2. Einführung in VHDL

Programm für heute:

- Motivation für eine Hardwarebeschreibungssprache
- Aufbau einer VHDL-Beschreibung
- Signale
    - Zuweisungen
    - Wertebereich
    - Schnittstellen
- Entity und Architecture
    - Hardwareanalogie
    - Deklaration
    - Hierarchie
    - Component
- Standard Schaltnetze
    - Multiplexer
    - Halbaddierer
    - Volladdierer
    - Ripple-Carry-Addierer
- Entwurf von Funktionselementen mit Prozessen
    - Deklaration
    - Empfindlichkeitsliste
    - Sequentielle Anweisungen
    - Case Anweisung
    - If Anweisung
    - Schleifen

Signale

Signale in VHDL sind Variablen für Werte auf Leitungen. Diese können zugewiesen und gelesen werden, so wie wir es von Variablen in anderen Programmiersprachen gewohnt sind.

Im Gegensatz zu „normalen“ Variablen haben Signale aber eine zeitliche Komponente, d.h. neben dem gegenwärtigen Wert gibt es Werte eines Signals in der Vergangenheit und in der Zukunft. Zukünftige Werte eines Signals können zugewiesen werden, z.B.

```
a <= ‘0‘ AFTER 10 ns;
```

Dieses Statement weist dem Signal a den Wert logisch ‘0‘ zu zum Zeitpunkt 10 ns vom Zeitpunkt des Anfangs einer Simulation an.

Jedes Signal ist von einem eindeutig zu definierenden Typ. Wir betrachten hier zunächst den Datentyp `bit`. Der Wertevorrat des Datentyps `bit` besteht aus den logischen Werten ‘0‘ und ‘1‘.

Ein `bit_vector` stellt einen aus mehreren `bit`-Signalen bestehenden Bus dar. Dieser kann entweder aufsteigend, z.B. als `bit_vector(0 to 7)`, oder aber abfallend als `bit_vector(7 downto 0)` bezeichnet werden.

Signale können in VHDL zugewiesen werden durch den Zuweisungsoperator <=.

Beispiele:

```
c <= ‘1‘;

c <= a or b;

bus <= “10010110“;       -- man beachte: bei bit_vector Anführungszeichen

bus <= ( ‘1‘, ‘0‘, ‘0‘, ‘1‘, ‘0‘, ‘1‘, ‘1‘, ‘0‘);– geht genauso
```

Wenn mehrere Zuweisungsanweisungen aufeinander folgen, kann man daraus aber nicht auf sequentielle Ausführung schließen. Stattdessen werden alle Zuweisungen nebenläufig ausgeführt.

Zuweisungen von Werten an Signale können in Abhängigkeiten von anderen Signalen ausgeführt werden.

Beispiel:

```
Y <= a when S = ‘1‘ else ‘0‘; -- nur wenn S einen high-Pegel hat, wird Y
                              -- auf den  Wert von a gesetzt, sonst auf
                              -- den low-Pegel (‘0‘).
```

Signale haben einen **Richtungsmodus**. Dieser wird bei ihrer Deklaration festgelegt.

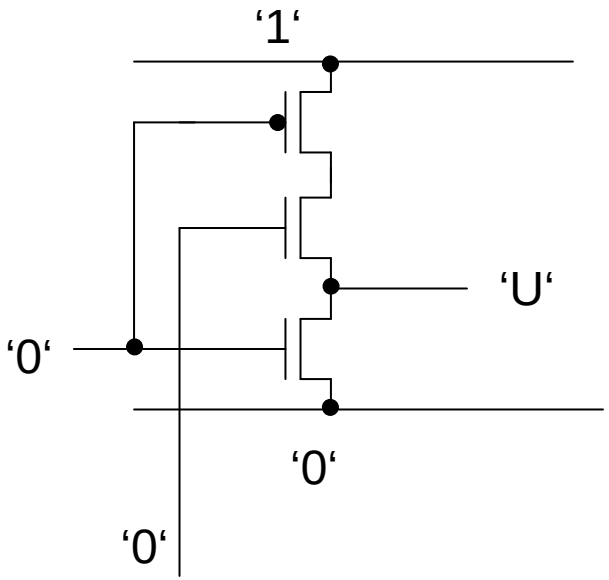
`in`	bezeichnet ein Eingangssignal. Dieses kann nicht modifiziert werden, darf also nur auf der rechten Seite einer Signal-zuweisung stehen oder in einer Signalabfrage.
`out`	bezeichnet ein Ausgangssignal. Dieses darf nur auf der linken Seite einer Signalzuweisung stehen.
`buffer`	Ausgangssignal, das auch als Eingang oder zur Signalabfrage verwendet werden darf.
`inout`	bidirektes Signal beim Datentyp std_logic

Der Datentyp `std_logic` ist eine Erweiterung des Datentyp bit, die besser den realen Anforderungen des Hardwareentwurfs genügt. Es kann nämlich in der Realität passieren, das Signale nicht 0 oder 1 sind, sondern andere Werte annehmen. Ferner kann ein Signal (nach dem Einschalten eines Geräts) noch undefiniert sein, und erst nach einer gewissen Zeit oder nach einem spezifischen Ereignis nimmt das Signal einen definierten Wert an. Ein Logik-Simulator, der einen Schaltungsentwurf simulieren soll, muss daher Werte für Signale zur Verfügung haben, die weder 0 noch 1 sind. Der Datentyp `std_logic` ist neunwertig:

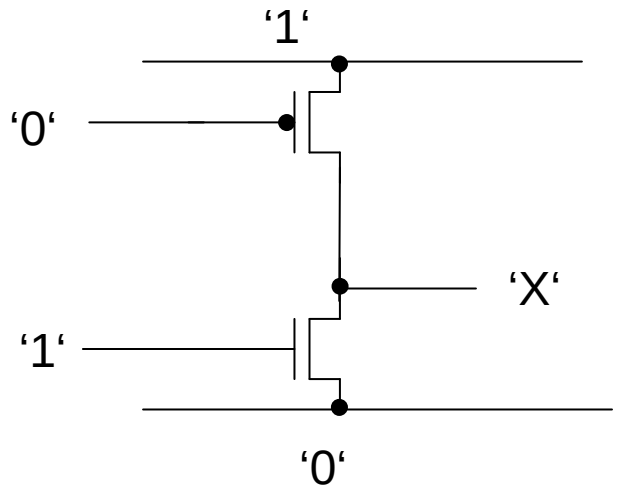
```
Type std_logic is (‘U‘,‘X‘,‘0‘,‘1‘,‘Z‘,‘L‘,‘H‘,‘W‘,‘-‘);
```

Wert	Bedeutung	Verwendung
‘U‘	Nicht initialisiert	Nicht initialisiertes Signal
‘X‘	Konflikt	Mehr als ein aktiver Signaltreiber
‘0‘	Starke logische 0	Entspricht ‘0‘ eines bit-Signals
‘1‘	Starke logische 1	Entspricht ‘1‘ eines bit-Signals
‘Z‘	Hochohmig	Ungetriebener Tri-State-Ausgang
‘L‘	Schache logische 0	Ausgang mit pull-down-Widerstand
‘H‘	Schwache logische 1	Ausgang mit pull-up-Widerstand
‘W‘	Schwach unbekannt	Konflikt zwischen ‘L‘ und ‘H‘‘
‘-‘	Don‘t care	Zustand bedeutungslos, kann für Minimierung verwendet werden

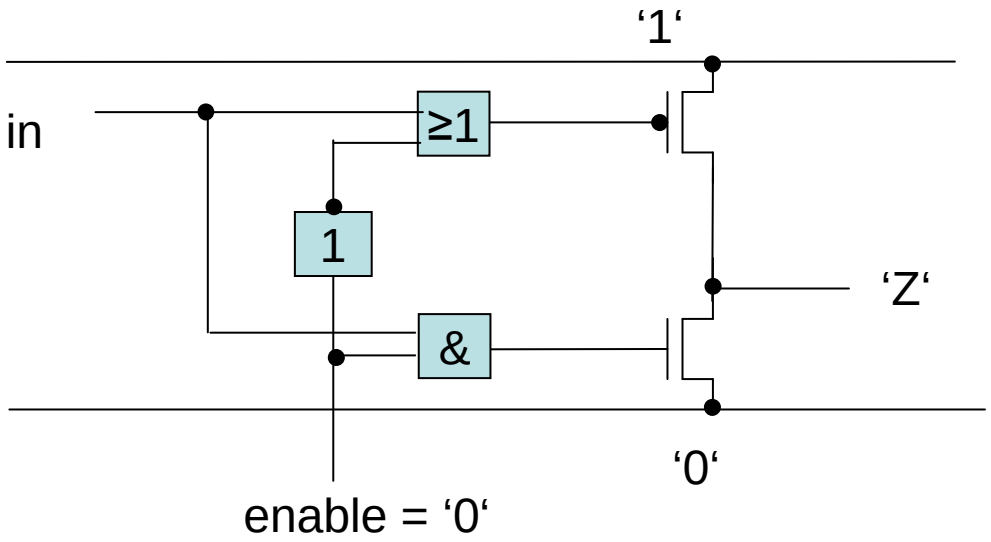
std_logic Wert ‘U‘



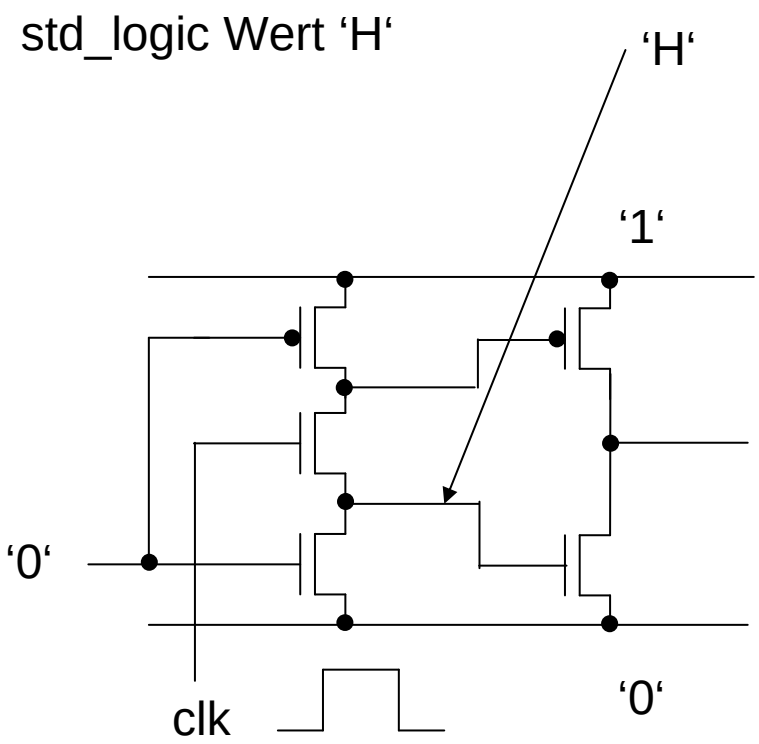
std_logic Wert ‘X‘



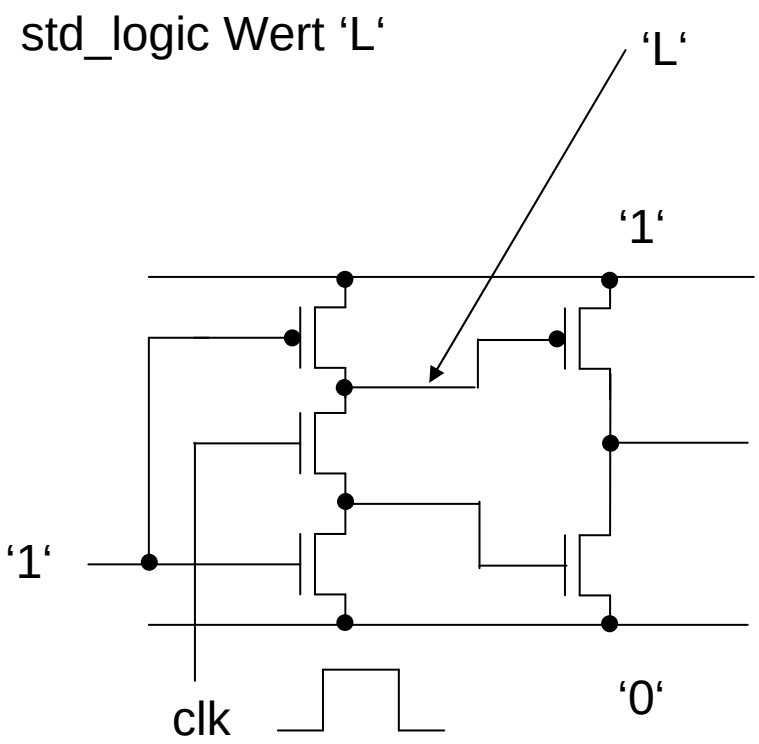
std_logic Wert ‘Z‘



Hochohmiger Zustand am Ausgang eines Tri-State-Treibers

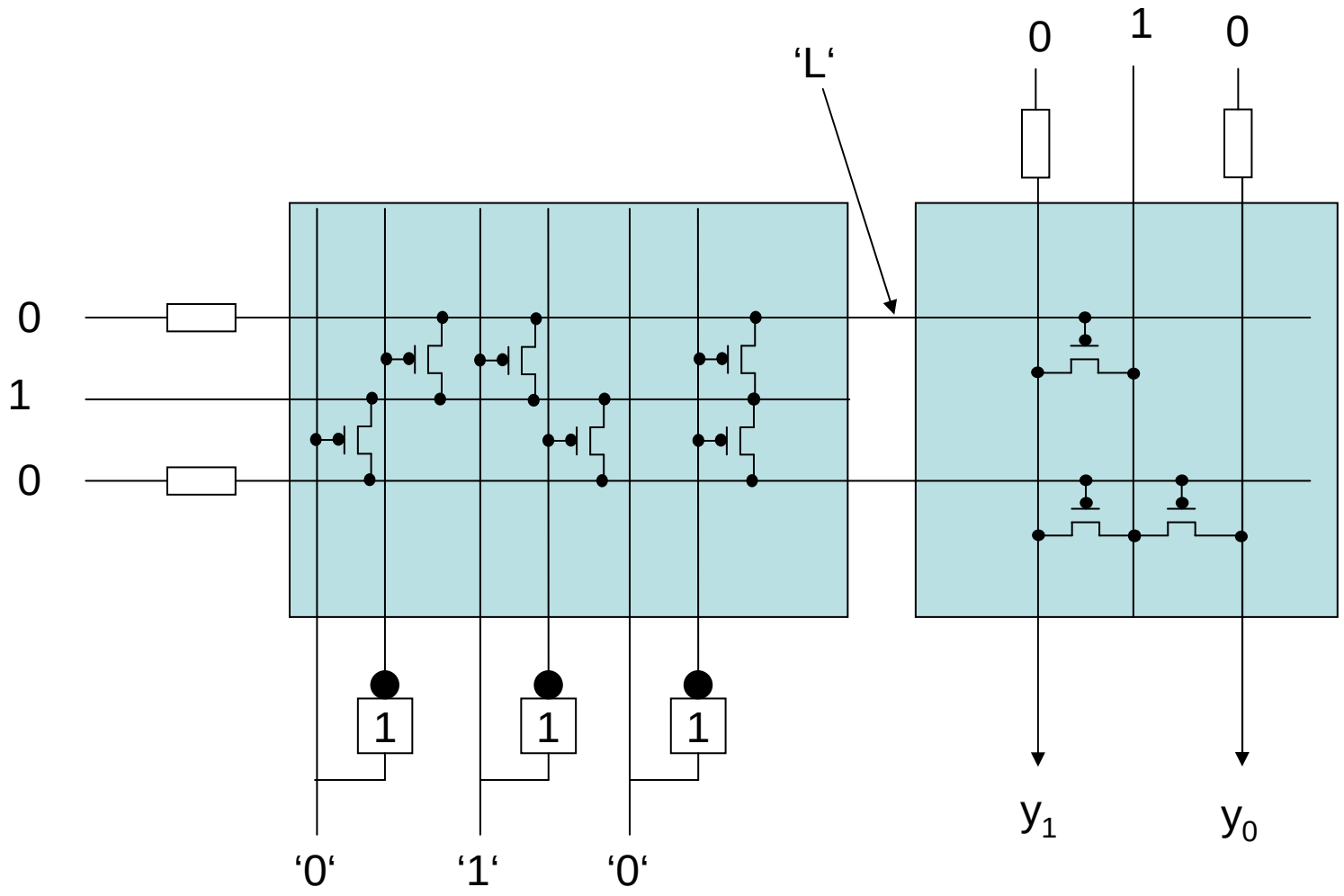


Schwache 1 dynamisch gespeichert auf dem Gate eines Transistors



Schwache 0 dynamisch gespeichert auf dem Gate eines Transistors

Weiteres Beispiel für eine schwache 0 (‘L‘)



Eingänge

Ausgänge

	‘U‘	‘X‘	‘0‘	‘1‘	‘Z‘	‘W‘	‘L‘	‘H‘	‘-‘
‘U‘	‘U‘	‘X‘	‘0‘	‘1‘	‘Z‘	‘W‘	‘L‘	‘H‘	‘-‘
‘X‘	‘X‘	‘X‘	‘X‘	‘X‘	‘X‘	‘X‘	‘X‘	‘X‘	‘X‘
‘0‘	‘0‘	‘X‘	‘0‘	‘X‘	‘0‘	‘0‘	‘0‘	‘0‘	‘0‘
‘1‘	‘1‘	‘X‘	‘X‘	‘1‘	‘1‘	‘1‘	‘1‘	‘1‘	‘1‘
‘Z‘	‘Z‘	‘X‘	‘0‘	‘1‘	‘Z‘	‘W‘	‘L‘	‘H‘	‘Z‘
‘W‘	‘W‘	‘X‘	‘0‘	‘1‘	‘W‘	‘W‘	‘W‘	‘W‘	‘W‘
‘L‘	‘L‘	‘X‘	‘0‘	‘1‘	‘L‘	‘W‘	‘L‘	‘W‘	‘L‘
‘H‘	‘H‘	‘X‘	‘0‘	‘1‘	‘H‘	‘W‘	‘W‘	‘H‘	‘H‘
‘-‘	‘-‘	‘X‘	‘0‘	‘1‘	‘Z‘	‘W‘	‘L‘	‘H‘	‘-‘

entity

In einer mit `entity` bezeichneten Entwurfseinheit werden die Schnittstellen eines Entwurfsblocks nach außen beschrieben.

Wenn man sich einen Chip vorstellt, entspricht die `entity` seinem Gehäusetyp, das durch Anzahl, Bezeichnung und Anordnung der Anschlüsse eindeutig bestimmt ist. Ferner können im Entity-Teil Konstanten und Unterprogramme deklariert werden, die für alle dieser Entity zugeordneten Architekturen gelten sollen.

Die Deklaration der Anschlüsse innerhalb der `entity` erfolgt mit Hilfe der `port`-Anweisung.

Ein `port`-Signal beschreibt die Kommunikation einer Entity nach außen.

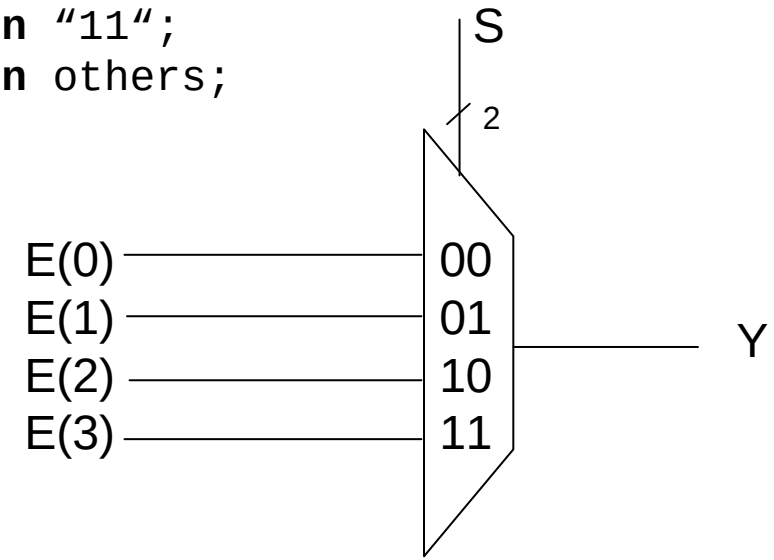
```
entity Mux4x1
port(   S: in std_logic_vector(1 downto 0);
        E: in std_logic_vector(3 downto 0);
        y: out std_logic);
end Mux4x1;
```

architecture

Die `architecture` beschreibt das Innenleben, d.h. die Funktionalität der Entwurfseinheit. Jeder `entity` muss mindestens eine `architecture` zugeordnet sein.

Während die entity die Schnittstelle einer Einheit nach draußen beschreibt, beschreibt die `architecture`, welche Funktion die Einheit ausführt.

```
entity Mux4x1
port(   S: in std_logic_vector(1 downto 0);
        E: in std_logic_vector(3 downto 0);
        Y: out std_logic);
end Mux4x1
architecture Verhalten of Mux4x1 is
begin
        with S select           -- Auswahlsignal
        Y <=    E(0) when “00“,
                E(1) when “01“,
                E(2) when “10“,
                E(3) when “11“;
                E(0) when others;
end Verhalten;
```



Ein paar Bemerkungen zu diesem Codestück:

Kommentare beginnen an beliebiger Stelle in der Zeile mit --

Groß-/Kleinschreibung wird ignoriert, aber Konvention:

- VHDL-Schlüsselworte werden klein geschrieben
- Eigendefinitionen wie z.B. Signale werden groß geschrieben.

Identifier müssen mit einem Buchstaben beginnen und danach Buchstaben, Ziffern oder Unterstrich (_).

VHDL-Anweisungen werden mit „;“ abgeschlossen. Zum Trennen dienen spezielle Schlüsselworte (**`with, select, when`**), manchmal auch ein Komma oder Doppelpunkt. Anweisungen können über mehrere Zeilen verteilt werden.

Zuweisung und Abfrage von Signalwerten vom Typ `std_logic` werden in Apostrophs eingefasst.

```
A <= ‘0‘;
```

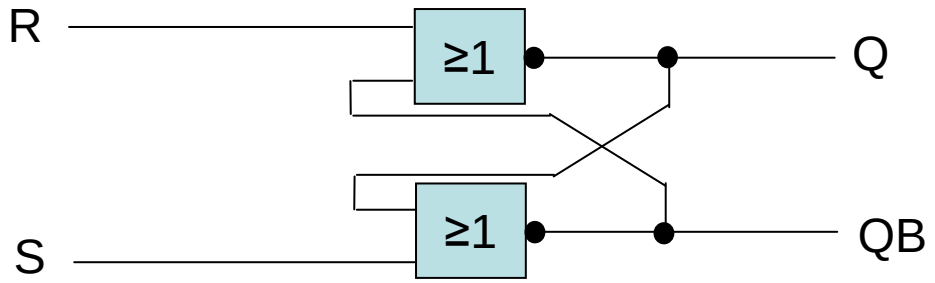
`Bei std_logic_vector` stattdessen Anführungszeichen

```
AV <= “0010“;
```

Die Angabe der Signalflussrichtung in der `port`-Anweisung der entity ist für die Verwendung der Signale in der `architecture` bindend. Das bedeutet insbesondere

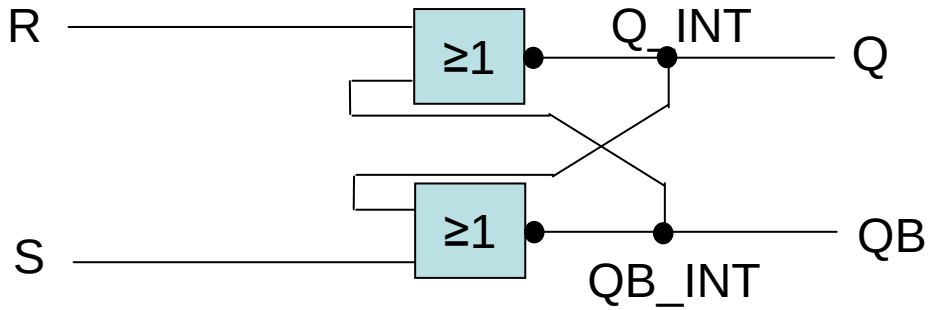
- Ein mit **`in`** deklariertes Signal kann nie auf der linken Seite einer Signalzuweisung stehen.
- Ein mit **`out`** deklariertes Signal kann nie auf der rechten Seite einer Signalzuweisung stehen.

Fehlerhafte Verwendung von Signalen



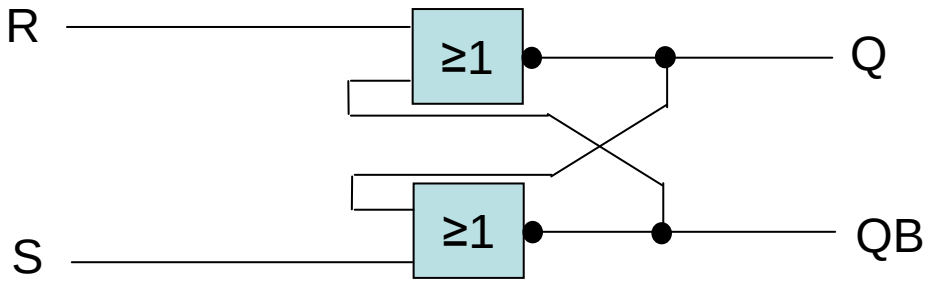

```
entity NorFlipFlop
port(   S: in std_logic;
        R: in std_logic;
        Q: out std_logic;
        QB: out std_logic);
end NorFlipFlop;
architecture Verhalten of NorFlipFlop is
begin

        Q <=    R nor QB;           -- Fehler: QB ist out
        QB <=   S nor Q;            -- Fehler: Q ist out

end Verhalten;
```

Korrigierte Lösung




```
entity NorFlipFlop
port(   R, S: in std_logic;
        Q, QB: out std_logic);
end NorFlipFlop;

architecture Verhalten of NorFlipFlop is
signal Q_INT, QB_INT: std_logic;              -- lokale Signale
begin

        Q_INT <= R nor QB_INT;
        QB_INT <= S nor Q_INT;
        Q <=    Q_INT;
        QB <=   QB_INT;

end Verhalten;
```

Auch möglich,
Aber schlechter
Stil



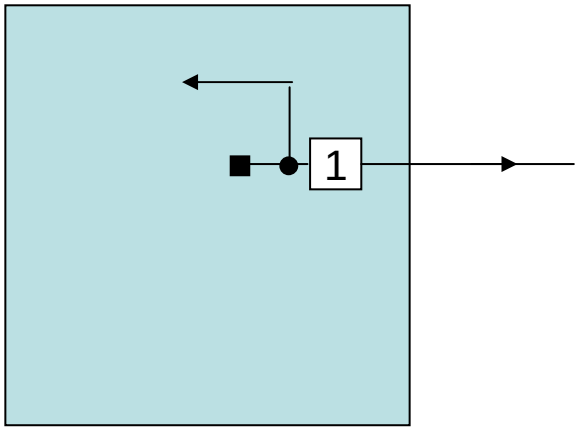

```
entity NorFlipFlop
port(   S: in std_logic;
        R: in std_logic;
        Q: buffer std_logic;
        QB: buffer std_logic);
end NorFlipFlop;
architecture Verhalten of NorFlipFlop is
begin

        Q <=    R nor QB;
        QB <=   S nor Q;

end Verhalten;
```

Unterschied zwischen **`buffer`** und **`inout`**:

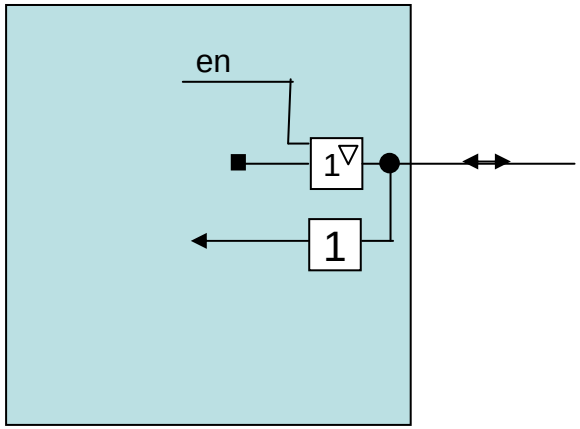
`buffer`: Signal, das am Ausgang liegt, aber auch auf der rechten Seite einer Signalzuweisung oder in einer Signalabfrage stehen darf.

`inout`: Signal, das bidirektional an der Schnittstelle einer Einheit fungiert, zum Beispiel ein Bus, der lesend oder schreibend genutzt werden kann.

`buffer`



`inout`



Deklaration von Bussignalen:

Absteigend: `S: in std_logic_vector(3 downto 0)`
Aufsteigend `S: in std_logic_vector(0 to 3)`

Die Zuweisung

```
S <= “0001“;
```

bedeutet bei absteigender Deklaration:

```
S(3) = ‘0‘, S(2) = ‘0‘, S(1) = ‘0‘, S(0) = ‘1‘;
```

und bei aufsteigender Deklaration:

```
S(3) = ‘1‘, S(2) = ‘0‘, S(1) = ‘0‘, S(0) = ‘0‘;
```

Bei der in der Digitaltechnik üblichen Interpretation von Binärzahlen ist daher die absteigende Deklaration zu wählen.

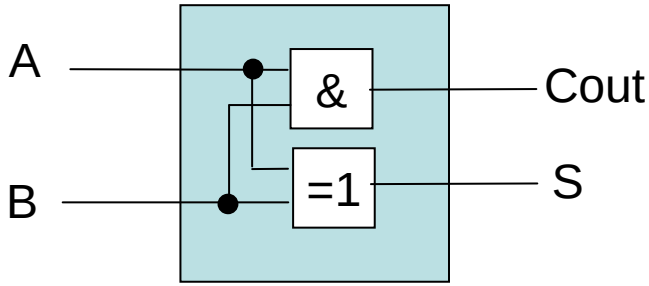
Hinweis zu Buszuweisungen:

Wenn alle Bits eines Busses mit demselben Wert belegt werden sollen, empfiehlt sich das `others`-Konstrukt:

```
.....
signal A, Q: std_logic_vector(15 downto 0);
.....
        A <= (others => '1');
        Q <= (others => '0');
.....
```

Damit wird A auf $FFFF und Q auf $0000 gesetzt.

Beispiele für Standard Schaltnetze




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

entity HALBADDIERER is
    port ( A : in std_logic;
           B : in std_logic;
           S : out std_logic;
           Cout : out std_logic);
end HALBADDIERER;

architecture Behavioral of HALBADDIERER is

begin
 S<=(not A and B) or (A and not B);
 Cout<=(A and B);
end Behavioral;
```

Nebenläufige Signalzuweisungen

Wenn man sich das Schaltbild ansieht, stellt man fest, dass die Signale S und Cout gleichzeitig oder zumindest zeitlich unabhängig voneinander erzeugt werden. Genau dies ist auch unter der nebenläufigen Zuweisung zu verstehen, die wir im VHDL-Code vorfinden:

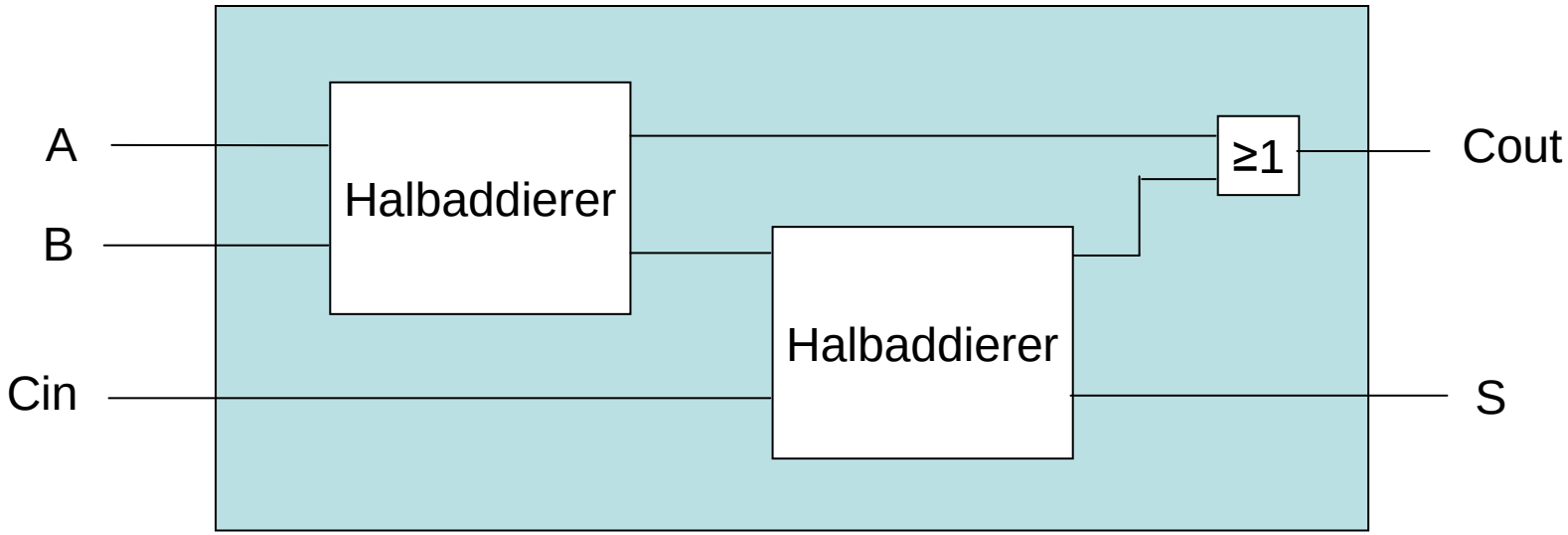
```
S<=(not A and B) or (A and not B);
Cout<=(A and B);
```

Es gibt also keine Reihenfolge, in der die Signale ihre neuen Werte bekommen. Dies ist ein Unterschied zu anderen Programmiersprachen und wird deshalb hier hervorgehoben. In diesem Sinne bitte ich auch noch einmal das NOR-Flipflop zu betrachten. Erst durch die Nebenläufigkeit macht der Code Sinn:

```
 Q <=   R nor QB;
QB <=   S nor Q;
```

Würde man dies als zwei sequentielle Befehle verstehen, könnte man das Flipflop nicht vernünftig setzen und rücksetzen.

Hierarchie: Ein Volladdierer



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

entity VOLLADDIERER is
    port ( A : in std_logic;
           B : in std_logic;
           Cin : in std_logic;
           S : out std_logic;
           Cout : out std_logic);
end VOLLADDIERER;
```

```
architecture Structural of VOLLADDIERER is

component HALBADDIERER
	port(  A: in std_logic;
		B: in std_logic;
		S: out std_logic;
		Cout : out std_logic);
	end component;

signal S1HA : std_logic;
signal C1HA : std_logic;
signal S2HA : std_logic;
signal C2HA : std_logic;
```

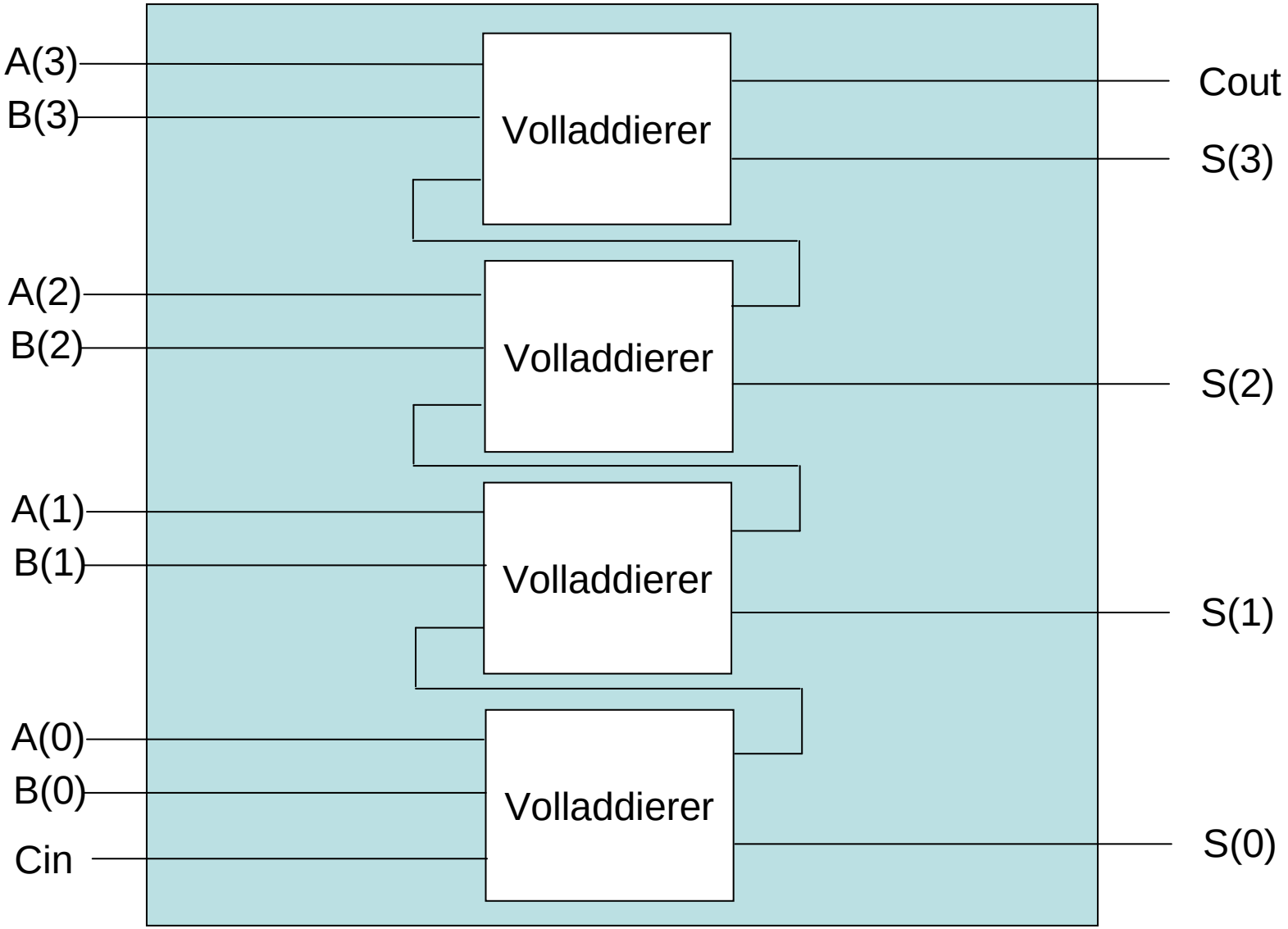
```
begin
        HA1: HALBADDIERER PORT MAP(
                A => A,
                B => B,
                S => S1HA,
                Cout => C1HA
        );
        HA2: HALBADDIERER PORT MAP(
                A => S1HA,
                B => Cin,
                S => S2HA,
                Cout => C2HA
        );
 Cout <= (C1HA or C2HA);
 S <= S2HA;
 end;
```

Komponentendeklaration

Mit einer Deklaration mit dem Schlüsselwort **`component`** macht man einen Verweis auf eine Komponente, die dem VHDL-System bereits bekannt ist. Danach folgt die Schnittstellenbeschreibung der Komponente, wobei die Signale in derselben Form und in derselben Reihenfolge auftreten, wie bei der **`entity`**-Deklaration der Komponente.
Dies entspricht genau dem typischen Fall im Hardwareentwurf. Man möchte eine Komponente verwenden, deren Funktionalität man benötigt. Damit diese richtig eingesetzt werden kann, müssen genau sich entsprechende Leitungen miteinander verbunden werden. Daher muss man (zum Beispiel im Datenblatt) eine Zuordnung der physikalischen Pins zu den logischen Anschlüssen haben, die in VHDL durch die **`component`**-Deklaration gegeben ist.

Bei der späteren **Instanzierung** der Komponente ist dann durch **`port map`** eine Zuordnung der äußeren Anschlüsse zu den an der Komponente vorhandenen Anschlüssen vorzunehmen. In der dafür erforderlichen Liste müssen wieder die Signale der **`entity`**-Deklaration in der richtigen Reihenfolge auf der linken Seite auftauchen gefolgt vom Zeichen => gefolgt vom äußeren Signal, an das das innere Signal angeschlossen werden soll.

Hierarchie: Ein 4-bit-Addierer



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

entity VIERBITADDIERER is
    port ( A : in std_logic_vector(3 downto 0);
           B : in std_logic_vector(3 downto 0);
           Cin : in std_logic;
           S : out std_logic_vector(3 downto 0);
           Cout : out std_logic
         );
end VIERBITADDIERER;
```

```
architecture Structural of VIERBITADDIERER is

component VOLLADDIERER
	port(
		A: in std_logic;
		B: in std_logic;
		Cin: in std_logic;
		S: out std_logic;
		Cout : out std_logic
		);
end component;

signal C : std_logic_vector(4 downto 1);
```

```
begin   VA0: VOLLADDIERER port map (
                A => A(0),
                B => B(0),
                Cin => Cin,
                S => S(0),
                Cout => C(1)          );
        VA1: VOLLADDIERER port map (
                A => A(1),
                B => B(1),
                Cin => C(1),
                S => S(1),
                Cout => C(2)          );
        VA2: VOLLADDIERER port map (
                A => A(2),
                B => B(2),
                Cin => C(2),
                S => S(2),
                Cout => C(3)          );
        VA3: VOLLADDIERER port map (
                A => A(3),
                B => B(3),
                Cin => C(3),
                S => S(3),
                Cout => C(4)          );
Cout <= C(4);
end;
```

Benutzung von Booleschen Operatoren:

In VHDL sind die Booleschen Operatoren

`not, and, nand, or, nor, xor, xnor`

vorhanden. `not` ist einstellig, alle anderen zweistellig. Die assoziativen Operatoren (`and, or, xor`) bedürfen keiner Klammerung, wenn mehr als zwei Operanden verknüpft werden sollen.

Beispiel:

```
Y <= not (A and B and C);      -- entspricht einem NAND3
Y <= (A nand B) nand C;        -- ist legal aber kein NAND3
Y <= A nand B nand C;          -- ist verboten!
```

Die Booleschen Operatoren können auf bit-Vektoren angewendet werden. Dabei wirken sie auf allen Bitstellen parallel. Natürlich müssen die Längen der Operanden und des Ergebnisses übereinstimmen.

Beispiel:

```
entity TEST is
        port (R : out std_logic_vector(3 downto 0);
end TEST;

architecture VERHALTEN of TEST is
signal A, B, C: std_logic_vector(7 downto 0);
signal Q: std_logic_vector(3 downto 0);

begin
        A <= “10101010“;
        B <= “01010101“;
        C <= A or B;
        R <= Q and B(7 downto 4);
end VERHALTEN;
```

Ansatz:

Wir wollen die ab jetzt die Sprache weiter kennen lernen anhand von digitalen Entwürfen mit wachsender Systemkomplexität. Dabei werden wiederkehrend folgende Entwurfsschritte durchlaufen:

- Digitaltechnische Problembeschreibung
- Hardwarebeschreibung mit VHDL
- Verifikation des VHDL-Codes durch Simulation
- Analyse des Syntheseergebnisses

Prozesse

Die uns bisher bekannten nebenläufigen Signalzuweisungen erlauben nur eine eingeschränkte Möglichkeit der Modellierung. Das Syntaxkonstrukt **`process`** erweitert die Möglichkeiten, Schaltnetze und Schaltwerke zu entwerfen und modellieren. Im Rahmen von diesen Prozessen können sogenannte sequentielle Anweisungen, die Verzweigungen und Scheifen erlauben, benutzt werden. Dadurch werden wir in die Lage versetzt, komplexere Digitalentwürfe zu machen.

Mit Prozessen werden Eingangssignale auf Ausgangssignale abgebildet. Alle Prozesse einer `architecture` werden nebenläufig ausgeführt. Die Kommunikation zwischen verschiedenen Prozessen erfolgt durch Verwendung lokaler Signale einer `architecture`.

Innerhalb der Prozesse gibt es spezielle Anweisungen, die sogenannten *sequential statements*: Deren besondere Eigenart ist es, dass sie im Simulator, wie aufeinander folgende Statements einer normalen Programmiersprache, nacheinander abgearbeitet werden. Im Synthesewerkzeug hingegen erfolgt eine Abbildung der einzelnen sequentiellen Anweisungen auf die bekannten Hardwarefunktionselemente.

Durch Prozesse bekommen wir ein Sprachkonstrukt an die Hand, das es uns ermöglicht, Funktionselemente zu beschreiben, die zeitliche Signalabfolgen definieren (z.B. Zähler).

In Prozessen sind unbedingte Signalzuweisungen erlaubt, selektive und bedingte Signalzuweisungen sind jedoch verboten. Stattdessen hat man hier die Möglichkeit, mit `case`- und `if`-Anweisungen zu arbeiten.

Bei unbedingten Signalzuweisungen innerhalb von Prozessen ist allerdings zu beachten, dass die tatsächliche Aktualisierung aller Signale immer erst am Prozessende erfolgt. Während der Prozessausführung ist es also nicht möglich, auf die aktuellen Werte bereits erfolgter Signalzuweisungen zuzugreifen. Ein und demselben Signal können innerhalb eines Prozesses mehrere Werte zugewiesen werden (da diese Zuweisungen im Prozess sequentiell interpretiert werden). Dies ist ja außerhalb von Prozessen verboten, weil durch die Nebenläufigkeit der Zuweisungen Konflikte entstehen würden. Der tatsächlich übernommene Signalwert bei Mehrfachzuweisung innerhalb eines Prozesses ist der Wert der zuletzt ausgeführten Signalzuweisung.

Beispiel für Mehrfachzuweisung innerhalb eines Prozesses

```
P1 : process(A)
   begin
      B <= '0';
      C <= B;
      B <= '1';
   end process P1;
```

Innerhalb von Prozessen können Variable unterschiedlicher Typen verwendet werden. Diese sind lokal und temporär, das heißt, sie existieren nur während der „Laufzeit“ des Prozesses. Ihr Wert ist außerhalb des Prozesses nicht zugreifbar. Wenn dies gewollt ist, muss der Wert der Variable noch im Prozess einem Signal zugewiesen werden. Dieses Signal steht dann außerhalb des Prozesses zur Verfügung. Allerdings steht der Wert der Variablen nach Beendigung des Prozesses wieder zur Verfügung, wenn der Prozess erneut aktiviert wird.
Variablen werden durch die Synthese entweder in real existierende Hardwarekomponenten gegossen oder wegoptimiert. Sie erlauben, ebenso wie das Konstrukt `process` selbst, die Verwendung hochsprachlicher Konstrukte zur Hardwarebeschreibung, die durch die Synthese automatisch und komfortabel in reale Bausteine umgesetzt wird, ohne dass der Entwerfer sich die Mühe der individuellen Realisierung machen muss.

Deklaration von Prozessen:

```
[<Prozessname>:] process [(<Empfindlichkeitsliste>)]
        <Deklarationsteil>
begin
        {<sequentielle Anweisungen>}
end process [<Prozessname>]
```

Deklaration von Prozessen:

```
[<Prozessname>:] process [(<Empfindlichkeitsliste>)]
        <Deklarationsteil>
begin
        {<sequentielle Anweisungen>}
end process [<Prozessname>]
```

<Prozessname>: Bezeichner; kann weggelassen werden, sollte aber verwendet werden, da es die Fehlersuche erleichtert.

<Empfindlichkeitsliste>: Eine geklammerte Liste von Signalen, durch Kommata getrennt. Eine Veränderung eines der Signale in der Empfindlichkeitsliste startet die Bearbeitung des Prozesses (bzw. aktiviert die durch den Prozess beschriebene Hardware).

<Deklarationsteil>: Variablen und Konstantendeklarationen.

Innerhalb des Prozesses: nur sequentielle Anweisungen. In **`begin`** und **`end`** geklammert.

Prozesse mit und ohne Empfindlichkeitsliste:

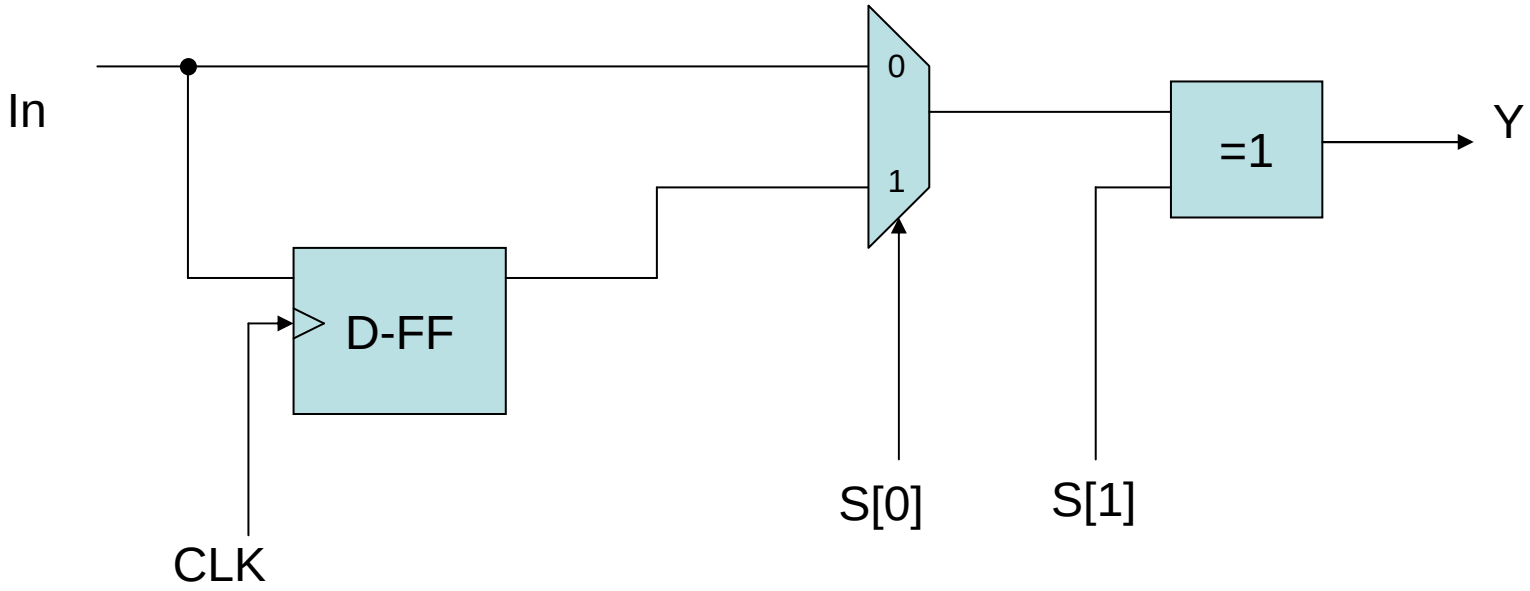
Mit Empfindlichkeitsliste: Prozess wird aktiviert, wenn ein Signal in der Empfindlichkeitsliste seinen Wert ändert.

Ohne Empfindlichkeitsliste: Prozess muss eine **`wait until`** Anweisung enthalten. Diese stellt eine Taktflankenabfrage dar. Solche Prozesse werden immer bis zur nächsten **`wait`** Anweisung ausgeführt. Die dabei auzuführenden Signaländerungen erfolgen bei Erreichen der **`wait`** Anweisung.

Ein Prozess mit Empfindlichkeitsliste kann keine wait Anweisungen enthalten.

Guter Stil der Programmierung (soll laut IEEE zur Vorschrift in VHDL werden): Ein Prozess darf nur eine **`wait`** Anweisung enthalten. Diese muss die erste Anweisung nach **`begin`** sein.

Beispiel für Verwendung von beiden Arten von Prozessen (mit und ohne Empfindlichkeitsliste). Eine typische CLB-Ausgangszelle:



```
entity OLMC is       -- OLMC: output logic macro cell
        port(   CLK, I: in std_logic;
                S:      in std_logic_vector(1 downto 0);
                Y:      out std_logic);
end OLMC;

architecture ARCH1 of OLMC is
signal TEMP1, TEMP2: std_logic;
begin
MUX: process(I, TEMP1, S(0))
        begin
                TEMP2 <= (I and not S(0)) or (TEMP1 and S(0));
        end process MUX;
FF: process
        begin
                wait until CLK = ‘1‘ and CLK‘event;
                TEMP1 <= I;
        end process FF;
    Y <= TEMP2 xor S(1);
end ARCH1;
```

Kommunikation zwischen den Prozessen innerhalb der **`architecture`** erfolgt mit den Signalen TEMP1 und TEMP2.

MUX ist ein kombinatorischer Prozess. Bei solchen ändert sich der Ausgang bei einer Änderung eines der Signale am Eingang. Daher müssen alle Eingangssignale in der Empfindlichkeitsliste auftauchen.

FF hingegen ist ein Prozess ohne Empfindlichkeitsliste.
Die ansteigende Flanke des Taktes CLK im D-Flipflop wird im Prozess FF durch die Formulierung

```
CLK = '1' and CLK'event
```

beschrieben. Entsprechend ließe sich eine fallende Flanke durch

```
CLK = '0' and CLK'event
```

beschreiben. Das Signalattribut `'event` ist Bestandteil der Sprache VHDL und bezeichnet einen beliebigen Signalwechsel.

Man kann jede nebenläufige Anweisung durch einen äquivalenten Prozess ersetzen, umgekehrt aber im allgemeinen nicht. Beispiel:

```
entity OLMC is      -- OLMC: output logic macro cell
        port(   CLK, I: in std_logic;
                S:      in std_logic_vector(1 downto 0);
                Y:      out std_logic);
end OLMC;

architecture ARCH2 of OLMC is
signal TEMP1, TEMP2: std_logic;
begin
FF: process
        begin
                wait until CLK = ‘1‘ and CLK‘event;
                TEMP1 <= I;
        end process FF;
    TEMP2 <= (I and not S(0)) or (TEMP1 and S(0));
    Y <= TEMP2 xor S(1);
end ARCH2;
```

Schaltnetze mit sequentiellen Anweisungen:

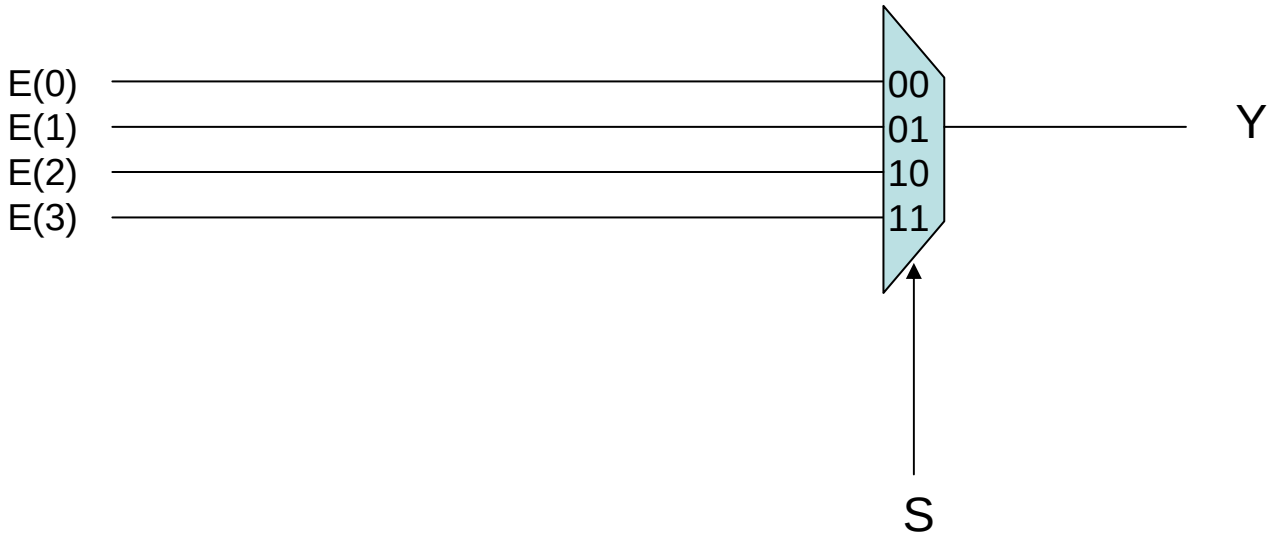
- case
- if
- for loop
- wait
- null

```
case <Kontrollausdruck> is
       when   <Testausdruck_1> => {<Sequentielle Anweisungen>};
       [when  <Testausdruck_2> => {<Sequentielle Anweisungen>};]
       ….
       [when  others => {<Sequentielle Anweisungen>};]
end case;
```

Kontrollausdruck ist dabei ein Signal oder eine Variable. Die Testausdrücke müssen erlaubte Werte des Kontrollausdrucks sein. Sie müssen sich gegenseitig ausschließen, d.h. kein Testausdruck darf in zwei oder mehr Verzweigungen berücksichtigt werden. Ferner müssen alle möglichen Werte für den Kontrollausdruck vorkommen. Wenn dies in den expliziten Testausdrücken bereits gegeben ist, kann auf den **others**-Zweig verzichtet werden.

```
case <Kontrollausdruck> is
        when   <Testausdruck_1> => {<Sequentielle Anweisungen>};
        [when  <Testausdruck_2> => {<Sequentielle Anweisungen>};]
        ….
        [when  others => {<Sequentielle Anweisungen>};]
end case;
```

Beispiel: 4x1Mulitplexer



```
entity MUX4X1_2 is
        port(   S:      in std_logic_vector(1 downto 0);
                E:      in std_logic_vector(3 downto 0);
                Y:      out std_logic);
end MUX4X1_2;

architecture VERHALTEN of MUX4X1_2 is
begin
MUX: process(S, E)
        begin
                case S is
                        when “00“ => Y <= E(0);
                        when “01“ => Y <= E(1);
                        when “10“ => Y <= E(2);
                        when “11“ => Y <= E(3);
                        when others => Y <= E(0);
                end case;
        end process MUX;
end VERHALTEN;
```

Mit der `case`-Anweisung kann man sehr komfortabel Wertetabellen in VHDL-Code umsetzen. Dies ist insbesondere für den FPGA-Entwurf interessant, weil kleinere Wertetabellen praktisch unverändert in die SRAM-Bereiche der CLBs geschrieben werden, wodurch die erwünschte Schaltnetzfunktion realisiert ist.

Wir nehmen als Beispiel die Wertetabelle eines Codeumsetzers, der eine im 8421-Code codierte Dezimalziffer am Eingang in eine AIKEN-codierte Zahl umsetzt. Ungültige 8421-Codeworte sollen in (das ungültige AIKEN-Codewort) 1010 umgesetzt werden.

4-Bit Codewandler: Dezimal -> Aiken

x_3	x_2	x_1	x_0	y_3	y_2	y_1	y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	1
0	1	0	0	0	1	0	0
0	1	0	1	1	0	1	1
0	1	1	0	1	1	0	0
0	1	1	1	1	1	0	1
1	0	0	0	1	1	1	0
1	0	0	1	1	1	1	1
1	0	1	0	1	0	1	0
1	0	1	1	1	0	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	0
1	1	1	0	1	0	1	0
1	1	1	1	1	0	1	0

```
entity CODEWANDLER is
        port(   E:      in std_logic_vector(3 downto 0);
                A:      out std_logic_vector(3 downto 0));
end CODEWANDLER;

architecture VERHALTEN of CODEWANDLER is
begin
CW: process(E)
        variable TEMP_A: std_logic_vector(3 downto 0);
        begin
                case E is
                        when “0000“ => TEMP_A := “0000“;
                        when “0001“ => TEMP_A := “0001“;
                        when “0010“ => TEMP_A := “0010“;
                        when “0011“ => TEMP_A := “0011“;
                        when “0100“ => TEMP_A := “0100“;
                        when “0101“ => TEMP_A := “1011“;
                        when “0110“ => TEMP_A := “1100“;
                        when “0111“ => TEMP_A := “1101“;
                        when “1000“ => TEMP_A := “1110“;
                        when “1001“ => TEMP_A := “1111“;
                        when others => TEMP_A := “1010“;
                end case;
                A <= TEMP_A;
        end process CW;
end VERHALTEN;
```

```
entity CODEWANDLER is
        port(   E:      in std_logic_vector(3 downto 0);
                A:      out std_logic_vector(3 downto 0));
end CODEWANDLER;

architecture VERHALTEN of CODEWANDLER is
begin
CW: process(E)
        variable TEMP_A: std_logic_vector(3 downto 0);
        begin
                case E is
                        when x“0“ => TEMP_A := x“0“;
                        when x“1“ => TEMP_A := x“1“;
                        when x“2“ => TEMP_A := x“2“;
                        when x“3“ => TEMP_A := x“3“;
                        when x“4“ => TEMP_A := x“4“;
                        when x“5“ => TEMP_A := x“B“;
                        when x“6“ => TEMP_A := x“C“;
                        when x“7“ => TEMP_A := x“D“;
                        when x“8“ => TEMP_A := x“E“;
                        when x“9“ => TEMP_A := x“F“;
                        when others => TEMP_A := x“A“;
                end case;
                A <= TEMP_A;
        end process CW;
end VERHALTEN;
```

```
entity VOLLADDIERER is
        port(   A, B, CIN:      in std_logic;
                S, COUT:        out std_logic;
end VOLLADDIERER;

architecture VERHALTEN of VOLLADDIERER is
begin
VA: process(A, B, CIN)
        variable TEMP_IN: std_logic_vector(2 downto 0);
        variable TEMP_OUT: std_logic_vector(1 downto 0);
        begin
                TEMP_IN := CIN&A&B;
                case TEMP_IN is
                        when “000“ => TEMP_OUT := “00“;
                        when “011“ => TEMP_OUT := “10“;
                        when “101“ => TEMP_OUT := “10“;
                        when “110“ => TEMP_OUT := “10“;
                        when “111“ => TEMP_OUT := “11“;
                        when others => TEMP_OUT := “01“;
                end case;
                S <= TEMP_OUT(0);
                COUT <= TEMP_OUT(1);
        end process VA;
end VERHALTEN;
```

Im Beispiel auf der letzten Folie wird dieselbe Technik angewendet, um einen Volladdierer zu modellieren. Die Möglichkeit, die drei Eingangssignale in einen `std_logic_vector` (TEMP_IN) zusammenzufassen erlaubt die sehr kompakte Formulierung in der `case`-Anweisung.

Man beachte, dass für den Vektor TEMP_OUT die Verwendung einer `variable` erforderlich ist. Hätte man TEMP_OUT als `signal` (außerhalb des Prozesses) deklariert, wäre auch ein syntaktisch korrektes VHDL-Programm entstanden, das aber eine andere Funktion modellieren würde (siehe nächste Folie). Warum?

```
entity VOLLADDIERER is
        port(   A, B, CIN:      in std_logic;
                S, COUT:        out std_logic);
end VOLLADDIERER;

architecture VERHALTEN of VOLLADDIERER is
signal TEMP_OUT: std_logic_vector(1 downto 0);
begin
VA: process(A, B, CIN)
        variable TEMP_IN: std_logic_vector(2 downto 0);
        begin
                TEMP_IN := CIN&A&B;
                case TEMP_IN is
                        when "000" => TEMP_OUT <= "00";
                        when "011" => TEMP_OUT <= "10";
                        when "101" => TEMP_OUT <= "10";
                        when "110" => TEMP_OUT <= "10";
                        when "111" => TEMP_OUT <= "11";
                        when others => TEMP_OUT <= "01";
                end case;
                S <= TEMP_OUT(0);
                COUT <= TEMP_OUT(1);
        end process VA;
end VERHALTEN;
```

Mehrere Alternativen in einer case-Anweisung können durch einen senkrechten (oder-) Strich in einer Zeile aufgeführt werden. Das ursprüngliche Beispiel des VOLLADDIERERS mit dieser Modifikation ist auf der folgenden Folie angegeben.

```
entity VOLLADDIERER is
        port(   A, B, CIN:      in std_logic;
                S, COUT:        out std_logic;
end VOLLADDIERER;

architecture VERHALTEN of VOLLADDIERER is
begin
VA: process(A, B, CIN)
        variable TEMP_IN: std_logic_vector(2 downto 0);
        variable TEMP_OUT: std_logic_vector(1 downto 0);
        begin
           TEMP_IN := CIN&A&B;
           case TEMP_IN is
                when “000“ => TEMP_OUT := “00“;
                when “011“ | “101“ | “110“ => TEMP_OUT := “10“;
                when “111“ => TEMP_OUT := “11“;
                when others => TEMP_OUT := “01“;
           end case;
           S <= TEMP_OUT(0);
           COUT <= TEMP_OUT(1);
        end process VA;
end VERHALTEN;
```