Teil B Die Sprache VHDL

1 Allgemeines

1.1 VHDL´87 oder VHDL´93 ?

Das Erscheinen dieses Buches fällt mit einem wichtigen Zeitpunkt zusammen: Nach der ersten Überarbeitung der VHDL-Norm (IEEE-Standard 1076) in den Jahren 1992 und 1993, fünf Jahre nach der Verabschiedung, sind die Softwarehersteller dabei, ihre Programme der neuen Syntax anzupassen. Nach und nach werden die VHDL´93-kompatiblen Programme ältere Versionen ersetzen.

Welcher Standard soll nun in einem "aktuellen" Buch beschrieben werden? VHDL´87, mit dem wahrscheinlich zum Zeitpunkt des Erscheinens die meisten Entwickler noch arbeiten, oder VHDL´93, das in den nächsten Jahren ältere Programmversionen ablösen wird. Erschwert wird die Problematik durch die Tatsache, daß die beiden Versionen nicht vollkommen aufwärtskompatibel sind. Es wurden nämlich auch einige Konstrukte aus der alten Syntax eliminiert.

Letztendlich haben sich die Autoren entschieden, der momentan heterogenen Situation Rechnung zu tragen und beide Versionen zu beschreiben. Dort, wo nichts besonderes vermerkt ist, gilt die Syntax für VHDL´87 und VHDL´93. Teile, die nur für VHDL´87 gelten, sind mit dem Zeichen ✓**87**, Teile der neuen Norm mit ✓**93** gekennzeichnet.

Zu den wesentlichen Neuerungen im 93-er Standard gehören:

- ein erweiterter Zeichensatz,
- Gruppen,
- globale Variablen,
- Schiebe- und Rotierfunktionen für Vektoren,
- dem Simulationszyklus nachgestellte Prozesse,
- Ergänzung und Elimination einiger vordefinierter Attribute.

1.2 Vorgehensweise und Nomenklatur

Die Vorgehensweise dieses Buches ist eine etwas andere als die herkömmlicher VHDL-Bücher. So soll die Hardwarebeschreibungssprache ausgehend von der Basis, dem benutzten Zeichenvorrat, erläutert werden. Mit der Beschreibung von Sprachelementen, Daten und Objekten wird die Basis für die im weiteren folgende Darstellung der Befehle zur strukturalen Modellierung und zur Verhaltensmodellierung gelegt. Dem Simulationsablauf und der Konfiguration in VHDL ist jeweils ein eigenes Kapitel gewidmet. Den Abschluß des Syntaxteils bildet ein Kapitel über spezielle Modellierungstechniken.

Für eine einsichtige Darstellung von Syntaxregeln und VHDL-Beispielen (lauffähiger Quellcode) ist eine klare und durchgehende Nomenklatur erforderlich.

Die üblicherweise zur Syntaxbeschreibung verwendete BNF (Backus Naur Form) erweist sich sehr wohl als sinnvoll zur vollständigen und korrekten Definition einer Syntax. Zum Erlernen einer Sprache erscheint uns diese BNF jedoch ungeeignet. Deshalb entschieden wir uns, zur Syntaxbeschreibung eine vereinfachte Variante der BNF zu wählen, in der folgende Nomenklatur gilt:

- anstelle von formalen, hierarchisch deduzierten Definitionen stehen mehrere konkrete Einzeldefinitionen (nur in wenigen Fällen wird, gekennzeichnet durch kursive Formatierung, auf vorher eingeführte Definitionen verwiesen),
- VHDL-Schlüsselwörter sind immer in Großbuchstaben verfaßt,
- frei wählbare Bezeichner (Typnamen, Objektnamen, ...) oder Ausdrücke sind klein geschrieben und tragen selbstbeschreibende Namen,
- optionale Angaben stehen in eckigen Klammern `[]`,
- beliebig oft wiederholbare Angaben stehen in geschweiften Klammern `{}`.

2 Sprachelemente

2.1 Sprachaufbau

Aus dem Zeichensatzvorrat werden durch gezielte Verknüpfungen und Kombinationen die lexikalischen Elemente und daraus wiederum die VHDL-Sprachkonstrukte aufgebaut. Diese ergeben in ihrem Zusammenwirken die Design-Einheiten ("design units"), welche die Komponenten der VHDL-Modelle bilden.

Dieser Aufbau der Modelle aus elementaren Elementen kann mit dem Aufbau der Materie aus Atomen und Molekülen verglichen werden. Abb. B-1 verdeutlicht den Sprachaufbau graphisch.

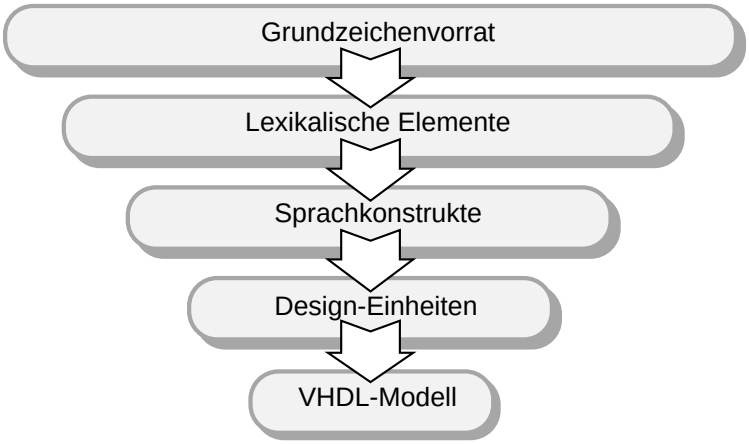

Abb. B-1: VHDL-Sprachaufbau

2.2 Zeichensatz

Der Zeichensatz von VHDL umfaßt in der ursprünglichen Version (✓87) nur 128 Zeichen, entsprechend der 7-Bit ISO 83-Norm. Neben den herkömmlichen Groß- und Kleinbuchstaben sind die Ziffern 0 bis 9, ein gewisser Satz an Sonderzeichen sowie unsichtbare Formatierungszeichen enthalten.

Der Umfang des Zeichensatzes von ✓87 wird am Beispiel der Deklaration für den Aufzähltyp `character` gezeigt:

```
TYPE character IS (
   NUL,   SOH,   STX,   ETX,   EOT,   ENQ,   ACK,   BEL,
   BS,    HT,    LF,    VT,    FF,    CR,    SO,    SI,
   DLE,   DC1,   DC2,   DC3,   DC4,   NAK,   SYN,   ETB,
   CAN,   EM,    SUB,   ESC,   FSP,   GSP,   RSP,   USP,
   ' ',   '!',   '"',   '#',   '$',   '%',   '&',   ''',
   '(',   ')',   '*',   '+',   ',',   '-',   '.',   '/',
   '0',   '1',   '2',   '3',   '4',   '5',   '6',   '7',
   '8',   '9',   ':',   ';',   '<',   '=',   '>',   '?',
   '@',   'A',   'B',   'C',   'D',   'E',   'F',   'G',
   'H',   'I',   'J',   'K',   'L',   'M',   'N',   'O',
   'P',   'Q',   'R',   'S',   'T',   'U',   'V',   'W',
   'X',   'Y',   'Z',   '[',   '\',   ']',   '^',   '_',
   '`',   'a',   'b',   'c',   'd',   'e',   'f',   'g',
   'h',   'i',   'j',   'k',   'l',   'm',   'n',   'o',
   'p',   'q',   'r',   's',   't',   'u',   'v',   'w',
   'x',   'y',   'z',   '{',   '|',   '}',   '~',   DEL);
```

Mit der neuen VHDL-Norm wurde die Zeichendarstellung von 7 auf 8 Bit, der Zeichenvorrat damit auf insgesamt 256 Zeichen entsprechend der Norm ISO 8859-1 erweitert. Er umfaßt nunmehr auch landesspezifische Umlaute und weitere Sonderzeichen. Der Umfang des neuen Zeichensatzes (✓93) wird am Beispiel der `character`-Typdeklaration gezeigt:

```
TYPE character IS (
   ...
   ...    -- alle Zeichen aus VHDL'87
   ...
   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,
   ' ',   '¡',   '¢',   '£',   '¤',   '¥',   '|',|', '§',
   '¨',   '©',   'ª',   '«',   '¬',   '-',   '®',    '¯',
   '°',   '±',   '2',   '3',   '´',   'µ',   '¶',    '.',
   '¸',   '1',   'º',   '»',   '1/4', '1/2', '3/4',  '¿',
   'À',   'Á',   'Â',   'Ã',   'Ä',   'Å',   'Æ',    'Ç',
   'È',   'É',   'Ê',   'Ë',   'Ì',   'Í',   'Î',    'Ï',
   '☆',   'Ñ',   'Ò',   'Ó',   'Ô',   'Õ',   'Ö',    '☆',
   'Ø',   'Ù',   'Ú',   'Û',   'Ü',   '☆',   '☆',    'ß',
   'à',   'á',   'â',   'ã',   'ä',   'å',   'æ',    'ç',
   'è',   'é',   'ê',   'ë',   'ì',   'í',   'î',    'ï',
   '☆',   'ñ',   'ò',   'ó',   'ô',   'õ',   'ö',    'Π',
   'Ø',   'ù',   'ú',   'û',   'ü',   '☆',   '☆',    'ÿ');
```

Die durch ein '☆'-gekennzeichneten Zeichen konnten mit dem Zeichensatz des verwendeten Textverarbeitungssystems leider nicht dargestellt werden.

VHDL ist im allgemeinen (syntaktische Elemente und Bezeichner) nicht case-sensitiv, d.h. Groß- und Kleinschreibung wird von den Anwendungsprogrammen nicht unterschieden. Ein Bezeichner namens `input12` hat die gleiche Bedeutung wie `INPUT12` oder `Input12`.

Eine Ausnahme von dieser Regel bilden lediglich die "extended identifier" (✓**93**), sowie Einzelzeichen ("character") und Zeichenketten ("strings").

Die Eigenschaft der "case-Insensitivität" bietet sich an, um eine bessere Lesbarkeit des VHDL-Codes zu erreichen. Eine von Anfang an konsequent beibehaltene Groß- und Kleinschreibung von syntaktischen Elementen zahlt sich mit Sicherheit aus. In diesem Buch werden zum Beispiel Schlüsselwörter und Attribute der VHDL-Syntax stets groß geschrieben.

Damit VHDL-Modelle auch auf Rechnern angelegt werden können, bei denen die Eingabe der drei Sonderzeichen " # | nicht möglich ist, erlaubt VHDL die Ersetzung durch die Zeichen % : ! in den Anweisungen. Ein Beispiel für eine Ersetzung:

```
CASE value_string IS                  -- Ausschnitt aus einem
   WHEN "high" | "undefined" =>       -- VHDL-Modell

CASE value_string IS                  -- aequivalente
   WHEN %high% ! %undefined% =>       -- Beschreibung
```

2.3 Lexikalische Elemente

Aus dem Zeichenvorrat, sozusagen den Atomen von VHDL, werden zunächst die sog. "lexikalischen Elemente" gebildet. Lexikalische Elemente sind also Kombinationen von Elementen des Zeichenvorrates, die eine bestimmte Bedeutung haben. Um beim Vergleich mit der Chemie zu bleiben, könnte man die lexikalischen Elemente etwa als Moleküle betrachten. Die Bedeutung der lexikalischen Elemente läßt sich in verschiedene Sprachelemente aufteilen. Aus der richtigen Kombination dieser Sprachelemente setzen sich wiederum die Design-Einheiten zusammen.

Lexikalische Elemente können in folgende Gruppen eingeteilt werden:

2.3.1 Kommentare

Kommentare dienen lediglich zur besseren Lesbarkeit von VHDL-Quellcode; sie haben keinerlei Bedeutung für die Funktion eines Modells. Eine Ausnahme hiervon bilden Steueranweisungen für Synthesewerkzeuge, die oft innerhalb eines Kommentares stehen.

Das Kommentarzeichen ist der doppelte Bindestrich ("--"); er kennzeichnet den Anfang eines Kommentares, der dann bis zum Ende der Zeile reicht. Das Kommentarzeichen kann zu Beginn einer Zeile oder nach VHDL-Anweisungen stehen.

```
-------------------- Ein Kommentar beginnt mit -- und reicht
ENTITY inv IS     -- bis zum Zeilenende. Er kann alle
                  -- moeglichen Zeichen (*'_>-<) beinhalten.
```

2.3.2 Bezeichner

Bezeichner, im Englischen "identifier", sind Namen von Design-Einheiten, Objekten, Objekttypen, Komponenten, Funktionen, etc. Bei der Wahl von Bezeichnern sind folgende Regeln zu beachten:

- ❒ Bezeichner bestehen aus Buchstaben, Ziffern und einzelnen Unterstrichen; sie dürfen keine Leer- und Sonderzeichen enthalten,
- ❒ Bezeichner sind case-insensitiv,
- ❒ das erste Zeichen eines Bezeichners muß ein Buchstabe sein,
- ❒ der Unterstrich ("_") darf nicht am Anfang oder Ende des Bezeichners und nicht zweimal unmittelbar aufeinanderfolgend verwendet werden,
- ❒ Bezeichner dürfen keine reservierten Worte sein.

Diese Regeln, v.a. hinsichtlich des ersten Zeichens und des Sonderzeichenverbots, stellen doch eine erhebliche Einschränkung dar. Modelle, z.B. von digitalen Grundbausteinen, können nicht ihre technische Bezeichnung als Bezeichner tragen. Demnach sind `MODULE#09`, `8085` und `74LS11` illegale Namen in ✓**87**. Diese Einschränkung bei der Wahl von Bezeichnern wurde mit der Einführung der sog. "extended identifier" in ✓**93** aufgehoben. Diese erweiterten Bezeichner stehen innerhalb nach links geneigter Schrägstriche ("`\...\`") und weisen folgende Eigenschaften auf:

- ❒ sie unterscheiden sich von herkömmlichen Bezeichnern gleichen Wortlauts,
- ❒ sie sind case-sensitiv,
- ❒ Graphikzeichen (jedoch keine Formatierungszeichen) dürfen enthalten sein,
- ❒ benachbarte Schrägstriche repräsentieren einen Schrägstrich im Namen,
- ❒ sie dürfen mit einer Ziffer beginnen,

- ❒ sie dürfen Leerzeichen enthalten,
- ❒ sie dürfen mehr als einen Unterstrich in Folge beinhalten,
- ❒ sie dürfen mit reservierten Worten identisch sein.

Durch die Verwendung der Schrägstriche lassen sich also mehr und aussagekräftigere Bezeichner verwenden als vorher. Einige Beispiele für normale und extended identifier:

```
node_a, xyz, comp12   -- Normale Bezeichner (Identifier)
abc, Abc, ABC         -- Identische Bezeichner
12_in_bus             -- !! illegal: 1.Zeichen k. Buchstabe
port, buffer          -- !! illegal: Reservierte Worte
bus__parity           -- !! illegal: Mehr als 1 Unterstrich
bus_#12               -- !! illegal: Sonderzeichen
sign a to f           -- !! illegal: Leerzeichen
```

```
\name_of_sign\        -- Extended Identifier ! nur VHDL'93 !
\abc\, \Abc\, \ABC\   -- Verschiedene Bezeichner
\12_in_bus\, \74LS11\ -- legal: 1.Zeichen ist Ziffer
\port\, \buffer\      -- legal: Reservierte Worte
\bus__parity\         -- legal: Mehr als ein Unterstrich
\bus_#12\             -- legal: Sonderzeichen
\sign a to f\         -- legal: Leerzeichen
```

2.3.3 Reservierte Wörter

Reservierte Wörter haben eine bestimmte Bedeutung für die Syntax und dürfen deshalb nicht als (normale) Bezeichner verwendet werden. Es handelt sich dabei um Operatoren, Befehle und sonstige VHDL-Schlüsselwörter. Schlüsselwörter der neuen Norm (✓**93**) sind in der folgenden Aufzählung gekennzeichnet.

ABS	ARRAY	BUFFER
ACCESS	ASSERT	BUS
AFTER	ATTRIBUTE	
ALIAS		CASE
ALL	BEGIN	COMPONENT
AND	BLOCK	CONFIGURATION
ARCHITECTURE	BODY	CONSTANT

		MAP		ROR	(✓93)
DISCONNECT		MOD			
DOWNTO				SELECT	
		NAND		SEVERITY	
ELSE		NEW		SHARED	(✓93)
ELSIF		NEXT		SIGNAL	
END		NOR		SLA	(✓93)
ENTITY		NOT		SLL	(✓93)
EXIT		NULL		SRA	(✓93)
				SRL	(✓93)
FILE		OF		SUBTYPE	
FOR		ON			
FUNCTION		OPEN		THEN	
		OR		TO	
GENERATE		OTHERS		TRANSPORT	
GENERIC		OUT		TYPE	
GROUP	(✓93)				
GUARDED		PACKAGE		UNAFFECTED	(✓93)
		PORT		UNITS	
IF		POSTPONED	(✓93)	UNTIL	
IMPURE	(✓93)	PROCEDURE		USE	
IN		PROCESS			
INERTIAL	(✓93)	PURE	(✓93)	VARIABLE	
INOUT					
IS		RANGE		WAIT	
		RECORD		WHEN	
LABEL		REGISTER		WHILE	
LIBRARY		REJECT	(✓93)	WITH	
LINKAGE		REM			
LITERAL	(✓93)	REPORT		XNOR	(✓93)
LOOP		RETURN		XOR	
		ROL	(✓93)		

2.3.4 Größen

Größen, im Englischen "literals", dienen zur Darstellung von Inhalten bestimmter Objekte oder fester Werte. Man unterscheidet hierbei zwischen numerischen Größen, einzelnen Zeichen ("character") und Zeichenketten ("strings"), Aufzählgrößen ("enumeration types") und sog. "Bit-Strings".

2.3.4.1 Numerische Größen

Numerische Größen können unterteilt werden in abstrakte, einheitenlose numerische Größen zu Basen im Bereich von 2 bis 16 (Basis 10 ist die herkömmliche Darstellung) und mit Einheiten behaftete, sog. "physikalische" Größen.

Abstrakte numerische Größen zur Basis 10

Der einfachste Größentyp ist von ganzzahliger Art ("integer") mit negativen oder positiven Werten. Erlaubt ist die Exponentialschreibweise mit nicht negativen, ganzzahligen Exponenten unter Einbeziehung von bedeutungslosen Unterstrichen (_). Diese Unterstriche dienen lediglich der besseren Lesbarkeit von besonders langen Zahlen. Nicht enthalten sein dürfen insbesondere Leerzeichen und Dezimalpunkte. Integerwerte können mit einer oder mehreren Nullen beginnen. Das Pluszeichen bei positiven Exponenten kann weggelassen werden.

```
2                   -- dies alles sind Beispiele von
00124789673         -- Integergroessen, die in VHDL
250_000_000         -- verwendet werden koennen; man
3E6                 -- beachte Exponentialschreibweise
2_346e+3            -- und Unterstriche
20.0                -- !!! kein Integerwert: Dezimalpunkt
3E-6                -- !!! kein Integerwert: neg. Exponent
```

Im Gegensatz zu den Integergrößen, die keinen Dezimalpunkt aufweisen dürfen, müssen Fließkommagrößen (reelle Größen) einen Dezimalpunkt enthalten. Außerdem können sie negative Exponenten besitzen.

```
2.0012              -- dies sind Beispiele von reellen
16.896E3            -- Groessen und ihre Darstellungs-
65.89E-6            -- moeglichkeiten in VHDL;
3_123.5e+6          -- auch hier gelten Exponential-
23.4e-3             -- schreibweise und Unterstriche.
234e-4              -- !!! kein reeller Wert: . fehlt
```

Abstrakte numerische Größen zu von 10 verschiedenen Basen

Bei diesen Typen muß die Basis (im Bereich von 2 bis 16) explizit angegeben werden:

```
basis#integerwert[.integerwert]#[exponent]
```

Die einzelnen Ziffernwerte müssen jeweils kleiner als die Basis sein. Für Zahlen größer als 9 gilt die hexadezimale Schreibweise:

"a" oder "A" entspricht der Basis 10,
"b" oder "B" entspricht der Basis 11,
"c" oder "C" entspricht der Basis 12,
"d" oder "D" entspricht der Basis 13,
"e" oder "E" entspricht der Basis 14,
"f" oder "F" entspricht der Basis 15.

Die Basis des Exponenten ist immer 10, unabhängig von der Basis.

```
2#110_010#          -- Beispiele von ganzahligen
3#221_002_012#      -- und reellen Groessen
8#476#E9            -- mit von 10 verschiedener
8#177_001#          -- Basis.
16#fff_abc#         -- Die Basis wird durch das
16#f9ac.0#          -- Nummernsymbol # abgehoben;
16#ACE.2#e-2        -- der Exponent steht dahinter.
17#ACG.2#e-2        -- !!! illegal: Basis > 16
8#787_119#          -- !!! illegal: Ziffernwerte zu gross
```

Physikalische Größen

Physikalische Größen bestehen aus einem Wert in beliebiger Darstellung (siehe oben) und einer Einheit. Die erlaubten Einheiten werden in einer Typdeklaration festgelegt (siehe Abschnitt 3.2). Die meistbenutzte physikalische Größe ist die Zeit. Als Basiseinheit ist hier `fs` (= Femtosekunden, 1.0E-15 sec) üblich. Daneben sind sog. abgeleitete Einheiten (`ps`, `ns`, `us`, etc.) erlaubt.

```
20.5 ms             -- Beispiele von physikalischen
5.3e3 ps            -- Groessen (Wert und Einheit)
1.8E-9 sec          -- hier: Zeitgroessen
```


2.3.4.2 Zeichengrößen

Zeichengrößen, im Englischen "characters", sind einzelne Zeichen, die in Hochkommata stehen müssen. Sie sind allerdings case-sensitiv, d.h. ein `'a'` und ein `'A'` werden in VHDL unterschieden.

```
'x' '2' '*' ''' '"'     -- dies alles sind gueltige und
'X' '0' '%' ' ' '?'     -- verschiedene Zeichengroessen.
```

2.3.4.3 Zeichenketten

Zeichenketten, im Englischen "strings", sind beliebig lange Ketten aus Einzelzeichen, die in Anführungszeichen stehen. Es ist auch eine leere Zeichenkette erlaubt. Wie bei Einzelzeichen wird auch bei den Zeichenketten zwischen Groß- und Kleinschreibung unterschieden; d.h. die Zeichenketten `"VHDL"`, `"Vhdl"` und `"vhdl"` sind voneinander verschieden. Zeichenketten innerhalb von Zeichenketten stehen in doppelten Anführungszeichen. Um Zeichenketten zu verknüpfen (zusammenzubinden), ist der Verknüpfungsoperator (`&`) zu verwenden.

Achtung: Eine Zeichenkette mit einem Element entspricht nicht dem dazu passenden Einzelzeichen, d.h. beispielsweise, daß die Zeichenkette `"A"` ungleich dem Zeichen `'A'` ist.

```
"Dies ist eine Zeichenkette"   -- in VHDL gueltige
"erster, " & "zweiter String"  -- Zeichenketten
"Alpha", "alpha"               -- verschiedene Zeichenketten
""                             -- leere Zeichenkette
"Ein Anfuehrungszeichen: "" "  -- Ein Anfuehrungszeichen: "
"Ein ""string"" in einem String"
```

2.3.4.4 Bit-String-Größen

Bit-Strings sind Zeichenketten, die aus den Ziffern `0` bis `9` und den Buchstaben `a (A)` bis `f (F)`, entsprechend dem hexadezimalen Zif-

fernsatz, bestehen und einen binären, oktalen oder hexadezimalen Wert darstellen. Ähnlich wie bei den numerischen Größen zu von 10 verschiedenen Basen wird vor der in Anführungsstrichen stehenden Zeichenkette die Basis vermerkt:

- der Buchstabe "b" oder "B" kennzeichnet binären Code; er kann auch weggelassen werden (Defaultwert),
- der Buchstabe "o" oder "O" kennzeichnet oktalen Code,
- der Buchstabe "x" oder "X" kennzeichnet hexadezimalen Code.

Der Wert der Ziffern in der Zeichenkette muß kleiner als die Basis sein. Bedeutungslose Unterstriche dürfen enthalten sein, falls die Basis explizit angegeben ist.

```
"1100100010000"       -- gueltige Bit-Strings
b"110_010_001_000"    -- fuer die Dezimalzahl 3208
o"6210"               -- in binaerem, oktalem und
X"C88"                -- hexadezimalem Code
```

Bit-Strings dienen zur Kurzdarstellung von Werten des Typs `bit_vector`. Bei der Darstellung in oktalem und hexadezimalem Code werden sie bei der Zuweisung entsprechend konvertiert. Der Bit-String `x"0fa"` entspricht dem Bit-String `b"0000_1111_1010"`. Führende Nullen werden also umgesetzt.

Bit-Strings sind nach ✓**87** nicht auf andere Typen anwendbar. Nach ✓**93** ist z.B. eine Zuweisung auch auf Objekte vom Typ einer mehrwertigen Logik möglich. Es stehen dabei selbstverständlich nur die "starke 0" und die "starke 1" als Wert zur Verfügung, da nur für diese eine allgemeingültige Umwandlung zwischen den drei Basen bekannt ist.

2.3.5 Trenn- und Begrenzungszeichen

Um verschiedene lexikalische Elemente, die nacheinander aufgeführt sind, richtig als eigenständige Elemente interpretieren zu können, müssen zwischen ihnen Zeichen stehen, die die einzelnen lexikalischen Elemente abgrenzen oder trennen.

Als Trenn- und Begrenzungszeichen dienen sowohl Leerzeichen, die außerhalb von Kommentaren, Zeichen und Zeichenketten stehen, als auch Formatierungszeichen und folgende Operatoren, Klammern und Kommentarzeichen:

Einzelzeichen: `( ) | ' . , : ; / * - < = > & +`

Zusammengesetzte Zeichen: `=> >= <= := /= <> ** --`

Die Verwendung von Leerzeichen und Zeilenumbrüchen ist, wenn ein Trennzeichen eingesetzt wird, nicht notwendig. Sie dienen dann nur der besseren Lesbarkeit des VHDL-Textes. Die Architektur `structural` des Halbaddierers aus Teil A könnte deshalb - syntaktisch vollkommen korrekt - auch so aussehen:

```
ARCHITECTURE structural OF halfadder IS COMPONENT xor2
PORT(c1,c2:IN bit;c3:OUT bit);END COMPONENT;COMPONENT--Hallo
and2 PORT (c4,c5:IN bit;c6:OUT bit);END COMPONENT;BEGIN
xor_instance:xor2 PORT MAP(sum_a,sum_b,sum);and_instance
:and2 PORT MAP (sum_a,sum_b,carry);END structural;--Leute!
```

2.4 Sprachkonstrukte

Unter Sprachkonstrukten sind sämtliche Kombinationen von lexikalischen Elementen zu verstehen, die eine syntaktische Bedeutung besitzen.

Es kann dabei grob zwischen Primitiven, Befehlen und syntaktischen Rahmen für Funktionen, Prozeduren, Design-Einheiten, etc. unterschieden werden.

2.4.1 Primitive

Primitive stellen in irgendeinerweise einen Wert dar. Primitive sind entweder einzelne Operanden oder Ausdrücke, bestehend aus Operanden und Operatoren. Das Ergebnis von zwei mit einem Operator verknüpften Operanden kann selbst wieder als Operand verwendet werden. Zu achten ist dabei jedoch auf Typkonformität. Einige Operato-

ren verlangen nach bestimmten Operandentypen, andere wiederum können verschiedene Operandentypen in unterschiedlicher Weise verarbeiten (sog. "überladene" Operatoren).

2.4.1.1 Operanden

Es gibt folgende Alternativen für Operanden:

- **Explizite Größenangaben:** numerische Größen, Zeichen und Zeichenketten sowie Bit-Strings; sie können direkt als Operanden eingesetzt werden.
- **Bezeichner:** Referenzname eines Objektes ("identifier").
- **Attribute:** sie dienen zur Abfrage bestimmter Eigenschaften von Objekten.
- **Aggregate:** im Englischen "aggregates"; sie kombinieren einen oder mehrere Werte in einem Feldtyp ("array") oder zusammengesetzten Typ ("record").
- **Qualifizierte Ausdrücke** ("qualified expression"): sie dienen zur expliziten Festlegung des Datentyps bei Operanden, die mehreren Typen entsprechen können. Die Syntax hierfür lautet:

  `type_name'(`*`ambiguous_operand_expr`*`)`
- **Funktionsaufrufe**
- **Typumwandlungen**

Einige Beispiele für verschiedene Operanden bzw. Ausdrücke:

```
a := 3.5 * 2.3;               -- reelle Operanden
b := 300 + 500;               -- ganzzahlige Operanden
c := b - 100;                 -- Bezeichner (b) als Operand
d := b'HIGH + q'LOW;          -- Attribute (HIGH, LOW)
e := NOT bit_vector'("001");  -- qualified expression
```

2.4.1.2 Operatoren und Prioritäten

Operatoren verknüpfen einen oder mehrere Operanden zu einem neuen Wert bzw. Operanden. Die verschiedenen Operatoren sind in

Gruppen mit gleicher Priorität eingeteilt. Die Priorität der Gruppen (entsprechend der nachfolgenden Aufzählung) regelt die Reihenfolge bei der Abarbeitung verketteter Operatoren.

Die Priorität der Operatoren nimmt in folgender Aufzählung nach unten hin ab:

- ❒ Diverse Operatoren (** ABS NOT)
- ❒ Multiplizierende Operatoren (* / MOD REM)
- ❒ Signum-Operatoren (+ -)
- ❒ Addierende Operatoren (+ - &)
- ❒ Vergleichsoperatoren (= /= < <= > >=)
- ❒ Logische Operatoren (AND NAND OR NOR XOR) (XNOR (nur ✓93))

Operatoren mit gleicher Priorität werden in der Reihenfolge des Auftretens im Quellcode abgearbeitet. Ist eine andere Ausführungsreihenfolge erwünscht, so muß dies durch Klammerung explizit gekennzeichnet werden. Dafür stehen nur runde Klammern zur Verfügung.

```
a := '0'; b := '1'; c := '1';
d := a AND (b OR c);                -- d = '0'
e := (a AND b) OR c;                -- e = '1'
u := x and y /= z;
v := x and (y /= z);                -- entspricht u
w := (x and y) /= z;                -- entspricht nicht u
```

Es sei hier bereits auf die Identität des Vergleichsoperators "<=" ("kleiner gleich") mit dem Symbol für eine Signalzuweisung, z.B. "a <= 32;" hingewiesen. Die korrekte Interpretation ergibt sich aus dem jeweiligen Umfeld, in dem dieses Symbol steht.

2.4.2 Befehle

Befehle sind Sequenzen von VHDL-Schlüsselwörtern, die eine bestimmte Funktion besitzen. Befehle können, angefangen von einfachen Signalzuweisungen bis hin zu komplexeren Gebilden wie Schleifen oder Verzweigungen, sehr vielfältiger Art sein. Die Beschreibung

der einzelnen Befehle erfolgt detailliert in den entsprechenden Kapiteln. Hier soll deshalb nur eine erste Unterteilung in die wichtigsten Befehlsklassen erfolgen:

Deklarationen

Hierbei handelt es sich im einzelnen um:

- Typdeklarationen,
- Objektdeklarationen,
- Schnittstellendeklarationen,
- Komponentendeklarationen,
- Funktions- und Prozedurdeklarationen.

Sequentielle Befehle

Sequentielle, d.h. nacheinander ablaufende Befehle, finden sich nur innerhalb von sog. Prozessen, Funktionen oder Prozeduren. Es handelt sich dabei um programmiersprachenähnliche Befehle (Schleifen, Verzweigungen, Variablenzuweisungen, etc.).

Nebenläufige Befehle

Im Gegensatz zu vielen, rein sequentiellen Programmiersprachen kennt man in VHDL auch parallele oder nebenläufige Befehle, die das spezielle (parallelartige) Verhalten von Hardware (z.B. von parallel geschalteten Flip-Flops in Registern) widerspiegeln.

Konfigurationsbefehle

Eine besondere Klasse von Befehlen dient zum Konfigurieren von VHDL-Modellen in der entsprechenden Design-Einheit oder in strukturalen Architekturen.

2.4.3 Syntaktische Rahmen

Hierbei handelt es sich um bestimmte Schlüsselwortkombinationen, die den syntaktischen Rahmen für Funktionen, Prozeduren, Design-Einheiten etc. bilden. Sie enthalten i.d.R. am Anfang das jeweilige Schlüsselwort und den Referenznamen und am Schluß eine Anweisung mit dem Schlüsselwort `END`.

3 Objekte

Sämtliche Daten in VHDL werden über sog. Objekte verwaltet. Jedes Objekt gehört einer bestimmten Objektklasse an und besitzt einen definierten Datentyp. Desweiteren benötigt jedes Objekt einen Referenznamen, den sog. Bezeichner (im Englischen "identifier").

Vor der eigentlichen Verwendung in der Modellbeschreibung muß das Objekt daher zunächst unter Angabe der Objektklasse, des Identifiers, des Datentyps und eventuell eines Defaultwertes deklariert werden.

Zur Festlegung eines Datentyps werden vor der Objektdeklaration getrennte Typdeklarationen verwendet.

3.1 Objektklassen

Konstanten

Konstanten sind Objekte, deren Wert nur einmal zugewiesen werden kann. Der Wert bleibt somit über der Zeit, d.h. während der gesamten Simulationsdauer, konstant.

Variablen

Variablen sind Objekte, deren aktueller Wert gelesen und neu zugewiesen werden kann. Der Variablenwert kann sich also im Laufe der Simulation ändern. Beim Lesen der Variable ist allerdings immer nur der aktuelle Wert verfügbar, auf vergangene Werte kann nicht zurückgegriffen werden.

Signale

Signale können wie Variablen jederzeit gelesen und neu zugewiesen werden; ihr Wert besitzt also ebenfalls einen zeitlich veränderlichen Verlauf. Im Gegensatz zu Variablen wird bei Signalen allerdings dieser zeitliche Verlauf gespeichert, so daß auch auf Werte in der Ver-

gangenheit zugegriffen werden kann. Außerdem ist es möglich, für Signale einen Wert in der Zukunft vorzusehen. Beispielsweise weist der Befehl "`a <= '0' AFTER 10 ns;`" dem Signal `a` den Wert `'0'` in 10 ns, vom momentanen Zeitpunkt ab gerechnet, zu.

Dateien

Dateien enthalten Folgen von Werten, die über bestimmte File-I/O-Funktionen zu einem bestimmten Zeitpunkt oder zeitverteilt gelesen oder geschrieben werden können.

3.2 Datentypen und Typdeklarationen

Bevor in einem VHDL-Modell mit Objekten gearbeitet werden kann, muß festgelegt werden, welche Werte das Objekt annehmen kann (z.B. "Signal a kann einen ganzzahligen Wert zwischen 0 und 10 besitzen"). Für diesen Zweck werden Datentypen eingesetzt, mit deren Hilfe die Wertebereiche definiert werden. Die Sprache VHDL besitzt einerseits nur sehr wenige, vorab definierte Datentypen, wie `real` oder `integer`. Andererseits bietet sie aber umfangreiche Möglichkeiten, benutzerdefinierte Datentypen zu definieren. Im nachstehenden Beispiel wird über den Datentyp `augenzahl` festgelegt, daß der Variablen `wuerfel` nur ganzzahlige Werte zwischen 1 und 6 zugewiesen werden können.

```
TYPE augenzahl IS RANGE 1 TO 6 ;
VARIABLE wuerfel : augenzahl ;
```

Bei der Normierung von VHDL wurde eine strenge Typisierung der Daten angestrebt, weil dadurch "Programmierfehler" oft schneller detektiert werden. Bei jeder Operation, die auf ein Objekt angewandt wird, wird überprüft, ob der Datentyp des Objekts von der jeweiligen Operation auch bearbeitet werden kann.


```
   ...
a := 0.4;         -- "real-Variable"
b := 1;           -- "integer-Variable"
IF b > a THEN     -- !!! Fehler; der vordefinierte Operator >
   ...            -- gilt nur fuer gleiche Datentypen!
```

Der VHDL-Anwender hat die Möglichkeit, den Anwendungsbereich der vordefinierten Operatoren so zu erweitern, daß auch die benutzereigenen Datentypen verarbeitet werden können (sog. "Overloading").

Üblicherweise werden Datentypen in skalare, Feld- und zusammengesetzte sowie sonstige Typen unterteilt. Letztere sind File- und Access-Typen. Diese Typen sind zur Erläuterung der grundlegenden VHDL-Konstrukte nicht notwendig. Sie werden deshalb erst am Ende von Teil B behandelt.

Typdeklarationen können an folgenden Stellen auftreten:

- im `ENTITY`-Deklarationsteil,
- im `ARCHITECTURE`-Deklarationsteil,
- im `PACKAGE`,
- im `PACKAGE BODY`,
- im `BLOCK`-Deklarationsteil,
- im `PROCESS`-Deklarationsteil,
- im `FUNCTION`- und `PROCEDURE`-Deklarationsteil.

3.2.1 Einfache Typen

3.2.1.1 Aufzähltypen

Objekte dieses Typs, im Englischen "enumeration type" genannt, können nur bestimmte Werte annehmen. Die endliche Anzahl von möglichen Werten wird in der Typdeklaration festgelegt:

```
TYPE enum_type_name IS ( value_1 { , value_n });
```

Die möglichen Werte (`value_1 {, value_n}`) müssen Bezeichner sein. Für die Bezeichner gelten die oben definierten Anforderun-

gen. Alternativ dazu können Einzelzeichen (`character`) in einfachen Hochkommata als Werte eines Aufzähltyps eingesetzt werden.

```
TYPE zustand IS (init, run, stop);  -- Bezeichner
TYPE log3    IS ('0', '1', 'Z');    -- Einzelzeichen (Char.)
TYPE fehler  IS (0, 1, Z);          -- !! illegal: Bezeichner
                                    -- 0 und 1 ungueltig
```

Folgende Aufzähltypen sind im Package `standard` vordefiniert und können in jedem VHDL-Modell eingesetzt werden:

```
TYPE boolean         IS (false, true);
TYPE bit             IS ('0', '1');
TYPE character       IS ( ... );     -- VHDL'87: 128 Zeichen
                                     -- VHDL'93: 256 Zeichen
TYPE severity_level  IS (note, warning, error, failure);
```

Die Bezeichner eines Aufzähltyps werden implizit mit ganzzahligen Werten von links nach rechts durchnumeriert. Das am weitesten links stehende Element nimmt die Position `0` ein. Beispielsweise hat das Element `run` des Typs `zustand` die Positionsnummer `1`. Auf die Positionsnummern kann mit Hilfe von Attributen zugegriffen werden.

3.2.1.2 Ganzzahlige Typen

Ganzzahlige Typen, im Englischen "integer types", werden durch direkte Angabe einer ganzzahligen Ober- und Untergrenze des möglichen Wertebereiches deklariert. Alternativ dazu kann der Wertebereich auch von einem anderen Typ abgeleitet werden:

```
TYPE  int_type_name IS RANGE   range_low
                    TO         range_high;

TYPE  int_type_name IS RANGE   range_high
                    DOWNTO     range_low;

TYPE  int_type_name IS RANGE
                    other_int_type_name'RANGE;
```

Der maximal mögliche Wertebereich für einen ganzzahligen Typ ist abhängig von der jeweiligen Rechnerumgebung. Die VHDL-Norm definiert jedoch, daß der mögliche Wertebereich mindestens von `-2147483647` bis `+2147483647` reicht. VHDL weicht damit von gängigen Programmiersprachen ab, die für ganzzahlige Werte einen Bereich von `-2147483648` bis `+2147483647` definieren.

3.2.1.3 Fließkommatypen

Entsprechend den ganzzahligen Typen werden auch Fließkommatypen, im Englischen "floating point types" oder "real types", deklariert. Der einzige Unterschied sind die Ober- und Untergrenze des Bereichs. Diese Grenzen müssen hier Fließkommawerte sein:

```
TYPE real_type_name IS RANGE range_low
                    TO       range_high;

TYPE real_type_name IS RANGE range_high
                    DOWNTO   range_low;

TYPE real_type_name IS RANGE
                 other_real_type_name'RANGE;
```

Auch bei den Fließkommatypen ist der Zahlenbereich von der Rechnerumgebung abhängig. Die VHDL-Norm definiert einen Mindestbereich von `-1.0E38` bis `+1.0E38`.

Auf der beigefügten Diskette befindet sich im File mit Namen `"TYP_ATTR.VHD"` ein VHDL-Modell, mit dem Sie den tatsächlichen Zahlenbereich Ihrer Rechnerumgebung bestimmen können.

```
TYPE neg_zweistellige IS RANGE -99 TO -10;  -- int. -99 - -10
TYPE stack_position IS RANGE 9 DOWNTO 0;    -- int. 9 - 0
TYPE stp IS RANGE stack_position'RANGE;     -- int. 9 - 0
TYPE scale IS RANGE -1.0 TO 1.0;            -- Fliesskomma
TYPE not_valid IS RANGE -1.0 TO 1;          -- !!! illegal
```

Die beiden Typen `integer` und `real` können ohne Deklaration verwendet werden, sie sind folgendermaßen vordeklariert:

```
TYPE integer         IS RANGE ... ;      -- systemabhängig
TYPE real            IS RANGE ... ;      -- systemabhängig
```

3.2.1.4 Physikalische Typen

Physikalische Werte bestehen aus einem ganzzahligen oder reellen Zahlenwert und einer Einheit. Neben einer sog. Basis-Einheit können in der Deklaration eines physikalischen Typs weitere, von vorhergehenden Einheiten abgeleitete Einheiten angegeben werden:

```
TYPE phys_type_name IS RANGE  range_low
                       TO     range_high
   UNITS
       base_unit;
     { derived_unit = multiplicator unit; }
   END UNITS;
```

Die Werte von `range_low`, `range_high` und `multiplicator` müssen ganzzahlig sein. Alle physikalischen Objekte können mit reellen Werten versehen werden, werden jedoch auf ein ganzzahliges Vielfaches der Basiseinheit gerundet.

Im folgenden ist ein Beispiel eines benutzerdefinierten, physikalischen Typs gezeigt:

```
TYPE length IS RANGE -1E9 TO 1E9      --  -1000 bis +1000 km
   UNITS mm;                          --  Basiseinheit mm;
      cm   = 10 mm;                   --  abgeleitete
      dm   = 10 cm;                   --  Einheiten
      m    = 10 dm;
      km   = 1E3 m;
      inch = 25 mm;                   --  nur ganzzahlige
      foot = 305 mm;                  --  Multiplikatoren!
      mile = 16093 dm;                --  Landmeile
   END UNITS;
```


Der einzige vordefinierte, physikalische Typ ist `time`:

```
TYPE time IS RANGE ...  -- systemabhaengiger Bereich
   UNITS fs;                          --  Basiseinheit: fs
      ps  = 1000 fs;                  --  sukzessiv
      ns  = 1000 ps;                  --  abgeleitete
      us  = 1000 ns;                  --  Einheiten
      ms  = 1000 us;                  --  bis hin
      sec = 1000 ms;                  --  zu:
      min =   60 sec;                 --  Minute und
      hr  =   60 min;                 --  Stunde
   END UNITS;
```

3.2.1.5 Abgeleitete einfache Typen

Man kann von bereits deklarierten Typen weitere Typen, sog. Untertypen (im Englischen "subtypes"), ableiten. Untertypen sind im Falle einfacher Typen im Wertebereich eingeschränkte Basistypen. Die Ableitung eines Untertyps von einem Untertyp ist nicht möglich.

Die Syntax einer einfachen Untertyp-Deklaration mit Einschränkung im Wertebereich lautet wie folgt:

```
SUBTYPE subtype_name IS base_type_name
        [RANGE range_low TO range_high];

SUBTYPE subtype_name IS base_type_name
        [RANGE range_high DOWNTO range_low];
```

Die Verwendung von abgeleiteten Typen oder Untertypen hat folgende Vorteile:

- Durch die meist kürzere Untertypdefinition kann VHDL-Code und Zeit eingespart werden.
- Durch die Einschränkung des zulässigen Wertebereiches eines VHDL-Objektes können Modellierungsfehler leichter entdeckt werden.
- Objekte mit verschiedenen Untertypen des gleichen Basistyps können mit den Operatoren des Basistyps verknüpft werden. Bei verschiedenen Typen ist dies i.d.R. nicht möglich.

Das nachstehende Beispiel illustriert den letztgenannten Vorteil:

```
PROCESS
   TYPE address1 IS RANGE 0 TO 63;          -- neue
   TYPE address2 IS RANGE 0 TO 127;         -- int.-Typen
   SUBTYPE add1 IS integer RANGE 0 TO 63;   -- abgeleitete
   SUBTYPE add2 IS integer RANGE 0 TO 127;  -- int.-Typen
   VARIABLE ta : address1; VARIABLE tb, tc : address2;
   VARIABLE sa : add1; VARIABLE sb, sc : add2;
BEGIN
   sc := sa + sb;       -- legal: gleicher Basistyp
   tc := ta + tb;       -- !!! illegal: verschiedene Typen
   ...
END PROCESS;
```

Unter Verwendung des Attributes `HIGH` vordefinierte Untertypen sind:

```
SUBTYPE natural      IS integer  RANGE 0 TO integer'HIGH;
SUBTYPE positive     IS integer  RANGE 1 TO integer'HIGH;
SUBTYPE delay_length IS time     RANGE 0 fs TO time'HIGH;
    --  delay_length nur in VHDL'93 vordefiniert!
```

Hinweis: Eine syntaktisch entsprechende Einschränkung des Wertebereiches kann auch erst in der Objektdeklaration erfolgen.

3.2.1.6 Typumwandlungen

VHDL bietet die Möglichkeit der Umwandlung zwischen verschiedenen Typen. Ein Anwendungsfall für solche Funktionen ist die Zusammenschaltung von verschiedenen VHDL-Modellen mit unterschiedlichen logischen Signaltypen. Hier müssen bei der Verdrahtung Funktionen zur Signalkonvertierung angegeben werden.

Implizit sind in VHDL Funktionen zur Umwandlung von Fließkommatypen in ganzzahlige Typen und umgekehrt, sowie zwischen verschiedenen ganzzahligen Typen und zwischen verschiedenen Fließkommatypen bekannt:

```
integer (float_object_name)
real (integer_object_name)

int_type_name (other_int_type_obj_name)
float_type_name (other_float_type_obj_name)
```

Bei der Wandlung zu ganzzahligen Werten wird der Fließkommawert gerundet (bis ausschließlich .5 abgerundet, ab .5 aufgerundet). Der Wertebereich des Zieltyps darf bei der Typumwandlung nicht überschritten werden.

```
va := 2;                        -- Variable ganzzahligen Typs
vb := 3.5;                      -- Fließkommavariable
va := va + integer(vb);         -- legal, va = 6
va := va + vb;                  -- !!! illegal: versch. Typen
```

Funktionen zur Umwandlung zwischen verschiedenen Logiktypen müssen selbst definiert werden.

3.2.2 Feldtypen

Sollen mehrere Werte in einem Objekt zusammengefaßt werden (z.B. die Werte einer Matrix), dann wird für dieses Objekt ein sog. Feldtyp ("array type") verwendet. Im eindimensionalen Fall nennt man die Felder "Vektoren", im zweidimensionalen Fall "Matrizen". Die Einzelelemente von Feldern können neben skalaren Typen auch andere Feldtypen oder zusammengesetzte Typen sein, müssen aber innerhalb des Feldes von ein- und demselben Typ sein.

Man unterscheidet bei Feldern zwischen Feldern mit unbeschränkter Größe ("unconstrained arrays") und eingeschränkter Größe ("constrained arrays").

3.2.2.1 Vektoren

Die Typdeklaration eines Feldes hat im **eindimensionalen Fall** folgendes Aussehen:

```
TYPE array_type_name IS ARRAY
  (index_type RANGE <>) OF base_type_name;

TYPE array_type_name IS ARRAY
  ([index_type RANGE] range_low TO range_high)
                        OF base_type_name;

TYPE array_type_name IS ARRAY
  ([index_type RANGE] range_high
                        DOWNTO range_low)
                        OF base_type_name;
```

Das Konstrukt `RANGE <>` bedeutet dabei unbeschränkte Länge des Vektors (im Rahmen des möglichen Bereiches des Index-Typs).

Als Index (`index_type`) können beliebige diskrete Typen, also neben ganzzahligen Typen auch Aufzähltypen, verwendet werden. Der Typ des Index bestimmt auch die Default-Indizierung. Bei eingeschränkter Indizierung und eindeutigem Typ ist die Angabe des Schlüsselwortes `RANGE` und des Index-Typs nicht unbedingt erforderlich.

```
TYPE color    IS (yellow, red, green, blue);
TYPE int_vec1 IS ARRAY (color RANGE <>)
              OF integer;
TYPE int_vec2 IS ARRAY (red TO blue)
              OF integer;              -- Vektorlaenge: 3
TYPE int_vec3 IS ARRAY (255 DOWNTO 0)
              OF integer;              -- Vektorlaenge: 256
```

Die Vektortypen `string` und `bit_vector` müssen nicht deklariert werden; sie sind bereits im Package `standard` enthalten. Bit-Vektoren eignen sich beispielsweise zur Beschreibung von Bussen, Registern oder Zeilen einer PLA-Matrix.

```
TYPE string     IS ARRAY (positive RANGE <>)
                OF character;      -- vordefinierter Typ
TYPE bit_vector IS ARRAY (natural RANGE <>)
                OF bit;            -- vordefinierter Typ
```

3.2.2.2 Mehrdimensionale Felder

Im **mehrdimensionalen Fall** (allgemeine Felder) muß entsprechend der Vektordeklaration für jede Dimension der Indextyp und der Indexbereich angegeben werden. Ein Vermischen der drei verschiedenen Indizierungsarten (unbeschränkt, beschränkt mit aufsteigender Indizierung, beschränkt mit abfallender Indizierung) ist in einem mehrdimensionalen Feld erlaubt. Die Typdeklaration lautet wie folgt:

```
TYPE array_type_name IS ARRAY
  ( index_type RANGE <>
  { , further_index } ) OF base_type_name;

TYPE array_type_name IS ARRAY
  ([index_type RANGE] range_low TO range_high
  { , further_index } ) OF base_type_name;

TYPE array_type_name IS ARRAY
  ([index_type RANGE] range_high
                      DOWNTO range_low
  { , further_index } ) OF base_type_name;
```

Der Basistyp des Feldes kann dabei auch wieder ein Feld sein.

```
TYPE int_matrix IS ARRAY       -- 3x6 Matrix
     (integer RANGE 1 TO 3,
      integer RANGE 1 TO 6) OF integer;
TYPE real_array IS ARRAY       -- dreidimensionales Feld
     (integer RANGE 8 DOWNTO 1,
      color   RANGE <>,
      color   RANGE red TO blue) OF real;
TYPE array_of_array IS ARRAY   -- Vektor mit Vektorelementen
     (color   RANGE red TO blue) OF int_vec3;
```

3.2.2.3 Abgeleitete Feldtypen

Wie von den einfachen Typen können auch von den Feldtypen Untertypen abgeleitet werden. Im Unterschied zu einfachen Typen wird bei Feldtypen nicht der Wertebereich, sondern der Indexbereich eingeschränkt. Mehrfache Untertypableitungen sind auch hier nicht möglich.

Die Syntax einer Untertyp-Deklaration von im allgemeinen mehrdimensionalen Feldtypen lautet folgendermaßen:

```
SUBTYPE subtype_name IS base_type_name
        ( range_low TO range_high
       { , further_index_constraints } );

SUBTYPE subtype_name IS base_type_name
        ( range_high DOWNTO range_low
       { , further_index_constraints } );
```

```
TYPE bit_matrix    IS ARRAY (1 TO 256, 1 TO 256) OF bit;
SUBTYPE nachname   IS string (1 TO 20);     -- 20 Zeichen
SUBTYPE word       IS bit_vector (1 TO 16); -- 16 Bit
SUBTYPE eight_word IS bit_matrix (1 TO 16, 1 TO 8);
SUBTYPE byte       IS word (1 TO 8);        -- !! illegal !!
```

Hinweis: Eine syntaktisch entsprechende Einschränkung des Indexbereiches kann auch erst bei der Objektdeklaration erfolgen.

3.2.3 Zusammengesetzte Typen

Will man mehrere Elemente unterschiedlichen Typs in einem Objekt kombinieren, so verwendet man zusammengesetzte Typen, im Englischen "records":

```
TYPE record_type_name IS RECORD
   record_element_1_name  : element_1_type  ;
 { record_element_n_name  : element_n_type  ;}
END RECORD ;
```


Mit ✓93 kann der Name des zusammengesetzten Typs in der END-Anweisung wiederholt werden. Damit wird eine Vereinheitlichung der Rahmensyntax für Design-Einheiten und Befehle realisiert.

```
TYPE record_type_name IS RECORD
   ...
END RECORD [record_type_name] ;
```

Zwei Beispiele für Records sind die folgenden Typen für Datum und komplexe Zahlen.

```
TYPE months  IS (january, february, march, ... , december);
SUBTYPE days IS integer RANGE 1 TO 31;
TYPE date    IS RECORD
                year      : natural;
                month     : months;
                day       : days;
END RECORD;

TYPE complex IS RECORD
                real_part : real;
                imag_part : real;
END RECORD;
```

Records selbst können wiederum Elemente eines Records sein. Man kann die Elemente eines Objekts mit zusammengesetztem Typ sowohl einzeln lesen als auch einzeln schreiben.

3.3 Objektdeklarationen

Bevor ein Objekt, beispielsweise eine Variable `delay_lh`, in einem VHDL-Modell verwendet werden kann, muß es deklariert werden, z.B.:

```
VARIABLE delay_lh : time := 3 ns;
```

Die Deklaration von Objekten enthält also Elemente, die

- das zu deklarierende Objekt einer Objektklasse zuordnen (hier: `VARIABLE`),
- den Referenznamen des Objektes festlegen (hier: `delay_lh`),
- den Datentyp durch Angabe seines Referenznamens festlegen (hier: `time`),
- gegebenenfalls einen Defaultwert für das Objekt angeben (hier: `3 ns`).

Mehrere, gleichartige Objekte können in einer Anweisung gemeinsam deklariert werden. Als Datentyp können deklarierte Typen und Untertypen übernommen oder spezielle Einschränkungen (entsprechend der Untertyp-Deklaration) vorgenommen werden.

3.3.1 Konstanten

Konstanten sind Objekte mit einem festem Wert, der sich im Laufe der Ausführung eines Modells nicht ändern kann. Dieser Wert muß in der Konstantendeklaration festgelegt werden, d.h. die Defaultwertvergabe ist hier obligatorisch. Der Typ von `value` muß dem Typ der Konstante entsprechen. Die Syntax der Konstantendeklaration ist wie folgt:

```
CONSTANT   const_name_1 { , const_name_n }
           : type_name := value ;
```

```
CONSTANT delay_lh  : time       := 12.5 ps;
CONSTANT x1, x2, x3: integer    := 5;
CONSTANT r_address : bit_vector := b"1001_1110";   -- 0 TO 7
CONSTANT offset    : bit_vector (1 TO 3) := "101";
CONSTANT message   : string     := "Segmentation fault";
```

Konstantendeklarationen dürfen an folgenden Stellen stehen:

- im `ENTITY`-Deklarationsteil,
- im `ARCHITECTURE`-Deklarationsteil,
- im `PACKAGE`,
- im `PACKAGE BODY`,

- im `BLOCK`-Deklarationsteil,
- im `PROCESS`-Deklarationsteil,
- im `FUNCTION`- und `PROCEDURE`-Deklarationsteil,
- in der Parameterliste von `FUNCTION` und `PROCEDURE` (nur Mode `IN`).

Im Package darf die Konstantendeklaration als einzige Ausnahme ohne Wertangabe stehen (sog. "deferred constant"). Der Wert wird in einer entsprechenden Anweisung im Package Body festgelegt.

3.3.2 Variablen

Variablen sind Objekte mit zeitlich veränderbaren Werten. In der ursprünglichen Norm (✓87) waren sie nur innerhalb eines einzigen Prozesses oder Unterprogramms (Funktion, Prozedur) gültig, um deterministisches Verhalten der VHDL-Modelle sicherzustellen.

Bei einigen Anwendungsfällen, z.B. bei der Systemmodellierung oder der Verwaltung von Zuständen in objektorientierten Modellen, wird dies jedoch als Nachteil empfunden. Das strikte Verbot von globalen Variablen wurde deshalb nach langanhaltender Diskussion in der überarbeiteten Norm (✓93) aufgegeben. Nunmehr sind Variablen (nach besonderer Deklaration) auch von mehreren Prozessen innerhalb des Gültigkeitsbereiches quasi gleichzeitig les- und schreibbar. Der weniger gefährlich klingende Name "shared variable" ändert nichts an der Tatsache, daß damit Modelle erzeugt werden können, deren Verhalten nicht vorhersagbar ist (nichtdeterministische Modelle). Deshalb sollten VHDL-Anwender diese neue Möglichkeit nur mit Vorsicht verwenden. Momentan beschäftigt sich eine eigene VHDL-Arbeitsgruppe ausschließlich mit dem Konzept der globalen Variablen.

Die Syntax der Variablendeklaration sieht in der Norm ✓87 wie folgt aus:

```
VARIABLE   var_name_1 { , var_name_n }
           : type_name [ := def_value ];
```

Die Angabe eines Defaultwertes (`def_value`) ist optional. Er darf ein beliebiger typkonformer Ausdruck sein und legt den initialen Wert

der Variablen fest. Ohne explizite Angabe eines Defaultwertes wird der Variablen zu Beginn der Simulation der Wert zugewiesen, der in der entsprechenden Typdeklaration am weitesten links steht. Der Defaultwert für Variablen vom Typ `bit` ist also `'0'`.

```
VARIABLE n1, n2   : natural;                -- default: 0
VARIABLE v_addr   : integer RANGE -10 TO 10; -- default: -10
VARIABLE o_thresh : real := 1.4;            -- default: 1.4
```

Die Deklarationen von Variablen können in ✓**87** nur an folgenden Stellen stehen:

- im `PROCESS`-Deklarationsteil,
- im `FUNCTION`- und `PROCEDURE`-Deklarationsteil.

Die in ✓**93** um das optionale Wort `SHARED` erweiterte Syntax lautet:

```
[SHARED] VARIABLE var_name_1 { , var_name_n }
                 : type_name [ := def_value ];
```

Ohne das Wort `SHARED` entspricht die Verwendung dem Standard von ✓**87**. Wird das Wort `SHARED` verwendet, darf die Deklaration nun in allen Deklarationsteilen mit Ausnahme von Prozessen und Unterprogrammen eingesetzt werden. Diese Art von Variablen darf in Zonen des Modells gelesen und geschrieben werden, in der auch normale Variablen gelesen und geschrieben werden können, also in Prozessen und Unterprogrammen.

3.3.3 Signale

Neben den Variablen und Konstanten, die auch von prozeduralen Programmiersprachen bekannt sind, verfügt VHDL noch über Objekte der Klasse "Signal". Diese wurde eingeführt, um die Eigenschaften elektronischer Systeme modellieren zu können. Änderungen von Signalwerten können beispielsweise zeitverzögert zugewiesen werden. Damit lassen sich die Laufzeiten von Hardware-Komponenten nachbilden. Signale dienen im wesentlichen dazu, Daten zwischen parallel arbeitenden Modulen auszutauschen. Verschiedene Module können

dabei mitunter auf ein- und dasselbe Signal schreiben. Diese Eigenschaft gestattet die Modellierung von Busleitungen mit VHDL.

Die Syntax der Signaldeklaration lautet wie folgt:

```
SIGNAL   sig_name_1 { , sig_name_n }
         : type_name [ := def_value ];
```

Der Ausdruck `def_value` ist hier ebenfalls ein optionaler typkonformer Ausdruck, der den initialen Wert des Signals festlegt (siehe Variablendeklaration).

Signaldeklarationen dürfen an folgenden Stellen der Design-Einheiten stehen:

- im `ENTITY`-Deklarationsteil,
- im `ARCHITECTURE`-Deklarationsteil,
- im `PACKAGE`,
- im `BLOCK`-Deklarationsteil.

```
SIGNAL flag_1  : boolean := true;          -- default: true
SIGNAL flag_2  : boolean;                  -- default: false
SIGNAL s1, s2  : bit;                      -- default: '0'
SIGNAL d_value : integer RANGE 0 TO 1023 := 1; -- default: 1
```

3.3.4 Aliase

Um ein Objekt oder einen Teil eines Objektes mit einem anderen Namen und einem anderen Untertyp (z.B. inverse Indizierung) ansprechen zu können, gibt es die Möglichkeit von Aliasen, deren Deklaration nachstehende Syntax besitzt:

```
ALIAS alias_name :  alias_type IS
                    aliased_object;
```

```
SIGNAL  bus_16   : bit_vector (0 TO 15);
ALIAS   b16      : bit_vector (15 DOWNTO 0) IS bus_16;
ALIAS   bus_low  : bit_vector (0 TO 7)  IS bus_16 (0 TO 7);
ALIAS   bus_high : bit_vector (0 TO 7)  IS bus_16 (8 TO 15);
```

Mit der Überarbeitung der Norm wurde der Einsatzbereich für Aliase erweitert. Nun können auch Typen und Unterprogramme mit Aliasen versehen werden. Die erweiterte Syntax in ✓**93** lautet:

```
ALIAS alias_name [ : alias_type]
                    IS   aliased_object;

ALIAS alias_name    IS   aliased_type;

ALIAS alias_name    IS   aliased_subprogram
              [[arg_1_type {, arg_n_type }]
                [RETURN result_type]];
```

Die Angabe des Alias-Typs ist bei Objekten in der neuen Norm optional. Die fett gedruckten, eckigen Klammern sind Teil der Alias-Deklarationssyntax. Einige Beispiele für die neue Norm (✓**93**):

```
ALIAS bus_low  IS bus_16 (0 TO 7);           -- Kurzform
TYPE  chars    IS ('1','2','3','4,','5');    -- in my_pack
ALIAS one2five IS work.my_pack.chars;        -- Typ-Alias
ALIAS und      IS "AND" [bit, bit RETURN bit]; -- Fkt.-Alias
```

3.3.5 Implizite Deklaration

Einen Sonderfall bei den Deklarationen bilden die ganzzahligen Laufvariablen in FOR-Schleifenkonstrukten (FOR...LOOP und FOR... GENERATE). Diese müssen nicht explizit deklariert werden.

3.3.6 Weitere Deklarationen

Neben den erwähnten Objektklassen gibt es in VHDL noch einige weitere Elemente, die vor ihrer Verwendung deklariert werden müssen:

- Unterprogramme (Funktionen und Prozeduren),
- Schnittstellen von VHDL-Modellen, d.h. die Schnittstellensignale (PORT) und Parameter (GENERIC) eines Modells,
- Ein- und Ausgabeargumente von Unterprogrammen,
- Komponenten.

3.4 Ansprechen von Objekten

3.4.1 Objekte mit einfachem Typ

VHDL-Objekte (Konstanten, Variablen und Signale), die einen einfachen Datentyp haben, werden durch ihren Namen referenziert:

```
PROCESS
   CONSTANT  tpd_default : time := 4 ns;
   VARIABLE  delay_lh    : time;
   VARIABLE  distance    : length;    -- Typ length von oben
BEGIN
   distance := 5 inch + 3 cm; -- distance = 155 mm
   delay_lh := tpd_default;   -- Zuweisung ueber Referenz-
   ...                        -- namen; delay_lh = 4 ns
END PROCESS;
```

3.4.2 Objekte mit Feldtyp

Um Einzelelemente von Feldern anzusprechen, muß neben dem Referenznamen des Feldes auch die Position des bzw. der Einzelelemente mit angegeben werden. Dies kann auf unterschiedliche Art und Weise erfolgen.

Indexed Names

Das direkte Ansprechen von Feldelementen geschieht über entsprechende Ausdrücke, die dem Referenznamen in runden Klammern nachgestellt werden. Die Anzahl der Ausdrücke muß mit der Dimension des Feldes übereinstimmen:

```
array_name (    index_1_type_expression
           { ,  index_n_type_expression  } )
```

Ein Beispiel zur Verwendung von "indexed names":

```
ARCHITECTURE arch_types OF types IS
   TYPE vector IS ARRAY (positive RANGE <>) OF bit;
   TYPE matrix IS ARRAY
        (positive RANGE <>, positive RANGE <>) OF bit;
   CONSTANT c1: vector (1 TO 4)         := "1001";
   CONSTANT c2: matrix (1 TO 3, 1 TO 3) :=
                                        ("100","010","111");
BEGIN
   PROCESS
      VARIABLE v_1, v_2, v_4 : bit;
      VARIABLE v_3 : array_of_array;  -- Dekl. siehe oben
   BEGIN
      v_1 := c1 (3);           -- Vektorelement, v_1 = '0'
      v_2 := c2 (1,2);         -- Matrixelement, v_2 = '0'
      v_4 := c2 (1);           -- !!! illegal
      v_3 (red)(234) := 1024;  -- Element aus array_of_array
      v_3 (red) := (1,4,3,0,1,...);   -- hier eindimens. An-
      ...                              -- sprechen erlaubt
   END PROCESS;
END arch_types;
```

Sliced Names

Um mehrere Einzelelemente eines eindimensionalen Feldes (Vektors) gleichzeitig anzusprechen, kann man zusammenhängende Elemente durch Angabe des diskreten Bereichs innerhalb des Vektors sozusagen "herausschneiden". Im Englischen spricht man von einem "slice":

```
vector_name (slice_low TO slice_high)
vector_name (slice_high DOWNTO slice_low)
```

Die Bereichsgrenzen müssen dabei dem Indextyp des Vektors entsprechen und im deklarierten Bereich liegen. Entspricht die Bereichsdefinition nicht der bei der Deklaration angegebenen Zählrichtung (`TO`, `DOWNTO`) oder ist die Länge des Bereiches Null, so spricht man von einem "null slice".

```
PROCESS
   VARIABLE v_1, v_2 : bit_vector (0 TO 3) := "1111";
   CONSTANT c_1      : bit_vector := b"1001_0111"; -- 0 TO 7
BEGIN
   v_1 := c_1 (2 TO 5);   -- v_1 = "0101"
   v_2 := c_1 (5 TO 8);   -- !!! illegal: Index 8 zu gross
   ...
END PROCESS;
```

Aggregate

Um mehrere, nicht unbedingt zusammenhängende Elemente eines Feldes mit einem Ausdruck zuzuweisen, bedient man sich eines sog. Aggregats (im Englischen "aggregate").

Durch Kommata getrennt werden Elementzuweisungen in einer runden Klammer aneinandergereiht, die entweder Einzelelementzuweisungen oder Zuweisungen über einen zusammenhängenden Bereich sind. Außerdem kann durch das Schlüsselwort `OTHERS` allen noch nicht zugewiesenen Elementen ein Wert gegeben werden. Man unterscheidet die Zuweisung durch Position ("positional association") und die Zuweisung durch explizites Zuordnen ("named association").

Bei der "**positional association**" korrespondieren das Ziel der Zuweisung und das Element im Aggregat durch ihre Position. Die Zuweisung kann nur erfolgen, wenn im Aggregat alle vorhergehenden Elemente aufgeführt worden sind.

Alternativ dazu kann mit dem Zuweisungszeichen "`=>`" ein Element direkt angesprochen werden. Man spricht von "**named association**". Mit dem Zeichen "`|`" können mehrere Elementzuweisungen zusammengefaßt werden:

```
[ elements_1 { | elements_n }  => ]
                                element_value
```

Es gibt folgende Möglichkeiten der Elementauswahl (*`elements`*) in Aggregaten:

- `single_element`
- `range_low TO range_high`

- `range_high DOWNTO range_low`
- `OTHERS`

Eine Mischung von "positional" und "named association" ist nicht möglich (Ausnahme: `OTHERS`). Die mit dem Schlüsselwort `OTHERS` beginnende Elementzuweisung kann nur an letzter Stelle innerhalb des Aggregats stehen.

Bei mehrdimensionalen Feldern kann jeweils nur die erste Dimension mit einem Aggregat belegt werden. Auf der rechten Seite des Zuweisungspfeiles stehen dann Werte von (n-1)-dimensionalem Typ.

```
PROCESS
   CONSTANT start:                  integer := 1;
   CONSTANT finish:                 integer := 8;
   TYPE int_vector IS ARRAY (start TO finish) OF integer;
   VARIABLE v0, v1, v2, v3, v4:     int_vector;
BEGIN
-- positional association,          v0 = (5,2,3,1,4,4,2,1)
   v0 := (5, 2, 3, 1, 4, 4, 2, 1);
-- !!! illegal: Mix aus positional und named association
   v1 := (6, 2, 4 => 1, 5 => 3, OTHERS => 0);
-- named association,               v2 = (8,1,1,1,8,8,8,8)
   v2 := (2 TO 4 => start, 1 | 5 TO 8 => finish);
-- named association mit OTHERS,    v3 = (24,0,0,0,0,0,0,24)
   v3 := (start | finish => 3*8, OTHERS => 0);
-- slice und aggregate,             v4 = (0,0,0,8,2,2,2,2)
   v4 (1 TO 3) := (0, 0, 0);
   v4 (4 TO 8) := (8, 2, 2, 2, 2);
   ...
END PROCESS;
```

3.4.3 Objekte mit zusammengesetztem Typ

Bei zusammengesetzten Typen ("records") spricht man die Einzelelemente mit sog. "selected names" an:

```
record_name.record_element_name
```

Der Referenzname des zusammengesetzten Typs (`record_name`) wird in der Objektdeklaration festgelegt, während der Name des anzusprechenden Einzelelements (`record_element_name`) aus der Typdeklaration stammt.

Die Zuweisung von kompletten Records oder von Einzelelementen kann auch über ein Aggregat ("positional" oder "named") erfolgen.

```
PROCESS
   VARIABLE v_1, v_2, v_3 : complex;     -- Deklaration s.o.
BEGIN
   v_1.real_part := 1.0;                 -- selected name
   v_1.imag_part := v1.real_part;        -- selected name
   v_2           := (2.0, 1.7);          -- posit. Aggregate
   v_3           := (real_part => 3.2,   -- named  Aggregate
                     imag_part => 3.0);
END PROCESS;
```

3.5 Attribute

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

Attribute werden unter Angabe des Objekt- oder Typnamens als Prefix folgendermaßen verwendet:

```
obj_or_type_name'attr_1_name{'attr_n_name}
```

Attribute können also auch mehrfach angewandt werden. Dabei ist jedoch zu beachten, daß der Ergebnistyp des ersten Attributs und der Prefixtyp des zweiten Attributs übereinstimmen müssen.

Es gibt sowohl eine Reihe von vordefinierten Attributen (Abschnitt 6.2), als auch die Möglichkeit, benutzerdefinierte Attribute zu deklarieren und mit Werten zu versehen (Abschnitt 11.1).