Eine Einführung in die Schaltungsentwicklung mit VHDL

Version Juli 2014

Urs Graf, Laszlo Arato

Beispiele und Illustrationen wurden entnommen aus:

„VHDL Kompakt“, http:// tams-www.informatik.uni-hamburg.de

„The Student’s Guide to VHDL“, P. Ashenden

“VHDL Tutorial”, http://www.vhdl-online.de/

1 Einführung in VHDL

Ziel dieser Einführung	Diese Einführung soll die wesentlichen Konzepte von VHDL aufzeigen. Anhand praktischer Beispiele wird der Gebrauch dieser Beschreibungssprache erläutert. Diese Einführung ist keine komplette Referenz aller VHDL-Konstrukte. Weiterführende Konzepte wie z.B. generische Module, GENERATE und Overload wurden weggelassen.
„VHDL“	**V**ery **H**igh Speed Integrated Circuit Hardware **D**escription **L**anguage. Mit VHDL kann man digitale Schaltungen und Systeme durchgängig von der Systemdefinition bis zum Schaltungsentwurf beschreiben und verifizieren.
Top-Down-Design	Top-Down-Design ist der Ansatz zur Lösung von Designproblemen bei zunehmend komplexerer Hardware. Bei dieser Methodik wird zuerst das Verhalten eines Designs auf einer hohen Abstraktionsebene beschrieben. Dabei ist es wichtig, dass noch keinerlei Rücksicht auf Technologien oder Details der Implementierung genommen werden muss. Um die Spezifikation selbst großer Schaltungen genau definieren und austesten zu können, wird eine Simulation aufgrund der abstrakten Beschreibung des Modells durchgeführt. Damit lassen sich mehrere Varianten der Spezifikation ohne großen Aufwand am Simulator miteinander vergleichen. Die Simulationsergebnisse einer solchen Spezifikation können dann über die gesamte Entwicklungsphase als Maßstab für die Übereinstimmung zwischen Spezifikation (Zielsetzung) und Design (Ergebnis) dienen.
Wie wird VHDL eingesetzt?	VHDL ist eine sehr mächtige und flexible Sprache. Im Gegensatz zu klassischen Programmiersprachen kann VHDL sowohl sequentielle wie auch parallel ablaufende Vorgänge beschreiben. VHDL wurde nicht nur für die Beschreibung von elektronischen Bauelementen entworfen, sondern für die Spezifikation und funktionelle Simulation von komplexen Schaltkreisen, Baugruppen und Systemen.
Synthetisierbares VHDL	Anders als beim schematischen Design mit Logik-Blöcken die manuell platziert und verbunden werden bietet die automatische Synthese einer in VHDL definierten Struktur viele Vorteile wie die Eliminierung von duplizierter Logik, Reduktion von kombinatorischen Vorgängen und Implementation der gewünschten Funktion mit verfügbaren Gattern und Logikbauteilen (z.B. in einem FPGA). Nur ein relativ kleiner Teil der Sprache VHDL kann wirklich effizient synthetisiert werden (z.B. `std_logic_vector`, `std_logic`, `signed` und `unsigned`). Andere Elemente werden zwar von einem Synthesewerkzeug umgesetzt, aber erzeugen ineffiziente Strukturen (z.B. `INTEGER`, `REAL`). Deshalb muss beim Schreiben von VHDL Code für FPGA und ASIC immer vor Augen gehalten werden, was das Synthesewerkzeug damit anstellen wird.
Nicht synthetisierbares VHDL	VHDL wird nicht nur beim Design der Logik in FPGAs und ASICs eingesetzt, sondern auch zur Beschreibung und Funktion von Testumgebungen in Simulationen, sogenannten Test-Benches. Da dieser Code immer nur simuliert und nie synthetisiert wird, darf man hier nach Belieben alle Sprachelemente und Konstrukte von VHDL verwenden.
VHDL 87	Im Jahre 1987 wurde VHDL als internationaler Standard mit der Bezeichnung IEEE Std. 1076-1987 eingeführt.

VHDL 93	Sechs Jahre später wurden etliche Modifikationen gemacht und die Sprache um einige Elemente erweitert. Dieser Standard heisst IEEE Std. 1076-1993, kurz VHDL93. In diesem Kurs verwenden wir diese neuere Fassung.
VHDL 2008	Mit VHDL 2008 wurde die Sprache um ein paar praktische Elemente erweitert. Während diese zusätzlichen Funktionen in Quartus II, Version 11.0 bereits implementiert sind, werden sie von ModelSim der Firma Mentor Graphics bis Version 6.6 noch nicht unterstützt … … deshalb verzichten wir im Rahmen dieses Kurses auf diese Funktionen.
VHDL-AMS	VHDL wurde auch um analoge und mixed-signal Sprachelement erweitert. Diese Erweiterung nennt sich VHDL-AMS (für Analog Mixed Signal) und ist eine Obermenge zu VHDL. Die rein digitalen Methoden von VHDL93 werden auch weiterhin gelten. Es ist absehbar, dass auf der analogen Seite bis auf weiteres nur die Simulation durchführbar ist.

Hinweis zur Benutzung dieses Leitfadens:

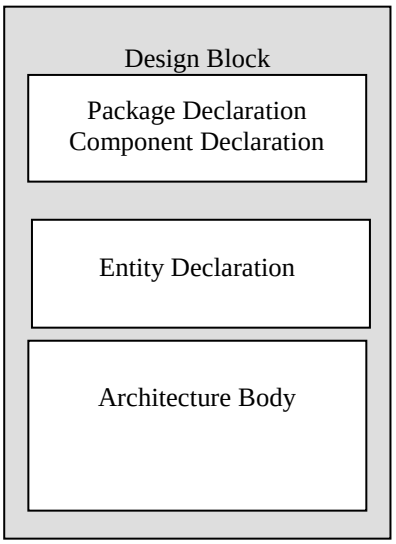
In den folgenden Kapiteln werden die wesentlichen Sprachkonstrukte von VHDL präsentiert. Die linke Spalte zeigt stets die Theorie und Erklärung dazu, auf der rechten Seite finden Sie passende Beispiele.

2 Grundstruktur eines VHDL-Modells und 1. Beispiel

2.1 Elemente des VHDL-Modells

Der Code eines VHDL-Modells besteht immer aus zwei zusammenhängenden Teilen: einer *Entity* und einer **ARCHITECTURE**. Der Begriff *Entity* bezeichnet ein zusammenhängendes, in sich abgeschlossenes System mit einer definierten Schnittstelle zur Umgebung. Neben anderen Dingen wird eben diese Schnittstelle in Form von Signalen in der Entity beschrieben. Eine Entity entspricht also einem sozusagen einem Symbol in einer grafischen Schaltungsbeschreibung.

Im Gegensatz dazu beschreibt die **ARCHITECTURE** den inneren Aufbau und die Funktion des Systems. Die **ARCHITECTURE** entspricht daher der Netzliste bei der grafischen Beschreibung. Während eine Architektur immer einer bestimmten Entity zugeordnet ist, kann eine Entity kann beliebig viele Architekturen „besitzen“. Dadurch lassen sich ohne Änderung des Systemnamens verschiedene Versionen oder Arten eines Modells verwalten.



2.1.1 Kommentare

Mit „--„ beginnender Text ist in VHDL bis zum Ende der jeweiligen Zeile als Kommentar definiert und wird vom Compiler ignoriert.

```
--  Dies ist nur rein Kommentar der auf
--  der zweiten Zeile fortgesetzt wird
```

2.1.2 Die Entity

Die Entity ist die Schnittstellenbeschreibung zwischen einem Moduls und der Umwelt.

Worte in Grossbuchstaben, wie beispielsweise ENTITY oder PORT sind VHDL-spezifische Schlüsselwörter mit einer bestimmten Bedeutung (ähnlich wie „for“ oder „int“ in Java). Diese Reservierten Wörter dürfen nicht für eigene Bezeichnungen verwendet werden.

```
ENTITY entity_name IS
    PORT ( port_list );
END [entity_name];
```

Der *entity_name* bezeichnet den Namen des Systems. Auf diesen Namen bezieht sich dann die jeweilige Umgebung des Systems.

Die *port_list* bezeichnet eine Liste von Ein- und Ausgangssignalen des Systems.

2.1.3 Die port_list

Bei der PORT Liste handelt es sich um eine durch Strichpunkt getrennte Liste von Eingangs- und Ausgangssignalen. Jedes Signal wird durch ein Bezeichner, ein Modus und ein Datentyp spezifiziert.

Der Strichpunkt trennt nur die einzelnen Elemente der Liste, und deshalb hat das letzte Element (output_1) am Ende **keinen** Strichpunkt!

```
ENTITY entity_name IS
    PORT (
        input_1  : signal_type;
        input_2  : signal_type;
        output_1 : signal_type
    );
END;
```

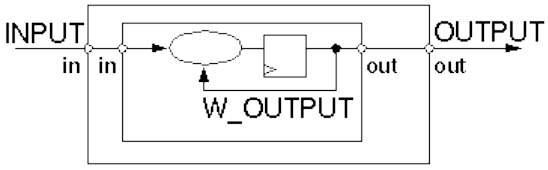
Als Modus kommen die Schlüsselwörter IN, OUT, INOUT oder BUFFER in Frage. Der Datentyp beschreibt die Art der Werte, die das PORT-Signal annehmen kann.

2.1.4 Port Modi

Der Modus eines Entity-Ports gibt die Datenflußrichtung vor.

Der Modus **IN** wird für rein lesbare Signale verwendet. Signale, deren Ports diesen Mode haben, können in der untergeordneten Architektur keine Werte zugewiesen werden.

Entsprechend bezeichnet der **OUT** Modus Signale, denen ein Wert zugewiesen werden kann, aber die nicht gelesen werden können.

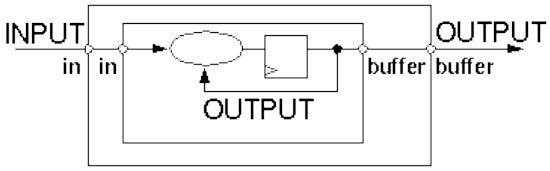



```
PORT (
    a, b : IN  std_logic;
    c    : OUT std_logic
);
```

Ein Port mit dem Modus **BUFFER** erlaubt es, ein Ausgangssignal auch intern zu „sehen“.

Die Verwendung des Port-Modus **BUFFER** scheint bequemer, als ein zusätzliches internes Signal zu erzeugen welches auf das Ausgangssignal kopiert wird.

Gerade bei hierarchischen Strukturen wird dies aber zur Zeitbombe, weil Ports mit Modus **BUFFER** nicht mit anderen Ports verbunden werden können!!!



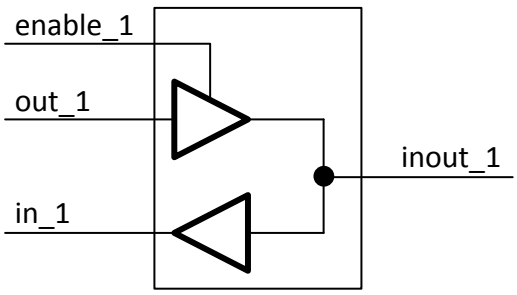

```
PORT (
    clk   : IN     std_logic;
    count : BUFFER std_logic
);
```

Wir werden diesen Port-Modus nicht verwenden!!!

Wenn ein Modul treibend und lesend auf einen Port zugreifen muss, wird der Modus **INOUT** verwendet.

INOUT beschreibt eigentlich nichts anderes als einen Tri-State Buffer.

INOUT macht nur für Signale an der Peripherie eines Chips Sinn, wie zum Beispiel für einen bidir. Datenbus!




```
PORT (
    in_1     : IN    std_logic;
    enable_1 : IN    std_logic;
    out_1    : OUT   std_logic;
    inout_1  : INOUT std_logic
);
```

Innerhalb eines FPGAs gibt es aus technischen Gründen keine echten Tri-State Signale, alles wird auf klar definierte Signale abgebildet.

Bei einem ASIC ist es möglich einen echten internen Tri-State Bus definieren … aber dann muss man GANZ GENAU wissen was man tut – weil dann z.B. keine automatische Optimierung und keine statische

Timing-Analyse mehr möglich ist.

2.1.5 Die Architektur

Die Architektur beschreibt das innere Verhalten eines Moduls.

Die Architektur beginnt mit dem Schlüssel-wort **ARCHITECTURE** und einem selbstdefinierten Namen. Dann folgt der Name der **ENTITY**, zu der diese Architektur gehört. Dies stellt einen eindeutigen Bezug zu einer **ENTITY** her und bietet andererseits die Möglichkeit einer Verknüpfung beliebiger Architekturen zu einer **ENTITY**.

Im Bereich vor dem Schlüsselwort **BEGIN** kann man architekturspezifische Objekte deklarieren, Eine Erläuterung hierzu folgt später.

```
ARCHITECTURE arch_name OF entity_name IS
--  declarative region

BEGIN
--  architecture body

END [ARCHITECTURE] arch_name;
```

2.2 Packages und Libraries

Im nächsten Beispiel wird der Typ STD_LOGIC verwendet.

Dieser Typ ist nicht in VHDL selbst enthalten, sondern wird in einem sogenannten Paket (**PACAKGE**) definiert. Pakete werden in einer Bibliothek (entspricht einem Verzeichnis auf Ihrem Rechner) gesammelt.

Mit der **USE**-Klausel wird ein Paket eingebunden. Das .**ALL** nach dem Paketnamen bedeutet, dass alle Teile aus einer Bibliothek verwendet werden. Das können z.B. Konstanten, Typen, Signale oder Komponenten sein.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
```

2.2.1 Vollständiges Beispiel

Das zu entwerfende System soll von zwei digitalen Signalen angesteuert werden und ein einzelnes Digitalsignal ausgeben. Wenn an beiden Eingängen der High-Zustand anliegt, soll der Ausgang den Low-Zustand annehmen. Für jede andere Kombination an den Eingängen soll der Ausgang im High-Zustand sein.

Beim Betrachten der Wahrheitstabelle erkennt man: Beim beschriebenen Verhalten handelt es sich um ein NAND-Gatter.

Natürlich bräuchte man dazu keine solch komplizierte Struktur sondern man könnte die in VHDL eingebaute Funktion verwenden. Um das erste Beispiel einfach zu halten, werden wir dennoch die Implementation dieses NANDs in verschiedenen Varianten zu untersuchen.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY nand_gate IS
    PORT (
        in_a   : std_logic;
        in_b   : std_logic;
        output : std_logic
    );
END;
```

Die erste Architektur „**truth**“ implementiert genau die Wahrheitstabelle mit einer Reihe von IF – THEN – ELSE Befehlen.

Nicht sehr elegant, aber genau der Spezifikation entsprechend –sollte man zumindest meinen.

Der Signal-Typ „std_logic“ dient, anders als der Typ BIT, nicht nur zur Darstellung von logisch ‚1‘ und ‚0‘, sondern zur Modellierung von realen Signalen. Reale Signale können neben den logischen Zuständen noch Zustände wie ‚U‘ (**Uninitialized**), ‚X‘ (**Unknown**), und noch einige mehr annehmen (siehe Kap. 4.1).

Was passiert nun, wenn der Eingang A auf ‚0‘ ist, und der Eingang B auf ‚X‘ steht? Intuitiv würde man meinen, dass dann der Ausgang 1 sein sollte, weil dieser nur dann auf ‚0‘ geht, wenn beide Eingänge auf ‚1‘ sind. Aber unsere Schaltung macht das nicht so.

```
ARCHITECTURE truth OF nand_gate IS
  --  Truth table
  --  ===========
  --   in_a  in_b  |  output
  --   ------------|--------
  --     0     0   |    1
  --     0     1   |    1
  --     1     0   |    1
  --     1     1   |    0
BEGIN
  PROCESS (in_a, in_b)
  BEGIN
    IF    in_a='0' AND in_b='0' THEN
          output <= '1';
    ELSIF in_a='0' AND in_b='1' THEN
          output <= '1';
    ELSIF in_a='1' AND in_b='0' THEN
          output <= '1';
    ELSE  output <= '0';
    END IF;
  END PROCESS;
END ARCHITECTURE truth;
```

Also definieren wir eine zweite Architektur „**smart**“, die diesen Fehler nicht hat und auf unspezifische Eingänge richtig reagiert.

In-VHDL gibt es relationale Operatoren wie = oder >=. Diese liefern einen Booleschen Wert zurück. Daneben existieren aber auch logische Operatoren wie AND, OR, NAND, usw. Mit diesen lässt sich das vorangegangene Beispiel eleganter formulieren.

Wenn man (wie wir hier) mehr als nur eine Architektur verwendet, dann muss man mit der **CONFIGURATION** definieren, welche der Architekturen man simulieren oder synthetisieren will.

```
ARCHITECTURE smart OF nand_gate IS
  --  Smart truth table
  --  =================
  --
  --   in_a  in_b  |  output
  --   ------------|--------
  --     0     X   |    1
  --     X     0   |    1
  --     1     1   |    0
  --     Other     |    X
BEGIN
  PROCESS (in_a, in_b)
  BEGIN

    IF    in_a='0' OR in_b='0' THEN
          output <= '1';
    ELSIF in_a='1' AND in_b='1' THEN
          output <= '0';
    ELSE  output <= 'X';
    END IF;
  END PROCESS;
END ARCHITECTURE smart;
```

```
ARCHITECTURE dataflow OF nand_gate
IS
BEGIN

    output <= in_a NAND in_b;

END ARCHITECTURE smart;
```

2.3 Konfiguration

Es gibt in VHDL die Möglichkeit, mit dem Schlüsselwort „**CONFIGURATION**“ eine bestimmte Architektur zur Implementation auszuwählen.

Dabei ist die Syntax etwas speziell, wie man am Beispiel rechts sehen kann.

Für ALTERA Quartus II funktioniert dies ganz gut.

```
CONFIGURATION nandconf OF nand_gate
IS
    FOR smart
    END FOR;
END CONFIGURATION nandconf;
```

Leider hat es sich in Versuchen gezeigt, dass ModelSim den Befehl „**CONFIGURATION**“ nicht beachtet, sondern dass zur Simulation immer die LETZTE Architektur verwendet wird.

Dies kann entsprechend zu sehr unterschiedlichen Ergebnissen zwischen Simulation und Implementation führen.

Wenn man mehrere Architekturen verwendet, und sicherstellen will dass die richtige Architektur simuliert wird, empfiehlt sich statt dem „**CONFIGURATION**“ Befehl die verwendung von je einem File pro Architektur.
Dadurch kann bei Quartus in der Liste der Design-Files einfach das File eingebunden werden, welches die gerade gewünschte Architektur enthält. Die Kompilations-Liste für Modelsim wird jeweils von Quartus automatisch erstellt, oder kann bei Bedarf auch manuell editiert werden.

3 Konzepte

3.1 Grundlagen

VHDL besitzt zwei Arten von Anweisungen.

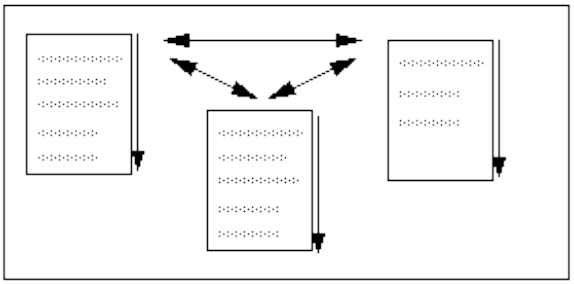
Sequentiell

Sequentielle Anweisungen werden wie bei Softwarebeschreibungssprachen strikt nacheinander abgearbeitet. Nachfolgende Anweisungen können vorhergehende dabei überschreiben. Die Reihenfolge der Anweisungen muss also beachtet werden. Damit sequentielle Anweisungen auch wirklich sequentiell bearbeitet werden, packt man sie in einen *Prozess*.

Concurrent

Nebenläufige Anweisungen sind gleichzeitig wirksam. Die Reihenfolge der Anweisungen spielt also keine Rolle. Mit nebenläufigen Anweisungen wird die Parallelität von echter Hardware nachgebildet.

Beispiel



Das Beispiel zeigt drei nebenläufige (concurrent) Blöcke mit sequentiellen Anweisungen.

Prozesse

Ein Prozess in VHDL ist ein besonderer Bereich mit sequentieller Abarbeitung. Ein Prozess wird von bestimmten Signaländerungen (definiert in einer Sensitivitätsliste) angestossen und dann analog einer konventionellen Programmiersprache abgearbeitet. Für interne Zwischenresultate können Variablen definiert werden.

Für die Modellierung werden 3 wichtige Methoden verwendet

Modularität

Die Modularität hat das Ziel, große Funktionsblöcke zu unterteilen und in abgeschlossene Unterblöcke, den so genannten Modulen, zusammenzufassen. Module werden in Packages und Libraries zusammengefasst und können als Dienstmodule in neuen Projekten wiederverwendet werden.

Abstraktion

Die Abstraktion erlaubt es, verschiedene Teile eines Modells unterschiedlich detailliert zu beschreiben. Module, die nur für die Simulation gebraucht werden, müssen z.B. nicht so genau beschrieben werden, wie Module die für die Synthese gedacht sind. Verschiedene Ebenen der Abstraktion sind z.B. die Verhaltensbeschreibung oder die Register-Transfer-Ebene (siehe weiter unten).

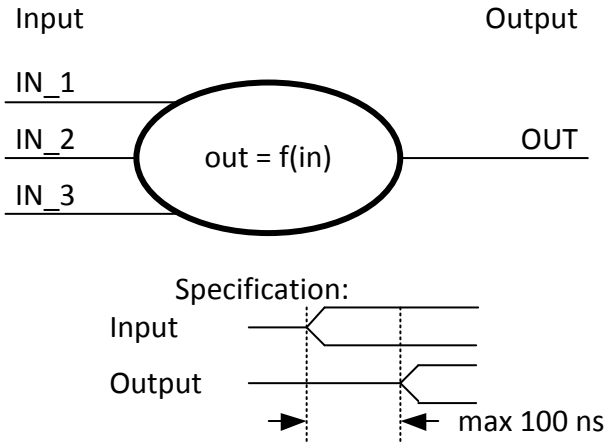
Hierarchie

Die Hierarchie erlaubt es, ein System aus mehreren Modulen, die wiederum aus mehreren Modulen bestehen können, aufzubauen. Eine Ebene in der Beschreibungshierarchie kann ein oder mehrere Module mit unterschiedlichen Abstraktionsgrad, beinhalten. Die Untermodule der darin enthaltenen Module bilden die nächste darunter liegende Hierarchieebene. Dadurch kann ein Entwurf schrittweise Top-Down verfeinert werden.

3.2 Abstraktionsebenen

Verhaltensbeschreibung (behaviour)

Die Verhaltensebene gibt die funktionale Beschreibung des Modells wieder. Es gibt keinen Systemtakt und die Signalwechsel sind asynchron, mit Schaltzeiten beschrieben. Man beschreibt Bussysteme oder komplexe Algorithmen, ohne auf die Synthetisierbarkeit einzugehen.

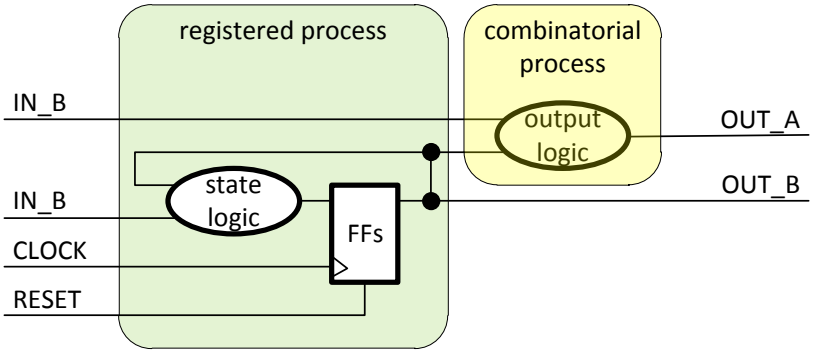


Das Bild zeigt eine einfache Vorgabe für die Funktion eines Moduls. Es soll aus den drei Eingängen I1, I2, I3 ein Ausgangswert (Output) berechnet werden. Des Weiteren ist vorgegeben, dass der Ausgangswert nach maximal 100 ns, nachdem sich die Eingänge geändert haben, stabil anliegen muss. Als Verhaltensbeschreibung in VHDL wird dies zum Beispiel als entsprechende Gleichung (I1 + I2*I3) modelliert. Das Ergebnis der Gleichung wird direkt an das Ausgangssignal mit der angenommenen maximal erlaubten Bearbeitungszeit verzögert (after 100 ns) an den Ausgang übergeben.

Das Verhaltensmodell ist eine einfache Art, das Verhalten einer Schaltung ähnlich den herkömmlichen Programmiersprachen wie PASCAL oder C zu beschreiben.

Register-Transfer-Level (RTL) oder Datenflussbeschreibung

Auf Register-Transfer-Ebene versucht man das System in Blöcken reiner Kombinatorik oder mit Registern zu beschreiben. Letztendlich besteht das Design in der RT-Ebene nur aus zwei unterschiedlichen, nebenläufigen Prozessen (siehe Kap. 8.1): Zum einen der rein kombinatorische Prozess und zum anderen der getaktete Prozess. Der getaktete Prozess erzeugt Flip-Flops und beschreibt letztendlich einen Zustandsautomaten.



Es werden bei der Modellierung also zusätzlich zu den Dateneingängen und Datenausgängen noch die Steuersignale berücksichtigt, wie ein Taktsignal (CLOCK) und gegebenenfalls eine Rücksetzsignal (RESET). Diese Steuersignale sind für die Register notwendig.

Man sieht, dass auf RT-Ebene nicht nur die reine Funktion (logic) beschrieben wird, sondern auch Strukturen (Trennung in speichernde und nichtspeichernde Elemente) und Zeit (in Form der zeitgleichen Aktualisierung durch ein Taktsignal) berücksichtigt werden.

Auf RT-Ebene ist der Takt das entscheidende Merkmal. Alle Operationen werden auf ein Taktsignal bezogen. Die RT-Level Simulation gibt keinen Aufschluss über das reale Zeitverhalten, d.h. ob auch wirklich innerhalb einer Taktperiode alle Signale zur Ruhe gekommen sind.

Logikebene

Auf Logikebene wird das gesamte Design durch Logikgatter und speichernde Elemente beschrieben. Das entspricht einer sog. Netzliste. Hierfür benötigt es eine Zellenbibliothek, in der die einzelnen Gatterparameter (fan-in, fan-out, Verzögerungszeiten, Temperaturabhängigkeiten...) enthalten sind. Den Schritt von der RT-Ebene zur Logikebene übernehmen die Synthesewerkzeuge.

Layout

Die unterste Ebene bildet bei einem ASIC das physikalische Layout der Standard-Zellen, beziehungsweise bei einem FPGA das Mapping der Funktionen auf die vorhandenen Logik-Zellen. Hier werden die verschiedenen elektronischen Bauelemente in der entsprechenden Technologie auf dem Chip verteilt: Bei einem ASIC, indem die entsprechenden Gatter platziert werden (Placing) oder bei einem FPGA, indem aus den frei verfügbaren Gattern diese zugeordnet werden (Fitting).

In einem zweiten Schritt werden dann die so definierten Gatter miteinander verbunden (Routing). Damit sind dann die Leitungslängen und die Leitungsverzögerungen bekannt.

Das Design kann auf dieser Ebene auch wieder simuliert werden, wobei jetzt weniger die Funktionalität als mehr das Zeitverhalten der gesamten Schaltung überprüft werden. Weil mit steigender Information auch der Aufwand für die Simulation stark ansteigt, werden solche „post-fitting“ Simulationen nur begrenzt und sehr gezielt durchgeführt. Statt dieser auch „back-annotated“ genannten Simulation werden Static-Timing-Analysis Werkzeuge verwendet, welche nicht jeden Pfad einzeln simulieren, sondern aus den verschiedenen echten Laufzeiten die gegenseitigen Abhängigkeiten berechnen und überprüfen.

3.3 Beispiel 3-Bit Komparator

Zur Veranschaulichung folgen zwei Beispiele, beide in zwei Versionen. Beide Versionen erzeugen das genau gleiche Verhalten auch wenn sie ev. unterschiedlich in einem programmierbaren Baustein implementiert werden.

Das erste Beispiel beschreibt einen 3-Bit Komparator. Bei gleichem Eingangssignal geht der Ausgang auf logisch 1.

Register-Level Beschreibung (Datenflussbeschreibung)

```
ENTITY ThreeBitComp IS
  PORT (
    a, b : IN  std_logic_vector(0 TO 2);
    eq   : OUT std_logic

    );
END ENTITY ThreeBitComp;


ARCHITECTURE dataflow OF ThreeBitComp IS
BEGIN
    eq <=     NOT(a(0) XOR b(0)
          AND NOT(a(1) XOR b(1)
          AND NOT(a(2) XOR b(2);
END ARCHITECTURE dataflow;
```

Diese Architektur verwendet in VHDL definierte asynchrone Logikbausteine um die Funktion zu beschreiben.

Auch wenn die Definition formell richtig ist, ist sie nicht übersichtlich oder einfach zu verstehen. Der Betrachter ist gezwungen die verschachtelten Logikfunktionen zu studieren um zu erkennen, dass

- Vektoren Bit-für-Bit verglichen werden,
- XOR bei Unterschieden zu ‚1' wird,
- Bitweise XOR Resultat invertiert wird,
- die drei Einzelbit-Resultate am Schluss UND verknüpft werden

Verhaltensbeschreibung (behaviour)

```
ENTITY ThreeBitComp IS
  PORT (
    a, b : IN  std_logic_vector(0 TO 2);
    eq   : OUT std_logic

    );
END ENTITY ThreeBitComp;


ARCHITECTURE behavioral OF ThreeBitComp IS
BEGIN
    compare: PROCESS (a, b)
    BEGIN
        IF a = b THEN
            eq <= '1';
        ELSE
            eq <= '0';
        END IF;
    END PROCESS compare;
END ARCHITECTURE behavioral;
```

Im Gegensatz zur Architektur links wird hier auf einer höheren Ebene Abstrahiert. Die einzelnen Bits werden gar nicht angesprochen, sondern nur die Vektoren a und b als Ganzes.

Sind die Vektoren identisch, ist das Resultat ‚1', und sonst ist es ‚0'.

In VHDL kann diese „IF-THEN-ELSE" Anweisung nur in einem Prozess stehen.

Auch wenn diese Implementation länger ist (mehr Zeilen benötigt) ist sie doch übersichtlicher, leichter zu verstehen und dadurch wartungsfreundlicher.

Allgemein muss ein guter Programmierer immer versuchen, die gewünschte Funktion möglichst einfach, übersichtlich aber auch effizient zu beschreiben. Die Effektive Umsetzung in Hardware-Gatter darf man ruhig dem Synthese-Werkzeug überlassen. Dennoch ist es wichtig, die Synthese bei der Formulierung im Hinterkopf zu halten um nicht beispielsweise irrtümlich das Einfügen von Latches zu verursachen.

3.4 Beispiel 3-Bit Zähler

Das nächste Beispiel beschreibt einen 3-Bit Zähler.

Register-Level Beschreibung (Datenflussbeschreibung)

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;


ENTITY three_bit_counter IS
  PORT (
    clk    : IN  std_logic;
    enable : IN  std_logic;
    count  : OUT unsigned (2 DOWNTO 0);
END ENTITY three_bit_counter;


ARCHITECTURE dataflow OF three_bit_counter
IS
    TYPE t_state IS (s0, s1, s2, s3,
                     s4, s5, s6, s7 );

    SIGNAL state, next_state: t_state;

BEGIN
    combinatorial : PROCESS (state)
    BEGIN
        CASE state IS
              WHEN s0 => next_state <= s1;
                         count <= "001";
              WHEN s1 => next_state <= s2;
                         count <= "010";
              WHEN s2 => next_state <= s3;
                         count <= "011";
              WHEN s3 => next_state <= s4;
                         count <= "100";
              WHEN s4 => next_state <= s5;
                         count <= "101";
              WHEN s5 => next_state <= s6;
                         count <= "110";
              WHEN s6 => next_state <= s7;
                         count <= "111";
              WHEN s7 => next_state <= s0;
                         count <= "000";
        END CASE;
    END PROCESS combinatorial;

    synch : PROCESS (clk)
    BEGIN
       IF rising_edge(clk)
          IF enable ='1' THEN
             state <= next_state AFTER 15 ns;
          END IF;
       END IF;
    END PROCESS synch;

END ARCHITECTURE dataflow;
```

Verhaltensbeschreibung (behaviour)

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

ENTITY three_bit_counter IS
  PORT (
    clk    : IN  std_logic;
    enable : IN  std_logic;
    count  : OUT unsigned (2 DOWNTO 0);
END ENTITY three_bit_counter;


ARCHITECTURE behave OF three_bit_counter IS

    SIGNAL current_count
              : unsigned (2 DOWNTO 0);



BEGIN
    my_comb_proc : PROCESS (current_count)
    BEGIN
        next_count <= current_count;
        IF enable ='1' THEN
            next_count <= current_count + 1;
                          AFTER 15 ns;
        END IF;
    END PROCESS my_comb_proc;

    -- Register Function
    my_reg_proc : PROCESS (clk)
    BEGIN
       IF rising_edge(clk)
        current_count <= next_count
            AFTER 15 ns;
       END IF;
    END PROCESS my_reg_proc;

    -- Output Function
    count <= current_count;

END ARCHITECTURE behave;
```

Auch hier ist die Verhaltensbeschreibung viel übersichtlicher als die Datenfluss-Beschreibung links.

Besonders zu beachten ist das Signal „**`count`**“ welches wegen der Richtung „**`OUT`**“ innerhalb der Architektur selbst nicht gelesen werden kann. Deshalb wurde mit „**`current_count`**“ ein internes Signal erzeugt, welches dann asynchron (das heisst ständig und direkt) auf das Ausgangssignal gespiegelt wird.

4 Datentypen

4.1 Skalare Datentypen

Der Datentyp definiert die möglichen Werte eines Objektes und die Operationen, die mit ihm ausgeführt werden können. So ist es z.B. nicht möglich, einem digitalen Signal einen Integer-Wert zuzuweisen.

Als Standarddatentypen sind in VHDL u.a. definiert:

BOOLEAN:	false / true
BIT:	'0' / '1'
INTEGER:	- 2 147 483 647 bis + 2 147 483 647
REAL:	-1.0E38 bis +1.0E38

```
SIGNAL   done      : BOOLEAN := FALSE;
SIGNAL   a         : BIT := '0';
VARIABLE start_year: INTEGER := 2004;

CONSTANT pi        : REAL := 3.14159;
```

Neben diesen Standarddatentypen bietet VHDL die Möglichkeit, beliebig eigene Datentypen zu definieren. (vergleichbar mit typedef in C/C++)

```
TYPE identifier IS type_definition;
```

```
TYPE my_integer IS RANGE -6 TO 25;
VARIABLE b : my_integer;
```

Mit **RANGE** kann man einen zulässigen Wertebereich angeben. Dies hat einen zweifachen Sinn:

- gerät bei der Simulation oder Compilation der Wert ausserhalb des spezifizierten Bereiches, so wird ein Fehlverhalten des Codes sofort als Fehler sichtbar
- durch die Einschränkung werden bei der Compilation nur die notwendigen Bits implementiert, und die ganze Hardware wird effizienter.

Die Variable b kann damit nur ganzzahlige Werte zwischen -6 und 25 annehmen.

Oder auch

```
TYPE my_real IS RANGE 0.5 TO 15.0;
VARIABLE a : my_real;
```

Durch die Angabe der Bereichsgrenzen in Form von Floating-Point-Werten umfasst der gesamte Bereich REAL-Zahlen innerhalb der gegebenen Grenzen.

Wir verwenden die Gross-Klein-Schreibung gezielt um die Sprachelemente von VHDL gegenüber den frei definierten Namen zu unterscheiden. Dies hilft nicht nur beim Unterricht, sondern macht bei konsequenter Anwendung auch den Code übersichtlicher und dadurch einfacher und eleganter.

In diesem Zusammenhang ist es nur konsequent, wenn Typen wie **BOOLEAN**, **BIT**, **INTEGER** und **REAL** gross geschrieben werden, da sie ein Teil der Sprache von VHDL sind.

Typen wie **std_logic**, **std_logic_vector**, **signed** und **unsigned** sind jedoch NICHT Teil der Sprachdefinition, sondern werden in Packages wie **IEEE.std_logic_1164** oder **IEEE.numeric_std** definiert.

Bei den Aufzählungstypen müssen alle einzelnen zulässigen Elemente dieses Typs explizit aufführt werden

Zum Beispiel könnte ein Steuersignal für eine ALU die Werte *add*, *sub*, *mul* und *div* annehmen.

```
TYPE alu_function IS (add, sub,
                      mul, div);
```

Für die Modellierung digitaler Hardware genügt der Typ BIT nicht. BIT kann für abstrakte Modelle verwendet werden, wo die elektrischen Details keine Rolle spielen.

Im Package **std_logic_1164**, das sich in der Bibliothek **IEEE** befindet, wird ein Typ **std_logic** definiert. Er hat die folgenden 9 definierten Zustände:

0	logische 0
1	logische 1
L	schwach auf 0 gehalten
H	schwach auf 1 gehalten
U	nicht initialisiert
W	schwach, zu unbestimmt
X	nicht bestimmbar
Z	hochohmig
-	don't care

std_logic_vector ist entsprechend ein Vektor aus einzelnen Bits, wobei jedes dieser Bits jeden beliebigen für **std_logic** definierten Zustand annehmen kann.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY test IS
  PORT(
    a, b : IN  std_logic;
    bus  : OUT std_logic_vector
               (7 DOWNTO 0)
  );
END ENTITY test;
```

4.2 Komplexe Typen (composite)

Komplexe Datentypen bestehen aus Werten, die mehrere Elemente beinhalten. Es gibt Arrays und Records.

4.2.1 Arrays

Arrays sind eine Sammlung von durchnummerierten Elemente des gleichen Typs

Benötigt man z.B. einen 4 Bit breiten Datenbus zur Verknüpfung logischer Komponenten, so könnte man einen entsprechenden Typ wie folgt deklarieren:

```
TYPE bus IS ARRAY (0 TO 3) OF BIT;
```

Zu den skalaren Typen sind bereits einige Arraytypen vordefiniert, insbesondere zu den Typen **BIT** und **STD_LOGIC**. Sie tragen den Zusatz **_VECTOR**

```
TYPE bus IS BIT_VECTOR (0 TO 3);

TYPE next_bus IS STD_LOGIC_VECTOR (0 TO 3);
```

Bei der Angabe ganzer Bereiche ist die Laufrichtung der Index' wichtig. Bereiche können aufwärts mit **TO** und abwärts mit **DOWNTO** angegeben werden.

```
SIGNAL a : BIT_VECTOR (0 TO 2);
a <= "011";

-- Dabei ist die Zuweisung
-- a <= "011"; gleichwertig zu

a(0)<='0'; a(1)<='1'; a(2)<='1';
```

Bei den Array-Zuweisungen gibt es sehr viele Möglichkeiten.

- 4-bit String-Zuweisung
- Concatenation (Aneinanderhängen)

```
SIGNAL c          : BIT_VECTOR (0 TO 3);
SIGNAL h, i, j, k : BIT;

c <= "0011";

c <= h & i & j & k;
```

- Aggregat-Zuweisung in
- Verschiedenste Kombinationen der obigen Varianten

```
a <= ('0','0', '1','0')
a <= ('1', i, '0', j OR k)
a <= (0 => '1', 3 => j OR k,
      1 => I  , 2 => '0');
a <= ('1', i, OTHERS => '0')
```

Auch Ausschnitte aus Arrays können adressiert werden.

```
SIGNAL a : BIT_VECTOR (3 DOWNTO 0);
SIGNAL b : BIT_VECTOR (8 DOWNTO 0);
b(6 DOWNTO 3) <= a;
```

4.2.2 Records

Ein **RECORD** besteht aus mehreren Elemente verschiedener Typen. Die einzelnen Felder eines Records werden durch den Elementnamen referenziert. Mit Records können beliebige assoziierte Signale zusammengefasst werden. Man könnte Records als eine Art von „Klasse mit Feldern, ohne Methoden“ betrachten.

```
TYPE alu_function IS (add, sub,
                      mul, div);

TYPE t_alu_input IS RECORD
  data_a    : std_logic_vector (0 TO 7);
  data_b    : std_logic_vector (0 TO 7);
  carry_in  : std_logic;
  operation : t_alu_function;
END RECORD;
```

Signale und Variablen können mit diesem Typ deklariert werden. Es kann auf einzelne Elemente eines Records zugegriffen werden.

```
SIGNAL alu_1_in, alu2_in : t_alu_input;
SIGNAL last_carry        : std_logic;


alu_1_in.data_a    <= X"F9";
alu_2_in.data_a(4) <= '1';
alu_1_in.data_b    <= X"1C";
alu_1_in.carry_in  <= last_carry;
alu_1_in.operation <= sub;
```

Natürlich können auch Objekte eines kompletten **RECORD** -Typs direkt zugewiesen werden.

```
alu_2_in           <= alu_1_in;
```

Mit Records kann man sehr elegant Signale von Schnittstellen zusammenfassen. Dabei ist jedoch unbedingt zu beachten, dass ein **RECORD** jeweils nur die Signale in einer Richtung enthalten darf.

```
TYPE t_hex_output IS RECORD
  sement_3 : std_logic_vector (0 TO 6);
  sement_2 : std_logic_vector (0 TO 6);
  sement_1 : std_logic_vector (0 TO 6);
  sement_0 : std_logic_vector (0 TO 6);
END RECORD;

ENTITY hex_4_digit_driver IS
  PORT (
   number  :IN std_logic_vector(16 DOWNTO 0);
   hex_out :OUT t_hex_output;
END ENTITY hex_4_digit_driver;
```

4.3 Attribute von Datentypen

VHDL kennt auch Signalattribute, die sich auf Arrays anwenden lassen:

`'LEFT`	liefert den Index des am weitesten links liegenden Elementes;
`'RIGHT`	liefert den Index des am weitesten rechts liegenden Elementes;
`'HIGH`	liefert den Index des obersten Elementes;
`'LOW`	liefert den Index des untersten Elementes;
`'LENGTH`	liefert die Länge des Array;
`'RANGE`	liefert den Wertebereich des Index;
`'REVERSE_RANGE`	liefert den Wertebereich des Index in umgekehrter Reihenfolge;

```
PROCESS (clock)
   VARIABLE max_cnt : INTEGER := 0;
BEGIN
   FOR i IN counter'RANGE LOOP
      IF counter(i) > max_cnt THEN
         max_cnt := counter(i);
      END IF;
   END LOOP;
END PROCESS;
```

Der **FOR … LOOP** wird im Abschnitt 8.4 erklärt …

```
TYPE t_data_ram IS ARRAY (0 TO 255)
                   OF INTEGER;

VARIABLE data : t_data_ram;

FOR i IN data'LOW TO data'HIGH LOOP …
```

4.4 Typumwandlungen

Da bei Signalzuweisungen die Signaltypen übereinstimmen müssen (starke Typisierung), werden für verschiedene Typen von Konvertierungsfunktionen benötigt.

Bei eng verwandten Typen (closely related types) reicht für die Typkonvertierung das Voranstellen des Ziel Typs. Eng verwandte Typen sind z.B. Arrays derselben Länge, mit derselben Indexmenge und denselben Elementtypen.

```
TYPE my_byte IS ARRAY (7 DOWNTO 0)
                OF std_logic;

SIGNAL byte   : my_byte;
SIGNAL vector : std_logic_vector
                 (7 DOWNTO 0);

byte <= my_byte(vector);
```

Andernfalls ist eine separate Konvertierungsfunktion notwendig, z.B. zur Umwandlung eines `std_logic_vector` in einen `BIT_VECTOR`. Solche Konvertierungsfunktionen sind z.B. im Package `std_logic_1164` definiert.

```
SIGNAL some_bits : BIT_VECTOR
                   (7 DOWNTO 0);

some_bits <= CONVERT_TO_BIT(vector)
```

4.5 Arithmetik mit Vektoren

VHDL selbst unterstützt keine mathematischen Operationen. Dafür wurden spezielle Packages geschaffen (siehe auch Kapitel 11).

Für synthetisierbare Funktionen die für die Ausführung in FPGAs und/oder ASICs bestimmt sind, verwendet man am besten das Paket **IEEE.numeric_std**, welches die Typen **unsigned** und **signed** als Vektoren vom Typ **std_logic** definiert.

Beim Schreiben von Testbench-Funktionen welche nicht synthetisiert werden darf man ruhig auch auf komplexere mathematische Operationen zurückgreifen, wie Sin, Cos oder Wurzeln. Dafür verwendet man mit Vorteil das Packet **IEEE.math_real** beziehungsweise **IEEE.math_complex**.

Es gibt noch viele alte Beispiele und Implementation, welche die IEEE Pakete **std_logic_arith** sowie **std_logic_signed** und **std_logic_unsigned** verwenden. Die darin definierten Funktionen können mit anderen Definitionen kollidieren und sollten deshalb nicht mehr verwendet werden.

Ebenso sollte das Packet **numeric_bit** nicht verwendet werden, weil es **unsigned** und **signed** als Vektoren vom Typ **BIT** definiert, und dies für reale Schaltungen nicht alle auftretenden Zustände abbilden kann.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

SIGNAL a: UNSIGNED;

a <= a + 10;

IF a < -18 THEN …
```

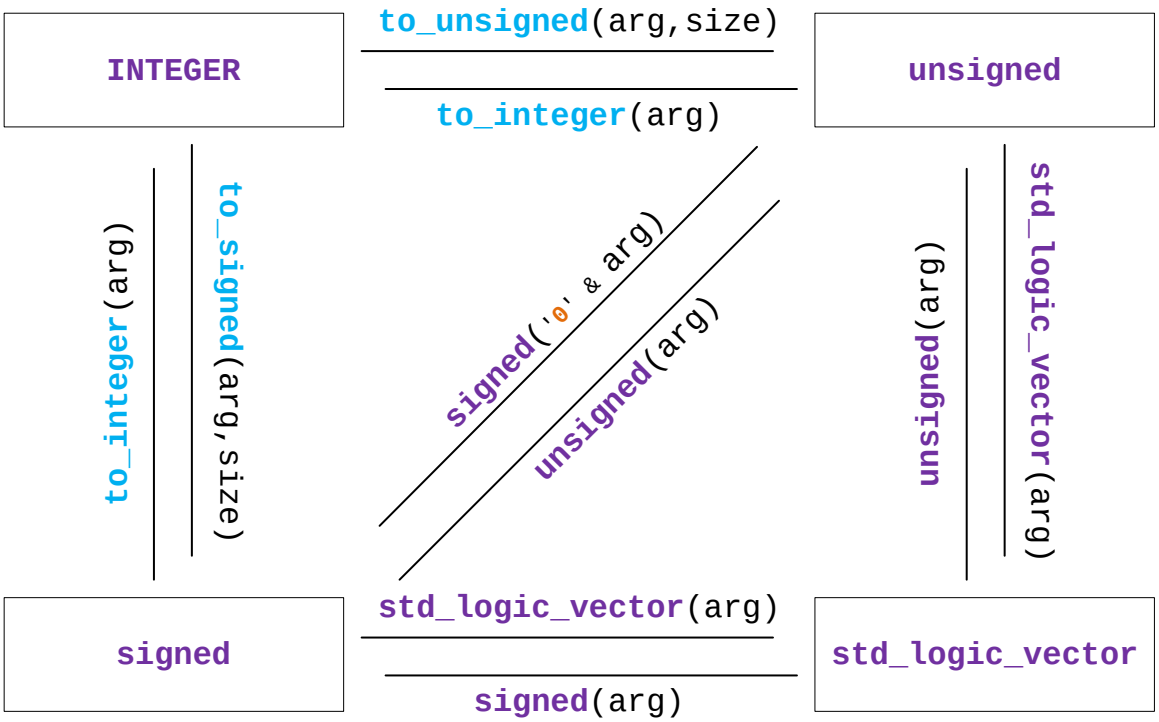
Absichtlicher Widerspruch: Eine Zahl vom Typ „**unsigned**“ kann nicht negativ werden …

Der Compiler wir dies automatisch erkennen und den Benutzer auf den Fehler aufmerksam machen!

4.6 Typenumwandlungen mit Vektoren

Die nachfolgende Tabelle zeigt alle Typumwandlungen im Package NUMERIC_STD.

Quellenoperand	Zieloperand	Funktion
`std_logic_vector`	`unsigned`	`unsigned(arg)`
`std_logic_vector`	`signed`	`signed(arg)`
`unsigned`	`std_logic_vector`	`std_logic_vector(arg)`
`signed`	`std_logic_vector`	`std_logic_vector(arg)`
`INTEGER`	`unsigned`	`to_unsigned(arg, size)`
`INTEGER`	`signed`	`to_signed(arg, size)`
`unsigned`	`INTEGER`	`to_integer(arg)`
`signed`	`INTEGER`	`to_integer(arg)`
`INTEGER`	`std_logic_vector`	Zuerst die `integer` Zahl in `unsigned` oder `signed` wandeln, dann in `std_logic_vector`
`std_logic_vector`	`INTEGER`	Zuerst `std_logic_vector` in `signed` oder `unsigned` wandeln, dann weiter zum Typ `INTEGER`
`unsigned + unsigned`	`std_logic_vector`	`std_logic_vector(arg1 + arg2)`
`signed + signed`	`std_logic_vector`	`std_logic_vector(arg1 + arg2)`



5 Operatoren

Die folgende Tabelle führt alle Klassen von Operatoren nach Prioritäten geordnet (von oben nach unten) auf.

NOT	**	ABS				
*	/	MOD	REM			
- (Negation)						
+	-	&				
SLL	SRL	SLA	SRA	ROL	ROR	SRL
=	/=	<	<=	>=	>	/=
AND	OR	NAND	NOR	XOR	XNOR	OR

Achtung: Für eine Signalzuweisung, z.B. c <= ‘0’ und für den relationalen Vergleich „kleiner gleich“ wird das gleiche Zeichen verwendet wird. Der Compiler erkennt aus dem Kontext des Aufrufs, welche Bedeutung das <= an einer bestimmten Stelle hat.

Relationale Operatoren sind nicht für jeden Datentyp deklariert. So kann man in Standard-VHDL einen „<“ -Operator (kleiner) nur für Objekte des Typs `INTEGER` und `REAL` benutzen.

Logische Operatoren sind für die Datentypen `BIT` (`'0'`, `'1'`), `std_logic` (`'0'`, `'1'`) und `BOOLEAN` (`FALSE, TRUE`) definiert.

```
c <= a AND NOT b;
```

`NOT` bindet stärker als `AND`, also ist keine Klammer nötig.

```
IF (a + 3) = 100 THEN ...
```

Das „+“ bindet stärker als =, also ist hier eigentlich keine Klammer nötig. Klammern können jedoch gezielt eingesetzt werden, um die Lesbarkeit zu erhöhen.

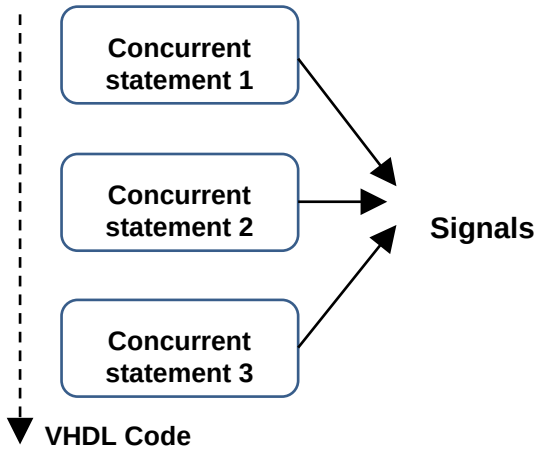
```
IF ((a OR b) = c) THEN ...
```

`OR` bindet schwächer als =, also ist hier eine Klammer nötig, und macht einen funktionellen Unterschied.

6 Parallel laufende Anweisungen

Innerhalb einer Architektur laufen alle Prozesse parallel (concurrent) ab. Die Reihenfolge der einzelnen Anweisungen hat weder auf das Verhalten noch auf die resultierende Hardware einen Einfluss hat.

Es ist nicht möglich, mit nebenläufigen Anweisungen alleine speichernde Elemente, wie z.B. Flip Flops zu modellieren. Allerdings kann umgekehrt jede concurrent Anweisung durch einen entsprechenden Prozess beschrieben werden.



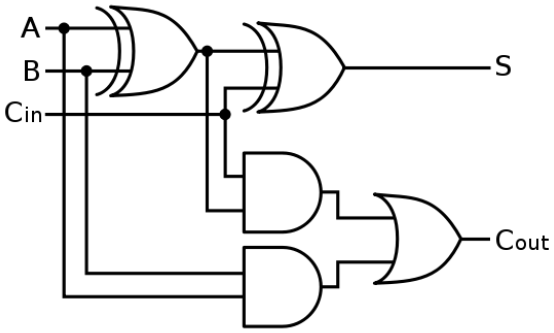
6.1 Signalzuweisung

Eine Signalzuweisung (Signal Assignment) ist ein Concurrent Statement; sie wird also gleichzeitig mit anderen Signalzuweisungen abgearbeitet.

target <= value;

Damit ein solches Statement abläuft, muss ein Ereignis (Änderung eines Signalwertes) rechts vom Zuweisungszeichen anliegen.

VHDL ermöglicht eine direkte Beschreibung des logischen Verhaltens auf der Grundlage von Wahrheitstabellen und Booleschen Gleichungen. Diese Art des Modellierens bezeichnet man als Datafluss (dataflow) oder RTL-Technik (Register Transfer Level). Da es sich hierbei um eine sehr hardwarenahe Art des Modellierens handelt, muss man sich recht genau über die darzustellende Hardware im Klaren sein.



Das Beispiel rechts zeigt die Architektur eines 1-Bit-Addierers als exakte Nachbildung der Netzliste, die einen solchen Addierer aus logischen Grundgattern aufbaut. Ein internes Signal ***sig*** dient für die Verbindung eines XOR-Gatters mit einem AND- und einem zweiten XOR-Gatter.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY one_bit_adder IS
    PORT (
        a,b,c_in  : IN  std_logic;
        sum,c_out : OUT std_logic
    );
END ENTITY one_bit_adder;

ARCHITECTURE rtl OF one_bit_adder IS
   SIGNAL sig : std_logic;
BEGIN
   sig   <= a XOR b;
   sum   <= int XOR c_in;
   c_out <= (a AND b)
            OR (sig AND c_in);
END ARCHITECTURE rtl;
```

7 Signale und Variablen

7.1 Deklaration von Signalen

Signale verbinden einzelne Elemente (z.B. Prozesse, Komponenten) – sie sind quasi Leitungen. Signale werden wie folgt deklariert:

```
SIGNAL signal_name : TYPE
                [: = init_val];
```

In der Syntax der Signaldeklaration kann man dem Signal optional einen definierten Anfangswert mitgeben.

Geschieht das nicht, wie in nebenstehendem Beispiel, so ist der Anfangswert des deklarierten Objektes bei einer FPGA-Implementation abhängig von seinem Datentyp:

- bei sogenannten skalaren Datentypen mit definierten Wertebereichen gilt dann automatisch der kleinste Wert
- bei **INTEGER** als skalarer Datentyp wird es zu -2 147 483 647 (= $-2^{31} + 1$)
- Bei **REAL** wird dies zu $-1 \cdot 10^{308}$
- bei Enumerationen, die eine explizite Aufzählung der einzelnen Werte des Datentyps darstellen, kommt der erste Wert in der Liste zur Anwendung.
- Bei **std_logic** wird der Wert zu „U“ für „Unknown“ (unbekannt). Das gleiche gilt auch für **std_logic_vector**.
- Bei **BIT** ist der Default Wert '0',
- bei **BOOLEAN** ist es **FALSE**

Im Gegensatz zu einem FPGA (wo alle Register automatisch initialisiert werden) wird bei einem ASIC ein nicht definiertes Signal nicht einfach „0“, sondern kann tatsächlich einen zufälligen Wert annehmen.

Deshalb ist es IMMER eine sehr gute Idee, alle Signale explizit mit einem Anfangswert zu definieren!

```
SIGNAL sel: std_logic := '1';

SIGNAL count : std_logic_vector
               (7 DOWNTO 0)
             := "00001111";

TYPE t_state IS (waiting, running,
                 dead);

SIGNAL my_state : t_state;

IF my_state = waiting THEN …
```

Dies liefert in einem FPGA das Resultat **TRUE** für ersten Durchlauf, da bei einem Aufzählungs-Typ der erste Wert als Default angenommen wird.

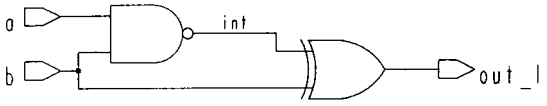
Signale können nur an zwei Stellen deklariert werden:

1. In der Port-Deklaration einer Entity.

2. Innerhalb der *Declarative Region* einer Architektur.

Die so genannten *Declarative Region* einer Architektur befindet sich nach dem Schlüsselwort **ARCHITECTURE … IS**, aber noch vor dem **BEGIN** Befehl.

```
ENTITY xyz IS
    PORT (
        clock : IN  std_logic;
        led   : OUT std_logic;
        q     : OUT std_logic
    );
END ENTITY xyz;
```




```
ARCHITECTURE rtl OF concurrent IS
   SIGNAL int : std_logic;
BEGIN
   int <= a NAND b;
   out_1 <= int XOR b;
END ARCHITECTURE rtl;
```

7.2 Verzögerungszeiten

Man kann für alle Signale auch das zeitliche Verhalten simulieren. VHDL kennt dazu mehrere Statements zur Erweiterung einer Signalzuweisung. Das häufigste ist die **AFTER** Anweisung.

Die **WAIT FOR** Anweisung ist Speziell für Test-Benches praktisch, um zeitliche Abläufe zu simulieren.

Wichtig: Beide Anweisungen (**AFTER** und **WAIT FOR**) gelten nur für die Simulation und lassen sich **NICHT** synthetisieren!

Die Anweisungen dienen lediglich dazu, die Simulation näher an das spätere, reale Verhalten in einer Schaltung anzupassen.

```
c <= a NOR b AFTER 5 ns;
```

Bei dieser Signalzuweisung nimmt c den Wert der NOR-Verknüpfung von a und b genau 5 ns nach einer Änderung (einem Event) an a oder b an.

```
in_a <= '0'; in_b <= '0';
WAIT FOR 10 ns;

in_a <= '0'; in_b <= '1';
WAIT FOR 10 ns;

in_a <= '1'; in_b <= '0';
WAIT FOR 10 ns;

in_a <= '1'; in_b <= '1';
WAIT FOR 10 ns;
```

7.3 Attribute von Signalen

Neben den Attributen für Datentypen (siehe Kapitel 4.3) gibt es in VHDL auch signalgebundene Attribute.

Signalattribute liefern Informationen über Signale, auf die sie angewendet werden. Das Format eines Standardaufrufes sieht folgendermaßen aus:

```
object_name ' attribute_designator
```

VHDL kennt viele Signalattribute. Ein häufig verwendetes ist `signal'EVENT`.

Es liefert **`TRUE`**, wenn in der aktuellen Δt-Periode ein Ereignis an **`signal`** anliegt. Mit der folgenden Abfrage könnte man beispielsweise überprüfen, ob das Signal **`clock`** innerhalb eines Prozesses einen Übergang von `'0'` auf `'1'` erfahren hat:

```
IF (clock'event AND clock = '1') THEN …
```

Eine zweite, elegantere Möglichkeit bietet sich mit der Funktion `rising_edge()` aus der Bibliothek `IEEE.std_logic_1164`.

Taktflankengesteuertes D-Latch mit Abfrage eines Signalattributes:

```
PROCESS (clk)
BEGIN
   IF clk = 'l' AND clk'EVENT THEN
      q <= d;
   END IF;
END PROCESS;
```

Das gleiche mit der Funktion `rising_edge`:

```
PROCESS (clk)
BEGIN
   IF rising_edge(clk) THEN q <= d;
   END IF;
END PROCESS;
```

Auf den ersten Blick erscheinen die beiden Möglichkeiten mit `clk = 'l' AND clk'EVENT` und `rising_edge` identisch.

Tatsächlich gibt es aber einen Unterschied, wenn man die effektive Implementation der Funktion betrachtet:

Bei der Funktion `rising_edge` wird nicht nur geprüft, ob der Takt einen `EVENT` hat und auf `'l'` steht, sondern auch, ob das Signal vorher auf `'0'` stand. Entsprechend wird eine Signaländerung von `'H'` auf `'l'` NICHT als steigende Flanke interpretiert!

Dieser kleine Unterschied kann später unliebsame Differenzen zwischen Simulation und echter Hardware bewirken …

Deshalb ist die Verwendung der Funktion `rising_edge` nicht nur eleganter, sondern auch technisch besser.

7.4 Variablen

In einem sequentiellen Code müssen Zuweisungen einer Zeile sehr oft in der nächsten Zeile schon zur Verfügung stehen, da es keine Iterationen gibt. Der Einsatz von Signalen ist hier also nicht immer sinnvoll. Daher bietet VHDL mit den Variablen eine weitere Objektklasse, der sofort ohne Zeitfaktor ein neuer Wert zugewiesen werden kann. Durch diese Eigenschaft sind Variablen speziell für die Verwendung in sequentiellem Code geeignet.

Variablen werden bei jedem Prozess-Aufruf neu „geboren“, und gelten nur bis zum Ende der Prozessbearbeitung. Sie existieren nicht ausserhalb eines Prozesses und können deshalb z.B. auch nicht in der *Declarative Region* einer Architektur definiert werden.

Variablen sollten nur für Zwischenresultate verwendet werden. Den Wert einer Variablen aus einem Prozess-Durchgang im nächsten Durchgang wieder weiter zu verwenden (z.B. für einen Zähler) ist eine sehr schlechte Praxis, weil man dabei riskiert dass ein Latch implementiert wird. Das führt dann zu dramatisch unterschiedlichen Resultaten zwischen Simulation und dem Timing in Hardware! Die Folge ist eine schlechte Performanz der Schaltung.

```
one_bit_add : PROCESS (a,b,c_in)

    VARIABLE sig : std_logic;

BEGIN
   sig := a XOR b;

   sum <= sig XOR c_in;
   c_out <= (a AND b)
            OR (sig AND c_in);
END PROCESS one_bit_add;
```

Das Zuweisung einer Variablen (:=) unterscheidet sich von der Zuweisung eines Signals (<=). Dies macht den Code lesbarer.

In diesem Beispiel wird die Variable **`sig`** bei jedem Durchgang zu Beginn auf `a XOR b` gesetzt.

Wenn der Prozess fertig abgearbeitet ist darf die Variable **`sig`** ihren Wert wieder „vergessen“ weil dieser nicht mehr benötigt wird.

7.5 Konstanten

Konstanten legen einmalig Werte innerhalb einer Deklarationseinheit fest.

Es ist eine gute Idee, möglichst alle statischen Zahlenwerte am Anfang des VHDL Codes als Konstanten zu definieren, statt über den ganzen Code verteilt „magic numbers“ zu haben.

Dadurch können diese Zahlen automatisch mit einem griffigen Namen dokumentiert werden, was den Code lesbarer macht. Zusätzlich muss im Falle eines späteren Wartungseingriffs nur noch eine einzelne Stelle geändert werden was die Fehleranfälligkeit senkt.

```
ARCHITECTURE rtl OF vga_driver IS

    CONSTANT vga_width
          :  unsigned(9 DOWNTO 0)
          := to_unsigned(1024, 10);

    CONSTANT vga_heigth
          :  unsigned(9 DOWNTO 0)
          := to_unsigned(768, 10);

BEGIN …
```

8 Sequentielle Anweisungen

Will man VHDL in einem Top-down-Designablauf einsetzen, so müssen Modelle ohne irgendwelche Vorstellungen über die entsprechende Hardware auf einer sehr hohen Abstraktionsebene dargestellt werden können.

8.1 Der Prozess

Nachdem im VHDL-Architekturbereich zwischen **BEGIN** und **END** nur „concurrent“ gearbeitet werden darf, schuf man in der VHDL-Syntax besondere Bereiche mit sequentiellem internem Ablauf.

Die ganze Architektur kann mit einem oder mehreren VHDL-Konstrukten gefüllt sein, die jeweils mit dem Schlüsselwort **PROCESS** beginnen und mit **END PROCESS** abgeschlossen werden.

Ein Prozess wird in seiner Gesamtheit parallel mit anderen abgearbeitet, innerhalb des Bereiches zwischen **BEGIN** und **END PROCESS** allerdings läuft der Code grundsätzlich nur sequentiell ab. In diesem Bereich können abstrakte Algorithmen beschrieben werden, wie sie aus anderen Hochsprachen bekannt sind.

```
ARCHITECTURE algorithm OF my_nor IS
BEGIN
   comb_proc : PROCESS
      VARIABLE int : std_logic;
   BEGIN
      IF (a = 'l') OR (b = 'l') THEN
         int := 'l';
      ELSE
         int := '0';
      END IF;
      c <= NOT int;
      WAIT ON a,b;
   END PROCESS comb_proc;
END ARCHITECTURE algorithm;
```

Algorithmisches Modell eines NOR-Gatters. Dieses Architekturbeispiel enthält nur einen Prozess.

Der Prozess startet sofort, und hält kurz vor dem Ende bei der **WAIT ON** Anweisung an. Sobald sich a oder b ändert, wird die **WAIT** Anweisung verlassen, und der Prozess beendet und sofort wieder neu begonnen.

8.1.1 Aktivierung für den logischen Ablauf

Ein Prozess muss durch eine Änderung eines oder mehrerer seiner Eingangssignale angestossen oder aktiviert werden. Dies passiert durch die Angabe von Signalen in seiner Sensitivitätsliste oder eines **WAIT** Befehls innerhalb des Prozesses. Ein Prozess kann aber nur entweder eine Sensitivitätsliste oder ein **WAIT**-Statement enthalten, nicht beides.

Dabei ist es wichtig festzuhalten, dass diese Angabe der Aktivierung primär für die Simulation nötig ist.

Die Sensitivitätsliste entspricht einem WAIT ON … am Ende des Prozesses.

Gleiche Funktion, unterschiedlicher Code:

```
my_adder_1 : PROCESS (a, b);
BEGIN
   q <= a + b;
END PROCESS my_adder_1;
```

```
my_adder_2 : PROCESS;
BEGIN
   q <= a + b;
   WAIT ON a, b;
END PROCESS my_adder_2;
```

Dieser Code ist NICHT synthetisierbar!!!

Obwohl die Funktion gleich bleibt, ist die erste Variante übersichtlicher und daher zu empfehlen.

Die zweite Variante ist auch nur für die Simulation brauchbar, sonst nicht.

Es gibt 4 Grundformen der **WAIT** Anweisung.

- Unterbrechen des Prozesses bis ein Signalwechsel passiert.
- Unterbrechen des Prozesses bis eine Bedingung erfüllt ist.
- Unterbrechen des Prozesses bis eine Zeitspanne verstrichen ist
- Unendlich langes Warten. Da ein Prozess immer aktiv ist, bietet diese Anweisung die einzige Möglichkeit ihn anzuhalten (z.B. nach der Initialisierungen)

```
WAIT ON a, b, c;

WAIT UNTIL x > 10;

WAIT FOR 10 ns;

WAIT;
```

Die Signale selber erhalten ihren Ausgangswert erst am Ende des Prozesses (im Falle einer Sensitivitätsliste) oder bei Erreichen eines **WAIT**-Statements.

Aus diesem Grund kann einem Signal innerhalb des Prozesses mehrfach ein Wert zugewiesen werden, weil die Zuweisung erst am Ende ausgeführt wird. Andererseits erklärt dies auch, warum die „Zwischenresultate“ von Signalen, anders als bei Variablen, nicht zur Verfügung stehen.

```
interesting : PROCESS (a, b, c)
BEGIN
    x <= '0';
    IF (a NAND b) THEN
        x <= '1';
    END IF;
END PROZESS interesting;
```

Der Wert von x wird hier NICHT überschrieben, da er erst am Ende des Prozesses zugewiesen wird.

```
the_same : PROCESS (a, b, c)
BEGIN
    IF (a NAND b) THEN
        x <= '1';
    ELSE
        x <= '0';
    END IF;
END PROZESS the_same;
```

Da ein Prozess als Ganzes eine gleichzeitige Anweisung ist (concurrent), kann der Modellierer beim Entwerfen von Modellen weiterhin in Blöcken denken. Damit lässt sich die Beschreibung eines komplexen Systems in übersichtliche funktionelle Blöcke aufteilen. Diese „concurrent“ Blöcke werden gleichzeitig abgearbeitet und können wie in einem Blockschaltbild über Signale miteinander verbunden werden.

Siehe dazu das Beispiel auf der rechten Seite. Zwei Prozesse simulieren eine Erzeuger / Verbraucher Situation. Über ein einfaches Handshake-Protokoll (**producer_done** und **consumer_done**) werden die Prozesse synchronisiert. Dabei wird angenommen, dass die Tätigkeit des „produce“ und „consume“ eine bestimmte Zeit oder Zyklen braucht.

Nachdem sequentieller Code immer nur zeilenweise von Anfang bis Ende abgearbeitet wird, muss er mit anderen Mitteln als durch Signaländerungen parallel gesteuert werden. Zu diesem Zweck gibt es in VHDL Konstrukte wie IF-Statements oder Schleifen.

```
ARCHITECTURE test OF prod_cons IS
  SIGNAL producer_done : std_logic
                       := '0';
  SIGNAL consumer_done : std_logic
                       := '1';
BEGIN
  producer : PROCESS
  BEGIN
    producer_done := '0';
    WAIT UNTIL consumer_done = '1';
    …       -- produce
    producer_done := '1';
    WAIT UNTIL consumer_done = '0';
  END PROCESS producer;

  consumer : PROCESS
   BEGIN
      consumer_done := '0';
      WAIT UNTIL producer = '1';
      …           -- consume
      consumer_done := '1';
      WAIT UNTIL producer_done ='0';
   END PROCESS consumer;
END ARCHITECTURE test;
```

8.1.2 Aktivierung für die Simulation

Im Gegensatz zur echten Hardware wie z.B. in einem FPGA oder ASIC muss bei der Simulation ein Prozess durch die Änderung eines Signals der Sensitivity-Liste aktiviert werden.

Dabei kann es wesentliche Unterschiede im Verhalten geben, wenn nicht alle notwendigen Signale in der Sensitivity-Liste aufgeführt sind.

Bei einem Latch soll der Eingang auf den Ausgang kopiert werden solange das Enable Signal hoch ist. Ist das Enable Signal tief, soll der letzte Zustand des Eingangs gehalten werden.

Von den drei Beispielen rechts funktioniert nur das dritte Modell richtig. Bei den ersten beiden wird der Ausgang sich nicht ändern, wenn Enable hoch ist, und der Eingang sich ändert weil der Prozess durch das Eingangs-Signal „in“ nicht aktiviert wird!

```
latch_1 : PROCESS (enable);
BEGIN
    q <= in;
END PROCESS latch;
```

Der Prozess wird nur bei einer Änderung von „enable“ ausgeführt!

```
latch_2 : PROCESS;
BEGIN
   q <= in;
   WAIT ON enable
END PROCESS latch;
```

Der Prozess wartet hier, bis sich „enable“ ändert. Erst dann wird er erneut ausgeführt.

```
latch_3 : PROCESS (enable, in);
BEGIN
    IF enable = '1' THEN
        q <= in;
    END IF;
END PROCESS latch;
```

Signalzuweisungen innerhalb eines Prozesses werden nicht sofort wirksam, sondern erst im folgenden Simulationszyklus (siehe Kapitel 8.1.1). Für einen Prozess mit einer Sensitivitätsliste ist dies am Ende des Prozesses, für einen Prozess ohne Sensitivity-Liste aber mit einem **WAIT**-Statement ist das vor dem Erreichen des **WAIT** -Statements.

```
tripple_ff_1 : PROCESS (clock);
BEGIN
    IF rising_edge(clock) THEN
        a <= input_bit;
        b <= a;
        c <= b;
    END IF;
END PROCESS latch;
```

Hier wird immer noch der alte Wert von „a“ und „b“ verwendet!!!

Daraus folgt, dass z.B. Signale in einem Prozess nicht wie Variablen (siehe Kapitel 7.6) als Zwischenspeicher für Werte benutzt werden können. Muss mit dem Signalwert gerechnet werden, kann man den Wert des Signals in einer Variablen zwischenspeichern, mit dieser Variablen arbeiten und am Schluss den neuen Wert an das Signal zuweisen.

```
and_bus : PROCESS (input_vector);
   VARIABLE x: std_logic;
BEGIN
   x := '1';
   FOR i IN 7 DOWNTO 0 LOOP
      x := input_vector(i) AND x;
   END LOOP;
   out <= x;
END PROCESS and_bus;
```

Wegen dieser speziellen Eigenschaften der Signalzuweisung kommt es oft zu Fehlern.

```
swap_signals : PROCESS (x, y);
BEGIN
    x <= y;
    y <= x;
    WAIT 100 ns;
END PROCESS swap_signals;
```

Werte von x und y werden alle 100 ns vertauscht.

8.2 Das IF-Statement

Ist die Bedingung nach dem **IF** nicht erfüllt, werden, wenn vorhanden, die Bedingung nach **ELSIF** geprüft (alternative zu ELSE IF um die Verschachtelung in Grenzen zu halten). Treffen diese ebenfalls nicht zu, so wird das Statement nach dem **ELSE** ohne weitere Überprüfung ausgeführt (falls vorhanden). Eine beliebige Anzahl von **ELSIF**-Statements darf vorkommen.

ELSE steht nur einmal am Ende des **IF** Statements oder kann ganz weggelassen werden.

Als einfaches Beispiel zeigt ein Prozess, der ein NAND-Gatter algorithmisch beschreibt.

```
ENTITY alg_nand IS
   PORT (
      a, b : IN  std_logic;
      c    : OUT std_logic
   );
END ENTITY alg_nand;


ARCHITECTURE rtl OF alg_nand IS
BEGIN
   nand_proc : PROCESS (a, b)
   BEGIN
      IF a = '1' AND b = '1' THEN
         c < = '0';
      ELSIF a = '0' OR b = '0' THEN
         c < = '1';
      ELSE
         c < = 'X';
      END IF;
   END PROCESS nand_proc;
END ARCHITECTURE rtl;
```

8.3 Das Case-Statement

Eine weitere Möglichkeit für die Steuerung von sequentiellem Code ist das **CASE** Statement.

Das Statement **CASE ... IS** überprüft den Zustand eines Signales und führt eine bestimmte Aktion abhängig vom Ergebnis aus. Das **CASE**-Statement muss vollständig auflösbar sein, das bedeutet, jedem möglichen Zustand (Datentyp und gegebenenfalls Wertebereich beachten!) des geprüften Ausdrucks muss eine Aktion zugewiesen werden. **WHEN OTHERS** dient dazu, sämtliche noch nicht abgefragten Zustände zu erfassen und diesen eine gemeinsame Aktion zuzuweisen (analog dem „default“-Statement in Java).

Im Beispiel rechts könnte man denken, dass mit den 4 Kombinationen alle Möglichkeiten bereits abgedeckt sind, und dass es deshalb das **WHEN OTHERS** nicht braucht.

Da aber `sel` vom Typ `std_logic` ist, kann das Signal auch Zustände wie „`UU`“ oder „`X1`“ annehmen.

```
ENTITY mux_4_to_1 IS
   PORT (
      a,b,c,d : IN  std_logic;
      sel     : IN  std_logic_vector
                    (1 DOWNTO 0);
      out     : OUT std_logic
   );
END ENTITY mux_4_to_1;


ARCHITECTURE rtl OF mux_4_to_1 IS
BEGIN
   mux_proc : PROCESS (a,b,c,d,sel)
   BEGIN
      CASE sel IS
         WHEN "00"   => out <= a;
         WHEN "01"   => out <= b;
         WHEN "10"   => out <= c;
         WHEN "11"   => out <= d;
         WHEN OTHERS => out <= 'X';
      END CASE;
   END PROCESS mux_proc;
END ARCHITECTURE rtl;
```

8.4 Die FOR-Schleife

Die generelle Syntax für die **FOR**-Schleife ist wie folgt:

```
[loop_label:]  FOR identifier IN
discrete_range LOOP
     {sequential_statements}
END LOOP [loop_label];
```

Die Laufvariable in der **FOR**-Schleife ist implizit deklariert, d.h. sie muss nirgendwo sonst im Code als Variable definiert werden. Die Variable nimmt automatisch den Datentyp der Elemente des angegebenen Wertebereichs an.

Damit die **FOR**-Schleife synthetisierbar ist, müssen die Grenzen des Wertebereichs konstant und zum Zeitpunkt der Compilation bekannt sein.

Es ist auch ganz wichtig zu verstehen, dass die **FOR**-Schleife keine sequentiell ausgeführte Logik beschreibt oder impliziert. Es ist nichts anderes als Art parallel auszuführenden Code mit regelmässiger Struktur effizient zu beschreiben.

Beide Beispiele rechts erzeugen die gleiche Logik und brauchen exakt gleich viel Zeit für die Ausführung!

Zwei Beispiele für einen Prozess, der die Reihenfolge (12 Bits) eines Vektors umkehrt. Die beiden Kodierungen sind absolut gleichwertig, aber unterschiedlich elegant …

```
reverse_proc_1 : PROCESS (input)
BEGIN
   output(11) <= input(0);
   output(10) <= input(1);
   output(9)  <= input(2);
   output(8)  <= input(3);
   output(7)  <= input(4);
   output(6)  <= input(5);
   output(5)  <= input(6);
   output(4)  <= input(7);
   output(3)  <= input(8);
   output(2)  <= input(9);
   output(1)  <= input(10);
   output(0)  <= input(11);
END PROCESS reverse_proc_1;
```

```
reverse_proc_2 : PROCESS (input)
  CONST bus_width :INTEGER := 11;
BEGIN
  FOR i IN 0 TO bits LOOP
    output(bus_width - i)
         <= input(i);
  END LOOP;
END PROCESS reverse_proc_2;
```

Im Codebeispiel ist i vom Typ **INTEGER**; i ist nur innerhalb der **FOR**-Schleife bekannt.

8.5 Die WHILE-Schleife

```
[loop_label:] WHILE condition
LOOP
      {sequential_statements}
END LOOP [loop_label];
```

Hier steht nach dem Schlüsselwort **WHILE** eine Bedingung, die vor Beginn jedes einzelnen Schleifendurchlaufs abgefragt wird. Der Durchlauf erfolgt abhängig vom Ergebnis dieser Abfrage. Anders als bei der **FOR**-Schleife muss man hier die Laufvariable ausserhalb der Schleife deklarieren. Auch die Änderung der Laufvariablen sowie der Abbruch müssen explizit gesteuert werden. Hier kann man also Endlosschleifen aufbauen, die nach dem Aufruf nie mehr verlassen werden.

```
ARCHITECTURE sim OF my_cosinus IS
BEGIN

   PROCESS (angle)
      VARIABLE sum, term : REAL;
      VARIABLE n         : INTEGER;
   BEGIN
      sum  := 1.0;
      term := 1.0;
      WHILE ABS term > ABS(sum/1.0E6)
      LOOP
         n := n + 2;
         term := (-term) * angle**2 /
                  real(((n-1)*n));
         sum := sum + term;
      END LOOP;
      result <= sum;
   END PROCESS;

END ARCHITECTURE sim;
```

Da die **WHILE** Schleife ihre Flexibilität aus der dynamisch zu bestimmenden Abbruchbedingung bezieht, ist sie NICHT synthetisierbar. Daher sollte man diesen Befehl nur für die Simulation oder temporäre Modellierung von Modulen verwenden welche später im Verlauf des Projektes durch synthetisierbaren Code ersetzt werden!

Dieser Prozess berechnet den Cosinus mit einer Reihenentwicklung. Dieser Code ist jedoch aus verschiedenen Gründen nicht synthetisierbar:
- **WHILE LOOP** mit dynamischen Grenzen
- Verwendung von **REAL** Werten
- Mathematische Division an zwei Stellen

8.6 Schleifen mit LOOP ... EXIT

```
[loop_label:] LOOP
      {sequential_statements}
      EXIT [loop_label] [WHEN
expr.]
END LOOP [loop_label];
```

Die Schleife wird durch das EXIT-Statement verlassen. Im Beispiel ist EXIT mit einer Bedingung verknüpft, der Befehl kann allerdings vor der Bedingung zusätzlich einen Label-Namen als Sprungziel enthalten oder auch ganz alleine stehen.

```
PROCESS (in_x)
   VARIABLE i: INTEGER := 0;
BEGIN
   LOOP
      i := i + 1;
      out_y(i) <= in_x(i);
      EXIT WHEN i = 8;
   END LOOP;
END PROCESS;
```

Auch diese Art der **LOOP** Konstruktion ist NICHT synthetisierbar. Es gelten die gleichen Anwendungs-Beschränkungen wie bei der **WHILE** Schleife.

8.7 Der NEXT-Befehl

Mit einem **NEXT**-Statement wird der aktuelle Durchlauf einer Schleife abgebrochen und die nächste Iteration begonnen (analog „continue“ in Java).

Wie auch schon die „normale“ **LOOP**-Schleife ist diese Konstruktion mit dem **NEXT**-Abbruch synthetisierbar, wenn der Wertebereich der Schleife zum Zeitpunkt der Kompilation bekannt und konstant ist.

Der **NEXT**-Abbruch muss nicht statisch sein, sondern wird einfach als eine Bedingte Ausführung für einen Teil des Codes implementiert, als ob dort z.B. eine **IF** Anweisung die entsprechenden Zeilen umgeben würde.

```
PROCESS (a)
   VARIABLE a, b : unsigned
                   (5 DOWNTO 0);
BEGIN
   a := (OTHERS => '0');
   b := (OTHERS => '0');
   loop_1 : FOR i IN 0 TO 5 LOOP
      a := a + 5;
      IF a > 20 THEN NEXT loop_1;
      END IF;
      b := b + 5;
   END LOOP loop_1;
END PROCESS;
```

Wird das **NEXT**-Kommando durch das **IF**-Statement erreicht, so wird die Schleife nicht fertig abgearbeitet; stattdessen beginnt sofort der nächste Schleifendurchgang.

"`b := b+5`" wird also nicht mehr ausgeführt, wenn a größer als 20 ist, so dass am Ende der Schleife a den Wert 25 und b den Wert 20 hat.

8.8 Beispiel für sequentiellen Code mit Schleifen

Der abgebildete Prozess dekodiert den binären 4 Bit breiten Eingang **in_array** derart, dass in einem 16 Bit breiten Ausgangsfeld **out_array** die Leitung gesetzt wird, die dem dezimalen Wert des Eingangs entspricht. Liegt am Eingang **in_array** also z.B. der Binärwert "0101" an, so wird der Ausgang **out_array** (5) auf '1' gesetzt. Natürlich wäre diese Aufgabe mit anderen Sprachkonstrukten wesentlich eleganter umsetzbar. Das Beispiel eignet sich jedoch auch gut zur Demonstration von Schleifen.

Liegt an **in_array** ein Ereignis vor, so wird in der ersten Schleife eine Variable **sum** auf den dezimalen Wert des binären Eingangs gesetzt. Jedes Element von **in_array** muss dabei vom Datentyp **INTEGER** sein, damit es multipliziert werden kann. Die Elemente werden darum in einen **INTEGER** -Typ umgewandelt.

Bei diesem Beispiel ist auch zu beachten, dass die Variable **sum** am Beginn des Prozesses auf 0 gesetzt werden muss, obwohl sie bereits während der Initialisierung zurückgesetzt wurde! Dies ist notwendig, da sich der Prozess nach einem erneuten Aufruf bei einer Zustandsänderung von **in_array** nur noch zwischen **BEGIN** und **END** bewegt, so dass keine erneute Initialisierung mehr erfolgt.

Beispielcode eines 4-zu-1 Decoders

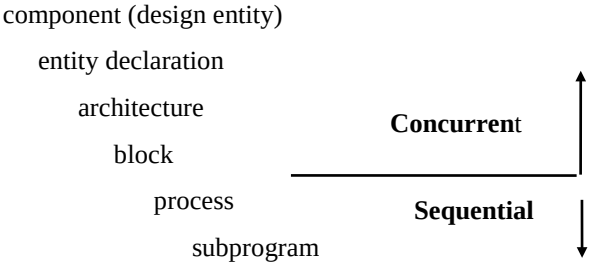
```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
ENTITY dec_4_1 IS
  PORT(
    in_array : IN std_logic_vector
                  ( 3 DOWNTO 0);
    out_array: OUT std_logic_vector
                  (15 DOWNTO 0));
END dec_4_1;


ARCHITECTURE behavior OF dec_4_1 IS
BEGIN

  PROCESS (in_array)
    VARIABLE sum : INTEGER
                   RANGE 0 TO 15;
  BEGIN

    sum := 0;
    sum_of_in_array: FOR i IN 0 TO 3
    LOOP
      IF in_array(i) = '1' THEN
          sum := sum + (2**i);
      END IF;
    END LOOP sum_of_in_array;

    zero_array : FOR j IN 0 TO 15
    LOOP
      out_array(j) <= '0';
    END LOOP zero_array;
    out_array(sum) <= '1';

  END PROCESS;
END ARCHITECTURE behavior;
```

9 Gültigkeitsbereich von Deklarationen

In VHDL bestimmt eine Deklaration die Bedeutung eines Objektes. Deklarationen werden in den *Declarative Regions* von VHDL-Elementen gemacht, meistens zwischen dem Namen eines Elementes und **BEGIN**. Nicht alle Arten von Objekten lassen sich in jedem Bereich deklarieren. So kann man in einer Architektur keine Variable deklarieren, in einem Prozess kein Signal.

Der Ort einer Objektdeklaration hat Einfluss auf den Gültigkeitsbereich, den sogenannten *Scope* der Deklaration. Prinzipiell ist ein Objekt im Bereich der verwendeten *Declarative Region* gültig. Ein Objekt, das innerhalb einer Architektur deklariert wurde, ist auch innerhalb jedes Prozesses oder Unterprogrammes im Inneren dieser Architektur bekannt.

Hierarchie der Gültigkeitsbereiche in VHDL



Die Sichtbarkeit (*Visibility*) eines Objektes beschreibt die Bereiche, in denen der Zugriff auf ein gültiges Objekt möglich ist.

Im Inneren des Prozesses ist die in der Architektur definierte Konstante x zwar gültig, aber nicht sichtbar. Sie wird von der deklarierten Variable x verdeckt. Code innerhalb des Prozesses wird also immer nur die Variable x sehen.

```
ARCHITECTURE test_1 OF example IS
   -- global definition
   CONSTANT x: INTEGER := 2;

BEGIN
   p1 : PROCESS (in)
   --  local definition
      VARIABLE x : INTEGER := 3;
   BEGIN
      ...
   END PROCESS p1;
END ARCHITECTURE test;
```

Man kann über *Selected Names* trotzdem auf nicht sichtbare Objekte zugreife. Stellt man vor den Objektnamen den Namen des Deklarationsabschnittes, in dem das Objekt deklariert wurde, so ist es für den Compiler sichtbar.

```
ARCHITECTURE test_2 OF example IS
   -- global definition
   CONSTANT x: INTEGER := 2;

BEGIN
   p2 : PROCESS (in)
   --  local definition
      VARIABLE x : INTEGER := 3;
   BEGIN
      out_1 <= test2.x * in;
      out_2 <= p2.x * in;
   END PROCESS p2;
END ARCHITECTURE test_2;
```

10 Unterprogramme

Es gibt in VHDL zwei Möglichkeiten Unterprogramme zu definieren:

FUNCTION mit genau einem Rückgabewert

PROCEDURE mit keinem, oder mehreren Rückgabewerten.

10.1 FUNCTION Unterprogramm

Eine Funktion hat meist mehrere Parameter und gibt genau einen Wert zurück. Damit ist der Aufruf einer Funktion von der Syntax her wie ein Ausdruck.

Eine Funktion kann eine wiederkehrende Gruppe von zusammenhängenden Operationen elegant zusammenfassen und dadurch den Code übersichtlicher gestalten.

Eine Funktion muss mindestens eine **RETURN** Anweisung enthalten, aber diese muss nicht zwingend am Ende stehen.

Funktionen dürfen keine **WAIT** Befehle verwenden.

```
FUNCTION limit (
   value : unsigned(15 DOWNTO 0);
   min   : unsigned(15 DOWNTO 0);
   max   : unsigned(15 DOWNTO 0)
) RETURN unsigned IS

BEGIN
  IF    value > max THEN RETURN max;
  ELSIF value < min THEN RETURN min;
  ELSE  RETURN value;
  END IF;
END FUNCTION limit;
```

Für indexierten Typen (wie `std_logic_vector`, `unsigned` und `signed`) darf beim der **RETURN** Definition **kein** Bereich stehen!

Die Funktion muss in der Declarative Region einer Architektur stehen, also im Bereich nach dem Schlüsselwort **ARCHITECTURE** und noch vor dem Schlüsselwort **BEGIN**.

Soll eine Funktion allgemein verfügbar sein, so kann sie statt in einer Architektur auch in einem **PACKAGE BODY** definiert werden. Siehe dazu Kapitel 11.

Innerhalb einer Architektur wird eine Funktion wie eine Zuweisung aufgerufen.

```
ranged_val <= limit(input,min,max);
```

10.2 PROCEDURE Unterprogramm

Benötigt man ein Unterprogramm mit mehreren Rückgabewerten, so muss man statt einer Funktion eine **PROCEDURE** verwenden.

Prozeduren sind vor allem bei der Programmierung von Test-Benches sehr hilfreich.

Wie man am Beispiel rechts sehen kann, bietet eine **PROCEDURE** viel mehr Flexibilität bei der Wahl der Ein- und Ausgangssignale, und kann **WAIT** Befehle verarbeiten.

```
PROCEDURE generate_clock (
   CONSTANT t_period : IN  TIME;
   CONSTANT t_pulse  : IN  TIME;
   CONSTANT t_phase  : IN  TIME;
   SIGNAL   clk_out  : OUT std_logic
) IS

BEGIN
  WAIT FOR t_phase;
  LOOP
     clk_out <= '1';
     clk_out <= '0','1' AFTER t_pulse;
     WAIT FOR t_period;
  END LOOP;
END PROCEDURE generate_clock;
```

Da Prozeduren keine Rückgabewerte liefern, werden sie ähnlich wie Module aufgerufen. Dabei gibt es verschiedene Möglichkeiten der Signalübergabe.

```
SIGNAL clock_1    : std_logic;
SIGNAL clock_2    : std_logic;
SIGNAL clock_3    : std_logic;
```

Die erste Variante ist am kürzesten aber auch anfällig für Fehler wenn man die Reihenfolge der Signale nicht genau richtig macht.

```
generate_clock(10 ns, 5 ns, 2 ns,
               clock_1);
```

Variante 2 ist viel übersichtlicher, wobei auch hier die Zuordnung der Signale über die Reihenfolge garantiert ist. Mit dem Zusatzaufwand der Kommentare wird es lesbar, aber es können sich unerkannt Fehler bei der Zuordnung einschleichen.

```
gen_clk_2 : generate_clock (
    10 ns,     --  Periode
     5 ns,     --  Pulse Länge
     2 ns,     --  Phase
     clock_2   --  Ausgang
);
```

Bei der Variante 3 geschieht die Zuordnung explizit über die Port-Namen. Verwechslungen sind ausgeschlossen, und aussagekräftige Namen der Signale liefern gleichzeitig auch eine gute Dokumentation. Diese Vorgehensweise ist auch robust gegenüber einer Änderung der Reihenfolge der Parameter.

```
gen_clk_3 : generate_clock (
    t_period => 10 ns,
    t_pulse  =>  5 ns,
    t_phase  =>  2 ns,
    clk_out  => clock_3
);
```

11 Bibliotheken und Packages

Benötigt man für VHDL-Modelle wiederholt bestimmte Typen, Objekte oder typspezifische Operatoren, so bietet sich die Verwendung von Bibliotheken an. Die entsprechenden Definitionen kann man dort in kompilierter Form abspeichern. Bibliotheken lassen sich dann in jedes VHDL-Modell einbinden und sind dann dort bekannt. Solche gemeinsamen Definitionen werden in *Packages* gemacht, die aus einer *Declaration* und einem *Body* bestehen. Gäbe es in diesem Package keine Funktion, so wäre auch kein *Body* erforderlich. Der Vorteil dieser Aufteilung: Bei einer Änderung an der Funktion muss nur der *Body* neu kompiliert werden.

```
--  function header only
PACKAGE mean3_pkg IS
   FUNCTION mean (a, b, c: REAL)
                  RETURN REAL;
END PACKAGE mean3_pkg;

--  function definition
PACKAGE BODY mean3_pkg IS
   FUNCTION mean (a, b, c: REAL)
                  RETURN REAL IS
   BEGIN
      RETURN (a+b+c) / 3.0;
   END FUNCTION mean;
END PACKAGE mean3_pkg;
```

Das Statement **LIBRARY** definiert den logischen Namen der Bibliothek. Er entspricht üblicherweise dem physikalischen Verzeichnis im Dateisystem des Rechners wo die Bibliothek gefunden werden kann.

Neben benutzerdefinierten Bibliotheken gibt es zwei Standardbibliotheken:

WORK: Default-Bibliothek des Projekts.

STD: Bibliothek mit vordefinierten Datentypen und Funktionen.

Wichtig ist v.a. die Bibliothek **IEEE**. Darin sind Typen und Funktionen für digitale Signale enthalten.

Das USE-Statement bestimmt, welches Package aus einer Bibliothek und welche Deklarationen aus einem Package eingebunden werden sollen. Das **ALL** Schlüsselwort legt fest, dass alle Deklarationen des Packages verwendet werden sollen. Natürlich könnte hier auch nur der Name einer bestimmten Deklaration aus dem Package stehen. Dann wäre auch nur diese Deklaration innerhalb des Modells bekannt. Damit kann man gezielt Überschneidungen und Kollisionen zwischen verschiedenen Packages vermeiden.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

LIBRARY IEEE;
USE IEEE.std_logic_1164.std_logic;
```

Oder mit dem Beispiel von oben:

```
LIBRARY work;
USE work.myPackage.ALL;

ENTITY average IS
   PORT(in_l, in_2, in _ 3: IN REAL;
   output : OUT REAL);
END average;

ARCHITECTURE pkg_example OF average
IS
BEGIN
  output <= mean( in_1, in_2, in_3);
END pkg_example;
```

12 Hierarchie durch strukturierte Modelle / Komponenten

Hierarchisches Design ist eine Technik um komplexe und Umfangreiche Elemente eines Designs in sinnvolle Gruppen zusammen zu fassen, und so für das Verständnis zu abstrahieren.

Wenn man vom Ganzen beginnt und es immer weiter in detailliertere Module aufteilt, nennt man das „top-down design“. Verwendet man so gebildete Module nicht nur einmal sondern immer wieder, gewinnt man an Effizienz, spart Zeit und vermeidet Flüchtigkeitsfehler.

Eine hierarchische Ebene welche keine RTL Logik besitz, sondern nur bestehende Module enthält und diese miteinander verbindet nennt man „Structure“ oder kurz **`struct`**. In der Regel ist die höchste Eben (top-level) von diesem Typ.

Ein einfaches Beispiel ist rechts gezeigt. Ein 4-Bit Addierer **`adder4`** wird aus vier einzelnen Ein-Bit Volladdierern **`full_add`** aufgebaut. In der Architekturbeschreibung von **`adder4`** wird zuerst eine Komponente **full_add** für die Entity **`full_add`** deklariert.

Es ist eine Eigenheit von VHDL, dass jedes Modul sowohl eine **`ENTITY`** wie auch eine (identische) **`COMPONENT`** Definition benötigt.

Die Spezifikation dieser **`COMPONENT`** Definition könnte als Teil von **`add_full`** in ein eigenes Package **`add_full_pkg`** integriert werden. Damit stünde sie auch für weitere Anwendungen zur Verfügung und müsste nicht mehr explizit im Code eingefügt werden. Stattdessen würde die Einbindung über die USE-Klausel erfolgen..

In diesem Beispiel hier wurde stattdessen die Komponente in der *Declarative Region* der Architektur von **`adder4`** platziert.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY full_add IS
  PORT (
    u, v, carry_in : IN  std_logic;
    sum, carry_out : OUT std_logic
  );
END full_add;

ARCHITECTURE rtl OF full_add IS
BEGIN
  sum       <= u XOR v XOR carry_in;
  carry_out <= (u AND v)
               OR ((u OR v)    AND
carry_in);
END ARCHITECTURE rtl;


--  ================================


LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY adder4 IS
  PORT(
    a, b : IN  std_logic_vector
               (3 DOWNTO 0);
    cin  : IN  std_logic;
    s    : OUT std_logic_vector
               (3 DOWNTO 0);
    cout : OUT std_logic);
END adder4;

ARCHITECTURE struct OF adder4 IS
  COMPONENT full_add
    PORT (
      u,v,carry_in  : IN  std_logic;
      sum,carry_out : OUT std_logic
    );
  END COMPONENT;

  SIGNAL c : std_logic_vector
             (0 DOWNTO 3);

BEGIN
A0: full_add PORT MAP (
      a(0), b(0), cin,  s(0), c(0));
  A1: full_add PORT MAP (
      a(1), b(1), c(0), s(1), c(1));
  A2: full_add PORT MAP (
      a(2), b(2), c(1), s(2), c(2));
  A3: full_add PORT MAP (
      a(3), b(3), c(2), s(3), cout);
END ARCHITECTURE struct;
```

13 Parametrisierbare Modelle

Alle Beispiele bis anhin zeigten eine in der Portbeschreibung fixierte Schnittstelle. VHDL stellt uns darüber hinaus einen Mechanismus zur Verfügung, wie allgemeinere (generische) Modelle formuliert werden können.

Als häufigste Anwendung davon wird eine Schnittstellenbeschreibung mit dem Schlüsselwort **GENERIC** parametrisiert. Dabei kann dem Parameter **width** gleich bei der Deklaration ein Wert zugewiesen werden. Diese *Generics* können im Code wie Konstanten verwendet werden. Sie bieten aber den Vorteil, dass sich bei der Integration „überschrieben“ werden können.

```
ENTITY generic_register IS
   GENERIC (
      width : positive := 16
   );

   PORT (
      d : IN  std_logic_vector
              (width-1 DOWNTO 0);
      q : OUT std_logic_vector
              (width-1 DOWNTO 0)
      );
END ENTITY generic_register;
```

Eine Parametrierbare Komponente macht nur dann sinn, wenn die Parametrierung kurz vor der Verwendung der Komponente erfolgen kann. VHDL bietet daher die Möglichkeit zum Festlegen der Parameter zum Zeitpunkt der Instanziierung einer Komponente.

Das nebenstehende Beispiel zeigt, wie die Komponente **generic_register** in einem weiteren VHDL-Modul verwendet werden kann. Dabei wird bei der Instanziierung die ursprüngliche Definition von **width** mit dem Wert aus der Konstante **bus_width** (**32**) überschrieben.

```
ARCHITECTURE struct OF example IS
    CONSTANT bus_width : INTEGER
                       := 32;
    SIGNAL input  : std_logic_vector
              (bus_width-1 DOWNTO 0);
    SIGNAL output : std_logic_vector
              (bus_width-1 DOWNTO 0);
BEGIN
   reg0 : ENTITY generic_register
   GENERIC MAP (width => bus_width)
   PORT MAP (
      d => input,
      q => output
   );
  …
END ARCHITECTURE struct;
```