

## Schaltalgebra:

**Operatoren:** UND  $\wedge$ , ODER  $\vee$ , Nicht  $\bar{A}$

**Vorrangregeln:** Negation  $\rightarrow$  Konjunktion (UND)  $\rightarrow$  Disjunktion (ODER)

**Kommutativgesetz:** Vertauschungsgesetz

$$A \wedge B \wedge C = C \wedge B \wedge A \rightarrow \text{UND}$$

$$A \vee B \vee C = C \vee B \vee A \rightarrow \text{ODER}$$

$$A \vee B \wedge C = A \vee (B \wedge C) = A \vee (C \wedge B) = A \vee C \wedge B \rightarrow \text{Gemischt}$$

**Assoziativgesetz:** Klammergesetz

$$A \wedge B \wedge C = (A \wedge B) \wedge C \rightarrow \text{UND}$$

$$A \vee B \vee C = (A \vee B) \vee C \rightarrow \text{ODER}$$

**Distributivgesetz:** Verteilungsgesetz

$$A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$$

$$A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$$

Für KV letzter Schritt

**Idempotenzgesetz:**

$$A \wedge A = A$$

$$A \vee A = A$$

**Neutralitätsgesetz:**

$$A \wedge 1 = A$$

$$A \vee 0 = A$$

**Extremalgesetz:**

$$A \wedge 0 = 0$$

$$A \vee 1 = 1$$

**Doppelnegationsgesetz:**

$$A = \bar{\bar{A}}$$

**De Morgansche Gesetz:**

$$\overline{A \wedge B} = \bar{A} \vee \bar{B} \rightarrow \text{UND}$$

$$\overline{A \vee B} = \bar{A} \wedge \bar{B} \rightarrow \text{ODER}$$

**Komplementärsgesetz:**

$$A \wedge \bar{A} = 0 \rightarrow \text{UND}$$

$$A \vee \bar{A} = 1 \rightarrow \text{ODER}$$

**Dualitätsgesetz:**

$$\bar{0} = 1$$

$$\bar{1} = 0$$

**Absorptionsgesetz:**

$$A \vee (A \wedge B) = A \rightarrow \text{UND}$$

$$A \wedge (A \vee B) = A \rightarrow \text{ODER}$$

**Weitere Gesetze:**

$$A \wedge 1 = A \rightarrow \text{UND } 1$$

$$A \wedge 0 = 0 \rightarrow \text{UND } 0$$

$$A \vee 1 = 1 \rightarrow \text{ODER } 1$$

$$A \vee 0 = A \rightarrow \text{ODER } 0$$

## Präfix Tabelle

Faktor	Präfix	---
$10^{24}$	Yotta	Y
$10^{21}$	Zetta	Z
$10^{18}$	Exa	E
$10^{15}$	Peta	P
$10^{12}$	Tera	T
$10^9$	Giga	G
$10^6$	Mega	M
$10^3$	Kilo	k
$10^2$	Hekto	h
$10^1$	Deka	da
$10^0$		
$10^{-1}$	Dezi	d
$10^{-2}$	Zenti	c
$10^{-3}$	Milli	m
$10^{-6}$	Mikro	$\mu$
$10^{-9}$	Nano	n
$10^{-12}$	Piko	p
$10^{-15}$	Femto	f
$10^{-18}$	Atto	a
$10^{-21}$	Zepto	z
$10^{-24}$	Yokto	Y

Wahrheitstabellen Bsp.

A	B	C	D	Y
0	0	0	0	?
0	0	0	1	?
0	0	1	0	?
0	0	1	1	?
0	1	0	0	?
0	1	0	1	?
0	1	1	0	?
0	1	1	1	?
1	0	0	0	?
1	0	0	1	?
1	0	1	0	?
1	0	1	1	?
1	1	0	0	?
1	1	0	1	?
1	1	1	0	?
1	1	1	1	?

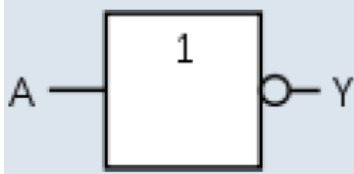

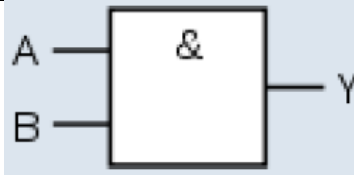

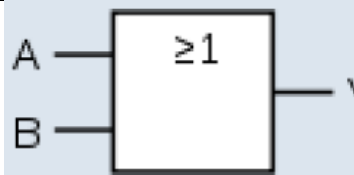

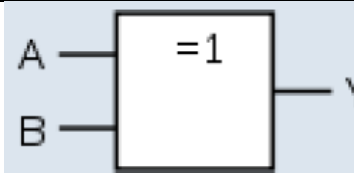

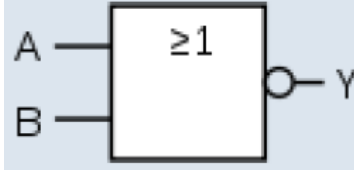
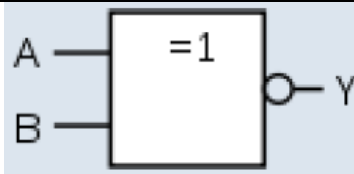
$\wedge$  = UND

$\vee$  = ODER

Herleitung:

$$\begin{aligned} & A \vee (A \wedge B) \\ &= (A \vee A) \wedge (A \vee B) \\ &= A \wedge (A \vee B) \\ &= A \wedge 1 \wedge (A \vee B) \\ &= A \wedge 1 \wedge 1 \\ &= A \end{aligned}$$

# Grundgatter:

Name	Funktion	Schaltsymbol	Wahrheitstabelle															
NOT-Gatter (Negation)	$Y = \bar{A}$		<table><tr><th>A</th><th>Y</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	Y	0	1	1	0									
A	Y																	
0	1																	
1	0																	
AND-Gatter (Konjunktion) (UND)	$Y = A \wedge B = A \cdot B = AB$ 		<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Y	0	0	0	0	1	0	1	0	0	1	1	1
A	B	Y																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
OR-Gatter (Disjunktion) (ODER)	$Y = A \vee B = A + B$ 		<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Y	0	0	0	0	1	1	1	0	1	1	1	1
A	B	Y																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
XOR-Gatter (Kontravalenz)	$Y = A \underline{\vee} B = A \oplus B$  $A \underline{\vee} B = (A \wedge \bar{B}) \vee (\bar{A} \wedge B)$		<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Y	0	0	0	0	1	1	1	0	1	1	1	0
A	B	Y																
0	0	0																
0	1	1																
1	0	1																
1	1	0																
NAND-Gatter	$Y = \overline{A \wedge B} = A \bar{\wedge} B = \overline{AB} = A B = A \uparrow B$		<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Y	0	0	1	0	1	1	1	0	1	1	1	0
A	B	Y																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
NOR-Gatter	$Y = \overline{A \vee B} = A \bar{\vee} B = \overline{A + B} = A \setminus B = A \downarrow B$		<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Y	0	0	1	0	1	0	1	0	0	1	1	0
A	B	Y																
0	0	1																
0	1	0																
1	0	0																
1	1	0																
XNOR-Gatter (Äquivalenz)	$Y = A \underline{\vee} B = \overline{A \vee B} = \overline{A \oplus B}$		<table><tr><th>A</th><th>B</th><th>Y</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Y	0	0	1	0	1	0	1	0	0	1	1	1
A	B	Y																
0	0	1																
0	1	0																
1	0	0																
1	1	1																

**Gatterlaufzeiten:** (=Änderungen an den Eingängen der Gatter sind nicht sofort an den Ausgängen sichtbar)

**Basic-Begriffe:**

Zeit für Änderungen von LOW nach HIGH =  $t_{PLH}$

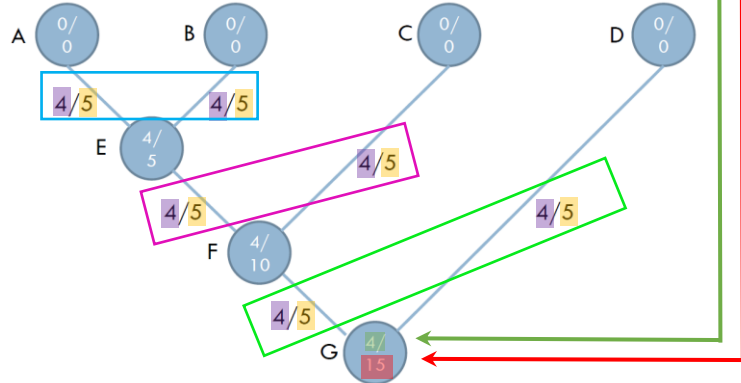
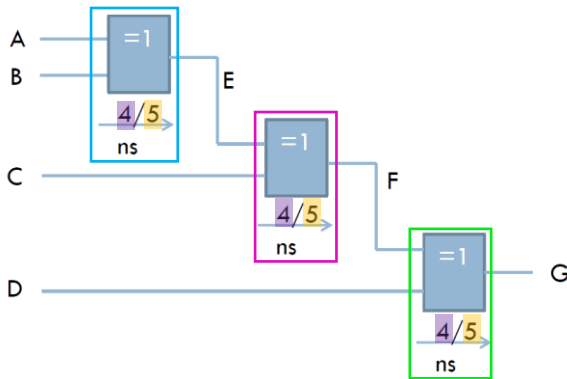
$$\text{maximale Taktfrequenz: } f_{\max} = \frac{1}{t_{\text{prop}}(\max)}$$

Zeit für Änderungen von HIGH nach LOW =  $t_{PHL}$

Langsamster Signalpfad (Kritischer Pfad) =  $t_{\text{prop}}(\max)$

Schnellster Signalpfad =  $t_{\text{prop}}(\min)$

**Statische Timing-Analyse:** (mit einem Beispiel als Erklärung)



**Funktionale Vollständigkeit:**

Mit diesen Verknüpfungen-(mengen) lassen sich alle anderen Verknüpfungen ausdrücken: {AND, NOT}; {NAND}; {NOR}

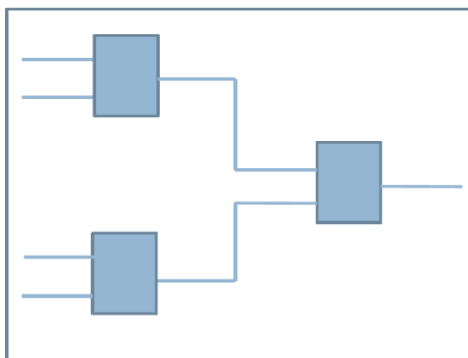
**NAND-Verknüpfungen**

Verknüpfung	Umsetzung
$\neg A$ (NOT)	$\overline{AA}$
$AB$ (AND)	$\overline{\overline{AB} \overline{AB}}$
$\overline{AB}$ (NAND)	$\overline{AB}$
$A \vee B$ (OR)	$\overline{\overline{AA} \overline{BB}}$
$\overline{A \vee B}$ (NOR)	$\overline{\overline{AA} \overline{BB} \overline{AA} \overline{BB}}$
$A \oplus B$ (XOR)	$\overline{A \overline{BB} \overline{AA} B}$
$\overline{A \oplus B}$ (XNOR)	$\overline{AB \overline{AA} \overline{BB}}$

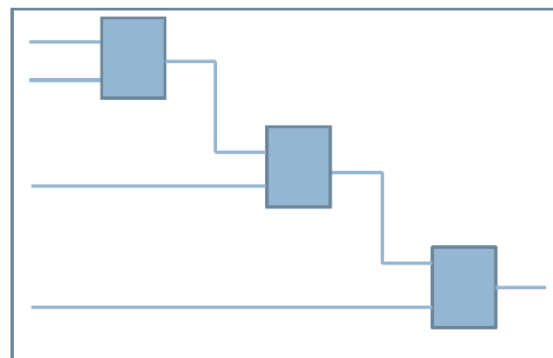
**Kaskadierung:** (Um Gatter mit mehr Eingängen zu erhalten)

**Basic-Begriffe:**

Schaltungstiefe = Anzahl der Gatter auf dem kritischen Pfad



**Baumstruktur**



**Kettenstruktur**

+ geringe Schaltungstiefe bei steigender Eingangsweite (= höhere Geschwindigkeit)	- hohe Schaltungstiefe bei steigender Eingangsweite
Folgende Zeile n = Eingangsbitanzahl	
Schaltungstiefe = $\text{ceil}(\log_2(n)) = \text{ceil}(\log_2(n))$	Schaltungstiefe = $n-1$
- hoher Gatterverbrauch bei steigender Ausgangsweite	+ geringer Gatterverbrauch bei steigenden Ausgangsweite
Schaltnetz mit n Eingangsbit und n-1 Ausgangsbits	
Gatteranzahl = $1 + 2 + 3 + \dots + (n-1) = \frac{(n-1)^2 + n - 1}{2}$	Gatteranzahl = $n-1$

## Wie viel Bit n für Dezimalzahl N?:

$$n = \log_2(N + 1) \quad \text{oder mit} \quad 64 \ 32 \ 16 \ 8 \ 4 \ 2 \ 1$$

## Disjunktive/Konjunktive Normalform:

### 1. Wahrheitstabelle aufstellen

DNF: Y = 1 markieren, Klauseln aufschreiben (Eingang 0 =  $\bar{A}$ , bei Eingang 1 = A)

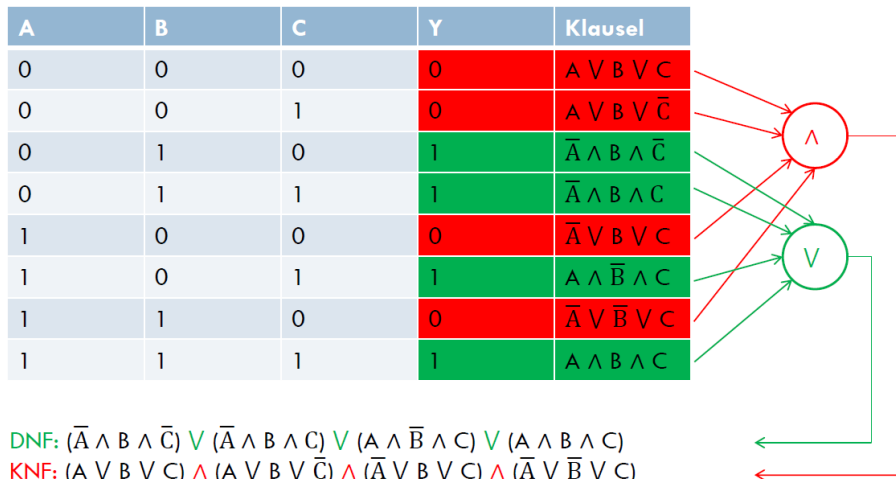
KNF: Y = 0 markieren, Klauseln aufschreiben (Eingang 0 = A, bei Eingang 1 =  $\bar{A}$ )

### 2. Formel mit gebildeten Klauseln aufstellen

DNF: (Klausel 1)  $\vee$  (Klausel 2)  $\vee$  (Klausel 3)  $\vee$  ...

KNF: (Klausel 1)  $\wedge$  (Klausel 2)  $\wedge$  (Klausel 3)  $\wedge$  ...

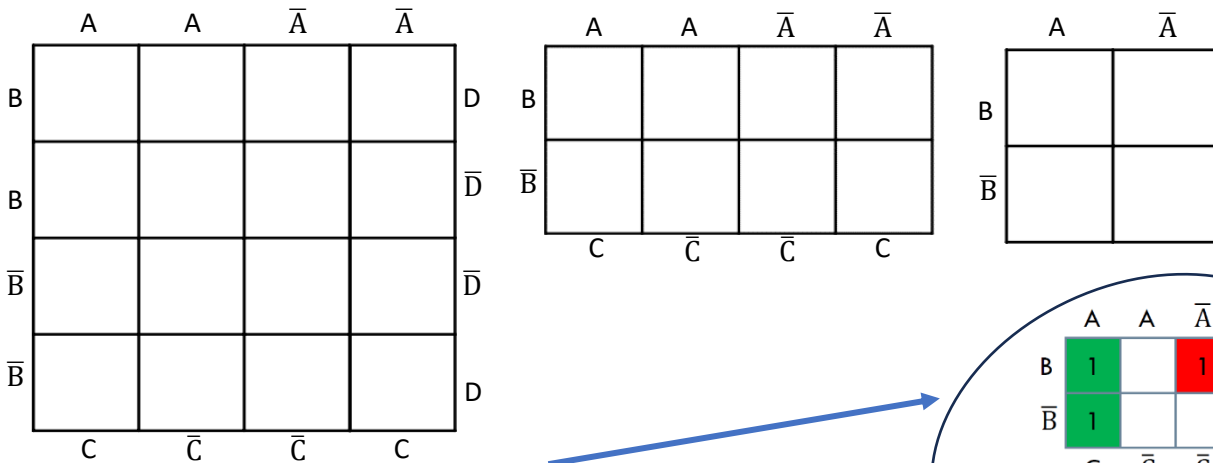
Beispiel:



## KV-Diagramm: (Vereinfachen einer booleschen Funktion)

### 1. Disjunktive Normalform (DNF) ist bekannt

### 2. DNF in KV-Diagramm einsetzen (1sen eintragen für die einzelnen Klauseln)



## Weitere Gatter:

Name	Funktion	Schaltsymbol	Wahrheitstabelle/Zusatz																																													
Halbaddierer	Zwei 1-Bit Eingänge (Summanden) werden zu zwei 1-Bit Ausgängen → S=niederwertigere Bit C=höherwertigere Bit		<table><tr><th>A</th><th>B</th><th>S (Summe)</th><th>C (Übertrag/ Carry)</th></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>1</td><td>0</td><td>1</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td><td>1</td></tr></table> <p><math>S = A \oplus B</math> (XOR) <math>C = A \wedge B</math> (AND)</p>	A	B	S (Summe)	C (Übertrag/ Carry)	0	0	0	0	1	0	1	0	0	1	1	0	1	1	0	1																									
A	B	S (Summe)	C (Übertrag/ Carry)																																													
0	0	0	0																																													
1	0	1	0																																													
0	1	1	0																																													
1	1	0	1																																													
Volladdierer	Drei 1-Bit Eingänge (Summanden) werden zu zwei 1-Bit Ausgängen → S=niederwertigere Bit C <sub>OUT</sub> =höherwertigere Bit		<table><tr><th>A</th><th>B</th><th>C<sub>in</sub></th><th>S</th><th>C<sub>out</sub></th></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table> <p><math>S = A \oplus B \oplus C_{in}</math> (XOR) <math>C_{out} = (A \wedge B) \vee (B \wedge C_{in}) \vee (A \wedge C_{in})</math> (AND)</p>	A	B	C <sub>in</sub>	S	C <sub>out</sub>	0	0	0	0	0	1	0	0	1	0	0	1	0	1	0	1	1	0	0	1	0	0	1	1	0	1	0	1	0	1	0	1	1	0	1	1	1	1	1	1
A	B	C <sub>in</sub>	S	C <sub>out</sub>																																												
0	0	0	0	0																																												
1	0	0	1	0																																												
0	1	0	1	0																																												
1	1	0	0	1																																												
0	0	1	1	0																																												
1	0	1	0	1																																												
0	1	1	0	1																																												
1	1	1	1	1																																												
Addiernetz	Addition zweier mehrstelliger Binärzahlen: Erste Eingang Bitzahl mit Bitbreite N Zweite Eingangs Bitzahl mit Bitbreite M Bitbreite Ausgang = max(N,M)+1  $C_{IN}+A_0A_1A_2A_3+B_0B_1B_2B_3=S_0S_1S_2S_3C_{OUT}$		<p>-hohe Schaltungstiefe, dauert lang</p>																																													
Multiplexer	Selektionsschaltung die aus mehrerer Eingänge einen auswählt und an den Ausgang schaltet.		<p>ODER-Gatter ; UND-Gatter</p>																																													
Komperator	Vergleicht zwei Eingangswerte	---	---																																													
Decoder	Bildet N-Bit Eingangsbit auf M-Ausgangsbit ab	Mehr Ausgangsbit M als Eingangsbit N	---																																													
Encoder	Bildet N-Bit Eingangsbit auf M-Ausgangsbit ab	Mehr Eingangsbit N als Ausgangsbit M	---																																													

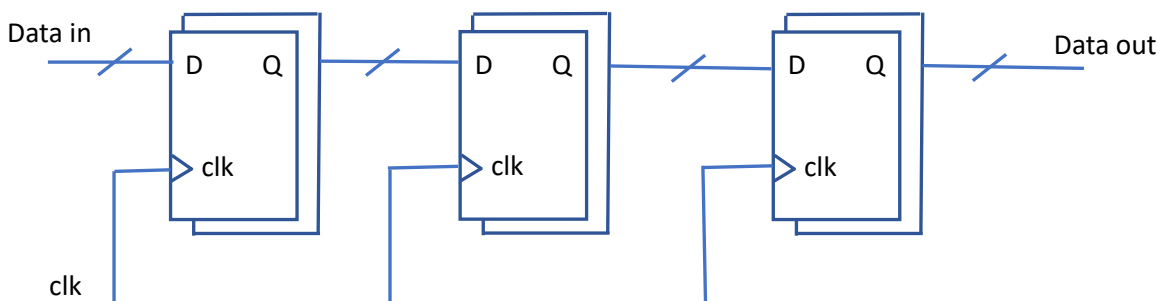
**Glitches:** (oder auch Hazards genannt) (=temporäre Falschaussage in einer kombinatorischen Schaltung)

*Ausgelöst:* durch Gatterlaufzeiten, unterschiedlich langen Leitungen

*Abhilfe:* Taktung der Schaltung, Angleichung der Verzögerungszeiten, Hinzufügen von Logikbausteinen

*Resultat:* Ausgang ist zwischen  $t_{prop}(min)$  und  $t_{prop}(max)$  undefiniert

Dafür Bsp. Delay mit 3 Takten Verzögerung



## Unterschied kombinatorische und sequenzielle Schaltung:

**Kombinatorische Schaltung:** keinen Takt, keine Rückkopplung

**Sequenzielle Schaltung:** Takt oder Rückkopplung ist vorhanden

**Schaltwerk:** (=sequenzielle Schaltung)

im Gegensatz zu einem Schaltnetz (kombinatorisch) (alles was davor in FS war) ist mindestens ein Ausgang rückgekoppelt.

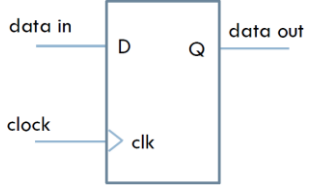
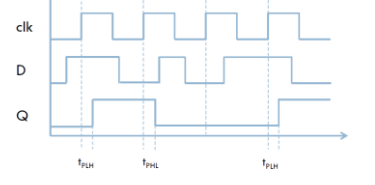
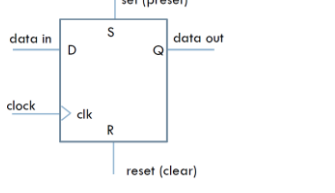
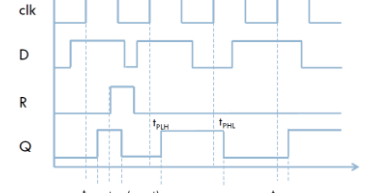
### Asynchrone Schaltwerke:

Kein Taktsignal

Rückkopplung über Verzögerungsglied (Leitung ist meistens ausreichend)

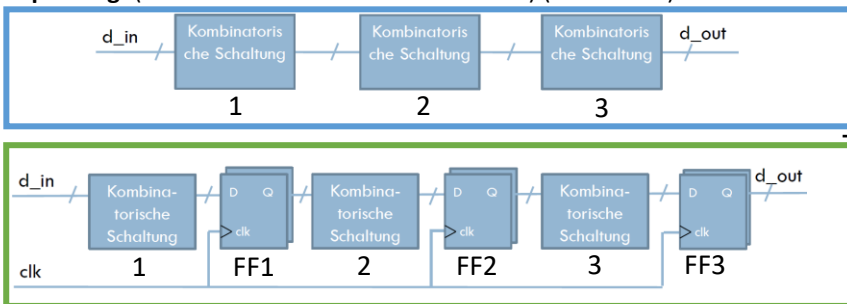
### Synchrone Schaltwerke:

Mit Taktsignal

D-Flipflop (Delay-Flipflop)	D wird zu Q bei jeder steigender Flanke von der clk mit einer Verzögerung von $t_{PLH}$ oder $t_{PHL}$ (beide auch genannt $t_{CO}$ )		
D-Flipflop (mit set/reset)	Set/reset sind meistens asynchron → bei reset wird Q zurückgesetzt		

**Register:** (= mehrere Flipflops in einer Gruppe um z.B. zu speichern)

**Pipelining:** (= bestimmte Struktur beim Aufbau) (vs. Normal)



$$\text{Laufzeit} = t_1 + t_2 + t_3$$

$$\text{Durchsatz} = \frac{\text{bit}}{\text{Laufzeit}}$$

$$t_{clk} = \frac{1}{f_{clk}}$$

$$\text{Laufzeit} = (t_{clk} \cdot \text{Anzahl FF}) + t_{ff3}$$

$$\text{Durchsatz} = \frac{\text{bit}}{t_{clk}} \quad (\text{Falls nicht } t_1 + t_2 + t_3 \text{ größer ist})$$

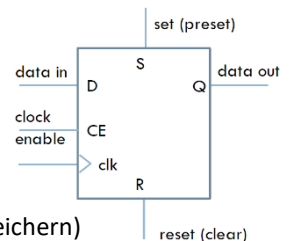
Dabei wird die Laufzeit/Latenz verschlechtert und der Durchsatz verbessert  
Ressourcenbedarf und Leistungsaufnahme erhöht sich auch

### Clock Enable:

Wenn clock enable = 0 ist wird auch bei einer steigender Flanke Q nicht von D übernommen

→ Schaltungsteile zeitweise deaktivieren

→ effektive Taktfrequenz in bestimmten Schaltungsteilen reduzieren



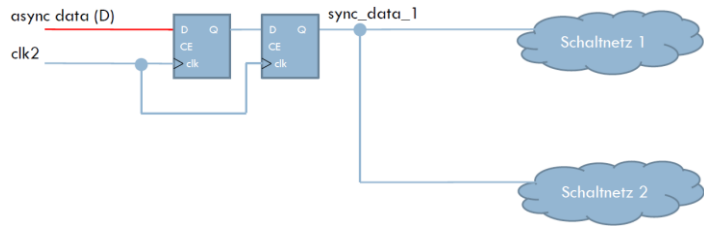
**Datenpfad:** (kombinatorische Schaltungen, die Funktionalität realisieren und mit Register zwischen-speichern)

**Kontrollpfad:** (steuern Ablauf der Datenpfade)

$t_{su}$  **Setup-time:** Zeit vor Tacktfllanke wo Stabilität gefordert ist **beim FF**  
 $t_{ho}$  **Hold-time:** Zeit nach Tacktfllanke wo Stabilität geforder ist **beim FF**  
 ➔ Wenn verletzt dann metastabiler Zustand am Ausgang  
 (Ausgang vom FF nicht innerhalb  $t_{co}$  seinen Wert an)

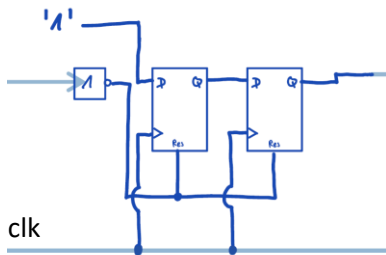
## Synchronisationsschaltung:

**Synchron:** (Aufgrund von Metastabilität von FFs darf nur an einer Stelle ein Signal synchronisiert werden)

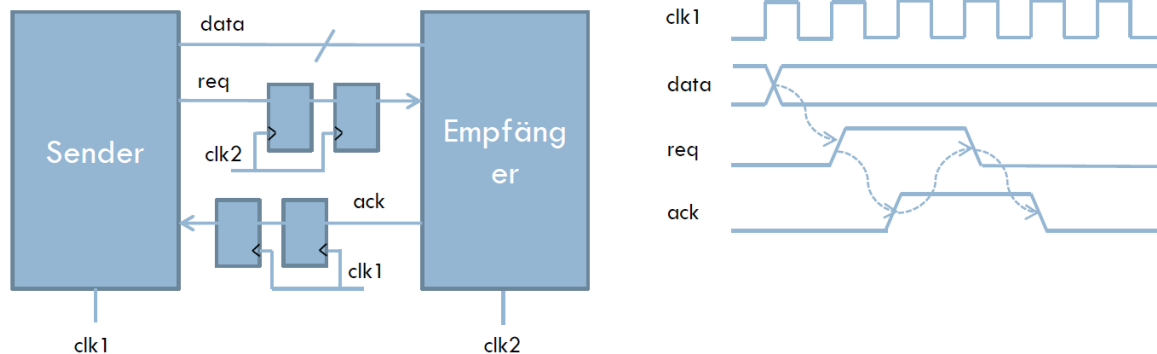


**Asynchron gesetzt und synchron rückgesetzt:**

Beispiel low-aktiv → wenn  $data_{in} = 0$  dann  $data_{out} = 0$  (asynchron)

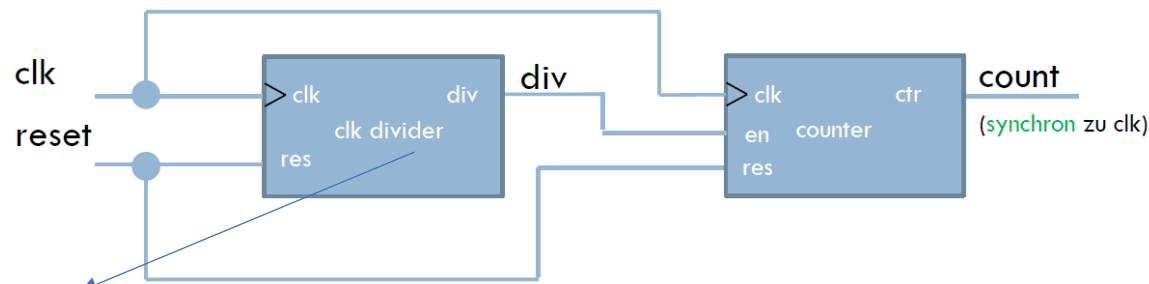


Bei Datenbus darf man es nicht wie oben machen, sondern z.B. einen Handshake Mechanismus verwenden:

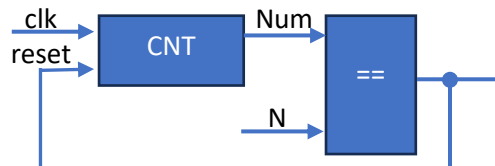


- Verlangsamen

**Synchrone Entwurfstechniken:**



Möglicher Aufbau:



## Resets:

Resetsignale bei FlipFlops haben folgende anforderungen:

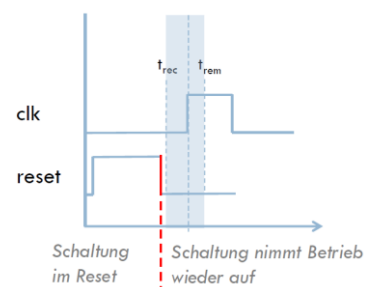
$t_{rec}$  **Recovery time:** Zeit vor einer clock-tackt-flanke wo reset sich nicht verändern darf

$t_{rem}$  **Removal time:** Zeit nach einer clock-tackt-flanke wo reset sich nicht verändern darf

→ Falls dies nicht beachtet wird und in dem Zeitfenster Ausgang vom FF ändern würde dann metastabil

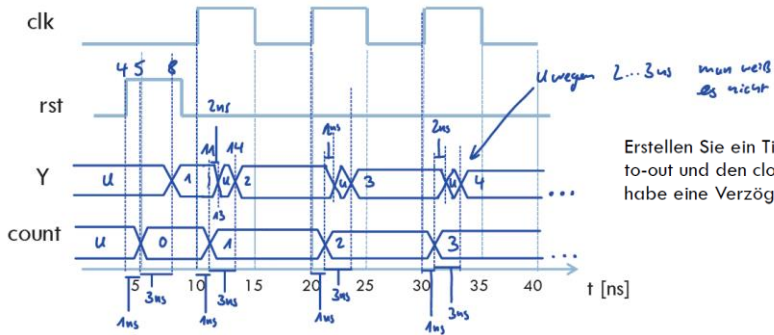
**Synchroner Reset** = nur mit Takt, Langsamer, keine Metastabilitätsprobleme

**Asynchroner Reset** = Funktioniert immer, schneller, Metastabilitätsprobleme





### Timing im Diagramm:



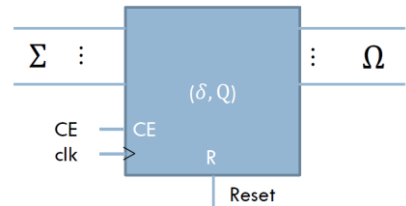
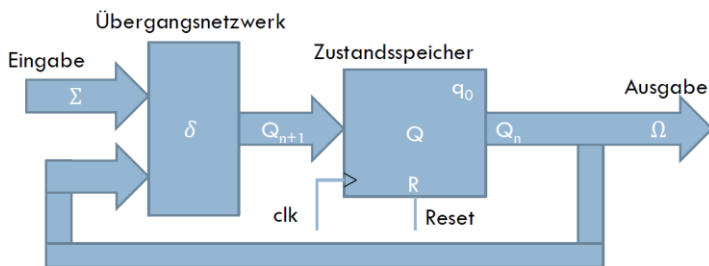
Erstellen Sie ein Timingdiagramm für den Counter, nehmen Sie für den reset-to-out und den clock-to-out  $t_{\text{PLH}} = t_{\text{PHL}} = 1\text{ ns}$  für das Flipflop an. Der Adder habe eine Verzögerung von  $t_{\text{prop}}(\text{min}) = 2\text{ ns}$  und  $t_{\text{prop}}(\text{max}) = 3\text{ ns}$ .

**Asynchrone Taktdomänen** (= abgeschlossener Schaltungsteil, der mit gleichen Takt betrieben wird)

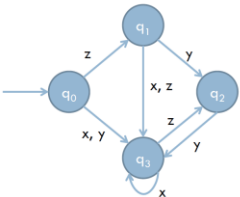
Synchron	Phasenverschiebung konstant ( $\Delta T = const$ ) Periodendauer gleich ( $T_1 = T_2$ )	
Mesochron	Phasenverschiebung unbekannt ( $\Delta T = ?$ ) Periodendauer gleich ( $T_1 = T_2$ )	
Plesiochron/Heterochron gering $\Delta f$ / groß $\Delta f$	Phasenverschiebung variable ( $\Delta T = var$ ) Periodendauer ungleich ( $T_1 \neq T_2$ )	

## Zustandsautomaten:

### Medwedew Automat:

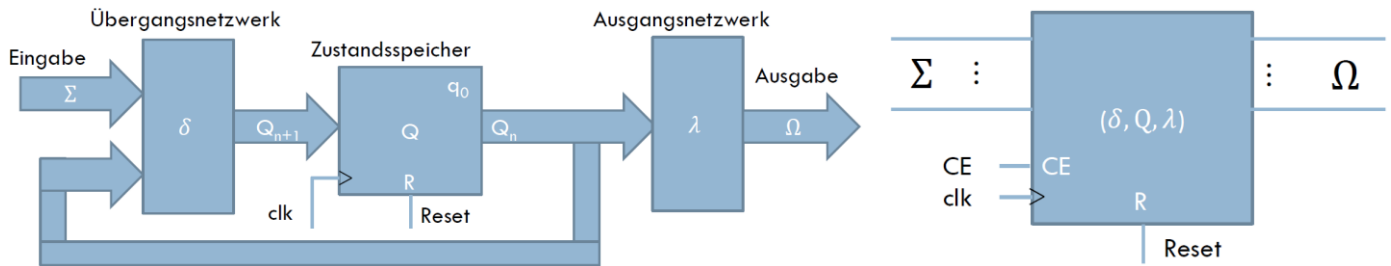


- Gegeben sei ein Medwedew Automat mit
- $Q = \{q_0, q_1, q_2, q_3\}$ ,  $q_0 = \text{Anfangszustand}$
- $\Sigma = \{x, y, z\}$
- Die Übergangsfunktion  $\delta$  lässt sich mit einem Graphen oder einer Tafel darstellen

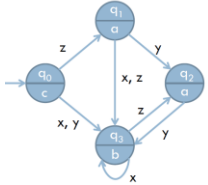


Aktueller Zustand Q	x	y	z
q <sub>0</sub>	q <sub>3</sub>	q <sub>3</sub>	q <sub>1</sub>
q <sub>1</sub>	q <sub>3</sub>	q <sub>2</sub>	q <sub>3</sub>
q <sub>2</sub>	-	q <sub>3</sub>	-
q <sub>3</sub>	q <sub>3</sub>	-	q <sub>2</sub>

## Moore Automat:

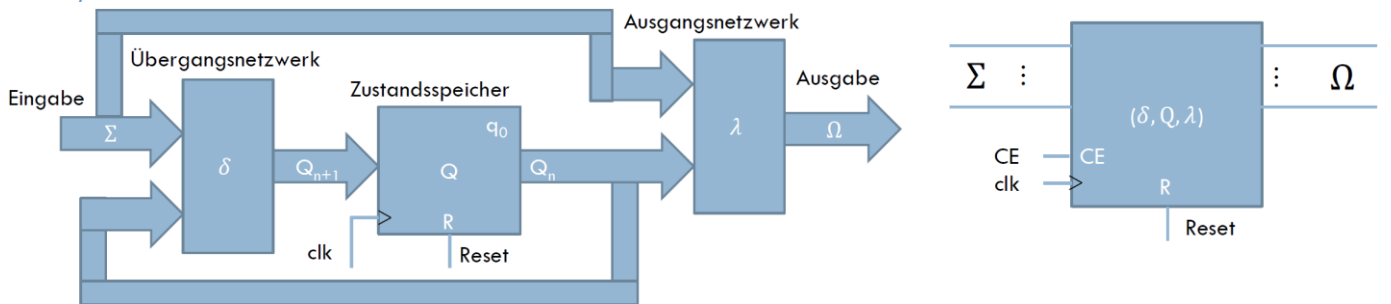


- Gegeben sei ein Moore Automat mit
- $Q = \{q_0, q_1, q_2, q_3\}$ ,  $q_0$ =Anfangszustand
- $\Sigma = \{x, y, z\}$
- $\Omega = \{a, b, c\}$
- Die Übergangsfunktion  $\delta$  lässt sich mit einem Graphen oder einer Tafel darstellen

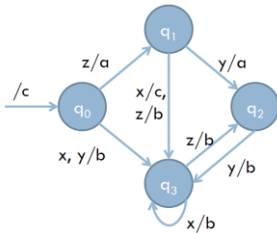


Aktueller Zustand Q	x	y	z	λ (Ausgabe)
q <sub>0</sub>	q <sub>3</sub>	q <sub>3</sub>	q <sub>1</sub>	c
q <sub>1</sub>	q <sub>3</sub>	q <sub>2</sub>	q <sub>3</sub>	a
q <sub>2</sub>	-	q <sub>3</sub>	-	a
q <sub>3</sub>	q <sub>3</sub>	-	q <sub>2</sub>	b

## Mealy Automat:



- Gegeben sei ein Mealy Automat mit
- $Q = \{q_0, q_1, q_2, q_3\}$ ,  $q_0$ =Anfangszustand
- $\Sigma = \{x, y, z\}$
- $\Omega = \{a, b, c\}$
- Die Übergangsfunktion  $\delta$  lässt sich mit einem Graphen oder einer Tafel darstellen



**Regel:**  
Bedingung für Zustandsänderung/passiert dann am Ausgang

## Schaltwerksynthese: (= umsetzen in Hardware)

Bei einem Schaltwerk mit  $n$  Zuständen sind  $k = \lceil \log_2(n) \rceil$  Zustandsregister nötig

### Zustandsminimierung:

Zustandsregister hat Bitbreite  $\rightarrow$  hängt von Anzahl Zuständen ab  $\rightarrow$  Versuch Zustandsanzahl zu minimieren

### Zustandskodierung:

Jeder Zustand bekommt binär Code mit länge  $k \rightarrow$  diese Wahl beeinflusst Ressourcenverbrauch und Verzögerungszeit

Zustandsübersicht und Ausgangstabelle:

Zustand	FF Ausgang	Eingänge		Folgezustand	FF Eingang	Ausgang
$Q_n$	$Q$	x	y	$Q_{n+1}$	$D$	a
q <sub>0</sub>	0			q <sub>0</sub>	0	
q <sub>1</sub>	1			q <sub>1</sub>	1	

Aufgabenspezifisch

Kodierungsmöglichkeiten: (Unbenutzte Möglichkeiten mit Don't Care versehen)

**Binary:** einfach fortlaufende Binärzahlen  $\rightarrow$  00, 01, 10, 11  $\rightarrow$  +kompaktes Zustandsregister(Ressourcenverbrauch) - Schlechte Schaltgeschwindigkeit

**One-Hot:** jeder Zustand ein Bit:  $\rightarrow$  001, 010, 100  $\rightarrow$  +beste Schaltgeschwindigkeit - Großes Zustandsregister

**Gray:** benachbarte Codewörter haben nur ein Bit unterschied  $\rightarrow$  00, 01, 11, 10  $\rightarrow$  +kompaktes Zustandsregister +gute Schaltgeschwindigkeit

Zeilenanzahl = 2<sup>Eingänge+(FFAusgang)</sup>

- a = Adresseingang (K =4)
- was soll in LUT ausgewählt werden
- DIN = data in
- Daten in LUT schreiben
- WR = falls 1 zu jeder Tclk wird geschrieben
- Gibt an, ob Schreibvorgang erfolgen soll
- WE =Write Enable
- zusätzliches Signal für Schreibvorgänge
- clk = clock
- selbsterklärend

## Einführung in VHDL

**Testbenches:** (=Prüfstand) VHDL-Datei, die VHDL Code mit Testdaten füttert (simuliert) um Verhalten und erzeugte Daten zu checken

**Abstraktionsebene:** Register-Transfer-Ebene

**Konzept:** VHDL arbeitet mit Prozessen die zueinander nebenläufig sind, Code innerhalb von Prozessen ist sequenziell

### Grundsätzlicher Aufbau:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY projectname IS
PORT (
-- hier werden die Eingangs-, Ausgangs- und Aus-Eingangssignale beschrieben
a : in std_logic_vector (7 downto 0); -- Eingang, Mehrbittig, Bitbreite
b : out std_logic; -- Ausgang, Einbittig
c : inout std_logic -- AusgangzuEingang, Einbittig (ohne ; zum Schluss)
);
END projectname;

ARCHITECTURE behave OF projectname IS
-- hier ist ein Declarationsteil (interne Signale deklariert)
BEGIN
-- beschreibt das interne Verhalten der Komponente (Verhaltensbeschreibung und Struckturbeschreibung)
END behave;
```

Datentypen – Konvertierung:		Bit	Sammlung von Bits	vorzeichenbehaftet	vorzeichenunbehaftet	
nach	von	std_logic	std_logic_vector	signed	unsigned	integer
Kommentar		Bit (einzelne Leitung)	Bits (mehrere Leitungen)	Positive Zahlen	Negative Zahlen	Konstanten im Quellcode
Bibliothek		std_logic_1164	std_logic_1164	numeric_std	numeric_std	-
std_logic	-	-	-	-	-	-
std_logic_vector	-	-	-	std_logic_vector(x)	std_logic_vector(x)	std_logic_vector(to_unsigned(x, len)) std_logic_vector(to_signed(x, len))
signed	-	-	signed(x)	-	signed(x)	to_signed(x, len)
unsigned	-	-	unsigned(x)	unsigned(x)	-	to_unsigned(x, len)
integer	-	-	to_integer(unsigned(x)) to_integer(signed(x))	to_integer(x)	to_integer(x)	-

### Funktionen für Bitbreitenanpassung:

```
-- resize Funktion:
resize(x, len); -- (x nur signed oder unsigned, gewünschte Bitlänge)
new_size <= resize(unsigned(old_size_std_logic_vector), 16);
new_size <= resize(signed(old_size_std_logic_vector), 16);
new_size <= resize(signed(old_size_std_logic_vector), new_size'length); -- (besser)
-- & Operator:
new_size : std_logic_vector (15 downto 0);
old_size : std_logic_vector (7 downto 0);
new_size <= x"00" & old_size; -- go from 8 to 16 bits, MSB with 0's
new_size <= old_size & x"00"; -- go from 8 to 16 bits, LSB with 0's
new_size <= old_size(15 downto 8); --go from 16 to 8 bits, use MSBs
new_size <= old_size(7 downto 0); --go from 16 to 8 bits, use LSBs
```

## Operationen:

```
y <= a + b -- benötigt eine Bitbreitenanpassung len(a)= len(b) = len(y)
y <= a * b -- benötigt keine Bitbreitenanpassung len(a)+ len(b) = len(y)
```

## Verhaltensbeschreibung: (besteht aus mehreren Prozessen)

```
ARCHITECTURE behave of projectname is
BEGIN
    PROCESS (clk, reset) -- (Sensitivitätsliste) wird ausgeführt wenn ein Signal sich ändert
    BEGIN
        -- sequentielle ausführung des Inhaltes
    END PROCESS;
END behave;
```

## Mit Takt und ohne Reset:

```
ARCHITECTURE behave OF adder IS
BEGIN
    PROCESS (clk)
    BEGIN
        IF rising_edge(clk) then
            y <= std_logic_vector(resize(unsigned(a), y'length) +
                                   resize(unsigned(b), y'length));
        END IF;
    END PROCESS;
END behave;
```

## High-aktiver synchroner Reset:

```
ARCHITECTURE behave OF adder IS
BEGIN
    PROCESS (clk)
    BEGIN
        IF rising_edge(clk) then
            IF reset = '1' THEN
                y <= (others => '0');
            ELSE
                y <= std_logic_vector(resize(unsigned(a), y'length) +
                                       +resize(unsigned(b), y'length));
            END IF;
        END IF;
    END PROCESS;
END behave;
```

## High-aktiver asynchroner Reset:

```
ARCHITECTURE behave OF adder IS
BEGIN
    PROCESS (clk, reset)
    BEGIN
        IF reset = '1' THEN
            y <= (others => '0');
        ELSIF rising_edge(clk) then
            y <= std_logic_vector(resize(unsigned(a), y'length) +
                                   resize(unsigned(b), y'length));
        END IF;
    END PROCESS;
END behave;
```

### Multiplexer: (mux.vhd)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY mux IS
PORT (
    sel: IN std_logic;
    a : IN std_logic;
    b : IN std_logic;
    y : OUT std_logic
);
END mux;

ARCHITECTURE behave OF mux IS
    SIGNAL y_int:std_logic;
BEGIN
    PROCESS(sel, a, b)
    BEGIN
        IF sel='1' THEN
            y_int<=a;
        ELSE
            y_int<=b;
        END IF;
    END PROCESS;
    y <= y_int;
END behave;
```

### Strukturbeschreibung:

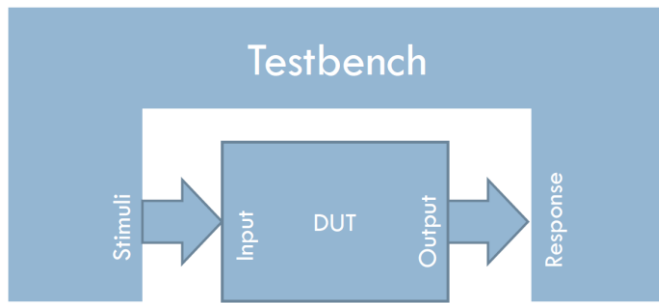
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity top_level is
Port(a : in STD_LOGIC_VECTOR(7 downto 0);
    b : in STD_LOGIC_VECTOR(7 downto 0);
    c : in STD_LOGIC_VECTOR(8 downto 0);
    z : out STD_LOGIC_VECTOR(17 downto 0);
    clk : in STD_LOGIC;
    reset : inSTD_LOGIC);
end top_level;
architecture Behavioral of top_level is
    component adder is
        Port (
            a1 : in STD_LOGIC_VECTOR(7 downto 0);
            a2 : in STD_LOGIC_VECTOR(7 downto 0);
            s : out STD_LOGIC_VECTOR(8 downto 0);
            clk : in STD_LOGIC;
            reset : in STD_LOGIC);
        end component;
    component multiplier is
        Port (
            f1 : in STD_LOGIC_VECTOR(8 downto 0);
            f2 : in STD_LOGIC_VECTOR(8 downto 0);
            p : out STD_LOGIC_VECTOR(17 downto 0);
            clk : in STD_LOGIC;
            reset : in STD_LOGIC);
        end component;
    signal adder_out : std_logic_vector(8 downto 0);
begin
    add : adder Port Map (a1=>a,a2=>b,s=>adder_out,clk=>clk,reset=>reset); --(comp,fremd)
```

Entity in anderen projekten muss genau so heißen

Instanzname (freiwählbar)

```
mul : multipliiert Port Map (f1=>adder_out, f2=>c, p=>z, clk=>clk, reset=>reset);
end Behavioral;
```

**Testbenches:** (Für die Simulation von VHDL-Modellen)



Anfang von Testbench immer einen Reset auslösen damit Schaltung in definierten Zustand ist  
Signale, die durch Reset nicht Initialisiert werden, werden bei std\_logic und std\_logic\_vector in Simulation mit U belegt

**Kompilieren von VHDL-Modellen:**

Übersetzungseinheiten: entity, architecture, configuration, package → werden jeweils separat kompiliert  
Bibliothek std wird automatisch referenziert (Packages textio)

**Simulation von VHDL-Modellen:**

Elaboration (Ausarbeitung):

- 1.) Hierarchische Expansion des Entwurfs → Compiler sucht alle Komponenten nach Hierarchy heraus
- 2.) Generics: Die aktuellen Werte der Generic werden eingesetzt
- 3.) Reservierung von Speicherplatz

Initialization:

- 1.) Alle Signale und Variablen werden auf ihren Standardwert oder auf Benutzer kodierter Wert gesetzt.
- 2.) Jeder Prozess wird einmal durchlaufen (entweder bis zum Ende oder bis Wait)

Execution:

„Ausführung des Simulationsmodells“ → Ergebnisorientiert (die Zeit wird nicht kontinuierlich simuliert)

**Variablen (:=) und Signale (<=)**

**Signale:** Zuweisung von Werten am Ende vom Prozess oder Wait (bei Mehrfach Zuweisung zählt letzte)

**Variablen:** siehe Programmbeispiel

```
process (clk)
    variable b : std_logic_vector(7 downto 0); -- Deklaration
begin
    if rising_edge(clk) then
        if reset = '0' then
            c <= (others => '0');
        else
            b := a + "10"; -- Zuweisung
            c <= b * "10";
        end if;
    end if;
end process;
```

**Objekt Gültigkeitsbereich:** Objektdекlaration in...

- ... Entity gilt für alle zur Entity gehörigen Architectures
- ... Architecture gilt nur für diese Architecture
- ... Prozess gilt nur für diesen Prozess

## Datentypen und Operatoren:

**Skalar – Integer:** positive und negative ganze Zahlen

```
TYPE type_name IS RANGE int_range_constraint; -- allgemeine Deklaration
TYPE integer IS RANGE -2147483648=[-2**(31-1)] TO 2147483647 = [2**(31-1)-1];
```

**Composite – Array:** mehrere Werte des gleichen Typs unter dem gleichen Identifizierer

```
TYPE array_type_name IS ARRAY (range_constraints) OF type; -- allgemeine Deklaration
```

Architecture Behavioral of toplevel is

```
TYPE Column IS RANGE 1 TO 80;
```

```
TYPE Row IS RANGE 1 TO 24;
```

```
TYPE Matrix IS ARRAY(Row,Column) OF boolean;
```

```
Signal my_matrix:Matrix;
```

```
begin
```

```
my_matrix(1,1) <= true; -- 1 Zeile , 1 Spalte = Element nun bekannt
```

```
end Behavioral;
```

Operatoren:

Operator	Operation	Datentyp linker Operand a	Datentyp rechter Operand b	Datentyp Ergebnis
**	$a^b$	Nur Basis 2	integer	integer
abs	$ b $	-	integer	integer
not	$\bar{b}$	-	bit, boolean, bit_vector	wie Operand

Operator	Operation	Datentyp linker Operand a	Datentyp rechter Operand b	Datentyp Ergebnis
*	$a*b$	integer	integer	integer
/	$a/b$	integer	2er-Potenz	integer
mod rem	Rest von $a/b$	integer	2er-Potenz	integer

Operator	Operation	Datentyp linker Operand a	Datentyp rechter Operand b	Datentyp Ergebnis
+ -	$\pm b$	-	integer	integer

Operator	Operation	Datentyp linker Operand a	Datentyp rechter Operand b	Datentyp Ergebnis
+ -	$a+b$ $a-b$	integer	integer	integer
&	Verkettung	bit_vector[n]	bit_vector[m]	bit_vector[n+m]



Operator	Operation	Datentyp linker Operand a	Datentyp rechter Operand b	Datentyp Ergebnis
sll	links (logisch)	bit_vector	integer	bit_vector
srl	rechts (logisch)			
sla	links (arith.)			
sra	rechts (arith.)			
rol	links rotieren			
ror	rechts rotieren			

```
--A = "10010101"
A sll2 = "01010100" --shift left logical, filled with 0
A srl3 = "00010010" --shift right logical, filled with 0
A sla3 = "10101111" --shift left arithmetic, filled with right bit
A sra2 = "11100101" --shift right arithmetic, filled with left bit
A rol3 = "10101100" --rotate left by 3
A ror5 = "10101100" --rotate right by 5
```

Operator	Operation	Datentyp linker Operand a	Datentyp rechter Operand b	Datentyp Ergebnis
=	a=b	alle Typen	wie linker Operand	boolean
/=	a≠b			
<	a<b			
<=	a≤b			
>	a>b			
>=	a≥b			

Operator	Operation	Datentyp linker Operand a	Datentyp rechter Operand b	Datentyp Ergebnis
and	$a \wedge b$	bit, boolean, bit_vector	wie linker Operand	wie linker Operand
or	$a \vee b$			
nand	$\overline{a \wedge b}$			
nor	$\overline{a \vee b}$			
xor	$a \underline{\vee} b$			
xnor	$\overline{a \underline{\vee} b}$			

### Sequenzielle Anweisungen: (nur innerhalb von Prozessen)

- **IF-Verzweigung:** keine Gedächtnisprogrammierung bei nicht getacktete Prozesse → Latches in Syntese
- **CASE-Verzweigung:** Funktionalität Multiplexer

```
Library IEEE;
useIEEE.STD_LOGIC_1164.ALL;
entity toplevel is
    Port(in0:inSTD_LOGIC;
          in1 : inSTD_LOGIC;
          sel : inintegerrange0to3;
          y : out STD_LOGIC);
end toplevel;
architecture Behavioral of toplevel is
begin
    P1 : process(sel,in0,in1)
    begin
        case sel is
            when 0=> y <=in0;
            when 1 => y <=in0;
            when 2 => y <=in1;
            when 3 => y <=in1;
        end case;
    end process;
end Behavioral;
```

- **Schleifen:** (spezielle Schleife mit statischen Grenzen)

```
Library ieee;
Use ieee.std_logic_1164.all;
Entity shift_registers_1 is
    port(CLK,SI :instd_logic;
          SO :outstd_logic);
End shift_registers_1;
Architecture archi of shift_registers_1is
    Signal tmp:std_logic_vector(7 downto 0);
begin
    process (CLK)
    begin
        if rising_edge(clk) then
            for i in 0 to 6 loop -- Hardcodierte Zahlen
                tmp(i+1)<=tmp(i);
            end loop;
            tmp(0)<=SI;
        end if;
    end process;
    SO <= tmp(7);
End archi;
```

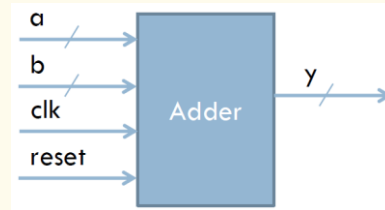
## Beispiele für VHDL-Codierung:

### Beispiel 8 Bit Addierer: (adder.vhd)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY adder IS
PORT(
clk : IN std_logic;
reset : IN std_logic;
a : IN std_logic_vector (7 downto 0);
b : IN std_logic_vector (7 downto 0);
y : OUT std_logic_vector (7 downto 0)
);
END adder;

ARCHITECTURE behave OF adder IS
BEGIN
y <= std_logic_vector(unsigned(a) + unsigned(b));
END behave;
```



### Beispiel 1 LED (LED.vhd)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY LED IS
PORT(
    LEDs : OUT std_logic_vector(7 downto 0)
);
END LED;

ARCHITECTURE behave of LED is

BEGIN
    LEDs <= "10101010"; -- <= Zuweisungsoperator, muss 8 bit sein weil oben auch
    -- Index:76543210
END behave;
```

### Beispiel 1 LED mit Taster: (LED\_Taster.vhd)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY LED_Taster IS
PORT(
    LEDs : OUT std_logic_vector(7 downto 0);
    Taster : IN std_logic
);
END LED_Taster;

ARCHITECTURE behave of LED_Taster is

SIGNAL Taster_inv: std_logic;

BEGIN
Taster_inv <= not Taster;
    LEDs <= (others => Taster_inv);
    -- others bedeutet auf jeden einzigen Index wird dieses Signal
    zugewiesen
END behave;
```

### Beispiel 1: LED mit zwei Taster und Gatter (LED\_Gatter.vhd)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY LED_Gatter IS
PORT(
    LEDs : OUT std_logic_vector(7 downto 0);
    Taster : IN std_logic;
    Taster2 : IN std_logic
);
END LED_Gatter;
ARCHITECTURE behave of LED_Gatter is
SIGNAL Taster_inv: std_logic;
SIGNAL Taster2_inv: std_logic;
BEGIN
Taster_inv <= not Taster;
Taster2_inv <= not Taster2;
    LEDs <= (others => (Taster_inv and Taster2_inv));
    -- others bedeutet auf jeden einzigen Index wird dieses Signal zugewiesen
END behave;
```

### Beispiel 1 weiter mit Gattern (LED\_XOR.vhd)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY LED_XOR IS
PORT(
    LEDs : OUT std_logic_vector(7 downto 0);
    Taster : IN std_logic;
    Taster2 : IN std_logic
);
END LED_XOR;
ARCHITECTURE behave of LED_XOR is
SIGNAL Taster_inv: std_logic;
SIGNAL Taster2_inv: std_logic;
BEGIN
Taster_inv <= not Taster;
Taster2_inv <= not Taster2;
LEDs(0) <= Taster_inv and Taster2_inv;
LEDs(1) <= Taster_inv or Taster2_inv;
LEDs(2) <= Taster_inv xor Taster2_inv;
LEDs(3) <= Taster2_inv;
LEDs(4) <= Taster_inv;
END behave;
```

### Beispiel 1 LED und Taster mit primitiver if-anweisung (LED\_XOR.vhd)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY LED_XOR IS
PORT(
    LEDs : OUT std_logic_vector(7 downto 0);
    Taster : IN std_logic;
    Taster2 : IN std_logic
);
END LED_XOR;
ARCHITECTURE behave of LED_XOR is
```

```

SIGNAL Taster_inv: std_logic;
SIGNAL Taster2_inv: std_logic;
BEGIN
Taster_inv <= not Taster;
Taster2_inv <= not Taster2;
    PROCESS(Taster_inv)
    BEGIN
        if Taster_inv = '1' then
            LEDs(0) <= '1';-- einfaches ' ' weil einbittig bei mehrbittig ""!
        else
            LEDs(0) <= '0';
        end if;
    END PROCESS;
END behave;

```

#### Beispiel 1 LED und Taster mit besserer if-anweisung (LED\_XOR.vhd)

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY LED_XOR IS
PORT(
    LEDs : OUT std_logic_vector(7 downto 0);
    Taster : IN std_logic;
    Taster2 : IN std_logic
);
END LED_XOR;
ARCHITECTURE behave of LED_XOR is
SIGNAL Taster_inv: std_logic;
SIGNAL Taster2_inv: std_logic;
BEGIN
Taster_inv <= not Taster;
Taster2_inv <= not Taster2;
    PROCESS(Taster_inv)
    BEGIN
        -- besser mit rising edge
        if rising_edge(Taster_inv) then -- dann gibt es keine else statement
auser bei asynchroner Reset
            LEDs(0) <= Taster2_inv;
        end if;
    END PROCESS;
END behave;

```

#### Beispiel 2 Addierer (adder.vhd)

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all; -- access to signed and unsigned types
ENTITY adder is
PORT(
    a, b: in std_logic_vector (7 downto 0);
    y: out std_logic_vector (7 downto 0);
    overflow: out std_logic
);
END adder;
ARCHITECTURE behave of adder is
SIGNAL res_9bit : std_logic_vector(8 downto 0);
BEGIN

```

```

y <= std_logic_vector(resize(unsigned(a), y'length) +
                        to_unsigned(3,y'length)
                        );
-- oder mit Überlauf:
res_9bit <= std_logic_vector(resize(unsigned(a), res_9bit'length) +
                             resize(unsigned(b), res_9bit'length)
                             );
y <= res_9bit(7 downto 0);
overflow <= res_9bit(8);
END behave;

```

### Beispiel 3: Multiplizierer (multipli.vhd) (8 Bit Eingang auf 16 Bit Ausgang)

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
ENTITY multipli IS
PORT(
    a, b: in std_logic_vector (7 downto 0);
    y: out std_logic_vector (15 downto 0)
);
END multipli;
ARCHITECTURE behave of multipli IS
BEGIN
    y <= std_logic_vector(signed(a)*signed(b));
END behave;

```

### Beispiel 4: FlipFlop (FlipFlop.vhd)

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY FlipFlop IS
PORT(
    D, CLK, CE, RST : in std_logic;
    Q : out std_logic
);
END FlipFlop;
ARCHITECTURE behave of FlipFlop IS
BEGIN
    PROCESS (CLK, RST, CE)
    BEGIN
        if RST = '1' then
            Q <= '0';
        elsif rising_edge(CLK) and CE = '1' then
            Q <= D;
        end if;
    END PROCESS;
END behave;

```

### Beispiel 8 Bit Zähler mit asynchronen Reset:

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all; -- für unsigned
ENTITY counter IS
PORT (
    USER_BTN, RESET : IN std_logic;
    LEDs : OUT std_logic_vector(7 downto 0)

```

```

);
END counter;
ARCHITECTURE behave of counter is
    SIGNAL counter: unsigned(7 downto 0);
BEGIN
    PROCESS (USER_BTN, RESET)
    BEGIN
        if RESET = '0' then
            counter <= (others => '0');
        elsif rising_edge(USER_BTN) then
            counter <= counter + 1;
        end if;
    END PROCESS;
    LEDs <= std_logic_vector(counter);
END behave;

```

#### Beispiel 8 Bit Zähler mit synchronen Reset:

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY counter IS
PORT (
    USER_BTN, RESET : IN std_logic;
    LEDs : OUT std_logic_vector(7 downto 0)
);
END counter;

ARCHITECTURE behave of counter is
    SIGNAL counter: unsigned(7 downto 0);
BEGIN
    PROCESS (USER_BTN)
    BEGIN
        if rising_edge(USER_BTN) then
            if RESET = '0' then -- reset innerhalb von rising_edge
                counter <= (others => '0');
            else
                counter <= counter + 1;
            end if;
        end if;
    END PROCESS;
    LEDs <= std_logic_vector(counter);
END behave;

```

#### Beispiel 8 Bit Zähler mit hoch und runterzählmöglichkeit

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY counter IS
PORT (
    USER_BTN, RESET : IN std_logic;
    LEDs : OUT std_logic_vector(7 downto 0)
);
END counter;

```

```

ARCHITECTURE behave of counter is
    SIGNAL counter: unsigned(7 downto 0);
BEGIN
    PROCESS (USER_BTN)
    BEGIN
        if rising_edge(USER_BTN) then
            if RESET = '0' then
                counter <= counter + 1;
            else
                counter <= counter - 1;
            end if;
        end if;
    END PROCESS;
    LEDs <= std_logic_vector(counter);
END behave;

```

#### Beispiel 8 Bit Zähler an PLL:

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY counter IS
PORT (
    CLK12M, USER_BTN, RESET : IN std_logic;
    LEDs : OUT std_logic_vector(7 downto 0)
);
END counter;

ARCHITECTURE behave of counter is
    SIGNAL counter: unsigned(7 downto 0);
    SIGNAL count12MHz: unsigned (23 downto 0);
BEGIN
    PROCESS (CLK12M)
    BEGIN
        if rising_edge(CLK12M) then
            if RESET = '0' then
                counter <= (others => '0');
                count12MHz <= (others => '0');
            else
                count12MHz <= count12MHz + 1;
                if count12MHz >= 11999999 then
                    count12MHz <= (others => '0');
                    counter <= counter + 1;
                end if;
            end if;
        end if;
    END PROCESS;
    LEDs <= std_logic_vector(counter);
END behave;

```

#### Beispiel Lauflicht:

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

```



```

ENTITY counter IS
PORT (
    CLK12M, USER_BTN, RESET : IN std_logic;
    LEDs : OUT std_logic_vector(7 downto 0)
);
END counter;
ARCHITECTURE behave of counter is
    SIGNAL shift_reg: unsigned(7 downto 0);
    SIGNAL count12MHz: unsigned (23 downto 0);
BEGIN
    PROCESS(CLK12M, RESET)
    BEGIN
        if RESET = '0' then
            shift_reg <= "00000001";
            count12MHz <= (others => '0');
        elsif rising_edge(CLK12M) then
            count12MHz <= count12MHz + 1;
            if count12MHz >= 11999999 then
                count12MHz <= (others => '0');
                -- dem Register shift_reg werden die
                -- untersten 7 Bit + das oberste Bit neu zugewiesen
                -- Kaufmannsund (&) ist der sog. concatenation operator
                shift_reg <= shift_reg(6 downto 0) & shift_reg(7);
                -- shift_reg <= "1000" & "0100";
                --shift_reg <= "10000100";
            end if;
        end if;
    END PROCESS;
    LEDs <= std_logic_vector(shift_reg);
END behave;

```

### Beispiel Modularisierter Adder, Multiplizierer, Subtrahierer:

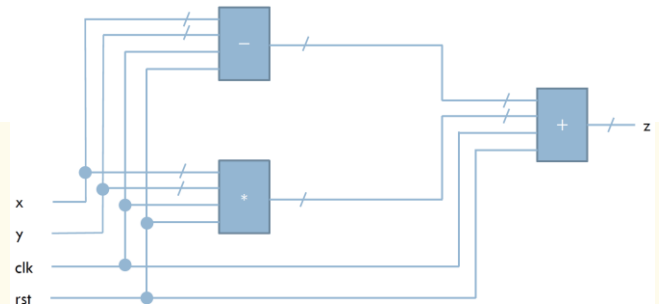
#### toplevel.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity topLevel is
    Port (
        clk      : in  STD_LOGIC;
        reset    : in  STD_LOGIC;
        x        : in  SIGNED(3 downto 0);
        y        : in  SIGNED(3 downto 0);
        z        : out SIGNED(8 downto 0)
    );
end topLevel;
architecture Structural of topLevel is
    signal mult_result : SIGNED(7 downto 0);
    signal sub_result  : SIGNED(7 downto 0);
    signal add_result  : SIGNED(8 downto 0);

    component Mult_Block is
    Port (
        a      : in  SIGNED(3 downto 0); -- Eingabe A (4 Bit signed)
        b      : in  SIGNED(3 downto 0); -- Eingabe B (4 Bit signed)
    );

```



```

        result: out SIGNED(7 downto 0)  -- Ausgabe (8 Bit signed)
    );
    end component;
    component Sub_Block IS
    PORT (
        a,b : IN SIGNED(7 downto 0);
        result: OUT SIGNED(7 downto 0)
    );
    END component;
    component Adder_Block is
    Port (
        a      : in  SIGNED(8 downto 0); -- Eingabe A (9 Bit signed)
        b      : in  SIGNED(8 downto 0); -- Eingabe B (9 Bit signed)
        result: out SIGNED(8 downto 0)  -- Ausgabe (9 Bit signed)
    );
    end component;
begin
    Mult_Inst : Mult_Block
        Port map (
            a => x,
            b => y,
            result => mult_result
        );
    Sub_Inst : Sub_Block
        Port map (
            a => resize(x, 8),
            b => resize(y, 8),
            result => sub_result
        );
    Add_Inst : Adder_Block
        Port map (
            a => resize(sub_result, 9),
            b => resize(mult_result, 9),
            result => add_result
        );
    z <= add_result;
end Structural;

```

#### Mult\_Block.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity Mult_Block is --gleiche Entity wie in ''main''
    Port (
        a      : in  SIGNED(3 downto 0); -- Eingabe A (4 Bit signed)
        b      : in  SIGNED(3 downto 0); -- Eingabe B (4 Bit signed)
        result: out SIGNED(7 downto 0)  -- Ausgabe (8 Bit signed)
    );
end Mult_Block;
architecture Behavioral of Mult_Block is
begin
    result <= a * b; -- Multiplikation
end Behavioral;

```

## Sub\_Block.vhd

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
ENTITY Sub_Block IS --gleiche Entity wie in ''main''
    PORT (
        a,b : IN SIGNED(7 downto 0);
        result: OUT SIGNED(7 downto 0)
    );
END Sub_Block;
ARCHITECTURE behave of Sub_Block IS
BEGIN
    result <= a - b;
END behave;
```

## Adder\_Block.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity Adder_Block is --gleiche Entity wie in ''main''
    Port (
        a      : in  SIGNED(8 downto 0); -- Eingabe A (9 Bit signed)
        b      : in  SIGNED(8 downto 0); -- Eingabe B (9 Bit signed)
        result: out SIGNED(8 downto 0)  -- Ausgabe (9 Bit signed)
    );
end Adder_Block;
architecture Behavioral of Adder_Block is
begin
    result <= a + b; -- Addition
end Behavioral;
```

## Beispiel Simulation "testbenches":

### adder.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity adder is
    Port(
        x, y : in std_logic_vector(7 downto 0);
        clk, reset : in std_logic;
        overflow : out std_logic;
        z : out std_logic_vector(7 downto 0)
    );
end adder;
architecture behave of adder is
    signal add : unsigned(8 downto 0);
begin
    process(clk, reset)
    begin
        if reset = '0' then
            add <= (others => '0');
        elsif rising_edge(clk) then
            add <= resize(unsigned(x), add'length) +
```

```

        resize(unsigned(y), add'length);
    end if;
end process;
overflow <= add(8);
z <= std_logic_vector(add(7 downto 0));
end behave;

```

#### testbench.vhd

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity testbench is
-- die entity von testbench ist immer leer!
end testbench;
architecture behave of testbench is
    -- Component Declaration for the Unit Under Test (UUT)
    component adder is -- gleich wie entity aus adder.vhd
        Port(
            x, y : in std_logic_vector(7 downto 0);
            clk, reset : in std_logic;
            overflow : out std_logic;
            z : out std_logic_vector(7 downto 0)
        );
    end component;
    -- Inputs
    signal x,y : std_logic_vector(7 downto 0);
    signal clk,reset : std_logic;
    -- Outputs
    signal overflow : std_logic;
    signal z : std_logic_vector(7 downto 0);
    -- Clock period definitions
    constant clk_period : time := 10 ns; -- Konstante (immer initialisieren), vom typ time
begin
    -- Instantiate the Unit Under Test (UUT)
    uut : adder Port Map( -- damit wird eine Instanz erzeugt die Verbindungen ''Leitungen'' erzeugt
        x => x,
        y => y,
        clk => clk,
        reset => reset,
        overflow => overflow,
        z => z
    );
    -- clock process definitions
    clk_process: process -- ohne sensitivitätsliste
    begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
    end process;
    -- Stimulus process
    stimuli : process -- läuft parallel zum clk-prozess ab
    begin
        reset <= '0';
        x <= (others => '0'); --std_logic_vector(to_unsigned(0, x'length)); (anderer Syntax)
    end process;

```

```

y <= (others => '0'); --std_logic_vector(to_unsigned(0, y'length)); (anderer Syntax)
wait for 10 ns;
reset <= '1';
x <= std_logic_vector(to_unsigned(3, x'length));
y <= std_logic_vector(to_unsigned(5, x'length));
wait for 10 ns;
x <= std_logic_vector(to_unsigned(255, x'length));
y <= std_logic_vector(to_unsigned(1, x'length));
wait for 10 ns;
x <= std_logic_vector(to_unsigned(255, x'length));
y <= std_logic_vector(to_unsigned(2, x'length));
wait for 10 ns;
reset <= '0';
wait;
end process;
end behave;

```

#### Beispiel Counter: toplevel\_4\_09.vhd

(16bit Zähler; counter = ARR → counter = 0; counter=0 → counter\_overflow = 1; counter<CRR → PWM high, sonst low)

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity counter is
    Port (
        enable, clk, reset : in STD_LOGIC;
        ARR : in STD_LOGIC_VECTOR (15 downto 0);
        CCR : in STD_LOGIC_VECTOR (15 downto 0);
        PWM: out STD_LOGIC;
        counter_overflow: out STD_LOGIC
    );
end counter;

architecture behave of counter is

    signal cnt: unsigned (15 downto 0);

begin

    cnt_proc: process(clk, reset)
    begin
        if reset = '1' then
            cnt <= (others => '0');
            PWM <= '0';
            counter_overflow <= '0';
        elsif rising_edge(clk) then
            if enable = '1' then

                counter_overflow <= '0';
                PWM <= '0';
                if cnt < unsigned(ARR) then
                    cnt <= cnt + 1;
                else
                    cnt <= (others => '0');

```

```

        counter_overflow <= '1';
        -- PWM auf 0 lassen, wenn CCR=0 ist
        if CCR /=
std_logic_vector(to_unsigned(0,CCR'LENGTH)) then
            PWM <= '1';
        end if;
    end if;

    -- PWM auf 0 lassen, wenn CCR=0 ist
    if CCR /= std_logic_vector(to_unsigned(0,CCR'LENGTH))
then
        if cnt < unsigned (CCR) - 1 then
            PWM <= '1';
        end if;
    end if;

    end if;
end if;
end process;

end behave;

```

**Beispiel ROM** (Adressbreite  $N = 3$  bit  $2^3 = 8$ , Datenbreite  $M = 8$  bit, Reset asynchron Low-aktiv, enable Eingang)

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
entity toplevel is
Port (
enable, clk, reset : in STD_LOGIC;
addr : in STD_LOGIC_VECTOR (2 downto 0);
data_out : out STD_LOGIC_VECTOR (7 downto 0)
);
end toplevel;
architecture behave of toplevel is
type rom_type is array (0 to 7) of std_logic_vector (7 downto 0);
signal ROM: rom_type := ("00000000", -- index 0
    "11110011", -- index 1
    "01001010", -- index 2
    "01100000", -- index 3
    "10000001", -- index 4
    "10110011", -- index 5
    "01101110", -- index 6
    "11110000" -- index 7
);
begin
    process(clk, reset)
    begin
        if reset = '0' then
            data_out <= (others => '0');
        elsif rising_edge(clk) then
            if enable = '1' then
                data_out <= ROM(to_integer(unsigned(addr)));
            end if; -- Zugriff muss Integer sein (oben)
        end if;
    end process;
end behave;

```

**Beispiel RAM** (Adressbreite  $N = 3$  bit  $2^3 = 8$ , Datenbreite = 8 bit, Reset asynchron Low-aktiv,  
bei  $WR = 1$  wird Data geschrieben in Ram ,  
bei enable wird kann Daten am Ausgang gelesen werden)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

entity toplevel is
Port (
enable, clk, reset, WR : in STD_LOGIC;
addr : in STD_LOGIC_VECTOR (2 downto 0);
data_in : in STD_LOGIC_VECTOR (7 downto 0);
data_out : out STD_LOGIC_VECTOR (7 downto 0)
);
end toplevel;

architecture behave of toplevel is

type ram_type is array (0 to 7) of std_logic_vector (7 downto 0);
signal RAM: ram_type;

begin
    process(clk, reset)
    begin
        if reset = '0' then
            data_out <= (others => '0');
        elsif rising_edge(clk) then
            if enable = '1' then
                if WR = '1' then
                    RAM(to_integer(unsigned(addr))) <= data_in;
                end if;
                data_out <= RAM(to_integer(unsigned(addr)));
            end if;
        end if;
    end process;
end behave;
```

## Privat-Beispiel einer LUT (wie bei FPGA baum das Blockdiagramm)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Entity Deklaration
entity LUT_4to1 is
    Port (
        clk : in  STD_LOGIC;           -- Taktsignal
        a   : in  STD_LOGIC_VECTOR(3 downto 0); -- Adress-Eingang (4 Bit)
        DIN  : in  STD_LOGIC_VECTOR(3 downto 0); -- Daten-Eingang (4 Bit, konsistent mit LUT)
        WR   : in  STD_LOGIC;           -- Write-Signal
        WE   : in  STD_LOGIC;           -- Write-Enable-Signal
        q    : out STD_LOGIC_VECTOR(3 downto 0) -- Ausgang (4 Bit, konsistent mit LUT)
    );
end LUT_4to1;

-- Architektur Definition
architecture Behavioral of LUT_4to1 is
    -- 16 Einträge für die LUT (4-Bit-Adresse → 16 mögliche Speicherplätze)
    type LUT_ARRAY is array (0 to 15) of STD_LOGIC_VECTOR(3 downto 0);
    -- signal LUT : LUT_ARRAY := (others => (others => '0')); -- Initialisiert auf 0
    signal LUT : LUT_ARRAY := (
        0 => "0000", -- a = "0000" q = 0
        1 => "0001",
        2 => "0010",
        3 => "0011",
        4 => "0100",
        5 => "0101",
        6 => "0110",
        7 => "0111",
        8 => "1000",
        9 => "1001",
        10 => "1010",
        11 => "1011",
        12 => "1100",
        13 => "1101",
        14 => "1110",
        15 => "1111" -- a = "1111" q = 15
    ); -- Vordefinierte Werte in der LUT
begin
    process (clk)
    begin
        if rising_edge(clk) then
            -- Schreibvorgang
            if WE = '1' and WR = '1' then
                LUT(to_integer(unsigned(a))) <= DIN; -- q-Wert in die LUT schreiben
            end if;
        end if;
    end process;
    -- Lesefunktion: Die aktuelle Adresse gibt den Ausgangswert
    q <= LUT(to_integer(unsigned(a)));
end Behavioral;
```