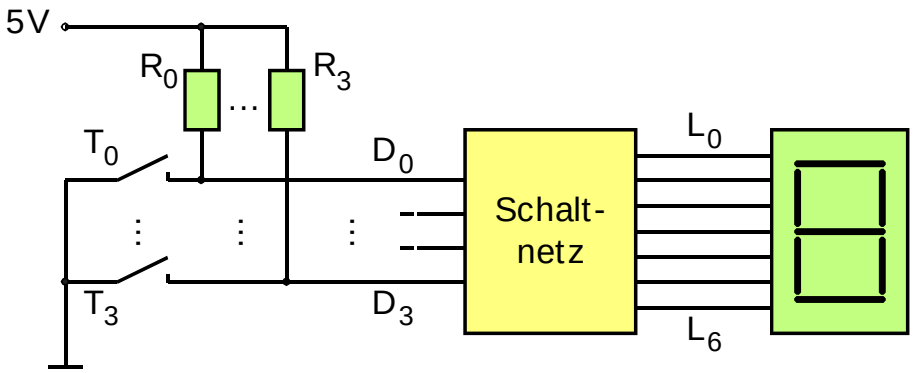
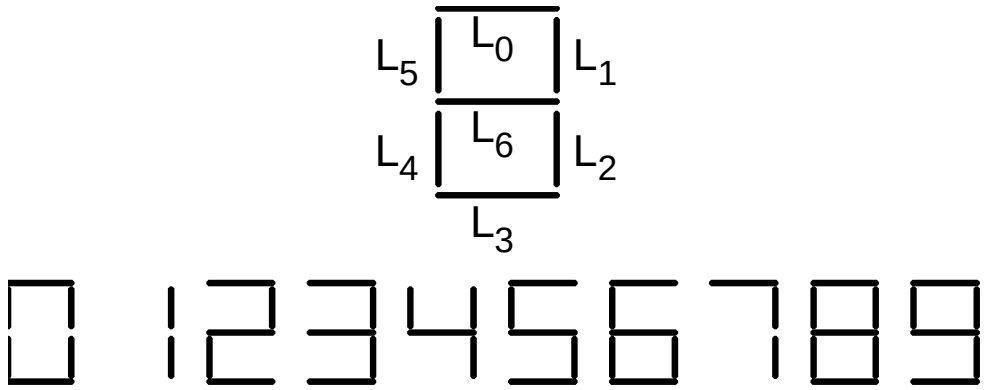
Kurzeinführung in VHDL

Überblick

1. Beispiel: Schaltnetz für 7-Segment-Anzeige



- Der Wert einer 4-stelligen Dualzahl (D)=(D_3,...,D_0) soll auf einer 7-Segment-LED-Anzeige angezeigt werden.
- Die Dualzahl wird über 4 Schalter eingestellt.
- Es sei (d)<10, für (d) ≥ 10 sei die Anzeige beliebig.
- Das Segment L_i der LED-Anzeige leuchtet, wenn L_i=1 ist.



Für das Schaltnetz ergibt sich folgende **Funktionstabelle**:

	Eingangsignale				*Ausgangsignale*						
	D_3	D_2	D_1	D_0	L_0	L_1	L_2	L_3	L_4	L_5	L_6
0	0	0	0	0	0	1	1	1	1	1	1
1	0	0	0	1	0	0	0	0	1	1	0
. . .	.	.	.	.	.	.	.	.	.	.	.
9	1	0	0	1	1	1	0	1	1	1	1
10	1	0	1	0	X	X	X	X	X	X	X
. . .	.	.	.	.	.	.	.	.	.	.	.
15	1	1	1	1	X	X	X	X	X	X	X

Alternativ kann die Schaltung auch durch Funktionsgleichungen dargestellt werden, z.B. $L_5 = /(/D_2 \cdot D_1 + /D_3 \cdot D_0)$

Beschreibung des Beispiels in VHDL:

Quartus-Projekt Einführung.qpf / Decoder1.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;          Standardbibliothek

ENTITY decoder1 IS                    Beschreibung der Ein und Ausgänge
  PORT (    D : IN  std_logic_vector(3 DOWNTO 0);  Eingangssignale
            L : OUT std_logic_vector(6 DOWNTO 0)   Ausgangssignale
      );                              (Achtung: Index in runden Klammern)
END decoder1;

ARCHITECTURE logic OF decoder1 IS     Beschreibung der Logik
BEGIN                   Beschreibung einer Funktionstabelle
  PROCESS(D)  <-------  mit IF ... THEN ... ELSE ... in einem Prozess
  BEGIN
```

```
                Eingangssignale              Ausgangssignale
    IF      D = "0000"   THEN   L <= "0111111"; -- Anzeige 0
    ELSIF   D = "0001"   THEN   L <= "0000110"; -- Anzeige 1
     . . .                 Dualzahl (Reihenfolge der Bits beachten!
    ELSIF   D =   X"9"   THEN   L <= "1101111"; -- Anzeige 9
    ELSE                        L <= "-------"; -- beliebige Anz.
    END IF;      Hexadezimalzahl
  END PROCESS;                        Don't care Werte      Kommentar
END logic;
```

Bemerkung: Für die Deklaration des signaltyps bei Signalbündel gibt es neben z.B. (6 downto 0) auch die Möglichkeit der Deklaration (0 to 6). Diese sollten Sie aber niemals verwenden.

2. VHDL-Syntax für Schaltnetze

VHDL (Very high speed Hardware Description Language) ist eine sehr umfangreiche Programmiersprache für die Synthese und Simulation von Hardware. Sie ist im internationalen IEEE1076-Standard definiert und zusammen mit VERILOG die seit 15 Jahren marktbeherrschende Beschreibungssprache für Hardware. Von der umfangreichen Syntax wird hier nur der für die Synthese wichtigste Teil dargestellt.

VHDL wurde gemeinsam mit ADA, einer Programmiersprache für Echtzeitsoftware, entwickelt und ähnelt in der Syntax der Programmiersprache PASCAL (DELPHI), die sich leider deutlich von der Syntax in C/C++ oder Java unterscheidet.

2.1 Aufbau eines VHDL-Programms

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;  Standardbibliothek

ENTITY entityName IS          --Beschreibung der Ein- und Ausgangssignale
  PORT ( signalname,... : modus signaltyp;   ← hier „;“
         ...
         signalname,... : modus signaltyp    ← Achtung: Hinter dem letzten Signal kein „;“
       );
END entityName;

ARCHITECTURE archName OF entityName IS  --Beschreibung der Funktion
     Deklaration von internen Signalen und Konstanten
BEGIN
     Beschreibung von Schaltnetzen mit Funktionstabellen
     oder Logikgleichungen

     Beschreibung von Schaltwerken mit Ablauftabellen
     oder Übergangsfunktionen
```

```
END archName;
```

VHDL trennt die Beschreibung der Ein- und Ausgänge (ENTITY) und der eigentlichen Logikfunktion (ARCHITECTURE). Eine größere Schaltung kann aus mehreren Teilschaltungen (ENTITY+ARCHITECTURE) hierarchisch zusammengesetzt werden, die über eine Strukturbeschreibung (sh. Kapitel 5) zusammengeschaltet werden.

Namen für Entitys, Architectures, Signale usw. (Bezeichner):

- In der Regel sollte als `entityName` der Name der Datei gewählt werden, also z.B. `entityName=logic31`, wenn das Design in der Datei `logic31.vhd` steht.
- erstes Zeichen muss ein Buchstabe sein, '_' zulässig, beliebige Länge
- VHDL ist nicht Case-Sensitiv (d.h. keine Unterscheidung von Groß- und Kleinbuchstaben), auch Schlüsselwörter wie ARCHITECTURE dürfen beliebig klein oder groß geschrieben werden!)

Kommentare

`--` text **Kommentar** bis zum Zeilenende

Leider gibt es keine mehrzeiligen Kommentare, so dass jede Kommentarzeile mit „`--`“ eingeleitet werden muss.

Block-Begrenzer - „Geschweifte“ Klammern „{„ und „}“

Wo in C/C++ die bekannten geschweiften Klammern stehen, um einen Block von Befehlen zusammenzufassen, stehen in VHDL die Schlüsselwörter **`BEGIN`** und **`END`**.

2.2 Ein- und Ausgangssignale

Signale werden über ihren Namen `signalname`, ihre Signalrichtung `modus` und ihren Signaltyp („Datentyp“) `signaltyp` definiert:

Für die **Signalrichtung** `modus` sind u.a. folgende Werte zulässig:

- **`IN`** Eingangssignal
- **`OUT`** Ausgangssignal
- **`INOUT`** bidirektionales Signal

Reine Ausgangssignale dürfen in VHDL nur geschrieben, aber nicht gelesen werden. Falls sie in irgendeinem Schaltungsteil innerhalb der Architektur auch als Eingangssignal verwendet werden, sollten interne Zwischensignale definiert werden.

Als **Signaltyp** `signaltyp` ist u.a. folgendes zulässig:

- **`std_logic`**
  1-bit-Signal mit den Werten **`'0'`**, **`'1'`**, **`'Z'`** (hochohmig), **`'-'`** (don’t care), **`'X'`** (unknown) und weiteren, seltener benötigten Werten
- **`std_logic_vector ( endindex DOWNTO anfangsindex )`**
  n-bit-Signal, dessen Wert als sogenannter Bit-String angegeben wird.
  Der Bit-String kann als Dualzahl dargestellt sein, z.B. `"01Z-"`, oder als hexadezimalzahl `X"3F0"` (mit vorangestelltem X). Die Anzahl der Bits wird indirekt über die Indexwerte in absteigender (`DOWNTO`) Reihenfolge angegeben. Als Index sind alle positiven Dezimalzahlen inkl. 0 zulässig.

Wertzuweisungen an Signale

Die Zuweisung eines Signalwertes erfolgt mit `signalname <= wert`, z.B.

- `L <= "0110000";`
- `L(3 DOWNTO 2) <= "00"; -- Zuweisung an einen Slice`
  `-- des Signalbündels`

Achtung:

- Die Werte von 1bit-Signalen müssen in einfache Anführungszeichen `'...'` eingeschlossen werden, z.B. `'1'`. Die Werte von n-bit-Signalen müssen in doppelten Anführungszeichen `"..."` stehen, z.B. `"1011"`.
- Die Stellenzahl bei der Wertangabe eines n-bit-Signal muss exakt so groß sein wie bei der Signaldefinition über die Angabe des Anfangs- und des Endindexes festgelegt, d.h. auch führende Nullen müssen angegeben werden.
- Teile eines n-bit-Signals können ebenfalls verwendet werden, z.B.

```
D : IN std_logic_vector(3 DOWNTO 0) -- 4bit Eingangssignal (D)
D(2 DOWNTO 0)  --Bit 2 bis 0 des Signals
D(3)           --Bit 3 des Signals
```

Index in VHDL in runden Klammern !

2.3 Funktionstabellen und Funktionsgleichungen

Jede Funktionstabelle wird als eigener „Prozess“ definiert. Während in einer ‚normalen‘ Programmiersprache wie C/C++ die Befehle zeitlich hintereinander („**Sequential Statements**“) ausgeführt werden, werden VHDL-Prozesse nebenläufig bearbeitet, d.h. alle Prozesse innerhalb eines VHDL-Programms werden als quasi-gleichzeitig betrachtet und bei der Simulation quasi-gleichzeitig ausgeführt („**Concurrent Statement**“). Die Anweisungen innerhalb eines Prozesses dagegen werden wie üblich sequentiell ausgeführt.

Die Nebenläufigkeit ist notwendig, da VHDL das Verhalten realer digitaler Schaltungen beschreiben soll, bei denen ja viele Gatter und Flipflops gleichzeitig aktiv sind. Die Syntax für einen Prozess zur Darstellung einer Funktionstabelle ist:

```
nameProzess: PROZESS(eingangssignal1, eingangssignal2, ... )
     Deklaration von Konstanten usw.
BEGIN
     IF     bedingung1  THEN  signalzuweisung;
                              signalzuweisung;
                              . . .
     ELSIF bedingung2   THEN  signalzuweisung;
                              . . .
     . . .
     ELSE                     signalzuweisung;
                              . . .
     END IF;
END PROCESS nameProzess;
```

*Liste **aller** Eingangssignale der Funktionstabelle*

Achtung: ELSIF ohne ‚e‘

- In der (...) müssen die Namen aller Eingangssignale der Funktionstabelle aufgelistet werden. Diese Liste wird als „**Sensivity List**“ bezeichnet, weil in der Simulation der

Prozess immer ausgeführt werden muss, wenn sich eines dieser Eingangssignale ändert. Der Name des Prozesses ist optional und darf entfallen.

- Jeder `IF`-, `ELSIF`- bzw. `ELSE`-Zweig entspricht einer Zeile der Funktionstabelle.
- In der Bedingung wird der Wert der Eingangssignale abgefragt, z.B.

```
IF     D = "0000"  THEN  L <= "1111110";
```

Diese Signalzuweisung wird ausgeführt, wenn das Eingangsignal D den Wert `"0000"` hat.

- Es können auch mehrere Bedingungen verknüpft werden. Die obere Anweisung kann z.B. auch als

```
IF     D(3)= '0'  AND D(2 DOWNTO 0) = "000"  THEN ...
```

oder

```
IF NOT(D(3)= '1') AND D(2 DOWNTO 0) = "000"  THEN ...
```

geschrieben werden. Dazu stehen die logischen Operatoren `AND, OR, XOR, NAND, NOR, NOT` zur Verfügung.

- In den Bedingungen dürfen keine Don't-Care-Werte vorkommen. Falls bei den Eingangssignalen in der Funktionstabelle Don't-Cares vorkommen, z.B. falls in einer Zeile das Eingangssignal `D(2)` beliebig sein darf, schreibt man

```
IF D(3)= '0'  AND D(1 DOWNTO 0) = "00"  THEN ...
```

d.h. für `D(2)` gibt es einfach keine Bedingung.

- Bei den Signalzuweisungen sind Don't-Care-Werte zulässig, z.B. `L <= "01--"`

Achtung:

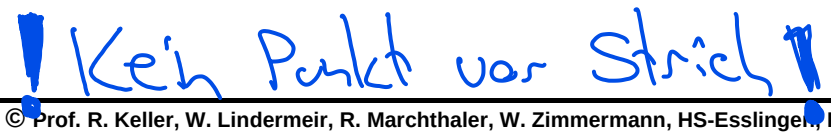
- In VHDL ist „**=**" der Vergleichsoperator für Gleichheit. Mit „**/=**" kann auf Ungleichheit getestet werden. Datentypen, die eine Ordnungsrelation implizieren (z.B. Integer) können in boolschen Ausdrücken mit „**<**", „**>**", „**<=**" oder „**>=**" verglichen werden. Der Vergleich von zwei Signalbündeln mit diesen Vergleichsoperatoren ist nicht definiert. Denken Sie daran, dass dieser Vergleich ja eine Codierung voraussetzen würde und das Vergleichsergebnis von z.B. „111" und „000" davon abhängt, ob es sich um eine Dual (Vergleich 7 mit 0) oder eine 2er-Komplement (Vergleich -1 mit 0) Codierung handelt.
- Wichtig ist, dass das Ausgangssignal der kombinatorischen Logik auf jedem möglichen Pfad durch den Prozess einen Wert zugewiesen bekommt. Dies kann mit einer Default-Zuweisung vor dem if-Konstrukt oder einem else-Zweig sichergestellt werden.
- In Bedingungen keine Don't-Care-Werte verwenden.
- Hinweis zum Operator „<=": Entweder kleiner gleich (sh. oben) oder Signalzuweisung. VHDL ist eine objektorientierte Sprache und dieser Operator ist überladen.

Funktionsgleichungen

Statt durch Funktionstabellen können Schaltnetze auch durch Funktionsgleichungen beschrieben werden, z.B. $L_5 = /(/D_2 \cdot D_1 + /D_3 \cdot D_0)$. Die Syntax lautet:

```
L(5) <= NOT( (NOT(D(2)) AND D(1)) OR (NOT(D(3)) AND D(0)) );
```

Dazu stehen die logischen Operatoren **`AND, OR, XOR, NAND, NOR, NOT`** zur Verfügung. Beachten Sie dabei, dass die Und- und die Oder-Operationen in VHDL völlig gleichberechtigt sind und von links nach rechts ausgeführt werden, während wir Menschen „Punkt vor Strich" interpretieren, d.h. in unserer in der Vorlesung Digitaltechnik vereinbarten Schreibweise haben die UND-Operationen eine höhere Priorität als die Oder-Operationen, so wird A v B·C von uns als A v (B · C) interpretiert, in VHDL dage-



gen als (A v B) · C. Um Fehlinterpretationen zu vermeiden, sollten Sie daher die Reihenfolge durch Klammern „(...)„ eindeutig vorgeben:

Syntax in VHDL	Logische Funktion
`A OR B AND C` entspricht: `(A OR B)AND C`	(A v B) · C
`A OR (B AND C)`	A v B·C = A v (B·C)

Dabei dürfen nicht nur einzelne Signale, sondern auch Signalvektoren in einer Funktionsgleichung vorkommen, aber alle Signale müssen dieselbe Bit-Anzahl haben.

Funktionsgleichungen stehen normalerweise innerhalb des Architecture-Begin-End-Blocks **außerhalb** von Prozessen. Man nennt sie dann **nebenläufige Signalzuweisungen (concurrent signal assignment)**. Sie werden nebenläufig (also gleichzeitig) mit allen anderen nebenläufigen Befehlen der Architektur, also z.B. anderen nebenläufigen Signalzuweisungen und Prozessen ausgeführt, d.h. die Reihenfolge dieser Signalzuweisungen und Prozesse spielt keine Rolle.

Jedes Signal (abgesehen von Tristate-Signalen) darf nur durch genau eine nebenläufige Anweisung (z.B. durch eine nebenläufige Signalzuweisung oder durch einen prozess) geschrieben werden.

Hinweise:

- Es gibt noch eine Vielzahl weiterer nebenläufiger Befehle, die wir hier nicht betrachten. Man erkennt sie daran, dass Sie direkt in der Architektur der Entwurfseinheit stehen.
- Die Abarbeitung innerhalb von prozessen erfolgt sequentiell, also nacheinander, wie man es von normalen Programmiersprachen her kennt. Diese Befehle nennt man sequentielle Befehle (sequenital statements). Es gibt auch Signalzuweisungen innerhalb von Prozessen. Diese werden als **nicht-nebenläufige** Signalzuweisungen (sequential signal assignment) bezeichnet. Sie werden betrachtet, wenn Sie bei der Ausführung des Prozesses an die Reihe kommen.

Konstanten und interne Signale

Gelegentlich wird die Darstellung übersichtlicher, wenn innerhalb einer Architektur interne Signale oder Konstanten eingeführt werden. Die Syntax lautet:

```
SIGNAL    signalname : signaltyp ;
CONSTANT  constantenname : signaltyp := wert ;
```

Als Signaltypen sind u.a. wieder **`std_logic`** und **`std_logic_vector`** möglich.

Sowas wie Variablen?

Bsp.:

```
ARCHITECTURE logic OF Dekoder IS
    SIGNAL     Y : std_logic;          -- internes Signal
    CONSTANT   H : std_logic := "1";   -- Konstante
BEGIN
    . . .
    Y    <= NOT(D(3));
```

```
    L(5) <= Y XOR H;
    . . .
```

Interne Signale oder Konstanten erhöhen den Schaltungsaufwand nicht, da sie bei der Logikoptimierung automatisch entfernt werden.

Vereinfachte Syntax für Funktionstabellen

Statt mit einer IF-THEN-ELSE-Kette können Funktionstabellen auch mit CASE-WHEN-Befehlen beschrieben werden:

Quartus-Projekt Einführung.qpf / Decoder2.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY decoder2 IS  --- Interface: Beschreibung der Ein-/Ausgänge
  PORT (    D : IN  std_logic_vector(3 DOWNTO 0);
            L : OUT std_logic_vector(6 DOWNTO 0)
       );
END decoder2;

ARCHITECTURE logic OF decoder2 IS     --- Beschreibung der Logik
BEGIN

  PROCESS(D) – Funktionstabelle des Siebensegmentdekoders mit CASE-WHEN
  BEGIN       -- statt IF-THEN-ELSE
    CASE D IS
    WHEN "0000" => L <= "0111111"; -- Anzeige 0
    WHEN "0001" => L <= "0000110"; -- Anzeige 1
     . . .
    WHEN  X"9"  => L <= "1101111"; -- Anzeige 9
    WHEN OTHERS => L <= "-------"; -- beliebige Anzeige für den Rest
    END CASE;
  END PROCESS;

END logic;
```

Aber Achtung: Einschränkungen bei der vereinfachten Syntax

- Bei den Werten der Eingangsgrößen dürfen keine Don't Cares ‚-' vorkommen, sie erhalten sonst bei der Schaltungssynthese nicht die von Ihnen gewünschte Schaltung!
- Als Eingangsgröße ist nur ein einziges Signal bzw. ein Signalbündel mit einem einzigen Namen zulässig (hier D). Falls mehrere Signale mit unterschiedlichen Namen als Eingangsgrößen verwendet werden sollen, muss in der Architektur ein internes Signalbündel von z.B. Typ std_logic_vector eingeführt werden (sh. unten).

Signalbündel (sogenannte **Aggregate)**

Mehrere Einzelsignale können zu einem Signalbündel zusammengefasst werden, z.B.:

```
    -- Einzelsignale
    SIGNAL D0, D1, D2, D3 : IN STD_LOGIC;
    SIGNAL L0, L1, L2, L3, L4, L5, L6 : OUT STD_LOGIC;

    -- Signalbündel
    SIGNAL X: Std_Logic_Vector(3 downto 0);
    SIGNAL Y: Std_Logic_Vector(6 downto 0);
```

```
-- Zusammenfassung von Einzelsignalen zu einem Bündel
X <= (D3, D2, D1, D0);
-- Aufspaltung eines Signalbündels in Einzelsignale
(L6, L5, L4, L3, L2, L1, L0) <= Y;
```

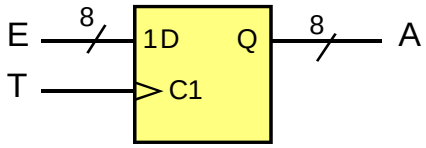
Mit Hilfe des Schlüsselworts OTHERS lassen sich auch Zuweisungen für mehrere Elemente eines Bit-Strings bzw. Signalbündels vornehmen:

```
X <= ('0','0', others => '1');
```

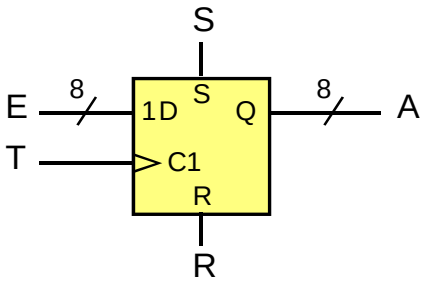
Die ersten beiden Bits von X erhalten hier den Wert ‘0’, der Rest den Wert ‘1’.

3. Flipflops und Register

Ein oder mehrere Signale können abhängig von einem Taktsignal in einem flankengesteuerten Flipflop oder einem Register gespeichert werden.



8bit-Register aus D-Flipflops



8bit-Register aus D-Flipflops mit asynchronem Preset und Reset

Die Beschreibung eines Flipflops erfolgt in einem PROCESS. Darin wird die Taktflankensteuerung mit der folgenden Syntax beschrieben:

`IF RISING_EDGE (takt_signal) THEN ...` **positive Flankentriggerung**
`IF FALLING_EDGE(takt_signal) THEN ...` **negative Flankentriggerung**.

Über weitere IF–ELSIF—ELSE-Zweige können auch asynchrone Rücksetz- bzw. Setzsignale realisiert werden.

Das folgende Beispiel zeigt zwei Funktionsgleichungen sowie ein Toggle-Flipflop, das jeweils bei der positiven Flanke des Taktsignals `clk` seinen Zustand wechselt. Beachten Sie, dass das eigentliche Ausgangssignal `Y(3)` des Flipflops, das in der Entity-Deklaration als `OUT` definiert ist, in VHDL nur geschrieben, aber nicht gelesen werden kann. Beides ist hier aber notwendig, da das Ausgangssignal sowohl auf der linken als auch auf der rechten Seite der Flipflop-Beschreibungsgleichung vorkommt. Daher wird für das Flipflop das interne Signal Q eingeführt.

Quartus-Projekt Einführung.qpf / Diverses.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY diverses IS  --- Interface: Beschreibung der Ein-/Ausgänge
  PORT (  D    : IN  STD_LOGIC_VECTOR(3 downto 0);
```

```
          clk  : IN  STD_LOGIC;
          Reset: IN  STD_LOGIC;
          Y    : OUT STD_LOGIC_VECTOR(3 downto 0)
       );
END diverses;

ARCHITECTURE logic OF diverses IS     --- Beschreibung der Logik
   SIGNAL Q : std_logic; -- Internes Signal für das Flipflop
BEGIN

     -- Funktionsgleichung für XOR-Funktion aus AND, OR und NOT
     Y(0) <= ( D(0) AND NOT(D(1)) ) OR ( D(1) AND NOT(D(0)) );
     -- bzw. direkt mit XOR-Funktion
     Y(1) <= D(0) XOR  D(1);

     -- Toggle-Flipflop mit Reset, kippt bei positiver Flanke im Takt clk
     PROCESS (clk, Reset)
     BEGIN
          IF Reset = '1' THEN
               Q <= '0';
          ELSIF rising_edge(clk) THEN
               Q <= NOT(Q);
          END IF;
     END PROCESS;

     Y(3) <= Q;   -- Ausgang des Flipflops

END logic;
```

Statt die Funktion rising_edge() (bzw. falling_edge) zu verwenden, finden Sie häufig auch die Bedingung **`(clk'EVENT AND clk='1')`**, die wahr wird, wenn das Signal `clk` einen anderen Wert annimmt (**`'EVENT`**) und anschließend den Wert ‚1' hat. Dies entspricht der Bedingung für eine positive Signalflanke. Der Ausdruck **´EVENT** ist ein sogenanntes VHDL-**Attribut**. VHDL kennt sehr viele Attribute, mit **`A'LENGTH`** kann man z.B. die Länge des Signalbündels A ermitteln.

4. Beschreibung von Zählern in VHDL

Als Beispiel wird ein mod 2^n Aufwärtszähler mit der periodischen Zählfolge 0, 1, 2, …, 2^n-1 betrachtet, der mit den Signalen `synReset` und `asynReset` auf 0 zurückgesetzt werden kann und nur zählt, wenn das Freigabesignal `Enable` aktiv ist. Der Zählerstand soll sich mit der positiven Taktflanke ändern. Beim Erreichen des maximalen Zählerstands wird zusätzlich der Ausgang Qmax aktiv.

```
LIBRARY ieee;                    Quartus-Projekt Einführung.qpf / Zaehler.vhd
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all; -- Zusätzliche Bibliothek für arith. Operationen

ENTITY Zaehler IS
  GENERIC(n : integer:= 8);   -- Breite des Zählers in bit
  PORT( clk : IN STD_LOGIC;
        synReset, asynReset : IN STD_LOGIC;
        Enable : IN STD_LOGIC;
```

```
        Q : OUT STD_LOGIC_VECTOR(n-1 downto 0);
        Qmax : OUT STD_LOGIC
     );
END Zaehler;

ARCHITECTURE logic OF Zaehler IS
  CONSTANT zmax : integer := 2**n -1; -- maximaler Zählerstand 2^n-1
  SIGNAL z, znext : integer RANGE 0 TO zmax; -- Zählerstand
BEGIN

  Comb: PROCESS (z, synReset, Enable) --Prozess Übergangs-
                                      -- u. Ausgangsfunktion
  BEGIN
  -- Übergangsfunktion: Nächster Zählerstand----------------------------------------
    znext <= z;

    IF (synReset = '1') THEN
      znext <= 0;             --Synchrones Rücksetzen des Zählers
    ELSE
      IF (Enable = '1') THEN
        IF (z < zmax) THEN
          znext <= z + 1;               --Zählen
        ELSE
          znext <= 0;                   --Überlauf
        END IF;
      END IF;
    END IF;
  -- Ausgangsfunktion: Ausgangssignale ----------------------------------------------
    Q <= std_logic_vector(to_unsigned(z, n)); --Zählerstand:
                                  Umwandlung Integer in n bit std_logic_vector
    IF ( z >= zmax) THEN                                    (Dualcode)
      Qmax <= '1';              --Anzeige Maximalwert erreicht
    ELSE
      Qmax <= '0';
    END IF;
  END PROCESS;
  Trigger: PROCESS (clk, asynReset)  --Prozess für Zustandsübergang
  BEGIN
    IF (asynReset = '1') THEN     --Asynchrones Rücksetzen des Zählers
      z <= 0;
    ELSIF rising_edge(clk) THEN  --Zählerstand bei pos. Taktflanke ändern
      z <= znext;
    END IF;
  END PROCESS;

END logic;
```

Statt als `std_logic_vector` wird der Zählerstand mit dem Datentyp `INTEGER` definiert. Durch `RANGE ...` wird dabei der Zählbereich (und damit letztlich die Stellenzahl des Zählers) definiert. Für diesen Datentyp sind dann die Operationen ‚+' oder ‚-', möglich sowie die Vergleichsoperationen ‚<', ‚<=', ‚=', ‚>' oder ‚>=' zulässig, so dass das Auf- oder Abwärtszählen genauso wie das Erkennen von bestimmten Zählerständen,

Über- oder Unterläufen einfach zu beschreiben ist. Für die Ausgabe muss der Zählerstand z dann allerdings von **`Integer`** wieder in ein n Bit Signal umgewandelt werden. Für die Typumwandlung existieren in der Bibliothek **`ieee.numeric_std.all`** die folgenden Funktionen:

```
std_logic_vector(to_unsigned(z, n))
```

dabei ist **`z`** der Integer-Wert, **`n`** die Länge des Bit-Vektors. Mit **`to_unsigned()`** wird eine natürliche Zahl (Betragszahl), mit **`to_signed()`** eine ganze Zahl (2er-Komplement) umgewandelt. In der umgekehrten Richtung ist eine Umwandlung mit

```
to_integer(unsigned(v))
```

möglich. Dabei ist v der Bit-Vektor, das Ergebnis ist eine Betragszahl. Soll in eine 2er-Komplementzahl gewandelt werden, muss **`signed()`** statt **`unsigned()`** benützt werden.

Für die Eingabe von Integer-Werten gilt:

`9497`, **`-33`**	positive bzw. negative Dezimalzahlen
`10#9497#`	andere Schreibweise für eine Dezimalzahl
`16#AB89#`	Hexadezimalzahl **`16#...#`**

Achtung: Verwechseln Sie Integer-Werte nicht mit Bit-Strings wie **`X"AB89"`**. Diese Typen dürfen nicht gemischt werden. Integer-Werte dürfen nur Größen vom Typ **`INTEGER`** zugewiesen werden, Bit-Strings nur Größen vom Typ **`std_logic_vector`**. Gegebenenfalls ist eine der o.g. Typumwandlungsfunktionen notwendig.

Im oberen Beispiel wurde die Breite n des Zählers als **`GENERIC`**-Parameter innerhalb der ENTITY-Deklaration definiert. Die zugehörige Syntax lautet:

```
GENERIC ( parameterName1 : parameterTyp := defaultWert );
```

`GENERIC`-Werte sind sinnvoll, wenn ein Block in allgemeiner Form beschrieben wird, bei der Verwendung aber mit unterschiedlichen Parameterwerten eingesetzt wird, z.B. in einer hierarchischen Schaltungsbeschreibung (sh. Abschnitt 5). In einer grafischen Schaltungsbeschreibung in Altera Quartus kann ein generischer Wert als Parameter eines Blocks definiert werden (mit rechter Maustaste auf Block klicken – Block Properties – Dialogbox Parameters.). Dieser Wert überschreibt dann den in der Entity-Definition bei GENERIC festgelegten Default-Wert.

Anstatt das Signal z als INTEGER mit begrenztem Wertebereich zu definieren, kann man auch einen „Unter"-typ von `integer` definieren, z.B.

```
SUBTYPE  counterTyp IS integer RANGE 0 TO zmax;
SIGNAL z : counterTyp;
```

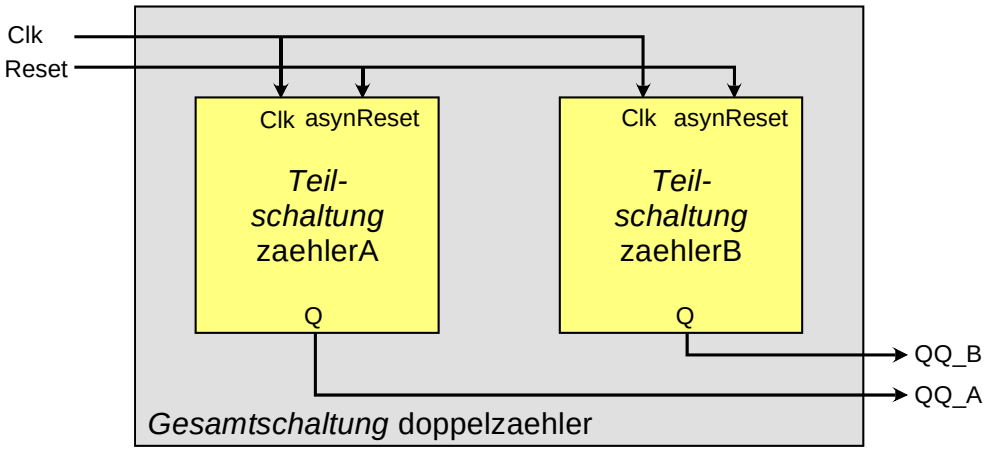
Diese Vorgehensweise bietet sich an, wenn mehrere Signale von diesem Typ benötigt werden, da die Bereichsangabe dann nur einmal gemacht werden muss und bei allen derartigen Signalen dann konsistent ist. Untertypen lassen sich von jedem Datentyp definieren und sind mit dem zugehörigen Oberdatentyp kompatibel.

Die oben praktizierte Aufteilung in zwei Prozesse und die Trennung in die Übergangsfunktion, die Ausgangsfunktion und den Zustandsübergang erscheint zunächst aufwendig, ist bei bei Zählern mit einigen zusätzlichen Ein- und Ausgängen wie generell bei Zustandsautomaten (vergl. auch Abschnitt 7) aber sinnvoll und sehr empfehlenswert.

5. Hierarchischer Schaltungsaufbau (Strukturbeschreibungen)

Um Schaltungen übersichtlicher zu gestalten, kann man sie in Teilschaltungen zerlegen und in einer übergeordneten Schaltung zusammenschalten:

Beispiel: Schaltung mit zwei Zählern aus Kapitel 4



Die zugehörige Schaltungsbeschreibung sieht folgendermassen aus:

Quartus-Projekt Einführung.qpf / Doppelzaehler.vhd

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY doppelzaehler IS
  PORT (  clk       : IN STD_LOGIC;
          asynReset : IN STD_LOGIC;
          QQ_A      : OUT STD_LOGIC_VECTOR(2 downto 0);
          QQ_B      : OUT STD_LOGIC_VECTOR(1 downto 0)
     );
END doppelzaehler;

ARCHITECTURE logic OF doppelzaehler IS
BEGIN
  ZaehlerA: work.Zaehler GENERIC MAP(3)       --Positional Association
            PORT MAP(clk, '0', asynReset, '1', QQ_A, open);
  ZaehlerB: work.Zaehler GENERIC MAP(n => 2) --Named Association
            PORT MAP(clk => clk, synReset => '0', Enable => '1',
                     asynReset => asynReset, Q => QQ_B,
                     Qmax => open);
END logic;
```

Es wird vorausgesetzt, dass eine Entwurfseinheit mit passender Schnittstelle (hier die Entity des Zählers mit der aus Kapitel 4 bekannten Schnittstelle) bereits vom VHDL-Compiler übersetzt wurde und in der Default-Bibliothek WORK abgelegt wurde. Die Instanziierung der Komponenten ist eine weitere nebenläufige Anweisung in VHDL und erfolgt mit der Syntax

```
InstanzName: WORK.nameDerReferenz  GENERIC MAP(…)
                                   PORT MAP(…);
```

In der **GENERIC MAP** werden die Parameter der Komponente festgelegt. Wenn die Komponente keine **GENERIC** Parameter besitzt oder die dort definierten Defaultwerte verwendet werden sollen, entfällt **GENERIC MAP**.
In der **PORT MAP** werden die Ein- und Ausgangssignale an die Komponenten „angeschlossen“.

Wie man sieht, werden dabei bei `ZaehlerA` und `ZaehlerB` unterschiedliche Syntaxoptionen verwendet. Bei `ZaehlerA` wird die von C/C++ und anderen Programmiersprachen bekannte Zuweisung von Parametern bzw. Signalen verwendet, bei der die Parameter bzw. Signale einfach in derselben Reihenfolge angegeben werden (***Positional Parameter Association***) wie bei der Definition der Schnittstelle in der **ENTITY**-Deklaration der Basiskomponente. Dieses Verfahren ist gefährlich, wenn die Reihenfolge der Parameter in der **ENTITY** von `Zaehler` geändert wird, ohne dass die Reihenfolge der Parameter bei der Verwendung der Componente in `ZaehlerA` bzw. `ZaehlerB` passend angepasst wird. Dieser Fehler kann insbesondere dann sehr leicht passieren, wenn die **ENTITY**-Definition, wie beispielsweise bei der grafischen Definition von Schaltungen wie in ALTERA QUARTUS automatisch erzeugt wird.

Daher ist es in strukturierten Schaltungsbeschreibungen grundsätzlich sicherer, den bei `ZaehlerB` verwendeten Parameteraufruf mit ***Named Parameter Association*** zu verwenden. Dabei wird der Name des Parameters oder des Signals der Referenz angegeben und mit dem Operator ‚=>’ auf den Namen des Parameters, des Signals oder der Konstanten der aktuellen Entwurfseinheit verknüpft. In diesem Fall spielt die Reihenfolge der Parameter keine Rolle, weil die Zuordnung über die Namen erfolgt. Z.B. ist in der Instantiierung des ZählersB das Signal Enable vorgezogen.

Hinweis: Falls bei Instantiierungen nicht alle Ausgangssignale einer Komponente verwendet werden, kann dies in der Association-Liste durch das VHDL-Tag **open** erfolgen.

Der **Instanzname** ist in VHDL nicht notwendig. Er ist aber sinnvoll, weil man dann auf interne Signale innerhalb der instantiierten Komponenten zugreifen kann. Beispielsweise kann man sich mit ModelSim in einem Impulsdiagramm interne Signale von solchen Komponenten ansehen, wenn man den Namen des Signals mit Pfadangabe im add wave Befehl angibt z.B. kann man sich die internen Zählerstände über …`/ZaehlerA/z` oder …`/ZaehlerB/z` anzeigen lassen.

6. Strukturen (Records), Arrays, Schleifen, Bibliotheken

Beispiel eines Arrays:

```
TYPE Ttabelle IS ARRAY (0 TO 1) (0 TO 2) OF std_logic;
CONSTANT myTable : Ttabelle :=  (    ( '0', '0', '1'),
                                     ( '0', '1', '1')
                                );
```

Im Beispiel wird mit der Typdefinition **TYPE** ein Array mit zwei Zeilen und drei Spalten als Datentyp deklariert, dessen Elemente vom Typ `std_logic` sind. Danach wird eine Konstante `myTable` dieses Typs definiert und initialisiert. Der Zugriff auf Array-Elemente, z.B. `myTable(1,2),` erfolgt mit den entsprechenden Indices, im Unterschied zu C stehen die Indices aber in runden `(…)` statt eckigen `[…]`. Die Arrayelemente selbst dürfen einen beliebigen Typ haben, können also selbst auch Arrays wie z.B. `std_logic_vector` sein.

Mit Arrays lassen sich ebenfalls Funktionstabellen darstellen:
Quartus-Projekt Einführung.qpf / Decoder3.vhd

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

ENTITY Decoder3 IS
  PORT (  D : IN STD_LOGIC_VECTOR(3 downto 0);
          L : OUT STD_LOGIC_VECTOR(6 downto 0)
       );
END Decoder3;

ARCHITECTURE logic OF Decoder3 IS
  TYPE funktionstabelle IS
     ARRAY (0 TO 15) OF std_logic_vector(6 DOWNTO 0);

  CONSTANT tabelle : funktionstabelle :=
     (    "0111111", -- Anzeige 0
          "0000110", -- Anzeige 1
          . . .
          "1101111", -- Anzeige 9
          "-------", -- beliebige Anzeige fuer 10
          . . .
          "-------", -- . . . bis 15
     );
BEGIN
     L <= tabelle(to_integer(unsigned(D)));
END logic;
```

Das Eingangssignal `D` wird in einen Integerwert umgewandelt und als Index für das Array `tabelle` verwendet, das die Funktionstabelle enthält. Die einzelnen Arrayelemente enthalten die zugehörigen Ausgangssignale `Y`.

Während Arrays nur Elemente ein und desselben Typs enthalten dürfen, lassen sich mit Strukturen, in VHDL **RECORD** genannt, auch Elemente unterschiedlicher Typen zusammenfassen.

Beispiel eines Records:

```
TYPE Tstruktur IS RECORD
     befehlscode : STD_LOGIC_VECTOR( 3 downto 0);
     operand     : STD_LOGIC_VECTOR( 7 downto 0);
END RECORD Tstruktur;

CONSTANT myRecord: Tstruktur :=   ( "1100", "11001101" );
```

Im Beispiel wird zunächst ein Typ `Tstruktur` für den Record deklariert und dann eine Konstante `myRecord` dieses Typs definiert und initialisiert. Der Zugriff ein Element des Records erfolgt z.B. mit `myRecord.operand`.

For -Schleifen haben folgende Syntax:

```
FOR zählvariable := anfangswert TO endwert LOOP
    anweisungen
END LOOP;
```

Die Zählvariable muss nicht ausdrücklich als Variable deklariert werden. Statt `TO` wird beim Abwärtszählen `DOWNTO` verwendet. Obwohl VHDL als Programmiersprache für den Anfangs- und Endwert theoretisch variable Größen zulässt, ist die Schaltungssynthese nur dann sinnvoll möglich, wenn beide Werte konstant sind, da die Schaltung im Betrieb natürlich nicht veränderbar ist.

Beispiel für Bibliotheken:
Häufig verwendete Typdeklarationen usw. werden in Bibliotheken zusammengefaßt. Die Bibliothek steht üblicherweise in einer einzelnen Datei. Der Aufbau einer Bibliothek ist folgendermassen:

```
PACKAGE packageName IS
     Deklaration von Typen, Funktionen etc.
END;

PACKAGE BODY packageName IS
     Definition von Funktionen, Prozeduren etc.
END packageName;
```

Soll eine Bibliothek verwendet werden, so muss sie deklariert werden:

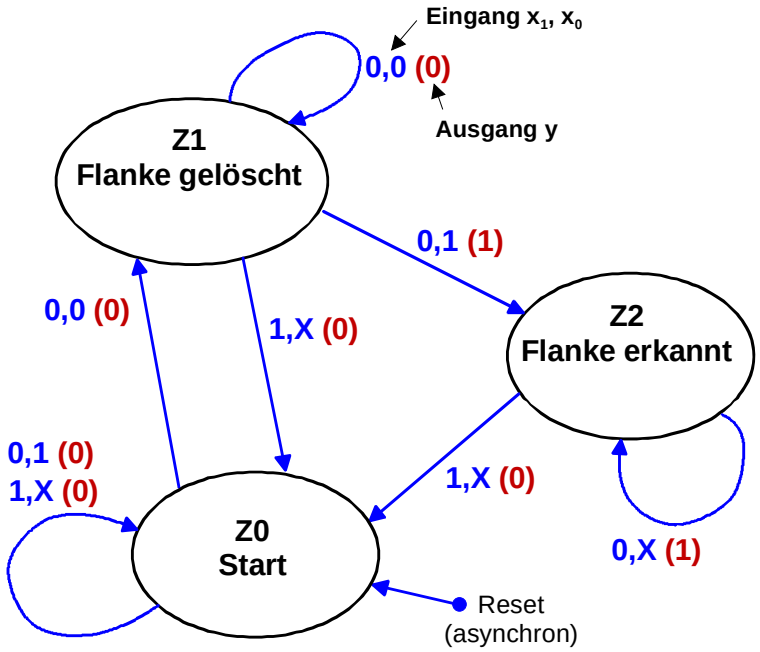
```
LIBRARY bibliotheksName;
USE bibliotheksName.packageName.ALL;
```

Achtung:
Eine Bibliotheksdeklaration bezieht sich nur auf die Entity, vor der sie direkt steht, d.h. für jede Entity muss die Deklaration wiederholt werden. Dies gilt auch für die IEEE-Bibliotheken

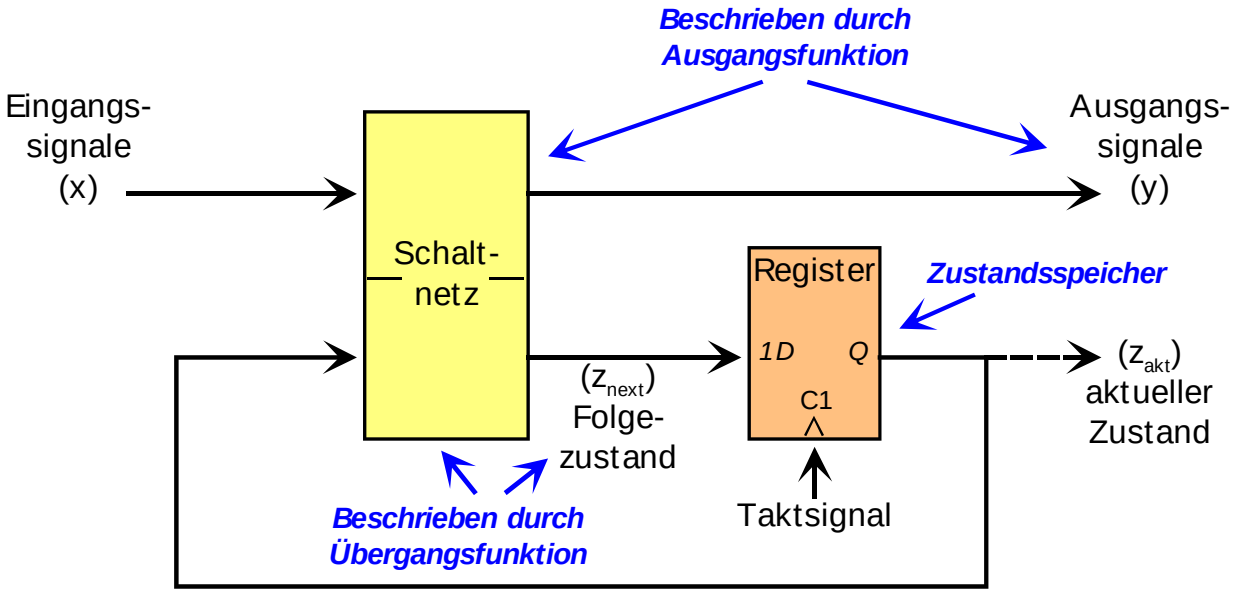
```
LIBRARY ieee;
     USE ieee.std_logic_1164.all;
     USE ieee.numeric_bit.all;
```

7. VHDL-Syntax für Zustandsautomaten (Finite State Machine FSM)

Beispiel:



Der Beschreibung in VHDL liegt folgende Schaltungsstruktur zugrunde:



Die Übergangs- und die Ausgangsfunktion werden in einem Prozess mit zwei CASE-WHEN-Konstrukten beschrieben. Dieser Prozess reagiert auf die Eingangssignale (X) und den aktuellen Zustand (z_{akt}).

Der Zustandsspeicher, d.h. das Register wird durch einen zweiten Prozess beschrieben, der auf das Taktsignal und gegebenenfalls auf asynchrone Lade- oder Resetsignale reagiert.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY FSM_Edgedetect IS
  PORT (  FSMClk : IN STD_LOGIC;
          X      : IN STD_LOGIC_VECTOR(1 downto 0);
          Reset  : IN STD_LOGIC;
          Y      : OUT STD_LOGIC
       );
END FSM_Edgedetect;

ARCHITECTURE logic OF FSM_Edgedetect IS
     TYPE State is (Z0, Z1, Z2); -- Enumeration für die Zustände
     SIGNAL Zakt:  State; -- Aktueller Zustand
     SIGNAL Znext: State; -- Folgezustand
BEGIN
Comp: PROCESS (X, Zakt) IS -- Prozess für Übergangs- und Ausgangsfunk-
tion
  BEGIN
  -- Übergangsfunktion
     Znext <= Zakt;
     CASE Zakt IS   -- Zustandsübergänge (Transistions)
       WHEN Z0 =>   IF X = "00" THEN
                      Znext <= Z1;
                    END IF;
       WHEN Z1 =>   IF X(1) = '1' THEN
                      Znext <= Z0;
                    ELSIF X = "01" THEN
                      Znext <= Z2;
                    END IF;
       WHEN Z2 =>   IF X(1)= '1' THEN
                      Znext <= Z0;
                    END IF;
     END CASE;
  -- Ausgangsfunktion (Activity, Action)
     CASE Zakt IS   -- Ausgangssignale
       WHEN Z0 =>   Y <= '0';       -- Moore-Ausgangssignalverhalten
       WHEN Z1 =>   IF X(1) = '1' THEN
                      Y <= '0';     -- Mealy Ausgangssignalverhalten
                    ELSE
                      Y <= X(0);
                    END IF;
       WHEN Z2 =>   IF X(1) = '1' THEN
                      Y <= '0';
                    ELSE
                      Y <= '1';
                    END IF;
     END CASE;
  END PROCESS Comp;
```

```
Trigger: PROCESS (Reset, FSMClk) IS  -- Prozess für den Zustandsspeicher
  BEGIN
    IF (Reset = '1') THEN
      Zakt <= Z0;              -- Asynchrones Rücksetzen
    ELSIF rising_edge(FSMClk) THEN
      Zakt <= Znext;           -- Übergang in den Folgezustand
    END IF;
  END PROCESS Trigger;
END logic;
```

Für die Zustände wird eine Enumeration als neuer Datentyp eingeführt:

```
TYPE zustandsTypName IS (zustand1, zustand2, ...);
```

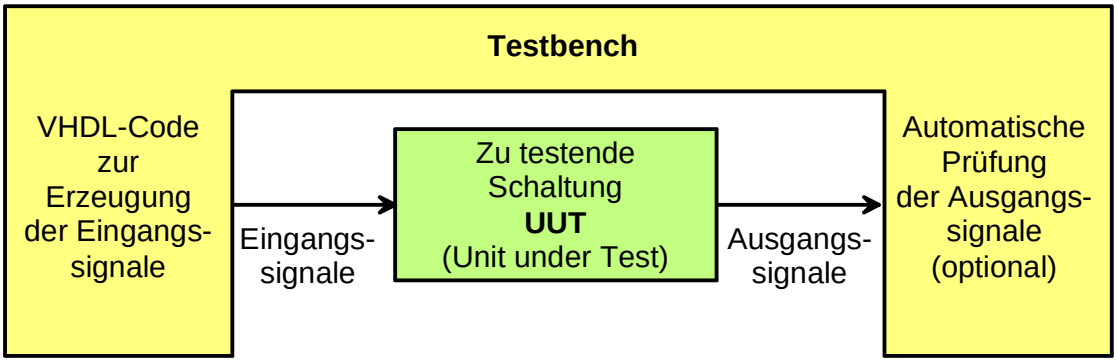
Die Bezeichnungen des Typs und der Zustände können beliebig gewählt werden. Für den aktuellen Zustand (z_{akt}) und den Folgezustand (z_{next}) werden interne Signale dieses Typs definiert:

```
SIGNAL aktuellerZustand, folgeZustand : zustandsTypName;
```

Die Zustandscodierung wird automatisch festgelegt, typischerweise 1-aus-n (1-hot) oder dual.
Die Unterscheidung zwischen Moore- und Mealy-Automat besteht lediglich darin, ob die Ausgangssignale in der Ausgangsfunktion zusätzlich von den Eingangssignalen abhängig gemacht werden (Mealy) oder nicht (Moore).

8. Erstellung einer Testbench für die Simulation

Um eine Schaltung durch Simulation zu testen, muss man zusätzlich zur Beschreibung der Funktion der Schaltung auch die Eingangssignale bereitstellen. Dies erfolgt durch eine sogenannte Testbench. Wenn man die Ausgangssignale der Schaltung nicht nur im Zeitdiagramm des Simulators manuell überprüfen will, kann man in die Testbench zusätzlich Funktionen aufnehmen, die die Ausgangssignale mit vorgegebenen oder in der Testbench berechneten Werten vergleichen und bei Abweichungen eine Meldung erzeugen (*Assert*).



Im Gegensatz zu einer üblichen Schaltung hat die Testumgebung keine externen Ein- und Ausgangssignale, sondern bettet die zu testende Schaltung ein. Die Entity-Deklaration hat daher keine Port-Signale. In Testbenches gibt es zusätzlich zu den bereits eingeführten Prozessen mit Sensitivity-Listen auch prozesse ohne diese Sensitivi-

ty-Listen. Lesen Sie zur Beschreibung dieser Prozesse bitte in der Laboranleitung CA1 zum Versuch Siebensegment-Anzeige nach.

Beispiel: Erzeugung eines einmaligen Signals, z.B. Reset zur Initialisierung und eines periodischen Taktsignals.

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

ENTITY testbench IS    -- Keine Schnittstellensignale der Entity
END testbench;

ARCHITECTURE tb_arch OF testbench IS
     SIGNAL reset : STD_LOGIC;     -- Definition der Signale
     SIGNAL clk : STD_LOGIC;

     COMPONENT testschaltung       -- Deklaration der zu testenden
          PORT ( takt : IN STD_LOGIC;                Schaltung
                 R    : IN STD_LOGIC;
                 . . .
               );
     END COMPONENT;
BEGIN

instanz1: testschaltung       -- Verdrahtung der Testbench mit der
     PORT MAP ( takt => clk, R => reset, ...);    -- Schaltung

     init: PROCESS                 -- Einmaliges Reset-Signal
     BEGIN
          reset <= '0';            -- Reset-Signal '0' für 10ns
          WAIT FOR 10ns;
          reset <= '1';            -- Reset-Signal '1' für 30ns
          WAIT FOR 30ns;
          reset <= '0';            -- Reset-Signal '0'
          WAIT;                    -- Warte für immer
     END PROCESS init;

     clkgen: PROCESS               -- Periodisches Taktsignal
     BEGIN
          clk <= '0';              -- Takt-Signal '0' für 25ns
          WAIT FOR 25ns;
          clk <= '1';              -- Takt-Signal '1' für 25ns
          WAIT FOR 25ns;
     END PROCESS clkgen;           -- Takt periodisch wiederholen

END tb_arch;
```

Achtung:

- **WAIT FOR … ist nur für die Simulation geeignet. Es ist NICHT möglich, damit in einer realen Hardwareschaltung eine definierte Zeitverzögerung zu erzeugen.**

Die optionale automatische Überprüfung von Ausgangssignalen der zu testenden Schaltung kann in der Testbench folgendermaßen erfolgen:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.NUMERIC_STD.all;     -- für die Funktion to_integer()

ENTITY testbench IS
END testbench;

ARCHITECTURE tb_arch OF testbench IS
     . . .
BEGIN
     checker: PROCESS(signalvector1, signal2)
          VARIABLE i : integer;
     BEGIN
          -- Wandelt den Vector in einen Integerwert
          i := to_integer(signalvector);

          -- Erzeugt eine Warnung, wenn i=2 ist, und dabei nicht
          -- gleichzeitig signal2 = '1' ist.
          IF (i = 2) THEN
               ASSERT NOT signal2 = '1';
               REPORT "Achtung Fehler: signal2=1 erwartet"
               SEVERITY WARNING;
          END IF;
     END PROCESS checker;
END testbench;
```

Literatur

[1] G. Lehmann; B. Wunder; M. Selz: Schaltungsdesign mit VHDL. Franzis Verlag, 1994.
http://www.itiv.kit.edu/653.php

[2] F. Schubert: VHDL-Syntax. Kurzreferenz. 2008.
http://users.etech.haw-hamburg.de/users/schubert/vorles.html

[3] P. Ashenden: The Designer's Guide to VHDL. Morgan Kaufmann Publishers, 3. Auflage, 2008.

[4] P. Ashenden: The VHDL Cookbook. 1990.
Veraltet, berücksichtigt das IEEE.std_logic Paket noch nicht
http://tams-www.informatik.uni-hamburg.de/vhdl/doc/cookbook/VHDL-Cookbook.pdf

[5] A. Mäder: VHDL Kompakt. Uni Hamburg, 2011
http://tams-www.informatik.uni-hamburg.de/research/vlsi/vhdl/doc/ajmMaterial/vhdl.pdf

Ausführliche Literaturlisten unter
http://www.vhdl.org/comp.lang.vhdl
und http://tams-www.informatik.uni-hamburg.de/research/vlsi/vhdl/index.php