VHDL handbook



Simply Better Results

Contents

LEXICAL ELEMENTS	4
Definition	
Character set	
Separators	
Delimiters	4
Identifiers	
LITERALS	
Numerical literals	
Enumeration literals	5
String literals.	
Bit string literals	
The NULL literal	
RESERVED WORDS	
SYNTAX	
Standards	
The Backus-Naur-format	7
TYPES AND OBJECTS	
Predefined types	
Predefined subtypes	
Types and subtypes	
ARRAY	
RECORD	
ACCESS TYPES (pointers)	
Aggregates	
GROUP	
ATTRIBUTE	
Constant declaration	
Variable declaration	
Signal declaration	
File declaration/File handling	19
File reading (TEXTIO)	20
File writing (TEXTIO)	21
ALIAS.	
LIBRARIES	
LIBRARY and USE	
DESIGN UNITS	24
PACKAGE DECLARATION	
PACKAGE BODY	
ENTITY A POLITICAL UP E	
ARCHITECTURE	
CONFIGURATION	
SEQUENTIAL STATEMENTS	29
WAIT	
IF	
CASE	
LOOP, NEXT and EXIT	
FUNCTION	
PROCEDURE	34

RETURN	35
Variable assignment	36
Signal assignment	37
CONCURRENT AND SEQUENTIAL STATEMENTS	38
ASSERT/REPORT	
Subprogram call	39
CONCURRENT STATEMENTS	40
PROCESS	
WHEN	
SELECT	
BLOCK	
GENERIC PARAMETERS AND GENERATE	44
GENERIC/GENERIC MAP	
GENERATE	
COMPONENTS	
Component declaration	
Component instantiation	
Default configuration	
Configuration specification	
Configuration declaration	
PREDEFINED ATTRIBUTES	
Attributes on types	
Attributes on arrays	
Attributes on signals	
Attributes on named entities	
IEEE	
VHDL standards	
Predefined packages	
STANDARD	
TEXTIO	
STD_LOGIC_1164 NUMERIC_BIT	
NUMERIC STD	
MATH_REAL	
MATH_COMPLEX	
VHDL GUIDE	
Introduction	
File notation	
Predefined packages	
VHDL syntax	
Simulation and synthesis	
VHDL'87 AND VHDL'93, DIFFERENCES	
INDEX	74

Lexical elements

LRM

§ 13

Definition

- The text of a design file is a sequence of lexical elements.
- Lexical elements are divided into the following groups:
 - delimiter
 - identifier (may be a reserved word)
 - abstract literal (integer or floating point type)
 - character literal (a graphical character surrounded by ', e.g.: 'H')
- string literal (a sequence of graphical characters surrounded by ", e.g.: "HARDI")
 - bit string literal (a sequence of extended digits* surrounded by ", e.g.: "011")
 - comment (preceded by -- and is valid until the end of the line)

Character set

The character set in VHDL'87 is 128 characters, in VHDL'93 it is 256 characters (see page 8, 56). The character set is divided into seven groups – Uppercase letters, Digits, Special characters, The space characters, Lowercase letters, Other special characters and format effector.

Separators

Separators are used to separate lexical elements. An example is the space character (SPACE).

Delimiters

A delimiter is one of the following characters or character combinations:

Identifiers

An identifier is either a name or a reserved word (see page 6). There are two kinds of identifiers:

- · Basic identifiers
 - Must begin with a letter.
 - May contain letters and digits.
 - May contain the character '_', but not as first or last character and not more than one in a row.
 - Are not case-sensitive.
- · Extended identifiers**
 - May contain letters and digits.
 - Begins and ends with the character '\'.
 - The \character may be included in the identifier, but must then be doubled, e.g.: \ab\\cd\
 - May include an unlimited amount of all graphical characters and in any order.
 - Are case-sensitive.

^{*} Possible values for an *extended digit* is determined by the base for the bit string literal (see page 5).

^{**} New to VHDL'93

Literals LRM

A literal is a written value of a type. The are in total five different kinds of literals

Numerical literals

[universal integer, universal real, literals of physical types]

Numerical literals of universal integer do not include a point, literals of universal real do include a point, while literals of physical types may include a point and must include a unit.

All numerical literals may include:

- ' 'to increase readability, e.g.: 1 000
- 'E' or 'e' to include an exponent, e.g.: 5E3 (i.e. 5000).
- '#' to describe a base, e.g.: 2#1010# (i.e. 10). It is possible to have a base between 2 and 16.

A physical type must include a space between its value and its unit, e.g.: 1 ns

Enumeration literals

[e.g.: BIT, BOOLEAN, CHARACTER]

Enumeration literals are graphical characters or identifiers (see page 4), e.g.: (reset, start, 'a', 'A').

String literals

[e.g.: STRING)

String literals are one-dimensional arrays including character literals. They always begin end end with a " (the " character may be included in the literal, but must then be doubled, e.g.: "A" character").

Bit string literals

[e.g: BIT VECTOR, STD LOGIC VECTOR*]

Bit string literals are one-dimensional arrays including extended digits (see page 4). They always begin and end with a ".

It is possible to include a base for a bit string literal. There are three bases:

- B Binary (possible values: 0 1).
- O Octal (possible values: 0 7). Each value is replaced by three values ('0' or '1').
- X Hexadecimal (possible values: 0 9, A F, a f). Each value is replaced by four values ('0' or '1').

A bit string literal may include '_' to increase readability, e.g.: "0100_ 0111".

The NULL literal

[NULL]

The NULL literal is only used for access types, i.e. pointers (see page 12), and imply that the pointer is empty, i.e. not pointing anywhere.

^{*} New to VHDL'93 (see page 73)



Reserved words

LRM

§ 13.9

abs	if	register
access	impure*	reject*
after	in	rem
alias	inertial*	report*
all	inout	return
and	is	rol*
architecture	label	ror*
array	library	select
assert	linkage	severity
attribute	literal*	signal
begin	loop	shared*
block	map	sla*
body	mod	sll*
buffer	nand	sra*
bus	new	srl*
case	next	subtype
component	nor	then
configuration	not	to
constant	null	transport
disconnect	of	type
downto	on	unaffected*
else	open	units
elsif	or	until
end	others	use

entity variable out exit package wait file port when for while postponed* function procedure with xnor* generate process generic pure* xor

group* range guarded record

^{*} New to VHDL'93

Syntax

Standards

The syntax in this handbook describes VHDL'93. At pages 70-73 the main differences between VHDL'87 and VHDL'93 are explained.

The Backus-Naur-format

All syntax in this handbook is described using the so called Backus-Naurformat. Here follows a short summary of the format:

- Words written using lower-case letters and possibly one or many hyphens, are used to denote a syntactical category, for example: entity-declaration.
- Reserved words are written with bold characters, for example: entity.
- Every replacement rule contains a left hand side expression and a right hand side expression separated by the sign →, which means "looks as" or "may be replaced with". The left hand side of the expression is always a syntactical category and may be replaced by the right hand side of the expression.
- |, a vertical line (the *pipe* sign) is used to separate many mutually exclusive alternatives.
- [], square brackets surround optional things that may occur once or not at all
- {}, braces surround optional things that may occur once, many times or not at all.
- (), parenthesis are used to clarify how and in which order a rule is evaluated
- Reserved words and characters surrounded by apostrophes, '', are included "as is" in the source code.
- Italicized words in the beginning of the name of a syntactical category give semantic information and have no syntactical meaning. For example entity-name-identifier is the same as identifier.

Types and objects Predefined types

LRM

§ 3, 4.3

Туре	Possible values	(by priority)
INTEGER	At least: -2147483647 - 2147483647	ABS ** * / MOD REM + - (sign) + - = /= < <= >>=
REAL	At least: -1.0E38 - 1.0E38	ABS ** * / + - (sign) + - = /= < <= >>=
TIME	2147483647 (fs, ps, ns,	ABS * / (sign) + - (sign) + - = /= < <= > >=
BIT	`0','1'	NOT = /= < <= > >= AND NAND OR NOR XOR XNOR*
BOOLEAN	FALSE, TRUE	NOT = /= < <= > >= AND NAND OR NOR XOR XNOR*
BIT_VECTOR	Unconstrained array of BIT	NOT & SLL* SRL* SRA* ROL* ROR* = /= < <= > >= AND NAND OR NOR XOR XNOR*

Туре	Possible values
CHARACTER	128 characters in VHDL'87 [ISO 646-1983] 256 characters in VHDL'93 [ISO 8859-1:1987(E)]
SEVERITY_LEVEL	NOTE, WARNING, ERROR, FAILURE
FILE_OPEN_KIND*	READ_MODE, WRITE_MODE, APPEND_MODE
FILE_OPEN_STATUS*	OPEN_OK, STATUS_ERROR, NAME_ERROR, MODE_ERROR
STRING	Unconstrained array of CHARACTER

Predefined subtypes

Туре	Possible values	Operators (by priority)
NATURAL		The same as for INTEGER
POSITIVE		The same as for INTEGER
DELAY_LENGTH*	0 fs - TIME'HIGH	The same as for TIME

^{*} New to VHDL'93

Types and subtypes

Syntax

LRM

§ 4.1, 4.2

```
type-declaration ->
    type identifier is type-indication ';'
subtype-declaration ->
    subtype identifier is subtype-indication ';'
subtype-indication, type-indication ->
    [resolution-function-name] type-name [range-constraint | index-constraint]
```

Examples

```
TYPE Weight IS RANGE 0 TO 10_000

UNITS
gr;
kg = 1000 gr;
ton = 1000 kg;
END UNITS;

ARCHITECTURE Behave OF Design IS
TYPE StateMachine IS (start,count,steady);
SIGNAL state, nextState : StateMachine;
BEGIN
...
END ARCHITECTURE Behave;

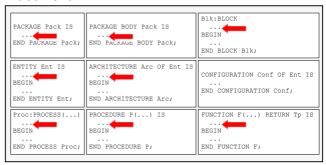
PROCESS
SUBTYPE MyArray IS BIT_VECTOR(7 DOWNTO 3);
BEGIN
...
END PROCESS;
```

Operators (by priority)

- Only relational operators (=, /=, <, <=, > and >=) are predefined for user-defined enumerated types. Other operators must be defined by the user.
- Logical operators are predefined for the predefined enumerated types BIT and BOOLEAN (see page 7).
- A subtype shares the same operators as its base type, including their priority.

Comments

- Operators must be defined by the user for user-defined enumerated types (except
 for the relational operators). It is therefore preferable to use subtypes since they
 share the same operators as their base type.
- Other relational operators than '=' and '/=' are dependant upon the order in the enumerated type. They shall therefore be used with care.
- A new type that is a part of an existing type, for example a part of the predefined unconstrained array BIT_VECTOR, must be declared as a subtype.



ARRAY

Syntax

LRM § 3.2.1

```
array-type-declaration ->
type identifier is array '(' type-name range '<>' { ',' type-name range '<>' } ')'
of element-subtype-indiciation ';' |
type identifier is array index-constraint of element-subtype-indication ';'
```

<u>element-</u>subtype-indication → [<u>resolution-function-name</u>] <u>type-name</u> [<u>range-constraint</u> | index-constraint]

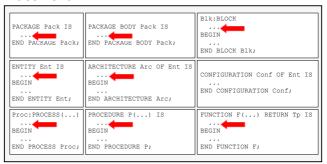
Examples

Operators (by priority)

1. NOT	(only for BIT and BOOLEAN)
2. &	
3. SLL*, SRL*, SLA*, SRA*, ROL*, ROR*	(only for BIT and BOOLEAN)
4. =, /=, <, <=, >, >=	
5. AND, OR, NAND, NOR, XOR, XNOR*	(only for BIT and BOOLEAN)
* New to VHDL'93	

Comments

- The logical operators and the shift operators are only defined for arrays with elements of the types BIT or BOOLEAN.
- · An array may be indexed in an unlimited amount of dimensions.
- The shift operators shifts either arithmetically (for example SLA) or logically (for example SLL). An arithmetic shift fills the last element with the same value it had before the shift, a logic shift fills it with '0' or FALSE.
- · An array may be indexed by any discrete type.





RECORD

Syntax

LRM § 3.2.2

```
record-type-declaration ->

type identifier is record
element-declaration
{ element-declaration }
end record [ record-type-name-identifier ] ';'
```

Examples

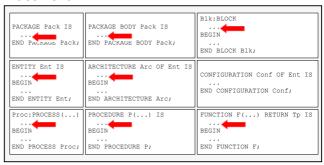
```
TYPE Clock IS RECORD
  Hour : INTEGER RANGE 0 TO 23;
 Min : INTEGER RANGE 0 TO 59;
  Sec : INTEGER RANGE 0 TO 59;
END RECORD Clock;
ARCHITECTURE Behave OF Design IS
 SIGNAL intTime : Clock := (0,0,0);
 PROCESS (clk)
 BEGIN
    IF clk'EVENT AND clk = '1' THEN
     IF intTime.Hour = 23 THEN
     REPORT "The time is " & INTEGER'IMAGE(intTime.Hour) & ":" &
                              INTEGER'IMAGE(intTime.Min)
                              INTEGER' IMAGE (intTime.Sec);
 END PROCESS;
END ARCHITECTURE Behave;
```

Operators (by priority)

• Only the relational operators (=, /=) are predefined for records.

Comments

- There are no predefined records in VHDL, but user-defined records can be very useful. A record holds several units within a "group" and the code gets easier to read
- · A record may contain an unlimited amount of elements.



ACCESS TYPES (pointers)

Syntax

LRM § 3.3

```
access-type-declaration ->
type identifier is access subtype-indication ';'

subtype-indication ->
[resolution-function-name] type-name [range-constraint | index-constraint]
```

Examples

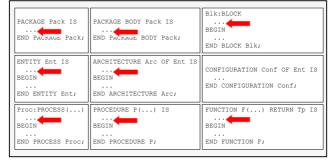
```
TYPE ListElement; -- Incomplete type declaration
TYPE ListPointer IS ACCESS ListElement;
TYPE ListElement IS RECORD
            : INTEGER RANGE 0 TO 31;
 NextPoint : ListPointer;
END RECORD ListElement;
VARIABLE list1, list2 : ListPointer;
list1 := NEW ListElement;
list1.Data := inData;
list1.NextPoint := NEW ListElement' (inData2, NULL);
IF list1.ALL = list2.ALL THEN ... -- If the elements pointed out
                                   -- have the same values
IF list1 = list2 THEN ... -- If the pointers point at the same
                           -- object
DEALLOCATE (list1.NextPoint); -- Remove "list1.NextPoint"
DEALLOCATE (list1);
                              -- Remove the pointer "list1"
```

Operators (by priority)

- Only the relational operators (=, /=) are predefined for access types (pointers).
- Two pointers are equal only if they point at the same object.

Comments

- · Access types (pointers) are not synthesizable.
- · An object of an access type must be a variable.
- Access types are for example used when flexible handling of the computers' memory is desired, for example when simulating large memories.
- There is one predefined access type in the package TEXTIO (see page 57). It is
 the type LINE that specifies which line that has been read or that shall be written
 to.



Aggregates

Syntax

```
LRM
§ 7.3.2
```

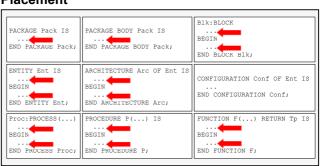
```
aggregate ->
    '(' element-association { ',' element-association } ')'
element-association ->
    [ choices '=>' ] expression
    choices -> choice { | choice }
    choice -> simple-expression | discrete-range | element-simple-name | others
```

Examples

```
TYPE Clock IS RECORD
   Hour : INTEGER RANGE 0 TO 23;
          : INTEGER RANGE 0 TO 59;
  Sec : INTEGER RANGE 0 TO 59;
END RECORD Clock;
TYPE Matrix IS ARRAY (0 TO 1, 0 TO 1) OF BIT;
SUBTYPE MyArray IS BIT_VECTOR(2 TO 5);
CONSTANT allZero : MyArray := (OTHERS => '0');
SIGNAL currentTime, alarmTime : Clock;
VARIABLE m1, m2 : Matrix;
VARIABLE v1, v2 : MyArray;
currentTime <= (10,15,5);
currentline <= (10,13,3);
alarmTime <= (Hour => 10, Min => 15, Sec => 5);
m1 := (('0','1'), (OTHERS => '0')); -- "01","00"
m2 := (OTHERS => (OTHERS => '1')); -- "11","11"
v1 := ('0', '1', '1', '1'); -- "0111"
v2 := (3 => '0', OTHERS => '1'); -- "1011"
                := ("0000","1111"); -- v1 = "0000", v2 = "1111"
-- For a BIT VECTOR this assignment is easier to write:
v2 := "1011";
```

Comments

- Aggregates are used to assign values to arrays and records. Both types and objects can get values using aggregates.
- It is possible to use named association (for example "alarmTime" above) or
 positional association (for example "currentTime" above). Named association is
 preferable since then the order of the parameters does not impact the assignment.
- OTHERS is used to assign values to the elements not already assigned. OTHERS
 must be placed as the last association in the aggregate.
- For records, but not for arrays, it is possible (but not recommendable) to mix named and positional association. The only rule is that the positional associations must be placed before the named.



GROUP

Syntax

group-template-declaration ->

{ ','(name | character-literal) } ')' ';

```
LRM
```

```
§ 4.6, 4.7
group identifier is '(' entity-class [ '�' ] { ',' entity-class [ '�' ] } ')' ';'
```

Examples

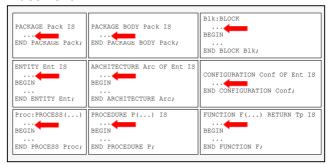
group-declaration ->

```
ENTITY Mux IS
                       STD_ULOGIC;
STD_ULOGIC_VECTOR(1 DOWNTO 0);
  PORT(a, b, c : IN
       choose : IN STD_ULOGIC_V
q : OUT STD_ULOGIC);
END ENTITY Mux;
ARCHITECTURE Behave OF Mux IS
  GROUP Ports IS (SIGNAL <>);
                                   -- Create a group template
  GROUP InPorts : Ports (a,b,c); -- Create a group of the template
  GROUP OutPort : Ports (q);
                                  -- Create another group
  GROUP InToOut IS (GROUP, GROUP); -- A 2-dim group template
  GROUP Timing : InToOut (InPorts,OutPort); -- The final group
  ATTRIBUTE synthesis_maxdelay : TIME; -- Use the groups ATTRIBUTE synthesis_maxdelay OF Timing : GROUP IS 9 ns;
BEGIN
  PROCESS (a, b, c, choose)
  REGIN
    CASE choose IS
                   => q <= a;
      WHEN "00"
      WHEN "01"
                    => q <= b;
      WHEN "10"
                  => q <= c;
      WHEN OTHERS => NULL;
    END CASE:
  END PROCESS;
END ARCHITECTURE Behave;
```

group identifier ':' group-template-name '(' (name | character-literal)

Comments

- · Groups are new to VHDL'93. They are intended to make the user-defined attributes (see page 15) more powerful by giving the possibility to set an attribute for a whole group, not just on named entities each by each.
- The usage of a group contains two parts group template declaration and group declaration. The group template declaration creates a template defining the design of the group, while the group declaration creates the group and includes its members.
- '<>' (pronounced: box), means that any number of elements of an entity class may be included in the group. '>' may only be used as the last element in a template list.



ATTRIBUTE

Syntax

LRM

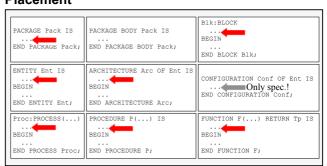
```
§ 4.4, 5.1
attribute-declaration ->
   attribute identifier ':' ( type-name | subtype-name ) ';'
attribute-specification ->
   attribute attribute-name-identifier of entity-class-name-list ':' (entity | architec.
      configuration | procedure | function | package | type | subtype | constant | signal
     variable | component | label | literal | units | group | file ) is expression ';
entity-class-name-list ->
   entity-class-tag [ signature ] { ',' entity-class-tag [ signature ] } | others | all
```

Examples

```
TYPE StateMachine IS (start, count, stop);
ATTRIBUTE syn_encoding : STRING; -- State machine encoding
ATTRIBUTE syn encoding OF StateMachine : TYPE IS "onehot";
ARCHITECTURE Behave OF Mux IS
  GROUP Ports IS (SIGNAL <>);
  GROUP InPorts : Ports (a,b,c);
  GROUP OutPort : Ports (q);
  GROUP InToOut IS (GROUP, GROUP);
  GROUP Timing : InToOut (InPorts,OutPort);
  ATTRIBUTE synthesis_maxdelay : TIME; -- Maximum delay
  ATTRIBUTE synthesis maxdelay OF Timing : GROUP IS 9 ns;
BEGIN
END ARCHITECTURE Behave;
ENTITY Count IS
  PORT(clock : IN BIT;
       counter : OUT INTEGER RANGE 0 TO 15);
  ATTRIBUTE pinnum : STRING; -- Pin numbering
ATTRIBUTE pinnum OF clock : SIGNAL IS "P2";
ATTRIBUTE pinnum OF counter : SIGNAL IS "P12,P14,P17,P21";
END ENTITY Count;
```

Comments

- The usage of attributes contains two parts attribute declaration and attribute specification. The attribute declaration defines the attribute while the attribute specification uses the attribute on a named entity, for example a signal, a variable, a function, a type etc.
- In VHDL'93 it is possible to group (GROUP, see page 14) a number of named entities and then define an attribute for the whole group.
- Attributes are used for documentation purposes, but above all to give commands to downstream tools, for example synthesis tools. Attributes used for downstream tools are not defined in the VHDL standard and differ between different tools.



Constant declaration

Syntax

LRM § 4.3.1.1

```
constant-declaration ->
  constant identifier { ',' identifier } ':' subtype-indication [ ':=' expression ] ','
subtype-indication ->
  [ resolution-function-name ] type-name [ range-constraint | index-constraint ]
```

Examples

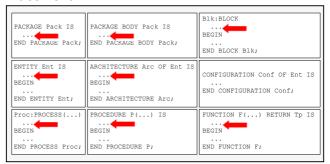
```
CONSTANT zero : STD_LOGIC_VECTOR(0 TO 3) := (OTHERS => \0');
SIGNAL data : STD_LOGIC_VECTOR(zero'RANGE) := zero;
CONSTANT bits : BIT_VECTOR := x"0FA3";

PACKAGE Useful IS
CONSTANT pi : REAL; -- declare pi here ...
CONSTANT one : NATURAL := 1;
CONSTANT two : NATURAL := 2*one;
CONSTANT four : NATURAL := two + one + one;
END PACKAGE Useful;

PACKAGE BODY Useful IS
CONSTANT pi : REAL := 3.1415; -- ... and give it its value here
END PACKAGE BODY Useful;
```

Comments

- A constant gets its value when it is declared and may only be read.
- A constant can be declared using a so called deferred constant declaration. It is
 then declared and named in the package declaration, but gets its value first in the
 package body. This coding style hides the value of the constant, it is not shown in
 the package declaration, and the user is not tempted to use the value directly. Another advantage is if the value of the constant is modifed. Then only the package
 body needs to be recompiled.



Variable declaration

Syntax

LRM § 4.3.1.3

```
variable-declaration ->
    [shared] variable identifier { ',' identifier } ':' subtype-indication
    [ ':=' expression] ';'

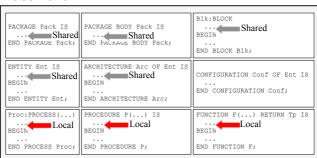
subtype-indication ->
    [resolution-function-name] type-name [range-constraint | index-constraint]
```

Examples

```
VARIABLE databus : STD LOGIC VECTOR(3 DOWNTO 0);
Clocked: PROCESS (clk)
 VARIABLE internal : REAL := 0.0;
BEGIN
 IF clk'EVENT AND clk = '1' THEN
    internal := internal + 1.0;
             <= internal;
    output
 END IF;
END PROCESS Clocked;
ARCHITECTURE Behave OF Design IS
 SHARED VARIABLE data : INTEGER := 0;
BEGIN
 One: PROCESS
 REGIN
   data := 0;
 END PROCESS One;
 Another: PROCESS
 BEGIN
    IF data = 0 THEN
 END PROCESS Another;
END ARCHITECTURE Behave;
```

Comments

- Variables are local in processes or subprograms and are therefore declared there.
 The exception is shared variables that are global just as signals. Since variables do not have the possibility to handle concurrent assignment from more than one process, as signals can, shared variables are to be avoided.
- · A variable has no direction and may therefore be both read and written.
- Variables are both assigned and get default values using ':='.



Signal declaration

Syntax

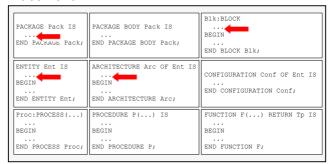
LRM § 4.3.1.2

```
signal-declaration ->
signal identifier { ',' identifier } ':' subtype-indication [ register | bus ]
[ ':=' expression ] ';'
subtype-indication ->
[ resolution-function-name ] type-name [ range-constraint | index-constraint ]
```

Examples

Comments

- Signals are global in an architecture or in a block. They may therefore not be locally declared in a process or in a function.
- A signal declared in a PORT must have a direction, while an internal signal (declared in an architecture, a block or in a package) does not have a direction.
- Ports of the mode OUT may only be assigned, not read. The workaround is to use internal variables or the attribute 'DRIVING VALUE (see page 54).
- Signals are assigned using '<=' but get default values using ':='.



File declaration/File handling

Syntax 5 4 1

LRM

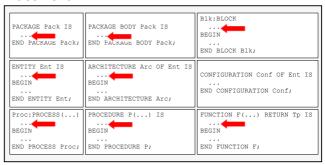
```
§ 3.4, 4.3.1.4
file-declaration ->
   file identifier { '.' identifier } ':' subtype-indication
     [ [ open file-open-kind-expression ] is string-expression ] ';'
subtype-indication ->
  type-name [ range-constraint | index-constraint ]
```

Examples

```
ARCHITECTURE Behave OF Files IS
  TYPE Times
               IS FILE OF TIME;
  TYPE Stimuli IS FILE OF BIT VECTOR (3 DOWNTO 0);
 FILE outFile : Stimuli OPEN WRITE_MODE IS "C:\proj\out.bin";
                          OPEN READ MODE IS "C:\proj\time.bin";
 FILE timeFile : Times
 FILE inData : Stimuli;
BEGIN
    VARIABLE ok : FILE OPEN STATUS;
    VARIABLE t : TIME;
    VARIABLE by : BIT VECTOR(3 DOWNTO 0);
    FILE OPEN (ok, inData, "C:\proj\indata.bin", READ MODE);
    IF o\overline{k} = OPEN OK THEN
      WHILE (NOT ENDFILE (inData) AND NOT ENDFILE (timeFile)) LOOP
        READ(timeFile,t);
        READ(inData,bv);
        WAIT FOR t;
        WRITE (outFile, bv);
   FILE CLOSE (outFile);
   FILE_CLOSE(timeFile);
    FILE CLOSE (inData);
END ARCHITECTURE Behave;
```

Comments

- A file may contain all types in VHDL except for files, access types (pointers) and multidimensional arrays.
- · The VHDL standard does not define how data shall be stored in a file. It is therefore preferable to use text files since they are easy to read and since there is a number of predefined procedures to handle them. The procedures are defined in the package TEXTIO (see page 57). By using this standardized package it is possible to move the files between different simulation environments.
- File handling in VHDL has been considerably modified between VHDL'87 and VHDL'93 (see page 71). The modifications are not backwards compatible.
- FILE OPEN and FILE CLOSE are new to VHDL'93.



File reading (TEXTIO)

Syntax

LRM § 14.3

```
(No syntax)
```

Examples

```
USE STD.TEXTIO.ALL; -- Make TEXTIO accessible
ARCHITECTURE Behave OF Filehandling IS
  TYPE Reg IS RECORD
   Number : POSITIVE;
    Sum
           : NATURAL;
  END RECORD;
  FILE MyFile : TEXT OPEN READ MODE IS "data.txt";
REGIN
  PROCESS
    VARIABLE 1
                      : LINE; -- declare a line variable
    VARIABLE fNumber : POSITIVE;
    VARIABLE fSum
                      : NATURAL;
  REGIN
    IF NOT ENDFILE (MyFile) THEN -- If the file is not empty ..
      READLINE (MyFile, 1);
                                   -- ... read a line ...
      ASSERT 1'LENGTH /= 0;
                                   -- ... if it isn't empty ..
      READ(1, fNumber);
                                   -- \dots read the 1st element
      READ(1, fSum);
                                   -- ... and then the 2nd element
END ARCHITECTURE Behave;
VARIABLE 1 : LINE;
1 := NEW STRING' ("My question");
WRITELINE(OUTPUT,1); -- Writes "My question" in the simulator READLINE(INPUT.1); -- Reads the answer from the keyboard
READLINE (INPUT, 1);
```

Comments

- When reading a text file, a whole line must first be read. That is done using READLINE. After that each element in the line is read using a number of READ operations. Each object assigned by a value from the file must be of the same type as the value. It is therefore important to know the order of the elements in the file.
- The predefined file INPUT (see example above) reads a value from the keyboard or from an input file during simulation. The handling of INPUT is tool dependant.
- · Files are not synthesizable.
- File handling in VHDL has been considerably modified between VHDL'87 and VHDL'93 (see page 71). The modifications are not backwards compatible. The examples above are according to VHDL'93.

	PACKAGE BODY Pack IS END PACKAGE BODY Pack;	Blk:BLOCK BEGIN END BLOCK Blk;
ENTITY Ent IS BEGIN END ENTITY Ent;	ARCHITECTURE Arc OF Ent IS BEGIN END ARCHITECTURE Arc;	CONFIGURATION Conf OF Ent IS END CONFIGURATION Conf;
Proc:PROCESS() BEGIN END PROCESS Proc;	PROCEDURE P() IS BEGIN END PROCEDURE P;	FUNCTION F() RETURN TP IS BEGIN END FUNCTION F;

File writing (TEXTIO)

LRM

§ 14.3

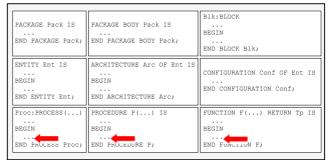
```
Syntax
(No syntax)
```

Examples

```
USE STD.TEXTIO.ALL; -- Make TEXTIO accessible
ARCHITECTURE Behave OF Filehandling IS
  TYPE Reg IS RECORD
    Number : POSITIVE;
            : NATURAL;
  END RECORD;
  FILE MyFile : TEXT OPEN WRITE MODE IS "data.txt";
REGIN
  PROCESS
    VARIABLE 1
                        : LINE; -- declare a line variable
    VARIABLE fNumber : POSITIVE;
                        : NATURAL;
    VARIABLE fSum
  REGIN
    WRITE(1, fNumber); -- Write an element to a line ...
    WRITE(1, ' '); -- ... separate with a blank ... WRITE(1,fSum); -- ... another element to the same line
    {\tt WRITELINE}\,({\tt MyFile,l})\,; -- ... and write the line to the file
  END PROCESS;
END ARCHITECTURE Behave;
VARIABLE 1 : LINE;
1 := NEW STRING'("My question");
WRITELINE(OUTPUT,1); -- Writes "My question" in the simulator READLINE(INPUT,1); -- Reads the answer from the keyboard
READLINE (INPUT, 1);
```

Comments

- When writing text to a text file, all elements are first written to a line using WRITE and finally the whole line is written to the file using WRITELINE.
- The predefined file OUTPUT (see example above) writes to the screen or to an
 output file during simulation. No information is given about the current simulation time as it is when using REPORT (see page 38). The handling of OUTPUT is
 tool dependant.
- · Files are not synthesizable.
- File handling in VHDL has been considerably modified between VHDL'87 and VHDL'93 (see page 71). The modifications are not backwards compatible. The examples above are according to VHDL'93.



ALIAS

Syntax

LRM § 4.3.3

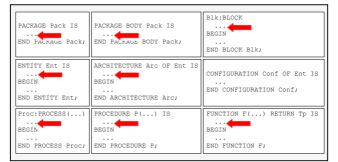
```
alias-declaration ->
    alias (identifier | character-literal | operator-symbol ) [ ':' subtype-indication ]
    is name [ signature ] ';'
signature -> 'i' [ type-name { ',' type-name } ] [ return type-name ] ']'
subtype-indication ->
[ resolution-function-name ] type-name [ range-constraint | index-constraint ]
```

Examples

```
SIGNAL data : SIGNED(7 DOWNTO 0);
ALIAS sign : BIT IS data(7);
REPORT "The sign bit is " & BIT'IMAGE(sign);
-- The alias below gives a certain index range to be able to
  calculate n not depending on v's index range
FUNCTION Bv2Natural(v : IN BIT VECTOR) RETURN NATURAL IS
 ALIAS aliasV : BIT VECTOR (v'LENGTH - 1 DOWNTO 0) IS v;
 VARIABLE n : NATURAL;
REGIN
  FOR i IN aliasV'RANGE LOOP
    IF aliasV(i) = '1' THEN
      n := n + 2**i;
 RETURN n;
END FUNCTION Bv2Natural;
CONSTANT MaxDelayClockToPad : TIME := 15 ns;
ALIAS MDC2P : TIME IS MaxDelayClockToPad;
-- MDC2P = MaxDelayClockToPad
SUBTYPE MyVeryVeryLongTypeName IS BIT VECTOR(3 DOWNTO 0);
ALIAS ShortName IS MyVeryVeryLongTypeName;
```

Comments

- An alias creates an alternative name for an existing object. It does not create a new object. It is often used to easier get access to elements in one-dimensional arrays
- In VHDL'87 it is only possible to declare aliases for objects. In VHDL'93 it is
 possible also for subprograms, operators, types and for all named entities except
 "labels", "loop parameters" and "generate parameters".



LIBRARY and USE

Syntax

LRM § 11.2, 10.4

```
library-clause ->
library logical-name-identifier { ',' logical-name-identifier } ';'
use-clause ->
use selected-name { ',' selected-name } ';'
selected-name -> prefix '.' suffix
```

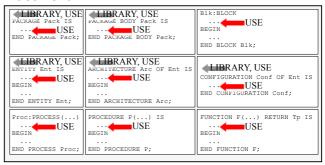
Examples

```
LIBRARY IEEE, HARDI;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL, HARDI.Devices.ALL;

LIBRARY PACK;
USE WORK.OnePackage.MyType; -- Select a unit from the package
USE PACK.AnotherPackage.ALL; -- Select the whole package
-- The following declarations exist implicitely:
LIBRARY WORK, STD;
USE STD.STANDARD.ALL;
```

Comments

- The LIBRARY clause declares the name of a library. After that the desired parts
 of the library are selected using USE.
- · Via ALL everything in a library or package is selected.
- The libraries WORK and STD and also the package STD.STANDARD are always accessible.
- LIBRARY must be placed before the design unit that shall use it, but USE can
 also be placed within the design units. It is however good practice to place both
 LIBRARY and USE before the design units that shall use them.
- An architecture has the same LIBRARY and USE as its entity. A package body
 has the same as its package declaration. A configuration declaration has the
 same as its entity and architecture.



PACKAGE DECLARATION

Syntax

LRM § 2.5

Examples

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

PACKAGE Various IS

CONSTANT hold: TIME; -- deferred constant
PROCEDURE Push(SIGNAL button: INOUT STD_LOGIC; hold: TIME);
TYPE Display IS RECORD
...

END RECORD;
COMPONENT Clock IS

PORT (mode, set, reset, clk: IN STD_LOGIC;
LCD : OUT Display;
backPlane, alarmSignal: OUT STD_LOGIC);
END COMPONENT Clock;
END PACKAGE Various;
```

Comments

- A package is used for declarations that shall be shared by several design units.
 The package contains two parts package declaration that contains the declarations and package body that implements the contents of the package.
- A package declaration does not need a package body.
- There is a number of packages available in VHDL. The packages standardized by IEEE are listed on pages 56-65.

	PACKAGE BODY Pack IS END PACKAGE BODY Pack;	Blk:BLOCK BEGIN END BLOCK Blk;
ENTITY Ent IS BEGIN END ENTITY Ent;	ARCHITECTURE Arc OF Ent IS BEGIN END ARCHITECTURE Arc;	CONFIGURATION Conf OF Ent IS END CONFIGURATION Conf;
Proc:PROCESS() BEGIN END PROCESS Proc;	PROCEDURE P() IS BEGIN END PROCEDURE P;	FUNCTION F() RETURN Tp IS END FUNCTION F;



PACKAGE BODY

Syntax LRM § 2.6

```
| package-body → | package body package-name-identifier is | { subprogram-declaration | subprogram-body | type-declaration | subtype-declaration | constant-declaration | shared-variable-declaration | file-declaration | alias-declaration | use-clause | group-template-declaration | group-declaration } | end [ package body ] [ package-name-identifier ] ';'
```

Examples

```
LIBRARY IEEE; -- These two lines could have been

USE IEEE.STD_LOGIC_1164.ALL; -- left out since they already are
-- declared in the package
-- declaration

PACKAGE BODY Various IS

CONSTANT hold: TIME:= 100 ns; -- deferred constant
PROCEDURE Push(SIGNAL button: INOUT STD_LOGIC; hold: TIME) IS
BEGIN
button <= '0', '1' AFTER hold;
WAIT FOR 2*hold;
END PROCEDURE Push;
END PACKAGE BODY Various;
```

Comments

- A package body is used to implement a package. Usually subprograms are implemented and so called deferred constants, constants declared but not assigned values in a package declaration, are assigned values.
- It is possible to declare types and subprograms in a *package body*, but they are then only accessible within the *package body*.
- There is a number of packages available in VHDL. The packages standardized by IEEE are listed on pages 56-65.
- A package body must have a package declaration and they must have the same name. The package body is compiled after its package declaration.

	PACKAGE BODY Pack IS END PACKAGE BODY Pack;	Blk:BLOCK BEGIN END BLOCK Blk;
ENTITY Ent IS BEGIN END ENTITY Ent;	ARCHITECTURE Arc OF Ent IS BEGIN END ARCHITECTURE Arc;	CONFIGURATION CONF OF Ent IS END CONFIGURATION CONF;
Proc:PROCESS() BEGIN END PROCESS Proc;	PROCEDURE P() IS BEGIN END PROCEDURE P;	FUNCTION F() RETURN Tp IS BEGIN END FUNCTION F;

ENTITY

Syntax

```
LRM
§ 1.1
```

```
entity-declaration ->
entity identifier is

[formal-generic-clause]

[formal-port-clause]

{subprogram-decl. | subprogram-body | type-decl. | subtype-decl. | constant-decl. | signal-decl. | sihared-variable-decl. | file-decl. | alias-decl. | attribute-decl. | attribute-decl. | attribute-decl. | group-template-decl. |
group-decl. }

[begin

{ concurrent-assertion-statement | passive-procedure-call | passive-process-statement } ]
end [entity | [entity-name-identifier] ';'
```

Examples

```
LIBRARY IEEE, TYPES, HARDI;
USE IEEE.STD_LOGIC_1164.ALL;
USE TYPES.TypePackage.ALL;
USE HARDI.Timing.ALL;

ENTITY Design IS

GENERIC (n : NATURAL);
PORT (data : IN STD_LOGIC_VECTOR(n DOWNTO 0);
clk : IN STD_LOGIC;
outData : OUT OutDataType);

BEGIN
PeriodCheck(clk, MaxPeriod); -- Passive procedure call
END ENTITY Design;
```

Comments

- An entity is the interface of a design.
- The *entity* contains a declaration part and a statement part. The declaration part
 declares the interface of the design, the statement part may contain passive statements, i.e. statements not assigning signals. The purpose of the statement part is
 to be able to verify the behavior of the signals declared in the declaration part, i.e.
 the ports.
- Each entity in a design must have a unique name while each entity can have several architectures. Everything declared in an entity is automatically accessible in its architectures.
- Note the order of the declarations in the declaration part of the *entity*.

	PACKAGE BODY Pack IS END PACKAGE BODY Pack;	Blk:BLOCK BEGIN END BLOCK Blk;
ENTITY Ent IS BEGIN END ENTITY Ent;	ARCHITECTURE Arc OF Ent IS BEGIN END ARCHITECTURE Arc;	CONFIGURATION CONF OF Ent IS END CONFIGURATION CONF;
Proc:PROCESS() BEGIN END PROCESS Proc;	PROCEDURE P() IS BEGIN END PROCEDURE P;	FUNCTION F() RETURN Tp IS BEGIN END FUNCTION F;



ARCHITECTURE

LRM

§ 1.2

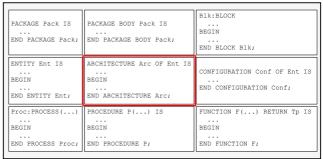
```
Syntax 5 4 1
architecture-body ->
   architecture identifier of entity-name is
      { subprogram-declaration | subprogram-body | type-declaration |
       subtype-declaration | constant-declaration | signal-declaration |
       shared-variable-declaration | file-declaration | alias-declaration |
       component-declaration | attribute-declaration | attribute-specification |
       configuration-specification | disconnect-specification |
       group-template-declaration | group-declaration }
   begin
      { concurrent-statement }
   end [ architecture ] [ architecture-name-identifier ] ';'
```

Examples

```
ARCHITECTURE Behave OF Design IS
  FUNCTION InternalCalc(v : STD LOGIC VECTOR(7 DOWNTO 0))
    RETURN BIT VECTOR (1 TO 4) IS
  BEGIN
  END FUNCTION InternalCalc;
  SUBTYPE MyArray IS BIT_VECTOR(3 DOWNTO 0);
SIGNAL internal : MyArray;
  SHARED VARIABLE data : STD LOGIC VECTOR (1 TO 8);
BEGIN
  PROCESS (clk)
  BEGIN
  END PROCESS;
  internal <= InternalCalc(data);
END ARCHITECTURE Behave;
```

Comments

- An architecture is the implementation of an entity. It contains a declaration part and a statement part. The declaration part may for example declare types, components and subprograms that shall be internal within the architecture.
- · An entity may have an unlimited amount of architectures. The architectures associated with the same entity must have unique names.
- · An entity and its architectures belong to the same declarative region. Everything declared in the *entity* is therefore accessible in its architectures.
- · At synthesis or simulation an architecure must be selected for each entity. If nothing else is specified, for example using a configuration (see page 28, 49-50), the last compiled architecture is used.



CONFIGURATION

Syntax

LRM § 1.3

```
configuration-declaration →
configuration identifier of entity-name is
{ use-clause | attribute-specification | group-declaration }
for block-specification
{ use-clause }
{ block-configuration | component-configuration }
end [ configuration ] [ configuration-name-identifier ] ';'

block-specification →
architecture-name | block-statement-label |
generate-statement-label [ '(' index-specification ')' ]
```

Examples

```
LIBRARY COMP;

CONFIGURATION MyConfiguration OF MyEntity IS

FOR MyArchitecture

FOR ALL: SubBlock USE ENTITY WORK.Ent(Arc)

GENERIC MAP(...);

END FOR;

FOR SubBlock2

FOR C1: AComponent USE ENTITY COMP.Ent2(Arc2);

END FOR;

END CONFIGURATION MyConfiguration;
```

Comments

- A CONFIGURATION (configuration declaration) is a separate design unit and is
 used to associate the entities and architectures in a design. It can also give values
 to generic parameters (see page 44-45).
- A CONFIGURATION is the most powerful of the three available configurations in VHDL. In VHDL'93 it may connect unconnected ports, ports that was not connected by a configuration specification (see page 49). That is called incremental binding.
- A CONFIGURATION connects all parts of a design and shall therefore be compiled as the last design unit.
- All three possible configurations are described on pages 48-50.

	PACKAGE BODY Pack IS END PACKAGE BODY Pack;	B1k:BLOCK BEGIN END BLOCK B1k;
ENTITY Ent IS BEGIN END ENTITY Ent;	ARCHITECTURE Arc OF Ent IS BEGIN END ARCHITECTURE Arc;	CONFIGURATION Conf OF Ent IS END CONFIGURATION Conf;
Proc:PROCESS() BEGIN END PROCESS Proc;	PROCEDURE P() IS BEGIN END PROCEDURE P;	FUNCTION F() RETURN TP IS BEGIN END FUNCTION F;

WAIT

Syntax LRM § 8.1

```
wait-statement ->
    [label ':'] wait [ on sensitivity-list ] [ until boolean-expression ]
    [for time-expression ] ';'

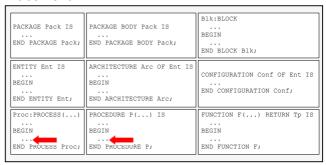
sensitivity-list -> signal-name { ',' signal-name }
```

Examples

```
SIGNAL s, p : BIT;
VARIABLE V : BIT;
VARIABLE t : TIME;
WAIT ON s;
                                 -- Wait for value changes on s
WAIT ON S UNTIL S = '1';
                                 -- Wait for a rising edge on s
-- Wait for a rising edge on s
WAIT UNTIL s = '1';
WATT:
                                 -- Never passed
WAIT UNTIL V:
                                 -- Never passed
WAIT UNTIL NOW = t;
                                 -- Never passed
WAIT ON s UNTIL p = '1' FOR t; -- Wait for value changes on s,
                                  -- then verify that p = '1', or
                                 -- wait at maximum the time t
                                   (timeout)
WAIT FOR 10 ns;
                                 -- Pass WAIT after 10 ns
                                 -- Pass WAIT after the time t
WAIT FOR t - NOW;
                                 -- minus current simulation time
```

Comments

- The WAIT statement has three conditional parts that may be combined: ON that
 detects value changes, UNTIL that verifies a logical expression and FOR that
 limits in time (timeout).
- · A WAIT statement may exist without a condition and will then never be passed.
- A variable does not have an event and does therefore not work in a WAIT ON statement. For the same reason expressions without signals do not work in a WAIT UNTIL statement. Such WAIT statements are suspended forever.
- Most synthesis tools accept just one WAIT statement for each process. The number is unlimited for simulation. See also page 40 (PROCESS).
- At simulation start every process executes until it reaches its first WAIT statement
- · A process shall have either a WAIT statement or a sensitivity list (see page 40).



IF

Syntax

```
LRM
§ 8.7
```

```
if-statement → [ if-label ':' ] if boolean-expression then
{ sequential-statement }
{ elsif boolean-expression then
{ sequential-statement } }
[ else { sequential-statement } ]
end if [ if-label ] ';'
```

Examples

```
PROCESS (reset, clk)
BEGIN
 IF reset = '1' THEN
 ELSIF clk'EVENT AND clk = '1' THEN
 END IF;
END PROCESS;
ANDgate: IF en = '1' THEN
 q <= d;
ELSE
 q <= '0';
END IF;
Latch: IF en = '1' THEN
 q <= d;
END IF:
IF a = Func(i*2#01001001#) THEN
END IF;
```

Comments

- All conditions in an IF statement must be of the type BOOLEAN.
- The syntax for the IF statement is quite odd ELSIF is spelled as one word without an intermediate 'E'. END IF is two separate words.

	PACKAGE BODY Pack IS END PACKAGE BODY Pack;	Blk:BLOCK BEGIN END BLOCK Blk;
ENTITY Ent IS BEGIN END ENTITY Ent;	ARCHITECTURE Arc OF Ent IS BEGIN END ARCHITECTURE Arc;	CONFIGURATION Conf OF Ent IS END CONFIGURATION Conf;
Proc:PROCESS() BEGIN END PROCESS Proc;	PROCEDURE P() IS BEGIN END PROCEDURE P;	FUNCTION F() RETURN TP IS BEGIN END FUNCTION F;

CASE

Syntax

```
LRM
§ 8.8
```

```
case-statement → [ case-label ':' ] case expression is

when choices '=>' { sequential-statement }

{ when choices '=>' { sequential-statement }
}

end case [ case-label ] ';'

choices → choice { | choice }

choice → simple-expression | discrete-range | element-name-identifier | others
```

Examples

```
PROCESS
BEGIN
  WAIT ON number;
  CASE number IS
    WHEN 0 => ... -- When "number" is 0
WHEN 2 | 5 | 243 => ... -- When "number" is 2, 5 or 243
WHEN 6 TO 18 => ... -- In the interval 6 to 18
                         => NULL; -- At all other values, do nothing
    WHEN OTHERS
  END CASE:
END PROCESS;
SIGNAL s : STD ULOGIC;
CASE s IS
  WHEN '0'
                 => ...
  WHEN '1'
                 => ...
  WHEN OTHERS => NULL; -- Must exist (or a compilation error
                             -- will occur)
END CASE;
```

Comments

- The CASE statement must specify all possible values of the expression. If not all possible values are covered, a compilation error will occur. OTHERS may be used to cover "all other values".
- The expression can be of an integer type, an enumerated type or a one-dimensional array with elements written as characters (for example STRING, BIT_VECTOR, STD_LOGIC_VECTOR etc.).
- Note that the types STD_LOGIC and STD_ULOGIC have more possible values than '0' and '1'. They must also be included in the CASE statement (see above).
- The reserved word NULL is useful in combination with OTHERS. Together they
 specify that "at all other values nothing shall happen".

	PACKAGE BODY Pack IS END PACKAGE BODY Pack;	Blk:BLOCK BEGIN END BLOCK Blk;
ENTITY Ent IS BEGIN END ENTITY Ent;	ARCHITECTURE Arc OF Ent IS BEGIN END ARCHITECTURE Arc;	CONFIGURATION Conf OF Ent IS END CONFIGURATION Conf;
Proc:PROCESS() BEGIN END PROCESS Proc;	PROCEDURE P() IS BEGIN END PROCEDURE P;	FUNCTION F() RETURN TP IS BEGIN END FUNCTION F;

LOOP, NEXT and EXIT

Syntax

LRM § 8.9-8.11

```
| loop-statement → [ loop-label ':'] [ while boolean-expression | for identifier in discrete-range ] | loop | sequential statement } end loop [ loop-label ] ';' exit=statement → [ label ':'] exit [ loop-label ] [ when boolean-expression ] ';' next-statement → [ label ':'] next [ loop-label ] [ when boolean-expression ] ';'
```

Examples

```
L1: LOOP
  L2: WHILE count < 15 LOOP
   NEXT L2 WHEN value = 12;
    count := count + Func(value);
  END LOOP L2;
END LOOP L1;
FOR i IN anArray'RANGE LOOP
 EXIT WHEN IS_X(anArray(i)); -- Exit if the array contains any
                                 'U', 'X', 'Z', 'W' or '-'
  IF anArray(i) = '1' THEN
   REPORT "There is a 1 at position " & INTEGER'IMAGE(i);
  END IF
END LOOP;
factorial := 1;
FOR j IN 1 TO n LOOP
 factorial := factorial*j;
END LOOP;
```

Comments

- FOR loops are in general synthesizable, but not WHILE loops.
- FOR loops loop according to a loop variable that shall be an integer or an enumerated type. The loop variable shall not be declared.
- WHILE loops loop as long as a BOOLEAN expression is true.
- EXIT jumps out of the loop and NEXT goes directly to the next iteration, not executing any code between NEXT and END LOOP.
- It is useful to name loops since it then is possible to specify what loop to exit or iterate using EXIT or NEXT.
- It is not possible to affect the length of the steps in a FOR loop.

	PACKAGE BODY Pack IS END PACKAGE BODY Pack;	Blk:BLOCK BEGIN END BLOCK Blk;
ENTITY Ent IS BEGIN END ENTITY Ent;	ARCHITECTURE Arc OF Ent IS BEGIN END ARCHITECTURE Arc;	CONFIGURATION CONF OF Ent IS END CONFIGURATION CONF;
Proc:PROCESS() BEGIN END PROCESS Proc;	PROCEDURE P() IS BEGIN END PROCEDURE P;	FUNCTION F() RETURN Tp IS BEGIN END FUNCTION F;

FUNCTION

Syntax

LRM § 2.1-2.3

Examples

```
-- declaration
FUNCTION AnyZeros (CONSTANT v : IN BIT VECTOR) RETURN BOOLEAN;

    implementation

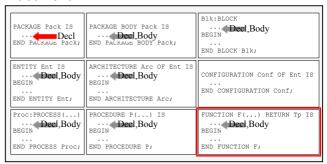
FUNCTION AnyZeros (CONSTANT v : IN BIT VECTOR) RETURN BOOLEAN IS
BEGIN
 FOR i IN V'RANGE LOOP
    IF v(i) = '1' THEN
      RETURN TRUE;
    END IF;
  END LOOP:
 RETURN FALSE;
END FUNCTION AnyZeros;

    Function call

q \leftarrow Func(p1 \Rightarrow v1, p2 \Rightarrow v2); -- Named association
q <= Func(v1, v2);
                                 -- Positional association
FUNCTION "AND" (...) RETURN ...; -- A userdefined operator
FUNCTION "+" (...) RETURN ...; -- Another one
-- Operator call
q <= a + b; -- The types of a, b and q determine which "+" to
             -- call (eventually a user-defined operator)
```

Comments

- A function returns a single value.
- The formal parameters of a function (the interface-list) are handled as constants with mode IN if nothing else is specified. Possible parameters are constants and signals, with mode IN, and files. Parameters with mode IN may only be read.
- · A function may not include any WAIT statements.
- · It is not permitted to declare signals in subprograms.
- Functions are either PURE or IMPURE. A PURE FUNCTION has no side-effects, i.e. it will always return the same value with the same input parameters, while an IMPURE FUNCTION may have side-effects (for example assigning signals, opening files etc.). If nothing else is said the function is considered PURE.



PROCEDURE

Syntax

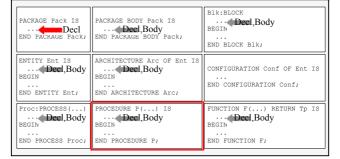
LRM § 2.1-2.3

Examples

```
- declaration
PROCEDURE AnyZeros (CONSTANT inArray : IN BIT_VECTOR;
                   VARIABLE result : OUT BOOLEAN);
 - implementation
PROCEDURE AnyZeros (CONSTANT inArray : IN BIT VECTOR;
                   VARIABLE result : OUT BOOLEAN) IS
BEGIN
 result := FALSE; -- default assignment
  FOR i IN inArray'RANGE LOOP
    IF inArray(i) = '1' THEN
      result := TRUE;
   END IF;
 END LOOP;
END PROCEDURE AnyZeros;
PROCEDURE Finish IS
BEGIN
 {\bf REPORT} "The simulation stopped at the time " & TIME'IMAGE(NOW);
END PROCEDURE Finish;
 - The procedure "Finish" is called when executing the line:
Finish:
```

Comments

- A procedure may contain a number of parameters that are read and/or modified at
 a procedure call. All parameters shall be declared in the procedure's interface-list
 and separated by a ';'. If nothing else is specified the parameters are handled as
 constants with the mode IN. It is possible to not have any parameters at all.
- The parameters can be constants, variables, signals or files. All parameters with
 the mode OUT or INOUT are as default variables, while parameters with the
 mode IN are constants. Constants do always have the mode IN, while variables
 and signals may be of the modes IN, INOUT or OUT. Files do not have a mode.
- · A procedure may contain WAIT statements.
- · It is not permitted to declare signals in subprograms.



RETURN

Syntax

LRM § 8.12

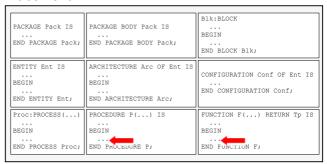
```
return-statement ->
    [label ':'] return [ expression ] ';'
```

Examples

```
FUNCTION AnyZeros (CONSTANT v : IN BIT_VECTOR) RETURN BOOLEAN IS
BEGIN
 FOR i IN v'RANGE LOOP
   IF v(i) = 1' THEN
     RETURN TRUE; -- Return the value TRUE
    END IF;
  END LOOP;
  RETURN FALSE; -- Return the value FALSE
END FUNCTION AnyZeros;
PROCEDURE AnyZeros (CONSTANT inArray : IN STD_LOGIC_VECTOR;
                   VARIABLE result : OUT BOOLEAN) IS
BEGIN
 IF IS_X(inArray) THEN
   REPORT "Unacceptable values";
    RETURN; -- Exit the procedure
 END IF:
  result := FALSE; -- default assignment
  FOR i IN inArray'RANGE LOOP
    IF inArray(i) = '1' THEN
     result := TRUE;
    END IF;
 END LOOP;
END PROCEDURE AnyZeros;
```

Comments

- RETURN is used to exit subprograms. A RETURN statement in a function must return a value, while a RETURN statement in a procedure must not have a value. A procedure returns all values via its formal parameters.
- A function should contain at least one RETURN statement (without RETURN the function is rather meaningless). It determines what value that will be returned from the function.
- A procedure may contain RETURN, but it is not necessary. If it is included it is
 used to exit the procedure. If no RETURN statement exists, the procedure will
 end after the final line has been executed. RETURN can not return any value in a
 procedure as it can in a function.



Variable assignment

Syntax

LRM § 8.5

```
variable-assignment-statement ->
[ label ':'] ( variable-name | variable-aggregate ) ':=' expression ';'
```

Examples

```
ARCHITECTURE Behave OF Design IS
  SHARED VARIABLE globalVariable : INTEGER;
REGIN
 PROCESS
    VARIABLE internal Variable : REAL;
 BEGIN
    sharedVariable := INTEGER(internalVariable);
 END PROCESS;
END ARCHITECTURE Behave;
PROCESS
 VARIABLE a : INTEGER := 0;
BEGIN
 First : a := a + 2;
 Second: a := a + 4;
 Final : a := a - 3;
  REPORT INTEGER' IMAGE(a); -- a = 3 (i.e. 0 + 2 + 4 - 3)
END PROCESS;
v := (2\#0100\# + Func(2.0*3.14))/ABS(x);
```

Comments

- A variable assignment immediately updates the value of the variable. The assignment uses ':=', i.e. the same sign used for default assignments.
- A variable can be assigned the value of a signal and vice versa.
- The expression in the variable assignment may be arbitrarily complex and for example include subprogram calls.
- A variable may be assigned using a so called "aggregate" (see page 13).
- Shared variables (see page 17) are global just as signals. Since variables do
 not have the possibility to handle concurrent assignments from more than one
 process, as signals can, shared variables are to be avoided.

	PACKAGE BODY Pack IS END PACKAGE BODY Pack;	Blk:BLOCK BEGIN END BLOCK Blk;
ENTITY Ent IS BEGIN END ENTITY Ent;	ARCHITECTURE Arc OF Ent IS BEGIN END ARCHITECTURE Arc;	CONFIGURATION Conf OF Ent IS END CONFIGURATION Conf;
Proc:PROCESS() BEGIN END PROCESS Proc;	PROCEDURE P() IS BEGIN END PROCEDURE P;	FUNCTION F() RETURN Tp IS BEGIN END FUNCTION F;

Signal assignment

Syntax

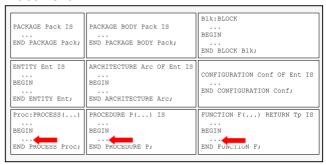
LRM § 8.4. 9.5

Examples

```
ENTITY Design IS
 PORT(externalSignal : OUT INTEGER);
END ENTITY Design;
ARCHITECTURE Behave OF Design IS
 SIGNAL internal Signal : REAL;
BEGIN
 externalSignal <= INTEGER(internalSignal);
END ARCHITECTURE Behave;
SIGNAL a : INTEGER := 0;
PROCESS
BEGIN
 First : a \le a + 2;
 Second: a \le a + 4;
 Final : a <= a - 3;
 WAIT FOR 5 ns;
 REPORT INTEGER'IMAGE(a); -- a = -3 (i.e. 0 - 3)
END PROCESS;
s1 <= INERTIAL s AFTER 10 ns;
s2 <= REJECT 5 ns INERTIAL s AFTER 10 ns;
s3 <= TRANSPORT s AFTER 10 ns;
```

Comments

- A signal assignment immediately puts a value in the signal's queue, but its driver is updated first when the process finishes its execution, i.e. when it reaches a WAIT statement or reaches the final line (when using a sensitivity list).
- · A signal can be assigned the value of a variable and vice versa.
- The expression in the signal assignment may be arbitrarily complex and for example include subprogram calls.
- A signal may be assigned using a so called "aggregate" (see page 13).
- · TRANSPORT generates a transmission delay of a signal.
- · REJECT specifies pulses to be filtered (must be combined with INERTIAL).
- · INERTIAL both filters and delays. INERTIAL is default.



ASSERT/REPORT

Syntax

LRM

§ 8.2, 8.3, 9.4

```
assertion-statement →

[ label ':' ] [ postponed' ] assert boolean-expression [ report string-expression ]

[ severity expression ] ';'
report-statement →

[ label ':' ] report string-expression [ severity expression ] ';'

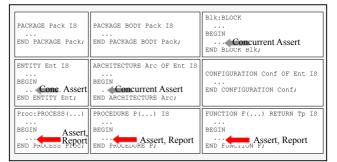
* postponed is only allowed in a concurrent ASSERT statement
```

Examples

```
ARCHITECTURE Behave OF Design IS
BEGIN
  PROCESS
  BEGIN
    ASSERT a = b -- sequential ASSERT
      \ensuremath{\mathbf{REPORT}} "a and b are not equal"
         SEVERITY WARNING;
    WAIT ON a, b;
    REPORT "WAIT was just passed";
  END PROCESS;
  Control: POSTPONED ASSERT a = 12 -- Concurrent ASSERT
REPORT "a is not 12";
END ARCHITECTURE Behave;
ASSERT ..
  REPORT "xxx";
                         -- The ASSERT statement ends here!
REPORT "yyy"
                         -- This line has nothing to do with ASSERT
                         -- NOTE defined for the second REPORT
  SEVERITY NOTE;
ASSERT .
  REPORT "xxx" & "yyy" -- Here both REPORT and SEVERITY are
SEVERITY FAILURE; -- connected to ASSERT
```

Comments

- With an ASSERT statement a logical expression is claimed to be true. If it is false
 the rest of the ASSERT statement is executed.
- SEVERITY has four possible values NOTE, WARNING, ERROR and FAI-LURE (see page 7, 57). In most simulators it is possible to set at which severity level the simulation shall be stopped.
- ASSERT has default severity level ERROR while REPORT has NOTE.
- ASSERT is both a sequential and a concurrent statement while REPORT only is a sequential statement. However a concurrent ASSERT statement may include a REPORT statement.
- · When using REPORT it is convenient to concatanate text strings using '&'.



Subprogram call

Syntax

LRM

§ 7.3.3, 8.6, 9.3

```
function-call → function-name [ '(' parameter-association-list ')']

procedure-call-statement →

[label ':'] [ postponed'] procedure-name [ '(' parameter-association-list ')'] ';'

parameter-association-list →

[ formal-part '=>'] actual-part { ',' [ formal-part '=>'] actual-part }

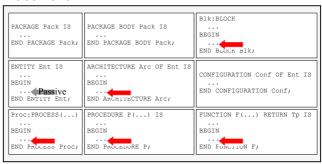
*postponed is only allowed in a concurrent function call
```

Examples

```
ENTITY Design IS
 PORT (d, clk : IN BIT; q : OUT BIT);
BEGIN
 PeriodCheck(clk); -- Passive procedure call
END ENTITY Design;
-- This function adds two arrays
FUNCTION "+" (a, b : MyArray) RETURN MyArray IS
REGIN
END FUNCTION "+";
SIGNAL d1, d2, sum : MyArray;
sum <= d1 + d2; -- The function "+" above is called
PROCEDURE Add(a,b : IN MyArray; sum : OUT MyArray) IS
BEGIN
END PROCEDURE Add;
Add (data1, data2, sum);
                                          -- Positional assoc.
Add(sum => sum, a => data1, b => data2); -- Named association
outData := AFunction(x,3.14,16#02AE#);
```

Comments

- A subprogram is called via its name. In the call actual parameters can be associated with the formal parameters declared in the subprogram. The association may be done named or positional where positional is preferable since the order of the parameters then does not impact the association.
- It is permitted to call passive procedures in the statement part of an *entity*, i.e. procedures not assigning signals. This is useful when verifying timing for ports declared in the *entity*. If any parameter is modified, the procedure is called.
- A concurrent subprogram call is executed whenever an input signal gets a new value.



PROCESS

Syntax

```
LRM
§ 9.2
```

```
process-statement →

[process-label ':'] [postponed] process [ '(' sensitivity-list ')'] [ is ]

{ subprogram-decl. | subprogram-body | type-decl. | subtype-decl. |

constant-decl. | variable-decl. | file-decl. | alias-decl. | attribute-decl. |

attribute-spec. | use-clause | group-template-decl. | group-decl. }

begin

{ sequential-statement }

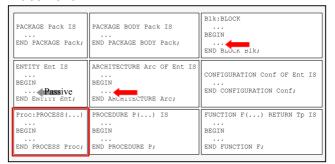
end [ postponed ] process [ process-label ] ';'
```

Examples

```
ARCHITECTURE Behave OF Design IS
REGIN
  FlipFlop: PROCESS (reset, clk)
 REGIN
    IF reset = '1' THEN
      q <= '0';
    ELSIF clk'EVENT AND clk = '1' THEN
      q <= d;
    END IF:
  END PROCESS FlipFlop;
END ARCHITECTURE Behave;
 - These two processes are equivalent
PROCESS
                                   PROCESS (s)
REGIN
                                   BEGIN
 REPORT "s has a new value";
                                     REPORT "s has a new value";
 WAIT ON s;
                                  END PROCESS;
END PROCESS:
```

Comments

- A PROCESS holds a number of sequential statements and executes parallell towards its environment.
- A PROCESS functions as an eternal loop. It must include at least one WAIT statement (see page 29) or a sensitivity list that specifies when the PROCESS shall execute its sequential statements.
- A sensitivity list is equivalent to a WAIT ON statement placed as the final line in the PROCESS. All sequential statements will execute once at simulation startup in such processes and after that the processes will suspend.
- POSTPONED defines that the PROCESS shall be executed as the final delta at a specific occasion.





WHEN

Syntax

LRM § 9.5.1

```
conditional-signal-assignment-statement →

[label ':'] [ postponed ] ( signal-name | signal-aggregate ) '<='

[ guarded ] [ delay-mechanism ]

{ waveform when boolean-expression else }

waveform [ when boolean-expression ] ';'

delay-mechanism → transport | [ reject time-expression ] inertial

waveform → waveform-element { ',' waveform-element } | unaffected

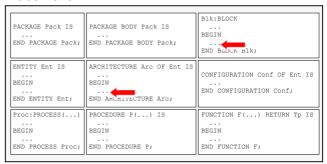
waveform-element → ( value-expression | null ) [ after time-expression ]
```

Examples

```
-- This architecture contains three processes. "One" is an
 -- ordinary process including a sequential signal assignment
  while "Two" and "Three" are concurrent signal assignments
ARCHITECTURE Behave OF Design IS
BEGIN
 One: PROCESS (data)
 BEGIN
   outputSignal <= data;
 END PROCESS One;
 Two: s <= '1'
                       WHEN sel = "00" ELSE
            UNAFFECTED WHEN sel = "11" ELSE
 Three: s2 <= REJECT 3 ns INERTIAL d AFTER 5 ns;
END ARCHITECTURE Behave;
MyBlock: BLOCK (en = '1')
BEGIN
  Latch: q <= GUARDED d;
END BLOCK MyBlock;
```

Comments

- A concurrent signal assignment is placed directly in an ARCHITECTURE or in a BLOCK without using a PROCESS.
- A concurrent signal assignment may preferably be named with a label. This label simplifies simulation since the assignment then can be identified just as a named PROCESS.
- · The WHEN statement is the concurrent equivalent to the sequential IF statement.
- UNAFFECTED is new to VHDL'93 and may be used to specify that a signal shall be left unaffected at a specific occasion, i.e. to keep its previous value.



SELECT

Syntax

LRM § 9.5.2

```
selected-signal-assignment-statement ->
[ label ':' ] [ postponed ] with expression select

( signal-name | signal-aggregate ) '<=' [ guarded ] [ delay-mechanism ]

{ waveform when choices ',' } waveform when choice { | choice } ';'

delay-mechanism -> transport | [ reject time-expression ] inertial

waveform -> waveform-element { ',' waveform-element } | unaffected

waveform-element -> ( value-expression | null ) [ after time-expression ]

choice -> simple-expression | discrete-range | element-name-identifier | others
```

Examples

```
ARCHITECTURE Behave OF Design IS
BEGIN
Choose: WITH sel SELECT
s <= '1' WHEN "00",
UNAFFECTED WHEN "11",
'0' WHEN OTHERS;
END ARCHITECTURE Behave;
```

Comments

- A concurrent signal assignment is placed directly in an ARCHITECTURE or in a BLOCK without using a PROCESS.
- A concurrent signal assignment may preferably be named with a label. This label simplifies simulation since the assignment then can be identified just as a named PROCESS.
- The SELECT statement is the concurrent equivalent to the sequential CASE statement
- UNAFFECTED is new to VHDL'93 and may be used to specify that a signal shall be left unaffected at a specific occasion, i.e. to keep its previous value.

	PACKAGE BODY Pack IS END PACKAGE BODY Pack;	Blk:BLOCK BEGIN END BLOCK blk;
ENTITY Ent IS BEGIN END ENTITY Ent;	ARCHITECTURE Arc OF Ent IS BEGIN END ARCHITECTURE Arc;	CONFIGURATION CONF OF Ent IS END CONFIGURATION CONF;
Proc:PROCESS() BEGIN END PROCESS Proc;	PROCEDURE P() IS BEGIN END PROCEDURE P;	FUNCTION F() RETURN Tp IS BEGIN END FUNCTION F;



BLOCK

Syntax LRM § 9.1

```
block-statement 

block-label ':' block [ '(' guard-expression ')'] [ is ]

[ generic-clause [ generic-map ';'] ]

[ port-clause [ port-map ';'] ]

begin

{ subprog.-decl. | subprog.-body | type/subtype-decl. | constant-decl. | signal-decl. | shared-variable-decl. | file-decl. | alias-decl. | component-decl. | attribute-decl. | attribute-spec. | config.-spec. | use-clause | group-temp.-decl. | group-decl. }

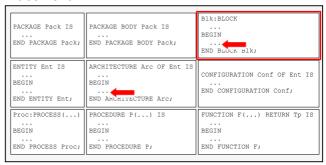
end block [ block-label ] ';'
```

Examples

```
ARCHITECTURE Behave OF Design IS
  CONSTANT holdTime : TIME := 5 ns;
  SIGNAL
          output
                    : BIT;
BEGIN
 Block1: BLOCK(en = '1')
   GENERIC t : TIME;
    GENERIC MAP(t => holdTime);
    PORT(d : IN BIT;
        q : OUT BIT);
    PORT MAP (d => data, q => output);
  REGIN
   q <= GUARDED d AFTER t;
  END BLOCK Block1;
END ARCHITECTURE Behave;
OneBlock: BLOCK(en = '1')
  Latch: q <= GUARDED d;
END BLOCK OneBlock;
```

Comments

- A BLOCK has two purposes to introduce a structure in the design and to be used in combination with *guarded signals*.
- A BLOCK may have generic parameters and a port list just as an ENTITY. It is
 however most common to use components when structure is desired. The advantage of components is that it exists powerful methods to instantiate and configure
 them (see pages 28, 46-50).
- Declarations inside a BLOCK are local in the BLOCK.
- A BLOCK must be named by a label.
- It is possible to declare a BLOCK inside another BLOCK. That creates a structure in the design.



GENERIC/GENERIC MAP

Syntax

LRM

§1.1.1.1,5.2.1.2

Examples

```
ENTITY LargeFlipFlop IS
 GENERIC (n : NATURAL;
        t : TIME);
 PORT (d
          : IN BIT VECTOR (n DOWNTO 0);
     clk : IN BIT;
          : OUT BIT VECTOR (n DOWNTO 0));
END ENTITY LargeFlipFlop;
ARCHITECTURE Behave OF LargeFlipFlop IS
REGIN
 q <= d AFTER t WHEN (clk'EVENT AND clk = '1');
END ARCHITECTURE Behave;
-- The entity LargeFlipFlop may be instantiated like this:
ARCHITECTURE Behave OF Top IS
 COMPONENT LargeFlipFlop IS
       -- The same declaration as in the ENTITY LargeFlipFlop
 END COMPONENT LargeFlipFlop;
BEGIN
  C1 : LargeFlipFlop GENERIC MAP(n => 5, t => 12 ns)
                     PORT MAP(d => dtop, clk => clk, q => qtop);
END ARCHITECTURE Behave;
```

Comments

- Generic parameters are used to create parameterizable units in a design. First
 when the unit shall be used the parameters must get values. Generic parameters
 may for example be used to define bus widths and delay parameters. Delay parameters are useful when generating backannotated VHDL files from synthesis and
 Place&Route tools.
- Generic parameters are "connected" to values using a GENERIC MAP that functions just as a PORT MAP does for signals (see page 47).
- A GENERIC MAP may exist in a component instantiation (see page 47), in a
 configuration declaration (see page 50) or in a BLOCK instantiation (see page
 43). The generic parameters may also be assigned values in the simulator or in
 the synthesis tool.

	PACKAGE BODY Pack IS END PACKAGE BODY Pack;	BIK:BLOCK BEGIN GENERIC & GENERIC MAP END BLOCK BIK;
ENTITY Ent IS GENERIC BEGIN END ENTITY Ent;	ARCHITECTURE Arc OF Ent IS BEGIN END ARCHITECTURE Arc;	CONFIGURATION CONF OF Ent IS END CONFIGURATION CONF;
Proc:PROCESS() BEGIN END PROCESS Proc;	PROCEDURE P() IS BEGIN END PROCEDURE P;	FUNCTION F() RETURN Tp IS BEGIN END FUNCTION F;

GENERATE

Syntax

LRM § 9.7

```
generate-statement -> generate-label ':'

( for identifier in discrete-range | if boolean-expression ) generate

[ subprog.-decl. | subprog.-body | type/subtype-decl. | constant-decl. | signal-decl. | shared-variable-decl. | file-decl. | alias-decl. | component-decl. | attribute-decl. | attribute-spec. | config.-spec. | use-clause | group-temp.-decl. | group-decl. | begin ]

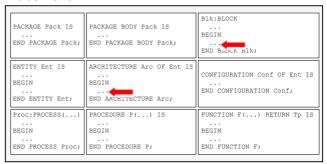
{ concurrent-statement }
end generate [ generate-label ] ';'
```

Examples

```
ENTITY FlipFlop IS
  PORT (d, clk : IN BIT;
              : OUT BIT);
END ENTITY FlipFlop;
ARCHITECTURE Behave OF FlipFlop IS ...
ENTITY LargeFlipFlop IS
  GENERIC (n : NATURAL);
  PORT (d
           : IN BIT_VECTOR(n DOWNTO 0);
       clk : IN
                 BIT;
       q
           : OUT BIT VECTOR (n DOWNTO 0));
END ENTITY LargeFlipFlop;
ARCHITECTURE Behave OF LargeFlipFlop IS
 COMPONENT FlipFlop ... -- The same as in the ENTITY FlipFlop
BEGIN
  Build: FOR i IN d'RANGE GENERATE
    FOR ALL : FlipFlop USE ENTITY WORK.FlipFlop (Behave);
  BEGIN
   D: FlipFlop PORT MAP(d \Rightarrow d(i), clk \Rightarrow clk, q \Rightarrow q(i));
  END GENERATE Build;
END ARCHITECTURE Behave;
```

Comments

- GENERATE is used to conditionally create processes. FOR GENERATE is
 useful to create a number of copies of processes, while IF GENERATE is useful
 when a part of a design shall be excluded during simulation or synthesis.
- The GENERATE statement must be named by a label. This label can be addressed in a configuration declaration (see page 50) and by that parameters in the GENERATE statement can be modified or completed.
- A configuration specification (see page 49) for components in a GENERATE statement must be placed between GENERATE and BEGIN in the statement (see example above).



Component declaration

Syntax

LRM § 4.5

```
component-declaration →

component identifier [ is ]

[ local-generic-clause ]

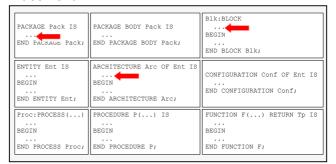
[ local-port-clause ]

end component [ component-name-identifier ] ';'
```

Examples

Comments

- A COMPONENT declares an "empty socket". There is no specification of what
 to be placed in the "socket", i.e. which ENTITY and ARCHITECTURE that will
 specify the functionality.
- Components are used to achieve a structure in a design. The number of hierarchical levels is unlimited.
- Three steps are performed when working with components component declaration, component instantiation (see page 47) and component configuration (see page 48-50). In VHDL'93 it is also possible to directly instantiate an ENTITY, a so called *direct instantiation* (see page 47). This method is however not recommended since the design then loses in reuseability.



Component instantiation

Syntax ERM § 9.6

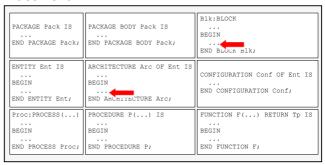
Examples

```
ENTITY LargeFlipFlop IS
  GENERIC(n : NATURAL; t : TIME);
       (d : IN BIT_VECTOR(n DOWNTO 0);
clk : IN BIT;
  PORT (d
             : OUT BIT VECTOR(n DOWNTO 0));
END ENTITY LargeFlipFlop;
ARCHITECTURE Behave OF LargeFlipFlop IS
ENTITY Design IS ...
ARCHITECTURE Behave OF Design IS
  COMPONENT LargeFlipFlop IS
    GENERIC (n : NATURAL; t : TIME);
             : IN BIT VECTOR (n DOWNTO 0);
          clk : IN BIT;
              : OUT BIT_VECTOR(n DOWNTO 0));
          q
  END COMPONENT LargeFlipFlop;
  C1 : LargeFlipFlop GENERIC MAP(t => 12 ns, n => 5)
  PORT MAP(clk \Rightarrow clk, q \Rightarrow q1, d \Rightarrow d1);
C2 : ENTITY WORK.LargeFlipFlop(Behave) GENERIC MAP(7, 15 ns)
                                               PORT MAP(d2, clk, q2);
END ARCHITECTURE Behave;
```

Comments

- A component instantiation specifies how a component, an entity (only in VHDL'93) or a configuration declaration is connected in a design. It is not recommended to directly instantiate entities, so called direct instantiation ('C2' above), since the design then loses in reuseability.
- Components are used to achieve structure in a design. The number of hierarchical levels is unlimited. Three steps are performed when working with components

 component declaration (see page 46), component instantiation and component configuration (see page 48-50).
- A PORT MAP connects ports from inside and out (component port => signal).
- · Using OPEN specifies that a port shall be unconnected.



Default configuration

Syntax

LRM § 5.2.2

```
(No syntax)
```

Examples

```
ENTITY LargeFlipFlop IS
  GENERIC (n : NATURAL; t : TIME);
          : IN BIT VECTOR (n DOWNTO 0);
      clk : IN BIT;
       q
           : OUT BIT VECTOR (n DOWNTO 0));
END ENTITY LargeFlipFlop;
ARCHITECTURE Behave OF LargeFlipFlop IS ...
ENTITY Design IS ...
ARCHITECTURE Behave OF Design IS
 COMPONENT LargeFlipFlop IS
    GENERIC (n : NATURAL; t : TIME);
            : IN BIT VECTOR (n DOWNTO 0);
   PORT (d
         clk : IN BIT;
             : OUT BIT_VECTOR(n DOWNTO 0));
 END COMPONENT LargeFlipFlop;
BEGIN
  -- The entity "LargeFlipFlop" and its architecture "behave"
   - will be used for C1 provided that they are compiled to WORK
  -- and that "behave" is the last compiled architecture
  C1 : LargeFlipFlop GENERIC MAP(n \Rightarrow 5, t \Rightarrow 12 ns)
                      PORT MAP(d1, clk, q1);
END ARCHITECTURE Behave;
```

Comments

- A default configuration implies that the simulator or the synthesis tool automatically will connect a COMPONENT to an ENTITY. This requires that they match perfectly regarding names, port names, port types, port directions, generic parameter names and generic parameter types. The latest compiled ARCHITECTURE for the ENTITY is used.
- The advantages by using a default configuration are that it is simple (no explicit configuration is needed) and that it may be overwritten by a configuration declaration (see page 50). The disadvantages are that the ENTITY must match the COMPONENT perfectly and that it is not unambigously defined which ARCHITECTURE that will be used.

	PACKAGE BODY Pack IS END PACKAGE BODY Pack;	Blk:BLOCK BEGIN END BLOCK Blk;
ENTITY Ent IS BEGIN END ENTITY Ent;	ARCHITECTURE Arc OF Ent IS BEGIN END ARCHITECTURE Arc;	CONFIGURATION CONF OF Ent IS END CONFIGURATION CONF;
Proc:PROCESS() BEGIN END PROCESS Proc;	PROCEDURE P() IS BEGIN END PROCEDURE P;	FUNCTION F() RETURN Tp IS BEGIN END FUNCTION F;



Configuration specification

Syntax

LRM § 5.2

```
configuration-specification →

for ( instantiation-label { `,' instantiation-label } | others | all ) `:'

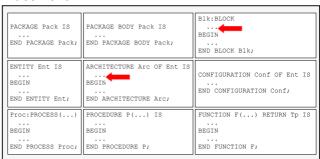
component-name [ use entity-aspect ] [ generic-map ] [ port-map ] `;'
```

Examples

```
ENTITY LargeFlipFlop IS
  GENERIC (t : TIME; n : NATURAL);
  PORT (clk
                : IN BIT;
                : IN BIT VECTOR (n DOWNTO 0);
       d
       q, qinv : OUT BIT VECTOR (n DOWNTO 0));
END ENTITY LargeFlipFlop;
ARCHITECTURE Behave OF LargeFlipFlop IS ...
ENTITY Design IS ...
ARCHITECTURE Behave OF Design IS
  COMPONENT LargeFlipFlop IS
    GENERIC (n : NATURAL; t : TIME);
    PORT(d : IN BIT VECTOR(n DOWNTO 0);
         clk : IN BIT;
             : OUT BIT_VECTOR(n DOWNTO 0));
  END COMPONENT LargeFlipFlop;
  FOR C1 : LargeFlipFlop USE ENTITY WORK.LargeFlipFlop (Behave)
   GENERIC MAP(12 ns, 5) PORT MAP(clk, d, q, OPEN);
  FOR OTHERS : LargeFlipFlop USE ...
 C1 : LargeFlipFlop GENERIC MAP(n => 5, t => 12 ns) PORT MAP C2 : LargeFlipFlop GENERIC MAP(...) PORT MAP (...);
END ARCHITECTURE Behave;
```

Comments

- A configuration specification connects a specific ENTITY and a specific ARCHI-TECTURE to a COMPONENT. The configuration specifies in detail how the connection shall be and that excludes the demand of perfect match. OPEN in a PORT MAP defines that a port shall be unconnected.
- The configuration specification is the mid alternative among the three possible configurations (see pages 48, 50). It is more powerful than a default configuration since the demand of perfect match between the ENTITY and the COMPONENT is excluded. It may on the other hand not configure sub blocks in the design hierarchy as a configuration declaration can. Modifications imply a recompilation of the whole architecture and since the component is locked to a specific entity/architecture pair it is not well suited for multilevel simulations.



Configuration declaration

Syntax

LRM § 1.3

```
configuration-declaration -> configuration identifier of entity-name is
{ use-clause | attribute-specification | group-declaration }
    block-configuration
end [ configuration ] [ configuration-name-identifier ] ';'

block-configuration -> for ( architecture-name | block-statement-label |
    generate-statement-label [ '(' index-specification ')' ] ) { use-clause }
    { block-configuration | component-configuration }
    end for ';'
```

Examples

```
CONFIGURATION MyConfiguration OF Design IS
FOR Behave
FOR C1 : LargeFlipFlop USE ENTITY WORK.LargeFlipFlop(Behave)
GENERIC MAP(7, 20 ns) FORT MAP(...);
END FOR;
FOR OTHERS : LargeFlipFlop USE ENTITY
WORK.LargeFlipFlop(Structure);
FOR Structure
FOR A1 : SubComponent USE ENTITY ...
END FOR;
FOR OTHERS : SubComponent USE CONFIGURATION ...
END FOR;
END CONFIGURATION MyConfiguration;
```

Comments

- A configuration declaration is a separate design unit in VHDL (see page 28).
 It is the most powerful of the three possible configurations (see pages 48-49) and it can modify previously performed configuration specifications (so called incremental binding).
- The *configuration declaration* is well suited for multilevel simulations since it can configure sub blocks in a design. It can also modify generic parameters, e.g. timing parameters, and also configure GENERATE statements (see page 45).
- The disadvantage with the configuration declaration is its complicated syntax.
- A configuration declaration connects an entire design and is therefore compiled as the final design unit.

	PACKAGE BODY Pack IS END PACKAGE BODY Pack;	Blk:BLOCK BEGIN END BLOCK Blk;
ENTITY Ent IS BEGIN END ENTITY Ent;	ARCHITECTURE Arc OF Ent IS BEGIN END ARCHITECTURE Arc;	CONFIGURATION Conf OF Ent IS END CONFIGURATION Conf;
Proc:PROCESS() BEGIN END PROCESS Proc;	PROCEDURE P() IS BEGIN END PROCEDURE P;	FUNCTION F() RETURN TP IS BEGIN END FUNCTION F;



Predefined attributes

LRM § 14.1

Attributes on types

T'BASE

Gives the base type of a type T. Can only be used as a prefix for another attribute.

Example:

SUBTYPE MyNat IS NATURAL RANGE 5 TO 15; MyNat'BASE'LOW = -2 147 483 647 (=INTEGER'LOW)

T'LEFT

Gives the leftmost value in the type T.

Example:

TYPE State IS (reset, start, count); State'LEFT = reset

T'RIGHT

Gives the rightmost value in the type T.

TYPE State IS (reset, start, count); State'RIGHT = count

T'HIGH

Gives the largest value in the type T.

Example:

SUBTYPE MyNat IS NATURAL RANGE 5 TO 15; MyNat'HIGH = 15

T'LOW

Gives the smallest value in the type T.

SUBTYPE MyNat IS NATURAL RANGE 5 TO 15; MyNat'LOW = 5

T'ASCENDING Returns a value of the type BOOLEAN that is TRUE if the type T is defined with an ascending range.

Example:

SUBTYPE MyNat IS NATURAL RANGE 5 TO 15; MyNat'ASCENDING = TRUE

T'IMAGE(X)*

Converts the value X, that is of the type or subtype T, to a text string. T must be a scalar type, i.e. an integer, a real, a physical type or an enumerated type.

Example:

TYPE State IS (reset, start, count); State'IMAGE(start) = "start"

T'VALUE(X)*

Converts the text string X to a value of the type T. T must be a scalar type, i.e. an integer, a real, a physical type or an enumerated type.

Example:

TYPE State IS (reset, start, count); State'VALUE("reset") = reset

T'POS(X)

Returns the position number of X within the type T.

Example:

TYPE State IS (reset, start, count); State'POS(start) = 1

^{*} New to VHDL'93

T'VAL(X) Returns the value on position X in the type T.

Example:

```
TYPE State IS (reset, start, count);
State'VAL(0) = reset
```

T'SUCC(X) Returns the value, of the type T, with the position number one greater than that of the parameter X.

Example:

```
TYPE MyInteger IS RANGE 5 DOWNTO -5;
MyInteger'SUCC(0) = 1
```

T'PRED(X) Returns the value, of the type T, with the position number one less than that of the parameter X.

Example:

```
TYPE MyInteger IS RANGE 5 DOWNTO -5;
MyInteger'PRED(0) = -1
```

T'LEFTOF(X) Returns the value to the left of the value X in the range of the type T.

```
Example:
```

```
TYPE MyInteger IS RANGE 5 DOWNTO -5;
MyInteger'LEFTOF(0) = 1
```

T'RIGHTOF(X) Returns the value to the right of the value X in the range of the type T.

Example:

```
TYPE MyInteger IS RANGE 5 DOWNTO -5;
MyInteger'RIGHTOF(0) = -1
```

Attributes on arrays

(All attributes on arrays are valid for both types and objects)

A'HIGH[(N)] Returns the numerical largest index limit in the array A for its index range N. N may be omitted and its default value is 1.

Example:

```
TYPE M IS ARRAY (0 TO 3, 2 DOWNTO 1) OF BIT;
VARIABLE matrix : M;
M'HIGH = 3
matrix'HIGH(2) = 2
```

A'LOW[(N)] Returns the numerical smallest index limit in the array A for its index range N. N may be omitted and its default value is 1.

Example:

```
TYPE M IS ARRAY (0 TO 3, 2 DOWNTO 1) OF BIT; VARIABLE matrix : M; M'LOW = 0 matrix'LOW(2) = 1
```

A'LEFT[(N)] Returns the left index limit for the array A for its index range N. N may be omitted and its default value is 1.

Example:

```
TYPE M IS ARRAY (0 TO 3, 2 DOWNTO 1) OF BIT;
VARIABLE matrix : M;
M'LEFT = 0
matrix'LEFT(2) = 2
```

A'RIGHT[(N)] Returns the right index limit for the array A for its index range N. N may be omitted and its default value is 1.

Example:

```
TYPE M IS ARRAY (0 TO 3, 2 DOWNTO 1) OF BIT;

VARIABLE matrix : M;

M'RIGHT = 3

matrix'RIGHT(2) = 1
```

A'RANGE[(N)] Returns the index range as a RANGE for the array A for its index range N. N may be omitted and its default value is 1.

Example:

```
TYPE M IS ARRAY (0 TO 3, 2 DOWNTO 1) OF BIT;
VARIABLE matrix: M;
M'RANGE = 0 TO 3
matrix'RANGE(2) = 2 DOWNTO 1
```

A'REVERSE RANGE[(N)]

Returns the index range as a RANGE, but with opposite direction, for the array A for its index range N. N may be omitted and its default value is 1.

Example:

```
TYPE M IS ARRAY (0 TO 3, 2 DOWNTO 1) OF BIT;
VARIABLE matrix : M;
M'REVERSE_RANGE = 3 DOWNTO 0
matrix'REVERSE RANGE(2) = 1 TO 2
```

A'LENGTH[(N)]

Returns the number of elements in the array A for its index range N. N may be omitted and its default value is

Example:

```
TYPE M IS ARRAY (0 TO 3, 2 DOWNTO 1) OF BIT;

VARIABLE matrix: M;

M'LENGTH = 4

matrix'LENGTH(2) = 2
```

A'ASCENDING[(N)]*

Returns a value of the type BOOLEAN that is TRUE if the index range N for array A is ascending. N may be omitted and its default value is 1.

Example:

```
TYPE M IS ARRAY (0 TO 3, 2 DOWNTO 1) OF BIT;
VARIABLE matrix : M;
M'ASCENDING = TRUE
matrix'ASCENDING(2) = FALSE
```

Attributes on signals

S'STABLE[(T)] Creates a new signal of the type BOOLEAN that returns TRUE as long as the signal S does not change its value.

The signal gets FALSE when S changes value and is FALSE during the time T. T may be omitted and its default value i 0 ns.

S'DELAYED[(T)]

^{*} New to VHDL'93



S'QUIET[(T)]

Functions exactly as 'STABLE but reacts on all updates of S from its driver queue, also when the signal is assigned the value it already has.

S'DELAYED[(T)]

Creates a copy of the signal S delayed the time T. T may be omitted and its default value is 0 ns, i.e. exactly S.

S'TRANSACTION

Creates a new signal of the type BIT that changes value every time the signal S gets a new value from its driver queue, i.e. also when it gets the same value as it already has. The initial value of the new signal is not specified.

S'

A function of the type BOOLEAN that returns TRUE during exactly one delta cycle when the signal S gets a new value

S'ACTIVE

Functions exactly as 'EVENT but reacts on all updates of S from its driver queue, also when the signal is assigned the value it already has.

S'LAST EVENT

A function of the type TIME returning the time since the last change of the value of the signal S.

S'LAST ACTIVE

A function of the type TIME returning the time since the the last update of the signal S.

S'LAST VALUE

A function of the same base type as S returning the value the signal S had before its last value change.

S'DRIVING*

A function of the type BOOLEAN returning TRUE if the driver for the signal S is on.

S'DRIVING VALUE*

A function of the same type as S returning the current value in the driver for the signal S in the current process.

Attributes on named entities

E'SIMPLE NAME*

Returns the name, in a text string with lower-case letters, of a named entity.



^{*} New to VHDL'93

E'INSTANCE_NAME*

Returns the hierarchical path including instances higher in the hierarchy, in a string with lower-case letters, to a named entity.

E'PATH_NAME*

Returns the hierarchical path not including instances higher in the hierarchy, in a string with lower-case letters, to a named entity.

```
Example:
    ENTITY E IS
    ...
    END ENTITY E;

ARCHITECTURE A OF E IS
BEGIN
    P: PROCESS (clock)
    VARIABLE inVar: NATURAL RANGE 0 TO 255;
BEGIN
    ...
    -- inVar'SIMPLE_NAME = "invar"
    -- inVar'INSTANCE_NAME = ":e(a):p:invar"
    -- inVar'PATH_NAME = ":e:p:invar"
    END PROCESS P;
END ARCHITECTURE A;
```

^{*} New to VHDL'93



IEEE

There is a number of predefined packages in VHDL. All packages standardized by IEEE are described in this chapter. Further there are a number of non standardized packages, e.g. STD_LOGIC_ARITH, STD_LOGIC_SIGNED, NUMERIC_SIGNED, STD_LOGIC_UNSIGNED and NUMERIC_UNSIGNED, but they are not recommended to be used since they are not standardized and may differ between different tools.

VHDL standards

VHDL is built upon a number of standards from IEEE:

• IEEE std 1076-1987	The first standard of VHDL. It is commonly ab-
	breviated VHDL'87.

• IEEE std 1076-1993	The second standard of VHDL. It is commonly
	abbreviated VHDL'93.

• IEEE std 1164-1993	The package STD_LOGIC_1164 (see page 58).
	Includes hardware related types and conversion
	functions

• IEEE std 1076a	Intended to improve the usage of shared variables
	(see page 17).

 IEEE std 1076.1 	VHDL-AMS,	analog extensions.
-------------------------------------	-----------	--------------------

• IEEE std 1076.2	Mathematical packages (see page 63-65). Divided
	into one package for real numbers and one for
	imaginary numbers.

• IEEE std 1076.3	Describes for example types and operators intended
	for synthesis.

```
• IEEE std 1076.5 Guidelines for modeling of libraries in VHDL.
```

• IEEE std 1076.6	Defines the part of VHDL intended for RTL syn-
	thesis.

Predefined packages

STANDARD

LRM

§ 14.2

The standard of VHDL. Is precompiled into the library "STD" (accessed via **USE STD. STANDARD.ALL** that is implicitly declared).

```
package STANDARD is
  type BOOLEAN is (FALSE, TRUE);
  type BIT is ('0', '1');
  type CHARACTER is (
    NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL,
    BS, HT, LF, VT, FF, CR, SO, SI, DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB,
             SUB, ESC, FSP, GSP, RSP, USP,
    ١٩,
                                    ۱&',
         19', 1:',
                    `;', `<',
                                          121
                    `C',
                               ۱E',
                         'D',
                                    ۱F',
    ۱@',
         'Α',
              `B',
                                         'G'
                               `M',
                                    'N',
                                          ١٥٠,
    'H', 'I', 'J', 'K', 'L', 'P', 'Q', 'R', 'S', 'T',
                                    ٠
'٧',
                                          ۱W',
    'X', 'Y', 'Z', '[', '\', ']',
```

```
``', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
   C128, C129, C130, C131, C132, C133, C134, C135,
   C136, C137, C138, C139, C140, C141, C142, C143,
   C144, C145, C146, C147, C148, C149, C150, C151,
   C152, C153, C154, C155, C156, C157, C158, C159,
        `;', `¢', `£', `¤', `\\', \\\', \\\',
   ۱þ′,
    ١ø٠,
        ۱Ù′,
            'Ú',
                 ۱Û′,
                      ۱Ü′,
                           ۱Ý',
                                     \B'
   'à', 'á', 'â', 'ă', 'a', 'å', 'æ', 'ç', 
'è', 'é', 'ê', 'ë', 'ì', 'i', 'î', 'ï', 'ï', 
'ð', 'ñ', 'ò', 'ô', 'ô', 'ô', 'ö', 'è',
   ۱ø′,
        'ù',
            'ú', 'û', 'ü', 'ý', 'þ',
 type SEVERITY_LEVEL is (NOTE, WARNING, ERROR, FAILURE);
 type INTEGER is range implementation defined;
 type REAL is range implementation_defined;
 type TIME is range implementation defined
   units
     fs;
     ps = 1000 fs;
     ns = 1000 ps;
     us = 1000 \text{ ns};
     ms = 1000 us;
     sec = 1000 ms;
     min = 60 sec;
     hr = 60 min;
   end units;
 subtype DELAY LENGTH is TIME range 0 fs to TIME'HIGH;
 impure function NOW return DELAY_LENGTH;
          is INTEGER range 0 to INTEGER'HIGH;
 subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;
 type STRING is array (POSITIVE range <>) of CHARACTER;
 type BIT VECTOR is array (NATURAL range <>) of BIT;
 type FILE_OPEN_KIND is (READ_MODE, WRITE_MODE, APPEND_MODE);
 attribute FOREIGN : STRING;
end package STANDARD;
```

TEXTIO

LRM

§ 14.3

Types and subprograms to handle text files. Is precompiled into the library "STD (accessed via USE STD.TEXTIO.ALL).

```
package TEXTIO is
  type LINE is access STRING;
  type TEXT is file of STRING;
  type SIDE is (RIGHT, LEFT);
  subtype WIDTH is natural;
 file INPUT : TEXT open READ_MODE is "STD INPUT";
 file OUTPUT : TEXT open WRITE MODE is "STD OUTPUT";
  procedure READLINE(file F:TEXT; L:out LINE);
 procedure READ(L:inout LINE; VALUE:out BIT;
                 GOOD: out BOOLEAN);
 procedure READ(L:inout LINE; VALUE:out BIT);
 procedure READ(L:inout LINE; VALUE:out BIT_VECTOR;
                 GOOD: out BOOLEAN);
 procedure READ(L:inout LINE; VALUE:out BIT VECTOR);
  procedure READ(L:inout LINE; VALUE:out BOOLEAN;
                 GOOD: out BOOLEAN);
 procedure READ(L:inout LINE; VALUE:out BOOLEAN);
```

```
procedure READ(L:inout LINE; VALUE:out CHARACTER;
                 GOOD: out BOOLEAN);
 procedure READ(L:inout LINE; VALUE:out CHARACTER);
 procedure READ(L:inout LINE; VALUE:out INTEGER;
                 GOOD: out BOOLEAN);
 procedure READ(L:inout LINE; VALUE:out INTEGER);
 procedure READ(L:inout LINE; VALUE:out REAL;
                 GOOD: out BOOLEAN);
 procedure READ(L:inout LINE; VALUE:out REAL);
 procedure READ(L:inout LINE; VALUE:out STRING;
                 GOOD: out BOOLEAN);
 procedure READ(L:inout LINE; VALUE:out STRING);
 procedure READ (L:inout LINE; VALUE:out TIME;
                 GOOD: out BOOLEAN);
 procedure READ(L:inout LINE; VALUE:out TIME);
 procedure WRITELINE(file f:TEXT; L:inout LINE);
 procedure WRITE(L:inout LINE; VALUE:in BIT;
                  JUSTIFIED: in SIDE := RIGHT;
                  FIELD: in WIDTH := 0);
 procedure WRITE(L:inout LINE; VALUE:in BIT_VECTOR;
                  JUSTIFIED:in SIDE := RIGHT;
                  FIELD: in WIDTH := 0);
 procedure WRITE (L:inout LINE; VALUE:in BOOLEAN;
                  JUSTIFIED: in SIDE := RIGHT;
                  FIELD:in WIDTH := 0);
 procedure WRITE (L:inout LINE; VALUE:in CHARACTER;
                  JUSTIFIED: in SIDE := RIGHT;
                  FIELD:in WIDTH := 0);
 procedure WRITE (L:inout LINE; VALUE:in INTEGER;
                  JUSTIFIED: in SIDE := RIGHT;
                  FIELD:in WIDTH := 0);
 procedure WRITE (L:inout LINE; VALUE:in REAL;
                  JUSTIFIED: in SIDE := RIGHT;
                  FIELD:in WIDTH := 0;
                  DIGITS:in NATURAL := 0);
 procedure WRITE(L:inout LINE; VALUE:in STRING;
                  JUSTIFIED: in SIDE := RIGHT;
                  FIELD: in WIDTH := 0);
 procedure WRITE(L:inout LINE; VALUE:in TIME;
                  JUSTIFIED: in SIDE := RIGHT;
                  FIELD: in WIDTH := 0;
                  UNIT:in TIME := ns);
  -- function ENDFILE (file F:TEXT) return BOOLEAN;
end package TEXTIO;
```

STD_LOGIC_1164

Hardware related, resolved types and conversion functions for them. Is precompiled into the library "IEEE" (accessed via USE IEEE.STD_LOGIC_1164.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;
  subtype STD LOGIC is RESOLVED STD ULOGIC;
  type STD LOGIC VECTOR is array (NATURAL range <>) of STD LOGIC;
  subtype X01
                   is resolved STD ULOGIC range 'X' to '1';
  subtype UX01 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';
  function "and" (1:STD_ULOGIC; r:STD_ULOGIC) return UX01;
```

```
function "nand" (1:STD ULOGIC; r:STD ULOGIC) return UX01;
  function "or" (1:STD_ULOGIC; r:STD_ULOGIC) return UX01; function "nor" (1:STD_ULOGIC; r:STD_ULOGIC) return UX01; function "xor" (1:STD_ULOGIC; r:STD_ULOGIC) return UX01; function "xnor" (1:STD_ULOGIC; r:STD_ULOGIC) return UX01; function "not" (1:STD_ULOGIC) return UX01; function "not" (1:STD_ULOGIC) return UX01;
  function "and" (1,r:STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
  function "and" (1, r:STD ULOGIC VECTOR)
                        return STD_ULOGIC_VECTOR;
  function "nand" (1,r:STD_LOGIC_VECTOR)
                        return STD LOGIC VECTOR;
  function "nand" (1, r:STD_ULOGIC_VECTOR)
                        return STD ULOGIC VECTOR;
  function "or"
                      (1,r:STD_LOGIC_VECTOR)
                       return STD_LOGIC_VECTOR;
  function "or"
                       (1, r:STD ULOGIC VECTOR)
                        return STD ULOGIC VECTOR;
  function "nor"
                     (l,r:STD LOGIC VECTOR)
                        return STD_LOGIC_VECTOR;
  function "nor"
                     (1,r:STD_ULOGIC_VECTOR)
                        return STD ULOGIC VECTOR;
  function "xor"
                     (1,r:STD LOGIC VECTOR)
                        return STD LOGIC VECTOR;
                      (1,r:STD_ULOGIC_VECTOR)
  function "xor"
                        return STD_ULOGIC_VECTOR;
  function "xnor" (1,r:STD_LOGIC_VECTOR)
                        return STD LOGIC VECTOR;
  function "xnor" (1, r:STD ULOGIC VECTOR)
                        return STD ULOGIC VECTOR;
  function "not"
                      (1:STD_LOGIC_VECTOR)
                        return STD_LOGIC_VECTOR;
  function "not" (1:STD_ULOGIC_VECTOR)
                        return STD_ULOGIC_VECTOR;
  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)
  runction TO_STDULOGICVECTOR (BIBIT_VECTOR)

return STD_ULOGIC_VECTOR;

function TO_STDULOGICVECTOR (s:STD_LOGIC_VECTOR)

return STD_ULOGIC_VECTOR;

function TO_X01 (s:STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;

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)
function TO_X01 (b:BIT_VECTOR)
                                                   return STD_LOGIC_VECTOR;
return STD_ULOGIC_VECTOR;
  function TO_X01 (b:BIT function TO X01 (b:BIT
                        (b:BIT)
                                                    return X01;
  function TO_X01Z (s:STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR; 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)
function TO_X01Z (b:BIT_VECTOR)
                                                    return STD_LOGIC_VECTOR;
return STD_ULOGIC_VECTOR;
  function TO_X01Z (b:BIT)
                                                    return X01\overline{Z};
  function TO_UX01 (s:STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
  function TO_UX01 (s:STD_ULOGIC_VECTOR) return STD_ULOGIC_VECTOR;
  function TO_UX01 (s:STD_ULOGIC)
function TO_UX01 (b:BIT_VECTOR)
function TO_UX01 (b:BIT_VECTOR)
                                                   return UX01;
                                                    return STD_LOGIC_VECTOR;
return STD_ULOGIC_VECTOR;
  function TO UX01 (b:BIT)
                                                    return UX01;
  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_LOGIC_VECTOR) return BOOLEAN;
  function IS_X (s:STD_ULOGIC)
                                               return BOOLEAN;
end package STD_LOGIC_1164;
```

NUMERIC BIT

Types and subprograms for designing with arrays of BIT. Is precompiled into the library "IEEE" (accessed via USE IEEE.NUMERIC BIT.ALL).

```
package NUMERIC_BIT is
   type UNSIGNED is array (NATURAL range <>) of BIT;
   type SIGNED
                        is array (NATURAL range <>) of BIT;
   function "abs" (ARG:SIGNED) return SIGNED;
   function
                           (ARG:SIGNED) return SIGNED;
   function "+"
                          (L,R:UNSIGNED)
                                                                    return UNSIGNED:
   function "+"
                          (L,R:SIGNED)
                                                                    return SIGNED:
   function "+"
                          (L:UNSIGNED; R:NATURAL) return UNSIGNED;
   function "+"
                           (L:NATURAL; R:UNSIGNED) return UNSIGNED;
                           (L:NATOLL:
(L:INTEGER; R:SIGNED;
TOMED; R:INTEGER)
   function "+"
                                                                    return SIGNED:
   function "+"
                                                                    return SIGNED;
   function "-"
                          (L,R:UNSIGNED)
                                                                    return UNSIGNED;
   function "-"
                          (L,R:SIGNED)
                                                                   return SIGNED;
   function "-"
                           (L:UNSIGNED; R:NATURAL) return UNSIGNED;
   function "-"
                           (L:NATURAL; R:UNSIGNED) return UNSIGNED;
   function "-"
                           (L:SIGNED;
                                                R:INTEGER)
                                                                    return SIGNED;
   function "-"
                          (L:INTEGER; R:SIGNED)
                                                                    return SIGNED;
   function "*"
                          (L,R:UNSIGNED)
                                                                    return UNSIGNED;
   function "*"
                           (L,R:SIGNED)
                                                                    return SIGNED;
   function "*"
                           (L:UNSIGNED; R:NATURAL)
                                                                    return UNSIGNED;
                           (L:NATURAL; R:UNSIGNEL R:INTEGER)
   function "*"
                                                R:UNSIGNED) return UNSIGNED;
   function "*"
                                                                   return SIGNED;
   function "*"
                          (L:INTEGER; R:SIGNED)
                                                                   return SIGNED;
   function "/"
                          (L,R:UNSIGNED)
                                                                   return UNSIGNED;
   function "/"
                           (L,R:SIGNED)
                                                                    return SIGNED;
   function "/"
                           (L:UNSIGNED; R:NATURAL)
                                                                    return UNSIGNED;
   function "/"
                           (L:NATURAL; R:UNSIGNED) return UNSIGNED;
   function "/"
                          (L:SIGNED;
                                               R:INTEGER) return SIGNED;
   function "/"
                           (L:INTEGER; R:SIGNED)
                                                                   return SIGNED;
   function "rem" (L,R:UNSIGNED)
                                                                    return UNSIGNED:
   function "rem"
                           (L,R:SIGNED)
                                                                    return SIGNED;
   function "rem" (L:UNSIGNED; R:NATURAL)
                                                                    return UNSIGNED;
   function "rem" (L:NATURAL; R:UNSIGNED) return UNSIGNED;
   function "rem" (L:SIGNED;
                                               R:INTEGER) return SIGNED;
   function "rem" (L:INTEGER; R:SIGNED)
                                                                    return SIGNED;
   function "mod"
                           (L,R:UNSIGNED)
                                                                    return UNSIGNED;
   function "mod" (L,R:SIGNED)
                                                                    return SIGNED;
   function "mod" (L:UNSIGNED; R:NATURAL) return UNSIGNED;
   function "mod" (L:NATURAL; R:UNSIGNED) return UNSIGNED;
   function "mod" (L:SIGNED;
                                               R:INTEGER) return SIGNED;
   function "mod" (L:INTEGER; R:SIGNED)
                                                                    return SIGNED;
   function ">"
                           (L,R:UNSIGNED)
                                                                    return BOOLEAN;
   function ">"
                           (L,R:SIGNED)
                                                                    return BOOLEAN;
   function ">"
                        (L:NATURAL; R:UNSIGNEI
(L:INTEGER; R:SIGNED)
                          (L:NATURAL; R:UNSIGNED) return BOOLEAN;
   function ">"
                                                                    return BOOLEAN;
   function ">"
                          (L:UNSIGNED; R:NATURAL) return BOOLEAN;
   function ">"
                           (L:SIGNED;
                                                R: INTEGER) return BOOLEAN;
   function "<"
                                                                    return BOOLEAN;
                           (L,R:UNSIGNED)
   function "<"
                           (L,R:SIGNED)
                                                                    return BOOLEAN;
   function "<"
                          (L:NATURAL; R:UNSIGNED) return BOOLEAN;
   function "<"
                          (L:INTEGER; R:SIGNED)
                                                                  return BOOLEAN;
   function "<"
                          (L:UNSIGNED; R:NATURAL)
                                                                    return BOOLEAN:
   function "<"
                           (L:SIGNED;
                                                R:INTEGER)
                                                                    return BOOLEAN;
   function "<="
                           (L,R:UNSIGNED)
                                                                    return BOOLEAN;
   function "<="
                          (L,R:SIGNED)
                                                                    return BOOLEAN;
   function "<="
                          (L:NATURAL; R:UNSIGNED) return BOOLEAN;
   function "<=" (L:INTEGER; R:SIGNED) return BOOLEAN; function "<=" (L:UNSIGNED D.NAMBERS | Control of the contro
   function "<="
                           (L:SIGNED;
                                             R:INTEGER)
                                                                    return BOOLEAN;
   function ">="
                         (L,R:UNSIGNED)
                                                                    return BOOLEAN;
   function ">="
                          (L,R:SIGNED)
                                                                    return BOOLEAN;
   function ">="
                          (L:NATURAL; R:UNSIGNED) return BOOLEAN;
   function ">="
                           (L:INTEGER;
                                                R:SIGNED)
                                                                    return BOOLEAN;
   function ">="
                           (L:UNSIGNED; R:NATURAL)
                                                                    return BOOLEAN;
   function ">="
                           (L.SIGNED:
                                               R:INTEGER)
                                                                  return BOOLEAN;
   function "="
                          (L,R:UNSIGNED)
                                                                    return BOOLEAN;
   function "="
                          (L,R:SIGNED)
                                                                    return BOOLEAN;
   function "="
                           (L:NATURAL; R:UNSIGNED) return BOOLEAN;
   function "="
                           (L:INTEGER; R:SIGNED)
                                                                    return BOOLEAN;
```

```
function "=" (L:UNSIGNED; R:NATURAL) return BOOLEAN; function "=" (L:SIGNED: P.TMMBODD:
  function "=" (L:SIGNED; R:INTEGER) return BOOLEAN; function "/=" (L.R:INSTENDED) return BOOLEAN;
  Lunction "/=" (L,R:UNSIGNED) function "/=" (T. B.O.
                                                            return BOOLEAN;
   function "/="
  function "/=" (L:NATURAL; R:UNSIGNED) return BOOLEAN;
function "/=" (L:INTEGER; R:SIGNED) return BOOLEAN;
  function "/=" (L:UNSIGNED; R:NATURAL) return BOOLEAN;
function "/=" (L:SIGNED; R:INTEGER) return BOOLEAN;
   function SHIFT_LEFT (ARG:UNSIGNED; COUNT:NATURAL)
                                return UNSIGNED;
  function SHIFT RIGHT
                                  (ARG:UNSIGNED; COUNT:NATURAL)
                                   return UNSIGNED;
                                 (ARG:SIGNED; COUNT:NATURAL) return SIGNED;
  function SHIFT_LEFT
  function SHIFT_RIGHT (ARG:SIGNED; COUNT:NATURAL) return SIGNED;
  function ROTATE LEFT (ARG:UNSIGNED; COUNT:NATURAL)
                                    return UNSIGNED;
  function ROTATE RIGHT (ARG:UNSIGNED; COUNT:NATURAL)
                                   return UNSIGNED;
  function ROTATE LEFT (ARG:SIGNED; COUNT:NATURAL) return SIGNED;
   function ROTATE_RIGHT (ARG:SIGNED; COUNT:NATURAL) return SIGNED;
  function "s11" (ARG:UNSIGNED; COUNT:INTEGER) return UNSIGNED;
function "s11" (ARG:SIGNED; COUNT:INTEGER) return SIGNED;
  function "srl" (ARG:UNSIGNED; COUNT:INTEGER) return UNSIGNED;
  function "srl" (ARG:SIGNED; COUNT:INTEGER) return SIGNED;
  function "rol" (ARG:UNSIGNED; COUNT:INTEGER) return UNSIGNED;
function "rol" (ARG:SIGNED; COUNT:INTEGER) return UNSIGNED;
function "ror" (ARG:UNSIGNED; COUNT:INTEGER) return UNSIGNED;
  function "ror" (ARG:SIGNED; COUNT:INTEGER) return SIGNED;
   function RESIZE (ARG:SIGNED; NEW SIZE:NATURAL) return SIGNED;
   function RESIZE (ARG: UNSIGNED; NEW SIZE: NATURAL) return UNSIGNED;
  function TO_INTEGER (ARG:UNSIGNED) return NATURAL;
   function TO
                    INTEGER
                                 (ARG:SIGNED)
                                                            return INTEGER;
  function TO UNSIGNED (ARG, SIZE:NATURAL) return UNSIGNED;
  function TO SIGNED (ARG: INTEGER; SIZE: NATURAL) return SIGNED;
  function "not" (L.R:UNSIGNED) return UNSIGNED;
function "and" (L,R:UNSIGNED) return UNSIGNED;
function "or" (L,R:UNSIGNED) return UNSIGNED;
function "nand" (L,R:UNSIGNED) return UNSIGNED;
  function "nor" (L,R:UNSIGNED) return UNSIGNED;
function "xor" (L,R:UNSIGNED) return UNSIGNED;
  function "xnor" (L,R:UNSIGNED) return UNSIGNED;

        function
        "not"
        (L:SIGNED)
        return
        SIGNED;

        function
        "and"
        (L,R:SIGNED)
        return
        SIGNED;

        function
        "or"
        (L,R:SIGNED)
        return
        SIGNED;

  function "nand" (L,R:SIGNED)
                                              return SIGNED;
  function "nor" (L,R:SIGNED) return SIGNED;
function "xor" (L,R:SIGNED) return SIGNED;
  function "xor" (L,R:SIGNED) return SIGNED;
function "xnor" (L,R:SIGNED) return SIGNED;
  function RISING EDGE (signal S:BIT) return BOOLEAN;
  function FALLING EDGE (signal S:BIT) return BOOLEAN;
end package NUMERIC BIT;
```

NUMERIC STD

use IEEE.std logic 1164.all;

library IEEE;

Types and subprograms for designing with arrays of STD_LOGIC. Is precompiled into the library "IEEE" (accessed via USE IEEE.NUMERIC STD.ALL).

```
package NUMERIC_STD is
  type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
type SIGNED is array (NATURAL range <>) of STD_LOGIC;
  function "abs" (X:SIGNED)
                                               return SIGNED;
  function "-" (ARG:SIGNED)
                                               return SIGNED;
  function "+"
                   (L,R:UNSIGNED)
                                               return UNSIGNED;
  function "+"
                   (L,R:SIGNED)
                                               return SIGNED;
  function "+"
                   (L:UNSIGNED; R:NATURAL) return UNSIGNED;
  function "+"
                  (L:NATURAL; R:UNSIGNED) return UNSIGNED;
  function "+"
                  (L:INTEGER; R:SIGNED)
(L:SIGNED; R:INTEGER)
                                              return SIGNED;
  function "+"
                                              return SIGNED;
  function "-"
                   (L,R:UNSIGNED)
                                               return UNSIGNED;
  function "-" (L,R:SIGNED)
                                               return SIGNED;
```

function "-" (L:UNSIGNED; R:NATURAL) return UNSIGNED;

```
(L:NATURAL; R:UNSIGNED) return UNSIGNED;
function "-"
function "-" (L:SIGNED; R:INTEGER) return UNSIGNED; function "-" (L:INTEGER: P.SIGNED;
function "*"
                    (L,R:UNSIGNED)
                                                     return UNSIGNED;
function "*"
                    (L,R:SIGNED)
                                                     return SIGNED;
function "*"
function "*"
function "*"
                    (L:UNSIGNED; R:NATURAL) return UNSIGNED;
                   (L:NATURAL; R:UNSIGNED) return UNSIGNED;
function "*"
                   (L:SIGNED;
                                    R:INTEGER) return SIGNED;
function "*"
                    (L:INTEGER; R:SIGNED)
                                                     return SIGNED;
function "/"
                    (L,R:UNSIGNED)
                                                     return UNSIGNED;
function // (L,R:SIGNED) return SIGNED;
function "/" (L:UNSIGNED; R:NATURAL) return UNSIGNED;
function "/" (L:NATURAL; R:UNSIGNED) return UNSIGNED;
function "/" (L:SIGNED; R:INTFGFP) --- UNSIGNED;
function "rem" (L.SIGNED; R:INTEGER) return UNSIGNE
function "rem" (L:NTEGER; R:SIGNED) return SIGNED;
function "rem" (L,R:UNSIGNED) return SIGNED;
function "rem" (L.B:SIGNED) return SIGNED;
function "rem" (L.B:SIGNED)
                                                     return UNSIGNED;
function "rem" (L:UNSIGNED; R:NATURAL) return UNSIGNED;
function "rem" (L:NATURAL; R:UNSIGNED) return UNSIGNED;
function "rem" (L:SIGNED; R:INTEGER) return SIGNED;
function "rem" (L:INTEGER; R:SIGNED) return SIGNED;
function "rem"
function "mod" (L,R:UNSIGNED)
                                                     return UNSIGNED;
function "mod" (L,R:SIGNED)
                                                     return SIGNED;
function "mod" (L:UNSIGNED; R:NATURAL) return UNSIGNED;
function "mod" (L:NATURAL; R:UNSIGNED) return UNSIGNED; function "mod" (L:SIGNED; R:INTEGER) return SIGNED; function "mod" (L:INTEGER; R:SIGNED) return SIGNED;
function ">"
                   (L,R:UNSIGNED)
                                                     return BOOLEAN;
function ">"
                   (L,R:SIGNED)
                                                     return BOOLEAN;
\quad \text{function } ">"
                   (L:NATURAL; R:UNSIGNED) return BOOLEAN;
function ">"
                    (L:INTEGER;
                                     R:SIGNED)
                                                     return BOOLEAN;
function ">"
                    (L:UNSIGNED; R:NATURAL)
                                                     return BOOLEAN;
function ">"
                   (L:SIGNED; R:INTEGER) return BOOLEAN;
function "<"
function "<"
                   (L,R:UNSIGNED)
                                                     return BOOLEAN;
                   (L,R:SIGNED)
                                                     return BOOLEAN;
function "<"
                    (L:NATURAL; R:UNSIGNED) return BOOLEAN; (L:INTEGER; R:SIGNED) return BOOLEAN;
function "<"
function "<" (L:UNSIGNED; R
function "<" (L:SIGNED; R
function "<=" (L,R:UNSIGNED)
function "<=" (L,R:SIGNED)
                                                   return BOOLEAN;
                   (L:UNSIGNED; R:NATURAL)
                   (L:SIGNED; R:INTEGER) return BOOLEAN;
                                                     return BOOLEAN;
                                                     return BOOLEAN;
function "<="
                    (L:NATURAL; R:UNSIGNED) return BOOLEAN; (L:INTEGER; R:SIGNED) return BOOLEAN;
function "<="
                                                     return BOOLEAN;
function "<=" (L:UNSIGNED; R:NATURAL) return BOOLEAN;</pre>
...cion "<=" (L:SIGNED;
function ">=" /T F
function ">=" /T F
                                   R:INTEGER) return BOOLEAN;
Lanction ">=" (L,R:UNSIGNED) function ">=" (T. P.OTT
                                                     return BOOLEAN;
                                                     return BOOLEAN;
function ">="
                    (L:NATURAL; R:UNSIGNED) return BOOLEAN;
function ">=" (L:INTEGER; R:SIGNED)
                                                     return BOOLEAN;
function ">="
function ">=" (L:UNSIGNED; R:NATURAL) return BOOLEAN;
function ">=" (L:SIGNED; R:INTEGER) return BOOLEAN;
                    (L,R:UNSIGNED)
function "="
                                                     return BOOLEAN;
function "="
                    (L,R:SIGNED)
                                                     return BOOLEAN;
function "="
                    (L:NATURAL; R:UNSIGNED) return BOOLEAN;
function "=" (L:INTEGER; R:SIGNED) return BOOLEAN; function "=" (L:UNSIGNED; R:NATURAL) return BOOLEAN; function "=" (L:SIGNED; R:INTEGER) return BOOLEAN;
function "/="
                    (L,R:UNSIGNED)
                                                     return BOOLEAN:
function "/="
                    (L,R:SIGNED)
                                                     return BOOLEAN;
function "/=" (L:NATURAL; R:UNSIGNED) return BOOLEAN;
function "/=" (L:INTEGER; R:SIGNED) return BOOLEAN; function "/=" (L:UNSIGNED: R:NATHIDAT | return BOOLEAN;
function "/=" (L:UNSIGNED; R:NATURAL) return BOOLEAN; function "/=" (L:SIGNED P.TUMBGET)
function SHIFT_LEFT (ARG:UNSIGNED; COUNT:NATURAL)
                             return UNSIGNED;
function SHIFT RIGHT (ARG:UNSIGNED; COUNT:NATURAL)
                             return UNSIGNED;
function SHIFT_LEFT (ARG:SIGNED; COUNT:NATURAL) return SIGNED;
function SHIFT
                    RIGHT (ARG:SIGNED; COUNT:NATURAL) return SIGNED;
function ROTATE_LEFT
                            (ARG:UNSIGNED; COUNT:NATURAL)
                              return UNSIGNED;
function ROTATE RIGHT (ARG:UNSIGNED; COUNT:NATURAL)
                              return UNSIGNED;
function ROTATE_LEFT (ARG:SIGNED; COUNT:NATURAL) return SIGNED;
function ROTATE RIGHT (ARG:SIGNED; COUNT:NATURAL) return SIGNED;
```

```
function "s11" (ARG:SIGNED; COUNT:INTEGER) return UNSIGNE function "sr1" (ARG:SIGNED; COUNT:INTEGER) return SIGNED; function "sr1" (ARG:UNSIGNED: COUNT:INTEGER)
                      (ARG:UNSIGNED; COUNT:INTEGER) return UNSIGNED;
  function "srl" (ARG:UNSIGNED; COUNT:INTEGER) return UNSIGNED;
function "srl" (ARG:SIGNED; COUNT:INTEGER) return SIGNED;
  function "rol" (ARG:UNSIGNED; COUNT:INTEGER) return UNSIGNED;
  function "rol" (ARG:SIGNED; COUNT:INTEGER) return SIGNED;
  function "ror"
                      (ARG:UNSIGNED; COUNT:INTEGER) return UNSIGNED;
  function "ror" (ARG:SIGNED; COUNT:INTEGER) return SIGNED;
  function RESIZE (ARG:SIGNED; NEW_SIZE:NATURAL) return SIGNED;
  function RESIZE (ARG:UNSIGNED; NEW SIZE:NATURAL) return UNSIGNED;
  function TO_INTEGER (ARG:UNSIGNED) return NATURAL;
  function TO INTEGER (ARG:SIGNED)
                                                      return INTEGER;
  function TO UNSIGNED (ARG, SIZE: NATURAL) return UNSIGNED;
  function TO_SIGNED (ARG:INTEGER; SIZE:NATURAL) return SIGNED;
  function "not" (L:UNSIGNED) return UNSIGNED;
function "and" (L,R:UNSIGNED) return UNSIGNED;
  function "or" (L,R:UNSIGNED) return UNSIGNED;
  function "nand" (L,R:UNSIGNED) return UNSIGNED;
  function "nor" (L,R:UNSIGNED) return UNSIGNED;
function "xor" (L,R:UNSIGNED) return UNSIGNED;
                       (L,R:UNSIGNED) return UNSIGNED;
  function "xnor" (L,R:UNSIGNED) return UNSIGNED;
  function "not"
                      (L:SIGNED)
                                          return SIGNED;
  function "and" (L,R:SIGNED) return SIGNED;
function "or" (L,R:SIGNED) return SIGNED;
  function "nand" (L,R:SIGNED) return SIGNED;
  function "nor" (L,R:SIGNED)
function "xor" (L,R:SIGNED)
                                        return SIGNED;
return SIGNED;
  function "xor" (L,R:SIGNED) return SIGNED;
function "xnor" (L,R:SIGNED) return SIGNED;
  function STD MATCH (L,R:STD ULOGIC) return BOOLEAN;
  function STD_MATCH (L,R:UNSIGNED) return BOOLEAN;
function STD MATCH (L,R:SIGNED) return BOOLEAN;
  function STD_MATCH (L,R:SIGNED)
  function STD_MATCH (L,R:SIGNED) Teturn BOOLEAN; function STD_MATCH (L,R:STD_LOGIC_VECTOR) return BOOLEAN; function STD_MATCH (L,R:STD_ULOGIC_VECTOR) return BOOLEAN;
  function TO 01 (S:UNSIGNED; XMAP:STD LOGIC:='0') return UNSIGNED;
  function TO 01 (S:SIGNED; XMAP:STD LOGIC:='0') return SIGNED;
end package NUMERIC_STD;
```

MATH_REAL

Constants and subprograms for real numbers. Is precompiled into the library "IEEE" (accessed via USE IEEE.MATH REAL.ALL).

```
function SQRT (X:REAL)
function CBRT (X:REAL)
function "**" (X:INTEGE
                                                              return REAL;
                                                             return REAL;
                            (X:INTEGER; Y:REAL) return REAL;
   function "**"

        function "**"
        (X:REAL;
        Y:REAL)
        return REAL;

        function EXP
        (X:REAL)
        return REAL;

        function LOG
        (X:REAL)
        return REAL;

   function LOG
   function LOG (X:REAL)

function LOG2 (X:REAL)
                                                              return REAL;
   function LOG10 (X:REAL)
                                                              return REAL;
   function LOG
                             (X:REAL; BASE:REAL) return REAL;
   function SIN
                              (X:REAL)
                                                              return REAL;
   function COS
                              (X · REAL)
  function TAN
                                                              return REAL;
                             (X:REAL)
                                                             return REAL;
                                                             return REAL;
   function ARCSIN (X:REAL)
   function ARCCOS (X:REAL)
                                                            return REAL;

        function
        ARCTAN
        (Y:REAL)
        return
        REAL;

        function
        ARCTAN
        (Y:REAL; X:REAL)
        return
        REAL;

        function
        SINH
        (X:REAL)
        return
        REAL;

   function COSH
                            (X:REAL)
                                                             return REAL;
   function TANH
                             (X:REAL)
                                                             return REAL;
   function ARCSINH (X:REAL)
                                                             return REAL;
   function ARCCOSH (X:REAL)
                                                              return REAL;
                                                            return REAL;
   function ARCTANH (X:REAL)
end package MATH REAL;
```

MATH COMPLEX

use IEEE.MATH REAL.all;

Constants and subprograms for complex numbers. Is precompiled into the library "IEEE" (accessed via USE IEEE.MATH COMPLEX.ALL).

```
package MATH COMPLEX is
  type COMPLEX is record
    RE : REAL; -- Real part
    IM : REAL; -- Imaginary part
  end record:
  subtype POSITIVE REAL is REAL range 0.0 to REAL'HIGH;
  subtype PRINCIPAL_VALUE is REAL range -MATH_PI to MATH_PI;
  type COMPLEX POLAR is record
    MAG : POSITIVE_REAL; -- Magnitude
ARG : PRINCIPAL_VALUE; -- Angle in radians
 function "=" (L:COMPLEX_POLAR; R:COMPLEX_POLAR)
                   return BOOLEAN;
  function "/=" (L:COMPLEX POLAR; R:COMPLEX POLAR)
                   return BOOLEAN;
  function CMPLX (X:REAL; Y:REAL:=0.0) return COMPLEX;
  function GET PRINCIPAL VALUE (X:REAL) return PRINCIPAL VALUE;
  function COMPLEX_TO_POLAR (Z:COMPLEX) return COMPLEX_POLAR;
  function POLAR_TO_COMPLEX (Z:COMPLEX_POLAR) return COMPLEX;
  function "ABS"
                    (Z:COMPLEX)
                                              return POSITIVE REAL;
  function "ABS" (Z:COMPLEX_POLAR)
                                             return POSITIVE_REAL;
  function ARG (Z:COMPLEX)
                                             return PRINCIPAL VALUE;
                   (Z:COMPLEX_POLAR) return PRINCIPAL_VALUE;
  function ARG
  runction "-" (Z:COMPLEX) return COMPLEX;
function "CONJ (Z:COMPLEX POLAR) return COMPLEX;
function CONJ (Z:COMPLEX) return COMPLEX;
function CONJ (Z:COMPLEX)
  function CONJ (Z:COMPLEX_POLAR) return COMPLEX_POLAR;
  function SQRT (Z:COMPLEX)
                                             return COMPLEX;
                                             return COMPLEX POLAR;
  function SQRT (Z:COMPLEX_POLAR)
function EXP (Z:COMPLEX)
  function EXP (Z:COMPLEX)
function EXP (Z:COMPLEX)
                                             return COMPLEX;
return COMPLEX POLAR;
                   (Z:COMPLEX_POLAR)
  function LOG
                                             return COMPLEX;
                   (Z:COMPLEX)
  function LOG2 (Z:COMPLEX)
                                              return COMPLEX;
                                             return COMPLEX;
  function LOG10 (Z:COMPLEX)

    function
    LOG
    (Z:COMPLEX_POLAR)
    return
    COMPLEX_POLAR;

    function
    LOG2
    (Z:COMPLEX_POLAR)
    return
    COMPLEX_POLAR;

  function LOG2 (Z:COMPLEX_POLAR) return COMPLEX_POLAR;
function LOG10 (Z:COMPLEX POLAR) return COMPLEX POLAR;
  function LOG (Z:COMPLEX; BASE:REAL) return COMPLEX;
```

```
function LOG
                 (Z:COMPLEX POLAR; BASE:REAL)
                   return COMPLEX POLAR;
  function SIN
                 (Z:COMPLEX)
                                            return COMPLEX;
                                           return COMPLEX POLAR;
  function SIN
                  (Z:COMPLEX POLAR)
  function COS
                  (Z:COMPLEX)
                                            return COMPLEX;
  function COS
                  (Z:COMPLEX POLAR)
                                           return COMPLEX_POLAR;
  function SINH (Z:COMPLEX)
                                           return COMPLEX;
  function SINH (Z:COMPLEX POLAR)
                                           return COMPLEX POLAR;
  function COSH (Z:COMPLEX)
                                           return COMPLEX;
  function COSH
                  (Z:COMPLEX POLAR)
                                            return COMPLEX POLAR;
                  (L:COMPLEX; R:COMPLEX) return COMPLEX;
  function "+"
  function "+"
                  (L:REAL; R:COMPLEX) return COMPLEX;
 function "+" (L:COMPLEX; R:REAL) return COMP
function "+" (L:COMPLEX_POLAR; R:COMPLEX_POLAR)
                                           return COMPLEX;
                   return COMPLEX POLAR;
  function "+"
                  (L:REAL; R:COMPLEX POLAR) return COMPLEX POLAR;
  function "+" (L:COMPLEX_POLAR; R:REAL) return COMPLEX_POLAR;
  function "-"
                (L:COMPLEX; R:COMPLEX) return COMPLEX;
  function "-"
                 (L:REAL; R:COMPLEX)
                                              return COMPLEX;
  function "-"
                  (L:COMPLEX; R:REAL)
                                               return COMPLEX:
  function "-"
                  (L:COMPLEX POLAR; R:COMPLEX POLAR)
                   return COMPLEX POLAR;
  function "-"
                 (L:REAL; R:COMPLEX POLAR) return COMPLEX POLAR;
  function "-"
                 (L:COMPLEX POLAR; R:REAL) return COMPLEX POLAR;
                 (L:COMPLEX; R:COMPLEX) return COMPLEX;
(L:REAL; R:COMPLEX) return COMPLEX;
(L:COMPLEX; R:REAL) return COMPLEX;
  function "*"
  function "*"
  function "*"
  function "*" (L:COMPLEX POLAR; R:COMPLEX POLAR)
                   return COMPLEX POLAR;
  function "*"
                 (L:REAL; R:COMPLEX_POLAR) return COMPLEX_POLAR;
  function "*"
                  (L:COMPLEX_POLAR; R:REAL) return COMPLEX_POLAR;
  function "/"
                  (L:COMPLEX; R:COMPLEX) return COMPLEX;
(L:REAL; R:COMPLEX) return COMPLEX;
                               R:COMFLL.,
R:COMPLEX)
  function "/"
  function "/"
                (L:COMPLEX; R:REAL)
                                              return COMPLEX;
  function "/"
                (L:COMPLEX POLAR; R:COMPLEX POLAR)
                   return COMPLEX POLAR;
  function "/"
 function "/" (L:REAL; R:COMPLEX_POLAR) return COMPLEX_POLAR;
function "/" (L:COMPLEX_POLAR; R:REAL) return COMPLEX_POLAR;
end package MATH COMPLEX;
```

VHDL guide

Introduction

This chapter describes how VHDL code is written at HARDI Electronics AB. This coding method is used and taught at the training course "VHDL for design and modelling".

File notation

Files for the five design units in VHDL are named according to the following:

```
Entity Name_e.vhd
Architecture Name_a.vhd
Package Name_p.vhd
Package Body Name_b.vhd
Configuration Name_c.vhd
```

Libraries are named arbitrarily.

Predefined packages

It is recommended to use the packages standardized by IEEE (see pages 56-65). Non standardized packages, for example STD_LOGIC_ARITH, STD_LOGIC_SIGNED, NUMERIC_SIGNED, STD_LOGIC_UNSIGNED and NUMERIC UNSIGNED, are to be avoided.

VHDL syntax

Reserved words and objects

All reserved words are written with upper-case letters. Objects (constants, variables, signals and files) are written with lower-case letters as compound words, and with upper-case letters separating the words. The first letter is always lower-case.

```
Counter: PROCESS
BEGIN
WAIT UNTIL clock = '1';
numberOfCycles <= numberOfCycles + 1;
END PROCESS Counter;</pre>
```

Types

Predefined types are written with upper-case letters. Userdefined types are written just as objects, but with the first letter as upper-case.

```
TYPE Matrix IS ARRAY (NATURAL RANGE <>, NATURAL RANGE <>) OF BIT; SUBTYPE MyNumbers IS NATURAL RANGE 0 TO 7;
```

Attributes, packages and libraries

Attributes, packages and libraries are written with upper-case letters except for user-defined attributes and packages that are written with lower-case letters (packages start with an upper-case letter).

```
WAIT UNTIL clk'EVENT AND clk = '1';
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ATTRIBUTE syn encoding OF State : TYPE IS "onehot";
```

Hexadecimal values

Hexadecimal values are written using upper-case letters.

```
myVariable := X"AB14";
```

Replacement characters

Replacement characters are not used (LRM 13.10).

Indentation

Indentation is used to make the VHDL code well structured and easy to read. Declarative regions and expressions are indented two spaces. TAB is not used since the TAB setting may differ between different computers.

```
ARCHITECTURE Behave OF Counter IS
BEGIN

Count: PROCESS (choose)

VARIABLE myNumber : INTEGER := 0;
BEGIN

IF choose = '1' THEN

myNumber := myNumber + 1;

REPORT "Adding";

ELSE

myNumber := myNumber - 1;

REPORT "Subtracting";

END IF;

outputData <= myNumber;

END PROCESS Count;

END ARCHITECTURE Behave;
```

Indentation of the CASE statement

CASE statements are indented to clearly show the different conditions.

Line construction

A line shall not include more than 80 characters. Longer lines shall be divided into several lines and all new lines shall continue at a similar expression on the previous line.

```
IF myVeryVeryLongVariableName AND
   mySecondVeryVeryLongVariableName THEN ...

PORT (clk : IN BIT;
   myNumber : IN INTEGER;
   ok : OUT BOOLEAN);
```

Comments

Comments are placed either before the code to be commented, or on the same line.

```
-- This process adds two numbers
Adder: PROCESS(number1, number2) IS
BEGIN
sum <= number1 + number2; -- Here is the addition
END PROCESS Adder;
```

Concurrent statements

Concurrent statements are separated by an empty line.

```
ARCHITECTURE Behave OF Design IS
BEGIN

ASSERT a = b
REPORT "a is not equal to b";

clock <= NOT clock AFTER 20 ns;

PROCESS(clock)
BEGIN
IF clock = '1' THEN
q <= d;
END IF;
END PROCESS;
END ARCHITECTURE Behave;
```

Operators

Space is used to separate operators binding harder or as hard as binary '+' and '-'. Operators written as text are always preceded by a space.

With space	Without space
+	*
-	/
MOD, REM, ABS	**
shift, rotation	
logical	
comparison	

Assignments

Every assignment is written on a separate line justified vertically according to equal lines.

```
outData <= inData AFTER 5 ns;
dataBus <= GUARDED memory(address);</pre>
```

Declarations

Each declaration is written on a separate line justified vertically according to equal lines.

```
SIGNAL outData : BIT;
SIGNAL dataBus : Resolve INTEGER BUS;
```

Associations

Named associations using more than one line are justified vertically. Named association is preferably used instead of positional association since the association then is independent of the order of the associations.

```
PORT MAP(data => outData, address => address,
    wr => wr, rd => rd);
```

Naming of concurrent statements

Concurrent statements shall be named to simplify simulation. The name shall directly be followed by a ':', then a space and then the named concurrent statement. The name is always started by an upper-case letter.

```
Clockpulse: clock <= NOT clock AFTER 5 ns;
Adder: PROCESS(number1, number2) IS
BEGIN
    sum <= number1 + number2;
END PROCESS Adder;</pre>
```



Loops

Nested loops shall be named. NEXT and EXIT shall denote which loop they belong to.

```
L1: WHILE a > b LOOP

L2: FOR i IN a TO b LOOP

a := i - b;
b := b - 1;
EXIT L1 WHEN a = b;
END LOOP L2;
END LOOP L1;
```

END

END shall be qualified when the syntax permits it.

Simulation and synthesis

Variables and signals

Variables are prefered over signals since they need less simulation memory and are handled faster

Enumerated types

Enumerated types are prefered for state machines.

```
TYPE state IS (Start, Glitch, Pulse, Stop);
```

Constraints

Integers must be constrained to avoid 32 bit data paths in code written for synthesis.

```
SIGNAL data : INTEGER RANGE 0 TO 255;
```

Named constants are used when constraining parameters.

```
CONSTANT length : NATURAL := 5;
SUBTYPE MyArrayType IS BIT_VECTOR(length - 1 DOWNTO 0);
SIGNAL myArray : MyArrayType;
```

VHDL'87 and VHDL'93, differences

This chapter describes the most essential differences between VHDL'87 and VHDL'93

Syntax

VHDL'93 has a more symmetric syntax, especially for the conclusion of composite statements and the five design units. Below follow examples of conclusions different in the two versions of the standard (note that the VHDL'87 syntax is permitted also in VHDL'93):

VHDL'87	VHDL'93	See page
<pre>end entity_name;</pre>	<pre>end entity entity_name;</pre>	24
end arch name;	end architecture arch name;	25
end pck name;	end package pck name;	22
end pck name;	end package body pck name;	23
end conf name;	end configuration conf name;	26, 48
end component;	end component comp name;	44
end fn name;	end function fn name;	31
end proc name;	end procedure proc name;	32
end record;	end record rec name;	9
	_	

Below follow examples where the start of statements differ:

VHDL'87	VHDL'93	See page
blk_name: block	blk_name: block is	41
proc_name: process	proc_name: process is	38
<pre>component comp_name</pre>	<pre>component comp_name is</pre>	44

Above the previous examples VHDL'93 permits labeling of all statements. Composite statements may then use the label also at the end of the statement, for example:

```
Control: IF a > b THEN
   ...
END IF Control;
```

That is not permitted in VHDL'87.

GENERATE

The GENERATE statement (see page 45) has in VHDL'93 been enhanced with a declarational part and has also been raised to a block with a local scope. The GENERATE statement in VHDL'87 does not have a declarational part. It is however possible to write code compatible with both standards by avoiding the declarational part and by putting a BLOCK statement within the GENERATE statement.

Concurrent signal assignment

Conditional concurrent signal assignments (see page 41) must in VHDL'87 have an concluding ELSE condition. The reserved word UNAFFECTED, that is new to VHDL'93, was included to be able to leave a signal unaffected during an assignment, i.e. to keep its previous value:

Files

The handling of files differ quite a lot between VHDL'87 and VHDL'93 (see page 19-21). Most changes are not backwards compatible. Below follow examples of the different versions of file declarations:

```
-- VHDL'87:

FILE f1: myFile IS IN "name_in_file_system";

FILE f2: mySecondFile IS OUT "name_in_file_system";

-- VHDL'93:

FILE f1: myFile OPEN READ_MODE IS "name_in_file_system";

FILE f2: mySecondFile OPEN WRITE_MODE IS "name_in_file_system";
```

Input files may be written in VHDL code compatible with both VHDL'87 and VHDL'93, but for output files that is not possible:

```
-- Declaration of an input file both for VHDL'87 and VHDL'93 FILE f : myFile IS "name_in_file_system";
```

The predefined subprograms FILE_OPEN and FILE_CLOSE does not exist in VHDL'87.

File parameters for subprograms do not have a mode in VHDL'93 as they do in VHDL'87. Input files for subprograms may be written in VHDL code compatible with both VHDL'87 and VHDL'93:

```
-- Subprogram with a file parameter for both VHDL'87 and VHDL'93 PROCEDURE ReadFile(FILE f : myFile; value : OUT INTEGER);
```

Functions using files outside their local scope must in VHDL'93 be declared as IMPURE. IMPURE does not exist in VHDL'87.

Character set

The character set in VHDL'93 (see page 7, 56-57) is completely ISO 8859-1: 1987(E) compatible and includes 256 characters. The character set in VHDL'87 is limited to the first 128 characters and does not include international characters, not even in comments. Many VHDL'87 tools do however support international characters in comments.

Extended identifiers

VHDL'93 permits the usage of extended identifiers. An extended identifier always starts and ends with a '\' (backslash) and may include for example spaces and reserved words. Note that extended identifiers are case sensitive.

Shared variables

VHDL'93 permits shared variables (see page 17) in concurrent declaration statements.

Impure functions

An impure function does not only work via its parameters and may therefore return different values with identical input parameters. A function calling an impure function, or a procedure with side-effects (a procedure not only working via its parameters), must be declared as impure. The function NOW, that returns current simulation time, is an impure function in VHDL'93. All functions utilizing NOW must therefore be declared as impure.

Direct instantiation

In VHDL'93 it is permitted to exclude the component declaration and directly instantiate an ENTITY or a CONFIGURATION DECLARATION. This is called direct instantiation (see pages 46, 47). In VHDL'87 a component declaration is needed.

Port associations

In VHDL'93 it is permitted to have a constant value as actual parameter for an input port in a *parameter association list* (see pages 47-50). In VHDL'87 an actual parameter must be a signal.

VHDL'93 does also permit, above type conversion functions, that direct type conversion (type conversion functions between closely related types) is used between formal and actual parameters (see pages 43, 44, 47). In VHDL'93 it is also possible to have a *slice* as formal parameter.

Attributes

A number of new attributes (see pages 51-55) are added to VHDL'93. They are 'ASCENDING, 'IMAGE, 'VALUE, 'DRIVING, 'DRIVING_VALUE, 'SIMPLE NAME, 'INSTANCE NAME and 'PATH NAME.

The lack of the attribute 'IMAGE in VHDL'87 may be quite annoying and one must write functions that convert values to text strings. In some cases it is possible to utilize STD.TEXTIO.READ and STD.TEXTIO.WRITE to create such functions, at least for the predefined types:

```
FUNCTION INTEGER IMAGE(i : INTEGER) RETURN STRING IS
 USE STD. TEXTIO. ALL:
  -- Determines the number of characters in the string
  FUNCTION length(i : INTEGER) RETURN NATURAL IS
    VARIABLE 1 : LINE;
    VARIABLE tmp : NATURAL;
  BEGIN
   WRITE(1,i);
   tmp := l'LENGTH;
    DEALLOCATE(1); -- Remove the line pointer
    RETURN tmp;
 END FUNCTION length;
  VARIABLE st : STRING(1 TO length(i));
 VARIABLE 1 : LINE;
BEGIN
  WRITE(1,i);
 st := 1.ALL;
 DEALLOCATE(1); -- Remove the line pointer
 RETURN st;
END FUNCTION INTEGER IMAGE;
```

The attributes 'STRUCTURE and 'BEHAVIOR were removed to VHDL'93.

REPORT

The REPORT statement is new to VHDL'93. In VHDL'87 it is possible to utilize REPORT in combination with ASSERT:

```
ASSERT FALSE REPORT "...";
```



Signal delay mechanisms

INERTIAL is new to VHDL'93 and is used to express an inertial delay (see page 37). In VHDL'93 it is possible to combine INERTIAL and TRANS-PORT in a signal assignment using REJECT. That is not possible in VHDL'87 and an extra signal is needed to obtain the same functionality:

```
-- VHDL'93:

a <= REJECT 2 ns INERTIAL b AFTER 5 ns;

-- VHDL'87:

tmp <= b AFTER 2 ns;

a <= TRANSPORT tmp AFTER 3 ns;
```

Delayed concurrent statements

In VHDL'93 it is possible to declare all concurrent statements active during simulation (see pages 38-42) as POSTPONED which means that they are executed as the final delta at a specific occasion. VHDL'87 does not have that functionality and there are no tricks to manually create it.

Alias

In VHDL'87 aliases (see page 22) may be declared only for objects, while it in VHDL'93 is possible to declare aliases also for subprograms, operators, types and for all named entities except "labels", "loop parameters" and "generate parameters".

Bit string literals

In VHDL'87 a bit string literal is always of the type BIT_VECTOR. In VHDL'93 the bit string literals have been generalized to be an alternative way to write an aggregate of any array type whose elements can have the values '0' or '1'.

```
-- Permitted in VHDL'93
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
...
SIGNAL s : STD_LOGIC_VECTOR(0 TO 15);
...
s <= x"A1B0";
```

Unfortunately this generalization may arise ambiguousnesses when overloaded subprograms are used. The assignment above should be written like this in VHDL'87:

```
s <= TO STDLOGICVECTOR(x"A1B0");
```

This will however result in a compilation error in VHDL'93 since the bit string literal fits many different array types, and it is therefore not possible for the compiler to determine which of all conversion functions named TO STDLOGICVECTOR to use.

The following line works fine both for VHDL'87 and for VHDL'93:

```
s <= TO_STDLOGICVECTOR(BIT_VECTOR'(x"A1B0"));
```

Index	File declaration 19, 71
	File handling 19, 71
A	File reading, TEXTIO 20 File writing, TEXTIO 21
'ACTIVE 54	Formal parameters 33, 34, 39
'ASCENDING Arrays 53	FUNCTION 33
'ASCENDING Types 51 ACCESS 12	G
Actual parameters 39	
AFTER 37, 41, 73	GENERATE 45 , 70 GENERIC 44
Aggregates 13	GENERIC MAP 44
ALIAS 22, 73	Generic parameters 44
ARCHITECTURE 27	GROUP 14
ARRAY 10	GUARD 43
ASSERT 38 ATTRIBUTE 15	H
Attributes	'HIGH, Arrays 52
Predefined 51, 72	'HIGH, Types 51
User-defined 15	
В	I
	'IMAGE 51, 72
'BEHAVIOR 72	'INSTANCE_NAME 55
'BASE 51 Backus-Naur-format 7	Identifier 4 IEEE standards 56
BIT 8, 56	IF 30
BIT VECTOR 8, 57	IMPURE FUNCTION 33, 71
Bit string literals 73	Incremental binding 50
BLOCK 43	INERTIAL 37 , 73
BOOLEAN 8, 56	INPUT 20
C	INTEGER 8, 57
CASE 31	L
CHARACTER 8, 56, 71	'LAST_ACTIVE 54
Component configuration 48, 49, 50	'LAST_EVENT 54
Component declaration 46	'LAST_VALUE 54
Component instantiation 47	'LEFT, Arrays 52
Concatenation operator (&) 10	'LEFT, Types 51
Concurrent signal assignment 41 , 42 , 70 Configuration (default) 48	'LEFTOF 52
CONFIGURATION (design unit) 28	'LENGTH 53 'LOW, Arrays 52
Configuration declaration 28, 50	'LOW, Types 51
Configuration specification 49	Lexical elements 4
CONSTANT 16	LIBRARY 23
D	LINE 12, 20, 21, 57
'DELAYED 54	Literals 5
'DRIVING 54	LOOP 32
'DRIVING_VALUE 18, 54	M
DEALLOCATE 12	MATH_COMPLEX package 64
Default configuration 48	MATH_REAL package 63
Deferred constant declaration 16	N
DELAY_LENGTH 8, 57	
Delay mechanisms 37, 73	Named association
Direct instantiation 46, 47, 72	Aggregates 13 Generic parameters 44
E	Ports 43, 47, 48, 49, 50
'EVENT 54	Subprograms 33, 34, 39
ENTITY 26	NATURAL 8, 57
EXIT 32	NEW 12
Extended identifiers 4, 71	NEXT 32
F	NOW 29 , 71
FILE_OPEN_KIND 8 , 57, 71	NULL 5, 12, 31
FILE OPEN STATUS 8, 57, 71	NUMERIC_BIT package 60 NUMERIC STD package 61
	NOWIENIC_STD package 01



O	KEPORI 38, /2
OPEN 47, 49	Reserved words 6
Operators	RETURN 35
Comparing 8 , 10, 11, 12	Rotation operators 8, 10
Concatenation (&) 10	S
Logical 8, 10	
Predefined 8	'SIMPLE_NAME 54
	'STABLE 53
Rotation 8, 10	'STRUCTURE 72
Shift 8, 10	'SUCC 52
OTHERS 13, 31	SELECT 42
OUTPUT 21	Sensitivity list 40
P	SEVERITY 8, 38, 57
2DATH NAME 55	SHARED VARIABLE 17, 71
'PATH_NAME 55	Shift operators 8, 10
'POS 51	SIGNAL 18
'PRED 52	Signal assignment 37, 41, 42
PACKAGE BODY 25	Signal declaration 18
PACKAGE DECLARATION 24	Slice 10, 72
Pointers 12	STANDARD package 56
PORT MAP 43, 44, 47, 72	STD_LOGIC_1164 package 58
Positional association	STRING 8, 57
Aggregates 13	Subprograms
Generic parameters 44	
Ports 43, 47, 48, 49, 50	Call 39 , 71 Function 33
Subprograms 33, 34, 39	Procedure 34
POSITIVE 8, 57	
POSTPONED 38, 40, 41, 42, 73	Subtypes
POSTPONED ASSERT 38, 73	Declaration 9
POSTPONED PROCESS 40, 73	Predefined 8, 56
Predefined attributes on:	Syntax description 7
Arrays 52	T
Named entities 54	2TD ANG ACTION 54
Signals 53	'TRANSACTION 54
Types 51	TEXTIO 19, 20, 21
Predefined operators 9	TEXTIO package 57
Predefined packages 56	TIME 8 , 57
MATH COMPLEX 64	TRANSPORT 37, 73
MATH REAL 63	Types
NUMERIC BIT 60	Declaration 9
	Predefined 8 , 56
NUMERIC_STD 61	U
STANDARD 56	
STD_LOGIC_1164_58	UNAFFECTED 41, 42
TEXTIO 57	USE 23
Predefined subtypes 8, 56	V
Predefined types 8 , 56	2374.1 52
PROCEDURE 34	'VAL 52
PROCESS 40	'VALUE 51
PURE FUNCTION 33	VARIABLE 17
Q	Variable assignment 36
•	Variable declaration 17
'QUIET 54	VHDL'87/93, differences 70
R	VHDL guide 66
2DANGE 52	\mathbf{W}
'RANGE 53	***
'REVERSE_RANGE 53	WAIT 29
'RIGHT, Arrays 53	WHEN 41
'RIGHT, Types 51	WORK 23
'RIGHTOF 52	WRITE 21
READ 20	WRITELINE 21
READLINE 20	
REAL 8, 57	
RECORD 11	
REJECT 37 , 73	

A wealth of experience

Founded in 1987, in the same year that saw the standardization of the original version of VHDL by the IEEE, **HARDI Electronics**, acquired by **Synplicity**, **Inc. in June 2007**, is a pioneer in the field of structured design. Our knowledge and experience ranges from ASIC design with silicon compilers, to the synthesis of programmable devices and the creation of VHDL designs.

We work in the fields of design, training and consultancy and carry out this work both according to the methods that we teach and with the tools that we sell. It is this philosophy that has provided us with the kind of in-depth, expert knowledge of EDA tools and rational working methods that has contributed to our holding a position as one of the market leaders in the structured hardware design industry.

For more information, visit http://www.synplicity.com/products/haps.



Simply Better Results

Synplicity, Inc.

600 West California Avenue Sunnyvale, CA 94086 Phone: (U.S.) +1 408 215-6000

Copyright © 2007 Synplicity, Inc. All rights reserved. Synplicity, the Synplicity logo, and "Simply Better Results" are registered trademarks of Synplicity, Inc.