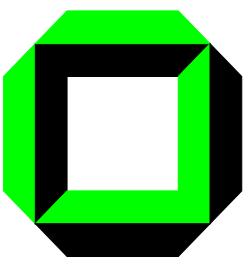
Einführung in VHDL



Universität Karlsruhe
Institut für Rechnerentwurf und Fehlertoleranz
(Prof. D. Schmid)
Geb. 20.20

Igor Katschan, Frank Mayer Tel.: 608-4354/ -7409
email: {igor, mayer}@ira.uka.de

1 Einleitung

VHDL (Very high speed integrated circuits Hardware Description Language) ist eine Sprache, mit der digitale elektronische Systeme beschrieben werden können. Sie erfüllt mehrere für den Entwurf notwendige Anforderungen. Zum einen läßt sich in VHDL ein Entwurf strukturell beschreiben, d.h. ein System läßt sich in hierarchisch gegliederte Untermodule aufspalten. Weiterhin läßt sich die Funktion einer Schaltung in Form einer Verhaltensbeschreibung mithilfe von aus anderen Programmiersprachen bekannten Anweisungen angeben. Schließlich ist es möglich, einen kompletten Entwurf zu simulieren, um Fehler schon vor der Herstellung beseitigen und ebenso alternative Entwürfe testen und miteinander vergleichen zu können. Der Unterschied zur Simulation sequentieller Programmiersprachen wie Pascal oder C liegt darin, daß die Auswirkung einer Anweisung nicht unmittelbar nach ihrer Ausführung, sondern bezogen auf einen Zeitmaßstab ausgegeben werden. Dadurch ist es möglich, parallele Ereignisse, wie sie bei digitalen Schaltungen üblich sind, mit sequentiellen Techniken zu beschreiben.
Die folgenden Kapitel geben eine kurze Einführung in VHDL. Sie konzentriert sich dabei auf diejenigen Aspekte, die zur Bewältigung der Praktikumsaufgaben benötigt werden. Sie ist bei weitem nicht vollständig, sondern gibt vieles nur exemplarisch wieder, anderes wird überhaupt nicht beschrieben. Ausführliche Literatur ist am Ende dieses Kapitels angegeben.

1.1 Strukturbeschreibung

Jedes elektronische System kommuniziert mit seiner Umgebung über eine Schnittstelle (interface). Bei einem Chip sind das zum Beispiel die einzelnen Pins. Ein System kann aus mehreren Modulen zusammengesetzt sein. Im allgemeinen betrachtet man ein Modul als “black box”, die über sogenannte Port-Signale mit anderen Modulen verbunden ist. Ein solches Modul wird in VHDL als *entity* bezeichnet. Module können selbst wieder aus mehreren Untermodulen zusammengesetzt sein. Dadurch ist es möglich Hierarchien mit beliebig vielen Hierarchiestufen aufzubauen.

1.2 Verhaltensbeschreibung

Sind Module nicht weiter untergliedert, muß die Schaltungsfunktion, d.h. die Architektur der Schaltung, durch eine sogenannte Verhaltensbeschreibung definiert werden. In VHDL werden dazu Programmkonstrukte verwendet, die denen in üblichen Programmiersprachen in vielen Punkten sehr ähnlich sind. Als Beispiel ein Ausschnitt aus einem beliebigen Programm:

```
        ...
y   := not(a) and b;
i   := 3*b + (k-c)/e;
        ...
```

1.3 Das Zeitmodell diskreter Ereignisse

Ein komplett beschriebenes Modul kann simuliert werden. Dazu wird die Verhaltensbeschreibung von einem Simulator ausgeführt.
Zu Beginn der Simulation wird allen Signalen in einer Initialisierungsphase ein Startwert zugewiesen und die Simulationszeit zu Null gesetzt. Danach werden die Vehaltensprogramme der Module ausgeführt, wodurch in der Regel im weiteren Verlauf Signaländerungen an den Ausgängen auftreten.
Sobald eine Signaländerung auftritt, wird die Simulationszeit aktualisiert und alle Signaländerungen, die gleichzeitig stattgefunden haben, registriert. Dann werden die Verhaltensprogramme all derer Module ausgeführt, die auf diese Signaländerungen reagieren, wodurch in der

Regel neue Signaländerungen verursacht werden. Sind alle Programme beendet, wartet der Simulator auf die nächste Signaländerung und startet dann einen neuen Simulationszyklus.
Durch die Zuweisung eines Simulationszyklus' an einen diskreten Simulationszeitpunkt, geschehen für den Anwender alle Signaländerungen, die durch die sequentiellen Anweisungen der Verhaltensprogramme verursacht wurden und in diesem Zyklus stattfinden, gleichzeitig.

2 VHDL als Programmiersprache

Wie schon erwähnt, kann das Verhalten eines Moduls als Programm ähnlich wie in Pascal oder C beschrieben werden. Im folgenden wird der Aufbau der Sprache kurz erläutert. Bei den Syntaxbeschreibungen bedeutet:

::=	:	Definition
{...}	:	Inhalt der Klammer kann beliebig oft hintereinander angefügt werden (auch keinmal)
[...]	:	Inhalt der Klammer kann maximal einmal angefügt werden.
|	:	Alternative Beschreibungsmöglichkeit

2.1 Lexikalische Elemente

2.1.1 Kommentare

Kommentare beginnen in VHDL mit zwei aufeinanderfolgenden Bindestrichen (‘--’) und reichen bis zum Ende der Zeile. Kommentare haben keinen Einfluß auf das Verhalten eines Moduls. Beispiel:

```
a := 3*b;          --  dies ist ein Kommentar
```

2.1.2 Namen (identifier)

Namen sind reservierte Worte und benutzerdefinierte Namen. Namen müssen mit einem Buchstaben beginnen, dem beliebig viele Buchstaben und Ziffern folgen dürfen, die wiederum mit Unterstreichungszeichen (‘_’) getrennt werden können. Groß- und Kleinschreibung werden dabei nicht unterschieden (NAME und name sind also identische Namen):

```
identifier ::= letter { [ underline ] letter | digit}
```

Beispiele:

```
xyz,   Ein_Name,     Zeichen_3_a45
```

2.1.3 Zahlen

Literale Zahlen können dezimale Zahlen darstellen oder Zahlen mit Basen zwischen zwei und sechzehn. Enthält das Literal einen Punkt oder ist der Exponent negativ, ist es eine Realzahl, sonst eine ganze Zahl (Integer).
Dezimale Zahlen sind wie folgt definiert:

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

Beispiele:

```
0      1     123_456_789        987E6       -- integer
0.0    1.5   3.141_6            1.6E-19     -- Realzahlen
```

Führende Nullen sind erlaubt. Eine Zahl darf keine Leerzeichen beeinhalten.

Literale Zahlen mit Basen sind wie folgt definiert:

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

Basis und Exponent werden als Dezimalzahl interpretiert. Die Großbuchstaben ‘A’ bis ‘F’ entsprechen den Zahlen 10 bis 15. Beispiele:

```
2#1100_0100#    16#C4#     4#301#E1          --  entspricht 196
2#1.1111_1111_111#E+11  16#F.FF#E2        -- entspricht 4095.0
```

2.1.4 Zeichen (Characters)

Literale Zeichen sind ASCII-Zeichen eingeschlossen in einfache Anführungszeichen. Beispiele:

‘A’ ‘*’ ‘’’ ‘ ‘

2.1.5 Zeichenketten (Strings)

Literale Zeichenketten sind Folgen von Zeichen, die durch doppelte Anführungszeichen umrahmt werden.

string_literal ::= “ { graphic_character } “

Um ein doppeltes Anführungszeichen innerhalb einer Zeichenkette anzugeben, müssen zwei dieser Anführungszeichen hintereinandergeschrieben werden (diese zählen dann als ein einzelnes Zeichen). *Eine Zeichenkette kann einem Array von Zeichen als Wert zugewiesen werden*. Beispiele:

```
“Eine Zeichenkette”
“”                      -- leere Zeichenkette
“ab””c”                 -- Doppeltes Anführungszeichen in Zeichenketten
```

2.2 Datentypen und Objekte

VHDL unterscheidet sogenannte skalare Typen wie Integer-Typen und Aufzählungs- (Enumeration-)Typen und sogenannte zusammengesetzte Typen wie Array-Typen. Zusammengesetzte Typen beinhalten selbst Elemente, die wieder von einem zusammengesetzten oder skalaren Typ sind. Skalare Typen sind dagegen nicht weiter unterteilbar. Eine Typdefinition sieht in VHDL wie folgt aus:

full_type_declaration ::= **type** identifier **is** type_definition ;

Type_definition kann je nach Typ unterschiedlich sein. Im folgenden werden nur die Type_definition von einigen Integer-, Aufzählungs- und Arraytypen betrachtet.

2.2.1 Integer Typen

Ein Integertyp ist ein Bereich von Integerzahlen innerhalb spezifizierter Bereichsgrenzen:

integer_type_definition ::= **range** range
range ::= simple_expression direction simple_expression
direction ::= **to** | **downto**

Simple_expression ist dabei irgendein Integerwert oder -ausdruck wie 12 oder (3*5+6). Die Schlüsselworte **to** und **downto** kennzeichnen dabei sogenannte *ansteigende* oder *abfallende* Typen. Beispiele:

type byte_int **is range** 0 **to** 255;
type bit_index is **range** 31 **downto** 0;

2.2.2 Aufzählungstypen

Ein Aufzählungstyp ist eine geordnete Menge von Namen oder Zeichen. Die Elemente innerhalb eines Aufzählungstyps müssen sich voneinander unterscheiden:

enumeration_type_definition ::= (enumeration_literal { , enumeration_literal })
enumeration_literal ::= identifier | character_literal

Beispiele:

type boolean is (false, true);
type bit is ('0', '1');
type octal_digit is ('0', '1', '2', '3', '4', '5', '6', '7');

2.2.3 Arraytypen

Arraytypen sind indizierte Mengen von Elementen des gleichen Typs. Arrays können ein- und mehrdimensional sein. Die Indexbereiche von Arrays können beschränkt und unbeschränkt sein. Der Indexbereich von unbeschränkten Arrays wird erst dann festgelegt, wenn der Arraytyp tatsächlich gebraucht wird.

array_type_definition ::= unconstrained_array_definition | constrained_array_definition
unconstrained_array_definition ::=
array (index_subtype_definition { , index_subtype_definition })
of *element*_subtype_indication
constrained_array_definition ::= **array** index_constraint **of** element_subtype_indication
index_subtype_definition ::= type_mark **range** <>
index_constraint ::= (discrete_range { , discrete_range })
discrete_range ::= *discrete*_subtype_indication | range

Beispiele für beschränkte Arrays:

type word **is array** (31 **down to** 0) **of** bit;
type memory **is array** (address) **of** word;
type transform **is array** (1 **to** 4, 1 **to** 4) **of** real;
type registerbank **is array** (byte **range** 0 **to** 132) **of** integer;

Beispiel für ein unbeschränktes Array:

type vector **is array** (integer **range** <>) **of** real;

Arrayelemente werden über ihre Indizes angesprochen, z.B. word(12) oder transform(2, 3). Es kann auch ein zusammenhängender Bereich von Elementen angesprochen werden, indem man einen Bereich als Index verwendet: z.B. word(7 **downto** 0).

Arraywerte können durch sogenannte Aggregate dargestellt werden. Gegeben sei der Typ:

type a **is array** (1 **to** 4) **of** character;

und es soll ein Array mit den Elementwerten ‘H’, ‘a’, ‘a’, ‘r’ dargestellt werden, dann ist dies möglich mit einem Aggregat mit sogenannter positioneller Assoziation:

(‘H’, ‘a’, ‘a’, ‘r’).

Man kann in einem Aggregat allerdings auch die Positionsnummern explizit angeben:

(1 => ‘H’, 3 => ‘a’, 4 => ‘r’, 2 => ‘a’)

Um mehrere Elemente mit gleichem Wert ansprechen zu können, gibt es das Schlüsselwort **others**:

(‘H’, 4 => ‘r’, **others** => ‘a’)

others spricht dabei alle nicht im Aggregat explizit oder positionell bezeichneten Indizes des zugehörigen Arrays an.

2.2.4 Subtypes

Mithilfe von Subtypes können Wertebereiche von Basistypen eingeschränkt werden.

subtype_declaration ::= **subtype** identifier **is** subtype_indication
subtype_indication ::= [resolution_function_name] type_mark [constraint]
constraint ::= range_constraint | index_constraint

Beispiele:

subtype Kalendertage **is** integer **range** 1 **to** 31;
subtype digit **is** character **range** ‘0’ **to** ‘9’
subtype dreidimensional **is** vector(3 **downto** 1)

Im Gegensatz zu Objekten mit unterschiedlichen Basistypen können Objekte unterschiedlicher Subtypes vom selben Basistyp miteinander verglichen oder einander zugewiesen werden.

2.2.5 Objekt Deklarationen

Ein Objekt in einer VHDL Beschreibung ist durch einem Namen gekennzeichnet. Zusätzlich besitzt es einen Wert von einem bestimmten Typ. In VHDL unterscheidet man drei Arten von Objekten: Konstanten, Variablen und Signale. Im folgenden werden zunächst nur Konstanten- und Variablendeklarationen behandelt.
Eine Konstante ist ein Objekt, das bei seiner Erzeugung mit einem Wert initialisiert wird und später nicht mehr geändert werden darf:

constant_declaration ::=
constant identifier_list : subtype_indication [:= expression] ;

Konstantendeklarationen, bei denen der initialisierende Ausdruck fehlt sollen hier nicht näher betrachtet werden.

Beispiel für eine Konstantendeklaration:

constant e : real := 2.71828;

Variablen sind Objekte, denen bei ihrer Erzeugung Initialwerte zugewiesen werden können und deren Werte danach verändert werden dürfen.

variable_declaration ::=
variable identifier_list : subtype_indication [:= expression] ;

Beispiele:

variable a,b,c : integer := 3*7;
variable x : real;
variable v : vector(5 downto 1) := (2.3, 1.7, 3.0, 0.0, 1.0);

Fehlt ein initialisierender Ausdruck, werden den Variablen bzw. bei Arrayvariablen jedem einzelnen Element der Wert der linken Begrenzung des Wertebereichs eines Typs als Initialisierungswert zugewiesen.

2.3 Ausdrücke (expressions) und Operatoren

Ausdrücke in VHDL werden genauso gebildet, wie in anderen Programmiersprachen auch. Ein Ausdruck ist z.B. 3*a + (4-i)/3.
In VHDL existieren folgende Operatoren:

Höchste Priorität:	******	**abs**	**not**			
	*****	**/**	**mod**	**rem**		
	+ (sign)		- (sign)			
	+	-	**&**			
	=	/=	<	<=	>	>=
Niedrigste Priorität:	**and**, **or**, **nand**, **nor**, **xor**, **not**					

Die logischen Operatoren **and**, **or**, **nand**, **nor**, **xor** und **not** verarbeiten Operanden vom Typ bit oder boolean (s.o.) und ebenso eindimensionale Arrays von diesen Typen. Bei Arrays werden korrespondierende Elemente verknüpft, das Ergebnis ist wieder ein Array der gleichen Länge und vom gleichen Typ.
Die Operanden der vergleichenden Operatoren =, /=, <, <=, >, >= müssen vom selben Typ sein. Bei = und /= sind beliebige Typen zulässig, bei den anderen Operatoren nur skalare Typen und eindimensionale Arrays von diskreten (das sind Integer- und Aufzählungs-) Typen. Das Ergebnis ist vom Typ boolean.
Die Vorzeichenoperatoren (+, -), der Additions- (+) und der Subtraktionsoperator (-) haben ihre übliche Bedeutung. Der Verkettungsoperator (&) verknüpft zwei eindimensionale Arrays und/ oder Einzelelemente zu einem einzigen Array (z.B. “ab” **&** “zug” => “abzug”).
Der Multiplikations- (*****) und der Divisionsoperator (/) werden wie gewohnt gebraucht und können auf numerische Operanden angewendet werden. Der Betragswertoperator (**abs**) kann auf jeden numerischen Typ angewendet werden, **mod** ist der Modulo-Operator. Der **rem**-Operator und der Exponentialoperator (******) werden hier nicht näher erläutert.

2.4 Sequentielle Anweisungen

In VHDL existieren mehrere Anweisungen, die die Werte von Objekten verändern und den Ablauf steuern können.

2.4.1 Variablen Zuweisung

Durch eine Variablenzuweisung wird einer Variablen auf der linken Seite der Wert der rechten Seite zugewiesen. Linke und rechte Seite müssen denselben Typ haben.

```
variable_assignment_statement ::= target := expression ;
target ::= name | aggregate
```

Zuweisungen an Aggregate werden hier nicht behandelt.
Beispiele:

```
a := 3*b+c;
a(5) := '1';
```

2.4.2 If Anweisung

Durch eine If-Anweisung lassen sich Anweisungsblöcke derart selektieren, daß ihre Ausführung von einer Bedingung abhängt.

```
if_statement ::=
   if condition then
         sequence_of_statements
   { elsif condition then
         sequence_of_statements }
   [ else
         sequence_of_statements ]
   end if;
```

Das Resultat einer Bedingung ist ein boolscher Wert. Ist die Bedingung *true*, werden die Anweisungen des If-Teils, andernfalls die des else-Teils ausgeführt.

2.4.3 Case Anweisung

Die Case-Anweisung erlaubt die selektive Ausführung von Anweisungsblöcken in Abhängigkeit des Wertes eines sogenannten Selektorausdrucks.

```
case_statement ::=
      case expression is
         case_statement_alternative
         { case_statement_alternative }
      end case ;
case_statement_alternative ::=
      when choices =>
         sequence_of_statements
choices ::= choice { | choice }
choice ::=
      simple_expression
      | discrete_range
```

| *element*_simple_name
| **others**

Das Ergebnis des Selektorausdrucks muß entweder ein diskreter Typ oder ein eindimensionales Array von Zeichen (Zeichenkette) sein. Die Anweisungen derjenigen Alternative werden ausgeführt, die einen Auswahlwert (choice) enthält, der identisch mit dem Wert des Selektorausdrucks ist. Die Auswahlwerte müssen alle unterschiedlich sein und zusammen den kompletten Wertebereich des Selektorausdrucks abdecken. Decken die explizit aufgeführten Auswahlwerte nicht den kompletten Wertebereich ab, so muß in der letzten Alternative der Auswahlwert **others** verwendet werden, der alle noch nicht aufgeführten Auswahlwerte beinhaltet.
Beispiel:

```
case Kalendertag is            -- Wertebereich von Kalendertag sei 1 bis 31
   when 1 | 16 =>
        Anweisungen für 1 und 16 ;
   when 2 =>
        Anweisungen für 2 ;
   when 7 to 15 =>
        Anweisungen für 7 bis 15 ;
   when others =>
        Anweisungen für 3 bis 6 und 17 bis 31 ;
end case;
```

2.4.4 Loop Anweisung

in VHDL lassen sich unendliche Schleifen, sogenannte While-Schleifen und sogenannte For-Schleifen realisieren:

```
loop_statement ::=
   [ loop_label : ]
      [ iteration_scheme ] loop
         sequence_of_statements
      end loop [ loop_label ] ;

iteration_scheme ::=
   while condition
   | for loop_parameter_specification

parameter_specification ::= identifier in discrete_range
```

Schleifen ohne Iterationsschema werden unendlich oft wiederholt:

```
loop
   Anweisungen ;
end loop;
```

While-Schleifen werden solange wiederholt, bis die Schleifenbedingung nicht mehr erfüllt ist. Die Schleifenbedingung wird dabei vor jedem Schleifendurchgang überprüft:

```
while a < b and c = 3*4 loop
      Anweisungen ;
end loop;
```

For-Schleifen werden solange wiederholt, bis die Schleifenvariable den angegebenen Wertebereich verläßt. Die Schleifenvariable durchläuft dabei nacheinander alle Werte dieses Bereichs. Sie wird innerhalb der Schleife als Konstante angesehen, man darf ihr also keinen Wert zuweisen:

```
for i in 3 to last_i loop
   Tabelle(i) := 0;
end loop;
```

Der interne Schleifenablauf kann durch zwei zusätzliche Anweisungen unterbrochen werden, der **next**- und der **exit**-Anweisung:

next_statement ::= **next** [*loop*_label] [**when** condition] ;
exit_statement ::= **exit** [*loop*_label] [**when** condition] ;

Ist die jeweilige Bedingung erfüllt, dann beenden beide Anweisungen die momentane Iteration. Bei der **next**-Anweisung wird mit einer neuen Iteration der durch das *loop*_label referenzierten Schleife fortgefahren, bei der **exit**-Anweisung wird die referenzierte Schleife verlassen. Fehlt ein *loop*_label, so wird die innerste Schleife referenziert.

2.4.5 Null Anweisung

Die **null**-Anweisung hat keinen Effekt. Sie wird vor allem in Alternativen von **case**-Anweisungen eingesetzt, bei denen keine Anweisung ausgeführt werden soll:

null_statement ::= **null** ;

2.5 Unterprogramme und Packages

Unterprogramme sind Funktionen und Prozeduren und werden ähnlich wie in anderen Programmiersprachen behandelt. Innerhalb von Unterprogrammen können nur übergebene Parameter und lokal definierte Objekte verändert werden. Auf Objekte, die außerhalb des Unterprogramms deklariert und nicht als Parameter übergeben worden sind, kann innerhalb des Unterprogramms nicht zugegriffen werden, sogenannte Seiteneffekte sind dadurch ausgeschlossen. Im Praktikum müssen keine Unterprogramme geschrieben werden, sie werden deshalb hier nicht weiter vertieft. Wie Unterprogramme aufgerufen werden, wird an gegebener Stelle erläutert.
Packages sind Sammlungen von Typen, Konstanten und Unterprogrammen, die von anderen Modulen verwendet werden können. Mehrere Packages können in einer Bibliothek (library) zusammengefaßt werden. Eine Bibliothek wird üblicherweise durch ein Directory auf dem Computer dargestellt. Die Packages einer Bibliothek entsprechen dann den Files eines Directories. Ein Package kann in zwei Teile aufgespalten werden, einem Deklarationsteil, der nur die Schnittstelle spezifiziert, und einem sogenannten Package body, in dem die Details beschrieben werden. Da im Praktikum auf das Schreiben von Unterprogrammen verzichtet wird, soll hier nur ein Beispiel für ein ungespaltenes Package, das nur Typ und Konstantendefinitionen enthält, gegeben werden:

```
package beispiel is
   type byte_int is range 0 to 255;
   constant PI : real := 3.1416;
end beispiel;
```

Das Beispiel deklariert ein Package mit dem Namen ‘beispiel’, in dem der Typ ‘byte_int’ und die Konstante ‘PI’ deklariert werden. Eine Möglichkeit dieses Package in anderen Modulen verwenden zu können, zeigt das folgende Beispiel:

```
library XYZ;
use beispiel.all;
```

Hier wird dem Modul mitgeteilt, daß im folgenden alle Deklarationen innerhalb des Packages ‘beispiel’ der Bibliothek ‘XYZ’ gelten sollen. Byte_int und PI sind nun auch innerhalb dieses Moduls deklariert und können wie andere Typen oder Konstanten verwendet werden.
Im Praktikum ist das Programmsystem SYNOPSYS so konfiguriert, daß die benötigten Bibliotheken und Packages den entworfenen Modulen automatisch bekanntgemacht werden. Eine tiefergehende Einführungen wird hier daher nicht gegeben.

3 Strukturbeschreibungen in VHDL

Wie schon in der Einleitung erklärt wird eine digitale Schaltung in VHDL durch hierarchisch zusammengesetzte Module beschrieben. Um Module in eine solche Beschreibung einfügen zu können benötigt jedes Modul eine Entity-Deklaration, die die Schnittstelle zur Umwelt festlegt. Eine Entity-Deklaration, wie sie für die Praktikumsaufgaben benötigt wird, sieht folgendermaßen aus:

```
entity name is
      Port ( A:   In    integer;
             B:   In    bit;
             Y:   Out   bit       );
end name;
```

Das Beispiel beschreibt ein Modul mit dem Namen *name*, den beiden Eingangssignalen A und B und dem Ausgangssignal Y. Signal A ist hierbei vom Typ integer, Signal B und Y vom Typ bit. Portsignale können Eingabesignale (In), Ausgabesignale (Out), Ein-Ausgabesignale (In-Out) und Buffersignale (Buffer) sein. Eingabesignale können vom Modul nur gelesen, Ausgabesignale nur beschrieben werden. Ein-Ausgabesignale kann das Modul sowohl lesen, als auch beschreiben, als Beispiel seien bidirektionale Datenbusse genannt. Buffersignale sind ähnlich wie Ein-Ausgabesignale, können im Gegensatz zu diesen jedoch nur vom Modul selbst beschrieben werden. Man kann sie sich daher auch als Ausgabesignale vorstellen, die in das Modul zurückgeführt werden. Ein Beispiel ist der Zählerstand eines Zählerbausteins, der vom Modul gelesen werden können muß, um den nächsten Zählerstand bestimmen zu können.

Die Implementation einer Schaltungsfunktion eines Moduls, die sogenannte Architektur, wird folgendermaßen deklariert:

```
architecture identifier of entity_name is
      architecture_declarative_part
begin
      architecture_statement_part
end [ architecture_simple_name ] ;
```

Im Deklarationsteil werden Typen, Objekte, Funktionen, usw. deklariert. Bei einer strukturellen Beschreibung der Architektur werden im Praktikum hier üblicherweise auch die Signale deklariert, die die einzelnen Untermodule miteinander verbinden. Diese Signale sind im Gegensatz zu den Portsignalen für die Umgebung des Moduls nicht sichtbar und können immer sowohl beschrieben, als auch gelesen werden, zeigen ansonsten aber ein identisches Verhalten.

Im Anweisungsteil der Architekturdeklaration wird bei strukturellen Beschreibungen angegeben, aus welchen Untermodulen sich das Modul zusammensetzt und wie diese miteinander verbunden sind. Die Untermodule selbst können wiederum aus verschiedenen Untermodulen bestehen. Die einzelnen Module stellen Komponenten dar, die aufgrund ihrer Schnittstelle leicht mit anderen Komponenten verknüpft und hierarchisch gegliedert werden können.

In diesem Praktikum werden die Zusammensetzung der Module, ihre Schnittstellen und die Verbindungen untereinander mithilfe des Programmsystems SYNOPSYS in graphischer Form eingegeben. Aus dieser Information erzeugt SYNOPSYS automatisch die strukturelle VHDL-Beschreibung.

4 Verhaltenbeschreibung in VHDL

Wie schon erwähnt kann die Architektur eines Moduls auch in Form einer Verhaltensbeschreibung, also einem Programm, implementiert werden. Fast alle nötigen Bestandteile eines Programms wurden in Abschnitt 2 erklärt.
Bei einer Verhaltensbeschreibung werden im Deklarationsteil des Architekturteils alle Typen und Objekte, die lokal für diese Architektur gelten sollen, deklariert. Der Programmablauf selbst wird im Anweisungsteil durch sequentielle Anweisungen beschrieben. An dieser Stelle sollen die schon erklärten Anweisungen durch solche erweitert werden, die Signalwerte betreffen und auf Signalwertänderungen reagieren können.

4.1 Signal Zuweisung

Signalen können ganze Signalverläufe zugewiesen werden:

signal_assignment ::= target <= [**transport**] waveform ;
target ::= name | aggregate
waveform ::= waveform_element { , waveform_element }
waveform_element ::=
*value*_expression [**after** time_expression]
| **null** [**after** *time*_expression]

Diese Signalverläufe sind vor allem für Simulationszwecke nützlich.
In diesem Praktikum genügt es, lediglich zwischen asynchronen und synchronen Signalen zu unterscheiden. Tatsächlich erlaubt das VHDL-Synthese-Subset von SYNOPSYS nur folgende Signalzuweisungen:

name <= *value*_expression;

Hier wird dem Signal ‘name’ ein Wert ohne Verzögerung zugewiesen.

4.2 Prozesse und Wait Anweisung

Ein Verhaltensprogramm besteht aus einem oder mehreren Prozessen. Ein Prozeß ist ein Programmteil bestehend aus sequentiellen Anweisungen, das durch Zustandsänderungen von Signalen aktiviert werden kann. Mehrere Prozesse können dabei gleichzeitig aktiv sein.

process_statement ::=
[*process*_label :]
process [(sensitivity_list)]
process_declarative_part
begin
process_statement_part
end process [*process*_label] ;

Im Deklarationsteil werden im Praktikum üblicherweise lokale Typen, Konstanten und Variablen definiert. Der Anweisungsteil enthält eine Abfolge von sequentiellen Anweisungen.
Ein Prozeß wird während der Initialisierungsphase der Simulation aktiviert, führt alle Anweisungen des Prozesses aus und wiederholt die Ausführung beginnend mit der ersten Anweisung. Das alles geschieht in einem Simulationszyklus. Werden Signalen Werte zugewiesen, so erhalten sie (im Gegensatz zu Variablen) diesen Wert erst am Ende des Zyklus. D.h.: wird ein Signal

selbst zugewiesen, so ändert sich der Wert des Signals in einem Zyklus nicht, auch wenn dem Signal im selben Zyklus vorher ein neuer Wert zugewiesen wurde. *Der Wert eines Signals wird also während eines Zyklus festgehalten*. Werden einem Signal mehrmals Werte (nach oben genannter Vorschrift) zugewiesen, so ist immer nur die zuletzt ausgeführte gültig.
Ein Zyklus kann nur durch eine **wait**- Anweisung beendet werden, der Prozeß wird dann an dieser Stelle unterbrochen:

wait_statement ::=
wait [sensitivity_clause] [condition_clause] [timeout_clause] ;
sensitivity_clause ::= **on** sensitivity_list
sensitivity_list ::= *signal*_name { , *signal*_name }
condition_clause ::= **until** condition
timeout_clause ::= **for** *time*_expression

In einer ‘Sensitivitylist’ werden all diejenigen Signale aufgeführt, deren Wertänderung den Prozeß neu aktivieren können. Der Prozeß wartet bei einer solchen **wait**- Anweisung, bis sich der Wert mindestens eines Signals der Sensitivitylist geändert hat und die anschließende Bedingung erfüllt ist. Fehlt die Bedingung, wird sie als erfüllt angesehen. Fehlt die Sensitivitätsliste, reagiert der Prozeß nur auf die Signale, die innerhalb der Bedingung (sofern vorhanden) auftreten (die Sensitivitylist enthält dann sozusagen diese Signale). Der Prozeß wird aber spätesten nach Ablauf der angegebenen Zeitdauer reaktiviert. Fehlt eine Zeitangabe, wird die maximal mögliche Zeitdauer (hängt von der VHDL-Implementierung ab) angenommen. Die Zeitangabe (*time*_expression) muß dabei ein Ausdruck vom Typ ‘TIME’ sein. Dieser vordefinierte Typ ist ein sogenannter physikalischer Typ. Physikalische Typen sind zusammengesetzt aus einem Zahlenwert und einer Einheit. Ohne nähere Erläuterung sei hier die Typdefinition von ‘TIME’ gegeben:

```
type TIME is range implementation_defined
  units
    fs;                    -- femtosecond
    ps   =  1000 fs;       -- picosecond
    ns   =  1000 ps;       -- nanosecond
    us   =  1000 ns;       -- microsecond
    ms   =  1000 us;       -- millisecond
    sec  =  1000 ms;       -- second
    min  =  60 sec;        -- minute
    hr   =  60 min;        -- hour
  end units;
```

Die Einheiten können ineinander umgerechnet werden, es gilt also z.B. 1 hr = 3600 sec.
Die kleinste Zeitauflösung bei der Simulation ist 1 fs. Zeitangaben die kleiner sind als 1 fs werden als 0 fs interpretiert. Die Zeitauflösung kann vergröbert werden, um z.B. die Simulation zu beschleunigen. Im Praktikum wird im allgemeinen eine Zeitauflösung von 1 ns gewählt.
Falls ein Prozeß eine Sensitivitylist enthält, so wird vom Prozeß implizit ein

wait on *Sensitivitätsliste*

als letzte Anweisung ausgeführt. **Solche Prozesse dürfen keine expliziten wait-Anweisungen enthalten.**

Um in einem Prozeß mit mehreren Signalen in der Sensitivitylist gezielt eine Taktflanke abfragen zu können, kann man das Attribut **event** verwenden. Die Abfrage nach einer steigenden Taktflanke eines Signals sieht dann folgendermaßen aus:

if *signal_name***'event and** *signal_name* = '1' **then** ...

Dadurch wird es möglich synchrone Ereignisse mit asynchronen zu vermischen.

Die folgenden vier Beispiele verdeutlichen den Gebrauch von Sensitivitylisten und wait-Anweisungen. Sie stellen eine Art Prototyp für die effiziente Beschreibung von synchronen und asynchronen (reine Kombinatorik) Prozessen mit und ohne synchronem bzw. asynchronem Reset dar und sollen in den Praktikumsaufgaben verwendet werden.

Arbeiten Sie die Beispiele besonders gründlich durch! Sie vermeiden dadurch unnötige Schwierigkeiten bei der Bearbeitung der Aufgaben.

Beispiele:

-- Synchroner Prozeß, der ein Taktsignal CLK vom Typ bit mit der Taktfrequenz 10 MHz erzeugt:

```
Taktsignal : process
        constant Half_Period: TIME := 50 ns;
    begin
        CLK <= '0';
        wait for Half_Period;
        CLK <= '1';
        wait for Half_Period;
    end process Taktsignal ;
```

-- Asynchroner Prozeß, der ein Und-Gatter mit den Eingängen X, Y und dem Ausgang Z beschreibt (reine Kombinatorik):

```
    process (X, Y)
    begin
        Z <= X and Y;
    end process;
```

-- Prozeß, der zuerst das Ausgangssignal Outp mit '1' initialisiert, sobald das Eingangs-
-- signal Inp '1' ist zehn steigende Taktflanke von CLK abwartet und dann Out auf '0' setzt.
--Wann immer In '0' wird, wird der Prozeß **synchron** neu gestartet:

```
    process
        variable i: integer;
    begin
        Outp <= '1';
        i := 0;
reset: loop
            while i <= 10 loop
```

```
            i := i + 1;
            wait until CLK'event and CLK = '1';
            exit reset when Inp = '0';
        end loop;
        Outp <= '0';
        wait until CLK'event and CLK = '1';
        exit reset when Inp = '0';
    end loop;
end process;
```

-- Prozess für einen synchronen 4-Bit Zähler mit asynchronem Reset. Bei aktivem Resetsignal wird der Zählerstand Int_out **asynchron** auf den Wert 0 setzt:

```
process (Clk, Reset)
begin
    if Reset = '1' then        -- Reset aktiv: zu 0 setzten
        Int_out <= 0;
    elsif Clk'event and Clk = '1' then
        Int_out <= (Int_out + 1) mod 16;
    end if;
end process;
```

5 Vordefinierte Typen und Funktionen

VHDL stellt einige vordefinierte Typen und Funktionen im Package STANDARD zu Verfügung. Im Praktikum werden dabei nicht alle diese Typen und Funktionen benötigt, teilweise dürfen sie aufgrund des VHDL-Synthese-Subsets von SYNOPSYS auch gar nicht verwendet werden.

5.1 VHDL STANDARD Package

```
package STANDARD is

    -- predefined enumeration types:

    type BOOLEAN is (FALSE, TRUE);

    type BIT is ('0', '1');

    type CHARACTER is (
        NUL,  SOH,  STX,  ETX,  EOT,  ENQ,  ACK,  BEL,
        BS,   HT,   LF,   VT,   FF,   CR,   SO,   SI,
        DLE,  DC1,  DC2,  DC3,  DC4,  NAK,  SYN,  ETB,
        CAN,  EM,   SUB,  ESC,  FSP,  GSP,  RSP,  USP,

        ' ',  '!',  '"',  '#',  '$',  '%',  '&',  ''',
        '(',  ')',  '*',  '+',  ',',  '-',  '.',  '/',
        '0',  '1',  '2',  '3',  '4',  '5',  '6',  '7',
        '8',  '9',  ':',  ';',  '<',  '=',  '>',  '?',

        '@',  'A',  'B',  'C',  'D',  'E',  'F',  'G',
        'H',  'I',  'J',  'K',  'L',  'M',  'N',  'O',
        'P',  'Q',  'R',  'S',  'T',  'U',  'V',  'W',
        'X',  'Y',  'Z',  '[',  '\',  ']',  '^',  '_',

        '`',  'a',  'b',  'c',  'd',  'e',  'f',  'g',
        'h',  'i',  'j',  'k',  'l',  'm',  'n',  'o',
        'p',  'q',  'r',  's',  't',  'u',  'v',  'w',
        'x',  'y',  'z',  '{',  '|',  '}',  '~',  DEL);

    type SEVERITY_LEVEL is (NOTE, WARNING, ERROR, FAILURE);

    -- predefined numeric types:

    type INTEGER is range implementation_defined;

    type REAL is range implementation_defined;
```

```
    -- predefined type TIME:

    type TIME is range implementation_defined
      units
        fs;                 -- femtosecond
        ps    =  1000 fs;   -- picosecond
        ns    =  1000 ps;   -- nanosecond
        us    =  1000 ns;   -- microsecond
        ms    =  1000 us;   -- millisecond
        sec   =  1000 ms;   -- second
        min   =  60 sec;    -- minute
        hr    =  60 min;    -- hour
      end units;

    -- function that returns the current simulation time:

    function NOW return TIME;

    -- predefined numeric subtypes:

    subtype NATURAL is INTEGER range 0 to INTEGER'HIGH;

    subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;

    -- predefined array types:

    type STRING is array (POSITIVE range <>) of CHARACTER;

    type BIT_VECTOR is array (NATURAL range <>) of BIT;

end STANDARD;
```

Bemerkung:

Die ASCII Zeichen für (FS), (GS), (RS) und (US) werden im Typ CHARACTER durch FSP, GSP, RSP und USP dargestellt, um Konflikte mit den Einheiten im Typ TIME zu vermeiden.

5.2 SYNOPSYS Packages

Das Programmpaket SYNOPSYS stellt in der Bibliothek IEEE einige nützliche Packages zu Verfügung, die auch im Praktikum Anwendung finden:

Package std_logic_1164

In diesem Package sind u.a. die Typen STD_ULOGIC und STD_ULOGIC_VECTOR, STD_LOGIC und STD_LOGIC_VECTOR definiert, mit denen die Zustände elektrischer Signale und Busse beschrieben werden, wie es in der Industrie üblich ist:

```
type std_ulogic is (  'U',      -- Uninitialized
                      'X',      -- Forcing Unknown
                      '0',      -- Forcing 0
                      '1',      -- Forcing 1
                      'Z',      -- High Impedance
                      'W',      -- Weak Unknown
                      'L',      -- Weak 0
                      'H',      -- Weak 1
                      '-'       -- Don't care
                   );

    type std_ulogic_vector is array (NATURAL range <>) of std_ulogic;

    subtype std_logic is resolved std_ulogic;

    subtype std_logic_vector is array (NATURAL range <>) of std_logic;
```

Der Subtype STD_LOGIC ist mit einer sogenannten "Resolution-Function" mit dem Namen 'resolved' verknüpft. Dadurch können mehrere gleichzeitige Zuweisungen an ein und dasselbe Signal erfolgen. Der Wert des Signals wird dann aus allen aktuell aktiven Zuweisungen mithilfe der Resolution-Function bestimmt. (Im Falle der oben beschriebenen Zuweisungsmethode würde die letzte Zuweisung dann nicht mehr die vorhergehenden überschreiben).

Package std_logic_arith

In diesem Package sind unter anderem die Typen SIGNED und UNSIGNED definiert, die angeben sollen, ob der Wert eines Busses als vorzeichenbehaftet (Zweierkomplement) oder nichtinterpretiert werden soll. Zusätzlich existieren Funktionen, die einen Wert vom Basistyp INTEGER in einen Wert vom Typ STD_LOGIC_VECTOR, sowie Werte vom Typ SIGNED und UNSIGNED in eine Integerzahl umwandeln können:

```
    type unsigned is array (NATURAL <>) of std_logic;

    type signed is array (NATURAL <>) of std_logic;

    function conv_std_logic_vector (arg: integer; size: integer) return std_logic_vector

    function conv_integer (arg: unsigned) return integer;

    function conv_integer (arg: signed) return integer;
```

Die Typen SIGNED, UNSIGNED und STD_LOGIC_VECTOR haben dieselbe Definition und können daher mithilfe sogenannter Typkonvertierungsfunktionen ineinander umgewandelt werden. Dadurch wird es möglich z.B. einer Variablen vom Typ SIGNED einen Wert vom Typ STD_LOGIC_VECTOR zuzuweisen, indem man seinen Typ vorher in SIGNED umwandelt. Dazu setzt man den Wert in Klammern und schreibt den neuen Typ davor:

variable_*signed* := SIGNED(value_*std_logic_vector*)

Die Funktion CONV_STD_LOGIC_VECTOR wandelt eine Integerzahl (arg) in eine Zweier-

komplementdarstellung vom Typ STD_LOGIC_VECTOR mit ‘size’ Bits um. Die Funktion berechnet dabei jedes Bit einzeln beginnend mit dem niedrigstwertigen Bit und bricht nach ‘size’ Bits ab, unabhängig davon, ob die Zweierkomplementdarstellung vollständig ist oder nicht. (Positive Zahlen können also auch eine führende ‘1’ haben, wenn nur die Bitbreite richtig gewählt wird).
Die Funktionen CONV_INTEGER wandeln einen Wert ‘arg’ vom Typ UNSIGNED bzw. SIGNED in eine Integerzahl um. Hat ‘arg’ den Typ SIGNED, wird der Wert als Zweierkomplement interpretiert, andernfalls als positiver Wert. Bitwerte ungleich ‘1’ oder ‘0’ werden als ‘0’ interpretiert.
Der Funktionsaufruf verläuft genauso wie z.B in Pascal oder C. Beispiel:

variable_*std_logic_vector* :=
CONV_STD_LOGIV_VECTOR (*arg*_expression_*integer*, *size*_expression_*integer*)

oder: byte_vec := CONV_STD_LOGIC_VECTOR (3*a, 8);

6 Das VHDL Synthesesubset von SYNOPSYS

Das High-Level-Synthesesystem SYNOPSYS erlaubt es, VHDL Schaltungsbeschreibungen in Gate-Level-Beschreibungen, d.h. eine Beschreibung bestehend aus üblichen elektronischen Bausteinen wie Flipflops, logischen Gattern usw., automatisch umzuwandeln. Nicht jede VHDL Beschreibung läßt sich jedoch mithilfe solcher Bausteine realisieren. Als Beispiel sei ein Prozeß ohne Sensitivitylist und ohne explizites **wait** genannt. Dieser Prozeß würde nie unterbrochen werden, d.h. es könnte nie Zeit vergehen. Auch sind manche Beschreibungen zu Komplex, um sie mit SYNOPSYS umzuwandeln. SYNOPSYS schränkt daher für die Synthese den Gebrauch von VHDL Konstrukten ein. Nicht erlaubte Konstrukte wurden dabei in dieser VHDL Beschreibung schon weitgehend weggelassen, da sie im Praktikum nicht benötigt werden. Es gelten allerdings einige Einschränkungen bei der Verwendung von wait-Anweisungen, die noch nicht erwähnt wurden:

- Existieren in einem Prozeß explizite wait-Anweisungen, so muß entlang **jedes** möglichen **Pfades** durch den Prozeß mindestens eine **wait**-Anweisung vorkommen.
- **While**-Schleifen und reine Schleifen **müssen** mindestens eine **wait**-Anweisung beeinhalten.
- **For**-Schleifen dürfen **keine wait**-Anweisungen beinhalten. (In diesem Praktikum soll auf For-Schleifen ganz verzichtet werden).
- Explizite wait-Anweisungen müssen die Form '**wait until** *clock_signal*'**event and** *clock_signal = Wert*' haben.

Eine weitere Einschränkung betrifft den Modulo-Operator **mod**:

- Existiert eine Anweisung vom Typ 'a mod N', muß N für die Synthese eine Zweierpotenz sein: $N = 2^k$; $k \in \mathbb{N}$.

Beachten Sie diese Einschränkungen unbedingt, um unnötige Fehlermeldungen bei der späteren Synthese und dadurch notwendig gewordene, z.T. recht aufwendige Änderungen der VHDL-Beschreibungen zu vermeiden.

7 Literatur

IEEE *Standard VHDL Language Reference Manual*. IEEE Std 1076-1987

Peter J. Ashenden, *The VHDL Cookbook*, Dept. Computer Science, University of Adelaide, South Australia

Nützliche www-Seite:
http://tech-www.informatik.uni-hamburg.de/vhdl/index.html