

**Schichtenmodell eines Computers:**

	Anwendungsoftware / Programme
	Betriebssysteme : Gerätetreiber, ...
	Architektur : Befehle, Register, ...
	Mikroarchitektur : Datenpfade, Steuerung, ...
DT {	Logik : Addierer, Speicher, ...
	Digitalschaltungen : UND Gatter, Invertierer, ...
	Anologschaltungen : Verstärker, Filter, ...
	Bauteile : Transistoren, Dioden, ...
	Physik : Elektronen, ...

Schnittstellen zwischen den Schichten

Vorteile: 1 Schicht kennen nicht z.B. bei den meisten Fällen, Austauschbarkeit von Schichten

Nachteile: ggf schlechtere Leistungsfähigkeit

**Die 3 Y's:**

• Hierarchie :

Aufteilen eines Systems in Module und Untermodule: Gatter, Multiplexer, Decoder, Register, Speicher, ...

• Modularität :

Wohldefinierte Schnittstellen und Funktionen

• Regularität :

Bewerze einheitliche Lösungen → Wiederverw.

z.B.: austauschbare Teile

Strukturen leicht auf versch. Größen anpassbar

Notiz: Bit = Binary + Digit

Größenfaktoren in der Informatik

1 Byte = 8 Bit ( $\rightarrow 2^8 = 256$  Kodierungen)

1 Nibble = 4 Bit

1 Wort: unterschiedlich / uneinheitlich: 16/32/64 Bit

$$k = 1024 = \underline{2^{10}} \mid \text{kilo/kibi}$$

$$M = 1024 \times 1024 = \underline{2^{20}} \mid \text{mega/mibi}$$

$$G = 1024 \times 1024 \times 1024 = \underline{2^{30}} \mid \text{giga/gibi}$$

$$T = 1024 \times 1024 \times 1024 \times 1024 = \underline{2^{40}} \mid \text{tera/tebi}$$

kilo:  $10^3$

mega:  $10^6$

giga:  $10^9$

tera:  $10^{12}$

Verteilung von natürl. Zahlen

Beispiele:

$$\begin{aligned} \text{Decimal: } 5347_{10} &= 7 \cdot 10^0 + 4 \cdot 10^1 + 3 \cdot 10^2 + 5 \cdot 10^3 \\ &= 7 + 40 + 300 + 5000 = 5347_{10} \end{aligned}$$

$$\begin{aligned} \text{Binär: } 1101_2 &= 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 \\ &= 1 + 0 + 4 + 8 = 13_{10} \end{aligned}$$

$$\begin{aligned} \text{Hex: } 1F3A_{16} &= 10 \cdot 16^0 + 3 \cdot 16^1 + 15 \cdot 16^2 + 1 \cdot 16^3 \\ &= 10 + 48 + 3840 + 4096 = 7994_{10} \end{aligned}$$

Allgemein / Abstrakt:

Basis  $b \in \mathbb{N} \wedge b \geq 2 \rightarrow \mathbb{Z}_b := \{0, 1, \dots, b-1\}$ : Menge Ziffern

$u_{b,k}$  bildet Ziffernfolge der Breite  $k \in \mathbb{N}$  auf nat. Zahl:

$$u_{b,k} (a_{k-1} \dots a_1 a_0) \in \mathbb{Z}_b^k \mapsto \sum_{i=0}^{k-1} a_i \cdot b^i \in \mathbb{N}$$

Kleinste Zahl: 0

$\Rightarrow$  Größte Zahl:  $b^k - 1$   
Anzahl Werte:  $b^k$

Hex:  
0x03

Notiz: Dual Octal:  
0b1101 0o323 / 0323

msb / lsb

Two small black marks on a grid background, resembling stylized arrows or checkmarks.

z. B.: 1011011010  
↓  
1011011010  
1

The diagram illustrates two distinct writing styles on lined paper. The upper line shows a single, continuous, sweeping stroke from left to right, labeled "MS Style". The lower line shows a shorter, more compact stroke, labeled "Nibble".

## Darstellung von ganzen Zahlen

(„Betrag & Vorzeichen“)

Basis  $b \in W$   $\wedge$   $b \geq z$ :

$Z_b := \{0, 1, \dots, b-1\}$ : Menge Ziffern

Abbildung einer kelln-breiten Ziffernfolge auf  
ganze Zahl durch Funktion  $b_{k,k}$ :

$$bv_{b,k} : (a_{k-1} \dots a_1 a_0) \in \{0,1\}^k \times \mathbb{Z}_b^{k-1}$$

$$\mapsto (-1)^{a_{k-1}} \cdot \sum_{i=0}^{k-2} a_i \cdot b^i \in \mathbb{Z}$$

$$\Rightarrow \text{ kleinste Zahl: } (-1)^1 \cdot \sum_{i=0}^{k-2} (b-1) \cdot b^i = -(b^{k-1} - 1)$$

$$\text{größte Zahl} \cdot (-1)^0 \cdot \sum_{i=0}^{k-2} (b-1) \cdot b^i = + (b^{k-1} - 1)$$

Achtung: Nicht eindeutig; doppelte Darstellung der 0

$$\text{z.B.: } \det_{2,4} (1110_2) = (0 \cdot 2^0 + \dots + 1 \cdot 2^3) \cdot (-1)^1 = -6_{10}$$

$$bv_{z_M}(0110_2) = (0 \cdot 2^0 + \dots + 1 \cdot 2^2) \cdot (-1)^0 = +6_{10}$$

→ inkompatibel mit Bineraddition!

Darstellung von  
ganzen Zahlen

- Zweierkomplement

„Goldstandard“

Abbildung kelli-breiter Bitfolge auf ganze Zahl:

$$s_k : (a_{k-1} \dots a_1 a_0) \in \{0,1\}^k$$

$$\mapsto a_{k-1} \cdot (-2^{k-1}) + \sum_{i=0}^{k-2} a_i \cdot 2^i \in \mathbb{Z}$$

$$\Rightarrow \text{ kleinste Zahl: } 1 \cdot (-2^{k-1}) + \sum_{i=0}^{k-2} 0 \cdot 2^i = -2^{k-1}$$

$$\text{größte Zahl: } 0 \cdot (-2^{k-1}) + \sum_{i=0}^{k-2} 1 \cdot 2^i = 2^{k-1} - 1$$

Anzahl darstellbare Werte:  $2^k$

Eindeutig / bijektiv auf Wertebereich  $\{-2^{k-1}, \dots, 2^{k-1} - 1\}$   
(für gegebenes  $k$ )

$$\text{Bsp: } s_4(1010_2) = 0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot -2^3 = -6_{10}$$

$$s_4(0110_2) = 0 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot -2^3 = +6_{10}$$

→ kompatibel mit binärer Addition!

Decimal  $\rightarrow$  Zweierkomplement:

1. Größtmögliche Zerpotenz abziehen:

$$\begin{aligned}-63_{10} &= -64 + 11 \\&= -64 + 8 + 3 \\&= -64 + 8 + 2 + 1 \\&= -2^6 + 2^3 + 2^1 + 2^0 \\&= 1001011_2\end{aligned}$$

2.) Betrag invertieren:

Bits lippeln, 1 addieren  
(immer; auch für anderes rum!)

### Bitbreitenerweiterung:

notwendig um unterschiedlich breite Bitfolgen zu addieren

zero extension:

führende Nullen (unzeichenlose Darstellung)

signed extension:

Wert des Vorzeichen-Bits (Zweiernkomplement Par..)

### Schätzungen:

$$2^{13} \approx 8 \text{ Tausend}$$

$$2^{45} \approx 32 \text{ Billionen}$$

$$2^{28} \approx 256 \text{ Millionen}$$

$$2^{34} \approx 16 \text{ Milliarden}$$

### Logikgatter:

NOT:  $A \rightarrowtail y \quad Y = \bar{A}; Y = !A; Y = \neg A$

BUF:  $A \rightarrowtail y \quad Y = A$

AND:  $\begin{array}{c} A \\[-1ex] \diagup \\[-1ex] B \end{array} \rightarrowtail y \quad Y = AB; Y = A \& B; Y = A \cap B$

OR:  $\begin{array}{c} A \\[-1ex] \diagdown \\[-1ex] B \end{array} \rightarrowtail y \quad Y = A + B; Y = A \mid B; Y = A \cup B$

XOR:  $\begin{array}{c} A \\[-1ex] \diagup \\[-1ex] B \end{array} \rightarrowtail y \quad Y = A \oplus B; Y = A \wedge B \wedge \neg$

NAND:  $\begin{array}{c} A \\[-1ex] \diagup \\[-1ex] B \end{array} \rightarrowtail y \quad Y = \overline{AB}$

NOR:  $\begin{array}{c} A \\[-1ex] \diagdown \\[-1ex] B \end{array} \rightarrowtail y \quad Y = \overline{A+B}$

XNOR:  $\begin{array}{c} A \\[-1ex] \diagup \\[-1ex] B \end{array} \rightarrowtail y \quad Y = \overline{A \oplus B}$

allgemein:  
 $Y=1$  wenn ungerade  
Anzahl 1 ist z.B.  
1 oder 3 Eingänge

**Binärwerte durch Spannungen:**

$$0V = GND = 0$$

80er Jahre

kleinere Transistoren  
→ weniger Spannung

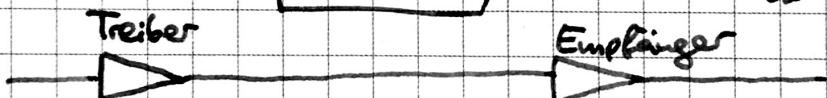
$$V_{DD} = 1 \text{ z.B.: } 5V, 3,3V, 2,5V, 1,8V, \dots$$

Rauschen: Störung der Nutzsignale z.B. unerwünschte Widerstände



Bereiche statt feste Werte

**Logikpegel:**

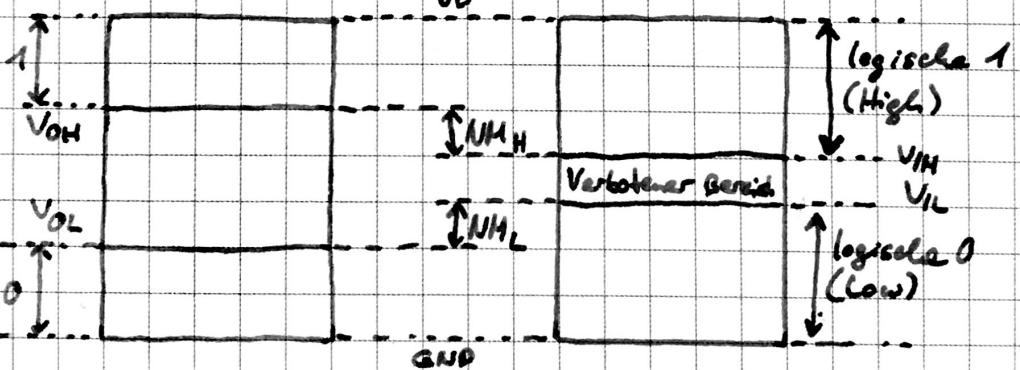


Ausgangsschar.

$V_{DD}$

Eingangsschar.

$V_{OH} = \text{Voltage out High}$   
 $V_{OL} = \text{"Low"}$



Hoch Störabstand:  $NM_H = V_{OH} - V_{IH}$

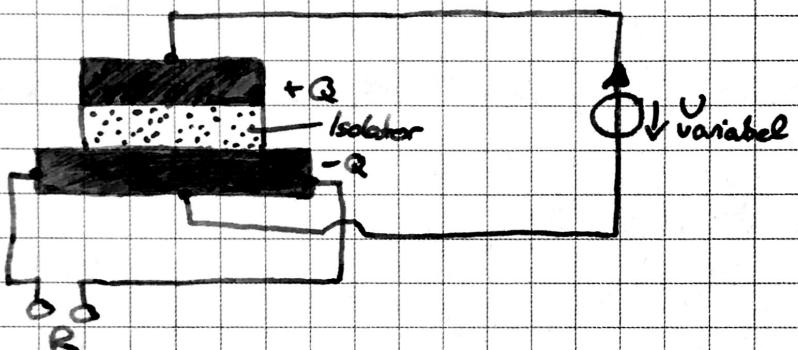
Niedriger Störabstand:  $NM_L = V_{IL} - V_{OL}$

$V_{IH} = \text{Voltage In High}$   
 $V_{IL} = \text{"Low"}$

**Transistoren:**

FET = Feldeffekttransistor (auch andere z.B. Bipolartr.)

Feldeffekt: Spannungsgesteuerter Widerstand:



Gleichspannung  $U$  am Metallplatten → Ausammung

gegenseitlicher Ladungsträger (Ladung:  $Q = C \cdot U$ )

Hier: wenig Elektronen oben, viele unten

→ mehr freie Ladungsträger unten

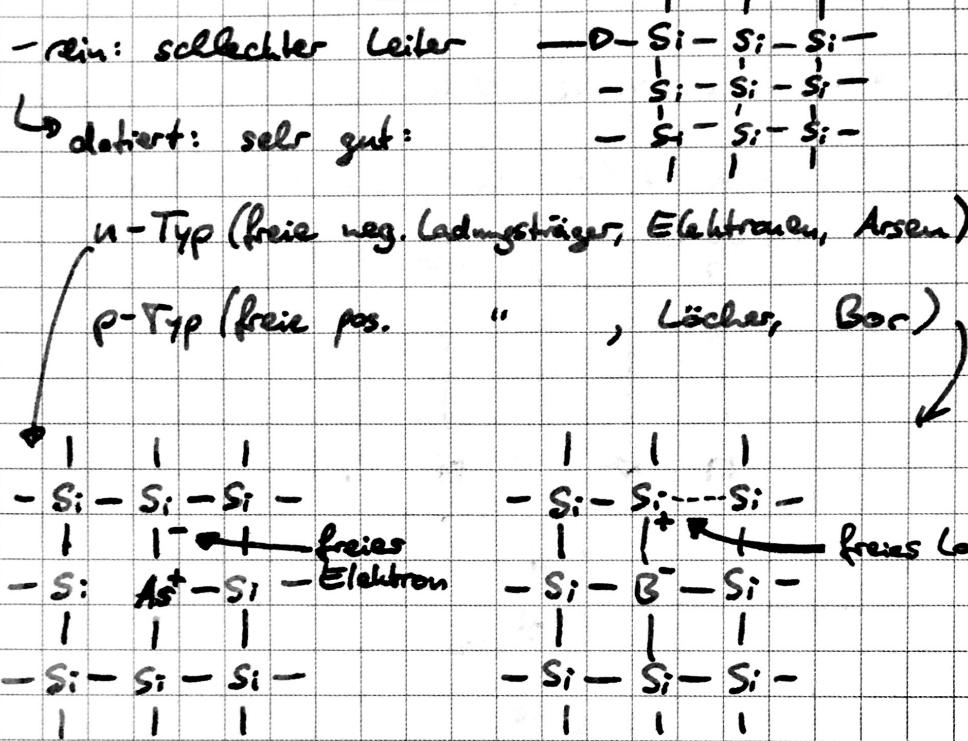
→ Widerstand unten durch  $Q$  / durch  $U$  regelbar

In Metallen: Ansammlung von Elektronen unerlässlich,

da „so“ schon sehr viele → hohen Unterschied

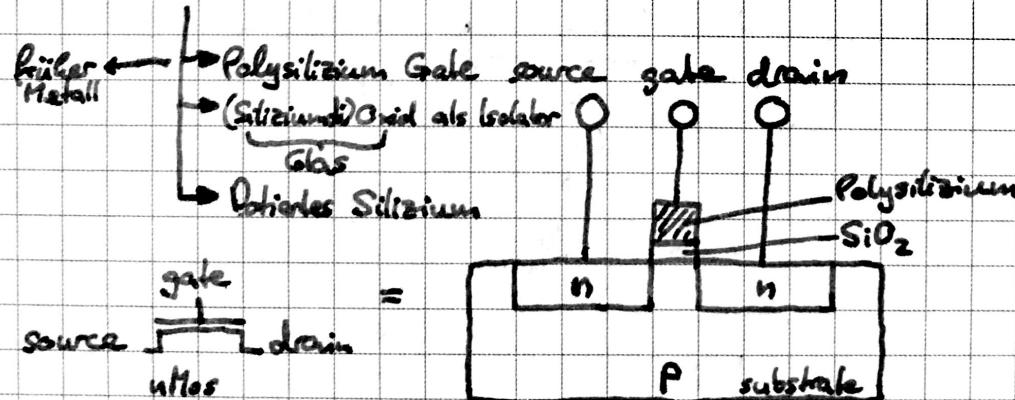
(ca.  $10^{22}$  Ladungsträger pro  $\text{cm}^3$ ). Halbleiter:  $10^{13} \cdot 10^{14} \text{ cm}^{-3}$

### Silizium:



### MosFET:

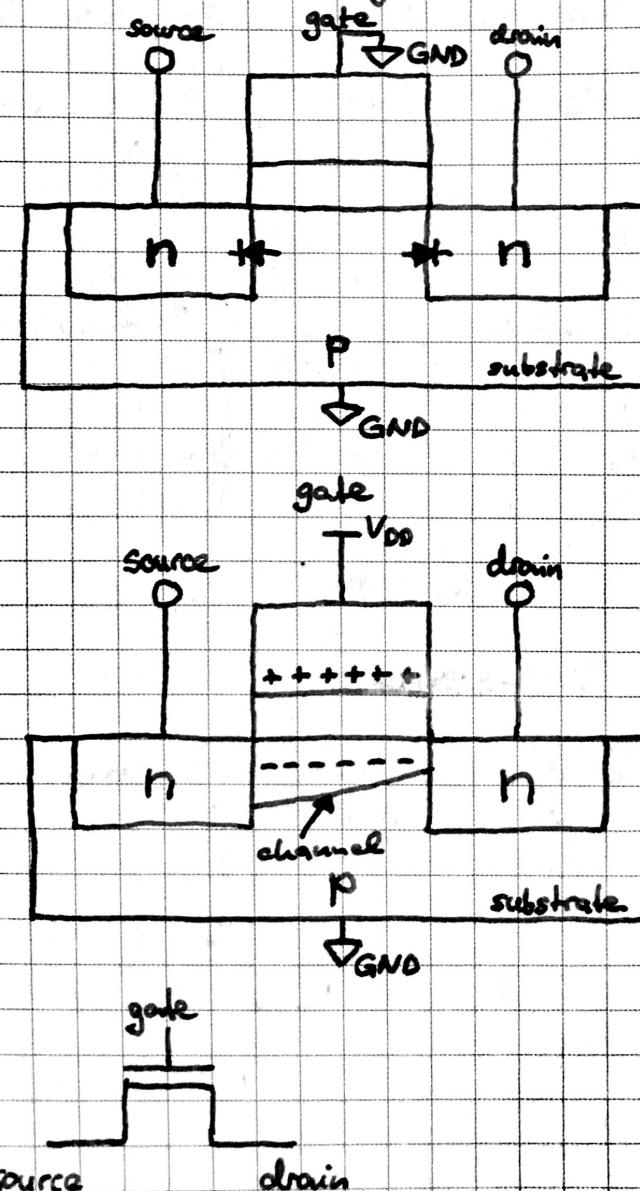
MOS = Metalloxid-Silizium



### nMOS:

Gate = 0  $\rightarrow$  aus  $\rightarrow$  keine Verbindung source / drain

Gate = 1  $\rightarrow$  ein  $\rightarrow$  leitfähiger Kanal source / drain

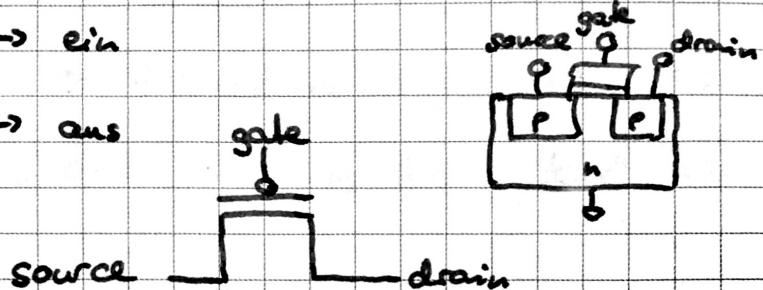


pMOS:

genau umgekehrt

Gate = 0 → ein

Gate = 1 → aus



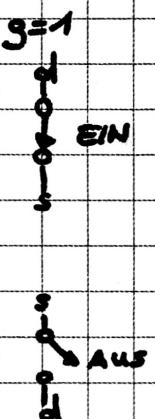
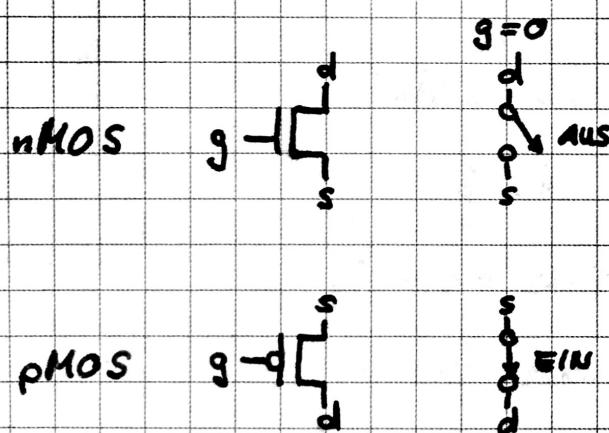
CMOS:

nMOS: leitet 0 gut zw. S und D → S an GND

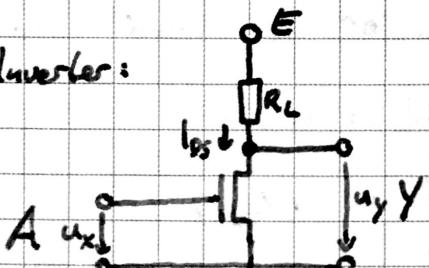
pMOS: leitet 1 gut zw. S und D → S an V<sub>DD</sub>

Complementary Metal-Oxide-Semiconductor

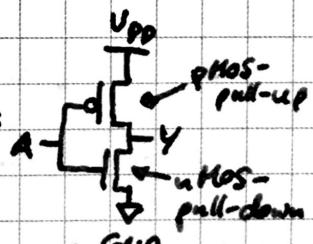
Funktion von Transistoren:



z.B. Inverter:



alternativ:



Besser: nMOS: 0V, 1V  
pMOS: 1V, 0V

⇒ Leistungsaufnahme:  $P = I \cdot V = (C \cdot V_{DD} \cdot f) \cdot V_{DD} = C \cdot V_{DD}^2 \cdot f$

Statische Leistungsaufnahme:

- Schaltung selbst

- Leckstrom  $I_L$

$$P_{\text{static}} = I_L \cdot V_{DD}$$

Dynamische Leistungsaufn.:

Gates umladen: kostet Leistung → wie Kondensator

↳ Ladung auf Kondensator der Kapazität C:  $Q = C \cdot V_{DD}$

Transistor: schaltet f-Kal. / selb.

$$I = Q/t = Q \cdot f = C \cdot V_{DD} \cdot f$$

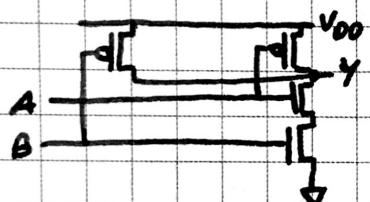
=> Dynamische Leistungsanforderung:

$$\sim P = \frac{1}{2} \cdot C \cdot V_{DD}^2 \cdot f$$

(Annahme: Hälfte der Zeit wird von  $1 \rightarrow 0$  entladen;

Entladen kostet nichts)

NAND als CMOS:



REGEL:

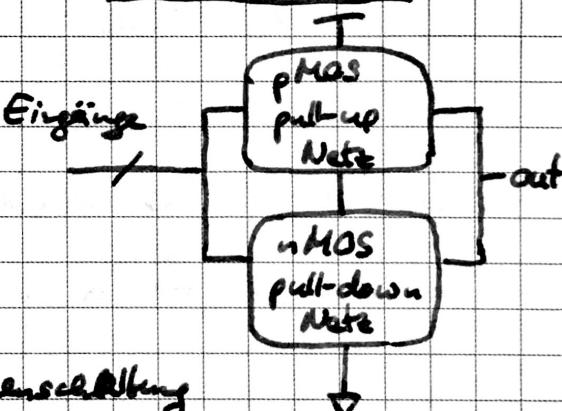
p-Transistoren in Reihenschaltung

→ n-Transistoren in Parallelschaltung.

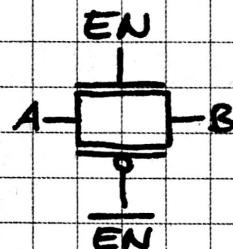
n-Transistoren in Reihenschaltung

→ p-Transistoren in Parallelschaltung

CMOS-Struktur:



Transmission Gates:



=> Leitet 0 + 1 gut

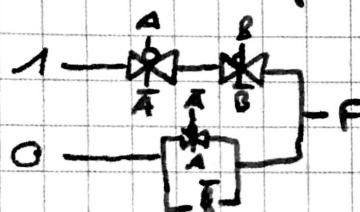
=> EN = 1 => A verbunden mit B

=> EN = 0 => A nicht verbunden mit B

Moore's Gesetz: (1965)

„Alle 18 Monate verdoppelt sich die Anzahl an Transistoren auf einem Chip“

TM-Gate Beispiel:



## Kombinatorische Schaltung:

- Verbindungsknoten / Nodus, z.B.: Ein-/Ausgangsterminals, interne Knoten (siehe später: SystemVerilog)
- Schaltungselemente: Je wiederum Schaltung
- Ausgänge hängen nur von aktuellen Eingängen ab  
→ auch genannt: Schaltkreis
- Regeln:
  - Jedes Schaltungselement ist knub.

- Knoten:
    - a) Eingänge in Schaltung
    - b) an Ausgang eines Elements
  - Keine Zyklen → maximal 1x jeden Knoten besuchen in jedem Pfad

## Sequentialle Logik:

- Ausgänge hängen von aktuellen Eingängen und von einem gespeicherten (→ Speicher) Zustand ab  
↳ also auch von vorherigen Eingangswerten
- Deshalb auch genannt: Schaltwerk

Definitionen für  
Boolesche Gleichungen:

Disjunktive Normalform  
DNF:

- Komplement: Variable mit Balken (invariert)
- Literal: Variable oder ihr Komplement
- Implikant: Produkt v. Literalen, z.B.:  $A\bar{C}$
- Minterm: Produkt (UND) über alle Eingangsvariablen
- Maxterm: Summe (ODER) über alle Eingangsvariablen

- Schen: SOP (sum of products)
- DNF: Disjunktion (ODER) der Minterme, die am Ausgang Wahr liefern.
  - Jede Zeile der Wahrheitstabelle enthält 1 Minterm
  - Alle Booleschen Funktionen können im DNF formuliert werden.
  - Minterm: Konjunktion der Literale

Konjunktive Normalform:

- Schemata: POS (product of sums)

- CNF: Konjunktion (UND) der Maxterme, die am

Ausgang FALSCH liefern.

→ Jede Zeile der Wahrheitstabelle enthält einen Maxterm.

→ Alle Booleschen Funktionen können im CNF formuliert werden.

→ Maxterm: Disjunktion der Literale

↓

"falsch rum":  $0 \rightarrow A$   
 $1 \rightarrow \bar{A}$

Axiome oder Booleschen Algebra:

A1:  $B=0 \neq B+1$

$B=1$  if  $B \neq 0$

Dualitätsgesetz

A2:  $\bar{0} = 1$

$\bar{1} = 0$

NOT

A3:  $0 \cdot 0 = 0$

$1 + 1 = 1$

AND/OR

A4:  $1 \cdot 1 = 1$

$0 + 0 = 0$

AND/OR

A5:  $0 \cdot 1 = 1 \cdot 0 = 0$

$1 + 0 = 0 + 1 = 1$

AND/OR

Sätze oder Booleschen Algebra:

T1:  $B \cdot 1 = B$

$B+0 = B$

Neutralitätsgesetze

T2:  $B \cdot 0 = 0$

$B+1 = 1$

Extremalgesetze

T3:  $B \cdot B = B$

$B+B = B$

Idempotenzgesetze

T4:  $\overline{\overline{B}} = B$

Inversion

T5:  $B \cdot \overline{B} = 0$

$B+\overline{B} = 1$

Komplementärgesetze

Sätze der  
Booleschen Algebren

mit mehreren Variablen

$$T6: B \cdot C = C \cdot B$$

$$T7: (B \cdot C) \cdot D = B \cdot (C \cdot D)$$

$$T8: B \cdot (C+D) = (B \cdot C) + (B \cdot D)$$

$$T9: B \cdot (B+C) = B$$

$$T10: (B \cdot C) + (B \cdot \bar{C}) = B$$

$$T11: (B \cdot C) + (\bar{B} \cdot D) + (C \cdot D)$$

$$= (B \cdot C) + (\bar{B} \cdot D)$$

$$T12: \overline{B_0 \cdot B_1 \cdot B_2 \dots} = \overline{B_0} + \overline{B_1} + \overline{B_2} \dots$$

Dualität:  $\cdot \leftrightarrow + ; \quad 0 \leftrightarrow 1$

$$T6: B + C = C + B$$

$$T10: (B+C) \cdot (B+\bar{C}) = B$$

$$T7: (B+C)+D = B+(C+D)$$

$$T11: (B+C) \cdot (\bar{B}+D) \cdot (C+D)$$

$$T8: B + (C \cdot D) = (B+C) \cdot (B+D)$$

$$= (B+C) \cdot (\bar{B}+D)$$

$$T9: B + (B \cdot C) = B$$

$$T12: \overline{B_0 + B_1 + B_2 + \dots}$$

$$= \overline{B_0} \cdot \overline{B_1} \cdot \overline{B_2} \cdots$$

Kommutativgesetz

Assoziativgesetz

Distributivgesetz

Absorptionsgesetz

Zusammenfassen

Konsensregeln

De Morgansche Gesetze

- rückwärts schreiben: AND  $\rightarrow$  OR

$$\begin{array}{l} A \rightarrow D \rightarrow Y \\ B \rightarrow D \end{array} = \begin{array}{l} A \rightarrow D \\ B \rightarrow D \end{array} \rightarrow Y$$

- vorwärts schreiben: AND  $\leftarrow$  OR

$$\begin{array}{l} A \rightarrow D \\ B \rightarrow D \end{array} \rightarrow Y = \begin{array}{l} A \rightarrow D \\ B \rightarrow D \end{array} \rightarrow Y$$

(nur wenn an allen Eingängen Blase ist)

- Regeln: - von Ausgang rückwärts Eingänge arbeiten

- Gattersorte tauschen

- Blasen auslösen (2 auf 1 Ladung)

- Nützlich um Schaltungen mit AND/NOR gates leichter zu verstehen!

**Regeln für Schaltpläne:**

- T-Kreuzung: verbunden
- | —
- Überkreuzung mit Punkt: verbunden
- + —
- Überkreuzung ohne Punkt: nicht verbunden
- | —

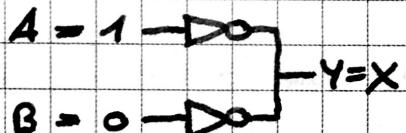
**PLA:**

- Programmable Logic Array
- Boole'sche Gleichung in SOP-Form kann so systematisch als Schematic gezeichnet werden:
  - 1) vertikale Striche als Eingänge
  - 2) Inverter dazwischen (falls nötig) für invertierte Eingänge / komplementäre Eingänge
  - 3) Zeilenweise AND-Gates für jeden Minterm
  - 4) Für jeden Ausgang: OR-Gate, verbunden mit den für diesen Ausgang relevanten Minterminen
- 2-Level-Technik
- Merkmal: Kaskadierung mehrerer einfacherer Funktionen → weniger Gatter!

**X: Treiberkonflikt**

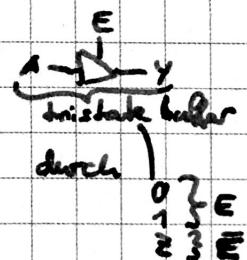
- Unterschied zu don't-care nur durch Kontext erkennbar
- Konflikt: Schaltung treibt 0 und 1 an einem Ausgang  $\rightarrow$  Wert: irgendwas...
- $\rightarrow$  Kurzschluss = hoher Energieverbrauch
- $\rightarrow$  fast immer Entwurfssfehler

Bsp.:



**Z: Hochlunger-Ausgang**

- Name: offen, ungetrieben, floating, high impedance
- 0 oder 1 oder irgendwas dazwischen (nicht!!! 0)
- Grund: kein aktiver Treiber
- Wert hängt von vorigen Ereignissen ab
- Nicht unbedingt ein Fehler:  
Solange der betroffene Knoten vor „Benutzung“ durch die Schaltung bzw. bevor er für eine Operation relevant ist, auf ein gültiges Logilevel gebracht wird.
- Vorwendungszweck: Tristate - Busse:  
Bus, der versch. Chips verbindet: alle durch Tristate-Buffer verbunden:  
1 Chip: EN aktiv  
Rest: EN inaktiv  $\rightarrow$  Z  $\rightarrow$  keine Konflikte



## Karnaugh-Diagramme:

- Minimierung, bis zu 4 Variablen
- Idee:  $P \cdot A + P \cdot \bar{A} = P$
- System: 1en in benachbarten Plätzen markieren, zu Vierseiten zusammenfassen  
 ↳ Jeder Bereich = 1 Minterm
- Literale, die im Bereich als Komplement sowie normal auftreten, fallen weg im Produkt
- Primimplikant: Implikant (Produkt) der größten zusammenhängenden Fläche im Karnaugh-Diagramm
- Minimierungsregeln:  
 - Jede 1 mindestens 1x markieren, X (don't care) möglich  
 - Je vierch. Ber.: Seitenlängen Zweierpotenzen  
 - Je Bereich: so groß wie möglich (Primimplikant)  
 - Bereich darf um Ränder kommen  
 → Ziel: möglichst wenige Primimplikanten, alle 1 abdecken.

## Espresso-Algorithmus zur Logikminimierung

Datenstruktur:

$$\begin{aligned} a &= 01 \\ \bar{a} &= 10 \\ x &= 11 \end{aligned}$$

z.B.:  $a b c \rightarrow 01 \ 01 \ 01$

→ Im Rechner als Matrix verarbeitet

### Operatoren:

#### EXPAND:

- 1.: Ordne alle Implikanten
- 2.: Expandiere jeden Implikanten
- 2.1: Überdecke andere Implikanten
- 2.2: Expandiere weiter
- 2.2.1: Schneide andere Implikanten
- 2.2.2: Expandiere maximal

#### IRREDUNDANT:

Entfernen von Redundanzen

- 1.: Bestimme alle redundanten Implikanten
- 2.: Löse alle total redundanten Implikanten
- 3.: Übernehme einen partiell redundanten Implikanten.

#### REDUCE:

- 1.: Ermittle nur von diesen Implikanten gedeckte Punkte (essentiell)
- 2.: Gib die kleinstmög. Schaltf. ab, die alle ess. P. umfasst.

#### Ablauf:

#### EXPAND

#### IRREDUNDANT

#### REDUCE

#### EXPAND

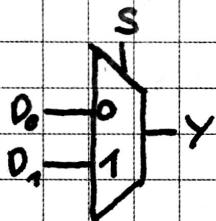
#### IRREDUNDANT

↓  
ja (Verbesserung?)  
nein

## Multiplexer (mehr)

Wählt einen von  $N$  Ausgängen:

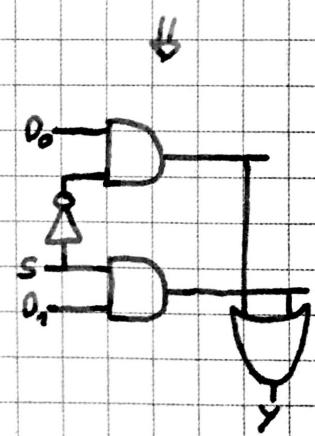
$\log_2 N$ -bit Selektor / Steuereingang (select input)



Implementierung:

Logikgitter:

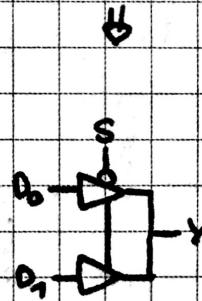
	00	01	11	10	
00	0	0	1	1	sofort/sofort
01	0	1	1	0	sofort/sofort
11	1	1	0	0	
10	1	0	0	1	



Tristate-Buffer:

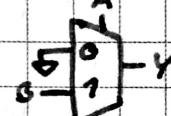
$N$  Eingänge  $\rightarrow N$  Tristates

1 Buffer an, Rest:  $Z$

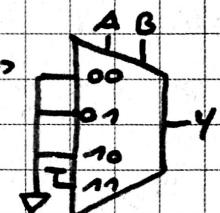


Logikfunktionen aus Multiplexern:

- Look-up-Table:  $Y = AB \rightarrow$

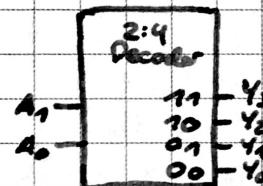


← reduziert:

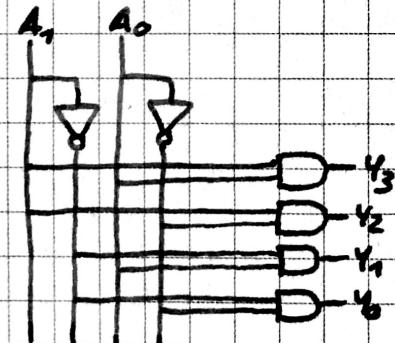


- Dekodierer / Decoder:

$N$  inputs,  $2^N$  outputs, Ausgänge: „one-hot“



Implementierung:



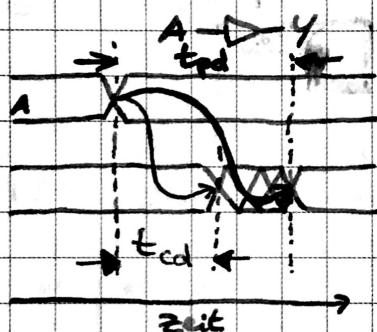
**Zeitverhalten /  
Timing:**

- Ausbreitungsverzögerung = propagation delay =  $t_{pd}$

↳ max. Zeit vom Eingang zum Ausgang

- Kontaminationsverzögerung = contamination delay =  $t_{cd}$

↳ min. Zeit vom Eingang zum Ausgang



Unterschiede durch:

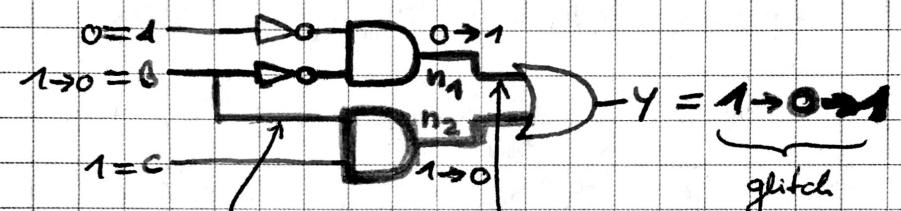
- versch. Verzögerungen (posedge, negedge)
- mehrere Ein-/Ausgänge mit versch. Verz.
- langsamer bei Erwärmung
- schneller bei Abkühlung

Kritischer Pfad =  $t_{pd}$ 's des längsten Pfades addieren

Kurzer Pfad:  $t_{cd}$ 's addieren (des kürzesten Pfades)

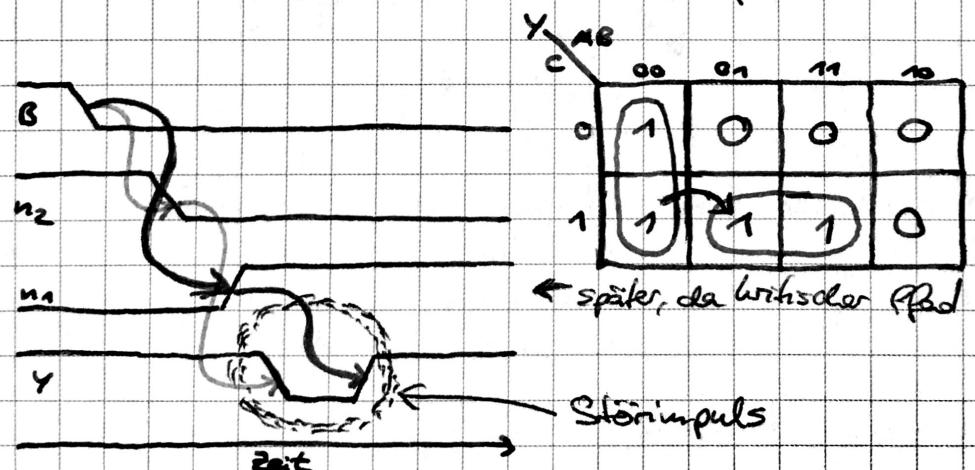
**Störimpulse /  
Glitches:**

- 1 Eingangsänderung → mehrere Ausgangsänderungen
- Durch geeignete Entwurfsspezifikation entschärfen
- Beispiel:  $Y = \bar{A}\bar{B} + BC$ ;  $A=0, B=1, C=1$ , B: fällt  $\rightarrow 0$

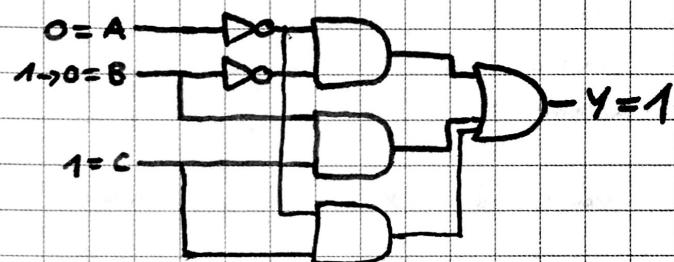


kurzer Pfad  
→ ändert zuerst

kritischer Pfad



## Störimpulse beseitigen:



→ In der Regel: verursachen Glitches keine Probleme

bei synchronem Entwurf

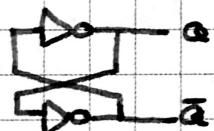
→ Sollten beim Debugging erkannt werden (Oscilloskop/Sim.)

→ Nicht alle können beseitigt werden! (z.B. Schalten mehrer Eingänge)

## Sequentielle Logik:

- Ausgänge hängen von aktuellen + vorigen Inputs ab
- Interne Zustände werden gespeichert (→ Automat)
- Sychrone sequentielle Schaltung = kombinatorische Logik gefolgt von Flip-Flops
- Durch (Kurzzeit-) „Gedächtnis“ Folgen von Ereignissen bearbeiten.
- Rückkopplungen, um Informationen zu speichern

## Bistabile Grundschaltung:

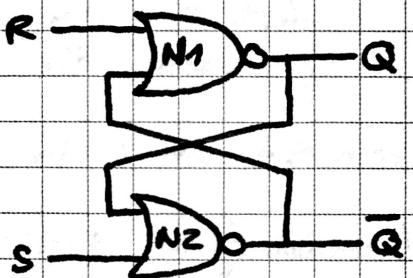


→ Speichert 1 Zustandsbit in Q/Q̄

↳ Ganz klein anderer Speicherelemente

**SR-Latch:**

Aufbau:



**Wertetabelle:**

S	R	Q	$\bar{Q}$
0	0	$Q_{\text{prev}}$	$\bar{Q}_{\text{prev}}$
0	1	0	1
1	0	1	0
1	1	0	0

→ gespeichert!

→ Set:

→ ungültiger Zustand / illegal

Schaltplansymbol:



→ 2 Eingänge:

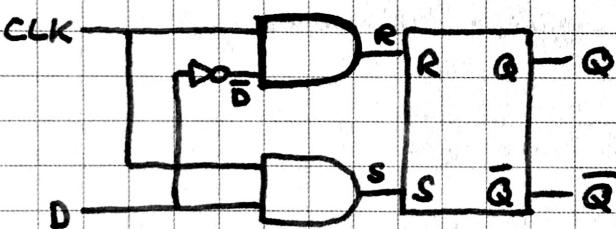
CLK = wann Änderung  
D = welche Änderung / Daten

CLK = 1  $\Rightarrow$  transparent  $\Rightarrow$  D an Q

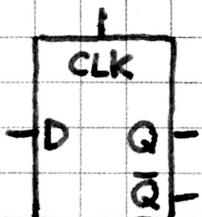
CLK = 0  $\Rightarrow$  nicht durchsichtig  $\Rightarrow$  Q bleibt gleich

→  $Q \neq \bar{Q}$  passiert nicht mehr.

Aufbau:



Schaltplansymbol:



Wertetabelle:

CLK	D	$\bar{D}$	S	R	Q	$\bar{Q}$
0	X	$\bar{X}$	0	0	$Q_{\text{prev}}$	$\bar{Q}_{\text{prev}}$
1	0	1	0	1	0	1
1	1	0	1	0	1	0

## D Flip-Flop:

→ 2 Eingänge: CLK, D

CLK: 0 → 1 ⇒ 0 an Q

sonst: Q bleibt gleich

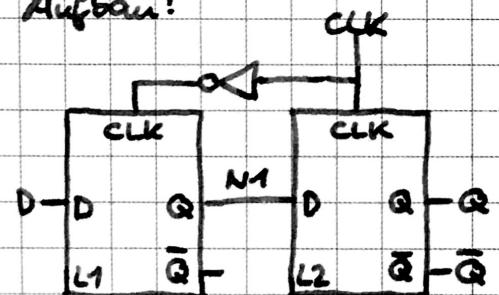
} flankengesteuert

| CLK = 0 ⇒ L<sub>1</sub> transparent, L<sub>2</sub> nicht ⇒ 0 an N1

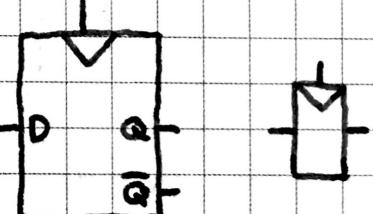
CLK = 1 ⇒ L<sub>2</sub> transparent, L<sub>1</sub> nicht ⇒ N1 an Q

→ steigende Flanke = 0 → 1 ⇒ D an Q

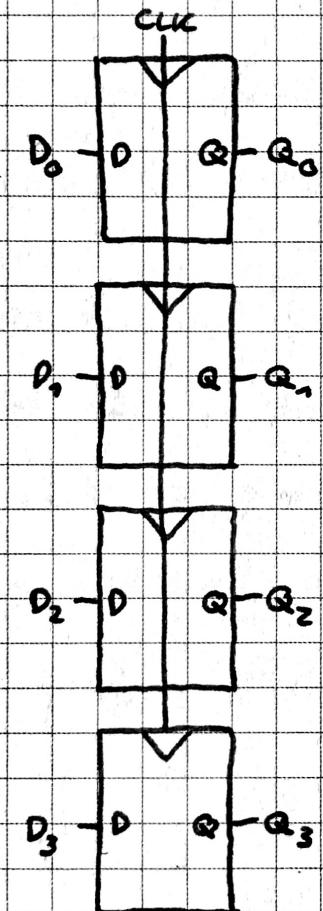
Aufbau:



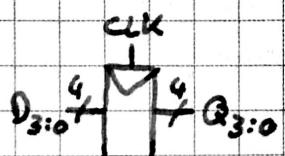
Schaltplansymbole:



- Spezialart wie Flip-Flop aber mehr Bits:



- Symbol:



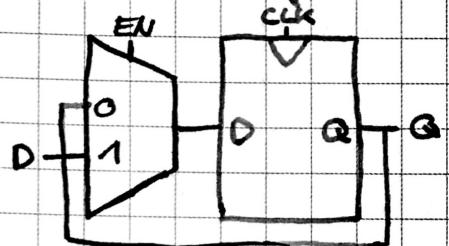
### Flip-Flops mit Clock enable

- Zusätzlicher Freigabeingang EN: wann D speichern

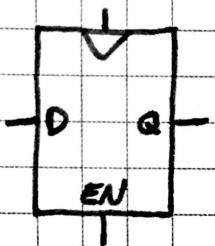
- Funktion:  $EN=1 \Rightarrow$  bei Taktflanke: D an Q

$EN=0 \Rightarrow Q$  bleibt gleich  
(Daten nicht geladen)

- interner Aufbau:



- Symbol:



### Zurücksetzbare Flip-Flops:

- Zusätzlicher RESET - Eingang

- Funktion:  $RESET=1 \rightarrow Q$  auf 0

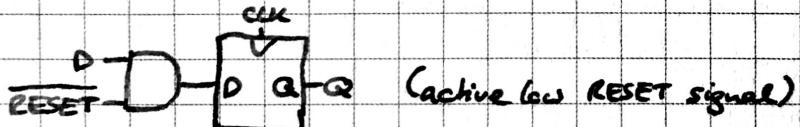
$RESET=0 \rightarrow$  normales D Flip-Flop

- Nützlich um z.B. Automat auf Zustand 0 zu setzen beim Starten

- Synchron: Zurücksetzen bei steigender Taktflanke

- Asynchron: Zurücksetzen bei RESET-Signal

Interner Aufbau für synchronen Reset:



Für asynchronen Reset: Flip-Flop-Schalter modifizieren

Schaltplansymbole:

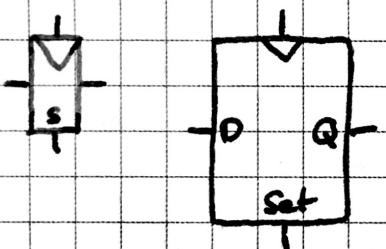


**Setzbare  
Flip-Flops:**

- Zusätzlicher Set-Eingang
- Funktion:  $\text{Set} = 1 \rightarrow Q \text{ auf } 1$

$\text{Set} = 0 \rightarrow \text{wie normales Flip-Flop}$

Schaltungsweise:



**Synchrone seq.  
Logik:**

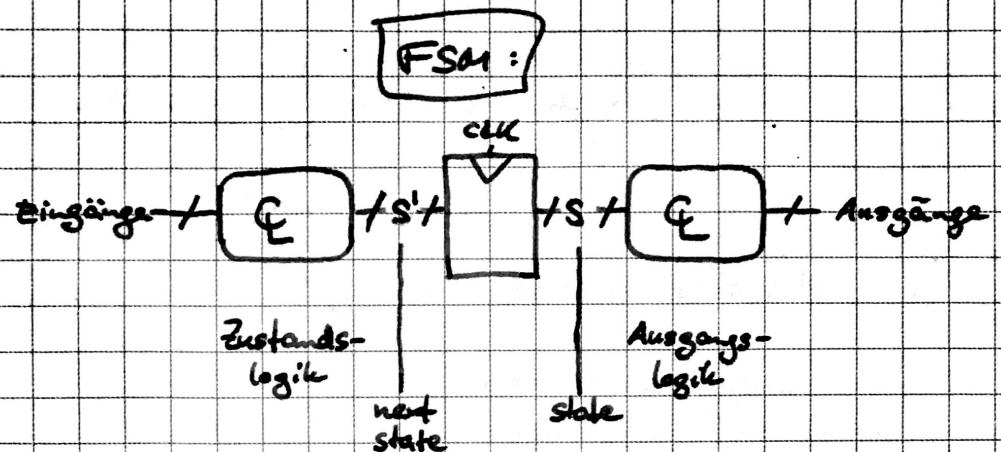
- Register: brechen Rückkopplungen, halten Zustand der Schaltung, ändern nur bei Taktflanken
- Schaltung wird mit Taktfläche synchronisiert

• Regeln:

- Jedes Element: entweder Reg. oder komb. Schaltung
- mindestens 1 Register
- 1 Taktsignal für alle Register
- Jeder Zyklus: mindestens 1 Register

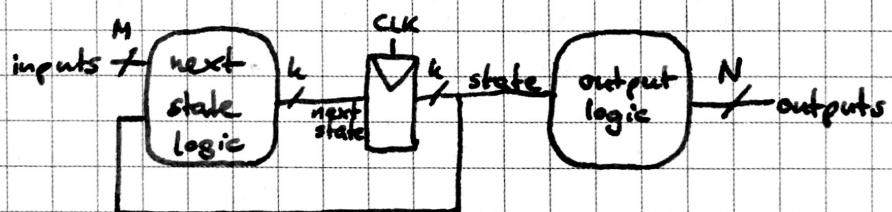
• 2 Beispiele:

- FSM
- Pipelines

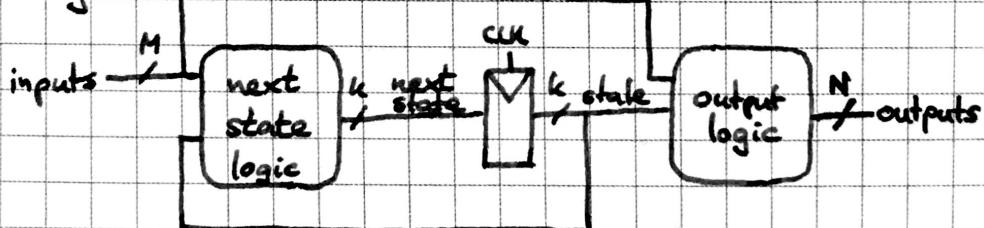


- Moore FSM: Ausgänge von Zustand abhängig
- Mealy FSM: Ausgänge hängen auch von Eingängen ab

→ Moore:



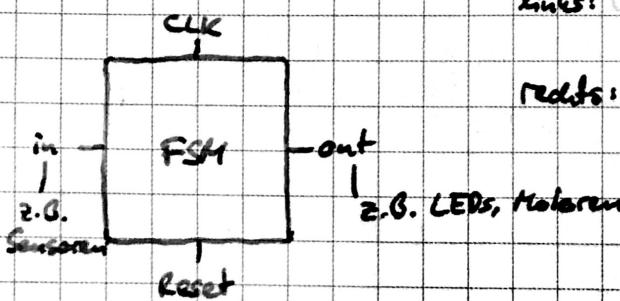
→ Mealy:



**Entwurfsverfahren für FSM:**

- Ein-/Ausgänge definieren
- Zustandsdiagramm zeichnen
- Zustandsübergangstabelle (bei Mealy: inkl. Ausgang/Ausgänge)
- Zustände kodieren (binär, one-hot, ...)
- Moore: kodierte Zustände in Zustandsübergangstabelle  
↳ Ausgangstabelle
- Boole'sche Gleichungen für Zustandsübergang und Ausgangslogiken aufstellen:  $S'_1 = \dots$ ,  $S'_0 = \dots$ ,  $Y_1 = S_1 \cdot \dots$ ,  $Y_2 = S_2 \cdot \dots$   
⇒ Schaltplan mit Gattern & Registern aufstellen

Block Box:



links: Zustandsübergangslogik

rechts: Ausgangslogik  
z.B. LEDs, Motoren

- Zustandsübergangs- und Ausgangslogiken mit „one-hot“-Kodierung oft kleiner & schneller, dafür mehr Flip-Flops als bei Binärkodierung

**Zeitanforderungen an Eingangssignale:**

- Dekomposition: Aufteilen komplexer FSM in einfachere interagierende FSMs / „Zerlegen“ von Zustandsaut.

- Setup-Zeit  $t_{\text{setup}}$ : Zeitintervall vor Taktflanke, in dem sich D nicht ändern darf
- Hold-Zeit  $t_{\text{hold}}$ : Zeitintervall nach Taktflanke, in dem D stabil sein muss
- Ablaufzeit  $t_a$ : Zeitintervall um Taktflanke, in dem D stabil sein muss;  $t_a = t_{\text{setup}} + t_{\text{hold}}$

Moore Zustandsübergangstabelle:

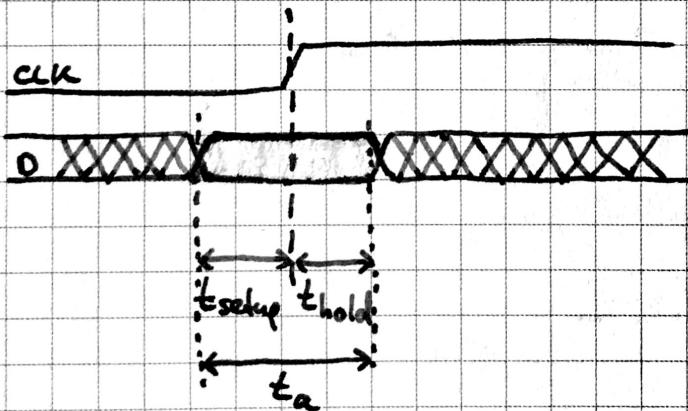
Alt. Zustand	Eingang	Nächster Zstd.
$S_1 \dots S_n$	A	$S'_1 \dots S'_n$

Moore Ausgangstabelle:

Alt. Zustand	Ausgang
$S_1 \dots S_n$	Y

Mealy Zustandsübergangs- und Ausgangstabelle:

Alt. Zustand: Eingang	Nächster Zstd: Ausgang
$S_1 \dots S_n$	A



→ Eingänge müssen mindestens ab  $t_{\text{setup}}$  vor Flanke und mindestens bis  $t_{\text{hold}}$  nach Taktflanke stabil sein

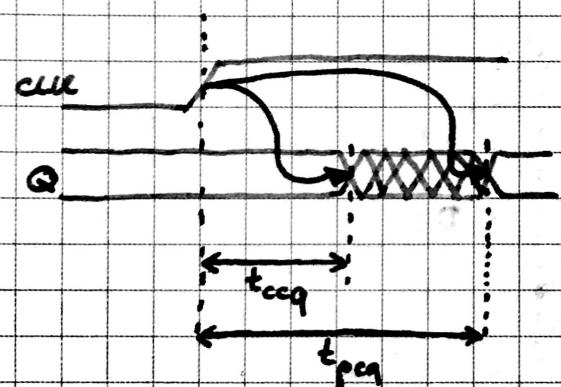
Zeitanforderungen  
an Ausgangssignale

- Laufzeitverzögerung (propagation delay)  $t_{pq}$

Zeitintervall nach Taktflanke, nach dem Q sich nicht mehr ändert

- Kontaminationaverszögerung (contamination delay)  $t_{ccq}$

Zeitintervall nach Taktflanke, nach dem Q beginnen könnte, sich zu ändern.



Anforderungen an  
Setup-Zeit:

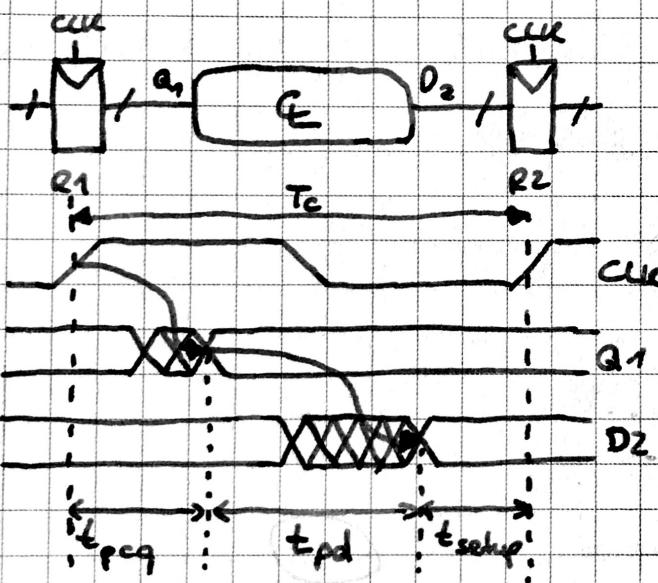
Takt  
/

$$T_c \geq t_{pq} + t_{pd} + t_{setup}$$

$$\Updownarrow t_{pd} \leq T_c - (t_{pq} + t_{setup})$$

$t_{pd} = \text{maximum}$   
propagation delay

sequencing overhead

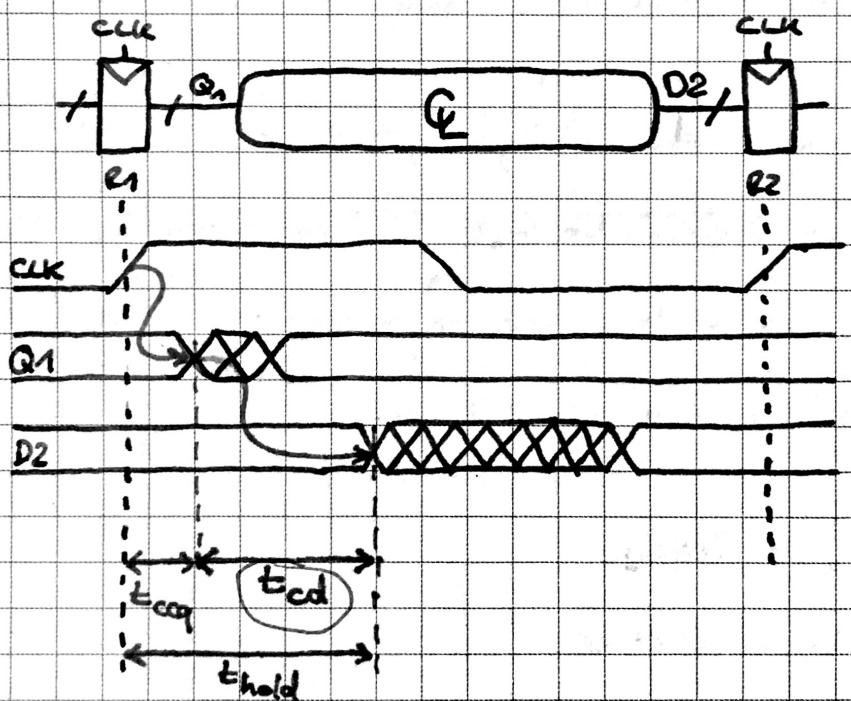


→ Einhalten der Setup-Zeit hängt von der Maximal-Verzögerung von Register R1 durch komb. Logik ab; D2 muss min. ab  $t_{setup}$  stabil sein vor Takt

Auflerungen an  
Hold-Zeit:

$$t_{hold} \leq t_{ccq} + t_{cd}$$

$$\Leftrightarrow t_{cd} \geq t_{hold} - t_{ccq} \rightarrow [t_{cd} = \text{minimum contention delay}]$$



→ Einhalten der Hold-Zeit hängt von der minimal-

Verzögerung von Register R1 durch komb. Logik ab

→ Der Eingang an Register R2 muss mindestens bis

$t_{hold}$  nach der Taktflanke stabil sein (D2)

Insgesamt:

Setup + Hold

geben maximalen und minimalen Delay der kombinatorischen Logik zwischen Flip-Flops.

Der maximale Delay begrenzt die Anzahl aufeinanderfolgender Gates auf dem kritischen Pfad und kann diese bei einer sehr schnellen Schaltung aber einschränken.

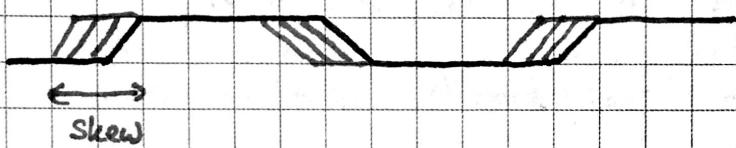
Ein paar Zeiten:

$$1\text{sek.} = 1000\text{ms}$$

$$1\text{ms} = 1000\mu\text{s} = 1000.000\text{ns} = 1000.000.000\text{ps}$$

**Taktverschiebung  
(clock skew)**

- Unterschiedliches Ankommen des Takts an versch. Reg.
- skew = Differenz der Ankunftszeit



→ immer mit worst-case arbeiten: R1 bekommt Letzte  
clock und C2 die Frühere;  $t_{hold}$  wird also  
bis zur frühesten-skew-clock gemessen.

Setup mit skew:

$$T_c \geq t_{pcq} + t_{pd} + t_{setup} + t_{skew}$$

$$T_{pd} \leq T_c - (t_{pcq} + t_{setup} + t_{skew})$$

Hold mit skew:

$$t_{ccq} + t_{cd} > t_{hold} + t_{skew}$$

$$t_{cd} > t_{hold} + t_{skew} - t_{ccq}$$

**Metastabilität:**

- jedes bistabile Element: 2 stable, 1 metastabiler Zustand dazwischen. z.B. Flip-Flop.
- metastabil = Q zwischen 1 und 0.
- wird durch koppelte Gatter regedreht.
- 0 oder 1. Paus: unbekannt!

### Parallelität:

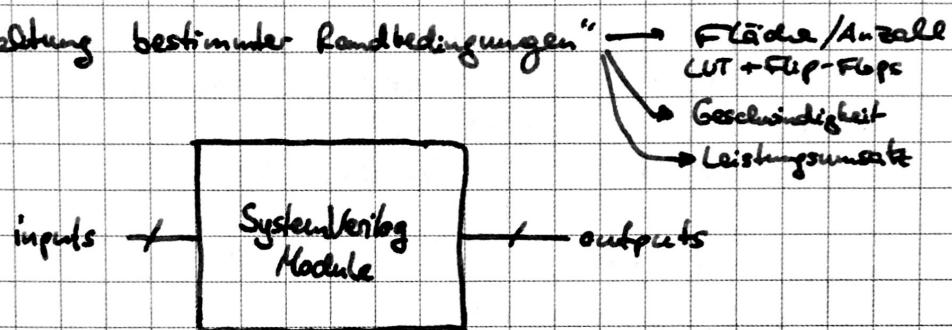
- räumlich & zeitlich (zeitl. = pipelining)
- Datensatz: Vektor aus zu verarbeitenden Eingabewerten
- Latenz: Eingabe Datensatz  $\rightarrow$  Ausgabe Ergebnis
- Durchsatz: Anzahl Datensätze pro Zeiteinheit
- $\rightarrow$  Parallelität erhält Durchsatz
- $\rightarrow$  Latenz durch Pipelining verlängert (auf Ergebnis warten)
  - ↳ dafür: höherer Durchsatz
  - $\Rightarrow$  Pipelining für viele Daten geeignet
- $\rightarrow$  zeitliche Parallelität: Fließbandprinzip (Buffierung, parallel bearbeitet  $\rightarrow$  Pipelining)

### Dynamische Disziplin:

Die Eingänge einer synchronen sequenziellen Schaltung müssen während der gesamten Abtastzeit  $t_a = t_{\text{setup}} + t_{\text{hold}}$  rund um die Taktflanke stabil sein.  
Damit ist gewährleistet, dass die Flip-Flops das richtige Signal verarbeiten und die Signale ein gültiges Logilevel besitzen / gültigen Wert.

## Synthese v. HDL:

- „Umsetzung einer algorithmischen Spezifikation in eine Schaltungssstruktur auf der Registertransfersene unter Einleitung bestimmter Randbedingungen“



→ 2 Arten von Beschreibung: Verhalten, Struktur

• SV ist case-sensitive!

• Bitweise Verknüpfungsoperatoren:

$\&$  → AND

$|$  → OR

$\wedge$  → XOR

$\sim$  → NOT

$\sim(A \& B) \rightarrow \text{NAND}$

$\sim(A | B) \rightarrow \text{NOR}$

• Bedingte Zuweisung:

assign  $y = s ? d1 : d0$

↓  
ternärer Operator

Höchste	$\sim$	NOT
$*, /, \%$	Multiplication, Division, Modulo	
$+, -$	Addition, Subtraktion	
$<, >$	Schrägen (logisch)	
$<<, >>$	Schrägen (arithmetisch)	
$<, \leq, >, \geq$	Vergleiche	
$=, !=$	gleich, ungleich	
$\&, \sim\&$	AND, NAND	
$\wedge, \sim\wedge$	XOR, XNOR	
$ , \sim $	OR, NOR	
?	ternärer Operator	

• Reduktionsoperator:  $y = \&a \Leftrightarrow y = a[3] \& a[2] \& a[1] \& a[0]$

Zellen:

$N^{\text{Bwert}}$  →  $N = \text{Bitbreite}$   
 $B = \text{Basis z.B. } b, d, o, h$

$$3^{\{b[0]\}} = b[0] \ b[0] \ b[0]$$

$$6'6\ 001\_010 = 6'6\ 001010 \quad (\text{--- ignoriert})$$

Look-up-Table:

(LUT)

Umsetzung durch Multiplizierer:

Select-Signal = Adresse

Inputs = "gespeicherte" Werte zum Ausgeben

→  $n$ -Bit LUT: beliebige  $n$ -Input boolesche Funktion:  
Wahrheitstabellen einfach in LUT speichern.

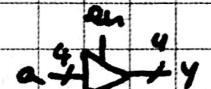
Hochleistung  
Ausgang  $z$  in  
SystemVerilog:

z.B.:

module tristate (input logic [3:0] a,  
input logic en,  
output tri [3:0] y);

assign  $y = en ? a : 4'bz$

endmodule



Floating value:  
einfach  $z$  und  
benötigte Breite  
in Bit angeben!

⇒  $N^{\text{Bz}}$

Verzögerung  
(Testbench):

# Zeiteinheiten

**always :**

allgemein: `always @ (sensitivity list)  
statement;`

→ Wenn Werte in s. list ändern: statement

**always\_comb:** Wenn immer sich ein Wert ändert

`begin  
:  
end` } bei mehr als 1 Anweisung

**always-latch:** Latch (in dieser VL nicht benutzt)

**always-f@ (sensitivity list):** Flip-Flop

↳ MERKE:  $\leftarrow$  / nicht bl. Zuweisungen um Wert zu übernehmen!

**Beispiele:**

→ Rücksetzbares D-Flip-Flop mit Taktfreigabe:

```
module flopren (input logic clk, reset, en,  
input logic [3:0] d,  
output logic [3:0] q);
```

```
//asynchroner reset mit clk enable:  
always @ (posedge clk, posedge reset)  
if (reset) q <= 4'b0;  
else if (en) q <= d;
```

endmodule

→ Latch:

```
module latch (input logic clk,  
input logic [3:0] d,  
output logic [3:0] q);
```

```
always_latch  
if (clk) q <= d;
```

endmodule

### case:

- genauso wie casez, if/else nur innerhalb always-Blöcken
- muss alle Möglichkeiten abdecken:

am besten durch default-Fall → wann wir passt  
(oder explizit alle Mögl. angeben)

- Variante: casez → akzeptiert "?" als don't-care.

### Zuweisungen:

- nicht-blockend: 

- wird parallel mit allen anderen nicht-blockenden Zuweisungen ausgeführt:

- ① „rechte Seite“ berechnet

- ② Alle Ergebnisse an „linker Seite“ zugewiesen

↳ Benutzung: always-if / synchrone seq. Logik

- blockend: 

- sorgt im Programmtext ausgeführt

- „blockierend“: während 1 blockende Zuweisung läuft,  
werden andere Zuweisungen blockiert.

- jede Aweisung berechnet für sich „rechte Seite“  
und weist linke Seite zu

**Regeln für Zuweisungen:**

- Synchron. Logik: always-ff @ (posedge clk) und <=
- einfache komb. Logik: assign ... = ...
- komplexere komb. Logik: always-comb und =
- vermeiden (NIEMALS tun!):
  - 1 Signal in mehreren Blöcken belegen
  - in 1 Block = und <= unzulässig
  - 1 Signal in 1 Block und in 1 assign

module form (input logic clk, reset,  
output logic

typedef enum logic [1:0] {S0, S1, S2, S3} stateType;  
stateType state, nextState;

Zustandsregister  
 $\left\{ \begin{array}{l} \text{always-ff @ (posedge clk, posedge reset)} \\ \quad \text{if(reset) state} \Leftarrow S0; \\ \quad \text{else state} \Leftarrow nextState; \end{array} \right.$

Zustandsübergangslogik  
 $\left\{ \begin{array}{l} \text{always-comb} \\ \quad \text{case(state)} \\ \quad \quad S0: \quad \text{if}(z) \text{nextState} = S1; \\ \quad \quad \quad \text{else} \text{nextState} = S3; \\ \quad \quad \vdots \\ \quad \quad \text{default: nextState} = S0; \\ \quad \text{endcase} \end{array} \right.$

Ausgangslogik  
 $\left\{ \begin{array}{l} \text{assign out} = (\text{state} == S3); \\ \text{endmodule} \end{array} \right.$

**FSM Template:**

Parameterisiertes  
Modul:

Beispiel: 2:1 Multiplexer mit beliebiger Busbreite:

module mux2

#(parameter WIDTH=8) // name und Standardwert

```
(input logic [WIDTH-1:0] d0, d1,  
input logic s,  
output logic [WIDTH-1:0] y);
```

assign y = s ? d1 : d0;

endmodule

↳ Instanz mit Standardwert: mux2 mymux(...);

→ Instanz mit 12-bit Bus: mux2  #(12) newmux(...);

Selbstprüfende  
Testbench:

module testbench();

logic a,b,y;

dummymodule dut(a,b,y);

initial begin

a=0; b=0; #10;

if(y != 1) \$display("00 failed");

} Pause, Wert prüfen,  
} bei ungültigem:  
\$display

a=1; #10;

if(y != 0) \$display("10 failed");

;

end

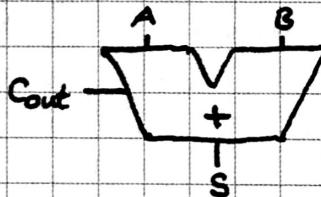
endmodule

Tastbuchen  
Info:

- Keine inputs/outputs
- Signale im Modell erstellen, mit initialisierten Signalen verbinden
- initial begin }  
:           } wird ix ausgeführt, der Reihe nach Signale testen zum Beispiel  
end }
- always begin }  
:           } wird unendlich oft ausgeführt, z.B für clock  
end }
- #S; : S Zeileinheiten werden
- immer aussagekräftige Stimuli überlegen

1-Bit  
Addierer:

- Halbaddierer:

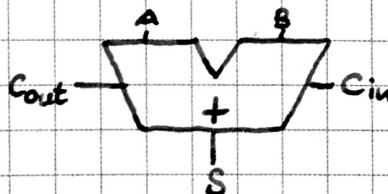


A	B	Cout	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$\rightarrow S = A \oplus B$$

$$Cout = AB$$

- Volladdierer:



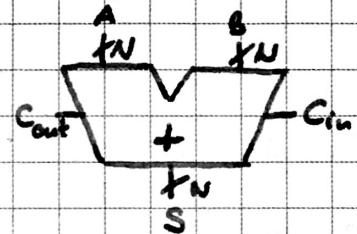
Cin	A	B	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$\rightarrow S = A \oplus B \oplus C$$

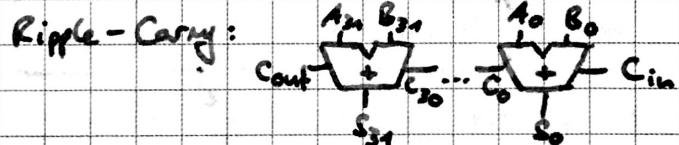
$$Cout = AB + AC_{in} + BC_{in}$$

**Halfbit-Addierer  
CDA:**

Carry-Propagator Adder: mehrbit-Addierer mit Weitergabe von Überträgen



- Typen:
- Ripple-Carry-Addierer (langsam)
  - Carry-Lookahead-Addierer (schneller) } mehr Fläche
  - Prefix-Addierer (noch schneller)



$$\hookrightarrow \text{Verzögerung } t_{\text{Ripple}} = N \cdot t_{\text{FA}} \quad (\text{N} = \text{Breite}, t_{\text{FA}} = \text{Verzögerung Full Adder})$$

**Carry-Lookahead  
Addierer:**

- Cout nicht Bit-zu-Bit, sondern aus k-Bit-Block

↳ 2 Signale:

- Generate: erzeugt Übertrag

- Propagate: keine weiter falls vorhanden

- Bits in Spalten organisiert: ( $i$ )

$$A_i \text{ und } B_i = 1 \iff G_i = A_i \cdot B_i \rightarrow \text{generated}$$

$$A_i \text{ oder } B_i = 1 \iff P_i = A_i + B_i \rightarrow \text{propagate}$$

↳ Spalte an Bit  $i$  erzeugt Übertrag  $C_i$ , wenn:

- sie einen erzeugt ( $G_i$ ) oder
- einen vom  $C_{i-1}$  eingehenden weiterleitet ( $P_i$ )

⇒ Übertrag  $C_i$  aus Spalte  $i$ :

$$C_i = (A_i \cdot B_i) + (A_i + B_i) \cdot C_{i-1}$$

$$= [G_i] + [P_i] \cdot C_{i-1}$$

⇒ Übertrag  $C_i$  durch  $i:j$  breiten Block:

$$C_i = G_{i:j} + P_{i:j} \cdot C_{j-1}$$

=> Addition:

① Berechne  $[G]$  und  $[P]$  für alle Spalten (Einzelbits)

② Berechne  $[G]$  und  $[P]$  für Gruppen von  $k$  Spalten  
( $k$  Bits)

③ Leite  $[C_i]$  nicht einzelnbitweise, sondern in  
 $k$ -Bit-Strängen weiter

(jeweils durch einen  $k$ -Bit Propagate/Generate  
Block)

Überlegung:

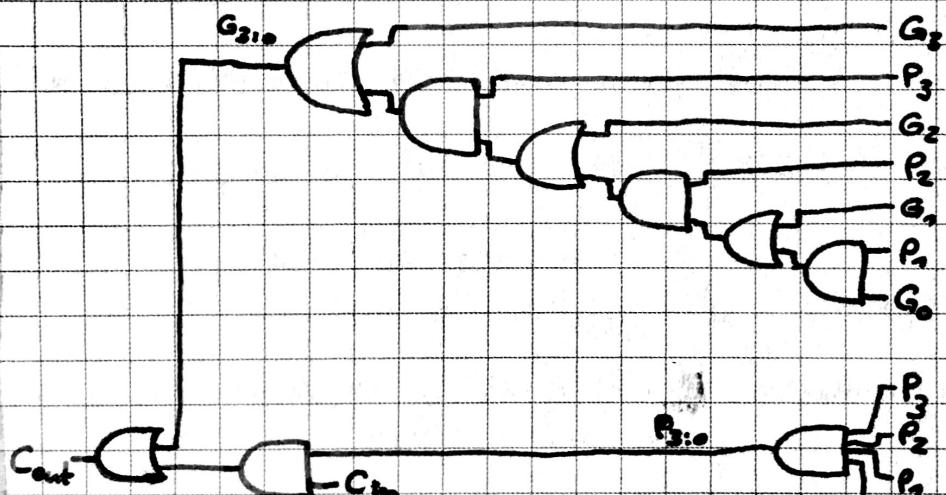
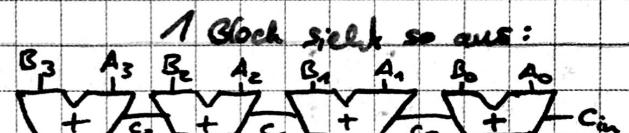
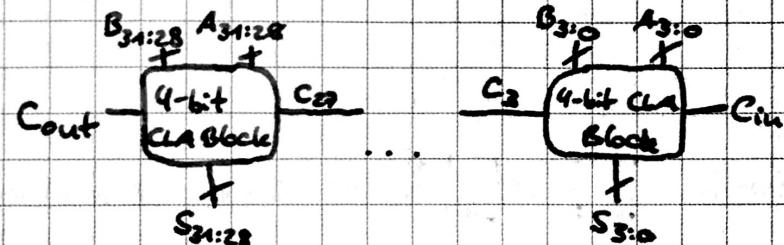
② 4 Bit Block:  $P_0 = P_3 \cdot P_2 \cdot P_1 \cdot P_0 \rightarrow$  Weiterleitung, wenn alle Spalten weiterleiten

$$G_{3:0} = G_3 + P_3 [G_2 + P_2 (G_1 + P_1 G_0)]$$

Spalte 3 erzeugt  
oder:

Spalte 3 leitet vorher ... recursive Darstellung  
erzeugten weiter

Komplettier CLA:



(mit  $G_i = A_i B_i$ ,  $P_i = A_i + B_i$ )

Verzögerung bei CLA:

$N$ -bit Addierer,  $k$ -bit Blöcke:

$$t_{CLA} = t_{PG} + t_{PG\text{-block}} + (N/k-1)t_{AND/OR} + k \cdot t_{PA}$$

mit:  $t_{PG}$ : Verzögerung P, G Berechnung für 1 Spalte

$t_{PG\text{-block}}$ : " " 1 Block

$t_{AND/OR}$ : Verzögerung durch AND/OR je  $k$ -bit CLA Block

$k \cdot t_{PA}$ : Verzögerung zur Berechnung der  $k$  höchstwertigen Summenbits

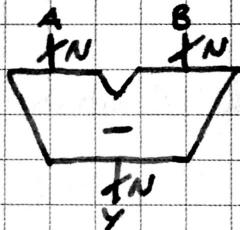
→ für  $N > 16$  ist ein CLA meist schneller als ein

Ripple-Carry Addierer

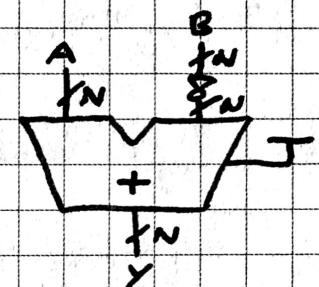
→ Verzögerung innerhalb von  $N$  abhängig!

Weitere Arithmetische Schaltungen:

Subtrahierer:



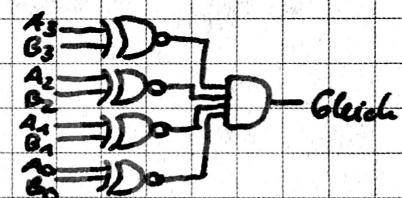
→ Implementierung:



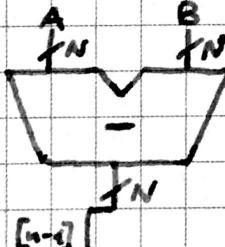
Vergleichar Gerlheit:



→ Implementierung:  
für  $N=4$

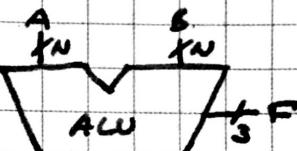
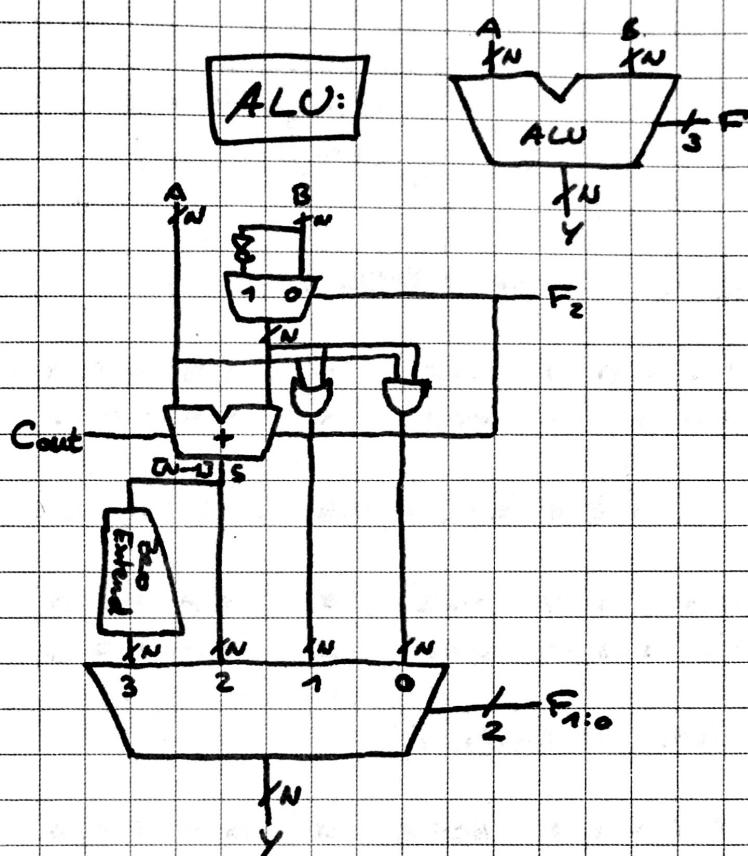


Vergleichar kleiner: ( $N$ -Bit Zweierkomplementzahlen)



$A < B$  wenn Ergebnis MSB 1 ist.

### ALU:



### Schiebeoperationen (Schiffler):

- Logisch = mit 0 füllen :

$$\begin{array}{r} 11001 \\ 11001 \end{array} \begin{array}{l} \gg 2 \\ \ll 2 \end{array} = \begin{array}{r} 00110 \\ 00100 \end{array}$$

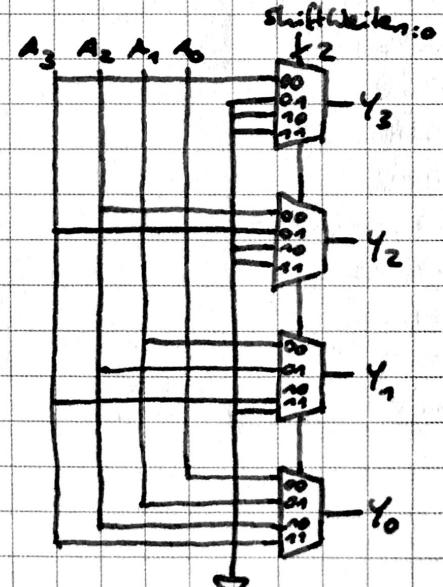
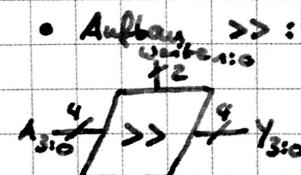
- Arithmetisch = beim Rechtsverschieben mit neg. Füllen:

$$\begin{array}{r} 11001 \\ 11001 \end{array} \begin{array}{l} \gg 2 \\ \ll 2 \end{array} = \begin{array}{r} 11110 \\ 00100 \end{array}$$

- Rotieren: unendlich

$$\begin{array}{r} 11001 \\ 11001 \end{array} \begin{array}{l} \text{ROR } 2 \\ \text{ROL } 2 \end{array} = \begin{array}{r} 01110 \\ 00111 \end{array}$$

- Aufbau >>:



F <sub>2:0</sub>	Funktion
000	A & B
001	A   B
010	A + B
011	nicht verwendet
100	A & ~B
101	A   ~B
110	A - B
111	SLT

set less than

### Anwendung:

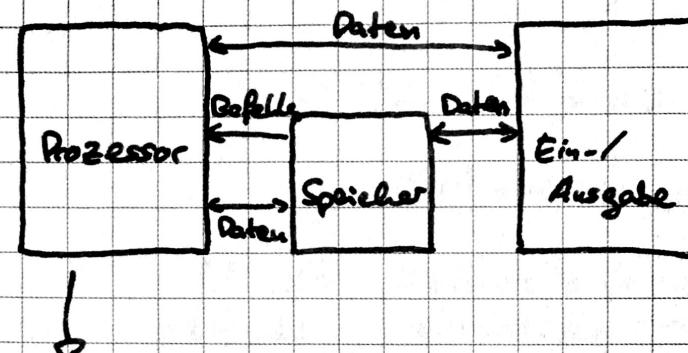
<< um N Stellen = Wert  $\cdot 2^N$

$$\hookrightarrow 0001 << 3 = 1000 \quad (1 \cdot 2^3)$$

>> um N Stellen = Wert /  $2^N$

$$\begin{aligned} \hookrightarrow 010000 >> 4 &= 000001 \quad (16 : 2^4) \\ 100000 >> 2 &= 111000 \quad (-32 : 2^2) \end{aligned}$$

### Rechensystem:



Operationswerk: arithmetische + logische Funktionen

Steuerwerk: steuert Operationswerk und I/O - System

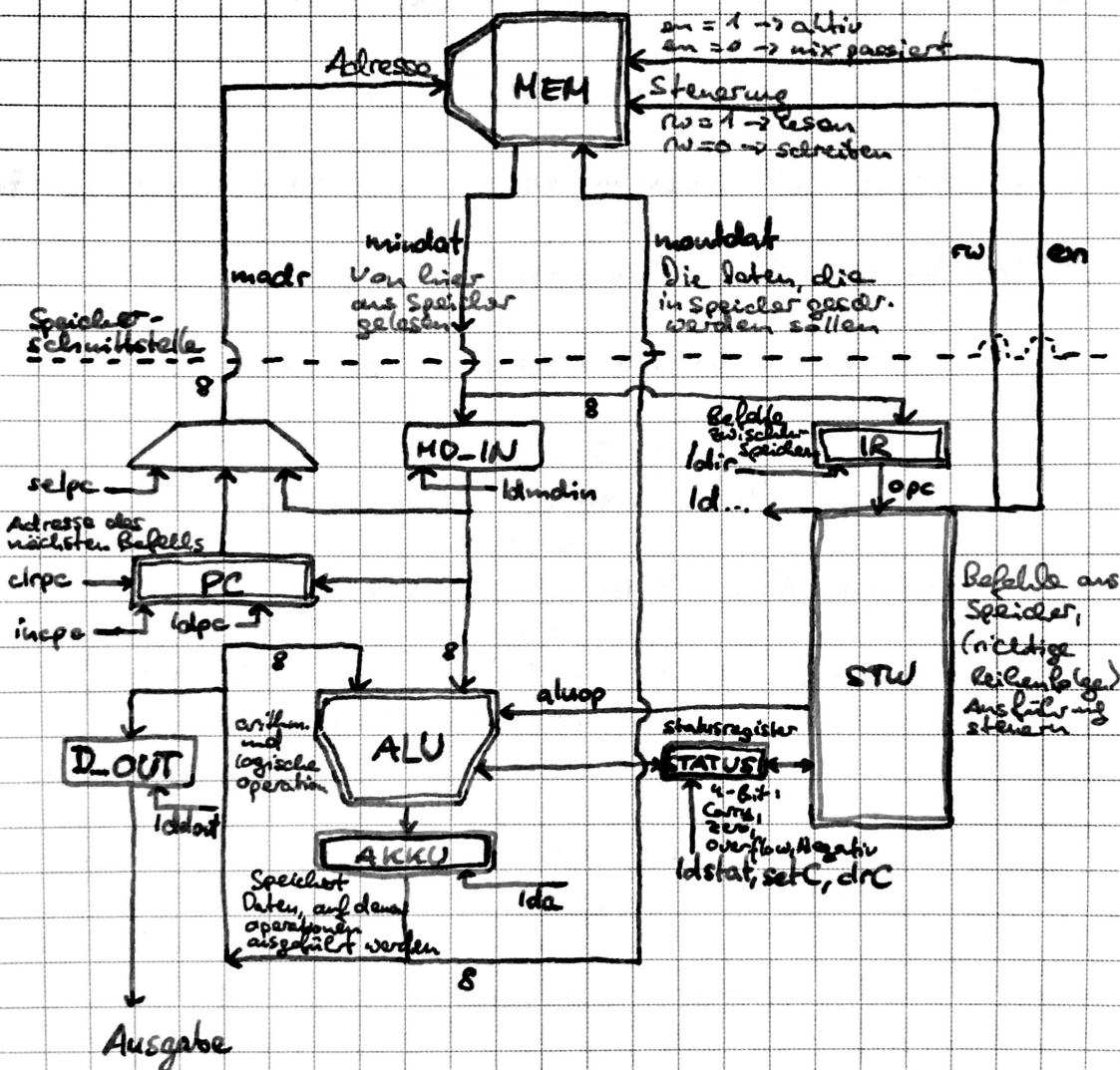
Steuerung passt durch Befehle

Maschinenebene:  
add  
not  
ldr

Befehlsatz

Mikroebene:  
ldr  
:  
realisieren  
Maschinenebene

## Struktur:



**MEM** = Speichersystem

**madr** = Adressen, die im Speicher angelegt werden

**mindr** = Daten, die im Speicher gel.

**moutdat** = Daten, die im Speicher gesetzt werden

**MEM** = Speicher

**STW** = Steuerwerte

**rw/en** = Steuerw. für Speicher

**PC** = Programmzähler

**ALU** = Rechen-/Operationswerk

**IR** = Befehlsregister

**ACKCU** = Alkumulator

## ALU:

- Schaltwerk: 2 Inputs → 1 Output

↳ rein kombinatorisch, kein Takt!

- Zusätzlich: Statusregister beeinflussen Berechnung / werden gesetzt
- Operationsauswahl: Steuercode
- Akkumulator: Zwischenergebnisse speichern

↳ vor oder hinter ALU

↳ Shiftoperationen ohne ALU möglich (evtl.)

⇒ Freiheit beim Entwurf → Leistung!

- Statusregister:   
 $Z = ZERO = \text{Alu-Inhalt ist } 0$   
 $N = NEGATIV = \text{Alu-Inhalt ist } < 0$   
 $C = CARRY = C_{in} \text{ bei neuer Addition, speichert Carry}$   
 $OV = OVERFLOW = \text{Überlauf entstanden}$

### Befehlscode:

- variiert: CISC (viele Befehle), RISC (reduced inst. set)  
complex i.s. computer
- so gut wie immer: AND, OR, NOT, ADD, ...
- OPC = opcode = Kodierung der Befehle
- MNEMONICS = symbolische Befehle, die OPC's repräsent.

### Interpretation / Ausführung:

- Befehlsholphase = instruction fetch
    - ↳ Prozessor / SW: Befehle aus Speicher lesen lassen
  - Befehlsdekodierung = instruction decode
    - ↳ Dekodieren, nachdem geladen wurde und im IR steht
  - Befehlausführung = instruction execute
    - ↳ ausgeführt, danach nächster Befehl aus MEM geladen
- => „3 Phasen der Befehlausführung“