INSTITUT FÜR
DATENVERARBEITUNGSANLAGEN

Praktikum

Rechnergestützter Schaltungsentwurf

Kurzeinführung in VHDL

Elmar Maas 21. April 1999

Inhalt

1 Einleitung

Diese Kurzeinführung soll Ihnen einen möglichst schnellen Einstieg in die Sprache VHDL ermöglichen. Sie erhebt weder Anspruch auf Vollständigkeit, noch kann sie die Lektüre von weiterführender Literatur ersetzen. Es seien hier insbesondere die Werke von Cohen [1] und Bhasker [2] sowie die IEEE-Norm [3] und das Synopsys VHDL-Handbuch [4] empfohlen.[1] Alle Synopsys Handbücher stehen in Form von Online-Dokumentationen zur Verfügung. Das Werkzeug zum Ansehen wird mit dem Kommando *iview &* in der Kommandozeile einer UNIX-Shell gestartet.

In diesem Skript streng zwischen synthetisierbaren VHDL-Konstrukten und solchen Konstrukten, die lediglich zur Simulation geeignet sind, unterschieden. Ein anschauliches Beispiel: In der Simulation ist es notwendig sagen zu können, daß nach einer bestimmten Zeitspanne ein bestimmtes Ereignis eintreten soll, beispielsweise um den Systemtakt zu simulieren oder Stimuli zu applizieren. Bei wirklich erzeugter Hardware wird es aber unmöglich sein, genau vorgegebene Zeiten einzuhalten; es macht keinen Sinn zu sagen, ein bestimmtes Signal soll nach genau 3 Nanosekunden am Ausgang sein, sondern es macht lediglich Sinn zu sagen, es soll *höchstens* beispielsweise 5 Nanosekunden benötigen. Außer in [1] und [4] wird in der Literatur diese Unterscheidung oft nicht gemacht, was einen nichtsahnenden VHDL-Einsteiger schon ein bißchen in Verzweiflung bringen kann, wenn sich seine Beschreibungen plötzlich nicht übersetzen lassen – deshalb hier die klare Trennung. Der nächste Abschnitt erklärt Ihnen kurz die Entstehung von VHDL und seine Berechtigung und dann geht es richtig los...

2 Zur Sprache VHDL

Der Name **VHDL** ist ein Akronym für **V**HSIC (**V**ery high speed integrated circuit) **H**ardware **D**escription **L**anguage. Im Rahmen des VHSIC Programms entstand im Jahre 1981 im amerikanischen Verteidigungsministerium der Wunsch nach einer einheitlichen Beschreibungs- und Dokumentationssprache für digitale Schaltungen und Systeme. Diese Sprache sollte die Kommunikation der unterschiedlichen an dem Projekt beteiligten Firmen vereinfachen und vereinheitlichen. Die konkrete Entwicklung begann 1983 durch die Firmen IBM, Texas Instruments und Intermetrics. Zwei Jahre später kam es zur ersten Veröffentlichung von VHDL und weitere zwei Jahre später wurde sie zur IEEE-Norm 1076-1987[3].

Entsprechend der erwähnten Zielsetzung ist VHDL eine Hardware-Beschreibungssprache zur Beschreibung digitaler Systeme. Sie hat viele Eigenschaften von der ebenfalls im Auftrag des amerikanischen Verteidigungsministeriums entwickelten Programmiersprache *Ada* geerbt. Als sowohl von Menschen als auch von Maschinen lesbare Hochsprache zur Generierung von Hardware stellt sie ein für den Hardware–Entwickler sehr angenehmes Kommunikationsmedium mit dem Rechner dar. Ebenso vereinfacht sie die Kommunikation von Entwicklern untereinander sowie die Dokumentation ganzer Projekte.

VHDL unterstützt Hierarchiestrukturen, d.h. sie ermöglicht den Entwurf und die Synthese ganzer Systeme und deren Untersysteme einschließlich der kleinsten Einzelkomponenten. Hierbei ist es sowohl möglich, auf oberster Systemebene zu beginnen und dann daraus die Untersysteme zu entwickeln (*Top-Down-Methode*), als auch aus den kleinsten Einzelkomponenten das Gesamtsystem zusammenzusetzen (*Bottom-Up Methode*). Es besteht die Möglichkeit, digitale Schaltungen auf verschiedenen Abstraktionsebenen zu beschreiben, von Systembeschreibungen über

[1] Diese Werke stehen im Institut zur Verfügung, Sie brauchen sie also keinesfalls zu kaufen (sehr teuer). Aber bitte lassen Sie sie auch dort (auf keinen Fall mit nach Hause nehmen!!!), denn andere sind auch dringend darauf angewiesen!

Register-Transfer-Beschreibungen bis hin zu Beschreibungen auf Logik-Gatterebene. Unterhalb der Gatterebene (Transitorebene und darunter) läßt sich mit Hilfe bestimmter Hilfsmodelle begrenzt arbeiten (46-level model), aber dieser Bereich gehört nicht zur eigentlichen Domäne der Sprache.

ANMERKUNG:
Der in diesem Praktikum entstehende ASIC is sinnvollerweise im wesentlichen auf der Register-Transfer-Ebene zu implementieren, einem Bereich, in dem die Sprache ihre Vorteile hervorragend ausspielen kann.

Die eben genannten Eigenschaften von VHDL setzen Sprachelemente zur *Strukturbeschreibung*, zur *Verhaltensbeschreibung* und zur *Datenflußbeschreibung* voraus. Diese Beschreibungsarten lassen sich auch beliebig vermischen. Die Strukturmethode eignet sich ideal für die Zusammensetzung eines Systems aus Einzelkomponenten, da sie die einfache Verbindung von Elementen gestattet. Die Verhaltensbeschreibung wird zur Implementierung von bedingten Abläufen, z.B. in Steuerwerken, eingesetzt. Die Datenflußbeschreibung findet dort ihren Platz, wo es um Zuweisung von Signalen, Registerinhalten u.ä. sowie deren logische Verknüpfung geht. Jede Beschreibungsart hat ihren eigenen Anwendungsbereich.[2]

VHDL ermöglicht die Darstellung von sequentiellen wie auch parallelen Abläufen und damit auch die Verwirklichung von sowohl asynchronen als auch synchronen Modellen. Trotz der Technologieunabhängigkeit der Sprache können bei Bedarf technologieabhängige Konstrukte aus Bibliotheken der ASIC-Anbieter sowohl in die Synthese als auch in die Simulation von digitalen Schaltungen eingebunden werden.

Die Simulation gehört zu den ganz großen Vorteilen von VHDL. Es lassen sich nicht nur nahezu beliebig große Designs synthetisieren, sondern auch auf allen Hierarchieebenen simulieren. Bei herkömmlichen Schematic-Entry-Tools, die auf Gatterebene simulieren, erweist sich die Simulation eines Gesamtsystems auch bei durchaus üppigen Hardwareressourcen leicht als nicht durchführbar: Die Simulationszeiten werden sehr schnell inakzeptabel. Komplexe Simulationssprachen lassen die Simulationsprogramme schnell unübersichtlich und damit auch unsicher werden. Zur Simulation von VHDL-Modellen braucht der Entwickler keine neue Sprache zu erlernen; VHDL ist gleichzeitig Synthese- und Simulationssprache. In der Sprachdefinition wurde eine Simulationssemantik für den gesamten synthetisierbaren Sprachanteil bereitgestellt. Daraus ergibt sich, daß der synthetisierbare Sprachanteil eine Untermenge des gesamten Sprachumfangs darstellt.

Da VHDL selbst technologieunabhängig ist, lassen sich Modelle unabhängig von der konkreten Hardwareumsetzung beschreiben und simulieren. Anschließend sind sie leicht auf eine beliebige Zieltechnologie portierbar.
In Tabelle 1 sind die Vor- und Nachteile von Schematic-Entry-Tools respektive VHDL-Synthesetools bei der Schaltungsentwicklung gegenübergestellt.

Überblickt man die gravierenden Vorteile von VHDL gegenüber herkömmlichen Schematic-Entry-Tools, so stellt sich die Frage, warum überhaupt noch jemand letztere benutzt. Hierfür mögen zwei Gründe entscheidend sein:

1. Erfahrene Designer erreichen mit herkömmlichen Tools im allgemeinen bessere Ergebnisse bezüglich Timing und Fläche als es mit VHDL möglich ist. Die Begründung hierfür liegt in den *heuristischen Optimierungsmethoden* von automatisierten Synthesetools [5]. Somit wird dieser Punkt noch auf absehbare Zeit sein Gewicht behalten.

[2] Während des Praktikums werden Sie reichlich Beispiele für den sinnvollen Einsatz jeder der drei Arten entdecken.

	Schematic-Entry-Tools		VHDL-Tools
⊕	sehr gute Ergebnisse bei erfahrenen Designern	⊖	i.A. schlechtere Ergebnisse als bei erfahrenen Designern
⊖	relativ schlechte Ergebnisse bei unerfahrenen Designern	⊕	relativ gute Ergebnisse auch ohne große Erfahrung erzielbar
⊖	Simulation sehr aufwendig und für größere Schaltungen kaum noch durchführbar	⊕	Simulation auch für große Schaltungen praktikabel
⊕	in vielen Industriebetrieben geeignetes Werkzeug bereits vorhanden	⊖	sehr neues Werkzeug, hohe (Neu–)Anschaffungskosten
⊕	Mitarbeiter häufig schon geschult	⊖	Erlernen der Sprache und des Umgangs mit VHDL-Tools erforderlich
⊖	rel. lange Entwicklungszeiten	⊕	kurze Entwicklungszeiten
⊖	schlechte Änderbarkeit	⊕	leicht änderbar
⊖	hoher Dokumentationsaufwand	⊕	selbstdokumentierend
⊖	technologieabhängig, nur mit sehr hohem Aufwand portierbar	⊕	weitgehend technologieunabhängig, leicht portierbar
⊕	optimale Ausnutzung der jeweiligen Technologievorteile möglich	⊙	Nutzung von Technologiemakros möglich, wobei aber die Portierbarkeit eingeschränkt wird
		⊙	sehr großer Sprachumfang, es reicht ein kleiner Teil davon zur Synthese

Tabelle 1: Vor- und Nachteile der Schaltungsentwicklung mit Schematic-Entry- bzw. VHDL-Tools

2. In der Industrie sind herkömmliche CAD-Tools bereits im Einsatz, das bedeutet, es wurden Investitionen getätigt und Mitarbeiter geschult. Die enorm hohen Anschaffungskosten der noch sehr neuen VHDL-Werkzeuge und der entsprechenden Hardwareressourcen sowie die unumgängliche Schulung der Mitarbeiter bilden für viele Unternehmen ein übergroßes Gegengewicht zu den Vorteilen von VHDL. Sicherlich ist es nur eine Frage der Zeit, bis sich dieser Punkt in sein Gegenteil verkehrt. Eine Sprache wie VHDL ist bedeutend leichter zu erlernen als der Umgang mit herkömmlichen CAD–Werkzeugen und den dazugehörigen Simulationssprachen. Dadurch wird die Schulung neuer Entwickler effektiver und kostengünstiger. Die Hardware ist einem laufenden “Aufrüstungsprozeß“ hin zu immer leistungsfähigeren Systemen unterworfen, so daß hierin kein langfristiges Hindernis gesehen werden muß. Somit ist mit dem Wechsel der Designer-Generation bzw. der Steigerung der Hardwareperformance eine Verlagerung der Schwerpunkte von herkömmlichen CAD-Tools hin zu VHDL-Tools zu erwarten.

3 Sprachelemente zur Synthese

In diesem Abschnitt werden die wesentlichen Sprachelemente zur Synthese kurz erläutert. Alle synthetisierbaren Sprachelemente sind auch simulierbar, d.h. "VHDL zur Simulation" ist eine Obermenge von "VHDL zur Synthese". Beide sind jedoch eine Untermenge des im VHDL-Standard [3] festgelegten Sprachumfangs.

3.1 Aufbau eines VHDL Programmes

Ein sinnvolles VHDL-Programm besteht zumindest aus einer *Entity* und einer *Architecture* .

3.1.1 Entity

Suchen Sie nicht nach einem deutschen Wort dafür, Sie werden keines finden außer vielleicht Entität, aber wir wollen nicht albern sein. Unter einer Entity können Sie sich eine Blackbox mit genau definierten Ein- und Ausgängen vorstellen. Es wird hierin die *Schnittstelle* eines Designs mit seiner Umwelt definiert, also z.B. Datentypen der Ports, sowie die Richtung (Ein- oder Ausgang). Über das, was im Innern des Designs vorgeht, ist an der Entity nichts zu sehen. Der Quelltext für eine Halbaddierer Entity könnte wie folgt aussehen:

```
ENTITY half_adder IS
  PORT ( clk,    -- das Taktsignal
         I1   : in     BIT;
         I2   : inout  BIT; -- bidirektionaler Port
         Cin  : in     BIT;
         O    : out    BIT;
         Cout : buffer BIT); -- auch intern lesbarer Port
END half_adder;
```

Aus den angeführten Beispielen soll immer möglichst viel erlernbar sein, deshalb werden oft auch Dinge auftreten, die für das gewählte Design vielleicht gar nicht in der Weise erforderlich sind; seien Sie da bitte nicht pedantisch, sondern dankbar, daß Sie nicht mit überflüssigem Text überschüttet werden.

3.1.1.1 Kommentare

An dem gegebenen Beispiel sehen Sie, daß *Kommentare* durch einen doppelten Bindestrich **- -** eingeleitet werden. Kommentare werden nicht mit übersetzt.[3]

3.1.1.2 Bezeichner

Für Bezeicher gelten in VHDL folgende Konventionen: Groß- und Kleinbuchstaben werden nicht unterschieden, d.h. Half_Adder und half_adder bezeichnen dasselbe Objekt. Versuchen Sie bitte trotzdem, für ein und denselben Bezeichner dieselbe Schreibweise zu verwenden. Am Anfang eines Bezeichners muß ein Buchstabe stehen. Innerhalb und am Ende des Bezeichners sind auch Ziffern und Unterstriche _ erlaubt; es dürfen jedoch nicht zwei Unterstriche unmittelbar aufeinander folgen.

[3] Es gibt da einige Ausnahmen, wenn nämlich in dem Kommentar eine Compilerdirektive steht. Solche Direktiven werden durch das Wort "pragma" eingeleitet. Ebenso ist das Wort "synopsys" die Einleitung einer Compilerdirektive. Da diese Direktiven erst bei speziellen Wünschen erforderlich sind, werden sie hier nicht weiter erläutert.

3.1.1.3 Portrichtungen

Ein Eingangsport erhält die Richtungsbezeichnung **in** und ein Ausgangsport entsprechend **out**. Ein bidirektionaler Port erhält den Richtungsbezeichner **inout** . In einen solchen Port dürfen sowohl Signale hineingeschreiben als auch daraus herausgelesen werden. Interessant ist der Richtungsbezeichner **buffer**: Ein Port mit diesem Attribut darf *innerhalb* der Entity nicht nur beschrieben, sondern auch wieder gelesen werden. Es handelt sich immer um einen Ausgangsport.

3.1.1.4 Sonstiges

Gibt es in einer Entity mehrere Eingänge mit demselben Typ, so können diese mit Hilfe von Kommas getrennt deklariert werden und es braucht nicht jeweils der Richtungsbezeichner und der Typ mit angeführt zu werden, sondern lediglich beim letzten. Dasselbe gilt natürlich für Ausgänge. Auf die verschiedenen Datentypen wird später noch genauer eingegangen.

3.1.2 Architecture

Bei der Entity wurde auf eine Eindeutschung verzichtet, also auch hier, wenn auch das Wort eher dazu reizt. In einer Architecture wird das Verhalten einer Entity beschrieben, also das, was wirklich passiert. Dies kann auf verschiedene Weisen geschehen: Man kann in Form einer Verhaltensbeschreibung (ähnlich herkömmlichen Programmiersprachen) vorgehen oder den Datenfluß beschreiben, also angeben, welche Leitung mit welcher wie verknüpft wird. Eine Architecture läßt sich aber auch aus mehreren Entities zusammensetzen, deren Ein- und Ausgänge verknüpft werden; das nennt man dann Strukturbeschreibung. Sie müssen das nicht so genau unterscheiden, denn Sie dürfen (zumindest mit den Synopsys-Tools) beliebig mischen, aber sie müssen noch ein wenig mehr dazu wissen, doch dazu später. Abbildung 1 zeigt die Syntax eines Architecture Rumpfes (architecture body). Einige der auftretenden Begriffe werden erst im weiteren Verlauf des Skriptes erläutert. Leider lassen sich diese Vorabdeklarationen an einigen Stellen des Skriptes nicht vermeiden, da ein möglichst kompakter Aufbau gewünscht war.

architecture *Architecture-Name* **of** *Entity-Name* **is**
    [*Deklarationen*]
begin
    Parallele Anweisungen (concurrent statements); im einzelnen:
        Prozesse
        Parallele Signalzuweisungen
        Komponenten-Instanzierungen
        einige andere, für die Synthese nicht relevante Konstrukte
end [*Architecture-Name*];

Abbildung 1: Syntax eines Architecture Rumpfes

```
ARCHITECTURE half_adder_arch IS
  hier stehen dann Signaldeklarationen, Prozesse, Concurrent-Statements
  (die Beschreibung dieser erfolgt in spaeteren Abschnitten)
END half_adder_arch;
```

3.2 Sprachelemente

3.2.1 Signale und Variablen

Variablen kennen Sie aus höheren Programmiersprachen. Allerdings gibt es in VHDL nur lokale Variablen. Signale sind immer global. Warum Signale nicht einfach globale Variablen heißen, soll jetzt kurz erläutert werden. Programmiersprachen wie PASCAL oder C/C++ sind grundsätzlich sequentielle Sprachen, d.h. es gibt keine Möglichkeit zu sagen, daß bestimmte Anweisungen parallel ausgeführt werden sollen. Bei herkömmlichen Prozessoren macht das ja auch keinen Sinn, denn sie arbeiten nur sequentiell. Mit VHDL beschreiben Sie jedoch Hardware, d.h. es ist durchaus möglich, mehrere Operationen parallel auszuführen. Deshalb muß die Sprache auch Möglichkeiten bieten, um dies zu beschreiben. Einfach gesprochen gibt es in VHDL Prozesse, Prozeduren, Funktionen und Concurrent Statements.[4] Insbesondere mit den Begriffen Prozeß und Concurrent Statement werden Sie nicht viel anfangen können, aber darauf wird später noch genauer eingegangen. Nur soviel: Concurrent[5] Statements sind alleinstehende Einzelanweisungen, die alle parallel ausgeführt werden. Prozesse sind den Ihnen bekannten Prozeduren ähnlich, nur daß sie nicht durch einen expliziten Aufruf aus einer anderen Prozedur heraus aufgerufen werden, sondern auf vorgegebene Ereignisse auf Signalen reagieren. Alle Prozesse und Concurrent Statements laufen grundsätzlich parallel. Innerhalb eines Prozesses werden die Anweisungen sequentiell abgearbeitet. Variablen können Sie nur innerhalb von Prozessen deklarieren und verwenden (lokal). Um Informationen zwischen verschiedenen Prozessen und mit Concurrent-Statements austauschen zu können, werden *Signale* benötigt. Man kann sich diese als Leitungen vorstellen, die parallel arbeitende Black-Boxes miteinander verbinden. Die Ports einer Entity sind automatisch als Signale bekannt. Für Signale und Variablen existieren unterschiedliche Zuweisungsoperatoren. Bsp.:

```
SIGNAL sig1;
PROCESS
  VARIABLE var1;
BEGIN
  var1 := sig1;
  sig1 <= var1;
END PROCESS;
```

Der Zuweisungsoperator hängt vom Ziel ab. Handelt es sich dabei um ein Signal, muß $\boxed{<=}$ verwendet werden. Bei einer Variablen sieht der Zuweisungsoperator so aus: $\boxed{:=}$.
Wichtig im Zusammenhang mit Signalen ist noch folgendes: Da Signale in allen Prozessen bekannt sind, muß eindeutig festgelegt werden, welcher Prozeß den Wert eines Signales verändern, also darauf schreiben darf. Versuchen beispielsweise zwei unterschiedliche Prozesse unterschiedliche Werte zur gleichen Zeit auf ein Signal zu schreiben, führt dies zu einer Mehrdeutigkeit, die aufgelöst werden muß. Zu diesem Zweck gibt es die Möglichkeit, sogenannte *Resolution Functions* mit einem Signal zu verknüpfen. Darin wird dann festgelegt, welche logische Verknüpfung mehrerer Treiber den wirklichen Wert der Variablen bestimmen soll (z.B. logisch ODER). Im allgemeinen sollten solche *Konfliktlösungsfunktionen* jedoch vermieden werden, denn Sie können zu sehr unübersichtlichem Schaltungsverhalten führen, wenn wider Erwarten doch einmal zwei Treiber gleichzeitig auf eine Variable schreiben und nur in diesen nicht erwarteten Sonderfällen

[4] Prozeduren und Funktionen machen in der Synthese mit Synopsys keinen Sinn, da sie in einem Taktschritt abgearbeitet werden müssen (es sind in ihnen keine Wait-Statements erlaubt), was die Möglichkeiten mit diesen sehr einschränkt. Sie werden deshalb in diesem Skript nicht erläutert.

[5] concurrent heißt soviel wie parallel

die Schaltung mit (völlig) falschen Werten reagiert.[6] Aus diesem Grunde wird der Aufbau von Konfliktlösungsfunktionen in diesem Skript nicht erläutert. Sie vermeiden Konfliktlösungsfunktionen, indem Sie ein Signal nur von einem Prozeß treiben lassen. Wenn unbedingt ein anderer Prozeß darauf zugreifen soll, dann lassen Sie Ihn mit dem Treiberprozeß kommunizieren.

3.2.2 Datentypen

In der *Sprachdefinition* von VHDL gibt es alle Datentypen, die zu einer modernen Hochsprache dazugehören, einschließlich Fließkommazahlen, Zeiger und Records und sogar physikalischer Typen, wie z.B. Zeit, Stromstärke und Spannung. Alle diese Datentypen in Hardware umzusetzen ist schwierig, in einigen Fällen, zumindest zur Zeit noch, unmöglich. Das bedeutet, ein VHDL-Compiler wird nur ein Subset der definierten Sprache übersetzen. Dies gilt insbesondere für Compiler zur Synthese. An dieser Stelle wird auf die Datentypen eingegangen, die auch der Synopsys Synthese Compiler (Design Compiler) unterstützt.

3.2.2.1 Skalare Datentypen

Zu den skalaren Datentypen zählen alle INTEGER-Typen und alle Aufzählungstypen, nicht jedoch Feldtypen (Arrays, Records) und Zeigertypen.

3.2.2.1.1 Der Datentyp BIT Der einfachste vorstellbare Datentyp ist ein einzelnes Bit, welches den Wert Null oder Eins annehmen kann. In Wirklichkeit ist ein Bit ein Aufzählungstyp der folgendermaßen definiert ist:

type BIT **is** ('0','1');

Dieser Datentyp steht Ihnen standardmäßig zur Verfügung, ohne daß spezielle Bibliotheken eingebunden werden müssen. Nun reichen für die eindeutige Beschreibung eines realen Signals die Werte Null und Eins nicht aus. Es interessiert z.B. weiterhin, ob ein Signal vielleicht undefiniert ist (U), ob es sich im Hochimpedanzzustand befindet (Z) oder ob die Synthese wissen soll, daß der Zustand eines Signales zu einem bestimmten Zeitpunkt beliebig sein darf (D), um eine bessere Optimierung durchführen zu können. Die Firma SYNOPSYS hat zu diesem Zweck eine Standard Logic Bibliothek entworfen, die nicht nur die entsprechenden Datentypen bereitstellt, sondern auch die Standardoperatoren darauf definiert. Sie machen sich diese Bibliothek zugänglich, indem Sie innerhalb Ihrer VHDL-Datei (der Übersichtlichkeit halber empfohlenermaßen am Anfang) eine Bibliothek deklarieren:

```
library IEEE;
```

und dann festlegen, welchen Teil der Bibliothek Sie verwenden möchten:

```
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
```

Die erste use-Anweisung macht die std_logic-Datentypen zugänglich, die zweite die Operatoren darauf. Der Typ, der sinnvollerweise statt des Typs BIT zu verwenden, ist heißt **std_logic**.

```
signal slb1 : std_logic;
signal slb2 : std_logic;

⋮
```

[6] Verschonen Sie Ihren Betreuer mit dem Gejammere, daß es doch eigentlich funktionieren müsste. Wenn Sie mit Resolution-Functions arbeiten, müssen Sie auch die Fehler selber finden...

```
slb1 <= '1';
slb2 <= slb1;
```

Bitte beachten Sie die einzelnen Anführungszeichen bei Zuweisungen von Konstanten. (Eine 1 ohne diese Zeichen würde dem Datentyp Integer zugeordnet und deshalb zu einem Fehler bei der Typüberprüfung führen.)

3.2.2.1.2 Der Datentyp BITVECTOR Oft ist es erforderlich, mehrere Bits zu einem Bus zusammenzufassen. Hierfür existiert der Datentyp BitVector. Wie bei einzelnen Bits existiert hier ein erweiterter Standard Logic Datentyp, der *std_logic_vector*. Die Deklaration eines Vektors aus 8 Bit geschieht z.B. folgendermaßen:

`signal myBus :` **std_logic_vector**`(7 downto 0)`

Hier ist das Bit mit der Nummer 7 das MSB und das Bit mit der Nummer 0 das LSB. Denkbar wäre auch die umgekehrte Notation:

`signal myBus :` **std_logic_vector**`(0 to 7)`

Hier wäre das Bit mit der Nummer 0 das MSB und das Bit mit der Nummer 7 das LSB. Im Prinzip kann das jeder handhaben wie er will. Trotzdem möchte ich vor einem "Mixen" warnen, da es zu unangenehmen Verdrehungen führen kann. Beachten Sie bitte die VHDL-Schlüsselwörter *downto* und *to*, die bei Bereichsangaben eingesetzt werden. Es ist auch die Zuweisung von einzelnen Bits und Bereichen möglich:

```
signal one_Bit: std_logic;
signal myBus  : std_logic_vector(7 downto 0);
signal bigBus : std_logic_vector(40 downto 3);

one_Bit          <= bigBus(3);
myBus(3 downto 0) <= bigBus(40 downto 37);
bigBus(7 downto 4) <= "0110";
```

Bits und Bitvektoren sind die Datentypen, mit denen Sie in mindestens 80 Prozent der Fälle im Eisenbahnpraktikum auskommen. Die anderen Datentypen sind jedoch keinesfalls unwichtig, da sie Schaltungsbeschreibungen auf höheren Ebenen ermöglichen. Im Rahmen des Praktikums wird empfohlen, diese Datentypen für **Datenbusse** *zu verwenden, da man hier alle Freiheiten hat, was Bedeutung bzw. Wertigkeit der einzelnen Bits anbelangt.*

3.2.2.1.3 Die Typen UNSIGNED und SIGNED Die Datentypen UNSIGNED und SIGNED gehören nicht zum Sprachumfang von VHDL sind jedoch fester Bestandteil der IEEE-Library. Sie bezeichnen Vektoren aus Elementen vom Typ std_logic und sollten überall da eingesetzt werden, wo es auf eine korrekte Behandlung des Vorzeichens ankommt (z.B. arithmetrische Funktionen).
Obwohl der Definiton dieser Typen ein std_logic_vector zu grunde liegt, ist eine direkte Zuweisung eines solchen Vektors an einen UNSIGNED oder umgekehrt nicht möglich. Es existiert jedoch eine Umwandlungsfunktionen, die ein UNSIGNED in ein std_logic_vector konvertieren kann.

```
signal std_vector: std_logic_vector(3 downto 0);
signal unsig_vector: unsigned(3 downto 0);

std_vector <= conv_std_logic_vector(unsig_vector,4);
```

Die folgende Tabelle gibt einen Überblick über die unterschiedlichen Ergebnisse, die bei der Anwendung von Vergleichsfunktionen auf diese drei Datentypen, auftreten. Bei den Typen SIGNED und UNSIGNED ist das Ergebnis unabhängig von der Länge des Vektors. Es wird nur der reine Zahlenwert zur Berechnung herangezogen. In dem Fall ist es egal, ob die Zahl 1 durch einen 8-Bit breiten Vektor oder durch einen 4-Bit breiten Vektor dargestellt wird. Das Ergebnis ist in beiden Fällen identisch. Beim Bitvektor ist das nicht der Fall.
Diesen Unterschied sollte man bei der gleichzeitigen Verwendung der Datentypen nicht aus den Augen verlieren.

ARG1	op	ARG2	UNSIGNED	SIGNED	BIT_VECTOR
“000“	=	“000“	TRUE	TRUE	TRUE
“00“	=	“000“	TRUE	TRUE	FALSE
“100“	=	“0100“	TRUE	FALSE	FALSE
“000“	<	“000“	FALSE	FALSE	FALSE
“00“	<	“000“	FALSE	FALSE	TRUE
“100“	<	“0100“	FALSE	TRUE	FALSE

Der Datentyp UNSIGNED wird für **Adressbusse** *empfohlen. Bei solchen Bussen ist es erforderlich, Adressen inkrementieren oder eine Konstante aufaddieren zu können. Für UNSIGNED existieren Routinen, die solche Operationen unterstützen - für BIT_VECTOR ist dies nur eingeschränkt möglich, da dem Compiler ja prinzipiell die Position von MSB und LSB nicht bekannt ist, er also bei einem Inkrement nicht wissen kann, ob er eine von '1' links oder rechts hinzuaddieren soll.*

3.2.2.1.4 Der Typ BOOLEAN Ein dem Bit sehr verwandter Datentyp ist der Aufzählungstyp BOOLEAN. Er ist folgendermaßen deklariert:

type BOOLEAN **is** (FALSE, TRUE);

Variablen dieses Datentyps werden Sie selten deklarieren müssen, es ist jedoch wichtig zu wissen, daß Ergebnisse von Vergleichen automatisch den Datentyp BOOLEAN besitzen und Kontrollstrukturen auch diesen Datentyp erwarten. Bsp.:
Es gelten folgende Deklarationen:

```
signal myBool : BOOLEAN;
signal myBit  : std_logic;
signal a      : std_logic_vector(7 downto 0);
signal b      : std_logic_vector(7 downto 0);
signal z      : std_logic_vector(7 downto 0);
```

Das Konstrukt:

```
if (a < b ) then
  z <= a;
```

könnte ersetzt werden durch:

```
myBool <= a < b;
if (myBool) then
  z <= a;
```

<u>nicht</u> aber durch:

```
myBit <= a < b;         -- FALSCH!!! (Ergebnis von Vergleich ist BOOLEAN)
if (myBit) then         -- FALSCH!!! (Kontrollstrukturen erwarten BOOLEAN)
  z <= a;
```

3.2.2.1.5 Aufzählungstypen Wie bereits erwähnt, handelt es sich bei den Typen Bit und Boolean um vordefinierte Aufzählungstypen. Sie wurden auf Grund ihrer herausragenden Rolle im Praktikum einzeln aufgeführt. Sie können sich jedoch auch beliebige andere Auszählungstypen selbst definieren. Bsp.:

type MICRO_OP **is** (LOAD,STORE,ADD,SUB,MUL,DIV);
subtype ARITH_OP **is** MICRO_OP **range** ADD **to** DIV;

Selbst definierte Aufzählungstypen können z.B. sinnvoll sein, um die Zustände einer Finite State Machine zu beschreiben. Beachten Sie bitte, daß außer Zuweisungen keine Operatoren auf selbstdefinierten Datentypen definiert sind. Wenn also welche benötigt werden, müssen Sie sie definieren. Das Operator-Overloading hier zu beschreiben, würde zu weit führen, es sei aber erwähnt, daß ein Operator-Overloading nicht nur möglich sondern auch praktikabel ist.

3.2.2.1.6 Integertypen Der Datentyp INTEGER ist in der Sprache VHDL vordefiniert. Der Bereich eines INTEGER ist implementierungsabhängig. Mindestens muß er jedoch Werte von $-2^{31}-1$ bis $+2^{31}-1$ umfassen, was 32 Bit entspricht. Dies ist auch genau der von der Firma SYNOPSYS gewählte Bereich für einen Integer. Sie können sich beliebige eigene Integertypen definieren, solange der angegebene Bereich kleiner gleich dem des Typs INTEGER ist. Bsp.:

type Spieler **is range** 1 to 11;
type Bitnummer **is range** 0 to 31;

Durch die Bereichsangabe wird *indirekt* die Bitbreite von Variablen dieses Typs festgelegt. Eine Variable des Typs Spieler würde bespielsweise 4 Bit breit sein, aber es dürften trotzdem nur Werte im Bereich von 1 bis 11 an sie zugewiesen werden, also nicht etwa 15! Eine Variable des Typs Bitnummer hätte eine Breite von 5 Bit und nicht etwa 32! Integer-Variablen dürfen einander zugewiesen werden, solange das Quellobjekt einen kleineren Bereich umfaßt als das Zielobjekt.

```
signal myInt : INTEGER;
signal Fritz : Spieler;
signal Letzter : Bitnummer;
signal myVect  : std_logic_vector(4 downto 0);

⋮

myInt   <= 137;
Fritz   <= 5;
Letzter <= 3;
Letzter <= Fritz; -- erlaubt
Fritz   <= Letzter; -- VERBOTEN, da Zieldatentyp kleiner als Quelldatentyp
myInt   <= Letzter; -- erlaubt
myVect(4 downto 0) <= Letzter; -- trotz gleicher Bitbreite VERBOTEN,
                               -- da unterschiedliche Typen
```

Bitvektoren und INTEGER-Typen sind völlig unterschiedliche Datentypen, deshalb ist eine direkte Zuweisung *nicht* erlaubt. Dies liegt daran, daß solche Zuweisungen nicht eindeutig wären: Soll z.B. das MSB als Vorzeichenbit gewertet werden oder nicht? SYNOPSYS stellt in der Arithmetik-Bibliothek Umwandlungsfunktionen zur Verfügung, mit deren Hilfe man STD_LOGIC_VECTOR-en in INTEGER-Typen umwandeln kann und umgekehrt:

CONV_STD_LOGIC_VECTOR(*Integervariable, Bitbreite des Ziels*)
CONV_INTEGER(*Bitvektorvariable*);

Beispiel:

```
myVect <= CONV_STD_LOCIC_VECTOR(Letzter, 5);
Letzter <= CONV_INTEGER(myVect);
```

Die erste Variante kann man dazu "mißbrauchen", sich Schreibarbeit und Zählfehler zu ersparen:

```
signal a : std_logic_vector(31 downto 0);
a <= "00000000000000000000000000000000";
-- kann auch geschrieben werden als:
a <= CONV_STD_STD_LOGIC_VECTOR(0, 32);
```

3.2.2.2 Feldtypen

Es ist möglich, Felder aus bereits deklarierten Typen aufzubauen. Solche Felder heißen Arrays. Es sind auch mehrdimensionale Arrays möglich. Bsp.:

type RAM **is array** (31 downto 0) of std_logic_vector(7 downto 0);
type Matrix **is array** (31 downto 0, 7 downto 0) of std_logic;

```
signal RAM_simple : RAM;
signal RAM_matrix : Matrix;
```

Im Prinzip bezeichnen beide Typdeklarationen das gleiche Feld. Allerdings unterscheidet sich die Indizierung: Um auf das MSB des zweiten Vektors in der Variablen (Signal) RAM_simple zuzugreifen, ist zu schreiben: RAM_simple(2)(7), aber um auf das gleiche Element im Signal RAM_matrix zuzugreifen, muß RAM_matrix(2,7) geschrieben werden.

An dieser Stelle sei ausdrücklich darauf hingewiesen, daß die Implementierung von RAM's oder ROM's in VHDL außer bei sehr kleinen Feldern unökonomisch, d.h. teuer ist. Jeder Hersteller bietet spezielle RAM's und ROM's zu seinen ASIC's an. Diese sollten unbedingt genutzt werden, auch wenn man hierfür die jeweilige Beschreibungsweise des Herstellers erlernen muß. Das verhältnismäßig kleine ROM auf dem im Praktikum zu erstellenden Eisenbahnchip nimmt bereits mehr als ein Drittel der Chipfläche ein, wenn es in VHDL implementiert wird!

3.2.2.3 Weitere Typen ... gibt es diverse. Wenn Sie daran interessiert sind, lesen Sie bitte in [2] und [3] nach. Beachten Sie aber, daß andere als die in diesem Skript erwähnten Datentypen *nicht* für die **Synthese** mit SYNOPSYS zur Verfügung stehen, d.h. zur Zeit unbrauchbar sind.[7]

[7] Einige Datentypen sind für die Simulation von größerer Bedeutung. Sie werden an entsprechender Stelle erläutert.

3.2.3 Operatoren

Die vordefinierten Operatoren in VHDL lassen sich in fünf Gruppen einteilen:

1. Logische Operatoren
2. Vergleichsoperatoren
3. Additionsoperatoren
4. Multiplikationsoperatoren
5. Sonstige Operatoren

Die Operatoren binden mit steigender Gruppennummer stärker.[8] Operatoren derselben Gruppe haben die gleiche Bindungsstärke und die Auswertung erfolgt von links nach rechts. Man kann die Standardbindung durch Klammerung von Ausdrücken außer Kraft setzen.

3.2.3.1 Logische Operatoren

Es gib sechs logische Operatoren. Diese sind

and **or** **nand** **nor** **xor** **not**

Diese Operatoren sind für die Datentypen BIT und BOOLEAN vordefiniert. Durch Einbinden der SYNOPSYS-Bibliothek IEEE.std_logic_arith stehen sie auch für den Typ std_logic zur Verfügung. Das Resultat einer logischen Operation hat den gleichen Datentyp wie die Operanden, welche im übrigen zumindest typverwandt sein müssen. Eine logische Verknüfung von einem Objekt des Typs BIT mit einem Objekt des Typs BOOLEAN ist nicht zulässig. Der **not** Operator ist ein ein unärer Operator, er besitzt dieselbe Bindung wie sonstige Operatoren, also höchste Priorität.

3.2.3.2 Vergleichsoperatoren

Es existieren ebenfalls sechs Vergleichsoperatoren:

$\boxed{=}$ (gleich) $\boxed{/=}$ (ungleich) $\boxed{<}$ (kleiner) $\boxed{<=}$ (kleiner gleich)
$\boxed{>}$ (größer) $\boxed{>=}$ (größer gleich)

Der Resultatyp für alle Vergleiche ist BOOLEAN. Als Operanden für den Gleichheits- und den Ungleichheitsoperator sind beliebige Datentypen erlaubt, außer Datei-Typen, die aber lediglich in der Simulation eine Rolle spielen. Die verbleibenden vier Operatoren dürfen auf Skalare Typen (z.B. INTEGER-Typen und Aufzählungstypen) sowie auf diskrete Feld-Typen (z.B. Felder – Arrays – deren Elemente einem skalaren Typ angehören) angewandt werden. Wenn die Operanden diskrete Feld-Typen sind, wird der Vergleich von links nach rechts und Element für Element durchgeführt.

3.2.3.3 Additionsoperatoren

Zu den Additionsoperatoren zählen

$\boxed{+}$ $\boxed{-}$ $\boxed{\&}$

Die Operanden des Additionsoperators (+) und des Subtraktionsoperators (-) müssen denselben numerischen Typ besitzen. Diesen Typ erhält auch das Ergebnis. Die Bibliothek IEEE.std_logic_arith überlädt diese Operatoren für den Datentyp std_logic und std_logic_vector. Beide Operatoren können auch als unäre Operatoren eingesetzt werden; bei std_logic Datentypen

[8] Man könnte auch sagen, sie besitzen mit steigender Gruppennummer stärkeren Vorrang

macht dies jedoch keinen Sinn. Der dritte Additionsoperator wird als Konkatenationsoperator bezeichnet. Er dient dazu, Einzelelemente hintereinanderzuhängen. Hiermit können z.B. Schiebeaktionen implementiert werden.[9]

```
signal a : std_logic_vector(7 downto 0);
signal b : std_logic_vector(7 downto 0);

a(7 downto 0) <= b(5 downto 0) & "00"; -- Linksschieben von b um 2 bit
```

3.2.3.4 Multiplikationsoperatoren

Diese Operatoren sind mit Vorsicht zu verwenden, da sie einen erheblichen Hardwareaufwand nach sich ziehen. Der Multiplikationsoperator $\boxed{*}$ ist der einzige, der in der Synthese wirklich vorbehaltlos zur Verfügung steht. Seine Operanden dürfen einen INTEGER-Typ oder auch einen std_logic-Typ besitzen.
Die weiteren Operatoren

$\boxed{/}$ (Division) **mod** (Modulo) **rem** (Remainder)

sind beim SYNOPSYS-Compier nur erlaubt, wenn der rechte Operand eine *konstante Potenz von Zwei* ist. In diesem Falle lassen sie sich durch geschickte Schiebeaktionen in Hardware umsetzen, was das Synthesetool auch tut.[10]

3.2.3.5 Sonstige Operatoren

Diese sind

abs $\boxed{**}$

Der **abs** Operator bildet den Absolutwert seines Operanden und ist mit beliebigen numerischen Typen zugelassen. Bei Potenzierungsoperator gelten für die Synthese wieder erhebliche Einschränkungen.

3.2.4 Die verschiedenen Beschreibungsarten

Es soll hier nach *Verhaltensbeschreibung*, *Datenflußbeschreibung* und *Strukturbeschreibung* unterschieden werden, obwohl sich nicht alle "Experten" immer ganz über die Zuordnung einzelner Konstrukte zu den Beschreibungsarten einig sind... Es geht hier aber nicht um Wortklauberei, sondern darum, möglichst gute Syntheseergebnisse zu erzielen. In diesem Sinne werden die Beschreibungsarten so definiert, daß relativ gut erläutert werden kann, wann welche Art am angebrachtesten erscheint.

3.2.5 Verhaltensbeschreibung

Wie schon der Ausdruck *Verhalten* andeutet, dient diese Beschreibungsart zur Beschreibung des Verhaltens einer Schaltung, d.h. zur Beschreibung des Kontrollflusses. Beispielsweise ist die Zustandsfortschaltung einer Finite-State-Machine ein ideales Beispiel für eine typische Verhaltensbeschreibung.

[9] Einen Shift-Operator gibt es in VHDL nicht!

[10] Auch Multiplikationen mit konstanten Zweierpotenzen werden erkannt und es wird in solchen Fällen kein Multiplizierer generiert, sondern eine Schiebevorrichtung.

3.2.5.1 Prozesse

Oft wird ein Prozeß auch als Process-Statement bezeichnet, was ich für irreführend halte, da ein Prozeß mehrere Anweisungen beinhalten kann und meistens auch beinhaltet. Die Anweisungen *innerhalb eines* Prozesses werden *nacheinander* (sequentiell) abgearbeitet. Trotzdem werden die Anweisungen alle in einem Zeitschritt (Takt) durchlaufen, es sei denn, sie sind durch "Wait-Statements"[11] voneinander getrennt. Unterschiedliche Prozesse laufen parallel! Jeder Prozeß sollte entweder mindestens ein Wait-Statement beinhalten oder eine Sensitivity-Liste besitzen. Parallele Prozesse, die dieses Kriterium nicht erfüllen, sind nicht simulierbar, da der Simulator letztendlich doch sequentiell arbeitet und zwischen Prozessen hin- und herschaltet. Das geht aber nicht, wenn ein Prozeß sich nie eine Pause gönnt. Zudem handelt es sich bei einem Prozeß ohne Wait-Statement und ohne Sensitivity-Liste im Prinzip um eine asynchrone Beschreibung, und vor solchen sollte man sich ohnehin hüten! Die Syntax eines Prozesses ist in Abbildung 2 dargestellt.

[*Prozeß-Label:*]**process**[**(***Sensitivity-Liste***)**]
    [*Prozeßlokale Deklarationen*]
begin
    Sequentielle Anweisungen; im einzelenen:
        Zuweisungen an Variablen
        Zuweisungen an Signale
        Wait-Statements
        If-Statements
        Case-Statements
        Loop-Statements
        Null-Statements
        Exit-Statements
        Next-Statements
        weitere nur für die Simulation relevante Anweisungen
end process [*Prozeß-Label*];

Abbildung 2: Syntax eines Prozesses

3.2.5.2 Wait Statements und Sensitivity Listen

Mit Hilfe von Wait-Statements ist es möglich, auf bestimmte Ereignisse zu warten. Das klingt sehr schön, ist jedoch stark eingeschränkt: Der SYNOPSYS-Compiler erlaubt zwar mehrere Wait-Statements innerhalb eines Prozesses, allerdings muß immer auf dasselbe Signal gewartet werden. Meistens wird man wohl auf das Auftreten einer Taktflanke warten (clk sei das als Bit oder std_logic definierte Taktsignal):

wait until clk'event and clk = '1';

Hier wird auf eine steigende Taktflanke gewartet (solange bis sich ein Ereignis auf dem Signal Takt zeigt und das Signal '1' ist – Taktflanken werden als beliebig steil angenommen). Ein Prozeß mit Wait-Statements läuft solange, bis er auf ein selbiges trifft. Dann hält er an und wartet auf das vorgegebene Ereignis. Tritt es ein, läuft er weiter, bis er entweder auf das nächste Wait-Statement stößt oder einmal durchgelaufen ist, von vorne angefangen hat und wieder beim selben wait angekommen ist. Die in der SYNOPSYS-Synthese erlaubte Syntax lautet:

[11] Es folgt noch ein eigener Abschnitt über Wait-Statements.

wait until *Boolscher Ausdruck*;

Wenn man nicht darauf angewiesen ist, einem Prozeß mehrere Takte Laufzeit zu gewähren, dann empfehle ich, die Sensitivity-Liste einem Wait-Statement vorzuziehen. WICHTIG: Ein Prozeß kann *entweder* Wait-Statements *oder* eine Sensitivity-Liste besitzen, aber nicht beides gleichzeitig. Eine Sensitivity-Liste besitzt den Vorteil, daß in ihr mehrere Signale angegeben werden können, auf die ein Prozeß warten soll, während in einem Wait-Statement immer nur ein Signal erlaubt ist.[12] Ein Standardfall für ein zweites Signal, auf das ein Prozeß möglichst reagieren soll, ist ein asynchroner Reset:

```
process (clk, clear)
        variable A : INTEGER;
begin
if clear = '1' then
        A := 0;
elsif clk'event and clk = '1' then
        A := A + 1;
end if;
end process
```

Um den gleichen Sachverhalt mit Hilfe eines Wait-Statements zu formulieren, muß man sich schon ziemlich herumwinden. Ein Prozeß mit einer Sensitivity-Liste wartet auf ein Ereignis auf einem beliebigen der in seiner Sensitivity-Liste angeführten Signale. Tritt ein solches auf, läuft er genau einmal durch und wartet dann erneut. Ein Tip: Schreiben Sie nicht zu viele Signale in die Sensitivity-Liste, denn Sie müssen ja auf unterschiedliche Signal-Ereignisse unterschiedlich reagieren und erhalten somit für jedes zusätzliche Signmal auch zusätzliche Kontrollanwisungen. Im wesentlichen sollte eine Schaltung synchron bleiben und somit hauptsächlich vom Takt abhängig sein.[13]

3.2.5.3 IF-Anweisung

Eine If-Anweisung selektiert eine Abfolge von Anweisungen in Abhängigkeit von einer Bedingung. Diese Bedingung kann ein beliebiger Ausdruck sein, dessen Auswertung den Datentyp BOOLEAN ergibt (vgl. Abschnitt über Operatoren Seite 12ff.). Die allgemeine Syntax einer IF-Anweisung zeigt Abbildung 3. Die Bedingungen in den verschiedenen Zweigen werden von oben

if *Boolscher Ausdruck* **then**
        Sequentielle Anweisungen
[**elsif** *Boolscher Ausdruck* **then**
        Sequentielle Anweisungen]
[**else**
        Sequentielle Anweisungen]
end if;

Abbildung 3: Syntax einer IF-Anweisung

nach unten geprüft und sobald eine zutrifft, wird der zugehörige Block von sequentiellen Anweisungen abgearbeitet. Da die IF-Anweisung selbst auch eine sequentielle Anweisung ist, sind auch

[12] In der angegebenen allgemeinen Syntax eines Wait-Statements läßt der Begriff Boolscher Ausdruck den Eindruck entstehen, daß hier beliebige Ausdrücke erlaubt sind. Das ist in der Sprachdefinition wohl auch so gedacht, im Compiler jedoch noch nicht realisiert.

[13] In der Simulation stellt sich dies anders dar; d.h. hier können längere Sensitivity-Listen durchaus Sinn machen.

beliebige Schachtelungen möglich. Ein Beispiel für die Verwendung einer IF-Anweisung haben Sie bereits im vorhergehenden Abschnitt über Sensitivity-Listen gesehen. Ist der VHDL-Code für die Hardwareerzeugung gedacht, sollte darauf geachtet werden, daß immer ein else-Zweig vorhanden ist, da nur in diesem Fall Flip-Flops generiert werden.[14] Die bei fehlendem else-Zweig generierten Latches können den Test eines synchronen Designs beliebig erschweren. Eine Ausnahme bilden clk'event Abfragen; nach Ihnen *darf* kein weiterer elsif-Zweig und auch kein else-Zweig folgen.

3.2.5.4 CASE-Anweisung

Die Syntax einer Case-Anweisung ist in Abbildung 4 dargestellt. Die Case-Anweisung wählt

case *expression* **is**
    when *choices* => *Sequentielle Anweisungen* - - erster Zweig
    when *choices* => *Sequentielle Anweisungen* - - zweiter Zweig
    - - eine beliebige Anzahl von Zweigen ist zulässig
    [**when others** => *Sequentielle Anweisungen*] - - letzter Zweig
end case;

Abbildung 4: Syntax der Case-Anweisung

abhängig vom Wert des Ausdrucks (expression) einen ihrer Zweige zur Ausführung aus. Der Typ des Ausdrucks muß ein Aufzählungstyp, ein Integer-Typ oder ein Eindimensionaler Array-Typ sein! Der Begriff *choices* steht für einen einzelnen Wert oder einen Bereich von Werten, wobei eine Oderverknüpfung von mehreren Werten durch einen senkrechten Strich (|) vorgenommen wird. Alle Werte, die der Ausdruck (expression) annehmen kann, müssen abgedeckt sein. Um dies zu erreichen, kann man den Begriff *others* verwenden, der für alle bisher nicht aufgeführten Werte steht. Wenn der optionale others-Zweig vorhanden ist, muß er der letzte sein.
ANMERKUNG: Die Case Anweisung kann in jedem Fall durch geschachtelte IF-Anweisungen ersetzt werden. Bei manchen Synthesetools ist dies auch angebracht, um die Syntheseergebnisse zu verbessern(z.B. VIEWLOGIC). Beim SYNOPSYS-Design-Compiler ließ sich jedoch keine nachteilige Wirkung des Case-Statements feststellen, so daß dieses Konstrukt, wo angebracht, auch verwendet werden sollte, um die Übersichtlichkeit des Quelltextes zu erhöhen.

3.2.5.5 LOOP-Anweisung

Die Loop-Anweisung dient zur Implementierung von Schleifen. Wie z.B. auch in C und PASCAL gibt es drei verschiedene Arten von Schleifen:

1. Schleifen mit fester Anzahl von Durchläufen. (FOR-Schleifen)
2. Abweisende Schleifen mit Abbruchbedingung. (WHILE-Schleifen)
3. Nicht abweisende Schleifen mit Abbruchbedingung. (REPEAT-Schleifen)

3.2.5.5.1 FOR-Schleifen Abbildung 5 zeigt die Syntax einer For-Schleife. Es sei hier das typische Beispiel zur Berechnung der Fakultät von einer Zahl N angeführt:

[14] Notfalls kann ein NULL-Statement eingefügt werden. Dieses generiert keine überflüssige Hardware.

[*Schleifen-Label:*] **for** *Bezeichner* **in** *range* **loop**
    Sequentielle Anweisungen
end loop [*Schleifen-Label*];

Abbildung 5: Syntax einer For-Schleife

```
Fakultaet := 1;
for Schleifenvariable in 2 to N loop
  Fakultaet := Fakultaet * Schleifenvariable;
end loop;
```

Der Bezeichner, im Beispiel Schleifenvariable genannt, wird *automatisch* als Integer-Variable mit dem Bereich (range) – im Beispiel 2 bis N – deklariert. Diese Variable braucht nicht nur nicht deklariert zu werden, sondern wenn eine Variable gleichen Namens deklariert wird, so ist diese innerhalb der Schleife von der Schleifenvariable überdeckt, d.h. nicht ansprechbar. Es handelt sich folglich um völlig verschiedene Objekte. Die Schleifenvariable darf zwar innerhalb der Schleife gelesen werden, aber ihr dürfen keine neuen Werte zugewiesen werden. Am Schleifenende wird die Varaible jeweils um Eins erhöht.
Leider gibt es eine synthesetoolbedingte Einschränkung, die For-Schleifen für die Synthese so gut wie wertlos machen: SYNOPSYS erlaubt innerhalb von For-Schleifen keine Wait-Statements. Das bedeutet, daß die gesamte Schleife (nicht nur der Schleifenrumpf!) in einem einzigen Taktschritt abgearbeitet werden müssen. Für das Fakultätsbeispiel von eben hieße das, es müßten zur Umsetzung in Hardware N Multiplizierer generiert werden. Dies ist nicht nur deshalb unmöglich, weil bereits ein einzelner Multiplizierer erheblichen Platzbedarf besitzt, sondern auch weil N eine Variable ist.

3.2.5.5.2 WHILE-Schleifen Abbildung 6 zeigt die Syntax einer abweisenden Schleife mit Abbruchbedingung. Die Anweisungen innerhalb des Schleifenrumpfes werden nacheinander aus-

[*Schleifen-Label:*] **while** *Boolscher Ausdruck* **loop**
    Sequentielle Anweisungen
end loop [*Schleifen-Label*];

Abbildung 6: Syntax einer While-Schleife

geführt, und der Rumpf immer wieder von vorne durchlaufen, solange die durch den Booleschen Ausdruck repäsentierte Bedingung den Wert TRUE hat. Ist dies nicht mehr der Fall, wird mit der auf das "end loop" folgenden Anweisung fortgesetzt.
Im Gegensatz zu For-Schleifen sind bei While-Schleifen und bei den noch zu erläuternden Repeat-Schleifen *Wait-Statements* innerhalb des Schleifenrumpfes nicht nur nicht verboten, sondern ausdrücklich *vorgeschrieben*. Dies ist wiederum kein Feature der Sprache VHDL sondern synthesetoolbedingt.

Mit dieser Vorgabe läßt sich das Fakultätsbeispiel äquivalent umformen:

```
Fakultaet := 1;
Schleifenzaehler := 1;
while Schleifenzaehler <= N loop
  wait until clk'event and clk = '1';
  Fakultaet := Fakultaet * Schleifenzaehler;
  Schleifenzaehler := Schleifenzaehler + 1;
end loop;
```

In dieser Form werden nur ein Multiplizierer und ein Addierer benötigt und die Beschreibung ist somit synthetisierbar.

3.2.5.5.3 REPEAT-Schleifen Der Ausdruck "REPEAT-Schleifen" wurde von mir gewählt, um nicht abweisende Schleifen mit Abbruchbedingung unter einem Sichwort fassen zu können. Das Wort REPEAT gehört nicht zum Sprachumfang von VHDL. Trotzdem ist es möglich, entsprechende Schleifen zu beschreiben. Abbildung 7 zeigt die Syntax. Das Exit-

[*Schleifen-Label:*] **loop**
  Sequentielle Anweisungen
  exit [*loop-label*] [**when** *Bedingung*];
end loop [*Schleifen-Label*];

Abbildung 7: Syntax einer Repeat-Schleife

Statement innerhalb des Schleifenrumpfes gehört nicht direkt zu diesem Schleifenkonstrukt.[15] Es wird im folgenden Abschnitt noch näher darauf eingegangen. Wie bei While-Schleifen erfordert die Synthese mit dem SYNOPSYS Compiler mindestens ein Wait-Statement im Schleifenrumpf. Hier nochmal die Fakultätsberechnung als REPEAT-Schleife:

```
Fakultaet := 1;
Schleifenzaehler := 2;
if Schleifenzaehler >= N then -- korrekte Behandlung von 0 und 1
  Fact:loop
    wait until clk'event and clk = '1';
    Fakultaet := Fakultaet * Schleifenzaehler;
    Schleifenzaehler := Schleifenzaehler + 1;
    exit Fact when Schleifenzaehler > N;
  end loop;
end if;
```

3.2.5.5.4 EXIT-Anweisung Die Exit-Anweisung darf nur innerhalb von Schleifen verwendet werden. Sie bewirkt einen Sprung aus der innersten Schleife heraus oder einen Sprung aus der Schleife hinaus, deren Schleifen-Label angegben ist. Die **when** Klausel ist optional. Wird sie weggelassen, so wird die Schleife in jedem Fall verlassen, sonst nur bei Eintreten der Bedingung. Es sind mehrere Exit-Anweisungen innerhalb einer Schleife zulässig.

[15] Natürlich braucht eine Schleife eine Abbruchbedingung, um irgendwann einmal zu terminieren. Deshalb wurde hier bereits auf die Exit-Anweisung vorgegriffen.

exit [*loop-label*] [**when** *Bedingung*];

Abbildung 8: Syntax einer Exit-Anweisung

3.2.5.5.5 NEXT-Anweisung

Die Next-Anweisung ist ebenfalls eine sequentielle Anweisung und besitzt die gleiche Syntax wie die Exit-Anweisung. (Abbildung 9) Diese Anweisung

next [*loop-label*] [**when** *Bedingung*];

Abbildung 9: Syntax einer Next-Anweisung

dient dazu, einen Schleifendurchlauf vorzeitig abzubrechen. Im Gegensatz zur Exit-Anweisung wird jedoch nicht die Schleife verlassen, sondern wieder mit der ersten Anweisung innerhalb des Schleifenrumpfes begonnen. Next-Statements sind innerhalb beliebiger Schleifen erlaubt. In For-Schleifen wird die Inkrementierung der Schleifenvariablen korrekt behandelt, d.h. ausgeführt, bevor die nächste Schleifeniteration beginnt.

3.2.6 Datenflußbeschreibung

Bei der Datenflußbeschreibungsmethode geht es darum, den Informationsfluß durch eine Entity zu beschreiben. Das wesentliche Mittel hierfür sind parallele Signalzuweisungen (concurrent signal assignment statements).

3.2.6.1 Parallele Signalzuweisungen

Signalzuweisungen außerhalb von Prozessen sind erlaubt und werden als Parallele Signalzuweisungen bezeichnet; im Gegensatz zu sequentiellen Signalzuweisungen innerhalb von Prozessen. Am einfachsten lassen sich parallele Signalzuweisungen wohl durch ein Beispiel erläutern. Abbildung 10 zeigt ein UND-Gatter.

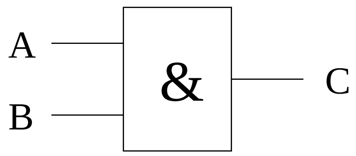

Abbildung 10: UND-Gatter

Dieses UND-Gatter kann mit einer parallelen Signalzuweisung wie folgt modelliert werden:

```
entity UND2 is
  port (A,
        B : in std_logic;
        C : out std_logic);
end UND2;

architecture Datenfluss of UND2 is
begin
  C <= A and B; -- parallele Signalzuweisung
end Datenfluss;
```

Genausogut hätte man dies in Form einer Verhaltensbeschreibung formulieren können (die Entity bleibt gleich):

```
architecture Verhalten of UND2 is
begin
  process(A,B)
  begin
    C <= A and B; -- sequentielle Signalzuweisung
  end process;
end Verhalten;
```

In diesem Fall handelt es sich um eine Sequentielle Signalzuweisung. Da in die Sensitivity-Liste beide Signale A und B eingetragen wurden, verhalten sich beide Beschreibungen völlig identisch. Es ist allerdings leicht einzusehen, daß die Datenfluß-Form bei mehreren Anweisungen – es müßte ja für jede Anweisung ein eigener Prozeß erzeugt werden – wesentlich übersichtlicher ist. Die Funktion sei nochmal etwas klarer hervorgehoben: Parallele Signalzuweisungen werden immer dann ausgewertet, wenn sich auf mindestens einem der Quellsignale etwas ereignet. Auf den ersten Blick scheinen also parallele Signalzuweisungen eine bequemere Schreibweise für bestimmte Ausdrücke zu ermöglichen, aber keine echte Neuerung zu bringen. Anders wird dies, wenn man in Betracht zieht, daß die Abfrage von Bedingungen innerhalb von parallelen Signalzuweisungen möglich ist. (Bedingte Signalzuweisung - conditional signal assignment). Im folgenden Beispiel sei M ein beliebiges Signal und S der aktuelle Zustand einer Finite State Machine:

```
M <= '1' when (S = S5) and (Res = 0) else
     'X' when (S = DC) else
     '0';
```

Sie werden im Praktikum selbst ein bis zwei Finite-State-Machines aufbauen. Es soll den Aufgaben hier nicht vorgegriffen werden, jedoch sei erwähnt, daß Sie dieses Konstrukt beim Entwurf von Finite-State-Machines im Hinterkopf haben sollten. EMPFEHLUNG: Trennung von Zustandsfortschaltung und Datenfluß. Probieren Sie beides einmal aus (also auch Mixen von Zustandsfortschaltung und Datenfluß); Sie werden es dann nicht mehr vergessen...

Auch an dieser Stelle sei nochmal darauf hingewiesen, daß ein Signal nur einen Treiber haben darf oder eine Konfliktlösungsfunktion besitzen muß. Das bedeutet, es kann einem Signal entweder nur in *einem* Prozeß oder nur in *einem* concurrent signal assignment statement ein Wert zugewiesen werden!

Es wird hier noch ein weiteres Beispiel angeführt, das im nächsten Kapitel wieder aufgegriffen werden soll: Ein Volladdierer in Form einer Datenflußbeschreibung:

```
entity Volladdierer is
  port (A,
        B,
        CIN : in std_logic;
        SUM,
COUT: out std_logic);
end Volladdierer;

architecture Volladdierer_Arch of Volladdierer is
begin
  SUM <= (A xor B) xor CIN;
  COUT <= (A and B) or (B and CIN) or (CIN and A);
end Volladdierer_Arch;
```

3.2.7 Strukturbeschreibung

Bei der Strukturbeschreibungsmethode geht es darum, konkret verschiedene Designblöcke, Komponenten oder Module miteinander zu "verdrahten". Das Verhalten einer Komponenten ist dabei nicht explizit sichtbar. Um eine Komponente verwenden zu können, muß sie zunächst einmal deklariert werden. Anschließend können beliebig viele Instanzen einer deklarierten Komponente erzeugt werden. An Hand des Aufbaus eines Mehrbit-Addierers soll das Vorgehen bei der Strukturbeschreibung beispielhaft erläutert werden. Ziel ist der Aufbau der in Abbildung 11 gezeigten Schaltung. Wie aus dem Schaubild hervorgeht, werden zwei Instanzen der Komponente Volladdierer benötigt.

3.2.7.1 Komponentendeklaration

In einer Komponentendeklaration wird der Name einer Komponente und die Spezifikation der Ports angegeben, d.h. sie ähnelt sehr der Deklaration einer Entity. Die Entity zur deklarierten Komponente muß entweder im aktuellen Verzeichnis in vorcompilierter Form oder in einer eingebundenen Bibliothek zu finden sein. Wenn dies nicht der Fall ist, gibt es noch die Möglichkeit, die Komponente mit Hilfe einer *configuration* einer Entity zuzuorden.[16] Die Syntax einer Komponentendeklaration ist in Abbildung 12 dargestellt. Da von configurations für die Synthese abgeraten wird, müssen die nicht nur die Typen und Richtungen der Ports mit der in der korrespondierenden Entity übereinstimmen, sondern auch die Portnamen. Der Ort für Komponen-

[16] Der Autor hat schlechte Erfahrungen mit configurations bei der Synthese gemacht, sprich, sie ließen sich nicht synthetisieren. Im Gegensatz dazu sind configurations im Bereich von Testbenches (Simulation) sehr hilfreich; sie werden dort näher erläutert.

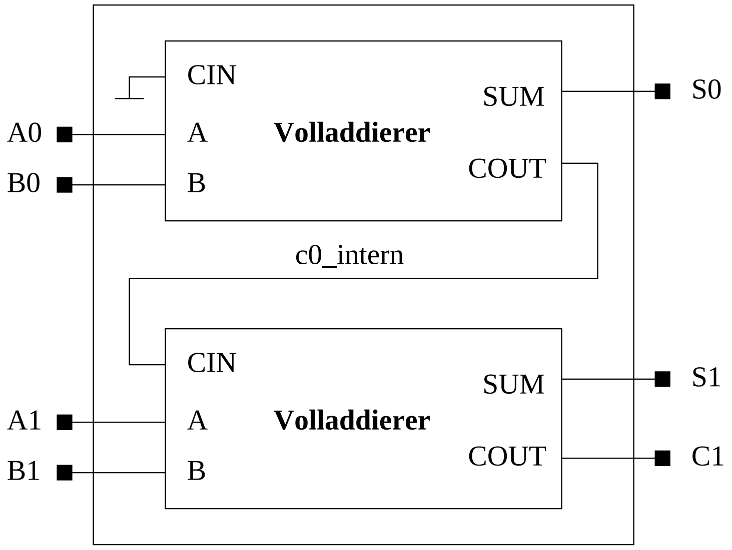

Abbildung 11: Zwei Bit Volladdierer

component *Komponenten-Name*
  port *Liste der Schnittstellenports*
end component

Abbildung 12: Syntax einer Komponentendeklaration

tendeklarationen ist der Deklarationsteil einer Architecture. Für das gewählte Addiererbeispiel sieht die Komponentendeklaration wie folgt aus:

```
component Volladdierer
  port (A,
        B,
        CIN : in std_logic;
        SUM,
        COUT: out std_logic);
end component;
```

3.2.7.2 Komponenteninstanzierung

Während durch eine Komponentendeklaration eine Komponente nur bekannt gemacht wird, erzeugt eine Instanzierung ein reales Objekt und die Ein- und Ausgänge der instanzierten Komponente werden mit Signalen verbunden. Die Syntax einer Komponenteninstanzierung zeigt Abbildung 13. Das Komponenten-Label kann als Name der Instanz (nicht der Komponente)

Komponenten-Label: *Komponenten-Name* **port map** (*Assoziierungsliste*);

Abbildung 13: Syntax einer Komponenteninstanzierung

betrachtet werden. Der Komponenten-Name muß der Name einer *vorher* durch eine Komponentendeklaration bekannt gemachten Komponente sein. Die Assoziierungsliste assoziiert Signale innerhalb der aktuellen Entity mit den Ports der Komponente. Es gibt zwei Möglichkeiten die Zuordnung der Signale zu den Ports in der Assoziierungsliste vorzunehmen:

1. Assoziierung durch Position. Hierbei muß die Reihenfolge der in der Port Map der Instanzierung eingetragenen Signale genau der Reihenfolge der in der Port Map der Deklaration eingetragenen – und den Signalen zuzuordnenden – Ports entsprechen. Zugegebenermaßen ein Bandwurmsatz, aber das Beispiel wird sehr schnell deutlich machen, was gemeint ist.

2. Assoziierung durch Benennung. Hierbei wird für jeden Komponentenport eine genaue Zuordung Portname => Signalname vorgenommen.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Beispieladdierer is
  port(A0,
       A1,
       B0,
       B1: in std_logic;
       S0,
       S1,
       C1: out std_logic);
end Beispieladdierer;

architecture Beispieladdierer_Struktur of Beispieladdierer is
-- Komponetendeklaration im Deklarationsteil der Architecture
  component Volladdierer
    port (A,
          B,
          CIN : in std_logic;
          SUM,
          COUT: out std_logic);
  end component;

-- Deklaration interner Signale
  signal c0_intern : std_logic; --internes Carry-Signal
  signal logisch_Null : std_logic;

begin

-- Komponenteninstanzierung mit Assoziierung durch Position
-- ( die Reihenfolge muss exakt eingehalten werden )
  Addierer0: Volladdierer port map
             (A0, B0, logisch_Null, S0, c0_intern);

-- Komponenteninstanzierung mit Assoziierung durch Benennung
-- ( die Reihenfolge spielt keine Rolle )
  Addierer1: Volladdierer port map
             (B=>B1, A=>A1, COUT=>C1, SUM=>S1, CIN=>c0_intern);

-- concurrent signal assignment
  logisch_Null <= '0';

end Beispieladdierer_Struktur;
```

Sicher ist der hier beschriebene Addierer in keinster Weise optimal; beispielsweise wäre sicher ein Halbaddierer in der untersten Stufe angebracht, um das Null-Signal und Fläche zu sparen uvam. Wenn Sie keine speziellen Wünsche haben, brauchen Sie ohnehin keinen Addierer zu bauen, da mindestens einer in jeder ASIC-Standardbibliothek vorhanden sein sollte. Aber ich hoffe, daß Ihnen durch dieses kleine Beispiel die Funktionsweise der Strukturbeschreibung klar geworden ist. Sie werden im Praktikum sicher regen Gebrauch von dieser Beschreibungsart machen, da sie sehr schön eine hierarchische Entwicklung und das Testen von Unterkomponenten ermöglicht.

Literatur

[1] Ben Cohen. *VHDL Coding Styles and MEthodologies.* Kluwer Academic Publishers, 1995.

[2] Jayaram Bhasker. *A VHDL Primer.* Prentice-Hall International Limited, 1992.

[3] IEEE Standard VHDL Language Reference Manual. The Instuitute of Electrical and Electronics Engineers,Inc, March 1988. IEEE Std 1076-1987.

[4] Synopsys Inc. *VHDL Compiler Reference Manual (VHDL Compiler)*, 1992.

[5] Prof. Dr.-Ing. R. Ernst. Automatisierte Synthese von Digitalschaltungen. Vorlesung an der Technischen Universität Braunschweig, 1993.

Index