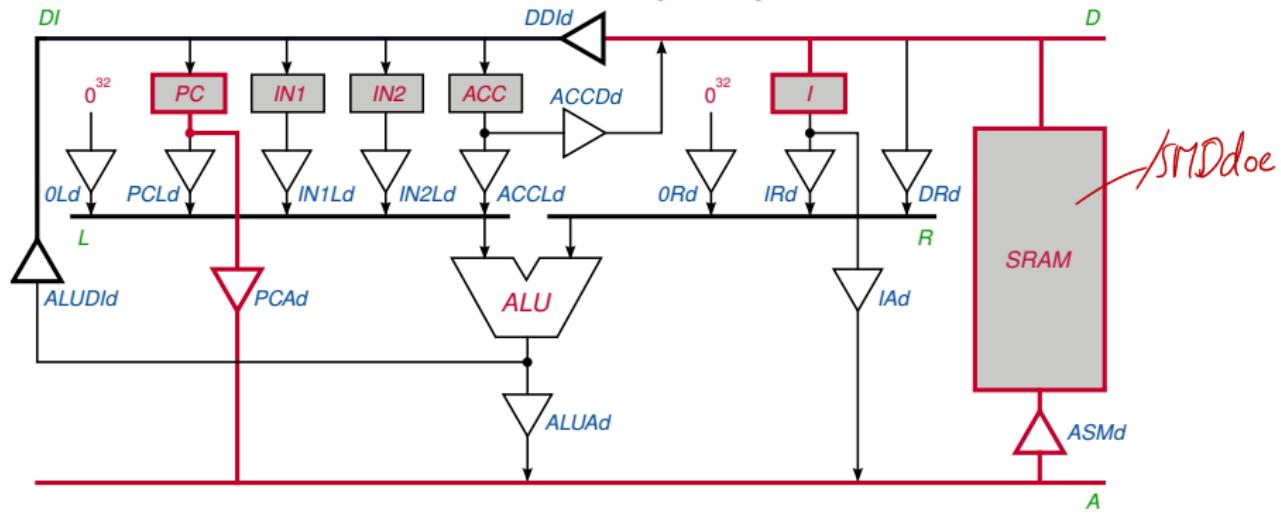
---

Welche Treiber müssen in der Fetch-Phase enabled sein?

- a. ALUAd
- b. ASMd
- c. DDId
- d. IAd
- e. IRd
- f. PCAd
- g. PCLd
- h. SMDd

# Datenpfade: Fetch-Phase

$PCAd = 0$ ,  $ASMDoe = 0$ ,  $SMDoc = 0$ ,  $Icken = 1$   
alle anderen Treiber inaktiv!

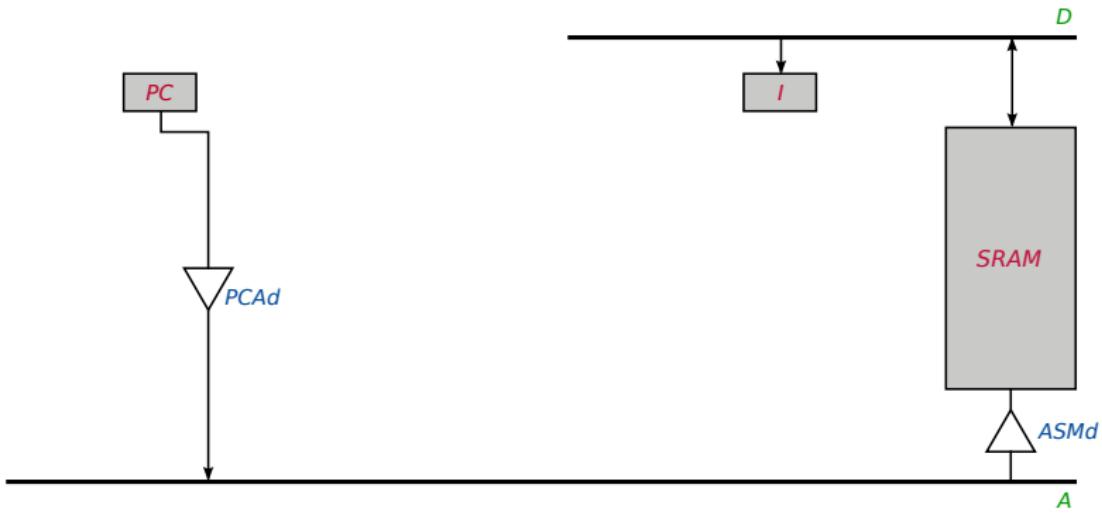


# Bedeutung des Diagramms

---

- In der Fetch-Phase muss:
  - Bus *A* mit *PC* verbunden sein, d.h. *PCAd* enabled.
  - Bus *D* mit *Speicher* verbunden sein, d.h. *SMDd* 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. *Icken* = 1, */PCAdoe* = 0, */SMDdoe* = 0, */ALUAdoe* = 1, usw. (Output-Enable-Signale active low!)

# Fetch: Die durchgeschalteten Pfade



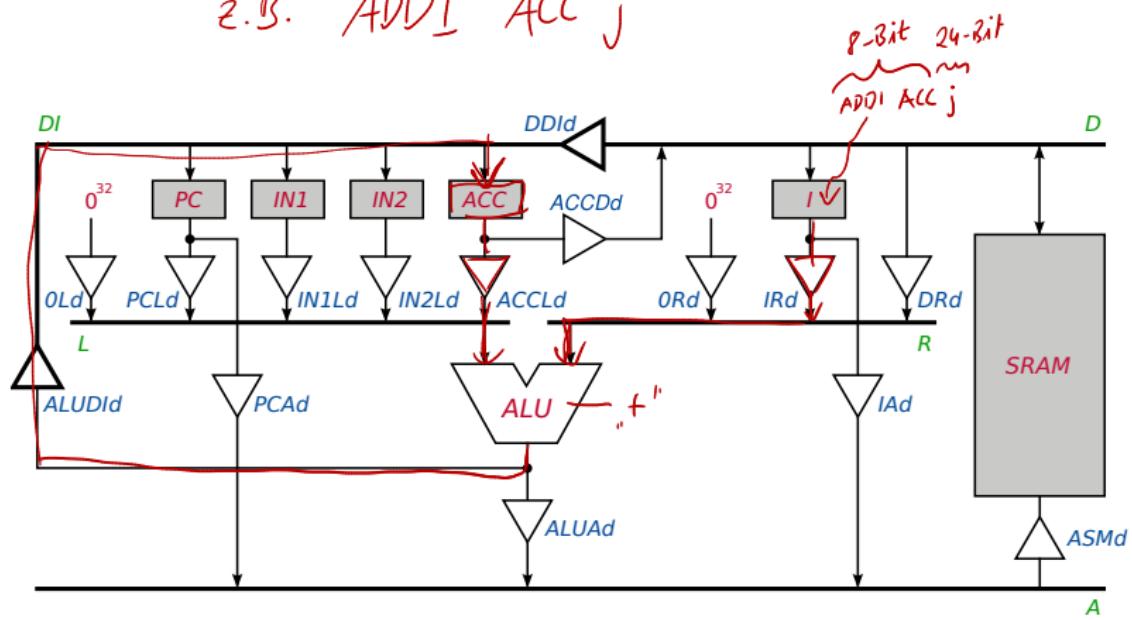
# Datenpfade in der Execute-Phase

---

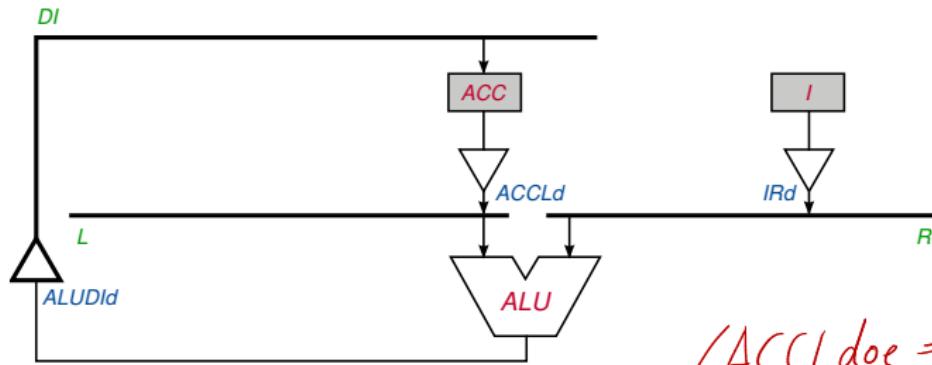
- Betrachte die Execute-Phase unterschiedlicher Befehlstypen.
  - *Compute Immediate*,
  - *Compute Memory*,
  - *JUMP*,
  - *LOAD*,
  - *LOADIN1* (*LOADIN2* analog),
  - *LOADI*,
  - *STORE*,
  - *STOREIN1*,
  - *MOVE*.

# Datenpfade: Compute Immediate (1/2)

z.B. ADDI ACC j



## Datenpfade: *Compute Immediate* (2/2)



$$\text{/ACCLd}oe = 0$$

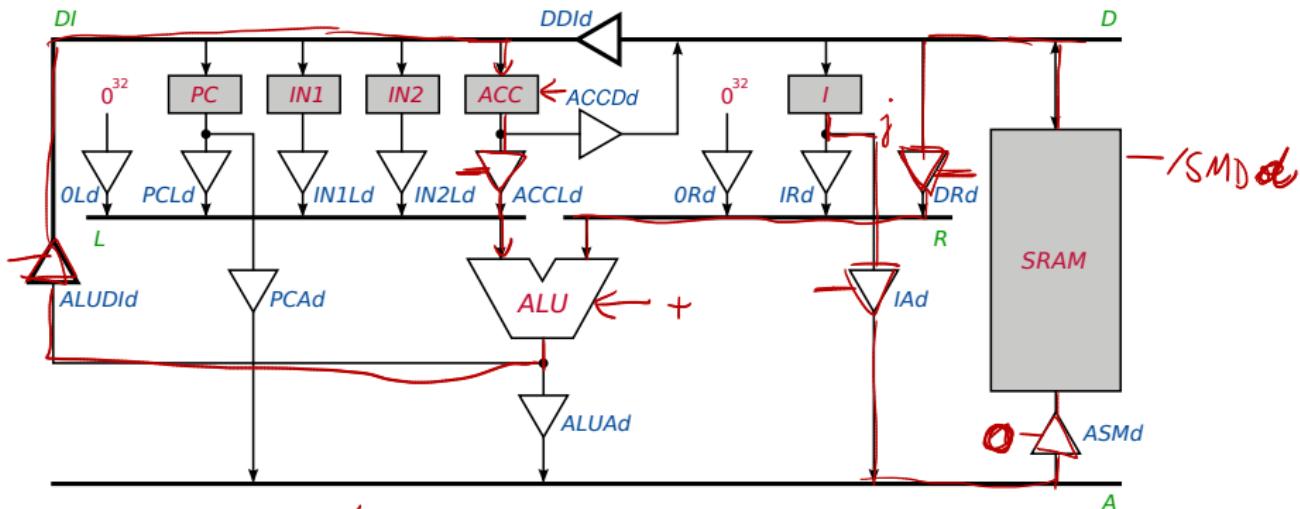
$$\text{/IRd}oe = 0$$

$$\text{/ALUDld}oe = 0$$

$$\text{ACC}cken = 1$$

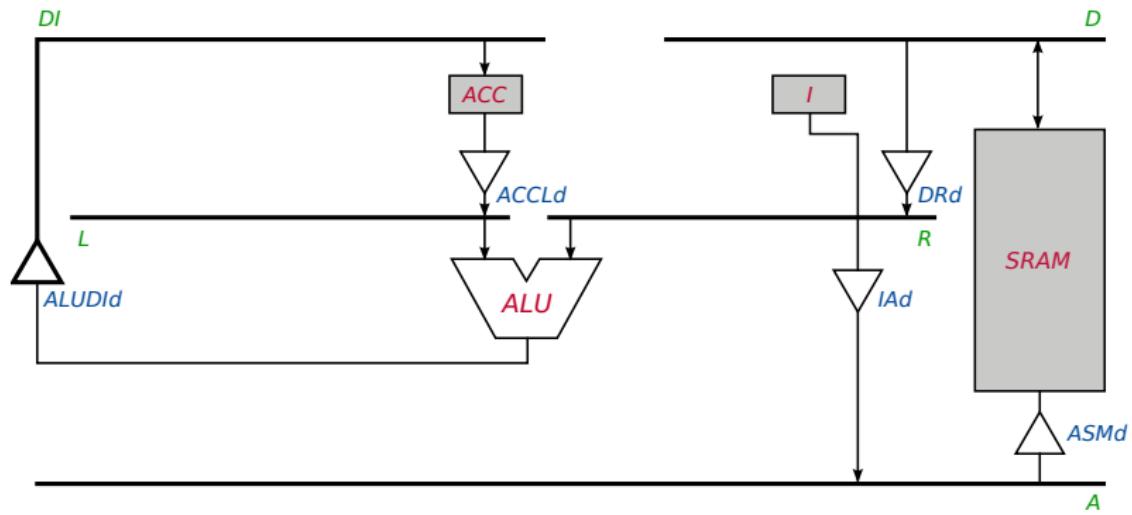
# Datenpfade: Compute Memory (1/2)

z. B.: ADD ACC j



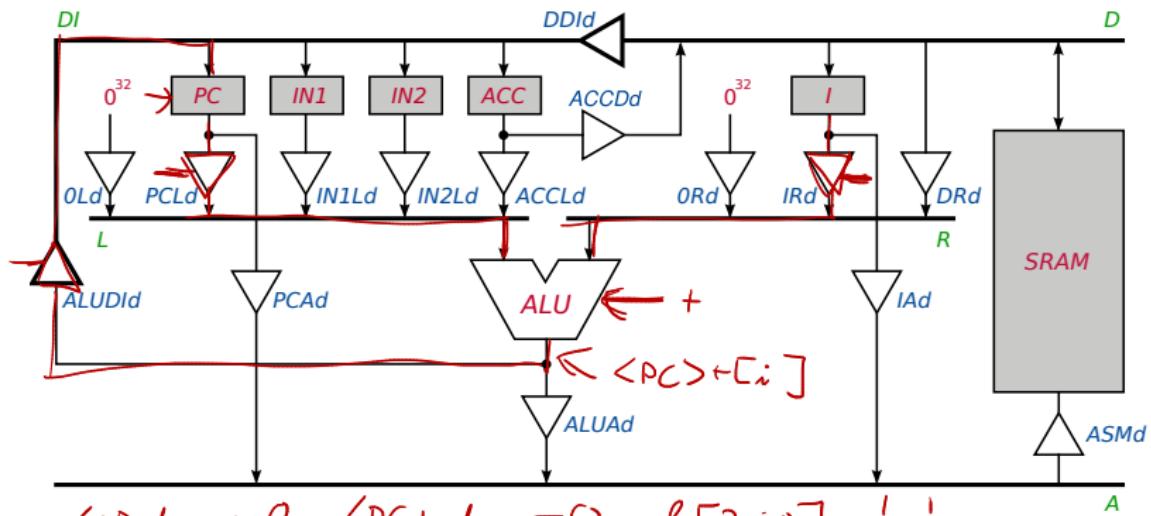
$/ACCLd \text{ oe} = 0, /IAd \text{ oe} = 0, /SMd \text{ oe} = 0, /DRd \text{ oe} = 0$   
 $f[0..2] = '+', /ALUDId \text{ oe} = 0, ACCd \text{ den} = 1$

# Datenpfade: *Compute Memory* (2/2)



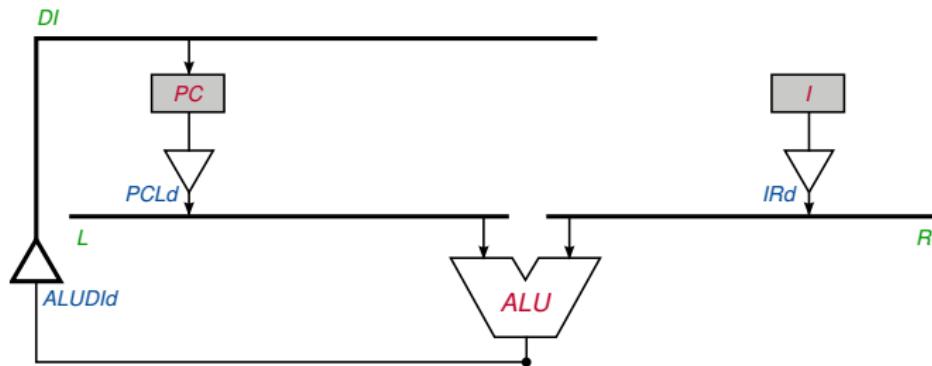
# Datenpfade: JUMP (1/2)

Bsp.:  $JUMP > i$

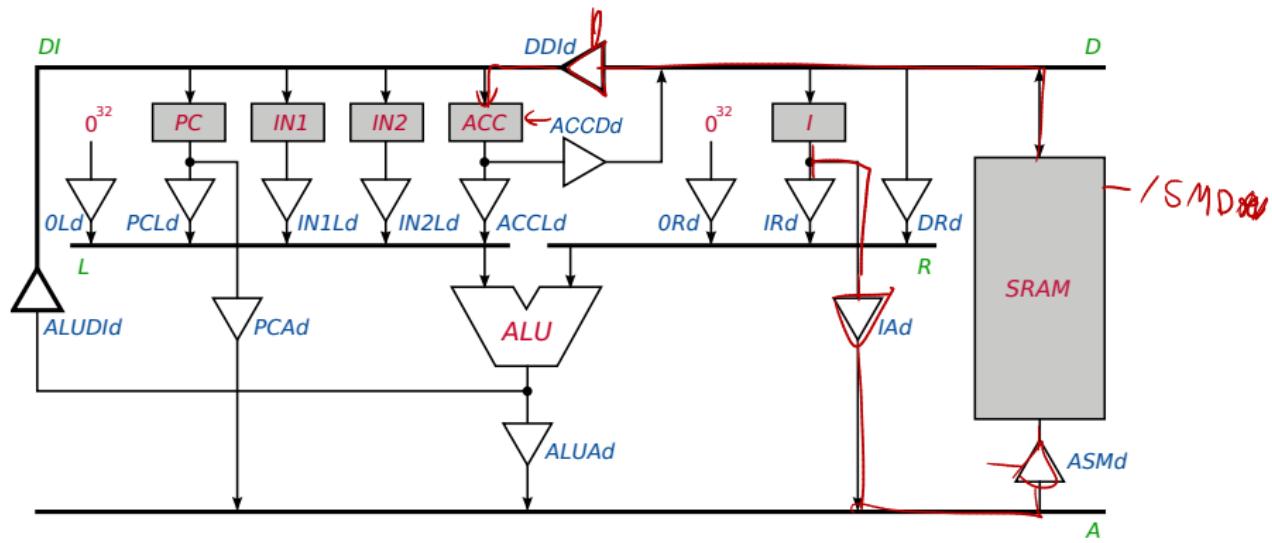


$/IR.doe = 0, /PCL.doe = 0, f[2:0] = '++',$   
 $/ALUD1.doe = 0$   
 $/PC LOAD = 0, \text{ falls Sprungbedingung erfüllt ist}$   
 $PCden = 1$

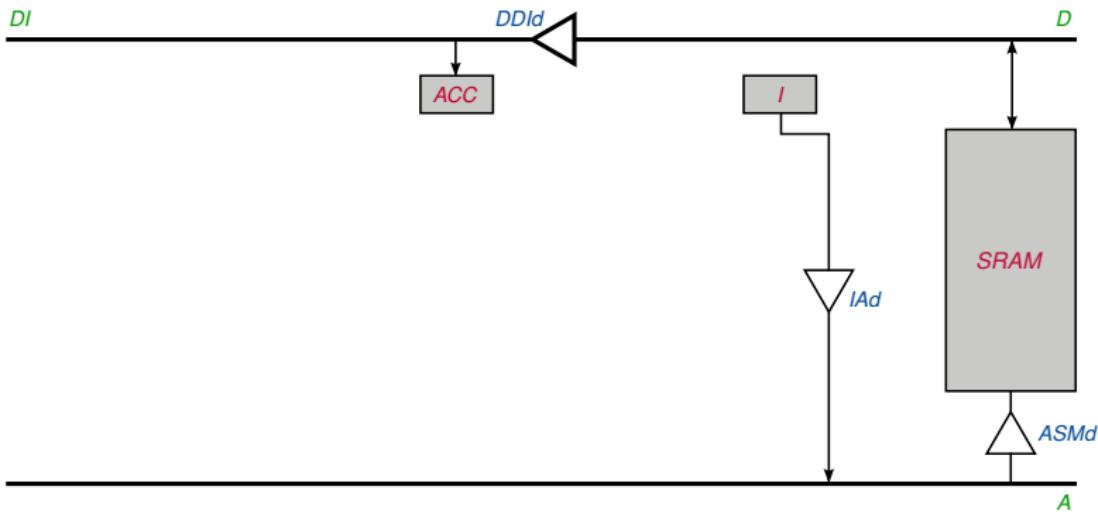
## Datenpfade: *JUMP* (2/2)



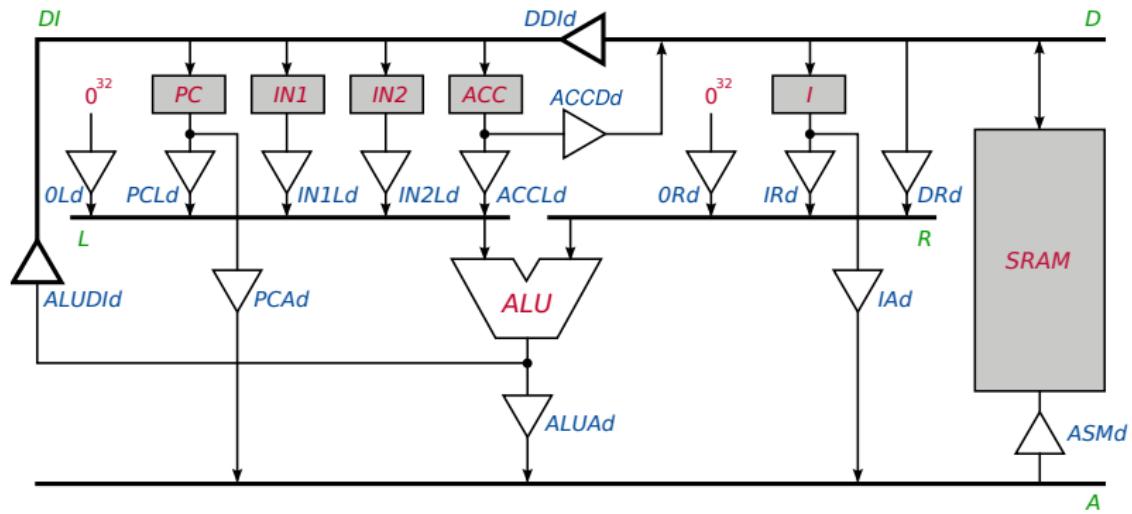
# Datenpfade: *LOAD i* (1/2)



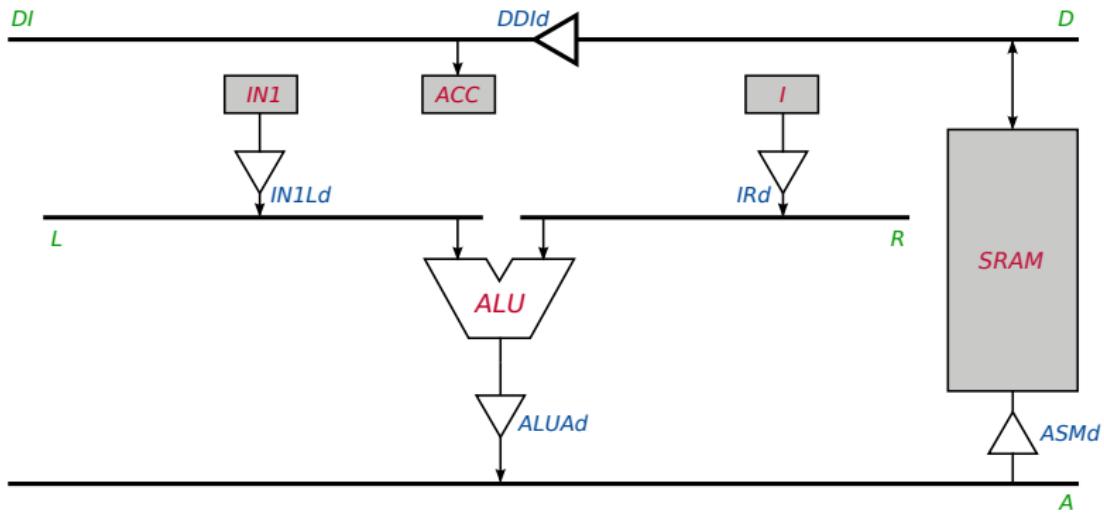
## Datenpfade: *LOAD i* (2/2)



# Datenpfade: *LOADIN1 i* (1/2)

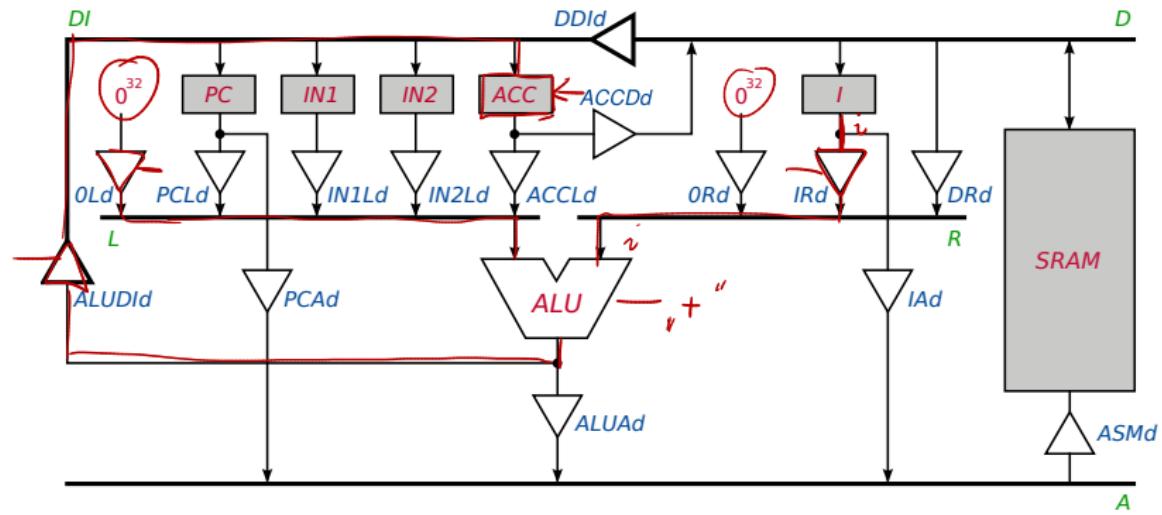


# Datenpfade: *LOADIN1 i* (2/2)

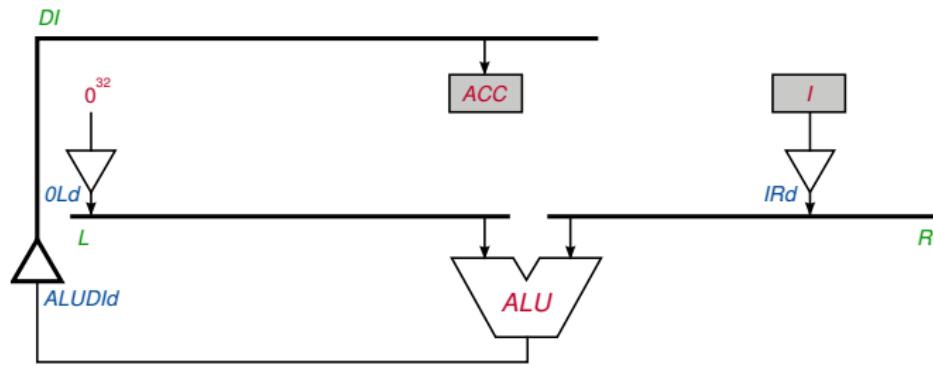


# Datenpfade: *LOADI i* (1/2)

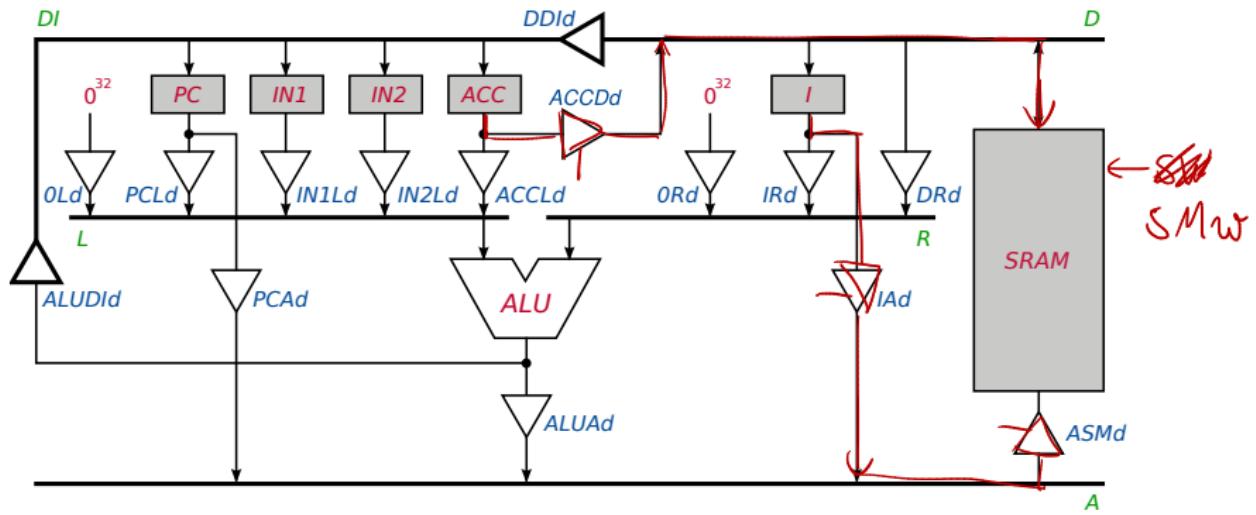
$[ACC] := 0^8i$



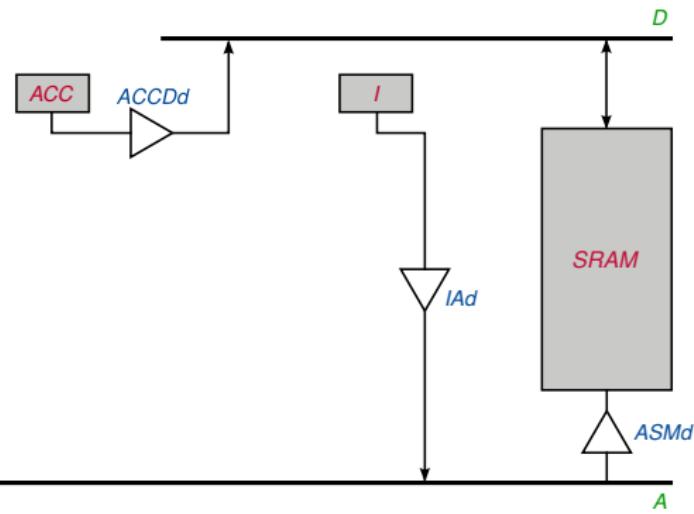
# Datenpfade: *LOADI i* (2/2)



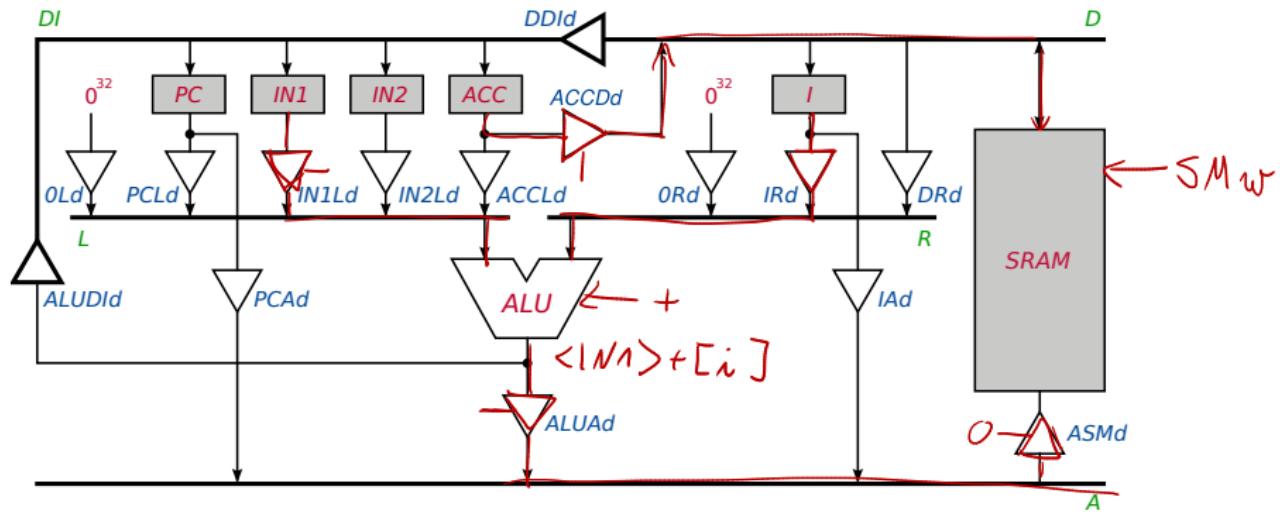
# Datenpfade: *STORE i* (1/2)



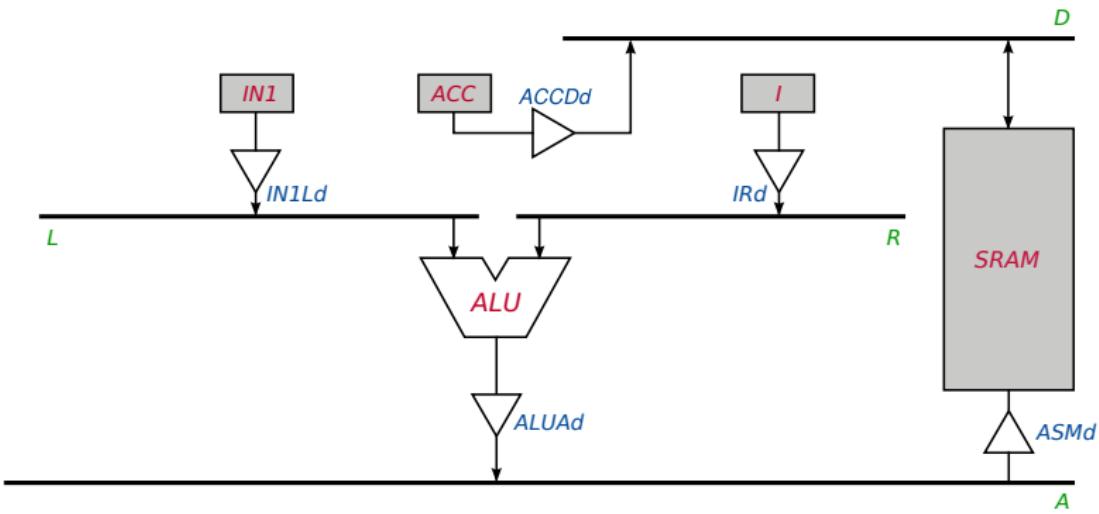
## Datenpfade: *STORE i* (2/2)



# Datenpfade: *STOREIN1 i* (1/2)

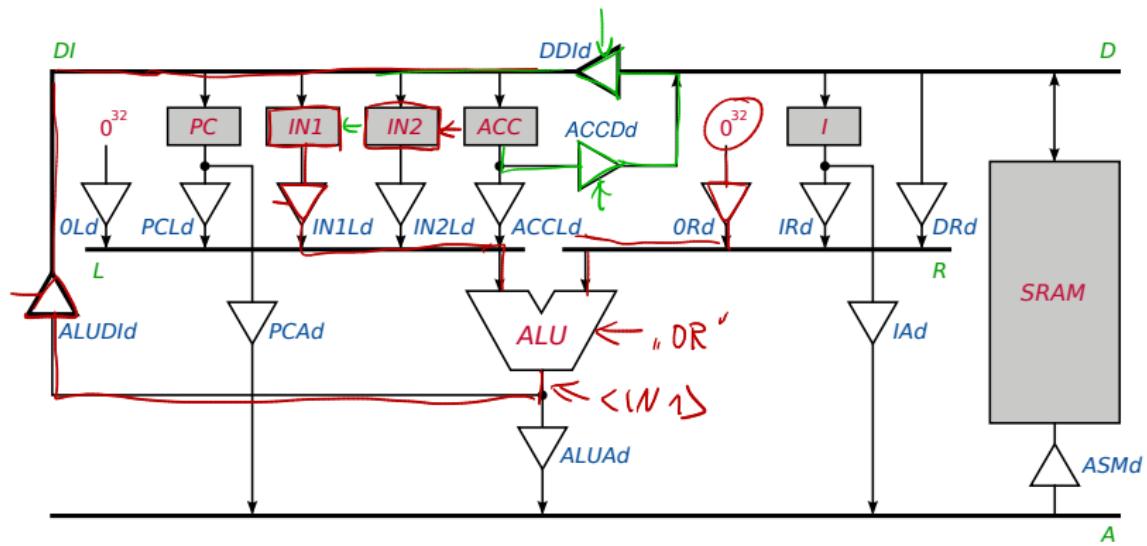


# Datenpfade: *STOREIN1 i* (2/2)

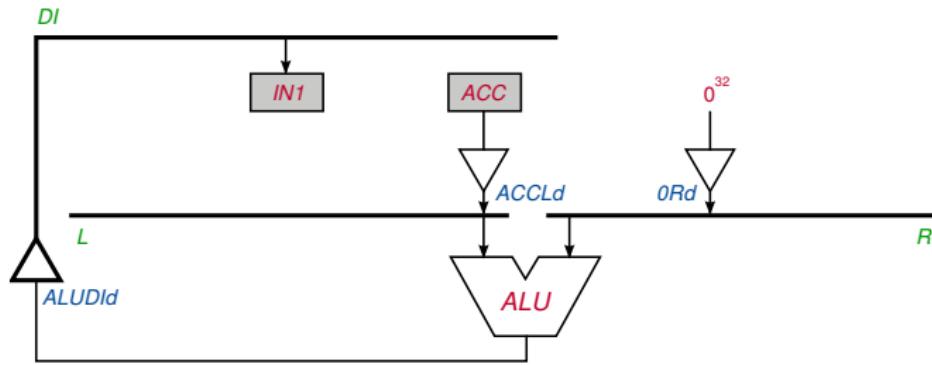


# Datenpfade: MOVE ACC IN1 (1/2)

MOVE IN1 IN2



# 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 bei den Vorlesungsmaterialien unter „Zusatzmaterial/Neumi“
  - 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 siehe Anleitungen in diesem Verzeichnis.

# 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 Kontrollsignale  $/SMDdoe$  (active low),  $SMw$ .

# 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

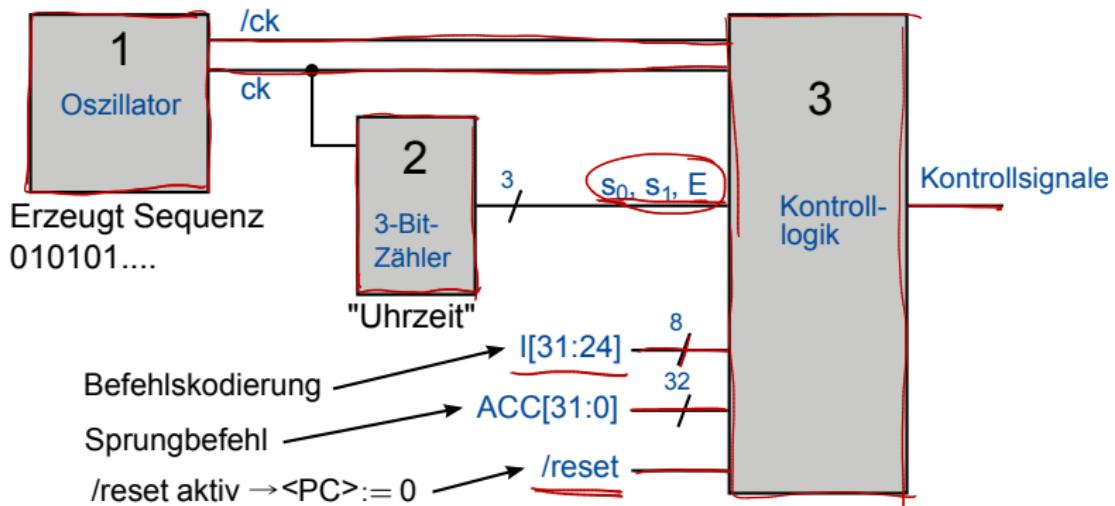
---

- 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.

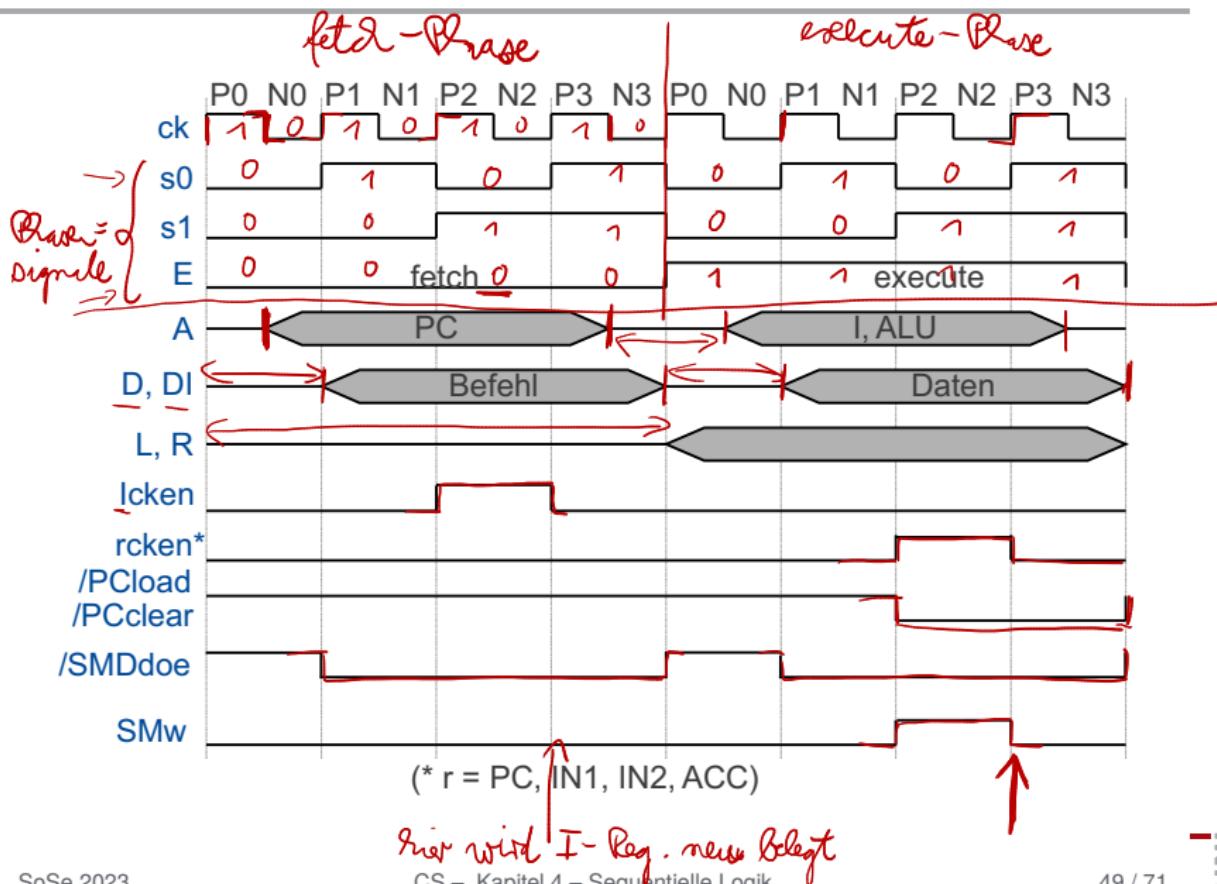
## Erzeugung der Phasensignale



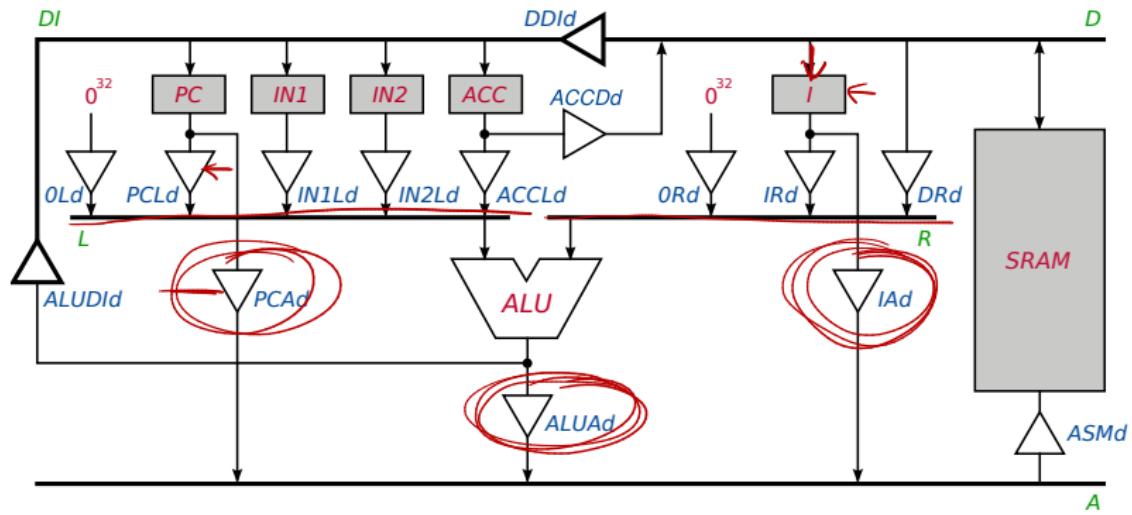
3-Bit-Zähler, zählt bei jeder positiven Flanke von  $ck$  um 1 hoch.



# Idealisiertes Timing-Diagramm



# Zur Erinnerung: Datenpfade



# Entwurfsziele

---

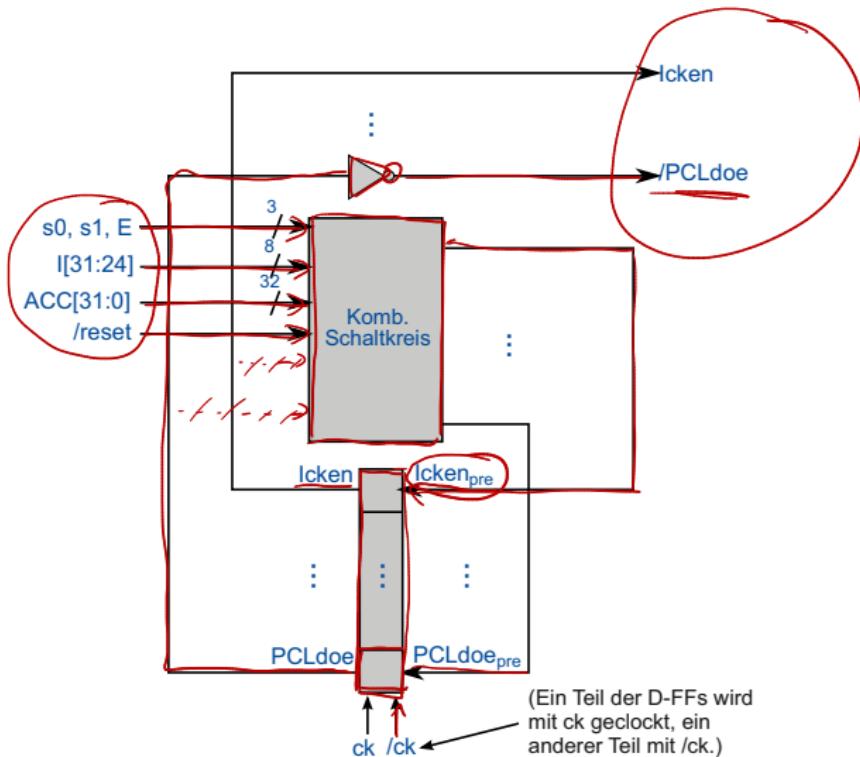
- Nutze Busse möglichst lange (unter Vermeidung von Bus Contention).
- Clock-Enable-Signale der Register möglichst spät  
→ viel Zeit für Berechnung neuer Daten.
- Nach dem Entwurfsende muss das Timing der CPU mit den konkreten Werten der eingesetzten Fertigungstechnologie überprüft werden. („Wie schnell kann man maximal takten, um korrektes Funktionieren zu garantieren?“)
  - Siehe nächstes Kapitel.

# Aufbau der Kontrolllogik (1/2)

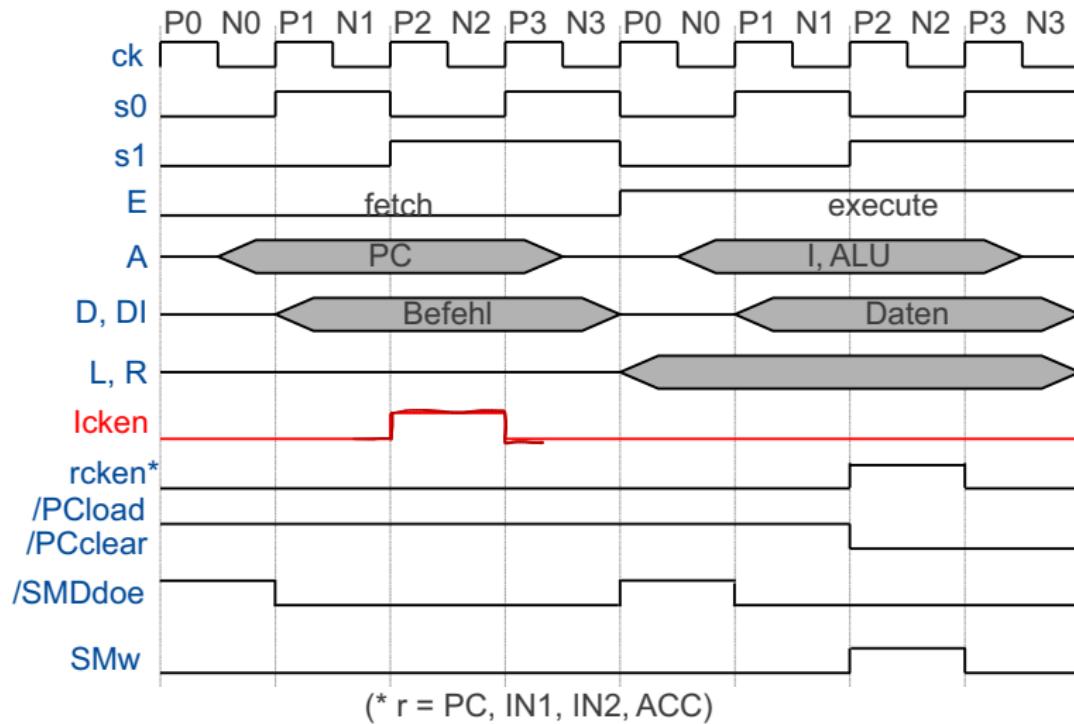
---

- Die eigentliche Kontrolllogik wird realisiert durch einen Endlichen Automaten.
- Die Kontrollsgrade sind als Ausgangssignale des Endlichen Automaten implementiert.
- Ist ein Kontrollsgrad *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 Kontrollsgrad *active high*, dann bezeichnen wir es z.B. mit  $x$ . Das Ausgangssignal  $x$  entspricht dem Ausgangssignal eines FFs mit Eingangssignal  $x_{pre}$ .

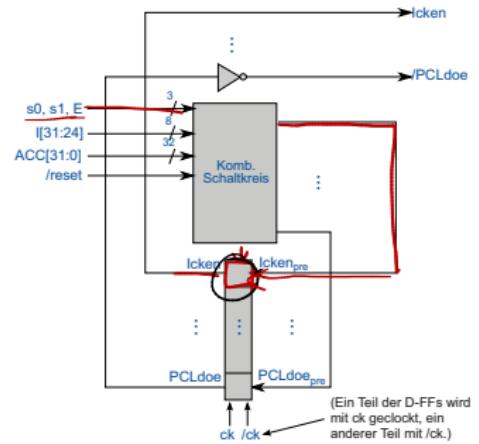
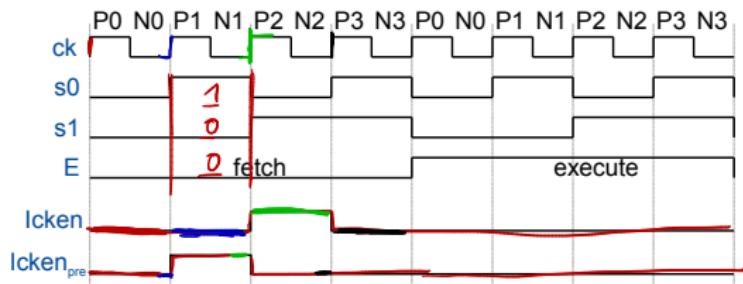
# Aufbau der Kontrolllogik (2/2)



# Berechnung von *Icken* als erstes Beispiel (1/2)



# Berechnung von *Icken* als erstes Beispiel (2/2)



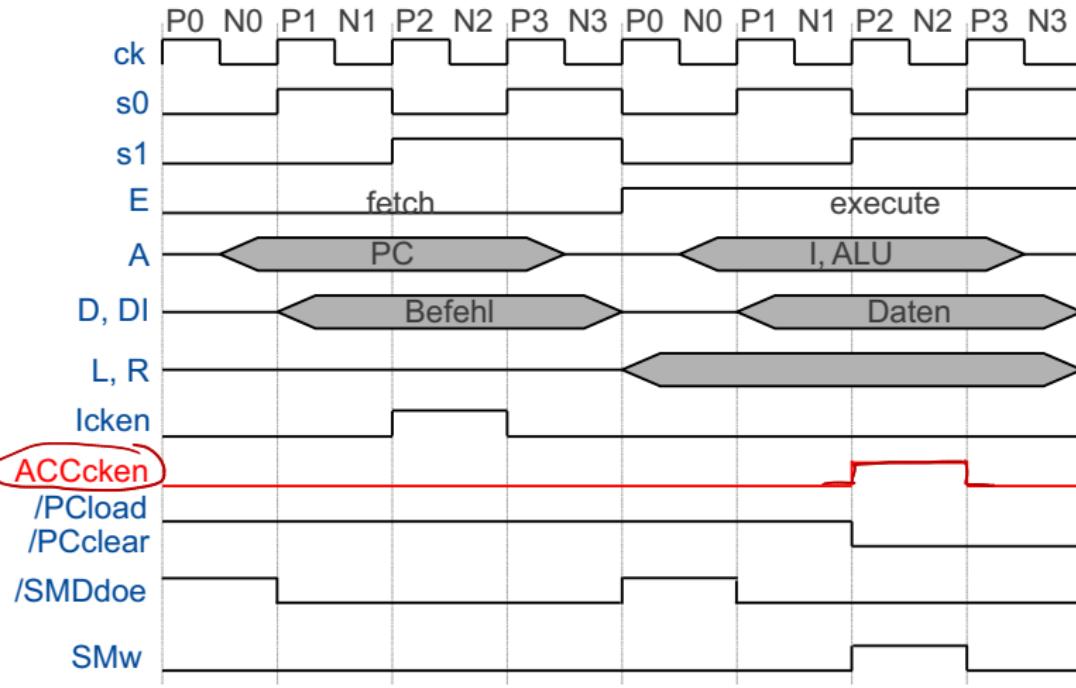
- $Icken_{pre}$  hat steigende Flanke bei *P1*, fallende bei *P2* von Fetch.
- Realisierung:  $Icken_{pre} = \overline{E} \cdot \overline{s_1} \cdot s_0$ .

# 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:  $I_{31} \cdot I_{30}$
- Load:  $I_{31} \cdot I_{30}$
- Move:  $I_{31} \cdot I_{30} \cdot I_{29} \cdot I_{28}$
- $D = ACC$ :  $I_{25} \cdot I_{24}$

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

## Berechnung von *ACCcken* als Beispiel (3/3)

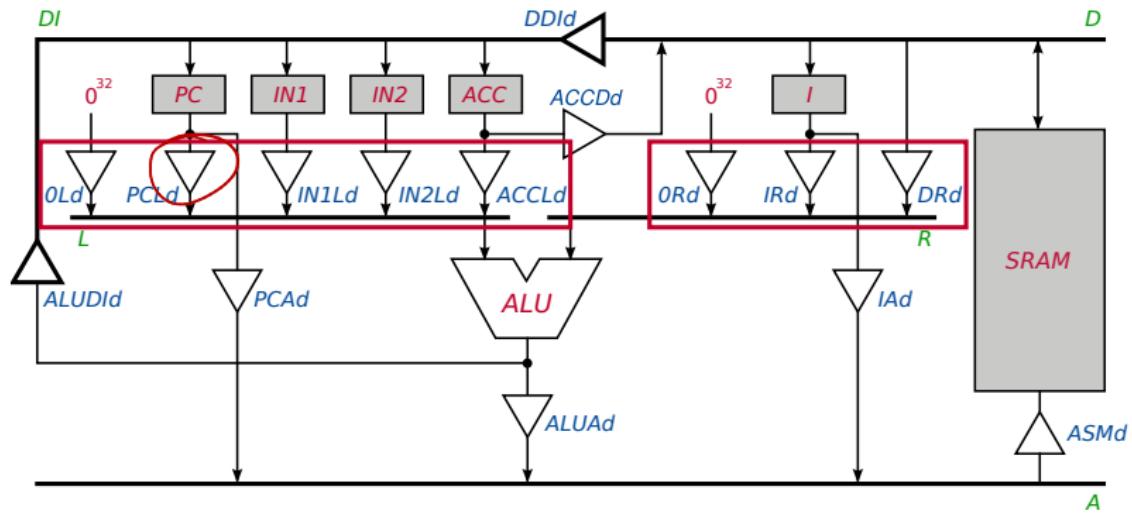
---

$$ACCcken_{pre} = \frac{E \cdot \overline{s_1} \cdot s_0 \cdot}{\cancel{I_{25}} \cdot \cancel{I_{24}} \cdot} \left( \cancel{\overline{I_{31} \cdot I_{30}}} + \cancel{I_{31} \cdot I_{30}} + \cancel{I_{31} \cdot \overline{I_{30}}} \cdot \cancel{I_{29}} \cdot \cancel{I_{28}} \right) \quad // P1 \text{ von execute}$$

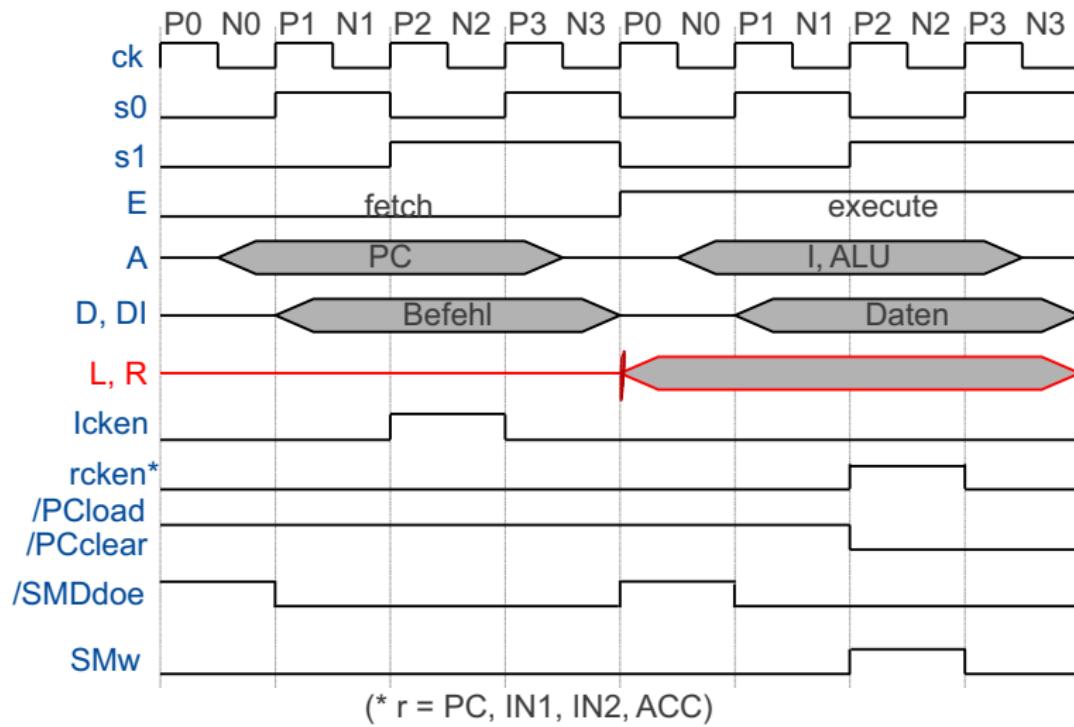
//  $D = ACC$

// Compute, Load oder Move

# Datenpfade und Treiber auf $L$ und $R$



# Idealisiertes Timingdiagramm

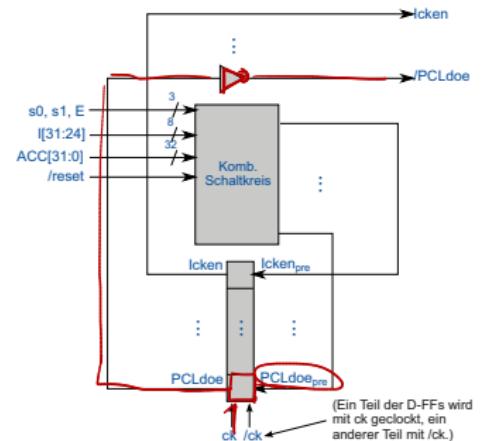
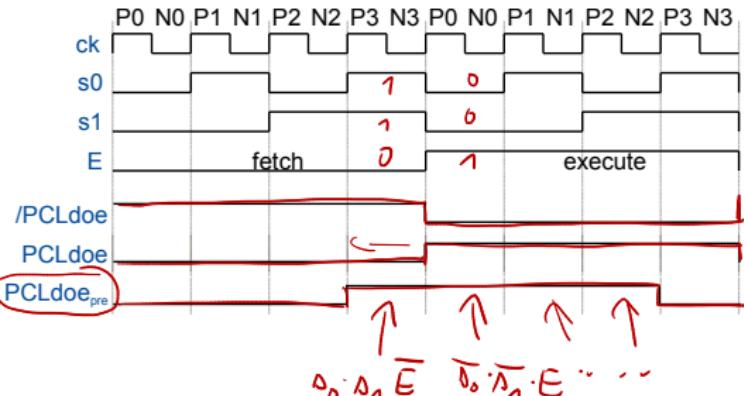


# Treiber auf Bussen $L$ und $R$

---

- $0Ld, PCLd, IN1Ld, IN2Ld, ACCLd, 0Rd, IRd, DRd.$
- 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.

## Berechnung von $PCLdoe$ (2/2)

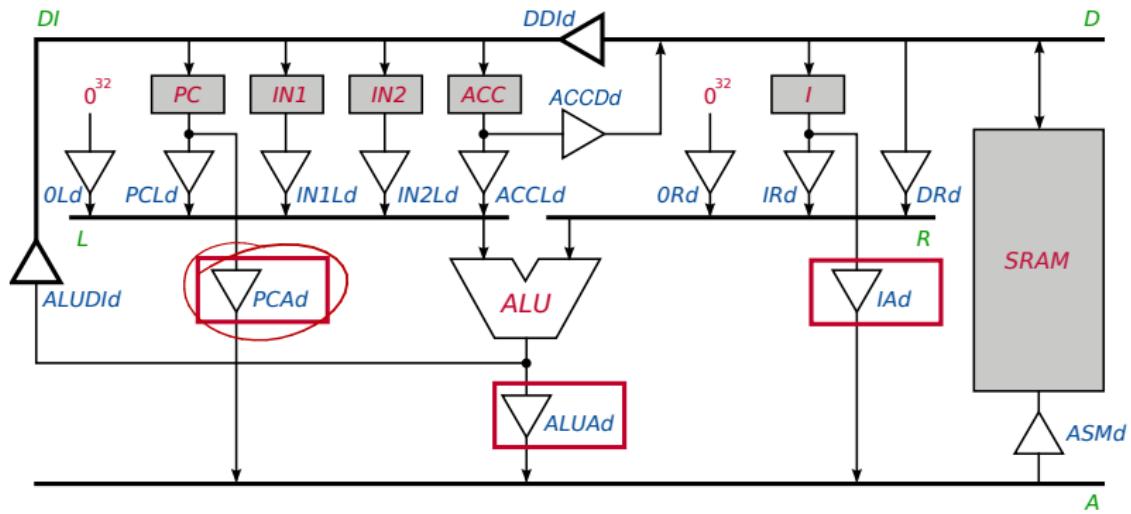
---

$$PCLdoe_{pre} = \left[ \begin{array}{l} \underline{\overline{E} \cdot s_1 \cdot s_0} \\ + \underline{\overline{E} \cdot \overline{s_1} \cdot \overline{s_0}} \\ + \underline{\overline{E} \cdot \overline{s_1} \cdot s_0} \\ + \underline{E \cdot s_1 \cdot \overline{s_0}} \end{array} \right] \cdot \left[ \begin{array}{l} \underline{\overline{I_{31} \cdot I_{30}}} \\ + \underline{\overline{I_{31} \cdot I_{30}} \cdot \overline{I_{25}} \cdot \overline{I_{24}}} \\ + \underline{\overline{I_{31} \cdot I_{30}} \cdot I_{29} \cdot I_{28} \cdot \overline{I_{27}} \cdot \overline{I_{26}}} \end{array} \right]$$

// P3 von fetch  
// Halten in Takt 0 von execute  
// Halten in Takt 1 von execute  
// Halten in Takt 2 von execute  
// JUMP  
// Compute mit  $D = PC$   
// MOVE mit  $S = PC$

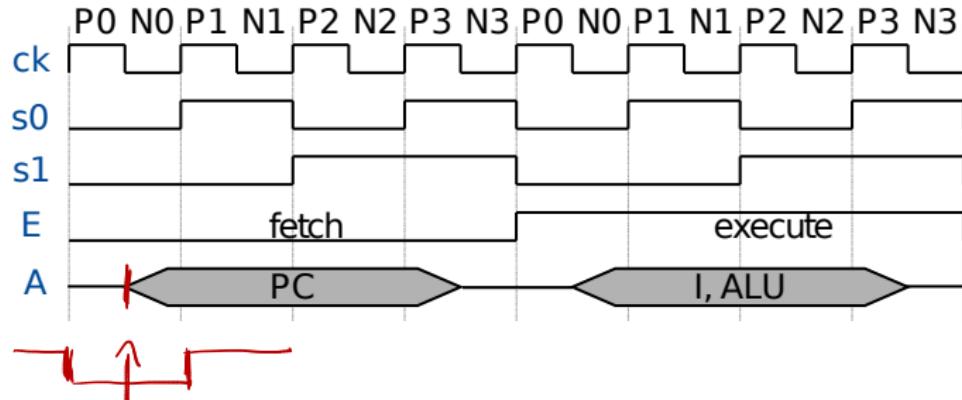
- Output-Enable-Signale für andere Treiber auf  $L$  und  $R$  analog.

# Datenpfade und Treiber auf Adressbus

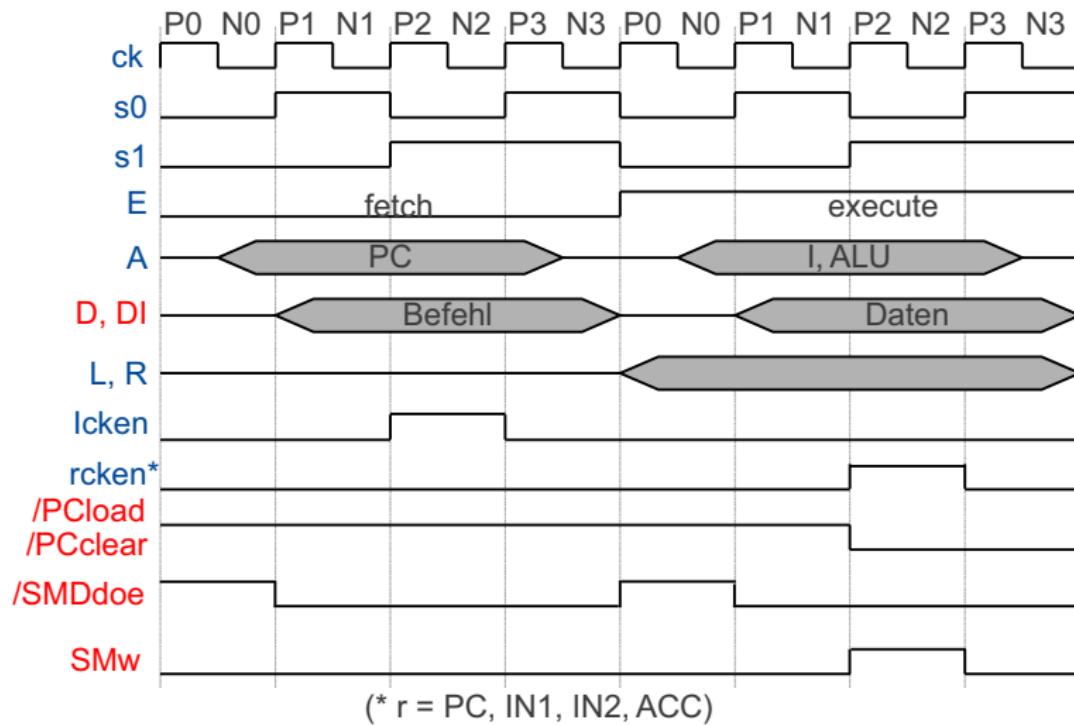


# Treiber auf Adressbus

- *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

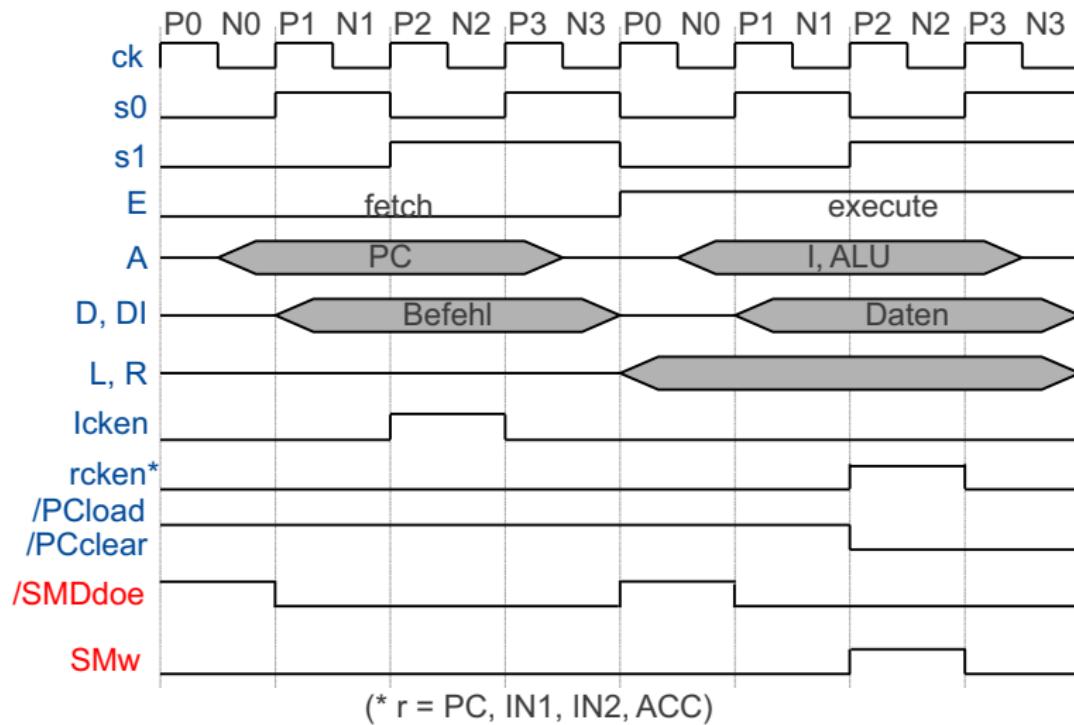


# Weitere Kontrollsignale

---

- 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

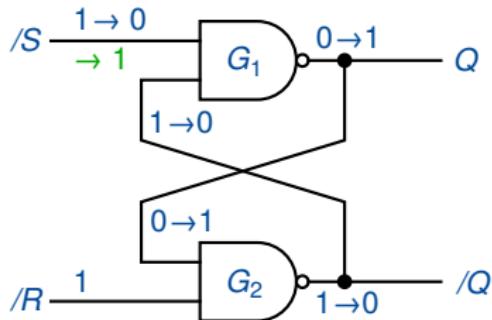
Albert-Ludwigs-Universität Freiburg

Prof. Dr. Christoph Scholl

Institut für Informatik  
Sommersemester 2023

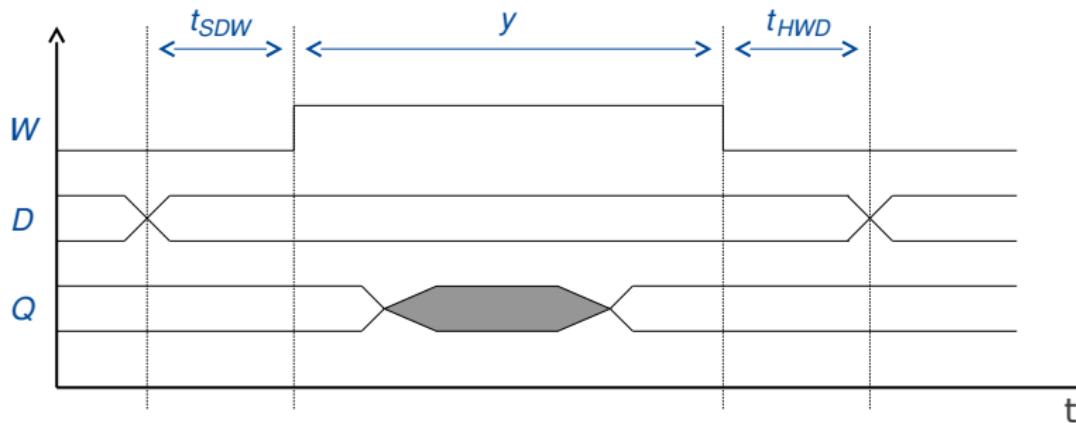
# Wiederholung: Übergang beim RS-Flipflop

- Zustand  $Q = 0 \rightarrow$  Zustand  $Q = 1$ :



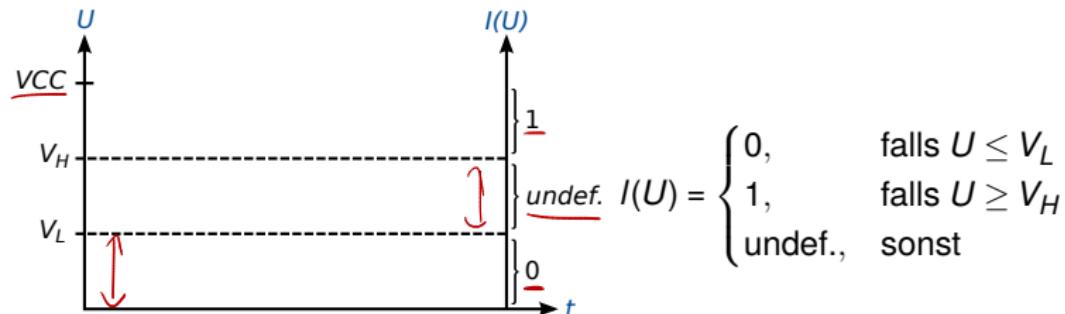
- Senke  $/S$  zur Zeit  $t_0$  ab und hebe zu  $t_0 + x$  wieder an (einen solchen Signalverlauf nennt man **Puls**).
- Nach Zeit  $t_{P/SQ}$  ist  $Q = 1$ . Nach Zeit  $t_{P/S/Q}$  ist  $/Q = 0$ .
- „Gatter brauchen Zeit zum Schalten!“ Aber wie lange ist  $t_{P/SQ}$ ,  $t_{P/S/Q}$ ? Oder wie lange muss ein Puls mindestens dauern? (= Pulsw W eite).

# 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  $y$ ?

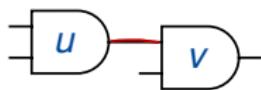
# Physikalische Signale $\leftrightarrow$ Logische Signale



- In jeder Technologie gibt es eine Versorgungsspannung  $V_{CC}$  (z.B. 1.1 V bei NanGate).
- Eine Spannung  $U \in [0, V_{CC}]$  wird als logischer Wert  $I(U)$  interpretiert.
  - Am Eingang (Input) eines Gatters:  $V_{IL}, V_{IH}$ .  
“Wie interpretiert ein Gatter anliegende Spannungswerte an seinen Inputs?”
  - Am Ausgang (Output) eines Gatters:  $V_{OL}, V_{OH}$ .  
“Mit welchem Spannungswert signalisiert ein Gatter eine logische 0/1 am Ausgang?”
- $V_{IL}, V_{IH}, V_{OL}, V_{OH}$  eines Bausteins sind gegeben.

# Zusammenschalten von Gattern

---

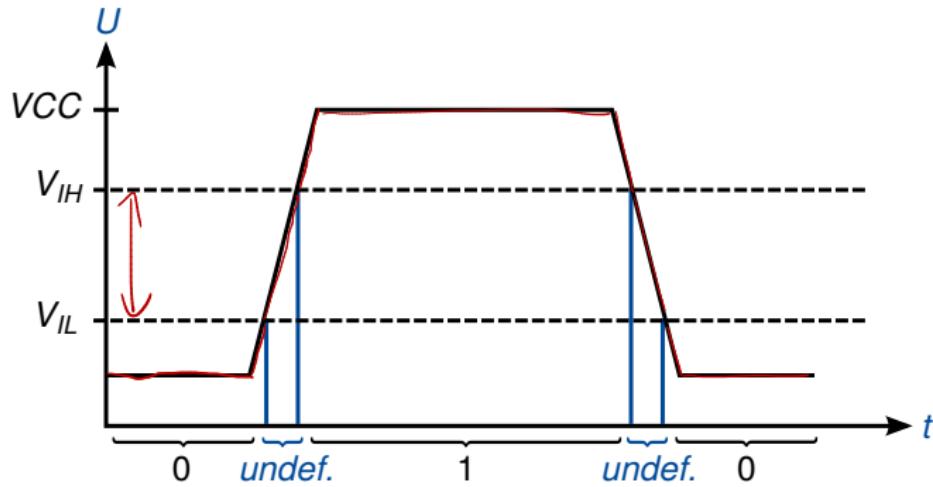


- Will man den Ausgang eines Gatters *u* mit dem Eingang eines Gatters *v* verbinden, dann sollte gelten:
  - $V_{OL}(u) \leq V_{IL}(v)$  und
  - $V_{OH}(u) \geq V_{IH}(v)$ .
- Sonst werden Signale falsch interpretiert.

# Beispiel: NanGate

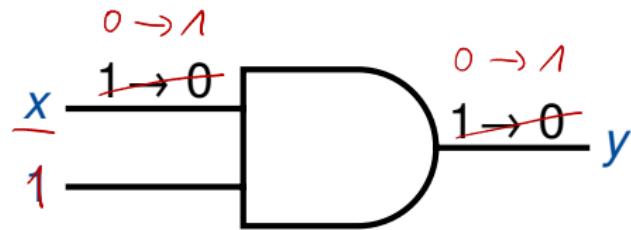
$$\underline{V_{IL}} = 30\% \cdot VCC = 0.33 \text{ V}$$
$$\underline{V_{IH}} = 70\% \cdot VCC = 0.77 \text{ V}$$

Entsprechend Output-Pegel  
 $V_{OL}, V_{OH}$ .

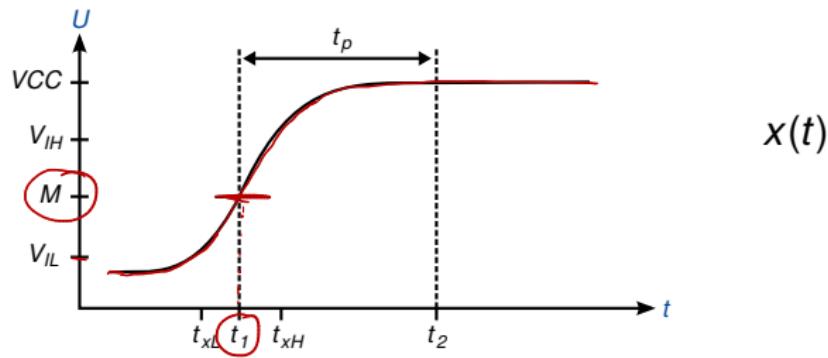


# Verzögerung

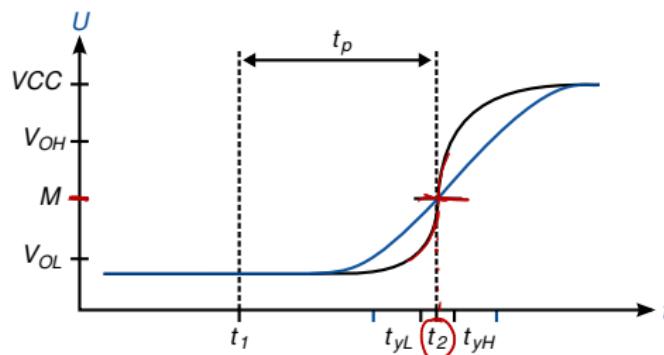
---



# Beispiel-Spannungsverlauf $x(t)$ , $y(t)$



Übergang:  $t_2 - t_1$



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

---

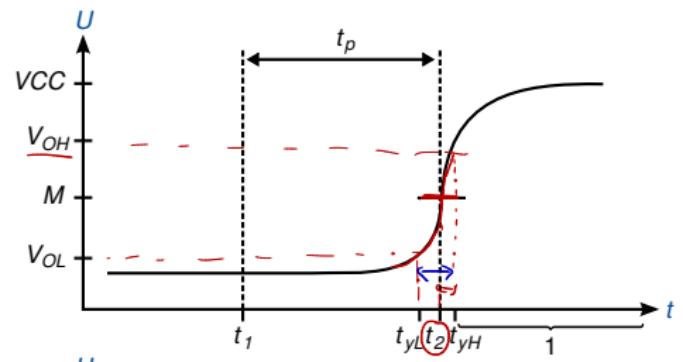
- In der Regel gibt es **verschiedene** Verzögerungszeiten für Übergänge am **Ausgang**:
  - $t_{PLH}$ : Verzögerungszeit bei  $0 \rightarrow 1$ .
  - $t_{PHL}$ : Verzögerungszeit bei  $1 \rightarrow 0$ .

# Modellierung der Verzögerungszeit

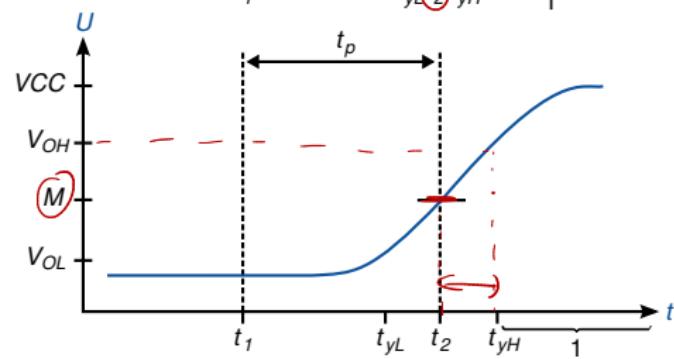
---

- **Problem** bei der Modellierung der Verzögerungszeit bezüglich fester Spannung  $M$ :
  - Keine Aussage darüber, wann logische Signale  $\underline{0}$  oder  $\underline{1}$  sind, d. h. physikalische Signale unterhalb  $V_{OL}$  oder oberhalb  $V_{OH}$  sind.

# Illustration des Problems



→ Ähnliches Problem am Gattereingang.



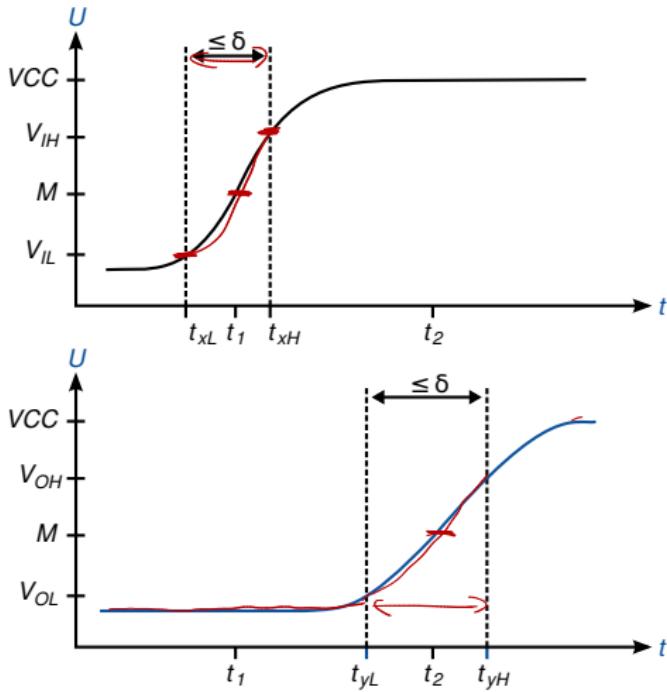
# Anstiegs- und Abfallzeiten

---

- 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

- 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.



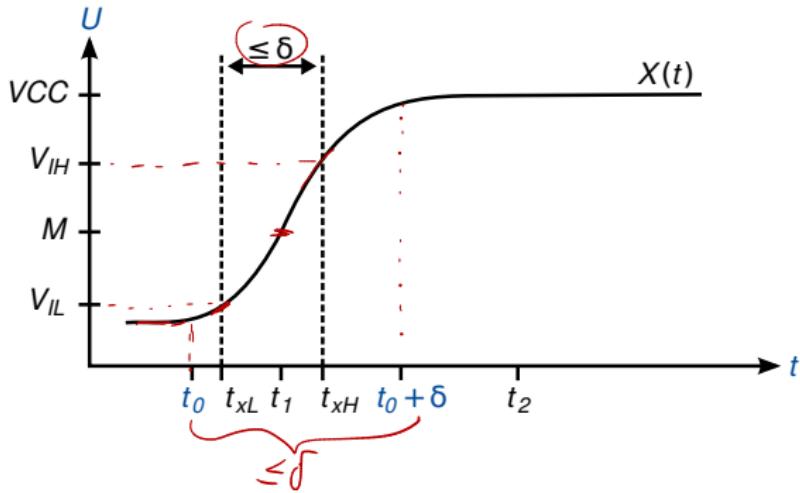
## Beispiel: NanGate

---

- $V_{IL} = 30\% \cdot VCC = 0.33 \text{ V}$   
 $V_{IH} = 70\% \cdot VCC = 0.77 \text{ V}$
- NanGate für  $M = 0.55 \text{ V}$  spezifiziert.  
Bausteine *NAND, NOT, AND, OR, EXOR*.
- $t_p$  zwischen 0.00 ns und 0.21 ns.
- $\delta = 0.13 \text{ ns}$  ( $1 \text{ ns} = 10^{-9} \text{ s}$ )
- Die Zeiten, an denen die entsprechenden Signale  
wohldefinierte logische Werte 0, 1 annehmen,  
unterscheiden sich von denen für  $M$  um höchstens  $\delta$ .

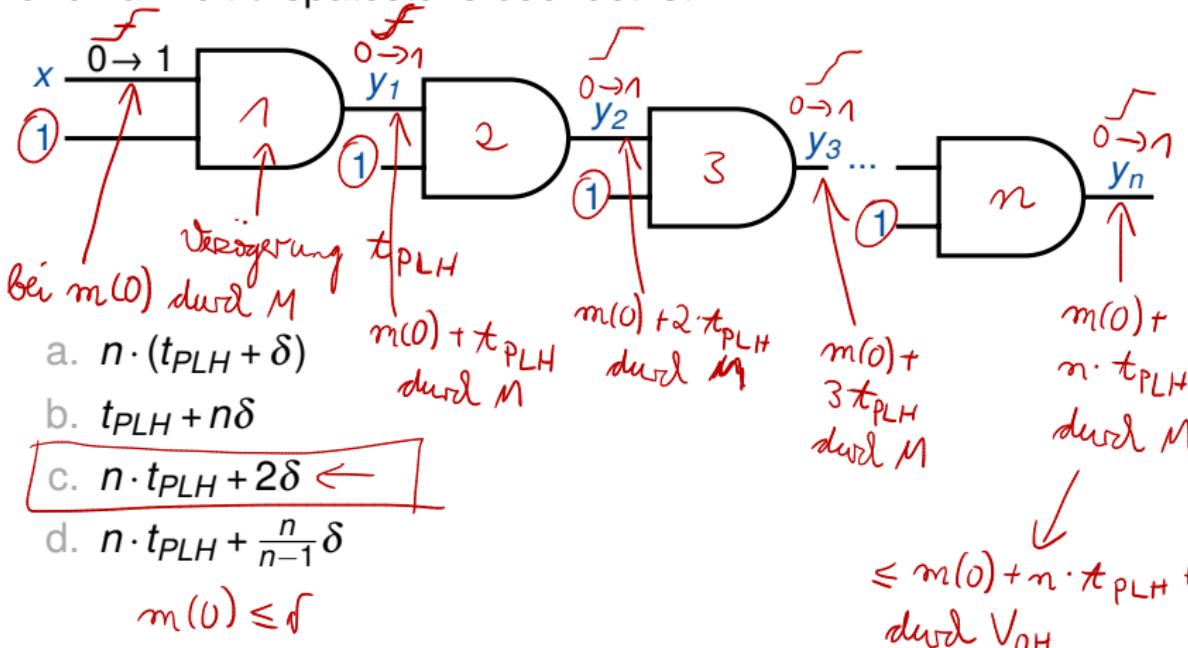
# Bemerkung

- 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.

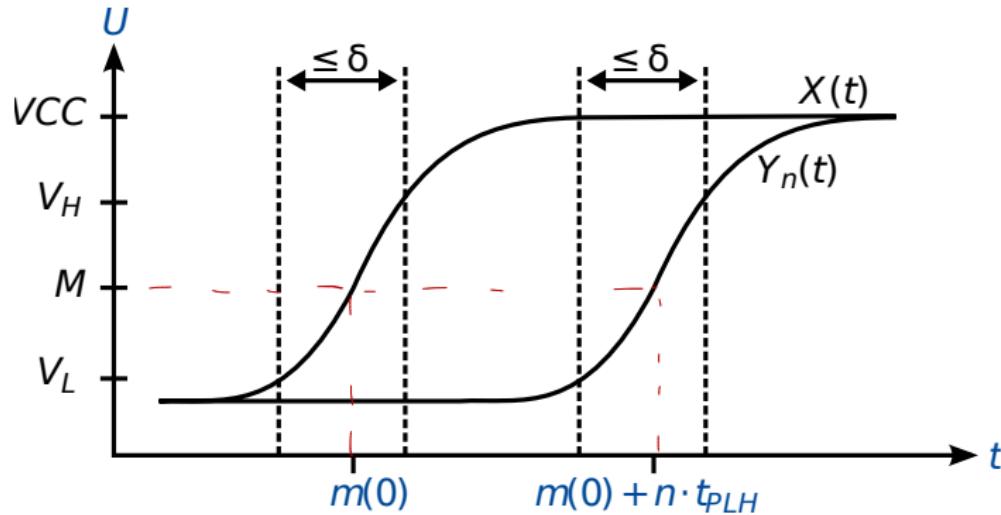


# SMILE – Verzögerungszeit

Nach welcher Zeit ist  $Y_n$  spätestens logisch 1, wenn der Schaltvorgang von 0 nach 1 bei  $x$  zur Zeit 0 angefangen hat und zur Zeit  $\delta$  spätestens beendet ist?



## 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.

# Vereinbarungen

---

- Im Folgenden soll  
Signal  $X$  wird zum Zeitpunkt  $t_1$  abgesenkt/angehoben  
bedeuten  
 $X$  wird abgesenkt/angehoben mit  $X(t_1) = M$ .
- Desweiteren sind alle Zeitangaben in *ns*.
- Wir nehmen außerdem in Zukunft immer an: *rise / fall times*  $\leq \delta$ .

# 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:
  - $t^{min}$  = 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  $\text{IR}$  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  $a$ )

# Exkurs: Rechnen mit Intervallarithmetik (2/2)

## Definition

Gegeben ein Operator  $\text{\textcircled{op}} \in \{+, -, \cdot\}$  in  $\mathbb{R}$ . Der dazugehörige Operator  $\text{\textcircled{op}}$  auf IR ist definiert als:

Für  $a, b, c, d \in \mathbb{R}$ :

$$[a, b] \text{\textcircled{op}} [c, d] := \{x \text{\textcircled{op}} y \mid x \in [a, b], y \in [c, d]\}$$

Beispiele:

- $[a, b] \oplus [c, d] = [a + c, b + d]$
- $[a, b] \ominus [c, d] = [a - d, b - c]$
- $[a, b] \odot [c, d] = [\min(a \cdot c, a \cdot d, b \cdot c, b \cdot d), \max(a \cdot c, a \cdot d, b \cdot c, b \cdot d)]$

Wieso nicht  $[a \cdot c, b \cdot d]$ ?

$$[-1, 5] \odot [-2, 8] = [-10, 40]$$

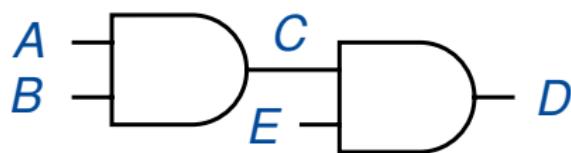
# Bemerkungen

---

- 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  $\underline{\tau} = [t^{\min}, t^{\max}]$ .

# Beispiel: AND-Gatter

---



AND

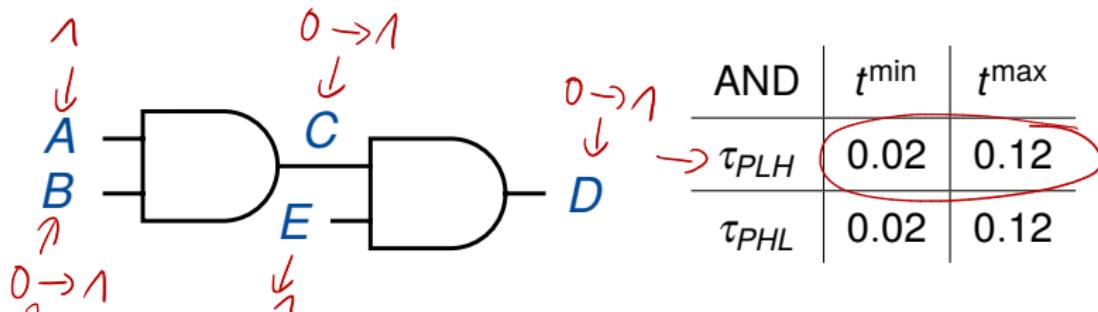
$$\tau_{PLH} = [0.02, 0.12]$$

$$\tau_{PHL} = [0.02, 0.12]$$

Bzw.:

AND	$t^{\min}$	$t^{\max}$
$\rightarrow \tau_{PLH}$	0.02	0.12
$\rightarrow \tau_{PHL}$	0.02	0.12

# Fall 1



- $A, E$  fest auf 1.
- $B$  von 0 auf 1 zum Zeitpunkt  $t_0$  (genauer: " $B$  geht von 0 auf 1 und durchläuft  $M$  zum Zeitpunkt  $t_0$ ").

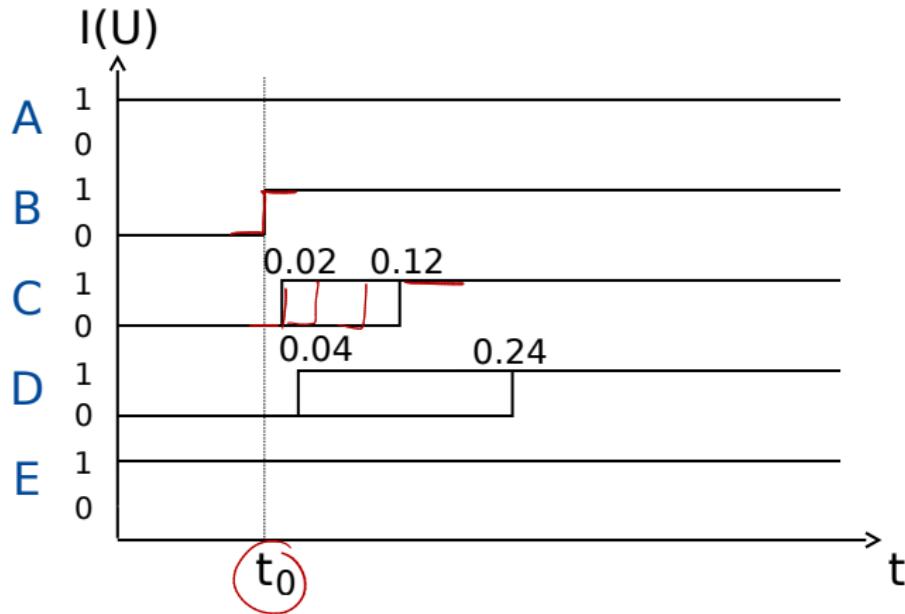
→ Änderung von  $C$  zur Zeit

$$\begin{aligned}\tau_1 &= t_0 + \tau_{PLH} (\text{AND}) \\ &= t_0 + [0.02, 0.12] = \underline{[t_0 + 0.02, t_0 + 0.12]}\end{aligned}$$

→ Änderung von  $D$  zur Zeit

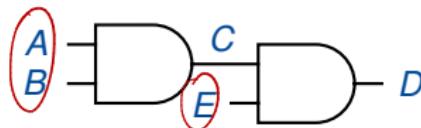
$$\begin{aligned}\tau_2 &= \tau_1 + \tau_{PLH} (\text{AND}) \\ &= t_0 + 2 \cdot \tau_{PLH} (\text{AND}) \\ &= t_0 + 2 \cdot [0.02, 0.12] \\ &= \underline{t_0 + [0.04, 0.24]}$$

# Fall 1 - Timing-Diagramm



## Fall 2

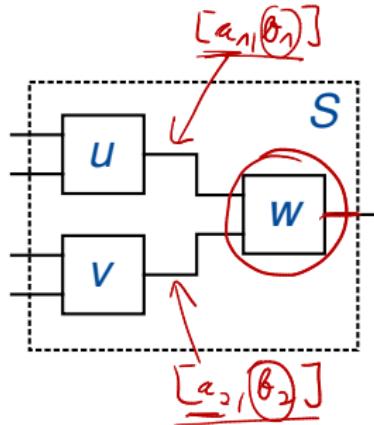
---



- $A, B, E$  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

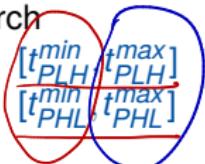
# Gröbere Abschätzung

- Bestimmung von Zeitintervallen, zu denen Gatter überhaupt schalten können:



Annahmen:

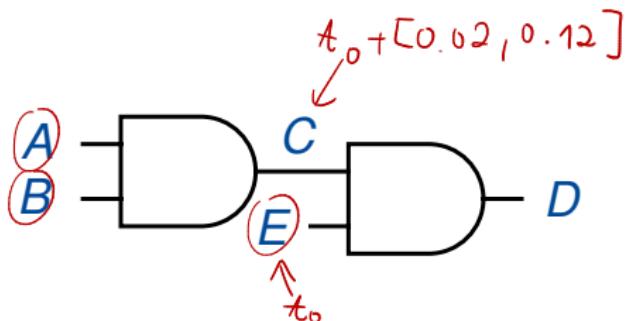
- $u$  schaltet im Intervall  $[a_1, b_1]$ .
- $v$  schaltet im Intervall  $[a_2, b_2]$ .
- Die Verzögerungszeiten von  $w$  sind gegeben durch

$$\begin{aligned}\tau_{PLH} &= [t_{PLH}^{\min}, t_{PLH}^{\max}] \\ \tau_{PHL} &= [t_{PHL}^{\min}, t_{PHL}^{\max}]\end{aligned}$$


- Dann gilt mit  $t_p^{\min} := \min(t_{PLH}^{\min}, t_{PHL}^{\min})$  und  $t_p^{\max} := \max(t_{PLH}^{\max}, t_{PHL}^{\max})$  w kann schalten im Intervall  $\underline{[\min(a_1, a_2), \max(b_1, b_2)] + [t_p^{\min}, t_p^{\max}]} =$

$$[\min(a_1, a_2) + t_p^{\min}, \max(b_1, b_2) + t_p^{\max}]$$

## Anwendung auf Beispiel, Fall 2



	AND	$t^{\min}$	$t^{\max}$
$\tau_{PLH}$	0.02	0.02	0.12
$\tau_{PHL}$	0.02	0.12	0.12

■ Wenn die Gatter schalten, dann in folgenden Intervallen:

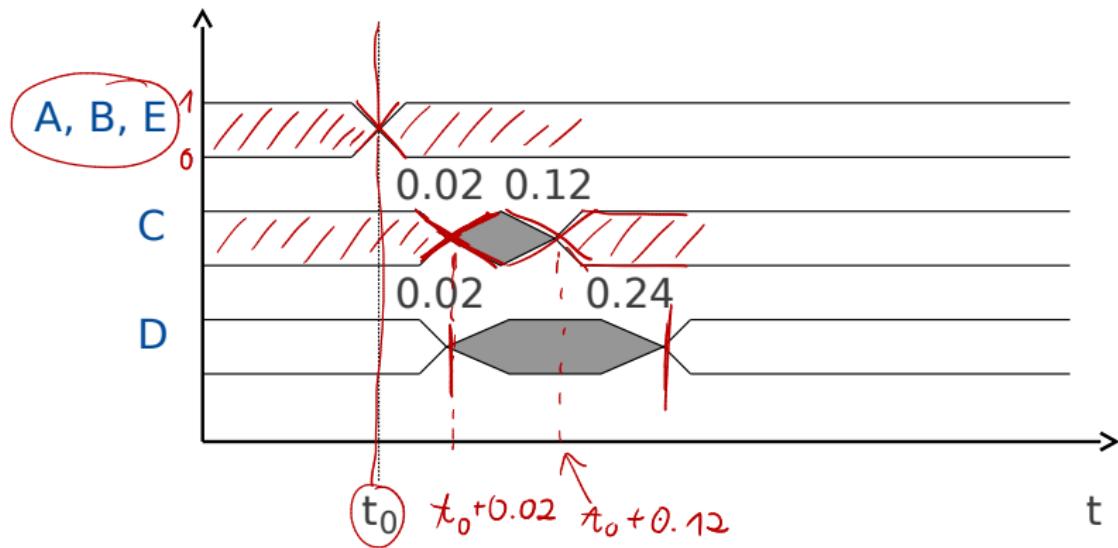
■ A, B, E:  $t_0 + [0.0, 0.0]$

■ C:  $t_0 + [0.02, 0.12] = t_0 + [\min(0.02, 0.02), \max(0.12, 0.12)]$

■ D:  $t_0 + [0.0, 0.12] + [0.02, 0.12] = t_0 + [0.02, 0.24]$

Intervall, in dem sich irgend ein Eingang des AND-Gatters ändern kann

# Fall 2 - Timing-Diagramm



# Interpretation des Timing-Diagramms

---

- Was kann im grauen Bereich passieren?

- **Beispiel:**

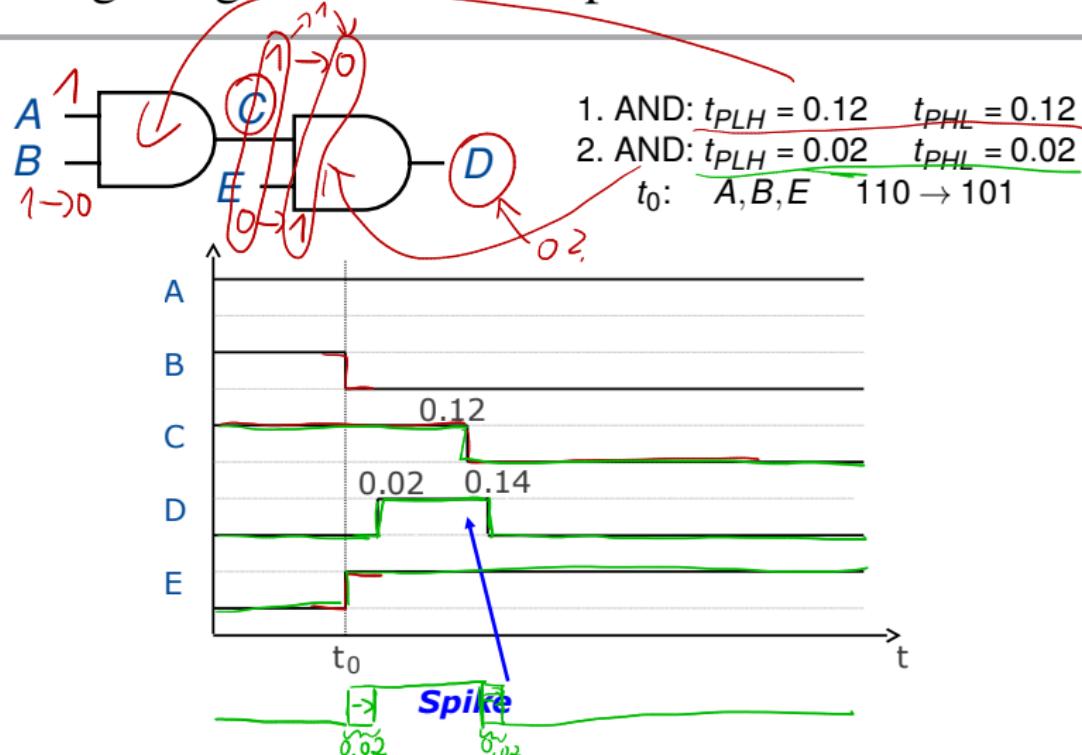
$t_0$ :  $A, B, E$     110  $\rightarrow$  101

- **Annahme:**

AND-Gatter haben folgende (feste) Verzögerungszeiten.

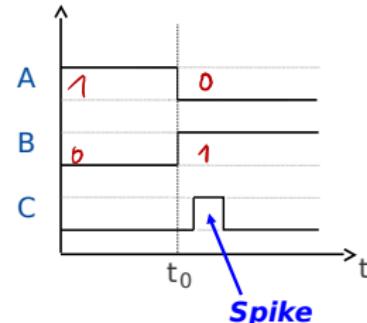
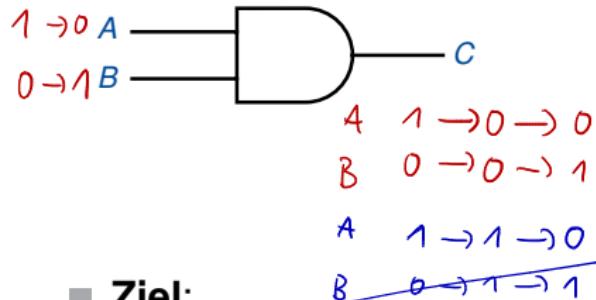
- 1. AND-Gatter:  $t_{PLH} = 0.12$ ,     $t_{PHL} = 0.12$   
2. AND-Gatter:  $t_{PLH} = 0.02$ ,     $t_{PHL} = 0.02$

# Timing-Diagramm zum Beispiel



- In manchen Anwendungen will man Spikes verhindern (siehe z.B. FlipFlops).

# Spikefreies Umschalten von Gattern



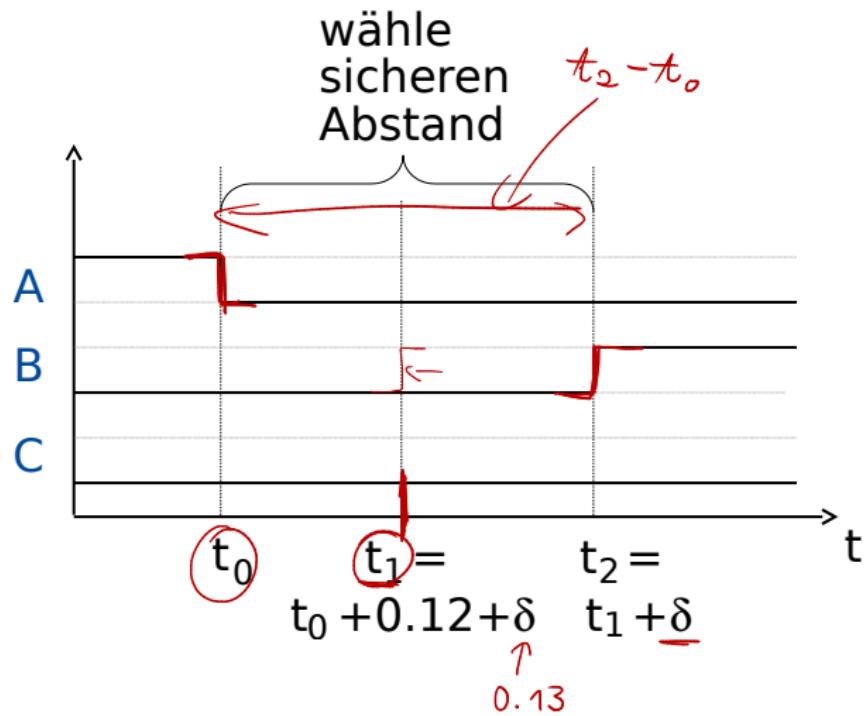
## Ziel:

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  $\tau_{PHL}^{max} = 0.12 \text{ ns}$  den Wert  $M$  durchlaufen und wäre 0 spätestens nach  $0.12 + \delta \text{ ns}$ .
- Interner Umschaltvorgang „ $C = 0$  wegen  $A = 0$ “ muss also spätestens nach  $0.12 + \delta \text{ ns}$  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  $t_1$ :  $B = 0 \Rightarrow C = 0$

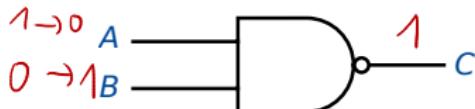
Nach  $t_1$ :  $A = 0 \Rightarrow C = 0$

→ Übergänge für  $A$  und  $B$  mit Abstand  
 $t_2 - t_0 = 0.12 + 2\delta = 0.38$  ( $\delta = 0.13$ ).

AND	$t^{\min}$	$t^{\max}$
$\tau_{PLH}$	0.02	0.12
$\tau_{PHL}$	0.02	0.12

# Analog: Spikefreies Umschalten bei NAND

## ■ Beispiel: NAND



NAND	$t^{\min}$	$t^{\max}$
$\tau_{PLH}$	0.01	0.15
$\tau_{PHL}$	0.01	0.12

- Kritischer Übergang: Zuerst  $A : 1 \rightarrow 0$ , dann  $B : 0 \rightarrow 1$ .
- Daraus ergibt sich der Abstand  $\underline{t_{PLH}^{\max} + 2\delta = 0.41}$

$$= 0.15 + 2 \cdot 0.13$$



# Kapitel 5 – Timing

1. Physikalische Eigenschaften
2. **Timing wichtiger Komponenten**
3. Exaktes Timing von ReTI

Albert-Ludwigs-Universität Freiburg

Prof. Dr. Christoph Scholl

Institut für Informatik  
Sommersemester 2023

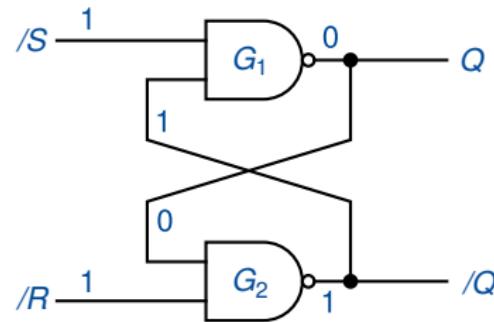
# Timing – Übersicht

---

- 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

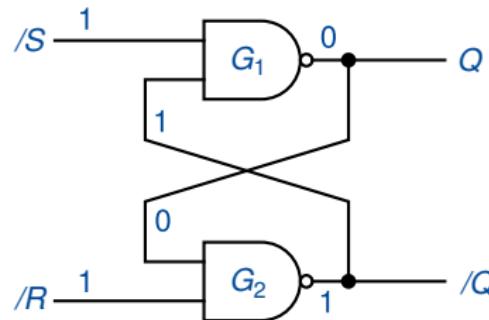
# RS-Flipflop

- Zustand  $Q = 0 \rightarrow$  Zustand  $Q = 1$ :



# RS-Flipflop

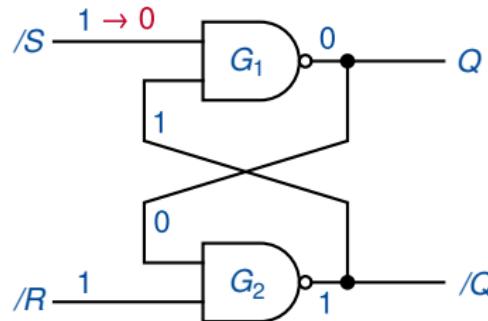
- Zustand  $Q = 0 \rightarrow$  Zustand  $Q = 1$ :



- Senke  $/S$  zur Zeit  $t_0$  ab und hebe zu  $t_0 + x$  wieder an (einen solchen Signalverlauf nennt man Puls).

# RS-Flipflop

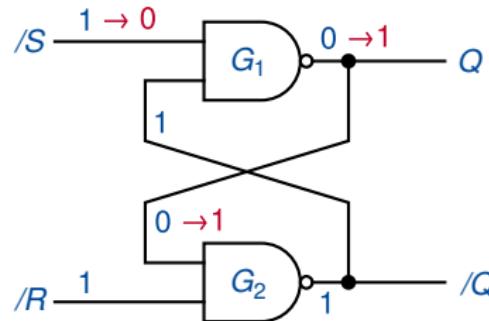
- Zustand  $Q = 0 \rightarrow$  Zustand  $Q = 1$ :



- Senke  $/S$  zur Zeit  $t_0$  ab und hebe zu  $t_0 + x$  wieder an (einen solchen Signalverlauf nennt man Puls).

# RS-Flipflop

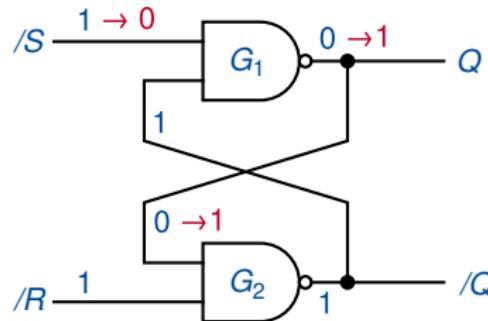
- Zustand  $Q = 0 \rightarrow$  Zustand  $Q = 1$ :



- Senke  $/S$  zur Zeit  $t_0$  ab und hebe zu  $t_0 + x$  wieder an (einen solchen Signalverlauf nennt man Puls).

# RS-Flipflop

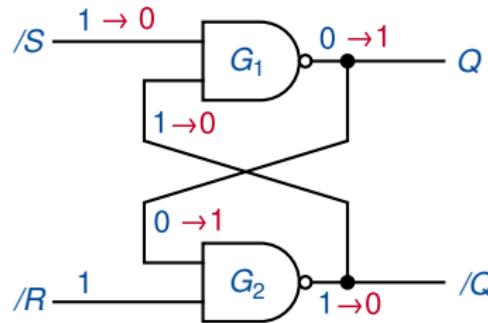
- Zustand  $Q = 0 \rightarrow$  Zustand  $Q = 1$ :



- 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

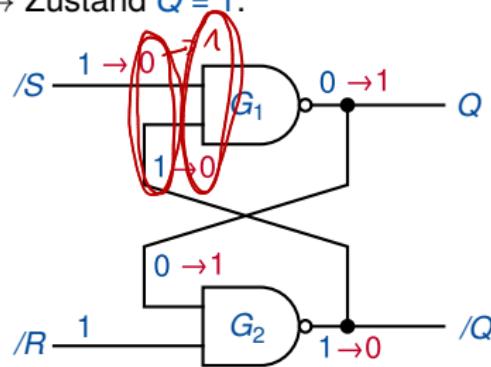
- Zustand  $Q = 0 \rightarrow$  Zustand  $Q = 1$ :



- 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

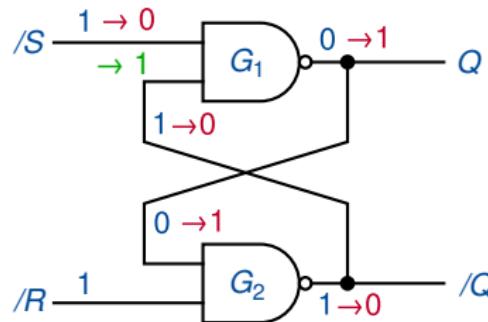
- Zustand  $Q = 0 \rightarrow$  Zustand  $Q = 1$ :



- 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

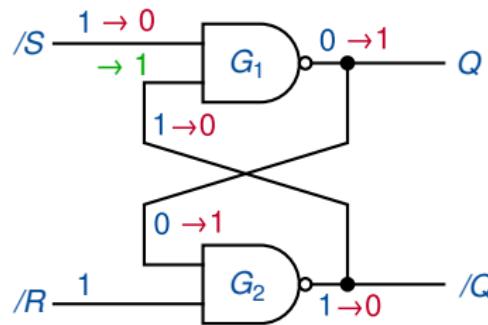
- Zustand  $Q = 0 \rightarrow$  Zustand  $Q = 1$ :



- 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  $Q = 1$ :

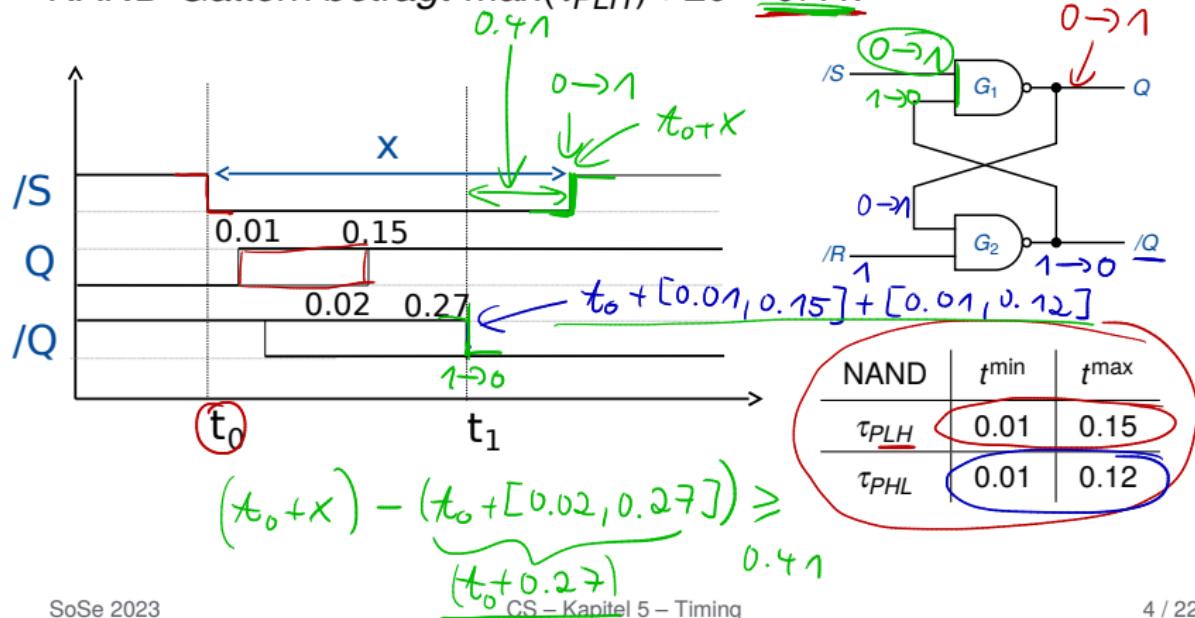


- Senke  $/S$  zur Zeit  $t_0$  ab und hebe zu  $t_0 + x$  wieder an (einen solchen Signalverlauf nennt man Puls).
- Nach Zeit  $t_{P/SQ}$  ist  $Q = 1$ .
- Nach Zeit  $t_{P/S/Q}$  ist  $/Q = 0$ .
- Wähle  $x$  so, dass kein Spike entsteht.

# Übergang – graphisch

Wie groß muss  $x$  gewählt werden, damit das RS-FlipFlop spikefrei schaltet?

**Zur Erinnerung:** Der Abstand zum spikefreien Umschalten von NAND-Gattern beträgt  $\max(\tau_{PLH}) + 2\delta = 0.41$ .



# Spikefreier Übergang

---

- Nach den Regeln des spikefreien Umschaltens von Gattern entsteht kein Spike, falls:

$$(t_0 + x) - \underline{(t_0 + 0.27)} \geq 0.41 \Leftrightarrow \underline{x} \geq 0.68ns$$

- Wechsel von Zustand  $Q = 1$  zu Zustand  $Q = 0$  aus Symmetriegründen analog.

# Symbole und Bezeichnungen

---

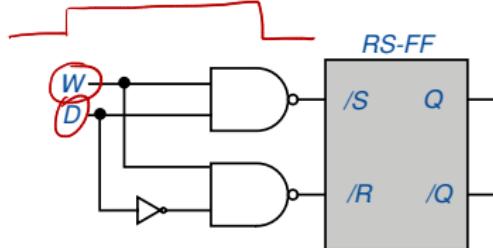
Symbol	Bezeichnung	$t^{\min}$	$t^{\max}$
$x$	Pulsweite	<u>0.68</u>	
$\tau_{P/SQ}$	Verzögerungszeit von /S bis Q	<u>0.01</u>	<u>0.15</u>
$\tau_{P/S/Q}$	Verzögerungszeit von /S bis /Q	<u>0.02</u>	<u>0.27</u>
$\tau_{P/RQ}$	Verzögerungszeit von /R bis Q	<u>0.02</u>	<u>0.27</u>
$\tau_{P/R/Q}$	Verzögerungszeit von /R bis /Q	<u>0.01</u>	<u>0.15</u>

# D-Latch

- $W$  ist *active high*.

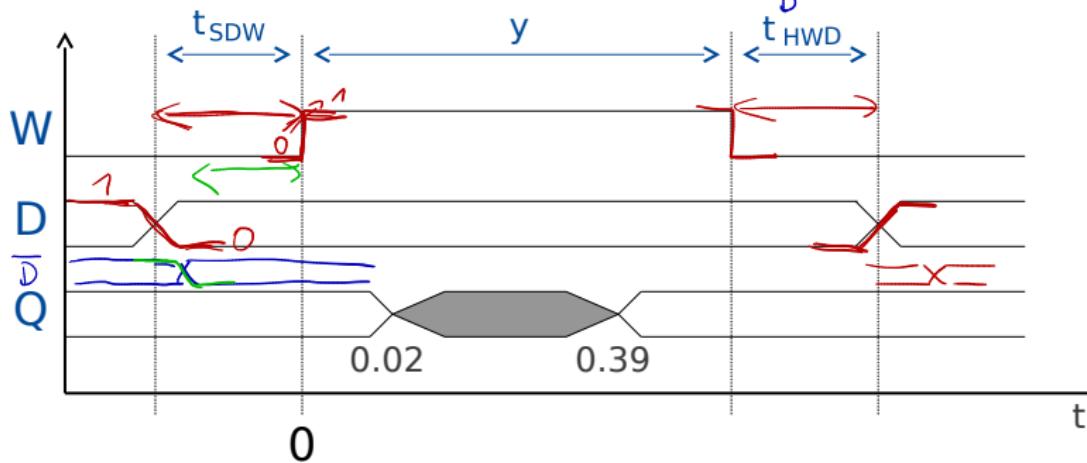
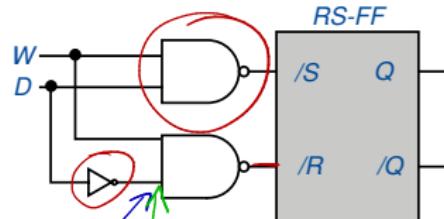
- $W = 0 \Rightarrow /S, /R$  inaktiv

- $W = 1 \Rightarrow \begin{cases} /S \text{ aktiv,} & \text{falls } D = 1 \\ /R \text{ aktiv,} & \text{falls } D = 0 \end{cases}$



- 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
  - beim Schreiben von 0 Spike auf  $/S$  zu verhindern,
  - beim Schreiben von 1 Spike auf  $/R$  zu verhindern (kritischer!).
- Nach  $W : 1 \rightarrow 0$  werden Daten für Zeit  $t_{HWD}$  stabil gehalten, um
  - beim Schreiben von 1 Spike auf  $/R$  zu verhindern,
  - beim Schreiben von 0 Spike auf  $/S$  zu verhindern (kritischer!)

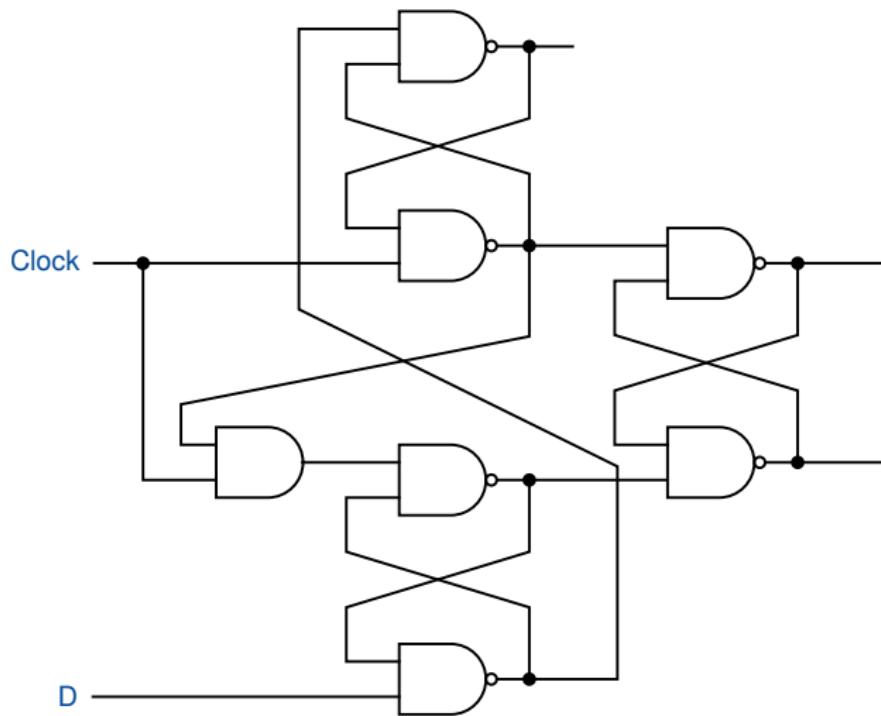
# Man rechnet nach:

---

Der Schreibvorgang beim D-Latch funktioniert mit den Parameterwerten aus der Tabelle ([Übung](#)).

Symbol	Bezeichnung	$t^{\min}$	$t^{\max}$
$y$	Pulsweite des Schreibimpulses	0.79	
$t_{SDW}$	Setupzeit von $D$ bis $W$	<u>0.49</u>	
$t_{HDW}$	Holdzeit von $W$ nach $D$	<u>0.41</u>	
$\tau_{PWQ}$	Verzögerungszeit von $W$ bis $Q$	0.02	0.39

# Mögliche Realisierung: D-Flipflop



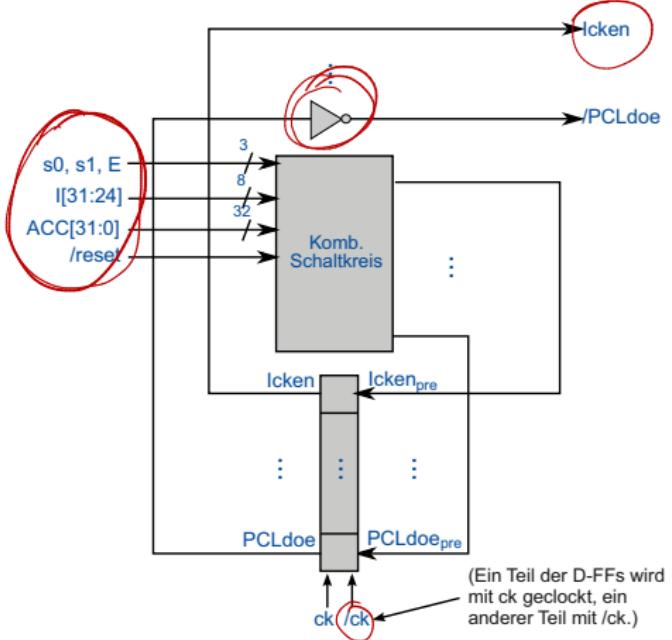
# Timing: D-Flipflop

---

- Vorgehen [analog](#) zu RS-Flipflop und D-Latch, aber wesentlich komplizierter.
- Wir verzichten daher auf die Analyse.
- Die [NanGate-Bibliothek](#) enthält bereits ein D-FF mit folgenden charakteristischen Zeiten (in ns):

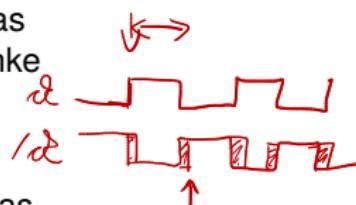
Symbol	Bezeichnung	$t^{\min}$	$t^{\max}$
$t_{SDC}$	Setupzeit von $D$ bis $ck$	<u>0.08</u>	
$t_{HCD}$	Holdzeit von $D$ nach $ck$	<u>0.14</u>	
$\tau_{PCQ}$	Verzögerungszeit von $ck$ bis $Q$	<u>0.12</u>	<u>0.26</u>

# 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  $x_{pre}$ .

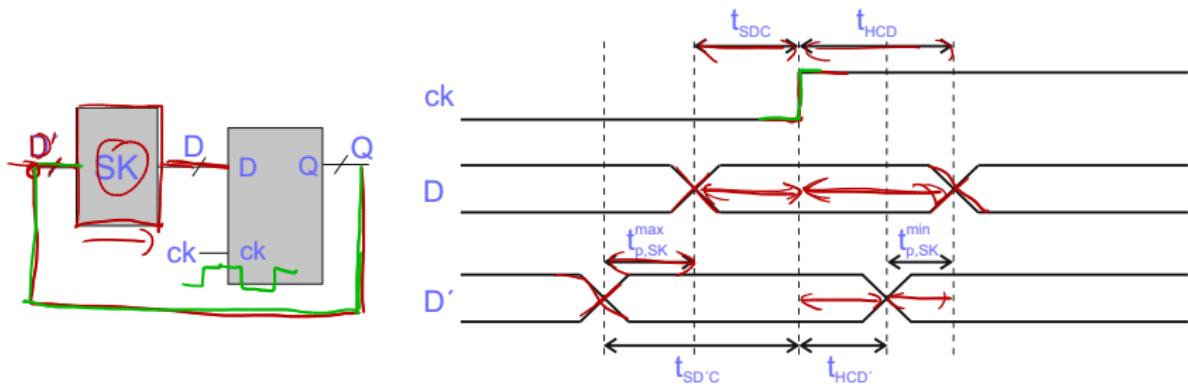
# Kontrolllogik

- 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  $\tau_{p,ah}^+ = [0.12, 0.26]$  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  $\tau_{p,al}^+ = [0.12, 0.26] + [0.00, 0.15]$  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	

# Grundsätzliche Überlegungen zur Timing-Analyse

- Ausgangssituation: D-FFs mit Setup-Zeit  $t_{SDC}$  und Hold-Zeit  $t_{HCD}$
- Die Eingangsdaten  $D$  eines Registers werden aus Daten  $D'$  durch einen kombinatorischen Schaltkreis  $SK$  berechnet:



- Erhöhe Setup-Zeit um maximale Verzögerung von  $SK$ :  $t_{SD'C} = \underline{t_{SDC}} + \underline{t_{p,SK}^{\max}}$
  - Verringere Hold-Zeit um minimale Verzögerung von  $SK$ :  $t_{HCD'} = \underline{t_{HCD}} - \underline{t_{p,SK}^{\min}}$
- Fall:  $Q = D'$ ; Verzögerungszeit D-FF +  $t_{p,SK}^{\max} + t_{SDC} \leq$  Taktperiode des FFA

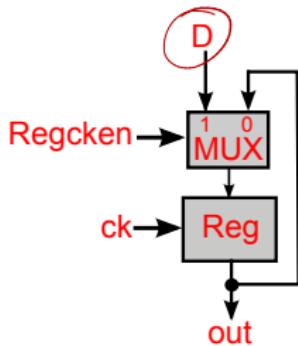
# Timing: Kontrolllogik

- 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 low)	$t_c/2 + 0.13$	$t_c/2 + 0.56$
$t_{SDC}^+$	Setupzeit von $D$ bis $ck$	<u>0.88</u>	
$t_{SDC}^-$	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

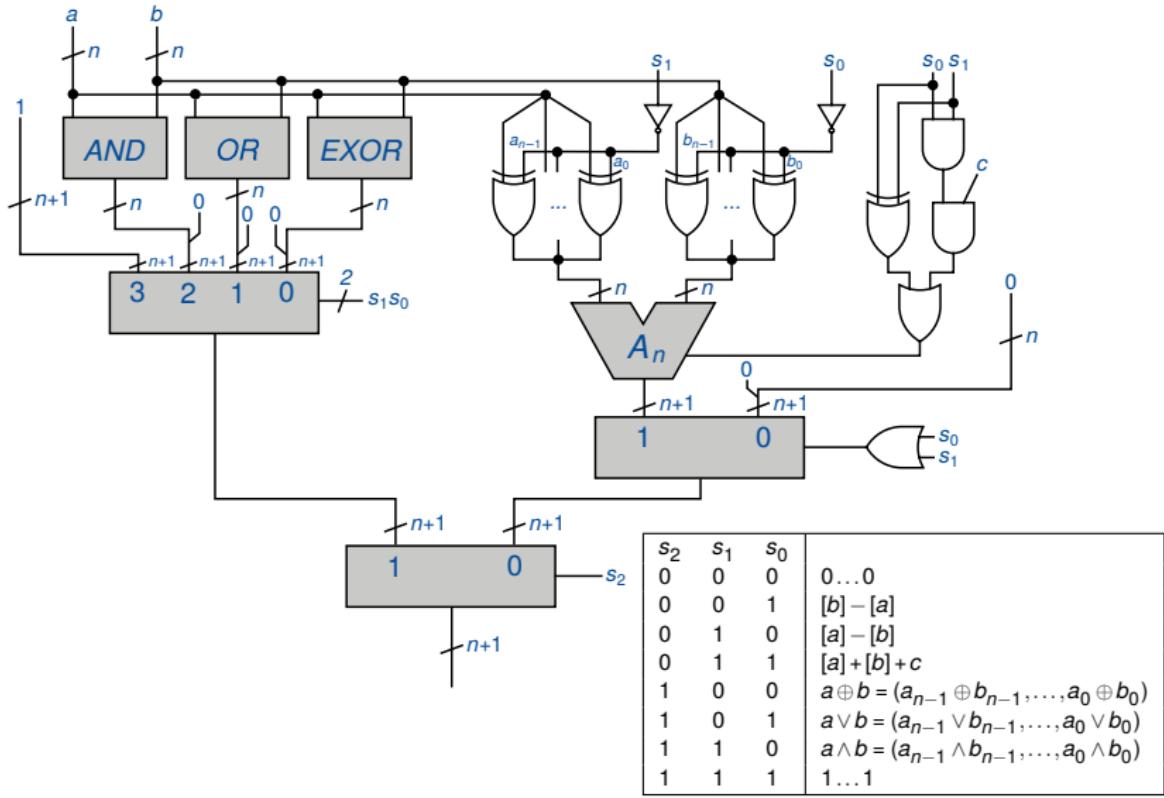
- Bei der Implementierung benötigen wir noch 4 Treiber (Tiefe 1), um *Regcken* auf 32 1-Bit-Multiplexer zu verteilen.



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>	0.35
$t_{HEC}$	Hold-Zeit von <i>Regcken</i> nach <i>ck</i>	0.10

- $t_{SDC}$  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) + maximale Verzögerungszeit Treiber ( $0.08 + 0.16 + 0.11$ ).
- $t_{HEC}$  ergibt sich aus Holdzeit D-FF – minimale Verzögerungszeit Multiplexer (Select bis Ausgang) - minimale Verzögerungszeit Treiber ( $0.14 - 0.02 - 0.02$ ).

# Schaltkreisrealisierung 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  $t_{select} = 0.28 \text{ ns}$  vor den Operanden gültig.
- Dann ist garantiert, dass der kritische Pfad nicht durch die select-Eingänge bestimmt wird.

■ Zeitverhalten der ALU:

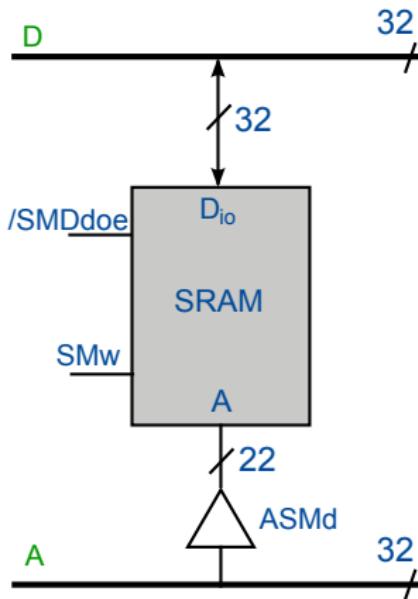
Symbol	Bezeichnung	$t^{\min}$	$t^{\max}$
$t_{ALU}$	Verzögerungszeit von $a$ , $b$ bzw. $c_{in}$ bis Ausgang		<u>3.25</u>

# Timing: SRAM

---

- 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		<u>12.0</u>
$t_{OED}$	Zeit von /SMD $d_{oe} = 0$ bis $D$		<u>7.0</u>
$t_{OEZ}$	Zeit von /SMD $d_{oe} = 1$ bis high-Z		<u>7.0</u>
$t_{wc}$	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

Prof. Dr. Christoph Scholl

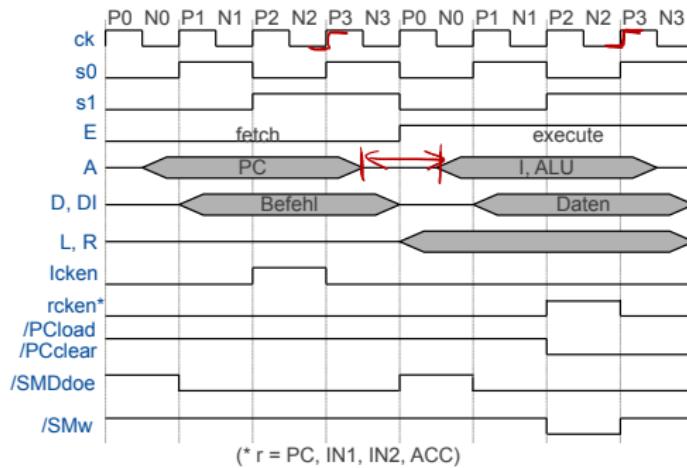
Institut für Informatik  
Sommersemester 2023

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  $t_c$



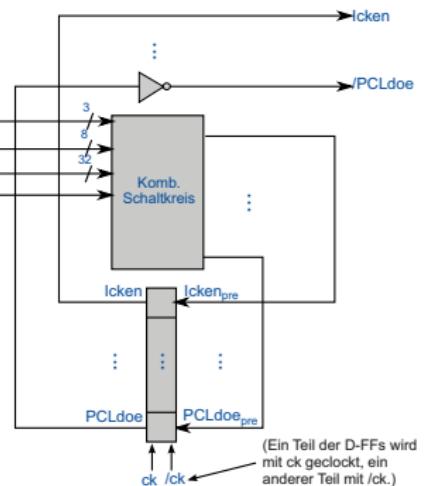
# Schritte der Analyse

---

- 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.

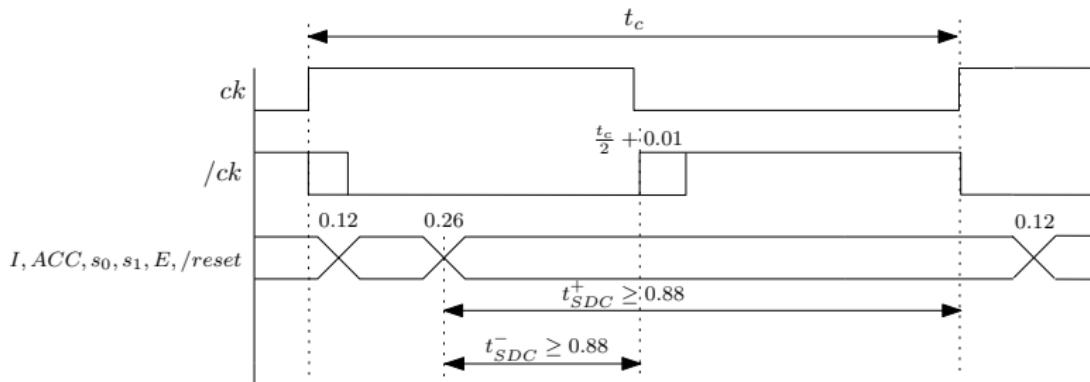
# Timing der Kontrolllogik (1/3)

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 low)	$t_c/2 + 0.13$	$t_c/2 + 0.56$
$t_{SDC}^+$	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 ( $s_0, s_1, E, I, ACC, reset$ ) 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  $\tau_{PCQ} = [0.12, 0.26]$  (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  $t_c$ .

$$\Rightarrow \frac{t_c}{2} + \underbrace{0.01}_{\text{Inverterverzögerung}} \geq \underbrace{0.88 + 0.26}_{\tau_{SDC}^+ \quad \tau_{PCQ} \text{ für FFs}}, \text{ d. h. } t_c \geq 2.26.$$

$$\Rightarrow t_c \geq \underbrace{0.88 + t_{SDC}^+}_{\tau_{SDC}^+ \quad \tau_{PCQ} \text{ für FFs}} + \underbrace{0.26}_{\tau_{PCQ} \text{ für FFs}} = 1.14$$

## 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).

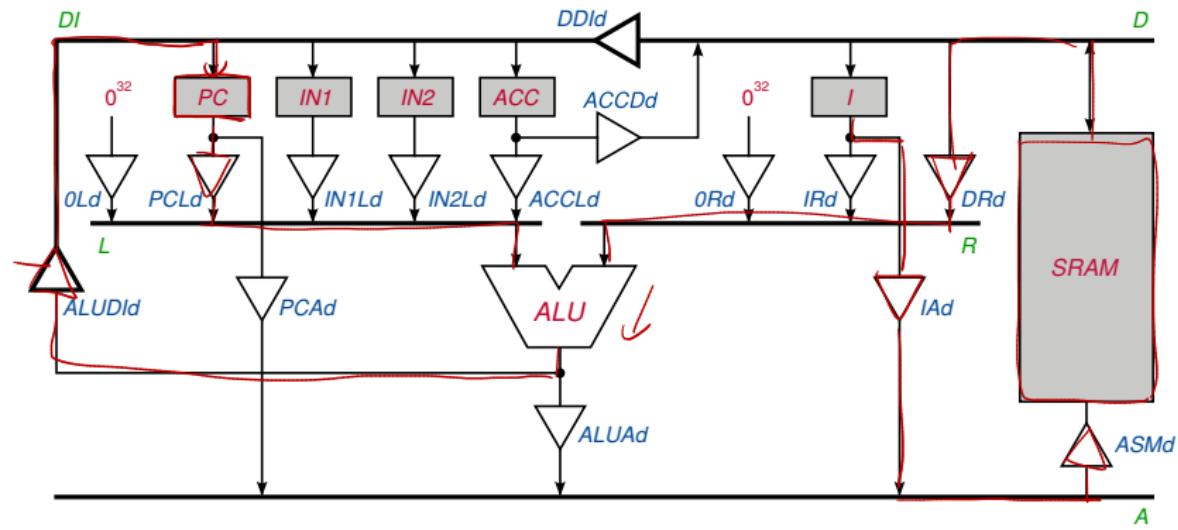
# Constraints

---

- $t_c \geq 2.26$

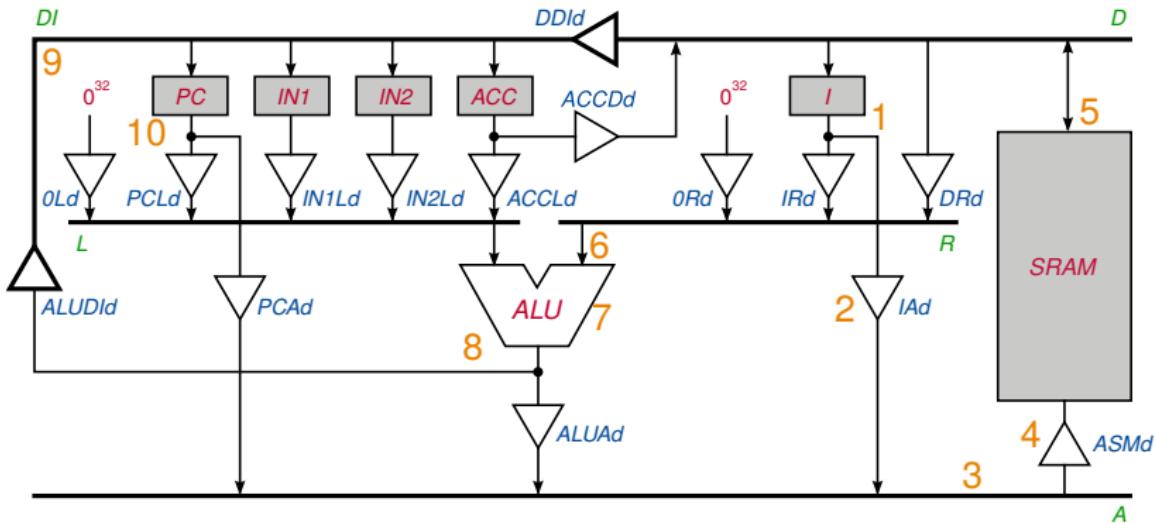
## Compute-Befehle

- Am zeitkritischsten ist Compute memory!



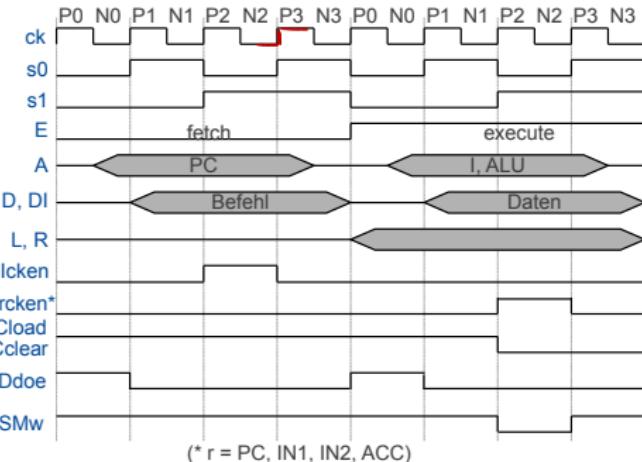
$[r] := [r] + [M(\langle i \rangle)]$ ; hier:  $[PC] := [PC] + [M(\langle i \rangle)]$

---



# Analyse allgemein

- Beginn der Analyse bei *P3* von Fetch als zeitlicher Bezugspunkt.
- Bei *P3* von Fetch wird der Befehl ins Instruktionsregister übernommen.



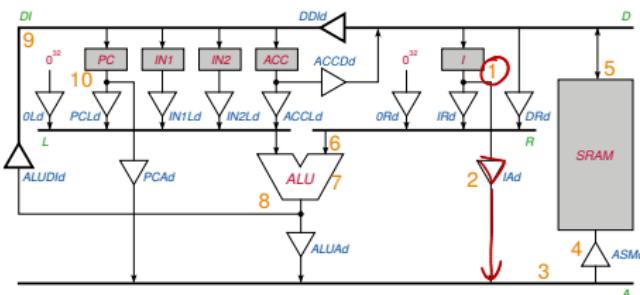
## I-Ausgänge (1/2)

I-Ausgänge gültig bei

$$\tau_1 = \underbrace{[0.12, 0.26]}_{\tau_{PCQ} \text{ von Register } I}$$

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

- $0^8 I_{23} \dots I_0$  wird über Treiber  $IAd$  auf Adressbus gegeben.
  - $0^8$  ist eine Konstante und steht daher ebenfalls zu  $\tau_1$  bereit.



# Constraints

---

- $\tau_1 = [0.12, 0.26]$
- $t_c \geq 2.26$

# I-Ausgänge (2/2)

$IAd$  enabled bei  $N0$  von Execute, d.h.

$/IAdoe$  aktiv zur Zeit

$$\begin{aligned}\tau'_2 &= t_c + \tau_{p,al}^- \\ &= t_c + \frac{t_c}{2} + [0.13, 0.56] \\ &= \frac{3}{2}t_c + [0.13, 0.56]\end{aligned}$$

$/IAdoe$  wird verteilt auf 32

Tristate-Treiber von  $IAd$ , also wieder 4

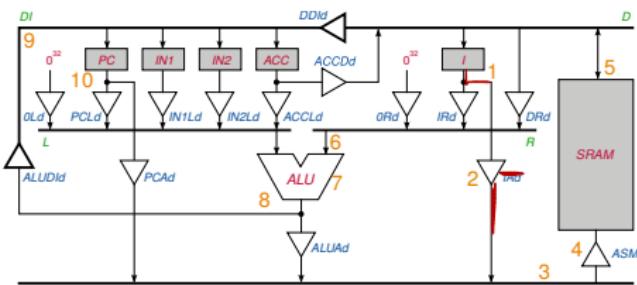
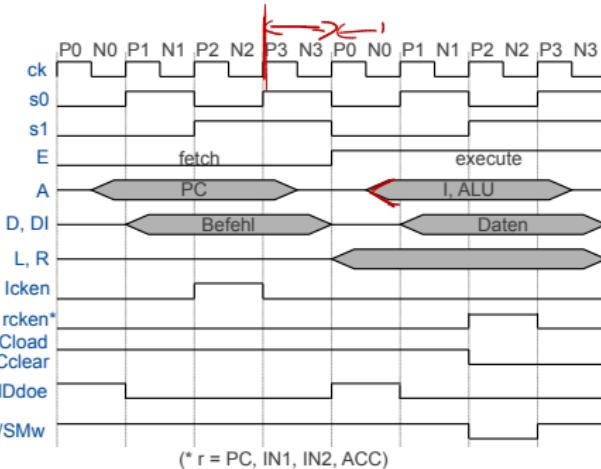
Treiber mit Tiefe 1.

$/IAdoe$  kommt bei  $IAd$  an zur Zeit

$$\begin{aligned}\tau_2 &= \tau'_2 + [0.02, 0.11] \\ &= \frac{3}{2}t_c + [0.15, 0.67]\end{aligned}$$

$I$  schon gültig vor Aktivierung von  $IAd$  bei Punkt 2, falls

$$\begin{aligned}\max(\tau_1) &\leq \min(\tau_2) \Leftrightarrow \\ 0.26 &\leq \frac{3}{2}t_c + 0.15 \Leftrightarrow \\ \frac{3}{2}t_c &\geq 0.11 \Leftrightarrow \\ t_c &\geq 0.08\end{aligned}$$



# Constraints

---

- $\tau_1 = [0.12, 0.26]$
- $\tau_2 = t_c + \tau_{p,al}^- = \frac{3}{2}t_c + [0.15, 0.67]$

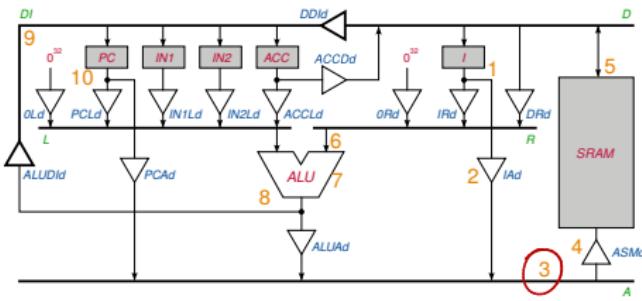
- $t_c \geq 2.26$
- $t_c \geq 0.08$

# Gültiges A (1/2)

→ A gültig zur Zeit

$$\begin{aligned}\tau_3 &= \tau_2 + \underbrace{[0.03, 0.11]}_{\text{Enable Zeit Treiber}} \\ &= \frac{3}{2} t_c + [0.18, 0.78]\end{aligned}$$

	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



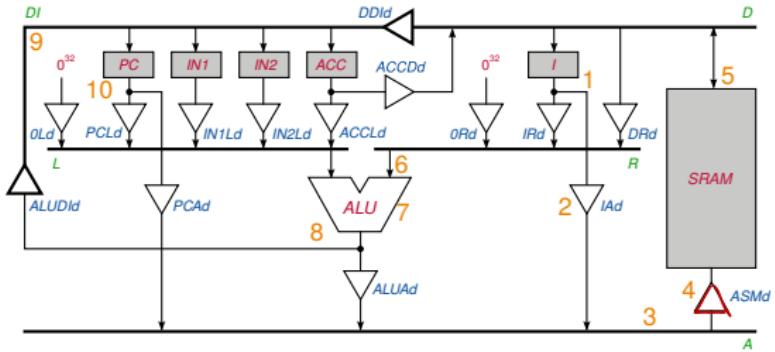
# Constraints

---

- $\tau_1 = [0.12, 0.26]$
- $\tau_2 = t_c + \tau_{p,al}^- = \frac{3}{2}t_c + [0.13, 0.56]$
- $\tau_3 = \tau_2 + [0.03, 0.11] = \frac{3}{2}t_c + [0.18, 0.78]$

- $t_c \geq 2.26$
- $t_c \geq 0.08$

# Gültiges A (2/2)



■  $ASMd$  immer enabled

- nur Treiber-Verzögerung berücksichtigt
- A an SM bei

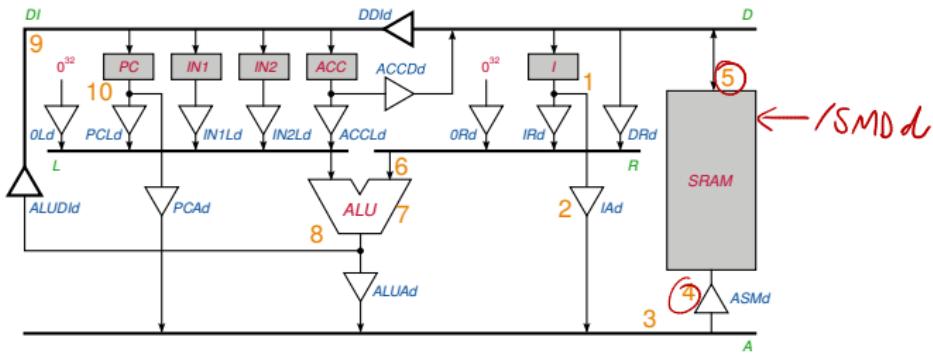
$$\tau_4 = \tau_3 + [0.02, 0.10] = \frac{3}{2} t_c + [0.20, 0.88]$$

# Constraints

---

- $\tau_1 = [0.12, 0.26]$
  - $\tau_2 = t_c + \tau_{p,al}^- = \frac{3}{2}t_c + [0.13, 0.56]$
  - $\tau_3 = \tau_2 + [0.03, 0.11] = \frac{3}{2}t_c + [0.18, 0.78]$
  - $\tau_4 = \tau_3 + [0.02, 0.10] = \frac{3}{2}t_c + [0.20, 0.88]$
- 
- $t_c \geq 2.26$
  - $t_c \geq 0.08$

# Daten am Speicherausgang (1/3)

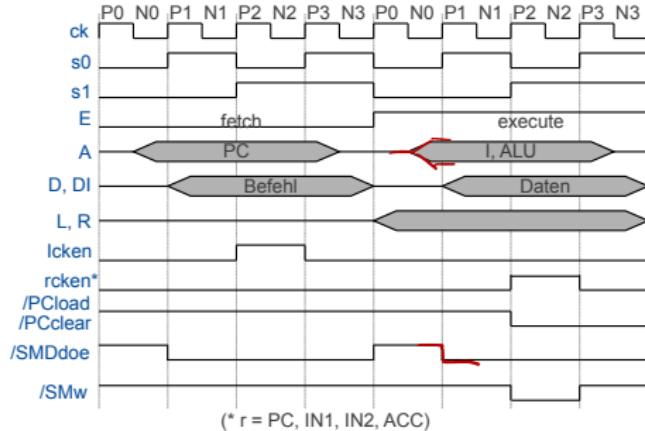


- Lesezugriffszeit von SRAM: [0.0, 12.0] (siehe Daten von CY7C1079DV33)  
→ Gültige Daten am Speicherausgang bei

$$\begin{aligned}\tau_5 &= \tau_4 + [0.0, 12.0] \\ &= \frac{3}{2} t_c + [0.20, 0.88] + [0.0, 12.0] \\ &= \frac{3}{2} t_c + [0.20, 12.88]\end{aligned}$$

- Das ist aber nur korrekt, wenn der Ausgangstreiber durch /SMDdoe 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 /SMDdoe = 0 bis D		7.0	
...				

- $/SMDdoe$  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  $\max(\tau_5) \geq \max(\tau'')$ :

$$\begin{aligned}\frac{3}{2}t_c + 12.88 &\geq 2 \cdot t_c + 7.41 \Leftrightarrow \\ \frac{1}{2}t_c &\leq 5.47 \Leftrightarrow \\ t_c &\leq \underline{\underline{10.94}}\end{aligned}$$

- Wir nehmen ab jetzt an, dass die Taktperiode  $t_c \leq 10.94$  ist und rechnen mit  $\max(\tau_5)$  weiter.
- (Es gilt auf jeden Fall  $\min(\tau_5) (= \frac{3}{2}t_c + 0.20) < \min(\tau'') (= 2 \cdot t_c + 0.12)$ )
- Sollte sich später ergeben, dass die minimale Taktperiode  $t_c > 10.94$ , dann müssten wir die Rechnung nochmals korrigieren.

## Constraints

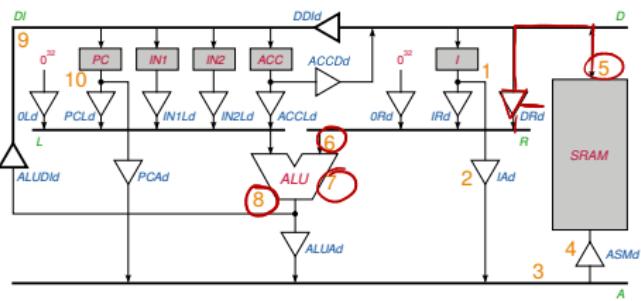
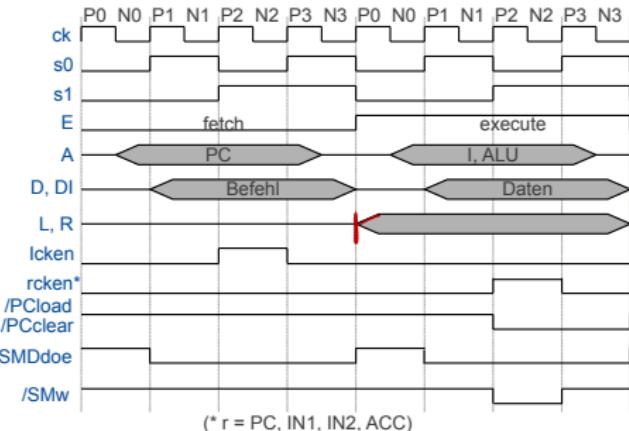
- $\tau_1 = [0.12, 0.26]$
  - $\tau_2 = t_c + \tau_{p,a/l}^- = \frac{3}{2}t_c + [0.13, 0.56]$
  - $\tau_3 = \tau_2 + [0.03, 0.11] = \frac{3}{2}t_c + [0.18, 0.78]$
  - $\tau_4 = \tau_3 + [0.02, 0.10] = \frac{3}{2}t_c + [0.20, 0.88]$
  - $\tau_5 = \tau_4 + [0.0, 12.0] = \frac{3}{2}t_c + [0.20, 12.88]$

$t_c \geq 2.26$   
 $t_c \geq 0.08$   
 $t_c \leq 10.94$

# Daten auf R

$DRd$  enabled bei  $P0$  von Execute, also einen Takt vor Ausgangstreiber von SM  
 → Enable nicht kritisch  
 → Daten auf R spätestens bei

$$\begin{aligned}\tau_6 &= \tau_5 + [0.02, 0.10] \text{ (Treiber-Verzögerung)} \\ &= \frac{3}{2} t_c + [0.22, 12.98]\end{aligned}$$



# Constraints

---

- $\tau_1 = [0.12, 0.26]$
- $\tau_2 = t_c + \tau_{p,al}^- = \frac{3}{2}t_c + [0.13, 0.56]$
- $\tau_3 = \tau_2 + [0.03, 0.11] = \frac{3}{2}t_c + [0.18, 0.78]$
- $\tau_4 = \tau_3 + [0.02, 0.10] = \frac{3}{2}t_c + [0.20, 0.88]$
- $\tau_5 = \tau_4 + [0.0, 12.0] = \frac{3}{2}t_c + [0.20, 12.88]$
- $\tau_6 = \tau_5 + [0.02, 0.10] = \frac{3}{2}t_c + [0.22, 12.98]$
- $t_c \geq 2.26$
- $t_c \geq 0.08$
- $t_c \leq 10.94$

- 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  $\tau_6$ .

- $f[2 : 0], c_{in}$  werden durch den kombinatorischen Schaltkreis der Kontrolllogik aus  $I_{31}, \dots, I_{24}$  berechnet.
  - $I$ -Ausgänge aber schon gültig bei  $\tau_1 = [0.12, 0.26]$ .
  - 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  $\max(\tau_6) = \frac{3}{2}t_c + 12.87$

# Constraints

---

- $\tau_1 = [0.12, 0.26]$
- $\tau_2 = t_c + \tau_{p,al}^- = \frac{3}{2}t_c + [0.13, 0.56]$
- $\tau_3 = \tau_2 + [0.03, 0.11] = \frac{3}{2}t_c + [0.18, 0.78]$
- $\tau_4 = \tau_3 + [0.02, 0.10] = \frac{3}{2}t_c + [0.20, 0.88]$
- $\tau_5 = \tau_4 + [0.0, 12.0] = \frac{3}{2}t_c + [0.20, 12.88]$
- $\tau_6 = \tau_5 + [0.02, 0.10] = \frac{3}{2}t_c + [0.22, 12.98]$
- $t_7 = 1.14$
- $t_c \geq 2.26$
- $t_c \geq 0.08$
- $t_c \leq 10.94$

# Voraussetzungen für die exakte Timinganalyse von *Compute memory*

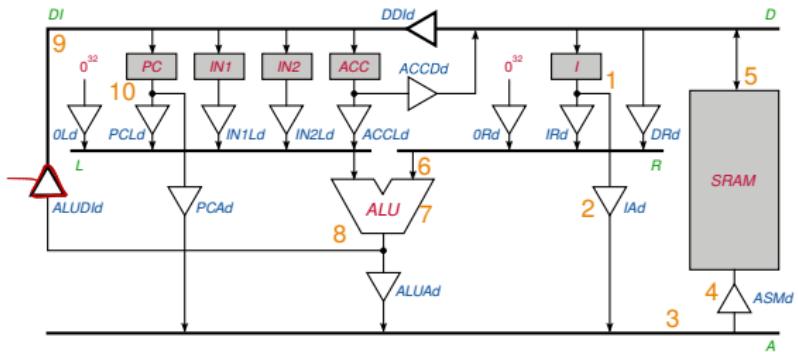
---

## ■ *ALU*

- Analyse der ALU (32-Bit mit Conditional Sum) unter folgender Annahme:
  - Funktionsselect-Signale liegen 0.28 ns vor den Daten an (unkritisch, da  $t_7 + 0.28 = 1.14 + 0.28 = 1.42 < \min(\tau_6) = \frac{3}{2}t_c + 0.22$ ).
  - Resultatsausgänge gültig 3.25 ns nachdem die Daten anliegen.

Symbol	Bezeichnung	$t^{\min}$	$t^{\max}$
$t_{select}$		0.28	
$t_{ALU}$	Verzögerungszeit von $a$ , $b$ bzw. $c_{in}$ bis Ausgang		3.25

# ALU-Ausgänge



■ Spätestens gültig bei

$$t_8 = \max(\tau_6) + \underbrace{3.25}_{\text{Delay ALU}} = \frac{3}{2} t_C + 12.98 + \underline{3.25}$$
$$= \frac{3}{2} t_C + 16.23$$

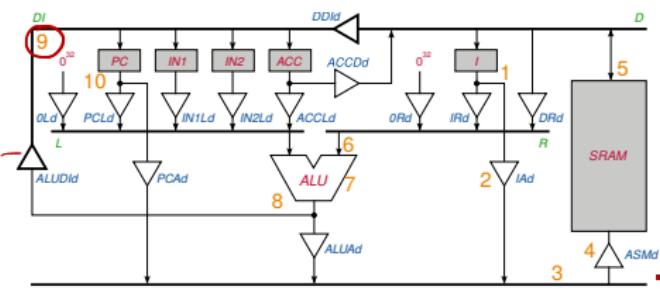
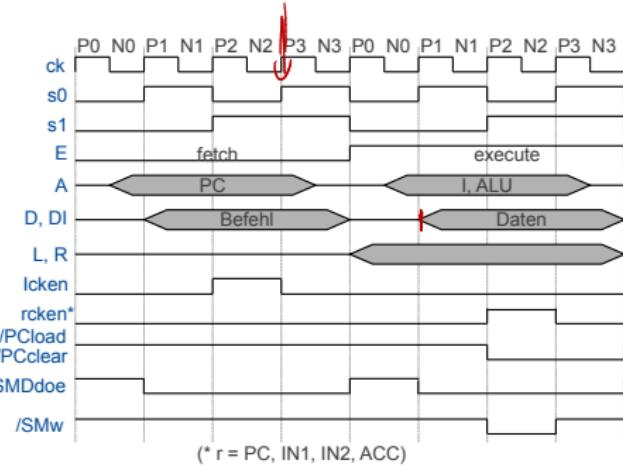
# Constraints

---

- $\tau_1 = [0.12, 0.26]$
- $\tau_2 = t_c + \tau_{p,al}^- = \frac{3}{2}t_c + [0.13, 0.56]$
- $\tau_3 = \tau_2 + [0.03, 0.11] = \frac{3}{2}t_c + [0.18, 0.78]$
- $\tau_4 = \tau_3 + [0.02, 0.10] = \frac{3}{2}t_c + [0.20, 0.88]$
- $\tau_5 = \tau_4 + [0.0, 12.0] = \frac{3}{2}t_c + [0.20, 12.88]$
- $\tau_6 = \tau_5 + [0.02, 0.10] = \frac{3}{2}t_c + [0.22, 12.98]$
- $t_7 = 1.14$
- $t_8 = \max(\tau_6) + 3.25 = \frac{3}{2}t_c + 16.23$
- $t_c \geq 2.26$
- $t_c \geq 0.08$
- $t_c \leq 10.94$

- /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  $t_c \leq 10.94$  ist auf jeden Fall auch für ALUDId gewährleistet, dass Treiber enabled, wenn Daten kommen.
- Berücksichtige nur Treiberverzögerung.
- Gültig spätestens bei

$$\begin{aligned}
 t_9 &= t_8 + 0.10 \\
 &= \frac{3}{2}t_c + 16.23 + 0.10 \\
 &= \boxed{\frac{3}{2}t_c + 16.33}
 \end{aligned}$$



# Constraints

---

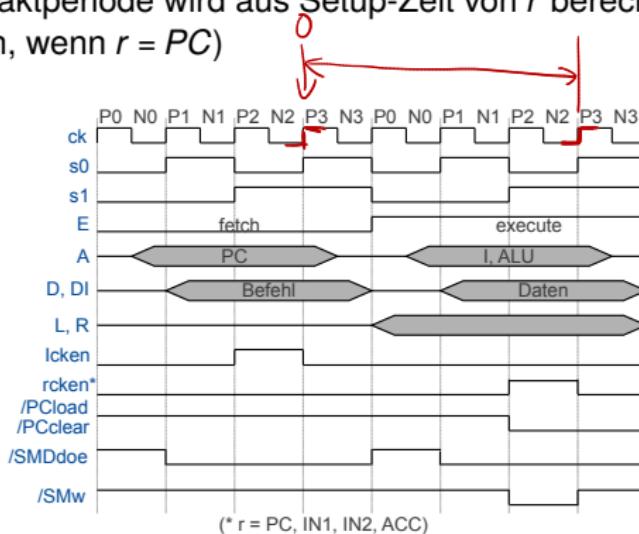
- $\tau_1 = [0.12, 0.26]$
- $\tau_2 = t_c + \tau_{p,al}^- = \frac{3}{2}t_c + [0.13, 0.56]$
- $\tau_3 = \tau_2 + [0.03, 0.11] = \frac{3}{2}t_c + [0.18, 0.78]$
- $\tau_4 = \tau_3 + [0.02, 0.10] = \frac{3}{2}t_c + [0.20, 0.88]$
- $\tau_5 = \tau_4 + [0.0, 12.0] = \frac{3}{2}t_c + [0.20, 12.88]$
- $\tau_6 = \tau_5 + [0.02, 0.10] = \frac{3}{2}t_c + [0.22, 12.98]$
- $t_7 = 1.14$
- $t_8 = \max(\tau_6) + 3.25 = \frac{3}{2}t_c + 16.23$
- $t_9 = t_8 + 0.10 = \frac{3}{2}t_c + 16.33$
- $t_c \geq 2.26$
- $t_c \geq 0.08$
- $t_c \leq 10.94$

# 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  $r = PC$ )



(\*  $r = PC, IN1, IN2, ACC$ )

# Constraints

---

- $\tau_1 = [0.12, 0.26]$
- $\tau_2 = t_c + \tau_{p,al}^- = \frac{3}{2}t_c + [0.13, 0.56]$
- $\tau_3 = \tau_2 + [0.03, 0.11] = \frac{3}{2}t_c + [0.18, 0.78]$
- $\tau_4 = \tau_3 + [0.02, 0.10] = \frac{3}{2}t_c + [0.20, 0.88]$
- $\tau_5 = \tau_4 + [0.0, 12.0] = \frac{3}{2}t_c + [0.20, 12.88]$
- $\tau_6 = \tau_5 + [0.02, 0.10] = \frac{3}{2}t_c + [0.22, 12.98]$
- $t_7 = 1.14$
- $t_8 = \max(\tau_6) + 3.25 = \frac{3}{2}t_c + 16.23$
- $t_9 = t_8 + 0.10 = \frac{3}{2}t_c + 16.33$
- $\tau_{10} = 4 \cdot t_c$
- $t_c \geq 2.26$
- $t_c \geq 0.08$
- $t_c \leq 10.94$

# 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 $ck$	0.65	
$t_{HLC}$	Hold-Zeit von $/L$ nach $ck$	0.04	
$t_{SEC}$	Setup-Zeit von $PCcken$ vor $ck$	0.35	
$t_{HEC}$	Hold-Zeit von $PCcken$ nach $ck$	0.10	
...	...	...	...

# Setup-Zeit von Zähler

- Setup-Zeit:  $t_{SDC} = 0.53 \text{ ns}$  (siehe Aufbau Zähler)

→ Bedingung:

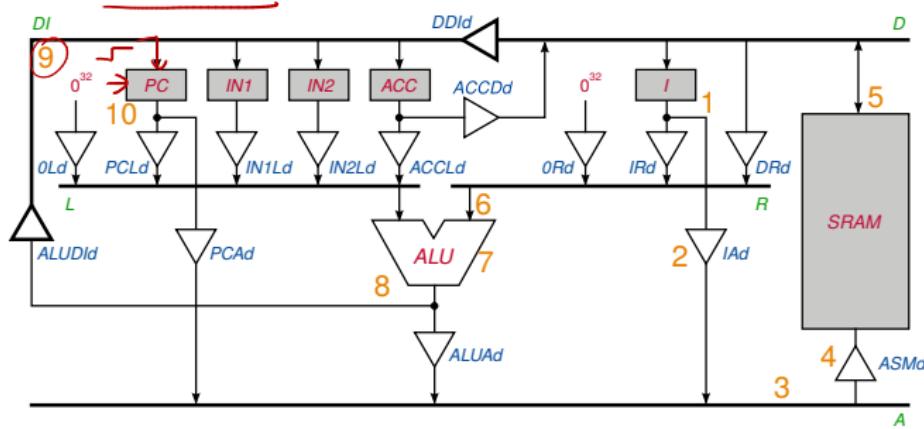
- $t_9 + 0.53 \leq \min(\tau_{10})$

$$\Leftrightarrow \frac{3}{2}t_c + 16.86 \leq 4t_c$$

$$\Leftrightarrow \frac{5}{2}t_c \geq 16.86$$

$$\Leftrightarrow t_c \geq 6.75$$

weil Abgrenzen bei P3 von execute



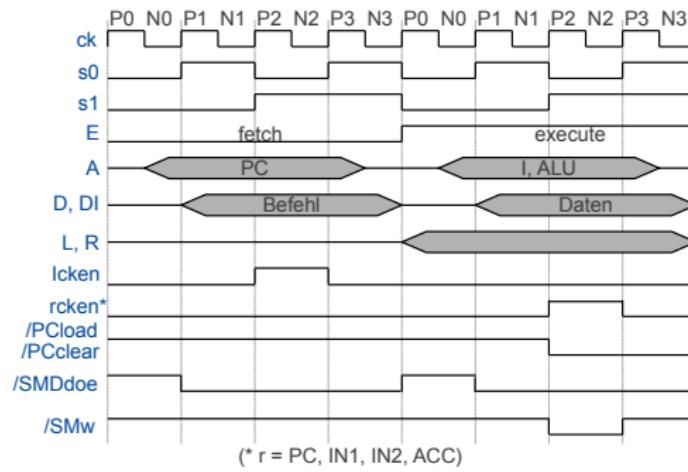
# Constraints

---

- $\tau_1 = [0.12, 0.26]$
- $\tau_2 = t_c + \tau_{p,al}^- = \frac{3}{2}t_c + [0.13, 0.56]$
- $\tau_3 = \tau_2 + [0.03, 0.11] = \frac{3}{2}t_c + [0.18, 0.78]$
- $\tau_4 = \tau_3 + [0.02, 0.10] = \frac{3}{2}t_c + [0.20, 0.88]$
- $\tau_5 = \tau_4 + [0.0, 12.0] = \frac{3}{2}t_c + [0.20, 12.88]$
- $\tau_6 = \tau_5 + [0.02, 0.10] = \frac{3}{2}t_c + [0.22, 12.98]$
- $t_7 = 1.14$
- $t_8 = \max(\tau_6) + 3.25 = \frac{3}{2}t_c + 16.23$
- $t_9 = t_8 + 0.10 = \frac{3}{2}t_c + 16.33$
- $\tau_{10} = 4 \cdot t_c$
- $t_c \geq 2.26$
- $t_c \geq 0.08$
- $t_c \leq 10.94$
- $t_c \geq 6.75$

# Es bleiben zu beachten:

- 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.



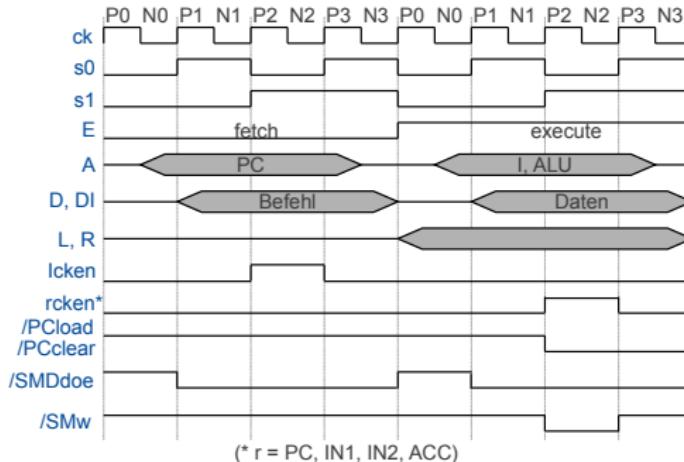
# Setup- und Hold-Zeiten von *rcken*

---

- *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  $r \in \{PC, ACC, IN1, IN2\}$ :  $t_{SEC} = 0.35$
  - ⇒  $\max(\tau_{11}) + 0.35 \leq 4t_c$ , d. h.  $3t_c + 0.26 + 0.35 \leq 4t_c$  bzw.
  - ⇒  $t_c \geq 0.61$  (unkritisch im Vergleich zu bisherigen Constraints)
- Hold-Zeit:
  - Für alle  $r \in \{PC, ACC, IN1, IN2\}$ :  $t_{HEC} = 0.10$
  - ⇒  $\min(\tau_{12}) \geq 4t_c + 0.10$ , d. h.  $0.12 \geq 0.10$
- (Analog auch für alle anderen Takte.)

# Setup- und Hold-Zeiten /PCload beim Zähler

- Setup /L bis  $ck$ :  $t_{SLC} = 0.65$
- Hold /L nach  $ck$ :  $t_{HLC} = 0.04$
- */PCload* (benötigt, wenn **neue** Werte in Zähler kommen) aktiv bei  $P2$  von Execute, inaktiv bei  $P0$  von Fetch



# Bedingungen für Setup- und Hold-Zeiten /*PCload* beim Zähler

- Setup: Am kritischsten, wenn /*PCload* seinen Wert bei der vorangegangenen Taktflanke geändert hat, d.h. bei P3 von execute bzw. P1 von fetch.

$$\max(\tau_{p,al}^+) + 0.65 \leq t_c \Leftrightarrow$$

$$0.41 + 0.65 \leq t_c \Leftrightarrow$$

$$t_c \geq 1.06$$

- Hold: Am kritischsten, wenn sich /*PCload* ändert, d.h. bei P2 von execute und P0 von fetch.

$$\min(\tau_{p,al}^+) \geq 0.04 \Leftrightarrow$$

$$0.12 \geq 0.04$$

→ Beide unkritisch im Vergleich zu bisherigen Constraints.

# Fazit: Zykluszeit und Befehlsrate

---

- Vorläufiges Ergebnis: Falls sich durch andere Befehle keine schärferen Bedingungen an die Zykluszeit ergeben, dann lautet sie:
  - $t_c \geq 6.75 \text{ ns}$   $= 6.75 \cdot 10^{-9} \text{ s}$
- Taktfrequenz:
  - $v = \frac{1}{6.75} \cdot 10^9 \text{ Hz} = \underline{\underline{148.1 \text{ MHz}}}$
- 8 Takte pro Befehl  $\rightarrow$  18.5 Millionen Befehle pro Sekunde, d. h. Befehlsrate von 18.5 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.

# Ausblick (1/2)

---

## ■ Beschleunigung

- Schnellere Komponenten, z. B. ALU
- Schnellere Adressberechnung (Treiber schon bei P0 öffnen)
- Evtl. Überdenken des kompletten Schemas des idealisierten Timings (z.B. Verkürzung des Fetch-Zyklus um 1 Takt)
- Schnellerer Speicher, Speicherhierarchie mit “Caches”
- Pipelining
- ...

# Ausblick (2/2)

---

- Fehlerbehandlung und Fehlertoleranz
  - Z.B. Überläufe bei ALU ...
  - Übertragungsfehler auf den Bus-Leitungen
  - ⇒ Parity-Check, Hamming-Code
- Verifikation
  - Ist der Entwurf der ReTI überhaupt korrekt?
  - Automatische Methoden für den Beweis der Korrektheit
- Architekturkonzepte  $\leadsto$  RA
  - Speicherhierarchie
  - Pipelining
  - Parallelität

