PEARSON Education

PEARSON Studium

Paul Molitor und Jörg Ritter

VHDL

Eine Einführung

PEARSON Studium

ein Imprint von Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Kapitel

3 Aufbau und Grundkonzepte von VHDL

Dieses Kapitel stellt die in VHDL verfügbaren Konstrukte, Datentypen, Attribute und Kontrollstrukturen vor. Ähnliche Kapitel findet man in der Regel in jedem Lehrbuch zu einer Programmiersprache, die alle — wie auch in diesem Buch — mehr oder weniger langweilig zu lesen sind. Nichtsdestotrotz sollten Sie diesem Kapitel Beachtung schenken, da VHDL doch einige Konzepte zur Verfügung stellt, die in „üblichen" Programmiersprachen nicht vorzufinden sind. Später können Sie dieses Kapitel wie ein Nachschlagewerk benutzen.

Wir geben keine vollständige Übersicht in dem Sinne, dass wirklich alle in VHDL zur Verfügung gestellten Elemente hier aufgezählt und erläutert sind. Wir haben uns aber bemüht, die wichtigsten, sprich die am häufigsten benutzten Konzepte hier anzusprechen. Sollten „ein oder zwei" fehlen, so bitten wir dies zu entschuldigen und verweisen für diesen Fall auf das fast 700-seitige Standardwerk von Ashenden [3].

Wir waren in diesem Kapitel auch gezwungen, zum Teil auf noch nicht eingeführte Elemente von VHDL vorzugreifen, um die vorgestellten VHDL-Elemente an sinnvollen Beispielen illustrieren zu können.

3.1 Genereller Aufbau einer VHDL-Spezifikation

Abbildung 3.1 zeigt den prinzipiellen Aufbau einer VHDL-Beschreibung, die aus einem Entity und einer oder mehreren Architekturen — im Regelfall einer Architektur — besteht. Packages, wie zum Beispiel die von IEEE vorgegebenen Packages `std_logic_1164` und `std_logic_textio`, auf die wir im Kapitel 5 detailliert eingehen werden, können eingebunden werden. Sie stellen oft benutzte Konstanten, Datentypen, Funktionen und Prozeduren zu Verfügung.

Da wir den Funktionen und Prozeduren als auch den Packages spezielle Kapitel in diesem Buch gewidmet haben (siehe Kapitel 4 und Kapitel 5), wollen wir an dieser Stelle nicht auf diese eingehen und uns auf die restlichen Konstrukte konzentrieren.

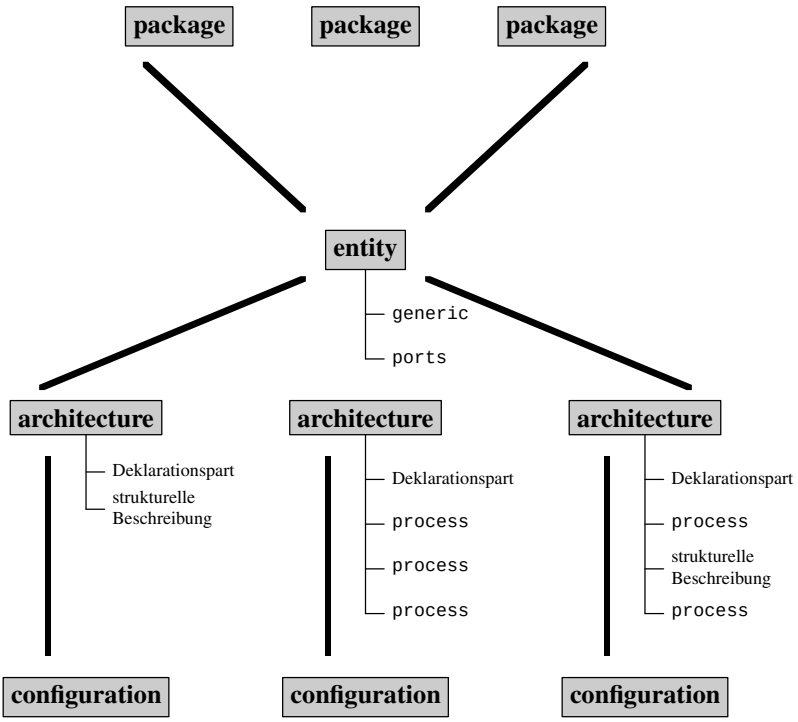


Abbildung 3.1: Aufbau einer VHDL-Beschreibung

3.1.1 Schnittstellen-Beschreibung eines Bausteins

Lassen Sie uns mit dem **`entity`**-Konstrukt beginnen. Wie Abbildung 3.1 zeigt, enthält die Beschreibung eines jeden mit VHDL beschriebenen Bausteins (beziehungsweise Schaltkreises) eine solche **`entity`**. Eine Entity ist in der Regel wie folgt aufgebaut:

```
entity Name_des_Bausteins is
   [[ generic ( Parameter_Liste ); ]]
   [[ port ( Liste_der_Ein_und_Ausgabepins ); ]]
end [[ entity ]] [[ Name_des_Bausteins ]] ;
```

Die Entity eines Bausteins legt dessen Namen fest und beschreibt seine Schnittstelle zur Außenwelt.[1] Die Schnittstelle wird im Wesentlichen durch die Ports beschrieben, die festlegen, welche Eingabe- und Ausgabesignale der Baustein hat. Diese Schnittstelle

[1] Die in [[und]] eingeschlossenen Zeichenketten sind optional und sind nur bei Bedarf Bestandteil des Konstrukts.Wie Sie feststellen werden, verwenden wir in diesem Kapitel zum Teil eine recht informale Schreibweise, um Ihnen die Syntax nahe zu bringen. Die exakte Syntax – exakt auch in Bezug auf die Schreibweise – ist in Kapitel 11 zu finden.

kann über generische Parameter parametrisiert sein. Zur Illustration sei hier bereits auf das Kapitel 6 vorgegriffen, in dem der mit der Wortbreite der Operanden parametrisierte serielle und der so genannte log n-Addierer/Subtrahierer mit VHDL beschrieben werden. Deren Schnittstelle könnte zum Beispiel durch

```
3   entity add_sub is
4     generic (n: positive);
5     port
6       (a, b: in std_logic_vector (n-1 downto 0);
7        op : in std_logic;
8        s : out std_logic_vector (n-1 downto 0);
9        ov : out std_logic);
10  end add_sub;
```

gegeben sein. Der Baustein bekommt mit dieser Entity-Beschreibung den Namen `add_sub` zugeordnet. Die Ports sind mit dem generischen Parameter `n`, der über den Datentyp `positive` deklariert ist, parametrisiert. Der Baustein besitzt drei Eingabeports, von denen zwei, nämlich die Ports `a` und `b`, jeweils einen Vektor über dem Datentyp `std_logic` der Länge `n` darstellen. Von den beiden Ausgabeports ist Port `s` ebenfalls ein Vektor der Länge `n`.

Bei den Ports unterscheidet VHDL zwischen Eingabeports, Ausgabeports und so genannten bidirektionalen Ports.[2] Die entsprechende Zuordnung eines Ports erfolgt in der Port-Deklaration über das Schlüsselwort **`in`**, **`out`** oder **`inout`**.

Fallstrick 3

Ist ein Port als **`in`**-Port deklariert, so kann nur lesend auf diesen Port durch den Baustein zugegriffen werden; ist er als **`out`**-Port deklariert, so ist nur schreibender Zugriff möglich.

Ports vom Modus **`inout`** stellen Ports dar, die, je nach Kontext beziehungsweise abwechselnd, als Eingabeports oder Ausgabeports fungieren. Bei Benutzung von **`inout`**-Ports muss auf die Synthetisierbarkeit des Entwurfs Acht gegeben werden. Ein Beispiel, wie bidirektionale Ports modelliert werden können, sodass diese Beschreibungen auch synthetisierbar sind, finden Sie im Abschnitt 9.2.

Nicht jeder Baustein ist generischer Art, sodass der generische Parameter im **`entity`**-Konstrukt optional ist. Beispiele nicht generischer Bausteine sind das Flipflop und das Schieberegister der Breite 4, deren Schnittstellen-Beschreibungen durch

[2] Neben diesen drei Modi stellt VHDL noch die Modi **`buffer`** und **`linkage`** zur Verfügung. Ein **`buffer`**-Port ist ein **`out`**-Port, auf den der Baustein auch lesend zugreifen kann. **`Buffer`**-Ports haben kein Äquivalent in wirklicher Hardware. Der Modus **`linkage`** dient dazu, um in VHDL spezifizierte Bausteine mit nicht in VHDL spezifizierten Bausteinen kombinieren zu können. Beide Modi finden nur selten Anwendung.

```
4   entity flipflop is
5     port ( clockpin, resetpin, data_in: in std_logic;
6            data_out: out std_logic );
7   end flipflop;
```

und

```
4   entity schieberegister_4 is
5     port ( clockpin, resetpin, data_in : in std_logic;
6            data_out : out std_logic );
7   end schieberegister_4;
```

angegeben werden können. Beide Bausteine werden wir noch im Rahmen der folgenden Abschnitte benötigen

Im Rahmen einer Schnittstellen-Beschreibung können des Weiteren Eingabeports für den Fall auf einen Default-Wert gesetzt werden, dass ein Eingabeport (oder mehrere) bei Verwendung des Bausteins als Komponente eines anderen Bausteins nicht durch ein Signal getrieben werden würde. Zur Illustration sehen wir uns nochmals das oben angegebene Entity des kombinierten Addierer/Subtrahierer an. Über den Eingabeport `op` wird die Art der Operation gewählt. Liegt am Eingabeport der Wert `'0'` an, so sollen die beiden Operanden `a` und `b` addiert werden, liegt der Wert `'1'` an, so soll der Operand `b` von dem Operanden `a` subtrahiert werden. VHDL bietet nun die Möglichkeit, schon in der **`entity`**-Deklaration zu spezifizieren, dass standardmäßig zum Beispiel addiert werden soll, wenn der Eingabeport `ov` offen bleibt — dies erfolgt dadurch, dass diesem Eingabeport während der Instanziierung der Komponente „das Schlüsselwort **`open`** zugewiesen“ wird. Die erweiterte Schnittstellen-Beschreibung hat dann das Aussehen

```
3a  entity add_sub is
3b    generic (n: positive);
3c    port
3d      (a, b: in std_logic_vector (n-1 downto 0);
3e       op : in std_logic:='0';
3f       s : out std_logic_vector (n-1 downto 0);
3g       ov : out std_logic);
3h  end add_sub;
```

Fallstrick 4

Alle Default-Belegungen werden bei der Synthese ignoriert. Die vorbelegten Ports müssen explizit verbunden werden. Eine Initialisierung von Signalen und Variablen muss über ein asynchrones oder synchrones Reset erfolgen!

Es sei auch darauf hingewiesen, dass die Angabe der Ports selbst optional ist, d. h. eine Entity der Art

```
4   entity test is
5   end test;
```

ist erlaubt und wird in der Tat auch im Rahmen der Implementierung von Testbenches benutzt. Testbenches, auf die wir detailliert im Teil IV noch zu sprechen kommen, könnten beispielsweise Stimuli und das dazugehörige Soll-Verhalten eines zu validierenden Bausteins aus einer Datei einlesen, diese Stimuli an den Baustein anlegen und das Ist-Verhalten mit dem Soll-Verhalten vergleichen. Die Ergebnisse der Vergleiche könnten wieder in eine Datei gespeichert werden oder mit **report**-Anweisungen auf die Standardausgabe ausgegeben werden. Insofern ist eine solche Testbench formal gesehen ein Baustein ohne Eingabe- und Ausgabeport.

Zuletzt sei noch bemerkt, dass Entities „Plausibilitätskontrollen“, so genannte nebenläufige **assert**-Anweisungen, auf die wir im Abschnitt 3.5 noch detailliert eingehen werden, enthalten können, sodass eine allgemeinere Form der Beschreibung von Entities wie folgt[3]

---

entity Name_des_Bausteins **is**
  ⟦ **generic** (Parameter_Liste); ⟧
  ⟦ **port** (Liste_der_Ein_und_Ausgabepins); ⟧
⟦ **begin**
  { nebenläufige_assert_Anweisung } ⟧
end ⟦ **entity** ⟧ ⟦ Name_des_Bausteins ⟧ ;

---

aussieht. Beispielsweise deklariert

```
4   entity single2double is
5     -- Baustein zur Konvertierung
6     -- einer n-bit Zweierkomplement-Darstellung
7     -- in eine m-bit Zweierkomplement-Darstellung mit n<=m
8     port (
9       inp: in std_logic_vector;
10      outp: out std_logic_vector);
11  begin
12    ueberpruefe_laenge:
13      assert inp'length <= outp'length
14      report "Die Wortbreite des Ausgangs ist zu klein!";
15
16    ueberpruefe_rechter_index:
17      assert inp'right = 0 and outp'right = 0
18      report "Rechte Komponente muss Index 0 haben!";
19
20  end entity single2double;
```

einen Baustein, der eine n-Bit-Zweierkomplement-Darstellung in eine m-Bit Zweierkomplement-Darstellung umwandelt. Beachten Sie bitte, dass es sich bei den Ports um

[3] Die in geschweiften Klammern eingeschlossenen Zeichenketten können beliebig oft hintereinander eingesetzt werden. Nähere Informationen zu den Schreibweisen sind in Abschnitt 11.1 auf Seite 255 zu finden.

unbeschränkte Felder (siehe Abschnitt 3.3.2 auf Seite 90) handelt. Aus diesem Grunde muss überprüft werden — diese Überprüfung erfolgt über die erste **assert**-Anweisung —, ob die Wortbreite `m` des Ausgangssignals `outp`, die über das Attribut `outp'length` abgefragt werden kann, breiter als die Wortbreite `n` (= `inp'length`) ist — ansonsten ist die durch den Baustein zu realisierende Operation unsinnig. Zudem sollte die rechte Komponente des übergebenen Signals jeweils den Index 0 haben.

3.1.2 Architekturen eines Bausteins

Lassen Sie uns jetzt zum VHDL-Konstrukt **architecture** kommen, welches das Herzstück einer VHDL-Beschreibung eines Bausteins bildet. Während das **entity**-Konstrukt das Interface, d. h. die Ein- und Ausgabeschnittstelle, eines Bausteins beschreibt, gibt das **architecture**-Konstrukt die Realisierung des Bausteins an.

Eine Architektur ist in der Regel wie folgt aufgebaut:

```
architecture Name_der_Architektur of Name_des_Bausteins is
   [[ Deklarationsteil ]]
begin
   { VHDL-Anweisung }
end Name_der_Architektur;
```

So spezifiziert der Quellcode[4]

```
9   architecture D_flipflop of flipflop is
10  begin
11    VERHALTEN: process(clockpin,resetpin)
12    begin
13      if resetpin='1' then
14        data_out <= '0';
15      elsif rising_edge(clockpin) then
16        data_out <= data_in;
17      end if;
18    end process;
19  end D_flipflop;
```

eine funktionale Beschreibung des Bausteins `flipflop`, dessen Interface in der schon oben angegebenen **entity**-Anweisung spezifiziert wurde. Die Architektur besteht aus einem Prozess namens `VERHALTEN`, der immer dann aktiviert wird, wenn eine Signaländerung auf der Taktleitung oder der Reset-Leitung, die durch das Signal `clockpin` beziehungsweise das Signal `resetpin` modelliert werden, erfolgt ist. Der Prozess beschreibt in Abhängigkeit der Belegung des Signals `resetpin` das Signal `data_out`. Ist die Reset-Taste gedrückt, d. h. gilt `reset_pin=1`, so wird das Signal `data_out` auf den

[4] Das Konstrukt `rising_edge(clk)` entspricht der Abfrage auf die steigenden Flanke der Taktleitung (`clk='1' and clk'event`).

Wert '0' zurückgesetzt. Ansonsten übernimmt es den Wert des Signals data_in bei steigender Flanke am Takteingang.

Wir wollen an dieser Stelle auch schon darauf hinweisen, dass zu einer Entity nicht nur eine Architektur angegeben werden darf, sondern auch mehrere Architekturen angegeben werden können. So könnte man sich überlegen, das Flipflop auch nach dem Master-Slave-Prinzip aufzubauen, was in der Architektur

```
21  architecture master_slave of flipflop is
22     signal zustand_des_masters: std_logic;
23  begin
24    MASTER: process(clockpin,resetpin)
25    begin
26      if resetpin='1' then
27        zustand_des_masters <= '0';
28      elsif rising_edge(clockpin) then
29        zustand_des_masters <= data_in;
30      end if;
31    end process;
32    SLAVE: process(clockpin,resetpin)
33    begin
34      if resetpin='1' then
35        data_out <= '0';
36      elsif falling_edge(clockpin) then
37        data_out <= zustand_des_masters;
38      end if;
39    end process;
40  end master_slave;
```

die im Wesentlichen aus zwei nebenläufigen Prozessen besteht, resultieren würde. Wird der flipflop-Baustein dann als Komponente in einer strukturellen Beschreibung eines anderen Schaltkreises verwendet, so wird über eine **configuration**-Anweisung für jeden der Instanzen angegeben, welche Architektur zu verwenden ist. Wir werden auf diesen Punkt in Abschnitt 3.1.3 im Detail zu sprechen kommen.

Funktionale Beschreibungen

Die beiden oben angegebenen Architekturen D_flipflop und master_slave des Bausteins flipflop wurden mit Hilfe funktionaler Beschreibungen spezifiziert.

> Funktionale Beschreibungen bestehen aus einer Menge von nebenläufigen (expliziten und impliziten) Prozessen.

Die Syntax eines Prozesses ist durch

```
[[ Marke : ]] process [[ ( Sensitivitätsliste ) ]] [[ is ]]
    Deklarationsteil
begin
    Ausführungsteil
end [[ process ]] [[ Marke ]] ;
```

gegeben.

Im Deklarationsteil eines Prozesses können Datentypen, Funktionen und Prozeduren definiert und Variablen sowie Konstanten deklariert werden.

Fallstrick 5

Signale können nicht in Prozessen, sondern nur im Deklarationsteil einer Architektur beziehungsweise im Port-Deklarationsteil einer Entity-Beschreibung deklariert werden.

Der Ausführungsteil eines Prozesses besteht aus einer Folge von sequentiellen Anweisungen, die wie der Name schon sagt, sequentiell nacheinander durch die Simulation ausgeführt werden.

Mehr zu Prozessen ist in Kapitel 2 im Abschnitt 2.2.3, der sich mit der Ausführung von Prozessen während eines Simulationslaufes beschäftigt, ab Seite 55 zu finden.

Strukturelle Beschreibungen

Neben funktionalen Beschreibungen erlaubt VHDL auch strukturelle Beschreibungen. Zur weiteren Illustration lassen Sie uns verschiedene Architekturen des 4-Bit-Schieberegisters angeben, dessen Schnittstellen-Beschreibung wir oben schon angegeben haben. Hiermit wollen wir, auch schon im Vorgriff auf die nächsten Abschnitte, zeigen, welche Möglichkeiten VHDL uns zur Realisierung von Architekturen zur Verfügung stellt.

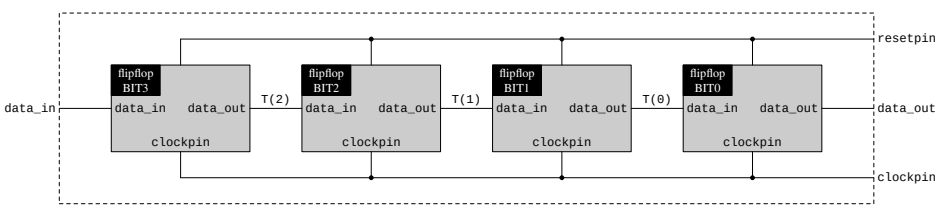


Abbildung 3.2: Aufbau eines 4-Bit-Schieberegisters

Abbildung 3.3 zeigt den VHDL-Quellcode des in Abbildung 3.2 gezeigten Aufbaus des Schieberegisters. Die Architektur haben wir `RTL1` genannt — die Abkürzung kommt von

```
 9  architecture RTL1 of schieberegister_4 is
10    component flipflop
11    port ( clockpin, resetpin, data_in: in std_logic;
12           data_out: out std_logic );
13    end component;
14    signal T: std_logic_vector(2 downto 0);
15  begin
16    BIT3 : flipflop
17      port map ( clockpin => clockpin, resetpin => resetpin,
18                 data_in => data_in, data_out => T(2) );
19    BIT2 : flipflop
20      port map ( clockpin => clockpin, resetpin => resetpin,
21                 data_in => T(2), data_out => T(1) );
22    BIT1 : flipflop
23      port map ( clockpin, resetpin, T(1), T(0) );
24    BIT0 : flipflop
25      port map ( data_in => T(0), data_out => data_out,
26                 resetpin => resetpin, clockpin => clockpin );
27  end RTL1;
```

Abbildung 3.3: Eine erste strukturelle Architektur des Bausteins `schieberegister_4`

der englischen Bezeichnung *Register Transfer Level*, die eine im Rahmen der Hardwarebeschreibung oft verwendete Abstraktionsebene bezeichnet.

Hier sehen wir etwas detaillierter, wie in der Regel eine Architektur, die auf einer strukturellen Beschreibung basiert, aussieht.

- Im Deklarationsteil werden neben notwendigen internen Signalen alle Komponenten, die zur strukturellen Beschreibung benötigt werden, über das Schlüsselwort **`component`** deklariert. In unserem Beispiel — betrachten Sie den Aufbau in Abbildung 3.2 — sind dies die Komponente `flipflop` und die internen, durch das Feld `T` modellierten, Signale, welche die einzelnen Instanzen dieser Komponente miteinander verbinden.

  Eine Komponenten-Deklaration sieht formal wie folgt aus:

  ```
  component Name_des_Bausteins is
    [[ generic ( Parameter_Liste ); ]]
    [[ port ( Liste_der_Ein_und_Ausgabepins ); ]]
  end component;
  ```

- Im eigentlichen Beschreibungsblock werden dann nacheinander die vier Instanzen der Komponente `flipflop` generiert — wir sprechen von der Instanziierung von Komponenten —; sie tragen die Namen `BIT3`, `BIT2`, `BIT1` und `BIT0`. Über die **`port map`**-Anweisung werden jeweils die Ports der Instanzen mit der Außenwelt verdrahtet. Wir erkennen zwei Arten der Port-Verdrahtung, das so genannte *name mapping* und das so genannte *positional mapping*.

– *Name mapping* haben wir in unserem Beispiel bei der Instanziierung von `BIT3`, `BIT2` und `BIT0` angewendet. Hier erfolgt die Verdrahtung direkt über die Portnamen. Die Reihenfolge, in der dies erfolgt, ist egal. Beispielsweise haben wir die Reihenfolgen bei den Instanziierungen von `BIT2` und `BIT0` unterschiedlich gewählt.

– Bei *Positional Mapping* erfolgt die Port-Verdrahtung, wie in vielen Programmiersprachen üblich, über die Position der Ports und der Signale — hier steckt also die Information, wer mit wem, in der Reihenfolge, in der die „aktuellen Parameter" übergeben werden. Ein Beispiel ist bei der Instanziierung von `BIT1` zu sehen.

Instanziierungen von Komponenten haben allgemein das Aussehen

---

Instanziierung_Marke**:** Name_der_Komponente
[[**generic map** (Zuweisung_an_die_generischen_Parameter **);**]]
[[**port map** (Anbindung_der_Ein_und_Ausgabepins **);**]]

Der in Abbildung 3.3 angegebene Quellcode wirkt beschwerlich, wenn man Schieberegister größerer Breite spezifizieren möchte, da man für jede benötigte Instanz der Komponente `flipflop` explizit eine Instanziierung-Anweisung angeben muss.

Um hier Abhilfe zu schaffen, stellt VHDL die Anweisungen **`for...generate`** und **`if...generate`** zur Verfügung. Abbildung 3.4 zeigt eine strukturelle Beschreibung unseres 4-Bit Schieberegisters, die diese Anweisungen verwendet. Man sieht leicht, dass entsprechend größere Schieberegister nun mit dem gleichen Aufwand beschrieben werden können wie das Schieberegister der Breite 4. Wir werden gleich nochmals hierauf zurückkommen.

In dem Quellcode in Abbildung 3.4 sind vier **`generate`**-Anweisungen enthalten. Eine **`for...generate`**-Anweisung und drei **`if...generate`**-Anweisungen. Die **`for...generate`**-Anweisung ermöglicht die iterative Instanziierung einer Komponente. So werden hier vier Instanziierungen der Komponente `flipflop` vorgenommen. Die Port-Verdrahtung der Instanzen wird über die **`if...generate`**-Anweisungen gesteuert. Ist die Laufvariable `i` — sie muss nicht (und darf auch nicht) deklariert werden; sie ist zudem nur innerhalb der **`for`**-Schleife gültig —, gleich 0, so soll die hintere Instanz des Schieberegisters generiert werden; ist die Laufvariable `i` gleich 3, so soll die vordere Instanz generiert werden, ansonsten handelt es sich um eine Instanz, die über die Signal `T(i)` und `T(i-1)` mit ihrer Umgebung verbunden ist.

Einer **`generate`**-Anweisung muss wie einer Instanziierung einer Komponente eine Marke voranstehen — dies sind in unserem Beispiel die Marken `G0`, `G1`, `G2` und `G3`. Diese Marken schaffen die Voraussetzung, die verschiedenen Instanzen einer Komponente an unterschiedliche Architekturen des zugehörigen Entity binden zu

können. Nähere Informationen zu diesem Themenkreis sind im Abschnitt 3.1.3 zu finden. Während der Simulation kann über die Marken die jeweilige Instanz ausgewählt werden. In diesem Kontext wirken die Marken wie ein Selektor innerhalb der Schaltungshierarchie.

```
29 architecture RTL2 of schieberegister_4 is
30   component flipflop
31   port ( clockpin, resetpin, data_in: in std_logic;
32          data_out: out std_logic );
33   end component;
34   signal T: std_logic_vector(2 downto 0);
35 begin
36   G0: for i in 3 downto 0 generate
37     G1: if (i = 3) generate
38       BIT3: flipflop
39         port map (clockpin,resetpin,data_in,T(2));
40     end generate;
41     G2: if (i>0) and (i<3) generate
42       BITm: flipflop
43         port map (clockpin,resetpin,T(i),T(i-1));
44     end generate;
45     G3: if (i=0) generate
46       BIT0: flipflop
47         port map (clockpin,resetpin,T(0),data_out);
48     end generate;
49   end generate;
50 end RTL2;
```

Abbildung 3.4: Strukturelle Architektur des Bausteins `schieberegister_4` unter Verwendung der **`generate`**-Anweisungen

Die **`for...generate`**- und **`if...generate`**-Anweisungen sind wesentlich, um generische Bausteine mit VHDL beschreiben zu können. Ein ausführliches Beispiel zu dieser Thematik befindet sich in Kapitel 6.

Zur Einstimmung wollen wir uns aber hier schon ein generisches Schieberegister anschauen. Die Länge des Schieberegisters kann über einen Parameter `n` den Erfordernissen angepasst werden. Abbildung 3.5 zeigt das Entity und eine dazugehörige Architektur. Wenn wir den Quellcode mit dem aus Abbildung 3.4 vergleichen, sehen wir in der Tat, welche Beschreibungsmöglichkeiten uns mit den **`generate`**-Anweisungen gegeben sind. Die Architektur ist eine Kopie der in Abbildung 3.4 gegebenen Architektur des 4-Bit-Schieberegisters mit Ausnahme, dass wir anstelle der Konstante `3` nunmehr den generischen Wert `n-1` verwenden.

```
4  entity schieberegister is
5    generic ( n : positive );
6      port ( clockpin, resetpin, data_in : in std_logic;
7             data_out : out std_logic );
8  end schieberegister;
9
10 architecture RTL of schieberegister is
11   component flipflop
12   port ( clockpin, resetpin, data_in: in std_logic;
13          data_out: out std_logic );
14   end component;
15   signal T: std_logic_vector(n-1 downto 0);
16 begin
17   G0: for i in n-1 downto 0 generate
18     G1: if (i = n-1) generate
19       BIT_high: flipflop
20         port map (clockpin,resetpin,data_in,T(n-1));
21     end generate;
22     G2: if (i>0) and (i<n-1) generate
23       BITm: flipflop
24         port map (clockpin,resetpin,T(i),T(i-1));
25     end generate;
26     G3: if (i=0) generate
27       BIT0: flipflop
28         port map (clockpin,resetpin,T(0),data_out);
29     end generate;
30   end generate;
31 end RTL;
```

Abbildung 3.5: VHDL-Beschreibung eines generischen Schieberegisters

3.1.3 Konfigurationen

Zu einer Entity können mehrere Architekturen definiert sein. Wir müssen also die Zuordnung zur Schnittstellenbeschreibung festlegen.

Dies erfolgt über so genannte Konfigurationen. Hiermit kann elegant für bestimmte oder auch alle Instanzen einer verwendeten Komponente entschieden werden, welche Realisierung zum Einsatz kommt. Eine Konfiguration sieht in ihrer einfachsten Form wie folgt aus:

```
configuration Name_der_Konfiguration of Name_des_Bausteins is
   for Name_der_Architektur
      { for Komponenten_Instanz_aus_der_Architektur :
         Zuordnung_der_Instanz_zu_einer_seiner_Architekturen ;
      end for ; }
   end for ;
end [[ configuration ]] [[ Name_des_Bausteins ]] ;
```

So könnten die Konfigurationen unseres 4-Bit-Schieberegisters zum Beispiel durch

```
52 configuration CFG1a of schieberegister_4 is
53    for RTL1
54       for all: flipflop
55          use entity work.flipflop(D_flipflop);
56       end for;
57    end for;
58 end CFG1a;
```

beziehungsweise

```
68 configuration CFG2 of schieberegister_4 is
69   for RTL2
70      for G0
71         for G1
72           for BIT3: flipflop
73             use entity work.flipflop(D_flipflop);
74           end for;
75         end for;
76      end for;
77      for others: flipflop
78         use entity work.flipflop(master_slave);
79      end for;
80   end for;
81 end CFG2;
```

gegeben sein. Wir wollen beide Codefragmente erläutern.

- Will man die Instanzen einer Komponente einer Architektur alle an die gleiche Architektur der entsprechenden Entity — die Anweisung

  ```
  55          use entity work.flipflop(D_flipflop);
  ```

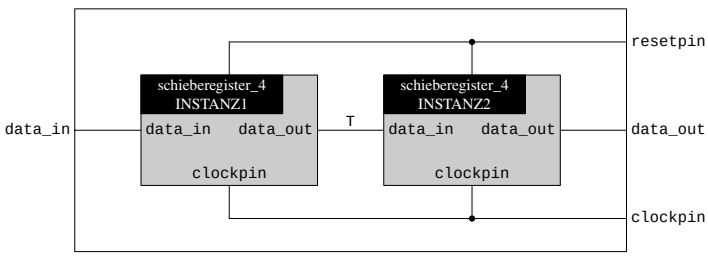
  besagt, dass die Architektur `D_flipflop` der Entity `flipflop`, die in der Bibliothek `work` abgelegt ist, verwendet werden soll — binden, so kann dies, wie oben im ersten Beispiel gezeigt, über eine **`for all`**-Anweisung, gefolgt von dem entsprechenden **`use`**-Befehl, erfolgen. Äquivalent dazu, vom Schreibaufwand aber aufwändiger, wäre die Konfiguration

  ```
  60 configuration CFG1b of schieberegister_4 is
  61    for RTL1
  62      for BIT3, BIT2, BIT1, BIT0: flipflop
  63         use entity work.flipflop(D_flipflop);
  64      end for;
  65   end for;
  66 end CFG1b;
  ```

- Ein bisschen komplizierter sieht es aus, wenn man verschiedenen Instanzen einer Komponente verschiedene Realisierungen zuordnen will. In diesem Fall sind wir, wie im zweiten Beispiel zu sehen ist, gezwungen, die verschiedenen Instanzen eindeutig zu spezifizieren. Dies erfolgt über den, durch die Marken, die jeder **`for...generate`**-

und **`if...generate`**-Anweisung sowie jeder eigentlichen Instanziierung einer Komponente vorstehen müssen, eindeutig definierten Pfad zu der einzelnen Instanz. So sehen wir in dem obigen zweiten Beispiel, dass der Instanz der Entity `flipflop`, die über den Pfad `G0` → `G1` → `BIT3` in der Architektur `RTL2` erreicht wird, die Architektur `D_flipflop` zugewiesen wird. Den übrigen Instanzen der Entity `flipflop` der Architektur `RTL2` wird die Architektur `master_slave` der Entity `flipflop`, die ebenfalls in der Bibliothek `work` abgespeichert ist, zugewiesen. Hierfür steht die **`for others`**-Anweisung zur Verfügung.

Noch komplizierter wird es, wenn die Hierarchie der (strukturellen) VHDL-Beschreibung tiefer als 1 ist. Wir wollen dies an einem 8-Bit-Schieberegister illustrieren. Schematic, **`entity`** und **`architecture`** des Bausteins sind in Abbildung 3.6 zu sehen. In dieser Architektur ist das 8-Bit-Schieberegister aus zwei 4-Bit-Schieberegistern aufgebaut, die, wie wir gesehen haben, selbst wieder über eine strukturelle Beschreibung beschrieben und über Flipflops aufgebaut sind. Die Hierarchie dieser strukturellen Architektur hat in diesem Sinne Tiefe 2.




```
4   entity schieberegister_8 is
5     port ( clockpin, resetpin, data_in : in std_logic;
6            data_out : out std_logic );
7   end schieberegister_8;
8
9   architecture RTL of schieberegister_8 is
10    component schieberegister_4
11    port ( clockpin, resetpin, data_in: in std_logic;
12           data_out: out std_logic );
13    end component;
14    signal T: std_logic;
15  begin
16    INSTANZ1: schieberegister_4
17      port map (clockpin => clockpin, resetpin=>resetpin,
18                data_in => data_in, data_out => T);
19    INSTANZ2: schieberegister_4
20      port map (clockpin => clockpin, resetpin=>resetpin,
21                data_in => T, data_out => data_out);
22  end RTL;
```

Abbildung 3.6: 8-Bit-Schieberegister aufgebaut aus zwei 4-Bit-Schieberegistern

Hier reicht es nun offensichtlich nicht, in der dazugehörigen Konfiguration anzugeben, welche Architekturen den beiden Instanzen der Komponente `schieberegister_4` zugeordnet werden sollen, da diese Architekturen wiederum Instanzen von Komponenten, in diesem Falle der Komponente `flipflop`, enthalten, denen ebenfalls wieder Realisierungen zugeordnet werden müssen.

Hier stellt nun VHDL zwei Alternativen zur Verfügung, wobei letztere in unseren Augen zu bevorzugen ist. Wir wollen die beiden Möglichkeiten für den Fall illustrieren, dass den Instanzen der Komponente `schieberegister_4`, die in der in Abbildung 3.6 gezeigten Architektur `RTL` des Bausteins `schieberegister_8` vorkommen, die Architektur `RTL2` zugeordnet wird.

- Als Erstes können wir so vorgehen, dass wir rekursiv den einzelnen Instanzen der Komponenten in der bekannten Art und Weise Architekturen zuordnen. Dies resultiert in einer Konfiguration mit etwa dem folgenden Aussehen.

```
24 configuration CFG1 of schieberegister_8 is
25   for RTL
26
27     for INSTANZ1: schieberegister_4
28       use entity work.schieberegister_4(RTL2);
29       for RTL2
30         for G0
31           for G1
32             for BIT3: flipflop
33               use entity work.flipflop(D_flipflop);
34             end for;
35           end for;
36         end for;
37         for others: flipflop
38           use entity work.flipflop(master_slave);
39         end for;
40       end for;
41     end for;
42
43     for INSTANZ2: schieberegister_4
44       use entity work.schieberegister_4(RTL2);
45       for RTL2
46         for all: flipflop
47           use entity work.flipflop(master_slave);
48         end for;
49       end for;
50     end for;
51
52   end for;
53 end CFG1;
```

Der Nachteil dieses Ansatzes besteht darin, dass dem Designer einer hierarchisch aufgebauten Architektur die gesamte Hierarchie detailliert bekannt sein muss. Änderungen an Komponenten innerhalb der Hierarchie bedingen Änderungen in dem

Konfigurationsblock des obersten Bausteins. Bei größeren Entwürfen kann dies zu einem nicht unerheblichen Mehraufwand an Zeit führen.

- Eine effizientere Art besteht darin, die in einer Architektur eines Bausteins vorkommenden Instanzen einer Komponente nicht an eine Architektur dieser Komponente zu binden, sondern an eine Konfiguration dieser Architektur. Hierzu stellt VHDL neben der **`use entity`**- die **`use configuration`**-Anweisung zur Verfügung.

```
55  configuration CFG2 of schieberegister_8 is
56    for RTL
57
58      for INSTANZ1: schieberegister_4
59        use configuration work.CFG2;
60      end for;
61
62      for INSTANZ2: schieberegister_4
63        use configuration work.CFG1a;
64      end for;
65
66    end for;
67  end CFG2;
```

Durch eine solche **`use configuration`**-Anweisung sind automatisch auch alle tiefer vorkommenden Komponenten-Instanzen an eine Realisierung gebunden.

Bei diesen Ausführungen wollen wir es belassen — wohl wissend, dass wir einige in VHDL enthaltene Aspekte wie zum Beispiel *direkte Instanziierung von konfigurierten Komponenten* und *Belegung generischer Parameter während der Konfigurierung* nicht angesprochen haben. Für eine Einführung in VHDL sollten aber die hier besprochenen Ansätze ausreichen. Für weitere, vertiefende Ausführungen verweisen wir auf das Buch von Ashenden [3].

3.2 Variablen, Signale und Konstanten

Ein wichtiges Element zur Modellierung in VHDL sind Signale (und natürlich auch Variablen), die jeweils über einen Datentyp deklariert werden müssen. Signale modellieren Leitungen beziehungsweise Busse.

3.2.1 Deklaration von Variablen, Signalen und Konstanten

Die Deklaration von Variablen und Signalen ähnelt den Variablendeklarationen aus anderen Programmiersprachen. Insbesondere erlaubt VHDL auch eine Initialisierung der Variablen und Signale in der Deklaration, sodass eine Variablen- beziehungsweise Signaldeklaration das Aussehen

---

variable Variable_Name { **,** Variable_Name } **:**
Datentyp_Name [[**:=** Ausdruck_vom_Datentyp_der_Variablen]] **;**

signal Signal_Name { **,** Signal_Name } **:**
Datentyp_Name [[**:=** Ausdruck_vom_Datentyp_der_Signale]] **;**

---

hat. Mit einer ähnlichen Syntax kann man Konstanten deklarieren. In der Regel sieht eine solche Deklaration wie folgt aus:

---

constant Konstanten_Name **:**
Datentyp_Name **:=**Ausdruck_vom_Datentyp_der_Konstante **;**

3.2.2 Variablen- und Signalzuweisungen

Auf Variablen und Signale, insbesondere auf die Syntax einer Variablen- beziehungsweise Signalzuweisung, und auf ihre unterschiedliche Behandlung während der funktionalen Simulation sind wir in Abschnitt 2.2.1 detailliert eingegangen. Um diese Erläuterungen nicht nochmals an dieser Stelle wiederholen zu müssen, verweisen wir auf diesen Abschnitt und bauen im Folgenden auf die dort vermittelten Kenntnisse auf.

Damit können wir zur Vorstellung der Attribute, die auf Signalen definiert sind, zu sprechen kommen.

3.2.3 Signalbasierte Attribute

Auf jedem Signal sind Attribute definiert, die dem Designer Informationen über das Signal geben. In der Regel generieren diese Attribute aus dem Signal ein neues, meist zeitversetztes Signal. Die zur Verfügung stehenden Attribute eines Signals `s` sind im Folgenden beschrieben. Dabei entspricht `t` stets einem Wert des physikalischen Datentyps `time`:

- Das Attribut `s'delayed(t)`, welches ein Signal vom gleichen Datentyp wie Signal `s` darstellt, das die gleichen Werte wie das Signal `s` trägt, aber verzögert um `t` Zeiteinheiten.
- Das Attribut `s'stable(t)`, das ein Signal vom Datentyp `boolean` ist und genau dann den Wert `true` trägt, wenn sich der Wert des Signals `s` in den letzten `t` Zeitschritten nicht geändert hat. Ist `t` gleich 0 ns, so ist `s'stable(t)` genau dann `false`, wenn sich in dem aktuellen Simulationsschritt der Wert des Signals *s* geändert hat.

- Das Attribut `s'quiet(t)`, das ebenfalls ein Signal vom Datentyp `boolean` ist und genau dann gleich `true` ist, wenn während der letzten `t` Zeitschritte keine Zuweisung an das Signal `s` erfolgt ist — beachten Sie bitte, dass eine Zuweisung an ein Signal nicht unbedingt eine Änderung der Signalbelegung zur Folge hat, sodass die Attribute `s'stable(t)` und `s'quiet` in der Regel verschieden sind.

- Das Attribut `s'transaction`, das ein Signal vom Datentyp `bit` ist. Es ändert seinen Wert in jedem Simulationszyklus, in welchem dem Signal `s` ein Wert zugewiesen wird.

- Das Attribut `s'event`, das ein Wert vom Datentyp `boolean` ist, der genau dann gleich `true` ist, wenn das Signal `s` in dem aktuellen Simulationszyklus seinen Wert ändert.

- Das Attribut `s'active`, das ein Wert vom Datentyp `boolean` ist, der genau dann gleich `true` ist, wenn dem Signal `s` in dem aktuellen Simulationszyklus ein Wert zugewiesen wird. Im Unterschied zum Attribut `s'event` braucht sich die Belegung des Signals `s` also nicht zu ändern.

- Das Attribut `s'last_event`, das ein Wert vom Datentyp `time` ist, der die Zeit angibt, die vergangen ist, seitdem das Signal `s` das letzte Mal seinen Wert geändert hat. Ist bisher keine Wertänderung auf dem Signal erfolgt, so ist der Wert des Attributs gleich `time'high` (siehe Abschnitt 3.3.1).

- Das Attribut `s'last_active`, das ein Wert vom Datentyp `time` ist, der die Zeit angibt, die vergangen ist, seitdem dem Signal `s` das letzte Mal ein Wert zugewiesen wurde. Ist bisher keine Wertzuweisung an das Signal erfolgt, so ist der Wert des Attributs gleich `time'high` (siehe Abschnitt 3.3.1).

- Das Attribut `s'last_value`, das ein Wert vom gleichen Datentyp wie das Signal `s` ist. Es gibt den Wert zurück, das Signal `s` vor der letzten Wertänderung hatte. Ist bisher keine Wertänderung auf Signal erfolgt, so gibt das Attribut den Wert von `s` zurück.

- Die Attribute `s'driving` und `s'driving_value`, auf deren nähere Erläuterung wir aber in diesem Buch verzichten wollen, da es weit über das hinaus ginge, was man in einem einführenden Buch zu VHDL erklären sollte und kann.

Wir kommen nun zu den Grundkonzepten, die VHDL neben den gerade vorgestellten Konstrukten **`package`**, **`entity`**, **`architecture`** und **`configuration`** sowie Signalen und Variablen bereitstellt.

3.3 Datentypen und Subtypen

Lassen Sie uns mit den Datentypen beginnen.

Fallstrick 6

VHDL ist eine **streng getypte Programmiersprache**. Dies besagt, dass einer Variablen beziehungsweise einem Signal nur Werte aus dem Datentyp (oder einem seiner Subtypen) zugewiesen werden dürfen, über den sie beziehungsweise es deklariert worden ist.

Betrachten Sie zum Beispiel die beiden Datentypen

```
type wortbreite is range 1 to 64;
type width is range 1 to 64;
```

Es handelt sich hierbei um zwei *verschiedene* Datentypen. Eine Zuweisung des Wertes einer Variablen vom Datentyp `wortbreite` an eine Variable des Datentyps `width` ist nicht erlaubt. Eine automatische Typentransformation erfolgt nicht.

Ähnliches gilt für die im Package `standard` (siehe Abschnitt 5.2) definierten Typen `integer` und `real`. Sie dürfen in arithmetischen Ausdrücken zum Beispiel nicht addiert werden, es sei denn einer von beiden wird explizit in den anderen Typ transformiert.

Eine explizite Typtransformation (oder -konvertierung), die naturgemäß nur zwischen „ähnlichen“ Datentypen gemacht werden kann, erfolgt durch Voranstellen des Namens des Datentyps, in den konvertiert werden soll, gefolgt von dem, in Klammern eingerahmten, zu konvertierenden Wert:

Ziel_Datentyp **(** Wert **)**

Zum Beispiel konvertiert `integer(3.2)` den `real`-Wert `3.2` in den `integer`-Wert 3.

Welche Datentypen stellt VHDL nun zur Verfügung? In einem gewissen Sinne ist VHDL in dieser Beziehung recht nackt, da es an sich keinen spezifischen Datentyp für den Benutzer bereit hält. Vielmehr gibt uns VHDL Werkzeuge an die Hand, um uns die von uns benötigten Datentypen selbst zu definieren. Hierzu steht die **`type`**-Anweisung beziehungsweise die **`subtype`**-Anweisung zur Verfügung, die allgemein das Aussehen

---

`type` Bezeichner **`is`** Datentyp_Spezifikation **`;`**

---

beziehungsweise

`subtype` Bezeichner **`is`** Subtyp_Spezifikation **`;`**

---

haben.

Auf jedem Subtyp `T` (sogar auf jedem Datentyp `T`) ist das Attribut `T'base` definiert. `T'base` gibt den Basistyp an, über den der Datentyp `T` definiert ist.

Dieses Attribut darf nur im Zusammenhang mit anderen Attributen verwendet werden. Entsprechende Anwendungsbeispiele werden wir im Abschnitt über auf skalaren Datentypen definierte Attribute kennen lernen.

Prinzipiell unterscheidet VHDL zwischen den fünf Klassen

- *Skalare Datentypen*
- *Felder*
- *Zeiger*
- *Verbunde*
- *Files*

von Datentypen, wobei die skalaren Datentypen nochmals untergliedert sind in so genannte

- *Integer-Datentypen*
- *Real-Datentypen*
- *Aufzählungstypen*
- *Physikalische Datentypen*.

Wir wollen auf die Besonderheiten dieser Datentypen im Folgenden einzeln eingehen. Eigenschaften, die aus anderen Programmiersprachen wohl bekannt sind, werden wir nicht besprechen.

3.3.1 Skalare Datentypen

Skalare Daten sind solche Daten, die sich nicht in andere Daten unterteilen lassen. In der Regel geht man davon aus — und dies ist auch der Fall in VHDL —, dass auf den Werten eines skalaren Datentyps eine totale Ordnung $\leq$ definiert ist, sodass man die Werte entsprechend miteinander vergleichen kann.

VHDL erlaubt die wichtigsten Arten von skalaren Datentypen zu definieren: ganzzahlige und Gleitkommazahlen, Aufzählungstypen, sowie physikalische Datentypen. Letztere enthalten in ihren Wertebereichen ganzzahlige Zahlen, die mit einer (in der Regel physikalischen) Einheit versehen sind.

Integer-Datentypen

Integer-Datentypen sind Datentypen, die als Wertebereiche einen zusammenhängenden Bereich der ganzen Zahlen haben — in einem gewissen Sinne ist also jeder Integer-Datentyp ein Subtyp eines „anonymen“ Datentyps, der als Wertebereich „alle“ ganzen Zahlen enthält. Dieser anonyme Datentyp wird `universal_integer` genannt. Der Basistyp eines jeden Integer-Datentyps ist somit der Datentyp `universal_integer`.

Formal werden Integer-Datentypen über die Anweisungen

```
type Bezeichner is range Integer_Zahl to Integer_Zahl ;
type Bezeichner is range Integer_Zahl downto Integer_Zahl ;
```

definiert. Beispielsweise definiert

```
35    type monat_im_jahr is range 1 to 12;
```

einen Integer-Datentyp, der aus 12 ganzzahligen Werten besteht, nämlich aus den ganzen Zahlen 1 bis 12.

Auf Integer-Datentypen sind neben den Vergleichsoperatoren = (gleich), /= (verschieden), < (echt kleiner), <= (kleiner gleich), > (echt größer) und >= (größer gleich) die für ganze Zahlen üblichen Operationen

- Addition +,
- Subtraktion −,
- Multiplikation *,
- ganzzahlige Division /,
- Potenzierung **,
- Absolutwert **abs**,
- Modulo-Berechnung **mod**
- Rest bei der ganzzahligen Division **rem**

definiert.

Exkurs: Modulo-Berechnung und Rest-Bildung

Die Modulo-Berechnung $x \bmod y$ und die Bestimmung des Restes $x \operatorname{rem} y$ bei der ganzzahligen Division von x durch y sind im Falle von $y \in \mathbb{N}$ sehr ähnliche Operationen.

- In der Tat, ist x eine nichtnegative ganze Zahl, so unterscheiden sich beide Operationen nicht.
- Unterschiedliche Ergebnisse erhält man jedoch in der Regel, wenn x eine negative ganze Zahl ist. Während die Division gegen die Null rundet, also zum Beispiel $-19 \operatorname{div} 8 = -2$ und somit $-19 \operatorname{rem} 8 = -3$ gilt, bildet die Modulo-Berechnung den Wert x in eine der Restklassen $0, \ldots, y-1$ ab. In unserem Beispiel gilt somit $-19 \bmod 8 = 5$.

Beide Operationen, wie auch die Division selbst, können in der Regel nur dann durch Synthese-Werkzeuge synthetisiert werden, wenn der Divisor y eine Zweierpotenz ist, da in diesem Fall die Operationen im Wesentlichen durch einen Shift, also recht einfach, realisiert werden können.

Integer-Konstanten lassen sich neben der uns gebräuchlichen Schreibweise zur Basis 10 auch zu anderen Basen aus dem Bereich zwischen 2 und 16 in VHDL schreiben. Das allgemeine Format ist durch

---

Basis#Darstellung_zu_der_angegebenen_Basis#

---

gegeben. So kann der Integer-Wert 123 nicht nur durch

```
123,
```

sondern auch durch

```
2#1111011#,
```

durch

```
8#173#,
```

durch

```
16#7B#
```

und so weiter dargestellt werden.

Zusätzlich kann eine (lange) Integer-Konstante zur besseren Lesbarkeit zwischen je zwei Ziffern einen Unterstrich enthalten. In diesem Sinne sind die Schreibweisen

```
-2#111001010#,
```

```
-2#1_1100_1010#
```

und

```
-16#1CA#
```

alle drei erlaubt und beschreiben den gleichen Integer-Wert, nämlich den Wert −458. Negative Werte werden durch ein vorangestelltes Minus vor der Basis kenntlich gemacht.

VHDL stellt über die Bibliothek `standard` (siehe Abschnitt 5.2), die stets implizit bei einem Entwurf eingebunden ist,

- den Integer-Datentyp `integer`, sowie
- die beiden Subtypen `natural` und `positive` des Datentyps `integer` zur Verfügung, welche die nichtnegativen beziehungsweise die positiven Werte des Datentyps `integer` umfassen.

Der Bereich von $-2^{31}+1$ bis $2^{31}-1$ ist als Mindestanforderung für den Wertebereich des Datentyps `integer` an VHDL-Implementierungen vorgegeben. VHDL-Implementierungen dürfen und stellen auch zum Teil größere Bereiche — meist wird der von der 32-Bit Zweierkomplement-Zahlendarstellung verwendete Bereich -2^{31} bis $2^{31}-1$ bereitgestellt — für den Integer-Datentyp `integer` zur Verfügung gestellt. In diesem Zusammenhang sind die auf jedem skalaren Datentyp definierten Attribute `low` and `high` sehr nützlich. Sie geben den kleinsten beziehungsweise den größten Wert eines skalaren Datentyps zurück. So liefert uns `integer'low` den kleinsten Wert, `integer'high` den größten Wert des Integer-Datentyps `integer`.

Mehr zu Attributen im Allgemeinen finden Sie im Abschnitt 3.6 auf Seite 114. Eine Liste der auf skalaren Datentypen definierten Attribute finden Sie auf Seite 86.

Real-Datentypen

Analog zu den gerade vorgestellten Integer-Datentypen gibt es die so genannten Real-Datentypen, die jeweils einen zusammenhängenden Bereich von Gleitkommazahlen darstellen. Die Typdefinition eines Real-Datentyps hat im Wesentlichen das gleiche Aussehen wie das, was wir bei den Integer-Datentypen schon kennen gelernt haben:

---

type Bezeichner **is range** Gleitkommazahl **to** Gleitkommazahl **;**
type Bezeichner **is range** Gleitkommazahl **downto** Gleitkommazahl **;**

Auf ihnen sind ebenfalls die üblichen Vergleichsoperatoren und arithmetischen Operatoren definiert.

Real-Konstanten lassen sich unter Verwendung unterschiedlicher Basen aus dem Bereich 2 bis 16 in Gleitkommadarstellung schreiben. So stellen die Darstellungen `0.75`, `75.0e-2` und `7.5E1` (zur Basis 10) jeweils die gleiche Zahl 0.75 dar. Wählt man eine andere Basis, so wird eine ähnliche Schreibweise wie bei den Integer-Datentypen verwendet, wobei ein möglicher Exponent hinter das zweite `#`-Zeichen als Dezimalzahl geschrieben wird und als Exponent zur angegebenen Basis zu verstehen ist. So beschreibt die Darstellung `2#11.0#e-3` den Wert $3 \cdot 2^{-3}$, also den Wert 0.375.

Die implizit eingebundene Bibliothek `standard` (siehe Abschnitt 5.2) stellt den Real-Datentyp `real` zur Verfügung, der als Wertebereich[5] mindestens den Bereich von $-1.7014110 \cdot 10^{38}$ bis $1.7014110 \cdot 10^{38}$ umfasst. Die genauen Grenzen können, wie beim Datentyp `integer` oder anderen Integer-Datentypen, über die Attribute `real'low` und `real'high` festgestellt werden.

[5] Der Wertebereich des Datentyps `real` kann wie der Datentyp `integer` von der benutzten VHDL-Implementierung abhängen.

Physikalische Datentypen

Eine Besonderheit von VHDL im Vergleich zu anderen Programmiersprachen ist die Möglichkeit, physikalische Datentypen definieren zu können. Sie werden angewendet, um einfach mit physikalischen Größen, wie zum Beispiel Längen-, Gewichts-, Geschwindigkeits- und Zeitangaben und Ähnlichem arbeiten zu können. So ist jedem physikalischen Datentyp neben einem Bereich von Integer-Werten eine *primäre Einheit* zugeordnet, in der die Angaben gemessen werden. Zu der primären Einheit kann es weitere Einheiten, so genannte *sekundäre Einheiten*, geben, die ein Vielfaches der primären Einheit darstellen müssen.

Lassen Sie uns auf eine formale Definition der Syntax physikalischer Datentypen verzichten und uns einfach nur zwei, drei Beispiele anschauen.

Der Quellcode

```
36    type laenge_einfach is range 0 to 1E9
37      units km;
38    end units laenge_einfach;
```

definiert einen physikalischen Datentyp mit Namen `laenge_einfach`, für den keine sekundären Einheiten definiert sind, sondern nur eine primäre Einheit — die primäre Einheit `km`, die für Kilometer stehen soll. Der Zahlenbereich erstreckt sich von dem Integer-Wert 0 bis zum Integer-Wert 10^9. Dieser physikalische Datentyp ist an sich nichts anderes als ein Integer-Datentyp, der mit einer Einheit versehen ist; er würde den Vorteil, den physikalische Datentypen bieten, nämlich die „automatische Umrechnung" zwischen den definierten Einheiten, nicht nutzen und würde demnach auch keinen großen Sinn ergeben. Die eigentlichen Vorteile physikalischer Datentypen würde hingegen der physikalische Datentyp `laenge`

```
39    type laenge is range  -2147483647 to 2147483647
40      units nm;
41          um =  1000 nm;   -- Mikrometer
42          mm =  1000 um;   -- Millimeter
43          cm =    10 mm;   -- Centimeter
44           m =   100 cm;   -- Meter
45        Zoll = 25400 um;   -- Zoll oder Inch
46    end units laenge;
```

bieten.

Auf den physikalischen Datentypen sind die üblichen Vergleichsoperatoren sowie Addition, Subtraktion, Absolutwert und Division von physikalischen Werten definiert.

Fallstrick 7

Die Division zweier Werte eines physikalischen Datentyps ergibt einen Integer-Wert. Somit ist der Ausdruck

```
10 Zoll + (3 m / 10 mm)
```

nicht zulässig, da ein Wert vom Datentyp `laenge` mit einem Integer-Wert addiert werden würde. Dies ist in VHDL, die eine streng getypte Programmiersprache ist, nicht erlaubt.

Einen Integer-Wert kann man in einen physikalischen Wert, sagen wir in einen Wert vom physikalischen Datentyp `laenge`, umwandeln, d. h. konvertieren, indem man den Integer-Wert mit einem der physikalischen Werte multipliziert. In unserem Beispiel bietet sich hierfür der Wert `1 nm` an.

Physikalische Werte können demnach mit ganzen Zahlen multipliziert beziehungsweise dividiert werden.

In den „arithmetischen" Ausdrücken können wir verschiedene Einheiten benutzen. So ist zum Beispiel der Ausdruck

```
10 Zoll + (3 m / 2)
```

erlaubt. Der VHDL-Simulator rechnet alle physikalischen Operatoren eines solchen Ausdrucks auf die primäre Einheit des entsprechenden physikalischen Datentyps zurück — in unserem Beispiel auf `nm`.

Physikalische Werte miteinander zu multiplizieren, ist in VHDL *nicht* erlaubt. In der Tat, es macht auch keinen großen Sinn. Die Multiplikation von zwei Längeneinheiten ergibt eine Flächeneinheit und ist somit kein Wert aus dem Wertebereich des gegebenen physikalischen Datentyps. Will man dennoch physikalische Werte — was auch immer der Grund hierfür sein möge — miteinander multiplizieren, kann man dies über die, auf skalaren Datentypen definierten, Attribute `pos` und `val` realisieren. Lassen Sie uns diese beiden Attribute — eine vollständige Aufzählung der auf physikalischen Datentypen definierte Datentypen folgt in Abschnitt 3.3.1 — beispielhaft an unserem Datentyp `laenge` erklären.

- Das Attribut `laenge'pos(X)` angewendet auf einen Wert `X` des physikalischen Datentyps `laenge` gibt den dazu gehörigen Integer-Wert in Bezug auf die primäre Einheit zurück. Beispiel: `laenge'pos(3 cm)` ist gleich 30 000 000.

  Das Attribut kann somit als Datentypkonvertierungeines physikalischen Wertes in einen Integer-Wert benutzt werden.

- Das Attribut `laenge'val(X)` wird auf einen Integer-Wert `X` angewendet und gibt dann den entsprechenden Wert des physikalischen Datentyps `laenge` zurück, ist also in einem gewissen Sinne die inverse Funktion zu dem Attribut `laenge'pos()`.

VHDL stellt standardmäßig über die Library `standard` den physikalischen Datentyp `time` bereit. Näheres zu diesem speziellen Datentyp ist in Abschnitt 5.2.2 auf Seite 136 zu finden.

Aufzählungstypen

Die Aufzählungstypen sind die einfachsten skalaren Datentypen. Hier werden die in dem Wertebereich verfügbaren Werte in der Datentypdefinition einfach nur aufgezählt. Die Reihenfolge, in der das passiert, bestimmt die auf dem Datentyp definierte totale Ordnung $\leq$. Allgemein hat die Definition eines Aufzählungstyp das Aussehen

---

type Bezeichner **is** (Element { , Element }) ;

Beispiele von solchen Aufzählungstypen sind in den in Kapitel 5 vorgestellten Packages zu finden. Im Package `standard` (siehe Abschnitt 5.2 Seite 133) zum Beispiel, das implizit in VHDL eingebunden ist, werden die Aufzählungstypen

- `boolean`, der aus den Werten `false` und `true` mit `false`$\leq$`true` besteht,
- `bit`, der aus den Werten `'0'` und `'1'` mit `'0'`$\leq$`'1'` besteht,
- `character`, der die 128 Zeichen des Maschinenalphabets umfasst,
- `severity_level`, der aus den Werten `note`, `warning`, `error` und `failure` besteht; ein Datentyp, der im Rahmen der **`assert`**-Anweisung (siehe Abschnitt 3.4 und Abschnitt 3.5 auf den Seiten 110–112) von Bedeutung ist, sowie
- die Aufzählungstypen `file_open_kind` und `file_open_status`, die im Rahmen des Zugriffs auf Dateien benötigt werden (siehe Abschnitt 5.2.4),

definiert.

Das Package `stg_logic_1164` (siehe Abschnitt 5.3) stellt eine 9-Werte-Logik zur Verfügung, die über einen Aufzählungstyp namens `std_ulogic` (beziehungsweise `std_logic`) definiert ist und die es ermöglicht, Hardware realitätsnäher zu beschreiben und zu simulieren als dies mit dem Datentyp `bit` zum Beispiel möglich ist.

Anwendung finden Aufzählungstypen oft bei der Beschreibung von endlichen Automaten zur Festlegung der möglichen Zustände:

```
8     type state is (idle, start, stopp);
```

Auf skalaren Datentypen definierte Attribute

Auf jedem skalaren Datentyp `T` sind, neben dem, schon auf jedem Datentyp definierten Attribut `T'base`, das den Basistyp des Datentyps `T` angibt, weitere Attribute definiert, die dem Designer Informationen zu dem Datentyp geben. Unabhängig von der Art des skalaren Datentyps stehen die sieben Attribute

- `T'high`, welches den größten Wert des Datentyps `T` liefert,
- `T'low`, welches den kleinsten Wert des Datentyps `T` liefert,

- `T'left`, das den Wert der linken Bereichsgrenze des Datentyps `T` angibt,
- `T'right`, das den Wert der rechten Bereichsgrenze des Datentyps `T` angibt,
- `T'ascending`, das vom Datentyp `boolean` ist und genau dann gleich dem Wert `true` ist, wenn der zu `T` gehörige Wertebereich aufsteigend ist, also über eine **`to`**-Anweisung (und nicht über eine **`downto`**-Anweisung) definiert wurde,
- `T'image(X)`, das eine Funktion darstellt, die angewendet auf einen Wert `X` aus dem Wertebereich von `T` einen Wert vom Datentyp `string` (siehe Abschnitt 5.2), der eine textuelle Darstellung des Wertes `X` ist, zurückgibt.
- `T'value(X)`, das, informal ausgedrückt, die zu der Funktion `T'image()` inverse Funktion darstellt. Angewendet auf eine Zeichenfolge `X` versucht diese Funktion diese Zeichenfolge als Wert des Datentyps `T` zu interpretieren und gibt diesen Wert zurück.

zur Verfügung.

Ist `T` ein Aufzählungstyp, so gilt `T'low=T'left` und `T'high=T'right`. Bei einem Integer-Datentyp `T` gelten diese beiden Gleichungen nur dann, wenn das Attribut `T'ascending` wahr ist.

Nur auf Aufzählungstypen, Integer-Datentypen und physikalischen Datentypen sind die folgenden Attribute definiert:

- Das Attribut `T'pos(X)`, das eine Funktion darstellt, die, angewendet auf einen Wert `X` aus dem Wertebereich des Datentyps `T`, die Position des Wertes `X` im Datentyp `T` zurückgibt. Der zurückgegebene Wert ist von dem speziellen Datentyp `universal_integer`.

  Der Begriff „Position“ ist in Abhängigkeit der Art des skalaren Datentyps unterschiedlich definiert:
  - Die Position eines Wertes `X` eines Aufzählungstyps ist als die Stelle definiert, an der der Wert `X` in der totalen Ordnung des Aufzählungstyps steht. Der erste in der totalen Ordnung vorkommende Wert hat Position 0.
  - Die Position eines Wertes `X` eines Integer-Datentyps ist der Wert `X` selbst — jedoch als Wert des Datentyps `universal_integer` interpretiert.
  - Die Position eines Wertes `X` eines physikalischen Datentyps ist der Integer-Wert des Wertes `X`, ausgedrückt über die primäre Einheit dieses physikalischen Datentyps.
- Das Attribut `T'val(X)`, das eine Funktion darstellt, die, angewendet auf einen Integer-Wert `X`, den `X`-ten Wert aus dem Wertebereich des Datentyps `T` zurückgibt.
- Das Attribut `T'succ(X)`, das eine Funktion darstellt, die, angewendet auf einen Wert `X` aus dem Wertebereich des Datentyps `T`, den Wert aus dem Wertebereich des Basistyps von `T` zurückgibt, der in der totalen Ordnung des Basistyps von `T` dem Wert *X* direkt folgt. Es gilt also

  `T'base'pos(T'succ(X))=T'base'pos(X)+1`.

Ist `X` schon das größte Element `T'base'high` aus dem Basistyp von `T`, so ist der Funktionswert `T'succ(X)` nicht definiert.

- Das Attribut `T'pred(X)`, das eine Funktion darstellt, die, angewendet auf einen Wert `X` aus dem Wertebereich des Datentyps `T`, den Wert aus dem Wertebereich des Basistyps von `T` zurückgibt, der in der totalen Ordnung des Basistyps von `T` genau vor dem Wert *X* steht. Es gilt also

  `T'base'pos(T'pred(X)) = T'base'pos(X)-1.`

  Ist `X` schon das kleinste Element `T'base'low` aus dem Basistyp von `T`, so ist der Funktionswert `T'pred(X)` nicht definiert.

- Das Attribut `T'leftof(X)`, das eine Funktion darstellt, die, angewendet auf einen Wert `X` aus dem Wertebereich des Datentyps `T`, den links von ihm stehenden Wert im Wertebereich des Basistyps von `T` zurückgibt.

  Ist `X` schon das am weitesten links stehende Element `T'base'left` aus dem Basistyp von `T`, so ist der Funktionswert `T'leftof(X)` nicht definiert.

- Das Attribut `T'rightof(X)`, das eine Funktion darstellt, die, angewendet auf einen Wert `X` aus dem Wertebereich des Datentyps `T`, den rechts von ihm stehenden Wert im Wertebereich des Basistyps von `T` zurückgibt.

  Ist `X` schon das am weitesten rechts stehende Element `T'base'right` aus dem Basistyp von `T`, so ist der Funktionswert `T'rightof(X)` nicht definiert.

3.3.2 Zusammengesetzte Datentypen

Lassen Sie uns damit zu den so genannten zusammengesetzten Datentypen kommen. Die von VHDL bereitgestellten Möglichkeiten, Zeiger zu deklarieren, wollen wir in diesem Abschnitt ebenfalls behandeln — auch wenn es sich hierbei nicht um einen zusammengesetzten Datentyp handelt —, da sie (fast inhärent) eng mit Verbunden (engl.: *records*) zusammenhängen.

Felder

Felder (engl.: *arrays*) setzen sich in VHDL, wie in jeder anderen Programmiersprache auch, aus Elementen des gleichen Datentyps zusammen. Es gibt in VHDL sowohl eindimensionale Felder, d. h. Felder, bei denen die Komponenten über nur einen Index indiziert sind, als auch mehrdimensionale Felder mit mehreren Indizes. Hierbei unterscheidet VHDL zwischen so genannten beschränkten (engl.: *constrained*) Feldern und unbeschränkten (engl.: *unconstrained*) Feldern.

Beschränkte Felder

Beschränkte Felder sind dadurch gekennzeichnet, dass der Wertebereich der Indizes durch Angabe einer unteren und oberen Grenze fest vorgegeben ist. Eine entsprechende Typdefinition sieht bei eindimensionalen Feldern (in der Regel) wie folgt aus:

---

type Bezeichner **is array**
(Diskreter_Werte_Bereich) **of** Datentyp_Name **;**

Bei mehrdimensionalen beschränkten Feldern werden die Wertebereiche einfach nur durch jeweils ein Komma getrennt.

---

type Bezeichner **is array**
(Diskreter_Wertebereich { **,** Diskreter_Wertebereich })
of Datentyp_Name **;**

Als Beispiel seien hier angeführt

```
47    type byte is array (7 downto 0) of bit;
48    type wort is array (1 to 4) of byte;
49    type ram is array (0 to 1023) of wort;
50    type brett is array (positive range 1 to 9,
51      positive range 1 to 9) of integer;
```

oder der in der Implementierung des Packages `std_logic_1164` benutzte Feld-Datentyp

```
166   type stdlogic_table is array(std_ulogic, std_ulogic)
167     of std_ulogic;
```

der als Wertebereich der beiden Indizes den Aufzählungstyp `std_ulogic` hat.

Komponentenweise Zugriffe oder Belegungen erfolgen durch Angabe des Bezeichners (Name der Variablen, des Signals oder der Konstante), gefolgt von den, in runden Klammern eingeschlossenen Indexwerten. Der Zugriff auf die Komponente `173` der Variable `ram_var` erfolgt also durch `ram_var(173)`. Für den feldweisen Zugriff oder die feldweise Belegung stehen mehrere Methoden zur Verfügung. Im Codefragment

```
98    byte_a <= ('0','0','0','1','1','1','1','0');
99    byte_b <= ( 4 | 3 | 2 | 1 => '1', others => '0');
100   byte_c <= ( 4 downto 1 => '1', others => '0');
```

in dem `byte_a`, `byte_b`, `byte_c` Signale vom Datentyp `byte` darstellen, sind einige mögliche Kombinationen gezeigt.

Die Belegung von mehrdimensionalen Feldern ist ähnlich einfach. Wie das Beispiel in Abbildung 3.7, in dem eine Konstante namens `kleines_Einmaleins` von dem oben definierten Datentyp `brett` definiert wird, zeigt, können zweidimensionale Felder — der erste Index eines zweidimensionalen Feldes steht für die Zeilen, der zweite Index für die Spalten — „zeilenweise“ belegt werden, d. h. man gibt zuerst den zur ersten Zeile

```
57   -- Funktionstafel des kleinen Einmaleins
58   constant kleines_Einmaleins : brett := (
59     --
60     --     1   2   3   4   5   6   7   8   9
61     --  ---------------------------------------------
62         (  1,  2,  3,  4,  5,  6,  7,  8,  9 ),  -- 1
63         (  2,  4,  6,  8, 10, 12, 14, 16, 18 ),  -- 2
64         (  3,  6,  9, 12, 15, 18, 21, 24, 27 ),  -- 3
65         (  4,  8, 12, 16, 20, 24, 28, 32, 36 ),  -- 4
66         (  5, 10, 15, 20, 25, 30, 35, 40, 45 ),  -- 5
67         (  6, 12, 18, 24, 30, 36, 42, 48, 54 ),  -- 6
68         (  7, 14, 21, 28, 35, 42, 49, 56, 63 ),  -- 7
69         (  8, 16, 24, 32, 40, 48, 56, 64, 72 ),  -- 8
70         (  9, 18, 27, 36, 45, 54, 63, 72, 81 )   -- 9
71   );
```

Abbildung 3.7: Deklaration und Definition eines zweidimensionalen Feldes als Konstante

gehörigen eindimensionalen Vektor an, dann den zur zweiten Zeile gehörigen und so weiter. In unserem Beispiel gilt für alle Werte `i` und `j` aus dem Wertebereich 1 bis 9

```
kleines_Einmaleins(i,j) = i*j.
```

VHDL stellt verschiedene Möglichkeiten zur Verfügung, um so genannte *Bitfolgen* bequem schreiben zu können. Ursprünglich nur für Felder über dem Datentyp `bit` gedacht, sind diese Schreibweisen seit VHDL'93 für alle eindimensionalen Arrays erlaubt, die über einem Aufzählungstyp definiert sind und die Zeichen `'0'` und `'1'` enthalten.

Bitfolgen kann man in binärer, in oktaler oder in hexadezimaler Darstellung schreiben. Die Darstellungsart wird durch den Präfix, der entweder `B`, `O` oder `X` ist, angegeben. Die Darstellung selbst folgt dann als endliche Folge über $\{0,1\}$, über $\{0,1,\ldots,7\}$ beziehungsweise über $\{0,1,\ldots,9,A,B,C,D,E,F\}$ umrahmt mit dem `"`-Zeichen.

So stellen zum Beispiel

- `O"4731"` die Bitfolge `"100111011001"`

und

- `X"9DA"` die Bitfolge `"100111011010"`

dar.

Zu beachten ist, dass eine oktal dargestellte Bitfolge immer ein (eindimensionales) Feld darstellt, dessen Länge ein Vielfaches von 3 ist; eine hexadezimal dargestellte Bitfolge immer ein Feld, dessen Länge ein Vielfaches von 4 ist.

Unbeschränkte Felder

Bei unbeschränkten Feldern legt man sich bei der Definition des Datentyps nicht auf genaue Grenzen für den Index beziehungsweise die Indizes fest, sondern verschiebt die

Festlegung auf später, zum Beispiel auf den Zeitpunkt der Variablendeklaration. So werden in dem Package `standard` (siehe Abschnitt 5.2) die Datentypen

```
65    type string is array (positive range <>) of character;
66    type bit_vector is array (natural range <>) of bit;
```

und in dem Package `std_logic_1164` (siehe Abschnitt 5.3) die Datentypen

```
20    type std_ulogic_vector is array ( natural range <> )
21      of std_ulogic;
```

und

```
37    type std_logic_vector is array ( natural range <>)
38      of std_logic;
```

zur Verfügung gestellt. Sie definieren jeweils unbeschränkte Felder. Die Grenzen der Indexbereiche sind nicht durch die Definition des Datentyps festgelegt.

Die formale Syntax, um unbeschränkte Felder zu definieren, ist durch

---

type Bezeichner **is array**
(Diskreter_Datentyp **range <>** { **,** Diskreter_Datentyp **<>** })
of Datentyp_Name **;**

---

gegeben.

Unbeschränkte Felder finden ihre Anwendung

- in Packages, wo sie es ermöglichen, von der Größe der Wertebereiche unabhängige Felder vorzudefinieren. Mit den im Package `std_logic_1164` definierten Datentypen `std_ulogic_vector` und `std_logic_vector` haben wir gerade zwei solche Beispiele erwähnt.
- Als formale Parameter von Funktionen und Prozeduren (siehe Kapitel 4). Hier werden die exakten Wertebereiche erst durch die aktuellen Parameter festgelegt.
- Als Ports von Bausteinen, wobei sie naturgemäß dann nicht nur im Rahmen der **entity**-Definition, sondern auch in den dazugehörigen Architekturen benutzt werden. Erst bei der Instanziierung einer Komponente dieses Bausteins werden den unbeschränkten Ports beschränkte Felder zugewiesen. Ein ausführliches Beispiel einer solchen Schnittstellenbeschreibung (mit zugehöriger Architektur) ist in Abschnitt 3.5 auf Seite 113 zu finden.

Wir wollen die Effektivität unbeschränkter Felder an einer Prozedur illustrieren, die eine n-Bit Zweierkomplement-Zahlendarstellung in eine m-Bit-Zweierkomplement-Darstellung mit $m \geq n$ umwandelt. Wie in den Exkursen zu Zweierkomplement-Darstellung und

Vorzeichenverdopplung in Kapitel 6 noch ausführlich erläutert wird, ist zur Realisierung einer solchen Erweiterung das Vorzeichen der n-Bit-Zweierkomplement-Darstellung $m - n$ Mal zu kopieren und der n-Bit-Zweierkomplement-Darstellung voranzustellen. Abbildung 3.8 zeigt eine der möglichen Realisierungen.

```
19    procedure sign_extend (
20      inp: in std_logic_vector;
21      result: out std_logic_vector ) is
22    begin
23      assert inp'length <= result'length
24        report "Der Ausgabevektor ist zu kurz!"
25        severity failure;
26      assert (inp'right = 0) and (result'right = 0)
27        report "Attribut 'right muss jeweils Index 0 liefern!"
28        severity failure;
29      result(inp'length-1 downto 0) := inp;
30      for i in inp'length to result'length-1 loop
31        result(i) := inp(inp'length-1);
32      end loop;
33    end;
```

Abbildung 3.8: Prozedur zur Transformation einer Zweierkomplement-Darstellung auf einer größere Wortbreite

Wie man sieht, wird in der Definition der Prozedur `sign_extend` weder die Länge des formalen Eingabeparameters `inp` — über das Schlüsselwort **`in`** wird festgelegt, dass es sich bei diesem Parameter um einen Eingabeparameter handelt — noch die Länge des formalen Ausgabeparameters `result` festgelegt. Erst bei Aufruf der Prozedur wird über die aktuellen Parameter die Länge dieser beiden Felder bestimmt. Um etwaige Fehler, die letztendlich auf die durch die Nicht-Festlegung der Bereichsgrenzen der formalen Parameter bedingten Freiheiten zurückzuführen sind, während der Laufzeit zu finden — in unserem Beispiel sollte der Ausgabeparameter nicht kürzer als der Eingabeparameter sein; zudem sollten die Felder von rechts her mit 0 beginnend aufwärts indiziert sein — sollte eine solche Prozedur überprüfen, ob die aktuellen Parameter den Voraussetzungen genügen. Dies erfolgt in der Regel über die auf Feldern definierten Attribute. In unserer Beispielprozedur erhalten wir die Längen der beiden Felder über die Attribute `inp'length` und `result'length`. Ist der erste Wert echt größer als der zweite Wert, so wird über die **`assert`**-Anweisung eine Fehlermeldung ausgegeben. Über die Attribute `inp'right` und `result'right` überprüfen wir, ob die rechten Komponenten der beiden Feldern den Index 0 haben.

Attribute auf Feldern

Lassen Sie uns detaillierter auf die auf Feldern definierten Attribute zu sprechen kommen. Sei hierzu `A` ein über einem Datentyp `T` definiertes k-dimensionale Feld und `N` ein beliebiger Integer-Wert aus dem Bereich von 1 bis k. Folgende auf Feldern definierte Attribute stellt uns VHDL zur Verfügung:

- Das Attribut `A'length(N)`, das uns die Größe des `N`-ten Indexbereiches von `A` angibt.
- Das Attribut `A'left(N)`, das den linken Rand des `N`-ten Indexbereiches von `A` angibt; das Attribut ist also von dem gleichen Datentyp wie die Indizes der `N`-ten Dimension des Feldes `A`.
- Das Attribut `A'right(N)`, das den rechten Rand des `N`-ten Indexbereiches von `A` angibt; das Attribut ist also ebenfalls von dem gleichen Datentyp wie die Indizes der `N`-ten Dimension des Feldes `A`.
- Das Attribut `A'low(N)`, das den kleinsten Wert des `N`-ten Indexbereiches von `A` angibt; dieser Wert ist je nachdem, ob in der Definition des Datentyps von `A` **`downto`** oder **`to`** zur Spezifikation des Indexbereiches benutzt wurde, gleich dem Wert `A'right` oder dem Wert `A'left`.
- Das Attribut `A'high(N)`, das den größten Wert des `N`-ten Indexbereiches von `A` angibt. Dies ist je nachdem, ob in der Definition des Datentyps von `A` **`downto`** oder **`to`** zur Spezifikation des Indexbereiches benutzt wurde, gleich dem Wert `A'left` oder dem Wert `A'right`.
- Das Attribut `A'range(N)`, das den Wertebereich des `N`-ten Indexbereiches von `A` angibt. Dies erlaubt uns auch bei unbeschränkten Feldern alle Komponenten zu durchlaufen. Wir werden, nachdem wir uns das nächste Attribut angesehen haben, ein kleines Beispiel dazu angeben.
- Das Attribut `A'reverse_range(N)`, das den gespiegelten Wertebereich des `N`-ten Indexbereiches von `A` angibt. Ist `A` zum Beispiel ein zweidimensionales Feld, so durchforstet der Programmcode

```
122     for i in A'range(1) loop
123        for j in A'reverse_range(2) loop
124          report positive'image(A(i,j));
125        end loop;
126     end loop;
```

  Das Feld `A`, wobei der Indexbereich der ersten Dimension in der Reihenfolge beginnend mit `A'left(1)` hin zu `A'right(1)` und der Indexbereich der zweiten Dimension in umgekehrter Reihenfolge, also von `A'right(2)` hin zu `A'left(2)` durchlaufen wird.
- Das Attribut `A'ascending(N)`, das vom Typ `boolean` ist und genau dann gleich `true` ist, wenn der `N`-te Indexbereich monoton steigend ausgelegt ist, d. h. über das Schlüsselwort **`to`** (und nicht über das Schlüsselwort **`downto`**) definiert wurde.

Der Parameter N ist optional. Defaultmäßig ist er mit dem Wert 1 vorbelegt.

Einige der Attribute sind sehr nahe miteinander verwandt. So gilt beispielsweise

```
A'ascending(N) = true  ⟹  A'left(N) = A'low(N)
A'ascending(N) = true  ⟹  A'right(N) = A'high(N)
```

```
A'ascending(N) = false  ⟹  A'left(N) = A'high(N)
A'ascending(N) = false  ⟹  A'right(N) = A'low(N).
```

Ist der `N`-te Indexbereich über Integer-Werte definiert, so gilt desweiteren

```
A'length = A'high(N) − A'low(N) + 1.
```

Operationen auf Feldern

Die in VHDL zur Verfügung gestellten Operationen auf Feldern beziehen sich alle nur auf eindimensionale Felder, die meisten sogar nur auf eindimensionale Felder, die über dem Datentyp `bit` oder dem Datentyp `boolean` definiert sind. Es sind dies

- Die binären logischen Operationen **`and`**, **`or`**, **`nand`**, **`nor`**, **`xor`**, und **`xnor`**, die nur auf eindimensionalen Feldern über dem Datentyp `boolean` oder `bit` gleicher Länge definiert sind. Ergebnis der Operationen ist jeweils die komponentenweise ausgeführte logische Basisoperation.

- Die Shift- und Rotate-Operatoren
  - Der Operator **`sll`**, der, angewendet auf einen Vektor vom Datentyp `bit` oder `boolean`, diesen um eine bestimmte Anzahl von Stellen nach links schiebt (engl.: *shift-left logical*). Rechts werden die Stellen mit dem Wert `'0'` beziehungsweise `false` aufgefüllt. So gilt zum Beispiel

    ```
    B"10110011" sll 2 = B"11001100".
    ```

  - Der Operator **`srl`**, der, angewendet auf einen Vektor vom Datentyp `bit` oder `boolean`, diesen um eine bestimmte Anzahl von Stellen nach rechts schiebt (engl.: *shift-right logical*). Links werden die Stellen mit dem Wert `'0'` beziehungsweise `false` aufgefüllt. So gilt zum Beispiel

    ```
    B"10110011" srl 2 = B"00101100".
    ```

  - Der Operator **`sla`**, der, angewendet auf einen Vektor vom Datentyp `bit` oder `boolean`, diesen um eine bestimmte Anzahl von Stellen nach links schiebt (engl.: *shift-left arithmetic*). Rechts werden die Stellen mit dem Wert aufgefüllt, der ursprünglich an der hinteren Stelle des Vektors stand. So gilt zum Beispiel

    ```
    B"10110011" sla 2 = B"11001111".
    ```

  - Der Operator **`sra`**, der, angewendet auf einen Vektor vom Datentyp `bit` oder `boolean`, diesen um eine bestimmte Anzahl von Stellen nach rechts schiebt (engl.: *shift-right arithmetic*). Links werden die Stellen mit dem Wert aufgefüllt, der ursprünglich an der vorderen Stelle des Vektors stand. So gilt zum Beispiel

    ```
    B"10110011" sra 2 = B"11101100".
    ```

  - Der Operator **`rol`**, der, angewendet auf einen Vektor vom Datentyp `bit` oder `boolean`, diesen um eine bestimmte Anzahl von Stellen nach links rotiert (engl.: *rotate-left*). Die links „herunter fallenden“ Werte werden rechts wieder angehängt. So gilt zum Beispiel

    ```
    B"10110011" rol 2 = B"11001110".
    ```

- Der Operator **ror**, der, angewendet auf einen Vektor vom Datentyp `bit` oder `boolean`, diesen um eine bestimmte Anzahl von Stellen nach rechts rotiert (engl.: *rotate-right*). So gilt zum Beispiel

  ```
  B"10110011" ror 2 = B"11101100".
  ```

  Während der linke Operand bei den Shift- und Rotate-Operationen der zu verändernde Vektor ist, gibt der rechte Operand die Anzahl der Stellen an, um die geschoben beziehungsweise rotiert werden soll. VHDL erlaubt, dass dieser rechte Operand auch eine negative ganze Zahl sein darf. In diesem Fall wird ein Richtungswechsel vollzogen, d. h. ein Links-Shift wird zu einem Rechts-Shift, Links-Rotation wird zu Rechts-Rotation und umgekehrt.

- Die Vergleichsoperatoren **=**, **/=**, **<**, **<=**, **>** und , die zwei über den gleichen Datentyp definierte eindimensionale Felder lexikografisch miteinander vergleichen.

- Die Konkatenation **&** zweier Felder, die eindimensionale, über dem gleichen Datentyp definierte Felder zu einem Feld verschmelzt. So beschreibt der Quellcode

```
9   procedure unsigned2signed (
10    inp: in std_logic_vector;
11    result: out std_logic_vector ) is
12  begin
13    assert inp'length = result'length-1
14      report "Die Vektoren haben falsche Längen"
15      severity failure;
16    result := '0' & inp;
17  end;
```

  eine Prozedur, die eine Binärzahl (ohne Vorzeichen) in eine vorzeichen-behaftete Darstellung überführt.

Damit wollen wir mit den **array**-Datentypen schließen und uns Verbunde und Zeiger anschauen.

Verbunde

Neben (beschränkten und unbeschränkten) Feldern stellt VHDL, wie die meisten anderen Programmiersprachen auch, noch eine zweite Klasse von zusammengesetzten Datentypen zur Verfügung: die *Verbunde* (engl.: *records*). Im Unterschied zu Feldern, deren Komponenten alle vom gleichen Typ sind, fasst ein Verbund Werte unterschiedlicher Datentypen in einem Objekt zusammen. So definieren wir beziehungsweise deklarieren wir mit

```
9     type ethernet_frame_format is
10      record
11        praeambel: std_logic_vector(0 to 63);
12        zieladresse: std_logic_vector(47 downto 0);
13        quelladresse: std_logic_vector(47 downto 0);
14        schluessel: std_logic_vector(0 to 15);
15        nachricht: std_logic_vector(1 to 368);
16        crc: std_logic_vector(31 downto 0);
17      end record;
18    signal ethernet_paket: ethernet_frame_format;
```

einen Datentyp, dessen Werte Ethernet-Pakete darstellen, und ein Signal namens `ethernet_paket` von diesem Datentyp. Ein Ethernet Paket besteht aus sechs Komponenten, die als Felder unterschiedlicher Länge über dem Datentyp `std_logic` deklariert sind[6]. Es sind dies die Komponenten

- `praeambel`, die 64 Bit lang ist und alternierend aus ’0’ und ’1’ besteht; sie erlaubt dem Empfänger, sich mit dem ankommenden Signal zu synchronisieren,
- `zieladresse` und `quelladresse`, zwei 48-Bit Adressen, die Ethernet-Adressen darstellen und den Empfänger und den Sender des Pakets angeben,
- `schluessel`, eine 16-Bit-Folge, welche entweder ein high-level-Protokoll angibt oder die Länge der eigentlichen Nachricht codiert,
- `nachricht`, die eigentliche Nachricht, die in einer Folge der Länge 348 abgespeichert wird — das IEEE-Format 802.3 schreibt vor, dass die Nachricht mindestens 368 Bit und höchstens 12 000 Bit lang sein darf — und
- `crc`, eine 32-Bit CRC-Prüfsumme zum Erkennen von Übertragungsfehlern.

Der Zugriff auf eine einzelne Komponente eines Verbundes erfolgt über den Namen der Variablen, des Signals oder der Konstante gefolgt von einem Punkt und dem Namen der Komponente. So bezeichnet

```
ethernet_paket.zieladresse
```

die Komponente `zieladresse` des Signals `ethernet_paket` und die Anweisung

```
26    ethernet_paket.zieladresse <= X"08002BE4B102";
```

weist dem in dem Signal `ethernet_paket` abgespeicherten Paket eine Zieladresse zu.

Wie bei Feldern kann ein Verbund auch als Ganzes beschrieben werden. Hier bieten sich wie bei Feldern verschiedene Möglichkeiten in VHDL an, beispielsweise die Zuordnung der Werte zu den Komponenten über die Position,

```
28    ethernet_paket <=
29     ( syn, ziel, quelle, key, nachrichtentext, pruefsumme );
```

6 Eine schöne Einführung zu der Thematik der Computer-Netzwerke findet man in [38].

wobei `syn`, `ziel`, `quelle`, `key`, `nachrichtentext` und `pruefsumme` entsprechend definierte Vektoren sind. Die Zuordnung kann auch über die Namen der Komponenten erfolgen.

```
31     ethernet_paket <=
32      ( quelladresse => quelle, nachricht => nachrichtentext,
33        zieladresse => ziel, schluessel => key,
34        praeambel => syn, crc => pruefsumme );
```

Die Verwendung des Schlüsselwortes **`others`**

```
36      ethernet_paket <=
37      ( nachricht => nachrichtentext, schluessel => key,
38        praeambel => syn, crc => pruefsumme,
39        others => X"000000000000" );
```

steht ebenfalls zu Verfügung und weist allen noch nicht belegten Komponenten *einen* Wert zu. Das soeben verwendete **`others`**-Konstrukt übergibt an die beiden Komponenten `zieladresse` und `quelladresse` des Verbundes `ethernet_paket` den Null-Vektor.

Fallstrick 8

Der Operator **`others`** kann nur dann angewendet werden, wenn die restlichen Komponenten alle vom gleichen Datentyp sind.

3.3.3 Zeiger in VHDL

VHDL stellt, wie die meisten anderen bekannten Programmiersprachen auch, Zeiger zur Verfügung, die in VHDL **`access`**-Variablen beziehungsweise **`access`**-Signale heißen. Da wir davon ausgehen, dass Sie schon mit einer Programmiersprache vertraut sind und das Konzept des Zeigers kennen, wollen wir hier nur ganz knapp, die verschiedenen, mit Zeiger zusammenhängenden Schreibweisen aus VHDL aufzählen, und an Beispielen erläutern.

Die formale Syntax der Definition eines **`access`**-Datentyps ist durch

---

`type` Bezeichner **`is access`** Datentyp_name;

---

gegeben. Ein **`access`**-Datentyp wird bei der Definition einem Datentyp zugeordnet, der angibt, auf welche Werte Zeiger von diesem **`access`**-Datentyp zeigen dürfen. Beispielsweise definieren die Typdefinitionen — das Beispiel haben wir zu großen Teilen aus [3] entnommen

```
9     type stimuli;
10    type stimuli_zeiger is access stimuli;
11    type stimuli is record
12      zeitpunkt: time;
13      wert: std_logic;
14      naechster_stimuli: stimuli_zeiger;
15    end record;
```

einen **access**-Datentyp `stimuli_zeiger`, der auf einen Verbund vom Datentyp `stimuli` zeigt. Da Werte des Datentyps `stimuli` jeweils Verbunde sind, die eine Komponente vom Datentyp `stimuli_zeiger` wieder enthalten, können mit Hilfe des Allokationsoperators **new** Listen von Werten vom Datentyp `stimuli` erzeugt werden. Nehmen wir an, dass `listenkopf` eine Variable vom Datentyp `stimuli_zeiger` und mit dem Wert **null** vorbelegt wäre — der Wert **null** bedeutet, dass der Zeiger keine Adresse enthält —. Mit der Anweisung

```
20      listenkopf:= new stimuli;
```

wird Speicherplatz für einen Verbund des Datentyps `stimuli` im Hauptspeicher unseres Rechners reserviert und die Adresse des so reservierten Speicherplatzes dem Zeiger `listenkopf` zugewiesen. Der Zeiger zeigt nun also auf einen (bisher nicht initialisierten) Verbund des Datentypen `stimuli`. Mit den Anweisungen

```
21      listenkopf.zeitpunkt := 25 ns;
22      listenkopf.wert := '1';
23      listenkopf.naechster_stimuli := null;
```

kann auf die Komponenten des Verbundes, auf den unser Zeiger `listenkopf` zeigt, zugegriffen und ihnen Werte zugewiesen werden. Diese Initialisierung hätten wir bereits bei der Allokation des Speicherplatzes machen können. Dies wäre über die Anweisung

```
24      listenkopf := new stimuli'(25 ns,'1',null);
```

erfolgt, die semantisch äquivalent zu den obigen vier Anweisungen ist.

Natürlich können jetzt noch weitere Verbundwerte an die Liste angefügt werden. Dies erfolgt entweder über eine weitere Variable `listenelement` des Datentyps `stimuli_zeiger` durch

```
25      listenelement := new stimuli'(50 ns,'0',null);
26      listenkopf.naechster_stimuli := listenelement;
```

oder über die Anweisung

```
27      listenkopf.naechster_stimuli :=
28        new stimuli'(50 ns,'0',null);
```

die man weiter ausbauen kann, sodass nicht nur ein Element an die Liste angehängt wird, sondern zwei (oder auch mehrere):

```
29      listenkopf.naechster_stimuli :=
30        new stimuli'(50 ns,'0',new stimuli'(75 ns,'1',null));
```

Die Freigabe von über den Operator **new** allokierten Speicherplatz erfolgt über den Deallokationsoperator **deallocate**. So gibt die Anweisung

```
31      deallocate(listenkopf);
```

den Speicherplatz frei, den der Verbund, auf den der Zeiger `listenkopf` zeigt, belegt. Will man die ganze Liste freigeben, so muss man Element für Element freigeben. Dies könnte zum Beispiel über den Programmcode

```
32      while listenkopf /= null loop
33        listenelement := listenkopf;
34        listenkopf := listenkopf.naechster_stimuli;
35        deallocate(listenelement);
36      end loop;
```

erfolgen, der die Liste vom Kopf ausgehend freigibt.

3.3.4 Das Datei-Konzept

Der Datentyp **file** spielt in VHDL, speziell im Rahmen von Testbenches, eine sehr wichtige Rolle. Dateien können zum Beispiel dazu dienen, im Rahmen der Validierung einer spezifizierten Schaltung, die einen Nurlesespeicher enthält, diesen zu belegen, oder, wie wir noch in den Kapiteln über Testbenches (siehe Teil IV) sehen werden, um die Validierung zu steuern. Zudem wird die Interaktion mit dem Benutzer, d.h. die Standard-Eingabe und Standard-Ausgabe, während eines Simulationslaufes über Dateizugriffe abgewickelt.

Wir wollen in diesem Abschnitt nur kurz darauf eingehen, wie der Dateityp **file** in VHDL deklariert wird. Wie schlussendlich das Öffnen einer Datei, d. h. die Zuordnung einer physikalischen Datei an einen Dateideskriptor, das Schließen einer Datei und der lesende und schreibende Zugriff auf Dateien genau erfolgt, wollen wir erst später detaillierter besprechen. Dies verbinden wir im Abschnitt 5.2 mit der Vorstellung der in VHDL vordefinierten Datentypen, die im Zusammenhang mit der Datei-Behandlung stehen. Im Abschnitt 5.4, in dem insbesondere die in dem Package `textio` vordefinierten Datentypen `line` und `text` sowie Prozeduren besprochen werden, lernen wir das elegante Arbeiten mit strukturierten Datei Ein-/Ausgaben kennen.

Der Datentyp **file** wird über das Schlüsselwort **file** deklariert. So hat eine Datei-Typdefinition die Form

---

type Bezeichner **is file of** Datentyp_name **;**

---

und eine Datei-Deklaration die Form[7]

[7] Neben der Möglichkeit, Dateien direkt bei der Deklaration zu öffnen, können diese auch über die implizit deklarierte Prozedur `file_open` geöffnet werden.

```
file Bezeichner : File_Datentyp_Name
    [[ open file_open_kind_Wert is "Datei_Name"]] ;
```

Über den anzugebenden Modus `file_open_kind` wird festgelegt, ob diese zum Lesen, zum Schreiben (löscht eine eventuell schon vorhandene Datei) oder zum Anhängen geöffnet wird. Der Modus wird als Aufzählungstyp im Package `standard` (siehe Abschnitt 5.2) definiert und besteht aus den Werten `read_mode`, `write_mode`und `append_mode`.Beispielsweise würde die Anweisung

```
22    type ethernet_pakete is file of ethernet_frame_format;
23    file ablage : ethernet_pakete open write_mode
24      is "puffer.dat";
```

einen Dateityp namens `ablage` deklarieren, an die physikalische Datei `"puffer.dat"` binden und zum Schreiben öffnen. In dieser Datei können Werte vom Datentyp `ethernet_frame_format` abgelegt werden. Dieser Datentyp ist ein in Abschnitt 3.3.2 definierter Verbund-Datentyp. Werte dieses Datentyps können jeweils ein Ethernet-Paket (bei einer Nachrichtenlänge von 368 Bit) aufnehmen.

Mit der Definition eines Datei-Datentyps werden implizit fünf Prozeduren zur Verfügung gestellt:

- Die Prozedur `file_open`, mit der Dateien in einem der oben angesprochenen Modi geöffnet werden können. Die Prozedur verfügt über einen Ausgabeparameter `status` vom Datentyp `file_open_status`, über den abgefragt werden kann, ob das Öffnen der Datei erfolgreich war, dem Parameter `f`, der den Dateideskriptor angibt, welcher geöffnet werden soll, einem `string`-Parameter `external_name`, über den der externe Name der physikalischen Datei angegeben wird, die an den Dateideskriptor `f` gebunden werden soll, und einem Parameter `open_kind`, der den Modus angibt, in dem die Datei geöffnet werden soll. Für weitere Informationen verweisen wir auf Abschnitt 5.2 auf Seite 138.

- Die Prozedur `read`, deren erster formaler Parameter `f` ein Dateideskriptor und deren zweiter formaler Parameter ein Ausgabeparameter namens `value` von dem, dem Datei-Datentyp zugrunde liegenden Datentyp ist. In unserem Beispiel wäre dies die Prozedur

```
25    procedure read (
26    file f: ethernet_pakete;
27    value: out ethernet_frame_format);
```

- Die Prozedur `write`, deren erster formaler Parameter `f` ein Dateideskriptor und deren zweiter formaler Parameter ein Eingabeparameter namens `value` von dem, dem File-Datentyp zugrunde liegenden Datentyp ist. Die Prozedur schreibt den in dem Parameter `value` abgespeicherten Wert in die über `f` spezifizierte Datei. In unserem Beispiel hätte die Prozedur das Aussehen

```
30    procedure write (
31    file f: ethernet_pakete;
32    value: in ethernet_frame_format);
```

- Die Funktion `endfile` vom Typ `boolean`, die als Parameter nur einen Dateideskriptor hat und im Rahmen des lesenden Zugriffs angewendet wird. Sie liefert bei Aufruf genau dann den Wert `true`, wenn das Ende der Datei erreicht ist.

```
35    function endfile (file f: ethernet_pakete) return boolean;
```

- Die Prozedur `file_close`, welche angewendet auf den übergebenen Dateideskriptor die zugehörige Datei schließt.

Will man die Dateien, in die man Werte speichert, manuell bearbeiten oder sich einfach nur anschauen, so ist es von Vorteil, über Zeichenketten definierte Dateien (d. h. **`file of`** `string`) zu benutzen. Wie in Abschnitt 5.4 noch dargestellt wird, sollte für diesen Zweck eine Prozedur `write` zur Verfügung gestellt werden, die — in unserem Beispiel — einen Verbund vom Typ `ethernet_frame_format` in einen String „konvertieren“ kann. Eine solche Prozedur könnte wie folgt aussehen:

```
27    procedure write (
28      L: inout line;
29      value: in ethernet_frame_format) is
30    begin
31      write(L,value.praeambel,right,64);
32      write(L,value.zieladresse,right,48);
33      write(L,value.quelladresse,right,48);
34      write(L,value.schluessel,right,16);
35      write(L,value.nachricht,right,368);
36      write(L,value.crc,right,32);
37    end;
```

Bei dieser Realisierung gehen wir davon aus, dass das Package `textio` (siehe Abschnitt 5.4) eingebunden ist und somit die überladene, auf `string`-Werten definierte Prozedur `write` zur Verfügung steht.

Diese schreibt eine Zeichenkette als „Zeile“ in eine Datei. Eine „Zeile“ wird über einen **`access`**-Datentyp für Zeichenketten repräsentiert. Die resultierende Ausgabe ist nun lesbar, davon abgesehen, dass die Komponenten eines Verbundes in unserem Beispiel nicht mit Zwischenräumen zwischen je zwei Komponenten in der Datei abgelegt wurden. Dem könnten wir aber einfach abhelfen.

Entsprechend könnte die `read`-Prozedur aussehen — auch hier gehen wir davon aus, dass uns die im Package `textio` auf Zeichenkettem definierte `read`-Prozedur zur Verfügung steht:

```
40    procedure read (
41      L: inout line;
42      value: out ethernet_frame_format) is
43    begin
44      read(L,value.praeambel);
45      read(L,value.zieladresse);
46      read(L,value.quelladresse);
47      read(L,value.schluessel);
48      read(L,value.nachricht);
49      read(L,value.crc);
50    end;
```

Wie wir noch im Abschnitt zum Package `textio` sehen werden, stellt uns diese Bibliothek auch noch die beiden Prozeduren

```
56    procedure writeline(file f : text; L : inout line);
```

und

```
21    procedure readline(file f: text; L: out line);
```

zur Verfügung. Mit Hilfe dieser beiden Prozeduren können Zeilen in eine Text-Datei (d. h. eine Datei über dem Datentyp `string`) geschrieben beziehungsweise aus dieser gelesen werden.

3.4 Sequentielle Anweisungen und Kontrollstrukturen

Wir konzentrieren uns in diesem Abschnitt auf die von VHDL zur Verfügung gestellten sequentiellen Anweisungen. Unter sequentiellen Anweisungen verstehen wir Anweisungen, die innerhalb von Funktionen und Prozessen sequentiell eine nach der anderen ausgeführt werden. „Sequentiell“ steht hier zur Abgrenzung zu „nebenläufig“.

Da die meisten der im Folgenden vorgestellten Anweisungsarten Ihnen aus anderen Programmiersprachen schon bekannt sein sollten, werden wir uns in diesem Abschnitt recht kurz fassen, bei verschiedenen Konstrukten geben wir im Wesentlichen nur die Syntax und ein Beispiel ohne große Kommentare an . Wir verweisen auf die englische Standardliteratur (zum Beispiel [3]) für weitere Erläuterungen.

Bei einigen der Anweisungen werden wir als Anwendungsbeispiel über einen (fiktiven) Baustein sprechen, der die Bus-Vergabe bei *Wide SCSI* (*Small Computer System Interface*) analysiert. Wir gehen im Folgenden davon aus, dass uns die Signale `data_bus` und `intern_grant_bus` sowie die Variable `the_winner_is` mit

```
8     subtype wide_scsi_id is integer range -1 to 15;
9     signal data_bus : bit_vector(15 downto 0);
10    signal intern_grant_bus : bit_vector(0 to 15);
    -- ...
17    variable the_winner_is : wide_scsi_id := -1;
```

zur Verfügung stehen. Das Signal `data_bus` modelliert den 16-Bit breiten Datenbus des Wide SCSI-Busses. Das Signal `intern_grant_bus` sei ein, ebenfalls 16-Bit breiter, interner Bus unseres Bausteins. In der Variable `the_winner_is` soll der „Gewinner" einer Bus-Zuteilung gespeichert werden. Der Default-Wert `-1` bedeutet, dass noch kein Gewinner bestimmt werden konnte.

Exkurs: Wide SCSI

Jedes SCSI-Gerät wird über den SCSI-Bus eindeutig über eine Nummer (ID) adressiert. Diese ID liegt bei (narrow) SCSI im Bereich von 0 bis 7, bei Wide SCSI im Bereich von 0 bis 15. Die Priorität, die ein SCSI-Gerät bei der Bus-Zuteilung hat, orientiert sich an seiner ID. Für die ersten acht SCSI-Geräte, d. h. die SCSI-Geräte mit den IDs 0 bis 7 gilt, dass bei einem Konflikt das Gerät mit der höheren ID den Bus zugeteilt bekommt. Bei Wide SCSI gilt für die acht zusätzlichen IDs das Gleiche, d. h. SCSI-Geräte mit höheren IDs gewinnen gegen SCSI-Geräte mit kleineren IDs. Jedoch haben die SCSI-Geräte mit den IDs 8 bis 15 kleinere Priorität als die mit den IDs 0 bis 7. Somit ergibt sich bei Wide-SCSI die Prioritäten-Reihenfolge 7, 6, 5, 4, 3, 2, 1 , 0, 15, 14, 13, 12, 11, 10, 9, 8.

Die Vergabe des SCSI-Busses — sofern er nicht belegt ist — erfolgt nun so, dass ein SCSI-Gerät (sagen wir mit ID p), das den Bus anfordert, „dem Busarbiter" seine Priorität mitteilt, indem es die p -te Datenleitung auf '1' zieht — bei Wide SCSI gibt es 16 Adressleitungen. Jedes SCSI-Gerät sieht also, welche Prioritäten seine Konkurrenten haben, und kann selbst feststellen, ob ihm der Bus gemäß oben angegebener Prioritäten-Reihenfolge zugeteilt wird oder nicht.

3.4.1 Bedingte Verzweigungen und Fallunterscheidungen

Wir beginnen mit den in VHDL zur Verfügung gestellten Möglichkeiten, bedingte Verzweigungen zu realisieren. An dieser Stelle möchten wir den Leser darauf aufmerksam machen, das zu den sequentiell bedingten Verzweigungen und Fallunterscheidungen jeweils auch ein nebenläufiges Konstrukt in VHDL bereitgestellt wird. Diese können nur außerhalb von Prozessen direkt im Rumpf von Architekturen verwendet werden. Solche Signalzuweisungen mit **`when...else`** - beziehungsweise **`with...select`** -Konstrukten werden im Abschnitt 3.5 besprochen.

if-Kontrukt

Die Syntax einer **if**-Abfrage ist durch

⟦ Marke **:** ⟧ **if** Bedingung **then**
  { sequentielle_Anweisung }
{ **elsif** Bedingung **then**
  { sequentielle_Anweisung } }
⟦ **else**
  { sequentielle_Anweisung } ⟧
end if;

---

gegeben. Sie erlaubt, dass sequentielle Anweisungen bedingt ausgeführt werden. So berechnet der Programmcode

```
32    if data_bus(7) = '1' then the_winner_is :=7;
33    elsif data_bus(6) = '1' then the_winner_is :=6;
34    elsif data_bus(5) = '1' then the_winner_is :=5;
      -- ...
39    elsif data_bus(0) = '1' then the_winner_is :=0;
40    elsif data_bus(15) = '1' then the_winner_is :=15;
41    elsif data_bus(14) = '1' then the_winner_is :=14;
      -- ...
47    elsif data_bus(8) = '1' then the_winner_is :=8;
48    end if;
```

welche der SCSI-Geräte eines Wide SCSI-Busses den Bus zugeordnet bekommt. Die **if**-Anweisung fragt die Datenleitungen gemäß der oben im Exkurs angegebenen Prioritäten-Reihenfolge ab.

Fallunterscheidungen

Die Syntax einer **case**-Anweisung ist durch

⟦ Marke **:** ⟧ **case** Ausdruck **is**
  when Auswahl **=>** { sequentielle_Anweisung }
  { **when** Auswahl **=>** { sequentielle_Anweisung } }
end case;

---

gegeben. Sie dient dazu, größere Fallunterscheidung übersichtlich gestalten zu können. Wir wollen dies an unserem Busarbiter wieder illustrieren. In dem Programmcode

```
50    case the_winner_is is
51      when 7  => intern_grant_bus <= X"0100";
52      when 6  => intern_grant_bus <= X"0200";
53      when 5  => intern_grant_bus <= X"0400";
      -- ...
58      when 0  => intern_grant_bus <= X"8000";
59      when 15 => intern_grant_bus <= X"0001";
60      when 14 => intern_grant_bus <= X"0002";
61      when 13 => intern_grant_bus <= X"0004";
      -- ...
66      when 8  => intern_grant_bus <= X"0080";
67      when others => intern_grant_bus <= X"0000"; -- -1
68    end case;
```

wird die p-te Komponente des Signals `intern_grant_bus` genau dann auf '1' gesetzt, wenn dem SCSI-Gerät mit ID p der Bus zugeteilt werden kann.

Über das Schlüsselwort **`when others`** spricht man alle nicht explizit aufgezählten Fälle an. Wird es verwendet, so muss es stets als letzter Eintrag angegeben werden. Fehlt das **`when others`**-Konstrukt in der **`case`**-Anweisung, so *müssen* alle Fälle aufgezählt sein. In unserem Beispiel ist der Fall `the_winner_is = -1` nicht explizit aufgezählt.

Fallstrick 9

Bei einer Fallunterscheidung über einem Signal oder einer Variable vom Datentyp `std_logic_vector` sind 9^n Fälle explizit anzugeben, falls das **`when others`**-Konstrukt nicht benutzt wird und n die Breite des Vektors ist. Ausschlaggebend hierfür ist die 9-wertige Logik, die dem Datentyp `std_logic` zu Grunde liegt.

Zur einfacheren Spezifikation mehrerer, gleich zu behandelnder Fälle stehen uns in VHDL die folgenden Möglichkeiten zur Verfügung

- die explizite Aufzählung, wie zum Beispiel
  `when` `7 | 5| 3,`
- die Bereichsangabe, wie zum Beispiel
  `when` `10` **`to`** `14,`
- Mischformen von beiden, wie zum Beispiel
  `when` `7| 5| 3| 10` **`to`** `14`
- und die **when others**-Anweisung.

3.4.2 Schleifen

In diesem Abschnitt wollen wir uns Schleifen und die mit Schleifen zusammenhängenden sequentiellen Anweisungen **`next`** und **`exit`** anschauen.

loop-Anweisung

Die einfachste Art einer Schleife hat das Aussehen

[Marke :] **loop**
  { sequentielle_Anweisung }
end loop [Marke] ;

Zum Beispiel berechnet die Schleife

```
72    j := 0;
73    elementare_schleife: loop
74      if data_bus(j) = '1' then
75         anzahl := anzahl + 1;
76      end if;
77      exit when j=15;
78      j:=j+1;
79    end loop elementare_schleife;
```

die Anzahl der SCSI-Geräte, welche den Bus gerade anfordern. Hierbei ist `anzahl` eine mit `0` vorbesetzte Variable vom Datentyp `integer` und `j` ebenfalls eine Integer-Variable, die wir als Laufvariable der Schleife eingesetzt haben. Die Schleife wird abgebrochen, wenn bei Ausführung der **`exit`**-Anweisung der Wert der Laufvariable `j` gleich 15 ist und somit alle Datenleitungen betrachtet wurden.

Eine solche elementare Schleife kann zu einer so genannten **`for`**-Schleife

---

[Marke :] **for** Bezeichner **in** diskreter_Bereich **loop**
  { sequentielle_Anweisung }
end loop [Marke] ;

---

oder zu einer **`while`**-Schleife

[Marke :] **while** Bedingung **loop**
  { sequentielle_Anweisung }
end loop [Marke] ;

---

erweitert werden. Die obige elementare **`loop`**-Schleife kann mit diesen Konstrukten äquivalent durch

```
84    for_schleife: for k in 0 to 15 loop
85      if data_bus(k) = '1' then
86        anzahl := anzahl + 1;
87      end if;
88    end loop for_schleife;
```

beziehungsweise

```
93    j := 0;
94    while_schleife: while j<=15 loop
95      if data_bus(j) = '1' then
96        anzahl := anzahl + 1;
97      end if;
98      j:=j+1;
99    end loop while_schleife;
```

beschrieben werden.

exit-Anweisung

⟦ Marke **:** ⟧ **exit** ⟦ Marke ⟧ ⟦ Bedingung ⟧ **;**

Der **exit**-Anweisung sind wir gerade bei der Vorstellung der elementaren **loop**-Schleife begegnet. Sie bewirkt, dass eine Schleife abgebrochen wird, falls die in der Anweisung angegebene Bedingung erfüllt ist. Die Marke hinter dem Schlüsselwort **exit** spezifiziert, welche Schleife abgebrochen wird — was nur Sinn bei verschachtelten Schleifen macht. Die Angabe dieser Marke ist optional. Wird sie nicht angegeben, so bezieht sich die **exit**-Anweisung auf die innerste Schleife.

Der Programmcode

```
105   wer_gewinnt_narrowSCSI: for i in 7 downto 0 loop
106     if data_bus(i) = '1' then
107       the_winner_is := i;
108     end if;
109     exit when the_winner_is > -1;
110   end loop wer_gewinnt_narrowSCSI;
111
112   wer_gewinnt_wideSCSI: for i in 15 downto 8 loop
113     exit when the_winner_is > -1;
114     if data_bus(i) = '1' then
115       the_winner_is := i;
116     end if;
117   end loop wer_gewinnt_wideSCSI;
```

berechnet wieder, welches der SCSI-Geräte den Bus zugeordnet bekommt. Hier beziehen sich die **exit**-Anweisungen jeweils auf die innerste Schleife, da die Schleifen nicht verschachtelt sind.

Um ein (sinnvolles) Beispiel angeben zu können, in dem sich die **exit**-Anweisung nicht auf die innere, sondern auf eine äußere Schleife bezieht, müssen wir etwas weiter ausholen. Wir nehmen an, unser SCSI-Analyse-Baustein zählt die Kollisionen zwischen je zwei SCSI-Geräten, d. h. wie oft zwei SCSI-Geräte gleichzeitig den Bus anfordern. Diese Information sei in der Variable `kollisionen`

```
21   type kollisionen_table is array (0 to 15, 0 to 15)
22     of integer;
23   variable kollisionen:
24     kollisionen_table :=(others=>(others=>0));
```

abgespeichert. Weiter sei `kollisionen_anzahl` eine mit 0 vorbesetzte `integer`-Variable. Beachten Sie die elegante Initialisierung des zweidimensionalen Feldes in Zeile (24). Dann zählt der Programmcode

```
126  AUSSEN: for i in 15 downto 0 loop
127    INNEN: for j in 0 to 15 loop
128      kollisionen_anzahl :=
129        kollisionen_anzahl + kollisionen(i,j);
130      exit AUSSEN
131        when kollisionen_anzahl > 10000;
132    end loop INNEN;
133  end loop AUSSEN;
134  assert kollisionen_anzahl <= 10000
135    report "Zuviele Kollisionen auf dem SCSI-Bus!"
136    severity warning;
```

die Gesamtanzahl der Kollisionen (mal 2). Ist die Anzahl der Kollisionen größer als 10 000, so wird die äußere Schleife verlassen und über die **assert**-Anweisung eine Warnung ausgegeben. **assert**-Anweisungen werden wir in Abschnitt 3.4.3 noch ausführlich besprechen.

next-Anweisung

Die **next**-Anweisung, deren Syntax durch

⟦ Marke **:** ⟧ **next** ⟦ Marke ⟧ ⟦ Bedingung ⟧ **;**

---

gegeben ist, ähnelt der **exit**-Anweisung. Ist die in der **next**-Anweisung angegebene Bedingung erfüllt, so wird der aktuelle Schleifendurchlauf an dieser Stelle unterbrochen und mit der nächsten Iteration begonnen. Die hinter dem Schlüsselwort **next** angegebene Marke spezifiziert die Schleife, auf die sich die **next**-Anweisung bezieht. Diese Angabe ist wieder optional. Wird die Marke nicht angegeben, so bezieht sich die **next**-Anweisung auf die innerste Schleife.

3.4.3 Weitere Anweisungen

null-Anweisung

```
null;
```

Die **null**-Anweisung tut genau das, was ihr Name auch sagt, nämlich nichts. Sie macht zum Beispiel innerhalb Fallunterscheidungen Sinn, wenn bei einigen Fällen nichts zu tun ist. Denken Sie daran, dass alle Fälle explizit aufgezählt werden müssen. Die **null**-Anweisung korrespondiert innerhalb von Prozessen zu dem Konstrukt **unaffected** in nebenläufigen Anweisungen (siehe Seite 47).

wait-Anweisung

Wait-Anweisungen begegnet man innerhalb von Prozessen. Wir haben die Rolle, die **wait**-Anweisungen während einer Simulation eines mit VHDL beschriebenen Bausteins spielen, ausführlich in Abschnitt 2.2.3 auf Seite 55 im Rahmen der Ausführung von Prozessen besprochen. Wir wollen an dieser Stelle auf diese Ausführungen verweisen und hier nur auf die reine Syntax und die Semantik der **wait**-Anweisungen eingehen.

Die Syntax einer **wait**-Anweisung ist durch

```
[[ Marke : ]] wait
   [[ on Sinal_Name { , Signal_Name } ]]
   [[ until Bedingung ]]
   [[ for Zeit_Angabe ]] ;
```

definiert. Sei s ein Signal eines beliebigen Datentyps. Zum Beispiel ist die Anweisung

```
14    wait on s;
```

äquivalent zu

```
15    wait until s'event;
```

(siehe auch Abschnitt 3.2.3 zu signalbezogenen Attributen). Der zugehörige Prozess verbleibt so lange im Ruhezustand bis ein Ereignis auf dem Signal s eintritt. Prozesse können auch für eine definierte Zeitdauer „schlafen gelegt“ werden. Die Anweisung

```
18    wait for 33333 ns;
```

bewirkt ein Aussetzen des umgebenden Prozesses für 33.333 ns.

Fallstrick 10

Bei der Spezifikation eines Bausteins sollten Sie sich bewusst sein, dass durch die Benutzung von **`wait for`**-Anweisungen die Spezifikation in der Regel nicht synthetisierbar ist.

Insofern sollte von der Verwendung dieses Konstrukts Abstand genommen werden, es sei denn, es ist nur zu Simulationszwecken, wie etwa das Anlegen von Stimuli durch eine Testbench (siehe Teil IV), vorgesehen. Anwendung findet diese Art von `wait`-Konstrukten ebenso bei der Modellierung von „Black Boxes“ zu Simulationszwecken, was wir ja bereits zur Taktgenerierung beim Morse-Code-Detektor genutzt haben. Weitere typische Anwendungsgebiete sind generierte Module (z. B. RAM-Blöcke) und die vielfältigen so genannten *IP-Cores* (*engl: intelligent property*).

Soll im Gegensatz zu den ersten beiden **`wait`**-Anweisungen der Prozess nur so lange angehalten werden bis eine Zuweisung — eine Zuweisung hat nicht immer eine Änderung zur Folge — an das Signal `s` erfolgt, so muss abgefragt werden, ob eine Transaktion am Signal `s` vorliegt. Dies erfolgt über das Attribut `s'transaction`, das ein Signal vom Datentyp `bit` ist und immer dann seinen Wert ändert, wenn eine Zuweisung an das Signal `s` erfolgt (siehe Abschnitt 3.2.3). Die entsprechende **`wait`**-Anweisung lautet

```
19    wait on s'transaction;
```

return-Anweisung

Die **`return`**-Anweisung, deren Syntax durch

⟦ Marke **:** ⟧ **`return`** ⟦ Wert ⟧ **;**

---

gegeben ist, wird in Funktionen und Prozeduren eingesetzt, um das Unterprogramm (vorzeitig) zu beenden. Innerhalb von Funktionen wird über die **`return`**-Anweisung der Funktionswert zurückgegeben.

assert- und report-Anweisung

Die **`assert`**-Anweisung, deren Syntax durch

⟦ Marke **:** ⟧ **`assert`** Bedingung
⟦ **`report`** String_Ausdruck ⟧
⟦ **`severity`** Ausdruck_vom_Typ_severity_level ⟧ **;**

---

gegeben ist, erlaubt die Überprüfung von Bedingungen während einer Simulation. Ist die Bedingung verletzt, so wird der hinter dem Schlüsselwort **report** angegebene Text auf der Standard-Ausgabe ausgegeben. Die Schwere des „Vergehens“ kann über den **severity_level**-Wert spezifiziert werden. Dieser wird ebenfalls „mitausgegeben“ und beendet gegebenenfalls den Simulationslauf.

Der Datentyp `severity_level` ist ein im Package `standard` (siehe Abschnitt 5.2) vordefinierter Aufzählungstyp, der die Werte `note`, `warning`, `error` und `failure` enthält.

Die **assert**-Anweisung findet insbesondere in Funktionen und Prozeduren Anwendung, welche als formale Parameter unbeschränkte Felder (siehe Abschnitt 3.3.2) haben. Abbildung 3.9 zeigt ein solches Beispiel. Die Funktion als solche werden wir in Kapitel 4 noch genauer erläutern.

```
40  function integer2bitvector (a: integer; laenge: natural)
41    return bit_vector is
42    variable tmp: bit_vector (laenge-1 downto 0);
43  begin
44    assert (a >= -2**(laenge-1)) and (a < 2**(laenge-1)-1)
45      report "Die Zahl ist mit dieser Längenbeschränkung
46              nicht darstellbar!"
47      severity failure;
48
49    -- Berechnung der Zweierkomplement-Darstellung von a,
50    -- abgespeichert in der Variable tmp
51    -- ...
52
53    return tmp;
54  end;
```

Abbildung 3.9: Plausibilitätskontrolle bei einer Funktion mit unbeschränkten Feldern als formale Parameter

Neben der **assert**-Anweisung bietet VHDL auch noch die Möglichkeit, ohne Überprüfung einer Bedingung Meldungen auf der Standard-Ausgabe auszugeben und diese mit einem `severity_level`-Wert zu verknüpfen. Eine solche **report**-Anweisung hat das Aussehen

---

⟦ Marke **:** ⟧ **report** Ausdruck
⟦ **severity** Ausdruck_vom_Typ_severity_level ⟧ **;**

In Kombination mit dem Attribut `image` können elegant Debug-Ausgaben folgender Form

```
117   for i in s'range loop
118     report "akt. Wert von s(" &
119            integer'image(i) & "):" &
120            std_logic'image(s(i));
121   end loop;
```

erzeugt werden.

3.4.4 Funktionen und Prozeduren

Wir verweisen auf Kapitel 4, das sich ausschließlich mit Funktionen und Prozeduren beschäftigt.

3.5 Nebenläufige Anweisungen und Konstrukte

Anweisungen (beziehungsweise Prozesse) heißen nebenläufig (engl. *concurrent*), wenn sie unter Beachtung vorhandener Wechselwirkungen unabhängig bearbeitet werden können.

VHDL kennt neben (expliziten und impliziten) Prozessen, welche jeweils aus einer Folge von sequentiellen Anweisungen bestehen, weitere nebenläufige Anweisungen, nämlich

- die nebenläufige **assert**-Anweisung,
- die **generate**-Anweisung und
- die schon in Abschnitt 3.4.1 angesprochenen nebenläufigen Signalzuweisungen.

Prozesse haben wir in diesem Buch schon in vielen Beispielen kennen gelernt. Wir sind zudem schon detailliert auf Prozesse in Kapitel 2 und in Abschnitt 3.1.2 eingegangen.

Die **generate**-Anweisung illustrieren wir noch ausführlich in Kapitel 6, in dem wir verschiedene Familien von Addierern mit VHDL beschreiben werden.

Aus diesem Grunde wollen wir an dieser Stelle nur auf die nebenläufige **assert**-Anweisung und auf nebenläufige Signalzuweisungen zu sprechen kommen und verweisen für die anderen nebenläufigen Anweisungen auf die gerade erwähnten Abschnitte.

3.5.1 Die nebenläufige assert-Anweisung

Die nebenläufige **assert**-Anweisung entspricht in ihrer Syntax der in Abschnitt 3.4.3 vorgestellten sequentiellen **assert**-Anweisung. Auch von der Semantik her verhalten sich beide Varianten gleich. Man unterscheidet zwischen den Varianten nur aus dem Grund, weil man die **assert**-Anweisung sowohl in funktionalen Beschreibungen innerhalb von Prozessen als auch in strukturellen Beschreibungen und innerhalb von **entity**-Definitionen einsetzen kann.

Abbildung 3.10 zeigt ein Anwendungsbeispiel der nebenläufigen **assert**-Anweisung. Es werden in der Schnittstellenbeschreibung des Bausteins `single2double`, der unbeschränkte Felder als Ein-/Ausgabeports hat, **assert**-Anweisungen spezifiziert. Diese stellen Bedingungen an die Signale, die mit diesen Ports verknüpft werden. So wird zum Beispiel verlangt, dass das Ausgabesignal breiter als das Eingabesignal sein muss. Wird nun eine Komponente vom Baustein `single2double` in einer strukturellen Beschreibung eines Bausteins instanziiert, so überprüft der Simulator, ob die in der Entity-Beschreibung des Bausteins enthaltenen **assert**-Anweisungen verletzt sind. Ist dies der Fall, erfolgt die entsprechende Fehlermeldung auf der Konsole.

```
4   entity single2double is
5     -- Baustein zur Konvertierung
6     -- einer n-bit Zweierkomplement-Darstellung
7     -- in eine m-bit Zweierkomplement-Darstellung mit n<=m
8     port (
9       inp: in std_logic_vector;
10      outp: out std_logic_vector);
11  begin
12    ueberpruefe_laenge:
13      assert inp'length <= outp'length
14      report "Die Wortbreite des Ausgangs ist zu klein!";
15
16    ueberpruefe_rechter_index:
17      assert inp'right = 0 and outp'right = 0
18      report "Rechte Komponente muss Index 0 haben!";
19
20  end entity single2double;
21  architecture verhalten of single2double is
22  begin
23    process(inp)
24    begin
25      for i in inp'length to outp'length-1 loop
26        outp(i) <= inp(inp'length-1);
27      end loop;
28      outp(inp'length-1 downto 0) <= inp;
29    end process;
30  end verhalten;
```

Abbildung 3.10: Illustration nebenläufiger **assert**-Anweisungen

3.5.2 Die nebenläufigen Signalzuweisungen

Bei den nebenläufigen Signalzuweisungen handelt es sich eigentlich um implizite Prozesse. Sie erlauben eine effizientere Schreibweise als dies mit (expliziten) Prozessen in der Regel möglich ist. In Abschnitt 2.2.1 haben wir auf Seite 46 die Syntax von Signalzuweisungen, wie sie in expliziten Prozessen verwendet werden dürfen, angegeben. Für nebenläufige Prozesse erlaubt VHDL erweiterte, ausdrucksstärkere Anweisungen, mit denen zum Beispiel bedingte Zuweisungen realisiert werden können. Hierzu stellt VHDL das **when...else**- und das **with...select**-Konstrukt zur Verfügung.

Wir wollen wiederum auf die Angabe der exakten Syntax verzichten und beide Konstrukte nur anhand von jeweils einem Beispiel illustrieren.

Die Architektur des in Kapitel 6, in Bild 6.1 verwendeten Bausteins `set_b_entry`, der den `b`-Eingang des Addierer/Subtrahierer-Bausteins in Abhängigkeit des Steuersignals `op` — das Steuersignal `op` gibt an, ob der Baustein `add_sub` eine Addition oder eine Subtraktion ausführen soll — setzt, braucht nicht über einen expliziten Prozess, wie wir dies in Kapitel 6 machen, definiert werden, sondern kann auch durch die nebenläufige Signalzuweisung

```
22  architecture structure1 of set_b_entry is
23  begin
24    b_out <= b_in when op='0' else not b_in;
25  end structure1;
```

beziehungsweise

```
26  architecture structure2 of set_b_entry is
27  begin
28    with op select b_out <= b_in when '0',
29                            not b_in when '1',
30                            unaffected when others;
31  end structure2;
```

spezifiziert werden.

3.6 Attribute

Attribute geben Eigenschaften von in VHDL verfügbaren oder benutzten Elementen an. In der Regel sind dies Eigenschaften von Datentypen oder Zugriffe auf Werte aus dem betreffenden Wertebereich. Auf ein Attribut kann über den Namen des betreffenden Datentyps, des betreffenden Signals oder der betreffenden Variablen gefolgt von einem Apostroph und der Bezeichnung des Attributs zugegriffen werden.

Neben vordefinierten Attributen kann der Designer auch eigene Attribute definieren.

3.6.1 Selbstdefinierte Attribute

VHDL erlaubt, dass der Programmierer auf fast allen in VHDL zur Verfügung stehenden Elementen Attribute definieren kann. Eine solche Definition ist aus zwei Teilen aufgebaut.

- Als Erstes muss der Datentyp des Attributs definiert werden. Dies erfolgt über eine Anweisung der Form

---

attribute Bezeichner **:** Datentyp_Name **;**

So würde die Anweisung

```
11    attribute bus_request: boolean;
```

ein Attribut mit dem Namen `bus_request` deklarieren und definieren, dass dieses Attribut Werte vom Datentyp `boolean` liefert. Durch diese Anweisung wird noch nicht festgelegt, welchem Element von VHDL `bus_request` als Attribut dienen soll.

- Die Zuordnung eines Attributs zu einem VHDL-Element erfolgt über die Attribut-Spezifikation. Um ein Beispiel einer solchen Attribut-Spezifikation geben zu können, lassen Sie uns noch mal auf unseren SCSI-Analyse-Baustein von vorhin (siehe Absatz 3.4.1 auf Seite 103) zurückkommen. Wir binden das deklarierte Attribut `bus_request` an die Variable `intern_grant_bus`, um darüber zu erfahren, ob ein SCSI-Gerät den Bus angefordert hat. Die folgenden beiden Codezeilen

```
12    attribute bus_request of intern_grant_bus:
13      signal is (intern_grant_bus/=X"0000");
```

deklarieren und definieren das über dem Datentyp `boolean` definierte Attribut `bus_request`. Es trägt genau dann den Wert `false`, wenn der Bus `intern_grant_bus` mit dem Null-Vektor belegt ist.

Die allgemeine Syntax einer solchen Attribut-Spezifikation ist durch

---

attribute Bezeichner **of** Element_Name:
Element_Klasse **is** Ausdruck_vom_Datentyp_des_Attributs **;**

---

gegeben. Die Elemente, denen Attribute zugeordnet werden können, müssen aus einer der Klassen **entity**, **architecture**, **configuration**, **package**, **procedure**, **function**, **type**, **subtype**, **constant**, **signal**, **variable**, **file**, **component**, **label**, **literal**, **units**, **group** sein. Für weitergehende Informationen zum Thema Attribut-Spezifikation verweisen wir auf [3].

Um ein selbstdefiniertes Attribut nun auch benutzen zu können, greift man, wie auch bei vordefinierten Attributen, über den Namen des VHDL-Elements gefolgt von einem Apostroph und dem Namen des Attributs zu. So überprüft die Anweisung

```
138   assert (intern_grant_bus'bus_request)
139     report "Der SCSI-Bus wurde von einem Gerät angefragt!"
140   severity note;
```

ob wenigstens ein SCSI-Gerät den SCSI-Bus angefordert hat.

3.6.2 Vordefinierte Attribute

Die vordefinierten Attribute kann man in die folgenden fünf Klassen einteilen, von denen wir alle, mit Ausnahme der so genannten allgemeinen Attribute, schon kennen gelernt haben — wir verweisen auf die entsprechenden Kapitel des Buches. Die fünf Klassen sind:

- signalbezogene Attribute (siehe Abschnitt 3.2.3),
- das auf jedem Datentyp `T` definierte Attribut `T'base` (siehe Abschnitt 3.3)
- auf skalaren Datentypen definierte Attribute (siehe in Abschnitt 3.3.1),
- feldbezogene Attribute (siehe Abschnitt 3.3.2)
- allgemeine, in der Regel im Rahmen der Diagnose angewendete, Attribute.

Allgemeine Attribute zu Diagnosezwecken

VHDL stellt drei weitere Attribute zur Verfügung, die im Wesentlichen dazu verwendet werden, um im Rahmen von Testbenches dem Entwerfer detaillierte Informationen während den Simulationsläufen zur Verfügung zu stellen.

- Das Attribut `E'simple_name`, welches den Namen des Elementes `E` als Zeichenkette liefert. Beispielsweise gilt
  `intern_grant_bus'simple_name = "intern_grant_bus"`.
- Das Attribut `M'path_name`, welches angewendet auf eine Marke `M` die Folge der Marken als Zeichenkette zurückgibt, über die man in dem, in der Regel hierarchischen, Entwurf zur Marke `M` gelangt.
- Das Attribut `M'instance_name`, welches eine ähnliche Funktion wie das Attribut `M'path_name` hat. Es stellt eine Erweiterung von `M'path_name` in dem Sinne dar, dass neben dem Pfad zur Marke `M` weitere Informationen in dem Ausgabestring enthalten sind.

Sehr hilfreich sind diese Attribute zur Lokalisierung von Fehler- und Hilfsausgaben während der Simulation. Diese sind nicht unmittelbar einer Komponente zugeordnet, sondern meist nur einer bestimmten Zeile einer Datei mit VHDL-Quellcode. Durch die Verwendung obiger Attribute kann diese Lokalisierung verbessert werden. Zum Beispiel liefert das Codefragment

```
142   l1: assert (intern_grant_bus'bus_request)
143     report l1'path_name & ": " &
144            "Der SCSI-Bus wurde von einem Gerät angefragt!"
145   severity note;
146   l2: assert (intern_grant_bus'bus_request)
147     report l2'instance_name & ": " &
148            "Der SCSI-Bus wurde von einem Gerät angefragt!"
149   severity note;
```

die folgende Konsolenausgabe:

```
ASSERT:
  NOTE at 90 ns+0: :fragments:check:l1:
    Der SCSI-Bus wurde von einem Gerät angefragt!

ASSERT:
  NOTE at 90 ns+0: :fragments(behavior):check:l2:
    Der SCSI-Bus wurde von einem Gerät angefragt!
```

Man erkennt, dass die beiden `assert`-Anweisungen innerhalb eines Bausteins mit Namen `fragments` und dort innerhalb eines Prozesses `check` aufgerufen wurden. Die Verwendung des Attributes `instance_name` erweitert diese Angaben noch um den Namen der Architektur. In unserem Beispiel trägt die Architektur den Namen `behavior`. Der angebene Zeitpunkt von 90 ns ergab sich aus der Simulation des Codefragments und ist in diesem Zusammenhang nicht weiter von Bedeutung.