6.3 Signalzuweisungen und Verzögerungsmodelle

Die wohl wichtigste Anweisung in VHDL ist die Zuweisung von neuen Werten an Signale. Signale dienen als Informationsträger innerhalb eines VHDL-Modells und zur Kommunikation mit dessen Umwelt.

6.3.1 Syntax

Signalzuweisungen können nebenläufig sein oder als sequentielle Anweisungen innerhalb von Prozessen, Funktionen oder Prozeduren stehen.

Das Ziel der Zuweisung (`sig_name`) kann ein einzelnes Signal oder ein Teil eines Vektors (slice), bestehend aus mehreren Signalen sein. Als zuzuweisendes Argument (*`value_expr`*) kann bei Signalzuweisungen wieder ein **Signal** gleichen Typs oder ein beliebiger **Ausdruck**, der einen Signal typkonformen Wert liefert, stehen. Der neue Signalwert kann einerseits nur um ein Delta verzögert zugewiesen werden, indem der optionale `AFTER`-Teil der Signalzuweisung weggelassen wird oder indem eine Null-Verzögerung angegeben wird ("`AFTER 0 ns`"). Andererseits ist die explizite Angabe einer Verzögerungszeit über das Schlüsselwort `AFTER` und eine nachfolgende Zeitangabe (*`time_expr`*) möglich. In VHDL stehen dazu zwei verschiedene Verzögerungsmodelle zur Verfügung, die hier erläutert werden sollen.

Die Grundsyntax für Signalzuweisungen lautet:

```
sig_name <= [TRANSPORT]
            value_expr_1 [AFTER time_expr_1]
          {, value_expr_n  AFTER time_expr_n } ;
```

Das Schlüsselwort `TRANSPORT` dient zur Kennzeichnung des "Transport"-Verzögerungsmodells. Falls das Schlüsselwort fehlt, wird das "Inertial"-Verzögerungsmodell angewandt. Beiden Modellen gemein ist der sog. "Preemption-Mechanismus".

6.3.2 Ereignisliste

Die Auswirkungen dieses Mechanismus und die Verzögerungsmodelle können am besten durch eine graphische Darstellung der Ereignisliste erläutert werden.

Die Ereignisliste ("waveform" oder "event queue") enthält alle zukünftigen Signalwechsel in einem Simulationslauf. Signalwechsel, die mit Verzögerungszeiten behaftet sind, werden an die entsprechende Stelle in dieser Liste eingetragen. Bei der Simulation wird diese Ereignisliste abgearbeitet, d.h. die dort vermerkten Signalwechsel ausgeführt und deren Auswirkungen berechnet, was in aller Regel neue Einträge in der Ereignisliste bewirkt.

Folgende, zum Zeitnullpunkt durchgeführte Signalzuweisung, entspricht der graphischen Ereignisliste der Abb. B-13.

```
sig_a <= '1' AFTER 2 ns,  '0' AFTER 5 ns,  '1' AFTER 10 ns,
         '1' AFTER 12 ns, '0' AFTER 15 ns, '1' AFTER 17 ns;
```

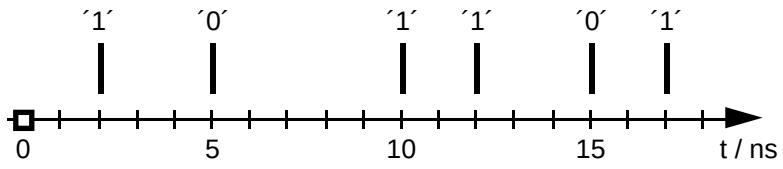

Abb. B-13: Ereignisliste für das Signal sig_a

Es sei an dieser Stelle auf einige Begriffe aus der "VHDL-Welt" hingewiesen, die im Zusammenhang mit den Signalzuweisungen stehen. Die einzelnen Einträge in der Ereignisliste, die jeweils aus einer Zeit- und einer Wertangabe bestehen, werden als **Transaktion** ("transaction") bezeichnet. Ein **Ereignis** ("event") auf einem Signal tritt auf, wenn ein Signal gerade seinen Wert ändert. Dagegen ist ein Signal auch **aktiv** ("active"), falls ihm gerade ein Wert zugewiesen wird, unabhängig davon, ob sich dadurch der Signalwert ändert oder nicht. In Abb. B-13 sind also zu den Zeitpunkten 2, 5, 10, 12, 15 und 17 ns Transaktionen festgelegt. Ein Ereignis auf dem Signal würde zum Zeitpunkt 2, 5, 10, 15 und 17 ns auftreten, sofern sich die Ereignisliste zwischenzeitlich

nicht ändert. Aktiv wäre das Signal dagegen an allen Ereigniszeitpunkten und zusätzlich bei 12 ns.

6.3.3 Preemption-Mechanismus

"Preemption" bezeichnet das Entfernen von geplanten Transaktionen aus der Ereignisliste. Der Premption-Mechanismus wird bei jeder neuen Signalzuweisung angewandt. Welche Transaktionen dabei gelöscht werden, hängt vom bei der Signalzuweisung verwendeten Verzögerungsmodell ab.

Für VHDL-Simulatoren ist der Preemption-Mechanismus durch die VHDL-Norm vorgeschrieben. Herkömmliche Digitalsimulatoren arbeiten oft nicht nach diesem Mechanismus.

6.3.4 "Transport"-Verzögerungsmodell

Bei diesem Verzögerungsmodell von VHDL werden alle Transaktionen gelöscht, die nach einem neuen Signalwert oder gleichzeitig mit diesem auftreten. Es führt dazu, daß eine Signalzuweisung "`sig_a <= TRANSPORT '1' AFTER 11 ns;`" die zum Zeitpunkt 2 ns durchgeführt wird, alle diesem neuen Eintrag bei 13 ns nachfolgenden oder zur gleichen Zeit stattfindenden Einträge aus der Ereignisliste löscht. Es ergibt sich damit eine neue Ereignisliste für das Signal `sig_a`:

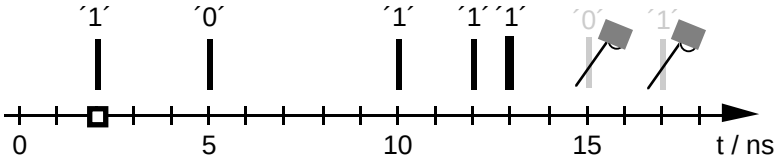

Abb. B-14: Neue Ereignisliste für das Signal sig_a nach dem "Transport"-Verzögerungsmodell

6.3.5 "Inertial"-Verzögerungsmodell

Beim "Inertial"-Verzögerungsmodell wird zusätzlich zum "Transport"-Verzögerungsmodell folgende Regel angewandt:

① Markiere die unmittelbar vor dem neuen Eintrag stattfindende Transaktion, falls sie den gleichen Wert besitzt.

② Markiere die aktuelle und die neue Transaktion.

③ Lösche alle nicht markierten Transaktionen.

Diese Vorgehensweise führt dazu, daß nur diejenigen Impulse, die eine längere (oder gleichlange) Dauer als die angegebene Verzögerungszeit besitzen, auch tatsächlich auftreten. Das Inertial-Verzögerungsmodell ist immer dann gültig, wenn in der Signalzuweisung nicht das Schlüsselwort `TRANSPORT` auftritt.

Eine Zuweisung "`sig_a <= '1' AFTER 11 ns;`" die zum Zeitpunkt 2 ns durchgeführt wird, führt aufgrund des Inertial-Modells auf folgende Ereignisliste für das Signal `sig_a`:

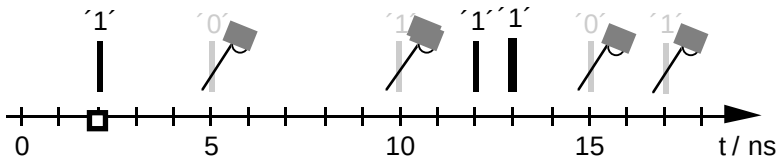

Abb. B-15: Neue Ereignisliste für das Signal sig_a nach dem "Inertial"-Verzögerungsmodell

6.3.6 "Reject-Inertial"-Verzögerungsmodell ✓93

Bei der Zuweisung "`sig_b <= sig_c AFTER 3 ns;`" wird das "Inertial"-Verzögerungsmodell angewandt. Eine wesentliche Konsequenz ist, daß jeder Impuls des Signals `sig_c`, der kürzer als die angegebene Verzögerungszeit von 3 ns ist, unterdrückt ("rejected") wird.

Dies ist bei manchen Komponenten sicherlich sinnvoll, die kurze Impulse ("Spikes") unterdrücken oder herausfiltern. Oft entspricht der Grenzwert dieser Impulsdauer aber nicht der Verzögerungszeit der Komponente. In ✓93 wurde deshalb ein drittes Verzögerungsmodell

eingeführt, das eine von der Verzögerungszeit unabhängige Zeitangabe für eine minimal übertragene Impulsdauer (*rej_time_expr*) gestattet. Es hat folgende Syntax:

```
sig_name <= [[REJECT rej_time_expr] INERTIAL]
          value_expr_1 [AFTER time_expr_1]
       {, value_expr_n  AFTER time_expr_n  } ;
```

Um eine deutlichere Kennzeichnung des (Default-) "Inertial"-Verzögerungsmodells zu erzielen, ist das Schlüsselwort `INERTIAL` nun auch ohne `REJECT` optional. Die Obergrenze für die Impulsdauer (*rej_time_expr*) darf bei diesem Verzögerungsmodell nicht größer als die erste Verzögerungszeit (*time_expr_1*) sein.

Folgendes Beispiel zeigt die Auswirkungen der drei Verzögerungsmodelle anhand von verschieden breiten Impulsen des Signals `sig_s`:

```
sig_s <= TRANSPORT '1' AFTER  1 ns, '0' AFTER  5 ns,
                   '1' AFTER 10 ns, '0' AFTER 13 ns,
                   '1' AFTER 18 ns, '0' AFTER 20 ns,
                   '1' AFTER 25 ns, '0' AFTER 26 ns;

sig_t <= TRANSPORT            sig_s AFTER 3 ns;
sig_i <=                      sig_s AFTER 3 ns;
sig_r <= REJECT 2 ns INERTIAL sig_s AFTER 3 ns; -- ! VHDL'93
```

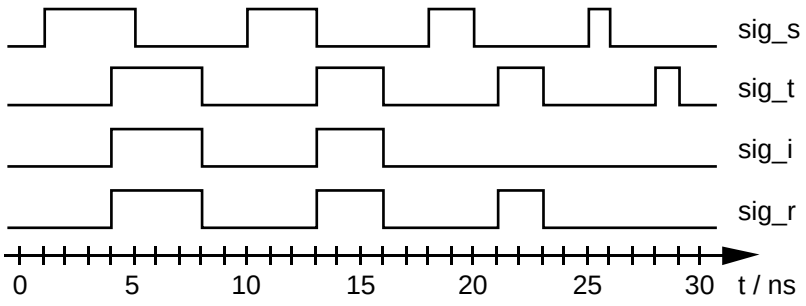

Abb. B-16: Signalverläufe bei unterschiedlichen Verzögerungsmodellen

Abb. B-16 zeigt folgende Effekte: Bei Anwendung des "Transport"-Verzögerungsmodells (`sig_t`) wird das Quellsignal (`sig_s`) unverändert in seiner Form um 3 ns verzögert. Beim "Inertial"-Verzögerungsmodell (`sig_i`) werden Impulse, die kürzer als die Verzögerungszeit sind, unterdrückt. Das "Reject-Inertial"-Modell schließlich unterdrückt nur Impulse, die kürzer als die nach dem Schlüsselwort `REJECT` spezifizierte Zeit von 2 ns sind.

Weitere, interessante Effekte treten auf, wenn unterschiedliche Verzögerungszeiten für steigende und fallende Signalflanken spezifiziert werden. Diesen Fall beleuchtet folgendes Beispiel:

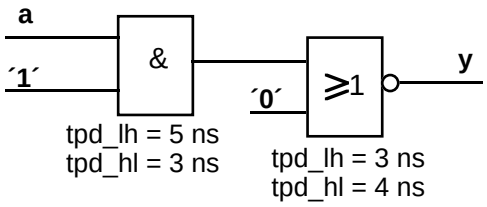

Abb. B-17: Flankenabhängige Laufzeiten

Wird für beide Gatter das "Transport"-Verzögerungsmodell verwendet, so passiert (durch den "Preemption"-Mechanismus) kein positiver Impuls des Signals `a`, der kürzer als 2 ns ist, das AND-Gatter. Impulse, die länger als 2 ns sind, werden um 2 ns verkürzt. Das zweite Gatter invertiert den positiven Impuls und verkürzt ihn wiederum um 1 ns. Bei der gegebenen Beschaltung gelangen also positive Impulse am Eingang `a` nur an den Ausgang `y`, wenn ihre Dauer größer als 3 ns ist.

Noch komplizierter wird der Fall, wenn das "Inertial"-Verzögerungsmodell verwendet wird. Positive Impulse an `a` werden um 2 ns verkürzt und erst ab 5 ns Dauer weitergegeben. Das zweite Gatter läßt positive Impulse nur passieren, wenn sie länger als 4 ns sind. Sie werden dabei um 1 ns verkürzt. Der kürzeste, nicht unterdrückte positive Impuls an `a` muß demnach 6 ns lang sein und erscheint am Ausgang mit einer Dauer von nur 3 ns.

Dem Leser wird empfohlen, sich diese Situation mit Hilfe von Ereignislisten zu verdeutlichen. Außerdem kann untersucht werden, wie sich der Ausgang bei negativen Impulsen am Eingang `a` verhält.

6.4 Nebenläufige Anweisungen

Nebenläufige Anweisungen enthalten optional innerhalb der Anweisung ein sog. Label. Dieses Label kann auch als Name der Anweisung interpretiert werden. Er dient zum späteren Referenzieren der spezifischen Anweisung (z.B. bei der Konfiguration eines strukturalen Modells) oder auch für benutzerdefinierte Attribute.

Sequentielle Anweisungen dürfen hingegen nach ✓**87** im allgemeinen keine Labels tragen. Erst mit ✓**93** können Labels für alle Anweisungen (nebenläufig und sequentiell) vergeben werden.

6.4.1 Signalzuweisungen (normal und bedingt)

Im Falle von nebenläufigen Signalzuweisungen lautet die Syntax der nicht-bedingter Zuweisung:

```
[assignment_label :] sig_name <= [TRANSPORT]
         value_expr_1 [AFTER time_expr_1]
      { , value_expr_n  AFTER time_expr_n } ;
```

Werden bei der Signalzuweisung mehrere Werte angegeben, so müssen diese in zeitlich aufsteigender Reihenfolge angeordnet sein.

Die erste Alternative **bedingter Signalzuweisungen** ("conditional signal assignment") basiert auf mehreren Zuweisungsalternativen, die jeweils durch Bedingungen (*condition*) gesteuert werden. Dies entspricht also einer sequentiellen IF-ELSIF-ELSE-Struktur. Die Syntax des "conditional signal assignment" hat folgendes Aussehen:

```
[assignment_label :] sig_name <= [TRANSPORT]
  {    value_expr_m [AFTER time_expr_m]
   {,  value_expr_n  AFTER time_expr_n }
                       WHEN condition_m ELSE }
       value_expr_o [AFTER time_expr_o]
   {,  value_expr_p  AFTER time_expr_p };
```

Die zweite Alternative ("selected signal assignment") entspricht in ihrem Verhalten der sequentiellen CASE-Anweisung. Sie basiert auf bestimmten Alternativen (*choice*) eines Ausdruckes (*expression*) und hat folgendes Aussehen:

```
[assignment_label :] WITH expression SELECT
sig_name <= [TRANSPORT]
  {    value_expr_m [AFTER time_expr_m]
   {,  value_expr_n  AFTER time_expr_n}
                          WHEN choice_m , }
       value_expr_o [AFTER time_expr_o]
   {,  value_expr_p  AFTER time_expr_p}
                          WHEN choice_o ;
```

Werden mit den einzelnen Auswahlalternativen nicht alle möglichen Werte von *expression* abgefragt, so muß an Stelle von *choice_o* das Schlüsselwort OTHERS gesetzt werden, um die nicht explizit abgefragten Werte zu erfassen.

```
ARCHITECTURE behavioral OF signals IS
  SIGNAL sig_a, sig_b, sig_c : std_ulogic;
BEGIN
-- nebenlaeufige Signalzuweisung mit TRANSPORT -------------
  sig_a <= TRANSPORT '0', '1' AFTER 2 ns, 'Z' AFTER 3 ns;

-- !!! illegal: Werte nicht zeitlich aufsteigend geordnet --
  sig_a <= TRANSPORT '0', 'Z' AFTER 3 ns, '1' AFTER 2 ns;

-- "conditional signal assignment" (csa) -------------------
  csa: sig_b <= '1', '0' AFTER 2 ns WHEN sel = 1 ELSE
                '0', '1' AFTER 3 ns WHEN sel = 2 ELSE
                'Z';

-- dem csa entsprechendes "selected signal assignment" (ssa)
  ssa: WITH sel SELECT
       sig_c <= '1', '0' AFTER 2 ns WHEN 1,
                '0', '1' AFTER 3 ns WHEN 2,
                'Z' WHEN OTHERS;
END behavioral;
```

In vielen Fällen von nebenläufigen, bedingten Signalzuweisungen ist bei bestimmten Bedingungen kein Signalwechsel erforderlich. Die einzige Möglichkeit, dies mit der alten VHDL-Norm zu realisieren, liegt in der Zuweisung des Signals an sich selbst, wie es das Beispiel eines einfachen taktpegelgesteuerten Speicherbausteins (Latch) zeigen soll:

```
ENTITY latch IS
  PORT (d, clk : IN bit; q : BUFFER bit) ;
END latch ;

ARCHITECTURE concurrent_1 OF latch IS
BEGIN
  q <= d WHEN clk = '1' ELSE q ;     -- csa-Alternative
END concurrent_1 ;

ARCHITECTURE concurrent_2 OF latch IS
BEGIN
  WITH clk SELECT                    -- ssa-Alternative
    q <= d WHEN '1' ,
         q WHEN OTHERS ;
END concurrent_2 ;
```

Diese Realisierungen haben jedoch beide den Nachteil, daß bei negativen Taktflanken unnötige Transaktionen des Signals erzeugt werden und durch den "Preemption"-Mechanismus möglicherweise Informationen verloren gehen.

Eine saubere Lösung des Problems bietet ✓**93** mit dem Schlüsselwort UNAFFECTED. Es kann bei den bedingten Signalzuweisungen eingesetzt werden und erzeugt keine Transaktion auf dem Signal:

```
ARCHITECTURE concurrent_3 OF latch IS    -- !!! nur VHDL'93
BEGIN
   q <= d WHEN clk = '1'                 -- csa
          ELSE UNAFFECTED ;
END concurrent_3 ;

ARCHITECTURE concurrent_4 OF latch IS    -- !!! nur VHDL'93
BEGIN
   WITH clk SELECT
      q <= d          WHEN '1',          -- ssa
           UNAFFECTED WHEN OTHERS ;
END concurrent_4 ;
```

6.4.2 Assertions

Assertions dienen zur Überprüfung von Bedingungen und zur Ausgabe von Warnungen bzw. Fehlermeldungen. Die Syntax lautet:

```
[assert_label :] ASSERT condition
                 [ REPORT "message_string"]
                 [ SEVERITY severity_level] ;
```

Diese Syntax wird folgendermaßen interpretiert:

> "Überprüfe, ob die Bedingung *condition* erfüllt ist; falls nicht, erzeuge die Meldung "message_string" und breche, abhängig von der Fehlerklasse severity_level, gegebenenfalls die Simulation ab."

Eine Fehlermeldung mit evtl. weiteren Konsequenzen tritt also nur auf, falls die angegebene Bedingung (*condition*) den Wert false ergibt.

Ohne Angabe der Fehlermeldung wird der String "Assertion violation." ausgegeben.

Die vier möglichen Fehlerklassen (entsprechend dem vordefinierten Aufzähltyp severity_level) haben folgende Bedeutung:

- note dient zur Ausgabe von allgemeinen Informationen,
- warning dient zur Anzeige von möglichen unerwünschten Bedingungen,
- error zeigt an, daß eine Aufgabe mit dem falschen Ergebnis abgeschlossen wurde,
- failure zeigt an, daß eine Aufgabe nicht abgeschlossen werden konnte.

Wird in der Anweisung keine Fehlerklasse angegeben, so wird sie mit der Klasse error versehen. Die Entscheidung, ab welcher Klasse die Simulation abgebrochen wird, legt man i.d.R. durch eine spezifische Simulatoreinstellung fest.

Zwei Beispiele zur Anzeige eines (low-aktiven) Resetsignals und zur Prüfung auf definierten Pegel eines Signals sig_a vom Typ std_ulogic lauten:

```
reset_check : ASSERT sig_reset /= '0'
                REPORT "Achtung: Reset ist aktiv !"
                SEVERITY note ;
```

```
ASSERT (now = 0 fs) OR (sig_a /= 'U')
  REPORT "sig_a ist nicht initialisiert !" ;
```

Im zweiten Beispiel wird die Ausgabe einer Fehlermeldung zum Zeitnullpunkt unterdrückt.

6.4.3 Prozesse

Prozesse dienen als Umgebung für sequentielle, d.h. nacheinander ablaufende Befehle. Sie werden also zur Modellierung prozeduraler Vorgänge verwendet. Die Prozesse selbst gelten als nebenläufige Anweisung, d.h. existieren mehrere Prozesse innerhalb einer Architektur, so können sie gleichzeitig aktiv sein. Prozesse werden durch zwei verschiedene Möglichkeiten aktiviert und gestoppt, die sich gegenseitig ausschließen:

- Durch eine **Liste sensitiver Signale** im Prozeß-Kopf:
  Prozesse dieser Art werden einmalig bei der Modell-Initialisierung komplett durchlaufen und zu späteren Zeitpunkten erst wieder aktiviert, wenn sich eines der Signale der "sensitivity list" ändert. Dann wird der Prozeß wieder bis zum Ende abgearbeitet, usw.
- Durch `WAIT`-**Anweisungen**:
  Bei der Modell-Initialisierung zum Zeitnullpunkt wird der Prozeß bis zur ersten `WAIT`-Anweisung abgearbeitet und erst wieder aktiviert, wenn die Bedingung der `WAIT`-Anweisung erfüllt ist oder die dort angegebene Zeit verstrichen ist (vgl. `WAIT`-Anweisung).

Prozesse ohne `WAIT`-Anweisung und ohne "sensitivity list" sind üblicherweise nicht sinnvoll, da diese Prozesse beim Simulationsstart aufgerufen und dann ständig zyklisch durchlaufen werden ("Endlosschleife").

Prozesse bestehen aus einem Deklarations- und einem Anweisungsteil. Die Syntax der beiden Varianten lautet:

```
[process_label :] PROCESS (sig_1 {, sig_n})
   ...
   ... -- Deklaration von: Typen und Unter-
   ... -- typen, Aliases, Konstanten, Files,
   ... -- Variablen, Unterprogrammen
   ... -- Definition von: Unterprogrammen,
   ... -- Attributen
   ... -- USE-Anweisungen
   ...
BEGIN
   ...
   ... -- sequentielle Anweisungen ohne WAIT
   ...
END PROCESS [process_label] ;

[process_label :] PROCESS
   ...
   ... -- Deklarationsteil wie oben
   ...
BEGIN
   ...
   ... -- sequentielle Anweisungen
   ...
   WAIT ... ; -- mind. eine WAIT-Anweisung
   ...
   ... -- sequentielle Anweisungen
   ...
END PROCESS [process_label] ;
```

Während (herkömmliche) Variablen nur innerhalb eines Prozesses gültig und außerhalb nicht sichtbar sind, können Signale auch außerhalb eines Prozesses gelesen und mit neuem Wert versehen, d.h. beschrieben werden. Sie können somit zur Kommunikation (Austausch von Informationen, gegenseitiges Aktivieren usw.) zwischen Prozessen verwendet werden.

Ein wesentlicher Aspekt im Zusammenhang mit Prozessen ist das Zeitverhalten, insbesondere der zeitliche Unterschied zwischen Variablen-

und Signalzuweisungen innerhalb eines Prozesses. Während Variablenwerte sofort bei der Abarbeitung der Anweisung zugewiesen werden, werden die neuen Werte von Signalen zunächst vorgemerkt und erst nach Abarbeitung aller aktiven Prozesse am Ende eines sog. Delta-Zyklus zugewiesen. Diese Problematik wird getrennt in Kapitel 8 behandelt.

Da sequentielle Anweisungen an dieser Stelle noch nicht behandelt wurden, wird für ausführliche Beispiele zu Prozessen auf die nachfolgenden Abschnitte verwiesen. An dieser Stelle wird deshalb nur ein kurzes Beispiel für ein D-Latch aufgeführt, das eine selbsterklärende `IF`-Struktur enthält:

```
ARCHITECTURE sequential_1 OF latch IS
BEGIN
-- Aktivierung des Prozesses durch Ereignisse auf d oder clk
   q_assignment: PROCESS (d, clk)
   BEGIN
      IF clk = '1' THEN
         q <= d ;
      END IF ;
   END PROCESS q_assignment ;
END sequential_1 ;
```

Die Vereinheitlichung der Rahmensyntax in ✓**93** führt zu einer optionalen Angabe des Schlüsselwortes `IS` :

```
[process_label :]
              PROCESS [(sig_1 {, sig_n})] [IS]
   ...
BEGIN
   ...
END PROCESS [process_label] ;
```

6.5 Sequentielle Anweisungen

Wie bereits erwähnt, kann mit der Überarbeitung der VHDL-Norm (✓**93**) jeder Anweisung, also auch den sequentiellen Anweisungen, ein Label zugeteilt werden.

6.5.1 Signalzuweisungen

Die Syntax für sequentielle Signalzuweisungen lautet:

```
sig_name <= [TRANSPORT]
          value_expr_1 [AFTER  time_expr_1]
      { ,  value_expr_n  AFTER  time_expr_n } ;
```

Die einzelnen Signalwerte müssen dabei in zeitlich aufsteigender Reihenfolge angeordnet sein.

Bedingte Signalzuweisungen sind im sequentiellen Fall nicht erlaubt. Sie können aber durch `IF-ELSIF-ELSE`- oder `CASE`-Anweisungen abgebildet werden.

Die Verzögerungsmodelle sind wie bei den nebenläufigen Zuweisungen zu verwenden ("Inertial" als Defaultmodell, "Transport" durch explizite Kennzeichnung mit dem Schlüsselwort `TRANSPORT`). Das "Reject-Inertial"-Modell (✓**93**) ist entsprechend der ebenfalls oben beschriebenen Syntax anzuwenden.

6.5.2 Variablenzuweisungen

Die innerhalb von Prozessen und Unterprogrammen verwendeten Variablen (zeitabhängige Objekte ohne Aufzeichnung des Verlaufs über der Zeit) können nur unverzögert zugewiesen werden. Sie eignen sich z.B. zur Speicherung von Zwischenergebnissen. Wenn möglich, sollten Variablen anstelle von Signalen verwendet werden, da sie bei der Simulation weniger Verwaltungsaufwand (Rechenzeit, Speicherplatz) als Signale erfordern.

Die Syntax für Variablenzuweisungen lautet:

```
var_name := value_expr ;
```

Als neuer Variablenwert (*value_expr*) kann entweder der Wert eines anderen Objektes (Signal, Variable, Konstante), ein expliziter Wert oder ein Ausdruck angegeben werden.

Man beachte den Unterschied zum Signalzuweisungsoperator (":=" anstelle von "<="). Ein weiterer Unterschied zur Signalzuweisung liegt

im Ausführungszeitpunkt: während Signalzuweisungen erst am Ende eines Delta-Zyklus nach Ausführung aller aktiven Prozesse durchgeführt werden, werden Variablen unmittelbar, d.h. bei Erreichen der entsprechenden Anweisung im sequentiellen Ablauf zugewiesen. Die Konsequenzen aus diesem Sachverhalt und Beispiele hierzu werden im Kapitel über den Simulationsablauf (Kapitel 8) aufgezeigt.

Das folgende Beispiel illustriert die Verwendung von sequentiellen Signal- und Variablenzuweisungen:

```
ENTITY mult IS
  PORT (a, b : IN integer := 0; y : OUT integer) ;
END mult ;

ARCHITECTURE number_one OF mult IS
BEGIN
  PROCESS (a,b) -- Aktivierung durch Ereignisse auf a oder b
    VARIABLE v1, v2 : integer := 0 ;
  BEGIN
    v1 := 3 * a + 7 * b ;      -- sequent. Variablenzuweisung
    v2 := a * b + 5 * v1 ;     -- sequent. Variablenzuweisung
    y <= v1 + v2 ;             -- sequent. Signalzuweisung
  END PROCESS ;
END number_one ;
```

6.5.3 Assertions

Die Ausgabe von Fehlermeldungen ist ebenfalls als sequentielle Anweisung möglich. Hier besteht zur Syntax der nebenläufigen Form nur ein Unterschied in der alten VHDL-Norm: Es dürfen keine Labels verwendet werden.

```
ASSERT condition [ REPORT "message_string"]
                 [ SEVERITY severity_level];
```

Wie bei allen Anweisungen in der überarbeiteten VHDL-Norm ist hier auch im sequentiellen Fall ein Label erlaubt.

Die Verknüpfung von Meldungen mit Assertions innerhalb einer Anweisung hat zur Folge, daß nicht bedingte Meldungen nur mit folgendem Konstrukt ausgegeben werden können:

```
ASSERT false REPORT "Dies ist eine Meldung" SEVERITY note ;
```

Mit der überarbeiteten VHDL-Syntax (✓**93**) kann nun eine Meldung auch ohne Assertion ausgegeben werden. Dazu ist das Schlüsselwort `REPORT` alleine (mit optionaler Fehlerklasse) ausreichend. Defaultwert für die Klasse der Meldung ist hier `note`:

```
[report_label :]  REPORT "message_string"
                  [  SEVERITY severity_level] ;
```

6.5.4 WAIT-Anweisung

`WAIT`-Anweisungen können die Abarbeitung von sequentiellen Anweisungen steuern. Sie dürfen nur in Prozessen ohne "sensitivity-list" und in Prozeduren, die nicht von Prozessen mit "sensitivity-list" aufgerufen werden, auftreten. Als Argumente einer `WAIT`-Anweisung können ein oder mehrere Signale, Bedingungen oder Zeitangaben verwendet werden. Ein "`WAIT;`" ohne jegliches Argument bedeutet "warte für immer" und beendet somit die Ausführung eines Prozesses oder einer Prozedur.

```
WAIT [ON signal_name_1 {, signal_name_n}]
     [UNTIL condition]
     [FOR time_expression] ;
```


Die einzelnen Argumente haben folgende Bedeutung für das Zeitverhalten von Prozessen:

- Eine Liste von Signalen bewirkt, daß solange gewartet wird, bis sich mindestens eines der Signale ändert, d.h. ein Ereignis auftritt. Ein Prozeß mit einer Liste von Signalen als Argument einer am Ende stehenden `WAIT`-Anweisung entspricht somit einem Prozeß mit den gleichen Signalen als Elemente der "sensitivity-list" im Prozeßkopf.

  Ist ein Signal der Liste ein Vektor oder ein höherdimensionales Feld, so erfüllt bereits die Änderung eines einzigen Elementes die `WAIT`-Bedingung.

- Eine Bedingung (*`condition`*) unterbricht die Prozeßabarbeitung solange, bis die Bedingung erfüllt ist.

  Bei Angabe von Bedingung und Signalliste muß die Bedingung erfüllt sein und der Signalwechsel auftreten.

- Die Angabe eines Ausdruckes, der als Ergebnis eine Zeitangabe liefert (*`time_expression`*), stoppt die Prozeßabarbeitung maximal für diese Zeitdauer.

Folgende Beispiele geben weitere Architekturen für das bereits erwähnte Latch wieder:

```
ARCHITECTURE sequential_2 OF latch IS
BEGIN
   q_assignment: PROCESS
   BEGIN
      IF clk = '1' THEN
         q <= d ;
      END IF ;
      WAIT ON d, clk ;     -- entspricht "sensitivity-list"
   END PROCESS q_assignment ;
END sequential_2 ;
```

```
ARCHITECTURE sequential_3 OF latch IS
BEGIN
   q_assignment: PROCESS
   BEGIN
      q <= d ;
      WAIT ON d, clk UNTIL clk = '1' ;    -- ersetzt IF-Anw.
   END PROCESS q_assignment ;
END sequential_3 ;
```

6.5.5 IF-ELSIF-ELSE-Anweisung

Bedingte Verzweigungen in sequentiellen Anweisungsteilen können mit der `IF-ELSIF-ELSE`-Anweisung folgendermaßen realisiert werden:

```
IF condition_1 THEN
   ...
   ...        -- sequentielle Anweisungen
   ...
{ ELSIF condition_n THEN
   ...
   ...        -- sequentielle Anweisungen
   ... }
[ ELSE
   ...
   ...        -- sequentielle Anweisungen
   ... ]
END IF ;
```

Zwingend erforderlich sind nur die erste Bedingung und die Kennzeichnung des Endes der Struktur durch "`END IF;`". Die `ELSIF`- und `ELSE`-Teile sind optional, wobei ersterer mehrfach auftreten kann.

Mit der überarbeiteten Syntax (✓93) kann einer IF-ELSIF-ELSE-Anweisung auch ein Label gegeben werden. Entsprechend kann in der END IF-Anweisung dieses Label wiederholt werden:

```
[if_label :] IF condition_1 THEN
   ...
END IF [if_label] ;
```

6.5.6 CASE-Anweisung

Eine weitere Möglichkeit der bedingten Ausführung bestimmter Anweisungen liegt in der CASE-Anweisung. Sie bietet sich insbesondere bei mehrfacher Verzweigung basierend auf einem Ausdruck an:

```
CASE expression IS
 {   WHEN value_n =>
        ...
        ...  -- sequentielle Anweisungen
        ... }
 [   WHEN OTHERS  =>
        ...
        ...  -- sequentielle Anweisungen
        ... ]
END CASE ;
```

Im Gegensatz zu IF-ELSIF-ELSE müssen hier allerdings alle Werte (*value_n*), die der Ausdruck (*expression*) annehmen kann, explizit angegeben werden. Um die noch nicht behandelten Werte abzufragen, kann auch das Schlüsselwort OTHERS dienen.

Folgende Beispiele einer bedingten Verzweigung sind äquivalent:

```
ENTITY four_byte_rom IS
  PORT (address  : IN  integer RANGE 1 TO 4;
        contents : OUT bit_vector(1 TO 8) ) ;
END four_byte_rom;
```

```
ARCHITECTURE if_variante OF four_byte_rom IS
BEGIN
  PROCESS (address)
  BEGIN
    IF address = 1 THEN
      contents <= ('0','0','0','0','1','1','1','1') ;
    ELSIF address = 2 THEN
      contents <= "00111111" ;
    ELSIF address = 3 THEN
      contents <= b"11111100" ;
    ELSE
      contents <= x"f0" ;
    END IF ;
  END PROCESS ;
END if_variante ;
```

```
ARCHITECTURE case_variante_1 OF four_byte_rom IS
BEGIN
  PROCESS (address)
  BEGIN
    CASE address IS
      WHEN 1 => contents <=
        ('0','0','0','0','1','1','1','1') ;
      WHEN 2 => contents <= "00111111"  ;
      WHEN 3 => contents <= b"11111100" ;
      WHEN OTHERS => contents <= x"f0"  ;
    END CASE ;
  END PROCESS ;
END case_variante_1 ;
```

Beide Varianten verwenden dabei zur weiteren Reduzierung des Code-Umfangs die Möglichkeit, Bit-Vektoren als Strings anzugeben. Anstelle der letzten Alternative ("`WHEN OTHERS`") hätte hier auch "`WHEN 4`" stehen können.

Mit Hilfe von Oder-Verknüpfungen ("`|`") oder Bereichsangaben (`TO`, `DOWNTO`) können mehrere Fälle des Ausdruckes für gleiche Anweisungsteile zusammengefaßt werden. Damit ergibt sich eine weitere Architekturalternative:

```
ARCHITECTURE case_variante_2 OF four_byte_rom IS
BEGIN
  PROCESS (address)
  BEGIN
    CASE address IS
      WHEN 1 | 2  => contents(1 TO 2) <= "00" ;
                     contents(7 TO 8) <= "11" ;
      WHEN OTHERS => contents(1 TO 2) <= "11" ;
                     contents(7 TO 8) <= "00" ;
    END CASE ;
    CASE address IS
      WHEN 2 TO 4 => contents(3 TO 4) <= "11" ;
      WHEN OTHERS => contents(3 TO 4) <= "00" ;
    END CASE ;
    CASE address IS
      WHEN 1 TO 3 => contents(5 TO 6) <= "11" ;
      WHEN OTHERS => contents(5 TO 6) <= "00" ;
    END CASE ;
  END PROCESS ;
END case_variante_2 ;
```

Die einheitliche Handhabung von Labels und Rahmensyntax in der überarbeiteten VHDL-Norm (✓93) erlaubt auch für die CASE-Anweisung folgende Syntaxvariante:

```
[case_label :] CASE expression IS
   ...
END CASE [case_label] ;
```

6.5.7 NULL-Anweisung

Die NULL-Anweisung:

```
NULL ;
```

führt keine Aktion aus. Sie dient zur expliziten Kennzeichnung von aktionslosen Fällen in IF- und vor allem in CASE-Anweisungen.

Die Modellierung des Latches mit Hilfe einer CASE-Anweisung könnte somit folgendes Aussehen haben:

```
ARCHITECTURE sequential_4 OF latch IS
BEGIN
   q_assignment: PROCESS (clk, d)
   BEGIN
      CASE clk IS
         WHEN '1'     => q <= d ;
         WHEN OTHERS  => NULL ;
      END CASE ;
   END PROCESS q_assignment ;
END sequential_4 ;
```

6.5.8 LOOP-Anweisung

Iterationsschleifen, d.h. mehrfach zu durchlaufende Anweisungsblökke, können mittels der LOOP-Anweisung realisiert werden. Dabei existieren die folgenden drei Alternativen: FOR-Schleife, WHILE-Schleife und Endlosschleife:

```
[loop_label :] FOR range LOOP
   ...
   ...     -- sequentielle Anweisungen
   ...
END LOOP [loop_label] ;

[loop_label :] WHILE condition LOOP
   ...
   ...     -- sequentielle Anweisungen
   ...
END LOOP [loop_label] ;

[loop_label :] LOOP
   ...
   ...     -- sequentielle Anweisungen
   ...
END LOOP [loop_label] ;
```

Die Schleifensteuerstrukturen *range* und *condition* werden wie bei der GENERATE-Anweisung verwendet. Für die Angabe von Bereichen (*range*) wird implizit eine Laufvariable deklariert.

6.5.9 EXIT- und NEXT-Anweisung

EXIT- und NEXT-Anweisungen dienen zum vorzeitigen Ausstieg aus Schleifenanweisungen bzw. zum vorzeitigen Abbruch des aktuellen Durchlaufs. Mit NEXT wird direkt zum Beginn des nächsten Schleifendurchlaufes gesprungen, mit EXIT wird die Schleife ganz verlassen. Da Schleifen, wie IF- und CASE-Anweisungen, auch verschachtelt aufgebaut sein können, kann mit EXIT und NEXT auch aus hierarchisch höher liegenden Schleifen ausgestiegen werden. Dazu muß das Label der entsprechenden Schleife angegeben werden:

```
NEXT [loop_label] [WHEN condition] ;
EXIT [loop_label] [WHEN condition] ;
```

Im folgenden werden zwei Beispiele für Modelle mit Schleifen gezeigt. Das Modell array_compare vergleicht zwei 9x9-Matrizen miteinander. Der Ausgang equal wird true, falls alle Elemente der Matrizen übereinstimmen.

```
PACKAGE array_pack IS
  TYPE bit_matrix IS ARRAY
    (integer RANGE <>, integer RANGE <>) OF bit ;
END array_pack ;
```

```
ENTITY array_compare IS
  PORT (a, b  : IN
        work.array_pack.bit_matrix(8 DOWNTO 0, 8 DOWNTO 0) ;
        equal : OUT boolean) ;
END array_compare ;
```

```
ARCHITECTURE behavioral OF array_compare IS
BEGIN
  cmp : PROCESS (a,b)
    VARIABLE equ : boolean ;
  BEGIN
    equ := true ;
    first_dim_loop : FOR k IN a'RANGE(1) LOOP
      second_dim_loop : FOR l IN a'RANGE(2) LOOP
        IF a(k,l) /= b(k,l) THEN      -- Elementvergleich
          equ := false ;
-- Ausstieg aus aeusserer Schleife beim ersten, ------------
-- nicht identischen Matrixelement -------------------------
          EXIT first_dim_loop ;
        END IF ;
      END LOOP ;
    END LOOP ;
    equal <= equ ;
  END PROCESS ;
END behavioral ;
```

Das folgende Modell (`n_time_left_shift`) rotiert den Eingangsvektor `in_table` um `n` Stellen nach links und gibt den neuen Vektor über den Port `out_table` aus.

```
ENTITY n_time_left_shift IS
  GENERIC (n : natural := 2) ;
  PORT (in_table  : IN  bit_vector (0 TO 31);
        out_table : OUT bit_vector (0 TO 31)) ;
END n_time_left_shift ;
```

```
ARCHITECTURE behavioral OF n_time_left_shift IS
BEGIN
  PROCESS (in_table)
    VARIABLE count : natural ;
    VARIABLE table : bit_vector (0 TO 32) ;
  BEGIN
    count := 0 ;
    table (0 TO 31) := in_table ;
    n_time_loop : WHILE count < n LOOP
      count := count + 1 ;
      table(table'HIGH) := table(table'LOW) ;
      left_shift_loop :
      FOR j IN table'LOW TO table'HIGH - 1 LOOP
        table(j) := table(j+1) ;
      END LOOP left_shift_loop;
    END LOOP n_time_loop ;
    out_table <= table (0 TO 31) ;
  END PROCESS ;
END behavioral ;
```

6.6 Unterprogramme

Ähnlich wie in höheren Programmiersprachen können in VHDL Unterprogramme in Form von Funktionen oder Prozeduren realisiert werden.

- **Funktionen** werden mit keinem, einem oder mehreren Argumenten aufgerufen und liefern einen Ergebniswert zurück. Der Funktionsaufruf kann an den gleichen Stellen in Ausdrücken und Anweisungen stehen, an denen der Typ des Ergebniswertes erlaubt ist. Funktionen werden explizit durch das Schlüsselwort RETURN unter Angabe des Rückgabewertes verlassen.
- **Prozeduren** werden mit einer Liste von Argumenten aufgerufen, die sowohl Eingaben als auch Ausgaben oder bidirektional sein können. Der Aufruf einer Prozedur ist ein eigenständiger Befehl (sequentiell oder nebenläufig). Prozeduren werden bis zu einer RETURN-Anweisung oder bis zum Ende (entsprechend einem Prozeß mit "sensitivity-list") abgearbeitet.

Funktionen und Prozeduren unterscheiden sich damit in folgenden Punkten:

	Funktionen	Prozeduren
Argumentmodi	`IN`	`IN, OUT, INOUT`
Argumentklassen	Konstanten, Signale	Konstanten, Signale, Variablen
Rückgabewerte	exakt einer	beliebig viele (auch 0)
Aufruf	in typkonformen Ausdrücken und Anweisungen	als eigenständige, sequentielle oder nebenläufige Anweisung
`RETURN`-Anweisung	obligatorisch	optional

Die Beschreibung der Funktionalität eines Unterprogramms kann (zusammen mit der Vereinbarung der Schnittstellen) in den Deklarationsteilen von Entity, Architektur, Block, Prozeß oder im Package Body stehen. Außerdem können Funktionen und Prozeduren selbst wieder in den Deklarationsteilen von anderen Funktionen und Prozeduren spezifiziert werden.

Weiterhin besteht die Möglichkeit, die Funktionalität (Unterprogrammdefinition) und die Schnittstellenbeschreibung (Unterprogrammdeklaration) zu trennen. Die Schnittstellenbeschreibung allein kann dann auch in Packages auftreten.

Eine solche Aufteilung bietet sich unter Ausnutzung der Abhängigkeiten beim Compilieren von Package und Package Body an: Die Schnittstellenbeschreibung wird im Package plaziert, während die Funktionalität erst im Package Body festgelegt wird. Eine nachträgliche Änderung der Funktionalität bedingt dann nur das Neucompilieren des Package Body. Die Design-Einheiten, die das Unterprogramm verwenden, müssen nicht neu übersetzt werden.

6.6.1 Funktionen

Wie bereits erwähnt, dienen Funktionen zur Berechnung eines Wertes und können in Ausdrücken und anderen Anweisungen direkt eingesetzt werden.

Für eine flexiblere Beschreibung (s.o.) bietet sich eine Aufteilung in Funktionsdeklaration (Schnittstellenbeschreibung) und -definition (Funktionalität) an.

6.6.1.1 Funktionsdeklaration

Die Funktionsdeklaration enthält eine Beschreibung der Funktionsargumente und des Funktionsergebnisses:

```
FUNCTION function_name
 [ ( { [arg_class_m] arg_name_m {,arg_name_n}
       : [IN] arg_type_m [:= def_value_m] ;}
       [arg_class_o] arg_name_o {,arg_name_p}
       : [IN] arg_type_o [:= def_value_o] ) ]
  RETURN result_type ;
```

Erlaubt sind also auch Funktionen ohne jegliches Argument.

Der **Funktionsname** (`function_name`) kann auch ein bereits definierter Funktionsname oder Operator sein. In diesem Fall spricht man von "Überladung" der Funktion. Diese Möglichkeit wird in Kapitel 11 ausführlich diskutiert.

Als **Argumentklasse** (`arg_class`) sind Signale und Konstanten erlaubt. Deren Angabe durch die Schlüsselwörter `SIGNAL` oder `CONSTANT` ist optional. Der Defaultwert ist `CONSTANT`.

Als **Argumenttypen** (`arg_type`) können in der Schnittstellenbeschreibung von Unterprogrammen neben eingeschränkten auch uneingeschränkte Typen verwendet werden.

```
FUNCTION demo (SIGNAL data1, data2 : IN integer;
               CONSTANT c1         : real) RETURN boolean;
```

6.6.1.2 Funktionsdefinition

Hier muß die Schnittstellenbeschreibung wiederholt werden. Das nachfolgende Schlüsselwort `IS` kennzeichnet den Beginn der Funktionsbeschreibung:

```
FUNCTION function_name
 [ ( { [arg_class_m] arg_name_m {,arg_name_n}
   : [IN] arg_type_m [:= def_value_m] ;}
   [arg_class_o] arg_name_o {,arg_name_p}
   : [IN] arg_type_o [:= def_value_o] ) ]
  RETURN result_type IS
   ...
   ... -- Deklarationsanweisungen
   ...
BEGIN
   ...
   ... -- sequentielle Anweisungen
   ... -- RETURN-Anweisung obligatorisch
   ... -- keine WAIT-Anweisung in Funktionen!
   ...
END [function_name] ;
```

Im Deklarationsteil von Funktionen können lokale Typen und Untertypen, Konstanten, Variablen, Files, Aliase, Attribute und Gruppen (✓93) deklariert werden. `USE`-Anweisungen und Attributdefinitionen sind dort ebenfalls erlaubt. Außerdem können im Deklarationsteil andere Prozeduren und Funktionen deklariert und definiert werden.

Die eigentliche Funktionsbeschreibung besteht aus sequentiellen Anweisungen (ausgenommen `WAIT`-Anweisung) und kann selbst wieder Unterprogrammaufrufe enthalten. Sämtliche Argumente können innerhalb der Funktion nur gelesen werden (Modus `IN`) und dürfen deshalb nicht verändert werden. Die Funktion muß an mindestens einer Stelle mit der `RETURN`-Anweisung verlassen werden:

```
RETURN result_value ;
```

Das Ergebnis der Funktion (Rückgabewert *`result_value`*) kann in Form von expliziten Wertangaben, mit Objektnamen oder durch Ausdrücke angegeben werden.

Die Vereinheitlichung der Rahmensyntax in ✓**93** führt zu folgender Syntaxalternative:

```
FUNCTION function_name ... IS
  ...
BEGIN
  ...
END [FUNCTION] [function_name] ;
```

Einige Beispiele für die Definition von Funktionen:

```
-- Umwandlung von bit in integer ('0' -> 0, '1' -> 1)
FUNCTION bit_to_integer (bit_a : bit) RETURN integer IS
BEGIN
    IF bit_a = '1' THEN RETURN 1 ;
    ELSE RETURN 0 ;
    END IF ;
END bit_to_integer ;
```

```
-- Zaehlen der Einsstellen in einem Bitvektor unbestimmter
-- Laenge (flexibles Modell). Abfrage der aktuellen Vektor-
-- laenge durch das Attribut RANGE.
FUNCTION count_ones (a : bit_vector) RETURN integer IS
    VARIABLE count : integer := 0 ;
BEGIN
    FOR c IN a'RANGE LOOP
      IF a(c) = '1' THEN count := count + 1 ;
      END IF ;
    END LOOP ;
    RETURN count ;
END count_ones ;
```

```
-- Exklusiv-NOR-Verknuepfung fuer allgemeine Bitvektoren
FUNCTION exnor (a, b : bit_vector) RETURN bit_vector IS
-- Normierung auf gleiche Indizierung der beiden Argumente
-- ueber Aliase c und d:
    ALIAS c : bit_vector(1 TO a'LENGTH) IS a ;
    ALIAS d : bit_vector(1 TO b'LENGTH) IS b ;
    VARIABLE result : bit_vector(1 TO a'LENGTH) ;
BEGIN
    ASSERT a'LENGTH = b'LENGTH
      REPORT "Different Length of vectors!" ;
    FOR k in c'RANGE LOOP
      IF (c(k) = '1' AND d(k) = '0') OR
         (c(k) = '0' AND d(k) = '1') THEN result(k) := '0' ;
      ELSE result(k) := '1' ;
      END IF ;
    END LOOP ;
    RETURN result ;
END exnor ;
```

6.6.1.3 *Funktionsaufruf*

Der Aufruf einer Funktion geschieht unter Angabe der aktuellen Argumentwerte in Form von expliziten Wertangaben, Objektnamen oder Ausdrücken. Der Aufruf muß an einer Stelle stehen, an der auch der Ergebnistyp der Funktion erlaubt ist. Für die Übergabe der Argumentwerte ist, wie bei der Port Map, die Zuordnung der Werte über ihre Position ("positional association") möglich:

```
function_name [(arg_1_value {, arg_n_value})]
```

oder kann durch die explizite Zuordnung zwischen den Funktionsargumenten (`arg_name`) und den Aufrufwerten ("named association") erfolgen:

```
function_name [( arg_name_1 => arg_1_value
               {, arg_name_n => arg_n_value})]
```

Die Kombination der beiden Aufrufmethoden ist unter Beachtung der gleichen Regeln wie bei der Port Map erlaubt. Wird ein Aufrufwert nicht angegeben, dann gilt der in der Funktionsdeklaration festgelegte Defaultwert (`def_value`).

Die oben definierten Funktionen lassen sich beispielsweise innerhalb von Signalzuweisungen aufrufen:

```
ENTITY functions IS
  PORT (in1, in2  : IN  bit_vector (8 DOWNTO 1):="11111111";
        exnor_out : OUT bit_vector (8 DOWNTO 1) );
END functions;

ARCHITECTURE behavioral OF functions IS
  SIGNAL x2  : bit_vector (2 DOWNTO 1);
  SIGNAL i,j : integer := 0;
BEGIN
  exnor_out <= exnor (in1, in2);
  x2 <= exnor (a => in1(2 DOWNTO 1), b => in2(2 DOWNTO 1));
  i <= bit_to_integer(bit_a => in1(8)) + count_ones(in2);
  j <= count_ones("111010010111001010010001");
END behavioral;
```

6.6.1.4 "Impure Functions" ✓93

Funktionen sollten nach der ursprünglichen Definition der Sprache VHDL keinerlei "Seiteneffekte" aufweisen, d.h. der Aufruf einer Funktion zu verschiedenen Zeitpunkten und an verschiedenen Stellen in VHDL-Modellen soll für gleiche Argumente auch gleiche Ergebnisse liefern. Auf Objekte, die nicht als Argumente oder in der Funktion selbst deklariert wurden, kann daher auch nicht zugegriffen werden.

Viele Hardwareentwickler empfanden diese im Englischen als "pure" bezeichneten Funktionen als eine große Einschränkung. Ihre Forderung nach Lockerung der Richtlinien für Funktionen wurde in ✓93 durch die Einführung von sog. "impure functions" berücksichtigt.

Herkömmliche, "pure" Funktionen werden wie bisher gehandhabt. Sie können nun aber optional mit dem Schlüsselwort PURE gekennzeichnet werden:

```
[PURE] FUNCTION function_name ... IS
  ...
BEGIN
  ...
END [FUNCTION] [function_name] ;
```

Die neue Klasse der "impure functions", gekennzeichnet durch das Wort IMPURE, kann nun auf globale Objekte wie z.B. Files zugreifen. Die Argumente müssen aber auch weiterhin den Modus IN besitzen. "Impure functions" werden wie folgt beschrieben (✓93):

```
IMPURE FUNCTION function_name ... IS
  ...
BEGIN
  ...
END [FUNCTION] [function_name] ;
```

Typische Anwendungsfälle für solche Funktionen sind z.B. die Erzeugung von Zufallszahlen oder das Lesen von Files.

6.6.2 Prozeduren

Prozeduren unterscheiden sich von Funktionen hauptsächlich durch ihren Aufruf und die Art der Argumente. Zusätzlich zum Modus IN sind auch OUT und der bidirektionale Modus INOUT erlaubt. Weiterhin sind neben Konstanten und Signalen auch Variablen als Argumentklasse gestattet.

6.6.2.1 Prozedurdeklaration

Die Prozedurdeklaration enthält die Beschreibung der an die Prozedur übergebenen Argumente (Modus IN und INOUT) und die von der Prozedur zurückgelieferten Ergebnisse (Modus INOUT und OUT):

```
PROCEDURE procedure_name
[({ [arg_class_m] arg_name_m {,arg_name_n} :
    [arg_modus_m] arg_type_m [:= def_value];}
    [arg_class_o] arg_name_o {,arg_name_p} :
     arg_modus_o  arg_type_o [:= def_value])];
```

Der Defaultwert des **Argumentmodus** (arg_modus) ist IN.

Die **Argumentklasse** (arg_class) kann neben SIGNAL und CONSTANT auch VARIABLE sein. Defaultwert der Klasse ist für den Modus IN CONSTANT, für die Modi OUT und INOUT VARIABLE.

```
PROCEDURE hello;
PROCEDURE d_ff ( CONSTANT delay  : IN  time := 2 ns ;
                 SIGNAL d, clk   : IN  bit ;
                 SIGNAL q, q_bar : OUT bit );
```

6.6.2.2 Prozedurdefinition

Entsprechend der Funktionsdefinition muß die Schnittstellenbeschreibung mit dem Schlüsselwort `IS` wiederholt werden:

```
PROCEDURE procedure_name
[({ [arg_class_m] arg_name_m {,arg_name_n} :
    [arg_modus_m] arg_type_m [:= def_value];}
    [arg_class_o] arg_name_o {,arg_name_p} :
     arg_modus_o  arg_type_o [:= def_value])]

    IS
    ...
    ... -- Deklarationsanweisungen
    ...
BEGIN
    ...
    ... -- sequentielle Anweisungen
    ... -- optional: RETURN-Anweisung
    ...
END [procedure_name] ;
```

Die Prozedurbeschreibung kann aus allen möglichen sequentiellen Anweisungen, einschließlich der `WAIT`-Anweisung, bestehen. Argumente vom Typ `IN` können innerhalb von Prozeduren nur gelesen werden; verändert werden dürfen nur Argumente des Typs `OUT` und `INOUT`. Prozeduren können explizit mit dem Schlüsselwort `RETURN` (ohne Argument) verlassen werden oder werden bis zum Ende abgearbeitet.

Die Vereinheitlichung der Rahmensyntax in ✓**93** führt bei optionaler Wiederholung des Schlüsselwortes `PROCEDURE` zu folgender Alternative für den Prozedurrahmen:

```
PROCEDURE procedure_name ... IS
   ...
BEGIN
   ...
END [PROCEDURE] [procedure_name] ;
```

Zwei Beispiele für Prozeduren:

```
-- Prozedur ohne Argumente
PROCEDURE hello IS
BEGIN
    ASSERT false REPORT "Hello world!" SEVERITY note ;
END hello ;
```

```
-- Beschreibung eines D-Flip-Flops innerhalb einer Prozedur
PROCEDURE d_ff (CONSTANT delay   : IN  time := 2 ns ;
                SIGNAL d, clk  : IN  bit ;
                SIGNAL q, q_bar: OUT bit ) IS
BEGIN
    IF clk = '1' AND clk'EVENT THEN
      q     <= d     AFTER delay ;
      q_bar <= NOT d AFTER delay ;
    END IF ;
END d_ff ;
```

6.6.2.3 Prozeduraufruf

Der Aufruf einer Prozedur erfolgt unter Angabe der Argumentwerte als eigenständige, sequentielle oder nebenläufige Anweisung. Wie beim Funktionsaufruf ist eine Angabe der Aufrufwerte durch Position ("positional association"):

```
procedure_name [(arg_1_value{, arg_n_value})];
```

oder eine explizite Zuordnung ("named association") möglich:

```
procedure_name  [( arg_name_1 => arg_1_value
                {,  arg_name_n => arg_n_value})];
```

Die Kombination der beiden Aufrufmethoden ist unter Beachtung der gleichen Regeln wie bei der Port Map erlaubt. Wird ein Aufrufwert

nicht angegeben, dann gilt der in der Prozedurdeklaration festgelegte Defaultwert (`def_value`).

Im nebenläufigen Fall des Prozeduraufrufes ist ein Label erlaubt:

```
[proc_call_label :] procedure_name [...] ;
```

Erläutert werden soll dies anhand einer alternativen Darstellung eines 4-bit-Registers, welches obige D-Flip-Flop-Prozedur verwendet:

```
ENTITY four_bit_register IS
  PORT (clk              : IN  bit ;
        in_d             : IN  bit_vector(3 DOWNTO 0) ;
        out_q, out_q_bar : OUT bit_vector(3 DOWNTO 0) ) ;
END four_bit_register ;
```

```
ARCHITECTURE with_procedures OF four_bit_register IS
  USE work.my_pack.all ; -- enthaelt Prozedur d_ff
BEGIN
   -- Varianten eines nebenlaeufigen Prozeduraufrufes
   d_ff_3 : d_ff (1.5 ns, in_d(3), clk,
                  out_q(3), out_q_bar(3)) ;
   d_ff_2 : d_ff (delay => 1.7 ns, d => in_d(2), clk => clk,
                  q => out_q(2), q_bar => out_q_bar(2)) ;
   d_ff_1 : d_ff (1.9 ns, in_d(1), clk,
                  q_bar => out_q_bar(1), q => out_q(1)) ;
   d_ff_0 : d_ff (q_bar => out_q_bar(0), clk => clk,
                  q => out_q(0), d => in_d(0)) ;
                  -- Defaultwert fuer delay: 2 ns
END with_procedures;
```

Bei einem nebenläufigen Prozeduraufruf wird die Prozedur immer dann aktiviert, wenn sich mindestens ein Argumentwert (Modus `IN` oder `INOUT`) ändert. Der Aufruf kann also auch als Prozeß interpretiert werden, welcher die genannten Argumente in seiner "sensitivity-list" enthält.

Diese einfache Möglichkeit, prozeßähnliche Konstrukte mehrfach in ein VHDL-Modell einzufügen, wird durch folgendes Beispiel verdeutlicht. Es handelt sich um zwei äquivalente Teile eines Modells, die ein Signal überwachen:

```
ENTITY anything_one IS
  PORT (sig_a, sig_b : IN bit; sig_c : OUT bit ) ;
  PROCEDURE monitoring (SIGNAL   a        : IN bit;
                        CONSTANT sig_name : IN string) IS
  BEGIN
    ASSERT false REPORT "Event on signal " & sig_name
    SEVERITY note ;
  END monitoring ;
BEGIN
  mon_sig_a : monitoring (sig_a, "anything_one:sig_a") ;
END anything_one ;
```

```
ARCHITECTURE behavioral OF anything_one IS
BEGIN
  mon_sig_b : monitoring (a => sig_b,
                          sig_name => "anything_one:sig_b") ;
  sig_c <= sig_a AND NOT sig_b ;
END behavioral ;
```

Diese erste Variante verwendet nebenläufige Prozeduraufrufe, die nicht nur in der Architektur, sondern auch im Anweisungsteil der Entity auftreten können (sofern es sich um sog. passive Prozeduren handelt).

Die nachfolgenden Varianten beschreiben das gleiche Verhalten mit Hilfe von ASSERT-Anweisungen und passiven Prozessen. Auch diese dürfen im Anweisungsteil einer Entity stehen:

```
ENTITY anything_two IS
  PORT (sig_a, sig_b : IN bit; sig_c : OUT bit ) ;
BEGIN
  mon_sig_a : PROCESS (sig_a)
  BEGIN
    ASSERT false REPORT "Event on signal anything_two:sig_a"
    SEVERITY note ;
  END PROCESS ;
END anything_two ;
```

```
ARCHITECTURE behavioral OF anything_two IS
BEGIN
  mon_sig_b : PROCESS (sig_b)
  BEGIN
    ASSERT false REPORT "Event on signal anything_two:sig_b"
    SEVERITY note ;
  END PROCESS ;
  sig_c <= sig_a AND NOT sig_b ;
END behavioral ;
```

```
ENTITY anything_three IS
  PORT (sig_a, sig_b : IN bit; sig_c : OUT bit ) ;
BEGIN
  mon_sig_a : ASSERT (sig_a'EVENT = false)
              REPORT "Event on signal anything_three:sig_a"
              SEVERITY note ;
END anything_three ;
```

```
ARCHITECTURE behavioral OF anything_three IS
BEGIN
  mon_sig_b : ASSERT (sig_b'EVENT = false)
              REPORT "Event on signal anything_three:sig_b"
              SEVERITY note ;
  sig_c <= sig_a AND NOT sig_b ;
END behavioral ;
```

7 Konfigurieren von VHDL-Modellen

Die Sprache VHDL bietet bei der strukturalen Beschreibung elektronischer Systeme ein hohes Maß an Flexibilität. So können unter anderem Modelle dadurch umkonfiguriert werden, daß man ihre interne Funktion austauscht, ihre Verdrahtung ändert oder die Modellparameter modifiziert.

"Konfigurieren" von VHDL-Modellen bedeutet im einzelnen also:

- die Auswahl der gewünschten Architekturalternative für eine Entity (d.h. Auswahl der internen Modellfunktion),
- die Auswahl der zu verwendenden Modelle für die einzelnen Instanzen bei strukturalen Modellen,
- das Verbinden von Signalen und Ports auf den unterschiedlichen Hierarchieebenen,
- die Zuweisung von Werten an die Parameter (Generics) der einzelnen Instanzen.

Diese Konfigurationsangaben werden in einer eigenen Design-Einheit, der "Configuration", zusammengefaßt. Änderungen an dieser Design-Einheit erfordern kein erneutes Compilieren des Gesamtmodells, so daß verschiedene Modellvarianten schnell untersucht werden können.

Daneben können auch in Deklarationsteilen von Architekturen und Blöcken und in den `GENERIC MAP`- und `PORT MAP`-Anweisungen der Komponenteninstantiierung Konfigurationsanweisungen stehen.

In vielen Fällen werden bei fehlenden Konfigurationskonstrukten Defaultwerte verwendet.

7.1 Konfiguration von Verhaltensmodellen

Bei Verhaltensmodellen ist nur eine Information erforderlich: Die Auswahl der gewünschten Architekturalternative für eine Schnittstellenbeschreibung. Dies wird in der Design-Einheit "Configuration" durch eine einfache Blockkonfigurationsanweisung vorgenommen:

```
CONFIGURATION conf_name OF entity_name IS
   ...
   ...  -- generelle USE-Anweisungen
   ...  -- Attributzuweisungen
   ...
   FOR arch_name    -- Architekturauswahl
   END FOR ;
END [CONFIGURATION] [conf_name] ;
```

Die optionale Wiederholung des Schlüsselwortes `CONFIGURATION` in der `END`-Anweisung ist nur in der neuen VHDL-Norm (✓93) gestattet.

7.2 Konfiguration von strukturalen Modellen

Bei strukturalen Modellen muß neben der Architekturauswahl jede instantiierte Komponente (jeder instantiierte Sockeltyp) konfiguriert werden, d.h. es muß festgelegt werden, in welcher Form ein existierendes VHDL-Modell in diese Sockelinstanzen eingesetzt wird. Dazu soll zunächst ein Blick auf die unterschiedlichen Ebenen bei strukturalen Modellen und deren Schnittstellen geworfen werden.

Das Konzept der strukturalen Modellierung über die Ebenen: Komponentendeklaration (Sockeltyp), Komponenteninstantiierung (Verwendung des Sockeltyps, "Sockelinstanz") und Komponentenkonfiguration (Einsetzen des Modells, "IC") weist folgende Schnittstellen auf:

- Schnittstelle zwischen dem eingesetzten Modell und der Komponente. Die Generics und Ports des ersteren werden dabei als "**formal**", die der letzteren als "**local**" bezeichnet. Hierfür benutzt man `GENERIC MAP`- und `PORT MAP`-Anweisungen in der Design-Einheit "Configuration".

- Schnittstelle zwischen der Komponenteninstanz und der Komponente. Die Signale, mit denen die Instanzen verdrahtet werden, und die Parameter, die an die Instanzen übergeben werden unter dem Begriff "**actual**" zusammengefaßt. Die Zuordnungen werden durch entsprechende `GENERIC MAP`- und `PORT MAP`-Anweisungen bei der Komponenteninstantiierung definiert.

In der "Configuration" wird mit hierarchisch geschachtelten `FOR-USE`-Anweisungen festgelegt, welche Modelle für die instantiierten Komponenten (Sockel) verwendet werden, wie die Ports verknüpft und welche Parameterwerte übergeben werden.

Man unterscheidet dabei zwischen Blockkonfigurationsanweisungen (für Architektur, `BLOCK`, `GENERATE`) und Komponentenkonfigurationsanweisungen (für die einzelnen Instanzen).

7.2.1 Konfiguration von Blöcken

Blöcke repräsentieren eine Hierarchieebene, die selbst wieder Komponenten und Blöcke enthalten kann. Dementsprechend können in einer Blockkonfiguration Komponentenkonfigurationen und auch weitere Blockkonfigurationen enthalten sein.

Auf oberster Ebene eines strukturalen Modells wird zunächst die gewünschte Architektur ausgewählt:

```
CONFIGURATION conf_name OF entity_name IS
   ...
   ...  -- generelle USE-Anweisungen
   ...  -- Attributzuweisungen
   ...
   FOR arch_name  -- Architekturauswahl
   ...  -- weitere Blockkonfigurationen
   ...  -- Komponentenkonfigurationen
   END FOR ;
END [CONFIGURATION] [conf_name] ;
```

Die Wiederholung des Schlüsselwortes `CONFIGURATION` am Ende der Design-Einheit ist nur in der neuen VHDL-Norm (✓**93**) gestattet.

Weitere Blockkonfigurationsanweisungen dienen zur näheren Beschreibung von GENERATE- und BLOCK-Anweisungen:

```
FOR block_name
   ...      -- weitere Blockkonfigurationen
   ...      -- Komponentenkonfigurationen
END FOR ;

FOR gen_name [(index_range)]
   ...      -- weitere Blockkonfigurationen
   ...      -- Komponentenkonfigurationen
END FOR ;
```

7.2.2 Konfiguration von Komponenten

Die Konfigurationsanweisungen für Komponenten stellen den Zusammenhang zwischen dem in der Architektur instantiierten Komponentensockel und dem darin einzusetzenden Modell her. Bei diesem Modell handelt es sich um ein bereits compiliertes Modell, das in der Bibliothek work oder in einer anderen Resource-Library abgelegt ist. Diese Modelle müssen also vor dem Übersetzen der Konfiguration angelegt und compiliert werden. Mit den von der Komponenteninstantiierung bekannten GENERIC MAP- und PORT MAP-Anweisungen werden die "local" und "formal" Ports und Generics verbunden.

Das jeweils einzusetzende Modell kann folgendermaßen beschrieben werden:

- durch Angabe seiner Konfiguration (conf_name):

```
FOR inst_name_1 {,inst_name_n} : comp_name
   USE CONFIGURATION conf_name
         [ GENERIC MAP (...) ]
         [ PORT MAP    (...) ] ;
END FOR ;
```

- oder durch den Namen seiner Schnittstellenbeschreibung (entity_name) mit gewünschter Architektur (arch_name):

```
FOR inst_name_1 {,inst_name_n} : comp_name
   USE ENTITY entity_name [(arch_name)]
        [ GENERIC MAP (...) ]
        [ PORT MAP    (...) ] ;
END FOR ;
```

- Möchte man einen Sockel (Komponenteninstanz) unbestückt lassen, genügt das Schlüsselwort `OPEN`.

```
FOR inst_name_1 {,inst_name_n} : comp_name
   USE OPEN ;
END FOR ;
```

Folgende Regeln werden angewandt, falls ein oder mehrere Teile dieser Angaben (`arch_name`, `GENERIC MAP`, `PORT MAP`) in der Konfiguration fehlen:

- Bei fehlender Architekturangabe (`arch_name`) wird ein Modell ohne Funktion bzw. nur mit den passiven Anweisungen der Entity eingesetzt.
- Stimmen Namen, Modi und Typen der Signale auf local- und formal-Ebene überein, so werden sie nach Namen miteinander verbunden (**Default**-`PORT MAP`).
- Jeder Parameter (Generic) in der Komponentendeklaration wird mit einem gleichnamigen Parameter der Entity verknüpft. Für den Parameterwert wird zuerst auf die `GENERIC MAP` der Komponenteninstantiierung zurückgegriffen. Wurden hier keine Parameterwerte angegeben, so gelten die Defaultwerte aus der Komponentendeklaration. Falls auch hier keine Werte für die Generics definiert sind, gelten die Defaultwerte aus der Entity (**Default**-`GENERIC MAP`).
- Fehlt die Komponentenkonfigurationsanweisung komplett, so wird diejenige Entity (incl. der Default-Architektur, d.h. der zuletzt compilierten Architektur der Entity) eingesetzt, deren Name mit dem Namen der Komponente übereinstimmt. Es gelten in diesem Fall die Regeln für die Default-`GENERIC MAP` und die Default-`PORT MAP`.

Wird als Komponentenkonfiguration ein Entity-Architecture-Paar angegeben und handelt es sich bei der beschriebenen Instanz ebenfalls um ein strukturales Modell, so muß dieses Modell durch weitere Konfigurationsanweisungen eine Ebene tiefer beschrieben werden.

Anstelle einzelner Instanzennamen (`inst_name`) können auch die Schlüsselwörter `OTHERS` (alle bisher noch nicht explizit beschriebenen Instanzen des Komponententyps `comp_name`) oder `ALL` (alle Instanzen des Typs `comp_name`) verwendet werden:

```
FOR OTHERS : comp_name USE ... ;
END FOR ;

FOR ALL :    comp_name USE ... ;
END FOR ;
```

Zwei Beispiele sollen diese nicht einfache Syntax verdeutlichen.

Die erste Konfiguration beschreibt den bereits eingeführten Halbaddierer (Abb. B-18), dessen Schnittstellenbeschreibung und strukturale Architektur im nachfolgenden aufgeführt ist. Die Komponenten (Sockeltypen) innerhalb dieses Modells tragen die Bezeichner `xor2` und `and2`. Die beiden Instanzen der Komponenten heißen `xor_instance` und `and_instance`. In diese Instanzen werden zwei Verhaltensmodelle aus der Bibliothek `work` mit unterschiedlichen Methoden eingesetzt. Das Modell für die erstgenannte Instanz wird durch die Angabe seiner Schnittstellenbeschreibung und seiner Architektur referenziert, das Modell für `and_instance` hingegen durch den Verweis auf dessen Konfiguration.

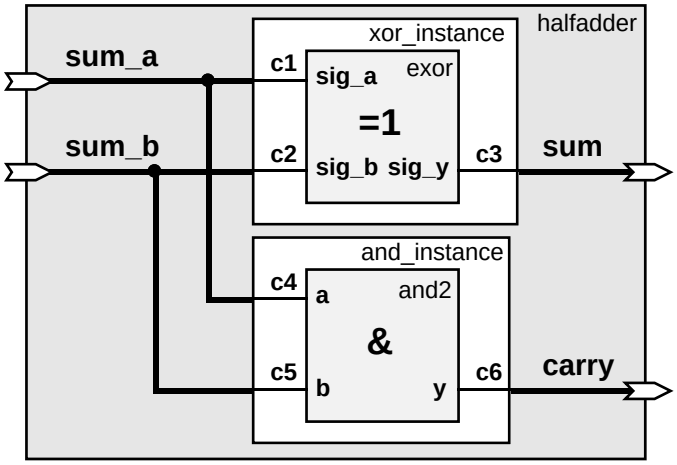

Abb. B-18: Struktur des Halbaddierers

```
ENTITY halfadder IS
  PORT (sum_a, sum_b : IN bit; sum, carry : OUT bit ) ;
END halfadder ;
```

```
ARCHITECTURE structural OF halfadder IS
-- Komponentendeklarationen
  COMPONENT xor2
    PORT (c1, c2 : IN bit; c3 : OUT bit) ;
  END COMPONENT ;
  COMPONENT and2
    PORT (c4, c5 : IN bit; c6 : OUT bit) ;
  END COMPONENT ;
BEGIN
-- Komponenteninstantiierungen
  xor_instance : xor2 PORT MAP (sum_a, sum_b, sum) ;
  and_instance : and2 PORT MAP (sum_a, sum_b, carry) ;
END structural ;
```



```
CONFIGURATION ha_config OF halfadder IS
  -- Blockkonfiguration
  FOR structural
    -- Komponentenkonfiguration
    FOR xor_instance: xor2
      USE ENTITY work.exor (behavioral)
      PORT MAP (c1,c2,c3) ;
    END FOR ;
    -- Komponentenkonfiguration
    FOR and_instance: and2
      USE CONFIGURATION work.and2_config
      PORT MAP (a => c4, b => c5, y => c6) ;
    END FOR ;
  END FOR ;
END ha_config ;
```

Das folgende VHDL-Beispiel zeigt eine mögliche Konfiguration für die Architektur `structural_4` des im Kapitel 5 (Abb. B-6) eingeführten 3-2-AND-OR-INVERT-Komplexgatters. Hier werden Default-`PORT MAP` und Default-`GENERIC MAP` verwendet. Weitere Beispiele für Konfigurationen können den VHDL-Modellen auf der Diskette entnommen werden.

```
CONFIGURATION aoi_config_4 OF aoi IS
  FOR structural_4      -- Blockkonf. (Architekturauswahl)
    FOR nor_stage       -- Blockkonfiguration
      -- Komponentenkonfiguration
      FOR or_c : or2 USE ENTITY work.or2  (behavioral);
      END FOR;
    END FOR;
    FOR and_stage       -- Blockkonfiguration
      -- Komponentenkonfigurationen
      FOR and_b : and2 USE ENTITY work.and2 (behavioral);
      END FOR;
      FOR and_a : and3 USE ENTITY work.and3 (behavioral);
      END FOR;
    END FOR;
  END FOR;
END aoi_config_4;
```

7.2.3 Konfiguration von Komponenten außerhalb der Design-Einheit "Configuration"

Die Konfiguration von Komponenten kann nicht nur in der Design-Einheit "Configuration", sondern auch in der Architektur selbst vorgenommen werden. Die dazu erforderlichen Komponentenkonfigurationsanweisungen sind im Deklarationsteil derjenigen Ebene anzugeben, auf der die entsprechenden Komponenteninstantiierungen selbst stehen. Dies kann also auch der Deklarationsteil eines Blockes sein.

Folgende Architektur des 3-2-AND-OR-INVERT-Komplexgatters benötigt keine eigene Design-Einheit "Configuration":

```
ARCHITECTURE structural_6 OF aoi IS
  SIGNAL a_out, b_out, or_out : bit;  -- interne Signale
  ..
  ..  -- Komponentendeklarationen
  ..
  --Komponentenkonfigurationen
  FOR ALL : inv  USE ENTITY work.not1 (behavioral);
  FOR ALL : or2  USE ENTITY work.or2  (behavioral);
  FOR ALL : and2 USE CONFIGURATION work.and2_config;
  FOR ALL : and3 USE CONFIGURATION work.and3_config;
BEGIN
  ..
  ..  -- Komponenteninstantiierungen
  ..
END structural_6 ;
```

7.2.4 Inkrementelles Konfigurieren ✓93

Die VHDL-Syntax in der ursprünglichen Version (✓87) erlaubt nur eine einmalige Konfiguration einer strukturalen Architektur, entweder innerhalb der Architektur (oder eines Blockes) oder in einer Konfiguration. Beim ASIC-Entwurf verhindert diese Einschränkung allerdings eine einfache Rückführung von exakten Verzögerungszeiten nach der Layouterzeugung (sog. "Backannotation"). Es wäre wünschenswert, die Auswahl des einzusetzenden Modells und die Signalverbindungen von der Zuweisung der Parameter (in diesem Fall: Verzögerungszeiten) zu trennen.

Dies ist nun mit der überarbeiteten Norm (✓93) möglich. Das sog. "incremental binding" erlaubt die Trennung der einzelnen Teile der Komponentenkonfiguration. Es sind sogar mehrfache GENERIC MAP-Anweisungen möglich. Ein Beispiel erläutert diese Vereinbarung:

```
ARCHITECTURE structural_7 OF aoi IS     -- !!! VHDL'93-Syntax
  SIGNAL a_out, b_out, or_out : bit;
  ..
  ..  -- Komponentendeklarationen
  ..
  -- einheitliche Laufzeiten pro Gattertyp (ALL)
  FOR ALL : inv  USE ENTITY work.not1 (behavioral)
                 GENERIC MAP (0.75 ns, 0.7 ns) ;
  FOR ALL : or2  USE ENTITY work.or2 (behavioral)
                 GENERIC MAP (1.6 ns, 1.5 ns) ;
  FOR ALL : and2 USE CONFIGURATION work.and2_config ;
  FOR ALL : and3 USE CONFIGURATION work.and3_config ;
BEGIN
  ..
  ..  -- Komponenteninstantiierungen
  ..
END structural_7 ;
```

```
CONFIGURATION aoi_config_7 OF aoi IS    -- !!! VHDL'93-Syntax
  FOR structural_7
    -- hier koennen instanzenspezifische Laufzeiten stehen
    FOR inv_d : inv  GENERIC MAP (0.9 ns, 0.8 ns); END FOR;
    FOR or_c  : or2  GENERIC MAP (1.8 ns, 1.7 ns); END FOR;
    FOR and_b : and2 GENERIC MAP (1.3 ns, 1.9 ns); END FOR;
    FOR and_a : and3 GENERIC MAP (1.4 ns, 2.0 ns); END FOR;
  END FOR;
END aoi_config_7;
```

Ein Simulationsaufruf nur mit den Konfigurationen aus der Architektur verwendet beispielsweise geschätzte Werte oder die Defaultwerte für die Verzögerungszeiten. Die Simulation durch Angabe der Konfiguration aoi_config_7 dagegen weist den Parametern der jeweiligen Gatter neue (z.B. aus dem Layout ermittelte) Verzögerungszeiten zu.