1.3 Grundbausteine der Digitaltechnik

Zur Vorlesung Rechenanlagen

SS 2019

1.3.1 Darstellung von Zeichen

Wir müssen Zeichen durch Zustände physikalischer Systeme realisieren.

Grundsatzfrage: Wieviele Zustände braucht man dazu mindestens?

Lemma

Sei *A* ein endliches Alphabet mit mindestens 2 Zeichen, und sei $\varphi : A^* \rightarrow B^*$ eine Kodierung. Dann hat *B* mindestens 2 Zeichen.

Beweis: indirekt

Wir nehmen an, dass *#B=1*, d.h. etwa *B={t}* und dass

$$\varphi : A^* \to B^*$$

eine Kodierung ist.

A enthält mindestens 2 Elemente etwa *a* und *b*. Sei nun

$$\varphi(a) = t^n \text{ und } \varphi(b) = t^m \text{ für } n, m \in \mathbf{N}_0.$$

Dann ist aber

$$\varphi(ab) = t^{n+m} = t^{m+n} = \varphi(ba),$$

weswegen φ nicht injektiv ist. ↯

Lemma

Sei *A* ein endliches Alphabet mit mindestens 2 Zeichen, und sei *B={0,1}.* Dann gibt es stets eine Kodierung

$$\varphi : A^* \to B^*$$

Beweis:

Wir betrachten die **Binärkodierung** von nichtnegativen Zahlen über *B* bei fester Wortlänge *k*:

$$u_k: \boldsymbol{B}^k \to \boldsymbol{N}_0 \text{ mit} \qquad u_k(a_0, \ldots, a_{k-1}) = 2^{k-1} \cdot \sum_{i=0}^{k-1} a_i \cdot 2^{-i}$$

u_k ist offenbar injektiv.

Kleinster Wert: $u_k(0, \dots, 0) = 2^{k-1} \cdot \sum_{i=0}^{k-1} 0 \cdot 2^{-i} = 0$

Größter Wert: $u_k(1, \dots, 1) = 2^{k-1} \cdot \sum_{i=0}^{k-1} 2^{-i}$

$$= 2^{k-1} \cdot (2 - 2^{-(k-1)})$$

$$= 2^k - 1$$

$u_k: \boldsymbol{B}^k \to \boldsymbol{N}_0$ ist demnach bijektiv.

Nummeriert man nun $A = \{a_0, \dots a_{k-1}\}$ einfach durch und setzt:

$$\varphi(a_i) = u^{-1}_{\lceil \log(n) \rceil}(i)$$

so erhält man einen Blockcode.

Anmerkungen:

Wir nennen eine solche Kodierung auch (eine!) **Binärkodierung** von *A*. Allerdings gibt es

$$\binom{2^{\lceil \log \#A \rceil}}{\#A} \cdot (\#A)!$$ viele verschiedene Binärkodierungen!

Beide Lemmata zusammen besagen, dass eine zweielementige Menge notwendig und hinreichend ist, um alles zu kodieren. Natürlich kann man die Konstruktion auch für ein größeres *B={0,…,p-1}* durchführen, indem man eine *p*-näre (Basis *p*, Radix *p* Darstellung) benutzt:

$$r_{p,k}(u_0 \cdots u_{k-1}) = p^{k-1} \cdot \sum_{i=0}^{k-1} u_i \cdot p^{-i}$$

Warum tut man dies nicht?

Beispiel:

Sie möchten sich bei Dunkelheit und klarem Wetter über eine Sichtentfernung von 2 km mit Lichtzeichen verständigen. Licht an / Licht aus ist gut wahrnehmbar!

Nun haben beide Parteien sich einen Dimmer zugelegt und vereinbaren 4-wertig nach den Zuständen

- Licht stark
- Licht weniger stark
- Licht schwach
- Licht aus

zu kodieren.

Die Erkennung eines Zeichens wird nun viel schwieriger!

Wie auch immer die Darstellung von Zeichen physikalisch aussehen wird, **Störsicherheit** und **Zuverlässigkeit** sind dabei stets entscheidend.

1.3.2. CMOS Technologie

Bei modernen **MOS** (Metal Oxide Semiconductor) Technologien spielen Transistoren als spannungsgesteuerte Schalter eine herausragende Rolle bei der Entwicklung von Elementarbausteinen.

Das Spiel ist, naiv gesehen, recht einfach:

Interpretiere Spannungen *U(x)* zwischen *x* und dem Bezugspol V_{SS} als Werte aus ***B***

- *U(x)* ~ V_{SS}(= *0* V) interpretiere als *0*
- *U(x)* ~ V_{DD} interpretiere als *1*

Dabei liegt an V_{SS} die Spannung von *0* V zu sich selbst, am Versorgungspol V_{DD} je nach Technologie *1...5*V.

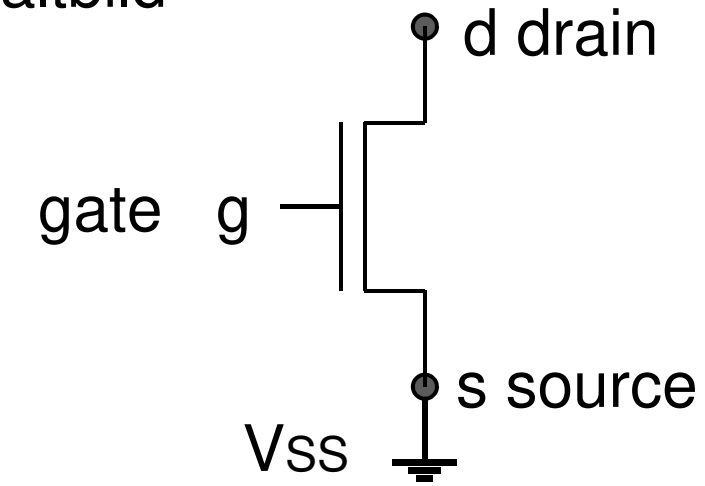
MOS FETs als Schalter

Feldeffekttransistoren (FETs) werden nun so hergestellt, dass sie als spannungsgesteuerte Schalter gegen V_{SS} bzw. V_{DD} arbeiten:

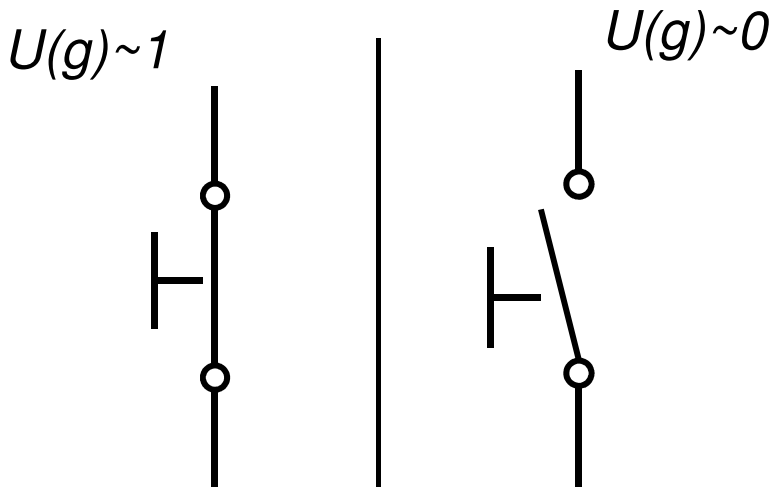
Wir unterscheiden zwei Typen, deren Schalterverhalten wir betrachten wollen:

n-Kanal Transistor:

Schaltbild

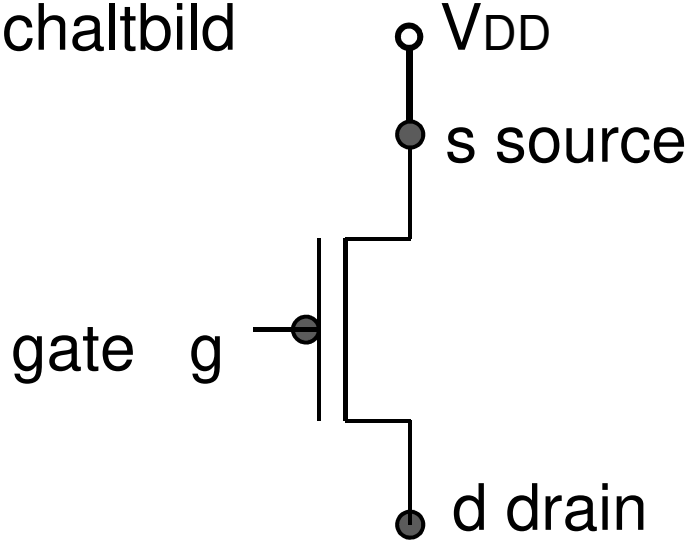


Verhalten

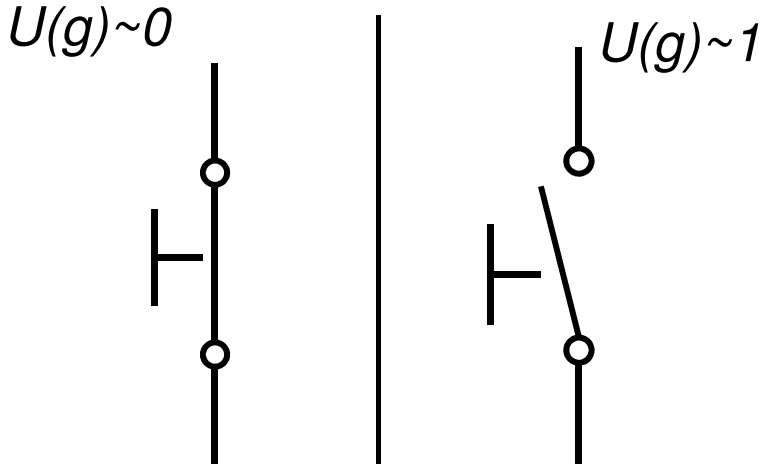


p-Kanal Transistor

Schaltbild



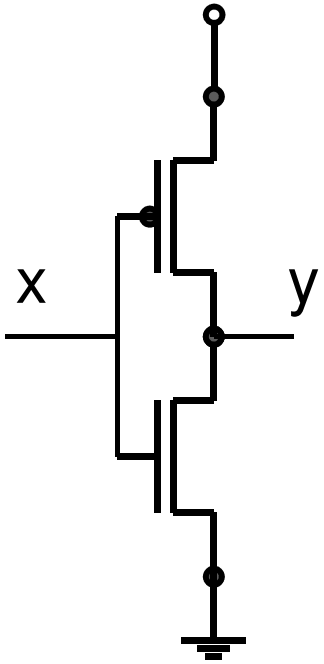
Verhalten



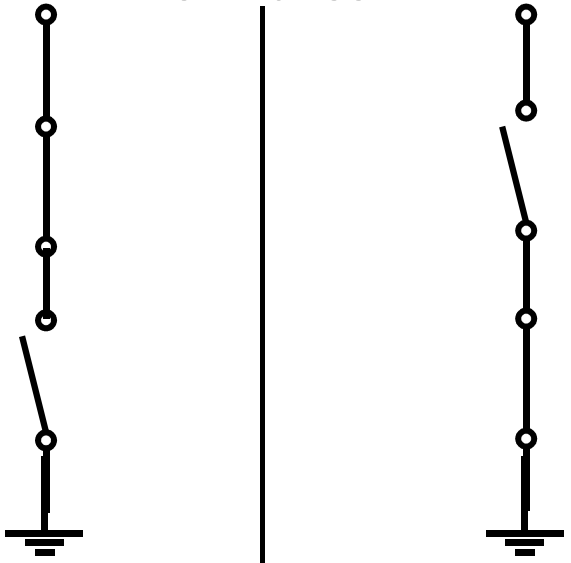
n-Kanal und p-Kanal Transistoren verhalten sich als Schalter **komplementär** zueinander.

Der CMOS Inverter

Wir können nun auf Basis solcher Schalter leicht Funktionen auf ***B*** implementieren. Die wohl einfachste nichttriviale Funktion ist die Negation. Sie wird durch folgende Schaltung, den **CMOS Inverter**, (complementary) realisiert:



Verhalten



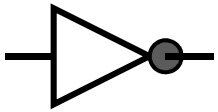
CMOS Inverter ff

Diese einfache Schaltung berechnet also die Funktion:

$$i: \boldsymbol{B} \rightarrow \boldsymbol{B}$$
mit *i(0) = 1* und *i(1) = 0*

D.h. die **Negation**, fasst man *0 (1)* als falsch (wahr) auf.

Wir benutzen fortan als Symbol für diese Schaltung das Schaltzeichen:



Eine Sache ist noch sehr vage formuliert: Wir sagten, der Ausgang y wird verbunden mit V_{SS} (V_{DD}) und wird somit zu 0 (1). Wann aber erreicht der Ausgang tatsächlich einen Pegel, der als 0 (1) interpretierbar ist?

Schaltzeiten eines Inverters

Wir nehmen an, dass sich die Schalter zu einem bestimmten Zeitpunkt wegen einer Änderung des Eingangs sehr schnell umlegen. Eine abrupte Veränderung bewirkt dies nicht, weil die Bausteine nicht ideal sind. In heutiger CMOS Technologie dominiert:

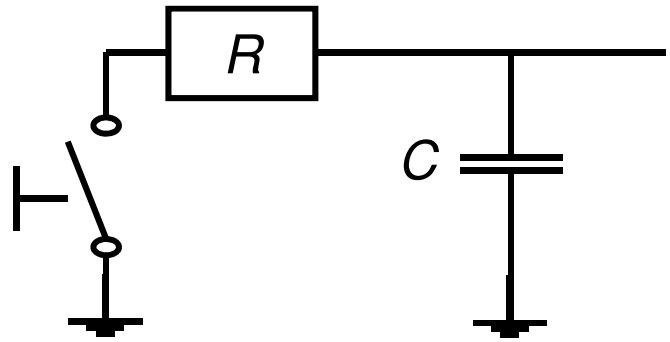
- Der **Widerstand *R*** des Transistors und der Leitungen

$$Ohmsches\ Gesetz\ U(t) = R \cdot I(t)$$

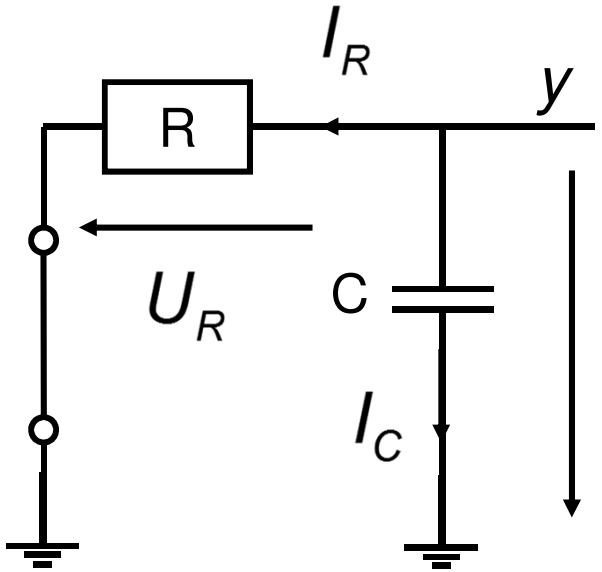
- Die **Kapazität *C*** der Gates der Folgetransistoren und der Leitungen

$$Kondensatorgleichung\ Q(t) = C \cdot U(t)$$

Modell:



Analyse



Wir nehmen an, dass sich zum Zeitpunkt *t=0* der Schalter schließt, und der Ausgang *y* auf V_{DD} aufgeladen war.

$I_R(t) + I_C(t) = 0$ (Knotenregel)

$-U_R(t) + U_y(t) = 0$ (Maschenregel)

$U_R(t) = R \cdot I_R(t)$ (Ohm‘sches Gesetz)

$I_C(t) = \frac{dQ_C(t)}{dt} = C \cdot \frac{dU_y(t)}{dt}$ (Kondensatorgleichung $Q(t) = C \cdot U(t)$)

Analyse ff

Nun gilt für den zeitlichen Verlauf der Ausgangsspannung:

$$U_y(t) = U_R(t)$$

$$= R \cdot I_R(t)$$

$$= -R \cdot I_C(t)$$

$$= -R \cdot C \cdot \frac{dU_y(t)}{dt}$$

$U_R(t) = R \cdot I_R(t)$

$I_R(t) + I_C(t) = 0$

$I_C(t) = C \cdot \frac{dU_y(t)}{dt}$

Lösung dieser Gleichung ist ein Verlauf der Form:

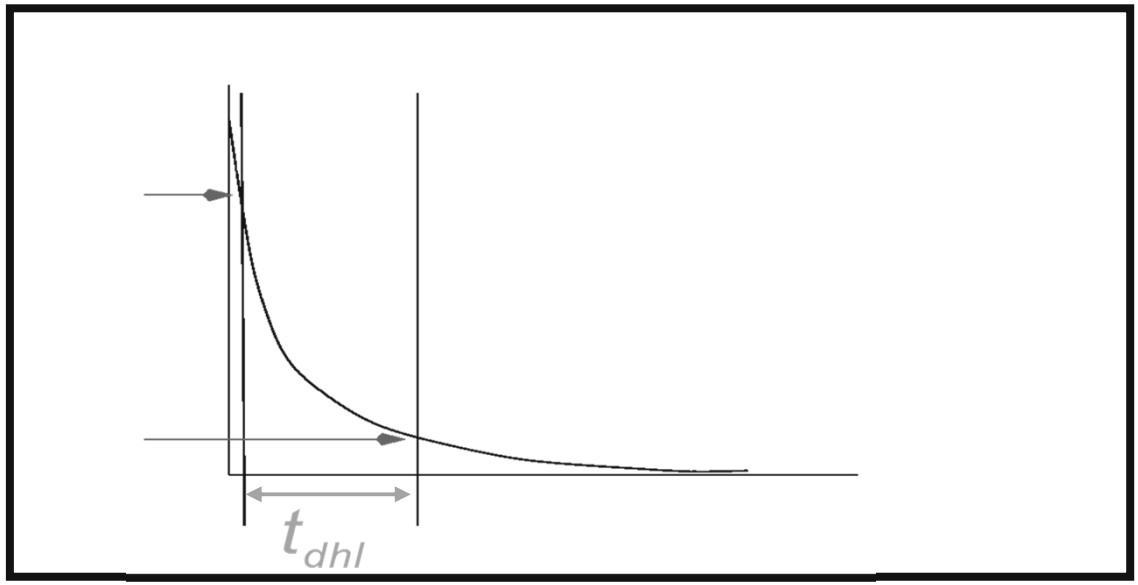
$$U_y(t) = A \cdot e^{-\frac{t}{R \cdot C}}$$

A bestimmt sich aus der Anfangsbedingung

$$V_{DD} = U_y(0) = A \cdot e^{-\frac{0}{R \cdot C}} = A$$

Analyse ff

Wir nennen die Zeit, die verstreicht, bis ein fallender (steigender) Ausgang wieder als Wert über 0 (1) interpretierbar ist, die **Abfallzeit (Anstiegszeit)** des Ausgangs. Sie wird häufig definiert als die Zeit, die verstreicht, um den Ausgang von 0.9 V_{DD} auf 0.1 V_{DD} (0.1V_{DD} auf 0.9 V_{DD}) zu bringen:



Analyse ff

Sei t_H der Zeitpunkt mit $U_y(t_H) = 0.9\, V_{DD}$

$$V_{DD} \cdot e^{-\frac{t_H}{R \cdot C}} = 0.9\, V_{DD} \Leftrightarrow t_H = -ln0.9 \cdot RC$$

Sei entsprechend t_L der Zeitpunkt mit $U_y(t_L) = 0.1\, V_{DD}$

$$V_{DD} \cdot e^{-\frac{t_L}{R \cdot C}} = 0.1\, V_{DD} \Leftrightarrow t_L = -ln0.1 \cdot RC$$

Damit ist

$$\begin{aligned} t_{dhl} &= t_L - t_H \\ &= (ln0.9 - ln0.1) \cdot RC \\ &= ln9 \cdot RC \approx \boxed{2RC} \end{aligned}$$

Analyse ff

R,C sind Materialgrößen. An ihnen muss man drehen, wenn man das Zeitverhalten eines Bausteins beeinflussen will.

Man erhält durch eine analoge Rechnung auch die Anstiegszeit. Sind Anstiegs- und Abfallzeit wie in dieser stark vereinfachten Modellrechnung gleich, definiert man die **Verzögerungszeit** als:

$$t_d = \frac{t_{dlh}}{2} = \frac{t_{dhl}}{2} = RC$$

Anmerkung: In der Praxis (CMOS Technologie) gilt $t_{dlh} \approx 2 \cdot t_{dhl}$

1.3.3 WüHDL-Beschreibung eines Bausteins

Den Inverter haben wir nun als ersten Baustein kennengelernt. Wir wollen nun sehen, wie man sein Verhalten in WüHDL beschreiben kann:

Eine Bausteindefinition besteht aus

- genau einer **Entity**
  - Spezifikation der Schnittstelle
- mindestens einer **Architecture**
  - Spezifikation des Verhaltens, durch
    - Programmcode (Prozesse), d.h. man beschreibt das Verhalten mit dem „Repertoire“ einer gewöhnlichen Programmiersprache (Ada ähnlich in WüHDL), oder
    - Strukturbeschreibung, d.h. man definiert das Verhalten durch Instanziierung und Verbindung anderer Komponenten (Schaltkreise).

Entities -- die Schnittstellen von Komponenten

Syntax:

<entity_declaration>

::= **ENTITY** <entity_identifier> **IS**

[<generic_declaration>] [<port_declaration>]

END [**ENTITY**] [<entity_identifier>]**;**

<port_declaration>

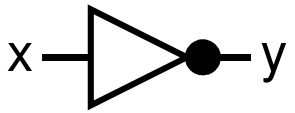
::= **PORT (** <port_list> **:** [<mode>] <type_id>

{**;**<port_list> **:** [<mode>] <type_id>}* **);**

<generic_declarations> erläutern wir später.

Beispiel: Deklaration eines Inverterbausteins

```
ENTITY inverter IS
        PORT (x: IN bit; y: OUT bit);
    END inverter;
```



Entities: Port-Arten

Es gibt in VHDL 5 Port Modes von denen wir in WüHDL nur 3 benutzen:

- **IN** -- der Port ist ein Eingangsport, d.h. die Komponente selbst verändert ihn nicht.
- **OUT** -- der Port ist ein Ausgangsport, d.h. die Komponente selbst verändert sein Signal, verhält sich aber unabhängig von den Werten seines Signals.
- **INOUT** -- sein Signal darf von der Komponente verändert werden, kann aber auch von außen verändert werden und Einfluss auf das Verhalten der Komponente nehmen.
- BUFFER -- Ausgangsport, nur die Komponente darf sein Signal verändern, sie darf sich aber auch abhängig von Werten seines Signals verhalten.
- LINKAGE -- unbekannt, ob Eingang oder Ausgang

Architekturen -- die Definition der Komponenten

Syntax: -- vorläufig

<architecture_declaration>

::= **ARCHITECTURE** <architecture_identifier>

OF <entity_identifier> **IS**

[<local_declarations>]

BEGIN [<statements>]

END [**ARCHITECTURE**] [<architecture_identifier>]**;**

<local_declarations>

::={ **USE** <...> | **TYPE** <...> | **CONSTANT** <...>

| **SIGNAL** <...> | **VARIABLE** <...> | **COMPONENT** <...>

| **FUNCTION** <...> | **PROCEDURE** <...> }*

<statements> ::= { <process_statement> | <conc_sig_assign>

| <comp_instantiation> }*

Architekturen -- die Definition der Komponenten

Definitionsmöglichkeiten

Man hat schon aus der Syntax ersichtlich mehrere Möglichkeiten, das Verhalten von Komponenten zu definieren. In WüHDL sehen wir folgende Optionen vor:

- Beschreibung durch Prozesse
- Beschreibung durch nebenläufige Signalzuweisungen (concurrent signal assignments)
- Beschreibung durch Instanziierung und Verbindung von Komponenten.

Die erste Variante kann man als Verhaltensdefinition auffassen, während die letztere eine Strukturdefinition ist (Aufbau über Instanzen vorhandener Komponenten). Die zweite Variante kann man sowohl als Verhaltens als auch als Strukturdefinition auffassen (vgl. spätere Kapitel). Wir klammern sie vorläufig aus.

Architekturen -- die Definition durch Prozesse

Prozesse sind sequentiell ablaufende Programme, ergänzt durch einige spezielle Anweisungen, zu denen wir noch kommen. Innerhalb eines Prozesses sind sichtbar:

- alle Ports vom Modus IN, INOUT als Werte zum aktuellen Zeitpunkt
- alle lokal in der Architecture definierten Signale, Variablen, Prozeduren, Funktionen, ...
- alle Ports vom Modus OUT, INOUT und alle lokal definierten Signale als Ziel einer Signalzuweisung.

Wir betrachten zunächst ausschließlich Prozesse mit einer **Sensitivitätsliste**. Eine Sensitivitätsliste enthält nur Ports der entity vom Modus IN, INOUT und ggf. lokale Signale. Ändert sich ein Signal der Sensitivitätsliste, dann wird das im <process_statement> aufgeführte Programmstück bis zur Terminierung ausgeführt.

-- Vorsicht: Terminiert das Programmstück nicht, hängt sich die Simulation bei der Ausführung auf!

Architekturen -- Beispiel zu Prozessen

Folgendes triviale Programmstück beschreibt das Verhalten unseres CMOS Inverters durch ein process statement:

```
ARCHITECTURE behavior1 OF inverter IS
  CONSTANT tdhl: TIME := 25 ps;
  CONSTANT tdlh: TIME := 35 ps;
      -- Festlegung der Verzögerungzeiten durch lokale
      -- Konstanten. Wird in der Praxis flexibler gelöst.
  BEGIN
  PROCESS (x)  -- sensitiv auf  IN, INOUT Ports der entity
      BEGIN -- Signalzuweisungen
            IF x = '0'
                  THEN   y <= not x  AFTER tdlh;
                  ELSE   y <= not x  AFTER tdhl;
            END IF;
      END PROCESS;
END behavior1;
```

Architekturen -- Prozesse

Prozesse bestehen im Allgemeinen aus Anweisungsfolgen, die das ganze Repertoire einer klassischen, imperativen Programmiersprache umfassen (Schleifen, Prozedur- und Funktionsaufrufe, …). Die einzigen Anweisungen, die einen Prozess zum Terminieren bringen, sind sogenannte WAIT Anweisungen.

Vorsicht: Der Abschluss END [PROCESS] der Deklaration **terminiert nicht** den Prozess, sondern bezeichnet nur das Ende der Deklaration. Der Prozess startet bei Beendung wieder von vorne durch!

Die Angabe einer Sensitivitätsliste ist äquivalent zu einer WAIT Anweisung als letzter Anweisung des Prozesses, d.h. im Grunde ist eine Prozess Anweisung

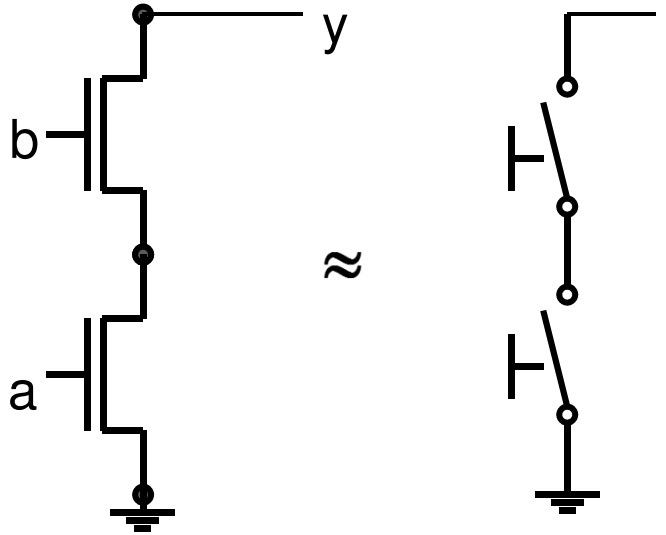
PROCESS (x_1,…,x_k) BEGIN … -- Anweisungen END PROCESS;	zu interpretieren als	LOOP … -- Anweisungen WAIT ON x_1,…,x_k END LOOP;

1.3.4 Einfache Verknüpfungen

Beim Inverter berechneten wir die Inversion durch Schalten einer leitenden Verbindung zwischen Ausgang und V_{DD} bzw. V_{SS}. Dies muss stets komplementär, d.h. unter gegenseitigem Ausschluss geschehen! (sonst Kurzschluss)

Betrachte nun folgende Schaltung:

(n-Kanal Serienschaltung)

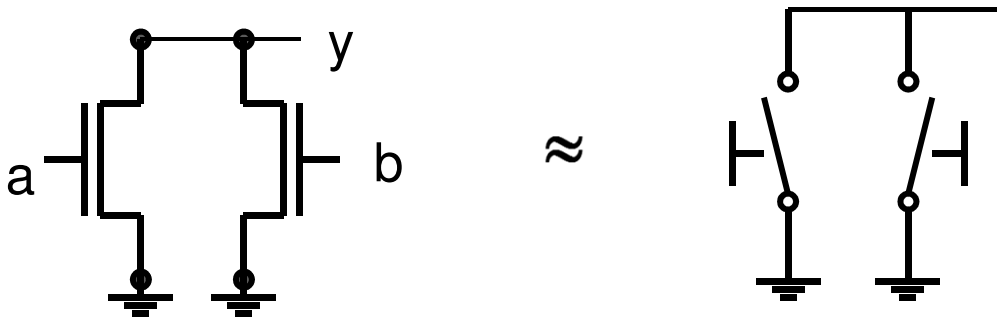


Offensichtlich entsteht eine Verbindung mit V_{SS} dann und nur dann, wenn beide Schalter leitend sind, d.h. wenn

a=1 **und** b=1

Einfache Verknüpfungen -- ff

Betrachten wir nun eine Parallelschaltung zweier n-Kanal Transistoren gegen Vss:



Offensichtlich entsteht eine Verbindung mit Vss dann und nur dann, wenn mindestens einer der beiden Schalter leitend ist, d.h. wenn a=1 **oder** b=1.

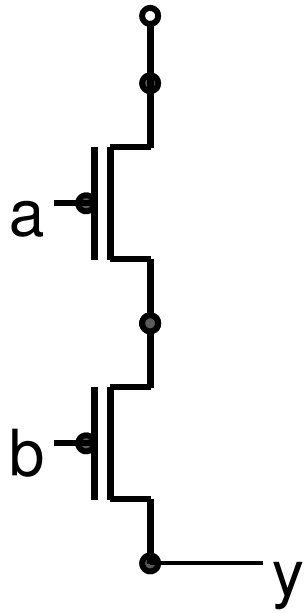
Interpretieren wir nun x=1 einfach als x bzw. „x ist wahr“ und x=0 einfach als nicht x bzw. „x ist falsch“, dann gilt:

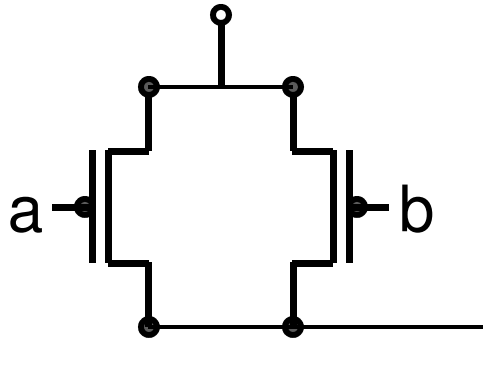
NAND und NOR Verknüpfung ff

Serienschaltung gegen Vss: **nicht y ⇔ a und b**

Parallelschaltung gegen Vss : **nicht y ⇔ a oder b**

Analog erhalten wir für p-Kanal Transistor Schaltungen gegen V_{DD}:





y=1 ⇔ a=0 **oder** b=0

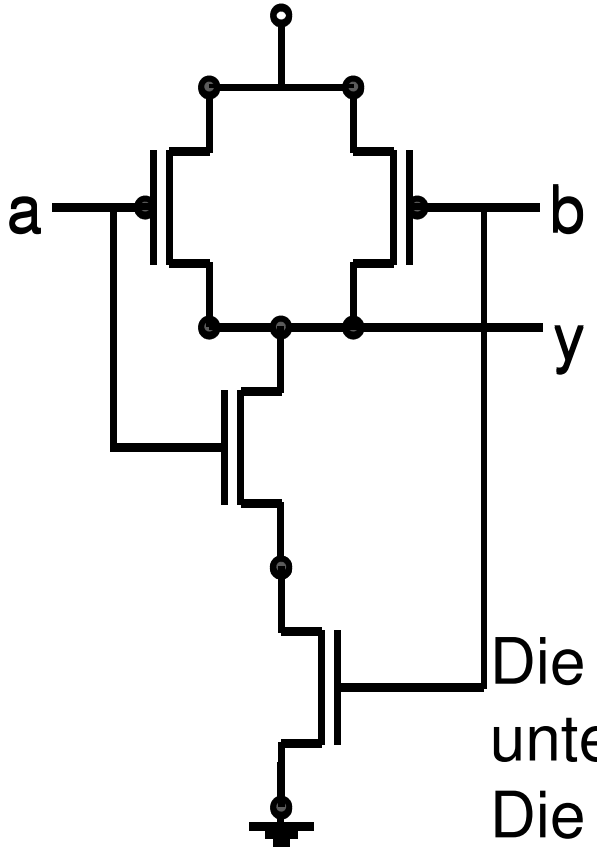
Parallelschaltung: **y ⇔ nicht(a und b)**

Serienschaltung: **y ⇔ nicht (a oder b)**

y=1⇔ a=0 **und** b=0

Einfache Verknüpfungen ff

Demnach liefert folgende Schaltung:



y ⇔ nicht(a und b)

(p-Kanal Parallelschaltung)

nicht y ⇔ a und b

(n-Kanal Serienschaltung)

Die leitenden Verbindungen bestehen unter **gegenseitigem Ausschluss**. Die Verschaltungen im n-Kanal, p-Kanal Teil sind **dual** zueinander (Serien ⇔ Parallel).

NAND und NOR Verknüpfung

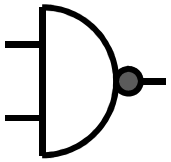
Die Schaltung berechnet also die Funktion ***nand***

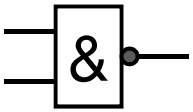
nand	*b* / *a*	0	1
	0	1	1
	1	1	0

$nand: \boldsymbol{B} \times \boldsymbol{B} \rightarrow \boldsymbol{B}$

mit $nand(a, b) := \overline{a \cdot b}$

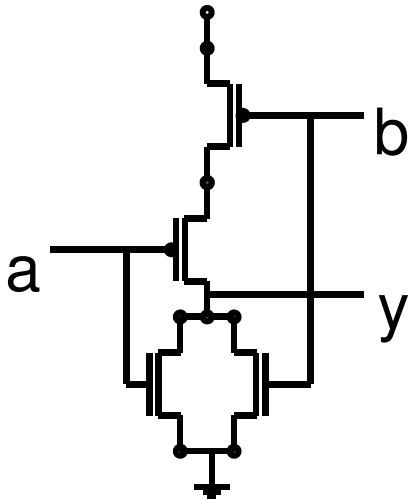
Schaltsymbol





(DIN)

Analog überlegt man sich, dass die Schaltung

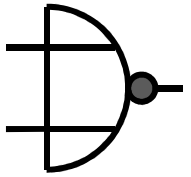


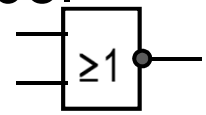
die Funktion **nor** berechnet

mit $nor(a, b) := \overline{a \vee b}$

nor	*b* / *a*	0	1
	0	1	0
	1	0	0

Schaltsymbol





(DIN)

Beschreibung in WüHDL

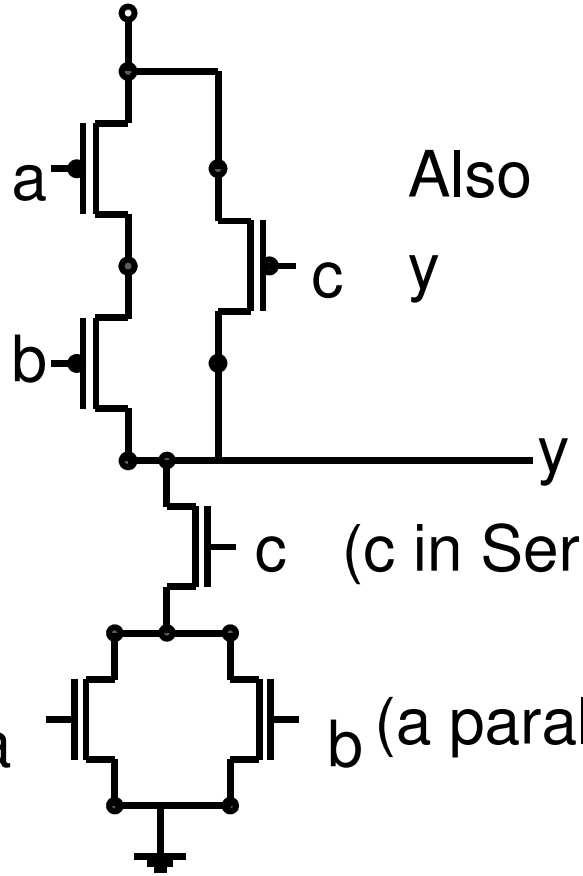
Wir können diesen Baustein ebenso leicht in WüHDL beschreiben:

```
ENTITY nand2 IS
        PORT (x,y: IN bit; z: OUT bit);
    END nand2;

    ARCHITECTURE behavior1 OF nand2 IS
        CONSTANT tdhl: TIME := 35 ps;
        CONSTANT tdlh: TIME := 50 ps;
            -- vgl. Inverter
    BEGIN
        PROCESS (x,y)  -- sensitiv auf  IN, INOUT Ports der entity
        BEGIN -- Signalzuweisungen
            IF x = '0' OR y = '0'
                THEN  z <= '1'  AFTER tdlh;
                ELSE   z <= '0'  AFTER tdhl;
            END IF;
        END PROCESS;
    END behavior1;
```

CMOS Komplexgatter

Man kann dieses Spiel beliebig (technisch hat es Grenzen) weitertreiben, indem man zueinander duale Serien/Parallel Netzwerke von Transistoren bildet:



Also

y = nicht (a oder b) oder nicht c

= nicht((a oder b) und c)

(c in Serie zu a,b da oben parallel)

(a parallel b da oben in Serie)

CMOS Komplexgatter ff

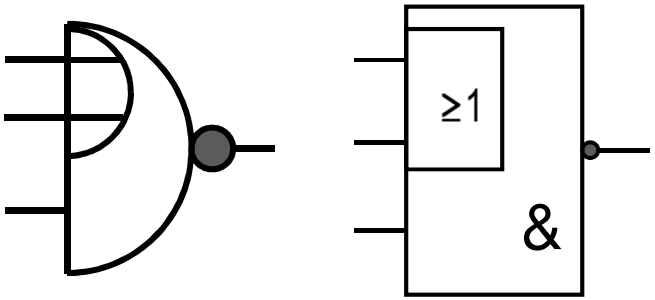
Die Schaltung berechnet also die Funktion

$$f(a, b, c) := \overline{(a \vee b) \cdot c}$$

Man nennt eine solche Schaltung, da sie nicht nur einfache Verknüpfungen berechnet, ein **Komplexgatter**.

(ein „oai21“)

Schaltsymbol



(DIN)

1.3.5 Speicherelemente

Was ist ein Speicher für ein Zeichen aus ***B*** ?

Allgemein ist ein Speicher für *k* Zeichen aus einer Menge *O* modelliert durch

$$\boldsymbol{F}([0{:}\,k-1], O)$$

-- die Menge der partiellen Funktionen von *[0:k-1]* nach *O*

Für ein einziges Zeichen aus ***B*** ist ein Speicherbaustein damit modellierbar durch

$$\boldsymbol{F}(\{0\}, \boldsymbol{B})$$

Es gibt also genau 3 Speicherzustände:

$$\Omega, \boldsymbol{s_0}, \text{ mit } s_0(0) = 0, \text{ und } \boldsymbol{s_1}, \text{ mit } s_1(0) = 1$$

Klammern wir den undefinierten Zustand Ω aus, gibt es also zwei definierte Zustände, das Speichern einer 0 und einer 1.

Speicherelemente

Wir brauchen also einen Baustein, der zwei Zustände annehmen kann. Er sollte auch kontrolliert zwischen diesen Zuständen hin und her schalten können. Der Zeitpunkt einer Zustandsänderung soll kontrollierbar sein, etwa durch das Ticken einer Uhr.

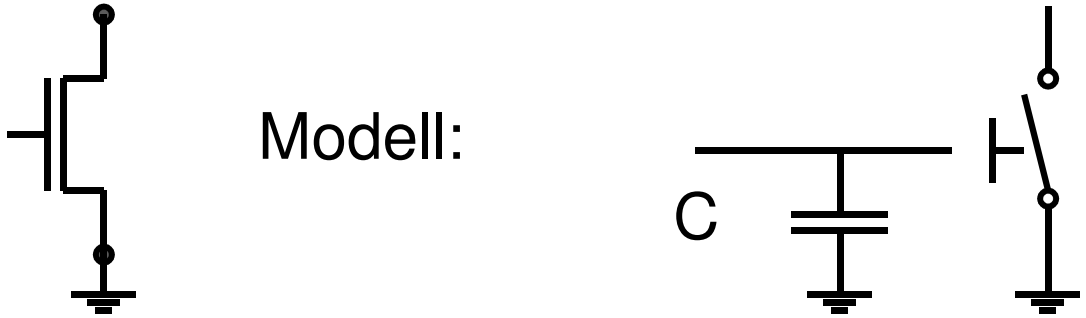
Wie realisiert man solche Bausteine?

Wir werden diese Frage auf 2 Wegen beantworten:

- Realisierung in CMOS Technologie
- Realisierung durch Logikverknüpfungen
  (ohne konkretes Technologiewissen)

CMOS Speicherelemente

Wir können in CMOS Schaltungen die Kapazität der Transistorgates und Leitungen zum Speichern von Informationen ausnutzen, da in das Gate nur ein winziger Leckstrom abfließt (RC Glied mit riesigem R).



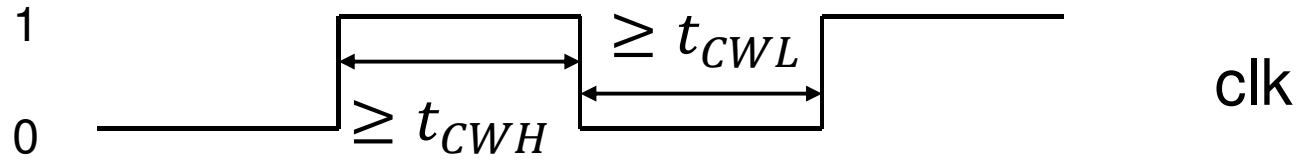
Je nach Ladung der Kapazität *C* haben wir die Zustände:

- *C* aufgeladen: dann ist U(g) ~ 1, Schalter ist geschlossen
- *C* entladen: dann ist U(g) ~ 0, Schalter ist offen.

Entsprechendes gilt dual für p-Kanal Transistoren.

CMOS Speicherelemente

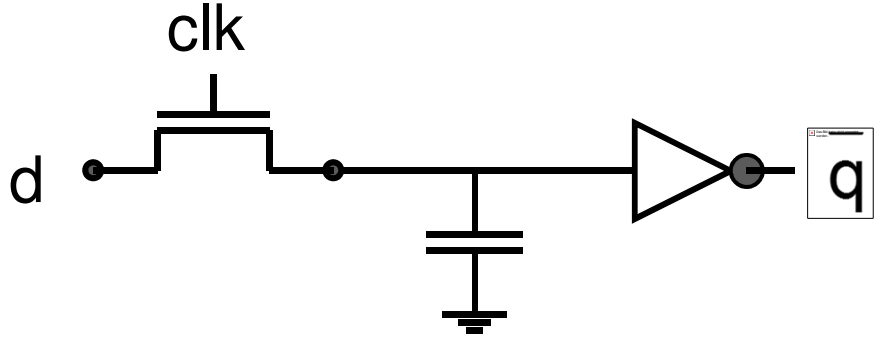
Wir nehmen nun noch einen weiteren Anschluss **clk** hinzu, mit dessen Hilfe wir die Änderung von Zuständen kontrollieren wollen. Liegt der Anschluss für eine hinreichend lange Zeit t_{CWH} auf 1, wollen wir dies als einen Übernahmetick interpretieren. Um zwei Ticks voneinander unterscheiden zu können müssen sie durch eine 0 Phase voneinander getrennt sein, die mindestens t_{CWL} dauert.



Wir nennen ein Diagramm, das den zeitlichen Verlauf von Spannungen mit Bedingungen daran wiedergibt, auch ein **Zeitdiagramm**.

CMOS Speicherelemente

Mit Hilfe des neuen Anschlusses braucht man nun nur noch das Laden der Eingangskapazität eines Inverters zu kontrollieren und kommt so zu folgender einfachen Lösung:



Bei jedem Tick (*clk=1*) wird *d* mit dem Eingang des Inverters verbunden und lädt die Kapazität auf. ($\bar{q} = \bar{d}$)
Wird *clk = 0* ist *d* und die Kapazität entkoppelt, der eingeschriebene Wert bleibt unabhängig von *d* erhalten, sozusagen verriegelt. Wir nennen diesen Baustein daher ein **dynamisches Daten-Latch** (D-Latch).

CMOS Speicherelemente

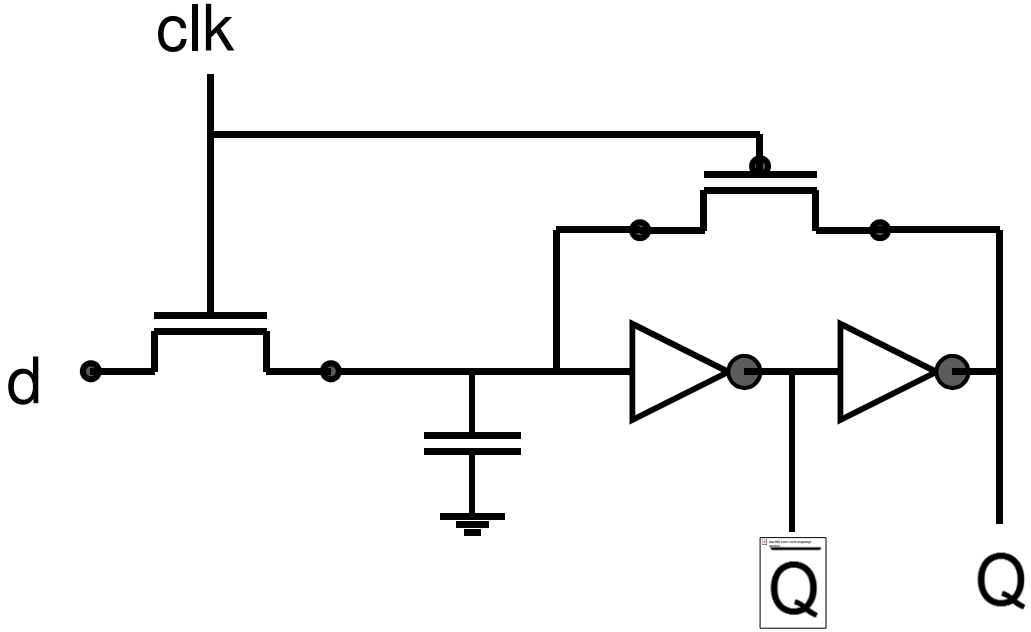
Problem: Durch die Leckströme fließt die eingespeicherte Ladung am Eingang des Inverters langsam ab. Dauert die 0 Phase von *clk* zu lange, kann dies zu einer ungewollten Zustandsänderung führen. Daher nennt man wegen der zusätzlichen Zeitbedingung das Latch **dynamisch**.

Auswege:

- Stelle sicher, dass die Kapazität in regelmäßigen Abständen aufgefrischt wird (refresh Bedingung).
- Ergänze die Schaltung um eine Halteschaltung

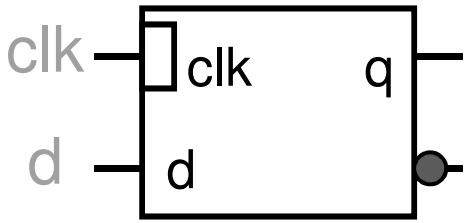
statisches D-Latch

Statisches Latch (CMOS)



Die Halteschaltung hält in der 0 Phase den Wert *q* über einen leitenden p-Kanal Transistor beliebig lange. Wir nennen diese Schaltung einfach ein **statisches D-Latch** (D für Data) oder kurz D-Latch.

Schaltbild:



Beschreibung in WüHDL

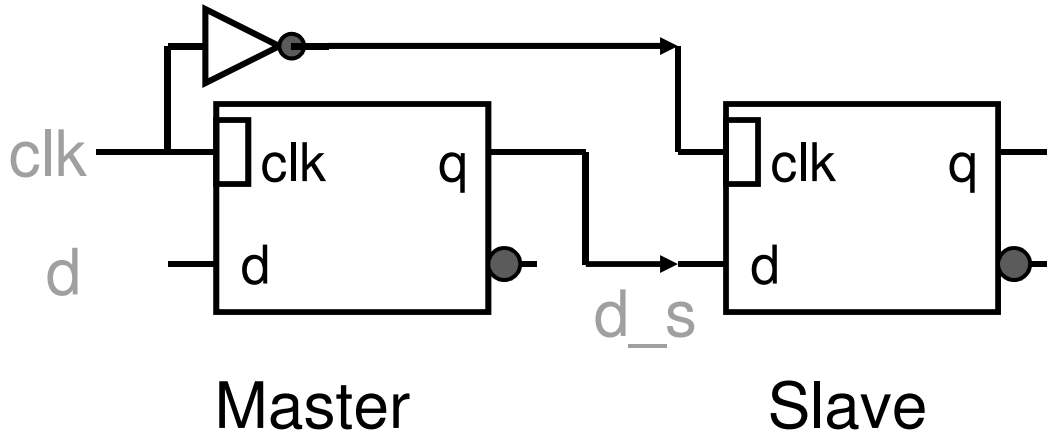
Wir können diese Bausteine ebenso leicht in WüHDL beschreiben:

```
ENTITY sdlatch IS
        PORT (clk,d: IN bit; q: OUT bit;
                qz: OUT bit := '1' ); -- explizite Initialisierung auf NOT q erforderlich!
    END sdlatch;

    ARCHITECTURE behavior1 OF sdlatch IS
        CONSTANT tdhl: TIME := 50 ps;
        CONSTANT tdlh: TIME := 60 ps;   -- Anmerkung vgl. Inverter
    BEGIN
        PROCESS (clk,d)  -- sensitiv auf  IN, INOUT Ports der entity
        BEGIN
                IF clk = '1'
                    THEN IF d = '0'
                            THEN qz <= '1'  AFTER tdlh; q <= '0' AFTER tdlh+tdhl;
                            ELSE qz <= ‘0'  AFTER tdhl; q  <= ‘1' AFTER tdhl+tdlh;
                        END IF;
                END IF;
        END PROCESS;
    END behavior1;
```

Master/Slave Latch

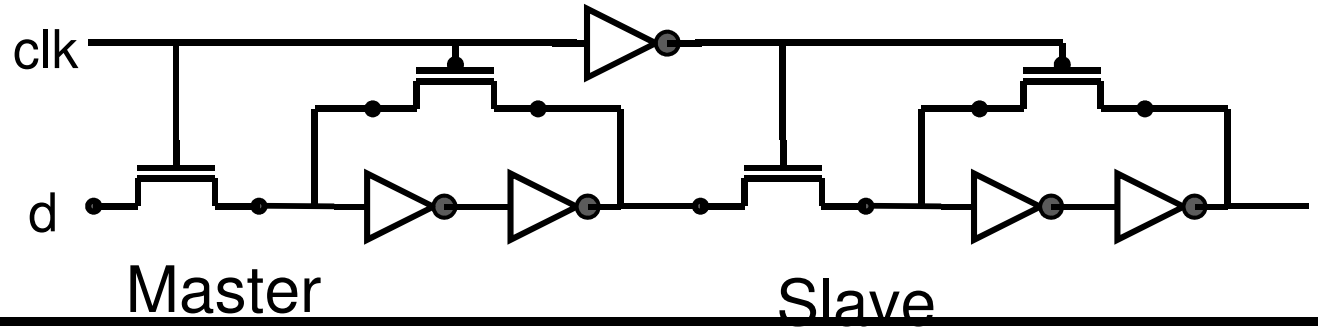
Häufig macht die Gefahr, dass sich *d* in der 1 Phase des Taktes *clk* ändern könnte, (dann ist ja *d* **transparent** mit *q* verbunden) den Entwurf schwierig. Man möchte daher die Zeit, in der sich *d* nicht ändern darf, so kurz wie möglich halten. Dies erreicht man, indem man den Übernahmezeitpunkt eng um die 1/0 Flanke des Taktes *clk*, wie in folgender Schaltung, legt:



Master/Slave Latch ff

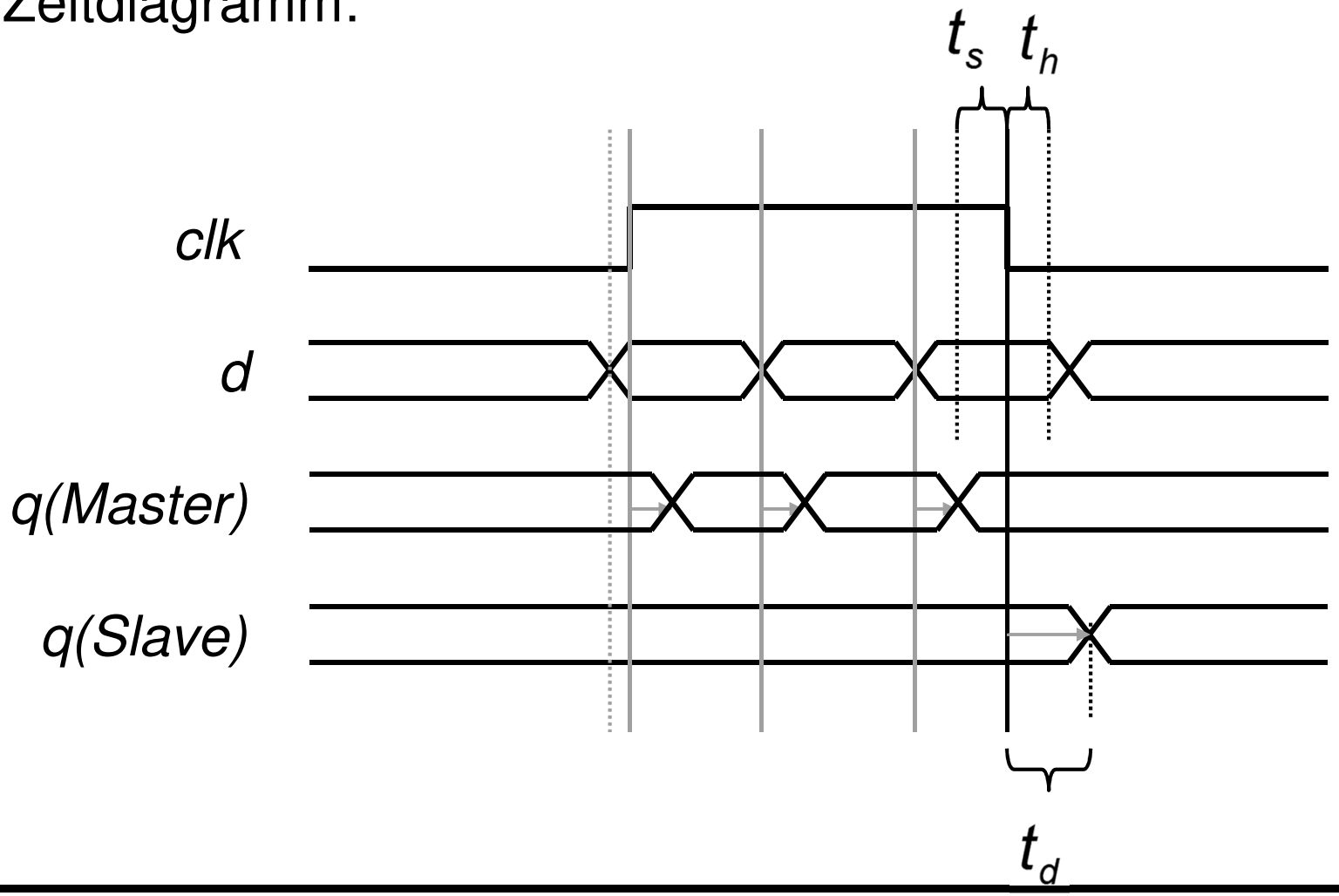
In der 1 Phase des Taktes schaltet das Master Latch *d* transparent nach *q_m,* das Slave Latch ist aber verriegelt. In der 0 Phase verriegelt der Master und der Slave schaltet den zu Beginn der 0 Phase an *q_m = d_m* liegenden Wert transparent nach *q*.

Man muss also nur noch sicherstellen, dass *d* und *q_m* in einem kurzen Zeitraum um die 1/0 Flanke stabil sind, um einen korrekten Zustandsübergang mit der Flanke sicherzustellen.



Master/Slave Latch: Timing

Verdeutlichung des Übernahmeverhaltens am Zeitdiagramm:



Master/Slave Latch: Timing ff

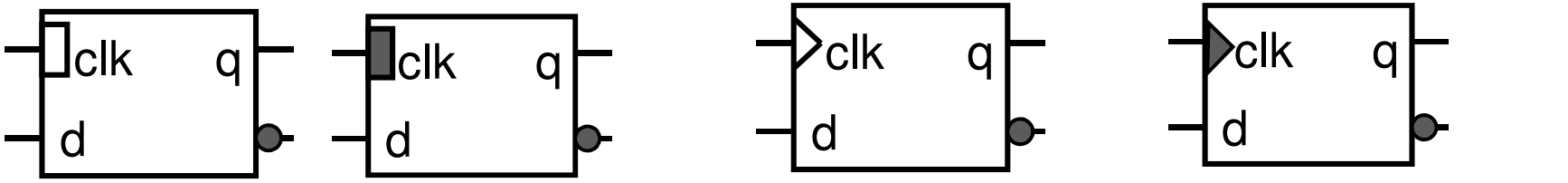
Meist gibt man 3 charakteristische Schaltzeiten an:

- die **delay** Zeit $\mathbf{t_d}$, bis der Ausgang übernimmt (ggf. t_{dhl}, t_{dlh})
- die **setup** Zeit $\mathbf{t_s}$, die *d* vor der Flanke stabil sein muss
- die **hold** Zeit $\mathbf{t_h}$, die *d* nach der Flanke stabil sein muss

Wir nennen ein solches Latch auch

flankengesteuertes D-Latch oder **D-FlipFlop**

Man kann Latches zustands- oder flankengesteuert nach verschiedenen Taktzuständen oder Flanken konstruieren. Wir zeigen dies an durch entsprechende Schaltsymbole:



1-Phasen- 0-Phasen- 0/1-Flanken- 1/0-Flankengesteuert

WüHDL-Beschreibung eines D-FlipFlops

Wir haben bisher die Bausteine stets definiert durch

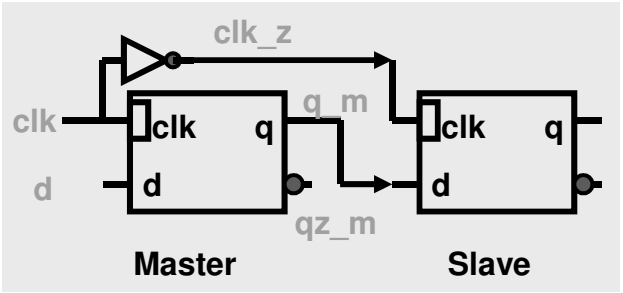
- eine **Entity**: Spezifikation der Schnittstelle
- eine **Architecture**: Spezifikation des Verhaltens, durch Programmcode (Prozesse).

Das flankengesteuerte D-Latch haben wir aber durch Benutzung von Instanzen anderer Bausteine (statische D-Latches) definiert. Also sollte man die Definition des Bausteins in WüHDL ebenso durch die schon erwähnte zweite Möglichkeit der

- Strukturbeschreibung, d.h. man definiert das Verhalten durch Instanziierung und Verbindung anderer Komponenten (Schaltkreise)

vornehmen.

WüHDL-Beschreibung eines D-FlipFlops




```
ENTITY dlatch IS
        PORT (clk,d: IN bit; q,qz: OUT bit);
    END dlatch;

    ARCHITECTURE structure OF dlatch IS
        COMPONENT sdlatch PORT (clk,d: IN bit; q,qz: OUT bit);
        END COMPONENT;
    COMPONENT inverter PORT (x: IN bit;y: OUT bit);
    END COMPONENT;
        SIGNAL q_m,qz_m,clk_z: bit;
    BEGIN
    -- Definition der Schaltung durch Instanziierungen
      INV: inverter PORT MAP (x => clk, y => clk_z);
      Master: sdlatch PORT MAP (clk => clk, d => d,q => q_m, qz => qz_m);
      Slave: sdlatch PORT MAP (clk => clk_z, d => q_m, q => q, qz => qz);
    END structure;
```

Problem: Wir haben keinerlei Überwachung der Einhaltung von Setup und Hold Zeiten!

Überwachung von Bedingungen

WüHDL sieht dazu sogenannte **Assertions** vor. Diese bestehen aus einer Bedingung, die zu überwachen ist, einer Fehlermeldung die bei Verletzung auszugeben ist, und einer Schwere des Fehlers.

Wir fügen einfach unserer Definition noch einen Überwachungsprozess hinzu:

```
ARCHITECTURE structure OF dlatch IS
    … CONSTANT setup:TIME := 40ps;  CONSTANT hold: TIME := 20ps;
BEGIN ...
    Setup_hold_check: PROCESS(clk,d)
    BEGIN IF clk'event AND clk = '0'
                THEN -- aktive Flanke der Clock, Setup Check
                    ASSERT d'stable(setup) REPORT "Setup-Violation "
                    SEVERITY error;
                ELSE IF d'event AND clk = '0'
                        THEN -- Datenänderung nach der aktiven Flanke
                           ASSERT clk'last_event > hold
                           REPORT "Hold-Violation " SEVERITY error;
                        END IF;
            END IF;
    END PROCESS setup_hold_check;
END structure;
```

Attribute von Signalen

WüHDL erlaubt die Definition von Attributen für alle möglichen Objekte. Zu einem Signal ***sig*** gibt es folgende vordefinierte Attribute, die sehr nützlich sind:

- ***sig***'stable(T) ist vom Typ Boolean und hat genau dann den Wert TRUE, wenn der Wert von ***sig*** sich in den letzten T Zeiteinheiten nicht verändert hat.
- ***sig***'event ist vom Typ Boolean und liefert genau dann den Wert TRUE, wenn zu dem aktuellen Zeitpunkt eine Änderung auf ***sig*** erfolgt. Vorsicht: Da Änderungen auch mit Verzögerung 0 gemacht werden können, ist ***sig*** 'event nicht zum gesamten Zeitpunkt TRUE sondern nur in der Iteration, in der das Ereignis bearbeitet wird.
- ***sig***'last_event ist vom Typ TIME und gibt die Zeit seit der letzten Änderung auf ***sig*** an.
- ***sig***'last_value ist vom gleichen Typ wie ***sig***. Es gibt den Wert von ***sig*** vor der letzten Änderung an.

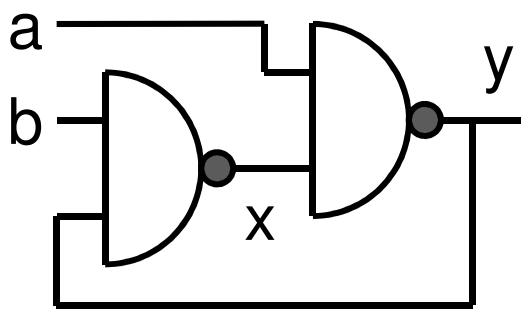
-- VHDL sieht noch mehr Attribute vor!

1.3.6 Bistabile Schaltungen

Wie konstruiert man Speicherglieder, wenn man lediglich über Logikgatter verfügt und keine spezielleren Eigenarten der Technologie auszunutzen weiß?

Idee: Konstruiere Bistabile Schaltungen aus Gattern.

Beispiel: Analysiere folgende Schaltung

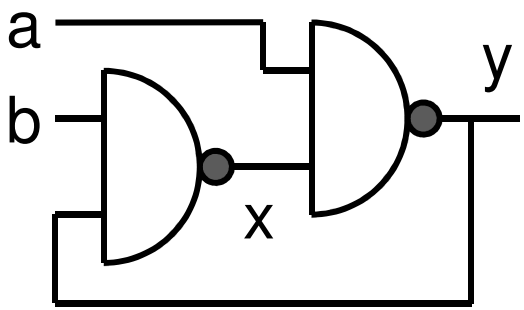


Es ist: $x = \overline{b \cdot y}$

und somit $y = \overline{a \cdot \overline{b \cdot y}} = \bar{a} \vee b \cdot y$

Bistabile Schaltungen

Falldiskussion:



$$y = \overline{a \cdot \overline{b \cdot y}} = \bar{a} \vee by$$

a=b=1: $y = \bar{a} \vee by = \bar{1} \vee 1y = 0 \vee y = y$ **hält y**

a=0, b=1: $y = \bar{0} \vee 1y = 1 \vee y = 1$ **setzen**

a=1, b=0: $y = \bar{1} \vee 0y = 0 \vee 0 = 0$ **rücksetzen**

a=b=0: $y = \bar{0} \vee 0y = 1 \vee 0 = 1$ **unbenutzt**

Basis R/S FlipFlop

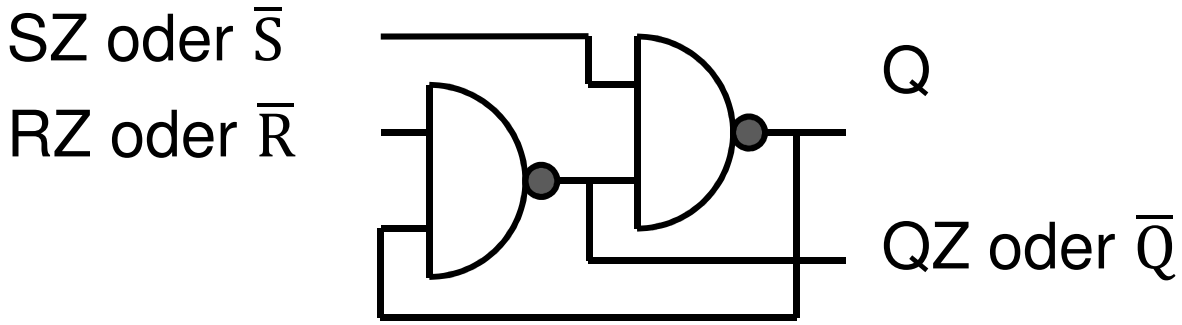
Wir nennen diese Schaltung auch **Basis R/S FlipFlop**, weil sie einen Haltezustand (a=b=1) hat, in dem sie jeden Wert *y* über die Rückkopplung halten kann, sowie eine **R**ücksetz-((a,b)=(1,0)) und eine **S**etzkombination ((a,b)=(0,1)). Allerdings hat sie noch keine Taktkontrolle.

Die Zustandsänderung wird durch Legen eines der Eingänge auf den Wert 0 erreicht. Da der Wert 0 die Änderung bewirkt, nennt man solche Eingänge auch low active und überstreicht ihren Namen bzw. hängt ein *Z* an: RZ oder $\overline{\mathrm{R}}$

Das Überstreichen ist als Kurzschreibweise für die Negation zu sehen. Die Aktivität, die mit dem Eingangsnamen verbunden wird, findet statt, wenn der Ausdruck wahr ist, d.h. Rücksetzen findet statt wenn *RZ=nicht(R) wahr*, also *R=0* ist.

Basis R/S FlipFlop ff

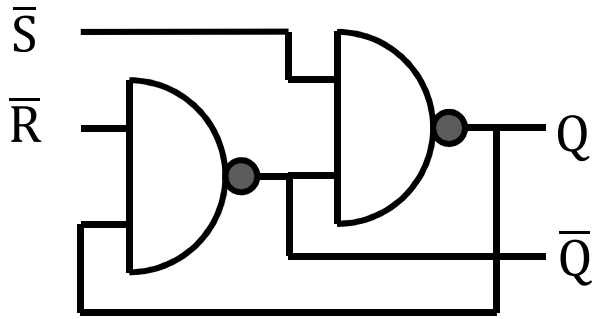
Damit ergibt sich folgende Benennung der Anschlüsse:

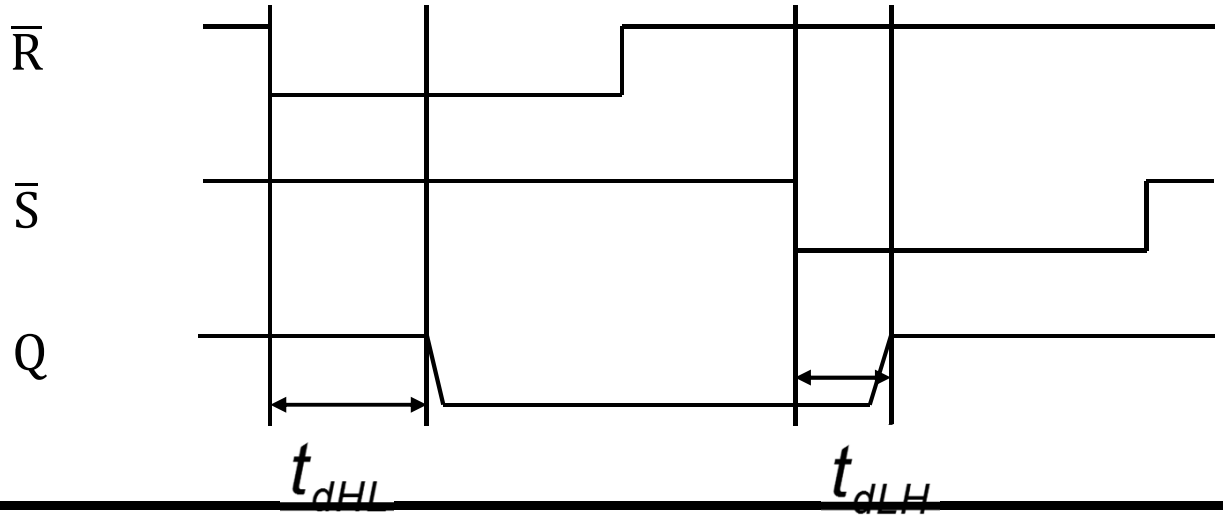


Folgende Tabelle gibt das Verhalten wieder: Sie gibt für die benutzten Eingangskombinationen *(RZ,SZ)<>(0,0)* den Folgezustand *Q‘* von *Q* an:

Basis R/S FlipFlop ff

Die Reaktionszeiten ergeben sich durch die Laufzeiten der Gatter und sind für Setzen und Rücksetzen daher nicht gleich.

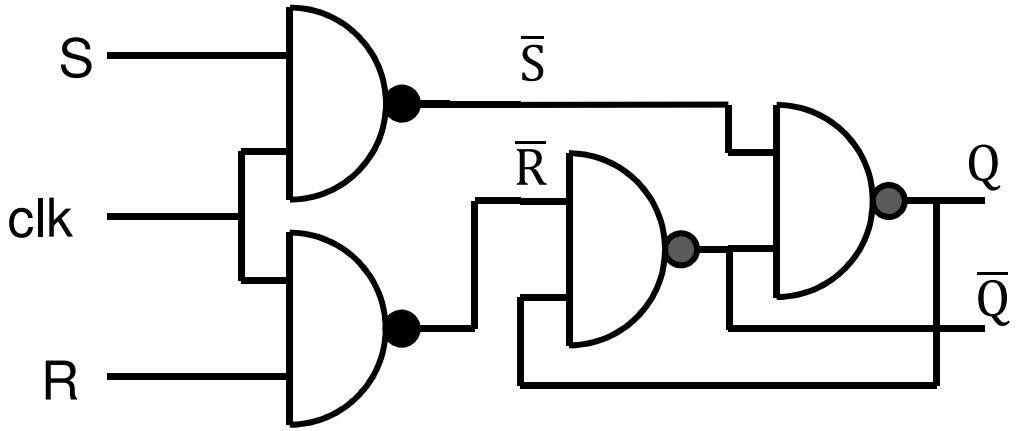




R/S FlipFlop ff

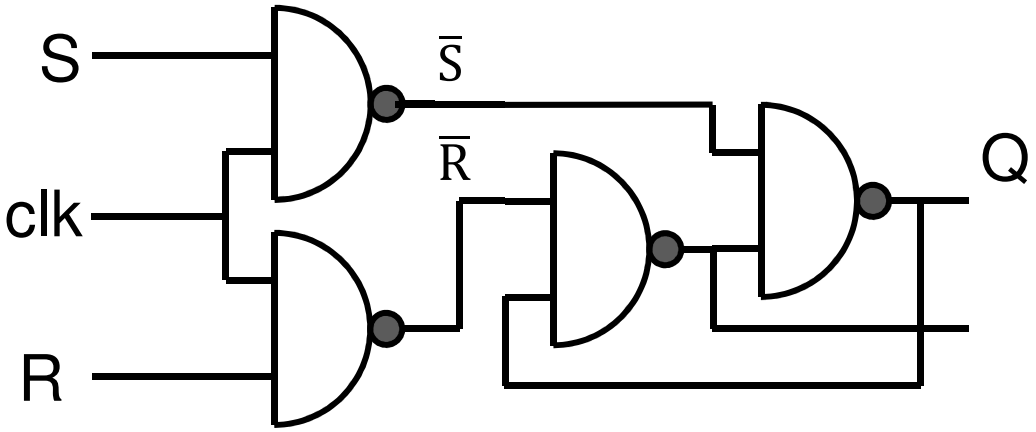
Aufgabe: Steuere das Basis R/S FlipFlop durch einen Taktzustand.

Lösung: Wir verknüpfen einfach den Takt mit den Kontrolleitungen RZ und SZ:

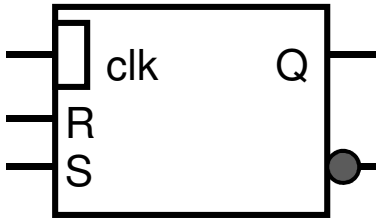


Analyse: Für clk=0 werden SZ, RZ auf nicht(0)=1 gezwungen.

R/S FlipFlop ff



Schaltsymbol:



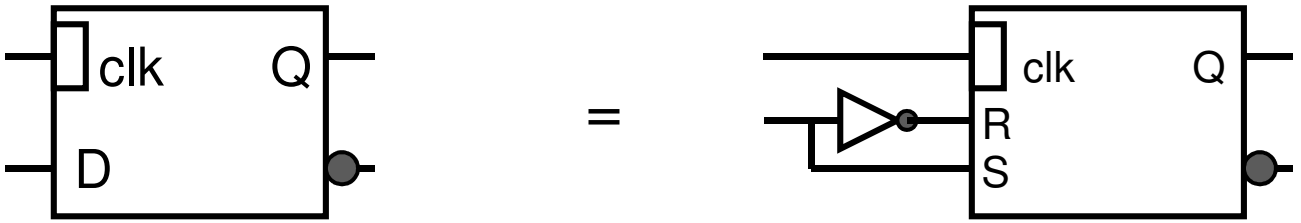
Analyse: Für *clk=0* werden SZ, RZ auf *nicht(0)=1* gezwungen. $\Rightarrow$ **Haltezustand**

Für *clk = 1* werden SZ(RZ) auf $SZ = \overline{1 \cdot S} = \bar{S}$ $(RZ = \overline{1 \cdot R} = \bar{R})$ abgebildet.

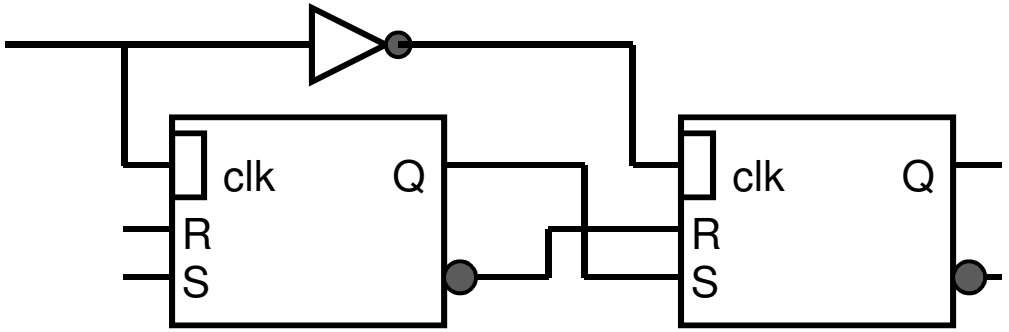
Man kann also nur in der 1 Phase setzen oder rücksetzen.

R/S FlipFlop ff

Offenbar kann man aus dem R/S FlipFlop leicht ein D-Latch bauen:



Und nach dem Master/Slave Prinzip ein flankengesteuertes R/S FlipFlop:

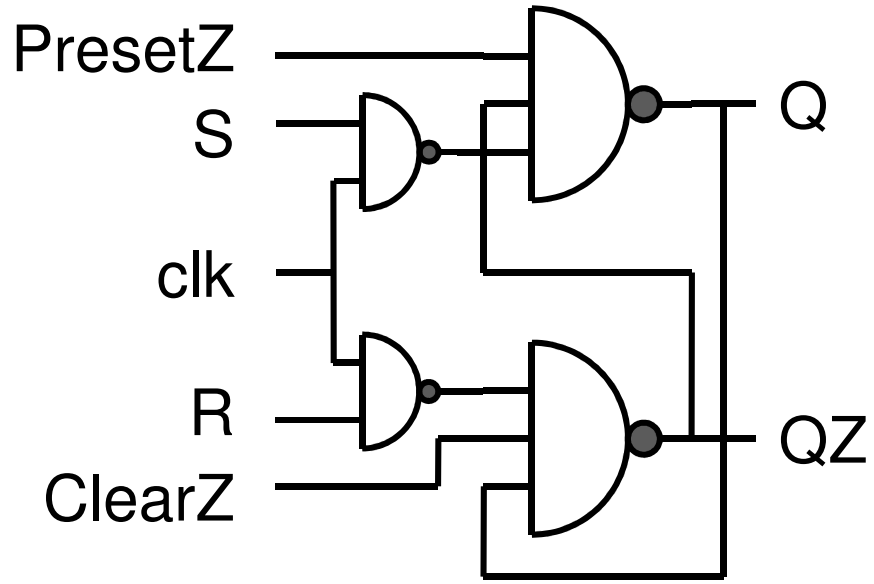


Es gibt einen wahren Wildwuchs von FlipFlop Typen, aus unterschiedlichen Zielsetzungen motiviert:

Synchrone und asynchrone Kontrolle

Ein Beispiel ist der Wunsch, den Ausgang *Q* synchron, d.h. durch den Takt, aber auch asynchron durch spezielle Kontrollleitungen manipulieren zu können. Ein typischer Vertreter ist das R/S FlipFLop mit asynchronem *PresetZ* und *ClearZ*:

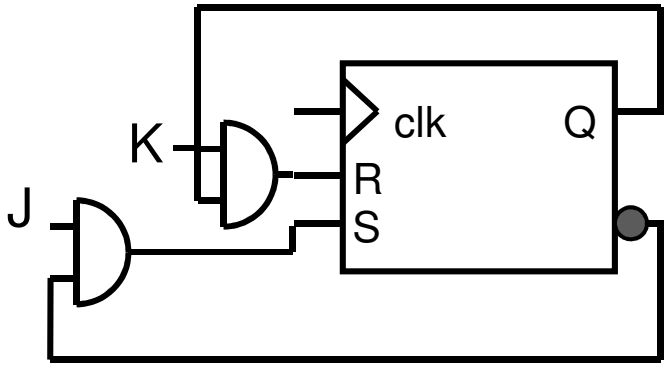
Man kann nun *Q, QZ* unabhängig von *clk* direkt setzen oder rück-setzen.



Ausbau redundanter Belegungen:

Die Belegung (S,R) = (1,1) darf nicht benutzt werden. Sie ist für den Betrieb bedeutungslos. Solche Belegungen nennt man auch redundant. Gibt man dieser Kombination durch entsprechende Verschaltung eine weitere Bedeutung so hat man eine Erweiterung des R/S-FlipFlops.

Beispiel: das J/K FlipFlop

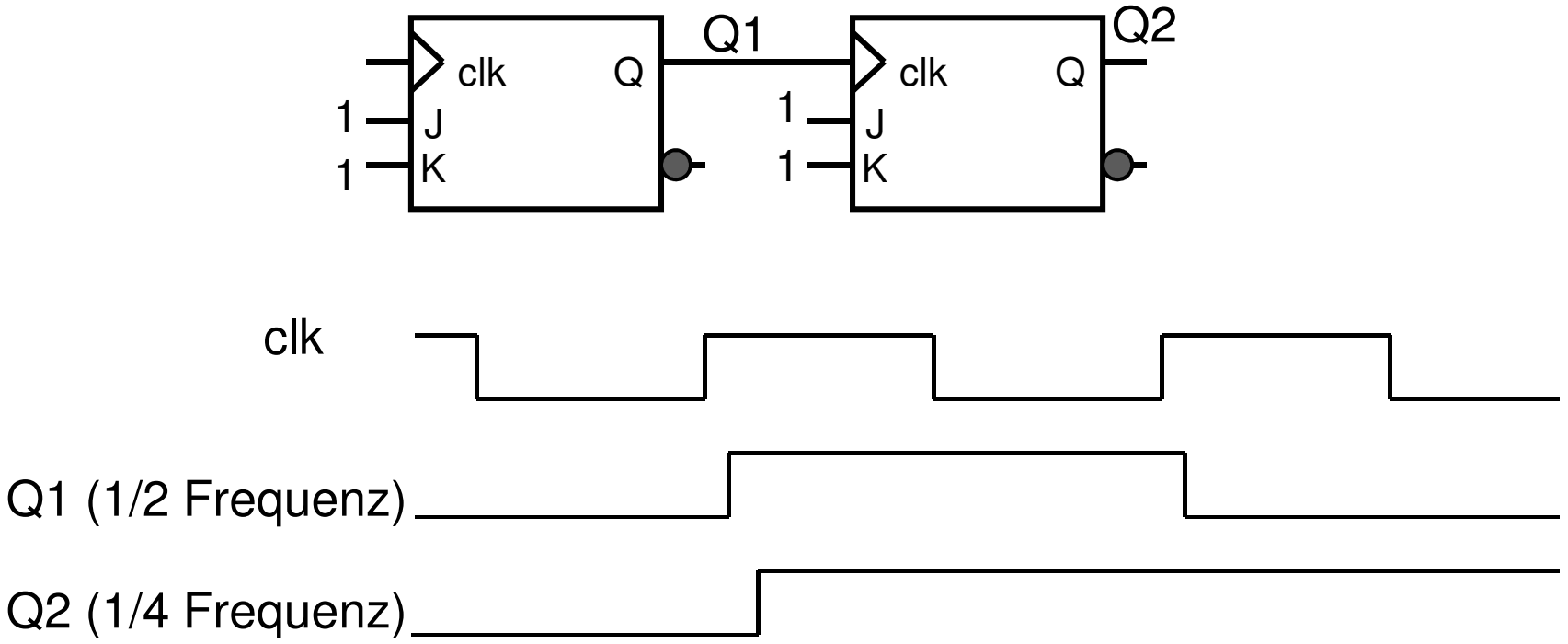


J	K	Q	R=K und Q	S=J und QZ	Q'
0	0	Q	0	0	Q
0	1	Q	Q	0	0
1	0	Q	0	QZ	1
1	1	Q	Q	QZ	QZ

Das FlipFlop kippt für (J,K)=(1,1) den aktuellen Zustand.

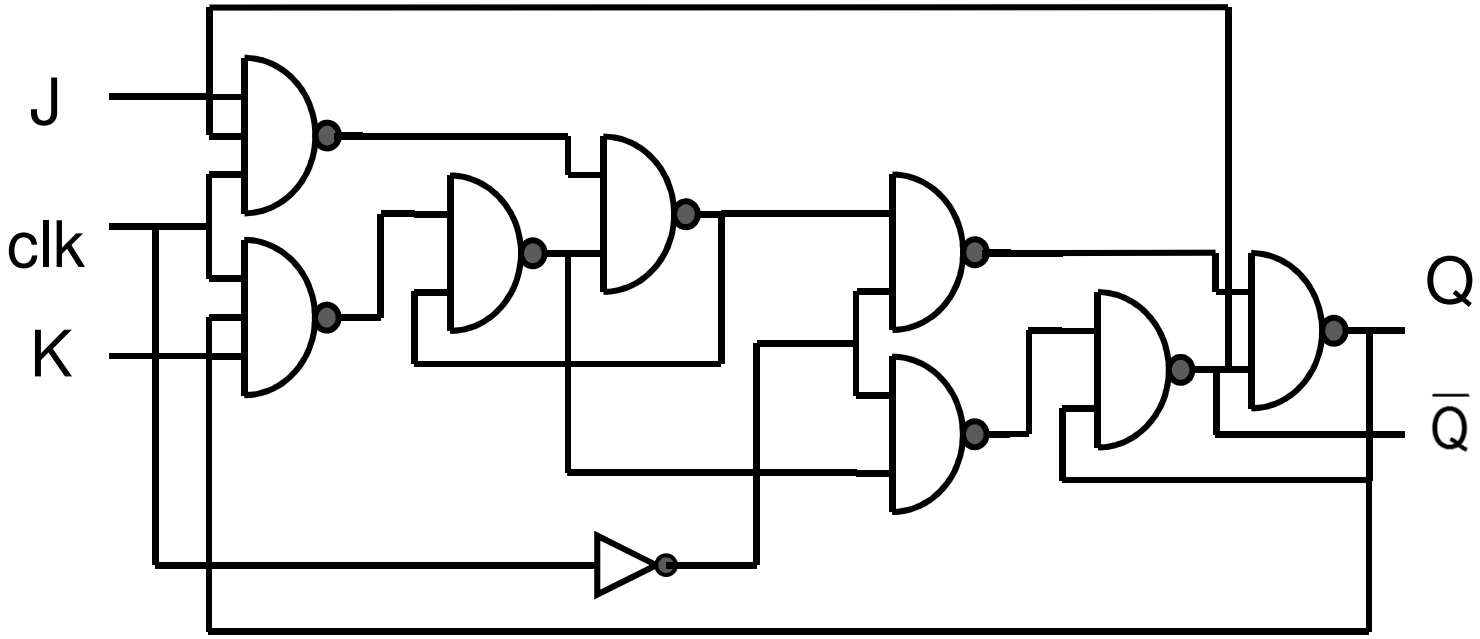
Das J/K FlipFlop

Beispiel: Ein typisches Beispiel für den Einsatz eines J/K FlipFlops ist das Teilen eines Taktes.



Das J/K FlipFlop ff

Aufbau bei direkter Benutzung von Basis R/S FlipFlops



1.3.7 Tristate Treiber und Busse

Unsere Gatter hatten bisher eine strenge Unterscheidung zwischen Eingängen (Transistorgates) und Ausgängen (getrieben durch leitende Pfade nach V_{SS} oder V_{DD}).

Dies reicht auch für viele Probleme zum Entwurf vollkommen aus.

Will man aber komplexere Komponenten miteinander verbinden, dann sind auch die Verbindungsleitungen mit hohen Kosten verbunden, besonders dann, wenn man über Chipgrenzen hinweg geht. (Lötpunkte, Stecker, Pins am Chipgehäuse)

Problem: Schaffe die Möglichkeit *n* Komponenten auf möglichst einfache Weise miteinander zu verbinden, so dass jede Komponente mit jeder Komponente direkten (ohne über andere Komponenten zu gehen) Kontakt aufnehmen kann.

Busse und Tristates ff

Die einzige Möglichkeit, diese direkte Verbindung zu schaffen, setzt voraus, dass

- jede Komponente *n-1* Eingangs- und *n-1* Ausgangsports hat,
- insgesamt *2n(n-1)* Verbindungen geschaffen werden müssen, und
- der Entwurf einer Komponente von vorneherein durch die Komponenten bestimmt wird, mit denen sie zusammenarbeiten soll.

Selbst wenn man die Nebenbedingung direkter Kommunikation weglässt, müsste man eine Menge von Kommunikationsbausteinen explizit miteinander verschalten, so dass alle Verbindungen möglich sind.

Ausweg:

Sehe Ports vor, die Eingang oder Ausgang sein können, oder sich neutral verhalten.

Busse und Tristates ff

Frage: Wann verhält sich ein Anschluss "neutral"?

Antwort: Ein Anschluss verhält sich neutral, wenn er einen sehr hohen Widerstand gegen beide Versorgungspole hat. Er verhält sich dann isoliert und bildet allenfalls noch eine kapazitive und induktive Last auf der Leitung an die er gebunden ist.

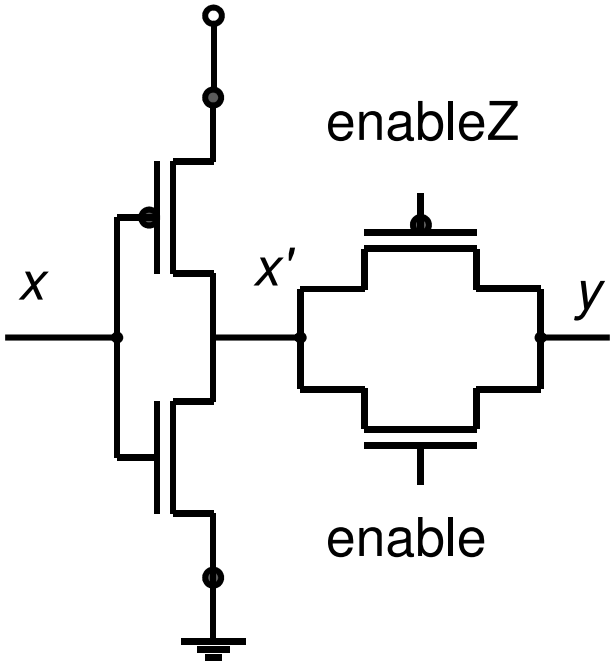
Man nennt einen solchen Zustand eines Anschlusses auch "hochohmig" oder "high impedance state".

Wir können ein solches Verhalten leicht mit CMOS Transistoren realisieren, da sie als Schalter arbeiten. Man muss lediglich mit einem Schalter den Ausgang von seinen Treibern trennen können.

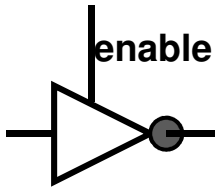
Dies leistet ein sogenannter **Tristate Treiber**:

Der Tristate Treiber

Die einfachste Möglichkeit, für einen Ausgang neutrales Verhalten zu erhalten, ist die Erweiterung eines Inverters mit einem Schalter:



Symbol



enable und enableZ sind komplementär zu wählen, d.h. stets

enable = nicht enableZ

Verhalten:

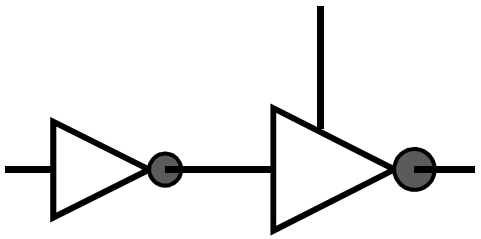
enable = 0: *y* ist von *x'* isoliert. Es gibt seitens des Inverters keine leitende Verbindung von *y* nach V_{SS} oder V_{DD}.

enable = 1: *y* ist über den (die) Transistoren mit *x'* verbunden. Benutzt man nur einen Transistor, verliert man aber beim Ausgangspegel die Schwellenspannung eines Transistors.

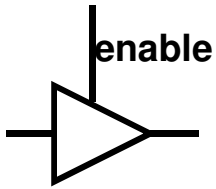
Der Tristate Treiber

Der Baustein auf der Basis eines Inverters ist allerdings invertierend, d.h. er kehrt das Eingangssignal um.

Es ist nun aber eine leichte Übung, auch einen nichtinvertierenden Treiber aufzubauen: Man benutze einfach einen Doppelinverter.



Symbol



Bemerkung:

Meistens nutzt man die Inverterkaskade zur Signalverstärkung, d.h. man benutzt im zweiten Inverter grössere Transistoren als im ersten Inverter. Dadurch kann ein solcher Treiber auch relativ hohe Lasten schnell schalten. Dies ist insbesondere bei Bussen wichtig.

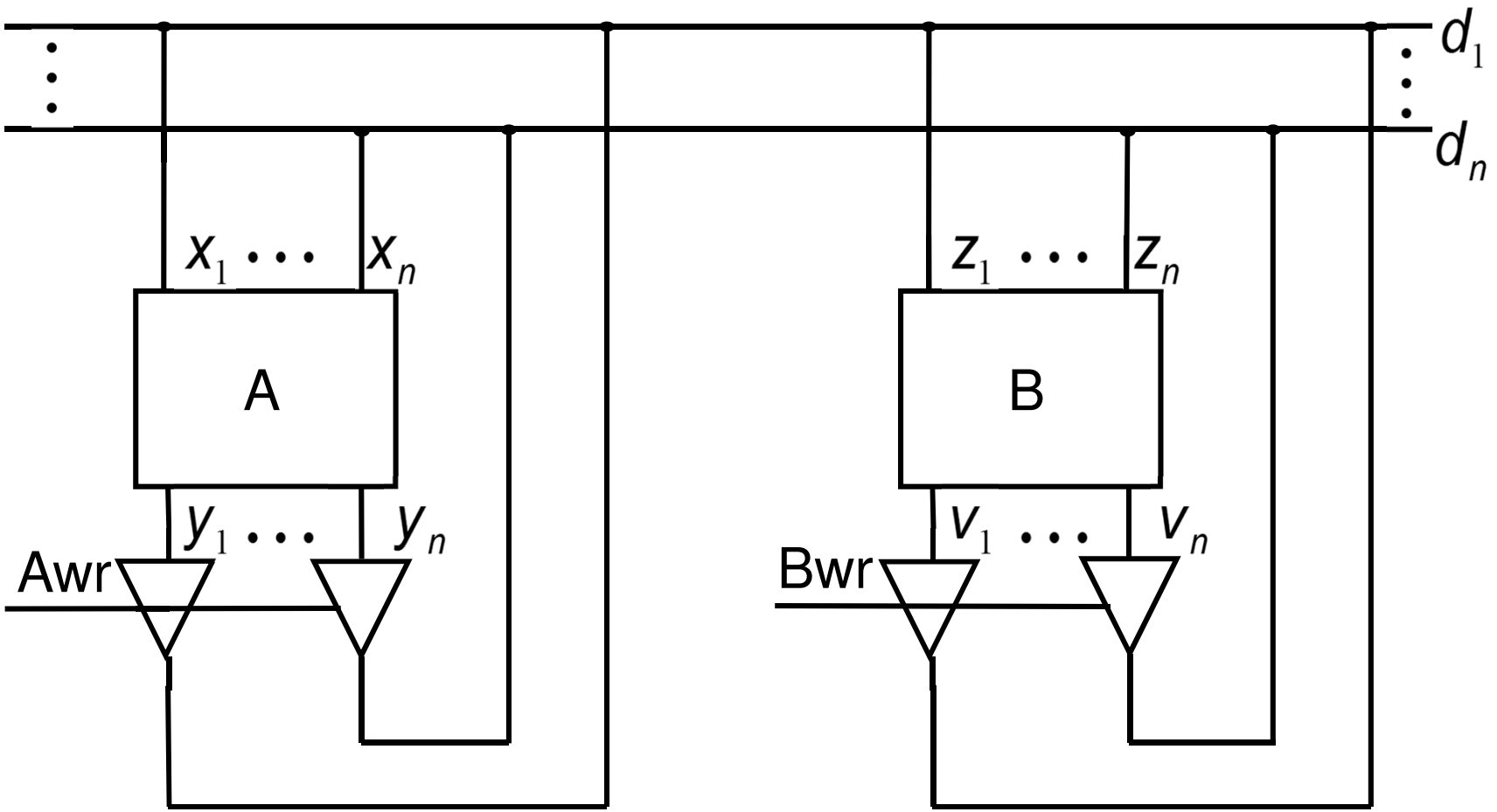
Busse

Ein Bus bietet nun auf der Basis solcher bidirektionaler Anschlüsse die Möglichkeit, viele Komponenten direkt miteinander verbinden zu können. Er besteht aus

- einer genormten Menge bidirektionaler Datenleitungen,
- einer genormten Menge von unidirektionalen Kontrollleitungen,
- ggf. einer Menge von Taktleitungen (synchrone Busse),
- einem Protokoll, in dem vereinbart wird, in welchem zeitlichen Ablauf über den Kontrollleitungen eine Komponente auf die Datenleitungen schreiben darf, bzw. von ihnen lesen kann.

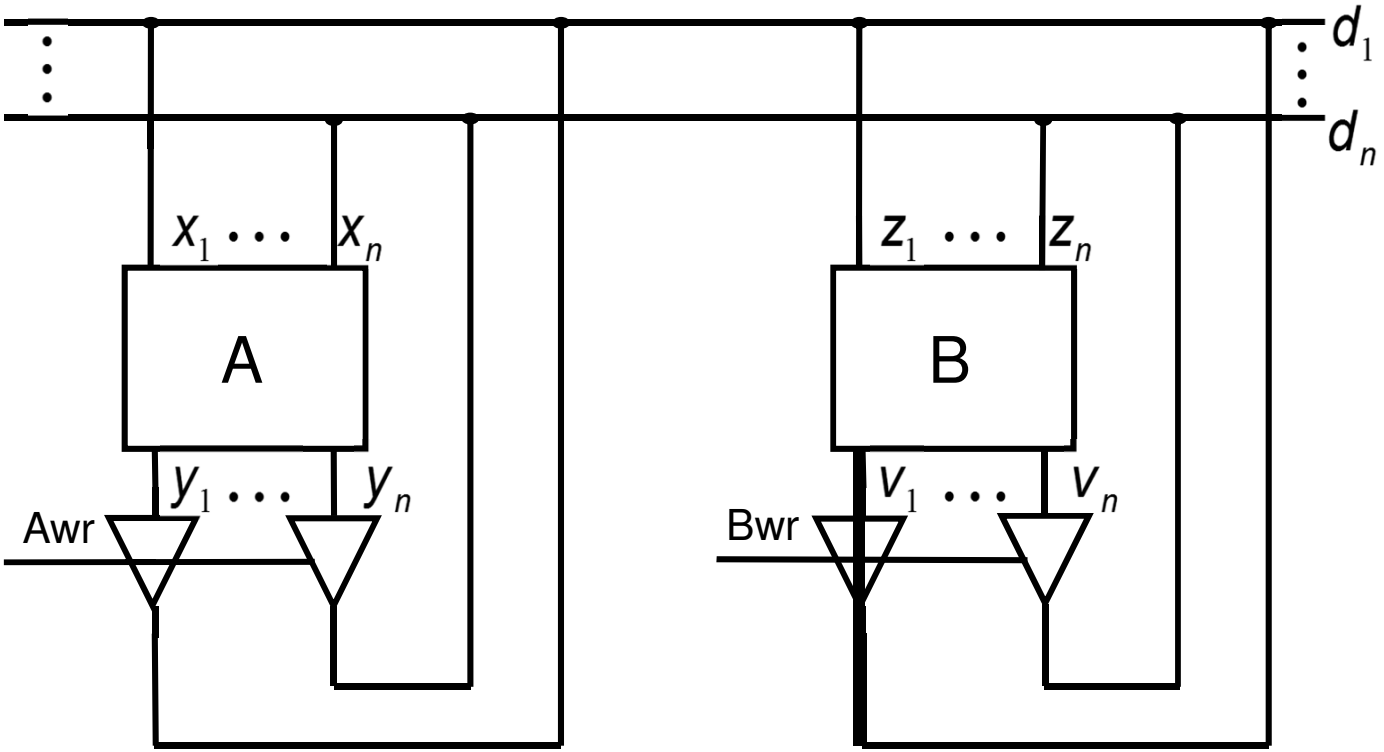
Wir wollen es vorerst bei einem einfachen Beispiel dazu belassen:

Beispiel: bidirektionaler Bus



Beide Komponenten A und B können von d lesen oder auf d schreiben. Aus den Kontrollsignalen (nicht eingezeichnet) sind nun Awr und Bwr korrekt zu erzeugen.

Beispiel: bidirektionaler Bus

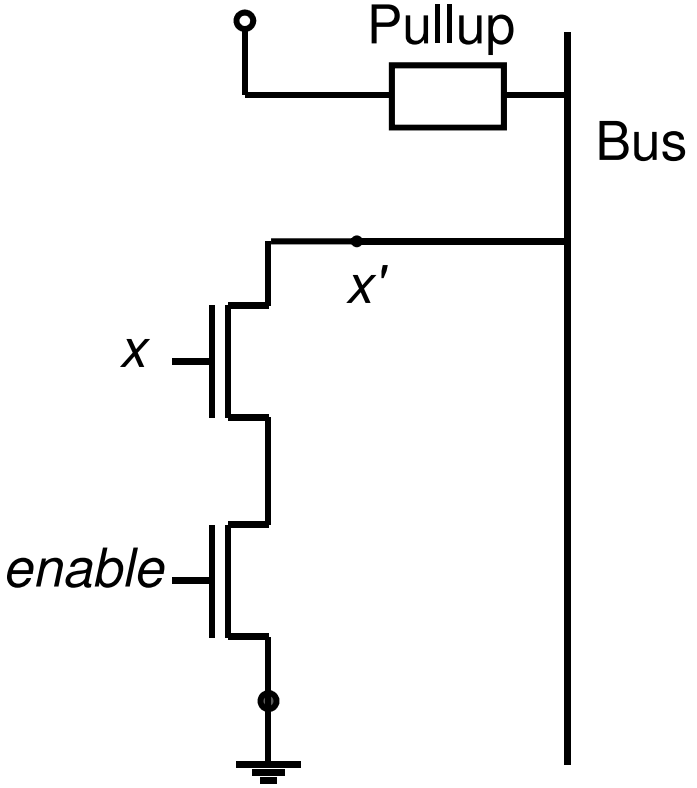


Awr = 0, Bwr = 1: Komponente A liest von Komponente B

Awr = 1, Bwr = 0: Komponente B liest von Komponente A

Ältere Tristate Techniken

Eine andere Möglichkeit, ein hochohmiges Verhalten zu realisieren, besteht darin, dass man eine der Verbindungen (V_{SS} oder V_{DD}) nicht herstellt, sondern über einen Abschlusswiderstand (Pulldown oder Pullup) realisiert, der sehr viel grösser ist, als der Widerstand des schaltbaren Pfades:



enable = 0: *x'* ist isoliert. Der Bus wird über den Abschlusswiderstand (Pullup) auf *1* gehalten

enable = 1: *x'* ist mit V_{SS} verbunden, falls *x = 1* ist. Der Bus wird auf *x' = 0* gezogen.

Bemerkung: Es erfolgt eine UND Verknüpfung (ODER bei Pulldown Abschluss) aller schreibenden Komponenten (Wired AND, Wired OR).

Modellierung von Tristates in WüHDL

Um diesem Konzept Rechnung zu tragen, müssen wir in WüHDL ein weiteres Konzept einführen:

- Es muss Signale geben, die von mehreren Prozessen beschrieben werden dürfen (mehrere Treiber).
- Wenn zu einem Zeitpunkt mehrere Treiber einem Signal einen Wert zuweisen, muss aus allen diesen Werten ein aufgelöster Wert berechnet werden.

Dazu sind sogenannte resolved Types vorgesehen:

Ein resolved Type *ResolvedT* muss Untertyp eines schon definierten Typs *UnresolvedT* sein, und es muss eine Auflösungsfunktion

FUNCTION *resolveT*(ARRAY(natural RANGE<>) OF *UnresolvedT)*

RETURN *UnresolvedT*

existieren, die die Auflösung berechnet.

Wir verdeutlichen dies zunächst an einem einfachen Beispiel, das nur die Benutzung eines hochohmigen Zustandes erlaubt.

Resolved Types

```
TYPE Utri_state_logic IS  -- nicht aufgelöster Typ
(  'Z', -- für den hochohmigen Zustand
   '0', -- für den Logikwert 0
   '1', -- für den Logikwert 1
   'X'  -- für Fehlerbedingungen (verschiedene Pegel ungleich Z)
);
-- Wird als Parametertyp für die Auflösungsfunktion benötigt.
TYPE  Utri_state_logic_vector
IS ARRAY (INTEGER RANGE <>) OF Utri_state_logic;
-- Nützlicher Typ zur Definition der Auflösung über eine Tabelle
-- von TYP x TYP
TYPE resolutiontable
IS ARRAY(Utri_state_logic, Utri_state_logic) OF Utri_state_logic;
```

Resolved Types -- ff

CONSTANT resolvetable : resolutiontable :=

```
------------------------------
--|'Z'|'0'|'1'|'X'|          |
( ('Z','0','1','X'), --'Z'|
  ('0','0','X','X'), --'0'|
  ('1','X','1','X'), --'1'|
  ('X','X','X','X'));--'X'|
------------------------------
```

Resolved Types

```
FUNCTION resolve_tri_state_logic (values: IN Utri_state_logic_vector)
      RETURN Utri_state_logic IS
      VARIABLE result: tri_state_logic := 'Z';
BEGIN
      FOR index IN values‘range
      LOOP
        result := resolvetable(result,values(index));
      END LOOP;
      RETURN result;
END FUNCTION resolve_tri_state_logic;

-- Definition des aufgelösten Typs
SUBTYPE tri_state_logic IS resolve_tri_state_logic Utri_state_logic;
```

Das Package IEEE.std_logic_1164

Sehr weite Verbreitung als aufgelöster Typ zum Ersatz des Typs BIT findet der in der Bibliothek **IEEE** im Paket **std_logic_1164** definierte, aufgelöste Typ **std_logic**.

Im Package werden die booleschen Operatoren entsprechend überladen und Vektoren ebenfalls als Typ bereitgestellt.

```
PACKAGE std_logic_1164 IS
    -- logic state system
    TYPE std_ulogic IS ( -- Unaufgelöste Fassung von std_logic
        'U',    -- Uninitialized
        'X',    -- Forcing Unknown      (0 gegen 1)
        '0',    -- Forcing 0
        '1',    -- Forcing 1
        'Z',    -- High Impedance       (Hochohmig)
        'W',    -- Weak Unknown         (Pullup gegen Pulldown)
        'L',    -- Weak 0               (Pulldown)
        'H',    -- Weak 1               (Pullup)
        '-' );  -- Don't care           (später!)
```