Schaltungsdesign mit VHDL

Gunther Lehmann, Bernhard Wunder, Manfred Selz

Vorwort

VHDL[1] ist ein weltweit akzeptierter Standard zur Dokumentation, funktionalen Simulation und zum Datenaustausch beim Entwurf digitaler Systeme. VHDL hat in den letzten Jahren ausgehend von den USA seinen weltweiten Siegeszug angetreten. Mittlerweile findet die Sprache in vielen Entwicklungsabteilungen Verwendung; kaum ein Unternehmen wird sich dem Einsatz von VHDL beim Entwurf digitaler Hardware entziehen können.

Das Einsatzgebiet von VHDL wurde im Laufe der Zeit in Richtung Synthese erweitert. Damit wurden neue, produktivere Wege in der Elektronikentwicklung eröffnet. Die aktuellen Bestrebungen internationaler Gremien gehen in Richtung analoger Erweiterung des Standards, was den technologischen Fortschritten und der Entwicklung hin zu gemischt analog-digitalen Systemen bzw. Mikrosystemen dienlich sein wird.

Die Probleme, die beim Einsatz von VHDL auftreten, dürfen jedoch nicht verschwiegen werden. Es handelt sich um eine sehr mächtige Sprache, die erst nach längerem praktischen Einsatz richtig beherrscht wird. Der Einstieg ist insbesondere für diejenigen Hardwareentwickler schwierig, die noch nicht intensiv mit einer Programmiersprache gearbeitet haben. Die psychologische Barriere darf dabei nicht unterschätzt werden. Hinzu kommt, daß es mit der Einführung der "Sprache" VHDL allein nicht getan ist: Die darauf basierende Entwurfsmethodik erfordert eine neue Arbeitsweise, ein Überdenken gewohnter Schemata und nicht zuletzt die Verwendung neuer Werkzeuge.

---

1 VHDL = VHSIC (Very High Speed Integrated Circuit) Hardware Description Language

Die anfänglichen technischen Probleme (fehlende Herstellerbibliotheken, relativ langsame Simulation auf Gatterebene, kein automatisiertes "Backannotation"[1]) sind schon weitgehend beseitigt. Eine grössere Herausforderung stellt allerdings die Tatsache dar, daß der durch verschiedene Syntheseprogramme unterstützte VHDL-Sprachumfang eingeschränkt und nicht identisch ist.

Ein gravierendes Problem ist auch die starke Abhängigkeit des Syntheseergebnisses von der Qualität der VHDL-Beschreibung, mit dem Schlagwort "what you write is what you get" treffend beschrieben.

In den letzten Jahren wurde versucht, durch Einführung sog. "Front-End-Tools" den Entwickler vom Erlernen und vollen Verständnis der Sprache zu entlasten. Diese Werkzeuge erzeugen aus einer graphisch definierten Verhaltensbeschreibung per Knopfdruck VHDL-Code, der oft als "synthesegerecht" bezeichnet wird. Dadurch gestaltet sich der Entwurfsablauf für viele Anwendungsfälle produktiver, denn ein Automatengraph oder ein Statechart ist nun einmal anschaulicher und leichter zu übersehen als seitenlange IF...THEN...ELSE- und CASE-Anweisungen.

Die oben genannten Abhängigkeiten des Syntheseergebnisses vom VHDL-Code stellen hohe Ansprüche an diese Werkzeuge. Da Front-End- und Synthesetools in der Regel jedoch von verschiedenen Herstellern angeboten werden, ist der erzeugte VHDL-Code für die anschließende Synthese oft wenig optimiert bzw. teilweise sogar ungeeignet. Die Abhängigkeiten sind dabei so komplex, daß die erforderlichen manuellen Änderungen am Quellcode nur von Experten beherrscht werden. Vor einem blinden Vertrauen auf das Ergebnis dieser Werkzeugkette soll deshalb gewarnt werden: VHDL nur als Datenaustauschformat ohne Verständnis der Syntax und Semantik eingesetzt, kann leicht zu unbefriedigenden oder gar schlechten Ergebnissen führen. Deshalb sind Bücher, die das notwendige Hintergrundwissen zur Syntax und zur Interpretation der Sprache VHDL liefern, auch beim Einsatz modernster Entwicklungswerkzeuge unverzichtbar.

---

1 Backannotation = Nachträgliche Berücksichtigung layoutabhängiger Verzögerungszeiten

In dem vorliegenden Buch ist es den Autoren gelungen, erstmals eine umfassende deutschsprachige Einführung in Syntax und Semantik der Sprache VHDL sowie deren Anwendung für Simulation und Synthese zu geben und anhand von einfachen Beispielen zu erläutern. Damit wird zur Verbreitung von VHDL im deutschsprachigen Raum ein wichtiger Beitrag erbracht.

Karlsruhe, im März 1994

Prof. Dr.-Ing. K. D. Müller-Glaser

Zu diesem Buch

Als eine der wenigen deutschsprachigen Veröffentlichungen über VHDL soll das vorliegende Werk eine breite Leserschicht ansprechen: Entscheidungsträger finden hier Informationen über Aufgaben von VHDL, die Designmethodik und Anwendungsmöglichkeiten der Hardwarebeschreibungssprache. Anwender werden vor allem durch den VHDL-Syntaxteil angesprochen, der sowohl für Anfänger als auch für Experten interessante Hinweise, Tips und Tricks und Nachschlagemöglichkeiten bietet. Für diese Gruppe von Lesern sind auch die Kapitel über die Anwendung von VHDL gedacht.

Das Buch gliedert sich in vier Teile:

Teil A legt als Einführung die geschichtliche Entwicklung sowie die Aufgabengebiete von VHDL dar. Ebenso wird der Einsatz von VHDL im strukturierten Elektronikentwurf und der grundsätzliche Aufbau von VHDL-Modellen erläutert.

Der zweite Teil (**Teil B**) bildet den Schwerpunkt des vorliegenden Werkes und enthält die Beschreibung der VHDL-Syntax. Hier werden alle Konstrukte der zur Zeit von Entwicklungswerkzeugen unterstützten Sprachnorm ("VHDL´87") vorgestellt. Daneben wird auf alle Veränderungen eingegangen, welche die überarbeitete Norm "VHDL´93" mit sich bringt.

In **Teil C** schließlich findet die praktische Anwendung von VHDL, mit den Schwerpunkten Simulation und Synthese, ihren Platz. Es werden zahlreiche Hinweise zur Erstellung von VHDL-Modellen gegeben.

Im letzten Abschnitt des Buches, dem Anhang (**Teil D**), finden sich neben einer Literaturliste auch Hinweise zu VHDL-Gremien und Arbeitsgruppen. Mehrere Übungsaufgaben ermöglichen die Vertiefung der erworbenen Kenntnisse. Die Musterlösungen dazu können der dem Buch beiliegenden Diskette entnommen werden.

Dieses Buch basiert auf den Kenntnissen und Erfahrungen, die wir während unserer Tätigkeit am Lehrstuhl für Rechnergestützten Schaltungsentwurf der Universität Erlangen-Nürnberg und am Institut für Technik der Informationsverarbeitung der Universität Karlsruhe erworben haben. Im Vordergrund standen hier die Betreuung zahlreicher VHDL-Sprachkurse für Industriekunden und die Vermittlung des Themas VHDL an Studenten in Erlangen und Karlsruhe.

Wir danken Herrn Prof. Dr.-Ing. K. D. Müller-Glaser (Institut für Technik der Informationsverarbeitung) und Herrn Dr.-Ing. H. Rauch (Lehrstuhl für Rechnergestützten Schaltungsentwurf) für ihr weitreichendes Interesse und die Unterstützung, welche die Entstehung dieses Buches erst ermöglicht haben.

Außerdem möchten wir uns bei Herrn G. Wahl, dem Programmleiter für Elektronikbücher des Franzis-Verlages, für die vielfältigen Anregungen und die Bereitschaft, das neue Thema "VHDL" aufzugreifen, bedanken.

Wir möchten weiterhin darauf hinweisen, daß das Titelbild lediglich symbolischen Charakter hat. Die unterschiedlichen Namen im Entity-Rahmen des abgedruckten VHDL-Modells haben sich leider beim Übertragen in die Druckvorlage eingeschlichen.

Karlsruhe und Erlangen, im März 1994

Gunther Lehmann, Bernhard Wunder, Manfred Selz

Inhalt

Teil A Einführung

Teil B Die Sprache VHDL

Teil D Anhang

Teil A Einführung

1 Entwurf elektronischer Systeme

1.1 Motivation

Die heutige Situation beim Entwurf elektronischer Systeme ist durch folgende Aspekte gekennzeichnet:

- Komplexität und Integrationsdichte nehmen ständig zu. Hauptgesichtspunkte dabei sind:
  - höhere Packungsdichten aufgrund geringerer Strukturgrößen beschleunigen das Vordringen von Produkten mit komplexem, elektronischem "Innenleben",
  - die Anforderungen an die Leistungsfähigkeit (Performance) der Elektronik (geringer Platzbedarf, hohe Taktrate, geringer Leistungsverbrauch, hohe Zuverlässigkeit) steigen.
- Wachsender Konkurrenzdruck und Anforderungen der Kunden bedingen immer kürzere Entwicklungszeiten. Eine Verzögerung der Markteinführung eines Produktes kann den Umsatz drastisch verringern und damit den Erfolg eines Unternehmens gefährden ("time to market"-Problematik).
- Die Komplexität, eine Wiederverwendung von Entwurfsdaten und die Wartung des fertigen Produktes erfordern eine vollständige, widerspruchsfreie und verständliche Dokumentation.

1.2 Entwurfssichten

Die Entwicklung elektronischer Systeme ist bei der heutigen Komplexität und den genannten Anforderungen nur durch eine strukturierte Vorgehensweise beherrschbar. Idealerweise wird man, ausgehend von einer Spezifikation auf Systemebene, die Schaltungsfunktion par-

titionieren ("Funktionale Dekomposition") und die grundsätzlichen Funktionen den einzelnen Modulen zuordnen. Schrittweise wird der Entwurf weiter strukturiert und zunehmend mit Details der Implementierung versehen, bis die für die Fertigung des elektronischen Systems notwendigen Daten vorliegen. Dies können Programmierdaten für Logikbausteine, Layouts für Leiterplatten oder Maskenbänder für die IC-Fertigung sein.

Man unterscheidet beim Entwurf elektronischer Systeme üblicherweise die drei Sichtweisen Verhalten, Struktur und Geometrie. Diese Einteilung wird durch das sog. Y-Diagramm verdeutlicht (Abb. A-1):

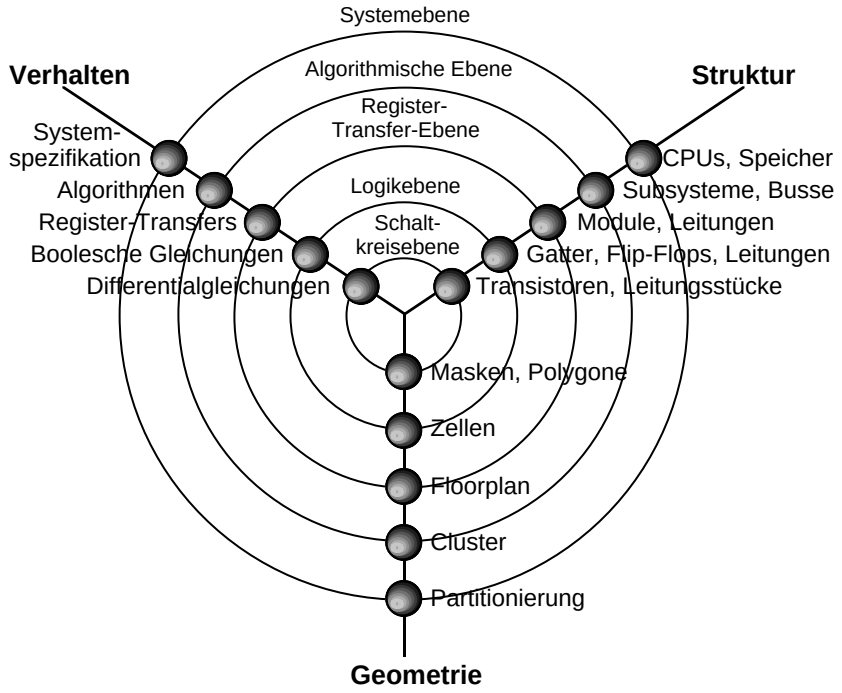

Abb. A-1: Y-Diagramm nach Gajski-Walker [WAL 85]

Gleichzeitig zu den drei Sichtweisen, die durch die Äste im Y-Diagramm repräsentiert werden, sind auch die verschiedenen Abstraktionsebenen durch Kreise mit unterschiedlichen Radien dargestellt. Ein großer Radius bedeutet hohe Abstraktion. Man kann nun vereinfacht den Entwurf elektronischer Systeme als eine Reihe von Transformatio-

nen (Wechsel der Sichtweise auf einem Abstraktionskreis) und Verfeinerungen (Wechsel der Abstraktionsebene innerhalb einer Sichtweise) im Y-Diagramm darstellen. Beginnend auf dem Verhaltensast in der Systemebene wird der Entwurf durch Verfeinerungs- und Syntheseschritte bis hin zum Layout auf dem geometrischen Ast durchgeführt. Ein "Place- and Route"-Werkzeug für Standardzellenentwürfe überführt beispielsweise eine strukturale Beschreibung in der Logikebene (Gatternetzliste) in eine geometrische Beschreibung in der Schaltkreisebene (IC-Layout).

Das reine Top-Down-Vorgehen (Entwicklung in Richtung Kreismittelpunkt) kann dabei nicht immer konsequent beibehalten werden. Verifikationsschritte zwischen den einzelnen Ebenen zeigen Fehler beim Entwurf auf. Gegebenenfalls muß man das jeweilige Entwurfsergebnis modifizieren, den Entwurfsschritt wiederholen oder sogar auf höherer Abstraktionsebene neu einsetzen. Man spricht deshalb auch vom "Jojo-Design".

1.3 Entwurfsebenen

1.3.1 Systemebene

Die Systemebene beschreibt die grundlegenden Charakteristika eines elektronischen Systems. In der Beschreibung werden typische Blöcke, wie Speicher, Prozessoren und Interface-Einheiten verwendet. Diese Module werden durch ihre Funktionalität (im Falle eines Prozessors z.B. durch dessen Befehlssatz), durch Protokolle oder durch stochastische Prozesse charakterisiert. Auf dieser Ebene dominieren meist noch die natürliche Sprache und Skizzen als Beschreibungsmittel.

Es werden auf Systemebene noch keinerlei Aussagen über Signale, Zeitverhalten und detailliertes funktionales Verhalten der einzelnen Blöcke getroffen. Vielmehr dient sie der Partitionierung der gesamten Schaltungsfunktion. In der geometrischen Sicht erfolgt auf dieser Ebene beispielsweise eine erste Unterteilung der Chipfläche.

1.3.2 Algorithmische Ebene

Auf dieser Ebene wird ein System oder eine Schaltung durch nebenläufige (d.h. parallel ablaufende) Algorithmen beschrieben. Typische Beschreibungselemente dieser Ebene sind Funktionen, Prozeduren, Prozesse und Kontrollstrukturen. Auf der Algorithmischen Ebene wird ein elektronisches System in der strukturalen Sicht durch allgemeine Blöcke beschrieben, die über Signale miteinander kommunizieren. In der Verhaltenssicht dagegen wird die Beschreibung des Verhaltens durch eine algorithmische Darstellung mit Variablen und Operatoren vorgenommen.

Es wird kein Bezug zur späteren Struktur der Realisierung (Hardwarepartitionierung) gegeben. Desgleichen werden auch keine zeitlichen Details durch Takt- oder Rücksetzsignale eingeführt.

1.3.3 Register-Transfer-Ebene

Bei Beschreibungen auf der Register-Transfer-Ebene ("Register Transfer Level", RTL) werden die Eigenschaften einer Schaltung durch Operationen (z.B. Addition) und durch den Transfer der verarbeiteten Daten zwischen Registern spezifiziert. Typischerweise werden in die Beschreibung Takt- und Rücksetzsignale integriert. Die einzelnen Operationen sind dann den Werten oder Flanken dieser Signale zugeordnet, so daß die zeitlichen Eigenschaften schon relativ genau definiert werden können.

In der strukturalen Sicht werden Elemente wie Register, Codierer, Multiplexer oder Addierer durch Signale miteinander verknüpft. In der Verhaltenssicht findet man vorwiegend Beschreibungen in Form von endlichen Automaten vor. Die Grobeinteilung der Chipfläche wird in der geometrischen Sicht zu einem sog. Floorplan verfeinert.

Eine mehr oder weniger automatisierte Umsetzung der Verhaltensbeschreibung in eine strukturale Beschreibung auf der Logikebene wird inzwischen durch kommerzielle Werkzeuge angeboten (d.h. Beschreibungen auf RT-Ebene sind meist synthetisierbar).

1.3.4 Logikebene

Auf der Logikebene werden die Eigenschaften eines elektronischen Systems durch logische Verknüpfungen und deren zeitliche Eigenschaften (i.a. Verzögerungszeiten) beschrieben. Der Verlauf der Ausgangssignale ergibt sich dabei durch die Anwendung dieser Verknüpfungen auf die Eingangssignale. Die Signalverläufe sind wertdiskret, d.h. die Signale nehmen nur bestimmte, vordefinierte Logikwerte (z.B. ´low´, ´high´, ´undefined´) an.

In der strukturalen Sichtweise wird der Elektronikentwurf durch eine Zusammenschaltung der Grundelemente (AND-, OR-, XOR-Gatter, Flip-Flops, etc.) dargestellt. Diese Grundelemente werden dabei von einer Bibliothek zur Verfügung gestellt. Innerhalb dieser Bibliothek sind die Eigenschaften der Grundelemente definiert. Diese bilden die Charakteristika der einzelnen Zellen der Zieltechnologie (z.B. 2 µm Standardzellen-Prozeß der Fa. XYZ, FPGA 1.0 der Fa. ABC) in vereinfachter Form nach. Zur Erstellung der strukturalen Beschreibung (Gatternetzliste) werden meistens graphische Eingabewerkzeuge ("Schematic Entry") verwendet.

Die Sichtweise "Verhalten" bedient sich dagegen vor allem einer Darstellung durch Boolesche Gleichungen (z.B. "y = (a AND b) XOR c") oder durch Funktionstabellen.

Der Übergang von der Verhaltenssichtweise auf die strukturale und technologiespezifische Sichtweise erfolgt entweder durch einen manuellen Entwurf oder ein Syntheseprogramm.

1.3.5 Schaltkreisebene

Auch auf der Schaltkreisebene besteht die strukturale Sichtweise aus einer Netzliste. Diesmal sind allerdings keine logischen Gatter kombiniert, sondern elektrische Bauelemente wie Transistoren, Kapazitäten und Widerstände. Einzelne Module werden nun nicht mehr durch eine logische Funktion mit einfachen Verzögerungen beschrieben, sondern durch ihren tatsächlichen Aufbau aus den Bauelementen.

In der geometrischen Sicht werden elektronische Systeme durch Polygonzüge dargestellt, die beispielsweise unterschiedliche Dotierungsschichten auf einem Halbleiter definieren. Die Verhaltenssicht verwendet vornehmlich Differentialgleichungen zur Modellierung des Sy-

stemverhaltens. Dementsprechend aufwendig und rechenzeitintensiv sind die Simulationsalgorithmen.

Die Signale auf Schaltkreisebene können im Gegensatz zur Logikebene prinzipiell beliebige Werte annehmen und weisen einen kontinuierlichen Verlauf über der Zeit auf, d.h. sie sind zeit- und wertkontinuierlich.

Abb. A-2 zeigt die typischen Ebenen beim Entwurf elektronischer Systeme im Überblick.

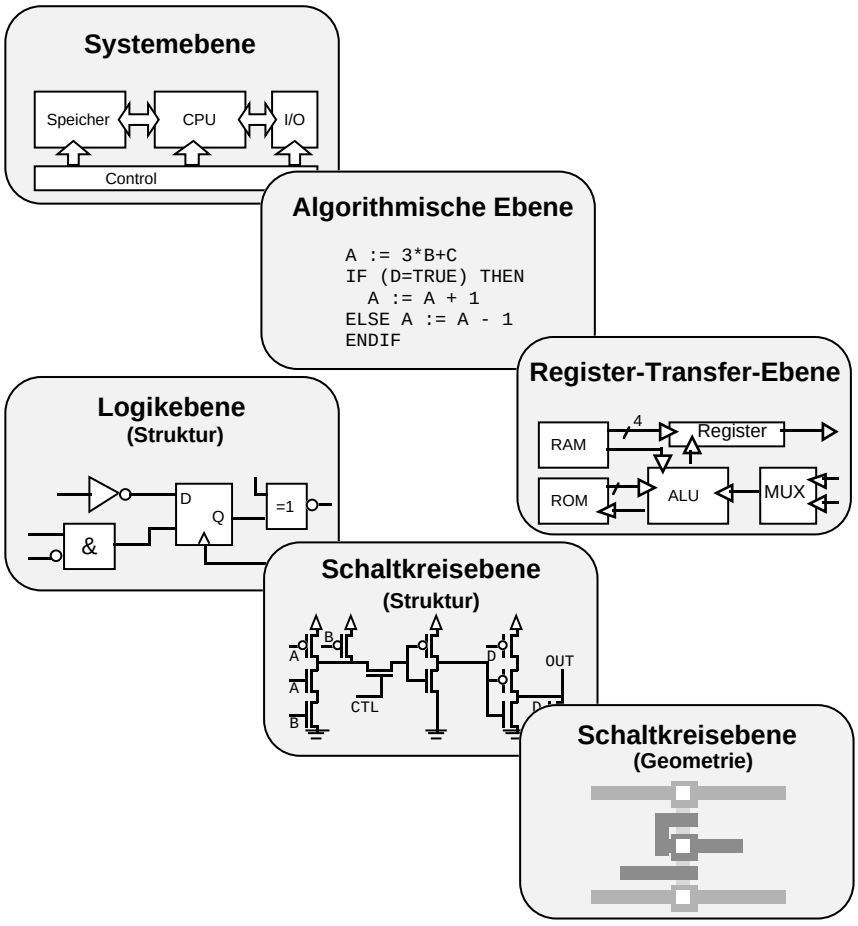

Abb. A-2: Ebenen beim Elektronik-Entwurf

Jede der vorgestellten Entwurfssebenen hat ihren Zweck. Während auf den oberen Ebenen hohe Komplexitäten gut beherrschbar sind, bieten die unteren Ebenen mehr Details und höhere Genauigkeit. So eignen sich die Systemebene und die Algorithmische Ebene für die Dokumentation und Simulation des Gesamtsystems, die Register-Transfer-Ebene für die Blocksimulation und synthesegerechte Modellierung und die Logikebene für Simulationen, mit denen beispielsweise die maximale Taktrate einer Schaltung bestimmt wird oder unerwünschte Impulse ("Spikes") detektiert werden. Auf jeder Ebene wird nur die benötigte Genauigkeit geboten; unwichtige Details sind nicht sichtbar (Abstraktionsprinzip).

Selbstverständlich wird nicht immer eine eindeutige Einordnung einer Beschreibung in eine bestimmten Ebene gelingen, da die Grenzen nicht immer klar definierbar sind. Außerdem ist es möglich, daß die Beschreibung der einzelnen Komponenten eines komplexen, elektronischen Systems auf unterschiedlichen Abstraktionsebenen stattfindet ("Multi-level-Beschreibung").

2 Motivation für eine normierte Hardwarebeschreibungs-sprache

Der Ablauf und die Randbedingungen beim Entwurf elektronischer Systeme haben sich in den letzten Jahren erheblich gewandelt. Dabei treten immer wieder die nachstehenden Probleme auf.

2.1 Komplexität

Die verbesserten Fertigungsmethoden der Halbleitertechnologie gestatten die Entwicklung hochintegrierter Bauelemente. Dies gilt sowohl für die anwendungsspezifischen Schaltkreise (z.B. Standardzellen-Schaltungen) als auch für die anwenderprogrammierbaren Bausteine (FPGAs, GALs, etc.) und die Standardbausteine (RAMs, Prozessoren, etc.). Selbst einfache Produkte, wie z.B. Haushaltsgeräte, enthalten heute umfangreiche elektronische Steuerungen.

Der Elektronikentwickler muß also den Entwurf immer komplexer werdender Systeme beherrschen. Daneben erwarten die Kunden immer kürzere Innovationszyklen bei den Produkten. Dies hat folgende Konsequenzen:

- Ein manueller Entwurf auf Logikebene ist durch graphische oder textuelle Eingabe von Gatternetzlisten nicht mehr beherrschbar, da dieser Vorgang sehr fehleranfällig ist und Simulationen des Gesamtsystems auf Logikebene zu hohe Rechenzeiten erfordern. Die Beschreibung des Systems muß deshalb auf einer abstrakteren Ebene vollzogen und durch geeignete Entwurfssoftware unterstützt werden. Die Überführung in die tieferliegenden Ebenen sollte weitgehend automatisiert sein.

- Entwicklungsaufgaben werden auf mehrere Personen oder Projektteams, die parallel arbeiten, aufgeteilt. Eine fehlerarme Kommunikation zwischen den Entwicklern ist dabei sicherzustellen.
- Entwurfsfehler müssen so früh wie möglich entdeckt werden, um die Kosten und den Zeitbedarf für den notwendigen Neuentwurf ("Redesign") gering zu halten. Um dies zu gewährleisten, sollten bereits Entwürfe der Systemebene oder der Algorithmischen Ebene simulierbar sein ("Simulierbare Spezifikation").
- Die Wiederverwendung von bestehenden Entwürfen kann die Entwicklungskosten und -zeiten verringern sowie das Entwicklungsrisiko eingrenzen. Neben einer geeigneten Dokumentation erfordert die Wiederverwendung Entwürfe mit einem breiten Anwendungsbereich. Details der Implementierung müssen deshalb vermieden werden, d.h. daß beispielsweise der RT-Entwurf eines Halbleiterschaltkreises nicht spezifisch auf eine Fertigungstechnologie ausgerichtet sein darf.

2.2 Datenaustausch

Während des Entwurfs eines elektronischen Systems entsteht eine Vielzahl von Informationen, die in der Summe seine Eigenschaften charakterisieren. Dazu zählen beispielsweise Beschreibungen der Systemfunktionalität, der dynamischen Eigenschaften oder der Einsatzbedingungen. Ein Austausch dieser Informationen findet dabei statt zwischen:

- Auftraggeber und -nehmer (Lasten- und Pflichtenheft),
- verschiedenen Entwicklern eines Projektteams,
- verschiedenen Entwurfsebenen,
- Entwurfsprogrammen verschiedener Hersteller,
- unterschiedlichen Rechnersystemen.

Aufwendige und fehlerträchtige Konvertierungen beim Austausch der Entwurfsdaten können nur vermieden werden, wenn die Beschreibungsmittel herstellerübergreifend normiert sind, keine Spezifika eines bestimmten Rechnersystems enthalten sind und mehrere Entwurfsebenen abgedeckt werden.

Eine Fixierung auf eine herstellerspezifische Beschreibungssprache würde außerdem ein hohes wirtschaftliches Risiko mit sich bringen: Zum einen kann der Hersteller aus verschiedenen Gründen aus dem Markt austreten (Konkurs, Aufkauf durch ein Konkurrenzunternehmen). Zum anderen zieht die Abhängigkeit von einer wenig verbreiteten Beschreibungssprache eine bedeutende Einschränkung bei der Auswahl der Entwurfswerkzeuge nach sich.

2.3 Dokumentation

Die Lebensdauer eines elektronischen Systems ist oft höher als die Verfügbarkeit der Systementwickler. Eine Wartung des Systems, die Wiederverwendung von Systemteilen oder eine technische Modifikation (z.B. der Wechsel auf eine modernere Halbleitertechnologie) machen deshalb eine klare, langlebige und einheitliche Dokumentation notwendig. Die Dokumentation sollte sowohl menschenlesbar als auch durch Rechnerwerkzeuge interpretierbar sein.

Außerdem nimmt mit der steigenden Komplexität der Bausteine auch der Umfang der Dokumentation zu. Die Konsistenz und Vollständigkeit der Beschreibung rücken damit in den Vordergrund.

Eine Verringerung oder Lösung der geschilderten Probleme beim Entwurf elektronischer Systeme ist durch den Einsatz einer Hardwarebeschreibungssprache möglich. Eine solche Sprache, die gleichzeitig für strukturale Beschreibungen und Verhaltensbeschreibungen auf mehreren Entwurfsebenen eingesetzt werden kann, erleichtert den Übergang von der Aufgabenstellung zur Implementierung. Durch die Möglichkeit der Simulation dieser Verhaltensbeschreibung auf hoher Abstraktionsebene kann der Entwurf frühzeitig überprüft werden. Die Synthese der verfeinerten Verhaltensbeschreibung verkürzt die weitere Implementierungszeit ganz erheblich. Ein weiterer wesentlicher Vorteil ist die menschenlesbare Form, die eine Art Selbstdokumentation darstellt. Wird eine solche Hardwarebeschreibungssprache normiert, kann sie auch als Austauschformat zwischen verschiedenen Werkzeugen, Designteams und zwischen Auftraggeber und -nehmer dienen.

Ein Beispiel für eine solche Hardwarebeschreibungssprache ist VHDL.

3 Geschichtliche Entwicklung von VHDL

Die Anfänge der Entwicklung von VHDL[1] reichen bis in die frühen achtziger Jahre zurück. Im Rahmen des VHSIC[2]-Programms wurde in den USA vom Verteidigungsministerium (Department of Defense, DoD) nach einer Sprache zur Dokumentation elektronischer Systeme gesucht. Dadurch sollten die enormen Kosten für Wartung und technische Nachbesserung bei militärischen Systemen, die etwa die Hälfte der Gesamtkosten ausmachten, reduziert werden. Besonderes Augenmerk wurde auf eine Möglichkeit zur sauberen und klaren Beschreibung von komplexen Schaltungen sowie auf die Austauschbarkeit von Modellen zwischen verschiedenen Entwicklungsgruppen gelegt.

Im Rahmen eines vom VHSIC-Programm finanzierten Workshops in Woods Hole, Massachusetts, wurden diese Ziele zuerst diskutiert und dann in eine Vorgabe umgesetzt. Nach mehreren Vorversuchen bekamen im Juli 1983 die Firmen Intermetrics, IBM und Texas Instruments den Auftrag, die neue Sprache mit Programmen zu ihrer Unterstützung zu entwickeln. Die Sprache sollte sich dabei an der existierenden Programmiersprache ADA anlehnen, da das DoD diese Sprache in weitem Umfang einsetzt. ADA-Programmierern werden daher etliche Übereinstimmungen und Ähnlichkeiten auffallen.

Im August 1985 konnte eine erste Version der Hardwarebeschreibungssprache VHDL (VHDL Version 7.2) vorgestellt werden, die dann im Februar 1986 zur Standardisierung an das IEEE[3] übergeben

---

1 VHDL = VHSIC Hardware Description Language

2 VHSIC = Very High Speed Integrated Circuit

3 IEEE = Institute of Electrical and Electronics Engineers

wurde. Mit dieser Aufgabe wurde die VHDL Analysis and Standardization Group (VASG) betraut.

Unter verstärkter Beteiligung der Software- und Hardwarehersteller aus der CAE[1]-Industrie konnte VHDL zunehmend Anhänger gewinnen und wurde schließlich im Dezember 1987 als IEEE 1076-1987 zum ersten und bislang einzigen IEEE-Standard für Hardwarebeschreibungssprachen. Dieser Standard definiert allerdings nur die Syntax und Semantik der Sprache selbst, nicht jedoch ihre Anwendung bzw. ein einheitliches Vorgehen bei der Anwendung. Mittlerweile ist VHDL auch als ANSI[2]-Standard definiert.

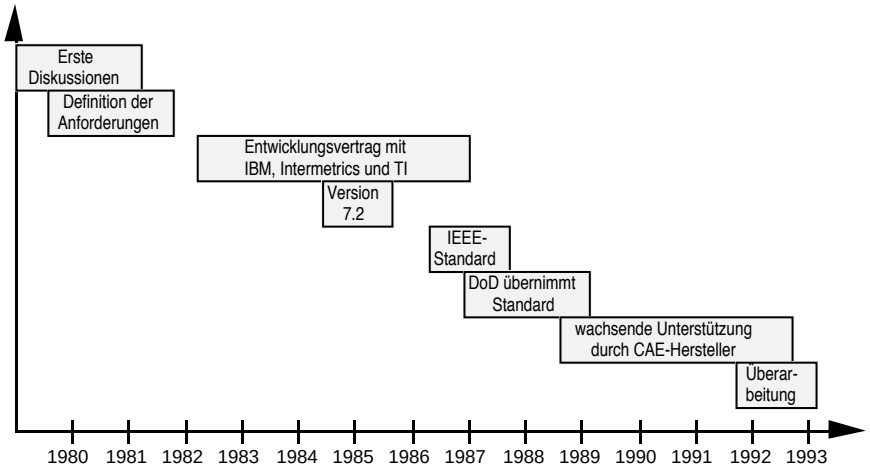

Abb. A-3: Geschichtliche Entwicklung von VHDL

Seit September 1988 müssen alle Elektronik-Zulieferer des DoD VHDL-Beschreibungen ihrer Komponenten und Subkomponenten bereitstellen. Ebenso sind VHDL-Modelle von Testumgebungen mitzuliefern.

---

1 CAE = Computer Aided Engineering

2 ANSI = American National Standards Institute

Nach den IEEE-Richtlinien muß ein Standard alle fünf Jahre überarbeitet werden, um nicht zu verfallen. Aus diesem Grund wurde im Zeitraum Anfang 1992 bis Mitte 1993 die Version IEEE 1076-1993 definiert. Die Dokumentation des neuen Standards, das sog. Language Reference Manual (LRM), wird Mitte 1994 durch das IEEE herausgegeben werden. Die neue Version enthält im wesentlichen "kosmetische" Änderungen gegenüber dem alten Standard, z.B. die Einführung eines XNOR-Operators. Im Teil B dieses Buches wird auf die Unterschiede zwischen dem alten und dem neuen Standard hingewiesen.

Seit Beginn der neunziger Jahre hat VHDL weltweit eine unerwartet große Zahl an Anhängern gefunden. Wurde die 1987er-Version noch durch das DoD finanziert, so ist der neue 1076-1993-Standard unter internationaler Beteiligung entstanden. Europa wirkt unter anderem über ESPRIT (ECIP2) daran mit. Auch Asien (vor allem Japan) hat den VHDL-Standard akzeptiert. Die heutige Konkurrenz von VHDL besteht vor allem in der Verilog HDL, die in den USA weit verbreitet ist, und UDL/I, welches in Japan eingesetzt wird.

Abb. A-4 zeigt eine Übersicht über die heute und in Zukunft verbreiteten Hardwarebeschreibungssprachen (Ergebnis einer Umfrage unter Ingenieuren in den USA [JAI 93]; Mehrfachnennungen waren möglich):

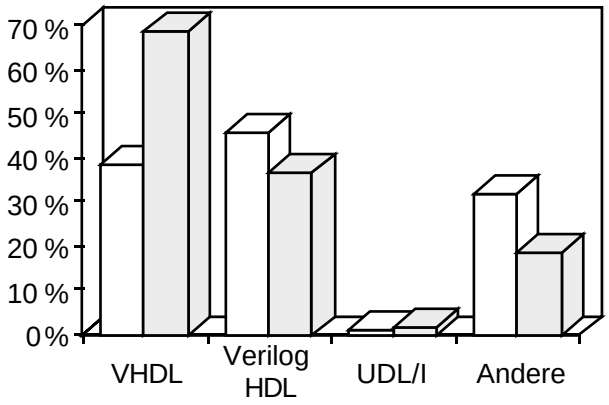

Abb. A-4: Anteile von HDLs heute (hell) und in Zukunft (dunkel) [JAI 93]

4 Aufbau einer VHDL-Beschreibung

Die Beschreibung eines VHDL-Modells besteht meist aus drei Teilen: einer Schnittstellenbeschreibung, einer oder mehreren Architekturen und einer oder mehreren Konfigurationen. VHDL-Beschreibungen können hierarchisch aufeinander aufbauen.

4.1 Schnittstellenbeschreibung (Entity)

In der Entity wird die Schnittstelle der zu modellierenden Komponente / des zu modellierenden Systems beschrieben, also die Ein- und Ausgänge sowie Konstanten, Unterprogramme und sonstige Vereinbarungen, die auch für alle dieser Entity zugeordneten Architekturen gelten sollen.

4.2 Architektur (Architecture)

Eine Architektur enthält die Beschreibung der Funktionalität eines Modells. Hierfür gibt es verschiedene Möglichkeiten. Das Modell kann aus einer Verhaltensbeschreibung bestehen oder strukturalen Charakter (Netzliste) haben. Beide Möglichkeiten können miteinander kombiniert werden. Für eine Entity können mehrere Architekturen definiert werden, d.h. es können für eine Komponentenschnittstelle mehrere Beschreibungen auf unterschiedlichen Abstraktionsebenen oder verschiedene Entwurfsalternativen existieren.

4.3 Konfiguration (Configuration)

In der Konfiguration wird festgelegt, welche der beschriebenen Architekturvarianten einer bestimmten Entity zugeordnet ist und welche Zuordnungen für die möglicherweise verwendeten Submodule in der Architektur gelten. Hier können auch hierarchisch den untergeordneten Entities bestimmte Architekturen zugeordnet werden; außerdem ist es möglich, Parameterwerte an hierarisch tieferliegende Komponenten zu übergeben.

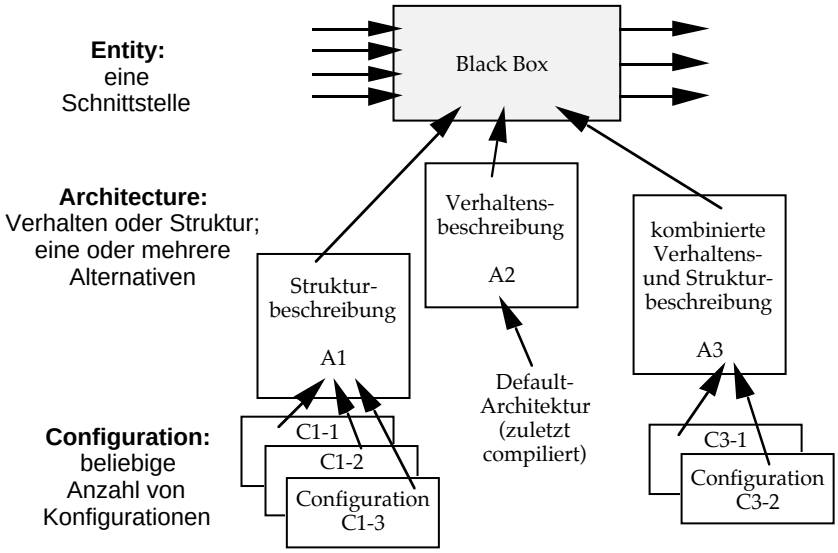

Abb. A-5: Entity, Architecture und Configuration

4.4 Package

Ein Package enthält Anweisungen wie Typ- oder Objektdeklarationen und die Beschreibung von Prozeduren und Funktionen, die in mehreren VHDL-Beschreibungen gebraucht werden. Zum Beispiel kann in einem Package der zu verwendende Logiktyp (zwei- oder mehrwertige Logik) mit allen korrespondierenden Operatoren definiert werden.

Vordefiniert sind bei VHDL die Packages `standard` (enthält den zweiwertigen Logiktyp "`bit`" sowie häufig zu verwendende Funktionen und Typen) und `textio`.

In Abb. A-6 sind die wichtigsten Bestandteile einer VHDL-Beschreibung und ihre Aufgaben im Überblick dargestellt.

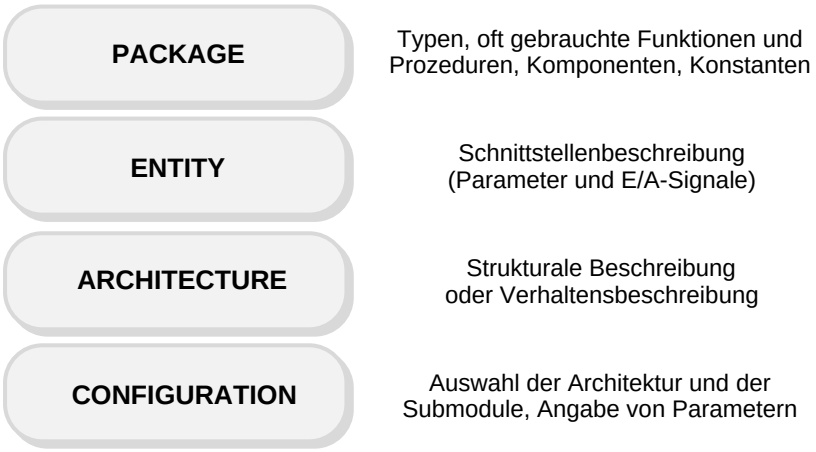

Abb. A-6: Grundbestandteile einer VHDL-Beschreibung

4.5 Beispiel eines VHDL-Modells

Anhand eines einfachen Beispieles soll der grundsätzliche Aufbau eines VHDL-Modells vorab gezeigt werden. Das gewählte AND2-Gatter ist in seinem Aufbau sehr einfach, so daß noch keine Kenntnisse der VHDL-Syntax erforderlich sind, um die Funktion des Modells zu verstehen.

Schnittstellenbeschreibung (Entity)

```
ENTITY and2 IS
   PORT (in1,in2: IN bit; and_out: OUT bit);
   -- definiere Pins als Signale vom Typ "bit"
END and2;
```

Architektur (Architecture)

```
ARCHITECTURE number_one OF and2 IS
BEGIN
   and_out <= in1 AND in2;         -- Verhaltensbeschreibung
END number_one;
```

Konfiguration (Configuration)

```
CONFIGURATION and2_config OF and2 IS
   FOR number_one                -- verknuepfe Architektur
   END FOR;                      -- "number_one" mit Entity
END and2_config;                 -- "and2"
```


5 Entwurfssichten in VHDL

Im vorgestellten Y-Diagramm werden drei Sichten unterschieden. Die Sprache VHDL ermöglicht eine Beschreibung in der strukturalen Sicht und in der Verhaltenssicht. Bei VHDL-Modellen wird deshalb prinzipiell zwischen:

- Verhaltensmodellierung ("behavioral modeling") und
- Strukturaler Modellierung ("structural modeling")

unterschieden. Die Modellierung der geometrischen Sicht eines elektronischen Systems, z.B. die Beschreibung der Layoutdaten, wird von VHDL nicht unterstützt.

5.1 Verhaltensmodellierung

Bei dieser Modellierungsart wird das Verhalten einer Komponente durch die Reaktion der Ausgangssignale auf Änderungen der Eingangssignale beschrieben. Die Komponente verzweigt nicht weiter in Unterkomponenten.

Am Beispiel eines Komparators werden die Vorteile der Verhaltensmodellierung deutlich. Der Komparator soll zwei Bit-Vektoren `a` und `b` miteinander vergleichen und eine logische `'1'` am Ausgang liefern, falls der Wert des Vektors `a` größer als der des Vektors `b` ist.

Das nachfolgende VHDL-Modell kann, gesteuert durch den Parameter `n`, beliebig breite Bit-Vektoren miteinander vergleichen. Eine dieses Verhalten realisierende Schaltung, die aus vielen Gattern aufgebaut ist, kann in der Verhaltenssicht mit VHDL durch wenige Zeilen Quellcode beschrieben werden:

```
ENTITY compare IS
   GENERIC (n  : positive := 4);
   PORT (a, b  : IN  bit_vector (n-1 DOWNTO 0);
         result: OUT bit);
END compare;
```

```
ARCHITECTURE behavioral_1 OF compare IS
BEGIN
   PROCESS (a, b)
   BEGIN
      IF a > b THEN
         result <= '1';
      ELSE
         result <= '0';
      END IF;
   END PROCESS;
END behavioral_1;
```

In der Verhaltenssichtweise werden zwei prinzipielle Beschreibungsmittel unterschieden:

- sequentielle Anweisungen ("sequential statements") und
- nebenläufige Anweisungen ("concurrent statements").

Zur genaueren Betrachtung dieser Zusammenhänge soll ein Halbaddierer dienen. Dessen Schnittstelle ist folgendermaßen aufgebaut:

```
ENTITY halfadder IS
   PORT (sum_a, sum_b: IN bit; sum, carry: OUT bit);
END halfadder;
```

In **sequentiellen** oder auch prozeduralen Beschreibungen werden Konstrukte wie Verzweigungen (`IF-ELSIF-ELSE`), Schleifen (`LOOP`) oder Unterprogrammaufrufe (`FUNCTION`, `PROCEDURE`) verwendet. Die einzelnen Anweisungen werden nacheinander (sequentiell) abgearbeitet. Diese Beschreibungen ähneln den Quelltexten höherer Programmiersprachen.

Eine entsprechende Architektur für den Halbaddierer lautet:

```
ARCHITECTURE behavioral_seq OF halfadder IS
BEGIN
   PROCESS (sum_a, sum_b)
   BEGIN
      IF (sum_a = '1' AND sum_b = '1') THEN
         sum <= '0'; carry <= '1';
      ELSE
         IF (sum_a = '1' OR sum_b = '1') THEN
            sum <= '1'; carry <= '0';
         ELSE
            sum <= '0'; carry <= '0';
         END IF;
      END IF;
   END PROCESS;
END behavioral_seq;
```

Im Gegensatz zu den gängigen Programmiersprachen mit ihren sequentiellen Konstrukten verfügt VHDL zusätzlich noch über **nebenläufige** Anweisungen, die es erlauben, parallel ablaufende Operationen zu beschreiben. Damit wird es möglich, die spezifischen Eigenschaften von Hardware (parallel arbeitende Funktionseinheiten) abzubilden.

Nachstehend ist die Architektur des Halbaddierers in dieser Beschreibungsvariante gezeigt. Die beiden Verknüpfungen `XOR` und `AND` können dabei gleichzeitig aktiv sein:

```
ARCHITECTURE behavioral_par OF halfadder IS
BEGIN
   sum   <= sum_a XOR sum_b;
   carry <= sum_a AND sum_b;
END behavioral_par;
```

5.2 Strukturale Modellierung

Bei der strukturalen Modellierung werden die Eigenschaften eines Modells durch seinen inneren Aufbau aus Unterkomponenten dargestellt. Die Eigenschaften der Unterkomponenten werden in unabhängigen VHDL-Modellen beschrieben. Diese stehen compiliert in Modellbibliotheken ("Libraries") zur Verfügung.

Für den Halbaddierer, der aus einem XOR2- und einem AND2-Gatter aufgebaut ist, ergibt sich beispielsweise folgende Beschreibung:

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

Eine eindeutige Einordnung eines VHDL-Modells in eine der beiden Modellierungsarten ist nicht immer möglich, da VHDL die Verwendung beider Beschreibungsmöglichkeiten innerhalb eines Modells gestattet.

6 Entwurfsebenen in VHDL

Die weiten Modellierungsmöglichkeiten von VHDL unterstützen Beschreibungen in verschiedenen Entwurfsebenen, ausgehend von der Systemebene bis hinab zur Logikebene. Folgende drei Beschreibungsebenen haben dabei die größte Bedeutung:

- Algorithmische Ebene,
- Register-Transfer-Ebene,
- Logikebene.

Daneben finden sich in der VHDL-Literatur Ansätze, die zeigen, daß auch eine Modellierung auf Schaltkreisebene bedingt möglich ist (z.B. [HAR 91]). Die Praxisrelevanz dieser Ansätze ist jedoch gering.

6.1 Algorithmische Ebene

Ein Beispiel für eine Beschreibung auf Algorithmischer Ebene zeigt einen Ausschnitt aus der Architektur eines Schnittstellenbausteins. Der Baustein soll immer dann, wenn er von einem Controller eine Aufforderung erhält, eine Adresse aus einem internen Register frühestens nach 10 ns auf den Bus legen.

Diese Beschreibung enthält keine Angaben über die spätere Schaltungsstruktur und keine Takt- oder Rücksetzsignale.

```
ARCHITECTURE algorithmic_level OF io_ctrl IS

BEGIN
   ...
   write_data_alg: PROCESS

   BEGIN
      WAIT UNTIL adr_request = '1';
      WAIT FOR 10 ns;
      bus_adr <= int_adr;
   END PROCESS write_data_alg;
   ...
END algorithmic_level;
```

6.2 Register-Transfer-Ebene

Um das obige Beispiel auf RT-Ebene darzustellen, wird ein Taktsignal (`clk`) und ggf. ein Rücksetzsignal hinzugefügt und die Operationen in Abhängigkeit von diesen Signalen beschrieben:

```
ARCHITECTURE register_transfer_level OF io_ctrl IS

BEGIN
   ...
   write_data_rtl : PROCESS (clk)
      VARIABLE tmp : boolean;
   BEGIN
      IF rising_edge(clk) THEN
         IF ((adr_request = '1') AND (tmp = false)) THEN
            tmp := true;
         ELSIF (tmp = true) THEN
            bus_adr <= int_adr;
            tmp := false;
         END IF;
      END IF;
   END PROCESS write_data_rtl;
   ...
END register_transfer_level;
```

Hier wird, wenn bei einer aktiven Taktflanke ein gesetztes `adr_req`-Signal entdeckt wird, zunächst die temporäre Variable `tmp` ge-

setzt, damit bei der nächsten aktiven Taktflanke (Wartezeit!) die Adresse auf den Bus geschrieben werden kann. Durch geeignete Wahl der Taktperiode ist sicherzustellen, daß die Wartezeit von mindestens 10 ns eingehalten wird.

Im Gegensatz zur Algorithmischen Ebene wird hier schon ein zeitliches Schema für den Ablauf der Operationen vorgegeben und implizit eine Schaltungsstruktur beschrieben.

Wie die beiden Beispielarchitekturen zeigen, werden Konstrukte der Verhaltensmodellierung sowohl auf Algorithmischer als auch auf Register-Transfer-Ebene verwendet.

6.3 Logikebene

Die Eigenschaften eines elektronischen Systems werden auf der Logikebene durch logische Verknüpfungen digitaler Signale und deren zeitliche Eigenschaften (i.a. durch Verzögerungszeiten der Verknüpfungen) beschrieben. Die Hardwarebeschreibungssprache VHDL besitzt dazu vordefinierte Operatoren (`AND`, `OR`, `XOR`, `NOT` etc.) für binäre Signale (`'0'`, `'1'`) und gestattet die Ergänzung weiterer, benutzerdefinierter Operatoren. Auch Konstrukte zur Modellierung zeitlicher Eigenschaften werden bereitgestellt. Nachstehend ist beispielhaft die Beschreibung der Halbaddierer-Architektur auf der Logikebene in der Verhaltenssichtweise abgebildet.

```
ARCHITECTURE logic_level OF halfadder IS
BEGIN

   sum   <= sum_a XOR sum_b AFTER 3 ns;
   carry <= sum_a AND sum_b AFTER 2 ns;

END logic_level;
```

Die Darstellung in strukturaler Sicht entspricht der obigen Architektur "`structural`". Die Verzögerungszeiten ergeben sich hier aus den internen Verzögerungszeiten der beiden Subkomponenten.

7 Design-Methodik mit VHDL

7.1 Entwurfsablauf

Abb. A-7 zeigt, wie der Entwurfsablauf unter Verwendung von VHDL aussehen könnte:

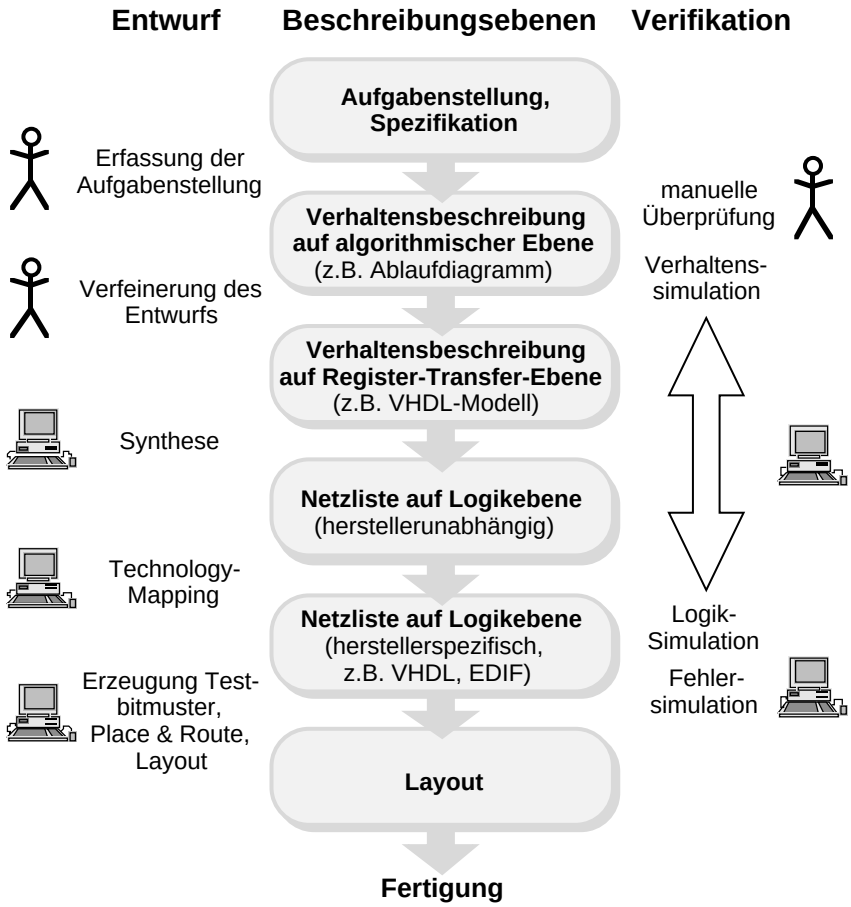

Abb. A-7: Entwurfsablauf mit VHDL

7.1.1 Erfassung der Spezifikation

Am Anfang steht die Erfassung der Spezifikation, die i.d.R. vom Auftraggeber in Form von Texten, Tabellen und Diagrammen an den Auftragnehmer übergeben wird. Diese Spezifikation enthält sowohl funktionale (Beschreibung der Schaltungsfunktion) als auch nichtfunktionale Angaben (Versorgungsspannung, Fertigungstechnologie, maximale Fläche und Verlustleistung, etc.).

Nach einem ggf. vorhandenen Zwischenschritt über manuell erstellte Graphiken (Ablaufpläne, Flußdiagramme, etc.) erfolgte **bisher** der Entwurf des elektronischen Systems von Hand bis zu einer Netzliste, die die Verdrahtung vorhandener Module aus einer technologiespezifischen Bibliothek definiert. Diese Netzliste wurde durch Schematic-Entry-Werkzeuge in den Rechner eingegeben und im weiteren mit entsprechenden Programmen bis zum Layout umgesetzt.

7.1.2 Beschreibung und Simulation auf System- oder Algorithmischer Ebene

Heute ist man jedoch in der Lage, auch in den frühen Phasen des Elektronik-Entwurfs gezielt mit Rechnerunterstützung zu arbeiten. Durch Verwendung einer Hardwarebeschreibungssprache wie VHDL kann nun, ausgehend von einer nichtformalen Spezifikation, auf der abstrakten Ebene von Algorithmen und Funktionsblöcken bereits ein erstes VHDL-Modell der Schaltung erstellt werden. Eine anschließende Simulation der Schaltung auf dieser Ebene zeigt die Korrektheit des Entwurfs oder veranlaßt schon sehr frühzeitig notwendige Korrekturen.

Für viele Einsatzwecke existieren auch Werkzeuge, die eine graphische Eingabe des Entwurfs ermöglichen. Die Erstellung und Simulation abstrakter Verhaltensmodelle wird also graphisch unterstützt. Häufig bieten solche Programme auch die Möglichkeit, VHDL-Code auf Register-Transfer-Ebene zu generieren. Stehen solche Werkzeuge nicht zur Verfügung, muß das VHDL-Modell auf Register-Transfer-Ebene manuell erstellt werden.

7.1.3 Beschreibung auf Register-Transfer-Ebene

Eine Beschreibung auf Algorithmischer Ebene kann von heute kommerziell verfügbaren Synthesewerkzeugen nicht in eine Gatternetzliste umgesetzt werden. Diese Programme erwarten eine Beschreibung auf Register-Transfer-Ebene. Die bereits erstellte abstrakte Beschreibung muß also manuell verfeinert werden. Dabei ist die Kenntnis des zu verwendenden Synthesewerkzeuges und des von ihm unterstützten VHDL-Sprachumfanges äußerst wichtig.

7.1.4 Logiksynthese und Technology Mapping

Bei der anschließenden Synthese der VHDL-RTL-Beschreibung wird weitgehend automatisch eine noch technologieunabhängige Beschreibung in der Logikebene erzeugt. Diese wird dann, nach Auswahl einer Zieltechnologie (z.B. Standardzellen-Prozeß 0.7 µm der Firma XYZ), in eine technologiespezifische Gatternetzliste umgesetzt. Der Begriff "Technology Mapping" bezeichnet diesen Vorgang. Er wird meist durch das Synthesewerkzeug durchgeführt. Anschließende Analysen können zeigen, ob die spezifizierten Eigenschaften, wie maximal zulässige Chipfläche, erreicht worden sind. Logiksimulationen lassen Aussagen über die Einhaltung von Zeitbedingungen zu.

7.1.5 Erzeugung der Testbitmuster

Nach der Erzeugung der Gatternetzliste müssen für den Produktionstest die Testbitmuster generiert werden. An dieser Stelle sind meist Modifikationen des Entwurfs notwendig, um eine gute Testbarkeit der Schaltung zu erreichen. Häufig wird der Entwickler auch bei dieser Tätigkeit durch Programme unterstützt, die das VHDL-Netzlistenmodell als Eingabe akzeptieren.

7.1.6 Layouterzeugung und Produktion

Ausgehend von der technologiespezifischen VHDL-Gatternetzliste werden im abschließenden Entwurfsschritt die für die Produktion des elektronischen Systems notwendigen Layout- oder Programmierdaten erstellt. Für diesen Entwurfsschritt existiert eine Vielzahl kommerzieller Werkzeuge.

Nach Erzeugung des Layouts können sehr genaue Aussagen über die Laufzeiten von Signalen gemacht werden, da jetzt die exakten geometrischen Daten der Verdrahtung bekannt sind. Diese "realen" Verzögerungszeiten werden in die Logikebene zurückgereicht ("Backannotation"), so daß in der Logiksimulation die vom Layout abhängigen Einflüsse berücksichtigt werden können.

7.2 VHDL-Software

Im Bereich des Entwurfs elektronischer Systeme gibt es unterstützende Werkzeuge für Spezifikationserfassung und Dokumentation, funktionale Simulation auf verschiedenen Abstraktionsebenen, Schaltplaneingabe, Fehlersimulation, Synthese, Layout oder Test.

Nicht alle Entwurfsschritte sind sinnvoll mit VHDL durchzuführen. Die Schwerpunkte beim Einsatz von VHDL liegen heute im Bereich der Simulation von Verhaltensmodellen der Algorithmischen oder der Register-Transfer-Ebene sowie der Synthese.

7.2.1 Texteditoren

Der erste Schritt bei der Arbeit mit VHDL besteht in der Regel darin, VHDL-Quellcode einzugeben. Dazu kann jeder beliebige Texteditor verwendet werden (emacs, vi, textedit, etc.). Manche Software-Hersteller bieten im Rahmen ihrer VHDL-Programme einen eigenen VHDL-Editor an. Solche speziellen Editoren können beispielsweise VHDL-Schlüsselwörter (durch Fettdruck o. ä.) besonders hervorheben oder sprachspezifisch auf Eingaben reagieren, z.B. durch automatisches Schließen von Blöcken mit "`END`"-Anweisungen.

7.2.2 Graphische Benutzerschnittstellen

Die Rechnerunterstützung in den frühen Entwurfsphasen gewinnt immer mehr an Bedeutung. Werkzeuge zur Erfassung eines Entwurfs auf hoher Abstraktionsebene finden zunehmende Verbreitung. Oft sind graphische Eingaben (Zeichnen von Automatengraphen oder Warteschlangenmodellen, etc.) möglich.

Eine frühzeitige Simulation des Entwurfs und die Ausgabe von (teilweise garantiert synthetisierbarem) VHDL-Code sind weitere Pluspunkte für diese Art von Programmen.

7.2.3 Simulatoren und Debugger

VHDL-Simulatoren dienen dazu, Schaltungen mit vorgegebenen Stimuli zu simulieren, d.h. die Reaktion aller internen Knoten und insbesondere der Ausgänge auf vorgegebene Änderungen der Eingänge zu ermitteln. Speziell zur Simulation von VHDL-Verhaltensmodellen gehört auch, daß der Zeitverlauf von VHDL-Objekten (Signalen und Variablen) dargestellt werden kann.

Gerade wegen des nebenläufigen Konzepts einer Hardwarebeschreibungssprache sind zusätzlich Funktionen notwendig, die von typischen Software-Debuggern bekannt sind. Dazu zählt beispielsweise das Setzen von sog. Breakpoints, welche die Simulation bei Eintreten einer bestimmten Bedingung (Erreichen einer vorgegebenen Simulationszeit, spezielle Werte von Objekten, etc.) anhalten.

Die meisten kommerziell angebotenen VHDL-Simulatoren können inzwischen jedes beliebige VHDL-Modell verarbeiten (100%-ige Codeabdeckung). Sieht man von wenigen Details ab, die vor allem auf Ungenauigkeiten der VHDL-Norm zurückzuführen sind, kann man von einer einheitlichen Interpretation eines VHDL-Modells durch verschiedene Simulatoren ausgehen.

Erwähnt werden sollte noch, daß auf Logikebene spezialisierte Simulatoren zur Zeit noch deutlich schneller als VHDL-Simulatoren sind. Die Hersteller der VHDL-Simulatoren arbeiten allerdings intensiv an einer Performance-Verbesserung durch neue Simulationstechniken (siehe Teil C) bzw. bieten eine Anbindung ihrer Werkzeuge an spezialisierte "Gate-Level-Simulatoren" an.

7.2.4 Syntheseprogramme

Syntheseprogramme können am meisten dazu beitragen, den Entwurf elektronischer Systeme produktiver zu gestalten. Sie setzen eine funktionale VHDL-Beschreibung (auf Register-Transfer-Ebene) in eine Netzliste auf Logikebene um. Zu diesem Zweck ist eine technologiespezifische Gatterbibliothek erforderlich. Sie enthält Informationen über Fläche, Laufzeiten, Verlustleistung usw. der einzelnen Gatter.

Ein wesentlicher Gesichtspunkt bei der Bewertung eines Syntheseprogrammes ist der Umfang von VHDL-Konstrukten, die synthetisiert werden können. Im Gegensatz zur Simulation kann bei der Synthese nicht der komplette Sprachumfang verarbeitet werden. Dies liegt v.a. daran, daß VHDL nicht primär auf Synthesezwecke hin ausgerichtet wurde, sondern auch viele Konstrukte enthält, die prinzipiell nicht in Hardware umgesetzt werden können.

Außerdem ist zu überprüfen, ob für eine gewünschte Zieltechnologie die werkzeugspezifische Gatterbibliothek verfügbar ist.

8 Bewertung von VHDL

Die vorhergehenden Kapitel haben gezeigt, daß der Einsatz einer normierten Hardwarebeschreibungssprache viele Vorteile für den Entwurf komplexer Elektronik bietet. Der erste Teil des Buches sollte jedoch nicht nur euphorisch VHDL loben, sondern auch einige kritische Worte enthalten. Im folgenden finden sich deshalb einige grundsätzliche Bemerkungen zu VHDL, die allerdings nicht immer objektiv möglich sind. Zum Beispiel kann die umfangreiche Syntax sowohl als Vorteil (viele Modellierungsmöglichkeiten), als auch als Nachteil (hoher Einarbeitungsaufwand) empfunden werden.

8.1 Vorteile von VHDL

8.1.1 Vielseitigkeit

Zweifellos als Vorteil kann es angesehen werden, daß VHDL eine Sprache für viele Zwecke ist.

VHDL ist sowohl für die Spezifikation und Simulation wie auch als Ein- und Ausgabesprache für die Synthese geeignet. Die menschenlesbare Form eignet sich gut zur Dokumentation. Beispielsweise bietet VHDL über benutzerdefinierte Attribute die Möglichkeit, entwurfsbegleitende Angaben, wie Vorgaben zur Fläche oder Laufzeit, zu dokumentieren.

Schließlich ist durch die firmenunabhängige Normierung der Sprache ein Datenaustausch zwischen verschiedenen Programmen, zwischen verschiedenen Entwurfsebenen, zwischen verschiedenen Projektteams und zwischen Entwickler und Hersteller möglich (siehe Abb. A-8).

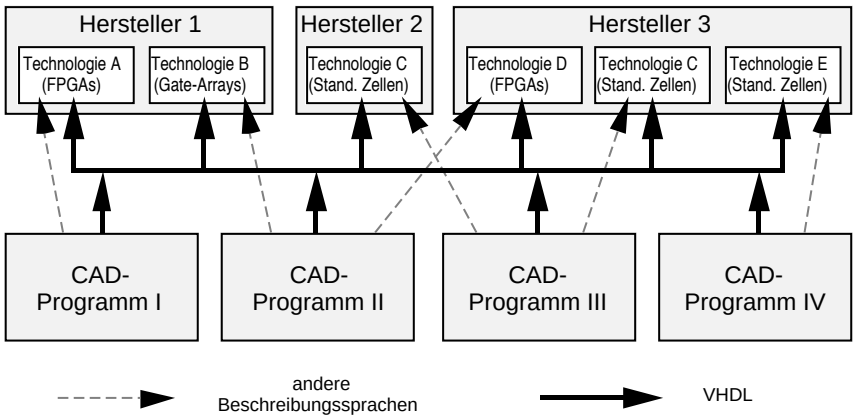

Abb. A-8: Datenaustausch zwischen CAD-Programmen und Halbleiterherstellern

8.1.2 Programmunabhängigkeit

Neben VHDL gibt es auch noch eine Reihe anderer Hardwarebeschreibungssprachen und Datenformate. Man unterscheidet dabei zwischen programmspezifischen und nicht-programmspezifischen Sprachen bzw. Formaten. Erstere sind an die Programme eines bestimmten Software-Herstellers gebunden und werden meist nicht von den Programmen anderer Hersteller unterstützt. Auch die heutige Hauptkonkurrenz von VHDL, die Verilog HDL, war lange Zeit herstellerspezifisch und hat sich erst in neuerer Zeit zu einer unabhängigen HDL entwickelt.

VHDL ist programmunabhängig. Es gibt mittlerweile eine Vielzahl von Software-Anbietern, die für viele Entwurfsschritte eine Lösung unter Einsatz von VHDL anbieten. Man kann somit unter mehreren Alternativen die für die geplante Anwendung optimale Software auswählen.

Außerdem wurde bei der Definition der Sprache VHDL sehr stark auf die Unabhängigkeit von einem bestimmten Rechnersystem geachtet. Die systemabhängigen Aspekte werden in Packages gekapselt, das VHDL-Modell selbst ist unabhängig vom eingesetzten Rechnersystem und damit (fast immer) portierbar.

8.1.3 Technologieunabhängigkeit

VHDL ist technologieunabhängig. Die Entscheidung für eine bestimmte Technologie (FPGA, Gate-Array, Standardzellen, etc.) muß erst zu einem relativ späten Zeitpunkt des Entwurfs getroffen werden. Ein späteres Umschwenken auf eine andere Technologie verursacht kein komplettes Redesign.

Auch die Offenheit der Sprache bzw. die umfangreichen Modellierungsmöglichkeiten tragen wesentlich zur Technologieunabhängigkeit bei. Durch die Freiheit zur Definition von:

- benutzereigenen Logiktypen mit entsprechenden Operatoren,
- Auflösungsfunktionen, die technologiespezifische Signalverknüpfungen (wired-or, wired-and, etc.) modellieren,
- neuen, bei strukturalen Modellen eingesetzten Komponentenbibliotheken

können die spezifischen Eigenschaften bestimmter Technologien mit VHDL abgebildet werden. Man sagt deshalb auch, VHDL arbeitet technologieunterstützend.

8.1.4 Modellierungsmöglichkeiten

VHDL stellt zahlreiche Konstrukte zur Beschreibung von Schaltungen und Systemen zur Verfügung. Mit diesen Konstrukten lassen sich Modelle auf verschiedenen Beschreibungsebenen erstellen: Algorithmische Ebene, Register-Transfer-Ebene und Logikebene. VHDL-Modelle können dabei Unterkomponenten verschiedener Entwurfssichten und -ebenen enthalten. Bei der strukturalen Modellierung kann eine mehrstufige Hierarchie implementiert werden.

Beschreibungen auf abstraktem Niveau haben mehrere Vorteile:

- sie sind kompakt und überschaubar, der Überblick über den gesamten Entwurf bleibt erhalten,
- sie ermöglichen kürzere Entwicklungszeiten,
- sie benötigen weniger Rechenzeit bei der Simulation,
- sie erlauben eine frühzeitige Verifikation.

Durch Kombination mit detaillierteren Beschreibungen ist eine ebenenübergreifende Simulation (sog. "Multi-Level-Simulation") möglich. Einzelne Projektteams können deshalb weitgehend unabhängig voneinander arbeiten, da eine Gesamtsimulation unterschiedlich weit verfeinerter Modelle durchführbar ist. Außerdem gestattet die Multi-Level-Simulation die Kontrolle eines Entwurfsschrittes, indem die Modelle vor und nach dem Entwurfsschritt durch gemeinsame Simulation miteinander verglichen werden können.

8.1.5 Unterstützung des Entwurfs komplexer Schaltungen

Die durch VHDL verstärkte Verbreitung von Synthesewerkzeugen erlaubt ein Umschwenken auf eine neue und produktivere Entwurfsmethodik mit strukturierter Top-Down-Vorgehensweise. Wesentliche Aspekte dabei sind:

- Da die Spezifikation eine simulierbare Beschreibung darstellt, kann der Entwurf frühzeitig überprüft werden,
- Verhaltensbeschreibungen in einer Hardwarebeschreibungssprache können synthetisiert werden,
- Zahlreiche Sprachkonstrukte zur Parametrisierung von Modellen erlauben ein unkompliziertes Variantendesign,
- Die Entwicklung wiederverwendbarer Modelle wird unterstützt,
- Es bestehen Umsetzungsmöglichkeiten auf verschiedene Technologien.

Alles zusammen führt zur Beherrschung von komplexeren Schaltungen und zu einer wesentlich verkürzten Entwicklungszeit.

8.1.6 Selbstdokumentation

VHDL ist eine menschenlesbare Sprache. Die Syntax ist sehr ausführlich gehalten und besitzt viele selbsterklärende Befehle. Dadurch entsteht eine Art Selbstdokumentation.

Bei der zusätzlichen Wahl von geeigneten Objektnamen enthält eine ohne weitere Kommentare versehene Beschreibung durchaus genü-

gend Informationen, um zu einem späteren Zeitpunkt noch ohne Probleme interpretiert werden zu können.

8.2 Nachteile von VHDL

8.2.1 Mehr als eine Sprache: Eine neue Methodik

Mit dem Einsatz von VHDL beim Entwurf ist weit mehr verbunden als bei der Einführung eines neuen Programmes oder eines neuen Formates. Der gesamte Entwurfsablauf hat sich vom manuellen Entwurf mit Schematic Entry auf Logikebene zur Schaltungsbeschreibung auf RT-Ebene mit anschließender Synthese gewandelt.

Dies hat mehrere Folgen:

- Es ist ein grundsätzlich neuer Entwurfsstil einzuführen, verbunden mit einem Umdenken bei den Hardware-Entwicklern. Erfahrungsgemäß haben aber gerade Entwickler, die in ihrer Ausbildung nicht mit modernen Programmiersprachen und der erforderlichen strukturierten Vorgehensweise vertraut gemacht wurden, dabei große Probleme: *"We don't know if to 'harden' a Software engineer or to 'soften' a Hardware engineer",* [BUR 92].
- Die erforderlichen Aus- und Weiterbildungsmaßnahmen verursachen Kosten und Ausfallzeiten.
- Die anfallenden Kosten für die Neuanschaffung einer ganzen Werkzeugpalette (Workstations, Speicher, Lizenzgebühren für Software, etc.) sind enorm.

8.2.2 Modellierung analoger Systeme

VHDL verfügt zwar über umfangreiche Beschreibungsmittel für digitale, elektronische Systeme, bietet aber auch in der neuen Norm (VHDL'93) keine Konstrukte zur Modellierung analoger elektronischer Systeme. Das gleiche gilt für Komponenten mit mechanischen, optischen, thermischen, akustischen oder hydraulischen Eigenschaften und Funktionen. Damit ist VHDL keine Sprache, die die vollständige Verhaltensmodellierung eines technischen Systems zuläßt.

Aktuelle Bestrebungen zielen allerdings auf die Definition einer erweiterten VHDL-Norm (IEEE 1076.1, AHDL), die die Modellierung analoger Schaltkreise mit wert- und zeitkontinuierlichen Signalen gestattet.

8.2.3 Komplexität

Die Komplexität der Sprache VHDL wird als Vorteil aufgrund der vielen Modellierungsmöglichkeiten angesehen, bringt aber auch einige Nachteile mit sich:

- VHDL erfordert einen hohen Einarbeitungsaufwand. Es ist mit einer weitaus längeren Einarbeitungszeit als bei anderen Sprachen zu rechnen. Insbesondere muß ein geeigneter Modellierungsstil eingeübt werden.
- Das Verhalten eines komplexen VHDL-Modells in der Simulation ist für den VHDL-Neuling kaum nachvollziehbar, da die zugehörigen Mechanismen (z.B. "preemption" von Ereignislisten) nicht von gängigen Programmiersprachen abgeleitet werden können.
- Die Semantik wurde in der ursprünglichen Version 1987 an vielen Stellen nicht eindeutig und klar genug festgelegt. In der Überarbeitung der Sprache wurden 1993 einige dieser Stellen beseitigt und die Interpretation damit vereinheitlicht.

Erschwerend zu diesen Problemen kommt, daß das Nachschlagewerk für VHDL, das "Language Reference Manual" (LRM), als eines der am schwersten zu lesenden Bücher der Welt einzustufen ist (*"The base type of a type is the type itself"*, [IEE 88]).

8.2.4 Synthese-Subsets

VHDL ist als Sprache in der Syntax und Simulationssemantik normiert, nicht jedoch für die Anwendung als Eingabeformat für Synthesewerkzeuge. Außerdem enthält VHDL Konstrukte, die sich prinzipiell nicht in eine Hardware-Realisierung umsetzen lassen. Darüber hinaus unterstützt jedes Synthesewerkzeug einen etwas anderen VHDL-Sprachumfang (VHDL-Subset) und erfordert einen spezifischen Modellierungsstil. Daraus resultieren folgende Probleme:

- VHDL-Modelle müssen auf ein spezielles Synthesewerkzeug zugeschnitten sein. Dies verhindert einen unkomplizierten Wechsel des Werkzeugs und erhöht die Abhängigkeit vom Werkzeughersteller.
- Der Elektronik-Entwickler muß die Anforderungen des gewählten Werkzeuges kennen und von Anfang an bei der Modellerstellung berücksichtigen.

8.2.5 Noch keine umfassende Unterstützung

Obwohl seit Ende der 80er Jahre die Unterstützung von VHDL durch Werkzeuge vieler Software-Hersteller enorm zugenommen hat, ist die heutige Situation noch nicht zufriedenstellend.

Ein Mangel besteht vor allem bei den Simulations- und Synthesebibliotheken für logische Gatter und Standardbausteine. Jede neue Technologie erzwingt eine komplette Neufassung der umfangreichen Daten, die oft erst zeitverzögert zur Verfügung gestellt werden. In diesem Punkt ist die schon länger existierende Hardwarebeschreibungssprache Verilog der neueren Sprache VHDL überlegen, da die Zahl der bestehenden Verilog-Bibliotheken noch weitaus größer ist.

8.2.6 Ausführlichkeit

Die Ausführlichkeit der Sprache VHDL kann auch als Nachteil empfunden werden. Der oft als "zu geschwätzig" empfundene Stil verursacht lange und umständliche Beschreibungen. Vor allem bei der manuellen Modellerstellung verhindert der Umfang des einzugebenden Textes ein schnelles Vorgehen.

Teil B Die Sprache VHDL

1 Allgemeines

1.1 VHDL´87 oder VHDL´93 ?

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

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

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

Zu den wesentlichen Neuerungen im 93-er Standard gehören:

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

1.2 Vorgehensweise und Nomenklatur

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

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

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

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

2 Sprachelemente

2.1 Sprachaufbau

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

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

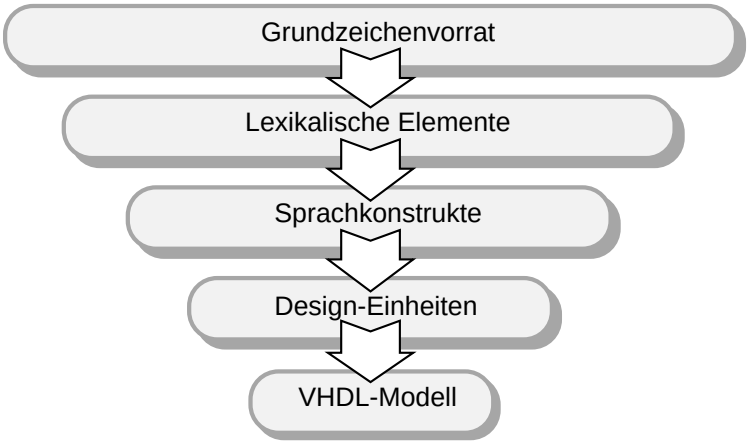

Abb. B-1: VHDL-Sprachaufbau

2.2 Zeichensatz

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

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

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

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

```
TYPE character IS (
   ...
   ...    -- alle Zeichen aus VHDL'87
   ...
   C128,  C129,  C130,  C131,  C132,  C133,  C134,   C135,
   C136,  C137,  C138,  C139,  C140,  C141,  C142,   C143,
   C144,  C145,  C146,  C147,  C148,  C149,  C150,   C151,
   C152,  C153,  C154,  C155,  C156,  C157,  C158,   C159,
   ' ',   '¡',   '¢',   '£',   '¤',   '¥',   '|',|', '§',
   '¨',   '©',   'ª',   '«',   '¬',   '-',   '®',    '¯',
   '°',   '±',   '2',   '3',   '´',   'µ',   '¶',    '.',
   '¸',   '1',   'º',   '»',   '1/4', '1/2', '3/4',  '¿',
   'À',   'Á',   'Â',   'Ã',   'Ä',   'Å',   'Æ',    'Ç',
   'È',   'É',   'Ê',   'Ë',   'Ì',   'Í',   'Î',    'Ï',
   '☆',   'Ñ',   'Ò',   'Ó',   'Ô',   'Õ',   'Ö',    '☆',
   'Ø',   'Ù',   'Ú',   'Û',   'Ü',   '☆',   '☆',    'ß',
   'à',   'á',   'â',   'ã',   'ä',   'å',   'æ',    'ç',
   'è',   'é',   'ê',   'ë',   'ì',   'í',   'î',    'ï',
   '☆',   'ñ',   'ò',   'ó',   'ô',   'õ',   'ö',    '∏',
   'Ø',   'ù',   'ú',   'û',   'ü',   '☆',   '☆',    'ÿ');
```

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

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

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

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

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

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

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

2.3 Lexikalische Elemente

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

Lexikalische Elemente können in folgende Gruppen eingeteilt werden:

2.3.1 Kommentare

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

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

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

2.3.2 Bezeichner

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

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

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

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

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

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

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

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

2.3.3 Reservierte Wörter

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

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

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

2.3.4 Größen

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

2.3.4.1 Numerische Größen

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

Abstrakte numerische Größen zur Basis 10

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

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

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

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

Abstrakte numerische Größen zu von 10 verschiedenen Basen

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

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

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

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

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

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

Physikalische Größen

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

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


2.3.4.2 Zeichengrößen

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

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

2.3.4.3 Zeichenketten

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

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

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

2.3.4.4 Bit-String-Größen

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

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

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

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

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

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

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

2.3.5 Trenn- und Begrenzungszeichen

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

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

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

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

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

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

2.4 Sprachkonstrukte

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

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

2.4.1 Primitive

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

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

2.4.1.1 Operanden

Es gibt folgende Alternativen für Operanden:

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

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

Einige Beispiele für verschiedene Operanden bzw. Ausdrücke:

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

2.4.1.2 Operatoren und Prioritäten

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

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

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

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

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

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

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

2.4.2 Befehle

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

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

Deklarationen

Hierbei handelt es sich im einzelnen um:

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

Sequentielle Befehle

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

Nebenläufige Befehle

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

Konfigurationsbefehle

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

2.4.3 Syntaktische Rahmen

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

3 Objekte

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

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

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

3.1 Objektklassen

Konstanten

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

Variablen

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

Signale

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

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

Dateien

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

3.2 Datentypen und Typdeklarationen

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

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

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


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

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

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

Typdeklarationen können an folgenden Stellen auftreten:

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

3.2.1 Einfache Typen

3.2.1.1 Aufzähltypen

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

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

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

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

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

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

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

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

3.2.1.2 Ganzzahlige Typen

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

```
TYPE  int_type_name IS RANGE   range_low
                    TO         range_high;

TYPE  int_type_name IS RANGE   range_high
                    DOWNTO     range_low;

TYPE  int_type_name IS RANGE
                    other_int_type_name'RANGE;
```

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

3.2.1.3 Fließkommatypen

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

```
TYPE real_type_name IS RANGE range_low
                    TO       range_high;

TYPE real_type_name IS RANGE range_high
                    DOWNTO   range_low;

TYPE real_type_name IS RANGE
                 other_real_type_name'RANGE;
```

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

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

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

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

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

3.2.1.4 Physikalische Typen

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

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

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

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

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


Der einzige vordefinierte, physikalische Typ ist `time`:

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

3.2.1.5 Abgeleitete einfache Typen

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

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

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

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

Die Verwendung von abgeleiteten Typen oder Untertypen hat folgende Vorteile:

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

Das nachstehende Beispiel illustriert den letztgenannten Vorteil:

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

Unter Verwendung des Attributes `HIGH` vordefinierte Untertypen sind:

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

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

3.2.1.6 Typumwandlungen

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

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

```
integer (float_object_name)
real (integer_object_name)

int_type_name (other_int_type_obj_name)
float_type_name (other_float_type_obj_name)
```

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

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

Funktionen zur Umwandlung zwischen verschiedenen Logiktypen müssen selbst definiert werden.

3.2.2 Feldtypen

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

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

3.2.2.1 Vektoren

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

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

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

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

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

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

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

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

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

3.2.2.2 Mehrdimensionale Felder

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

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

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

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

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

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

3.2.2.3 Abgeleitete Feldtypen

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

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

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

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

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

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

3.2.3 Zusammengesetzte Typen

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

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


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

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

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

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

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

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

3.3 Objektdeklarationen

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

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

Die Deklaration von Objekten enthält also Elemente, die

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

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

3.3.1 Konstanten

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

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

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

Konstantendeklarationen dürfen an folgenden Stellen stehen:

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

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

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

3.3.2 Variablen

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

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

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

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

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

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

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

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

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

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

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

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

3.3.3 Signale

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

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

Die Syntax der Signaldeklaration lautet wie folgt:

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

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

Signaldeklarationen dürfen an folgenden Stellen der Design-Einheiten stehen:

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

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

3.3.4 Aliase

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

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

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

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

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

ALIAS alias_name    IS   aliased_type;

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

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

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

3.3.5 Implizite Deklaration

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

3.3.6 Weitere Deklarationen

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

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

3.4 Ansprechen von Objekten

3.4.1 Objekte mit einfachem Typ

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

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

3.4.2 Objekte mit Feldtyp

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

Indexed Names

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

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

Ein Beispiel zur Verwendung von "indexed names":

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

Sliced Names

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

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

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

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

Aggregate

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

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

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

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

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

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

- `single_element`
- `range_low TO range_high`

- `range_high DOWNTO range_low`
- `OTHERS`

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

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

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

3.4.3 Objekte mit zusammengesetztem Typ

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

```
record_name.record_element_name
```

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

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

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

3.5 Attribute

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

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

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

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

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

4 Aufbau eines VHDL-Modells

Ein komplettes VHDL-Modell besteht aus mehreren Teilen (Design-Einheiten): einer Schnittstellenbeschreibung, der sog. "Entity", einer oder mehreren Verhaltens- oder Strukturbeschreibungen, den sog. "Architectures" und gegebenenfalls mehreren Konfigurationen, den sog. "Configurations".

Die Aufteilung der einzelnen Design-Einheiten auf eine oder mehrere Dateien ist willkürlich; es muß lediglich die Reihenfolge beim Compilieren der einzelnen Design-Einheiten für die Simulation beachtet werden: Entity-Architecture-Configuration:

- Stehen alle zu einem Modell gehörenden Design-Einheiten in einer Datei, dann müssen sie in der angegebenen Reihenfolge verfaßt sein, da der Compiler die Datei sequentiell abarbeitet.
- Stehen die Design-Einheiten in verschiedenen Dateien, so müssen die Dateien in der entsprechenden Reihenfolge compiliert werden.

Werden bestimmte Objekttypen, Funktionen oder Prozeduren von mehreren Modellen benötigt, so werden sie üblicherweise in unabhängigen Einheiten, den sog. "Packages", abgelegt.

4.1 Bibliotheken

Struktural aufgebaute Modelle greifen auf hierarchisch tiefer liegende VHDL-Beschreibungen zu. Diese Abhängigkeiten von anderen Modellen und Packages wird durch das Konzept der Bibliotheken ("Libraries") gehandhabt.

Bibliotheken dienen als Aufbewahrungsort für compilierte und (wieder) zu verwendende Design-Einheiten. In einem Dateisystem werden die Bibliotheken meist als eigene Verzeichnisse realisiert. Damit dieses Konzept jedoch unabhängig von den Namenskonventionen eines be-

stimmten Betriebssystems bleibt, werden die Bibliotheken nur über logische Namen (VHDL-Bezeichner) angesprochen. Der Bezug zwischen einem Verzeichnispfad und dem logischen VHDL-Namen wird in werkzeugspezifischen Konfigurationsdateien festgelegt oder über spezielle Aktionen hergestellt.

Standardmäßig legt der VHDL-Compiler die compilierten Design-Einheiten in der Bibliothek mit dem logischen Namen `work` ab. Neben den Modellen aus dieser Bibliothek (auch als Working-Library bezeichnet), können Modelle auch aus weiteren, sog. Resource-Libraries verwendet werden.

Abb. B-2 zeigt diese Zusammenhänge:

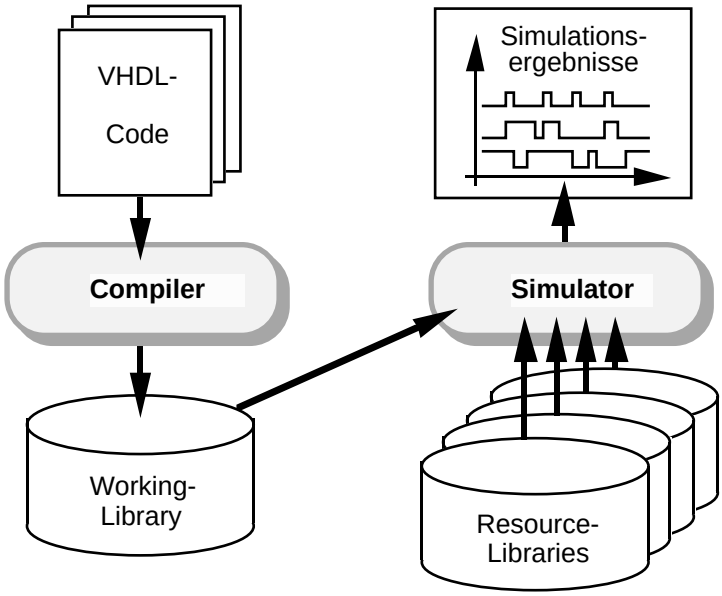

Abb. B-2: Konzept der VHDL-Bibliotheken

Innerhalb einer Bibliothek müssen die Namen (Bezeichner) der Design-Einheiten Package, Entity und Configuration eindeutig, d.h. unterscheidbar sein. Bei den Architekturen hingegen müssen nur die Bezeichner der zu ein- und derselben Entity gehörenden Architekturen voneinander verschieden sein.

4.1.1 Die LIBRARY-Anweisung

Vor dem Ansprechen eines Bibliotheksobjektes muß die verwendete Bibliothek der entsprechenden Design-Einheit bekanntgemacht werden. Dies geschieht durch folgende Anweisung:

```
LIBRARY library_name_1 {, library_name_n} ;
```

Implizit (d.h. ohne `LIBRARY`-Anweisung) sind die Bibliotheken `std` und `work` bekannt. In `std` sind die allgemeinen Packages abgelegt. Die Bibliothek `work` dient, wie bereits erwähnt, zum Abspeichern eigener, compilierter Modelle.

Die `LIBRARY`-Anweisung kann vor einer Entity, vor einer Architektur, vor einer Configuration, vor einem Package oder Package Body stehen (in der sog. "context clause" der Design-Einheiten).

4.1.2 Die USE-Anweisung

Nach der Bekanntgabe der Bibliothek geschieht das Ansprechen von Bibliothekselementen mit Hilfe von "selected names" durch den vorangestellten logischen Bibliotheksnamen. Soll ein Element, das in einem Package definiert wurde, angesprochen werden, so muß zusätzlich zum Bibliotheksnamen der Package-Name im "selected name" enthalten sein:

```
LIBRARY cmos_lib;
...
a := cmos_lib.tech_data.c1;
-- Konstante c1 im Package tech_data der Library cmos_lib
```

Das direkte Ansprechen von Funktionen und Modellen (d.h. ohne vorangestellten Bibliotheksnamen und Package-Namen) kann erfolgen, wenn vor Verwendung des Elementes in einem VHDL-Modell eine `USE`-Anweisung eingesetzt wird, z.B.:

```
USE library_name.ALL ;
USE library_name.element_name ;
```

```
USE library_name.package_name.ALL ;
USE library_name.package_name.element_name ;
```

Damit sind alle bzw. nur die spezifizierten Elemente der Bibliothek oder des Packages sichtbar. In einer USE-Anweisung können, durch Kommata getrennt, auch mehrere Bibliothekselemente stehen.

Die USE-Anweisung kann vor der Design-Einheit stehen, für die sie Gültigkeit haben soll, oder im Deklarationsteil von Design-Einheiten, Blöcken, Prozessen, Funktionen und Prozeduren sowie innerhalb der Packages. Der Gültigkeitsraum ist dementsprechend eingeschränkt.

Implizit, d.h. auch ohne USE-Anweisung, sind alle Elemente des Packages `std.standard` sichtbar. Die Elemente des normierten Packages `std.textio` müssen explizit mit einer USE-Anweisung sichtbar gemacht werden.

```
LIBRARY work, std;              -- implizit enthalten
USE std.standard.ALL;           -- implizit enthalten
USE std.textio.ALL;             -- Elemente in textio sichtbar
LIBRARY cmos_lib;               -- Bib. cmos_lib gueltig
USE cmos_lib.tech_data.ALL;     -- c1 jetzt direkt ansprechbar
```

4.2 Schnittstellenbeschreibung (Entity)

Die einzelnen Modelle eines komplexen VHDL-Entwurfs kommunizieren über die Entity, d.h. über deren Schnittstellenbeschreibung, miteinander. Die "Kommunikationskanäle" nach außen sind die sog. Ports eines Modells. Für diese werden in der Entity Name, Datentyp und Signalflußrichtung festgelegt.

Außerdem werden in der Schnittstellenbeschreibung die Parameter deklariert, die dem Modell übergeben werden können (sog. "Generics"). Mit Hilfe dieser Parameter lassen sich beispielsweise die Verzögerungs- oder Set-Up-Zeiten eines Modells von außen an das Modell übergeben. Generics können auch die Bitbreite der Ports bestimmen oder den Namen einer Datei enthalten, in der die Programmierdaten eines PLA-Modells abgelegt sind. Auf diese Weise bieten Generics eine hohe Flexibilität bei der Konfiguration von Modellen, da eine

Änderung dieser Übergabeparameter kein Neu-Compilieren des Modells erfordert.

Neben den Ports und Generics können in der Schnittstellenbeschreibung Deklarationen gemacht werden, die für die Entity und damit auch für alle zugehörigen Architekturen Gültigkeit besitzen.

Im optionalen Anweisungsteil ("Entity Statement Part", zwischen `BEGIN` und `END`) können darüberhinaus passive Prozesse, der Aufruf passiver Prozeduren oder nebenläufige Assertions stehen. Passiv bedeutet, daß keine Signalzuweisung innerhalb des Prozesses / der Prozedur erfolgt.

Die Entity ist folgendermaßen aufgebaut:

```
ENTITY entity_name IS
 [ GENERIC (
      param_1 {, param_n } :    type_name
              [ := def_value ]
  { ; further_generic_declarations  } );]

 [ PORT (
  {  port_1  {, port_n }  : IN type_name
              [ := def_value ] }
  { ; port_declarations_of_mode_OUT }
  { ; port_declarations_of_mode_INOUT }
  { ; port_declarations_of_mode_BUFFER } );]
  ...
  ...   -- USE-Anweisungen, Disconnections
  ...   -- Deklaration von:
  ...   -- Typen und Untertypen, Aliases,
  ...   -- Konstanten, Signalen, Files,
  ...   -- Unterprogrammen, Attributen
  ...   -- Definition von:
  ...   -- Unterprogrammen, Attributen
  ...   -- VHDL'93: Groups, Shared Variables
  ...
[ BEGIN
  ...
  ...   -- passive Befehle, Assertions
  ... ]
END [ENTITY] [entity_name] ;
```


Die optionale Wiederholung des Schlüsselwortes ENTITY in der END-Anweisung ist nur in ✓93 möglich.

Mit jeder Port-Deklaration der Entity wird implizit ein Signal gleichen Typs und gleichen Namens deklariert, das unter bestimmten Einschränkungen - abhängig vom Modus des Ports - in der Entity und in den zugehörigen Architekturen verwendet werden kann:

- Modus IN: Ports können nicht geschrieben werden,
- Modus OUT: Ports können nicht gelesen werden,
- Modus INOUT: Ports können gelesen und geschrieben werden,
- Modus BUFFER: Ports können gelesen und nur von einer Quelle geschrieben werden.

Zusätzliche interne Signale müssen im Deklarationsteil der Entity oder der Architektur deklariert werden.

Ein einfaches Beispiel für eine Entity (NAND3-Gatter):

```
ENTITY nand3 IS
   GENERIC       ( delay       : TIME := 2.2 ns ) ;
   PORT          ( a, b, c     : IN  std_ulogic := '0';
                   y           : OUT std_ulogic ) ;
   TYPE          tristate IS ('0', '1', 'Z');
BEGIN
   ASSERT ((a /= 'X') AND (b /= 'X') AND (c /= 'X'))
      REPORT "Ungueltiger Wert am Eingang";
END nand3;
```

4.3 Architektur (Architecture)

Die Architektur enthält die Beschreibung der Modelleigenschaften. Diese Beschreibung kann sowohl aus der Verhaltenssichtweise als auch aus der strukturalen Sichtweise erfolgen. Ein bestimmtes Modell kann also durch sein Verhalten oder durch seinen Aufbau (interne Module und deren Verbindungen) beschrieben werden. Mischformen beider Beschreibungsvarianten sind innerhalb eines Modells möglich.

Einer Schnittstellenbeschreibung können keine, eine oder mehrere Architekturen besitzen; beispielsweise kann eine Architektur eine Model-

lierung auf Register-Transfer-Ebene enthalten, während die andere eine technologiespezifische Beschreibung auf Logikebene mit detaillierten Verzögerungszeiten bereitstellt.

Auch die Architektur besteht aus einem Deklarationsteil (vor `BEGIN`) für lokale Typen, Objekte, Komponenten usw. und einem Bereich mit Anweisungen, dem "Architecture Statement Part" (zwischen `BEGIN` und `END`). Dieser enthält die eigentliche Modellbeschreibung.

Der prinzipielle Aufbau einer Architektur ist wie folgt:

```
ARCHITECTURE arch_name OF entity_name IS
   ...
   ...  -- USE-Anweisungen, Disconnections
   ...  -- Deklaration von:
   ...  -- Typen und Untertypen,
   ...  -- Aliases, Konstanten,
   ...  -- Signalen, Files, Komponenten,
   ...  -- Unterprogrammen, Attributen
   ...  -- Definition von:
   ...  -- Unterprogrammen, Attributen,
   ...  -- Konfigurationen
   ...
   ...  -- VHDL'93: Groups, Shared Variables
   ...
BEGIN
   ...
   ...  -- nebenlaeufige Anweisungen
   ...  -- zur strukturalen Modellierung
   ...  -- und Verhaltensmodellierung
   ...
END [ARCHITECTURE] [arch_name];
```

Die Wiederholung des Schlüsselwortes `ARCHITECTURE` in der `END`-Anweisung ist mit ✓**93** optional möglich.

Im Unterschied zu Programmiersprachen, wie z.B. "C", hat VHDL die Aufgabe, neben rein sequentiellen Funktionsabläufen auch Hardware zu beschreiben. Die in einer Hardwareeinheit enthaltenen Funktionsmodule arbeiten aber nicht ausschließlich sequentiell, sondern parallel,

d.h. die einzelnen Funktionen laufen nicht immer nacheinander, sondern eventuell auch gleichzeitig ab.

Um diese Eigenschaft zu beschreiben, wird das Konzept der parallelen (nebenläufigen, oder auch "concurrent") Anweisungen verwendet, die im Idealfall gleichzeitig bearbeitet werden. Da die Simulation eines VHDL-Modells in der Regel jedoch auf **einem** Prozessor abläuft, ist dies nicht möglich. Deshalb wurde ein spezieller Simulationsalgorithmus entwickelt, der ein "quasi paralleles", d.h. für den Benutzer nicht direkt sichtbares, sequentielles Abarbeiten der nebenläufigen Befehle gestattet. Dieser Ablauf wird in Kapitel 8 erläutert.

Aus oben genanntem Grund sind die VHDL-Anweisungen in zwei Kategorien einzuteilen: sequentielle und nebenläufige Anweisungen.

Innerhalb des Anweisungsteils einer Architektur sind alle Sprachkonstrukte nebenläufig. **Nebenläufige Anweisungen** sind z.B. die Instantiierung von Komponenten, die `BLOCK`- und `GENERATE`-Anweisungen sowie insbesondere auch sämtliche Prozesse:

`... <= ...` (Signalzuweisung)
`ASSERT`-Anweisung
`BLOCK`-Anweisung
Komponenteninstantiierung
`GENERATE`-Anweisung
`PROCESS`-Anweisung
Prozeduraufruf

Sequentielle Anweisungen entsprechen den von Programmiersprachen her bekannten Konstrukten. Sie dürfen nur innerhalb von Prozessen, Funktionen oder Prozeduren stehen:

`... <= ...` (Signalzuweisung)
`... := ...` (Variablenzuweis.)
`ASSERT`-Anweisung
`CASE`-Anweisung
`EXIT`-Anweisung
`IF`-Anweisung
`LOOP`-Anweisung
`NEXT`-Anweisung
`NULL`-Anweisung
Prozeduraufruf
`REPORT`-Anweisung ✓93
`RETURN`-Anweisung
`WAIT`-Anweisung

4.4 Konfiguration (Configuration)

Die Design-Einheit Konfiguration dient zur Beschreibung der Konfigurationsdaten eines VHDL-Modells. Zunächst wird darin festgelegt, welche Architektur zu verwenden ist. Bei strukturalen Beschreibungen kann außerdem angegeben werden, aus welchen Bibliotheken die einzelnen Submodule entnommen werden, wie sie eingesetzt (verdrahtet) werden und welche Parameterwerte (Generics) für die Submodule gelten. Eine Entity kann mehrere Konfigurationen besitzen.

In der Konfiguration wird zwischen deklarativen und den eigentlichen Konfigurationsanweisungen unterschieden. Die Konfigurationsanweisungen beschreiben - gegebenenfalls hierarchisch - die Parameter und Instanzen der verwendeten Architektur.

Die Rahmensyntax der Design-Einheit Konfiguration lautet wie folgt:

```
CONFIGURATION conf_name OF entity_name IS
   ...
   ...  -- USE- Anweisungen und
   ...  -- Attributzuweisungen,
   ...  -- Konfigurationsanweisungen
   ...
END  [CONFIGURATION]  [conf_name] ;
```

Die optionale Wiederholung des Schlüsselwortes `CONFIGURATION` ist wieder nur in ✓**93** gestattet.

Da die Sprache VHDL in hohem Maße die Wiederverwendung existierender Modelle unterstützen soll, bietet sie eine Vielzahl an Konfigurationsmöglichkeiten. Aus diesem Grund wird in Kapitel 7 des Teils B näher auf die Details dieser Design-Einheit eingegangen.

4.5 Package

Packages dienen dazu, häufig benötigte Datentypen, Komponenten, Objekte, etc. einmalig zu deklarieren. Diese Deklarationen können dann in verschiedenen VHDL-Modellen verwendet werden. Packages eignen sich insbesondere, um globale Informationen innerhalb eines komplexen Entwurfs oder innerhalb eines Projektteams einmalig und

damit widerspruchsfrei festzulegen. Typische Anwendungen sind Packages, die einen bestimmten Logiktyp (z.B. vierwertige Logik) und die zugehörigen Operatoren beschreiben. Andere Packages könnten beispielsweise eine Sammlung von mathematischen Funktionen (`sin(x)`, `cos(x)`, etc.) enthalten.

VHDL unterscheidet zwischen **Package** und **Package Body**, die beide eigenständig sind und auch getrennt compiliert werden. Ursache für diese Vereinbarung sind die Abhängigkeiten beim Compilieren der Design-Einheiten. Da die anderen Design-Einheiten auf den verwendeten Packages aufbauen, müßten bei der Änderung eines Unterprogramms im Package alle vom Package abhängigen Design-Einheiten neu compiliert werden. Durch die Trennung von Deklaration (enthalten im Package) und Funktionalität bzw. Definition (enthalten im Package Body) kann dies vermieden werden. Entities, Architectures und Configurations basieren nur auf den Deklarationen des Packages. Änderungen im Package Body erfordern damit kein Neu-Compilieren der übrigen Design-Einheiten. Dieses Konzept, das auch als "deferring" bezeichnet wird, unterstützt die Änderungsfreundlichkeit der VHDL-Modelle ("deferring" steht für "Verschiebung" der Implementierung in den Package Body).

Die Syntax der Design-Einheit **Package** lautet wie folgt:

```
PACKAGE pack_name IS
   ...
   ...  -- USE-Anweisungen, Disconnections
   ...  -- Deklarationen von:
   ...  -- Typen und Untertypen,
   ...  -- Aliases, Konstanten,
   ...  -- Signalen, Files, Komponenten,
   ...  -- Unterprogrammen, Attributen
   ...  -- Definition von:
   ...  -- Attributen
   ...
   ...  -- VHDL'93: Groups, Shared Variables
   ...
END [PACKAGE] [pack_name] ;
```

Das Schlüsselwort `PACKAGE` kann nur in ✓**93** wiederholt werden.

Der **Package Body** kann neben der Definition von Unterprogrammen auch spezielle Deklarationsanweisungen enthalten. Der Zusammenhang mit dem Package wird durch den identischen Namen (`pack_name`) hergestellt. Der Package Body weist folgende Struktur auf:

```
PACKAGE BODY pack_name IS
   ...
   ...  -- Deklarationen von: Typen und
   ...  -- Untertypen, Aliases, Konstanten,
   ...  -- Files, Unterprogrammen
   ...  -- Definition von: Unterprogrammen
   ...  -- USE-Anweisungen
   ...
END [PACKAGE BODY] [pack_name] ;
```

Die optionale Wiederholung der Schlüsselworte `PACKAGE BODY` ist wiederum nur in ✓**93** möglich.

```
PACKAGE fft_projekt IS
   TYPE tristate IS ('0', '1', 'Z');
   CONSTANT standard_delay : time;    -- ohne Wertangabe!
END fft_projekt;
```

```
PACKAGE BODY fft_projekt IS
   CONSTANT standard_delay : time := 2 ns;
   -- Wiederholung der Deklaration,
   -- Wertangabe aber nur im Package Body: "deferring"
END fft_projekt;
```

4.6 Abhängigkeiten beim Compilieren

Die bei der Erläuterung der Packages bereits angeklungenen Abhängigkeiten beim Compilieren der Design-Einheiten werden in Abb. B-3 illustriert. Beispielsweise müssen nach einer Änderung in einer Entity auch alle zugehörigen Architekturen und Konfigurationen neu übersetzt werden. Eine Änderung im Package Body erfordert dagegen nur das erneute Compilieren dieser Design-Einheit.

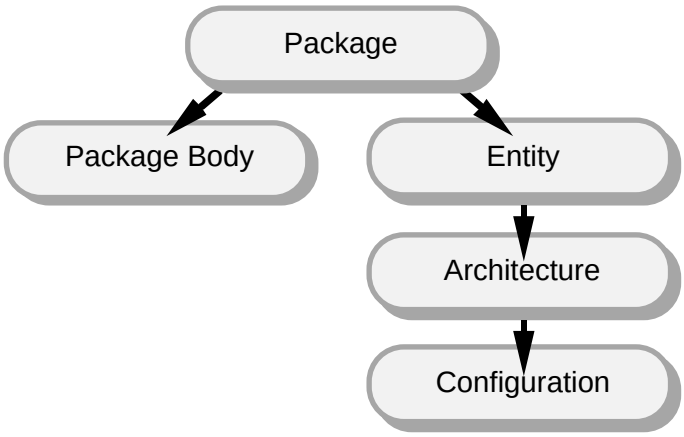

Abb. B-3: Abhängigkeiten beim Compilieren von VHDL-Modellen

5 Strukturale Modellierung

Strukturale Modellierung bedeutet im allgemeinen die Verwendung (= Instantiierung) und das Verdrahten von Komponenten in Form einer Netzliste. VHDL bedient sich dazu einer dreistufigen Vorgehensweise, die zwar viele Freiheitsgrade bietet, für den Anfänger jedoch sehr unübersichtlich ist. Zur Einführung in die strukturale Modellierung soll deshalb ein RS-Flip-Flop betrachtet werden, das aus zwei gleichartigen NAND2-Gattern aufgebaut ist (siehe Abb. B-4).

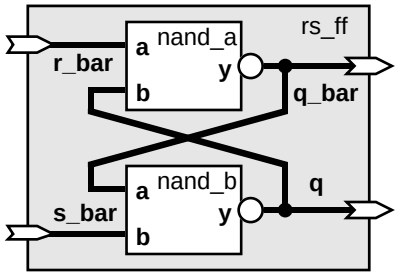

Abb. B-4: Struktur eine RS-Flip-Flops

Die Schnittstelle des RS-Flip-Flops hat folgendes Aussehen:

```
ENTITY rs_ff IS
   PORT (r_bar, s_bar : IN bit := '0';
         q, q_bar     : INOUT bit);
   -- Ports als INOUT, da sie auch gelesen werden muessen
END rs_ff;
```

Für die strukturale Modellierung des Flip-Flops mit der Sprache VHDL wird eine Komponentendeklaration, zwei Komponenteninstantiierungen und eine Komponentenkonfiguration benötigt:

```
ARCHITECTURE structural OF rs_ff IS
   COMPONENT nand2_socket      -- Komponentendeklaration
      PORT (a, b : IN bit; y : OUT bit);
   END COMPONENT;
BEGIN
   nand_a : nand2_socket       -- Komponenteninstantiierung
      PORT MAP (a => r_bar, b => q, y => q_bar);
   nand_b : nand2_socket       -- Komponenteninstantiierung
      PORT MAP (a => q_bar, b => s_bar, y => q);
END structural;
```

```
CONFIGURATION rs_ff_config OF rs_ff IS
   FOR structural
      FOR ALL : nand2_socket   -- Komponentenkonfiguration
         USE ENTITY work.nand2 (behavioral);
      END FOR;
   END FOR;
END rs_ff_config;
```

Für ein besseres Verständnis kann man sich diese dreistufige VHDL-Syntax mit ICs und zugehörigen Sockeln bzw. Sockeltypen vorstellen. Das strukturale Modell des Flip-Flops würde also eine Leiterplatte mit zwei gesockelten NAND2-Gattern nachbilden.

- **Komponentendeklaration**
  Als erster Schritt wird ein Prototyp des Komponentensockels im Deklarationsteil der Architektur, im Deklarationsteil eines Blocks oder im Package deklariert (hier: `nand2_socket`).
- **Komponenteninstantiierung**
  In der Architektur / im Block werden Instanzen dieses Sockeltyps gebildet und verdrahtet (hier: `nand_a` und `nand_b`).
- **Komponentenkonfiguration**
  In der Konfiguration schließlich wird festgelegt, welches bereits compilierte VHDL-Modell für die jeweiligen Instanzen der Komponenten einer strukturalen Architektur verwendet wird; mit anderen Worten: welcher IC-Typ in die einzelnen Sockel eingesetzt wird. Im Beispiel wird für alle Instanzen der Komponente `nand2_socket` die Entity `nand2` aus der Bibliothek `work` verwendet (in Klammern: zugehörige Architektur).

5.1 Komponentendeklaration und -instantiierung

In einer **Komponentendeklaration** werden im allgemeinen neben den Ports (entsprechend den Pins des Sockeltyps) auch die zu übergebenden Parameter (Generics) aufgeführt. Die Komponentendeklaration ist somit ein Abbild der Entity des einzusetzenden Modells (ICs).

```
COMPONENT comp_name
 [ GENERIC (
      param_1 {, param_n } :    type_name
              [ := def_value ]
   { ; further_generic_declarations  } );]

 [ PORT (
   {  port_1  {, port_n }   : IN type_name
              [ := def_value ] }
   { ; port_declarations_of_mode_OUT }
   { ; port_declarations_of_mode_INOUT }
   { ; port_declarations_of_mode_BUFFER } );]
END COMPONENT ;
```

Auch hier greift die Vereinheitlichung der Rahmensyntax von ✓**93**, so daß sich folgende Syntax-Alternative ergibt:

```
COMPONENT comp_name [IS]
   ...
END COMPONENT [comp_name] ;
```

Einige Beispiele von Komponentendeklarationen:

```
COMPONENT inv
  GENERIC (tpd_lh, tpd_hl : time := 0.8 ns) ;
  PORT (a : IN bit; y : OUT bit) ;
END COMPONENT ;

COMPONENT or2
  GENERIC (tpd_lh : time := 1.5 ns; tpd_hl : time := 1 ns) ;
  PORT (a,b : IN bit; y : OUT bit) ;
END COMPONENT ;
```



```
COMPONENT and2
  GENERIC (tpd_lh : time := 1 ns; tpd_hl : time := 1.5 ns) ;
  PORT (a,b : IN bit; y : OUT bit) ;
END COMPONENT ;

COMPONENT and3
  GENERIC (tpd_lh : time := 1 ns; tpd_hl : time := 1.8 ns) ;
  PORT (a,b,c : IN bit; y : OUT bit) ;
END COMPONENT ;
```

Die eigentliche Netzliste wird erst durch das **Instantiieren** dieser Sokkeltypen und gleichzeitiges Verdrahten durch Zuweisung von Signalnamen an die Ports erzeugt. Dabei können auch Parameter übergeben werden.

Jede Komponente erhält bei der Instantiierung einen eigenen Referenznamen (`inst_name`):

```
inst_name : comp_name
  [ GENERIC MAP ( ... ) ]  --  Generic-Map-Liste
  [ PORT MAP ( ... )    ]; --  Port-Map-Liste
```

Es existieren verschiedene Möglichkeiten, die Ports und Parameter von Komponente und zugehöriger Instanz zuzuordnen. Um die dafür erforderliche Syntax zu verstehen, sind zunächst folgende Ebenen zu unterscheiden:

- Ports und Generics in der Schnittstellenbeschreibung (Entity) der zu instantiierenden Komponente (sog. "**formals**"),
- Ports und Generics in der Komponentendeklaration (sog. "**locals**") und
- lokale Signale innerhalb der Architektur oder die von der Architektur zu übergebenden Parameterwerte (sog. "**actuals**").

Bei der Komponenteninstantiierung innerhalb von strukturalen Architekturen werden **locals** mit **actuals** verbunden. Angaben von Parameterwerten in der Generic-Map überschreiben dabei die Defaultwerte in der Komponentendeklaration.

Generic-Map und Port-Map bestehen:

- aus einer durch Kommata getrennten Liste von unmittelbar aufeinanderfolgenden Signalnamen (actuals) bzw. Parameterwerten, wobei die Zuweisung in der gleichen Reihenfolge wie in der Komponentendeklaration (locals) erfolgt ("positional association"):

```
actual_1 {, actual_n}
```

- oder aus einer durch Kommata getrennten Liste von expliziten Zuweisungen in beliebiger Reihenfolge ("named association"):

```
      local_1   =>  actual_1
  {,  local_n   =>  actual_n}
```

- oder aus einer Kombination beider Möglichkeiten, wobei die zweite Variante der ersten nachfolgen muß.

Das folgende Beispiel für eine strukturale Architektur greift die oben gezeigten Komponentendeklarationen auf. Es handelt sich um ein 3-2-AND-OR-INVERT-Komplexgatter mit folgendem Schaltbild:

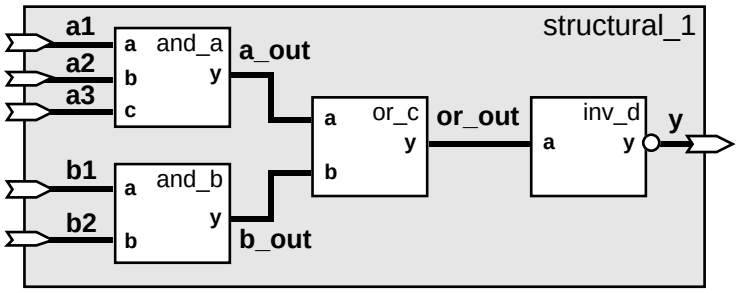

Abb. B-5: Schaltbild eines 3-2-AND-OR-INVERT-Komplexgatters

```
ENTITY aoi IS
  PORT ( a1, a2, a3, b1, b2 : IN bit;
         y : OUT bit ) ;
END aoi ;
```



```
ARCHITECTURE structural_1 OF aoi IS
  SIGNAL a_out, b_out, or_out : bit ;  -- interne Signale
  ...
  ...           -- Komponentendeklarationen wie oben
  ...
BEGIN
---- verschiedene Varianten der Instantiierung: ------------
  -- position. association, tpd_lh = 1.2 ns, tpd_hl = 2.4 ns
  and_a : and3 GENERIC MAP (1.2 ns,2.4 ns)
               PORT MAP (a1,a2,a3,a_out) ;
  -- named association
  and_b : and2 GENERIC MAP (tpd_hl=>1.9 ns,tpd_lh=>1.1 ns)
               PORT MAP (b=>b1,y=>b_out,a=>b2) ;
  -- ohne Generic-Map: Default Generics: 1.5 ns bzw. 1.0 ns
  or_c  : or2  PORT MAP (a_out,y=>or_out,b=>b_out) ;
  -- unvollst. Generic-Map: tpd_hl = Defaultwert: 0.8 ns
  inv_d : inv  GENERIC MAP (0.7 ns) PORT MAP (or_out,y) ;
END structural_1 ;
```

Zusätzlich zu diesem Beispiel sei noch auf die einfachere Architektur `structural` des Halbaddierers aus Teil A zur Verdeutlichung der strukturalen Modellierung verwiesen.

Erlaubt ist in VHDL auch das Schlüsselwort `OPEN` als actual zur Kennzeichnung nicht angeschlossener Ports (nicht verdrahteter Sokkelpins). Für solche Ports wird bei Simulationsbeginn der Defaultwert angenommen, der in der Komponentendeklaration angegeben ist.

Eine weitere Vereinbarung für Eingangsports in der Syntax ✓**93** erlaubt die Angabe von Ausdrücken als actuals.

Ein Beispiel verdeutlicht die Möglichkeiten von nicht angeschlossenen Ports bzw. die Zuweisung von Ausdrücken nach ✓**93**. Es handelt sich um das obige Komplexgatter mit veränderter Architektur. Der Inverter und das ODER-Gatter sind hier zu einem NOR-Gatter zusammengefaßt.

```
ARCHITECTURE structural_2 OF aoi IS  -- !!! VHDL'93 !!!
  SIGNAL a_out, b_out : bit ;        -- interne Signale
  COMPONENT nor3                     -- invertierendes OR3
    PORT (a,b,c : IN bit := '0' ; y : OUT bit) ;
  END COMPONENT nor3;
  COMPONENT and3                     -- nur 3-fach AND
    PORT (a,b,c : IN bit := '0' ; y : OUT bit) ;
  END COMPONENT ;
BEGIN
  and_a : and3 PORT MAP (a1,a2,a3,a_out) ;
  and_b : and3 PORT MAP (a=>b2,b=>b1,c=>'1',y=>b_out) ;
         -- VHDL'93: Port "c" wird konstant auf '1' gelegt
  nor_c : nor3 PORT MAP (OPEN,a_out,y=>y,b=>b_out) ;
         -- VHDL'87/93: Port "a" bleibt unangeschlossen: '0'
END structural_2 ;
```

Mit ✓**93** wurde eine Möglichkeit geschaffen, bereits bei der Komponenteninstantiierung anzugeben, welches Modell zu verwenden ist. Diese sog. "direkte Instantiierung" kann mit dem Einlöten eines ICs in die Leiterplatte ohne Sockel verglichen werden. Bei einmalig verwendeten Komponenten hat dies durchaus Vorteile, mehrfach verwendete Komponenten hingegen sollten mit der herkömmlichen Methode instantiiert werden.

Die direkte Instantiierung ✓**93** benötigt weder eine Komponentendeklaration noch eine Konfiguration. Sie hat folgende Syntax:

```
inst_name : CONFIGURATION config_name
 [ GENERIC MAP ( ... ) ]  -- Generic-Map-Liste
 [ PORT MAP ( ... )    ]; -- Port-Map Liste

inst_name : ENTITY entity_name [(arch_name)]
 [ GENERIC MAP ( ... ) ]  -- Generic-Map-Liste
 [ PORT MAP ( ... )    ]; -- Port-Map-Liste
```

Wird im zweiten Fall keine Architektur angegeben, so wird die Entity ohne Architektur instantiiert. Dies ist u.a. dann sinnvoll, wenn die Funktion des Modells nur im Anweisungsteil der Entity (durch passive Prozeduren, Prozesse und Assertions) definiert ist.

Natürlich ist die herkömmliche Instantiierung in ✓93 weiterhin erlaubt. Das Schlüsselwort COMPONENT kann nun hinzugefügt werden:

```
inst_name : [ COMPONENT ] comp_name
 [ GENERIC MAP ( ... ) ]  -- Generic-Map-Liste
 [ PORT MAP ( ... )    ]; -- Port-Map Liste
```

Als Beispiel für die direkte Instantiierung in ✓93 soll wieder das Komplexgatter dienen. Man beachte, daß für diese Version weder Komponentendeklarationen noch eine Konfiguration benötigt werden:

```
ARCHITECTURE structural_3 OF aoi IS  -- !! VHDL'93-Syntax !!
  SIGNAL a_out, b_out : bit ;        -- interne Signale
BEGIN
  and_a : ENTITY work.and3 (behavioral)
          PORT MAP (a1,a2,a3,a_out) ;
  and_b : ENTITY work.and2 (behavioral)
          PORT MAP (b1,b2,b_out) ;
  nor_c : CONFIGURATION work.nor2_config
          PORT MAP (a_out,b_out,y) ;
END structural_3 ;
```

5.2 BLOCK-Anweisung

Die BLOCK-Anweisung dient zur Gliederung eines Modells, ohne eine Modellhierarchie mit instantiierten Untermodellen einführen zu müssen. Ähnlich einer Architektur können in einem Block lokale Deklarationen getroffen werden. Ein Block kann sogar wie eine eigenständige Einheit mit Generics und Ports verwaltet werden. Auch ist es möglich, daß innerhalb eines Blocks wieder eine Partitionierung in Blöcke stattfindet, so daß sich prinzipiell in einer Architektur beliebig komplexe Strukturen aufbauen lassen.

Die Syntax von Blöcken hat folgende Form:

```
block_name : BLOCK   [IS]
     ...
     ...   -- USE-Anweisungen, Disconnections
     ...   -- Generics und Generic-Map
     ...   -- Ports und Port-Map
```

```
      ...   -- Deklaration von:
      ...   -- Typen und Untertypen,
      ...   -- Aliases, Konstanten,
      ...   -- Signalen, Files, Komponenten,
      ...   -- Unterprogrammen, Attributen
      ...   -- Definition von:
      ...   -- Unterprogrammen, Attributen
      ...   -- Konfigurationen
      ...
      ...   -- VHDL'93: Groups, Shared Variables
      ...
  BEGIN
      ...
      ...   -- nebenlaeufige Anweisungen
      ...   -- zur strukturalen Modellierung
      ...   -- und Verhaltensmodellierung
      ...
  END BLOCK [block_name] ;
```

Die Verwendung des Schlüsselwortes `IS` in der `BLOCK`-Anweisung ist nur in ✓**93** möglich.

Als Beispiel für die Block-Anweisung dient erneut das AOI-Komplexgatter, das nun in zwei Stufen partitioniert wurde. Die beiden Stufen sind als Blöcke realisiert (siehe Abb. B-6). Der zweite Block hat zur Veranschaulichung der `BLOCK`-Syntax eigene Ports mit einer entsprechenden Port-Map.

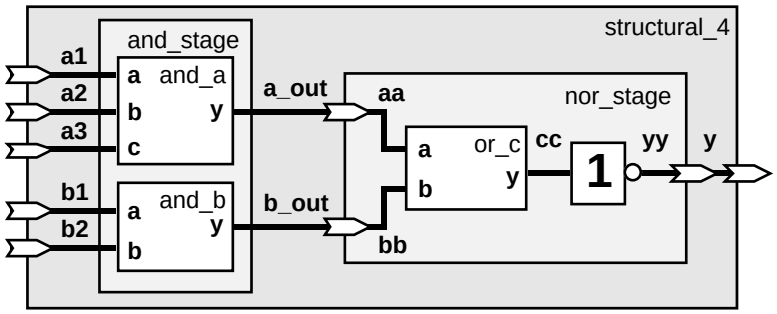

Abb. B-6: Strukturierung des Komplexgatters durch Blöcke


```
ARCHITECTURE structural_4 OF aoi IS
  SIGNAL a_out, b_out : bit ;        -- interne Signale
BEGIN
  and_stage : BLOCK
    COMPONENT and2
      PORT (a,b : IN bit; y : OUT bit) ;
    END COMPONENT ;
    COMPONENT and3
      PORT (a,b,c : IN bit; y : OUT bit) ;
    END COMPONENT ;
  BEGIN
     and_a : and3 PORT MAP (a1,a2,a3,a_out);
     and_b : and2 PORT MAP (b1,b2,b_out);
  END BLOCK and_stage;

  nor_stage : BLOCK
    PORT (aa,bb : IN bit; yy : OUT bit) ;
    PORT MAP (aa=>a_out, bb=>b_out, yy=>y);
    SIGNAL cc : bit;            -- block-internes Signal
    COMPONENT or2               -- nicht invertierend !
      PORT (a,b : IN bit; y : OUT bit) ;
    END COMPONENT ;
  BEGIN
    or_c : or2 PORT MAP (a=>aa,b=>bb,y=>cc) ;
    yy <= not cc;               -- Invertierung
  END BLOCK nor_stage ;
END structural_4 ;
```

5.3 GENERATE-Anweisung

In vielen Fällen treten in elektronischen Systemen regelmäßige Strukturen auf, bei denen ein Modul aus mehreren gleichartigen Submodulen besteht. Eine Darstellung solcher Strukturen mit den bisher eingeführten Beschreibungsmitteln der Sprache VHDL würde sehr umfangreich werden. Mit Hilfe der GENERATE-Anweisung lassen sich jedoch regelmäßige Strukturen einfach und auch flexibel gestalten.

Abhängig von einer Bedingung (*condition*) oder einem diskreten Bereich (*disc_range*) wird eine Reihe von nebenläufigen Anweisungen ein- oder mehrfach ausgeführt. Der diskrete Bereich kann wie gewohnt durch zwei diskrete Bereichsgrenzen und eines der beiden

Schlüsselwörter `TO` oder `DOWNTO` bzw. mit einem diskreten Wertebereich mit Hilfe des Attributs `RANGE` angegeben werden:

```
gen_name : IF condition GENERATE
    ...
    ...  -- Nebenlaeufige Anweisungen
    ...
END GENERATE [gen_name] ;

gen_name : FOR var_name IN disc_range GENERATE
    ...
    ...  -- Nebenlaeufige Anweisungen
    ...
END GENERATE [gen_name] ;
```

Bei der zweiten Beschreibungsvariante wird eine Laufvariable für den Index durch die Rahmensyntax automatisch deklariert; sie kann innerhalb der Anweisung verwendet werden. Typische Anwendungsfälle für `GENERATE`-Anweisungen sind aus mehreren Speicherelementen aufgebaute Register. Hierzu ein Beispiel (Abb. B-7):

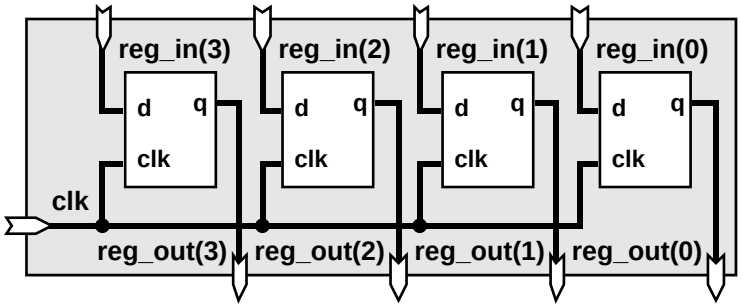

Abb. B-7: n-Bit Register (n=4)

```
ENTITY n_bit_register IS
  GENERIC (n : IN positive := 4) ;
  PORT (clk     : IN  bit ;
        reg_in  : IN  bit_vector(n-1 DOWNTO 0) ;
        reg_out : OUT bit_vector(n-1 DOWNTO 0) ) ;
END n_bit_register ;
```

```
ARCHITECTURE structural OF n_bit_register IS
  COMPONENT d_ff_socket
    PORT (d, clk : IN bit ; q : OUT bit) ;
  END COMPONENT ;
BEGIN
  reg : FOR i IN n-1 DOWNTO 0 GENERATE
    d_ff_instance : d_ff_socket
      PORT MAP (reg_in(i),clk, reg_out(i)) ;
  END GENERATE ;
END structural ;
```

Die Länge dieses Registers ist in der Beschreibung nicht fixiert. Sie kann erst beim Konfigurieren des Modells über den Parameter n festgelegt werden. Damit lassen sich von diesem Modell auch mehrere Instanzen unterschiedlicher Länge erzeugen.

Falls nicht alle Instanzen nach dem gleichen Schema verdrahtet sind, müssen in der GENERATE-Anweisung Bedingungen eingesetzt werden. Als Beispiel dient hier ein Schieberegister beliebiger Länge, bei dem das erste und letzte Modul eine spezielle Verdrahtung besitzen (Abb. B-8):

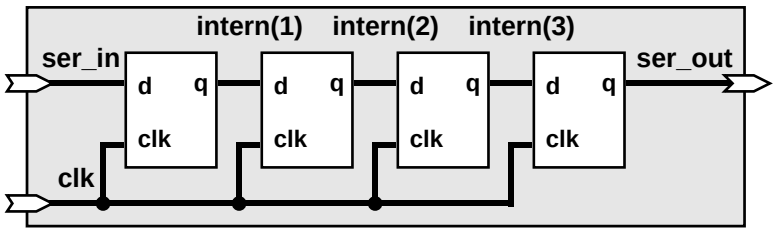

Abb. B-8: n-Bit Schieberegister (n=4)

```
ENTITY shift_register IS
  GENERIC (n: IN positive RANGE 2 TO 64 := 4);
  PORT (clk, ser_in : IN  bit ;
        ser_out     : OUT bit ) ;
END shift_register ;
```

```
ARCHITECTURE structural OF shift_register IS
  SIGNAL intern : bit_vector(1 TO n-1) ;
  COMPONENT d_ff_socket
    PORT (d, clk : IN bit ; q : OUT bit) ;
  END COMPONENT ;
BEGIN
  reg : FOR i IN n DOWNTO 1 GENERATE
-- erstes D-FF: mit Eingang verdrahtet ---------------------
    reg_begin : IF i = 1 GENERATE
      d_ff_begin : d_ff_socket
        PORT MAP (ser_in,clk,intern(i)) ;
    END GENERATE ;
-- mittlere D-FFs ------------------------------------------
    reg_middle : IF i > 1 AND i < n GENERATE
      d_ff_middle : d_ff_socket
        PORT MAP (intern(i-1),clk,intern(i)) ;
    END GENERATE ;
-- letztes D-FF: mit Ausgang verdrahtet --------------------
    reg_end : IF i = n GENERATE
      d_ff_end : d_ff_socket
        PORT MAP (intern(i-1),clk,ser_out) ;
    END GENERATE ;
  END GENERATE ;
END structural ;
```

Mit ✓**93** wird ein optionaler Deklarationsteil für die GENERATE-Anweisung ermöglicht. Er entspricht dem Deklarationsteil von BLOCK oder ARCHITECTURE:

```
gen_name : FOR var_name IN disc_range GENERATE
[   ...
    ...  Deklarationsteil
    ...
BEGIN ]
    ...
END GENERATE [gen_name] ;
```

Die gleiche Erweiterungsmöglichkeit von ✓**93** gilt auch für die IF-Variante der GENERATE-Anweisung.

6 Verhaltensmodellierung

Im vorangegangenen Kapitel wurde gezeigt, wie sich hierarchische Strukturen, wie z.B. die in Abb. B-9 gezeigte, modellieren lassen. Die unterste Ebene in den einzelnen Zweigen des Strukturbaumes jedoch kann nicht struktural beschrieben werden, da diese Modelle nicht weiter in Sub-Modelle unterteilt sind. Für diese Modelle stellt VHDL zahlreiche Konstrukte zur Verfügung, mit denen sich das Modellverhalten nachbilden läßt.

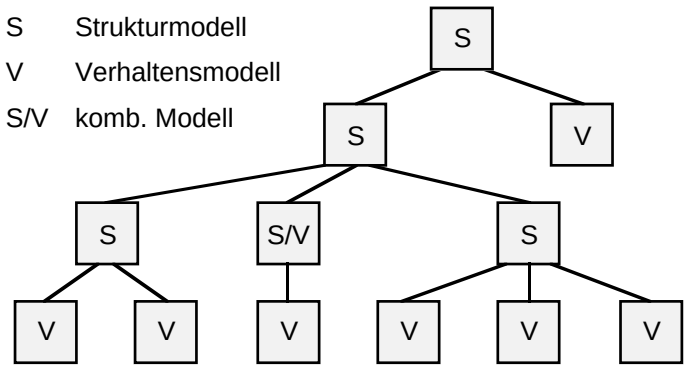

Abb. B-9: Hierarchischer Modellaufbau

Als einführendes Beispiel zur Verhaltensmodellierung soll das nachstehende Modell eines Zählers dienen, der bei steigenden Taktflanken zyklisch von 0 bis 4 zählt, falls der `enable`-Eingang den Wert `'1'` besitzt. Mit einem low-aktiven `reset`-Signal kann der Zähler asynchron zurückgesetzt werden.

```
ENTITY count5 IS
  PORT (clk, enable, reset : IN  bit;
        q                  : OUT bit_vector (2 DOWNTO 0));
END count5;
```

```
ARCHITECTURE behavioral OF count5 IS
  SIGNAL state : integer RANGE 0 TO 4 := 0; -- Zaehlerstand
BEGIN
  count : PROCESS (clk, reset)     -- Prozess zum Zaehlen
  BEGIN
    IF reset = '0' THEN
      state <= 0;                  -- Ruecksetzen
    ELSIF (enable = '1' AND clk'EVENT AND clk = '1') THEN
      IF state < 4 THEN
        state <= state + 1;        -- Hochzaehlen
      ELSE
        state <= 0;                -- Ueberlauf
      END IF;
    END IF;
  END PROCESS count;
-- nebenlaeufige Signalzuweisung an den Ausgang q ----------
  q <= ('0','0','0') WHEN state = 0 ELSE
       ('0','0','1') WHEN state = 1 ELSE
       ('0','1','0') WHEN state = 2 ELSE
       ('0','1','1') WHEN state = 3 ELSE
       ('1','0','0');
END behavioral;
```

Wie aus dem Beispiel hervorgeht, beschreibt die Verhaltensmodellierung, wie die Eingangs- in Ausgangssignale überführt werden. Eingesetzt werden bei einer Verhaltensmodellierung u.a. verschiedene Operatoren (`AND`, `<`, `=`), vordefinierte Attribute (`EVENT`), Signalzuweisungen (`q <= ...`) oder `IF-ELSIF-ELSE`-Anweisungen. Selbstverständlich können innerhalb eines VHDL-Modells auch Konstrukte der strukturalen und der Verhaltensmodellierung gleichzeitig verwendet werden.

In den folgenden Abschnitten werden die VHDL-Sprachelemente vorgestellt, die vorrangig der Verhaltensmodellierung dienen.

Die nächsten beiden Abschnitte erläutern zunächst die Details der Anwendung von Operatoren und Attributen. Der "eilige" Leser kann diese beiden Abschnitte, die eher zum Nachschlagen dienen, überspringen.

6.1 Operatoren

Operatoren verknüpfen zwei Operanden bzw. verändern einen Operanden. Das Ergebnis kann wieder als Operand verwendet werden. Werden in einer Anweisung mehrere Operatoren ohne Klammerung eingesetzt, so ist die Priorität der Operatoren bei der Abarbeitung relevant. Bei gleicher Priorität werden die Operatoren in der Reihenfolge ihres Auftretens abgearbeitet. Die Anwendung der Operatoren wird in späteren Beispielen verdeutlicht.

Abb. B-10 gibt die Priorität der einzelnen Operatoren untereinander an:

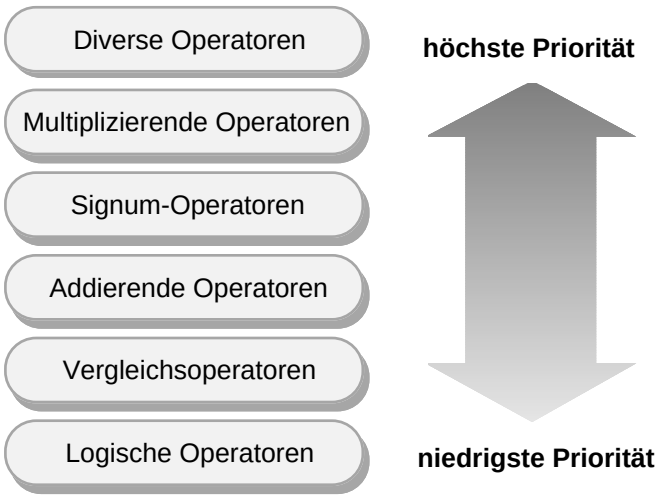

Abb. B-10: Prioritäten von Operatoren

6.1.1 Logische Operatoren

Logische Operatoren wirken auf Einzelobjekte oder Vektoren vom Typ `bit` oder `boolean`; das Ergebnis von logischen Operatoren ist entweder gleich `'1'` bzw. `true` oder gleich `'0'` bzw. `false`. VHDL kennt folgende logische Operatoren:

- `NOT` Negation (nur ein rechtsstehender Operand)
- `AND` UND-Verknüpfung
- `NAND` Negierte UND-Verknüpfung
- `OR` ODER-Verknüpfung
- `NOR` Negierte ODER-Verknüpfung
- `XOR` Exklusiv-ODER-Verknüpfung
- `XNOR` Neg. Exklusiv-ODER-Verknüpfung (nur ✓**93**)

Bei der Anwendung der logischen Operatoren auf Vektoren ist zu beachten, daß die Vektoren gleiche Länge besitzen müssen und daß die Operation elementweise vorgenommen wird. Das Ergebnis erhält den Typ und die Indizierung des linken Operanden.

Sequenzen von mehr als einem Operator sind nur für `AND`, `OR` und `XOR` möglich, da die Reihenfolge der Ausführung hier unerheblich ist. Bei den Operatoren `NAND`, `NOR` und `XNOR` ist dagegen durch Klammerung die Ausführungsreihenfolge festzulegen.

```
a := NOT (x AND y AND z) ;     -- legal: NAND3
b := (x NAND y) NAND z ;       -- legal, aber kein NAND3
c := x NAND y NAND z ;         -- !!! illegal
```

Zu beachten ist weiterhin, daß der logische Operator `NOT` die hohe Priorität der "diversen Operatoren" besitzt.

6.1.2 Vergleichsoperatoren

Operator	Funktion	Typ linker Operand	Typ rechter Operand	Typ Ergebnis
`=`	Prüfung auf Gleichheit der Operanden	alle mögl. Typen außer File-Typen	gleicher Typ wie links	vordef. Typ `boolean`
`/=`	Prüfung auf Ungleichheit der Operanden	alle mögl. Typen außer File-Typen	gleicher Typ wie links	vordef. Typ `boolean`
`<`	Vergleich der beiden Operanden	skal. Typen oder diskr. Vektoren	gleicher Typ wie links	vordef. Typ `boolean`
`<=`	Vergleich der beiden Operanden	skal. Typen oder diskr. Vektoren	gleicher Typ wie links	vordef. Typ `boolean`
`>`	Vergleich der beiden Operanden	skal. Typen oder diskr. Vektoren	gleicher Typ wie links	vordef. Typ `boolean`
`>=`	Vergleich der beiden Operanden	skal. Typen oder diskr. Vektoren	gleicher Typ wie links	vordef. Typ `boolean`

Als diskrete Vektoren werden in der obigen Tabelle die eindimensionalen Feldtypen bezeichnet, die als Elementtyp einen diskreten Typ (ganzzahliger Typ oder Aufzähltyp) besitzen.

Der Gleichheitsoperator für zusammengesetzte Typen (Felder, Records) liefert den Wert `true` zurück, falls jedes Element des linken Operanden ein entsprechendes Element im rechten Operanden besitzt (und umgekehrt) und falls alle entsprechenden Elemente im Wert übereinstimmen. Ansonsten wird der Wert `false` zurückgeliefert.

Entsprechendes gilt für den Ungleichheitsoperator mit vertauschtem Ergebnis.

Beim Arbeiten mit Fließkommazahlen ist darauf zu achten, daß hier die Darstellungs- und Rechengenauigkeit der Hardware in das Ergebnis einfließt. Auf exakte Gleichheit sollte deshalb nie geprüft werden, vielmehr ist die Angabe eines Toleranzbereichs sinnvoll, wie sie etwa in der folgenden Art erfolgen kann:

```
PROCESS
   VARIABLE xyz : real;
BEGIN
   IF (xyz = 0.05) THEN                  -- schlechter Stil
     ...
   END IF;
   ...
   IF ABS(xyz - 0.05) <= 0.000001 THEN   -- besserer Stil
     ...
   END IF;
   ...
END PROCESS;
```

Die Wirkung von Vergleichsoperatoren ist bei den diskreten Vektoren als Operanden folgendermaßen zu verstehen: Die Operation ">" z.B. liefert den Wert `true` zurück, falls gilt:

- der linke Operand ist kein leerer Vektor, der rechte Operand ist leer.
- im anderen Fall (kein leerer rechter Operand) muß gelten:
  - das am weitesten links stehende Element des linken Operanden ist "größer als" das am weitesten links stehende Element des rechten Operanden.
  - falls die beiden am weitesten links stehenden Elemente der beiden Operanden gleich sind, wird der Restvektor in gleicher Weise untersucht.

Die Operation ">=" bedeutet, daß entweder der Fall ">" oder der Fall "=" zutrifft. Ein entsprechend komplementäres Verhalten zeigen die Operatoren "<" und "<=".

6.1.3 Arithmetische Operatoren

6.1.3.1 Addierende Operatoren

Operator	Funktion	Typ linker Operand	Typ rechter Operand	Typ Ergebnis
+	Addition	jeder numerische Typ	gleicher Typ wie links	gleicher Typ wie links
-	Subtraktion	jeder numerische Typ	gleicher Typ wie links	gleicher Typ wie links
&	Zusammenbinden	jeder Vektortyp	gleicher Typ von Vektor wie links	gleicher Typ von Vektor wie links
		jeder Vektortyp	Element vom Typ wie Vektor links	gleicher Typ von Vektor wie links
		Element vom Typ wie Vektor rechts	jeder Vektortyp	gleicher Typ von Vektor wie rechts
		jeder Elementtyp	gleicher Typ wie links	gleicher Typ von Vektor wie Element links

Für den "&"-Operator, das Zusammenbinden von Operanden (im Englischen "concatenation" genannt), gibt es verschiedene Arten von Operandenkombinationen:

- beide Operanden sind Einzelelemente gleichen Typs,
- ein Operand ist ein Vektor (eindimensionales Feld), der andere ein Einzelelement vom Typ der Vektorelemente,
- beide Operanden sind Vektoren gleichen Typs.

Das Ergebnis wird auf jeden Fall ein Vektor mit dem Elementtyp wie die Operanden selbst sein. Die Indizierung des Ergebnisses entspricht der vom linken Operanden fortgeführten Indizierung; Einzelelemente werden dabei als Vektor mit der Länge 1 angesehen.

Die Fortsetzung der Indizierung des linken Vektors kann allerdings zu unerwarteten oder verbotenen Bereichsgrenzen führen. Mit ✓**93** wurde deshalb folgende Vereinbarung getroffen:

- Sind beide Operanden abfallend indiziert (`DOWNTO`), so haben rechter Operand und Ergebnis den gleichen rechten Index,
- sind beide Operanden steigend indiziert (`TO`), so haben linker Operand und Ergebnis den gleichen linken Index.

6.1.3.2 Signum-Operatoren

Signum-Operatoren dienen zur Festlegung des Vorzeichens von Objekten numerischen Typs.

Operator	Funktion	Typ linker Operand	Typ rechter Operand	Typ Ergebnis
+	Identität	-	jeder numerische Typ	gleicher Typ wie Operand
-	Negation	-	jeder numerische Typ	gleicher Typ wie Operand

Die Signum-Operatoren dürfen ungeklammert nicht in Kombination mit multiplizierenden, addierenden und diversen Operatoren stehen (z.B. " `A/(-B)` " anstelle des unerlaubten Ausdrucks " `A/-B` ").

6.1.3.3 Multiplizierende Operatoren

Operator	Funktion	Typ linker Operand	Typ rechter Operand	Typ Ergebnis
`*`	Multiplikation	jeder ganzzahlige Typ	gleicher Typ wie links	gleicher Typ wie links
		jeder Fließkomma-Typ	gleicher Typ wie links	gleicher Typ wie links
		jeder physikalische Typ	vordef. Typ `integer`	gleicher Typ wie links
		jeder physikalische Typ	vordef. Typ `real`	gleicher Typ wie links
		vordef. Typ `integer`	jeder physikalische Typ	gleicher Typ wie rechts
		vordef. Typ `real`	jeder physikalische Typ	gleicher Typ wie rechts
`/`	Division	jeder ganzzahlige Typ	gleicher Typ wie links	gleicher Typ wie links
		jeder Fließkomma-Typ	gleicher Typ wie links	gleicher Typ wie links
		jeder physikalische Typ	vordef. Typ `integer`	gleicher Typ wie links
		jeder physikalische Typ	vordef. Typ `real`	gleicher Typ wie links
		jeder physikalische Typ	gleicher Typ wie links	vordef. Typ `integer`
`MOD`	Modulo-Operator	jeder ganzzahlige Typ	gleicher Typ wie links	gleicher Typ wie links
`REM`	Remainder-Operator	jeder ganzzahlige Typ	gleicher Typ wie links	gleicher Typ wie links

Der Remainder-Operator (`a REM b`) berechnet den Rest bei einer Integerdivision, so daß gilt: `a = (a/b)*b + (a REM b)`

`(a REM b)` hat das Vorzeichen von `a` und einen absoluten Wert, der kleiner als der absolute Wert von `b` ist.

Der Modulo-Operator (`a MOD b`) berechnet den Rest bei einer Integerdivision, so daß gilt: `a = int_value*b + (a MOD b)`

`(a MOD b)` hat das Vorzeichen von `b` und einen absoluten Wert, der kleiner als der absolute Wert von `b` ist.

6.1.3.4 Diverse Operatoren

Die sog. "diversen Operatoren" besitzen die höchste Priorität bei der Abarbeitung. Der Vollständigkeit halber soll hier auch der logische Operator "`NOT`" nochmals erwähnt werden, der aufgrund der Priorität zu dieser Gruppe gehört. Weitere Operatoren dieser Gruppe sind:

Operator	Funktion	Typ linker Operand	Typ rechter Operand	Typ Ergebnis
`**`	Exponentiation	jeder ganzzahlige Typ	vordef. Typ `integer`	gleicher Typ wie links
		jeder Fließkomma-Typ	vordef. Typ `integer`	gleicher Typ wie links
ABS	Absolutwertbildung	-	jeder numerische Typ	gleicher Typ wie Operand

6.1.4 Schiebe- und Rotieroperatoren ✓93

Mit ✓93 wurden neben dem negierten Exklusiv-ODER weitere Operatoren in den Sprachumfang aufgenommen. Es handelt sich um sechs Schiebe- und Rotieroperatoren, die auf Vektoren angewandt werden. Als linker Operand steht der Vektor selbst, als rechter Operand steht jeweils ein Integerwert, um dessen Wert der Vektorinhalt verschoben bzw. rotiert wird. Die Elemente des Vektors müssen vom Typ `bit` oder `boolean` sein.

Operator	Funktion	Typ linker Operand	Typ rechter Operand	Typ Ergebnis
SLL	Schiebe logisch links	Vektor, Elemente: `bit`, `boolean`	vordef. Typ `integer`	gleicher Typ wie links
SRL	Schiebe logisch rechts	Vektor, Elemente: `bit`, `boolean`	vordef. Typ `integer`	gleicher Typ wie links
SLA	Schiebe arithmetisch links	Vektor, Elemente: `bit`, `boolean`	vordef. Typ `integer`	gleicher Typ wie links
SRA	Schiebe arithmetisch rechts	Vektor, Elemente: `bit`, `boolean`	vordef. Typ `integer`	gleicher Typ wie links
ROL	Rotiere links	Vektor, Elemente: `bit`, `boolean`	vordef. Typ `integer`	gleicher Typ wie links
ROR	Rotiere rechts	Vektor, Elemente: `bit`, `boolean`	vordef. Typ `integer`	gleicher Typ wie links

Bei "logischen Schiebeoperationen" um eine Stelle geht jeweils das linke bzw. rechte Vektorelement verloren. Auf der gegenüberliegenden Seite wird der Vektor mit dem Initialisierungswert des Basistyps aufgefüllt. Bei "arithmetischen Schiebeoperationen" wird hierzu das letzte Vektorelement dupliziert. Bei mehrfachen Schiebe- oder Rotieroperationen wird dies entsprechend dem rechten Operanden wiederholt. Negative rechte Operanden entsprechen dem gegensätzlichen Operator mit dem Absolutwert des rechten Operanden ("`b ROL -4`" entspricht "`b ROR 4`").

Abb. B-11 verdeutlicht diese Operationen:

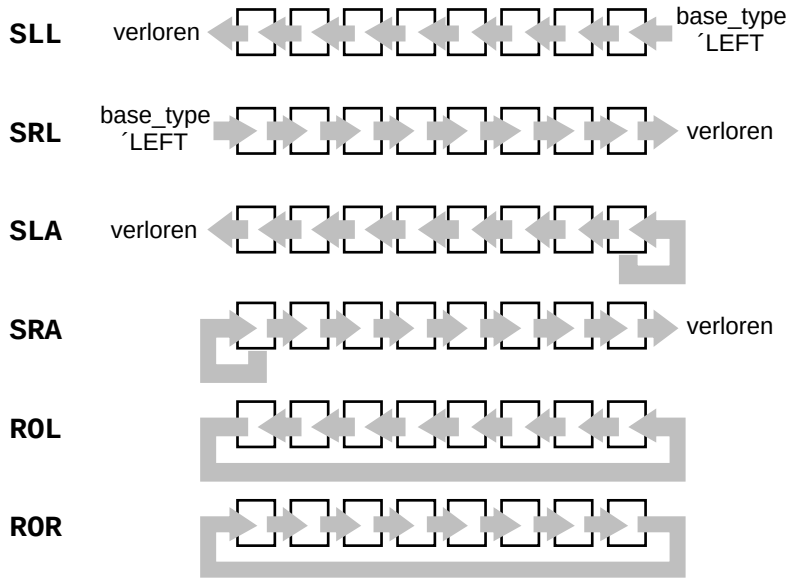

Abb. B-11: Schiebe- und Rotieroperatoren von ✓93

6.2 Attribute

Im folgenden sind die vordefinierten Attribute mit ihrer Funktion aufgelistet. Sie werden nach dem Prefix unterschieden, auf das sie angewandt werden.

6.2.1 Typbezogene Attribute

Typbezogene Attribute liefern Informationen zu diskreten Datentypen, wie z.B. Werte von bestimmten Elementen bei Aufzähltypen.

Name	Funktion
`t'BASE`	liefert den Basistyp des Prefixtyps `t` (nur in Verbindung mit nachfolgendem weiteren Attribut möglich)
`t'LEFT`	liefert die linke Grenze des Prefixtyps `t`
`t'RIGHT`	liefert die rechte Grenze des Prefixtyps `t`
`t'HIGH`	liefert die obere Grenze des Prefixtyps `t`
`t'LOW`	liefert die untere Grenze des Prefixtyps `t`
`t'POS(x)`	liefert die Position (`integer`-Index) des Elements `x` im Prefixtyp `t`
`t'VAL(y)`	liefert den Wert des Elements an Position `y` im Prefixtyp `t` (`y` ist `integer`)
`t'SUCC(x)`	liefert den Nachfolger von `x` im Prefixtyp `t`
`t'PRED(x)`	liefert den Vorgänger von `x` im Prefixtyp `t`
`t'LEFTOF(x)`	liefert das Element links von `x` im Prefixtyp `t`
`t'RIGHTOF(x)`	liefert das Element rechts von `x` im Prefixtyp `t`

Im neuen Standard (✓**93**) wurden einige weitere typbezogene Attribute aufgenommen:

Name	Funktion
`t'ASCENDING`	liefert `true`, falls der Typ `t` eine steigende Indizierung besitzt, ansonsten `false`
`t'IMAGE(x)`	konvertiert den Wert `x` in eine Zeichenkette `t`
`t'VALUE(x)`	konvertiert die Zeichenkette `x` in einen Wert des Typs `t`

Die oben beschriebenen typbezogenen Attribute sollen nun anhand einiger Beispiele erläutert werden:

```
PROCESS
   TYPE    asc_int    IS RANGE 2 TO 8;       -- steigend
   TYPE    desc_int   IS RANGE 8 DOWNTO 2;   -- fallend
   TYPE    colors     IS (white, red, yellow, green, blue);
   SUBTYPE signal_colors IS colors RANGE (red TO green);
   VARIABLE a : integer := 0;
   VARIABLE b : colors  := blue;
   VARIABLE c : boolean := true;
BEGIN
   a := asc_int'LEFT;          -- Ergebnis: 2
   a := asc_int'RIGHT;         -- Ergebnis: 8
   a := asc_int'LOW;           -- Ergebnis: 2
   a := asc_int'HIGH;          -- Ergebnis: 8
   a := desc_int'LEFT;         -- Ergebnis: 8
   a := desc_int'RIGHT;        -- Ergebnis: 2
   a := desc_int'LOW;          -- Ergebnis: 2
   a := desc_int'HIGH;         -- Ergebnis: 8
   a := asc_int'PRED(7);       -- Ergebnis: 6
   a := asc_int'SUCC(4);       -- Ergebnis: 5
   a := asc_int'LEFTOF(7);     -- Ergebnis: 6
   a := asc_int'RIGHTOF(4);    -- Ergebnis: 5
   a := desc_int'PRED(7);      -- Ergebnis: 6
   a := desc_int'SUCC(4);      -- Ergebnis: 5
   a := desc_int'LEFTOF(7);    -- Ergebnis: 8
   a := desc_int'RIGHTOF(4);   -- Ergebnis: 3
   a := asc_int'PRED(2);       -- !! illegal: kein Vorgaenger
   a := asc_int'SUCC(8);       -- !! illegal: kein Nachfolger
   a := asc_int'LEFTOF(15);    -- !! illegal: falscher Index
   b := signal_colors'RIGHT;   -- Ergebnis: green
   b := signal_colors'BASE'RIGHT;     -- Ergebnis: blue
   b := colors'LEFTOF(red);    -- Ergebnis: white
   b := colors'RIGHTOF(red);   -- Ergebnis: yellow
   a := colors'POS(red);       -- Ergebnis: 1 (Aufzaehltypen
                               -- werden von 0 an indiziert)
   b := colors'VAL(2);         -- Ergebnis: yellow
   c := asc_int'ASCENDING;     -- Ergebnis: true  !!! VHDL'93
   c := desc_int'ASCENDING;    -- Ergebnis: false !!! VHDL'93
   b := colors'VALUE("red ");  -- Ergebnis: red   (VHDL'93)
   WAIT;
END PROCESS;
```

Das Ergebnis der Attribute `RIGHT/LEFT` und `RIGHTOF/LEFTOF` hängt also von der Indizierungsrichtung ab. Bei fallender Indizierung sind die Ergebnisse identisch mit den Ergebnissen von `LOW/HIGH`

und PRED/SUCC. Die Aufzähltypen (z.B. colors) besitzen implizit eine steigende Indizierung, so daß beispielsweise RIGHTOF dem Attribut SUCC entspricht.

6.2.2 Feldbezogene Attribute

Bei den feldbezogenen Attributen finden sich viele typbezogene Attribute wieder. Sie werden hier auf eingeschränkte Typen von Feldern und auf konkrete Objekte angewandt.

Name	Funktion
a'LEFT [(n)]	liefert die linke Grenze der n-ten Dimension des Feldes a
a'RIGHT [(n)]	liefert die rechte Grenze der n-ten Dimension des Feldes a
a'LOW [(n)]	liefert die untere Grenze der n-ten Dimension des Feldes a
a'HIGH [(n)]	liefert die obere Grenze der n-ten Dimension des Feldes a
a'LENGTH [(n)]	liefert die Bereichslänge der n-ten Dimension des Feldes a
a'RANGE [(n)]	liefert den Bereich der n-ten Dimension des Feldes a
a'REVERSE _RANGE [(n)]	liefert den Bereich der n-ten Dimension des Feldes a in umgekehrter Reihenfolge

In ✓93 wurde das Attribut ASCENDING ergänzt:

Name	Funktion
a'ASCENDING [(n)]	liefert true, falls das Feld a in der n-ten Dimension eine steigende Indizierung besitzt

Die feldbezogenen Attribute beziehen sich immer auf eine Indizierung. Diese ist bei eindimensionalen Feldern (Vektoren) eindeutig. Die Angabe der Dimension `(n)`, auf die das Attribut angewandt werden soll, ist daher nur bei mehrdimensionalen Feldern relevant. Wird sie weggelassen, so wird das Attribut immer auf die erste Dimension angewandt.

```
PROCESS
   TYPE asc_vec    IS ARRAY (1 TO 5) OF integer;
   TYPE int_matrix IS ARRAY (0 TO 4, 9 DOWNTO 0) OF integer;
   CONSTANT mask        : asc_vec     := (OTHERS => 0);
   VARIABLE ar1         : int_matrix;
   VARIABLE a           : integer := 0;
   VARIABLE b           : boolean;
BEGIN
   a := mask'LEFT;             -- Ergebnis: 1
   a := int_matrix'HIGH(2);    -- Ergebnis: 9
   a := ar1'LEFT(1);           -- Ergebnis: 0
   a := int_matrix'LEFT(1);    -- Ergebnis: 0
   a := ar1'LEFT(2);           -- Ergebnis: 9
   a := ar1'LENGTH(1);         -- Ergebnis: 5
   a := ar1'LENGTH(2);         -- Ergebnis: 10
   b := ar1'ASCENDING(2);      -- VHDL'93, Erg.: false
   b := ar1'ASCENDING(1);      -- VHDL'93, Erg.: true
   WAIT;
END PROCESS;
```

6.2.3 Signalbezogene Attribute

Folgende Attribute werden auf konkrete Objekte der Klasse Signal angewandt:

Name	Funktion
`s'DELAYED [(t)]`	liefert ein auf `s` basierendes Signal, welches um eine Zeit `t` (default: 1 Delta) verzögert ist
`s'STABLE [(t)]`	liefert `true`, falls das Signal `s` eine Zeit `t` (default: 1 Delta) ohne Ereignis war, sonst `false`

`s'QUIET [(t)]`	liefert `true`, falls das Signal `s` eine Zeit `t` (default: 1 Delta) nicht aktiv war, sonst `false`
`s'TRANSACTION`	liefert ein Signal vom Typ `bit`, welches bei jedem Simulationszyklus wechselt, in dem das Signal `s` aktiv ist
`s'EVENT`	liefert `true`, falls beim Signal `s` während des aktuellen Simulationszyklus ein Ereignis auftritt, sonst `false`
`s'ACTIVE`	liefert `true`, falls das Signal `s` während des aktuellen Simulationszyklus aktiv ist, sonst `false`
`s'LAST_EVENT`	liefert die Zeitdifferenz vom aktuellen Simulationszeitpunkt zum letzten Ereignis des Signals `s`
`s'LAST_ACTIVE`	liefert die Zeitdifferenz vom aktuellen Simulationszeitpunkt zum letzten aktiven Zeitpunkt des Signals `s`
`s'LAST_VALUE`	liefert den Wert des Signals `s` vor dem letzten Ereignis

Einige signalbezogene Attribute können mit einem Zeitparameter `(t)` versehen werden, um z.B. beim Attribut `DELAYED` ein um `t` verzögertes Signal zu erhalten. Wird kein Parameter angegeben, so gilt als Defaultwert die minimale Zeit "Delta" (siehe Kapitel 8). Das Signal "`sig1'DELAYED`" ändert z.B. immer ein Delta später seinen Wert als das Signal "`sig1`".

Die Zusammenhänge bei signalbezogenen Attributen sollen in einem kleinen Beispiel verdeutlicht werden. Gegeben sei folgender Signalverlauf:

```
PROCESS
 CONSTANT c : time := 3 ns;
BEGIN
 sig_a <= '1' AFTER  5 ns, '0' AFTER  7 ns, '0' AFTER 12 ns,
          '1' AFTER 17 ns, '0' AFTER 24 ns,
          '1' AFTER 26 ns, '0' AFTER 30 ns, '0' AFTER 34 ns,
          '1' AFTER 38 ns;
 WAIT;
END PROCESS;
```

Die Verläufe der verschiedenen Attribute des Signals `sig_a` über der Zeit werden in Abb. B-12 dargestellt.

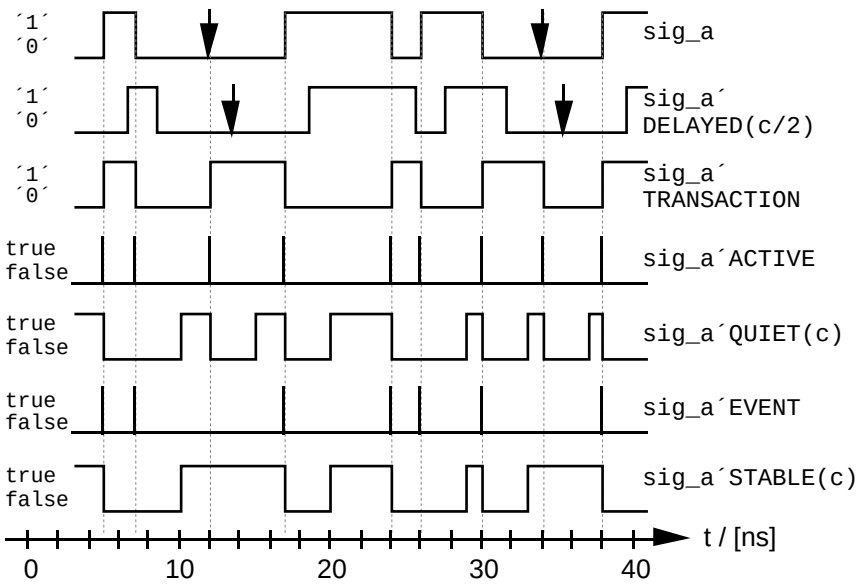

Abb. B-12: Signalbezogene Attribute

Mit ✓**93** sind weitere signalbezogene Attribute verfügbar:

Name	Funktion
`s'DRIVING`	liefert `false`, falls der Treiber des Signals `s` gerade abgeschaltet ("disconnected") ist, sonst `true` (siehe Kapitel 9)
`s'DRIVING_VALUE`	liefert den aktuellen Wert des Treibers für das Signal `s` (siehe Kapitel 9)

Die beiden letztgenannten Attribute können in Prozessen, die dem Signal `s` einen Wert zuweisen, oder in Prozeduren, die `s` als `OUT`-, `BUFFER`- oder `INOUT`-Signal besitzen, verwendet werden. Sie sind sinnvoll, um Signale vor der Behandlung mit Auflösungsfunktionen oder Ports vom Modus `OUT` lesen zu können.

Anwendungen der signalbezogenen Attribute finden sich in den Abschnitten über Signalzuweisungen und den VHDL-Beispielen.

6.2.4 Blockbezogene Attribute ✓**87**

Die folgenden Attribute sind nur in der alten Norm (✓**87**) verfügbar. Mit der Überarbeitung der Norm wurden beide Attribute gestrichen!

Name	Funktion
`b'BEHAVIOR`	liefert `true`, falls der Block (oder die Architektur) `b` eine reine Verhaltensbeschreibung enthält (d.h. falls keine Komponenteninstantiierungen enthalten sind), sonst `false`
`b'STRUCTURE`	liefert `true`, falls der Block (oder die Architektur) `b` eine rein strukturale Beschreibung enthält (d.h. falls keine Signalzuweisungen enthalten sind), sonst `false`

6.2.5 Allgemeine Attribute ✓93

In ✓93 sind auch Attribute verfügbar, die den Namen oder den Pfad von Objekten in der Hierarchie eines VHDL-Modells ermitteln. Dies kann zur Fehlersuche im Zusammenspiel mit der ASSERT-Anweisung nützlich sein.

Diese allgemeinen Attribute lassen sich nicht nur auf Objekte anwenden, sondern meistens auch auf alle VHDL-Einheiten (vgl. Gruppen), die einen Namen besitzen:

Name	Funktion
e'SIMPLE_NAME	liefert den Namen der Einheit e als Zeichenkette (string) in Kleinbuchstaben
e'PATH_NAME	liefert den Pfad der Einheit e innerhalb des Modells als Zeichenkette in Kleinbuchstaben.
e'INSTANCE_NAME	wie PATH_NAME, zusätzlich mit Informationen über Konfigurationen bei Komponenten

6.3 Signalzuweisungen und Verzögerungsmodelle

Die wohl wichtigste Anweisung in VHDL ist die Zuweisung von neuen Werten an Signale. Signale dienen als Informationsträger innerhalb eines VHDL-Modells und zur Kommunikation mit dessen Umwelt.

6.3.1 Syntax

Signalzuweisungen können nebenläufig sein oder als sequentielle Anweisungen innerhalb von Prozessen, Funktionen oder Prozeduren stehen.

Das Ziel der Zuweisung (`sig_name`) kann ein einzelnes Signal oder ein Teil eines Vektors (slice), bestehend aus mehreren Signalen sein. Als zuzuweisendes Argument (*`value_expr`*) kann bei Signalzuweisungen wieder ein **Signal** gleichen Typs oder ein beliebiger **Ausdruck**, der einen Signal typkonformen Wert liefert, stehen. Der neue Signalwert kann einerseits nur um ein Delta verzögert zugewiesen werden, indem der optionale `AFTER`-Teil der Signalzuweisung weggelassen wird oder indem eine Null-Verzögerung angegeben wird ("`AFTER 0 ns`"). Andererseits ist die explizite Angabe einer Verzögerungszeit über das Schlüsselwort `AFTER` und eine nachfolgende Zeitangabe (*`time_expr`*) möglich. In VHDL stehen dazu zwei verschiedene Verzögerungsmodelle zur Verfügung, die hier erläutert werden sollen.

Die Grundsyntax für Signalzuweisungen lautet:

```
sig_name <= [TRANSPORT]
            value_expr_1 [AFTER time_expr_1]
          {, value_expr_n  AFTER time_expr_n } ;
```

Das Schlüsselwort `TRANSPORT` dient zur Kennzeichnung des "Transport"-Verzögerungsmodells. Falls das Schlüsselwort fehlt, wird das "Inertial"-Verzögerungsmodell angewandt. Beiden Modellen gemein ist der sog. "Preemption-Mechanismus".

6.3.2 Ereignisliste

Die Auswirkungen dieses Mechanismus und die Verzögerungsmodelle können am besten durch eine graphische Darstellung der Ereignisliste erläutert werden.

Die Ereignisliste ("waveform" oder "event queue") enthält alle zukünftigen Signalwechsel in einem Simulationslauf. Signalwechsel, die mit Verzögerungszeiten behaftet sind, werden an die entsprechende Stelle in dieser Liste eingetragen. Bei der Simulation wird diese Ereignisliste abgearbeitet, d.h. die dort vermerkten Signalwechsel ausgeführt und deren Auswirkungen berechnet, was in aller Regel neue Einträge in der Ereignisliste bewirkt.

Folgende, zum Zeitnullpunkt durchgeführte Signalzuweisung, entspricht der graphischen Ereignisliste der Abb. B-13.

```
sig_a <= '1' AFTER 2 ns,  '0' AFTER 5 ns,  '1' AFTER 10 ns,
         '1' AFTER 12 ns, '0' AFTER 15 ns, '1' AFTER 17 ns;
```

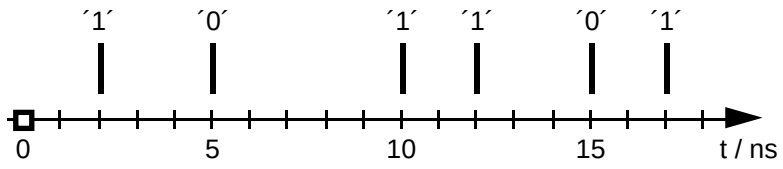

Abb. B-13: Ereignisliste für das Signal sig_a

Es sei an dieser Stelle auf einige Begriffe aus der "VHDL-Welt" hingewiesen, die im Zusammenhang mit den Signalzuweisungen stehen. Die einzelnen Einträge in der Ereignisliste, die jeweils aus einer Zeit- und einer Wertangabe bestehen, werden als **Transaktion** ("transaction") bezeichnet. Ein **Ereignis** ("event") auf einem Signal tritt auf, wenn ein Signal gerade seinen Wert ändert. Dagegen ist ein Signal auch **aktiv** ("active"), falls ihm gerade ein Wert zugewiesen wird, unabhängig davon, ob sich dadurch der Signalwert ändert oder nicht. In Abb. B-13 sind also zu den Zeitpunkten 2, 5, 10, 12, 15 und 17 ns Transaktionen festgelegt. Ein Ereignis auf dem Signal würde zum Zeitpunkt 2, 5, 10, 15 und 17 ns auftreten, sofern sich die Ereignisliste zwischenzeitlich

nicht ändert. Aktiv wäre das Signal dagegen an allen Ereigniszeitpunkten und zusätzlich bei 12 ns.

6.3.3 Preemption-Mechanismus

"Preemption" bezeichnet das Entfernen von geplanten Transaktionen aus der Ereignisliste. Der Premption-Mechanismus wird bei jeder neuen Signalzuweisung angewandt. Welche Transaktionen dabei gelöscht werden, hängt vom bei der Signalzuweisung verwendeten Verzögerungsmodell ab.

Für VHDL-Simulatoren ist der Preemption-Mechanismus durch die VHDL-Norm vorgeschrieben. Herkömmliche Digitalsimulatoren arbeiten oft nicht nach diesem Mechanismus.

6.3.4 "Transport"-Verzögerungsmodell

Bei diesem Verzögerungsmodell von VHDL werden alle Transaktionen gelöscht, die nach einem neuen Signalwert oder gleichzeitig mit diesem auftreten. Es führt dazu, daß eine Signalzuweisung "`sig_a <= TRANSPORT '1' AFTER 11 ns;`" die zum Zeitpunkt 2 ns durchgeführt wird, alle diesem neuen Eintrag bei 13 ns nachfolgenden oder zur gleichen Zeit stattfindenden Einträge aus der Ereignisliste löscht. Es ergibt sich damit eine neue Ereignisliste für das Signal `sig_a`:

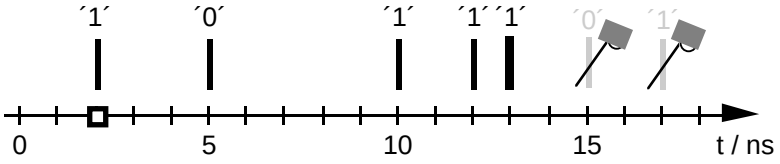

Abb. B-14: Neue Ereignisliste für das Signal sig_a nach dem "Transport"-Verzögerungsmodell

6.3.5 "Inertial"-Verzögerungsmodell

Beim "Inertial"-Verzögerungsmodell wird zusätzlich zum "Transport"-Verzögerungsmodell folgende Regel angewandt:

① Markiere die unmittelbar vor dem neuen Eintrag stattfindende Transaktion, falls sie den gleichen Wert besitzt.

② Markiere die aktuelle und die neue Transaktion.

③ Lösche alle nicht markierten Transaktionen.

Diese Vorgehensweise führt dazu, daß nur diejenigen Impulse, die eine längere (oder gleichlange) Dauer als die angegebene Verzögerungszeit besitzen, auch tatsächlich auftreten. Das Inertial-Verzögerungsmodell ist immer dann gültig, wenn in der Signalzuweisung nicht das Schlüsselwort `TRANSPORT` auftritt.

Eine Zuweisung "`sig_a <= '1' AFTER 11 ns;`" die zum Zeitpunkt 2 ns durchgeführt wird, führt aufgrund des Inertial-Modells auf folgende Ereignisliste für das Signal `sig_a`:

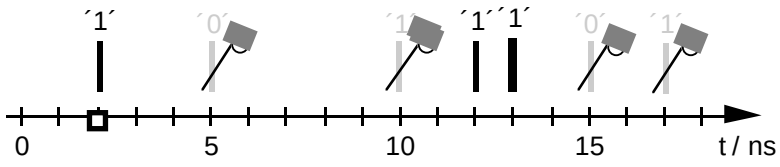

Abb. B-15: Neue Ereignisliste für das Signal sig_a nach dem "Inertial"-Verzögerungsmodell

6.3.6 "Reject-Inertial"-Verzögerungsmodell ✓93

Bei der Zuweisung "`sig_b <= sig_c AFTER 3 ns;`" wird das "Inertial"-Verzögerungsmodell angewandt. Eine wesentliche Konsequenz ist, daß jeder Impuls des Signals `sig_c`, der kürzer als die angegebene Verzögerungszeit von 3 ns ist, unterdrückt ("rejected") wird.

Dies ist bei manchen Komponenten sicherlich sinnvoll, die kurze Impulse ("Spikes") unterdrücken oder herausfiltern. Oft entspricht der Grenzwert dieser Impulsdauer aber nicht der Verzögerungszeit der Komponente. In ✓93 wurde deshalb ein drittes Verzögerungsmodell

eingeführt, das eine von der Verzögerungszeit unabhängige Zeitangabe für eine minimal übertragene Impulsdauer (*rej_time_expr*) gestattet. Es hat folgende Syntax:

```
sig_name <= [[REJECT rej_time_expr] INERTIAL]
          value_expr_1 [AFTER time_expr_1]
       {, value_expr_n  AFTER time_expr_n  } ;
```

Um eine deutlichere Kennzeichnung des (Default-) "Inertial"-Verzögerungsmodells zu erzielen, ist das Schlüsselwort `INERTIAL` nun auch ohne `REJECT` optional. Die Obergrenze für die Impulsdauer (*rej_time_expr*) darf bei diesem Verzögerungsmodell nicht größer als die erste Verzögerungszeit (*time_expr_1*) sein.

Folgendes Beispiel zeigt die Auswirkungen der drei Verzögerungsmodelle anhand von verschieden breiten Impulsen des Signals `sig_s`:

```
sig_s <= TRANSPORT '1' AFTER  1 ns, '0' AFTER  5 ns,
                   '1' AFTER 10 ns, '0' AFTER 13 ns,
                   '1' AFTER 18 ns, '0' AFTER 20 ns,
                   '1' AFTER 25 ns, '0' AFTER 26 ns;

sig_t <= TRANSPORT            sig_s AFTER 3 ns;
sig_i <=                      sig_s AFTER 3 ns;
sig_r <= REJECT 2 ns INERTIAL sig_s AFTER 3 ns; -- ! VHDL'93
```

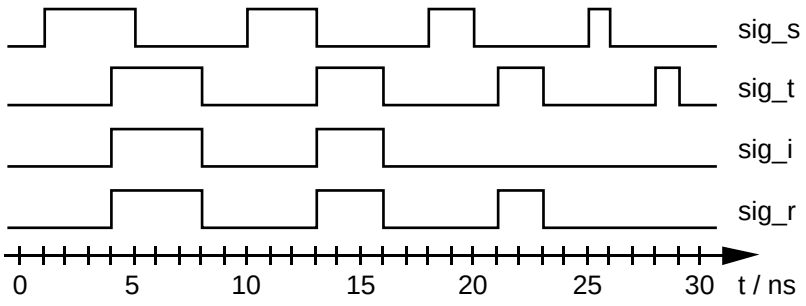

Abb. B-16: Signalverläufe bei unterschiedlichen Verzögerungsmodellen

Abb. B-16 zeigt folgende Effekte: Bei Anwendung des "Transport"-Verzögerungsmodells (`sig_t`) wird das Quellsignal (`sig_s`) unverändert in seiner Form um 3 ns verzögert. Beim "Inertial"-Verzögerungsmodell (`sig_i`) werden Impulse, die kürzer als die Verzögerungszeit sind, unterdrückt. Das "Reject-Inertial"-Modell schließlich unterdrückt nur Impulse, die kürzer als die nach dem Schlüsselwort `REJECT` spezifizierte Zeit von 2 ns sind.

Weitere, interessante Effekte treten auf, wenn unterschiedliche Verzögerungszeiten für steigende und fallende Signalflanken spezifiziert werden. Diesen Fall beleuchtet folgendes Beispiel:

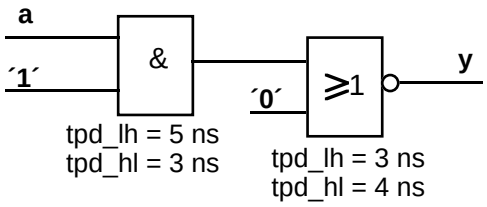

Abb. B-17: Flankenabhängige Laufzeiten

Wird für beide Gatter das "Transport"-Verzögerungsmodell verwendet, so passiert (durch den "Preemption"-Mechanismus) kein positiver Impuls des Signals `a`, der kürzer als 2 ns ist, das AND-Gatter. Impulse, die länger als 2 ns sind, werden um 2 ns verkürzt. Das zweite Gatter invertiert den positiven Impuls und verkürzt ihn wiederum um 1 ns. Bei der gegebenen Beschaltung gelangen also positive Impulse am Eingang `a` nur an den Ausgang `y`, wenn ihre Dauer größer als 3 ns ist.

Noch komplizierter wird der Fall, wenn das "Inertial"-Verzögerungsmodell verwendet wird. Positive Impulse an `a` werden um 2 ns verkürzt und erst ab 5 ns Dauer weitergegeben. Das zweite Gatter läßt positive Impulse nur passieren, wenn sie länger als 4 ns sind. Sie werden dabei um 1 ns verkürzt. Der kürzeste, nicht unterdrückte positive Impuls an `a` muß demnach 6 ns lang sein und erscheint am Ausgang mit einer Dauer von nur 3 ns.

Dem Leser wird empfohlen, sich diese Situation mit Hilfe von Ereignislisten zu verdeutlichen. Außerdem kann untersucht werden, wie sich der Ausgang bei negativen Impulsen am Eingang `a` verhält.

6.4 Nebenläufige Anweisungen

Nebenläufige Anweisungen enthalten optional innerhalb der Anweisung ein sog. Label. Dieses Label kann auch als Name der Anweisung interpretiert werden. Er dient zum späteren Referenzieren der spezifischen Anweisung (z.B. bei der Konfiguration eines strukturalen Modells) oder auch für benutzerdefinierte Attribute.

Sequentielle Anweisungen dürfen hingegen nach ✓**87** im allgemeinen keine Labels tragen. Erst mit ✓**93** können Labels für alle Anweisungen (nebenläufig und sequentiell) vergeben werden.

6.4.1 Signalzuweisungen (normal und bedingt)

Im Falle von nebenläufigen Signalzuweisungen lautet die Syntax der nicht-bedingter Zuweisung:

```
[assignment_label :] sig_name <= [TRANSPORT]
         value_expr_1 [AFTER time_expr_1]
      { , value_expr_n  AFTER time_expr_n } ;
```

Werden bei der Signalzuweisung mehrere Werte angegeben, so müssen diese in zeitlich aufsteigender Reihenfolge angeordnet sein.

Die erste Alternative **bedingter Signalzuweisungen** ("conditional signal assignment") basiert auf mehreren Zuweisungsalternativen, die jeweils durch Bedingungen (*condition*) gesteuert werden. Dies entspricht also einer sequentiellen IF-ELSIF-ELSE-Struktur. Die Syntax des "conditional signal assignment" hat folgendes Aussehen:

```
[assignment_label :] sig_name <= [TRANSPORT]
  {    value_expr_m [AFTER time_expr_m]
   {,  value_expr_n  AFTER time_expr_n }
                       WHEN condition_m ELSE }
       value_expr_o [AFTER time_expr_o]
   {,  value_expr_p  AFTER time_expr_p };
```

Die zweite Alternative ("selected signal assignment") entspricht in ihrem Verhalten der sequentiellen CASE-Anweisung. Sie basiert auf bestimmten Alternativen (*choice*) eines Ausdruckes (*expression*) und hat folgendes Aussehen:

```
[assignment_label :] WITH expression SELECT
sig_name <= [TRANSPORT]
  {    value_expr_m [AFTER time_expr_m]
   {,  value_expr_n  AFTER time_expr_n}
                          WHEN choice_m , }
       value_expr_o [AFTER time_expr_o]
   {,  value_expr_p  AFTER time_expr_p}
                          WHEN choice_o ;
```

Werden mit den einzelnen Auswahlalternativen nicht alle möglichen Werte von *expression* abgefragt, so muß an Stelle von *choice_o* das Schlüsselwort OTHERS gesetzt werden, um die nicht explizit abgefragten Werte zu erfassen.

```
ARCHITECTURE behavioral OF signals IS
  SIGNAL sig_a, sig_b, sig_c : std_ulogic;
BEGIN
-- nebenlaeufige Signalzuweisung mit TRANSPORT -------------
  sig_a <= TRANSPORT '0', '1' AFTER 2 ns, 'Z' AFTER 3 ns;

-- !!! illegal: Werte nicht zeitlich aufsteigend geordnet --
  sig_a <= TRANSPORT '0', 'Z' AFTER 3 ns, '1' AFTER 2 ns;

-- "conditional signal assignment" (csa) -------------------
  csa: sig_b <= '1', '0' AFTER 2 ns WHEN sel = 1 ELSE
                '0', '1' AFTER 3 ns WHEN sel = 2 ELSE
                'Z';

-- dem csa entsprechendes "selected signal assignment" (ssa)
  ssa: WITH sel SELECT
       sig_c <= '1', '0' AFTER 2 ns WHEN 1,
                '0', '1' AFTER 3 ns WHEN 2,
                'Z' WHEN OTHERS;
END behavioral;
```

In vielen Fällen von nebenläufigen, bedingten Signalzuweisungen ist bei bestimmten Bedingungen kein Signalwechsel erforderlich. Die einzige Möglichkeit, dies mit der alten VHDL-Norm zu realisieren, liegt in der Zuweisung des Signals an sich selbst, wie es das Beispiel eines einfachen taktpegelgesteuerten Speicherbausteins (Latch) zeigen soll:

```
ENTITY latch IS
  PORT (d, clk : IN bit; q : BUFFER bit) ;
END latch ;

ARCHITECTURE concurrent_1 OF latch IS
BEGIN
  q <= d WHEN clk = '1' ELSE q ;     -- csa-Alternative
END concurrent_1 ;

ARCHITECTURE concurrent_2 OF latch IS
BEGIN
  WITH clk SELECT                    -- ssa-Alternative
    q <= d WHEN '1' ,
         q WHEN OTHERS ;
END concurrent_2 ;
```

Diese Realisierungen haben jedoch beide den Nachteil, daß bei negativen Taktflanken unnötige Transaktionen des Signals erzeugt werden und durch den "Preemption"-Mechanismus möglicherweise Informationen verloren gehen.

Eine saubere Lösung des Problems bietet ✓**93** mit dem Schlüsselwort UNAFFECTED. Es kann bei den bedingten Signalzuweisungen eingesetzt werden und erzeugt keine Transaktion auf dem Signal:

```
ARCHITECTURE concurrent_3 OF latch IS    -- !!! nur VHDL'93
BEGIN
   q <= d WHEN clk = '1'                 -- csa
          ELSE UNAFFECTED ;
END concurrent_3 ;

ARCHITECTURE concurrent_4 OF latch IS    -- !!! nur VHDL'93
BEGIN
   WITH clk SELECT
      q <= d          WHEN '1',          -- ssa
           UNAFFECTED WHEN OTHERS ;
END concurrent_4 ;
```

6.4.2 Assertions

Assertions dienen zur Überprüfung von Bedingungen und zur Ausgabe von Warnungen bzw. Fehlermeldungen. Die Syntax lautet:

```
[assert_label :] ASSERT condition
                 [ REPORT "message_string"]
                 [ SEVERITY severity_level] ;
```

Diese Syntax wird folgendermaßen interpretiert:

> "Überprüfe, ob die Bedingung *condition* erfüllt ist; falls nicht, erzeuge die Meldung "message_string" und breche, abhängig von der Fehlerklasse severity_level, gegebenenfalls die Simulation ab."

Eine Fehlermeldung mit evtl. weiteren Konsequenzen tritt also nur auf, falls die angegebene Bedingung (*condition*) den Wert false ergibt.

Ohne Angabe der Fehlermeldung wird der String "Assertion violation." ausgegeben.

Die vier möglichen Fehlerklassen (entsprechend dem vordefinierten Aufzähltyp severity_level) haben folgende Bedeutung:

- note dient zur Ausgabe von allgemeinen Informationen,
- warning dient zur Anzeige von möglichen unerwünschten Bedingungen,
- error zeigt an, daß eine Aufgabe mit dem falschen Ergebnis abgeschlossen wurde,
- failure zeigt an, daß eine Aufgabe nicht abgeschlossen werden konnte.

Wird in der Anweisung keine Fehlerklasse angegeben, so wird sie mit der Klasse error versehen. Die Entscheidung, ab welcher Klasse die Simulation abgebrochen wird, legt man i.d.R. durch eine spezifische Simulatoreinstellung fest.

Zwei Beispiele zur Anzeige eines (low-aktiven) Resetsignals und zur Prüfung auf definierten Pegel eines Signals sig_a vom Typ std_ulogic lauten:

```
reset_check : ASSERT sig_reset /= '0'
                REPORT "Achtung: Reset ist aktiv !"
                SEVERITY note ;
```

```
ASSERT (now = 0 fs) OR (sig_a /= 'U')
  REPORT "sig_a ist nicht initialisiert !" ;
```

Im zweiten Beispiel wird die Ausgabe einer Fehlermeldung zum Zeitnullpunkt unterdrückt.

6.4.3 Prozesse

Prozesse dienen als Umgebung für sequentielle, d.h. nacheinander ablaufende Befehle. Sie werden also zur Modellierung prozeduraler Vorgänge verwendet. Die Prozesse selbst gelten als nebenläufige Anweisung, d.h. existieren mehrere Prozesse innerhalb einer Architektur, so können sie gleichzeitig aktiv sein. Prozesse werden durch zwei verschiedene Möglichkeiten aktiviert und gestoppt, die sich gegenseitig ausschließen:

- Durch eine **Liste sensitiver Signale** im Prozeß-Kopf:
  Prozesse dieser Art werden einmalig bei der Modell-Initialisierung komplett durchlaufen und zu späteren Zeitpunkten erst wieder aktiviert, wenn sich eines der Signale der "sensitivity list" ändert. Dann wird der Prozeß wieder bis zum Ende abgearbeitet, usw.
- Durch `WAIT`-**Anweisungen**:
  Bei der Modell-Initialisierung zum Zeitnullpunkt wird der Prozeß bis zur ersten `WAIT`-Anweisung abgearbeitet und erst wieder aktiviert, wenn die Bedingung der `WAIT`-Anweisung erfüllt ist oder die dort angegebene Zeit verstrichen ist (vgl. `WAIT`-Anweisung).

Prozesse ohne `WAIT`-Anweisung und ohne "sensitivity list" sind üblicherweise nicht sinnvoll, da diese Prozesse beim Simulationsstart aufgerufen und dann ständig zyklisch durchlaufen werden ("Endlosschleife").

Prozesse bestehen aus einem Deklarations- und einem Anweisungsteil. Die Syntax der beiden Varianten lautet:

```
[process_label :] PROCESS (sig_1 {, sig_n})
   ...
   ... -- Deklaration von: Typen und Unter-
   ... -- typen, Aliases, Konstanten, Files,
   ... -- Variablen, Unterprogrammen
   ... -- Definition von: Unterprogrammen,
   ... -- Attributen
   ... -- USE-Anweisungen
   ...
BEGIN
   ...
   ... -- sequentielle Anweisungen ohne WAIT
   ...
END PROCESS [process_label] ;

[process_label :] PROCESS
   ...
   ... -- Deklarationsteil wie oben
   ...
BEGIN
   ...
   ... -- sequentielle Anweisungen
   ...
   WAIT ... ; -- mind. eine WAIT-Anweisung
   ...
   ... -- sequentielle Anweisungen
   ...
END PROCESS [process_label] ;
```

Während (herkömmliche) Variablen nur innerhalb eines Prozesses gültig und außerhalb nicht sichtbar sind, können Signale auch außerhalb eines Prozesses gelesen und mit neuem Wert versehen, d.h. beschrieben werden. Sie können somit zur Kommunikation (Austausch von Informationen, gegenseitiges Aktivieren usw.) zwischen Prozessen verwendet werden.

Ein wesentlicher Aspekt im Zusammenhang mit Prozessen ist das Zeitverhalten, insbesondere der zeitliche Unterschied zwischen Variablen-

und Signalzuweisungen innerhalb eines Prozesses. Während Variablenwerte sofort bei der Abarbeitung der Anweisung zugewiesen werden, werden die neuen Werte von Signalen zunächst vorgemerkt und erst nach Abarbeitung aller aktiven Prozesse am Ende eines sog. Delta-Zyklus zugewiesen. Diese Problematik wird getrennt in Kapitel 8 behandelt.

Da sequentielle Anweisungen an dieser Stelle noch nicht behandelt wurden, wird für ausführliche Beispiele zu Prozessen auf die nachfolgenden Abschnitte verwiesen. An dieser Stelle wird deshalb nur ein kurzes Beispiel für ein D-Latch aufgeführt, das eine selbsterklärende `IF`-Struktur enthält:

```
ARCHITECTURE sequential_1 OF latch IS
BEGIN
-- Aktivierung des Prozesses durch Ereignisse auf d oder clk
   q_assignment: PROCESS (d, clk)
   BEGIN
      IF clk = '1' THEN
         q <= d ;
      END IF ;
   END PROCESS q_assignment ;
END sequential_1 ;
```

Die Vereinheitlichung der Rahmensyntax in ✓**93** führt zu einer optionalen Angabe des Schlüsselwortes `IS` :

```
[process_label :]
              PROCESS [(sig_1 {, sig_n})] [IS]
   ...
BEGIN
   ...
END PROCESS [process_label] ;
```

6.5 Sequentielle Anweisungen

Wie bereits erwähnt, kann mit der Überarbeitung der VHDL-Norm (✓**93**) jeder Anweisung, also auch den sequentiellen Anweisungen, ein Label zugeteilt werden.

6.5.1 Signalzuweisungen

Die Syntax für sequentielle Signalzuweisungen lautet:

```
sig_name <= [TRANSPORT]
          value_expr_1 [AFTER  time_expr_1]
      { ,  value_expr_n  AFTER  time_expr_n } ;
```

Die einzelnen Signalwerte müssen dabei in zeitlich aufsteigender Reihenfolge angeordnet sein.

Bedingte Signalzuweisungen sind im sequentiellen Fall nicht erlaubt. Sie können aber durch `IF-ELSIF-ELSE`- oder `CASE`-Anweisungen abgebildet werden.

Die Verzögerungsmodelle sind wie bei den nebenläufigen Zuweisungen zu verwenden ("Inertial" als Defaultmodell, "Transport" durch explizite Kennzeichnung mit dem Schlüsselwort `TRANSPORT`). Das "Reject-Inertial"-Modell (✓**93**) ist entsprechend der ebenfalls oben beschriebenen Syntax anzuwenden.

6.5.2 Variablenzuweisungen

Die innerhalb von Prozessen und Unterprogrammen verwendeten Variablen (zeitabhängige Objekte ohne Aufzeichnung des Verlaufs über der Zeit) können nur unverzögert zugewiesen werden. Sie eignen sich z.B. zur Speicherung von Zwischenergebnissen. Wenn möglich, sollten Variablen anstelle von Signalen verwendet werden, da sie bei der Simulation weniger Verwaltungsaufwand (Rechenzeit, Speicherplatz) als Signale erfordern.

Die Syntax für Variablenzuweisungen lautet:

```
var_name := value_expr ;
```

Als neuer Variablenwert (*value_expr*) kann entweder der Wert eines anderen Objektes (Signal, Variable, Konstante), ein expliziter Wert oder ein Ausdruck angegeben werden.

Man beachte den Unterschied zum Signalzuweisungsoperator (":=" anstelle von "<="). Ein weiterer Unterschied zur Signalzuweisung liegt

im Ausführungszeitpunkt: während Signalzuweisungen erst am Ende eines Delta-Zyklus nach Ausführung aller aktiven Prozesse durchgeführt werden, werden Variablen unmittelbar, d.h. bei Erreichen der entsprechenden Anweisung im sequentiellen Ablauf zugewiesen. Die Konsequenzen aus diesem Sachverhalt und Beispiele hierzu werden im Kapitel über den Simulationsablauf (Kapitel 8) aufgezeigt.

Das folgende Beispiel illustriert die Verwendung von sequentiellen Signal- und Variablenzuweisungen:

```
ENTITY mult IS
  PORT (a, b : IN integer := 0; y : OUT integer) ;
END mult ;

ARCHITECTURE number_one OF mult IS
BEGIN
  PROCESS (a,b) -- Aktivierung durch Ereignisse auf a oder b
    VARIABLE v1, v2 : integer := 0 ;
  BEGIN
    v1 := 3 * a + 7 * b ;      -- sequent. Variablenzuweisung
    v2 := a * b + 5 * v1 ;     -- sequent. Variablenzuweisung
    y <= v1 + v2 ;             -- sequent. Signalzuweisung
  END PROCESS ;
END number_one ;
```

6.5.3 Assertions

Die Ausgabe von Fehlermeldungen ist ebenfalls als sequentielle Anweisung möglich. Hier besteht zur Syntax der nebenläufigen Form nur ein Unterschied in der alten VHDL-Norm: Es dürfen keine Labels verwendet werden.

```
ASSERT condition [ REPORT "message_string"]
                 [ SEVERITY severity_level];
```

Wie bei allen Anweisungen in der überarbeiteten VHDL-Norm ist hier auch im sequentiellen Fall ein Label erlaubt.

Die Verknüpfung von Meldungen mit Assertions innerhalb einer Anweisung hat zur Folge, daß nicht bedingte Meldungen nur mit folgendem Konstrukt ausgegeben werden können:

```
ASSERT false REPORT "Dies ist eine Meldung" SEVERITY note ;
```

Mit der überarbeiteten VHDL-Syntax (✓**93**) kann nun eine Meldung auch ohne Assertion ausgegeben werden. Dazu ist das Schlüsselwort `REPORT` alleine (mit optionaler Fehlerklasse) ausreichend. Defaultwert für die Klasse der Meldung ist hier `note`:

```
[report_label :]  REPORT "message_string"
                  [  SEVERITY severity_level] ;
```

6.5.4 WAIT-Anweisung

`WAIT`-Anweisungen können die Abarbeitung von sequentiellen Anweisungen steuern. Sie dürfen nur in Prozessen ohne "sensitivity-list" und in Prozeduren, die nicht von Prozessen mit "sensitivity-list" aufgerufen werden, auftreten. Als Argumente einer `WAIT`-Anweisung können ein oder mehrere Signale, Bedingungen oder Zeitangaben verwendet werden. Ein "`WAIT;`" ohne jegliches Argument bedeutet "warte für immer" und beendet somit die Ausführung eines Prozesses oder einer Prozedur.

```
WAIT [ON signal_name_1 {, signal_name_n}]
     [UNTIL condition]
     [FOR time_expression] ;
```


Die einzelnen Argumente haben folgende Bedeutung für das Zeitverhalten von Prozessen:

- Eine Liste von Signalen bewirkt, daß solange gewartet wird, bis sich mindestens eines der Signale ändert, d.h. ein Ereignis auftritt. Ein Prozeß mit einer Liste von Signalen als Argument einer am Ende stehenden `WAIT`-Anweisung entspricht somit einem Prozeß mit den gleichen Signalen als Elemente der "sensitivity-list" im Prozeßkopf.

  Ist ein Signal der Liste ein Vektor oder ein höherdimensionales Feld, so erfüllt bereits die Änderung eines einzigen Elementes die `WAIT`-Bedingung.

- Eine Bedingung (*`condition`*) unterbricht die Prozeßabarbeitung solange, bis die Bedingung erfüllt ist.

  Bei Angabe von Bedingung und Signalliste muß die Bedingung erfüllt sein und der Signalwechsel auftreten.

- Die Angabe eines Ausdruckes, der als Ergebnis eine Zeitangabe liefert (*`time_expression`*), stoppt die Prozeßabarbeitung maximal für diese Zeitdauer.

Folgende Beispiele geben weitere Architekturen für das bereits erwähnte Latch wieder:

```
ARCHITECTURE sequential_2 OF latch IS
BEGIN
   q_assignment: PROCESS
   BEGIN
      IF clk = '1' THEN
         q <= d ;
      END IF ;
      WAIT ON d, clk ;     -- entspricht "sensitivity-list"
   END PROCESS q_assignment ;
END sequential_2 ;
```

```
ARCHITECTURE sequential_3 OF latch IS
BEGIN
   q_assignment: PROCESS
   BEGIN
      q <= d ;
      WAIT ON d, clk UNTIL clk = '1' ;    -- ersetzt IF-Anw.
   END PROCESS q_assignment ;
END sequential_3 ;
```

6.5.5 IF-ELSIF-ELSE-Anweisung

Bedingte Verzweigungen in sequentiellen Anweisungsteilen können mit der `IF-ELSIF-ELSE`-Anweisung folgendermaßen realisiert werden:

```
IF condition_1 THEN
   ...
   ...        -- sequentielle Anweisungen
   ...
{ ELSIF condition_n THEN
   ...
   ...        -- sequentielle Anweisungen
   ... }
[ ELSE
   ...
   ...        -- sequentielle Anweisungen
   ... ]
END IF ;
```

Zwingend erforderlich sind nur die erste Bedingung und die Kennzeichnung des Endes der Struktur durch "`END IF;`". Die `ELSIF`- und `ELSE`-Teile sind optional, wobei ersterer mehrfach auftreten kann.

Mit der überarbeiteten Syntax (✓93) kann einer IF-ELSIF-ELSE-Anweisung auch ein Label gegeben werden. Entsprechend kann in der END IF-Anweisung dieses Label wiederholt werden:

```
[if_label :] IF condition_1 THEN
   ...
END IF [if_label] ;
```

6.5.6 CASE-Anweisung

Eine weitere Möglichkeit der bedingten Ausführung bestimmter Anweisungen liegt in der CASE-Anweisung. Sie bietet sich insbesondere bei mehrfacher Verzweigung basierend auf einem Ausdruck an:

```
CASE expression IS
 {   WHEN value_n =>
        ...
        ...  -- sequentielle Anweisungen
        ... }
 [   WHEN OTHERS  =>
        ...
        ...  -- sequentielle Anweisungen
        ... ]
END CASE ;
```

Im Gegensatz zu IF-ELSIF-ELSE müssen hier allerdings alle Werte (*value_n*), die der Ausdruck (*expression*) annehmen kann, explizit angegeben werden. Um die noch nicht behandelten Werte abzufragen, kann auch das Schlüsselwort OTHERS dienen.

Folgende Beispiele einer bedingten Verzweigung sind äquivalent:

```
ENTITY four_byte_rom IS
  PORT (address  : IN  integer RANGE 1 TO 4;
        contents : OUT bit_vector(1 TO 8) ) ;
END four_byte_rom;
```

```
ARCHITECTURE if_variante OF four_byte_rom IS
BEGIN
  PROCESS (address)
  BEGIN
    IF address = 1 THEN
      contents <= ('0','0','0','0','1','1','1','1') ;
    ELSIF address = 2 THEN
      contents <= "00111111" ;
    ELSIF address = 3 THEN
      contents <= b"11111100" ;
    ELSE
      contents <= x"f0" ;
    END IF ;
  END PROCESS ;
END if_variante ;
```

```
ARCHITECTURE case_variante_1 OF four_byte_rom IS
BEGIN
  PROCESS (address)
  BEGIN
    CASE address IS
      WHEN 1 => contents <=
        ('0','0','0','0','1','1','1','1') ;
      WHEN 2 => contents <= "00111111"  ;
      WHEN 3 => contents <= b"11111100" ;
      WHEN OTHERS => contents <= x"f0"  ;
    END CASE ;
  END PROCESS ;
END case_variante_1 ;
```

Beide Varianten verwenden dabei zur weiteren Reduzierung des Code-Umfangs die Möglichkeit, Bit-Vektoren als Strings anzugeben. Anstelle der letzten Alternative ("`WHEN OTHERS`") hätte hier auch "`WHEN 4`" stehen können.

Mit Hilfe von Oder-Verknüpfungen ("`|`") oder Bereichsangaben (`TO`, `DOWNTO`) können mehrere Fälle des Ausdruckes für gleiche Anweisungsteile zusammengefaßt werden. Damit ergibt sich eine weitere Architekturalternative:

```
ARCHITECTURE case_variante_2 OF four_byte_rom IS
BEGIN
  PROCESS (address)
  BEGIN
    CASE address IS
      WHEN 1 | 2  => contents(1 TO 2) <= "00" ;
                     contents(7 TO 8) <= "11" ;
      WHEN OTHERS => contents(1 TO 2) <= "11" ;
                     contents(7 TO 8) <= "00" ;
    END CASE ;
    CASE address IS
      WHEN 2 TO 4 => contents(3 TO 4) <= "11" ;
      WHEN OTHERS => contents(3 TO 4) <= "00" ;
    END CASE ;
    CASE address IS
      WHEN 1 TO 3 => contents(5 TO 6) <= "11" ;
      WHEN OTHERS => contents(5 TO 6) <= "00" ;
    END CASE ;
  END PROCESS ;
END case_variante_2 ;
```

Die einheitliche Handhabung von Labels und Rahmensyntax in der überarbeiteten VHDL-Norm (✓93) erlaubt auch für die CASE-Anweisung folgende Syntaxvariante:

```
[case_label :] CASE expression IS
   ...
END CASE [case_label] ;
```

6.5.7 NULL-Anweisung

Die NULL-Anweisung:

```
NULL ;
```

führt keine Aktion aus. Sie dient zur expliziten Kennzeichnung von aktionslosen Fällen in IF- und vor allem in CASE-Anweisungen.

Die Modellierung des Latches mit Hilfe einer CASE-Anweisung könnte somit folgendes Aussehen haben:

```
ARCHITECTURE sequential_4 OF latch IS
BEGIN
   q_assignment: PROCESS (clk, d)
   BEGIN
      CASE clk IS
         WHEN '1'     => q <= d ;
         WHEN OTHERS  => NULL ;
      END CASE ;
   END PROCESS q_assignment ;
END sequential_4 ;
```

6.5.8 LOOP-Anweisung

Iterationsschleifen, d.h. mehrfach zu durchlaufende Anweisungsblökke, können mittels der LOOP-Anweisung realisiert werden. Dabei existieren die folgenden drei Alternativen: FOR-Schleife, WHILE-Schleife und Endlosschleife:

```
[loop_label :] FOR range LOOP
   ...
   ...     -- sequentielle Anweisungen
   ...
END LOOP [loop_label] ;

[loop_label :] WHILE condition LOOP
   ...
   ...     -- sequentielle Anweisungen
   ...
END LOOP [loop_label] ;

[loop_label :] LOOP
   ...
   ...     -- sequentielle Anweisungen
   ...
END LOOP [loop_label] ;
```

Die Schleifensteuerstrukturen *range* und *condition* werden wie bei der GENERATE-Anweisung verwendet. Für die Angabe von Bereichen (*range*) wird implizit eine Laufvariable deklariert.

6.5.9 EXIT- und NEXT-Anweisung

EXIT- und NEXT-Anweisungen dienen zum vorzeitigen Ausstieg aus Schleifenanweisungen bzw. zum vorzeitigen Abbruch des aktuellen Durchlaufs. Mit NEXT wird direkt zum Beginn des nächsten Schleifendurchlaufes gesprungen, mit EXIT wird die Schleife ganz verlassen. Da Schleifen, wie IF- und CASE-Anweisungen, auch verschachtelt aufgebaut sein können, kann mit EXIT und NEXT auch aus hierarchisch höher liegenden Schleifen ausgestiegen werden. Dazu muß das Label der entsprechenden Schleife angegeben werden:

```
NEXT [loop_label] [WHEN condition] ;
EXIT [loop_label] [WHEN condition] ;
```

Im folgenden werden zwei Beispiele für Modelle mit Schleifen gezeigt. Das Modell array_compare vergleicht zwei 9x9-Matrizen miteinander. Der Ausgang equal wird true, falls alle Elemente der Matrizen übereinstimmen.

```
PACKAGE array_pack IS
  TYPE bit_matrix IS ARRAY
    (integer RANGE <>, integer RANGE <>) OF bit ;
END array_pack ;
```

```
ENTITY array_compare IS
  PORT (a, b  : IN
        work.array_pack.bit_matrix(8 DOWNTO 0, 8 DOWNTO 0) ;
        equal : OUT boolean) ;
END array_compare ;
```

```
ARCHITECTURE behavioral OF array_compare IS
BEGIN
  cmp : PROCESS (a,b)
    VARIABLE equ : boolean ;
  BEGIN
    equ := true ;
    first_dim_loop : FOR k IN a'RANGE(1) LOOP
      second_dim_loop : FOR l IN a'RANGE(2) LOOP
        IF a(k,l) /= b(k,l) THEN      -- Elementvergleich
          equ := false ;
-- Ausstieg aus aeusserer Schleife beim ersten, ------------
-- nicht identischen Matrixelement -------------------------
          EXIT first_dim_loop ;
        END IF ;
      END LOOP ;
    END LOOP ;
    equal <= equ ;
  END PROCESS ;
END behavioral ;
```

Das folgende Modell (`n_time_left_shift`) rotiert den Eingangsvektor `in_table` um `n` Stellen nach links und gibt den neuen Vektor über den Port `out_table` aus.

```
ENTITY n_time_left_shift IS
  GENERIC (n : natural := 2) ;
  PORT (in_table  : IN  bit_vector (0 TO 31);
        out_table : OUT bit_vector (0 TO 31)) ;
END n_time_left_shift ;
```

```
ARCHITECTURE behavioral OF n_time_left_shift IS
BEGIN
  PROCESS (in_table)
    VARIABLE count : natural ;
    VARIABLE table : bit_vector (0 TO 32) ;
  BEGIN
    count := 0 ;
    table (0 TO 31) := in_table ;
    n_time_loop : WHILE count < n LOOP
      count := count + 1 ;
      table(table'HIGH) := table(table'LOW) ;
      left_shift_loop :
      FOR j IN table'LOW TO table'HIGH - 1 LOOP
        table(j) := table(j+1) ;
      END LOOP left_shift_loop;
    END LOOP n_time_loop ;
    out_table <= table (0 TO 31) ;
  END PROCESS ;
END behavioral ;
```

6.6 Unterprogramme

Ähnlich wie in höheren Programmiersprachen können in VHDL Unterprogramme in Form von Funktionen oder Prozeduren realisiert werden.

- **Funktionen** werden mit keinem, einem oder mehreren Argumenten aufgerufen und liefern einen Ergebniswert zurück. Der Funktionsaufruf kann an den gleichen Stellen in Ausdrücken und Anweisungen stehen, an denen der Typ des Ergebniswertes erlaubt ist. Funktionen werden explizit durch das Schlüsselwort RETURN unter Angabe des Rückgabewertes verlassen.
- **Prozeduren** werden mit einer Liste von Argumenten aufgerufen, die sowohl Eingaben als auch Ausgaben oder bidirektional sein können. Der Aufruf einer Prozedur ist ein eigenständiger Befehl (sequentiell oder nebenläufig). Prozeduren werden bis zu einer RETURN-Anweisung oder bis zum Ende (entsprechend einem Prozeß mit "sensitivity-list") abgearbeitet.

Funktionen und Prozeduren unterscheiden sich damit in folgenden Punkten:

	Funktionen	Prozeduren
Argumentmodi	`IN`	`IN, OUT, INOUT`
Argumentklassen	Konstanten, Signale	Konstanten, Signale, Variablen
Rückgabewerte	exakt einer	beliebig viele (auch 0)
Aufruf	in typkonformen Ausdrücken und Anweisungen	als eigenständige, sequentielle oder nebenläufige Anweisung
`RETURN`-Anweisung	obligatorisch	optional

Die Beschreibung der Funktionalität eines Unterprogramms kann (zusammen mit der Vereinbarung der Schnittstellen) in den Deklarationsteilen von Entity, Architektur, Block, Prozeß oder im Package Body stehen. Außerdem können Funktionen und Prozeduren selbst wieder in den Deklarationsteilen von anderen Funktionen und Prozeduren spezifiziert werden.

Weiterhin besteht die Möglichkeit, die Funktionalität (Unterprogrammdefinition) und die Schnittstellenbeschreibung (Unterprogrammdeklaration) zu trennen. Die Schnittstellenbeschreibung allein kann dann auch in Packages auftreten.

Eine solche Aufteilung bietet sich unter Ausnutzung der Abhängigkeiten beim Compilieren von Package und Package Body an: Die Schnittstellenbeschreibung wird im Package plaziert, während die Funktionalität erst im Package Body festgelegt wird. Eine nachträgliche Änderung der Funktionalität bedingt dann nur das Neucompilieren des Package Body. Die Design-Einheiten, die das Unterprogramm verwenden, müssen nicht neu übersetzt werden.

6.6.1 Funktionen

Wie bereits erwähnt, dienen Funktionen zur Berechnung eines Wertes und können in Ausdrücken und anderen Anweisungen direkt eingesetzt werden.

Für eine flexiblere Beschreibung (s.o.) bietet sich eine Aufteilung in Funktionsdeklaration (Schnittstellenbeschreibung) und -definition (Funktionalität) an.

6.6.1.1 Funktionsdeklaration

Die Funktionsdeklaration enthält eine Beschreibung der Funktionsargumente und des Funktionsergebnisses:

```
FUNCTION function_name
 [ ( { [arg_class_m] arg_name_m {,arg_name_n}
       : [IN] arg_type_m [:= def_value_m] ;}
       [arg_class_o] arg_name_o {,arg_name_p}
       : [IN] arg_type_o [:= def_value_o] ) ]
  RETURN result_type ;
```

Erlaubt sind also auch Funktionen ohne jegliches Argument.

Der **Funktionsname** (`function_name`) kann auch ein bereits definierter Funktionsname oder Operator sein. In diesem Fall spricht man von "Überladung" der Funktion. Diese Möglichkeit wird in Kapitel 11 ausführlich diskutiert.

Als **Argumentklasse** (`arg_class`) sind Signale und Konstanten erlaubt. Deren Angabe durch die Schlüsselwörter `SIGNAL` oder `CONSTANT` ist optional. Der Defaultwert ist `CONSTANT`.

Als **Argumenttypen** (`arg_type`) können in der Schnittstellenbeschreibung von Unterprogrammen neben eingeschränkten auch uneingeschränkte Typen verwendet werden.

```
FUNCTION demo (SIGNAL data1, data2 : IN integer;
               CONSTANT c1         : real) RETURN boolean;
```

6.6.1.2 Funktionsdefinition

Hier muß die Schnittstellenbeschreibung wiederholt werden. Das nachfolgende Schlüsselwort `IS` kennzeichnet den Beginn der Funktionsbeschreibung:

```
FUNCTION function_name
 [ ( { [arg_class_m] arg_name_m {,arg_name_n}
   : [IN] arg_type_m [:= def_value_m] ;}
   [arg_class_o] arg_name_o {,arg_name_p}
   : [IN] arg_type_o [:= def_value_o] ) ]
  RETURN result_type IS
   ...
   ... -- Deklarationsanweisungen
   ...
BEGIN
   ...
   ... -- sequentielle Anweisungen
   ... -- RETURN-Anweisung obligatorisch
   ... -- keine WAIT-Anweisung in Funktionen!
   ...
END [function_name] ;
```

Im Deklarationsteil von Funktionen können lokale Typen und Untertypen, Konstanten, Variablen, Files, Aliase, Attribute und Gruppen (✓93) deklariert werden. `USE`-Anweisungen und Attributdefinitionen sind dort ebenfalls erlaubt. Außerdem können im Deklarationsteil andere Prozeduren und Funktionen deklariert und definiert werden.

Die eigentliche Funktionsbeschreibung besteht aus sequentiellen Anweisungen (ausgenommen `WAIT`-Anweisung) und kann selbst wieder Unterprogrammaufrufe enthalten. Sämtliche Argumente können innerhalb der Funktion nur gelesen werden (Modus `IN`) und dürfen deshalb nicht verändert werden. Die Funktion muß an mindestens einer Stelle mit der `RETURN`-Anweisung verlassen werden:

```
RETURN result_value ;
```

Das Ergebnis der Funktion (Rückgabewert *`result_value`*) kann in Form von expliziten Wertangaben, mit Objektnamen oder durch Ausdrücke angegeben werden.

Die Vereinheitlichung der Rahmensyntax in ✓**93** führt zu folgender Syntaxalternative:

```
FUNCTION function_name ... IS
  ...
BEGIN
  ...
END [FUNCTION] [function_name] ;
```

Einige Beispiele für die Definition von Funktionen:

```
-- Umwandlung von bit in integer ('0' -> 0, '1' -> 1)
FUNCTION bit_to_integer (bit_a : bit) RETURN integer IS
BEGIN
    IF bit_a = '1' THEN RETURN 1 ;
    ELSE RETURN 0 ;
    END IF ;
END bit_to_integer ;
```

```
-- Zaehlen der Einsstellen in einem Bitvektor unbestimmter
-- Laenge (flexibles Modell). Abfrage der aktuellen Vektor-
-- laenge durch das Attribut RANGE.
FUNCTION count_ones (a : bit_vector) RETURN integer IS
    VARIABLE count : integer := 0 ;
BEGIN
    FOR c IN a'RANGE LOOP
      IF a(c) = '1' THEN count := count + 1 ;
      END IF ;
    END LOOP ;
    RETURN count ;
END count_ones ;
```

```
-- Exklusiv-NOR-Verknuepfung fuer allgemeine Bitvektoren
FUNCTION exnor (a, b : bit_vector) RETURN bit_vector IS
-- Normierung auf gleiche Indizierung der beiden Argumente
-- ueber Aliase c und d:
    ALIAS c : bit_vector(1 TO a'LENGTH) IS a ;
    ALIAS d : bit_vector(1 TO b'LENGTH) IS b ;
    VARIABLE result : bit_vector(1 TO a'LENGTH) ;
BEGIN
    ASSERT a'LENGTH = b'LENGTH
      REPORT "Different Length of vectors!" ;
    FOR k in c'RANGE LOOP
      IF (c(k) = '1' AND d(k) = '0') OR
         (c(k) = '0' AND d(k) = '1') THEN result(k) := '0' ;
      ELSE result(k) := '1' ;
      END IF ;
    END LOOP ;
    RETURN result ;
END exnor ;
```

6.6.1.3 *Funktionsaufruf*

Der Aufruf einer Funktion geschieht unter Angabe der aktuellen Argumentwerte in Form von expliziten Wertangaben, Objektnamen oder Ausdrücken. Der Aufruf muß an einer Stelle stehen, an der auch der Ergebnistyp der Funktion erlaubt ist. Für die Übergabe der Argumentwerte ist, wie bei der Port Map, die Zuordnung der Werte über ihre Position ("positional association") möglich:

```
function_name [(arg_1_value {, arg_n_value})]
```

oder kann durch die explizite Zuordnung zwischen den Funktionsargumenten (`arg_name`) und den Aufrufwerten ("named association") erfolgen:

```
function_name [( arg_name_1 => arg_1_value
               {, arg_name_n => arg_n_value})]
```

Die Kombination der beiden Aufrufmethoden ist unter Beachtung der gleichen Regeln wie bei der Port Map erlaubt. Wird ein Aufrufwert nicht angegeben, dann gilt der in der Funktionsdeklaration festgelegte Defaultwert (`def_value`).

Die oben definierten Funktionen lassen sich beispielsweise innerhalb von Signalzuweisungen aufrufen:

```
ENTITY functions IS
  PORT (in1, in2  : IN  bit_vector (8 DOWNTO 1):="11111111";
        exnor_out : OUT bit_vector (8 DOWNTO 1) );
END functions;

ARCHITECTURE behavioral OF functions IS
  SIGNAL x2  : bit_vector (2 DOWNTO 1);
  SIGNAL i,j : integer := 0;
BEGIN
  exnor_out <= exnor (in1, in2);
  x2 <= exnor (a => in1(2 DOWNTO 1), b => in2(2 DOWNTO 1));
  i <= bit_to_integer(bit_a => in1(8)) + count_ones(in2);
  j <= count_ones("111010010111001010010001");
END behavioral;
```

6.6.1.4 "Impure Functions" ✓93

Funktionen sollten nach der ursprünglichen Definition der Sprache VHDL keinerlei "Seiteneffekte" aufweisen, d.h. der Aufruf einer Funktion zu verschiedenen Zeitpunkten und an verschiedenen Stellen in VHDL-Modellen soll für gleiche Argumente auch gleiche Ergebnisse liefern. Auf Objekte, die nicht als Argumente oder in der Funktion selbst deklariert wurden, kann daher auch nicht zugegriffen werden.

Viele Hardwareentwickler empfanden diese im Englischen als "pure" bezeichneten Funktionen als eine große Einschränkung. Ihre Forderung nach Lockerung der Richtlinien für Funktionen wurde in ✓93 durch die Einführung von sog. "impure functions" berücksichtigt.

Herkömmliche, "pure" Funktionen werden wie bisher gehandhabt. Sie können nun aber optional mit dem Schlüsselwort PURE gekennzeichnet werden:

```
[PURE] FUNCTION function_name ... IS
  ...
BEGIN
  ...
END [FUNCTION] [function_name] ;
```

Die neue Klasse der "impure functions", gekennzeichnet durch das Wort IMPURE, kann nun auf globale Objekte wie z.B. Files zugreifen. Die Argumente müssen aber auch weiterhin den Modus IN besitzen. "Impure functions" werden wie folgt beschrieben (✓93):

```
IMPURE FUNCTION function_name ... IS
  ...
BEGIN
  ...
END [FUNCTION] [function_name] ;
```

Typische Anwendungsfälle für solche Funktionen sind z.B. die Erzeugung von Zufallszahlen oder das Lesen von Files.

6.6.2 Prozeduren

Prozeduren unterscheiden sich von Funktionen hauptsächlich durch ihren Aufruf und die Art der Argumente. Zusätzlich zum Modus IN sind auch OUT und der bidirektionale Modus INOUT erlaubt. Weiterhin sind neben Konstanten und Signalen auch Variablen als Argumentklasse gestattet.

6.6.2.1 Prozedurdeklaration

Die Prozedurdeklaration enthält die Beschreibung der an die Prozedur übergebenen Argumente (Modus IN und INOUT) und die von der Prozedur zurückgelieferten Ergebnisse (Modus INOUT und OUT):

```
PROCEDURE procedure_name
[({ [arg_class_m] arg_name_m {,arg_name_n} :
    [arg_modus_m] arg_type_m [:= def_value];}
    [arg_class_o] arg_name_o {,arg_name_p} :
     arg_modus_o  arg_type_o [:= def_value])];
```

Der Defaultwert des **Argumentmodus** (arg_modus) ist IN.

Die **Argumentklasse** (arg_class) kann neben SIGNAL und CONSTANT auch VARIABLE sein. Defaultwert der Klasse ist für den Modus IN CONSTANT, für die Modi OUT und INOUT VARIABLE.

```
PROCEDURE hello;
PROCEDURE d_ff ( CONSTANT delay  : IN  time := 2 ns ;
                 SIGNAL d, clk   : IN  bit ;
                 SIGNAL q, q_bar : OUT bit );
```

6.6.2.2 Prozedurdefinition

Entsprechend der Funktionsdefinition muß die Schnittstellenbeschreibung mit dem Schlüsselwort `IS` wiederholt werden:

```
PROCEDURE procedure_name
[({ [arg_class_m] arg_name_m {,arg_name_n} :
    [arg_modus_m] arg_type_m [:= def_value];}
    [arg_class_o] arg_name_o {,arg_name_p} :
     arg_modus_o  arg_type_o [:= def_value])]

    IS
    ...
    ... -- Deklarationsanweisungen
    ...
BEGIN
    ...
    ... -- sequentielle Anweisungen
    ... -- optional: RETURN-Anweisung
    ...
END [procedure_name] ;
```

Die Prozedurbeschreibung kann aus allen möglichen sequentiellen Anweisungen, einschließlich der `WAIT`-Anweisung, bestehen. Argumente vom Typ `IN` können innerhalb von Prozeduren nur gelesen werden; verändert werden dürfen nur Argumente des Typs `OUT` und `INOUT`. Prozeduren können explizit mit dem Schlüsselwort `RETURN` (ohne Argument) verlassen werden oder werden bis zum Ende abgearbeitet.

Die Vereinheitlichung der Rahmensyntax in ✓**93** führt bei optionaler Wiederholung des Schlüsselwortes `PROCEDURE` zu folgender Alternative für den Prozedurrahmen:

```
PROCEDURE procedure_name ... IS
   ...
BEGIN
   ...
END [PROCEDURE] [procedure_name] ;
```

Zwei Beispiele für Prozeduren:

```
-- Prozedur ohne Argumente
PROCEDURE hello IS
BEGIN
    ASSERT false REPORT "Hello world!" SEVERITY note ;
END hello ;
```

```
-- Beschreibung eines D-Flip-Flops innerhalb einer Prozedur
PROCEDURE d_ff (CONSTANT delay   : IN  time := 2 ns ;
                SIGNAL d, clk  : IN  bit ;
                SIGNAL q, q_bar: OUT bit ) IS
BEGIN
    IF clk = '1' AND clk'EVENT THEN
      q     <= d     AFTER delay ;
      q_bar <= NOT d AFTER delay ;
    END IF ;
END d_ff ;
```

6.6.2.3 Prozeduraufruf

Der Aufruf einer Prozedur erfolgt unter Angabe der Argumentwerte als eigenständige, sequentielle oder nebenläufige Anweisung. Wie beim Funktionsaufruf ist eine Angabe der Aufrufwerte durch Position ("positional association"):

```
procedure_name [(arg_1_value{, arg_n_value})];
```

oder eine explizite Zuordnung ("named association") möglich:

```
procedure_name  [( arg_name_1 => arg_1_value
                {,  arg_name_n => arg_n_value})];
```

Die Kombination der beiden Aufrufmethoden ist unter Beachtung der gleichen Regeln wie bei der Port Map erlaubt. Wird ein Aufrufwert

nicht angegeben, dann gilt der in der Prozedurdeklaration festgelegte Defaultwert (`def_value`).

Im nebenläufigen Fall des Prozeduraufrufes ist ein Label erlaubt:

```
[proc_call_label :] procedure_name [...] ;
```

Erläutert werden soll dies anhand einer alternativen Darstellung eines 4-bit-Registers, welches obige D-Flip-Flop-Prozedur verwendet:

```
ENTITY four_bit_register IS
  PORT (clk              : IN  bit ;
        in_d             : IN  bit_vector(3 DOWNTO 0) ;
        out_q, out_q_bar : OUT bit_vector(3 DOWNTO 0) ) ;
END four_bit_register ;
```

```
ARCHITECTURE with_procedures OF four_bit_register IS
  USE work.my_pack.all ; -- enthaelt Prozedur d_ff
BEGIN
   -- Varianten eines nebenlaeufigen Prozeduraufrufes
   d_ff_3 : d_ff (1.5 ns, in_d(3), clk,
                  out_q(3), out_q_bar(3)) ;
   d_ff_2 : d_ff (delay => 1.7 ns, d => in_d(2), clk => clk,
                  q => out_q(2), q_bar => out_q_bar(2)) ;
   d_ff_1 : d_ff (1.9 ns, in_d(1), clk,
                  q_bar => out_q_bar(1), q => out_q(1)) ;
   d_ff_0 : d_ff (q_bar => out_q_bar(0), clk => clk,
                  q => out_q(0), d => in_d(0)) ;
                  -- Defaultwert fuer delay: 2 ns
END with_procedures;
```

Bei einem nebenläufigen Prozeduraufruf wird die Prozedur immer dann aktiviert, wenn sich mindestens ein Argumentwert (Modus `IN` oder `INOUT`) ändert. Der Aufruf kann also auch als Prozeß interpretiert werden, welcher die genannten Argumente in seiner "sensitivity-list" enthält.

Diese einfache Möglichkeit, prozeßähnliche Konstrukte mehrfach in ein VHDL-Modell einzufügen, wird durch folgendes Beispiel verdeutlicht. Es handelt sich um zwei äquivalente Teile eines Modells, die ein Signal überwachen:

```
ENTITY anything_one IS
  PORT (sig_a, sig_b : IN bit; sig_c : OUT bit ) ;
  PROCEDURE monitoring (SIGNAL   a        : IN bit;
                        CONSTANT sig_name : IN string) IS
  BEGIN
    ASSERT false REPORT "Event on signal " & sig_name
    SEVERITY note ;
  END monitoring ;
BEGIN
  mon_sig_a : monitoring (sig_a, "anything_one:sig_a") ;
END anything_one ;
```

```
ARCHITECTURE behavioral OF anything_one IS
BEGIN
  mon_sig_b : monitoring (a => sig_b,
                          sig_name => "anything_one:sig_b") ;
  sig_c <= sig_a AND NOT sig_b ;
END behavioral ;
```

Diese erste Variante verwendet nebenläufige Prozeduraufrufe, die nicht nur in der Architektur, sondern auch im Anweisungsteil der Entity auftreten können (sofern es sich um sog. passive Prozeduren handelt).

Die nachfolgenden Varianten beschreiben das gleiche Verhalten mit Hilfe von ASSERT-Anweisungen und passiven Prozessen. Auch diese dürfen im Anweisungsteil einer Entity stehen:

```
ENTITY anything_two IS
  PORT (sig_a, sig_b : IN bit; sig_c : OUT bit ) ;
BEGIN
  mon_sig_a : PROCESS (sig_a)
  BEGIN
    ASSERT false REPORT "Event on signal anything_two:sig_a"
    SEVERITY note ;
  END PROCESS ;
END anything_two ;
```

```
ARCHITECTURE behavioral OF anything_two IS
BEGIN
  mon_sig_b : PROCESS (sig_b)
  BEGIN
    ASSERT false REPORT "Event on signal anything_two:sig_b"
    SEVERITY note ;
  END PROCESS ;
  sig_c <= sig_a AND NOT sig_b ;
END behavioral ;
```

```
ENTITY anything_three IS
  PORT (sig_a, sig_b : IN bit; sig_c : OUT bit ) ;
BEGIN
  mon_sig_a : ASSERT (sig_a'EVENT = false)
              REPORT "Event on signal anything_three:sig_a"
              SEVERITY note ;
END anything_three ;
```

```
ARCHITECTURE behavioral OF anything_three IS
BEGIN
  mon_sig_b : ASSERT (sig_b'EVENT = false)
              REPORT "Event on signal anything_three:sig_b"
              SEVERITY note ;
  sig_c <= sig_a AND NOT sig_b ;
END behavioral ;
```

7 Konfigurieren von VHDL-Modellen

Die Sprache VHDL bietet bei der strukturalen Beschreibung elektronischer Systeme ein hohes Maß an Flexibilität. So können unter anderem Modelle dadurch umkonfiguriert werden, daß man ihre interne Funktion austauscht, ihre Verdrahtung ändert oder die Modellparameter modifiziert.

"Konfigurieren" von VHDL-Modellen bedeutet im einzelnen also:

- die Auswahl der gewünschten Architekturalternative für eine Entity (d.h. Auswahl der internen Modellfunktion),
- die Auswahl der zu verwendenden Modelle für die einzelnen Instanzen bei strukturalen Modellen,
- das Verbinden von Signalen und Ports auf den unterschiedlichen Hierarchieebenen,
- die Zuweisung von Werten an die Parameter (Generics) der einzelnen Instanzen.

Diese Konfigurationsangaben werden in einer eigenen Design-Einheit, der "Configuration", zusammengefaßt. Änderungen an dieser Design-Einheit erfordern kein erneutes Compilieren des Gesamtmodells, so daß verschiedene Modellvarianten schnell untersucht werden können.

Daneben können auch in Deklarationsteilen von Architekturen und Blöcken und in den `GENERIC MAP`- und `PORT MAP`-Anweisungen der Komponenteninstantiierung Konfigurationsanweisungen stehen.

In vielen Fällen werden bei fehlenden Konfigurationskonstrukten Defaultwerte verwendet.

7.1 Konfiguration von Verhaltensmodellen

Bei Verhaltensmodellen ist nur eine Information erforderlich: Die Auswahl der gewünschten Architekturalternative für eine Schnittstellenbeschreibung. Dies wird in der Design-Einheit "Configuration" durch eine einfache Blockkonfigurationsanweisung vorgenommen:

```
CONFIGURATION conf_name OF entity_name IS
   ...
   ...  -- generelle USE-Anweisungen
   ...  -- Attributzuweisungen
   ...
   FOR arch_name    -- Architekturauswahl
   END FOR ;
END [CONFIGURATION] [conf_name] ;
```

Die optionale Wiederholung des Schlüsselwortes `CONFIGURATION` in der `END`-Anweisung ist nur in der neuen VHDL-Norm (✓93) gestattet.

7.2 Konfiguration von strukturalen Modellen

Bei strukturalen Modellen muß neben der Architekturauswahl jede instantiierte Komponente (jeder instantiierte Sockeltyp) konfiguriert werden, d.h. es muß festgelegt werden, in welcher Form ein existierendes VHDL-Modell in diese Sockelinstanzen eingesetzt wird. Dazu soll zunächst ein Blick auf die unterschiedlichen Ebenen bei strukturalen Modellen und deren Schnittstellen geworfen werden.

Das Konzept der strukturalen Modellierung über die Ebenen: Komponentendeklaration (Sockeltyp), Komponenteninstantiierung (Verwendung des Sockeltyps, "Sockelinstanz") und Komponentenkonfiguration (Einsetzen des Modells, "IC") weist folgende Schnittstellen auf:

- Schnittstelle zwischen dem eingesetzten Modell und der Komponente. Die Generics und Ports des ersteren werden dabei als "**formal**", die der letzteren als "**local**" bezeichnet. Hierfür benutzt man `GENERIC MAP`- und `PORT MAP`-Anweisungen in der Design-Einheit "Configuration".

- Schnittstelle zwischen der Komponenteninstanz und der Komponente. Die Signale, mit denen die Instanzen verdrahtet werden, und die Parameter, die an die Instanzen übergeben werden unter dem Begriff "**actual**" zusammengefaßt. Die Zuordnungen werden durch entsprechende `GENERIC MAP`- und `PORT MAP`-Anweisungen bei der Komponenteninstantiierung definiert.

In der "Configuration" wird mit hierarchisch geschachtelten `FOR-USE`-Anweisungen festgelegt, welche Modelle für die instantiierten Komponenten (Sockel) verwendet werden, wie die Ports verknüpft und welche Parameterwerte übergeben werden.

Man unterscheidet dabei zwischen Blockkonfigurationsanweisungen (für Architektur, `BLOCK`, `GENERATE`) und Komponentenkonfigurationsanweisungen (für die einzelnen Instanzen).

7.2.1 Konfiguration von Blöcken

Blöcke repräsentieren eine Hierarchieebene, die selbst wieder Komponenten und Blöcke enthalten kann. Dementsprechend können in einer Blockkonfiguration Komponentenkonfigurationen und auch weitere Blockkonfigurationen enthalten sein.

Auf oberster Ebene eines strukturalen Modells wird zunächst die gewünschte Architektur ausgewählt:

```
CONFIGURATION conf_name OF entity_name IS
   ...
   ...  -- generelle USE-Anweisungen
   ...  -- Attributzuweisungen
   ...
   FOR arch_name  -- Architekturauswahl
   ...  -- weitere Blockkonfigurationen
   ...  -- Komponentenkonfigurationen
   END FOR ;
END [CONFIGURATION] [conf_name] ;
```

Die Wiederholung des Schlüsselwortes `CONFIGURATION` am Ende der Design-Einheit ist nur in der neuen VHDL-Norm (✓**93**) gestattet.

Weitere Blockkonfigurationsanweisungen dienen zur näheren Beschreibung von GENERATE- und BLOCK-Anweisungen:

```
FOR block_name
   ...      -- weitere Blockkonfigurationen
   ...      -- Komponentenkonfigurationen
END FOR ;

FOR gen_name [(index_range)]
   ...      -- weitere Blockkonfigurationen
   ...      -- Komponentenkonfigurationen
END FOR ;
```

7.2.2 Konfiguration von Komponenten

Die Konfigurationsanweisungen für Komponenten stellen den Zusammenhang zwischen dem in der Architektur instantiierten Komponentensockel und dem darin einzusetzenden Modell her. Bei diesem Modell handelt es sich um ein bereits compiliertes Modell, das in der Bibliothek work oder in einer anderen Resource-Library abgelegt ist. Diese Modelle müssen also vor dem Übersetzen der Konfiguration angelegt und compiliert werden. Mit den von der Komponenteninstantiierung bekannten GENERIC MAP- und PORT MAP-Anweisungen werden die "local" und "formal" Ports und Generics verbunden.

Das jeweils einzusetzende Modell kann folgendermaßen beschrieben werden:

- durch Angabe seiner Konfiguration (conf_name):

```
FOR inst_name_1 {,inst_name_n} : comp_name
   USE CONFIGURATION conf_name
         [ GENERIC MAP (...) ]
         [ PORT MAP    (...) ] ;
END FOR ;
```

- oder durch den Namen seiner Schnittstellenbeschreibung (entity_name) mit gewünschter Architektur (arch_name):

```
FOR inst_name_1 {,inst_name_n} : comp_name
   USE ENTITY entity_name [(arch_name)]
        [ GENERIC MAP (...) ]
        [ PORT MAP    (...) ] ;
END FOR ;
```

- Möchte man einen Sockel (Komponenteninstanz) unbestückt lassen, genügt das Schlüsselwort `OPEN`.

```
FOR inst_name_1 {,inst_name_n} : comp_name
   USE OPEN ;
END FOR ;
```

Folgende Regeln werden angewandt, falls ein oder mehrere Teile dieser Angaben (`arch_name`, `GENERIC MAP`, `PORT MAP`) in der Konfiguration fehlen:

- Bei fehlender Architekturangabe (`arch_name`) wird ein Modell ohne Funktion bzw. nur mit den passiven Anweisungen der Entity eingesetzt.
- Stimmen Namen, Modi und Typen der Signale auf local- und formal-Ebene überein, so werden sie nach Namen miteinander verbunden (**Default**-`PORT MAP`).
- Jeder Parameter (Generic) in der Komponentendeklaration wird mit einem gleichnamigen Parameter der Entity verknüpft. Für den Parameterwert wird zuerst auf die `GENERIC MAP` der Komponenteninstantiierung zurückgegriffen. Wurden hier keine Parameterwerte angegeben, so gelten die Defaultwerte aus der Komponentendeklaration. Falls auch hier keine Werte für die Generics definiert sind, gelten die Defaultwerte aus der Entity (**Default**-`GENERIC MAP`).
- Fehlt die Komponentenkonfigurationsanweisung komplett, so wird diejenige Entity (incl. der Default-Architektur, d.h. der zuletzt compilierten Architektur der Entity) eingesetzt, deren Name mit dem Namen der Komponente übereinstimmt. Es gelten in diesem Fall die Regeln für die Default-`GENERIC MAP` und die Default-`PORT MAP`.

Wird als Komponentenkonfiguration ein Entity-Architecture-Paar angegeben und handelt es sich bei der beschriebenen Instanz ebenfalls um ein strukturales Modell, so muß dieses Modell durch weitere Konfigurationsanweisungen eine Ebene tiefer beschrieben werden.

Anstelle einzelner Instanzennamen (`inst_name`) können auch die Schlüsselwörter `OTHERS` (alle bisher noch nicht explizit beschriebenen Instanzen des Komponententyps `comp_name`) oder `ALL` (alle Instanzen des Typs `comp_name`) verwendet werden:

```
FOR OTHERS : comp_name USE ... ;
END FOR ;

FOR ALL :    comp_name USE ... ;
END FOR ;
```

Zwei Beispiele sollen diese nicht einfache Syntax verdeutlichen.

Die erste Konfiguration beschreibt den bereits eingeführten Halbaddierer (Abb. B-18), dessen Schnittstellenbeschreibung und strukturale Architektur im nachfolgenden aufgeführt ist. Die Komponenten (Sockeltypen) innerhalb dieses Modells tragen die Bezeichner `xor2` und `and2`. Die beiden Instanzen der Komponenten heißen `xor_instance` und `and_instance`. In diese Instanzen werden zwei Verhaltensmodelle aus der Bibliothek `work` mit unterschiedlichen Methoden eingesetzt. Das Modell für die erstgenannte Instanz wird durch die Angabe seiner Schnittstellenbeschreibung und seiner Architektur referenziert, das Modell für `and_instance` hingegen durch den Verweis auf dessen Konfiguration.

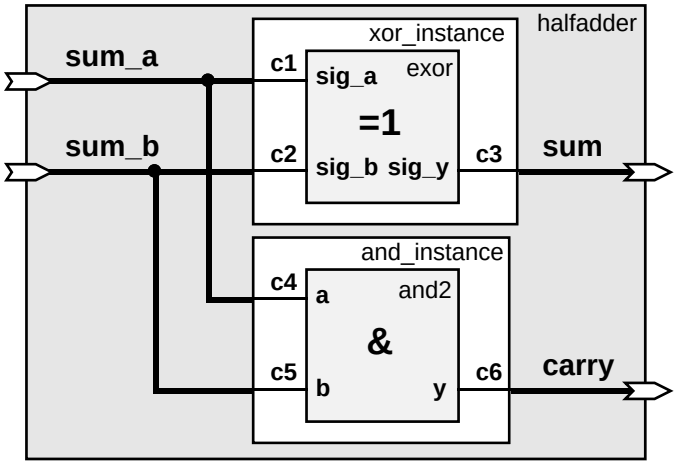

Abb. B-18: Struktur des Halbaddierers

```
ENTITY halfadder IS
  PORT (sum_a, sum_b : IN bit; sum, carry : OUT bit ) ;
END halfadder ;
```

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



```
CONFIGURATION ha_config OF halfadder IS
  -- Blockkonfiguration
  FOR structural
    -- Komponentenkonfiguration
    FOR xor_instance: xor2
      USE ENTITY work.exor (behavioral)
      PORT MAP (c1,c2,c3) ;
    END FOR ;
    -- Komponentenkonfiguration
    FOR and_instance: and2
      USE CONFIGURATION work.and2_config
      PORT MAP (a => c4, b => c5, y => c6) ;
    END FOR ;
  END FOR ;
END ha_config ;
```

Das folgende VHDL-Beispiel zeigt eine mögliche Konfiguration für die Architektur `structural_4` des im Kapitel 5 (Abb. B-6) eingeführten 3-2-AND-OR-INVERT-Komplexgatters. Hier werden Default-`PORT MAP` und Default-`GENERIC MAP` verwendet. Weitere Beispiele für Konfigurationen können den VHDL-Modellen auf der Diskette entnommen werden.

```
CONFIGURATION aoi_config_4 OF aoi IS
  FOR structural_4      -- Blockkonf. (Architekturauswahl)
    FOR nor_stage       -- Blockkonfiguration
      -- Komponentenkonfiguration
      FOR or_c : or2 USE ENTITY work.or2  (behavioral);
      END FOR;
    END FOR;
    FOR and_stage       -- Blockkonfiguration
      -- Komponentenkonfigurationen
      FOR and_b : and2 USE ENTITY work.and2 (behavioral);
      END FOR;
      FOR and_a : and3 USE ENTITY work.and3 (behavioral);
      END FOR;
    END FOR;
  END FOR;
END aoi_config_4;
```

7.2.3 Konfiguration von Komponenten außerhalb der Design-Einheit "Configuration"

Die Konfiguration von Komponenten kann nicht nur in der Design-Einheit "Configuration", sondern auch in der Architektur selbst vorgenommen werden. Die dazu erforderlichen Komponentenkonfigurationsanweisungen sind im Deklarationsteil derjenigen Ebene anzugeben, auf der die entsprechenden Komponenteninstantiierungen selbst stehen. Dies kann also auch der Deklarationsteil eines Blockes sein.

Folgende Architektur des 3-2-AND-OR-INVERT-Komplexgatters benötigt keine eigene Design-Einheit "Configuration":

```
ARCHITECTURE structural_6 OF aoi IS
  SIGNAL a_out, b_out, or_out : bit;  -- interne Signale
  ..
  ..  -- Komponentendeklarationen
  ..
  --Komponentenkonfigurationen
  FOR ALL : inv  USE ENTITY work.not1 (behavioral);
  FOR ALL : or2  USE ENTITY work.or2  (behavioral);
  FOR ALL : and2 USE CONFIGURATION work.and2_config;
  FOR ALL : and3 USE CONFIGURATION work.and3_config;
BEGIN
  ..
  ..  -- Komponenteninstantiierungen
  ..
END structural_6 ;
```

7.2.4 Inkrementelles Konfigurieren ✓93

Die VHDL-Syntax in der ursprünglichen Version (✓87) erlaubt nur eine einmalige Konfiguration einer strukturalen Architektur, entweder innerhalb der Architektur (oder eines Blockes) oder in einer Konfiguration. Beim ASIC-Entwurf verhindert diese Einschränkung allerdings eine einfache Rückführung von exakten Verzögerungszeiten nach der Layouterzeugung (sog. "Backannotation"). Es wäre wünschenswert, die Auswahl des einzusetzenden Modells und die Signalverbindungen von der Zuweisung der Parameter (in diesem Fall: Verzögerungszeiten) zu trennen.

Dies ist nun mit der überarbeiteten Norm (✓93) möglich. Das sog. "incremental binding" erlaubt die Trennung der einzelnen Teile der Komponentenkonfiguration. Es sind sogar mehrfache GENERIC MAP-Anweisungen möglich. Ein Beispiel erläutert diese Vereinbarung:

```
ARCHITECTURE structural_7 OF aoi IS     -- !!! VHDL'93-Syntax
  SIGNAL a_out, b_out, or_out : bit;
  ..
  ..  -- Komponentendeklarationen
  ..
  -- einheitliche Laufzeiten pro Gattertyp (ALL)
  FOR ALL : inv  USE ENTITY work.not1 (behavioral)
                 GENERIC MAP (0.75 ns, 0.7 ns) ;
  FOR ALL : or2  USE ENTITY work.or2 (behavioral)
                 GENERIC MAP (1.6 ns, 1.5 ns) ;
  FOR ALL : and2 USE CONFIGURATION work.and2_config ;
  FOR ALL : and3 USE CONFIGURATION work.and3_config ;
BEGIN
  ..
  ..  -- Komponenteninstantiierungen
  ..
END structural_7 ;
```

```
CONFIGURATION aoi_config_7 OF aoi IS    -- !!! VHDL'93-Syntax
  FOR structural_7
    -- hier koennen instanzenspezifische Laufzeiten stehen
    FOR inv_d : inv  GENERIC MAP (0.9 ns, 0.8 ns); END FOR;
    FOR or_c  : or2  GENERIC MAP (1.8 ns, 1.7 ns); END FOR;
    FOR and_b : and2 GENERIC MAP (1.3 ns, 1.9 ns); END FOR;
    FOR and_a : and3 GENERIC MAP (1.4 ns, 2.0 ns); END FOR;
  END FOR;
END aoi_config_7;
```

Ein Simulationsaufruf nur mit den Konfigurationen aus der Architektur verwendet beispielsweise geschätzte Werte oder die Defaultwerte für die Verzögerungszeiten. Die Simulation durch Angabe der Konfiguration aoi_config_7 dagegen weist den Parametern der jeweiligen Gatter neue (z.B. aus dem Layout ermittelte) Verzögerungszeiten zu.

8 Simulationsablauf

Da die nebenläufigen Anweisungen i.d.R. auf einem Prozessor, d.h. nur "quasi" parallel ablaufen, ist ein spezieller Simulationsablauf nötig, der die Nebenläufigkeit von VHDL "nachbildet".

Dazu sind die Zusammenhänge zwischen nebenläufigen und sequentiellen Anweisungen und der Unterschied zwischen Signal- und Variablenzuweisungen näher zu beleuchten.

Sequentielle Anweisungen stehen in Prozessen, Funktionen oder Prozeduren. Funktionen und Prozeduren werden durch spezielle Aufrufe aktiviert, die selbst wieder sequentiell oder nebenläufig sind. Prozesse gelten als nebenläufige Anweisung. Sie werden durch eine **Liste sensitiver Signale** im Prozeß-Kopf oder durch `WAIT`-**Anweisungen** innerhalb des Prozesses gesteuert. Die Prozesse werden immer dann aktiviert und ausgeführt, wenn auf mindestens einem der Signale aus der "sensitivity list" ein Ereignis auftritt. Ist keine "sensitivity list" vorhanden, so wird ein Prozeß dadurch reaktiviert, daß die Bedingung der `WAIT`-Anweisung erfüllt wird.

8.1 Delta-Zyklus

Während der Simulation schreitet die (Simulations-)Zeit fort, wobei jeder Zeitpunkt, an dem Transaktionen eingetragen sind, bearbeitet wird. Ein Simulationszeitpunkt besteht dabei im allgemeinen aus mehreren Zyklen, die um eine infinitesimal kleine Zeit, mit "Delta" (Δ) bezeichnet, versetzt sind. Jeder dieser Delta-Zyklen besteht wiederum aus zwei aufeinanderfolgenden Phasen:

1. **Prozeß-Ausführungsphase** ("process evaluation"):
   Hierbei werden alle aktiven Prozesse bis zur END-Anweisung bzw. bis zur nächsten WAIT-Anweisung abgearbeitet. Dies beinhaltet das Ausführen aller enthaltenen Anweisungen bis auf die

Signalzuweisungen, also insbesondere auch die Zuweisung von Variablen.

2. **Signalzuweisungsphase** ("signal update"):
   Nach Ausführung des oder der für dieses "Delta" aktiven Prozesse werden die in diesen Prozessen zugewiesenen Signaländerungen durchgeführt. Dadurch können weitere Prozesse oder nebenläufige Signalzuweisungen aktiviert werden. In diesem Fall wird der Zyklus ein "Delta" später wieder von vorne durchlaufen.

Zu Beginn der Bearbeitung eines Simulationszeitpunktes t_n werden zunächst die dort vorgesehenen Signalzuweisungen durchgeführt, danach alle sensitiven Prozesse ausgeführt, um anschließend die von diesen Prozessen ausgelösten Signalzuweisungen durchzuführen, usw. Dieser Ablauf wird an einem Zeitpunkt t_n solange wiederholt, bis sich ein stabiler Zustand einstellt, d.h. bis sich keine neue Signalzuweisung mehr ergibt und kein weiterer Prozeß aktiviert wird. Man erhält ein Ablaufschema, das in folgender Art und Weise dargestellt werden kann:

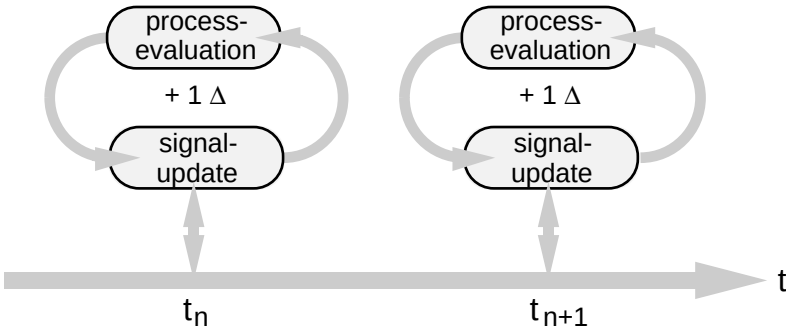

Abb. B-19: Simulationsablauf

Es ist an dieser Stelle zu beachten, daß auch die nebenläufigen Anweisungen als Prozesse interpretiert werden. Letztendlich läßt sich nämlich jede nebenläufige Anweisung durch einen äquivalenten Prozeß ersetzen. Die Signale auf der rechten Seite einer nebenläufigen Signalzuweisung werden bei dieser Umsetzung in die "sensitivity-list" aufgenommen. Für folgende nebenläufige Signalzuweisung:

```
csa: sig_b <= '1', '0' AFTER 2 ns WHEN sel = 1 ELSE
              '0', '1' AFTER 3 ns WHEN sel = 2 ELSE
              'Z';
```

lautet die äquivalente Darstellung als Prozeß:

```
csa: PROCESS (sel)
BEGIN
  IF    sel = 1 THEN sig_b <= '1', '0' AFTER 2 ns;
  ELSIF sel = 2 THEN sig_b <= '0', '1' AFTER 3 ns;
  ELSE               sig_b <= 'Z';
  END IF;
END PROCESS csa;
```

Damit wird ersichtlich, daß nebenläufige Signalzuweisungen immer durch Ereignisse auf den Signalen der rechten Seite aktiviert werden.

8.2 Zeitverhalten von Signal- und Variablenzuweisungen

Es sei noch einmal explizit erwähnt, daß Signale zwar beim Abarbeiten der entsprechenden sequentiellen Signalzuweisungen projektiert werden - was soviel heißt wie ein Vormerken des neuen Signalwertes ("scheduling") - aber erst am Ende des Prozesses oder beim Erreichen der nächsten `WAIT`-Anweisung den neuen Wert annehmen.

Besondere Beachtung erfordern deshalb Prozesse, die gemischt Variablen- und Signalzuweisungen verwenden und gegenseitige Abhängigkeiten zwischen Variablen und Signalen aufweisen. Auch bei der Prozeßkommunikation ist darauf zu achten, daß sich die triggernden Signale zum richtigen Zeitpunkt ändern.

Ein kleines Beispiel soll verdeutlichen, daß Signalzuweisungen innerhalb von Prozessen sich nicht wie herkömmliche sequentielle Anweisungen verhalten:

```
ARCHITECTURE number_one OF example IS
  SIGNAL a, b : integer := 0;
BEGIN
  a <= 1 AFTER 1 ns, 2 AFTER 2 ns, 3 AFTER 3 ns;
  PROCESS (a)           -- Aufruf bei Ereignis auf Signal a
    VARIABLE c : integer := 0;
  BEGIN
    b <= a + 2;         -- Signalzuweisung
    c := 2 * b;         -- Variablenzuweisung
  END PROCESS;
END number_one;
```

Der Prozeß wird (nach dem initialen Durchlauf) zum Zeitpunkt 1 ns + 1Δ durch ein Ereignis auf dem Signal a getriggert. Das Signal b wird erst nach Ende des Prozeßdurchlaufes in der Signalzuweisungsphase den Wert 3 erhalten. Zum Zeitpunkt der Variablenzuweisung liegt noch der alte Wert von b (2), herrührend von der Initialisierungsphase, vor. Die Variable c wird also zum Zeitpunkt 1 ns + 1Δ nicht wie erwartet den Wert 6, sondern den Wert 4 annehmen. Der gewünschte Wert 6 wird erst beim nächsten Prozeßdurchlauf erreicht; in diesem Fall zum Zeitpunkt 2 ns + 1Δ.

Dieses Problem kann gelöst werden, indem die arithmetischen Operationen nur mit Variablen durchgeführt werden, da diese den neuen Wert ohne jegliche Verzögerung annehmen:

```
ARCHITECTURE number_four OF example IS
  SIGNAL a : integer := 0;
BEGIN
  a <= 1 AFTER 1 ns, 2 AFTER 2 ns, 3 AFTER 3 ns;
  PROCESS (a)           -- Aufruf bei Ereignis auf Signal a
    VARIABLE b, c : integer := 0;
  BEGIN
    b := a + 2;         -- Variablenzuweisung
    c := 2 * b;         -- Variablenzuweisung
  END PROCESS;
END number_four;
```

Die Variable b nimmt hier sofort den Wert 3 an, so daß c wie gewünscht den Wert 6 erhält.

Alternativ zu dieser Lösung könnte neben a auch das Signal b in die "sensitivity-list" des Prozesses aufgenommen werden:

```
ARCHITECTURE number_three OF example IS
  SIGNAL a, b : integer := 0;
BEGIN
  a <= 1 AFTER 1 ns, 2 AFTER 2 ns, 3 AFTER 3 ns;
  PROCESS (a, b) -- Aufruf bei Ereignis auf Signal a oder b
    VARIABLE c : integer := 0;
  BEGIN
    b <= a + 2;         -- Signalzuweisung
    c := 2 * b;         -- Variablenzuweisung
  END PROCESS;
END number_three;
```

Der Prozeß wird bei dieser Variante zum Zeitpunkt 1 ns + 1Δ zuerst durch das Ereignis auf dem Signal a aktiviert. Anschließend aktiviert die Änderung des Signals b von 2 auf 3 erneut den Prozeß (bei 1 ns + 2Δ). Beim zweiten Prozeßdurchlauf wird der Wert der Variablen c mit dem inzwischen aktualisierten Wert von b richtig berechnet.

Eine dritte Variante zur Lösung des Problems unter Verwendung von WAIT-Anweisungen findet sich auf der beiliegenden Diskette.

Da Signale nicht im gleichen Delta-Zyklus neu zugewiesen werden müssen, auch wenn sie am Ende des Simulationszeitpunktes gleiche Werte besitzen, können Assertions, die auf Gleichheit zweier Signale prüfen, "falsche" Fehlermeldungen produzieren.

8.3 Aktivierung zum letzten Delta-Zyklus ✓93

In ✓93 wurde der Begriff "postponed" im Zusammenhang mit Prozessen, Prozeduraufrufen, Signalzuweisungen und Assertions eingeführt. Er besagt, daß die entsprechend markierten VHDL-Anweisungen erst im definitiv letzten Delta-Zyklus eines Simulationszeitpunktes aktiviert werden.

8.3.1 Prozesse

Die erweiterte Syntax der Prozeßanweisung einschließlich des optionalen Schlüsselwortes `POSTPONED` lautet nun (✓93) :

```
[process_label :] [POSTPONED] PROCESS
      [(sig_name_1 {, sig_name_n})] [IS]
  ...
BEGIN
  ...
END [POSTPONED] PROCESS [process_label] ;
```

Die Kennzeichnung eines Prozesses als `POSTPONED` hat folgende Konsequenzen:

- Da der Prozeß erst zum letzten Delta-Zyklus aktiviert wird, befinden sich alle Signale in einem für diesen Simulationszeitpunkt stabilen Zustand.
- Derartige Prozesse dürfen keine neuen Delta-Zyklen mehr verursachen. Dies bedeutet insbesondere:
  - es dürfen keine unverzögerten Signalzuweisungen enthalten sein.
  - es dürfen keine "`WAIT FOR 0 fs;`"-Anweisungen enthalten sein,
- Falls der Prozeß auf mehrere Signale sensitiv ist, kann im letzten Delta-Zyklus durch Attribute wie z.B. `EVENT` nicht mehr festgestellt werden, welches Signal den Prozeß aktiviert hat.

8.3.2 Assertions

Gerade bei Assertions ist es wichtig, daß evtl. zu überprüfende Signale einen stabilen Zustand erreicht haben. Unnötige Fehlermeldungen werden so vermieden. Bei nebenläufigen Assertions lautet die Syntax mit dem Schlüsselwort `POSTPONED` (✓93):

```
[assert_label :] [POSTPONED] ASSERT condition
                 [REPORT "message_string"]
                 [SEVERITY severity_level] ;
```

8.3.3 Signalzuweisungen

Auch bei nebenläufigen Signalzuweisungen ist eine Verlagerung in den letzten Delta-Zyklus durch das Schlüsselwort POSTPONED möglich. Signalzuweisungen ohne jegliche Verzögerung sind dabei allerdings nicht erlaubt. Die entsprechende Syntax am Beispiel der nicht bedingten Signalzuweisung lautet (✓**93**):

```
[assignment_label :] [POSTPONED] sig_name
     <= [TRANSPORT] value_1 [AFTER time_1]
                 {, value_n  AFTER time_n } ;
```

8.3.4 Prozeduraufrufe

Ein weiterer Befehl, der in den letzten Delta-Zyklus verlegt werden kann, ist der nebenläufige Aufruf einer Prozedur nach der folgenden Syntax mit dem Schlüsselwort POSTPONED (✓**93**):

```
[proc_call_label :]
                  [POSTPONED] procedure_name
                  [(argument_values)] ;
```

9 Besonderheiten bei Signalen

Neben den obigen Erläuterungen über nebenläufige und sequentielle Signalzuweisungen gibt es noch einige Besonderheiten bei der Handhabung von Signalen in VHDL, die hier erwähnt werden sollen.

9.1 Signaltreiber und Auflösungsfunktionen

Ein Signal ist ein Informationsträger, der verschiedene Signalwerte annehmen kann. Im Falle herkömmlicher Signale, wie sie in vorhergehenden Kapiteln erläutert wurden, wird dieser Wert durch Signalzuweisungen festgelegt. Dahinter verbirgt sich aber ein Mechanismus, der erst bei mehrfachen, gleichzeitigen Zuweisungen an ein und dasselbe Signal wichtig wird: Das Konzept von Signaltreibern und Auflösungsfunktionen ("resolution functions").

Soll z.B. ein Bus oder eine bidirektionale Verbindung aufgebaut werden, sind jeweils mehrere Signalquellen für ein Signal vorzusehen, die in gewisser Weise einen Teil zum resultierenden Signalwert auf dem Bus oder der bidirektionalen Leitung beitragen. Diese einzelnen Signalquellen werden in VHDL als Signaltreiber ("driver") bezeichnet.

Signaltreiber werden durch VHDL-Signalzuweisungen erzeugt. Dies geschieht oft auch unbeabsichtigt durch mehrfache Signalzuweisungen. Das Signal `w` im nachfolgenden Beispiel besitzt durch die beiden Signalzuweisungen zwei Signaltreiber, die u.U. auch gleichzeitig aktiv sein können, da die Anweisungen nebenläufig (parallel) ablaufen.

```
ARCHITECTURE arch_one OF anything IS
BEGIN
   w <= y AFTER 25 ns WHEN sel = 1 ELSE '0' ;
   w <= z AFTER 30 ns WHEN sel = 2 ELSE '0' ;
END arch_one ;
```

Bei mehreren gleichzeitig aktiven Signaltreibern ist es möglich, daß von den einzelnen Treibern unterschiedliche Signalwerte zugewiesen werden. Solche Fälle werden durch die sog. Auflösungsfunktionen geregelt. Abb. B-20 zeigt schematisch die Aufgabe dieser Funktionen:

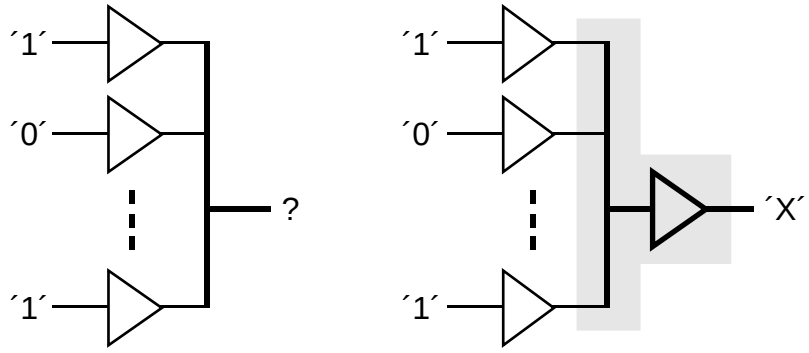

Abb. B-20: Wirkung von Auflösungsfunktionen

Bei Deklaration eines eigenen Logiktyps und Verwendung mehrfacher Treiber muß auch die entsprechende Auflösungsfunktionen beschrieben werden. Bei der Deklaration werden alle Signale und Ports gekennzeichnet, auf die eine solche Funktion angewandt werden soll; man spricht dann von aufgelösten Signalen (sog. "resolved signals"). Dies kann auf zwei Arten erfolgen:

- durch Deklaration eines Untertyps (`res_type_name`), der vom Basistyp (`unres_type_name`) durch Angabe der Auflösungsfunktion (`res_fct_name`) abgeleitet wird:

```
SUBTYPE  res_type_name
          IS res_fct_name unres_type_name ;
SIGNAL   res_sig_name_1 {, res_sig_name_n} :
          res_type_name [:= def_value] ;
```

- durch Deklaration des Signals direkt unter Angabe der Auflösungsfunktion:

```
SIGNAL   res_sig_name_1 {, res_sig_name_n} :
          res_fct_name unres_type_name
         [:= def_value] ;
```


Die Auflösungsfunktion wird bei jeder Signalzuweisung implizit aufgerufen und berechnet den neuen Signalwert aus den Beiträgen der einzelnen Treiber. Theoretisch können alle verwendeten Ports und Signale mit aufgelöstem Typ deklariert werden. Der implizite Aufruf der entsprechenden Funktion bei jeder Signalzuweisung erhöht die Simulationszeit jedoch ganz erheblich.

Auflösungsfunktionen müssen beliebig viele Treiber berücksichtigen können und einen Defaultwert zurückliefern, falls keiner der Treiber aktiv einen Signalpegel erzeugt (siehe nachfolgenden Abschnitt über kontrollierte Signale und `DISCONNECT`-Anweisung).

Am Beispiel einer 4-wertigen Logik `mvl4` mit den möglichen Signalwerten `'X'`, `'0'`, `'1'`, `'Z'` soll das typische Vorgehen bei der Deklaration von aufgelösten Signalen und Auflösungsfunktionen erläutert werden:

```
PACKAGE fourval IS
  TYPE mvl4 IS ('X', '0', '1', 'Z');
  TYPE mvl4_vector IS ARRAY (natural RANGE <>) OF mvl4;
  FUNCTION resolved (a: mvl4_vector) RETURN mvl4;
  SUBTYPE mvl4_r IS resolved mvl4;
  TYPE mvl4_r_vector IS ARRAY (natural RANGE <>) OF mvl4_r;
END fourval;
```

Nach Deklaration des Basistyps `mvl4` und des entsprechenden Vektors `mvl4_vector` wird die Funktion `resolved` bekanntgegeben. Ihre Funktionalität wird im nachfolgend aufgeführten Package Body beschrieben. Daraufhin kann ein aufgelöster Untertyp `mvl4_r` abgeleitet werden. Der aufgelöste Vektortyp `mvl4_r_vector` besteht wiederum aus aufgelösten Einzelelementen.

```
PACKAGE BODY fourval IS
  FUNCTION resolved (a: mvl4_vector) RETURN mvl4 IS
    VARIABLE result : mvl4 := 'Z';
    -- Defaultwert: 'Z': schwaechster Logikwert
  BEGIN
      FOR m in a'RANGE LOOP
        IF  a(m) = 'X' OR
           (a(m) = '1' AND result = '0') OR
           (a(m) = '0' AND result = '1') THEN RETURN 'X';
        ELSIF
           (a(m) = '0' AND result = 'Z') OR
           (a(m) = '1' AND result = 'Z') THEN
             result := a(m);
        END IF;
      END LOOP;
    RETURN result;
  END resolved;
END fourval;
```

Folgendes Beispiel zeigt die Anwendung des Typs `mvl4_r`:

```
ENTITY resolve IS
  PORT (sel : IN positive; x: OUT mvl4_r);
END resolve;

ARCHITECTURE behavioral OF resolve IS
BEGIN
  x <= '0' WHEN sel = 1 ELSE 'Z';
  x <= '1' WHEN sel = 2 ELSE 'X';
END behavioral;
```

Falls `sel` auf `2` wechselt, würde die Auflösungsfunktion `resolved` für das Signal `x` den Wert `'1'` aus den beiden konkurrierenden Werten `'1'` und `'Z'` bestimmen. Wechselt `sel` dagegen auf `1`, so ergibt sich der Wert `'X'` für das Signal `x`.

Das explizite Abschalten eines einzelnen Treibers für ein aufgelöstes Signal kann unter Kenntnis der Auflösungsfunktion durch Angabe des Defaultwertes (in diesem Fall `'Z'`) erfolgen:

```
ARCHITECTURE behavioral OF demo IS
BEGIN
   y <= '1' AFTER 5 ns,       -- aktive '1' treiben
        'Z' AFTER 40 ns ;     -- 'Z' entspricht Abschalten
END behavioral ;
```

9.2 Kontrollierte Signalzuweisungen

VHDL bietet neben den bedingten Zuweisungen eine weitere Möglichkeit, Signalzuweisungen zu steuern, die sog. kontrollierten Signalzuweisungen ("guarded signal assignment"). Dies sind nebenläufige Signalzuweisungen, die durch das Schlüsselwort `GUARDED` gekennzeichnet werden:

```
[label :] sig_name <= GUARDED sig_waveform ;
```

Die kontrollierende Bedingung, die erfüllt sein muß, damit solche Signalzuweisungen auch durchgeführt werden, nennt man "*guard_expression*". Die dadurch kontrollierten Signalzuweisungen müssen innerhalb eines Blockes stehen, der die Kontrollbedingung im Blockrahmen enthält. Die entsprechende Blocksyntax lautet:

```
block_label : BLOCK (guard_expression)
  ...
BEGIN
  ...
END BLOCK ;
```

Dieses Konstrukt wird durch den VHDL-Compiler in eine äquivalente `IF`-Anweisung umgewandelt:

```
IF guard_expression THEN
  sig_name <= sig_waveform ;
END IF ;
```

Diese Anweisung steht in einem äquivalenten Prozeß mit gleichem Label (`block_label`), der eine entsprechende `WAIT ON`-Anweisung am Ende mit allen Signalen enthält, die im vorgesehenen Signalverlauf (*sig_waveform*) oder in der Bedingung (*guard_expression*) auftreten.

Mit kontrollierten Signalzuweisungen kann z.B. auch die Funktionalität des bereits erwähnten D-Latch realisiert werden:

```
ARCHITECTURE concurrent_5 OF latch IS
BEGIN
  q_assignment : BLOCK (clk = '1')
  BEGIN
    q <= GUARDED d;
  END BLOCK;
END concurrent_5;
```

9.3 Kontrollierte Signale

Die im vorhergehenden Abschnitt besprochenen kontrollierten Signalzuweisungen arbeiten mit herkömmlichen Signalen. Werden die in solchen Anweisungen neu zugewiesenen Signale aber explizit als kontrollierte Signale ("guarded signals") deklariert, so erfolgt bei einer nicht erfüllten *guard_expression* das Abschalten des entsprechenden Signaltreibers.

Die Kennzeichnung eines Signals als kontrolliertes Signal erfolgt durch zusätzliche Angabe eines der Schlüsselwörter `REGISTER` oder `BUS` bei der Deklaration. Kontrollierte Signale müssen einen aufgelösten Typ besitzen:

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

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

Auf solche Signale angewandte kontrollierte Signalzuweisungen haben folgende Bedeutung (gezeigt anhand der äquivalenten `IF`-Anweisung):

```
IF guard_expression THEN
   sig_name <= sig_waveform ;
ELSE
   sig_name <= NULL ;
END IF ;
```


Die Zuweisung von NULL auf das Signal bedeutet nichts anderes, als daß der Treiber dieser Signalzuweisung abgeschaltet wird. Für das resultierende (aufgelöste) Signal hat dies folgende Konsequenzen:

- Wurden nicht alle Treiber abgeschaltet, so wird das resultierende Signal nur anhand der nicht abgeschalteten Treiber ermittelt,
- wurden alle Treiber abgeschaltet, so wird im Falle der Signaldeklaration als REGISTER der zuletzt vorhandene Signalwert beibehalten,
- wurden alle Treiber abgeschaltet, so wird im Falle der Signaldeklaration als BUS der in der "resolution function" angegebene Defaultwert angenommen.

Das Abschalten des Signaltreibers erfolgt unmittelbar, d.h. ohne Verzögerung, nachdem die *guard_expression* den Wert false angenommen hat, es sei denn, es wurde für das Signal im Anschluß an dessen Deklaration als kontrolliertes Signal eine explizite Verzögerungszeit (*time_expr*) durch die sog. DISCONNECT-Anweisung vereinbart:

```
DISCONNECT sig_name_1 {, sig_name_n} :
     res_type_name AFTER time_expr ;

DISCONNECT OTHERS :
     res_type_name AFTER time_expr ;

DISCONNECT ALL :
     res_type_name AFTER time_expr ;
```

Jedes kontrollierte Signal darf nur eine DISCONNECT-Anweisung erhalten. Die Schlüsselwörter OTHERS und ALL beschreiben alle noch nicht explizit mit einer Abschaltverzögerung versehenen bzw. alle Signale des aufgelösten Typs res_type_name.

Die Handhabung von kontrollierten Signalen soll anhand eines Beispiels verdeutlicht werden:

```
USE work.fourval.ALL;
ENTITY mux_2_1 IS
  PORT (in_signals : IN  mvl4_vector (1 TO 2);
        choice     : IN  integer;
        out_signal : OUT mvl4_r BUS );
END mux_2_1;
```

```
ARCHITECTURE with_guards OF mux_2_1 IS
  DISCONNECT out_signal : mvl4_r AFTER 25 ns;
BEGIN

  choice_1 : BLOCK (choice = 1)
  BEGIN
    out_signal <= GUARDED in_signals(1) AFTER 20 ns;
  END BLOCK;

  choice_2 : BLOCK (choice = 2)
  BEGIN
    out_signal <= GUARDED in_signals(2) AFTER 18 ns;
  END BLOCK;

END with_guards;
```

Wechselt das Eingangssignal `choice` auf `1`, so wird der Signaltreiber des ersten Blockes aktiv und gibt das erste Eingangssignal nach 20 ns an den Ausgang weiter. Wechselt `choice` auf `2`, so wird durch den zweiten Block das zweite Eingangssignal nach 18 ns ebenfalls auf den Ausgang gelegt. Da der erste Signaltreiber aber erst nach 25 ns abschaltet, sind für 7 ns zwei Treiber aktiv. Das resultierende Signal wird durch die Auflösungsfunktion ermittelt. Wechselt `choice` auf einen Wert ungleich `1` oder `2`, so schaltet der zuletzt aktive Treiber nach 25 ns ab und das Ausgangssignal wechselt in den Defaultwert (in diesem Falle `'Z'`), da es als `BUS` deklariert ist. Bei einer Deklaration als `REGISTER` würde der letzte Signalwert erhalten bleiben.

10 Gültigkeit und Sichtbarkeit

Um bei großen Entwürfen die Übersicht über die Vielzahl von auftretenden Objekten, Typen, Unterprogrammen usw. nicht zu verlieren, ist es sinnvoll, eine eindeutige Struktur der Namensgebung einzuführen und konsequent beizubehalten. Dafür soll ein Blick auf die Regeln geworfen werden, nach denen entschieden wird, ob ein Objekt in einer Anweisung verwendet werden kann und welches VHDL-Element effektiv zum Einsatz kommt.

10.1 Gültigkeit

Der Gültigkeitsbereich eines Objektes oder eines VHDL-Elementes hängt im wesentlichen vom Ort der Deklaration ab und umschließt alle hierarchisch tieferliegenden Design- und Syntax-Einheiten nach folgender Schichtung:

- Deklarationen im Package gelten für alle Design-Einheiten, die das Package verwenden.
- Deklaration im Deklarationsteil einer Entity gelten für alle dieser Entity zugehörenden Architekturen und die darin enthaltenen Blöcke und Anweisungen.
- Deklaration im Deklarationsteil einer Architektur haben für alle enthaltenen Blöcke und Anweisungen Gültigkeit.
- Erfolgt die Deklaration in einem Block, so umfaßt der Gültigkeitsbereich alle im Block enthaltenen Anweisungen.
- Deklaration innerhalb eines Prozesses gelten nur für die im Prozeß enthaltenen Anweisungen.
- Deklarationen in einer Schleife (Laufvariable) oder im Deklarationsteil einer Funktion bzw. einer Prozedur haben nur die eingeschränkte Gültigkeit für diese spezielle Anweisung.

10.2 Sichtbarkeit

Entscheidend dafür, ob ein VHDL-Element in einer Anweisung verwendet werden kann ist, ob es am Ort des Auftretens sichtbar ist. Sichtbarkeit bedeutet dabei entweder direkte Sichtbarkeit oder Sichtbarkeit durch Auswahl.

10.2.1 Direkte Sichtbarkeit

Der Bereich, in dem VHDL-Elemente direkt sichtbar sind, umfaßt in der Regel ihren Gültigkeitsbereich nach der Deklaration, es sei denn, das VHDL-Element ist verborgen.

Verborgenheit liegt vor, falls an einer Stelle mehrere gleichartige, gültige VHDL-Elemente existieren, die an hierarchisch verschiedenen Orten deklariert wurden. In diesem Fall maskiert oder verbirgt das lokal (hierarchisch niedriger) deklarierte VHDL-Element die anderen.

Sind gleichzeitig mehrere VHDL-Elemente mit gleichem Namen direkt sichtbar, werden die Regeln der Überladung (siehe Kapitel 11) angewandt. Treffen die Regeln der Überladung nicht zu, kann die Anweisung nicht ausgeführt werden.

10.2.2 Sichtbarkeit durch Auswahl

Nicht direkt sichtbare VHDL-Elemente können durch die evtl. hierarchische, vorangestellte Angabe des Deklarationsortes (nach den Regeln der sog. "selected names") angesprochen werden. Das Zeichen zur Trennung zwischen Prefix und eigentlichem Namen ist der Punkt.

"Selected names" werden nach folgender Syntax, z.B. für Objekte eines Packages, für Design-Einheiten aus einer Bibliothek oder für Record-Elemente verwendet:

```
lib_name.pack_name.obj_name_in_pack
lib_name.design_unit_name
record_name.record_element_name
```

Um bestimmte Elemente / Funktionen aus Packages oder Design-Einheiten aus Bibliotheken lokal ohne vollständigen Deklarationspfad an-

sprechen zu können, kann die USE-Anweisung auch innerhalb der Deklarationsanweisungen von Architekturen, Blöcken, Prozessen, usw. stehen.

Weitere Hinweise zur Sichtbarkeit von Bibliotheken und Packages finden sich im Abschnitt zu den LIBRARY- und USE-Anweisungen.

```
PACKAGE p IS
  CONSTANT c : integer := 1;
END p;
```

```
ENTITY e IS
  CONSTANT c : integer := 2;      -- in ARCH. zu e bekannt
END e;
```

```
ARCHITECTURE a OF e IS
  SIGNAL s1,s2,s3,s4,s5,s6,s7 : integer := 0;
BEGIN
  -- PACKAGE p hier nicht bekannt: sel.name work.p.c noetig
  s6 <= work.p.c ;       -- benutzt c aus PACKAGE p :   1
  s7 <= c;               -- benutzt c aus ENTITY  e :   2
  b : BLOCK
    CONSTANT c : integer := 3;
  BEGIN
    s3 <= c;             -- benutzt c aus BLOCK   b :  3
    x : PROCESS
      CONSTANT c : integer := 4;
      USE work.p;        -- mache PACKAGE p lokal sichtbar
    BEGIN
      s4 <= c;           -- benutzt c aus PROCESS x :   4
      l : FOR c IN 5 TO 5 LOOP
        s1 <= p.c;       -- benutzt c aus PACKAGE p :   1
        s2 <= e.c;       -- benutzt c aus ENTITY  e :   2
        s5 <= c;         -- benutzt c aus LOOP    l :   5
      END LOOP;
      WAIT;
    END PROCESS;
  END BLOCK;
END a;
```

11 Spezielle Modellierungstechniken

In diesem Kapitel sollen VHDL-Konstrukte und Modellierungstechniken erläutert werden, die bei einfachen Aufgabenstellungen in der Regel nicht benötigt werden. Es richtet sich deshalb vorwiegend an den fortgeschrittenen VHDL-Anwender.

11.1 Benutzerdefinierte Attribute

Neben den vordefinierten Attributen können auch vom Benutzer Attribute vergeben werden. Diese können allerdings im Gegensatz zu den vordefinierten Attributen nur konstante Werte besitzen.

Benutzerdefinierte Attribute können nicht nur für Signale, Variablen und Konstanten vergeben werden, sondern auch für die Design-Einheiten (Entity, Architecture, Configuration, Package) und weitere VHDL-Elemente wie Prozeduren, Funktionen, Typen, Untertypen, Komponenten und sogar für Labels.

Mit Hilfe von Attributen können diesen verschiedenen Elementen zusätzliche Informationen mitgegeben werden, die sich mit den bisher eingeführten VHDL-Konstrukten nicht abbilden lassen. Beispielsweise können einer Architektur Angaben über die maximalen Gehäuseabmessungen, den Bauteillieferanten oder die erwarteten Herstellungskosten zugeordnet werden.

Ein bedeutendes Anwendungsgebiet für benutzerdefinierte Attribute ist die VHDL-Synthese. Hier lassen sich mit programmspezifischen Attributen Vorgaben zur Zustandscodierung oder Pfadlaufzeit festlegen. Während der VHDL-Simulator diese Attribute nicht weiter bearbeitet, werden sie vom Synthesewerkzeug erkannt und entsprechend umgesetzt.

Gegenüber Kommentaren bietet die Verwendung von Attributen den Vorteil der strengen Typüberwachung und die Möglichkeit, den Attributwert mit Hilfe des Attributnamens (wie bei den vordefinierten Attributen) in den VHDL-Modellen abzufragen.

Bevor das Attribut einem VHDL-Element zugewiesen und mit einem Wert versehen werden kann, muß zunächst eine Attributdeklaration erfolgen.

Deklaration von Attributen

Attributdeklarationen haben folgendes Aussehen:

```
ATTRIBUTE attribute_name : type_name ;
```

Der Typ des Attributs (`type_name`) kann ein vordefinierter oder ein eigendefinierter Typ sein.

Definition von Attributen

Die Verknüpfung des Attributs mit einem oder mehreren Elementen unter gleichzeitiger Wertzuweisung geschieht durch Angabe des entsprechenden VHDL-Elements (`element_type`: Konstante, Variable, Signal, Entity, ... Label) mit Hilfe folgender Anweisung:

```
ATTRIBUTE attribute_name OF  element_name_1
                         {,  element_name_n}
     : element_type IS attribute_value ;
```

Anstelle der Elementnamen (oder Labels von bestimmten Anweisungen) sind auch die Schlüsselwörter `OTHERS` und `ALL` möglich.

Vordefinierte Attribute können auf diese Weise nicht mit neuen Werten belegt werden; sie werden ausschließlich vom Simulator belegt und können nur abgefragt werden.

Anwendung von Attributen

Die benutzerdefinierten Attribute können in der gleichen Art und Weise wie die vordefinierten Attribute abgefragt werden:

```
element_name'attribute_name
```

```
ENTITY attr_demo IS
  -- Typdeklarationen -----------------------------------
  TYPE level IS (empty, medium, full);
  TYPE state IS (instabil, stabil);
  TYPE res_type IS RANGE 0.0 TO 1.0E6;
  TYPE loc_type IS RECORD x_pos : integer;
                  y_pos : integer;
                  END RECORD;
  -- Attributdeklarationen ------------------------------
  ATTRIBUTE technology    : string;
  ATTRIBUTE priority      : level;
  ATTRIBUTE sig_state     : state;
  ATTRIBUTE resistance    : res_type;
  ATTRIBUTE location      : loc_type;
  -- Attributdefinition ---------------------------------
  ATTRIBUTE technology OF attr_demo : ENTITY IS "ttl";
END attr_demo;
```

```
ARCHITECTURE demo OF attr_demo IS
  SIGNAL in_a, in_b, in_c : bit := '1';
  -- Attributdefinition ---------------------------------
  ATTRIBUTE sig_state  OF in_a       : SIGNAL IS instabil;
  ATTRIBUTE sig_state  OF OTHERS     : SIGNAL IS stabil;
BEGIN
  -- Attributanwendung ----------------------------------
  ASSERT attr_demo'technology = "cmos"
         REPORT "Kein CMOS-Modul" SEVERITY note;
END demo;
```

Die neue Norm (✓**93**) erlaubt, daß jede Anweisung - auch sequentielle - ein Label erhalten kann. Somit kann nahezu jede Zeile aus dem VHDL-Quelltext mit einem Attribut versehen werden.

Ebenso können in ✓**93** Gruppen mit Attributen belegt werden.

11.2 Gruppen ✓93

Mit der Überarbeitung der Norm wurde auch der Begriff der Gruppe in die Sprache eingeführt. Mehrere Objekte, Design-Einheiten und Unterprogramme können in einer Gruppe zusammengefaßt und gemeinsam mit Attributen dekoriert werden. Gruppenelemente können alle mit Namen versehene VHDL-Elemente sein (siehe auch Aufzählung bei den benutzerdefinierten Attributen; in ✓93 kommen dazu noch `LITERAL`, `UNITS`, `GROUP` und `FILE`).

Da auch Labels in eine Gruppe aufgenommen werden können, lassen sich z.B. Prozesse oder nebenläufige Signalzuweisungen, die mit einem Label versehen sind, in Gruppen zusammenfassen.

Typdeklaration von Gruppen

Bevor man jedoch konkrete Gruppen bildet, muß ähnlich wie bei herkömmlichen Objekten, in einer Deklaration der Gruppentyp festgelegt werden:

```
GROUP group_type_name IS
            (    element_1_type  [<>]
             { , element_n_type  [<>] } );
```

Die optionale Angabe der Zeichen `<>` bedeutet, daß beliebig viele Elemente des genannten Typs auftreten können. Beispiele:

```
GROUP path IS (SIGNAL, SIGNAL);  -- Pfad: 2 Signale
GROUP pins IS (SIGNAL <>);       -- Pins: beliebig
                                 -- viele Signale (<>)
```

Deklaration von Gruppen

Nachdem der Gruppentyp bekanntgegeben wurde, können in der Gruppendeklaration konkrete VHDL-Einheiten zu einer Gruppe zusammenfügt werden.

Die Syntax für die Deklaration einer Gruppe lautet:

```
GROUP   group_name : group_type_name
    ( group_element_1 { , group_element_n } );
```

Anzahl und Typen der Gruppenelemente müssen dabei der Typdeklaration entsprechen.

Anwendung von Gruppen

Ein Anwendungsgebiet für Gruppen kann z.B. die Angabe von Verzögerungszeiten einer Gruppe `path`, die aus zwei Signalen besteht, mit Hilfe von benutzerdefinierten Attributen sein:

```
ENTITY dlatch_93 IS                -- !!! VHDL'93-Syntax !!!
  PORT (d, clk  : IN  bit;
        q, qbar : OUT bit);

  GROUP path IS (SIGNAL, SIGNAL);
  GROUP d_to_q      : path (d, q);
  GROUP d_to_qbar   : path (d, qbar);
  GROUP clk_to_q    : path (clk, q);
  GROUP clk_to_qbar : path (clk, qbar);
  ATTRIBUTE propagation : time;
  ATTRIBUTE propagation OF d_to_q      : GROUP IS 3.8 ns;
  ATTRIBUTE propagation OF d_to_qbar   : GROUP IS 4.2 ns;
  ATTRIBUTE propagation OF clk_to_q    : GROUP IS 2.8 ns;
  ATTRIBUTE propagation OF clk_to_qbar : GROUP IS 2.9 ns;
END dlatch_93;
```

```
ARCHITECTURE with_path_attributes OF dlatch_93 IS
BEGIN                              -- !!! VHDL'93-Syntax !!!
  PROCESS (d, clk)
  BEGIN
    IF clk'EVENT AND clk = '1' THEN
      q    <= d AFTER clk_to_q'propagation;
      qbar <= d AFTER clk_to_qbar'propagation;
    ELSIF d'EVENT THEN
      q    <= d AFTER d_to_q'propagation;
      qbar <= d AFTER d_to_qbar'propagation;
    END IF;
  END PROCESS;
END with_path_attributes;
```

11.3 Überladung

Ein Punkt, der erneut die Verwandtschaft von VHDL mit höheren Programmiersprachen zeigt, ist die Möglichkeit zur Überladung ("overloading") von:

- Unterprogrammen (Funktionen und Prozeduren),
- Operatoren und
- Werten von Aufzähltypen.

Unter Überladung versteht man die gleichzeitige Sichtbarkeit von mehreren gleichnamigen Unterprogrammen, Operatoren bzw. von Objektwerten die zu unterschiedlichen Aufzähltypen gehören können. Mit Hilfe der Überladung gelingt es beispielsweise, den Anwendungsbereich einer Funktion zu vergrößern.

Die verschiedenen Varianten eines Unterprogramms oder eines Operators unterscheiden sich nur in Anzahl und Typ ihrer Argumente und Ergebnisse. VHDL-Programme erkennen aus dem Kontext heraus (d.h. aus der Argumentzahl und deren Typen), welche der sichtbaren Varianten anzuwenden ist. Ist diese Entscheidung nicht eindeutig, d.h. sollten sich mehrere sichtbare Alternativen als "passend" für die zu erfüllende Aufgabe erweisen, erfolgt die Ausgabe einer Fehlermeldung.

Durch das Konzept der Überladung werden VHDL-Modelle übersichtlicher, da nicht für jede Variante einer bestimmten Funktionalität ein neuer Bezeichner vergeben werden muß.

11.3.1 Überladen von Unterprogrammen

Verschiedene Unterprogramme mit gleichem Namen, aber unterschiedlichem Verhalten, werden wie gewöhnlich deklariert. Dies gilt gleichermaßen für Prozeduren und Funktionen. Ein Beispiel:

```
ENTITY overload IS
----- Bestimmung des Maximums von zwei integer-Werten ------
  FUNCTION max (a_i, b_i : integer) RETURN integer IS
  BEGIN
    IF a_i >= b_i THEN RETURN a_i;
    ELSE              RETURN b_i;
    END IF;
  END max;
----- Bestimmung des Maximums von drei integer-Werten ------
  FUNCTION max (a_i, b_i, c_i : integer) RETURN integer IS
  BEGIN
    IF    a_i >= b_i AND a_i >= c_i THEN RETURN a_i;
    ELSIF b_i >= a_i AND b_i >= c_i THEN RETURN b_i;
    ELSE                                 RETURN c_i;
    END IF;
  END max;
----- Bestimmung des Maximums von zwei real-Werten ---------
  FUNCTION max (a_r, b_r : real) RETURN real IS
  BEGIN
    IF a_r >= b_r THEN RETURN a_r;
    ELSE               RETURN b_r;
    END IF;
  END max;
----- Bestimmung des Maximums von drei real-Werten ---------
  FUNCTION max (a_r, b_r, c_r : real) RETURN real IS
  BEGIN
    IF    a_r >= b_r AND a_r >= c_r THEN RETURN a_r;
    ELSIF b_r >= a_r AND b_r >= c_r THEN RETURN b_r;
    ELSE                                 RETURN c_r;
    END IF;
  END max;
END overload;
```

Die vier Funktionen `max` seien in einer Architektur gleichzeitig sichtbar. Für jeden Funktionsaufruf von `max` wählt das VHDL-Programm die Variante aus, die in folgenden Punkten dem Aufruf entspricht:

① Zahl der Argumente,

② Typen der Argumente,

③ Namen der Argumente (bei "named association"),

④ Typ des Rückgabewertes.

Folgendes Beispiel zeigt die Anwendung der oben beschriebenen, mehrfach überladenen Funktion `max`:

```
ARCHITECTURE behavioral OF overload IS
BEGIN
  PROCESS
    VARIABLE a, d : real := 0.0;
    VARIABLE b, c : integer := 0;
  BEGIN
    a := max(3.2, 2.1);   -- 3. Variante von max, a = 3.2
    b := max(1, 2, 3);    -- 2. Variante von max, b = 3
    c := max(0, 9);       -- 1. Variante von max, c = 9
    d := max(real(6), 4.3, 2.1); -- 4. Var. von max, d = 6.0
    WAIT;
  END PROCESS;
END behavioral;
```

Besitzen bei einem Funktionsaufruf mehrere Varianten Gültigkeit, so verbirgt die lokal deklarierte Variante (z.B. im Deklarationsteil der Architektur) die hierarchisch höher deklarierte Variante (z.B. aus einem Package). Mit vollständiger hierarchischer Angabe des Unterprogrammnamens (durch "selected names") kann trotzdem auf die gewünschte Variante zugegriffen werden.

11.3.2 Überladen von Operatoren

Operatoren unterscheiden sich von einfachen Funktionen durch zwei Punkte:

- Der Name bzw. das Symbol von Operatoren gilt nicht als "normaler" Bezeichner, sondern ist eine Zeichenkette (String) und steht deshalb bei der Deklaration in Anführungsstrichen.
- Die Operanden stehen beim üblichen Aufruf eines Operators vor bzw. nach dem Operator selbst (bei unären Operatoren nur danach) und nicht in nachgestellten runden Klammern. Operatoren können aber auch in der Syntax normaler Funktionsaufrufe angesprochen werden:

  `"c <= a AND b;"` entspricht `"c <= "AND" (a,b);"`

Die Handhabung von Operatoren als String ist bei der Deklaration von überladenen Operatoren zu beachten. Ansonsten entspricht diese dem Überladen von Funktionen und Prozeduren. Folgendes Beispiel zeigt die Überladung des "="-Operators. Bei Verwendung des Packages `fourval` sind damit drei Varianten des "="-Operators verfügbar: Die Variante `[0]` stellt den vordefinierten Operator "=" dar, welcher ein Ergebnis vom Typ `boolean` zurückliefert. Die beiden benutzerdefinierten Varianten `[1]`, `[2]` dagegen vergleichen zwei Werte vom Typ `mvl4` und errechnen ein Ergebnis vom Typ `bit` bzw. `mvl4`.

```
PACKAGE fourval IS
   TYPE mvl4 IS ('X', '0', '1', 'Z');
   -- Variante [0]: vordefinierter Operator "="
   -- FUNCTION "=" (a,b : mvl4) RETURN boolean;
   -- Variante [1]: eigene Definition des Operators "="
   FUNCTION "=" (a,b : mvl4) RETURN bit;
   -- Variante [2]: eigene Definition des Operators "="
   FUNCTION "=" (a,b : mvl4) RETURN mvl4;
END fourval;
```

```
PACKAGE BODY fourval IS
  FUNCTION "=" (a, b : mvl4) RETURN bit IS          -- [1]
    VARIABLE result : bit := '0';
  BEGIN
    IF (a = '1' AND b = '1') OR (a = '0' AND b = '0') OR
       (a = 'X' AND b = 'X') OR (a = 'Z' AND b = 'Z')
       THEN result := '1';
    END IF;
    RETURN result;
  END "=";

  FUNCTION "=" (a, b : mvl4) RETURN mvl4 IS         -- [2]
    VARIABLE result : mvl4 := '0';
  BEGIN
    IF (a = '1' AND b = '1') OR (a = '0' AND b = '0') OR
       (a = 'X' AND b = 'X') OR (a = 'Z' AND b = 'Z')
       THEN result := '1';
    END IF;
    RETURN result;
  END "=";
END fourval;
```

11.3.3 Überladen von Aufzähltypwerten

Neben Unterprogrammen und Operatoren können auch die einzelnen Werte eines Aufzähltyps überladen werden. Man denke z.B. an den Signalwert `'0'` des Logiktyps `bit` und die `'0'` des benutzerdefinierten Logiktyps `mvl4`. Tritt ein Ausdruck mit solchen Werten im VHDL-Quelltext auf, so muß aus dem Kontext heraus klar sein, um welchen Typ es sich dabei handelt. Kann dies nicht eindeutig festgestellt werden, so ist eine explizite Typkennzeichnung erforderlich.

11.3.4 Explizite Typkennzeichnung

Bei mehrdeutigen Ausdrücken (s.o.) und Typen von Operanden ist die explizite Kennzeichnung des gewünschten Typs durch sog. qualifizierte Ausdrücke ("qualified expressions") erforderlich. Unter Angabe des Typ- bzw. Untertypnamens haben sie folgende Syntax:

`type_name'(`*`expression`*`)`

Eine mögliche Konstellation für die Notwendigkeit eines solchen Konstruktes zeigt folgendes Beispiel. Hier wird durch die Typkennzeichnung explizit angegeben, welche Variante des mehrfach überladenen Vergleichsoperators "=" zu verwenden ist.

```
USE work.fourval.ALL;  -- siehe oben
ENTITY equal IS
  PORT (a,b,c,d : IN  mvl4;
        w :       OUT mvl4; x,y,z :   OUT boolean );
END equal;

ARCHITECTURE behavioral OF equal IS
BEGIN                                -- verw. Funktion in [ ]
  w <= (a = b) = (c = d);            -- (a [2] b) [2] (c [2] d)
  x <= (a = b) = mvl4'(c = d);       -- (a [2] b) [0] (c [2] d)
  y <= (a = b) = bit'(c = d);        -- (a [1] b) [0] (c [1] d)
  z <= (a = b) = boolean'(c = d); -- (a [0] b) [0] (c [0] d)
END behavioral;
```

11.4 PORT MAP bei strukturalen Modellen

Mit Hilfe von PORT MAP-Anweisungen können Signale auf unterschiedlichen Ebenen auf verschiedene Arten miteinander verbunden werden.

PORT MAP-Anweisungen können an verschiedenen Stellen auftreten:

- In der Design-Einheit "Configuration" verbinden sie "formals" (Ports der instantiierten Entity) mit "locals" (Ports der Komponente).
- In der Anweisung zur Komponenteninstantiierung verbinden sie "locals" (Ports der Komponente) mit "actuals" (Signale in der Architektur).

Für den erstgenannten Fall sollen einige Aspekte der PORT MAP aufgezeigt werden, die auch für Komponenteninstantiierungen gelten:

- Im einfachsten Fall werden durch eine mit Kommata getrennte Liste von "local"-Portnamen beide Signalebenen (in der gleichen Reihenfolge wie in der Komponentendeklaration angegeben) miteinander verbunden (sog. "positional association").
- Explizites Zuweisen von Signalen mit dem Zuweisungszeichen "=>" eröffnet weitaus mehr Möglichkeiten (sog. "named association"):
  - Signale können in unterschiedlicher Reihenfolge miteinander verknüpft werden,
  - "formal"-Ports können unbeschaltet bleiben (Schlüsselwort OPEN). In diesem Fall muß ein "formal"-Port mit dem Modus IN einen Defaultwert besitzen,
  - ein "local"-Port kann mit mehr als einem "formal"-Port verknüpft werden.

In Abb. B-21 sind die erlaubten Konstellationen dargestellt.

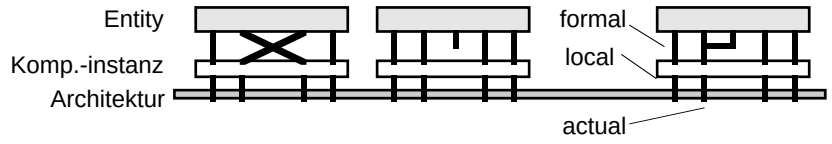

Abb. B-21: Erlaubte PORT MAP-Konstrukte in der Konfiguration

Nicht erlaubt hingegen sind unbeschaltete "locals" und "formals", die mit mehr als einem "local" verknüpft werden.

Folgende Alternative des Komplexgatters aus Kapitel 5 zeigt die Anwendung des Schlüsselwortes OPEN:

```
CONFIGURATION aoi_config_4a OF aoi IS
  FOR structural_4
    FOR nor_stage
      FOR or_c  : or2  USE ENTITY work.or2 (behavioral);
      END FOR;
    END FOR;
    FOR and_stage
-- hier:Verwendung eines AND3-Gatters in einem AND2-Sockel
-- durch Angabe von OPEN als ACTUAL des dritten Eingangs-
-- ports, work.and3 muss dort einen Defaultwert besitzen!
      FOR and_b : and2 USE ENTITY work.and3 (behavioral)
        PORT MAP (a => OPEN, b => a, c => b, y => y);
      END FOR;
      FOR and_a : and3 USE ENTITY work.and3 (behavioral);
      END FOR;
    END FOR;
  END FOR;
END aoi_config_4a;
```

11.5 File - I/O

Files (Dateien) sind serielle Anordnungen von Daten eines bestimmten Typs, an deren Ende ein "end of file" (EOF)-Zeichen steht. Files werden nicht wie herkömmliche Objekte gelesen und zugewiesen, sondern können nur mit Hilfe spezieller Funktionen gehandhabt werden. Man unterscheidet bei VHDL zwischen Textfiles und typspezifischen Files.

Bei ersteren handelt es sich um beliebig strukturierte ASCII-Dateien, die mit Hilfe der Prozeduren aus dem Package `textio` gelesen und geschrieben werden können. Die typspezifischen Files dagegen können jeweils nur Daten eines Typs enthalten. Sie werden vom jeweiligen VHDL-Simulator mit den implizit vorhandenen Prozeduren `read` und `write` gelesen bzw. geschrieben. Dieser Dateityp ist nicht menschenlesbar und kann nicht mit ASCII-Editoren erzeugt werden.

Typdeklaration von Files

Wie bei allen VHDL-Objekten ist zunächt eine Typdeklaration erforderlich, die den Namen des Filetyps mit dem Typ der Elemente verbindet:

```
TYPE file_type_name IS FILE OF element_type ;
```

Neben den Basistypen (`bit`, `integer`, ...) können auch benutzerdefinierte Typen, Arrays und Records in einem File enthalten sein.

Deklaration von Files

Daraufhin kann nun ein konkretes Fileobjekt von diesem Typ deklariert werden. Dazu muß der Modus (`IN`: Leseoperation möglich, `OUT`: Schreiboperation möglich) und der Filename angegeben werden:

```
FILE file_name : file_type_name IS IN
          "physical_file_name" ;

FILE file_name : file_type_name IS OUT
          "physical_file_name" ;
```

Durch den String `"physical_file_name"` wird der Dateiname im Filesystem angegeben, während `file_name` den Bezeichner des Objektes darstellt. Die Anwendung von Files in VHDL ist allerdings nicht sehr flexibel, da Files nur gelesen oder geschrieben werden können (es ist kein `INOUT`-Modus möglich) und auch die Reihenfolge streng sequentiell ist.

11.5.1 Typspezifische Files

In diesem Fall werden Daten ein und desselben Typs in einem File abgespeichert oder gelesen, so z.B. `integer`, `bit`, `bit_vector`. Es stehen dafür folgende Leseprozeduren zur Verfügung:

```
read (file_name, object_name) ;

read (file_name, object_name, length) ;
```

Beide Prozeduren lesen Daten aus dem File `file_name` in die Variable `object_name`. Handelt es sich beim einzulesenden Objekt um einen unbeschränkten Vektortyp (z.B.: `bit_vector`), so stellt die zweite Variante die Länge (`length`) des tatsächlich gelesenen Vektors zur Verfügung.

Das Gegenstück zu `read` ist die Prozedur `write`, die den Inhalt von `object_name` auf das File `file_name` schreibt:

```
write (file_name, object_name) ;
```

Die folgende Funktion liefert eine Boolesche Variable, die entweder den Wert `true` für erreichtes Dateiende annimmt oder `false` ist, falls noch weitere Daten vorhanden sind.

```
endfile (file_name)
```

Folgendes Beispiel zeigt ein VHDL-Modell, das den Verlauf des Eingangssignals `data_8` bis zum Zeitpunkt 100 ns auf die Datei `data_8.out` schreibt. Diese Datei könnte anschließend in einem anderen Modell mit der Prozedur `read` wieder eingelesen werden.

```
PACKAGE memory_types IS
  SUBTYPE byte IS bit_vector(0 TO 7);
END memory_types;
```

```
USE work.memory_types.ALL;
ENTITY write_data IS
  PORT (data_8 : IN byte);
END write_data;
```

```
ARCHITECTURE behavioral OF write_data IS
BEGIN
  write_process: PROCESS
    TYPE byte_file IS FILE OF byte;
    FILE output : byte_file IS OUT "data_8.out";
  BEGIN
    WAIT on data_8;
    IF now <= 100 ns THEN
      write (output, data_8);
    ELSE
      WAIT;
    END IF;
  END PROCESS;
END behavioral;
```

11.5.2 Textfiles

Eine Möglichkeit, innerhalb von Files auch unterschiedliche Typen abzuspeichern, bietet das Package `textio` aus der Bibliothek `std`. Es muß vor der Verwendung seiner Routinen erst mit folgender Anweisung bekanntgemacht werden:

```
USE std.textio.ALL ;
```

Das Package deklariert den Typ `line` und einen entsprechenden Filetyp `text`. Files dieses Typs können mit folgenden Anweisungen zeilenweise gelesen und geschrieben werden:

```
readline  (file_name, line_object_name) ;
writeline (file_name, line_object_name) ;
```

Mit Hilfe der überladenen Prozeduren aus dem Package `textio`:

```
read (line_object_name, object_name) ;
write (line_object_name, object_name) ;
```

können verschiedene Objekttypen innerhalb der Zeilen gelesen und geschrieben werden.

Die möglichen Typen sind:

- `bit`
- `bit_vector`
- `boolean`
- `time`
- `integer`
- `real`
- `character`
- `string`

Für die Leseprozeduren existieren auch Varianten mit Booleschem Prüfwert zur Kontrolle auf erfolgreiche Datei-Operation. Für die Schreibprozeduren existieren Varianten zum Ausrichten der Objekte innerhalb eines gegebenen Bereiches. Weiterhin gibt es die Funktion:

```
endline (line_object_name)
```

zur Überprüfung auf erreichtes Zeilenende (vgl. `endfile`).

Das prinzipielle Vorgehen beim Arbeiten mit Text-Files soll an einem Beispiel verdeutlicht werden. Es stellt eine flexible Testbench für ein beliebiges Modell mit drei Ein- und einem Ausgangsport dar. Die Zeilen im File `"stimres"` sind folgendermaßen aufgebaut:

Signalname (Typ `character`) - Leerzeichen - Signalwert (Typ `bit`) - Leerzeichen - Zeitangabe der Zuweisung (Typ `time`):

```
a 0 1 ns
b 0 2 ns
r 1 4 ns
  ...
  ...
```

Neben den Stimuli (Signale `a` , `b`, `c`) wird gleichzeitig die erwartete Antwort (Signal `r`) definiert und zu den entsprechenden Zeitpunkten mit dem Ausgangssignal `y` verglichen. Bei nicht übereinstimmenden Werten wird eine Fehlermeldung in das File `"errors"` geschrieben:

```
USE std.textio.ALL;

ENTITY tb_3pin IS
END tb_3pin;
```

```
ARCHITECTURE arch_1 OF tb_3pin IS
  SIGNAL a,b,c,r,y: bit;
  COMPONENT mut_socket
    PORT (a,b,c: IN  bit; y: OUT bit);
  END COMPONENT;
  FILE stimres : text IS IN  "stimres";
  FILE errors  : text IS OUT "errors";
BEGIN
  model_under_test: mut_socket PORT MAP (a,b,c,y);
  read_stimuli: PROCESS --------- Lese Stimuli ----------
    VARIABLE lin          : line; VARIABLE boo: boolean;
    VARIABLE t            : time;
    VARIABLE str_var,space: character; VARIABLE data: bit;
  BEGIN
    WHILE (endfile(stimres) = false) LOOP
      readline (stimres, lin);
      read (lin,str_var,boo);
      ASSERT boo REPORT "Error reading file!";
      IF ((str_var = 'a') OR (str_var = 'b')
       OR (str_var = 'c') OR (str_var = 'r')) THEN
        read (lin,space); read (lin,data);
        read (lin,space); read (lin,t);
        CASE str_var IS
          WHEN 'a'    => a <= TRANSPORT data AFTER t;
          WHEN 'b'    => b <= TRANSPORT data AFTER t;
          WHEN 'c'    => c <= TRANSPORT data AFTER t;
          WHEN 'r'    => r <= TRANSPORT data AFTER t;
          WHEN OTHERS => NULL;
        END CASE;
      END IF;
    END LOOP;
    WAIT;
  END PROCESS;
  write_errors : PROCESS ----- Schreibe Fehlermeldung ------
    VARIABLE lin : line;
  BEGIN
    WAIT ON r'TRANSACTION;
    IF (y /= r) THEN
      write (lin, string'("y isn't "));  write (lin, r);
      write (lin, string'(" at time ")); write (lin, now);
      writeline (errors,lin);
    END IF;
  END PROCESS;
END arch_1;
```


11.5.3 Handhabung von Files in ✓93

Die ursprüngliche Definition der Syntax zur Handhabung von Files (✓87) enthält einige unklare Punkte und schränkt den Datenim- und -export von und zu Files erheblich ein.

Die wesentlichen Neuerungen in ✓93 bezüglich Files sind:

- Neben den Signalen, Variablen und Konstanten erhalten auch die Files explizit den Status einer eigenen Objektklasse.
- Files können als Argumente an Unterprogramme übergeben werden.
- Files können durch Angabe eines Prozeduraufrufes explizit geöffnet oder geschlossen werden. Damit wird auch eine Abfrage auf Existenz der Datei möglich.
- Files können neben "read" und "write" den Modus "append" annehmen.

Mit diesen Neuerungen ist es, zusammen den "impure functions", nun möglich, mit Hilfe verschiedener Funktionen aus einem File zu lesen.

11.6 Zeiger

Wie in vielen Programmiersprachen kann auch in VHDL-Modellen mit Zeigern gearbeitet werden. Dadurch lassen sich sehr abstrakte, implementierungsunabhängige Modelle für elektronische Systeme erstellen. Beispiele für die Anwendung von Zeigern sind dynamische Warteschlangen oder Kellerspeicher.

In VHDL sind Zeiger spezielle Variablen, die die Adresse für ein Objekt speichern, das selbst nicht direkt ansprechbar ist, also keinen eigenen Bezeichner (identifier) besitzt. Da Zeiger immer Variablen sind, können sie nur innerhalb von Prozessen oder Unterprogrammen eingesetzt werden.

Typdeklaration von Zeigern

Bevor eine Zeigervariable deklariert werden kann, ist die Deklaration des Zeigertyps (sog. "access-type") unter Angabe des Typs, auf den gezeigt werden soll (`type_name`), erforderlich:

```
TYPE pointer_type IS ACCESS type_name ;
```

Deklaration von Zeigern

Die Zeigervariable kann dann in drei Varianten deklariert werden:

```
VARIABLE pointer_name : pointer_type
        := NEW type_name ;

VARIABLE pointer_name : pointer_type
        := NEW type_name'(def_value) ;

VARIABLE pointer_name : pointer_type
        := NULL ;
```

Bei der letzten Variante wird lediglich ein Zeiger angelegt, der auf kein Objekt zeigt. Die beiden anderen Varianten hingegen reservieren durch das Schlüsselwort `NEW` für ein Objekt vom Typ `type_name` den notwendigen Speicherplatz, legen dieses Objekt an und weisen ihm einen Defaultwert zu. Wird dieser Wert nicht wie in der zweiten Variante explizit angegeben, so entspricht der Defaultwert dem am weitesten links stehenden Wert in der Deklaration von `type_name`.

Um den Speicherplatz eines Objektes wieder freizugeben, existiert die Prozedur `deallocate`, deren einziges Argument der entsprechende Zeigername ist:

```
deallocate (pointer_name) ;
```

Anwendung von Zeigern

Folgendes Beispiel soll die Anwendung von Zeigern verdeutlichen:

```
ENTITY acc_types IS
END acc_types;
```



```
ARCHITECTURE behavioral OF acc_types IS
BEGIN
  PROCESS
    TYPE pos2 IS ARRAY (1 DOWNTO 0) OF positive;   -- (1)
    TYPE access_pos2 IS ACCESS pos2;               -- (2)
    VARIABLE p1 : access_pos2 := NEW pos2'(2,3);   -- (3)
    VARIABLE p2 : access_pos2 := NEW pos2;         -- (4)
    VARIABLE p3,p4 : access_pos2 := NULL;          -- (5)
  BEGIN
    p2.ALL (0) := 8;                               -- (6)
    p2 (1) := 7;                                   -- (7)
    p3 := p2;                                      -- (8)
    p4 := NEW pos2;                                -- (9)
    p4.ALL := p2.ALL;                              -- (10)
    deallocate (p1);                               -- (11)
    WAIT;
  END PROCESS;
END behavioral;
```

In Zeile (2) wird ein Zeigertyp für den in Zeile (1) deklarierten Vektortyp deklariert. Die Zeilen (3) bis (5) deklarieren die vier Zeigervariablen p1 bis p4. Zusätzlich wird in Zeile (3) und (4) jeweils ein nicht benanntes Objekt von Typ pos2 angelegt und initialisiert. Nach Abarbeitung dieser Zeilen, d.h. nach Ausführung der Prozeßinitialisierung, ergibt sich die in Abb. B-22 links dargestellte Situation.

In den Zeilen (6) und (7) werden den Elementen des Objektes, auf das der Zeiger p2 zeigt, Werte zugewiesen. Das Schlüsselwort ALL bewirkt ein sog. "Dereferenzieren" des Zeigers, d.h. daß nicht der Zeiger, sondern das Objekt, auf das gezeigt wird, angesprochen ist. Das Schlüsselwort ALL kann entfallen, wenn eine Bereichseinschränkung (wie in Zeile (7)) verwendet wird.

In Zeile (8) wird dem Zeiger p3 die Adresse des Objektes übergeben, auf das der Zeiger p2 zeigt. Zeile (9) bewirkt durch das Schlüsselwort NEW eine Speicherplatzreservierung für ein Objekt. Die Adresse dieses Objekts wird im Zeiger p4 gehalten. Sein initialer Wert wird durch die Zuweisung in Zeile (10) überschrieben.

Mit der Prozedur deallocate() wird in Zeile (11) der Speicherbereich des Objekts freigegeben, auf das p1 gezeigt hat.

Nach Abarbeitung der Zeilen `(6)` bis `(11)` stellt sich die in Abb. B-22 rechts dargestellte Situation ein.

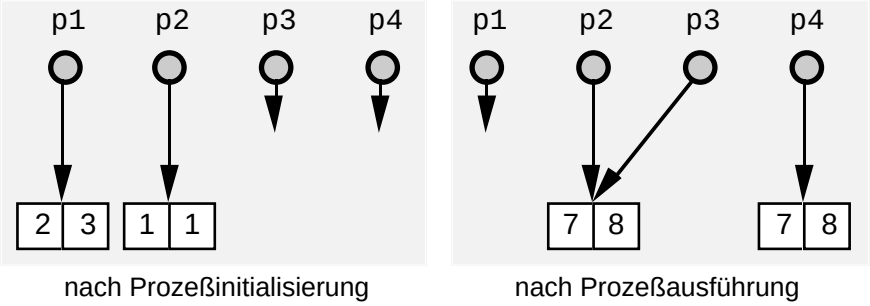

Abb. B-22: Verwendung von Zeigern

Unvollständige Typdeklaration

Um mit VHDL auch rekursive Strukturen modellieren zu können, sind neben den Zeigern auch die sog. "**incomplete types**" notwendig. Wie der Name besagt, stellen diese eine unvollständige Typdeklaration dar:

```
TYPE type_name ;
```

Die zugehörige vollständige Typdeklaration muß innerhalb desselben Deklarationsbereiches nachfolgen.

Anwendung von unvollständigen Typdeklarationen

Die Anwendung einer unvollständigen Typdeklaration zur Modellierung einer verketteten Liste wird am Beispiel einer Warteschlange gezeigt. Sie besteht aus beliebig vielen Elementen des Typs `chain_element`, die aus dem eigentlichen Datenelement vom Typ `integer` und einem Zeiger bestehen. Die Verkettung wird dadurch realisiert, daß der Zeiger jedes Elementes auf das nächste Element zeigt.

Zur Handhabung der Funktionalität sind zusätzlich zwei Zeiger (`first` und `last`) erforderlich, die auf das erste bzw. letzte Element der Warteschlange zeigen (Abb. B-23).

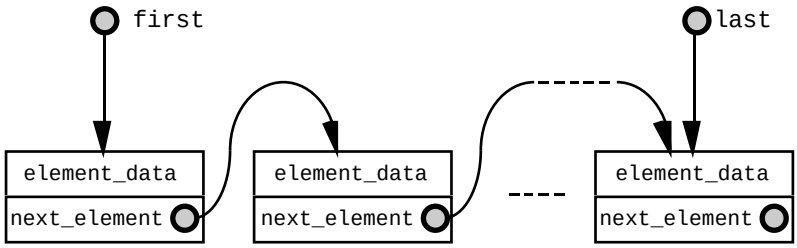

Abb. B-23: Aufbau der Warteschlange

Das VHDL-Modell zur Realisierung einer Warteschlange mit diesem Aufbau hat zwei Schnittstellen: Am Port `command` wird der Befehl für eine Operation auf der Warteschlange übergeben. Die Werte der Datenelemente werden über den Port `value` ausgegeben bzw. eingelesen.

```
ENTITY queue IS
  PORT (command : IN    queue_access;
        value   : INOUT integer);
END queue;
```

Die Alternativen für das Kommando sind im Typ `queue_access` deklariert:

- `nul`: keine Aktion,
- `add_element`: füge Element hinten (in Abb. B-23 rechts) an die Warteschlange an, wobei der aktuell anliegende Wert des Ports `value` abgespeichert wird,
- `delete_element`: lösche das erste Element (ganz links) der Warteschlange nachdem dessen Wert am Port `value` ausgegeben wurde.
- `read_element`: lese das erste Element, ohne es zu löschen. Auch hier wird dessen Wert über den Port `value` bereitgestellt.

```
ARCHITECTURE demo OF queue IS
  TYPE chain_element;              -- unvollstaendige Dekl.
  TYPE pointer IS ACCESS chain_element;
  TYPE chain_element IS RECORD     -- vollst. Deklaration
    next_element : pointer;
    element_data : integer;
  END RECORD;
BEGIN
  handle_queue : PROCESS (command)
    VARIABLE empty_flag : boolean := true;
    VARIABLE first, last, help : pointer := NULL;
  BEGIN
    CASE command IS
      WHEN add_element =>
        IF empty_flag THEN
          first := NEW chain_element; last  := first;
          empty_flag := false;
        ELSE
          last.next_element := NEW chain_element;
          last := last.next_element;
        END IF;
        last.element_data := value;
      WHEN delete_element =>
        IF empty_flag THEN
          ASSERT false REPORT "Empty queue!" SEVERITY note;
        ELSE
          value <= first.element_data, 0 AFTER 10 ns;
          help := first;
          IF first = last THEN empty_flag := true;
          ELSE first := first.next_element; END IF;
          deallocate (help);
        END IF;
      WHEN read_element =>
        IF empty_flag THEN
          ASSERT false REPORT "Empty queue!" SEVERITY note;
        ELSE
          value <= first.element_data, 0 AFTER 10 ns;
        END IF;
      WHEN nul => NULL;
    END CASE;
  END PROCESS;
END demo;
```


Das Beispiel zeigt, wie der Typ `chain_element` mit Hilfe der unvollständigen Typdeklaration im Typ `pointer` bereits verwendet werden kann, bevor er vollständig deklariert wird. Gleichzeitig wird deutlich, wie ein Record und dessen Elemente durch seinen Zeiger mit Hilfe von "selected names" referenziert werden kann.

Funktionsweise des Modells:

Bei der Prozeßinitialisierung werden keine Speicherplätze für Elemente der Warteschlange allokiert. Dies geschieht erstmals durch das Kommando `add_element`. In diesem Fall wird der Zeiger `first` für das hinzuzufügende Element verwendet. Sind bereits Elemente vorhanden, wird das neue Element durch `last.next_element` referenziert. Der Zeiger `last` wird im Anschluß daran auf das jeweils letzte Element gesetzt.

Das Löschen eines Elementes (`delete_element`) erfordert den temporären Zeiger `help`, der das später zu entfernende Element referenziert. Hier muß geprüft werden, ob es sich um das letzte Warteschlangenelement handelt. Der Wert des zu löschenden Elementes wird an den Port `value` angelegt und der Zeiger `first` wird bei noch vorhandenen weiteren Elementen auf das nächste Element gesetzt.

Das Kommando `read_element` gibt lediglich den Datenwert des ersten Elementes an den Ausgang weiter, ohne daß ein Element gelöscht wird.

11.7 Externe Unterprogramme und Architekturen

VHDL bietet sehr viele Funktionen zur Handhabung von vordefinierten Typen. Für eigene Typen können Routinen selbst verfaßt werden. Trotzdem kann es zur Handhabung komplexer Operationen oder großer Datenmengen sinnvoll sein, die Fähigkeiten von höheren Programmiersprachen zu nutzen. Prinzipiell ist es daher möglich, durch geeignete Schnittstellen externe Unterprogramme mit VHDL-Modellen zu verbinden. Eine weitere Anwendung von Schnittstellen wäre die Anbindung von fremden Architekturen oder gar von kompletten Modellen anderer Simulatoren.

Das LRM von ✓87 sieht zwar die Möglichkeit von Schnittstellen zu externen Funktionen über Packages vor, gibt aber über nähere Details keine Auskunft, d.h. es erlaubt die Verwendung, gibt aber keine Vorgaben, wie die Schnittstelle aussehen muß, wie die externen Funktionen anzusprechen sind, usw.

Die Situation hat sich heute dahingehend entwickelt, daß manche Softwarefirmen simulatorspezifische, und damit uneinheitliche Schnittstellen zu externen Funktionen (üblicherweise in "C") vorsehen, deren Handhabung aber alles andere als intuitiv ist.

Mit ✓93 wird durch die Definition eines Attributes zumindest der Versuch einer einheitlichen Handhabung unternommen. Die Definition der Schnittstelle (Signal-/Datentypen und -modi) bleibt aber weiterhin dem Softwarehersteller überlassen, ist also nach wie vor implementierungsabhängig.

Im Package `standard` von ✓93 ist dazu das Attribut `FOREIGN` deklariert, das als einziges vordefiniertes Attribut vom Benutzer zugewiesen werden kann:

```
ATTRIBUTE FOREIGN : string;
```

Externe Unterprogramme und Architekturen erhalten nun einheitlich einen VHDL-Rahmen und werden mit einem Attribut `FOREIGN` dekoriert. Dieses Attribut hat implementierungsabhängige Bedeutung und stellt die Verbindung zwischen dem VHDL-Bezeichner und dem Namen der externen Architektur bzw. des Unterprogramms her. Diese können dann wie herkömmliche Architekturen instantiiert oder wie normale Unterprogramme verwendet werden. Ein Beispiel:

```
ARCHITECTURE ext OF anything IS
  ATTRIBUTE FOREIGN OF ext : ARCHITECTURE IS "lib_z/aoi223";
BEGIN
END ext;
```

Teil C Anwendung von VHDL

1 Simulation

1.1 Überblick

Die Simulation dient im allgemeinen der Verifikation von Entwurfsschritten. Bei einer Designmethodik mit VHDL unter Verwendung von Synthesewerkzeugen werden vorwiegend Verhaltensmodelle auf abstrakten Entwurfsebenen (System-, Algorithmische und Register-Transfer-Ebene) und die entsprechenden strukturalen Modelle auf Logikebene eingesetzt. Die Simulation von VHDL-Modellen hat dabei konkret folgende Aufgaben zu erfüllen:

1.1.1 Simulation von Verhaltensmodellen

In der Regel werden Verhaltensmodelle von Hand erstellt oder durch ein Front-End-Tool generiert. Verhaltensmodelle dienen

- zur frühzeitigen Verifikation des Entwurfs,
- als Eingabe für ein Synthesewerkzeug.

Meist wird für das Verhaltensmodell bereits auf abstrakter Ebene eine Testumgebung ("Testbench") des Modells erstellt, welche die Eingangssignale (Stimuli) für das Modell zur Verfügung stellt und dessen Ausgangssignale (Ist-Antworten) mit den erwarteten Werten (Soll-Antworten) vergleicht. Durch die Angabe von erwarteten Antworten kann ein aufwendiges und fehlerträchtiges, manuelles Überprüfen der Ausgangssignale entfallen.

Eine Simulation von Verhaltensmodellen auf abstraktem Niveau muß folgende Fragen beantworten:

Ist das Modell syntaktisch korrekt?

Manuell erstellte VHDL-Modelle sind in der Regel nicht von vorne herein syntaktisch korrekt. Eine entsprechende Überprüfung kann vom Compiler-Modul des VHDL-Simulators oder von speziellen Syn-

tax-Checkern durchgeführt werden. Wurde das Verhaltensmodell durch ein Front-End-Tool generiert, kann man von der syntaktischen Korrektheit des Modells ausgehen.

Stimmt das Modell mit der Spezifikation überein?

Die Überprüfung der funktionalen Korrektheit des Entwurfsschrittes von der Spezifikation zum abstrakten Verhaltensmodell ist der eigentliche Sinn einer Simulation. Dabei stellt sich das Problem, daß durch die Simulation zwar die Anwesenheit eines Fehlers gezeigt, für größere Schaltungen aber nie die Abwesenheit von Fehlern bewiesen werden kann. Es existieren zur Verifikation des funktionalen Verhaltens zwar andere Verfahren, die dies leisten (formale Verifikation), ausgereifte Werkzeuge zur Handhabung komplexer Schaltungen stehen aber dafür nicht zur Verfügung.

Welche Eigenschaften besitzt das modellierte System?

Neben einer Überprüfung der Funktionalität kann durch die Simulation des Verhaltensmodells beispielsweise die Auslastung von Bussen oder eine geeignete Synchronisation von Submodulen bestimmt werden.

1.1.2 Simulation von strukturalen Modellen

Die VHDL-Gatternetzlisten werden kaum manuell erstellt. Sie werden in der Regel, unter Verwendung von technologiespezifischen Logikmodellen, mit Hilfe von Synthesewerkzeugen generiert. Die Simulation solcher Modelle dient zur Untersuchung der Frage:

Stimmt die Gatternetzliste mit dem Verhaltensmodell funktional überein und erfüllt sie die zeitlichen Anforderungen?

Bei einer Simulation der Gatternetzliste werden nicht nur funktionale Aspekte, sondern auch die zeitlichen Randbedingungen untersucht. Dazu kann die Testbench des Verhaltensmodells, ggf. nach Hinzufügen von weiteren zeitlichen Informationen oder genauerer Überprüfung der Ausgänge, verwendet werden. Sollen auch die Einflüsse des Layouts auf das zeitliche Verhalten der Gatternetzliste untersucht werden, so muß ein Backannotation-Schritt erfolgen, d.h. Informationen aus dem Layout - dabei handelt es sich typischerweise um die Ein-

flüsse der Leitungskapazitäten - werden in das Logikmodell zurückgeführt.

1.2 Simulationstechniken

Eine Kenntnis der unterschiedlichen Simulationstechniken ist u.a. für die Auswahl eines geeigneten Simulators bei gegebenen Schaltungsgrößen wichtig, denn sie beeinflussen die Performance des Simulators erheblich. Prinzipiell unterscheidet man, ähnlich wie bei Interpretern und Compilern für Programmiersprachen, zwei Konzepte für VHDL-Simulatoren, das interpretierende und das compilierende.

1.2.1 Interpretierende Simulationstechnik

Sie kann als die klassische Methode angesehen werden. Der VHDL-Quellcode wird bei der Simulationsvorbereitung in einen Pseudo-Code umgewandelt, der mit einem, meist auf der Programmiersprache "C" basierenden Simulator abgearbeitet werden kann. Kennzeichen der interpretierenden Simulationstechnik sind kurze Zeiten bei der Simulationsvorbereitung und längere Zeiten bei der Simulation selbst.

1.2.2 Compilierende Simulationstechnik

Hierbei wird der VHDL-Quellcode bei der Simulationsvorbereitung zunächst komplett in "C" übersetzt und mit einem C-Compiler in Objektcode umgewandelt. Bei der eigentlichen Simulation kann also direkt ein Maschinenprogramm ausgeführt werden. Längere Zeiten bei der Simulationsvorbereitung stehen einem schnelleren Simulationsablauf gegenüber.

Normalerweise ist die interpretierende Technik eher für kleinere Modelle geeignet, bei denen die Simulationszeit nicht sehr ins Gewicht fällt. Bei großen Modellen und langen Simulationszeiten macht sich jedoch aufgrund einer schnelleren Simulation der Vorteil einer compilierenden Technik bemerkbar.

1.2.3 Native-Compiled Simulationstechnik

Neuartige Simulatoren gehen einen Zwischenweg und versuchen, die Vorteile beider Ansätze zu verknüpfen. Bei der Methode der Native-Compiled-Simulation entfällt die Übersetzung in "C"; statt dessen wird aus dem VHDL-Quellcode direkt der Maschinencode erzeugt. Dadurch wird einerseits die Simulationsvorbereitungszeit im Bereich eines interpretierenden Simulators liegen, während andererseits ein ausgesprochen optimierter Objektcode vorliegt. Die eigentliche Simulation sollte noch deutlich schneller als bei compilierenden Simulatoren ablaufen.

Abb. C-1 stellt die drei Simulationstechniken gegenüber.

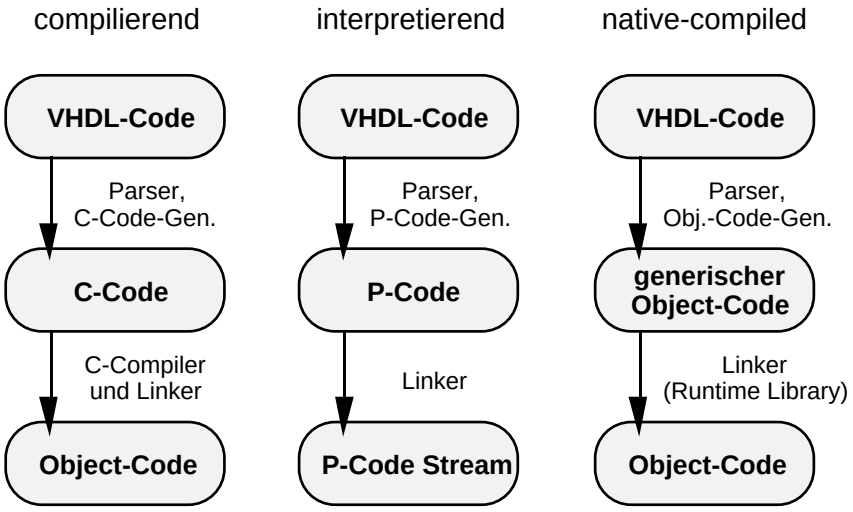

Abb. C-1: Simulationstechniken

1.3 Simulationsphasen

Gemäß dem LRM[1] der Sprache VHDL erfolgt die Simulation einer VHDL-Beschreibung in den drei Schritten "Elaboration", "Initialization" und "Execution".

In der **"Elaboration"**-Phase wird das Netzwerk der Schaltung mit allen Hierarchieebenen, Blöcken und Signalen aus den compilierten VHDL-Modellen aufgebaut. Die Elaboration-Phase ist vergleichbar mit dem "Link"-Vorgang bei der Software-Entwicklung.

In der **"Initialization"**-Phase werden alle Signale, Variablen und Konstanten mit Anfangswerten versehen. Anfangswert ist entweder der Wert, der in der VHDL-Beschreibung durch explizite Angabe bei der Deklaration des Objektes vorgegeben wurde oder der Wert, der durch das `LEFT`-Attribut des entsprechenden Typs spezifiziert ist. Außerdem wird in dieser Phase jeder Prozeß einmal gestartet und bis zur ersten `WAIT`-Anweisung bzw. bis zum Ende ausgeführt.

In der **"Execution"**-Phase wird die eigentliche Simulation durchgeführt. Zu jedem Zeitpunkt der Simulation erfolgen bis zum Eintreten eines stabilen Zustandes ein oder mehrere Delta-Zyklen. Anschließend wird die Simulationszeit bis zum nächsten Eintrag in der Ereignisliste erhöht. Die Simulation ist beendet, wenn eine spezifizierte Simulationsdauer erreicht ist oder wenn keine weiteren Signaländerungen mehr auftreten.

1.4 Testumgebungen

Zu einer kompletten Beschreibung eines elektronischen Systems in VHDL gehört auch eine Testumgebung (im Englischen "Testbench"). Darunter versteht man die Bereitstellung von Eingangssignalen (Stimuli) und die Überprüfung der Ausgangssignale (Ist-Antworten) mit den erwarteten Werten (Soll-Antworten). Testumgebungen können

---

1 LRM = Language Reference Manual

auch dazu verwendet werden, verschiedene Architekturen einer Entity miteinander zu vergleichen: ein Verhaltensmodell auf RT-Ebene kann z.B. mit dessen Syntheseergebnis (Gatternetzliste) verglichen werden.

Für die Bereitstellung der Stimuli und der Soll-Antworten sowie für die Instantiierung des zu testenden Modells ("model under test", MUT) sind verschiedene Strategien denkbar.

Die kompakteste Möglichkeit besteht darin, in der Testbench selbst Stimuli zu beschreiben und die Antworten des Modells zu überprüfen. Dies kann z.B. in getrennten Prozessen erfolgen. In dieser Testbench wird gleichzeitig auch das MUT instantiiert und mit den Stimuli bzw. Antwortsignalen verdrahtet (siehe Abb. C-2).

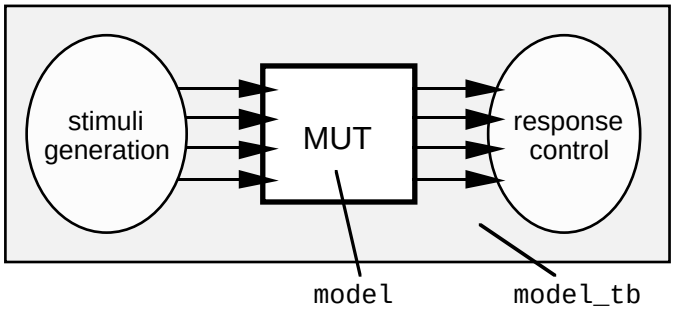

Abb. C-2: Testbenchstrategie mit einem VHDL-Modell

Daneben können Stimulibeschreibung und Antwortkontrolle auch in einem oder zwei unabhängigen VHDL-Modellen erfolgen (siehe Abb. C-3). Die Testbench dient in diesem Fall nur der Zusammenschaltung der zwei bzw. drei Modelle. Sie ist also rein struktural.

Eine Testbenchstrategie, die auf mehreren Modellen basiert, ist aufwendiger zu erstellen, als eine aus einem einzigen Modell bestehende Testbench. Allerdings bietet eine feinere Strukturierung den Vorteil, daß sich die einzelnen Modelle leichter in anderen Entwürfen wiederverwenden lassen und die Stimuli-Datensätze einfacher ausgewechselt werden können.

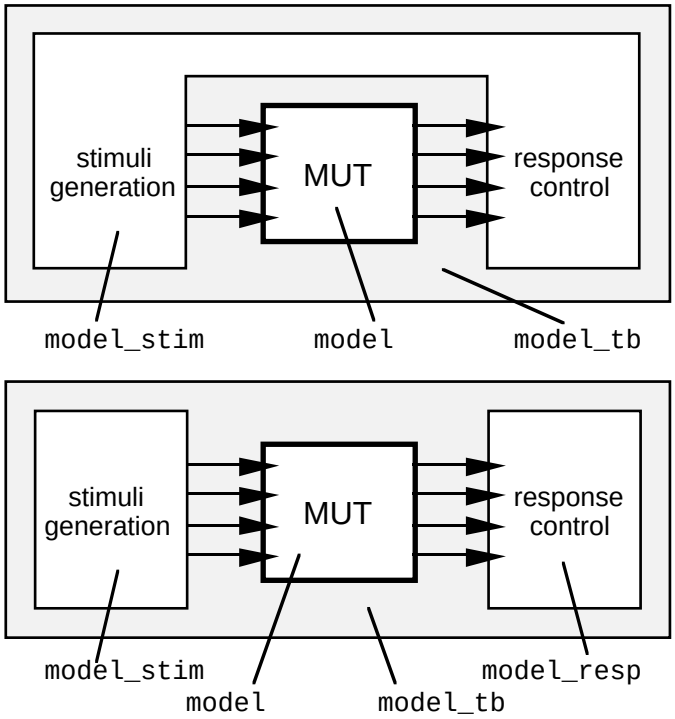

Abb. C-3: Testbenchstrategie mit zwei oder drei VHDL-Modellen

Bei der Beschreibung der Stimuli selbst bietet sich eine kombinierte Zuweisung von mehreren Signalen in einer Signalzuweisung an. Dazu ist i.d.R. ein qualifizierter Ausdruck erforderlich.

Folgendes Beispiel beschreibt die Stimuli für ein NAND2-Modell und überprüft die Antworten jeweils 2 ns danach. Die Testbench ist nach der erstgenannten Strategie angelegt.

```
ENTITY nand2_tb IS                    -- Testbench-Entity
END nand2_tb;                         -- ohne Ports
```



```
ARCHITECTURE strategy_1 OF nand2_tb IS
   COMPONENT nand2_socket
      PORT (in1, in2 : IN bit ; out1 : OUT bit);
   END COMPONENT;
   SIGNAL a,b,c : bit;
   SUBTYPE t2 IS bit_vector (1 TO 2);
BEGIN
----- Instantiierung des Model under Test (mut) ------------
   mut : nand2_socket PORT MAP (a,b,c);
----- Beschreibung der Eingangssignale (Stimuli) -----------
   stimuli_generation: PROCESS
   BEGIN
      -- eine Zuweisung zum Zeitpunkt 0 ns
      (a, b) <= t2'("01") AFTER 10 ns,  -- qualifizierter
                t2'("10") AFTER 20 ns,  -- Ausdruck t2'(...)
                t2'("11") AFTER 30 ns,
                t2'("00") AFTER 40 ns;
      WAIT;
   END PROCESS;
----- Ueberpruefung der Modellantworten (responses) --------
   response_control : PROCESS
   BEGIN
      WAIT FOR 12 ns;                   -- absolut: 12 ns
      ASSERT c='1' REPORT "wrong result" SEVERITY note;
      WAIT FOR 10 ns;                   -- absolut: 22 ns
      ASSERT c='1' REPORT "wrong result" SEVERITY note;
      WAIT FOR 10 ns;                   -- absolut: 32 ns
      ASSERT c='0' REPORT "wrong result" SEVERITY note;
      WAIT FOR 10 ns;                   -- absolut: 42 ns
      ASSERT c='1' REPORT "wrong result" SEVERITY note;
      WAIT;
   END PROCESS;
END strategy_1;
```

Die Signalzuweisung der Stimuli erfolgt im ersten Prozeß komplett zum Zeitnullpunkt. Sie könnte alternativ auch als nebenläufige Anweisung erfolgen. Die Assertions im zweiten Prozeß hingegen müssen, durch `WAIT`-Anweisungen gesteuert, zum entsprechenden Zeitpunkt ausgeführt werden.

Werden bei der Stimulibeschreibung mehrere Einzelanweisungen verwendet, so ist das "Transport"-Verzögerungsmodell einzusetzen, da

sonst durch das "Inertial"-Verzögerungsmodell die vorhergehenden Signalwechsel wieder gelöscht werden:

```
single_step_stimuli_generation : PROCESS
BEGIN
   -- 4 Zuweisungen zum Zeitpunkt 0 ns
   (a, b) <= TRANSPORT t2'("01") AFTER 10 ns;
   (a, b) <= TRANSPORT t2'("10") AFTER 20 ns;
   (a, b) <= TRANSPORT t2'("11") AFTER 30 ns;
   (a, b) <= TRANSPORT t2'("00") AFTER 40 ns;
   WAIT;
END PROCESS;
```

Es können alternativ dazu die Stimuli auch zu den entsprechenden Zeiten erzeugt werden. Dies ist mit `WAIT`-Anweisungen zu steuern. Denkbar wäre in diesem Fall auch eine Kombination mit der Antwortkontrolle:

```
multiple_step_stimuli_generation : PROCESS
BEGIN  -- 4 Zuweisungen zu verschiedenen Zeitpunkten
   WAIT FOR 10 ns; (a, b) <= t2'("01");  -- absolut 10 ns
   WAIT FOR 10 ns; (a, b) <= t2'("10");  -- absolut 20 ns
   WAIT FOR 10 ns; (a, b) <= t2'("11");  -- absolut 30 ns
   WAIT FOR 10 ns; (a, b) <= t2'("00");  -- absolut 40 ns
   WAIT;
END PROCESS;
```

```
stimuli_generation_and_response_control : PROCESS
BEGIN
   WAIT FOR 10 ns; (a, b) <= t2'("01");  -- absolut 10 ns
   WAIT FOR  2 ns;                       -- absolut 12 ns
   ASSERT c='1' REPORT "wrong result" SEVERITY note;
   WAIT FOR  8 ns; (a, b) <= t2'("10");  -- absolut 20 ns
   WAIT FOR  2 ns;                       -- absolut 22 ns
   ASSERT c='1' REPORT "wrong result" SEVERITY note;
   WAIT FOR  8 ns; (a, b) <= t2'("11");  -- absolut 30 ns
   WAIT FOR  2 ns;                       -- absolut 32 ns
   ASSERT c='0' REPORT "wrong result" SEVERITY note;
-------- Fortsetzung auf naechster Seite ------------------
```

```
-------- Fortsetzung von vorhergehender Seite --------------
      WAIT FOR  8 ns; (a, b) <= t2'("00");  -- absolut 40 ns
      WAIT FOR  2 ns;                       -- absolut 42 ns
      ASSERT c='1' REPORT "wrong result" SEVERITY note;
      WAIT;
   END PROCESS;
```

Zur Beschreibung regelmäßiger oder komplexer Stimuli eignen sich die sequentiellen Konstrukte der Sprache VHDL. Es sei hier auf eine der Übungsaufgaben in Teil D zur Erzeugung eines Taktsignals verwiesen. Die Stimuli für einen 8-Bit Decoder können etwa auf folgende Art beschrieben werden:

```
ENTITY dec_stim IS
   PORT (stim : OUT bit_vector(7 DOWNTO 0));
END dec_stim;

ARCHITECTURE behavioral OF dec_stim IS
-- Funktion zur Wandlung einer Integerzahl in e. Bit-Vektor
   FUNCTION integer_to_bit (a: integer) RETURN bit_vector IS
      ...
      ...
   END integer_to_bit;
BEGIN
-- Zyklische Ausgabe von 0 bis 255 -------------------------
   stimuli_generation: PROCESS
      VARIABLE a : integer := 0;
   BEGIN
      WAIT FOR 10 ns;  -- Stimulifrequenz: 100 MHz
      stim <= integer_to_bit(a);
      a := a + 1;
      IF a > 255 THEN a := a - 256; END IF;
   END PROCESS;
END behavioral;
```

1.5 Simulation von VHDL-Gatternetzlisten

Beim Einsatz von VHDL zur Dokumentation und Simulation von elektronischen Schaltungen beschränkt man sich hauptsächlich auf abstrakte Beschreibungsebenen. Typischerweise wird VHDL auf Systemebene, Algorithmischer Ebene und auf der Register-Transfer-Ebene als Eingabeformat für Syntheseprogramme eingesetzt. Bei der Verifikation der generierten Netzliste hingegen setzt man oft noch auf spezielle Digitalsimulatoren. Da VHDL leistungsfähige Konzepte zum Vergleich von Beschreibungen auf RT-Ebene und Logikebene anbietet (Auflösungsfunktionen, Assertions), drängt sich ein Einsatz auch auf dieser Ebene auf. Dies wird momentan jedoch durch zwei Probleme erschwert:

1.5.1 Performance-Nachteile

VHDL-Simulatoren arbeiten auf der Logikebene heute noch wesentlich langsamer als reine Digitalsimulatoren, die speziell für diesen Zweck entwickelt wurden. Führende Softwarehersteller haben jedoch angekündigt, daß die Leistung ihrer VHDL-Simulatoren entweder bald (d.h. bis Mitte 1994) die Leistung der konventionellen Digitalsimulatoren erreichen werde oder daß sie ihren VHDL-Simulator mit dem Digitalsimulator verschmelzen wollen.

1.5.2 Verfügbarkeit von Technologiebibliotheken

Zur Zeit sind kaum verifizierte VHDL-Gattermodelle in technologiespezifischen Bibliotheken verfügbar.

Einen Lösungsansatz zur Beseitigung dieses Engpasses könnte die Initiative VITAL[1] darstellen. Sie dient dem Zweck, ASIC-Bibliotheken für VHDL schneller verfügbar zu machen. Der Grundstein hierfür wurde 1992 beim "VHDL International Users Forum" (VIUF) und auf der "Design Automation Conference" (DAC) gelegt. Im Oktober 1992

---

1 VITAL = VHDL Initiative Towards ASIC Libraries

wurde bereits eine technische Spezifikation vorgelegt. VITAL umfaßt eine Beschreibungsform für das Zeitverhalten in ASIC-Modellen durch ein spezielles Format, SDF ("standard delay format"), und ermöglicht den Zugriff auf Standardbibliotheken der Hersteller.

VITAL erfreut sich einer starken Unterstützung durch CAE- und ASIC-Hersteller. Die Softwarehersteller wollen unmittelbar nach Festlegung des technologieunabhängigen Standards ihre Werkzeuge anpassen. Falls zu diesem Zeitpunkt auch leistungsfähigere Simulatoren zur Verfügung stehen, dürfte die VHDL-Simulation auf Logikebene keine Nachteile gegenüber der Simulation mit speziellen Digitalsimulatoren mehr aufweisen.

2 Synthese

2.1 Synthesearten

Unter Synthese versteht man allgemein den Übergang von der formalen Beschreibung eines Verhaltens zu einer dieses Verhalten realisierenden Struktur. Abhängig vom Abstraktionsgrad der Beschreibung, die als Eingabe für die Synthese dient, spricht man von Logiksynthese, Register-Transfer-Synthese, Algorithmischer Synthese und Systemsynthese. Während die Systemsynthese gegenwärtig noch vom Entwickler von Hand durchzuführen ist, stehen für die übrigen Synthesearten bereits Programme zur Verfügung. Die meisten beschränken sich dabei jedoch auf die Register-Transfer-Ebene oder Logikebene.

2.1.1 Systemsynthese

Auf der Systemebene wird ein Modul global durch seine Leistung und Funktion beschrieben. Die Systemsynthese entwickelt aus einer formalen Spezifikation einzelne Teilprozesse und entscheidet aufgrund der Vorgaben über einen günstigen Parallelitätsgrad in der Abarbeitung der Prozesse. Es ergibt sich eine Grobstruktur aus mehreren Subsystemen.

2.1.2 Algorithmische Synthese

Die Algorithmische Synthese transformiert ein Verhaltensmodell in eine Struktur auf Register-Transfer-Ebene. Das Verhaltensmodell enthält dabei lediglich den Algorithmus, der die Eingabedaten in Ausgabedaten überführt. Die Darstellung erfolgt mittels einer Hardwarebeschreibung, die Sequenzen, Iterationen und Verzweigungen enthält.

Auf der resultierenden Register-Transfer-Ebene wird die Schaltung durch eine Struktur aus Registern, Funktionseinheiten (z.B. Addierer, Multiplizierer, Komparatoren, etc.), Multiplexern und Verbindungs-

strukturen beschrieben. Zur Ansteuerung der Hardwaremodule wird die Zustandsübergangstabelle eines endlichen Zustandsautomaten (FSM = Finite State Machine) generiert.

Grundprinzip der algorithmischen Synthese ist meistens die Umsetzung der algorithmischen Beschreibung in einen Datenfluß- und einen Kontrollflußgraphen (Abb. C-4).

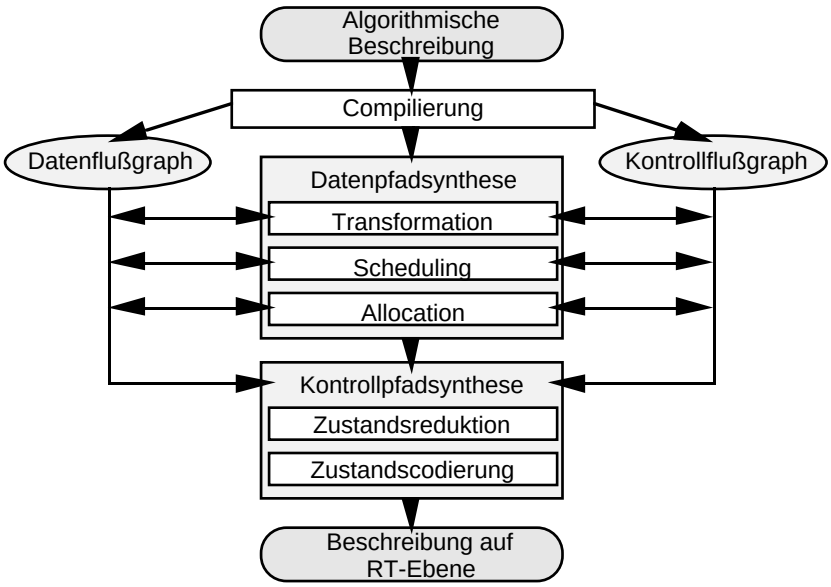

Abb. C-4: Ablauf der Algorithmischen Synthese [BIT 92]

Mit dem Datenflußgraphen werden die einzelnen Operationen, die die Eingangssignale in Ausgangssignale überführen, beschrieben. Die Knoten des Datenflußgraphen repräsentieren die verschiedenen Operationen; Kanten geben Variablen oder Konstanten wieder und definieren die Abhängigkeiten der Operatoren. Der Graph muß nicht zusammenhängend sein, parallele Abläufe sind möglich. Der zeitliche Ablauf der einzelnen Operationen wird dagegen im Kontrollflußgraphen abgebildet. Die Knoten dieses Graphen sind die Zustände des endlichen Automaten, die Kanten die Zustandsübergänge.

Die Synthese des Datenpfades besteht in der Realisierung des spezifizierten Algorithmus mit einer geeigneten Auswahl und Anzahl von Hardwaremodulen (z.B. Addierer, Register, Speicher sowie Multiplexer

und Busse zur Verbindung der Funktionseinheiten). Zur Ansteuerung dieser Module generiert die Kontrollpfadsynthese einen endlichen Automaten, der die Datenpfadregister lädt, Multiplexer und Busse schaltet und die auszuführenden Operationen auswählt.

2.1.3 Register-Transfer-Synthese

Auf Register-Transfer-Ebene wird der Datenfluß einer Schaltung dargestellt. Die Beschreibung enthält neben Funktions- auch Strukturangaben des Entwurfs. Die dabei verwendeten Signale und Zuweisungen an Signale können nahezu direkt in eine Struktur aus Registern und Verarbeitungseinheiten ("Transfers") zwischen den Registern übertragen werden.

Während auf Algorithmischer Ebene eine Schaltung aus **imperativer** Sicht, d.h. der Sicht des Steuerwerks, beschrieben wird, das die einzelnen Aktionen sequentiell zu vorhergehenden Aktionen anstößt, wird das Verhalten auf RT-Ebene aus der **reaktiven** Sicht der Elemente dargestellt. Das bedeutet, daß die einzelnen Objekte "beobachten", ob eine bestimmte Triggerbedingung wahr wird, und dann entsprechende Aktionen ausführen.

Die einzelnen Komponenten der Register-Transfer-Ebene sind keiner Ordnung unterworfen. Als typische Sprachelemente zur Definition der Objekte dienen sog. "guarded commands". Sie repräsentieren Verarbeitungen, die dann aktiviert werden, wenn bestimmte Ereignisse eintreten. Die Register-Transfer-Ebene enthält außerdem ein bestimmtes Synchronisationsschema, das durch die Triggerbedingungen definiert ist.

Ein Beispiel:

> Ausgegangen wird von einer Schaltung, die unter anderem zwei Register enthält, welche bidirektional mit zwei Bussen verbunden sind. Auf einer ALU können Additionen, Subtraktionen und UND-Verknüpfungen durchgeführt werden. Das Operationsergebnis wird in einem dritten Register gespeichert, welches auf die beiden Busse schreiben kann. Die steigende Flanke des Taktes triggert die Operatoren und Register.

Die Register-Transfer-Synthese setzt das Verhaltensmodell eins-zu-eins in eine äquivalente Blockstruktur auf gleicher Ebene um. Ähnlich der algorithmischen Synthese werden Kontroll- und Datenpfad getrennt behandelt.

Bei der Datenpfadsynthese wird zunächst festgestellt, welche Signale gespeichert werden müssen. Dafür werden Flip-Flops bzw. Register angelegt. Die Art der Flip-Flops (D-Flip-Flop, J-K-Flip-Flop, getriggert durch steigende oder fallende Flanke, etc.) gibt die Triggerbedingung und Blockbeschreibung vor. Für die Verarbeitung der nicht zu speichernden Signale werden nur kombinatorische Logik und Verbindungsleitungen vorgesehen.

Durch Analyse der Signalabhängigkeiten können die Verbindungsstrukturen ermittelt werden. Diese bestehen aus dedizierten Leitungen und Bussen sowie aus Treibern und Multiplexern zwischen den Registern und zwischen Registern und Operationseinheiten.

Die Kontrollpfadsynthese erzeugt die Steuerung der Register-Transfers aus den Triggerbedingungen. Wie in der Algorithmischen Synthese wird dazu ein Automat angelegt, um Treiber und Multiplexer anzusteuern. Das Übergangsnetzwerk des Automaten, das durch Boolesche Gleichungen repräsentiert wird, kann mit der anschließenden Logiksynthese implementiert werden.

Aus den Ergebnissen der Daten- und Kontrollpfadsynthese wird meist eine generische Netzliste aus den Elementen einer technologieunabhängigen Bibliothek erzeugt. In dieser Bibliothek sind die Gatterstrukturen der einzelnen Elemente hinterlegt. Die Abbildung der Gatter auf eine technologiespezifische Bibliothek geschieht im Technology Mapping der Logiksynthese.

2.1.4 Logiksynthese

Bei der Logiksynthese werden die realisierungsunabhängigen Booleschen Beschreibungen der kombinatorischen Netzwerke (Multiplexer, Operatoren, Übergangsnetzwerke der Automaten, etc.) optimiert und anschließend mit den Elementen der gewählten Zieltechnologie aufgebaut. Diese technologiespezifische Netzliste wird wiederum optimiert, um die Benutzervorgaben zu erreichen.

Folgende Einzelschritte laufen bei der Logiksynthese mit kommerziellen Werkzeugen im allgemeinen ab:

Flattening

Alle Zwischenvariablen der Booleschen Ausdrücke werden zunächst entfernt und alle Klammern aufgelöst. Damit erhält man beispielsweise eine zweistufige AND-OR-INVERT-Darstellung:

- Vor dem Flattening:
  $f = f1 \wedge f2;$ mit: $f1 = a \vee (e \wedge (c \vee d));$ $f2 = c \vee b$
- Nach dem Flattening:
  $f = (a \wedge c) \vee (a \wedge b) \vee (c \wedge e) \vee (e \wedge d \wedge c) \vee (e \wedge c \wedge b) \vee (e \wedge d \wedge b)$

Das Flattening löst also die vorgegebene Struktur der Logik auf. Die Auflösung eines vorher strukturierten Blockes in seine Produktterme kann eventuell bzgl. Geschwindigkeit und Fläche schlechtere Ergebnisse liefern. Bei unstrukturierter, krauser Logik ist es aber möglich, durch die anschließende Minimierung sowohl schnellere als auch kleinere Schaltungen zu erzeugen. Weil durch das Auflösen der Zwischenterme große Datenmengen entstehen, kann in den meisten Synthesesystemen der Grad des Flattenings vorgegeben werden.

Logikminimierung

Die Darstellung aus Produkttermen wird mit Minimierungsverfahren, wie z.B. dem Nelson-Verfahren, weiterverarbeitet. Jede Funktion kann dabei einzeln oder innerhalb eines Funktionsbündels minimiert werden. Die Anzahl der Produktterme reduziert sich dadurch und redundante Logik wird entfernt.

Structuring

Beim Structuring oder Factoring werden gemeinsame Unterausdrücke ausgeklammert und als temporäre Variablen verwendet. Die Schaltung erhält erneut eine Struktur. Dabei wird zunächst ein Liste angelegt, die die möglichen Faktoren enthält. Die Bewertung der Faktoren (benötigte Halbleiterfläche, Anzahl ihrer Verwendung) wird so oft wiederholt, bis kein neuer Faktor ermittelt werden kann, der die Schaltung verbessert. Die Faktoren, die die Logik am stärksten reduzieren, werden zu temporären Variablen. Ein Beispiel zum Structuring:

- Vor dem Structuring:
  $f = (a \wedge d) \vee (a \wedge e) \vee (b \wedge c \wedge d) \vee (b \wedge c \wedge e)$
- Nach dem Structuring:
  $f = t0 \wedge t1;$ mit: $t0 = a \vee (b \wedge c);$ $t1 = d \vee e$

Auswirkungen der Optimierungen

Die richtige Anwendung der verwendeten Strategien beim Einsatz eines Synthesewerkzeuges hat entscheidenden Einfluß auf das Syntheseergebnis. Spaltet man ein Design durch Flattening vollständig in Produktterme auf, minimiert anschließend die einzelnen Funktionen getrennt und verzichtet auf Structuring, so erhält man eine große, aber sehr schnelle Schaltung, da nur wenige Logikstufen zwischen den Ein- und Ausgängen liegen. Wird die ursprüngliche Struktur allerdings beibehalten und zusätzlich das Structuring verwendet, so kann eine kleine, aber langsame Schaltung entstehen.

Die Strategie, mit der eine Schaltung optimiert werden kann, hängt von verschiedenen Faktoren ab, wie ihre Komplexität, die Anzahl der Ein- und Ausgänge oder die Güte der vorgegebenen Struktur. Dadurch bietet es sich an, mit Synthesewerkzeugen verschiedene Möglichkeiten auszuprobieren.

Technology Mapping

Vor dem Technology Mapping ist die synthetisierte Schaltung noch technologieunabhängig. Das Technology Mapping setzt die optimierte Logik und die Flip-Flops in die Gatter einer bestimmten Technologiebibliothek um.

Zunächst wird die Schaltung vollständig mit technologiespezifischen Gattern abgebildet. Durch lokales Neuarrangieren von Komponenten oder Verwendung von Bausteinen mit unterschiedlicher Anzahl an Eingängen wird versucht, die "constraints" des Entwicklers zu erfüllen. Die benutzerdefinierten Einschränkungen beziehen sich neben einer maximal zulässigen Fläche, maximaler Laufzeit oder Taktrate auch auf die Setup- und Hold-Zeiten für die Flip-Flops. Für diese Parameter wird eine Kostenfunktion erstellt, die durch geeignete Partitionierung und lokale Substitutionen von Gatterkonfigurationen minimiert wird. Hierbei werden heuristische Techniken angewandt.

Ein wichtiges Leistungsmerkmal eines Mapping-Algorithmus ist seine universelle Anwendbarkeit auf verschiedene Technologiebibliotheken, die sehr unterschiedliche Komplexgatter und Makrofunktionen enthalten können. Ein Problem ist die schnelle Zunahme an benötigter Rechenzeit bei großen Schaltungen mit vielen Gattern.

Abb. C-5 zeigt zwei Beispiele zum Technology Mapping. Beispielsweise muß bei diesem Vorgang eine logische Verknüpfung mit vier Eingangsvariablen durch eine funktional äquivalente Verknüpfung aus drei Gattern mit jeweils zwei Eingangsvariablen ersetzt werden, da nur diese in der Technologiebibliothek verfügbar sind (oberes Beispiel). Eine Einsparung von Gattern beim Mapping ist unter anderem möglich, wenn in der Bibliothek Module mit negierten Ausgängen vorliegen (unteres Beispiel).

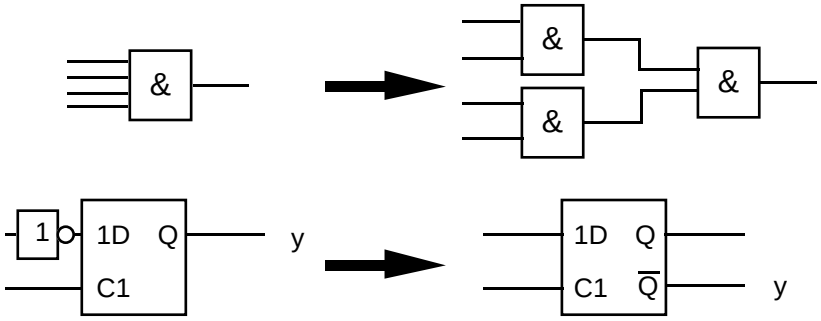

Abb. C-5: Beispiele zum Technology Mapping

Auch manuell erstellte Netzlisten können mit den Logiksynthese-Werkzeugen optimiert oder von einer Technologiebibliothek auf eine andere umgesetzt werden. Das Ergebnis der Synthese kann als technologiespezifische Netzliste (z.B. im EDIF oder VHDL-Format) ausgegeben werden.

2.2 Einsatz der Syntheseprogramme

In diesem und in allen weiteren Abschnitten zur Synthese von VHDL-Beschreibungen wird nur noch auf Werkzeuge eingegangen, die eine Umsetzung von einer RT-Beschreibung in eine Gatternetzliste unter-

stützen, da bei der Anwendung von VHDL z.Zt. diese Werkzeuge die größte Bedeutung haben.

2.2.1 Umstellung von Schematic Entry

Der Einsatz von Syntheseprogrammen bedeutet nicht, daß die komplette bisherige Entwurfsumgebung durch eine neue ersetzt wird. Nach [SYN 92] kann eine Umstellung vom gewöhnlichen Schematic-Entry (Graphische Schaltungseingabe auf Logikebene) auf die Synthesewerkzeuge in mehreren Schritten erfolgen:

① Nachträgliche Optimierung der manuell erstellten Schaltungen mit Syntheseprogrammen,

② nachträgliche Optimierung und Verbesserung der Testbarkeit,

③ teilweise Erfassung des Entwurfs mit VHDL und Synthese,

④ volle Beschreibung des Entwurfs mit VHDL und Synthese.

Es ist zu beachten, daß auch durch die Verwendung von Synthesewerkzeugen Iterationszyklen nicht vermieden werden können. Diese spielen sich lediglich auf einer anderen Ebene ab (*"We´ve got still the same problems, but on a higher level"*). Die Synthesewerkzeuge bieten dem Entwickler aber die Möglichkeit, die Ebene des Entwurfs von der Logikebene auf die abstraktere Register-Transfer-Ebene zu verlagern, wodurch Komplexitäten besser beherrscht werden. Ganz losgelöst von der "Hardware" (der Logikebene) kann sich jedoch kein Entwickler bewegen, weil nur über die Analyse des Syntheseergebnisses auf Logikebene auf eine geeignete VHDL-Modellierungstechnik auf RT-Ebene geschlossen werden kann.

2.2.2 Zielsetzung

Eine der wichtigsten Überlegungen beim Einsatz eines Syntheseprogramms gilt dem Ziel, Einfluß auf die Optimierung der Schaltung zu nehmen.

2.2.2.1 Optionen und Randbedingungen

Jedes Syntheseprogramm bietet die Möglichkeit, durch Einstellung von bestimmten Optionen Randbedingungen bzw. Optimierungskriterien bei der Synthese vorzugeben (Setzen von "constraints"). Dies können neben der Auswahl von optimaler Fläche oder optimaler Laufzeit meist noch Laufzeitgrenzen für einzelne Pfade und weitere Randbedingungen, wie z.B. die Vorgabe einer Zustandscodierung, sein.

Bei vielen Syntheseprogrammen zeigt sich jedoch, daß mit den jeweiligen Einstellungen der Randbedingungen (Fläche, Laufzeit) nicht optimal auf die Schaltung Einfluß genommen werden kann. Für die Generierung der Schaltung in Abhängigkeit von den Randbedingungen werden nämlich teilweise Heuristiken verwendet, so daß man nie sicher sein kann, wirklich die optimale Schaltung erzeugt zu haben. Eine Schaltung, die beispielsweise eine Laufzeitvorgabe von 0 ns (also möglichst schnell) hatte, kann langsamer sein als eine mit der Laufzeitvorgabe 30 ns.

2.2.2.2 Modellierungsstil

Durch geschickte Verwendung von VHDL-Sprachelementen kann bereits bei der Modellierung eine Entscheidung über eine mehr oder weniger geeignete Schaltungsarchitektur getroffen werden. Dazu ist erstens eine detaillierte Kenntnis von VHDL und zweitens das Wissen über die spätere Umsetzung der VHDL-Sprachkonstrukte durch das eingesetzte Synthesewerkzeug erforderlich.

Beispiel:

> Einen Zähler kann man beispielsweise als Zustandsautomaten oder als sequentiellen Zähler modellieren, wobei nicht immer eine Beschreibungsart die optimale ist. In vielen Fällen, insbesondere bei 2er-Potenzen als Zähllängen, ergibt sich bei der sequentiellen Beschreibung ein besseres Ergebnis, während bei Zähllängen, die knapp über einer 2er-Potenz liegen (z.B. 129), die Realisierung als Automat aufgrund der umfangreichen Minimierung der Übergangslogik (viele freie Zustände) günstiger ist.

Im folgenden soll deshalb anhand einiger VHDL-Beispiele illustriert werden, welchen Einfluß der Modellierungsstil und die gewählten Randbedingungen ("Constraints") auf das Syntheseergebnis haben.

Für diese Betrachtungen wurden mehrere kommerzielle Syntheseprogramme herangezogen, um programmspezifische Besonderheiten ausmitteln zu können. Einschränkungen hinsichtlich der Verwendbarkeit von VHDL-Anweisungen und des Beschreibungsstils werden mit zukünftigen Programmversionen zunehmend geringer werden.

2.3 Synthese von kombinatorischen Schaltungen

In diesem Abschnitt soll dargestellt werden, wie VHDL-Modelle von kombinatorischen Funktionen in Schaltungsarchitekturen umgesetzt werden. Nähere Angaben finden sich in den Dokumentationen zu den jeweiligen Synthesewerkzeugen.

2.3.1 Einführung

In VHDL gibt es zwei Möglichkeiten, kombinatorische Schaltungen zu beschreiben: die Modellierung mit Hilfe nebenläufiger Anweisungen und mit Hilfe sequentieller Anweisungen (innerhalb von Prozessen und Unterprogrammen).

Am einfachen Beispiel eines achtfachen NAND-Gatters (siehe Abb. C-6) sollen vier verschiedene Beschreibungsarten gezeigt werden.

Dieses und alle weiteren VHDL-Beispiele verwenden dabei das IEEE-Package `std_logic_1164` mit dem 9-wertigen Logiktyp `std_ulogic`.

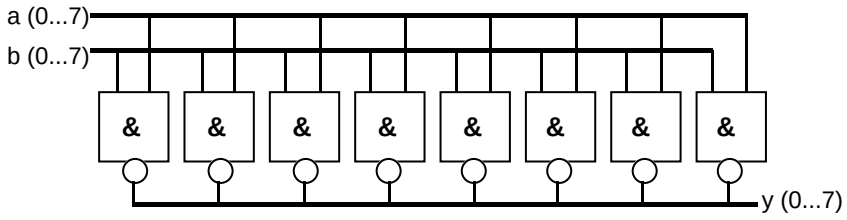

Abb. C-6: Schaltbild des 8-fach NAND-Gatters

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY nand2 IS
   PORT (a,b: IN  std_ulogic_vector (0 TO 7);
         y:   OUT std_ulogic_vector (0 TO 7));
END ENTITY;
```

Die Architekturen `one` und `two` verwenden die überladene Funktion `NAND` aus dem Package `std_logic_1164` in vektorieller und Einzelbitversion.

```
ARCHITECTURE one OF nand2 IS
BEGIN
   y <= a NAND b;
END one;
```

```
ARCHITECTURE two OF nand2 IS
BEGIN
   PROCESS (a,b)
   BEGIN
      FOR i IN a'RANGE LOOP
         y(i) <= a(i) NAND b(i);
      END LOOP;
   END PROCESS;
END two;
```

Die folgenden beiden Architekturen (`three` und `four`) verwenden logische Gleichungen zur Beschreibung der Funktion. Sie haben den Nachteil, daß im Rahmen einer funktionalen Simulation beim Auftreten von Signalen wie `'X'`, `'Z'` oder `'H'` an einem Eingang das Verhalten des Gatters nicht korrekt modelliert ist, da in diesem Fall der Ausgang den Wert `'1'` annehmen würde. Infolgedessen ist es hier zu empfehlen, den vordefinierten `NAND`-Operator aus dem IEEE-Package (wie in Architektur `one` oder `two`) zu verwenden.

```
ARCHITECTURE three OF nand2 IS
BEGIN
   y(0) <= '0' WHEN a(0)='1' AND b(0)='1' ELSE '1';
   ...
   y(7) <= '0' WHEN a(7)='1' AND b(7)='1' ELSE '1';
END three;
```

```
ARCHITECTURE four OF nand2 IS
BEGIN
   PROCESS (a,b)
   BEGIN
      FOR i IN a'RANGE LOOP
         CASE a(i) & b(i) IS
            WHEN "11"   => y(i) <= '0';
            WHEN OTHERS => y(i) <= '1';
         END CASE;
      END LOOP;
   END PROCESS;
END four;
```

Bei der Synthese einer derart einfachen Beschreibung bestehen keine Unterschiede in der Implementierung der verschiedenen Beschreibungsarten. Für die Architekturen `three` und `four` wird zwar zunächst eine recht komplizierte Schaltungsarchitektur generiert, diese dann aber bei der Optimierung wieder in acht einfache NAND-Gatter aufgelöst. Man benötigt mit diesen Versionen lediglich größere Rechenzeiten bei der Synthese.

2.3.2 Verzweigungen

Anhand des folgenden Modells wird betrachtet, wie die Verzweigungsanweisungen `IF` und `CASE` in Hardware umgesetzt werden:

```
ENTITY if_und_case IS
  PORT (i:     IN  integer RANGE 0 TO 9;
        a,b,c: IN  std_ulogic_vector (7 DOWNTO 0);
        z:     OUT std_ulogic_vector (7 DOWNTO 0) );
END if_und_case;
```

```
ARCHITECTURE if_variante OF if_und_case IS
BEGIN
  p1: PROCESS (i,a,b,c)
  BEGIN
    IF    (i = 3) THEN z <= a;
    ELSIF (i < 3) THEN z <= b;
    ELSE               z <= c;
    END IF;
  END PROCESS p1;
END if_variante;
```

```
ARCHITECTURE case_variante OF if_und_case IS
BEGIN
  p1: PROCESS (i,a,b,c)
  BEGIN
    CASE i IS
      WHEN 3      => z <= a;
      WHEN 0 TO 2 => z <= b;
      WHEN OTHERS => z <= c;
    END CASE;
  END PROCESS p1;
END case_variante;
```

Die beiden Architekturen `if_variante` und `case_variante` sind funktional identisch. Bei der Synthese jedoch werden unterschiedliche Schaltungen erzeugt. Das liegt daran, daß eine `IF`-Anweisung grundsätzlich eine Priorität beinhaltet, nämlich die bevorzugte Abfrage des ersten Zweiges (hier: `i=3`). Bei der `CASE`-Anweisung sind dagegen alle Zweige gleichberechtigt. Nachstehende Schaltungen

ergeben sich zuerst aufgrund der obigen Beschreibungen. Bei der anschließenden Optimierung werden aber dann in der Regel wieder identische Schaltungen erzeugt.

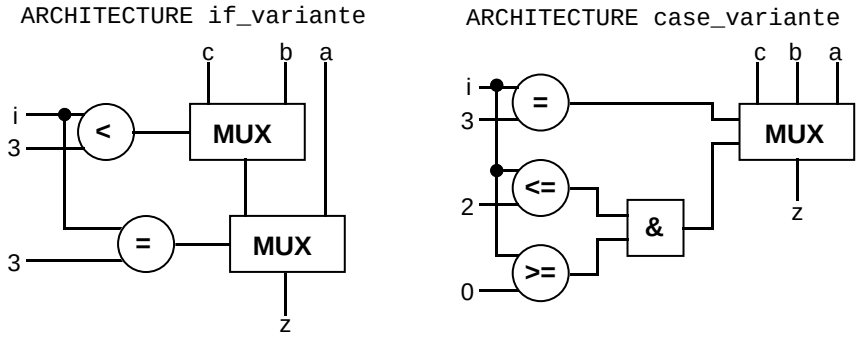

Abb. C-7: Schaltbild der IF- und der CASE-Variante

2.3.3 Signale und Variablen

Bei der Beschreibung von Algorithmen ist normalerweise die Speicherung von Zwischenergebnissen notwendig. Dazu könnten prinzipiell Signale oder Variablen verwendet werden. Da Zuweisungen an Signale immer erst ein "Delta" später wirksam werden, führt der Einsatz von Signalen als Zwischenspeicher in Algorithmen jedoch häufig zu Modellierungsfehlern und damit zu unerwarteten Syntheseergebnissen.

Zur Illustration soll hier ein Beispiel gezeigt werden, bei dem mit Hilfe einer Schleife eine regelmäßige Schaltungsstruktur (XOR-Kette) beschrieben werden soll:

```
ENTITY kette IS
  PORT ( hbyte: IN  std_ulogic_vector (0 TO 3) := "0000";
         value: OUT std_ulogic );
END kette;
```

```
ARCHITECTURE richtig OF kette IS
BEGIN
  PROCESS (hbyte)
    VARIABLE merker: std_ulogic := '0';
  BEGIN
    merker := '0';
    FOR i IN hbyte'RANGE LOOP
      merker := merker XOR hbyte(i);
    END LOOP;
    value <= merker;
  END PROCESS;
END richtig;
```

```
ARCHITECTURE falsch OF kette IS
  SIGNAL merker: std_ulogic := '0';
BEGIN
  PROCESS (hbyte)
  BEGIN
    FOR i IN hbyte'RANGE LOOP
      merker <= merker XOR hbyte(i);
    END LOOP;
  END PROCESS;
  value <= merker;
END falsch;
```

Bei der Architektur `richtig` wird die Variable `merker` in der Schleife der Reihe nach mit allen Bits des Signals `hbyte` verknüpft, so daß primär bei der Synthese eine Kette von XOR-Gattern entsteht (siehe Abb. C-8).

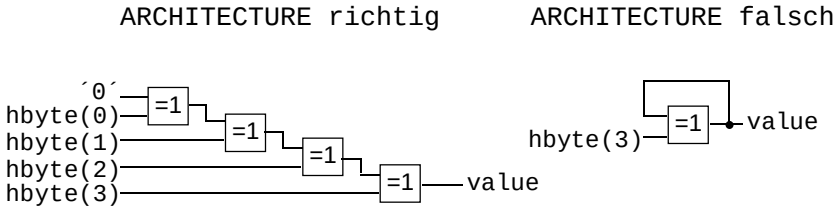

Abb. C-8: Syntheseergebnisse des VHDL-Modells kette

Bei der Architektur `falsch` hingegen werden Signale eingesetzt. Da Signalzuweisungen im Prozeß nicht sofort ausgeführt werden, werden die Zuweisungen an `merker` nicht wirksam, so daß nur noch eine Zuweisung, nämlich "`merker <= merker XOR hbyte(3)`" verbleibt, also ein einzelnes rückgekoppeltes XOR-Gatter.

2.3.4 Arithmetische Operatoren

Um die Probleme bei der Umsetzung von arithmetischen Operatoren näher zu beleuchten, wird das Beispiel eines 4-Bit-Volladdierers aufgegriffen. Dessen Schnittstellenbeschreibung lautet:

```
ENTITY addierer IS
  PORT ( a,b:  IN  std_logic_vector (3 DOWNTO 0);
         cin:  IN  std_logic;
         s:    OUT std_logic_vector (3 DOWNTO 0);
         cout: OUT std_logic );
END addierer;
```

Die Eingangssignale `a` und `b` stellen die Summanden des Addierers dar. Die Ports `cin` und `cout` entsprechen dem Übertragsbit ("carry") auf der Ein- bzw. Ausgangsseite. Das vier Bit breite Ausgangssignal `s` schließlich steht für die Summe.

Im folgenden werden verschiedene VHDL-Beschreibungen des Addierer-Verhaltens betrachtet. Die Architektur `zwei_plus` beschreibt den Addierer sehr einfach, indem die beiden Eingangssignale und der Carry-Eingang durch Hinzufügen von `"0"`-Stellen mit zwei Pluszeichen verknüpft werden:

```
ARCHITECTURE zwei_plus OF addierer IS
  SIGNAL temp: std_logic_vector (4 DOWNTO 0);      -- 5 Bit
BEGIN
  temp <= ("0" & a) + ("0" & b) + ("0000" & cin);
  cout <= temp(4);                              -- Uebertrag
  s <= temp(3 DOWNTO 0);                        -- Summe
END zwei_plus;
```

Da zu erwarten ist, daß manche Syntheseprogramme aus zwei Pluszeichen auch zwei Addierer aufbauen, wird die Beschreibung in der Architektur `ein_plus` auf ein Pluszeichen reduziert:

```
ARCHITECTURE ein_plus OF addierer IS
  SIGNAL temp: std_logic_vector (5 DOWNTO 0);  -- 6 Bit
BEGIN
  temp <= ("0" & a & cin) + ("0" & b & "1");
  cout <= temp(5);                              -- Uebertrag
  s <= temp(4 DOWNTO 1);                        -- Summe
END ein_plus;
```

Als Alternative zu diesen beiden funktionalen Beschreibungen kann man aber auch "hardware-orientiert" modellieren und beispielsweise direkt eine "Ripple-Carry-Struktur" vorgeben:

```
ARCHITECTURE ripple OF addierer IS
  SIGNAL c: std_logic_vector (3 DOWNTO 0);
BEGIN
  s <= (a XOR b) XOR (c(2 DOWNTO 0) & cin);
  c <= ((a XOR b) AND (c(2 DOWNTO 0) & cin)) OR (a AND b);
  cout <= c(3);
END ripple;
```

Natürlich kann man alternativ auch eine "Carry-Look-Ahead-Struktur" beschreiben:

```
ARCHITECTURE cla OF addierer IS
  SIGNAL c:   std_logic_vector (2 DOWNTO 0);
  SIGNAL p,g: std_logic_vector (3 DOWNTO 0);
BEGIN
  p <= a XOR b;
  g <= a AND b;
  s <= p XOR (c & cin);
  c(0) <= g(0) OR (p(0) AND cin);
  c(1) <= g(1) OR (p(1) AND c(0));
  c(2) <= g(2) OR (p(2) AND c(1));
  cout <= g(3) OR (p(3) AND c(2));
END cla;
```

Bei der Synthese der verschiedenen Architekturen ergibt sich, daß mit `ein_plus` meistens bessere Ergebnisse erreicht werden als mit `zwei_plus`. Abb. C-9 zeigt die Ergebnisse (Fläche in Gatteräquivalenten, GÄ; Laufzeit in ns) bei der Synthese von 4-Bit- und 32-Bit-Addierern mit den vier unterschiedlichen Architekturen, jeweils auf minimale Fläche (4F, 32F) bzw. minimale Laufzeit (4L, 32L) optimiert:

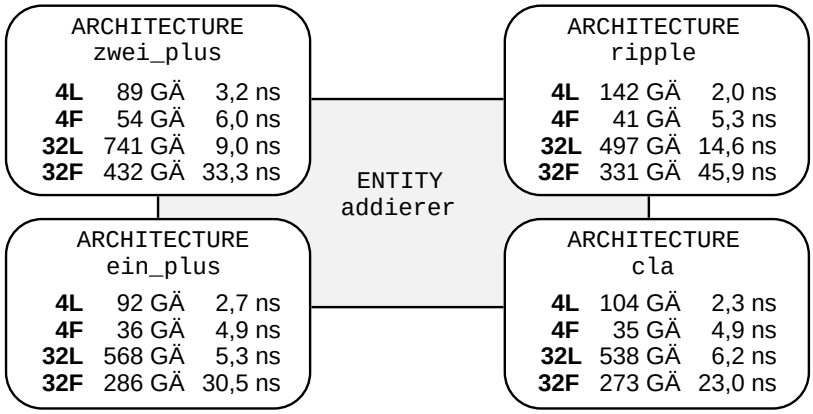

Abb. C-9: Syntheseergebnis verschiedener Addiererarchitekturen

Abb. C-9 zeigt, daß sich das Syntheseergebnis sowohl durch die "constraints" (Optimierungsbedingungen) als auch durch den Modellierungsstil erheblich beeinflussen läßt. Dies gilt für alle betrachteten

Syntheseprogramme. Besonders fällt hier auf, daß die Architektur `cla` (Carry-Look-Ahead) meist auf eine kleinere Schaltung als die Ripple-Carry-Architektur führt. Beim manuellen Entwurf hätte man durch Einsatz einer Ripple-Carry-Architektur eine kleinere Schaltung erzielt als mit der Carry-Look-Ahead-Architektur. Dies zeigt, daß die Syntheseprogramme die gegebenen Gleichungen nicht als Addierer erkennen können und in diesem Fall die Gleichungen für den Carry-Look-Ahead-Addierer offensichtlich mehr Spielraum für die Flächenoptimierung bieten.

Bei neueren Versionen der Syntheseprogramme ist die Verwendung von arithmetischen Operatoren sinnvoller ist als die Angabe von logischen Gleichungen, da die Programme oft optimierte Module für die Synthese von arithmetischen Operatoren zur Verfügung stellen.

2.3.5 Schleifen

Von den drei Schleifenarten, die in VHDL möglich sind (`FOR`-Schleife, `WHILE`-Schleife und Endlosschleife) wird nur die `FOR`-Schleife von den Syntheseprogrammen uneingeschränkt unterstützt, weil hier die Anzahl der Schleifendurchläufe vor der Ausführung der Schleife bekannt ist.

Das folgende Beispiel zeigt, wie durch einen für die Simulation korrekten, aber für die Synthese ungünstigen Einsatz von Schleifen ein hoher Flächenaufwand impliziert wird. Dazu soll das Modell eines Barrel-Shifters betrachtet werden, der 8-Bit breite Daten (`data_in`) in Abhängigkeit von einem Steuersignal (`adresse`) um bis zu sieben Stellen rotieren kann.

```
ENTITY barrel IS
   PORT (data_in  : IN  std_ulogic_vector (7 DOWNTO 0);
         adresse  : IN  std_logic_vector  (2 DOWNTO 0);
         data_out : OUT std_ulogic_vector (7 DOWNTO 0));
END barrel;
```

```
ARCHITECTURE eins OF barrel IS
   SIGNAL adr : integer RANGE 0 TO 7 := 0;
BEGIN
   adr <= conv_integer(adresse); -- Konvertierung in integer
   PROCESS (data_in, adr)
   BEGIN
      FOR i IN 7 DOWNTO 0 LOOP
         data_out((i + adr) mod 8) <= data_in(i);
      END LOOP;
   END PROCESS;
END eins;
```

```
ARCHITECTURE zwei OF barrel IS
   SIGNAL adr : integer RANGE 0 TO 7;
BEGIN
   PROCESS (data_in, adresse)
      VARIABLE puffer : std_ulogic_vector(7 DOWNTO 0);
   BEGIN
      puffer := data_in;
      FOR i IN 0 TO 2 LOOP
         IF (adresse(i) = '1') THEN
            puffer := puffer(7-2**i DOWNTO 0) &
                      puffer(7 DOWNTO 7-2**i+1);
         END IF;
      END LOOP;
      data_out <= puffer;
   END PROCESS;
END zwei;
```

Bei der Synthese der Architektur `eins` ist zu erkennen, daß für jeden Schleifendurchlauf ein eigener Addierer generiert wird, also insgesamt acht Addierer entstehen. Zwar können diese Addierer bei entsprechender Optimierungseinstellung zum Teil wieder reduziert werden, aber die Schaltung bleibt für immer größer und langsamer als bei geschickter Modellierung unter Verwendung einer Puffervariablen (Architekur `zwei`). Eine Untersuchung der Synthese mit fünf verschiedenen Beschreibungsvarianten für einen 32-Bit-Linksrotierer ergab Gatteräquivalente zwischen 480 und 3096 und Laufzeiten zwischen 9,5 und 72,5 ns. Hieraus wird ersichtlich, daß man, zumindest in einigen extre-

men Fällen, durch ungeschickte Modellierung einige 100% an zusätzlicher Logik erzeugen kann.

Die Ergebnisse können dahingehend verallgemeinert werden, daß Schleifen zur Verarbeitung von einzelnen Signalen vermieden werden sollten, da dies zu einer Vervielfachung der Hardware führt.

2.3.6 Zusammenfassung

Nach Untersuchung einer Vielzahl von kombinatorischen Schaltungen unter Verwendung von diversen Modellierungsarten zeichnen sich folgende Ergebnisse ab:

- Bei **kleinen Schaltungen** hängen die Syntheseergebnisse **nicht** von der **Art der Beschreibung** ab. Es spielt also keine Rolle, ob eine bestimmte Funktion mit `IF`, `CASE`, `SELECT` oder durch direkte Angabe der logischen Funktion mit arithmetischen und Booleschen Operatoren beschrieben wird.
- Bei **großen Schaltungen** hingegen ist eine tabellarische Beschreibung der Funktion mit Hilfe von sequentiellen Anweisungen (`IF`, `CASE` etc.) kaum noch möglich. Hier müssen möglichst **einfache und kompakte Operatoren** gewählt werden. Gegenüber eigenen Beschreibungen der Funktionalität haben diese Operatoren darüberhinaus den Vorteil, daß sie vom Syntheseprogramm besser interpretiert werden können, d.h. durch geeignetes Setzen von "constraints" kann man i.d.R. die Optimierungsziele leichter erreichen.
- Innerhalb algorithmischer Beschreibungen sollten **Variablen** verwendet werden. Das Ergebnis des Algorithmus wird dann den **Signalen** oder Ports zugewiesen (vgl. Beispiel "`kette`").
- In einigen Fällen kann man durch eine komplexere, **hardwarenähere Modellierung** auch ein besseres Syntheseergebnis erzielen (sieht man vom Beispiel des "Ripple-Carry-Addierers" einmal ab).
- Beim Einsatz von **Schleifen** zur Abarbeitung der einzelnen Elemente eines Vektors wird oft vielfache Logik erzeugt (vgl. Bsp. "`barrel`"). Schleifen sollten deshalb nur dann eingesetzt werden, wenn genau dieses erwünscht ist (vgl. Bsp. "`kette`").

2.4 Synthese von sequentiellen Schaltungen

2.4.1 Latches

Gegeben sei folgende VHDL-Beschreibung:

```
ENTITY was_ist_das IS
  PORT ( a,b: IN  bit;
         c:   OUT bit );
END was_ist_das;

ARCHITECTURE behave OF was_ist_das IS
BEGIN
  PROCESS (a,b)
  BEGIN
    IF (a = '0') THEN c <= b;
    END IF;
  END PROCESS;
END behave;
```

Wenn das Signal `a` auf `'0'` liegt, wird dem Ausgang `c` der Wert des Eingangs `b` zugewiesen. Im Falle `a = '1'` erfolgt keine explizite Zuweisung des Ausgangs `c`, so daß der vorhergehende Wert beibehalten wird. Dies bedeutet aber wiederum, daß der Wert gespeichert werden muß. Folglich beschreibt das obige Modell ein Speicherelement, ein D-Latch (Abb. C-10). Bei diesem Latch ist `a` das Enable-Signal, `b` der Dateneingang und `c` der Ausgang. Bei `a = '0'` ist das Latch transparent, so daß alle Änderungen des Eingangs `b` sofort am Ausgang `c` erscheinen. Bei `a = '1'` ist das Latch gesperrt (Halten), so daß Änderungen des Eingangs `b` sich nicht auf den Ausgang `c` auswirken.

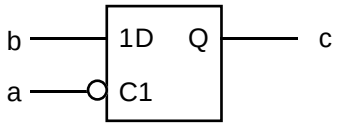

Abb. C-10: Schaltbild eines D-Latch

Aus dieser Beschreibung kann man nicht nur ableiten, wie man prinzipiell ein Latch modelliert, sondern auch die Gefahr der Synthese unerwünschter Speicherelemente bei unvollständigen IF-Anweisungen in VHDL-Modellen erkennen. Immer dann, wenn in einer IF-Anweisung bestimmte Signale nur in einem Teil der Zweige auf der linken Seite von Signalzuweisungen stehen, muß ein Speicherelement erzeugt werden. Dies gilt auch für unvollständige Zuweisungen in CASE-Anweisungen.

2.4.2 Flip-Flops

Flip-Flops unterscheiden sich von Latches durch ihre **Taktflankensteuerung**. Zur Modellierung muß ein Pegelübergang an einem Taktsignal erkannt werden, wozu sich das VHDL-Attribut EVENT eignet. Dieses Attribut bezieht man auf das Taktsignal und plaziert es in einem Prozeß entweder in einer WAIT- oder in einer IF-Anweisung:

```
ENTITY dff IS
  PORT (clk,d: IN  std_ulogic;
        q:     OUT std_ulogic);
END dff;
```

```
ARCHITECTURE variante1 OF dff IS
BEGIN
  PROCESS
  BEGIN
    WAIT UNTIL clk'EVENT AND clk = '1';
    q <= d;
  END PROCESS;
END variante1;
```

```
ARCHITECTURE variante2 OF dff IS
BEGIN
  PROCESS (clk)
  BEGIN
    IF clk'EVENT AND clk = '1' THEN
      q <= d;
    END IF;
  END PROCESS;
END variante2;
```

Beide Architekturen beschreiben ein D-Flip-Flop. Streng genommen sind sie jedoch nicht ganz korrekt, denn der Fall eines Wechsels am Signal `clk` von `'X'` nach `'1'` ist nicht erfaßt. Die Beschreibungen würden dann fälschlich eine steigende Taktflanke erkennen. Eine zusätzliche Überprüfung, ob das Taktsignal vorher `'0'` war, kann mit Hilfe des Attributs `LAST_VALUE` geschehen:

```
ARCHITECTURE variante3 OF dff IS
BEGIN
  PROCESS
  BEGIN
    WAIT UNTIL clk'EVENT AND clk = '1'
                         AND clk'LAST_VALUE = '0';
    q <= d;
  END PROCESS;
END variante3;
```

```
ARCHITECTURE variante4 OF dff IS
BEGIN
  PROCESS (clk)
  BEGIN
    IF clk'EVENT AND clk = '1' AND clk'LAST_VALUE = '0'
      THEN q <= d;
    END IF;
  END PROCESS;
END variante4;
```

Da eine Erkennung von steigenden oder fallenden Flanken eines Signals häufig benötigt wird, sind die zwei Funktionen `RISING_EDGE` und `FALLING_EDGE` in das IEEE-Package integriert worden. Sie enthalten die Beschreibung gemäß `variante4` und geben ein Signal vom Typ `boolean` zurück, welches `true` ist, wenn eine steigende bzw. fallende Flanke erkannt wurde. Allerdings wird das Attribut `LAST_VALUE`, und damit auch diese beiden Funktionen, nicht von allen Syntheseprogrammen unterstützt.

Wenn man statt des D-Flip-Flops ein T-Flip-Flop (Toggle-Flip-Flop) beschreiben möchte, kann man die folgende Architektur verwenden.

```
ENTITY t_ff IS
  PORT (clk, enable: IN     std_ulogic;
        q:           BUFFER std_ulogic := '0');
END t_ff;
```

```
ARCHITECTURE behavioral OF t_ff IS
BEGIN
  PROCESS
  BEGIN
    WAIT UNTIL clk'EVENT AND clk = '1' AND
      clk'LAST_VALUE = '0';
    IF enable = '1' THEN q <= not q;
    END IF;
  END PROCESS;
END behavioral;
```

Mit dieser Beschreibung wird von den Syntheseprogrammen entweder ein Toggle-Flip-Flop oder, falls ein solches in der Technologiebibliothek nicht vorhanden ist, ein D-Flip-Flop mit vorgeschaltetem XOR-Gatter eingesetzt.

Zu beachten ist, daß gewünschte **asynchrone Eingänge** von Speicherelementen nicht automatisch von einem Syntheseprogramm erzeugt werden, sondern in VHDL beschrieben werden müssen. Am Beispiel eines D-Flip-Flops mit asynchronem Rücksetzeingang sei dies gezeigt:

```
ENTITY dff IS
  PORT (clk,d,reset: IN  std_ulogic;
        q:           OUT std_ulogic );
END dff;
```

```
ARCHITECTURE async_reset OF dff IS
BEGIN
  PROCESS (clk,reset)
  BEGIN
   IF reset = '0' THEN               -- low-aktives Reset
     q <= '0';                       -- hat erste Prioritaet
   ELSIF clk'EVENT AND clk = '1' AND -- Abfrage auf Taktfl.
     clk'LAST_VALUE = '0' THEN       -- nur, wenn kein reset
     q <= d;
   END IF;
  END PROCESS;
END async_reset;
```

2.4.3 Zustandsautomaten

Bei der Modellierung von endlichen Zustandsautomaten (FSM) muß man zwischen Mealy-, Moore- und Medvedev-Automaten unterscheiden. Bei einem Mealy-Automaten hängt der Ausgangsvektor vom momentanen Zustand und vom Eingangsvektor ab, beim Moore-Automaten dagegen nur vom Zustand. Ein Medvedev-Automat ist dadurch gekennzeichnet, daß jeder Ausgang des Automaten mit dem Ausgang eines Zustands-Flip-Flops identisch ist. Abb. C-11 beschreibt ein Blockschaltbild für den Mealy-Automatentyp.

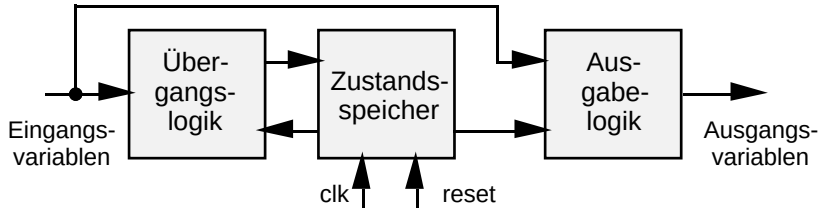

Abb. C-11: Blockschaltbild eines Zustandsautomaten (Mealy)

Die folgenden drei VHDL-Prozesse zeigen die prinzipielle, synthesegerechte Modellierung eines Mealy-Automaten. Die Blöcke aus Abb. C-11 sind hier in getrennten Prozessen realisiert.

```
zustandsspeicher: PROCESS (clk, reset)
BEGIN
  IF (reset = '1') THEN
    zustand <= reset_zustand;
  ELSIF (clk'event AND clk='1' AND clk'LAST_VALUE = '0')
    THEN zustand <= folge_zustand;
  END IF;
END PROCESS zustandsspeicher;
```

```
uebergangslogik: PROCESS (zustand, in1, in2, ...)
BEGIN
  CASE zustand IS
    WHEN zustand1 =>
      IF (in1 = ... AND in2 = ... AND ...)
         THEN folge_zustand <= ...;
      ELSIF ...
      ...
    WHEN zustand2 =>
    ...
  END CASE;
END PROCESS uebergangslogik;
```

```
ausgabelogik: PROCESS (zustand, in1, in2, ...)
BEGIN
  CASE zustand IS
    WHEN zustand1 =>
      IF (in1 = ... AND in2 = ... AND ...)
        THEN out1 <= ...; out2 <= ...; ...
      ELSIF ...
      ...
    WHEN zustand2 =>
      ...
  END CASE;
END PROCESS ausgabelogik;
```

Da die Prozesse zur Beschreibung der Übergangslogik und der Ausgabelogik sehr ähnlich sind, können sie auch zusammengefaßt werden. Eine mögliche Fehlerquelle hierbei ist, daß Latches für die Ausgänge erzeugt werden, wenn diese nicht in jedem Zustand und bei jeder Kombination der Eingangssignale einen Wert zugewiesen bekommen.

Wenn man versucht, die FSM komplett in einem Prozeß zu modellieren, der lediglich vom Takt und dem Rücksetzsignal getriggert wird, werden Flip-Flops für die Ausgänge erzeugt. Als Folgerung daraus kann man empfehlen, bei Mealy- und Moore-Automaten einen Prozeß für die Zustandsspeicherung und einen weiteren Prozeß für den rein kombinatorischen Teil zu verwenden.

2.5 Optimierung der "Constraints"

Neben der Optimierung des VHDL-Modells können zur Verbesserung der Syntheseresultate bei jedem Syntheseprogramm eine Reihe von Optionen angegeben werden, mit denen man die gewünschte Zielsetzung bei der Synthese näher spezifizieren kann (sog. "constraints").

2.5.1 Ziele und Randbedingungen

Grundlegende Ziele bei der Synthese sind neben einer funktionierenden Schaltung:

- geringer Flächenbedarf,
- hohe Geschwindigkeit,
- geringe Verlustleistung.

Daneben lassen sich auch viel detailliertere Angaben machen:

- Festlegung der Treiberstärken an den primären Eingängen,
- Festlegung der Lastkapazitäten an den primären Ausgängen,
- Angabe von Schätzwerten bzw. Modellen zur Berücksichtigung von Leitungskapazitäten,
- Angaben zur Schwankungsbreite der Laufzeiten durch Variation von Temperatur, Versorgungsspannung und Prozeß.

Die angegebenen "constraints" beziehen sich auf die gesamte Schaltung. Meist ist es aber wünschenswert, sie speziell auf einen Teil der Schaltung zu beziehen. Denkbar wäre die Angabe einer maximalen Verzögerungszeit von einem Eingang zu einem Ausgang bei sonst möglichst kleiner Schaltung.

2.5.2 Constraint-Strategien

Es ist i.d.R. nicht sinnvoll, "constraints" für die Synthese intuitiv zu setzen. Die Erfahrung zeigt, daß man bessere Schaltungsergebnisse erhält, wenn man Randbedingungen nicht auf unmögliche Werte setzt (z.B. Laufzeit 0 ns), sondern Werte nahe des Machbaren verwendet. Dies liegt u.a. daran, daß bei unmöglichen Angaben die Gewichtungen bei den heuristischen Methoden der Synthesewerkzeuge ungünstig gesetzt werden.

Durch den Trade-Off Fläche-Geschwindigkeit liegen die optimalen Lösungen einer gegebenen Schaltung, aufgetragen in einem Diagramm Laufzeit über Fläche, auf einer Hyperbel. Da in der Praxis eine endliche Anzahl von nicht immer optimalen Syntheseergebnissen vorliegt, erhält man eine Reihe von diskreten Lösungen im sog. Entwurfsraum.

Abb. C-12 zeigt den Entwurfsraum eines Zustandsautomaten. Die Punkte im Entwurfsraum wurden durch viele Syntheseläufe einer VHDL-Beschreibung unter Verwendung verschiedenster "constraints" generiert. Leider steht ein solcher Überblick über die möglichen Lösungen zu Beginn der Synthese nicht zur Verfügung.

Das Ziel beim Einsatz von "constraints" ist es, dem Syntheseprogramm mitzuteilen, welche Realisierung gewünscht wird. Man kann allerdings kaum vorhersagen, welche "constraints" zu welcher Schaltung führen, so daß ein iteratives Vorgehen notwendig wird.

Man geht dabei sinnvollerweise so vor, daß man mit realen, d.h. leicht verwirklichbaren "constraints" beginnt und diese so lange verschärft, bis das Syntheseprogramm keine bessere Schaltung mehr liefert.

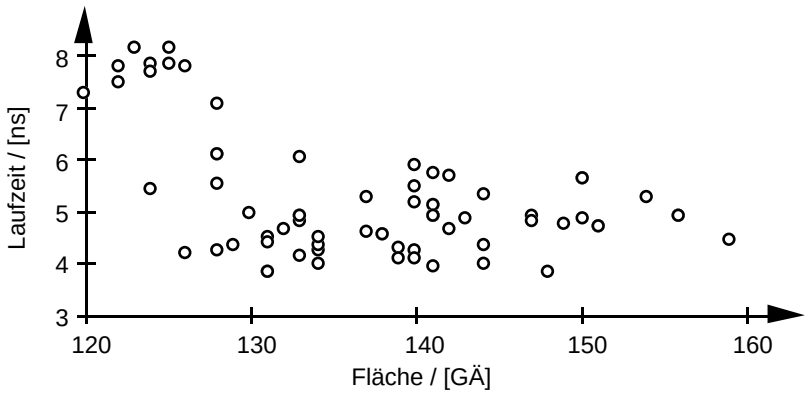

Abb. C-12: Entwurfsraum eines Zustandsautomaten

Am Beispiel des Zustandsautomaten sollen einige mögliche Strategien dargestellt und bewertet werden. Die Ergebnisse haben keine allgemeine Gültigkeit, sollen jedoch die grundsätzliche Problematik aufzeigen.

2.5.2.1 Laufzeit Null

Eine einfache Strategie besteht darin, die gewünschte Laufzeit der Schaltung, ohne Abschätzung der tatsächlich machbaren Laufzeit, auf "Null" zu setzen. Abb. C-13 zeigt das Ergebnis dieser Strategie im Entwurfsraum des erwähnten Beispiels (schwarz ausgefüllter Kreis).

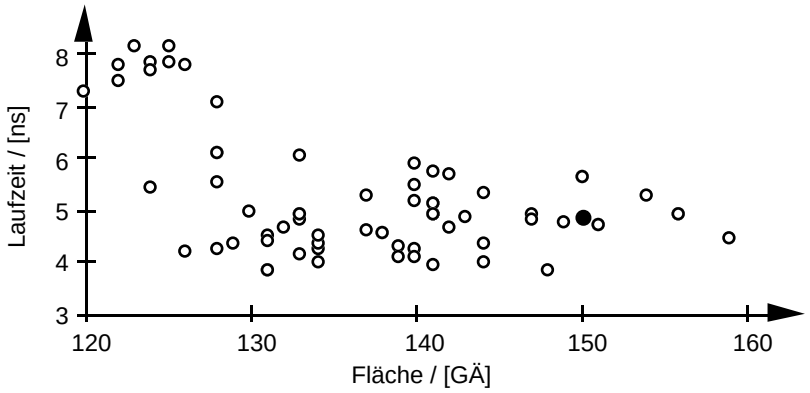

Abb. C-13: Ergebnis der Strategie "Laufzeit Null"

Wie aus der Abbildung ersichtlich ist, führt diese Strategie keineswegs zu einem guten Resultat.

2.5.2.2 Laufzeit "in der Nähe des Machbaren"

Eine andere Strategie besteht darin, ein Laufzeit-Constraint "in der Nähe des Machbaren" zu setzen. Hintergrund ist, daß bei der Verfolgung dieser Strategie die Gewichtungsfaktoren bei der Optimierung eine bessere Wirkung zeigen als bei der zu starken Gewichtung der Null-Laufzeit. Diese Strategie wird auch von vielen Syntheseprogrammherstellern empfohlen.

Für den vorliegenden Fall wurde das Laufzeit-Constraint einmal auf 3 ns und einmal auf 4 ns gesetzt. Die Ergebnisse sind als schwarz ausgefüllte Kreise in Abb. C-14 aufgetragen.

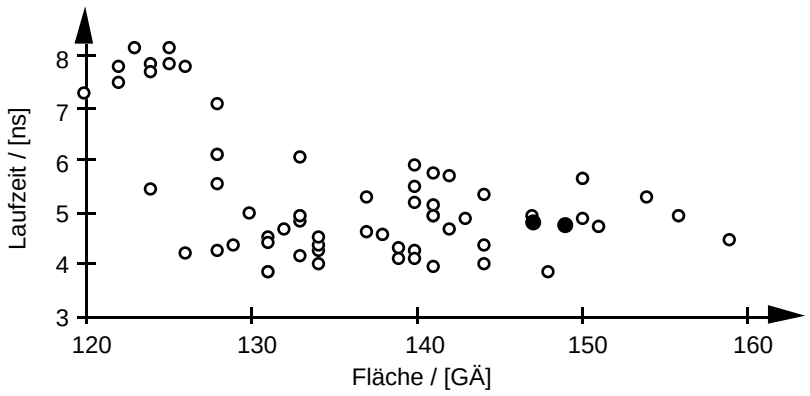

Abb. C-14: Ergebnisse der Strategie "machbare Laufzeit"

Auch damit werden nicht die schnellsten Schaltungen generiert. Vielmehr erhält man ähnliche Ergebnisse wie bei der ersten Strategie.

2.5.2.3 Mehrfache Optimierung

Eine weitere Strategie besteht darin, das Ergebnis obiger Strategien durch mehrere aufeinanderfolgende Syntheseläufe zu optimieren.

Eine neunfache Optimierung des Zustandsautomaten, auf der Basis der Laufzeit 3 ns, führt auf die in Abb. C-15 dargestellten Ergebnisse innerhalb des Entwurfsraums (schwarz ausgefüllte Kreise).

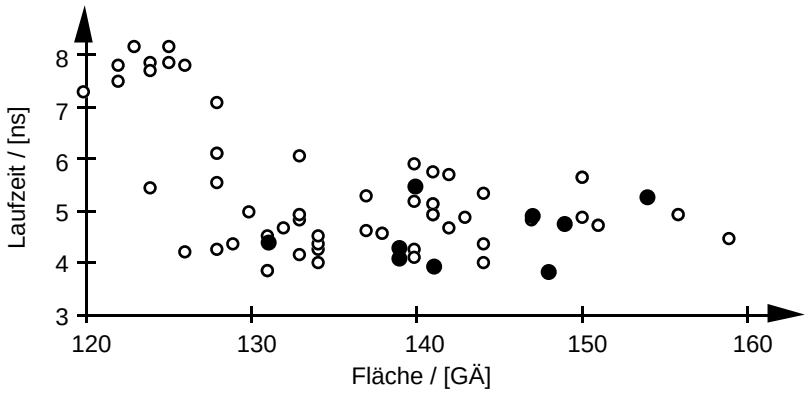

Abb. C-15: Ergebnisse der Strategie "mehrfache Optimierung"

Mit dieser Strategie erhält man eine der schnellsten Schaltungen im Feld mit ca. 4 ns Laufzeit. Aufgrund der mehrfachen Optimierung ergeben sich jedoch deutlich höhere Rechenzeiten. Außerdem läßt sich nicht vorhersagen, nach welcher Iteration das beste Ergebnis, d.h. ein globales Minimum, erreicht wird.

Abb. C-16 zeigt die nach jedem Iterationsschritt erreichte Laufzeit:

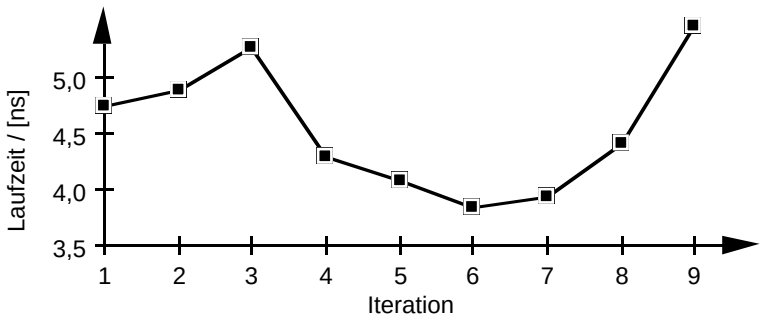

Abb. C-16: Laufzeit bei mehrfacher Zeitoptimierung

An den Ergebnissen des Beispiels läßt sich erkennen, daß eine wiederholte Optimierung nicht immer den gewünschten Erfolg mit sich bringt. Eine mehrfache Optimierung ist jedoch trotz erhöhter Rechenzeiten dem blinden Vertrauen auf ein gutes Ergebnis nach nur einem Syntheselauf vorzuziehen.

2.6 Ressourcenbedarf bei der Synthese

Neben den Performancedaten der von einem Syntheseprogramm erzeugten Schaltung spielt auch der Zeitbedarf für die Synthese eine wichtige Rolle. Kann bei einzelnen Syntheseläufen eine hohe Rechenzeit vielleicht noch akzeptiert werden, so wird bei größeren Entwürfen und mehreren Synthesedurchläufen der Faktor Rechenzeit unmittelbar bestimmend für die gesamte Entwurfszeit. Daneben spielen auch Anforderungen an die benötigte Hardwareplattform, der erforderliche Arbeits- und Swap-Speicher und der vom Programm benötigte Speicherplatz eine Rolle.

Verfügbare Syntheseprogramme bieten leider keine Möglichkeit, die Rechenzeit abzuschätzen oder gar ein Limit dafür zu setzen. Es ist lediglich möglich, die Anzahl der Optimierungszyklen durch Optionen auf niedrig, mittel oder hoch einzustellen. Der Anwender sollte also vor dem Start des Syntheseprozesses eine Vorstellung von der für die Synthese benötigten Rechenzeit haben. Die wichtigsten Einflußfaktoren dafür sind:

- Leistungsfähigkeit der Rechnerumgebung,
- Art des Syntheseprogramms,
- Art der VHDL-Beschreibung,
- gesetzte "constraints",
- verwendete Technologiebibliothek.

Die folgenden beiden Abbildungen zeigen die benötigte Rechenzeit (reine CPU-Zeit) für drei verschiedene Schaltungen, abhängig von der Größe der Schaltung, einmal für Flächenoptimierung und einmal für Laufzeitoptimierung. Zur Synthese wurde ein Rechner vom Typ SUN Sparc IPX mit 32 MB Arbeitsspeicher verwendet.

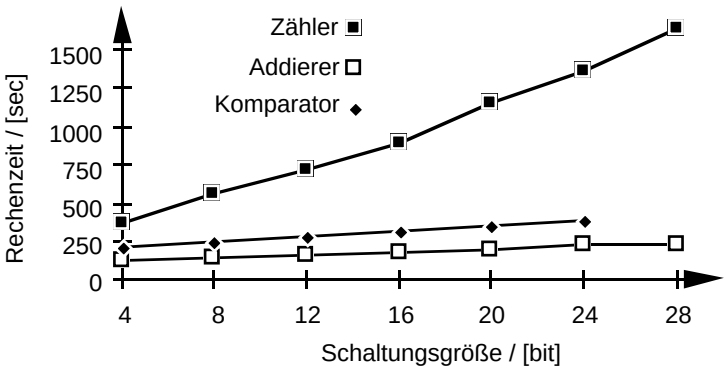

Abb. C-17: CPU-Zeit bei Flächenoptimierung

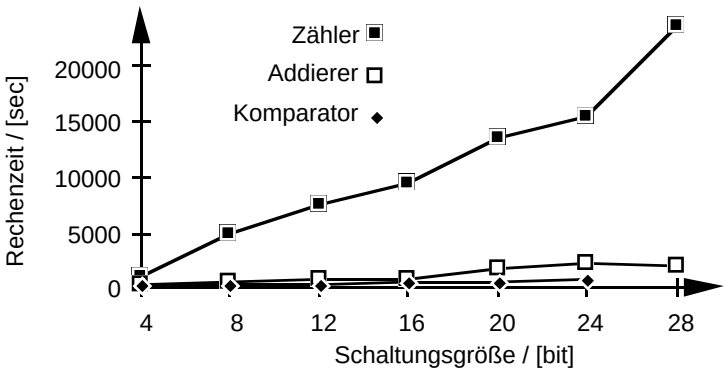

Abb. C-18: CPU-Zeit bei Laufzeitoptimierung

Die beiden Abbildungen zeigen deutlich, daß die Laufzeitoptimierung sehr viel mehr CPU-Zeit erfordert als die Optimierung auf geringste Fläche. Bei den dargestellten Schaltungen, die sich im Bereich von einigen hundert Gatteräquivalenten bewegen, ist ein Unterschied bis etwa zum Faktor 10 festzustellen, während bei größeren Schaltungen (mehrere tausend Gatteräquivalente) Unterschiede bis zum Faktor 100 auftreten.

Außerdem zeigen die letzten beiden Abbildungen, daß die Rechenzeiten für die Addierer- und Komparatorschaltung weit weniger von der Schaltungsgröße abhängig sind als die der Zählerschaltung. Ursache hierfür ist, daß das Synthesewerkzeug bei Addierern und Komparatoren auf programminterne Makros zurückgreifen kann.

Teil D Anhang

1 Packages

1.1 Das Package standard

Das Package `standard` liegt meist nicht in Textform vor, sondern ist im Simulations- bzw. Syntheseprogramm integriert. Es werden in der Version ✓87 folgende Basistypen deklariert:

```
TYPE boolean        IS (false, true);
TYPE bit            IS ('0', '1');
TYPE character      IS ( ... );       -- 128 ASCII-Character
TYPE severity_level IS (note, warning, error, failure);
TYPE integer        IS RANGE ... ;    -- rechnerabhängig
TYPE real           IS RANGE ... ;    -- rechnerabhängig
SUBTYPE natural     IS integer RANGE 0 TO integer'HIGH;
SUBTYPE positive    IS integer RANGE 1 TO integer'HIGH;
TYPE time           IS RANGE ...      -- rechnerabhängig
  UNITS fs;
        ps  =  1000 fs;
        ns  =  1000 ps;
        us  =  1000 ns;
        ms  =  1000 us;
        sec =  1000 ms;
        min =    60 sec;
        hr  =    60 min;
  END UNITS;
TYPE string     IS ARRAY (positive RANGE <>) OF character;
TYPE bit_vector IS ARRAY (natural RANGE <>) OF bit;
```

Weiterhin werden folgende Operatoren deklariert:

- logische Operatoren für Operanden vom Typ `bit`, `bit_vector` und `boolean`,
- die Vergleichsoperatoren für alle deklarierten Typen,
- sämtliche mathematische Operatoren für die Typen `integer` und `real`,

- Multiplikations- und Divisionsoperator für einen `time`-Operand und einen `integer`- oder `real`-Operand,
- der "concatenation"-Operator (&) für Operanden vom Typ `character` und `string` bzw. Kombinationen von beiden, für `bit` und `bit_vector` bzw. Kombinationen von beiden und
- die Funktion `now`, die die aktuelle Simulationszeit liefert.

1.2 Das Package textio

Das Package `textio` stellt einige einfache Funktionen zum Lesen und Schreiben von Informationen in Dateien vom Typ `text` bereit. Die Vorgehensweise bei der Kommunikation mit solchen Dateien wurde bereits in Teil B behandelt. Das Package enthält in der Version ✓**87** folgende Typen:

```
TYPE line          IS ACCESS string;
TYPE text          IS FILE OF string;
TYPE side          IS (right, left);
SUBTYPE width      IS natural;
FILE input  : text IS IN  "STD_INPUT";
FILE output : text IS OUT "STD_OUTPUT";
```

Um zeilenweise aus einem File zu lesen bzw. in ein File zu schreiben und um das Ende von Files zu erkennen sind folgende Funktionen und Prozeduren implementiert:

```
PROCEDURE readline  (f : IN  text; l : OUT line);
PROCEDURE writeline (f : OUT text; l : IN  line);
FUNCTION  endfile   (f : IN  text) RETURN boolean;
```

Prozeduren zum Lesen aus den Zeilen werden für alle Basistypen aus dem Package `standard` (`bit`, `bit_vector`, `boolean`, `integer`, `real`, `character`, `string`, `time`) jeweils mit und ohne

Prüfwert `good` deklariert. Zum Erkennen des Zeilenendes existiert wiederum eine Funktion (`endline`):

```
PROCEDURE read (l : INOUT line; value : OUT bit);
PROCEDURE read (l : INOUT line; value : OUT bit;
            good : OUT boolean);
...
...  -- read-Funktionen fuer andere Typen
...
FUNCTION  endline   (l : IN  line) RETURN boolean;
```

Auch Prozeduren zum Schreiben in die Zeile sind für alle Basistypen aus `standard` vorgesehen. Dabei können optional die zu schreibenden Daten innerhalb eines Bereiches (Bereichslänge: `field`) über den Parameter `justified` ausgerichtet werden:

```
PROCEDURE write (l : INOUT line; value : IN bit;
         justified : IN side:=right; field : IN width:=0);
...
...  -- write-Funktionen fuer andere Typen
...
```

Besonderheiten ergeben sich beim Schreiben von Objekten der Typen `real` und `time`.

```
PROCEDURE write (l : INOUT line; value : IN real;
           justified : IN side:=right; field : IN width:=0;
           digits : IN natural := 0);
PROCEDURE write (l : INOUT line; value : IN time;
           justified : IN side:=right; field : IN width:=0;
           unit : IN time := ns);
```

Bei reellen Zahlen kann die Zahl der Nachkommastellen (`digits`) angegeben werden. Der Defaultwert 0 bedeutet, daß das Objekt in der Exponentialform (z.B. 1.987456E-17) geschrieben wird.

Bei Objekten des Typs `time` wird mit der optionalen Angabe von `units` die Einheit festgelegt, in der der Objektwert gespeichert wird. Defaultwert hierfür ist ns (Nanosekunden).

1.3 IEEE-Package 1164

Das 9-wertige Logiksystem im Package `std_logic_1164` wurde vom IEEE entwickelt und normiert, um einen Standard für Logikgatter zu setzen, die genauer modelliert sind, als dies eine zweiwertige Logik zu leisten vermag. Die hier beschriebenen Deklarationen und Funktionen beziehen sich auf die Version 4.200 des Packages.

Der Basistyp des Logiksystems `std_ulogic` (`u` steht für "unresolved"), ist als Aufzähltyp der folgenden Signalwerte deklariert:

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

Die unterschiedlichen Signalwerte haben folgende Bedeutung:

- Starke Signalwerte (`'0'`, `'1'`, `'X'`) beschreiben eine Technologie, die aktiv die Pegel ´High´ und ´Low´ treibt (Totem-Pole-Endstufen, CMOS).
- Schwache Signale (`'L'`, `'H'`, `'W'`) dienen für Technologien mit schwach treibenden Ausgangsstufen (z.B. NMOS-Logik mit Widerständen als Lastelemente).
- Mit dem Wert `'Z'` können Tristate-Ausgänge beschrieben werden.
- Der Wert `'U'` kennzeichnet nichtinitialisierte Signale.
- Der Wert `'-'` dient zur Kennzeichnung von "don´t cares", die bei der Logikoptimierung verwendet werden.

Dieser Logiktyp, davon abgeleitete Untertypen, zugehörige Konvertierungsfunktionen, Operatoren und "resolution functions" werden im Package `std_logic_1164` deklariert und die Funktionen im zugehörenden Package Body definiert.

Folgende abgeleitete Typen des Basistyps `std_ulogic` sind im Package deklariert:

```
TYPE std_ulogic_vector IS ARRAY
                      ( natural RANGE <> ) OF std_ulogic;
FUNCTION resolved ( s : std_ulogic_vector )
                        RETURN std_ulogic;
SUBTYPE std_logic IS resolved std_ulogic;
TYPE std_logic_vector  IS ARRAY
                      ( natural RANGE <> ) OF std_logic;
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';
```

Die Untertypen `X01`, `X01Z`, `UX01` und `UX01Z` bilden mehrwertige Logiksysteme, die auf die schwach treibenden Signalwerte und das "don´t care" verzichten.

Für den Basistyp `std_ulogic` (und damit implizit auch für dessen Untertypen `std_logic`, `X01`, `X01Z`, `UX01` und `UX01Z`), die Vektortypen `std_ulogic_vector` und `std_logic_vector` sind die folgenden, überladenen Operatoren definiert:

- `"NOT"`
- `"AND"`
- `"NAND"`
- `"OR"`
- `"NOR"`
- `"XOR"`

Am Beispiel des Operators `"NAND"` sollen die existierenden Varianten für die diversen Operandentypen gezeigt werden:

```
FUNCTION "NAND" ( l : std_ulogic; r : std_ulogic )
                        RETURN UX01;
FUNCTION "NAND" ( l, r : std_logic_vector  )
                        RETURN std_logic_vector;
FUNCTION "NAND" ( l, r : std_ulogic_vector )
                        RETURN std_ulogic_vector;
```

Die Funktion `xnor` wird in der hier beschriebenen Package-Version für die 9-wertige Logik ebenfalls definiert, allerdings nicht als Operator, sondern als herkömmliche Funktion:

```
FUNCTION xnor   ( l : std_ulogic; r : std_ulogic )
                        RETURN UX01;
FUNCTION xnor   ( l, r : std_logic_vector  )
                        RETURN std_logic_vector;
FUNCTION xnor   ( l, r : std_ulogic_vector )
                        RETURN std_ulogic_vector;
```

Zwischen den Typen des Packages und den herkömmlichen VHDL-Typen wurden folgende Konvertierungsfunktionen erforderlich:

- `To_bit` für Operanden vom Typ `std_ulogic`,
- `To_bitvector` für Operanden vom Typ `std_logic_vector` und `std_ulogic_vector`,
- `To_StdULogic` für Operanden vom Typ `bit`,
- `To_StdLogicVector` für Operanden vom Typ `bit_vector` und `std_ulogic_vector`,
- `To_StdULogicVector` für Operanden vom Typ `bit_vector` und `std_logic_vector`,
- `To_X01` für Operanden vom Typ `bit`, `bit_vector`, `std_ulogic`, `std_ulogic_vector`, `std_logic_vector`,

- `To_X01Z` für Operanden vom Typ `bit`, `bit_vector`, `std_ulogic`, `std_ulogic_vector`, `std_logic_vector`,
- `To_UX01` für Operanden vom Typ `bit`, `bit_vector`, `std_ulogic`, `std_ulogic_vector`, `std_logic_vector`.

Weitere Funktionen des Packages prüfen auf steigende oder fallende Flanken eines Signals und auf den Wert `'X'`:

```
FUNCTION rising_edge  (SIGNAL s : std_ulogic)
                       RETURN boolean;
FUNCTION falling_edge (SIGNAL s : std_ulogic)
                       RETURN boolean;

FUNCTION Is_X ( s : std_ulogic_vector ) RETURN  boolean;
FUNCTION Is_X ( s : std_logic_vector  ) RETURN  boolean;
FUNCTION Is_X ( s : std_ulogic        ) RETURN  boolean;
```

Eine vollständige Beschreibung der Funktionalität aller im Package enthaltenen Unterprogramme würde hier zu weit führen. Dennoch soll anhand einiger Beispiele das Vorgehen gezeigt werden.

Viele Funktionen werden über Tabellen realisiert, die mit dem Aufzähltyp `std_ulogic` indiziert sind. Das Ergebnis der entsprechenden Funktion wird dann nur noch durch Ansprechen eines Tabellenelementes mit den beiden Operanden als Index (= Adresse) erzeugt.

Anhand der Invertierungsfunktion und der "resolution function" soll dies beispielhaft gezeigt werden. Zunächst wird ein Vektor- und ein Matrixtyp deklariert, die mit `std_ulogic` indiziert sind und deshalb 9 bzw. (9 x 9) Elemente des Typs `std_ulogic` enthält:

```
TYPE stdlogic_1d    IS ARRAY (std_ulogic) OF std_ulogic;
TYPE stdlogic_table IS ARRAY(std_ulogic, std_ulogic)
                    OF std_ulogic;
```

Die Auflösungsfunktion wird folgendermaßen realisiert:

```
CONSTANT resolution_table : stdlogic_table := (
-- ---------------------------------------------------------
-- |  U    X    0    1    Z    W    L    H    -        |   |
-- ---------------------------------------------------------
   ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
   ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
   ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
   ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
   ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
   ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
   ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
   ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
   ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ));-- | - |

FUNCTION resolved ( s : std_ulogic_vector )
                    RETURN std_ulogic IS
  VARIABLE result : std_ulogic := 'Z';
                    -- weakest state default
BEGIN
  IF    (s'LENGTH = 1) THEN
    RETURN s(s'LOW);
  ELSE
    FOR i IN s'RANGE LOOP
      result := resolution_table(result, s(i));
    END LOOP;
  END IF;
  RETURN result;
END resolved;
```

Die Auflösungsfunktion arbeitet beliebig lange Eingangsvektoren ab, deren Elemente die Werte der einzelnen Signaltreiber für das aufzulösende Signal darstellen. Man geht hierbei vom schwächsten Zustand ("high impedance", `'Z'`, als Defaultwert des Ergebnisses) aus. Es wird dann sukzessive der (vorläufige) Ergebniswert mit dem nächsten Element des Eingangsvektors verknüpft, bis alle Elemente abgearbeitet sind.

Ein weiteres Beispiel zeigt die Realisierung der Invertierungsfunktion:

```
CONSTANT not_table: stdlogic_1d :=
--  ------------------------------------------------
--  |   U    X    0    1    Z    W    L    H    -   |
--  ------------------------------------------------
     ( 'U', 'X', '1', '0', 'X', 'X', '1', '0', 'X' );

FUNCTION "NOT"  ( l : std_ulogic ) RETURN UX01 IS
BEGIN
   RETURN (not_table(l));
END "NOT";
```

Die vektoriellen Funktionen führen die Operatoren bitweise für jede einzelne Stelle im Vektor durch. Als Beispiel sei hier der `NAND`-Operator für zwei `std_ulogic`-Vektoren aufgeführt:

```
FUNCTION "NAND"  ( l,r : std_ulogic_vector )
                   RETURN std_ulogic_vector IS
  ALIAS lv : std_ulogic_vector ( 1 TO l'LENGTH ) IS l;
  ALIAS rv : std_ulogic_vector ( 1 TO r'LENGTH ) IS r;
  VARIABLE result : std_ulogic_vector ( 1 TO l'LENGTH );
BEGIN
  IF ( l'LENGTH /= r'LENGTH ) THEN
    ASSERT false
        REPORT "arguments of overloaded 'NAND'-operator "
             & "are not of the same length"
        SEVERITY failure;
  ELSE
    FOR i IN result'RANGE LOOP
      result(i) := not_table(and_table (lv(i), rv(i)));
    END LOOP;
  END IF;
  RETURN result;
END "NAND";
```

Mit entsprechenden Tabellen werden nach dem gleichen Schema die zahlreichen Konvertierungsfunktionen realisiert.

Als Beispiel einer Funktion, die ein Boolesches Ergebnis liefert, sei hier die Erkennung von undefinierten Werten in einem `std_ulogic` Objekt dargestellt:

```
FUNCTION Is_X ( s : std_ulogic ) RETURN boolean IS
BEGIN
  CASE s IS
    WHEN 'U' | 'X' | 'Z' | 'W' | '-' => RETURN true;
    WHEN OTHERS => NULL;
  END CASE;
  RETURN false;
END;
```

2 VHDL-Übungsbeispiele

Die im folgenden aufgeführten Übungsaufgaben sind in zwei Kategorien eingeteilt: Übungen zu grundlegenden VHDL-Konstrukten und etwas komplexere, komplette Entwurfsbeispiele. Falls nicht anders vermerkt, sollen Signale vom Typ `bit` verwendet werden.

Beispiele zur Lösung der Übungsaufgaben finden sich ausschließlich auf der beiliegenden Diskette. Das Verzeichnis `UEBUNGEN` verzweigt in die vier Unterverzeichnisse `BASICS`, `BCD`, `ALU` und `TAB`. Dort finden sich entsprechend bezeichnete ASCII-Files mit Lösungen zu den einzelnen Übungsaufgaben.

2.1 Grundlegende VHDL-Konstrukte

2.1.1 Typen

- Deklarieren Sie einen Aufzähltyp namens `wochentag`, der die Werte `montag, dienstag, ... sonntag` annehmen kann.
- Deklarieren Sie einen zusammengesetzten Typ (`RECORD`) namens `datum`, der folgende Elemente enthalten soll:
  - `jahr` (`integer` 0-3000)
  - `monat` (`integer` 1-12)
  - `tag` (`integer` 1-31)
  - `wochentag` (siehe oben)

2.1.2 Strukturale Modellierung

- Beschreiben Sie einen Halbaddierer, der aus einem AND2- und einem XOR2-Gatter aufgebaut werden soll.

  Schreiben Sie zunächst eine Entity mit den Eingangssignalen `sum_a` und `sum_b` sowie den Ausgangssignalen `sum` und `carry`.

  Erstellen Sie dazu eine strukturale Architektur.

- Beschreiben Sie einen 1-Bit Volladdierer
  mit den Eingängen: `in_1`, `in_2`, `in_carry`,
  und den Ausgängen: `sum`, `carry.`

  Das strukturale Modell soll aus zwei Halbaddierern und einem OR2-Gatter aufgebaut werden.

- Erstellen Sie ein strukturales Modell für einen 8-Bit Ripple-Carry Addierer
  mit den Eingängen: `in_1(7 DOWNTO 0)`, `in_2(7 DOWNTO 0)`, `in_carry`,
  und den Ausgängen: `sum(7 DOWNTO 0)`, `carry`.

  Verwenden Sie die `GENERATE`-Anweisung und Volladdierer.

- Konfigurieren Sie das hierarchische Modell des 8-Bit-Addierers. Als Basisgatter stehen folgende VHDL-Modelle zur Verfügung:
  - `or2 (behavioral)`
    Eingänge: `a` und `b`, Ausgang: `y`
  - `and2 (behavioral)`
    Eingänge: `a` und `b`, Ausgang: `y`
  - `exor (behavioral)`
    Eingänge: `sig_a` und `sig_b`, Ausgang: `sig_y`.

2.1.3 Verhaltensmodellierung

- Beschreiben Sie ein symmetrisches Taktsignal `clk`, das eine Frequenz von 10 MHz besitzen soll. Verwenden Sie dazu ein Signal vom Typ `bit` und alternativ ein Signal vom Typ `std_ulogic`.

- Gegeben sei folgende Architektur eines 4:1-Multiplexers:

```
ARCHITECTURE behavioral_1 OF mux IS
BEGIN
  sig_out <= sig_a WHEN sel = "00" ELSE
             sig_b WHEN sel = "01" ELSE
             sig_c WHEN sel = "10" ELSE
             sig_d ;
END behavioral_1 ;
```

  Beschreiben Sie diese Funktionalität mit anderen nebenläufigen oder sequentiellen Anweisungen.

- Folgende Funktion beschreibt die elementweise Multiplikation zweier Integer-Vektoren (eigendefinierter Typ `int_vector`) mit einer `WHILE`-Schleife:

```
FUNCTION mult_while (a,b : int_vector(1 TO 8))
         RETURN int_vector IS
  VARIABLE count  : integer ;
  VARIABLE result : int_vector(1 TO 8) ;
BEGIN
  count := 0 ;
  WHILE count < 8 LOOP
    count := count + 1 ;
    result(count) := a(count)*b(count) ;
  END LOOP ;
  RETURN result ;
END mult_while ;
```

  Beschreiben Sie diese Funktion alternativ mit einer `FOR`-Schleife.

  Beschreiben Sie diese Funktion mit einer Endlos-Schleife und einer `EXIT`-Anweisung.

  Verändern Sie obige Funktion derart, daß sie allgemeine, gleichlange Integervektoren unbestimmter Länge und unterschiedlicher Indizierung verarbeiten kann.

❒ Gegeben seien folgende Anweisungen innerhalb einer Architektur:

```
ARCHITECTURE assignment OF example IS
  SIGNAL a,b : integer := 0;
BEGIN
  a <= 1 AFTER 2 ms, 4 AFTER 5 ms;
  b <= 4 AFTER 3 ms, 3 AFTER 4 ms;
  p : PROCESS (a,b)
    VARIABLE c, d : integer := 0;
  BEGIN
      c := a + b;
      d := c + 2;
  END PROCESS p ;
END assignment ;
```

Welchen Werteverlauf besitzt die Variable d für diesen Fall?

Welchen Verlauf besitzt d, falls es als Signal deklariert wird?

Welchen Verlauf besitzt d, falls c ein Signal ist?

Welchen Verlauf besitzt d, falls c und d Signale sind?

Welchen Verlauf besitzt d, falls c und d Signale sind und c in der "sensitivity-list" des Prozesses p enthalten ist?

2.1.4 Testumgebungen

❒ Schreiben Sie für obigen 1-Bit-Volladdierer eine Testumgebung, die neben der Stimulierzeugung auch die Ergebnisauswertung enthält. Verwenden Sie zur Zuweisung der erforderlichen Stimuli (drei Signale) eine möglichst kompakte Anweisung.

2.2 Komplexe Modelle

2.2.1 Vierstellige Siebensegment-Anzeige

Es soll ein VHDL-Modell für eine Schaltung `bcd_4bit` (Abb. D-1) entwickelt werden, die eine vierstellige BCD-Zahl über eine vierstellige Siebensegment-Anzeige ausgibt. Da nur ein BCD / Siebensegment-Codierer (Modul `bcd_7seg`) verwendet werden soll, müssen die vier, jeweils 4-Bit breiten Eingangssignale im Zeitmultiplex auf den Co-

dierer geschaltet werden. Ein Demultiplexer (Modul `dx`) aktiviert über die Signale `stelle1` bis `stelle4` immer nur eines der vier Anzeigeelemente. Der Multiplexer (Modul `mux`) und der Demultiplexer werden über einen zyklischen Dualzähler (Modul `ctr`) angesteuert. Die Ausgangssignale des Gesamtmodells sind die Bitsignale `a` bis `g` und `stelle1` bis `stelle4`. Als Eingangssignale besitzt das Modell das Bitsignal `takt` und ein komplexes Signal `bcd4`, das die vier BCD-Zahlen über insgesamt 16 Bitsignale überträgt.

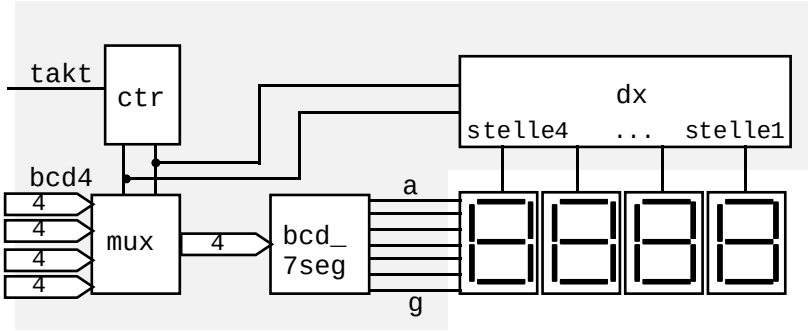

Abb. D-1: Blockschaltbild der Anzeigesteuerung bcd_4bit

- Entwickeln Sie ein VHDL-Modell für den synchronen Dualzähler `ctr`, der zyklisch die Zahlen 0 bis 3 binär codiert über zwei Ausgangsports ausgibt. Der Zählerstand soll bei einer positiven Taktflanke erhöht werden.
- Schreiben Sie ein VHDL-Modell für den 1-aus-4-Demultiplexer. Er soll low-aktive Ausgänge besitzen, die als gemeinsame Kathode der Siebensegmentanzeigen dienen: eine `'0'` am Ausgang bedeutet, daß das entsprechende Siebensegment-Modul aktiviert ist.
- Legen Sie ein VHDL-Modell für den 4-zu-1-Multiplexer an. Beachten Sie, daß hier ein 4-Bit breites Signal durchgeschaltet wird.
- Entwickeln Sie ein VHDL-Modell, das eine BCD-Zahl (Zahlen 0 bis 9 binär codiert) in Steuersignale für die sieben Segmente der BCD-Anzeige umwandelt. Eine `'1'` heißt, daß das zugeordnete

Segment leuchten soll. Falls keine gültige BCD-Zahl anliegt (Zahlen 10 bis 15), soll die Anzeige dunkel bleiben.

❒ Schalten Sie die Module entsprechend der Abbildung in einem strukturalen VHDL-Modell zusammen und testen Sie die Funktion der Gesamtschaltung mit Hilfe einer Testbench. Verwenden Sie Assertions zur Überprüfung der Ausgangssignale.

2.2.2 Arithmetisch-logische Einheit

Es soll das Modell einer arithmetisch-logischen Einheit (ALU) erstellt werden, das einen eingeschränkten Befehlssatz bearbeiten kann. Als Operanden (`a` und `b`) sollen zwei Bit-Vektoren (jeweils 8 Bit) mit einem zusätzlichen Paritätsbit angelegt werden und als Ergebnis ein 16-Bit breiter Vektor, ebenfalls mit Paritätsbit, erzeugt werden. Folgende Operatoren sind zu modellieren:

`con_ab:`	Zusammenfügen der beiden Operanden (`a & b`),
`add_ab:`	Addieren der beiden Operanden,
`add_of:`	Addieren eines konstanten Offsets zu `b`,
`mlt_ab:`	Multiplikation beider Operanden,
`sh_l_b:`	Links-Shift des Operanden `b` um eine Stelle,
`sh_r_b:`	Rechts-Shift des Operanden `b` um eine Stelle,
`sh_l_k:`	Links-Shift des Vektors `a & b` um eine Stelle,
`sh_r_k:`	Rechts-Shift des Vektors `a & b` um eine Stelle,
`flip_b:`	Vertauschen der niederwertigen und höherwertigen vier Bits des Operanden `b`,
`flip_k:`	Vertauschen der vier niederwertigen mit den vier höherwertigen Bits, sowie der jeweils vier mittleren Bits des Vektors `a & b`,
`turn_b:`	Drehen des Operanden `b` (Indexinvertierung),
`turn_k:`	Drehen des Vektors `a & b` (Indexinvertierung).

Bei Operationen mit nur einem Eingangsvektor sollen die niederwertigen Bits des Ergebnisvektors belegt werden, die höherwertigen zu Null gesetzt werden.

Erstellen eines Package

Zunächst soll ein Package `alu_pack` mit entsprechendem Package Body verfaßt werden, das folgende Funktionen und Deklarationen enthält:

- Deklaration eines Typs `command` als Aufzähltyp der oben aufgeführten Befehlsalternativen,
- Deklaration eines Record-Typs `parity_8`, bestehend aus 8-Bit breitem Vektor plus Paritätsbit,
- Deklaration eines Record-Typs `parity_16`, bestehend aus 16-Bit breitem Vektor plus Paritätsbit,
- Definition des konstanten Offsets als "deferred constant" (`ram_offset`),
- Prozedur (passiv) zur Prüfung des Paritätsbits, welche eine Fehlermeldung liefert, falls eine ungerade Anzahl von Eins-Stellen vorliegt und das Paritätsbit = `'0'` ist (und umgekehrt),
- Funktion zur Erzeugung des Paritätsbits für den Ergebnisvektor (`'1'` für eine ungerade Zahl von Eins-Stellen),
- Funktion zur Umwandlung eines 8-Bit breiten Bitvektors in eine Integerzahl zwischen 0 und 255,
- Funktion zur Umwandlung einer Integerzahl (0 bis 65535) in einen 16-Bit breiten Bitvektor.

Schnittstellenbeschreibung

Mit obigen Deklarationen kann nun die Schnittstelle (Entity) der ALU beschrieben werden, die den Aufruf der passiven Prozedur zur Paritätsprüfung im Anweisungsteil enthält.

Architekturbeschreibung

Die Architektur soll die Abarbeitung der einzelnen Kommandos und die Erzeugung des Paritätsbits enthalten. Die arithmetischen Operationen (`add_ab`, `add_of` und `mlt_ab`) können im Integerbereich mit Hilfe der Konvertierungsfunktionen durchgeführt werden.

Erstellen einer Testbench

In Anlehnung an die Programmierung eines Prozessors soll die Testbench die Kommandos (Operatoren) und Operanden aus einem Textfile einlesen, an die ALU weitergeben und das Ergebnis in ein

zweites File schreiben. Jede Zeile im Eingabefile enthält den Operator und die beiden Operanden, jeweils durch ein Leerzeichen getrennt.

Für diese Vorgehensweise sind einige Erweiterungen des Package `alu_pack` erforderlich:

- Definition des Eingabefilenamens als "deferred constant" (`input_filename`),
- Definition des Ausgabefilenamens als "deferred constant" (`output_filename`),
- Da man mit den Funktionen aus dem Package `textio` keine eigendefinierten Typen lesen kann, müssen die Operatoren als Zeichenkette abgelegt werden und innerhalb der Testbench in den Typ `command` umgewandelt werden. Dazu ist eine entsprechende Funktion zu schreiben und im Package abzulegen.

Schreiben Sie daraufhin eine Testbench (Entity und Architecture), die folgendes Ablaufschema durchläuft:

① Lese eine Zeile aus dem Eingabefile, solange das Fileende nicht erreicht ist.

② Lese Operator und Operanden aus dieser Zeile und lege sie an die ALU (model under test) an.

③ Warte bis ein Ergebnis geliefert wird.

④ Schreibe das Ergebnis in das Ergebnisfile.

⑤ Gehe zurück zu ①.

Zum Testen des Modells kann das File "`ALU_COMM`" auf der beiliegenden Diskette als Eingabefile dienen.

2.2.3 Zustandsautomat

Als weiteres Entwurfsbeispiel soll ein endlicher Zustandsautomat (FSM) dienen:

Für einen Telefonanrufbeantworter soll eine einfache Steuerung des Aufnahmebandes und des Ansagebandes aufgebaut werden. Dazu ist folgendes Blockschaltbild zu betrachten:

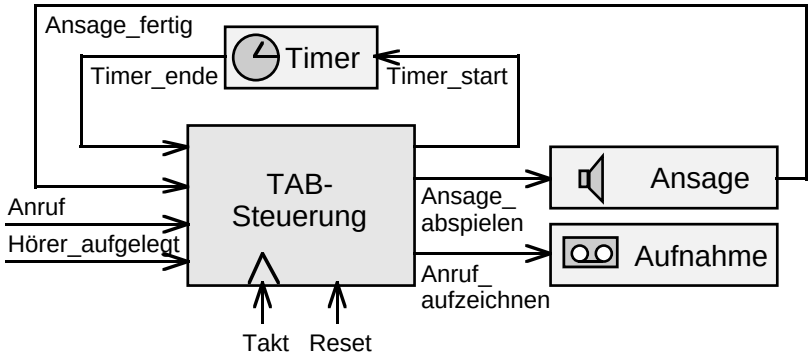

Abb. D-2: Blockschaltbild des Telefonanrufbeantworters

Schnittstellenbeschreibung

Beschreiben Sie die Schnittstelle der Steuerung für den Telefonanrufbeantworter nach obigem Blockschaltbild. Insbesondere ist auch ein Takt- und ein Rücksetzsignal vorzusehen.

Architekturbeschreibung

Die Steuerung soll folgende Funktionalität besitzen:

① Beim ersten Klingelzeichen (Anruf = `'1'`) wird ein Timer gestartet (Timer_start = `'1'`).

② Wurde nach Ablauf der Wartezeit (Timer_ende = `'1'`) der Hörer nicht abgenommen, wird die Ansage abgespielt (Ansage_ abspielen = `'1'`).

③ Wurde nach Ablauf der Nachricht (Ansage_fertig = `'1'`) immer noch nicht abgehoben, beginnt die Aufzeichnung des Anrufes (Anruf_aufzeichnen = `'1'`).

④ Hat der Anrufer aufgelegt (Anruf = `'0'`), so wird die Aufnahme beendet (Anruf_aufzeichnen = `'0'`) und auf den nächsten Anruf gewartet (Ruhezustand).

⑤ Legt der Anrufer vorzeitig auf, wird ebenfalls in den Ruhezustand zurückgekehrt.

⑥ Das Abnehmen des Hörers (Hörer_aufgelegt = `'0'`) am lokalen Apparat hat Vorrang und führt immer in den Zustand des Gegensprechens ("Telefonieren").

Diese Funktionalität kann durch einen Automatengraphen (Moore-Automat, Abb. D-3) mit fünf Zuständen realisiert werden. An den einzelnen Übergängen sind die **Eingangssignale** in der Reihenfolge

Anruf, Hörer_aufgelegt, Timer_ende, Ansage_fertig

angezeichnet, in den Zuständen sind die **Ausgangssignale** vermerkt:

Timer_start, Ansage_abspielen, Anruf_aufzeichnen

Durch die Verwendung von sog. "don´t cares" ("-") zur Kennzeichnung nicht relevanter Eingangsvariablen kann die Beschreibung eines solchen Automatengraphen wesentlich kürzer gestaltet werden:

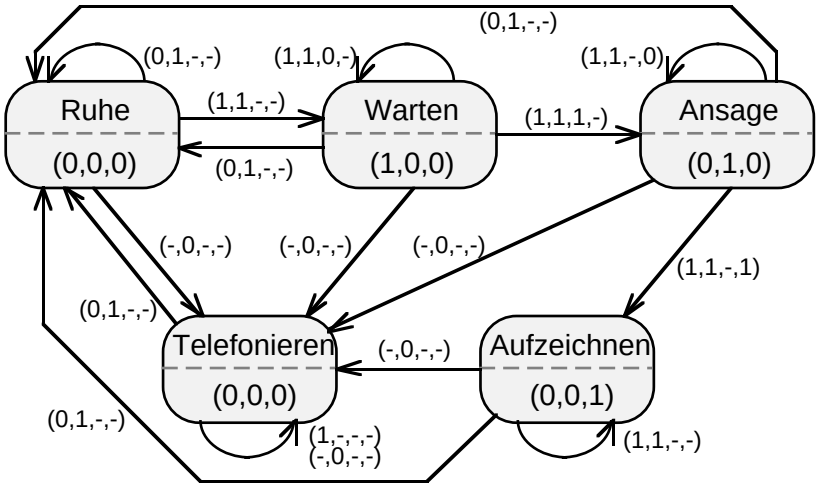

Abb. D-3: Automatengraph des Telefonanrufbeantworters

Erstellen Sie ein Verhaltensmodell für eine entsprechende Steuerung. Definieren Sie sich dazu ein Signal, das einen der fünf Zustände annehmen kann. Das Rücksetzsignal soll asynchron wirken, der Zustand jeweils an der positiven Taktflanke wechseln. Die Ausgangssignale sollen in Abhängigkeit vom Zustand zugewiesen werden.

Testbench

Schreiben Sie eine Testbench zur Überprüfung Ihres Modells. Simulieren Sie dabei nicht nur einen typischen Ablauf, sondern auch vorzeitiges Auflegen des Anrufenden, Beginn eines Gespräches vom lokalen Apparat usw.

3 VHDL-Gremien und Informationsquellen

Auf internationaler Basis arbeiten viele Organisationen und Arbeitsgruppen an der Weiterentwicklung und Verbreitung der Sprache VHDL. Viele davon bieten Informationen für "jedermann", z.B. über regelmäßig erscheinende Newsletters, mailing-Listen oder ftp-Server, die über das internationale Rechnernetz (Internet) frei zugänglich sind. Einigen der Arbeitsgruppen kann man beitreten und sich aktiv an deren Tätigkeiten und Diskussionen beteiligen.

Die folgenden Abschnitte sollen einen Überblick über die wichtigsten dieser Institutionen geben. Die Aufzählung erhebt keinen Anspruch auf Vollständigkeit. Es existieren neben den erwähnten Institutionen noch viele weitere, nationale oder lokale Benutzervereinigungen und spezialisierte Arbeitsgruppen.

3.1 VHDL-News-Group

Im internationalen News-Net wurde im Januar 1991 eine Gruppe speziell für das Themengebiet VHDL eingerichtet. Sie trägt die Bezeichnung `comp.lang.vhdl` und wird täglich von mehreren hundert VHDL-Anwendern gelesen.

Aktuelle Themen und Probleme zur Syntax, zu Programmen und zum praktischen Einsatz von VHDL werden dort intensiv diskutiert und neueste Informationen ausgetauscht.

Regelmäßig werden von Herrn Thomas Dettmer aktuelle Informationen über VHDL-Literatur, Public-Domain-Software, häufige Fragen zu VHDL, Anschriften von VHDL-Gremien, Arbeitsgruppen sowie Softwareherstellern zusammengestellt und in dieser News-Group veröffentlicht.

3.2 VHDL International

Unter den Arbeitsgruppen, die sich international mit dem Thema VHDL beschäftigen, sind die meisten in den Vereinigten Staaten beheimatet. Dort ist, neben den verschiedenen Komitees vom IEEE, in erster Linie die Gruppe "VHDL International, Inc." (kurz: VI) zu nennen. Hierbei handelt es sich um eine gemeinnützige Vereinigung, deren Absicht es ist, gemeinsam die Sprache VHDL als weltweiten Standard für den Entwurf und die Beschreibung von elektronischen Systemen voranzutreiben. Zu den Mitgliedern von VI gehören alle führenden EDA-Hersteller nebst einigen bedeutenden Elektronikkonzernen.

Kontakt: VHDL International, 407 Chester Street,
Menlo Park, CA 94025, USA
e-mail: `cms@cis.stanford.edu`

Von VHDL International wird auch eine Arbeitsgruppe gesponsort, das "VHDL International Users Forum" (VIUF, früher VHDL Users Group, VUG). Das VIUF organisiert unter anderem Konferenzen und publiziert Newsletters.

Kontakt: Allen M. Dewey,
IBM Enterprise Systems, P.O. Box 950, MS P360,
Poughkeepsie, NY 12602, USA
e-mail: `adewey@vnet.ibm.com`

3.3 VHDL Forum for CAD in Europe

In Europa treffen sich die VHDL-Experten meist zweimal im Jahr im Rahmen der Konferenz EURO-DAC / EURO-VHDL und bei den Veranstaltungen des "VHDL Forum for CAD in Europe" (VFE). Während auf der großen Herbst-Konferenz mehr die theoretischen Vorträge im Vordergrund stehen, liegt der Schwerpunkt bei den VFE-Tagungen, die im Frühjahr stattfinden, auch auf den praktischen VHDL-Anwendungen. Beide Termine werden von Vorführungen bzw. Präsentationen der EDA-Hersteller begleitet. Im Rahmen des "VHDL Forum for CAD in Europe" sind einige Arbeitsgruppen entstanden. Dazu gehört z.B. die "European VHDL synthesis working group", die sich mit

Problemstellungen aus dem Bereich der VHDL-Synthese auseinandersetzt.

Kontakt: Andreas Hohl,
Siemens AG, Abteilung ZFE IS EA
Otto-Hahn-Ring 6, D-81739 München
e-mail: ah@ztivax.siemens.com

3.4 European CAD Standardization Initiative

Eine weitere Organisation in Europa ist die "European CAD Standardization Initiative" (ECSI). Ihre Aktivitäten umfassen nicht nur VHDL, sondern auch die Standards EDIF[1] und CFI[2]. Die noch relativ junge Organisation wird von deren Mitgliedern und über ein ESPRIT-Projekt durch Mittel der Europäischen Union getragen.

Kontakt: ECSI Office, Parc Equation, 2 Ave de Vignate,
38610 Gières, France

Zu den Aktivitäten gehört u.a. die Einrichtung sog. "Technical Centers", die Expertenunterstützung beim Einsatz der Standards bieten, mailing-Listen und ftp-Server verwalten, Schulungskurse anbieten, Software entwickeln und vieles mehr.

Speziell für VHDL wird von der ECSI das Informationsblatt "VHDL Newsletter" europaweit herausgegeben, in dem über die Arbeit der verschiedenen, internationalen Gremien berichtet wird, Zusammenfassungen von wichtigen Tagungen und Konferenzen wiedergegeben werden und ein Veranstaltungskalender abgedruckt ist.

---

1 EDIF = Electronic Design Interchange Format

2 CFI = CAD Framework Initiative

Herausgeber: Jaques Rouillard und Jean P. Mermet,
Institut Méditerranéen de Technologie,
Technopôle de Château-Gombert
13451 Marseille Cedex, France

3.5 AHDL 1076.1 Working Group

Auf dem Gebiet der analogen Elektronik, die mit zeit- und wertkontinuierlichen Signalen arbeitet, gibt es zur Zeit keine genormte Beschreibungssprache. Es existiert zwar eine Vielzahl von werkzeugspezifischen Sprachen und Formaten, eine Austauschmöglichkeit, wie im digitalen Fall durch VHDL, ist dort jedoch in aller Regel nicht gegeben.

Die Arbeitsgruppe 1076.1 hat sich deshalb zum Ziel gesetzt, analoge Erweiterungen für VHDL zu entwickeln, um auch zeit- und wertkontinuierliche Signale behandeln und analoge Komponenten beschreiben zu können.

Es ist geplant, die Norm IEEE 1076 in einer späteren Überarbeitung um die angesprochenen Fähigkeiten zu erweitern. Dazu werden zunächst Anforderungen aus einem möglichst großen Publikum gesammelt, analysiert und diskutiert. Daraus entstehen konkrete Vorschläge, wie eine künftige Syntax auszusehen hat. Nach deren Validierung schließt eine Abstimmung der Komiteemitglieder den komplexen Normierungsprozeß ab.

Kontakt: Wojtek Sakowski,
IMAG Institute, University of Grenoble, France,
e-mail: `sakowski@imag.fr`

Die Arbeitsgruppe unterhält auch einen ftp-Server und einen e-mail-Verteiler für interessierte VHDL-Anwender.

Kontakt: e-mail: `1076-1-request@epfl.ch`

3.6 VHDL Initiative Towards ASIC Libraries

Über die "VHDL Initiative Towards ASIC Libraries" (VITAL) wurde bereits im Teil C (Abschnitt 1.5) des Buches berichtet. Der Mangel an verfügbaren Technologiebibliotheken auf Logikebene soll durch die Aktivitäten der Initiative behoben werden.

Kontakt: Erik Huyskens, Alcatel NV, Tel. +32 3 240-7535
e-mail: `ehuy@sh.alcbel.be`

Auch von Vital wird ein e-mail-Verteiler betrieben.

Kontakt: e-mail: `vital-request@vhdl.org`

3.7 E-mail Synopsys Users Group

Ein Beispiel für einen werkzeugspezifischen e-mail-Verteiler ist der sog. "ESNUG-Reflector". ESNUG steht für "E-mail Synopsys Users Group". Hier werden speziell Probleme und Fragestellungen beim Umgang mit den Synthese-Werkzeugen der Firma Synopsys® behandelt.

Kontakt: John Cooley,
e-mail: `jcooley@world.std.com`

4 Disketteninhalt

Dem Buch liegt eine Diskette mit Lösungsmöglichkeiten der Übungsaufgaben aus Teil D, sowie den Beispielen aus den ersten drei Teilen (A bis C) als komplette VHDL-Modelle bei. Die Modelle sind als ASCII-Files mit selbstbeschreibenden Namen abgelegt und wurden mit einem kommerziellen VHDL-Compiler bzw. -Simulator getestet.

Die Diskette gliedert sich in folgende Verzeichnisse:

- `BSP_A` enthält die VHDL-Beispiele aus Teil A,
- `BSP_B` enthält die VHDL-Beispiele aus Teil B,
- `BSP_C` enthält die VHDL-Beispiele aus Teil C,
- `UEBUNGEN` enthält die Lösungen zu den Übungsaufgaben aus Teil D.

  Dieses Verzeichnis verzweigt weiter in die Unterverzeichnisse:

  - `BASICS` enthält die Lösungen zu "Grundlegende VHDL-Konstrukte",
  - `BCD` enthält die Lösungen zur Siebensegment-Anzeige,
  - `ALU` enthält die Lösungen zur arithmetisch-logischen Einheit,
  - `TAB` enthält die Lösungen zum Telefonanrufbeantworter.
    Dieses Verzeichnis enthält auch ein mögliches Syntheseergebnis des Verhaltensmodells als Gatternetzliste in VHDL und als Schematic im PostScript-Format.

In einem weiteren Verzeichnis (`GATE1164`) befinden sich Modelle einschließlich Testbenches für einfache Grundgatter, welche die 9-wertige Logik aus dem Package `std_logic_1164` verwenden. Diese können zum Aufbau eigener, strukturaler Modelle mit diesem Logiksystem herangezogen werden.

Literatur

Die im folgenden aufgeführte Literaturliste beinhaltet nicht nur die im Text verwendeten Zitate, sondern auch weiterführende VHDL-Bücher:

[ARM 89] J. R. Armstrong:
"Chip-Level Modeling with VHDL",
Prentice Hall, Englewood Cliffs, 1989

[ARM 93] J. R. Armstrong, F. G. Gray:
"Structured Logic Design with VHDL",
Prentice Hall, Englewood Cliffs, 1993

[ASH 90] P. J. Ashenden:
"The VHDL Cookbook",
1990, per ftp erhältlich u.a. von folgenden Servern:
`chook.adelaide.edu.au` oder
`du9ds4.fb9dv.uni-duisburg.de`

[BAK 93] L. Baker:
"VHDL Programming with Advanced Topics",
John Wiley & Sons, New York, 1993

[BER 92] J.-M. Bergé, A. Fonkoua, S. Maginot:
"VHDL Designer´s Reference",
Kluwer Academic Publishers, Dordrecht, 1992

[BER 93] J.-M. Bergé, A. Fonkoua, S. Maginot, J. Rouillard:
"VHDL '92",
Kluwer Academic Publishers, Boston, 1993

[BHA 92] J. Bhasker:
"A VHDL Primer",
Prentice Hall, Englewood Cliffs, 1992

[BIL 93] W. Billowitch:
"IEEE 1164: helping designers share VHDL models",
in: IEEE Spectrum, Juni 1993, S. 37

[BIT 92] D. Bittruf:
"Schaltungssynthese auf Basis von HDL-Beschreibungen",
Seminar am LRS, Universität Erlangen-Nürnberg, 1992

[BUR 92] R. Burriel:
Beitrag zum Panel
"Industrial Use of VHDL in ASIC-Design",
EURO-VHDL, Hamburg, 1992

[CAM 91a] R. Camposano:
"High-Level Synthesis from VHDL",
in: IEEE Design and Test of Computers, Heft 3, 1991

[CAM 91b] R. Camposano, L. Saunders, R. Tabet:
"VHDL as Input for High-Level Synthesis",
in: IEEE Design and Test of Computers, 1991

[CAR 91] S. Carlson:
"Introduction to HDL-based Design using VHDL",
Synopsys Inc., 1991

[CAR 93] M. Carroll:
"VHDL - panacea or hype?",
in: IEEE Spectrum, Juni 1993, S. 34 ff.

[COE 89] D. Coelho:
"The VHDL Handbook",
Kluwer Academic Publishers, Boston, 1989

[DAR 90] J. Darringer, F. Ramming:
"Computer Hardware Description Languages and their Applications",
North Holland, Amsterdam, 1990

[GAJ 88] D. Gajski:
"Introduction to Silicon Compilation",
in: Silicon Compilation, Addison-Wesley,
Reading (MA), 1988

[GUY 92] A. Guyler:
"VHDL 1076-1992 Language Changes",
EURO-DAC, Hamburg, 1992

[HAR 91] R. E. Harr, A. Stanculesco:
"Applications of VHDL to circuit design",
Kluwer Academic Publishers, Boston, 1991

[IEE 88] The Institute of Electrical and Electronics Engineers:
"IEEE Standard VHDL Language Reference Manual (IEEE-1076-1987)",
New York, 1988

[IEE 93] The Institute of Electrical and Electronics Engineers:
"IEEE Standard VHDL Language Reference Manual (IEEE-1076-1992/B)",
New York, 1993

[JAI 93] M. Jain:
"The VHDL forecast",
in: IEEE Spectrum, Juni 1993, S. 36

[LEU 89] S. S. Leung, M. A. Shanblatt:
"ASIC system design with VHDL: a paradigm",
Kluwer Academic Publishers, Boston, 1989

[LIS 89] J. Lis, D. Gajski:
"VHDL Synthesis Using Structured Modelling",
in: 26th Design Automation Conference, 1989

[LIP 89] R. Lipsett et al.:
"VHDL: Hardware Description and Design",
Kluwer Academic Publishers, Boston, 1989

[MAZ 92] S. Mazor, P. Langstraat:
"A guide to VHDL",
Kluwer Academic Publishers, Boston, 1992

[MER 92] J. P. Mermet:
"VHDL for simulation, synthesis and formal proofs",
Kluwer international series in engineering and computer science, Dordrecht, 1992

[MER 93] J. P. Mermet:
"Fundamentals and Standards in Hardware Description Languages", NATO ASI Series,
Kluwer Academic Publishers, Dordrecht, 1993

[MGL 92] K. D. Müller-Glaser:
"VHDL - Unix der CAE-Branche",
in: Elektronik, Heft 18/1992

[MIC 92] P. Michel, U. Lauther, P. Duzy (Hrsg.):
"The synthesis approach to digital system design",
Kluwer Academic Publishers, Boston, 1992

[PER 91] D. L. Perry:
"VHDL",
Mc Graw-Hill, New York, 1991

[RAC 93] Racal-Redac Systems Limited:
"SilcSyn VHDL Synthesis Reference Guide",
Tewkesbury, 1993

[SCH 92] J. M. Schoen:
"Performance and fault modeling with VHDL",
Prentice Hall, Englewood Cliffs (NJ), 1992

[SEL 92] M. Selz, R. Zavala, K. D. Müller-Glaser:
"Integration of VHDL models for standard functions",
VHDL-Forum for CAD in Europe,
Santander (Spanien), April 1992

[SEL 93a] M. Selz, S. Bartels, J. Syassen:
"Untersuchungen zur Schaltungssynthese mit VHDL",
in: Elektronik, Heft 10 und 11/1993

[SEL 93b] M. Selz, K. D. Müller-Glaser:
"Synthesis with VHDL",
VHDL-Forum for CAD in Europe, Innsbruck, 1993

[SEL 93c] M. Selz, H. Rauch, K. D. Müller-Glaser:
"VHDL code and constraints for synthesis",
in: Proceedings of VHDL-Forum for CAD in Europe,
Hamburg, 1993

[SEL 93d] M. Selz, K. D. Müller-Glaser:
"Die Problematik der Synthese mit VHDL",
in: 6. E.I.S-Workshop, Tübingen, November 1993

[SEL 94] M. Selz:
"Untersuchungen zur synthesegerechten
Verhaltensbeschreibung mit VHDL",
Dissertation am LRS,
Universität Erlangen-Nürnberg, 1994

[SYN 92] Synopsys, Inc.:
"VHDL Compiler™ Reference Manual, Version 3.0",
Mountain View, 1992

[WAL 85] R. Walker, D. Thomas:
"A Model of Design Representation and Synthesis",
in: 22nd Design Automation Conference, Las Vegas,
1985

Sachverzeichnis

Die im folgenden aufgelisteten Schlagwörter sind zum Teil auch VHDL-Schlüsselwörter, vordefinierte Attribute und gebräuchliche englische Begriffe. In den ersten beiden Fällen werden sie komplett groß geschrieben, im letzteren Fall klein.