5. VHDL – Datentyp `stdlogic`

Während sich das Verhalten der Standardausgange mit dem Datentyp `bit` in VHDL leicht modellieren lasst, erfordert die Modellierung der Open-Drain- und Three-State-Funktionalitäten einen Datentyp, der mehr als die bisherigen Zustände 0 und 1 annehmen kann. Es werden daher die mehrwertigen Datentypen `std_ulogic` und `std_logic` eingeführt, die hier nun weiter erläutert werden sollen.

5.1. Mehrwertige Datentypen

In der IEEE-Bibliothek ist der 9-wertige Datentyp `std_ulogic` bzw. `std_ulogic_vector` definiert, dessen Signalwertbedeutung in Tabelle 5-1erläutert wird.

Tabelle 5-1: Signalwerte des 9-wertigen Datentyps

Wert	Bedeutung	Verwendung
'U'	Nicht initialisiert	Das Signal ist im Simulator (noch) nicht initialisiert.
'X'	Undefinierter Pegel	Simulator erkennt mehr als einen aktiven Signaltreiber (Buskonflikt)
'0'	Starke logische 0	L-Pegel eines Standardausgangs
'1'	Starke logische 1	H-Pegel eines Standardausgangs
'Z'	Hochohmig bzw. floatend	Three-State-Ausgang
'W'	Schwach unbekannt	Simulator erkennt Buskonflikt zwischen schwachen L- und H-Pegeln
'L'	Schwacher L-Pegel	Open-Source Ausgang mit Pull-Down- Widerstand
'H'	Schwacher H-Pegel	Open-Drain-Ausgang mit Pull-Up- Widerstand
'-'	Don't-Care	Logikzustand des Ausgangssignals bedeutungslos, kann für Minimierung verwendet werden

5.2. Physikalische Implementierung und Beschaltung von Logikgattern

Die Verwendung dieses Datentyps erfordert die Einbindung der IEEE-Bibliothek durch die folgenden VHDL- Codezeilen vor jeder `entity`, die diesen Datentyp referenziert:

```
library ieee;
use ieee.std_logic_1164.all;
```

Ähnlich wie beim Datentyp `bit` darf für jedes Signal beim Datentyp `std_ulogic` nur ein Treiber existieren. Das bedeutet, dass jedes Signal seine Signalwertzuweisungen nur aus *einem* Prozess bzw. aus *einer* nebenläufigen Anweisung erhalten darf. Da bei der

Verwendung von Gattern mit Open-Drain- oder Three-State- Ausgang sich der Signalwert aber erst nach Auswertung mehrerer Gatterausgange einstellt, ist der Datentyp `std_ulogic` zur Modellierung von Schaltungen mit diesem Ausgangstyp nicht geeignet.

5.3. Datentypen mit Auflösungsfunktion

Für die bei Open-Drain- und Three-State- Ausgängen vorhandene Situation, dass mehrere Ausgänge, also mehrere VHDL-Prozesse bzw. nebenläufige Anweisungen den tatsachlichen Signalwert definieren, wird in dem ebenfalls 9-wertigen Datentyp `std_logic` bzw. `std_logic_vector` eine Auflösungsfunktion in Form einer Matrix definiert. Der sich tatsachlich einstellende Signalwert für den Fall, dass zwei verschiedene Treiber A und B mit unterschiedlichen Signalwerten existieren, ergibt sich aus der Tabelle 5-2. Einige Details dieser Matrix sollen hier erläutert werden:

- Wenn ein Treiber ein undefiniertes Signal aufweist, so ist auch der Ausgang undefiniert 'U' (Zeile 1 bzw. Spalte 1 der Auflösungsmatrix).
- Wenn ein Treiber eine '0' und der andere Treiber eine '1' besitzt, so macht die Auflösungsmatrix dies durch einen Signalkonflikt 'X' deutlich (z.B. Schnittpunkt aus Spalte 3 und Zeile 4).
- Wenn einer der beiden Treiber bereits einen Signalkonflikt anzeigt, so zeigt die Auflösungsmatrix ebenfalls einen Signalkonflikt an (Zeile 2 bzw. Spalte 2). Ein einmal existierender Signalkonflikt wird also durch die gesamte Schaltung weitergereicht.

Tabelle 5-2: Auflösungsfunktion `resolved` für den 9-wertigen Datentyp `std_logic`

Signalwert von Treiber B \ Signalwert von Treiber A	U	X	0	1	Z	W	L	H	-
U	U	U	U	U	U	U	U	U	U
X	U	X	X	X	X	X	X	X	X
0	U	X	0	X	0	0	0	0	X
1	U	X	X	1	1	1	1	1	X
Z	U	X	0	1	Z	W	L	H	X
W	U	X	0	1	W	W	W	W	X
L	U	X	0	1	L	W	L	W	X
H	U	X	0	1	H	W	W	H	X
-	U	X	X	X	X	X	X	X	X

Eine Open-Drain-Ausgangsschaltung lässt sich mit diesem Datentyp dadurch modellieren, dass entweder eine '0' oder aber der Wert 'H' zugewiesen wird. Der Tabelle X ist zu

entnehmen, dass beim Zusammenschalten zweier Open-Drain-Ausgänge der Wert '0' dominant ist (Schnittpunkt aus Spalte 8 und Zeile 3). Ein entsprechendes VHDL-Modell wird im Beispiel X vorgestellt.

Ebenso lassen sich Ausgänge verbinden, wenn ein Signaltreiber hochohmig wird. Wenn nun der zweite Treiber einen der Werte '0', '1', 'L' oder 'H' annimmt, so dominiert dieser Treiber mit seinem Signalwert (Spalte 5 der Auflösungsmatrix). Dieser Fall wird in Beispiel X detailliert erläutert.

Angesichts der vielen Möglichkeiten, die der Datentyp `std_logic` bietet, stellt sich die Frage, ob es nicht sinnvoll sein konnte, die VHDL-Modellierung ausschließlich mit diesem Datentyp vorzunehmen. Dabei sind jedoch die folgenden Nachteile zu berücksichtigen:

- Der Simulationsaufwand ist höher, als der bei unaufgelösten Signalen, wie z.B. beim Datentyp `bit` oder `std_ulogic`, da für jede Zuweisung auf ein `std_logic`-Signal zusätzlich die Auflösungsfunktion ausgewertet werden muss.
- Mit der Erfahrung aus prozeduralen Programmiersprachen sind weniger erfahrene Entwickler leicht dazu geneigt, unbedachterweise ein und demselben Signal an mehreren Stellen der Architektur, also in verschiedenen Prozessen bzw. nebenläufigen Anweisungen, einen Wert zuzuweisen. Aus Hardwaresicht ist dies jedoch gleichwertig mit dem Kurzschließen von Standardausgängen, sodass dieser Fall von den Implementierungswerkzeugen verboten wird. Bei einer VHDL-Simulation fallen derartige Fehler mit den unaufgelösten Datentypen `bit` und `std_ulogic` durch Meldungen des Simulationscompilers schnell auf. Bei Verwendung des Datentyps `std_logic` werden diese Fehler hingegen erst deutlich später im Entwurfsablauf, nämlich während der Synthese festgestellt. Dann sind häufig zeitraubende, konzeptionelle Änderungen des VHDL-Modells erforderlich.

Weniger erfahrene VHDL-Entwickler sollten den Datentyp `std_logic` nur an den Stellen verwenden, wo er wirklich erforderlich ist:

5.4. VHDL-Modellierungsbeispiele

Die VHDL-Syntax fordert auf der rechten und linken Seite einer Signalzuweisung neben der gleichen Busbreite auch einen gleichen Datentyp. Dies erfordert, dass an den Schnittstellen zwischen `bit` und `std_logic` bzw. `std_ulogic` die in Tabelle 5-3 angegebenen Konversionsfunktionen verwendet werden müssen, die in der Bibliothek `ieee.std_logic.1164` definiert sind.

Tabelle 5-3: Konversionsfunktionenzwischen den Datentypen `bit` und `std_logic` bzw. `std_ulogic`. Man beachte, dass die Gross-Kleinschreibung in den Funktionsnamen der ersten Spalte beliebig ist

Konversionsfunktion	Argumenttyp	Ergebnistype
`To_bit()`	- `std_ulogic` - `std_logic`	- `bit`
`To_StdUlogic()`	- `bit`	- `std_ulogic` oder `std_logik`
`To_bitvector()`	- `std_ulogic_vector` - `std_logic_vector`	- `bit_vector`
`To_StdULogicVector()`	- `bit_vector` - `std_logic_vector`	- `std_ulogic_vector`
`ToStdLogicVector()`	- `bit_vector` - `std_ulogic_vector`	- `std_logic__vector`

In der Tabelle 5-3 fällt auf, dass das Ergebnis der `To_StdULogic()` - Funktion entweder vom Typ `std_ulogic` oder vom Typ `std_logic` sein kann. Der VHDL-Compiler erlaubt hier beide Datentypen da der Datentyp `std_logic` in der Bibliothek `ieee.std_logic_1164` nicht als eigenständiger Datentyp sondern als Untermenge (subtype) des Datentyps `std_ulogic` definiert ist, für den die in Tabelle 5-3 definierte Auflösungsfunktion `resolved` definiert ist:

```
SUBTYPE std_logic IS RESOLVED std_ulogic;
```

Beispiel 5-1: Zusammenschaltung von Open-Drain-Ausgängen und Wired-AND- Verknüpfung

Listing xx zeigt die Modellierung von Open-Drain-Ausgängen und deren Wired-AND-Verknüpfung. Die Eingangssignale `IN1` und `IN2` sind vom Datentyp `bit`, sie müssen daher in den Prozessen `P1` und `P2` durch die Konversionsfunktion `To_stdlogic()` in den Datentyp `std_logic` konvertiert werden. Außerdem wird in diesen Prozessen ein eventuell vorhandener Signalwert 1 in den Signalwert H umgesetzt. Die Wired-AND-Verknüpfung geschieht dadurch, dass beide Prozesse das gleiche Ausgangssignal `OD_OUT` beschreiben. Der tatsächliche Wert dieses Signals ergibt sich also durch die Auflösungsfunktion des Datentyps `std_logic`.

Listing 5-1: Modellierung von OPEN-DRAIN Ausgängen und WIRED-OR Verknüpfungen durch zwei Prozesse, die das gleiche Signal beschreiben.

```
library ieee;
use ieee.std_logic_1164.all;

entity OPEN_DRAIN is
```

```
     port (
          IN1, IN2: in bit;
          OD_OUT: out std_logic
     );
end OPEN_DRAIN;

architecture TEST of OPEN_DRAIN is

function To_stdlogic ( b : bit) return std_logic is
begin
     case b is
          when '0' => return ('0');
          when '1' => return ('1');
          when others => return '0';
     end case
end;

begin
P1:  process( IN1 )
     begin
          OD_OUT <= To_stdlogic(IN1);
          if IN1 = '1' then OD_OUT <= 'H';
          end if;
     end process P1;

P2:  process( IN2 )
     begin
          OD_OUT <= To_stdlogic(IN2);
          if IN2 = '1' then OD_OUT <= 'H';
          end if;
     end process P2;

end TEST;
```

5.5. Einfache aufgelöste Signale (Basic Resolved Signals)

Betrachten wir ein digitales System dessen Ausgangssignal durch zwei Treiberausgänge getrieben wird, so können wir das Ergebnis, basierend auf der analogen Schaltungstheorie, relativ leicht bestimmen. Das Signal wird einen Zwischenzustand annehmen, welcher von der elektrischen Charakteristik der beteiligten Treiber abhängig ist. Dieser Zwischenzustand kann (muss aber nicht) einen gültigen logischen Zustand besitzen. Man verbindet in der Regel nur Ausgänge, wenn höchstens einer der Ausgänge zu einem bestimmten Zeitpunkt aktiv ist. In der restlichen Zeit befinden sich die Ausgänge in einen hochohmigen Zustand. In diesem Fall sollte der resultierende Zustand des Ausgangssignals dem Zustand des Treibers entsprechen. Darüber hinaus wird in der Form von 'Pull-up's' festgelegt, welchen Zustand das Ausgangssignals annimmt, wenn alle Treiber inaktiv sind.

Während dieser Ansatz bei einigen einfachen Modellen zufriedenstellend ist, gibt es andere Fälle, wo eine weitere Auflösung erforderlich ist. Einer der Gründe für die Simulation eines Modells ist, Fehler, z.B. mehrere gleichzeitig aktive verbundene Ausgänge, zu erkennen. In diesem Fall müssten wir um solche Fehler zu erkennen den einfachen Ansatz verlassen. Ein weiteres Problem entsteht, wenn wir die Modellierung auf einer höheren Abstraktionsebene durchführen oder komplexe Signaltypen verwenden. Wir müssen bestimmen, welchen Signalzustand der Ausgang annehmen soll, wenn man mehrere Treiberausgänge miteinander verbunden sind. Das Vorgehen in VHDL ist eine sehr allgemein. Die Sprache erfordert, dass der Designer die Werte spezifiziert welche beim Zusammenschalten mehrerer Treiberausgänge austreten können.

Dies geschieht durch aufgelöste Signale, welche eine Erweiterung der Basissignale sind. Ein aufgelöstes Signal enthält in seiner Definition eine Funktion, die Auflösung-Funktion, welche verwendet wird, um den resultierenden Signalzustand aus den Zuständen aller seiner Quellensignale zu berechnen.

Das nachfolgende Beispiel gibt einen Einblick wie eine Auflösungsfunktion arbeitet. Es modelliert unterschiedliche TRI-STATE Signalzustände als eine einfache Erweiterung des vordefinierten Signaltyps `bit`:

```
type tri_state_logic is ('0', '1', 'Z');
```

Der zusätzliche Zustand 'Z' wird vom Signal verwendet um anzugeben, dass es sich im hochohmigen Zustand befindet. Als nächstes ist eine Funktion zu schreiben, welche eine Liste von Signalen dieses Typs akzeptiert und den daraus resultierenden Rückgabezustand dem angeschlossenen Signal zuweist. In diesem Beispiel gehen wir davon aus, dass nur ein Treiber zu einem Zeitpunkt aktiv ist ('0' oder '1'). Die Schwierigkeit beim Schreiben der Funktion ist, dass wir die Funktion nicht auf eine feste Anzahl von Treibersignalen beschränken können. Man kann dies vermeiden, indem man

ein Array von `tri_state_logic` Werten als Übergabe- Parameter für die Auflösungsfunktion verwendet. Dieses Array wird durch die folgende Typdeklaration definiert:

```
type tri_state_logic_array is
     array (integer range <>) of tri_state_logic;
```

Die Funktionsdeklaration ist:

```
function resolve_tri_state_logic ( values : in tri_state_logic_array )
     return tri_state_logic is
     variable result : tri_state_logic := 'Z';
begin
     for index in values'range loop
          if values(index) /= 'Z' then
               result := values(index);
          end if;
     end loop;
     return result;
end function resolve_tri_state_logic;
```

Der letzte Schritt um ein aufgelöst Signal zu erzeugen ist, das Signal wie folgt zu deklarieren:

```
signal s1 : resolve_tri_state_logic tri_state_logic;
```

Diese Signaldeklaration ist fast identisch zu einer normalen Signaldeklaration. Die Deklaration wird mit dem Zusatz des Funktionsnamens der Auflösungsfunktion vor dem Signal ergänzt. Das Signal `s1` ist vom Typ `tri_state_logic`, die Angabe des Funktionsnamens weist darauf hin, dass das Signal ein aufgelöstes Signal ist. Für die Auflösung wird die angegebene Auflösungsfunktion verwendet. Die Tatsache, dass s1 ein aufgelöstes Signal ist, bewirkt die Möglichkeit zur Verwendung mehrerer Quellen als Treiber dieses Signals. (Quellen sind Treiber in Prozesse und Output-Ports von Komponenten, die dem Signal zugeordnet sind.) Wenn eine Zuweisung für das Signal erfolgt, wird der Zustand dem Signal nicht direkt zugewiesen, stattdessen werden aus allen Werten aller Quellen, an denen das Signal angeschlossen ist, ein Array gebildet (einschließlich den neuen Wert aus der Zuweisung) und der Auflösungsfunktion übergeben.

Das von der Funktion zurückgegebene Ergebnis wird dem Signal als neuer Zustand zugewiesen.

Der VHDL-Mechanismus der Syntaxregel für Auflösungsfunktionen ist:

```
subtype_indication ⇐ [ <resolution_function_name> ]
type_mark [ range ( < range_attribute_name >| simple_expression ( to |
downto ) simple_expression ) | ( discrete_range { , … } ) ]
```

5.6. Bibliotheksreferenz für std_logic_1164 (Multivalue Logic System Package)

```
use std.textio.all;

package std_logic_1164 is
type std_ulogic is (
     'U',                              -- Uninitialized
     'X',                              -- Forcing Unknown
     '0',                              -- Forcing 0
     '1',                              -- Forcing 1
     'Z',                              -- High Impedance
     'W',                              -- Weak Unknown
     'L',                              -- Weak 0
     'H',                              -- Weak 1
     '-'                               -- Don't care
);

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

function resolved (s : std_ulogic_vector)
          return std_ulogic;

-- Subtyps of std_ulogic
subtype std_logic is resolved std_ulogic;
subtype std_logic_vector is (resolved) std_ulogic_vector;
subtype X01 is resolved std_ulogic range 'X' to '1';
subtype X01Z is resolved std_ulogic range 'X' to 'Z';
subtype UX01 is resolved std_ulogic range 'U' to '1';
subtype UX01Z is resolved std_ulogic range 'U' to 'Z';

-- Logical operators of argumenttyps std_ulogic
function "and" (l : std_ulogic; r : std_ulogic)
          return UX01;
function "nand" (l : std_ulogic; r : std_ulogic)
          return UX01;
function "or" (l : std_ulogic; r : std_ulogic)
          return UX01;
function "nor" (l : std_ulogic; r : std_ulogic)
          return UX01;
function "xor" (l : std_ulogic; r : std_ulogic)
          return UX01;
function "xnor" (l : std_ulogic; r : std_ulogic)
          return ux01;
function "not" (l : std_ulogic)
          return UX01;
```

```
function "and" (l, r : std_ulogic_vector)
         return std_ulogic_vector;
function "nand" (l, r : std_ulogic_vector)
         return std_ulogic_vector;
function "or" (l, r : std_ulogic_vector)
         return std_ulogic_vector;
function "nor" (l, r : std_ulogic_vector)
         return std_ulogic_vector;
function "xor" (l, r : std_ulogic_vector)
         return std_ulogic_vector;
function "xnor" (l, r : std_ulogic_vector)
         return std_ulogic_vector;
function "not" (l : std_ulogic_vector)
         return std_ulogic_vector;

-- Logical operators of argumenttyps std_ulogic_vector and std_ulogic
function  "and" (l : std_ulogic_vector; r : std_ulogic)
         return std_ulogic_vector;
function  "and" (l : std_ulogic; r : std_ulogic_vector)
         return std_ulogic_vector;
function  "nand" (l : std_ulogic_vector; r : std_ulogic)
         return std_ulogic_vector;
function  "nand" (l : std_ulogic; r : std_ulogic_vector)
         return std_ulogic_vector;
function  "or" (l : std_ulogic_vector; r : std_ulogic)
         return std_ulogic_vector;
function  "or" (l : std_ulogic; r : std_ulogic_vector)
         return std_ulogic_vector;
function "nor" (l : std_ulogic_vector; r : std_ulogic)
         return std_ulogic_vector;
function "nor" (l : std_ulogic; r : std_ulogic_vector)
         return std_ulogic_vector;
function "xor" (l : std_ulogic_vector; r : std_ulogic)
         return std_ulogic_vector;
function "xor" (l : std_ulogic; r : std_ulogic_vector)
         return std_ulogic_vector;
function "xnor" (l : std_ulogic_vector; r : std_ulogic)
         return std_ulogic_vector;
function "xnor" (l : std_ulogic; r : std_ulogic_vector)
         return std_ulogic_vector;

-- Logical operators of argumenttyps std_ulogic_vector
function "and" (l : std_ulogic_vector)
         return std_ulogic;
function "nand" (l : std_ulogic_vector)
         return std_ulogic;
```

```
function "or" (l : std_ulogic_vector)
         return std_ulogic;
function "nor" (l : std_ulogic_vector)
         return std_ulogic;
function "xor" (l : std_ulogic_vector)
         return std_ulogic;
function "xnor" (l : std_ulogic_vector)
         return std_ulogic;

-- Shift and rol operators of argumenttype std_ulogic_vector
function "sll" (l : std_ulogic_vector; r : integer)
         return std_ulogic_vector;
function "srl" (l : std_ulogic_vector; r : integer)
         return std_ulogic_vector;
function "rol" (l : std_ulogic_vector; r : integer)
         return std_ulogic_vector;
function "ror" (l : std_ulogic_vector; r : integer)
         return std_ulogic_vector;

-- Type cast functions
function To_bit (s : std_ulogic; xmap : bit := '0')
         return bit;
function To_bitvector (s : std_ulogic_vector; xmap : bit := '0')
         return bit_vector;
function To_StdULogic (b : bit)
         return std_ulogic;
function To_StdLogicVector (b : bit_vector)
         return std_logic_vector;
function To_StdLogicVector (s : std_ulogic_vector)
         return std_logic_vector;
function To_StdULogicVector (b : bit_vector)
         return std_ulogic_vector;
function To_StdULogicVector (s : std_logic_vector)
         return std_ulogic_vector;

-- Aliasnames of type cast functions
alias    To_Bit_Vector is
         To_bitvector[std_ulogic_vector, bit return bit_vector];
alias    To_BV is
         To_bitvector[std_ulogic_vector, bit return bit_vector];
alias    To_Std_Logic_Vector is
         To_StdLogicVector[bit_vector return std_logic_vector];
alias    To_SLV is
         To_StdLogicVector[bit_vector return std_logic_vector];
alias    To_Std_Logic_Vector is
         To_StdLogicVector[std_ulogic_vector return std_logic_vector];
```

```
alias     To_SLV is
          To_StdLogicVector[std_ulogic_vector return std_logic_vector];
alias     To_Std_ULogic_Vector is
          To_StdULogicVector[bit_vector return std_ulogic_vector];
alias     To_SULV is
          To_StdULogicVector[bit_vector return std_ulogic_vector];
alias     To_Std_ULogic_Vector is
          To_StdULogicVector[std_logic_vector return std_ulogic_vector];
alias     To_SULV is
          To_StdULogicVector[std_logic_vector return std_ulogic_vector];

-- Type cast functions to sybtyps
function  To_01 (s : std_ulogic_vector; xmap : std_ulogic := '0')
          return std_ulogic_vector;
function  To_01 (s : std_ulogic; xmap : std_ulogic := '0')
          return std_ulogic;
function  To_01 (s : bit_vector; xmap : std_ulogic := '0')
          return std_ulogic_vector;
function  To_01 (s : bit xmap : std_ulogic := '0')
          return std_ulogic;
function  To_X01 (s : std_ulogic_vector)
          return std_ulogic_vector;
function  To_X01 (s : std_ulogic)
          return X01;
function  To_X01 (b : bit_vector)
          return std_ulogic_vector;
function  To_X01 (b : bit)
          return X01;
function  To_X01Z (s : std_ulogic_vector)
          return std_ulogic_vector;
function  To_X01Z (s : std_ulogic)
          return X01Z;
function  To_X01Z (b : bit_vector)
          return std_ulogic_vector;
function  To_X01Z (b : bit)
          return X01Z;
function  To_UX01 (s : std_ulogic_vector)
          return std_ulogic_vector;
function  To_UX01 (s : std_ulogic)
          return UX01;
function  To_UX01 (b : bit_vector)
          return std_ulogic_vector;
function  To_UX01 (b : bit)
          return UX01;

function  "??" (l : std_ulogic)
          return boolean;
```

```
function  rising_edge (signal s : std_ulogic)
          return boolean;
function  falling_edge (signal s : std_ulogic)
          return boolean;
function  Is_X (s : std_ulogic_vector)
          return boolean;
function  Is_X (s : std_ulogic)
          return boolean;

-- the following operations are predefined
-- function "?=" (l, r : std_ulogic) return std_ulogic;
-- function "?=" (l, r : std_ulogic_vector) return std_ulogic;
-- function "?/=" (l, r : std_ulogic) return std_ulogic;
-- function "?/=" (l, r : std_ulogic_vector) return std_ulogic;
-- function "?<" (l, r : std_ulogic) return std_ulogic;
-- function "?<=" (l, r : std_ulogic) return std_ulogic;
-- function "?>" (l, r : std_ulogic) return std_ulogic;
-- function "?>=" (l, r : std_ulogic) return std_ulogic;
-- the following operations are predefined
-- function to_string (value : std_ulogic) return string;
-- function to_string (value : std_ulogic_vector) return string;

-- Alias function or function std_ulogic_vector to string
alias     to_bstring is
          to_string [std_ulogic_vector return string];
alias     to_binary_string is
          to_string [std_ulogic_vector return string];
function  to_ostring (value : std_ulogic_vector)
          return string;
alias     to_octal_string is
           to_ostring [std_ulogic_vector return string];
function  to_hstring (value : std_ulogic_vector)
          return string;
alias     to_hex_string is
          to_hstring [std_ulogic_vector return string];

-- Alias read/write procedures or read/write procedures
procedure
          read (l : inout line; value : out std_ulogic; good : out boolean);
procedure
          read (l : inout line; value : out std_ulogic);
procedure
          read (l : inout line; value : out std_ulogic_vector;
          good : out boolean);
procedure
          read (l : inout line; value : out std_ulogic_vector);
```

```
procedure
          write (l : inout line; value : in std_ulogic;
          justified : in side := right; field : in width := 0);
procedure
          write (l : inout line; value : in std_ulogic_vector;
          justified : in side := right; field : in width := 0);
alias     bread is
          read [line, std_ulogic_vector, boolean];
alias     bread is
          read [line, std_ulogic_vector];
alias     binary_read is
          read [line, std_ulogic_vector, boolean];
alias binary_read is
          read [line, std_ulogic_vector];
procedure
          oread (l : inout line; value : out std_ulogic_vector;
          good : out boolean);
procedure
          oread (l : inout line; value : out std_ulogic_vector);
alias     octal_read is
          oread [line, std_ulogic_vector, boolean];
alias     octal_read is
          oread [line, std_ulogic_vector];
procedure
          hread (l : inout line; value : out std_ulogic_vector;
          good : out boolean);
procedure
          hread (l : inout line; value : out std_ulogic_vector);
alias     hex_read is
          hread [line, std_ulogic_vector, boolean];
alias     hex_read is
          hread [line, std_ulogic_vector];
alias     bwrite is
          write [line, std_ulogic_vector, side, width];
alias     binary_write is
          write [line, std_ulogic_vector, side, width];
procedure
          owrite (l : inout line; value : in std_ulogic_vector;
          justified : in side := right; field : in width := 0);
alias     octal_write is
          owrite [line, std_ulogic_vector, side, width];
procedure
          hwrite (l : inout line; value : in std_ulogic_vector;
          justified : in side := right; field : in width := 0);
alias     hex_write is
          hwrite [line, std_ulogic_vector, side, width];
end package std_logic_1164;
```

5.6.1. VHDL-87, -93, und -2002

Im `std_logic_1164` Paket für die Versionen VHDL-87, -93 und -2002 wird `std_logic_vector` als ein verschiedener Typ, nicht als ein Subtyp von `std_ulogic_vector` erklärt. Operationen welche durch das Paket `std_ulogic_vector` zur Verfügung werden, sind auch für `std_logic_vector` überladen.

Das Paket für diese Versionen stellt die folgenden Ausdrücke nicht zur Verfügung:

- `array` / `scalar` und die logischen Reduktionsoperatoren;
- Verschiebungsoperatoren;
- Aliasnamen für die Umwandlungsfunktionen;
- Die `To_01`-Funktion;
- Den "`??`" Operator;
- Die „matching" Vergleichsoperatoren;
- Die `to_string`, `to_bstring`, `to_ostring`, und `to_hstring` Funktionen und zugeordnete Aliasnamen;
- Die `read`, `oread`, `hread`, `wirte`, `owrite`, und `hwrite` Operationen und zugeordnete Aliasnamen.

5.6.2. VHDL-87

Die überladene Version des `xnor` Operators ist nicht in der Version VHDL-87 des `standard-logic` Pakets beinhaltet.