DT Reference Sheet

Jonas Milkovits

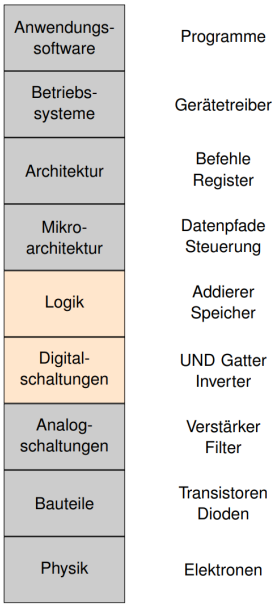
Last Edited: 26. April 2020

Inhaltsverzeichnis

1 Grundlagen

1.1 Digitale Abstraktion und ihre technische Umsetzung

- **Abstraktion**
  - Beschränken auf wesentliche Eigenschaften
  - Redundantes wird aufgrund von Abstraktion weggelassen
- **Schichtenmodell**



  - Untere Schicht erbringt Dienstleistungen für nächst höhere Schicht
  - Obere Schicht nutzt nur Dienste der nächst niedrigeren Schicht
  - Eindeutige Schnittstelle zwischen den Schichten
  - Vorteile:
    - Austauschbarkeit einzelner Schichten
    - Nur bearbeitende Schicht muss dem Nutzer bekannt sein
    - Festdefinierte Funktionalität niedrigerer Schichten
  - Nachteile:
    - ggf. geringe Leistungsfähigkeit des Systems
- **Disziplin**
  - wissentliche Beschränkung der Realisierungsmöglichkeiten
  - z.B.: Digitale Entwurfsdisziplin | Digitale Abstraktion
    - Arbeit mit diskreten statt stetigen Spannungspegeln
    - Einfacher Entwurf → Entwurf komplexerer Schaltungen
- **Wesentliche Techniken**
  - Hierarch**y** | Aufteilen eines Systems in Module und Untermodule
  - Modularit**y** | wohldefinierte Schnittstellen und Funktionen
  - Regularit**y** | bevorzuge einfache Lösungen für einfachere Wiederverwendbarkeit
- **Bits und Bytes | Digitale Abstraktion**
  - Grundlagen
    - Beschränkung auf zwei unterschiedliche Werte 0 | 1
    - Bit (binary digit) Maßeinheit für Information (kleinstmöglich)
    - Bitfolgen → mehrere Bits hinteinander
    - Anzahl der möglichen Zustände: 2^n
    - $2^5 = 32$ | $2^{10} = 1024$
  - Größenordnungen

1 Ki	Kibi	2^{10}	1024
1 Mi	Mebi	2^{20}	1024×1024
1 Gi	Gibi	2^{30}	$1024 \times 1024 \times 1024$
1 Ti	Tebi	2^{40}	$1024 \times 1024 \times 1024 \times 1024$
1 k	Kilo	10^3	1000
1 M	Mega	10^6	1000×1000
1 G	Giga	10^9	$1000 \times 1000 \times 1000$
1 T	Tera	10^{12}	$1000 \times 1000 \times 1000 \times 1000$

- Nomenklatur
  - Nibble: Besteht auf 4 Bit
  - Byte: Besteht auf 8 Bit
  - Halbwort: Abhängig von Registerbreite (32Bit/64Bit) | Hälfte eines Worts
  - Wort: Entspricht Registerbreite (32Bit/64Bit)

1.2 Zahlensysteme

- **Darstellung von natürlichen Zahlen**
  - Allgemein | vorzeichenloses Stellenwertsystem
    - Basis $b \in \mathbb{N} \wedge b \geq 2$
    - Menge der verfügbaren Ziffern $Z_b := 0, 1, .., b-1$
    - $u_{b,k}$ bildet Ziffernfolge der Breite $k \in \mathbb{N}$ auf eine natürliche Zahl ab
    - $u_{b,k} : (a_{k-1}...a_1a_0) \in Z_b^k \rightarrow \sum_{i=0}^{k-1} a_i * b^i \in \mathbb{N}$
    - polyadisches Zahlensystem → Wertigkeit von Position abhängig
    - niedrigstwertige Stelle (**LSD**, least significant digit): a_0
    - höchstwertige Stelle (**MSD**, most significant digit): a_{k-1}
    - Anzahl der darstellbaren Werte: b^k
  - Beispiele:
    - Dezimal: $302 = 3 * 100 + 0 * 10 + 2 * 1 = 302_{10}$
    - binär: $1101_2 = 1 * 2^3 + 1 * 2^2 * 0 * 2^1 + 1 * 2^0 = 13_{10}$
    - hexadezimal: $1F3A_16 = 1 * 16^3 + 15 * 16^2 + 3 * 16^1 + 10 * 16^0 = 7994_{10}$
- **Umrechnen von Zahlensystemen**
  - Binär/Hexadezimal → Dezimal
    - polyadische Abbildung verwenden
    - $u_{2,5}(10011_2) = 2^0 + 2^1 + 2^4 = 19_{10}$
    - $u_{16,3}(4AF_{16}) = 15 * 16^0 + 10 * 16^1 + 4 * 16^2 = 1199_{10}$
    - (Hinweis: $16^2 = 256$ | $16^3 = 4096$)
  - Binär ↔ Hexadezimal
    - Nibble-weise umwandeln
    - bei LSD beginnen
    - führende Nullen weglassen oder ergänzen (je nach geforderter Bitbreite)
    - $11\ 1010\ 0110\ 1000_2 = 3A68_{16}$
    - $7BF_{16} = 111\ 1011\ 1111_2$
  - Dezimal → Binär

$$
\begin{aligned}
& 53_{10} \\
&= 32 + \underline{21} \\
&= 32 + 16 + \underline{5} \\
&= 32 + 16 + 4 + 1 \\
&= 2^5 + 2^4 + 2^2 + 2^0 \\
&= 11\ 0101_2
\end{aligned}
$$

Abbildung 1: Maximale Zweierpotenzen abziehen

$$
\begin{aligned}
& 53_{10} \\
&= 2 \cdot \underline{26} + 1 \\
&= 2 \cdot (2 \cdot \underline{13} + 0) + 1 \\
&= 2 \cdot (2 \cdot (2 \cdot \underline{6} + 1) + 0) + 1 \\
&= 2 \cdot (2 \cdot (2 \cdot (2 \cdot \underline{3} + 0) + 1) + 0) + 1 \\
&= 2 \cdot (2 \cdot (2 \cdot (2 \cdot \underline{1} + \underline{1}) + \underline{0}) + \underline{1}) + \underline{0}) + \underline{1} \\
&= 11\ 0101_2
\end{aligned}
$$

Abbildung 2: Halbieren mit Rest

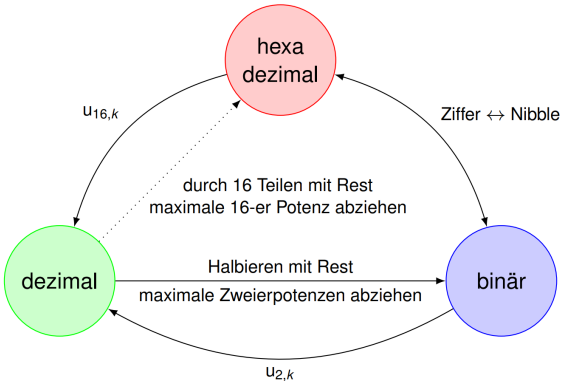


Abbildung 3: Übersicht über Umrechnungsverfahren

- **Addition von vorzeichenlosen Binärzahlen**

		1	1	1			Übertrag
			1	0	1	1	Summand
–	+		0	1	1	0	Summand
	=	1	0	0	0	1	Summe

Überlauf

  - Überlauf bei festen Bitbreiten
  - Informationsverlust!
  - z.B.: 4-Bit Addierer $11 + 6 = 1$

- **Vorzeichenbehaftete Binärzahlen**
  - Vorzeichen und Betrag
    - Erstes Bit als Vorzeichen ($* (-1)^1$ oder $* (-1)^0$)
    - z.B.: $vb_{2,4}(1110_2) = (0 * 2^0 + 1 * 2^1 + 1 * 2^2) * (-1)^1 = -6_{10}$
    - inkompatibel mit binärer Addition
  - Zweierkomplement
    - $s_k : (a_{k-1}, ..., a_1, a_0) \in \mathbb{B} \rightarrow a_{k-1} * (-2^{k-1}) + \sum_{i=0}^{k-2} a_i * 2^i \in \mathbb{Z}$
    - Funktion s_k bildet eine Bitfolge der Breite k $\in \mathbb{N}$ auf eine ganze Zahl ab
    - Anzahl darstellbarer Werte: 2^k
    - Wertebereich: $\{-2^{k-1}, ..., 2^{k-1} - 1\}$
    - z.B.: $s_4(1010_2) = 0 * 2^0 + 1 * 2^1 + 0 * 2^2 + 1 * (-2^3) = -6_{10}$
    - kompatibel mit binärer Addition
  - Dezimal → Zweierkomplement
    - in beiden Fallen achten auf korrekte/geforderte Bitbreite
    - ggf. führende Nullen vor Betragsdarstellung

$$
\begin{aligned}
-53_{10} &= -64 + \underline{11} \\
&= -64 + 8 + \underline{3} \\
&= -64 + 8 + 2 + 1 \\
&= -2^6 + 2^3 + 2^1 + 2^0 \\
&= 100\ 1011_2
\end{aligned}
$$

Abbildung 4: Maximale Zweierpotenzen abziehen

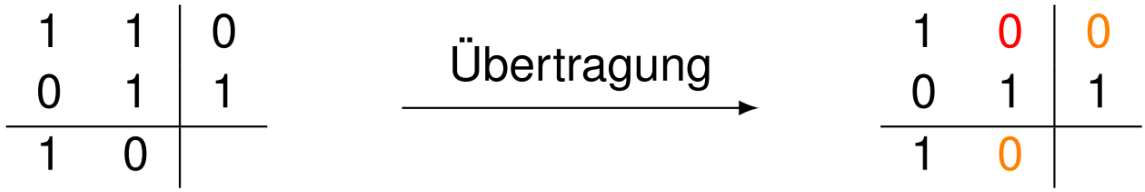
$$
\begin{aligned}
-53_{10} &= \overline{53_{10}} + 1 \\
&= \overline{011\ 0101_2} + 1 \\
&= 100\ 1010_2 + 1 \\
&= 100\ 1011_2
\end{aligned}
$$

Abbildung 5: Negieren und Inkrement(+1)

- **Bitbreitenerweiterung**
  - Notwendig für Addition verschiedener Bitbreiten
  - zero extension: Auffüllen mit führenden Nullen für vorzeichenlose Darstellung
  - sign extension: Auffüllen mit Wert des Vorzeichen-Bits für Zweierkomplement-Darstellung
  - z.B.: Falls in Aufgaben eine Bitbreite von 8Bit gefordert wird
- **Vergleich der binären Zahlendarstellungen**

1.3 Logikgatter

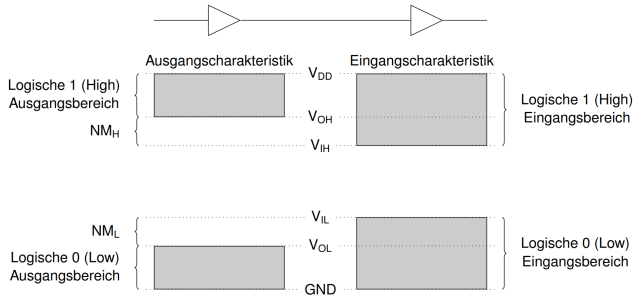
- **Logische Operationen**
  - verknüpfen binäre Werte $\mathbb{B}^n \to \mathbb{B}^k$
  - Charakterisierung durch Wahrheitstabellen
  - z.B.: $n = 1 : NOT \mid n = 2 : AND, OR, XOR \mid n = 3 : MUX$
- **Gatterarten auf Merkblatt**
- **Fehlerkorrektur mit Paritätsfunktion**
  - $Y = 1$, falls Anzahl an eingehenden Einsen ungerade
  - Verwendung in Fehlerkorrektur bei Übertragung
  - 1. Anhängen eines Paritätsbits
  - 2. Gesamtparität nach Übertragung berechnen
  - Verwendung von mehreren Paritätsbits für Nachricht für Fehlerkorrektur



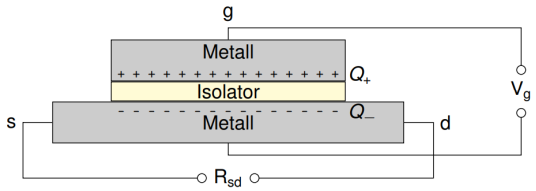
1.4 MOSFET Transistoren und CMOS Gatter

- **Spannungen als Logikpegel**
  - Definition von Logikpegeln für 0 und 1
  - $0V \rightarrow 0$ (GND, Voltage Source Source (V_{SS}))
  - $5V \rightarrow 1$ (Versorgungsspannung, Voltage Drain Drain (V_{DD}))

  - Definition von Spannungsbereichen aufgrund von Rauschen (z.B. Widerstände)
    - V_{IL}: größte Spannung, die Empfänger als 0 interpretiert (Voltage Input Low)
    - V_{IH}: kleinste Spannung, die Empfänger als 1 interpretiert (Voltage Input High)
    - V_{OL}: größte Spannung, die Treiber als 0 ausgibt (Voltage Output Low)
    - V_{OH}: kleinste Spannung, die Treiber als 1 ausgibt (Voltage Output High)
    - $NM_H = V_{OH} - V_{IH}$: oberer Störabstand (Noise Margin High)
    - $NM_L = V_{IL} - V_{OL}$: unterer Störabstand (Noise Margin Low)



- **Feldeffekt-Transistoren**
  - Logikgatter werden aus Transistoren aufgebaut (heutzutage hauptsächlich FET)
  - Transistor:
    - Spannungsgesteuerte Schalter
    - Zwei Anschlüsse werden je nach Spannung am 3. Eingang (Gate) getrennt oder verbunden
  - Der Feldeffekt
    - Prinzip des spannungsgesteuerten Widerstands



    - Metallische Streifen bilden Plattenkondensator
    - Steuerspannung V_g beeinflusst Menge der freien Ladungsträger
    - Steuerung des Widerstands R_{sd} mithilfe der Steuerspannung V_g
    - Nutzung von Halbleitern, da der Feldeffekt dort technisch nutzbar
    - meist dotierte Silizium-basierte Halbleiter
  - MOSFETs
    - Metalloxid-Halbleiter (MOS) Transistoren
    - Undotiertes Silizium als Gate
    - Oxid (Siliziumdioxid) als Isolator
    - Dotiertes Silizium als Substrat und Anschlüsse

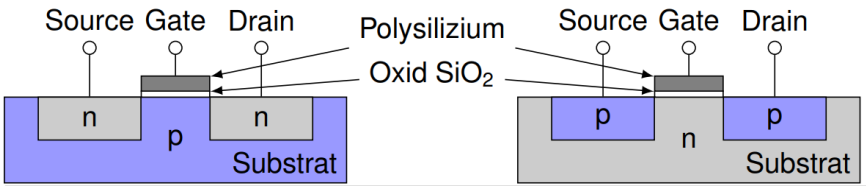


Abbildung 6: Links: nMOS | Rechts: pMOS

* nMOS: $Gate = 0 \rightarrow AUS \quad Gate = 1 \rightarrow AN$
* pMOS: $Gate = 0 \rightarrow AN \quad Gate = 1 \rightarrow AUS$

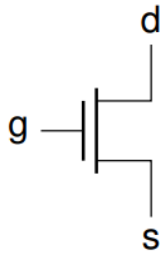


Abbildung 7: nMOS

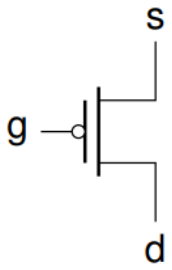
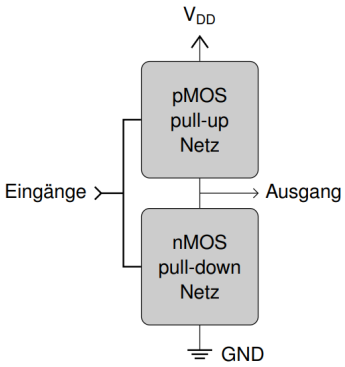


Abbildung 8: pMOS (Kreis wie im p)

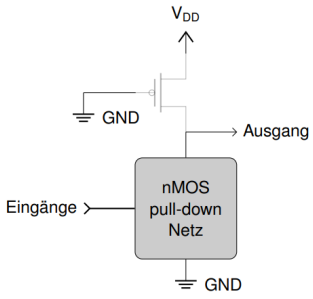
- **CMOS-Gatter**



* Kombinieren von komplementären Transistoren
* nMOS leiten 0'en gut weiter → Source an GND anschließen
* pMOS leiten 1'en gut weiter → Source an V_{DD} anschließen

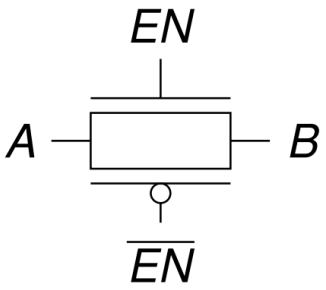
⇒ Complementary Metal-Oxide-Semiconductor sn(CMOS) Logik

– Struktur:
  * pMOS Parallelschaltung ⇔ nMos Serienschaltung
  * nMOS Parallelschaltung ⇔ pMos Serienschaltung
– Pseudo-nMOS Gatter



· Ersetzt das pMOS-Pull-Up-Netz
→ durch schwachen, immer eingeschalteten pMOS
· Pull-Up kann durch Pull-Down überstimmt werden

– Transmissionsgatter



· besseres Schalter ⇒ leitet 0 und 1 gut weiter

· $EN = 1$ *und* $\overline{EN} = 0 \Rightarrow$ EIN (A mit B verbunden)

· $EN = 0$ *und* $\overline{EN} = 1 \Rightarrow$ AUS (A nicht mit B verbunden)

1.5 Leistungsaufnahme

- **Leistung** = Energieumsatz/-verbrauch pro Zeiteinheit
- **Statische Leistungsaufnahme:**
  - Leistungsbedarf wenn kein Gatter schaltet
  - verursacht durch Leckstrom I_{DD} (nicht vollständiges Abschalten, Pseudo-nMOS,...)
  - $P_{static} = I_{DD} * V_{DD}$
- **Dynamische Leistungsaufnahme:**
  - Aufladen der Gate-Kapazität C von $0As$ auf $Q = C * V_{DD}$
  - Schaltung wird mit Frequenz f betrieben ⇒ Transistoren schalten f-mal pro Sekunde
  - Nur die Hälfte davon sind Aufladungen
  - $P_{dynamic} = I * V = \frac{1}{2} C * V_{DD}^2 * f$
  - Beispiel Leistungsaufnahme:

▶ Abschätzen der Leistungsaufnahme für einen Netbook-Prozessor
- ▶ Versorgungsspannung $V_{DD} = 1{,}2\,V$
- ▶ Transistorkapazität $C = 20\,nF$
- ▶ Taktfrequenz $f = 1\,GHz$
- ▶ Leckstrom $I_{DD} = 20\,mA$

⇒ $P = \frac{1}{2} \cdot C \cdot V_{DD}^2 \cdot f + I_{DD} \cdot V_{DD} = 14{,}4\,W + 24\,mW$

- **Moore's sches Gesetz** (Alle 18 Monate verdoppelt sich die Anzahl der Transistoren auf einem Chip)

2 Kombinatorische Schaltungen

2.1 Boole'sche Gleichungen und Algebra

- **Kombinatorische Logik**
  - Eingänge führen durch bestimmtes Verhalten zu Ausgängen (Funktionales und Zeitverhalten)
  - kombinatorische Logik ⇒ hängt nur von Eingangswerten ab
  - sequentielle Logik ⇒ hängt von Eingangswerten und internem Zustand ab
  - Eigenschaften:
    - Jedes Schaltungselement ist selbst kombinatorisch
    - Jeder Pfad besucht jeden Knoten maximal einmal (zyklenfrei)
- **Bool'sche Gleichungen**
  - Grundlagen:
    - beschreiben Ausgänge als Funktion der Eingänge ⇒ Spezifikation des funktionalen Verhaltens
    - Operatoren: (sortiert nach Operatorpräzedenz)
      - NOT: $\overline{A}$
      - AND: $A\ B = A * B$

        - XOR: $A \oplus B$
        - OR: $A + B$
    - Komplement: Intervierte boole'sche Variable ($\overline{A}$)
    - Literal: Variable oder ihr Komplement($A, \overline{A}$)
    - Impliktant: Produkt von Literalen (ABC)
    - Minterm: Produkt (UND, Konjunktion) über alle Eingangsvariablen (ABC)
    - Maxterm: Summe (ODER, Disjunktion) über alle Eingangsvariablen ($A + B + C$)
  - Minterm:
    - Produkt, das jede Eingangsvariable genau einmal enthält
    - einspricht einer Zeile in Wahrheitstabelle
    - Jeder Minterm wird für genau eine Eingangskombination **wahr**
    - Disjunktive Normalform(DNF) = Sum-Of-Products(SOP)
      - Summe aller Minterme, für welche die **Funktion wahr** ist ⇒ nur eine DNF
      - z.B.: $Y = m_1 + m_2 = \overline{A}\ B + A\ \overline{B}$
  - Maxterm:
    - Produkt, das jede Eingangsvariable genau einmal enthält
    - einspricht einer Zeile in Wahrheitstabelle
    - Jeder Maxterm wird für genau eine Eingangskombination **falsch**
    - Konjunktive Normalform(KNF) = Product-of-sums (POS)
      - Produkt aller Maxterme, für welche die **Funktion falsch** ist ⇒ nur eine KNF
      - z.B.: $Y = M_0\ M_3 = (A + B)(\overline{A} + \overline{B})$

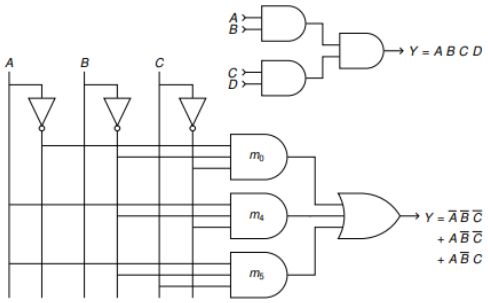
- **Boole'sche Algebra**
  - Axiome: grundlegende Annahmen der Algebra (nicht beweisbar)
  - Theoreme: komplexere Regeln, die sich aus Axiomen ergeben (beweisbar)
  - Optimierungen durch Begrenzung auf $\mathbb{B}$
  - Axiome und Theoreme zu finden auf Hilfsblatt

- **Logikminimierung**
  - TO-DO (Übungen)
  - Bubble-Pushing
    - Verschieben von Invertierungsblasen zur Vereinfachung von Schaltern
    - Über Gatter hinweg ⇒ De Morgan | And ⇔ OR | Verschieben der Blasen
    - Zwischen Gattern ⇒ Über Leitungen | Involution (doppelte Blasen)
    - Entfernen verbleibender Buffer

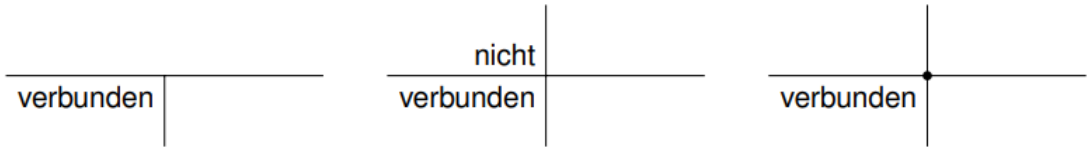
2.2 Kombinatorische Grundelemente

- **Zweistufige Logik**
  - direkte Umsetzung der disjunktiven Normalform (DNF)
  - aufwändige Darstellung und Realisierung
  - Eingangsliterale: Ein Inverter pro Variable
  - Minterme: Je ein AND Gatter an passende Literale anschließen
  - Summe: ALle Minterme an ein OR Gatter anschließen

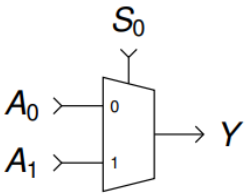
  ⇒ jede boole'sche Funktion mit Basisgattern realisierbar
  - z.B.:



- **Konventionen für Schaltpläne**
  - Eingänge links/oben | Ausgänge rechts/unten
  - gerade und rechtwinklige Verbindungen

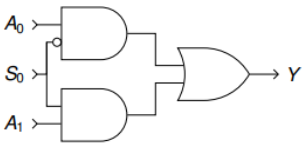


- **Multiplexer** $MUXn : \mathbb{B}^{n+[log_2 n]} \to \mathbb{B}$
  - Selektiert einen der Datenausgänge $A_0, ..., A_n - 1$ als Ausgang Y
  - $k = [log_2 n]$ Steuersignale $S_0, ..., S_{k-1}$
  - $Y = A_{u_{2,k}}(S_{k-1}...S_0)$



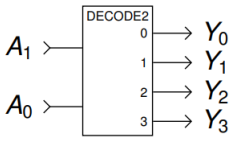
S_0	A_0	A_1	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

S_0	Y
0	A_0
1	A_1



- **Dekodierer** $DECODEn : \mathbb{B}^n \to \mathbb{B}^{2^n}$
  - n Eingänge $A_0, ..., A_{n-1}$
  - 2^n Ausgänge $Y_0, ..., Y_{2^n-1}$
  - $Y_i = u_{2,n}(A_{n-1}...A_0) == i \ ? \ 1 : 0$ (”One-Hot” Kodierung)

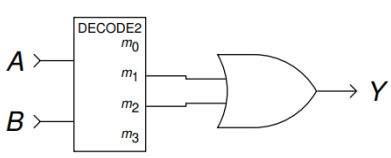
A_1	A_0	Y_0	Y_1	Y_2	Y_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1



  - Logikrealisierung: Summe über Minterme, auf denen Zielfunktion wahr ist

⇒ Decoder ersetzt erste Stufe der zweistufigen Logikrealisierung

A	B	$Y = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0



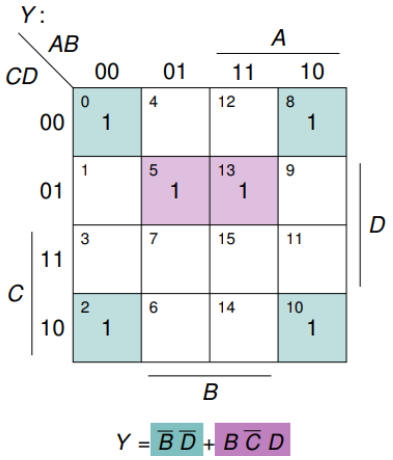
2.3 Karnaugh-Diagramme (KV)

- **Allgemein**
  - Minimierung boole'scher Ausdrücke durch Zusammenfassen von Mintermen
  - $Y = AB + A\overline{B} = A$

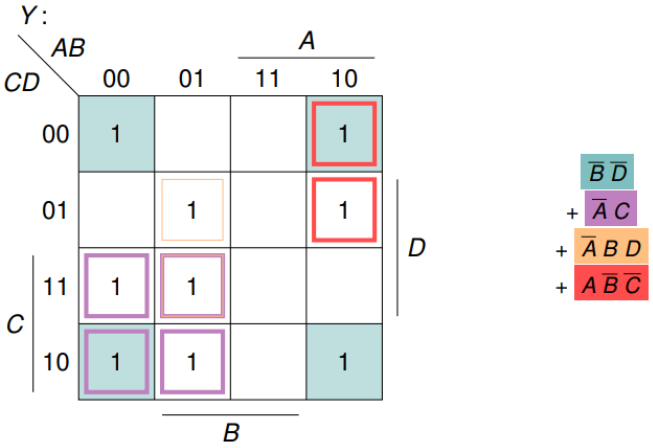
  $\Rightarrow$ Graphische Darstellung der Zusammenhänge durch Karnaugh-Diagramme
  - via GrayCode (immer nur ein geändertes Bit nebeneinander)
  - Zusammenhängende Minterme dadurch besser erkennbar
  - Don't Cares sowohl in DNF als auch KNF

- **Beispiel für vier Eingänge**

A	B	C	D	Y	Minterm
0	0	0	0	1	$m_0 = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D}$
0	0	0	1	0	$m_1 = \overline{A}\,\overline{B}\,\overline{C}\,D$
0	0	1	0	1	$m_2 = \overline{A}\,\overline{B}\,C\,\overline{D}$
0	0	1	1	0	$m_3 = \overline{A}\,\overline{B}\,C\,D$
0	1	0	0	0	$m_4 = \overline{A}\,B\,\overline{C}\,\overline{D}$
0	1	0	1	1	$m_5 = \overline{A}\,B\,\overline{C}\,D$
0	1	1	0	0	$m_6 = \overline{A}\,B\,C\,\overline{D}$
0	1	1	1	0	$m_7 = \overline{A}\,B\,C\,D$
1	0	0	0	1	$m_8 = A\,\overline{B}\,\overline{C}\,\overline{D}$
1	0	0	1	0	$m_9 = A\,\overline{B}\,\overline{C}\,D$
1	0	1	0	1	$m_{10} = A\,\overline{B}\,C\,\overline{D}$
1	0	1	1	0	$m_{11} = A\,\overline{B}\,C\,D$
1	1	0	0	0	$m_{12} = A\,B\,\overline{C}\,\overline{D}$
1	1	0	1	1	$m_{13} = A\,B\,\overline{C}\,D$
1	1	1	0	0	$m_{14} = A\,B\,C\,\overline{D}$
1	1	1	1	0	$m_{15} = A\,B\,C\,D$



- **Minimierungsregeln**
  - Eintragen von Mintermen (Einsen aus Tabelle und "Don't cares" (*))
  - Markieren von Implikanten
    - Bereiche dürfen nur 1 und * enthalten
    - nur Rechtecke mit 2^k Einträgen
    - Dürfen um Ränder herum reichen
    - Müssen so groß wie möglich sein (Primimplikanten)
  - Ziel: Überdeckung aller Einsen mit möglichst wenigen Primimplikanten

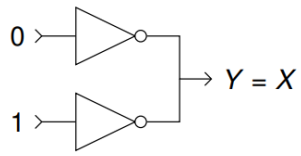


2.4 Minimierung von Ausdrücken

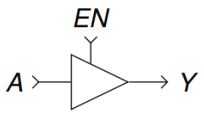
- **Ziel:** Minimiere Anzahl der zur Darstellung einer Funktion notwendigen Implikanten
- **Espresso-Heuristik**
  - Keywords:
    - .i: Anzahl n_i der Eingänge (erforderlich)
    - .o: Anzahl n_o der Ausgänge (erforderlich)
    - .p: Anzahl der Tabellenzeilen

2.5 Vierwertige Logik (0,1,X,Z)

- **Allgemein**
  - Unterscheidung von zwei weiteren Logikwerten neben 0 und 1
  - X mehrfach getrieben: fehlerhaft
  - Z ungetrieben/hochohmig: gezielt (high impedance)
  - Nicht mit "Don't care" verwechseln
- **X (mehrfach getrieben): Konkurrierende Ausgänge**
  - mehrere Treiber für den selben Schaltungsknoten
  - Konflikt, sobald Treiber in entgegengesetzte Richtung ziehen
  - Meist Entwurfsfehler (Doppelzuweisung in Verilog)

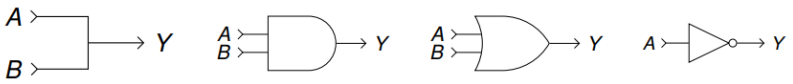


- **Z (ungetrieben/hochohmig): Tristate-Buffer**
  - Zusätzliches Enable-Signal EN für Buffer
  - EN = 1: Funktion wie normaler Buffer
  - EN = 0: Ausgang hochohmig → Z



EN	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

  - Verwendungin Bussen zur Zuschaltung von nur einem Treiber zur selben Zeit
- **Mehrwertige Logik in Schaltnetzen**



A/B	X	0	1	Z
X	X	X	X	X
0	X	0	X	0
1	X	X	1	1
Z	X	0	1	Z

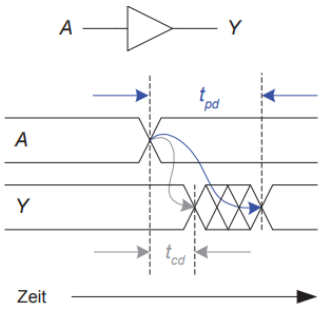
A/B	X	0	1	Z
X	X	0	X	X
0	0	0	0	0
1	X	0	1	X
Z	X	0	X	X

A/B	X	0	1	Z
X	X	X	1	X
0	X	0	1	X
1	1	1	1	1
Z	X	X	1	X

A	Y
X	X
0	1
1	0
Z	X

2.6 Zeitverhalten

- **Allgemein**
  - reale Schaltungselemente benötigen Zeit um Änderung vom Eingang auf Ausgang zu übertragen
  - Eingang kann Ausgang über verschiedene Pfade beeinflussen
  - Führt zu Verzögerungen
- **Verzögerungen**
  - Ausbreitungsverzögerung t_{pd}: Maximale Zeit vom Eingang zum Ausgang
  - Kontaminationsverzlgerung t_{cd}: Minimale Zeit vom Eingang zum Ausgang



  - Kritischer Pfad: Längster Pfad durch Schaltung
  - Kurzer Pfad: Kürzester Pfad durch Schaltung
- **Glitches**
  - eine Änderung des Eingangs verursacht mehrere Änderungen des Ausgangs
  - Können durch geeignete Entwurfsdisziplin entschärft werden
  - Erkennen in Karnaugh-Diagrammen:
    - Nebeneinanderliegende Einsen, die nicht zwingend verbunden werden müssen

    → Überdeckung der Stelle mit zusätzlichem Implikanten

3 Sequentielle Schaltungen

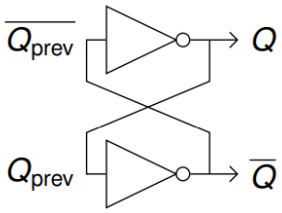
3.1 Allgemein

- Ausgänge hängen von aktuellen und vorherigen Eingabewerten ab
- sequentielle Schaltungen speichern internen Zustand

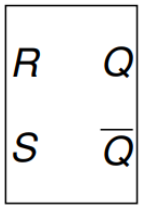
  → realisiert durch Rückkopplungen von Ausgängen
- Zeitverhalten besser kontrollierbar als bei Kombinatorischen

3.2 Latches

- **Bistabile Grundschaltung**

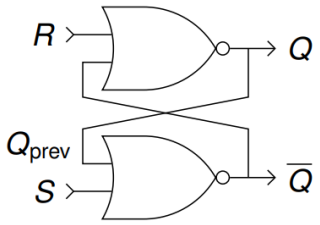


  - Grundlage des Zustandsspeichers
  - zwei Inverter mit Rückkopplung
  - Speichert 1 Bit durch zwei stabile Zustände
  - $Q = 0 \Rightarrow \overline{Q} = 1 \Rightarrow Q = 0$
  - $Q = 1 \Rightarrow \overline{Q} = 0 \Rightarrow Q = 1$
  - Keine Einflüsse auf gespeicherten Zustand
- **SR-Latch**

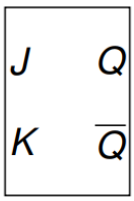


* bistabile Grundschaltung mit NOR statt NOT
* NOR: Ausgang 0 wenn einer der Inputs 1 ist
* $\overline{S}\ \overline{R} \rightarrow$ Zustand halten (latch = verriegeln)
* $\overline{S}\ R \rightarrow$ Zustand 0 rücksetzen (reset R)
* $S\ \overline{R} \rightarrow$ Zustand auf 1 setzen (set S)
* $S\ R \rightarrow$ ungültiger Zustand ($Q = \overline{Q} = 0$)

S	R	Q_{prev}	Q	$\overline{Q}$
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	0	1
1	0	0	1	0
1	0	1	1	0
1	1	0	0	0
1	1	1	0	0

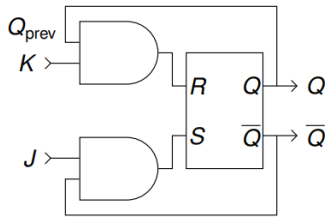


- **JK-Latch**

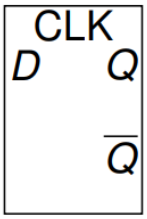


* Ungültigen Zustand SR am SR-Latch verhindern
* $\overline{J}\ \overline{K} \rightarrow$ Zustand halten
* $\overline{J}\ K \rightarrow$ Zustand 0 rücksetzen,falls nötig
* $J\ \overline{K} \rightarrow$ Zustand auf 1 setzen,falls nötig
* $J\ K \rightarrow$ Zustand invertieren

J	K	Q_{prev}	S	R	Q	$\overline{Q}$
0	0	0	0	0	0	1
0	0	1	0	0	1	0
0	1	0	0	0	0	1
0	1	1	0	1	0	1
1	0	0	1	0	1	0
1	0	1	0	0	1	0
1	1	0	1	0	1	0
1	1	1	0	1	0	1

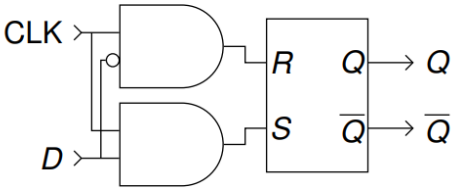


- **D-Latch**



* Daten-Latch mit Taktsignal (CLK) und Dateneingang (D)
* CLK = 1 $\rightarrow$ Zustand auf D setzen (Latch transparent)
* CLK = 0 $\rightarrow$ Zustand halten (Latch nicht transparent)
* ungültiger Zustand am SR-Latch wird vermieden
* Rückkopplung nur noch im SR-Latch

CLK	D	S	R	Q
0	0	0	0	Q_{prev}
0	1	0	0	Q_{prev}
1	0	0	1	0
1	1	1	0	1



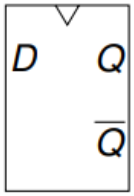
– Probleme des D-Latch:

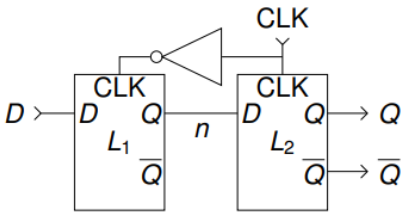
* D-Latch ist taktphasen-gesteuert (Hälfte der Zeit transparent, Hälfte der Zeit kombinatorisch)
* breites "Abtastfenster" sorgt für Unschärfe

* periodische Taktsignale symmetrisch (0-Phase und 1-Phase gleich lang)

3.3 Flip-Flops

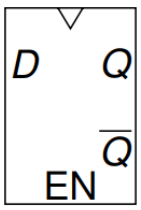
- **D-Flip-Flop**

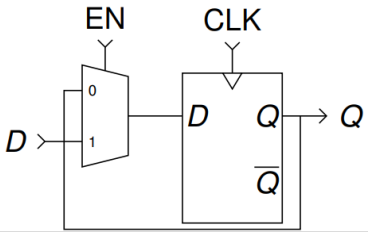




  - Zwei D-Latches in Serie (L_1 = Master | L_2 = Slave | komplementäre Taktsignale)
  - CLK = 0:
    * Master transparent $\rightarrow n = D$
    * Slave nicht transparent $\rightarrow$ Q unverändert
  - CLK = 1:
    * Master nicht transparent $\rightarrow n$ unverändert
    * Slave transparent $\rightarrow Q = n$
  - Taktflanken-gesteuert
    * genau bei steigender CLK Flanke wird $Q = D$
    * Übernahme des Wertes von D, der unmittelbar vor der Taktflanke anliegt

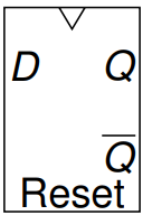
- **Flip-Flops mit Taktfreigabe**

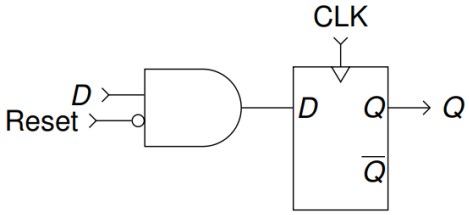




  - Freigabeeingang steuert, wann Daten gespeichert werden
  - $EN = 1 \rightarrow D$ wird bei steigender CLK-Flanke gespeichert
  - $EN = 0 \rightarrow Q$ bleibt auch bei steigender CLK-Flanke unverändert

- **Zurücksetzbare Flip-Flops**





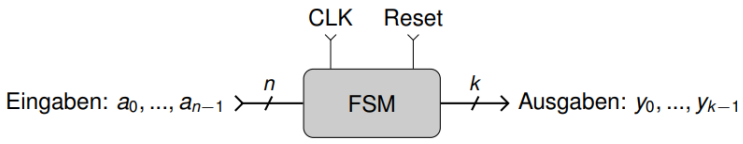
  - Reset setzt internen Zustand unabhängig von D auf 0
  - synchron: nur zur steigenden Taktflanke wirksam
  - asynchron: jederzeit (unabhängig von CLK)

3.4 Entwurf synchroner Schaltungen

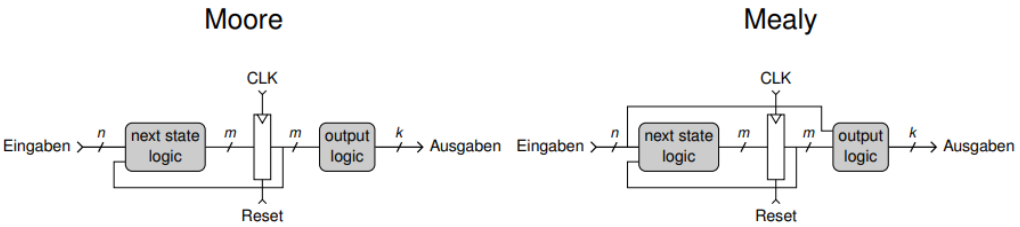
- **Rückkopplungen durch Register aufbrechen**
  - halten den Zustand der Schaltung
  - ändern Zustand nur zur Taktflanke
    → gesamte Schaltung synchronisiert mit Taktflanke
- **Regeln für Aufbau**
  - jedes Schaltungselement ist entweder Register oder kombinatorische Schaltung
  - mindestens ein Schaltungselement ist ein Register
  - alle Register werden durch gleiches Taktsignal gesteuert
  - jeder zyklische Pfad enthält mindestens ein Register
  - z.B.: Anwendung in endlichen Zustandsautomaten

3.5 Endliche Automaten

- **Finite State Machines (FSM)**
  - synchrone sequentielle Schaltungen mit:
    * n Eingabebits | k Ausgabebits
    * ein interner Zustand ($m \geq 1$ Bits) | Takt und Reset
  - in jedem Takt (zur steigenden Flanke):
    * Reset aktiv → Zustand = Startzustand
    * Reset inaktiv → neuen Zustand/Ausgaben aus aktuellem Zustand/Eingaben



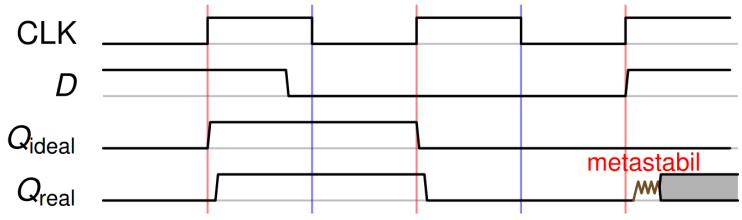
  - **FSM als gerichtete Graphen**
    * Zustände (States) als Knoten (S_0, S_1)
    * Zustandsübergänge (Transitions) als Kanten
      · keine Selbstschleifen
      · Pfade eindeutig
      · leere Bedingung entspricht 1
    * genau eine Kante ohne Startpunkte für Reset
    * Ausgaben
      · An Kanten (Mealy-Automat)
      · in Zuständen (Moore-Automat)
      · als boole'scher Ausdruck (Mintern)
  - **Zustandsübergangs- und Ausgabetabelle**
    * maschinenlesbare Darstellung
    * kann Don't Cares verwenden
    * aktueller Zustand S | nächster Zustand S'
    * implizite Bedingungen (Selbstschleifen) beim Ableiten aus Diagrammen beachten
  - **FSM als synchrone sequentielle Schaltung**
    * Zustandsregister (Speichern von S, Übernahme von S'
    * Zustandsübergangstabelle und Ausgangstabelle mithilfe von kombinatorischer Logik
      → binäre Kodierung der Zustände und Ein-/Ausgaben notwendig



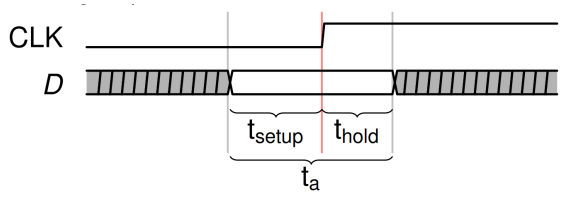
- **Zustandskodierung** $cs : S \to \mathbb{B}^m$
  - weist jedem Zustand einen m Bit breiten Wert zu
  - Freie Wahl, z.B. durchnummerieren $cs(S_k) = (s_{m-1}...s_0)$ mit $u_{2,m}(s_{m-1}...s_0) = k$
  - Effizienz der Kodierung abhängig von Anwendungsfall
  - Kodierung der Ein-/Ausgänge ist i.d.R. von der Anwendung vorgegeben
  - Beispiele für Kodierungen und Automaten auf Folien (Foliensatz 8 oder Übungen)
- **Entwurfsverfahren**
  - Definiere Ein- und Ausgänge
  - Wähle zwischen Moore- und Mealy-Automat
  - Zeichne Zustandsdiagramm
  - Kodiere Zustände (und ggf. Eingänge-/Ausgänge)
  - Stelle Zustandsübergangstabelle und Ausgabetabelle auf
  - Stelle boole'sche Gleichungen für Zustandsübergangs und Ausgangslogik mit Don't Cares auf
  - Entwerfe Schaltplan: Gatter + Register
- **Mealy vs Moore**
  - muss von Fall zu Fall neu bewertet werden
  - Moore besser, wenn Ausgaben statisch
  - Mealy besser, wenn Ausgaben kurzfristige Aktionen auslösen
  - Mealy reagiert schneller auf Änderungen der Eingabe
  - Beispiele zur Vrdeutlichung auf den Folien
- **Zerlegung von Zustandsautomaten**
  - Aufteilen komplexer FSMs in einfachere interagierende FSMs
  - zerlegte FSMs kommunizieren untereinander
  - Ampelbeispiel auf Folien

3.6 Zeitverhalten

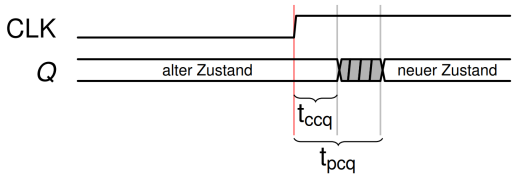
- **Zentrale Fragestellungen**
  - Flip-Flop übernimmt D zur steigenden Taktflanke
  - Was passiert bei zeitgleicher Änderung von D und CLK?
  - Was heißt unmittelbar vor der Taktflanke?
  - Wie schnell wird neuer Zustand am Ausgang sichtbar?
  - Was muss beachtet werden?



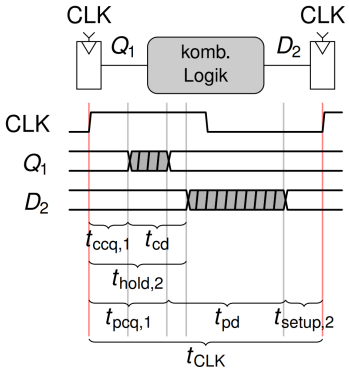
- **Metastabilität**
  - zeitlich begrenzter und undefinierter Zustand
  - geht nach zufälliger Verzögerung in einen stabilen Zustand über
- **Zeitanforderungen an DFF Eingangssignal**
  - Dateneingang D muss vor und nach dem Abtasten stabil sein
    $\rightarrow$ Vermeidung von Metastabilität
  - t_{setup}: Zeitintervall vor Taktflanke, in dem D stabil sein muss
  - t_{hold}: Zeitintervall nach Taktflanke, in dem D stabil sein muss
  - t_a: Abtastzeitfenster : $t_{setup} + t_{hold}$ ("aperture time")



- **Zeitcharakteristik des DFF Ausgangssignals**
  - Verzögerung des Registerausgangs relativ zur steigenden Taktflanke
  - Kontaminationsverzögerung (t_{ccq}): kürzeste Zeit bis Q umschaltet ("contamination delay clock-to-Q")
  - Laufzeitverzögerung (t_{pcq}): längste Zeit bis Q sich stabilisiert ("propagation delay clock-to-Q")

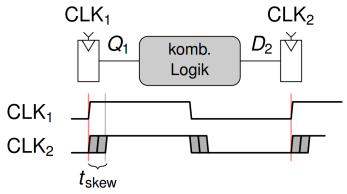


- **Dynamische Entwurfsdisziplin**

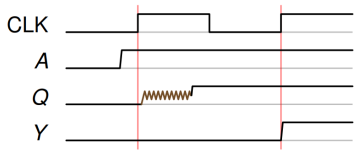


  * D_2 abhängig von Verzögerungen des ersten Registers + Gatter
  * Timing Bedingungen des zweiten Registers müssen erfüllt sein:
    $\rightarrow$ $t_{ccq,1} + t_{cd} \geq t_{hold,2}$
    $\rightarrow$ $t_{pcq,1} + t_{pd} + t_{setup,2} \leq t_{CLK}$
  * Maximale Taktrate wird durch kritischen Pfad bestimmt
    $\rightarrow$ $f_{CLK} = \frac{1}{t_{CLK}} \leq \frac{1}{t_{pcq}+t_{pd}+t_{setup}}$
  * Falls Hold-Bedingung verletzt:
    $\rightarrow$ Einfügen von Buffern auf kürzestem Pfad

- **Taktverschiebung**
  - Takt kommt nicht bei allen Registern gleichzeitig an (Chip-Unterschiede,..)
  - t_{skew} ist maximale Differenz der Taktankunftszeit zwischen zei Registern



- **Timing-Bedingunge mit Takt-Verschiebung**
  - Bedingungen müssen auch im Worst-Case eingehalten werden
    → $t_{ccq,1} + t_{cd} \geq t_{hold,2} + t_{skew}$
    → $t_{pcq,1} + t_{pd} + t_{setup,2} + t_{skew} \leq t_{CLK}$
  - Timing wird dadurch meist enger
- **Verletzung der dynamischen Entwurfsdisziplin**



∗ asynchrone Eingänge
⇒ Eventuelle Nichteinhaltung der Timing-Bedingungen
∗ Schieberegister für Synchronisation
∗ erstes Flip-Flop kann metastabil werden
∗ kippt i.d.R. vor nächster Taktflanke in stabilen Zustand
⇒ zweites Flip-Flop wird nicht metastabil

3.7 Parallelität

- **Arten der Parallelität**
  - räumliche Parallelität:
    → mehrere Aufgaben durch vervielfachte Hardware gleichzeitig bearbeiten
  - zeitliche Parallelität:
    → Aufgaben in mehrere Unteraufgaben aufteilen
    → Unteraufgaben parallel ausführen
- **Begriffe**
  - Datensatz: Vektor aus Eingabewerten → Vektor aus Ausgabewerten
  - Latenz: Zeit von Eingabe des Datensatzes bis zur Ausgabe des Ergebnisses
  - Durchsatz: Anzahl von Durchsätzen pro Zeiteinheit
    → Parallelität erhöht Durchsatz
- **Pipelining**
  - Pipelinestufen sollten möglichst lang sein
    → längste Stufe bestimmt maximale Taktfrequenz f_{CLK}
    → Latenz = Pipelinestufen * Taktfrequenz
  - mehr Pipelinestufen:
    → höherer Durchsatz, da höhere Taktfrequenz
    → aber auch höhere Latenz
    → lohnt sich nur bei vielen Datensätzen

4 Hardware-Beschreibungssprachen

4.1 Allgemein

- **Notwendigkeit und Anwendung**
  - verständliche und einheitliche Beschreibungssprache
  - Benötigt, um steigende Komplexität zu beherrschen
  - Aktuelle Tendenz zu höheren Abstraktionsleveln bei der Entwicklung
- **Von HDL zu Logikgattern**
  - *Simulation* des Verhaltens der beschriebenen Schaltung (Fehlersuche einfacher als in realer Schaltung)
  - *Synthese* übersetzt Hardware-Beschreibungen in Netzliste
  - *Netzliste* beschreibt die Schaltungselemente und Verbindungsknoten

4.2 Kombinatorische Logik (Einführung)

- **SystemVerilog Module**
  - Modul beschreibt, wie eine Aufgabe durchgeführt wird
  - Schnittstellenbeschreibung mithilfe von Eingängen, Ausgängen und Parameter
  - zwei Arten der Beschreibung
    → Struktur (Wie ist die Schaltung aus Modulen aufgebaut?)
    → Verhalten (Was tut die Schaltung?)
- **Modulbeschreibung**
  - Befehle auf Hilfsblatt
  - ∼ NOT & AND | OR
- **Syntax**
  - Case-Sensitive
  - Bezeichner dürfen nicht mit Ziffern anfangen
  - Anzahl von blank space irrelevant
  - Kommentare wie in Java (// und /*...*/)

4.3 Modulhierarchie

- **Modulinstanzierung: (innerhalb eines anderen Moduls)**

```
1  module and3 (input logic a, b, c, output logic y);
2    assign y = a & b & c;
3  endmodule
```

```
1  module inv (input logic a, output logic y);
2    assign y = ~a;
3  endmodule
```

```
1  module nand3 (input logic d, e, f, output logic w);
2    logic s;          //Internes Signal zur Verbindung
3    and3 andgate(d, e, f, s);      //Instanz von and3 namens andgate
4    inv inverter(s, w);
5  endmodule
```

- **Portzuweisung nach Position oder Namen**

```
1   module nand3_named (input logic d, e, f, output logic w);
2     logic s;
3     and3 andgate(.a(d), .b(e), .c(f), .y(s));
4     inv inverter(.a(s), .y(w));
5   endmodule
```

  - 10-100 Ports pro Modul nicht unüblich

    → absolute Portzuweisung per Namen übersichtlicher

- **Bitweise Verknüpfungsoperatoren**

```
1   module gates (input logic [3:0] a, b,
2         output logic [3:0] y1, y2, y3, y4, y5);
3     assign y1 = a & b; // AND
4     assign y2 = a | b; // OR
5     assign y3 = a ^ b; // XOR
6     assign y4 = ~(a & b); // NAND
7     assign y5 = ~(a | b); // NOR
8   endmodule
```

  3:0 → Bitbreite = 4

- **Reduktionsoperationen(unär)**

```
1   module and8 (input logic [7:0] a, output logic y);
2     assign y = &a; // assign y = a[7] & a[6] & ... & a[0];
3   endmodule
```

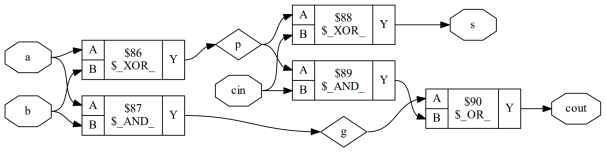
  - Weitere Operationen: | (OR) ∧ (XOR) ~| (NOR) ~& (NAND) ~ ∧ (XNOR)

- **Bedingte Zuweisung**

  - $y = s\ ?\ d1\ :\ d0$;
  - Falls s = 1, dann y = d1, sonst y = d0

- **Interne Verbindungsknoten**

```
1   module fulladder (input logic a, b, cin, output logic s, count);
2     logic p, g; //interne Verbindungsknoten
3     assign p = a ^ b;
4     assign g = a & b;
5     assign s = p ^ cin;
6     assign cout = g | (g & cin);
7   endmodule
```



- **Syntax für numerische Literale**

  - <N>'<B><wert>
  - Bitbreite N, Basis B (d,b,o,h) (default: 32'd)
  - Unterstriche als optische Trenner möglich
  - Beispiele:
    - 8'b11 ⇒ 0000 0011
    - 3'd6 ⇒ 110
    - 42 ⇒ 0000...0101010

- **Konkatenation**

```
1  module concat (input logic [2:0] a, b, output logic [11:0] y);
2   assign y = {a[2:1], {3{b[0]}}, a[0], 6'b100010};
3   // y = a[2] a[1] b[0] b[0] b[0] a[0] 1 0 0 0 1 0;
4  endmodule
```

- **Hochohmiger Ausgang Z**
  - Z darf nur an Ausgängen verwendet werden
  - für interne Signale aber simulierbar
- **Verzögerung: #Zeiteinheiten**
  - Festlegen der Verzögerung vor Modul:
    → 'timescale 1ns / 10 ps (Zeiteinheit / Präzision für Rundung)
  - nach assign #n für Verzögerung um n Zeiteinheiten

4.4 Datentypen

- **Auswahl wichtiger Datentypen**
  - bit = 1'b0, 1'b1 (zweiwertige Logik)
  - logic = 1'b0, 1'b1, 1'bx, 1'by (vierwertige Logik)
  - int = -2**31,...,2**31-1 = bit signed [31:0]
  - integer = -2**31,...,2**31-1 = logic signed [31:0]
  - enum = Aufzählung symbolischer Werte
  - Vektoren und Arrays
- **Vektoren und Arrays**
  - Allgemein:

```
1  // Deklaration
2  logic [7:0] bitVector = 8'hAB; // 8 Bit Vektor, jedes Element 8'hAB [MSB:LSB]
3  logic   bitArray [0:7]   // 8 Bit Array [first:last]
4
5
6  // Zugriffe / Modifikationen
7  initial begin
8   #1 bitVector    = 8'hCD; // alle Vektorenbits ueberschreiben
9   #1 bitVector[5]  = 1'b1;  // Vektorbits einzeln ueberschreiben
10  #1 bitVector[3:0] = 4'hF;  // Vektorbereich ueberschreiben
11
12  // Array-Zugriff nur elementweise moeglich
13  for (int i = 0; i <$size(bitArray); i++} #1 bitArray[i] = bitVector[i];
14 end
```

– Operationen auf Vektoren

```
1
2   module vecop (input logic [3:0] A, input logic [3:0] B,
3       output logic U, V, output logic [3:0] W,
4       output logic [1:0] X, output logic [5:0] Y,
5       output logic [7:0] Z);
6
7     // Reduktion
8     assign U = & A;     // U = A[0] & A[1] & A[2] & A[3]
9
10    // logische Verknuepfung
11    assign V = A && B    // V = (A[0] | A[1] |...) & (B[0] | B[1] |...)
12
13    // bitweise Verknuepfung
14    assign W = A & B // W[0] = (A[0] & B[0]) ,...
15
16    // Konkatenation
17    assign {X,Y} = {A,B}; // X = A[3:2], Y[5:4] = A[1:0], Y[3:0] = B
18
19    // (unsigned) Arithmetrik
20    assign Z = A * B;
```

– Einschränkungen auf Arrays

* nicht als Ports verwendbar
* nicht mit assign verwendbar (kein Part Select (nur einzelne Elemente), keine Zuweisung ganzer Arrays)
* keine Reduktion/Konkatenation
* keine bitweisen/logischen/arithmetischen Operationen

– Speicher als Vektor Arrays

```
1  //     Breite     Tiefe
2  logic [3:0] mem [0:15]; // 16 Worte zu je 4 Bit
3
4  initial begin
5    for (int i = 0; i <$size(mem); i++) #1 mem[i] = i;
6
7    // Zugriff auf Bits in Vektor-Array
8    mem[0][3] = ~mem[0][3];
9  end
```

4.5 Modellierung kombinatorischer Schaltungen

- **assign Statement**

```
1   module example (input logic a, b, c, output logic y);
2    assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;
3   endmodule
```

– auch continuous assignment genannt
– Linke Seite (LHS, left hand side): Variable oder Port
– Rechte Seite (RHS, right hand side): logischer Ausdruck
– Zuweisung, wenn sich der Wert von RHS ändert

- **always-comb Block**

– always_comb <instruction>
→ zum Zeitpunkt 0, nachdem alle initial und always Blöcke gestartet sind
→ wenn sich der Wert von RHS ändert
– LHS Variablen dürfen nicht von anderen Blöcken geschrieben werden

- **Fallunterscheidungen (case)**

```
1   module sevenseg (input logic [3:0] A, output logic [6:0] S);
2    always_comb case (A)
3      0: S = 7'b011_1111;
4      1: S = 7'b000_0110;
5      ...
6      default:  S = 7'b000_0000;
7    endcase
8   endmodule
```

  - case darf nur in always/always_comb Blöcken verwendet werden
  - Alle Eingabeoptionen abdecken (explizit oder über default)

- **Fallunterscheidungen (casez)**

```
1   module priority_encoder (input logic [3:0] A, output logic [3:0] Y);
2    always_comb casez(A) //casez erlaubt Don't cares als ?
3      4'b1???: Y = 4'b1000;
4      4'b01??: Y = 4'b0100;
5      4'b001?: Y = 4'b0010;
6      4'b0001: Y = 4'b0001;
7      default: Y = 4'b0000;
8    endcase
9   endmodule
```

- **Eigenschaften von assign und always_comb**

  - werden immer ausgeführt, wenn sich ein Signal auf der rechten Seite ändert
    → Eigenes Sprachkonstrukt für sequentielle Logik aufgrund interner Zustände
  - Reihenfolge im Quellcode nicht relevant
    → Blockierende Signalzuweisungen (a = b) innerhalb von Blöcken (begin/end) seriell ausgeführt

4.6 Modellierung sequentieller Schaltungen

- **Grundkonzept von always-Blöcken**

  - führt eine Instruktion als Endlosschleife aus
  - durch Klammerung (begin...end) werden Instruktionen zusammengefasst
  - alle always-Blöcke werden parallel ausgeführt
  - ohne explizite Verzögerungsangaben (#n) wird die simulierte Zeit durch Ausführung nicht erhöht

```
1   logic a;
2   always begin
3    a = 1;
4    #1;
5    a = 0;
6    #2;
7   end
8
9   logic b = 0;
10  always #0.5 b = !b;
```

- **Interpretation von Verzögerungszeiten**
  - 'timescale <base> / <precision> (vor Modul spezifiziert)
  - <base> wird mit angegebenem <tval> multipliziert
  - Auch arithmetische Ausdrücke für <tval>

```
1   'timescale 1 ns / 10 ps
2
3   module delay;
4    logic c = 0;
5    always #(1/3.0) c = !c; // 0.33ns
6
7    logic [3:0] d = 0;
8    always #(d + 1) d = d + 1;
9   endmodule
```

- **Warten auf Ereignisse**
  - @ <expr> wartet auf Änderung von kombinatorischen Ausdruck <expr>
  - @ (posedge <expr>) wartet auf steigende Flanke von <expr> (0 → 1, 0 → x,..)
  - @ (nededge <expr>) wartet auf fallende Flanke von <expr> (1 → 0, 1 → x,..)
  - @ (<event1> or <event2>) wartet auf das Eintreten eines der aufgelisteten Ereignisse
    - → or kann auch durch , ersetzt werden
    - → wird auch als Sensitivitätsliste bezeichnet
  - @* wartet auf Änderung eines der im always Block gelesenen Signale
  - Warte-Statements können an beliebiger Stelle im always Block stehen
- **Zuweisungssequenzen in always-Blöcken**
  - blockierende Zuweisungen: <signal> = <expr>;
    - → <expr> wird ausgewertet und an <signal> überweisen, bevor nächste Zuweisung behandelt wird
    - → blockierende Zuweisungen werden sequentiell behandelt
  - Nicht-blockierende Zuweisungen: <signal> <= <expr>;
    - → <expr> wird zwar ausgewertet, aber noch nicht an <signal> überwiesen
    - → Zuweisung an <signal> erfolgt erst bei Fortschreiten der Systemzeit (# oder @)
    - → nicht-blockierende Zuweisungen werden parallel bearbeitet
- **Einmalige und kombinatorische Ausführung**
  - initial <instruction>
    - → entspricht always begin <instruction> @(0); end
    - → für Initialisierung in der Simulation verwenden
  - always_comb <instruction>
    - → verbessert always @* <instruction>
    - → einmalige Ausführung zu Beginn der Simulation, auch ohne Änderung der Eingabesignale
    - → für komplexe, kombinatorische Logik (for, if/else, case, casez) verwenden

- **Modellierung von Speichelementen**
  - D-Latch

```
1  module latch (input logic CLK, D, output logic Q);
2    always_latch if (CLK) Q <= D;
3  endmodule
```

    * always_latch <instruction>
      → für Schaltungen mit Latches
      → entspricht always_comb <instruction>
      → Latches werden kaum benutzt, entstehen meist durch Fehler
  - D-Flip-Flop

```
1  module dff (intput logic CLK, D, output logic Q);
2    always_ff @(posedge CLK) Q <= D;
3  endmodule
```

    * always_ff <instruction>
      · für Schaltungen mit Flip-Flops
      · entspricht always <instruction>
      · vergleichbare Verbesserungen wie bei always_comb
  - Asynchron rücksetzbarer D-Flip-Flop

```
1  module dffar (input logic CLK, RST, D, output logic Q);
2    always_ff @(posedge CLK, posedge RST)
3      if (RST) Q <= 0;
4      else   Q <= D;
5  endmodule
```

  - Synchron rücksetzbarer D-Flip-Flop

```
1  module dffr (input logic CLK, RST, D, output logic Q);
2    always_ff @(posedge CLK)
3      if (RST) Q <= 0;
4      else   Q <= D;
5  endmodule
```

  - D-Flip-Flop mit Taktfreigabe

```
1  module dffe (input logic CLK, RST, EN, D, output logic Q);
2    always_ff @(posedge CLK)
3      if   (RST) Q <= D;
4      else if  (EN) Q <= D;
5  endmodule
```

- **Allgemeine Regeln für Signalzuweisungen (synchrone sequentielle Schaltungen)**
  - interne Zustände
    → innerhalb von always_ff @(posedge CLK)
    → mit nicht-blockierenden Zuweisungen (<=)
    → möglichst ein/wenige Zustände pro always_ff Block
  - einfache kombinatorische Logik durch nebenläufige Zuweisungen (assign)
  - komplexere kombinatorische Logik
    → innerhalb von always_comb
    → mit blockierenden Zuweisungen (=)
  - ein Signal darf NICHT
    → von mehreren nebenläufigen Prozessen (assign/always) beschrieben werden
    → innerhalb eines always-Blocks mit block. & nicht-block. Zuweisungen beschrieben werden

4.7 Parametrisierte Module

- **Parametrisierte Module**
  - Definieren von Parametern durch Modulschnittstelle
  - parametrisierte Eigenschaften werden bei Instanziierung durch konkrete Werte ersetzt
  - Vergleichbar mit Java-Generics
  - Typische Parameter: Port-Breite, Speichertiefe, Anzahl der Submodule,..

```
1   module register #(parameter WIDTH = 8, parameter DEPTH = 32) (input logic ...)
```

4.8 Testrahmen

- **Testumgebungen (testbenches**
  - HDL-Programm zum Testen eines HDL-Moduls
  - getestes Modul (Device under test DUT, Unit under test UUT)
  - Arten von Testrahmen:
    → einfach: Testdaten an UUT anlegen und Ausgaben anzeigen
    → selbstprüfend: Ausgaben zusätzlich auf Korrektheit prüfen
    → selbstprüfend mit Testvektoren: variable Testdaten (z.B. aus Datei)

- **Einfacher Testrahmen**

```
1   module simble_tb;
2    logic a, b, c, y;
3    simple uut(a, b, c, y);
4
5    initial begin
6     //dump changes of all variables to this file
7     $dumpfile("simple_tb.vcd");
8     $dumpvars;
9
10    a = 0; b = 0; c = 0; #10;
11         c = 1; #10;
12        b = 1; c = 0; #10;
13                c = 1; #10;
14
15    $display("FINISHED simple_tb"); // Textausgabe
16    $finish;        // beendet Simulation
17   end
18  endmodule
```

- **Selbstprüfender Testrahmen**

```
1   module simble_tb2;
2    logic a, b, c, y;
3    simple uut(a, b, c, y);
4
5    initial begin
6     //dump changes of all variables to this file
7     $dumpfile("simple_tb2.vcd");
8     $dumpvars;
9
10    a = 0; b = 0; c = 0; #10; assert (y === 1) else $error("000 failed.");
11         c = 1; #10; assert (y === 0) else $error("001 failed.");
12        b = 1; c = 0; #10; assert (y === 0) else $error("010 failed.");
13                c = 1; #10; assert (y === 0) else $error("011 failed.");
14
15    $display("FINISHED simple_tb"); // Textausgabe
16    $finish;        // beendet Simulation
17   end
18  endmodule
```

- **Vorhergehensweise**
  - Modul ohne Ports
  - Stimuli erzeugen (Takt, Reset, Eingabedaten)
  - ”unit under test” instanziieren
  - Ausgabedaten und Timing spezifizieren (erschöpfend/zufällig, Grenzfälle bedenken)
  - wird nicht synthetisiert
- **Ausgabe von Statusmeldungen**
  - $display(<format>, <values>*);
  - Platzhalter:
    → %d %b %h für dezimal, binär und hexadezimal
    → %m für Modulname
    → %t für Zeit (mit Einheit)
  - $timeformat(-9, 1, ”ns”, 8); zum Erstellen des Zeitformats
    → Skalierung auf 10^{-9} | Eine Nachkommastelle
    → ”ns” als Einheiten-Suffix | 8 anzuzeigende Zeichen
- **Auslesen der Simulationszeit**
  - $time: aktuelle Systemzeit als ganze Zahl (int)
  - $realtime: aktuelle Systemzeit als rationale Zahl (real)
  - Zeitspanne zwischen zwei Signalflanken bestimmen:

```
1   'timescale 1 ns / 10 ps
2   module deltat;
3    logic a = 0; always #3 a <= ~a;
4    logic b = 0; always #2 b <= ~b;
5
6    real aEvent; always @a aEvent <= $realtime;
7    real delay;  always @b delay  <= $realtime - aEvent;
8   endmodule
```

4.9 Modellierung endlicher Automaten

- **Grundidee für FSM-Modellierung**
  - Logikvektor oder enum für Zustände
  - rücksetzbare Flip-Flops als Zustandsspeicher
  - kombinatorische next-state Logik durch case in always_comb Block
  - kombinatorische Ausgabe-Logik durch nebenläufige Zuweisungen
- **Moore Automat für 1101 Mustererkennung**

```
1   module moore (input logic CLK, RST, A, output logic Y);
2    typedef enum logic [2:0] {S0, S1, S2, S3, S4} statetype;
3    statetype state, nextstate;
4    always_ff @(posedge CLK) state <= RST ? S0 : nextstate;
5
6    // next state logic
7    always_comb case (state)
8     S0:   nextstate = A ? S1 : S0;
9     S1:   nextstate = A ? S2 : S0;
10    S2:   nextstate = A ? S2 : S3;
11    S3:   nextstate = A ? S4 : S0;
12    S4:   nextstate = A ? S2 : S0;
13    default: nextstate = S0;
14   endcase
15
16   //output logic
17   assign Y = (state == S4);
18  endmodule
```

- **Mealy Automat für 1101 Mustererkennung**

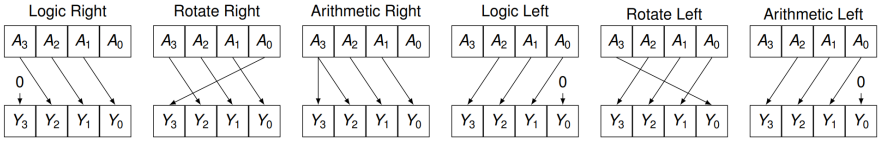
```
1   module moore (input logic CLK, RST, A, output logic Y);
2    typedef enum logic [1:0] {S0, S1, S2, S3} statetype;
3    statetype state, nextstate;
4    always_ff @(posedge CLK) state <= RST ? S0 : nextstate;
5
6    // next state logic
7    always_comb case (state)
8     S0:   nextstate = A ? S1 : S0;
9     S1:   nextstate = A ? S2 : S0;
10    S2:   nextstate = A ? S2 : S3;
11    S3:   nextstate = A ? S1 : S0;
12    default: nextstate = S0;
13   endcase
14
15   //output logic
16   assign Y = (state == S3 && A);
17  endmodule
```

- **Simulation vs Synthese**
  - alle SystemVerilog Konstrukte sind grundsätzlich simulierbar
  - nicht synthetisierbar sind (mit realer Hardware umsetzbar)
    - → Signalinitialisierung bei der Deklaration
    - → initial Blöcke
    - → explizite Verzögerungen
    - → die meisten Funktionen ($time, $display,..)

5 Grundelemente digitaler Schaltungen

5.1 Arithmetische Schaltungen

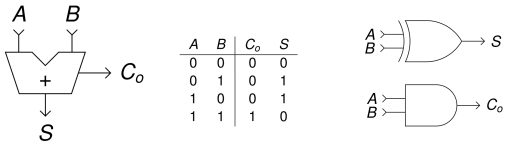
- **Shifter**
  - A um B Stellen nach links/rechts verschieben
  - Strategien zum Auffüllen der freien Stellen (B = 1):
    - → logischer Rechts-/ Linksshift: Auffüllen mit Nullen
    - → umlaufender Rechts-/Linksshift: Auffüllen mit herausfallenden Bits (Rotation)
    - → arithmetischer Rechtsshift: Auffüllen mit MSB (Division durch 2^B)
    - → arithmetischer Linksshift: Auffüllen mit Nullen (Multiplikation mit 2^B)



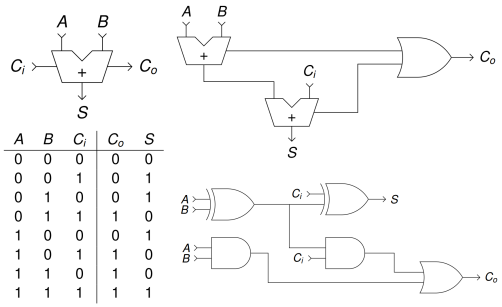
- **Arithmetische Shifter als Multiplizierer und Dividierer**
  - Arithmetischer Linkshift um n Stellen multipliziert den Zahlenwert um 2^n
    $\rightarrow\ 00001_2 <<< 3 = 01000_2 = 1 * 2^3 = 8$
    $\rightarrow\ 11101_2 <<< 2 = 10100_2 = -3 * 2^2 = -12$

  $\Rightarrow$ Multiplikation mit Konstanten kann zusammengesetzt werden
    $\rightarrow\ a * 6 = a * 110_2 = (a <<< 2) + (a <<< 1)$
  - Arithmetischer Rechtsshift um n Stellen dividiert den Zahlenwert um 2^n
    $\rightarrow\ 010000_2 >>> 4 = 000001_2 = \frac{16}{2^4} = 1$
    $\rightarrow\ 100000_2 >>> 2 = 111000_2 = \frac{-32}{2^2} = -8$

- **Halbaddierer**



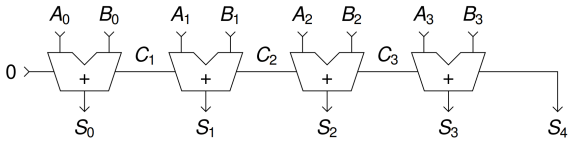
- **Volladdierer**



- **Ripple-Carry-Adder (RCA)**
  - Überträge werden über Kette von 1 Bit Volladdiereren vom LSB zum MSB weitergegeben

  $\Rightarrow$ langer kritischer Pfad

  $\Rightarrow$ schnelle Addierer müssen Übertragskette aufbrechen

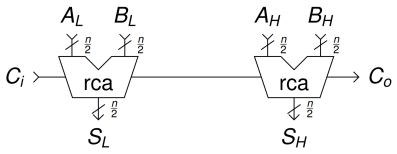


- Rekursiver Aufbau:

  $\rightarrow$ Aufteilen in unteres (**L**) und oberes Halbwort (**H**)

  $\rightarrow$ zweiter Addierer wartet auf Übertrag aus erstem Addierer

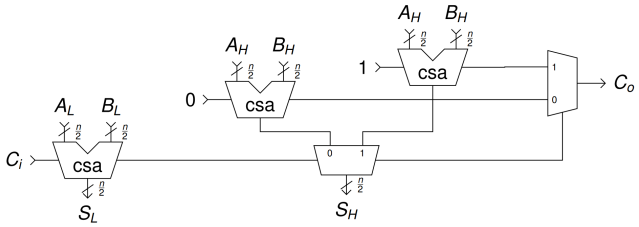
  $\rightarrow$ Bearbeiten beider Teilwörter gleichzeitig für schnellen Addierer



- **Conditional Sum Adder (CSA)**
  - Übertrag vom unteren in oberes Halbwort kann nur 0 oder 1 sein
  - Berechnung des oberen Halbworts für beide Optionen

  ⇒ Auswahl des richtiges Ergebnisses sobald Übertrag bekannt ist

  ⇒ Kritischer Pfad nur noch halber CSA + MUX



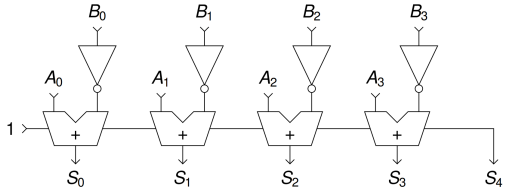
- **Carry Lookahead Adder (CLA)**
  - Motivation
    - für A_i B_i ist $C_i = 1$ unabhängig von C_{i-1}
      → Spalte i generiert Übertrag (generate)
    - für $A_i + B_i = 1$ ist $C_i = 1$ wenn $C_{i-1} = 1$
      → Spalte i leitet Übertrag weiter (propagate)
    - für $A_i + B_i = 0$ ist $C_i = 0$ unabhängig von C_{i-1}
      → Spalte i leitet Übertrag nicht weiter
  - Generate und Propagate pro Spalte
    - Generate-Flag Spalte i: $G_i = A_i\ B_i$
    - Propagate-Flag Spalte i: $P_i = A_1 + B_i$

    ⇒ Übertrag aus Spalte i: $C_i = G_i + P_i\ C_{i-1}$
    - Bei naiver Verwendung kein Vorteil gegenüber Volladdierer (selber kritischer Pfad)
  - Generate und Propagate über mehrere Spalten
    - Kombinierung über mehrere Spalten (hier für $k = 4$)
    - k-Spalten propagiert Übertrag, wenn jede Spalte propagiert
      → $P_{3:0} = P3\ P2\ P1\ P0$
    - k-Spaltenblock generiert Übertrag, wenn eine Spalte generiert und alle anderen propagieren
      → $G_{3:0} = G_3 + P_3\ G_2 + P_3\ P_2\ G_1 + P_3\ P_2\ P_2\ G_0$
    - Übertrag überspringt k-Spalten auf einmal:

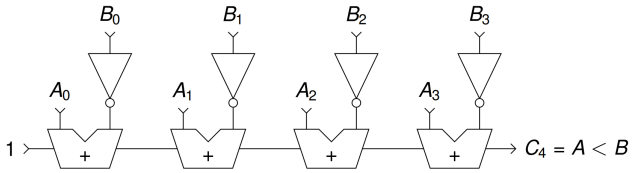
$$C_n = G_{n:n-k+1} + C_{n-k} * Pn : n-k+1$$
$$= (G_n + P_n(G_{n-1} + ... + (P_{n-k+2}\ G_{n-k+1}))) + C_{n-k} * \prod_{i=n-k+1}^{n} P_i$$

  - Kritischer Pfad
    - Propagate und Generate Signale können in allen Blöcken gleichzeitig berechnet werden
    - für große Bitbreiten N dominiert $\frac{N}{k} * (t_{pd,AND} + t_{pd,OR})$
      → Blöcke möglichst groß wählen (ressourcenlastiger)
    - CLA bereits ab $N = 8$Bit schneller als RCA

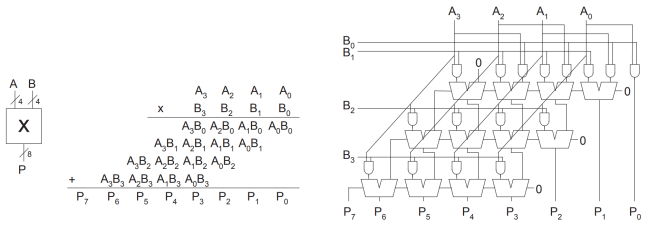
- **Subtrahierer**
  - kann mit Addition und Negation realisiert werden
    → $A - B = A + (-B)$
    → RCA mit NOT-Gatter an B Eingängen und $C_0 = 1$



- **Vergleich kleiner als**
  - kann mit Subtraktion realisiert werden
    $\rightarrow A < B \Leftrightarrow A - B < 0$

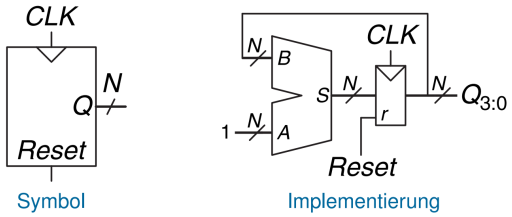


- **Gleichheit**
  - Bitweise (A0 & B0, A1 & B1,..) erst XNOR (Gleichheit, 1 falls alle Inputs gleich) danach nur ANDs
- **Multiplizierer**
  - Produkt von n und m Bit breiten Faktoren ist $n + m$ Bit breit
  - Teilprodukte aus einzelnen Ziffern des Multiplikators mit dem Multiplikanden
  - Addieren der verschiedenen Teilprodukte
  - Kombinatorische 4x4 Multiplikation:

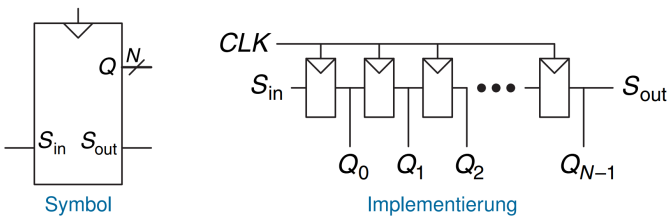


5.2 Sequentielle Grundelemente

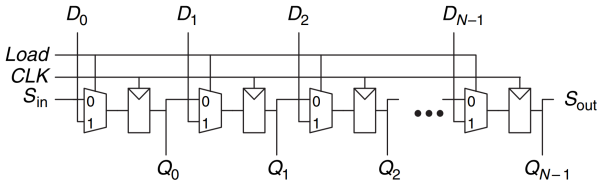
- **Zähler**
  - Erhöht sich bei jeder steigenden Taktflanke
  - Dient zum Durchlaufen von Zahlen (000,001,..)



- **Schieberegister**
  - Bei jeder steigenden Taktflanke → Weiterschieben des Inhalts um einen Flip-Flop
    - FIFO-Prinzip (First In, First Out)
    - Neues Bit S_{in} wird eingelesen
    - Letztes Bit S_{out} wird nach außen geschoben verschoben/verworfen
  - Seriell-Parallel-Wandler:
    → Wandelt den seriellen Eingang (S_{in}) in den parallelen Ausgang ($Q_{0:N-1}$) um

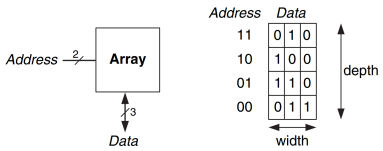


- **Schieberegister mit parallelem Laden**
  - Für $Load = 1$: normales N-Bit Register
  - Für $Load = 0$: Schieberegister
  - Kann dadurch als Seriell-Parallel-Wandler (S_{in} zu $Q_{0:N-1}$, $Load = 0$) oder
  - als Parallel-Seriell-Wandler ($D_{0:N-1}$ zu S_{out}, $Load = 1$) fungieren

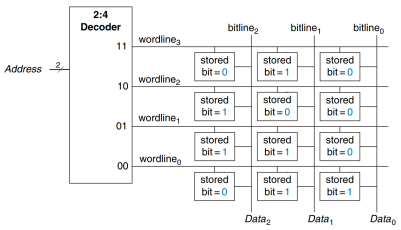


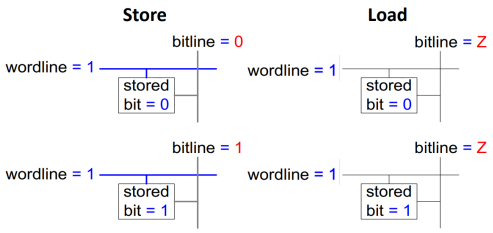
5.3 Speicherfelder

- **Speicherfeld**
  - 2-dimensionales Array von Bitzellen
  - Jede Bitzelle speichert ein Bit
  - N Adressbits und M Datenbits
    - → 2^N Zeilen und M Spalten
    - → Tiefe: Anzahl der Zeilen(Wörter)
    - → Breite: Anzahl der Spalten (Wortbreite)
    - → Größe: Tiefe x Breite = 2^N x M

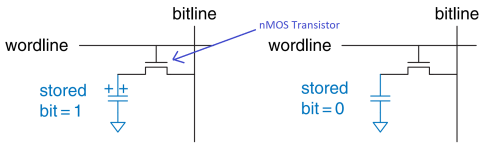


  - Wordline:
    - Vergleichbar zu ENABLE Signal
    - Einzelne Zeile wird gelesen/geschrieben
    - Entspricht eindeutiger Adresse

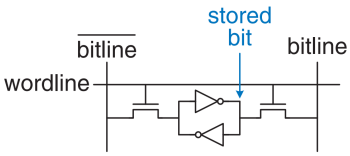




- **RAM**
  - Allgemein:
    - Direktzugriffsspeicher (random access memory, RAM)
    - Flüchtig: Verliert Daten beim Ausschalten
    - Schnelles Lesen und Schreiben
  - DRAM:
    - dynamic random access memory
    - Datenbits werden in Kondensator gespeichert
    - Dynamisch, da Wert regelmäßig und nach Lesen neu geschrieben werden muss
      → Ladungsverlust des Kondensators verschlecht Wert mit der Zeit
      → Lesen zerstört gespeicherten Wert

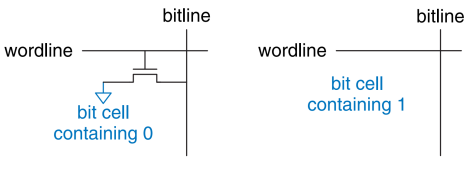


  - SRAM:
    - static random access memory
    - verwendet Inverter mit Rückkopplung zur Datenspeicherung

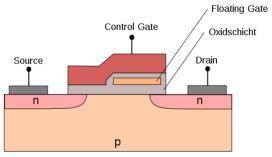


- **ROM**
  - Allgemein:
    - Festwertspeicher (read-only memory, ROM)
    - Nicht flüchtig: Daten bleiben beim Ausschalten erhalten
    - schnelles Lesen, aber Schreiben unmöglich oder langsam
    - Flash-Speicher ist ROM (allerdings sind diese mittlerweile schreibbar)
  - ROM-Punktnotation

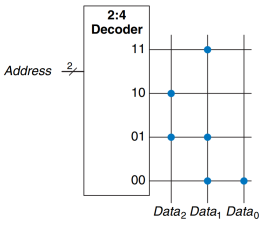
  - Lesen:
    - Bitline auf weak high setzen und danach wordline auf 1 setzen
    - Wenn Transistor vorhanden, zieht dieser bitline auf 0, sonst bleibt diese bei 1



- Flash:
  * Floating Gate kann durch das Anlegen hoher Spannung geladen/entladen werden

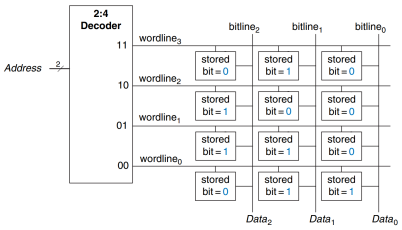


- Logik via ROM:



$Data_2 = A_1 \oplus A_0$
$Data_1 = \overline{A_1} + A_0$
$Data_0 = A_1 \ \overline{A_0}$

- Logik via Speicherfeld:



$Data_2 = A_1 \oplus A_0$
$Data_1 = \overline{A_1} + A_0$
$Data_0 = A_1 \ \overline{A_0}$

- **SystemVerilog**
  - RAM:

```
1   module ram #(parameter N=6, M=32)
2    (input logic clk, input logic we, input logic [N-1:0] adr,
3    (input logic [M-1:0] din, output logic [M-1:0] dout);
4
5    logic [M-1:0] mem [2**N-1:0];
6
7    // write
8    always_ff @(posedge clk)
9     if (we)
10     mem [adr] <= din;
11
12   // read
13   assign dout = mem[adr];
14  endmodule
```

  - ROM:

```
1   module rom (input logic [1:0] adr, output logic [2:0] dout);
2
3    always_comb
4     case (adr)
5      2'b00: dout = 3'b011;
6      2'b01: dout = 3'b110;
7      2'b10: dout = 3'b100;
8      2'b11: dout = 3'b010;
9     endcase
10  endmodule
```

5.4 Logikfelder

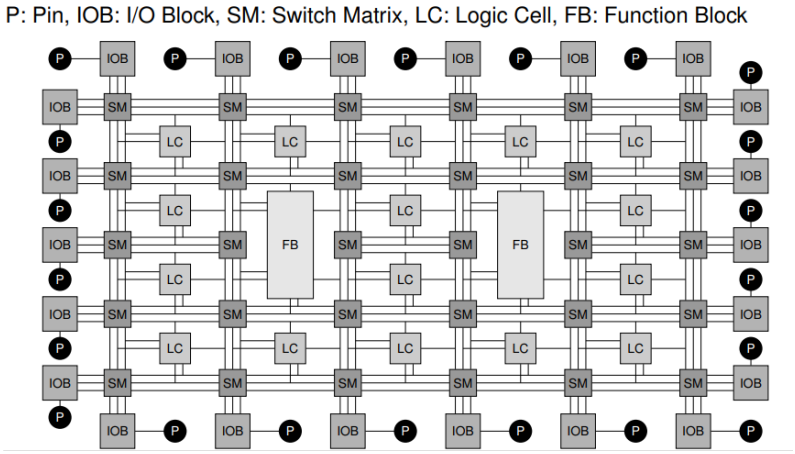
- **Programmierbares Logikfeld (Progamable Logic Array PLA)**

  - realisiert einfache kombinatorische Logik via Sum-Of-Products Form (DNF)
  - zweistufige Logik mit programmierbaren Schaltern in Eingabefeld (links) und Ausgabefeld (rechts)

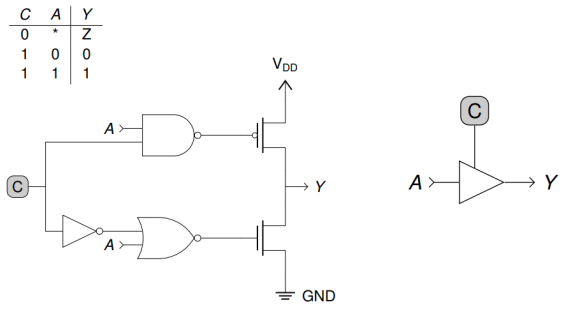
- **Performanz vs Flexibilität**
  - Anwendungsspezifische integrierte Schaltung (ASIC)
    * führt für eine Anwendung optimierte (parallele) Datenpfade aus
    * Basisgatterschaltungen durch optisch/chemische Prozesse auf Silikon-Wafer realisiert
      → zur Laufzeit nicht an neue Anwendung anpassbar
  - Software-Prozessor
    * führt generische Instruktionen sequentiell aus
    * nur generische Architektur in Hardware realisiert
      → zur Laufzeit durch Austausch der Sequenz anpassbar

  ⇒ Field Programmable Gate Array (FPGA) vereint:
    * Flexibilität von Software-Prozessoren
    * mit Performanz von ASICs

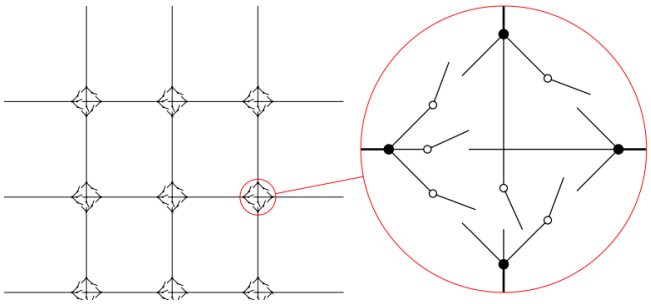
- **FPGA Konfigurationsspeicher**
  - Verwenden feingranulare (bitweise) Konfigurationsspeicher statt wortweisen Instruktionsspeichern
  - kann mit verschiedenen Speicher-Technologien realisiert werden:
    * volatil (SRAM): schnell beschreibbar, benötigt permanente Stromversorgung
    * nicht-volatil (Flash): aufwendiger Schreibzugriff,Zustand bleibt auch ohne Strom erhalten



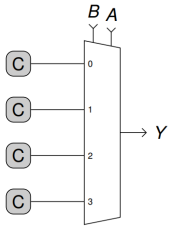
- **Programmierbare Schalter**



- **Programmierbare Leitungskreuzungen (Switch Matrix / SM)**

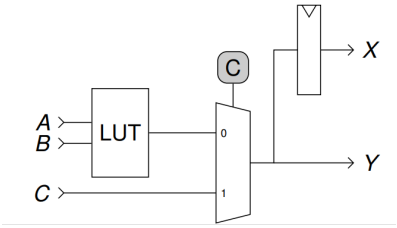


- **Programmierbare Tabellen (Lookup Table / LUT)**

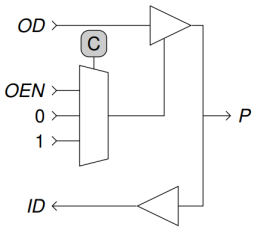


  - realisiert kombinatorische Logik
  - 2 bis 6 Eingänge

- **Programmierbare Logikzelle (Logic Cell / LC)**
  - kann als kombinatorische Logik (Y) und/oder Speicher(X) verwendet werden
  - häufig auch spezielle Carry In/Out (C) für schnelle Arithmetik



- **Programmierbare Ein-/Ausgänge (Input-/Output Blocks / IOB)**



  - Ausgabetreiber permanent oder zur Laufzeit (OEN) deaktivierbar
  - P wird mit physikalischen Pins verbunden

- **Funktionsblöcke (FB)**
  - häufig verwendete Logikbausteine als begrenzte Ressourcen verfügbar
  - Block RAM (BRAM): kleine SRAM Speicher
  - Phase-Locked Loop (PLL): Taktmodifikation
  - ...

Digitaltechnik Wintersemester 2019/2020 Hilfsblatt

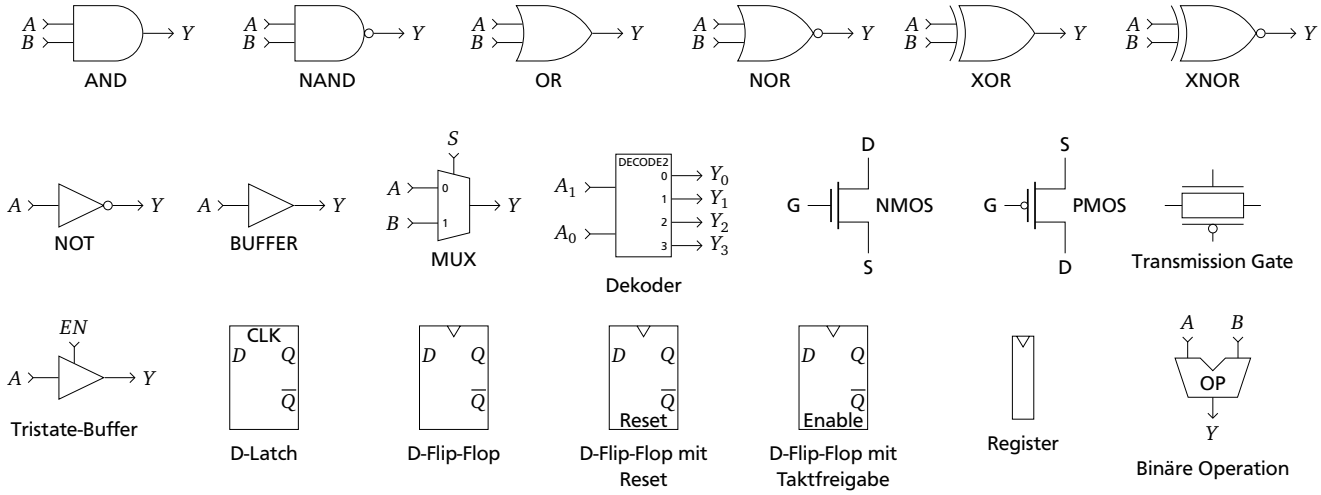
TECHNISCHE UNIVERSITÄT DARMSTADT

Prof. Dr.-Ing. Thomas Schneider, M.Sc. Christian Weinert

Einheitenvorsätze

Bezeichnung	Kürzel	Wert	Bezeichnung	Kürzel	Wert	Bezeichnung	Kürzel	Wert
Milli	m	10^{-3}	Kilo	k	10^{3}	Kibi	Ki	2^{10}
Mikro	μ	10^{-6}	Mega	M	10^{6}	Mebi	Mi	2^{20}
Nano	n	10^{-9}	Giga	G	10^{9}	Gibi	Gi	2^{30}
Piko	p	10^{-12}	Tera	T	10^{12}	Tebi	Ti	2^{40}

Schaltsymbole



Axiome und Theoreme der boole'schen Algebra

	Axiom	Dual	Bedeutung
A1	$B \neq 1 \Rightarrow B = 0$	A1'	$B \neq 0 \Rightarrow B = 1$	Dualität
A2	$\overline{0} = 1$	A2'	$\overline{1} = 0$	Negieren
A3	$0 \cdot 0 = 0$	A3'	$1 + 1 = 1$	Und / Oder
A4	$1 \cdot 1 = 1$	A4'	$0 + 0 = 0$	Und / Oder
A5	$0 \cdot 1 = 1 \cdot 0 = 0$	A5'	$1 + 0 = 0 + 1 = 1$	Und / Oder

	Theorem	Dual	Bedeutung
T1	$A \cdot 1 = A$	T1'	$A + 0 = A$	Neutralität
T2	$A \cdot 0 = 0$	T2'	$A + 1 = 1$	Extremum
T3	$A \cdot A = A$	T3'	$A + A = A$	Idempotenz
T4	$\overline{\overline{A}} = A$			Involution
T5	$A \cdot \overline{A} = 0$	T5'	$A + \overline{A} = 1$	Komplement

	Theorem	Dual	Bedeutung
T6	$A\,B = B\,A$	T6'	$A + B = B + A$	Kommutativität
T7	$A\,(B\,C) = (A\,B)\,C$	T7'	$A + (B + C) = (A + B) + C$	Assoziativität
T8	$A\,(B + C) = (A\,B) + (A\,C)$	T8'	$A + (B\,C) = (A + B)\,(A + C)$	Distributivität
T9	$A\,(A + B) = A$	T9'	$A + (A\,B) = A$	Absorption
T10	$(A\,B) + (A\,\overline{B}) = A$	T10'	$(A + B)\,(A + \overline{B}) = A$	Zusammenfassen
T11	$(A\,B) + (\overline{A}\,C) + (B\,C) = (A\,B) + (\overline{A}\,C)$	T11'	$(A + B)\,(\overline{A} + C)\,(B + C) = (A + B)\,(\overline{A} + C)$	Konsensus
T12	$\overline{A\,B\,C\ldots} = \overline{A} + \overline{B} + \overline{C}\ldots$	T12'	$\overline{A + B + C\ldots} = \overline{A}\,\overline{B}\,\overline{C}\ldots$	De Morgan

SystemVerilog Syntax (Auszug)

Modul Deklaration

```
module modul_ID
 #(parameter param_ID = wert)
  (input  datentyp /*[n:m]*/ in_port_ID,
   output datentyp /*[n:m]*/ out_port_ID);

  // lokale Signale
  datentyp /*[n:m]*/ signal_ID /*[k:l]*/;

  // parallele Anweisungen
  assign /* #delay */ signal = ausdruck;
  always sequentielle_anweisung
  submodule #(parameter_map) instanz (port_map);

  // generische Anweisungen
  genvar id;
  generate
    if  (bedingung) begin
      // lokale Signale, parallele Anweisungen
    end
    for (init; cond; step) begin
      // lokale Signale, parallele Anweisungen
    end
  endgenerate
endmodule
```

Sequentielle Anweisungen

```
// Zuweisung
signal  = ausdruck; // blockierend
signal <= ausdruck; // nicht-blockierend

// verzögerte Anweisungen
#delay anweisung
@(ausdruck) anweisung
@(posedge ausdruck) anweisung
@(negedge ausdruck) anweisung
@* anweisung

// bedingte Anweisungen
if (bedingung) anweisung1 else anweisung2
case (ausdruck)
  wert1  : anweisung1
  wert2  : anweisung2
  default: anweisung3
endcase

// wiederholte Anweisung
for (init; cond; step) anweisung

// kombinierte Anweisung
begin anweisung1 anweisung2 ... end
```

Vertikale Gruppierung nach Präzedenz, beginnend mit der höchsten

Operator	Bedeutung
[]	Zugriff auf Vektorelement
~	bitweise NOT
!	logisches NOT
-	unäre Negation
&	unäre Reduktion mit AND
|	unäre Reduktion mit OR
^	unäre Reduktion mit XOR
~&	unäre Reduktion mit NAND
~|	unäre Reduktion mit NOR
~^	unäre Reduktion mit XNOR
**	Exponentialfunktion
*	Multiplikation
/	Division
%	Modulo
+ -	Addition, Subtraktion
<< >>	logischer Shift
<<< >>>	arithmetischer Shift
<	kleiner als
<=	kleiner oder gleich
>	größer als
>=	größer oder gleich
==	gleich
!=	ungleich
===	bitweise gleich
!==	bitweise ungleich
& ~&	bitweise AND, NAND
^ ~^	bitweise XOR, XNOR
| ~|	bitweise OR, NOR
&&	logisches AND
||	logisches OR
?:	ternärer Operator
{}	Konkatenation

Numerische Literale

```
// Bitbreite 'Basis Ziffernfolge
64'h0123456789abcd // hexadezimal
27'd0123456789     // dezimal
24'o01234567       // oktal
 4'bxz01           // binär (vierwertig)
   // x - unbekannt/ungültig
   // z - hochomig
```

Elementare Datentypen

```
bit     // zweiwertige Logik
logic   // vierwertige Logik
byte    // 8 bit signed
integer // 32 bit signed
longint // 64 bit signed
time    // 64 bit signed for Zeitwerte
real    // Gleitkomma-Werte
```

System Funktionen

```
// Basis und Genauigkeit der Simulationszeit setzen
'timescale base / precision;
$time           // aktuelle Systemzeit (als int)
$realtime       // aktuelle Systemzeit (als real)

$clog2(num)     // Logarithmus zur Basis 2
$dumpfile(pfad);// VCD Ausgabedatei setzen
$dumpvars;      // (alle) Signale beobachten
$finish;        // Simulation beenden
$display(format, ausdrücke); // Meldung ausgeben
      // %b binary format
      // %c ASCII character format
      // %d decimal format
      // %h hex format
      // %o octal format
      // %s string format
      // %t time format
```

7 Nützliches

7.1 Links

- Interaktiver Moodles Kurs mit Lerneinheiten und Übungsaufgaben (Englisch)
- https://moodle.informatik.tu-darmstadt.de/enrol/index.php?id=757

- Binäre Addition Übungen: Nabla -> Aufgabenkatalog -> Rechnerarchitektur
- https://nabla.algo.informatik.tu-darmstadt.de/