VHDL

Syntax

Höhere Technische
Bundes-Lehr- und
Versuchsanstalt
BULME Graz – Gösting

V1.0

F. Wolf

Graz, Februar 2002

Inhaltsverzeichnis

1 Einführung 3

2 Kurze Beschreibung der Design-Einheiten 4

2.1 Schnittstellenbeschreibung (Entity) 4

2.2 Architektur (Architecture) 5

2.3 Konfiguration (Configuration) 8

2.4 Package 9

2.5 Library 9

3 Elemente der Sprache VHDL 10

3.1 VHDL Objekte 11

3.1.1 Signale versus Variable 12

3.2 Signalzuweisungen (Signal Assignments) 13

3.2.1 Concurrent Signal Assignment 13

3.2.2 Interne Signale 14

3.2.3 Bedingte Signalzuweisung 15

3.3 Datentypen und Typdeklarationen 16

3.3.1 Vordefinierte Datentypen 16

3.3.2 Feldtypen 17

3.3.3 Benutzerdefinierte Datentypen 17

3.3.4 Aufzählungstypen 17

3.3.5 Physikalische Typen 18

3.4 Operatoren 19

3.5 Attribute 20

3.6 Interface-Listen 21

3.6.1 Ports 21

3.6.2 Verbindungs-Listen *(port map)* 22

4 Prozesse und Funktionen 23

4.1 Prozesse 23

4.1.1 Eigenschaften von Prozessen 24

4.1.2 Modellierung der Zeit in VHDL 27

4.1.3 Wie kann man ungewollte Latches in einem Prozess vermeiden ? 28

4.2 Funktionen 29

5 Einige VHDL Konventionen 31

6 Ausgewählte Beispiele 32

6.1 Verhaltensbeschreibung (RTL) 32

6.1.1 Beschreibung des „and2“ Bausteines 32

6.1.2 If-, Case-Statement bei einem Multiplexer-Baustein 34

6.2 Strukturelle-Beschreibung 35

6.2.1 Beschreibung des „and4“-Bausteines 35

6.2.2 Beschreibung von einer Kompination von 3 und Gattern 36

6.2.3 Struktur eine RS-Flip-Flops 37

VHDL

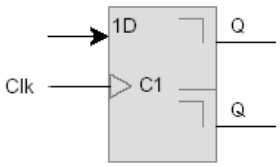
Einführung

1 Einführung

VHDL Beschreibung eines Designs besteht meist aus vier Einheiten:

Package	Typen, Funktionen, Prozeduren, Komponenten, Konstanten
Entity	Schnittstellenbeschreibung (I/O-Signale und Attribute)
Architecture	Strukturale Beschreibung oder Verhaltungsbeschreibung
Configuration	Zuordnung der Architekturvarianten und der Submodule der

Beispiel:




```
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_components.all;
```

```
entity NAND is
port(          a, b:          in   std_logic;
               y    :          out std_logic);
end NAND;
```

```
architecture structural of NAND is
begin
      y <= not (a and b)
end structural;
```

```
configuration NAND_CFG of NAND is
   for structural
   end for;
end structural_CFG;
```

2 Kurze Beschreibung der Design-Einheiten

2.1 Schnittstellenbeschreibung (Entity)

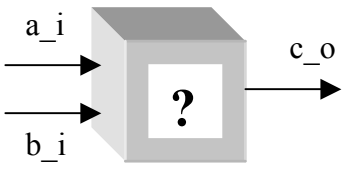
Die einzelnen Modelle eines komplexen VHDL Entwurfs kommunizieren über die Entity, das heißt. über deren Schnittstellenbeschreibung, miteinander. Es wird nichts über das Verhalten ausgesagt. Die "Kommunikationskanäle" nach außen sind die sog. Ports eines Modells. Für diese werden in der Entity Name, Datentyp und Signalflussrichtung festgelegt. Außerdem werden in der Schnittstellenbeschreibung die Parameter deklariert, die dem Modell übergeben werden können (sogenannte "Generics"). Mit Hilfe dieser Parameter lassen sich beispielsweise die Verzögerungs- oder Set-Up-Zeiten eines Modells von außen an das Modell übergeben. Generics können auch die Bitbreite der Ports bestimmen oder den Namen einer Datei enthalten, in der die Programmierdaten eines PLA-Modells abgelegt sind. Auf diese Weise bieten Generics eine hohe Flexibilität bei der Konfiguration von Modellen, da eine Änderung dieser Übergabeparameter kein Neu Compilieren des Modells erfordert.
Im Entity Anweisungsteil (zwischen BEGIN und END) können darüber hinaus noch Anweisungen stehen, die auch für alle der Entity zugeordneten Architekturen gelten sollen.

- ⇨ Schnittstelle des Systems zur „Außenwelt“
- ⇨ Wichtigstes Element sind die Eingänge und Ausgänge (Sie werden als „ports“ Bezeichnet)
- ⇨ Konstanten, Unterprogramme und sonstige Vereinbarungen
- ⇨ Entspricht einer Modulbeschreibung oder einer Funktionsdeklaration in der Programmiersprache C

Syntax:

```
entity identifier is
        generic     ( param1, param2 : [IN] type_name := default_value);
        port        ( port_name1, port_name2, ... : IN type_name;
                      port_name3, port_name4, ... : OUT type_name;
                      port_name5, port_name6, ... : INOUT type_name);
        declarations;
begin
    passive statements,
    concurrent assertion statements
end [identifier];
```

Beispiel:




```
entity black_box is
        generic  ( delay: time := 5 ns );
        port  (
                a_i, b_i  :  in   bit;
                c_o       :  out  bit
              );

end black_box;
```

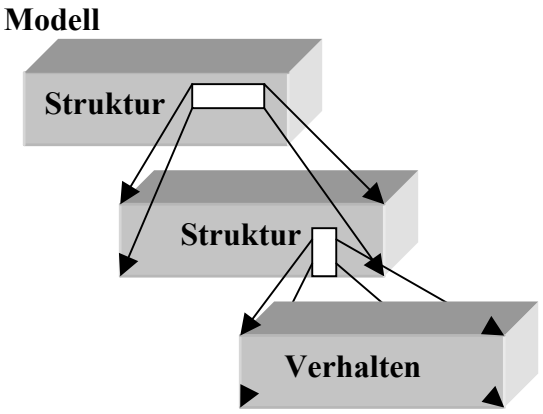
2.2 Architektur (Architecture)

Enthält die Beschreibung der Funktionalität einer Entity. Eine Entity kann mehrere Architekturen enthalten, d.h. es können für eine Komponentenschnittstelle mehrere Beschreibungen auf unterschiedlichen Abstraktionsebenen oder verschiedenen Entwurfsalternativen eingebunden werden.

Dies kann in drei Verschiedenen Beschreibungsmodelle durchgeführt werden:

1. Strukturelle (Hierarchische) Beschreibung

   Strukturale Modellierung bedeutet im allgemeinen die Verwendung und das Verdrahten von Komponenten (Instanzierung von anderen Entities) in Form einer Netzliste. Dies ähnelt dem Schematic Modell in herkömmlichen Designs. Ein Strukturmodell nutzt zur Realisierung der logischen Funktion bereits in VHDL beschriebene Teilkomponenten (z. B. Realisierung eines Mikroprozessors durch Synthese aus ALU, Internal Cache, Register). Strukturmodelle beschreiben, wie die einzelnen Teilkomponenten untereinander zu verbinden sind. Auch ein Strukturmodell kann unterschiedlich stark abstrahieren. Der Abstraktionsgrad ist abhängig von der Komplexität bzw. dem Abstraktionsgrad der im Strukturmodell verwendeten Subkomponenten. Subkomponenten können sowohl als Verhaltensmodell als auch als Strukturmodelle beschrieben sein.



Unterste Ebnen eine Strukturbeschreibung muss eine Verhaltensbeschreibung sein !

2. Verhaltensbeschreibung (Behavior / RTL)

   VHDL wird benutzt, um die Funktion des Designs zu beschreiben. Es ist nur die logische Funktion bekannt, aber keine Architektur. Das Timing ist nur soweit bekannt, um die Funktion zu spezifizieren.

   ⇨ RTL (Register Transfer Logik)

   Das Design enthält Informationen über Architektur logischer Funktionen und Registern. Auf Taktzyklen basierende Timing ist bekannt (Einheitsverzögerung, unit delay). Keine Details über die Technologie. Absolute Delays sind unbekannt. Sehr viele Synthesewerkzeuge verlangen den Code in RTL. Auf dieser Ebene hat der Designer die gesamte Kontrolle über die Register in einem Design.

   Die Zeit wird hier im synchrone Design über einen Takt modelliert, Verwendung von Speicherelementen (FlipFlop's)

   Beispiel: if clock=`1` and clock'event then .....

⇨ Behavior

VHDL wird benutzt, um die Funktion des Designs zu beschreiben. Es ist nur der die logische Funktion bekannt, aber keine Architektur. Das Timing ist nur soweit bekannt um die Funktion zu spezifizieren. Die Synthesewerkzeuge generiern automatisch eine Architektur von Registern und kombinatorischer Logik aus einer reinern Verhaltensmodellierung. Behavioral VHDL wird verwendet um einen Stimulus (Testfunktion für Logik) der einen Standartbaustein zu modelliern, z.B. Nachbildung einen Mikroprozessors, oder es können simulierbare Spezifikationen des systems erstellt werden (Gesamtes System kann auf Funktion simuliert werden)

Zeit wird direkt mit statements modelliert.

Beispiel: z <= a and b after 10 ns;  after ist nicht syntetesierbar !!

3. Datenflussbeschreibung

Mischform zwischen Struktur- und Verhaltensbeschreibung. Alle Anweisungen werden innerhalb einer Architecture "nebenläufig" (quasi parallel) ausgeführt, innerhalb von Prozessen und Unterprogrammen jedoch sequentiell (nacheinander).

Alle drei Beschreibungsstile können in einer einzigen Architektur beliebig gemischt werden !

Syntax:

architecture identifier **of** entity_name **is**
data declarations;
component declarations;
function or procedure declarations;
begin
statemenets;
end identifier;

Der Entity-Name bindet den Architecture-Eintrag mit dem Entity-Eintrag zusammen. Da zu einem Entity-Eintrag auch mehrere Architecture-Einträge geschrieben werden können, hat die Arhitecture eine eigene Identifizierung. In diesem Identifier-Namen ist es sinnvoll auf die Art Logik oder Struktur hinzuweisen. Als Deklaration können hier interne Signale, Variablen, Konstante und bei Strukturelle-Beschreibungen auch Komponente angegeben werden. Bei den Statements kann die Logikbeschreibung in Form von Prozesse, Signalzuweisungen, Wait-Statements, Procedure-Aufruf, Function-Aufruf usw. durchgeführt werden.

Strukturelle (Hierarchische) Beschreibung	**Verhaltensbeschreibung Behavior**	**Datenflussbeschreibung**

architecture structural of rs_ff is
component nand2_socket
port (
a_i, b_i : in bit;
y_o : out bit
);
end component;
begin
u1: nand2_socket
port map (a_i => r_bar, b_i => q, y_o => q_bar);
u2: nand2_socket
port map (a_i => q_bar, b_i => s_bar, y_o => q);
end structural;
	architecture sequentiell of nand is
	begin
	process (a , b)
	begin
	if a = ‘1‘ and b = ‘1‘ then
	y <= ‘0‘;
	else
	y <= ‘1‘;
	end if;
	end process;
	end sequentiell;
		architecture behavioral of nand is
		begin
		y <= not (a and b)
		end behavioral_par;

⇨ Hierarchische Beschreibung ⇨ äquivalent zu Blockschaltbild, Verdrahtung der IC-Sockel ⇨ Deklaration von Komponenten und Signalen ⇨ Komponenten werden instanziert und mittels Signalen verbunden ⇨ Sinnvoll bei großen Designs, die natürlicherweise aus mehreren Modulen zusammengesetzt sind	⇨ Alle Statements innerhalb einer Architekture werden der Reihe nach abgearbeitet ⇨ sequential Statements ⇨ Dabei spielt die Reihenfolge der Statements eine Rolle.	⇨ Hier spricht man von „Concurrent Statements“, es handelt sich um nebenläufige also von gleichzeitigen Ereignissen. ⇨ concurrent signal assignments ⇨ nebenläufige Signalzuweisungen wie bei echter Hardware ⇨ können unter Kontrolle von Steuersignalen stehen ⇨ Reihenfolge der Signalzuweisungen spielt keine Rolle

2.3 Konfiguration (Configuration)

Die Design-Einheit Konfiguration dient zur Beschreibung der Konfigurationsdaten eines VHDL-Modells. Zunächst wird darin festgelegt, welche Architektur zu verwenden ist. Bei strukturalen Beschreibungen kann außerdem angegeben werden, aus welchen Bibliotheken die einzelnen Submodule entnommen werden, wie sie eingesetzt (verdrahtet) werden und welche Parameterwerte (Generics) für die Submodule gelten. Eine Entity kann mehrere Konfigurationen besitzen. In der Konfiguration wird zwischen deklarativen und den eigentlichen Konfigurationsanweisungen unterschieden. Die Konfigurationsanweisungen beschreiben - gegebenenfalls hierarchisch - die Parameter und Instanzen der verwendeten Architektur.

Syntax:

```
configuration conf_name of entity_name is
        ...
        ... -- use- Anweisungen und
        ... -- Attributzuweisungen,
        ... -- Konfigurationsanweisungen
        ...
end [configuration [conf_name] ;
```

Beispiel:

```
configuration rs_ff_config of OF rs_ff is
   for structural
      for all: nand2_socket -- Komponentenkonfiguration
            use entity work.nand2 (behavioral);
      end for;
   end for;
end rs_ff_config;
```

2.4 Package

Es gibt bestimmte Teile in den Entity- bzw. Architecture-Eintrag, die der Anwender auch in anderen Modellen verwenden möchte. Die Beschreibung solche Teile wie zum Beispiel Typdeklarationen, Konstanten oder Prozeduren und Funktionen können wir in sogenannten Packages abspeichern. Der Package ist Teil einer Library. Standardmäßig werden bei VHDL die Packages standard und textio bereitgestellt, die allgemeine Typen wie character, string, boolean, bit, bit_vector, integer usw., sowie die zugehörigen Grundfunktionen enthalten. In der Regel steht auch ein Package für den verwendeten Logiktyp zur Verfügung, z.B. die Definitionen der neunwertigen Logik std_logic im Package std_logic_1164 von IEEE.

In konkreten Fällen können wir diese Beschreibungen in den Entity- bzw. Architecture-Einträgen aus den Packages namentlich aufrufen.

Syntax:

package *identifier* ***is***
    declarations;
end *identifier;*

Der Package Body kann neben der Definition von Unterprogrammen auch spezielle Deklarationsanweisungen enthalten. Der Zusammenhang mit dem Package wird durch den identischen Namen (pack_name) hergestellt.

Syntax:

package body *identifier* ***is***
    declarations;
end identifier;

2.5 Library

Bei der Verwendung von vordefinierten Funktionen, Proceduren usw. müssen wir die Bibliotheken angeben, wo die Packages abgespeichert sind. Die sogenannte "library"- bzw. "use"-Klausel stehen dafür im Entity- oder eventuell im Architecture-Eintrag.

Die "use"-Klausel hat drei mögliche Formen:

- ⇨ *library_name.package_name*
- ⇨ *package_name.object*
- ⇨ *library_name.package_name.object*

Syntax: **library** lib_name;
use lib_name.package.all, lib-name.package.xy,...

*Beispiel***:** library **STD;** use **STD.STANDARD.**all**;**

3 Elemente der Sprache VHDL

⇨ VHDL ist nicht case sensitiv

⇨ Kommentare: Kommentare werden bei VHDL mit zwei aufeinanderfolgenden Bindestrichen dargestellt.

Syntax: --

Beispiel: -- Technische Informatik

⇨ Zahlen: Zahlen können als Dezimalzahl oder als Zahl in Zusammenhang mit einer Basis dargestellt werden.

Syntax: decimal literal
integer[.integer][exponent]
based literal
based # based_integer[based_integer]#

Beispiel: 1, 2, 3, ...
1.3, 4.711
2#10010011# Dual

⇨ Zeichen: Zeichen werden mit einfachen Anführungszeichen beschrieben Zeichengrößen, im Englischen "characters", sind einzelne Zeichen, die in Hochkomma stehen müssen. Sie sind allerdings case-sensitiv, das heißt ein 'a' und ein 'A' werden in VHDL unterschieden.

Syntax: 'character'

Beispiel: 'F', '*', '?', ...

⇨ Zeichenketten: Zeichenketten werden mit doppelten Anführungszeichen beschrieben

Syntax: "string"

Beispiel: "Technische Informatik"

3.1 VHDL Objekte

Bei der Beschreibung des Logik und Zeitverhalten zwischen Aus- und Eingängen mit Hilfe von Deklarationen und Statements verwenden wir Objekte, die bestimmte Daten oder einen bestimmten Wert halten können. Alle Daten in VHDL werden über Objekte verwaltet. Objekte besitzen einen definierten Datentyp und einen Bezeichner - den identifier, z.B. Integer, Bit, Bit_Vektor usw., und kann einen bestimmten Anfangswert annehmen.

Es gibt vier verschiedene Klassen von Objekte können in VHDL, um Daten zu verwalten definiert werden:

- ⇨ Signale: bekommt den Wert nach einer Verzögerungszeit zugewiesen. Wert kann gelesen und verändert werden. Zusätzlich wird der zeitliche Verlauf gespeichert, so das auch auf Werte in der Vergangenheit zugegriffen werden kann, bzw. Werte für die Zukunft vorbestimmt werden können. (x <= '0' after 10 ns;)
- ⇨ Variable**:** bekommt Wert unmittelbar, Wert kann gelesen und neu zugewiesen werden, Wert hat aber keine Vergangenheit
- ⇨ Konstanten: hat immer den selben Wert, Wert wird nur einmal zugewiesen
- ⇨ Daten: Folgen von Werten die geschrieben oder gelesen werden

Syntax:

object_class data_name1, data_name2, ...: data_type_name {:= default_value};

Syntax für Daten:

file data_name1, data_name2, ...: data_type_name **is in | out** file_name;

Beispiele:

```
signal      ist1, ist2   : bit := ´0´;
variable    zeit_v       : integer := 255;
constant    stadt_c      : integer := 50;
file        input        : string is in "/usr/textin";
```

3.1.1 Signale versus Variable

Signale erhalten nach einer Zuweisung ihren neuen Wert erst nach einem Δt. Variablen erhalten nach einer Zuweisung den neuen Wert sofort. Variablen werden innerhalb eines Prozesses deklariert, sie gelten nur in diesem Prozess.

- ⇨ Zuweisungen auf Variable werden sofort wirksam
- ⇨ Zuweisungen auf Signale werden erst am Ende eines Prozesses wirksam
- ⇨ Variable behalten ihren Wert zwischen den Prozessabläufen
- ⇨ Signale können Variablen, Variable können Signale zugewiesen werden.

	Variable	**Signal**
Symbol	**:=**	**<=**
Deklaration	nur im Prozess	nur bei Architecture, Block, Entity
Gültigkeit	sofort	bei erreichen des nächsten wait, vorher nur vorgemerkt, können jederzeit wieder überschrieben werden
Simulationszeitpunkt	mehrere Werte annehmen	nur eine Wert
Vergangenheit	hat keine	hat eine Wert in der Vergangenheit, diesen kann man abfragen

Syntax: signal_name <= wert;
variable_name := wert;

Beispiel:

```
signal    a, b  : bit;
variable  c_v   : integer;
process
   begin
      a   <= 0;
      c_v := 1;
      wait ....
      a   <= 1;
      b   <= a;      -- b = 0 !
      c_v := 2;      -- c_v = 2 !
      wait ...
      b <= a;        -- b = 1!
end process;
```

Signale dienen dazu, Daten zwischen parallel arbeitenden Modulen auszutauschen.
Verschiedene Module können auf das gleiche Signal schreiben.

Dadurch können Busleitungen modelliert werden.
Signale entsprechen, vereinfacht gesprochen, Verbindungsleitungen in elektronischen Schaltungen.

Variablen sind (prinzipiell) nur innerhalb eines
Prozesses gültig.

3.2 Signalzuweisungen (Signal Assignments)

3.2.1 Concurrent Signal Assignment

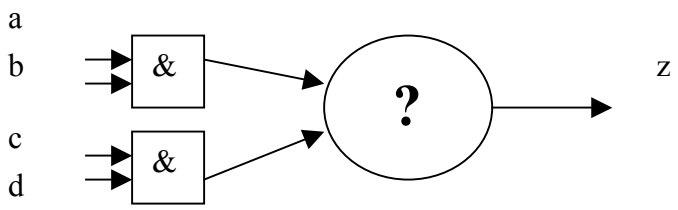
Eine Signalzuweisung (signal Assignment) ist ein 'Concurrent' - Statement; sie ist das wichtigste 'Behavioral' - Statement von VHDL. Signale übertragen Informationen zwischen Prozessen. Die Signalzuweisung ändert links nur dann den Wert, wenn rechts ein Ereignis auftritt. Der zugewiesene Wert ist ein Folgeereignis, das um ein Δt verzögert wirkt.

Syntax: signal_name <= wert;

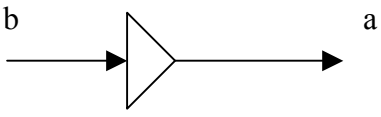
Vorsicht bei nebenläufigen, parallelen (concurrent) Mehrfachzuweisungen. Sie verhalten sich anders als bei sequentiellen Mehrfachzuweisungen.

Beispiel:

```
architecture concurrent of multiple is
        signal a, b, c, d  :    std_logic;
        signal z           :    std_logic;
begin
        z  <= a and b;          -- durch Verwendung von gstd_logic
                                -- (Resolution Function)
        z  <= c and d;          --  kann das Problem behoben werden !
end concurrent;
```



Beispiel: a <= b; --entspricht einen Treiber (Buffer)

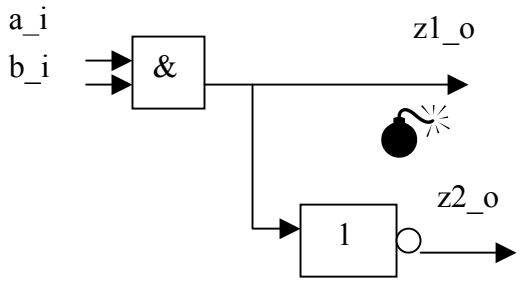


3.2.2 Interne Signale

Innerhalb der Schaltung werden Signale als Logische Verknüpfungen verwendet. Es kann zu folgenden Problem führen:

Beispiel:

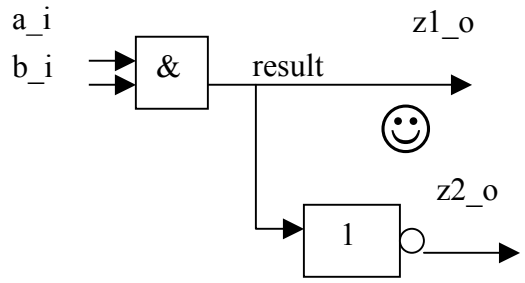
```
---------- e n t i t y ----------
entity  logic is
        port (
                a_i, b_i      :  std_logic ;
                z1_o, z2_o  :  std_logic
               );
end logic;
---------- a r c h i t e c t u r e ----------
architecture mistake of logic is
      begin
           z1_o  <= a_i and b_i;
           z2_o  <= not z1_o;
end mistake;
```



z1_o kann nur ausgegeben werden, jedoch nicht zurückgelesen werden. !

Beispiel:

```
---------- e n t i t y ----------
entity  logic is
        port (
                a_i, b_i      :  std_logic ;
                z1_o, z2_o  :  std_logic
               );
end logic;
---------- a r c h i t e c t u r e ----------
architecture ok of logic is
     signal result : std_logic ;
     begin
           result  <= a_i and b_i;
           z1_o  <= result;
           z2_o  <= result;
end ok;
```



3.2.3 Bedingte Signalzuweisung

⇨ Conditional Signal Assignment

when - Konstrukt: Eine Signalzuweisung hängt oftmals von Bedingungen ab. Um nicht mit Prozessen bzw. dem IF - THEN - ELSE - Statement arbeiten zu müssen, gibt es eine Möglichkeit,mittels WHEN eine Zuweisung nur dann erfolgen zu lassen, wenn die nachfolgende(n) Bedingung(en) wahr ist (sind). Es sind auch mehrere ELSE Abfragen erlaubt, die allerdings in ihrer Reihenfolge abgearbeitet werden. Trifft eine Bedingung zu, werden die folgenden Abfragen nicht mehr bearbeitet.

Beispiel: output **<=** ’1’ **after** 10 ns **when** enable = ’1’ **else** ’0’ **after** 10 ns;

⇨ Select Signal Assignment

with - select - Konstrukt: Diese Anweisung kann nur von einer Bedingung abhängen. Sie erlaubt aber, eine große Anzahl an Werten mit wenig Programmieraufwand abzufragen. Innerhalb der WITH - SELECT - Anweisung werden die Abfragen sequentiell abgearbeitet. Dies bedeutet, dass bei der Programmierung auf Prioritäten bei Wertebereichsüberschneidungen geachtet werden muss. Aus diesen Grund müssen alle Bedingungen formuliert werden und es darf keine Bedingung doppelt auftreten.

Beispiel:
```
signal int : integer;
with  int  select
                out <= "000" when  0,
                       "001" when  1,
                       "010" when  2,
                       "011" when  3,
                       "111" when  others;
```

3.3 Datentypen und Typdeklarationen

3.3.1 Vordefinierte Datentypen

Datentypen der IEEE Standard-Bibliothek:

- ⇨ bit, bit_vector, kann den Logischen Werte 0 oder 1 annehmen
- ⇨ boolean, kann die Werte false oder true annehmen
- ⇨ string, Array Typ zu character
- ⇨ character, kann die Werte des ASCII-Zeichencodes annehmen
- ⇨ real, Floating Point, $-1.0*10^{38}$ bis $+1.0*10^{38}$
- ⇨ integer, Fix Point, Bereich $-2^{31}-1$ bis $+2^{31}-1$

Beispiele:
```
var_1_v : real;
var_2_v : integer;
var_3_v : bit_vector (3 downto 0);
var_4_v : std_logic_vector (3 downto 0);
```

Was tun, wen das Signal bei der Simulation einen hochohmigen Anfangswert (tristate, high impedance) annehmen soll? Der vordefinierte Typ Bit hat nur die Wahl ´0´ oder ´1´. Für solche Fälle stehen von verschiedenen Anbietern vordefinierte Pakete zur Verfügung. Abgelegt sind diese in einer Standard Bibliothek

Ein oft verwendeter Datentyp, der nicht in der Sprache VHDL definiert ist, jedoch von externen Bibliotheken, mit dem Befehl "USE IEEE.std_logic_1164.all", importiert werden kann, ist std_logic. Dieser Datentyp implementiert eine 9-wertige Logik. Diese zusätzliche Zustände existieren nicht im physikalischen Sinne. Sie werden sehr wohl in der Simulation verwendet. Dadurch wird die Auswertung bzw. die Fehlersuche in den Simulationsergebnissen wesendlich vereinfacht.

Der Basistyp des Logiksystems std_ulogic (u steht für "unresolved"), ist als Aufzähltyp für folgenden Signalwerte deklariert:

```
Type std_ulogic is (
                      ´0´   -- Logikpegel '0'
                      ´1´   -- Logikpegel '1'
                      ´L´   -- gezogener Logikpegel "Low", der durch Pull-Down-Widerstand
                      ´H´   -- gezogener Logikpegel "High", der durch Pull-Up-Widerstand
                      ´X´   -- unbekannter Zustand (unknown)
                      ´W´   -- "weak":  analog den Zustand 'X' für die Pegel 'L' und 'H'
                      ´U´   -- uninitialisierter Zustand
                      ´Z´   -- hochohmiger Zustand (Tristate)
                      ´-´   -- "dont't care", kann zur Logikoptimierung verwendet werden
                    );
```

Weiters gibt es auch noch std_logic:

Standart resolved Logic, steht für mehrfach Zuweisungen, z.B. bei Tristate-Bussystemen oder Wired or-verknüfungen Auagängen. Hir gibt es eine sogennate resolution-function, das ist eine Funktion, die klärt, welche logischen Pegel sich bei einer Mehrfachzuweisung ergeben.

3.3.2 Feldtypen

⇨ Vektoren können als eindimensionales Feld aufgefasst werden.

Syntax: **type** *vector_type* **is array** *index_constraint* **of** *base_typ*;

Als Index (index_constraint) können beliebige diskrete Zahlentypen (ganzzahlige Zahlen, Aufzählungen,...) verwendet werden.

Beispiel: **type** a **is array** (1 to 10) **of character**;

⇨ Mehrdimensionale Felder

Matrizen können als zweidimensionales Feld aufgefasst werden.

Syntax: **type** *vector_type* **is array** *index_constraint* **of** *base_typ*;

Beispiel: **type** vector **is array** (1 to 10) **of integer**;
type matrix1 **is array** (1 to 5) **of array**;
type matrix2 **is array** (1 to 10, 1 downto 100) **of integer**;

3.3.3 Benutzerdefinierte Datentypen

Syntax: **type** *my_type* **is integer**
type *new_typ* **is range** *range_low* **to** *range_high*
type *new_typ* **is range** *range_high* **downto** *range_low*

Beispiel: type my_type is range 0 to 99 of integer

3.3.4 Aufzählungstypen

Um eine 4-wertige Logik mit den Zuständen '0', '1', 'X' und 'Z' zu implementieren, benötigt man die Definition eines neuen Typen:

Beispiel: **type** fourval **is** (´0´, ´1´, ´X´, ´Z´);

Hier sind die einzelnen Zustände Zeichen, es können jedoch auch andere Datentypen sein:

Beispiel: **type** color **is** (red, green, blue);

3.3.5 Physikalische Typen

Diese Typart ist sowohl vordefiniert, als auch vom Benutzer zu definieren. Der einzige vordefinierte physikalische Datentyp im IEEE Standard ist die Zeit (time).

Beispiel:
```
type time is range -(2**31-1) to (2**31-1)
        units
                fs;
                ps = 1000 fs;
                ns = 1000 ps;
                us = 1000 ns;
                ms = 1000 us;
                sec = 1000 ms;
                min = 60 sec;
                hr = 60 min;
        end units;
```

Beispiel:
```
type  widerstand is range  1 to 10E6
        units
                ohm;
                kohm = 1000 ohm;
                Mohm = 1000 kohm;
        end units;
```

Diese Typendefinition ist vom Typ 'Physical', wobei `ohm` die Basiseinheit und `kohm`, bzw. `Mohm` sekundären Einheiten entsprechen. Eine sekundäre Einheit ist die Basiseinheit, multipliziert mit einem Faktor. Weitere sekundäre Einheiten können auf diesen weiter aufbauen. Dies führt zu einer sehr übersichtlichen Darstellung und verdeutlicht die Zusammenhänge der verschiedenen Einheiten. Es ist darauf zu achten, daß Sekundäreinheiten auf Einheiten aufbauen, die schon definiert wurden. Das bedeutet, daß die beiden Definitionen für `kohm` und `Mohm` nicht vertauscht werden können, da `Mohm` auf der Definition der Einheit kohm aufbaut.

3.4 Operatoren

Operatoren werden wie in anderen Programmiersprachen behandelt. Sie verknüpfen Operanden miteinander und lösen Berechnungsfunktionen aus, die vordefiniert sind.

Gruppe	Symbol	Funktion
arithmetisch (binär)	+	Addition
	-	Subtraktion
	*	Multiplikation
	/	Division
	mod	modulo
	rem	remainder
	**	exponential
arithmetisch (unär)	+	unär plus
	-	unär minus
	abs	Absolutwert
Vergleich	=	gleich
	/=	ungleich
	<	kleiner als
	>	größer als
	<=	kleiner als oder gleich
	>=	größer als oder gleich
logisch (binär)	and	logisches und
	or	logisches oder
	nand	Komplement von and
	nor	Komplement von or
	xor	logisches exklusives oder
logisch (unär)	not	Komplement
Verknüpfung (concatenation)	&	Verknüpfen von Vektoren

3.5 Attribute

Mit Hilfe von Attributen können bestimmte Eigenschaften von Objekten oder Typen abgefragt werden. Die Verwendung von Attributen kann eine VHDL - Beschreibung wesentlich kürzer und eleganter gestalten. Außerdem lässt sich mit Hilfe von Attributen der Anwendungsbereich von VHDL - Modellen erhöhen.

Zur Realisierung taktgesteuerter Vorgänge wie z.B. State Machines ist es notwendig den Zustand eines Signals (allg. Objekts) zu kennen, den es vor dem Aufruf (z.B. in einem Prozess) hat. Soll eine State Machines mit der steigenden Flanke von Clock getaktet werden, so ist es er erforderlich den Wert von Clock vor dem Aufruf sicherzustellen. Dazu kennt VHDL Attribute mit einem Hochkomma (sprich: tick) gekennzeichnet sind. Mit Attributen werden Eigenschaften von VHDL -Konstrukten festgelegt.

Syntax: objektname' attribute = identifier;

Beispiele:

Name	Funktion
T'left	Linke Grenze von T
T'right	Rechte Grenze von T
T'low	untere Grenze von T
T'high	obere Grenze von T
T'pos(x)	Nummer der Position von x in T
T'val(n)	Wert der Position n in T
T'pred(x)	Wert des Elementes welches sich eine Position unter dem Element x befindet.
T'succ(x)	Wert des Elementes welches sich eine Position über dem Element x befindet.
T'event	Änderung von T im aktuellen Simulationszyklus (Signalattribut)
T'stable	Keine Flanke im aktuellen Simulationszyklus (Signalattribut)

3.6 Interface-Listen

3.6.1 Ports

Die Ports einer Entity beschreiben die Schnittstellen der Schaltung zur Außenwelt. Innerhalb der Schaltung werden Signale in logischen Verknüpfungen verwendet.

Syntax: **port** (name1..n **:** Portrichtung Signaltyp**);**

Portrichtung:

- ⇨ in: Eingangsport, kann nur gelesen werden. Auf ein Signal in kann keine Zuweisung gemacht werden. Es kann nur auf der rechten Seite von Signalzuweisungen stehen.
- ⇨ out: Ausgangsport, kann nicht gelesen werden, es kann nur auf der linken Seite von Signalzuweisungen stehen.
- ⇨ inout: Bidirektionaler Port, es kann gelesen und beschrieben werden, es kann auf beiden Seiten von Signalzuweisungen stehen.
- ⇨ buffer: Ausgangsport wie out, es kann auch gelesen werden

Beispiel:
```
port (
        a_i, b_i   :  in   bit;
        c_o        :  out  bit
      );
```

Die Objekt-Klasse kann beim Port nur Signal und bei Generic/Parameter nur Constant sein, und muß deshalb nicht angegeben werden. Als zusätzliche Information erscheint hier die Verbindungsrichtung (Mode) in, out, inout bzw. buffer vor dem Datentyp. GENERIC optional zur Paramterübergabe an Architecture

3.6.2 Verbindungs-Listen *(port map)*

Für die Verbindung von Komponenten in einem strukturellen Design benützen wir Verbindungs-(Association) Listen. In dem folgenden Beispiel sehen wir wie die im lokalen Komponenten-Deklaration angegebenen Pins a_i, b_i und c_i mit dem "örtlichen" Pins in1, in2, in3, in4 in einem sogenannten port map verbunden werden. Auch Verbindungen zwischen den Komponenten innerhalb der Strukturelle Modellbeschreibung können im port map mit intern definierten Hilfssignalen realisiert werden.

Syntax: **port map (**globaler NAME **=>** lokaler NAME**, ..,..,..);**

Beispiel:

```
……
component AND2
        port (
              a_i, b_i  :  in Bit;
              c_o       :  out Bit
             );
end component;
…….
…….

begin
        u1: and2        port map (a_i => in1, b_i => in2, c_i => ist1);
        u2: and2        port map (a_i => in3, b_i => in4, c_i => ist2);
…….
```

4 Prozesse und Funktionen

In VHDL gibt es, ähnlich zu Programmiersprachen wie C oder Pascal, die Möglichkeit der Realisierung von Unterprogrammen mittels Prozeduren und Funktionen.

⇨ Prozeduren werden mit einer Liste von Argumenten aufgerufen, die nicht nur Eingänge, sondern auch Ausgänge oder bidirektional sein können. Der Aufruf einer Prozedur ist ein eigenständiger Befehl, er kann nebenläufig oder sequentiell erfolgen.

⇨ Funktionen hingegen werden mit verschiedenen Argumenten aufgerufen und liefern einen Ergebniswert zurück. Der Funktionsaufruf (mit Eingabeargumenten) kann an der gleichen Stelle in Ausdrücken stehen, an der Typ des Ergebniswertes erlaubt ist.

4.1 Prozesse

Prozesse dienen als Umgebung für sequentielle Befehle. Prozesse selbst gelten als nebenläufige Anweisungen, d.h. innerhalb einer Architektur können mehrere Prozesse definiert werden, die gleichzeitig aktiv sind. Prozesse werden durch die folgenden Elemente aktiviert oder gestoppt:

⇨ Sensitivity-Liste — Ein Prozess wird bei der Initialisierung der Simulation einmal durchlaufen und danach erst wieder aktiviert, wenn sich ein oder mehrere Signale in der Sensitivity-Liste ändern.

⇨ Wait-Anweisung — Bei der Initialisierung wird der Prozess bis zur ersten Wait Anweisung durchlaufen. Danach wird der Prozess erst dann wieder aktiviert, wenn die Bedingung der Wait Anweisung erfüllt ist.

⇨ ohne Sensitivity / Wait

→ Diese Prozesse werden ständig zyklisch durchlaufen.

→ entsprechen Programmen konventioneller Programmiersprachen wie

→ tauscht nur über Signale Informationen mit seiner Umwelt aus

→ wird ein Zeit- bzw. Kausalitätsverhalten zugeordnet

4.1.1 Eigenschaften von Prozessen

⇨ Sequentielle Anweisungen

→ Definieren Algorithmen
→ werden nacheinander abgearbeitet
→ können "normale" Software repräsentieren
→ sind der Inhalt vom Prozess-Anweisungsteil
→ werden ohne Zeitverzug ausgeführt

⇨ Prozess entspricht einer Hardwarekomponente die auf Eingangsänderungen reagiert

⇨ Prozess ist auf diese Eingänge = Signale sensitiv

⇨ Er ist nur aktiv, wenn sich diese Signale ändern

Syntax:

mit Sensitivity List	ohne Sensitivity List
process [(sig1 [, sig2, ...])] declaration of types, subtypes constants, files variables, functions definition of functions, attributes **begin** sequential statements **end process**;	**process** declaration of types, subtypes constants, files variables, functions definition of functions, attributes **begin** sequential statements **wait** *expression*; sequential statements **end process**;

Beispiel:

```
architecture example of and2 is
begin
        p: process (A, B)
                variable x_v: integer;
           begin
                x_v := A and B;
           end process p;
end example;
```

Beispiel:

```
architecture  sequential_2 of  latch is
begin
        q_assignment:process  -- Aktivierung über wait-Anweisung
                        begin
                                if clock = '1' then q <= d;
                        end if;
                        wait on d, clock ; -- entspricht "sensitivity-list"
                        end process  q_assignment ;
end  sequential_2;
```

Es wird solange gewartet, bis eine Änderung auf dem Signal d oder clock stattfindet. Es ist NICHT möglich zeitliches Verhalten in eine synthetisierbare VHDL- Beschreibung einzubauen. Konstrukte wie beispielsweise „wait for 10 ns“ können nicht auf eine Technologie abgebildet werden!

Beispiel:

```
---------------------- e n t i t y ----------------------------------
entity mult is
        port (
                a_i, b_i   : in integer := 0;
                y_o        : out integer
               ) ;
end mult ;
---------------------- a r c h i t e c t u r e  ----------------------
architecture number_one of mult is
begin
        process (a, b)  --Aktivierung über Sensitivity-Liste
                variable v1_v, v2_v : integer := 0;
        begin
                v1_v := 3 * a + 7 * b ;    -- Variablenzuweisung
                v2_v := a * b + 5 * v1 ;   -- Variablenzuweisung
                y <= v1_v + v2_v ;         -- Signalzuweisung (Ports sind interne Signale)
        end process;
end number_one;
```

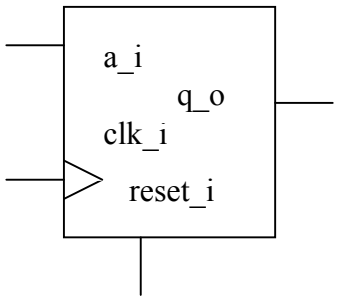
Die Prozess wird nur dann durchlaufen, falls eine Änderung auf den Signalen der Sensitivity Liste (hier "a" und "b") stattfindet.

Innerhalb eines Prozesses werden alle Anweisungen sequentiell ausgeführt. In VHDL sind folgende sequentielle Anweisungen definiert:

- ⇨ **Signalzuweisung**
- ⇨ **Variablenzuweisung**
- ⇨ **Report**
- ⇨ **Wait**
- ⇨ **if - elsif - else - end if**
- ⇨ **case - when - end case**
- ⇨ **loop - end loop**
- ⇨ **while - end loop**
- ⇨ **for - end loop**

Beispiel:

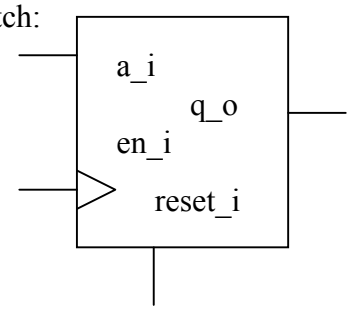
ff:




```
---------------------- e n t i t y ----------------------------------
entity ff is
        port (
                a_i, resret_i, clk_i :  in std_ulogic;
                q_o                  :  out sdt_ulogic
              );
        end ff;
---------------------- a r c h i t e c t u r e  ----------------------
architecture behavioral of ff is
begin
     flipflop : process (clk_i,reset_i)        -- prozess ist auf clk_i  und reset_i sensitiv
                begin
                        if reset_i = '1' then                    -- Beginn asynchrone Teil
                          q_o <= 0;                              -- Rücksetzen
                        elsif (clk'event and clk = '1') then     -- Beginn synchrone Teil
                          q_o <= a_i;
                         end if;
                end process flipflop;
end behavioral;
```

Beispiel:

latch:




```
latch: process (a_i, en_i, reset_i)   -- prozess ist auf a, en und reset sensitiv
        begin
                if reset_i = '1' then
                        q_o<= 0;
                elsif  en_i = '1'  then
                        q_o <= a;
                end if;
       end process latch;
```

4.1.2 Modellierung der Zeit in VHDL

⇨ Behavioural:

Zeit wird direkt mit statements modelliert. Vorsicht, man kann Beschreibungen mittels „after...“ nicht für die Synthese verwenden..

Beispiel: c **<=** a **and** b **after** 10 ns**;**

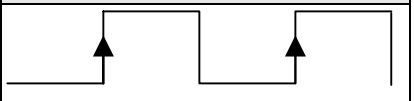
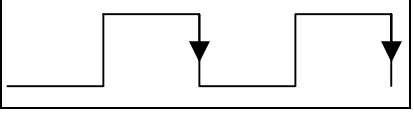
⇨ RTL

In synchronen Designs wird Zeit über einen Takt modelliert (Verwendung von Attributen), das heißt durch Verwendung von Speicherelementen. Diese Elemente sind Synthese fähig. Eine Synchrone Schaltung bedeutet dabei, dass jedes taktgesteuertes Bauteil zur gleichen Zeit des Takt erhält. Die Maximale Taktfrequenz (critical path) ist durch Delayzeiten der einzelnen Bauelemente gegeben.

Beispiel:

```
process (clk_i, reset_i)        -- prozess ist auf clk_i  und reset_i sensitiv
        begin
                if reset_i = '1' then                   -- Beginn asynchrone Teil
                  q_o <= 0;
                elsif (clk'event and clk = '1') then    -- Beginn synchrone Teil
                  q_o <= a_i;
                 end if;
        end process flipflop;
```

Durch die Zeile **elsif** (clk**'event** and clk = '1') **then** … bekommt man ein synchrones Verhalten.

Flanke	Syntax	
steigende	**elsif** (clk**'event** and clk = '1') **then** ….	
fallende	**elsif** (clk**'event** and clk = '0') **then** ….	

4.1.3 Wie kann man ungewollte Latches in einem Prozess vermeiden ?

Es können unabsichtliche Latches erzeugt werden, wenn ein einem kombinatorischen Prozess nicht alle möglichen Zweige einer IF Anweisung berücksichtigt werden. Das heißt sie entstehen durch unvollständige Signalzuweisungen.

Folgende Abhilfen kann man vornehmen:

⇨ Bei einer if Anweisung immer einen else Zweig verwenden, um der Ausgangsgröße immer einen Wert zuzuweisen.

Beispiel:

```
process (a,b)
begin
        if a = '1' then
                q<= b;
        else
                q<= 0;
        end if;
end process;
```

⇨ Eine Default Anweisung vor der if Anweisung verwenden, damit wird der Ausgangsgröße auf jeden Fall eine Wert zugewiesen.

Beispiel:

```
process (a,b)
begin
        q<= 0;
        if a = '1' then
            q<= b;
        end if;
end process;
```

⇨ Transparentes Latch entsteht im untenstehenden Beispiel dadurch, dass der else Zweig fehlt. Nimmt clock den Wert clock = '0' an, kann dem Signal x kein neuer Wert zugewiesen werden, dass heißt Signal x behält vorherigen Wert.

Der fehlende else Zweig führt zu einer unvollständigen Signalzuweisung. Obwohl der Prozess bei einer Änderung des Signals a aktiviert wird, ändert x den Wert nicht, solange das clock = '0' ist. Bei clock = '1' verhält sich der Prozess transparent, das heißt das Ausgangssignal x folgt unmittelbar dem Eingang a.

Beispiel:

```
process (clock_i, a)
begin
        if (clock = '1') then
            x <= a;
        end if;
end process;
```

4.2 Funktionen

In Funktionen kann der Anwender bestimmte Operationen selbst definieren. Wir sollen hier die Definition und den Aufruf der Funktion klar unterscheiden. Die Definition und Beschreibung der Funktion erfolgt im Architecture - Eintrag bei der Deklarationen. Der Aufruf der Funktion erfolgt in den Statements des Architecture - Eintrages.

Syntax:

```
function identifier (interface_list) return type is
        declarations;
begin
        statements;
        return value;
end identifier;
```

Beispiel:

```
---------------------- e n t i t y -----------------------------------
entity and2 is
        port (
                a_i, b_i   : in bit;
                c_o        : out bit
              );
end and2;
---------------------- a r c h i t e c t u r e  ----------------------
architecture beh_and2 of and2 is
        function and_fu (x,y:in bit) return bit is
        begin
                if x = '1' and y = '1' then
                        return '1';
                else
                        return '0';
                end if;
        end and_fu;

begin
        --aufruf in statements-session
        c <= and_fu (a,b) after 10ns;
        wait on a,b;
        .......
        .......
```

Eine Alternative ist die Definition der Funktion in einem Package. Dazu können wir zum Beispiel unter der Library test_lib ein Package definieren. In dem Package können wir mehrere Funktionen oder auch andere Elemente beschreiben. Deshalb bekommt das Package einen Sammelnamen, z.B. MEINFUNC. Wichtig ist den Unterschied zwischen Package und Package-Body zu verstehen. Am besten können wir es mit der Definition und Aufruf einer Funktion vergleichen. Man könnte etwa sagen, der Aufruf kommt in das Package und die Definition ins Package-Body hinein.

Beispiel:

```
---------------------- p a c k a g e ------------------------------------
package meinfunc is            --- wird in Library test_lib definiert
        function and_fu (x,y:in bit) return bit
end meinfunc;
---------------------- p a c k a g e   b o d y -------------------------
package body meinfunc is
        function and_fu (x,y:in bit) return bit is
        begin
                if x='1' and y='1' then
                        return '1';
                else
                        return '0';
                end if;
        end and_fu;
end meinfunc;
```

Das "Package" / "Package Body" Paar wird wie Entity / Architecture analisiert. Wenn wir diese Funktion jetzt in verschiedenen Modellen anwenden wollen, müssen wir die Bibliothek (worin das Package analysiert wurde !!) den Package-Namen und entweder die gewünschte Funktion oder "all" für alles in dem Package angeben.

Unsere Beispiel für das AND2-Gatter können wir somit wie folgt abkürzen:

```
---------------------- e n t i t y ---------------------------------
library test_lib;
use test_lib. meinfunc.and_fu;

entity and2 is
        port  (
                a_i, b_i :  in bit;
                c_o      :  out bit
               );
end and2;
--------------------- a r c h i t e c t u r e  ----------------------
architecture  beh_and2 of and2 is
begin
        c <= and_fu(a,b) after 10ns;
.......
.......
```

5 Einige VHDL Konventionen

- ⇨ Alle Dateien tragen die Dateiendung ".vhd"
- ⇨ Jede VHDL Datei beinhaltet genaue eine Entity, Architecture und Configuration.
- ⇨ Die VHDL Dateien tragen die Namen der zugehörigen Entity's
- ⇨ Für die Architekturen sind vorzugsweise die Bezeichnungen "RTL" für Beschreibungen auf Register-Transfer-Ebene, Verhaltensbeschreibungen für "behavor" und "structural" für Strukturbeschreibungen zu verwenden.
- ⇨ Die Konfiguration trägt den Namen der Entity mit dem Zusatz "_CFG"
- ⇨ Alle Taktsignale tragen die Bezeichnung "clk" oder "clock"
- ⇨ Eingangssignale haben den Zusatz "_i"
- ⇨ Ausgangssignale haben den Zusatz "_o"
- ⇨ Variable haben den Zusatz "_v"
- ⇨ Konstante haben den Zusatz "_c"
- ⇨ Reine Signale haben keinen Zusatz
- ⇨ Invertierte oder low-aktive Signale werden durch den Zusatz "_n" gekennzeichnet
- ⇨ Signale vom Typ buffer werden nicht verwendet
- ⇨ Um die Simulations- und Syntheseergebnisse möglichst einfach nachvollziehen zukönnen, ist es wichtig, alle ausgeführten Schritte exakt zu dokumentieren; dies geschieht am einfachsten mit Hilfe von Skript-Dateien, über die der Entwurf gesteuert wird.
- ⇨ Grundsätzlich gilt: Synthetisierbarer VHDL Code ist „einfach" und enthält keine verschachtelten Schleifenkonstrukte und umständlichen Spagetticode. Die zu implementierende Architektur sollte vor der Beschreibung in VHDL bereits bekannt sein.

☺ ***Think Hardware*** ☺

- ⇨ Eine VHDL Beschreibung ist die textuelle Beschreibung von Hardwarekomponenten
- ⇨ Das Syntheseergebnis ist bei einer einfachen Beschreibung oft optimal
- ⇨ Vor der Synthese wird die Schaltung getestet. Dadurch kann die Funktionalität der Schaltung überprüft werden
- ⇨ Für den Test der Schaltung wird das Modell in eine Testbench eingebettet

6 Ausgewählte Beispiele

6.1 Verhaltensbeschreibung (RTL)

6.1.1 Beschreibung des „and2“ Bausteines

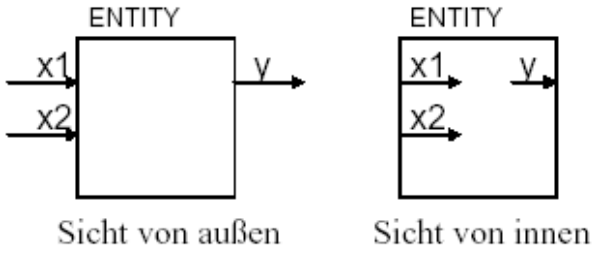
ENTITY

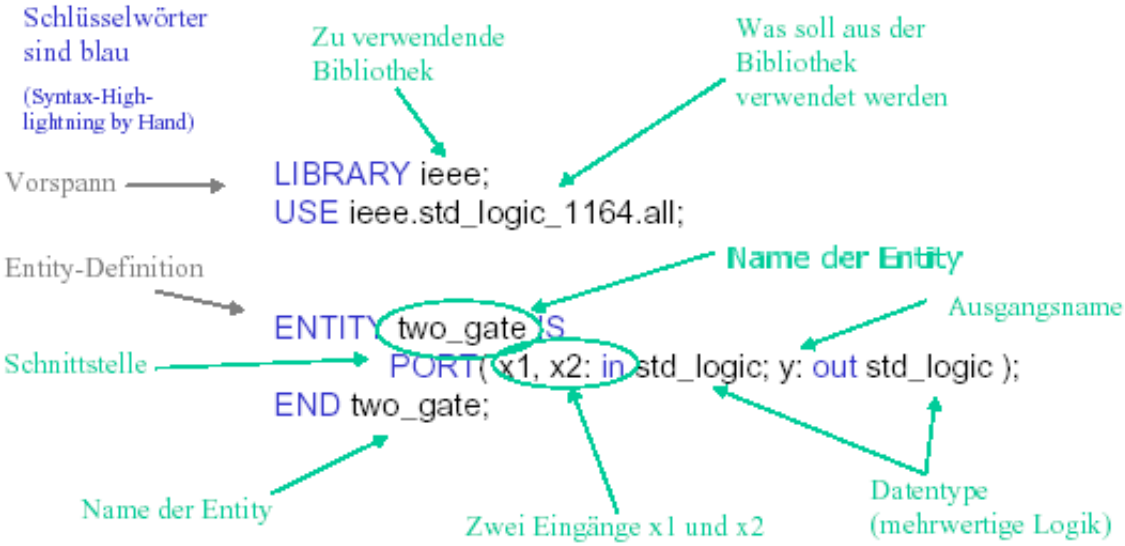
⇨ logische Kapselung einer Einheit und Definition der Schnittstelle

⇨ Daten-Schnittstellen haben Bezeichner, Richtung und Datentyp

→ Bezeichner sind innen wie außen bekannt.
→ Richtungs Beispiel. : in, out, inout, buffer
→ Datentyp Beispiel. : bit, std_logic, vektor, usw.

⇨ Konfigurationsparameter benutzen eine eigene Schnittstellendefinition



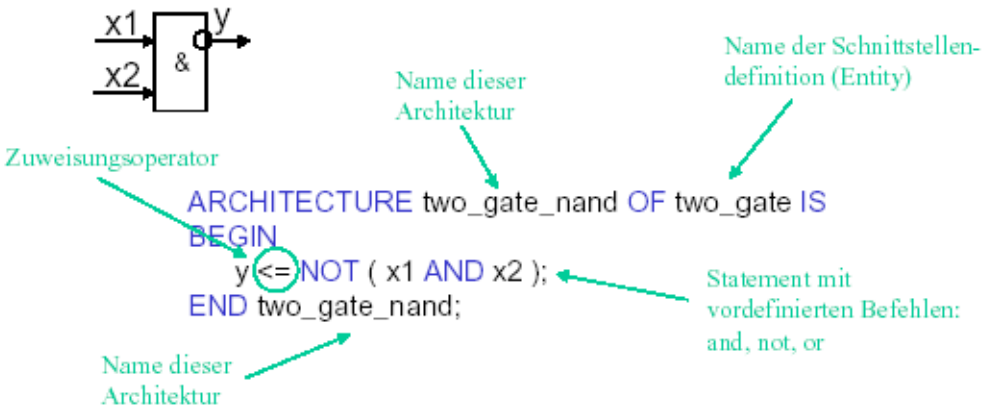


ARCHITECTURE

⇨ Die Verhalten einer Entity werden in einer Architektur beschrieben. Dabei können innerhalb der Architektur verschiedene Abstraktionsebenen verwendet werden.

⇨ Die verschiedenen Architekturen werden dann für dieselbe Entity erstellt.

Beispiel: nand Gatters basierend auf der vorhergehenden Entity Definition:



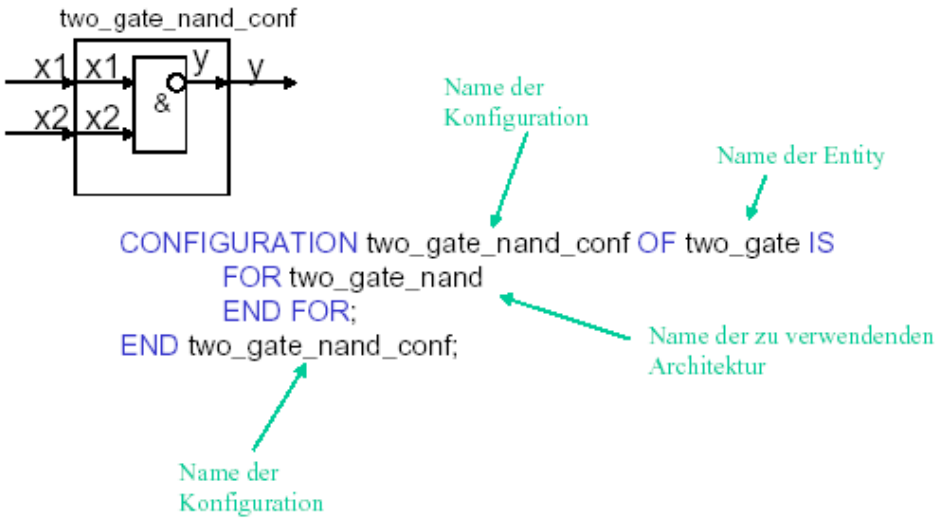
Konfiguration

⇨ Die Konfiguration bettet eine Architektur in eine Entity ein.

⇨ Durch Konfigurationen wird eine Verhaltensbeschreibung einer Entity zugeordnet.

⇨ Unterschiedliche Verhaltensbeschreibungen für eine Entity für die reine logische Simulation, mit Zeitabhängigkeiten oder zur Synthese.

→ verschiedene Ausprägungen des Verhaltens
→ Simulation unterschiedlich aufwendig und schnell
→ Schnittstellendefinition gilt für eine Klasse von Entitys
z.B.: und,oder, xor,...-Gatter mit zwei Eingängen
→ nur neues Erstellen der Verhaltensbeschreibung



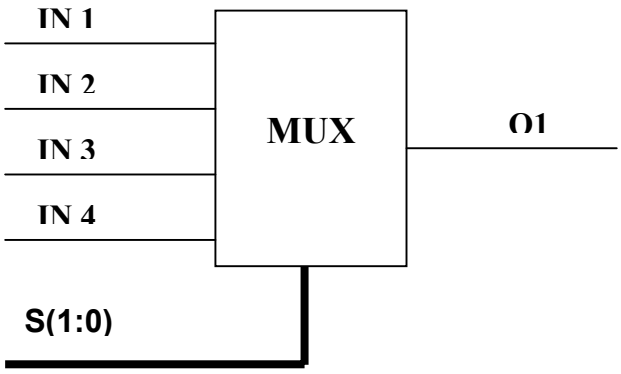
6.1.2 If-, Case-Statement bei einem Multiplexer-Baustein

Multiplexer-Baustein, Eingänge IN1, IN2, IN3, IN4, Selektor Eingänge S0, S1, Ausgang O1.

Wahrheitstabelle:

S1	S0	O1
0	0	IN1
0	1	IN2
1	0	IN3
1	1	IN4

Schaltung:




```
---------------------- e n t i t y ----------------------------------
entity mux is
        port (
                in1,in2,in3,in4  : in bit;
                s                : in bit_vector(1 downto 0);
                o1               : out bit := '0'
              );
end mux;
--------------------- a r c h i t e c t u r e ----------------------
architecture beh_mux of mux is
begin
        process (s)
        begin
                case s is
                        when "00" =>          -- select Eingänge s(1) und s(0) = "00"
                                o1 <= in1 after 15ns;
                        when "01" =>          -- select Eingänge s(1) und s(0) = "01"
                                o1 <= in2 after 15ns;
                        when "10" =>          -- select Eingänge s(1) und s(0) = "10"
                                o1 <= in3 after 15ns;
                        when "11" =>          -- select Eingänge s(1) und s(0) = "11"
                                o1 <= in4 after 15ns;
                end case;
        end process;
end beh_mux;
```

6.2 Strukturelle-Beschreibung

6.2.1 Beschreibung des „and4“-Bausteines

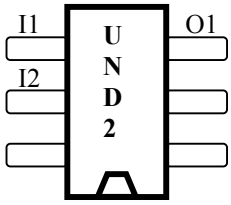
```
---------------------- e n t i t y ----------------------------------
entity and4 is
        port (
                a_i, b_i, c_i, d_i  :  in bit;
                out_o                  : out bit
                );
end and4;
---------------------- a r c h i t e c t u r e  ----------------------
architecture struk_and4 of and4 is
        signal ist1 : bit; -- einführung von internen signalen für die „verdrahtung“ --
        signal ist2 : bit;

        component and2       -- angabe des „and2“-bausteines --
                        port (
                                x_i, y_i  :  in bit;
                                z_o       :  out bit
                               );
        end component;
begin
        u1:and2   port map (x_i  => a_i , y_i   => a_i , z_o => ist1);
        u2:and2   port map (x_i  => c_i , y_i   => d_i , z_o => ist2);
        u3:and2   port map (x_i  => ist1, y_i   => ist2, z_o  => out_o);
end struk_and4;
```

6.2.2 Beschreibung von einer Kompination von 3 und Gattern

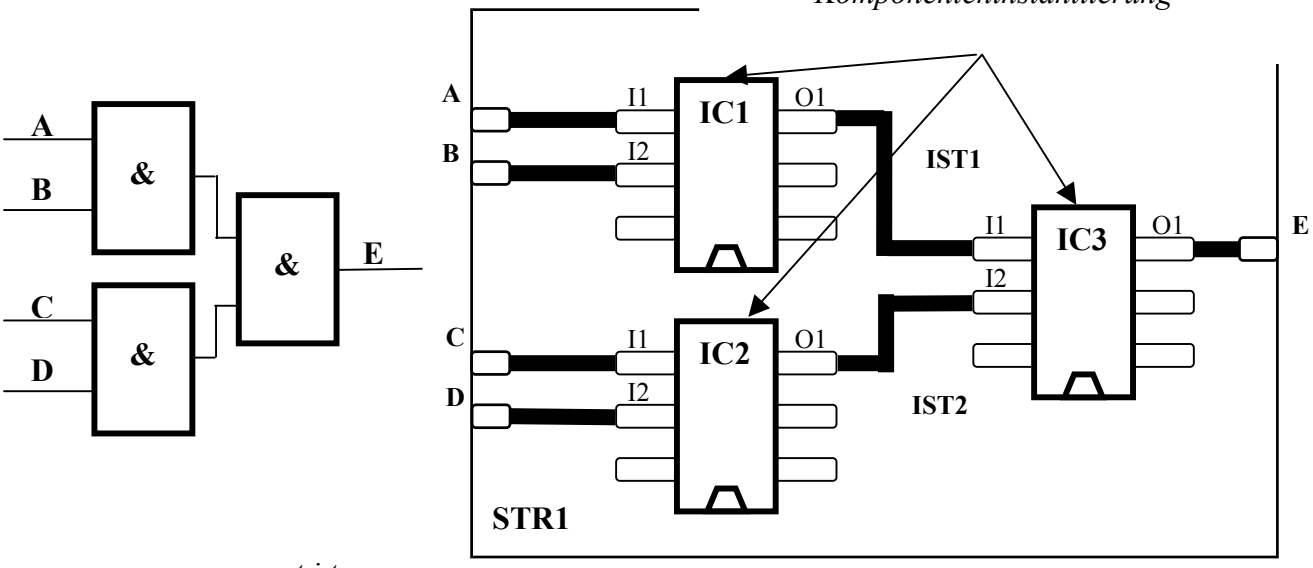
Komponentendeklaration:

component UND2



Schaltung:

Komponenteninstantiierung



---------- *e n t i t y* ----------

```
entity STR1 is
        port (
                A,B,C,D : in bit;
                E       : out bit
              );
end str1;
```

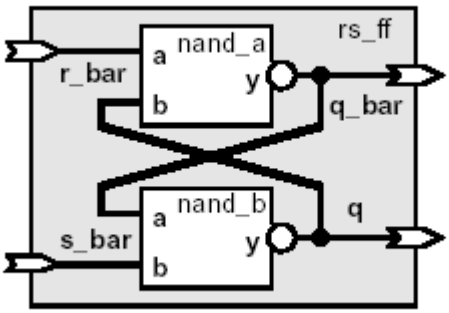
---------- *a r c h i t e c t u r e* ----------

```
architecture V1 of STR1 is
        signal IST1, IST2 : bit ;

        component UND2
                port  (
                        I1,   I2 : in bit ;
                        O1       : ou bit
                       );
        end component;
begin
        IC1 : UND2    port map ( I1 => a,  I2 => b, O1 => ist1 );
        IC2 : UND2    port map ( I1 => c,  I2 => d, O1 => ist2 );
        IC3 : UND2    port map ( I1 => IST1, I2 => IST2, O1 => e );
end V1;
```

-- Verdrahtung, z.B.: in Sockel IC3 heißen die Pins I1,I2 IST1und IST2 ....
-- Welche Komponente in welche Sockel z.B.: UND2 in IC1

6.2.3 Struktur eine RS-Flip-Flops




```
---------------------- e n t i t y ----------------------------------
entity rs_ff is
   port (
            r_bar, s_bar : in bit := '0';
            q, q_bar     : inout bit
         );
            -- ports als inout, da sie auch gelesen werden muessen
end rs_ff;
---------------------- a r c h i t e c t u r e  ----------------------
architecture structural of rs_ff is
         component nand2_socket -- komponentendeklaration
                  port (
                           a, b : in bit;
                           y    : out bit
                        );
         end component;
begin
         nand_a : nand2_socket -- komponenteninstantiierung
                  port map (
                              a => r_bar,
                              b => q,
                              y => q_bar
                            );
         nand_b : nand2_socket -- komponenteninstantiierung
                  port map (
                              a => q_bar,
                              b => s_bar,
                              y => q
                            );
end structural;
---------------------- c o n f i g u r a t i o n ----------------------
configuration rs_ff_config of rs_ff is
      for structural
            for all : nand2_socket -- komponentenkonfiguration
                  use entity work.nand2 (behavioral);
            end for;
      end for;
end rs_ff_config;
```