OST
Ostschweizer
Fachhochschule

Einführung in VHDL

Eine Einführung in die Schaltungsentwicklung mit VHDL

Autoren:	Prof. Laszlo Arato und Prof. Dr. Urs Graf
Version:	2020-1

Erstellt am:	Juli 2014
Letzte Änderung am:	08. Sep. 2020

Zum Teil wurden Beispiele und Illustrationen entnommen aus:

- «VHDL Kompakt», http:// tams-www.informatik.uni-hamburg.de
- «The Student's Guide to VHDL», P. Ashenden
- «VHDL Tutorial», https://www.vhdl-online.de/

OST
Ostschweizer
Fachhochschule

Änderungsnachweis

Version	Änderungsgrund	Kurz-Z.	Datum
2020-1	Format-Anpassung an die OST, und viele kleine Korrekturen	ARAL	08.09.2020

Inhaltsverzeichnis

1 Einführung in VHDL 5
2 Grundstruktur eines VHDL-Modells und 1. Beispiel 7
2.1 Elemente des VHDL-Modells 7
2.1.1 Kommentare 7
2.1.2 Die Entity 7
2.1.3 Die Port_Liste 8
2.1.4 Port Modi 8
2.1.5 Die Architektur 9
2.2 Packages und Libraries 10
2.2.1 Vollständiges Beispiel 10
2.3 Konfiguration 11
3 Datentypen 12
3.1 Vorgegebene Datentypen 12
3.1.1 VHDL Definierte Datentypen 12
3.1.2 Freie Definition von Typen 12
3.1.3 Aufzählungstypen 12
3.1.4 Standard Logic 13
3.2 Komplexe Typen (composite) 13
3.2.1 Arrays 14
3.2.2 Records 15
3.3 Attribute von Datentypen 16
3.4 Typenumwandlungen 16
3.5 Typenumwandlungen mit Vektoren 17
4 Konzepte 18
4.1 Grundlagen 18
4.2 Abstraktionsebenen 19
4.3 Beispiel 3-Bit Komparator 21
4.4 Beispiel 3-Bit Zähler 22
5 Operatoren 24
5.1 Arithmetik mit Vektoren 25
6 Parallellaufende Anweisungen (Concurrent) 26
6.1 Signalzuweisung 26
7 Signale und Variablen 27
7.1 Deklaration von Signalen 27
7.2 Verzögerungszeiten 28
7.3 Attribute von Signalen 29
7.4 Variablen 30
7.5 Konstanten 30
8 Sequentielle Anweisungen 31
8.1 Der Prozess 31
8.1.1 Aktivierung für den logischen Ablauf 32

8.1.2	Ausgangswert bei einem Prozess	32
8.1.3	Aktivierung für die Simulation	33
8.1.4	Ausführung von Signalzuweisungen	34
8.2	Das IF-Statement	35
8.3	Das Case-Statement	35
8.4	Die FOR-Schleife	36
8.5	Die WHILE-Schleife	38
8.6	Schleifen mit LOOP ... EXIT	38
8.7	Der NEXT-Befehl	39
8.8	Beispiel für sequentiellen Code mit Schleifen	40
9	**Gültigkeitsbereich von Deklarationen**	**41**
10	**Unterprogramme**	**42**
10.1	FUNCTION Unterprogramm	42
10.2	PROCEDURE Unterprogramm	43
10.2.1	PROCEDURE Beispiel	43
10.2.2	Beispiele für PROCEDURE Aufruf	43
11	**Bibliotheken und Packages**	**44**
11.1	PACKAGE	44
11.2	LIBRARY	44
12	**Hierarchie durch strukturierte Modelle / Komponenten**	**45**
13	**Parametrisierbare Modelle**	**46**

OST
Ostschweizer
Fachhochschule

Hinweis zur Benutzung dieses Leitfadens:

In den folgenden Kapiteln zeigt in der Regel die linke Spalte stets Theorie und Erklärung, während in der rechten Spalte dazu passende Beispiele aufgezeigt werden.

Da wir die nächsten 2 Semester sehr viel mit Xilinx «Vivado» als Werkzeug für VHDL und FPGA arbeiten werden, richtet sich die farblichen Markierungen der Code- Beispiele nach dem Xilinx Farbschema:

VHDL Code, wie auch zitierte Schlüsselwörter im Text sind im Font «Courier New» gehalten.

VHDL Schlüsselwörter wie z.B. **`ENTITY`** und **`BEGIN`** sind zusätzlich fett und lila markiert.

Boole'sche Ausdrücke wie «**`FALSE`**» und «**`null`**» sind fett und rosarot markiert.

String Ausdrücke in doppelten Anführungszeichen **`"any string"`** sind fett und dunkelblau.

Kommentare in VHDL beginnen mit « **`-- Remark`** » und sind fett und hellgrau markiert.

Label wie z.B Namen der Module und Prozesse wie «**`signal_generator : PROCESS`**» sind fett und hellgrün. Leider wird diese Farbgebung für Labels in Vivado zurzeit noch nicht wirklich konsequent angewendet … die grüne Färbung ist daher wenig aussagekräftig.

VHDL Schlüsselwörter sind grossgeschrieben, damit sie besser von den Namen für Signale, Variablen und Funktionen unterschieden werden können.

Man könnte auch die Signalnamen grossschreiben, und die VHDL Schlüsselbegriffe klein, den VHDL selbst achten nicht auf Gross/Kleinschreibung …
… aber viele Simulationswerkzeug ändern alle Signalnamen zu Kleinschreibung. Deshalb machen «Camel Case» Namen auch keinen Sinn – was dann im Code noch gut lesbar wäre (z.B. «**`myStableSignal`**») wird zu einem unübersichtlichen Ausdruck «**`mystablesignal`**». Besser sind Underscore _ zum Trennen der Namen, wie z.B. «**`my_stable_signal`**».

OST
Ostschweizer
Fachhochschule

1 Einführung in VHDL

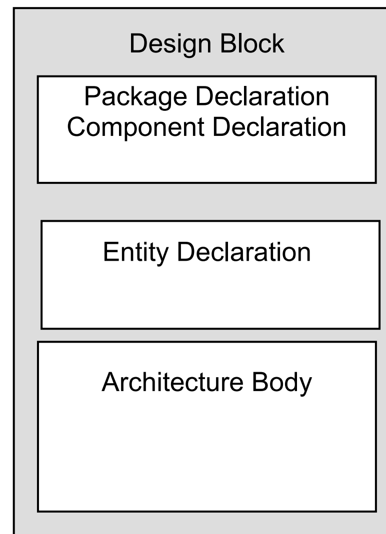
Ziel dieser Einführung	Diese Einführung soll die wesentlichen Konzepte von VHDL aufzeigen. Anhand praktischer Beispiele wird der Gebrauch dieser Beschreibungssprache erläutert. Diese Einführung ist keine komplette Referenz aller VHDL-Konstrukte. Weiterführende Konzepte wie z.B. generische Module, GENERATE und Overloading wurden weggelassen.
„VHDL“	**V**ery High-Speed Integrated Circuit **H**ardware **D**escription **L**anguage. Mit VHDL kann man digitale Schaltungen und Systeme durchgängig von der Systemdefinition bis zum Schaltungsentwurf beschreiben und verifizieren. Die weltweit einzige Alternative ist «Verilog», eine syntaktisch sehr C-ähnliche Sprache. Während VHDL als typenstarke Sprache die Syntax sehr stark auf korrekte Zuweisungen prüft, erlaubt Verilog sehr wilde, unkontrollierte Zuweisungen zwischen verschiedenen Datentypen. Die Welt ist heute grob 50-50 zwischen VHDL und Verilog geteilt, wobei Verfechter von Verilog dessen Effizienz bei der Codierung preisen – zum Nachteil von Klarheit und Lesbarkeit.
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.
Bottom-Up Design	VHDL unterstützt auch die Entwicklung von einzelnen Teilmodulen, ohne dass bereits das ganze Design existieren muss. Schritt für Schritt können dann einzelne Blöcke und Hierarchien hinzugefügt werden. Auf diese Weise kann man sich zuerst auf die grössten Probleme stürzen, und wenn man weiss wie diese gelöst sind, kann man den Rest anhängen.
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 Teil der Sprache VHDL kann wirklich effizient synthetisiert werden (z.B. Datentypen `STD_LOGIC`, `STD_LOGIC_VECTOR`, `SIGNED` und `UNSIGNED`). Andere Elemente werden zwar von einem Synthesewerkzeug umgesetzt, aber erzeugen ineffiziente Strukturen (z.B. für Datentypen `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 «Testbenches». Da dieser Code immer nur simuliert und nie synthetisiert wird, darf man hier nach Belieben alle Sprachelemente und Konstrukte von VHDL verwenden.
VHDL 87	1981 erliess das Militär (DOD) einen Aufruf zur Standardisierung einer Hochsprache zur Beschreibung von Hardware. Die Industrie und Tool-Entwickler wurden schon früh ins Projekt eingebunden, und hatten noch vor der Herausgabe des Standards schon Zeit, die Entwicklungswerkzeuge zeitgerecht zur Verabschiedung des Standards bereitzustellen. Im Jahre 1987 war es dann soweit, VHDL wurde 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 (und auch schon VHDL 2002) wurde die Sprache um ein paar praktische Elemente erweitert.
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.

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 der Bereich **ARCHITECTURE** den inneren Aufbau und die Funktion des Systems. Die **ARCHITECTURE** entspricht daher der Netzliste bei der grafischen Beschreibung. Während eine Architektur immer genau **einer** bestimmten **ENTITY** zugeordnet ist, kann eine Entity 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 Kommentar mit
--  Fortsetzung auf Zeile zwei
```

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][entity_name];
```

Entity_name bezeichnet den Namen der Einheit. Diesen Namen referenziert dann jeweils das nächst hörere System.
Die *port_list* bezeichnet eine Liste von Ein- und Ausgangssignalen des Systems.
Bei «`END ENTITY entity_name;`» sind das Schlüsselwort «**ENTITY**» und auch der Name der Entity optional.

2.1.3 Die Port_Liste

Bei der PORT Liste handelt es sich um eine durch Strichpunkt getrennte Liste von Eingangs- und Ausgangssignalen. Jedes Signal wird durch einen Bezeichner, den Modus und Datentyp spezifiziert.
Der Strichpunkt trennt nur die einzelnen Elemente der Liste, und deshalb hat das letzte Element (output_1) am Ende **keinen** Strichpunkt!
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.
Zusätzlich muss jeweils auch der Typ des Signals angegeben werden. Für synthetisierbare Logik ist das typischerweise nur `STD_LOGIC` für Bit, und `STD_LOGIC_VECTOR`, `SIGNED` und `UNSIGNED`, für mehrere Bits, zusammen mit der Angabe der Zahl der Bits.

```
ENTITY entity_name IS
  PORT (
    in_1 : <modus> <type>;
    in_2 : <modus> <type>;
    out  : <modus> <type>
  );
END [ENTITY][entity_name];
```

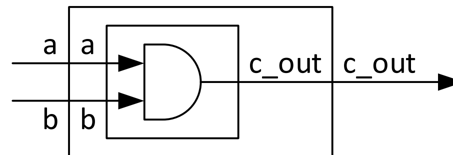
Signale werden ausführlich im Kapitel 7 «Signale und Variablen» behandelt.

2.1.4 Port Modi

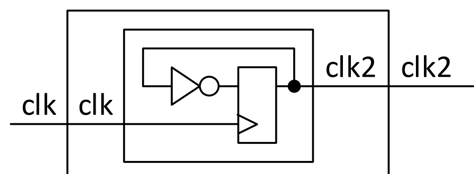
Der Modus gibt für jeden Port die Richtung der Daten vor.
Der Modus `IN` wird für rein lesbare Signale verwendet. Signale, deren Ports diesen Mode haben, können in der untergeordneten Architektur keine Werte zugewiesen werden. Entsprechend bezeichnet der `OUT` Modus Signale, denen ein Wert zugewiesen werden kann, aber die nicht gelesen werden können.

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 für spätere Kompatibilität, weil Ports mit Modus `BUFFER` nicht mit anderen Ports verbunden werden können!!!




```
PORT (
      a, b    : IN  STD_LOGIC;
      c_out   : OUT STD_LOGIC
);
```




```
PORT (
      clk    : IN     STD_LOGIC;
      clk2   : BUFFER STD_LOGIC
);
```

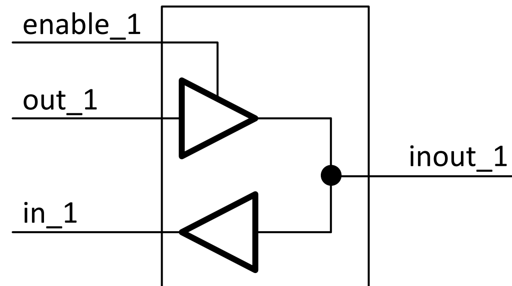
Wir werden diesen Modus nicht verwenden!!!

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 Sinn Signale an der Peripherie eines Chips, wie zum Beispiel für die bidirektionalen Daten bei der I2C Schnittstelle!

Innerhalb eines FPGAs gibt es aus technischen Gründen **keine** echten Tri-State Signale, alles wird auf Signale mit nur einem zugelassenen Treiber und «H» oder «L» Pegel abgebildet.

Bei einem ASIC ist es möglich echte interne Tri-State Signale und Busse zu definieren … aber dann muss man GANZ GENAU wissen was man tut – weil dann unter anderem keine automatische Optimierung und keine statische Timing-Analyse mehr möglich ist.




```
PORT (
     in_1     : IN    STD_LOGIC;
     enable_1 : IN    STD_LOGIC;
     inout_1  : INOUT STD_LOGIC
);
```

2.1.5 Die Architektur

Die Architektur beschreibt das innere Verhalten eines Moduls.

Die Architektur beginnt mit dem Schlüsselwort **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.

Bei «**END ARCHITECTURE** arch_name;» sind das Schlüsselwort «**ARCHITECTURE**» und auch der Name der Architektur optional.

```
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 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;
```

Das IEEE (Institue of Electrical and Electronics Engineers) hat viel zur Standardisierung von VHDL beigetragen.
«.`STD_LOGIC_1164`» definier Signaltypen für Bits und Bit-Vektoren, während die Bibliothek « `numeric_std`» die Beste Definition für Zahlen wie «`SIGNED`» und «`UNSIGNED`» ist.

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

Statt dem VHDL Befehl `NAND` nehmen wir jedoch bewusst verschiedene explizite Implementationen.

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 meinen.

Der Signal-Typ „`STD_LOGIC`“ dient, anders als der Typ `BIT`, nicht nur zur Darstellung von logisch `'1'` und `'0'`, sondern auch 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. 3.1.4). Was passiert, 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.

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

ENTITY nand_gate IS
    PORT (
        a       : STD_LOGIC;
        b       : STD_LOGIC;
        c_out   : STD_LOGIC
    );
END;


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

Also definieren wir eine bessere 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.

Natürlich kann man auch direkt den VHDL Befehl NAND nehmen – und es macht in der Regel alles richtig.

Das Beispiel zeigt nicht, wie man Funktionen aus VHDL auch umständlich definieren kann, sondern soll nur als einfaches Beispiel für die Auswirkung der effektiven Implementation auf das Verhalten bei unvorhergesehenen Signalen aufzeigen.

Gerade das Verhalten bei unspezifizierten Eingangssignalen wird oft vergessen.

```
ARCHITECTURE smart OF nand_gate IS
  --  Smart truth table
  --  =================
  --
  --     a     b    |  c_out
  --   ------------|--------
  --     0     X    |    1
  --     X     0    |    1
  --     1     1    |    0
  --     Other      |    X
BEGIN
  PROCESS (a, b)
  BEGIN

    IF    a = '0' OR b = '0' THEN
          c_out <= '1';
    ELSIF a = '1' AND b = '1' THEN
          c_out <= '0';
    ELSE  c_out <= 'X';
    END IF;

  END PROCESS;
END ARCHITECTURE smart;


ARCHITECTURE dataflow OF nand_gate
IS
BEGIN
    c_out <= a NAND b;
END ARCHITECTURE dataflow;
```

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.

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

Leider hat es sich in Versuchen gezeigt, dass weder ModelSim noch FPGA Tools den Befehl CONFIGURATION beachten, sondern immer die LETZTE Architektur im File verwendet wird.
Deshalb werden wir im Kurs VHDL auf dieses Feature verzichten.

Will man mehrere Architekturen verwenden, empfiehlt sich statt dem CONFIGURATION Befehl die Verwendung von je einem File pro Architektur. Dadurch kann bei sowohl bei ModelSim, wie auch z.B. bei Quartus und Vivado in der Liste der Design-Files einfach das File eingebunden werden, welches die gerade gewünschte Architektur enthält. Man kann dazu die ENTITY und ARCHITECTURE
Definition je getrennt in Files abspeichern, und ins Projekt einbinden.

3 Datentypen

3.1 Vorgegebene 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.

3.1.1 VHDL Definierte Datentypen

Als Standard 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 := TRUE;
SIGNAL   a    : BIT := '0';
VARIABLE year : INTEGER := 2004;
CONSTANT pi   : REAL := 3.14159;
```

3.1.2 Freie Definition von Typen

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

```
TYPE <identifier> IS <def>;
```

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

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

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

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

Oder bei Fliesskommazahlen:

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

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

3.1.3 Aufzählungstypen

Aufzählungstypen sind sehr praktisch für die Zustände von endlichen Automaten (FSM) oder Leitungen. Dazu müssen alle 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);
```

Es kann sinnvoll sein, eigene Namen der Zustände gross zu schreiben, um diese von Signalen zu unterscheiden.

3.1.4 Standard Logic

Für die Modellierung digitaler Hardware genügt der Typ **BIT** nicht. Typ **BIT** kann für abstrakte Modelle verwendet werden, wo die elektrischen Details keine Rolle spielen. Für reale physikalische Signale kann es mehr Zustände geben als nur «**0**» und «**1**».

Im Package **STD_LOGIC_1164**, welches sich in der Bibliothek **IEEE** befindet, wird der Typ **STD_LOGIC** definiert. Er hat die folgenden neun 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

```
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;
```

Entsprechend ist **STD_LOGIC_VECTOR** ein Vektor aus einzelnen Bits, wobei jedes dieser Bits als **STD_LOGIC** definiert ist und Bit-für-Bit jeden der neun Zustände annehmen kann.

Beim Design verwendet man nur Zustände **0**, **1** und **Z**.

U und **X** sind besonders bei der Simulation wichtig, um nicht initialisierte Signale und Kollisionen zu erkennen.

L, **H**, **W** und **-** werden heute kaum mehr verwendet, und machen nur bei ASIC Verifikationen Sinn.

3.2 Komplexe Typen (composite)

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

3.2.1 Arrays

Arrays sind eine Sammlung von durchnummerierten Elemente des gleichen Typs

Einen 4 Bit breiten Datenbus kann man auf verschiedene Arten deklarieren:

Eigene Definition auf der Basis «BIT»:

```
TYPE t_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.

Verwendung der Definition aus VHDL:

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

Verwendung des Package std_logic_1164

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

Bei der Angabe ganzer Bereiche ist die Laufrichtung des Index wichtig. Bereiche können aufwärts mit TO und abwärts mit DOWNTO angegeben werden.
Die Reihenfolge mit «DOWNTO» entspricht dabei unserer gewohnten Wertigkeit, mit dem MSB oder höchstem Bit links und LSB rechts.

```
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.

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

- String-Zuweisung in Anführungszeichen

```
c <= "0011";
```

- Concatenation (Aneinanderhängen)

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

- Aggregat-Zuweisung

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

- Kombination der obigen Varianten

```
a <= ('1', i, '0', j OR k);
```

- Explizite Zuweisung der Positionen mit dem « => » Operator

```
a <= (0 => '1',         -- Bit 0
      3 => j OR k,      -- Bit 3
      OTHERS => '0');   -- Rest
```

« OTHERS » wird häufig nur zur Initialisierung eines ganzen Vektors auf z.B. «0» verwendet. Dabei ist es in Wirklichkeit Teil einer sehr viel mächtigeren Zuweisungsart!

```
a <= (OTHERS => '0');
```

Man kann auch nur Teile eines Arrays zuweisen, und gibt dabei explizit die Bit-Positionen vor.

Im Beispiel rechts ist a nur 3-Bit breit, und wird an bestimmte Positionen von b kopiert.

```
SIGNAL a : STD_LOGIC_VECTOR
           (3 DOWNTO 0);
SIGNAL b : STD_LOGIC_VECTOR
           (8 DOWNTO 0);

b(6 DOWNTO 3) <= a;
```

Auf die gleiche Art, durch explizite Zuweisung der Positionen kann man z.B. die Verschiebung in einem Schieberegister beschreiben.
Rechts wird ein 8-Bit Schieberegister nach links geschoben und ein `'0'` von rechts eingefügt.

```
SIGNAL shift : STD_LOGIC_VECTOR
               (7 DOWNTO 0);

shift(7 DOWNTO 1) <=
             shift(6 DOWNTO 0);
shift(0)          <= '0';
```

3.2.2 Records

Ein **RECORD** besteht aus mehreren Elementen verschiedener Typen. Die einzelnen Felder eines Records werden durch den Element-namen 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);
  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 carry    : STD_LOGIC;

alu_1_in.data_a    <= x"F9";
alu_1_in.data_b    <= x"1C";
alu_1_in.operation <= SUB;
```

Natürlich können auch alle Objekte von einem Record zu einem anderen Record als Ganzes direkt zugewiesen werden.

```
alu_2_in           <= alu_1_in;
```

OST Ostschweizer Fachhochschule

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_7segment_display IS RECORD
  sement_1 : STD_LOGIC_VECTOR (0 TO 6);
  sement_0 : STD_LOGIC_VECTOR (0 TO 6);
END RECORD;

ENTITY hex_to_7segment_driver IS
  PORT (
    num     : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    display  : OUT t_7segment_display
  );
END ENTITY hex_4_digit_driver;
```

3.3 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 …
```

3.4 Typenumwandlungen

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);
```

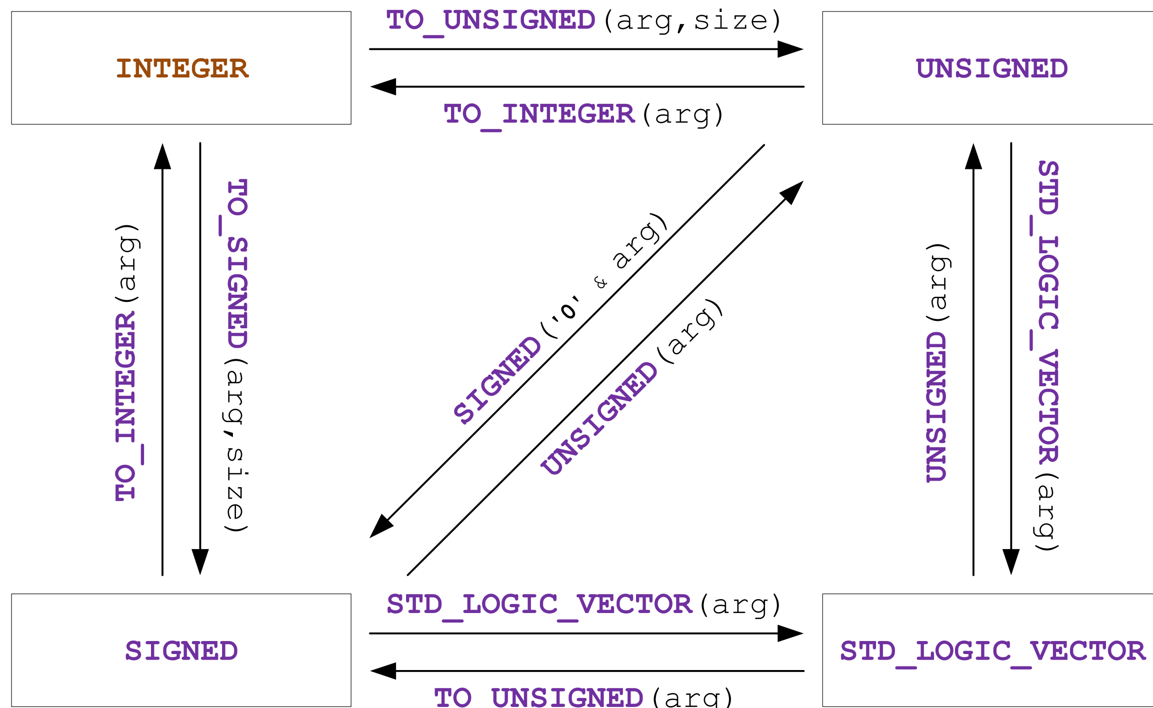
Andernfalls ist eine separate Konvertierungs-funktion notwendig, z.B. zur Umwandlung eines **STD_LOGIC_VECTOR** in einen **BIT_VECTOR**. Solche Konvertierungsfunktionen sind z.B. im Package **STD_LOGIC_1164** definiert.

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

some_bits <= TO_BITVECTOR(vector);
```

3.5 Typenumwandlungen mit Vektoren

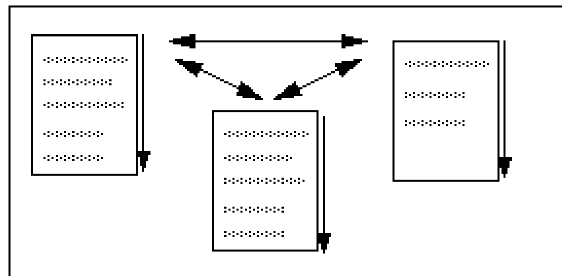
Die nachfolgende Tabelle zeigt alle Typumwandlungen im Package `IEEE.NUMERIC_STD`.

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*)



4 Konzepte

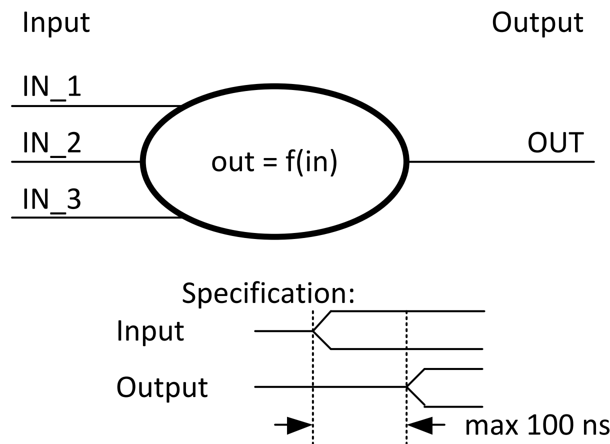
4.1 Grundlagen

Sequentiell	VHDL besitzt zwei Arten von Anweisungen. 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	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	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 «behavioral» oder «rtl» (siehe weiter unten).
Hierarchie	Hierarchie erlaubt es, ein System aus mehreren Modulen aufzubauen, wobei jedes dieser Module wiederum aus mehreren Modulen bestehen kann. Dadurch kann ein Entwurf schrittweise Top-Down verfeinert werden.

4.2 Abstraktionsebenen

Verhaltens-beschreibung (behaviour)

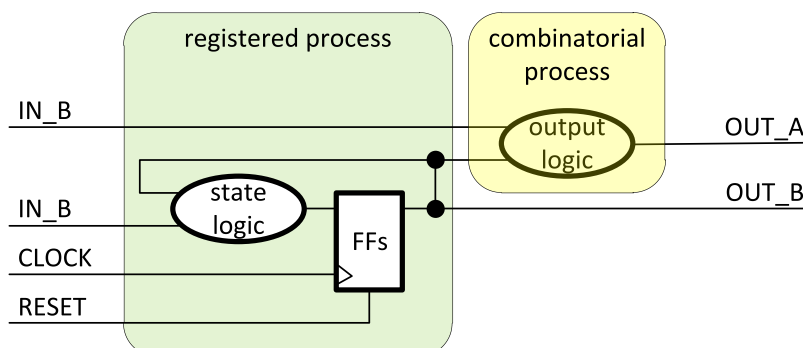
Die Verhaltensebene gibt die funktionale Beschreibung des Modells wieder. Es gibt keinen Systemtakt, die Signalwechsel sind asynchron und mit Schaltzeiten beschrieben. Man beschreibt Bussysteme oder komplexe Algorithmen, ohne auf die Synthetisierbarkeit einzugehen.



Das Bild zeigt eine einfache Vorgabe für die Funktion eines Moduls. Es soll aus den drei Eingängen IN_1, IN_2, IN_3 ein Ausgangswert (OUT) berechnet werden. Des Weiteren ist vorgegeben, dass der Ausgangswert spätestens 100 ns nach einer Änderung der Eingänge stabil sein 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.

Datenfluss-beschreibung auch RTL (Register-Transfer-Level) genannt

Auf der RTL Ebene (Register-Transfer-Level) versucht man das System in Blöcken mit kombinatorischer Logik und mit Flanken-getriggerten Registern (Flip-Flops) zu beschreiben. Letztendlich besteht das Design in der RTL-Ebene nur aus zwei unterschiedlichen, nebenläufigen Prozessen (siehe Kap. 8.1): Zum einen der rein kombinatorische Prozess und zum anderen der rein getaktete Prozess. Der getaktete Prozess erzeugt Flip-Flops und beides zusammen beschreibt immer einen Zustandsautomaten.



Bei der Modellierung werden also zusätzlich zu den Dateneingängen und Datenausgängen auch noch Steuersignale berücksichtigt, wie ein

Taktsignal (CLOCK) und gegebenenfalls eine Rücksetzsignal (RESET). Diese Steuersignale sind für die Register notwendig.

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

Bei der RTL Beschreibung ist der Takt das entscheidende Merkmal. Alle Operationen werden auf ein Taktsignal bezogen. Die RTL Beschreibung und Simulation gibt Aufschluss über das logische Verhalten und Abläufe, jedoch **nicht** zum realen Zeitverhalten, d.h. ob auch schnell genug funktioniert. Zeitverzögerungen können bei RTL nicht abgeschätzt werden, da die logischen Zusammenhänge noch in abstrakten Gleichungen und nicht auf Logikelemente synthetisiert ist.

Logikebene

Auf Logikebene wird das gesamte Design durch Logikgatter und speichernde Elemente (Flip-Flops, RAMs) 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 RTL zur Logikebene übernehmen die Synthesewerkzeuge.

Layout

Die Ebene, welche der Wirklichkeit am nächsten kommt, ist beim ASIC das physikalische Layout der Standard-Zellen, oder 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 auch die Leitungslängen und die Leitungsverzögerungen bekannt.

Das Design kann auf dieser Ebene auch wieder simuliert werden, wobei jetzt nicht mehr die Funktionalität, sondern das Zeitverhalten der Schaltung überprüft werden. Weil mit steigender Information auch der Aufwand für diese Art von Simulation sehr 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.

4.3 Beispiel 3-Bit Komparator

Zur Veranschaulichung folgen zwei Beispiele in je zwei Versionen. Beide Versionen erzeugen jeweils das genau gleiche Verhalten auch wenn sie in einem FPGA möglicherweise unterschiedlich implementiert werden.
Das erste Beispiel beschreibt einen 3-Bit Komparator. Bei gleichen Eingangssignalen a[2:0] und b[2:0] geht der Ausgang auf logisch 1.

Gatter-Level Beschreibung

```
ENTITY three_bit_compare IS
  PORT (
     a, b : IN  STD_LOGIC_VECTOR
                 (0 TO 2);
     eq   : OUT STD_LOGIC

  );
END ENTITY three_bit_compare;


ARCHITECTURE logic_gates
          OF three_bit_compare 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 logic_gates;
```

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 **AND** verknüpft werden

Register-Transfer Level Beschreibung (RTL)

```
ENTITY three_bit_compare IS
  PORT (
     a, b : IN  STD_LOGIC_VECTOR
                 (0 TO 2);
     eq   : OUT STD_LOGIC

  );
END ENTITY three_bit_compare;


ARCHITECTURE rtl
          OF three_bit_compare IS

BEGIN

    compare: PROCESS (a, b)
    BEGIN
       IF a = b THEN
           eq <= '1';
       ELSE
           eq <= '0';
       END IF;
    END PROCESS compare;

END ARCHITECTURE rtl;
```

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.

Das synthetisierte Resultat ist für beide Architekturen praktisch gleich, und damit auch gleich schnell!

OST
Ostschweizer
Fachhochschule

4.4 Beispiel 3-Bit Zähler

Das nächste Beispiel beschreibt einen 3-Bit Zähler.

Beschreibung als endlicher Automat (FSM)

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

    SIGNAL state      : t_state := s0;
    SIGNAL 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) THEN
        IF enable ='1' THEN
          state <= next_state AFTER 15 ns;
        END IF;
      END IF;
    END PROCESS synch;

END ARCHITECTURE fsm;
```

Wichtig ist die Initialisierung von „**state**“ auf den Wert ersten Wert. Wenn das fehlt, ist das Signal in der Simulation «U» für «Unknown», und wird sich auch nie fangen.

Da wir kein RESET Signal haben, initialisieren wir dieses Signal gleich bei der Deklaration.

Verhaltensbeschreibung (behavior)

```
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 rtl OF three_bit_counter IS

   SIGNAL curr_count : UNSIGNED(2 DOWNTO 0)
                     := (OTHERS => '1');
   SIGNAL next_count : UNSIGNED(2 DOWNTO 0);

BEGIN
    comb_proc : PROCESS (curr_count,
                         enable)
    BEGIN
        next_count <= curr_count;
        IF enable ='1' THEN
           next_count <= curr_count + 1
                         AFTER 3 ns;
        END IF;
    END PROCESS comb_proc;

    -- Register Function
    reg_proc : PROCESS (clk)
    BEGIN
      IF rising_edge(clk) THEN
        curr_count <= next_count
          AFTER 1 ns;
      END IF;
    END PROCESS reg_proc;

    -- Output Function
    count <= curr_count;

END ARCHITECTURE rtl;
```

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 „**curr_count**“ ein internes Signal erzeugt, welches dann asynchron (das heisst ständig und direkt) auf das Ausgangssignal gespiegelt wird.

Wichtig ist auch die Initialisierung von „**curr_count**“ auf den Wert Null, indem mit «OTHERS» alle Bits auf '0' gesetzt werden. Die Initialisierung kann Teil der Deklaration sein (wie hier), oder aber auf ein RESET Signal hin erfolgen.

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.

---

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

```
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.

Für eine Signalzuweisung, z.B. « `c <= '0'` » und für den relationalen Vergleich «kleiner gleich» wird das gleiche Zeichen verwendet. Der Compiler erkennt aus dem Zusammenhang 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 als) nur für Objekte der Typen `INTEGER`, `REAL`, `UNSIGNED` und `SIGNED` verwenden.

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

OTHERS

Ein spezieller Operator ist die Zuweisung eines Signals mit «OTHERS». Dabei werden alle Bits einzeln auf einen bestimmten Wert gelegt, unabhängig von der Zahl der Bits im Ziel-Signal. Die Syntax mit «OTHERS» ist im Grunde nur ein Teil einer viel mächtigeren Zuweisungs-Funktion, bei welcher für jedes Bit der Wert einzeln angegeben werden kann …
… und «OTHERS» bezieht sich dann auf den Rest der nicht einzeln spezifizierten Bits. Dies funktioniert nicht nur bei der Signal-Deklaration, sondern überall im Code!

```
SIGNAL my_sig : STD_LOGIC_VECTOR
                (7 DOWNTO 0)
                :=(OTHERS => '0');
```

Alle Bits werden `'0'`.

```
SIGNAL my_sig : STD_LOGIC_VECTOR
                (7 DOWNTO 0)
                := (5 => '1',
                    OTHERS => '0');
```

Nur Bit 5 wird zu `'1'`, alle anderen Bits werden zu `'0'`.

5.1 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 aus einzelnen Bits vom Typ **STD_LOGIC** definiert.

Beim Schreiben von Testbench-Funktionen welche nicht synthetisiert werden darf man ruhig auch komplexere mathematische Operationen verwenden, wie z.B. Sinus, Cosinus, Log oder Wurzeln. Dafür verwendet man mit Vorteil das Packet **IEEE**.**MATH_REAL** beziehungsweise **IEEE**.**MATH_COMPLEX**.

```
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!

Es gibt noch viele alte Beispiele und Implementation, welche die unsauber definierten **IEEE** Bibliotheken **STD_LOGIC_ARITH** sowie **STD_LOGIC_SIGNED** und **STD_LOGIC_UNSIGNED** verwenden.

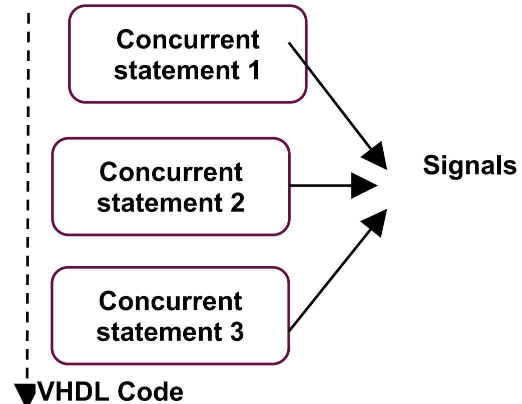
Die darin definierten Funktionen können mit anderen Definitionen kollidieren und deshalb sollten diese Bibliotheken nicht mehr verwendet werden.

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

6 Parallellaufende Anweisungen (Concurrent)

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.

Was das genau bedeutet wird in den nächsten Abschnitten ausführlich erklärt.



6.1 Signalzuweisung

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

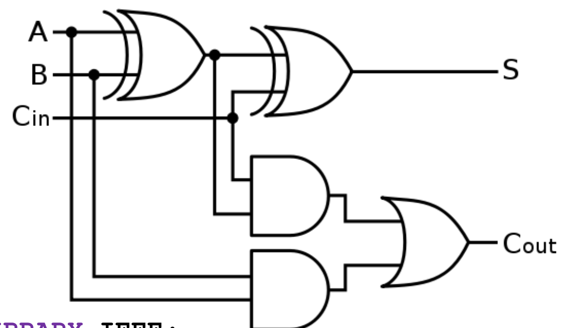
target <= value;

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 « `temp_sig` » dient als Verbindung vom
ersten XOR-Gatter mit einem AND- und einem zweiten XOR-Gatter.




```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

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

ARCHITECTURE rtl OF full_adder IS
   SIGNAL temp_sig : STD_LOGIC;
BEGIN
   temp_sig <= a XOR b;
   sum      <= temp_sig XOR c_in;
   c_out    <= (a AND b)
               OR (temp_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 <name> : <TYPE>
                [: = init_value];
```

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 der Wert `FALSE`

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

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

```
TYPE t_state IS (WAITING,
                 RUNNING,
                 STOPPED);

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.

Im Gegensatz zu einem FPGA (wo in der Regel alle Register automatisch zu `'0'` 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!

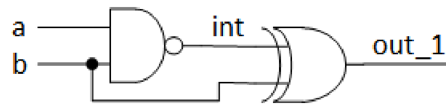
Signale können nur an zwei Stellen deklariert werden:

1. In der Port-Deklaration einer Entity

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

… oder …

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.

```
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 Testbenches praktisch, um zeitliche Abläufe zu simulieren.

```
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.

Wichtig: Anweisungen wie «**AFTER**» und

«**WAIT FOR**» gelten **nur** für die Simulation und lassen sich **NICHT** synthetisieren!

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

Typische Befehlsfolge in einer Testbench für Stimulus Signale:

```
a <= '0'; b <= '0';
WAIT FOR 10 ns;

a <= '0'; b <= '1';
WAIT FOR 10 ns;

a <= '1'; b <= '0';
WAIT FOR 10 ns;

a <= '1'; b <= '1';
WAIT FOR 10 ns;
```

Diese Sequenz generiert jeweils 10 ns lang nacheinander die vier möglichen Kombinationen für zwei Signale `a` und `b`.

7.3 Attribute von Signalen

Neben den Attributen für Datentypen (siehe Kapitel 3.3) 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> '<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 = '1' 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 = '1' AND clk'EVENT` und **rising_edge** identisch.

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

Bei der «**rising_edge**» wird nicht nur geprüft, ob der Takt einen «**EVENT**» hat und auf **'1'** steht, sondern auch, ob das Signal vorher auf **'0'** stand. Entsprechend wird eine Signaländerung von **'H'** auf **'1'** **NICHT** als steigende Flanke interpretiert!

Dieser kleine Unterschied kann später helfen, unliebsame Differenzen zwischen Simulation und echter Hardware zu vermeiden …

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! Dies kann dann Fehler mit ganz seltsamen Auswirkungen zur Folge haben.

```
full_adder : 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 full_adder;
```

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 als „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_drive
IS

  CONSTANT VGA_IMG_WIDTH
       :  UNSIGNED(9 DOWNTO 0)
       := TO_UNSIGNED(1024, 10);

  CONSTANT VGA_IMG_HEIGHT
       :  UNSIGNED(9 DOWNTO 0)
       := TO_UNSIGNED(768, 10);

BEGIN …
```

Eine weitere Möglichkeit ist, die Deklaration von Konstanten in ein **`PACKAGE`** auszulagern.

8 Sequentielle Anweisungen

Will man VHDL in einem Top-down Designablauf einsetzen, so müssen Modelle ohne konkrete Informationen über die zu verwendende Hardware auf einer hohen Abstraktionsebene dargestellt werden können. Durch die Verwendung der RTL (Register Transfer Level) Abstraktionsebene können logische Zusammenhänge und Funktionen definiert werden, welche erst später flexibel auf einem ASIC Prozess oder FPGA irgendeines Herstellers synthetisiert werden.

8.1 Der Prozess

Bisher haben wir im VHDL-Architekturbereich zwischen **BEGIN** und **END** nur «concurrent» ausgeführte Anweisungen behandelt.

Zusätzlich gibt es in der VHDL-Syntax besondere Bereiche mit sequentiellem interpretiertem Ablauf.

Viele VHDL Befehle können nur innerhalb von einem Prozess verwendet werden, wie die Entscheidungen **IF** und **CASE**, oder Schleifen-strukturen mit **FOR** oder **WHILE**.

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, im Bereiches zwischen den Schlüsslewörtern **BEGIN** und **END PROCESS** läuft der Code jedoch grundsätzlich nur sequentiell von oben nach unten ab.

In diesem Bereich können abstrakte Algo-rithmen beschrieben werden, wie sie aus anderen Hochsprachen bekannt sind.

Zu beachten ist jedoch, dass Zuweisungen an Signale zwar sequentiell bearbeitet (sprich «zur Ausführung markiert») werden … aber erst am Ende des Prozesses tatsächlich ausgeführt werden!

Einzige Ausnahmen sind dazu

- Zuweisungen an Variablen – diese werden sofort ausgeführt, die Zwischenresultate stehen sofort zur weiteren Verarbeitung bereit
- Signalzuweisungen in einem «**WAIT**» gesteuerten Prozess, wie es in einer Testbench typisch ist.

```
ARCHITECTURE first OF my_nor IS
BEGIN

    comb_proc : PROCESS (a,b)
       VARIABLE temp :
STD_LOGIC;
    BEGIN

       IF (a = '1')
       OR (b = '1') THEN
          temp := '1';
       ELSE
          temp := '0';
       END IF;
       c <= NOT temp;

    END PROCESS comb_proc;

END ARCHITECTURE first;
```

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

Die Parameter nach dem VHDL Schlüsselwort «**PROCESS**» bezeichnet man als «**Sensitivity List**». Ganz am Anfang jeder Simulation läuft jeder Prozess einmal durch. Von da weg startet ein Prozess nur, wenn eines der Signale in der «Sensitivity List» verändert wurde. Auf diesen «Trigger» hin wird dann wieder berechnet, ob es etwas Neues am Ausgang gibt.

Dies gilt natürlich nur für die Simulation – und spart dort viel Rechenzeit. **Dort ist sie aber absolut notwendig** … und Schaltungen können sehr «seltsam» reagieren, wenn Signale in der Liste fehlen.

Bei einer «echten» Implementation auf FPGA oder ASIC braucht es natürlich keinen «Trigger» - die physikalischen Gatter laufen alle parallel und reagieren sofort auf sich ändernde Eingänge.

8.1.1 Aktivierung für den logischen Ablauf

Ein Prozess muss (in der Simulation) durch eine Änderung eines oder mehrerer seiner Eingangssignale angestossen oder aktiviert werden. Dies passiert entweder

- durch die Angabe von Signalen in seiner Sensitivitätsliste, oder
- durch einen **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 ist dasselbe wie eine «**WAIT ON**» Anweisung am Ende vom Prozess.

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 mit «**WAIT**» ist **NICHT** synthetisierbar!! !

Obwohl die Funktion gleichbleibt, ist die erste Variante übersichtlicher und synthetisierbar … … und daher zu empfehlen.

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

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

- Unterbrechen des Prozesses bis ein Signalwechsel passiert.
  ```
  WAIT ON a, b, c;
  ```
- Unterbrechen des Prozesses bis eine Bedingung erfüllt ist.
  ```
  WAIT UNTIL x > 10;
  ```
- Unterbrechen des Prozesses bis eine Zeitspanne verstrichen ist
  ```
  WAIT FOR 10 ns;
  ```
- Unendlich langes Warten. Da ein Prozess immer aktiv ist, bietet diese Anweisung die einzige Möglichkeit ihn anzuhalten (z.B. nach der Initialisierungen)
  ```
  WAIT;
  ```

8.1.2 Ausgangswert bei einem Prozess

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 nicht **hier** überschrieben, sondern erst am Ende des Prozesses zugewiesen.

OST
Ostschweizer
Fachhochschule

Dieser Code macht effektiv das gleiche wie der Code vom Prozess «interesting» der vorherigen Seite
…
Wie im ersten Beispiel ist jeder mögliche Pfad immer abgedeckt, jedoch explizit durch Zuweisungen in jedem möglichen Pfad!

```
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 Entwickler beim Entwerfen von Modellen weiterhin in Blöcken und Modulen 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 producer_consumer 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;
```

8.1.3 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.

```
latch_1 : PROCESS (enable);
BEGIN
    IF enable = '1' THEN
        q <= d;
    END IF;
END PROCESS latch_1;
```

Der Prozess wird nur bei einer Änderung von «`enable`» ausgeführt! Wenn sich «`d`» ändert, passiert nix!!

Von den drei Beispielen rechts (latch_1 bis latch_3) funktioniert nur das dritte Modell richtig.

Bei den ersten beiden wird der Ausgang sich nicht ändern, wenn «`enable`» bereits hoch ist, und der Eingang «d» sich ändert, weil der Prozess durch das Eingangs-Signal «d» nicht aktiviert wird!

```
latch_2 : PROCESS;
BEGIN
   IF enable = '1' THEN
      q <= d;
   END IF;
   WAIT ON enable
END PROCESS latch_2;
```

Der Prozess wartet hier, bis sich «enable» ändert. Erst dann wird der Code erneut ausgeführt.

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

8.1.4 Ausführung von Signalzuweisungen

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

Hier werden immer noch die alte Werte von „a“ und „b“ verwendet!!!

Daraus folgt, dass z.B. Signale in einem Prozess nicht wie Variablen (siehe Kapitel 7.4) als Zwischenspeicher für Werte benutzt werden können. Muss mit dem Signalwert gerechnet werden, muss 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 (in_vector);
   VARIABLE x: STD_LOGIC;
BEGIN

   x := '1';
   FOR i IN 7 DOWNTO 0 LOOP
      x := in_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;
```

Das ist erlaubt! Werte von x und y werden alle 100 ns vertauscht.

8.2 Das IF-Statement

Ist die Bedingung nach dem **IF** nicht erfüllt, werden, sofern vorhanden, die Bedingung von **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). Es darf eine beliebige Anzahl von **ELSIF**-Statements vorkommen.

ELSE steht nur einmal am Ende des **IF** Statements, oder kann ganz weggelassen werden wenn es bei der nicht-erfüllung der Bedingung nichts zu tun gibt.

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.

Wie das Beispiel rechts zeigt, eignet sich **CASE** ausgezeichnet für die Programmierung von endliche Automaten (FSM).

Bei jeder «**WHEN**» Bedingung kann dann in den folgenden Zeilen ausgeführt werden, was genau bei den Zuständen geschehen soll, wann der Zustand wieder ändert, etc.

```
ARCHITECTURE rtl OF fsm_unit IS

TYPE t_fsm_states IS
    (init, standby, calc, done);
SIGNAL fsm_state : t_fsm_states;

BEGIN

  case_demo: PROZESS (fsm_state)
  BEGIN
    CASE (fsm_state) IS
        WHEN init    => …

        WHEN standby => …

        WHEN calc    => …

        WHEN OTHERS  => …
    END CASE;
  END PROCESS case_demo;

END ARCHITECTURE rtl;
```

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 deshalb das «**WHEN OTHERS**» nicht gebraucht wird.

Da aber «**sel**» vom Typ **STD_LOGIC** ist, kann das Signal auch viele zusätzliche Zustände wie «**"UU"**» oder «**"X0"**» annehmen.
Um all diese Möglichkeiten pauschal abzudecken gibt es die «**WHEN OTHERS**» Anweisung am Ende von «**Case**»

```
ENTITY mux_4_to_1 IS
  PORT (
     a,b,c,d : IN  STD_LOGIC;
     sel     : IN
STD_LOGIC_VECTOR
                              (1 DOWNTO 0);
     m_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"    => m_out <= a;
      WHEN "01"    => m_out <= b;
      WHEN "10"    => m_out <= c;
      WHEN "11"    => m_out <= d;
      WHEN OTHERS => m_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 <variable> 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 Kompilation bekannt sein.

Ganz wichtig ist zu verstehen, dass die **FOR**-Schleife keine sequentiell ausgeführte Logik beschreibt oder impliziert.
Es ist lediglich eine Möglichkeit, parallel auszuführenden Code mit regelmässiger Struktur effizient zu beschreiben.

Zwei Beispiele für einen Prozess, der die Reihenfolge (8 Bits) eines Vektors umkehrt. Die beiden Kodierungen sind absolut gleichwertig, aber unterschiedlich elegant …

```
reverse_proc_1 : PROCESS (i_bus)
BEGIN
   o_bus (7)  <= i_bus (0);
   o_bus (6)  <= i_bus (1);
   o_bus (5)  <= i_bus (2);
   o_bus (4)  <= i_bus (3);
   o_bus (3)  <= i_bus (4);
   o_bus (2)  <= i_bus (5);
   o_bus (1)  <= i_bus (6);
   o_bus (0)  <= i_bus (7);
END PROCESS reverse_proc_1;


reverse_proc_2 : PROCESS (i_bus)
BEGIN
   FOR i IN 0 TO 7 LOOP
      i_bus(7 - i) <= i_bus(i);
   END LOOP;
END PROCESS reverse_proc_2;
```

«i» ist hier vom Typ **INTEGER** und nur innerhalb der **FOR**-Schleife bekannt.

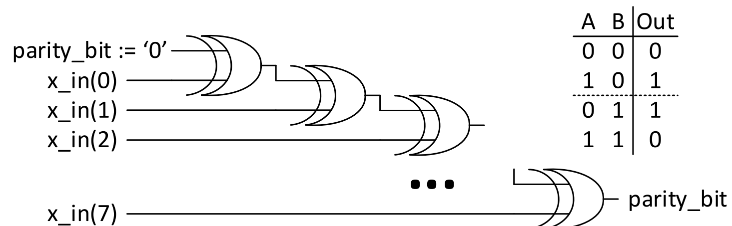
Ein gutes Beispiel für das Wirken einer **FOR**-Schleife stellt der Parity Generator in der Übung U4-2 dar.

```
ARCHITECTURE rtl OF parity IS
BEGIN

    comb_proc : PROCESS (x_in)
        VARIABLE parity_bit : STD_LOGIC;
    BEGIN

        parity_bit  := '0';
        FOR i IN 0 TO 7 LOOP
            IF x_in(i)  = '1' THEN
                parity_bit := NOT parity_bit;
            END IF;
        END LOOP;

        x_out   <= parity_bit & x_in;

    END PROCESS comb_proc;

END ARCHITECTURE rtl;
```

Der 8-Bit Std-Logic Vektor «x_in» dient als Eingang, und der 9-Bit Vektor «x_out» als Ausgang mit einem Parity Bit mehr.

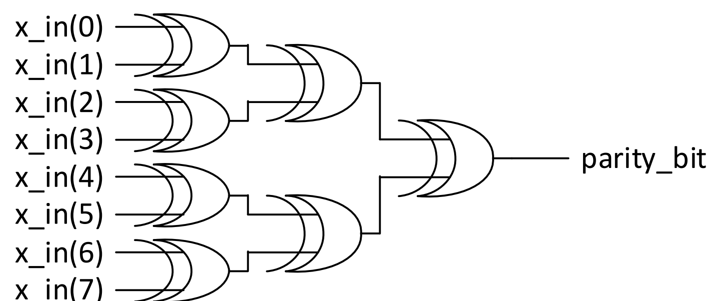
Die **FOR** –Schleife ist dabei lediglich eine effiziente Art der Beschreibung, und impliziert keine sequentielle Bearbeitung.

Bei der **IF** Befehl wird das «parity_bit» invertiert, wenn das betreffende Bit vom Bus «x_in» gesetzt ist. Dies wird in Hardware durch ein XOR Gatter erreicht:
Wie in der Logiktabelle rechts gezeigt, wirkt der XOR Eingang «B» als Steuersignal, ob das Signal «A» invertiert werden soll, oder nicht.



Die Implmentation der durch die **FOR** -Schleife deklarierten Logik resultiert in einer Kette von XOR Gattern, mit einem ersten Eingang des parity_bit = 0, und dann einzeln den Bits von «x_in».

In der Synthese wird daraus dann das erste XOR Gatter eliminiert, weil «0 XOR x_in(0)» = «x_in(0)» ergibt. Auch werden die XOR Gatter etwas anderes angeordnet um die Laufzeiten zu reduzieren …



… aber es wird auf jeden Fall eine parallele kombinatorische Logik erstellt – und keine über mehrere Takte verteilte Sequenz!

OST Ostschweizer Fachhochschule

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.

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

```
ARCHITECTURE sim OF my_cosine 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;
```

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

➔ Auch wenn ein Code nicht synthetisierbar ist, kann er immer noch bei einer Testbench sehr nützlich sein!

8.6 Schleifen mit LOOP ... EXIT

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

Das Ende einer **LOOP** Schleife kann statt durch Bedingungen beim Beginn der Schleife auch durch das **EXIT** –Statement begrenzt werden.
Im Beispiel ist **EXIT** mit einer Bedingung
verknüpft, der Befehl kann allerdings vor der Bedingung zusätzlich einen Label-Namen als Sprungziel ent-halten 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 **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 Java mit der Anweisung «continue»).

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

Für Anwendungen in der Simulation (z.B. Testbench) muss der **NEXT**-Abbruch 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_select** derart, dass in einem 16 Bit breiten Ausgangsfeld mit Namen **out_array** genau nur die Leitung gesetzt wird, die dem dezimalen Wert des Eingangs entspricht. Liegt am Eingang **in_select** 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_select** nur noch zwischen **BEGIN** und **END** bewegt, so dass keine erneute Initialisierung mehr erfolgt.

Die Konstruktion bei «`zero_loop`» ist nur zur Veranschaulichung mit einer Schleife gelöst worden. Ähnlich wie ein Prozess kann auch eine FOR-Schleife mit einem Label versehen werden (hier «zero_loop»).
Löschen von «sum» erreicht man auch mit

```
out_array <= (OTHERS => '0');
```

Beispielcode eines 4-zu-1 Decoders, welcher mit einem 4-Bit Eingang jeweils genau eines der 16 Ausgangssignale aktiv schaltet:

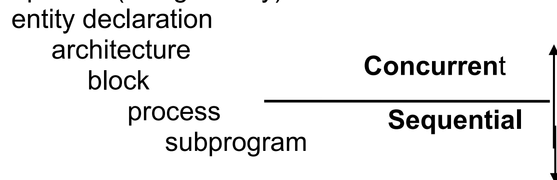
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY dec_4_1 IS
  PORT(
    in_select : IN  STD_LOGIC_VECTOR
                    ( 3 DOWNTO 0);
    out_array : OUT STD_LOGIC_VECTOR
                    (15 DOWNTO 0)
  );
END ENTITY dec_4_1;


ARCHITECTURE behavior OF dec_4_1 IS
BEGIN

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

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

   z_loop : FOR i IN 0 TO 15 LOOP
      out_array(i) <= '0';
   END LOOP z_loop;
   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
component (design entity)



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, nicht die globale Konstante X!

```
ARCHITECTURE tst_1 OF example IS
   -- global definition of x
   CONSTANT x: INTEGER := 2;

BEGIN
   p1  : PROCESS (in)
   --   local re-definition of x
      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 tst_2 OF example IS
   -- global definition of x
   CONSTANT x: INTEGER := 2;

BEGIN
   p2  : PROCESS (in)
   --   local re-definition of x
      VARIABLE x : INTEGER := 3;
   BEGIN
      out_1 <= tst_2.x * in;
      out_2 <= p2.x * in;
   END PROCESS p2;
END ARCHITECTURE test_2;
```

Zugriff auf die globale Konstante, gesteuert durch das Präfix «tst_2»

Zugriff auf die lokale Variable, gesteuert durch das Präfix «p2»

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.

```
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;
```

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 können keine **WAIT** Befehle enthalten.

Für indexierten Typen (wie **STD_LOGIC_VECTOR**, **SIGNED** und **UNSIGNED**) darf bei der **RETURN** Definition **kein** Bereich stehen (keine Anzahl Bits) !

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 und so ausgelagert werden. Siehe dazu Kapitel 11.

Innerhalb einer Architektur wird eine Funktion wie eine Zuweisung aufgerufen:

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

OST Ostschweizer Fachhochschule

10.2 PROCEDURE Unterprogramm

10.2.1 PROCEDURE Beispiel

Benötigt man ein Unterprogramm mit mehreren Rückgabewerten, so muss man statt einer Funktion eine `PROCEDURE` verwenden.
Prozeduren sind vor allem bei der Testbench-Programmierung sehr hilfreich.
Wie man am Beispiel rechts sehen kann, bietet eine `PROCEDURE` viel mehr Flexibilität bei der Wahl der Ein- und Ausgangssignale, und kann auch `WAIT` Befehle verarbeiten.

« `AFTER`» ergänzt hier die Zuweisung:
- Zum Zeitpunkt der Ausführung wird clk_out auf `'0'` gesetzt.
- «`t_pulse`» Zeit später geht es auf `'1'`

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

10.2.2 Beispiele für PROCEDURE Aufruf

Die Beispiele für die Prozedur-Aufrufe sind in eine Beispiel-Architektur eingebettet, welche im deklarativen Teil die Definition der Prozedur enthält

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

Die erste Variante ist am kürzesten, aber auch sehr anfällig für Fehler, da man die Reihenfolge der Signale genau richtig treffen muss.

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

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.

```
ARCHITECTURE sim OF proc_demo IS

   SIGNAL clock_1  : STD_LOGIC;
   SIGNAL clock_2  : STD_LOGIC;
   SIGNAL clock_3  : STD_LOGIC;

   PROCEDURE generate_clock (
   …
   END PROCEDURE generate_clock;
BEGIN

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


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

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

11 Bibliotheken und Packages

11.1 PACKAGE

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 Definitionen werden in «*Packages*» zusammengefasst, die aus einer «*Declaration*» und einem «*Body*» bestehen. Ohne Funktionen und Procedures wäre auch kein «*Package Body*» erforderlich. Der Vorteil dieser Aufteilung kommt noch aus einer Ära der langsamen Computer: Bei einer Änderung an der Funktion muss nur der «*Body*» neu kompiliert werden.

In diesem trivialen Beispiel bildet «**mean3**» aus a, b und c den Mittelwert.

```
--  function header only
PACKAGE mean3_pkg IS
   FUNCTION mean3 (a, b, c: REAL)
                  RETURN REAL;
END PACKAGE mean3_pkg;

--  function definition
PACKAGE BODY mean3_pkg IS
   FUNCTION mean3 (a, b, c: REAL)
                  RETURN REAL IS
   BEGIN
      RETURN (a+b+c) / 3.0;
   END FUNCTION mean;
END PACKAGE mean3_pkg;
```

11.2 LIBRARY

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

Da es viele Bibliotheken gibt, und diese Speicherplatz belegen, lädt man nur die benötigten.

Besondere Bedeutung hat die Bibliothek **IEEE**, mit standard Typen und Definitionen.

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.

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.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

LIBRARY IEEE;
USE IEEE. STD_LOGIC_1164.STD_LOGIC;
```

Als Beispiel die Verwendung «**mean3**»:

```
LIBRARY work;
USE work.mean3_pkg.ALL;

ENTITY average IS
   PORT(
      in1, in2, in3 : IN  REAL;
      output        : OUT REAL
   );
END ENTITY average;

ARCHITECTURE example OF average IS
BEGIN
   output <= mean3(in1, in2, in3);
END example;
```

Hier wird die Einbindung der Prozedur «**mean3**» aus Kapitel 11.1 gezeigt!

12 Hierarchie durch strukturierte Modelle / Komponenten

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

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.

Als Beispiel sind rechts zwei Architekturen «struct_1» und «struct_2» gezeigt.
Bei beiden wird ein 4-Bit Addierer **`adder4`** aus vier einzelnen Ein-Bit Volladdierern aufgebaut.

Das Package, Entity und Architektur für **`full_add`** setzten wir als gegeben voraus:
- «a», «b» und «cin» sind 1-Bit Eingänge
- «sum» und «cout» sind 1-Bit Ausgänge.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY adder4 IS  -- 4 Bit Adder
  PORT (
     a    : IN  UNSIGNED(3 DOWNTO 0);
     b    : IN  UNSIGNED(3 DOWNTO 0);
     cin  : IN  STD_LOGIC;
     sum  : OUT UNSIGNED(3 DOWNTO 0);
     cout: OUT STD_LOGIC
   );
END adder4;
```

Beispiel mit «effizienter» aber unübersichtlicher Instanziierung und Signalzuweisung durch die Reihenfolge der Signale.

```
ARCHITECTURE struct_1 OF adder4 IS
  SIGNAL c : STD_LOGIC_VECTOR
              (2 DOWNTO 0);
BEGIN
  A0: full_add PORT MAP (
      a(0),b(0),cin,sum(0),c(0));
  A1: full_add PORT MAP (
      a(1),b(1),c(0),sum(1),c(1));
  A2: full_add PORT MAP (
      a(2),b(2),c(1),sum(2),c(2));
  A3: full_add PORT MAP (
      a(3),b(3),c(2),sum(3),cout);
END ARCHITECTURE struct_1;
```

Unübersichtliches Beispiel. Ähnlich wie schon beim Beispiel für den Procedure-Aufruf ist die Zuordnung der Signale nur durch die Reihenfolge gegeben, und daher schlecht intuitiv überprüfbar.

Beispiel mit expliziter Signalzuweisung:

```
ARCHITECTURE struct_2 OF adder4 IS
  SIGNAL c : STD_LOGIC_VECTOR
              (2 DOWNTO 0);
BEGIN
    A0: full_add PORT MAP (
          a    => a(0),
          b    => b(0),
          cin  => cin,
          sum  => sum(0),
          cout => c(0)
    );

    A1: full_add PORT MAP (
          a    => a(1),
          b    => b(1),
          cin  => c(0),
          sum  => sum(1),
          cout => c(1)
    );
    -- A2 und A3 fehlen hier ...
```

Gutes Beispiel mit expliziter Signalzuweisung.

Aber Aufgepasst: der Operator « **=>** » zeigt **nicht** die Richtung der Verbindung!!!

Wie das Beispiel auch zeigt …
… benötigt die explizite Signalzuweisung auch mehr Platz.
Zum Glück muss nicht mehr jeder Code ausgedruckt werden – und übersichtliche Überprüfbarkeit hat auch seine Vorteile!

OST Ostschweizer Fachhochschule

13 Parametrisierbare Modelle

Alle Beispiele bis anhin zeigten eine in der Portdefinition vollständig fixierte Schnittstelle.

VHDL stellt uns darüber hinaus eine Methode 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 beim Einbinden (Integration) nach Bedarf geändert und «überschrieben» werden können.

```
ENTITY generic_register IS
  GENERIC (
    width : POSITIVE := 16
  );
  PORT (
    clk : IN  STD_LOGIC;
    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 erst bei der Verwendung der Komponente benötigt wird und auch von Anwendung zu Anwendung ändern 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 16 aus der Konstante **bus_width** (Wert 32) überschrieben.

```
ARCHITECTURE struct OF example IS
  CONSTANT bus_width : INTEGER:= 32;
  SIGNAL in_bus : STD_LOGIC_VECTOR
            (bus_width-1 DOWNTO 0);
  SIGNAL out_bus : STD_LOGIC_VECTOR
            (bus_width-1 DOWNTO 0);

BEGIN
   -- Register instanciation
   reg0 : generic_register
   GENERIC MAP (width => bus_width)
   PORT MAP (
      clk => clk,
      d   => in_bus,
      q   => out_bus
   );

   --  Rest of the code ...
END ARCHITECTURE struct;
```