

Array-Zuweisungen

Array-Zuweisungen können über die Position, den Namen oder gemischt erfolgen.

Beispiel

```
variable C      : bit_vector (0 to 3);  
variable H, I, J, K: bit;
```

mögliche Zuweisungen

```
C := "1010";
```

4-bit string

```
C := H & I & J & K;
```

Konkatenation

```
C := ('1', '0', '1', '0');
```

Aggregat

Aggregatzuweisungen

```
C := ('1', I, '0', J or K);
```

Position

```
C := (0 => '1', 3 => J or K, 1 => I, 2 => '0');
```

Name

```
C := ('1', I, others => '0');
```

gemischt

Ausschnitte (slices) werden über die Indizes gebildet.

Beispiel

```
variable A: bit_vector (3 downto 0);
```

```
variable B: bit_vector (8 downto 1);
```

```
...
```

```
B(6 downto 3) := A;
```

4 bit Slice von A

Mehrdimensionale Arrays

Mehrdimensionale Arrays werden durch Aufzählung von Indexbereichen erzeugt.

Beispiel

```
type MEMORY is array (0 to 7, 0 to 3) of bit;           8 × 4 bit Array
...
constant ROM: MEMORY := (('0','0','0','0'),
                           ('0','0','0','1'),
                           ('0','0','1','0'),
                           ('0','0','1','1'),
                           ('0','1','0','0'),
                           ('0','1','0','1'),
                           ('0','1','1','0'),
                           ('0','1','1','1'));
variable DATA_BIT    : bit;
...
DATA_BIT := ROM(5,3);           = '1'
```

Mehrdimensionale Arrays

Es lassen sich auch Arrays von Arrays bilden; die Indizes werden hierbei getrennt behandelt.

Beispiel

```
type WORD    is array (0 to 3) of bit;           4 bit Zeile
type MEMORY is array (0 to 7) of WORD;           8 × 4 bit Array
...
constant ROM: MEMORY := (('0','0','0','0'),
                          ('0','0','0','1'),
                          ...
                          ('0','1','1','1'));

variable DATA      : WORD;
variable DATA_BIT   : bit;
variable ADDR, INDEX : integer;
...
DATA      := ROM(ADDR);
DATA_BIT := ROM(ADDR)(INDEX);
```

Untertypen und Alias

Untertypen Zu vordefinierten, bzw. zu eigenen Typen lassen sich Untertypen bilden, dies geschieht durch Einschränkung der Wertebereiche und/oder bei Array-Typen Begrenzung der Indexbereiche.

Beispiel

```
subtype DIGIT is integer range 0 to 9;           Untertyp zu integer
...
variable MSD, LSD: DIGIT;

— ist äquivalent zu —
variable MSD, LSD: integer range 0 to 9;
```

Alias Deklarationen

Zu Typen, Unterprogrammen oder Objekten können Alias-Namen vergeben werden. Teilstrukturen komplexer Typen können so direkt über Namen referenziert werden.

Beispiel

```
signal ACNT: bit_vector(1 to 9);           vergl. voriges Beispiel
alias  SIGN: bit is ACNT(1);
alias  MSD : bit_vector(1 to 4) is ACNT(2 to 5);
alias  LSD : bit_vector(1 to 4) is ACNT(6 to 9);
...
SIGN := '1';                               Benutzung
MSD  := "1001";
LSD  := MSD;

— identische Zuweisung —
ACNT := "110011001";
```

Ausdrücke

Um Ausdrücke zu bilden, gelten die folgenden Regeln:

- Ausdrücke werden aus Operatoren und Objekten, Literalen, Funktionsaufrufen oder Aggregaten gebildet.
- Die Operatoren besitzen unterschiedliche Prioritäten (siehe Nummerierung).

Alle Operatoren innerhalb einer Gruppe haben die gleiche Priorität!

Oft werden Fehler in boole'schen Ausdrücken gemacht, da **and** und **or** gleichwertig sind.

- Gegebenenfalls muss die Reihenfolge der Auswertung durch Klammerung festgelegt werden.
- Wegen der Typbindungen müssen entweder explizite Angaben des Typs (Typqualifizierungen) oder Typkonvertierungen vorgenommen werden.

Der VHDL-Sprachstandard enthält die in der Tabelle aufgeführten Operatoren; sll. . . ror und xnor wurden in VHDL'93 [IEEE93a] ergänzt.

Syntax

1. logische Operatoren		Typ- a	Typ- b	Typ- $\langle op \rangle$
and	$a \wedge b$	bit bit_vector boolean	$= a$	$= a$
or	$a \vee b$	bit bit_vector boolean	$= a$	$= a$
nand	$\neg(a \wedge b)$	bit bit_vector boolean	$= a$	$= a$
nor	$\neg(a \vee b)$	bit bit_vector boolean	$= a$	$= a$
xor	$\neg(a \equiv b)$	bit bit_vector boolean	$= a$	$= a$
xnor	$a \equiv b$	bit bit_vector boolean	$= a$	$= a$
2. relationale Operatoren		Typ- a	Typ- b	Typ- $\langle op \rangle$
=	$a = b$	beliebiger Typ	$= a$	boolean
/=	$a \neq b$	beliebiger Typ	$= a$	boolean
<	$a < b$	skalarer Typ 1-dim. Array	$= a$	boolean
<=	$a \leq b$	skalarer Typ 1-dim. Array	$= a$	boolean
>	$a > b$	skalarer Typ 1-dim. Array	$= a$	boolean
>=	$a \geq b$	skalarer Typ 1-dim. Array	$= a$	boolean

3. schiebende Operatoren		Typ- a	Typ- b	Typ- $\langle op \rangle$
sll	$(a_{n-1-b} \dots a_0, 0_{b \dots 1})$	bit_vector bit/bool-Array	integer	$= a$
srl	$(0_{1 \dots b}, a_{n-1} \dots a_b)$	bit_vector bit/bool-Array	integer	$= a$
sla	$(a_{n-1-b} \dots a_0, a_{0,b \dots 1})$	bit_vector bit/bool-Array	integer	$= a$
sra	$(a_{n-1,1 \dots b}, a_{n-1} \dots a_b)$	bit_vector bit/bool-Array	integer	$= a$
rol	$(a_{n-1-b} \dots a_0, a_{n-1} \dots a_{n-b})$	bit_vector bit/bool-Array	integer	$= a$
ror	$(a_{b-1} \dots a_0, a_{n-1} \dots a_b)$	bit_vector bit/bool-Array	integer	$= a$
4. additive Operatoren		Typ- a	Typ- b	Typ- $\langle op \rangle$
+	$a + b$	integer real phys. Typ	$= a$	$= a$
-	$a - b$	integer real phys. Typ	$= a$	$= a$
&	$(a_n \dots a_0, b_m \dots b_0)$	skalärer Typ 1-dim. Array	a -Skalar/Array	a -Array
5. vorzeichen Operatoren		Typ- a	Typ- b	Typ- $\langle op \rangle$
+	$+a$	integer real phys. Typ		$= a$
-	$-a$	integer real phys. Typ		$= a$
6. multiplikative Operatoren		Typ- a	Typ- b	Typ- $\langle op \rangle$
*	$a * b$	integer real phys. Typ	$= a$	$= a$
/	a/b	integer real phys. Typ	$= a$	$= a$
mod	Modulus	integer	$= a$	$= a$
rem	Teilerrest	integer	$= a$	$= a$
7. sonstige Operatoren		Typ- a	Typ- b	Typ- $\langle op \rangle$
**	a^b	integer real	integer	$= a$
abs	$ a $	integer real phys. Typ		$= a$
not	$\neg a$	bit bit_vector boolean		$= a$

Typkonvertierung

Für die Standardtypen sind Konvertierungsfunktionen vordefiniert, insbesondere bei Benutzung der "Bitvektoren" `signed`, `unsigned` und `std_logic_vector` werden Konvertierungsfunktionen häufig benötigt.

`std_logic_1164`

```
to_bit      (<std_ulogic>[, <xMap>]) : bit
to_bitvector (<std_(u)logic_vector>[, <xMap>]) : bit_vector
to_stdulogic (<bit>) : std_ulogic
to_stdlogicvector (<bit_vector> | <std_ulogic_vector>) : std_logic_vector
to_stdulogicvector (<bit_vector> | <std_logic_vector>) : std_ulogic_vector
```

`numeric_std / numeric_bit`

```
to_integer (<signed>) : integer
to_integer (<unsigned>) : natural
to_signed (<integer>, <size>) : signed
to_unsigned (<natural>, <size>) : unsigned
```

In dem Beispiel wird ein **integer-Literal** in ein **std_logic_vector** umgerechnet.

Beispiel

```
signal SEL : std_logic_vector(3 downto 0);
...
SEL <= std_logic_vector(to_unsigned(3, 4));
```

Bei eigenen Typen müssen Konvertierungsfunktionen bei Bedarf durch den Benutzer angegeben werden.

Beispiel

```
type FOURVAL is ('X', '0', '1', 'Z');           vierwertige Logik
...
function STD_TO_FOURVAL (S: std_logic) return FOURVAL is
begin                                           Konvertierungsfunktion
    case S is
        when 'L' | '0'    => return '0';
        when 'H' | '1'    => return '1';
        when 'Z'          => return 'Z';
        when others       => return 'X';       'U' 'X' 'W' '-'
    end case;
end function STD_TO_FOURVAL;
...
signal  S  : std_logic;
signal  SF : FOURVAL;

...
SF <= STD_TO_FOURVAL(S);                      Aufruf in Signalzuweisung
```

Sequenzielle Beschreibungen

Die zentrale Rolle bei sequenziellen Beschreibungen spielt der Prozess. Das process-Statement wird für die Verhaltensbeschreibung von Architekturen benutzt, und begrenzt einen Bereich, in dem Anweisungen sequenziell abgearbeitet werden.

Das process-Statement selber ist eine konkurrente Anweisung, d.h. es können beliebig viele Prozesse gleichzeitig aktiv sein, ihre Reihenfolge im VHDL-Code ist irrelevant.

Syntax

```
[<label>:] process [(sensitivity list)< >] [is]
  [local declarations]
begin
  sequential statements
end process [<label>];
```

```
<local declarations> ::=
  {type <...>      } | {subtype <...>    } |
  {file <...>      } | {alias <...>      } |
  {function <...>  } | {procedure <...>  } |
  {constant <...> } |
  {variable <...> }
```

Hier noch eine Anmerkung: das optionale Label <label> ist bei der Simulation für das Debugging der Schaltung nützlich und sollte deshalb immer vergeben werden.

Das Beispiel simuliert die Auswahl des Minimums und Maximums aus drei Eingangswerten mit Hilfe von zwei Prozessen.

Beispiel

```
entity LOW_HIGH is
  port (A, B, C : in  integer;
        MI, MA  : out integer);
end entity LOW_HIGH;
```

Eingänge
Ausgänge

```
architecture BEHAV of LOW_HIGH is
begin
```

```
  L_P: process (A, B, C)
    variable LO : integer := 0;
  begin
    if A < B then LO := A;
    else LO := B;
  end if;
  if C < LO then LO := C;
  end if;
  MI <= LO after 1 ns;
end process L_P;
```

Minimum bestimmen

```
  H_P: process (A, B, C)
    variable HI : integer := 0;
  begin
    if A > B then HI := A;
    else HI := B;
  end if;
  if C > HI then HI := C;
  end if;
  MA <= HI after 1 ns;
end process H_P;
end architecture BEHAV;
```

Maximum bestimmen

Anweisungen

Signalzuweisung

Syntax

```
[⟨label⟩:] ⟨signalObj⟩ <= [⟨delay mode⟩] ⟨wave expression⟩;
```

Beispiel:

Signalzuweisung: `S <= A xor B after 10 ns;`

Variablenzuweisung

Syntax

```
[⟨label⟩:] ⟨variableObj⟩ := ⟨expression⟩;
```

Beispiel:

Variablenzuweisung: `Var := A xor B;`

If Verzweigung wie in Programmiersprachen; durch die Schachtelung von Bedingungen ergeben sich Präferenzen, die sich beispielsweise bei der Hardwaresynthese als geschachtelte (2-fach) Multiplexer wiederfinden.

Syntax

```
[⟨label⟩:] if ⟨condition⟩ then  
  ⟨sequential statements⟩  
{elsif ⟨condition⟩ then  
  ⟨sequential statements⟩}  
[else  
  ⟨sequential statements⟩]  
end if [⟨label⟩];
```

Case mehrfach-Verzweigung wie in Programmiersprachen; in der Hardwareumsetzung ergeben sich (im Idealfall) entsprechende Decodierer oder Multiplexer.

Syntax

```
[<label>:] case <expression> is  
  {when <choices> => <sequential statements>}  
end case [<label>];
```

<choices> ::=	<value>		genau ein Wert
	<value> { <value> }		Aufzählung
	<value> to <value>		Bereich
	others		alle übrigen

Für <expression> müssen alle möglichen Werte aufgezählt werden. Dies geht am einfachsten durch **when others** als letzte Auswahl.

Ein häufig gemachter Fehler ist, dass die metalogischen Werte von `std_logic` nicht berücksichtigt werden. In dem Decoderbeispiel ersetzt others nicht benötigte Fälle, während es im zweiten Beispiel den letzten Fall eines Multiplexers beschreibt — implizite Annahme, dass nur '0' oder '1' im Vektor enthalten sind.

Beispiel

```
case BCD is
  when "0000" => LED := "1111110";
  when "0001" => LED := "1100000";
  when "0010" => LED := "1011011";
  when "0011" => LED := "1110011";
  when "0100" => LED := "1100101";
  when "0101" => LED := "0110111";
  when "0110" => LED := "0111111";
  when "0111" => LED := "1100010";
  when "1000" => LED := "1111111";
  when "1001" => LED := "1110111";
  when others => LED := "-----";
end case;
```

Decoder: BCD zu 7-Segment

don't care: std_logic_1164

```
case SEL is
  when "00"  => O  <= A;
  when "01"  => O  <= B;
  when "10"  => O  <= C;
  when others => O  <= D;
end case;
```

4-fach Multiplexer

Unterprogramme

VHDL beinhaltet sowohl Prozeduren (mehrere Return-Werte via Parameter) als auch Funktionen (sie liefern genau einen Wert zurück) als Unterprogramme. Die Funktionen werden beispielsweise zur Typkonvertierung oder als Auflösungsfunktion benutzt.

Deklaration

Typischerweise werden Unterprogramme innerhalb des entsprechenden Kontexts definiert, also in einer architecture oder lokal in dem benutzenden process. Um Unterprogramme im Entwurf mehrfach zu nutzen, sollten sie in einem VHDL-Package deklariert werden.

Variablen

In Unterprogrammen können zwar lokale Variablen deklariert und benutzt werden, deren Werte sind aber nur bis zum Verlassen des Unterprogramms definiert — im Gegensatz zu Variablen im process, die einem lokalen Speicher entsprechen!

Function hat (meistens) mehrere Parameter und gibt genau einen Wert zurück—entspricht damit einem Ausdruck.

Syntax

```
function <functionId> [<parameter declaration>] return <typeId>;      nur Kopf

function <functionId> [<parameter declaration>] return <typeId> is
  [<local declarations>]
begin
  <sequential statements>                                           mit return
end [function] [<functionId>];

<parameter declaration> ::=
  ( [<class>] <formal list> : [in] <typeId> [:= <expression>] { ;
    [<class>] <formal list> : [in] <typeId> [:= <expression>] } )
<class> ::= constant|signal|file                                    Objektklasse
```

In dem Beispiel wird ein Bitvektor in eine Integer Zahl umgerechnet, dabei wird der Vektor als vorzeichenlose Zahl, mit MSB. . . LSB, interpretiert.

Procedure hat mehrere Parameter auf die lesend/schreibend zugegriffen werden kann. Bei der Deklaration wird dazu ein Modus als Wirkungsrichtung angegeben.

in - Eingangswert, darf nur gelesen werden.

out - Ausgangswert, darf nur auf der linken Seite von Zuweisungen stehen.

inout - Ein-/Ausgangswert, kann in der Prozedur universell eingesetzt werden.

Für die Parameter sind außer Variablen auch Signale zulässig. Bei Ausgangsparametern ist dabei auf den "passenden"

Zuweisungsoperator zu achten: $\leq$ oder $:=$

Prozeduren werden wie Anweisungen, sequenziell oder konkurrent, abgearbeitet.

Die Prozedur des Beispiels dient, wie die Funktion oben, der Umrechnung eines Vektors in eine vorzeichenlose Integer-Zahl. Das Argument ist vom Typ `std_logic_vector` und ein zusätzliches Flag zeigt an, ob in der Eingabe andere Werte außer '0' und '1' enthalten sind.

```
architecture ...
```

```
...
```

```
procedure SV_TO_INT (VEC : in std_logic_vector;  
                    INT : inout integer;  
                    FLG : out boolean) is
```

```
begin
```

```
    INT := 0;
```

```
    FLG := false;
```

```
    for I in VEC'range loop
```

```
        if VEC(I) = '1' then
```

```
            INT := INT * 2;
```

```
            INT := INT + 1;
```

```
        elsif VEC(I) /= '0' then
```

```
            FLG := true;
```

```
        end if;
```

```
    end loop;
```

```
end procedure SV_TO_INT;
```

```
begin
```

```
...
```

```
process ...
```

```
...
```

```
SV_TO_INT (XVEC, XINT, XFLG);
```

```
...
```

```
end process ...
```

lesen+schreiben: inout

Prozeduraufruf
hier sequenziell

Aufruf

Die Benutzung von Unterprogrammen im VHDL-Code kann sowohl im sequenziellen Kontext, also innerhalb von Prozessen oder anderen Unterprogrammen, als auch im konkurrenten Anweisungsteil einer Architektur erfolgen. Bei der Parameterübergabe hat man verschiedene Möglichkeiten die formalen Parameter (aus der Deklaration) durch aktuelle Parameter, bzw. Ausdrücke (bei in-Parametern), zu ersetzen.

- über die Position, entsprechend der Reihenfolge bei der Deklaration
- über den Namen: <formal parameter> => <actual parameter>
- auch Mischformen aus Position und Name sind möglich.
- **open** steht für nicht benutzte Ausgangs- oder Eingangsparameter mit Default-Wert.