

Rechnergesteuerte Systeme 2 (SS)
Modellierung und Simulation mit VHDL
(Veranstaltung über 1 Semester: V2 / Ü1 / P2)

Teil 3

Inhaltsübersicht:

5	Grundlagen der Sprache VHDL	32
5.1	Lexikalische Elemente	32
5.2	Syntax Beschreibungen	35
5.3	Steuerung des sequenziellen und nebenläufigen Programmablaufs	37
5.3.1	Sequenz vs. Block	38
5.3.2	Alternativen im sequenziellen und parallelen Fluss	40
5.3.3	Selektion / Alternative; case_Anweisung	42
5.4	Iterationen / Schleifen; loop_Anweisungen	44
5.5	Unterprogramme und Packages	48
5.5.1	Prozeduren und Funktionen	49
5.5.2	Package Konzept	57
5.6	Datenobjekte	59
5.6.1	Datentypen	59
5.6.2	Typ Vereinbarungen	60
5.6.3	Typ-Verträglichkeiten und Konvertierungen	60
5.6.4	Skalare Datentypen	61
5.6.5	Auflösefunktion (Resolution Function)	68
5.6.6	Attribute	70
5.6.7	Operationen	72
5.6.8	Operator-Überlagerung (operator overloading)	72
5.6.9	Zugriffstypen	72
5.6.10	Filetypen	72
5.6.11	Ein- und Ausgabe, Filetypen	73
5.6.12	Kompositorische Datentypen	73
6	Verhaltensorientierte Modellierung	76
7	Datenfluss-Modelle	76
8	Strukturorientierte Modellierung	77
8.1	Modellierung von Bussignalen	77
8.2	Komponenten und Konfigurationen	77
8.3	Generics	77
9	Synthesefähige Modellierung	77
10	Programmierung von Komplexlogiken mit VHDL	77
11	VHDL Syntax	78
12	Literatur	82

5 Grundlagen der Sprache VHDL

Wir wollen in diesem Kapitel die Sprache VHDL erlernen, um dann weiter in die oben angedeuteten Techniken der Modellierungen im Detail einsteigen zu können. Der grundsätzliche Aufbau einer VHDL Beschreibung wurde ja bereits weniger formal gezeigt und auch angewandt.

Wir werden uns jetzt zuerst die sequenziellen und nebenläufigen (concurrent) Programmkonstrukte ansehen. Wie bereits erwähnt, sind sequenzielle Konstrukte in Prozessen angesiedelt, während die nebenläufigen Konstrukte in der Architektur direkt auftreten und damit auch nebenläufig zu Prozessen stehen. Einige der Konstrukte hatten wir bereits benutzt, ohne dass wir die genaue Syntax dazu vorgestellt hatten. Dies wollen wir jetzt im Detail behandeln.

Erst im zweiten Schritt werden wir die wichtigsten Datentypen betrachten und die Sprachelemente dazu kennen lernen. Wir werden sehen, dass die Mächtigkeit der Sprache VHDL weit über die der Sprache C hinaus geht. Zudem werden wir die speziellen physikalischen Typen kennenlernen, die natürlich für eine HW-Beschreibungssprache unerlässlich sind.

Wir beginnen zunächst mit den gültigen Worten der Sprache VHDL, den lexikalischen Elementen.

5.1 Lexikalische Elemente

Lexikalische Elemente sind Folgen of Buchstaben, die die fundamentalen Elemente einer Sprache ausmachen. Eine VHDL Beschreibung ist z. B. eine Folge solcher Elemente, die durch Separatoren voneinander getrennt sind.

Zu den lexikalischen Elementen, also den gültigen Worten einer Sprache zählen:

- besondere Begrenzungszeichen (delimiter),
- Bezeichner (identifier),
- Kommentare (comment),
- Zeichen (character literal),
- Zeichenketten (string literal),
- Bitzeichenketten (bit string literal) und
- abstrakte Zahlen (abstract literal).

Zwischen den Elementen sind beliebige Anzahlen von Leerzeichen, Tabulatoren und/oder Zeilenumbrüche (Separatoren) zugelassen.

Im Allgemeinen werden die gültigen Wörter einer Sprache mit sog. regulären Ausdrücken beschrieben und mit Hilfe von Automaten erkannt. Wir wollen uns hier aber die Theorie ersparen und verbal beschreiben, welche Wörter zulässig sind.

Sonderzeichen (delimiter) sind in VHDL im allgemeinen benutzt für Operationen, als Begrenzer für bestimmte Teilkonstrukte der Sprache und zur Interpunktion. Diese sind:

" # & ' () * + - , . / : ; < = > [] |

Ferner gibt es **Paare von Sonderzeichen** mit besonderer Bedeutung:

=> ** := /= >= <= <>

Eine besondere Bedeutung für das "<=" ist „kleiner oder gleich“, in einem anderen Kontext hat es aber auch eine andere Bedeutung.

Begrenzer haben eine gewisse Ersatzfunktion für Separatoren, wie das Beispiel (A * B) kann auch ohne Leerzeichen geschrieben werden als (A*B).

Bezeichner (identifier) benennen Objekte in VHDL, wie z.B. Variablen, Signale, Prozesse u.s.w., sind aber auch reservierte Worte der Sprache.

Einige Charakteristika von Bezeichnern sind:

- sie dürfen nur Groß- und Kleinbuchstaben, Zahlen und den Unterstrich "_" enthalten,
- sie müssen mit einem Buchstaben beginnen,

- sie dürfen *nicht* mit einem Unterstrich beginnen oder enden und
 - sie dürfen *keine zwei aufeinander folgenden* Unterstriche beinhalten.
- VHDL ist nicht *case-sensitive*, d.h. es unterscheidet nicht zwischen Groß- und Kleinschreibung.

Einige Beispiele erlaubter Bezeichner:

COUNT ist identisch **count**; **last_value**, **h3Z25**, **Date_2305**

Einige Beispiele unerlaubter Bezeichner:

last@value	-- illegales Zeichen
5bit_counter	-- beginnt mit numerischen Zeichen
_A0	-- beginnt mit Underline
A0_	-- endet mit Underline
clock__pulse	-- zwei Underlines
end	-- reserviertes Wort ist nicht erlaubt

Die Liste der **reservierten Worte** der Sprache ist Bild 5.1 gegeben. Sie sind die Schlüsselworte hier für die Sprache VHDL und müssen nicht explizit zur Identifizierung angegeben werden.

abs	disconnect	label	package	sla
access	downto	library	port	sll
after	else	linkage	postponed	sra
alias	elsif	literal	procedure	srl
all	end	loop	process	subtype
and	entity	map	protected	then
architecture	exit	mod	pure	to
array	file	nand	range	transport
assert	for	new	record	type
attribute	function	next	register	unaffected
begin	generate	nor	reject	units
block	generic	not	rem	until
body	group	null	report	use
buffer	guarded	of	return	variable
bus	if	on	rol	wait
case	impure	open	ror	when
component	in	or	select	while
configuration	inertial	others	severity	with
constant	inout	out	shared	xnor
	is		signal	xor

Bild 5.1: Reservierte Worte in VHDL

Kommentare in VHDL werden eingeleitet mit "--" und enden am Ende einer Zeile. Der dem "--" folgende Text kann alle Buchstaben und Zeichen enthalten. Der Beginn kann beliebig in der Zeile sein. Dieser Teil nach dem Doppelpunkt wird vom Compiler einfach ignoriert, bis ein Zeilenende kommt. In den obigen Beispielprogrammen hatten wir schon viele solcher Kommentare gesehen.

```
-- es folgt eine Variablenzuweisung
C := A + B;      -- dies ist eine gültige Variablenzuweisung
```

Zeichen (character literal) werden in VHDL in Hochkommas eingerahmt. Einige Beispiele dazu sind:

Großbuchstaben: 'A', 'B', ..., 'Z';	Kleinbuchstaben: 'a', 'b', ..., 'z';
Ziffern: '0', '1', ..., '9';	Sonderzeichen: '!', '\$', '&', ...;
Hochkomma: "'" und	Leerzeichen ' '.

Zeichenketten (string literal) bestehen aus einer Folge von Einzelzeichen und werden in Anführungszeichen angegeben. Beispiele hierzu sind:

"Eine Zeichenkette" oder auch **"any printing chars (&%@^*)"**, **"0011ZZZ"** oder auch **""**, der leere String.

Zeichenketten, die nicht in eine Zeile passen, können auch mit dem *Kettungsoperator für Strings*, dem **"&"** aneinander gekettet werden, wie z.B.

"Eine Zeichenkette, die nicht in eine Zeile passt" & "kann mit dem Operator & gekettet werden"

um die zwei Teilstrings zu einem zu binden.

Im Praktikumsversuch 2 wird der Datentyp **bit_vector** verwendet, der einen String darstellt (ein String ist in VHDL ein eindimensionales Feld, ein Vektor). Das Verschieben eines 8 stelligen strings **d** wird durch die Verkettung des *character_literal* **'0'** mit dem String (bit_vector) von links nach rechts ohne die letzte Stelle erreicht, also durch

`d := '0' & d(7 downto 1);` -- Verkettung eines Zeichens mit einer Zeichenkette über **&**

Bitzeichenketten (bit string literal) sind eine Besonderheit, da sie in VHDL der Hardware sehr nahe kommen. Für Schaltungsentwicklungen müssen alle numerischen Werte in solche Bitketten umgesetzt und alle Operationen auf solche Bitketten angewandt werden.

Solche Folgen von Zeichen werden also häufig benötigt und sind **binär** mit **B"..."**, **oktal** mit **O"..."** oder **hexadezimal** mit **X"..."** anzugeben. Entsprechend sind die im String zugelassenen Zeichen bei B nur 0 oder 1, bei O sind es die Zeichen 0, 1, 2, 3, 4, 5, 6, 7 und bei X die Zeichen 0, 1, ..., 9, A, B, C, D, E, F. Bitketten ermöglichen also dem Entwickler, z.B. Speicherinhalte nicht immer binär sondern auch gekürzt als Oktal- oder Hexadezimalwert anzugeben. Das Resultat ist in jedem Fall ein als Bit String angegebene Bitmuster. Beispiele dazu:

B"0100011", oder **B"10"**, oder auch **b"1111_0010_1010_0101"**, oder **B""**.

Oktal angegebene Strings fassen 3-bit Gruppen zusammen, wie

O"372" ist äquivalent zu **B"011_111_010"**, oder auch **o"00"**, das äquivalent **B"000_000"** ist.

Hexadezimal angegebene Strings fassen 4-bit Gruppen in gleicher Weise zusammen, wie

X"FA" äquivalent zu **B"1111_1010"**, oder auch **x"0d"** äquivalent zu **B"0000_1101"**.

Man beachte, dass **O"372"** nicht äquivalent zu **X"FA"** ist, da der erste 9 Bitstellen anspricht, der letzte jedoch nur 8!!!

Zahlenangaben (abstract literal) haben eine besondere Bedeutung, vor allem, wenn man ihre Darstellung in verschiedenen Zahlenbasen betrachtet. Dabei können wir zwischen Ganzzahlen (Integer) und gebrochen rationalen Zahlen (Real) unterscheiden.

Ein **Integer (integer literal)** repräsentiert eine Ganzzahl und besteht aus Stellenzahlen (**digits**) ohne einen Dezimalpunkt. Dezimale Integer-Literale sind z.B. **23 0 146**.

Reals (real literal) repräsentieren gebrochen rationale Zahlen und haben immer einen Dezimalpunkt, durch den sie sich von den integer literals unterscheiden, wie z.B. **23.1 0.0 3.14159**.

Ohne Beweis sollte klar sein, dass die maschinelle Repräsentation solcher Rationalzahlen auf Grund der Stellenbeschränkung nur eine Näherung darstellen kann.

Zahlenangaben können in Dezimalform sein, wie es in vielen Programmiersprachen der Fall ist, oder in einer anderen Zahlenbasis angegeben werden. Entsprechend unterscheidet das LRM für das **abstract_literal** einerseits **decimal_literal** und andererseits **based_literal**.

Beide Literale können auch in der sog. E-Notation, der halblogarithmischen Darstellung erscheinen. Die Zahl hinter dem "E" oder dem "e" bezeichnet dann die Anzahl der Positionen, die der Stellenpunkt nach links (negativ) oder nach rechts zu verschieben ist.

Dezimale Numerikangaben für Integer sind z.B. die folgenden:

46E5 entspricht der Ganzzahl 4600000,
1E+12 ist entsprechend 10^{12} und
19e00 ist eben gleich 19.

Beispiele für `real_literal` in E-Notation wären:

1.234E09 ist $1.234 \cdot 10^9$,
98.6E+21 ist $0.986 \cdot 10^{23}$ und
34.0e-08 ist 0.00000034.

Man kann in VHDL natürlich auch die Zahlen in verschiedenen Basis-Systemen als sog. **based_literal** angeben.

Beispiel sind:

2#11111101# (Dualzahl) = **8#375#** (Oktalzahl) = **16#0fd#** = **16#FD#** (Hexadezimalzahl).

Die Basis muss in VHDL zwischen 2 und 16 liegen!

Natürlich gilt das auch für Real-Zahlen. Als Beispiel ist dezimal 0.5 gleich

2#0.100# (Dualzahl) = **8#0.4#** (Oktalzahl) = **12#0.6#** (Basis 12 Zahl)

Auch die E-Notation gilt:

2#1#E10 = 2^{10} , **16#4#E2** = $4 \cdot 16^2$ **10#1024#E+00** = 1024.

Man beachte, dass die Zahlenangabe hinter dem „E“ eine dezimale ist.

Aus Gründen der besseren Lesbarkeit kann man auch Gruppen von Ziffern mit dem Unterstrich „_“ abtrennen, ähnlich der gewohnten Tausender-Trennung im Dezimalen. Wie z.B.:

123_456 , **3.141_592_6** , **2#1111_1100_0000_0000#**.

5.2 Syntax Beschreibungen

Im allgemeinen gibt es eine Vielfalt in der Darstellung von Grammatikregeln. Mittlerweile hat sich aber die erweiterte Backus-Naur-Form (EBNF) durchgesetzt, die im Zusammenhang mit der Sprache ALGOL60 bekannt wurde und auch im VHDL LRM Verwendung findet. Die EBNF versucht, die Sprache in sog. Syntaktische Einheiten einzuteilen. Für jede solcher Syntaktischer Einheiten beschreibt man Regeln, die Sprachklauseln der Zielsprache aus anderen **Syntaktischen Einheiten** zusammensetzen.

Wir starten mit dem Beispiel der Beschreibung einer Variablenzuweisung in der EBNF:

variable_assignment_statement ::= target := expression;

Die Syntaktische Einheit ist hier **Variablenzuweisung**. Das Zeichen "**::=**" steht für "**ist definiert zu**". Hier besteht die Beschreibung der Zuweisung aus einer einfachen Aneinanderreihung von "**target**" als weitere Einheit, die den Variablennamen darstellt, einem Symbol "**:=**" und einer weiteren Einheit "**expression**". Setzen wir die Kenntnis der Beschreibung der Einheiten voraus, so lassen sich alle Zuweisungen mit dieser Regel auf Gültigkeit prüfen.

Wir wollen nicht in die Theorie und in die Tiefen solcher Beschreibungen gehen, sondern sie nur nutzen, um die Konstrukte etwas allgemeiner zu beschreiben. Dazu müssen wir noch ein paar weitere Regeln der Beschreibung kennen und sollten auch eine Ableitung versuchen.

Mit "[...]" beschreibt man eine Option, d.h. der geklammerte Teil kann einbezogen oder ausgelassen werden. Ein Beispiel dazu ist die wait-Anweisung, die wir bisher weniger formal betrachtet haben. Die EBNF ist

wait_statement ::= wait [sensitivity_clause] [condition_clause] [timeout_clause];

Wir sehen hier die drei möglichen Optionen, die alle fehlen dürfen, um beim reinen wait zu enden. Weiter ist jede Kombination der drei Klauseln zulässig.

Versuchen wir die erste Klausel aufzulösen, so finden wir:

sensitivity_clause ::= on sensitivity_list

also das Schlüsselwort on, gefolgt von der Sensitäts-Signalliste und unter dieser finden wir im LRM:

sensitivity_list ::= signal_name { , signal_name }

Die syntaktische Einheit *signal_name* endlich ist gegeben zu:

signal_name ::= identifier

womit die Ableitung terminiert, d. h. die Klausel ist, falls eine auf Gültigkeit geprüft wird, syntaktisch korrekt.

Es sei hier angemerkt, dass **name** natürlich alle möglichen Namen benennt und hier durch den Kursivdruck eine semantische Information eingefügt wird, dass es sich hier speziell um einen Signalnamen handelt.

Wir haben oben die geschweifte Klammer "{...}" gesehen, die andeutet, dass der Inhalt einmal oder beliebig oft wiederholt werden kann.

Die Bedingung, also die zweite Klausel im *wait* ist wie folgt gegeben:

condition_clause ::= until condition

unter condition ist gegeben zu:

condition ::= boolean_expression

Den Ausdruck (expression) werden wir unten noch behandeln, aber auch hier wieder die Angabe, dass ein rein boolescher Ausdruck gemeint ist, der zu **true** oder **false** evaluierbar ist, dass eben eine Bedingung entsteht.

Der Vollständigkeit halber noch die dritte Klausel:

timeout_clause ::= for time_expression

an der wir das Schlüsselwort **for** sehen und wieder einen Ausdruck aber jetzt einen Zeitausdruck (*time_expression*). Dieser Ausdruck wird zu einer Zeitangabe evaluiert, nämlich die Wartezeit.

Wir haben oben das Element **expression** gesehen und hier **boolean_expression** und **time_expression**. Solche Ausdrücke sind sehr komplex in der LRM Syntax angegeben und in ihrer Ableitung hier viel zu aufwendig. Dies hat seinen Grund in den Vorrangregeln der Operatoren. Wir wollen das ein wenig vereinfachen.

So wollen wir unter einem numerischen Ausdruck abweichend vom LRM **numerical_expression** verstehen als:

numerical_expression ::= [+ | -] num_term { (+ | - |) num_term }

num_term ::= [abs] numerical_factor { (* | / | mod | rem) numerical_factor }

numerical_factor ::= constant | variable | signal -- als numerischer Typ

condition ::= boolean_expression

boolean_expression ::= [not] boolean_factor { (and | or |) [not] factor }

boolean_factor ::= constant | variable | signal -- als boolean Typ

Wir erlauben uns hier den etwas vereinfachenden Freiheitsgrad.

Zudem sollten wir aus Vereinfachungsgründen vereinbaren, dass wir

Abfrage synonym zu **Bedingung (condition)** und zu **boolean_expression** sehen und dass wir häufig **Ausdruck (expression)** statt **numerical_expression** gebrauchen.

Wir wollen weiter unter **Wert-Bestimmung** (*value_expression*) die Alternative:

value_expression ::= *numerical_expression* | *boolean_expression*

verstehen.

So würde bei der Variablenzuweisung (*variable_assignment_statement*) oben für *expression* eben *value_expression* stehen.

variable_assignment_statement ::= *target* := *value_expression*;

Etwas schwieriger bildet sich *signal_assignment_statement* zu:

signal_assignment_statement ::= *target* <= {*delay_mechanism*} *waveform*;

Das *target* ist wieder der Signalname. Der Delay Mechanismus ist gegeben mit:

delay_mechanism ::= **transport** | [**reject** *time_expression*] **inertial**

und *waveform* ist:

waveform ::= *waveform_element* { , *waveform_element* }

waveform_element ::= *value_expression* [**after** *time_expression*]

5.3 Steuerung des sequenziellen und nebenläufigen Programmablaufs

Wie bereits erwähnt ist das Grundkonstrukt für die sequenziellen Bearbeitung und der Variablenzuweisungen der VHDL-Prozess. Die EBNF Regel für die Prozess-Definition kann wie folgt gegeben werden:

```
process_statement ::=
[ process_label :]
process [ (sensitivity_list) ] [is]
  process_declarative_part
begin
  process_statement_part
end process [ process_label ];
```

Dabei ist das *process_label* ein einfacher Bezeichner. Unter dem *process_declarative_part* finden wir die allgemeinen Vereinbarungen wie unter anderem die bekannten Variablenvereinbarungen.

Wir wollen jetzt die sequenziellen Anweisungen weiter beschreiben, also das, was zwischen *begin* und *end* als Wiederholung { *process_statement_part* } angeführt ist und seinerseits beschrieben ist mit

process_statement_part ::= { *sequential_statement* }

Auf der anderen Seite wissen wir, dass der Prozess einerseits nur in einer Architektur auftreten kann und andererseits nicht geschachtelt werden darf. Damit zählt das *process_statement* zu den nebenläufigen Befehlen, *concurrent_statement*.

Bei der Betrachtung einer Architektur hatten wir festgestellt, dass sie zwischen *begin* und *end* nur nebenläufige Befehle enthält.

Wie man bei imperativen Hochsprachen z.B. in Ada Sequenzen von Befehlen zu sog. Programm- oder Strukturblöcken zusammenfassen kann, so gilt dies in VHDL in Architekturen für Blöcke von nebenläufigen Befehlen, die auch geschachtelt werden können. Die Architektur selber ist auch so ein Block.

Bemerkung: Ada erlaubt Blöcke mit eigenem Deklarations- und Ausführungsteil, die mit dem Schlüsselwort **block** beginnen und die **begin-end** „Klammerung“ haben. Dies ist in C vergleichbar mit der geschweiften Einklammerung von Funktionskörpern { ... }.

Somit hat die EBNF für eine Architektur das folgende Aussehen:

```
architecture_body ::=
  architecture identifier of entity_name is
    architecture_declarative_part
  begin
    architecture_statement_part
  end [architecture] [ architecture_name ];
```

Der *identifier* ist identisch mit dem *architecture_name*, wie z.B. *behavior* oder *dataflow*.

Der Block von nebenläufigen Befehlen ist damit sehr ähnlich und man kann ihn dem Prozess gegenüber stellen. Der VHDL Block gehorcht der EBNF Produktionsregel:

```
block_statement ::=
  [ block_label :]
  block [ (guard_expression) ] [is]
    block_header
    block_declarative_part
  begin
    block_statement_part
  end block [ block_label ];
```

Hier gilt dann:

```
architecture_statement_part ::= { concurrent_statement }
block_statement_part ::= { concurrent_statement }
```

5.3.1 Sequenz vs. Block

Oben definierten wir die **Variablenzuweisung** (*variable_assignment_statement*) wie auch die **wait** Anweisung. Wir finden beide nur unter **sequential_statement**, während die auch oben definierte **Signalzuweisung** (*signal_assignment_statement*) sowohl hier, als auch unter **concurrent_statement**, dort allerdings als **concurrent_signal_assignment_statement**, zu finden ist. Es bestehen also Unterschiede, wie bereits oben angedeutet.

Was in Ada oder C als Programmblock (s.o.) auftritt ist in VHDL als **sequence_of_statements** wie folgt definiert:

```
sequence_of_statements ::= { sequential_statement }
sequential_statement ::=
  wait_statement
  | assertion_statement
  | report_statement
  | signal_assignment_statement
  | variable_assignment_statement
  | procedure_call_statement
  | if_statement
  | case_statement
  | loop_statement
  | next_statement
  | exit_statement
  | return_statement
  | null_statement
```

Hier ist die Sequenz Bestandteil des Prozesses, in dem es keine gesonderte Blockung mehr gibt. Die Programmblöcke sind halt gegeben durch die Steuerkonstrukte der Sprache wie **wait**, **is**, **case**, **loop**, **next** und **exit**.

Diesen Konstrukten können wir dem nebenläufigen Zuweisungsblock gegenüber stellen:

```

block_statement_part ::= { concurrent_statement }
concurrent_statement ::=  block_statement
                             | process_statement
                             | concurrent_procedure_call_statement
                             | concurrent_assertion_statement
                             | concurrent_signal_assignment_statement
                             | component_instantiation_statement
                             | generate_statement
    
```

Da hier alles als nebenläufig, also quasi gleichzeitig arbeitend anzunehmen ist, ist natürlich der Steuerungsmechanismus ein anderer. Das **block_statement** als erste Alternative in der obigen EBNF ist weiter zu betrachten. Darunter finden wir in der entsprechenden EBNF das optionale [(*guard_expression*)]. Der sogenannte **Guard** ist eine Art logisches Signal (*true* | *false*), das für den **Guarded-Block** eine Steuerfunktion hat. Der Guard Ausdruck ist also eine Bedingung, die auf den *Guard* evaluiert wird. Der Guard wirkt steuernd auf alle nebenläufigen Signalzuweisungen (EBNF siehe später), die innerhalb des Blockes mit dem Schlüsselwort **guarded** beginnen. Die Regeln dazu sind:

- ist der **Guard = true**, dann werden alle Guarded-Signalzuweisungen im Block ausgeführt, wenn sich die Zuweisungswerte ändern (wie bekannt),
- ist der **Guard = false**, werden diese Signalzuweisungen deaktiviert und
- **wechselt der Guard von false nach true**, werden die Signalzuweisungen auf jeden Fall ausgeführt.

Hier haben wir die Möglichkeit des direkten Vergleichs der Konstrukte Block und Prozess. Der folgende Guarded-Block

```

GaBlo: block (guard-expression)
      signal sig: bit;
      begin
        sig <= guarded waveform-elements;
      end block GaBlo;
    
```

hat ein direkt vergleichbares Prozess-Konstrukt, dessen Verhalten identisch ist.

```

architecture .....
  signal sig: bit;
  signal guard: boolean;
  begin
  ....
  guard <= guard-expression;

  GaBlo:
  process (guard, signals in waveform-elements) is
  begin
    if guard then
      sig <= waveform-elements;
    end if;
  end process GaBlo;
end architecture ...;
    
```

Auf das IF-Konstrukt im Prozess werden wir direkt im Anschluss eingehen. Wir können hier festhalten:

Wir haben ohne Nachweis der Allgemeingültigkeit die Alternative zwischen der sequenziellen Beschreibung durch ein Prozess-Konstrukt und der nebenläufigen Beschreibung mit Block-Konstrukten gesehen. Bei Prozess-Konstrukten entscheidet die Sensitätsliste über die Aktivierung eines Prozesses, was für Signalzuweisungen in gleicher Weise für die Änderung der an der Zuweisung beteiligten Signale gilt. Der Guard eines Blockes erlaubt die Steuerung über die eingeschlossenen Guarded-Signalzuweisungen.

Die Sequenz ist die Folge sequenzieller Anweisungen, die in ihrer angegebenen Folge auch bearbeitet werden. Dieser sog. sequenzielle Fluss kann durch steuernd wirkende Befehls-Konstrukte, die **if**, die **case**- und **loop-Anweisung**, beeinflusst werden. Diese wollen wir jetzt näher ansehen.

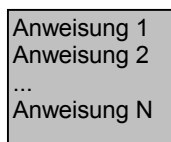
Wir werden auch feststellen, dass es für die nebenläufigen Signalzuweisungen ähnliche Konstrukte gibt und wir wollen hier auch den direkten Vergleich ziehen.

Um einfache Sequenzen von Anweisungen zu kennzeichnen und in ihrer Funktion rein visuell voneinander abzugrenzen hat VHDL im Prozess das Begrenzerpaar **begin...end**; (C nutzt hier {...}!). auch der Block ist durch dieses Begrenzerpaar gekennzeichnet.

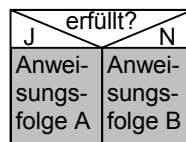
Im Falle des sequenziellen Steuerflusses ist VHDL vergleichbar mit einer ganz normalen, höheren Programmiersprache wie z.B. C oder Ada. Solche sogenannte blockorientierte Programmiersprachen betrachten funktional z.B. in natürlicher Sprache beschreibbare Teile des Programms als Strukturblock. Strukturblöcke haben Schachtelungscharakter, d.h. ein Strukturblock kann aus mehreren Strukturblöcken bestehen, es entsteht also eine Hierarchie, der Voraussetzung der sog. strukturierten Programmierung ist.

Wir sollten hier noch einer möglichen Begriffsirritation vorbeugen. Solche Strukturblöcke haben zunächst nichts mit den in der Betrachtung der Nebenläufigkeiten oben beschriebenen Block zu tun. Dort beschreibt er aber auch eine funktional zusammengehörige Einheit von eben nebenläufigen Anweisungen. Daher nutzen wir für den sequenziellen Fall zur Abgrenzung den Begriff *Strukturblock*.

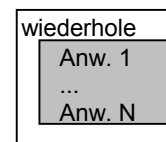
Man unterscheidet in der strukturierten Programmierung zwischen drei Strukturblöcken, die auch grafisch als sog. Struktogramme gezeichnet werden können.



1. Sequenz



2. Selektion



3. Iteration

Die Eigenschaft eines Strukturblockes ist, dass die Bearbeitung der Befehlsfolgen oben beginnt und unten im Strukturblock, also nur an einer gemeinsamen Stelle im Programm endet. Nur so ist gewährleistet, dass die Strukturblöcke aneinander anschließen können.

5.3.2 Alternativen im sequenziellen und parallelen Fluss

Das Grundkonstrukt zur Selektion ist das sog ITE-Konstrukt (*if-then-else*).

Dazu die EBNF. Man beachte, daß der *elsif*- wie auch der *else*-Zweig weggelassen werden können aber in VHDL der Strukturblock mit *end if* abzuschließen ist..

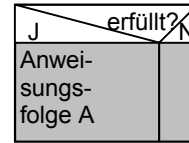
```

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

Drei Beispiele für den Praktikumsversuch 2 zeigen mögliche Alternativen auf. Wir hatten **d** deklariert als 8-stelligen Bitvektor (`variable d : bit_vector(7 downto 0);`). Man kann nun folgende Konstrukte formulieren:

Einfaches if Konstrukt

```
if (d and "00000001") = "00000001" then
  n := n + 1;
end if;
```



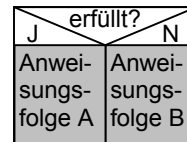
Hier wird der Bitvektor **d** bitweise logisch UND verknüpft mit dem 8-bit Wert, der ganz rechts eine '1' hat. Somit maskiert man das untere Bit von **d** aus, denn nur hier kann durch die Verknüpfung mit der '1' der Bitwert von **d** wirksam werden. Alle mit '0' angegebenen Bits ergeben eben '0'!

Man beachte, dass dieses Konstrukt keine Alternative hat. Zur Angabe der Alternative ist das Schlüsselwort *else* anzugeben.

Wir könnten nun auch die obere und untere Hälfte von **d** ausfiltern (auf '1' gesetzte Bits abfragen). Zuerst nur die Abfrage, ob ein Bit in der unteren Hälfte ungleich '0' ist. Entsprechend wird ein Indikator *no1_lh* (no 1 in lower half), ein logisches Signal, gesetzt:

if-then-else Konstrukt

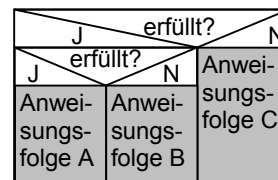
```
if (d and "00001111") = "00000000" then
  no1_lh <= true;
else
  no1_lh <= false;
end if;
```



Wollen wir den Test auch für die Hälfte der unteren 4 Bits in gleicher Weise durchführen, können wir einen weiteren Indikator *no1_lq* (no 1 in lower quarter), einführen.

if Schachtelung

```
if (d and "00001111") /= "00000000" then
  no1_lh <= false;
  if (d and "00000011") = "00000000" then
    no1_lq <= true;
  else
    no1_lq <= false;
  end if;
else
  no1_lh <= true;
  no1_lq <= true;
end if;
```



Man beachte, dass VHDL für jedes *if*-Konstrukt ein *end if* benötigt.

VHDL definiert auch ein *elsif*-Konstrukt, das ausschließenden Charakter hat, d.h. bei Nichtzutreffen der ersten Bedingung können mit *elsif* weitere Bedingungen frei formuliert werden. Das letzte *else* ist dann die Alternative zum ersten *if*!

Ein Beispiel dazu ist das generelle Zulassen bestimmter Zuweisungen durch ein sog. Enable-Signal **en**. Ist eine Eingangsdurchschaltung nicht erlaubt (*disable*; *en* = '0') so wird der Ausgang *result* festgelegt (hier zu 0). Ist eine Durchschaltung erlaubt, wird abhängig von einem Signal oder einer Variablen *op* (kann yes oder no sein) eines der Eingangssignale, *input_0* oder *input_1* auf den Ausgang *result* durchgeschaltet.

if- elsif Konstrukt

```

if en = '0' then
  result <= 0;
elsif op = yes then
  result <= input_0;
else
  result <= input_1;
end if;

```

Auch zu dem *if*-Konstrukt kann man, wenn man die nebenläufigen Signalzuweisungen betrachtet, ein vergleichbares Konstrukt finden.

Zunächst die nebenläufige Signalzuweisung im Detail in der EBNF:

```

concurrent_signal_assignment_statement ::=
  [ label : ] conditional_signal_assignment
  | [ label : ] selected_signal_assignment

```

Unter der bedingten Signalzuweisung finden wir:

```

conditional_signal_assignment ::=
  target <= options conditional_waveforms;

```

Unter options findet man:

```

options ::= [ guarded ] [ delay_mechanism ]

```

Den delay_machnism hatten wir bereits aufgezeigt. Hier ist das *guarded* zu finden!

Letztlich die *conditional_waveforms*:

```

conditional_waveforms ::=
  { waveform when condition else }
  waveform [ when condition ]

```

wobei wir *waveform* und *condition* oben bereits definiert hatten.

Setzen wir das obige *elsif*-Konstrukt um in eine solche Signalzuweisung mit gleichem Verhalten, so folgt:

```

result <= 0 when en = '0' else
  input_0 when op = yes else
  input_1;

```

5.3.3 Selektion / Alternative; case_Anweisung

Sucht man Abhängigkeiten von Alternativen eines Objekttyps oder einer Evaluation eines Ausdrucks, so kann man beim sequenziellen Fluss in einem Prozess das *case*-Konstrukt nutzen. Dazu wieder die EBF:

```

case_statement ::=
  [ case_label : ]
  case expression is
  case_statement_alternative
  { case_statement_alternative }
  end case [ case_label ];

case_statement_alternative ::=
  when choices =>
  sequence_of_statements

```

```

choices ::=
  expression
  | discrete range
  | element_name
  | others

```

Die Auswahl kann also aus

- einem Ausdruck, der evaluiert wird und mehrere Alternativen bezeichnet,
 - einem diskreten Bereich eines Objekttyps (werden wir später behandeln),
 - einem Elementnamen, der sich auf Aufzählungstypen bezieht (auch die werden wir im später behandeln) oder
 - einfach aus *others*, was grundsätzlich den Rest, also die nicht bestimmten Fallunterscheidungen meint
- bestehen.

Einige einfache Beispiele:

Die Variablen *state* und *act* seien vom Typ integer. Dann kann man z. B. formulieren

```

case state is
  when 0           => act := 0;
  when 1 to 7     => act := 1;
  when 15 downto 8 => act := 2;
  when others    => act := 3;
end case;

```

Obwohl noch nicht detailliert behandelt sei die Variable *state* ein Aufzählungstyp. Die Variable *act* bleibt wie gehabt. Dann kann man z. B. formulieren

```

type states is (idle, active, waiting, undefined);
variable state: states;
.....
case state is
  when active   => act := 0;
  when waiting => act := 1;
  when others => act := 3;
end case;

```

Weiterhin sollte angemerkt werden, dass im *case* stets alle Fälle abgedeckt werden müssen! Dazu ist einerseits das *others* notwendig und auf der anderen Seite muss noch eine Möglichkeit geschaffen werden, auch Fälle ohne einen Befehl oder eine Befehlsfolge abzudecken. Hierzu ist die leere Anweisung (kein Befehl) *null* eingeführt worden. Wir finden es unter den *sequential_statements*.

```

null_statement ::= [ label : ] null;

```

Das obige *case* kann *ohne das others* auch formuliert werden:

```

case state is
  when active       => act := 0;
  when waiting      => act := 1;
  when idle | undefined => act := 3;
end case;

```

oder, falls im *others keine Aktivität* geplant ist

```

case state is
  when active    => act := 0;
  when waiting  => act := 1;
  when others   => null;
end case;

```

Greifen wir unser Bitvektor-Beispiel wieder auf, so können wir folgendes Beispiel anführen, was den Ansatz gäbe aus jedem Bitmuster eine Integerzahl zu generieren. Dies ist jedoch für alle 256 Werte eine recht umfangreiche Lösung!

```

case d is
  when "00000000" => no1 := 0;
  when "00000001" => no1 := 1;
  when "00000010" => no1 := 2;
  when "00000011" => no1 := 3;
  .....
  when others => no1 := 255;
end case;

```

Auch hierzu wollen wir wieder das nebenläufige Vergleichskonstrukt betrachten. Wir finden es in der Definition der Signalzuweisung in der Alternative:

```

selected_signal_assignment ::=
  with expression select
    target <= options selected_waveforms;

```

Unter der Waveform Auswahl finden wir:

```

selected_waveform ::=
  { waveform when choices, }
  waveform when choices

```

Greifen wir das erste der obigen Beispiele wieder auf und zeigen das vergleichbare Konstrukt auf:

```

with state select
  act := 0 when 0,
  act := 1 when 1 to 7,
  act := 2 when 15 downto 8,
  act := 3 when others;

```

5.4 Iterationen / Schleifen; loop_Anweisungen

Schleifenkonstrukte machen natürlich nur Sinn, wenn man sequenziellen Programmfluss betrachtet. Es gibt zwar bei nebenläufigen Konstrukten auch Zählschleifen, aber die sind nur zur Konstruktionsvereinfachung da. Sie finden Verwendung zur Vermehrfachung gleicher Komponenten. Hier wollen wir nur den sequenziellen Steuerfluss betrachten.

Bei den Schleifenkonstrukten kann man im allgemeinen drei Typen unterscheiden, die Zählschleife, die abweisende und die annehmende Schleife. Alle drei Konstrukte sind in der folgenden EBNF enthalten. Aber VHDL erlaubt auch ein vollkommen verallgemeinertes Loop-Konstrukt, das wir am Ende behandeln.

```

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

```

In der Option [*iteration_scheme*] ist der Unterschied der drei Standard-Schleifenkonstrukte verborgen:

```
iteration_scheme ::=
  while condition
  | for loop_parameter_specification
```

Die abweisende **while**-Schleife ist in fast allen Sprachen vorhanden. Die annehmende **do { } while**-Schleife aus der C-Sprache ist in VHDL nur nachbildbar, wie wir noch sehen werden. Die **for**-Schleife (Zählschleife) ist in allen Sprachen bekannt.

VHDL (wie auch Ada) definiert eine offene, prinzipiell endlose Schleife, was offensichtlich mit den Möglichkeiten der Parallelprogrammierung zusammenhängt. Durch die zusätzlichen bedingten und unbedingten Anweisungen **next** und **exit** lassen sich aber prinzipiell alle Schleifenkonstrukte mit diesem einen *loop*-Konstrukt nachbilden.

Beginnen wir, weil diese die wohl am meisten verwendete Schleife ist, mit der *Zählschleife*.

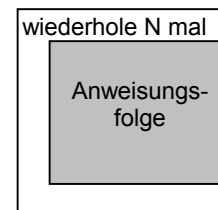
Zählschleife; **for_loop**

Die *for*-Schleife weist eine Parameter-Spezifikation aus, die definiert ist zu

```
parameter_specification ::= identifier in discrete_range
```

Das Struktogramm-Element dazu ist nebenstehend angegeben.

Man beachte die voll umgebende "Klammer" zur Kennzeichnung, dass man sie komplett kontrolliert, d.h. es wird definitiv die Anzahl der Durchläufe vorher festgelegt. Das *N* im Struktogramm gibt die Anzahl der Durchläufe an.



In der EBNF kann man dann die *for*-Loop angeben mit:

```
for-loop_Anweisung ::=
  [ loop_identifier:]
  for Zähler in diskreter_Bereich loop
  sequence_of_statements
  end loop [ loop_identifier ];
```

Für den *Zähler* gilt, dass er nicht wie in anderen Programmiersprachen jede beliebig definierte Variable sein kann, sondern er bildet einen eigenen Typ, der nur innerhalb seines Schleifenkörpers automatisch mit dem Schleifenkopf definiert ist.

Beispiele:

Bemühen wir als Erstes wieder unseren Praktikumsversuch 2. Hier hatten wir *d* als Zeichenkette deklariert. Das Schleifenkonstrukt soll 8 mal das niedrigstwertige Bit von *d* abfragen und danach jeweils *d* um eine Stelle nach rechts verschieben. Die Abfrage hatten wir bei der *if*-Anweisung gezeigt und den Rechtsverschub bei den Bitstring-Literalen aufgezeigt. Bleibt uns nur noch das Schleifenkonstrukt darum zu konstruieren. Der Zähler ist *k*:

```
for k in 1 to 8 loop           -- for Schleife
  if (d and "00000001") = "00000001" then
    n := n + 1;                 -- zähle die '1' en
  end if;
  d := '0' & d(7 downto 1);     -- verschiebe nach rechts
end loop;
```

Wenn Folgendes programmiert wird, so ist die Variable *a* im Schleifenkörper als Zähler ein andere Variable als die außerhalb definierte. Man sagt, im Körper ist die Variable *a* überdeckt, *b* jedoch nicht! Im Beispiel ist also nach dem Ablauf der Schleife *a* = 10 und *b* = 5!

```
example: process is
  variable a, b: integer;
begin
  a := 10;
  for a in 10 downto 5 loop
    b := a;
  end loop;
end process;
```

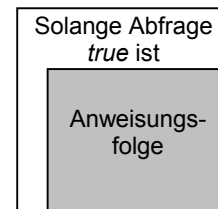
Ebenso kann der Zähler auch durch einen Aufzählungstyp gebildet werden, also geht auch

```
type state is (initial, idle, active, error);
variable b: state;
.....
for a in state loop
  b := a;
end loop;
```

Abweisende Schleife; while-loop

Das Struktogramm-Element dazu ist nebenstehend angegeben.

Man beachte die unten abgeschnittene "Klammer" zur Kennzeichnung, dass die Kontrolle den Schleifenkörper so abschirmt, dass der Körper auch *nicht* durchlaufen werden kann (abweisend).



Die EBNF dazu:

```
while-loop_Anweisung ::=
  [ loop_identifizier:]
  while Abfrage loop
    sequence_of_statements
  end loop [ loop_identifizier ];
```

Die Abfrage muss *true* ergeben, damit der Fluss in den Schleifenkörper eindringt und dieser wird solange durchlaufen, bis die Abfrage zu *false* entwickelt wird. Daraus folgt, dass nur bei *Abfrage = true* der Körper grundsätzlich ausgeführt wird und dass im Körper so programmiert werden sollte, dass sich die Abfrage zu *false* entwickelt, damit es zu einem Ausstieg aus dem Körper kommt.

Als Beispiel wählen wird die obige for-loop, bei der wir die Überdeckung der Variablen A im Schleifenkörper vermeiden können. Nach Ablauf der Schleife wird hier $a = 4$ und $b = 5$ sein!

```
alt_example: process is
  variable a, b: integer;
begin
  a := 10;
  while a >= 5 loop
    b := a;
    a := a - 1;
  end loop;
end process;
```

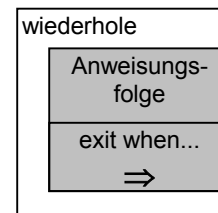
Wie oben schon angedeutet hat VHDL die Möglichkeit der allgemeinen Schleife, die als endlose Schleife operiert und nur durch spezielle Befehle zum Abbruch gebracht werden kann.

Allgemeine Schleife; loop_Anweisung

Das Struktogramm-Element dazu ist nebenstehend angegeben.

Man beachte die auch hier voll umgebende "Klammer" zur Kennzeichnung, dass man sie komplett kontrolliert und dass sie endlos läuft.

Diese Schleife kann nur durch das *exit* verlassen werden. Das *exit* kann aber beliebig oft an beliebiger Stelle im Schleifenkörper stehen.



Die EBNF dazu:

```
loop_Anweisung ::=
  [ loop_identifizier: ]
  loop
  sequence_of_statements
  end loop [ loop_identifizier ];
```

Die generell endlose Schleife kann nur mit Hilfe der zwei folgenden Sonderanweisungen verlassen werden:

```
sonder_Anweisungen ::= exit_statement | next_statement
```

Die *exit*-Anweisung ist nur in Schleifenkörpern (in allen) erlaubt und bewirkt den sofortigen Abbruch der Schleife. Sie ist natürlich nicht nur unbedingt möglich, sondern erlaubt auch den Abbruch unter einer Bedingung. Die EBNF:

```
exit_statement ::= [ label : ] exit [ loop_label ] [ when condition ];
```

Die *next*-Anweisung ist ebenfalls nur in Schleifenkörpern (in allen) erlaubt und bewirkt den sofortigen nächsten Durchlauf des Schleifenkörpers. Sie ist ebenfalls unbedingt und bedingt möglich. Die EBNF:

```
next_statement ::= [ label : ] next [ loop_label ] [ when condition ];
```

Als Beispiele versuchen wir zunächst, einige der obigen Schleifen als loop-Anweisung darzustellen. Man beachte, dass wir auch hier wieder etwas vorgreifen und Aufzählungen und Attribute anwenden, die wir erst bei den Datentypen genauer betrachten werden.

```
type state is (initial, idle, active, error);
variable a, b: state;
.....
a := state'left;
loop
  exit when a = state'right;
  b := a;
  a := state'succ(a);
end loop;
```

Hier wird nach Ablauf *b = active* und *a = error* sein. Will man aber erreichen, dass *a* und *b* auf dem gleichen Endwert stehen, so ist zu programmieren. Wir kehren die Reihenfolge jetzt um, so dass *a* und *b* jetzt *initial* sein werden:

```

type state is (initial, idle, active, error);
variable a, b: state;
.....
a := state'right;
loop
  b := a;
  a := state'pred(a);
  if a = state'left then
    b := a;
    exit;
  end if;
end loop;

```

Als Anwendung für die *next*-Anweisung wollen wir nur gerade Zahlen auf *b* zuweisen:

```

variable a, b: integer;
.....
a := 0;
loop
  a := a + 1;
  exit when a > 16;
  if a mod 2 = 0 then
    b := a;
  end if;
end loop;

```

Dies führt zum gleichen Ergebnis wie die folgende Schleife:

```

variable a, b: integer;
.....
a := 0;
loop
  a := a + 1;
  exit when a > 16;
  next when a mod 2 = 0;
  b := a;
end loop;

```

5.5 Unterprogramme und Packages

Mehrfach benötigte, gleiche Befehlsfolgen, die i.a. beschreibbare Teilaufgaben erfüllen, kann man wieder in einem Block zusammenfassen, mit einem Namen versehen und als Unterprogramm deklarieren. Allgemein sind Unterprogramme definiert zu:

```

subprogram_body ::=
  subprogram_specification is
    subprogram_declarative_part
  begin
    subprogram_statement_part
  end [ subprogram_kind ] [ designator ];

subprogram_specification ::=
  procedure identifier [ ( formal_parameter_list ) ]
  | function designator [ ( formal_parameter_list ) ]
  return type_mark

```

Man unterscheidet also Prozeduren und Funktionen. Beide sind über ihren Namen (*designator*) identifizierbar, wobei *designator* sowohl *identifizier* als auch *operator_symbol* sein kann. Damit sind auch alle Operatoren Unterprogramme. Das werden wir noch gesondert behandeln.

Der wesentliche Unterschied zwischen beiden ist zunächst das **return** bei der Funktion, denn Funktionen werden in Zuweisungen angegeben, so dass sie einen Wert zurückgeben müssen.

Der Deklarationsteil *subprogram_declarative_part* beinhaltet alle Deklarationen, die wir im folgenden Kapitel noch weiter betrachten werden. Signal- und Variablendeklarationen hatten wir bereits benutzt.

Der Ausführungsteil *subprogram_statement_part* besteht aus der Wiederholung *{sequential_statement}*, wie es aus dem Prozess bereits bekannt ist. Die Unterprogramm-Körper beinhalten also wie der Prozess sequenziellen Code.

5.5.1 Prozeduren und Funktionen

Prozeduren im Besonderen werden deklariert mit

```
procedure_subprogram_body ::=
  procedure identifier [ ( formal_parameter_list ) ] is
    subprogram_declarative_part
  begin
    subprogram_statement_part
  end [ procedure ] [ identifier ];
```

Die Deklaration besteht aus drei wichtigen Teilen: der Schnittstellen-Vereinbarung (*formal_parameter_list*) in Klammern als Option angegeben, dem Vereinbarungsteil *subprogram_declarative_part* und dem Anweisungsteil *subprogram_statement_part*. Wie bereits beim Prozess gesehen, ist der Anweisungsteil mit **begin** und **end** abgegrenzt. Davor liegt der Vereinbarungsteil mit den Variablenvereinbarungen und das Schlüsselwort **is** trennt den operationalen Teil von der Vereinbarung der Schnittstelle.

Unter dem Begriff **Schnittstelle** verstehen wir die Kommunikations-Struktur, die das Unterprogramm zur Kommunikation mit allen rufenden Programmen nutzt. Sie besteht zum einen aus dem Bezeichner **identifizier**, dem Aufruf-Name dieser Prozedur, und zum anderen aus einer Liste von sog. Parametern, die Informationen, die mit dem rufenden Programmteil ausgetauscht werden können. Die Parameter in der Vereinbarung heißen **Formalparameter**, die beim Aufruf einer Prozedur angegebenen sind die **Aktualparameter**. Die Schnittstellen-Vereinbarung im Syntaxdiagramm ist wie folgt:

```
formal_parameter_list ::= parameter_interface_list
interface_list ::= interface_element { ; interface_element }
interface_element ::= identifier_list : mode type_name
mode ::= in | out | inout
```

Als Parameter können Konstante, Variable und auch Signale fungieren. Die Parameterliste (*identifier_list*) ist also eine Aufzählung von Elementen getrennt durch “;”, wobei die Elemente aus einem oder mehreren Parameternamen getrennt durch “,” besteht, die mit einem Mode und Ihrem Datentyp nach einem “:” angeführt werden. Dies ist ähnlich zu anderen Sprachen, doch ist in VHDL und auch in Ada die Syntax wesentlich strenger durch die zwingende Angabe eines Übergabemodus *mode*.

Der Übergabemodus regelt klar die Verwendung des Parameters in Bezug auf das Unterprogramm. Modus **in** und **out** sind die bekannten Wertübergaben (*call by value* in C), d.h. der Wert einer Variablen wird als Konstante übergeben. Zusätzlich legt VHDL noch die Richtung der Übergabe fest und erklärt damit, ob der Parameter dem Unterprogramm übergeben wird (das ihn nicht verändern darf) oder ob der Parameter im Unterprogramm gesetzt wird, dann kann das rufende Programm ihn nur lesen und verwerten. Die Referenzübergabe (*call by name* oder *call by reference* in C) ist durch den Modus **inout** ausge-

drückt. Hier dürfen rufendes wie gerufenes Programm den Parameter in gleicher Weise lesend und schreibend nutzen.

Funktionen nehmen eine Sonderstellung ein, da sie als erweiterte Operation gesehen werden können. Demgemäß darf bei Funktionen in der Parameterliste *nur der Modus in* genommen werden, wobei man per Default das *in auch weglassen* kann. Entsprechend liefert eine Funktion ihr Ergebnis über den Funktionsnamen, wie bekannt aus der Trigonometrie z.B. $M = \sin(X)$ für die Nutzung der Sinus-Funktion. Hier ist *sin* der Funktionsname und *X* das Funktionsargument in der Programmiersprache ein Parameter. Entsprechend ist die Funktionsdeklaration etwas anders:

```
function_subprogram_body ::=
  function designator [ ( formal_parameter_list ) ]
    return type_mark is
  subprogram_declarative_part
begin
  subprogram_statement_part
end [ function ] [ identifier ];
```

Der Bezeichner *designator* hier kann auch ein Operatorsymbol sein, wie bereits erwähnt. Zudem wird in der Spezifikation der Typ des zurückzugebenden Wertes nach dem Schlüsselwort **return** festgelegt. Der Rest ist identisch mit der Prozedur. Es sollte aber angemerkt werden, dass im Ablaufteil mindestens einmal eine Wertrückgabe mit **return expression**; stattfinden muss! Expression muss zu dem in der Deklaration angegebenen Typ evaluierbar sein.

Im engen Zusammenhang mit Unterprogrammen steht die Modularisierung von Programmen. Bei größeren Projekten ist es sinnvoll, eigene, logisch zusammengehörende Programmteile wie Deklarationen und Unterprogramme in eigenen Dateien zu verwalten und möglicherweise auch für sich getrennt zu compilieren. Solche Teile werden in sog. Packages verwaltet, die wir im Anschluss noch betrachten werden. Dabei ist es aber für die Entwicklung wichtig, zum Testen zwar die Schnittstellen zu Unterprogrammen klar festgelegt zu haben, aber die Funktionalität der Routinen noch nicht. Ebenso können mehrere Implementierungen derselben Routine zum Test vorliegen. Die Sprache C sieht hier sog. *Funktions-Prototypen* vor, die lediglich Namen und Formalparameter-Struktur angeben. VHDL kennt hier die *Spezifikation (subprogram specification)* für Prozeduren und Funktionen. Diese gibt lediglich die *Schnittstelle* wieder und muß identisch mit der Unterprogramm-Deklaration sein. Während die Deklaration nach der *Schnittstelle* mit dem Schlüsselwort **is** (s.o.) weitergeführt wird endet die Spezifikation hier mit dem Semikolon. Der Hintergrund ist einfach zu erklären: Hat der Compiler die Schnittstelle, so kann er alle Aufrufe der Prozedur mit allen Aktualparametern generieren und vor allem überprüfen. Die EBNF der Deklaration gilt auch für Funktionen:

```
subprogram_declaration ::= subprogram_specification ;
```

Hatte man z.B. in einem Package eine Prozedur und eine Funktion wie folgt deklariert:

```
procedure myproc (a, b, c : in integer; y : out boolean; z : inout positive) is
  ....
begin
  ....
end procedure;

function myfunct (a, b, c : in positive) return boolean is
  ....
begin
  ....
  return true;
  ....
end procedure;
```

so sind die entsprechenden Spezifikationen (Prototypen) in den Programmteilen, die die Unterprogramme nutzen wie folgt anzugeben:

```
procedure myproc (a, b, c : in integer; y : out boolean; z : inout positive);
function myfunct (a, b, c : in positive) return boolean;
```

Aufrufe und Parametertausch

Interessant sind nun wieder die Aufrufe im Programm. Beginnen wir mit der Prozedur.

```
procedure_call ::= procedure_name [ ( actual_parameter_part ) ]
actual_parameter_part ::= parameter_association_list
association_list ::= association_element { , association_element }
association element ::= [ formal_part => ] actual_part
```

Dazu direkt ein paar Beispiele für die Aufrufe der obigen Prozedur *myproc*:

```
myproc (av, bv, cv, yb, zvar);           myproc (33, bv, 55, yb, zvar);
myproc (16*35, bv, cv, yb, zvar);      myproc (33, b, c, y, z);
```

sind erlaubte Aufrufe, wenn die Variablen auf den entsprechenden Formalparameterplätzen vom gleichen Typ deklariert wurden.

Die folgenden Aufrufe sind nicht erlaubt, wobei die Gründe klar sein sollten:

```
myproc (av, bv, cv, true, zvar);        myproc (33, bv, -55, yb, 25);
myproc (true, bv, cv, 20, zvar);       myproc (33, b, c, yb, -77);
```

VHDL verlangt, dass bei Aufrufen die Parameter-Reihenfolge strikt eingehalten wird. Will man davon abweichen, so nutzt man den oben angegebenen *formal_part*, in dem man den Formalparameternamen voranstellt mit dem => Pfeil:

```
myproc (a => av, b => bv, c => cv, y => yb, z => zvar);
myproc (b => bv, z => zvar, c => cv, y => yb, a => av);
```

sind zulässige Aufrufe.

Als ein kleines Beispiel zur Prozedurdeklaration und Aufruf betrachten wir das vorher behandelte Loop Beispiel mit unserem Bitvektor. Wir wollen eine Prozedur entwickeln, die den Bitvektor *d* um *nb* Stellen nach links (zu höheren Bitwertigkeiten), wenn *dir* '1' ist oder nach rechts (zu niedrigeren Bitwertigkeiten), wenn *dir* '0' ist, verschiebt. Dazu können wir wie folgt deklarieren:

```
-- vorher war deklariert
subtype myvector is bit_vector(7 downto 0);
variable d : myvector;
.....
procedure rotate (d : inout myvector; nb : in natural; dir : in bit) is
  variable merk : bit;
begin
  for i in 0 to nb loop
    if dir = '0' then
      merk := d(0);
      d := merk & d(7 downto 1);
    else
      merk := d(7);
      d := d(6 downto 0) & merk;
    end if;
```

```
end loop;
end procedure;
```

Die Verschuboperationen bei Stringvariablen wurden bereits behandelt. Neu an diesem Beispiel ist, die Einführung des Untertyps (*subtype*) `myvector`, der mit diesem Schlüsselwort von einem nicht längenbeschränkten String `bit_vector` abgeleitet wird. Wir werden das bei den Datentypen im nächsten Kapitel näher betrachten.

Der Aufruf dieser Prozedur in einem Prozess kann über die folgenden Aufrufbeispiel geschehen:

```
rotate(d, 3, '0') - rotiere d um 3 Stellen nach rechts
rotate(d, 1, '1') - rotiere d um 1 Stelle nach links
rotate(d=>d, dir=>'0', nb=>0) - rotiere d keine Stelle
```

Der Aufruf einer Funktion ist in der EBNF etwas schwieriger abzuleiten. Daher nur in kurzer Erläuterung: ein Ausdruck *expression* ist sehr komplex, weil er die Priorität der Operationen und Relationen darstellen muss. Ein Terminal des Faktors *fact* ist dann der *function_call*. Dieser ist wiederum dem *procedure_call* sehr ähnlich.

```
function_call ::= function_name [ ( actual_parameter_part ) ]
```

Allerdings taucht dieser Aufruf eben nur in Zuweisungen auf, wie z.B.

```
y := myfunct (av, bv, cv);           y := myfunct (33, bv, -55);
y := myfunct (b, c, 20);           y := myfunct (33, 44, -77);
```

Hier muss natürlich die Variable `y` vom Type *boolean* sein! Die Parameter können auch, wie oben angedeutet mit dem Formalparameternamen versehen und vertauscht werden. Man beachte, dass alle Parameter hier nur den Modus *in* haben dürfen!

Als Funktionsbeispiel betrachten wir die Umsetzung eines Bitvektors in eine entsprechende Integerzahl. Das Beispiel zeigt die Funktion deklariert im Bereich der Architektur, wo sie auch aufgerufen wird. Die Funktion hat also die Eigenschaft, dass sie ohne Zeitverzögerung Wertberechnungen vornimmt. In so fern kann eine Funktion sowohl für nebenläufige Signalzuweisungen, als auch in Prozessen für sequenzielle Signal- oder Variablenzuweisungen genutzt werden.

```
architecture base of convert is
  subtype myvector is bit_vector(7 downto 0);
  signal a: integer;
  signal b: myvector;

  function value(vect : myvector) return integer is
    variable val: integer := 0;
  begin
    for i in 0 to 7 loop
      val := val + a * 2**i;
    end loop;
    return val;
  end function;

begin
  b <= "00001111", "00100001" after 2 ns,
       "00111111" after 3 ns, "11111111" after 4 ns;

  a <= value(b);
end base;
```

Bei Funktionsvereinbarungen unterscheidet VHDL noch zwischen der **pure function** oder der **impure function**. Der Standard ist die **pure function**, was für die Funktion bedeutet, dass sie stets denselben Wert bei gleichen Eingabeparametern liefert. Bei einer **impure function** ist das nicht der Fall. Sie kann unterschiedliche Werte zurück liefern. Ein typisches Beispiel einer solchen Funktion ist die Berechnung einer Zufallszahl. Mit jedem Aufruf liefert sie einen Zufallswert. Eine wichtige und bei Simulation ggf. benötigte **impure function** ist **now**. Der Aufruf:

```
simtime := now;
```

liefert die aktuelle Simulationszeit zum Aufrufzeitpunkt, wenn *simtime* als Typ *time* vereinbart ist. Auch hier ändert sich der Rückgabewert mit jedem Aufruf.

Sichtbarkeitsregeln für Deklarationen in Unterprogrammen

Mit der Unterprogramm-*Schnittstelle* verbindet sich auch der Begriff der Sichtbarkeit von deklarierten Namen. Grundsätzlich wissen wir bereits, dass jede Architektur, jeder Prozess und jedes Unterprogramm seinen eigenen Deklarationsteil hat und dass alles hier Deklarierte im seinem lokalen Ausführungsteil sichtbar ist. Außerhalb dieses Bereiches, d.h. vor der Deklaration, in der Deklaration und nach dem Ausführungsteil kann zu deklarierten Namen nicht zugegriffen werden. Ebenso gilt für alle Prozeduren und Funktionen, die in einem Prozess deklariert sind, dass die im Prozess deklarierten Namen auch in den Unterprogrammen sichtbar sind. Das gilt natürlich auch für Prozesse, die in Architekturen deklariert sind. Nur so sind wir in der Lage, Signale in Prozessen zuzuweisen.

Wir wollen zunächst an einem allgemeinen Beispiel aus [PA02] die Sichtbarkeit erläutern:

```
architecture arch of ent is
  type t is ...;
  signal s : t;

  procedure p1 (...) is
    variable v1 : t;
  begin
    v1 := s;
  end procedure p1;

begin -- arch

  proc1:
  process is
    variable v2 : t;

    procedure p2 (...) is
      variable v3 : t;
    begin
      p1 (v2, v3, ...);
    end procedure p2;

  begin -- proc1
    p2 (v2, ...);
  end process proc1;

  proc2:
  process is
    ....
  begin -- proc2
    p1 (...);
  end process p2;

end architecture arch;
```

Das Beispiel zeigt eine Typdefinition und einige Deklarationen. Der Sichtbarkeitsbereich der einzelnen Deklarationen ist angezeigt.

Zunächst enthält die Architektur die Typvereinbarung eines Typs *t*, der über die gesamte Architektur sichtbar ist. Das gilt auch für das Signal *s* und die Prozedur *p1*, die im Deklarationsteil der Architektur aufgeführt ist. Der Ausführungsteil der Architektur besteht aus zwei Prozessen, *proc1* und *proc2*.

Die **Prozedur p1** vereinbart eine Variable *v1*, die nur innerhalb dieser Prozedur lokal sichtbar ist. Der Typ *t* darf verwendet werden, da er auch innerhalb von *p1* sichtbar ist. Da das Signal *s* auch innerhalb von *p1* sichtbar ist, ist die Zuweisung *v1 := s*; zulässig und korrekt. Die Prozedur *p1* kann über den gesamten Sichtbarkeitsbereich der Architektur aufgerufen werden, also von beiden Prozessen in seinem Ausführungsteil.

Der Prozess **proc1** erklärt eine Variable *v2* vom Typ *t*, da *t* auch hier sichtbar ist. Diese Variable *v2* ist für den gesamten Prozess vereinbart und damit auch in der eingebetteten Prozedur *p2* sichtbar. Die für den Prozess *proc1* lokale Prozedur *p2* kann nur hier aufgerufen werden. Die Prozedur *p2* kann *p1* aufrufen, da *p1* auch innerhalb von *p2* sichtbar ist. Sie kann auch, wie angedeutet, als Parameter ihre lokale Variable *v3* als auch die Variable *v2* des Prozesses übergeben, da sie hier sichtbar ist.

Der nur kurz angedeutete Prozess **proc2** kann zum Typ *t*, zum Signal *s* und zur Prozedur *p1* zugreifen, nicht aber zur Prozedur *p2*, die im Prozess *proc1* verborgen ist.

Im obigen Beispiel haben wir noch nicht betrachtet, dass Unterprogramm auch geschachtelt werden können, d.h. ein Unterprogramm deklariert lokal ein weiteres. In diesem geschachtelten Unterprogramm gelten natürlich die gleichen Sichtbarkeitsregeln, doch welchen Effekt hat die Vereinbarung unter dem gleichen Namen? In Anlehnung an [PA02] auch dazu das folgende Programmgerüst:

```

procedure p1 is
  variable v : integer;

  procedure p2 is
    variable v : integer;
    begin -- p2
      .....
      v := v + 1;
      .....
    end procedure p2;

  begin -- p1
    .....
    v := 2 * v;
    .....
  end procedure p1;
    
```

Die **Prozedur p1** vereinbart eine Variable *v* und eine lokale Prozedur *p2*. Die Prozedur *p2* sieht zwar die Variable *v* aus *p1*, vereinbart aber eine eigene Variable *v*, die sie lokal verändert.

Der Effekt dieser Situation ist, dass die lokal vereinbarte Variable *v* die sichtbare von *p1* überdeckt, d.h. die lokale Zuweisung hat auf die Variable *v* in *p1* keinen Effekt. Vor einer unbeabsichtigten Programmierung in der Weise sei also gewarnt.

Da aber VHDL wegen seines Grundkonzepts - Vereinbarungen in einer Architektur auch in Prozessen ohne eigene Vereinbarungen anzuwenden - mehr auf eine globale Sichtbarkeit zielt, hat man in der obigen Situation noch die Möglichkeit des sog. "*visibility by selection*" anzuwenden. Trotz der lokalen Verdeckung der Variablen *v* in *p2* kann *p2* die Variable *v* von *p1* zugreifen, nämlich durch Vorstellen des Prozedurnamens. In *p2* kann somit durch z.B.

p1.v := v + 33;

die Variable *v* von *p1* verändert werden.

Diese Möglichkeit geht nur in geschachtelten Routinen zu den deklarierenden Routinen, nicht umgekehrt, d.h. in *p1* oder *außerhalb von p1* ist ein solches Konstrukt für den Zugriff auf *v* in *p2* nicht möglich!

Das folgende, praktische Beispielprogramm soll Anwendung von Prozedur und Funktion zeigen und die Situation der Sichtbarkeit verdeutlichen.

```

entity visibility_1 is
end visibility_1;

architecture algo of visibility_1 is
begin

visibility:
process is
variable A, B : integer := 200;
variable Var_Out : integer;
constant Fak : integer := 55;

procedure InnerA (Ab, Ac : in Positive; Re : inout integer) is
variable Var_Out : positive;
variable Var_In : integer := 333;
begin -- InnerA
Var_Out := Ab * Ac + Re;
Re := Var_Out + Var_In;
end InnerA;

function InnerB (Bb, Bc : in natural) return integer is
-- procedure InnerB (Bb, Bc : in natural) is
-- variable Var_Out : positive;
-- variable A : positive;
begin -- InnerB
Var_Out := Bb * Bc;
A := Bb + Bc;
return A;
end InnerB;

-- variable Var_Out : integer;
begin -- of process
Var_Out := 30;
InnerA (50, 70, Var_Out);
-- InnerB (11 * 34, Fak); -- procedure call
B := InnerB (11 * 34, Fak); -- function call
wait;
end process;
end architecture algo;

```

Sichtbarkeit von Parametern; sequenzieller Prozeduraufruf

Die Architektur besteht aus einem einzigen Prozess, der mit einem *wait* endet. Somit startet er mit der Initialisierung und endet nach einem Durchlauf. **A** ist eine Variable die der Prozess als *integer* deklariert und mit dem Wert *200* initialisiert. Die Variable **B** ist in gleicher Weise deklariert und dient nur dem korrekten Funktionsaufruf, wenn dieser benutzt wird. Der Prozess deklariert zwei auf gleicher Ebene geschachtelte Prozeduren **InnerA** und **InnerB**. *InnerB* kann auch als Funktion umgestaltet werden. Beide sind im Deklarationsteil des Prozesses erklärt und haben ihrerseits im Vereinbarungsteil keine weiteren Unterprogramme vereinbart. Beide Unterprogramme könnten sich gegenseitig aufrufen. Der Prozess vereinbart zudem die Variable *Var_Out* ebenfalls vom Typ *integer*. Die gleiche Deklaration ist auskommentiert vor dem eigenlichen Ausführungsteil des Prozesses noch einmal aufgeführt. Kehrt man die Auskommentierung so um, dass die Deklaration dieser Variablen erst nach der Deklaration der Unterprogramme erfolgt, akzeptiert der Compiler das nicht, denn für die Unterprogramme ist diese globale Variable noch nicht sichtbar!

Wird *Var_Out* vor den beiden Unterprogrammen deklariert – dies ist die übliche Form -, kennen beide Routinen diese Variable als globale Variable. Die Prozedur (Funktion) *InnerB* verändert diese Variable, was für einen Programmierer, der nicht generell den Stil der globalen Variablennutzung anwendet, nicht

ganz offensichtlich ist. *InnerA* definiert eine eigene *Variable mit dem gleichen Namen Var_Out*, eine sog. lokale Variable. Dies führt dazu, daß hier die globale Variable gleichen Namens „überdeckt wird“. Hier entsteht *keine Veränderung der Variablen gleichen Namens im Prozess*. Die auskommentierten Deklarationen in der Prozedur *InnerB* können sukzessive einbezogen werden, so dass man am Ergebnis für *A* und *Var_Out* die Reaktion ablesen kann. Die Prozedur *InnerB* kann auch als Funktion deklariert und genutzt werden, indem man die entsprechenden Kommentare ändert.

Nebenläufiger Prozeduraufruf

Wir haben oben bei den Betrachtungen der Steuerkonstrukte Sequenzialität und Nebenläufigkeit zu vergleichen. Wir haben auch gesehen, dass nebenläufige Signalzuweisungen eigentlich Kurzformen von Prozesskonstrukten sind. (siehe Seiten 23-25). Ähnliches kann man auch mit Prozeduren machen. Der folgende Prozess enthält einen sequenziellen Prozeduraufruf zu einer in der Architektur deklarierten Prozedur. Der Aufruf hat als Übergabeparameter nur Signale (hier z.B. *s1* und *s2*), die nur im *in* oder *inout* Modus auftreten und ggf. auch Konstante (hier *val1*), die naturgemäß nur im *in* Modus auftreten. Man beachte, dass der Prozess auf *s1* und *s2* sensitiv ist.

```

process is
begin
  mypro (s1, s2, val1);
  wait on s1, s2;
end process;

```

Ein solches Konstrukt zeigt in einer Architektur das gleiche Verhalten wie der folgende, nebenläufige Prozeduraufruf, der eine Kurzform der Prozessdeklaration darstellt:

```

mypro(s1, s2, val1);

```

Der nebenläufige Aufruf unterscheidet sich primär nicht von dem sequenziellen, doch sollte man sich die Einschränkungen der Parameter klar machen.

Da die Aufrufe nebenläufig sind, ist einsichtig, dass keine Variablen als Parameter auftreten.

Bezüglich der Signalparameter ist ersichtlich, dass in nebenläufige Prozeduren nur solche im Modus *in* oder *inout* Sinn machen, da nur sie einen nebenläufigen Prozeduraufruf auslösen.

Der nebenläufige Prozeduraufruf ist dem Prozess mit Sensitivitätsliste sehr ähnlich, doch ist zu beachten, dass alle Variablendeklarationen innerhalb von Prozeduren bei jedem Aufruf neu initialisiert werden, während Prozesse die Variablenwerte halten!

Prozeduren dürfen auch wait-Statements enthalten. Man sollte aber beachten, dass solche Prozeduren nicht in Prozessen mit Sensitivitätsliste aufgerufen werden dürfen. In Funktionen sind wait-Statements nicht erlaubt, da Funktionen in VHDL dazu dienen, unmittelbar Ergebnisse zu liefern.

Als eine Aufgabe zu Selbststudium kann das oben gezeigte Programm zu Verdeutlichung der Sichtbarkeit so umgeschrieben werden, dass ein nebenläufiger Prozeduraufruf für **Outer** entsteht. Die Sichtbarkeitsregeln bleiben gültig. Kurzanleitung dazu:

```

entity visibility_1 is
end visibility_1;

architecture algo of visibility_2 is
  signal as, bs;

  procedure outer(signal as, bs: inout integer) is
    -- Deklarationen und Schachtelungen
  begin -- outer
    ....
  end procedure outer;

begin -- architecture
  outer(as, bs);

```

```
end architecture algo;
```

Beispiel (fragmentarisch) zum nebenläufigen Prozeduraufruf

Eine interessante Applikation nebenläufiger Prozeduraufrufe stammt aus [PA02]. Die Anwendung von nebenläufigen Prozeduren ist gerade dann nützlich, wenn mehrere gleichartige Prozesse zu programmieren sind. Das Programm erzeugt einen Zweiphasentakt, wobei alle Parameter der Takte vorgebar sind. Die Prozedur *generate_clock()* wird zweimal instanziiert zur Erzeugung der Signale *phi1* und *phi2*. Jedes Signal hat eine Periodendauer *Tper* und eine Pulszeit (Signal = '1') von *Tpul*. Der Phasenverschub ist mit *Tpha* anzugeben. Interessant hier ist, dass die Prozeduren keine Sensitätsliste benötigen, da sie mit der Initialisierung starten und dann endlos in einer Schleife laufen, in der sie jeweils die Eins- und Nullphase des Taktes auf den Signaltreiber schreiben.

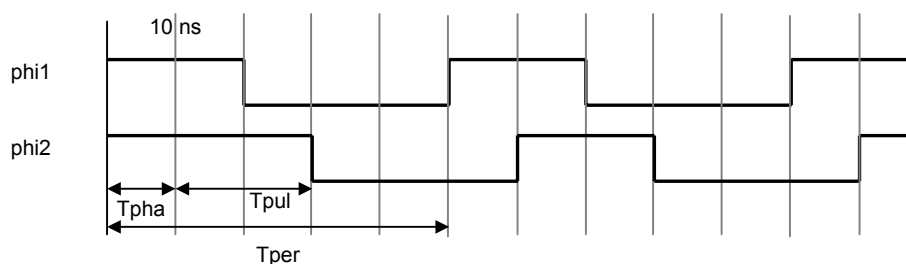
```
entity proc2clk is
end proc2clk;
architecture algo of proc2clk is
  signal phi1, phi2 : bit;

  procedure generate_clock(signal clk : out bit;
                           constant Tper, Tpul, Tpha: in time) is
  begin -- of procedure generate_clock
    wait for Tpha;
    loop
      clk <= '1', '0' after Tpul;
      wait for Tper;
    end loop;
  end generate_clock;

begin -- architecture
  gen_phi1: generate_clock(phi1, Tper => 50 ns, Tpul => 20 ns, Tpha => 0 ns);
  gen_phi2: generate_clock(phi2, Tper => 50 ns, Tpul => 20 ns, Tpha => 10 ns);
end architecture algo;
```

Zweiphasentakt mit nebenläufigen Prozeduraufrufen

Das Simulationsergebnis sieht folgendermaßen aus:



5.5.2 Package Konzept

Das Package Konzept ermöglicht es, Daten und Unterprogramme eines Modells zu verwalten. Ein Package sollte funktional zusammengehörnde Vereinbarungen gruppieren und unter einem aussagekräftigen Namen ablegen. Zusammenfassungen könnten auch eine Gruppe von Unterprogrammen zur Behandlung bestimmter Daten sein. Packages sind somit eigene Entwurfseinheiten, die unabhängig voneinander bearbeitet werden können und die für mehrere, verschiedene Modelle einsetzbar sind. Das legt zunächst den Grundstein zur Wiederverwendbarkeit von Modellen und Modellteilen, denn einmal compilierte Einheiten können von allen Programmen benutzt werden, wenn ihre Schnittstelle be-

kannt ist. Mit dem Package Konzept ist vor allem eine Trennung von Vereinbarungsteil und Implementierung von Modellen möglich.

Packages sind vergleichbar mit den Include-Files von C. Was in C unter dem Namen Header-Files verbirgt, sind hier die Package-Vereinbarungen, die keine Unterprogramm-Vereinbarungen enthalten. Solche Packages, die Prozedur- oder Funktions-Prototypen enthalten, benötigen den sog. package body, der letztendlich die vollständigen Unterprogramme enthält, die natürlich in den Formalparameterlisten konform zu den Prototypen sein müssen.

Zunächst wieder die formale Seite:

```
package_declaration ::=
package identifier is
  { package_declarative_item }
end [ package ] [ identifier ];
```

Im Wesentlichen enthält der Vereinbarungsteil (package_declarative_item) Typ-, Signal-, Variablen-, Konstanten- und weitere Vereinbarungen, deren Vollständigkeit im LRM nachzulesen ist. Wie oben erwähnt, ist die Unterscheidung wichtig, ob dieser Teil auch Unterprogramme vereinbart. Ist das der Fall, muss das Package auch einen Rumpf (package_body) haben, in dem die Unterprogramme vollständig beschrieben sind. Der Package-Rumpf muss den gleichen Namen haben und unterscheidet sich nur durch das Schlüsselwort:

```
package_body ::=
package body identifier is
  { package_body_declarative_item }
end [ package ] [ identifier ];
```

Packages sind im allgemeinen Bestandteile von Bibliotheken, die man referieren muss. In allen Modellen, incl. denen im Praktikum, wurden bereits implizit Bibliotheken verwendet. Diese sind STD und WORK. Die Bibliothek STD enthält unter anderem alle bisher benutzten Typ-Definitionen. Die Bibliothek WORK ist die, die für unsere Umgebung alle Entwurfseinheiten aufnimmt. Da diese standardmäßig enthalten sind, ist die direkte Referenz dazu mit der library clause

```
library STD, WORK;
```

nicht explizit nötig. Verwenden wir aber andere Bibliotheken, so müssen wir diese Referenz anführen, wie z.B. im Praktikumsversuch 3, der die IEEE Bibliothek als Referenz vorgibt mit:

```
library ieee;
```

Alle Packages sind in solchen Bibliotheken abgelegt und man kann darauf zugreifen, in dem man ihren Namen als selected_name angibt:

```
selected_name ::= name. ( identifier | character_literal | operator_symbol | all )
```

In unserem Beispielfall aus dem Praktikum könnte man auf Datentyp std_logic nur zugreifen mit der vollständigen Referenz, also z.B.

```
variable carry: ieee.std_logic_1164.std_logic;
```

Um sich die umfangreiche Schreibarbeit und Unübersichtlichkeit im Programmkontext zu ersparen, hat VHDL die sog. **use clause**. Führt man diese mit an, so macht diese Klausel hier den Typ std_logic sichtbar, kann man einfach mit std_logic referieren. Weiter noch verallgemeinern kann man durch all, womit dann alle Deklarationen im Package so referiert werden können. Also führt die Kombination aus den beiden Klauseln dazu, dass folgendes möglich ist:

```
library ieee;
use ieee.std_logic_1164.all;    -- use ieee.std_logic_1164.std_logic
.....
variable carry: std_logic;
.....
```

Da in unserer Entwicklungsumgebung die Arbeitsbibliothek implizit vorgegeben ist, nutzen wir nur die *use clause*, wenn wir z.B. in den Testbenches die Entitäten von Testmodellen aus der work Bibliothek. Versuch 1 hatte folgende Vorgabe:

```
for DUT: eq use entity work.eq(arch1);
```

Dies ermöglicht dann den verkürzten Zugriff bei der Aufstellung der Portmap:

```
DUT: eq port map(x, y, z);
```

Man beachte, dass hier in *work* kein Package, sondern die Entity angesprochen wird.

5.6 Datenobjekte

Wir wollen uns jetzt der Speicherung und Verarbeitung von **Datenobjekten** in VHDL zuwenden. Mit *Objekt* bezeichnet man allgemein in Programmiersprachen *die Einheit bestehend aus einem Ort und einem damit verbundenen Wert*. Die Objekt-Werte (Daten) - in objektorientierten Sprachen sind das die Zustände eines Objekts - werden typisiert und können als einfache (elementare) Typen bestehen, wie wir sie bereits mit Integer, Float u.s.w. genutzt haben, können aber auch komplexer beschaffen sein, wie als Zeiger (Zugriff) auf weitere Objekte oder eine Zusammenfassung von elementaren Typen (kompositorisch) sein, wie es im Bild 4.2 unten dargestellt ist.

⇒ Im Kontext dieses Skripts werden wir aus Traditionsgründen statt des Begriffs Objektwert häufig auch die Begriffe Daten oder Objektdaten als ein Synonym verwenden!

Objekte können weiterhin klassifiziert werden. In VHDL unterscheiden wir vier Objektklassen, nämlich **Konstante**, **Variable**, **Signale** und **Dateien**, die sich in der Art und Ausführung ihrer Wertzuweisung wesentlich voneinander unterscheiden.

5.6.1 Datentypen

Wir wollen zunächst einen allgemeinen Überblick über die Grundtypisierung der Objekte geben. Vielfach findet man auch noch die Bezeichnung Datentype. Die folgende Grafik im Bild 4.2 ordnet die Typen ein.

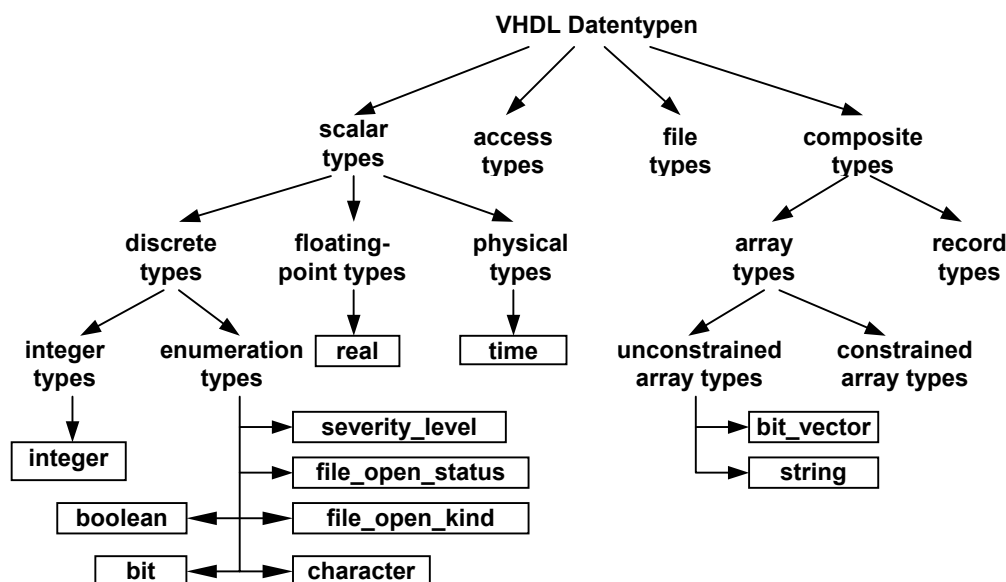


Bild 4.2: Objekttypen in VHDL

Die im Bild als *scalar* bezeichneten Objekttypen wurden bisher bereits mehrfach als *integer* oder *bit* verwendet. Dies gilt auch für *Aufzählungstypen*, so daß wir hier in Kap. 4.3.2 bei näherer Betrachtung der *Type* Vereinbarung noch einmal darauf zurückgreifen werden.

Zugriffstypen sind zwar Bestandteil der Sprache, haben aber, obwohl sie eine zentrale Rolle in der modernen Programmierung wegen der Möglichkeiten zur dynamischen Speicherverwaltung spielen, für unsere Nutzanwendungen weniger Bedeutung.

Bei den kompositorischen Typen Feld und Record werden wir nur die Felder behandeln, da Records für den Entwurf nicht von Bedeutung sind.

Wie man sich denken kann, nehmen für unsere Anwendungen zur Hardware-Beschreibung und zur Simulation die skalaren Datentypen eine besondere Stellung ein und sie werden auch noch gegenüber den in Sprachen wie C oder Ada zu erweitern sein.

5.6.2 Typ Vereinbarungen

VHDL ist eine streng typisierte Sprache, d.h. jedes Objekt darf nur Werte vom vorher definierten Typ annehmen. Das gilt natürlich auch für Operationen auf Daten, die nur vom gleichen Typ sein dürfen. Ist dies nicht der Fall, muss eine Typkonvertierung (type conversion, casting) vorgenommen werden.

Zunächst die Vereinbarung von skalaren Typen, bevor dann zur Typisierung von kompositorischen Typen übergegangen wird.

Die **TypVereinbarung** in VHDL läßt sich in der EBNF wie folgt darstellen:

```
Typ_Vereinbarung ::= type Bezeichner is Typ_Definition;
```

Da in VHDL, übrigens wie auch in Ada, alle skalaren Datentypen aus Aufzählungen hergeleitet werden, ist die Aufzählung bzw. der Typ auch einzuschränken durch das Schlüsselwort **range**. Der Compiler erkennt dann an der Formulierung *range*, daß eine begrenzte, ganzzahlige Aufzählung folgt. Ein Beispiel:

```
type MeinTyp is range 1 to 100;
```

Dieser Typ beinhaltet die Aufzählung ganzer Zahlen von 1 bis 100. man beachte hier aber die klare und eindeutige Typisierung, denn hier *MeinTyp* ist ein eigenständiger Datentyp.

5.6.3 Typ-Verträglichkeiten und Konvertierungen

Vereinbart man, wie unten, *MeinTyp* und *DeinTyp*, so haben beide Typen, wenn sie so deklariert sind, keine Gemeinsamkeit, d.h. Variablen vom Typ *MeinTyp* können nicht ohne Typ-Konvertierung zu Variablen vom Typ *DeinTyp* zugewiesen werden. Hier muß also korrekt ein Konvertierung programmiert werden:

```
type MeinTyp is range 1 to 100;  
type DeinTyp is range 3 to 95;
```

```
VarA : MeinTyp;
```

```
VarB : DeinTyp;
```

```
...
```

```
VarA := MeinTyp(VarB);
```

```
...
```

Oder ein weiteres Beispiel:

```
type apples is range 0 to 100;  
type oranges is range 0 to 100;
```

Verknüpft man Variablen (Signale) beider Typen, so muss das Ergebnis und einer der Operanden definitiv konvertiert werden. Sei z.B. die Variable A vom Typ *oranges* und B und C vom Typ *apples*, dann ist die Addition nur durch

```
C := apples(A) + B;
```

legal durchführbar.

Verträglich wäre *DeinTyp* oben nur, wenn er als **subtype** abgeleitet wird. Mathematisch gesehen bedeutet das, dass Objekte nur eine Teilmenge des Datentyps annehmen sollte. Die **Subtype** Vereinbarung kann wie folgt in der EBNF angegeben werden:

```
Subtyp_Vereinbarung ::= subtype Bezeichner is Subtyp_Indikator;
Subtyp_Indikator ::=
    Basistyp[range Ausdruck (to | downto) Ausdruck]
```

DeinTyp muss also dann definiert werden zu

```
subtype DeinTyp is MeinTyp range 3 to 95;
```

Wenn wir jetzt das obige Beispiel nehmen, und z.B. setzen

```
type apples is range 0 to 100;
subtype oranges is apples range 0 to 90;
```

dann gilt bei gleicher Variablenvereinbarung wie oben für die Addition

```
C := A + B;
```

Man beachte hierbei, dass das die Wertehaltung automatisch von der Arbeitsumgebung kontrolliert wird. Bei Über-/Unterschreitungen erfolgt eine entsprechende Ausnahmebehandlung (exception handling).

Wir sollten hier vorab auf eine Besonderheit aufmerksam machen. Wir sahen in den ersten Beispielen, dass die Entities die Vereinbarungen der Portsignale vom Typ *bit* beinhaltet. Sollen hier eigene Typvereinbarungen aufgenommen werden, ist dies nur durch eine eigene *Package* Vereinbarung möglich, um dies allen Modellen zugänglich zu machen. Wir werden die Bedeutung von Packages noch später näher betrachten.

Ein Anwendungsbeispiel dafür ist

```
package byte_type is
    type byte is range 0 to 255;
end package int_types;
```

bei dem ein Typ *byte* deklariert wird. Die Benutzung in einer Entity ist dann wie folgt gegeben:

```
use work.byte_type.all;

entity small_adder is
    port(a, b: in byte; s: out byte);
end entity small_adder;
```

die *use*-Klausel referiert damit das Package mit allen enthaltenen Typvereinbarungen ".all".

5.6.4 Skalare Datentypen

Beginnen wir die Betrachtungen zunächst mit dem Grundtypus Integer.

Integer Typen

Die Integer-Typen sind in VHDL vordefiniert und beinhalten primär alle positiven und negativen Ganzzahlen im Darstellungsbereich der jeweiligen Implementierung, bei 32-bit Darstellungen also von $-2^{31} \leq x \leq (2^{31}-1)$.

Mit der Wertbeschränkung (sog. range constraint) kann man aus dem vordefinierten Integer-Typ eigene Typen ableiten. In EBNF mit

```
Integer_Typdefinition ::=
    range Ausdruck (to | downto) Ausdruck
```

dazu das folgende Beispiel

```
type day_of_month is range 0 to 31;
type year is range 0 to 2100;
```

damit können nun Variablen z.B. wie folgt deklariert werden:

```
variable today: day_of_month := 9;
variable start_year: year := 2003;
```

Obwohl jetzt eine gewisse Überdeckung zwischen beiden Variablen besteht, ist die Zuweisung
`start_year := today;`

nicht korrekt und bedarf einer Typkonvertierung, wie oben angedeutet.

Um Type-Vereinbarungen in ihren Grenzen variabel zu gestalten, ist die Definition von Konstanten ebenfalls möglich. Will man z.B. die Bits eines Wortes von 0 bis zu der Bitanzahl pro Wort indizieren, so kann man wie folgt vorgehen:

```
constant no_of_bits: integer := 32;
type bit_index: is range 0 to no_of_bits-1;
```

Für Integer-Typen sind die folgenden Primitiv-Operationen definiert:

+	Addition	-	Subtraktion
*	Multiplikation	 	Division
mod	Modulo	rem	Restklasse
abs	Absolutbetrag	**	Exponentiation

Wichtig ist hier vor allem die Unterscheidung zwischen **mod** und **rem**. Hier kann man prinzipiell folgende Regeln beachten:

Bei **rem** richtet sich das Vorzeichen des Ergebnisses nach der Variablen A. Die **rem** Operation entstammt der Erfüllung der Operation:

$$A = (A / B) * B + (A \text{ rem } B) \qquad \text{bzw.} \quad A \text{ rem } B = A - (A/B)*B$$

Damit ist $(A \text{ rem } B)$ im Beispiel:

$$5 \text{ rem } 3 := 2, \quad (-5) \text{ rem } 3 = -2, \quad 5 \text{ rem } (-3) = 2; \quad (-5) \text{ rem } (-3) = -2$$

Diese Definition ist die des Divisionsrestes, wobei das Vorzeichen des Dividenden maßgeblich ist. Der Divisor wird als Absolutwert angenommen.

Bei **mod** richtet sich das Vorzeichen des Ergebnisses nach dem Modul, also der Variablen B. Die **mod** Operation entstammt der Erfüllung der Operation:

$$A = B * N + (A \text{ mod } B) \qquad \text{bzw.} \quad A \text{ mod } B = A - B * N$$

Damit ist $(A \text{ mod } B)$ im Beispiel:

$$5 \text{ mod } 3 = 2, \quad (-5) \text{ mod } 3 = 1, \quad 5 \text{ mod } (-3) = -1, \quad (-5) \text{ mod } (-3) = -2$$

Das Ergebnis hat das Vorzeichen des Moduls (B) und sein Absolutwert muss kleiner als der Absolutwert von B sein.

Beachte:

Für $B > 0$ und $A > 0$, wie für $B < 0$ und $A < 0$ ist $f_{\text{rem}}(A,B) = f_{\text{mod}}(A,B)$.

Für $B > 0$ und $A < 0$ gilt $f_{\text{mod}}(A,B) = B - f_{\text{rem}}(A,B)$ und

für $B < 0$ und $A > 0$ gilt $f_{\text{mod}}(A,B) = -(B - f_{\text{rem}}(A,B))$.

Gleitkomma Typen

Gleitkommazahlen repräsentieren reelle Zahlen. Aufgrund der beschränkten Stellenanzahl zur Zahlendarstellung in einem Digitalrechner lassen sich reelle Zahlen natürlich nur angenähert darstellen. Zur Darstellung wird die IEEE-Norm 754 oder 854 angewandt und sie werden mit 64 bit gespeichert. Damit überdecken sie mit ca. 15-stelliger Dezimalgenauigkeit den Zahlenbereich von $-1,8 \cdot 10^{308}$ bis $+1,8 \cdot 10^{308}$.

Die Gleitkomma-Typdefinition ist ähnlich der Integer-Typdefinition, d.h. die Unterscheidung erfolgt erst durch den Ausdruck. Während bei der Integer Definition reine Ganzzahlen angegeben werden, ist hier das Kennzeichen der **Dezimalpunkt** im Ausdruck.

```
Gleitkomma_Typdefinition ::=
    range Ausdruck (to | downto) Ausdruck
```

Zwei Beispiele dazu:

```
type input_level is range -10.0 to +10.0;
type probability is range 0.0 to 1.0;
```

Variablen, die von diesem Typ vereinbart werden, erhalten zu Anfang den links stehenden Wert in der Definition, also z.B.

```
variable input_A: input_level;
```

erhält hier die Variable input_A zu Beginn den Wert -10.0.

Physikalische Typen

Da VHDL zum Ziel hat, Hardware zu beschreiben, müssen natürlich auch Zahlen mit physikalischer Bedeutung existieren, wie z.B. Länge, Masse, Zeit und Strom. Physikalische Datentypen haben *eine primäre* Maßeinheit und können auch eine oder mehrere *sekundäre* Maßeinheiten beinhalten. Vereinfacht kann die Definition wie folgt aussehen:

```
Physikalische_Typdefinition ::=
    range Ausdruck (to | downto) Ausdruck
    units
    Bezeichner;
    { Bezeichner = Physikalisches_Literal }
    end units [Bezeichner]
Physikalisches_Literal ::= [ Dezimales_Literal | Basistyp_Literal ] unit_Name
```

Diese Definition ist wie eine reine Integer Definition, nur dass sie um Einheiten (**units**) ergänzt wird.

Einige Beispiele sollen die Benutzung verdeutlichen:

Widerstände:

```
type resistance is range 0 to 1E9
    units
    ohm;
    end units resistance;
.....
variable R1, R2, R3: resistance;
.....
R1 := 5 ohm; R2 := 22 ohm; R3 := 471_000 ohm;
```

Natürlich sind bei resistance auch sekundäre Maßeinheiten möglich, wie Folgendes für Längenangaben.

Längen:

```
type length is range 0 to 1E9
    units
    um;                -- primary unit: micron
    mm = 1000 um;      -- metric units
    m = 1000 mm;
    inch = 25400 um;   -- English units
    foot = 12 inch;
    end units length;
```

```
.....
variable M1, M2, M3: length;
```

```
.....
M1 := 23 mm; M2 := 2 foot; M3 := 9 inch;
```

Bei Definition von *secodary units* ist es auch möglich, Bruchzahlen in *primary units* anzugeben. Bei solchen Bruchzahlen wird immer auf die nächst niedrigere Einheit gerundet. Zur Verdeutlichung dieses Sachverhalts die folgenden drei Zuweisungen, bei denen alle drei Werte gleich *2540 um* sind:

```
M1 := 0.1 inch; M2 := 2.54 mm; M3 := 2.540526 mm;
```

In Zuweisungen können physikalische Typen nur mit gleichen Typen oder mit integer oder real verarbeitet aber nicht potenziert werden. Einige Beispiele:

```
5 mm * 6 = 30 mm; 18 kohm / 2.0 = 9 kohm; 33 mm / 22 mm = 1500;
```

Es können auch Absolutbeträge der physikalischen Typen angegeben werden:

```
abs 2 foot = 2 foot; abs (-2 foot) = 2 foot;
```

Zeiten:

```
type time is range implementierungsabhängig
```

```
units
```

```
fs;           -- femtosecond
ps = 1000 fs; -- picoseconds
ns = 1000 ps; -- nanoseconds
us = 1000 ns; -- microseconds
ms = 1000 us; -- milliseconds
sec = 1000 ms; -- seconds
min = 60 sec;  -- minutes
hr = 60 min;   -- hours
```

```
end units time;
```

Für VHDL Modelle ist dies der wichtigste *physical type*, da alle Modelle zeitabhängig sind und damit auch so beschrieben werden müssen. Die größte Auflösung ist die Femtosekunde.

Ein kleines Beispiel zur Demonstration nutzt bestimmt eine Funktion *veloc()* zur Berechnung einer Geschwindigkeit in m/sec benutzt in einem Prozess mit einem *wait* zur Ausführung.

```
-- phystyp.vhd
-- physikalische Datentypen
entity phystyp is
end phystyp;

architecture algo of phystyp is
  type length is range 0 to 1E9
  units
    um; -- primary unit: micron
    mm = 1000 um; -- metric units
    cm = 10 mm;
    m = 1000 mm;
    inch = 25400 um; -- English units
    foot = 12 inch;
  end units length;

  type speed is range 0 to 1000
  units
    msec;
  end units speed;
```

```

function veloc(s: in length; t: time) return speed is
  variable a, b, c: integer;
  -- variable c: real;
begin
  a := s / 1 m;
  b := t / 1 sec;
  c := a / b;
  return c * 1 msec;
end;

begin -- architecture

  process is
    variable M1, M2, M3: length := 0 m;
    variable X: integer;
    variable S: speed;
    variable T : time := 1 sec;
  begin
    M1 := 23 m; M2 := 200 foot; M3 := 9 inch;
    S := veloc(M2, T);

    wait;
  end process;
end algo;

```

Die Benutzung solcher Datentypen ist allerdings sehr eingeschränkt.

Aufzählungstypen

VHDL benutzt, wie auch die anderen Programmiersprachen, den Aufzählungstypen, um eine Menge von Objekten zu beschreiben und zu benutzen. Die EBNF ist:

Aufzählung_Typdefinition ::= ((Bezeichner | Character_Literal) [{ , ...}])

In der Aufzählung muss mindestens ein Element enthalten sein und jedes Element muss ein Bezeichner oder mindestens ein Buchstabe (character) sein. Beispiele dazu:

```

type alu_function is (disable, pass, add, aabstract, multiply, divide);
type octal_digit is ('0', '1', '2', '3', '4', '5', '6', '7');
.....
variable alu_op: alu_function;
variable last_digit: octal_digit := '0';
.....
alu_op := substract;
last_digit := '7';

```

Die Elemente von Aufzählungen können durchaus gleich sein. Der Compiler erkennt aus dem Typen der Zuweisung automatisch, aus welcher Aufzählung das Element stammt. Dieser Sachverhalt wird auch mit **Overloading** bezeichnet.

```

Type logic_level is (unknown, low, undriven, high);
variable control: logic_level;
type water_level is (dangerously_low, low, ok);
variable water_sensor: water_level;
.....
control := low;
water_sensor := low;

```

Voreingestellte Aufzählungstypen sind (diese sind für Assertions wichtig, die später behandelt werden):

type severity_level **is** (note, warning, error, failure);
type file_open_status **is** (open_ok, status_error, name_error, mode_error);
type file_open_kind **is** (read_mode, write_mode, append_mode);

Zeichentyp (character)

Der Zeichentyp ist ein Aufzählungstyp der alle 256 ASCII-Zeichen des ISO 8859 Latin-1 Alphabets beinhaltet. Dies beinhaltet alle Sonderzeichen und Klein- und Großbuchstaben. Der vollständige, in VHDL implementierte Character-Set ist wie folgt:

```

type character is (
  nul,    soh,    stx,    etx,    eot,    enq,    ack,    bel,
  bs,     hat,    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',
  c128,  c129,  c130,  c131,  c132,  c133,  c134,  c135,
  c136,  c137,  c138,  c139,  c140,  c141,  c142,  c143,
  c144,  c145,  c146,  c147,  c148,  c149,  c150,  c151,
  c152,  c153,  c154,  c155,  c156,  c157,  c158,  c159,
  '¡',    'í',    'ñ',    '£',    '¢',    '¥',    '¦',    '§',
  '¨',    '©',    'ª',    '«',    '¬',    '­',    '®',    '¯',
  '°',    '±',    '²',    '³',    '´',    'µ',    '¶',    '·',
  '¸',    '¹',    'º',    '»',    '¼',    '½',    '¾',    '¿',
  'À',    'Á',    'Â',    'Ã',    'Ä',    'Å',    'Æ',    'Ç',
  'È',    'É',    'Ê',    'Ë',    'Ì',    'Í',    'Î',    'Ï',
  'Ð',    'Ñ',    'Ò',    'Ó',    'Ô',    'Õ',    'Ö',    '×',
  'Ø',    'Ù',    'Ú',    'Û',    'Ü',    'Ý',    'Þ',    'ß',
  'à',    'á',    'â',    'ã',    'ä',    'å',    'æ',    'ç',
  'è',    'é',    'ê',    'ë',    'ì',    'í',    'î',    'ï',
  'ð',    'ñ',    'ò',    'ó',    'ô',    'õ',    'ö',    '÷',
  'ø',    'ù',    'ú',    'û',    'ü',    'ý',    'þ',    'ÿ');

```

Die ersten Zeichen in dem Satz sind nicht druckbare Steuerzeichen des Satzes und die Zeichen mit den Bezeichnern c128 bis c159 haben keine Bedeutung im Standard-Buchstabensatz, daher setzt VHDL hier Bezeichner ein, die die Position im Characterset angeben. Die Zeichen ab der Position 160 sind Sonderzeichen.

Variablen diesen Typs sind wie folgt zu definieren und anzuwenden:

```

variable cmd_char, terminator: character;
.....
cmd_char := 'P'; terminator := cr;

```

Boolean Typ

Der Boolean-Typ ist von sehr großer Bedeutung in VHDL. Er ist ebenfalls ein Aufzählungstyp der folgenden Art:

```

type boolean is (false, true);

```

Dieser Typ ist Grundlage aller bedingten Abfragen in Modellen. Zu dem Boole-Typen gibt es eine Reihe von Operatoren, die auf bestimmte Werte angewandt werden, um Boole'sche Werte zu ergeben, wie z.B. Gleichheit "=" und Ungleichheit "/=", die auf alle Typen angewandt werden können.

Die folgenden Abfragen ergeben **true**:

123 = 123, 'A' = 'A', 7 ns = 7 ns

Die folgenden Abfragen ergeben **false**:

123 = 486, 'A' = 'z', 7 ns = 2 us

Die folgenden relationalen Operatoren prüfen die Rangfolge von Elementen einer Liste. Es sind kleiner "<", kleiner-gleich "<=", größer ">" und größer-gleich ">=".

Die folgenden Abfragen ergeben **true**:

123 < 496, 789 ps <= 789 ps, '1' > '0'

Die folgenden Abfragen ergeben **false**:

96 >= 102, 2 us < 4 ns, 'X' < 'X'

Die logischen Verknüpfungen gelten natürlich nur für boole'sche Variablen und folgen den bekannten Wahrheitstabellen der entsprechenden Funktionen. Implementiert sind: **and**, **or**, **nand**, **nor**, **xor** und **not**. Es sollte noch erwähnt werden, dass bedingte Ausdrücke von links nach rechts evaluiert werden, d.h. wenn z.B.

(b /= 0) **and** (a/b >1)

die Abfrage ist und der erste Teil in **false** resultiert, so wird der zweite Term nicht mehr betrachtet. Gleiches gilt für **nand** und wenn der erste Teil eines **or** oder **nor** in **true** resultiert.

Bit Typ

Da VHDL Hardware modellieren soll, insbesondere digitale Hardware, ist der Aufzählungstyp **bit** von Bedeutung.

Type bit is ('0', '1');

Für diesen Typ gelten die gleichen Operationen wie für den boolean Typ oben. So gilt z.B.

'0' **and** '1' = '0', oder '1' **xor** '1' = '0'

Man beachte jedoch die strikte Typ-Unverträglichkeit zwischen *bit* und *boolean*. Der Bittyp dient halt der Modellierung von Hardware.

Standard Logik Typ

Der Bittyp ist für reale Schaltungen und deren Entwicklung ein noch recht ungenaues Modell, ohne Berücksichtigung technischer Details. Neben den Zeitverzögerungen von Signalen in digitalen Schaltungen zählen auch Details der Schaltungen in der Interpretation der Signalpegel in Bezug auf die Logik. So ist z.B. beim Einschalten der Zustand von Flipflops im allgemeinen unbestimmt. Weiter gibt es Treiberschaltungen und Gatterschaltungen, deren Ausgang so zu steuern ist, dass er signaltechnisch unwirksam ist, d.h. sich in einem sog. hochohmigen Zustand, auch Tristate genannt, befindet. Wir werden nur einige, wenige derartige Details noch näher betrachten. Das ausführlichste und mittlerweile standardisierte Logikmodell ist das sog. **9-wertigen Logikmodell**. Die seitens des Standards festgelegte Bibliothek ist **ieee** und das Package **std_logic_1164**, die jeder Simulator enthalten sollte. Hier sind mehrere Logiktypen definiert, die allesamt Untertypen des 9-wertigen Logiktypen sind.

Der vollständige 9-wertige Logiktyp ist wie folgt:

```

type std_ulogic is ('U',      -- uninitialized; nicht initialisiertes Signal
                    'X',      -- forcing unknown; mehr als ein Treiber an einem Signal
                    '0',      -- forcing zero; wie bittyp '0'
                    '1',      -- forcing one; wie bittyp '1'
```

'Z',	-- high impedance; Tristate
'W',	-- weak unknown; Konflikt zwischen 'L' und 'H'
'L',	-- weak zero; offener Ausgang nach '0'
'H',	-- weak one; offener Ausgang nach '1'
'-',	-- don't care; frei wählbares Signal

Natürlich gelten für diesen Datentyp auch alle logischen Funktionen, die bei der Verknüpfung in gewisser Weise ein logisches Resultat bildet, das im Ergebnis 'U', 'X', '0' oder '1' ergibt, wie z.B. 'U' or '1' = '1' oder 'U' and 'U' = 'U' liefert.

Soll eine solche Bibliothek, wie hier die Standard-Bibliothek **ieee** mit einem Package in ein Programm eingefügt werden, so erfolgt das durch die **use** Klausel. Hier startet das Programm also wie folgt:

```
library ieee;  
use ieee.std_logic_1164.all;
```

Die erste Zeile referiert die Bibliothek **ieee** und die zweite Zeile wählt daraus das Package **std_logic_1164** und daraus wiederum alles Enthaltene **all** (Vereinbarungen und Operationen) aus.

5.6.5 Auflösungsfunktion (Resolution Function)

Die obige Bibliothek mit dem Package **std_logic_1164** enthält einige Signalwerte, deren Interpretation hier notwendig ist.

Bedeutung von 'U'

Diesen Wert nehmen alle Signalwerte an, die noch nicht spezifiziert wurden. Der Unterschied zum *bit* Typ ist eben, dass damit diese Situation nicht erkennbar ist.

Bedeutung von '-'

Dieser Wert besagt, dass der Signalwert egal ist. Dieser Datentyp erlaubt eine Abkürzung der Modellbeschreibungen, denn damit können z.B. Don't Cares aus Wahrheitstabellen übernommen werden. Für die Synthesefähigkeit eines Modells hat das besondere Auswirkungen. In Verhaltensmodellen sollte dieser Wert auch mit Vorsicht benutzt werden, da er keine reale Nachbildung hat.

Die weiteren Aufzählungselemente 'X', 'Z', 'W', 'H' und 'L' betreffen die Zusammenschaltung mehrerer Signale auf die selbe Leitung. Im Modell führt das auf den Fall, dass zwei Einheiten auf den selben Signaltreiber zuweisen. Im allgemeinen ist das in einer Architecture nicht möglich, mehrere Zuweisungen auf ein und dasselbe Signal zu machen, denn eine Architektur legt für jedes Signal einen eigenen Treiber an und damit wird eine mehrfache Zuweisung zunächst unterbunden wegen der Namensgleichheit. In einem Prozess ist dies eben nur möglich, da der Prozess für Signalzuweisungen nur einen Treiber anlegt. Die Situation ist also wie bei der Hardware, wo nur ein Signal von einem Modul getrieben werden sollte.

Für den Fall, dass mehr als ein Signal auf die gleiche Leitung wirken soll, was elektronisch möglich ist, sind besondere Vorkehrungen nötig. VHDL trägt diesen Sonderfällen auf zwei Arten Rechnung, einerseits durch die angedeuteten Elemente der Signalwert-Aufzählung, andererseits durch die sog. *resolution function*. Diese bestimmt zum aktuellen Simulationszeitpunkt aus den aufzulösenden Signalwerten der Treiber einen resultierenden Signalwert, der dann weiter anzuwenden ist.

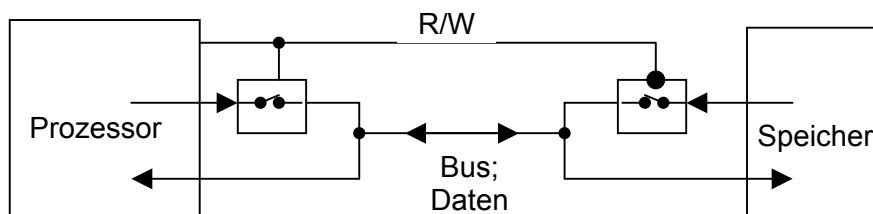
Zunächst zu den Signalwerten:

Der Signalwert 'X' wird angenommen, wenn das Modell Diskrepanzen in den Zuweisungen findet, wenn z.B. der eine Treiber '0' und ein anderer '1' setzt.

'Z' bedeutet hochohmig was gleich bedeutend ist mit Abkoppeln des Signals. Ohne detailliert die elektronische Realisierung zu betrachten, reicht die Vorstellung, dass der Signalzustand 'Z' bedeutet, das

Signal wirkt nicht auf die Leitung, bzw. beeinflusst nicht den Treibersignalwert. Das kann modellhaft durch einen Schalter erfolgen, der einfach öffnet und damit abtrennt.

Die typische Anwendung ist der bidirektionale Datenbus eines Rechnersystems, der mit dem Mode *inout* an Ports modelliert wird. Bei einem Schreib-/Lesespeicher (RAM) werden normalerweise die Daten über ein Leitungsbündel, dem Datenbus, gelesen und auch geschrieben. Die bedienende Einheit ist im allgemeinen der Prozessor des Rechnersystems. Demgemäß dient ein Signal der Steuerung der Übertragungsrichtung (siehe Prinzipschaltbild unten). Im einfachsten Fall ist dies eine Leitung, die man R/W (read not write) benennt. Liest der Prozessor den Speicher, ist der Signalwert '1' für das Lesen des Speichers, d.h. der Speicher schreibt auf den Datenbus und der Prozessor muss sich abkoppeln (linker Schalter auf im Bild). Mit dem Signalwert '0' and R/W schreibt der Prozessor in den Speicher. Hier müssen also die Signale abgetrennt vom Schreiben, also 'Z'). Natürlich muss dabei vermieden werden, dass die Einheit, die den Speicher betreibt, nicht gleichzeitig auf den Bus schreibend zugreift. Das führt zu einem Konflikt auf dem Bus. Daher sind bidirektionale Busse über sog. Bustreiber angekoppelt, die die Ausgänge in den sog. hochohmigen Zustand versetzen, wenn sie über die Leitung lesend zugreifen. Nur im Schreibfall sind die Ausgangstreiber auf den Bus durchzuschalten.



Prinzipschaltbild Datenbus

Beide Treiberstufen sind dann wie folgt zu modellieren:

Auf der Prozessorseite:

```
readdata <= DB when rnw = '1';           -- Lesedaten
DB <= writedata when rnw = '0' else "ZZZZZZZ"; -- Schalter geschlossen
```

Auf der Speicherseite:

```
storedata <= DB when rnw = '0';         -- Einspeicherdaten
DB <= readdata when rnw = '1' else "ZZZZZZZ"; -- Schalter geschlossen
```

Der Lesezugriff kann immer erfolgen.

Man erkennt aber, dass beide Seiten auf den Datenbus zugreifen und dass hier der alternative Zugriff festgelegt wurde. Entsprechend muss die Simulation den Signalwert 'Z' akzeptieren und zulassen, dass sich eine '0' bzw. eine '1' der schreibenden Einheit durchsetzt.

Die Datenbusleitung muss also entsprechend mit einer Auflösungsfunktion betrieben werden. Diese unterscheidet sich zunächst in ihrer Deklaration nicht von der normalen Funktion.

Ihre Formalparameterliste muss eine Felddeklaration beinhalten, die alle Signaltreiber aufnehmen kann, die aufzulösen sind. Bei z.B. zwei Treibern auf das gleiche Signal muss das Feld mindestens zwei Elemente haben. Der Rückgabebetyp muss gleich sein. Die Funktion ermittelt nun aus den Feldwerten den resultierenden Signalwert und gibt ihn zurück.

Die Deklaration von Signalen, die aufzulösen sind, erfolgt mit Einschluss der Funktion, also

```
Signalvereinbarung_RF ::= signal name : resolution_function_name Typ;
```

Nur Signalen mit dieser Vereinbarung dürfen in der Architektur mehrfach Werte zugewiesen werden und das Signal wird mit der entsprechenden Auflösungsfunktion aufgelöst.

Dazu das folgende Beispiel, das einen eigenen Typ `tristate_logic` definiert, der die Werte 'Z' und 'X' zusätzlich enthält. Entsprechend der obigen Erläuterung des Verhaltens dieser Signalwerte ist die Funktion **myres** programmiert. Sie löst 0-1 Konflikte mit 'X' auf und duldet 'Z'.

Die Signalvereinbarung des aufzulösenden Signals ist hervorgehoben. Das Beispiel kann zur Probe genutzt werden, in dem man die nebenläufigen Signalzuweisungen entsprechend variiert.

```

entity resolute is
end resolute;

architecture basic of resolute is
  type tristate_logic is ('0', '1', 'Z', 'X');
  type tristate_logic_array is array (integer range 0 to 2) of tristate_logic;

  function myres(inp: tristate_logic_array) return tristate_logic is
    variable x: tristate_logic := 'Z';
  begin
    for i in inp'range loop
      if x /= inp(i) then -- wenn gleich keine Änderung x
        if x = 'Z' then -- wenn x = 'Z' übernehme jeden Wert
          x := inp(i);
        elsif x /= inp(i) then
          x := 'X';
          exit;
        else x := inp(i);
        end if;
      end if;
    end loop;
    return x;
  end function;

  signal d : myres tristate_logic;

begin -- architecture
  d <= '0', 'Z' after 10 ns;
  d <= '0', '0' after 5 ns;
  d <= '0', '1' after 5 ns;

end basic;

```

Die restlichen Signalwerte 'W', 'H' und 'L' betreffen besondere Ausgangsstufen. Teilen sich mehrere Einheiten einen Bus, auf dem sie untereinander den Zugriff regeln, nutzen sie eine gemeinsame Steuerleitung, die über sog. open-drain oder open-collector Treiber angekoppelt ist. Die Steuerleitung ist an einen Bus-Terminator gekoppelt, der so wirkt, dass er das Signal nur auf log. '1' ziehen kann, wenn alle angekoppelten Treiber im hochohmigen Zustand sind.

%%&%

5.6.6 Attribute

An dieser Stelle sollte ein mächtiges Instrument der Sprachen Ada und damit auch VHDL erwähnt werden, die sog. **Attribute**, das die Programmierung wesentlich komfortabler und verständlicher macht. Wir werden die Nützlichkeit von Attributen noch kennen lernen.

Attribute kennzeichnen eine Vielzahl von Eigenschaften von Variablen, Typen, Feldern u.s.w.. Sie werden gekennzeichnet durch das Hochkomma. Hier wollen wir zunächst die Attribute der oben erwähnten skalaren Datentypen aufzeigen. Dazu die erste Liste von Attributen eines beliebigen skalaren Typen **T**:

T'left	erstes (linkes) Element von T
T'right	letztes (rechtes) Element von T
T'low	kleinster Wert von T
T'high	größter Wert von T
T'ascending	true , wenn T in aufsteigender Folge, false sonst

Betrachtet man nun einzelne Elemente aus der Liste eines skalaren Typs, so ergeben sich weitere Attribute wie:

T'image(x)	ein String, der den Variablenwert x vom Typ T repräsentiert
-------------------	---

T'value(s) ein Wert vom Typ T, der von einem String **s** repräsentiert wird

Einschränkend auf diskrete und physikalische skalare Typen ergeben sich weitere Attribute wie:

T'pred(x) Vorgänger von **x**
T'succ(x) Nachfolger von **x**
T'leftof(x) Vorgänger von **x**
T'rightof(x) Nachfolger von **x**
T'pos(s) die Positionsnummer von **x** in der Aufzählung
T'val(x) das Aufzählungselement auf der Position **x**

Wir wollen uns ein praktisches Beispiel dazu ansehen und nehmen uns dazu den **physikalischen Typ resistance** zur Beschreibung elektronischer Widerstandswerte vor.

```
Type resistance is range 0 to 1E9
units
  ohm;
  kohm = 1_000 ohm;
  Mohm = 1_000 kohm;
end units resistance;
```

```
type set_index_range is range 21 downto 11;
type logic_level is (unknown, low, undriven, high);
```

für diesen Typen gilt:

resistance'left = 0 ohm	resistance'right = 1E9 ohm
resistance'low = 0 ohm	resistance'high = 1e9 ohm
resistance'ascending = true	
resistance'image(2 kohm) = "2000 ohm"	resistance'value("5 Mohm") = 5000000 ohm
set_index_range'left = 21	set_index_range'right = 11
set_index_range'low = 21	set_index_range'high = 11
set_index_range'ascending = false	
set_index_range'image(14) = "14"	set_index_range'value("20") = 20
logic_level'left = unknown	logic_level'right = high
logic_level'low = unknown	logic_level'high = high
logic_level'ascending = true	
logic_level'image(undriven) = "undriven"	logic_level'value("Low") = low

Da *logic_level* ein Aufzählungstyp resultierend aus einer Elementliste ist, gilt ebenso:

logic_level'pos(unknown) = 0	logic_level'val(3) = high
logic_level'succ(unknown) = low	logic_level'pred(undriven) = low

Für die physikalischen Typen gilt hier die Positionsnummer ab der Basiseinheit, also z.B.

time'pos(4 ns) = 4_000_000

also der Abstand von der Basis Femtosekunden **fs**.

Ein schönes Beispiel der Mächtigkeit dieser Konstrukte ist die Flächenberechnung. Wir definieren

```
type length is range integer'low to integer'high
units
  mm;
end units length;

type area is range integer'low to integer'high
units
```

```
qmm;
end units area;
```

```
variable L1, L2: length;
variable A: area;
```

Natürlich ist die Zuweisung $A := L1 * L2$; falsch, da keine Typenverträglichkeit gegeben ist. Aber mit der Hilfe von Attributen und der Tatsache, dass `length` und `area` demselben Wertebereich des Integertyps entstammen, gilt:

```
A := area'val(length'pos(L1) * length'pos(L2));
```

5.6.7 Operationen

Wir wollen jetzt noch einen Überblick über die Operationen geben, bevor wir uns die in VHDL etwas sel-
tteren kompositorischen Datentypen ansehen.

Zunächst die **Grundrechenoperationen** wie `+`, `-`, `**` und `/`, die natürlich and den definierten Daten-
typ als Operation gekoppelt ist. Also ist in VHDL stets eine definitive Typkonvertierung durchzuführen!

Die **Exponentiation** wird mit `**` angegeben und verlangt einen ganzzahligen Exponenten.

Absolutbetrag durch Voranstellung von `abs`, **Modulo** durch Zwischenstellen von `mod` und **Divisions-
restbildung** durch `rem`.

Zeichenketten (String) können mit `&` zusammengefügt werden.

Bei **Vektoren (Worten) von boolean oder bit** gelten die Operationen

- **Stelleninvertierung** `not`, **logischer** Links-/Rechtsverschub `sll` und `srl`, **arithmetischer** Links-
/Rechtsverschub `sla` und `sra` und **Rotation** durch `rol` und `ror`.
- **Logische Verknüpfungen** `and`, `or`, `nand`, `nor`, `xor` und `xnor`.

In Ausdrücken gelten die folgenden boole'schen Evaluierungen:

`=` Gleichheit, `/=` Ungleichheit, `<` kleiner, `<=` kleiner-gleich, `>` größer und `>=` größer-gleich.

5.6.8 Operator-Überlagerung (operator overloading)

5.6.9 Zugriffstypen

Wie oben bereits angedeutet, wollen wir diesen Datentypen in VHDL nicht näher betrachten, da er für
unsere Ziele nur von untergeordneter Bedeutung ist. Daher soll hier ein Verweis auf das Grundlagen-
buch von Ashenden [PA02], Seiten 487 ff genügen, um ggf. dort weitere Informationen zu erhalten.

5.6.10 Filetypen

Ein VHDL File ist eine Klasse von Objekten zur Speicherung von Daten. Dazu ist eine Filetypen-
Definition vorhanden dessen EBNF ist:

```
Filetyp_Definition ::= file of Objekt_Typ
```

Damit ist einfach nur der Datentyp bestimmt, der zu speichern ist. Ein Beispieltyp zur Speicherung reiner Integer-Daten ist

```
type integer_file is file of integer;
```

Damit darf die Datei *integer_file* nur Integer Daten enthalten.

Wir werden auf diesen Datentyp später noch bei der Ein- und Ausgabe zurückkommen müssen.

=====

5.6.11 Ein- und Ausgabe, Filetypen

VHDL hat nur eine rudimentäre Datei Ein- und Ausgabemöglichkeit. Der Grund ist, dass Dateien lediglich der einfacheren und kürzeren Handhabung von großen Testmustersätzen dienen. Entsprechend ist auch nur eine rein sequenzielle Dateihandhabung definiert .

Ein VHDL File ist eine Klasse von Objekten zur Speicherung von Daten. Dazu ist eine Filetypen-Definition vorhanden dessen EBNF ist:

```
Filetyp_Definition ::= file of Datentyp
```

Damit ist einfach nur der Datentyp bestimmt, der zu speichern ist. Ein Beispieltyp zur Speicherung reiner Integer-Daten ist

```
type integer_file is file of integer;
```

Damit darf die Datei *integer_file* nur Integer Daten enthalten. Mit dieser *Filetyp* Definition können wir nun Fileobjekte deklarieren nach folgendem Syntaxschema:

```
File_Vereinbarung ::=  
  file Bezeichner {,...}: Filetyp  
  [ [ open file_open_kind_Ausdruck ] is Filename ] ;
```

Dieser etwas komplizierte Ausdruck setzt einen Bezeichner als Repräsentant einer Datei namens **File-name**, die Objekte des Typs *Filetyp* beinhaltet und die eine bestimmte Nutzungsart aus der Liste *file_open_kind* hat. Die Liste *file_open_kind* unterscheidet drei Elemente:

```
type file_open_kind is (read_mode, write_mode, append_mode);
```

%%&

=====

5.6.12 Kompositorische Datentypen

Wie bereits anfangs erwähnt, wollen wir bei diesen Typen nur die Feldtypen betrachten, da

Felder

Der erste und einfachste zusammengesetzte (kompositorische) Datentyp ist das Feld. Unter einem Feld (array) versteht man eine Anreihung von Objekten **gleichen** Typs. Jedes Objekt der Reihung (Komponente, Element) hat seine festgelegte Position, die mit Index bezeichnet wird. Der Index kann ein beliebiger Aufzählungstyp sein. Die Typvereinbarung

```
Feldtyp_Definition ::= array ( diskreter_Bereich { ,... } ) of Objekt_Typ
```

Diskreter_Bereich kann jeder Aufzählungs-Untertyp sein, der im Bereich eingeschränkt ist. *Elementtyp* kann skalar oder kompositorisch sein.

Wir wollen einige nützliche Beispiele betrachten, die auch Grundlage einiger weiterer Betrachtungen sind. Betrachtet man z.B. ein 16 bit Rechnerwort im sog. **big-endian** Format so definiert man:

```
type word is array (0 to 31) of bit;
```

Benutzt man dagegen das sog. **little-endian** Format, so definiert man:

```
type word is array (31 downto 0) of bit;
```

Aber nicht allein ein Integer-Untertyp kann Index eines Feldes sein, sondern auch normale Aufzählungsmengen, wie z.B.

```
type controller_state is (initial, idle, active, error);  
type state_counts is array (idle to error) of natural;
```

Der Typ *natural* ist ein vordefinierter Untertyp zu Integer. Da hier bei der Indexmenge ggf. nicht ganz klar sein könnte, zu welchem Typ sie gehört, kann man auch deklarieren zu

```
type state_counts is  
array (controller_state range idle to error) of natural;
```

eine weitere Möglichkeit ist die gesamte Indexmenge anzuführen, wie im folgenden

```
subtype coff_ram_address is Integer range 0 to 63;  
type coeff_array is array (coff_ram_address) of real;
```

Objekte solcher Typen sind dann z.B.

```
variable buffer_register, data_register: word;  
variable counters: state_counts;  
variable coeff: coeff_array;
```

Zuweisungen gestalten sich dann z.B. wie folgt:

```
coeff(0) := 0.0;
```

Der erste Index vom Integertyp ist gleich 0!

Zuweisungen zu Feldkomponenten mit Index vom Aufzählungstyp sind wie folgt:

```
counters(active) := counters(active) + 1;
```

Und last but not least kann das komplette Feld zugewiesen werden, wie folgt:

```
data_register := buffer_register;
```

Mehrdimensionale Felder

Natürlich können auch mehrdimensionale Felder definiert werden. Dazu folgendes Beispiel:

```
type symbol is ('a', 't', 'd', 'h', digit, cr, error);  
type stage is range 0 to 6;  
  
type transition_matrix is array (state, symbol) of state;  
.....  
  
variable transition_table: transition_matrix;  
.....
```

```
transition_table(5, 'd') := 3;
```

Aggregate

Ähnlich der Initialisierung von Variablen bei der Vereinbarung ist dies auch mit Feldern möglich. Eine solche Festlegung des Feldes auf Initialwerte wird **Aggregat** genannt. Aggregate können positionsbezogen oder frei definiert werden und es gibt Abkürzungskonstrukte.

Ein erstes Beispiel zu Aggregaten:

```
type point is array (1 to 3) of real;
constant origin: point := (0.0, 0.0, 0.0);
variable view_point: point := (10.0, 20.0, 0.0);
```

Dabei zählt der Index von links nach rechts. Bei zweidimensionalen Feldern zählt die Indizierung wie folgt:

```
Index:          1,1  1,2  1,3  2,1  2,2  2,3  3,1  3,2  3,3
A2: array (1..3, 1..3) of float := ((0.0, 0.0, 0.0), (0.0, 0.0, 0.0), (0.0, 0.0, 0.0));
```

Man kann hier die folgende Abkürzung des Aggregats einführen:

```
A2: array (1..3, 1..3) of float := ((1.0, others => 0.0),
                                   (others => 0.0),
                                   (others => 0.0));
```

An dieser Stelle sei bemerkt, daß wir hier die direkte Feld-Vereinbarung benutzen, d.h. ohne vorher eine type Definition anzugeben. Man bedenke hierbei, daß diese so deklarierten Felder keine Typverwandten haben können!

Man kann eine Unabhängigkeit der Position durch Mitführen des Index-Namens bei Feldern erreichen.

```
Type symbol is ('a', 't', 'd', 'h', digit, cr, error);
type stage is range 0 to 6;
type transition_matrix is array (state, symbol) of state;
constant next_state : transition_matrix :=
  ( 0 => ('a' => 1, others => 6),
    1 => ('t' => 2, others => 6),
    2 => ('d' => 1, 'h' => 5, others => 6),
    3 => (digit => 4, others => 6),
    4 => (digit => 4, cr => 0, others => 6),
    5 => (cr => 0, others => 6),
    6 => (cr => 0, others => 6);
```

Feld Attribute

Für Variable vom Typ *array* sind auch Attribute definiert. Diese können bei einem Felddtyp **A** und einem Index **N** wie folgt aufgelistet werden:

A'left(N)	linke Komponente von A der Dimension N
A'right(N)	rechte Komponente von A der Dimension N
A'low(N)	linke Komponente von A der Dimension N
A'high(N)	rechte Komponente von A der Dimension N
A'range(N)	Index Bereich von A der Dimension N
A'reverse_range(N)	umgekehrter Index Bereich von A der Dimension N
A'length(N)	Länge des Index Bereichs von A der Dimension N
A'ascending(N)	true , wenn Index Bereich von A der Dimension N in aufsteigender Folge, false sonst

Auch hierzu einige Beispiele zum besseren Verständnis. Gegeben sei folgendes Feld:

type A is array (1 to 4, 31 downto 0) of boolean;

Einige Attribute dazu sind:

A'left(1) = 1	A'low(1) = 1
A'right(2) = 0	A'high(2) = 31
A'range(1) ist 1 to 4	A'reverse_range(2) ist 0 to 31
A'length(1) = 4	A'length(2) = 32
A'ascending(1) = true	A'ascending(2) = false

Bei eindimensionalen Feldern kann man die Dimension weglassen.

Eingeschränkte (constrained) und uneingeschränkte (unconstrained) Feldtypen

Die bisher gezeigten Feldtyp-Definitionen zeigten in ihrer Größe festgelegte Felder. In *der type-Vereinbarung* erlaubt VHDL jedoch auch die Definition von Feldtypen mit zunächst unbegrenzter Größe. Die endgültige Größe wird erst bei der Vereinbarung einer Variablen dieses Typs festgelegt. Die dann einzuschränkende Dimensionierung kann daraufhin auf zwei Arten erfolgen:

- implizit durch Anführen eines Aggregats oder
- explizit durch Untertypen der definierten Index-Typen.

Type sample is array (natural range <>) of integer;

variable short_sample: sample(0 to 63);

constant x_sample: sample := (1=>0, 3=>4, 2=>7, 4=>2);

Für *short_sample* wird das Feld dann auf 64 Plätze mit Index von 0 bis 63 beschränkt. Das konstante Feld *x_sample* hat vier Komponenten indiziert mit 1 bis 4.

Strings (Zeichenfolgen)

In VHDL gibt es einen vorgegebenen uneingeschränkten Typen, den String.

type string is array (positive range <>) of character;

Damit kann man zum Beispiel Zeilen von Text zur Anzeige wie folgt deklarieren:

constant LCD_length: positive := 20;

subtype LCD_string is string(1 to LCD_length);

Bitvektoren

Ein weiterer, für Modelle wichtiger uneingeschränkter Feldtyp ist der Bitvektor-Typ. Damit werden wir parallele Datenleitungen von z.B. Arithmetiksaltungen beschreiben. Die Typdefinition:

type bit_vector is array (natural range <>) of bit;

Damit sind dann 8 bit Worte im little-endian Format deklarierbar mit

subtype byte is bit_vector(7 downto 0);

6 Verhaltensorientierte Modellierung

7 Datenfluss-Modelle

8 Strukturorientierte Modellierung

8.1 Modellierung von Bussignalen

8.2 Komponenten und Konfigurationen

8.3 Generics

9 Synthesefähige Modellierung

10 Programmierung von Komplexlogiken mit VHDL

11 VHDL Syntax

Hier sind alle im Skript vorkommenden EBNF-Syntaxbeschreibungen der Sprache VHDL zusammengefasst. Wir ersparen uns den grundsätzlichen Aufbau von Bezeichnern und Ableitungen zu Konstanten-, Variablen- und Signal-Bezeichnern.

Faktor ::= Konstante | Variable | Signal

Wir kürzen Faktor teilweise wie folgt ab und überlassen die Diversifizierung dem jeweiligen Kontext, bzw. spezifizieren wir durch Voranstellung in Kursivschrift zu:

*num*_Faktor ::= Konstante | Variable | Signal -- numerischer Typ
*boole*_Faktor ::= Konstante | Variable | Signal -- boolean Typ

Gleiches machen wir auch mit einem Ausdruck:

Ausdruck ::= Faktor { (+ | - | * | / | mod | rem) Faktor }

Ist die kürzere Form für:

*num*_Ausdruck ::= [abs] *num*_Faktor { (+ | - | * | / | mod | rem) [abs] *num*_Faktor }
*boole*_Ausdruck ::= [not] *boole*_Faktor { (and | or | ...) [not] Faktor }

wir benutzen zudem:

Abfrage synonym zu *boole*_Ausdruck und **Ausdruck** statt *num*_Ausdruck .

Weiter benutzen wir:

Parameterliste ::= Bezeichner { , ... }
Modus ::= in | out | inout
Term ::= Faktor { (* | / | mod | rem) Faktor }
Auswahl ::= (Ausdruck | diskreter_Bereich | **others**) { | ... }
diskreter_Bereich ::= Ausdruck (to | downto) Ausdruck

Typdefinitionen:

Typ_Definition ::= Alternative zwischen den folgenden
Integer_Typdefinition ::= **range** Ausdruck (to | downto) Ausdruck
Gleitkomma_Typdefinition ::= **range** Ausdruck (to | downto) Ausdruck
Physikalische_Typdefinition ::=
range Ausdruck (to | downto) Ausdruck
units
Bezeichner;
{ Bezeichner = Physikalisches_Literal }
end units [Bezeichner]
Physikalisches_Literal ::= [Dezimales_Literal | Basistyp_Literal] *unit*_Name
Aufzählung_Typdefinition ::= ((Bezeichner | Character_Literal) [{ , ... }])
Filetyp_Definition ::= **file of** Objekt_Typ
Feldtyp_Definition ::= **array** (diskreter_Bereich { , ... }) **of** Objekt_Typ

Typvereinbarungen:

Typ_Vereinbarung ::= **type** Bezeichner **is** Typ_Definition;

Subtyp_Vereinbarung ::= **subtype** Bezeichner **is** Subtyp_Indikator;
Subtyp_Indikator ::=
Basistyp [**range** Ausdruck (**to** | **downto**) Ausdruck]

Objektvereinbarungen:

Objekt_Vereinbarung ::= **variable** Bezeichner : Typ_Bezeichner;
| **signal** Bezeichner : Typ_Bezeichner;

Anweisungen:

Eine Anweisung ist als Alternative der folgenden Anweisungstypen gegeben, also

Anweisung ::= Zuweisung | *if*_Anweisung | *case*_Anweisung
| *loop*_Anweisungen | *sonder*_Anweisungen

Eine Sequenz von Anweisungen kann dann gegeben werden zu:

sequenzielle_Anweisung ::= Anweisung { ... }

Ein Block von Anweisungen kann dann gegeben werden zu:

konkurrente_Anweisung ::= Signalzuweisung { ... }

Anweisungstypen:

Zuweisung ::= Variablenzuweisung | Signalzuweisung;

Variablenzuweisung ::= Variable := Ausdruck;

Signalzuweisung ::= Signal <= Signalausdruck;

***If*_Anweisung** ::=
[*if*_Bezeichner:]
if Abfrage **then**
 { sequenzielle_Anweisung }
{ elsif Abfrage **then**
 { sequenzielle_Anweisung } }
[else
 { sequenzielle_Anweisung }]
end if [*if*_Bezeichner],

***case*_Anweisung** ::=
[*case*_Bezeichner:]
case Ausdruck **is**
 { **when** Auswahl => { sequenzielle_Anweisung } }
 { ... }
end case [*case*_Bezeichner];

*loop*_Anweisungen ::= *for-loop*_Anweisung | *while-loop*_Anweisung | *loop*_Anweisung

***for-loop*_Anweisung** ::=
[*loop*_Bezeichner:]
for Zähler **in** diskreter_Bereich **loop**
 { sequenzieller Befehl }
end loop [*loop*_Bezeichner];

```
while-loop_Anweisung ::=
  [ loop_Bezeichnung: ]
  while Abfrage loop
  { sequenzielle_Anweisung }
  end loop [ loop_Bezeichnung ];
```

```
loop_Anweisung ::=
  [ loop_Bezeichnung: ]
  loop
  { sequenzielle_Anweisung }
  end loop [ loop_Bezeichnung ];
```

```
sonder_Anweisungen ::= exit_Anweisung | next_Anweisung
exit_Anweisung ::= [ Bezeichner: ] exit [ loop_Bezeichnung ] [ when Abfrage ];
next_Anweisung ::= [ Bezeichner: ] next [ loop_Bezeichnung ] [ when Abfrage ];
```

Konkurrenente Anweisungen:

Wir sahen oben die Signalzuweisung wie folgt:

```
Signalzuweisung ::= Signal <= Signalausdruck;
```

Signalausdruck beinhaltet weitere Möglichkeiten der Steuerung (Bedingung):

Die erste Möglichkeit besteht in der Modifikation des Signalausdrucks:

```
Signalausdruck ::= Waveform | cond_Waveform
Waveform ::= Wave_Element { ,... };
Wave-Element ::= Ausdruck [ after Zeitausdruck ]
```

```
cond_Waveform ::= [ Wave_Element when Bedingung else ] [ { ... } ]
  Wave_Element [ when Bedingung ];
```

Die zweite Möglichkeit besteht in der Auswahl der Signalzuweisung:

```
Signalzuweisung_Auswahl ::=
  with Ausdruck select
  Signal <= [ Wave_Element when Auswahl, ] [ { ... } ]
  Wave_Element when Auswahl;
```

Zur Abrundung jetzt der Prozess als Domäne sequenzieller Anweisungen und Variablen:

```
Prozess_Definition ::=
  [ Prozess_Bezeichnung: ]
  process [ (Sensitivity_Liste) ] is
  { Prozess_Vereinbarung }
  begin
  { sequenzielle_Anweisung }
  end process [ Prozess_Bezeichnung ];
```

und dann der Block als Domäne konkurrierender Anweisungen und Signalen:

```
block_Anweisung ::=
  [ block_Bezeichnung: ]
  block [ (guard_Abfrage) ] [ is ]
  [ Blockdeklarationen ]
```

```
begin  
  { konkurrenente_Anweisung }  
end block [ block_Bezeichnung ];
```

%&%

12 Literatur

- [PA02] P.J. Ashenden: The Designer's Guide to VHDL, Second Edition; Morgan Kaufmann Publishers, 2002, ISBN: 1-55860-674-2
- [RS00] J. Reichardt, B. Schwarz: VHDL-Synthese, Entwurf digitaler Schaltungen und Systeme; Oldenbourg Verlag, 2000, ISBN: 3-486-25128-7
- [CS01] C. Siemers: Hardwaremodellierung, Einführung in Simulation und Synthese von Hardware; HANSER Verlag, 2001; ISBN: 3-446-21361-9
- [JB98] J. Bhasker: A VHDL Primer, Third Edition; Prentice Hall PTR, 1998, ISBN: 0130965758
- [AG93] J.R. Armstrong, F.G. Gray: Structured Logic Design with VHDL; Prentice-Hall, 1993; ISBN: 0-13-885206-1
- [AG00] J.R. Armstrong, F.G. Gray: VHDL Design Representation and Synthesis, 2nd ed.; Prentice-Hall, 2000; ISBN: 0-13-021670-4; vergriffen!!!
- [CS99] C. Siemers: Prozessorbau, Eine konstruktive Einführung in das Hardware / Software-Interface; HANSER Verlag, 1999; ISBN: 3-446-19330-8
- [SiDr02] A. Sikora, R. Drechsler: Software-Engineering und Hardware-Design, Eine systematische Einführung; HANSER Verlag, 2002; ISBN: 3-446-21861-0
- [Wo01] W. Wolf: Computers as Components, Principles of Embedded Computing System Design; Morgan Kaufmann Publishers, 2001; ISBN: 1-55860-693-9
- [BRJ99] G. Booch, J. Rumbaugh, I. Jacobson: Das UML-Benutzerhandbuch, 2. Auflage; Eddison-Wesley, 1999; ISBN: 3-8237-1486-0
- [OF03] W. Oberschelp, G. Vossen: Rechneraufbau und Rechnerstrukturen; Oldenbourg Verlag, 2003; ISBN: 3-4862-7206-3