

# Simulation digitaler Schaltungen in **VHDL**

Blockpraktikum  
Stand: 22. Juni 2020



Robotics Research Institute  
Abteilung Informationstechnik

Prof. Dr.-Ing. Uwe Schwiegelshohn

Institut für Roboterforschung  
Abteilung Informationstechnik  
Universität Dortmund

Edwin Naroska  
Elmar Ofenbach  
Dirk Reinecke  
Jürgen Kemper  
Daniel Hauschildt  
Arne Moos

# Inhaltsverzeichnis

<b>1 Modellbildung einer Schaltung</b>	<b>7</b>
1.1 Beschreibung eines digitalen Systems . . . . .	7
1.2 Grundstruktur des VHDL–Modells . . . . .	9
1.2.1 Der Abschnitt <i>entity declaration</i> . . . . .	10
1.2.2 Der Abschnitt <i>architecture body</i> . . . . .	13
<b>2 Die Betriebsebenen</b>	<b>16</b>
2.1 Die parallele Betriebsebene . . . . .	16
2.1.1 Signalzuweisungen in der parallelen Ebene . . . . .	16
2.1.2 Bedingte Signalzuweisungen in der parallelen Ebene . . . . .	18
2.1.3 Ausgabe von Fehlernachrichten in der parallelen Ebene . . . . .	20
2.2 Die sequentielle Betriebsebene . . . . .	21
2.2.1 Prozesse . . . . .	21
2.2.2 Fehlernachrichten in der sequentiellen Ebene . . . . .	24
2.2.3 Bedingte Anweisungen in der sequentiellen Ebene . . . . .	24
2.2.4 Schleifen in der sequentiellen Ebene . . . . .	25
2.2.5 Prozeduraufufe in der sequentiellen Ebene . . . . .	27
<b>3 Simulationsablauf</b>	<b>28</b>
3.1 Das 2–Stufen–Modell . . . . .	28
3.2 Die parallele Abarbeitung . . . . .	29
3.3 Verarbeitung von Signalzuweisungen . . . . .	30
3.3.1 Signalzuweisungen ohne Verzögerung . . . . .	35
<b>4 Datentypen</b>	<b>39</b>
4.1 Scalar–Typen . . . . .	39
4.2 Composite Typen . . . . .	41
4.3 Access Typen . . . . .	45
4.4 File Typen . . . . .	46
4.5 Subtypen . . . . .	46
<b>5 Unterprogramme</b>	<b>48</b>
5.1 Funktionen . . . . .	48
5.2 Resolution Funktionen . . . . .	49
5.3 Prozeduren . . . . .	50
<b>6 Packages</b>	<b>52</b>
6.1 Spezielle Pakete . . . . .	53
6.1.1 <i>std-logic</i> -Package . . . . .	54
6.1.2 <i>numeric_std</i> & <i>numeric-bit</i> -Package . . . . .	56
<b>7 Attribute</b>	<b>60</b>
7.1 Value-Kind-Attribute . . . . .	60
7.1.1 Value Type . . . . .	60
7.1.2 Value Array . . . . .	61

7.1.3	Value Block . . . . .	61
7.2	Funktionsattribute . . . . .	61
7.2.1	Funktionstypen-Attribute . . . . .	61
7.2.2	Funktionsarray-Attribute . . . . .	62
7.2.3	Funktionssignal-Attribute . . . . .	63
<b>8</b>	<b>Generate Anweisung</b>	<b>65</b>
8.1	Iterative Generate Anweisung . . . . .	65
8.2	Konditionale Generate Anweisung . . . . .	68
<b>9</b>	<b>Synthese</b>	<b>70</b>
9.1	Modellierung kombinatorischer Logik in VHDL . . . . .	72
9.1.1	Multiplexer . . . . .	73
9.1.2	Enkoder . . . . .	75
9.1.3	Prioritäts Enkoder . . . . .	78
9.1.4	Dekoder . . . . .	80
9.1.5	Komparatoren . . . . .	83
9.2	Modellierung sequentieller Logik . . . . .	84
9.2.1	Modellierung pegelgesteuerter D-Flipflops . . . . .	85
9.2.2	Modellierung flankengetriggter D-Flipflops . . . . .	87
9.2.3	Zähler . . . . .	90
9.2.4	Automaten (FSM) . . . . .	93
<b>A</b>	<b>VHDL–Benutzung</b>	<b>100</b>
A.1	Reservierte Wörter und Operatoren . . . . .	100

## Einleitung

VHDL (**VHSIC Hardware Description Language**) ist eine umfassende Hardwarebeschreibungssprache, mit der digitale Systeme auf unterschiedlichen Abstraktionsebenen modelliert und simuliert werden können. Die Sprache wurde im Rahmen des VHSIC-Programms (**Very High Speed Integrated Circuit**) des *Department of Defense* Anfang der 80er Jahre entwickelt. Ziel des Entwurfes war ein vom jeweiligen Stand der Mikroelektronik unabhängiger Sprachstandard, der Elemente bekannter Hochsprachen mit den notwendigen Elementen zur effektiven Modellierung von parallelen Systemen auf verschiedenen Abstraktionsebenen kombiniert. VHDL wurde im Jahr 1987 vom IEEE standardisiert (IEEE 1076-1987).

Die Sprache VHDL ist maschinen- und anwenderlesbar, so dass sie gleichermaßen dazu geeignet ist, Designs zu beschreiben als auch mit Hilfe eines VHDL-Simulators zu simulieren. Zudem gewinnt VHDL als Schnittstellensprache zwischen Entwurfstools an Bedeutung. Ein wichtiges Merkmal der Sprache ist, dass mit ihrer Hilfe ein komplexes System vom “ersten abstrakten Gedanken” bis hinunter zur Gatterebene modelliert und simuliert werden kann. Dabei können verschiedene Abstraktionsebenen in einem Modell auch gemischt werden.

Die Möglichkeiten der Sprache sind so reichhaltig, dass man sie nicht mit wenigen Seiten eingehend darstellen kann. Aus diesem Grund erhebt dieses Skript keinen Anspruch auf Vollständigkeit, sondern dient lediglich dazu, die am meisten gebrauchten Elemente von VHDL zu beschreiben und sie anhand von Beispielen zu vertiefen.

Kapitel 1 zeigt den grundsätzlichen Aufbau eines VHDL-Modells. In Kapitel 2 werden die verschiedenen Betriebsebenen und die dazugehörigen Anweisungen von VHDL vorgestellt. Kapitel 3 beschreibt die Simulation von VHDL-Modellen. Anschließend werden in Kapitel 4 die standardmäßig in VHDL vorhandenen Datentypen, sowie die Definition eigener Typen behandelt. Die aus anderen Hochsprachen bekannten Sprachmittel wie Funktionen, Prozeduren und einige VHDL spezifische Besonderheiten sind in Kapitel 5 beschrieben. Im nächsten Kapitel werden Packages beschrieben, mit denen häufig wiederkehrende Deklarationen gebündelt werden können und außerdem wird auf die zwei von der IEEE standardisierten Packages *std\_logic* und *numeric\_std* eingegangen. Kapitel 7 befasst sich mit den verschiedenen Arten von Informationen über VHDL-Objekte (Attribute), die dem Anwender zur Verfügung stehen. Kapitel 8 bespricht Möglichkeiten zur programmgesteuerten Beeinflussung der Struktur eines Modells und Kapitel 9 geht auf die Synthese von VHDL-Code ein und zeigt die Modellierung einiger grundlegender, digitaler Komponenten anhand von Beispielen.

Anhang A gibt schließlich nochmals einen kurzen Überblick über die reservierten Schlüsselwörter und Operatoren in VHDL.

## VHDL-Syntaxregeln

Hier vorab ein paar Regeln für die VHDL-Syntax:

- VHDL unterscheidet **nicht** zwischen Klein- und Großschreibung
- Deutsche Umlaute sind **nicht** zulässig.
- Kommentaren werden zwei Minuszeichen vorangestellt.  
Beispiel: [statement] -- Kommentar
- Anweisungen werden mit einem ";" abgeschlossen.

## VHDL-Operatoren und Zuweisungen

VHDL kennt folgende relationale Operatoren:

= gleich  
/ = ungleich  
< kleiner als  
<= kleiner/gleich  
> größer  
>= größer/gleich

Bei der Zuweisung werden folgende Symbole verwendet:

:= Variablenzuweisung  
=< Signalzuweisung

Der Compiler erkennt aus dem Kontext, ob es sich um eine Signalzuweisung oder einen Vergleich kleiner/gleich handelt.

## Definitionen

Zum besseren Verständnis der einzelnen Beispiele und Definitionen ist in dieser Anleitung folgender Zusammenhang zwischen der Wahl des Zeichenformates und der Textbedeutung gewählt worden:

- Reservierte Wörter der VHDL-Syntax werden klein und fett gedruckt.

Beispiel: **begin** oder **end**

- Durch eine Library vordefinierte Bezeichnungen werden groß (und nicht fett) gedruckt. Die Bedeutung dieser Bezeichnungen kann durch eigene Definitionen geändert werden.

Beispiel: BIT oder INTEGER oder AND

- Frei wählbare Bezeichnungen sind normal in Kleinschrift gedruckt. Dies betrifft hauptsächlich Variablen- oder Signallamen und sonstige frei bestimmbare Namen.

- Ausdrücke, die noch weiter aufgelöst werden müssen, werden kursiv gedruckt.

Beispiel: *interface-list* oder *statements*

# 1 Modellbildung einer Schaltung

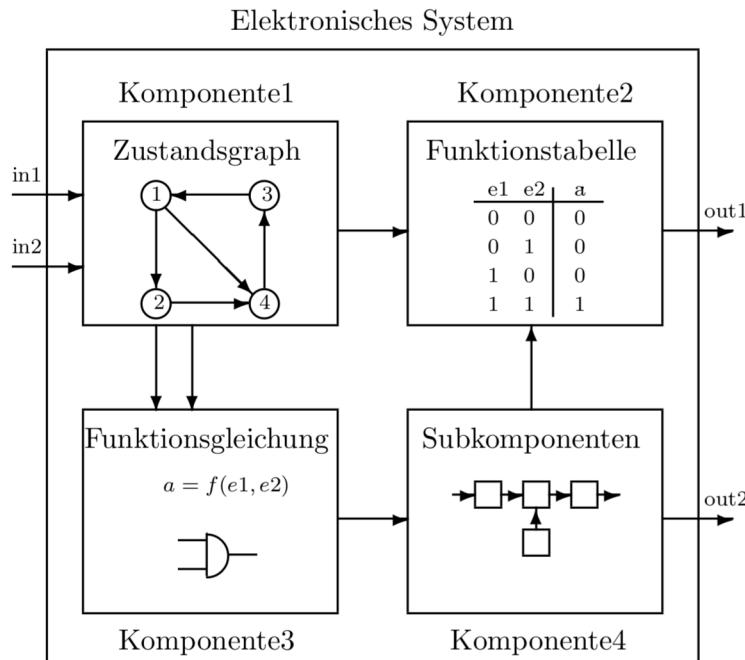
## 1.1 Beschreibung eines digitalen Systems

Ein elektronisches System oder eine elektronische Komponente (*design entity* oder *entity*) besteht aus einer Anzahl von Eingängen, einer Verarbeitungseinheit und einer Anzahl von Ausgängen. Der Zweck einer solchen Komponente besteht darin, in Abhängigkeit von der Zeit, den Eingangssignalen und der Vorgeschichte bestimmte Ausgangsmuster zu generieren.

Der Aufbau (*architecture*) eines solchen Systems oder einer solchen Komponente kann durch zwei unterschiedliche Methoden beschrieben werden:

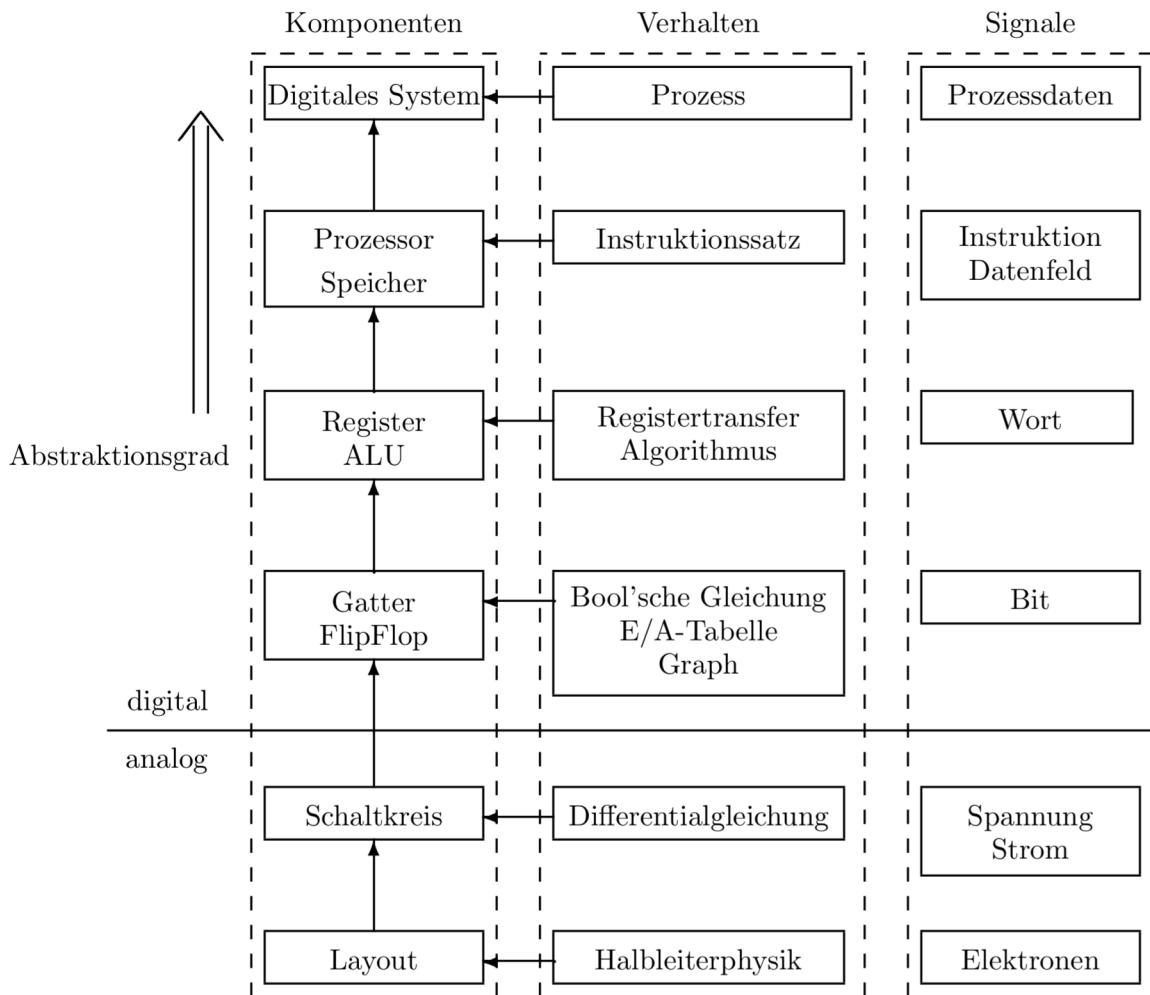
1. Die verhaltensorientierte Beschreibung (*behavior*) gibt anhand von Tabellen, Algorithmen, Gleichungen, Graphen, etc. das Ein-/Ausgabeverhalten (Übertragungsfunktion) an.
2. Bei der schaltungstechnischen Beschreibung wird das System durch die Verbindung einzelner Komponenten beschrieben (*structure*).

Die bei der Strukturbeschreibung verwendeten Komponenten müssen ihrerseits wiederum durch eine Übertragungsfunktion oder durch eine Strukturbeschreibung definiert sein. Zu jeder Komponente gehört eine genaue Spezifikation ihrer Schnittstellen (*entity declaration*). Diese Schnittstellenspezifikation beinhaltet die Anzahl, die Namen, die Wertebereiche und die Übertragungsrichtungen der Signalleitungen, die die Komponente nach außen verbindet.



Daraus ergibt sich, dass ein System auf zwei verschiedene Arten betrachtet werden kann; entweder als Black-Box, bei der nur das Übertragungsverhalten bekannt ist (berücksichtigt

wird) oder aber auf einer der unteren Ebenen, in denen eine Zerlegung in einzelne Komponenten erfolgt und die Funktion basierend auf Verhaltensbeschreibungen kodiert wird. Die sich hieraus ergebenden Abstraktionsebenen, die Verhaltensbeschreibungen und die Art der Signale auf den jeweiligen Ebenen sind im folgenden Bild dargestellt:



Bei einem Hardware-Testaufbau der Schaltung gibt man auf die Eingänge der Schaltung bestimmte Muster (Patterngenerator) und untersucht, wie sich die Ausgänge verhalten (Logikanalysator). Dazu muss die Schaltung komplett aufgebaut werden, was einen nicht geringen Aufwand darstellt. Außerdem sind solche Testaufbauten relativ schwer zu modifizieren. Mit der Verbreitung von universell programmierbaren Rechnern ging man immer mehr dazu über, Schaltungen durch Hardware-Beschreibungssprachen (**HDL**, **Hardware Description Language**) zu spezifizieren und dann mit Hilfe von Simulatoren zu testen. Ein minimales Simulationssystem für den beschriebenen Einsatzbereich muss folgende Eigenschaften besitzen:

- Schnittstelle zu einer HDL

- Festlegung der Eingangssignale (Stimuli)
- Visualisierung der Ausgangssignale

Weitere zusätzliche Module für ein Simulationssystem sind:

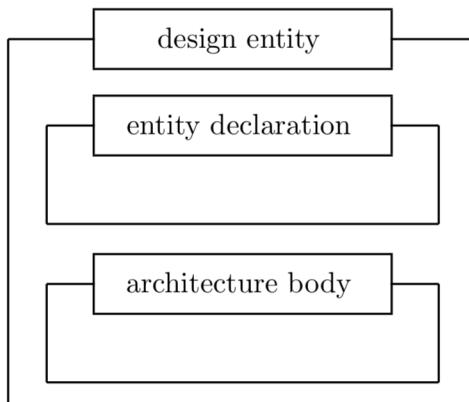
- Synthesetools (Generierung von Programmcode für programmierbare Bausteine, z.B. Gals, FPGAs, ASICs usw.)
- Debugger
- Graphische Eingabe des Schaltplans (schematic entry)
- Steuerprogramm für die Tools (framework)

Aus den oben angegebenen Betrachtungen über die Methodik der Schaltungsbeschreibung lassen sich folgende Anforderungen an eine Hardwarebeschreibungssprache stellen:

- Eine Schaltung muss sich aus Komponenten zusammensetzen lassen.
- Es muss möglich sein, jede Komponente durch ihr E/A-Verhalten oder durch die Zusammensetzung aus Subkomponenten zu beschreiben.
- Die für eine Abstraktionsebene notwendigen Datentypen müssen unterstützt werden.
- Es muss möglich sein, zeitlich parallele Vorgänge zu simulieren.

## 1.2 Grundstruktur des VHDL–Modells

Der Grundkörper eines VHDL–Modells (*design entity*) besteht aus der Spezifikation seiner E/A-Signale (*entity declaration*) und der Beschreibung seines Aufbaus (*architecture body*). Der Deklarationsteil und die Architekturbeschreibung sind zwei getrennte Abschnitte. Der Bezug der jeweiligen *entity declaration* zu dem zugehörigen *architecture body* wird über einen gemeinsamen, frei wählbaren Namen hergestellt.



### 1.2.1 Der Abschnitt *entity declaration*

Über die in dem Abschnitt *entity declaration* spezifizierten Signale, kann die Komponente mit der Außenwelt kommunizieren. In diesem Abschnitt sind die Schnittstellen, also die Ein- und Ausgänge der Komponente definiert. Die deklarierten Signale entsprechen den Anschlüssen eines elektronischen Bauteils z.B. eines IC's.

**Nomenklatur der Komponentenspezifikation:**

```
entity entity-identifier is
  [generic interface-list]
  [port interface-list]
  [declarations]
[begin
  statements]
end entity-identifier;
```

Der frei wählbare Komponentenname wird zwischen den Schlüsselwörtern **entity** und **is** angegeben. Dieser sollte die Funktion des Bausteins widerspiegeln, also z. B. *and*, *or* oder *multiplex* (Dieser Name wird auch später in der entsprechenden Architekturbeschreibung zwischen den Schlüsselwörtern **of** und **is** angegeben.)

Nach dem Schlüsselwort **port** werden die elektrischen Anschlüsse (Signale) des Bausteins spezifiziert. Dabei müssen die Signalnamen, die Wertetypen und die Übertragungsrichtungen angegeben werden.

Nach dem Schlüsselwort **generic** können Parameter für diese Komponente definiert werden, die ihr mit übergeben werden können. Diese Parameter sind keine elektrischen Signale! Sie dienen vielmehr dazu, dem Bauteil bestimmte Eigenschaften zu übergeben, z.B. eine Verzögerungszeit.

In dem Abschnitt *declarations* können Typen und Alias-Namen vereinbart werden. Der Block **[begin statements]** kann Funktions- und Prozessdefinitionen enthalten.

#### Beispiel 1: Verhaltensbeschreibung eines UND-Gatters

Es soll ein UND-Gatter mit zwei Eingängen ('e1' und 'e2') und einem Ausgang 'a' in VHDL modelliert werden. Der Aufbau des Gatters wird durch sein Verhalten in Form einer booleschen Gleichung definiert.

```
entity und_2 is
  port ( e1, e2: in BIT;
         a: out BIT);
end und_2;

architecture behavior of und_2 is
begin
  a <= e1 AND e2;
end behavior;
```

In diesem Beispiel wurde 'und\_2' als Name der Komponente gewählt. Dieser Name soll die Funktion der Komponente sowie die Anzahl der Eingänge beschreiben. Über den Namen wird die Beziehung zwischen dem Deklarationsteil und der Architekturdefinition hergestellt. Die Architekturdefinition bekommt noch einen eigenen Namen, hier 'behavior'. Als Typ der Ein- und Ausgangssignale ist der vordefinierte Typ 'BIT' gewählt worden, für die UND-Verknüpfung wurde die vordefinierte Library-Funktion 'AND' gewählt. Die Signalzuweisung auf das Signal 'a' erfolgt mit Hilfe der Zeichen '<='.

### Die PORT–Deklaration

Die Ein– bzw. Ausgänge der Komponente werden in der nach dem Schlüsselwort **port** angegebenen *interface-list* deklariert. Sie gehören zur Klasse der Signale. Sie dienen (unter anderem) dazu, Komponenten miteinander zu verbinden, sind also für den Datenaustausch zwischen Komponenten zuständig. Im Gegensatz zu Variablen können Signale mit einem zeitlichen Verhalten (delay) versehen werden.

#### Nomenklatur der PORT–Deklaration:

```
port (identifier-list : [in|out|inout|buffer] type [:=expression][;...]);
```

Die *identifier-list* ist eine durch Kommas getrennte Auflistung eines oder mehrerer Signallnamen, die zu dem selben Modus und Typ gehören. Der Modus legt die Übertragungsrichtung des Signals fest und der Typ die Art der zu übertragenden Daten. Mit '*expression*' kann dem Signal ein Anfangswert zugeordnet werden, also ein Wert den das Signal zu Beginn der Simulation hat.

Die in VHDL verfügbaren Modi für Signale sind:

- **in**

Der Datenfluss geht nur in die Komponente, aber nicht hinaus. Ein solcher Port kann von der Komponente nur gelesen, aber nicht beschrieben werden.

- **out**

Der Datenfluss geht nur aus der Komponente heraus. Dieser Port kann von der Komponente nur beschrieben, aber nicht gelesen werden.

- **inout**

Der Port kann beschrieben und gelesen werden, da der Datenfluss sowohl aus der Komponente hinein, als auch hinaus geht.

- **buffer**

Dieser Port ist beschreib- und lesbar, allerdings kann im Gegensatz zum **inout**-Port nur eine Signalquelle angeschlossen werden.

Als Datentypen können entweder selbst definierte Typen oder aber die Standard-Datentypen verwendet werden, die in der Standard-Library vordefiniert sind. Diese vordefinierten Typen sind die Folgenden:

- **BIT**

gültige Werte: '0' und '1'

- BOOLEAN  
gültige Werte: FALSE und TRUE
- CHARACTER  
gültige Werte: alle verfügbaren Zeichen
- SEVERITY\_LEVEL  
gültige Werte: NOTE, WARNING, ERROR, FAILURE (Fehlerbehandlung)
- INTEGER  
gültige Werte: ganze Zahlen
- REAL  
gültige Werte: Fließkommazahlen
- TIME  
gültige Werte: Zeitangaben mit Einheiten (fs, ps, ns, ... min, hr)
- STRING  
gültige Werte: aus CHARACTER zusammengesetzte Zeichenkette
- BIT\_VECTOR  
gültige Werte: aus BIT zusammengesetztes, eindimensionales Feld

### Die GENERIC-Deklaration

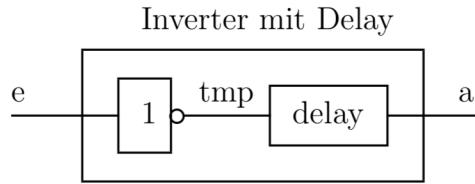
Mit Hilfe der Angabe **generic** kann eine Komponente parametrisiert werden. Über einen generischen Parameter kann der Komponente von außen ein Wert übergeben werden, der kein Signal darstellt. Damit wird es möglich, auf einfache Art und Weise Komponenten zu konfigurieren, also deren Verhalten zu beeinflussen. Im Gegensatz zu Signalen, können generische Parameter ihre Werte allerdings zur Laufzeit nicht verändern, d. h. sie behalten ihren zu Beginn der Simulation festgelegten Wert bis zum Ende der Simulation bei.

#### Nomenklatur der GENERIC-Deklaration:

```
generic (identifier-list : type [:=expression][;...]);
```

#### Beispiel 2: Inverter mit einstellbarer Verzögerung

Bei der Erstellung einer ALU werden z. B. mehrfach Inverter mit verschiedenen Verzögerungswerten benötigt. Um nun nicht für jede Verzögerungszeit eine eigene Komponente definieren zu müssen, wird nur eine Komponente mit einer über einen generischen Parameter einstellbaren Verzögerungszeit entworfen. Bei jeder Instantiierung der Komponente kann dem Parameter 'delay' ein neuer Wert zugewiesen werden. Fehlt diese Zuweisung, so gilt die Voreinstellung von 10 ns.



```

entity nicht is
  generic (delay : TIME := 10 ns);
  port ( e : in BIT := '0';
         a : out BIT);
end nicht;

architecture behavior of nicht is
signal tmp : BIT;
begin
  tmp <= NOT e;
  a <= tmp after delay;
end behavior;
  
```

### 1.2.2 Der Abschnitt *architecture body*

In dem Abschnitt *architecture body* ist die eigentliche Funktion einer Komponente angegeben. Eine Komponente kann mehrere dieser Architekturbeschreibungen besitzen, von denen jedoch nur eine aktiv sein darf. So ist z. B. ein Flipflop sowohl vom Verhalten her (normale sequentielle Funktionsbeschreibung wie sie auch bei anderen Hochsprachen verwendet wird) als auch in einer strukturellen Blockbildung (es werden mehrere Bausteine definiert und dann über Signale miteinander verbunden) beschreibbar.

#### Nomenklatur der Architekturbeschreibung:

```

architecture architecture-identifier of entity-identifier is
  [declarations]
begin
  [statements]
end architecture-identifier;
  
```

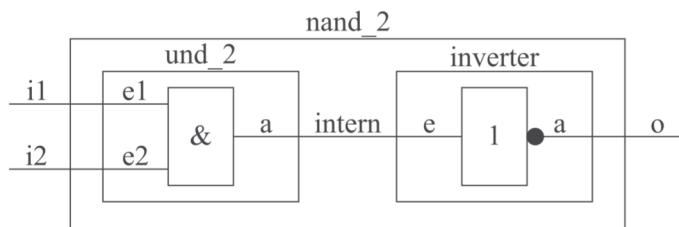
Der zwischen den Schlüsselwörtern **of** und **is** stehende Komponentenbezeichner ist identisch mit dem Bezeichner des Deklarationsteils dieser Komponente. Existieren zu einer Komponente mehrere Architekturbeschreibungen, so können diese anhand des unterschiedlichen Architekturnamens unterschieden werden. Die in der Architekturbeschreibung möglichen Deklarationen werden in den folgenden Kapiteln näher beschrieben. Diese Deklarationen sind von außen nicht sichtbar, sie stehen nur innerhalb der Architekturbeschreibung zur Verfügung. Zwischen den reservierten Wörtern **begin** und **end** liegt die eigentliche Funktionsbeschreibung der Komponente. Diese Funktionsbeschreibung besteht aus verschiedenen Anweisungen (*statements*), die, bis auf Anweisungen innerhalb einer Prozessdefinition, zeitlich parallel (*concurrent*) bearbeitet werden.

Folgende Anweisungstypen können in einer Architekturbeschreibung vorkommen:

- Concurrent Signal Assignment  
Mit dieser Anweisung wird eine Zuweisung auf ein Signal vorgenommen.
- Concurrent Assertion  
Hiermit werden Rückmeldungen in ein Programm eingebunden, die zur Laufzeit auf mögliche Fehlerzustände oder Ereignisse aufmerksam machen.
- Block  
Die Block-Anweisung erlaubt dem Anwender eine Architekturbeschreibung in mehrere logische Bereiche zu unterteilen. In einem Block können lokale Signale, Konstanten, etc. deklariert werden.
- Concurrent Procedure Call  
Aufruf einer Prozedur (Unterprogramm), die z.B. im Architekturdeklarationsteil definiert wurde.
- Component Instantiation  
Das Verhalten eines Bausteins kann teilweise oder ganz durch ein Zusammenschalten von anderen Bausteinen beschrieben werden. Diese werden dann als Komponente deklariert. In der Architekturbeschreibung kann hier die Verdrahtung der einzelnen Subkomponenten definiert werden.
- Process  
Der Prozess stellt die Verbindung zwischen den zwei Betriebsebenen *concurrent* und *sequential* her. Als Ganzes wird der Prozess parallel zu anderen Anweisungen ausgeführt, innerhalb des Prozesses werden die Anweisungen aber sequentiell bearbeitet. Mit der **PROCESS**-Anweisung wird der sequentielle Code in den parallelen Code eingebettet.

### Beispiel 3: Strukturbeschreibung eines NAND-Gatters

In diesem Beispiel soll ein NAND-Gatter aus dem im Beispiel 1 erstellten UND-Gatter und dem in Beispiel 2 erstellten Inverter zusammengesetzt werden. Die Verzögerungszeit des Gatters soll 20 ns betragen. Die Komponenten werden mit dem **component**-Statement in der Architekturbeschreibung angemeldet und ihre Ports werden dort nochmals deklariert. Danach kann dann durch Zusammenschalten des UND-Gatters und des Inverters das NAND-Gatter definiert werden.



```
use WORK.und_2;
use WORK.inverter;

entity nand_2 is
    port ( i1, i2: in BIT;
           o: inout BIT);
end nand_2;

architecture structure of nand_2 is
    signal intern : BIT;
    component und_2
        port (e1, e2: in BIT;
              a: out BIT);
    end component;
    component inverter
        generic (delay : TIME := 10 ns);
        port (e : in BIT;
              a : out BIT);
    end component;
begin
    U0: und_2
        port map (e1 => i1, e2 => i2, a => intern);
    U1: inverter
        generic map (20 ns) -- Achtung: Kein Semikolon!
        port map (e => intern, a => o);
end structure;
```

## 2 Die Betriebsebenen

In einer (digitalen) Schaltung operieren alle Komponenten und Subkomponenten gleichzeitig. Beispielsweise stellt ein in der Schaltung enthaltenes Gatter seine Funktionalität kontinuierlich zur Verfügung, parallel zu der Funktionalität anderer, in der Schaltung enthaltener Gatter.

Bei einer sequentiellen Programmiersprache (z.B. C++, Pascal, etc.) wird immer nur eine Anweisung zu einem Zeitpunkt ausgeführt. Deshalb eignet sich eine sequentielle Programmiersprache nicht für die Beschreibung von Schaltungen und Systemen mit zeitlich parallelen Vorgängen und Ereignissen. Das Programmiermodell von VHDL geht davon aus, dass alle Anweisungen in den Architekturbeschreibungen gleichzeitig ablaufen. Diese parallele Betriebsebene wird *concurrent level* genannt. Neben dieser parallelen Betriebsebene gibt es in VHDL aber auch die Möglichkeit auf der sequentiellen Ebene (*sequential level*) Algorithmen sequentiell zu beschreiben, ähnlich wie in anderen Programmiersprachen.

### 2.1 Die parallele Betriebsebene

Die parallele Betriebsebene (*concurrent level*) ist für die Verbindung von Komponenten und Prozessen verantwortlich. Alle Komponenten und Prozesse operieren gleichzeitig. Der Datenübertragung zwischen den Prozessen und Komponenten mit Hilfe von Signalen kommt dabei eine besondere Bedeutung zu.

#### 2.1.1 Signalzuweisungen in der parallelen Ebene

Signalzuweisungen stellen die grundlegenden Anweisungen in VHDL dar. Mit ihrer Hilfe kann einem Signal ein Wert zugewiesen werden. Signale übertragen Informationen zwischen Prozessen und sind für die Vernetzung des gesamten Designs zuständig. Eine Wertzuweisung auf ein Signal kann unmittelbar erfolgen oder aber mit einer zeitlichen Verzögerung belegt werden.

**Nomenklatur einer unverzögerten Signalzuweisung:**

```
signal <= value;
```

Mit dieser Anweisung wird der Wert *value* dem Signal mit dem Namen 'signal' unmittelbar zugewiesen. Da es sich hierbei um eine Anweisung auf der parallelen Betriebsebene handelt, findet diese Zuweisung ständig statt. Ein Simulator, der i. d. R. auf einem sequentiellen Rechner läuft, wird eine solche Zuweisung auf ein Signal natürlich nur zu bestimmten Zeiten (Simulationszyklen) vornehmen. Der Simulator bildet aber die Gleichzeitigkeit der Ereignisse nach, so dass sich für den Programmierer ein perfektes zeitgleiches Verhalten seiner Anweisungen auf der parallelen Betriebsebene ergibt. Für *value* kann jede Erklärung eingesetzt werden, die einen dafür vorgesehenen Wert liefert, beispielsweise:

- Funktionsaufrufe
- arithmetische Ausdrücke
- direkte Wertzuweisungen

Bei der Transport/After-Anweisung wird jede Wertänderung von *value* mit der angegebenen Zeitverzögerung dem Signal zugewiesen.

**Nomenklatur für die TRANSPORT/AFTER-Anweisung:**

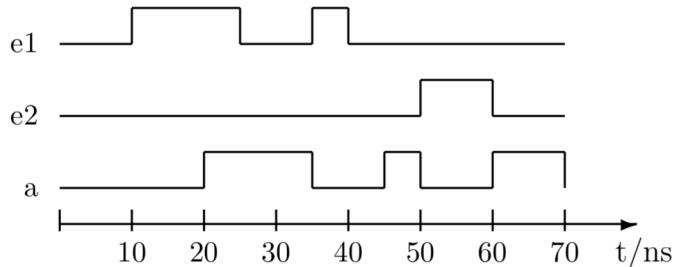
```
signal <= transport value after time;
```

**Beispiel 4:** ODER-Gatter mit TRANSPORT/AFTER-Verzögerung

```
entity oder_2 is
  port (e1, e2: in BIT;
        a: out BIT);
end oder_2;

architecture behavior of oder_2 is
begin
  a <= transport e1 OR e2 after 10 ns;
end behavior;
```

Das Zeitverhalten einer mit der TRANSPORT/AFTER-Anweisung verzögerten Signalzuweisung zeigt folgendes Diagramm:



Bei der AFTER-Anweisung ohne den Zusatz **transport** handelt es sich um eine Zeitverzögerung, die Wertänderungen des Signals, welche kürzer als die angegebene Verzögerungszeit sind, unterdrückt. Diese Anweisung eignet sich also sehr gut dazu, Spikes oder Peaks zu unterdrücken.

**Nomenklatur für die AFTER-Anweisung:**

```
signal <= value after time;
```

**Beispiel 5: ODER-Gatter mit AFTER–Verzögerung**

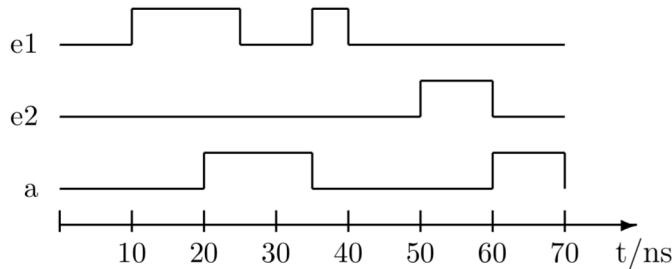
```

entity or_2 is
    port (e1, e2: in BIT;
          a: out BIT);
end or_2;

architecture behavior of or_2 is
begin
    a <= e1 OR e2 after 10 ns;
end behavior;

```

Das Zeitverhalten einer mit einer AFTER-Anweisung verzögerten Signalzuweisung zeigt folgendes Diagramm:

**2.1.2 Bedingte Signalzuweisungen in der parallelen Ebene**

Für eine bedingte Signalzuweisung stehen in der parallelen Betriebsebene zwei Konstrukte zur Verfügung. Die *Concurrent Conditional Signal Assignment Anweisung* mit dem **when/else**-Konstrukt entspricht dem **if**-Konstrukt in der sequentiellen Ebene. Die *Selected Signal Assignment Anweisung* mit dem **with/select**-Konstrukt entspricht dem **case**-Konstrukt in der sequentiellen Ebene.

Bei diesen Zuweisungen wird eine ausführliche Modulierung vorgenommen, gleichbedeutend mit einer Auflistung boolescher Bedingungen. Diese werden solange überprüft bis eine Bedingung zutrifft. Dann erst wird dem Signal ein Wert zugewiesen.

**Nomenklatur der bedingten Signalzuweisung mit WHEN/ELSE:**

```

signal <= options
    waveform1 when condition1 else
    waveform2 when condition2 else
    ...
    waveformN [when conditionN];

```

In der sequentiellen Betriebsebene wird der gleiche Effekt mit dem **if**-Konstrukt erzielt. Der Platzhalter *options* steht hierbei für die mögliche Angabe eines **transport**-Statements,

*waveform* steht für einen gültigen Signalwert mit oder ohne Verzögerung oder einen ganzen Wellenzug (beispielsweise '0' **after** 10 ns, '1' **after** 20 ns). Bezugspunkt für die Zeitangaben bei einem Wellenzug ist der Zeitpunkt, an dem eine *condition* zum ersten Mal erfüllt ist.

### **Beispiel 6:** NOR-Gatter mit Verhaltensbeschreibung

Das NOR-Gatter soll hier durch seine Verhaltensbeschreibung definiert werden. Die logische Funktion ist explizit programmiert (*conditional signal assignment*).

```
entity nor_2 is
    port (e1, e2: in BIT;
          a: out BIT);
end nor_2;

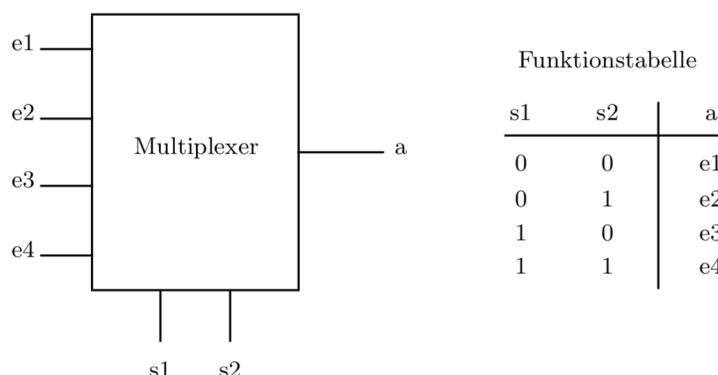
architecture behavior of nor_2 is
begin
    a <= '1' when e1 = '0' AND e2 = '0' else '0';
end behavior;
```

Nomenklatur der bedingten Signalzuweisung mit WITH/SELECT:

```
with expression select
signal <= options
    waveform1 when choices1,
    ...
    waveformN when choicesN;
```

### **Beispiel 7:** Multiplexer mit Verhaltensbeschreibung

Mit Hilfe der zwei Statements für die bedingte Signalzuweisung soll ein 4zu1-Multiplexer realisiert werden. Das Schlüsselwort **others** in der WITH/SELECT-Anweisung steht für alle Werte von „sel“, denen bis dahin noch keine Aktion zugewiesen worden ist.



```

entity multiplexer_4 is
    port (e1, e2, e3, e4, s1, s2: in BIT;
            a: out BIT);
end multiplexer_4;

architecture behavior of multiplexer_4 is
    signal sel : INTEGER;
begin
    with sel select
        a <=e1 when 0,
        e2 when 1,
        e3 when 2,
        e4 when others;
    sel <=0 when s1 = '0' AND s2 = '0' else
        1 when s1 = '0' AND s2 = '1' else
        2 when s1 = '1' AND s2 = '0' else
        3 when s1 = '1' AND s2 = '1';
end behavior;

```

Der 4 zu 1–Multiplexer hat vier Eingangsparts, zwei Steuerparts und einen Ausgangsport. Die Ports 'e1', 'e2', 'e3' und 'e4' sind für die anliegenden Signale vorgesehen, 's1' und 's2' sind für die Auswahl des durchzuschaltenden Signals verantwortlich.

Das Conditional Signal Assignment übernimmt die Funktion der Wahrheitstabelle, mit dem Selected Signal Assignment wird das zur jeweiligen Eingangskombination (an 's1','s2') zugehörige Eingangssignal an den Ausgang 'a' weitergeleitet.

### 2.1.3 Ausgabe von Fehlernachrichten in der parallelen Ebene

Das Assertion–Statement erlaubt dem Anwender Nachrichten in ein Programm einzubinden. Dies ist sehr nützlich, vor allem in großen Programmen, um erklärende Nachrichten als Rückmeldung zu benutzen.

Das Statement überprüft einen booleschen Ausdruck auf 'TRUE' oder 'FALSE'. Bei 'TRUE' läuft das Programm normal weiter, bei 'FALSE' wird ein Textstring (report message) ausgegeben. Dieser String kann unterschiedlichen Wertestufen (severity levels) zugeordnet werden:

1. NOTE  
für einfache Nachrichten
2. WARNING  
um auf mögliche Fehler bei der Simulation des Programms hinzuweisen
3. ERROR  
bei einer falschen Eingabe
4. FAILURE  
bei einem möglichen Absturz des Systems

**Nomenklatur des Concurrent Assertion Statements:**

```
assert condition
  report message
  severity level;
```

**Beispiel 8: Gatter mit Fehlernachricht**

Liegt bei dem Gatter an e1 oder e2 weder ein Wert '0' noch '1' vom Typ BIT an, gibt das Assertion-Statement eine Rückmeldung über diesen Zustand.

```
assert e1 /= 'X' and e2 /= 'X'
  report "An einem der beiden Eingaenge liegt kein eindeutiges Signal an"
  severity error;
```

**2.2 Die sequentielle Betriebsebene****2.2.1 Prozesse**

In der Betriebsebene *Sequential Level* wird nicht parallel, sondern sequentiell gearbeitet. Die sequentiellen Abschnitte eines VHDL-Programms heißen Prozesse und sind mit dem **process**-Statement in die parallele Programmierumgebung eingebettet. Ein Prozess wird in seiner Gesamtheit als *concurrent* behandelt, intern läuft der Prozess jedoch sequentiell ab. Das **process**-Statement stellt somit die Schnittstelle zwischen der parallelen und der sequentiellen Ebene dar. Die Programmierung eines Prozesses erfolgt wie bei anderen (sequentiellen) Programmiersprachen. Prozesse sind untereinander mit Signalen verbunden, über die sie kommunizieren können. Innerhalb eines Prozesses können Variablen definiert und benutzt werden, sind aber für andere Prozesse nicht sichtbar.

**Nomenklatur des PROCESS-Statements:**

```
process
  [declarations]
begin
  [statements]
end process;
```

Im Deklarationsteil sind Typen, Variablen, Konstanten, Subprograms, etc. deklarierbar. Variablen können, genauso wie Signale, nach ihrer Initialisierung ihre Werte ändern. Jedoch ist zum einen bei der Zuweisung auf Variablen keine Zeitverzögerung möglich, zum anderen sind Variablen nur lokal in einem Prozess sichtbar. Um im Schriftbild den Unterschied zwischen Signalen und Variablen hervorzuheben ist die Syntax des Variable Assignment anders (':= statt '<='). Signale können wie gewohnt gelesen und beschrieben werden.

**Nomenklatur einer Variablenzuweisung:**

`variable := expression;`

Im Bereich zwischen **begin** und **end process** werden die Statements sequentiell abgearbeitet. Wird ein Prozess ausgeführt bzw. befindet er sich in der Ausführung ist er im Zustand *active*. Die Statements werden nacheinander von der ersten bis zur letzten Anweisung (Statement) bearbeitet. Ist die letzte Anweisung ausgeführt, wird im Prozess wieder zur ersten Zeile gesprungen und die Ausführung des Prozesses wird wiederholt. Um den Zustand *active* zu verlassen und in den Zustand *suspended* überzugehen (der Prozess befindet sich in Wartestellung und wartet auf ein Ereignis, das ihn wieder aktiviert) wird ein weiteres Statement benötigt, das Wait–Statement.

**WAIT–Statement****Ereignis**

Mit dem Wait–Statement geht ein Prozess von dem Zustand *active* in den Zustand *suspended* über. Die Bedingungen unter denen er wieder aktiviert werden soll, können hierbei wie folgt spezifiziert werden:

**Nomenklatur einer wait–Anweisung:**

`wait [on signal_list] [until condition] [for time];`

Eine Wait–Anweisung besteht aus drei optionalen Teilen, die beliebig miteinander kombiniert werden können:

**on** *signal-list* spezifiziert die Signale (Sensitivitätsliste), auf deren Änderung der Prozess warten soll.

**until** *condition* gibt eine boolesche Bedingung an, die erfüllt sein muss, damit der Prozess wieder aktiviert wird. Ist keine explizite Sensitivitätsliste in Form einer *signal-list* angegeben, dann extrahiert der VHDL–Compiler alle Signale aus der Wartebedingung *condition* und erzeugt daraus eine implizite Sensitivitätsliste.

**for** *time* spezifiziert eine maximale Wartezeit (*time*) nach der der Prozess aktiviert werden soll, falls das nicht schon vorher durch einen Signalwechsel geschehen ist. Fehlen sowohl *signal-list* als auch *condition*, so wartet der Prozeß immer genau *time* Zeit-einheiten.

Zu beachten ist, dass ein Prozess letztlich nur auf Änderungen von *Signalen* oder eine vorgegebene Zeitspanne warten kann. Ist die (explizite oder implizite) Sensitivitätsliste leer, dann schläft der Prozess für den Rest der Simulation ein bzw. wird nach einer vorgegebenen Zeit wieder aktiviert, falls eine Wartezeit *time* angegeben wurde.

Der Prozess stoppt genau an der Stelle, wo er auf ein **wait**-Statement trifft und startet dann auch genau an dieser Stelle wieder.

### Beispiel 9: UND-Gatter realisiert mit process/wait on

```
entity und_3 is
    port (e1, e2, e3: in BIT := '0';
          a: out BIT);
end und_3;
architecture behavior of und_3 is
begin
    process
    begin
        a <= e1 AND e2 AND e3 after 1 ns;
        wait on e1, e2, e3;
    end process;
end behavior;
```

Soll ein Prozess bei jedem Lauf einmal komplett durchlaufen werden und immer auf die selben Signale sensitiv sein, dann kann die Sensitivitätsliste auch direkt nach dem Schlüsselwort **process** angegeben werden. Die Liste enthält dabei die Namen der Signale, auf die der Prozess triggern soll. Bei einer Änderung eines dieser Signale wird der Prozess neu angestoßen. Ein Prozess, der mit einer *sensitivity list* gesteuert wird, geht immer am Ende des Programmkkörpers in den *suspended*-Modus und beginnt im Falle einer Aktivierung seinen Programmlauf immer am mit der ersten Anweisung des Programmkkörpers. Hierbei dürfen keine **wait**-Anweisungen mehr in dem Prozess vorkommen, auch nicht in Unterprogrammen (subprogram), die der Prozess aufrufen könnte.

#### Nomenklatur eines Prozesses mit Sensitivitätsliste:

```
process (signal-list)
    [declarations]
begin
    [statements]
end process;
```

**Beispiel 10: ODER-Gatter mit Sensitivitätsliste**

```

entity oder_3 is
  port (e1, e2, e3: in BIT := '0';
          a: out BIT);
end oder_3;

architecture behavior of oder_3 is
begin
  process (e1, e2, e3)
  begin
    a <= e1 OR e2 OR e3 after 10 ns;
  end process;
end behavior;

```

**2.2.2 Fehlernachrichten in der sequentiellen Ebene**

Auch in der sequentiellen Ebene ist es möglich, Nachrichten vom Programm gesteuert auszugeben. Der Befehl und die entsprechende Nomenklatur ist identisch mit dem Befehl und der Nomenklatur in der parallelen Ebene, mit dem Unterschied, dass hier die Anweisung in einem Prozess eingebunden ist.

**2.2.3 Bedingte Anweisungen in der sequentiellen Ebene**

Zur Klasse 'condition control' gehören das **if**-Statement und das **case**-Statement. Diese Konstrukte sind aus anderen sequentiellen Programmiersprachen bekannt.

**Nomenklatur des IF-Statements:**

```

if condition then
  statements;
[else
  statements;]
end if;

```

Bei mehreren Bedingungen ist auch folgende Nomenklatur möglich, dabei darf auch **else** statt **elsif** verwendet werden:

```
if condition then
    statements;
elsif condition then
    statements;
    ...
elsif condition then
    statements;
end if;
```

#### Nomenklatur des CASE-Statements:

```
case expression is
    when value =>
        statements;
    ...
    [when others=>
        statements;]
end case;
```

Für den Platzhalter *value* gelten ein paar Regeln. Dort kann ein einzelner Wert, eine Liste von Werten oder ein Wertebereich stehen. In jedem Fall müssen die Werte Konstanten sein. Eine Werteliste ist wie folgt aufgebaut:

*value<sub>1</sub>* | *value<sub>2</sub>* | ... | *value<sub>N</sub>*

Das syntaktische Zeichen ' | ' erfüllt dabei die Funktion einer ODER-Verknüpfung.  
Ein Wertebereich wird wie folgt gekennzeichnet:

*value<sub>1</sub>* **to** *value<sub>2</sub>* oder *value<sub>1</sub>* **downto** *value<sub>2</sub>*

Bei der Angabe von Wertebereichen ist darauf zu achten, dass kein anderer Wert oder Wertebereich diesen Wertebereich überschneiden darf.

Der **when others**-Befehl muss bei Benutzung am Ende des Statements stehen. Er deckt alle noch nicht abgefragten Zustände ab.

**Achtung:** Es müssen mit den Wertangaben **alle** im Typ der *expression* auftretenden Werte abgedeckt werden. D. h. bei nicht abzählbaren Mengen **muss** der **when others**-Befehl aufgeführt werden.

#### 2.2.4 Schleifen in der sequentiellen Ebene

Wenn Statements mehrmals hintereinander ausgeführt werden sollen, werden sie als Schleife realisiert. Es gibt drei Arten, die wahllos ineinander verschachtelt werden können. Für jede Schleife kann optional ein Schleifenbezeichner 'loop-name' angegeben werden.

**Nomenklatur der einfachen Schleife:**

```
[loop_name] : loop
  [statements]
end loop [loop_name];
```

Dies ist der einfachste Schleifenaufbau. Er ist jedoch nicht ganz unproblematisch, da nicht angezeigt wird, wie die Schleife verlassen wird.

**Nomenklatur der for-Schleife:**

```
[loop_name :]
for loop_variable in range loop
  [statements]
end loop [loop_name];
```

Diese Schleifenart wird benutzt, falls die Iteration einen festgelegten Zahlenbereich umfassen soll. Die Schleifenvariable muss nicht deklariert werden. Der VHDL Compiler generiert automatisch eine *neue* Laufvariable, die aber nur innerhalb des Schleifenkörpers sichtbar ist und dort auch nur gelesen werden kann. Zu beachten ist, dass die Schleifenvariable innerhalb des Schleifenkörpers alle anderen Objekte gleichen Namens überdeckt. Der Ausdruck *range* bestimmt die Werte, die die Laufvariable annehmen soll. Hierbei können sowohl ansteigende als auch absteigende Zahlenfolgen vorgegeben werden.

**Beispiel 11: Zwei for-Schleifen**

In der ersten Schleife nimmt "i" die Werte 0, 1, 2, ... 10 in aufsteigender Reihenfolge an, während in der zweiten Schleife "i" denselben Zahlenbereich in umgekehrter Reihenfolge durchläuft.

```
for i in 0 to 10 loop
  ...
end loop;

for i in 10 downto 0 loop
  ...
end loop;
```

**Nomenklatur der While-Schleife:**

```
[loop_name :]
while condition loop
  [statements]
end loop [loop_name];
```

Soll ein Schleifenkörper abhängig von einer booleschen Bedingung *condition* ausgeführt werden, so kann man dazu eine **while**-Schleife benutzen. Jedesmal bevor der Schlei-

fenkörper ausgeführt werden soll, wird hierbei die Bedingung überprüft. Nur wenn *condition* 'TRUE' ist, wird der Schleifenkörper ausgeführt.

Mit dem **exit**-Statement können die Schleifen verlassen werden. Ist der Schleifenbezeichner 'loop\_name' nicht angegeben, wird die gerade offene Schleife verlassen.

#### Nomenklatur des exit-Statements:

```
exit [loop_name] [when condition];
```

Zum Abbrechen von Schleifen kann auch das **next**-Statement benutzt werden. Mit der Ausführung von **next** wird wieder zum Anfang der aktuellen oder bezeichneten Schleife gesprungen. Es besteht damit die Möglichkeit Schleifen nicht vollständig abarbeiten zu lassen und andere Schleifen anzuspringen. Bei einer **for**-Schleife wird zudem nach jeder Ausführung des **next**-Statements die 'loop-variable' erhöht. Bei der **while**-Schleife wird die Bedingung auf 'TRUE' überprüft, ist dies der Fall wird die laufende Schleife verlassen.

#### Nomenklatur des next-Statements:

```
next [loop-name] when condition;
```

Das **null**-Statement wird für bestimmte Zustände genutzt, in denen keine Aktion des Programms erfolgen soll.

#### Nomenklatur des null-Statements:

```
when values => null;
```

### 2.2.5 Prozeduraufrufe in der sequentiellen Ebene

Um eine bessere Übersicht von komplexen Programmen zu erreichen, ist das Procedure-Calls-Statement sehr hilfreich. Mit ihm können komplizierte Abschnitte von Prozessen isoliert werden. Durch die Kapselung sind diese Abschnitte dann auch für andere Prozesse nutzbar.

#### Nomenklatur eines Procedure-Aufrufes

```
procedure-name (association-list);
```

### 3 Simulationsablauf

Das Schema nach dem VHDL arbeitet, basiert auf dem Prinzip des Actio–Reactio–Systems wobei nur die aktiven Zeitpunkte und nur die aktiven Elemente der Schaltung simuliert werden. Die Elemente reagieren hierbei auf eine Anregung (Actio) an ihren Eingängen mit einer Antwort (Reactio) an ihren Ausgängen. Die Antwort findet – ebenso wie die Anregung – zu einem bestimmten simulierten Zeitpunkt und nicht etwa kontinuierlich statt. Außerdem reagiert ein Element mit einer Verzögerung größer oder gleich 0 auf eine Anregung, d. h. der simulierte Zeitpunkt einer Antwort ist mindestens genauso groß wie der simulierte Zeitpunkt der entsprechenden Anregung.

In einem VHDL-Modell existieren nun im Allgemeinen eine große Menge von Elementen, die parallel in der simulierten Zeit auf Anregungen reagieren und Antworten erzeugen. Um diese parallelen Vorgänge zeitlich korrekt simulieren zu können, verwendet VHDL das diskrete Event-Simulations-Verfahren.

#### 3.1 Das 2-Stufen-Modell

Ein VHDL-Modell besteht letztlich aus einer Menge von Prozessen, die über Signale miteinander kommunizieren. Die Prozesse stellen also die aktiven Elemente dar, die von Signaländerungen angeregt werden und ihrerseits mit Signalzuweisungen antworten. Alle Signalzuweisungen – auch Transactions genannt – werden aber nicht direkt auf das entsprechende Signal durchgeführt sondern zunächst vom Simulationssystem in einer sogenannten *globalen Transaction-Liste* gespeichert. Das Simulationsystem speichert hierbei neben dem Wert der Transaction auch den simulierten Zeitpunkt – auch Zeitstempel (Time-Stamp) genannt – an dem das Signal den Wert annehmen soll. Erst wenn die simulierte Zeit im Modell soweit fortgeschritten ist, dass die Zuweisung für das Signal gültig wird, weist das System dem Signal seinen neuen Wert zu.

Um die Zuweisung auf Signale und den Aufruf von Prozessen in der richtigen Reihenfolge durchführen zu können, ist das Simulation-Verfahren von VHDL in zwei Stufen unterteilt, die periodisch wiederholt werden. Eine komplette Abarbeitung der beiden Stufen wird Simulationszyklus genannt.

- Die 1. Simulationsstufe

Zuerst wird der als nächstes zu simulierende Zeitpunkt bestimmt. Dieser ergibt sich aus dem minimalen simulierten Zeitpunkt aller in der globalen Transaction-Liste gespeicherter Signalzuweisungen (Transactions). Die Simulationszeit wird auf diesen Zeitpunkt gesetzt. Sind keine Signalzuweisungen mehr vorhanden oder ist das Ende der Simulationszeit erreicht, so wird die Simulation beendet.

Andernfalls werden alle Transactions, deren Zeitstempel gleich der aktuellen Simulationszeit ist (die verarbeiteten Transactions werden anschließend aus dem System entfernt), auf die jeweiligen Signale zugewiesen. Ändert sich durch eine Zuweisung der Wert eines Signals, so markiert das System alle Prozesse, die auf das Signal sensitiv sind, um sie in der *zweiten* Stufe zu aktivieren.

- Die 2. Simulationsstufe

Hier werden nun alle Prozesse ausgeführt, die durch eine Signaländerung in der

vorhergehenden Simulationsstufe aktiviert wurden. Ein Prozess wird dabei solange ausgeführt bis er durch ein Wait-Statement in den Wartemodus übergeht. Die zweite Stufe ist abgeschlossen, wenn alle aktiven bzw. aktivierte Prozesse in den Wartemodus gegangen sind.

### 3.2 Die parallele Abarbeitung

Der Zeitstempel einer Signalzuweisung wird in VHDL aus der aktuellen Simulationszeit plus der in der Zuweisungsanweisung angegebenen Verzögerung (Delay) berechnet. Wird in der Anweisung keine Verzögerung spezifiziert, dann verwendet VHDL selbstständig eine Verzögerung ( $1d$ ). Dieser zusätzliche Wert  $1d$  erneuert bzw. verändert nicht die Simulationszeit, vielmehr wird dadurch lediglich der korrekte Ablauf der Simulationszyklen gewährleistet.

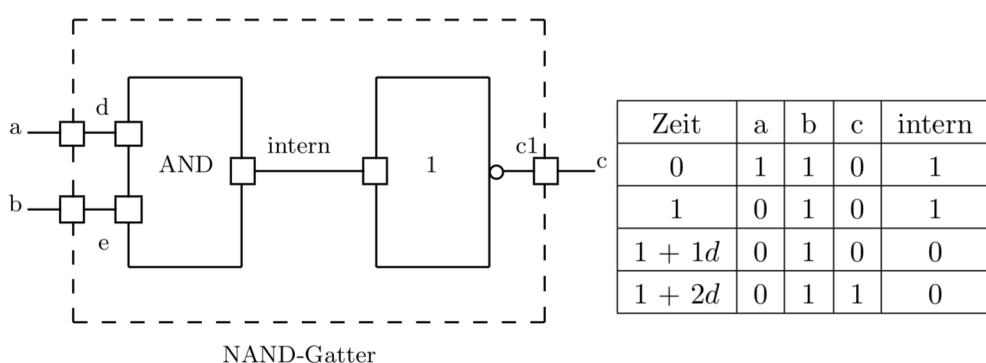
Durch die  $d$ -Verzögerung erfolgt die dazugehörige eigentliche Zuweisung des Wertes auf das Signal erst im *nächsten* Simulationszyklus, so dass alle Prozesse, die im aktuellen Zyklus ausgeführt werden, die selben Signalwerte von den Signalen lesen. Damit ist sichergestellt, dass das Verhalten des Modells unabhängig von der Reihenfolge ist, in der die Prozesse innerhalb eines Simulationszyklus ausgeführt werden.

$d$  wird auch *Delta* genannt und ist keine *reale* Zeitspanne, sondern wird als unendlich klein angenommen, weshalb selbst eine beliebig große Summe von  $d$ -Werten immer noch die Zeit 0 ergeben würde. Hierbei kann es aber sehr wohl vorkommen, dass mehrere sogenannte Delta-Zyklen, das sind Simulationszyklen, deren Simulationszeit sich nur im  $d$ -Anteil unterscheidet, hintereinander ausgeführt werden. In speziellen Fällen kann ein fehlerhaftes Modell sogar dazu führen, dass während der Simulation lediglich der Delta-Anteil der Simulationszeit anwächst.

Nach der obigen Darstellung des Simulationszyklus ist klar, dass eine Wertzuweisung, die bei einem Prozess in einer Programmzeile vorgenommen wird, nicht sofort, sondern frühestens ein *Delta* später (also frühestens im nächsten Simulationszyklus) vom Signal gelesen werden kann. Während die Prozesse in einem Simulationszyklus ausgeführt werden, sind die Signalwerte also *konstant*.

#### Beispiel 12:

Das Nand-Gatter aus Kapitel 1.2 wurde bei den Signalzuweisungen ohne Verzögerungszeit realisiert.



Zu Beginn der Simulation haben beide Eingänge (a,b) den Wert 1, der Ausgang c ist 0, demzufolge ist *intern* auf 1. Der Wert von a wird nun auf 0 geändert. Zum Zeitpunkt  $t = 1$  vollzieht sich keine Änderung am Ausgang c und auch nicht am Signal *intern*. Nach der ersten Iteration (Zeitpunkt  $t = 1 + 1d$ ) ändert sich das Signal *intern* und nimmt den Wert 0 an. c bleibt immer noch unverändert. Erst nach einer weiteren Iteration (Zeitpunkt  $t = 1 + 2d$ ) nimmt der Ausgang c seinen endgültigen Wert an. Trotz dieser Iterationen wird dem Ausgang c in der Simulation sofort das Ergebnis zugewiesen, das sich nach dem Ablauf der letzten Iteration einstellt.

### 3.3 Verarbeitung von Signalzuweisungen

Neben der globalen Transaction-Liste verwaltet der Kernel für jeden Prozess und jedes Signal, auf das der Prozess eine Zuweisung durchführen kann, eine *lokale* (so genannte) Driver-Liste. Diese enthält ausschließlich Zuweisungen *eines* Prozesses auf ein bestimmtes Signal. Eine Transaction wird also immer in *zwei* Listen eingefügt: einmal in die globale Transaction-Liste und einmal in eine Driver-Liste.

Wichtig ist, dass ein Prozess immer nur *eine* Driver-Liste für ein einfaches Signal bzw. für jedes Element eines zusammengesetzten Signals besitzt, unabhängig davon, wie viele Signalzuweisungen für das Signal innerhalb des Prozesskörpers vorhanden sind. Für die Bestimmung der Driver-Listen spielt es dabei keine Rolle, ob bzw. wie oft während der Simulation eine Signalzuweisung tatsächlich ausgeführt wird.

#### Beispiel 13: Prozess mit drei Signalzuweisungen

Der Prozess besitzt jeweils eine Driver-Liste für das Signal "a" und "b".

```
process
begin
    a <= e1 AND e2 AND e3 after 1 ns;
    if b = '1' then
        b <= e2;
        a <= not e3;
    end if;
    wait on e1, e2, e3;
end process;
```

Alle Signalzuweisungen werden dabei nach dem im VHDL-Standard definierten "Reader-Driver-Konzept" abgewickelt:

- Reader: Der Reader eines Signals trägt den zum aktuellen Simulationszeitpunkt gültigen Wert des Signals.
- Driver: Der Driver eines Signals ist eine Liste mit zukünftigen, also noch nicht auf den Reader zugewiesenen Transactions. Jedem Signal-Prozess-Paar ist dabei eine *eigene* Driver-Liste zugeordnet. Wird eine Transaction  $TR_n = (Wert_n, t_n)$  -  $Wert_n$  ist der Wert der Transaction und  $t_n$  sein Zeitstempel - vom Prozesscode erzeugt, so werden die folgenden Regeln angewendet, um die Driver-Liste zu aktualisieren:

1. Lösche alle Transactions  $TR_i$  mit  $t_i \geq t_n$  aus der Driver-Liste.

2. Ist die Verzögerungsart **inertial**, dann suche die Transaction  $TR_k$  mit dem größten Time-Stamp  $t_k$  für die  $Wert_k \neq Wert_n$  gilt. Falls  $TR_k$  existiert, lösche alle Transactions  $TR_j$  mit  $t_n - t_{prl} \leq t_j \leq t_k$ .

Der Wert  $t_{prl}$  ist dabei die “Puls-Reject-Grenze”, die in der Zuweisung angegeben wurde.

Mit Hilfe des Reader/Driver-Konzepts und der Regeln, mit denen die Driver-Listen manipuliert werden, können Impulse auf einfache Art und Weise unterdrückt werden (siehe auch Kapitel 2.1.1).

Durch eine Signalzuweisung kann nicht nur *eine* neue Transaction für ein Signal erzeugt, sondern ebenso eine ganze Kette von Transactions – eine sogenannte *Waveform* – spezifiziert werden. Beim Einfügen der Transactions in die Driver-Liste ist zu beachten, dass nur für die erste Transaction der Kette gegebenenfalls der **inertial**-Verzögerungsmechanismus verwendet wird. Alle übrigen Transactions werden mit **transport**-Verzögerung eingefügt.

#### **Beispiel 14:**

Wird die aktuelle Simulationszeit als  $t_{sim}$  bezeichnet, dann generiert die Zeile eine Transaction für den Simulationszeitpunkt  $t_{sim} + 11\text{ns}$ , eine für den Zeitpunkt  $t_{sim} + 15\text{ns}$  und eine für  $t_{sim} + 19\text{ns}$ .

```
sig <= 70 after 11 ns, 80 after 15 ns, 55 after 19 ns;
```

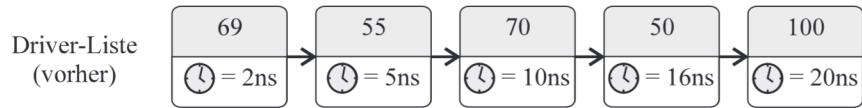
Das folgende Beispiel soll die Regeln zum Einfügen von Transactions in Driver-Listen verdeutlichen.

#### **Beispiel 15:**

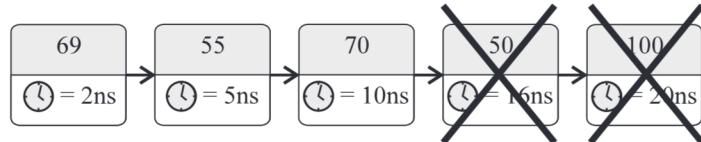
Zum Simulationszeitpunkt 1 ns wird die folgende Signalzuweisung auf das Signal “sig” in einem Prozess ausgeführt:

```
sig <= reject 9 ns inertial 70 after 11 ns, 55 after 19 ns;
```

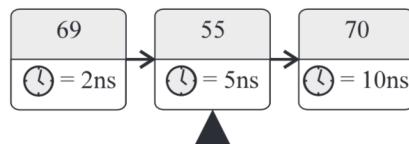
In der entsprechenden Driver-Liste seien zu Beginn der Zuweisung schon 5 Einträge vorhanden. In dem Beispiel werden Transactions durch ein Rechteck symbolisiert, bei dem der Wert der Transaction im oberen Feld und der Zeitstempel im unteren Feld eingetragen sind.



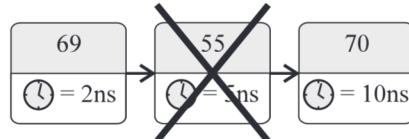
1. Alle Transactions mit einem Time-Stamp groesser gleich 12ns loeschen



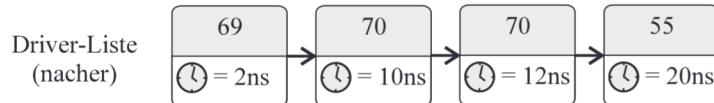
2. Die groesste Transactions mit einem Wert ungleich 70 suchen



3. Alle Transactions mit einem Time-Stamp zwischen 12ns-9ns=3ns (einschliesslich) und 5ns (einschliesslich) loeschen



3. Neue Transaction(s) an das Ende der Liste haengen



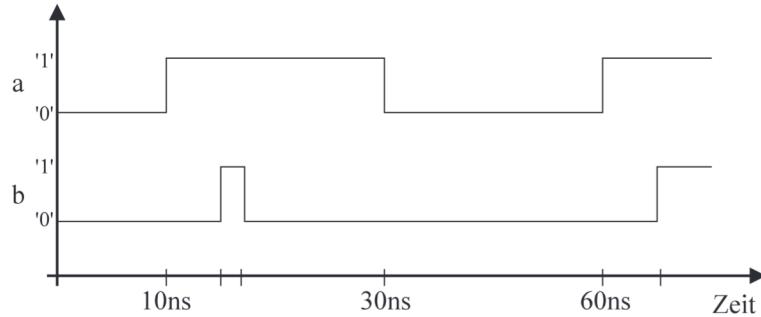
### Beispiel 16:

Das folgende Beispiel soll die Wirkung des **inertial**-Verzögerungsmechanismus an einem einfachen Modell verdeutlichen. Die Architecture besteht aus drei parallelen Signalzuweisungen, wobei die ersten beiden Zuweisungen lediglich die "Stimuli" für die dritte erzeugen.

```

architecture struct of test is
  signal a, b, erg : BIT := '0';
begin
  -- Parallelzuweisung ①
  a <= '1' after 10 ns, '0' after 30 ns, '1' after 60 ns;
  -- Parallelzuweisung ②
  b <= '1' after 15 ns, '0' after 17 ns, '1' after 65 ns;
  -- Parallelzuweisung ③
  erg <= reject 8 ns inertial a AND b after 10 ns;
end struct;
```

Die ersten beiden Parallelzuweisungen in der Architecture erzeugen den folgenden Signalverlauf für "a" und "b":

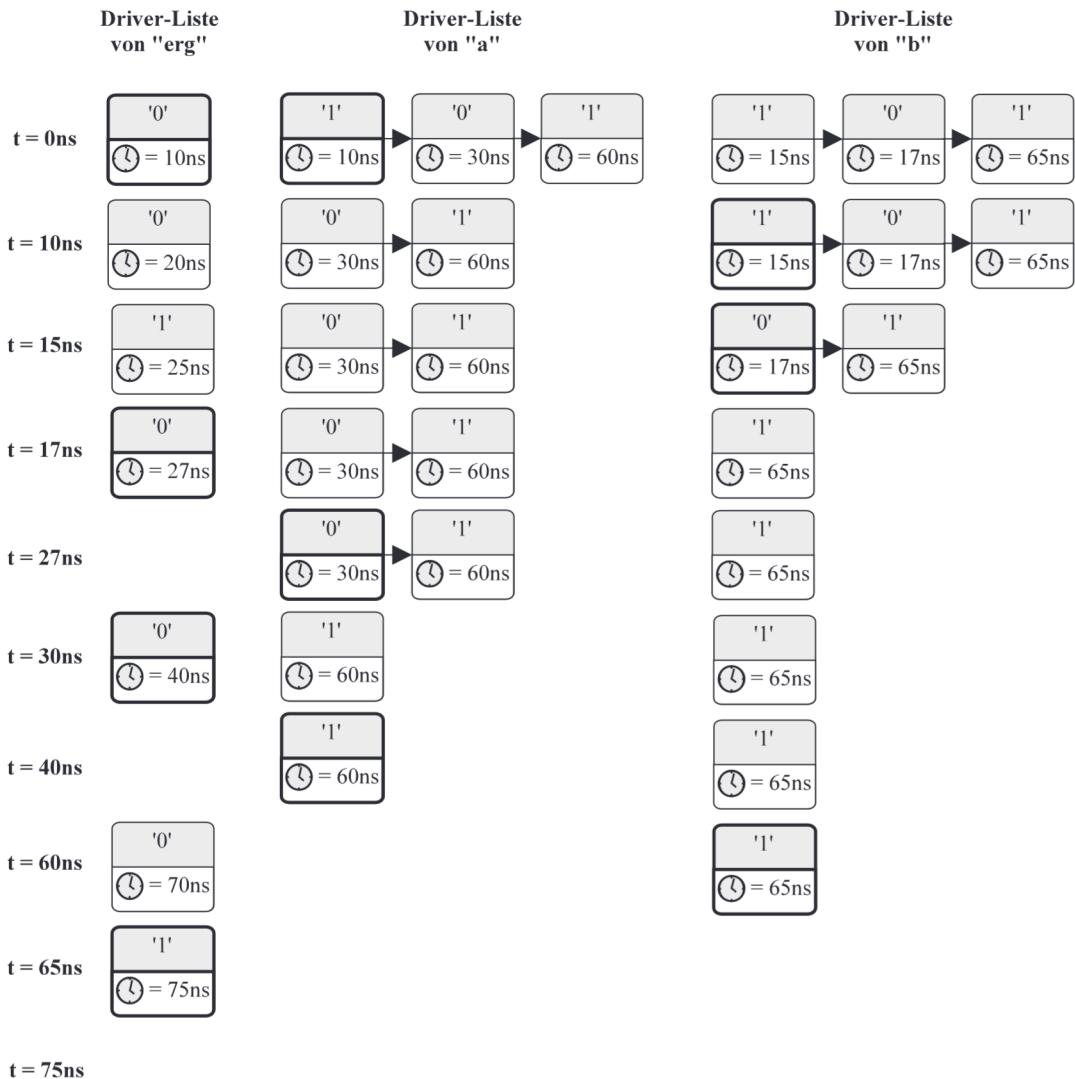


Die folgende Abbildung zeigt die Driver-Listen der Signale "erg", "a" und "b" am Ende der verschiedenen Simulationszeitpunkte.

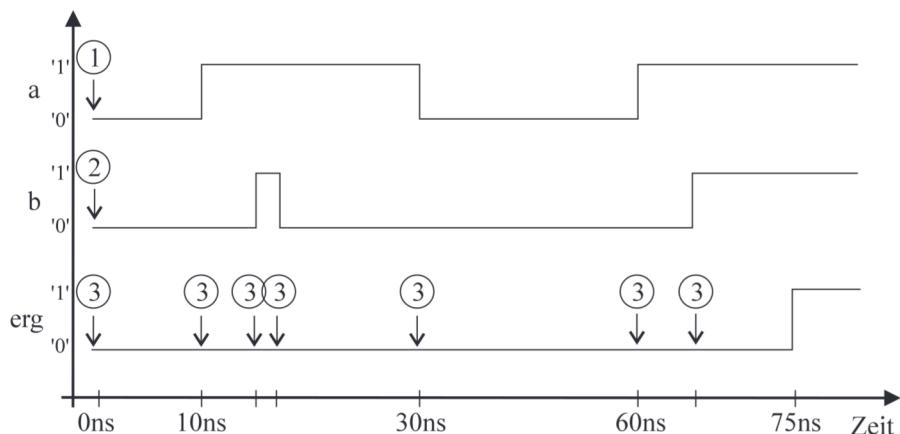
Zu Beginn der Simulation wird jeder Prozess einmal ausgeführt. Das erzeugt für  $t = 0\text{ns}$  jeweils drei Transactions für die Driver-Liste von "a" und "b", sowie eine Transaction für die Driver-Liste von "erg". Die jüngste Transaction (also die Transaction mit dem kleinsten Zeitstempel) trägt den Time-Stamp  $t = 10\text{ns}$ . Daher wird die Simulationszeit für den nächsten Simulationszyklus auf  $t = 10\text{ns}$  gesetzt.

Zum Zeitpunkt  $t = 10\text{ns}$  wird nun die Transaction aus der Driver-Liste für "erg" sowie die erste Transaction aus der Driver-Liste für "a" entfernt und auf die entsprechenden Signale zugewiesen. Die Zuweisung für "a" erzeugt einen Event, da der neue Wert ('1') sich vom alten Wert ('0') unterscheidet. Daher wird die Parallelzuweisung ③ ausgeführt, wodurch eine neue Transaction für den Simulationszeitpunkt  $t = 10\text{ns} + 10\text{ns} = 20\text{ns}$  entsteht. Der Wert der Transaction berechnet sich zu ' $1 \text{ AND } 0 = 0$ '. Die entsprechende Zeile  $t = 10\text{ns}$  in der Abbildung zeigt den Inhalt der Driver-Listen am Ende des Simulationszeitpunktes.

Im nächsten Simulationsschritt wird die Simulationszeit auf  $t = 15\text{ns}$  gesetzt. Danach wird die erste Transaction aus der Driver-Liste von "b" entfernt und auf das Signal zugewiesen. Dadurch ändert sich der Wert von "b", was wiederum die Ausführung von Zuweisung ③ nach sich zieht. Die daraus entstehende neue Transaction für die Driver-Liste von "erg" trägt den Wert '1' und den Zeitstempel  $t = 25\text{ns}$ . Aufgrund des **inertial**-Verzögerungsmechanismus wird dabei die alte Transaction mit dem Wert '0' und dem Zeitstempel  $t = 20\text{ns}$  aus dem System gelöscht, ohne vorher auf das Signal zugewiesen worden zu sein. In der Abbildung ist in der Zeile  $t = 15\text{ns}$  die Belegung der Driver-Listen am Ende dieses Simulationszyklusses zu sehen. Die nachfolgenden Simulationsschritte ergeben sich analog aus der eben beschriebenen Vorgehensweise.



Somit ergibt sich für den Signalverlauf von "a", "b" und "erg" (Die Pfeile markieren die Zeitpunkte, zu denen die Parallelzuweisungen ①, ② oder ③ ausgeführt worden sind):



### 3.3.1 Signalzuweisungen ohne Verzögerung

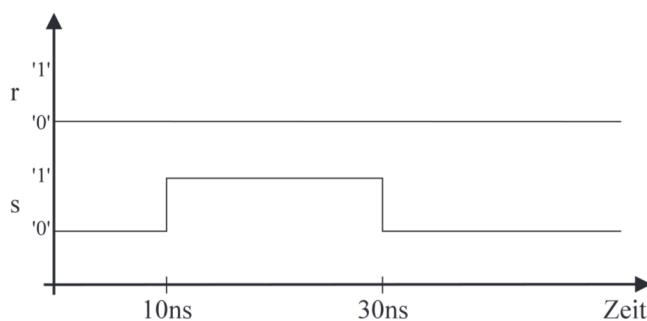
Wie in Kapitel 3.2 schon erläutert wurde, können bei der Ausführung von Prozessen oder parallelen Signalzuweisungen auch Transactions (Zuweisungen) ohne Verzögerung erzeugt werden. Zwar entspricht deren Time-Stamp der aktuellen virtuellen Simulationszeit, trotzdem wird der Wert einer solchen Transaction erst im *nächsten* Simulationszyklus auf den Reader des Signals zugewiesen.

#### Beispiel 17:

Modell eines RS-Flipflops mit Signalzuweisungen ohne Verzögerung. Die Funktion des Bausteins wird über zwei kreuzweise rückgekoppelte NOR-Gatter realisiert.

```
architecture struct of test2 is
    signal r, s, q : BIT := '0';
    signal qn : BIT := '1';
begin
    -- Parallelzuweisung ①
    r <= '0';
    -- Parallelzuweisung ②
    s <= '1' after 10 ns, '0' after 30 ns;
    -- Parallelzuweisung ③
    q <= qn NOR r;
    -- Parallelzuweisung ④
    qn <= q NOR s;
end struct;
```

Die ersten beiden Parallelzuweisungen in der Architecture erzeugen den folgenden Signalverlauf für "r" und "s":



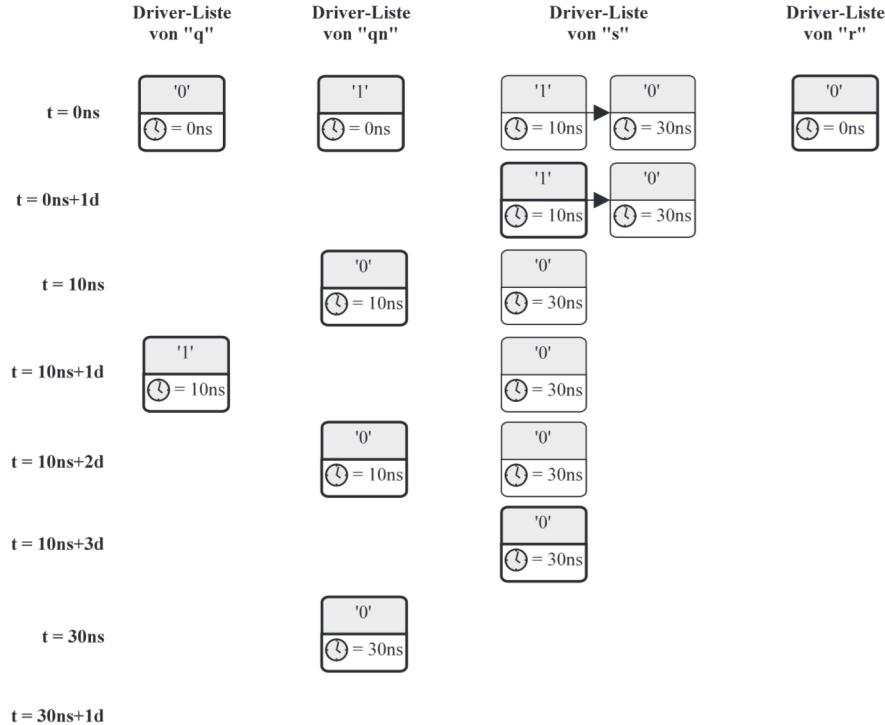
Die folgende Abbildung zeigt die Driver-Listen der Signale “r”, “s”, “q” und “qn” am Ende der verschiedenen Simulationszeitpunkte.

Zu Beginn der Simulation wird jeder Prozess einmal ausgeführt. Das erzeugt für  $t = 0\text{ns}$  zwei Transactions für die Driver-Liste von “s” und jeweils eine Transaction für alle übrigen Driver-Listen. Die jüngste Transaction (also die Transaction mit dem kleinsten Zeitstempel) trägt den Zeitstempel  $t = 0\text{ns}$ , ist also identisch zur aktuellen Simulationszeit. Daher wird die Simulationszeit für den nächsten Simulationszyklus auf  $t = 0\text{ns} + 1\text{d}$  gesetzt. Der nächste Simulationszyklus hat also dieselbe reale Zeit wie der vorhergehende und ist dementsprechend ein Delta-Zyklus.

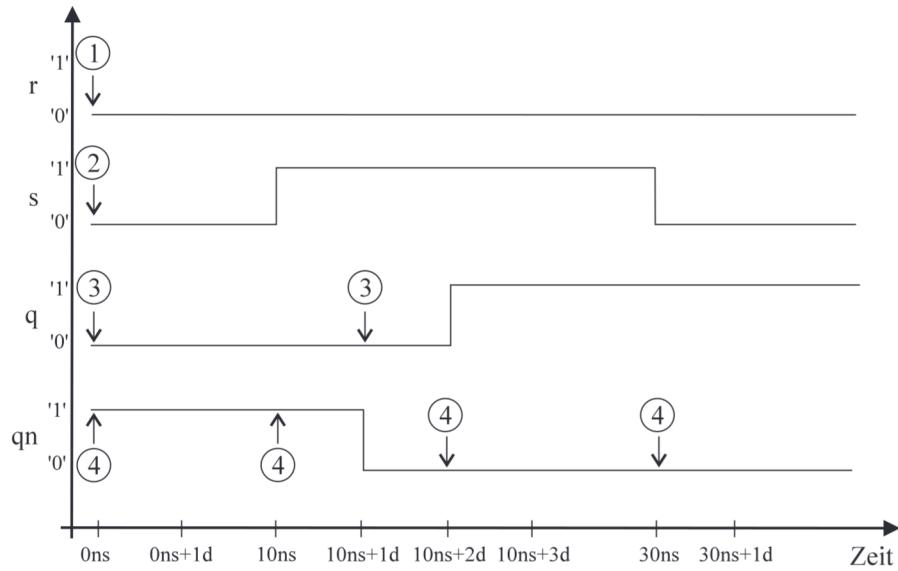
Zum Zeitpunkt  $t = 0\text{ns} + 1\text{d}$  werden nun die Transactions der Driver “q”, “qn” und “r” auf die entsprechenden Signale zugewiesen. In diesem Fall werden durch die Zuweisungen keine neuen Signalzuweisungen angestoßen und somit auch keine neuen Transactions erzeugt. Daher ist der nächste Simulationszeitpunkt  $t = 10\text{ns}$ .

Zum Zeitpunkt  $t = 10\text{ns}$  wird die entsprechende Transaction aus Driver-Liste “s” entnommen und auf das Signal zugewiesen. Durch das daraus entstehenden Event wird die Parallelzuweisung ④ zur Ausführung gebracht und erzeugt eine neue Transaction mit Wert ’0’ zum Zeitpunkt  $t = 10\text{ns}$  für das Signal “qn”. Da der Zeitstempel der jüngsten Transaction der aktuellen Simulationszeit entspricht, ist der nachfolgende Simulationszyklus wieder eine Delta-Zyklus.

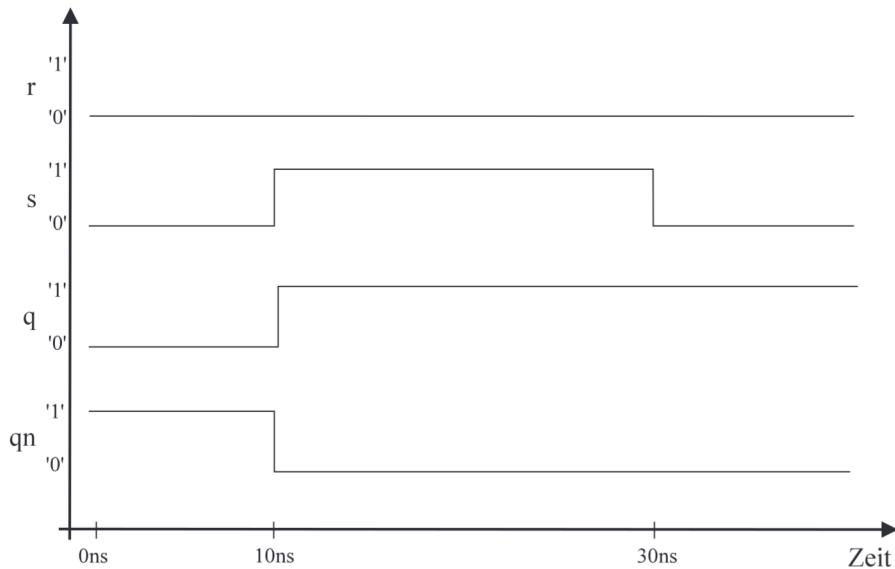
Im nächsten Zyklus  $t = 10\text{ns} + 1\text{d}$  ändert sich der Wert von signal “qn”, was die Ausführung des Parallelzuweisung ③ initiiert. Daraus entsteht eine neue Transaction für den Driver von “q” mit dem Wert ’1’ und dem Zeitstempel  $t = 10\text{ns}$ . Der nächste Simulationszyklus hat daher den Zeitwert  $t = 10\text{ns} + 2\text{d}$ . Die nachfolgenden Simulationsschritte erfolgen analog zu der eben beschriebenen Vorgehensweise.



Zusammenfassend ergibt sich der folgende Signalverlauf für "r", "s", "q" und "qn" (Die Pfeile markieren die Zeitpunkte, zu denen die Parallelzuweisungen ①, ②, ③ oder ④ ausgeführt worden sind):



Werden alle zusammengehörigen “Delta-Schritte” zu einem Zeitpunkt verschmolzen, so ergibt sich der erwartete Kurvenverlauf. Wie in der nächsten Abbildung zu sehen ist, reagiert das RS-Flipflop sofort –also ohne Verzögerung– auf die Eingangs-Stimuli:



## 4 Datentypen

Die Werte eines Objektes und die ausführbaren Operationen werden durch die Datentypen definiert. Mit dieser Festlegung ist eine saubere Definition der Datenpfade im Programm realisierbar. Falsche Wertzuweisungen (ein Integer-Wert wird z. B. einem digitalem Signal zugewiesen) werden nicht ausgeführt. In VHDL stehen vier Hauptarten von Typen zur Verfügung.

- SCALAR-Typen
- COMPOSITE-Typen
- ACCESS-Typen
- FILE-Typen

### 4.1 Scalar-Typen

Typen die zur Klasse der Skalare gehören, bestehen aus Werten, die keine eigenen Elemente mehr enthalten. Dazu gehören

- INTEGER-Typen
- real-Typen
- enumeration-Typen
- physical-Typen

Neben diesen vordefinierten Typen kann man aber auch eigene erzeugen. Die allgemeine Syntax um einen neuen Typen zu erzeugen ist

**type identifier is type-definition;**

Die *type-definition* definiert den zulässigen Wertebereichen des Typs. Der Wertebereich kann hierbei wie folgt definiert werden:

- Bei den Integer- und Real-Typen ist die *type-definition* ein 'range-constraint'. Das bedeutet, dass hier sowohl die Grenzen des Wertebereiches, als auch die Bereichsrichtung (auf- oder absteigend), angegeben werden kann. Formal wird dies durch die Worte RANGE und TO bzw. DOWNTO angezeigt.

Die Zahlen, die den gültigen Wertebereich markieren, definieren zudem auch ob es sich bei dem neuen Typ um einen Integer- oder einen Real-Typen handelt. Enthalten die Zahlen einen Dezimalpunkt, so handelt es sich um einen Real-Typen, anderenfalls um einen Integer-Typen.

#### **Beispiel 18: Deklaration eines neuen real-Typen**

Hier ist ein Realbereich von -11.5 bis 33.3 deklariert worden. Daher sind die Werte 33.1 oder -9.7 für den Typ *myreal* zulässig, da beide im Wertebereich liegen, während die Zuweisung des Wertes 33.4 auf z.B. eine Variable vom Typ *myreal* eine Fehlermeldung verursachen würde. Die auf- bzw. absteigende Zählrichtung ist vor allem bei der Definition von Arrays sehr nützlich.

**type myreal is range -11.5 to 33.3;**

- Enumeration-Typen sind für das abstrakte Modellieren von Vorteil. Alle Werte von diesem Typus sind benutzerdefiniert und müssen explizit in der Deklaration aufgeführt werden. Erlaubt sind einzelne Zeichen und Wörter, wobei erstere jeweils in Hochkommata einzuschließen sind.

**Beispiel 19:** Ein aus drei Werten bestehender *Enumeration*-Typ

Mit der Deklaration dieses Typs kann z. B. einem digitalen Eingangssignal auch ein unbestimmter Wert (hier 'X') zugeordnet werden, ohne dass der Compiler dies als falsche Zuweisung melden würde. Es ist jedoch zu beachten, dass die Werte des Typen für VHDL keine weitere Bedeutung haben. Die bekommen sie erst, wenn entsprechende Operatoren für den neuen Typen definiert werden.

```
type drei_stufig is ('X', '0', '1');
```

- Der vierte skalare Datentyp ist der physical-Typ. Er dient hauptsächlich dazu physikalische Größen darzustellen. Ein physical Wert besteht aus zwei Komponenten:
  - einer Einheit und
  - einem Zahlenwert, welcher die Größe des Wertes bezogen auf die Einheiten angibt.

Die Basiseinheit eines physical-Typen muss immer spezifiziert werden; alle anderen Einheiten sind optional. Der kleinste Wert eines physical Typen besteht nur aus einer einzigen Basiseinheit.

**Beispiel 20:** Ein *physical*-Typ zur Darstellung von Spannung

```
type Spannung is range 0 to 380000000
units
  uV;    -- Microvolt
  mV = 1000 uV;  -- Millivolt
  V = 1000 mV;   -- Volt
end units;
```

Die erste Zeile gibt den Typ sowie den dazugehörigen Wertebereich (Range) an. Der Bereich bezieht sich auf die Basiseinheit. Danach wird die Basiseinheit (hier Microvolt) mit den anderen darauf aufbauenden Einheiten angegeben. Die Units sind dabei ein Vielfaches der Basiseinheit oder durch vorhergehende Einheiten definiert. In diesem Beispiel ist "Volt" über die Einheit "Millivolt" definiert worden.

In VHDL existiert nur der vordefinierte physical-Typ TIME:

```

type TIME is range -(2**31-1) to (2**31-1)
units
    fs;
    ps = 1000 fs;
    ns = 1000 ps;
    us = 1000 ns;
    ms = 1000 us;
    sec = 1000 ms;
    min = 60 sec;
    hr = 60 min;
end units;

```

## 4.2 Composite Typen

Arrays und Records gehören zur Klasse der Composite Typen. Dies sind Objekte, die aus mehreren Elementen bestehen. Der Unterschied zwischen Arrays und Records ist, dass bei Arrays alle Elemente vom gleichen Datentyp sind, während bei Records verschiedene Datentypen zusammengefasst werden können.

- **Constrained Arrays**

Mit Arrays besteht die Möglichkeit Felder mit einer beliebigen Anzahl von Dimensionen zu definieren. Als Index können sowohl ganzzahlige Werte (Integer) als auch Enumeration-Typen dienen.

**Nomenklatur einer Constrained Array-Typ-Definition:**

**type** *word* **is array** (*Range*) **of** *type-mark*;

Range definiert für jede einzelne Dimension die Werte, die der Index annehmen kann. Es können hierbei die obere Indexgrenze, die untere Indexgrenze sowie die Bereichsrichtung (auf- oder absteigend) definiert werden.

**Beispiel 21:** Ein paar Feldtypdefinitionen

```

type Matrix2d is array (1 to 10, 1 to 25) of BOOLEAN;
type Matrix1d is array (10 downto 1) of INTEGER;
type Matrix1dM2d is array (10 downto 1) of Matrix2d;

```

- **Unconstrained Arrays**

Unconstrained Arrays sind Feld-Typen deren Indexbereich nicht speziell angegeben ist und damit auch nicht deren Größe. Sie werden meist als Basistypen benutzt, von denen dann so genannte Subtypen erstellt werden können. Zwischen (Sub-)Typen, die von einem gemeinsamen Basistypen abgeleitet worden sind, herrscht eine “Verwandtschaftsbeziehung”. So können z. B. verwandte Typen aufeinander zugewiesen werden.

**Nomenklatur für eine eindimensionalen unconstrained Array-Definition:**

```
type neuer_typ_name is array (index_typ range <>) of element_typ
```

*index\\_typ* ist ein Integertyp, der den maximalen Bereich bezeichnet, innerhalb dessen sich der Index bewegen darf. Das Schlüsselwort **range** mit dem anschließenden Zeichen <> (Box genannt) zeigt an, dass es sich hier um eine *unconstrained*-Array-Typ-Definition handelt.

#### Beispiel 22: Zwei in VHDL vordefinierte *unconstrained*-Arrays

```
type BIT_VECTOR is array (NATURAL range <>) of BIT;
```

Der erste Basistyp deklariert ein Array vom Typ Bit. Die Anzahl der Elemente (Bits) ist nicht näher angegeben. Dies wird durch das Schlüsselwort **range** <> angezeigt. Der Type "NATURAL" vor dem Schlüsselwort ist ein vordefinierter Integertyp der den Wertebereich von 0 bis INTEGER'HIGH<sup>1</sup> angibt. Jedes Element vom Typ BIT-VECTOR ist vom Typ BIT.

```
type STRING is array (POSITIVE range <>) of CHARACTER;
```

Der zweite vordefinierte Basistyp ist ein Array vom Typ String. Jedes Element dieses Arrays ist vom Typ CHARACTER. Die unspezifizierten Grenzen des Arrays werden wieder durch das Schlüsselwort **range** <> angezeigt. Der Typ "POSITIVE" umfasst einen Bereich von 1 bis "INTEGER'HIGH".

Auch wenn man in VHDL *unconstrained*-Feld-Typen definieren kann, so ist es dennoch nicht möglich Feldobjekte mit einer variablen Länge anzulegen. Vielmehr bekommt ein solches Feldobjekt während seiner Erzeugung eine Größe zugewiesen, welche nachträglich nicht mehr verändert werden kann. Eine Möglichkeit die Größe eines unconstrained Feld-Objektes zu definieren, ist die Spezifikation der Grenzen bei der Objektdeklaration:

---

<sup>1</sup>Attribut, mit dem die obere Grenze eines Typs (hier INTEGER) angesprochen wird. Näheres siehe Kapitel 6.

**Beispiel 23:** Definition der Feldgrenzen bei der Objektdeklaration

Die folgenden Variablen-Deklarationen definieren drei Feld-Objekte, die jeweils aus 8 BIT-Werten bestehen. Die Deklarationen erzeugen jeweils einen internen Typ, der mit dem unconstrained Array-Typ BIT\_VECTOR verwandt ist. Durch diese Verwandtschaftsbeziehung sind z. B. Zuweisungen zwischen den Variablen erlaubt. Hätte man drei eigenständige Feld-Typen benutzt, so wären die Zuweisungen illegal.

```
variable var1 : BIT_VECTOR(0 to 7);
variable var2 : BIT_VECTOR(7 downto 0);
variable var3 : BIT_VECTOR(15 to 8);
...
var1 := var2;
var2 := var3;
var3 := var1;
```

Außerdem können auch Array-Ports und Generic-Parameter von einem *unconstrained*-Typ sein. Zwar müssen auch dann die Feldgrenzen festgelegt werden, das geschieht aber erst wenn die entsprechende Komponente instantiiert wird.

**Beispiel 24:** Eine Komponente mit *unconstrained*-Array-Signalen

Port-Signale oder auch Generics können von einem unconstrained Feld-Typ sein. Ihre Größe bekommen sie zugewiesen, wenn die Komponente instantiiert wird. Die Grenzen werden dann von den Signalen bzw. Werten bestimmt, die mit den Port-Signalen bzw. Generics verbunden werden.

```
entity preset is
  generic (delay : TIME := 10 ns);
  port ( gate: in BIT;
         preset_val: in BIT_VECTOR;
         data, erg: inout BIT_VECTOR);
end preset;
architecture struct of preset is
  signal preset_sig : BIT_VECTOR(preset_val'range);
  -- preset_val'range ist ein sogenanntes Attribut und liefert den
  -- Index-Bereich des Vektors "preset_val". Dementsprechend ist also
  -- "preset_sig" ein Feld mit demselben Index-Bereich und der
  -- gleichen Index-Bereichsrichtung wie "preset_val".
begin
  preset_sig <= preset_val after delay;
  erg <= data when gate = '0' else preset_sig;
end struct;
```

**Beispiel 25:** Instanziierung der Komponente aus vorherigem Beispiel

```

subtype word is BIT_VECTOR (15 downto 0);
subtype byte is BIT_VECTOR (7 downto 0);
signal contr : BIT;
signal daten, dbus, init_daten : byte;
signal adr, abus, init_adr : word;
...
-- Diese Instanz von "preset" hat einen Vektor "erg" mit
-- 8 Elementen.
k1: entity preset
    port map (preset_val => init_daten,
                data => daten, gate => contr,
                erg => dbus);
-- während die folgende Instanz einen Vektor "erg" mit 16
-- Elementen besitzt.
k2: entity preset
    port map (preset_val => init_adr,
                data => adr, gate => contr,
                erg => abus);

```

Des Weiteren können aber auch so genannte Subtypen von *unconstrained*-Feldtypen abgeleitet werden. Diese können dann wie normale Typen bei der Deklaration von Objekten eingesetzt werden.

**Beispiel 26:** Ableitung eines *unconstrained*-Arrays

Das Schlüsselwort PACKAGE und deren genaue Anwendung wird in Kapitel 6 näher erläutert. Parameter eines Unterprogrammes können von einem *unconstrained*-Array-Typ sein. Die tatsächlichen Grenzen werden dann von Aufruf zu Aufruf von dem Wert bestimmt, der an den Parameter jeweils übergeben wird.

```

package paket1 is
    subtype bit12 is BIT_VECTOR(0 to 11);
    subtype bit4 is BIT_VECTOR(0 to 3);
    signal sig1 : bit12;
    constant sig2 : bit4 := ('0','0','0','0');
    function Addition(Erg: BIT_VECTOR) return BIT_VECTOR;
end paket1;
...
sig1 <= Addition(sig2);

```

- Records

Records sind zusammengesetzte Typen, die mehrere Elemente von verschiedenen Datentypen enthalten können. Records können aus allen Arten von Datentypen aufgebaut werden, wobei auch Arrays und Records selber vorkommen können.

**Nomenklatur einer Record-Typ-Definition:**

```
type record_name is
  record
    [element-declarations]
  end record;
```

**Beispiel 27: Zwei Record-Definitionen**

```
type wort is array 0 to 15 of CHARACTER;
type bit3 is ('0', '1', 'X');
type Adresse is
  record
    source: INTEGER;
    drain: INTEGER;
  end record;
type Paket is
  record
    Name: wort;
    E1: bit3;
    E2: bit3;
    Ziel: Adresse;
  end record;
```

Zeile 1 definiert einen Array mit einem Speicherbereich von 16 Buchstaben. In Zeile 2 wird der Typ 'bit3' deklariert, der nur die Werte '0', '1' und 'X' zulässt. "Adresse" ist ein Record, der Source und Drain (beides Integerwerte) zu einem Element zusammenfasst.

Diese Deklarationen werden im Typ "Paket" zu einem Objekt vereinigt, welches in einem Prozess so aufgerufen werden könnte:

```
process
begin
  ...  Paket := ("yes", '1', '1', (12, 3));
end process;
```

### 4.3 Access Typen

Die aus anderen Sprachen bekannten *Pointer (Zeiger)* heißen in VHDL Access Typen. Zeiger sind Datentypen die auf Objekte verweisen, für die Speicherplatz assoziiert worden ist. Nur Variablen dürfen als Access Typen deklariert werden, demzufolge können Zeiger

auch nur in sequentiellen Prozessen vorkommen. Dynamische Listen oder FIFOs (**First In First Out**) sind Anwendungsbeispiele für Access Typen.

#### 4.4 File Typen

In VHDL können auch Dateien geschrieben und gelesen werden. Der Zugriff auf Dateien erfolgt über spezielle File Typen.

#### 4.5 Subtypen

Typen, die in VHDL über verschiedene Typdeklarationen erzeugt worden sind, sind zueinander *inkompatibel*, selbst wenn die Typdeklarationen (abgesehen vom Typnamen) identisch sind. So existieren z. B. keinen Operatoren (wie z. B. Vergleichsoperatoren, Zuweisungoperatoren, ...) zwischen solchen Typen.

Im Gegensatz dazu können mit Hilfe von Subtypen Typgruppen erzeugt werden. Trotzdem die Typen der Gruppe sich dabei in bestimmten Eigenschaften unterscheiden können, sind dennoch bestimmte Operationen innerhalb der gesamten Gruppe erlaubt. So können z. B. Zuweisung zwischen den Typen der Gruppe durchgeführt werden.

Ein Subtyp wird in VHDL mit Hilfe einer sogenannten Subtyp-Deklaration erzeugt.

**Nomenklatur einer Subtyp-Definition:**

```
subtype identifier is subtype-indication;
```

Die 'subtype-indication' gibt den Basisdatentyp sowie den begrenzten Wertebereich des Subtyps an. Der begrenzte Wertebereich ist entweder RANGE oder INDEX. Der Bereich RANGE ist von den skalaren Typen her bekannt und definiert den gültigen Wertebereich des neuen Typs. Hierbei muss der Bereich aber im Wertebereich des Basisdatentypen enthalten sein.

Mit INDEX wird einem Array, dass ohne bestimmte Grenzen definiert worden ist (unconstrained Array), ein fester Bereich für jede Dimension zugeordnet. Auch hier gilt, dass der neue Indexbereich im Indexbereich des unbegrenzten Feldes enthalten sein muss.

#### Beispiel 28: Deklaration von Subtypen

```
subtype fourlevel is INTEGER range 0 to 3;
subtype wahrscheinlichkeit is REAL range 0.0 to 1.0;
subtype countdown is INTEGER range 10 downto 0;
subtype byte is BIT-VECTOR(0 to 7);
subtype word is BIT-VECTOR(15 downto 0);
```

Von einem Subtyp können nun wie gewohnt Objekte erzeugt werden.

**Beispiel 29:** Anwendung von Subtypen

```
signal sig1 : fourlevel;
constant immer : wahrscheinlichkeit := 1.0;
signal start : countdown;
signal daten byte;
signal adresse : word;
```

## 5 Unterprogramme

Unterprogramme sind eine Sequenz von Statements (inkl. Wait-Statements) und Deklarationen die an verschiedenen Stellen eines Programms gebraucht werden können. Sie bestehen aus Prozeduren und Funktionen. Im Gegensatz zu Funktionen geben Prozeduren keinen Rückgabewert zurück.

Alle Statements, die im Process-Statement zur Verfügung stehen, können innerhalb eines Unterprogramms benutzt werden. Allerdings dürfen WAIT-Anweisungen nur in Prozeduren vorkommen. Die Abarbeitung erfolgt auch hier sequentiell.

**Nomenklatur einer Unterprogramm-Definition:**

```
subprogram-specification is
    [declarations];
begin
    [statements];
end identifier;
```

### 5.1 Funktionen

Alle an eine Funktionen übergebenen Parameter sind innerhalb der Funktion konstant, d. h. die Funktion kann die Parameterwerte nicht ändern. Stattdessen liefern Funktionen einen Wert an die aufrufende Umgebung zurück. Der Aufruf einer Funktion verhält also sich wie eine 'expression'.

**Nomenklatur einer Function-Definition:**

```
function identifier interface-list return type-mark is
    [declarations]
begin
    [statements]
end identifier;
```

Mit *identifier interface-list* wird der Name der Funktion, sowie die Liste der zu übergebenden Parameter angegeben. Nach dem Wort RETURN steht der Datentyp des Wertes, den die Funktion an die Umgebung zurückgibt. Die Parameter können immer nur vom Modus IN sein und gehören zur Klasse der Signale oder der Konstanten.

#### Beispiel 30: Funktion zur Bestimmung des Minimums zweier Integer

Die Funktion vergleicht zwei Werte die vom Typ Integer sein müssen und gibt den niedrigeren der beiden an die Umgebung zurück. Würde a oder b bei einem Aufruf ein Parameter vom Typ 'Bit' zugewiesen, käme es zu einer Fehlermeldung. Die Rückgabe wird durch das RETURN-Statement im 'Function body' ausgeführt, das damit auch die Funktion abschliesst.

```
function Min (a, b : INTEGER) return INTEGER is
begin
    if a < b then;
        return a;
    else
        return b;
    end if;
end Min;
```

Mit dem Return-Statement wird die Ausführung eines Unterprogramms beendet. Danach springt das Programm wieder an die Stelle, wo Return aufgerufen wurde. Bei Funktionen geschieht dies mit

**return** value;

bei Prozeduren mit

**return;**

Parameter werden an ein Unterprogramm in der Reihenfolge ihrer Benennung an übergeben. So übergibt z. B. ein Funktionsaufruf “**if** Min (x, y) **then** ...;” den Parameter x an a und y an b. Diese Parameterübergabe heisst **”Positional Association”** und ist auch aus anderen Hochsprachen bekannt.

VHDL bietet noch eine Möglichkeit der Parameterübergabe, die sogenannte **”Named Association”**. Die Übergabe von Parametern erfolgt analog zu der Technik mit der bei Component Instantiation Anweisungen Signale mit Port-Signalen verbunden wurden<sup>3</sup>. Im obigen Beispiel würde der Aufruf dann wie folgt aussehen: “**if** Min (a => x, b => y) **then** ...;” Die Reihenfolge der Parameter im Code ist in diesem Fall nicht von Belang.

## 5.2 Resolution Funktionen

Wird ein Signal von mehreren Quellen (Prozessen) beschrieben, muß eine Resolution-Funktion definiert und benutzt werden. Diese Funktion wird aufgerufen, wenn an einem der Treiber ein Event (Signaländerung) anliegt. Sie bestimmt aus der Kombination der anliegenden Treiberwerte einen Wert, der dem Signal zugeordnet wird. Solch eine Situation ist z. B. bei einer Wired Or-Verknüpfung gegeben. Der Wert mit der höchsten Priorität wird dem Signal zugeordnet.

### Beispiel 31: Definition einer Resolution-Funktion

Der neu deklarierte Typ *fourlevel* beinhaltet den Wert 'X' für einen unbekannten logischen Signalzustand und den Wert 'Z' für einen hochohmigen Ausgang. Liegt ein Signal mit dem Wert '1' an, wird die Loop-Schleife sofort wieder verlassen und die Funktion Wired Or gibt den Wert '1' zurück. Ist dies nicht der Fall werden die anderen Werte überprüft. Hat ein Signal nicht den Wert '1', aber den Wert 'X', so wird dieser an solution übergeben. Nur bei den Werten 'Z' und '0' wird keine Zuweisung ausgeführt. Eine Wired Or-Verknüpfung könnte z. B. sein:

---

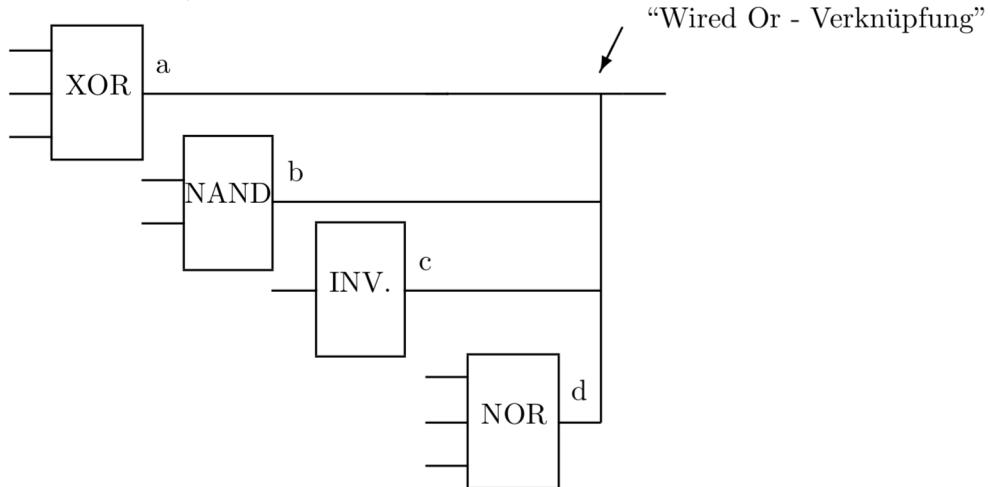
<sup>3</sup>Beispiel in Kapitel 1.2

```

type 4level is ('X', '0', '1', 'Z');
4level_vector is array (INTEGER range <>) of 4level;

function WiredOr (a : 4level_vector) return fourlevel is
    variable solution: fourlevel := '0';
begin
    for x in a'range loop
        if a(x) = '1' then
            solution := '1';
            exit;
        elsif a(x) = 'X' then
            solution := 'X';
        end if;
    end loop;
    return solution;
end WiredOr;

```



Ist  $a = 'X'$ ,  $b = 'Z'$ ,  $c = '0'$  und  $d = '1'$  so hat  $solution$  in der ersten Schleifenausführung den Wert  $'X'$  der erst in der vierten Schleife mit dem Wert  $'1'$  überschrieben wird und dann an das Hauptprogramm übergeben wird, das die Funktion `WiredOr` aufgerufen hat.

### 5.3 Prozeduren

Im Gegensatz zu den Funktionen können Prozeduren die an sie übergebenen Parameter verändern jedoch keinen Rückgabewert. Aufgerufen wird eine Prozedur mit dem Procedure-Call-Statement.

Erlaubte Modi für die Übergabeparameter sind: **in**, **out** oder **inout**. Ist der Modus nicht explizit angegeben setzt VHDL den Parameter automatisch auf den Modus **in**. Ist die Klasse der Parameter in einer Prozedur auch nicht definiert, werden sie beim Modus **in** als Konstanten deklariert, bei **out** und **inout** als Variablen. Grundsätzlich gelten bei Prozeduren die gleichen Syntaxregeln wie bei Funktionen. Insbesondere können Parameter ebenso wie auch bei Funktionen per **“Named Association”** übergeben werden.

Eine Besonderheit ist jedoch, dass skalare Parameter per **“Copy Back”** übergeben wer-

den. D. h. beim Aufruf der Prozedur werden die skalaren Parameter in eine lokale Kopie übertragen. Ist der Modus eines skalaren Parameters **out** oder **inout**, dann wird *erst* nach Beendigung der Prozedur, die Kopie zurück in den Orginalparameter kopiert. Dieses Verhalten ist insbesondere zu beachten, wenn eine Prozedur WAIT-Anweisungen enthält. Bei nicht skalaren Parametern ist das Verhalten bei der Parameterübergabe von der Implementierung des Simulators abhängig.

#### Nomenklatur einer Prozedur-Definition:

```
procedure identifier interface-list is
    [declarations]
begin
    [statements]
end identifier;
```

#### Beispiel 32:

Die folgende Prozedur führt abhängig vom übergebenen Parameter bestimmte Operationen auf den Signalen “op1” und “op2” aus und speichert das Ergebnis in “erg”. Zu beachten ist, dass eine Prozedur alle Objekte benutzen kann, die an der Stelle ihrer Deklaration sichtbar sind. In diesem Beispiel müssen also “op1”, “op2” und “erg” nicht als Parameter übergeben werden.

```
type alu_func is (add, sub, mult);
signal erg, op1, op2 : INTEGER := 2;
procedure operation (sel : alu_func) is
    variable test : INTEGER;
begin
    case sel is
        when add => test := op1 + op2;
        when sub => test := op1 - op2;
        when mult => test := op1 * op2;
    end case;
    if test < 100 then
        erg <= test;
    else
        report "Das Ergebnis ist zu gross";
    end if;
end operation;
...
p: process (... )
begin
    operation(add); -- "op1" und "op2" addieren.
    operation(sub); -- "op1" und "op2" subtrahieren.
    operation(mult); -- "op1" und "op2" multiplizieren.
end
```

## 6 Packages

Wenn Konstanten, Datentypen oder Unterprogramme in Entities oder Architecture-Bodies deklariert werden, so sind sie nur dort zugänglich (bei Entity-Deklarationen auch in der zugehörigen Architecture). Will man diese für eine Vielzahl von Entities oder Architecture-Bodies definieren, so kann man dafür die so genannten Packages verwenden. In einem Package kann man also z. B. häufig benötigte Typen, Konstanten und Unterprogramme zusammenfassen, um sie bei der Definition von verschiedenen Komponenten einzusetzen. Das Package besteht aus zwei Teilen, der 'Package declaration' (Schnittstelle nach außen) und dem 'Package body' (gleiche Bedeutung wie ein Architecture Body).

**Nomenklatur einer Package-Deklaration:**

```
package identifier is
    [declarations]
end identifier;
package body identifier is
    [declarations]
end identifier;
```

In der Package-Deklaration können die Schnittstellen von Unterprogrammen angegeben werden. Diese Unterprogramme sind dann nach außen hin sichtbar und können also z. B. in einer Architecture aufgerufen werden. Im zugehörigen Package-Body muss aber das komplette Unterprogramm definiert werden.

In dem Package Body können zusätzlich lokale Unterprogramme abgelegt werden, die nicht im Package deklariert sind, intern (im Body) aber benutzt werden. Wenn zum Beispiel in einem Package die Funktion *Addierer* deklariert und im Package Body noch die Funktion *Vektor->Integer* aufgeführt ist, so ist diese von außen weder sichtbar noch nutzbar. Die Funktion *Addierer* kann aber auf sie zurückgreifen und auch alle anderen im Package-Body vorkommenden Funktionen. Bedingung ist allerdings, dass die Funktion *Vektor->Integer* **vor den Funktionen deklariert** ist, die sie benutzen.

Damit der Inhalt eines Package von anderen VHDL Einheiten (Entities) genutzt werden kann, muss vor der jeweiligen Entity mit dem Befehl **use *identifier*.ALL** das Package "sichtbar" gemacht werden. Damit stehen alle Deklarationen vom Package der Entity zur Verfügung.

**Beispiel 33: Deklaration eines Packages**

Ein Inverter soll die Möglichkeit bieten 3 Zustände eines Signals zu unterscheiden und bei dem Wert 'Z' keine Invertierung vorzunehmen.

Im Package *Logic* sind der Datentyp *level3* und die Funktion *Invert* abgelegt. Damit die Entity *Inverter* sie nutzen kann, wird mit dem Befehl **use** *Logic.ALL* der Packageinhalt sichtbar gemacht. Dadurch kann jetzt die Funktion in der Architecture *inv*. aufgerufen werden, ohne dass eine Fehlermeldung beim Compilieren von *Inverter* eintritt. Das Package *Logic* selbst muß allerdings vorher compiliert worden sein.

```

package Logic is
    type level3 is ('0', '1', 'Z');
    function Invert (Input : in level3) return level3;
end Logic;

package body Logic is
    function Invert (Input : in level3) return level3 is
        begin
            case Input is
                when '0' => return '1';
                when '1' => return '0';
                when 'Z' => return 'Z';
            end case;
        end Invert;
    end Logic;

    use Logic. all;
    entity Inverter is
        port (E: in level3;
              A: out level3);
    end Inverter;

    architecture behavior of Inverter is
        begin
            process (E)
                begin
                    A <= Invert (E) after 10 ns;
                end process;
        end behavior;

```

**6.1 Spezielle Pakete**

Die Standard-Datentypen in VHDL, wie BOOLEAN oder BIT, die in Abschnitt 4 vorgestellt wurden, werden im so genannten *Standard-Package* definiert. Da dieses Paket nicht explizit über die *library*- und *use*-Anweisungen eingebunden werden muss, gehören die

dort definierten Datentypen gewissermaßen zum Sprachumfang von VDHL. Daneben gibt es noch einige weitere Pakete, die spezielle Datentypen und Funktionen zur Verfügung stellen, von denen folgend zwei vorgestellt werden.

### 6.1.1 *std-logic-Package*

Um eine hardwarenähere Beschreibung und Simulation von digitalen Komponenten zu ermöglichen wurde von der *IEEE* das ***std-logic-Package*** (*std\_logic\_1164*) standardisiert. Es spezifiziert eine 9-wertige Logik auf Basis eines Enumeration-Typs namens *std\_ulogic* und befindet sich in der *IEEE*-Library:

```
library IEEE;
use IEEE.std_logic_1164.ALL;
```

Die neun Elemente dieses Datentyps sind:

```
type std_ulogic is(
  'U',  -- nicht initialisiert
  'X',  -- unbekannt (stark)
  '0',  -- 0 (stark)
  '1',  -- 1 (stark)
  'Z',  -- hochohmig
  'W',  -- unbekannt (schwach)
  'L',  -- 0 (schwach)
  'H',  -- 1 (schwach)
  '-' , -- don't care);
```

Das *u* im Namen zeigt an, dass es sich dabei um einen *unresolved*-Typen handelt, d. h. ein Signal dieser Typs darf nur von einer Quelle beschrieben werden. Subtypen von *std\_ulogic* sind **X01**(mit den Zuständen 'X', '0', '1'), **X01Z** ('X', '0', '1', 'Z'), **UX01** ('U', 'X', '0', '1') und **UX01Z** ('U', 'X', '0', '1', 'Z').

Durch die Einführung der „schwachen“ Werte ('H', 'W', 'L') ist es möglich, Bausteine mit hohem Innenwiderstand von solchen mit niedrigem Innenwiderstand zu unterscheiden. Physikalisch bedeutet das, dass wenn zwei Signale die gleiche Leitung treiben, das starke das schwache Signal dominiert.

Analog zu *bit* bzw. *bit\_vector* ist in diesem Package auch der unbegrenzte Feld-Typ *std\_ulogic\_vector* definiert.

***std-logic*** Um zu ermöglichen, dass ein Signal von verschiedenen Prozessen getrieben werden kann, wurde das Package zusätzlich um einen *resolved*-Datentyp *std\_logic* erweitert, der *std\_ulogic* mit einer *Resolution*-Funktion kombiniert. Der entsprechende Vektortyp heisst *std\_logic\_vector*.

Für diese vier Typen sind – ähnlich wie für *bit* und *bit\_vector* die logischen Operatoren **AND**, **NAND**, **OR**, **NOR**, **XOR** und **NOT** definiert, wobei sie auf den Vektor-Versionen bitweise arbeiten.

**Konvertierungsfunktionen** Des Weiteren werden verschiedene Konvertierungsfunktionen zur Verfügung gestellt, um die einzelnen Typen des Pakets ineinander oder aber in *bit* bzw. *bit\_vector* umwandeln zu können. Die folgende Tabelle zeigt die vorhandenen Konvertierungsfunktionen: Die ersten drei Funktionen in Tabelle 1 haben zusätzlich einen

Funktion	Operandentyp	Ergebnistyp
To_bit	std_ulogic	bit
To_bit_vector	std_ulogic_vector	bit_vector
To_bit_vector	std_logic_vector	bit_vector
To_StdULogic	bit	std_ulogic
To_StdLogicVector	bit_vector	std_logic_vector
To_StdLogicVector	std_ulogic_vector	std_logic_vector
To_StdULogicVector	bit_vector	std_ulogic_vector
To_StdULogicVector	std_logic_vector	std_ulogic_vector
To_X01	...	...
To_X01Z	...	...
To_UX01	...	...

Tabelle 1: Konvertierungsfunktionen

weiteren Parameter, der spezifiziert, ob die Werte 'U', 'X', 'Z' und 'W' als '0' oder '1' interpretiert werden sollen.

Die letzten drei Funktionen der Tabelle wiederum sind so genannte *Strength-Stripper*, die die Wertebereiche der verschiedenen Logik-Typen aufeinander abbilden. Die folgende Tabelle zeigt, wie damit der Wertebereich von *std\_ulogic* in andere Typen überführt wird:

std_ulogic	bit	X01	X01Z	UX01
'U'	0/1	'X'	'X'	'U'
'X'	0/1	'X'	'X'	'X'
'0'	'0'	'0'	'0'	'0'
'1'	'1'	'1'	'1'	'1'
'Z'	0/1	'X'	'Z'	'X'
'W'	0/1	'X'	'X'	'X'
'L'	'0'	'0'	'0'	'0'
'H'	'1'	'1'	'1'	'1'
'_'	x	'X'	'X'	'X'

**Flankendetektion** Da der *std\_ulogic*-Typ aus neun verschiedenen Werten besteht, ist die Flankendetektion schwieriger als bei dem BIT-Typ. Darum stellt das *std\_logic\_1164*-Package zwei Funktionen zur Verfügung, die im Wesentlichen auf den Attributen *Si-*

*gnal'event* und *Signal'last\_value* und auf der Konvertierung des Signals nach *X01* beruhen. Dabei dient die Funktion *rising\_edge(param)* der Erkennung steigender Flanken und *falling\_edge(param)* entsprechend der Erkennung fallender Flanken, deren Parameter das beobachtete Signal und deren Rückgabewert ein BOOLEAN ist.

**Weitere Funktionen** Zusätzlich enthält das *std\_logic\_1164* noch eine weitere mehrfach überladene Funktion *Is\_X*, mit der ein *std\_(u)logic*-Wert darauf getestet werden kann, ob er 'X', 'U', 'W' oder 'Z' ist. Trifft das zu, liefert die Funktion *true* zurück, ansonsten *false*. Bei einem *std\_(u)logic\_vector* wiederum wird damit geprüft, ob einer der oben aufgeführten Werte enthalten ist.

```
function Is_X(s: std_(u)logic)
    return BOOLEAN;
function Is_X(s: std_(u)logic_vector)
    return BOOLEAN;
```

### 6.1.2 numeric\_std & numeric\_bit-Package

Um Zahlen in Form von Vektoren darstellen zu können, bietet der *IEEE*-Standard die synthetisierbaren Pakete *numeric\_bit* und *numeric\_std*.

```
library IEEE;
use IEEE.numeric_std.all;
use IEEE.numeric_bit.all;
```

Beide Pakete definieren zwei Arten von „Zahlenvektoren“, einen vorzeichenbehafteten namens *signed* und einen vorzeichenlosen namens *unsigned*.

```
type signed is array (NATURAL range<>) of std_logic (bzw. bit);
type unsigned is array (NATURAL range<>) of std_logic (bzw. bit);
```

Zusätzlich existieren entsprechend Operationen und Funktionen, die auf diese Typen angewandt werden können.

#### Anmerkung:

Da sich *numeric\_std* und *numeric\_bit* insgesamt sehr ähnlich sind, erfolgt im weiteren Verlauf eine Beschränkung auf die Beschreibung von *numeric\_std*.

Bei *numeric\_std* ist zu beachten, dass *signed* und *unsigned* „resolved“-Signale sind. Des Weiteren steht bei beiden das MSB links und die Darstellung des *signed*-Typs erfolgt im 2er-Komplement.

Die Operationen, die für *signed* und *unsigned* zur Verfügung stehen, zeigt Tabelle 2: Dabei zeigt die rechte Seite bezogen auf den vertikalen Doppelstrich jeweils alle (überladenen) Funktionen für den jeweiligen Operationsblock (begrenzt durch die horizontalen Linien).

Operator	Operation	Linker Operand	Rechter Operand	Ergebnis
<b>abs</b>	Absolutwert		signed	signed
<b>-</b>	Negation			
<b>+</b>	Addition	unsigned	unsigned	unsigned
<b>-</b>	Subtraktion	signed	signed	signed
<b>*</b>	Multiplikation	unsigned	natural	unsigned
<b>/</b>	Division	natural	unsigned	unsigned
<b>rem</b>	Rest	signed	integer	signed
<b>mod</b>	Modulo	integer	signed	signed
<b>=</b>	gleich	unsigned	unsigned	boolean
<b>/=</b>	ungleich	signed	signed	boolean
<b>&lt;</b>	kleiner als	unsigned	natural	boolean
<b>&lt;=</b>	kleiner gleich	natural	unsigned	boolean
<b>&gt;</b>	größer	signed	integer	boolean
<b>&gt;=</b>	größer gleich	integer	signed	boolean
<b>sll</b>	shift-left logical	unsigned	integer	unsigned
<b>srl</b>	shift-right logical	signed	integer	signed
<b>rol</b>	Links-Rotieren			
<b>ror</b>	Rechts-Rotieren			
<b>not</b>	Invertierung		unsigned signed	unsigned signed
<b>and</b>	logisches Und	unsigned	unsigned	unsigned
<b>or</b>	logisches Oder	signed	signed	signed
<b>nand</b>	negiertes Und			
<b>nor</b>	negiertes Oder			
<b>xor</b>	exklusive Oder			
<b>xnor</b>	negiertes XOR			

Tabelle 2: Operationen des *numeric\_std*-Packages

Bei arithmetischen und Vergleichs-Operatoren müssen die Operanden nicht die gleiche Länge besitzen. Bei Ungleichheit hängt die Ergebnislänge folgendermaßen mit der Länge der Operanden zusammen:

- Bei '+' und '-'
  - ⇒ Länge des längeren Operanden (bei Vektor / Vektor)
  - ⇒ Länge des Vektors (bei Integer bzw. Natural / Vektor)
- Multiplikation
  - ⇒ Summe der Operandenlängen (bei Vektor / Vektor)
  - ⇒ Zweifache Länge des Vektoroperanden (bei Integer bzw. Natural / Vektor)

- Division
  - ⇒ Länge des linken Operanden (bei Vektor / Vektor)
  - ⇒ Länge des Vektoroperanden (bei Integer bzw. Natural / Vektor)<sup>4</sup>
- Rest und Modulo
  - ⇒ Länge des rechten Operanden (bei Vektor / Vektor)
  - ⇒ Länge des Vektoroperanden (bei Integer bzw. Natural / Vektor)<sup>4</sup>

**Sonstige Funktionen** Neben den in Tabelle 2 aufgeführten Operatoren und Operationen existieren einige weitere Funktionen, die folgenden kurz beschrieben werden:

- Schiebefunktionen (*shift\_left()*, *shift\_right()*, *rotate\_left()*, *rotate\_right()*) mit zwei Argumenten  
Dabei ist das erste Argument der zu schiebende Vektor (*signed*, *unsigned*) und der zweite die Schiebeweite (NATURAL).
- *resize()*  
*resize()* liefert die Kopie eines Vektors mit veränderter Länge. Linksseitige Leerstellen werden mit dem Vorzeichen (*signed*) oder Nullen (*unsigned*) aufgefüllt. Beim Kürzen eines Vektors werden entsprechend viele MSBs abgeschnitten. Bei einer vorzeichenbehafteten Zahl, wird der Vektor um eine zusätzliche Stelle gekürzt. Dafür wird aber das Vorzeichen beibehalten.
- Konvertierungsfunktionen
  - *to\_integer()*  
Wandelt *signed* bzw. *unsigned* in INTEGER bzw. NATURAL um.
  - *to\_unsigned()* und *to\_signed()* mit jeweils zwei Argumenten  
Wandeln INTEGER- bzw. NATURAL-Zahl (erstes Argument) in *signed* bzw *unsigned* um, wobei das zweite Argument die Länge des Ergebnisvektors angibt.
  - *signed()*, *unsigned()*, *std\_logic\_vector()*  
Sind implizite Konvertierungsfunktionen, mit denen *std\_logic*-Vektoren in *signed* und *unsigned* umgewandelt werden können bzw. umgekehrt.
- *std\_match()* return BOOLEAN  
Vergleicht zwei Argumente (*unsigned*, *signed*, *std\_logic*, *std\_ulogic*, *std\_logic\_vector* und *std\_ulogic\_vector*) unter Berücksichtigung von „Don’t Cares“ und unter Vernachlässigung der Signalstärke (z. B. ’1’ oder ’H’). Dabei müssen die Argumente vom gleichen Typ sein. So sind z. B. „1001“ und „1–1“ unter Berücksichtigung der „Don’t Cares“ gleich und die Funktion liefert *true* zurück.
- *to\_01()* mit optionalem zweiten Parameter  
Wandelt die ’L’- und ’H’-Werte eines Vektors in ’0’ und ’1’ um. Enthält der Vektor andere Werte als ’0’, ’1’, ’L’ und ’H’ wird dieses Bit auf ’0’ gesetzt bzw. auf den Wert des optionalen zweiten Parameters. Diese Funktion ist allerdings nur für das *numeric\_std*-Package definiert.

---

<sup>4</sup>Ist der Ergebniswert zu groß, wird er abgeschnitten und es wird während der Simulation eine Warnung ausgegeben.

Des Weiteren enthält das *numeric\_bit*-Package noch die Funktion *rising\_edge()* und *falling\_edge()*, die bei *numeric\_std* wegen des *std\_ulogic*-Basistyps ja bereits vorhanden sind.

## 7 Attribute

Attribute liefern Informationen über Typen und Objekte. Diese Informationen können genutzt werden, um zum einen die Lesbarkeit des VHDL-Quellcodes zu verbessern und zum anderen parametrisierbare Modelle zu erzeugen. Der Designer hat hierbei sowohl die Möglichkeit auf eine Menge von vordefinierten Attributen zurückzugreifen, als auch eigene Attribute zu definieren. Ein Attribut eines Objektes bzw. Typs wird dabei durch den Ausdruck abgefragt:

**Nomenklatur einer Attributabfrage:**

<code>name'attribute_identifier</code>
--

“name” bezeichnet den Namen eines Typs, Subtyps oder Objektes und “**attribut\_identifier**” ist der Name des Attributs.

### 7.1 Value-Kind-Attribute

Mit den Value-Attributen können Informationen über Typen oder Felder abgefragt werden. Die gelieferten Werte sind dabei für das jeweilige VHDL-Objekt bzw. den VHDL-Typen konstant. Value-Kind-Attribute sind in drei Unterbereiche aufteilbar.

#### 7.1.1 Value Type

Value Type Attribute ermitteln die Grenzen eines Typs  $T^5$ . Die Grenzwerte eines Typs

`type acht is (0 to 7);`

sind z. B. 0 für den unteren und 7 für den oberen Grenze. Diese beiden Werte können nun mit Value Kind Attributen abgefragt werden. Hierfür gibt es vier vordefinierte Attribute:

**$T'\text{left}$ :** ermittelt den linken Grenzwert eines Typs bzw. Subtyps. Ist ein Typ beispielsweise folgendermaßen deklariert:

`type sieben is ( 6 downto 0)`

liefert  $T'\text{left}$  den Wert 6. Bei einer Typendeklaration mit **to** statt **downto** liefert es den Wert 0.

**$T'\text{right}$ :** Der rechte Grenzwert eines skalaren Typs bzw. Subtyps wird zurückgegeben.

**$T'\text{high}$ :** Dieses Attribut liefert die obere Grenze eines skalaren Typs oder Subtyps. Der Wert stimmt nicht automatisch mit dem Wert von  $T'\text{right}$  überein. Bei **downto** ändert sich der Wert von  $T'\text{right}$  der Wert von  $T'\text{high}$  aber nicht.

**$T'\text{low}$ :** Die untere Grenze eines Typs wird an das Programm zurückgegeben. Dabei ist es unerheblich, ob **to** bzw. **downto** in der Typendeklaration benutzt wurde.

---

<sup>5</sup>  $T$  steht für einen skalaren Typ

### 7.1.2 Value Array

Die Value Array Attribute liefern Informationen über die Index-Bereiche von Feldern bzw. Feld-Typen. Die Attribute können auf jede Art von Arrays  $A$ <sup>6</sup> angewendet werden, also auch auf mehrdimensionale Felder. Dann muss allerdings ein Parameter in runden Klammern angehängt werden, welcher die Dimension angibt, über die das Attribut entsprechende Informationen liefern soll.

$A'\text{length}$  gibt die Länge eines Arrays an.

$A'\text{ascending}$  liefert 'TRUE', falls der Indexbereich der entsprechenden Dimension aufsteigend ist, ansonsten 'FALSE'.

### 7.1.3 Value Block

Die Attribute **'structure'** und **'behavior'** haben den Wertetyp BOOLEAN.

Ist eine Architecture nur aus Component Instantiation Statements aufgebaut, also passiven Prozessen<sup>7</sup> oder dazu äquivalenten Statements, liefert das Attribut **'structure'** den Wert TRUE, ansonsten FALSE.

**'behavior'** liefert den Wert TRUE, wenn keine Component Instantiation Statements in einer Architecture vorhanden sind, ansonsten FALSE.

Bei Architekturen, die sowohl strukturell als auch sequentiell aufgebaut sind, liefern beide Attribute den Wert FALSE.

## 7.2 Funktionsattribute

Funktionsattribute geben Informationen über Typen, Arrays und Signale an den Anwender zurück, die von Aufruf zu Aufruf verschieden sein können.

### 7.2.1 Funktionstypen-Attribute

Mit dieser Art von Attributen können Informationen über einzelne Werte eines diskreten oder physikalischen Typs  $T$  ermittelt werden. VHDL ordnet hierbei jedem möglichen Wert des Typs eine Positionsnummer zu. Ist der Typ eine Enumeration Typ, dann hat der am weitesten links in der Typ-Deklaration stehende Wert die Positionsnummer 0, der rechts davon stehende die Nummer 1 u.s.w. Bei Integer-Typen ist die Positionsnummer einer Zahl gleich dem Wert der Zahl. Bei physical Typen gibt die Positionsnummer die Größe der jeweiligen Zahl in Bezug auf die Basiseinheiten an.

$T'\text{pos}(\text{value})$ : Es wird der Positionswert des Eingangswertes zurückgegeben.

$T'\text{val}(\text{position})$ : Der Positionswert wird wieder in einen Wert des entsprechenden Typs  $T$  umgewandelt.

$T'\text{succ}(\text{value})$ : Liefert den Wert der eine um 1 größere Positionsnummer als "value" besitzt.

---

<sup>6</sup>  $A$  steht für einen Array Typen

<sup>7</sup> Das sind Prozesse, die keine Signalzuweisungen enthalten.

*T'pred(value)*: Liefert den Wert der eine um 1 niedrigere Positionsnummer als "value" besitzt.

*T'leftof(value)*: Gibt den Wert an das Programm zurück, der in der Aufzählung links vom Eingangswert liegt.

*T'rightof(value)*: Gleiche Funktion wie *T'leftof* mit dem Unterschied, daß der rechts von "value" stehende Wert zurückgeliefert wird.

### 7.2.2 Funktionsarray-Attribute

Diese Attribute erfüllen die gleiche Funktion, die wir schon bei den Value Typen kennengelernt haben, mit dem Unterschied, daß sie auf Felder bzw. Feld-Typen angewendet werden. Es gibt vier Array Attribute:

*A'left(n)*: Gibt den linken Grenzwert des Indexwertebereichs der Dimension n zurück.

*A'right(n)*: Der rechte Indexgrenzwert der Dimension n wird an das Programm zurückgegeben.

*A'high(n)*: Rückgabe des oberen Indexgrenzwertes der n-ten Dimension.

*A'low(n)*: Rückgabe des unteren Indexgrenzwertes der n-ten Dimension.

Der Parameter "n" ist optional. Fehlt er, so wird das Attribut automatisch auf die erste Dimension angewendet.

#### Beispiel 34: Verwendung von Feld-Attributen

Die Zuweisungen für laenge1 und laenge2 im Prozess benutzen den Typ *dim2*. Da ein mehrdimensionales Array mehr als einen Wertebereich hat, besteht die Möglichkeit mit einer Angabe hinter **'length** einen Wertebereich auszusuchen. Es wird laenge1 die Länge des ersten Bereichs zugeordnet, laenge2 die des zweiten. Somit gibt laenge1 den Wert 3 zurück und laenge2 den Wert 2. Ist kein Bereich hinter **'length** angegeben, ermittelt VHDL die Länge des ersten Bereichs.

```

package array_pack is
    type state3 is ('X', 'Y', 'Z');
    type state2 is ('A', 'B');
    type dim1 is array (state3 'low to state3'high) of state3;
    type dim2 is array (state3'low to state3'high, state3'low
    to state3'high) of INTEGER;
end array_pack;

use WORK.array_pack.ALL;
 $\dots$ 
process (e)
    variable laenge1, laenge2 : INTEGER;
begin
    laenge1 := dim2'length(1);
    laenge2 := dim2'length(2);
end process;

```

### 7.2.3 Funktionssignal-Attribute

Mit der Verwendung von Funktionssignal-Attributen können Informationen über das Verhalten von Signalen  $S^8$  an das aufrufende Programm gegeben werden. So kann eine Meldung ausgegeben werden, wenn sich der Wert eines Signals ändert, wie viel Zeit seit dem letzten Event (Wertänderung auf dem Signal) verstrichen ist oder welchen Wert das Signal vor der letzten Änderung hatte. Diese Informationen werden mit folgenden Attributen abgefragt:

$S'\text{event}$ : Liefert den Wert TRUE vom Typ BOOLEAN an das aufrufende Programm zurück. Dies erfolgt, wenn in der aktuellen  $\Delta t$ -Periode  $^9$  an dem Signal mit dem Aufruf **'event** ein Event anliegt.

$S'\text{active}$ : Gibt den Wert TRUE vom Datentyp BOOLEAN zurück, wenn im aktuellen Simulationszyklus ein Wert auf das Signal  $S$  zugewiesen wurde. Dabei ist es unerheblich ob durch die Zuweisung das Signal seinen Wert geändert hat.

$S'\text{last\_event}$ : Gibt an das Programm einen Wert des Typs TIME zurück. Es liefert die Zeit, die seit dem letzten Event vergangen ist.

$S'\text{last\_value}$ : Mit dem Aufruf des Attributs  $S'\text{last\_value}$  wird der vorhergehende Wert des Signals vor dem letzten Event zurückgeliefert.

$S'\text{last\_active}$ : Gibt einen Wert vom Datentyp TIME zurück. Der Wert ist die Zeit, die seit der letzten Transaktion an dem Signal vergangen ist.

$S'\text{delayed}(t)$ : Der zurückgegebene Wert ist ein neues *Signal* vom gleichen Typ wie  $S$ . Der Signalverlauf des neuen Signals folgt hierbei  $S$  um  $t$  Zeiteinheiten verzögert.

---

<sup>8</sup> $S$  steht für einen Signal Typ

<sup>9</sup> $\Delta t$  = virtuelle Verzögerungszeit, die VHDL benutzt, um ein paralleles Arbeiten zu ermöglichen.

Der Unterschied zwischen einer entsprechenden Signalzuweisung mit **transport**-Verzögerung und diesem Attribut besteht darin, dass bei **S'delayed** kein neues Signal deklariert werden muß.

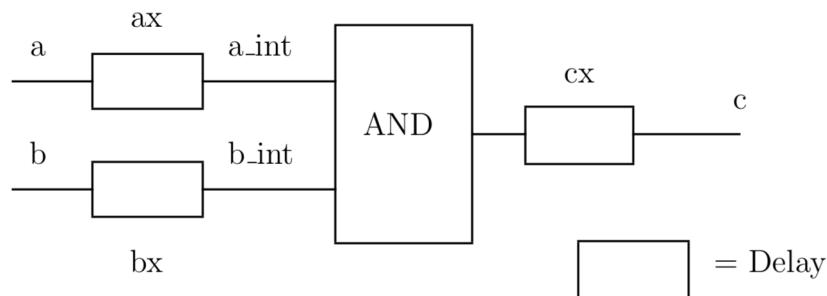
**S'stabled(t)**: Gibt ein implizites Signal vom Typ BOOLEAN zurück, dass immer dann TRUE ist, wenn seit t Zeiteinheiten kein Event am Signal S angelegen hat.

**S'quiet(t)**: Liefert ebenfalls ein implizites Signal vom Typ BOOLEAN zurück, dass TRUE ist, falls seit t Zeiteinheiten keine Transaktion an S angelegen hat.

**S'transaction**: Das Attribut liefert ein implizites Signal vom Typ Bit an das Programm zurück. Bei jeder Transaktion auf dem 'Signal' wird der Signalzustand ('0', '1') invertiert (er toggelt).

### Beispiel 35: Verwendung von signalliefernden Attributen

Die beiden aufgeführten Architekturen zeigen deutlich den Unterschied und die Vereinfachung mit der Benutzung des Attributs **S'delayed(t)** auf. Ist für (t) kein Wert angegeben, so wird eine Zeit von 0 angenommen.



```
use Logic ALL;
entity and3 is
  generic ( ax, bx, cx : time);
  port (a, b : in 3level; c : out 3level);
end and3;
```

```
architecture ohne_attribute of and3 is
  signal a-int, b-int : 3level;
begin
  a-int <= transport a after ax;
  b-int <= transport b after bx;
  c <= a-int AND b-int after cx;
end ohne_attribute;
```

```
architecture mit_attributen of and3 is
begin
  c <= a'delayed(ax) AND b'delayed(bx) after cx;
end mit_attributen;
```

## 8 Generate Anweisung

Die Generate-Anweisung ist ein sehr mächtiges Hilfsmittel bei der Erzeugung von parametrisierbaren Modellen. Es erlaubt dem Designer die Struktur seines Modells während der Elaborationsphase abhängig von Parametern zu verändern. Insgesamt stehen zwei Varianten zur Erzeugung von Strukturelementen zur Verfügung:

- Iterativ und
- Konditional.

Während die Iterative Generate-Anweisung eine beliebige Anzahl von parallelen Anweisungen mit Hilfe einer Schleife erzeugt, generiert die konditionale Generate-Anweisung einen parallelen Anweisungsblock falls der entsprechende konditionaler Ausdruck “TRUE” ist bzw. überspringt ihn, falls die Bedingung “FALSE” ergibt.

Die Generate-Anweisung gehört zur Klasse der “parallelen Anweisungen”, sie darf also z. B. im Anweisungsteil einer Archtitecture vorkommen. Da die Struktur des Modells nur einmal, nämlich in der Elaborationphase (während die Komponenten des Modells instantiiert und miteinander verdrahtet werden) am Anfang der Simulation erzeugt wird, werden die Generate-Anweisungen auch nur in dieser Phase ausgeführt. Während eine herkömmliche Schleife oder If-Anweisung in einem Prozess also beliebig häufig zur Simulationszeit durchlaufen werden kann, *erzeugen* die Generate-Anweisungen lediglich Prozesse, Komponenten oder auch parallele Signalzuweisungen. Die Generate-Anweisungen verändern also den Aufbau eines Modells (Anzahl von Komponenten, Prozessen und parallelen Signalzuweisungen).

### 8.1 Iterative Generate Anweisung

Mit der Iterativen Generate Anweisung werden parallele Anweisungen mit Hilfe einer Schleife in eine Schaltung eingefügt. Die Schleife wird während der Elaboration ausgeführt und erzeugt eine entsprechende Anzahl von Komponenten, Prozessen oder auch parallelen Signalzuweisungen.

**Die Syntax einer iterativen Generate Anweisung ist:**

```
generate_label :  
for identifier in discrete_range generate  
  { concurrent_statements }  
end generate [ generate_label ];
```

“generate\_label” wird zur Identifizierung der von der Generate-Anweisung erzeugten Elemente benötigt. “identifier” ist eine Schleifenvariable, die den durch “discrete-range” definierten Zahlenbereich durchläuft. Analog zu den Schleifenvariablen einer **for ... loop** Anweisung legt der Compiler selbständig eine Schleifenvariable mit dem Namen “identifier” an. “concurrent\_statements” schließlich sind die Parallelanweisungen die für jeden Durchlauf der **for**-Schleife erzeugt werden. Die Schleifenvariablen können innerhalb des “concurrent\_statements”-Bereiches verwendet werden.

**Beispiel 36:** Ein Register-Baustein mit einer variablen Anzahl von Bits

“dff” ist ein D-Flipflop. Durch die Generate-Schleife werden “breite” Instanzen des “dff” Bausteins erzeugt und mit Ein- bzw. Ausgabevektoren von “register” verdrahtet.

```
entity register is
    generic (breite : INTEGER);
    port (gate : BIT;
          i : in BIT_VECTOR(0 to breite - 1);
          q : inout BIT_VECTOR(0 to breite - 1));
end register;
architecture struct of register is
begin
    gen: for j in 0 to breite-1 generate
        dflipflop: entity dff
            port map(d.in => i(j), gate => gate,
                      d.out => q(j));
    end generate;
end struct;
```

**Beispiel 37:**

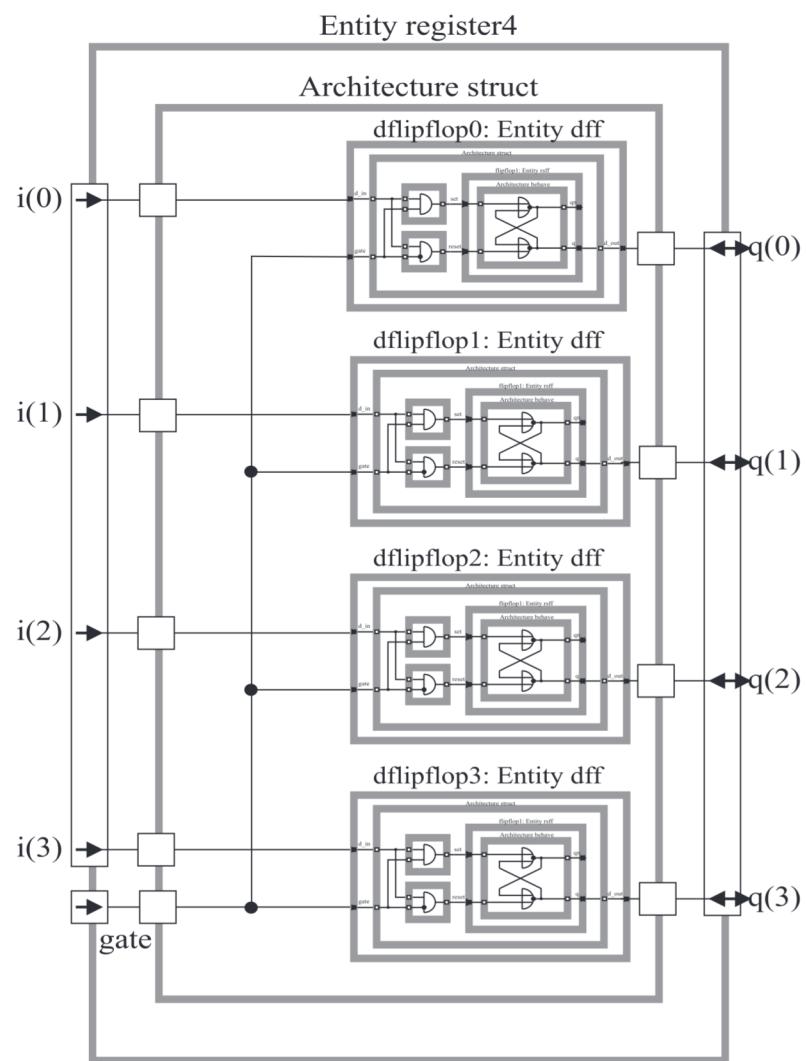
Mit “register” kann z. B. wie folgt ein 4 Bit breites Latch aufgebaut werden.

```
architecture struct of register4 is
    signal a, b : BIT_VECTOR(0 to 3);
    signal clk : BIT;
begin
    latch: entity register
        generic map(4) -- Achtung kein Semikolon
        port map(gate => clk, i => a, q => b);
end struct;
```

Innerhalb des “concurrent\_statements” Bereich einer Generate Deklaration können auch Parallelzuweisungen oder ganze Prozesse angegeben werden.

**Beispiel 38:** Generate-Anweisung mit eingebetteten Parallelzuweisungen

Das folgende Modell addiert die beiden binären Zahlenwerte “a” und “b” und speichert das Ergebnis in Feld “erg” ab. Achtung: Das Feld “ov” nimmt den Überlauf zwischen den Bit-Stellen auf und muss um eine Bitstelle größer sein als die anderen Felder. Die eigentliche Addition wird mit Hilfe von parallelen Signalzuweisungen implementiert.



```

architecture struct of unbekannt is
    signal a, b, erg : BIT_VECTOR(0 to 31);
    signal ov : BIT_VECTOR(0 to 32) := (others => '0'); – Initialisierung
    von ov
begin
    gen1: for i in 0 to 31 generate
        erg(i) <= (a(i) XOR b(i)) XOR ov(i) after 10 ns;
        ov(i+1) <= (a(i) AND b(i)) OR
            (a(i) AND ov(i)) OR
            (b(i) AND ov(i)) after 10 ns;
    end generate;
end struct;

```

### Beispiel 39: Generate-Anweisung mit eingebetteten Prozessen

Das Modell addiert die beiden binären Zahlenwerte “a” und “b” und speichert das Ergebnis in Feld “erg” ab. Achtung: Das Feld “ov” nimmt den Überlauf zwischen den Bit-Stellen auf und muss um eine Bitstelle größer sein als die anderen Felder. Diesmal wird die Addition durch einen Prozess ausgeführt.

```

architecture struct of unbekannt is
    signal a, b, erg : BIT_VECTOR(0 to 31);
    signal ov : BIT_VECTOR(0 to 32) := (others => '0'); – Initialisierung
    von ov
begin
    gen1: for i in 0 to 31 generate
        p: process (a, b)
        begin
            erg(i) <= (a(i) XOR b(i)) XOR ov(i) after 10 ns;
            ov(i+1) <= (a(i) AND b(i)) OR
                (a(i) AND ov(i)) OR
                (b(i) AND ov(i)) after 10 ns;
        end process;
    end generate;
end struct;

```

## 8.2 Konditionale Generate Anweisung

In den vorhergehenden Beispielen war die Struktur der in jedem Iterationsschritt erzeugten Komponenten bzw. Prozesse gleich. Manchmal ist es jedoch erforderlich an bestimmten Stellen in einer ansonsten regulären Schaltung andere Komponenten oder auch Prozesse zu installieren. Diese Sonderfälle können auf einfache Weise mit einer konditionalen Generate Anweisung berücksichtigt werden.

### Nomenklatur einer konditionalen Generate-Anweisung:

```
generate_label :  
if boolean_expression generate  
  { concurrent_statements }  
end generate [ generate_label ];
```

Die Anweisung erzeugt in Abhängigkeit von dem booleschen Ausdruck “boolean\_expression” die im “concurrent\_statements” Bereich angegebenen Anweisungen. “generate\_label” wird wie schon bei der iterativen Generate Anweisung dazu benötigt, die durch die Anweisung erzeugten Strukturen zu identifizieren.

Achtung: “If-Then-Else”-Ketten sind mit der konditionalen Generate Anweisung nicht direkt möglich.

### Beispiel 40: Anwendung von konditionalen Generate-Anweisungen

Das Modell addiert die beiden binären Zahlenwerte “a” und “b” und speichert das Ergebnis in Feld “erg” ab. Diesmal ist das Feld “ov” genauso groß wie die Felder “a”, “b” und “erg”.

```
architecture struct of unbekannt is
  signal a, b, erg: BIT_VECTOR(0 to 31);
  signal ov : BIT_VECTOR(0 to 31) := (others => '0'); – Initialisierung
  von ov
begin
  it: for i in 0 to 31 generate
    cond_erg0: if i = 0 generate
      erg(i) <= a(i) XOR b(i) after 10 ns;
    end generate cond_erg0;
    cond_erg: if i / = 0 generate
      erg(i) <= (a(i) XOR b(i)) XOR ov(i) after 10 ns;
    end generate cond_erg;
    cond_ov31: if i / = 31 generate
      ov(i+1) <= (a(i) AND b(i)) OR
                  (a(i) AND ov(i)) OR
                  (b(i) AND ov(i)) after 10 ns;
    end generate cond_ov31;
  end generate it;
end struct;
```

## 9 Synthese

Neben der Möglichkeit digitale Schaltungen mit VHDL auf unterschiedlichen Abstraktebenen zu modellieren und durch Simulation zu überprüfen, ist basierend auf der VHDL-Beschreibung auch eine Schaltungssynthese mit Hilfe eines so genannten Synthesetools durchführbar, bei der die abstrakte VHDL-Beschreibung auf reale Hardware abgebildet wird.

Synthesetools sind spezielle Programme, die aus einer VHDL-Beschreibung eine für eine bestimmte Zieltechnologie (FPGA, CPLD, ASIC, ...) optimierte digitale Schaltung generieren. Das Ergebnis eines Syntheselaufes ist eine Netzliste, die – nach ein paar weiteren hier nicht näher betrachteten Verarbeitungsschritten – in einen programmierbaren Baustein (z.B. FPGA) heruntergeladen werden kann oder auch an einen ASIC-Hersteller gesendet wird, der daraus wiederum eine IC produziert.

Synthesetools werden von verschiedenen Herstellern wie Synplicity angeboten. Zwar ist bei diesen Tools der allgemeine Syntheseprozess sehr ähnlich, es besteht allerdings ein Unterschied im Befehlssatz und in der Art, wie Synthese-Constraints spezifiziert werden. Entscheidender ist aber, dass zum einen von den Synthese-Tools nicht der gesamte VHDL-Wortschatz unterstützen wird und sich sie zum anderen im unterstützten Wortschatz unterscheiden. Daraus resultiert, dass heutige Synthesetools nicht in der Lage sind jede beliebige VHDL-Beschreibung in eine Schaltung umzuwandeln. Das ist aufgrund der Komplexität von VHDL aber auch nicht effektiv möglich. Vielmehr können in einer „synthesierbaren“ Beschreibung nur ganz bestimmte Konstrukte und Datentypen verwendet werden. Offensichtlich nicht synthetisierbar sind z.B. Datei-Operationen, Report-Meldungen oder Ähnliches.

Folgend wird deshalb kurz auf synthetisierbare und nicht synthetisierbare VHDL-Konstrukte bzw. auf die besonderen Eigenschaften von VHDL eingegangen. Anschließend werden die Modellierung einiger grundlegender digitaler Komponenten anhand von Beispielen aufgezeigt:

- Da **Parallelanweisungen** (Concurrent Statements) parallel ausgeführt werden, ist die Reihenfolge im VHDL-Code nicht relevant
- **Signale und Variablen** sind synthetisierbar. Dabei werden Signale direkt in Hardware umgesetzt, falls sie nicht bei der Optimierung entfernt werden. Variablen hingegen dienen oft nur dazu, Zwischenergebnisse zu speichern
- **Synthetisierbare Hardware**

Neben einfachen logischen Verknüpfungen (Gattern) wie UND oder ODER mit zwei oder mehr Eingängen sind natürlich auch Register basierend auf flankengesteuerten Flipflops synthetisierbar, die entweder auf die positive oder die negative Flanke sensitiv sein können. Dabei können sowohl synchrone als auch asynchrone Resets, Presets und Clear-Eingänge realisiert werden. Daneben ist auch die Synthese einfacher peigelgesteuerter Flipflops möglich. Mit diesen Grundkonstrukten lässt sich neben Signalzuweisungen zum einen einfache kombinatorische Logik mit Hilfe des WITH-SELECT-Statements bzw. des WHEN-Statements in der parallelen Betriebsebenen als auch mit Hilfe des IF- und CASE-SELECT-Statements innerhalb von Prozessen realisieren. Zum anderen ist damit die Modellierung sequentieller Logik wie taktflan-

kengesteuerten und zustandsgesteuerten Prozessen als auch von Zustandsautomaten möglich.

- **Synthetisierbare Datentypen**

Um synthetisierbare Modelle erzeugen zu können, ist die Beschränkung auf folgende Datentypen notwendig:

- Enumeration Typen, was auch BOOLEAN, BIT und CHARACTER beinhaltet
- INTEGER
- Eindimensionale Felder, wie die vordefinierten Typen BIT\_VECTOR und STRING
- STD\_LOGIC, STD\_LOGIC\_VECTOR, STD\_ULOGIC, STD\_ULOGIC\_VECTOR des *std\_logic*-Packages
- UNSIGNED und SIGNED des *numeric\_bit*- und des *numeric\_std*-Packages

- **Schleifen und Slices**

Das Synthesetool erzeugt statische (unveränderbare) Hardware aus einem VHDL-Modell. Daraus folgt, dass

- Schleifen bzw. Slices entrollt werden und
- der Indexbereich von Schleifen bzw. Slices besonderen Restriktionen unterliegt, d. h. er muss zur Compile- bzw. Elaborationszeit bestimmbar sein.

- **Nicht synthetisierbare Konstrukte** sind z. B. Dateien, ASSERT- und REPORT-Anweisungen, Zeitverzögerungen in Signalzuweisungen usw.. Außerdem werden „don’t cares“ typischerweise nicht unterstützt. Schwer bzw. nicht zu synthetisieren sind aber auch weniger komplex erscheinende Operationen wie Multiplikation oder Division. Das liegt daran, dass es für solche Operationen eine Vielzahl von Realisierungsmöglichkeiten gibt, die sich sehr stark in ihren Eigenschaften unterscheiden (Genauigkeit, Geschwindigkeit, Platzbedarf, ...), so dass das Synthesetool nur schwer eine geeignete Lösung selbstständig wählen kann.

Letztlich muss der Designer bei der Erstellung der Schaltung – insbesondere wenn bestimmte Randbedingungen (Geschwindigkeit, Platzbedarf, ...) erfüllt werden müssen – vorgegebene Beschreibungsschemata und Operationen verwenden. Diese Schemata variieren abhängig vom verwendeten Synthesetool etwas, orientieren sich aber immer an in Hardware realisierbaren Schaltungsprimitiven.

Schaltungsprimitive sind elementare, digitale Baugruppen und lassen sich in zwei Klassen unterteilen:

- **Kombinatorische Logik**

Bei diesen Schaltungsprimitiven sind die Ausgangswerte nur von den aktuell am Eingang angelegten Werten abhängig. Diese Primitive sind zustandslos, enthalten also keine speichernden Elemente.

- **Sequentielle Logik**

Sequentielle Logik Bausteine hingegen benutzen Flipflops um Informationen aus der Vergangenheit zu speichern. Ihre Ausgangswerte sind somit sowohl abhängig

von den aktuellen wie auch den in der Vergangenheit an den Baustein angelegten Eingangswerten.

In den folgenden Kapitel werden einige zentrale Primitive und deren Beschreibung in VHDL vorgestellt.

### 9.1 Modellierung kombinatorischer Logik in VHDL

Kombinatorische Logik kann auf unterschiedliche Arten modelliert werden. Am einfachsten erfolgt die Beschreibung mit Hilfe von Parallelzuweisungen.

#### Beispiel 41:

Kodierung von einfachen logischen Verknüpfungen mit Hilfe von Parallelzuweisungen.

```
architecture komb of test is
  signal a, b, c, d, e : BIT;
begin
  d <= a AND (b OR c);
  e <= a OR b OR c;
end komb;
```

Ebenso können aber auch Prozesse zur Modellierung kombinatorischer Logik verwendet werden. Dabei ist jedoch zu beachten, dass alle Signale, von denen innerhalb des Prozesses gelesen wird, in die Sensitivitätsliste des Prozesses aufgenommen werden müssen. Zusätzlich können zur Berechnung der entsprechenden Signalwerte auch Variablen benutzt werden.

#### Beispiel 42:

Die beiden Prozesse sind äquivalent zu den Parallelzuweisungen aus dem vorhergehenden Beispiel.

```
architecture komb of test is
  signal a, b, c, d, e : BIT;
begin
  P1: process (a, b, c)
    variable var : BIT;
  begin
    var := b OR c;
    d <= a AND var;
  end process;
  P2: process (a, b, c)
  begin
    e <= a OR b OR c;
  end process;
end komb;
```

### 9.1.1 Multiplexer

Ein Multiplexer hat zwei oder mehrere Dateneingänge, einen Selektiereingang und einen Datenausgang. In Abhängigkeit von dem Selektiersignal wird ein Datenpfad von einem der Eingänge zum Ausgang geschaltet. Die Wahrheitstabelle eines 2 zu 1 Multiplexers ist wie folgt:

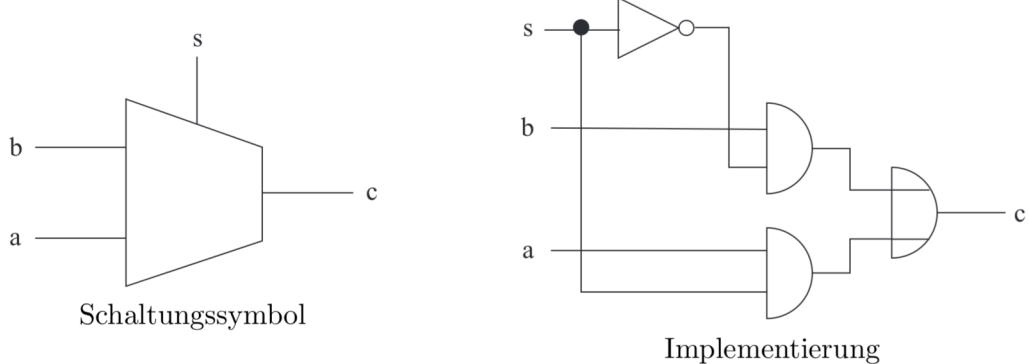
s	a	b	c
0	0	-	0
0	1	-	1
1	-	0	0
1	-	1	1

“s” ist der Selektiereingang, “a” und “b” die Dateneingänge und “c” der Datenausgang des Multiplexers.

Synthesetools können im Allgemeinen nur schwer Multiplexer in einer Modellbeschreibung erkennen und diese effektiv auf die Zieltechnologie abbilden. Im Normalfall sollte man also entsprechende Makro-Zellen aus der verwendeten Technologiebibliothek benutzen. Trotzdem ist es in manchen Fällen sinnvoll die Multiplexerfunktionalität explizit zu Programmieren um dem Synthesetool weitergehende Optimierungen mit der den Multiplexer umgebenden Schaltung zu ermöglichen.

#### Beispiel 43:

Schaltbild und Realisierung eines 2 zu 1 Multiplexers.



Die folgenden Beispiele zeigen vier Arten einen 4 zu 1 Multiplexer in VHDL zu beschreiben. Die Entity-Deklaration der Schaltung sei hierbei jedesmal:

```
entity mux4 is
  port (sel : in std_ulogic_vector(1 downto 0);
        a, b, c, d : in std_ulogic;
        y : out std_ulogic);
end mux4;
```

**Beispiel 44:**

Implementierung eines Multiplexers mit Hilfe einer bedingten parallelen Signalzuweisung.

```
architecture conditional of mux4 is
begin
    y <= a when sel = "00" else
        b when sel = "01" else
        c when sel = "10" else
        d;
end conditional;
```

**Beispiel 45:**

Implementierung eines Multiplexers mit Hilfe einer selektierten parallelen Signalzuweisung.

```
architecture selected of mux4 is
begin
    with sel select
        y <= a when "00",
        b when "01",
        c when "10",
        d when others;
end selected;
```

**Beispiel 46:**

Multiplexerimplementierung mit Hilfe von verschachtelten If-Then-Else-Anweisungen.

```
architecture ifthenelse of mux4 is
begin
    process (sel, a, b, c, d)
        if sel(1)='0' then
            if sel(0)='0' then
                y <= a;
            else
                y <= b;
            end if;
        else
            if sel(0)='0' then
                y <= c;
            else
                y <= d;
            end if;
        end if;
    end process;
end ifthenelse;
```

**Beispiel 47:**

Multiplexerimplementierung mit Hilfe einer Case-Anweisung.

```
architecture casestatement of mux4 is
begin
    process (sel, a, b, c, d)
        case sel is
            when "00" => y <= a;
            when "01" => y <= b;
            when "10" => y <= c;
            when "11" => y <= d;
        end case;
    end process;
end casestatement;
```

**9.1.2 Enkoder**

Ein Enkoder hat  $m$  numerierte Eingangssignale und  $n$  Ausgangssignale, wobei die Ungleichung  $2^n \geq m$  gilt. Der Baustein wandelt die Nummer eines aktivierten Eingangs in eine entsprechend kodierte Binärzahl um. Um die Umwandlung eindeutig durchführen zu können darf hierbei stets nur *ein* Eingang aktiviert sein.

**Beispiel 48:**

Die Wahrheitstabelle eines 4 zu 2 Enkoders. Ein Eingang ist hierbei aktiviert, falls er den logischen Pegel '1' trägt.

Eingänge				Ausgänge	
i3	i2	i1	i0	y1	y0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1
sonst				X	X

Zur Implementierung eines Enkoders in VHDL können dieselben Techniken angewendet werden wie sie schon im vorhergehenden Kapitel vorgestellt worden sind. Zu Beachten ist, dass die Case-Anweisung einen Default-Zuweisung (hier "00") enthalten muss, damit keine speichernden Elemente vom Synthesetool erzeugt werden.

**Beispiel 49:**

Implementierung eines 4 zu 2 Enkoders mit Hilfe einer **case**-Anweisung.

```
entity enc4_2 is
    port (i : in BIT_VECTOR(3 downto 0);
          y : out BIT_VECTOR(1 downto 0));
end enc4_2;
architecture casestatement of enc4_2 is
begin
    process (i)
        case i is
            when "0001" => y <= "00";
            when "0010" => y <= "01";
            when "0100" => y <= "10";
            when "1000" => y <= "11";
            when others => y <= "00";
        end case;
    end process;
end casestatement;
```

**Beispiel 50:**

Implementierung des Enkoders mit Hilfe einer selektierten parallelen Signalzuweisung.

```
architecture selected of enc4_2 is
begin
    with i select
        y <= "00" when "0001",
        "01" when "0010",
        "10" when "0100",
        "11" when "0001";
        "00" when others;
end selected;
```

Soll der Enkoder eine sehr große Wortbreite besitzen, dann bietet sich eine Implementierung mit Hilfe eines Prozesses an.

**Beispiel 51:**

Implementierung eines 128-7 Enkoders mit Hilfe eines Prozesses. „std\_logic“ ist ein spezieller Enumeration-Typ mit dem neben '0' und '1' noch weitere Werte wie z.B. unbekannt ('X') oder nicht initialisiert ('U') kodiert werden können. „std\_logic“ ist zusammen mit weiteren Funktionen und Operatoren im Package „std\_logic\_1164“ der Library “IEEE” definiert. Der Typ “unsigned“ ist ein Feld von „std\_logic“-Werten. Er ist zusammen mit einigen weiteren Typen sowie arithmetischen und logischen Funktionen in dem “numeric\_std” Package der Library “IEEE” enthalten. Achtung: die Initialisierung des Ausgabewertes mit 'X' eröffnet einem Synthesetools Möglichkeiten zur Optimierung der Schaltungslogik.

```

library IEEE;
use IEEE.std_logic_1164.ALL; use IEEE.numeric_std.ALL;
entity enc128_7 is
    port (i : in UNSIGNED(127 downto 0);
          y : out UNSIGNED(6 downto 0));
end enc128_7;
architecture proc of enc128_7 is
begin
    process (i)
        variable test : UNSIGNED(127 downto 0);
    begin
        test := (0 => '1', others => '0');
        y <= (others => 'X');
        for n in 127 downto 0 loop
            if test = i then
                -- "to_unsigned(n, 7)" wandelt die Zahl "n"
                -- in einen 7-stelligen unsigend-Vektor um
                y <= to_unsigned(n, 7);
                exit;
            end if;
            -- "shift_left(test, 1)" schiebt den Vektor
            -- test um eine Stelle nach links
            test := shift_left(test, 1);
        end loop;
    end process;
end proc;

```

### 9.1.3 Prioritäts Enkoder

Im Gegensatz zum normalen Enkoder akzeptiert der Prioritäts-Enkoder auch mehrere aktivierte Eingänge. Am Ausgang wird dann die Position des höherwertigsten aktivierten Eingangs angezeigt.

#### Beispiel 52:

Die Wahrheitstabelle eines 4 zu 2 Prioritäts Enkoders. Ein Eingang ist hierbei „aktiviert“, falls er den logischen Pegel '1' trägt.

Eingänge				Ausgänge	
i3	i2	i1	i0	y1	y0
0	0	0	0	X	X
0	0	0	1	0	0
0	0	1	?	0	1
0	1	?	?	1	0
1	?	?	?	1	1
sonst				X	X

**Beispiel 53:**

Implementierung eines 4-2 Prioritäts-Enkoders mit Hilfe einer If-Then-Else-Kette. Achtung: Die Zuweisung von “X...” in der letzten If-Then-Else-Verzweigung ermöglicht dem Synthesetool die Optimierung der Schaltung.

```
library IEEE;
use IEEE.std_logic_1164.ALL; use IEEE.numeric_std.ALL;
entity penc4_2 is
    port (i : in UNSIGNED(3 downto 0);
          y : out UNSIGNED(1 downto 0));
end penc4_2;
architecture proc of penc4_2 is
begin
    process (i)
    begin
        if i(3) = '1' then y <= "11"
        elsif i(2) = '1' then y <= "10"
        elsif i(1) = '1' then y <= "01"
        elsif i(0) = '1' then y <= "00"
        else y <= "XX";
        end if;
    end process;
end proc;
```

Zur Beschreibung von Prioritäts-Enkodern mit einer großen Wortbreite kann man am einfachsten einen Prozess mit einer eingebetteten For-Schleife verwenden.

**Beispiel 54:**

Implementierung eines 128 zu 7 Prioritäts-Enkoders mit Hilfe eines Prozesses. Der Typ “unsigned” ist ein Feld von “std\_logic”-Werten. Er ist zusammen mit einigen weiteren Typen sowie arithmetischen und logischen Funktionen im Package “numeric\_std” definiert (siehe Kapitel 9.1.2).

```
library IEEE;
use IEEE.std_logic_1164.ALL; use IEEE.numeric_std.ALL;
entity penc128_7 is
    port (i : in UNSIGNED(127 downto 0);
          y : out UNSIGNED(6 downto 0));
end penc128_7;
architecture proc of penc128_7 is
begin
    process (i)
    begin
        y <= (others => 'X'); -- Wichtig für die Optimierung.
        for n in 127 downto 0 loop
            if i(n) = '1' then
                -- "to_unsigned(n, 7)" wandelt die Zahl "n"
                -- in einen 7-stelligen unsigned-Vektor um
                y <= to_unsigned(n, 7);
                exit;
            end if;
        end loop;
    end process;
end proc;
```

**9.1.4 Dekoder**

Dekoder aktivieren in Abhängigkeit einer Eingangszahl “n” eine der  $m \leq 2^n$  Ausgänge des Bausteins. Sie sind also praktisch die inverse Schaltung zu einem Enkoder.

**Beispiel 55:**

Die Wahrheitstabelle eines 2 zu 4 Dekoders.

Eingänge		Ausgänge			
i3	i2	y3	y2	y1	y0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Die Implementierung eines Dekoders kann analog zu der eines Enkoders durchgeführt werden. Beispielhaft sei hier die Realisierung mit Hilfe einer For-Schleife vorgestellt.

**Beispiel 56:**

Implementierung eines Enkoders mit Hilfe einer Schleife.

```
library IEEE;
use IEEE.std_logic_1164.ALL; use IEEE.numeric_std.ALL;
entity dec7_128 is
    port (i : in UNSIGNED(6 downto 0);
          y : out UNSIGNED(127 downto 0));
end dec7_128;
architecture proc of dec7_128 is
begin
    process (i)
    begin
        for n in 127 downto 0 loop
            -- "to_unsigned(n, 7)" wandelt die Zahl "n"
            -- in einen 7-stelligen unsigend-Vektor um
            if i = to_unsigned(n, 7) then
                y(n) <= '1';
            else
                y(n) <= '0';
            end if;
        end loop;
    end process;
end proc;
```

Benötigt man ein zusätzlichen Kontrolleingang, über den der Ausgang des Bausteins wahlweise aktiviert bzw. deaktiviert werden kann, so kann das am einfachsten mit einer If-Anweisung realisiert werden.

**Beispiel 57:**

Ein Dekoder mit Enable-Eingang. Ist das Enable-Signal '0', dann werden alle Ausgangleitungen auf '0' gesetzt, anderenfalls wird der Eingangswert wie gewohnt dekodiert.

```
library IEEE;
use IEEE.std_logic_1164.ALL; use IEEE.numeric_std.ALL;
entity dec7_128_enable is
    port (enable : std_logic;
          i : in UNSIGNED(6 downto 0);
          y : out UNSIGNED(127 downto 0));
end dec7_128_enable;
architecture proc of dec7_128_enable is
begin
    process (i, enable)
    begin
        if enable = '0' then y <= (others => '0');
        else
            for n in 127 downto 0 loop
                if i = to_unsigned(n, 7) then
                    y(n) <= '1';
                else
                    y(n) <= '0';
                end if;
            end loop;
        end if;
    end process;
end proc;
```

### 9.1.5 Komparatoren

Komparatoren vergleichen zwei Zahlen miteinander. Wie die einzelnen Bitstellen hierbei interpretiert werden, hängt von der verwendeten Zahlendarstellung ab (konegativ, Betrag-Offset ...).

#### Beispiel 58:

Komparator zum Vergleich von 2 vier Bit breiten Zahlen. Der Ausgang "y" ist '1', wenn beide Zahlen gleich sind.

```
entity comparator is
    port (a, b : in BIT_VECTOR(3 downto 0);
          y : out BIT);
end comparator;
architecture proc of comparator is
begin
    process (a, b)
    begin
        y <= '1';
        for n in 3 downto 0 loop
            if a(n) /= b(n) then
                y <= '0';
                exit;
            end if;
        end loop;
    end process;
end proc;
```

Falls das Synthesetool es unterstützt, so kann auch der Vergleichsoperator zum Vergleich von Bit-Vektoren verwendet werden.

#### Beispiel 59:

Diese VHDL-Beschreibung nutzt den Operator "==" zum Vergleichen von zwei "unsigned"-Vektoren.

```
architecture operator of comparator is
begin
    process (a, b)
    begin
        if a = b then
            y <= '1';
        else
            y <= '0';
        end if;
    end process;
end operator;
```

## 9.2 Modellierung sequentieller Logik

Sequentielle Schaltungen bestehen aus Gruppen von kombinatorischer Logik, die mit speichernden Elementen (Flipflops) miteinander verbunden sind. Die am häufigsten eingesetzten speichernden Elemente sind

- transparente, pegelgesteuerte Daten-Flipflops (D-Flipflops) und
- flankengesteuerte D-Flipflops.

Neben diesen Typen existieren noch eine Vielzahl von anderen Flipflop-Arten (Toggle-Flipflops, JK-Flipflops, RS-Flipflops ...).

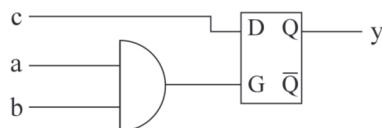
Das Synthesetool kann auf verschiedenen Arten dazu gezwungen werden Flipflops in die Schaltung einzufügen. Bei der Synthese wird hierbei immer dann ein Flipflop zur Speicherung eines bestimmten Signalwertes erzeugt, wenn es

- a) einen Prozess gibt, der schreibend auf das Signal zugreift und
- b) bei mindestens einem Lauf dieses Prozesses dem Signal *kein* Wert zugewiesen wird.

### Beispiel 60:

Dieser Prozess erzeugt bei der Synthese ein transparentes pegelgesteuertes D-Flipflop für das Signal "y", da nur falls die Signale "a" und "b" beide '1' sind, "y" ein neuer Wert zugeordnet wird. Sind "a" und "b" hingegen nicht '1', so behält "y" den zuletzt zugewiesenen Wert, d.h. den Wert den das Signal "c" hatte, als zuletzt "a" und "b" beide '1' waren.

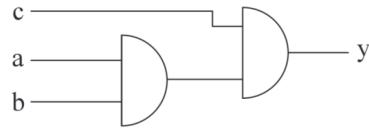
```
architecture behave of test1 is
    signal a, b, c, y : BIT;
begin
    process (a, b, c)
    begin
        if a AND b = '1' then
            y <= c;
        end if;
    end process;
end operator;
```



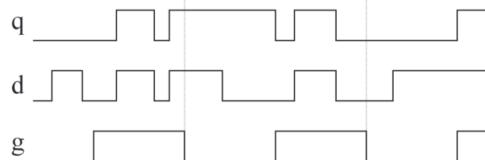
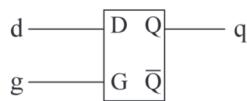
**Beispiel 61:**

Wird hingegen dem Signal "y" immer ein Wert zugewiesen, dann entfällt das Flipflop.

```
architecture behave of test2 is
    signal a, b, c, y : BIT;
begin
    process (a, b, c)
    begin
        y <= '0';
        if a AND b = '1' then
            y <= c;
        end if;
        end process;
end behave;
```

**9.2.1 Modellierung pegelgesteuerter D-Flipflops**

Transparente, pegelgesteuerte D-Flipflops haben einen Dateneingang "D", einen Steuereingang "G" und einen Datenausgang "Q" sowie den dazu inversen Ausgang " $\bar{Q}$ ". Ist der Steuereingang '1', dann wird das Signal "D" unverändert (abgesehen von einer laufzeitbedingten Verzögerung) an den Ausgang "Q" (bzw. invertiert an " $\bar{Q}$ ") weitergegeben. Wechselt der Pegel des Signals "G" von '1' nach '0', so wird der zum Zeitpunkt der Flanke am Eingang "D" vorhandene Wert auf die Ausgänge gelegt. Die Ausgangssignale werden also praktisch mit der fallenden Flanke "eingefroren".

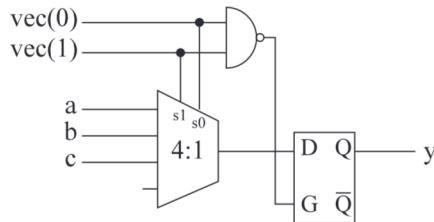


Eine Möglichkeit das Synthesetools dazu zu zwingen ein transparentes pegelgesteuertes Flipflop in die Schaltung einzufügen, wurde schon im vorhergehenden Kapitel gezeigt. Insgesamt existieren in VHDL aber noch weitere Möglichkeiten.

**Beispiel 62:**

Erzeugung eines transparenten, pegelgesteuerten Flipflops mit Hilfe einer **case**-Anweisung. Da für innerhalb der **case**-Anweisung eine Alternative existiert, in der dem Ausgangssignal "y" kein Wert zugewiesen wird, muss dementsprechend für "vec"="11" der alte Wert von "y" zwischengespeichert werden.

```
architecture behave of test3 is
    signal a, b, c, y : BIT;
    signal vec : BIT_VECTOR;
begin
    process (a, b, c, vec)
    begin
        case vec is
            when "00" => y <= a;
            when "01" => y <= b;
            when "10" => y <= c;
            when "11" => null;
        end case;
    end process;
end behave;
```

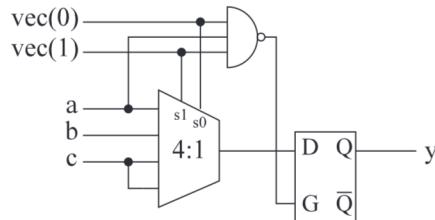


Auch wenn mehrere **case**- und/oder **if**-Anweisungen ineinander verschachtelt sind, wird immer dann, wenn nicht in *allen* Alternativzweigen ein bestimmtes Signal beschrieben wird, automatisch ein Flipflop für dieses Signal angelegt.

**Beispiel 63:**

Ein Modell mit einer verschachtelten **case-if**-Anweisung.

```
architecture behave of test4 is
    signal a, b, c, y : BIT;
    signal vec : BIT_VECTOR;
begin
    process (a, b, c, vec)
    begin
        case vec is
            when "00" => y <= a;
            when "01" => y <= b;
            when "10" => y <= c;
            when "11" =>
                if a = '0' then
                    y <= c;
                end if;
        end case;
    end process;
end behave;
```

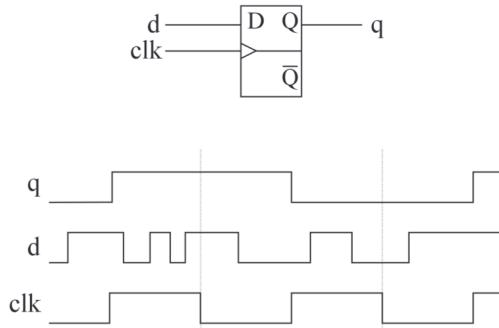
**9.2.2 Modellierung flankengetriggter D-Flipflops**

Flankengetriggerte D-Flipflops übernehmen die Daten an ihrem Dateieingang “D” nur zu den Zeitpunkten, an denen ein Signalwechsel auf ihrem Taktsignal stattfindet. Hierbei werden positiv und negativ flankengetriggerten Flipflops unterschieden. Erstere übernehmen die Daten bei einer steigenden, letztere bei einer fallenden Flanke des Taktsignals.

Zusätzlich besitzen D-Flipflops meist einen Reset und/oder einen Preset-Eingang um den Baustein in einen definierten Zustand versetzen zu können. Diese zusätzlichen Eingänge können hierbei synchron (also nur zu den entsprechenden Taktflanken) oder asynchron (sofort) wirken.

**Beispiel 64:**

Schaltungssymbol und Verhalten eines positiv flankengetriggertes D-Flipflop.



Insgesamt stehen in VHDL eine Vielzahl von Möglichkeiten zur Verfügung flankengetriggerte Flipflops zu beschreiben. Welche der Möglichkeiten in einem konkreten Design tatsächlich eingesetzt werden können, hängt jedoch von dem verwendeten Synthesetool ab.

Ist das Signal vom Typ bit, so kann eine positive Flanke i.a. über die folgenden Anweisungen erkannt werden ("clk" sei hierbei ein Taktsignal vom Typ bit):

```
if clk'event AND clk = '1' then ...
wait on clk until clk = '1'; ...
wait until clk'event AND clk = '1'; ...
```

Entsprechend können die nachfolgenden Anweisungen verwendet werden, um eine fallende Flanke zu erkennen:

```
if clk'event AND clk = '0' then ...
wait on clk until clk = '0'; ...
wait until clk'event AND clk = '0'; ...
```

Komplizierter wird die Situation, wenn das Taktsignal nicht nur zwei Werte annehmen kann, also z.B. vom Typ "std-logic" ist. Während in solchen Fällen für die Synthese die bereits vorgestellten Möglichkeiten ohne Nachteil verwendet werden können, treten bei der Simulation jedoch Probleme auf, da durch diese Anweisungen auch Wechsel von z.B. 'L' nach '0' als fallende Flanke erkannt werden. In solchen Fällen sollten wenn möglich die im "std-logic\_1164"-Package definierten Funktionen "rising-edge" und "falling-edge" eingesetzt werden:

```
if rising-edge(clk) then ...
wait until rising-edge(clk); ...
```

bzw.

```
if falling_edge(clk) then ...
wait until falling_edge(clk); ...
```

Ein ähnliches Ergebnis kann auch mit den folgenden Anweisungen erzielt werden:

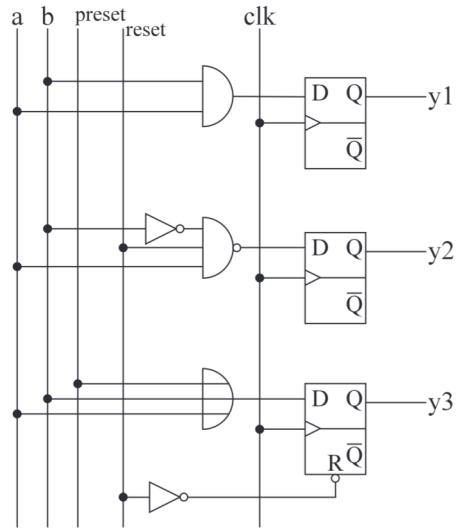
```
if clk'event AND clk = '1' AND clk'last_value = '0' then ...
if clk'event AND clk = '0' AND clk'last_value = '1' then ...
```

Allerdings werden hierbei während der *Simulation* keine Wechsel von z.B. '0' nach 'H' erkannt.

### Beispiel 65:

Drei flankengetriggerte Flipflops.

```
architecture behave of fgetriggert is
    signal a, b, y1, y2, y3 : BIT;
    signal clk, reset, preset : BIT;
begin
    process (a, b, reset, preset, clk)
    begin
        -- ohne Reset
        if rising_edge(clk) then y1 <= a AND b;
        end if;
        -- mit synchronem Reset
        if rising_edge(clk) then
            if reset AND a = '1' then y2 <= '0';
            else
                y2 <= b;
            end if;
        end if;
        -- negativ flankengetriggert, asynchroner Reset und Preset
        if reset then y3 <= '0';
        else
            if falling_edge(clk) then
                y3 <= a OR b OR preset;
            end if;
        end if;
    end process;
end behave;
```



### 9.2.3 Zähler

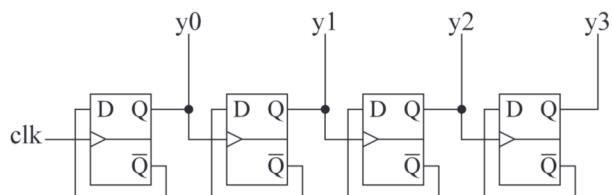
Bei Zählern unterschiedet man zwischen synchronen und asynchronen Zählern. Die Flipflops von synchronen Zählern wechseln ihren Zustand gleichzeitig mit dem zu zählenden Taktimpuls, während bei asynchronen Zählern die einzelnen Flipflops ihren Zustand zu unterschiedlichen Zeiten ändern (können).

- Asynchrone Zähler

Im allgemeinen werden asynchrone Zähler durch eine Hintereinanderschaltung von Toggle-Flipflops erzeugt, wobei der Datenausgang eines Flipflops mit den Takteingang des nachfolgenden Flipflops verdrahtet ist.

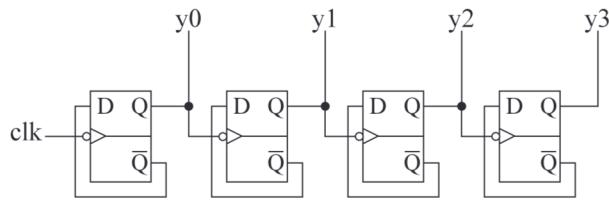
#### Beispiel 66:

Ein asynchroner 4-Bit Rückwärtszähler. Die Toggle-Flipflops wurden durch entsprechend rückgekoppelte D-Flipflops realisiert.



**Beispiel 67:**

Ein asynchroner 4-Bit Vorwärtsszähler bekommt man z.B. indem man negativ flankengetriggerte Flipflops verwendet.



Ein Vorteil von asynchronen Zählern ist ihr einfacher Aufbau. Als ungünstig erweist es sich jedoch in machen Fällen, dass die Zeit bis ein Ausgangsbit seinen korrekten Zustand einnimmt, von Bitstelle zu Bitstelle verschieden ist und zudem mit wachsender Bitstelle ansteigt. Somit ist die maximale Frequenz, für die ein asynchroner Zähler noch korrekte Zählergebnisse liefert, von der Zählerbreite begrenzt.

Um zumindest ein gleichzeitiges Umschalten aller Ausgangssignale zu gewährleisten, können die Zählerausgänge durch zusätzliche D-Flipflops mit dem Takt synchronisiert werden. Allerdings ist hierbei zu beachten, dass die Zählerergebnisse dann um einen Takt verzögert ausgegeben werden.

**Beispiel 68:**

VHDL-Beschreibung eines asynchronen 3-Bit Zählers mit Reset.

```

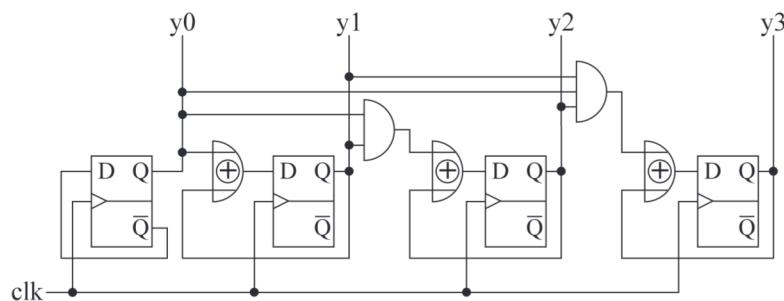
architecture behave of async3bit is
    signal clk, reset, y0, y1, y2 : bit;
begin
    process (clk, y0, y1, y2)
    begin
        if reset = '1' then y0 <= '0';
        elsif rising-edge(clk) then
            y0 <= NOT y0;
        end if;
        if reset = '1' then y1 <= '0';
        elsif rising-edge(y0) then
            y1 <= NOT y1;
        end if;
        if reset = '1' then y2 <= '0';
        elsif rising-edge(y1) then
            y2 <= NOT y2;
        end if;
    end process;
end behave;
```

- Synchrone Zähler

Im Gegensatz zu den asynchronen Zählern ändern die synchronen ihre Ausgangsbits gleichzeitig, d.h. die Takteingänge *aller* Flipflops eines synchronen Zählers sind direkt mit dem Taktsignal verbunden. Somit ist keine Nachsynchrosisierung der Ausgangssignale mehr notwendig. Zudem können sie meist mit einer höheren Frequenz betrieben werden als ein entsprechender asynchroner Baustein, da der nächste Wert jedes Flipflops direkt aus den aktuellen Flipflop-Ausgangswerten gewonnen wird.

**Beispiel 69:**

Ein synchroner 4-Bit Vorwärtzzähler.



Ein synchroner Zähler kann in VHDL z.B. durch eine entsprechende Verschaltung der benötigten Gatter- bzw. Flipflop-Bausteine beschrieben werden. Üblicher Weise kann aber auch direkt der Additionsoperator zum Inkrementieren eines Bit-Vektors verwendet werden, falls entsprechende Datentypen und Operatoren zur Verfügung stehen. Der Typ “`unsigned`” aus dem “`numeric_std`” Package verfügt z.B. über einen Additionsoperator.

**Beispiel 70:**

VHDL-Beschreibung eines synchronen 4-Bit-Zählers.

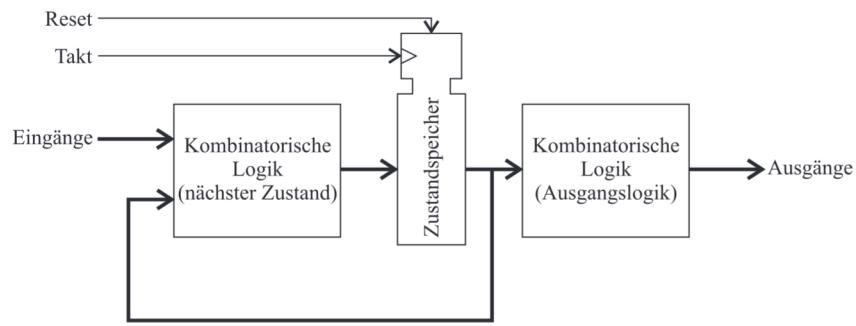
```

architecture behave of sync4bit is
    signal clk, reset : std_logic;
    signal y : unsigned(3 downto 0);
begin
    process (clk, reset)
    begin
        if reset = '1' then
            y <= (others => '0');
        elsif rising_edge(clk) then
            y <= y + 1;
        end if;
    end process;
end behave;
```

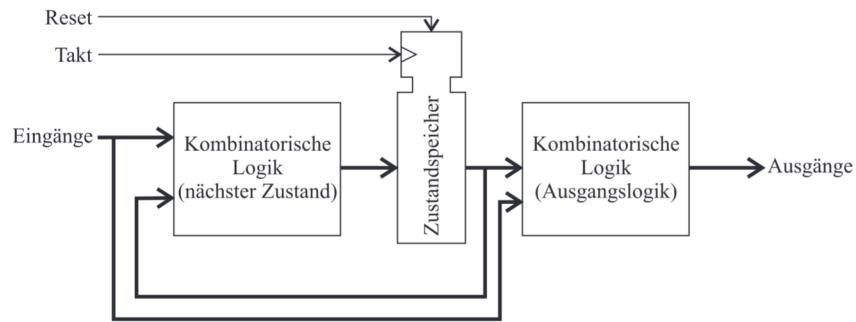
### 9.2.4 Automaten (FSM)

Automaten (Finite State Machine = FSM) werden für “kompliziertere” Steuerungs und Regelungszwecke eingesetzt. Während einfache kombinatorische Logik nämlich nur in der Lage ist auf aktuelle Eingangssignale zu reagieren, können Automaten in Abhängigkeit sowohl der aktuellen wie auch der in der Vergangenheit angelegten Signale neue Ausgangswerte erzeugen. Sie enthalten also ein “Gedächtnis” in Form eines Zustandsspeichers, welcher alle Informationen aus der Vergangenheit enthält, die der Automat für zukünftige Entscheidungen benötigt.

Ein Automat besitzt somit eine Menge von Ein- und Ausgangssignalen sowie einen Zustandsspeicher. Werden die Ausgangssignale ausschließlich mit Hilfe der im Zustandsspeicher enthaltenen Informationen erzeugt, so spricht man von einem *Moore-Automaten*. In die Berechnung der Ausgangsdaten eines *Mealy-Automaten* hingegen fließen auch die aktuellen Werte der Eingänge ein.



**Moore-Automat**

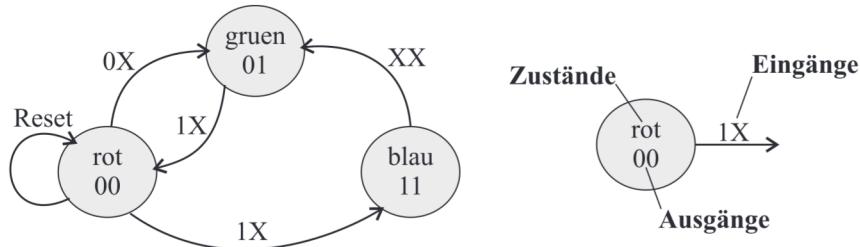


**Mealy-Automat**

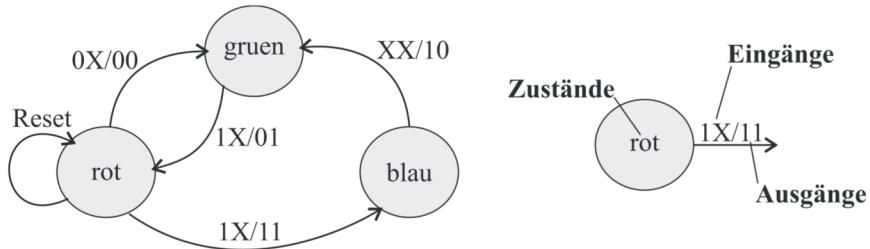
Automaten lassen sich (meist) übersichtlich mit Hilfe sogenannter Zustandsdiagramme darstellen. Kreise repräsentieren hierbei die Zustände und Pfeile die möglichen Übergänge zwischen den Zuständen. Die verschiedenen Zustände werden durch entsprechende Kennungen innerhalb der Kreise unterschieden. Bei Moore-Automaten sind außerdem die Ausgangsvariablen mit in den Kreisen angegeben. An den Übergangspfeilen befinden sich bei Moore-Automaten der Eingangsvektor, der den entsprechenden Übergang aktiviert, bei Mealy-Automaten zusätzlich noch der zu einem Übergang gehörige Ausgangsvektor.

**Beispiel 71:**

Ein Moore-Automat mit drei Zuständen

**Beispiel 72:**

Ein Mealy-Automat mit drei Zuständen



**Kodierung von Zuständen** Einen wesentlichen Einfluss auf die Eigenschaften eines Automaten hat die Kodierung der Zustände. Durch eine geeignete Wahl lassen sich z.B. die folgenden Eigenschaften beeinflussen:

- Aufwand für die Implementierung der kombinatorischen Logik
- Verzögerungszeit für die Ausgangsvariablen (werden z.B. die Zustände so kodiert, dass zumindest ein Teil der Zustandsvariablen den Ausgangsvariablen entsprechen, dann entfällt der Logik-Block an den Ausgängen)
- Robustheit gegen sich asynchron ändernde Eingangssignale

Bei der Wahl der Zustandskodierung hat man zunächst zwei grundsätzliche Möglichkeiten:

- Man kann das Synthesetool die Zustandskodierung bestimmen lassen. Das geschieht am einfachsten, indem man einen Enumerationstyp für die Zustandsvariablen des Automaten definiert.

**Beispiel 73:**

Ein Enumeration-Typ zur Kodierung der Zustände einer Ampel.

```
type Ampel.Zustaende is (rot, gelb, gruen_gelb, gruen);
```

- Die explizite Definition der Zustandsbits erlaubt es bestimmte Eigenschaften des Automaten zu optimieren. Hierbei sind die folgenden Kodierungen üblich:

Zustand	Sequentiell	Gray	One-Hot
0	000	000	00000001
1	001	001	00000010
2	010	011	00000100
3	011	010	00001000
4	100	110	00010000
5	101	111	00100000
6	110	101	01000000
7	111	100	10000000

Die Kodierung der Zustände als binäre Zahl kommt mit einer minimalen Anzahl von Zustandsspeichern aus. Der Gray-Code hat als weiteren Vorteil, dass direkt benachbarte Zustände sich lediglich in *einem* Bit unterscheiden. Die One-Hot-Kodierung benötigt zwar soviele Flipflops wie Zustände, gestattet aber eine sehr einfache Abfrage des aktuellen Zustands.

Außerdem kann natürlich eine selbstgewählte, von diesen Schemata abweichende Kodierung verwendet werden.

**Beispiel 74:**

Direkte Kodierung der Zustandsbits.

```
signal Zustand : bit_vector(1 downto 0);
constant rot : bit_vector(1 downto 0) := "00";
constant gelb : bit_vector(1 downto 0) := "01";
constant gruen : bit_vector(1 downto 0) := "10";
```

**VHDL-Beschreibung von Automaten**

Automaten bestehen im wesentlichen aus den Baugruppen Zustandsspeicher, „Nächster Zustand Logik“ und der Ausgangslogik. In der VHDL-Beschreibung können diese verschiedenen Blöcke in einem gemeinsamen Prozess definiert oder auch verschiedenen Prozessen zugeordnet werden.

**Beispiel 75:**

Ein Moore-Automat mit getrennten Prozessen für den Zustandsspeicher, die “Nächster Zustand Logik” und die Ausgangslogik. Die Beschreibung entspricht dem Beispiel-Moore-Automaten aus Kapitel 9.2.4.

Die **others**-Alternative in der **case**-Anweisung des “Nächster Zustand” Prozesses (“NZ”) ist u.U. notwendig, um das Einfügen von unnötigen Latches für das Signal “Naechster\_Zustand” durch das Synthesetool zu verhindern. Gegebenenfalls muss auch eine **others**-Alternative in die **case**-Anweisung des Ausgangslogik-Prozesses (“A”) eingefügt werden, um die Synthesierung von zwei zusätzlichen Latches für die Ausgänge zu verhindern.

```

architecture drei_prozesse of moore is
  type Zustaende is (rot, gruen, blau);
  signal Akt_Zustand, Naechster_Zustand : Zustaende;
  signal Ein, Aus : bit_vector(1 downto 0);
  signal Takt, Reset : bit;
  begin
    -- Der “Nächste Zustand”-Prozess
    NZ: process (Ein, Akt_Zustand)
    begin
      case Akt_Zustand is
        when rot =>
          if Ein(1) = '1' then Naechster_Zustand <= blau;
          else Naechster_Zustand <= gruen;
          end if;
        when gruen => Naechster_Zustand <= rot;
        when blau => Naechster_Zustand <= gruen;
        when others => Naechster_Zustand <= rot;
      end case;
    end process;
    -- Der Zustandsspeicher-Prozess
    Z: process (Takt, Reset)
    begin
      if Reset = '1' then Akt_Zustand <= rot;
      elsif Takt'event and Takt = '1' then
        Akt_Zustand <= Naechster_Zustand;
      end if;
    end process;

```

```
-- Die Ausgangslogik
A: process (Akt_Zustand)
begin
    case Akt_Zustand is
        when rot => Aus <= "00";
        when gruen => Aus <= "01";
        when blau => Aus <= "11";
        when others => Aus <= "00";
    end case;
    end process;
end drei_prozesse;
```

Wird ein Mealy-Automat mit separaten Prozessen beschrieben, so unterschieden sich die “Nächster Zustand”- und Zustandsspeicherprozesse strukturell nicht von denen eines Moore-Automaten. In die Erzeugung der Ausgänge fließen lediglich noch die Eingangssignale ein.

### Beispiel 76:

Ausgangslogik-Prozess für den Beispiel-Mealy-Automaten aus Kapitel 9.2.4

```
-- Die Ausgangslogik eines Mealy-Automaten
A: process (Akt_Zustand, Ein)
begin
    case Akt_Zustand is
        when rot => Aus <= "00";
        if Ein(1) = '1' then Aus <= "11";
        else Aus <= "00";
        end if;
        when gruen => Aus <= "01";
        when blau => Aus <= "10";
        when others => Aus <= "00";
    end case;
end process;
```

Alle Prozesse eines Automaten können auch zu einem einzigen verschmolzen werden. Der resultierende Prozess besteht dann aus zwei Abschnitten, wovon der eine die Zustandsübergänge steuert und der andere die Ausgangswerte erzeugt.

**Beispiel 77:**

Beschreibung des Moore-Automaten mit *einem* einzigen Prozess.

```

process (Takt, Ein, Reset)
  type Zustende is (rot, gruen, blau);
  variable Zustand : Zustende;
    begin
      if Reset = '1' then Zustand := rot;
      elsif Takt'event and Takt = '1' then
        case Zustand is
          when rot =>
            if Ein(1) = '1' then Zustand := blau;
            else Zustand := gruen;
            end if;
          when gruen => Zustand := rot;
          when blau => Zustand := gruen;
          when others => Zustand := rot;
        end case;
      end if;
      case Zustand is
        when rot => Aus <= "00";
        when gruen => Aus <= "01";
        when blau => Aus <= "11";
        when others => Aus <= "00";
      end case;
    end process;
  
```

Im Falle eines Moore-Automaten können die beiden Abschnitte auch noch miteinander verschmolzen werden.

**Beispiel 78:**

Alternative Beschreibung eines Moore-Automaten mit *einem einzigen* Prozess.

```
process (Takt, Ein, Reset)
  type Zustende is (rot, gruen, blau);
  variable Zustand : Zustende;
  begin
    if Reset = '1' then
      Zustand := rot;
      Aus <= "00";
    elsif Takt'event and Takt = '1' then
      case Zustand is
        when rot =>
          if Ein(1) = '1' then
            Zustand := blau;
            Aus <= "11";
          else Zustand := gruen;
          end if;
        when gruen => Zustand := rot;
        Aus <= "00";
        when blau => Zustand := gruen;
        Aus <= "01";
        when others => Zustand := rot;
        Aus <= "00";
      end case;
    end if;
  end process;
```

## A VHDL-Benutzung

### A.1 Reservierte Wörter und Operatoren

Die folgenden Wörter sind in VHDL reserviert. Sie können nicht für andere Zwecke deklariert werden.

ABS	ELSE	MAP	SELECT
ACCESS	ELSIF	MOD	SEVERITY
AFTER	END		SIGNAL
ALIAS	ENTITY	NAND	SUBTYPE
ALL	EXIT	NEW	
AND		NEXT	THEN
ARCHITECTURE		NOR	TO
ARRAY	FILE	NOT	TRANSPORT
ASSERT	FOR	NULL	TYPE
ATTRIBUTE	FUNCTION		
		OF	UNITS
BEGIN	GENERATE	ON	UNTIL
BLOCK	GENERIC	OPEN	USE
BODY	GUARDED	OR	
BUFFER		OTHERS	VARIABLE
BUS	IF	OUT	
	IN		WAIT
CASE	INOUT	PACKAGE	WHEN
COMPONENT	IS	PORT	WHILE
CONFIGURATION		PROCEDURE	WITH
CONSTANT		PROCESS	
	LABEL		XOR
DISCONNECT	LIBRARY	RANGE	
DOWNTO	LINKAGE	RECORD	
	LOOP	REGISTER	
		REM	
		REPORT	
		RETURN	

VHDL beinhaltet folgende vordefinierte Operatoren, mit denen Ausdrücke ('expressions') konstruiert werden können. Die vordefinierten Operatoren können in vier Gruppen unterteilt werden: arithmetisch, relational, logik und Verkettung.

Gruppe	Symbol	Funktion
arithmetisch (binär)	+	Addition
	-	Subtraktion
	*	Multiplikation
	/	Division
	<b>mod</b>	Modulus
	<b>rem</b>	Rest
	**	Exponent
arithmetisch (singulär)	+	Plus
	-	Minus
	<b>abs</b>	absoluter Wert
relational	=	gleich
	/=	ungleich
	<	kleiner als
	>	größer als
	<=	kleiner gleich
	>=	größer gleich
logik (binär)	<b>and</b>	logisches und
	<b>or</b>	logisches oder
	<b>nand</b>	Komplement von and
	<b>nor</b>	Komplement von or
	<b>xor</b>	Exlusiv–oder
logik (singulär)	<b>not</b>	Komplement
Verkettung	<b>&amp;</b>	Verkettung