VHDL Grundlagen

Proseminar FPGAs
SS2003

Markus Holisch

Inhalt

Quellen: A.Sikora ‚Programmierbare Logikbauelemente – Architekturen und Anwendungen' C. Hanser Verlag, 2001

M. Wannemacher ‚Das FPGA-Kochbuch (mit CD-ROM)' Thomson Publishing, 1998

Link zum Autor der Beispielschaltung: http://wwwbode.cs.tum.edu/~jeitner/

1. Grundlagen

1.1 Was ist VHDL?

VHDL ist eine Hardwarebeschreibungssprache, die im Auftrag der US-Regierung anfangs der 80er Jahre entwickelt und im Jahre 1987 als IEEE 1076-87 standardisiert wurde. Eine leicht überarbeitete Version wurde später 1993 als IEEE 1076-93 genormt.

Das Kürzel VHDL setzt sich zusammen aus: V (steht für VHSIC, was ***V****ery* ***H****igh* ***S****peed* ***I****ntegrated* ***C****ircuit* bedeutet) und HDL (=***H****ardware* ***D****escription* ***L****anguage*).

Die Spezifikation von VHDL ist im Language Reference Manual (LRM) niederlegt. Es ist eine komplizierte und mächtige Sprache, wird daher von Spöttern als *„**V**ery **H**ard to **D**eal with **L**anguage“* gedeutet. Dies mag zutreffen, wen man in die tiefen Details von VHDL, wie z.B.: Erstellung von Synthesealgorithmen, eintauchen will. Ein weiteres Problem rührt daher, dass VHDL nicht vollständig beschrieben wurde und im LRM nur die Simulationssemantik niederlegt worden ist.

Es ist nur eine HDL unter vielen. Es existiert eine Menge einfacher aufgebauter HDLs, die vorwiegend für den Entwurf von PLD eingesetzt werde. Zu nennen wären u.A.: PALASM, ABEL, ALTERA-HDL, CUPL, LOG/iC.

In den 70er und 80er Jahren wurden außer VHDL weitere komplexe HDL entwickelt, die sich mit Ausnahme von Verilog, welches vorwiegend in den USA eingesetzt wird, nicht durchsetzen konnten. In Europa verwendet man heutzutage etwa zu 60 % VHDL und zu 40 % Verilog.

1.2 Anwendungen

Man kann VHDL für die wesentlichen drei Arbeitsschritte beim System- und Schaltungsentwurf einsetzen:

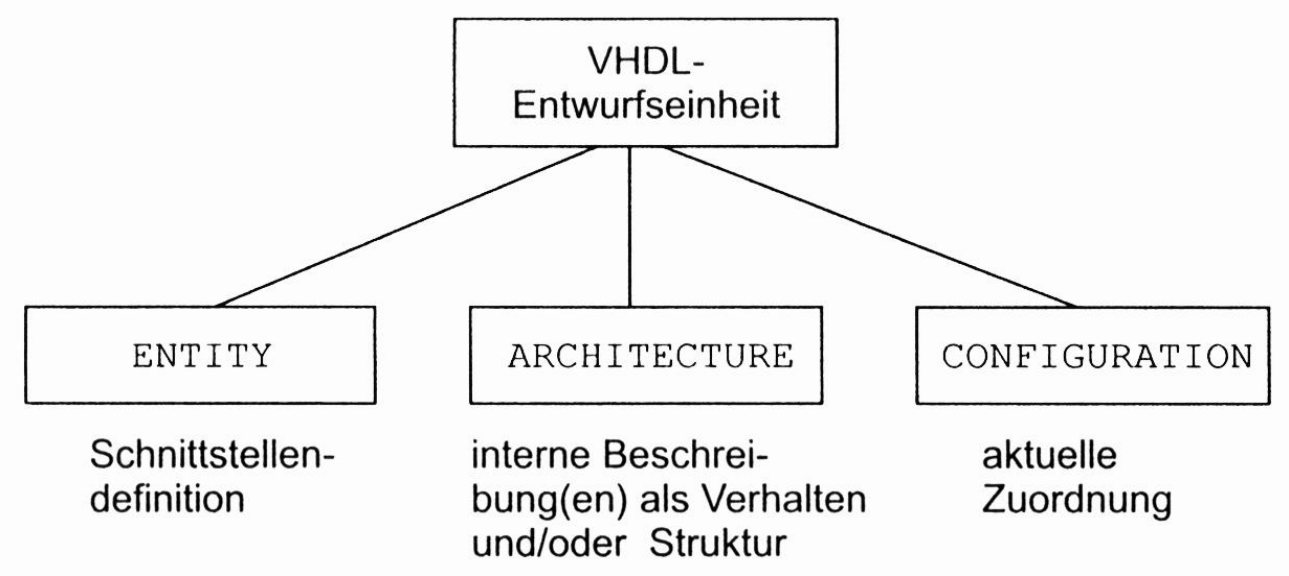
Anwendung	**Verwendung**	**Adressat**
Dokumentation	Spezifikation	Menschlicher Leser
Simulation	Funktionale Beschreibung auf Systemebene	Simulator
Synthese	Beschreibung auf RTL-Ebene	Synthesewerkzeug

Mit Hilfe von VHDL können digitale Schaltungen auf allen drei Abstraktionsebenen beschrieben werden, d.h. von der funktionalen Verhaltensbeschreibung auf der höchsten Abstraktionsebene über die RTL-Beschreibung (RTL = Register Transfer Level) bis hin zur Beschreibung der Schaltung auf der Gatterebene.

Darüber hinaus ist mit VHDL die Simulation anderer (z.B.: nicht-mikroelektronischer) Systeme möglich. Insbesondere die Erweiterung auf analoge Teilsysteme ermöglicht die Simulation gemischter digital-analoger Systeme in einheitlicher Umgebung.

1.3 Aufbau

Eine Hardware-Schaltung besteht aus Funktionseinheiten (z.B.: Multiplexer, Flip-Flops...) welche durch Signale miteinander verbunden sind. In VHDL werden solche Funktionseinheiten durch Entwurfseinheiten *(Entities)* repräsentiert, welche wiederum aus weiteren Einheiten zusammengesetzt sein können. Jeder Entwurfseinheit setzt sich aus drei Teilbereichen zusammen:

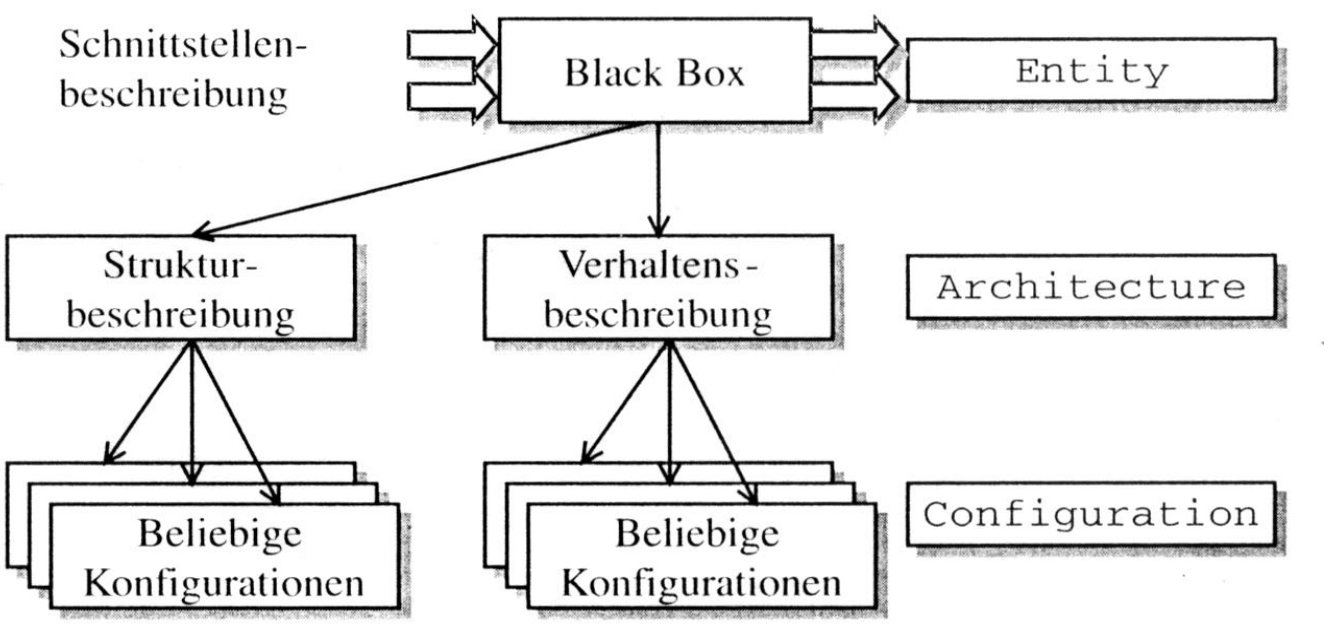


Es hier zu beachten, dass das Wort *„Entity"* sowohl als ein Schlüsselwort zur Schnittstellendefinition, als auch als der englische Begriff der Entwurfseinheit verwendet wird. Eine Entwurfseinheit kann auch mehrere Architekturbeschreibungen besitzen. In einem solchen Fall ist eine Konfigurationseinheit notwendig.

Durch die Trennung zwischen der Schnittstellendefinition (*Entity*) und der Systembeschreibung (*Architecture*) auf der jeweiligen Abstraktionsebene entspricht die Bauweise einer VHDL-Entwurfseinheit dem Black-Box-Prinzip (Kapselung). Somit können bestimmte Teilkomponenten eines Systems gefahrlos von Drittzulieferern realisiert werden, wenn die Schnittstellendefinition unverschlüsselt und die Architektur verschlüsselt übergeben wird.

Ein weiterer Vorteil ist, dass mit zunehmenden Implementierungsdetails mehrere Architekturen auf unterschiedlichen Abstraktionsgraden entwickelt werden können, die wahlweise in die Entwurfseinheit eingebunden werden können, solange die Schnittstellendefinitionen identisch sind.

Es ergibt sich somit folgende Hierarchie der Entwurfsobjekte:



Darüber hinaus gibt es in VHDL sogenannte *Packages*, die den Bibliotheken (*Libraries*) der prozeduralen Programmiersprachen entsprechen. Solche *Packages* enthalten Typen, Konstanten, Komponenten, oder oft gebrauchte Funktionen und Prozeduren. Vom besonderen Interesse sind die durch den IEEE-Standard vordefinierten Bibliotheken, wie zum Beispiel std_logic_1164 welches ein 9-wertiges Logiksystem beinhaltet.

2. Schaltungsbeschreibungen

2.1 Verhaltens-/Strukturmodelle

In VHDL kann eine Schaltungsbeschreibung auf zweierlei Arten erfolgen, nämlich als ein Verhaltens- oder als ein Strukturmodell

Verhaltensmodell:

Hier werden keine weiteren Komponenten instanziiert, sondern das Verhalten der Einheit beschrieben. Es gibt an, wie sich die Ausgangssignale der Entwurfseinheit aufgrund der Zustände der Eingangssignale verhalten

Strukturmodell:

Es ist ein Modell einer Einheit, deren Struktur durch Instanziierung von Komponenten und deren Verbindungen beschrieben wird. Die Darlegung der Eigenschaften der Unterkomponenten erfolgt in unabhängigen VHDL-Modellen, welche auch kompiliert in Packages zur Verfügung gestellt werden können.

Jede benutzte Unterkomponente muß in der Architekturbeschreibung mit ihrem Schnittstellenverhalten aufgeführt werden bevor sie mir eindeutigem Namen instanziiert werden darf. Dies geschieht über das Schlüsselwort *Component*. Dabei ist die Reihenfolge des Aufrufs zu beachten, d.h. die Komponenten müssen zuerst bekannt sein

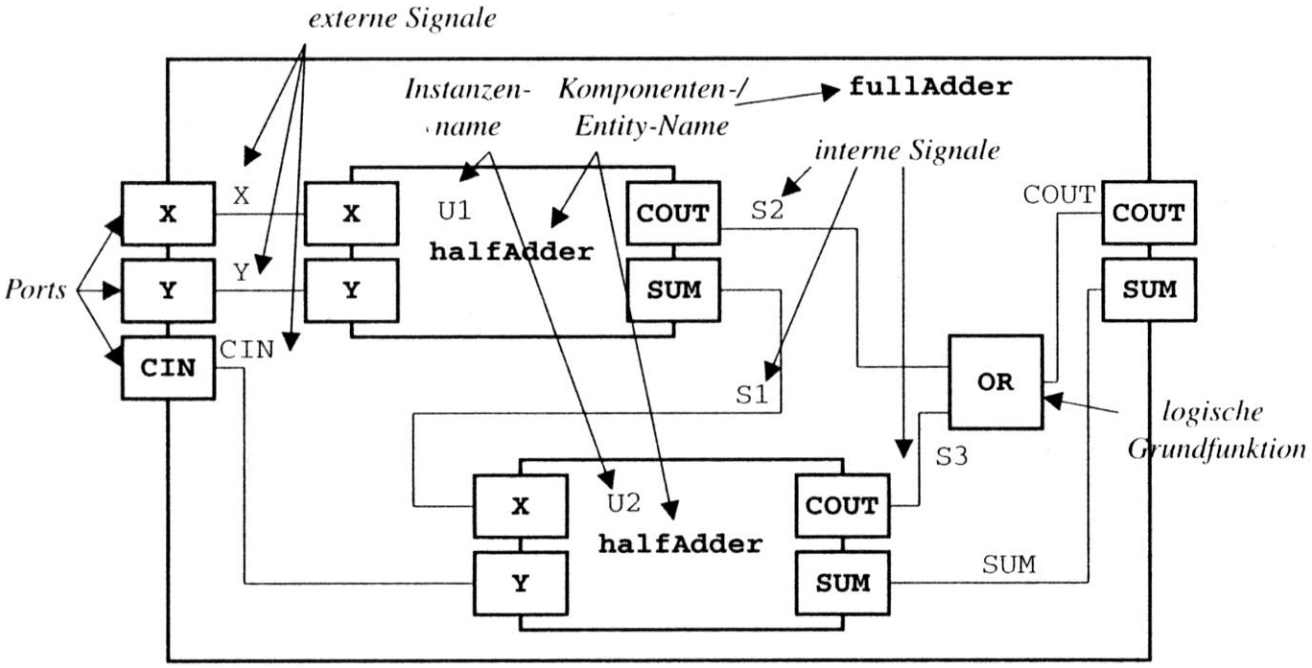
Beispiel für ein Verhaltensmodell:

```
ENTITY halfAdder IS
   PORT  (
      X, Y      : IN  Bit;
      SUM, COUT : OUT Bit);
END halfAdder;

ARCHITECTURE behavioural OF halfAdder IS
BEGIN
   SUM   <= X XOR Y ;
   COUT  <= X AND Y ;
END behavioural ;
```

Beispiel für ein Strukturmodell :

```
ENTITY fullAdder IS
   PORT  (
      X, Y, CIN : IN  Bit;
      SUM, COUT : OUT Bit);
END fullAdder;

ARCHITECTURE structural OF fullAdder IS
SIGNAL S1, S2, S3;
COMPONENT halfAdder
   PORT  (
               X, Y     : IN  Bit;
         SUM, COUT : OUT Bit);
END COMPONENT;
BEGIN
   U1 : halfAdder   PORT MAP (X, Y, S1, S2);
   U2 : halfAdder   PORT MAP (S1, CIN, SUM, S3);
   COUT <= S2 OR S3;
END structural;
```

Auf der Abbildung sieht man den Schaltplan des in den Beispielen erwähnten Volladdierers mit Beschreibungen der einzelnen Teilelemente:



2.2 Schnittstellen

Die Schnittstelle einer Einheit wird durch eine so genannte *Port Clause* deklariert. Deren Anschlüsse heißen *Ports* und stellen die Kanäle zur Kommunikation zwischen einer *Entity* und ihrer Umgebung.

Die *Ports* besitzen:

- eindeutigen Namen
- einen Modus, der die Signalrichtung angibt
- einen Signaltyp *(siehe Abschnitt 5)*

Für die Zuordnung der *Ports* zu den *Signalen (Port Map)* gibt es folgende zwei Möglichkeiten:

Positionszuordnung
(connection by position)

```
Instanzname: Komponentenname
PORT MAP(Signal_1,
         Signal_2,
         ....
         Signal_n);
```

Namenszuordnung
(connection by name)

```
Instanzname: Komponentenname
PORT MAP(Port_A => Signal_1,
         Port_B => Signal_2,
         ....
         Port_X => Signal_n);
```

2.3 Signale

Es gibt in VHDL zwei Sorten von Signalen:

Externe Signale:

Sie verbinden eine Einheit mit anderen Elementen im System. Externe Signale können durch ein *Port*-Statement definiert werden und stehen dann auch innerhalb der Einheit zur Verfügung

Interne Signale:

Sie werden durch das Schlüsselwort *Signal* in der Architekturbeschreibung definiert und sind somit nur innerhalb einer Einheit sichtbar. Die internen Signale dienen der Verbindung der Prozesse (siehe Abschnitt 4.1) und Komponenten innerhalb einer Einheit. Sie benötigen keinen Modus und keine Richtungsangabe.

Während also die Entwurfseinheiten *(Entities)* den in der Schaltung verwendeten Funktionseinheiten bzw. Bauelementen entsprechen, symbolisieren die Signale die verwendeten Verbindungsleitungen. Daher wäre es nicht sinnvoll, diesen unmittelbar einen Zustand zuzuweisen.

3. Zeit

Um eine Schaltung im zeitlichen Ablauf modellieren zu können, müssen auch entsprechende Konstrukte in der Sprache verfügbar sein.

Dabei sind u.a. Verzögerungszeiten zu beachten, die durch Gatter und Verbindungsleitungen entstehen. Es müssen auch zeitliche Bedingungen eingehalten werden, die notwendig für eine korrekte und zuverlässige Funktionalität sind, z.B.: Setup- und Holdzeiten von Registern. Ferner müßten schließlich Wartezeiten berücksichtigt werden, die z.B. bei einem Oszillator entstehen.

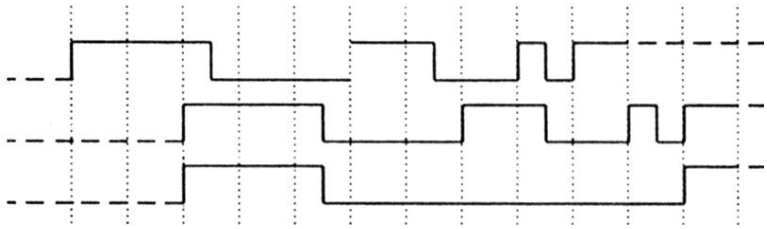
Die Verzögerungszeit kann auf der Gatterebene auf verschiedene Arten modelliert werden, dabei werden jegliche Angaben in Nanosekunden ausgedrückt:

- Die *transport*-Anweisung verzögert alle Signalwechsel um die angegebene Zeit

- Die *inertial*-Anweisung verzögert die Signalwechsel um die angegebene Zeit und unterdrückt zusätzlich alle Impulse, die kürzer sind als das Angegebene Intervall

Diese Möglichkeiten sind im folgenden Beispiel dargestellt::

```
sigA
sigB <=    TRANSPORT sigA AFTER 2ns;
sigC <=    INERTIAL  sigA AFTER 2ns;
```



Es können auch andersartige Verzögerungen mit Hilfe der *wait*-Anweisung beschrieben werden:

Anweisung	**Bedeutung**
`WAIT FOR 25ns;`	verzögert den weiteren Ablauf immer um 25 ns
`WAIT UNTIL sigA = '1';`	verzögert den weiteren Ablauf, bis *sigA* den Zustand '1' angenommen hat
`WAIT ON sigA;`	verzögert den Ablauf bis ein Zustandswechsel am *sigA* stattgefunden hat

zu Beachten: absolute Zeitangaben haben keine Entsprechung in der Hardware, da es keine Bausteine zu deren Realisierung gibt.

4. Nebenläufigkeit

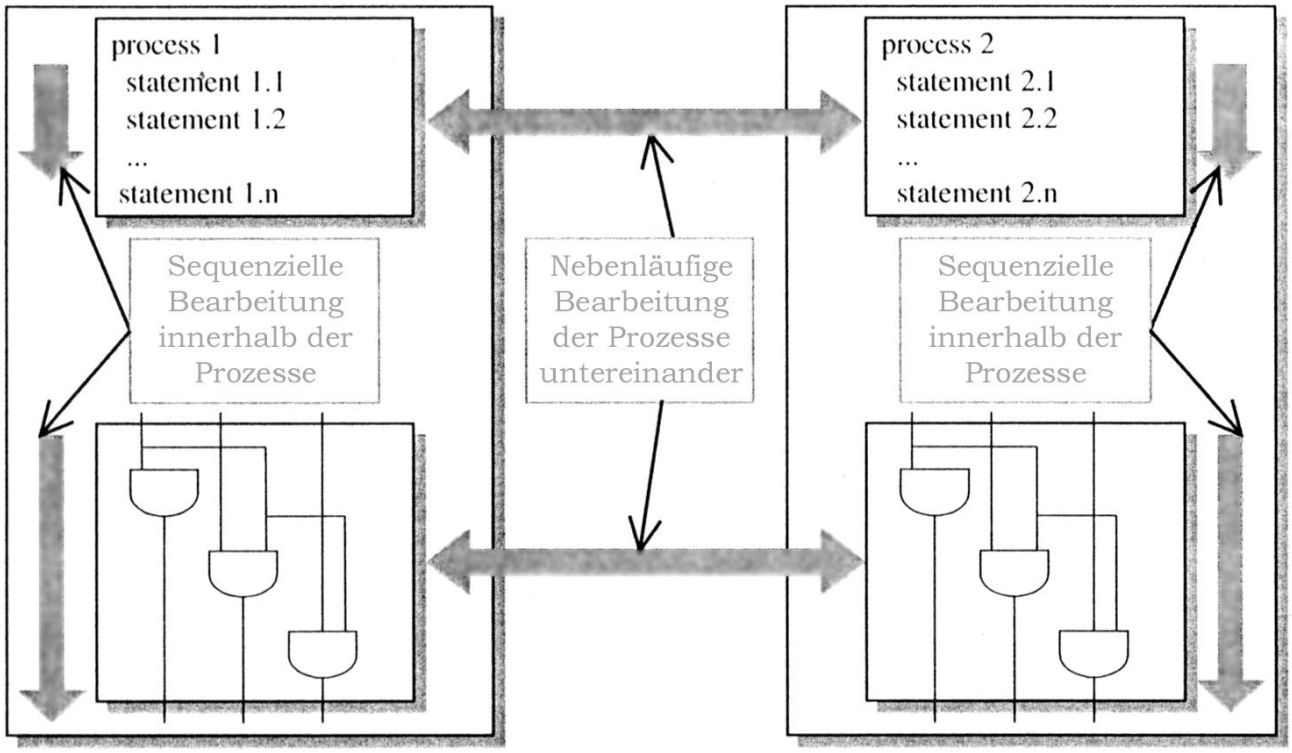
4.1 Prozesse

Wie bereits angedeutet besteht ein vollständiges Design in VHDL aus miteinander verbundenen *Entities*, welche über Signale kommunizieren. Da die *Entities* verschiedenen Bauelementen entsprechen ist es leicht nachvollziehbar, dass sie nebenläufig aktiv sind.

Innerhalb der *Entities* sind Prozesse die Basiselemente. Als Prozesse versteht man unter anderem: *Process*-Anweisungen, Signalzuweisungen mit „<=“, Prozeduraufrufe oder Komponenteninstanziierungen

D.h. die *Process*-Anweisung ist nur eine Variante eines VHDL-Prozesses.

Die zeitliche Abarbeitung der Prozesse:



Während die Prozesse also untereinander nebenläufig aktiv sind, werden die Statements innerhalb von jedem Prozess sequentiell abgearbeitet.

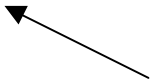
Dies geschieht, weil ein VHDL-Prozess das zeitlich sequenzielle Verhalten eines Funktionselements beschreibt und die Funktionselemente in einer Schaltung parallel angesteuert werden

4.2 Prozessbearbeitung/Signalzuweisungen

Es werden nur die aktiven Prozesse bearbeitet. Ob ein Prozess aktiv ist oder nur schlafend entscheidet die *Sensitivity*-Liste, die man zu jedem Prozess angeben kann. Darin werden alle für den jeweiligen Prozess relevanten Signal aufgeführt. Ändert sich der Zustand mindestens eines der Signale aus der *Sensitivity*-Liste, wird der Prozess geweckt und abgearbeitet.

z.B.:

```
abc:  PROCESS (S1, S2, S2)
   BEGIN
   .
   .
   .
   END abc;
```



Sensitivity-Liste

Um widersprüchliche Beschreibungen zu vermeiden, darf ein Prozess mit einer Sensitivity-Liste keine *wait on*-Anweisungen beinhalten.

Eine alternative Modellierung kann ohne die *Sensitivity*-Liste aber mit mindestens einer *wait on*-Anweisung erfolgen.

z.B.:

```
abc:  PROCESS
BEGIN
.
.
.
   WAIT ON S1, S2, S3;
END abc;
```

Es ist zu beachten, daß die Zuweisung von Signalen erst nach der vollständigen Bearbeitung aller aktiven Prozesse stattfindet. Dadurch wird erreicht, daß das Ergebnis der Simulation unabhängig ist von der Reihenfolge der Prozesse im Quelltext bzw. der Reihenfolge, in der die Prozesse simuliert werden. Es ist auch sehr wichtig alle relevanten Signale in die *Sensitivity*-Liste aufzunehmen.

5. Variable, Typen

Neben Signalen existieren in VHDL Variablen. Diese haben keine unmittelbare Entsprechung in der Hardware, können aber mit Signalen des gleichen Typs verbunden werden.

Die wichtigsten Eigenschaften und Unterschiede zwischen Signalen und Variablen:

Signale	**Variable**
werden durch Ports in Entities oder im Deklarationsteil einer Architekturbeschreibung mit „`SIGNAL`“ deklariert	besitzen lokalen Charakter und können in Prozessen und Unterprogrammen deklariert werden
nehmen ihren Zustand an, wenn der Prozess beendet ist	nehmen unmittelbar nach der Zuweisung ihren Zustand an
Zuweisungsoperator „ <= “	Zuweisungsoperator „ := “
können mit einer Zeitverzögerung beaufschlagt werden	keine Zeitverzögerung möglich
3 Eigenschaften: Typ, Wert, Zeit	2 Eigenschaften: Typ, Wert

Die Variablen entsprechend weitgehend den Variablen aus den bisher gekannten Programmiersprachen. Ein wichtiger Unterschied besteht jedoch darin, daß die Variablen in VHDL nach Durchlauf eines Prozesses nicht gelöscht werden, da die Prozesse ja auch nicht gelöscht werden, sondern in einen „schlafenden“ Zustand übergehen.

VHDL ist eine streng typorientierte Sprache. Folgende Grundtypen stehen zur Verfügung:

Typ	**Beschreibung**
`Scalar`	keine Unterelemente
`Composite`	Unterelemente. Aufbau von Arrays möglich
`Access`	erlaubt Zugriff auf Objekte
`Files`	erlaubt Zugriff auf Daten

Der im Volladdierer-Beispiel aufgetretene Typ *Bit* ist Untertyp der Menge *Scalar*.

Darüber hinaus können auch gebündelte Signale *(Busse)* definiert werden, z.B.:

Bei absteigender Signalreihenfolge wird in der Klammer statt „*TO*“ das Wort „*DOWNTO*“ benutzt.

Ein weit verbreiterter Satz von logischen Typen ist das bereits erwähnte *Package* std_logic_1164. Es wird dort neben drei Signalstärken „*stark*“, „*schwach*“ und „*hochohmig*“ auch zwischen drei Logikpegeln, „*0*“, „*1*“ sowie „*unbestimmt*“, unterschieden. Somit ergeben sich in der Kombination 9 mögliche logische Zustände.

6. Beispielschaltung: 7-Segmentanzeige

Dieses Beispiel beschreibt eine Schaltung, die eine 7-Segmentanzeige ansteuert. Es sind darüber hinaus zwei Modi für die Schaltung vorgesehen. Im Normalmodus wird ein systeminterner Takt zum fortwährenden Hochzählen der Anzeige verwendet. Im Benutzermodus hingegen, kann die Anzeige durch Drücken eines Schalters (und nur dadurch) um eins erhöht werden.

Deine LED dient zur Indikation welcher Modus gerade aktiv ist. Im Benutzermodus soll sie leuchten.

```
--------------------------------------------------
-- Beispielschaltung fuer das TGI-Praktikum VHDL --
-- Autor:   Juergen Jeitner                      --
-- Datum:   21.05.2002                           --
--------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity BeispielSchaltung is
    port (
        Clock       : in STD_LOGIC;         -- Systemtakt
        Reset       : in STD_LOGIC;     -- Asynchroner Reset
        CountClock  : in STD_LOGIC;     -- Zusaetzlicher Zaehlertakt
        SetDisplay  : in STD_LOGIC;     -- Schaltet in Benutzer- bzw. Normalmodus
        ButtonPressed  : in STD_LOGIC;     -- Benutzergeseteuerte Zaehlung
        ToggleDot      : in STD_LOGIC;     -- Ansteuerung des Punktes der Anzeige
        Dot         : buffer STD_LOGIC;     -- Punkt der Digitalanzeige
        Digit       : out STD_LOGIC_VECTOR(0 to 6); -- 7 Segmente der Anzeige
        ControlLED  : out STD_LOGIC-- Zeigt an, ob der Benutzermodus aktiv
    );
end BeispielSchaltung;
```

```
-- Architekturbeschreibung zu obiger Beispielschaltung

architecture BeispielSchaltung_arch of BeispielSchaltung is

-- interne Signale fuer den Prozess "system_mode"
signal set_display_pressed  : STD_LOGIC;
signal user_set_mode        : STD_LOGIC;

-- interne Signale fuer den Prozess "user_button_control"
signal button_pressed       : STD_LOGIC;
signal count                : STD_LOGIC;

-- internes Signal fuer den Prozess "system_control"
signal counter              : STD_LOGIC_VECTOR(3 downto 0);

-- Start der Architekturbeschreibung
begin

   -- Wechsel von Normalbetrieb in den benutzergesteuerten Modus.

   system_mode: process (Clock, Reset, SetDisplay, set_display_pressed)
   begin
      if (Reset = '1') then                    -- Ein asynchroner Reset ist aufgetreten, dann
         set_display_pressed <= '0';           -- werden alle für diesen Prozess relevanten
         user_set_mode <= '0';                 -- Signale in den Initialzustand zurückgesetzt.
      elsif (Clock'event and Clock = '1') then    -- Ansonsten, bei jeder steigenden Taktflanke,
         if (SetDisplay = '1' and set_display_pressed = '0') then  -- falls Benutzer den
                                                                   -- Modus wechseln will
            set_display_pressed <= '1';
         end if;
         if (SetDisplay = '0' and set_display_pressed = '1') then  -- wird dies beim
            user_set_mode <= not user_set_mode;   -- Wiederloslassen des Schalters erledigt.
            set_display_pressed <= '0';
         end if;
      end if;
   end process system_mode;


   -- Überwacht die manuelle Zählung der Digitalanzeige

   user_button_control: process (Clock, Reset, ButtonPressed, button_pressed)
   begin
      if (Reset = '1') then
         count <= '0';
         button_pressed <= '0';
      elsif (Clock'event and Clock = '1') then
         count <= '0';
         if (ButtonPressed = '1' and button_pressed = '0') then
            button_pressed <= '1';
         end if;
         if (ButtonPressed = '0' and button_pressed = '1') then
            count <= '1';
            button_pressed <= '0';
         end if;
      end if;
   end process user_button_control;
```

```
   -- Sytemprozess zur Durchfuehrung der Zählererhöhung. Entweder manuell oder
   -- über internes "CountClock" Signal.

   system_control: process (Clock, Reset, ToggleDot, CountClock, Dot, count, counter)
   begin
      if (Reset = '1') then                              -- Bei Reset einen
         ControlLED <= '1';                              -- (sinnvollen,) selbstgewählten
         Dot <= '0';                                     -- Zustand herbeiführen,
         counter <= "0000";                              -- sonst bei
      elsif (Clock'event and Clock = '1') then           -- steigender Taktflanke
         ControlLED <= '1';                              -- LED deaktivieren.
         if (ToggleDot = '1') then                       -- Bei aktivem ToggleDot
            Dot <= not Dot;                              -- den Dot-Ausgang invertieren.
         end if;
         if (user_set_mode = '1') then                   -- Ist der Benutzermode aktiv
            ControlLED <= '0';                           -- LED aktivieren
            if (count = '1') then                        -- und benutzergesteuert
               if (counter >= "1001") then               -- entweder den Zähler wieder
               ´            counter <= "0000";           -- auf null setzen
               else                                      -- oder
                  counter <= counter + 1;                -- um eins erhöhen.
               end if;
            end if;
         elsif (CountClock = '1') then                   -- Im Normalmodus bei aktivem
            if (counter >= "1001") then                  -- internem Signal
               counter <= "0000";                        -- den Zähler entsprechend
            else                                         -- ansteuern.
               counter <= counter + 1;
            end if;
         end if;
      end if;
   end process system_control;


   -- Prozess zur Umcodierung des internen 4-Bit Zählers "counter" in eine 7-Segment-Anzeige

   digit_control: process (Clock, Reset, counter)
   begin
      if (Reset = '1') then
         Digit <= "0000000";
      elsif (Clock'event and Clock = '1') then
         case counter is
            when "0000" => Digit <= "1111110";
            when "0001" => Digit <= "0110000";
            when "0010" => Digit <= "1101101";
            when "0011" => Digit <= "1111001";
            when "0100" => Digit <= "0110011";
            when "0101" => Digit <= "1011011";
            when "0110" => Digit <= "1011111";
            when "0111" => Digit <= "1110000";
            when "1000" => Digit <= "1111111";
            when "1001" => Digit <= "1111011";
            when others =>
            end case;
         end if;
   end process digit_control;

end BeispielSchaltung_arch;
```