

VHDL Kompakt

Andreas Mäder

Inhaltsverzeichnis

1	Konzepte von VHDL	1
1.1	Entwurfsparadigmen	1
1.2	Bibliotheken und compilierbare Einheiten	3
1.2.1	Package	4
1.2.2	Entity	4
1.2.3	Architecture	5
1.2.4	Configuration	7
1.3	Simulation	9
2	Datentypen	11
2.1	Skalare	11
2.2	komplexe Typen	13
2.3	Untertypen und Alias	19
2.4	Attribute	20
3	Bezeichner und Deklarationen	23
4	Ausdrücke	26
5	Sequenzielle Beschreibungen	30
5.1	Anweisungen	31
5.2	Unterprogramme	36
6	Signale	41
6.1	Deklaration	41
6.2	Signalzuweisungen im Prozess	42
6.3	Implizite Typauflösungen und Bustreiber	44
6.4	Attribute	47
7	Konkurrente Beschreibungen	48
8	Strukturbeschreibungen	52
8.1	Hierarchische Beschreibungen	52
8.1.1	Benutzung von Packages	55
8.1.2	Konfigurationen	55
8.1.3	Parametrisierung von Entities durch generische Werte	58
8.2	Strukturierende Anweisungen	60

INHALTSVERZEICHNIS

9 Bibliotheken und Packages	62
9.1 Package	62
9.2 VHDL-Einheiten, Dateien und Bibliotheken	64
A Syntaxbeschreibung	66
A.1 Übersicht	66
A.2 Bibliothekseinheiten	69
A.3 Deklarationen / Spezifikationen	73
A.4 sequenzielle Anweisungen	87
A.5 konkurrente Anweisungen	100
A.6 Sprachstandard	107
A.7 std_logic_1164	109
A.8 numeric_std / numeric_bit	110
A.9 textio	112
A.10 std_logic_textio	112
A.11 Attribute	113
A.12 reservierte Bezeichner	114
Literaturverzeichnis	115
Standards	116
WWW-Links	118
Index	119

Kapitel 1

Konzepte von VHDL

VHDL ist eine Hardwarebeschreibungssprache; dabei steht der Name für:
VHSIC Hardware Description Language
Very High Speed Integrated Circuit

VHDL wurde 1983 vom amerikanischen „Department of Defense“ initiiert und ist seit Ende 1987 als IEEE Standard 1076 genormt [IEEE 1076 87]. Seitdem hat sich VHDL (neben Verilog) zur *Standard*-Hardwarebeschreibungssprache entwickelt. Der Sprachstandard (Syntax und Semantik) wird regelmäßig überarbeitet [IEEE 1076 93; IEEE 1076 02; IEEE 1076 08] und seit 2004 ist VHDL als IEC Standard [IEC 61691-1-1 04] weltweit festgelegt. Daneben wurden Erweiterungen vorgenommen, die zusätzliche Aspekte behandeln, wie

- die Modellierung von Hardware und Zellbibliotheken [IEC 61691-2 01; IEC 61691-5 04]
- die Synthese von Hardware [IEC 61691-3-3 01; IEC 62050 04]
- mathematische Typen und Funktionen [IEC 61691-3-2 01]
- die Modellierung und (Co-) Simulation analoger Schaltungen [IEC 61691-6 09]

1.1 Entwurfsparadigmen

Die Sprache VHDL dient der Beschreibung und Simulation digitaler Systeme und deren Umgebung. Das Entwurfsziel kann ein FPGA, ein ASIC oder eine ganze Platine sein. Alle, während des Entwurfsvorgangs anfallenden Beschreibungen der Schaltung, werden von Sprachumfang abgedeckt. Dabei ist die Schaltung jederzeit simulierbar. In VHDL sind die folgenden Konzepte verwirklicht:

Hierarchie Die Unterteilung des Entwurfs in (Teil-)Komponenten wird unterstützt. Der Gesamtentwurf wird dann hierarchisch aus diesen Teilen zusammengesetzt. Die Komplexität dieser Teile kann vom einfachen Gatter (z.B. NAND) bis hin zu komplexen Funktionseinheiten (z.B. Prozessorkern) reichen.

Modelle Jede Design-Einheit (Entity) kann auf unterschiedliche Weise beschrieben sein. Dabei ist grundsätzlich zwischen folgenden Möglichkeiten zu unterscheiden:

Verhalten ist die algorithmische Beschreibung mit den Mitteln einer höheren Programmiersprache. Sowohl sequenzielle als auch parallele Abläufe können modelliert werden.

1. Konzepte von VHDL

Datenfluss beschreibt den Aufbau von Datenpfaden, wobei die Operationen auf den Daten als *elementare* Funktionen vorhanden sind — Mischform zwischen Struktur- und Verhaltensbeschreibung.

Struktur ist die direkte Umsetzung von Schaltplänen. Komponenten werden in einer Hierarchie miteinander verbunden.

Durch die Definition benutzereigener Datentypen kann die Aufgabe unabhängig von konkreten Hardwarerealisierungen spezifiziert werden.

Datenhaltung Das in der Sprache benutzte Bibliothekskonzept erlaubt:

- den Zugriff auf gemeinsame Datenbestände durch Arbeitsgruppen
- die Wiederverwendung bestehender (Teil-)Entwürfe
- das Einbinden herstellereigener Bibliotheken (z.B. für Standardzellen)

Alternativen — Schlagwort: *exploring the design-space* Die Trennung von Schnittstelle und der Implementation der Schaltung ermöglicht Entwurfsalternativen. Zu einer Schnittstelle können mehrere Realisierungen existieren, die sich meist auch hinsichtlich des Abstraktionsgrads unterscheiden.

Abstraktionsebenen

Im Entwurf werden verschiedene Abstraktionsebenen unterschieden, auf jeder Ebene dienen unterschiedlichste Beschreibungsformen dazu das Verhalten zu modellieren. Komplexe Modelle lassen sich aus Strukturen elementarer Elemente hierarchisch aufbauen. Jedes dieser „Primitive“ ist wiederum eine komplexe Beschreibung der nächstniedrigeren Ebene. Zusätzlich zu den Strukturbeschreibungen werden zum Teil auch geometrische Strukturen benutzt, die, als Repräsentation eines späteren Layouts, die Anordnung der Teile beschreiben.

	Verhalten	Struktur	aus
Architekturebene	Leistungsanforderungen	Netzwerk	Prozessoren, Speicher Busse, Controller...
Algorithmen- / funktionale Ebene	Algorithmen formale Funktionsbeschr.	Blockschaltbild	Hardware Module
Register-Transfer Ebene	Daten- und Steuerfluss endliche Automaten	RT-Diagramm	Register, Multiplexer Decodierer, ALUs...
Logikebene	boole'sche Gleichungen	Logiknetzliste	Flipflops, Gatter...
elektrische Ebene	Differenzialgleichungen	elektr. Schaltbild	Transistoren, Kondensatoren Dioden, Widerstände...

Entwurfsvorgehen

Der eigentliche Entwurfsprozess, bei Verwendung von VHDL, entspricht einem *top-down* Vorgehen.

Algorithmen-design Ausgangspunkt ist die Verhaltensbeschreibung der Schaltung, deren Funktionalität durch Simulationen geprüft wird. So können verschiedene Algorithmen implementiert und miteinander verglichen werden.

Auf oberster Ebene ist dies eine Beschreibung der zu entwerfenden ICs oder Systems, sowie eine Testumgebung, die das Interface zu der Schaltung darstellt.

Top-Down Strukturierung Im weiteren Vorgehen wird die Schaltung in Funktionsblöcke gegliedert, so dass man eine Strukturbeschreibung erhält.

Diese Vorgehensweise – Algorithmischer Entwurf von Funktionseinheiten, hierarchische Verfeinerung und Umsetzung in Strukturbeschreibungen – wird rekursiv ausgeführt, bis man letztendlich bei Elementen einer Zellbibliothek angekommen ist und die Schaltung praktisch realisiert werden kann.

Durch den Einsatz von Synthesewerkzeugen wird die Entwurfsaufgabe (auf den unteren Abstraktionsebenen) dabei zunehmend vereinfacht: ausgehend von Verhaltensbeschreibungen werden Netzlisten für Zielbibliotheken generiert. Derzeitiger Stand der Technik ist, dass die Synthese für Logik (Schaltnetze) und für endliche Automaten problemlos beherrscht wird. Für die Synthese komplexerer Algorithmen gibt es viele gute Ansätze, die zumindest bei Einschränkungen auf bestimmte Anwendungsfelder (Einschränkung des Suchraums), mit den Entwürfen guter Designer konkurrieren können.

1.2 Bibliotheken und compilierbare Einheiten

Die Entwürfe sind in Bibliotheken organisiert, wobei die Bibliotheken jeweils compilierten und durch den Simulator ausführbaren VHDL-Code enthalten. Bibliotheken können folgende vier Teile enthalten:

package : globale Deklarationen
entity : Design – Sicht von Außen (black box)
architecture : Design Implementation
configuration: Festlegung einer Design-Version (Zuordnung: entity – architecture)

Neben herstellereigenen- und benutzerdefinierten Bibliotheken gibt es zwei Standardbibliotheken:

WORK : Default-Bibliothek des Benutzers. Wenn nicht anders angegeben, dann ist WORK die Bibliothek, mit der die Programme arbeiten.
STD : enthält die beiden Packages STANDARD und TEXTIO mit vordefinierten Datentypen und Funktionen.

Compilation, Elaboration und Simulation, Synthese ...

VHDL-Beschreibungen werden in mehreren Schritten bearbeitet:

1. Die *Analyse* (Compilation) prüft die Syntax und die Konsistenz des VHDL-Codes und schreibt die Ausgabe (in einem programmspezifischen Format) in die entsprechende Bibliothek, normalerweise WORK.
2. Vor der weiteren Verarbeitung muss die Hierarchie aufgelöst und parametrisierbare Elemente entsprechend bearbeitet werden. Dieser Schritt der *Elaboration* wird oft gemeinsam mit der nachfolgenden Simulation oder Synthese ausgeführt.
3. Bei der *Simulation* von Elementen wird dann die Funktion der eingegebenen Schaltung überprüft.

Bei der *Synthese* wird der (zuvor simulierte) VHDL-Code so umgesetzt, dass er letztendlich als Hardware (FPGA, ASIC) realisiert wird.

1. Konzepte von VHDL

Die genaue Vorgehensweise hängt von den jeweils vorhandenen Programmen ab, für die Simulation wären dies beispielsweise:

Simulator		Analyse	Elaboration	Simulation
SYNOPSISYS	VSS	vhdlan		vhdl _{sim} , vhdl _{dbx}
	Cyclone	vhdlan	cylab	cysim
	Scirocco	vhdlan	scs	scsim
CADENCE	Leapfrog	cv	ev	sv
	NC-Sim	ncvhdl	ncelab	ncsim
MENTOR GRAPHICS	ModelSim	vcom		vsim

1.2.1 Package

Deklarationen die in mehreren Entwürfen benutzt werden, z.B.: Komponenten, Unterprogramme (Funktionen, Prozeduren) oder Typen, Konstanten, Dateien... , lassen sich in Packages sammeln und in Bibliotheken hinterlegen.

Neben eigenen Bibliotheken werden so auch die Zellbibliotheken der ASIC-Hersteller ausgeliefert. Auch die Hersteller von CAD-Software stellen Hilfsroutinen und Funktionen als Packages bereit, die den Umgang mit den Werkzeugen erleichtern.

Die Handhabung von Bibliotheken und die Syntax von package-Deklarationen ist im abschließenden Kapitel 9 ab Seite 62 beschrieben.

1.2.2 Entity

Ein entity definiert für eine Komponente des Entwurfs die externe Sichtweise. Dabei werden der Name, die Ein- und Ausgänge und zusätzliche Deklarationen festgelegt. Die interne Realisierung wird dann als, der Entity zugehörige, architecture beschrieben. Eine entity kann als Komponente anderer Entwürfe in deren Hierarchie eingebunden werden.

Syntax

```
entity <entityId> is
  [ <generic declaration> ]
  [ <port declaration> ]
  [ <local declarations> ]
  [begin                                     normalerweise nicht benutzt
    <passive statements> ]
end [entity] [ <entityId> ];

<generic declaration> ::=                               Parameter
  generic ( <generic list> : <typeId> [ := <expression> ] { ;
           <generic list> : <typeId> [ := <expression> ] } );

<port declaration> ::=                                  Ein- und Ausgänge
  port ( <port list>      : [ <mode> ] <typeId> [ := <expression> ] { ;
        <port list>      : [ <mode> ] <typeId> [ := <expression> ] } );
<mode> ::= in|out|inout|buffer                          „Richtung“
```

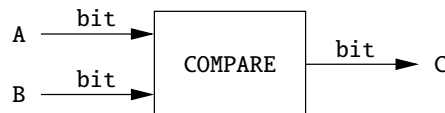
Die Entity-Deklaration kann neben den Ein- und Ausgängen (ports) auch Parameter der Entity (generics) festlegen. Mit ihnen lassen sich interne Eigenschaften und Wortbreiten einstellen. In den Architekturen entsprechen die Generics dann Konstanten. Beispiele dazu

folgen in Kapitel 8, ab Seite 58 und 60. Das nachfolgende Beispiel deklariert einen Bitvergleich.

Beispiel

```
entity COMPARE is
  port (A, B: in bit;
        C: out bit);
end entity COMPARE;
```

Ein-/Ausgänge: Richtung und Datentyp



1.2.3 Architecture

Eine Architektur beschreibt das Verhalten, bzw. die interne Realisierung einer vorher analysierten *entity*. Dabei können mehrere Architekturen für eine *entity* deklariert werden (Unterstützung von Alternativen), die Auswahl einer bestimmten *architecture* erfolgt dann später über eine *configuration*.

Syntax

```
architecture <architectureId> of <entityId> is
  [ <local declarations> ]
begin
  <statements>
end [architecture] [ <architectureId> ];

<local declarations> ::=
{use <...> } |
{function <...> } | {procedure <...> } |
{type <...> } | {subtype <...> } |
{file <...> } | {alias <...> } |
{component <...> } | { <configSpec> } |
{constant <...> } |
{signal <...> }
```

Der Deklarationsteil definiert Typen, Unterprogramme und Konstanten, die in dieser Architektur benötigt werden, während die Anweisungen *<statements>* das eigentliche Design beschreiben. Für diese Beschreibung gibt es drei *Stile*: Verhalten, Datenfluss, Struktur. Innerhalb des Anweisungsteils kann einer oder eine beliebige Kombination dieser *Stile* benutzt werden.

Verhalten

Das Grundkonstrukt der Verhaltensbeschreibung ist der *process*:

interner Aufbau Ein VHDL-Prozess ist einem sequenziellen Task einer Programmiersprache vergleichbar, mit den üblichen Konzepten:

- sequenzielle Abarbeitung der Anweisungen
- Kontrollanweisungen zur Steuerung des Ablaufs

1. Konzepte von VHDL

- Verwendung lokaler Variablen, -Datentypen
- Unterprogrammtechniken (Prozeduren und Funktionen)

Aktivierung Da das Ziel der VHDL Beschreibung ein durch den Simulator ausführbares Verhalten ist, gibt es spezielle Konstrukte, die festlegen wann der Prozess zu aktivieren ist — im Gegensatz zu Programmen in herkömmlichen Sinne sind Hardwareelemente *immer, gleichzeitig* aktiv.

Ein Prozess hat deshalb entweder eine *sensitivity list* oder enthält *wait*-Anweisungen. Beide Methoden bewirken, dass bei der Änderung von Eingangswerten der Architektur der Prozess von Simulator aktiviert wird, die Anweisungen sequenziell abgearbeitet werden und dadurch neue Ausgangswerte erzeugt werden.

Ein-/Ausgabe Nur über *Signale* können Prozesse nach außen hin aktiv werden. Diese Signale können zum einen Ein- und Ausgänge der Schaltung sein, zum anderen können auch mehrere Prozesse über (architektur-)interne Signale kommunizieren.

In Gegensatz zu den Signalen können die *Variablen* des Prozesses nur prozessintern verwendet werden und stellen so etwas wie lokalen Speicher dar.

Um Parallelitäten im Verhalten einer *architecture* zu beschreiben, können innerhalb des Anweisungsteils beliebig viele Prozesse beschrieben werden.

Beispiel

```
architecture ARCH_BEHAV of COMPARE is
begin
  P1: process (A, B)                                Prozess, sensitiv zu Eingängen A und B
  begin
    if (A = B) then
      C <= '1' after 1 ns;
    else
      C <= '0' after 2 ns;
    end if;
  end process P1;
end architecture ARCH_BEHAV;
```

Datenfluss

Bei dieser Beschreibung wird der Datenfluss über kombinatorische logische Funktionen (Addierer, Komparatoren, Decoder, Gatter...) modelliert.

Beispiel

```
architecture ARCH_DATFL of COMPARE is
begin
  C <= not (A xor B) after 1 ns;                    Datenfluss von Eingängen, über
end architecture ARCH_DATFL;                       xor und not, zum Ausgang
```

Struktur

Strukturbeschreibungen sind Netzlisten aus Bibliothekselementen: diese Elemente werden instanziiert und über Signale miteinander verbunden.

Beispiel

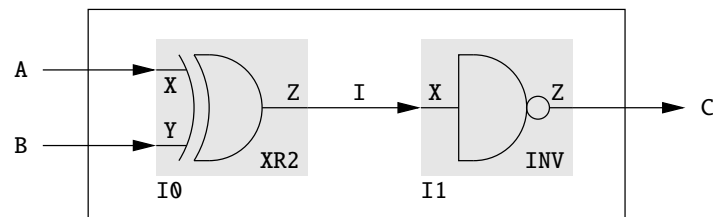
```

architecture ARCH_STRUC of COMPARE is
    signal I: bit;
    component XR2 is
        port (X,Y: in bit; Z: out bit);
    end component XR2;
    component INV is
        port (X: in bit; Z: out bit);
    end component INV;
begin
    I0: XR2 port map (A,B,I);
    I1: INV port map (I,C);
end architecture ARCH_STRUC;
    
```

lokales Signal

Komponentendeklaration
der Bibliothekselemente

Beschreibung der Netzliste
Benutzung der Komponenten



1.2.4 Configuration

Durch Konfigurationen kann der Entwerfer zwischen verschiedenen Design-Alternativen und -Versionen auswählen. Dabei bestimmt eine Konfiguration, welche Realisierung – von möglicherweise mehreren vorhandenen Architekturen – für ein Entity aus der Bibliothek, benutzt wird. Dabei kann ein Konfiguration zwei Arten hierarchischer Zuordnungen enthalten:

- Auf oberster Ebene wird eine Architektur für die top-level Entity ausgewählt.
- Innerhalb der Architektur werden für instanziierte Komponenten Paare aus Entity und Architektur bestimmt. Diese *Bindung* der Komponenten kann beliebig weit in die Hierarchie hinein stattfinden.

Existiert zu einer entity keine expliziten Konfiguration, so wird jeweils die (zeitlich) zuletzt analysierte Architektur benutzt — die *null* Konfiguration.

1. Konzepte von VHDL

Syntax

```
configuration <configurationId> of <entityId> is
  for <architectureId>
    {<component configuration>}
  end for;
end [configuration] [<configurationId>];

<component configuration> ::=
  for <instance>: <componentId>
    use entity          [<libraryId>.]<entityId>[(<archId>)] [<mapping>] ;|
    use configuration  [<libraryId>.]<configId>          [<mapping>] ;
  [ for <architectureId>
    {<component configuration>}
    end for; ]
  end for;

<instance> ::= <label>{, <label>} | others | all
<mapping>  ::= [<generic map>] [<port map>]
```

In dem nachfolgenden Beispiel wird die entity COMPARE in einer ALU benutzt (strukturelle Beschreibung der architecture ... of ALU).

Beispiel

```
entity ALU is                                     Entity Beschreibung der ALU
  port ( opcode: ...
end entity ALU;

architecture FIRST of ALU is                       Architektur der ALU (Strukturbeschreibung)
  component COMPARE is
    port (A, B: in bit; C: out bit);
  end component COMPARE;
  ...
begin
  I0: COMPARE port map (S, D, Q);                  Instanz I0 der entity COMPARE
  ...
end architecture FIRST;

configuration FAST_ALU of ALU is
  for FIRST                                       Architektur die konfiguriert wird
    for I0: COMPARE use entity WORK.COMPARE(ARCH_BEHAV);
    ...                                         legt Entitynamen und dessen Architektur fest
    ...
  end configuration FAST_ALU;
```

Da Konfigurationen separate Entwurfseinheiten sind, können sie auch direkt für die Simulation benutzt werden. In obigem Beispiel wäre es möglich diese Konfiguration zu simulieren als: `<mySimulator> WORK.FAST_ALU`

Später in dem Abschnitt zur Hierarchie 8.1.2, ab Seite 55, werden Konfigurationen noch genauer vorgestellt.

1.3 Simulation

Die Simulation von VHDL-Schaltungen (Entity + Architektur) ist integraler Bestandteil der Semantik. In dem Sprachstandard wird deshalb auch ein Simulationsalgorithmus definiert, dessen Verhalten die Abarbeitung konkurrenter und sequenzieller VHDL-Codeabschnitte definiert.

Für den Simulator besteht eine VHDL-Beschreibung aus einer Menge *konkurrent aktiven* Codes — wie bei der Hardware: Funktionseinheiten arbeiten gleichzeitig! Wie oben bei den Architekturen beschrieben, können dies Anweisungen einer Datenflussbeschreibung, Instanzen in einer hierarchischen Modellierung oder Prozesse sein. Die Prozesse wiederum, begrenzen *sequenziell abzuarbeitenden* Code, der in Verhaltensbeschreibungen benutzt wird. Der VHDL-Simulationsalgorithmus beschreibt jetzt das prinzipielle Verfahren, wie die konkurrenten Modelle vom Simulator behandelt werden müssen, damit das Ergebnis unabhängig von der sequenziellen Abarbeitungsreihenfolge des Programms ist.

Für die Simulation wird ein ereignisgesteuerter Simulator angenommen, der eine zentrale Liste (*Schedule*) besitzt in der zukünftige Ereignisse stehen. Diese wurden durch den bisherigen Verlauf der Simulation erzeugt. Zwei Arten von Ereignissen sind möglich:

1. Wertewechsel von Signalen (Verbindung der konkurrenten Teile untereinander):
Ereignis = Zeitpunkt + Signal + Wert
2. Abarbeitung von sequenziellem Code in einem Prozess:
Ereignis = Zeitpunkt bzw. Bedingung + Prozess + Einsprungstelle

Eine fiktive Zeiteinheit (*delta-Time*) erlaubt die Behandlung von Signalzuweisungen ohne Verzögerungszeit. Dadurch kann der Simulator die Grundschleife der Simulation mehrfach durchlaufen, ohne dass die simulierte Zeit fortschreitet. Um die Wirkungsweise von Signalzuweisungen besser zu verstehen, ist der Simulationszyklus hier kurz skizziert:

1. Aktivierung des Zyklus zu einem Zeitpunkt t_0 . Alle Ereignisse, die für diesen Zeitpunkt im Schedule sind, sind abzuarbeiten. Dazu werden alle Signalzuweisungen des Schedule ausgeführt und die neu berechneten Werte der Signale gespeichert.
2. Parallele Anweisungen, die diese Signale als Eingänge benutzen – zu den Signalen *sensitiv* sind –, müssen in diesem Zyklus ausgeführt werden. Zusammen mit Prozessen aus dem Schedule wird so eine Menge aktiver Anweisungen bestimmt.
3. Jede dieser konkurrenten Anweisungen / Prozesse wird von dem Simulator abgearbeitet. Die Signalwerte werden der unter Schritt 1. berechneten Datenstruktur entnommen. Wenn die Simulation der Anweisungen neue Ereignisse erzeugt, dann werden diese in extra Datenstrukturen verwaltet.

Wegen dieser Trennung *können sich die Signalwerte innerhalb eines Simulationszyklus nicht ändern* — die Abarbeitung ist unabhängig von der Reihenfolge.

Der sequenzielle Code in Prozessen wird, beginnend an der Einsprungstelle, so lange abgearbeitet, bis der Prozess terminiert. Gerät die Abarbeitung in eine Endlosschleife, dann... hat man ein Problem.

4. Nach der Abarbeitung aller aktiven Anweisungen ist der Zyklus für den Zeitpunkt t_0 abgeschlossen. Die simulierten Ereignisse werden entfernt und „neue Ereignisse“ aus Schritt 3. werden im Schedule wirksam.

1. Konzepte von VHDL

- Die simulierte Zeit schreitet fort und der nächste Zyklus beginnt mit Schritt 1. Waren bei den unter 3. berechneten Ereignissen welche ohne Zeitfortschritt, beispielsweise Signalzuweisungen ohne Verzögerungszeit, dann spricht man von einem *delta*-Zyklus und die aktuelle Zeit ist $t_0 + \delta$. Ansonsten schreitet die simulierte Zeit bis zum nächsten t_1 fort.

Dieser Algorithmus beschreibt ein Modell, das ein VHDL-Designer „vor Augen“ haben sollte, um die Abarbeitung von Signalzuweisungen und Prozessen in der Simulation zu verstehen. Die tatsächlich in den Simulatoren implementierten Algorithmen können allerdings stark davon abweichen. Insbesondere für eine schnelle Simulation (Programmlaufzeit), sind andere Vorgehensweisen, die nicht auf (dynamischen) Ereignislisten basieren, besser geeignet. Meist hat der Entwerfer in der Simulation verschiedene Möglichkeiten zwischen sehr kompatiblen Modi, die man für das Debugging braucht, und stark optimierten Algorithmen zu wählen.

Kapitel 2

Datentypen

VHDL ist eine stark *typisierte* Sprache, d.h. Konstante, Variablen und Signale haben einen, durch deren Deklaration festgelegten Typ. Bei der Codeanalyse wird die Konsistenz der Datentypen bei Operationen und Zuweisungen überprüft. Gegebenenfalls müssen *Konvertierungsfunktionen* benutzt werden.

Die Typen des Package `standard` sind immer bekannt (= Sprachstandard). Im praktischen Umgang mit VHDL werden meist noch weitere Typen, beziehungsweise Deklarationen, aus zusätzlichen Packages gebraucht:

Package	Bibliothek
<code>textio</code>	<code>std</code>
<code>std_logic_1164</code>	<code>ieee</code>
<code>std_logic_textio</code>	<code>ieee</code>
<code>numeric_std, numeric_bit</code>	<code>ieee</code>

2.1 Skalare

Die einfachen VHDL-Datentypen sind denen in Standard-Programmiersprachen vergleichbar:

character entspricht dem ISO 8859-1 Zeichensatz, die darstellbaren Zeichen werden dabei in einfache Hochkommas eingeschlossen: `'0'... '9'`, `'a'... 'z'`, `'A'... 'Z'` usw.¹

bit Die beiden logischen Werte `'0'` und `'1'` sind `bit` Literale.¹

std_logic / std_ulogic Der „IEEE Standard 1164“ [IEEE 1164 93] ist extern definiert: in der Bibliothek `ieee`, Package `std_logic_1164`. Dort wird ein Logiksystem mit neun Signalwerten, bzw. Treiberstärken definiert, das für die Simulation und Synthese von Hardware besser geeignet ist als der Typ `Bit`. Die Werte sind im einzelnen:

¹Wegen der Typbindung in VHDL kann es notwendig sein, zur Unterscheidung der Typen deren Werte explizit zu klassifizieren: `character('1')`
`bit('1')`

Syntax

```
type <typeId> is ( <enumLiteral> {,<enumLiteral>} );
```

Dabei können für *<enumLiteral>* beliebige Bezeichner (erstes Beispiel) aber auch Literale (character Literale im zweiten Beispiel) benutzt werden.

Beispiel

```
type AMPEL is (ROT, GELB, GRUEN);           für Ampelsteuerung
type FOURVAL is ('X', '0', '1', 'Z');       vierwertige Logik für Simulation
```

2.2 komplexe Typen

Array Wie in Programmiersprachen bestehen Arrays aus durchnummerierten Elementen gleichen Typs. Zu den skalaren Standardtypen sind folgende Array-Typen vordefiniert:

string Array Typ zu character — in standard

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

bit_vector Array Typ zu bit — in standard

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

std_logic_vector / std_ulogic_vector als Array zu std_logic, bzw. std_ulogic

```
type std_logic_vector is array ( natural range <>) of std_logic;
```

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

signed / unsigned Mit den Bitvektoren (bit-, std_logic_vector) sind zwar logische Operationen möglich, aber Zahlendarstellung und Arithmetik sind nicht definiert. Dazu wird entweder das Package numeric_std oder numeric_bit benötigt!

```
type signed is array (natural range <>) of std_logic|bit;
```

```
type unsigned is array (natural range <>) of std_logic|bit;
```

Zusammen mit den Typen werden die arithmetischen Operatoren für eine Zahlendarstellung als 2'Komplement- und als vorzeichenlose Zahl festgelegt. Die Operatoren sind auf Seite 27 beschrieben.

Die Deklaration eigener Array Typen besteht dabei aus der Angabe des Arraytyp-Namens, der Beschreibung des Index (der Indices bei mehrdimensionalen Arrays) durch Index-Typ(en) und Index-Bereich(e) und der Angabe des Element-Typs.

Syntax

```
type <typeId> is array (<index>) of <element typeId>;
```

<index> ::= <range>		integer Bereich
<typeId>		Aufzählungstyp
<typeId> range <range>		allgemeiner Bereich
<typeId> range <>		unbegrenzt, Bereichs bei Obj.-Dekl.

Nachfolgend werden einige Eigenschaften von Arrays anhand von Beispielen genauer erläutert.

2. Datentypen

Index-Typen Neben den üblichen Integer-Indices können auch eigene Aufzählungstypen (-untertypen) benutzt werden.

Beispiel

```
type INSTRUCTION is (ADD, SUB, LDA, LDB, STA, STB, OUTA);
subtype FLAGS is integer range (0 to 7);
...
type INSTR_FLAG is array (INSTRUCTION) of FLAGS;           Array von Flag-Werten
```

Benutzung als Laufindex Indices können innerhalb von Schleifen über Variablen inkrementiert/dekrementiert werden.

Beispiel

```
...
process ...
  variable INFO    : bit_vector (0 to 49);
  variable START   : integer;
  variable OUTBYTE : bit_vector (0 to 7);
begin
  for I in 0 to 7 loop
    OUTBYTE(I) := INFO(I + START);
  end loop;
end process;
```

Unbegrenzte Indices Oft werden Indices über den gesamten Wertebereich eines Typs deklariert und dann später bei der Deklaration von Objekten werden erst die Bereichseinschränkungen vorgenommen.

Beispiel

```
type bit_vector is array (natural range <>) of bit;
...
variable BYTE: bit_vector (0 to 7);           Deklaration aus standard
```

Index-Bereiche Bei der Angabe ganzer Bereiche – meist durch Integer-Bereiche – ist die *Laufrichtung* des Index' wichtig.

Beispiel

```
type AVEC is array (0 to 3) of bit;
type BVEC is array (3 downto 0) of bit;
...
variable AV: AVEC;
variable BV: BVEC;
...
AV := "0101";           ⇒ AV(0)='0' AV(1)='1' AV(2)='0' AV(3)='1'
BV := "0101";           ⇒ BV(0)='1' BV(1)='0' BV(2)='1' BV(3)='0'
```

Array-Zuweisungen können über die *Position*, den *Namen* oder gemischt erfolgen.

Syntax

[<typeId> ']	optionale Typ-Qualifizierung
([<expression> {, <expression> } ,]	Position
[<selector> => <expression> {,	+ Name
<selector> => <expression> } ,]	
[others => <expression>])	+ Rest

Beispiel

```
variable C      : bit_vector (0 to 3);
variable H, I, J, K: bit;
```

mögliche Zuweisungen

```
C := "1010";           4-bit string
C := H & I & J & K;    Konkatination
C := ('1', '0', '1', '0');  Aggregat
```

Aggregatzuweisungen

```
C := ('1', I, '0', J or K);      Position
C := (0 => '1', 3 => J or K, 1 => I, 2 => '0');  Name
C := ('1', I, others => '0');    gemischt
```

Ausschnitte werden über die Indizes gebildet.

Beispiel

```
variable A: bit_vector (3 downto 0);
variable B: bit_vector (8 downto 1);
...
B(6 downto 3) := A;
```

4 bit Slice von A

Mehrdimensionale Arrays werden durch Aufzählung von Indexbereichen erzeugt.

Beispiel

```
type MEMORY is array (0 to 7, 0 to 3) of bit;      8 x 4 bit Array
...
constant ROM: MEMORY := (('0','0','0','0'),
                          ('0','0','0','1'),
                          ('0','0','1','0'),
                          ('0','0','1','1'),
                          ('0','1','0','0'),
                          ('0','1','0','1'),
                          ('0','1','1','0'),
                          ('0','1','1','1'));
variable DATA_BIT : bit;
...
DATA_BIT := ROM(5,3);           = '1'
```

Es lassen sich auch Arrays von Arrays bilden; die Indizes werden hierbei getrennt

2. Datentypen

behandelt.

Beispiel

```
type WORD is array (0 to 3) of bit;           4 bit Zeile
type MEMORY is array (0 to 7) of WORD;       8 × 4 bit Array
...
constant ROM: MEMORY := (('0','0','0','0'),
                          ('0','0','0','1'),
                          ...
                          ('0','1','1','1'));
variable DATA : WORD;
variable DATA_BIT : bit;
variable ADDR, INDEX : integer;
...
DATA := ROM(ADDR);
DATA_BIT := ROM(ADDR)(INDEX);
```

Array Untertypen lassen sich zu bestehenden Arrays, bzw. zu unbegrenzten Arrays definieren.

Beispiel

```
subtype BYTE is bit_vector (7 downto 0);      unbegrenzter Typ bit_vector
```

Record Elemente verschiedener Typen (Skalare oder zusammengesetzte Typen) können mit Hilfe von Records zusammengefasst werden, um abstrakte Datenmodelle zu bilden. Die einzelnen Felder des Records werden über die Namen dereferenziert.

Syntax

```
type <typeId> is record
  {<fieldId> : <typeId>;}
end record [<typeId>];
```

Beispiel

```
type TWO_DIGIT is record                       Zahlen von -99 bis +99
  SIGN : bit;
  MSD : integer range 0 to 9;
  LSD : integer range 0 to 9;
end record TWO_DIGIT;
...
process ...
  variable ACNT, BCNT: TWO_DIGIT;
begin
  ACNT.SIGN := '1';                             Zugriff auf Felder
  ACNT.MSD := 1;
  ACNT.LSD := ACNT.MSD;
  ...
  BCNTR := TWO_DIGIT('0',3,6);                   Aggregat, Typ-qualifiziert
  ...
```

Dateitypen Die Sprache VHDL erlaubt die Deklaration beliebiger Dateitypen und -Objekte, sowie den Lese- und Schreibzugriff auf diese Dateien. Die Handhabung des Dateizugriffs wurde vom ersten Standard von '87 [IEEE 1076 87] zu den aktuelleren Versionen ab VHDL'93 [IEEE 1076 93] stark verändert, außerdem ist die Implementation an den Simulator gebunden — die Dateien sind i.A. nicht portabel.

Syntax

```
type <typeId> is file of <base typeId>;
```

Deshalb sollte man *Textdateien* benutzen, für deren Handhabung das package TEXTIO vordefiniert ist. Dort sind die Datentypen text und line deklariert, sowie Funktionen die den (aus Programmiersprachen gewohnten) Zugriff auf Textdateien erlauben; so ist es beispielsweise möglich Testvektoren aus einer Datei einzulesen und in der Simulation zu benutzen.

TEXTIO

type line is access string;	Typen
type text is file of string;	
file input : text open read_mode is "STD_INPUT";	Dateien
file output : text open write_mode is "STD_OUTPUT";	
read (<lineVar>, <vhdObj> [, <status>]);	
readline (<fileObj>, <lineVar>);	
write (<lineVar>, <vhdObj> [, right left, <width>]);	
write (<lineVar>, <realObj> [, right left, <width>, <digits>]);	
write (<lineVar>, <timeObj> [, right left, <width>, <unit>]);	
writeline (<fileObj>, <lineVar>);	
endfile (<fileObj>) : boolean	
zusätzlich in ieee.std_logic_textio	
read (<lineVar>, <vhdObj> [, <status>]);	bin.
hread (<lineVar>, <vhdObj> [, <status>]);	hex.
oread (<lineVar>, <vhdObj> [, <status>]);	oct.
write (<lineVar>, <vhdObj> [, right left, <width>]);	
hwrite (<lineVar>, <vhdObj> [, right left, <width>]);	
owrite (<lineVar>, <vhdObj> [, right left, <width>]);	

2. Datentypen

In dem Beispiel einer Testumgebung werden zwei Zahlen aus einer Datei gelesen und ein Ergebnis in eine Ausgabedatei geschrieben.

Beispiel

```
use std.textio.all;
...
signal M, N, P      : integer;
...
FILE_P: process
  file IN_DAT       : text  open read_mode  is "testdat.in";    Eingabedatei
  file OUT_DAT      : text  open write_mode is "testdat.out";   Ausgabedatei
  variable LI, LO : line;
  variable MV, NV, PV : integer;
begin
  while not (endfile(IN_DAT)) loop                               bis zum Dateiende
    readline (IN_DAT, LI);                                       liest Zeile aus Datei
    read (LI, MV);    M <= MV;                                    liest Werte aus Zeile
    read (LI, NV);    N <= NV;                                    Format: <M> <N>
    wait for 10 ns; PV := P;          write (LO, PV);           schreibt Wert in Zeile
    writeline (OUT_DAT, LO);                                               schreibt Zeile in Datei
  end loop;
  file_close(IN_DAT);
  file_close(OUT_DAT);
  wait;
end process FILE_P;
...
```

Zugriffstypen Wie Zeiger in Programmiersprachen können Zugriffstypen dazu benutzt werden dynamisch Speicherstrukturen zu allozieren. Variablen vom Typ `access` sind Zeiger auf skalare oder komplexe Datenstrukturen.

Syntax

```
type <typeId> is access <base typeId>;
```

Für die Arbeit mit Zugriffstypen sind zwei Operatoren definiert.

`new` wird bei *Zuweisungen* an eine Variable des Zugriffstyps benutzt, um Speicher anzufordern; dabei sind Initialisierungen möglich. Bei Zeigern auf unbeschränkte Array-Typen, wie z.B. bei `string`, sind Bereichseinschränkungen notwendig.

`deallocate` gibt die Speicherbereiche wieder frei, wobei eine Variable des Zugriffstyps als Parameter übergeben wird.

Beispiel

```
type CELL;                               unvollständige Typdeklaration
type LINK is access CELL;                Zugriffstyp
type CELL is record                       genaue Typdeklaration
  VALUE : integer;
  NEXTP : LINK;
end record CELL;
...
```

```

variable HEAD, TMP : LINK;                                Zeiger auf CELL
...
TEMP := new CELL'(0, null);                               neues Element, mit Initialisierung
for I in 1 to 5 loop
  HEAD      := new CELL;                                  weitere Elemente
  HEAD.VALUE := I;                                       Zugriff auf record
  HEAD.NEXTP := TEMP;
  TEMP      := HEAD;
end loop;
...
Speicherfreigabe
deallocate(TEMP);

Speicheranforderung
... := new CELL;                                         neues Element
... := new CELL'(I, TEMP);                               ... mit Initialisierung

... mit notwendiger Bereichsbeschränkung
... := new bit_vector (15 downto 0);                    durch Index
... := new bit_vector'("001101110");                  durch Initialisierung

```

2.3 Untertypen und Alias

Untertypen Zu vordefinierten, bzw. zu eigenen Typen lassen sich Untertypen bilden, dies geschieht durch Einschränkung der Wertebereiche und/oder bei Array-Typen Begrenzung der Indexbereiche.

Bereichseinschränkungen lassen sich zwar auch bei Objektdeklarationen angeben, Untertypen bieten aber den Vorteil, dass solche Begrenzungen zentral vorgenommen werden können.

Syntax

```

subtype <sub typeId> is <base typeId> [range <range>] ;      Wert begrenzt
subtype <sub typeId> is <base typeId>                       Index begrenzt
  (<range> | <typeId> {, <range> | <typeId>});

<range> ::= <low expr> to      <high expr> |
           <high expr> downto <low expr>

```

Beispiel

```

subtype DIGIT is integer range 0 to 9;                      Untertyp zu integer
...
variable MSD, LSD: DIGIT;

— ist äquivalent zu —
variable MSD, LSD: integer range 0 to 9;

```

2. Datentypen

Alias Deklarationen Zu Typen, Unterprogrammen oder Objekten können Alias-Namen vergeben werden. Teilstrukturen komplexer Typen können so direkt über Namen referenziert werden.

Syntax

```
alias <aliasId> : <typeId> is <aliasObj>;
```

In dem Beispiel wird der Record-Typ TWO_DIGIT (s.o.) mit Hilfe von alias Anweisungen nachgebildet.

Beispiel

```
signal ACNT: bit_vector(1 to 9);                                vergl. voriges Beispiel
alias  SIGN: bit is ACNT(1);
alias  MSD : bit_vector(1 to 4) is ACNT(2 to 5);
alias  LSD : bit_vector(1 to 4) is ACNT(6 to 9);
...
SIGN := '1';                                                  Benutzung
MSD  := "1001";
LSD  := MSD;

— identische Zuweisung —
ACNT := "110011001";
```

2.4 Attribute

Symbolische Attribute in VHDL erlauben allgemeineren Code zu schreiben, da Konstanten oder Literale nicht an mehreren Stellen stehen müssen, sondern über Attributierungsmechanismen zum Zeitpunkt der Übersetzung ermittelt werden.

Es können zwar auch eigene Attribute deklariert und spezifiziert werden, da die Auswertung aber durch die VHDL-verarbeitenden Programme erfolgt, werden im folgenden nur die im Standard vordefinierten Attribute vorgestellt.

Dimensionierung Die Attribute ermitteln für Array- und Aufzählungstypen, beziehungsweise Variablen und Signale dieser Typen, Bereichsgrenzen und Längen. Bei mehrdimensionalen Arrays wird die Ordnungsnummer des Index mit angegeben.

Syntax

```
Bereichsgrenzen
<type/obj>'left  [(⟨n⟩)]      : <index>    -linke Grenze (⟨n⟩)
<type/obj>'right [(⟨n⟩)]      : <index>    -rechte Grenze (⟨n⟩)
<type/obj>'high  [(⟨n⟩)]      : <index>    -obere Grenze (⟨n⟩)
<type/obj>'low   [(⟨n⟩)]      : <index>    -untere Grenze (⟨n⟩)

Array- und Typdefinition
<type/obj>'length[(⟨n⟩)]      : <integer>  -Anzahl der Elemente (⟨n⟩)
<type/obj>'ascending[(⟨n⟩)]   : <boolean>  -aufsteigender Index (⟨n⟩)

Bereiche
<type/obj>'range [(⟨n⟩)]      : <range>    -Indexbereich (⟨n⟩)    to|downto
<type/obj>'reverse_range[(⟨n⟩)] : <range>    -Indexbereich (⟨n⟩)    downto|to
```

Beispiel

```

Bereichsgrenzen
type T_RAM_DAT is array (0 to 511) of integer;
variable RAM_DAT : T_RAM_DAT;
...
for I in RAM_DAT'low to RAM_DAT'high loop
    ...

```

s.u. Bereiche

```

Bereichsgrenzen mehrdimensional
variable MEM (0 to 15, 7 downto 0) of MEM_DAT;
...
MEM'left(1)           = 0
MEM'right(1)          = 15
MEM'left(2)           = 7
MEM'right(2)          = 0
MEM'low(2)            = 0
MEM'high(2)           = 7

```

Arraylängen

```

type BIT4 is array (3 downto 0) of bit;
type BITX is array (10 to 30) of bit;
...
BIT4'length           = 4
BITX'length           = 21

```

Bereiche

```

function BV_TO_INT (VEC: bit_vector) return integer is
...
begin
    for I in VEC'range loop
        ...
    end loop
end function;

```

2. Datentypen

Ordnung Die Attribute ermitteln für Aufzählungstypen Werte, Ordnungszahlen und übergeordnete Typen (bei Untertypen).

Syntax

Wertermittlung			
$\langle type \rangle$ 'succ	$(\langle typeExpr \rangle)$: $\langle value \rangle$	-nachfolgender Wert zu $\langle typeExpr \rangle$
$\langle type \rangle$ 'pred	$(\langle typeExpr \rangle)$: $\langle value \rangle$	-vorhergehender –"– $\langle typeExpr \rangle$
$\langle type \rangle$ 'leftof	$(\langle typeExpr \rangle)$: $\langle value \rangle$	-linker –"– $\langle typeExpr \rangle$
$\langle type \rangle$ 'rightof	$(\langle typeExpr \rangle)$: $\langle value \rangle$	-rechter –"– $\langle typeExpr \rangle$
Ordnung			
$\langle type \rangle$ 'pos	$(\langle typeExpr \rangle)$: $\langle integer \rangle$	-Position von $\langle typeExpr \rangle$
$\langle type \rangle$ 'val	$(\langle position \rangle)$: $\langle integer \rangle$	-Wert von $\langle position \rangle$
Ein- und Ausgabe			
$\langle type \rangle$ 'image	$(\langle typeExpr \rangle)$: $\langle string \rangle$	-Text $\langle typeExpr \rangle$
$\langle type \rangle$ 'value	$(\langle string \rangle)$: $\langle value \rangle$	-Wert zu $\langle string \rangle$
übergeordnete Typen – als Argument weiterer Attribute			
$\langle type \rangle$ 'base	$(\langle attribute \rangle)$: $\langle baseType \rangle$	-Basistyp zu $\langle type \rangle$

Beispiel

```
type    COLOR is (RED, BLUE, GREEN, YELLOW, BROWN, BLACK);
subtype TLCOL is COLOR range RED to GREEN;
...
COLOR'low           =    RED
COLOR'succ(RED)     =    BLUE
TLCOL'base'right
spc13= BLACK
COLOR'base'left     =    RED
TLCOL'base'succ(GREEN) = YELLOW
```

Kapitel 3

Bezeichner und Deklarationen

Mit Ausnahme der reservierten Wörter kann der Benutzer beliebige Bezeichner vergeben, um Objekte zu benennen. Dabei gilt:¹

- Zeichensatz 'a'... 'z', '0'... '9', '_'.
- das erste Zeichen muss ein Buchstabe sein.
- keine Unterscheidung zwischen Groß- und Kleinschreibung in VHDL

Bei Verwendung von Bibliotheken und Packages müssen die Elemente gegebenenfalls über komplette Namen dereferenziert werden, wie: $\langle libraryId \rangle . \langle packageId \rangle . \langle itemId \rangle$

Kommentare beginnen mit zwei -- Zeichen und gehen bis zum Ende der Zeile.

Konstanten legen einmalig Werte innerhalb von `package`, `entity` oder `architecture` fest.

Syntax

```
constant <identifier> : <typeId> [<range>] [:= <expression>];
```

Beispiel

```
constant VCC    : real      := 4.5;
constant CYCLE  : time      := 100 ns;
constant PI     : real      := 3.147592;
constant FIVE   : bit_vector := "0101";
```

Variablen speichern Werte innerhalb eines `process` und werden dort, durch den Kontrollfluss gesteuert, sequenziell benutzt. Variablen können *nicht* benutzt werden, um Informationen zwischen Prozessen auszutauschen.²

Syntax

```
variable <identifier list> : <typeId> [<range>] [:= <expression>];
```

¹In VHDL'93 wurden erweiterter Bezeichner eingeführt, diese sollten aber aus Kompatibilitätsgründen nicht benutzt werden.

²„globale Variablen“ aus VHDL'93 werden hier nicht weiter beschrieben.

3. Bezeichner und Deklarationen

Bei der Deklaration können die Wertebereiche der Variablen eingeschränkt werden und die Initialisierung mit Werten ist möglich.

Beispiel

```
variable INDEX      : integer range 1 to 50 := 10;
variable CYC_TIME  : range 10 ns to 1 ms  := CYCLE;
variable REG       : std_logic_vector (7 downto 0);
variable X, Y      : integer;
```

Signale verbinden Entwurfseinheiten (*Entities*) untereinander und übertragen Wertewechsel innerhalb der Schaltung. Die Kommunikation zwischen Prozessen findet über Signale statt. Wegen der besonderen Bedeutung von Signalen in VHDL wird auf sie später, in Kapitel 6, noch genauer eingegangen.

Syntax

```
signal <identifier list> : <typeId> [ <range> ] [ := <expression> ] ;
```

Beispiel

```
signal COUNT       : integer range 1 to 50;
signal GROUND      : bit := '0';
signal D_BUS       : std_logic_vector (15 downto 0);
```

Achtung: Signale können *nicht* innerhalb eines Prozesses deklariert werden. Innerhalb eines process können sie zwar wie Variablen gelesen werden, aber Zuweisungen werden erst durch die Simulationszeit wirksam. Das heißt, dass aus Sicht des Prozesses Signalzuweisungen *nicht* in der sequenziellen Reihenfolge ausgeführt werden, sondern erst wenn der Prozess ein wait-Statement erreicht, beziehungsweise anhält.

Um den zeitlichen Charakter der Signalzuweisung hervorzuheben, wird auch ein anderer Zuweisungsoperator als bei Variablen benutzt. Zusätzlich können Verzögerungen bei der Signalzuweisung modelliert werden.

Beispiel

```
signal S, A, B : signed(7 downto 0);
...
S <= A + B after 5 ns;
```

Die Benutzung von Signalen im sequenziellen Ablauf des Prozesses führt daher oft zu (unerwartet) fehlerhaften Ergebnissen. Deshalb sollte man in Prozessen mit *Schreib- und Leseoperationen* mit Variablen „rechnen“ und die Ergebnisse dann abschließend, vor dem nächsten wait, an die Ausgangssignale zuweisen.

Dateien besser Dateiobjekte, können nach Deklaration der Dateitypen definiert werden. Der Bezeichner der Datei (*file string*) muss den Konventionen des Betriebssystems entsprechen.

Syntax

```
file <identifier> : <typeId> is [in|out] <file string>;                                VHDL '87
file <identifier> : <typeId> [[open <mode>] is <file string>];                        '93
<mode> ::= read_mode|write_mode|append_mode
```

Zusammen mit den Dateitypen sind auch Lese- und Schreiboperationen definiert.

Syntax

```
endfile      (<fileObj>)                               : boolean
read        (<fileObj>, <vhdObj>);
write       (<fileObj>, <vhdObj>);

file_close  (<fileObj>);                               neu in VHDL'93
file_open   ([<status>], <fileObj>, <file string> [, <mode>]);
read        (<fileObj>, <arrayObj>, <length>);
```

Abschließend noch zwei Anmerkungen zum Umgang mit Variablen und Signalen:

Initialisierung Werden Variable oder Signale bei der Deklaration nicht explizit initialisiert, so werden sie bei der Simulation folgendermaßen vorbesetzt:

Aufzählungstypen : der erste Wert der Aufzählungsliste

integer, real : der niedrigste darstellbare Wert

Dies wird gerade bei den Aufzählungstypen oft ausgenutzt, beispielsweise indem man den Startzustand eines endlichen Automaten als ersten definiert oder bei dem Typ `std_logic` wo mit 'U' (*nicht initialisiert*) begonnen wird.

Allerdings wirken sich solche Initialisierungen nur auf die Simulation aus und können bei der Synthese von Gatternetzlisten aus RT-Beschreibungen nicht berücksichtigt werden. Hier muss explizit ein „Reset“ eingebaut, also im VHDL-Code beschrieben werden.

Bereichseinschränkung Um Variable oder Signale für die Hardwaresynthese zu benutzen, sollte entsprechend der geplanten Bitbreite eine Bereichseinschränkung vorgenommen werden — dies ist gerade bei integer-Typen notwendig, da sonst 32-bit Datenpfade generiert werden.

Beispiel

```
signal CNT100 : integer range 0 to 99;                unsigned 7-bit
signal ADDR_BUS : std_logic_vector (7 downto 0);      8-bit
```

Kapitel 4

Ausdrücke

Um Ausdrücke zu bilden, gelten die folgenden Regeln:

- Ausdrücke werden aus Operatoren und Objekten, Literalen, Funktionsaufrufen oder Aggregaten gebildet.
- Die Operatoren besitzen unterschiedliche Prioritäten (siehe Nummerierung).
Alle Operatoren innerhalb einer Gruppe haben die gleiche Priorität!
Oft werden Fehler in boole'schen Ausdrücken gemacht, da `and` und `or` gleichwertig sind.
- Gegebenenfalls muss die Reihenfolge der Auswertung durch Klammerung festgelegt werden.
- Wegen der Typbindung müssen entweder explizite Angaben des Typs (Typqualifizierungen) oder Typkonvertierungen vorgenommen werden.

Der VHDL-Sprachstandard enthält die in der Tabelle aufgeführten Operatoren; `sll...ror` und `xnor` wurden in VHDL'93 [IEEE 1076 93] ergänzt.

Syntax

1. logische Operatoren	Typ- <i>a</i>	Typ- <i>b</i>	Typ-⟨ <i>op</i> ⟩
<code>and</code> $a \wedge b$	bit bit_vector boolean	= <i>a</i>	= <i>a</i>
<code>or</code> $a \vee b$	bit bit_vector boolean	= <i>a</i>	= <i>a</i>
<code>nand</code> $\neg(a \wedge b)$	bit bit_vector boolean	= <i>a</i>	= <i>a</i>
<code>nor</code> $\neg(a \vee b)$	bit bit_vector boolean	= <i>a</i>	= <i>a</i>
<code>xor</code> $\neg(a \equiv b)$	bit bit_vector boolean	= <i>a</i>	= <i>a</i>
<code>xnor</code> $a \equiv b$	bit bit_vector boolean	= <i>a</i>	= <i>a</i>
2. relationale Operatoren	Typ- <i>a</i>	Typ- <i>b</i>	Typ-⟨ <i>op</i> ⟩
<code>=</code> $a = b$	beliebiger Typ	= <i>a</i>	boolean
<code>/=</code> $a \neq b$	beliebiger Typ	= <i>a</i>	boolean
<code><</code> $a < b$	skalarer Typ 1-dim. Array	= <i>a</i>	boolean
<code><=</code> $a \leq b$	skalarer Typ 1-dim. Array	= <i>a</i>	boolean
<code>></code> $a > b$	skalarer Typ 1-dim. Array	= <i>a</i>	boolean
<code>>=</code> $a \geq b$	skalarer Typ 1-dim. Array	= <i>a</i>	boolean

3. schiebende Operatoren				
sll	$(a_{n-1-b} \dots a_0, 0_{b \dots 1})$	Typ- <i>a</i> bit_vector bit/bool-Array	Typ- <i>b</i> integer	Typ- $\langle op \rangle$ = <i>a</i>
srl	$(0_{1 \dots b}, a_{n-1} \dots a_b)$	bit_vector bit/bool-Array	integer	= <i>a</i>
sla	$(a_{n-1-b} \dots a_0, a_{0,b \dots 1})$	bit_vector bit/bool-Array	integer	= <i>a</i>
sra	$(a_{n-1}, 1 \dots b, a_{n-1} \dots a_b)$	bit_vector bit/bool-Array	integer	= <i>a</i>
rol	$(a_{n-1-b} \dots a_0, a_{n-1} \dots a_{n-b})$	bit_vector bit/bool-Array	integer	= <i>a</i>
ror	$(a_{b-1} \dots a_0, a_{n-1} \dots a_b)$	bit_vector bit/bool-Array	integer	= <i>a</i>
4. additive Operatoren				
+	$a + b$	Typ- <i>a</i> integer real phys. Typ	Typ- <i>b</i> = <i>a</i>	Typ- $\langle op \rangle$ = <i>a</i>
-	$a - b$	integer real phys. Typ	= <i>a</i>	= <i>a</i>
&	$(a_n \dots a_0, b_m \dots b_0)$	skalärer Typ 1-dim. Array	<i>a</i> -Skalar / Array	<i>a</i> -Array
5. vorzeichen Operatoren				
+	$+a$	Typ- <i>a</i> integer real phys. Typ	Typ- <i>b</i>	Typ- $\langle op \rangle$ = <i>a</i>
-	$-a$	integer real phys. Typ		= <i>a</i>
6. multiplikative Operatoren				
*	$a * b$	Typ- <i>a</i> integer real phys. Typ	Typ- <i>b</i> = <i>a</i>	Typ- $\langle op \rangle$ = <i>a</i>
/	a / b	integer real phys. Typ	= <i>a</i>	= <i>a</i>
mod	Modulus	integer	= <i>a</i>	= <i>a</i>
rem	Teilerrest	integer	= <i>a</i>	= <i>a</i>
7. sonstige Operatoren				
**	a^b	Typ- <i>a</i> integer real	Typ- <i>b</i> integer	Typ- $\langle op \rangle$ = <i>a</i>
abs	$ a $	integer real phys. Typ		= <i>a</i>
not	$\neg a$	bit bit_vector boolean		= <i>a</i>

Für die zusätzlichen Datentypen aus der Bibliothek IEEE sind die Standardoperatoren entsprechend überladen. Insbesondere bei den numerischen Typen signed und unsigned sind auch gemischte Operationen innerhalb der Zahlendarstellung mit den „passenden“ Integer-Typen möglich. 2'Komplement : signed ↔ integer
vorzeichenlos : unsigned ↔ natural

std_logic_1164

and or nand	Typ- <i>a</i> std_(u) logic	Typ- <i>b</i> = <i>a</i>	Typ- $\langle op \rangle$ = <i>a</i>
nor xor xnor	std_(u) logic_vector		
not	_"_		= <i>a</i>

4. Ausdrücke

numeric_std / numeric_bit

and or nand nor xor xnor	Typ- <i>a</i> signed unsigned	Typ- <i>b</i> = <i>a</i>	Typ-⟨ <i>op</i> ⟩ = <i>a</i>
= /= < <= > >=	signed integer unsigned natural	⇒ ⇒	boolean boolean
shift_left shift_right rotate_left rotate_right sll srl rol ror	signed unsigned	integer	= <i>a</i>
+ -	signed integer unsigned natural	⇒ ⇒	signed unsigned
-	signed		signed
* / mod rem	signed integer unsigned natural	⇒ ⇒	signed unsigned
abs	signed		signed
not	signed unsigned		= <i>a</i>

Typqualifizierung

Qualifizierungen erlauben die explizite Angabe eines Typs, beispielsweise bei Literalen, wenn keine eindeutige Zuordnung möglich ist.

Syntax

```
⟨typeId⟩'⟨expression⟩
```

Beispiel

```
type MONTH is (APRIL, MAY, JUNE);
type NAMES is (APRIL, JUNE, JUDY);

... MONTH'(JUNE) ...           Monat
... NAMES'(JUNE) ...          Namen

variable SPEED : signed (3 downto 0);
...
if SPEED > signed'"0011"' then ...
```

Typkonvertierung

Für die Standardtypen sind Konvertierungsfunktionen vordefiniert, insbesondere bei Benutzung der „Bitvektoren“ signed, unsigned und std_logic_vector werden Konvertierungsfunktionen häufig benötigt.

Sprachstandard

integer	(⟨integer⟩ ⟨real⟩)	: integer	
real	(⟨integer⟩ ⟨real⟩)	: real	
⟨typeId⟩	(⟨relatedType⟩)	: ⟨typeId⟩	für „ähnliche“ Typen

std_logic_1164

to_bit	(⟨std_ulogic⟩ [, ⟨xMap⟩])	: bit	
to_bitvector	(⟨std_(u)logic_vector⟩ [, ⟨xMap⟩])	: bit_vector	
to_stdulogic	(⟨bit⟩)	: std_ulogic	
to_stdlogicvector	(⟨bit_vector⟩ ⟨std_ulogic_vector⟩)	: std_logic_vector	
to_stdulogicvector	(⟨bit_vector⟩ ⟨std_logic_vector⟩)	: std_ulogic_vector	

numeric_std / numeric_bit

to_integer	(⟨signed⟩)	: integer	
to_integer	(⟨unsigned⟩)	: natural	
to_signed	(⟨integer⟩, ⟨size⟩)	: signed	
to_unsigned	(⟨natural⟩, ⟨size⟩)	: unsigned	

„Ähnliche Typen“ sind Typ und Untertyp, bzw. Arrays davon. Für sie sind Typkonvertierungen implizit deklariert. In dem Beispiel wird ein integer-Literal in ein std_logic_vector umgerechnet.

Beispiel

```
signal SEL : std_logic_vector(3 downto 0);  
...  
SEL <= std_logic_vector(to_unsigned(3, 4));
```

Bei eigenen Typen müssen Konvertierungsfunktionen bei Bedarf durch den Benutzer angegeben werden; Funktionen sind in Abschnitt 5.2, Seite 36 beschrieben.

Beispiel

```
type FOURVAL is ('X', '0', '1', 'Z'); vierwertige Logik  
...  
function STD_TO_FOURVAL (S: std_logic) return FOURVAL is Konvertierungsfunktion  
begin  
  case S is  
    when 'L' | '0' => return '0';  
    when 'H' | '1' => return '1';  
    when 'Z' => return 'Z';  
    when others => return 'X'; 'U' 'X' 'W' '-'  
  end case;  
end function STD_TO_FOURVAL;  
...  
signal S : std_logic;  
signal SF : FOURVAL;  
...  
SF <= STD_TO_FOURVAL(S); Aufruf in Signalzuweisung
```

Kapitel 5

Sequenzielle Beschreibungen

Die zentrale Rolle bei sequenziellen Beschreibungen spielt der Prozess. Das `process`-Statement wird für die Verhaltensbeschreibung von Architekturen benutzt, und begrenzt einen Bereich, in dem Anweisungen sequenziell abgearbeitet werden.

Das `process`-Statement selber ist eine konkurrente Anweisung, d.h. es können beliebig viele Prozesse gleichzeitig aktiv sein, ihre Reihenfolge im VHDL-Code ist irrelevant. Weitere Erläuterungen zu der `process`-Anweisung und deren Abarbeitung in der Simulation finden sich in Abschnitt 7, Seite 48.

Syntax

```
[<label>:] process [(sensitivity list)] [is]
  [<local declarations>]
begin
  <sequential statements>
end process [<label>];

<local declarations> ::=
  {type <...>      } | {subtype <...>  } |
  {file <...>     } | {alias <...>   } |
  {function <...> } | {procedure <...> } |
  {constant <...>} |
  {variable <...> }
```

Hier noch eine Anmerkung: das optionale Label `<label>` ist bei der Simulation für das Debugging der Schaltung nützlich und sollte deshalb immer vergeben werden.

Das Beispiel simuliert die Auswahl des Minimums und Maximums aus drei Eingangswerten mit Hilfe von zwei Prozessen.

Beispiel

```
entity LOW_HIGH is
  port (A, B, C : in integer;           Eingänge
        MI, MA : out integer);         Ausgänge
end entity LOW_HIGH;
```

```

architecture BEHAV of LOW_HIGH is
begin
  L_P: process (A, B, C)                                Minimum bestimmen
    variable LO : integer := 0;
    begin
      if A < B    then LO := A;
                else LO := B;
      end if;
      if C < LO  then LO := C;
      end if;
      MI <= LO after 1 ns;
    end process L_P;

  H_P: process (A, B, C)                                Maximum bestimmen
    variable HI : integer := 0;
    begin
      if A > B    then HI := A;
                else HI := B;
      end if;
      if C > HI  then HI := C;
      end if;
      MA <= HI after 1 ns;
    end process H_P;
end architecture BEHAV;

```

5.1 Anweisungen

Signalzuweisung Die Signalzuweisung innerhalb von Prozessen ist in dem extra Abschnitt 6.2, Seite 42 beschrieben. Auch wenn sie in der sequenziellen Folge eines Prozesses steht, wird sie bei der Simulationsabarbeitung anders behandelt!

Syntax

```
[<label>:] <signalObj> <= [ <delay mode> ] <wave expression>;
```

Variablenzuweisung Die Typen der Objekte/Ausdrücke müssen bei der Zuweisung zusammenpassen, was gegebenenfalls Attributierungen oder Konvertierungsfunktionen erfordert.

Syntax

```
[<label>:] <variableObj> := <expression>;
```

If Verzweigung wie in Programmiersprachen; durch die Schachtelung von Bedingungen ergeben sich Präferenzen, die sich beispielsweise bei der Hardwaresynthese als geschachtelte (2-fach) Multiplexer wiederfinden.

5. Sequenzielle Beschreibungen

Syntax

```
[<label>:] if <condition> then
  <sequential statements>
{elsif <condition> then
  <sequential statements>}
[else
  <sequential statements>]
end if [<label>];
```

Case mehrfach-Verzweigung wie in Programmiersprachen; in der Hardwareumsetzung ergeben sich (im Idealfall) entsprechende Decodierer oder Multiplexer.

Syntax

```
[<label>:] case <expression> is
  {when <choices> => <sequential statements>}
end case [<label>];
```

<choices> ::= <value>		genau ein Wert
<value> {^ <value>}		Aufzählung
<value> to <value>		Bereich
others		alle übrigen

Für <expression> müssen alle möglichen Werte aufgezählt werden. Dies geht am einfachsten durch `when others` als letzte Auswahl.

Ein häufig gemachter Fehler ist, dass die metalogischen Werte von `std_logic` nicht berücksichtigt werden. In dem Decoderbeispiel ersetzt `others` nicht benötigte Fälle, während es im zweiten Beispiel den letzten Fall eines Multiplexers beschreibt — implizite Annahme, dass nur '0' oder '1' im Vektor enthalten sind.

Beispiel

```
case BCD is
  when "0000" => LED := "1111110";
  when "0001" => LED := "1100000";
  when "0010" => LED := "1011011";
  when "0011" => LED := "1110011";
  when "0100" => LED := "1100101";
  when "0101" => LED := "0110111";
  when "0110" => LED := "0111111";
  when "0111" => LED := "1100010";
  when "1000" => LED := "1111111";
  when "1001" => LED := "1110111";
  when others => LED := "-----";
end case;

case SEL is
  when "00" => 0 <= A;
  when "01" => 0 <= B;
  when "10" => 0 <= C;
  when others => 0 <= D;
end case;
```

Decoder: BCD zu 7-Segment

don't care: `std_logic_1164`

4-fach Multiplexer

Loop Modellierung verschiedener Schleifen. Neben der Vereinfachung von Ausdrücken, lassen sich so auch Wiederholungen von Strukturen, beispielsweise der Bitbreite entsprechend, beschreiben.

Syntax

```
[<label>:] while <expression> loop | boole'sche Bedingung
      for <rangeVar> in <range> loop | Schleifenvariable, Wertebereich
      loop | beliebig oft
      <sequential statements>
end loop [<label>];
```

Die Laufvariable der for-Schleife muss nicht extra deklariert werden; *<loopVariable>* gilt als lokale Variable in der Schleife; Zuweisungen sowie externer Zugriff sind nicht möglich.

Next bewirkt den vorzeitigen Abbruch eines Schleifendurchlaufs, die zusätzliche Angabe einer Bedingung ist möglich.

Syntax

```
[<label>:] next [<loop label>] [when <condition>];
```

Beispiel

```
L1: for I in 1 to ITER_MAX loop           äußere Schleife
    SUM := 0;
    L2: while SUM < SUM_MAX loop         innere Schleife
        ...
        next L1 when SUM < 0;           Sprung, neue Iteration
        ...
    end loop L2;
    ...
end loop L1;                             Sprungziel
```

Exit Sprung aus einer Schleife, optionale Bedingung.

Syntax

```
[<label>:] exit [<loop label>] [when <condition>];
```

Beispiel

```
A_IND := -1;
for I in Q'range loop
    if (Q(A_IND) = A) then A_IND := I;
                           exit;           Sprung aus Schleife
    end if;
end loop;
if (A_IND = -1) ...           Sprungziel
```

5. Sequenzielle Beschreibungen

Assert ermöglicht die Überprüfung von Bedingungen zur Laufzeit des VHDL-Simulators. Dies ist beispielsweise sinnvoll um Zeitbedingungen zu gewährleisten (set-up, hold...), um Bereichsgrenzen zu prüfen usw.

Syntax

```
[<label>:] assert <condition>
  [report <string expression>]
  [severity failure|error|warning|note];
```

Ist *<condition>* nicht erfüllt, dann wird eine Meldung *<string expression>* ausgegeben.

Beispiel

```
process (CLK, DIN)                                     Verhaltensmodell eines D-FF
  variable S_TIME: time := 0 ns;
  ...
begin
  ...
  assert (S_TIME > 1 ns)
    report "setup violation"
    severity warning;
  ...
```

Report ist die Ausgabe einer Meldung wie bei assert; die Anweisung ist in VHDL-'87 nicht enthalten.

Syntax

```
[<label>:] report <string expression>]
  [severity failure|error|warning|note];
```

Wait kontrolliert dynamisch die Ausführung/Unterbrechung des Prozesses. Dadurch wird das sequenziell berechnete Verhalten auf die simulierte Zeit abgestimmt. Signalabhängige oder periodische Aktivitäten lassen sich dadurch modellieren und neu berechnete Signalwerte werden wirksam. In den Abschnitten 6.2 und 1.3 (Seiten 43, 9) wird Wirkungsweise der wait-Anweisung genauer erklärt.

Syntax

```
[<label>:] wait
  [on <signalObj>{, <signalObj>}]
  [until <condition>]
  [for <time expression>];
```

Die *<sensitivity list>* eines process entspricht einem wait on ... am Ende des Prozesses. Es gibt vier Grundformen der wait-Anweisung:

wait on A, B; Unterbrechen des Prozesses, bis ein Ereignis geschieht:
ein Wertewechsel auf einem der Signale A oder B.

wait until X > 10; Unterbrechen des Prozesses, bis eine Bedingung erfüllt ist:
das Signal X einen Wert > 10 hat.

wait for 10 ns; Unterbrechen des Prozesses, bis eine Zeitspanne verstrichen ist:
10 ns simulierter Zeit.

`wait;` Unendlich langes Warten... Da ein VHDL-process immer aktiv ist, bietet diese Anweisung am Ende des Prozesses die einzige Möglichkeit ihn quasi anzuhalten. Dies wird beispielsweise in Testumgebungen oder bei Initialisierungen benutzt, die nur einmal abgearbeitet werden sollen.

In dem Beispiel wird eine Architektur modelliert, die über zwei Prozesse eine Erzeuger/Verbraucher Situation simuliert. Über ein einfaches Handshake-Protokoll (zwei Leitungen, mit je zwei aktiven Zuständen) werden die Prozesse synchronisiert.

Beispiel

```
entity PRODUCER_CONSUMER is
  ...
end entity PRODUCER_CONSUMER;

architecture BEHAV of PRODUCER_CONSUMER is
  signal PROD: boolean := false;
  signal CONS: boolean := true;
begin
  PROD_P: process
  begin
    PROD <= false;
    wait until CONS;
    ...
    PROD <= true;
    wait until not CONS;
  end process PROD_P;

  CONS_P: process
  begin
    CONS <= true;
    wait until PROD;
    CONS <= false;
    ...
    wait until not PROD;
  end process CONS_P;
end architecture BEHAV;
```

erzeugt Semaphor
verbraucht Semaphor

Erzeuger

produce...

Verbraucher

consume...

Return darf nur in Prozeduren und Funktionen benutzt werden und führt zum Verlassen des Unterprogramms. Bei Funktionen muss ein Rückgabewert angegeben werden.

Syntax

```
[<label>:] return [<expression>];
```

Prozeduraufrufe werden im nächsten Abschnitt auf Seite 38 vorgestellt.

Null ist die leere Anweisung, sie wird gelegentlich für case-Anweisungen gebraucht, in Verzweigungen ohne Aktion.

Syntax

```
[<label>:] null;
```

5.2 Unterprogramme

VHDL beinhaltet sowohl Prozeduren (mehrere Return-Werte via Parameter) als auch Funktionen (sie liefern genau einen Wert zurück) als Unterprogramme. Die Funktionen werden beispielsweise zur Typkonvertierung oder als Auflösungsfunktion benutzt, siehe Abschnitt 6.3, Seite 44).

Deklaration

Typischerweise werden Unterprogramme innerhalb des entsprechenden Kontexts definiert, also in einer `architecture` oder lokal in dem benutzenden `process`. Um Unterprogramme im Entwurf mehrfach zu nutzen, sollten sie in einem VHDL-Package deklariert werden. Dabei müssen die Deklarationen (wie auch das Package) in Kopf und „Body“ unterteilt werden, ein Beispiel folgt auf Seite 63.

Variablen

In Unterprogrammen können zwar lokale Variablen deklariert und benutzt werden, deren Werte sind aber nur bis zum Verlassen des Unterprogramms definiert — im Gegensatz zu Variablen im `process`, die einem lokalen Speicher entsprechen!

Function hat (meistens) mehrere Parameter und gibt genau einen Wert zurück — entspricht damit einem Ausdruck.

Syntax

```
function <functionId> [parameter declaration] return <typeId>;           nur Kopf

function <functionId> [parameter declaration] return <typeId> is
  [local declarations]
begin
  <sequential statements>                                           mit return
end [function] [functionId];

<parameter declaration> ::=
  ( [class] <formal list> : [in] <typeId> [:= <expression>] { ;
    [class] <formal list> : [in] <typeId> [:= <expression>] } )
  <class> ::= constant|signal|file                                   Objektklasse
```

Die Objektklasse wird meist nicht benötigt und deshalb weggelassen, wie auch die Richtung, die bei Funktionen immer `in` ist. Der Anweisungsteil muss so aufgebaut sein, dass *immer* eine `return`-Anweisung erreicht wird, diese muss allerdings nicht zwangsläufig die letzte Anweisung sein. `wait`-Anweisungen sind in Funktionen nicht erlaubt.

In dem Beispiel wird ein Bitvektor in eine Integer Zahl umgerechnet, dabei wird der Vektor als vorzeichenlose Zahl, mit MSB...LSB, interpretiert.

Beispiel

```

architecture ...
  ...
  function BV_TO_INT (VEC: bit_vector) return integer is
    variable INT: integer := 0;
  begin
    for I in VEC'range loop
      INT := INT * 2;
      if VEC(I) = '1' then INT := INT + 1;
      end if;
    end loop;
    return INT;
  end function BV_TO_INT;

begin
  ...
  process ...
    ...
    XINT := BV_TO_INT (XVEC);
    ...
  end process ...

```

lokale Variable

Funktionsaufruf
hier sequenziell

Procedure hat mehrere Parameter auf die lesend/schreibend zugegriffen werden kann. Bei der Deklaration wird dazu ein Modus als Wirkungsrichtung angegeben.

in Eingangswert, darf nur gelesen werden.

out Ausgangswert, darf nur auf der linken Seite von Zuweisungen stehen.

inout Ein-/Ausgangswert, kann in der Prozedur universell eingesetzt werden.

Für die Parameter sind außer Variablen auch Signale zulässig. Bei Ausgangsparametern ist dabei auf den „passenden“ Zuweisungsoperator zu achten: \leq oder $:=$

Prozeduren werden wie Anweisungen, sequenziell oder konkurrent, abgearbeitet.

Syntax

```

procedure <procedureId> [ <parameter declaration> ] ;
                                                                    nur Kopf

procedure <procedureId> [ <parameter declaration> ] is
  [ <local declarations> ]
begin
  <sequential statements>
end [procedure] [ <procedureId> ];

<parameter declaration> ::=
  ( [ <class> ] <formal list> : [ <mode> ] <typeId> [ := <expression> ] { ;
    [ <class> ] <formal list> : [ <mode> ] <typeId> [ := <expression> ] } )

<class> ::= constant | signal | variable | file
<mode>  ::= in | out | inout
                                                                    Objektklasse
                                                                    Wirkungsrichtung

```

5. Sequenzielle Beschreibungen

Die Prozedur des Beispiels dient, wie die Funktion oben, der Umrechnung eines Vektors in eine vorzeichenlose Integer-Zahl. Das Argument ist vom Typ `std_logic_vector` und ein zusätzliches Flag zeigt an, ob in der Eingabe andere Werte außer '0' und '1' enthalten sind.

Beispiel

```
architecture ...
...
procedure SV_TO_INT (VEC : in    std_logic_vector;
                    INT : inout integer;
                    FLG : out   boolean) is
begin
  INT := 0;
  FLG := false;
  for I in VEC'range loop
    INT := INT * 2;
    if   VEC(I) = '1'   then INT := INT + 1;
    elsif VEC(I) /= '0' then FLG := true;
    end if;
  end loop;
end procedure SV_TO_INT;

begin
...
process ...
...
  SV_TO_INT (XVEC, XINT, XFLG);
...
end process ...
```

lesen+schreiben: inout
Prozeduraufruf
hier sequenziell

Aufruf

Die Benutzung von Unterprogrammen im VHDL-Code kann sowohl im *sequenziellen* Kontext, also innerhalb von Prozessen oder anderen Unterprogrammen, als auch im *konkurrenten* Anweisungsteil einer Architektur erfolgen. Bei der Parameterübergabe hat man verschiedene Möglichkeiten die formalen Parameter (aus der Deklaration) durch aktuelle Parameter, bzw. Ausdrücke (bei *in*-Parametern), zu ersetzen.

- Über die Position, entsprechend der Reihenfolge bei der Deklaration
- Über den Namen: $\langle \text{formal parameter} \rangle \Rightarrow \langle \text{actual parameter} \rangle$
- Auch Mischformen aus Position und Name sind möglich.
- `open` steht für nicht benutzte Ausgangs- oder Eingangsparameter mit Default-Wert.

Syntax

```
[ <label> : ] <procedureId> [ ( [ <formal> => ] <actual> | open { ,  
                                [ <formal> => ] <actual> | open } ) ] ;
```

Overloading

Wie in einigen Programmiersprachen können auch in VHDL Funktionen und Prozeduren überladen werden, indem sie mehrfach definiert sind, sich dabei aber durch unterschiedliche Typen oder die Anzahl der Parameter unterscheiden. Beim Aufruf wird dann, entsprechend Anzahl und Typ der Argumente, die entsprechende Funktion/Prozedur ausgewählt. Durch Overloading können, trotz strikter Typbindung, Operatoren und Funktionen sehr allgemein benutzt werden.

Argument-Typ Zwischen den Unterprogrammen wird durch den Typ der Argumente unterschieden.

Beispiel

```
function DECR (X: integer) return integer is           integerArgument
begin
    return (X - 1);
end function DECR;

function DECR (X: real)    return real    is           realArgument
begin
    return (X - 1.0);
end function DECR;

...
variable A, B: integer;
...
B := DECR(A);                                         benutzt erste Funktion
```

Argument-Anzahl Zwischen der Unterprogrammen wird durch die Anzahl der Argumente unterschieden.

Beispiel

```
function MAX (A0, A1: integer) return integer is      2 Argumente
begin
    ...
end function MAX;

function MAX (A0, A1, A2: integer) return integer is 3 Argumente
begin
    ...
end function MAX;

function MAX (A0, A1, A2, A3: integer) return integer is 4 Argumente
begin
    ...
end function MAX;
```

Auch die bestehenden Operatoren können noch weiter überladen werden, was insbesondere bei zusätzlichen Arithmetiken (`numeric_bit`, `numeric_std`) oft benutzt wird.

Aber auch andere, herstellereigene Packages nutzen diese Möglichkeit, dabei werden neben eigenen Typen auch logische (`and`, `or`...), arithmetische (`+`, `-`, `*`...) und Vergleichsoperatoren (`=`, `/=`, `>`, `>=`...) auf diesen Typen definiert. Beispielsweise stellt SYNOPSIS für

5. Sequenzielle Beschreibungen

std_logic_1164 noch zwei zusätzliche Packages zur Verfügung, die eine vorzeichenlose-std_logic_unsigned oder 2'-Komplement Zahlendarstellung std_logic_signed direkt für den Datentyp std_logic_vector definiert.¹ Für Funktionen mit zwei Argumenten ist auch die gewohnte *Infix-Notation* möglich.

In dem Beispiel wird eine Addition für bit_vector definiert.

Beispiel

```
function "+" (L, R      : bit_vector) return bit_vector is
  constant SIZE      : natural := MAX(L'length, R'length);      siehe oben: MAX
  variable CAR       : bit      := '0';
  variable L_OP, R_OP : bit_vector (SIZE-1 downto 0);
  variable RESULT    : bit_vector (SIZE-1 downto 0);
begin
  if L'length = SIZE                                     L normieren
  then L_OP := L;
  else L_OP(SIZE-1 downto L'length) := (others=> '0');
       L_OP(L'length-1 downto 0)    := L;
  end if;
  if R'length = SIZE                                     R normieren
  then R_OP := R;
  else R_OP(SIZE-1 downto R'length) := (others=> '0');
       R_OP(R'length-1 downto 0)    := R;
  end if;
  for I in RESULT'reverse_range loop                    Volladdierer Schleife
    RESULT(I) := L_OP(I) xor R_OP(I) xor CAR;           Summe
    CAR       := (L_OP(I) and R_OP(I)) or (L_OP(I) and (R_OP(I) and CAR));      Carry
  end loop;
  return RESULT;
end function "+";
```

¹Der Standard ist die Benutzung des Package numeric_std und die Interpretation durch die beiden Datentypen signed/unsigned

Kapitel 6

Signale

Während die VHDL-Verhaltensbeschreibungen – Prozesse mit Variablen und sequenziellen Anweisungen – den algorithmischen Abläufen in Programmiersprachen entsprechen, besitzen Signale und konkurrente Blöcke Eigenschaften, die für Strukturbeschreibungen und deren Simulation typisch sind.

Dabei sind Signale die einzige Möglichkeit, quasi als *Leitungen*, die Elemente struktureller Beschreibungen miteinander zu verbinden sowie die Kommunikation zwischen Prozessen zu ermöglichen. Bei der Simulation wird eine zeitliche Ordnung von Ereignissen – im Sinne von Ursache und Wirkung – über Signale geregelt.

6.1 Deklaration

Signale können an folgenden Stellen im VHDL-Code deklariert werden:

1. innerhalb eines `package` für globale Signale.
2. als `port ...` der `entity`-Deklaration für `entity`-globale Signale.
3. innerhalb einer `architecture` als `architecture`-globale Signale. Im Allgemeinen werden alle Signale, die keine Ports sind, so deklariert.

Syntax

```
signal <identifier list> : <typeId> [ <range> ] [ := <expression> ] ;

<port declaration> ::=                               Ein- und Ausgänge
  port ( <port list>      : [ <mode> ] <typeId> [ := <expression> ] { ;
        <port list>      : [ <mode> ] <typeId> [ := <expression> ] } );
<mode> ::= in|out|inout|buffer                          „Richtung“
```

Für den Modus der Ein-/Ausgänge einer Portdeklaration gilt:

`in` Eingang, nur auf rechter Seite von Variablen-/Signalzuweisungen, also in Ausdrücken, zulässig.

`out` Ausgang, nur auf linker Seite von Signalzuweisungen zulässig.

`inout` bidirektionale Leitung, kann im Code lesend und schreibend benutzt werden.

`buffer` prinzipiell ein Ausgang, darf intern aber auch gelesen werden. Insbesondere gibt es nur einen Treiber für den Port: eine Prozess, eine konkurrente Anweisung oder eine treibende Instanz.

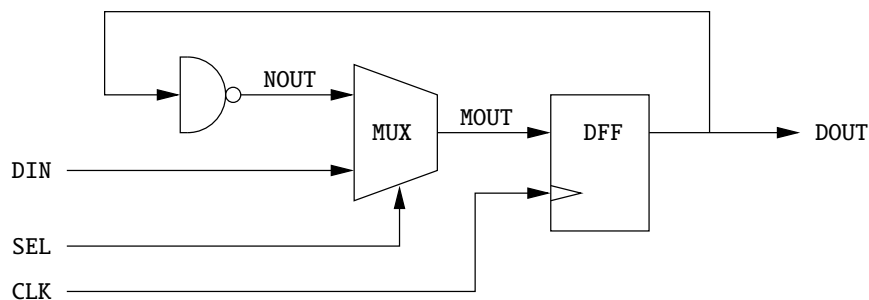
6. Signale

Beispiel

```
package SIGDECL is
  signal VCC: std_logic := '1';
  signal GND: std_logic := '0';
end package SIGDECL;

entity MUXFF is
  port ( DIN : in    bit;
        SEL : in    bit;
        CLK : in    bit;
        DOUT : buffer bit);
end entity MUXFF;

architecture STRUCT of MUXFF is
  signal MOUT : bit;
  signal NOUT : bit;
begin
  ...
end architecture STRUCT;
```



6.2 Signalzuweisungen im Prozess

Die Verbindung von Prozessen mit der *Außenwelt* (andere Prozesse, instanziierte Komponenten, Ports der umgebenden Entity...) geschieht ausschließlich über Signale. Dabei sind beim schreibenden Zugriff, also bei Signalzuweisungen durch den Prozess, einige Punkte zu beachten.

Verzögerungszeiten Das Ziel der einer VHDL-Beschreibung ist letztendlich die Simulation *realer* Schaltungen mit Verzögerungszeiten, entsprechend den Zeitkonstanten der elektrischen Netze. Während auf algorithmischer- oder auf Register-Transfer Ebene noch gar keine, oder nur taktsynchrone Zeitmodelle eine Rolle spielen, beinhalten Modelle auf Gatterebene auch immer die entsprechenden Verzögerungszeiten.

Zur Modellierung werden bei der Signalzuweisung zusätzlich die Zeiten angegeben, so dass sich ein Paar aus Wert und Zeit ergibt. Für die Abarbeitung durch den Simulator heißt das, dass der neue Wert erst nach Ablauf der Verzögerungszeit auf dem Signal (für nachfolgende Eingänge) wirksam wird.

Syntax

```
[⟨label⟩:] ⟨signalObj⟩ <= [⟨delay mode⟩] ⟨wave expression⟩;

⟨delay mode⟩      ::= transport | [reject ⟨time expression⟩] inertial
⟨wave expression⟩ ::= ⟨expression⟩ [after ⟨time expression⟩] { ,
                       ⟨expression⟩ [after ⟨time expression⟩] }
```

Die Verzögerungszeit `after . . .` ist relativ zu der aktuellen Simulationszeit beim Erreichen der Signalzuweisung angegeben. Fehlt die (erste) Angabe `after`, dann entspricht das einer Verzögerungszeit von 0. In einer Signalzuweisung können gleich mehrere Werte, als zeitliche Abfolge, zugewiesen werden, wobei die relativen Zeitangaben `⟨time expression⟩` jeweils größer werden. Diese zeitliche Folge wird dabei vom Simulationsalgorithmus in die Liste zukünftiger Ereignisse aufgenommen (*scheduling*). Der zeitliche Ablauf der Simulation wurde in Abschnitt 1.3 kurz vorgestellt.

Beispiel

```
R <= "1010";
S <= '1' after 4 ns, '0' after 7 ns;
T <= 0, 1 after 1 ns, 3 after 2 ns, 6 after 8 ns;
CLK <= not CLK after PERIOD/2;
```

Optional können auch noch Verzögerungsmodelle für die Signalzuweisung ausgewählt werden: `transport`-Delay ist die 1-zu-1 Wiedergabe einer Wellenform, wie sie auf Leitungen stattfindet, während `inertial`-Delay Pulse einer bestimmten Mindestbreite unterdrückt und damit das Verhalten an (MOS-) Gattereingängen simuliert. Die Voreinstellung ist `Inertial-Delay`, wobei die „erste“ Verzögerungszeit die Mindestpulsbreite angibt; ohnehin wird diese Unterscheidung in der Praxis meist nicht benötigt.

Aktivierung der Zuweisung Obwohl Signalzuweisungen innerhalb eines Prozesses oder einer Prozedur in einer Umgebung stehen, die sequenziell abgearbeitet wird, werden solche Zuweisungen *nicht in der Reihenfolge der sequenziellen Anweisungen wirksam!*

Signalzuweisungen werden erst im folgenden Simulationszyklus wirksam, also bei Erreichen der nächsten `wait`-Anweisung oder, bei Verwendung einer *sensitivity-list*, am Ende des Prozesses. Daraus ergeben sich folgende Konsequenzen für Signalzuweisungen in einem sequenziellen Kontext:

- Wird innerhalb eines Simulationszyklus erst schreibend, dann lesend auf ein Signal zugegriffen, dann liest man den „falschen“ Wert.
- Signale können im Prozess nicht wie Variable als Zwischenspeicher für Werte benutzt werden.
- Um den VHDL-Code übersichtlicher (und fehlerfreier) zu machen, sollte deshalb pro Signal nur eine einzige Zuweisung möglichst „spät“ im Prozesslauf stattfinden.

Soll mit dem Signalwert gerechnet werden, kann man den Wert des Signals (zu Beginn) in einer Variablen zwischenspeichern, mit dieser Variablen arbeiten und (am Ende) den neuen Wert an das Signal zuweisen.

Wegen dieser speziellen Eigenschaften der Signalzuweisung kommt es (gerade bei VHDL-Anfängern) oft zu Fehlern, deshalb noch einige Beispiele:

6. Signale

Beispiel

X <= Y;	beide Anweisungen werden bei dem wait gleichzeitig ausgeführt:
Y <= X;	⇒ die Werte von X und Y werden vertauscht
wait ...	⇒ die Reihenfolge der Zuweisungen ist irrelevant
V := 1;	V wird 1 — sofort
S <= V;	S wird V, also 1 — bei wait
A := S;	A erhält <i>alten</i> Wert von S! — sofort
wait ...	
X <= 1;	Achtung: wird durch zweite Zuweisung unwirksam!
Y <= 3;	Y wird 3 — bei wait
...	
X <= 2;	überschreibt obige Zuweisung: X wird 2 — bei wait
wait ...	

6.3 Implizite Typauflösungen und Bustreiber

Alle bisher beschriebenen Signalzuweisungen gingen von einem Treiber pro Signal aus — ein Treiber heißt, dass dem Signal in genau einer konkurrenten Anweisung (ein Prozess, eine konkurrente Signalzuweisung, ein konkurrenter Prozeduraufruf, eine treibende Instanz) Werte zugewiesen werden. Für die Modellierung von Bussystemen mit mehreren Treibern, ggf. mit zusätzlichen logischen Funktionen (wired-or, wired-and...), sind zusätzliche Mechanismen (*resolution function*) notwendig:

Array-Typ Zu einem Basistyp, der Typ des Signals mit mehreren Treibern, wird ein unbegrenzter Arraytyp deklariert.

Syntax

```
type <array typeId> is array (integer range <>) of <typeId>;
```

Auflösungsfunktion Die *resolution function* wird wie eine normale Funktion deklariert und hat folgende Eigenschaften:

- Die Funktion besitzt genau ein Argument: das Array variabler Länge.
- In der Funktion wird aus diesen (beliebig vielen) Werten ein Rückgabewert vom *ursprünglichen* Typ: <typeId> berechnet.
- Durch die Verbindung der Funktion an Objekte oder Untertypen (a.u.) wird sie bei jeder Signalzuweisung auf diesen Typ automatisch aufgerufen.

Typ und Untertyp Zu dem Basistyp wird ein, mit der Auflösungsfunktion verbundener, Untertyp deklariert.

Syntax

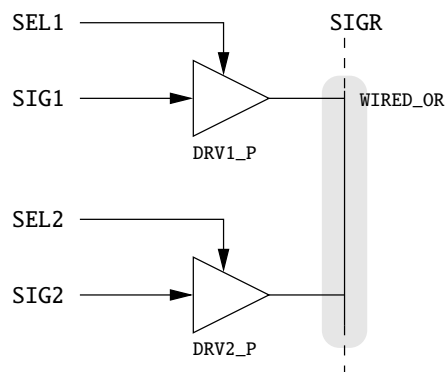
```
subtype <sub typeId> is <functionId> <typeId>;
```

Bei Signalzuweisungen auf ein Signal des aufgelösten Typs wird *implizit* die damit verbundene Funktion aufgerufen. Sie berechnet den effektiven Wert des Signals aus allen Treibern. Analog zu dem subtype kann auch ein aufzulösendes Signal deklariert werden als:

Syntax

```
signal <identifier list> : <sub typeId>;
signal <identifier list> : <functionId> <typeId>;
```

In dem Beispiel wird eine Auflösungsfunktion in Form eines *wired-or* auf einem 4-wertigen Datentyp beschrieben. Zwei Prozesse, die tristate-Treiber modellieren, benutzen ein gemeinsames Ausgangssignal.



Beispiel

```
4-wertiger Typ und entsprechender Array-Typ für Funktion
type FOURVAL is ('X', '0', '1', 'Z');
type FOUR_VECTOR is array (integer range <>) of FOURVAL;
```

Auflösungsfunktion

```
function WIRED_OR (INP: FOUR_VECTOR) return FOURVAL is
  variable RESULT : FOURVAL := '0';
begin
  for I in INP'range loop
    if INP(I) = '1' then return '1';
    elsif INP(I) = 'X' then RESULT := 'X';
    -- else null;
  end if;
end loop;
return RESULT;
end function WIRED_OR;
```

Ergebnis, Bus mit pull-down
für alle Eingänge fertig
INP(I)='Z' oder '0'
also: 'X' oder '0'

Untertyp mit Auflösungsfunktion

```
subtype FOUR_RES is WIRED_OR FOURVAL;
```

6. Signale

```
...
architecture BEHAV of TRISTATE is
  signal SEL1, SEL2 : boolean;
  signal SIG1, SIG2 : FOURVAL;
  signal SIGR      : FOUR_RES;
begin
  ...
  DRV1_P: process (SEL1, SIG1)
  begin
    if SEL1 then SIGR <= SIG1;
    else SIGR <= 'Z';
    end if;
  end process DRV1_P;

  DRV2_P: process (SEL2, SIG2)
  begin
    if SEL2 then SIGR <= SIG2;
    else SIGR <= 'Z';
    end if;
  end process DRV2_P;
```

Selektoren
Eingänge
Ausgangssignal

erste Quelle

zweite Quelle

Mit den Datentypen `std_logic` und `std_logic_vector` können Signale mit mehreren Treibern direkt modelliert werden. Sie sind, mit einer Auflösungsfunktion versehene, Untertypen zu `std_ulogic` / `std_ulogic_vector` (**unresolved**). Die Funktion realisiert aber keine logische Verknüpfung, wie in dem vorherigen Beispiel, sondern behandelt lediglich die unterschiedlichen Treiberstärken. Auch für die beiden Untertypen `signed` und `unsigned` aus dem Package `numeric_std` gilt diese Auflösungsfunktion.

Beispiel

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

entity PROCESSOR is
generic ( BIT_WD : integer range 4 to 128 );
port ( ...
      D_BUS : inout std_logic_vector (BIT_WD-1 downto 0);
      ...);
end entity PROCESSOR;

architecture BEHAV of PROCESSOR is
  ...
  signal D_ENA : std_logic;
  signal D_OUT : std_logic_vector (BIT_WD-1 downto 0);
  signal D_IN  : std_logic_vector (BIT_WD-1 downto 0);
begin
  ...
  D_BUS <= D_OUT when D_ENA = '1' else
           (others => 'Z');
  D_IN  <= D_BUS;
```

Treiber aktiv
-- inaktiv

Bus lesen

Kapitel 7

Konkurrenente Beschreibungen

Um die gleichzeitige Aktivität von Hardwarekomponenten auszudrücken, dienen konkurrenente Beschreibungsformen.

Prozess Die wichtigste konkurrenente Anweisung, der `process`, wurde schon als Umgebung sequenzieller Anweisungen vorgestellt (Abschnitt 5, Seite 30), seine Merkmale sind:

- alle Prozesse sind *parallel* aktiv.
- ein Prozess definiert einen Bereich in dem Anweisungen (programmiersprachen-ähnlich) sequenziell ausgeführt werden, um Verhalten zu beschreiben.
- ein Prozess muss entweder eine sensitivity-list oder explizite `wait`-Anweisungen beinhalten.
- innerhalb des Prozesses werden Ports der `entity` und Signale der `architecture` gelesen und verändert, wodurch der Prozess nach „Außen“ wirkt.

Da ein Prozess in VHDL letztendlich das Verhalten eines Hardwareelementes modellieren soll, das ständig aktiv ist, hat ein Prozess einige spezielle Eigenschaften:

Prozessabarbeitung Ein Prozess entspricht einer Endlosschleife. Bei Beginn der Simulation wird, quasi als Initialisierung, jeder Prozess aktiviert und bis zu einem `wait` ausgeführt. Anschließend wird die Prozessausführung entsprechend der Bedingung der `wait`-Anweisung unterbrochen.

Wird der Prozess später durch Erfüllung der `wait`-Bedingung wieder aktiviert, werden die Anweisungen von dort ausgehend sequenziell weiter ausgeführt bis ein nächstes `wait` erreicht wird. Ist der Prozesscode zu Ende (`end process...`), so *beginnt die Abarbeitung von vorn*. Man kann sich dies vorstellen als:

Beispiel

```
process ...
begin
  loop;                                Start der Schleife
  ...
  wait ...                               mindestens ein wait, bzw. sensitivity-list
  ...
  end loop;                               Ende der Schleife
end process...
```

Aktivierung Wie oben schon erläutert, wird ein Prozess durch den Simulator sequenziell abgearbeitet, dann an einer oder mehreren Stellen unterbrochen und bei Eintreten bestimmter Ereignisse *event* wieder aktiviert.

Daraus ergibt sich, dass ein Prozess mindestens eine *wait*-Anweisung oder eine *sensitivity-list* enthalten muss! Die *sensitivity-list* entspricht einem *wait on ...* am Ende des Prozesses.

Beispiel

```
SUM_P: process (A, B)                                sensitivity-list
begin
  O <= A + B;
end process SUM_P;

ist äquivalent zu:
SUM_P: process
begin
  O <= A + B
  wait on A, B;                                     wait-Anweisung
end process SUM_P;
```

Soll ein Datenfluss beschrieben werden, so entspräche jede Operation einem Prozess der jeweils nur eine einzige Anweisung enthält, wie in dem vorigen Beispiel. Als „Kurzschreibweise“ dienen die konkurrenten Anweisungen. Sie stehen innerhalb der *architecture* und entsprechen jeweils einem eigenen Prozess. Ihre Reihenfolge im VHDL-Code ist irrelevant.

konkurrente Signalzuweisungen sind zu einem Prozess äquivalent, der nur aus einer Signalzuweisung mit entsprechender *sensitivity-list* besteht.

Syntax

```
[<label>:] <signalObj> <= [(<delay mode>)] <wave expression>;

<delay mode> ::= transport | [reject <time expression>] inertial
<wave expression> ::= <expression> [after <time expression>] { ,
                        <expression> [after <time expression>] }
```

Die Syntax entspricht der, zuvor beschriebenen, sequenziellen Form. Die Anweisung wird aktiviert, wenn sich eines der Signale in den Ausdrücken *<expression>* ändert.

Beispiel

```
architecture ...          ist äquivalent zu          architecture ...
...
begin                      ...
  SUM_P: process (A, B)    begin
  begin
    O <= A + B;            O <= A + B;
  end process SUM_P;
...                          ...
```

7. Konkurrente Beschreibungen

bedingte Signalzuweisungen sind zu einem Prozess äquivalent, der nur aus einer if-Anweisung und Signalzuweisungen nach der Verzweigung besteht. Auch geschachtelte if... elsif... sind möglich.

Syntax

```
[⟨label⟩:] ⟨signal⟩ <= [guarded] [⟨delay mode⟩]
    {⟨wave expression⟩ when ⟨condition⟩ else}
    ⟨wave expression⟩ [when ⟨condition⟩] ;
```

Die Anweisung wird aktiviert, wenn sich eines der Signale ändert, das auf der rechten Seite der Zuweisung steht (in Ausdruck oder Bedingung).

Beispiel

```
0 <= A when (X > 10) else
    B when (X < 0) else
    C;
```

selektive Signalzuweisungen entsprechen einem Prozess, der nur aus einer case-Anweisung und Signalzuweisungen nach der Verzweigung besteht.

Syntax

```
[⟨label⟩:] with ⟨expression⟩ select
    ⟨signal⟩ <= [⟨delay mode⟩] ⟨wave expression⟩ when ⟨choices⟩ { ,
    ⟨wave expression⟩ when ⟨choices⟩ } ;
```

⟨choices⟩ ::=	⟨value⟩		genau ein Wert
	⟨value⟩ {^ ⟨value⟩}		Aufzählung
	⟨value⟩ to ⟨value⟩		Bereich
	others		alle übrigen

Die Anweisung wird aktiviert, wenn sich eines der Signale der Auswahlbedingung oder der Ausdrücke auf der rechten Seite der Zuweisung ändert.

Beispiel

```
with SEL select
    MUX <= A when "00",
    B when "01",
    C when "10",
    D when "11";
```

geschützte Signalzuweisungen sind eine besondere Form obiger Signalzuweisungen. Die Zuweisung wird nur dann durchgeführt, wenn ein boole'sches Signal guard den Wert true hat. Dieses Signal kann explizit deklariert und benutzt werden, kommt aber üblicherweise als implizites Signal aus einem *geschützten Block* (siehe Seite 60).

Syntax

```
... ⟨signalObj⟩ <= guarded [⟨delay mode⟩] ⟨wave expression⟩ ...
```

konkurrente Prozeduraufrufe entsprechen einem Prozess der nur diesen Prozeduraufruf beinhaltet. Die Parameter der Prozedur sind in, out und inout Signale. Im Anweisungsteil sind alle sequenziellen Anweisungen zulässig. Die Aktivierung erfolgt durch Wertewechsel der Eingangssignale oder über (ggf. intern vorhandene) wait-Anweisungen.

Syntax

```
[⟨label⟩:] ⟨procedureId⟩ [( [⟨formal⟩ =>] ⟨actual⟩ | open { ,  
[⟨formal⟩ =>] ⟨actual⟩ | open } )] ;
```

Beispiel

```
architecture ...  
  procedure INT2VEC  
    (signal INT : in integer;  
     signal VEC : out bit_vector;  
     signal FLAG: out boolean) is ...  
    ...  
begin  
  INT2VEC (NUMBER, VECTOR, ERR_FLAG);  
  ...
```

ist äquivalent zu:

```
architecture ...  
  procedure INT2VEC ...  
  ...  
begin  
  I2V_P: process (NUMBER)  
  begin  
    INT2VEC (NUMBER, VECTOR, ERR_FLAG);  
  end process I2V_P;  
  ...
```

Assert analog zu der sequenziellen Anweisung, gibt es seit VHDL'93 auch eine konkurrente Form des assert. Sie prüft Bedingungen zur Simulationslaufzeit und darf als *passive Anweisung* im Anweisungsteil von Entities stehen.

Syntax

```
[⟨label⟩:] assert ⟨condition⟩  
[report ⟨string expression⟩]  
[severity failure|error|warning|note];
```

Block / Generate In dem folgenden Kapitel, werden ab Seite 60, noch zwei weitere Anweisungen vorgestellt: block und generate. Mit ihnen kann konkurrent arbeitender VHDL-Code (Prozesse, Anweisungen oder Instanzen) gruppiert werden. Generics der Entity steuern eine bedingte oder wiederholte Ausführung, beispielsweise zur Anpassung von Wortbreiten.

Kapitel 8

Strukturbeschreibungen

Der strukturelle VHDL-Beschreibungsstil ist die textuelle Repräsentation von Netzlisten oder Blockdiagrammen. Dabei werden die Komponenten einer Architektur und deren Verbindungen untereinander beschrieben.

8.1 Hierarchische Beschreibungen

Der eigentliche Aufbau der Hierarchie erfolgt dabei in mehreren Schritten: Zuerst werden Komponenten deklariert und dann Instanzen dieser Komponenten erzeugt, wobei die verbindenden Signale auf die Anschlüsse abgebildet werden. In der Simulation müssen diese Komponenten schließlich an Paare aus einer VHDL-Entity und -Architektur gebunden werden. Die Komponente dient dabei als zusätzliche „Zwischenstufe“ in der Hierarchie.

In VHDL '93 wurde die Syntax der Instanziierung erweitert, so dass direkt Entities und Konfigurationen benutzt werden können. Dadurch entfallen die Schritte der Komponentendeklaration und der Bindung.

Komponentendeklaration innerhalb einer *architecture*, bei den Deklarationen, oder in einem extra *package*. Sie ist meist mit der entsprechenden Entity-Deklaration identisch.

Syntax

```
component <componentId> [is]
  [ <generic declaration> ]
  [ <port declaration> ]
end component [ <componentId> ];

<generic declaration> ::=                               Parameter
  generic ( <generic list> : <typeId> [ := <expression> ] { ;
           <generic list> : <typeId> [ := <expression> ] } );

<port declaration> ::=                                  Ein- und Ausgänge
  port ( <port list> : [ <mode> ] <typeId> [ := <expression> ] { ;
        <port list> : [ <mode> ] <typeId> [ := <expression> ] } );
<mode> ::= in|out|inout|buffer                          „Richtung“
```

Instanziierung der Komponente im Anweisungsteil einer *architecture*. Vom Simulator wie die Instanz wie eine konkurrente Anweisung behandelt.

Syntax

```

<label>: <componentId> [ <generic map> ] [ <port map> ];
                                                    neu in VHDL'93
<label>: [ component ] <componentId>
<label>: entity      [ <libraryId> . ] <entityId> [ ( <archId> ) ] |
<label>: configuration [ <libraryId> . ] <configId>
                [ <generic map> ] [ <port map> ] ;

<generic map> ::=
  generic map ( [ <formal generic> => ] <expression> | open { ;
                [ <formal generic> => ] <expression> | open } )
<port map>    ::=
  port map ( [ <formal port> => ] <signalId> | open { ;
             [ <formal port> => ] <signalId> | open } )

```

Die Abbildung der Parameter und der Signale an den Anschlüssen kann entweder über die Position oder über den Namen erfolgen. Wird an einen der Ports kein Signal angeschlossen (z.B. bei nicht benutzten Ausgängen), so kann der reservierte Bezeichner *open* benutzt werden. Anstelle der Signalnamen ist auch ein Funktionsaufruf möglich, dadurch können Typkonvertierungen direkt bei der Instanziierung von Komponenten vorgenommen werden.

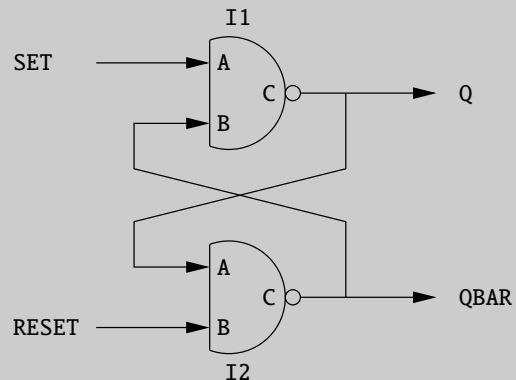
Beispiel

```

entity RSFF is
  port (SET, RESET : in bit;
        Q, QBAR : buffer bit);
end entity RSFF;

architecture NETLIST1 of RSFF is
  component NAND2 is
    port (A, B: in bit; C: out bit);
  end component NAND2;
begin
  I1: NAND2 port map (SET, QBAR, Q);
  I2: NAND2 port map (Q, RESET, QBAR);
end architecture NETLIST1;

```



— Instanziierung mit Signalabbildung über Namen:

```

I1: NAND2 port map (C => Q, A => SET, B => QBAR);
I2: NAND2 port map (A => Q, B => RESET, C => QBAR);

```

8. Strukturbeschreibungen

Bindung der Instanz (Komponente) an ein Paar aus Architektur und Entity.

Syntax

```
for <instance>: <componentId>
  use entity      [ <libraryId> . ] <entityId> [ ( <archId> ) ] [ <mapping> ] ; |
  use configuration [ <libraryId> . ] <configId> [ <mapping> ] ;
  [ for <architectureId>
    { <component configuration> }
  end for; ]
end for;

<instance> ::= <label> { , <label> } | others | all
<mapping>  ::= [ <generic map> ] [ <port map> ]
```

Beispiel

```
entity NAND2 is
  port (A, B      : in    bit;
        C        : out   bit);
end entity NAND2;

architecture NO_DELAY of NAND2 is                               erste architecture
begin
  C <= A nand B;
end architecture NO_DELAY;

architecture GATE_DELAY of NAND2 is                             zweite architecture
begin
  C <= A nand B after 235 ps;
end architecture GATE_DELAY;

architecture NETLIST2 of RSFF is                                 siehe oben RSFF
  component NAND2 is
    port (A, B: in bit; C: out bit);
  end component NAND2;

  for I1: NAND2 use entity work.NAND2(NO_DELAY);
begin
  I1: NAND2 port map (SET, QBAR, Q);                             explizite Bindung
  I2: NAND2 port map (Q, RESET, QBAR);                           implizite Bindung
end architecture NETLIST2;
```

Die Bindung kann schon in der Architektur erfolgen, meist wird aber eine „späte“ Bindung in Form einer externen Konfiguration benutzt — Beispiele dazu folgen in nächsten Abschnitt.

Auch wenn die Instanziierung von Komponenten komplizierter aussieht, hat diese Methode durch die spätere Bindung viele Vorteile und *sollte deshalb immer benutzt werden*. Man erreicht dadurch größtmögliche Flexibilität bei der Zuordnung:

- der Namen $\langle componentId \rangle \leftrightarrow \langle entityId \rangle, \langle archId \rangle,$
- der Bezeichner sowie der Reihenfolge von Generics und Ports
- und sogar deren Anzahl.

8.1.1 Benutzung von Packages

Bei den obigen Beispielen wurden die Komponenten immer innerhalb der Architektur deklariert. Wenn Komponenten häufig benutzt werden, beispielsweise bei Zellbibliotheken, dann kann man sich mit Packages diese Arbeit vereinfachen.

Die Komponentendeklarationen sind in dem Package gesammelt, das mit Hilfe eine use-Klausel in dem jeweiligen Kontext bekannt gegeben wird.

Beispiel

```
package MY_COMPONENTS is
  component NAND2 is
    port (A, B: in bit; C: out bit);
  end component NAND2;

  ... weitere Deklarationen
end package MY_COMPONENTS;

use work.MY_COMPONENTS.all;
architecture NETLIST1 of RSFF is
begin
  I1: NAND2 port map (SET, QBAR, Q);
  I2: NAND2 port map (Q, RESET, QBAR);
end architecture NETLIST1;
```

8.1.2 Konfigurationen

Eigenschaften, die beim Entwurf komplexer Systeme benötigt werden, wie

- Schrittweise top-down Verfeinerung (von black-box Verhalten zu Struktur),
- Untersuchung von Alternativen,
- Unterstützung von Versionen,

werden in VHDL dadurch realisiert, dass zu einer *entity* verschiedene Implementierungen als *architecture* möglich sind. Konfigurationen haben dabei zwei zentrale Aufgaben:

1. Auf oberster Ebene legen sie fest, welche Architektur (von mehreren) einer Entity von Simulations- und Syntheseprogrammen verarbeitet wird.
2. Instanziiert die Architektur Komponenten, dann beschreibt die Konfiguration welche Entities, bzw. Architekturen, den einzelnen Instanzen entsprechen. Diese Zuordnung – auch Bindung genannt – kann beliebig weit in die Hierarchie hineinreichen. Ist sie nicht für alle Komponenten angegeben, so gelten Voreinstellungen.

8. Strukturbeschreibungen

Als Voreinstellung für den Bindungsmechanismus gelten folgende Regeln:

top-level: Alle CAD-Programme, die VHDL verarbeiten, haben einen Schritt der Codeanalyse: wenn die Dateien mit VHDL-Code eingelesen werden. Existieren mehrere alternative Architekturen, dann wird die (zeitlich) zuletzt analysierte für die Entity benutzt. Sie wird auch als *null* Konfiguration bezeichnet.

Komponente - Entity(Architektur): Sind die Namen und die Deklarationen identisch, dann wird die Komponente durch die null-Konfiguration der gleichnamigen Entity ersetzt.

In obigem Beispiel wird deshalb als implizite Konfiguration der Instanz I2: NAND2 das Paar aus NAND2(GATE_DELAY) benutzt;

Syntax

```
configuration <configurationId> of <entityId> is
  for <architectureId>
    { <component configuration> }
  end for;
end [configuration] [ <configurationId> ];

<component configuration> ::=
  for <instance>: <componentId>
    use entity      [ <libraryId> . ] <entityId> [ ( <archId> ) ] [ <mapping> ] ; |
    use configuration [ <libraryId> . ] <configId> [ <mapping> ] ;
  [ for <architectureId>
    { <component configuration> }
  end for; ]
end for;

<instance> ::= <label> { , <label> } | others | all
<mapping>  ::= [ <generic map> ] [ <port map> ]
```

Eine Konfiguration ist eine separate Entwurfseinheit, die die Bindung für einen Teil der Hierarchie beschreibt. Sie ersetzt in übergeordneten Entwürfen in Bindungsanweisungen ein Paar aus Entity und Architektur und sie kann einzeln analysiert und simuliert werden kann.

Bei der VHDL-Simulation hat man meist schon dadurch eine Hierarchie, dass man eine Testumgebung einsetzt, welche die Schaltung als Komponente instanziiert. Über Prozesse oder andere Stimuligeneratoren werden die Eingänge getrieben, die Ausgänge kann man sich interaktiv ansehen oder prozedural auswerten. Viele Simulationsprogramme erwarten eine Konfiguration der Testumgebung — meist wird hier die null-Konfiguration benutzt:

Syntax

```
configuration <configurationId> of <entityId> is
  for <architectureId>
  end for;
end [configuration] [ <configurationId> ];
```

Beispiel

```

entity TEST_RSFF is
end entity TEST_RSFF;

architecture TB of TEST_RSFF is
  component RSFF is
    port (SET, RESET : in    bit;
          Q, QBAR    : buffer bit);
  end component RSFF;

  signal S, R, Q, QB : bit;
begin
  TST_I: RSFF port map (S, R, Q, QBAR);

  STI_P: process
  begin
    S <= '0';    Q <= '1';    wait for 20 ns;
    S <= '1';    Q <= '1';    wait for 20 ns;
    S <= '1';    Q <= '0';    wait for 20 ns;
    S <= '1';    Q <= '1';    wait for 20 ns;
    S <= '0';    Q <= '0';    wait for 20 ns;
    S <= '1';    Q <= '1';    wait for 20 ns;
    S <= '0';    Q <= '1';    wait for 20 ns;
    S <= '1';    Q <= '1';    wait;
  end process STI_P;
end architecture TB;

configuration RSFF_TB0 of TEST_RSFF is                                null-Konfiguration
  for TB
  end for;
end configuration RSFF_TB0

configuration RSFF_TB1 of TEST_RSFF is                                explizite Konfiguration
  for TB
    for TST_I: RSFF use entity work.RSFF(NETLIST1);
      for NETLIST1
        for all: NAND2 use entity work.NAND2(GATE_DELAY);
        end for;
      end for;
    end for;
  end for;
end configuration RSFF_TB1

```

Reihenfolge der Codeanalyse	Entity	Architektur 1	Architektur 2
Beispiele Seite 53, 54	NAND2	NO_DELAY	GATE_DELAY
	RSFF	NETLIST1	NETLIST2
	TEST_RSFF	TB	

Für die Konfigurationen ergibt sich dann die folgende Bindung:

RSFF_TB0 ⇒	TEST_RSFF(TB)	TST_I: RSFF(NETLIST2)	I1: NAND2(NO_DELAY)
			I2: NAND2(GATE_DELAY)
RSFF_TB1 ⇒	TEST_RSFF(TB)	TST_I: RSFF(NETLIST1)	I1: NAND2(GATE_DELAY)
			I2: NAND2(GATE_DELAY)

8. Strukturbeschreibungen

Konfigurationen erlauben weiterhin eine neu-Abbildung der Anschlüsse der Komponenten (`component ...`) zu denen des zugrundeliegenden Entwurfs (`entity ...`). Meist sind Komponenten- und Entity-Deklaration identisch, aber in einigen Fällen sind Umbenennungen oder Umordnungen notwendig.

- Beispielsweise kann mit *generischen* Zellbibliotheken gearbeitet werden, die dann durch Konfigurationen auf Zielbibliotheken verschiedener Hersteller abgebildet werden.
- Außerdem können Elemente des Entwurfs, durch Spezialisierung anderer, ähnlicher Teile ersetzt werden.

In dem Beispiel wird der Inverter in COMPARE (vergl. Seite 7) an einen geeignet beschaltetes Nand-Gatter gebunden.

Beispiel

```
configuration NANDY of COMPARE is
  for ARCH_STRUCT
    for all: INV use entity work.NAND2(NO_DELAY)
      port map (A => X, B => X, C => Z);
    end for;
  end for;
end configuration NANDY;
```

8.1.3 Parametrisierung von Entities durch generische Werte

Während VHDL-Entwürfe über Signale an den Ein- und Ausgängen im Sinne einer Struktur in der Umgebung eingebunden werden, kann deren Verhalten über generische Werte verändert werden. In den zugehörigen Architekturen und in der (den Generics folgenden) Port-Deklaration der Entity können die Werte als Konstanten benutzt werden. Typische Beispiele dazu sind Wortbreiten, Zähler(end)stände oder Verzögerungszeiten.

Beispiel

```
entity ADDER is
  generic ( BIT_WIDTH : integer range 2 to 64 := 16);
  port ( A, B : in signed(BIT_WIDTH-1 downto 0);
        SUM : out signed(BIT_WIDTH downto 0));
end entity ADDER;
...

entity NAND2 is
  generic ( DEL : time := 185 ps);
  port ( A, B : in bit;
        C : out bit);
end entity NAND2;

architecture GEN_DELAY of NAND2 is
begin
  C <= A nand B after DEL;
end architecture GEN_DELAY;
```

Die Übergabe, bzw. die Festlegung konkreter Werte kann an mehreren Stellen stattfinden:

1. default-Wert der entity-Deklaration
2. default-Wert der component-Deklaration, allerdings muss ein generic map von Komponente zu Entity existieren.
3. aktueller Wert bei der Instanziierung der Komponente in der architecture
4. aktueller Wert bei expliziter Bindung, z.B. als configuration

Beispiel

```
architecture NETLIST of RSFF is
  component NAND2 is
    generic (DEL: time); port (A, B: in bit; C: out bit);
  end component NAND2;
begin
  I1: NAND2    port map (SET, QBAR, Q);           Entity-Deklaration 185 ps
  ...

  ...
begin
  I1: NAND2 generic map (DEL => 215 ns)         Wert bei Instanziierung 215 ps
             port map (SET, QBAR, Q);
  ...

  ...
begin
  I1: NAND2    port map (SET, QBAR, Q);
  ...

configuration TIMED_RSFF of RSFF is
  for NETLIST1
    for all: NAND2 use entity work.NAND2(GEN_DELAY)
                       generic map (DEL => 145 ps);           Wert bei Bindung 145 ps
    end for;
  end for;
end configuration TIMED_RSFF;
```

8.2 Strukturierende Anweisungen

Konkurrente Anweisungen (Prozesse, Prozeduraufrufe oder Signalzuweisungen) und Instanziierungen können mit den folgenden Befehlen gruppiert werden.

Block Der `block`-Befehl definiert eine lokale Umgebung, in der neben eigenen Deklarationen (Datentypen, Signale...) sogar eigene Schnittstellen (Ports des Blocks) definiert sein können. In der Praxis werden Blöcke allerdings kaum benutzt — der Aufbau der Hierarchie mit Entities ist flexibler.

Syntax

```

<label>: block [(<guard expression>)] [is]
  [<generic declaration>] [<generic map>];]
  [<port declaration>] [<port map>; ]
  [<local declarations>]
begin
  <statements>
end block [<label>];
    
```

Geschützte Blöcke besitzen zusätzlich den boole'schen Ausdruck *<guard expression>* der ein *implicit* vorhandenes Signal `guard` treibt. Dieses Signal kann innerhalb des Blocks für die Kontrolle von Anweisungen genutzt werden — wegen des direkten Bezugs auf `guard` sind dies meist *geschützte Signalzuweisungen* in ihren verschiedenen Formen.

Syntax

```

... <signalObj> <= guarded [(<delay mode>)] <wave expression> ...
    
```

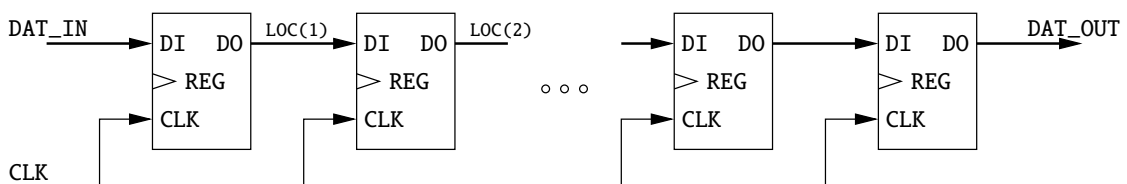
Generate ermöglicht die Gruppierung, bzw. Erzeugung, konkurrenenter Anweisungen oder Instanzen durch Wiederholung – `for`-Schleife.
Auswahl – `if`-Bedingung.

Syntax

```

<label>: for <rangeVar> in <range> generate |                               Laufindex
<label>: if <expression> generate                                         Bedingung
  [<local declarations>]
begin]
  <statements>
end generate [<label>];
    
```

Meist wird die `generate`-Anweisung benutzt, um array-artige Hardwarestrukturen zu beschreiben, wie in dem folgenden Beispiel eines Fifo-Speichers.



Beispiel

```
entity FIFO is
  generic ( DAT_WD   : integer range 2 to 64 := 16;
            FIFO_LEN : integer range 4 to 48 := 8);
  port    ( CLK      : in  std_logic;
            DAT_IN   : in  std_logic_vector (DAT_WD-1 downto 0);
            DAT_OUT  : out std_logic_vector (DAT_WD-1 downto 0));
end entity FIFO;

architecture NETLIST of FIFO is
  component REG is
    generic ( WID   : integer range 2 to 64);
    port    ( CLK   : in  std_logic;
              DI    : in  std_logic_vector (WID-1 downto 0);
              DO    : out std_logic_vector (WID-1 downto 0));
  end component REG;

  subtype SIG_WD_TY is std_logic_vector (DAT_WD-1 downto 0);
  type   SIG_ARR_TY is array (0 to FIFO_LEN) of SIG_WD_TY;
  signal LOC          : SIG_ARR_TY;
begin
  LOC(0) <= DAT_IN;
  DAT_OUT <= LOC(FIFO_LEN);

  GEN1: for I in 1 to FIFO_LEN generate
    GEN1_I: REG generic map (DAT_WD)
      port map (CLK, LOC(I-1), LOC(I));
  end generate GEN1;
end architecture NETLIST;
```

Kapitel 9

Bibliotheken und Packages

9.1 Package

Ein `package` ist, wie `entity`, `architecture` und `component`, eine eigenständige VHDL-Einheit. In Packages werden Deklarationen gesammelt, die an mehreren Stellen im Entwurf benötigt werden, beispielsweise für: Typen und Untertypen, Konstanten, Komponenten, Funktionen und Prozeduren ...

In einigen Fällen werden Packages in „Header“ und „Body“ unterteilt. Der Kopf enthält dabei die nach außen sichtbaren Deklarationen, während die Implementationen in dem „Body“ stehen. Beide Teile können in getrennten Dateien enthalten sein — mit dem Vorteil, dass bei Änderungen möglichst kleine Teile des VHDL-Codes ausgetauscht und neu analysiert werden müssen. Diese Unterteilung ist in zwei Fällen sinnvoll, bei Unterprogrammen sogar notwendig:

zurückgestellte (deferred) Konstante: Die Deklaration der Konstanten befindet sich im Kopf, während die Festlegung eines Wertes im „Body“ stattfindet.

Unterprogramme — Funktionen und Prozeduren: Im Packageheader ist nur die Deklaration des Unterprogramms, der Programmrumpf folgt im „Body“.

Syntax

```
package <packageId> is
  {use <...>      } |
  {type <...>     } | {subtype <...>  } |
  {file <...>    } | {alias <...>   } |
  {function <...>} | {procedure <...>} |
  {component <...>} |
  {constant <...>} |
  {signal <...>  }
end [package] [<packageId>];

package body <packageId> is
  {type <...>     } | {subtype <...>  } |
  {file <...>    } | {alias <...>   } |
  {function <...>} | {procedure <...>} |
  {constant <...>}
end [package body] [<packageId>];
```

Um auf die Deklarationen aus Packages zuzugreifen, werden diese mit der use-Klausel benutzbar gemacht. Diese Anweisung kann im VHDL-Code lokal innerhalb eines Deklarationsteils stehen (mit entsprechendem Gültigkeitsbereich) oder direkt vor der VHDL-Einheit. Befinden sich diese Packages nicht in der Bibliothek work (Voreinstellung), dann muss mit einer library-Anweisung die Bibliothek bekanntgegeben werden.

Syntax

```
library <libraryId>{, <libraryId>};           vor VHDL-Einheit

use [ <libraryId> . ] <packageId> . <objectId>; |
use [ <libraryId> . ] <packageId> . all;
```

Nach obigen Deklarationen kann auf Elemente aus Bibliotheken und Packages direkt über deren Namen *<objectId>* zugegriffen werden. Auch in VHDL gelten, den Programmiersprachen vergleichbare, Regeln zum Sichtbarkeitsbereich von Deklarationen und Objekten. Um bei Überdeckungen gezielt Objekte anzusprechen, kann auch der komplette Bezeichner angegeben werden.

Syntax

```
[ <libraryId> . ] <packageId> . <objectId>
```

Beispiel

```
package MY_DEFS is                               Konstantendeklaration
    constant GATE_DEL : time := 170 ps;
    ...
end package MY_DEFS;

library PROJECT_LIB;                             hier nicht work
use PROJECT_LIB.MY_DEFS.all;

entity NAND2 is
    generic ( DEL           : time := GATE_DEL);
    port    ( A, B         : in bit;
             C             : out bit);
end entity NAND2;

package MY_DEFS is                               oder als „deferred“ constant
    constant GATE_DEL : time;
    ...
end package MY_DEFS;

package body MY_DEFS is
    constant GATE_DEL : time := 170 ps;
end package MY_DEFS;
```

9. Bibliotheken und Packages

```
package TEMPCONV is
    function C2F (C: real) return real;
    function F2C (F: real) return real;
end package TEMPCONV;

package body TEMPCONV is
    function C2F (C: real) return real is
    begin
        return (C * 9.0 / 5.0) + 32.0;
    end function C2F;

    function F2C (F: real) return real is
    ...
end package TEMPCONV;
```

Unterprogramm
nur Deklaration

Funktionsrumpf

9.2 VHDL-Einheiten, Dateien und Bibliotheken

VHDL besitzt vier (fünf) elementare Teile, die als „Einheiten“ der Sprache bezeichnet werden: *entity*, *architecture*, *component* und *package*, bzw. *package body*. Jede dieser Einheiten stellt einen lokalen Kontext bereit. Für die weitere Verarbeitung durch Simulations- oder Syntheseprogramme kann jede Einheit für die Codeanalyse (Compilation) benutzt werden, wobei zeitliche Abhängigkeiten gelten:

erst *entity* dann *architecture* dann *configuration*
package package body

Bei der Codeanalyse die Syntax überprüft, meist wird aber auch eine interne Repräsentation der VHDL-Einheit erzeugt.

Bibliotheken

Die schon mehrfach erwähnten Bibliotheken stellen eigentlich eine zusätzlich Gliederungsstufe zur Gruppierung von VHDL-Einheiten dar. Eine Bibliothek entspricht einer Sammlung von VHDL-Einheiten, wobei diese im Quelltext vorliegen können oder, was meist der Fall ist, schon als kompilierter Code.

Meist werden die VHDL-Bibliotheken auf das Dateisystem eines Rechners abgebildet, wobei die Bibliothek einem Unterverzeichnis entspricht. Diese Zuordnung geschieht außerhalb der Sprache VHDL durch Konfigurationsdateien oder Kommandozeilenparameter, und ist vom verwendeten CAD-Programm abhängig — immer die Dokumentation lesen!

Außer der Standardbibliothek *work*, in sich (i.A.) der aktuelle Entwurf befindet, werden zusätzliche Bibliotheken und Packages im Entwurf benutzt, zur:

Sammlung von IP Neben eigenen Deklarationen, können auch Teilentwürfe oder Komponenten – Stichwort: *intellectual property* – zusammengefasst und so innerhalb einer Arbeitsgruppe oder eines Unternehmens global bereitgestellt werden.

Erweiterung von VHDL Sowohl die IEEE Arbeitsgruppen, die sich mit der VHDL Standardisierung befassen, als auch die Hersteller von VHDL-Werkzeugen, bieten Ergänzungen zum Sprachstandard an. Dies sind zusätzliche Datentypen und Funktionen für:

- mehrwertige Logik: `std_logic_1164`
- Arithmetiken: `numeric_std`, `numeric_bit`
- mathematische Typen und Funktionen: `math_real`, `math_complex`
- Hilfsroutinen: Zufallszahlengeneratoren, Queue-Modellierung...

Benutzung von Zellbibliotheken Die ASIC-Hersteller stellen ihre Zellbibliotheken für die Simulation von Strukturbeschreibungen, bzw. deren Synthese, in Form von VHDL-Libraries zur Verfügung.

VHDL-Dateien

Die Zuordnung von VHDL-Einheiten auf Dateien wurde noch nicht erläutert, da sie oft von den benutzten Programmen abhängt. Prinzipiell hat man die Möglichkeiten von „pro VHDL-Einheit eine Datei“ bis zu „eine einzige Datei für den gesamten Entwurf“. Die Dateinamen sind beliebig wählbar, als Namenserverweiterung hat sich `.vhd` eingebürgert.

Einige Programme haben bestimmte Konventionen, was den Inhalt und die Dateinamen angeht, beispielsweise: immer `entity+architecture`, in Datei `<entityId>.vhd`

Anhang A

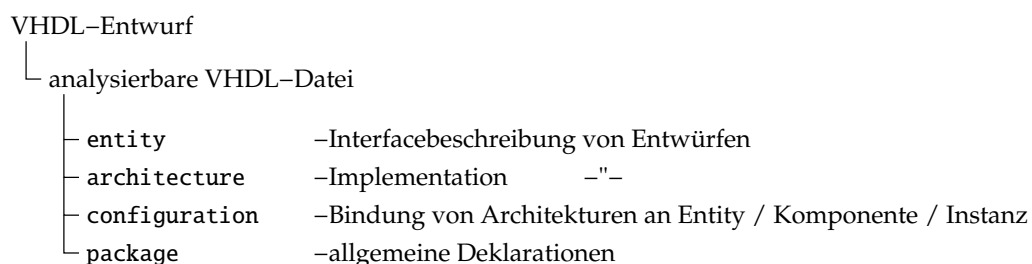
Syntaxbeschreibung

Der Anhang ist keine vollständige, formale Syntaxbeschreibung von VHDL, hier sei auf das „VHDL Language Reference Manual“ des Standards [IEEE 1076 08; IEC 61691-1-1 04; IEEE 1076 02; IEEE 1076 93; IEEE 1076 87] verwiesen. Er ist vielmehr als Nachschlagehilfe bei Fragen zur Syntax und der Anordnung von VHDL-Konstrukten gedacht.

Einige Teile der VHDL-Syntax fehlen auch, so wurden selten benutzte Sprachkonstrukte, die nicht auch zuvor im Text erläutert wurden, weggelassen. Dementsprechend sind nicht alle Produktionen dargestellt – die verbleibenden Bezeichner sollten aber für sich sprechen –, dafür wurde Wert darauf gelegt zu zeigen wo welche VHDL-Anweisungen im Code stehen können.

A.1 Übersicht

VHDL-Entwurf Ein gesamter Entwurf besteht üblicherweise aus einer Anzahl von Dateien, die wiederum die analysierbaren Einheiten enthalten.

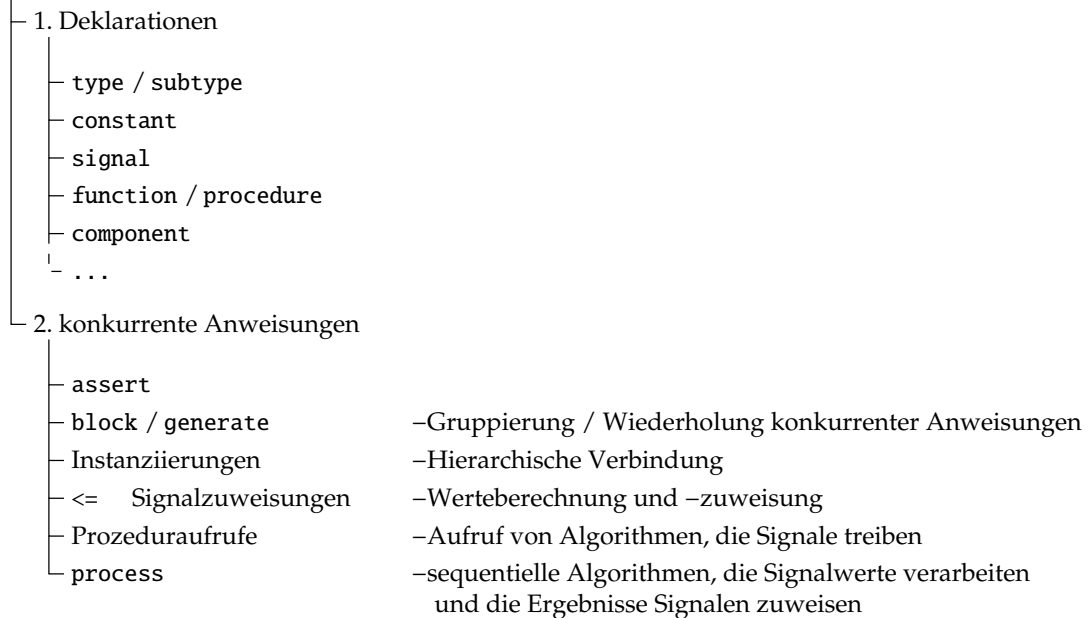


Entities sind die „Teile“ eines zu entwerfenden Systems, wie

- Systemkomponenten, die unter Umständen nichts mit der Hardware zu tun haben (Softwaremodelle, Protokolle, Simulationsumgebungen).
- Hardwaremodelle vorgegebener Teile (Prozessoren, Speicher).
- das zu entwerfende ASIC.
- ein Teil (eine Komponente) eines hierarchischen Entwurfs.
- ein Gatter einer Netzliste.

Architekturen beschreiben die Implementation einer Entity. Sie bestehen aus einem Teil für lokale Deklarationen und *konkurrenten Anweisungen* und *Instanzen von Komponenten*. Diese können in beliebiger Reihenfolge im VHDL-Code stehen.

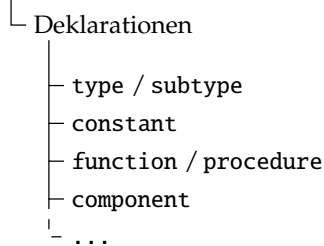
Architektur



Konfigurationen legen für eine Entity (das Interface) fest, welche Architektur (die Implementation) benutzt wird. In Hierarchien werden die einzelnen Instanzen an solche „Paare“ aus Entity und Architektur gebunden.

Packages fassen Deklarationen zusammen, die an mehreren Stellen im Entwurf gebraucht werden (insbesondere in mehreren Dateien).

Package



A. Syntaxbeschreibung

Prozesse und Unterprogramme dienen der Verhaltensbeschreibung durch einen Ablauf *sequenzieller Anweisungen*.

Prozesse verarbeiten die Werte von Signalen und weisen ihnen neue Werte zu. Signalzuweisungen werden erst im folgenden Simulationszyklus und damit *außerhalb der sequenziellen Abarbeitung* wirksam. Die Synchronisation zwischen der Abarbeitung der Anweisungen und den Simulationszyklen (Verlauf an simulierter Zeit) geschieht durch besondere Anweisungen (sensitivity-Liste, wait).

Prozess / Unterprogramm

- 1. Deklarationen
 - type / subtype
 - constant
 - variable
 - function / procedure
 - ...
- 2. sequentielle Anweisungen
 - <= Signalzuweisungen
 - := Variablenzuweisungen
 - if
 - case
 - loop
 - next
 - exit
 - null
 - wait
 - assert
 - report
 - return —nur bei Unterprogrammen—
 - Prozeduraufrufe

A.2 Bibliothekseinheiten

```

<entity declaration>
entity <entityId> is
  [ <generic declaration> ]
  [ <port declaration> ]
  [ <local declarations> ]
  [begin                               normalerweise nicht benutzt
    <passive statements> ]
end [entity] [ <entityId> ];

<generic declaration> ::=                               Parameter
  generic ( <generic list> : <typeId> [ := <expression> ] { ;
           <generic list> : <typeId> [ := <expression> ] } );

<port declaration> ::=                                   Ein- und Ausgänge
  port ( <port list>      : [ <mode> ] <typeId> [ := <expression> ] { ;
        <port list>      : [ <mode> ] <typeId> [ := <expression> ] } );
<mode> ::= in|out|inout|buffer                          „Richtung“

```

```

entity <entityId> is
  ...
begin
  ...
end entity <entityId>;

```

```

architecture <archId> of <entityId> is
  ...
begin
  ...
end architecture <archId>;

```

```

package <packageId> is
  ...
end package <packageId>;

```

```

package body <packageId> is
  ...
end package body <packageId>;

```

```

configuration <configId> of <entityId> is
  ...
end configuration <configId>;

```

```

procedure <procId> (<paraList>) is
  ...
begin
  ...
end procedure <procId>;

```

```

<bLabel>: block (<sigList>) is
  ...
begin
  ...
end block <bLabel>;

```

```

<pLabel>: process (<sigList>) is
  ...
begin
  ...
end process <pLabel>;

```

A. Syntaxbeschreibung

<architecture body>

```
architecture <architectureId> of <entityId> is
  [ <local declarations> ]
begin
  <statements>
end [architecture] [ <architectureId> ];
```

```
<local declarations> ::=
  {use <...> } |
  {function <...> } | {procedure <...> } |
  {type <...> } | {subtype <...> } |
  {file <...> } | {alias <...> } |
  {component <...> } | {<configSpec>} |
  {constant <...> } |
  {signal <...> }
```

```
entity <entityId> is
  ...
begin
  ...
end entity <entityId>;
```

```
architecture <archId> of <entityId> is
  ...
begin
  ...
end architecture <archId>;
```

```
package <packageId> is
  ...
end package <packageId>;
```

```
package body <packageId> is
  ...
end package body <packageId>;
```

```
configuration <configId> of <entityId> is
  ...
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is
  ...
begin
  ...
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is
  ...
begin
  ...
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is
  ...
begin
  ...
end process <pLabel>;
```

<package declaration>

```

package <packageId> is
  {use <...>      } |
  {type <...>     } | {subtype <...>  } |
  {file <...>    } | {alias <...>   } |
  {function <...>} | {procedure <...>} |
  {component <...>} |
  {constant <...>} |
  {signal <...>  }
end [package] [<packageId>];

package body <packageId> is
  {type <...>     } | {subtype <...>  } |
  {file <...>    } | {alias <...>   } |
  {function <...>} | {procedure <...>} |
  {constant <...>}
end [package body] [<packageId>];

```

```

entity <entityId> is
  ...
begin
  ...
end entity <entityId>;

```

```

architecture <archId> of <entityId> is
  ...
begin
  ...
end architecture <archId>;

```

```

package <packageId> is
  ...
end package <packageId>;

```

```

package body <packageId> is
  ...
end package body <packageId>;

```

```

configuration <configId> of <entityId> is
  ...
end configuration <configId>;

```

```

procedure <procId> (<paraList>) is
  ...
begin
  ...
end procedure <procId>;

```

```

<bLabel>: block (<sigList>) is
  ...
begin
  ...
end block <bLabel>;

```

```

<pLabel>: process (<sigList>) is
  ...
begin
  ...
end process <pLabel>;

```

A. Syntaxbeschreibung

<configuration declaration>

```
configuration <configurationId> of <entityId> is
  for <architectureId>
    {<component configuration>}
  end for;
end [configuration] [<configurationId>];

<component configuration> ::=
  for <instance>: <componentId>
    use entity      [<libraryId>.]<entityId>[( <archId>)] [<mapping>] ;|
    use configuration [<libraryId>.]<configId>          [<mapping>] ;
  [ for <architectureId>
    {<component configuration>}
    end for; ]
  end for;

<instance> ::= <label>{, <label>} | others | all
<mapping>  ::= [<generic map>] [<port map>]

<generic map> ::=
  generic map ( [ <formal generic> => ] <expression> | open { ;
               [ <formal generic> => ] <expression> | open } )

<port map>    ::=
  port map ( [ <formal port>      => ] <signalId> | open { ;
             [ <formal port>      => ] <signalId> | open } )
```

```
entity <entityId> is
  ...
begin
  ...
end entity <entityId>;
```

```
architecture <archId> of <entityId> is
  ...
begin
  ...
end architecture <archId>;
```

```
package <packageId> is
  ...
end package <packageId>;
```

```
package body <packageId> is
  ...
end package body <packageId>;
```

```
configuration <configId> of <entityId> is
  ...
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is
  ...
begin
  ...
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is
  ...
begin
  ...
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is
  ...
begin
  ...
end process <pLabel>;
```

A.3 Deklarationen / Spezifikationen

<type declaration>

```
type <typeId> is ( <enumLiteral> {, <enumLiteral> } );
type <typeId> is array ( <index> ) of <element typeId>;
type <typeId> is record
  { <fieldId> : <typeId>; }
end record [ <typeId> ];
type <typeId> is file of <base typeId>;
type <typeId> is access <base typeId>;
```

```
subtype <sub typeId> is <base typeId> [range <range>] ;           Wert begrenzt
subtype <sub typeId> is <base typeId>                             Index begrenzt
  ( <range> | <typeId> {, <range> | <typeId> } );
```

<i><index></i> ::= <i><range></i>		integer Bereich
<i><typeId></i>		Aufzählungstyp
<i><typeId></i> range <i><range></i>		allgemeiner Bereich
<i><typeId></i> range <>		unbegrenzt, Bereichs bei Obj.-Dekl.
<i><range></i> ::= <i><low expr></i> to <i><high expr></i>		
<i><high expr></i> downto <i><low expr></i>		

```
entity <entityId> is
  <type declaration>
begin
  ...
end entity <entityId>;
```

```
architecture <archId> of <entityId> is
  <type declaration>
begin
  ...
end architecture <archId>;
```

```
package <packageId> is
  <type declaration>
end package <packageId>;
```

```
package body <packageId> is
  <type declaration>
end package body <packageId>;
```

```
configuration <configId> of <entityId> is
  ...
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is
  <type declaration>
begin
  ...
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is
  <type declaration>
begin
  ...
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is
  <type declaration>
begin
  ...
end process <pLabel>;
```

A. Syntaxbeschreibung

<alias declaration>

```
alias <aliasId> : <typeId> is <aliasObj>;
```

```
entity <entityId> is  
  <alias declaration>  
begin  
  ...  
end entity <entityId>;
```

```
architecture <archId> of <entityId> is  
  <alias declaration>  
begin  
  ...  
end architecture <archId>;
```

```
package <packageId> is  
  <alias declaration>  
end package <packageId>;
```

```
package body <packageId> is  
  <alias declaration>  
end package body <packageId>;
```

```
configuration <configId> of <entityId> is  
  ...  
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is  
  <alias declaration>  
begin  
  ...  
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is  
  <alias declaration>  
begin  
  ...  
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is  
  <alias declaration>  
begin  
  ...  
end process <pLabel>;
```

<constant declaration>

```
constant <identifier> : <typeId> [<range>] [:= <expression>];
```

```
entity <entityId> is
  <constant declaration>
begin
  ...
end entity <entityId>;
```

```
architecture <archId> of <entityId> is
  <constant declaration>
begin
  ...
end architecture <archId>;
```

```
package <packageId> is
  <constant declaration>
end package <packageId>;
```

```
package body <packageId> is
  <constant declaration>
end package body <packageId>;
```

```
configuration <configId> of <entityId> is
  ...
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is
  <constant declaration>
begin
  ...
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is
  <constant declaration>
begin
  ...
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is
  <constant declaration>
begin
  ...
end process <pLabel>;
```

A. Syntaxbeschreibung

<variable declaration>

```
variable <identifier list> : <typeId> [<range>] [:= <expression>];
```

```
entity <entityId> is  
  ...  
begin  
  ...  
end entity <entityId>;
```

```
architecture <archId> of <entityId> is  
  ...  
begin  
  ...  
end architecture <archId>;
```

```
package <packageId> is  
  ...  
end package <packageId>;
```

```
package body <packageId> is  
  ...  
end package body <packageId>;
```

```
configuration <configId> of <entityId> is  
  ...  
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is  
  <variable declaration>  
begin  
  ...  
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is  
  ...  
begin  
  ...  
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is  
  <variable declaration>  
begin  
  ...  
end process <pLabel>;
```

<signal declaration>

```
signal <identifier list> : <typeId> [<range>] [:= <expression>];
```

```
entity <entityId> is
  <signal declaration>
begin
  ...
end entity <entityId>;
```

```
architecture <archId> of <entityId> is
  <signal declaration>
begin
  ...
end architecture <archId>;
```

```
package <packageId> is
  <signal declaration>
end package <packageId>;
```

```
package body <packageId> is
  ...
end package body <packageId>;
```

```
configuration <configId> of <entityId> is
  ...
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is
  ...
begin
  ...
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is
  <signal declaration>
begin
  ...
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is
  ...
begin
  ...
end process <pLabel>;
```

A. Syntaxbeschreibung

<file declaration>

```
file <identifier> : <typeId> is [in|out] <file string>; VHDL '87
```

```
file <identifier> : <typeId> [[open <mode>] is <file string>]; '93  
<mode> ::= read_mode|write_mode|append_mode
```

```
entity <entityId> is  
  <file declaration>  
begin  
  ...  
end entity <entityId>;
```

```
architecture <archId> of <entityId> is  
  <file declaration>  
begin  
  ...  
end architecture <archId>;
```

```
package <packageId> is  
  <file declaration>  
end package <packageId>;
```

```
package body <packageId> is  
  <file declaration>  
end package body <packageId>;
```

```
configuration <configId> of <entityId> is  
  ...  
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is  
  <file declaration>  
begin  
  ...  
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is  
  <file declaration>  
begin  
  ...  
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is  
  <file declaration>  
begin  
  ...  
end process <pLabel>;
```

<subprogram declaration>

```
function <functionId> [<parameter declaration>] return <typeId>;
```

```
<parameter declaration> ::=
```

```
( [<class>] <formal list> : [<in>] <typeId> [:= <expression>]{ ;  
  [<class>] <formal list> : [<in>] <typeId> [:= <expression>]} )
```

```
<class> ::= constant|signal|file
```

Objektklasse

```
procedure <procedureId> [<parameter declaration>] ;
```

```
<parameter declaration> ::=
```

```
( [<class>] <formal list> : [<mode>] <typeId> [:= <expression>]{ ;  
  [<class>] <formal list> : [<mode>] <typeId> [:= <expression>]} )
```

```
<class> ::= constant|signal|variable|file
```

Objektklasse

```
<mode> ::= in|out|inout
```

Wirkungsrichtung

```
entity <entityId> is  
  <subprogram declaration>  
begin  
  ...  
end entity <entityId>;
```

```
architecture <archId> of <entityId> is  
  <subprogram declaration>  
begin  
  ...  
end architecture <archId>;
```

```
package <packageId> is  
  <subprogram declaration>  
end package <packageId>;
```

```
package body <packageId> is  
  <subprogram declaration>  
end package body <packageId>;
```

```
configuration <configId> of <entityId> is  
  ...  
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is  
  <subprogram declaration>  
begin  
  ...  
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is  
  <subprogram declaration>  
begin  
  ...  
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is  
  <subprogram declaration>  
begin  
  ...  
end process <pLabel>;
```

A. Syntaxbeschreibung

<subprogram body>

```
function <functionId> [<parameter declaration>] return <typeId> is
  [<local declarations>]
begin
  <sequential statements>                                mit return
end [function] [<functionId>];

<parameter declaration> ::=
  ( [<class>] <formal list> : [in] <typeId> [:= <expression>] { ;
    [<class>] <formal list> : [in] <typeId> [:= <expression>] } )
<class> ::= constant|signal|file                            Objektklasse

procedure <procedureId> [<parameter declaration>] is
  [<local declarations>]
begin
  <sequential statements>
end [procedure] [<procedureId>];

<parameter declaration> ::=
  ( [<class>] <formal list> : [<mode>] <typeId> [:= <expression>] { ;
    [<class>] <formal list> : [<mode>] <typeId> [:= <expression>] } )

<class> ::= constant|signal|variable|file                    Objektklasse
<mode> ::= in|out|inout                                    Wirkungsrichtung
```

```
entity <entityId> is
  <subprogram body>
begin
  ...
end entity <entityId>;
```

```
architecture <archId> of <entityId> is
  <subprogram body>
begin
  ...
end architecture <archId>;
```

```
package <packageId> is
  ...
end package <packageId>;
```

```
package body <packageId> is
  <subprogram body>
end package body <packageId>;
```

```
configuration <configId> of <entityId> is
  ...
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is
  <subprogram body>
begin
  ...
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is
  <subprogram body>
begin
  ...
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is
  <subprogram body>
begin
  ...
end process <pLabel>;
```

<attribute declaration>

```
attribute <attributeId> : <typeId>;
```

```
entity <entityId> is
  <attribute declaration>
begin
  ...
end entity <entityId>;
```

```
architecture <archId> of <entityId> is
  <attribute declaration>
begin
  ...
end architecture <archId>;
```

```
package <packageId> is
  <attribute declaration>
end package <packageId>;
```

```
package body <packageId> is
  ...
end package body <packageId>;
```

```
configuration <configId> of <entityId> is
  ...
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is
  <attribute declaration>
begin
  ...
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is
  <attribute declaration>
begin
  ...
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is
  <attribute declaration>
begin
  ...
end process <pLabel>;
```

A. Syntaxbeschreibung

<attribute specification>

```
attribute <attributeId> of <object list> : <object class> is <expression>;
```

```
entity <entityId> is  
  <attribute specification>  
begin  
  ...  
end entity <entityId>;
```

```
architecture <archId> of <entityId> is  
  <attribute specification>  
begin  
  ...  
end architecture <archId>;
```

```
package <packageId> is  
  <attribute specification>  
end package <packageId>;
```

```
package body <packageId> is  
  <attribute specification>  
end package body <packageId>;
```

```
configuration <configId> of <entityId> is  
  ...  
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is  
  <attribute specification>  
begin  
  ...  
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is  
  <attribute specification>  
begin  
  ...  
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is  
  <attribute specification>  
begin  
  ...  
end process <pLabel>;
```

<component declaration>

```

component <componentId> [is]
  [ <generic declaration> ]
  [ <port declaration> ]
end component [ <componentId> ];

<generic declaration> ::=                               Parameter
  generic ( <generic list> : <typeId> [ := <expression> ] { ;
           <generic list> : <typeId> [ := <expression> ] } );

<port declaration> ::=                                  Ein- und Ausgänge
  port ( <port list>      : [ <mode> ] <typeId> [ := <expression> ] { ;
        <port list>      : [ <mode> ] <typeId> [ := <expression> ] } );
<mode> ::= in|out|inout|buffer                          „Richtung“
  
```

```

entity <entityId> is
  ...
begin
  ...
end entity <entityId>;
  
```

```

architecture <archId> of <entityId> is
  <component declaration>
begin
  ...
end architecture <archId>;
  
```

```

package <packageId> is
  <component declaration>
end package <packageId>;
  
```

```

package body <packageId> is
  ...
end package body <packageId>;
  
```

```

configuration <configId> of <entityId> is
  ...
end configuration <configId>;
  
```

```

procedure <procId> (<paraList>) is
  ...
begin
  ...
end procedure <procId>;
  
```

```

<bLabel>: block (<sigList>) is
  <component declaration>
begin
  ...
end block <bLabel>;
  
```

```

<pLabel>: process (<sigList>) is
  ...
begin
  ...
end process <pLabel>;
  
```

A. Syntaxbeschreibung

<configuration specification>

```
for <instance>: <componentId>
  use entity      [ <libraryId> . ] <entityId> [ ( <archId> ) ] [ <mapping> ] ; |
  use configuration [ <libraryId> . ] <configId> [ <mapping> ] ;
  [ for <architectureId>
    { <component configuration> }
  end for; ]
end for;
```

<instance> ::= *<label>*{, *<label>*} | *others* | *all*

<mapping> ::= [*<generic map>*] [*<port map>*]

<generic map> ::=

```
generic map ( [ <formal generic> => ] <expression> | open { ;
              [ <formal generic> => ] <expression> | open } )
```

<port map> ::=

```
port map ( [ <formal port> => ] <signalId> | open { ;
            [ <formal port> => ] <signalId> | open } )
```

```
entity <entityId> is
  ...
begin
  ...
end entity <entityId>;
```

```
architecture <archId> of <entityId> is
  <configuration specification>
begin
  ...
end architecture <archId>;
```

```
package <packageId> is
  ...
end package <packageId>;
```

```
package body <packageId> is
  ...
end package body <packageId>;
```

```
configuration <configId> of <entityId> is
  ...
end configuration <configId>;
```

```
procedure <procId> ( <paraList> ) is
  ...
begin
  ...
end procedure <procId>;
```

```
<bLabel>: block ( <sigList> ) is
  <configuration specification>
begin
  ...
end block <bLabel>;
```

```
<pLabel>: process ( <sigList> ) is
  ...
begin
  ...
end process <pLabel>;
```

<library clause>

```
library <libraryId>{,<libraryId>;
```

```
<library clause>  
entity <entityId> is  
  ...  
begin  
  ...  
end entity <entityId>;
```

```
<library clause>  
architecture <archId> of <entityId> is  
  ...  
begin  
  ...  
end architecture <archId>;
```

```
<library clause>  
package <packageId> is  
  ...  
end package <packageId>;
```

```
<library clause>  
package body <packageId> is  
  ...  
end package body <packageId>;
```

```
<library clause>  
configuration <configId> of <entityId> is  
  ...  
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is  
  ...  
begin  
  ...  
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is  
  ...  
begin  
  ...  
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is  
  ...  
begin  
  ...  
end process <pLabel>;
```

A. Syntaxbeschreibung

<use clause>

```
use [<libraryId>.]<packageId>.<objectId>; |  
use [<libraryId>.]<packageId>.all;
```

```
<use clause>  
entity <entityId> is  
  <use clause>  
begin  
  ...  
end entity <entityId>;
```

```
<use clause>  
architecture <archId> of <entityId> is  
  <use clause>  
begin  
  ...  
end architecture <archId>;
```

```
<use clause>  
package <packageId> is  
  <use clause>  
end package <packageId>;
```

```
<use clause>  
package body <packageId> is  
  <use clause>  
end package body <packageId>;
```

```
<use clause>  
configuration <configId> of <entityId> is  
  <use clause>  
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is  
  <use clause>  
begin  
  ...  
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is  
  <use clause>  
begin  
  ...  
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is  
  <use clause>  
begin  
  ...  
end process <pLabel>;
```

A.4 sequenzielle Anweisungen

```

<wait statement>
[<label>:] wait
  [on <signalObj>{, <signalObj>}]
  [until <condition>]
  [for <time expression>];

```

```

entity <entityId> is
  ...
begin
  ...
end entity <entityId>;

```

```

architecture <archId> of <entityId> is
  ...
begin
  ...
end architecture <archId>;

```

```

package <packageId> is
  ...
end package <packageId>;

```

```

package body <packageId> is
  ...
end package body <packageId>;

```

```

configuration <configId> of <entityId> is
  ...
end configuration <configId>;

```

```

procedure <procId> (<paraList>) is
  ...
begin
  <wait statement>
end procedure <procId>;

```

```

<bLabel>: block (<sigList>) is
  ...
begin
  ...
end block <bLabel>;

```

```

<pLabel>: process (<sigList>) is
  ...
begin
  <wait statement>
end process <pLabel>;

```

A. Syntaxbeschreibung

<assertion statement>

```
[<label>:] assert <condition>
  [report <string expression>]
  [severity failure|error|warning|note];
```

```
entity <entityId> is
  ...
begin
  ...
end entity <entityId>;
```

```
package <packageId> is
  ...
end package <packageId>;
```

```
configuration <configId> of <entityId> is
  ...
end configuration <configId>;
```

```
<bLabel>: block (<sigList>) is
  ...
begin
  ...
end block <bLabel>;
```

```
architecture <archId> of <entityId> is
  ...
begin
  ...
end architecture <archId>;
```

```
package body <packageId> is
  ...
end package body <packageId>;
```

```
procedure <procId> (<paraList>) is
  ...
begin
  <assertion statement>
end procedure <procId>;
```

```
<pLabel>: process (<sigList>) is
  ...
begin
  <assertion statement>
end process <pLabel>;
```

<report statement>

```
[<label>:] report <string expression>
  [severity failure|error|warning|note];
```

```
entity <entityId> is
  ...
begin
  ...
end entity <entityId>;
```

```
architecture <archId> of <entityId> is
  ...
begin
  ...
end architecture <archId>;
```

```
package <packageId> is
  ...
end package <packageId>;
```

```
package body <packageId> is
  ...
end package body <packageId>;
```

```
configuration <configId> of <entityId> is
  ...
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is
  ...
begin
  <report statement>
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is
  ...
begin
  ...
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is
  ...
begin
  <report statement>
end process <pLabel>;
```

A. Syntaxbeschreibung

<signal assignment>

```
[<label>:] <signalObj> <= [ <delay mode> ] <wave expression>;
```

```
entity <entityId> is  
  ...  
begin  
  ...  
end entity <entityId>;
```

```
architecture <archId> of <entityId> is  
  ...  
begin  
  ...  
end architecture <archId>;
```

```
package <packageId> is  
  ...  
end package <packageId>;
```

```
package body <packageId> is  
  ...  
end package body <packageId>;
```

```
configuration <configId> of <entityId> is  
  ...  
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is  
  ...  
begin  
  <signal assignment>  
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is  
  ...  
begin  
  ...  
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is  
  ...  
begin  
  <signal assignment>  
end process <pLabel>;
```

<variable assignment>

```
[<label>:] <variableObj> := <expression>;
```

```
entity <entityId> is
  ...
begin
  ...
end entity <entityId>;
```

```
architecture <archId> of <entityId> is
  ...
begin
  ...
end architecture <archId>;
```

```
package <packageId> is
  ...
end package <packageId>;
```

```
package body <packageId> is
  ...
end package body <packageId>;
```

```
configuration <configId> of <entityId> is
  ...
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is
  ...
begin
  <variable assignment>
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is
  ...
begin
  ...
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is
  ...
begin
  <variable assignment>
end process <pLabel>;
```

A. Syntaxbeschreibung

<procedure call>

```
[<label>:] <procedureId> [( [<formal> =>] <actual> | open { ,  
[<formal> =>] <actual> | open } )] ;
```

```
entity <entityId> is  
  ...  
begin  
  ...  
end entity <entityId>;
```

```
architecture <archId> of <entityId> is  
  ...  
begin  
  ...  
end architecture <archId>;
```

```
package <packageId> is  
  ...  
end package <packageId>;
```

```
package body <packageId> is  
  ...  
end package body <packageId>;
```

```
configuration <configId> of <entityId> is  
  ...  
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is  
  ...  
begin  
  <procedure call>  
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is  
  ...  
begin  
  ...  
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is  
  ...  
begin  
  <procedure call>  
end process <pLabel>;
```

<if statement>

```
[<label>:] if <condition> then
  <sequential statements>
{elsif <condition> then
  <sequential statements>}
[else
  <sequential statements>]
end if [<label>];
```

```
entity <entityId> is
  ...
begin
  ...
end entity <entityId>;
```

```
architecture <archId> of <entityId> is
  ...
begin
  ...
end architecture <archId>;
```

```
package <packageId> is
  ...
end package <packageId>;
```

```
package body <packageId> is
  ...
end package body <packageId>;
```

```
configuration <configId> of <entityId> is
  ...
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is
  ...
begin
  <i><if statement></i>
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is
  ...
begin
  ...
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is
  ...
begin
  <i><if statement></i>
end process <pLabel>;
```

A. Syntaxbeschreibung

<case statement>

```
[<label>:] case <expression> is
  {when <choices> => <sequential statements>}
end case [<label>];
```

<i><choices></i> ::=	<i><value></i>		genau ein Wert
	<i><value></i> { ^ <i><value></i> }		Aufzählung
	<i><value></i> to <i><value></i>		Bereich
	others		alle übrigen

```
entity <entityId> is
  ...
begin
  ...
end entity <entityId>;
```

```
architecture <archId> of <entityId> is
  ...
begin
  ...
end architecture <archId>;
```

```
package <packageId> is
  ...
end package <packageId>;
```

```
package body <packageId> is
  ...
end package body <packageId>;
```

```
configuration <configId> of <entityId> is
  ...
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is
  ...
begin
  <case statement>
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is
  ...
begin
  ...
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is
  ...
begin
  <case statement>
end process <pLabel>;
```

<loop statement>

<pre>[<label>:] while <expression> loop for <rangeVar> in <range> loop loop <sequential statements> end loop end loop [<label>];</pre>	<table border="0"> <tr> <td style="border-left: 1px solid black; padding-left: 5px;"> </td> <td>boole'sche Bedingung</td> </tr> <tr> <td style="border-left: 1px solid black; padding-left: 5px;"> </td> <td>Schleifenvariable, Wertebereich</td> </tr> <tr> <td style="border-left: 1px solid black; padding-left: 5px;"> </td> <td>beliebig oft</td> </tr> </table>		boole'sche Bedingung		Schleifenvariable, Wertebereich		beliebig oft
	boole'sche Bedingung						
	Schleifenvariable, Wertebereich						
	beliebig oft						

```
entity <entityId> is
    ...
begin
    ...
end entity <entityId>;
```

```
architecture <archId> of <entityId> is
    ...
begin
    ...
end architecture <archId>;
```

```
package <packageId> is
    ...
end package <packageId>;
```

```
package body <packageId> is
    ...
end package body <packageId>;
```

```
configuration <configId> of <entityId> is
    ...
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is
    ...
begin
    <loop statement>
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is
    ...
begin
    ...
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is
    ...
begin
    <loop statement>
end process <pLabel>;
```

A. Syntaxbeschreibung

<next statement>

```
[<label>:] next [<loop label>] [when <condition>];
```

```
entity <entityId> is
  ...
begin
  ...
end entity <entityId>;
```

```
architecture <archId> of <entityId> is
  ...
begin
  ...
end architecture <archId>;
```

```
package <packageId> is
  ...
end package <packageId>;
```

```
package body <packageId> is
  ...
end package body <packageId>;
```

```
configuration <configId> of <entityId> is
  ...
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is
  ...
begin
  <next statement>
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is
  ...
begin
  ...
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is
  ...
begin
  <next statement>
end process <pLabel>;
```

<exit statement>

```
[<label>:] exit [<loop label>] [when <condition>];
```

```
entity <entityId> is  
  ...  
begin  
  ...  
end entity <entityId>;
```

```
architecture <archId> of <entityId> is  
  ...  
begin  
  ...  
end architecture <archId>;
```

```
package <packageId> is  
  ...  
end package <packageId>;
```

```
package body <packageId> is  
  ...  
end package body <packageId>;
```

```
configuration <configId> of <entityId> is  
  ...  
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is  
  ...  
begin  
  <exit statement>  
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is  
  ...  
begin  
  ...  
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is  
  ...  
begin  
  <exit statement>  
end process <pLabel>;
```

A. Syntaxbeschreibung

<null statement>

```
[<label>:] null;
```

```
entity <entityId> is
  ...
begin
  ...
end entity <entityId>;
```

```
architecture <archId> of <entityId> is
  ...
begin
  ...
end architecture <archId>;
```

```
package <packageId> is
  ...
end package <packageId>;
```

```
package body <packageId> is
  ...
end package body <packageId>;
```

```
configuration <configId> of <entityId> is
  ...
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is
  ...
begin
  <null statement>
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is
  ...
begin
  ...
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is
  ...
begin
  <null statement>
end process <pLabel>;
```

```
<return statement>  
[<label>:] return [<expression>];
```

```
entity <entityId> is  
  ...  
begin  
  ...  
end entity <entityId>;
```

```
architecture <archId> of <entityId> is  
  ...  
begin  
  ...  
end architecture <archId>;
```

```
package <packageId> is  
  ...  
end package <packageId>;
```

```
package body <packageId> is  
  ...  
end package body <packageId>;
```

```
configuration <configId> of <entityId> is  
  ...  
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is  
  ...  
begin  
  <return statement>  
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is  
  ...  
begin  
  ...  
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is  
  ...  
begin  
  ...  
end process <pLabel>;
```

A.5 konkurrente Anweisungen

```
<process statement>
[<label>:] process [(<sensitivity list>)] [is]
  [<local declarations>]
begin
  <sequential statements>
end process [<label>];

<local declarations> ::=
{type <...>      } | {subtype <...>  } |
{file <...>     } | {alias <...>    } |
{function <...> } | {procedure <...> } |
{constant <...>} |
{variable <...>}
```

```
entity <entityId> is
  ...
begin
  <process statement>
end entity <entityId>;
```

```
architecture <archId> of <entityId> is
  ...
begin
  <process statement>
end architecture <archId>;
```

```
package <packageId> is
  ...
end package <packageId>;
```

```
package body <packageId> is
  ...
end package body <packageId>;
```

```
configuration <configId> of <entityId> is
  ...
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is
  ...
begin
  ...
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is
  ...
begin
  <process statement>
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is
  ...
begin
  ...
end process <pLabel>;
```

<procedure call>

```
[<label>:] <procedureId> [( [<formal> =>] <actual> | open { ,
                             [<formal> =>] <actual> | open } )] ;
```

```
entity <entityId> is
  ...
begin
  <procedure call>
end entity <entityId>;
```

```
architecture <archId> of <entityId> is
  ...
begin
  <procedure call>
end architecture <archId>;
```

```
package <packageId> is
  ...
end package <packageId>;
```

```
package body <packageId> is
  ...
end package body <packageId>;
```

```
configuration <configId> of <entityId> is
  ...
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is
  ...
begin
  ...
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is
  ...
begin
  <procedure call>
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is
  ...
begin
  ...
end process <pLabel>;
```

A. Syntaxbeschreibung

<assertion statement>

```
[<label>:] assert <condition>
  [report <string expression>]
  [severity failure|error|warning|note];
```

```
entity <entityId> is
  ...
begin
  <assertion statement>
end entity <entityId>;
```

```
architecture <archId> of <entityId> is
  ...
begin
  <assertion statement>
end architecture <archId>;
```

```
package <packageId> is
  ...
end package <packageId>;
```

```
package body <packageId> is
  ...
end package body <packageId>;
```

```
configuration <configId> of <entityId> is
  ...
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is
  ...
begin
  ...
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is
  ...
begin
  <assertion statement>
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is
  ...
begin
  ...
end process <pLabel>;
```

<signal assignment>

```
[<label>:] <signalObj> <= [ <delay mode> ] <wave expression>;

[<label>:] <signal> <= [guarded] [ <delay mode> ]
    { <wave expression> when <condition> else }
    <wave expression> [when <condition>] ;

[<label>:] with <expression> select
    <signal> <= [ <delay mode> ] <wave expression> when <choices> { ,
        <wave expression> when <choices> } ;

... <signalObj> <= guarded [ <delay mode> ] <wave expression> ...

<choices> ::= <value> |                                genau ein Wert
             <value> { ^ <value> } |                  Aufzählung
             <value> to <value> |                     Bereich
             others |                                  alle übrigen

<delay mode> ::= transport | [reject <time expression>] inertial
<wave expression> ::= <expression> [after <time expression>] { ,
                     <expression> [after <time expression>] }
```

```
entity <entityId> is
...
begin
...
end entity <entityId>;
```

```
architecture <archId> of <entityId> is
...
begin
    <signal assignment>
end architecture <archId>;
```

```
package <packageId> is
...
end package <packageId>;
```

```
package body <packageId> is
...
end package body <packageId>;
```

```
configuration <configId> of <entityId> is
...
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is
...
begin
...
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is
...
begin
    <signal assignment>
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is
...
begin
...
end process <pLabel>;
```

A. Syntaxbeschreibung

```

<block statement>
<label>: block [( <guard expression> )] [is]
  [ <generic declaration> [ <generic map> ; ] ]
  [ <port declaration> [ <port map> ; ] ]
  [ <local declarations> ]
begin
  <statements>
end block [ <label> ];

<generic declaration> ::=                                     Parameter
  generic ( <generic list> : <typeId> [ := <expression> ] { ;
           <generic list> : <typeId> [ := <expression> ] } );

<port declaration> ::=                                       Ein- und Ausgänge
  port ( <port list>      : [ <mode> ] <typeId> [ := <expression> ] { ;
        <port list>      : [ <mode> ] <typeId> [ := <expression> ] } );
<mode> ::= in | out | inout | buffer                          „Richtung“

<generic map> ::=
  generic map ( [ <formal generic> => ] <expression> | open { ;
               [ <formal generic> => ] <expression> | open } )

<port map> ::=
  port map ( [ <formal port>      => ] <signalId> | open { ;
             [ <formal port>      => ] <signalId> | open } )

```

```

entity <entityId> is
  ...
begin
  ...
end entity <entityId>;

```

```

architecture <archId> of <entityId> is
  ...
begin
  <block statement>
end architecture <archId>;

```

```

package <packageId> is
  ...
end package <packageId>;

```

```

package body <packageId> is
  ...
end package body <packageId>;

```

```

configuration <configId> of <entityId> is
  ...
end configuration <configId>;

```

```

procedure <procId> (<paraList>) is
  ...
begin
  ...
end procedure <procId>;

```

```

<bLabel>: block (<sigList>) is
  ...
begin
  <block statement>
end block <bLabel>;

```

```

<pLabel>: process (<sigList>) is
  ...
begin
  ...
end process <pLabel>;

```

<generate statement>

<i><label></i> : for <i><rangeVar></i> in <i><range></i> generate	Laufindex
<i><label></i> : if <i><expression></i> generate	Bedingung
[<i><local declarations></i>	
begin]	
<i><statements></i>	
end generate [<i><label></i>];	

```
entity <entityId> is
  ...
begin
  ...
end entity <entityId>;
```

```
architecture <archId> of <entityId> is
  ...
begin
  <generate statement>
end architecture <archId>;
```

```
package <packageId> is
  ...
end package <packageId>;
```

```
package body <packageId> is
  ...
end package body <packageId>;
```

```
configuration <configId> of <entityId> is
  ...
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is
  ...
begin
  ...
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is
  ...
begin
  <generate statement>
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is
  ...
begin
  ...
end process <pLabel>;
```

A. Syntaxbeschreibung

<component instantiation>

```
<label>: <componentId> [<generic map>] [<port map>];
```

neu in VHDL'93

```
<label>: [component] <componentId> |  
<label>: entity [<libraryId>.] <entityId> [(<archId>)] |  
<label>: configuration [<libraryId>.] <configId>  
      [<generic map>] [<port map>] ;
```

```
<generic map> ::=  
  generic map ([<formal generic> =>] <expression>) | open{ ;  
                [<formal generic> =>] <expression>) | open{ )
```

```
<port map> ::=  
  port map ([<formal port> =>] <signalId>) | open{ ;  
            [<formal port> =>] <signalId>) | open{ )
```

```
entity <entityId> is  
  ...  
begin  
  ...  
end entity <entityId>;
```

```
architecture <archId> of <entityId> is  
  ...  
begin  
  <component instantiation>  
end architecture <archId>;
```

```
package <packageId> is  
  ...  
end package <packageId>;
```

```
package body <packageId> is  
  ...  
end package body <packageId>;
```

```
configuration <configId> of <entityId> is  
  ...  
end configuration <configId>;
```

```
procedure <procId> (<paraList>) is  
  ...  
begin  
  ...  
end procedure <procId>;
```

```
<bLabel>: block (<sigList>) is  
  ...  
begin  
  <component instantiation>  
end block <bLabel>;
```

```
<pLabel>: process (<sigList>) is  
  ...  
begin  
  ...  
end process <pLabel>;
```

A.6 Sprachstandard

Operatoren

1. logische Operatoren			
and	$a \wedge b$	Typ- <i>a</i> bit bit_vector boolean	Typ- <i>b</i> = <i>a</i>
or	$a \vee b$	bit bit_vector boolean	= <i>a</i>
nand	$\neg(a \wedge b)$	bit bit_vector boolean	= <i>a</i>
nor	$\neg(a \vee b)$	bit bit_vector boolean	= <i>a</i>
xor	$\neg(a \equiv b)$	bit bit_vector boolean	= <i>a</i>
xnor	$a \equiv b$	bit bit_vector boolean	= <i>a</i>
2. relationale Operatoren			
=	$a = b$	Typ- <i>a</i> beliebiger Typ	Typ-⟨ <i>op</i> ⟩ boolean
/=	$a \neq b$	beliebiger Typ	= <i>a</i> boolean
<	$a < b$	skalarer Typ 1-dim. Array	= <i>a</i> boolean
<=	$a \leq b$	skalarer Typ 1-dim. Array	= <i>a</i> boolean
>	$a > b$	skalarer Typ 1-dim. Array	= <i>a</i> boolean
>=	$a \geq b$	skalarer Typ 1-dim. Array	= <i>a</i> boolean
3. schiebende Operatoren			
sll	$(a_{n-1-b} \dots a_0, 0_{b \dots 1})$	Typ- <i>a</i> bit_vector bit/bool-Array	Typ- <i>b</i> integer
srl	$(0_{1 \dots b}, a_{n-1} \dots a_b)$	bit_vector bit/bool-Array	integer
sla	$(a_{n-1-b} \dots a_0, a_{0,b \dots 1})$	bit_vector bit/bool-Array	integer
sra	$(a_{n-1,1 \dots b}, a_{n-1} \dots a_b)$	bit_vector bit/bool-Array	integer
rol	$(a_{n-1-b} \dots a_0, a_{n-1} \dots a_{n-b})$	bit_vector bit/bool-Array	integer
ror	$(a_{b-1} \dots a_0, a_{n-1} \dots a_b)$	bit_vector bit/bool-Array	integer
4. additive Operatoren			
+	$a + b$	Typ- <i>a</i> integer real phys. Typ	Typ- <i>b</i> = <i>a</i>
-	$a - b$	integer real phys. Typ	= <i>a</i>
&	$(a_n \dots a_0, b_m \dots b_0)$	skalarer Typ 1-dim. Array	Typ-⟨ <i>op</i> ⟩ <i>a</i> -Skalar / Array <i>a</i> -Array
5. vorzeichen Operatoren			
+	$+a$	Typ- <i>a</i> integer real phys. Typ	Typ- <i>b</i> = <i>a</i>
-	$-a$	integer real phys. Typ	= <i>a</i>
6. multiplikative Operatoren			
*	$a * b$	Typ- <i>a</i> integer real phys. Typ	Typ- <i>b</i> = <i>a</i>
/	a / b	integer real phys. Typ	= <i>a</i>
mod	Modulus	integer	= <i>a</i>
rem	Teilerrest	integer	= <i>a</i>
7. sonstige Operatoren			
**	a^b	Typ- <i>a</i> integer real	Typ- <i>b</i> integer
abs	$ a $	integer real phys. Typ	= <i>a</i>
not	$\neg a$	bit bit_vector boolean	= <i>a</i>

A. Syntaxbeschreibung

Unterprogramme

<code>integer</code>	<code>(⟨integer⟩ ⟨real⟩)</code>	<code>: integer</code>	
<code>real</code>	<code>(⟨integer⟩ ⟨real⟩)</code>	<code>: real</code>	
<code>⟨typeId⟩</code>	<code>(⟨relatedType⟩)</code>	<code>: ⟨typeId⟩</code>	für „ähnliche“ Typen
<code>endfile</code>	<code>(⟨fileObj⟩)</code>	<code>: boolean</code>	
<code>read</code>	<code>(⟨fileObj⟩, ⟨vhdObj⟩);</code>		
<code>write</code>	<code>(⟨fileObj⟩, ⟨vhdObj⟩);</code>		
<code>file_close</code>	<code>(⟨fileObj⟩);</code>		neu in VHDL'93
<code>file_open</code>	<code>([⟨status⟩,] ⟨fileObj⟩, ⟨file string⟩ [, ⟨mode⟩]);</code>		
<code>read</code>	<code>(⟨fileObj⟩, ⟨arrayObj⟩, ⟨length⟩);</code>		

A.7 std_logic_1164

Benutzung

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

Deklarationen

```
type std_ulogic          is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');

type std_ulogic_vector is array (natural range <>) of std_ulogic;
type std_logic_vector  is array (natural range <>) of std_logic;

subtype std_logic      is resolved std_ulogic;
subtype x01           is resolved std_ulogic range 'X' to '1';
subtype x01z          is resolved std_ulogic range 'X' to 'Z';
subtype ux01          is resolved std_ulogic range 'U' to '1';
subtype ux01z         is resolved std_ulogic range 'U' to 'Z';

to_bit                (<std_ulogic> [, <xMap>])           : bit
to_bitvector          (<std_(u)logic_vector> [, <xMap>]) : bit_vector
to_stdulogic          (<bit>)                             : std_ulogic
to_stdlogicvector     (<bit_vector> | <std_ulogic_vector>) : std_logic_vector
to_stdulogicvector    (<bit_vector> | <std_logic_vector>) : std_ulogic_vector

to_x01                (<std_(u)logic> | <std_(u)logic_vector>) : x01
                    (<bit> | <bit_vector>)                   : x01
to_x01z              (<std_(u)logic> | <std_(u)logic_vector>) : x01z
                    (<bit> | <bit_vector>)                   : x01z
to_ux01              (<std_(u)logic> | <std_(u)logic_vector>) : ux01
                    (<bit> | <bit_vector>)                   : ux01

is_x                  (<std_(u)logic> | <std_(u)logic_vector>) : boolean

rising_edge (<std_(u)logic>)                               : boolean
falling_edge (<std_(u)logic>)                              : boolean
```

Operatoren

	Typ- <i>a</i>	Typ- <i>b</i>	Typ- <i>op</i>
and or nand	std_(u) logic	= <i>a</i>	= <i>a</i>
nor xor xnor	std_(u) logic_vector		
not	—		= <i>a</i>

A.8 numeric_std / numeric_bit

Benutzung

```
library ieee;                                     Package: numeric_std
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

use ieee.numeric_bit.all;                         Package: numeric_bit
```

Deklarationen

```
type unsigned      is array (natural range <>) of std_logic | bit;
type signed        is array (natural range <>) of std_logic | bit;

to_integer (<signed>)                : integer
to_integer (<unsigned>)              : natural
to_signed  (<integer>, <size>)       : signed
to_unsigned (<natural>, <size>)      : unsigned

resize (<signed>, <size>)            : signed
resize (<unsigned>, <size>)          : unsigned

to_01 (<signed> [, <xMap>])          : signed
to_01 (<unsigned> [, <xMap>])        : unsigned

std_match (<unsigned>, <unsigned>)   : boolean
std_match (<signed>, <signed>)       : boolean
std_match (<std_(u)logic>, <...>)    : boolean
std_match (<std_(u)logic_vector>, <...>) : boolean

rising_edge (<bit>)                  : boolean
falling_edge (<bit>)                  : boolean
```

Operatoren

and or nand nor xor xnor	Typ- <i>a</i> signed unsigned	Typ- <i>b</i> = <i>a</i>	Typ-⟨ <i>op</i> ⟩ = <i>a</i>
= /= < <= > >=	signed integer unsigned natural	⇒ ⇒	boolean boolean
shift_left shift_right rotate_left rotate_right sll srl rol ror	signed unsigned	integer	= <i>a</i>
+ -	signed integer unsigned natural	⇒ ⇒	signed unsigned
-	signed		signed
* / mod rem	signed integer unsigned natural	⇒ ⇒	signed unsigned
abs	signed		signed
not	signed unsigned		= <i>a</i>

A.9 textio

Benutzung

```
use std.textio.all;
```

Deklarationen

```
type line is access string;                                     Typen
type text is file of string;

file input  : text open read_mode is "STD_INPUT";           Dateien
file output : text open write_mode is "STD_OUTPUT";

read      (<lineVar>, <vhdObj> [, <status>]);
readline  (<fileObj>, <lineVar>);
write     (<lineVar>, <vhdObj> [, right|left, <width>]);
write     (<lineVar>, <realObj> [, right|left, <width>, <digits>]);
write     (<lineVar>, <timeObj> [, right|left, <width>, <unit>]);
writeline (<fileObj>, <lineVar>);
endfile   (<fileObj>)                                     : boolean
```

A.10 std_logic_textio

Benutzung

```
library ieee;
use std.textio.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_textio.all;
```

Deklarationen

```
read      (<lineVar>, <vhdObj> [, <status>]);                bin.
hread     (<lineVar>, <vhdObj> [, <status>]);                hex.
oread     (<lineVar>, <vhdObj> [, <status>]);                oct.
write     (<lineVar>, <vhdObj> [, right|left, <width>]);
hwrite    (<lineVar>, <vhdObj> [, right|left, <width>]);
owrite    (<lineVar>, <vhdObj> [, right|left, <width>]);
```

A.11 Attribute

zu Typen/Objekten

Wertermittlung			
$\langle \text{type} \rangle$ 'succ ($\langle \text{typeExpr} \rangle$)	: $\langle \text{value} \rangle$	-nachfolgender Wert zu $\langle \text{typeExpr} \rangle$	
$\langle \text{type} \rangle$ 'pred ($\langle \text{typeExpr} \rangle$)	: $\langle \text{value} \rangle$	-vorhergehender	-- $\langle \text{typeExpr} \rangle$
$\langle \text{type} \rangle$ 'leftof ($\langle \text{typeExpr} \rangle$)	: $\langle \text{value} \rangle$	-linker	-- $\langle \text{typeExpr} \rangle$
$\langle \text{type} \rangle$ 'rightof($\langle \text{typeExpr} \rangle$)	: $\langle \text{value} \rangle$	-rechter	-- $\langle \text{typeExpr} \rangle$
Ordnung			
$\langle \text{type} \rangle$ 'pos ($\langle \text{typeExpr} \rangle$)	: $\langle \text{integer} \rangle$	-Position von $\langle \text{typeExpr} \rangle$	
$\langle \text{type} \rangle$ 'val ($\langle \text{position} \rangle$)	: $\langle \text{integer} \rangle$	-Wert von $\langle \text{position} \rangle$	
Ein- und Ausgabe			
$\langle \text{type} \rangle$ 'image ($\langle \text{typeExpr} \rangle$)	: $\langle \text{string} \rangle$	-Text $\langle \text{typeExpr} \rangle$	
$\langle \text{type} \rangle$ 'value ($\langle \text{string} \rangle$)	: $\langle \text{value} \rangle$	-Wert zu $\langle \text{string} \rangle$	
übergeordnete Typen – als Argument weiterer Attribute			
$\langle \text{type} \rangle$ 'base($\langle \text{attribute} \rangle$)	: $\langle \text{baseType} \rangle$	-Basistyp zu $\langle \text{type} \rangle$	
Bereichsgrenzen			
$\langle \text{type/obj} \rangle$ 'left [$\langle \langle n \rangle \rangle$]	: $\langle \text{index} \rangle$	-linke Grenze ($\langle n \rangle$)	
$\langle \text{type/obj} \rangle$ 'right [$\langle \langle n \rangle \rangle$]	: $\langle \text{index} \rangle$	-rechte Grenze ($\langle n \rangle$)	
$\langle \text{type/obj} \rangle$ 'high [$\langle \langle n \rangle \rangle$]	: $\langle \text{index} \rangle$	-obere Grenze ($\langle n \rangle$)	
$\langle \text{type/obj} \rangle$ 'low [$\langle \langle n \rangle \rangle$]	: $\langle \text{index} \rangle$	-untere Grenze ($\langle n \rangle$)	
Array- und Typdefinition			
$\langle \text{type/obj} \rangle$ 'length[$\langle \langle n \rangle \rangle$]	: $\langle \text{integer} \rangle$	-Anzahl der Elemente ($\langle n \rangle$)	
$\langle \text{type/obj} \rangle$ 'ascending[$\langle \langle n \rangle \rangle$]	: $\langle \text{boolean} \rangle$	-aufsteigender Index ($\langle n \rangle$)	
Bereiche			
$\langle \text{type/obj} \rangle$ 'range [$\langle \langle n \rangle \rangle$]	: $\langle \text{range} \rangle$	-Indexbereich ($\langle n \rangle$)	to downto
$\langle \text{type/obj} \rangle$ 'reverse_range[$\langle \langle n \rangle \rangle$]	: $\langle \text{range} \rangle$	-Indexbereich ($\langle n \rangle$)	downto to

zu Signalen

aktueller Zeitpunkt, liefert Wert			
$\langle \text{signal} \rangle$ 'event	: $\langle \text{boolean} \rangle$	-Signaländerung	
$\langle \text{signal} \rangle$ 'active	: $\langle \text{boolean} \rangle$	-Signalaktivität	
vorheriger Zeitpunkt, liefert Wert			
$\langle \text{signal} \rangle$ 'last_event	: $\langle \text{time} \rangle$	-Zeit seit letzter Signaländerung	
$\langle \text{signal} \rangle$ 'last_active	: $\langle \text{time} \rangle$	-	-- Signalaktivität
$\langle \text{signal} \rangle$ 'last_value	: $\langle \text{value} \rangle$	-Wert vor letzter Signaländerung	
abgeleitete Signale			
$\langle \text{signal} \rangle$ 'delayed[$\langle \langle \text{timeExpr} \rangle \rangle$]	signal: $\langle \text{type} \rangle$	-Verzögerung $\langle \text{timeExpr} \rangle$	
$\langle \text{signal} \rangle$ 'stable [$\langle \langle \text{timeExpr} \rangle \rangle$]	signal: boolean	-keine Änderung seit $\langle \text{timeExpr} \rangle$	
$\langle \text{signal} \rangle$ 'quiet [$\langle \langle \text{timeExpr} \rangle \rangle$]	signal: boolean	-keine Aktivität seit $\langle \text{timeExpr} \rangle$	
$\langle \text{signal} \rangle$ 'transaction	signal: bit	-Wertewechsel bei Aktivität	

A.12 reservierte Bezeichner

VHDL'93

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

Literaturverzeichnis

- [AG00] James R. Armstrong, Gail F. Gray: *VHDL design representation and synthesis*. Prentice-Hall, Inc.; Englewood Cliffs, NJ, 2000. ISBN 0-13-021670-4
- [APT02] Peter J. Ashenden, Gregory D. Peterson, Darrell A. Teegarden: *The System Designer's Guide to VHDL-AMS*. Morgan Kaufmann Publishers, Inc.; San Mateo, CA, 2002. ISBN 1-55860-749-8
- [Ash01] Peter J. Ashenden: *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers, Inc.; San Mateo, CA, 2001. ISBN 1-55860-674-2
- [Bha99] Jayaram Bhasker: *A VHDL primer*. Prentice-Hall, Inc.; Englewood Cliffs, NJ, 1999. ISBN 0-13-096575-8
- [Cha99] K. C. Chang: *Digital systems design with VHDL and synthesis – An integrated approach*. IEEE Computer Society Press; Los Alamitos, CA, 1999. ISBN 0-7695-0023-4
- [Coh99] Ben Cohen: *VHDL coding styles and methodologies*. Kluwer Academic Publishers; Boston, MA, 1999. ISBN 0-7923-8474-1
- [H⁺00] Ulrich Heinkel [u. a.]: *The VHDL Reference: A Practical Guide to Computer-Aided Integrated Circuit Design including VHDL-AMS*. John Wiley & Sons; New York, NY, 2000. ISBN 0-471-89972-0
- [LWS94] Gunther Lehmann, Bernhard Wunder, Manfred Selz: *Schaltungsdesign mit VHDL*. Franzis'-Verlag; Poing, 1994. ISBN 3-7723-6163-3
- [ML93] Stanley Mazor, Patricia Langstraat: *A guide to VHDL*. Kluwer Academic Publishers; Boston, MA, 1993. ISBN 0-7923-9387-2
- [Per98] Douglas L. Perry: *VHDL*. Mc Graw-Hill; London, 1998. ISBN 0-07-049436-3
- [PT97] David Pellerin, Douglas Taylor: *VHDL Made Easy!* Prentice-Hall, Inc.; Englewood Cliffs, NJ, 1997. ISBN 0-13-650763-8
- [Yal01] Sudhakar Yalamanchili: *Introductory VHDL: From simulation to synthesis*. Prentice-Hall, Inc.; Englewood Cliffs, NJ, 2001. ISBN 0-13-080982-9

Standards

- [IEEE 1076 87] *Standard 1076, IEEE Standard VHDL Language Reference Manual*. Institute of Electrical and Electronics Engineers, Inc.; New York, NY, 1987
- [IEEE 1076 93] *Standard 1076-1993, IEEE Standard VHDL Language Reference Manual*. Institute of Electrical and Electronics Engineers, Inc.; New York, NY, 1993. ISBN 1-55937-376-8
- [IEEE 1076 02] *Standard 1076-2002, IEEE Standard VHDL Language Reference Manual*. Institute of Electrical and Electronics Engineers, Inc.; New York, NY, 2002. ISBN 0-7381-3247-0
- [IEC 61691-1-1 04] *IEC 61691-1-1 First edition 2004-10; IEEE 1076 – Behavioural languages - Part 1-1: VHDL language reference manual*. International Electrotechnical Commission; Genf, 2004. ISBN 2-8318-7691-5
- [IEEE 1076 08] *Standard 1076-2008, IEEE Standard VHDL Language Reference Manual*. Institute of Electrical and Electronics Engineers, Inc.; New York, NY, 2009. ISBN 978-0-7381-5801-3
- [IEEE 1076.1 07] *Standard 1076.1-2007, IEEE Standard VHDL Analog and Mixed-Signal Extensions*. Institute of Electrical and Electronics Engineers, Inc.; New York, NY, 2007. ISBN 0-7381-5627-2
- [IEC 61691-6 09] *IEC 61691-6; IEEE 1076.1-2009 – Behavioural languages - Part 6: VHDL Analog and Mixed-Signal Extensions*. Institute of Electrical and Electronics Engineers, Inc.; New York, NY, 2009. ISBN 978-0-7381-6283-6
- [IEEE 1076.2 96] *Standard 1076.2-1996, IEEE Standard VHDL Mathematical Packages*. Institute of Electrical and Electronics Engineers, Inc.; New York, NY, 1996. ISBN 0-7381-0988-6
- [IEC 61691-3-2 01] *IEC 61691-3-2 First edition 2001-06 – Behavioural languages - Part 3-2: Mathematical operation in VHDL*. International Electrotechnical Commission; Genf, 2001. ISBN 0-580-39086-1
- [IEEE 1076.3 97] *Standard 1076.3-1997, IEEE Standard VHDL Synthesis Packages*. Institute of Electrical and Electronics Engineers, Inc.; New York, NY, 1997. ISBN 1-5593-7923-5
- [IEC 61691-3-3 01] *IEC 61691-3-3 First edition 2001-06 – Behavioural languages - Part 3-3: Synthesis in VHDL*. International Electrotechnical Commission; Genf, 2001. ISBN 0-580-39087-X

- [IEEE 1076.4 01] *Standard 1076.4-2000, IEEE Standard VITAL ASIC (Application Specific Integrated Circuit) Modeling Specification 2001*. Institute of Electrical and Electronics Engineers, Inc.; New York, NY, 2001. ISBN 0-7381-2691-0
- [IEC 61691-5 04] *IEC 61691-5 First edition 2004-10; IEEE 1076.4 – Behavioural languages - Part 5: VITAL ASIC (application specific integrated circuit) modeling specification*. International Electrotechnical Commission; Genf, 2004. ISBN 2-8318-7684-2
- [IEEE 1076.6 99] *Standard 1076.6-1999, IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis*. Institute of Electrical and Electronics Engineers, Inc.; New York, NY, 1999. ISBN 0-7381-1819-2
- [IEC 62050 04] *IEC 62050 First edition 2005-07; IEEE 1076.6 – IEEE Standard for VHDL Register Transfer Level (RTL) synthesis*. International Electrotechnical Commission; Genf, 2004. ISBN 0-7381-4065-1
- [IEEE 1164 93] *Standard 1164-1993, IEEE Standard Multivalued Logic System for VHDL Model Interoperability*. Institute of Electrical and Electronics Engineers, Inc.; New York, NY, 1993. ISBN 1-55937-299-0 (withdrawn)
- [IEC 61691-2 01] *IEC 61691-2 First edition 2001-06 – Behavioural languages - Part 2: VHDL multilogic system for model interoperability*. International Electrotechnical Commission; Genf, 2001. ISBN 0-580-39266-X

WWW-Links

Die folgende Liste beinhaltet einige VHDL-relevante Internet-Adressen. Sie soll als Ausgangspunkt für die weitere Suche dienen, da wegen der Dynamik des Netzes nur ein Bruchteil der interessanten Links dargestellt werden kann. Stand: April 2011

Hamburg VHDL-Server	http://tams.informatik.uni-hamburg.de/vhdl
Organisationen	
EDA Industry Working Groups	http://www.eda.org
Accellera	http://www.accellera.org
Standards	
Design Automation Standards Committee	http://www.dasc.org
IEEE Standards	http://standards.ieee.org
IEC Standards	http://std.iec.ch
kommerzielle CAD-Programme	
Cadence	http://www.cadence.com
Mentor Graphics	http://www.mentor.com
Model Technology	http://model.com
Symphony EDA	http://www.symphonyeda.com
Synopsys	http://www.synopsys.com
freie Projekte / Links	
GHDL	http://ghdl.free.fr
FreeHDL Project	http://freehdl.seul.org
GPL Electronic Design Automation	http://www.gpleda.org
HADES	http://tams.informatik.uni-hamburg.de/applets/hades/webdemos
Linux EDA	http://www.linuxeda.com
Scientific Applications on Linux	http://sal.jyu.fi/Z/1
Intellectual Properties	
OpenCores	http://www.opencores.org
Free Model Foundry	http://www.freemodelfoundry.com
Diverses	
VHDL Online	http://www.vhdl-online.de
Design Automation Cafe	http://www.dacafe.com

Index

Symbole	
:=	31, 91
<=	31, 43, 49, 90, 103
guarded	50, 60, 103
select	50, 103
when	50, 103
A	
access	18, 73
alias	20, 74
architecture	5, 70
array	13, 73
Aggregat	15
Index	13, 73
assert	34, 51, 88, 102
attribute	81, 82
vordefiniert	20, 22, 47, 113
B	
block	60, 104
C	
case	32, 94
component	52, 83
Bindung	8, 54, 56, 72, 84
Instanziierung	53, 106
configuration	8, 56, 72
constant	23, 75
D	
downto	19, 73
E	
entity	4, 69
exit	33, 97
F	
file	17, 24, 73, 78
Funktionen	25, 108
function	36, 79, 80
G	
generate	60, 105
generic	4, 52, 69, 83, 104
map	53, 72, 84, 104, 106
I	
if	32, 93
inertial	43, 49, 103
L	
library	63, 85
loop	33, 95
N	
next	33, 96
null	35, 98
numeric_std / numeric_bit	29, 110
P	
package	62, 71
Objekt	63
port	4, 41, 52, 69, 83, 104
map	53, 72, 84, 104, 106
procedure	37, 79, 80
Anweisung	38, 51, 92, 101
process	30, 100
R	
record	16, 73
reject	43, 49, 103
report	34, 89
return	35, 99

INDEX

S.....	
signal	24, 41, 77
Auflösungsfunktion	45
std_logic_1164	29, 109
std_logic_textio	17, 112
subtype	19, 44, 73
T.....	
textio	17, 112
to	19, 73
transport	43, 49, 103
type	
access	18, 73
array	13, 73
Aufzählungs-	13, 73
file	17, 73
Konvertierung	29, 108ff.
Qualifizierung	28
U.....	
use	63, 86
V.....	
variable	23, 76
W.....	
wait	34, 87