4 Aufbau eines VHDL-Modells

Ein komplettes VHDL-Modell besteht aus mehreren Teilen (Design-Einheiten): einer Schnittstellenbeschreibung, der sog. "Entity", einer oder mehreren Verhaltens- oder Strukturbeschreibungen, den sog. "Architectures" und gegebenenfalls mehreren Konfigurationen, den sog. "Configurations".

Die Aufteilung der einzelnen Design-Einheiten auf eine oder mehrere Dateien ist willkürlich; es muß lediglich die Reihenfolge beim Compilieren der einzelnen Design-Einheiten für die Simulation beachtet werden: Entity-Architecture-Configuration:

- Stehen alle zu einem Modell gehörenden Design-Einheiten in einer Datei, dann müssen sie in der angegebenen Reihenfolge verfaßt sein, da der Compiler die Datei sequentiell abarbeitet.
- Stehen die Design-Einheiten in verschiedenen Dateien, so müssen die Dateien in der entsprechenden Reihenfolge compiliert werden.

Werden bestimmte Objekttypen, Funktionen oder Prozeduren von mehreren Modellen benötigt, so werden sie üblicherweise in unabhängigen Einheiten, den sog. "Packages", abgelegt.

4.1 Bibliotheken

Struktural aufgebaute Modelle greifen auf hierarchisch tiefer liegende VHDL-Beschreibungen zu. Diese Abhängigkeiten von anderen Modellen und Packages wird durch das Konzept der Bibliotheken ("Libraries") gehandhabt.

Bibliotheken dienen als Aufbewahrungsort für compilierte und (wieder) zu verwendende Design-Einheiten. In einem Dateisystem werden die Bibliotheken meist als eigene Verzeichnisse realisiert. Damit dieses Konzept jedoch unabhängig von den Namenskonventionen eines be-

stimmten Betriebssystems bleibt, werden die Bibliotheken nur über logische Namen (VHDL-Bezeichner) angesprochen. Der Bezug zwischen einem Verzeichnispfad und dem logischen VHDL-Namen wird in werkzeugspezifischen Konfigurationsdateien festgelegt oder über spezielle Aktionen hergestellt.

Standardmäßig legt der VHDL-Compiler die compilierten Design-Einheiten in der Bibliothek mit dem logischen Namen `work` ab. Neben den Modellen aus dieser Bibliothek (auch als Working-Library bezeichnet), können Modelle auch aus weiteren, sog. Resource-Libraries verwendet werden.

Abb. B-2 zeigt diese Zusammenhänge:

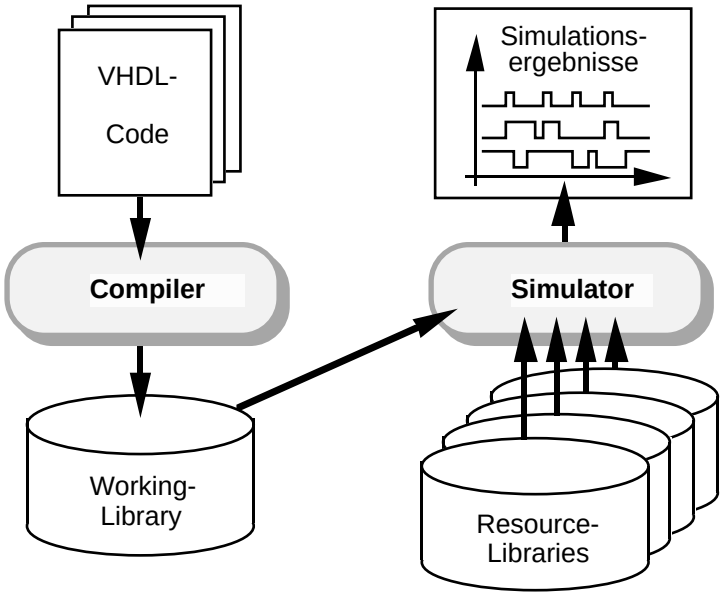

Abb. B-2: Konzept der VHDL-Bibliotheken

Innerhalb einer Bibliothek müssen die Namen (Bezeichner) der Design-Einheiten Package, Entity und Configuration eindeutig, d.h. unterscheidbar sein. Bei den Architekturen hingegen müssen nur die Bezeichner der zu ein- und derselben Entity gehörenden Architekturen voneinander verschieden sein.

4.1.1 Die LIBRARY-Anweisung

Vor dem Ansprechen eines Bibliotheksobjektes muß die verwendete Bibliothek der entsprechenden Design-Einheit bekanntgemacht werden. Dies geschieht durch folgende Anweisung:

```
LIBRARY library_name_1 {, library_name_n} ;
```

Implizit (d.h. ohne `LIBRARY`-Anweisung) sind die Bibliotheken `std` und `work` bekannt. In `std` sind die allgemeinen Packages abgelegt. Die Bibliothek `work` dient, wie bereits erwähnt, zum Abspeichern eigener, compilierter Modelle.

Die `LIBRARY`-Anweisung kann vor einer Entity, vor einer Architektur, vor einer Configuration, vor einem Package oder Package Body stehen (in der sog. "context clause" der Design-Einheiten).

4.1.2 Die USE-Anweisung

Nach der Bekanntgabe der Bibliothek geschieht das Ansprechen von Bibliothekselementen mit Hilfe von "selected names" durch den vorangestellten logischen Bibliotheksnamen. Soll ein Element, das in einem Package definiert wurde, angesprochen werden, so muß zusätzlich zum Bibliotheksnamen der Package-Name im "selected name" enthalten sein:

```
LIBRARY cmos_lib;
...
a := cmos_lib.tech_data.c1;
-- Konstante c1 im Package tech_data der Library cmos_lib
```

Das direkte Ansprechen von Funktionen und Modellen (d.h. ohne vorangestellten Bibliotheksnamen und Package-Namen) kann erfolgen, wenn vor Verwendung des Elementes in einem VHDL-Modell eine `USE`-Anweisung eingesetzt wird, z.B.:

```
USE library_name.ALL ;
USE library_name.element_name ;
```

```
USE library_name.package_name.ALL ;
USE library_name.package_name.element_name ;
```

Damit sind alle bzw. nur die spezifizierten Elemente der Bibliothek oder des Packages sichtbar. In einer USE-Anweisung können, durch Kommata getrennt, auch mehrere Bibliothekselemente stehen.

Die USE-Anweisung kann vor der Design-Einheit stehen, für die sie Gültigkeit haben soll, oder im Deklarationsteil von Design-Einheiten, Blöcken, Prozessen, Funktionen und Prozeduren sowie innerhalb der Packages. Der Gültigkeitsraum ist dementsprechend eingeschränkt.

Implizit, d.h. auch ohne USE-Anweisung, sind alle Elemente des Packages `std.standard` sichtbar. Die Elemente des normierten Packages `std.textio` müssen explizit mit einer USE-Anweisung sichtbar gemacht werden.

```
LIBRARY work, std;              -- implizit enthalten
USE std.standard.ALL;           -- implizit enthalten
USE std.textio.ALL;             -- Elemente in textio sichtbar
LIBRARY cmos_lib;               -- Bib. cmos_lib gueltig
USE cmos_lib.tech_data.ALL;     -- c1 jetzt direkt ansprechbar
```

4.2 Schnittstellenbeschreibung (Entity)

Die einzelnen Modelle eines komplexen VHDL-Entwurfs kommunizieren über die Entity, d.h. über deren Schnittstellenbeschreibung, miteinander. Die "Kommunikationskanäle" nach außen sind die sog. Ports eines Modells. Für diese werden in der Entity Name, Datentyp und Signalflußrichtung festgelegt.

Außerdem werden in der Schnittstellenbeschreibung die Parameter deklariert, die dem Modell übergeben werden können (sog. "Generics"). Mit Hilfe dieser Parameter lassen sich beispielsweise die Verzögerungs- oder Set-Up-Zeiten eines Modells von außen an das Modell übergeben. Generics können auch die Bitbreite der Ports bestimmen oder den Namen einer Datei enthalten, in der die Programmierdaten eines PLA-Modells abgelegt sind. Auf diese Weise bieten Generics eine hohe Flexibilität bei der Konfiguration von Modellen, da eine

Änderung dieser Übergabeparameter kein Neu-Compilieren des Modells erfordert.

Neben den Ports und Generics können in der Schnittstellenbeschreibung Deklarationen gemacht werden, die für die Entity und damit auch für alle zugehörigen Architekturen Gültigkeit besitzen.

Im optionalen Anweisungsteil ("Entity Statement Part", zwischen `BEGIN` und `END`) können darüberhinaus passive Prozesse, der Aufruf passiver Prozeduren oder nebenläufige Assertions stehen. Passiv bedeutet, daß keine Signalzuweisung innerhalb des Prozesses / der Prozedur erfolgt.

Die Entity ist folgendermaßen aufgebaut:

```
ENTITY entity_name IS
 [ GENERIC (
      param_1 {, param_n } :    type_name
              [ := def_value ]
  { ; further_generic_declarations  } );]

 [ PORT (
  {  port_1  {, port_n }  : IN type_name
              [ := def_value ] }
  { ; port_declarations_of_mode_OUT }
  { ; port_declarations_of_mode_INOUT }
  { ; port_declarations_of_mode_BUFFER } );]
  ...
  ...   -- USE-Anweisungen, Disconnections
  ...   -- Deklaration von:
  ...   -- Typen und Untertypen, Aliases,
  ...   -- Konstanten, Signalen, Files,
  ...   -- Unterprogrammen, Attributen
  ...   -- Definition von:
  ...   -- Unterprogrammen, Attributen
  ...   -- VHDL'93: Groups, Shared Variables
  ...
[ BEGIN
  ...
  ...   -- passive Befehle, Assertions
  ... ]
END [ENTITY] [entity_name] ;
```


Die optionale Wiederholung des Schlüsselwortes ENTITY in der END-Anweisung ist nur in ✓93 möglich.

Mit jeder Port-Deklaration der Entity wird implizit ein Signal gleichen Typs und gleichen Namens deklariert, das unter bestimmten Einschränkungen - abhängig vom Modus des Ports - in der Entity und in den zugehörigen Architekturen verwendet werden kann:

- Modus IN: Ports können nicht geschrieben werden,
- Modus OUT: Ports können nicht gelesen werden,
- Modus INOUT: Ports können gelesen und geschrieben werden,
- Modus BUFFER: Ports können gelesen und nur von einer Quelle geschrieben werden.

Zusätzliche interne Signale müssen im Deklarationsteil der Entity oder der Architektur deklariert werden.

Ein einfaches Beispiel für eine Entity (NAND3-Gatter):

```
ENTITY nand3 IS
   GENERIC       ( delay       : TIME := 2.2 ns ) ;
   PORT          ( a, b, c     : IN  std_ulogic := '0';
                   y           : OUT std_ulogic ) ;
   TYPE          tristate IS ('0', '1', 'Z');
BEGIN
   ASSERT ((a /= 'X') AND (b /= 'X') AND (c /= 'X'))
      REPORT "Ungueltiger Wert am Eingang";
END nand3;
```

4.3 Architektur (Architecture)

Die Architektur enthält die Beschreibung der Modelleigenschaften. Diese Beschreibung kann sowohl aus der Verhaltenssichtweise als auch aus der strukturalen Sichtweise erfolgen. Ein bestimmtes Modell kann also durch sein Verhalten oder durch seinen Aufbau (interne Module und deren Verbindungen) beschrieben werden. Mischformen beider Beschreibungsvarianten sind innerhalb eines Modells möglich.

Einer Schnittstellenbeschreibung können keine, eine oder mehrere Architekturen besitzen; beispielsweise kann eine Architektur eine Model-

lierung auf Register-Transfer-Ebene enthalten, während die andere eine technologiespezifische Beschreibung auf Logikebene mit detaillierten Verzögerungszeiten bereitstellt.

Auch die Architektur besteht aus einem Deklarationsteil (vor `BEGIN`) für lokale Typen, Objekte, Komponenten usw. und einem Bereich mit Anweisungen, dem "Architecture Statement Part" (zwischen `BEGIN` und `END`). Dieser enthält die eigentliche Modellbeschreibung.

Der prinzipielle Aufbau einer Architektur ist wie folgt:

```
ARCHITECTURE arch_name OF entity_name IS
   ...
   ...  -- USE-Anweisungen, Disconnections
   ...  -- Deklaration von:
   ...  -- Typen und Untertypen,
   ...  -- Aliases, Konstanten,
   ...  -- Signalen, Files, Komponenten,
   ...  -- Unterprogrammen, Attributen
   ...  -- Definition von:
   ...  -- Unterprogrammen, Attributen,
   ...  -- Konfigurationen
   ...
   ...  -- VHDL'93: Groups, Shared Variables
   ...
BEGIN
   ...
   ...  -- nebenlaeufige Anweisungen
   ...  -- zur strukturalen Modellierung
   ...  -- und Verhaltensmodellierung
   ...
END [ARCHITECTURE] [arch_name];
```

Die Wiederholung des Schlüsselwortes `ARCHITECTURE` in der `END`-Anweisung ist mit ✓**93** optional möglich.

Im Unterschied zu Programmiersprachen, wie z.B. "C", hat VHDL die Aufgabe, neben rein sequentiellen Funktionsabläufen auch Hardware zu beschreiben. Die in einer Hardwareeinheit enthaltenen Funktionsmodule arbeiten aber nicht ausschließlich sequentiell, sondern parallel,

d.h. die einzelnen Funktionen laufen nicht immer nacheinander, sondern eventuell auch gleichzeitig ab.

Um diese Eigenschaft zu beschreiben, wird das Konzept der parallelen (nebenläufigen, oder auch "concurrent") Anweisungen verwendet, die im Idealfall gleichzeitig bearbeitet werden. Da die Simulation eines VHDL-Modells in der Regel jedoch auf **einem** Prozessor abläuft, ist dies nicht möglich. Deshalb wurde ein spezieller Simulationsalgorithmus entwickelt, der ein "quasi paralleles", d.h. für den Benutzer nicht direkt sichtbares, sequentielles Abarbeiten der nebenläufigen Befehle gestattet. Dieser Ablauf wird in Kapitel 8 erläutert.

Aus oben genanntem Grund sind die VHDL-Anweisungen in zwei Kategorien einzuteilen: sequentielle und nebenläufige Anweisungen.

Innerhalb des Anweisungsteils einer Architektur sind alle Sprachkonstrukte nebenläufig. **Nebenläufige Anweisungen** sind z.B. die Instantiierung von Komponenten, die `BLOCK`- und `GENERATE`-Anweisungen sowie insbesondere auch sämtliche Prozesse:

`... <= ...` (Signalzuweisung)
`ASSERT`-Anweisung
`BLOCK`-Anweisung
Komponenteninstantiierung
`GENERATE`-Anweisung
`PROCESS`-Anweisung
Prozeduraufruf

Sequentielle Anweisungen entsprechen den von Programmiersprachen her bekannten Konstrukten. Sie dürfen nur innerhalb von Prozessen, Funktionen oder Prozeduren stehen:

`... <= ...` (Signalzuweisung)
`... := ...` (Variablenzuweis.)
`ASSERT`-Anweisung
`CASE`-Anweisung
`EXIT`-Anweisung
`IF`-Anweisung
`LOOP`-Anweisung
`NEXT`-Anweisung
`NULL`-Anweisung
Prozeduraufruf
`REPORT`-Anweisung ✓93
`RETURN`-Anweisung
`WAIT`-Anweisung

4.4 Konfiguration (Configuration)

Die Design-Einheit Konfiguration dient zur Beschreibung der Konfigurationsdaten eines VHDL-Modells. Zunächst wird darin festgelegt, welche Architektur zu verwenden ist. Bei strukturalen Beschreibungen kann außerdem angegeben werden, aus welchen Bibliotheken die einzelnen Submodule entnommen werden, wie sie eingesetzt (verdrahtet) werden und welche Parameterwerte (Generics) für die Submodule gelten. Eine Entity kann mehrere Konfigurationen besitzen.

In der Konfiguration wird zwischen deklarativen und den eigentlichen Konfigurationsanweisungen unterschieden. Die Konfigurationsanweisungen beschreiben - gegebenenfalls hierarchisch - die Parameter und Instanzen der verwendeten Architektur.

Die Rahmensyntax der Design-Einheit Konfiguration lautet wie folgt:

```
CONFIGURATION conf_name OF entity_name IS
   ...
   ...  -- USE- Anweisungen und
   ...  -- Attributzuweisungen,
   ...  -- Konfigurationsanweisungen
   ...
END  [CONFIGURATION]  [conf_name] ;
```

Die optionale Wiederholung des Schlüsselwortes `CONFIGURATION` ist wieder nur in ✓**93** gestattet.

Da die Sprache VHDL in hohem Maße die Wiederverwendung existierender Modelle unterstützen soll, bietet sie eine Vielzahl an Konfigurationsmöglichkeiten. Aus diesem Grund wird in Kapitel 7 des Teils B näher auf die Details dieser Design-Einheit eingegangen.

4.5 Package

Packages dienen dazu, häufig benötigte Datentypen, Komponenten, Objekte, etc. einmalig zu deklarieren. Diese Deklarationen können dann in verschiedenen VHDL-Modellen verwendet werden. Packages eignen sich insbesondere, um globale Informationen innerhalb eines komplexen Entwurfs oder innerhalb eines Projektteams einmalig und

damit widerspruchsfrei festzulegen. Typische Anwendungen sind Packages, die einen bestimmten Logiktyp (z.B. vierwertige Logik) und die zugehörigen Operatoren beschreiben. Andere Packages könnten beispielsweise eine Sammlung von mathematischen Funktionen (`sin(x)`, `cos(x)`, etc.) enthalten.

VHDL unterscheidet zwischen **Package** und **Package Body**, die beide eigenständig sind und auch getrennt compiliert werden. Ursache für diese Vereinbarung sind die Abhängigkeiten beim Compilieren der Design-Einheiten. Da die anderen Design-Einheiten auf den verwendeten Packages aufbauen, müßten bei der Änderung eines Unterprogramms im Package alle vom Package abhängigen Design-Einheiten neu compiliert werden. Durch die Trennung von Deklaration (enthalten im Package) und Funktionalität bzw. Definition (enthalten im Package Body) kann dies vermieden werden. Entities, Architectures und Configurations basieren nur auf den Deklarationen des Packages. Änderungen im Package Body erfordern damit kein Neu-Compilieren der übrigen Design-Einheiten. Dieses Konzept, das auch als "deferring" bezeichnet wird, unterstützt die Änderungsfreundlichkeit der VHDL-Modelle ("deferring" steht für "Verschiebung" der Implementierung in den Package Body).

Die Syntax der Design-Einheit **Package** lautet wie folgt:

```
PACKAGE pack_name IS
   ...
   ...  -- USE-Anweisungen, Disconnections
   ...  -- Deklarationen von:
   ...  -- Typen und Untertypen,
   ...  -- Aliases, Konstanten,
   ...  -- Signalen, Files, Komponenten,
   ...  -- Unterprogrammen, Attributen
   ...  -- Definition von:
   ...  -- Attributen
   ...
   ...  -- VHDL'93: Groups, Shared Variables
   ...
END [PACKAGE] [pack_name] ;
```

Das Schlüsselwort `PACKAGE` kann nur in ✓**93** wiederholt werden.

Der **Package Body** kann neben der Definition von Unterprogrammen auch spezielle Deklarationsanweisungen enthalten. Der Zusammenhang mit dem Package wird durch den identischen Namen (`pack_name`) hergestellt. Der Package Body weist folgende Struktur auf:

```
PACKAGE BODY pack_name IS
   ...
   ...  -- Deklarationen von: Typen und
   ...  -- Untertypen, Aliases, Konstanten,
   ...  -- Files, Unterprogrammen
   ...  -- Definition von: Unterprogrammen
   ...  -- USE-Anweisungen
   ...
END [PACKAGE BODY] [pack_name] ;
```

Die optionale Wiederholung der Schlüsselworte `PACKAGE BODY` ist wiederum nur in ✓**93** möglich.

```
PACKAGE fft_projekt IS
   TYPE tristate IS ('0', '1', 'Z');
   CONSTANT standard_delay : time;    -- ohne Wertangabe!
END fft_projekt;
```

```
PACKAGE BODY fft_projekt IS
   CONSTANT standard_delay : time := 2 ns;
   -- Wiederholung der Deklaration,
   -- Wertangabe aber nur im Package Body: "deferring"
END fft_projekt;
```

4.6 Abhängigkeiten beim Compilieren

Die bei der Erläuterung der Packages bereits angeklungenen Abhängigkeiten beim Compilieren der Design-Einheiten werden in Abb. B-3 illustriert. Beispielsweise müssen nach einer Änderung in einer Entity auch alle zugehörigen Architekturen und Konfigurationen neu übersetzt werden. Eine Änderung im Package Body erfordert dagegen nur das erneute Compilieren dieser Design-Einheit.

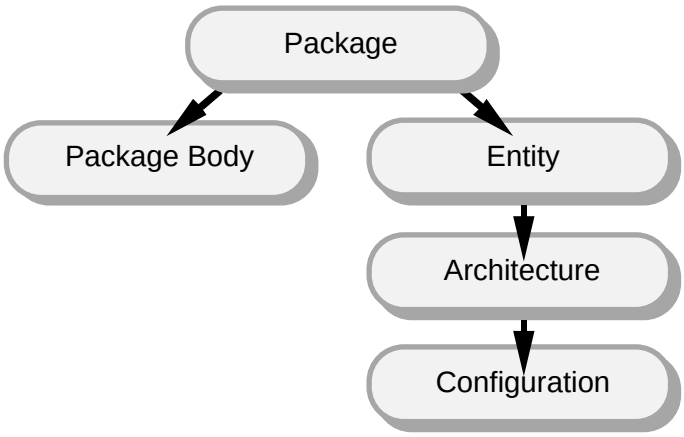

Abb. B-3: Abhängigkeiten beim Compilieren von VHDL-Modellen

5 Strukturale Modellierung

Strukturale Modellierung bedeutet im allgemeinen die Verwendung (= Instantiierung) und das Verdrahten von Komponenten in Form einer Netzliste. VHDL bedient sich dazu einer dreistufigen Vorgehensweise, die zwar viele Freiheitsgrade bietet, für den Anfänger jedoch sehr unübersichtlich ist. Zur Einführung in die strukturale Modellierung soll deshalb ein RS-Flip-Flop betrachtet werden, das aus zwei gleichartigen NAND2-Gattern aufgebaut ist (siehe Abb. B-4).

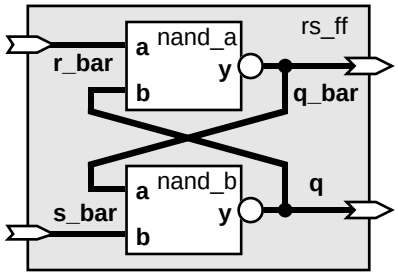

Abb. B-4: Struktur eine RS-Flip-Flops

Die Schnittstelle des RS-Flip-Flops hat folgendes Aussehen:

```
ENTITY rs_ff IS
   PORT (r_bar, s_bar : IN bit := '0';
         q, q_bar     : INOUT bit);
   -- Ports als INOUT, da sie auch gelesen werden muessen
END rs_ff;
```

Für die strukturale Modellierung des Flip-Flops mit der Sprache VHDL wird eine Komponentendeklaration, zwei Komponenteninstantiierungen und eine Komponentenkonfiguration benötigt:

```
ARCHITECTURE structural OF rs_ff IS
   COMPONENT nand2_socket      -- Komponentendeklaration
      PORT (a, b : IN bit; y : OUT bit);
   END COMPONENT;
BEGIN
   nand_a : nand2_socket       -- Komponenteninstantiierung
      PORT MAP (a => r_bar, b => q, y => q_bar);
   nand_b : nand2_socket       -- Komponenteninstantiierung
      PORT MAP (a => q_bar, b => s_bar, y => q);
END structural;
```

```
CONFIGURATION rs_ff_config OF rs_ff IS
   FOR structural
      FOR ALL : nand2_socket   -- Komponentenkonfiguration
         USE ENTITY work.nand2 (behavioral);
      END FOR;
   END FOR;
END rs_ff_config;
```

Für ein besseres Verständnis kann man sich diese dreistufige VHDL-Syntax mit ICs und zugehörigen Sockeln bzw. Sockeltypen vorstellen. Das strukturale Modell des Flip-Flops würde also eine Leiterplatte mit zwei gesockelten NAND2-Gattern nachbilden.

- **Komponentendeklaration**
  Als erster Schritt wird ein Prototyp des Komponentensockels im Deklarationsteil der Architektur, im Deklarationsteil eines Blocks oder im Package deklariert (hier: `nand2_socket`).
- **Komponenteninstantiierung**
  In der Architektur / im Block werden Instanzen dieses Sockeltyps gebildet und verdrahtet (hier: `nand_a` und `nand_b`).
- **Komponentenkonfiguration**
  In der Konfiguration schließlich wird festgelegt, welches bereits compilierte VHDL-Modell für die jeweiligen Instanzen der Komponenten einer strukturalen Architektur verwendet wird; mit anderen Worten: welcher IC-Typ in die einzelnen Sockel eingesetzt wird. Im Beispiel wird für alle Instanzen der Komponente `nand2_socket` die Entity `nand2` aus der Bibliothek `work` verwendet (in Klammern: zugehörige Architektur).

5.1 Komponentendeklaration und -instantiierung

In einer **Komponentendeklaration** werden im allgemeinen neben den Ports (entsprechend den Pins des Sockeltyps) auch die zu übergebenden Parameter (Generics) aufgeführt. Die Komponentendeklaration ist somit ein Abbild der Entity des einzusetzenden Modells (ICs).

```
COMPONENT comp_name
 [ GENERIC (
      param_1 {, param_n } :    type_name
              [ := def_value ]
   { ; further_generic_declarations  } );]

 [ PORT (
   {  port_1  {, port_n }   : IN type_name
              [ := def_value ] }
   { ; port_declarations_of_mode_OUT }
   { ; port_declarations_of_mode_INOUT }
   { ; port_declarations_of_mode_BUFFER } );]
END COMPONENT ;
```

Auch hier greift die Vereinheitlichung der Rahmensyntax von ✓**93**, so daß sich folgende Syntax-Alternative ergibt:

```
COMPONENT comp_name [IS]
   ...
END COMPONENT [comp_name] ;
```

Einige Beispiele von Komponentendeklarationen:

```
COMPONENT inv
  GENERIC (tpd_lh, tpd_hl : time := 0.8 ns) ;
  PORT (a : IN bit; y : OUT bit) ;
END COMPONENT ;

COMPONENT or2
  GENERIC (tpd_lh : time := 1.5 ns; tpd_hl : time := 1 ns) ;
  PORT (a,b : IN bit; y : OUT bit) ;
END COMPONENT ;
```



```
COMPONENT and2
  GENERIC (tpd_lh : time := 1 ns; tpd_hl : time := 1.5 ns) ;
  PORT (a,b : IN bit; y : OUT bit) ;
END COMPONENT ;

COMPONENT and3
  GENERIC (tpd_lh : time := 1 ns; tpd_hl : time := 1.8 ns) ;
  PORT (a,b,c : IN bit; y : OUT bit) ;
END COMPONENT ;
```

Die eigentliche Netzliste wird erst durch das **Instantiieren** dieser Sokkeltypen und gleichzeitiges Verdrahten durch Zuweisung von Signalnamen an die Ports erzeugt. Dabei können auch Parameter übergeben werden.

Jede Komponente erhält bei der Instantiierung einen eigenen Referenznamen (`inst_name`):

```
inst_name : comp_name
  [ GENERIC MAP ( ... ) ]  --  Generic-Map-Liste
  [ PORT MAP ( ... )    ]; --  Port-Map-Liste
```

Es existieren verschiedene Möglichkeiten, die Ports und Parameter von Komponente und zugehöriger Instanz zuzuordnen. Um die dafür erforderliche Syntax zu verstehen, sind zunächst folgende Ebenen zu unterscheiden:

- Ports und Generics in der Schnittstellenbeschreibung (Entity) der zu instantiierenden Komponente (sog. "**formals**"),
- Ports und Generics in der Komponentendeklaration (sog. "**locals**") und
- lokale Signale innerhalb der Architektur oder die von der Architektur zu übergebenden Parameterwerte (sog. "**actuals**").

Bei der Komponenteninstantiierung innerhalb von strukturalen Architekturen werden **locals** mit **actuals** verbunden. Angaben von Parameterwerten in der Generic-Map überschreiben dabei die Defaultwerte in der Komponentendeklaration.

Generic-Map und Port-Map bestehen:

- aus einer durch Kommata getrennten Liste von unmittelbar aufeinanderfolgenden Signalnamen (actuals) bzw. Parameterwerten, wobei die Zuweisung in der gleichen Reihenfolge wie in der Komponentendeklaration (locals) erfolgt ("positional association"):

```
actual_1 {, actual_n}
```

- oder aus einer durch Kommata getrennten Liste von expliziten Zuweisungen in beliebiger Reihenfolge ("named association"):

```
      local_1   =>  actual_1
  {,  local_n   =>  actual_n}
```

- oder aus einer Kombination beider Möglichkeiten, wobei die zweite Variante der ersten nachfolgen muß.

Das folgende Beispiel für eine strukturale Architektur greift die oben gezeigten Komponentendeklarationen auf. Es handelt sich um ein 3-2-AND-OR-INVERT-Komplexgatter mit folgendem Schaltbild:

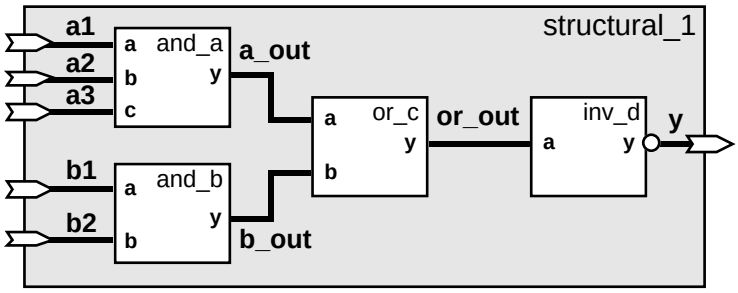

Abb. B-5: Schaltbild eines 3-2-AND-OR-INVERT-Komplexgatters

```
ENTITY aoi IS
  PORT ( a1, a2, a3, b1, b2 : IN bit;
         y : OUT bit ) ;
END aoi ;
```



```
ARCHITECTURE structural_1 OF aoi IS
  SIGNAL a_out, b_out, or_out : bit ;  -- interne Signale
  ...
  ...           -- Komponentendeklarationen wie oben
  ...
BEGIN
---- verschiedene Varianten der Instantiierung: ------------
  -- position. association, tpd_lh = 1.2 ns, tpd_hl = 2.4 ns
  and_a : and3 GENERIC MAP (1.2 ns,2.4 ns)
               PORT MAP (a1,a2,a3,a_out) ;
  -- named association
  and_b : and2 GENERIC MAP (tpd_hl=>1.9 ns,tpd_lh=>1.1 ns)
               PORT MAP (b=>b1,y=>b_out,a=>b2) ;
  -- ohne Generic-Map: Default Generics: 1.5 ns bzw. 1.0 ns
  or_c  : or2  PORT MAP (a_out,y=>or_out,b=>b_out) ;
  -- unvollst. Generic-Map: tpd_hl = Defaultwert: 0.8 ns
  inv_d : inv  GENERIC MAP (0.7 ns) PORT MAP (or_out,y) ;
END structural_1 ;
```

Zusätzlich zu diesem Beispiel sei noch auf die einfachere Architektur `structural` des Halbaddierers aus Teil A zur Verdeutlichung der strukturalen Modellierung verwiesen.

Erlaubt ist in VHDL auch das Schlüsselwort `OPEN` als actual zur Kennzeichnung nicht angeschlossener Ports (nicht verdrahteter Sokkelpins). Für solche Ports wird bei Simulationsbeginn der Defaultwert angenommen, der in der Komponentendeklaration angegeben ist.

Eine weitere Vereinbarung für Eingangsports in der Syntax ✓**93** erlaubt die Angabe von Ausdrücken als actuals.

Ein Beispiel verdeutlicht die Möglichkeiten von nicht angeschlossenen Ports bzw. die Zuweisung von Ausdrücken nach ✓**93**. Es handelt sich um das obige Komplexgatter mit veränderter Architektur. Der Inverter und das ODER-Gatter sind hier zu einem NOR-Gatter zusammengefaßt.

```
ARCHITECTURE structural_2 OF aoi IS  -- !!! VHDL'93 !!!
  SIGNAL a_out, b_out : bit ;        -- interne Signale
  COMPONENT nor3                     -- invertierendes OR3
    PORT (a,b,c : IN bit := '0' ; y : OUT bit) ;
  END COMPONENT nor3;
  COMPONENT and3                     -- nur 3-fach AND
    PORT (a,b,c : IN bit := '0' ; y : OUT bit) ;
  END COMPONENT ;
BEGIN
  and_a : and3 PORT MAP (a1,a2,a3,a_out) ;
  and_b : and3 PORT MAP (a=>b2,b=>b1,c=>'1',y=>b_out) ;
         -- VHDL'93: Port "c" wird konstant auf '1' gelegt
  nor_c : nor3 PORT MAP (OPEN,a_out,y=>y,b=>b_out) ;
         -- VHDL'87/93: Port "a" bleibt unangeschlossen: '0'
END structural_2 ;
```

Mit ✓**93** wurde eine Möglichkeit geschaffen, bereits bei der Komponenteninstantiierung anzugeben, welches Modell zu verwenden ist. Diese sog. "direkte Instantiierung" kann mit dem Einlöten eines ICs in die Leiterplatte ohne Sockel verglichen werden. Bei einmalig verwendeten Komponenten hat dies durchaus Vorteile, mehrfach verwendete Komponenten hingegen sollten mit der herkömmlichen Methode instantiiert werden.

Die direkte Instantiierung ✓**93** benötigt weder eine Komponentendeklaration noch eine Konfiguration. Sie hat folgende Syntax:

```
inst_name : CONFIGURATION config_name
 [ GENERIC MAP ( ... ) ]  -- Generic-Map-Liste
 [ PORT MAP ( ... )    ]; -- Port-Map Liste

inst_name : ENTITY entity_name [(arch_name)]
 [ GENERIC MAP ( ... ) ]  -- Generic-Map-Liste
 [ PORT MAP ( ... )    ]; -- Port-Map-Liste
```

Wird im zweiten Fall keine Architektur angegeben, so wird die Entity ohne Architektur instantiiert. Dies ist u.a. dann sinnvoll, wenn die Funktion des Modells nur im Anweisungsteil der Entity (durch passive Prozeduren, Prozesse und Assertions) definiert ist.

Natürlich ist die herkömmliche Instantiierung in ✓93 weiterhin erlaubt. Das Schlüsselwort COMPONENT kann nun hinzugefügt werden:

```
inst_name : [ COMPONENT ] comp_name
 [ GENERIC MAP ( ... ) ]  -- Generic-Map-Liste
 [ PORT MAP ( ... )    ]; -- Port-Map Liste
```

Als Beispiel für die direkte Instantiierung in ✓93 soll wieder das Komplexgatter dienen. Man beachte, daß für diese Version weder Komponentendeklarationen noch eine Konfiguration benötigt werden:

```
ARCHITECTURE structural_3 OF aoi IS  -- !! VHDL'93-Syntax !!
  SIGNAL a_out, b_out : bit ;        -- interne Signale
BEGIN
  and_a : ENTITY work.and3 (behavioral)
          PORT MAP (a1,a2,a3,a_out) ;
  and_b : ENTITY work.and2 (behavioral)
          PORT MAP (b1,b2,b_out) ;
  nor_c : CONFIGURATION work.nor2_config
          PORT MAP (a_out,b_out,y) ;
END structural_3 ;
```

5.2 BLOCK-Anweisung

Die BLOCK-Anweisung dient zur Gliederung eines Modells, ohne eine Modellhierarchie mit instantiierten Untermodellen einführen zu müssen. Ähnlich einer Architektur können in einem Block lokale Deklarationen getroffen werden. Ein Block kann sogar wie eine eigenständige Einheit mit Generics und Ports verwaltet werden. Auch ist es möglich, daß innerhalb eines Blocks wieder eine Partitionierung in Blöcke stattfindet, so daß sich prinzipiell in einer Architektur beliebig komplexe Strukturen aufbauen lassen.

Die Syntax von Blöcken hat folgende Form:

```
block_name : BLOCK   [IS]
     ...
     ...   -- USE-Anweisungen, Disconnections
     ...   -- Generics und Generic-Map
     ...   -- Ports und Port-Map
```

```
      ...   -- Deklaration von:
      ...   -- Typen und Untertypen,
      ...   -- Aliases, Konstanten,
      ...   -- Signalen, Files, Komponenten,
      ...   -- Unterprogrammen, Attributen
      ...   -- Definition von:
      ...   -- Unterprogrammen, Attributen
      ...   -- Konfigurationen
      ...
      ...   -- VHDL'93: Groups, Shared Variables
      ...
  BEGIN
      ...
      ...   -- nebenlaeufige Anweisungen
      ...   -- zur strukturalen Modellierung
      ...   -- und Verhaltensmodellierung
      ...
  END BLOCK [block_name] ;
```

Die Verwendung des Schlüsselwortes `IS` in der `BLOCK`-Anweisung ist nur in ✓**93** möglich.

Als Beispiel für die Block-Anweisung dient erneut das AOI-Komplexgatter, das nun in zwei Stufen partitioniert wurde. Die beiden Stufen sind als Blöcke realisiert (siehe Abb. B-6). Der zweite Block hat zur Veranschaulichung der `BLOCK`-Syntax eigene Ports mit einer entsprechenden Port-Map.

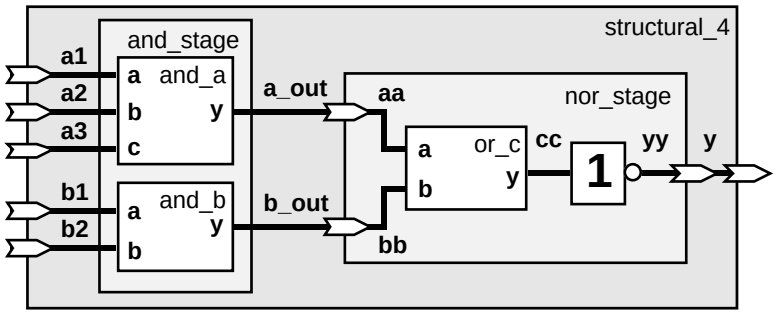

Abb. B-6: Strukturierung des Komplexgatters durch Blöcke


```
ARCHITECTURE structural_4 OF aoi IS
  SIGNAL a_out, b_out : bit ;        -- interne Signale
BEGIN
  and_stage : BLOCK
    COMPONENT and2
      PORT (a,b : IN bit; y : OUT bit) ;
    END COMPONENT ;
    COMPONENT and3
      PORT (a,b,c : IN bit; y : OUT bit) ;
    END COMPONENT ;
  BEGIN
     and_a : and3 PORT MAP (a1,a2,a3,a_out);
     and_b : and2 PORT MAP (b1,b2,b_out);
  END BLOCK and_stage;

  nor_stage : BLOCK
    PORT (aa,bb : IN bit; yy : OUT bit) ;
    PORT MAP (aa=>a_out, bb=>b_out, yy=>y);
    SIGNAL cc : bit;            -- block-internes Signal
    COMPONENT or2               -- nicht invertierend !
      PORT (a,b : IN bit; y : OUT bit) ;
    END COMPONENT ;
  BEGIN
    or_c : or2 PORT MAP (a=>aa,b=>bb,y=>cc) ;
    yy <= not cc;               -- Invertierung
  END BLOCK nor_stage ;
END structural_4 ;
```

5.3 GENERATE-Anweisung

In vielen Fällen treten in elektronischen Systemen regelmäßige Strukturen auf, bei denen ein Modul aus mehreren gleichartigen Submodulen besteht. Eine Darstellung solcher Strukturen mit den bisher eingeführten Beschreibungsmitteln der Sprache VHDL würde sehr umfangreich werden. Mit Hilfe der GENERATE-Anweisung lassen sich jedoch regelmäßige Strukturen einfach und auch flexibel gestalten.

Abhängig von einer Bedingung (*condition*) oder einem diskreten Bereich (*disc_range*) wird eine Reihe von nebenläufigen Anweisungen ein- oder mehrfach ausgeführt. Der diskrete Bereich kann wie gewohnt durch zwei diskrete Bereichsgrenzen und eines der beiden

Schlüsselwörter `TO` oder `DOWNTO` bzw. mit einem diskreten Wertebereich mit Hilfe des Attributs `RANGE` angegeben werden:

```
gen_name : IF condition GENERATE
    ...
    ...  -- Nebenlaeufige Anweisungen
    ...
END GENERATE [gen_name] ;

gen_name : FOR var_name IN disc_range GENERATE
    ...
    ...  -- Nebenlaeufige Anweisungen
    ...
END GENERATE [gen_name] ;
```

Bei der zweiten Beschreibungsvariante wird eine Laufvariable für den Index durch die Rahmensyntax automatisch deklariert; sie kann innerhalb der Anweisung verwendet werden. Typische Anwendungsfälle für `GENERATE`-Anweisungen sind aus mehreren Speicherelementen aufgebaute Register. Hierzu ein Beispiel (Abb. B-7):

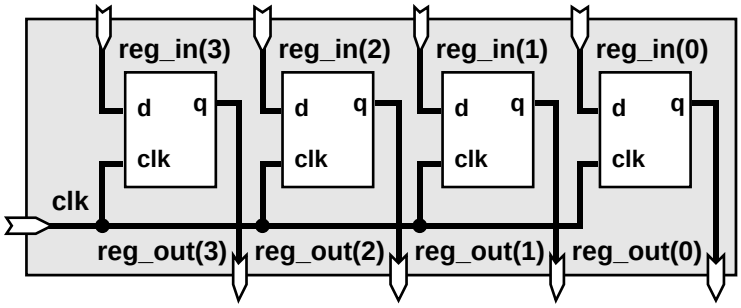

Abb. B-7: n-Bit Register (n=4)

```
ENTITY n_bit_register IS
  GENERIC (n : IN positive := 4) ;
  PORT (clk     : IN  bit ;
        reg_in  : IN  bit_vector(n-1 DOWNTO 0) ;
        reg_out : OUT bit_vector(n-1 DOWNTO 0) ) ;
END n_bit_register ;
```

```
ARCHITECTURE structural OF n_bit_register IS
  COMPONENT d_ff_socket
    PORT (d, clk : IN bit ; q : OUT bit) ;
  END COMPONENT ;
BEGIN
  reg : FOR i IN n-1 DOWNTO 0 GENERATE
    d_ff_instance : d_ff_socket
      PORT MAP (reg_in(i),clk, reg_out(i)) ;
  END GENERATE ;
END structural ;
```

Die Länge dieses Registers ist in der Beschreibung nicht fixiert. Sie kann erst beim Konfigurieren des Modells über den Parameter n festgelegt werden. Damit lassen sich von diesem Modell auch mehrere Instanzen unterschiedlicher Länge erzeugen.

Falls nicht alle Instanzen nach dem gleichen Schema verdrahtet sind, müssen in der GENERATE-Anweisung Bedingungen eingesetzt werden. Als Beispiel dient hier ein Schieberegister beliebiger Länge, bei dem das erste und letzte Modul eine spezielle Verdrahtung besitzen (Abb. B-8):

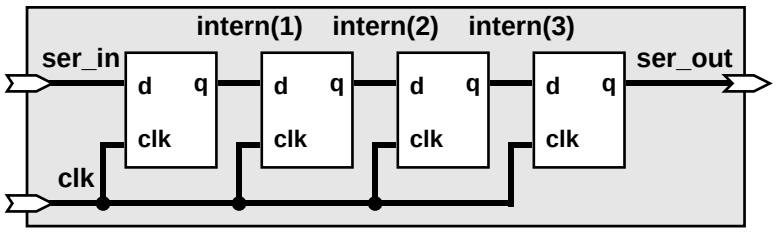

Abb. B-8: n-Bit Schieberegister (n=4)

```
ENTITY shift_register IS
  GENERIC (n: IN positive RANGE 2 TO 64 := 4);
  PORT (clk, ser_in : IN  bit ;
        ser_out     : OUT bit ) ;
END shift_register ;
```

```
ARCHITECTURE structural OF shift_register IS
  SIGNAL intern : bit_vector(1 TO n-1) ;
  COMPONENT d_ff_socket
    PORT (d, clk : IN bit ; q : OUT bit) ;
  END COMPONENT ;
BEGIN
  reg : FOR i IN n DOWNTO 1 GENERATE
-- erstes D-FF: mit Eingang verdrahtet ---------------------
    reg_begin : IF i = 1 GENERATE
      d_ff_begin : d_ff_socket
        PORT MAP (ser_in,clk,intern(i)) ;
    END GENERATE ;
-- mittlere D-FFs ------------------------------------------
    reg_middle : IF i > 1 AND i < n GENERATE
      d_ff_middle : d_ff_socket
        PORT MAP (intern(i-1),clk,intern(i)) ;
    END GENERATE ;
-- letztes D-FF: mit Ausgang verdrahtet --------------------
    reg_end : IF i = n GENERATE
      d_ff_end : d_ff_socket
        PORT MAP (intern(i-1),clk,ser_out) ;
    END GENERATE ;
  END GENERATE ;
END structural ;
```

Mit ✓**93** wird ein optionaler Deklarationsteil für die GENERATE-Anweisung ermöglicht. Er entspricht dem Deklarationsteil von BLOCK oder ARCHITECTURE:

```
gen_name : FOR var_name IN disc_range GENERATE
[   ...
    ...  Deklarationsteil
    ...
BEGIN ]
    ...
END GENERATE [gen_name] ;
```

Die gleiche Erweiterungsmöglichkeit von ✓**93** gilt auch für die IF-Variante der GENERATE-Anweisung.

6 Verhaltensmodellierung

Im vorangegangenen Kapitel wurde gezeigt, wie sich hierarchische Strukturen, wie z.B. die in Abb. B-9 gezeigte, modellieren lassen. Die unterste Ebene in den einzelnen Zweigen des Strukturbaumes jedoch kann nicht struktural beschrieben werden, da diese Modelle nicht weiter in Sub-Modelle unterteilt sind. Für diese Modelle stellt VHDL zahlreiche Konstrukte zur Verfügung, mit denen sich das Modellverhalten nachbilden läßt.

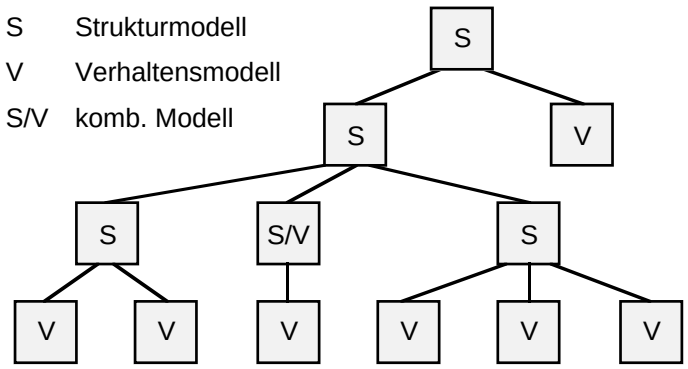

Abb. B-9: Hierarchischer Modellaufbau

Als einführendes Beispiel zur Verhaltensmodellierung soll das nachstehende Modell eines Zählers dienen, der bei steigenden Taktflanken zyklisch von 0 bis 4 zählt, falls der `enable`-Eingang den Wert `'1'` besitzt. Mit einem low-aktiven `reset`-Signal kann der Zähler asynchron zurückgesetzt werden.

```
ENTITY count5 IS
  PORT (clk, enable, reset : IN  bit;
        q                  : OUT bit_vector (2 DOWNTO 0));
END count5;
```

```
ARCHITECTURE behavioral OF count5 IS
  SIGNAL state : integer RANGE 0 TO 4 := 0; -- Zaehlerstand
BEGIN
  count : PROCESS (clk, reset)     -- Prozess zum Zaehlen
  BEGIN
    IF reset = '0' THEN
      state <= 0;                  -- Ruecksetzen
    ELSIF (enable = '1' AND clk'EVENT AND clk = '1') THEN
      IF state < 4 THEN
        state <= state + 1;        -- Hochzaehlen
      ELSE
        state <= 0;                -- Ueberlauf
      END IF;
    END IF;
  END PROCESS count;
-- nebenlaeufige Signalzuweisung an den Ausgang q ----------
  q <= ('0','0','0') WHEN state = 0 ELSE
       ('0','0','1') WHEN state = 1 ELSE
       ('0','1','0') WHEN state = 2 ELSE
       ('0','1','1') WHEN state = 3 ELSE
       ('1','0','0');
END behavioral;
```

Wie aus dem Beispiel hervorgeht, beschreibt die Verhaltensmodellierung, wie die Eingangs- in Ausgangssignale überführt werden. Eingesetzt werden bei einer Verhaltensmodellierung u.a. verschiedene Operatoren (`AND`, `<`, `=`), vordefinierte Attribute (`EVENT`), Signalzuweisungen (`q <= ...`) oder `IF-ELSIF-ELSE`-Anweisungen. Selbstverständlich können innerhalb eines VHDL-Modells auch Konstrukte der strukturalen und der Verhaltensmodellierung gleichzeitig verwendet werden.

In den folgenden Abschnitten werden die VHDL-Sprachelemente vorgestellt, die vorrangig der Verhaltensmodellierung dienen.

Die nächsten beiden Abschnitte erläutern zunächst die Details der Anwendung von Operatoren und Attributen. Der "eilige" Leser kann diese beiden Abschnitte, die eher zum Nachschlagen dienen, überspringen.

6.1 Operatoren

Operatoren verknüpfen zwei Operanden bzw. verändern einen Operanden. Das Ergebnis kann wieder als Operand verwendet werden. Werden in einer Anweisung mehrere Operatoren ohne Klammerung eingesetzt, so ist die Priorität der Operatoren bei der Abarbeitung relevant. Bei gleicher Priorität werden die Operatoren in der Reihenfolge ihres Auftretens abgearbeitet. Die Anwendung der Operatoren wird in späteren Beispielen verdeutlicht.

Abb. B-10 gibt die Priorität der einzelnen Operatoren untereinander an:

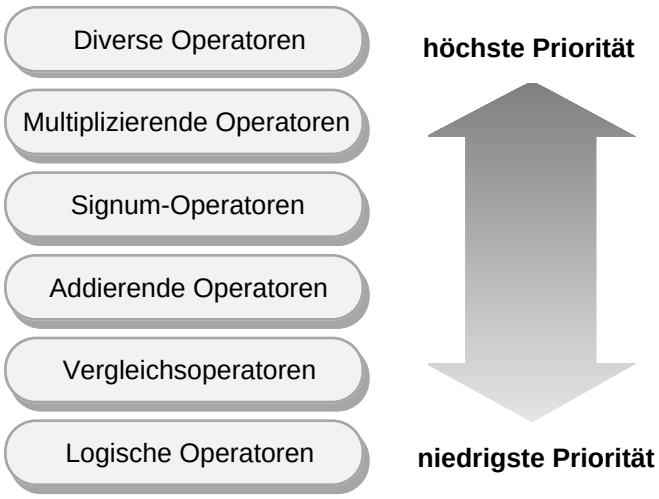

Abb. B-10: Prioritäten von Operatoren

6.1.1 Logische Operatoren

Logische Operatoren wirken auf Einzelobjekte oder Vektoren vom Typ `bit` oder `boolean`; das Ergebnis von logischen Operatoren ist entweder gleich `'1'` bzw. `true` oder gleich `'0'` bzw. `false`. VHDL kennt folgende logische Operatoren:

- `NOT` Negation (nur ein rechtsstehender Operand)
- `AND` UND-Verknüpfung
- `NAND` Negierte UND-Verknüpfung
- `OR` ODER-Verknüpfung
- `NOR` Negierte ODER-Verknüpfung
- `XOR` Exklusiv-ODER-Verknüpfung
- `XNOR` Neg. Exklusiv-ODER-Verknüpfung (nur ✓**93**)

Bei der Anwendung der logischen Operatoren auf Vektoren ist zu beachten, daß die Vektoren gleiche Länge besitzen müssen und daß die Operation elementweise vorgenommen wird. Das Ergebnis erhält den Typ und die Indizierung des linken Operanden.

Sequenzen von mehr als einem Operator sind nur für `AND`, `OR` und `XOR` möglich, da die Reihenfolge der Ausführung hier unerheblich ist. Bei den Operatoren `NAND`, `NOR` und `XNOR` ist dagegen durch Klammerung die Ausführungsreihenfolge festzulegen.

```
a := NOT (x AND y AND z) ;     -- legal: NAND3
b := (x NAND y) NAND z ;       -- legal, aber kein NAND3
c := x NAND y NAND z ;         -- !!! illegal
```

Zu beachten ist weiterhin, daß der logische Operator `NOT` die hohe Priorität der "diversen Operatoren" besitzt.

6.1.2 Vergleichsoperatoren

Operator	Funktion	Typ linker Operand	Typ rechter Operand	Typ Ergebnis
`=`	Prüfung auf Gleichheit der Operanden	alle mögl. Typen außer File-Typen	gleicher Typ wie links	vordef. Typ `boolean`
`/=`	Prüfung auf Ungleichheit der Operanden	alle mögl. Typen außer File-Typen	gleicher Typ wie links	vordef. Typ `boolean`
`<`	Vergleich der beiden Operanden	skal. Typen oder diskr. Vektoren	gleicher Typ wie links	vordef. Typ `boolean`
`<=`	Vergleich der beiden Operanden	skal. Typen oder diskr. Vektoren	gleicher Typ wie links	vordef. Typ `boolean`
`>`	Vergleich der beiden Operanden	skal. Typen oder diskr. Vektoren	gleicher Typ wie links	vordef. Typ `boolean`
`>=`	Vergleich der beiden Operanden	skal. Typen oder diskr. Vektoren	gleicher Typ wie links	vordef. Typ `boolean`

Als diskrete Vektoren werden in der obigen Tabelle die eindimensionalen Feldtypen bezeichnet, die als Elementtyp einen diskreten Typ (ganzzahliger Typ oder Aufzähltyp) besitzen.

Der Gleichheitsoperator für zusammengesetzte Typen (Felder, Records) liefert den Wert `true` zurück, falls jedes Element des linken Operanden ein entsprechendes Element im rechten Operanden besitzt (und umgekehrt) und falls alle entsprechenden Elemente im Wert übereinstimmen. Ansonsten wird der Wert `false` zurückgeliefert.

Entsprechendes gilt für den Ungleichheitsoperator mit vertauschtem Ergebnis.

Beim Arbeiten mit Fließkommazahlen ist darauf zu achten, daß hier die Darstellungs- und Rechengenauigkeit der Hardware in das Ergebnis einfließt. Auf exakte Gleichheit sollte deshalb nie geprüft werden, vielmehr ist die Angabe eines Toleranzbereichs sinnvoll, wie sie etwa in der folgenden Art erfolgen kann:

```
PROCESS
   VARIABLE xyz : real;
BEGIN
   IF (xyz = 0.05) THEN                  -- schlechter Stil
     ...
   END IF;
   ...
   IF ABS(xyz - 0.05) <= 0.000001 THEN   -- besserer Stil
     ...
   END IF;
   ...
END PROCESS;
```

Die Wirkung von Vergleichsoperatoren ist bei den diskreten Vektoren als Operanden folgendermaßen zu verstehen: Die Operation ">" z.B. liefert den Wert `true` zurück, falls gilt:

- der linke Operand ist kein leerer Vektor, der rechte Operand ist leer.
- im anderen Fall (kein leerer rechter Operand) muß gelten:
  - das am weitesten links stehende Element des linken Operanden ist "größer als" das am weitesten links stehende Element des rechten Operanden.
  - falls die beiden am weitesten links stehenden Elemente der beiden Operanden gleich sind, wird der Restvektor in gleicher Weise untersucht.

Die Operation ">=" bedeutet, daß entweder der Fall ">" oder der Fall "=" zutrifft. Ein entsprechend komplementäres Verhalten zeigen die Operatoren "<" und "<=".

6.1.3 Arithmetische Operatoren

6.1.3.1 Addierende Operatoren

Operator	Funktion	Typ linker Operand	Typ rechter Operand	Typ Ergebnis
+	Addition	jeder numerische Typ	gleicher Typ wie links	gleicher Typ wie links
-	Subtraktion	jeder numerische Typ	gleicher Typ wie links	gleicher Typ wie links
&	Zusammenbinden	jeder Vektortyp	gleicher Typ von Vektor wie links	gleicher Typ von Vektor wie links
		jeder Vektortyp	Element vom Typ wie Vektor links	gleicher Typ von Vektor wie links
		Element vom Typ wie Vektor rechts	jeder Vektortyp	gleicher Typ von Vektor wie rechts
		jeder Elementtyp	gleicher Typ wie links	gleicher Typ von Vektor wie Element links

Für den "&"-Operator, das Zusammenbinden von Operanden (im Englischen "concatenation" genannt), gibt es verschiedene Arten von Operandenkombinationen:

- beide Operanden sind Einzelelemente gleichen Typs,
- ein Operand ist ein Vektor (eindimensionales Feld), der andere ein Einzelelement vom Typ der Vektorelemente,
- beide Operanden sind Vektoren gleichen Typs.

Das Ergebnis wird auf jeden Fall ein Vektor mit dem Elementtyp wie die Operanden selbst sein. Die Indizierung des Ergebnisses entspricht der vom linken Operanden fortgeführten Indizierung; Einzelelemente werden dabei als Vektor mit der Länge 1 angesehen.

Die Fortsetzung der Indizierung des linken Vektors kann allerdings zu unerwarteten oder verbotenen Bereichsgrenzen führen. Mit ✓**93** wurde deshalb folgende Vereinbarung getroffen:

- Sind beide Operanden abfallend indiziert (`DOWNTO`), so haben rechter Operand und Ergebnis den gleichen rechten Index,
- sind beide Operanden steigend indiziert (`TO`), so haben linker Operand und Ergebnis den gleichen linken Index.

6.1.3.2 Signum-Operatoren

Signum-Operatoren dienen zur Festlegung des Vorzeichens von Objekten numerischen Typs.

Operator	Funktion	Typ linker Operand	Typ rechter Operand	Typ Ergebnis
+	Identität	-	jeder numerische Typ	gleicher Typ wie Operand
-	Negation	-	jeder numerische Typ	gleicher Typ wie Operand

Die Signum-Operatoren dürfen ungeklammert nicht in Kombination mit multiplizierenden, addierenden und diversen Operatoren stehen (z.B. " `A/(-B)` " anstelle des unerlaubten Ausdrucks " `A/-B` ").

6.1.3.3 Multiplizierende Operatoren

Operator	Funktion	Typ linker Operand	Typ rechter Operand	Typ Ergebnis
`*`	Multiplikation	jeder ganzzahlige Typ	gleicher Typ wie links	gleicher Typ wie links
		jeder Fließkomma-Typ	gleicher Typ wie links	gleicher Typ wie links
		jeder physikalische Typ	vordef. Typ `integer`	gleicher Typ wie links
		jeder physikalische Typ	vordef. Typ `real`	gleicher Typ wie links
		vordef. Typ `integer`	jeder physikalische Typ	gleicher Typ wie rechts
		vordef. Typ `real`	jeder physikalische Typ	gleicher Typ wie rechts
`/`	Division	jeder ganzzahlige Typ	gleicher Typ wie links	gleicher Typ wie links
		jeder Fließkomma-Typ	gleicher Typ wie links	gleicher Typ wie links
		jeder physikalische Typ	vordef. Typ `integer`	gleicher Typ wie links
		jeder physikalische Typ	vordef. Typ `real`	gleicher Typ wie links
		jeder physikalische Typ	gleicher Typ wie links	vordef. Typ `integer`
`MOD`	Modulo-Operator	jeder ganzzahlige Typ	gleicher Typ wie links	gleicher Typ wie links
`REM`	Remainder-Operator	jeder ganzzahlige Typ	gleicher Typ wie links	gleicher Typ wie links

Der Remainder-Operator (`a REM b`) berechnet den Rest bei einer Integerdivision, so daß gilt: `a = (a/b)*b + (a REM b)`

`(a REM b)` hat das Vorzeichen von `a` und einen absoluten Wert, der kleiner als der absolute Wert von `b` ist.

Der Modulo-Operator (`a MOD b`) berechnet den Rest bei einer Integerdivision, so daß gilt: `a = int_value*b + (a MOD b)`

`(a MOD b)` hat das Vorzeichen von `b` und einen absoluten Wert, der kleiner als der absolute Wert von `b` ist.

6.1.3.4 Diverse Operatoren

Die sog. "diversen Operatoren" besitzen die höchste Priorität bei der Abarbeitung. Der Vollständigkeit halber soll hier auch der logische Operator "`NOT`" nochmals erwähnt werden, der aufgrund der Priorität zu dieser Gruppe gehört. Weitere Operatoren dieser Gruppe sind:

Operator	Funktion	Typ linker Operand	Typ rechter Operand	Typ Ergebnis
`**`	Exponentiation	jeder ganzzahlige Typ	vordef. Typ `integer`	gleicher Typ wie links
		jeder Fließkomma-Typ	vordef. Typ `integer`	gleicher Typ wie links
ABS	Absolutwertbildung	-	jeder numerische Typ	gleicher Typ wie Operand

6.1.4 Schiebe- und Rotieroperatoren ✓93

Mit ✓93 wurden neben dem negierten Exklusiv-ODER weitere Operatoren in den Sprachumfang aufgenommen. Es handelt sich um sechs Schiebe- und Rotieroperatoren, die auf Vektoren angewandt werden. Als linker Operand steht der Vektor selbst, als rechter Operand steht jeweils ein Integerwert, um dessen Wert der Vektorinhalt verschoben bzw. rotiert wird. Die Elemente des Vektors müssen vom Typ `bit` oder `boolean` sein.

Operator	Funktion	Typ linker Operand	Typ rechter Operand	Typ Ergebnis
SLL	Schiebe logisch links	Vektor, Elemente: `bit`, `boolean`	vordef. Typ `integer`	gleicher Typ wie links
SRL	Schiebe logisch rechts	Vektor, Elemente: `bit`, `boolean`	vordef. Typ `integer`	gleicher Typ wie links
SLA	Schiebe arithmetisch links	Vektor, Elemente: `bit`, `boolean`	vordef. Typ `integer`	gleicher Typ wie links
SRA	Schiebe arithmetisch rechts	Vektor, Elemente: `bit`, `boolean`	vordef. Typ `integer`	gleicher Typ wie links
ROL	Rotiere links	Vektor, Elemente: `bit`, `boolean`	vordef. Typ `integer`	gleicher Typ wie links
ROR	Rotiere rechts	Vektor, Elemente: `bit`, `boolean`	vordef. Typ `integer`	gleicher Typ wie links

Bei "logischen Schiebeoperationen" um eine Stelle geht jeweils das linke bzw. rechte Vektorelement verloren. Auf der gegenüberliegenden Seite wird der Vektor mit dem Initialisierungswert des Basistyps aufgefüllt. Bei "arithmetischen Schiebeoperationen" wird hierzu das letzte Vektorelement dupliziert. Bei mehrfachen Schiebe- oder Rotieroperationen wird dies entsprechend dem rechten Operanden wiederholt. Negative rechte Operanden entsprechen dem gegensätzlichen Operator mit dem Absolutwert des rechten Operanden ("`b ROL -4`" entspricht "`b ROR 4`").

Abb. B-11 verdeutlicht diese Operationen:

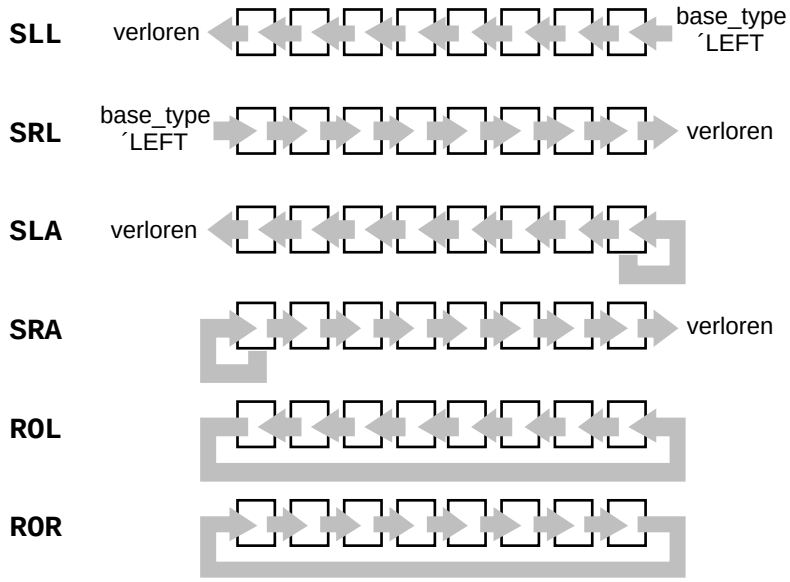

Abb. B-11: Schiebe- und Rotieroperatoren von ✓93

6.2 Attribute

Im folgenden sind die vordefinierten Attribute mit ihrer Funktion aufgelistet. Sie werden nach dem Prefix unterschieden, auf das sie angewandt werden.

6.2.1 Typbezogene Attribute

Typbezogene Attribute liefern Informationen zu diskreten Datentypen, wie z.B. Werte von bestimmten Elementen bei Aufzähltypen.

Name	Funktion
`t'BASE`	liefert den Basistyp des Prefixtyps `t` (nur in Verbindung mit nachfolgendem weiteren Attribut möglich)
`t'LEFT`	liefert die linke Grenze des Prefixtyps `t`
`t'RIGHT`	liefert die rechte Grenze des Prefixtyps `t`
`t'HIGH`	liefert die obere Grenze des Prefixtyps `t`
`t'LOW`	liefert die untere Grenze des Prefixtyps `t`
`t'POS(x)`	liefert die Position (`integer`-Index) des Elements `x` im Prefixtyp `t`
`t'VAL(y)`	liefert den Wert des Elements an Position `y` im Prefixtyp `t` (`y` ist `integer`)
`t'SUCC(x)`	liefert den Nachfolger von `x` im Prefixtyp `t`
`t'PRED(x)`	liefert den Vorgänger von `x` im Prefixtyp `t`
`t'LEFTOF(x)`	liefert das Element links von `x` im Prefixtyp `t`
`t'RIGHTOF(x)`	liefert das Element rechts von `x` im Prefixtyp `t`

Im neuen Standard (✓**93**) wurden einige weitere typbezogene Attribute aufgenommen:

Name	Funktion
`t'ASCENDING`	liefert `true`, falls der Typ `t` eine steigende Indizierung besitzt, ansonsten `false`
`t'IMAGE(x)`	konvertiert den Wert `x` in eine Zeichenkette `t`
`t'VALUE(x)`	konvertiert die Zeichenkette `x` in einen Wert des Typs `t`

Die oben beschriebenen typbezogenen Attribute sollen nun anhand einiger Beispiele erläutert werden:

```
PROCESS
   TYPE    asc_int    IS RANGE 2 TO 8;       -- steigend
   TYPE    desc_int   IS RANGE 8 DOWNTO 2;   -- fallend
   TYPE    colors     IS (white, red, yellow, green, blue);
   SUBTYPE signal_colors IS colors RANGE (red TO green);
   VARIABLE a : integer := 0;
   VARIABLE b : colors  := blue;
   VARIABLE c : boolean := true;
BEGIN
   a := asc_int'LEFT;          -- Ergebnis: 2
   a := asc_int'RIGHT;         -- Ergebnis: 8
   a := asc_int'LOW;           -- Ergebnis: 2
   a := asc_int'HIGH;          -- Ergebnis: 8
   a := desc_int'LEFT;         -- Ergebnis: 8
   a := desc_int'RIGHT;        -- Ergebnis: 2
   a := desc_int'LOW;          -- Ergebnis: 2
   a := desc_int'HIGH;         -- Ergebnis: 8
   a := asc_int'PRED(7);       -- Ergebnis: 6
   a := asc_int'SUCC(4);       -- Ergebnis: 5
   a := asc_int'LEFTOF(7);     -- Ergebnis: 6
   a := asc_int'RIGHTOF(4);    -- Ergebnis: 5
   a := desc_int'PRED(7);      -- Ergebnis: 6
   a := desc_int'SUCC(4);      -- Ergebnis: 5
   a := desc_int'LEFTOF(7);    -- Ergebnis: 8
   a := desc_int'RIGHTOF(4);   -- Ergebnis: 3
   a := asc_int'PRED(2);       -- !! illegal: kein Vorgaenger
   a := asc_int'SUCC(8);       -- !! illegal: kein Nachfolger
   a := asc_int'LEFTOF(15);    -- !! illegal: falscher Index
   b := signal_colors'RIGHT;   -- Ergebnis: green
   b := signal_colors'BASE'RIGHT;     -- Ergebnis: blue
   b := colors'LEFTOF(red);    -- Ergebnis: white
   b := colors'RIGHTOF(red);   -- Ergebnis: yellow
   a := colors'POS(red);       -- Ergebnis: 1 (Aufzaehltypen
                               -- werden von 0 an indiziert)
   b := colors'VAL(2);         -- Ergebnis: yellow
   c := asc_int'ASCENDING;     -- Ergebnis: true  !!! VHDL'93
   c := desc_int'ASCENDING;    -- Ergebnis: false !!! VHDL'93
   b := colors'VALUE("red ");  -- Ergebnis: red   (VHDL'93)
   WAIT;
END PROCESS;
```

Das Ergebnis der Attribute `RIGHT/LEFT` und `RIGHTOF/LEFTOF` hängt also von der Indizierungsrichtung ab. Bei fallender Indizierung sind die Ergebnisse identisch mit den Ergebnissen von `LOW/HIGH`

und PRED/SUCC. Die Aufzähltypen (z.B. colors) besitzen implizit eine steigende Indizierung, so daß beispielsweise RIGHTOF dem Attribut SUCC entspricht.

6.2.2 Feldbezogene Attribute

Bei den feldbezogenen Attributen finden sich viele typbezogene Attribute wieder. Sie werden hier auf eingeschränkte Typen von Feldern und auf konkrete Objekte angewandt.

Name	Funktion
a'LEFT [(n)]	liefert die linke Grenze der n-ten Dimension des Feldes a
a'RIGHT [(n)]	liefert die rechte Grenze der n-ten Dimension des Feldes a
a'LOW [(n)]	liefert die untere Grenze der n-ten Dimension des Feldes a
a'HIGH [(n)]	liefert die obere Grenze der n-ten Dimension des Feldes a
a'LENGTH [(n)]	liefert die Bereichslänge der n-ten Dimension des Feldes a
a'RANGE [(n)]	liefert den Bereich der n-ten Dimension des Feldes a
a'REVERSE _RANGE [(n)]	liefert den Bereich der n-ten Dimension des Feldes a in umgekehrter Reihenfolge

In ✓93 wurde das Attribut ASCENDING ergänzt:

Name	Funktion
a'ASCENDING [(n)]	liefert true, falls das Feld a in der n-ten Dimension eine steigende Indizierung besitzt

Die feldbezogenen Attribute beziehen sich immer auf eine Indizierung. Diese ist bei eindimensionalen Feldern (Vektoren) eindeutig. Die Angabe der Dimension `(n)`, auf die das Attribut angewandt werden soll, ist daher nur bei mehrdimensionalen Feldern relevant. Wird sie weggelassen, so wird das Attribut immer auf die erste Dimension angewandt.

```
PROCESS
   TYPE asc_vec    IS ARRAY (1 TO 5) OF integer;
   TYPE int_matrix IS ARRAY (0 TO 4, 9 DOWNTO 0) OF integer;
   CONSTANT mask        : asc_vec     := (OTHERS => 0);
   VARIABLE ar1         : int_matrix;
   VARIABLE a           : integer := 0;
   VARIABLE b           : boolean;
BEGIN
   a := mask'LEFT;             -- Ergebnis: 1
   a := int_matrix'HIGH(2);    -- Ergebnis: 9
   a := ar1'LEFT(1);           -- Ergebnis: 0
   a := int_matrix'LEFT(1);    -- Ergebnis: 0
   a := ar1'LEFT(2);           -- Ergebnis: 9
   a := ar1'LENGTH(1);         -- Ergebnis: 5
   a := ar1'LENGTH(2);         -- Ergebnis: 10
   b := ar1'ASCENDING(2);      -- VHDL'93, Erg.: false
   b := ar1'ASCENDING(1);      -- VHDL'93, Erg.: true
   WAIT;
END PROCESS;
```

6.2.3 Signalbezogene Attribute

Folgende Attribute werden auf konkrete Objekte der Klasse Signal angewandt:

Name	Funktion
`s'DELAYED [(t)]`	liefert ein auf `s` basierendes Signal, welches um eine Zeit `t` (default: 1 Delta) verzögert ist
`s'STABLE [(t)]`	liefert `true`, falls das Signal `s` eine Zeit `t` (default: 1 Delta) ohne Ereignis war, sonst `false`

`s'QUIET [(t)]`	liefert `true`, falls das Signal `s` eine Zeit `t` (default: 1 Delta) nicht aktiv war, sonst `false`
`s'TRANSACTION`	liefert ein Signal vom Typ `bit`, welches bei jedem Simulationszyklus wechselt, in dem das Signal `s` aktiv ist
`s'EVENT`	liefert `true`, falls beim Signal `s` während des aktuellen Simulationszyklus ein Ereignis auftritt, sonst `false`
`s'ACTIVE`	liefert `true`, falls das Signal `s` während des aktuellen Simulationszyklus aktiv ist, sonst `false`
`s'LAST_EVENT`	liefert die Zeitdifferenz vom aktuellen Simulationszeitpunkt zum letzten Ereignis des Signals `s`
`s'LAST_ACTIVE`	liefert die Zeitdifferenz vom aktuellen Simulationszeitpunkt zum letzten aktiven Zeitpunkt des Signals `s`
`s'LAST_VALUE`	liefert den Wert des Signals `s` vor dem letzten Ereignis

Einige signalbezogene Attribute können mit einem Zeitparameter `(t)` versehen werden, um z.B. beim Attribut `DELAYED` ein um `t` verzögertes Signal zu erhalten. Wird kein Parameter angegeben, so gilt als Defaultwert die minimale Zeit "Delta" (siehe Kapitel 8). Das Signal "`sig1'DELAYED`" ändert z.B. immer ein Delta später seinen Wert als das Signal "`sig1`".

Die Zusammenhänge bei signalbezogenen Attributen sollen in einem kleinen Beispiel verdeutlicht werden. Gegeben sei folgender Signalverlauf:

```
PROCESS
 CONSTANT c : time := 3 ns;
BEGIN
 sig_a <= '1' AFTER  5 ns, '0' AFTER  7 ns, '0' AFTER 12 ns,
          '1' AFTER 17 ns, '0' AFTER 24 ns,
          '1' AFTER 26 ns, '0' AFTER 30 ns, '0' AFTER 34 ns,
          '1' AFTER 38 ns;
 WAIT;
END PROCESS;
```

Die Verläufe der verschiedenen Attribute des Signals `sig_a` über der Zeit werden in Abb. B-12 dargestellt.

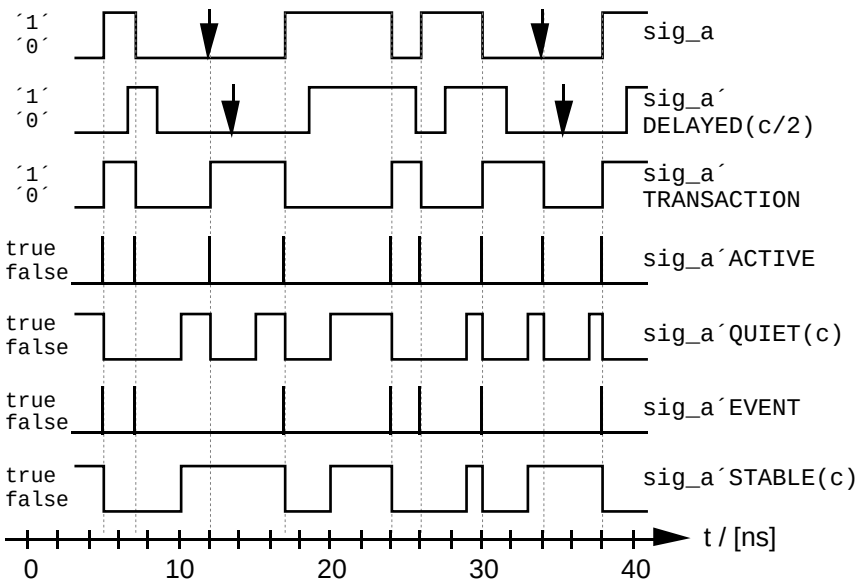

Abb. B-12: Signalbezogene Attribute

Mit ✓**93** sind weitere signalbezogene Attribute verfügbar:

Name	Funktion
`s'DRIVING`	liefert `false`, falls der Treiber des Signals `s` gerade abgeschaltet ("disconnected") ist, sonst `true` (siehe Kapitel 9)
`s'DRIVING_VALUE`	liefert den aktuellen Wert des Treibers für das Signal `s` (siehe Kapitel 9)

Die beiden letztgenannten Attribute können in Prozessen, die dem Signal `s` einen Wert zuweisen, oder in Prozeduren, die `s` als `OUT`-, `BUFFER`- oder `INOUT`-Signal besitzen, verwendet werden. Sie sind sinnvoll, um Signale vor der Behandlung mit Auflösungsfunktionen oder Ports vom Modus `OUT` lesen zu können.

Anwendungen der signalbezogenen Attribute finden sich in den Abschnitten über Signalzuweisungen und den VHDL-Beispielen.

6.2.4 Blockbezogene Attribute ✓**87**

Die folgenden Attribute sind nur in der alten Norm (✓**87**) verfügbar. Mit der Überarbeitung der Norm wurden beide Attribute gestrichen!

Name	Funktion
`b'BEHAVIOR`	liefert `true`, falls der Block (oder die Architektur) `b` eine reine Verhaltensbeschreibung enthält (d.h. falls keine Komponenteninstantiierungen enthalten sind), sonst `false`
`b'STRUCTURE`	liefert `true`, falls der Block (oder die Architektur) `b` eine rein strukturale Beschreibung enthält (d.h. falls keine Signalzuweisungen enthalten sind), sonst `false`

6.2.5 Allgemeine Attribute ✓93

In ✓93 sind auch Attribute verfügbar, die den Namen oder den Pfad von Objekten in der Hierarchie eines VHDL-Modells ermitteln. Dies kann zur Fehlersuche im Zusammenspiel mit der ASSERT-Anweisung nützlich sein.

Diese allgemeinen Attribute lassen sich nicht nur auf Objekte anwenden, sondern meistens auch auf alle VHDL-Einheiten (vgl. Gruppen), die einen Namen besitzen:

Name	Funktion
e'SIMPLE_NAME	liefert den Namen der Einheit e als Zeichenkette (string) in Kleinbuchstaben
e'PATH_NAME	liefert den Pfad der Einheit e innerhalb des Modells als Zeichenkette in Kleinbuchstaben.
e'INSTANCE_NAME	wie PATH_NAME, zusätzlich mit Informationen über Konfigurationen bei Komponenten

