

# Vorlesungs-Skript

# **Grundlagen**

# **Technische Informatik**

Prof. Dr.-Ing. Markus Weinhardt



**HOCHSCHULE OSNABRÜCK**  
UNIVERSITY OF APPLIED SCIENCES

Labor für Digital- und Mikroprozessortechnik  
Ausgabe September 2020 (WS 2020/21)

# Inhalt

---

## Vorlesung

1. Einführung .....	3
2. Grundbegriffe und boolesche Funktionen .....	15
3. Grundlagen digitaler Schaltungen .....	27
4. Realisierung digitaler Schaltungen .....	47
5. Boolesche Algebra .....	53
6. Digitale Grundschaltungen .....	83
7. Asynchrone Rückkopplungen und Flipflops .....	91
8. Komplexe Logikbausteine .....	99
9. Zahlendarstellung, Arithmetik und Zeichencodierung .....	111
10. Synchrone Schaltungen .....	147
11. Strukturbeschreibung mit VHDL .....	171

## Zusatz-Dokumente

- Schaltalgebra-Regeln .....
  - Bereichsüberschreitungen bei Addition/Subtraktion .....
- 183  
185

# Regeln für diese Veranstaltung

## ► Vorlesung

- Keine Anwesenheitspflicht (aber dringend empfohlen!)
- Bitte pünktlich kommen!
- Zwischenfragen sind willkommen!



## ► Praktikum

- Anwesenheitspflicht (auch online!)  
→ Im Krankheitsfall bitte Attest nachliefern.
- Aufgaben müssen zu Hause vorbereitet werden! Für die Online-Versuche müssen Praktikums-Berichte abgegeben werden.
- Aufgaben werden in Teams aus drei Studierenden bearbeitet.
- Praktikumstermine dienen der Vertiefung des Lehrstoffs mit Unterstützung durch den Dozenten und einen Mitarbeiter!
- Bei erfolgreicher Teilnahme gibt es einen "Leistungsnachweis", aber keine Note.



# Was ist Technische Informatik?

- **Technische Informatik** ist eine der Säulen der Informatik (neben Theoretischer, Praktischer und Angewandter Informatik)
- Beschäftigt sich mit dem Entwurf, der logischen Struktur und der technischen Realisierung von Computer-Hardware
- Computer-Hardware: Digitale Schaltungen steigender Komplexität
  - elementare Hardware-Komponenten: *Transistoren* und *Logikgatter* (Kap. 3)
  - einfache Schaltnetze (s. Kap. 4 - 6)
  - mit Speicherelementen angereicherte Schaltwerke (s. Kap. 7, 8 und 10)
- Durch fortschreitende **Miniaturisierung** der Bauelemente werden Computer **ubiquitär** (allgegenwärtig) und sind häufig gar nicht mehr zu erkennen
  - Millionen zusammenspielende Schaltelemente ermöglichen die Konstruktion von Systemen höchster Komplexität (z. B. moderne Multicore-Prozessoren)
  - Alle Systeme lassen sich auf die gleichen Grundprinzipien zurückführen  
→ Verständnis der technischen Funktionsweise heutiger Computersysteme
- Mit **Theoretischer Informatik** verknüpft
  - Logik, Codierungstheorie, Endliche Automaten

# Einordnung der Vorlesung im Studienplan

6	Projekt/Projektmanagement			Bachelorarbeit und Kolloquium			
5	Software Architektur - Konzepte und Anwendungen	Software Engineering Projekt	Systems Engineering	Embedded Systems		Wahlpflichtmodul b)	
4	Objektorientierte Analyse und Design	Verteilte Systeme	Theoretische Informatik	IT-Sicherheit	Modellbildung und Simulation	Wahlpflichtmodul b)	
3	Programmierung 3 (TI)	Datenbanken	Mathematik 3 (TI)	Betriebssysteme	Computerarchitektur	Wahlpflichtmodul (überfachlich) a)	
2	Programmierung 2 (I)	Algorithmen und Datenstrukturen	Mathematik 2 (I)	Kommunikationsnetze	Rechnerorganisation	Orientierung und Methoden	
1	Programmierung 1 (I)	Mathematik 1 (I)	Grundlagen Technische Informatik	Elektrotechnisch-physikalische Grundlagen für TI			

## Technische Informatik im Curriculum:

1. Semester: Grundlagen Technische Informatik
2. Semester: Rechnerorganisation
3. Semester: Computerarchitektur
- ...
5. Semester: Embedded Systems

**5. Semester:**  
„Mobilitätsfenster“ für Auslandsaufenthalt!

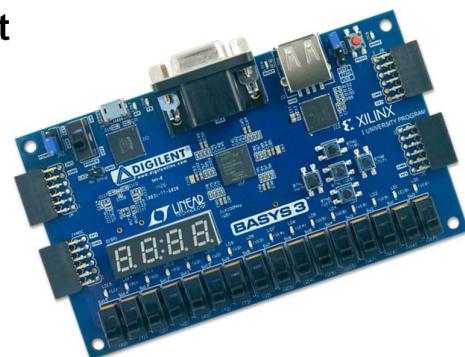
# Vorlesung - Themenübersicht

- 1. Einführung**
- 2. Grundbegriffe und boolesche Funktionen**
- 3. Grundlagen digitaler Schaltungen**
- 4. Realisierung digitaler Schaltungen**
- 5. Boolesche Algebra**
- 6. Digitale Grundschaltungen**
- 7. Asynchrone Rückkopplungen und Flipflops**
- 8. Komplexe Logikbausteine**
- 9. Zahlendarstellung, Arithmetik und Zeichencodierung**
- 10. Sychrone Schaltungen**
- 11. Strukturbeschreibung mit VHDL**



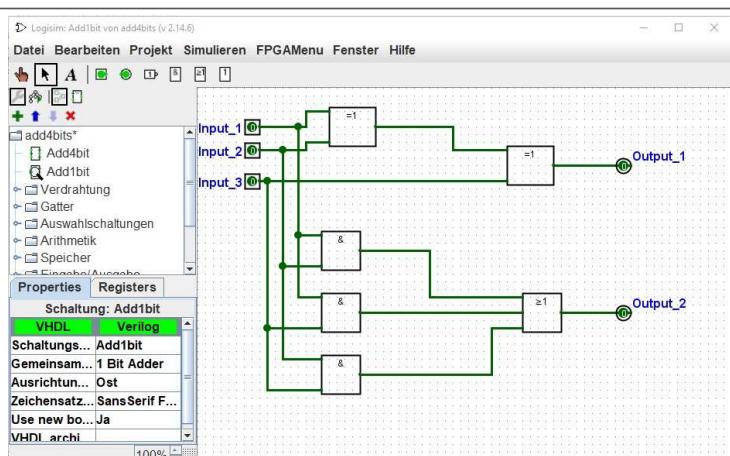
# Praktikums-Versuche

1. **Simulation von einfachen Logik-Gattern und Normalformen einfacher Schaltungen**
2. **Minimierung und Simulation einfacher Schaltungen**
3. **Simulation eines Volladdierers und eines Code-Umsetzers**
4. **Praktischer Aufbau einer Schaltung mit realen Logikbausteinen und FPGAs**  
→ Dieser Versuch findet im Labor für Digital- und Mikroprozessor-Technik statt. Die Termine werden noch bekanntgegeben!
5. **Synchrone Schaltungen**



## Praktikum: Entwicklungs-Software Logisim-Evolution

- ▶ Im Praktikum wird das kostenlose Programm **Logisim-Evolution** für die Simulation einfacher Schaltungen und für die Programmierung von FPGAs (konfigurierbarer Hardware) verwendet.



- ▶ **Installation**

Datei *logisim-evolution.zip* aus NetCase (Unterordner Praktikum/Tools) herunterladen und entpacken. Dann *logisim-evolution.bat* ausführen. Java muss installiert sein! (Details finden Sie in der Datei *Einrichten\_der\_Entwicklungsumgebung.pdf*.)

- ▶ **Zur Durchführung der Simulationen benötigt!**

(Zur Synthese auf FPGAs in Versuch 4 werden im Labor weitere Tools zur Verfügung gestellt.)

## Literaturhinweise (Auswahl)

---

Dirk W. Hoffmann

„Grundlagen der Technischen Informatik“ (5. Auflage)

Hanser Verlag, München 2016

Bibliothek: TVA 140 930/5

Bernd Becker, Paul Molitor

„Technische Informatik: Eine einführende Darstellung“

Oldenbourg-Verlag, München 2008

Bibliothek: TTA-A 12/2008

Winfried Gehrke, Marco Winzker, Klaus Urbanski, Roland Woitowitz

„Digitaltechnik“ (7. Auflage)

Springer, Heidelberg 2016.

Christian Siemers, Axel Sikora

„Taschenbuch Digitaltechnik“ (2. Auflage)

Fachbuchverlag Leipzig im Hanser-Verlag, 2007.



→ zu finden in der HS Zentralbibliothek Westerberg

## Geschichte

# Geschichte: 300 Jahre Computertechnik

## Um 1700: Duales Zahlensystem

Gottfried Wilhelm Leibniz

Leibniz erfindet eine mechanische Rechenmaschine für die vier Grundrechenarten

## 1847: Mathematische Formulierung der Logik

George Boole

Grundlagen, die die Basis heutiger digitaler Schaltungen darstellen

## 1936: Erster frei programmierbarer Rechner (Z1)

Konrad Zuse

Mechanischer Rechner

Später: Z3 (basierend auf Relais)

## 1944: Erster elektronischer Rechner (ENIAC)

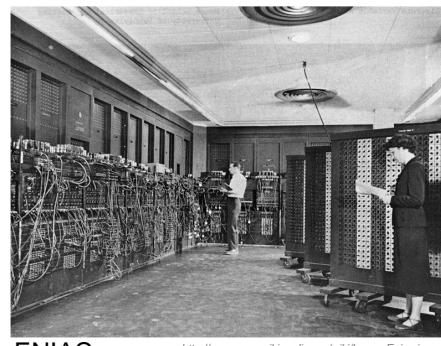
Eckert und Mauchly

Basierend auf Elektronenröhren

Flächenbedarf: 140 m<sup>2</sup>, Verlustleistung: 150 kW  
(18 000 Elektronenröhren)

## 1953: Erster Rechner auf Transistorbasis

Universität Manchester



<http://commons.wikimedia.org/wiki/Image:Eniac.jpg>

# Technologische Meilesteine

## 1947: Erfindung des Transistors (Schalter auf Halbleiterbasis)

## 1958: Realisierung des ersten integrierten Schaltkreises (IC)

2 Transistoren

## 1961: Erfindung der Transistor-Transistor-Logik (TTL)

Einfache logische Verknüpfungen in einem IC

## 1963: Die CMOS-Technologie wird erfunden

Grundlage heutiger integrierter Digitalschaltungen (u.a. auch PC-Prozessoren)

## 1966: Die Logikfamilie 74xx wird eingeführt

## 1966: Erfindung des digitalen Speichers

## 1968: Integrierte Standardbausteine auf Basis der CMOS-Technologie

Entscheidender Vorteil: Geringere Verlustleistung gegenüber TTL

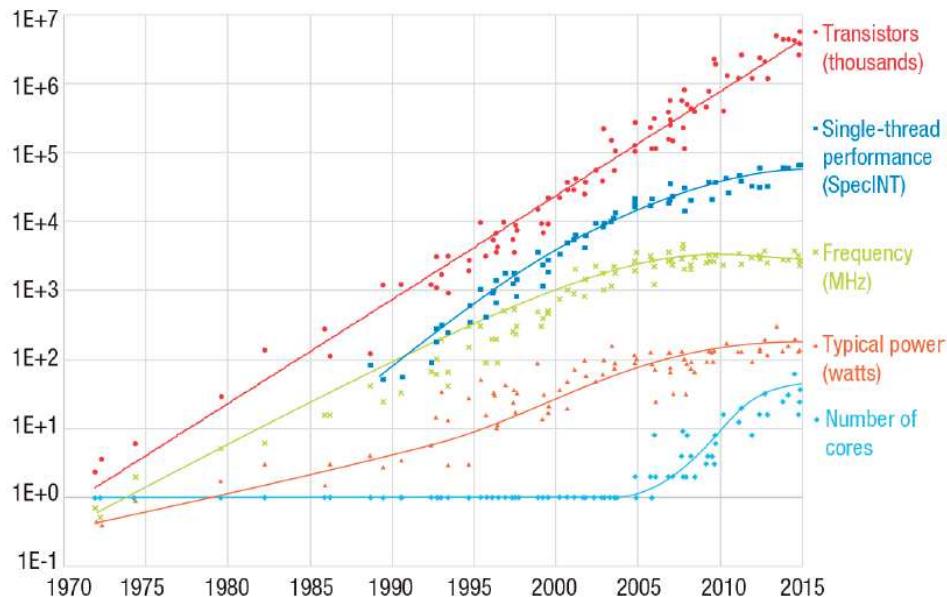
## 1971: Erster Mikroprozessor (auf einem IC): Intel 4004

4-Bit-Prozessor(!), 2250 Transistoren, 740 kHz

# Mikroprozessoren & Moores Law

→ Zahl der Transistoren verdoppelt sich etwa alle 18 Monate!

1971: Intel 4004 (2250 Transistoren), 2011: 10-Core Xeon Westmere-EX (2,6 Mrd. Trans.)



Quelle:  
[www.hipeac.org/roadmap](http://www.hipeac.org/roadmap)

**Trends:** - Transistorzahl nimmt weiter zu, aber nicht die Frequenz!

(Zu hohe Leistungsaufnahme, Kühlung nicht mehr möglich!)

- Deshalb Multi-Core-Prozessoren! (Aber: Software muss angepasst werden!)

## Aufbau Computer-Hardware

- ▶ Schichtenmodell/Programmierung
- ▶ Digitaltechnik
- ▶ Von Neumann Architektur
- ▶ Hauptspeicher
- ▶ Mikroprozessor

# Schichtenmodell

---

## ► „Was ist ein Computer überhaupt?“

Die Antwort auf diese Frage hängt stark von der Sichtweise ab, also wie weit man von dem eigentlichen physikalischen Rechner abstrahiert.

## ► Schichtenmodell

Anwendungsebene	Webbrowser, Tabellenkalkulation etc.
Ebene der höheren Programmiersprachen	Java, C, C++ etc.
Assembler-Ebene	Prozessorspezifisch, z. B. ARM-Assembler
Betriebssystem-Ebene	Windows, Linux, RTOS etc.
Maschinensprache-Ebene	Hardware/Software-Schnittstelle, binär
<b>Hardware-Ebene</b>	CPU/Mikroprozessor (z. B. Intel, ARM, MIPS)

- Die Technische Informatik beschäftigt sich vor allem mit der Hardware-Ebene, also mit **digitalen Schaltungen**.



# Programmierung der Computer-Hardware

---

- Einen Prozessor in seiner eigenen Maschinensprache zu programmieren, ist sehr mühsam (niedrige Abstraktionsebene).
- Programme müssten für jeden neuen Prozessor angepasst (portiert) werden!
- Deshalb werden *höhere Programmiersprachen* wie Java, C oder C++ verwendet!
- Da ein Prozessor Java oder C nicht (direkt) versteht, benötigt man einen **Übersetzer** oder **Compiler**, der ein Java- oder C-Programm (*Quellcode* in Textdatei/en) in das jeweilige Maschinenprogramm (*Maschinencode*) übersetzt.
- Für die Programmierung sind neben Compilern auch Editoren und weitere Werkzeuge notwendig. Diese werden als **Entwicklungs-umgebung** zusammengefasst.  
➔ siehe Vorlesung *Programmierung 1 (I)*

# Hardware: Die Welt der Bits

---

Computer-Hardware basiert auf der **Digitaltechnik**.

Um die Implementierung einer digitalen Schaltung möglichst einfach zu gestalten, werden die Werte als Nullen und Einsen (Bits = „binary digits“, logische oder boolesche Werte) codiert.

In einfachen Fällen reicht ein einzelnes Bit:

Taster geschlossen, Taster offen

Licht an, Licht aus

Strom fließt, Strom fließt nicht usw.

Möchte man die Werte genauer erfassen, werden mehrere Bits verwendet:

Für ein Signal mit N diskreten Werten werden  $\log_2(N)$  Bits benötigt

(Falls N keine 2er-Potenz ist, nächsthöheren ganzzahligen Wert für die Bitanzahl wählen!)

Beispiel Audio-CD: 16 bit, also  $2^{16}$  unterschiedliche Werte

## Digitaltechnik: Warum Bits bzw. Nullen und Einsen?

---

Die Darstellung von Informationen in Form von Bits hat zwei entscheidende Vorteile:

### Störsicherheit

Wird beispielsweise festgelegt, dass Nullen und Einsen durch Spannungsbereiche dargestellt werden,

z.B.: „Null“ = kleiner 1V und „Eins“ = größer 2V,

dann ist es völlig egal, ob eine Null durch ein Signal mit einer Spannung von 0,2V oder 0,8V dargestellt wird.

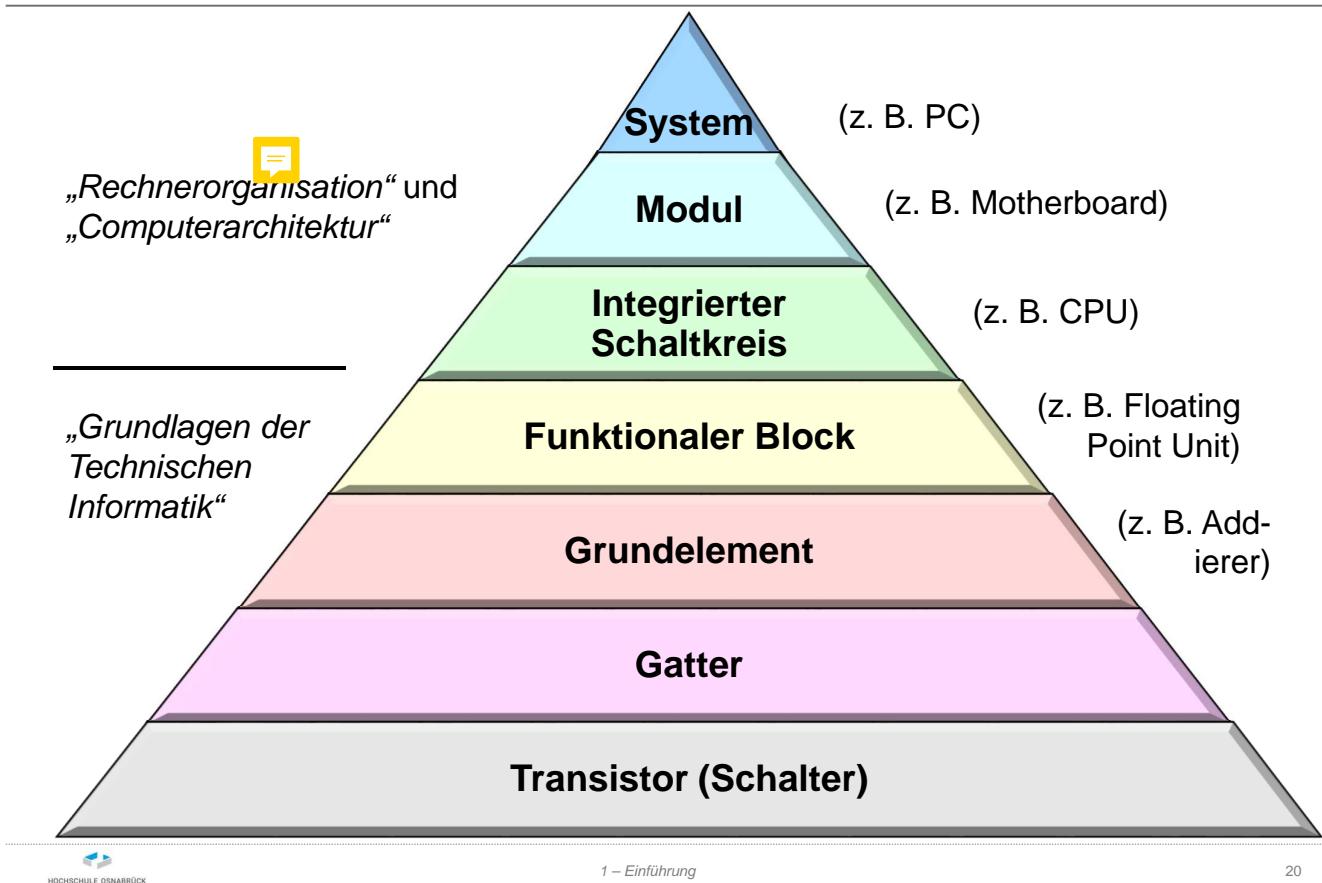
... Kleinere Störungen auf der Signalleitung haben keinen Einfluss!

### Implementierungsfreundlichkeit

Berechnungen, die mit Bits ausgeführt werden, lassen sich mit Hilfe elektrischer Schalter (Transistoren) realisieren (Schalter an, Schalter aus)

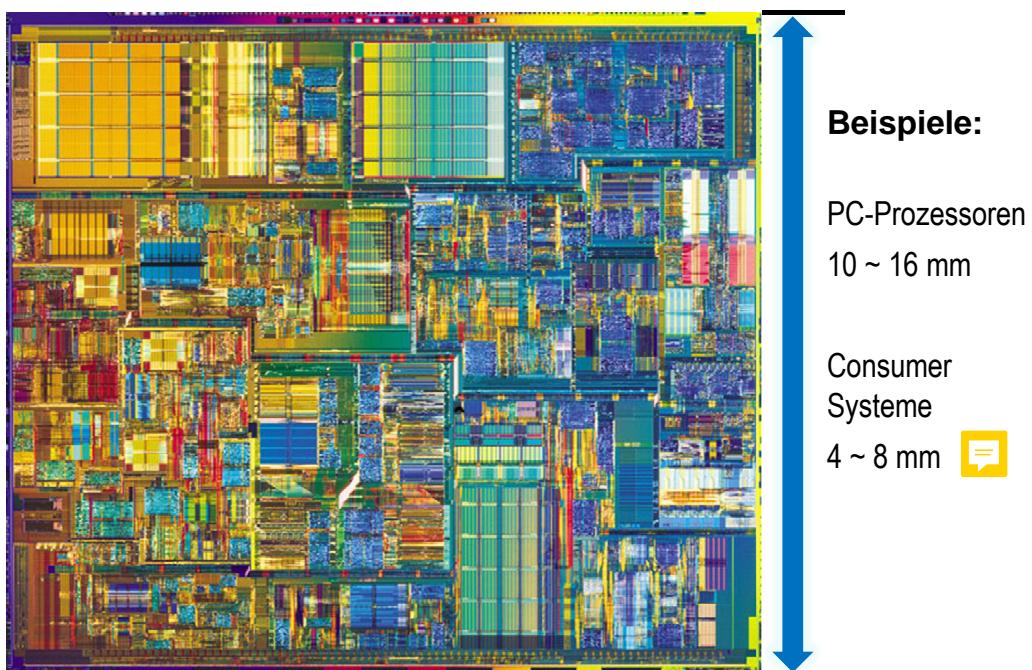
Diese Technik haben wir im Griff: Auf der Größe eines Daumennagels lassen sich auf einem Halbleiter-Kristall heute mehrere Milliarden solcher Schalter in einer digitalen Schaltung realisieren!

# Hierarchie eines digitalen Systems



## Digitale Systeme: Kleine Fläche, Große Leistung

Bringt man viele Transistoren auf einem Stück Silizium unter, kann man Systeme realisieren, die komplexe Aufgaben übernehmen können

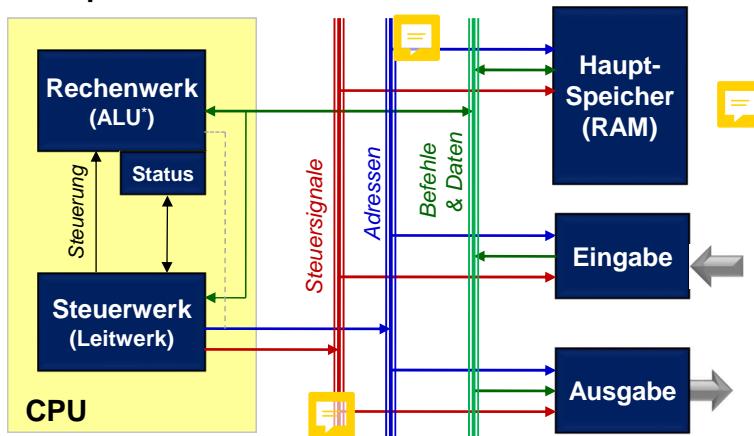


# Hardware-Aufbau eines programmierbaren Rechners

## Von Neumann Architektur:

- ▶ Grundlage der meisten heute gebräuchlichen Computer
- ▶ Hauptspeicher enthält Programmdaten *und* Programmbefehle!

Mikroprozessor:



\*ALU: Arithmetic Logical Unit

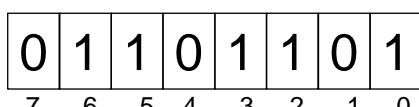
## Hardware-Komponenten: Hauptspeicher

- ▶ **Hauptspeicher wird durch *Halbleiterspeicher* realisiert, enthält:**

- Programmdaten
- Programmbefehle

- ▶ **Besteht aus einer linearen Folge von Speicherworten**

- Ein Speicherwort besteht aus n Bits (häufig n=8, 16, 32 oder 64).
- Ein Wort der Länge n=8 heißt **Byte**.



Bit 7: MSB (most-significant bit)  
Bit 0: LSB (least-significant bit)

- ▶ **Der Prozessor greift über Adressen auf die Speicherworte zu**

- Die Adresse gibt an, an welcher Stelle das Speicherwort im Hauptspeicher steht.

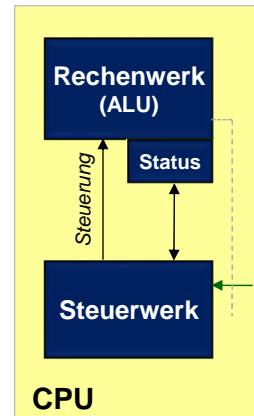
- ▶ **Hauptspeicher ist Random Access Memory (RAM)**

- Adressen können in beliebiger Reihenfolge verwendet werden („wahlfreier Zugriff“).
- Daneben gibt es unveränderbaren **ROM** (Read-Only Memory, siehe Kapitel 5) und **Register** (siehe Kapitel 7).

# Hardware-Komponenten: Mikroprozessor



- ▶ Führt eine Folge von Befehlen („Programm“) aus, die zusammen mit den Daten im Hauptspeicher stehen.
- ▶ Das Steuerwerk kontrolliert den Ablauf:
  - Hole nächsten Befehl aus dem Hauptspeicher.
  - Interpretiere den Befehl.
  - Führe den Befehl aus (z. B. Daten einlesen, verknüpfen, speichern oder ausgeben).
- ▶ Jeder Prozessor implementiert einen eigenen Befehlssatz („*instruction set*“)
  - Jeder Befehl ist binär kodiert.
  - Ein Dekodierer im Prozessor weist das Steuerwerk an, den Befehl auszuführen.



Details folgen in den Vorlesungen *Rechnerorganisation* und *Computerarchitektur*.



## 2. Grundbegriffe und boolesche Funktionen

### Digitale Signale

**Digitale Signale sind im allgemeinen sowohl**

**wertdiskret**

**als auch**



**zeitdiskret**



**„Und...? Was bringt einem das??“**

#### **Wertdiskret**

Die Größe kann durch einen Zahlenwert mit endlicher Stellenzahl repräsentiert werden, z.B. binär in Form einzelner Bits



#### **Zeitdiskret**

Ändert sich der Eingangswert einer Schaltung für eine bestimmte Zeit nicht, kann während dieser Zeit die Berechnung der Ausgangswerte erfolgen  
*(ohne dass auf geänderte Eingangswerte reagiert werden müsste)*



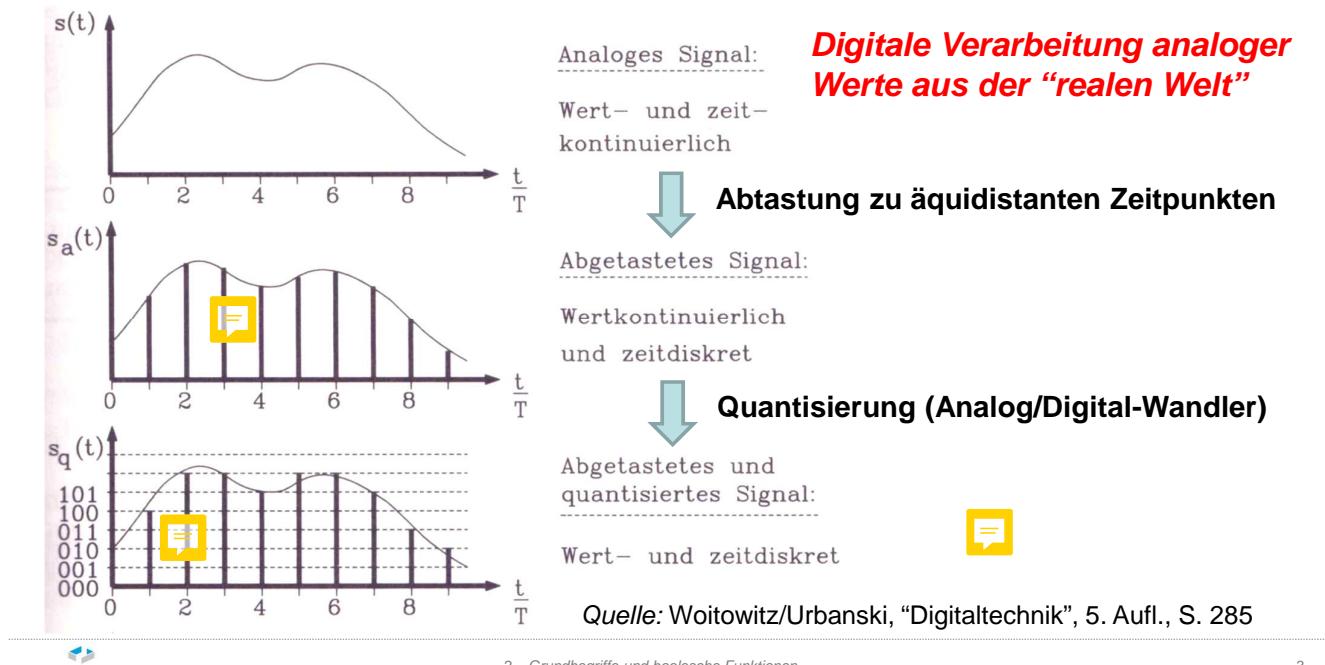
# Digitale und analoge Größen

## Analoge Größen

- können (innerhalb des Dynamikbereiches) jeden beliebigen Wert annehmen

## Digitale Größen

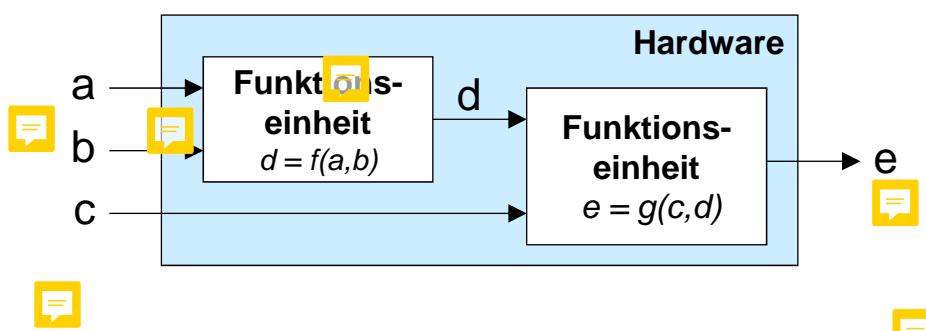
- können nur bestimmte, fest definierte Werte annehmen



# Aufbau und Funktionsweise (digitaler) Hardware

Hardware besteht aus **Funktionseinheiten** und **Signalen**

Beispiel:



## Funktionseinheiten:

Verknüpfen Werte der Eingangssignale

Ergebnis der Verknüpfung: Ausgangswerte ( $\rightarrow$  Ausgangssignale)

**Signale:** Verbindung von Funktionseinheiten, Übertragung von Werten.

Binärsignale: Nehmen nur die *logischen* oder *booleschen* Werte 0 oder 1 an.

**Digitale Hardware:** Verknüpft nur Binärsignale

# Ereignisgesteuerte Verarbeitung & Nebenläufigkeit

Eine Hardware arbeitet **ereignisgesteuert** und **nebenläufig**.

## Ereignissesteuerung:

Ereignisse (d.h. Änderungen) an den Eingängen einer Funktionseinheit bewirken eine Neuberechnung der Ausgangswerte durch die Funktionseinheit.

Damit werden neue Ereignisse auf den Ausgangssignalen erzeugt, welche in nachfolgenden Schaltungsteilen weitere Berechnungen bewirken können.

## Nebenläufigkeit:

Alle Funktionseinheiten einer Hardware führen gleichzeitig die Verknüpfungen der jeweiligen Eingangssignale durch.

## Berechnungszeit:

Eine reale Funktionseinheit benötigt eine (vom Aufbau der Einheit abhängige) Zeit für die Verknüpfung der Eingangssignale

→ Ein Ereignis am Eingang wirkt sich erst verzögert am Ausgang aus.

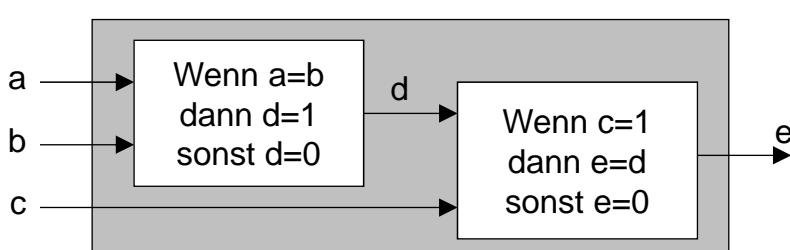
Die Berechnungszeit wird als **Verzögerungszeit** bezeichnet.

Hat eine Funktionseinheit mehr als einen Eingang und einen Ausgang, existieren mehrere Verzögerungszeiten (zeitl. Beziehungen zwischen Ein- und Ausgängen)



## Beispiel: Ereignisgesteuerte Hardware

### Beispiel einer digitalen Hardware

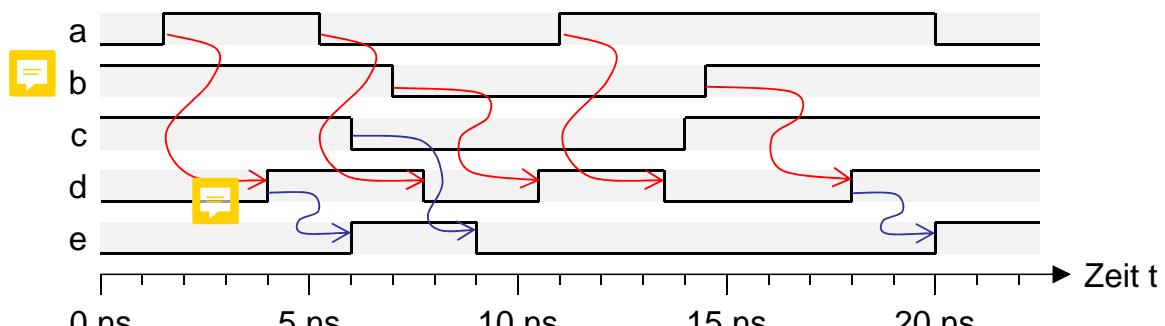


Angenommene  
Verzögerungszeiten:

$$\begin{aligned}td_{a,d} &= 2,5 \text{ ns} \\td_{b,d} &= 3,5 \text{ ns} \\td_{c,e} &= 3,0 \text{ ns} \\td_{d,e} &= 2,0 \text{ ns}\end{aligned}$$



### Beispielhaftes Zeitdiagramm



# Software vs. Hardware

## Software

Sequentielle Abarbeitung einzelner Befehle eines Programms.

Typische Programmiersprachen wie Java oder C/C++ ermöglichen keine explizite Formulierung einer ereignisgesteuerten Verarbeitung.

Parallelität auf Multicore-Prozessoren nur zwischen Programmen (Tasks, Threads) oder durch spezielle Programmiersprachen möglich.

## Hardware

(Viele) gleichzeitig aktive, parallele Funktionseinheiten.

Steuerung der Abläufe durch Signalverdrahtung festgelegt.

**Hardware arbeitet anders als Software!**

# Darstellung von Logikwerten

Binäre Werte können durch beliebige physikalische Größen dargestellt werden.

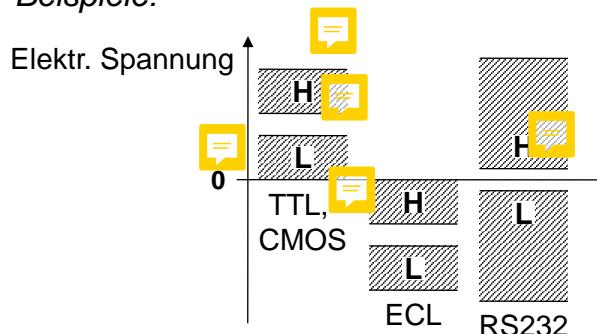
Beispiele: Spannung, Strom, Licht, Druck, Frequenz

Gängig ist heute die Darstellung durch Spannungen.

## Logikpegel:

Wertebereich der physikalischen Größe wird den Logikpegeln H und L zugeordnet.

### Beispiele:



Vorteile der binären Informationsdarstellung:

- Störsicherheit
- Implementierbarkeit

# Logikpegel vs. Logikzustand

**Logikpegel:** Dient zur Beschreibung der elektrischen Arbeitsweise einer Schaltung.

Der Logikpegel reflektiert den elektrischen Wert (H,L) eines Signals.

**Logikzustand:** Dient zur Beschreibung der logischen Arbeitsweise einer Schaltung.

Der Logikzustand reflektiert den logischen Wert (0,1) eines Signals.

Die Zuordnung **Logikzustand  $\leftrightarrow$  Logikpegel** muss zusätzlich festgelegt werden!

## Zuordnung Logikpegel $\leftrightarrow$ Logikzustand

Für die Zuordnung **Logikzustand  $\leftrightarrow$  Logikpegel** existieren zwei Möglichkeiten:

 **positive Logik:**  $(H,L) \Leftrightarrow (1,0)$   
**negative Logik:**  $(H,L) \Leftrightarrow (0,1)$

Meist wird positive Logik verwendet!

Aber es gibt auch Ausnahmen, z.B. die RS232-Schnittstelle.



In Schaltbildern werden Signale mit negativer Logik mit dem **Polaritätsindikator** gekennzeichnet.

## Dualzahlen

Folgen von logischen Werten (d. h. Nullen und Einsen) dienen zur Repräsentation von (natürlichen) Zahlen im Zweiersystem.

→ **Dualzahlen** sind aus Bits ("binary digits") zusammengesetzt.  
(Mehr zu Zahlensystemen in Kap. 9!)

Wert einer Zahl A mit n Stellen (Bits):

(Summenformel)

$$V(A) = \sum_{i=0}^{n-1} a_i \cdot 2^i = a_0 \cdot 2^0 + a_1 \cdot 2^1 + \dots + a_{n-1} \cdot 2^{n-1}$$

mit  $A = (a_{n-1} a_{n-2} \dots a_1 a_0)$  Ziffernfolge

$a_i \in \{0,1\}$  Zifferngewicht

Beispiel: ( $n = 5$ )

$$A = (a_4 a_3 a_2 a_1 a_0) = (10011)$$

$$\begin{aligned} V(A) &= a_0 \cdot 2^0 + a_1 \cdot 2^1 + a_2 \cdot 2^2 + a_3 \cdot 2^3 + a_4 \cdot 2^4 \\ &= 1 \cdot 1 + 1 \cdot 2 + 0 \cdot 4 + 0 \cdot 8 + 1 \cdot 16 = 1 + 2 + 16 = 19 \end{aligned}$$

## Schaltnetze oder Kombinatorische Schaltungen

**Schaltnetz (kombinatorische Schaltung, Kombinatorik):**

- Zunächst werden Funktionseinheiten betrachtet, bei denen die Werte der Ausgangssignale direkt aus den Werten der Eingangssignalen berechnet werden können.
- Diese Funktionseinheiten besitzen keine internen Speicher und somit auch kein „Gedächtnis“.
- Die Ausgangssignale hängen von den Werten der Eingangssignalen ab, sind also mathematisch betrachtet eine *Funktion* (oder Abbildung) der Eingangssignalen!



**Für Schaltnetze gilt:**

**Gleiche Eingangswerte führen zu den gleichen Ausgangswerten, unabhängig davon was vorher in der Schaltung passiert ist.**

# Beschreibung von Schaltnetzen mit Tabellen

Der Zusammenhang zwischen den Werten der Ein- und Ausgangssignale kann durch Tabellen dargestellt werden.

Beispiele:

Pegeltabelle		Wahrheitstabellen Logiktabellen	
X1	X2	X1	X2
L	L	0	0
L	H	0	1
H	L	1	0
H	H	1	1

positive Logik      negative Logik

**Pegeltabelle:**

Darstellung der Funktion mittels Logikpegeln

**Wahrheits-/Logiktabellen:** Darstellung der Funktion mittels Logikzuständen

# Beschreibung von Schaltnetzen mit Gleichungen

Rechnen mit Binärsignalen basiert auf der *Schaltalgebra*

→ definiert Operationen auf booleschen Werten und ihre Eigenschaften.

Die *Schaltalgebra* ist ein Spezialfall der *Booleschen Algebra* und dient der Beschreibung digitaler Funktionen:

- Variablen werden über Operatoren verknüpft.
- Es lassen sich Gleichungen aufstellen und umformen.
- Unterschied zur bekannten Algebra: Variablen nehmen nur die Werte 0 und 1 an.

Die Theorie der Booleschen Algebra wurde 1854 von dem Mathematiker George Boole entwickelt. In ihr können Aussagen nur wahr (1) oder falsch (0) sein.

→ Ausführliche Behandlung in Kapitel 4!

# Beschreibung von Schaltnetzen mit Gleichungen

Die grundlegenden Operationen sind die UND- und die ODER-Verknüpfung (Konjunktion und Disjunktion) boolescher Werte sowie die NEGATION.

Operator für die UND-Funktion:  $\wedge$

Operator für die ODER-Funktion:  $\vee$

Operator für die NEGATION:  $\neg$  oder Überstrich über Signalnamen/Teilformel

## Definition der Grundoperationen mit Wahrheitstabellen:

UND		
X	Y	$X \wedge Y$
0	0	0
0	1	0
1	0	0
1	1	1

ODER		
X	Y	$X \vee Y$
0	0	0
0	1	1
1	0	1
1	1	1

NEGATION	
X	$\neg X$
0	1
1	0

Die Eingangsbelegungen werden in den Tabellen meist als aufsteigende Dualzahlen angeordnet!



## Boolesche Ausdrücke oder Terme

### Definition: Boolescher Ausdruck

Sei  $V = \{x_1, x_2, \dots, x_n\}$  eine Menge boolescher Variablen. Dann gilt:

- 0, 1,  $x_i$  sind boolesche Ausdrücke.
- Mit  $\Phi$  und  $\psi$  sind auch,  $\Phi \wedge \psi$ ,  $\Phi \vee \psi$ ,  $\neg \Phi$  und  $(\Phi)$  boolesche Ausdrücke.

Bei booleschen Ausdrücken gilt folgende **Operator-Präzedenz**:

$\neg$  (stärkste Bindung)  $\rightarrow \wedge$  (mittel)  $\rightarrow \vee$  (schwächste Bindung)

Daraus folgt:

- UND wirkt wie eine Klammer!  
*Beispiel:*  $x \vee y \wedge z = x \vee (y \wedge z)$
- Der NEGATIONS-Überstrich negiert das Ergebnis des darunterstehenden Ausdrucks, d. h. er wirkt auch wie eine Klammer.  
*Beispiel:*  $x \vee y = \neg(\neg(x \vee y))$

Achtung: Nach DIN-Norm binden  $\wedge$  und  $\vee$  gleich stark, werden also von links nach rechts zusammengefasst. Dies ist aber in der Praxis unüblich!

# Boolesche Funktionen

## Definition: Boolesche Funktion

Jede Funktion der Form  $y = f(x_1, x_2, \dots, x_n)$  mit  $x_1, x_2, \dots, x_n, y \in \{0, 1\}$  heißt *boolesche Funktion* oder *Schaltfunktion* der *Stelligkeit n*.

**Beispiel:** Schaltnetz mit zwei Eingängen  $X_1, X_2$  und Ausgang Y

$$Y = f(X_1, X_2) = X_1 \wedge (\neg X_2) = X_1 \wedge \bar{X}_2 \text{ (zweistellige Fkt.)}$$

$$f : \{0,1\}^2 \rightarrow \{0,1\}$$

Definitionsreich Wertebereich

$$n \text{ Eingangsvariablen: } f : \{0,1\}^n \rightarrow \{0,1\} \text{ (n-stellige Funktion)}$$

Jeder boolesche Ausdruck entspricht einer booleschen Funktion. Der Ausdruck ist aber nicht eindeutig, da verschiedene Ausdrücke dieselbe Funktion repräsentieren können, d. h. dieselbe Wertetabelle haben!

Die **Grundoperatoren** bilden ein **vollständiges Operatorensystem**, d. h., alle booleschen Funktionen können mit diesen Operatoren dargestellt werden! 

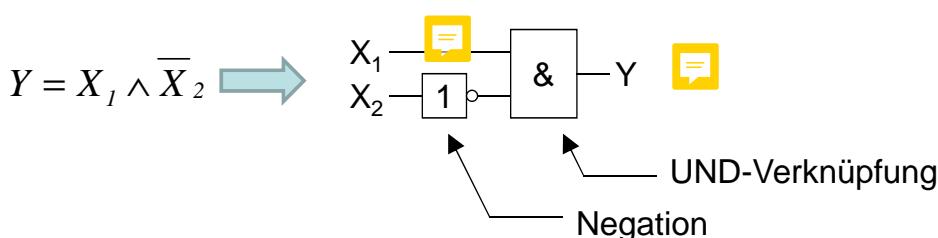
## Beschreibung von Schaltnetzen durch Schaltsymbole

Den Operatoren der booleschen Algebra sind **Schaltsymbole** zugeordnet. Die Hardware-Schaltung eines Operators wird **Gatter** (engl. gate) genannt.

Für Schaltsymbole existieren drei gebräuchliche Standards:

- DIN-Norm 40900, bzw. IEC (neue Norm)
- ANSI bzw. IEEE (amerikanische Norm)
- DIN (alte Norm)

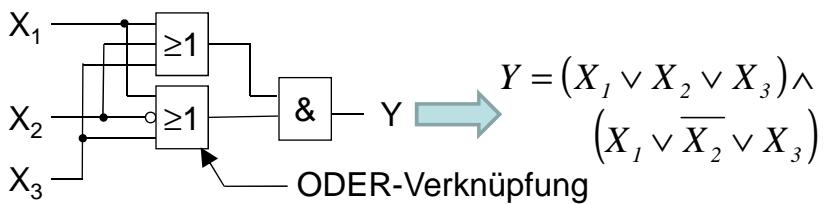
**Beispiel:** Beschreibung eines Schaltnetzes gemäß DIN-Norm 40900



- Das baumförmige Schaltbild ergibt sich direkt aus dem Term der Gleichung
- Signalrichtung meist von links nach rechts, evtl. mit Pfeilen
- Verzweigungen werden durch Punkt an der Schnittstelle gekennzeichnet
- Negation kann mit Eingang oder Ausgang kombiniert werden

# Formeln und Schaltungen

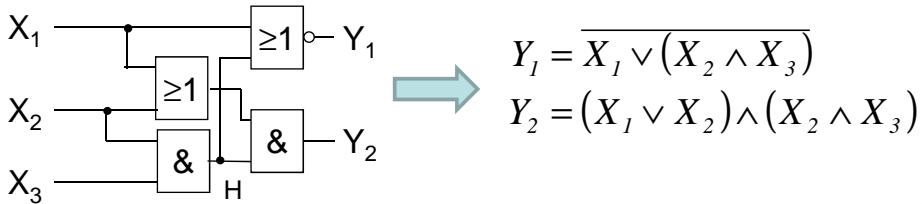
- Ebenso kann aus Schaltbild die Formel extrahiert werden



Achtung:

- Verzweigung eines Signals erlaubt
- Aber Signale dürfen nicht zusammengeführt werden (Kurzschluss!)

- Aber:** Bei nicht baumförmigen Schaltungen („Rekonvergenzen“) oder mehreren Ausgängen müssen Formelteile wiederholt werden (ineffiziente Beschreibung durch Formeln!)



Schwarze Punkte  
bezeichnen  
Leitungsverbin-  
dungen (Knoten-/  
Kontaktpunkte)

- Lösung:** Hilfssignale für „Zwischenergebnisse“ (vgl. Programmieren!)

$$H = (X_2 \wedge X_3)$$

$$Y_1 = \overline{X}_1 \vee H$$

$$Y_2 = (X_1 \vee X_2) \wedge H$$

**Definition Fanout:**

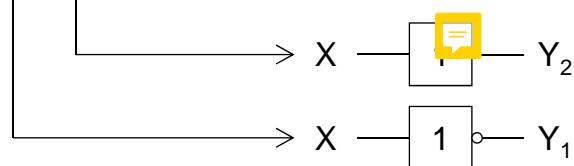
Anzahl der Eingänge, die an einem Ausgang angeschlossen sind (z. B. Fanout=2 für Signal H)

## Funktionen mit einem Eingang

Bezeichnungen:

Konstante 0	
NICHT, Negation, Inversion, Inverter [ $\neg$ ]	
Abbildung, Treiber, Identität, Buffer	
Konstante 1	
X	$Y_0$   $Y_1$   $Y_2$   $Y_3$
0	0   1   0   1
1	0   0   1   1

Schaltsymbole (DIN 40900):



Treiber gibt Wert unverändert weiter, Inverter negiert ihn.

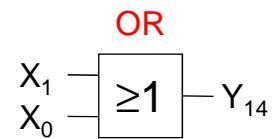
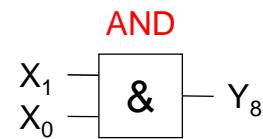
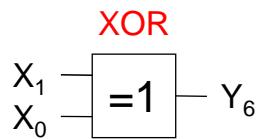
Beide Schaltungen verstärken und „verbessern“ elektrisches Signal (Signalintegrität), aber verzögern es auch!

# Funktionen mit zwei Eingängen

Eingänge:  $X_1, X_0$

Ausgang: Y

		Konstante 0	NOR [ $\bar{v}$ ]		Inhibition		Negation ( $X_1$ ) [ $\neg$ ]		Inhibition		Negation ( $X_0$ ) [ $\neg$ ]		XOR (Antivalenz) [ $\oplus$ ]		UND, AND [ $\wedge$ ]		XNOR (Äquivalenz) [ $\equiv$ ]		Identität ( $X_0$ )		Implikation		Identität ( $X_1$ )		Implikation		ODER, OR [ $\vee$ ]		Konstante 1	
$X_1$	$X_0$	$Y_0$	$Y_1$	$Y_2$	$Y_3$	$Y_4$	$Y_5$	$Y_6$	$Y_7$	$Y_8$	$Y_9$	$Y_{10}$	$Y_{11}$	$Y_{12}$	$Y_{13}$	$Y_{14}$	$Y_{15}$	$Y_{16}$	$Y_{17}$	$Y_{18}$	$Y_{19}$	$Y_{20}$	$Y_{21}$	$Y_{22}$	$Y_{23}$	$Y_{24}$	$Y_{25}$			
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1			
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	0	0	1	0	0	1	1	0	1	1	1			
1	0	0	0	0	0	1	1	1	1	1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1			
1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1			



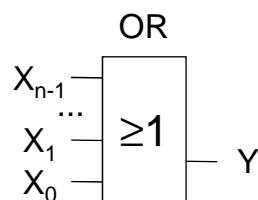
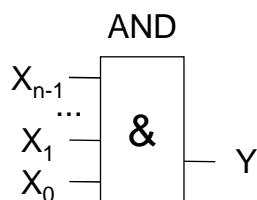
Abgeleitete Funktionen (z. B. XOR) können durch Grundoperationen ersetzt werden!

# Funktionen mit mehr als zwei Eingängen

Boolesche Funktionen mit mehreren Eingängen

$X_{n-1}$	$X_{n-2}$	...	$X_1$	$X_0$	Y	2 <sup>n</sup> Funktionswerte pro Funktion
0	0	...	0	0	$f_0$	$2^{2^n}$ Unterschiedliche Funktionen mit n Eingängen
0	0	...	0	1	$f_1$	$n=1$ 4 Funktionen
...	...	...	...	...	...	$n=2$ 16 Funktionen
1	1	...	1	0	$f_{2^{n-2}}$	$n=3$ 256 Funktionen
1	1	...	1	1	$f_{2^n-1}$	

Schaltsymbole für UND und ODER mit mehr als 2 Eingängen





### **3. Grundlagen digitaler Schaltungen**

**Elektronische Schaltkreise**

**Halbleiter**

**Transistorschaltungen**

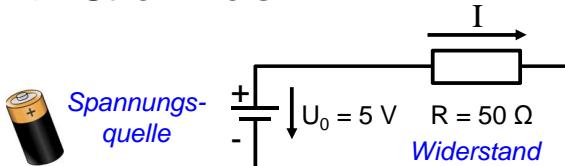
**Chip-Fertigung**

---

**Elektronische Schaltkreise**

# Stromkreis und Schalter

## ► Stromkreis



Achtung:  $R = 0 \Omega$   
→ Kurzschluss!



„Widerstand“ bezeichnet  
im Deutschen physikalische  
Größe *und*  
elektronisches Bauteil!

- Die *elektrische Spannung*  $U$  (Einheit: Volt – V) an einem Punkt eines Schaltkreises gegenüber festem Bezugspunkt heißt *Potential* des Punktes. Bezugspunkt heißt *Masse* ( $\perp$  Potential 0 V).
- Stromstärke*  $I$  wird durch Ohmsches Gesetz bestimmt:  $I = U / R$
- Physikalische Stromrichtung*: Elektronen fließen vom negativen Pol der Spannungsquelle über den Widerstand zum positiven Pol.
- Technische Stromrichtung*: Vom Plus- zum Minuspol (Historische Gründe!)
- Wird als Widerstand z. B. eine Glühbirne eingesetzt, wird sichtbar, ob ein Strom fließt oder nicht.

## ► Schalter

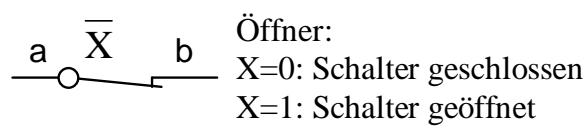
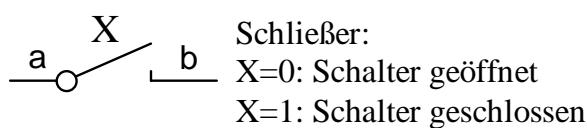


- Durch einen Schalter kann der Stromkreis unterbrochen werden. (Widerstand wechselt zwischen  $0 \Omega$  (geschlossen) und  $\infty \Omega$  (offen))

# Schalter und Schalterlogik

## ► Definition: Schalter

- Bauelement, das abhängig von einem elektrischen oder mechanischen Steuersignal zwei Leitungssegmente verbindet: Zustand *geschlossen/offen*
- Man unterscheidet **Schließer** (bei Steuersignal 0 bzw. nicht gedrückt: offen) und **Öffner** (bei Steuersignal 0/nicht gedrückt: geschlossen)



Kontaktschaltungen zur Realisierung logischer Funktionen → Schalterlogik:

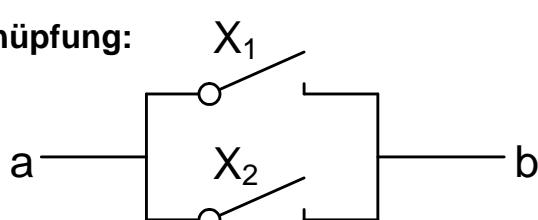
### Ergebnis der Schaltfunktion:

- wenn eine Verbindung zwischen a und b besteht
- sonst

## UND-Verknüpfung:



## ODER-Verknüpfung:

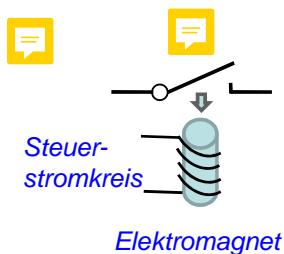


# Schalter und Schalterlogik

## ► Elektrische Schalter (der Vor-Transistor-Ära)

- Relais

Magnet steuert Schalter

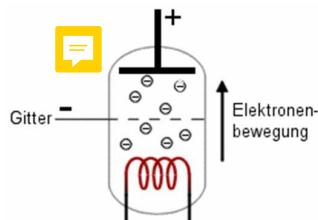


Nachteile:

- langsam
- hohe Verlustleistung
- störanfällig

- Röhrentriode (auch Verstärker!)

Spannung am Gitter steuert Strom von Katode (Glühwendel) zu Anode (+)



Nachteile:

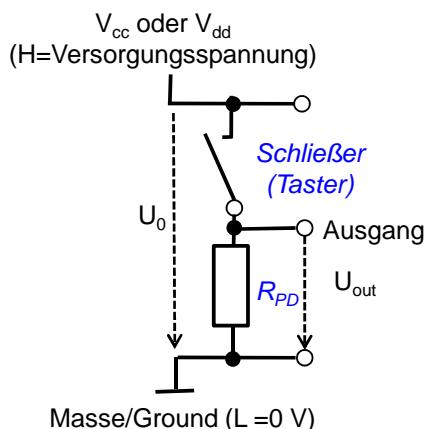
- hohe Verlustleistung
- störanfällig

Quelle: elektronikinfo.de

Claude E. Shannon zeigte um 1940, dass die Eigenschaften von Serien- und Parallelschaltungen von Schaltern und Relais gut mit der Schaltalgebra beschrieben werden können. ➔ Kapitel 5 (Boolesche Algebra)

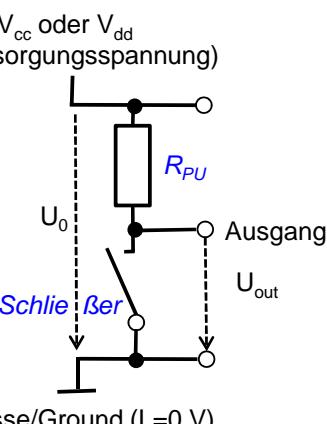
## Taster zur Eingabe von Logikpegeln

Anforderung: Logikpegel am Ausgang (= Eingang der folgenden Schaltung) muss immer definiert sein ➔ Spannungsteiler; ein Taster ersetzt einen Widerstand



Statt Spannungsquelle H- und L-Verbindungen!  
H:  $V_{cc}/V_{dd}$  oder ↑  
L: ⊥ oder ↓

Nachteil dieser Schaltungen?



### Nichtinvertierende Taste

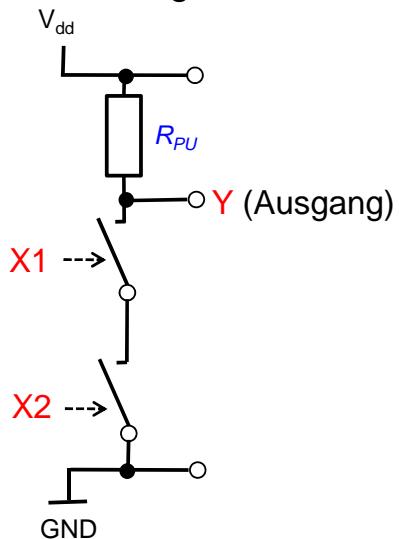
- Schalter offen: Ausgang liegt defaultmäßig auf Masse (L)  
→  $R_{PD}$  heisst Pull-Down-Widerstand
- Schalter geschlossen: Ausgang mit H ( $V_{cc}/V_{dd}$ ) verbunden,  $U_0$  fällt an R ab

### Invertierende Taste

- Schalter offen: Ausgang liegt defaultmäßig auf  $V_{cc}$  (H)  
→  $R_{PU}$  heisst Pull-Up-Widerstand
- Schalter geschlossen: Ausgang mit L (Masse) verbunden,  $U_0$  fällt an R ab

# Realisierung einfacher Logik-Gatter

- Wenn wir bei der invertierenden Taste einen *elektrischen Schalter* verwenden, haben wir schon einen **Inverter**!
- **NAND-Gatter** mit 2 (oder mehr) Eingängen können durch Serien-Schaltung mehrerer Schalter realisiert werden



Bei positiver Logik ergibt sich folgendes:

Wenn beide Taster geschlossen sind, liegt am Ausgang eine niedrige Spannung an, also logisch 0.

Andernfalls wird der Ausgang über den Widerstand auf eine hohe Spannung, also logisch 1, gezogen.

## Implementierung in Halbleiter-Technologie:

- Schalter werden durch *Transistoren* realisiert.
- Leistungsaufnahme durch Widerstand kann in *CMOS-Schaltungen* vermieden werden!

- **NOR-Gatter:** Analog mit Parallelschaltung der Schalter

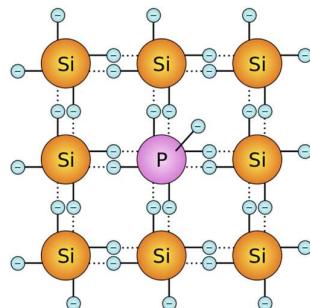
# Halbleiter



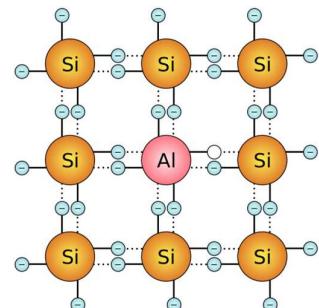
# Halbleitermaterialien

- ▶ **Reiner Halbleiterkristall - Silizium (Si) oder Germanium (Ge)**
  - Schlechte Leitfähigkeit (zwischen Leitern und Nichtleitern)
  - Alle Elektronen im Kristallgitter eingebunden → kaum beweglich
- ▶ **Dotierter Halbleiter**
  - Gezielte Zugabe anderer Elemente mit anderer Elektronenzahl
  - Mehr Elektronen als Si, z. B. Phosphor (P)
    - zusätzliche Elektronen (Überschuss) führen zu einem *n-Leiter*
  - Weniger Elektronen als Si, z. B. Aluminium (Al) – „Defektelektronen“
    - künstliche Löcher im Kristallgitter (Elektronenmangel) ergeben *p-Leiter*

**n-Dotierung:**  
zusätzliche Elektronen können sich frei bewegen → höhere Leitfähigkeit



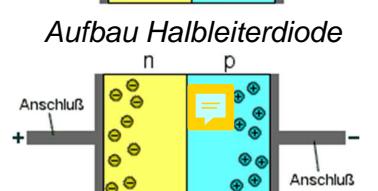
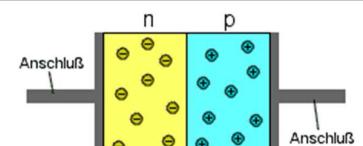
**p-Dotierung:**  
„Löcher“ können sich bewegen, in dem Elektronen von Loch zu Loch springen → höhere Leitfähigkeit



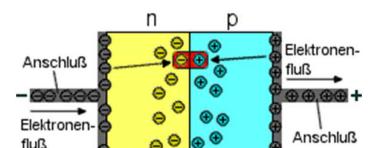
Quelle: Wikipedia

## Halbleiterdiode

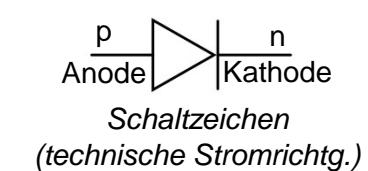
- ▶ **Diode:** begrenzt Stromfluss richtungsabhängig
  - Halbleiterdiode besteht aus p-dotiertem und n-dotiertem Halbleiter.
  - Am pn-Übergang füllen einige Elektronen des n-Leiters die Löcher des p-Leiters, bis die ungleiche Ladungsverteilung weitere Verschiebungen verhindert. → Entstehung einer Sperrschicht (wenig leitfähig, keine freien Ladungsträger)
- ▶ **Anlegen einer Spannung in Sperrrichtung**
  - Minuspol an p-Schicht, Pluspol an n-Schicht
  - Elektronen werden vom Pluspol und Löcher vom Minuspol angezogen
  - Sperrschicht wird größer, Diode sperrt!
- ▶ **Anlegen einer Spannung in Durchlassrichtung**
  - Bei umgekehrter Polung drängen Elektronen in n-Leiter und Defektelektronen in p-Leiter.
  - Sperrschicht wird kleiner, Elektronen und Löcher rekombinieren. Nach Übersteigen einer Spannungsschwelle fließt Strom fast ungehindert!



Spannung in Sperrrichtung



Spannung in Durchlassrichtung



Quelle: elektronikinfo.de

10

# Transistor - Bipolartransistor

## ► Transistor (Kunstwort aus *Transfer* & *Resistor* = „steuerbarer Widerstand“)

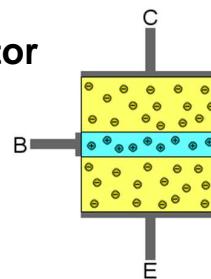
- Funktion wie Röhrentriode: Kann Signal verstärken oder (in digitalen Schaltungen) ein- oder ausschalten → Realisierung von Logik-Gattern!

## ► Ältester Transistortyp: Bipolartransistor

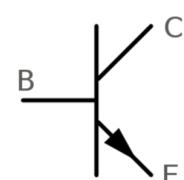
- oder Sperrsichttransistor
- Stromgesteuerter Schalter
  - Aufbau aus zwei Dioden, Schichtanordnung *npn* oder *pnp*
  - Anschlüsse: Basis B (schwach p-dotiert) Kollektor C, Emitter E (n-dotiert)
  - Stromfluss durch B (Steueranschluß!) steuert Strom zwischen C und E

### • Sperrbetrieb

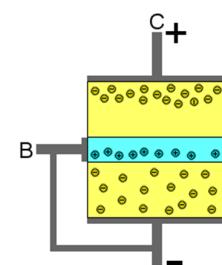
- Betriebsspannung zwischen C (+) und E (-) angelegt
- Wird B auf 0 V (Masse) gelegt, fließt kein Strom B→E  
→ C-B-Diode ist in Sperrrichtung, kein Strom C→E!  
→ **Transistor sperrt!**



*npn-Transistor*



Schaltzeichen (*npn*)

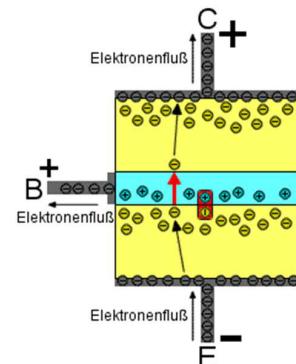


Gesperrter  
*npn-Transistor*

# Bipolartransistor

### • Leitung

- Niedrige positive Spannung zwischen B und E  
→ B-E-Diode in Flussrichtung, B-E-Strom fließt!
- Aus der stark dotierten n-Schicht gelangen mehr Elektronen in die p-Schicht, als die Basis absorbiert kann.
- Also gelangen einige Elektronen durch die dünne p-Schicht in die obere n-Schicht und damit in den Sog des Kollektors  
→ Es fließt ein Strom von C nach E!
- Bei einem ausreichend großen (aber immer noch im Bereich  $\mu\text{A}$ ) Strom durch die Basis schaltet der Transistor durch und zwischen Kollektor und Emitter fließt Strom.  
(Stromverstärkung bis zum Faktor 1000!)



*Leitender npn-Transistor*

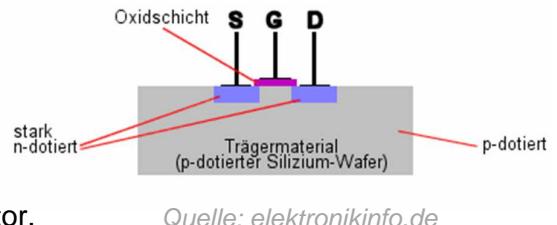
### • Analog funktionieren pnp-Transistoren

In Digitalschaltungen werden Transistoren als schnelle, verschleißfreie und stromsparende Schalter verwendet!

Quelle: elektronikinfo.de

# Feldeffekttransistor (Unipolartransistor)

- ▶ **Stromsparender Transistortyp: Feldeffekttransistor (FET)**
  - Spannungsgesteuerter Widerstand (oder Schalter)
  - Es gibt wieder zwei Polungsvarianten: n-Kanal-FET und p-Kanal-FET
  - *Nachteil:* Anfällig gegenüber statischer Aufladung!
- ▶ **Bauart 1: Sperrsichttransistor/Junction FET (JFET)**
  - diskrete Bauelemente, vergleichbar zu Bipolaren Transistoren
- ▶ **Bauart 2: Metalloxid-FET (MOSFET)**
  - MOSFETs können in planarer (flacher) Bauweise direkt auf einem Si-Kristall aufgebaut werden
  - *Aufbau n-Kanal-MOSFET:*
    - Steueranschluß G (Gate) ist durch eine dünne Isolatorschicht (Metalloxid) vom Trägermaterial (Substrat) getrennt, bildet also einen mikroskopisch kleinen Kondensator.
    - G steuert den Widerstand zwischen Drain (D) und Source (S).
    - Im p-dotierten Substrat befinden sich zwei stark n-dotierte Gebiete (D-S-Anschluss)
    - Source liegt in der Regel auf gleichem Potential wie das Substrat.
    - Bei fehlender G-S-Spannung ist der Widerstand zwischen D und S sehr hoch, der Transistor sperrt. (Eine der beiden Dioden des npn-Übergangs sperrt immer!)



Quelle: elektronikinfo.de

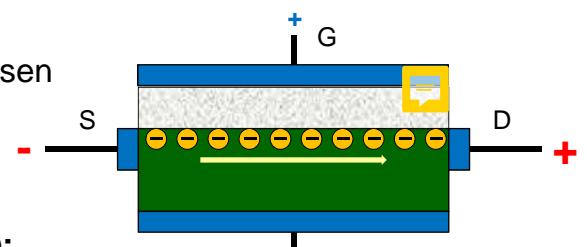
## n-Kanal-MOSFET-Transistor (nMOS): Das Grundprinzip

### Unbeschalteter Zustand:

Hoher Widerstand zwischen den Anschlüssen

Drain (D) und Source (S)

→ Kein Stromfluss zwischen D und S



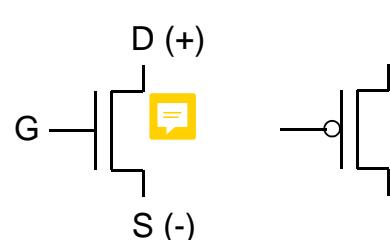
### Positive Spannung an Anschluss Gate (G):

- Elektronen werden von dem positiven Anschluß G angezogen, füllen die Löcher im p-Gebiet und sammeln sich am Isolator. → Es entsteht ein n-Kanal!
- Durch Spannung zwischen D und S können die Elektronen vom linken n-Gebiet über den n-Kanal zum rechten n-Gebiet in Bewegung versetzt werden  
→ Strom von D nach S!
- Bei ausreichend großer Gate-Source Spannung schaltet der Transistor also durch und der Widerstand ist sehr gering.

### Analog funktioniert der p-Kanal-MOSFET

Negative Spannung an G schaltet pMOS durch!

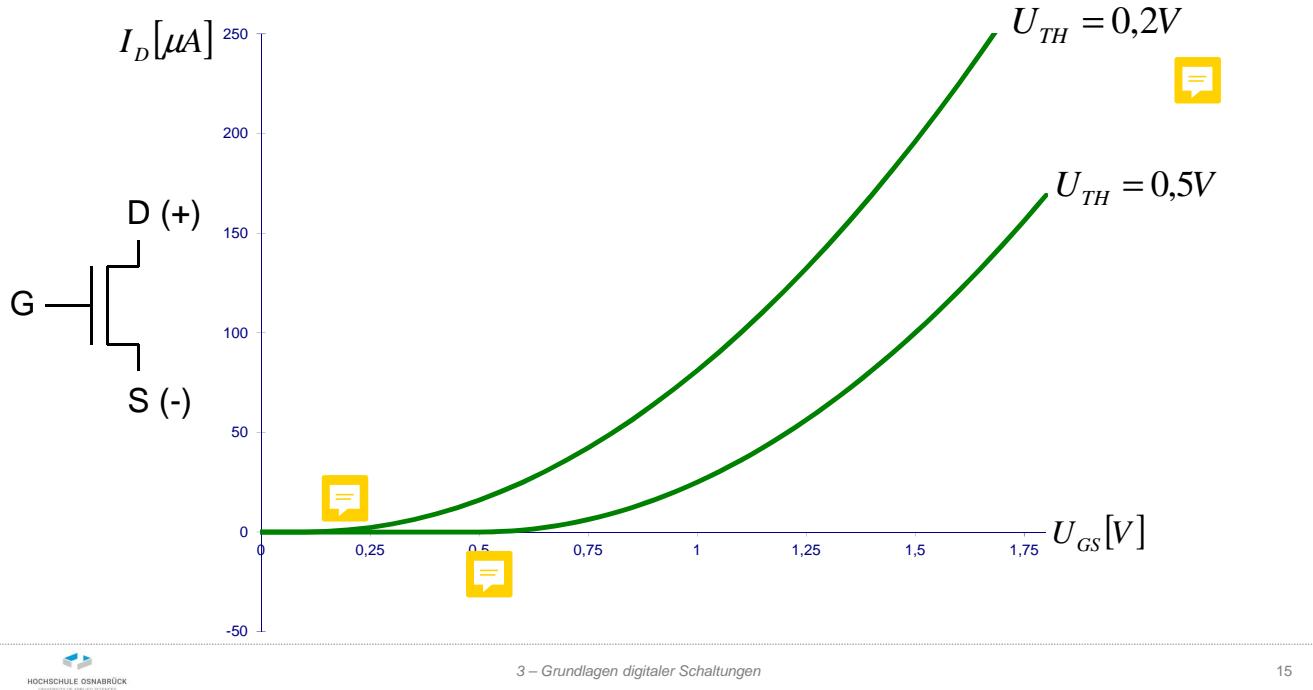
Symbol nMOS:      Symbol pMOS:



# MOS-Transistor Eingangskennlinie

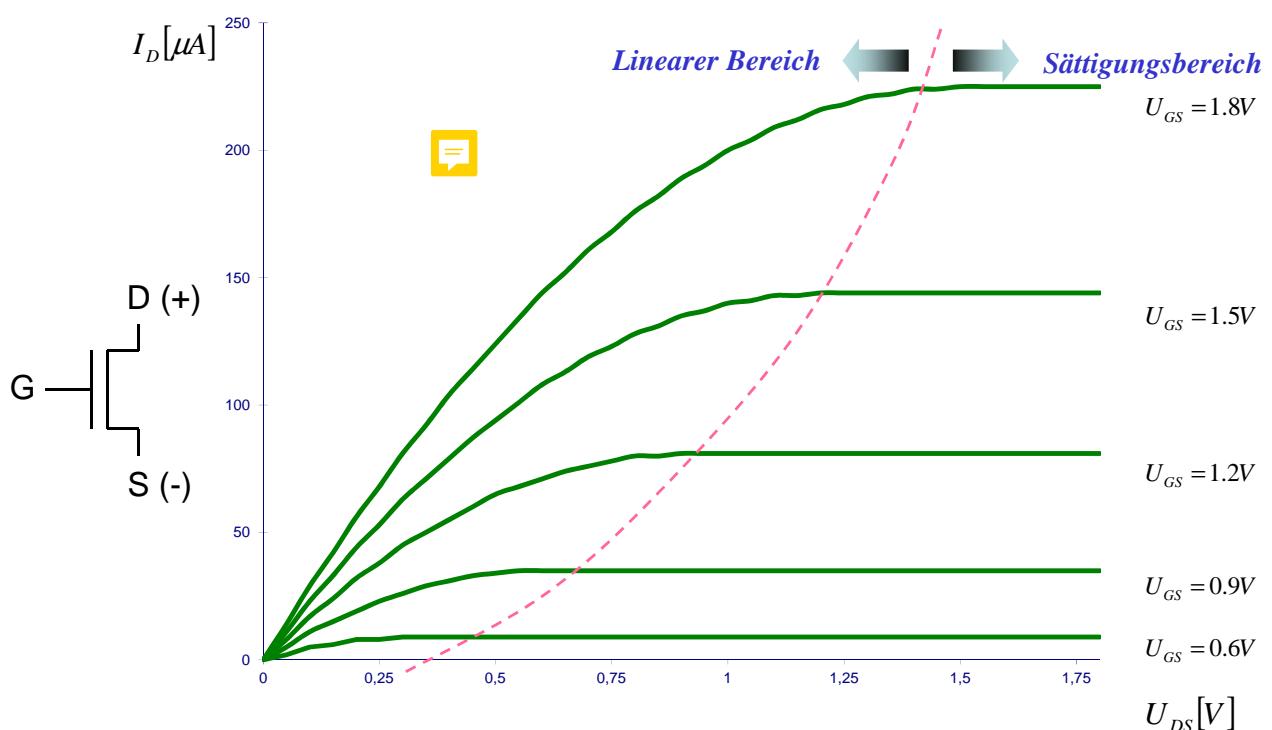
- Liegt die Gate-Source-Spannung unterhalb einer transistorspezifischen Schwellspannung  $U_{TH}$ , fließt kein Strom zwischen den Drain- und Source-Gebieten

Beispiel: nMOS,  $U_{DS} = 1,8V$   $W/L = 3$   $\beta = 200 \mu A/V^2$

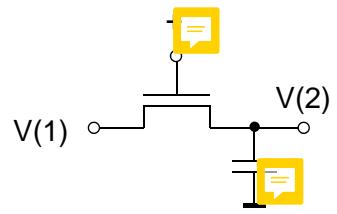
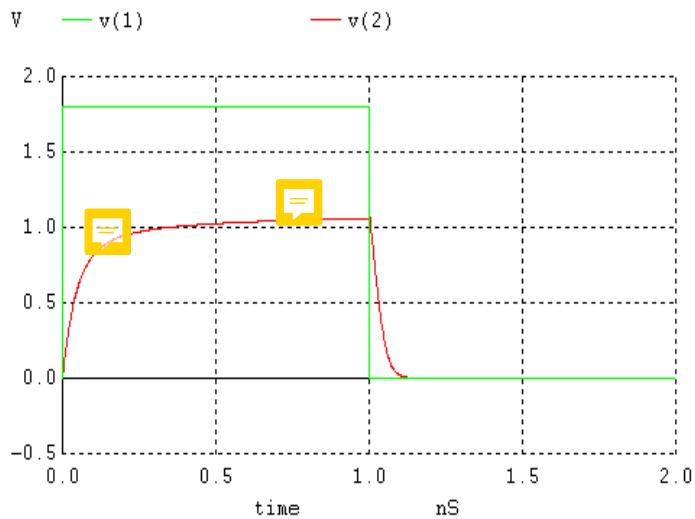


# MOS-Transistor Ausgangskennlinie

Beispiel: nMOS,  $W/L = 3$ ,  $\beta = 200 \mu A/V^2$



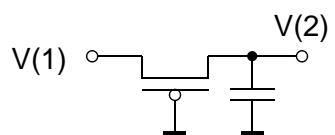
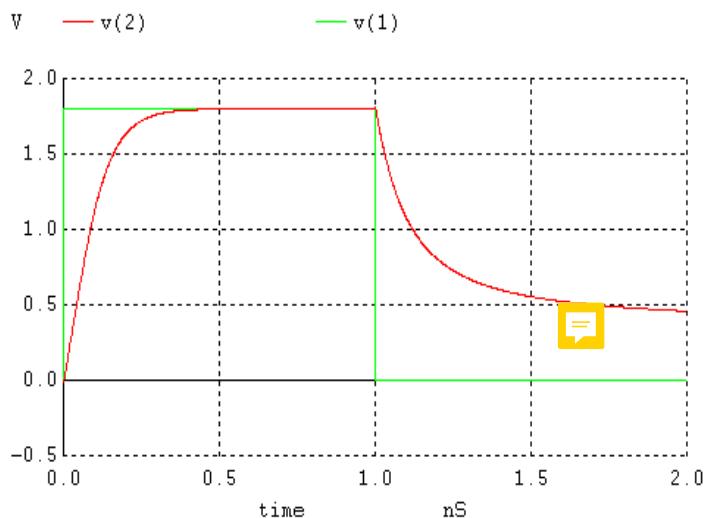
# Der nMOS-Transistor als Schalter



## Signaltransfer:

niedrige Spannung („Low“) = gut  
hohe Spannung („High“) = nicht gut

# Der pMOS-Transistor als Schalter



## Signaltransfer:

niedrige Spannung („Low“) = nicht gut  
hohe Spannung („High“) = gut

---

# Transistorschaltungen

---

## Schaltungsfamilien

Abhängig von den verwendeten Transistor-Familien gibt es unterschiedliche Methoden, Logik-Gatter aufzubauen:

► **Bipolar-Transistoren**

- TTL-Technik (Transistor-Transistor-Logic)
- ECL-Technik (Emitter-Coupled-Logic)
- sowie weitere Varianten, die jedoch keine große Relevanz mehr haben

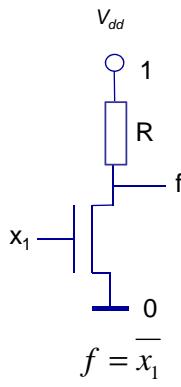
► **MOSFET-Transistoren**

- NMOS-Schaltungen: Verwenden nur n-Kanal-MOSFETs und Widerstände
- CMOS (Complementary MOS): Verwenden n-Kanal- und p-Kanal-MOSFETs

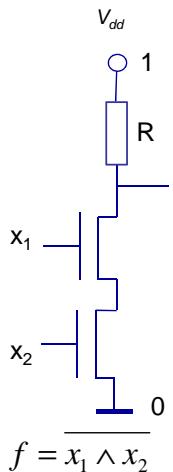
## NMOS-Technologie

- Aufbau der Gatter aus parallel oder in Reihe geschalteten n-Kanal-MOSFET-Transistoren (im *Pull-Down-Pfad*) und Pull-Up-Widerstand R (R sorgt für  $f = V_{dd}$  wenn nMOS-Schaltung offen, sonst  $f = 0 \text{ V}$ )

**Beispiel Inverter:**



**Beispiel NAND:**



Zur Erinnerung:  
nMOS überträgt  
LOW-Signal gut!

n-Kanal-MOSFET-  
Transistor (nMOS)  
entspricht einem  
Schließer!

- **Nachteil:**
  - Stromverbrauch im 0-Zustand (Strom durch R)
  - Schlechtes Schaltverhalten

## CMOS - Technologie

Weit über 95% der integrierten Schaltkreise werden heute in „CMOS-Technologie“ produziert!

CMOS: Complementary Metal Oxide Semiconductor

**Idee:**

Ersetze Pull-Up-Widerstand der NMOS-Schaltung durch komplementäre (d. h. invertierte) Schaltung aus p-Kanal-MOSFET-Transistoren (pMOS).

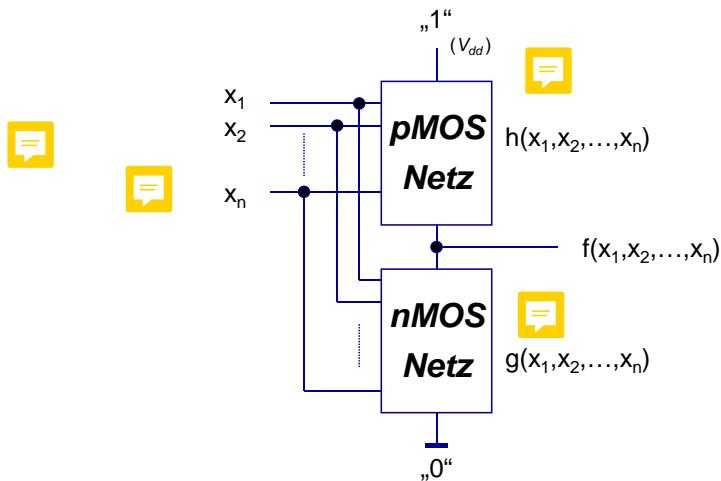
→ Der Pull-Up-Pfad stellt genau dann eine Verbindung zu  $V_{dd}$  her, wenn der Pull-Down-Pfad offen ist (und umgekehrt).

p-Kanal-MOSFET-Transistor (pMOS) entspricht einem Öffner.

CMOS kombiniert „Schließer“ und „Öffner“ zur Realisierung logischer Funktionen!

# CMOS-Gatterlogik

## Aufbau eines CMOS-Gatters



Funktion des pMOS-Netzes ist gleich der Funktion des Gatters:

$$h(X) = f(X)$$

Funktion des nMOS-Netzes ist komplementär zu der des pMOS-Netzes:

$$g(X) = \bar{h}(X) = \bar{f}(X)$$

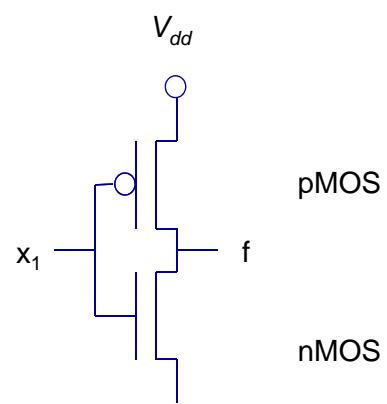
### Komplementbildung mit Schalterlogik:

- |                   |   |
|-------------------|---|
| Reihenschaltung   | ⇒ Parallelschaltung   |
| Parallelschaltung | ⇒ Reihenschaltung   |
| Schalter          | ⇒ Komplementärschalter      (z.B.: nMOS-Trans. ⇒ pMOS-Trans.) |

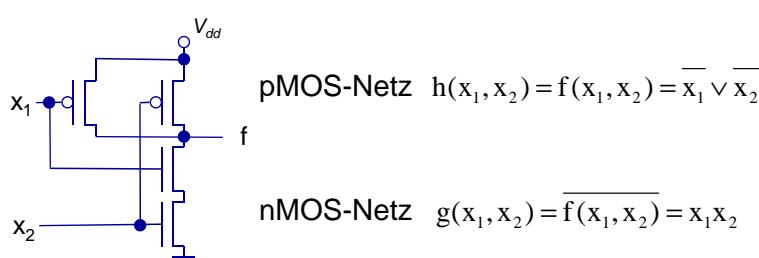
## CMOS - Beispiele

**Inverter:**  $f(x_1) = \bar{x}_1$

- ▶ **Niedrige Spannung am Eingang / logische „0“.**
  - Nur die p-Kanal-Komponente leitet
  - Die Versorgungsspannung ist mit dem Ausgang verbunden
  - Am Ausgang liegt die logische „1“ an!
- ▶ **Höhere Spannung am Eingang / logische „1“**
  - Nur die n-Kanal-Komponente leitet
  - Die Masse ist mit dem Ausgang verbunden
  - Am Ausgang liegt die logische „0“ an!



**NAND mit 2 Eingängen:**  $f(x_1, x_2) = \overline{x_1 x_2} = \overline{x_1} \vee \overline{x_2}$



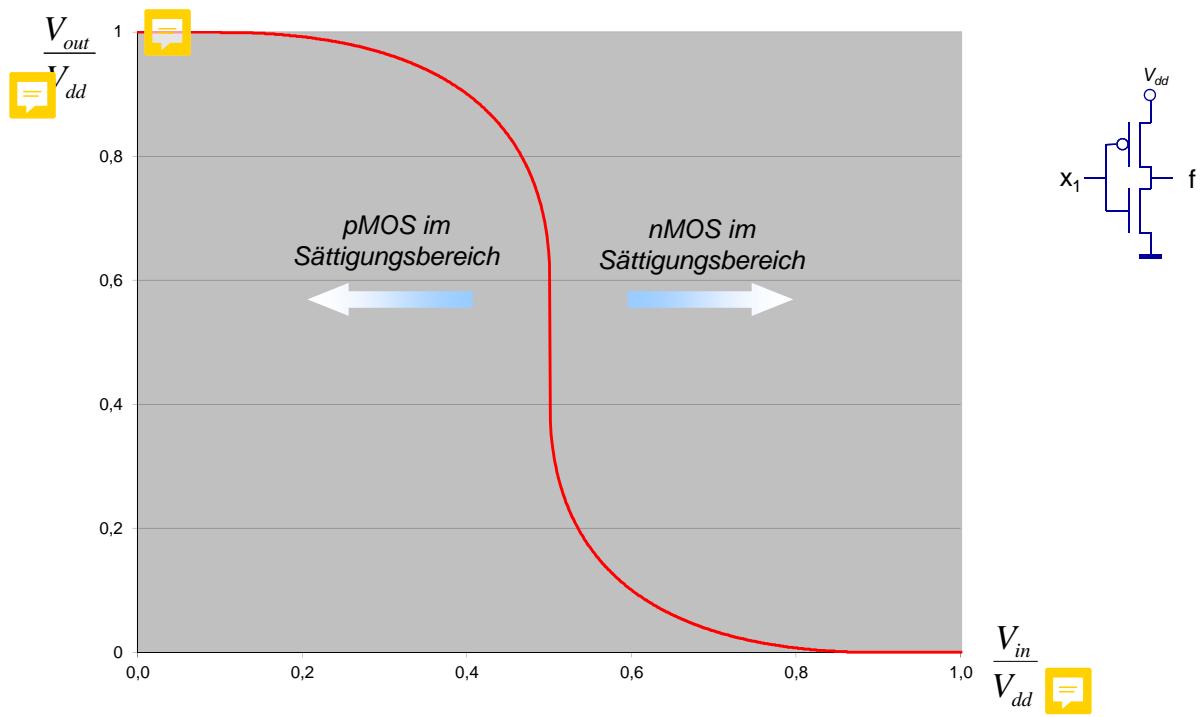
### Warum NAND?

Vertauschen von  $V_{dd}$  und  $\perp$  ergäbe doch ein AND –Gatter!

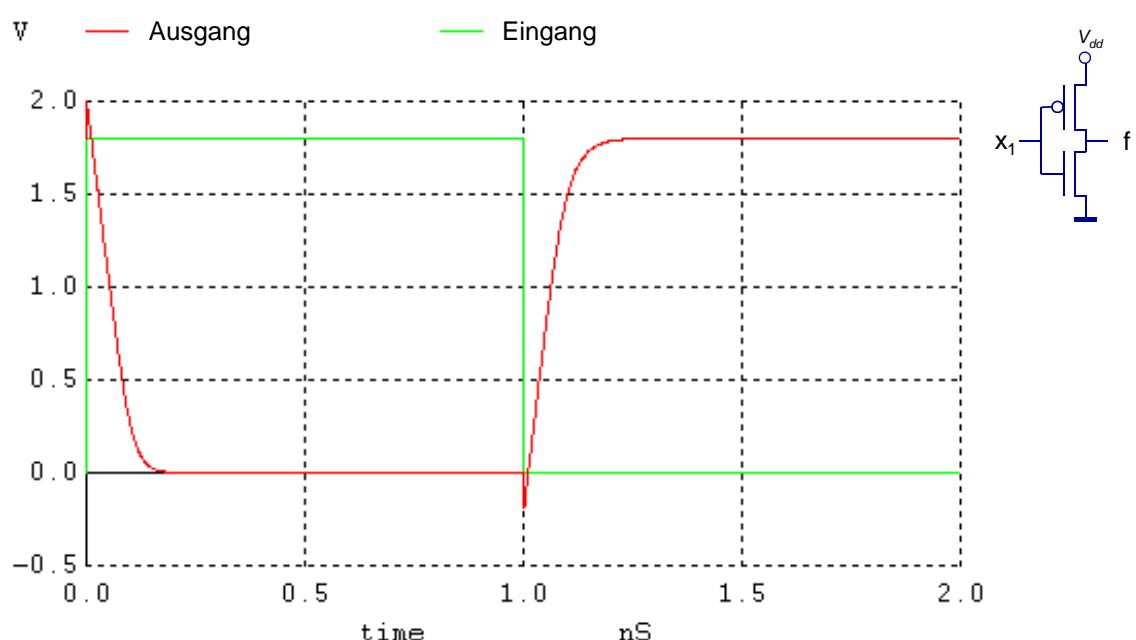
Nein! Denn pMOS-FET überträgt  $\perp$  nicht gut, und nMOS-FET überträgt  $V_{dd}$  nicht gut!

→ Grundelemente sind NAND (und analog NOR)!

# Kennlinie des CMOS-Inverters



## Zeitverhalten eines Inverters



# Verlustleistung bei CMOS-Gattern

## Ein ideales CMOS-Gatter verbraucht keine Verlustleistung

Über das Gate fließt kein Strom, da es hochohmig angeschlossen ist

Da der Ausgang nur an Gates angeschlossen ist, fließt kein Strom über den Ausgang zu den Eingängen nachfolgender Gatter

Da immer nur entweder die Versorgungsspannung (über p-Kanal-Transistoren) oder Masse (über n-Kanal-Transistoren) auf den Ausgang geschaltet wird, kann kein Querstrom fließen

(Querstrom: Strom, der „quer“ durch das CMOS-Gatter fließt  nicht über den Ausgang)

**Und wieso hat dann zum Beispiel eine CPU im PC einen Lüfter ?**

**Weil die Realität nicht ganz so ideal wie oben beschrieben ist...**

**Es gibt drei wesentliche Komponenten, die zur Verlustleistung von CMOS-Schaltungen beitragen**

Querströme

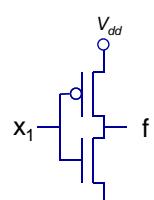
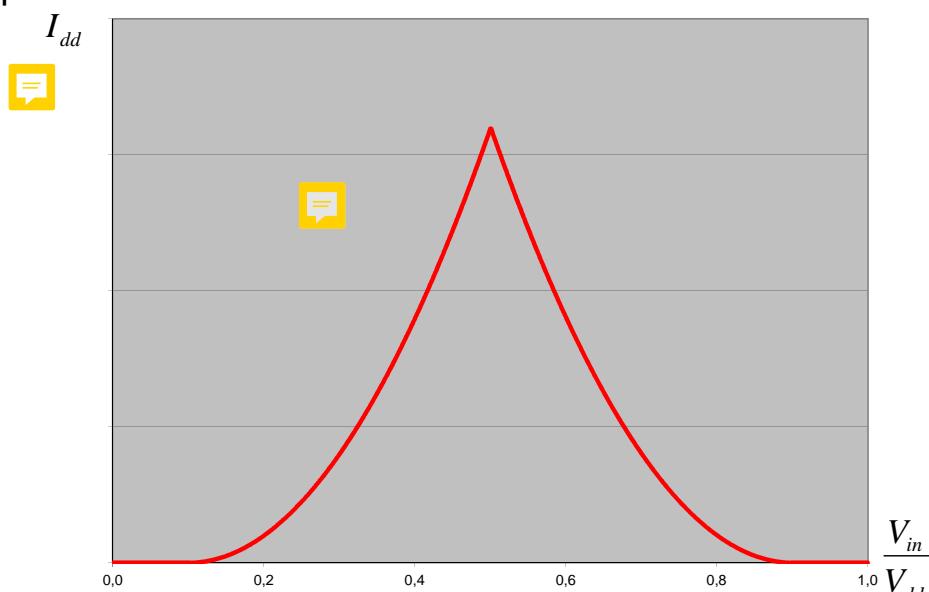
Ströme zum Umladen parasitärer Kapazitäten

Leckströme

## Querströme

Während des Schaltvorgangs sind sowohl p-Kanal- als auch n-Kanal-Transistoren durchgeschaltet

Beispiel: Querstrom bei einem Inverter



# Parasitäre Effekte

## Kapazitäten

Die Gates der Transistoren nachfolgender Logik-Gatter besitzen Kapazitäten  
Die Verbindungsleitungen besitzen Kapazitäten

Diese Kapazitäten müssen während des Schaltens von 0 auf 1  
(oder umgekehrt) umgeladen werden

## Leckströme

Über die Transistoren fließt ein geringer Strom von der Versorgungsspannung gegen Masse (auch wenn nicht geschaltet wird)  
 $\sim nA$  pro Transistor

Die Leckströme können sich bei hochintegrierten Schaltungen zu signifikanten Werten aufsummieren

Heutige Prozessoren:

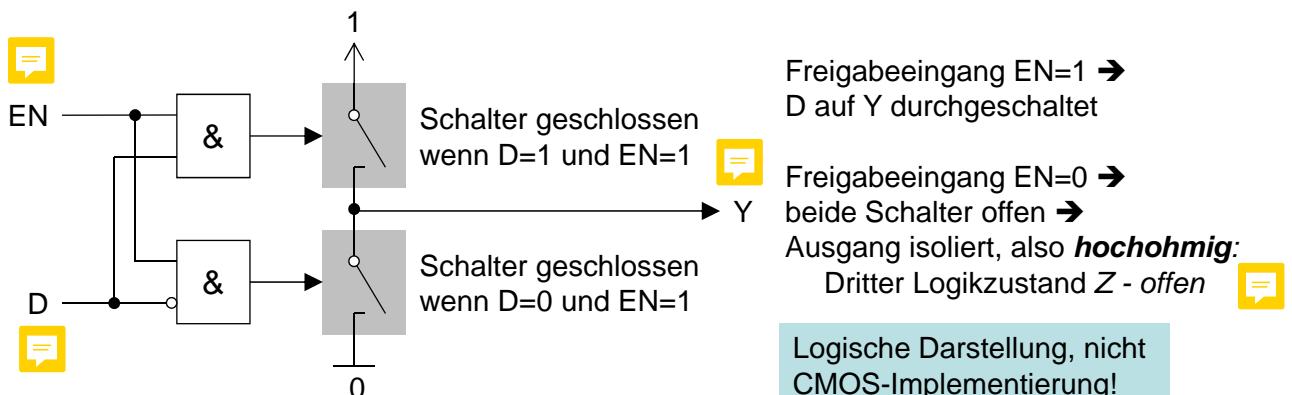
Annähernd die Hälfte der Verlustleistung entsteht durch Leckströme

# Ausgänge logischer Schaltungen



- **Push-Pull-Ausgang** treibt das Ausgangssignal immer aktiv auf einen der zwei Zustände: 1 ( $V_{dd}$ ) oder 0 ( $\perp$ )  
→ Zusammenschalten zweier Ausgänge kann fatalen Kurzschluß bewirken!
- **Tri-State-Ausgang** hat einen dritten, hochohmigen Zustand  $Z$  („dreiwertige Logik“):
  - Ausgang ist „offen“, d. h. sowohl Pull-Up- als auch Pull-Down-Pfad sind getrennt (hochohmig)
  - Ansteuerung über *Enable*- oder *Freigabeeingang* EN

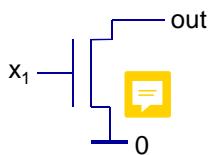
Besteht eine Schaltung nur aus einem Tri-State-Ausgang, spricht man von einem **Tri-State-Treiber**:



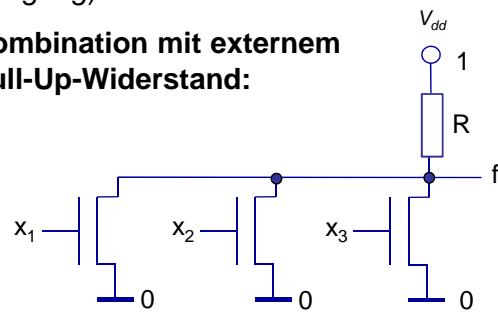
# Open-Drain-Ausgang

- Open-Drain-Ausgang ist eine unvollständige Schaltung!
  - ➔ Nur Pull-Down-Pfad; Pull-Up-Pfad oder -Widerstand fehlt, der Drain-Anschluß des MOS-FET ist also offen!
  - (analog in TTL-Technologie: Open-Collector-Ausgang)

**Beispiel:**  
Inverter mit Open-Drain-Ausgang



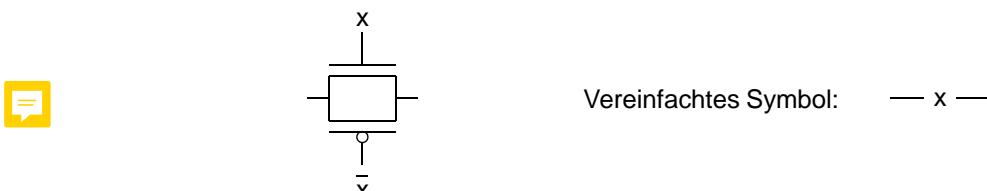
**Kombination mit externem Pull-Up-Widerstand:**



- Schaltung muss durch externen Pull-Up-Widerstand vervollständigt werden
- Verbindung mehrerer Ausgänge mit gemeinsamem Pull-Up-Widerstand möglich
  - Ergibt NOR-Verknüpfung: (man spricht auch von *wired-OR/-AND*)
    - alle Eingänge auf 0 ➔ Pull-Up-Widerstand zieht Ausgang auf 1
    - ein oder mehrere Eingänge auf 1 ➔ Ausgang wird auf 0 gezogen
- Ermöglicht Zusammenführen mehrerer Ausgänge ohne Kurzschluß-Gefahr!  
(Verbindung auch chip-extern auf der Platine möglich)

## Sonderfall: Transmission Gates

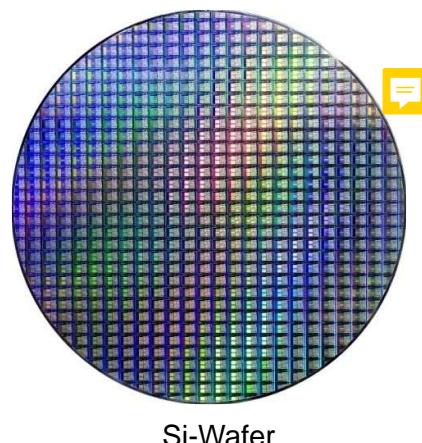
- Neben der Verwendung von nMOS- und pMOS-Transistoren in Logik-Gattern werden MOSFETs auch zum gesteuerten Verbinden *bidirekionaler* Leitungs-Segmente verwendet  
→ in Speicherelementen und FPGAs, siehe Kap. 8
- Dazu eignet sich ein aus einem nMOS und einem pMOS kombiniertes **Transmission Gate** (= „Übertragungs-Gatter“)



- Bei  $x = 0$  ist das Transmission Gate offen; bei  $x = 1$  sind die Leitungssegmente links und rechts verbunden.
- Die beiden MOSFETs realisieren dieselbe Schalt-Funktion, ermöglichen durch ihre unterschiedlichen Transfer-Eigenschaften aber sowohl eine gute Übertragung des L-Pegels als auch des H-Pegels, also einen idealen Schalter.

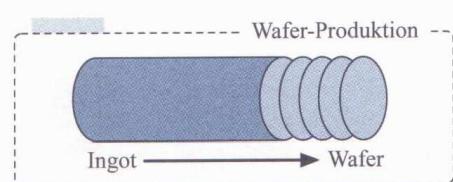


## Chip-Fertigung



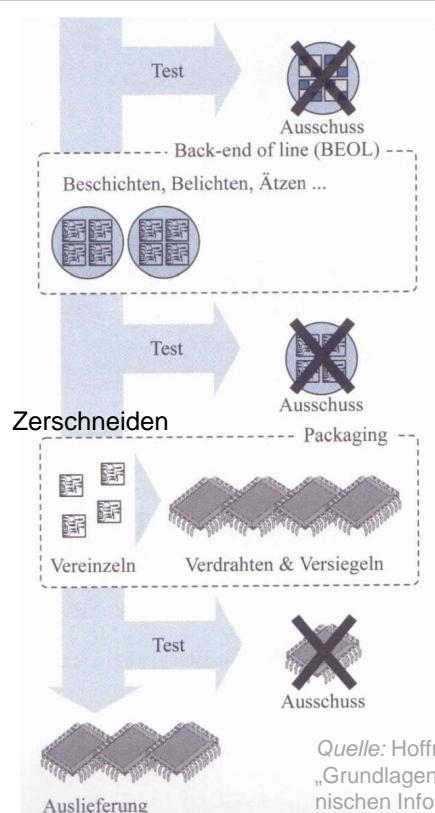
## Chip-Fertigung – Übersicht

Aus hochreinem Silizium wird ein zylindrischer Einkristall (Ingot) gezogen und in Wafer zersägt (Durchmesser: 300-450 mm)



**Waferdicke:**  
0,5 – 1,5 mm  
(Substrat)

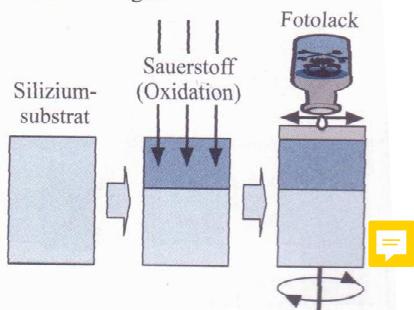
Viele gleiche  
Chips werden  
auf einem Wa-  
fer produziert



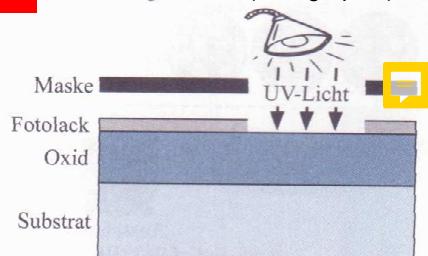
# FEOL - Basistechnologien

- Oxidation im Ofen bei 1000° C
- Auftragen von lichtempfindlichem Fotolack (dünne Schicht)

## 1 Beschichtungstechnik



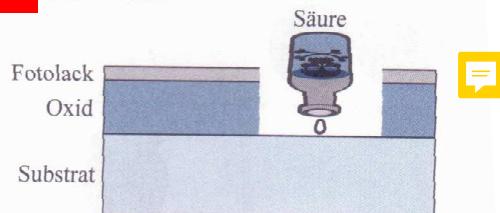
## 2 Belichtungstechnik (Lithographie)



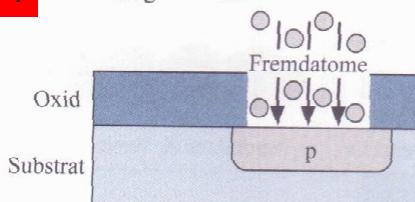
- Lackhärtung mit kurzwelligem UV-Licht
- belichtete Stellen werden weich/herausgelöst

- Mit Säure wird Oxidschicht an freiliegenden Stellen entfernt
- Danach wird Lack gründlich entfernt

## 3 Ätztechnik



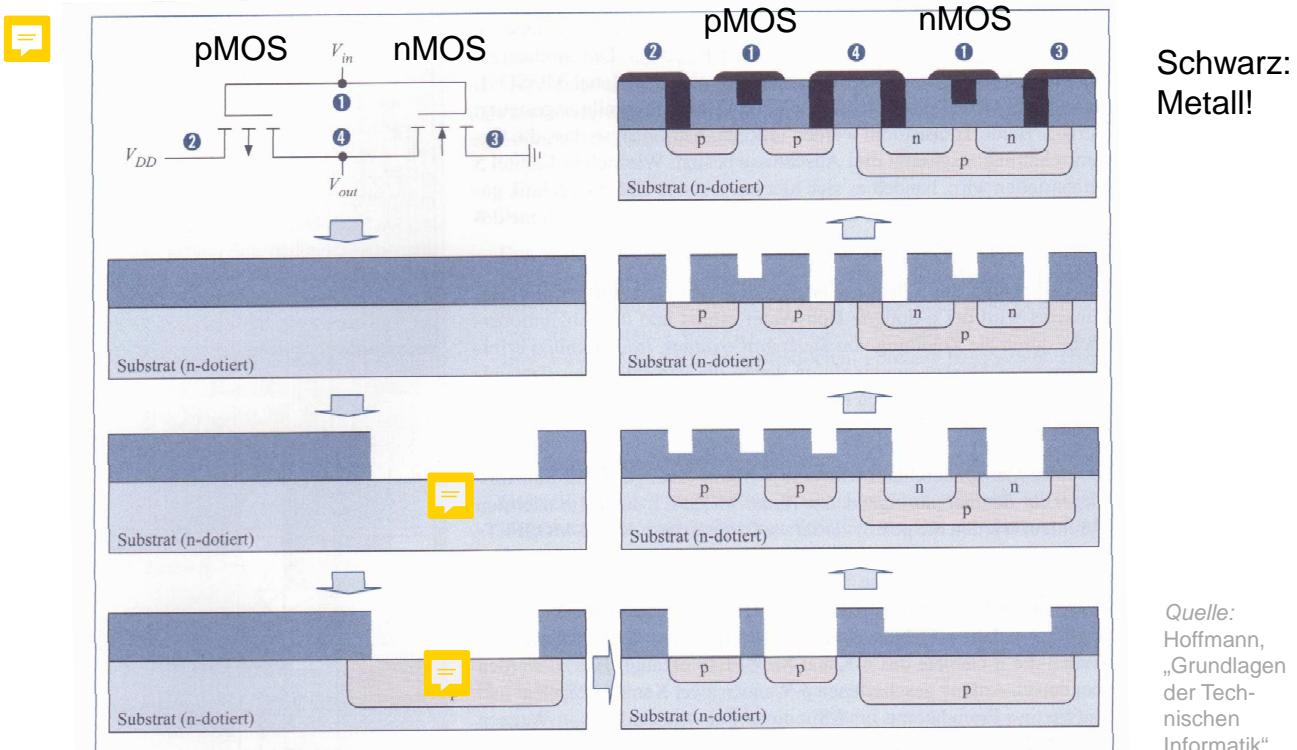
## 4 Dotierungstechnik



Quelle: Hoffmann, „Grundlagen der Technischen Informatik“

- Wafer wird erhitzt, und zugängliche Stellen mit Fremdatomen versetzt (z. B. durch Ionenbeschuss, Diffusion aus Dotiergas/-lack)

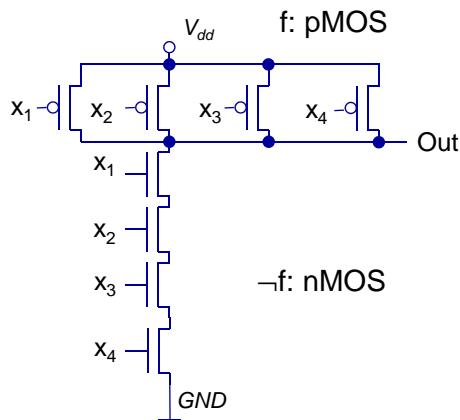
# Chip-Fertigung – Schaltung in Planartechnik



**Strukturbreite:** Breite des Kanals/Abstand von D und S (Intel 4004: 10 µm, heute ~14 nm)

# Layout von CMOS-Gattern

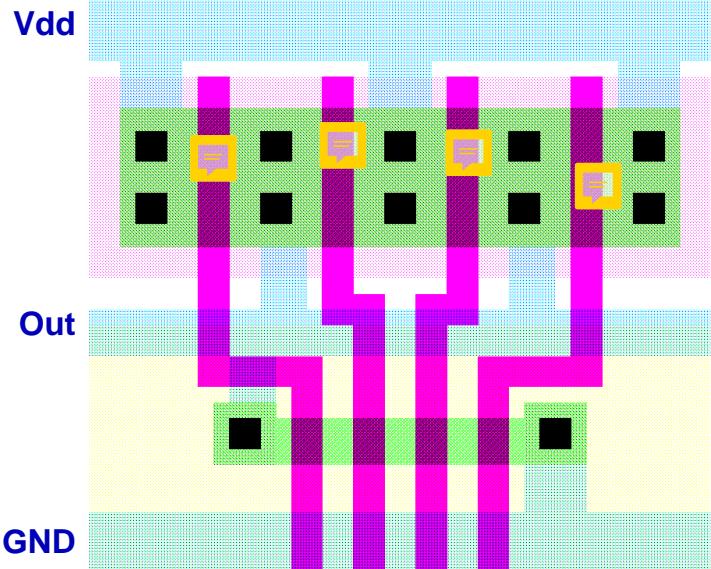
**Beispiel NAND4:**  
NAND-Gatter mit 4 Eingängen



Komplementbildung mit Schalterlogik:

Reihenschaltung  $\Rightarrow$  Parallelschaltung  
Parallelschaltung  $\Rightarrow$  Reihenschaltung  
Schalter  $\Rightarrow$  Komplementärschalter

Chip-Layout:  
MOSFETs an den Kreuzungen  
Grün/Rosa!



Copyright 1996 UCB

Rosa: Metall 1  $\rightarrow$  Eingänge (Gates)  
Blau: Metall 2  $\rightarrow$  Vdd, GND, Out  
Grün: Halbleiter

## Siliziumfläche

**Die Fläche setzt sich (vereinfacht betrachtet) aus verschiedenen Komponenten zusammen**

Logikkomponenten (Gatter, Flip-Flops)

Speicher

Verdrahtung

Anzahl der Ein- und Ausgänge

**Große Daumenwerte für einen 45 nm Prozess**

(Statische) Speicherzelle:  $\approx 0,25 \mu\text{m}^2$

NAND-Gatter mit zwei Eingängen:  $\approx 0,5 \mu\text{m}^2$

Verdrahtungsaufwand:  $\approx$  Flächenerhöhung um c Faktor 1,3

In der Praxis wird häufig eine Angabe in kGates und kbit verwendet.

Entsprechend wird ein Halbleiterprozess durch kGates/mm<sup>2</sup> bzw. kbit/mm<sup>2</sup> beschrieben.  
Als Referenz für ein „Gate“ wird i.d.R. ein NAND2-Gatter (4 Transistoren) verwendet.

# Produktionskosten und Fläche

Die Kosten für die Herstellung eines integrierten Schaltkreises setzen sich aus verschiedenen Komponenten zusammen

- Entwicklungs- und Vertriebsaufwand
- Gehäusekosten
- Testkosten (Fertigungstest)
- eigentliche Produktion

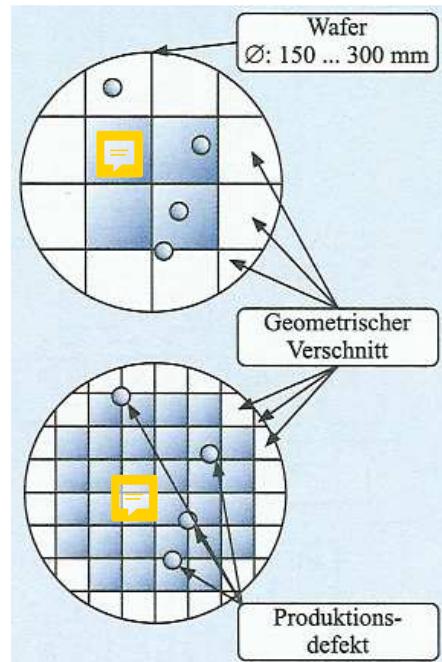
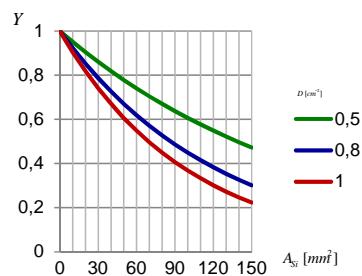
Eine extrem wichtige Kostenkomponente: Fläche

Flächenzuwachs geht überproportional in die Produktionskosten ein

**Grund:** Flächenzuwachs führt zu Erhöhung der Wahrscheinlichkeit, dass ein einzelner Chip nicht funktioniert

Ausbeute:  $Y = e^{-D \cdot A_{Si}}$   
(Yield)

mit  $D$  – Defektaufrite  
 $A_{Si}$  – Fläche



Quelle: Hoffmann, „Grundlagen der Technischen Informatik“



## 4. Realisierung digitaler Schaltungen

### Einleitung Logikbausteine

### Einleitung

Digitale Schaltungen werden zumeist durch elektronische Bauelemente realisiert, grundsätzlich sind jedoch auch andere Realisierungen möglich (z.B. pneumatische Logik mittels Druck).



Logikbausteine im „Dual In-line“-Gehäuse  
(DIP, dual in-line package)



Oberflächen-montierbare  
Logikbausteine  
(SMD surface mount devices)

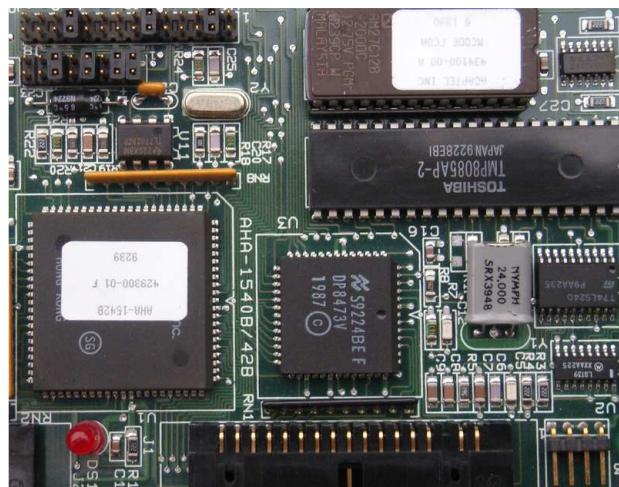
# Integration von Komponenten

Elektronische Bauteile werden zusammen mit notwendigen weiteren Bauelementen (Stecker, Widerstände, Quarze, usw.) auf einer Platine aufgelötet.

Benötigte digitale Funktionen werden durch Bausteine bereitgestellt.

Auf den Außenseiten und im Inneren der Platine können Leiterbahnen verlaufen, welche die Anschlüsse der Bausteine untereinander verbinden.

Ausschnitt aus einer Platine mit elektronischen Bauelementen



4 - Realisierung digitaler Schaltungen

3

## Möglichkeiten zur Realisierung digitaler Funktionen

Zur Realisierung digitaler Funktionen existieren unterschiedliche Arten von Bausteinen:

- **Transistorschaltung:**

Aufbau von Logik mit einzelnen Transistoren.

Wenig kompakt, wird daher nur noch selten direkt verwendet.

- **Logikbausteine:**

Die Bausteine enthalten digitale Grundelemente aus vielen Transistoren.

Durch Kombination mehrerer Bausteine auf einer Platine lassen sich eigene Schaltungen realisieren.

- **Programmierbare Logik (PLD):**

Konfigurierbare logische Grundstruktur, die mit der gewünschten Funktion programmiert wird. Hohe Dichte an Logikfunktionen auf kleinem Raum.

- **ASIC**

Sehr viele Logikfunktionen auf kleinstem Raum.

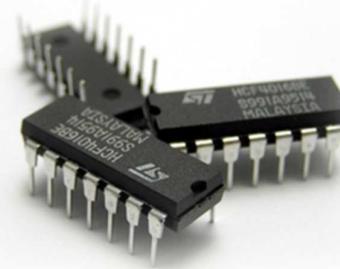
Fertigung in einer Halbleiterfabrik.

- **Speicher:**

Sehr kompaktes Speichern von Information auf kleinstem Raum.

Besitzen eine sehr regelmäßige Struktur.

# Logikbausteine



## Datenblätter

### ► Alle wichtigen Informationen über reale Bausteine stehen in Datenblättern

- Logisch Funktion
- Pinbelegung / Gehäuseform
- Spannungsversorgung
- Zeitverhalten

### ► In der Regel gibt es verschiedene Hersteller für Bausteine mit identischer logischer Funktion

- Datenblätter gibt es von den Herstellern
- Bezeichnung einheitlich, bestehend aus Kürzel für die Logikfamilie und Kürzel (Zahl) für die Funktion

**SN7404, SN74LS04, SNT404, SN74A04, SN74LSA04, SNT404 HEX INVERTERS**

**74HC74**

**Dual D Flip-Flop with Set and Reset**

**High-Performance Silicon-Gate CMOS**

The 74HC74 is identical in pinout to the 7474. The device operates at speeds up to 100 MHz and is available in a variety of packages. The 74HC74 is pin compatible with LSTTL logic.

This device contains two D flip-flops with individual Set and Reset inputs. Information in a D flip-flop is maintained in the complementary state of the output until a Set or Reset signal is applied. Both Q and Q<sup>complement</sup> outputs are available from each flip-flop. The Set and Reset inputs are active-low.

**Features**

- Output Drive Capability: 10 LSTTL Loads
- Output Drive Compatible with CMOS, NMOS, and TTL
- Low Power Consumption
- Wide Supply Voltage Range: 2.0 to 6.0 V
- High-Speed Operation
- High Noise Immunity Characteristics of CMOS Devices
- ESD Protection: HBM > 2000 V, MM > 1000 V
- ESD Protection: BDM > 2000 V, Machine Model > 200 V
- Input Protection: ESD > 2000 V, ESD > 2000 V
- Pb-Free Packages are Available

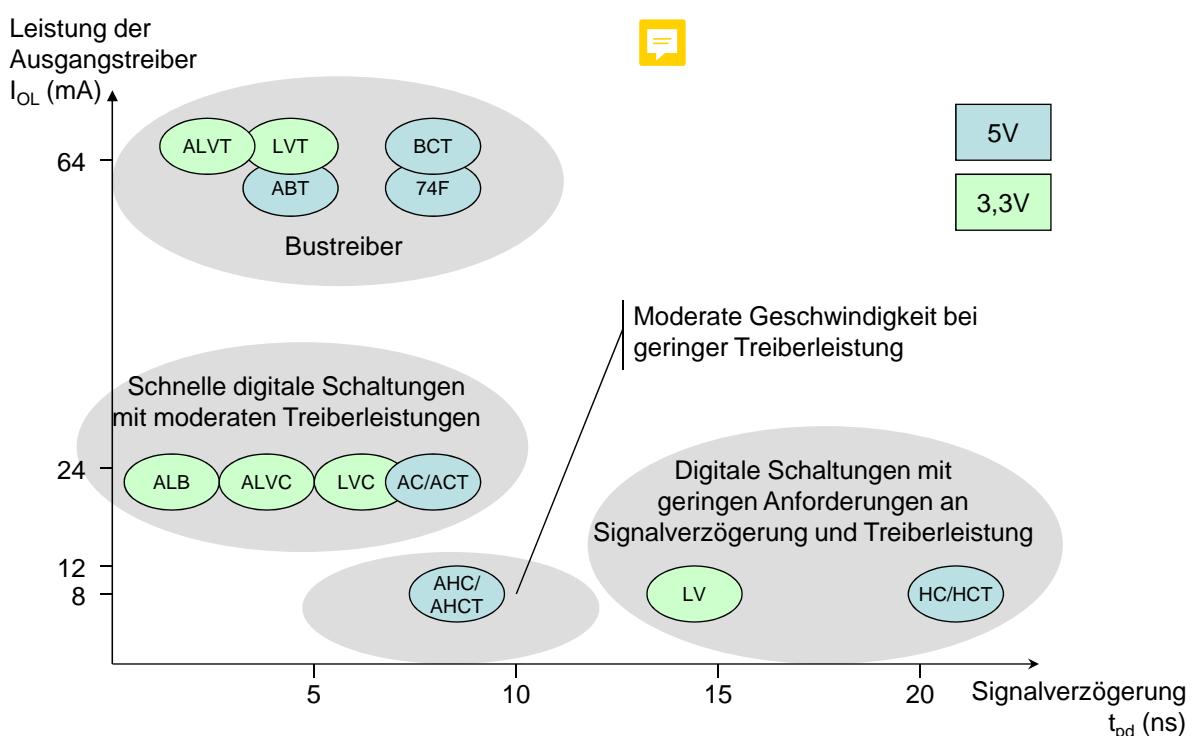
**MARKING DIAGRAMS**

**ORDERING INFORMATION**

# Logikfamilien im Überblick

Kürzel	Bezeichnung	Status
74F	Fast Logic	Aktuell
ABT	Advanced BiCMOS Technology	Aktuell
AC	Advanced CMOS Logic	Aktuell
ACT	Advanced CMOS Logic	Aktuell
AHC	Advanced High-Speed CMOS	Aktuell
AHCT	Advanced High-Speed CMOS	Aktuell
ALB	Advanced Low-Voltage BiCMOS	Aktuell
ALS	Advanced Low-Power Schottky Logic	
ALVC	Advanced Low-Voltage CMOS Technology	Aktuell
ALVT	Advanced Low-Voltage BiCMOS Technology	Aktuell
AS	Advanced Schottky Logic	
AVC	Advanced Very-Low-Voltage CMOS Logic	Aktuell
BCT	BiCMOS Technology	Aktuell
CD4000	CMOS Logic	
ECL	Emitter Coupled Logic	
FCT	Fast CMOS Technology	
HC	High-Speed CMOS Logic	Aktuell
HCT	High-Speed CMOS Logic	Aktuell
LS	Low-Power Schottky Logic	
LV-A	Low-Voltage CMOS Technology	Aktuell
LVC	Low Voltage CMOS Technology	Aktuell
LVT	Low-Voltage BiCMOS Technology	Aktuell
S	Schottky Logic	
TTL	Transistor-Transistor Logic	

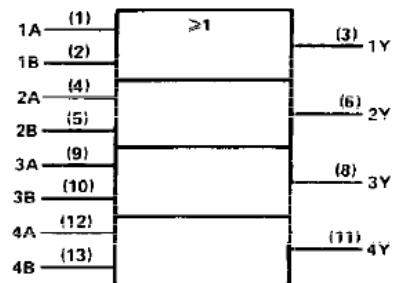
## Logikfamilien: Verzögerung / Treiberleistung



# Beispiel: 74xx32 (4 ODER-Gatter)



logic symbol†

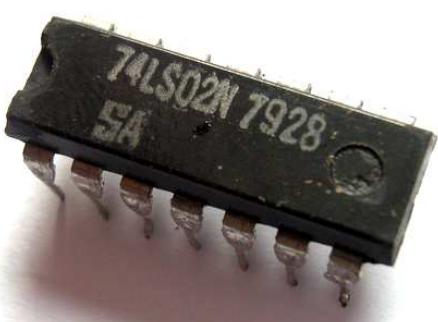


FUNCTION TABLE (each gate)

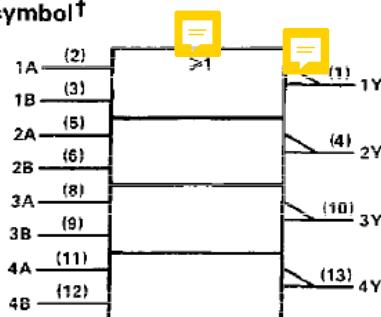
(TOP VIEW)	
1A	1
1B	2
1Y	3
2A	4
2B	5
2Y	6
GND	7
VCC	14
4B	13
4A	12
4Y	11
3B	10
3A	9
3Y	8

INPUTS		OUTPUT
A	B	Y
H	X	H
X	H	H
L	L	L

# Beispiel: 74xx02 ( 4 NOR-Gatter)



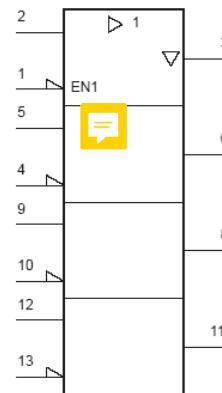
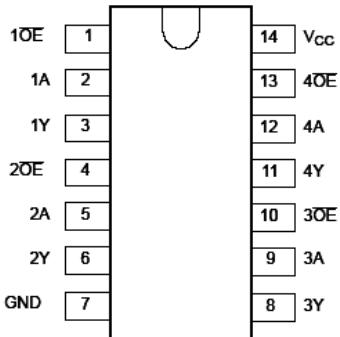
logic symbol†



(TOP VIEW)	
1Y	1
1A	2
1B	3
2Y	4
2A	5
2B	6
GND	7
VCC	14
4Y	13
4B	12
4A	11
3Y	10
3B	9
3A	8

INPUTS		OUTPUT
A	B	Y
H	X	L
X	H	L
L	L	H

# Beispiel: 74xx125 (4 Tri-State Bustreiber)



FUNCTION TABLE

INPUTS		OUTPUT
n̄OE	nA	nY
L	L	L
L	H	H
H	X	Z

NOTES:

H = HIGH voltage level

L = LOW voltage level

X = don't care

Z = high impedance OFF-state

## **5. Boolesche Algebra**

**Booleschen Algebra: Mengen- und Schaltalgebra**

**Abgeleitete Gesetze der Booleschen Algebra**

**Normalformen**

**Minimierung logischer Funktionen**

**Zeitverhalten und Logik-Hazards**

---

## **Boolesche Algebra**

- Mengenalgebra**
- Schaltalgebra**

# Boolesche Algebra: Definition

Gegeben: Menge  $V$ , Operatoren  $\bullet, + : V \times V \rightarrow V$



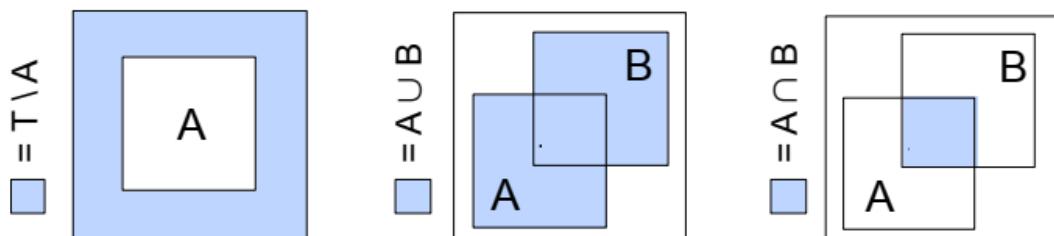
Tripel  $(V, \bullet, +)$  heißt boolesche Algebra,  
wenn die folgenden vier Huntington'schen Axiome für alle  $a, b, c \in V$  gelten:

- ▶ Kommutativgesetze (K):  $a \bullet b = b \bullet a$   
 $a + b = b + a$
- ▶ Distributivgesetze (D):  $a \bullet (b + c) = (a \bullet b) + (a \bullet c)$   
 $a + (b \bullet c) = (a + b) \bullet (a + c)$
- ▶ Neutrale Elemente (N):   
Es existieren  $e, n \in V$  mit  
 $a \bullet e = a$  und  $a + n = a$
- ▶ Inverse Elemente (I):   
Für jedes  $a \in V$  existiert ein  $a'$  mit  
 $a \bullet a' = n$  und  $a + a' = e$

# Boolesche Algebra: Mengenalgebra

- ▶ Mengenalgebra  $(P(T), \cap, \cup)$  über einer Trägermenge  $T$

Boolesche Algebra	Mengenalgebra	
$V$	$P(T)$	Potenzmenge der Trägermenge $T$
$\bullet$	$\cap$	Schnittmenge
$+$	$\cup$	Vereinigungsmenge
$n$	$\emptyset$	Leere Menge
$e$	$T$	Trägermenge
$a'$	$T \setminus A$	Komplementärmenge

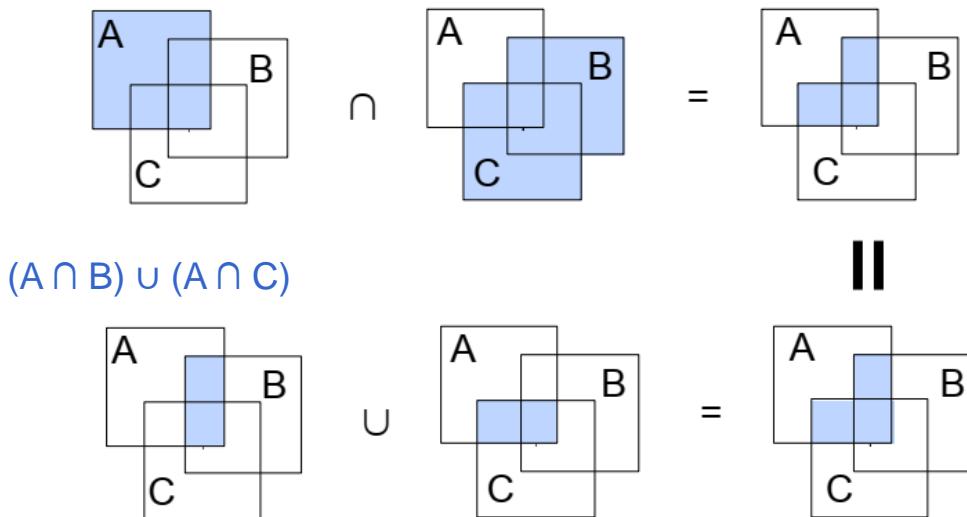


- ▶ Kommutativgesetze gelten trivialerweise!

# Boolesche Algebra: Mengenalgebra

## ► 1. Distributivgesetz:

$$A \cap (B \cup C)$$



- 2. Distributivgesetz kann analog nachgewiesen werden.
- ➔ Mengenalgebra ist eine boolesche Algebra!

# Boolesche Algebra: Schaltalgebra

## ► Auch die Schaltalgebra ( $\{0, 1\}$ , $\wedge$ , $\vee$ ) ist eine boolesche Algebra:

Boolesche Algebra	Schaltalgebra	
$\vee$	$\{0, 1\}$	Wahrheitswerte (TRUE, FALSE)
$\cdot$	$\wedge$	Konjunktion (UND-Operator)
$+$	$\vee$	Disjunktion (ODER-Operator)
$n$	0	„Falsch“ (FALSE)
$e$	1	„Wahr“ (TRUE)
$a'$	$\neg a$	Negation (Verneinung)

*Axiome:* K:  $X \wedge Y = Y \wedge X$   
 $X \vee Y = Y \vee X$

*Beweis:* über Wahrheitstabellen!

D:  $X \wedge (Y \vee Z) = (X \wedge Y) \vee (X \wedge Z)$   
 $X \vee (Y \wedge Z) = (X \vee Y) \wedge (X \vee Z)$

Es gibt weitere boolesche Algebren,  
aber reelle Zahlen ( $\mathbb{R}, *, +$ ) ist keine!

N:  $X \wedge 1 = X$      $X \vee 0 = X$

Warum?

I:  $X \wedge \neg X = 0$      $X \vee \neg X = 1$

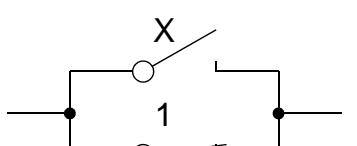
## Abgeleitete Gesetze der Booleschen Algebra

Im Folgenden werden *Schaltalgebra* und *Boolesche Algebra* synonym verwendet!

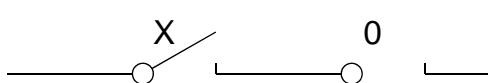
Aus den Axiomen lassen sich alle weiteren Gesetze der Booleschen Algebra ableiten.

## Eliminationsgesetze

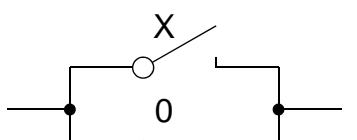
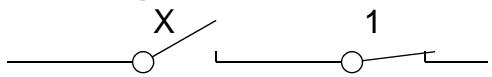
Eliminationsgesetze („Null- und Eins-Gesetze“)

$$X \vee 1 = 1$$


(Analogien in Schalterlogik)

$$X \wedge 0 = 0$$


Auch die Axiome der neutralen Elemente haben Entsprechungen in der Schalterlogik:

$$X \vee 0 = X$$

$$X \wedge 1 = X$$


# Idempotenz- und Komplementgesetze, Doppelnegation

## Idempotenzgesetze

$$X \vee X = X$$
$$X \wedge X = X$$

## Doppelnegation

$$\overline{\overline{X}} = X$$

Axiome der inversen Elemente – auch „Komplementgesetze“ genannt

$$X \vee \overline{X} = 1$$

$$X \wedge \overline{X} = 0$$

# Assoziativgesetze

## Assoziativgesetze

UND-Verknüpfung ist assoziativ:  $(X_1 \wedge X_2) \wedge X_3 = X_1 \wedge (X_2 \wedge X_3)$

ODER-Verknüpfung ist assoziativ:  $(X_1 \vee X_2) \vee X_3 = X_1 \vee (X_2 \vee X_3)$

Bearbeitungsreihenfolge gleicher Operatoren  $\wedge$  bzw.  $\vee$  ist somit egal.

Zur eindeutigen Spezifikation der Funktion werden Klammern nicht benötigt.  
 $\wedge$  kann in eindeutigen Fällen weggelassen werden.

z.B. gilt:  $(X_1 \wedge X_2) \wedge X_3 = X_1 \wedge (X_2 \wedge X_3) = X_1 \wedge X_2 \wedge X_3 = X_1 X_2 X_3$ .

Aufgrund des Kommutativgesetzes können weiterhin die Operanden in ihrer Reihenfolge vertauscht werden.

z.B. gilt:  $(X_1 \wedge X_2) \wedge X_3 = X_1 \wedge X_2 \wedge X_3 = X_3 \wedge X_1 \wedge X_2$ .

# Kürzungsregeln

## Kürzungsregeln

1. Kürzungsregel:  $X_1 \vee (X_1 \wedge X_2) = X_1$  (Absorptionsgesetze)
2. Kürzungsregel:  $X_1 \wedge (X_1 \vee X_2) = X_1$
3. Kürzungsregel:  $X_1 \vee (\overline{X}_1 \wedge X_2) = X_1 \vee X_2$  
4. Kürzungsregel:  $X_1 \wedge (\overline{X}_1 \vee X_2) = X_1 \wedge X_2$
5. Kürzungsregel:  $(X_1 \wedge X_2) \vee (X_1 \wedge \overline{X}_2) = X_1$     
Wichtig!
6. Kürzungsregel:  $(X_1 \vee X_2) \wedge (X_1 \vee \overline{X}_2) = X_1$

## De Morgansche Gesetze

**1. De Morgansches Gesetz:**  $\overline{X_1 \wedge X_2} = \overline{X_1} \vee \overline{X_2}$

**2. De Morgansches Gesetz:**  $\overline{X_1 \vee X_2} = \overline{X_1} \wedge \overline{X_2}$

Die De Morganschen Gesetze gelten auch für mehr als 2 Eingangswerte.  
Ein Beweis hierzu ist mittels der Distributivgesetze möglich.

Es gilt dann:  $\overline{X_1 \wedge X_2 \wedge \dots \wedge X_n} = \overline{X_1} \vee \overline{X_2} \vee \dots \vee \overline{X_n}$

**Achtung:** NAND und NOR sind nicht assoziativ!

$$\overline{A \wedge B \wedge C} \neq \overline{\overline{A \wedge B} \wedge C} \neq \overline{A \wedge \overline{B \wedge C}}$$

$$\overline{A \wedge B \wedge C} \neq (\overline{A \wedge B}) \wedge C \neq A \wedge (\overline{B \wedge C})$$
 

# Shannonsches Gesetz

## Shannonsches Gesetz:

Der invertierte Wert einer Funktion  $f$  ist gleich dem Wert, den die Funktion liefert, wenn man alle Operanden negiert und die Operatoren UND und ODER vertauscht.

$$\overline{f(X_1, X_2, \dots, X_n, \wedge, \vee)} = f(\overline{X_1}, \overline{X_2}, \dots, \overline{X_n}, \vee, \wedge)$$

## Beispiel

mit schrittweiser Anwendung der De Morganschen Gesetze:

$$\begin{aligned} & (\overline{X_1} \vee X_2) \wedge (\overline{X_3} \vee X_4) \\ &= (\overline{X_1} \vee \overline{\overline{X_2}}) \wedge (\overline{X_3} \vee \overline{\overline{X_4}}) && \text{Doppelnegation Eingänge } X_2 \text{ und } X_4 \\ &= (\overline{X_1} \wedge \overline{X_2}) \wedge (\overline{X_3} \wedge \overline{X_4}) && \text{Umwandlung der Ausdrücke in Klammern} \\ &= (\overline{X_1} \wedge \overline{X_2}) \vee (\overline{X_3} \wedge \overline{X_4}) && \text{Umwandlung der Verknüpfung der Klammern} \end{aligned}$$

*Beweis aller Gesetze:* Durch Wahrheitstabellen oder durch Anwendung der Axiome!

# Vollständige Operatorensysteme

Zur Erinnerung:

UND ( $\wedge$ ), ODER ( $\vee$ ) und Negation ( $\neg$ ) sind **Elementaroperatoren**, die ein **vollständiges Operatorensystem** bilden, d. h. mit diesen drei Operatoren können **alle** booleschen Funktionen dargestellt werden!

Es gibt auch noch andere vollständige Operatorensysteme:

NAND und NOR stellen jeweils alleine bereits ein vollständiges Operatorensystem dar!

D. h. jede beliebige boolesche Verknüpfung von beliebig vielen Eingängen mit beliebig vielen Ausgängen lässt sich mit nur einem Typ von Gatter aufbauen: NAND oder NOR

Wie im Kapitel „Elektrotechnische Grundlagen“ gesehen, lässt sich das NAND-System besonders einfach realisieren!

# Vereinfachung boolescher Ausdrücke

---

- ▶ Auflösen aller abgeleiteten Operatoren
- ▶ Anwendung der Rechenregeln
- ▶ Wichtig: Gesetze gelten für alle Terme, nicht nur für Variablen oder Konstanten!

## Beispiel:

 
$$\begin{aligned} (X \oplus 1) \vee X &= ((X \wedge \bar{1}) \vee (\bar{X} \wedge 1)) \vee X && (\oplus \text{ auflösen}) \\ &= ((X \wedge 0) \vee (\bar{X} \wedge 1)) \vee X && (1 \text{ negieren}) \\ &= (0 \vee \bar{X}) \vee X && (\text{Eliminationsgesetz/neutrales Elt.}) \\ &= \bar{X} \vee X && (\text{neutrales Element}) \\ &= 1 && (\text{Komplementgesetz/inverses Elt.}) \end{aligned}$$

## Normalformen



# Definition: Minterme & Maxterme

Die einheitliche Beschreibung von Funktionen erfolgt durch die Verwendung von **Mintermen** und **Maxtermen**.

## Minterm (auch: „Vollkonjunktion“)

Konjunktive Verknüpfung (UND) aller Eingangsvariablen

## Maxterm (auch: „Volldisjunktion“)

Disjunktive Verknüpfung (ODER) aller Eingangsvariablen

Die Eingänge können mit oder ohne Negierung einbezogen werden.

Somit ergeben sich bei N Eingangsvariablen  $2^N$  Minterme und  $2^N$  Maxterme.

# Minterme & Maxterme II

**Beispiel:** Min- & Maxterme einer Funktion mit 3 Eingängen

Alle  $2^3=8$  **Minterme** einer Funktion mit 3 Eingängen

$$\begin{array}{ll} m_0 = \bar{X}_3 \wedge \bar{X}_2 \wedge \bar{X}_1 & m_4 = X_3 \wedge \bar{X}_2 \wedge \bar{X}_1 \\ m_1 = \bar{X}_3 \wedge \bar{X}_2 \wedge X_1 & m_5 = X_3 \wedge \bar{X}_2 \wedge X_1 \\ m_2 = \bar{X}_3 \wedge X_2 \wedge \bar{X}_1 & m_6 = X_3 \wedge X_2 \wedge \bar{X}_1 \\ m_3 = \bar{X}_3 \wedge X_2 \wedge X_1 & m_7 = X_3 \wedge X_2 \wedge X_1 \end{array}$$

Alle **Maxterme** einer Funktion mit 3 Eingängen

$$\begin{array}{ll} M_0 = X_3 \vee X_2 \vee X_1 & M_4 = \bar{X}_3 \vee X_2 \vee X_1 \\ M_1 = X_3 \vee X_2 \vee \bar{X}_1 & M_5 = \bar{X}_3 \vee X_2 \vee \bar{X}_1 \\ M_2 = X_3 \vee \bar{X}_2 \vee X_1 & M_6 = \bar{X}_3 \vee \bar{X}_2 \vee X_1 \\ M_3 = X_3 \vee \bar{X}_2 \vee \bar{X}_1 & M_7 = \bar{X}_3 \vee \bar{X}_2 \vee \bar{X}_1 \end{array}$$

**Achtung:**  $m_0$  verknüpft nur invertierte Variablen,  $M_0$  nur nicht-invertierte!

## Minterme & Maxterme III

Ein **Minterm** liefert nur für eine Eingangskombination eine **1** und für alle anderen Eingangskombinationen eine **0**

Ein **Maxterm** liefert nur für eine Eingangskombination eine **0** und für alle anderen Eingangskombinationen eine **1**

**Beispiele:**

$$m_3 = \overline{X}_3 \wedge X_2 \wedge X_1$$

$$M_5 = \overline{X}_3 \vee X_2 \vee \overline{X}_1$$

$X_3$	$X_2$	$X_1$	$m_3$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

$X_3$	$X_2$	$X_1$	$M_5$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

## Normalformen

Boolsche Funktionen lassen sich durch die Verknüpfung von

**Mintermen oder Maxtermen** realisieren:

### Disjunktive Normalform (DNF)

Disjunktive (ODER) Verknüpfung von Mintermen

### Konjunktive Normalform (KNF)

Konjunktive (UND) Verknüpfung von Maxtermen

⇒ DNF und KNF sind immer zweistufige Schaltnetze!

**Zweistufig:** Vom Eingang zum Ausgang durchlaufen die Signale zwei boolesche Funktionen, wobei Invertierungen nicht mitgezählt werden.

# Disjunktive Normalform (DNF)

**Gegeben:** Logiktabelle der Funktion  $F$

**Vorgehen:**

- **Minterme auswählen**, die an den Stellen eine 1 liefern, an denen die Funktion  $F$  eine 1 liefert.
- Die **disjunktive Verknüpfung** dieser Minterme liefert dann  $F$ .

**Beispiel:** DNF für eine Funktion mit 3 Eingangsvariablen

	$X_3$	$X_2$	$X_1$	$Y$
	0	0	0	0
	0	0	1	1
	0	1	0	0
	0	1	1	1
	1	0	0	1
	1	0	1	0
	1	1	0	1
	1	1	1	1

m<sub>0</sub> nicht verwendet  
m<sub>1</sub> verwendet  
m<sub>2</sub> nicht verwendet  
m<sub>3</sub> verwendet  
m<sub>4</sub> verwendet  
m<sub>5</sub> nicht verwendet  
m<sub>6</sub> verwendet  
m<sub>7</sub> verwendet

$$\begin{aligned}
Y &= m_1 \vee m_3 \vee m_4 \vee m_6 \vee m_7 \\
&= (\overline{X}_3 \wedge \overline{X}_2 \wedge X_1) \\
&\quad \vee (\overline{X}_3 \wedge X_2 \wedge X_1) \\
&\quad \vee (X_3 \wedge \overline{X}_2 \wedge \overline{X}_1) \\
&\quad \vee (X_3 \wedge X_2 \wedge \overline{X}_1) \\
&\quad \vee (X_3 \wedge X_2 \wedge X_1)
\end{aligned}$$

# Konjunktive Normalform (KNF)

**Gegeben:** Logiktabelle der Funktion  $F$

**Vorgehen:**

- Maxterme auswählen**, die an den Stellen eine 0 liefern, an denen die Funktion  $F$  eine 0 liefert.
- Die **konjunktive Verknüpfung** dieser Maxterme liefert dann  $F$ .

**Beispiel:** KNF für eine Funktion mit 3 Eingangsvariablen

	$X_3$	$X_2$	$X_1$	$Y$
	0	0	0	0
	0	0	1	1
	0	1	0	0
	0	1	1	1
	1	0	0	1
	1	0	1	0
	1	1	0	1
	1	1	1	1

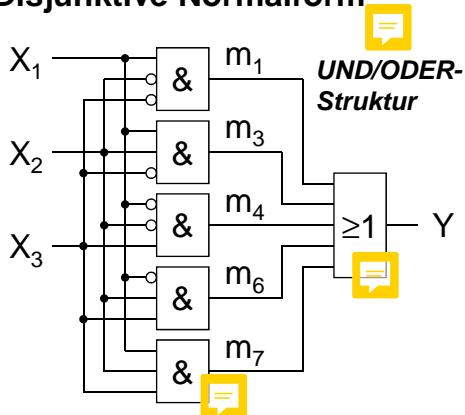
M<sub>0</sub> verwendet  
M<sub>1</sub> nicht verwendet  
M<sub>2</sub> verwendet  
M<sub>3</sub> nicht verwendet  
M<sub>4</sub> nicht verwendet  
M<sub>5</sub> verwendet  
M<sub>6</sub> nicht verwendet  
M<sub>7</sub> nicht verwendet

$$\begin{aligned}
Y &= M_0 \wedge M_2 \wedge M_5 \\
&= (X_3 \vee X_2 \vee X_1) \\
&\quad \wedge (X_3 \vee \overline{X}_2 \vee X_1) \\
&\quad \wedge (\overline{X}_3 \vee X_2 \vee \overline{X}_1)
\end{aligned}$$

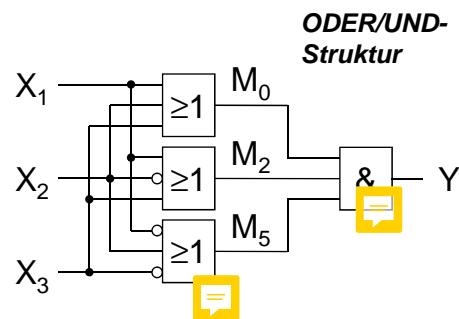
# DNF und KNF in Logikschaltungen

Die disjunktive und konjunktive Normalform führt zu zweistufigen Logikschaltungen.

## Disjunktive Normalform



## Konjunktive Normalform



### Aufwand (oder Komplexität):

Der Flächen-Aufwand wird häufig durch die Summe der Eingänge der Logikverknüpfungen (ohne Inverter) abgeschätzt

Aufwand für das Beispiel:

DNF: 20

KNF: 12

## Konversion der Normalformen

Eine UND/ODER- Struktur kann in eine ODER/UND-Struktur umgewandelt werden (oder umgekehrt).

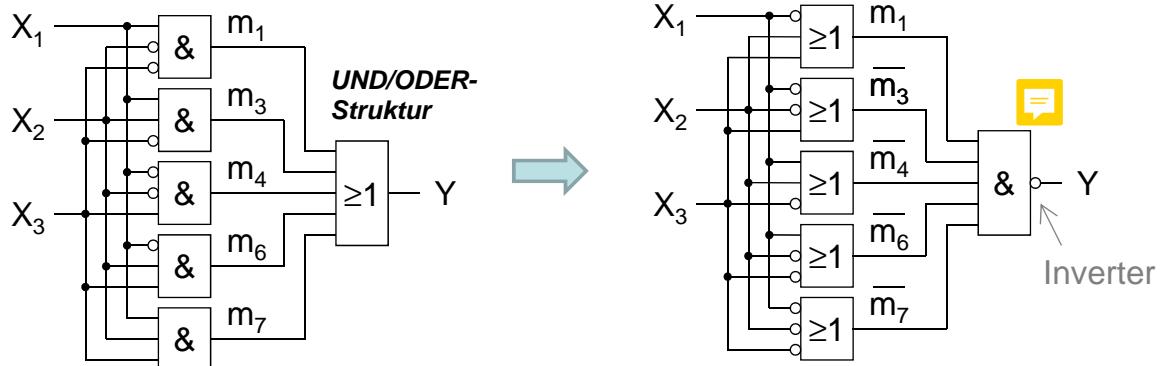
### Vorgehen:

(gemäß dem Shannon'schen Gesetz)

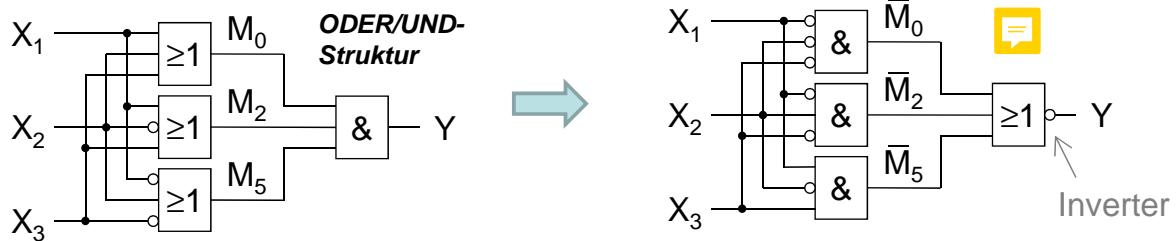
- Eingänge invertieren
- Ausgang invertieren
- UND-Gatter durch ODER-Gatter ersetzen
- ODER-Gatter durch UND-Gatter ersetzen

# Beispiele: Konversion der Normalformen

## Disjunktive Normalform



## Konjunktive Normalform



## Minimierung logischer Funktionen

# Minimierungsverfahren

## Boolesche Algebra:

Umformen der vorgegebenen Gleichungen mit den vorgestellten **Rechenregeln** für boolesche Gleichungen, bis eine Minimalform erzielt wird.

*Selbst für Gleichungen mit wenigen Unbekannten schwierig!*

## Grafische Verfahren:

Geeignete Darstellung boolescher Funktionen, so dass Terme anschaulich zusammengefasst und dabei vereinfacht werden.

Bekanntestes Verfahren: **Karnaugh-Veitch-Diagramms (KV-Diagramm)**

*Bis etwa 4-5 Eingangsvariablen einsetzbar. Bei mehr Eingangsvariablen wird das Verfahren schnell unübersichtlich.*

*(Algorithmische Verfahren sind dann im Vorteil, insbesondere weil sie in Software implementiert und dann automatisiert ausgeführt werden können.)*

## Algorithmische Verfahren:

Sukzessives Zusammenfassen von Min- bzw. Maxtermen, so dass die Terme vereinfacht werden.

Bekanntester Algorithmus: **Quine-McCluskey-Verfahren**

*Algorithmische Verfahren bilden die Grundlage für die Minimierung boolescher Funktionen mittels Software (z.B. Synthese beim ASIC-Entwurf).*

# Minimierung

## Definition: Implikant

Ein Konjunktionsterm I einer Funktion F ist ein Implikant, wenn gilt:

$$I = 1 \Rightarrow F = 1$$

*Hinweis:* Alle Minterme sind Implikanten. (Alle werden für F benötigt!)

## Ziel der Minimierung:

Erreiche gleiches Ergebnis (d. h. gleiche Funktion) durch möglichst wenige, möglichst kleine Implikanten, d. h. geringen Aufwand!

Logik bleibt dabei zweistufig, d. h. Verzögerung bleibt (ungefähr) gleich.

## Idee der Minimierung der DNF:

**Zusammenfassung** von Mintermen der DNF **zu neuen Implikanten**, so dass der Aufwand der Implikanten geringer als der der ursprünglichen Minterme ist

## Verwendung der 5. Kürzungsregel:

$$\forall A, B : (A \wedge B) \vee (A \wedge \overline{B}) = A$$

**(Dies gilt für beliebige Variablen oder Terme A und B!)**

# Minimierung mit Hilfe der Rechenregeln

**Beispiel:**

$$Y = m_1 \vee m_3 \vee m_4 \vee m_6 \vee m_7$$

$$\begin{aligned} &= (\overline{X_3} \wedge \overline{X_2} \wedge X_1) \\ &\quad \vee (\overline{X_3} \wedge X_2 \wedge X_1) \\ &\quad \vee (X_3 \wedge \overline{X_2} \wedge \overline{X_1}) \\ &\quad \boxed{\vee (X_3 \wedge X_2 \wedge \overline{X_1})} \\ &\quad \vee (X_3 \wedge X_2 \wedge X_1) \end{aligned}$$

5. Kürzungsregel:

$$(A \wedge B) \vee (A \wedge \overline{B}) = A$$

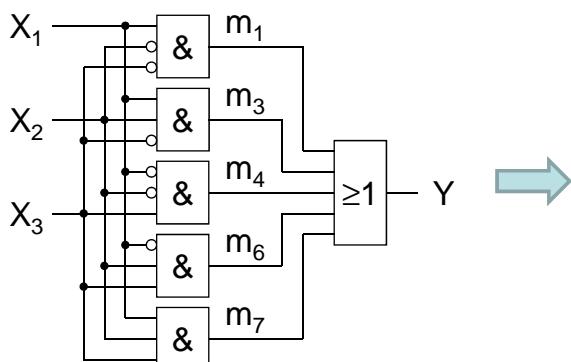
$$(\overline{X_3} \wedge \overline{X_2} \wedge X_1) \vee (\overline{X_3} \wedge X_2 \wedge X_1) = (\overline{X_3} \wedge X_1)$$

$$(X_3 \wedge X_2 \wedge \overline{X_1}) \vee (X_3 \wedge X_2 \wedge X_1) = (X_3 \wedge X_2)$$

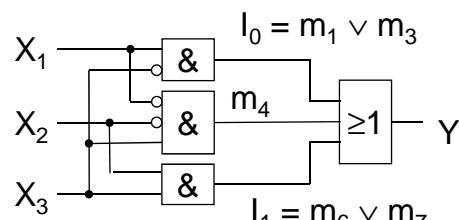
**Ziel:** Erreiche gleiches Ergebnis (d. h. gleiche Funktion) durch möglichst wenige, möglichst kleine Implikanten!

## Minimierung mit Hilfe der Rechenregeln II

$$\begin{aligned} Y &= m_1 \vee m_3 \vee m_4 \vee m_6 \vee m_7 \\ &= (\overline{X_3} \wedge X_1) \\ &\quad \vee (X_3 \wedge \overline{X_2} \wedge \overline{X_1}) \\ &\quad \vee (X_3 \wedge X_2) \end{aligned}$$



Aufwand: 20



Aufwand: 10

**Beachte:** Die Logik bleibt bei dieser Minimierung immer zweistufig!

# Karnaugh-Veitch-Diagramm

## KV-Diagramm:

Mehrdimensionale grafische Darstellung einer Funktion, bei der sich benachbarte Zellen nur im Hinblick auf **eine** Eingangsvariable unterscheiden.  
Jedes Feld entspricht einer Zeile der Logiktabelle.

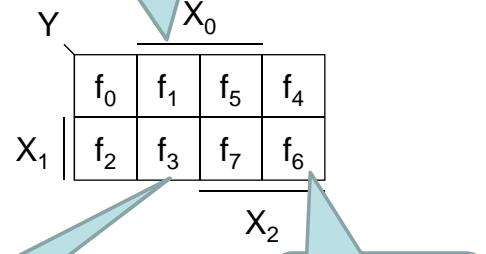
**Beispiel:** 3 Eingangsvariablen

Der Strich in der Darstellung bedeutet, dass die Variable den Wert 1 hat!

Logiktabelle

$X_2$	$X_1$	$X_0$	$Y$
0	0	0	$f_0$
0	0	1	$f_1$
0	1	0	$f_2$
0	1	1	$f_3$
1	0	0	$f_4$
1	0	1	$f_5$
1	1	0	$f_6$
1	1	1	$f_7$

KV-Diagramm



$X_0=1$   
 $X_1=1$   
 $X_2=0$

$X_0=0$   
 $X_1=1$   
 $X_2=1$

## Das Ausfüllen des KV-Diagramms

- ▶ Ein KV-Diagramm für  $n$  Eingangsvariablen hat  $2^n$  Felder.
- ▶ Das KV-Diagramm wird mit den Variablen an den Rändern beschriftet:
  - Jede Variable kommt in negierter und nicht-negierter Form vor.
  - Die Zuordnung der Variablen zu den einzelnen Feldern kann beliebig erfolgen, aber
  - ... horizontal und vertikal benachbarte Felder dürfen sich nur in genau einer Variablen unterscheiden.
- ▶ Mit Hilfe der Wahrheitstabelle der zu optimierenden Funktion werden die einzelnen Felder gefüllt:
  - Eine 1 wird eingetragen, wenn ein Minterm der Funktion vorliegt.
  - Andernfalls wird eine 0 eingetragen.

## Beispiel: Ausfüllen des KV-Diagramms

Logik-/Wahrheitstabelle

$X_2$	$X_1$	$X_0$	$Y$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

KV-Diagramm

$Y$		$X_0$	
	0	1	0
$X_1$	0	1	1

... entspricht der Belegung

$Y$	$\overline{X_0}$	$X_0$	$X_0$	$\overline{X_0}$
	0	1	0	1
$X_1$	0	1	1	0

$X_2$        $X_2$        $X_2$        $X_2$

## Minimierung der DNF mit dem KV-Diagramm

**Zusammenfassung benachbarter Einträge mit Einsen in einem Kästchen**  
(Für die KNF-Minimierung werden später Nullen zusammengefaßt.)

Für die Darstellung des resultierenden Implikanten werden weniger Eingangsvariablen benötigt (durch wiederholte Anwendung der 5. Kürzungsregel).

Reduktion um...

**1** Eingangsvariable:      **2** benachbarte Felder zusammenfassen

**2** Eingangsvariablen:      **4** benachbarte Felder zusammenfassen

**3** Eingangsvariablen:      **8** benachbarte Felder zusammenfassen

usw.

**Generell:** Es werden immer möglichst viele Felder zusammengefasst  
→ möglichst kurze Formeln (d. h. wenige Variablen) für die Implikanten

Diejenige(n) Variable(n), die in einem Kästchen sowohl in invertierter als auch nicht-invertierter Form vorkommen, können entfernt werden!

# Minimierung mit dem KV-Diagramm II

**Beispiel:**

**Logiktabelle**

$X_2$	$X_1$	$X_0$	$Y$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

**KV-Diagramm**

**Implikanten**

 $I_0 = \bar{X}_2 \wedge X_0$ 
 $I_1 = X_1 \wedge X_0$ 
 $I_2 = X_2 \wedge \bar{X}_1 \wedge \bar{X}_0$

## Definition: Primimplikanten & Kern-Primimplikanten

### Primimplikant:

Ein Implikant, der in keinem anderen Implikanten vollständig enthalten ist, wird als Primimplikant (*PI*) bezeichnet

(*Primimplikanten werden auch als Primterm bezeichnet*)

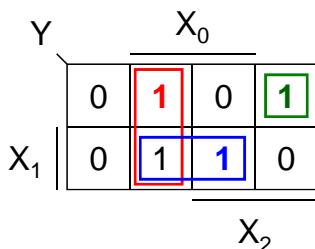
→ maximale Zusammenfassung der Felder!

### Kern-Primimplikant:

Ein Primimplikant, welcher einen exklusiven Min- oder Maxterm enthält, wird als Kern-Primimplikant (*KPI*) bezeichnet.

→ Exklusiver Min-/Maxterm ist in keinem anderen PI enthalten!

### Beispiel:



Die oben gezeigten Implikanten  $I_0$ ,  $I_1$  und  $I_2$  sind alle KPIs.  
(Exklusive Minterme fettgedruckt!)

# Ermittlung der Minimalform mit Hilfe des KV-Diagramms

Zur Ermittlung der **disjunktiven Minimalform** werden im KV-Diagramm zunächst alle Einsen zu Implikanten zusammengefasst.

Dann werden möglichst wenige Implikanten ausgewählt und disjunktiv zur Minimalform verknüpft.

Um die ursprüngliche Funktion zu erhalten, müssen die Implikanten so ausgewählt werden, dass sie alle Minterme (d. h. Einsen im KV-Diagramm) enthalten. D. h., die Einsen müssen von den Implikanten *überdeckt* werden.

→ **Problem:** Auswahl ist nicht immer eindeutig!

## Vorgehensweise:

1. Zunächst alle Kern-Primimplikanten aufnehmen  
(Müssen berücksichtigt werden, da sie einen exklusiven Min- bzw. Maxterm enthalten.)
2. Anschließend weitere Primimplikanten hinzufügen, bis alle Min- bzw. Maxterme berücksichtigt sind

# Ermittlung der Minimalform mit Hilfe des KV-Diagramms

## Beispiel: Ermittlung der disjunktiven Minimalform

		X <sub>0</sub>		
		0	1	0
		0	1	1
		X <sub>1</sub>		X <sub>2</sub>
Y		0	1	0
	X <sub>1</sub>	0	1	0

Das Beispiel enthält nur Kern-Primimplikanten, somit ergibt sich die folgende disjunktive Minimalform:

$$Y = I_0 \vee I_1 \vee I_2 = (\bar{X}_2 \wedge X_0) \vee (X_1 \wedge X_0) \vee (X_2 \wedge \bar{X}_1 \wedge \bar{X}_0)$$

# KV-Diagramm für 4 Eingangsvariablen

$X_3$	$X_2$	$X_1$	$X_0$	$Y$
0	0	0	0	$f_0$
0	0	0	1	$f_1$
0	0	1	0	$f_2$
0	0	1	1	$f_3$
0	1	0	0	$f_4$
0	1	0	1	$f_5$
0	1	1	0	$f_6$
0	1	1	1	$f_7$
1	0	0	0	$f_8$
1	0	0	1	$f_9$
1	0	1	0	$f_{10}$
1	0	1	1	$f_{11}$
1	1	0	0	$f_{12}$
1	1	0	1	$f_{13}$
1	1	1	0	$f_{14}$
1	1	1	1	$f_{15}$

$Y$	$X_0$			
	$f_0$	$f_1$	$f_5$	$f_4$
$X_1$	$f_2$	$f_3$	$f_7$	$f_6$
	$f_{10}$	$f_{11}$	$f_{15}$	$f_{14}$
$X_3$	$f_8$	$f_9$	$f_{13}$	$f_{12}$
$X_2$				

## Beispiel: Funktion mit 4 Eingängen

$X_3$	$X_2$	$X_1$	$X_0$	$Y$
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

$Y$	$X_0$			
	0	0	0	$1$
$X_1$	0	1	1	1
	1	1	0	0
$X_3$	1	1	0	0
$X_2$				

Die Primimplikanten  $I_1$  und  $I_5$  enthalten die farbig gekennzeichneten exklusiven Minterme.  
 Sie sind somit Kern-Primimplikanten.  
 Primimplikant  $I_3$  deckt die verbleibenden Minterme ab.  
 Damit ergibt sich folgende Minimalform:  

$$Y = I_1 \vee I_3 \vee I_5 = (X_3 \wedge \overline{X}_2) \vee (\overline{X}_3 \wedge X_1 \wedge X_0) \vee (\overline{X}_3 \wedge X_2 \wedge \overline{X}_0)$$

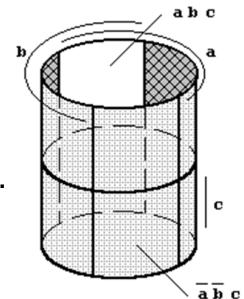
# Nachbarschaften im KV-Diagramm

**Das KV-Diagramm besitzt eine Torus-Topologie!**

Felder am linken und rechten Rand sind benachbart.

Felder am oberen und unteren Rand sind ebenfalls benachbart.

Dies ist bei der Ermittlung der Primimplikanten zu berücksichtigen.



**Beispiel:** Ermittlung der Primimplikanten bei Nachbarschaft am Rand

		$X_0$			
		0	1	1	0
		0	0	0	0
$X_1$		1	0	0	1
		0	1	1	0
		$X_2$		$X_3$	

		$X_0$			
		1	0	0	1
		0	0	0	0
$X_1$		0	0	0	0
		1	0	0	1
		$X_2$		$X_3$	

## Sonderfälle

Einige Funktionen besitzen keine Kern-Primimplikanten.

Bei anderen Funktionen lassen sich keine Minterme/Maxterme zusammenfassen, sie können somit nicht minimiert werden.

**Beispiel:** Funktion ohne Kern-Primimplikanten

		$X_0$			
		0	1	0	0
		1	1	1	0
$X_1$		1	0	1	0
		1	1	1	0
		$X_2$		$X_3$	

**Beispiel:** Nicht minimierbare Funktion

		$X_0$			
		0	1	0	1
		1	0	1	0
$X_1$		0	1	0	1
		1	0	1	0
		$X_2$		$X_3$	

→ Erstellung einer zweistufigen Logik kann zu großer Schaltung/Formel führen!

Es gibt auch andere Methoden der „Formelsynthese“, die kleinere, mehrstufige Logik erzeugen können. (Z. B. die Funktion rechts:  $Y_2 = X_1 \oplus X_2 \oplus X_3 \oplus X_4$ )

# Ermittlung der konjunktiven Minimalform

Bei der **konjunktiven Minimalform** werden **Maxterme** (Nullen im KV-Diagramm) zusammengefasst.

**Vorgehen:** Inverse Blockbildung

- Bilden der *disjunktiven Minimalform* für *inverse Funktion*  $\bar{Y}$  durch Zusammenfassen der *Nullen* im KV-Diagramm
- Anwendung der De-Morgan'schen Regel ergibt *konjunktive Minimalform* der *Originalfunktion*  $Y$

**Beispiel:**

		$X_0$			
		0	1	0	0
$X_1$	0	0	1	0	0
	1	0	1	1	1
				$X_3$	

**Implikanten für  $\bar{Y}$ !**

$$I_1 = \overline{X}_3 \wedge \overline{X}_0$$

$$I_2 = \overline{X}_3 \wedge X_2$$

$$I_3 = \overline{X}_2 \wedge X_1 \wedge \overline{X}_0$$

$$I_4 = X_2 \wedge \overline{X}_1 \wedge \overline{X}_0$$

$$\begin{aligned} \bar{Y} &= I_1 \vee I_2 \vee I_3 \vee I_4 = (\overline{X}_3 \wedge \overline{X}_0) \vee (\overline{X}_3 \wedge X_2) \vee (\overline{X}_2 \wedge X_1 \wedge \overline{X}_0) \vee (X_2 \wedge \overline{X}_1 \wedge \overline{X}_0) \\ Y &= \overline{I_1} \wedge \overline{I_2} \wedge \overline{I_3} \wedge \overline{I_4} = (X_3 \vee X_0) \wedge (X_3 \vee \overline{X}_2) \wedge (X_2 \vee \overline{X}_1 \vee X_0) \wedge (\overline{X}_2 \vee X_1 \vee X_0) \end{aligned}$$

## Ausnutzung von Redundanzen

Es gibt Schaltungen, bei denen der Ausgangswert bei bestimmten Eingangskombinationen beliebig/egal ist. (engl. „*don't care*“)

Dies kann zur Verbesserung der Minimierung verwendet werden.

Strategie: Die notwendigen Implikanten werden möglichst groß gemacht!

**Beispiel:** Disjuktive Minimierung mit Redundanzen

		$X_0$			
		0	1	0	0
$X_1$	1	*	1	*	1
	0	*		*	1
				$X_3$	

$$Y = (\overline{X}_3 \wedge X_1) \vee (\overline{X}_2 \wedge X_0) \vee (X_2 \wedge X_1)$$

\* = „*don't care*“

# Ausnutzung von Redundanzen II

**Beispiel:** Konjunktive Minimierung mit Redundanzen

Lösung 1:

		X <sub>0</sub>	
		0	*
Y		0	0
X <sub>1</sub>	1	1	*
	0	*	*
	0	1	0

X<sub>2</sub>      X<sub>3</sub>

Lösung 2:

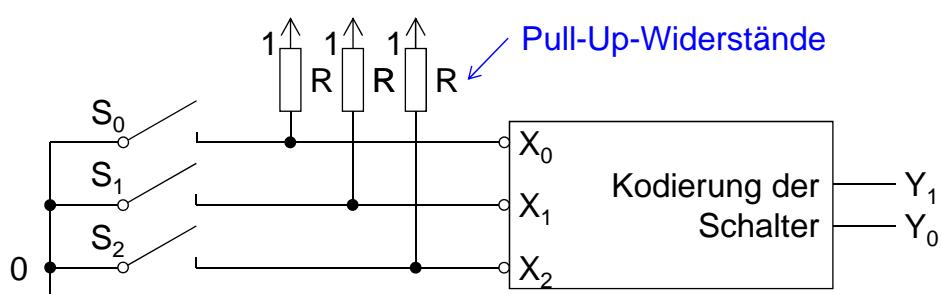
		X <sub>0</sub>	
		0	*
Y		0	0
X <sub>1</sub>	1	1	*
	0	*	*
	0	1	0

X<sub>2</sub>      X<sub>3</sub>

*Übung: Gleichung und Aufwand der beiden Lösungen*

## Anwendungsbeispiel: Kodierung von „Radioschaltern“

„Radioschalter“: Kombination von mehreren Schaltern, bei denen sichergestellt ist, dass jeweils nur ein Schalter geschlossen ist



X <sub>2</sub>	X <sub>1</sub>	X <sub>0</sub>	Y <sub>1</sub>	Y <sub>0</sub>
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	*	*
1	0	0	1	1
1	0	1	*	*
1	1	0	*	*
1	1	1	*	*

		X <sub>0</sub>	
		0	*
Y <sub>1</sub>		0	1
X <sub>1</sub>	1	*	1
	1	*	*
	0	0	*

X<sub>2</sub>

$$Y_1 = X_2 \vee X_1$$

		X <sub>0</sub>	
		0	*
Y <sub>0</sub>		1	0
X <sub>1</sub>	0	*	*
	1	*	*
	0	1	*

X<sub>2</sub>

$$Y_0 = X_2 \vee X_0$$

# Minimierung nach dem Quine-McCluskey-Verfahren

---

Das Minimierungsverfahren nach W. V. Quine (1952) wurde von und E. J. McCluskey (1956) erweitert und basiert auf einer iterativen Vorgehensweise.

**Grundprinzip** ähnlich wie KV-Diagramm:

Suchen nach benachbarten Termen, die aufgrund der Gleichung

$$(X_i \wedge X_k) \vee (X_i \wedge \overline{X_k}) = X_i$$

vereinfacht werden können.

Terme werden systematisch in Tabellen gesammelt und vereinfacht.

**Vorteile** gegenüber der Minimierung mit KV-Diagrammen:

- algorithmisches Verfahren, das sich besser in Software implementieren lässt
- auch für mehr als 5-6 Eingangsvariablen gut geeignet

*Die Rechenleistung steigt exponentiell mit der Anzahl der Eingangsvariablen*

*Für sehr komplexe Optimierungsprobleme werden daher auch heuristische Verfahren eingesetzt, die nicht immer zu einer vollständigen Minimierung führen.*

*(Bekanntestes heuristisches Verfahren: ESPRESSO-Algorithmus)*



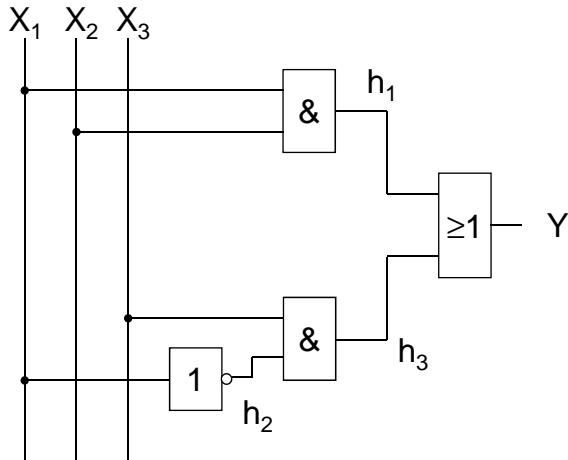
## Zeitverhalten und Hazards

# Verzögerungszeit

Reale Hardware benötigt für die Berechnung einer logischen Funktion eine gewisse Zeit.

Diese Zeit wird auch als **Verzögerungszeit** bezeichnet.

**Beispiel:**



Vereinfachende Annahme: Alle Gatter besitzen die identische Verzögerungszeit 1ns.

Wie reagiert die Schaltung auf Wechsel der Eingangssignale ?

## Beispiele

**Beispiel 1:**

$$(X_3, X_2, X_1): (1, 1, 1) \rightarrow (0, 1, 0)$$

Erwartetes Verhalten Y: 1 → 0

**Logiktabelle**

X <sub>3</sub>	X <sub>2</sub>	X <sub>1</sub>	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

**Beispiel 2:**

$$(X_3, X_2, X_1): (1, 1, 1) \rightarrow (1, 1, 0)$$

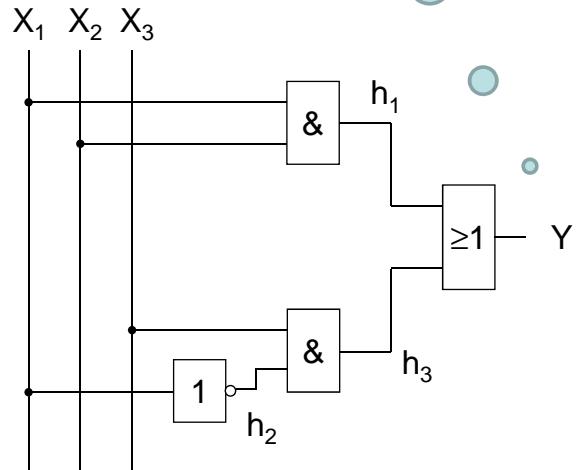
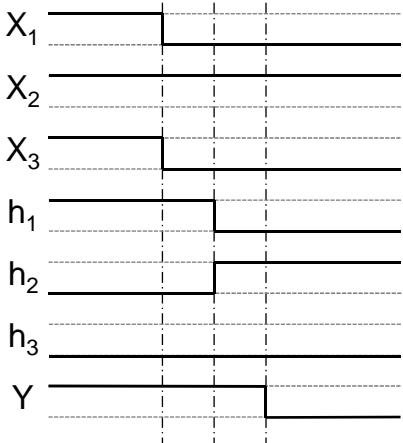
Erwartetes Verhalten Y: 1 → 1

# Analyse des Zeitverhaltens

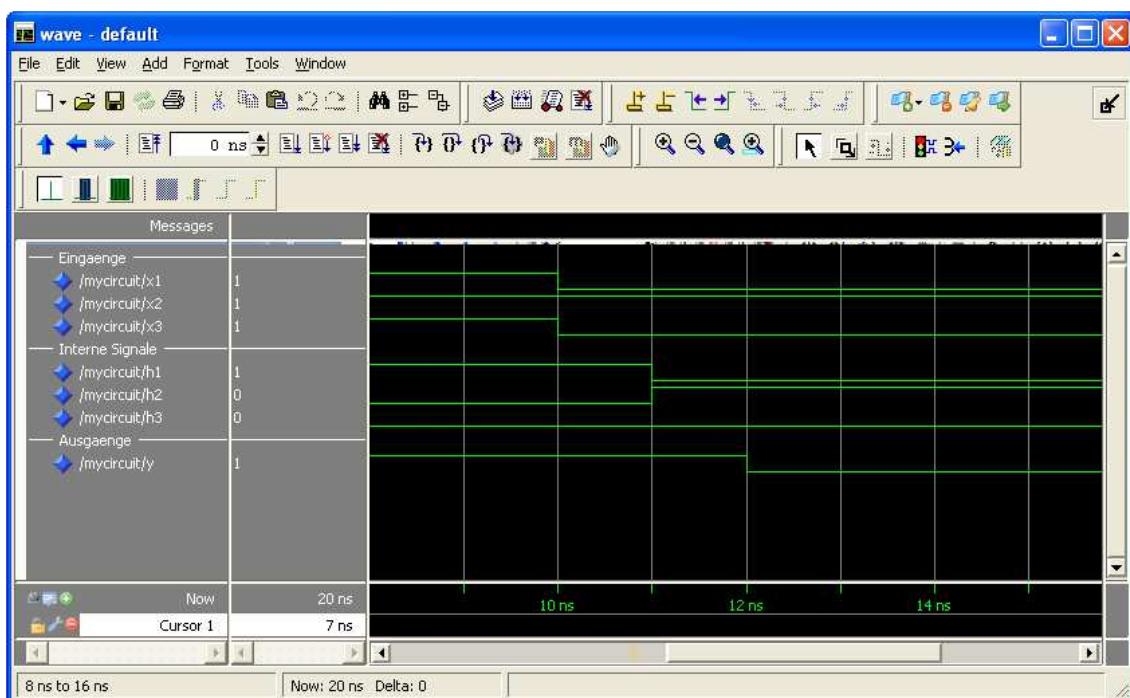
Definition von „Hilfssignalen“  
Sukzessive zeitliche Analyse der Schaltung

Verzögert, aber  
im Prinzip das  
erwartete  
Verhalten!

**Beispiel 1**  $(X_3, X_2, X_1): (1,1,1) \rightarrow (0,1,0)$

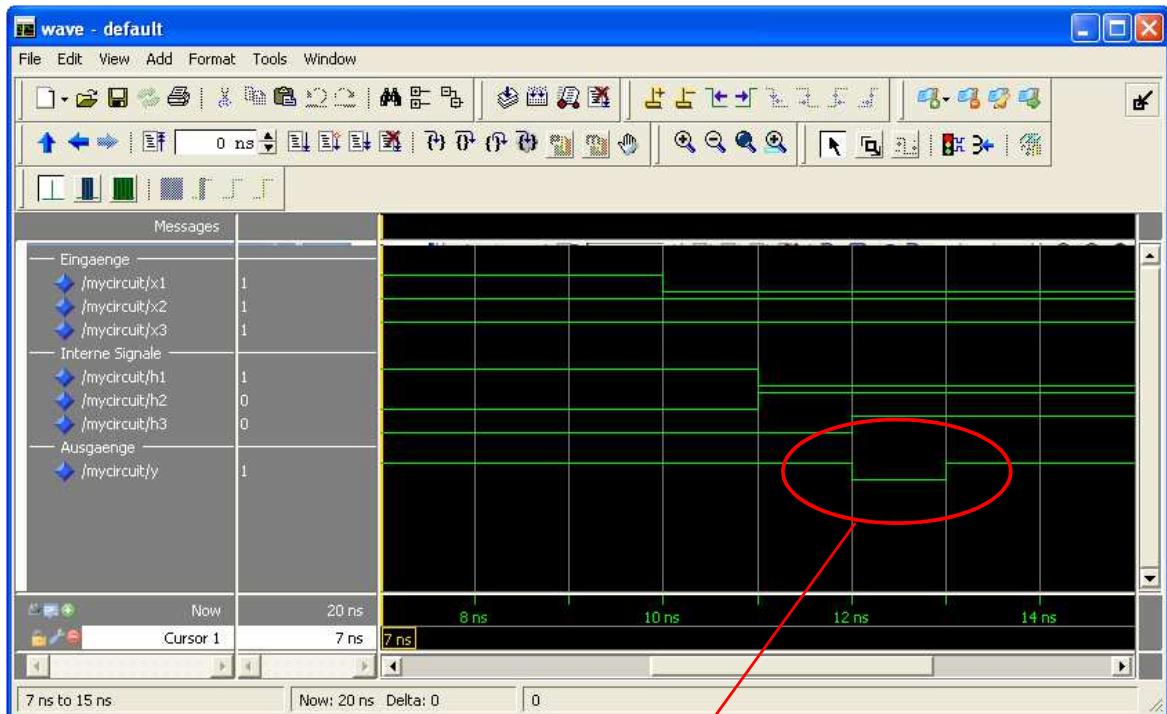


## Simulation mit Logiksimulator - $(X_3, X_2, X_1): (1,1,1) \rightarrow (0,1,0)$



Auch hier: Erwarteter Wechsel des Ausgangssignals nach 2 ns

## Simulation Beispiel 2 - $(X_3, X_2, X_1): (1,1,1) \rightarrow (1,1,0)$



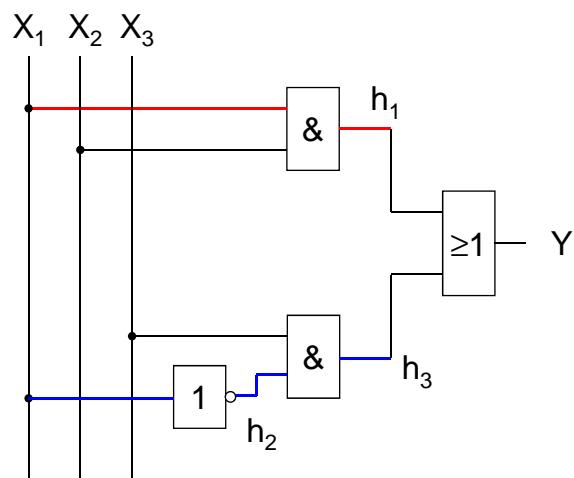
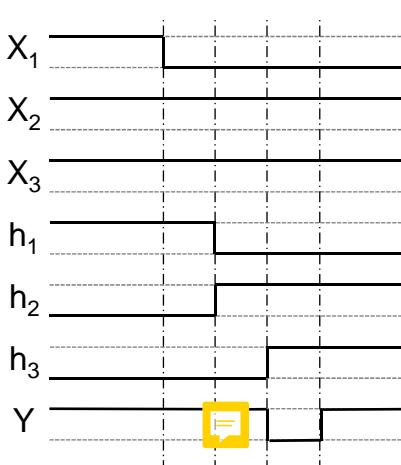
Eigentlich müsste Y gemäß Logiktafel konstant 1 sein. Was ist hier passiert ?

## Analyse des Beispiels 2

Das Signal X1 wird dem oberen UND-Gatter direkt zugeführt.  
Dem unteren UND-Gatter wird es invertiert und verzögert zugeführt.

Daher ändern sich die Signale h1 und h3 nicht gleichzeitig

Für eine kurze Zeit sind beide Signale 0 und damit ist auch  $Y=0$



# Logik-Hazards

**Das in Beispiel 2 beobachtete Verhalten kann zu Fehlfunktionen in einer Hardware führen,**

z.B. wenn nachfolgende Schaltungsteile auf Flanken des Signals Y reagieren!

Das kurzzeitige Auftreten eines „falschen“ Ausgangswertes bei der Änderung eines Eingangssignals hat verschiedene (englische) synonyme Bezeichnungen:

**Glitch**

**Logik-Hazard**

**Spike**

**Störimpuls**

**Man unterscheidet:**

➤ **Statische Hazards**

Ein eigentlich konstantes Signal nimmt kurzzeitig einen falschen Wert an (*s. Beispiel*).

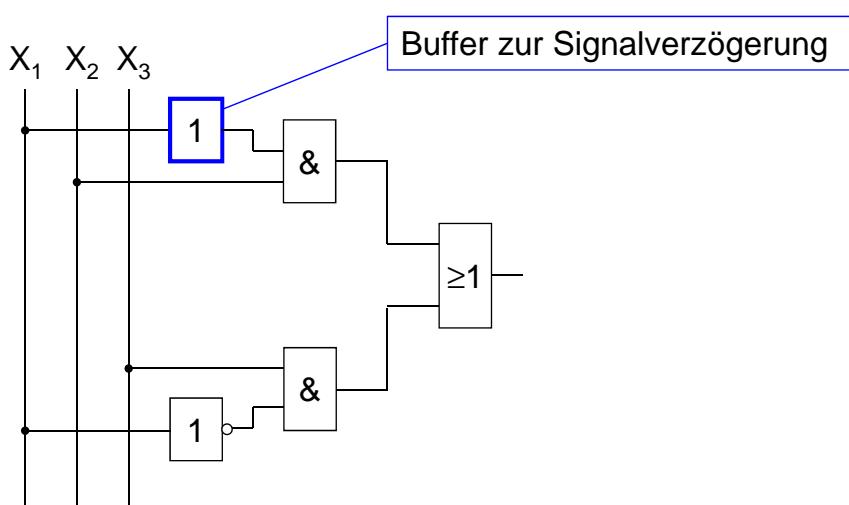
➤ **Dynamische Hazards**

Bei einem Signalwechsel wird zunächst der richtige Wert angenommen, dann der falsche und am Ende wieder der richtige.

## Vermeidung von Logik-Hazards

### Möglichkeit 1: Verzögerungszeiten angleichen

Im Beispiel:



**Nachteil:** Unsicher, da Verzögerungszeiten realer Gatter variieren  
(und i.a. auch nicht identisch sind)

## Vermeidung von Logik-Hazards II

Möglichkeit 2 (besser): Redundante Implikanten hinzufügen

Im Beispiel:

Würde man der Logikfunktion den Implikanten

$$I = (X_3 \wedge X_2)$$

hinzufügen, wäre sichergestellt, dass unabhängig von den anderen ausgewählten Implikanten - d. h. auch beim Wechsel von  $X_1$  - eine 1 am Ausgang Y erscheint.

$X_3$	$X_2$	$X_1$	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Nicht vergessen:

Durch das Hinzufügen redundanter Implikanten erhöht sich der Aufwand !

## Hazardfreie Logikminimierung mit dem KV-Diagramm

Strategie:

1. Logikminimierung wie bisher beschrieben
2. Hinzufügen weiterer Implikanten, so dass alle benachbarten Implikanten durch die neu hinzugefügten Implikanten verbunden werden

Im Beispiel:

		$X_2$			
		0	1		
		0	1		
$X_3$	0	0	1	0	
	1	1	1	0	

**minimal**



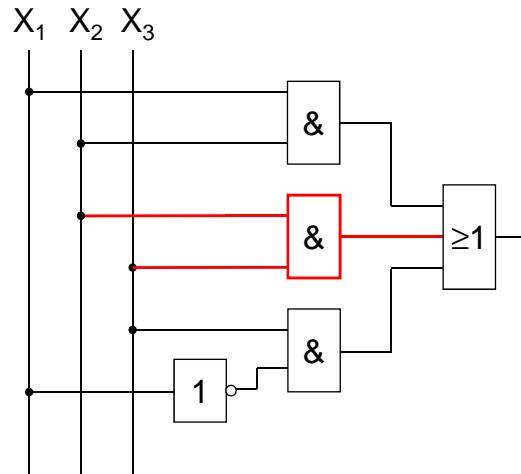
		$X_2$			
		0	1		
		0	1		
$X_3$	0	0	1	0	
	1	1	1	0	

**hazardfrei**

aber unsicher, da beim 1-0-Wechsel von  $X_1$  kurzfristig keiner der blauen Implikanten gelten kann → Ausgabe 0

sicher, da roter Implikant auch bei Wechsel von  $X_1$  immer gilt → Ausgabe 1

## Hazardfreie Implementierung des Beispiels



Jetzt kann kein Logik-Hazard beim Wechsel eines Signals mehr auftreten!

**Aber:** Beim *gleichzeitigem Wechsel mehrerer Signale* können immer noch sogenannte **Funktions-Hazards** auftreten, die nicht durch zusätzliche Implikanten vermieden werden können!

# **6. Digitale Grundschatungen**

**Multiplexer, Demultiplexer und Busse**

**Code-Umsetzer**

---

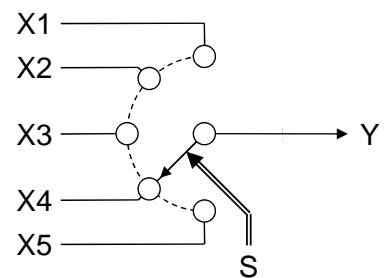
**Multiplexer, Demultiplexer und Busse**

# Funktion eines Multiplexers

**Aufgabe:** Auswahl von einem aus mehreren Eingangssignalen → verbindet Y selektiv mit mehreren Eingängen ohne Kurzschluss!

**Funktionsweise:** wie ein Drehschalter

- Wählt ein Eingangssignal aus und schaltet es auf den Ausgang
- Auswahl des Eingangssignal erfolgt durch Steuersignal(e) S



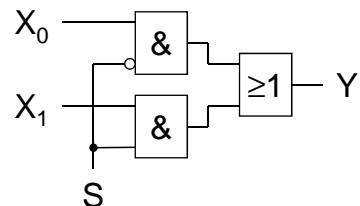
**Beispiel:** 2:1 Multiplexer

S	X <sub>1</sub>	X <sub>0</sub>	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

kompaktere Darstellung:

S	Y
0	X <sub>0</sub>
1	X <sub>1</sub>

Schaltung:

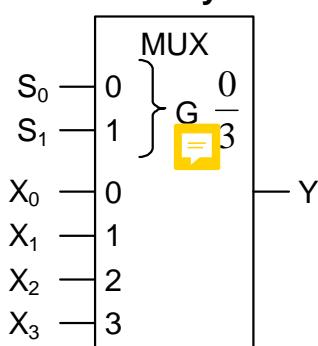


## 4:1 Multiplexer

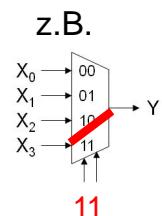
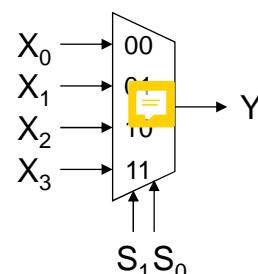
**Funktionstabelle**

S <sub>1</sub>	S <sub>0</sub>	Y
0	0	X <sub>0</sub>
0	1	X <sub>1</sub>
1	0	X <sub>2</sub>
1	1	X <sub>3</sub>

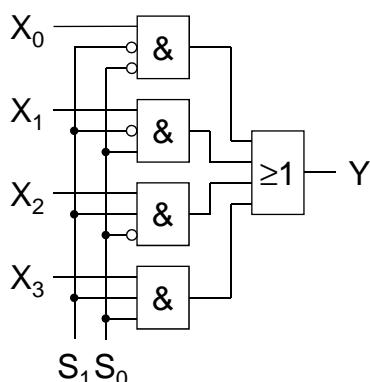
**DIN 40900-Symbol**



**Blockschaltbild-Symbol**



**Schaltung**



Ein **Blockschaltbild** ist ein vereinfachtes Schaltbild ohne Details der Kombinatorik, das Eingangs- und Ausgangssignale und Signale zwischen den Blöcken darstellt.

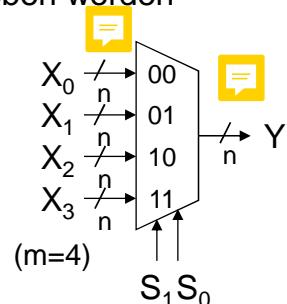
Es wird auch zur hierarchischen Strukturierung großer Schaltungen verwendet.

# Multiplexer und Busse

## Definition: Bus

„Gemeinsam genutzter Transportweg zwischen verschiedenen Funktionseinheiten“  
(aus: Hoffmann, „Grundlagen der Technischen Informatik“)

- Busse bestehen aus mehreren Signalleitungen, sind also **Signalvektoren**.
- Zu jedem Zeitpunkt muss jedes Bit einen genau definierten Pegel haben.
- Soll ein Bus von mehreren Signalquellen (Eingängen) getrieben werden können, muss ein Eingang ausgewählt werden:
  - **Standard-Busansteuerung:**  
Multiplexer schaltet von mehreren Eingängen genau einen auf den Bus durch. Eventuell müssen die Steuersignale des Multiplexers durch einen *Encoder* (s. Abschnitt *Code-Umsetzer*) bereitgestellt werden.
    - Bei n-Bit-Bussen (d. h. Bus und Eingänge sind Signalvektoren) benötigt jedes Bit (Einzelsignal) einen Multiplexer.  
→ Bei m Eingängen sind also n m:1-Multiplexer erforderlich!
    - **Nachteil:**  
Bei vielen Eingängen werden Multiplexer groß und langsam.

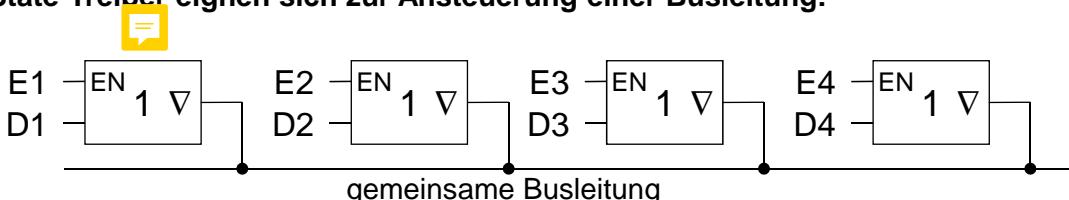


X<sub>0</sub> - X<sub>3</sub> und Y sind  
Signalvektoren mit  
jeweils n Bit Breite!

In Blockschaltbildern können mehrere zusammengehörige Signale zu einem **Signalvektor** zusammengefasst werden. Die Zahl der Bits wird im Schaltbild neben dem Vektor notiert.

# Alternative Bus-Ansteuerung mit Tri-State-Treibern

Tri-State-Treiber eignen sich zur Ansteuerung einer Busleitung:



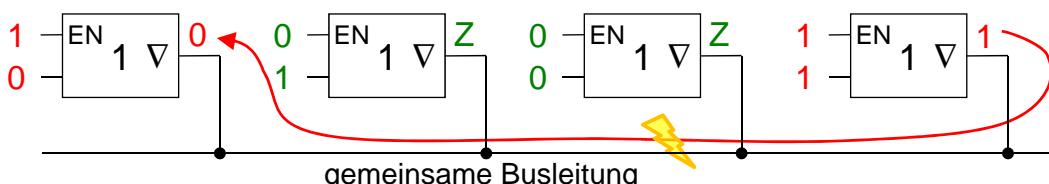
Bei Bussen mit Tri-State-Treibern dürfen Signale an einem Eingang (d. h. an einer Leitung) zusammengeführt werden!

**Vorteil:** Kompakter als Auswahl über Multiplexer!

**Aber:** Durch die Gesamtschaltung muss sichergestellt werden, dass nur maximal einer der Freigabeeingänge (E1, E2, E3, E4) aktiv ist und damit seinen zugehörigen Dateneingang auf die Busleitung schaltet.

Sind mehrere Freigabeeingänge aktiv, kann es zum Kurzschluss kommen:

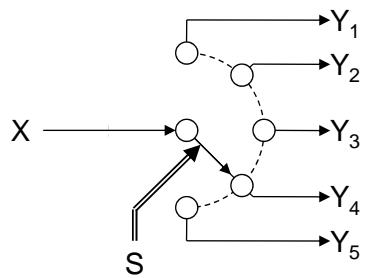
**Beispiel: Kurzschluss: Treiber legen gleichzeitig 0 V und V<sub>dd</sub> an die Busleitung an!**



# Demultiplexer

**Aufgabe:** Von mehreren Ausgängen einen auswählen, auf den Eingang geschaltet wird  
(andere Ausgänge: Wert 0)

Gegenstück zum Multiplexer!



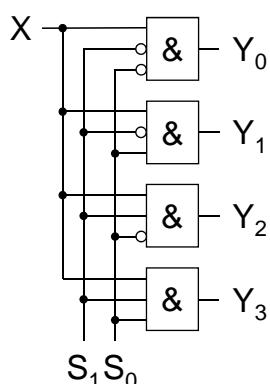
**Funktionsweise:** wie ein Drehschalter

- Wählt genau einen Ausgang aus und verbindet ihn mit dem Eingang
- Auswahl des Ausgangs erfolgt durch Steuersignal(e) S

**Beispiel: 1:4 Demultiplexer**

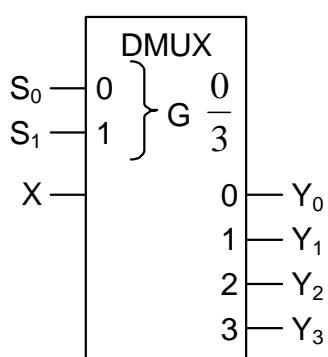
S <sub>1</sub>	S <sub>0</sub>	Y <sub>0</sub>	Y <sub>1</sub>	Y <sub>2</sub>	Y <sub>3</sub>
0	0	X	0	0	0
0	1	0	X	0	0
1	0	0	0	X	0
1	1	0	0	0	X

Schaltung:

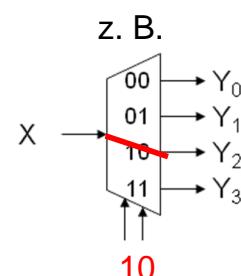
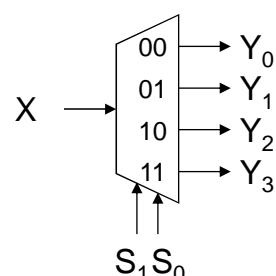


## Symbole eines 1:4 Demultiplexers

DIN 40900-Symbol:



Blockschaltbildsymbol



# Code-Umsetzer

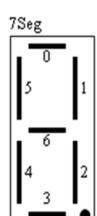
# Code-Umsetzer

- ▶ In der Technischen Informatik werden vielfältige Codes zur Informationsdarstellung verwendet, z. B. Dual- und Hexadezimalzahlen, BCD, Gray-Code, etc. (siehe Kap. 9).
- ▶ Ein **Code-Umsetzer** wandelt Wert aus einem Code in einen anderen Code.

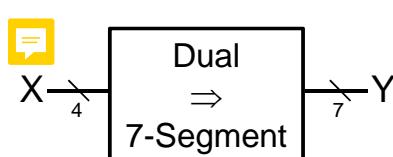
## Beispiel:

- ▶ Umsetzung von X (4-Bit-Dualzahlen) zur Darstellung der Hexadezimal-Ziffern (0, 1, ..., 9, A, B, C, D, E, F) auf einer 7-Segment-Anzeige Y

Bitnummern der  
7-Segment-Anzeige:



Blockschaltbild des  
Code-Umsetzers:



X(3:0)	Y(6:0)	Hex
0000	011111	0
0001	0000110	1
...		
1010	1110111	A
1011	1111100	b
1100	0111001	C
1101	1011110	d
1110	1111001	E
1111	1110001	F

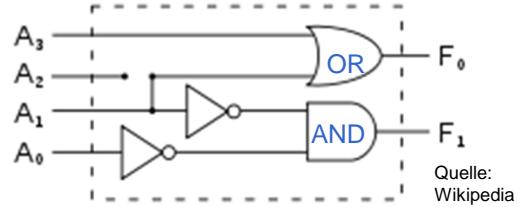
# Encoder und Decoder

- **Einfacher Encoder:** Kodiert Index des aktiven Eingangs binär  
Von  $2^n$  Eingangssignalen (A) darf nur eines 1 sein. Dann wird der Index dieses Signals binär kodiert auf n Ausgangssignalen (F) ausgegeben.  
(ähnlich der Kodierung der Radio-Knöpfe!)

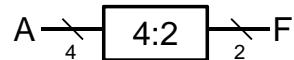
**Beispiel:** 4-zu-2-Encoder

A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>	F <sub>1</sub>	F <sub>0</sub>
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

Schaltbild (US-Norm)



Blockschaltbild:



- **Decoder:** Dekodiert Binärwert am n-Bit-Eingang →  $2^n$  Ausgangssignale

*Implementierung:*

Demultiplexer mit konstanter 1 am Dateneingang setzt wie gewünscht nur den durch Steuervektor als Index selektierten Ausgang auf 1, alle anderen auf 0.

## Prioritätsencoder I

**Erweiterung des einfachen Encoders:**

- Überwacht alle Eingänge
- Falls einer 1 ist, wird der Ausgang Akt auf 1 gesetzt (0 sonst)
- Ausgang Index gibt den Index des höchsten Eingangs mit Wert 1 an.

**Beispiel:** Prioritätsencoder mit 4 Eingängen

X <sub>3</sub>	X <sub>2</sub>	X <sub>1</sub>	X <sub>0</sub>	Akt	Index
0	0	0	0	0	*
0	0	0	1	1	0
0	0	1	0	1	0
0	0	1	1	1	0
0	1	0	0	1	0
0	1	0	1	1	0
0	1	1	0	1	0
0	1	1	1	1	0
1	0	0	0	1	1
1	0	0	1	1	1
1	0	1	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1
1	1	0	1	1	1
1	1	1	0	1	1
1	1	1	1	1	1

$$\text{Akt} = X_3 \vee X_2 \vee X_1 \vee X_0$$

$$\text{Index}_1 = X_3 \vee X_2$$

$$\text{Index}_0 = X_3 \vee (\overline{X}_2 \wedge X_1)$$

Index <sub>1</sub>		X <sub>0</sub>			
X <sub>1</sub>	X <sub>3</sub>	*	0	1	1
0	0	0	0	1	1
0	1	0	1	1	1
1	0	1	1	1	1
1	1	1	1	1	1

Index <sub>0</sub>		X <sub>0</sub>			
X <sub>1</sub>	X <sub>3</sub>	*	0	0	0
0	0	0	0	0	0
0	1	0	1	1	1
1	0	1	1	1	1
1	1	1	1	1	1

## Prioritätsencoder II

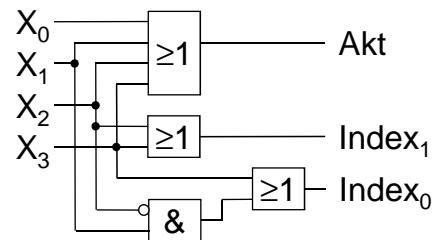
**Beispiel:** Prioritätsencoder mit 4 Eingängen (Fortsetzung)

$$\text{Akt} = X_3 \vee X_2 \vee X_1 \vee X_0$$

$$\text{Index}_1 = X_3 \vee X_2$$

$$\text{Index}_0 = X_3 \vee (\overline{X}_2 \wedge X_1)$$

Schaltung:



### Anwendung

Häufig in Rechnerschaltungen verwendet, wenn Ereignisse parallel auftreten und auf einzelnen Leitungen signalisiert werden, aber nicht parallel bearbeitet werden können.

- Prioritätsencoder signalisiert dem Rechner, welches Ereignis mit höchster Priorität bearbeitet werden muss!

### Beispiele (in einem Mikroprozessor)

- Arbiter zur Datenbussteuerung (in Verbindung mit Multiplexern)
- Interrupt-Schaltungen

## Read-Only Memory (ROM) als Code-Umsetzer

### ► Read-Only Memory (ROM oder Festwertspeicher)

Speicherbaustein, dessen Werte nur ausgelesen, also nicht verändert werden können.

### ► Beispiel: m Adressbits, n Datenbits → ROM speichert $2^m$ n-Bit-Worte



- Diese rein kombinatorische Schaltung wird durch folgende Logiktabelle mit m Eingängen und n Ausgängen beschrieben und kann wie jede boolesche Funktion minimiert werden:

A <sub>m-1</sub> A <sub>m-2</sub> ... A <sub>1</sub> A <sub>0</sub>	D <sub>n-1</sub> D <sub>n-2</sub> ... D <sub>0</sub>
0 0 ... 0 0	Datenwort 0
0 0 ... 0 1	Datenwort 1
...	...
1 1 ... 1 1	Datenwort 2 <sup>m</sup> -1

- Es gibt noch viele andere Möglichkeiten, ROM zu implementieren.



# 7. Asynchrone Rückkopplungen und Flipflops

## Asynchron rückgekoppelte Schaltungen

### Bisher: Schaltnetze ohne Rückkopplungen

- Ausgänge hängen nur (funktional) von aktuellen Eingangswerten ab
- kein „Gedächtnis“/ Speicher / Zustand

### Jetzt: Schaltnetze mit externer (asynchroner) Rückkopplung



- Schaltnetz „rechnet“, bis sich stabiler Wert der Rückkopplungssignale einstellt.  
**Problem:** Instabile Rückkopplungen können schwingen (oszillieren)!
- Rückkopplungssignale müssen hazardfrei erzeugt werden  
→ andernfalls kann nicht erwartetes Verhalten auftreten

### Sonderfall: Ausgänge werden als Rückkopplungssignale verwendet

# Zustände

Nachdem die Rückkopplungssignale einen stabilen Wert angenommen haben und sich die Eingänge nicht ändern, werden auch die Ausgänge einen stabilen Wert annehmen:

- Schaltung befindet sich in einem stabilen **Zustand**.
- Rückkopplungssignale kennzeichnen den Zustand: **Zustandsvektor**
- Änderungen der Eingangssignale können zu einer Änderung der Ausgangssignale und/oder der Rückkopplungssignale führen.  
→ Schaltung nimmt einen neuen Zustand ein.
- Zustand ist „Gedächtnis“ der Schaltung, das von früheren Eingangswerten abhängt.
- Man spricht von *sequenziellen Schaltungen*, da sie eine Folge (Sequenz) von Zuständen einnimmt, oder von *Schaltwerken*.

## Beispiel: Einfaches RS-Flipflop

### Speicherelement für 1 Bit

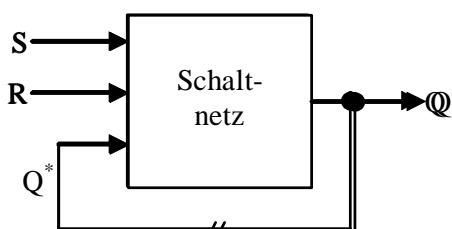
### RS: Reset-Set

Drei erlaubte Eingangskombinationen:

- Halten (Wert unverändert): R,S=00
- Rücksetzen (Wert 0 speichern): R,S=10
- Setzen (Wert 1 speichern): R,S=01

Die Kombination R,S=11 sollte vermieden werden, da sie zu einem nicht spezifizierten Ausgangsverhalten führt!

### Blockschaltbild

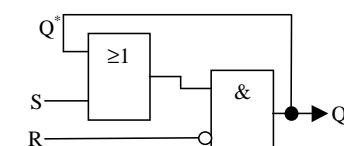


Durch gedankliches Auftrennen der Rückkopplung erhält man neuen Eingang  $Q^*$  und kann Funktions-tabelle für das Schaltnetz aufstellen:

S	R	$Q^*$	Q
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	*
1	1	1	*

### Schaltung

Konjunktive Minimalform



$$Q = (\neg R) \wedge (S \vee Q^*)$$

# Zustandsdiagramm des einfachen RS-Flipflops

**Zustandsdiagramme:** zur grafische Darstellung von Zustandsänderungen

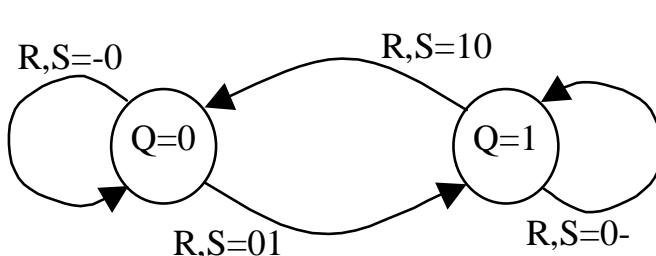
**Zustände werden durch Kreise gekennzeichnet**

enthalten Zustand (Bezeichnung oder Zustandsvektor),  
also die Rückkopplungswerte

**Pfeile kennzeichnen den Übergang von einem Zustand in einen anderen**

Die für diesen Übergang benötigten Eingangswerte werden  
an den Pfeil geschrieben.

Bei asynchronen Schaltungen erfolgt Zustandsübergang sofort nach Änderung  
eines (Eingangs-)Signals!



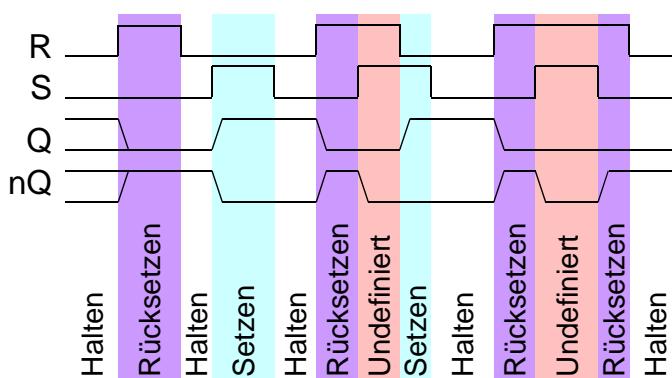
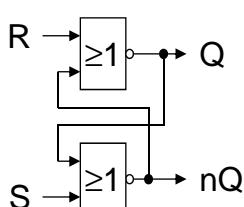
S	R	$Q^*$	Q
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	*
1	1	1	*

**RS-Flipflop:** - zwei Zustände ( $Q=0, Q=1$ )  
- unvollständig (Übergänge für  $R,S=11$  nicht spezifiziert)

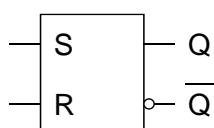
## Einfaches RS-Flip-Flop: Dynamisches Verhalten

### Schaltung

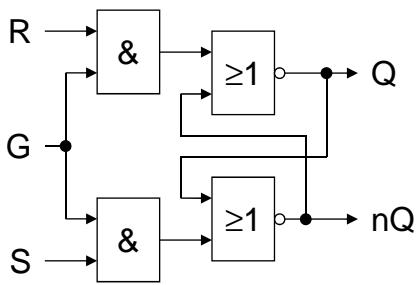
- symmetrische Darstellung mit  $nQ$
- trotzdem nur ein Rückkopplungssignal!



Schalsymbol  
DIN 40900



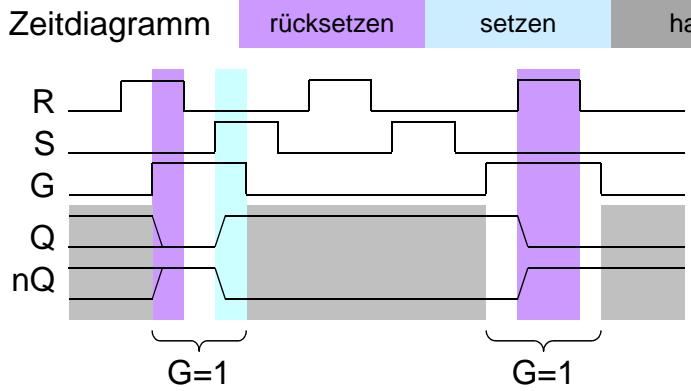
## Pegelgesteuertes RS-Flip-Flop (Latch)



Dem einfachen RS-FlipFlop wird eine Torschaltung („Gate“) aus zwei UND-Gattern vorangestellt:

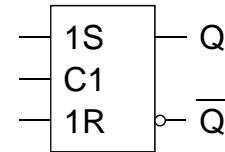
- Das FlipFlop kann nur noch mit  $G=1$  gesetzt oder rückgesetzt werden.
- Mit  $G=0$  hält das FlipFlop seinen alten Wert.

Zeitdiagramm



Schaltsymbol

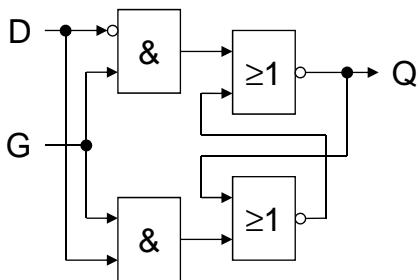
DIN 40900



Das Steuersignal C wird mit 1 nummeriert.

Die Signale S und R können nur bei aktivem Steuersignal 1 (d.h. C) die Ausgänge schalten.

## Pegelgesteuertes D-Flip-Flop (Latch)

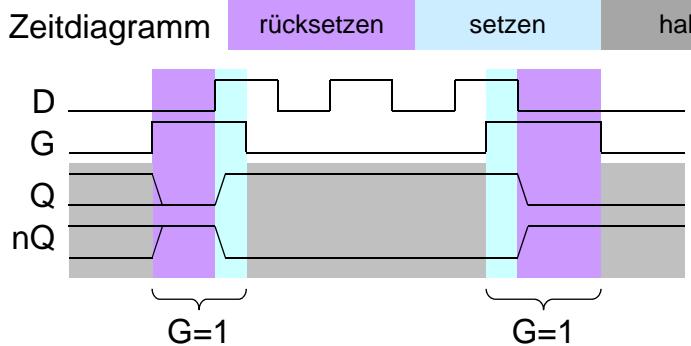


Das pegelgesteuerte RS-FlipFlop wird zum D-FlipFlop, wenn man den D-Eingang direkt mit S und invertiert mit R verbindet.

Die undefinierte Engabe  $R,S=11$  ist so nicht mehr möglich!

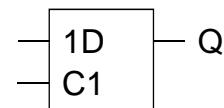
- Mit  $G=1$  folgt der Ausgang dem D-Signal.
- Mit  $G=0$  hält das FlipFlop seinen alten Wert.

Zeitdiagramm



Schaltsymbol

DIN 40900



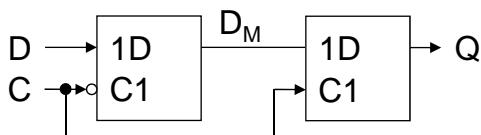
Das Steuersignal C wird mit 1 nummeriert.

Das Signal D verändert nur bei aktivem Steuersignal 1 (d.h. Signal C) den Ausgang Q.

**Frage:** Warum gibt es kein D-FF ohne Torschaltung (Gate), also einfaches RS-FF, bei dem  $R=\neg S$  festverdrahtet ist? (Würde Problem mit  $R,S=11$  eliminieren...)

# Flankengesteuertes D-Flip-Flop

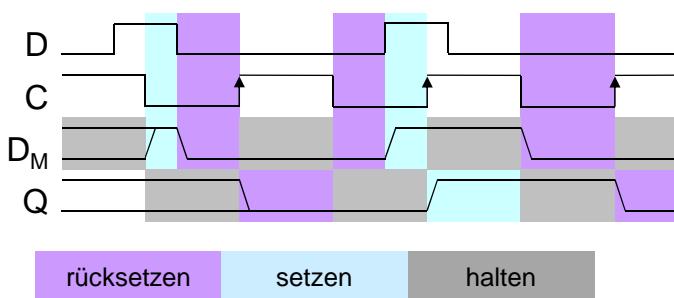
Flankengesteuertes D-FlipFlop in Master-Slave-Architektur



Durch Hintereinanderschalten zweier pegelgesteuerter D-FlipFlops ergibt sich ein flankengesteuertes D-FlipFlop:

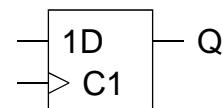
- Mit der steigenden Flanke des Eingangs C wird der Wert von D am Ausgang übernommen.
- Sonst hält das FlipFlop seinen alten Wert.

Zeitdiagramm



Schaltsymbol

DIN 40900



Steuersignal C wird mit 1 nummeriert.

→ Dieses Flip-Flop ist das heute am häufigsten verwendete!  
(FPGAs enthalten nur D-FFs!)

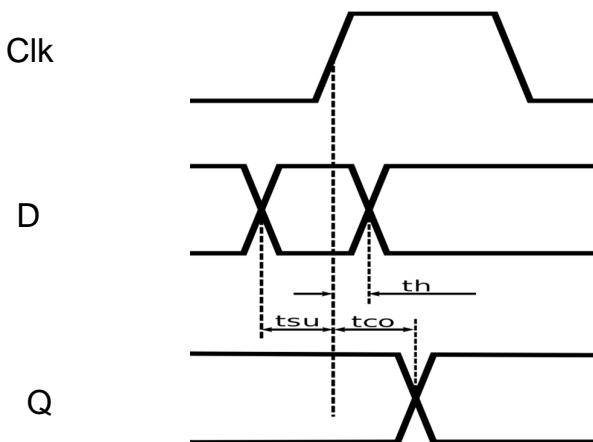
Mit aktivem Steuersignal 1 (d.h. mit der steigenden Flanke von Signal C) wird D übernommen.

## Flip-Flops und Taktsignale

- Eingang C oder Clk heißt **Takt** (engl. **Clock**), da er die Übernahme des Eingangs D in das Flipflop zu einem festgelegten Zeitpunkt (der steigenden oder positiven Taktflanke) synchronisiert.  
(→ „**Synchrone Schaltungen**“).
- Das Taktsignal ist eine periodische Rechteckschwingung mit fester Taktfrequenz  $f_{\text{clk}}$ , die meist von einem Oszillator generiert wird.
- Es gilt: Periodendauer einer Schwingung  $T = 1/f_{\text{clk}}$
- *Beispiel:*  $T = 20 \text{ ns} = 1 / (50 \text{ MHz})$
- Damit ein D-FF korrekt funktioniert, darf sich das Eingangssignal für die Dauer  $t_{\text{setup}}$  (Setup-Zeit) **vor der Taktflanke** und  $t_{\text{hold}}$  (Hold-Zeit) **nach der Taktflanke** nicht ändern. Zwischen Setup- und Hold-Zeit ist das „Zeitfenster für gültige Eingangssignale“ („Data-Valid-Window“).

**Hinweis:** Oft spricht man bei flankengesteuerten Flipflops von der „**aktiven Flanke**“, da manche Flip-Flops die Daten auch mit der fallenden Flanke übernehmen.

# Setup- und Hold-Zeiten als Grafik



$t_{su}$ :  
 $t_h$ :

Setup-Zeit (wie lange vor der Flanke muss der Eingang stabil sein)  
 Hold-Zeit (wie lange nach der Flanke muss der Eingang stabil bleiben)

$t_{co}$ :

„Clock-to-Q“-Delay (wie lange nach der Flanke erscheint das neue Ausgangssignal)

## Varianten flankengesteuerter D-FFs

Zusätzliche Steuereingänge flankengesteuerter FlipFlops:

**Reset:** Rücksetzen des FlipFlops auf den Ausgangswert.

Dazu existieren zwei Möglichkeiten:

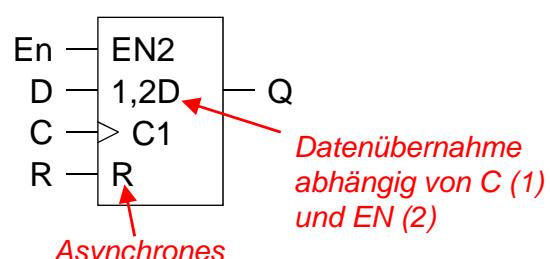
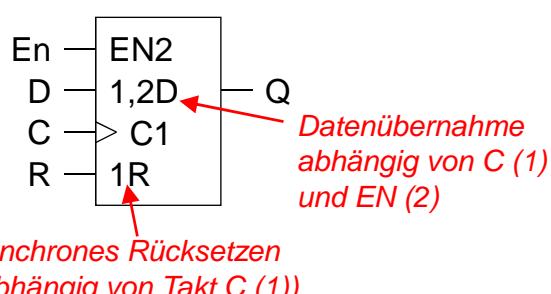
- Synchrones Rücksetzen mit der aktiven Taktflanke,
- Asynchrones Rücksetzen sobald das Reset-Signal aktiv ist.

**Enable:** Freigabe des FlipFlops.

Nur mit aktivem Freigabesignal verändert sich der Wert des FlipFlops entsprechend den Dateneingängen (z.B. R,S oder D).

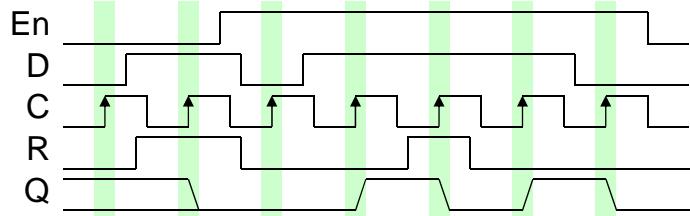
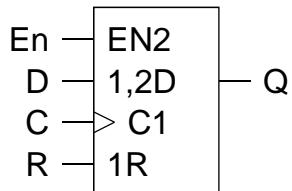
Das Freigabesignal wird bei der aktiven Taktflanke ausgewertet.

Schalsymbole für D-FlipFlop mit Freigabe und (a)synchronem Rücksetzen

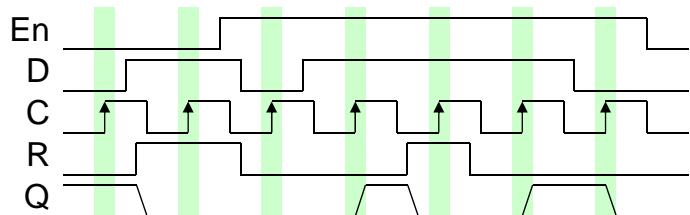
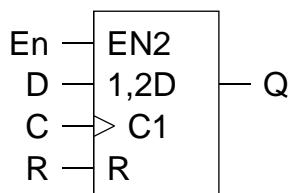


## Varianten flankengesteuerter D-FFs II

**Beispiel:** D-FlipFlop mit synchronem Reset und mit Enable

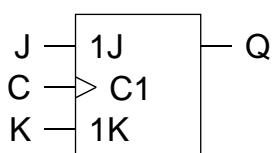


**Beispiel:** D-FlipFlop mit asynchronem Reset und mit Enable

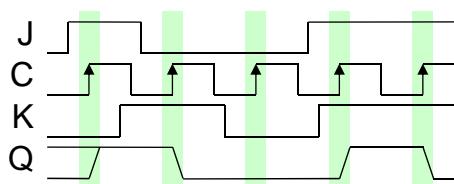


## Weitere Flip-Flops

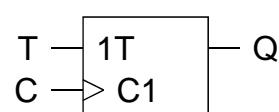
**JK-FF: „Jump-Kill“-FlipFlop**



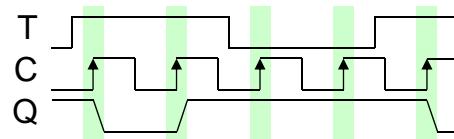
J	K	C	Q
*	*	0	$Q^*$
*	*	1	$Q^*$
0	0	$\uparrow$	$Q^*$
0	1	$\uparrow$	0
1	0	$\uparrow$	1
1	1	$\uparrow$	$\overline{Q^*}$



**T-FF: „Toggle“-FlipFlop**

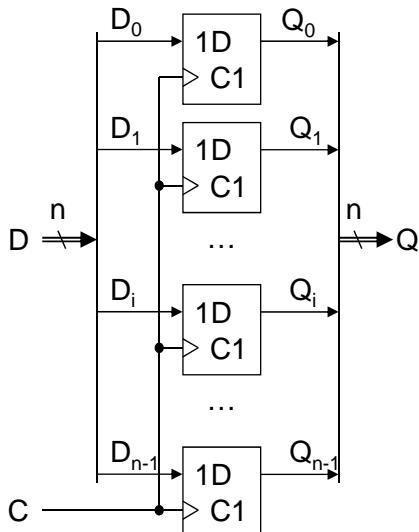


T	C	Q
*	0	$Q^*$
*	1	$Q^*$
0	$\uparrow$	$Q^*$
1	$\uparrow$	$\overline{Q^*}$



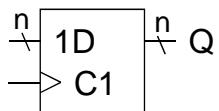
# Register: Wortspeicher mit D-FFs

Register ergeben sich durch Zusammenfassen von FlipFlops zu einem Wort-Speicher



- Das Register besitzt einen gemeinsamen Takt für alle angeschlossenen FlipFlops.
- Ein- und Ausgänge werden zu Vektoren zusammengefasst.
- Wird normalerweise als generische Komponente bereitgestellt, bei der die benötigte Wortbreiten ( $n$ ) bei der Instanzierung angepasst wird.
- Ermöglicht Speicherung von Zahlen etc.
- Mit zusätzlichem Enable-Eingang kann Speicherdauer gesteuert werden.

Kompaktes Schaltsymbol:



## Zusammenfassung: Schaltnetz und Schaltwerk

- ▶ **Schaltalgebra**
  - Spezielle Ausprägung der Booleschen Algebra
  - Hilfsmittel zur Berechnung und Beschreibung von Schaltnetzen und Schaltwerken
  - Kennt nur die Schalterzustände geöffnet und geschlossen
- ▶ **Schaltnetz**
  - Ausgänge hängen nur von den Eingängen ab
  - Schaltungstechnische Realisierung einer booleschen Funktion
  - Verwendung von UND-, ODER- und NICHT-Gattern
  - Synonymer Begriff: Kombinatorische Schaltung
  - *Beispiele:* Multiplexer, Addierer
- ▶ **Schaltwerk**
  - Rückkopplung von Signalen an den Eingang
  - Ausgänge hängen von den Eingängen und dem vorherigen Zustand ab  
➔ Schaltwerk hat ein "Gedächtnis"
  - Schaltwerk heißt synchron, wenn die Eingänge und Rückkopplungen durch Takte signaliert sind, andernfalls heißt es asynchron.
  - In Schaltwerken kommen zusätzlich Flipflops zum Einsatz.
  - Synonyme Begriffe: Sequentielle Schaltung, Automat
  - *Beispiele:* Register, Zähler

## 8. Komplexe Logikbausteine

**Programmierbare UND/ODER-Bausteine  
FPGA Bausteine  
ASICs**

---

**Programmierbare UND/ODER-Bausteine**



# Programmierbare UND/ODER-Bausteine I

## Kennzeichen:

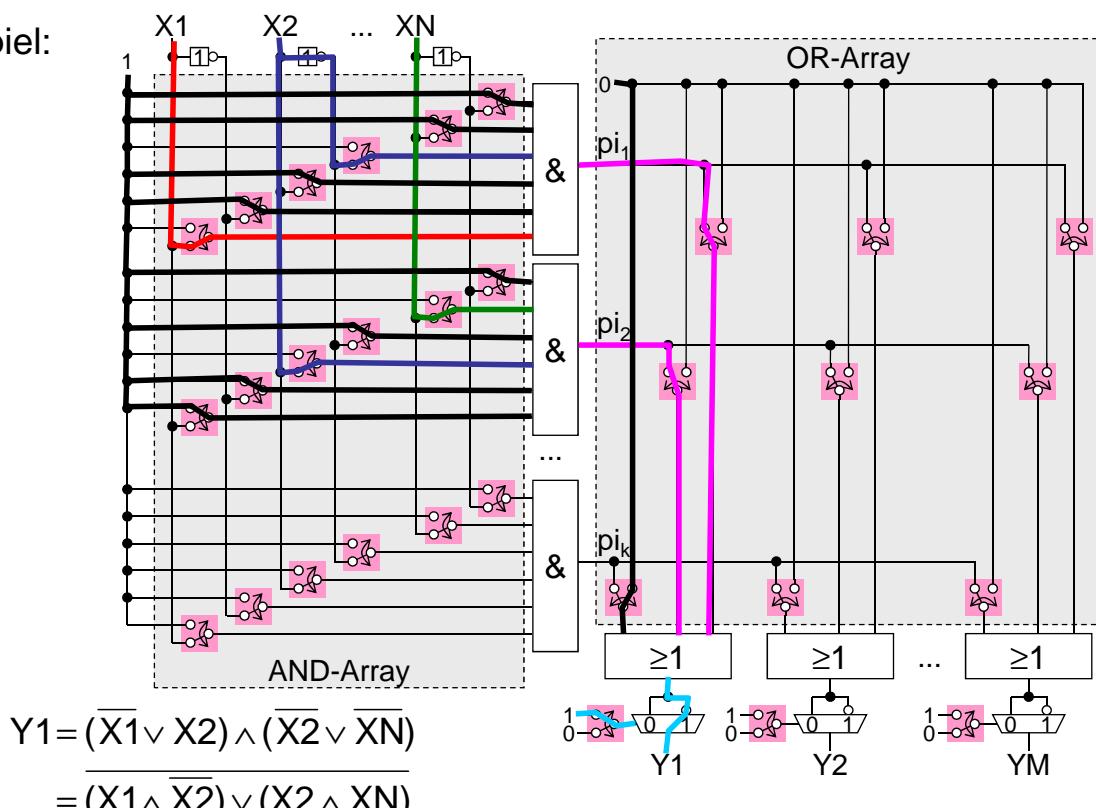
- Programmierbares UND-Feld (AND-Array), welches die Eingangssignale direkt oder invertiert mit den UND-Gattern der ersten Schicht verbindet
- Programmierbares ODER-Feld (OR-Array), welches die Produktterme der ersten Schicht direkt mit den ODER-Gattern der zweiten Schicht verbindet
- Programmierbar zuschaltbare Inverter an den Ausgängen

## Eigenschaften:

- Unmittelbare Abbildung disjunktiver Gleichungen auf die UND/ODER-Struktur
- Direkte Abbildung konjunktiver Gleichungen auf die UND/ODER-Struktur nach Umformung mittels Shannonschem Gesetz
- Komplexität der Terme durch Breite der UND- und ODER-Gatter beschränkt
- Einmal-Programmierung durch "Anti-Fuses": Durchbrennen der Isolator-"Anti-Sicherung" schafft eine Verbindung!

# Programmierbare UND/ODER-Bausteine II

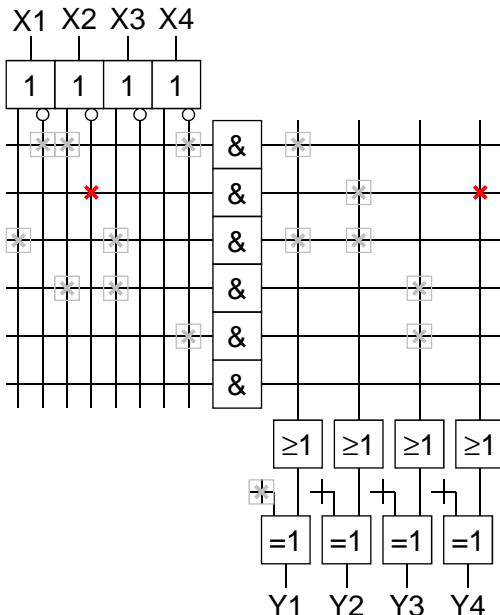
Beispiel:



# Kompakte Darstellung von UND/ODER-Bausteinen

Alle Gattereingänge werden durch eine einzelne Linie symbolisiert.

Der Anschluss eines Eingangs an ein Gatter wird durch eine Markierung auf der Linie spezifiziert



Beispiel:

$$Y_1 = (X_1 \vee \overline{X_2} \vee X_4) \wedge (\overline{X_1} \vee \overline{X_3}) \\ = (\overline{X_1} \wedge X_2 \wedge \overline{X_4}) \vee (X_1 \wedge X_3)$$

$$Y_2 = \overline{X_2} \vee (X_1 \wedge X_3)$$

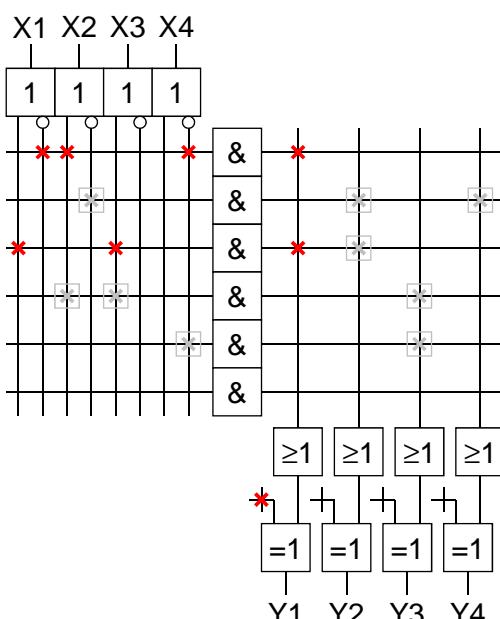
$$Y_3 = (X_2 \wedge X_3) \vee \overline{X_4}$$

$$Y_4 = \overline{X_2}$$

# Kompakte Darstellung von UND/ODER-Bausteinen

Alle Gattereingänge werden durch eine einzelne Linie symbolisiert.

Der Anschluss eines Eingangs an ein Gatter wird durch eine Markierung auf der Linie spezifiziert



Beispiel:

$$Y_1 = (X_1 \vee \overline{X_2} \vee X_4) \wedge (\overline{X_1} \vee \overline{X_3}) \\ = (\overline{X_1} \wedge X_2 \wedge \overline{X_4}) \vee (X_1 \wedge X_3)$$

$$Y_2 = \overline{X_2} \vee (X_1 \wedge X_3)$$

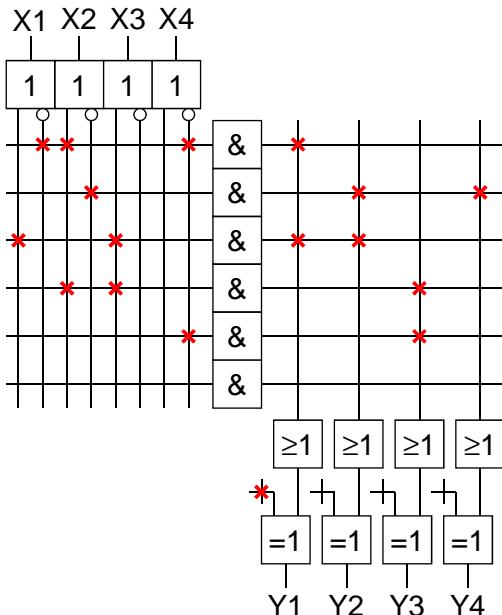
$$Y_3 = (X_2 \wedge X_3) \vee \overline{X_4}$$

$$Y_4 = \overline{X_2}$$

# Kompakte Darstellung von UND/ODER-Bausteinen

Alle Gattereingänge werden durch eine einzelne Linie symbolisiert.

Der Anschluss eines Eingangs an ein Gatter wird durch eine Markierung auf der Linie spezifiziert



Beispiel:

$$Y_1 = (X_1 \vee \overline{X_2} \vee X_4) \wedge (\overline{X_1} \vee \overline{X_3}) \\ = \overline{(X_1 \wedge X_2 \wedge X_4)} \vee (X_1 \wedge X_3)$$

$$Y_2 = \overline{X_2} \vee (X_1 \wedge X_3)$$

$$Y_3 = (X_2 \wedge X_3) \vee \overline{X_4}$$

$$Y_4 = \overline{X_2}$$

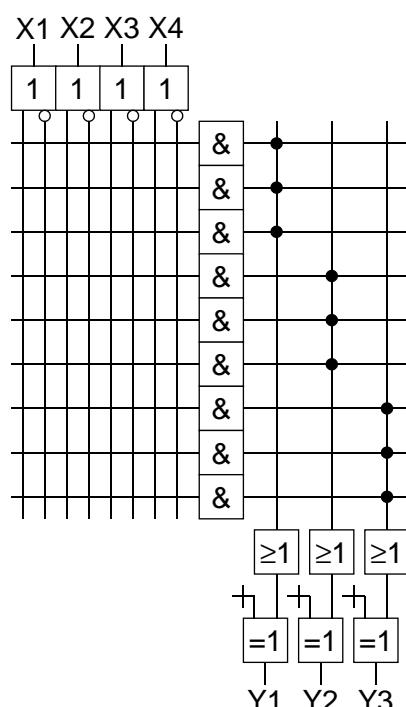
# Konzept der PAL-Bausteine

**PAL = Programmable Array Logic**

ODER-Feld fest vorprogrammiert

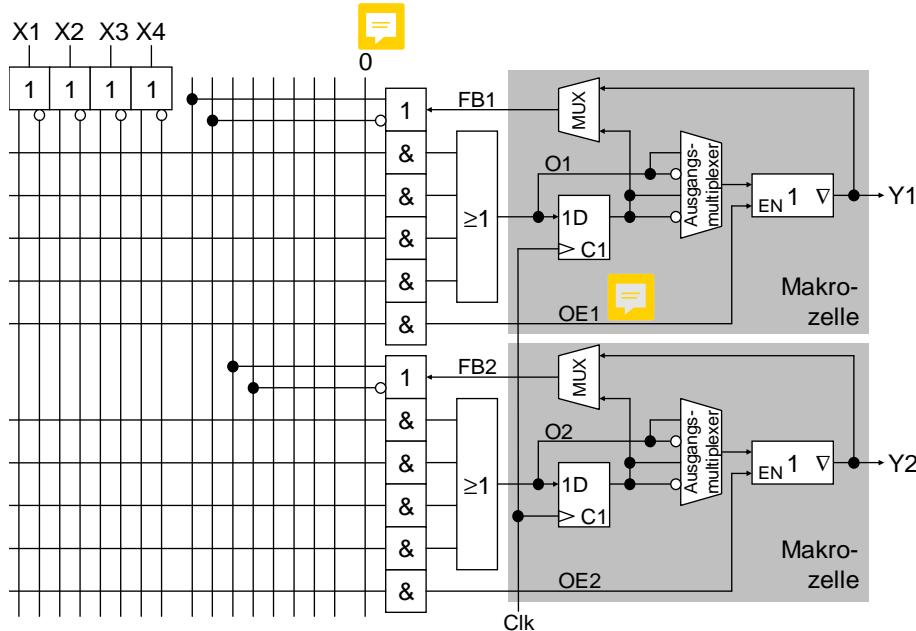
UND-Feld kann programmiert werden

Jeder UND-Term der ersten Stufe ist damit fest einem ODER-Gatter und somit einem einzigen Ausgang zugeordnet



# Reale PAL-Bausteine

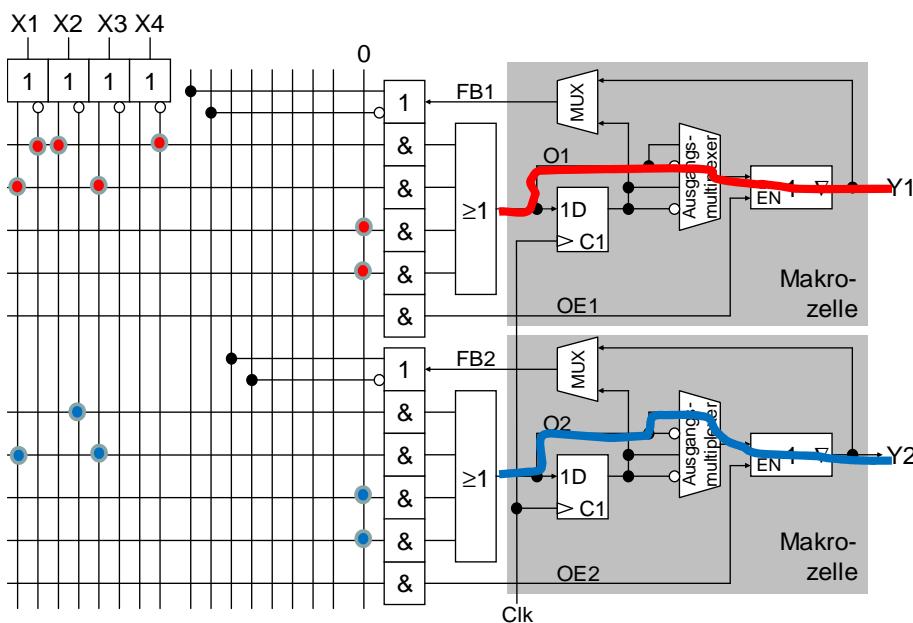
- Programmierbare Makrozellen zur Invertierung und Speicherung der Ausgangssignale
- Ausgangssignale werden in das programmierbare AND-Array zurückgeführt
- Ausgänge können deaktiviert (tri-state) und der Pin als Eingang verwendet werden
- *Man beachte:* Nicht verbundene Eingänge haben automatisch den Wert 1!



## Beispiel für Schaltnetz in PAL

$$\begin{aligned} Y_1 &= (X_1 \vee \overline{X_2} \vee X_4) \wedge (\overline{X_1} \vee \overline{X_3}) \\ &= (\overline{X_1} \wedge X_2 \wedge \overline{X_4}) \vee (X_1 \wedge X_3) \end{aligned}$$

$$Y_2 = \overline{X_2} \vee (X_1 \wedge X_3)$$

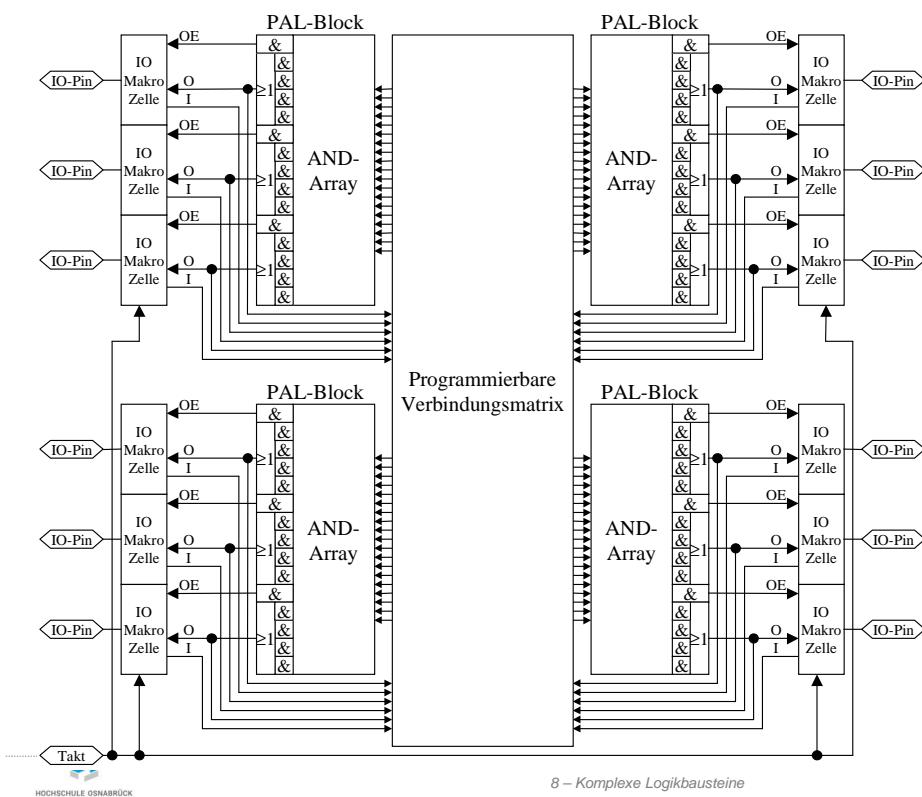


**Hinweis:** Nicht-verbundene Eingänge der UND-Gatter haben automatisch den Wert 1!

# CPLD-Bausteine



## CPLD = Complex Programmable Logic Device



**Mehrere UND/ODER-Blöcke werden über eine programmierbare Verbindungsmaatrix zusammengeschaltet**

Die Verbindungsmaatrix kann beliebige Verbindungen zwischen ihren Ein- und Ausgängen mit sehr kurzer Verzögerungszeit realisieren

Die IO-Zellen besitzen eine ähnliche Struktur wie die Makrozellen der PAL-Bausteine.

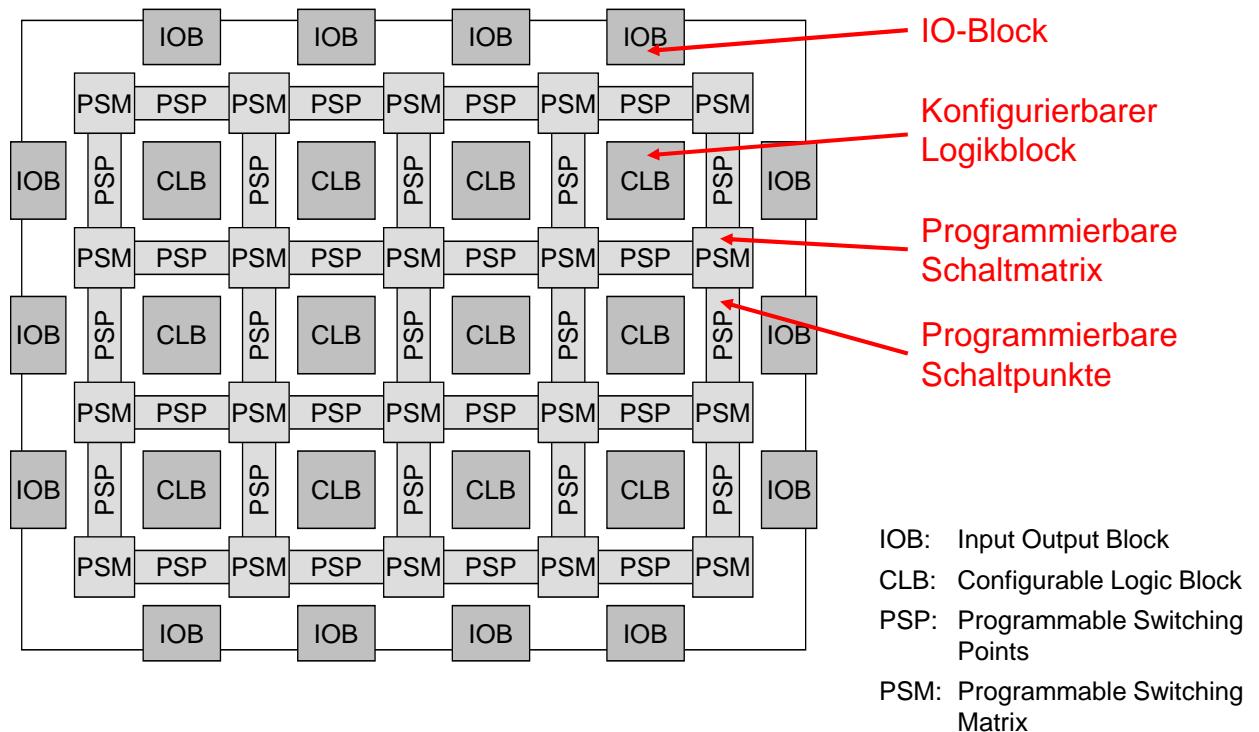
11

# FPGA-Bausteine



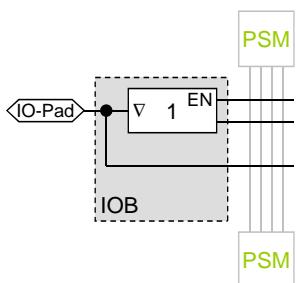
# FPGA-Bausteine

## FPGA = Field Programmable Gate Array



## Beispiel-FPGA

### Beispielhafter IO-Block

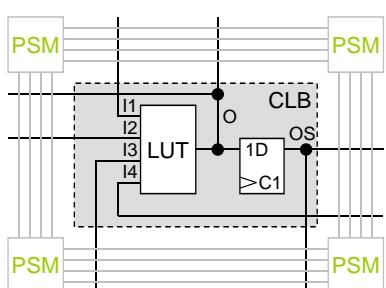


Signale können über Tri-State Bustreiber an die IO-Pins geschaltet werden.

Der Wert des IO-Pins wird auch in das FPGA hineingeführt.

Über PSPs wird der IOB an die Leitungssegmente angeschlossen.

### Beispielhafter CLB



Von jeder Seite wird ein Signal in die LUT des CLBs geführt.

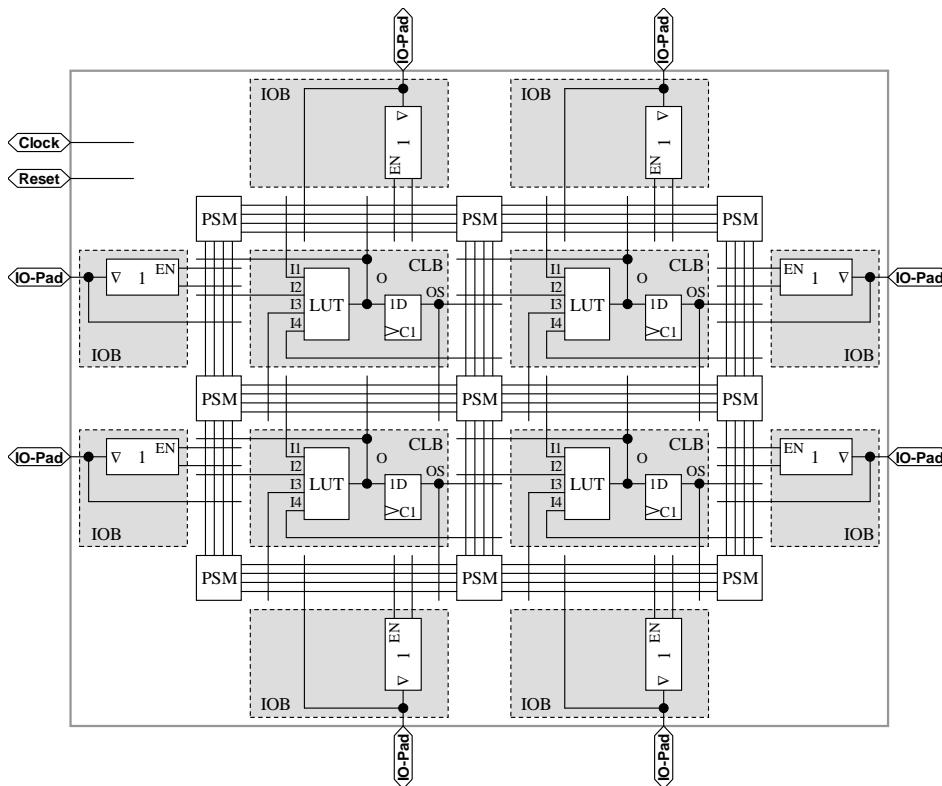
Das Ausgangssignal der LUT kann direkt oder über ein FlipFlop geführt ausgegeben werden.

Über PSPs werden die Ein- und Ausgänge des CLBs an die Leitungssegmente angeschlossen.

LUT ("Look-Up" Tabelle) ist 1-Bit breiter Speicher, d. h. für jede Eingangskombination kann gewünschter Wert gespeichert werden → LUT realisiert beliebige boolesche Funktion!

Programmierung der LUTs, PSMs etc. meistens flüchtig (RAM)!

# Beispiel-FPGA II



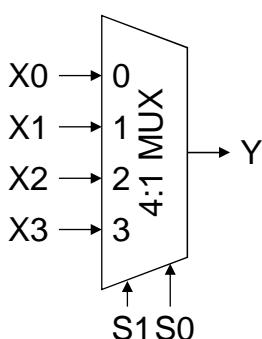
## Programmierbare Ressourcen:

- **LUT**  
Funktion der CLBs
- **PSP**  
CLB-Anschlüsse mit Leitungssegmenten verbinden
- **PSM**  
Leitungssegmente zu gemeinsamer Leitung verbinden

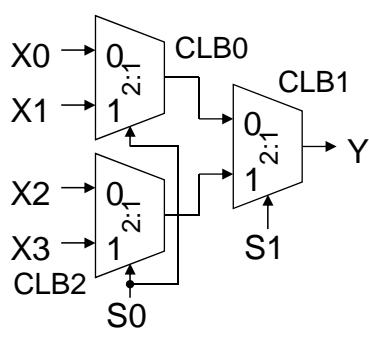
Vielfältige Möglichkeiten  
für die Realisierung  
digitaler Schaltungen

# Beispiel: Multiplexer-Realisierung mit FPGA

Gewünschte Funktion



Zerlegung in Einheiten mit maximal 4 Eingängen

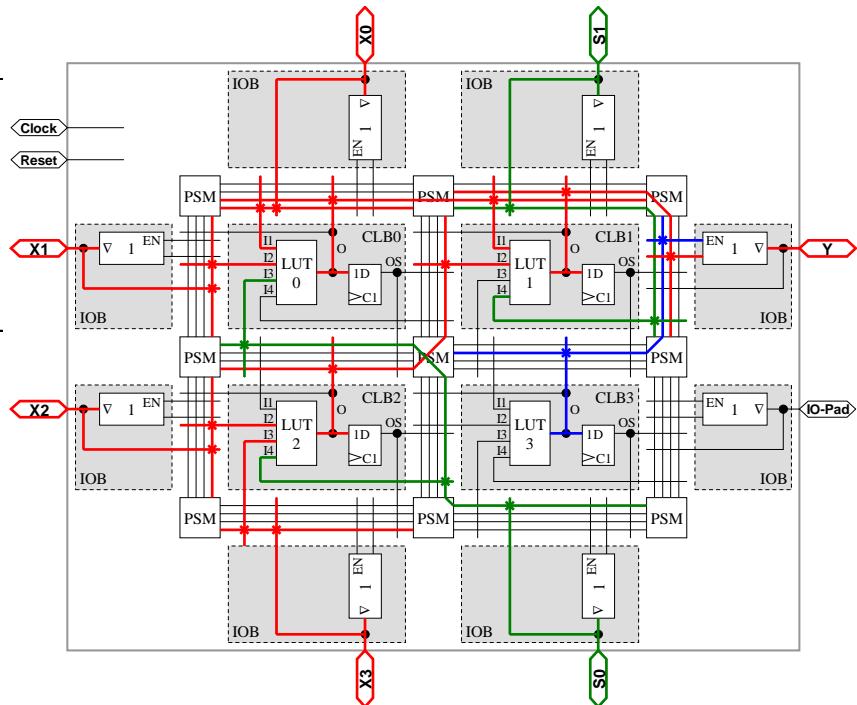
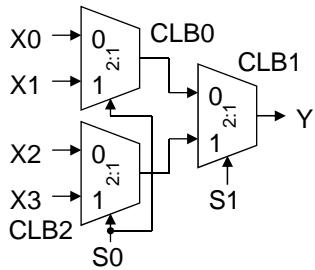


- Jede Funktionseinheit wird auf ein CLB abgebildet.
- Ein- und Ausgänge werden über IOBs mit der Außenwelt verbunden.
- Signale zwischen den CLBs und zu den IOBs werden mit PSPs an Leitungssegmente angeschaltet.
- Die Leitungssegmente werden mittels PSMs zu gewünschten Signalleitungen zusammengeschaltet.

# Beispiel: Multiplexer-Realisierung mit FPGA II

LUT0				LUT1					
I1	I2	I3	I4	O	I1	I2	I3	I4	O
0	-	0	-	0	0	-	-	0	0
1	-	0	-	1	1	-	-	0	1
-	0	1	-	0	-	0	-	1	0
-	1	1	-	1	-	1	-	1	1

LUT2				LUT3					
I1	I2	I3	I4	O	I1	I2	I3	I4	O
-	0	-	0	0	-	-	-	-	1
-	1	-	0	1	-	-	-	-	-
-	-	0	1	0	-	-	-	-	-
-	-	1	1	1	-	-	-	-	-



# Beispiel: Xilinx Artix-7 FPGA-Familie

## Artix-7 FPGA Feature Summary

Quelle: Xilinx 7 Series FPGAs Overview (DS180)

Table 2: Artix-7 FPGA Feature Summary by Device

Device	Logic Cells	Configurable Logic Blocks (CLBs)		DSP48E1 Slices <sup>(1)</sup>	Block RAM Blocks <sup>(3)</sup>			CMTs <sup>(4)</sup>	PCIe <sup>(5)</sup>	GTPs	XADC Blocks	Total I/O Banks <sup>(6)</sup>	Max User I/O <sup>(7)</sup>
		Slices <sup>(1)</sup>	Max Distributed RAM (Kb)		18 Kb	36 Kb	Max (Kb)						
XC7A15T	16,640	2,600	200	45	50	25	900	5	1	4	1	5	250
XC7A35T	33,280	5,200	400	90	100	50	1,800	5	1	4	1	5	250
XC7A50T	52,160	8,150	600	120	150	75	2,700	5	1	4	1	5	250
XC7A75T	75,520	11,800	892	180	210	105	3,780	6	1	8	1	6	300
XC7A100T	101,440	15,850	1,188	240	270	135	4,860	6	1	8	1	6	300
XC7A200T	215,360	33,650	2,888	740	730	365	13,140	10	1	16	1	10	500

Notes:

- Each 7 series FPGA slice contains four LUTs and eight flip-flops; only some slices can use their LUTs as distributed RAM or SRLs.
- Each DSP slice contains a pre-adder, a  $25 \times 18$  multiplier, an adder, and an accumulator.
- Block RAMs are fundamentally 36 Kb in size; each block can also be used as two independent 18 Kb blocks.
- Each CMT contains one MMC and one PLL.
- Artix-7 FPGA Interface Blocks for PCI Express support up to x4 Gen 2.
- Does not include configuration Bank 0.
- This number does not include GTP transceivers.

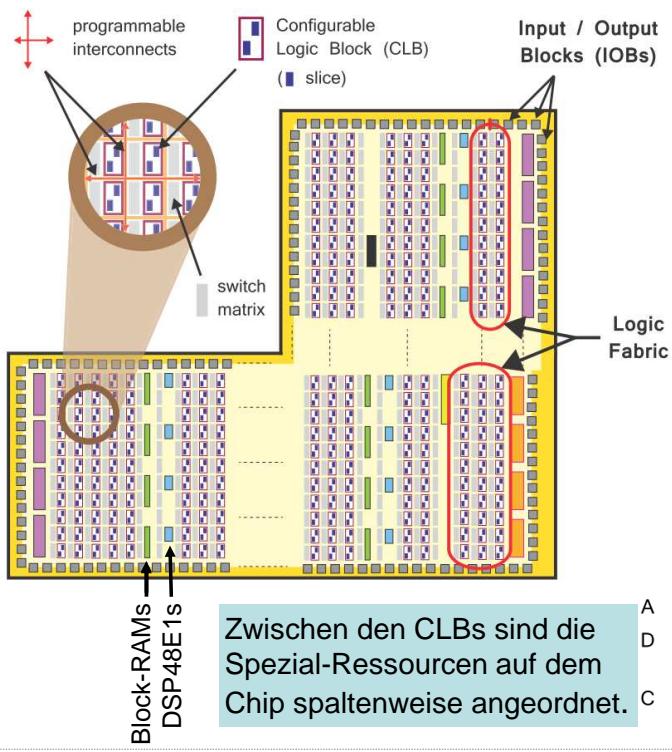
XC7A35T wird im Basys 3 Board verwendet!

## Ressourcen:

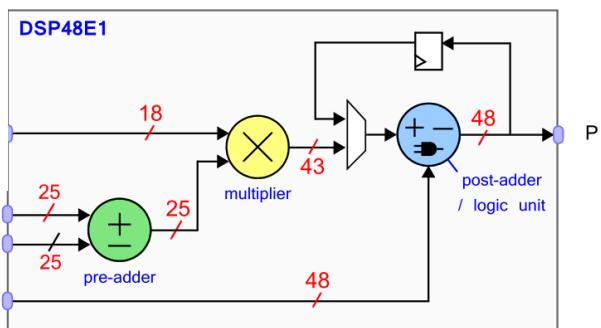
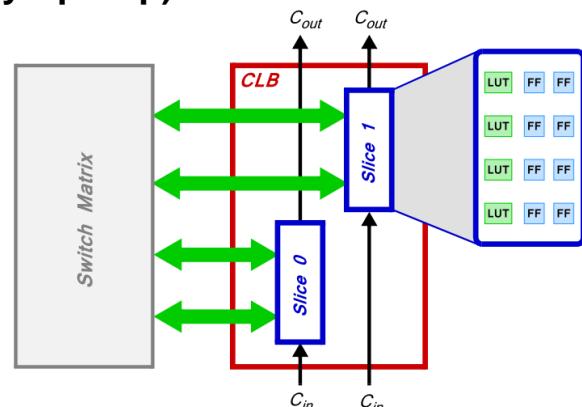
- IO-Blöcke mit FlipFlops
- Logik mit 6-Input LUTs
- FlipFlops
- Lokale Leitungssegmente
- Mittellange Leitungen
- Lange Verbindungsleitungen
- Logik für schnelle Addierer
- DSP-Blöcke (Multiplizierer + 48-Bit-Addierer)
- Verteilter Speicher
- Blockspeicher
- Mehrere Taktbereiche
- PCI-Express-Schnittst.
- Gigabit Transceiver
- Analog/Digital-Wandler

# Beispiel: Xilinx Artix-7 FPGA-Familie (Z-7010)

## ► Aufbau der FPGA-Logik (in einem Zynq-Chip)



Quelle: "The Zynq-Book"



# ASICs

# ASIC-Entwurf



## ASIC: Application Specific Integrated Circuit

Ähnliches Grundprinzip wie eine Realisierung aus Logikbausteinen

Entscheidender Unterschied: Platzierung und Verdrahtung auf dem Chip nicht auf einer Platine

## Die Logikschaltung wird aus vorhandenen Bibliothekselementen aufgebaut

Grundgatter (NAND, NOR, AND, OR, EXOR) mit unterschiedlichen Anzahlen von Eingängen

Komplexere Grundelemente (z.B. Volladdierer)

Flip-Flops

Speicher (RAM, ROM, Flash)

## Hohe Flexibilität

Anzahl und Platzierung der Grundelemente auf dem Chip ist frei wählbar

Die Verdrahtung der Grundelemente ist frei wählbar

Begrenzung durch die Anzahl der verfügbaren Verdrahtungslagen

## Eigenschaften eines ASIC-Designs

### Optimierte Schaltung mit hoher „Logikdichte“

Direkte Implementierung der Grundfunktionen (kein „Umweg“ über LUT)

Freie Auswahl der Grundelemente

Optimierung der Verdrahtung

→ schneller und kleiner als äquivalente Schaltung in FPGAs!

### Hohe Initialkosten

Entwurfsprozess ist deutlich aufwändiger als für ein FPGA

Ein Fehler im Design kann teuer werden (u.U. 8-stellige Euro-Beträge)

### Kostengünstige Produktion bei hohen Stückzahlen

> 100 k sollten es i.d.R. schon sein...



# **9. Zahlendarstellung, Arithmetik und Zeichencodierung**

**Einführung**

**Positive ganze Zahlen**

**Negative ganze Zahlen**

**Weitere Binärcodes**

**Festkomma-Zahlen**

**Gleitkomma-Zahlen**

**Addierer-Varianten**

**Arithmetisch-Logische Einheit (ALU)**

**Multiplizierer**

**Zeichencodierung**

## **Einführung**

---

Zahlen werden durch Ziffern (aus einem Symbolvorrat) dargestellt:

**Römische Zahlen**

Symbolvorrat: I, V, X, L, C, D, M

**Dezimalzahlen**

Symbolvorrat: 0, 1, 2, 3, 4, 5, 6, 7, 8 ,9

**Dualzahlen**

Symbolvorrat: 0, 1

# Additions- und Stellenwertsysteme

Mit Hilfe einzelner Ziffern lässt sich nur ein begrenzter Zahlenbereich darstellen.

Für die Darstellung eines größeren Zahlenbereichs werden Ziffern zu Ziffernfolgen kombiniert.

Neben dem Ziffernvorrat müssen Regeln vereinbart werden, die beschreiben, wie der dargestellte **Wert** aus der **Ziffernfolge** berechnet wird.

## Additionssystem

Römische Zahlen: I, II, III, IV, ..., IIX, IX, X, ..., MMIX, ...

Werte werden prinzipiell aufaddiert, bei niedrigen Ziffern vor höherwertigen aber subtrahiert.

## Stellenwertsystem

Jeder Position einer Ziffer wird eine Wertigkeit (Stellengewicht) zugeordnet.

Die Summe der Produkte der einzelnen Werte aus Zifferngewicht und Stellengewicht ergibt den Wert der dargestellten Zahl.

# Polyadische Stellenwertsysteme

Bei **polyadischen Stellenwertsystemen** hängt das Stellengewicht von der Anzahl der zur Verfügung stehenden Ziffern  $n_z$  ab.

$n_z$  wird üblicherweise als Basis oder **Radix R** bezeichnet. Es gilt:  $R \geq 2!$

Wert der Zahl A:  $V(A) = \sum_{i=0}^{n-1} a_i \cdot R^i = a_0 \cdot R^0 + a_1 \cdot R^1 + \dots + a_{n-1} \cdot R^{n-1}$   
(Summenformel)

mit  $A = (a_{n-1} a_{n-2} \dots a_1 a_0)$  Ziffernfolge  
 $a_i$  ( $0 \leq a_i < R$ ) Zifferngewicht  
 $R^i$  Stellengewicht

**Hinweis:** Bei polyadischen Stellenwertsystemen entspricht jedem Wert eine eindeutige Ziffernfolge A!

**Beispiel Dezimalsystem ( $n_z = R = 10$ ):**

$$(a_{n-1} a_{n-2} \dots a_1 a_0) = (1 2 3 4)$$

$$A = \sum_{i=0}^{n-1} a_i \cdot R^i = \sum_{i=0}^{n-1} a_i \cdot 10^i$$

$$V(A) = 1 \cdot 1000 + 2 \cdot 100 + 3 \cdot 10 + 4 \cdot 1$$

---

# Positive ganze Zahlen

## Addition positiver ganzer Zahlen

---

# Verbreitete Zahlensysteme in der Computertechnik

## Dualzahlen

Direkte Verwendung der Logikzustände zur Darstellung einer Ziffer

Ziffernvorrat: 0, 1

Radix: 2

## Oktalzahlen

Kombination von 3 Logikzuständen zur Darstellung einer Ziffer

Ziffernvorrat: 0, 1, 2, 3, 4, 5, 6, 7

Radix: 8

## Hexadezimalzahlen (oft durch den Präfix *0x* gekennzeichnet)

Kombination von 4 Logikzuständen zur Darstellung einer Ziffer

Ziffernvorrat: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Radix: 16

## Kennzeichnung des verwendeten Zahlensystems

- Ziffernfolge in Klammern setzen, Index (Suffix) gibt den Radix an, z. B.  $(25)_8$
- Kein Index: R=10 bzw. R ist aus dem Zusammenhang eindeutig festgelegt

# Umwandlung zwischen Zahlensystemen

Statt der direkten Umwandlung von Radix R<sub>1</sub> in Radix R<sub>2</sub> ist immer der „Umweg“ über das Dezimalsystem möglich!

## Umwandlung: Darstellung mit Radix R → Dezimalsystem

Direkte Verwendung der **Summenformel** oder Verwendung des **Hornerschemas**

$$A = \sum_{i=0}^{n-1} a_i \cdot R^i = (\dots((a_{n-1} \cdot R + a_{n-2}) \cdot R + a_{n-3}) \cdot R \dots + a_1) \cdot R + a_0$$

Beispiele ( $\rightarrow : \cdot R$ )

$$(100110)_2 =$$

$$1 \cdot 32 + 0 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 = 38$$

$$\begin{array}{r} 2 \ 4 \ 8 \ 18 \ 38 \\ \diagup \ \diagup \ \diagup \ \diagup \\ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \\ \hline +: 1 \ 2 \ 4 \ 9 \ 19 \ 38 \end{array} \quad R=2$$

$$(3AC)_{16} =$$

$$3 \cdot 256 + 10 \cdot 16 + 12 \cdot 1 = 940$$

$$\begin{array}{r} 48 \ 928 \\ \diagup \ \diagup \\ 3 \ 10 \ 12 \\ \hline +: 3 \ 58 \ 940 \end{array} \quad R=16$$

# Umwandlung zwischen Zahlensystemen II

## Umwandlung: Dezimalsystem → Zahlendarstellung mit Radix R

$$A = \sum_{i=0}^{n-1} a_i \cdot R^i = (\dots((a_{n-1} \cdot R + a_{n-2}) \cdot R + a_{n-3}) \cdot R \dots + a_1) \cdot R + a_0 \quad (\text{Hornerschema})$$

Division durch R ergibt:

$$A / R = Q_0 = (\dots((a_{n-1} \cdot R + a_{n-2}) \cdot R + a_{n-3}) \cdot R \dots + a_1) \cdot R + a_0 \quad \text{Rest } a_0$$

**Sukzessive Berechnung der Ziffern durch Division:**

$$1. \quad A / R = Q_0 \quad \text{Rest } a_0$$

$$2. \quad Q_{i-1} / R = Q_i \quad \text{Rest } a_i \quad \text{für } n > i > 0 \quad (\text{bis } Q_i = 0)$$



**Das Ergebnis ist dann:** (a<sub>n-1</sub> a<sub>n-2</sub> ... a<sub>0</sub>)<sub>R</sub>

**Hinweis:** Soll statt vom Dezimalsystem von einem beliebigen Radix R<sub>x</sub> in Radix R umgerechnet werden, müssen die obigen Berechnungen im R<sub>x</sub>-System durchgeführt werden, was uns viel schwerer fällt!

# Umwandlung zwischen Zahlensystemen III

Beispiel: 179 in das Dualsystem wandeln

$179 / 2 = 89$	Rest 1	<span style="border: 1px solid red; padding: 2px;">Niederwertigstes Bit</span>       <span style="border: 1px solid red; padding: 2px;">Höchstwertigstes Bit</span>
$89 / 2 = 44$	Rest 1	
$44 / 2 = 22$	Rest 0	
$22 / 2 = 11$	Rest 0	
$11 / 2 = 5$	Rest 1	
$5 / 2 = 2$	Rest 1	
$2 / 2 = 1$	Rest 0	
$1 / 2 = 0$	Rest 1	

$179 = (\boxed{10110011})_2$

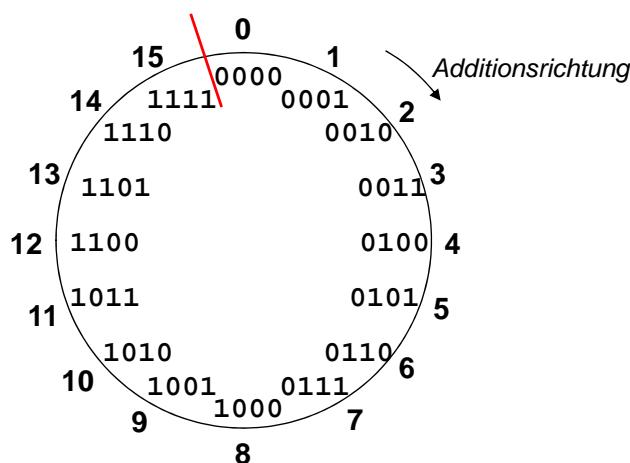
Dualzahlen können einfach in Oktal- und Hexadezimalzahlen umgewandelt werden, in dem 3 bzw. 4 Bits (beginnend mit dem niederwertigsten Bit) zusammengefasst werden. Ggf. müssen führende Nullen ergänzt werden.

$$\begin{array}{c} (\underline{\hspace{1cm}0\hspace{1cm}\hspace{1cm}1\hspace{1cm}\hspace{1cm}0\hspace{1cm}\hspace{1cm}0\hspace{1cm}\hspace{1cm}1\hspace{1cm}})_2 \\ = (2 \quad 6 \quad 3)_8 [= 2 \cdot 64 + 6 \cdot 8 + 3 \cdot 1] \end{array}$$

$$\begin{array}{c} (\underline{\hspace{1cm}1\hspace{1cm}\hspace{1cm}0\hspace{1cm}\hspace{1cm}1\hspace{1cm}\hspace{1cm}0\hspace{1cm}\hspace{1cm}1\hspace{1cm}})_2 \\ = (B \quad 3)_{16} [= 11 \cdot 16 + 3 \cdot 1] \end{array}$$

## Zahlenkreis positiver ganzer Zahlen

Beispiel: 4 bit



Bei der Addition zweier Zahlen wird der Zahlenkreis im Uhrzeigersinn durchlaufen

Wird hierbei der Übergang 1111 → 0000 durchlaufen, ist ein Überlauf aufgetreten

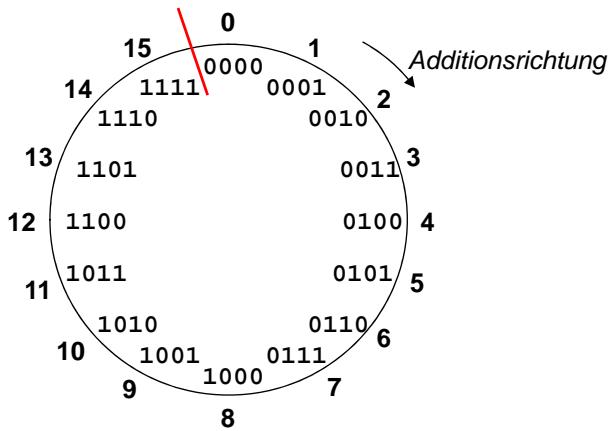
**Wertebereich:** 0 ...  $2^n - 1$

# Mehrfaches Durchlaufen des Zahlenkreises

Für eine feste, vorgegebene Wortbreite n  
(Ergebniswortbreite = Summandenwortbreite), gilt für eine Wortbreite n:

$$A = A + R^n = A + k \cdot R^n \quad k \in \{\dots -2, -1, 0, 1, 2, \dots\}$$

Anschaulich: Der Zahlenkreis wird bei der Addition von  $k \cdot R^n$  genau k-mal komplett ( $360^\circ$ ) durchlaufen



Anmerkung: Diese Erkenntnis ist insbesondere für die Darstellung negativer Zahlen hilfreich

## Addition: Beispiele

Die Addition von Dualzahlen erfolgt analog der Addition von Dezimalzahlen:

Stellenweise Addition, beginnend mit der niedrigwertigsten Ziffer  
Gegebenenfalls Übertrag in nächsthöherer Stelle berücksichtigen

Dezimal: Übertrag ist 1, wenn die Summe der Ziffern (incl. Übertrag aus vorangegangener Stelle) den Wert **9** überschreitet

Dual: Übertrag ist 1, wenn die Summe der Ziffern (incl. Übertrag aus vorangegangener Stelle) den Wert **1** überschreitet

Wortbreite: 4 bit

$$\begin{array}{r} 1001 \\ + 0011 \\ \hline 0011 \\ \hline 1100 \end{array} \quad \begin{array}{l} (9)_{10} \\ (3)_{10} \\ \hline (12)_{10} \end{array}$$

$$\begin{array}{r} 1101 \\ + 0110 \\ \hline 1100 \\ \hline 10011 \end{array} \quad \begin{array}{l} (13)_{10} \\ (6)_{10} \\ \hline (19)_{10} \end{array}$$

Ergebnis lässt sich mit 4 bit darstellen

Ergebnis lässt sich nicht mit 4 bit darstellen (Überlauf)

# Addition einzelner Bits in Hardware

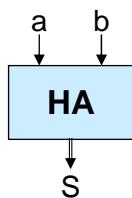
**Halbaddierer:** Addition von 2 Bits

**Volladdierer:** Addition von 3 Bits  
(3. Bit ist Übertrag)

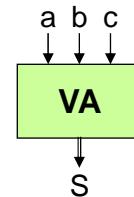
Ergebnis: 0,1,2

Ergebnis: 0,1,2,3

Das Ergebnis muss mit Hilfe von zwei Bits dargestellt werden!

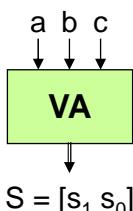


a	b	$S = [s_1 \ s_0]$
0	0	00
0	1	01
1	0	01
1	1	10

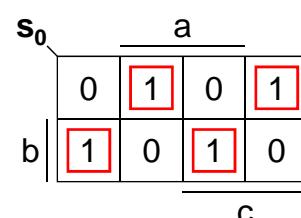
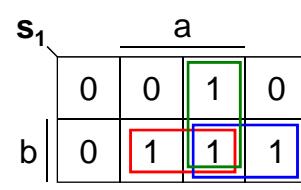


a	b	c	$S = [s_1 \ s_0]$
0	0	0	00
0	0	1	01
0	1	0	01
0	1	1	10
1	0	0	01
1	0	1	10
1	1	0	10
1	1	1	11

## Minimierung eines Volladdierers



a	b	c	S
0	0	0	00
0	0	1	01
0	1	0	01
0	1	1	10
1	0	0	01
1	0	1	10
1	1	0	10
1	1	1	11

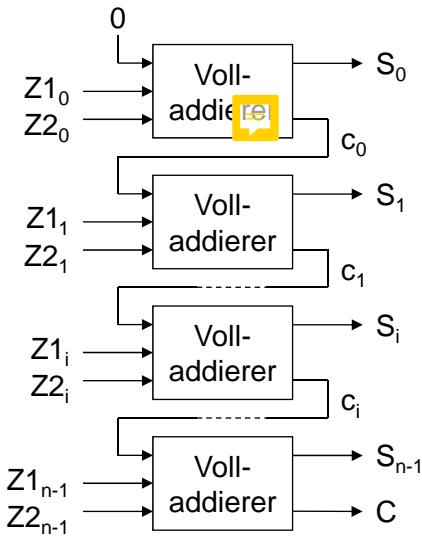


$$s_1 = (a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$$

$$s_0 = (\bar{a} \wedge b \wedge \bar{c}) \vee (a \wedge \bar{b} \wedge \bar{c}) \vee (a \wedge b \wedge c) \vee (\bar{a} \wedge \bar{b} \wedge c) = (a \oplus b) \oplus c$$

# Addierer für positive ganze Zahlen

Pro Stelle wird eine Summenziffer und ein Übertrag berechnet. Die Addition erfolgt mit Volladdierern, wie im folgenden Blockschaltbild gezeigt:



Verzögerungszeit:

$$T_{D,Add} = (n-1) \cdot T_{D,VA}$$

langsam...

Flächenaufwand:

$$A = O(n)$$



**Ripple-Carry-Addierer:** Carry-Bit „rieselt“ von oben nach unten.

# Negative ganze Zahlen

## Kombination Addierer/Subtrahierer

# Vorzeichen-Betrag-Darstellung I

## Übliche Zahlendarstellung im täglichen Leben:

Der Betrag des Zahlenwertes wird durch die Ziffernfolge angegeben

Ein Sonderzeichen vor der Zahl stellt das Vorzeichen des Zahlenwertes dar

„Plus“: Positiver Wert

„Minus“ Negativer Wert

Für eine mathematische Beschreibung des Zahlenwertes kann dem Vorzeichen  $s$  ein Zahlenwert zugeordnet werden:

„Plus“:  $s = 1$

„Minus“:  $s = -1$

Dann ergibt sich der Wert der Zahl zu:  $A = s \cdot \sum_{i=0}^{n-1} a_i \cdot 10^i$

### Beispiele:

$$+1234 = 1 \cdot (1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0)$$

$$-3711 = -1 \cdot (3 \cdot 10^3 + 7 \cdot 10^2 + 1 \cdot 10^1 + 1 \cdot 10^0)$$

# Vorzeichen-Betrag-Darstellung II

Die Trennung von Vorzeichen und Betrag kann auch für eine Zahlendarstellung mit binären Ziffern verwendet werden

## Übliche Vereinbarung

Für das Vorzeichen wird ein Bit  $s$  reserviert

Positiver Zahlenwert:  $s = 0$

Negativer Zahlenwert:  $s = 1$

## Wert einer n-bit breiten Zahl in Vorzeichen-Betrag-Darstellung

$s \ a_{n-2} \dots a_1 \ a_0$

$$A = (-1)^s \cdot \sum_{i=0}^{n-2} a_i \cdot 2^i$$

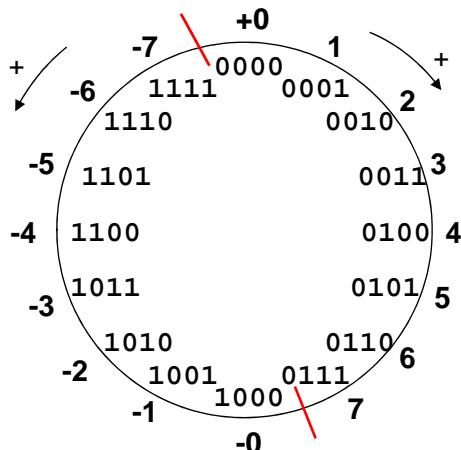
### Beispiele

$$0101 \rightarrow (5)$$

$$1101 \rightarrow (-5)$$

# Zahlenkreis für Vorzeichen-Betrag-Darstellung

Beispiel: 4 bit



## Nachteile:

Es existieren zwei „Nullen“

Es existieren zwei Stellen im Zahlenkreis, an denen ein Überlauf auftreten kann  
Additionsrichtung vorzeichenabhängig

Vergleich zu polyadischer Darstellung positiver Zahlen

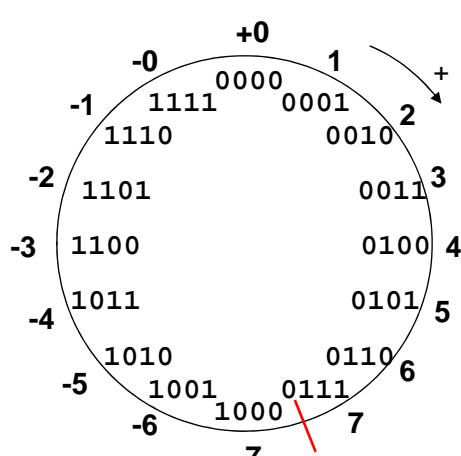
Positive Zahlen → Additionsrichtung = Uhrzeigersinn

Vorzeichen-Betrag-Darstellung → Additionsrichtung ist vorzeichenabhängig

# Einerkomplement

**Einerkomplement:** Negation einer Zahl durch Invertieren aller Bits

Beispiel: 4 bit



## Vorteile:

Nur eine Überlauf-Stelle

Nur eine Additionsrichtung

Symmetrischer Zahlenbereich

## Nachteile:

Da auch zwei „Nullen“ existieren, ist das Additionsergebnis um 1 zu klein, wenn die Null übersprungen wird. → Sonderfall, 1 dazuaddieren!



## Motivation: Radix-Komplement-Darstellung I

**Ziel:** Keine Fallunterscheidung bei der Addition (z.B. positive/negative Zahl),  
immer gleiche Additionsrichtung

Einfache Rechenregeln sollten ohne Fallunterscheidung Gültigkeit haben, z.B.:

$$A + (-A) = 0 \quad (1)$$

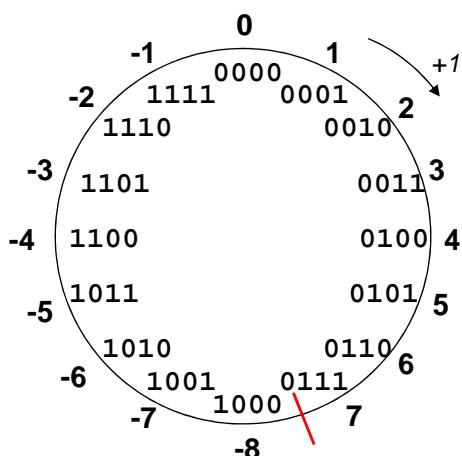
$$A + (-A) + 1 = 1 \quad (2)$$

Aus (1): Die Darstellung für  $-A$  ergibt sich, wenn man auf dem Zahlenkreis von Null aus  $A$  Schritte entgegen dem Uhrzeigersinn „geht“

Aus (2): Zwischen den Darstellungen für  $-1$  und  $1$  darf es nur eine Darstellung geben, die den Wert  $0$  repräsentiert

## Motivation: Radix-Komplement-Darstellung II

Bei diesem Zahlenkreis gibt es nur eine Null. Die Darstellung für  $-A$  ergibt sich, wenn man auf dem Zahlenkreis von Null aus  $A$  Schritte entgegen dem Uhrzeigersinn „geht“. Es gibt nur eine Stelle, an der ein Überlauf auftritt.



Bei  $n=2$  spricht man von der **Zweierkomplement-Darstellung**.

**Wertebereich:**  $-2^{n-1} \dots 2^{n-1} - 1$

**Nachteile:** - Überlauf nun an anderer Stelle im Zahlenkreis  
- Asymmetrischer Wertebereich

# Die Vorzeichenstelle im Zweierkomplement

Das MSB (most significant bit = höchstwertigstes Bit) einer Zahl in Zweierkomplementdarstellung gibt das Vorzeichen der Zahl an:

MSB = 0	positive Zahl
MSB = 1	negative Zahl

Kann man zur Berechnung des dargestellten Zahlenwertes auch eine Summenformel angeben ?

Ja, hierbei wird die Vorzeichenstelle negativ gewichtet, alle anderen Stellen positiv:

$$A = -a_{n-1} \cdot R^{n-1} + \sum_{i=0}^{n-2} a_i \cdot R^i$$

Beispiel (4 Bit): -1:  $(1111)_2 = -2^3 + 2^2 + 2^1 + 2^0 = -8 + 7 = -1$

## Bildung des Zweierkomplements

Zur Erinnerung: Bei identischer Wortbreite von Summanden und Summe gilt:

$$A = A + 2^n \quad (360^\circ \text{ im Zahlenkreis})$$

Bei fester Wortbreite  $n$  lassen sich die Ziffern des Zweierkomplements deshalb folgendermaßen berechnen:

$$-A = 0 - A = 2^n - A \quad (2^n \text{ Werte darstellbar} \rightarrow 2^n = 0)$$

Beispiel:  $A = 0101 \rightarrow -A = 1000 - 0101 = 1011$

**Praktische Vorgehensweise** zur Bildung des Zweierkomplements:

1. Alle Ziffernwerte  $a_i$  wird durch  $1-a_i$  ersetzt. (Einerkomplement!)
2. Die so erhaltene Zahl wird inkrementiert (+1 addiert)

$$(1-a_i) = \overline{a_i}$$

*entspricht Invertierung:*

$$\begin{array}{ll} a_i = 0: & (1-a_i) = 1 \\ a_i = 1: & (1-a_i) = 0 \end{array}$$

# Beispiele zur Bildung des Zweierkomplements

## Zweierkomplement

Invertieren:	0101	0101100
	1010	1010011
Inkrementieren:	1011	1010100

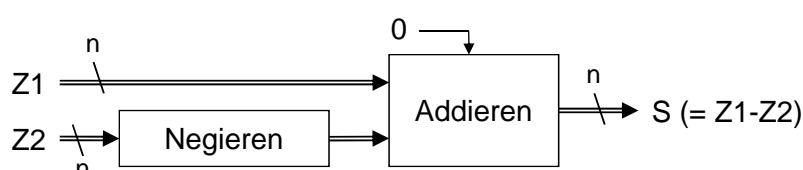
Kontrollrechnung  $A + (-A)$ :

0101	0101100
+ 1011	+ 1010100
0000	0000000

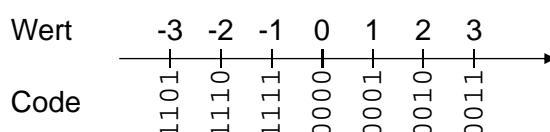
Dieses Verfahren ist in genau der gleichen Weise auch zur "Rückverwandlung" negativer Zahlen geeignet!

## Subtraktion ganzer Zahlen

Idee: Subtrahend negieren und dann zum Minuend hinzuaddieren



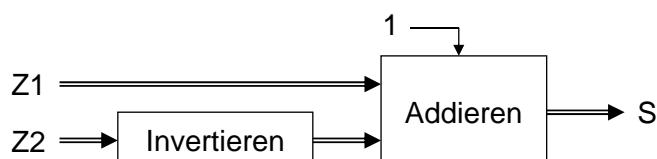
Wiederholung: Wie kann man eine 2er-Komplement Zahl negieren?



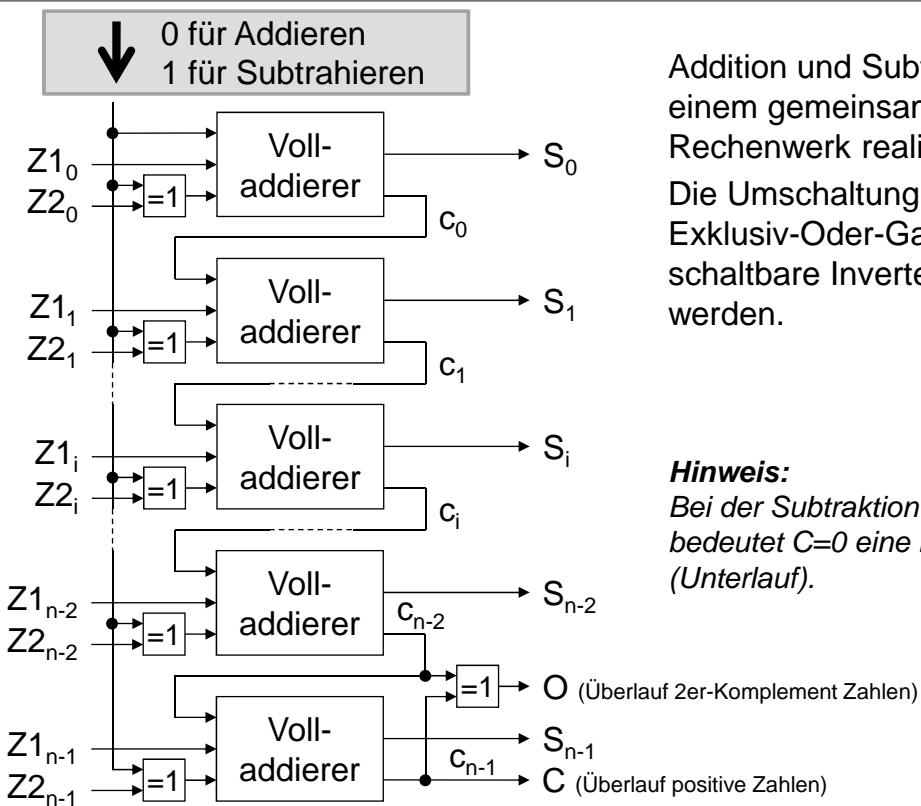
Antwort: Alle Bits invertieren und dann noch 1 addieren.

### Implementierung:

Alle Bits des Subtrahenden invertieren und zum Minuend hinzuaddieren. Dabei Übertrageingang des Addierers zu 1 setzen.



# Kombination Addierer/Subtrahierer



Addition und Subtraktion können in einem gemeinsamen, umschaltbaren Rechenwerk realisiert werden.

Die Umschaltung erfolgt mit Exklusiv-Oder-Gatter, welche als schaltbare Inverter verwendet werden.

### Hinweis:

Bei der Subtraktion positiver Zahlen bedeutet  $C=0$  eine Bereichsüberschreitung (Unterlauf).

## Detektion von Bereichsüberschreitungen (Addition)

### Regel:

Eine Bereichsüberschreitung ist aufgetreten, wenn die Vorzeichenbits (höchstwertige Bits)  $Z_{1_{n-1}}$  und  $Z_{2_{n-1}}$  der beiden Operanden gleich sind (0,1), das Vorzeichenbit  $S_{n-1}$  der Summe jedoch einen anderen Wert (1,0) aufweist.

$S_{n-1}=1 \rightarrow$  Überlauf

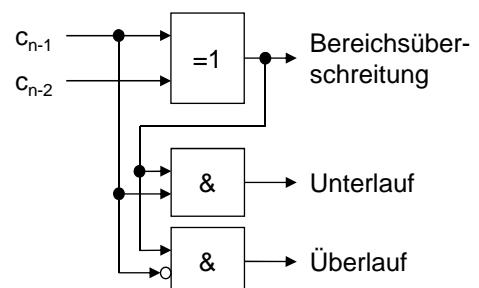
$S_{n-1}=0 \rightarrow$  Unterlauf

### Alternative Regel:

Sind die beiden höchstwertigen Übertragbits  $c_{n-1}$  und  $c_{n-2}$  ungleich ( $c_{n-1} \neq c_{n-2}$ ), ist eine Bereichsüberschreitung aufgetreten.

$c_{n-1}=0 \rightarrow$  Überlauf

$c_{n-1}=1 \rightarrow$  Unterlauf



# Bereichsüberschreitungen 2er Komplement, Vergleicher

## Addition

Z1	Z2	Bereichsüberschreitung möglich?	Kennzeichen
>0	>0	Ja	Ergebnis MSB=1 (negativ)
>0	<0	Nein	-
<0	>0	Nein	-
<0	<0	Ja	Ergebnis MSB=0 (positiv)

## Subtraktion

Z1	Z2	Bereichsüberschreitung möglich?	Kennzeichen
>0	>0	Nein	-
>0	<0	Ja	Ergebnis MSB=1 (negativ)
<0	>0	Ja	Ergebnis MSB=0 (positiv)
<0	<0	Nein	-

Vergleicher können durch Subtraktion realisiert werden:

$$x < y \iff x - y \text{ erzeugt Unterlauf}$$

## Weitere Binärcodes

# Weitere Binärcodes für ganze Zahlen

- ▶ BCD-Code (Binary Coded Decimal)
  - Jede Dezimalziffer wird unabhängig als 4-Bit-Dualzahl kodiert.
  - Nicht verwendete, ungültige Dualzahlen (0xA-0xF) heißen *Pseudotetraden*.
  - **Vorteil:** Einfache Umrechnung Dezimalzahl  $\leftrightarrow$  BCD
  - **Nachteil:** 4 Bits unvollständig genutzt, komplizierte Rechnung mit 10er-Stellen
- ▶ Excess-n-Code (z. B. Excess-3-Code = Stibitz-Code)
  - Wert wird gegenüber Dual um n verschoben: Excess = Dual + n
- ▶ Gray-Code
  - *Stetiger Code:* benachbarte Codewörter unterscheiden sich nur in einem einzigen Bit; gilt auch für letzten und ersten Wert. Arithmetik unmöglich!

Wert	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Dual	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
BCD	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	<i>Pseudotetraden (nicht genutzt)</i>					
Stibitz-Code	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111	0000	0001	0010
Gray-Code	0000	0001	0011	0010	0110	0111	0101	0100	1100	1101	1111	1110	1010	1011	1001	1000

## Fehlererkennende Codes - Parity Bits

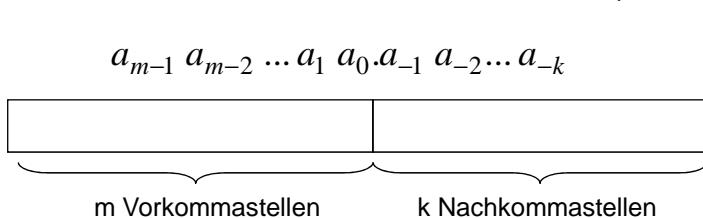
- ▶ **Problem in der Praxis:**
  - Einzelne Bits „kippen“ ungewollt, so dass aus 0 eine 1 wird oder umgekehrt.
- ▶ **Lösung:**
  - Codes, die eine Fehlererkennung ermöglichen
  - Codes, die eine Fehlerkorrektur ermöglichen
- ▶ **Einfachste Variante für die Fehlererkennung ist das Parity Bit**
- ▶ **Idee:**
  - Gezieltes Setzen eines Bits, so dass die Zahl der Einsen im gesamten (z.B.) Byte immer gerade ist („even parity“) oder immer ungerade ist („odd parity“).
  - Sollte einschließlich des Parity-Bits die Zahl der 1 nicht gerade (ungerade) sein, so liegt ein Fehler vor.
  - Nachteile:
    - Nur Fehler erster Ordnung, d. h. 1 Bit „gekippt“ werden erkannt
    - Das Parity Bit kann für die Zahlendarstellung nicht genutzt werden
  - Beispiel: Darstellung der Zahl 7 mit „even parity“
    - Dualcode: 111
    - Als Byte: 0000 0111
    - Mit Parity 1000 0111

# Festkommazahlen

## Positive Zahlen in Festkommadarstellung

### Festkommadarstellung (engl. *fixed-point*):

- ▶ Darstellung rationaler (gebrochener) Zahlen durch Verschieben der Gewichtungsfaktoren der Ziffern  
→ feste Anzahl von (dualen) Vor- und Nachkommastellen
- ▶ Wird nur von wenigen Programmiersprachen und Prozessoren direkt unterstützt (und kann durch Ganzzahlen ersetzt werden).



Erweiterung der  
Summenformel:

$$V(A) = \sum_{i=-k}^{m-1} a_i \cdot R^i$$

- ▶ Wertebereich:  $0 \dots R^m - R^{-k}$       Genauigkeit (Abstand benachbarter Zahlen)  $R^{-k}$

**Beispiel:**  $R=2$ ,  $m=4$ ,  $k=2$  (Gesamtlänge  $n = m+k = 6$  Bit)

0101.11 Umrechnung in Dezimalsystem:  $2^2 + 2^0 + 2^{-1} + 2^{-2} = 4 + 1 + \frac{1}{2} + \frac{1}{4} = 5,75$

1110.01

$2^3 + 2^2 + 2^1 + 2^{-2} = 8 + 4 + 2 + \frac{1}{4} = 14,25$

# Umwandlung von Festkommazahlen I

## ► Umwandlung vom Dezimalsystem (oder $R_x$ ) in Radix R

- Separate Rechnung für Vor- und Nachkommastellen
- **Vorkomma-Anteil  $A_v$ :** Verfahren wie bei ganzen Zahlen (siehe Folie 8/9)
- **Nachkomma-Anteil  $A_n$ :**

$$\text{Multiplikation mit } R \text{ ergibt: } A_n \cdot R = \left( \sum_{i=-k}^{-1} a_i \cdot R^i \right) \cdot R = \sum_{i=-k}^{-1} a_i \cdot R^{i+1} = \sum_{i=-k}^{-2} a_i \cdot R^{i+1} + a_{-1}$$

Nur  $a_{-1} \cdot R$  kann  $\geq 1$  sein

→ also entspricht der Vorkomma-Anteil von  $A_n \cdot R$  der Stelle  $a_{-1}$ !

**Sukzessive Berechnung der Ziffern durch Multiplikation:**

1.  $A \cdot R = M_{-1} + a_{-1}$  ( $M_i < 1 / a_i = 0$  oder  $a_i = 1$ )
2.  $M_i \cdot R = M_{i-1} + a_{i-1}$  für  $0 > i > -k$  (bis  $M_{i-1} = 0$ )

- **Setze das Ergebnis aus Vorkomma- und Nachkomma-Anteil zusammen:**

$$A = (a_{m-1} a_{m-2} \dots a_0, a_{-1} a_{-2} \dots a_{-k})_R$$

# Umwandlung von Festkommazahlen II

Beispiel:  $A = 6,625$  in das Dualsystem wandeln

*Vorkomma:*  $A_v = 6$

$$\begin{array}{rcl} 6 / 2 = 3 & \text{Rest } 0 & \rightarrow a_0 = 0 \\ 3 / 2 = 1 & \text{Rest } 1 & \rightarrow a_1 = 1 \\ 1 / 2 = 0 & \text{Rest } 1 & \rightarrow a_2 = 1 \end{array} \quad 0 \text{ zeigt Ende der Rechnung an!}$$

*Nachkomma:*  $A_n = 0,625$

$$\begin{array}{rcl} 0,625 \cdot 2 = 1,25 & \rightarrow a_{-1} = 1 \\ 0,25 \cdot 2 = 0,5 & \rightarrow a_{-2} = 0 \\ 0,5 \cdot 2 = 1,0 & \rightarrow a_{-3} = 1 \end{array} \quad 0 \text{ zeigt Ende der Rechnung an!}$$

Ergebnis zusammengesetzt:  $(A)_2 = 110,101$

Achtung: Für manche rationale Zahlen ergibt sich keine endliche Darstellung im Dualsystem, sondern ein periodischer Bruch:

z. B.  $A = 0,1$ :  $0,1 \cdot 2 = 0,2$

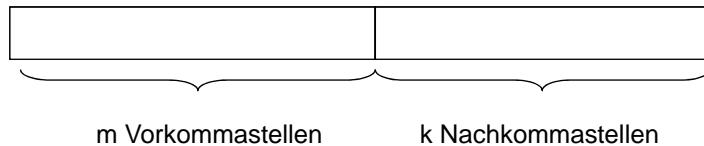
$$0,2 \cdot 2 = 0,4$$

$$\begin{array}{l} \rightarrow 0,4 \cdot 2 = 0,8 \\ 0,8 \cdot 2 = 1,6 \\ 0,6 \cdot 2 = 1,2 \\ 0,2 \cdot 2 = 0,4 \end{array} \quad (A)_2 = 0,00011001100\dots$$

# Festkommadarstellung (positive und negative Zahlen)

- Die Festkommadarstellung für Zweierkomplementzahlen (oder allgemein für Radixkomplement-Zahlen) wird analog der Festkommadarstellung für positive Zahlen gewählt:

$$A = -a_{m-1} \cdot 2^{m-1} + \sum_{i=-k}^{m-2} a_i \cdot 2^i$$



**Beispiel:** l=4 k=2

$$0101.11 \quad 2^2+2^0+2^{-1}+2^{-2} = 5,75$$

$$1110.01 \quad -2^3+2^2+2^1+2^{-2} = -1,75$$

Wertebereich:  $-R^{m-1} \dots R^{m-1} - R^{-k}$

## Gleitkommazahlen

## Gleitkommadarstellung

## Gleit- oder Fließkommadarstellung (engl. *floating-point*)

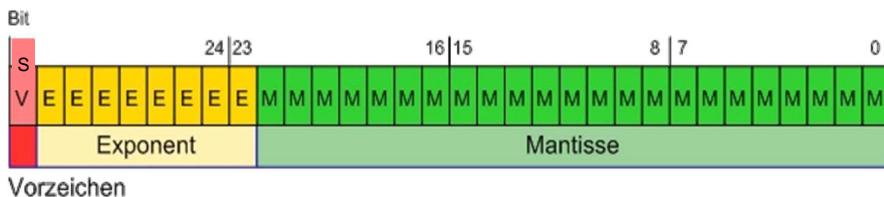
- Trennung zwischen Wertebereich und Genauigkeit  
→ erlaubt dynamische Anpassung der Darstellung
  - entspricht Exponentialdarstellung bei wissenschaftlichen Taschenrechnern mit Mantisse und Exponent, z. B.  $1,073 \cdot 10^{-24}$ .
  - Die Zahl wird in der Form  $A = (-1)^s \cdot m \cdot 2^e$  mit *Vorzeichen s*, *Mantisse m* (vorzeichenlos) und *Exponent e* (vorzeichenbehaftet) dargestellt.

### **Gebräuchliches Format (insgesamt 32 oder 64 Bit)**

Einfache Genauigkeit (single precision/float): M: 23 bit E: 8 bit

Doppelte Genauigkeit (double precision/double): M: 52 bit E: 11 bit

## Standardisierung durch IEEE 754



Quelle: wikipedia.de

# Normalisierung

- ▶ Mehrere Darstellungen sind für den gleichen Gleitkommawert möglich
  - ▶ **Beispiel:**  $12,3 = 1,23 \cdot 10^1 = 0,123 \cdot 10^2 = 0,0123 \cdot 10^3 = 0,00123 \cdot 10^4$
  - ▶ **Vereinbarung in IEEE 754:**
    - Der Exponent wird so gewählt, dass die Mantisse eine führende 1 besitzt, also:
$$1 \leq m < 2$$
  - ▶ Gegebenenfalls muss nach der Berechnung eines Ergebnisses eine **Normalisierung** durchgeführt werden, um wieder das vereinbarte Format zu erhalten
    - Mantisse: schieben
    - Exponent: addieren/subtrahieren

## Verstecktes Bit (Hidden Bit)

- ▶ Die Vereinbarung zur Normalisierung führt dazu, dass die Mantisse immer ein führendes Bit mit dem Wert 1 besitzt
- ▶ Also: Das oberste Bit der Mantisse muss nicht gespeichert werden
  - Wir kennen den Wert ja...
- ▶ Bei Berechnungen muss das Bit hinzugefügt werden
- ▶ Der Vorteil: Die Genauigkeit der Mantisse wird um ein Bit erhöht

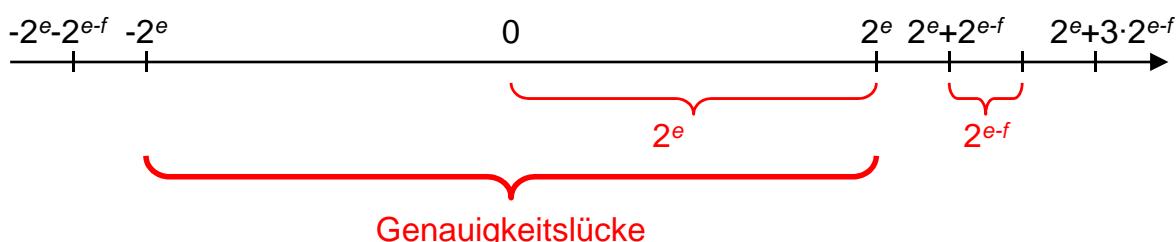
**Beispiel:** Darstellung in 8 Bit Mantisse, Wert 12,25

$$12,25 = (1100,01)_2 = + (1, \overbrace{10001000}^{\text{8 Bit Mantisse}}) \cdot 2^3$$

↑  
Verstecktes Bit  
(wird nicht abgespeichert)

## Genauigkeitslücke

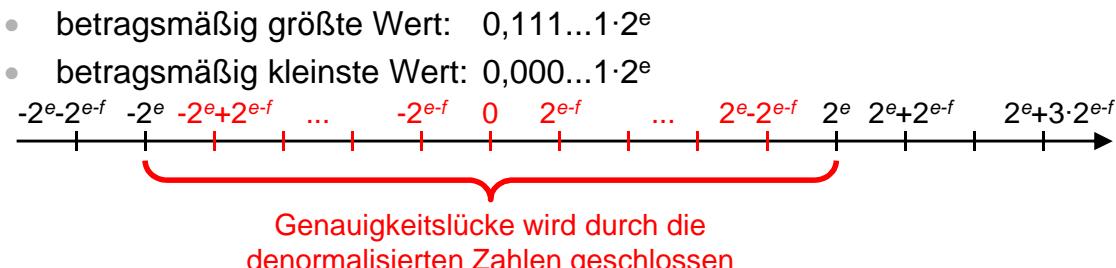
- ▶ Darstellung mit:  $f$  bits Mantisse,  $e$  kleinster Exponent
- ▶ Kleinster darstellbarer Betrag:  $1,000\dots0 \cdot 2^e = 2^e$
- ▶ Betragsmäßig nächstgrößere Zahl:  $1,000\dots1 \cdot 2^e = 2^e + 2^e \cdot 2^{-f}$
- ▶ Also: Differenz zwischen kleinster und nächstgrößter Zahl:  $2^e \cdot 2^{-f}$
- ▶ Dieser Wert ist kleiner als der Abstand der kleinsten Zahl zu Null !



## Füllen der Genauigkeitslücke

- Zum Füllen der Genauigkeitslücke kann für besonders kleine Zahlen auf die Normalisierung verzichtet werden
  - „Denormalisierte Zahlendarstellung“
  - „Hidden Bit“ wird dann zu Null gesetzt

- In dieser Darstellung ist der



- Kennzeichnung der denormalisierten Darstellung durch besonderen Wert im Exponenten
  - IEEE 754: Exponent = 0 kennzeichnet denormalisierte Darstellung (Kodierung des Exponenten: siehe Folie 45!)

## IEEE 754: Zahlentypen

Typ	Kennzeichen
Normalisierte Zahl	Exponent im Bereich $-126..127$ (einfache Genauigkeit) bzw. $-1022..1023$ (doppelte Genauigkeit). Vorzeichenbit spezifiziert das Vorzeichen der Zahl.
Denormalisierte Zahl	Im Exponent steht reservierter Code 0000..000. Vorzeichenbit spezifiziert das Vorzeichen der Zahl.
Unendlich	Exponent hat reservierten Code 1111..111. Alle Ziffern der Mantisse sind 0. Das Vorzeichen bestimmt, ob es sich um $+\infty$ oder $-\infty$ handelt.
Keine Zahl (NaN, Not a number)	Exponent hat reservierten Code $(1111..111)_2$ . Zumindest eine Ziffer der Mantisse ist ungleich 0. Vorzeichen ist beliebig.

# IEEE 754: Beispielcodierungen

## Codierung des Exponenten in IEEE754-Norm:

- Der Exponent  $e$  wird in Excess-Darstellung konvertiert → **Charakteristik  $c$**   
(Excess-127 für 32-Bit und Excess-1023 für 64-Bit, d. h. die Exponenten werden in den positiven Bereich verschoben)
- Umrechnung: 32-Bit:  $c = e + 127$    64-Bit:  $c = e + 1023$  bzw.  
 $e = c - 127$                             $e = c - 1023$

**Beispiel:** Bitmuster: 0 | 10001010 | 0000011101011110000000

Genauigkeit: einfach  
Zahlentyp: Normalisierte Zahl  
Vorzeichen: 0 (positiv)  
Exponent:  $(10001010)_2, \text{Excess127} = 138 - 127 = 11$   
Mantisse:  $(1,0000011101011110000000)_2$   
 $= 1,0300140380859375$   
 $a = +(1,0000011101011110000000)_2 \times 2^{11} = 2109,46875$

# IEEE 754: Beispielcodierungen II

**Beispiel:** Bitmuster: 1 | 00000001 | 11001110000011110000000

Genauigkeit: einfach  
Zahlentyp: Normalisierte Zahl  
Vorzeichen: 1 (negativ)  
Exponent:  $(00000001)_2, \text{Excess127} = 1 - 127 = -126$   
Mantisse:  $(1,11001110000011110000000)_2$   
 $a = -(1,11001110000011110000000)_2 \times 2^{-126} \approx -2,121669010555 \times 10^{-38}$

**Beispiel:** Bitmuster: 1 | 00000000 | 11001110000011110000000

Genauigkeit: einfach  
Zahlentyp: Denormalisierte Zahl (Exponent alle Bits 0)  
Vorzeichen: 1 (negativ)  
Exponent:  $(00000000)_2, \text{Spezialcode} = -126$   
Mantisse:  $(0,11001110000011110000000)_2$   
 $a = -(0,11001110000011110000000)_2 \times 2^{-126} \approx -9,461746597325 \times 10^{-39}$

# Gleitkomma-Arithmetik: Beispiel Addition

---

- Für die Addition zweier Zahlen in Gleitkomma-Darstellung müssen beide Zahlenwerte den gleichen Exponenten aufweisen

- Dies kann durch Schieben der Mantisse erreicht werden

► Beispiel

- Zum leichteren Verständnis: Gleitkomma-Darstellung zur **Basis 10** (!!)

$$A = 123456,7 = 1,234567 \cdot 10^5$$

$$B = 101,7654 = 1,017654 \cdot 10^2$$

- Schieben der Mantisse von B

$$B = 0,001017654 \cdot 10^5$$

- Addition von A und B ist nun direkt möglich

$$A + B = (1,234567 + 0,001017654) \cdot 10^5$$

$$= 1,235584654 \cdot 10^5$$

- Rundung der Mantisse liefert das Ergebnis  $A + B = 1,235585 \cdot 10^5$

## Addierer-Varianten

## Addierer-Varianten

- Der Volladdierer als Grundelement für Schaltungen, die addieren oder subtrahieren sollen, ist gegeben.
- Da normalerweise aber mehr als nur drei Bits addiert werden sollen, muss man sich entscheiden:

Wie baue ich meine Schaltung tatsächlich auf? Welche Kriterien soll die Schaltung erfüllen?

### Mögliche Kriterien:

- Schnell
- Wenig Aufwand
- Einfach zu verstehen
- Flexibel anpassbar und erweiterbar
- Anzahl der Operanden
- usw.

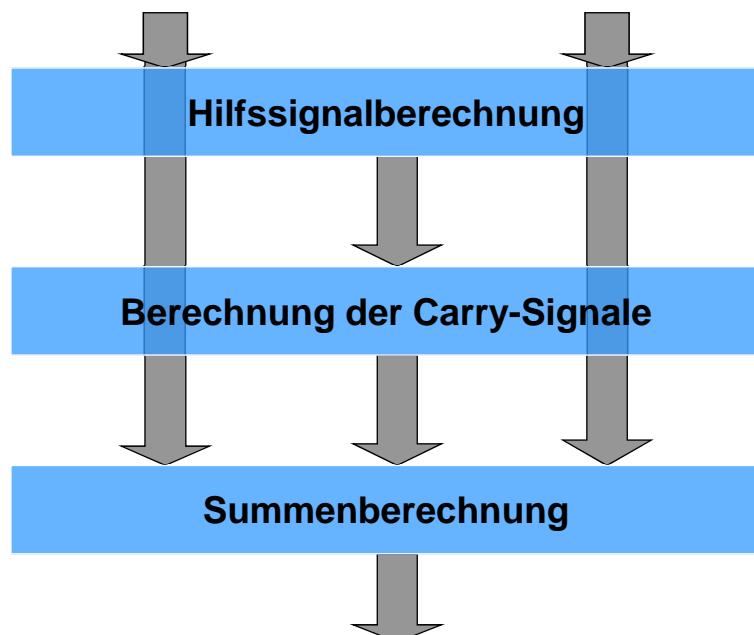
**Es stehen für die Addition mehrere Arten zur Verfügung!**

## Carry-Lookahead-Addierer (CLA)

Carry-Signal-Berechnung bestimmt Laufzeit des Ripple-Carry-Addierers

- Verkürzung der Additionszeit durch Beschleunigung der Carry-Signale

### Prinzip:



# CLA: Hilfssignalberechnung

---

## Unterscheidung von drei Fällen (in Abhängigkeit von $a_i$ und $b_i$ )

- I) Ausgangscarry ist Null ( $a_i=0, b_i=0$ )
- II) Ausgangscarry ist gleich dem Eingangscarry ( $a_i=1, b_i=0$  oder  $a_i=0, b_i=1$ )
- III) Ausgangscarry ist Eins ( $a_i=1, b_i=1$ )

Entsprechend werden für jedes Bit drei *Hilfssignale* bestimmt

- I) „carry-kill“  $k_i = \bar{a}_i \bar{b}_i$
- II) „carry-propagate“  $p_i = a_i \oplus b_i$
- III) „carry-generate“  $g_i = a_i b_i$

Vereinfachung für das „carry-propagate“ Signal

$$p'_i = a_i \vee b_i \quad (\text{für } a_i=1, b_i=1 \text{ gilt sowieso carry-generate})$$

# CLA: Bestimmung der Carry-Signale aus den Hilfssignalen

---

$$\begin{aligned}c_{i+1} &= a_i b_i \vee c_i (a_i \oplus b_i) = g_i \vee c_i p_i \\&= a_i b_i \vee c_i (a_i \vee b_i) = g_i \vee c_i p'_i\end{aligned}$$

## Vorteil

- Die Hilfssignale einzelner Bitstufen können unabhängig voneinander berechnet werden.
- Die Berechnung der Carry-Signale auf Basis der Hilfssignale erfolgt vorab und damit schneller.

# Berechnung der Carry-Signale aus den Hilfssignalen

$$c_1 = g_0 \vee p_0 c_0$$

$$c_2 = g_1 \vee p_1 c_1 = g_1 \vee p_1(g_0 \vee p_0 c_0) = g_1 \vee p_1 g_0 \vee p_1 p_0 c_0$$

$$c_3 = g_2 \vee p_2 c_2 = g_2 \vee p_2 g_1 \vee p_2 p_1 g_0 \vee p_2 p_1 p_0 c_0$$

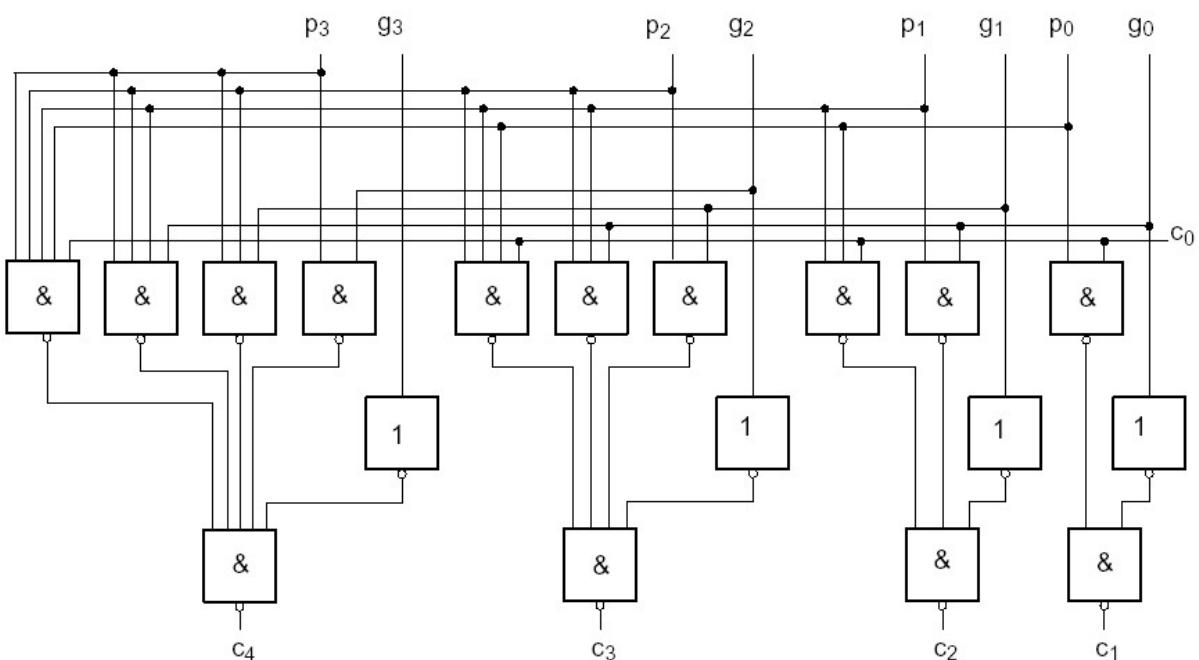
$$c_4 = g_3 \vee p_3 c_3 = g_3 \vee p_3 g_2 \vee p_3 p_2 g_1 \vee p_3 p_2 p_1 g_0 \vee p_3 p_2 p_1 p_0 c_0$$

$$\begin{aligned} c_{i+1} &= g_i \vee p_i c_i \\ &= g_i \vee p_i g_{i-1} \vee p_i p_{i-1} g_{i-2} \vee \dots \vee p_i p_{i-1} \dots p_1 p_0 c_0 \\ &= g_i \vee \sum_{j=0}^{i-1} \left( \prod_{k=j+1}^i p_k \right) g_j \vee \prod_{k=0}^i (p_k) c_0 \end{aligned}$$

## CLA: Gatterrealisierung zur Berechnung der Carry-Signale

### Beispiel: Berechnung von $c_1$ bis $c_4$

(Berechnung der Hilfssignale und der Summenbits erfolgt in 1 bzw. 2 Stufen!)



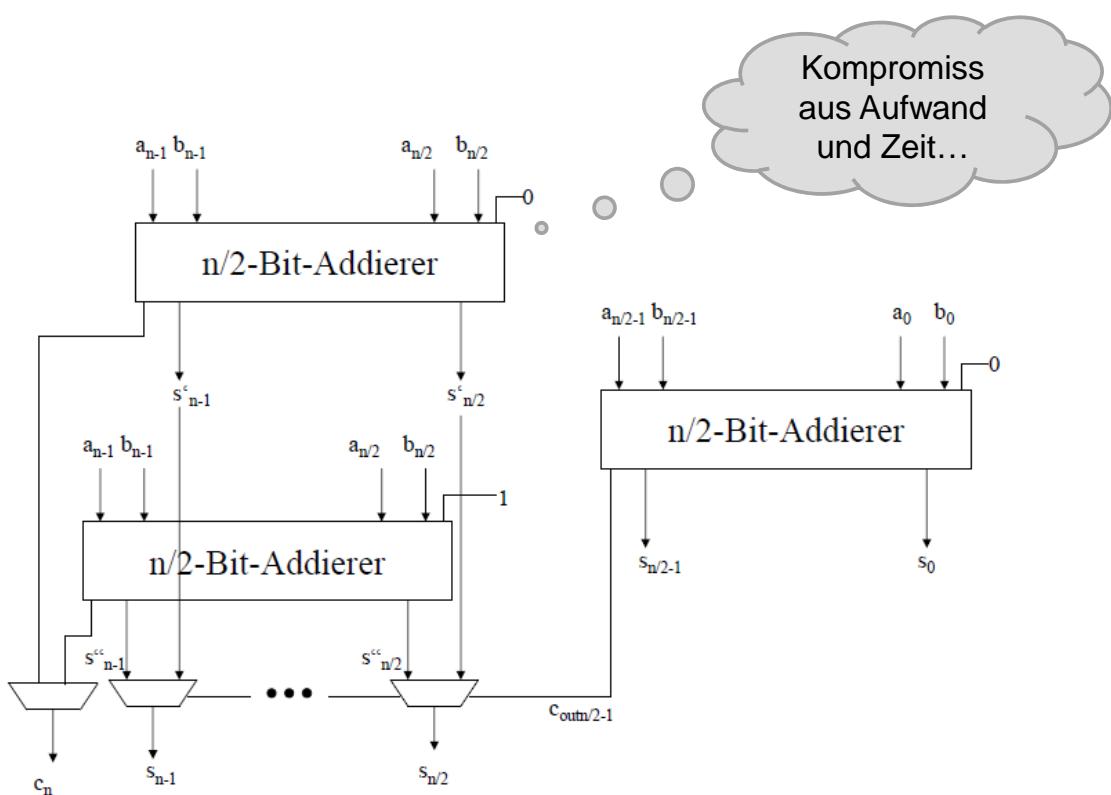
Verzögerungszeit (vereinfacht):  $T_D = O(1)$       Aufwand:  $A = O(n^3)$       sehr groß!

# Carry-Select-Addierer

## Strategie

- Bildung von Bitgruppen (Größe typischerweise 2 oder 4 bit)
- Berechnung der Summen- und Carry-Signale jeweils für  $c_{in}=0$  und  $c_{in}=1$
- Auswahl der richtigen Alternative (Ergebnis) anhand der ermittelten Carry-Signale

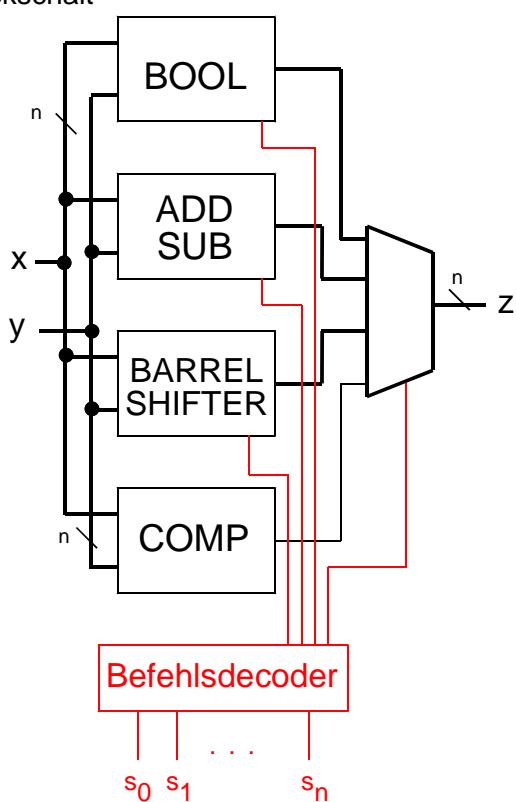
# Carry-Select-Addierer



# Arithmetisch-Logische Einheit (ALU)

# Arithmetisch-Logische Einheit (ALU)

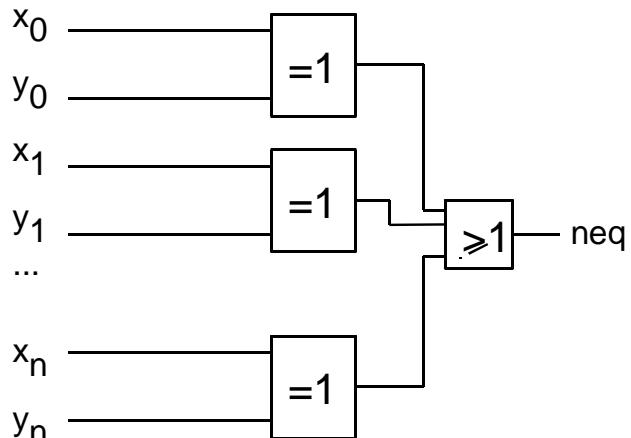
Blockschaltbild:



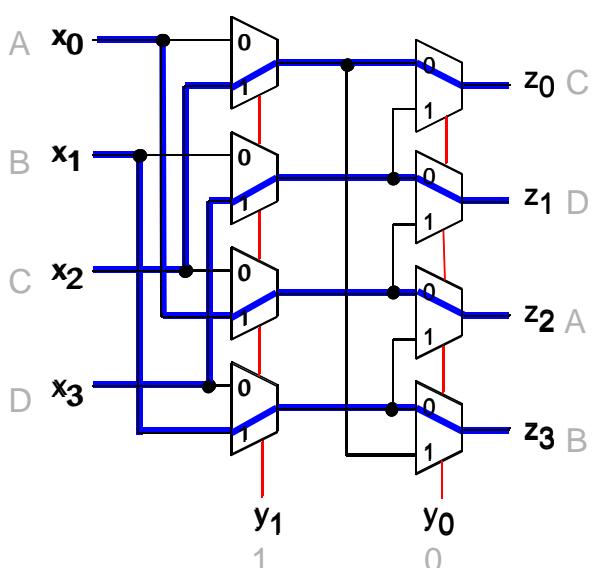
- ▶ Universalbaustein zur befehlsgesteuerten Ausführung verschiedener Operationen
- ▶ Zentraler Bestandteil von Prozessoren
- ▶ Auswahl der Operation durch Steuerleitungen
- ▶ Auswahl des Ergebnisses durch Multiplexer
- ▶ Evtl. Befehlsdecoder zur Generierung der Steuer- und Auswahlleitungen
- ▶ Je nach Anwendungsgebiet kann die ALU auch weniger oder mehr als die gezeigten Funktionen enthalten, z. B. Multiplikation
- ▶ Die komplexeren Fließkomma-Operationen werden meist in separaten Einheiten (in mehreren Takten) berechnet.

## ALU: Vergleicher (Comparator)

- ▶ Kleiner/Größer-Vergleich durch spezielle Schaltung realisiert
  - oder durch vorhandenen Subtrahierer:
    - $x < y \Leftrightarrow x - y$  erzeugt Unterlauf
- ▶ Ungleichheit wird durch die nebenstehende Schaltung realisiert
- ▶ Durch Kombination bzw. Invertieren der Ergebnisse erhält man:  $<, \leq, >, \geq, =$



## ALU: Barrel-Shifter/Barrel-Rotator



- ▶ Barrel-Shifter und Barrel-Rotator verschieben den Bitvektor am Eingang  $x$  um die durch  $y$  bestimmte Anzahl Bits.
- ▶ Beim Shifter werden Nullen nachgefüllt.
- ▶ Beim Rotator werden die herausgeschobenen Bits wieder in das Ergebnis hineingeschoben (zyklischer Shift).

Beispiel Rotator  
(x: 4 Bit, y: 2 Bit)

## ALU: Logik-Einheit

- ▶ In der Logik-Einheit werden die Eingabesignale bitweise mit der durch Steuersignale gewählten booleschen Funktion verknüpft.

- ▶ **Implementierung:**

Sollen z. B. die Und-, Oder- und Exklusiv-Oder-Verknüpfung sowie die Invertierung realisiert werden, wird für jedes Bit eine boolesche Funktion mit den vier Eingängen ( $x_0, x_1, s_0, s_1$ ) nach nebenstehender Funktionstabelle realisiert.

<b><math>x_1</math></b>	<b><math>x_0</math></b>	<b><math>s_1</math></b>	<b><math>s_0</math></b>	<b><math>z_i</math></b>
-	-	0	0	$x_i \wedge y_i$
-	-	0	1	$x_i \vee y_i$
-	-	1	0	$x_i \oplus y_i$
-	-	0	0	$\neg x_i$

### Hinweis:

Optimierte ALUs in Standard-Schaltungen sind nicht so modular aufgebaut wie hier gezeigt. Vielmehr werden die Funktionen für jede Bit-Stelle kombiniert und optimiert.

### Beispiel: Kombination von Logik-Einheit, Invertierer und Komparator

## Multiplizierer

# Binäre Multiplikation I (Positive Zahlen)

**Multiplikand**       $A = a_{m-1}a_{m-2}\dots a_1a_0$

**Multiplikator**       $B = b_{n-1}b_{n-2}\dots b_1b_0$

**Produkt**             $P = p_{m+n-1}\dots p_1p_0$

$$\begin{aligned} V(P) &= \sum_{k=0}^{m+n-1} p_k 2^k \\ &= V(A) \cdot V(B) = \left[ \sum_{i=0}^{m-1} a_i 2^i \right] \cdot \left[ \sum_{j=0}^{n-1} b_j 2^j \right] \\ &= \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} a_i 2^i b_j 2^j = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} (a_i b_j) 2^{i+j} \end{aligned}$$

*Teilprodukt auf Bitebene  
(UND-Verknüpfung)*

# Binäre Multiplikation II (Positive Zahlen)

**Array-Multiplizierer als „direkte“ Hardware-Implementierung der Formel**

$$V(P) = V(A) \cdot V(B) = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} (a_i b_j) 2^{i+j}$$

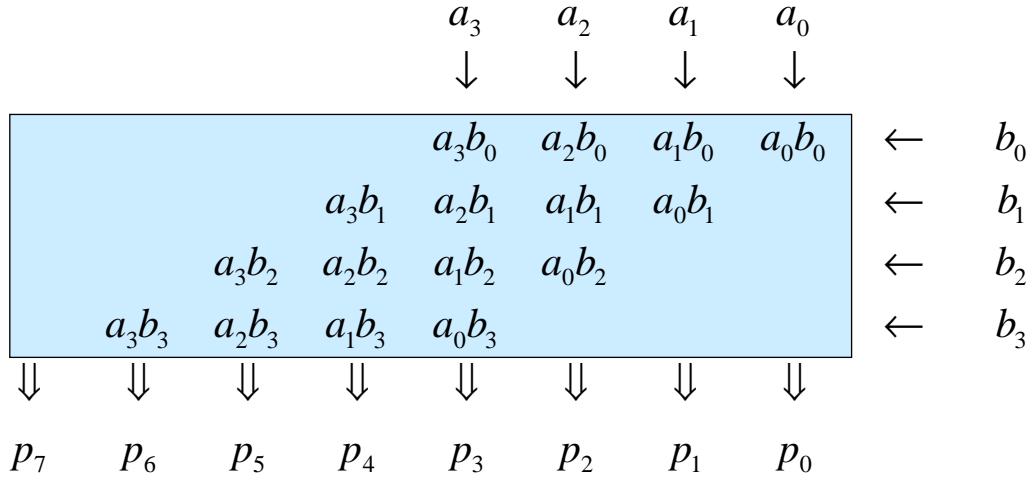
## Benötigte Grundelemente

$a_i b_j$     UND-Verknüpfung (Bildung der Teilprodukte auf Bitebene)

$\sum_{i=0}^{m-1} \sum_{j=0}^{n-1}$     Addierer

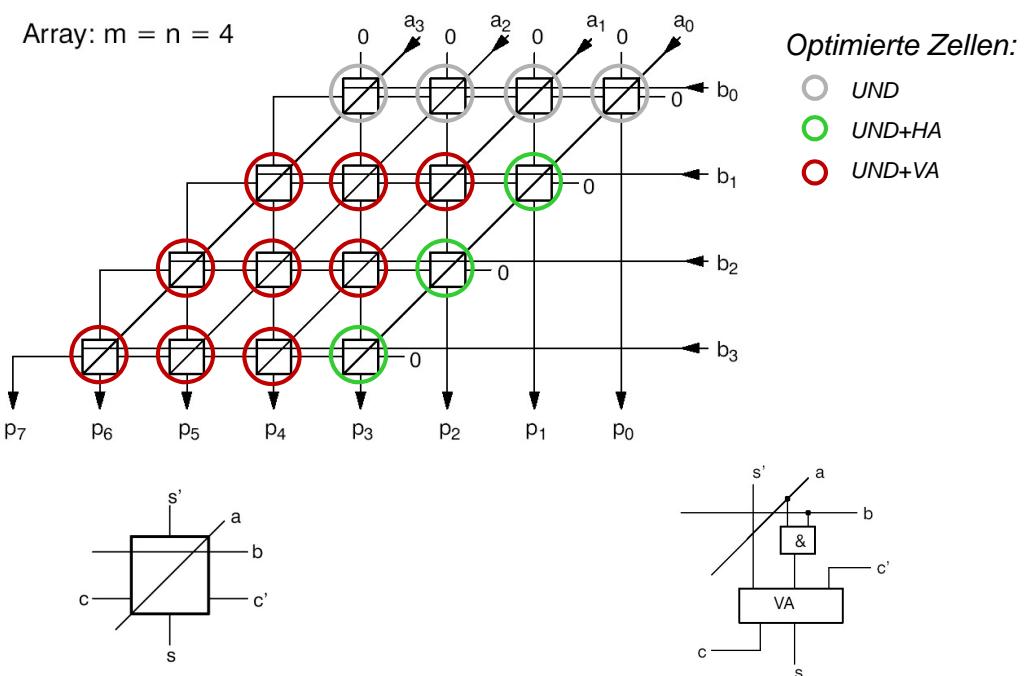
$2^{i+j}$     Unterschiedliche Gewichtung der Teilprodukte ( $a_i b_j$ ) durch „Verdrahtung“ realisiert

# Binäre Multiplikation III (Positive Zahlen)



Teilproduktbildung & Summation

## Ripple-Carry-Multiplizierer



**Multiplizierer für Zweierkomplement-Zahlen durch Absolutwert-Bildung, Standard-Multiplizierer und ggf. Negierung, oder direkte Implementierung der Zweierkomplement-Addition durch angepasste Zellen im Array-Multiplizierer.**

---

# Zeichencodierung

## Zeichencodierung: Einführung

- ▶ Zeichencodes ordnen einer Menge von Schriftzeichen (Zeichensatz) eine Menge von Dualzahlen zu. Je nach der Anzahl der Dualziffern spricht man von 7-Bit, 8-Bit oder 16-Bit Codes.
- ▶ Verwendet zum internen Speichern ein- oder auszugebender Zeichen
- ▶ **ASCII** (American Standard Code for Information Interchange)
  - Ältester Standard
  - 7-Bit Code → 128 Symbole
  - Steuersymbole, Buchstaben, Ziffern, Sonderzeichen.
  - Enthält keine weiteren sprachspezifischen (lateinischen) Zeichen wie z. B. Umlaute.
- ▶ **ISO 8859** (internationale Norm)
  - 8-Bit Code → 256 Symbole; untere Hälfte entspricht ASCII-Norm
  - 15 Teillisten für Sprachgruppen
  - ISO 8859-1 (Latin-1) für westeuropäische Sprachen
- ▶ **Unicode**
  - universeller Code, umfasst 16 Bereiche mit je 65536 Zeichen (16 Bit)
  - Codierung mit 8, 16 oder 32 Bit möglich (UTF-8, UTF-16, UTF-32)

# Zeichencodierung: ASCII

		ASCII-Zeichtabellen, hexadezimale Nummerierung															
dez.:	Code	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...A	...B	...C	...D	...E	...F
0...	0...	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
16...	1...	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
32...	2...	SP	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
48...	3...	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64...	4...	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80...	5...	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	-
96...	6...	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112...	7...	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Darstellbare Zeichen ab 0x20!

Quelle: [www.wikipedia.de](http://www.wikipedia.de)



# Zeichencodierung: ISO 8859

## ► ISO 8859-1 (Latin-1)

		Code	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...A	...B	...C	...D	...E	...F
		0...	Steuerzeichen															
		1...	SP	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	
		2...	0	1	2	3	4	5	6	7	8	9	:	;	<	=	> ?	
		3...	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	
		4...	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^ -	
		5...	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	
		6...	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
		7...	Steuerzeichen															
		8...	Steuerzeichen															
		9...	NBSP	í	¢	£	¤	¥	₩	₪	₪	₪	₪	₪	₪	₪	₪	
		A...	°	±	²	³	‘	μ	¶	·	,	†	º	»	¼	½	¾	
		B...	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Ï	
		C...	Ð	Ñ	Ò	Ó	Ô	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	þ	
		D...	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	ï	
		E...	ð	ñ	ò	ó	ô	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ	
		F...																

Quelle: [www.wikipedia.de](http://www.wikipedia.de)



# 10. Synchrone Schaltungen

Einleitung (Definition und Beispiel)

Endliche Automaten

Zähler

Schieberegister

Systeme zur synchronen Verarbeitung digitaler Daten

## Einleitung - Wiederholung

### ► Schaltnetz/Kombinatorik

- Ausgänge hängen nur von den Eingängen ab
- Verwendung von UND-, ODER- und NICHT-Gattern
- Beispiele: Multiplexer, Addierer

### ► Schaltwerk/Sequenzielle Schaltung

- Rückkopplung von Signalen an den Eingang
- Ausgänge hängen von Eingängen und vorherigem Zustand ab  
➔ Schaltwerk hat "Gedächtnis"

### ► Motivation für *synchrone Schaltwerke*

- Asynchron rückgekoppelte Schaltungen sind aufwendig zu implementieren und fehleranfällig!
    - Gefahr des Schwingens:  
hazardfrei implementieren, bei Übergang nur ein Zustandssignal ändern
- ➔ Asynchrone Rückkopplungen sollten vermieden werden!
- ➔ Beschränkung auf *synchrone Schaltungen*

# Einleitung

## Definition: Synchrone Schaltung

- Jeder Rückkopplungspfad enthält taktflankengesteuerte Flip-Flops als Speicherelemente.  
→ Zustand ausschließlich durch Flip-Flop-Werte definiert!
- Alle Flip-Flops werden durch *dasselbe* (globale) Taktsignal flankengesteuert.  
→ Dieser Takt synchronisiert die Zustandsübergänge.  
→ In jeder Taktperiode befindet sich die Schaltung in einem eindeutigen Zustand. Der Zustand bleibt während der ganzen Taktperiode unverändert.

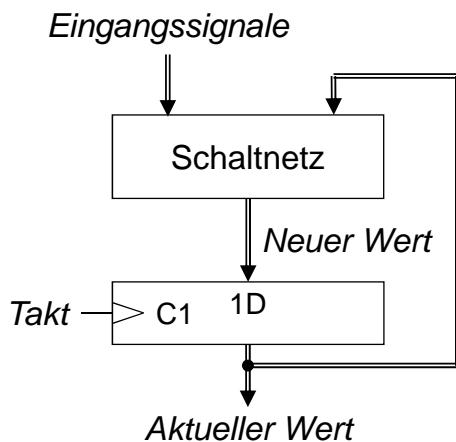
# Einleitung

## Eigenschaften synchroner Schaltungen

- Änderungen eines Eingangs wirken sich nicht sofort auf den Zustand aus, sondern erst mit der nächsten Taktflanke.
- Beschränkung auf flankengesteuerte Flip-Flops (eigentlich selbst asynchron rückgekoppelte Schaltungen!) verhindert instabile Schaltungen, falls folgendes gilt:
  - Die Eingänge der Flip-Flops sind von kurz vor der Taktflanke (Setup-Time) bis kurz nach der Flanke (Hold-Time) stabil.
- Mehrere Zustandssignale dürfen sich bei einem Zustandsübergang ändern, und es dürfen auch Hazards auftreten. Die Flip-Flop-Eingänge müssen sich nur vor der Taktflanke stabilisiert haben, d. h. ihren endgültigen Wert erreicht haben.
- Die Takt-Frequenz einer synchronen Schaltung wird durch die maximale Signalverzögerung der Schaltnetze begrenzt
  - Die Verzögerung zwischen einem Flip-Flop-Ausgang (oder einem Schaltungs-Eingang) und einem Flip-Flop-Eingang darf nicht länger als die Taktperiode (Abstand zweier positiver Taktflanken) minus der Setup-Time der Flip-Flops sein.

# Einleitung

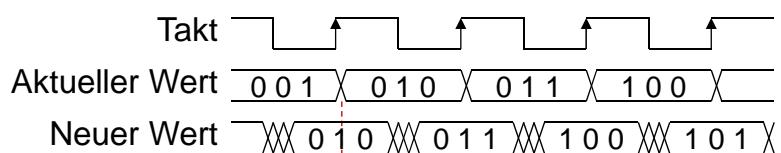
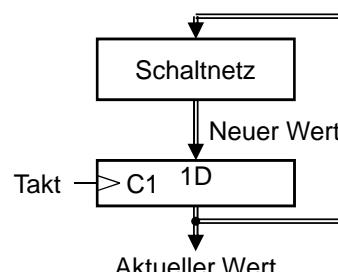
## Synchrone Schaltung zur Erzeugen einfacher Sequenzen



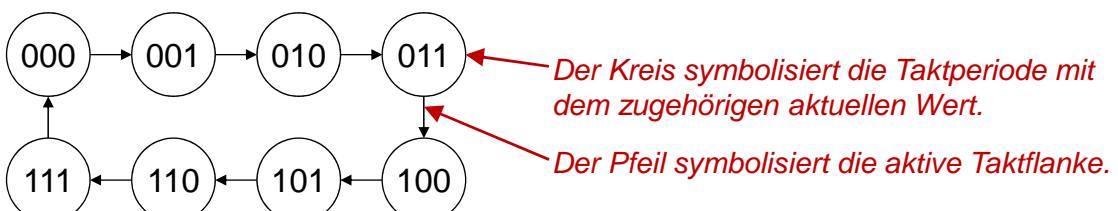
- Mit jeder aktiven Taktflanke wird „Neuer Wert“ als „Aktueller Wert“ übernommen.
- In der Takelperiode wird mittels des Schaltnetzes aus „Aktueller Wert“ und „Eingangssignalen“ ein „Neuer Wert“ berechnet.
- Die Verzögerung des Schaltnetzes muss so klein sein, dass „Neuer Wert“ rechtzeitig vor der nächsten aktiven Taktflanke stabil bereit steht.

## Beispiel: 3-bit Binärzähler (ohne Eingänge)

Aktueller Wert	Neuer Wert
0 0 0	0 0 1
0 0 1	0 1 0
0 1 0	0 1 1
0 1 1	1 0 0
1 0 0	1 0 1
1 0 1	1 1 0
1 1 0	1 1 1
1 1 1	0 0 0



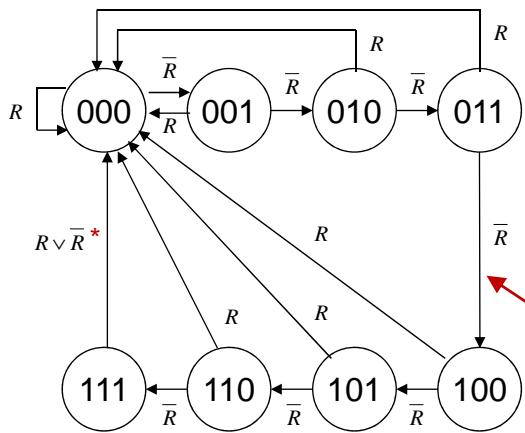
Einfaches Zustandsdiagramm:  
(Übergang zum nächsten Zustand  
nur bei aktiver Taktflanke!)



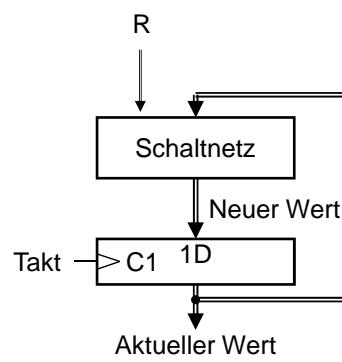
# Beispiel: 3-bit Binärzähler (mit Reset-Eingang R)

## Zustandsdiagramm:

(Übergang zum nächsten Zustand nur bei aktiver Taktflanke!)



\* Bedingung, die immer wahr ist, kann weggelassen werden!  
(Unbedingter Zustandsübergang)



Der Pfeil (Kante) bestimmt den Zustandsübergang und ist mit der Übergangsbedingung markiert.

Alternative Schreibweise:  
0 für  $\neg R$ , 1 für  $R$

Bei mehreren Eingangssignalen können Bedingungen kombiniert werden, z. B.  $\neg R \vee S$

## Entwurf synchroner Schaltungen

- ▶ **Synchronre Schaltungen werden für Ablaufsteuerungen verschiedenster Art verwendet.**
- ▶ **Diese sollten in zwei Schritten entworfen werden:**
  - Modellierung/Spezifikation als *Endlicher Automat* (engl. *Finite State Machine – FSM*)  
Festlegung der Zustände, Übergänge, Ein-/Ausgänge
  - Implementierung als konkrete *synchronre Schaltung*  
Kodierung der Zustände durch Flipflops, Realisierung der Schaltnetze

# Endliche Automaten

## Endliche Automaten: Definition

► Ein endlicher Automat besteht aus folgenden Komponenten:

- Eingabemenge  $E$
- Ausgabemenge  $A$
- Zustandsmenge  $Z$
- Startzustand  $z^0$
- Zustandsübergangsfunktion  $f_Z: Z \times E \rightarrow Z$
- Ausgabefunktion  $f_A: Z \times E \rightarrow A$

► Eingabe- und Ausgabevektoren

- *Eingabevektor*  $e \in E$ : mögliche Belegung der Eingabesignale  $e_0, e_1, e_2, \dots$ , z. B.  $e = (e_0, e_1, e_2, e_3) = (0, 1, 1, 0)$
- Menge  $E$  enthält also alle möglichen Belegungen der Eingangsvariablen.
- *Ausgabevektor*  $a \in A$ : mögliche Belegung der Ausgabesignale  $a_0, a_1, a_2, \dots$ ,
- Zur Vereinfachung der Darstellung sollen alle Eingabe- und Ausgabesignale nur binäre Werte annehmen.
- Im Gegensatz dazu enthält die Menge  $Z$  alle möglichen Zustände, in denen sich der Automat befinden kann. Muss (noch) nicht binär kodiert sein!

# Endliche Automaten: Definition (Forts.)

## ► Zustandsübergangsfunktion $f_Z$

- $f_Z(z^t, e^t) = z^{t+1}$  definiert, in welchen neuen Zustand  $z^{t+1}$  der Automat aus dem Zustand  $z^t$  wechselt, wenn der Vektor  $e^t$  an den Eingangssignalen anliegt.
- $f_Z$  kann auch als Zustandsübergangstabelle oder -diagramm dargestellt werden (siehe einführendes Beispiel). Realisierung durch Schaltnetz!

## ► Ausgabefunktion $f_A$

- $f_A(z^t, e^t) = a^t$  definiert für dieselbe Situation, welcher Ausgangvektor  $a^t$  erzeugt wird.  $f_A$  kann auch als Ausgabetabelle dargestellt werden. (Schaltnetz!)
- Man unterscheidet folgende Fälle:
  - Falls  $f_A$  von  $z^t$  **und**  $e^t$  abhängt, handelt es sich um einen **Mealy-Automaten**.
  - Falls  $f_A$  nur von  $z^t$  abhängt, also  $a^t = f_A(z^t)$ , spricht man von einem **Moore-Automaten**.
  - Falls  $a^t = z^t$ , der Zustand also direkt ausgegeben wird, spricht man auch von **Medwedew-Automaten** (Spezialfall des Moore-Automaten).

Oft werden für den aktuellen (alten) Zustand  $z^t$  und die Eingabe- und Ausgabevektoren  $e^t$  und  $a^t$  auch einfach  $z$ ,  $e$  und  $a$  verwendet.

Der neue Zustand  $z^{t+1}$  wird dann zur Unterscheidung mit  $z^*$  bezeichnet.

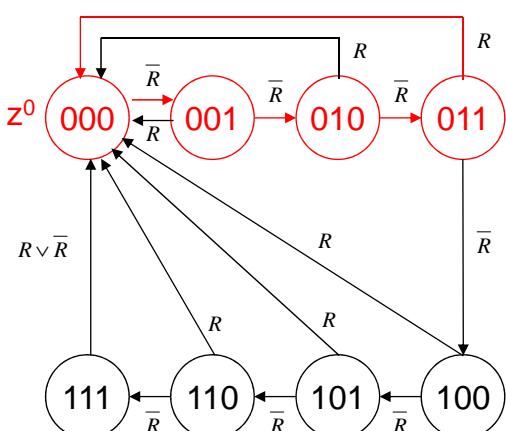
**Achtung:** Bei asynchronen Automaten wurde der *alte* Zustand mit  $z^*$  bezeichnet!

# Endliche Automaten: Definition (Forts.)

## ► Funktionsweise des endlichen Automaten:

Mit den Funktionen  $f_Z$  und  $f_A$  bzw. mit dem Zustandsübergangsdiagramm werden aus dem Startzustand  $z^0$  und einer Folge von Eingabevektoren (zu den abstrakten Zeitpunkten  $t=0, 1, 2, 3, \dots$ ) eine Folge von neuen Zuständen und Ausgabevektoren „berechnet“.

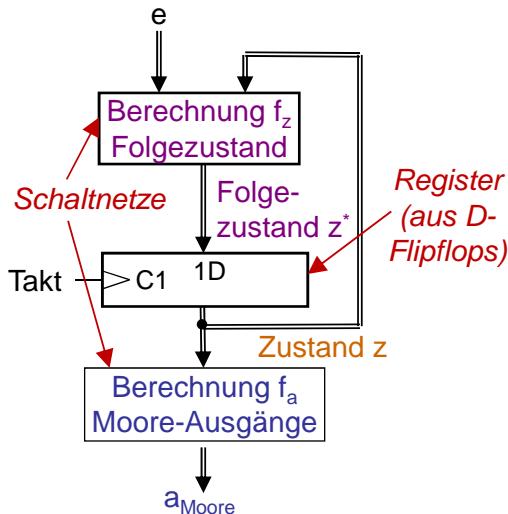
Beispiel:



$$\begin{aligned}z^0=000, R^0=0 &\rightarrow z^1=001 \\R^1=0 &\rightarrow z^2=010 \\R^2=0 &\rightarrow z^3=011 \\R^3=1 &\rightarrow z^4=000 \\&\dots\end{aligned}$$

( $a^t = z^t \rightarrow$  Medwedew-Automat!)

# Implementierung: Synchrone Schaltung für Moore-Automat



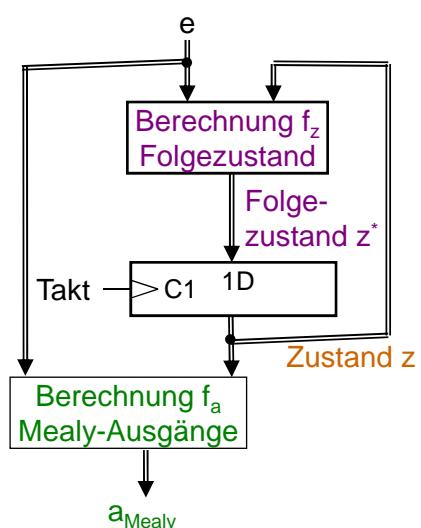
- Steuerung der Zustands-Sequenz in Abhängigkeit von Eingängen  $e$  und Zustand  $z$  durch Schaltnetz für  $f_z$
- Bei Moore-Automaten gilt  $a = f_A(z)$ , d. h. Ausgangssignale hängen nur von Zustand ab (nicht von Übergang nach  $z$ )
  - ➔ Werte während der Taktperioden stabil
- Schaltnetz für  $f_A$  berechnet Ausgänge aus Zustand  $z$
- Diese Ausgangssignale werden als *Moore-Ausgänge* bezeichnet

Ein Übergang wird durch eine Übergangsbedingung  $f_z(e)$  ausgewählt.

Übergangsbedingung  $f_z(e)$

Der Kreis symbolisiert die Taktperiode, in der ein fester Zustand eingenommen und feste Werte für die Moore-Ausgänge ausgegeben werden

# Implementierung: Synchrone Schaltung für Mealy-Automat

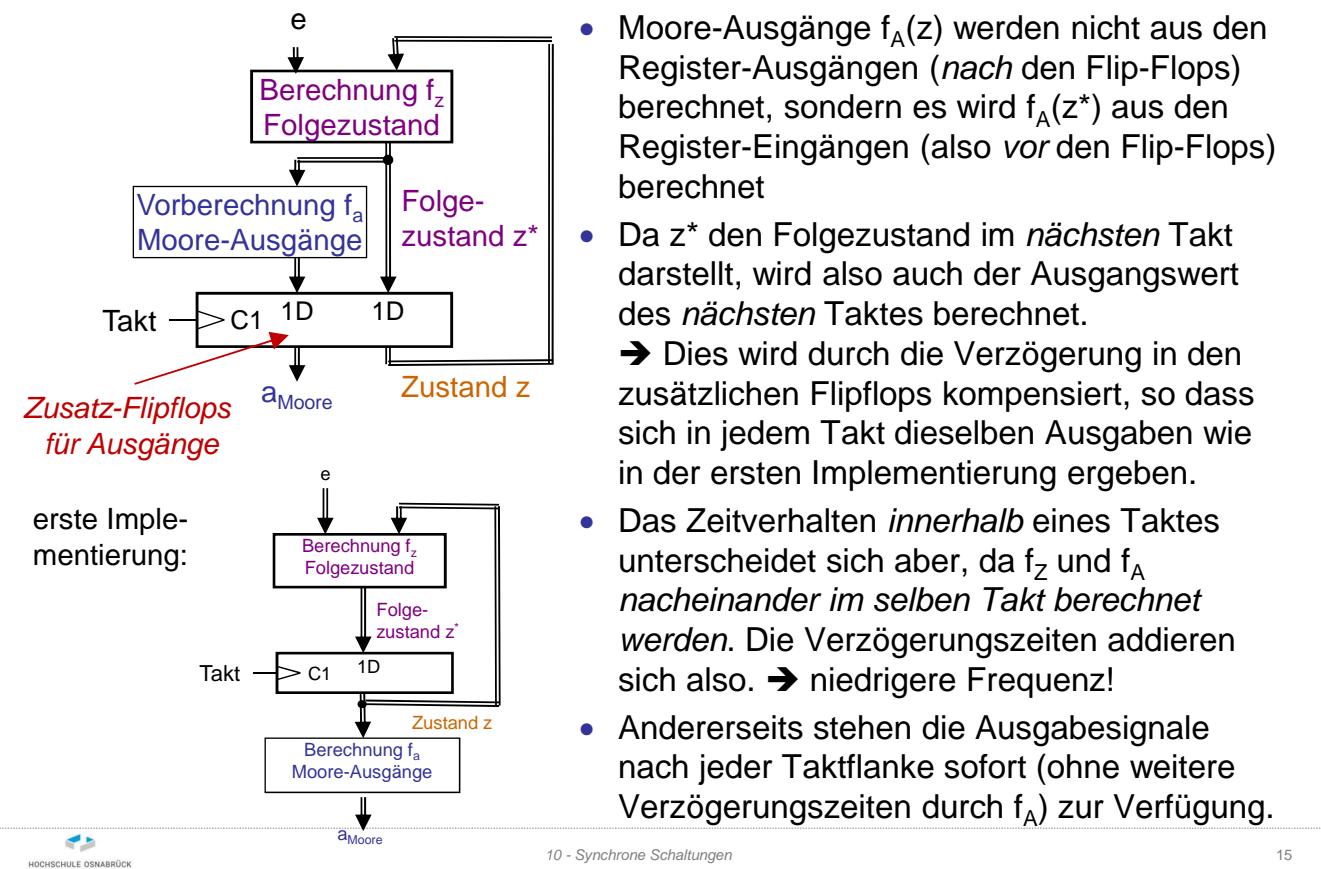


- Schaltnetz für  $f_z$  wie bei Moore-Automat
- Bei Mealy-Automaten gilt  $a = f_A(z, e)$ , d. h. einige Ausgangssignale hängen von Zustand und Eingang ab
  - ➔ Werte reagieren schnell auf Änderungen der Eingangssignale  $e$
- Schaltnetz für  $f_A$  berechnet Ausgänge aus Zustand  $z$  und Eingängen  $e$
- Ausgangssignale, die von  $e$  abhängen, werden als *Mealy-Ausgänge* bezeichnet
- Mealy-Automaten können auch reine Moore-Ausgänge enthalten

Übergangsbedingung  $f_z(e)$  /  
Werte  $a_{Mealy}$  der Mealy-Ausgänge

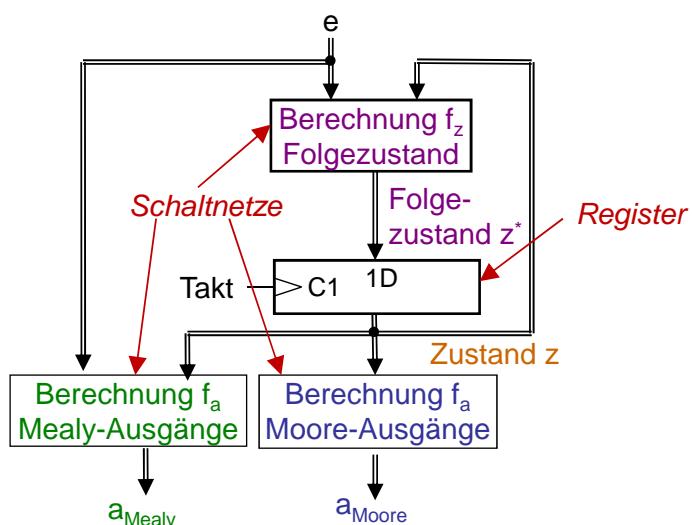
Ein Übergang wird durch eine Übergangsbedingung  $f_z(e)$  ausgewählt. Ihm werden feste Werte für die Mealy-Ausgänge zugeordnet (nach Schrägstrich).

# Alternative Implementierung eines Moore-Automaten

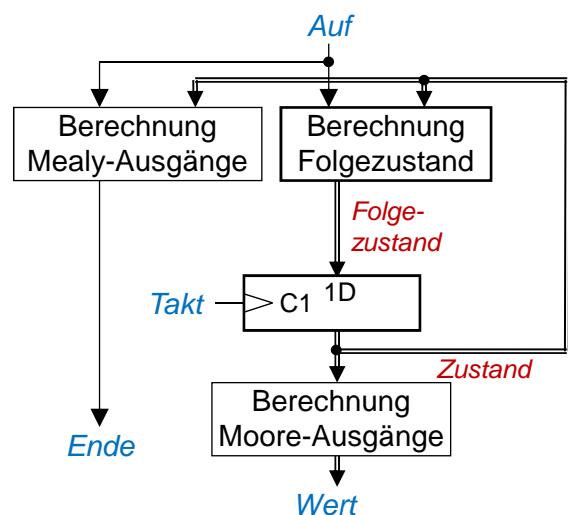
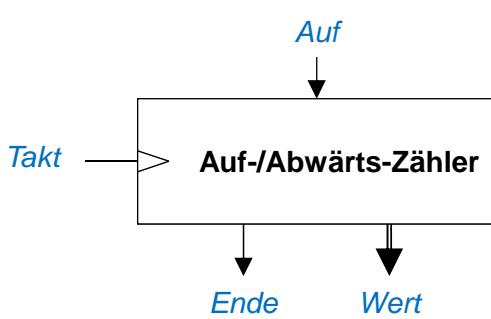


## Kombinierter Mealy/Moore-Automat

Ein Mealy-Automat, der auch Moore-Ausgänge enthält, wird auch "kombinierter Mealy/Moore-Automat" genannt und kann durch das folgende allgemeine Blockschaltbild realisiert werden:



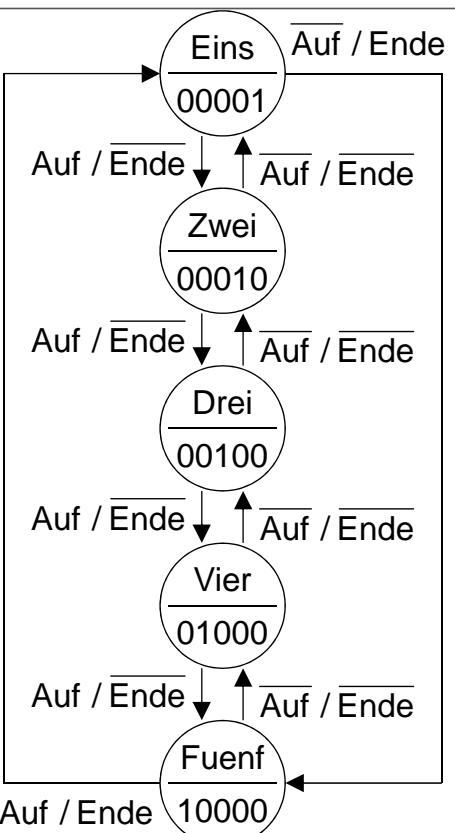
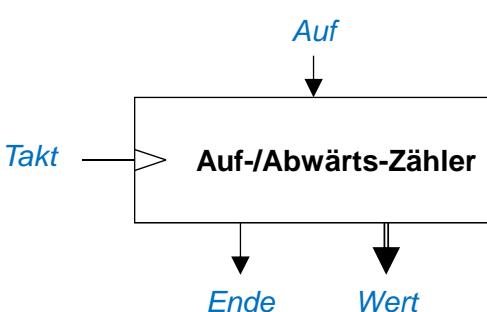
## Beispiel: Auf-/Abwärtszähler



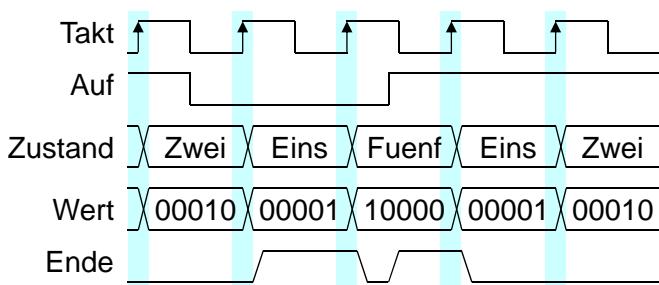
### Beschreibung:

- Mit 5 Werten auf- bzw. abwärts zählen
- Werte sind: 00001, 00010, 00100, 01000 und 10000  
Sie sollen während der Taktperiode stabil sein
- Bei kleinstem Wert mit Auf=0 (abwärts zählen) sowie größtem Wert mit Auf=1 (aufwärts zählen) wird ein Ende-Signal erzeugt

## Auf-/Abwärtszähler: Zustandsdiagramm



### Zeitdiagramm



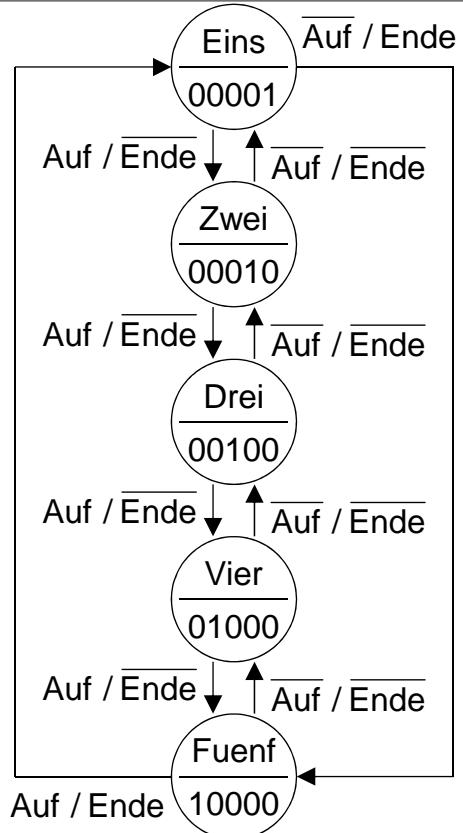
# Auf-/Abwärtszähler: Zustandskodierung

Realisierung erfordert Zuordnung der Zustände zu Binärwerten (**Zustandskodierung**)

- Dazu gibt es vielfältige Möglichkeiten (z. B. als Dualzahl oder dekodiert → One-Hot-Kodierung wie Moore-Ausg.)
- Kann manuell oder automatisch durch ein Synthesetool erfolgen
- Wahl der Kodierung beeinflusst Aufwand (Schaltnetzen und Registerbreite)

## Beispielhafte Kodierung:

Zustand	Kodierung
Eins	000
Zwei	001
Drei	010
Vier	100
Fuenf	111

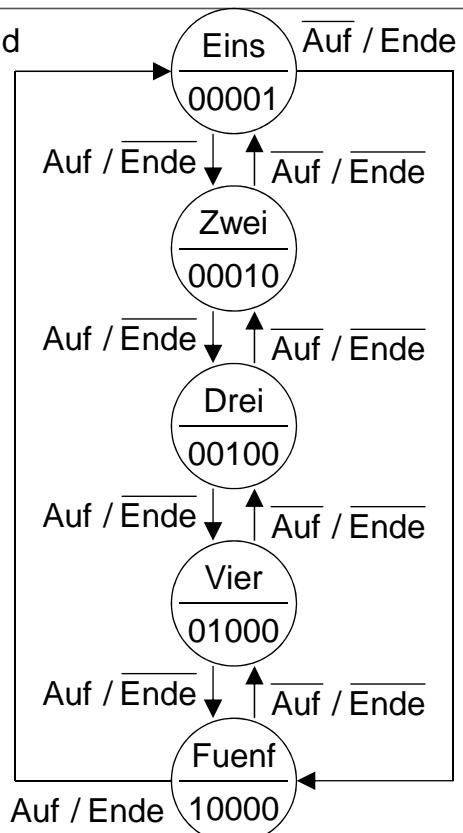


# Auf-/Abwärtszähler: Folgezustände

Das Zustandsdiagramm definiert, welcher Zustand (unter Berücksichtigung der Eingangswerte) als nächstes eingenommen werden muss

Der **Folgezustand** wird durch das Schaltnetz berechnet:      (--- = don't care)

Auf	Zustand	Folgezustand
0	000 (Eins)	111 (Fuenf)
0	001 (Zwei)	000 (Eins)
0	010 (Drei)	001 (Zwei)
0	011	---
0	100 (Vier)	010 (Drei)
0	101	---
0	110	---
0	111 (Fuenf)	100 (Vier)
1	000 (Eins)	001 (Zwei)
1	001 (Zwei)	010 (Drei)
1	010 (Drei)	100 (Vier)
1	011	---
1	100 (Vier)	111 (Fuenf)
1	101	---
1	110	---
1	111 (Fuenf)	000 (Eins)



# Auf-/Abwärtszähler: Berechnung der Ausgänge

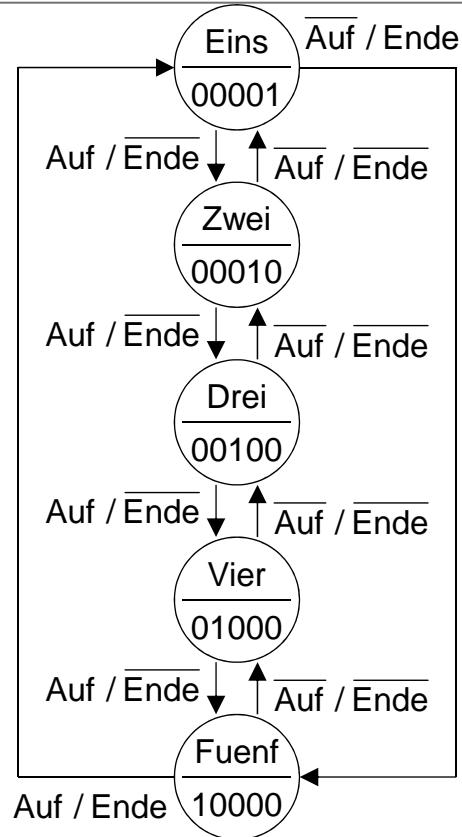
Neben den Folgezuständen bestimmt das Schaltnetz den Wert der Ausgänge

**Mealy-Ausgänge** werden mit Hilfe des Zustands und der Eingangswerte definiert:

Auf	Zustand	Ende
0	000 (Eins)	1
1	111 (Fuenf)	1
	Rest	0

**Moore-Ausgänge** werden nur mit Hilfe des Zustands definiert:

Zustand	$Y_{\text{Moore}}$
000 (Eins)	00001
001 (Zwei)	00010
010 (Drei)	00100
011	-----
100 (Vier)	01000
101	-----
110	-----
111 (Fuenf)	10000

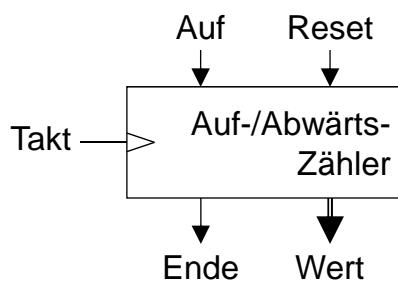


# Auf-/Abwärtszähler: Realisierung

**Was soll zum Beispiel nach dem Einschalten passieren ?**

*Erforderlich:* Einnehmen eines definierten Zustands (Startzustand  $z^0$ )  
 → Einschaltbedingung durch zusätzlichen Eingang kennzeichnen („Reset“)

Was würde ohne „Reset“ passieren?



**Weitere Anforderung:** Eingänge eines Endl. Automaten müssen durch D-FFs synchronisiert werden, damit sie zur Taktflanke einen definierten Wert haben.  
 Sonst ist Signal in Übergangslogik evtl. undefiniert (zwischen H und L) und kann zu nicht definierten Zuständen des Automaten führen!

# Das Wichtigste: Synchrone Schaltung/Endlicher Automat

## ► Synchrone Schaltung

- Alle Zustandssignale werden in taktflankengesteuerten Flipflops gespeichert.
- Ein globales Taktsignal synchronisiert Zustandsübergänge.
- Einfacher zu entwerfen als asynchron rückgekoppelte Schaltungen.

## ► Endlicher Automat

- Besteht aus: Mengen  $E$ ,  $A$ ,  $Z$ ; Startzustand  $z^0$ ; Funktionen  $f_z$  und  $f_a$
- Moore-Ausgänge hängen nur vom Zustand ab.
- Mealy-Ausgänge hängen vom Zustand und von Eingängen ab.
- Moore-Automaten haben nur Moore-Ausgänge. Sonst spricht man von Mealy-Automaten oder auch von *kombinierten Mealy/Moore-Automaten*.
- Endliche Automaten werden durch Flipflops (für Zustand) und Schaltnetzen (für Übergangs- und Ausgabefunktion) realisiert.

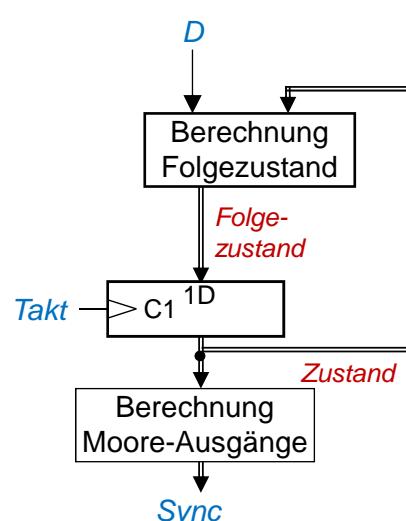
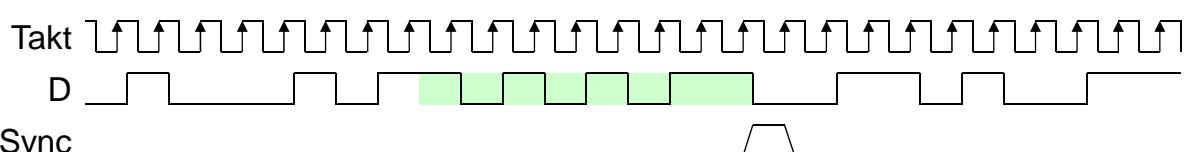
# Endlicher Automat zum Empfang serieller Daten

## Beispiel: Erkennung von Synchronisationssequenzen in seriellen Daten

Beim Empfang serieller Daten in Datennetzen (z.B. Ethernet) ist es notwendig, Synchronisationspunkte im Datenstrom zu finden, die den Beginn von Datenpaketen markieren.

- Seriellen Datenpaketen wird im Beispiel die Synchronisationssequenz (Präambel) „10101011“ (links nach rechts) vorangestellt.
- Die Datensequenz, in welche die Präambel eingebettet ist, wird sequentiell mit jedem Takt über ein Signal  $D$  empfangen.

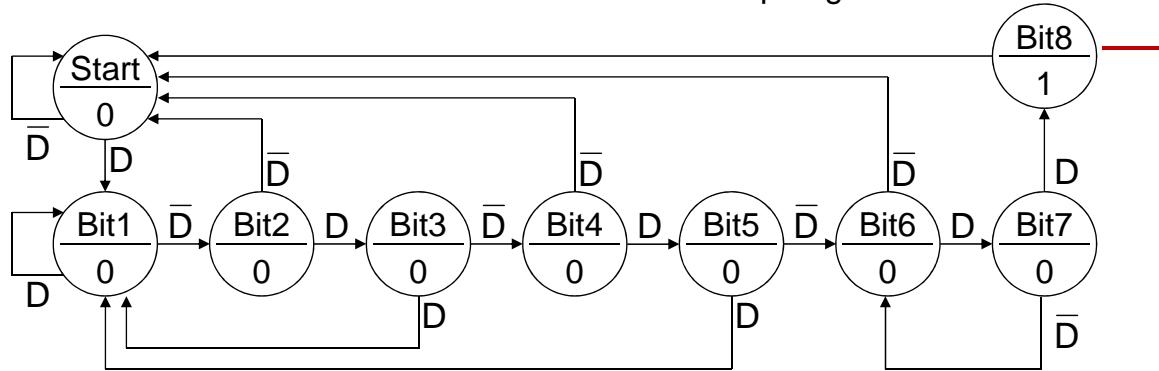
Die Erkennung solcher Präambeln kann mit einem **Moore-Automat** erfolgen



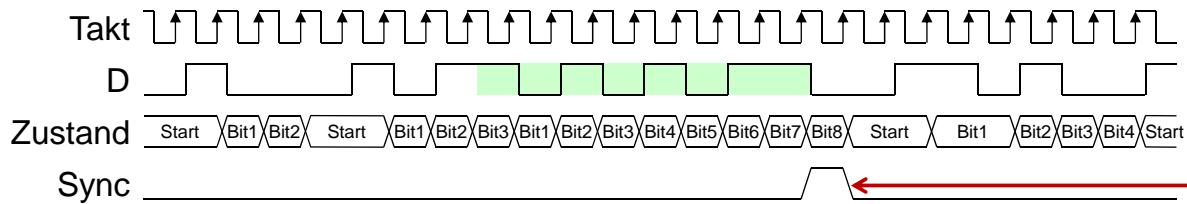
# Endlicher Automat: Präambel-Erkennung

Zustandsgraph zur Erkennung der Sequenz „10101011“

Zustand "Bit $n$ " bedeutet: "Bit 1 bis  $n$  wurde korrekt empfangen"



Zeitdiagramm



## Zähler

# Aufwärtszähler

Bei **Verwendung von T-Flip-Flops** im Register ergibt sich beim Entwurf von Zählern eine Schaltnetz mit geringem Aufwand:

Wert	Neuer Wert	
0 0 0 0	0 0 0 1	Wechsel in Bit 0
0 0 0 1	0 0 1 0	Wechsel in Bits 1,0
0 0 1 0	0 0 1 1	Wechsel in Bit 0
0 0 1 1	0 1 0 0	Wechsel in Bits 2,1,0
0 1 0 0	0 1 0 1	Wechsel in Bit 0
0 1 0 1	0 1 1 0	Wechsel in Bits 1,0
0 1 1 0	0 1 1 1	Wechsel in Bit 0
0 1 1 1	1 0 0 0	Wechsel in Bits 3,2,1,0
1 0 0 0	1 0 0 1	Wechsel in Bit 0
1 0 0 1	1 0 1 0	Wechsel in Bits 1,0
1 0 1 0	1 0 1 1	Wechsel in Bit 0
1 0 1 1	1 1 0 0	Wechsel in Bits 2,1,0
1 1 0 0	1 1 0 1	Wechsel in Bit 0
1 1 0 1	1 1 1 0	Wechsel in Bits 1,0
1 1 1 0	1 1 1 1	Wechsel in Bit 0
1 1 1 1	0 0 0 0	Wechsel in Bits 3,2,1,0 (Überlauf)

Toggle-Bedingungen  
zum Aufwärtszählen:

$$T(0) = 1$$

$$T(i) = T(i-1) \wedge \text{Wert}(i-1)$$

$$\text{Überlauf} = T(N)$$

## Aufwärtszähler II

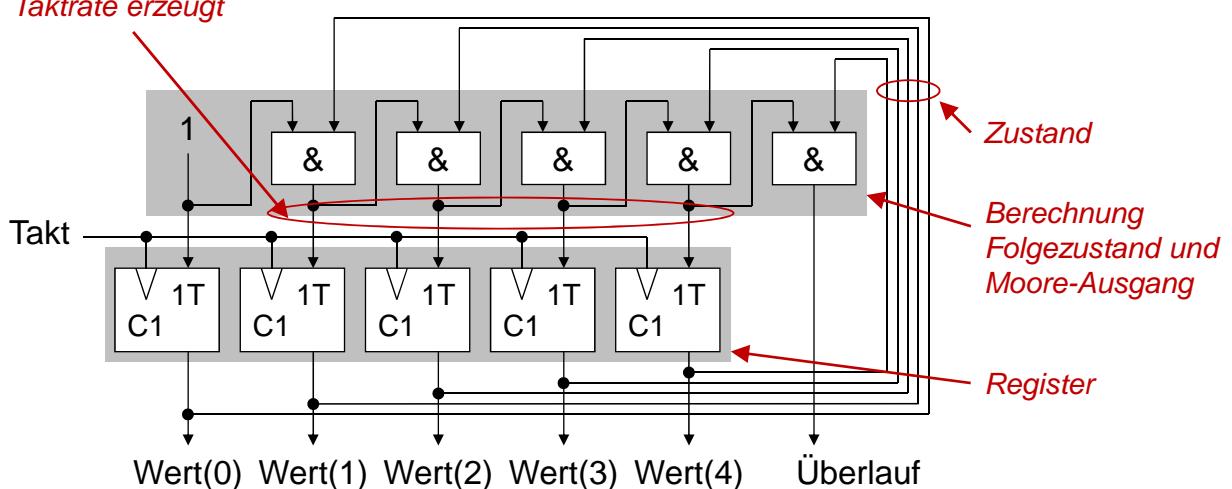
Toggle-Bedingungen zum Aufwärtszählen:

$$T(0) = 1$$

$$T(i) = T(i-1) \wedge \text{Wert}(i-1)$$

$$\text{Überlauf} = T(N)$$

Hier liegen werden  
Taktsignale mit  $\frac{1}{2}$ ,  $\frac{1}{4}$  usw.  
Taktrate erzeugt



Dieser Zähler muss nicht direkt so implementiert werden. Er kann auch aus einem Addierer (ein Eingang konstant Eins) und einem D-Register aufgebaut werden!

# Anwendung Zähler: Pulsweitenmodulation (PWM) I

Zähler können zur *Pulsweitenmodulation* (PWM) verwendet werden.

**Definition des Tastverhältnisses eines Rechtecksignals:**

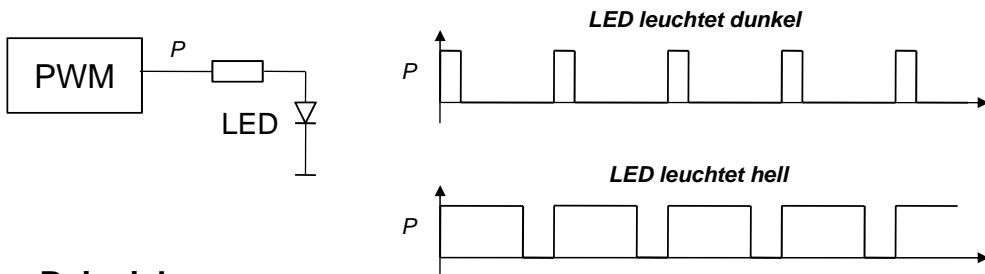
- $g = \text{Länge der 1-Phase} / \text{Periodendauer}$

Taktsignale haben normalerweise das Tastverhältnis 0,5 (50%)

**PWM: Einstellbares Tastverhältnis**

**Einfaches Anwendungsbeispiel: LED-Dimmer**

menschliches Auge nimmt nur den Mittelwert der LED-Helligkeit wahr



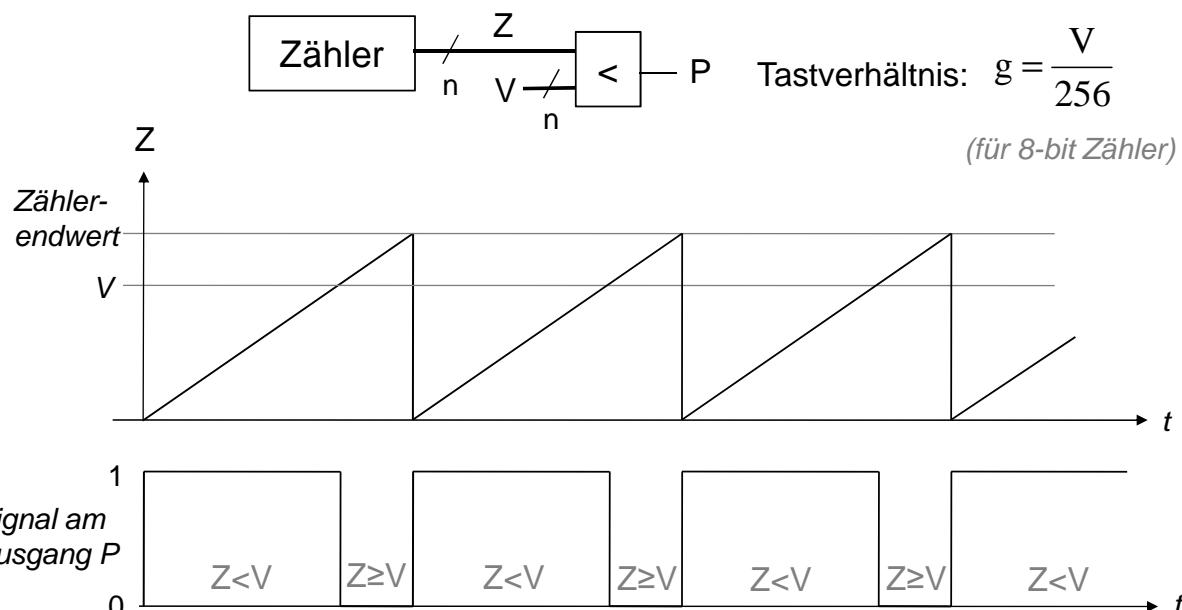
**Weitere Beispiele**

Motorsteuerung, Erzeugung von analogen Signalen etc.

# Anwendung Zähler: Pulsweitenmodulation (PWM) II

► **Wie kann eine variable Pulsweite erzeugt werden?**

- Kombination Endlos-Aufwärtzzähler (mit Überlauf) und Vergleicher
- Vergleichswert  $V$  bestimmt Tastverhältnis des Pulses  $P$



## Weitere Zähler

Neben Aufwärtzählern und Modulo-Zählern sind noch folgende Zähler gebräuchlich:

- ▶ **Zähler mit Freigabesignal EN: Zählt nur weiter, wenn EN=1**
- ▶ **Abwärtzähler**
- ▶ **Kombinierte Auf-/Abwärtzähler (Steuereingang Auf)**
- ▶ **Zähler ohne Bereichsüberschreitung (hört bei Endwert auf zu zählen)**

## Asynchrone Zähler

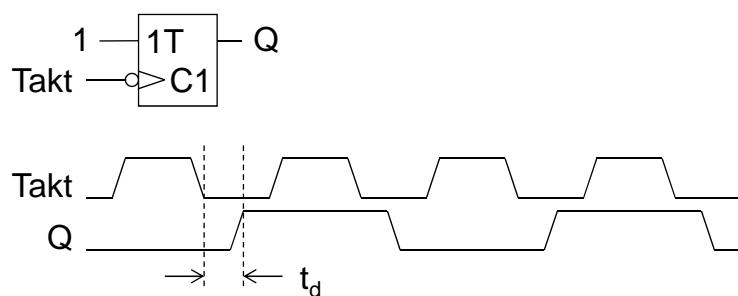
**Asynchrone Zähler basieren auf Frequenzteilern**

Der Ausgang des ersten Frequenzteilers dient als Eingang des zweiten Teilers.

Damit werden die Flip-Flops **nicht mehr synchron**

mit einem gemeinsamen Taktsignal sondern  
asynchron mit individuellen Takten angesteuert.

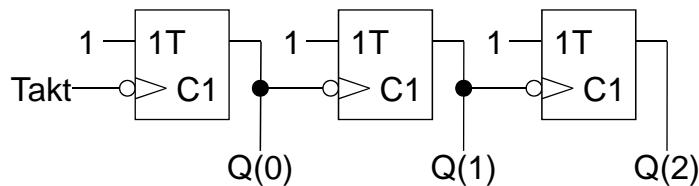
Ein **einfacher Frequenzteiler** kann durch ein T-FlipFlop mit  $T=1$  realisiert werden:



Der Ausgang Q des T-FlipFlops liefert ein Signal mit der halben Frequenz des Takt-Signals.

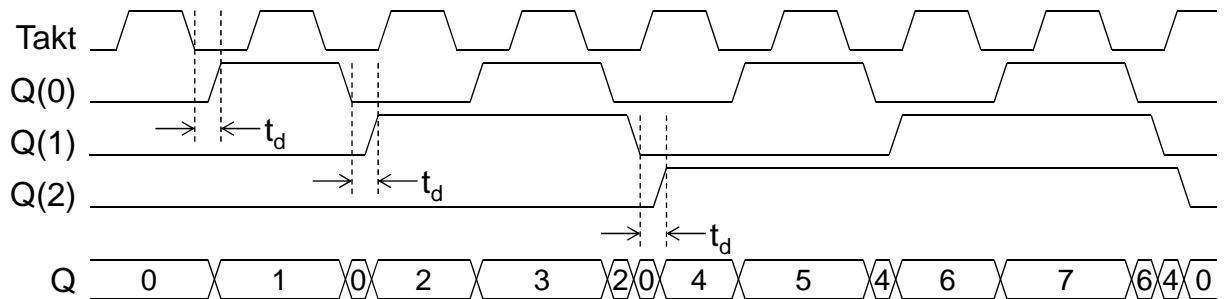
## Asynchrone Zähler II

Verbindet man den Ausgang Q eines T-Flip-Flops mit dem Takt-Eingang des nachfolgenden T-Flip-Flops so ergibt sich ein asynchroner Zähler:



Die aktive Flanke der T-Flip-Flops ist die fallende Flanke.

Das **Zeitdiagramm** zeigt die Problematik des Zählers:



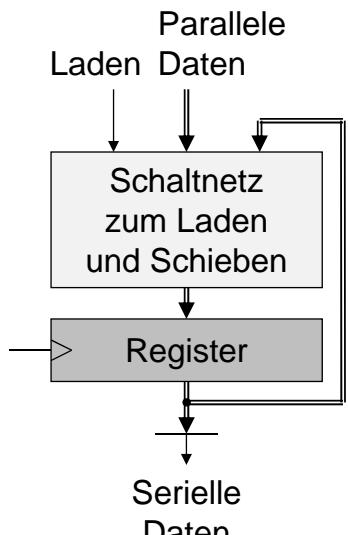
**Asynchrone Zähler sind nur als Frequenzteiler geeignet!**

## Schieberegister



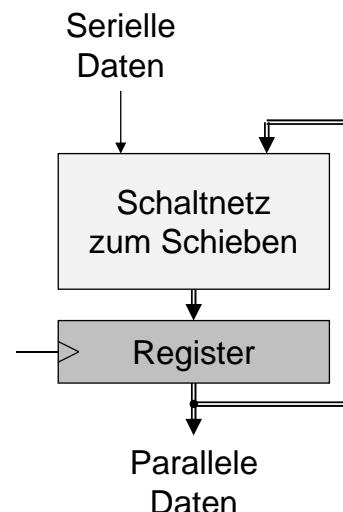
# Anwendungen von Schieberegistern

## Parallele Daten serialisieren (Parallel-Seriell-Wandlung)



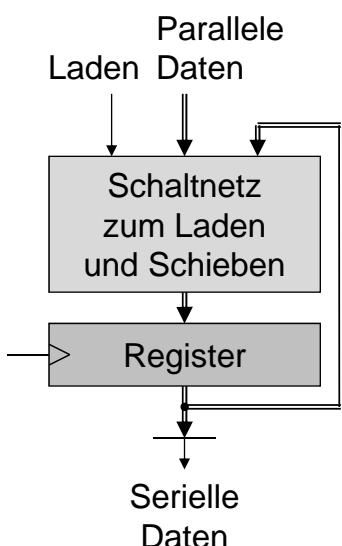
serieller Sender (z. B. Modem)

## Serielle Daten parallelisieren (Seriell-Parallel-Wandlung)

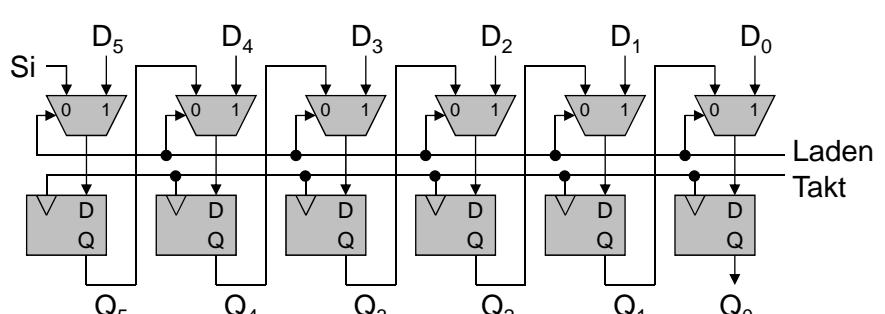


serieller Empfänger (z. B. Modem)

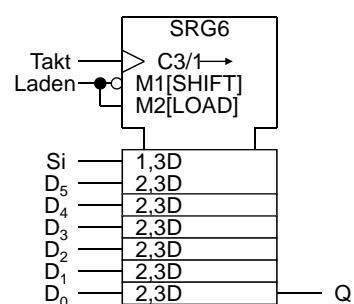
## Parallel-Seriell-Wandler



Beispiel: Parallel ladbares Rechts-Schieberegister

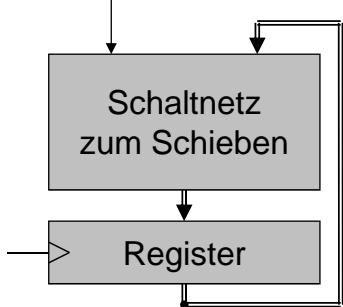


DIN-Symbol:

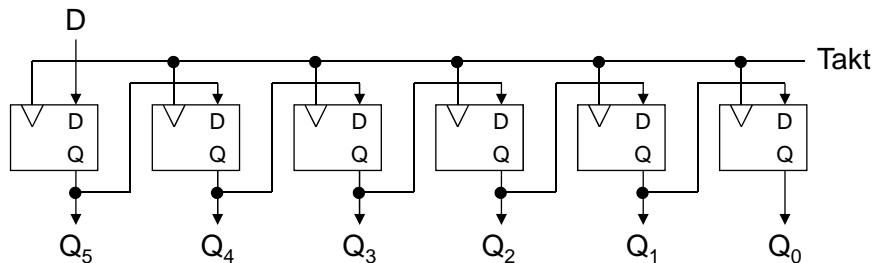


# Seriell-Parallel-Wandler

Serielle  
Daten

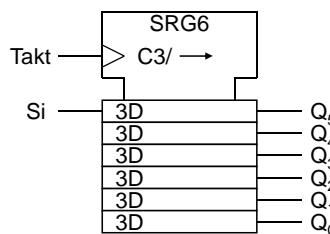


Beispiel: Rechts-Schieberegister



Parallele  
Daten

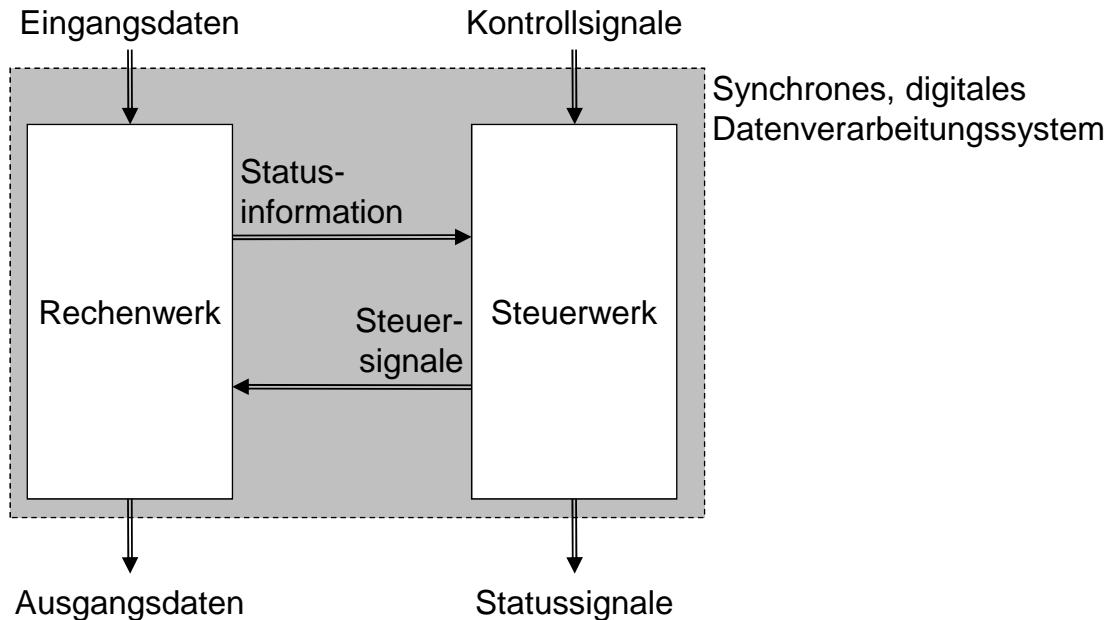
DIN-Symbol:



## Systeme zur synchronen Verarbeitung digitaler Daten

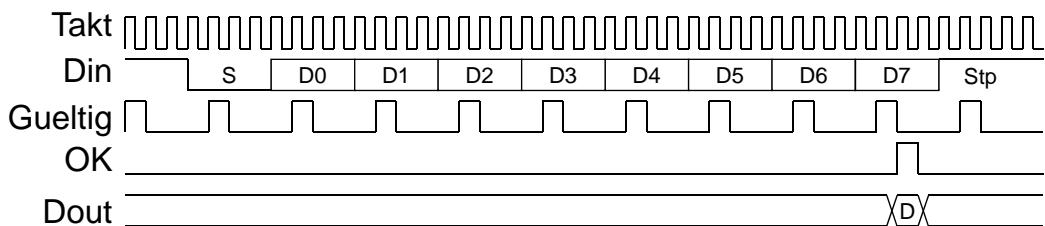
# Basisarchitektur

- Die Verarbeitung der Daten erfolgt im Rechenwerk  
(Kombination vpm Operatoren, Multiplexer, Registern)
  - Das Steuerwerk bestimmt den zeitlichen Ablauf zur Verarbeitung der Daten  
(als EA spezifiziert)

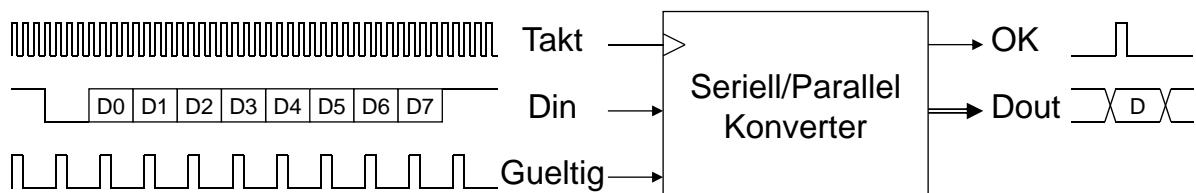


## Beispiel: Parallelwandlung serieller Daten

- Serielle Übertragung eines Datenworts wird durch 0-Startbit S eingeleitet.
  - Es folgen die Datenbits D0 bis D7.
  - Abschluss der Übertragung mit einem 1-Stopbit Stp.
  - Gültig-Signal wird von einem separaten Automaten geliefert.

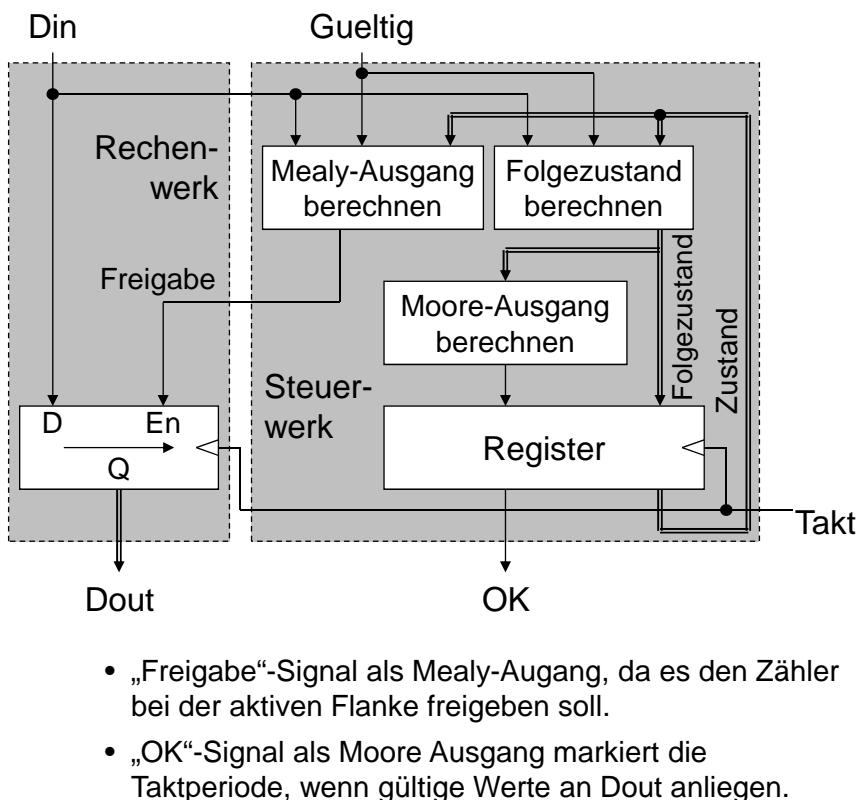


- In einem Seriell/Parallel Konverter müssen das Stopbit erkannt und die Datenbits aufgesammelt werden.
  - Nach Abschluss der Übertragung wird das empfangene Datenwort parallel ausgegeben und mit einem Signal OK markiert.

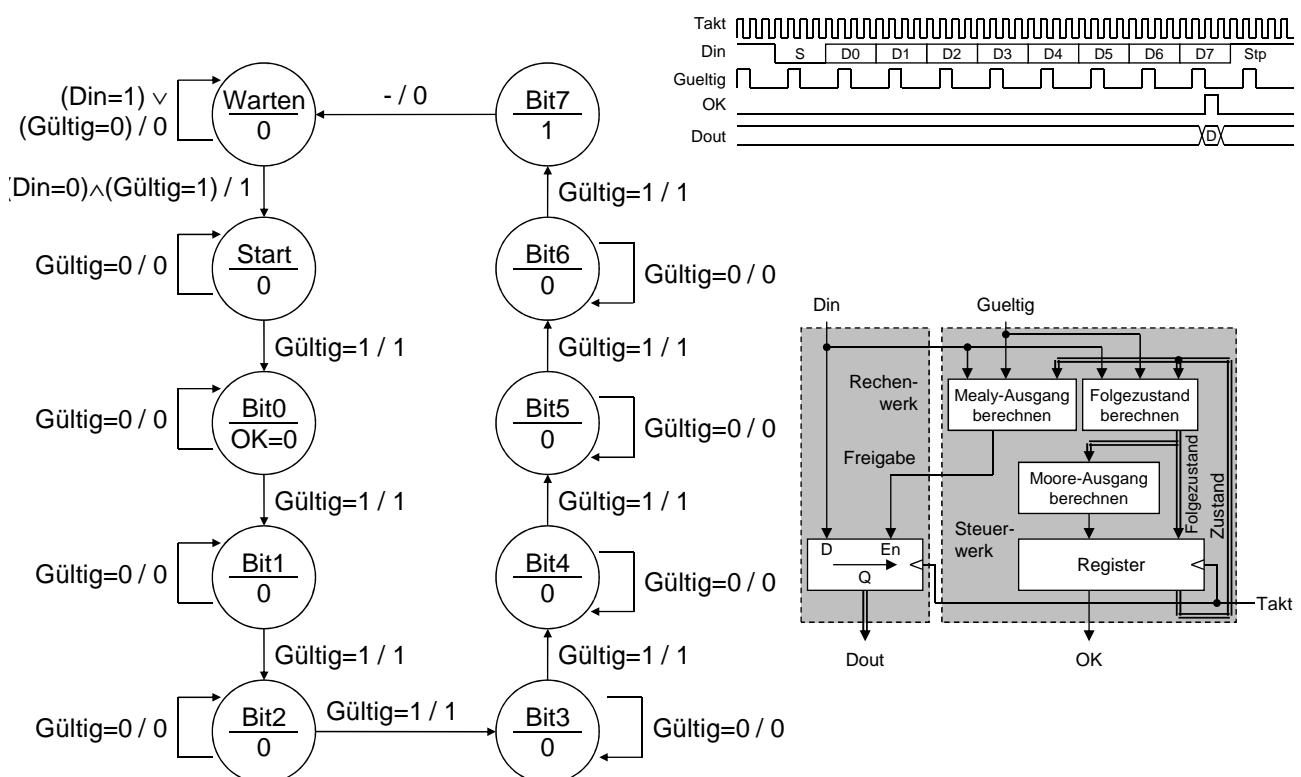


## Beispiel: Parallelwandlung serieller Daten II

- Realisierung mit einem System aus Rechen- und Steuerwerk.
- Im Rechenwerk erfolgt die Seriell/Parallel-Wandlung mit einem Schieberegister.
- Im Steuerwerk wird auf das Startbit gewartet, danach werden die Datenbits gezählt. Sind alle Datenbits empfangen, wird das OK-Signal für einen Taktzyklus aktiviert.



## Parallelwandlung: Zustandsdiagramm

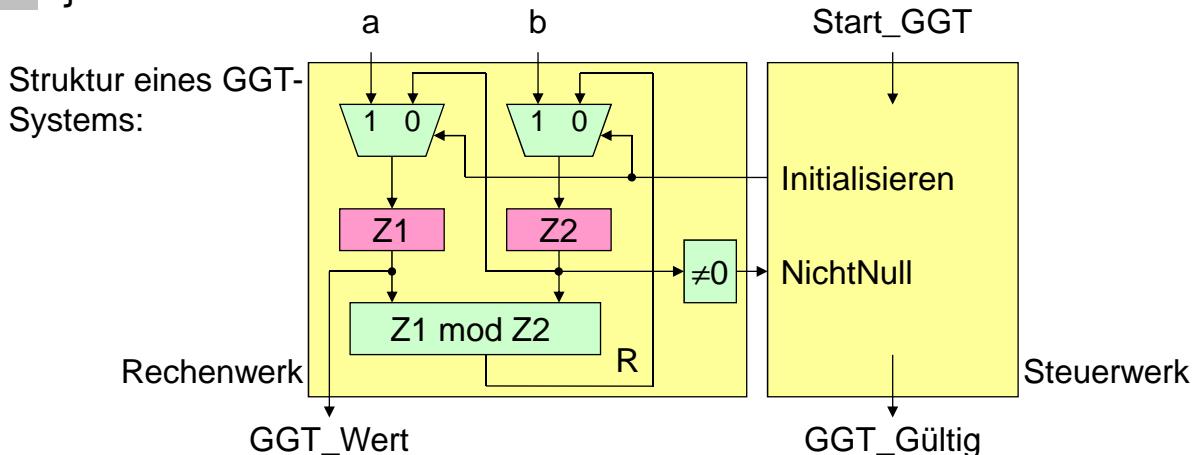


# Berechnung des GGT

Euklid'scher Algorithmus:

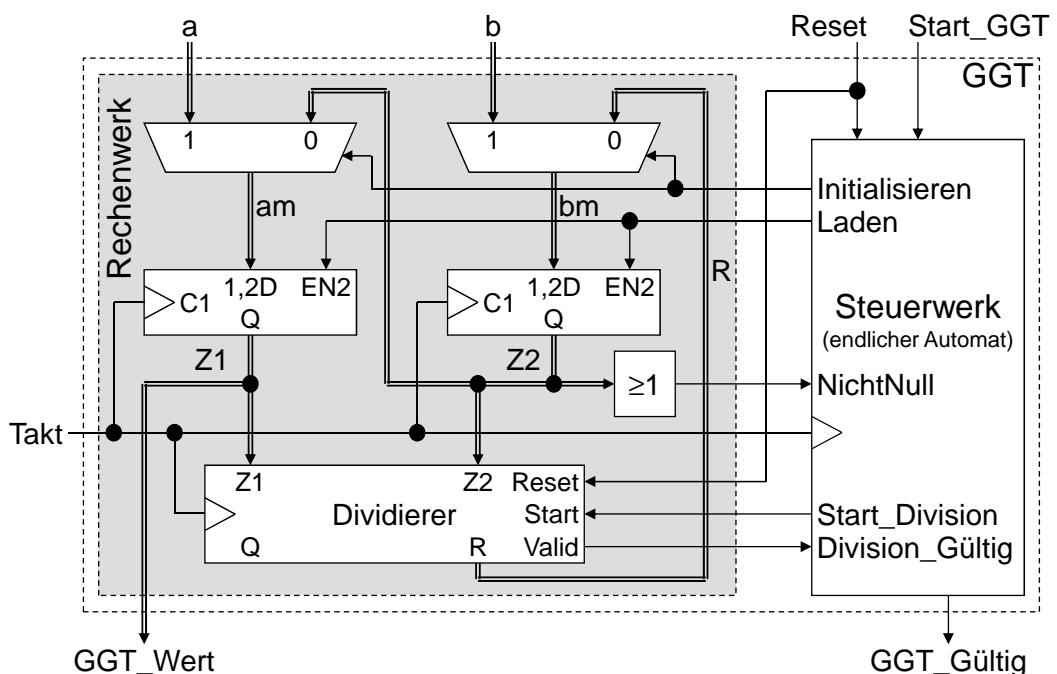
```

1: GGT(a,b) {
2:   Z1=a; Z2=b;
3:   while (Z2≠0) {
4:     R = Z1 mod Z2; -- Rest der Ganzzahldivision
5:     Z1=Z2; Z2=R;
6:   }
7:   return Z1;
8: }
```

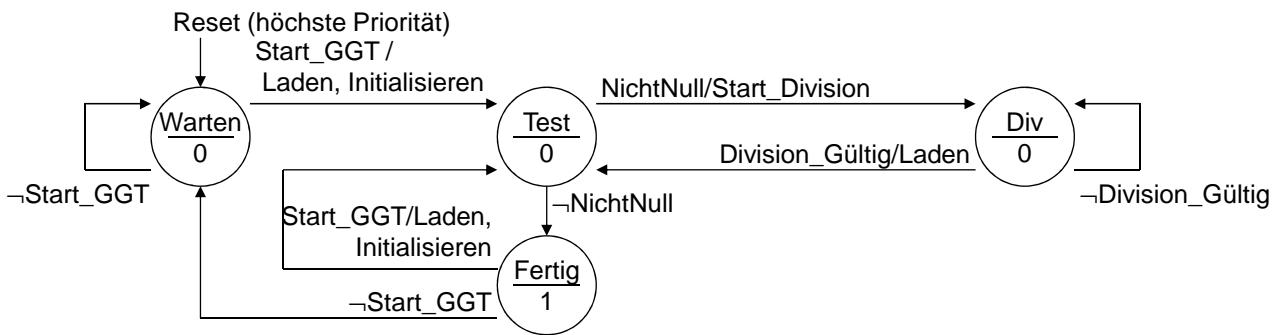


## Rechenwerk zur GGT-Berechnung

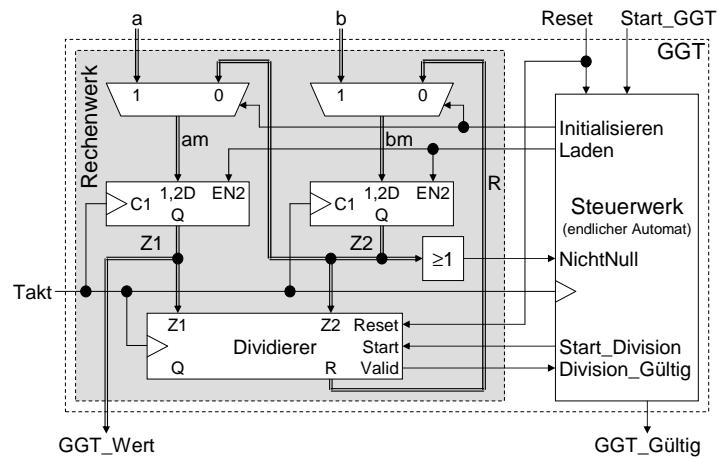
Rechenwerk mit serielllem Dividierer



# Zustandsdiagramm des GGT-Systems



- Moore-Ausgang ist „GGT\_Gültig“.
- Mealy-Ausgänge sind nur angegeben, wenn sie aktiv (Wert 1) sind.





---

# 11. Strukturbeschreibung in VHDL

**Einführung**

**Entity und Architecture**

**Nebenläufige Signalzuweisungen**

**Strukturbeschreibung**



## Einführung

**VHDL:** Very high speed integrated circuit **H**ardware **D**escription **L**anguage

**Ziel:** Beschreibung digitaler Hardware-Komponenten / -Systeme  
Beschreibung von drei Aspekten digitaler Systeme

Zeitliches Verhalten (time)

- Was passiert wann?

Logisches Verhalten (behavior)

- Was soll kombiniert/gespeichert/berechnet werden?

Struktur (structure)

- Welche Komponenten werden wie zusammengesetzt und verknüpft?

# Einführung

VHDL: Very high speed integrated circuit Hardware Description Language

- ▶ Normierung im Standard IEEE 1076
- ▶ VHDL-AMS
  - Erweiterung für die Beschreibung analoger Komponenten
  - Beschreibt z. B. das elektrische Signalverhalten eines Temperatursensors als  $U=f(T)$
  - Integrale, Ableitungen und Differentialgleichungen möglich
  - (Zur Zeit) Keine Synthese von analogen Schaltungen möglich
  - Simulatoren sind verfügbar
- ▶ Alternativen
  - Verilog(-HDL): Direkter „Konkurrent“ (Konzeption ähnlich zu VHDL)
  - Schematic Entry: Graphische Eingabe eines Schaltplans
  - SystemC: Meist zur Modellierung (weniger zum Entwurf) komplexer Systeme, vereinzelt auch für HW-Entwurf (Behavioral Synthesis)

## Beschreibung mit Hardwarebeschreibungssprache

Die Beschreibung von Schaltungen durch eine Hardwarebeschreibungssprache (HDL) erscheint ähnlich wie die Programmierung in modernen, sequentiellen Hochsprachen, dahinterliegende Paradigmen unterscheiden sich jedoch grundsätzlich:

### Sequentielle Programmiersprachen:

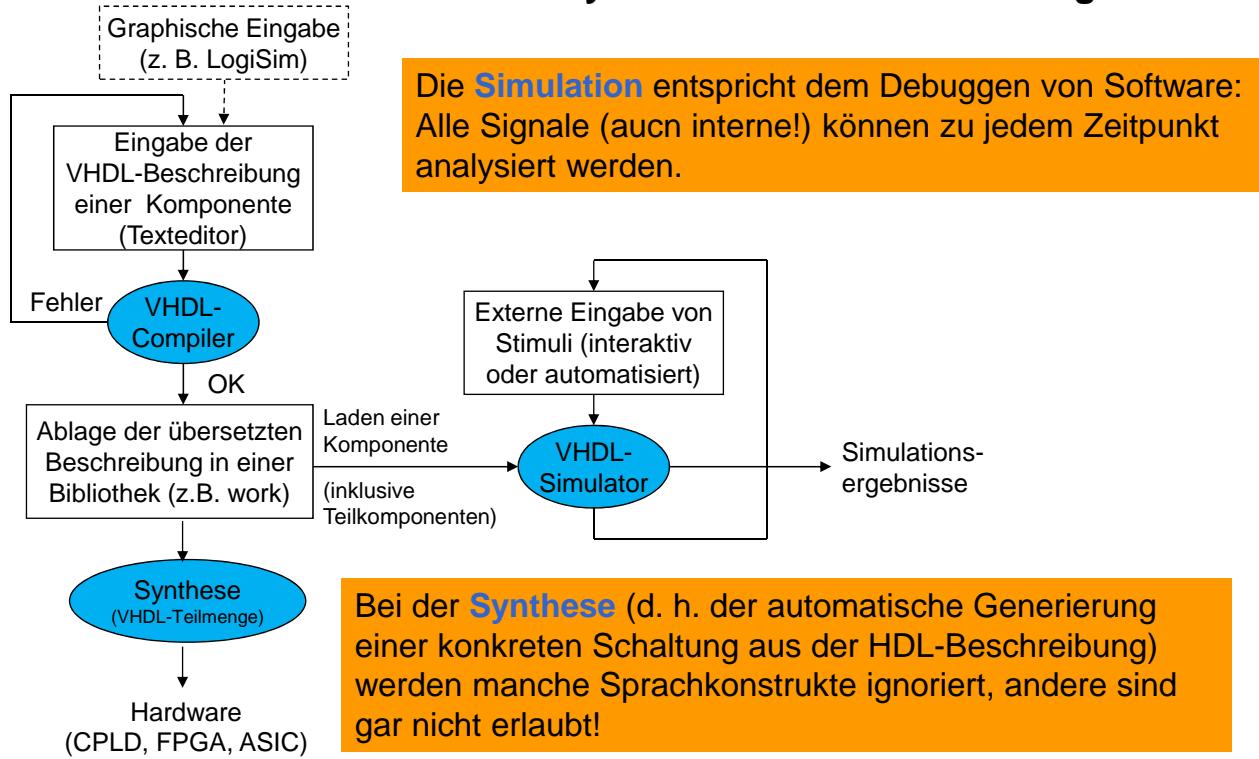
Die Anweisungen werden **sequentiell**, eine nach der anderen abgearbeitet. Parallel Bearbeitung von Programmteilen muss explizit (z.B. durch Verwendung von parallelen Prozessen oder Threads) erzwungen werden.  
*Sprachen wie C/C++ erlauben nur die Verhaltens-Beschreibung!*

### Hardwarebeschreibungssprachen:

Alle Anweisungen werden **parallel** ausgeführt, wie dies auch in einer realen Hardware erfolgt.  
Eine sequentielle Bearbeitung von Instruktionen muss durch spezielle Sprachkonstrukte erzwungen werden.

# VHDL Design Flow

## Übersetzen und Simulieren bzw. Synthetisieren von VHDL-Programmen



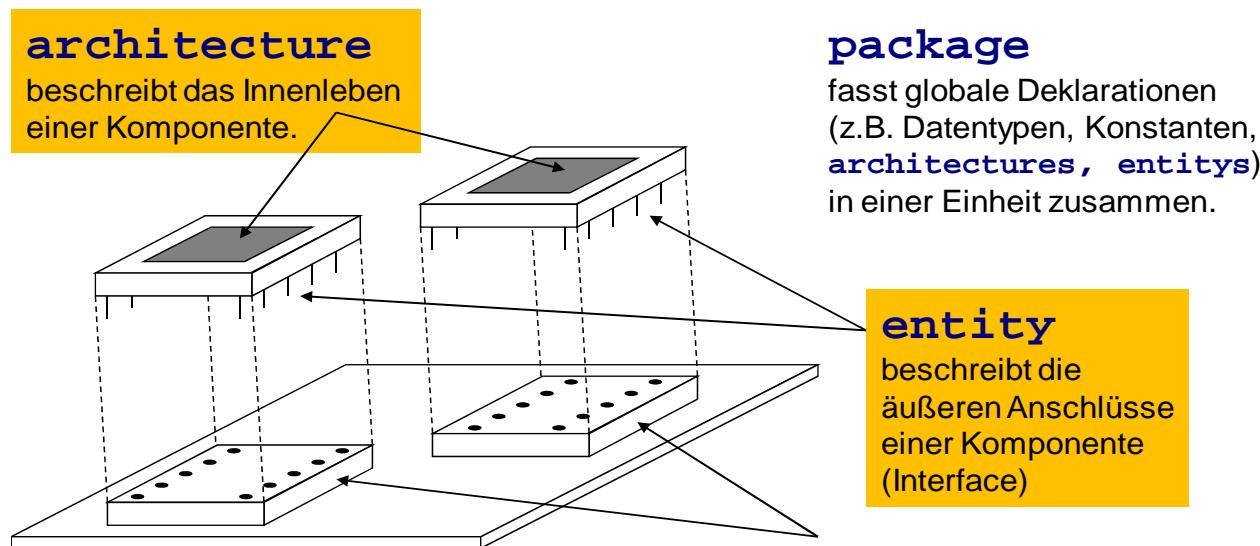
## Grundlagen VHDL-Syntax

- ▶ **Lexikalische Einheiten**
  - Keywords (Schlüsselwörter), Bezeichner, Operatoren, Konstanten etc.
- ▶ **Bezeichner**
  - Beginnen mit Buchstaben, enthalten alphanumerische Zeichen und ,\_‘
- ▶ **Keine Unterscheidung zw. Groß- und Kleinschreibung**
  - bei Keywords, Bezeichnern etc., sondern nur bei Zeichen ('a', '0',...) und Zeichenketten ("string", "0101",...)
- ▶ **Kommentare**
  - Durch zwei Bindestriche eingeleitet, gelten bis zum Zeilenende:  
-- Dies ist ein Kommentar

# Entity und Architecture

## Entwurfseinheiten (Design Units) in VHDL

Eine einfache VHDL-Datei besteht aus einer Entity- und einer Architecture-Definition!



Mehrere Komponenten (*design units*) können hierarchisch in einer übergeordneten **entity** zusammengefasst werden.

# Syntax Entity-Definition I

Entity-Definition zur Beschreibung des Interface einer Komponente:

```
entity Entity_Name is
  [generic (Generische_Parameterliste);]
  [port (Port_Parameterliste);]
end [Entity_Name];
```

Eine Port\_Parameterliste besteht aus einem oder mehreren Port\_Parameter, die durch ; getrennt werden. Sie beschreiben die Ein- und Ausgänge der Komponente.  
Ein Port\_Parameter hat folgende Syntax:

Port\_Name: (in | out) Datentyp [ := Konstante ]

Die wichtigsten vordefinierten Datentypen sind:

- Bit (für Signale, Werte '0'/'1')
- Integer (v. a. für generische Parameter)
- Time (für Verzögerungszeiten, Einheit ps, ns, us etc.)
- Boolean (v. a. für Ergebnis von Vergleichen in Bedingungen, Werte true/false)

Regeln der Syntax-Definitionen:

- Direkt übernommene Zeichen und Schlüsselwörter: **fett**
- Platzhalter für Bezeichner oder anderen VHDL-Codes: *kursiv*
- Optionale Elemente: in [ eckigen Klammern ]
- Alternativen: durch vertikale Striche | getrennt

# Syntax Entity-Definition II

Eine Generische\_Parameterliste besteht aus einem oder mehreren Generischer\_Parameter, die durch ; getrennt werden.

Ein Generischer\_Parameter hat folgende Syntax:

Parameter\_Name: Datentyp [ := Konstante ]

Diese Parameter beschreiben Konstanten, die in der zur Entity gehörigen Architecture-Definition zugreifbar sind, z B. Größe, Verzögerungszeit etc.

Bei allen Parametern ist Konstante ein Default-Wert, der zum Datentyp passen muss. Wird der Parameter bei der Instanzierung (als Teil einer größeren Komponente) nicht zugewiesen, gilt der Default-Wert.

**Beispiel:** Interface BeispielKomponente mit Eingängen X1, X2 und Ausgang Y:

```
entity BeispielKomponente is
  generic (td: time := 6 ns);
  port (
    X1: in Bit;
    X2: in Bit;
    Y: out Bit
  );
end BeispielKomponente;
```

Alternativer Bit-Datentyp:  
std\_logic/std\_ulogic  
(auch für hochohmige  
Signale und mehr)

# Syntax Architecture-Definition

---

Architecture-Definition zur Beschreibung der Funktion einer Komponente:

```
architecture Architektur_Name of Entity_Name is
  Deklarationen
begin
  Nebenläufige_Anweisungen
end [Architektur_Name];
```

**Hinweis:** Die *Port\_Parameterliste* aus der Entity muss nicht wiederholt werden, sie steht in der Architecture automatisch zur Verfügung!

Folgende *Deklarationen* können auftreten:

- Typ-Deklarationen
- **Signal-Deklarationen**
- Konstanten-Deklarationen
- Komponenten-Deklarationen

Folgende *Nebenläufige\_Anweisungen* können auftreten:

- **Signalzuweisungen** (einfach, bedingt, oder ausgewählt)
- **Komponenten-Instanzierungen**
- Generate-Anweisungen
- Prozesse

## Nebenläufige Signalzuweisungen

# Signale und Signalzuweisungen

Signal-Deklaration zur Definition interner Signale hat folgende Syntax:

**signal Signal\_Name: Datentyp [ := Konstante ];**

Syntax der einfachen Signalzuweisung (Datentyp links und rechts muss übereinstimmen):

**Ergebnissignal <= Ausdruck [ after Zeitangabe ];**

- „Verdrahtet“ Ergebnissignal (internes Signal oder out-Port) mit dem Ergebnis eines Ausdrucks. Jedes Ergebnissignal darf nur einmal zugewiesen werden, um Kurzschlüsse zu verhindern. (Ausnahme: Tri-State-Busse mit speziellen Datentypen)

- Beispiel:** `outport1 <= (inport1 and sig1) xor inport2;`
- Ausdrücke verknüpfen interne Signale oder in-Ports durch Operatoren. Syntax:  
**Signal | in-Port | Konstante | (Ausdruck bop Ausdruck) | (unop Ausdruck)**
  - Operatoren für Datentyp Bit: binär (bop): `and | nand | or | nor | xor | xnor`  
unär (unop): `not`
  - Achtung:** Gleiche Priorität für `and` und `or` → Klammern!
  - **Ausdruck wird sofort ausgewertet, wenn sich ein Eingabesignal ändert!**  
Dadurch erhält auch das Ergebnissignal sofort einen neuen Wert!
    - ➔ Alle Signalzuweisungen und sonstigen nebenläufigen Anweisungen erfolgen gleichzeitig! Die Reihenfolge der Anweisungen ist egal!
    - ➔ Die Programmierung entspricht dem parallelen Datenfluß-Modell.
  - **Zeitangabe (optional)** bestimmt die Verzögerungszeit (nur für Simulation).

## Beispiel Architecture-Definition I

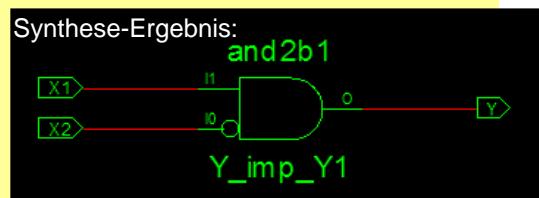
**Beispiel:** Verhaltensbeschreibung zur Entity *BeispielKomponente*

```
architecture Bsp_Architecture1 of BeispielKomponente is
    signal notX2: Bit;
begin
    Y      <= X1 and notX2;
    notX2 <= not X2;
end Bsp_Architecture;

entity BeispielKomponente is
generic (td: time := 6 ns);
port (
    X1: in Bit;
    X2: in Bit;
    Y:   out Bit
);
end BeispielKomponente;
```

Für jede Entity können mehrere Architectures existieren. Zur Simulation oder Synthese muss dann eine ausgewählt werden, z. B. durch *configuration*-Anweisung.

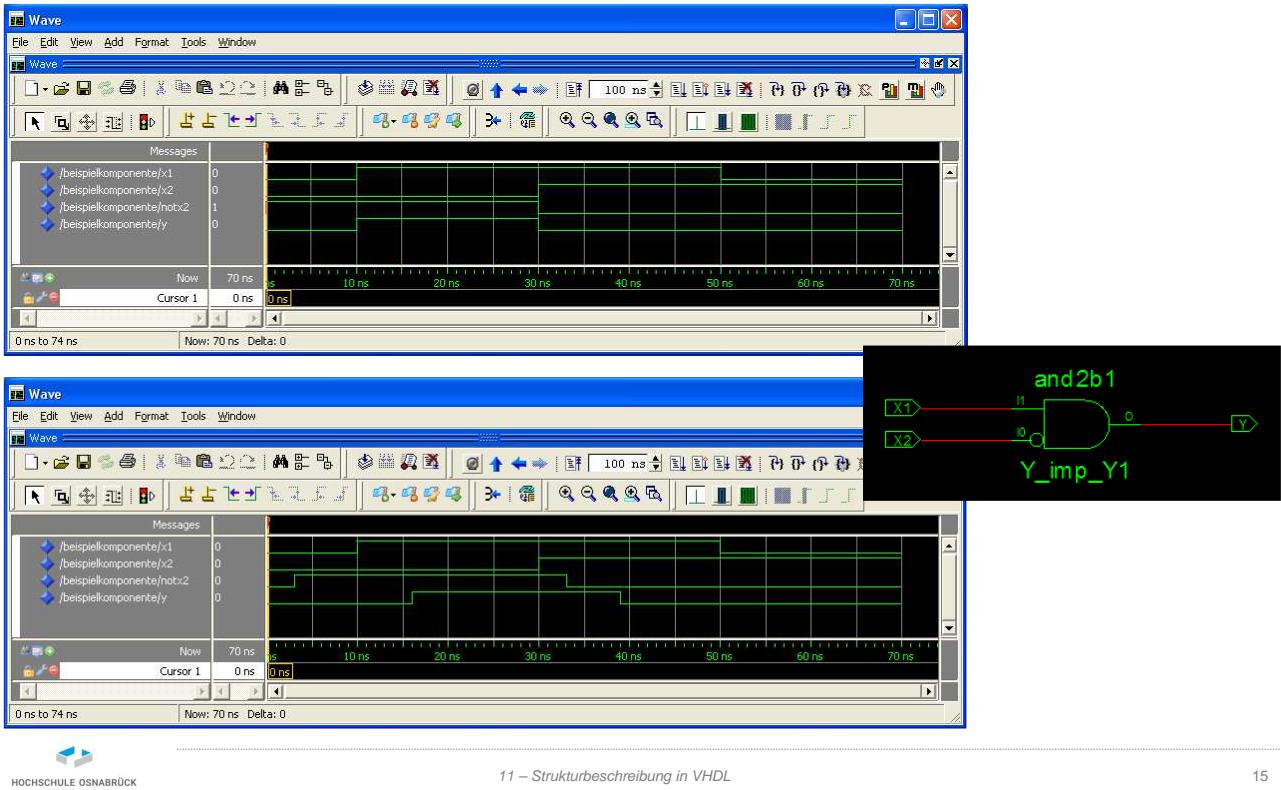
```
architecture Bsp_Architecture2 of BeispielKomponente is
    signal notX2: Bit;
begin
    Y      <= X1 and notX2 after td;
    notX2 <= not X2 after 3 ns;
end Bsp_Architecture;
```



**Aufgabe:** Beschreiben Sie KNF und DNF der Funktion aus Kap. 5/Folie 21/22 in VHDL!

# Beispiel Architecture-Definition II

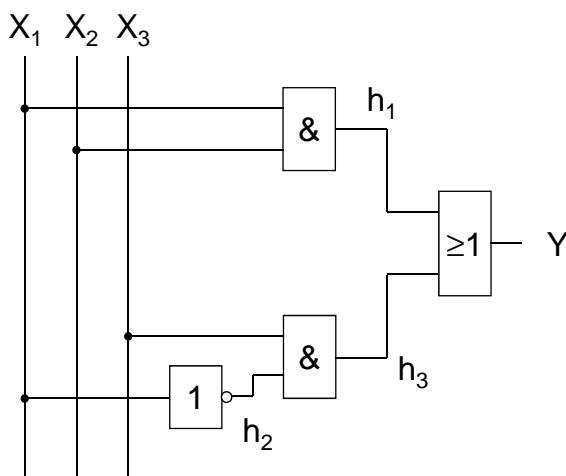
Simulationsergebnisse ohne (oben) und mit (unten) Verzögerungszeiten



## Beispiel: Hazard-Analyse mit VHDL

Schaltung mit statischem Logik-Hazard aus Kapitel 5:

Durch Beschreibung der Schaltung in VHDL (mit Zeitverhalten der Gatter)  
lässt sich der Hazard simulieren



```
entity MyCircuit is
  Port (
    X1, X2, X3: in Bit;
    Y:          out Bit
  );
end MyCircuit;
```

Ports können zusammengefasst werden!

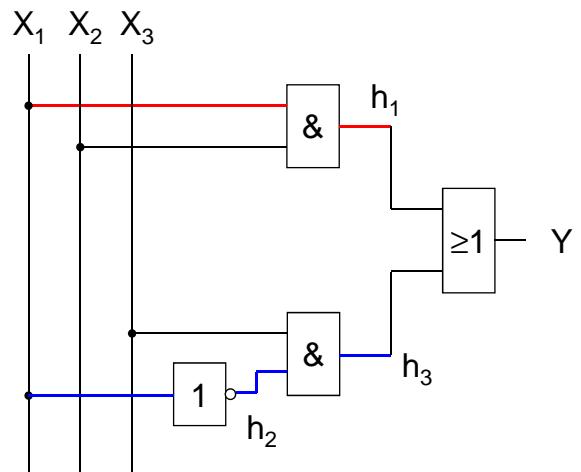
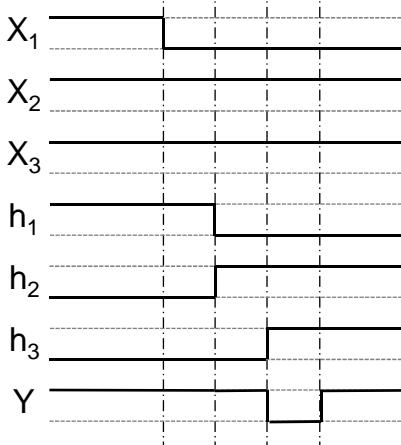
```
architecture myarch of MyCircuit is
  signal h1, h2, h3: Bit;
begin
  Y <= h1 or h3 after 1 ns;
  h1 <= X1 and X2 after 1 ns;
  h2 <= not X1 after 1 ns;
  h3 <= h2 and X3 after 1 ns;
end myarch;
```

## Wiederholung: Analyse des Beispiels

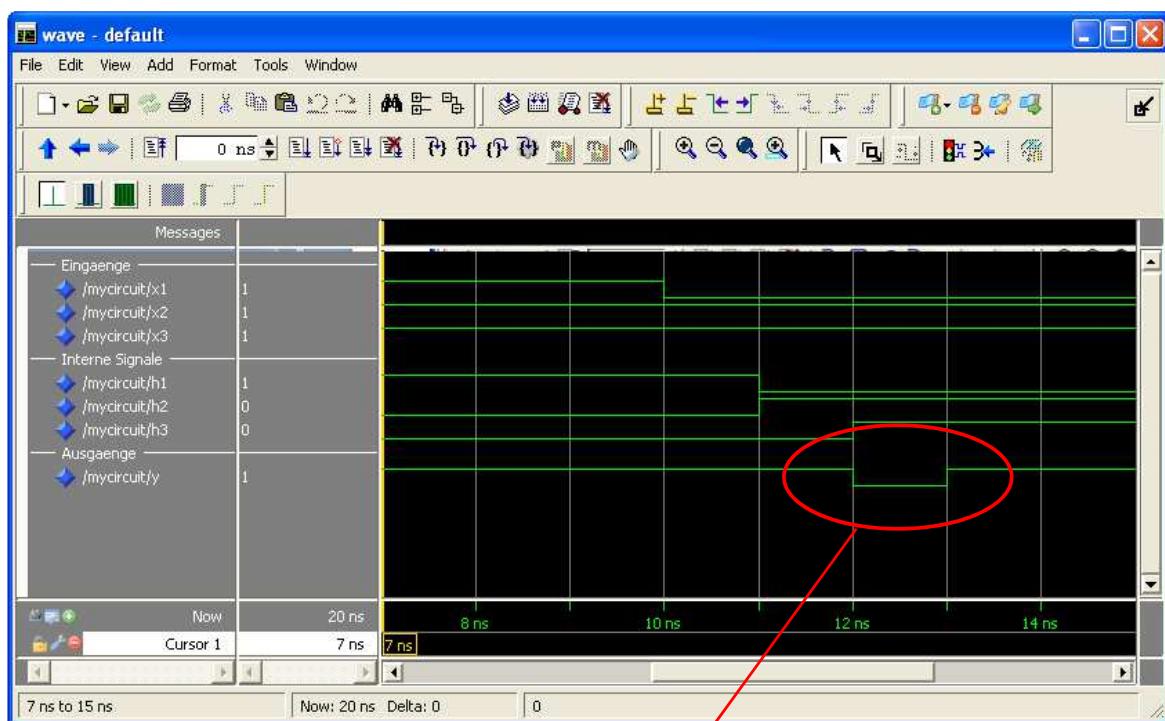
Übergang Eingang ( $X_3, X_2, X_1$ ):  $(1,1,1) \rightarrow (1,1,0)$

Erwartetes Verhalten Y:  $1 \rightarrow 1$

Das Signal  $X_1$  wird dem oberen UND-Gatter direkt zugeführt. Dem unteren UND-Gatter wird es invertiert und verzögert zugeführt. Daher ändern sich die Signale  $h_1$  und  $h_3$  nicht gleichzeitig. Für eine kurze Zeit sind beide Signale 0 und damit ist auch  $Y=0$ .



## Analyse Hazard beim Übergang ( $X_3, X_2, X_1$ ): $(1,1,1) \rightarrow (1,1,0)$



Eigentlich müsste Y gemäß Logiktabelle konstant 1 sein.

# Strukturbeschreibung

**Bibliothekskonzept**  
**Komponenten-Instanzierung**  
**Generate-Anweisungen**



## Bibliothekskonzept

- Zur Wiederverwendung (wie in SW-Bibliotheken) werden Design Units (entity, architecture etc.) in VHDL in **Bibliotheken (Libraries)** abgelegt. Sie enthalten Packages und Komponenten.
- Falls nichts anderes angegeben, werden alle Design Units in der Default-Library **work** abgelegt.
- Es existieren verschiedene vordefinierte Bibliotheken. Die wichtigsten sind:
  - std** : Packages für die Basisdatentypen, Text-IO, usw.
  - ieee** : Standardisierte Erweiterungen, z. B. `std_[u]logic`

Vor der **Verwendung von Bibliotheken** sind zwei Schritte erforderlich

1. Bekanntmachen der Bibliothek    `library ieee;`
2. Auswahl des Packages                `use ieee.std_logic_1164.all;`

Statt „all“ könnte man auch nur einzelne Komponenten des Packages auswählen (mit jeweils eigener use-Anweisung).

### Anmerkungen:

Die Bibliotheken **std** und **work** sind ohne library-Anweisung bekannt.

Man kann eigene Bibliotheken erstellen und so VHDL-Code in verschiedenen Projekten verwenden.

Die Bibliothek **work** sollte für die Design Units verwendet werden, die nur in dem aktuellen Projekt verwendet werden.

# Strukturbeschreibung: Komponenten-Instanzierung I

## Instanziieren von Komponenten

Existierende Komponenten aus **entity** und **architecture** können als Teilblock in eine übergeordnete Komponente eingebaut (instanziert) werden.

Im Prinzip könnten alle Schaltungen rein strukturell aus entsprechenden Grundelementen (Gatter, Flipflops etc.) aufgebaut werden! Dies ist jedoch sehr mühsam!

Im Anweisungsteil der **architecture** wird eine Instanz der Komponente erzeugt. Den formalen Parametern der **generic**-Parameterliste werden aktuelle Werte und den formalen Parametern der **port**-Parameterliste aktuelle Signale zugeordnet:

```
Instanz_Name : entity Library . Component_Name [(Architecture_Name)]
  [generic map (Aktuelle_Parameterliste)]
  [port map (Aktuelle_Parameterliste)]
```

*Library* ist normalerweise **work**. *Aktuelle\_Parameterliste* enthält durch Kommas getrennte *Parameter\_Zuordnungen mit Assoziationsoperator „=>“*:

*Generischer\_Parameter => Aktueller\_Parameter*

Über die Zuordnung der Port-Parameterliste wird die neue Instanz mit der übergeordneten Komponente „verdrahtet“.

Alternativ ist eine Zuordnung über die Reihenfolge der Parameter möglich.

# Strukturbeschreibung: Komponenten-Instanzierung II

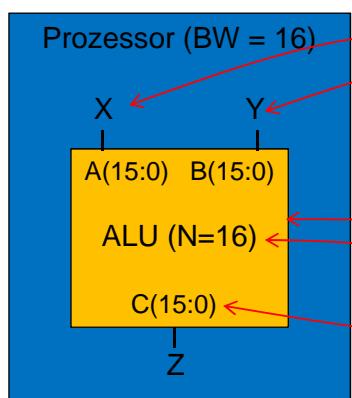
## Beispiel: ALU-Instanz in Prozessor

```
entity ALU is
  generic (N: integer := 8);
  port(
    A, B: in bit_vector(N-1 downto 0);
    C:   out bit_vector(N-1 downto 0));
end ALU;
```

Vektor bestehend aus N Bits!

```
entity Prozessor is
  generic (BW: integer := 16);
  port (...);
end Prozessor;
```

Blockschaltbild:



```
architecture arch of Prozessor is ...
begin ...
  ALUInstanz: entity work.ALU
    generic map(N => BW)
    port map(A => X, B => Y, C => Z);
end arch;
```

**Zuordnung:** Links: Port-Name des verwendeten Moduls (z. B. A aus ALU)  
Rechts: Signal-Name aus aktueller Architecture (z. B. X aus arch)



# Regeln der Schaltalgebra

1. Eliminationsgesetze („Null und Eins Gesetze“ )
  - a.  $X \vee 0 = X$  (1. Axiom neutrales Element)
  - b.  $X \vee 1 = 1$
  - c.  $X \wedge 0 = 0$
  - d.  $X \wedge 1 = X$  (2. Axiom neutrales Element)
2. Idempotenzgesetze
  - a.  $X \vee X = X$
  - b.  $X \wedge X = X$
3. Komplementgesetze („Inverse Elemente“)
  - a.  $X \vee \overline{X} = 1$  (1. Axiom inverses Element)
  - b.  $X \wedge \overline{X} = 0$  (2. Axiom inverses Element)
  - c.  $\overline{\overline{X}} = X$
4. Kommutativgesetze
  - a.  $X_1 \wedge X_2 = X_2 \wedge X_1$
  - b.  $X_1 \vee X_2 = X_2 \vee X_1$
5. Assoziativgesetze
  - a.  $(X_1 \wedge X_2) \wedge X_3 = X_1 \wedge (X_2 \wedge X_3)$
  - b.  $(X_1 \vee X_2) \vee X_3 = X_1 \vee (X_2 \vee X_3)$
6. Distributivgesetze
  - a.  $X_1 \wedge (X_2 \vee X_3) = (X_1 \wedge X_2) \vee (X_1 \wedge X_3)$
  - b.  $X_1 \vee (X_2 \wedge X_3) = (X_1 \vee X_2) \wedge (X_1 \vee X_3)$
7. Kürzungsregeln
  - a.  $X_1 \vee (X_1 \wedge X_2) = X_1$
  - b.  $X_1 \wedge (X_1 \vee X_2) = X_1$
  - c.  $X_1 \vee (\overline{X}_1 \wedge X_2) = X_1 \vee X_2$
  - d.  $X_1 \wedge (\overline{X}_1 \vee X_2) = X_1 \wedge X_2$
  - e.  $(X_1 \wedge X_2) \vee (X_1 \wedge \overline{X}_2) = X_1$
  - f.  $(X_1 \vee X_2) \wedge (X_1 \vee \overline{X}_2) = X_1$
8. De Morgansche Gesetze (gelten auch für n Variablen  $X_1 \dots X_n$ )
  - a.  $\overline{X_1 \wedge X_2} = \overline{X_1} \vee \overline{X_2}$
  - b.  $\overline{X_1 \vee X_2} = \overline{X_1} \wedge \overline{X_2}$
9. Shannonsches Gesetz
$$\overline{f(X_1, X_2, \dots, X_n, \wedge, \vee)} = f(\overline{X_1}, \overline{X_2}, \dots, \overline{X_n}, \vee, \wedge)$$



# Bereichsüberschreitungen bei Addition/Subtraktion

Bei der Addition und Subtraktion zweier ganzer Zahlen  $Z_1$  und  $Z_2$  (Typ „integer“), die mit  $n$  Bits dargestellt werden, kann es zu folgenden beiden Typen der **Bereichsüberschreitung** kommen:

- **Überlauf:** Ergebnis  $S$  ist zu **groß** für die Darstellung mit  $n$  Bits
- **Unterlauf:** Ergebnis  $S$  ist zu **klein** für die Darstellung mit  $n$  Bits  
(bzw. negatives Ergebnis bei der Subtraktion positiver Zahlen)

## 1. Addition

### Positive Zahlen

Überlauf, wenn  $c_{n-1} = 1$

(Unterlauf nicht möglich)

*Beispiel:*

$$\begin{array}{r} 1000 \quad (8) \\ +1001 \quad (9) \\ \hline 10000 \quad c_{n-1}=1, \text{ also Überlauf!} \\ 0001 \quad (1 \neq 17) \end{array}$$

### 2er-Komplement-Zahlen

Bereichsüberschreitung tritt auf, wenn beide Operanden gleiches Vorzeichenbit (MSB) haben ( $Z_{1,n-1}=Z_{2,n-1}$ ), die Summe aber ein anderes Vorzeichenbit  $S_{n-1}$  hat.

*Alternativregel:*

Bereichsüberschreitung genau dann, wenn  $c_{n-1} \neq c_{n-2}$ . Dann gilt:

- $c_{n-1}=0$ : Überlauf
- $c_{n-1}=1$ : Unterlauf

*Beachte:* Bei 2er-Komplement-Addition sagt  $c_{n-1}$  alleine nichts aus!

*Beispiele ohne Bereichsüberschreitung:*

$\begin{array}{r} 0001 \quad (1) \\ +1010 \quad (-6) \\ \hline 0000 \end{array}$	$\begin{array}{r} 0111 \quad (7) \\ +1010 \quad (-6) \\ \hline 1100 \end{array}$	$\begin{array}{r} 1100 \quad (-4) \\ +1110 \quad (-2) \\ \hline 1100 \end{array}$
OK, da $Z_{1,n-1} \neq Z_{2,n-1}$ bzw. $c_{n-1} = c_{n-2}$	OK, da $Z_{1,n-1} \neq Z_{2,n-1}$ bzw. $c_{n-1} = c_{n-2}$ ( $c_{n-1} = 1$ irrelevant!)	OK, da zwar $Z_{1,n-1} = Z_{2,n-1} = 1$ , aber auch $S_{n-1} = 1$ bzw. $c_{n-1} = c_{n-2}$ ( $c_{n-1} = 1$ irrelevant!)

Beispiele mit Bereichsüberschreitung:

$$\begin{array}{rcl}
 0110 \text{ (6)} & & 1000 \text{ (-8)} \\
 +0011 \text{ (3)} & & +1000 \text{ (-8)} \\
 \underline{01100} & & \underline{10000} \\
 1001 \text{ (-7≠9)} & & 0000 \text{ (0≠-16)}
 \end{array}$$

$Z1_{n-1}=Z2_{n-1}=0$  aber  $S_{n-1}=1$   
bzw.  $c_{n-1} \neq c_{n-2}$  und  $c_{n-1}=0$   
also: Überlauf!

$Z1_{n-1}=Z2_{n-1}=1$  aber  $S_{n-1}=0$   
bzw.  $c_{n-1} \neq c_{n-2}$  und  $c_{n-1}=1$   
also: Unterlauf!

## 2. Subtraktion

Beachte: Die Subtraktion wird wie im kombinierten Addierer/Subtrahierer mit SEL=1 durch Invertieren und Addieren unter Verwendung von SEL=1 als Übertrag für Bit 0 (carry-in) berechnet.  $c_{n-1}$  und  $c_{n-2}$  beziehen sich immer auf die Überträge in diesem Operator!

### Positive Zahlen

Unterlauf, wenn  $c_{n-1} = 0$   
(Überlauf nicht möglich)

Beispiel: nach Invertieren:

$$\begin{array}{rcl}
 0010 \text{ (2)} & & 0010 \\
 -0100 \text{ (4)} & \rightarrow & +1011 \\
 & & \underline{00111} \text{ (carry-in = 1!)} \quad c_{n-1}=0, \text{ also Unterlauf!} \\
 & & 1110 \text{ (14≠-2)}
 \end{array}$$

### 2er-Komplement-Zahlen

Regel für Bereichsüberschreitung: Verwende Alternativregel der Addition

Beispiel ohne Bereichsüberschreitung:

$$\begin{array}{rcl}
 1001 \text{ (-7)} & & 1001 \\
 -1000 \text{ (-8)} & \rightarrow & +0111 \\
 & & \underline{11111} \text{ (carry-in = 1!)} \quad \text{OK, da } c_{n-1} = c_{n-2} \\
 & & 0001 \text{ (1)}
 \end{array}$$

Beispiele mit Bereichsüberschreitung:

$$\begin{array}{rcl}
 0111 \text{ (7)} & & 0111 \\
 -1010 \text{ (-6)} & \rightarrow & +0101 \\
 & & \underline{01111} \text{ (carry-in = 1!)} \quad c_{n-1} \neq c_{n-2} \text{ und } c_{n-1}=0: \text{Überlauf (zu groÙe positive Zahl)} \\
 & & 1101 \text{ (-3≠+13)} \\
 \\ 
 1011 \text{ (-5)} & & 1011 \\
 -0110 \text{ (6)} & \rightarrow & +1001 \\
 & & \underline{10111} \text{ (carry-in = 1!)} \quad c_{n-1} \neq c_{n-2} \text{ und } c_{n-1}=1: \text{Unterlauf (zu kleine negative Zahl)} \\
 & & 0101 \text{ (5≠-11)}
 \end{array}$$