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

Teil 1

Inhaltsübersicht:

1 Einleitung und Übersicht 2
2 Grundlagen und Konzepte 4
2.1 Modellierung 4
2.2 Hardware und Software 4
2.2.1 Hardware 4
2.2.2 Software 5
2.3 Digitale Systeme 5
2.4 Domänen der Modellierung 6
2.5 Modellierungssprachen 8
3 VHDL Modellierungskonzepte 9
3.1 Grundsätzlicher Aufbau von VHDL-Beschreibungen 9
3.1.1 Schnittstelle eines Modells / Entity 9
3.1.2 Funktionalität eines Modells / Architecture 9
3.1.3 Beispiel eines Modells 10
3.2 Beschreibungsformen mit sequenzieller Bearbeitung 11
3.3 Beschreibungsformen mit paralleler Bearbeitung 11
3.4 Formen der Strukturbeschreibung 12
3.5 Gemischte Verhaltens- und Strukturbeschreibung 14
3.6 Testbenches für VHDL-Modelle 14

1 Einleitung und Übersicht

Die Motivation zu dieser Veranstaltung ist es, nicht eine weitere prozedurale (imperative) Programmiersprache zu vermitteln, sondern die Programmierung einmal etwas anders zu betrachten. VHDL hat zwar den Charakter einer Programmiersprache, denn VHDL stammt von Ada ab, ist aber nicht rein prozedural. VHDL ist auch keine rein deskriptive Sprache, wie wir noch sehen werden. VHDL ist ein Mix aus beiden.
VHDL ist eine Sprache, die es ermöglicht sowohl den statischen Charakter von Systemen zu beschreiben (Strukturbeschreibungen) als auch ihr dynamisches Verhalten. VHDL Modelle sind simulierbar, d.h. das Modellverhalten kann dynamisch in einen zeitlichen Ablauf gebracht und über sog. Testbenches beobachtet werden.
Der Ursprung von VHDL (Very high speed integrated circuits Hardware Description Language) zielt zwar auf die Beschreibung von vornehmlich digitalen Schaltungen, doch hat man erkannt, dass die Sprache wesentlich mächtiger ist. Hierzu werden wir noch Beispiele sehen.
Trotzdem kommen wir nicht umhin, zunächst aus der Sicht des Entwurfs digitaler Schaltungen zu beginnen. Die Mächtigkeit der Sprache VHDL ermöglicht die Modellierung über alle Ebenen und Sichten eines Schaltungsentwurfs. Damit erfüllt sie alle Kriterien, die zu einem abstrakten Entwurf über Modellbildungen eine sukzessiver Verfeinerung der Modelle ermöglicht und mit der Verfeinerung bei einer Beschreibungsform (strukturelles VHDL) endet, über die ein Layout des realen Systems automatisch generiert werden kann. Damit bilden VHDL Modelle von der hohen Ebene der Spezifikation bis zur letzten Stufe vor der automatischen Umsetzung auf den Chip die Möglichkeit des durch Simulation validierbaren Entwurfs. Moderne Ansätze nutzen VHDL Modelle auch zur Verifikation des Modellentwurfs.

Der hier verfolgte Ansatz von der abstrakten Beschreibungsebene zur konkreten Programmierung hochkomplexer Logiken soll den Hörer von einer bekannten Programmiersprache hinüberführen zu Modellierungs- und Beschreibungssprachen und darlegen, was eine Simulation bedeutet und wie man sie am Beispiel der Sprache VHDL realisieren kann.

Der in den praktischen Modellbeispielen verfolgte Lern-Nebeneffekt ist erstens das Erarbeiten und Vertiefen der Grundkenntnisse über digitale Schaltungen sowie der Struktur und Arbeitsweise von Prozessoren durch die Modellierung auf der Verhaltensebene und zweitens das Erlernen der Grundlagen zu Hardware-Interfaces verbunden mit der Programmierung von komplexen Logiken (CPLDs). Komplexe Logikbausteine FPGAs und insbesondere CPLDs werden zunehmend bei der Konstruktion von RISC Prozessorsystemen eingesetzt, um im Wesentlichen die Adressierung von Speicher- und Interfaces zu übernehmen, aber auch zur Ansteuerung einfacher Peripherie zu dienen.

Übersicht

Ein kleiner Überblick soll aufzeigen, wie wir den Kurs aufbauen und uns den Anwendungsmöglichkeiten von VHDL nähern.

Da die Kenntnis einer Programmiersprache wie z.B. Pascal, C oder Ada vorausgesetzt wird, starten wir den VHDL Exkurs auf einer sehr hohen Verhaltensebene, um den „Umstieg“ zu erleichtern. Dabei gehen wir zunächst von bekannten Algorithmus in C bzw. Ada aus und konvertieren direkt zu VHDL-Modellen. Es wird gezeigt, worin die wesentlichen Unterschiede zwischen klassischen, prozeduralen Programmiersprachen und einer Hardware-Beschreibungssprache wie VHDL liegen.

Wir werden dann sukzessive anhand vieler Beispiele aus der Digitaltechnik die Sprachsyntax und die Grundlagen der Simulation kennen lernen.

Weiterhin werden wir eine Reihe von Modellierungsparadigmen erarbeiten und auf Beispiele aus digitalen Rechnertechnik mit den wichtigsten Komponenten eines Rechners (Prozessoren, Speicher und Interfaces) anwenden, wobei wir mit dem Lernen der Sprache auch Wissen über Prozessoren auf der Maschinenebene erlangen und anwenden. Wir werden dies aus verschiedenen Sichten der Beschreibung und Modellierung in VHDL Beschreibungen realisieren. Wir lernen dazu die Beschreibungsmittel und

den Programmierstil in VHDL. Gleichzeitig werden die Regeln aufgezeigt, die ein VHDL Modell synthesefähig macht.

In Vorbereitung auf moderne, insbesondere eingebettete Systeme (embedded systems) werden wir zum Ende des Kurses die VHDL Restriktionen erarbeiten, die die VHDL Beschreibung zur Programmierung komplexer, programmierbarer Logikbausteine ermöglicht. Derartige Logikbausteine (z.B. CPLDs) finden ihren Einsatz zur Adress-Decodierung, zur Implementierung von Finite-State Machines (FSMs) und von peripheren Einheiten zu Microcontrollern in eingebetteten Systemen.

Historische Entwicklung von VHDL

Der Vollständigkeit halber ist hier ein kurzer Abriss der Entstehungsgeschichte der Sprache VHDL chronologisch angegeben.

1981
VHSIC Programm des US DoD; verschiedene Chiplieferanten sollten einheitliche Beschreibungen ihrer Produkte vorweisen können

1983
IBM, Texas Instruments und Intermetrics entwickeln zusammen VHDL

1985
erste veröffentlichte Version von VHDL, V. 7.2

1986
Übergabe von VHDL an den IEEE zur Standardisierung

12/1987
IEEE Standard 1076-1987; Language Reference Maunual (LRM)

1993
Überarbeitete Version des Standards 1076-1993
Vereinheitlichung der Modellierung von Logiksignalen zur sog. 9-wertigen Logik (Real sind die Logiksignale nicht nur auf 0/1 bzw. false/true beschränkt!) mit dem IEEE STD_LOGIC_1164 Packet.

Seit 9/1988 verlangt das US Militär nach ihrer Standard MIL 454, dass alle Zulieferer ihre Produkte in VHDL auf der Verhaltens- und Strukturebene beschreiben und auch die Testsätze dafür, in VHDL beschrieben, beistellen.

2 Grundlagen und Konzepte

2.1 Modellierung

Zu Beginn sollten wir uns zuerst über den Begriff MODELL klar werden. Nach [BRJ99], dem Benutzerhandbuch zu UML (unified modeling language) ist die einfachste Interpretation dieses Begriffs:

Ein Modell ist eine Vereinfachung der Realität.

Unter der "Realität" kann man hier die unterschiedlichsten Dinge verstehen, doch wollen wir uns hier auf Modelle von Systemen beschränken. Unter einem System wiederum können wir ganz allgemein eine Gruppierung von kooperierenden, interagierenden Objekten verstehen, die als die Gesamtheit System gegenüber ihrer Umwelt ein Verhalten zeigen.
In [BRJ99] heißt es weiter: Ein Modell liefert die Entwürfe zu einem System. Modelle können sowohl detailliertere Pläne umfassen als auch eher allgemeine Pläne, die einen Überblick über das zu betrachtende System gewähren. Ein gutes Modell betrachtet die Elemente, die weitreichende Auswirkungen haben und übergeht die, die für das gegebene Abstraktionsniveau nicht relevant sind. Jedes System kann unter verschiedenen Aspekten mit unterschiedlichen Modellen beschrieben werden, jedes Modell ist daher eine semantisch abgeschlossene Abstraktion des Systems. Modelle können strukturbezogen sein, d.h. die Oragnisation des Systems steht im Vordergrund, oder sich auf das Verhalten beziehen, dann ist die Dynamik des Systems von Bedeutung.

Wir bauen Modelle, um das zu entwickelnde System besser verstehen zu können.

Während UML eine allgemeine, nicht ausführbare Beschreibungssprache ist, sie dient mehr der Spezifikation und Dokumentation von Systemen, hat VHDL den Charakter einer Programmiersprache, d.h. VHDL liefert auch ausführbaren Code, oder besser formuliert Programmcode als Beschreibungsmodell, das anhand von Tests in Form von Stimulanzmustern (Eingaben) und Reaktionen darauf (Ausgaben) dynamisch ausführbar ist.
Genauer betrachtet beschreibt das VHDL Modell ein System in seiner Funktion (das was simuliert werden soll) und erst die Stimulanz ermöglicht das Simulieren der Dynamik. Hier erkennt man den Ursprung der Sprache, nämlich die Verhaltensbeschreibung einer digitalen Schaltung, z.B. ein logisches Gatter oder ein Flipflop, die zunächst in ihrer Struktur und Funktion beschreiben wird und Anschlusssignale in Form von Eingängen und Ausgängen haben. Erst das Belegen der Eingänge (Stimulus) bewirkt eine Ausgabe (Reaktion, Antwort). Wir werden sehen, dass der Großteil aller VHDL Modellsimulationen aus diesen zwei Teilen (Prozessen) bestehen wird, das Modell und die sog. Testbench.
Da wir also im Wesentlichen mit Prozessoren und mit deren Programmen umgehen wollen, sollten wir noch einige Begriffsklärungen vornehmen.

2.2 Hardware und Software

Bevor wir also in die Details der Modellierung einsteigen, sind die Begriffe Hardware und Software klar zu stellen.

2.2.1 Hardware

In [CS01] finden wir dazu Folgendes:

Hardware ist die physikalische Implementierung einer Funktionalität.

Wir wollen hier darunter mikroelektronische Systeme sehen, die eine gewünschte Funktionalität erfüllen. Weiter einschränkend sollen diese rein digitale Schaltungen sein, die wir in diesem Kurs beschreiben (modellieren) und simulieren wollen.

Die Funktionalität der Hardware kann weiter darauf beschränkt werden, dass sie einen Befehlsumfang implementiert, also einen sog. Prozessor oder in der technischen Informatik auch mit Maschine bezeichnet. Zusammen mit einem Speicher und darin befindlicher Programme als Abfolge solcher Befehle kommt man dann zum Begriff der Software.

2.2.2 Software

Bemühen wir wieder einmal [CS01],

Unter Software versteht man die logische Implementierung einer Funktionalität zur Ausführung durch ein mikroelektronisches System (Hardware, Prozessor). Im engeren Sinne ist darunter die Folge von Befehlen gefasst, die von dieser Hardware zur Verfügung gestellt und in einem strukturierten Ablauf gebracht wird.

Wir sehen in beiden Definitionen, dass es um die Implementierungen einer *Funktionalität* geht. Demgemäß ist dieselbe Funktionalität gesamtheitlich sowohl in Hard- auch als in Software implementierbar. Im Fall der reinen Hardware-Implementierung erfüllt eine Verschaltung elektronischer Komponenten die Funktionalität. Im Software-Fall unterliegt der programmatischen Beschreibung eine Standard-Hardware zur Interpretation. Der Interpreter hat damit die Funktion, einen Befehlssatz, die sog. Programmiersprache, zu implementieren, einen Speicher zur Speicherung von Befehlsfolgen der Programmiersprache zu adaptieren und über einen Automatismus zur Abarbeitung der gespeicherten Programme zu verfügen.

Wie bereits einleitend gesagt, werden wir uns bei der Modellierung stark auf Prozessoren konzentrieren, weil ein Nebenziel dieser Veranstaltung die Strukturmerkmale moderner Prozessoren ist.

2.3 Digitale Systeme

Da wir uns, wenn wir in diesem Kurs Hardware-Modelle bilden, auf digitale Systeme konzentrieren, wollen wir auch hier insbesondere ansehen, was man unter programmierbarer Hardware versteht. Dies sollte nicht mit der Software verwechselt werden. Die Grafik im Bild 2.1 veranschaulicht das.

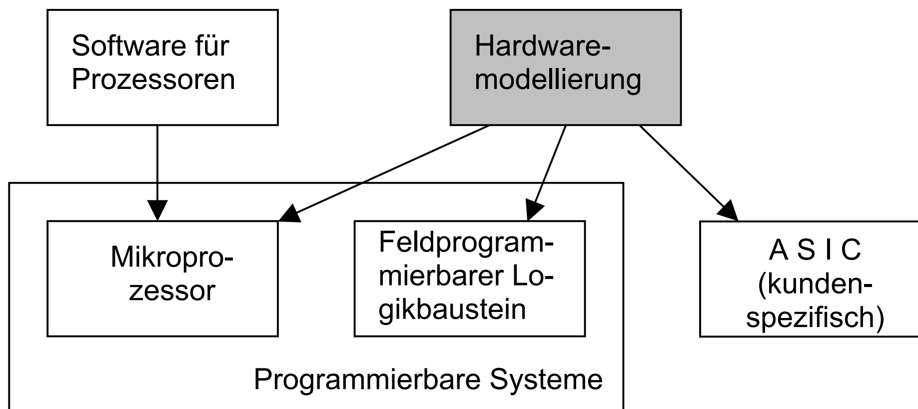

Bild 2.1: Betrachtete digitale Systeme

Der Mikroprozessor ist nach den obigen Betrachtungen klar ein programmierbares System, das Software im oben beschriebenen Sinne interpretiert. Dem gegenüber steht die Programmierbarkeit von komplexen Logikbausteinen. Hier bezeichnet der Begriff Programmierung die flexible Verschaltung funktional bestimmbarer Logikblocks. Bild 2.2 zeigt ein Prinzipschaltbild sog. CPLDs (complex programmable logic devices), deren Modellierung und Programmierung wird gegen Ende der Kurses näher betrachten.

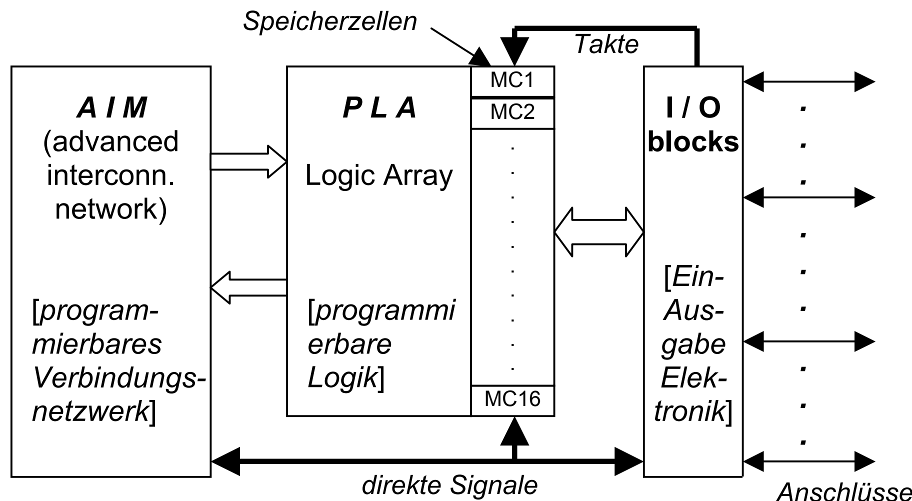

Bild 2.2: Prinzipschaltbild eines CPLD

Wie aus dem Bild 2.2 zu ersehen, kann man 4 Blöcke unterschiedlicher Funktionalität unterscheiden. Betrachten wir zunächst die Speicherelemente MCi. Diese bestehen aus Flipflops - wir setzen die Kenntnis der Flipfloptypen voraus - mit programmierbaren Takteingängen und Dateneingängen, die ebenso programmierbar sind, da sie aus den Produkttermen eines PLA gebildet werden. Ein Verbindungsnetzwerk verbindet die Eingänge des PLA und die Ausgänge der MCi-Speicherelemente mit den primären I/O Pins des Chips.
Auf Grund der freien, nichtflüchtigen Programmierbarkeit der Flipflop-Taktung, der Flipflop-Eingänge, der logischen Funktion dieser Eingänge im PLA sowie der Verbindungsstruktur, ist das CPLD ein äußerst universell einsetzbares Modul.
Für diese Bausteine gibt es überaus leistungsfähige Werkzeuge und heute nutzt man VHDL für den Entwurf.
ASIC, d.h. Kundenspezifische Schaltungen werden heute fast ausschließlich mit VHDL entwickelt. VHDL erlaubt es, von der hochfunktionalen Beschreibung im Rahmen der VHDL-Umgebung die Schaltungen stückweise herunterzubrechen und alle gewonnenen Neuausprägungen der Schaltung konsequent zu simulieren. Wir werden noch betrachten, wie man das letzte Modell in VHDL so gestaltet, dass damit Synthetisierwerkzeuge automatisch Layouts generieren können.

2.4 Domänen der Modellierung

Wir wollen uns jetzt ansehen, wie man vorgehen kann, wenn man ein System entwerfen will. Dazu sollten wir zunächst aus einer höheren Ebene betrachten, wie ein Entwurf ablaufen kann und welche Abstraktionsebenen der Betrachtung angeführt werden können. Eine relativ anerkannte Darstellung geht auf Gajski zurück und ist in Bild 2.3 (aus [CS01]) dargestellt. Sie ist in fast jedem Lehrbuch zum Entwurf zu finden.

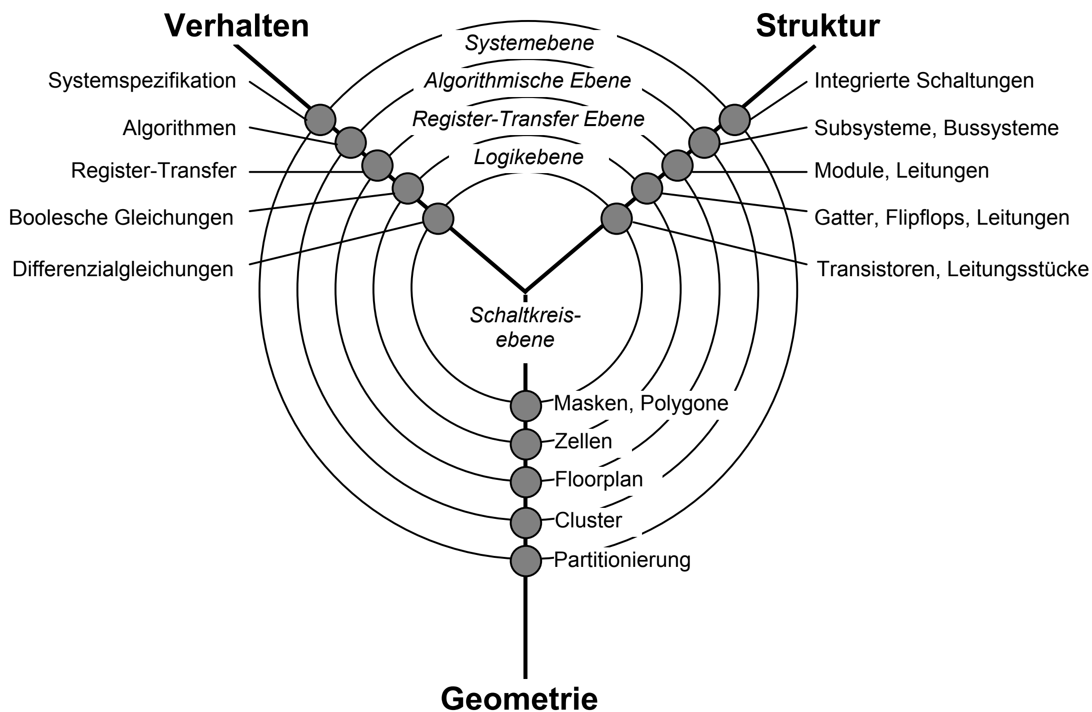

Bild 2.3: Y-Diagramm nach Gajski; aus [CS01]

Nach der Grafik im Bild 2.3 unterscheidet man prinzipiell drei Sichten auf ein System, die Verhaltenssicht, die Struktursicht und die Geometriesicht.
Die Verhaltenssicht beschreibt die Ausgangs-Reaktionen eines Systems aufgrund von Eingangs-Stimuli, beschreibt also mehr oder weniger exakt die Funktionsweise des Systems.
Eine daraus endgültig realisierte Schaltung, sofern man das Entwurfsziel eines ICs hat, wird aus der Geometriesicht betrachtet. Diese Sicht ist für uns hier nicht von Interesse, da wir nur auf System-Simulationen und auf die Programmierung von Komplexlogiken (CPLDs) zielen.
Die Struktursicht ist jedoch für diesen Kurs von Belang, da sie die klassischen Elemente der funktionalen Beschreibung von Systemen beinhaltet und das wichtigste Mittel der Strukturierung eines Entwurfs ist. Wir werden sehen, dass ein VHDL-Modell erst durch das Einführen struktureller Elemente die Modelle beherrschbar und vor allem synthetisierbar macht.
Das Schalenmodell nach Bild 2.3 wird von außen nach innen mit zunehmender Konkretisierung betrachtet, d.h. dass es nach innen eine zunehmende Detailtreue des Entwurfs gibt. Bild 2.4 zeigt die wesentlichen Zusammenhänge.
Die abstrakteste Ebene ist die Systemebene, auf der das Verhalten durch die Spezifikation des Systems gegeben ist. Die Struktur zeigt hier den IC, d.h. einen Block mit seinen Ein- und Ausgängen.
Die Algorithmische Ebene offenbart schon den ersten Ansatz, eine innere Struktur zu erkennen, d.h. aus der Verhaltenssicht einen Algorithmus zur Implementierung der Spezifikation, aus der Struktursicht die Aufteilung des Systems in Subsysteme, wie CPU, Speicher, Ein-/Ausgabe, Busse etc.. Armstrong und Gray [AG 93] bezeichnen diese Ebene treffender mit Chipebene.
Die Register-Transferebene sieht aus der Verhaltenssicht den Datenfluss des Systems und ist vergleichbar auf der Strukturebene mit Modulen (z.B. Register, ALU, MUX etc.) und Leitungsverbindungen (Datenwege).
Die Gatterebene sieht auf der Verhaltensseite Boolesche Gleichungen und auf der Strukturseite die logischen Gatter (AND, OR, XOR) und Flipflops zu ihrer schaltungstechnischen Realisation.
Die Gatterebene ist schließlich die, bei der die Betrachtungen in diesem Kurs enden, denn die beiden darunter liegenden sind das Feld derer, die die Chips letztendlich in ihren elektronischen Eigenschaften berechnen, das Layout erstellen und den Chip dann fertigen und testen.
Bild 2.4 kennzeichnet das allgemeine Vorgehen, wenn man Systeme entwirft.

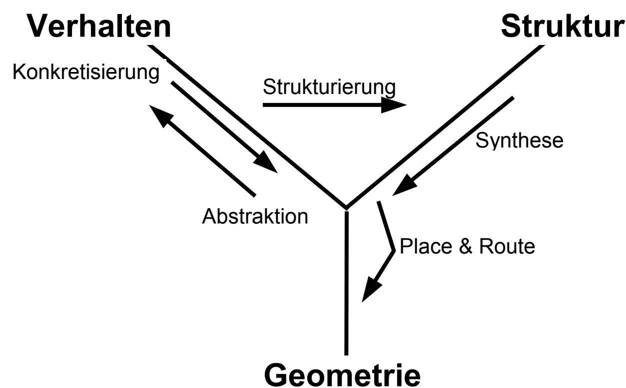

Bild 2.4: Entwurfsprozess im Y-Diagramm; aus [CS01]

Gajski beschrieb als erster an diesem Diagramm theoretisch das Entwurfsvorgehen. Man beginnt mit der Spezifikation als die höchste Abstraktionsstufe, erstellt einen Algorithmus, der das erste Modell zur Abdeckung der Spezifikation bildet und verfeinert den Algorithmus, falls aus der Verhaltenssicht möglich.
Die Strukturierung des Verhaltensmodells führt dann zur Struktursicht des Modells, in der man insbesondere auch Fragen der Wiederverwendbarkeit von Strukturelementen klären kann. In Praxi gibt es während des Entwurfs eine häufiges "Springen" zwischen beiden Sichten, ehe ein konkretes, der Spezifikation entsprechendes Modell entstanden ist.
Ein wichtige Rolle spielt hier auch die Synthese. Der Schritt von einem Strukturmodell zum endgültigen Layout -Place & Route - wird heute weitgehend automatisch mit einem Software-Werkzeug vollzogen. Das gilt auch für die Programmierung von CPLDs. Voraussetzung dafür ist aber die Einschränkung des Strukturmodells auf ganz bestimmte Komponenten. Setzt man VHDL voraus, so spricht man hier von synthesefähigem VHDL, das eine maschinelle Synthese des Modells erlaubt.

Siemers [CS01] gibt eine treffende Definition des Entwurfs:

Der Entwurf im Sinne eines Hardwaredesigns ist die Transformation eines abstrakten Modells in eine Verschaltung verfügbarer Komponenten unter Berücksichtigung weiterer Randbedingungen wie Geschwindigkeiten und Ressourcen dieser Komponenten sowie Energieumsetzung etc.

2.5 Modellierungssprachen

Die Modellierungssprache VHDL ist mächtig genug, aus Verhaltens- und Struktursicht auf allen oben beschriebenen Ebenen das programmtechnische Grundgerüst zu stellen, um Modellierungen vorzunehmen.
Vielfach, wie auch in [CS01] dargestellt, werden für das hochabstrakte Modell auch Sprachen wie Hardware-C++ oder HardJava benutzt, doch wird durch den sehr hohen Abstraktionsgrad das Umsetzen zunehmend abhängig vom Geschick und der Erfahrung des Programmierers. Die "Modelltreue" ist hier ein wichtiger Faktor. Diese geht natürlich mit zunehmend menschlicher Intuition bei der Modellierung verloren und muss durch intensive Tests belegt (validiert) werden. Eine Verifikation ist nur bei streng formaler Spezifikation möglich. Dazu sind dann formale Modelltransformationen notwendig, die streng mathematisch bewiesen sein müssen. Hier besteht nach wie vor noch viel Forschungsbedarf.
VHDL-Modelle sind auch zur Spezifikation geeignet, doch hochkomplexe Modelle führen zu sehr hohen Simulationszeiten.
Eine in Grenzen zu VHDL konkurrierende Hardware-Beschreibungssprache ist *Verilog*. Die Unterschiede sollen hier nicht diskutiert werden, doch bietet Verilog im Gegensatz zu VHDL nicht die Möglichkeit, Verhaltens- und Strukturmodelle zu mischen.
Nicht unerwähnt bleiben sollte im Zusammenhang mit der Programmierung von Komplexlogiken die Sprache Abel-HDL. Ein Überblick dazu wird in [CS01] gegeben. Abel-HDL ist absolut struktur-orientiert, d.h. sie erlaubt nur die Beschreibung von Booleschen Gleichungen und Flipflops, die in einem z.B.

FPGA auftreten. Allerdings erlaubt Abel-HDL die Angabe von Testvektoren so, dass eine Simulation möglich wird. Abel-HDL dient allein der Programmierung von Komplexlogiken.

Alles in allem ist VHDL allen anderen Sprachen und Ansätzen anderer Art zur Modellierung und Simulation von Hardware überlegen und besitzt eine wesentlich höhere Mächtigkeit, um auch darüber hinaus gehende Modellbildungen vorzunehmen.

3 VHDL Modellierungskonzepte

3.1 Grundsätzlicher Aufbau von VHDL-Beschreibungen

Die VHDL-Beschreibung eines Modells besteht grundsätzlich aus zwei Teilen, dem *Entity*- und dem *Architecture*-Teil. Da die Sprache auf die Beschreibung von Hardware zielt, entspricht ein Modell letztendlicher einer Hardware-Einheit, die mit anderen verschaltet werden kann. Die *Entity* beschreibt daher die Schnittstelle der Einheit oder allgemeiner des Modells. Die *Architecture* beschreibt dann die Funktionalität des Modells. Dabei ist es durchaus möglich, mehrere Architekturen für das selbe Modell zu definieren. Dies ermöglicht das Modellieren aus verschiedenen Sichten.

3.1.1 Schnittstelle eines Modells / Entity

Pro Modell darf nur eine Entity existieren. Die vollständige Entity mit allen Vereinbarungen ist in Bild 3.1 dargestellt. Alle optionalen Teile sind geklammert.

```
entity entity_name is
[  generic (par_1 {, par_k}: type_name [:= def_val] {; further_gen_decl} );  ]
[  port ( { port_i1 {, port_ik}: IN type_name  [:= def_val]; }
          { port_o1 {, port_ok}: OUT type_name  [:= def_val]; }
          { port_io1 {, port_iok}: INOUT type_name  [:= def_val]; }
          { port_buf1 {, port_bufk}: BUFFER type_name  [:= def_val]; }
         );  ]
end  [entity]  [entity_name];
```

Bild 3.1: Entity Aufbau

Der wichtigste Bestandteil der Entity eines Modells ist die Port-Vereinbarung, da sie die Schnittstellensignale zu anderen Entitäten beschreibt. Den Signalen sind folgende Modi zugeordnet, die den Zugriff beschränken:
IN: Signal darf nicht geschrieben werden, OUT: Signal kann nicht gelesen werden,
BUFFER: Signal kann gelesen werden, hat aber nur eine schreibende Quelle und letztlich INOUT: Signal ist ohne Einschränkung les- und schreibbar. Signaltypen und Werte werden später detailliert behandelt.
Die Generic-Vereinbarung ermöglicht eine Parametrierung der Entity, auf die später noch eingegangen wird.

3.1.2 Funktionalität eines Modells / Architecture

Da es für eine Entity mehrere Architekturen geben kann, erhält die Architecture eine Bezeichnung. Das Schlüsselwort *OF* gefolgt vom Entity-Namen kennzeichnet die Zugehörigkeit der Architektur. Wie bei allen höheren Sprachen kennzeichnen BEGIN und END den Programmierungsteil der die Funktionalität beschreibt. Bild 3.2 zeigt den generellen Aufbau einer Architektur.

```
architecture arch_name of entity_name is
--  lokaler Deklarationsteil:
--  USE-Anweisungen
--  Typen, Untertypen,
--  Alias, Konstanten, Variablen, Signalen
--  Files, Komponenten,
--  Unterprogramme, Attribute, Konfigurationen
--  Definitionen von Unterprogrammen, Attributen,
```

```
--  Shared Variables (global)
begin
  --  nebenläufige Anweisungen zur Modellierung
  --  (strukturelle ~, Verhaltensbeschreibung)
end  [architecture]  [arch_name];
```

Bild 3.2: Architecture Aufbau

3.1.3 Beispiel eines Modells

Am Beispiel einer XOR-Funktion sei gezeigt, wie ein VHDL-Modell dazu aussehen kann. Bild 3.3 zeigt die Entity der XOR-Funktion, wobei wir hier noch keine generischen Parameter vorgesehen haben. Diese legt fest, welche Ein- und Ausgänge das Modell hat und von welchem Typ diese sind. Da wir hier zunächst reine boolesche Logik betrachten, nehmen wir auch den Typ ***boolean*** an.

```
entity xor_gate is
  port (a, b : in  boolean;      -- bit
        c    : out boolean);     -- bit
end xor_gate;

architecture Specification of xor_gate is
begin
  c <= a xor b;
end architecture Specification;
```

Bild 3.3: XOR-Gattermodell

Man beachte, dass in der Architektur eine sog. Signalzuweisung auftritt, da ***c*** eben ein Ausgangssignal des Modells ist. Signale und derartige Zuweisungen werden später noch näher erläutert. Die Architektur kann frei benannt werden, wobei wir hier den Namen *Specification* gewählt haben, um zu zeigen, dass die Namenswahl einer Architektur frei ist.

Im Bild 3.3 ist eine mögliche Architekturbeschreibung angegeben, die als sehr abstrakt angesehen werden kann, denn die Funktionalität drückt sich allein in dem Operator *xor* der Signalzuweisung aus.
Einer Entity kann durchaus mehr als eine Architektur zugeordnet werden. Um ein Gefühl dafür zu bekommen, was sich hinter den Ebenen und Sichten des Gajski Diagramms (Bild 2.3) verbirgt, wollen wir das XOR als weitere Architekturen beschreiben. Wohlbemerkt gilt der Kopf und die Entity wie oben gehabt weiter.

Zunächst sei aber die erste, wichtige Analogie betrachtet. Wir wollen **true** und **false**, also die logischen Werte in Analogie zu den Bitwerten **’0’** und **’1’** betrachten. Dies ermöglicht überhaupt erst den Übergang zu technischen Systemen. Man bezeichnet die Zuordnung **false <=> ’0’** / **true <=> ’1’** auch mit positiver Logik. Die Umkehrung dieser Zuordnung wird mit negativer Logik bezeichnet. Mit dieser Analogie lassen sich ’0’ und ’1’ als zwei physikalische Zustände eines Signals festlegen, hier z.B. eine Leitung hat einen bestimmten Spannungspegel (üblich auch VCC bezeichnet) als logische ’1’ und hat keine Spannung, also 0V (auch GND bezeichnet) als logische ’0’.
VHDL bietet hierzu die sprachlichen Möglichkeiten, in dem es einerseits den Datentyp **boolean** kennt und andererseits den Typ **bit**. Die logischen Grundfunktionen, wie z.B. hier das **xor** werden automatisch mit der Deklaration solcher Objekte vergeben. Damit ist das Gattermodell in Bild 3.3 auch beschreibbar durch Austauschen des in Kommentaren angeführten Typs **bit** gegen **boolean**.

Das XOR Gatter wird in Schaltplänen nach der DIN-Norm wie folgt angegeben. Rechts die bekannte Wahrheitstabelle dazu.

a	b	c
0	0	0
0	1	1
1	0	1
1	1	0

3.2 Beschreibungsformen mit sequenzieller Bearbeitung

Wir wollen zunächst den Begriff eines reinen VHDL-Verhaltensmodells klären. In [CS01] findet man die Definition:

*Das Modell gilt als ein reines **Verhaltensmodell**, wenn es keine weiteren, untergeordneten Komponenten im Rahmen der Beschreibung instanziert.*

Diese Definition gilt immer dann, wenn es eine monolithische, also aus einem Prozess bestehende Beschreibung geht. Innerhalb von Prozessen sieht VHDL eine prinzipiell sequenzielle Bearbeitung der Befehlsfolgen vor. Ein Prozess ist daher auch als ein Stück Software zu betrachten und, wie wir sehen werden auch (fast) genauso zu programmieren. Ein Beipiel der Architektur für unser XOR ist im Bild 3.4 gegeben.

```
architecture Algorithmic of xor_gate is
begin
  process(a, b)
  begin
    if a /= b then
      c <= '1';           -- boolean
    else c <= '0';        -- boolean
    end if;
  end process;
end architecture Algorithmic;
```

Bild 3.4: XOR-Prozessmodell

3.3 Beschreibungsformen mit paralleler Bearbeitung

Da VHDL ursprünglich definiert wurde, um Hardware zu beschreiben und zu simulieren, musste auch die Möglichkeit paralleler Bearbeitung vorgesehen werden. Hierzu wurden Signale und Signalzuweisungen vorgesehen, die bei der Simulation als nebenläufig behandelt werden. Nur so ist es unter anderem möglich, auch Verzögerungszeiten für Signale einzubeziehen.
Bild 3.6 zeigt unser XOR als ein Modell mit diesen Eigenschaften.

```
architecture Datenfluss of xor_gate is
  signal x, y: bit;
begin
  c <= x or y;
  x <= a and not b;
  y <= not a and b;
end architecture Datenfluss;
```

Bild 3.6: XOR Modell mit Signalzuweisungen

In diesem Modell macht es keinerlei Unterschied, in welcher Reihenfolge die Signalzuweisungen angeführt werden. Das Simulationsergebnis ist immer gleich. Wie das realisiert wird, werden wir noch später behandeln.

Man kann am Modell im Bild 3.5 aber ablesen, dass ein gewisser Signalfluss oder Datenfluss beschreiben ist. Daher findet man für solche Modelle auch den Begriff Datenfluss-Modell. Werden später noch Speicherelemente (Register) beteiligt, spricht man von Register-Transfer Modellen.
Dass man bis auf die Ebene der Booleschen Gleichungen und Wahrheitstabellen gehen kann, sollen die folgenden zwei Architekturen im Bild 3.6 andeuten.

```
architecture BooleGln of xor_gate is
begin
  c <= (a and (not b)) or ((not a) and b);
end BooleGln;


architecture WHT of xor_gate is
begin
  c <=
    '0' when (a = '0') and (b = '0') else
    '1' when (a = '0') and (b = '1') else
    '1' when (a = '1') and (b = '0') else
    '0' when (a = '1') and (b = '1') else
    '0';
end architecture WHT;


architecture Arithmetic of xor_gate is
begin
  c <= b when (a = '0') else not b;
end architecture Arithmetic;
```

Bild 3.6: XOR Modell auf der binären Logik basierend

Eine interessante Betrachtungsweise der XOR-Funktion mit Hinblick auf die später behandelten arithmetischen Schaltungen sieht die Eingänge des XOR-Gatters als Steuer- und Datenleitung. Sieht man in der Wahrheitstabelle die Funktion für **a = ’0’**, so erkennt man, dass damit **c = b** wird. Sieht man die Fälle **a = ’1’**, so folgt, dass **c = not b** ist, also invertiert! Wenn wir später die Dualdarstellung von Ganzzahlen (Integer) betrachten, kann man bei Stellenbeschränkung negative Zahlen im Komplement darstellen. Man unterscheidet hier Einer- und Zweierkomplement. Die Einerkomplement-Darstellung entsteht einfach durch die Invertierung aller Bitstellen. Jede Datenstelle (Bit) kann also mit dem Steuersignal **a = ’0’** positiv oder mit **a = ’1’** negativ gemacht werden.
Im VHDL-Modell kann man auch dieser Betrachtung Rechnung tragen, wie im Bild 3.6 mit der Architektur *Arithmetic* angedeutet.

3.4 Formen der Strukturbeschreibung

Von reiner Strukturbeschreibung redet man, wenn man an einen Schaltplan denkt. Sehen wir z.B. die XOR Funktionalität als Schaltung aus logischen Gattern, wie im Bild 3.7 gezeigt.

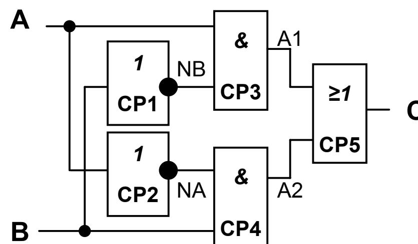

Bild 3.7: XOR Schaltnetz

Dazu kann man ein VHDL Modell schreiben, dass dieser Schaltung entspricht. Man beschreibt dabei die Funktionalität der Komponenten, hier UND-Gatter, ODER-Gatter und Inverter und beschreibt in der Architektur die Verschaltung der Komponenten, die sog. Netzliste.

```
entity inverter is
  port(A: in bit; B: out bit);
end inverter;

architecture basic of inverter is
begin
  B <= not A;
end architecture basic;

--------------------

entity and2 is
  port(A, B: in bit; C: out bit);
end and2;

architecture basic of and2 is
begin
  C <= A and B;
end architecture basic;

--------------------

entity or2 is
  port(A, B: in bit; C: out bit);
end or2;

architecture basic of or2 is
begin
  C <= A or B;
end architecture basic;
--------------------

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

architecture Structure of xor_gate is
  signal NA, NB, A1, A2: bit;
begin
  CP1: entity work.inverter(basic)
    port map(A, NA);
  CP2: entity work.inverter(basic)
    port map(B, NB);
  CP3: entity work.and2(basic)
    port map(A, NB, A1);
  CP4: entity work.and2(basic)
    port map(NA, B, A2);
  CP5: entity work.or2(basic)
    port map(A1, A2, C);
end architecture Structure;
```

Bild 3.8: XOR als VHDL-Strukturmodell

Man kann sich vorstellen, dass dieses Modell der Ausgangspunkt eines Layouts sein kann. Statt der eigenen Entity-Vereinbarungen der Komponenten entnimmt man diese aus vorentwickelten Baustein-Bibliotheken und beschreibt nur noch die Netzliste.

Hier ist also der Übergang zu sehen, ab wo man die VHDL-Beschreibung verläßt und zu einem automatischen Layout wechselt.

3.5 Gemischte Verhaltens- und Strukturbeschreibung

Ein wesentliches Merkmal von VHDL-Beschreibungen ist es, dass man beide Beschreibungsformen beliebig mischen kann. Dies ist die Voraussetzung, um Modelle auf allen oben beschriebenen Betrachtungsebenen zu erstellen und vor allem auch zu simulieren. Bild 3.9 soll nur einen Eindruck vermitteln, wie ein solches Modell aussehen kann. Das Modell werden wir später noch näher beleuchten. Es entstammt dem Buch [PA02] von P.J. Ashenden und wird einen Festkomma Multiplizierer modellieren.

```
entity multiplier is
  port (Clk, Reset: in  bit;
        Multiplicand, Multilpier: in integer;
        Product: out integer);
end multiplier;

architecture mixed of multiplier is
  signal partial_product, full_product: integer;
  signal arith_control, result_en, mult_bit, mult_load: bit
begin   -- mixed
  arith_unit: entity work.shift_adder(behavior)
    port map(addend => multiplicand, augend => full_product,
             sum => partial_product,
             add_control => arith_control);
  result: entity work.reg(behavior)
    port map(d => partial_product, q => full_product,
             en => result_en, reset => reset);
  multiplier_sr: entity work.shift_reg(behavior)
    port map(d => multiplier, q => mult_bit,
             load => mult_load, clk => Clk);
  product <= full_product;
  control_section: process is
    -- variable declarations
    -- .....
  begin   -- control section
    -- sequential staments to assign values to control signals
    -- .....
    wait on Clk, Reset;
  end process control_section;
end architecture mixed;
```

Bild 3.9: Gemischtes Verhaltens- und Struktur-Modell aus [PA02]

Bei diesem Modell ist zu beachten, dass die Komponenten *shift_adder*, *reg* und *shift_reg* bereits definiert wurden und im Arbeitsbereich *work* untergebracht sind. Der Prozess *control_section* ist der verhaltensorientiert beschriebene Teil, der hier alle Kontrollsignale erzeugen soll.
Man ist also in der Lage, beliebig Prozesse und Komponenten als Entities miteinander zu verschalten und damit auch simulationstechnisch aneinander zu binden.

3.6 Testbenches für VHDL-Modelle

Wir haben oben eine Anzahl von VHDL-Modellen gesehen, aber wir wissen noch nicht, wie man diese simuliert. Von der bekannten Software in höheren Programmiersprachen wissen wir, dass sie zu compilieren und zu binden ist, um damit ablauffähigen Programmcode in der Zielmaschinensprache zu erzeugen. Dieser kann dann in den Speicher der Zielmaschine geladen werden. Aufgrund des Abarbeitungsmechanismus (von Neumann Zyklus) kann dann ein so in den Arbeitsspeicher der Zielmaschine gespeichertes Programm dadurch gestartet werden, dass der Programmzähler mit der Startadresse geladen wird. Ab da läuft dann das Programm automatisch ab, d.h. ohne dass der Benutzer die Details kennen

muss. Er betrachtet sein Programm als in seiner Programmiersprache Befehl für Befehl in der geplanten Reihenfolge ablaufend.
Denkt ein Hochsprachenprogrammierer an Nebenläufigkeiten, müssen der Hochsprache Elemente zur Beschreibung solcher Nebenläufigkeiten zugefügt werden, wie z.B. *Fork* und *Join*. Ohne hier in die Details zu gehen kann das vom Betriebssystem erfolgen oder Bestandteil der Hochsprache sein, wie es z.B. in Ada der Fall ist.
In all diesen Fällen geht Hochsprachen-Programmierer aber davon aus, dass die Programmteile Prozesse (in Ada Tasks) oder Threads stets permanent abgearbeitet werden und sich Prozesse zur Synchronisation gegenseitig anhalten müssen.

Da VHDL auf Hardware-Beschreibungen zielt, aber eine enge Sprachverwandtschaft zu Ada hat, muss das Modell simuliert werden können. Man kann sich in der "VHDL-Welt" vorstellen, dass sich im Gegesatz zu der oben angedeuteten "Selbstaktivität" im Ablauf “normaler“ Hochsprachen ein VHDL Modell eine bestimmte "Aktivierung" erhalten muss. Das gilt insbesondere für VHDL-Prozesse, die zwar in sich autonom laufen wie ein Hochsprachenprogramm, doch dazu erst zu aktivieren sind. Dazu hat ein VHDL-Prozess eine Signalliste ähnlich wie die Übergabeparameterliste von Unterprogrammen, auf die er “sensibel“ ist (sensitivity list), d.h. wenn sich an den Signalen etwas ändert, wird der Prozess “ablaufen“.

Die Aktivierungsregel lautet wie folgt:

Signalzuweisungen in VHDL werden aktiviert (ausgeführt),
- wenn die Simulation gestartet wird und
- immer dann, wenn sich auf der rechten Seite der Zuweisung etwas ändert.

Prozesse in VHDL werden aktiviert (ausgeführt),
- wenn die Simulation gestartet wird und
- wenn sich ein Signal in der Sensitätsliste ändert.

Insofern sind Signalzuweisungen und Prozesse direkt vergleichbar, wobei der Prozess im Prinzip die Verknüpfung der Signale auf der rechten Seite der Signalzuweisung durch einen komplexen, sequenziell ablaufenden Algorithmus bildet.
In VHDL kann man auch Warten in Abhängigkeit von einem Signalwechsel bis zur nächsten Bearbeitung der folgenden Befehle programmieren. Ein Prozess mit Sensitätsliste entspricht der Beschreibung eines Prozesses mit einem einzigen ***wait-on*** Statement auf die Signale der Sensitätsliste an seinem Ende. Beide Prozesse werden, wenn ihre Funktion ansonsten gleich ist, sich bei der Simulation bzgl. Der Zeit gleich verhalten. Das Problem ist, dass man nicht in der Lage ist, ein Wait auf Signale so einfach in eine Schaltung umzusetzen, zu synthetisieren.
Es sei jedoch bemerkt, dass Prozesse durchaus auch beschrieben werden können, in denen es mehr als ein *wait* und das auch auf unterschiedliche Signale geben kann, doch damit steigt die Komplexität der Funktionalität des Modells stark an. In dem oben angedeuteten Sinn zerlegt man solche Prozesse besser in kommunizierende Einzelprozesse, ohne an Allgemeingültigkeit einzubüßen. Wir werden noch später auf solche Problematiken eingehen.

Unsere Modelle, wie die Beispiele oben, müssen natürlich simuliert werden. Dazu sind sie also zur Aktivität anzuregen, d.h. es müssen wechselnde Signale als Testmuster angegeben werden.

Dazu werden im Folgenden drei Vorgehensweisen vorgeschlagen, die als Standard-Template dienen können.

Template 1 (direkte Testeinbettung in das Modell)

Die schnellste und einfachste Art, solche Aktivierungs- bzw. Testmuster anzugeben ist es, auf die Eingangssignale der Entity lokal in der Architektur Signalzuweisungen mit Delays zu programmieren. Das bedingt natürlich, dass in der Entity die Portangabe auskommentiert wird, da Eingangssignale logischerweise lokal nicht verändert werden dürfen. Wir hatten oben bereits Signalzuweisungen angewendet, hatten aber noch nicht dazu behandelt, dass diese mit einer Verzögerungszeit versehen werden können. Das wird im folgenden Abschnitt noch intensiv erläutert, wird aber hier bereits angewandt.

Bei unserem XOR Modell aus Bild 3.3 können wir wie in Bild 3.10 gezeigt verfahren.

```
entity xor_gate is
--  port (a, b : in bit;
--          c    : out bit);
end xor_gate;

architecture Specification of xor_gate is
  signal a, b: bit;           -- hinzufügen
begin
----- Testmuster -----
a <= '0',
       '1' after 20 ns,
       '0' after 40 ns,
       '1' after 60 ns;

  b <= '0',
       '0' after 20 ns,
       '1' after 40 ns,
       '1' after 60 ns;
----- Testmuster -----

  c <= a xor b;
end architecture Specification;
```

Bild 3.10: Testtemplate 1 für XOR-Gattermodell

Es hat sich in der Praxis durchgesetzt, VHDL Modelle wie reale Schaltungen in ein Testbett von Kontaktnadeln in eine eigene Testentity, sog. Testbench, einzubetten, die die Testmuster für das Modell enthält. Da die Eingangssignale sich zeitlich im Simulator wie Signalwellen auf einem Oszilloskop darstellen, spricht man hier auch von Waveforms bzw. Waveform-Generation.zur Erzeugung solcher Waveforms dienen die weiteren zwei Templates.

Template 2 (Testbench für Waveforms)

Die Testentity hat keine Portdeklaration und hat eine eigene Architektur, in der das zu testende Modell als sog. Device under Test (DUT) eingebettet ist. Eine Signalvereinbarung gibt die Eingänge des DUT vor, auf der dann die Waveforms wie im Template 1 programmiert werden. Als Beispiel nehmen wir wieder unser XOR-Beispiel. Da wir die XOR-Entity und die Testbench voneinander getrennt haben, muss die Testbench auf den Arbeitsbereich **work** zugreifen, um die XOR-Beschreibung zu nutzen. Die Einbindung in die Testbench ist in Bild 3.11 gezeigt.

```
entity test is
end test;

use work.all                         -- (1)

architecture behavior of TEST is
  signal a, b, c: bit;               -- (5)

component xorg is                    -- (2)
  port (a, b : in  bit;
        c    : out bit);
end component;
for all: xorg use entity work.xor_gate(Specification);    -- (3)
--for all: xorg use entity work.xor_gate(Algorithmic);
-- .....

begin
  a <= '0',
       '1' after 20 ns,
```

```
        '0' after 40 ns,
        '1' after 60 ns;

   b <= '0',
        '0' after 20 ns,
        '1' after 40 ns,
        '1' after 60 ns;

   DUT: xorg port map(a, b, c);          -- (4)

 end behavior;
```

Bild 3.11: Testbench Template 2 für ein XOR-Modelle

(1) Zunächst referiert man den Arbeitsbereich *work* und in diesem alle vorhandenen Entities mit *use work.all* . Der Arbeitsbereich ist ein sog. VHDL-Package, in dem sich mehrere Entities befinden können. Als Nächstes (2) spezifiziert man unter einem neuen Namen, hier *xorg* , die Anschluss-Beschreibung als *component* und ordnet dieser letztlich die Beschreibung *xor_gate* aus dem Workpackage zu (3). Hier erkennt man die Nutzung der unterschiedlichen Architekturen der Entity, die in Klammern eingefügt werden muss. Die Einbettung erfolgt mit (4), wobei der Name, hier DUT, zur Instanziierung einer Komponente **xorg** frei wählbar ist. Die Signale zur Ankopplung der Komponenteninstanz DUT müssen natürlich deklariert werden (5). Ihre Waveform zum Test des DUT ist dann wie bereits in Template 1 (Bild 3.10) beschrieben vorzunehmen.

Die oben angewandten Methoden zur Erzeugung von Signalverläufen stellen festgelegte, vorgegebene Waveforms dar. Will man periodische erzeugen, kann man sich für die Erzeugung einen Prozess nutzen. Das dritte hier vorgeschlagenen Template ist in Bild 3.12 angegeben.

```
 ... wie in Bild 3.11
 begin
   process is
     a <= '0';  b <= '0';
     wait for 20 ns;
     a <= '1';  b <= '0';
     wait for 20 ns;
     a <= '0';  b <= '1';
     wait for 20 ns;
     a <= '1';  b <= '1';
     wait for 20 ns;
   end process;

   DUT: xorg port map(a, b, c);          -- (4)

 end behavior;
```

Bild 3.12: Testbench Template 3 für ein XOR-Modelle

Diese Art der Testbench hat den Vorteil, dass man wie hier die Signalsetzungen zu einem Zeitpunkt zusammen setzt. Das ist in manchen Fällen überschaubarer. Diese Art der Stimulanzmuster-Erzeugung führt zudem zu periodischen Wiederholungen der Waveform. Den Hintergrund dazu wollen wir im folgenden Abschnitt näher beleuchten.