VHDL

Very High Speed Integrated Circuits-
Hardware **D**escription Language

Sven Marcus

26. Oktober 1994

Seminar
der
Projektgruppe Systementwurf
WS94/95 und SS95

Inhaltsverzeichnis

1 Einleitung **2**

2 Aufbau einer Beschreibung in VHDL **2**
2.1 Strukturbeschreibung 3
2.2 Algorithmische Beschreibung 4
2.3 Datenflußbeschreibung 5

3 Programmieren in VHDL **5**
3.1 Lexikalische Elemente 5
3.2 Datentypen 7
3.2.1 Objekte 8
3.2.2 Attribute 9
3.3 Sequentielle Statements 9
3.4 Unterprogramme 10
3.5 Packages 12

4 Beschreiben mit VHDL **13**
4.1 Strukturbeschreibung 13
4.2 Datenflußbeschreibung 14

5 Signalzuweisungen und Zeitverhalten **15**

1 Einleitung

VHDL ist eine Hardware-Beschreibungssprache, die im Rahmen des VHSIC[1] Programmes der USA entwickelt wurde. Ziel der Entwicklung war, eine möglichst weit standardisierte Sprache für die Beschreibung von Hardware-Komponenten bereitzustellen. Diese sollte einerseits möglichst mächtig und universell sein, und andererseits eine strukturierte Beschreibung und gute Lesbarkeit des Quellcodes ermöglichen. VHDL ist auf Grundlage der Programmiersprache ADA entwickelt worden. Im Jahre 1987 wurde VHDL von der IEEE standardisiert.

In Kapitel 2 wird der grobe Aufbau von Beschreibungen in VHDL erläutert. Kapitel 3 geht auf die Elemente der algorithmischen Beschreibung ein, die ein Programmieren, änlich zu höheren Programmiersprachen, erlauben. In Kapitel 4 werden die HDL-spezifischen Beschreibungsformen (Strukturbeschreibung, Datenflußbeschreibung) behandelt. Zum Abschluß werden in Kapitel 5 noch das Zeitverhalten in VHDL und die damit verbundenen Besonderheiten bei der Signalzuweisung erläutert.

2 Aufbau einer Beschreibung in VHDL

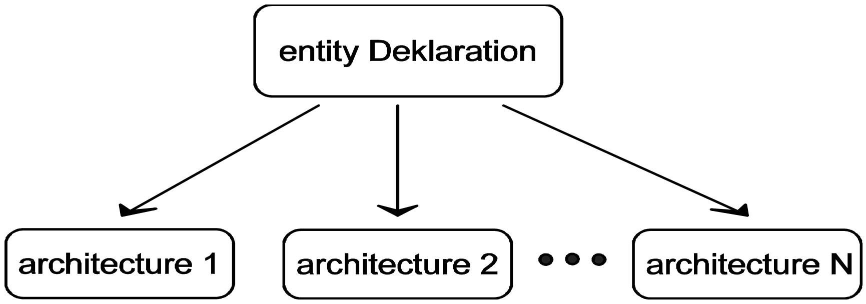

Abbildung 1: allgemeiner Aufbau einer VHDL-Beschreibung

Eine VHDL-Beschreibung besteht jeweils aus *genau einer* Entity-Deklaration und *mindestens einer* dazugehörigen Architecture. Die Entity-Deklaration bestimmt das externe Erscheinungsbild des Bausteins, während die Architecture-Implementierung die interne Realisierung beschreibt.

Am Beispiel eines Volladdierers sollen im folgenden die Grundstrukturen der einzelnen Komponenten erläutert werden.

Entity
Die Entity-Deklaration legt das äußere Erscheinungsbild der „Black Box“ mit Ein- und Ausgängen sowie „generic“-Konstanten fest.

[1]VHSIC = Very High Speed Integrated Circuits

Beispiel: Volladdierer

Abbildung 2: Volladdierer als Black-Box

```
entity Volladdierer is

    port(a, b, c_in:    in bit;     -- Eingänge
         sum, c_out:  out bit);  -- Ausgänge

end Volladdierer;
```

Architecture

Eine Architecture beschreibt eine interne Realisierung des Bausteins.

Es gibt drei Beschreibungsarten:

1. Strukturbeschreibung (structural)
2. Datenflußbeschreibung (dataflow)
3. Algorithmische Beschreibung (behavioral)

2.1 Strukturbeschreibung

Die Strukturbeschreibung beschreibt, wie ein Bauteil aus einzelnen Unter-Bausteinen zusammengesetzt ist. Hierdurch ergibt sich die Möglichkeit eines hierarchischen Aufbaus einer Hardware-Beschreibung in VHDL.

Beispiel: Volladdierer

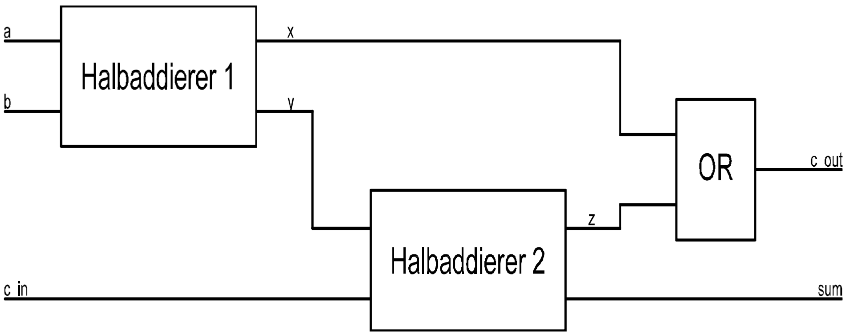

Abbildung 3: Struktur eines Volladdierers

```
architecture Struktur of Volladdierer is

component Halbaddierer
        port(  i1, i2:   in bit;
               carry:    out bit;
               sum:      out bit);
end component;
component Or_Gatter
        port(  i1, i2:   in bit;
               o:        out bit);
end component
        signal x, y, z: bit;    -- interne Leitungen zwischen den einzelnen Bausteinen
begin
        Baustein1: Halbaddierer port map (a, b, x, y);
        Baustein2: Halbaddierer port map (y, c_in, z, sum);
        Baustein3: Or_Gatter port map (x, z, c_out);
end Struktur;
```

Das Verhalten des Gesamtbauteils ergibt sich aus dem Verhalten der einzelnen Bausteine (component), aus denen es zusammengesetzt ist und der internen Verdrahtung dieser Bausteine. Dies geht natürlich nur bis zu einer bestimmten Ebene herab, da es in VHDL keine vorgefertigten Bauteile gibt. Hier muß eine der anderen Beschreibungsformen gewählt werden.

2.2 Algorithmische Beschreibung

Die algorithmische Beschreibung wird benutzt um das Verhalten eines Bauteils zu beschreiben. Dies geschieht ähnlich zu höheren Programmiersprachen. Kennzeichnend für die algorithmische Beschreibung ist das Schlüsselwort „process“.

Beispiel: Volladdierer

```
architecture Verhalten of Volladdierer is begin
    process
            variable i:                integer;     -- Index-Variable
            constant sum_vektor:   bit_vector (0 to 3) := "0101";
            constant carry_vektor: bit_vector (0 to 3) := "0011";
    begin
            i :=0;
            if a = '1' then i := i+1; end if;
            if b = '1' then i := i+1; end if;
            if c_in = '1' then i := i+1; end if;
            sum <= sum_vektor(i) after 20 ns;
            c_out <= carry_vektor(i) after 20 ns;
            wait on a, b, c_in;     -- Prozeß wartet auf ein Ereignis von a, b, c_in
    end process;
end Verhalten;
```

Ein Prozeß besteht aus mehreren Anweisungen, die sequentiell abgearbeitet werden. Prozeße werden gestartet und laufen dann endlos weiter. Deshalb muß in jedem Prozeß mindestens ein „Wait-Statement" vorhanden sein. Mit Hilfe von „Wait-Statements" werden die verschiedenen Prozeße untereinander synchronisiert.

2.3 Datenflußbeschreibung

Die Datenflußbeschreibung wird benutzt, um den Datenfluß zwischen Signalen zu beschreiben. Hierzu stehen die logischen Verknüpfungen and, or, nand, nor, xor und not zur Verfügung.

Beispiel: Volladdierer

```
architecture Datenfluss of Volladdierer is
signal s: bit;
begin
    s <= a xor b after 5 ns;
    sum <= s xor c_in after 5 ns;
    c_out <= (a and b) or (s and c_in) after 10 ns;
end Datenfluss;
```

Hierbei werden alle Anweisungen parallel ausgeführt. Jede einzelne Anweisung entspricht dabei implizit einem eigenständigen Prozeß.

Beispiel:

```
s <= a xor b after 5 ns;
```

entspricht

```
process
begin

    s <= a xor b after 5 ns;
    wait on a, b;

end process;
```

3 Programmieren in VHDL

3.1 Lexikalische Elemente

Kommentare

Kommentare beginnen mit zwei Minuszeichen '−−' und gehen bis zum Zeilenende.

Bezeichner

Bezeichner müssen folgender Form entsprechen:

identifier ::= letter { [underline] letter_or_digit}

Zahlen

Zahlen können als Dezimalzahl, oder mit Basis von 2 bis 16 angegeben werden.
Dezimalzahlen:

decimal_literal ::= integer [. integer] [exponent]
integer ::= digit { [underline] digit}
exponent ::= E [+] integer | E - integer
Beispiele:

```
0     5     23_65       67E-3     -- integer Literale
0.0   5.4   34.43_43    23.6E-6   -- real Literale
```

Zahlen mit Basis:

based_literal ::= base # based_integer [. based_integer]#[exponent]
base ::= integer
based_integer ::= extended_digit { [underline] extended_digit }
extended_digit ::= digit | letter
Beispiele:

```
2# 1100_0100#     16#C4#     4#301#E1     -- integer 196
2#1.1111_1111_11#E+11        16#F.FF#E2   -- real 4095.0
```

Zeichen (character)

Zeichen werden als ASCII-Zeichen in Hochkommata dargestellt.
Beispiele:

```
'a'   ' '   '*'   '''
```

Zeichenketten (string)

Zeichenketten bestehen aus mehreren Zeichen in Anführungszeichen.
Beispiele:

```
"Dies ist eine Zeichenkette"
"Er sagte: ""Hallo""."     -- so werden Anführungszeichen in eine Zeichenkette gesetzt
""                         -- leere Zeichenkette
```

Bit Strings

Bit Strings haben folgende Form:

bit_string_literal ::= base_specifier "bit_value"
base_specifier ::= B | O | X
bit_value ::= extended_digit { [underline] extended_digit}

B gibt Basis 2 (binär), O Basis 8 (octal) und X Basis 16 (hexadezimal) an. Die Basen stellen nur eine Vereinfachung für die Eingabe dar.
Beispiele:

```
B"1010"   O"14"   X"C"   -- stellen alle den gleichen Bit-String dar.
```

3.2 Datentypen

Integer Typen
Integer Typen sind eine Teilmenge der ganzen Zahlen.
VHDL unterscheidet hierbei zwischen auf- und absteigenden Typen.
Beispiel:

```
type hoch is range 0 to 20
type runter is range 20 downto 0
```

Der vordefinierte Typ integer ist *mindestens*:

```
type integer is range -2147483647 to +2147483647
```

Physikalische Typen
Physikalische Typen sind numerische Typen, bei denen die Maßeinheit mit angegeben wird. Außerdem sind Abstufungen der Einheit möglich.
Beispiel: der vordefinierte Typ time

```
type time is range Implementationsabhängig

    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;
```

Fließkomma Typen
Fließkommatypen sind eine Teilmenge der rellen Zahlen. Genauigkeit und maximaler Zahlenbereich sind dabei Implementationsabhängig. Der maximale Zahlenberich ist mindestens -1E38 bis +1E38.
Beispiel: vordefinierter Typ real (Minimalgröße)

```
type real is range -1E38 to +1E38
```

Aufzählungstypen
Aufzählungstypen sind Typen, die aus geordneten Literalen oder Zeichen (character) bestehen.
Beispiele: vordefinierte Typen

```
type boolean is (false, true);
type bit is ('0', '1');
type severity_level is (note, warning, error, failure);

type character is ( NUL,  SOH,  STX,  ETX,  ...
                    ...,  'A',  'B',  'C',  ...
                    ...,  '|',  '}',  ' ',  DEL);
```

Untertypen
Untertypen schränken den Wertebereich eines Typen ein, übernehmen aber ansonsten seine Eigenschaften.
Beispiele: vordefinierte Typen

```
subtype natural is integer range 0 to integer'high;
subtype positive is integer range 1 to integer'high;
```

Verbundtypen
Es gibt die Verbundtypen **array** und **record**. Diese sind wie bei höheren Programmiersprachen definiert.
Beispiele:

```
type matrix is array range (1 to 5, 1 to 5) of real;
-- jetzt kommen die zwei vordefinierten array Typen:
type string is array (positive range <>) of character;
type bit_vector is array (natural range <>) of bit;

type complex is
    record
        re: real;
        im: real;
    end record;
```

3.2.1 Objekte

Objekte sind benannte Container, die einen Wert repräsentieren. Es gibt drei Arten von Objekten:

Konstanten
Konstanten werden mit einem Wert initialisiert und können danach nur noch gelesen werden. Deklarierbar sind sie in paralleler und sequentieller Umgebung.
Beispiele:

```
constant pi:    real     := 3.141592;
constant lotto: positive := 13_983_816;
```

Variablen
Variablen können initialisiert, gelesen und geschrieben werden. Sie sind nur in sequentieller Umgebung (Prozeß) deklarierbar. Zuweisungen an Variablen erfolgen mit „:=“.
Beispiele:

```
variable bus16: bit_vector(0 to15);
variable count: natural := 10;
```

Signale
Signale können ebenfalls initialisiert, gelesen und auch geschrieben werden. Sie sind nicht in sequentieller Umgebung deklarierbar. Signale besitzen eine zeitliche Vergangenheit und Zukunft. Zuweisungen an Signale erfolgen mit „<=“.
Beispiele:

```
signal S1, S2: bit := '1';
signal bus8:   bit_vector(0 to 7);
```

3.2.2 Attribute

Attribute liefern zusätzliche Informationen zu Daten.

Vordefinierte Attribute ändern ihren Wert automatisch.
Beispiele für vordefinierte Attribute:

Attribut	**Resultat**
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'range	Bereich von T
T'length	Länge von T
T'event	Ist dann 1, wenn T seinen Wert ändert (Ereignis)

Benutzerdefinierte Attribute werden explizit durch VHDL Statements geändert.

3.3 Sequentielle Statements

In sequentiellen Prozeßen bietet VHDL die Möglichkeit „Programme“ zu schreiben, die eine Mächtigkeit besitzen, ähnlich wie höhere Programmiersprachen.

Wait Statement

Bei den *Wait Statements* handelt es sich um eine Besonderheit bei VHDL. Sie werden benötigt, damit die einzelnen Prozeße nicht endlos laufen und untereinander synchronisiert werden können. In jedem Prozeß muß daher mindestens ein *Wait Statement* vorkommen.
Beispiele:

```
wait;                  -- wartet für immer
wait on s1,...,sN;     -- wartet auf ein Ereignis bei einem der Signale
wait until cond;       -- wartet bis die Bedingung cond wahr wird
wait on s1,...,sN      -- wartet auf ein Ereignis,
     until cond;         -- testet danach die Bedingung
wait for time;         -- wartet für eine Dauer der Länge „time“
wait on s1,...,sN      -- wartet auf ein Ereignis,
     until cond          -- testet dann die Bedingung
     for time;           -- wartet aber maximal „time“ lang
```

If Statement

Erlaubt eine Verzweigung, in Abhängigkeit von einer Bedingung. Beispiele:

```
if A='1' then B := '0';

if A=1 then Tag := Montag;
        else if A=2 then Tag := Dienstag;
             else Tag := Anders;
                  B := '0';
end if;
```

Case Statement

Das Case Statement erlaubt eine Auswahl aus mehreren Möglichkeiten. Hiermit können unter anderem if-Kaskaden unterbunden werden.
Beispiele:

```
case A of
    when '1' => Tag := Montag;
    when '2' => Tag := Dienstag;
    when others => Tag := Anders; B := '0';
end case;
```

Wichtig bei Case Statements ist, daß alle Möglichkeiten abgedeckt werden. Hierzu steht das Konstrukt „others“ zur Verfügung, welches alle noch nicht behandelten Fälle abdeckt.

Loop Statements

Mit loop kann man Schleifen programmieren. Es gibt drei Arten von Schleifen: „endlos“-, for- und while-Schleifen.
Durch die Verwendung von „next“ wird die nächste Iteration vorzeitig begonnen, und mit „exit“ wird die Schleife vorzeitig verlassen.
Beispiele:

```
loop
    z := z + 1;
    exit when z > 42;
end loop;

while i > 99 loop
    i := i - 1;
end loop;

for i in -100 to 100 loop
    next when i = 0;
    z := 1/i;
end loop;
```

Null Statement

Befehl, der explizit aussagt, daß nichts gemacht werden soll. Dieser Befehl mag bei ersten Überlegungen sinnlos erscheinen, aber er erfüllt, z.B. in case Verzweigungen, durchaus seinen Zweck.

```
case ...
    when others => null;
end case;
```

3.4 Unterprogramme

Es gibt zwei Arten von Unterprogrammen. *Procedure* und *function*. Sie unterscheiden sich darin, daß Funktionen mit Rückgabetyp sind, und Prozeduren ohne. Es gibt jeweils den Deklarationsteil und den Unterprogrammkörper.

Funktion

Deklaration:

```
function Name [(formale_Parameter)] return typ;
```

Körper:

```
function Name [(formale_Parameter)] return typ
     is
     Deklarationen
begin
     sequentielle Statements
     return Ausdruck;
     [sequentielle Statements]
end [Name];
```

Prozedur

Deklaration:

```
procedure Name [(formale_Parameter)];
```

Körper:

```
procedure Name [(formale_Parameter)]
     is
     Deklarationen
begin
     sequentielle Statements
end [Name];
```

Parameter

Als Parameter können Konstanten, Variablen und Signale übergeben werden. Es wird jeweils zwischen Eingabe und Ausgabetypen unterschieden.

constant: Entspricht einem Ausdruck im aufrufenden Programm. Ein constant Parameter kann daher nur gelesen werden, muß also vom Typ in sein.

signal: Ein signal Parameter muß einem Signal entsprechen. Er darf also, je nach Typ, gelesen und geschrieben werden.

variable: Variable Parameter müßen dementsprechend einer Variablen im aufrufenden Programm entsprechen. Die Beschränkungen bezüglich Lesbarkeit, bzw. Schreibbarkeit ergeben sich nur aus dem Typ.

Typ	**Lesen**	**Schreiben**
in	ja	nein
out	nein	ja
inout	ja	ja

Unterprogrammaufruf

Der Unterprogrammaufruf entspricht quasi dem Einsetzen des Unterprogrammes an der Stelle des Aufrufes. Beim Unterprogrammaufruf können Parameter weggelassen werden. Hier werden dann die default-Werte eingesetzt. Wenn kein default-Wert explizit angegeben wird, ist der default-Wert automatisch Null.

Überladung

Mehrere Unterprogramme können denselben Namen besitzen, sofern sie sich im Parameter- und Return-Typ-Profil unterscheiden.
Beispiel:

```
function bitvalue (s: boolean) return bit is
begin
     return bit'val(boolean'pos(s));
end;

function bitvalue (x:natural) return bit is
begin
     return bit'val(x mod 2);
end;
```

3.5 Packages

Mit Packages bietet VHDL die Möglichkeit, häufig benutzte Unterprogramme und Typen zusammenzufassen. So sind beispielsweise alle vordefinierten Datentypen im package STANDARD definiert.

Beispiel:

```
package BitZwei is
    type ZweiBit is array (1 downto 0) of bit;
    function "+"(a, b: ZweiBit) return ZweiBit;
end BitZwei;

package body BitZwei is
    function "+"(a, b: ZweiBit) return ZweiBit is
        variable Ergebnis : natural := 0;
        if a(1) = '1' then Ergebnis := Ergebnis + 2;
        if a(0) = '1' then Ergebnis := Ergebnis + 1;
        if b(1) = '1' then Ergebnis := Ergebnis + 2;
        if b(0) = '1' then Ergebnis := Ergebnis + 1;
        Ergbnis := Ergebnis mod 4;
        case Ergebnis of
            when 0 => return '00';
            when 1 => return '01';
            when 2 => return '10';
            when others => return '11'
    end "+";
```

4 Beschreiben mit VHDL

4.1 Strukturbeschreibung

Entsprechend zu den Gliederungsmöglichkeiten in der algorithmischen Beschreibungsform, gibt es auch in der Strukturbeschreibung Möglichkeiten zur Gliederung.

Block

Ein Block ist die kleinste Zusammenfassungsmöglichkeit.

```
block [(guard_expression)]
        block_header
        block_declarative_part
begin
        concurrent_statements
end block;
```

Die einfache Form des Blocks dient hierbei nur der Zusammenfassung von einzelnen Anweisungen, um die Beschreibung übersichtlicher zu gestalten. Mit „guard“ kann man den Zugang mehrerer Quellen zu einem Kanal regeln.

Component

Über den Befehl component kann man auf schon fertiggestellte entitys zurückgreifen. Beispiel: bei gegebener entity nand3

Deklaration:

```
component nand3
      generic (Tpd: Time := 1 ns);
      port (a, b, c: in logic_level; y: out logic_level);
end component;
```

Einrichtung:

```
enable_gate: nand3
      port map (a => en1, b => en2, c => int_req, y => interrupt);
```

Entity

Eine Entity beschreibt ein Bauteil. In der Beschreibung wird das äußere Erscheinungsbild angegeben. Hierzu gehören die Ports, welche vom Typ in, out oder inout sein können, sowie die „generic“ Konstanten für dieses Bauteil.

Architecture

Eine Architecture enthält eine Realisierung des Bauteils zu der jeweiligen Entity-Deklaration. Zu einer einzelnen Entity-Deklaration können mehrere verschiedene Architectures existieren. Die Realisierung kann in der algorithmischen Beschreibung, der Strukturbeschreibung oder der Datenflußbeschreibung vorgenommen werden. Auch Mischformen aus diesen Dreien sind möglich.

Configuration

In der Configuration kann man die Components einer Entity extern zusammenfassen. Dies erhöht zwar den Arbeitsaufwand und den Quellcode, aber dieser längere Quellcode ist, vor allem bei großen Beschreibungen, besser lesbar und besser zu Warten.

Libraries

Man kann seine Arbeit in Libraris zusammenfassen. Hierzu muß vor jede Instanz (entity, architecture, package, package body, configuration)

```
library Name;
```

gestellt werden.
Implizit werden jeweils zwei Libraries benutzt. Library STD enthält alle vordefinierten Typen, und Library WORK enthält alle aktuellen Instanzen der jeweiligen Arbeitssitzung.

Zum Zugriff auf einzelne Komponenten wird die Referenzierungsschreibweise angewandt.

Z.B.: `library1.package2.type3`

Use Klausel

Mit der Use Klausel kann man auf einzelne Komponenten zugreifen, ohne jedesmal referenzieren zu müssen. Z.B. wird standardmäßig

```
use STD.STANDARD.all
```

benutzt. Hierdurch sind alle vordefinierten Datentypen erreichbar

z.B.: `i: integer` statt `i: STD.STANDARD.integer`

Sollten durch verschiedene mit „use“ angemeldete Libraries Konflikte auftreten, muß doch wieder explizit referenziert werden.

4.2 Datenflußbeschreibung

In der Datenflußbeschreibung lassen sich die Signale mit Hilfe der mathematischen Operatoren berechnen. Auf den zweiwertigen Typen bit , bzw. boolean arbeiten die Operatoren and, or, nand, nor, xor und not .

Mit diesen Operatoren lassen sich alle Aussagenlogischen Formeln berechnen.

Beispiel: $y = (\overline{a} + b * c) * (a * (\overline{b} + c))$

```
entity Formel is
    port( a, b, c:    in bit;
          y:          out bit);
end Formel;

architecture dataflow of formel is
begin
     y <= (not a or b and c) and (a and (not b or c));
end dataflow;
```

5 Signalzuweisungen und Zeitverhalten

In VHDL gibt es keine sofortigen Signalzuweisungen. Eine Signalzuweisung hat daher immer die folgende Form:

```
s <= x [after time];
```

Auch ohne explizite Angabe einer Zeitspanne wird implizit „after 0 ns“ eingesetzt. VHDL plant also nur eine Signalzuweisung ein. Hierzu existiert zu jedem Signal ein Treiber, der diese geplanten Zuweisungen speichert.

Simulationszyklus
VHDL führt immer erst alle Prozeße bis zu einem Wait Statement durch. Danach wird die Zeit aktualisiert, und erst danach werden die Signalzuweisungen durchgeführt. Dann werden wieder alle aktiven Prozeße gestartet und der Simulationszyklus beginnt von neuem.

Inertial Delay
Durch dieses Simulationsverhalten können bereits eingeplante Signalzuweisungen von neuen Signalzuweisungen überschrieben werden.
Beispiel:

```
process
begin
        s2 <= s after 7 ns;
        wait on s;
end process;
```

Diesen Prozeß kann man sich als einen Kanal vorstellen, der das Eingangssignal s nach 7 ns an seinem Ausgang s2 bereitstellt. Dieser Kanal überträgt keine Impulse, die kürzer als 7 ns sind. Wenn das Signal s vorher seinen Wert ändert, wird die eingeplante Zuweisung im Treiber von der neuen überschrieben.

Transport Delay
Durch das reservierte Wort „transport“ kann man erreichen, daß auch kurze Signale übertragen werden. So müßte man im obigen Beispiel die Zuweisung wie folgt fomulieren:

```
s2 <= transport s after 7 ns;
```

Hierdurch werden alle eingeplanten Zuweisungen im Treiber gespeichert und mit der gewünschten Verzögerung ausgeführt.

Resolved Signals
Da in VHDL normalerweise nur eine Quelle pro Signal zulässig ist, muß man sogenannte „Resolution-Funktionen“ einführen, um mehreren Quellen Zugriff zu gewähren. Eine „Resolution-Funktion“ wählt aus den Eingabe-Quellen auf beliebeige Art aus, welchen Wert das Signal erhalten soll.

Beispiel:

```
-- Funktion, die eine '1' sendet, sofern mindestens eine Quelle eine '1' schreiben will

function Resolution (Quellen: bit_vector) return bit is
    variable Ergebnis: Bit := 0;
begin
    for i in Quellen'left to Quellen'right loop
          Ergebnis := Ergebnis or Quellen(i);
    end loop;
    return Ergebnis;
end Resolution;
```

Wichtig ist, daß die Anzahl der möglichen Quellen nicht beschränkt ist, da nicht vorausgesehen werden kann, wieviele Quellen gleichzeitig schreiben wollen. Deshalb muß auch die Schleife in Abhängigkeit von den jeweiligen Werten der Attribute durchlaufen werden.
Hiermit kann dann ein Subtype deklariert werden, der die Resolution-Funktion benutzt.

```
subtype Resolved_Bit is Resolution bit;
```

Guarded Signals
Mit Guards kann man einerseits mehreren Quellen Zugang zu einem Signal erlauben, wie auch mit Resolution-Funktionen allein, aber andererseits auch exklusiven Zugang gewährleisten. Es gibt zwei Arten von guarded signals: bus und register.
Deklaration:

```
signal b: Resolution bus;
signal r: Resolution register;
```

Sie unterscheiden sich hauptsächlich darin, daß ein Register seinen Wert behält, während ein Bus danach nicht mehr gelesen werden kann, also quasi einen undefinierten Wert annimmt.