VHDL Instant

This document aims at giving essential information on VHDL syntax including small examples. It does not intend to provide a complete coverage of the language.

History

v1.0	2002	Initial version.
v1.1	2003	Minor corrections.
v2.0	2004	New 2-column format.
v2.1	2005	Minor corrections.
v3.0	2006	Minor corrections.
v3.1	2008	Minor corrections.

Conventions

bold	reserved word
ID	identifier
[*term*]	optional
{ *term* }	repetition (zero or more)
term {...}	repetition (one or more)
term | *term*	select one in list
( ) , ;	punctuation to be used as is
black	VHDL-87/93/2001 (except noted)
green	added in VHDL-93
red	added in VHDL-2001

Table of contents

Syntax definitions

1. Basic language elements

1.1. Reserved words

abs	**access**	**after**	**alias**	**all**	**and**	
architecture	**array**	**assert**	**attribute**			
begin	**block**	**body**	**buffer**	**bus**		
case	**component**	**configuration**	**constant**			
disconnect	**downto**					
else	**elsif**	**end**	**entity**	**exit**		
file	**for**	**function**				
generate	**generic**	group	**guarded**			
if	impure	**in**	inertial	**inout**	**is**	
label	**library**	**linkage**	literal	**loop**		
map	**mod**					
nand	**new**	**next**	**nor**	**not**	**null**	
of	**on**	**open**	**or**	**others**	**out**	
package	**port**	postponed	**procedure**	**process**	protected	pure
range	**record**	**register**	reject	**rem**	**report**	**return**
rol	ror					
select	**severity**	**signal**	shared	sla	sll	sra
srl	**subtype**					
then	**to**	**transport**	**type**			
unaffected	**units**	**until**	**use**			
variable						
wait	**when**	**while**	**with**			
xnor	**xor**					

1.2. Identifiers

- First character must be alphabetic ('A'..'Z','a'..'z'). Next characters can be alphabetic or numeric ('0'..'9').
- Single underscore ('_') allowed. Not as first or last character.
- Any length. All characters are meaningful.
- Case insensitive.

Legal: X F1234 VHDL1076 VhDl_1076 long_identifier
Legal but same: Alert ALERT alert
Illegal: 74LS00 Q_ A__Z _Q $non no$n

1.3. Special symbols

Operators, delimiters, punctuation.

& ' " # ( ) * + , - . / : ; < = > |
=> ** := /= >= <= <> == --

1.4. Character literals

Any printable character in ISO 8-bit character set.
Compatible with predefined types BIT and CHARACTER.

'A' 'a' ' ' (space) ''' (single quote) '"' (double quote)
CR LF

1.5. String literals

Sequence of characters. Compatible with predefined types STRING and BIT_VECTOR.
"A" is a string, 'A' is a character.

"A string"
"An embedded string: ""here""!"
"this is a string ",
& "broken in two lines."

1.6. Bit string literals

Compatible with predefined type BIT_VECTOR.

(B/b: binary, O/o: octal, X/x: hexadecimal):
"00111010000" B"001110100000" b"001110100000"
B"001_110_100_000" O"1640" o"1640" (= 928)
X"3A0" x"3a0" X"3_A_0" (= 928)

1.7. Numeric literals

Integer numbers. Ex.: 12 0 99 1e6 5E2 123_456
Real numbers. Ex.: 0.0 -4.56 1.076e3 2.5E-2 3.141_593
Numeric literals in bases 2 to 16: *base* # *mantissa* # [*exponent*]
Ex.: 2#0111_1101# 8#175# 16#7D# 16#07d# 10#125#
2#1#e10 (exponent = 2^{10}) 16#4#E2 (exponent = 16^2) 10#1024#e+00 (exponent = 10^0)

1.8. Comments

Any text after two dashes ("--") until the end of the line.

```
...not a comment...  -- comment on one line
-- comment on
-- several
-- lines
```

2. Types and subtypes

Type: set of values with associated operations.
Subtype: type with a constraint on the legal values. Constraint may be static (defined at compile time) or dynamic (defined at elaboration time). A subtype is not a new type. It is derived from a base type and hence compatible with it.

Syntax 1: Type and subtype declaration

type *type-name* [**is** *type-definition*] ;
subtype *subtype-name* **is**
[*resolution-function*] *type-name* [*constraint*] ;

4(5) classes of types:
- ***Scalar types***: integer, real, enumerated, physical.
- ***Composite types***: array, record.
- ***Access types***: pointer to objects of a given type.
- ***File types***: file, external storage.
- ***Protected types***: to manage shared variables.

There is a number of predefined types and subtypes in the package STANDARD in library STD. They can be used without explicit context clause (see).

Type qualification is sometimes required to disambiguate the actual (sub)type of expressions.

Syntax 2: Type qualification

type-name ' (*expression*)

States the type of the expression. See §2.2.

Syntax 3: Type conversion

type-name (*expression*)

Changes the type and the value of the expression. See §6.6.

2.1. Numeric types

Predefined types INTEGER and REAL.
Predefined subtypes NATURAL and POSITIVE.

Supported operators (see §6):
- Arithmetic: + - (unary and binary) * / **abs** **mod** **rem** (only integer types) ** (integer power)
- Relational: = /= < <= > >=

Attribute 'HIGH denotes the highest representable value for the annotated type.

```
type integer is range implementation defined;
   -- minimum range [–2147483647, +2147483647]
type real is range implementation defined;
   -- minimum range [–1.0E38, +1.0E38]
   -- 64 bits minimum according to IEEE Std 754 or 854
subtype natural is integer range 0 to integer'high;
subtype positive is integer range 1 to integer'high;
```

Examples of user-defined numeric types:

```
type byte is range 0 to 255;  -- ascending range
type bit_index is range 31 downto 0;  -- descending range
type signal_level is range -15.0 to 15.0;
subtype probability is real range 0.0 to 1.0;
```

2.2. Enumeration types

Ordered set of distinct identifiers or characters.
Predefined types BIT, BOOLEAN, CHARACTER and SEVERITY_LEVEL.

Supported operators (see §6):
- Relational: = /= < <= > >=
- Logical (only for types BIT and BOOLEAN):
  and **or** **nand** **nor** **xor** **xnor** **not**

```
type bit is ('0', '1');   -- characters
type boolean is (false, true);  -- identifiers
type character is
   (...printable and non printable characters...);
type severity_level is (note, warning, error, failure);
```

Examples of user-defined enumeration types:

```
type states is (idle, init, check, shift, add);
type mixed is (false, 'A', 'B', idle);
```

illustration of ***overloading*** (between types state, mixed and boolean) and mix of characters and identifiers
type qualification may be required to disambiguate identifiers,
e.g., states'(idle) is different from mixed'(idle)

2.3. Physical types

Have a ***base unit***, a range of legal values and a collection of sub-units.
Predefined type TIME.
Predefined subtype DELAY_LENGTH.

Supported operators (see §6):
• Arithmetic: + - (unary and binary) * / **abs**
• Relational: = /= < <= > >=

The base unit (1 fs) defines the minimum ***resolution limit*** for simulation. It is possible to define a ***secondary unit*** multiple of the base unit (e.g., ns) to increase the range of simulated time, but at the price of reduced time accuracy.

A physical literal is a value that defines an integral multiple of some base unit. A space is mandatory between the number and the unit. A physical literal consisting solely of a unit name is equivalent to the integer 1 followed by the unit name.

It is possible to convert a time literal into an integer literal and vice-versa by using division or multiplication.

```
type time is range same range as integer type
   units
      fs;                -- base unit (femtosecond)
      ps = 1000 fs;      -- picosecond
      ns = 1000 ps;      -- nanosecond
      us  = 1000 ns;     -- microsecond
      ms = 1000 us;      -- millisecond
      sec = 1000 ms;     -- second
      min = 60 sec;      -- minute
      hr = 60 min;       -- hour
   end units;
subtype delay_length is time range 0 to integer'high;
```

```
10 ns     25 pf     4.70 kOhm     ns

-- not physical literals:
10ns     25pf
```

```
10 ms / ms = 10          25 * ns = 25 ns
```

2.4. Array types

Indexed collection of values from the same base type. Indexing may be in one dimension or more. Index range may be ***constrained*** (fixed range) or ***unconstrained*** (open range).

Predefined array types BIT_VECTOR, STRING and
Note: The box notation "<>" denotes an unconstrained array.

Supported operators (see §6):
• Relational: = /= < <= > >=
• Logical (*): **and or nand nor xor xnor not**
• Shift and rotate (*): **sll srl sla sra rol ror**
• Concatenation (only one-dimension array types): &
(*) Only one-dimension bit and boolean array types.

```
type bit_vector is array (natural range <>) of bit;
type string is array (natural range <>) of character;
Examples of user-defined array types:
type word is array (31 downto 0) of bit;
   -- vector, descending index range
type address is natural range 0 to 255:
type memory is array (address) of word;
   -- matrix, ascending index range
type truth_table is array (bit, bit) of bit;  -- matrix
```

Access to array elements:
• Let W be an object of type word. W(0), W(8), W(31) refer to one bit of the word. W(31 **downto** 16) refer to one ***slice*** of the word including the 16 bits on the left. The notation W(0 **to** 15) is illegal here as the range of type word is descending.
• Let TT be an object of type truth_table. TT('0', '1') refers to the element at the first line and second column of the matrix. The notation TT('0')('1') is illegal here.
• Let M be an object of type memory. M(12)(15) refers to the bit 15 of address word 12. The notation M(12,15) is illegal here.
• Use of aggregates. Example:

```
type charstr is array (1 to 4) of character;
Access by position:     ('T', 'O', 'T', 'O')
Access by name:         (1 => 'T', 2 => 'O', 3 => 'T', 4 => 'O')
Default access:         (1 | 3 => 'T', others => 'O')
```

2.5. Record types

Collection of values from possibly different base types.

Access to a ***record field***:
• By field selection using a period ('.'). Let inst be an object of type instruction, then the notation inst.opcode refers to the first field of the record.
• Using aggregates.

```
type processor_operation is (op_load, op_add, ...);
type mode is (none, indirect, direct, ...);
type instruction is record
   opcode: processor_operation;
   addrmode: mode;
   op1, op2: INTEGER range 0 to 15;
end record instruction;
```

```
(opcode => op_add, addrmode => none,
 op1 => 2, op2 => 15)          -- named association
(op_load, indirect, 7, 8)      -- positional association
```

2.6. Access types

Pointers, references.

Predefined access type LINE (from package TEXTIO).

The **new** operator allows for allocating memory:

```
new link     -- creates a pointer to a record of type cell
   -- whose pointed value is initially (integer'left, null)
new link'(15, null)   -- explicit initialization
   -- aggregate must be qualified (link')
new link'(1, new link'(2, null))   -- chained allocation
```

The **null** literal denotes a pointer that refers to nothing.

Procedure deallocate frees the memory refered to and resets the pointer value to **null**.

```
type line is access string;
```

Example of user-defined access type:

```
type cell;  -- incomplete type declaration
type link is access cell;
type cell is record
   value: integer;
   succ: link;
end record cell;
```

Let cellptr be an object of type link, then the notation cellptr.**all** refers to the name of the referenced record. The selective notation (e.g., cellptr.value, cellptr.succ) allows accessing individual fields in the record.

```
deallocate (cellptr);  -- cellptr = null
```

2.7. File types

Sequence of values stored in some external device.

Predefined type TEXT (in package TEXTIO).

Predefined operations on files:

```
procedure file_open (file f: file-type;
      external_name: in string;
      open_kind: in file_open_kind := read_mode);
procedure file_open (status: out file_open_status;
      file f: file-type;
      external_name: in string;
      open_kind: in file_open_kind := read_mode);
procedure read (file f: file-type; value: out element-type);
function endfile (file f: file-type) return boolean;
procedure write (file f: file-type; value: in element-type);
procedure file_close (file f: file-type);
```

```
type text is file of string;
```

Examples of user-defined file types:

```
type str_file is file of string;
type nat_file is file of natural;
```

The types file_open_kind and file_open_status are predefined enumerated types (see).

In VHDL-87, procedures file_open and file_close are implicit. Files are opened at their declarations and closed when the simulation leaves the scope in which the file objects are declared.

3. Objects

4 classes of objects: ***constants***, ***variables***, ***signals***, ***files***. An object declaration assigns a type to the object.

3.1. Constants

Hold values that cannot be changed during simulation.

Syntax 4: Constant declaration

constant *constant-name* {, ...} : *(sub)type-name* [:= *expression*] ;

Deferred constant: declaration only allowed in a package declaration (see), value defined in related package body (see §7.4 and §7.5).

```
constant PI: real := 3.141_593;
constant N: natural := 4;
constant index_max: integer := 10*N - 1;
constant delay: delay_length := 5 ns;
-- use of aggregates:
constant null_bv: bit_vector(0 to 15) := (others => '0');
constant TT: truth_table := (others => (others => '0'));
constant instadd1: instruction :=
                  (opcode => op_add, addrmode => none,
                   op1 => 2, op2 => 15);
```

3.2. Variables

Hold values that can be changed during simulation through variable assignment statements (see).

Syntax 5: Variable declaration

[**shared**]
variable *variable-name* {, ...} : *(sub)type -name*
[:= *expression*] ;

The expression defines an initial value. The default initial value of a variable of type T is defined by T'left. If T is a composite type, the rule applies to each element of the type.
The reserved word **shared** allows for declaring shared variables that can be global to an architecture.

```
variable count: natural;
   -- default initial value: count = 0 (= natural'left)
variable isHigh: boolean;
   --default initial value: isHigh = false (= boolean'left)
variable currentState: state := idle; -- explicit initialization
variable memory: bit_matrix(0 to 7, 0 to 1023);
   -- if all elements are of type bit, the default initial value
   -- of each element is '0' (bit'left)
   -- equivalent to (others => (others => '0'))
variable config_word: word := (3 downto 0 => '1',
                               others => '0');
   -- initial value = b"0...01111"
```

3.3. Signals

Hold a logic waveform as a discrete set of time/value pairs. Get their values through signal assignment statements (see).

Syntax 6: Signal declaration

signal *signal-name* {, ...} : *(sub)type -name*
[:= *expression*] ;

The expression defines an initial value. The default initial value of a signal of type T is defined by T'left. If T is a composite type, the rule applies to each element of the type.
A signal cannot be of a file type or of an access type.

```
signal S: bit_vector(15 downto 0);
   -- default initial value: (others => '0')
signal temp: real;
   -- default initial value = real'left = -1.0e38
signal clk: bit := '1';   -- explicit initial value
```

3.4. Files

External storage that depends on operating system.

Syntax 7: File declaration

VHDL-87: **file** *file-name* : *file-(sub)type -name* **is**
[**in** | **out**] *external-name*;
VHDL-93: **file** *file-name* {, ...} : *file-(sub)type-name*
[[**open** *open-kind*] **is** *external-name* ;

The (sub)type shall be a file (sub)type.
The VHDL-87 syntax is not supported by VHDL-93 compilers.
Files are closed at the end of the simulation or by using the file_close procedure (see §2.7).
It is recommended to use the VHDL-93 syntax.

```
type integer_file is file of integer;
file file1: integer_file is "test.dat";
   -- local file opened in read mode (in by default)
file file1: integer_file is out "test.dat";
   -- local file opened in read mode
file file1: integer_file;
   -- local file opened in read mode (read_mode by default)
   -- explicit call to file_open required
file file2: integer_file is "test.dat";
   -- read mode (default) and binding to external file
   -- implicit call to procedure
   --    file_open(F => file2, external_name => "test.dat",
   --              open_kind => read_mode)
file file3: integer_file open write_mode is "test.dat";
   -- write mode and binding to external name
   -- implicit call to procedure
   --    file_open(f => file3, external_name => "test.dat",
   --              open_kind => write_mode)
```

4. Attributes

An attribute represents a characteristic associated to a ***named item***. Predefined attributes denote values, functions, types and ranges. Result type is given between () below.

4.1. Attributes of scalar types

Let T be a scalar type, X an expression type T, S a string:

T'LEFT	(same as T) leftmost value of T.
T'RIGHT	(same as T) rightmost value of T.
T'LOW	(same as T) least value in T.
T'HIGH	(same as T) greatest value in T.
T'IMAGE(X)	(string) textual representation of X of type T.
T'VALUE(S)	(base type of T) value in T represented by string S.

```
type address is integer range 7 downto 0;

address'low = 0      address'high = 7
adress'left  = 7     address'right = 0

time'image(5 ns) = "5000000 fs"  -- f(resolution limit)
time'value("25 ms") = 25_000_000_000_000 fs
```

4.2. Attributes of discrete types

Let T be a enumeration or a physical type, X an expression of type T, N an expression of integer type:

T'POS(X)	(integer) position number of X in T.
T'VAL(N)	(base type of T) value in T at pos. N.
T'SUCC(X)	(base type of T) value at next pos.
T'PRED(X)	(base type of T) value at previous pos.

```
type level is ('U', '0', '1', 'Z');

level'pos('U') = 0       level'pos('1') = 2
level'val(2) = '1'
level'succ('1') = 'Z'    level'pred('0') = 'U'

time'pos(2 ns) = 2_000_000 fs  -- f(base unit)
```

4.3. Attributes of array types and objects

Let A be an array type, object or slice, N an integer expression (by default = 1):

A'LEFT[(N)]	leftmost value in index range of dim. N.
A'RIGHT[(N)]	rightmost val. in index range of dim. N.
A'LOW[(N)]	least value in index range of dim. N.
A'HIGH[(N)]	greatest value in index range of dim. N.
A'RANGE[(N)]	index range of dimension N.
A'REVERSE_RANGE[(N)]	index range of dimension N reversed in direction and bounds.
A'LENGTH[(N)]	length of index in dimension N.
A'ASCENDING[(N)]	(V) true if index range of dimension N is ascending, false otherwise.

```
type word is array (31 downto 0) of bit;
type memory is array (7 downto 0) of word;
variable mem: memory;

mem'low = 0      mem'high = 7     mem'left = 7
mem'right = 0    mem'range = 7 downto 0,
mem'reverse_range = 0 to 7
mem'length = 8
mem'length(2) = 32
mem'range(2) = 31 downto 0
mem'high(2) = 31
mem'ascending(1) = false
```

4.4. Attributes of signals

Let S be an object of class signal, T an expression of type TIME (T ≥ 0 fs, by default T = 0 fs):

S'DELAYED[(T)]	(base type of S) implicit signal with same value as S but delayed by T time units.
S'DRIVING	(boolean) true if the containing process is driving S, false otherwise.
S'DRIVING_VALUE	(base type of S) value contributed by the driver for S in the containing process.
S'EVENT	(boolean) true if an event has occurred on S in the current simulation cycle, false otherwise.
S'LAST_EVENT	(time) time since last event occurred on S, or time'high if no event has yet occurred.
S'LAST_VALUE	(base type of S) value of S before last event occurred on it.
S'STABLE[(T)]	(boolean) implicit signal, true when no event has occurred on S for T time units, false otherwise.
S'TRANSACTION	(bit) implicit signal, changes value in simulation cycle in which a transaction occurs on S.

5. Alias

An alias declaration allows for defining alternate names for named entities.

Syntax 8: Alias declaration

alias *alias-name* [: *(sub)type*] **is** *object-name* ;

```
variable real_number: bit_vector(0 to 31);
alias sign: bit is real_number(0);
alias mantissa: bit_vector(23 downto 0) is
        real_number(8 to 31);
```

6. Expressions and operators

Expression: terms bound by operators. Predefined operators in decreasing order of precedence (parentheses can be used to change the order of evaluation):

**	**abs**	**not**			
*	/	**mod**	**rem**		
+	- *(unary)*				
+	-	&			
sll	**srl**	**sla**	**sra**	**rol**	**ror**
=	/=	<	<=	>	>=
and	**or**	**nand**	**nor**	**xor**	**xnor**

6.1. Logical operators

Legal operands: objects or expressions of scalar types or one-dimensional arrays of types BIT or BOOLEAN;
Short-circuit evaluation: right operand is not evaluated if left operand is '1'/true (**and**, **nand**) or '0'/false (**or**, **nor**). One-dimensional arrays are handled bitwise.

and **or** **nand** **nor** **xor** **xnor**

```
B /= 0 and A/B > 1  -- no division by 0 if B = 0
```

6.2. Relational operators

Legal operands: objects or expressions of the same type. "=" and "/=" shall not have operands of a file type. Other relational operators shall have operands of scalar types or one-dimensional arrays with discrete ranges (integer or enumerated).

=	/=	<	<=
equal	*different*	*less than*	*less or equal than*
>		**>=**	
greater than		*greater or equal than*	

6.3. Shift and rotate operators

Legal operands: left operand is a one-dimensional array of type BIT or BOOLEAN, right operand is an integer expression.

sll	**srl**	**sla**
shift left logical	*shift right logical*	*shift left arithmetic*
sra	**rol**	**ror**
shift right arithmetic	*rotate left*	*rotate right*

```
b"10001010"  sll 3  = b"01010000"
b"10001010"  sll -2 = b"00100010"      -- same as srl 2
b"10001010"  sll 0  = b"10001010"
b"10010111"  srl 2  = b"00100101"
b"10010111"  srl -6 = b"11000000"      -- same as sll 6

b"01001011"  sra 3 = b"00001000"
b"10010111"  sra 3 = b"11110010"
b"00001100"  sla 2 = b"00110000"
b"00010001"  sla 2 = b"01000111"
b"10010011"  rol 1  = b"00100111"
b"10010011"  ror 1 = b"11001001"
```

6.4. Concatenation operator

Legal operands: one-dimensional arrays or elements thereof.
Can be used to emulate shift and rotate operators.

```
constant BY0: byte := b"0000" & b"0000";  -- "00000000"
constant C: bit_vector := BY0 & BY0;  -- C'length = 8
variable B: bit_vector(7 downto 0);
B := '0' & B(B'left downto 1);          -- same as B srl 1
B := B(B'left-1 downto 0) & '0';        -- same as B sll 1
B := B(B'left-1 downto 0) & B(0);       -- same as B sla 1
B := B(B'left) & B(B'left downto 1)     -- same as B sra 1
B := B(B'left-1 downto 0) & B(B'left)   -- same as B rol 1
B := B(B'right) & B(B'left downto 1)    -- same as B ror 1
```

6.5. Aggregates

Association of values to elements in an array or in a record. Positional associations must be done before named associations, if any. The **others** association shall be the last element in the aggregate.

```
(1, 2, 3, 4, 5)
    5-element array or 5-field record; positional association.
(Day => 21, Month => Sep, Year => 1998)
(Month => Sep, Day => 21, Year => 1998)
    3-field record; named association.
('1', '0', '0', '1', others => '0')
    bit vector = "100100000..."; mixed association.
(('X', '0', 'X'), ('0', '0', '0'), ('X', '0', '1'))
("X0X", "000", "X01")
    2-dimension array; positional association.
```

6.6. Arithmetic operators

Legal operands:
- "*", "/": types integer, real or physical.
- **mod**, **rem**: type integer.
- **abs**: numerical types.
- "**": left operand of type integer or real, right operand of type integer (if <0, left operand shall be of type real).
- "+", "-": types integer or real.

```
**      abs     not
*       /       mod     rem
+       -   (unary))
+       -

A**2 + B**2 = (A**2) + (B**2)
4*(A + B) /= 4*A + B
(A + 1) mod B /= A + 1 mod B
PI*R**2 / 2.0 = PI*(R**2) / 2.0

Type conversion:
real(987) = 987.0
integer(4.2) = 4      integer(4.5) = 4      integer(4.8) = 5
```

7. Design units

A design unit is the smallest compilable module stored in a ***design library***. There are five design units: entity declaration, architecture body, configuration declaration, package declaration and package body.
A ***design entity*** is the primary hardware abstraction. It is made of an entity/architecture pair.

7.1. Context clause

Declares one or more design libraries and one or more paths to named entities.

Syntax 9: Context clause

library *library-name* {, ...} ;
  | **use** *library-name* [. *pkg-name*] [. **all**] {, ...} ;

Library names are ***logical names***. Associations to actual physical locations (e.g., Unix directories) must be defined in the VHDL tool.

```
library ieee, cmos_lib;
use ieee.std_logic_1164.all;
use cmos_lib.stdcells_components_pkg.all;
use std.textio.all;
```

Predefined design libraries:
- WORK: only library with read/write access.
- STD: only two packages (STANDARD and TEXTIO).

Implicit context clause for any design unit:

```
library std, work;
use std.standard.all;
```

7.2. Entity declaration

Defines the interface (external view) of a model.

Syntax 10: Entity declaration

[*context-clause*]
entity *entity-name* **is**
    [**generic** (*interface-constant* {, ...}) ;]
    [**port** (*interface-signal* {; ...}) ;]
    { *declaration* }
[**begin**
    { *passive-concurrent-statement* }]
end [**entity**] [*entity-name*] ;

```
entity ent is
    generic (N: positive := 4);
    port (
        signal s1, s2: in bit;
        signal s3: out bit_vector(0 to N-1));
begin
    assert s1 /= s2 report "s1 = s2" severity error;
end entity ent;
```

Legal declarations: (sub)type, constant, signal, shared variable, file, alias, subprogram (declaration and body), use clause.

Legal concurrent statements: concurrent assertion, concurrent procedure call, process. Statements shall be ***passive***, i.e. they may only read object values.

7.3. Architecture body

Defines the internal view of a model.

Syntax 11: Architecture body

[*context-clause*]
architecture *architecture-name* **of** *entity-name* **is**
    { *declaration* }
[**begin**
    { *concurrent-statement* }]
end [**architecture**] [*architecture-name*] ;

Legal declarations: (sub)type, constant, signal, shared variable, file, alias, subprogram (declaration and body), component declaration, use clause.

Legal concurrent statements: concurrent assertion, concurrent procedure call, process, concurrent signal assignment, component instantiation, generate statement.

```
architecture arch of ent is
   constant DELAY: time := 10 ns;
   signal s: bit;
begin
   s <= s1 and s2;
   s3 <= (0 to N-2 => '0') & s after DELAY;
end architecture arch;
```

7.4. Package declaration

Groups declarations that may be used/shared by other design units.

Syntax 12: Package declaration

[*context-clause*]
package *package-name* **is**
    { *declaration* }
end [**package**] [*package-name*] ;

Legal declarations: (sub)type, constant, signal, shared variable, file, alias, subprogram specification, component declaration, use clause.

```
package pkg is
   constant MAX: integer := 10;
   constant MAX_SIZE: natural;  -- deferred constant
   subtype bv10 is bit_vector(MAX-1 downto 0);
   procedure proc (A: in bv10; B: out bv10);
   function func (A, B: in bv10) return bv10;
end package pkg;
```

7.5. Package body

Includes the definitions of declarations made in the related package declaration.

Syntax 13: Package body

[*context-clause*]
package body *package-name* **is**
    { *declaration* }
end [**package body**] [*package-name*] ;

Legal declarations: (sub)type, constant, signal, shared variable, file, alias, subprogram declaration, use clause.

```
package body pkg is
   constant MAX_SIZE: natural := 200;
   procedure proc (A: in bv10; B: out bv10) is
   begin
      B := abs(A);
   end procedure proc;
   function func (A, B: in bv10) return bv10 is
      variable V: bv10;
   begin
      V := A and B;
      return (not(V));
   end function func;
end package body pkg;
```

7.6. Configuration declaration

Defines the bindings between formal component instances and actual design entities (entity/architecture pairs).

Syntax 14: Configuration declaration

[*context-clause*]
configuration *configuration-name* **of** *entity-name* **is**
  for *architecture-name*
    { **for** *component-specification*
      binding-indication **;**
    end for; }
  end for;
end [**configuration**] [*configuration-name*] ;

Syntax 15: Component specification

instance-name {, ...} | **others** | **all** : *component-name*

The component specification indentifies the component instance to be configured; **others** means all instances not yet configured; **all** means all instances of the same component.

Syntax 16: Binding indication

use entity *entity-name* [(*architecture-name*)]
  [**generic map** (*generic-association-list*)]
  [**port map** (*port-association-list*)]

The binding indication defines the mapping between formal and actual generic parameter and port declarations.

```
configuration conf of ent2 is
  for arch2
    for c: comp
      use entity dff(a3)
      port map (clk, d, q, qb);
    end for;
    for all: comp2
      use entity dff(a2)
      generic map (TPROP => 2 ns)
      port map (clk, d, q, qb);
    end for;
  end for;
end configuration conf;
```

8. Interface declarations

Interface declarations are used in entity declarations, component declarations and subprograms.

8.1. Interface constant declaration

An interface constant can be a generic parameter of a subprogram parameter.

Syntax 17: Interface constant declaration

[**constant**] *constant-name* {, ... } : [**in**] *(sub)type-name* [:= *expression*]

8.2. Interface signal declaration

An interface signal can be a port of a subprogram parameter.

Syntax 18: Interface signal declaration

[**signal**] *signal-name* {, ... } : [*mode*]
  (sub)type-name [:= *expression*]

The ***mode*** can be:
- **in**: signal may only be read; only allowed mode for a function parameter; default mode.
- **out**: signal may only be assigned some value.
- **inout**: signal may be read or assigned.
- **buffer**:signal may be read or assigned; the assignment shall be single-source; this mode is not allowed for procedure parameters.

8.3. Interface variable declaration

An interface variable can only be a subprogram parameter.

Syntax 19: Interface variable declaration

[**variable**] *variable-name* {, ... } : [*mode*]
  (sub)type-name [:= *expression*]

The ***mode*** can be:
- **in**: variable may only be read; only allowed mode for a function parameter; default mode.
- **out**: variable may only be assigned some value.
- **inout**: variable may be read or assigned.

8.4. Interface file declaration

An interface file can only be a subprogram parameter.

Syntax 20: Interface file declaration

file *file-name* {, ... } : *file-(sub)type-name*

VHDL-87 declares interface files as interface variables of mode **in** (file is read) or **out** (file is written). This form may not be supported by VHDL-93 compilers anymore.

8.5. Interface association

Defines the mapping between the formal part and the actual part of an interface.

Syntax 21: Interface association

[*formal-part* =>] *actual-part*

The formal part can be a generic name, a port name or a parameter name. ***Named association*** specifies the formal part explicitly. ***Positional association*** only uses the actual part. The corresponding formal part is defined by the position in the interface list.

The actual part can be an expression, a signal name, a variable name, a file name or the reserved word **open**.

8.6. Port association

Possible port associations:

formal	***actual***	***restriction***
signal port	signal | expression | **open**	same type

Unassociated ports (open):
- Signal ports of mode **in**: default value of port declaration is used as initial value
- Signal ports of mode **out**: (only if the port is of a constrained array type) value is ignored
- Signal ports of mode **inout**: value is the value assigned in the design entity

Hierarchical port association:
- Formal port = component instance's port
  Actual port = enclosing entity's port
- Legal hierarchical ***modal*** port associations:

Object class	***Formal***	***Actual***
Signal of mode:	**in**	**in** | **out**
"	**out**	**out** | **inout**
"	**inout**	**inout**

```
entity aoi is
   port (a1, a2, b1, b2: in bit := '1'; z: out bit);
end entity aoi;
-- F <= not ((A and B) or C):
I_AOI1: entity work.aoi(dfl)
   port map (a1 => A, a2 => B, b1 => C, b2 => open,
             z => F); -- or:
   port map (a1 => A, a2 => B, b1 => C, z => F);
   port map (a1 => A, a2 => B, b1 => C, b2 => '1', z => F);
```

8.7. Parameter association

Possible parameter associations:

formal	***actual***
constant	constant literal or expression, variable or signal
variable	variable
signal	signal
file	file

9. Component declaration

A component declaration defines a ***template*** for a design unit to be instantiated.

Syntax 22: Component declaration

component *component-name* [**is**]
   [**generic** (*interface-constant* {; ...}) ;]
   [**port** (*interface-signal* {; ... }) ;]
end component [*component-name*] ;

```
component flipflop is
   generic (Tprop, Tsetup, Thold: time := 0 ns);
   port (clk, d: in bit; q: out bit);
end component flipflop;
```

10. Sequential statements

Sequential statements shall only appear in the body of a process or a subprogram.

10.1. Signal assignment statement

Syntax 23: Signal assignment statement

[*label* :] *signal-name* <= [*delay-mode*] *waveform* ;

Syntax 24: Delay mode

transport | [**reject** *rejection-time*] **inertial**

Default delay mode is **inertial**. The rejection time shall be a non negative value of type time. It shall be less or equal to the time expression of the first waveform element (which defines the value of the inertial delay).

Syntax 25: Waveform

value-expression [**after** *time-expression*] {, ...}

The time expression is equal to 0 ns by default. Time expressions in a waveform shall be specified in increasing time values. The time expression of the first waveform element defines the value of the inertial delay.

```
A <= B after 5 ns, not B after 10 ns;
   -- inertial (rejection) delay of 5 ns

A <= inertial B after 5 ns;
   -- inertial (rejection) delay of 5 ns
A <= reject 2 ns inertial B after 5 ns;
   -- rejection delay of 2 ns

A <= transport B after 5 ns;

S <= A xor B after 5 ns;
S <= "0011";
```

A signal ***never*** takes its new value immediately after the assignment, even when the time expression is 0 ns (***delta delay***).

10.2. Wait statement

Allows for synchronizing processes.

Syntax 26: Wait statement

[*label* :] **wait**
   [**on** *signal-name* {, ...}]
   [**until** *boolean-expression*]
   [**for** *time-expression*] ;

```
wait on S1, S2, S3;
   -- wait on an event on either signal (sensitivity list)

signal clk, enable: bit;
variable stop: boolean;
wait until clk = '1';            -- equivalent
wait on clk until clk = '1';     -- forms
   -- wait until rising edge of clk
wait on enable until clk = '1';
   -- wait on an event on enable and condition
   -- no more sensitive to an event on clk
wait until stop;       -- wait forever (stop is not a signal)

wait for 15 ns;   -- timer

wait until enable = '1' for 20 ns;
wait on enable until enable = '1' for 20 ns;
   -- equivalent forms; cannot wait more than 20 ns

wait;     -- wait forever
```

10.3. Variable assignment statement

Syntax 27: Variable assignment statement

[*label* **:** **]** *variable-name* := *expression* ;

```
variable count: integer;
count := (count + 1)/2;

variable current_state: states;  -- see §2.2
current_state := shift;
```

10.4. If statement

Syntax 28: If statement

[*label* :] **if** *boolean-expression* **then**
   sequential-statement {...}
{ **elsif** *boolean-expression* **then**
   sequential-statement {...} }
[**else**
   sequential-statement {...}]
end if [*label*] ;

```
signal enable, d, q: bit;
if enable = '1' then
   q <= d;
end if;

variable v1, v2, vmax: integer;
signal is_equal: boolean;
if v1 > v2 then
   vmax := v1;
elsif v1 < v2 then
   vmax := v2;
else
   vmax := integer'low;
   is_equal <= true;
end if;
```

10.5. Case statement

Syntax 29: Case statement

[*label* :] **case** *selector-expression* **is**
  when *choices* => *sequential-statement* {...}
  {...}
end case [*label*] ;

Syntax 30: Choices

expression {...} | *discrete-range* {...} | **others**

Choices shall be of the same type as the selector expression. Each choice shall appear only once. The choice **others** shall be the last alternative and shall be the only choice. A **when others** clause is required when all possible choices are not explicitly enumerated.

```
case int is                -- int is of type integer
   when 0 =>               -- single choice
      V := 4;
      S <= '1' after 5 ns;
   when 1 | 2 | 7 =>       -- discrete range
      V := 6;
      S <= '1' after 10 ns;
   when 3 to 6 =>          -- discrete range (or 6 downto 3)
      V := 8;
      S <= '1' after 15 ns;
   when 9 => null;         -- no operation
   when others  =>         -- catches non covered choices
      V := 0;
      S <= '0';
end case;
```

10.6. Loop statement

Syntax 31: Loop statements

[*label* :]
[**while** *boolean-expression*
| **for** *loop-identifier* **in** *discrete-range*] **loop**
  sequential-statement {...}
end loop [*label*] ;

The loop identifier does not need to be declared. It is only visible in the loop body and its value may only be read.

Syntax 32: Next and exit statements

[*label* :] **next** | **exit**
[*loop-label*] [**when** *boolean-expression*] ;

The **next** (resp. **exit**) statement leaves the current iteration, possibly under some condition, and continue to the next (resp. first) iteration of the loop. A label may be used to denote the concerned loop.

```
-- infinite loop:
loop
   wait until clk = '1';
   q <= d after 5 ns;
end loop;

-- while loop:
while i < str'length
   and str(i) /= ' ' loop
   -- skip iteration when true
   next when i = 5;
   i := i + 1;
end loop;
```

```
-- generic loop with exit
L: loop
   exit L when value = 0;
   value := value / 2;
end loop L;

   -- for loop:
L1: for i in 15 downto 0 loop
   L2: for j in 0 to 7 loop
      -- skip to outer loop
      exit L1 when i = j;
      tab(i,j,) := i*j + 5;
   end loop L2;
end loop L1;
```

10.7. Assertion and report statements

Allow for verifying assertions and reporting messages.

Syntax 33: Assertion statement

[*label* :] **assert** *boolean-expression*
  [**report** *string-expression*]
  [**severity** *severity-level*] ;

Severity levels: NOTE (default), WARNING, ERROR, FAILURE.

Syntax 34: Report statement

[*label* :] **report** *string-expression*
  [**severity** *severity-level*] ;

The actual simulation behavior w.r.t. the severity level is defined in the simulator tool.

```
assert initial_value <= max_value severity error;
   -- default message is "Assertion violation"

assert c >= '0' and c <= '9'
   report "Incorrect number " & character'image(c)
   severity warning;

-- equivalent forms:
assert false report "Message always produced";
report "Message always produced";
```

11. Subprograms

11.1. Procedures

A procedure encapsulates sequential statements.

Syntax 35: Procedure declaration

procedure *procedure-name*
[(*interface-parameter-list*)] **is**
{ *declaration* }
begin
sequential-statement {...}
end [**procedure**] [*procedure-name*] ;

Legal interface parameters: constant, variable, signal, file (see §8). Default values shall be only defined for parameters of mode **in** and of class constant (§8.1) or variable (§8.3).
Legal declarations: (sub)type, constant, variable, subprogram declaration. Local declarations are elaborated anew at each procedure call (no state retention).

The procedure call is a concurrent or sequential statement depending on where it is located.

Syntax 36: Procedure call

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

See §8.7 for the syntax of parameter association list.

```
procedure p (f1: in t1; f2: in t2; f3: out t3; f4: in t4 := v4) is
begin
   ...
end procedure p;
-- possible procedure calls ('a' means actual parameter):
p(a1, a2, a3, a4);
p(f1 => a1, f2 => a2, f4 => a4, f3 => a3);
p(a1, a2, f4 => open, f3 => a3);
p(a1, a2, a3);
```

Parameter classes and modes:

Mode	*Default class*
in	constant
out, **inout**	variable

11.2. Functions

A function encapsulates sequential statements and returns a result. A function is a generalization of an expression.

Syntax 37: Function declaration

[**pure** | **impure**]
function *function-name* [(*interface-parameter-list*)]
return *(sub)type-name* **is**
{ *declaration* }
begin
sequential-statement {...}
end [**function**] [*function-name*] ;

Legal interface parameters: constant, signal, file (see §8). Only parameters of mode **in** are allowed. Default values shall be only defined for parameters of class constant (§8.1) or variable (§8.3).
Legal declarations: (sub)type, constant, variable, subprogram declaration. Local declarations are elaborated anew at each procedure call (no state retention).
Wait statements are not allowed in a function body.

A function is said to be ***impure*** if it does refer to variables or signals that are not locally declared. It is said to be ***pure*** otherwise. A function is pure by default.

Syntax 38: Return statement

[*label* :] **return** *expression* ;A function returns its computed value with a return statement.Function call (term in expression)

function-name [(*parameter-association-list*)]

See §8.7 for the syntax of parameter association list.

```
function limit (val,min,max,gain: integer) return integer is
   variable v: integer;
begin
   if val > max then
      v := max;
   elsif val < min then
      v := min;
   else
      v := value;
   end if;
   return gain*v;
end function limit;
```

Function calls:

```
newval := limit(value, min => 10, max => 100, gain => 2);
new_speed := old_speed + scale * limit(error, -10, +10, 2);
```

The predefined function NOW returns the current simulated time:

```
(VHDL-87) function now return time;
(VHDL-93) impure function now return delay_length;
(VHDL-2001) pure function now return delay_length;
```

11.3. Overloading

Overloading is a mechanism that allows for declaring several subprograms having the same names and doing the same kind of operations but acting on parameters of different types.

The procedure or function call defines which version of the subprogram to execute from the number and types of the actual parameters supplied.

It is also possible to overload predefined operators such as "+", "-", **and**, **or**, etc.

Three overloaded procedures to convert a real value, a bit value or a bit_vector value to an integer value:

```
procedure convert (r: in real; result: out integer) is ...
procedure convert (b: in bit; result: out integer) is ...
procedure convert (bv: in bit_vector;
                   result: out integer) is ...
```

Three overloaded functions to compute a minimum value:

```
function min (a, b: in integer) return integer is ...
function min (a: in real; b: in real) return real is ...
function min (a, b: in bit) return bit is ...
```

Overloading of predefined logical operators:

```
type logic4 is ('0', '1', 'X', 'Z');
function "and" (a, b: in logic4) return logic4 is ...
function "or" (a, b: in logic4) return logic4 is ...
```

11.4. Subprograms in a package

Subprograms shall be placed in a package in two parts. The first part is called a ***subprogram specification*** and is placed in the package declaration.

Syntax 39: Subprogram specification

function | **procedure** *subprogram-name*
[(*interface-parameter-list*)] ;

The second part is the complete subprogram declaration ans is placed in the related package body.

Declaration of a 32 bit vector type and its associated operations:

```
package bv32_pkg is
   subtype word32 is bit_vector(31 downto 0);

   procedure add (
      a, b: in word32;
      result: out word32; overflow: out boolean);
   function "<" (a, b: in word32) return boolean;
   ... -- other subprogram specifications
end package bv32_pkg;

package body bv32_pkg is
   procedure add (
      a, b: in word32;
      result: out word32; overflow: out boolean) is
      ... -- local declarations
   begin
      ... -- procedure body
   end procedure add;
   function "<" (a, b: in word32) return boolean is
      ... -- local declarations
   begin
      ... -- function body
   end function "<";
   ... -- other subprogram declarations
end package body bv32_pkg;
```

12. Concurrent statements

Concurrent statements shall only appear in the body of an architecture.

12.1. Process statement

Basic concurrent statement. A process statement defines a portion of code whose statements are executed in the given sequence.

Syntax 40: Process statement

[*label* :] [**postponed**]
process [(*sensitivity-list*)] [**is**]
{ *declaration* }
begin
sequential-statement {...}
end [**postponed**] **process** [*label*] ;

Legal declarations: (sub)type, constant, variable, file, alias, subprogram declaration, use clause. Local declarations are elaborated only once at the beginning of the simulation (state retention between process activations).
Legal statements: all sequential statements.

A process execution is triggered when an event occurs on one of the signals in the ***sensitivity list***. The sensitivity list shall not exist if the process includes wait statements, and reciprocally a process with a sensitivity list shall not include wait statements.

Postponed processes are executed only when all other processes that do not have this attribute have executed their last delta cycle.

signal s1, s2, ...: bit;

Processes with equivalent sensitivity lists:

```
P1a: process (s1, s2, ...) is
begin
   ...-- sequential stmts
end process P1a;
```

```
P1b: process
begin
   ...-- sequential stmts
   wait on s1, s2, ...;
end process P1b;
```

12.2. Concurrent signal assignment statements

Syntax 41: Simple signal assignment statement

[*label* :] [**postponed**]
    signal-name <= [*delay-mode*] *waveform* ;

Syntax 42: Equivalent process form of Syntax 41

[*label* :] [**postponed**]
process (*signal(s)-in-waveform*)
begin
    signal-name <= [*delay-mode*] *waveform* ;
end process [*label*] ;

See §10.1 for delay mode and waveform syntaxes.

```
begin -- of architecture
   ...
   s <= a xor b after 5 ns;
   ...
end architecture;
```

```
begin -- of architecture
   ...
   process (a, b)
   begin
      s <= a xor b after 5 ns;
   end process;
   ...
end architecture;
```

```
lbl: data <= "11" after 2 ns, "01" after 4 ns, "00" after 10 ns;
-- inertial delay of 2 ns; equivalent process:
lbl: process begin
   data <= "11" after 2 ns, "01" after 4 ns, "00" after 10 ns;
   wait;  -- forever
end process lbl;
```

Syntax 43: Conditional signal assignment stmt

[*label* :] [**postponed**]
    signal-name <= [*delay-mode*]
    { *waveform* **when** *boolean-expression* **else** }
    waveform [**when** *boolean-expression*] ;

The delay mode is applied to waveforms in either branch. The reserved word **unaffected** may be used as a waveform element to denote that the target signal remains unchanged for a branch.

```
A <= B after 10 ns when Z = '1' else C after 15 ns;
-- equivalent process:
process (B, C, Z)
begin
   if Z = '1' then
      A <= B after 10 ns;
   else
      A <= C after 15 ns;
   end if;
end process;
```

Syntax 44: Equivalent process form of Syntax 43

[*label* :] [**postponed**]
process (*signal(s)-in-waveform(s)-and-condition(s)*)
begin
    sequential-if-statement ;
end process [*label*] ;

Syntax 45: Selected signal assignment statement

[*label* :] [**postponed**]
with *expression* **select**
    signal-name <= [*delay-mode*]
    waveform **when** *choices* {,...} ;

The delay mode is applied to waveforms in either branch. See §10.5 for the syntax of choices. The reserved word **unaffected** may be used as a waveform element to denote that the target signal remains unchanged for a branch.

```
with muxval select
   S <= A after  5 ns when "00",
        B after 10 ns when "01" | "10",
        C after 15 ns when others;
-- equivalent process:
process (A, B, C, muxval)
begin
   case muxval is
      when "00"         => S <= A after 5 ns;
      when "01" | "10" => S <= B after 10 ns;
      when others       => S <= C after 15 ns;
   end case;
end process;
```

Syntax 46: Equivalent process form of Syntax 45

[*label* :] [**postponed**]
process (*signal(s)-in-expression-and-waveform(s)*)
begin
    sequential-case-statement ;
end process [*label*] ;

12.3. Concurrent procedure call.

Syntax 47: Concurrent procedure call

[*label* :] [**postponed**] *procedure-call* ;

See §11.1 for the syntax of the procedure call.

The ***concurrent procedure call*** is equivalent to a process sensitive to all actual signal parameters of mode **in** or **inout**.

```
procedure proc (
   signal s1, s2: in bit;
   constant C1: integer := 5) is
begin
   ...
end procedure proc;

begin -- of architecture
   ...
   -- concurrent call
   proc(s1, s2, c1);
   ...
end architecture;
```

```
begin -- of architecture
   ...
   -- equivalent process
   process begin
      proc(s1, s2, c1);
      wait on s1, s2;
   end process;
   ...
end architecture;
```

12.4. Concurrent assertion statement

Syntax 48: Assertion statement

[*label* :] **assert** *boolean-expression*
[**report** *string-expression*]
[**severity** *severity-level*] ;

The syntax is exactly the same as for the sequential assertion statement (Syntax 33).

Syntax 49: Equivalent process form of Syntax 48

[*label* :] [**postponed**]
process
begin
sequential-assertion-statement ;
wait [on *signal(s)-in-condition*] ;
end process [*label*] ;

```
entity srff is
   port (s, r: in bit; q, qb: out bit);
begin
   assert not (s = '1' and r = '1')
      report "Both set and reset = '1'"
      severity error;
end entity srff;
```

12.5. Component instantiation statement

Syntax 50: Component instantiation statement

label : *component-indication*
[**generic map** (*generic-association-list*)]
[**port map** (*port-association-list*)] ;

Syntax 51: Component indication

[**component**] *component-name*
| **entity** *entity-name* [(*architecture-name*)]

The form using a component name requires a component declaration (§9) and a configuration (§7.6). A ***default configuration*** is possible when the component declaration has the same signature as an analyzed entity in the library WORK (same generic/port classes, names, modes and types).
The form using the entity/architecture pair is also known as ***direct instantiation*** and does not require a component declaration. If the architecture name is omitted, the ***most recently analyzed*** architecture is used.

```
-- see component declaration in §9
DFF1: component flipflop
   generic map (
      Tprop => 2 ns, Tsetup => 1 ns, Thold => 1 ns)
   port map (
      clk => mclk, d => data, q => data_reg);

-- see entity declaration in
I_E1: entity work.ent(arch)
   port map (s1 => sb1, s2 => sb2, s3 => sbv);
   -- N = 4 (default) and sbv must be a bit_vector(0 to 3)
```

12.6. Generate statement

Encapsulates concurrent statements in a kind of macro instruction that is processed before simulation.

Syntax 52: Generate statement

label :
for *identifier* **in** *discrete-range* | **if** *boolean-expression*
generate
[{ *declaration* }
begin]
{ *concurrent-statement* }
end generate [*label*] ;

The ***iterative form*** (**for** ... **generate**) duplicates the encapsulated statements as many times as defined by the identifier's range. The identifier does not need to be declared; it is only visible in the body of the generate statement and may only be read.
The ***conditional form*** (**if** ... **generate**) includes the encapsulated statements in the model if and only if the condition expression evaluates to true.

```
Gen: for i in 1 to N generate
   signal S: bit_vector(1 to N-1);
   signal S2: bit_vector(1 to N);
begin
   First: if i = 1 generate
      C1: comp port map (CLK, D => A, Q => S(i));
      S2(i) <= S(i) after 10 ns;
   end generate First;
   Int: if i > 1 and i < N generate
      CI: comp port map (CLK, D => S(i-1), Q => S(i));
      S2(i) <= S(i-1) after 10 ns;
   end generate Int;
   Last: if i = N generate
      CN: comp port map (CLK, D => S(i-1), Q => B);
      S2(i) <= S(i-1) after 10 ns;
   end generate Last;
end generate Gen;
```

13. Simulation

Simulation include ***time-domain simulation*** (an initialization phase followed by time-domain simulation cycles).

13.1. Initialization

1) Tc := 0 ns (Tc: current time).
2) Assign initial values to variables and signals.
3) Execute all processes until they suspend.

13.2. Time domain simulation cycle

1) Tn := time of next earliest pending signal transaction or process resumption. If no pending transaction or process resumption, simulation is complete.
2) Tc := Tn.
3) Update signals.
4) Execute all processes sensitive to updated signals until they suspend.
5) If remaining transactions at Tc: go to 3 (delta cycle).
6) Go to 1 (advance time).

14. VHDL predefined packages

14.1. Package STANDARD

Defines predefined VHDL (sub)types and functions. Stored in library STD. Implicit context clause (§7.1).

```
package standard is

    type boolean is (false, true);
    -- implicitly declared operators (return type):
    -- "and", "or", "nand", "nor", "xor", "xnor", "not" (boolean)
    -- "=", "/=", "<", "<=", ">", ">=" (boolean)

    type bit is ('0', '1');
    -- implicitly declared operators (return type):
    -- "and", "or", "nand", "nor", "xor", "xnor", "not" (bit)
    -- "=", "/=", "<", "<=", ">", ">=" (boolean)

    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,
        CAN, EM,  SUB, ESC, FSP, GSP, RSP, USP,
        ' ', '!', '"', '#', '$', '%', '&', ''',
        '(', ')', '*', '+', ',', '-', '.', '/',
        '0', '1', '2', '3', '4', '5', '6', '7',
        '8', '9', ':', ';', '<', '=', '>', '?',
        '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
        'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
        'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
        'X', 'Y', 'Z', '[', '\', ']', '^', '_',
        '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
        'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
        'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
        'x', 'y', 'z', '{', '|', '}', '~', DEL,
        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,
        ' ', '¡', '¢', '£', '¤', '¥', '¦', '§',
        '¨', '©', 'ª', '«', '¬', '-', '®', '¯',
        '°', '±', '²', '³', '´', 'µ', '¶', '·',
        '¸', '¹', 'º', '»', '¼', '½', '¾', '¿',
        'À', 'Á', 'Â', 'Ã', 'Ä', 'Å', 'Æ', 'Ç',
        'È', 'É', 'Ê', 'Ë', 'Ì', 'Í', 'Î', 'Ï',
        'Ð', 'Ñ', 'Ò', 'Ó', 'Ô', 'Õ', 'Ö', '×',
        'Ø', 'Ù', 'Ú', 'Û', 'Ü', 'Ý', 'Þ', 'ß',
        'à', 'á', 'â', 'ã', 'ä', 'å', 'æ', 'ç',
        'è', 'é', 'ê', 'ë', 'ì', 'í', 'î', 'ï',
        'ð', 'ñ', 'ò', 'ó', 'ô', 'õ', 'ö', '÷',
        'ø', 'ù', 'ú', 'û', 'ü', 'ý', 'þ', 'ÿ');
    -- implicitly declared operators (return type):
    -- "=", "/=", "<", "<=", ">", ">=" (boolean)

    type severity_level is
        (note, warning, error, failure);
    -- implicitly declared operators (return type):
    -- "=", "/=", "<", "<=", ">", ">=" (boolean)

    type integer is range implementation defined;
    subtype natural is integer range 0 to integer'high;
    subtype positive is integer range 1 to integer'high;
    -- implicitly declared operators (return type):
    -- "=", "/=", "<", "<=", ">", ">=" (boolean)
    -- "**", "*", "/", "+", "-", "abs", "rem", "mod" (integer)

    type real is range implementation defined;
    -- implicitly declared operators (return type):
    -- "=", "/=", "<", "<=", ">", ">=" (boolean)
    -- "**", "*", "/", "+", "-", "abs" (real)

    type time is range implementation defined
        units
            fs;
            ps  = 1000 fs;
            ns  = 1000 ps;
            us  = 1000 ns;
            ms  = 1000 us;
            sec = 1000 ms;
            min = 60 sec;
            hr  = 60 min;
        end units;
    subtype delay_length is time range 0 to time'high;
    -- implicitly declared operators (return type):
    -- "=", "/=", "<", "<=", ">", ">=" (boolean)
    -- "*", "+", "-", "abs" (time)
    -- "/" ( time or integer)

    pure function now return delay_length;

    type string is array (positive range <>) of character;
    -- implicitly declared operators (return type):
    -- "=", "/=", "<", "<=", ">", ">=" (boolean)
    -- "&" (string)

    type bit_vector is array (naturaL range <>) of  bit;
    -- implicitly declared operators (return type):
    -- "and", "or", "nand", "nor", "xor", "xnor", "not"
    -- (bit_vector)
    -- "sll", "srl", "sla", "sra", "rol", "ror" (bit_vector)
    -- "=", "/=", "<", "<=", ">", ">=" (boolean)
    -- "&" (bit_vector)

    type file_open_kind is
        (read_mode, write_mode, append_mode);
    type file_open_status is
        (open_ok, status_error, name_error, mode_error);
    -- implicitly declared operators (return type):
    -- "=", "/=", "<", "<=", ">", ">=" (boolean)

    attribute foreign: string;

end package standard;
```

14.2. Package TEXTIO

Defines types and associated subprograms for handling text files. Stored in library STD and requires the following use clause:

```
use std.textio.all;

package textio is

    -- type definitions for text I/O:
    type line is access string;
    type text is file of string;
    type side is (right, left);
    subtype width is natural;

    -- standard text files:
    file input: text is in "std_input";
    file output: text is out "std_input";
    file input: text open read_mode  is "std_input";
    file output: text open write_mode is "std_output";

    -- input routines for standard types:
    procedure readline (f: in text; l: out line);
    procedure readline (file f: text; l: out line);

    procedure read (l: inout line; value: out bit);
    procedure read (l: inout line; value: out bit;
                    good: out boolean);
    procedure read (l: inout line; value: out bit_vector);
    procedure read (l: inout line; value: out bit_vector;
                    good: out boolean);
    procedure read (l: inout line; value: out boolean);
    procedure read (l: inout line; value: out boolean;
                    good: out boolean);
    procedure read (l: inout line; value: out character);
    procedure read (l: inout line; value: out character;
                    good: out boolean);
    procedure read (l: inout line; value: out integer);
    procedure read (l: inout line; value: out integer;
                    good: out boolean);
    procedure read (l: inout line; value: out real);
    procedure read (l: inout line; value: out real;
                    good: out boolean);
    procedure read (l: inout line; value: out string);
    procedure read (l: inout line; value: out string;
                    good: out boolean);

    procedure read (l: inout line; value: out time);
    procedure read (l: inout line; value: out time;
                    good: out boolean);

    -- output routines for standard types:
    procedure writeline (f: out text; l: in line);
    procedure WRITELINE (file f: TEXT; l: in 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);
end package textio;
```

VHDL-87 also defines the function ENDLINE:

```
    function endline (l: line) return boolean;
```

VHDL-93 does not support this function anymore. It is replaced by the expression l'length = 0.

15. IEEE standard packages

15.1. Package STD_LOGIC_1164

IEEE standard 1164. Defines a 9-state logic value type and associated logical operators.
Stored in library IEEE and requires the following context clause:

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

- Types and subtypes:

```
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
-- unconstrained array of std_ulogic
type std_ulogic_vector is
    array (natural range <>) of std_ulogic;
-- resolution function
function resolved (s: std_ulogic_vector)
    return std_ulogic;
-- *** industry standard logic type ***
subtype std_logic is resolved std_ulogic;
-- constrained array of std_logic
type std_logic_vector is
    array (natural range <>) of std_logic;
```

Plus subtypes X01, X01Z, UX01 and UX01Z that include only the values listed in the subtype names.
Notation conventions: *b*: bit, *bv*: bit_vector, *su*: std_ulogic, *s*: std_logic, *suv*: std_ulogic_vector, *sv*: std_logic_vector.

- Operators: The package defines overloaded versions of the logical operators **and**, **nand**, **or**, **xor**, **xnor**, and **not** on both scalar and vector types.

- Conversion functions:

function	***argument type → result type***
To_bit(*su*, *xmap*)	*su* → *b*
To_bitvector(*sv*, *xmap*)	*sv* → *bv*

xmap is a bit value ('0' or '1') to use in place of other logic values than '0', '1', 'L' and 'H'.

To_StdULogic	*b* → *su*	
To_StdLogicVector	*bv* → *sv*	*suv* → *sv*
To_StdULogicVector	*bv* → *suv*	*sv* → *suv*

- Strength strippers and type convertors: Convert the logic states not included in the function names to 'X'.

function	***argument type → result type***		
To_X01	*sv* → *sv*	*suv* → *suv*	*su* → X01
	bv → *sv*	*bv* → *suv*	*b* → X01
To_X01Z	*sv* → *sv*	*suv* → *suv*	*su* → X01Z
	bv → *sv*	*bv* → *suv*	*b* → X01Z
To_UX01	*sv* → *sv*	*suv* → *suv*	*su* → UX01
	bv → *sv*	*bv* → *suv*	*b* → UX01

- Edge detection:
  rising_edge(*su*) → boolean
  falling_edge(*su*) → boolean

- Unknown detection:

function	***argument type → result type***
is_X	*suv* | *sv* | *su* → boolean

The 1164 standard defines a ***resolution function*** called RESOLVED that has an unconstrained array of signal drivers as argument and that returns a single resolved logic value:

- If there is a single driver, the function returns the value of that source.
- If the driver array is empty, the function returns a 'Z'.
- Forcing logic values '0', '1', 'X' override weak logic values 'L', 'H', 'W'.
- If two drivers have the same forces but different values, the function returns an 'X' or a 'W' depending on the force.
- The high-impedance logic value 'Z' is dominated by forcing and weak values.
- The resolution of the don't care logic value '-' with any other value gives an 'X'.
- The resolution of the logic value 'U' with any other value gives a 'U', indicating that signals have not been properly initialized.

15.2. Packages NUMERIC_BIT/_STD

IEEE standard 1076.3. Define arithmetic operations on integers represented using vectors of elements of type BIT (NUMERIC_BIT) and STD_LOGIC (NUMERIC_STD). Stored in library IEEE. Require the followwing context clauses:

```
library ieee;
use ieee.numeric_bit.all;
```

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

- Types:

```
type unsigned is array (natural range <>) of bit
type signed is array (natural range <>) of bit
```

- Types:

```
type unsigned is array (natural range <>) of std_logic
type signed is array (natural range <>) of std_logic
```

Notation conventions: *b*: bit, *i*: integer, *n*: natural, *s*: signed, *s(u)l*: std_(u)logic, *s(u)lv*: std_(u)logic_vector, *u*: unsigned. ARG, COUNT, SIZE, etc.: arguments, : L: left argument, R: right argument.

- Arithmetic operators:

operator	*arg/type*		*result type*	*note*
"**abs**"	ARG/*s*		signed*(*ARG'length-1 **downto** 0)	abs value of vector ARG
"-"	ARG/*s*		signed*(*ARG'length-1 **downto** 0)	unary minus operation on vector ARG
"+"	L/*u*	R/*u*	unsigned(max(L'length, R'length)-1 **downto** 0)	vectors may be of different lengths
	L/*s*	R/*s*	signed(max(L'length, R'length)-1 **downto** 0)	*id.*
	L/*u*	R/*n*	unsigned(L'length-1 **downto** 0)	
	L/*n*	R/*u*	unsigned(R'length-1 **downto** 0)	
	L/*s*	R/*i*	unsigned(L'length-1 **downto** 0)	
	L/*i*	R/*s*	unsigned(R'length-1 **downto** 0)	
"-"	L/*u*	R/*u*	unsigned(max(L'length, R'length)-1 **downto** 0)	vectors may be of different lengths
	L/*s*	R/*s*	signed(max(L'length, R'length)-1 **downto** 0)	*id.*
	L/*u*	R/*n*	unsigned(L'length-1 **downto** 0)	
	L/*n*	R/*u*	unsigned(R'length-1 **downto** 0)	
	L/*s*	R/*i*	unsigned(L'length-1 **downto** 0)	
	L/*i*	R/*s*	unsigned(R'length-1 **downto** 0)	
"*"	L/*u*	R/*u*	unsigned(L'length+R'length-1 **downto** 0)	vectors may be of different lengths
	L/*s*	R/*s*	signed(L'length+R'length-1 **downto** 0)	*id.*
	L/*u*	R/*n*	unsigned(2*L'length-1 **downto** 0)	
	L/*n*	R/*u*	unsigned(2*R'length-1 **downto** 0)	
	L/*s*	R/*i*	unsigned(2*L'length-1 **downto** 0)	
	L/*i*	R/*s*	unsigned(2*R'length-1 **downto** 0)	
"/" [a]	L/*u*	R/*u*	unsigned(L'length-1 **downto** 0)	
	L/*s*	R/*s*	signed(L'length-1 **downto** 0)	
	L/*u*	R/*n*	unsigned(L'length-1 **downto** 0)	result truncated to vector'length
	L/*n*	R/*u*	unsigned(R'length-1 **downto** 0)	*id.*
	L/*s*	R/*i*	unsigned(L'length-1 **downto** 0)	*id.*
	L/*i*	R/*s*	unsigned(R'length-1 **downto** 0)	*id.*
"**rem**" [a]	L/*u*	R/*u*	unsigned(R'length-1 **downto** 0)	
	L/*s*	R/*s*	signed(R'length-1 **downto** 0)	
	L/*u*	R/*n*	unsigned(L'length-1 **downto** 0)	result truncated to vector'length
	L/*n*	R/*u*	unsigned(R'length-1 **downto** 0)	*id.*
	L/*s*	R/*i*	unsigned(L'length-1 **downto** 0)	*id.*
	L/*i*	R/*s*	unsigned(R'length-1 **downto** 0)	*id.*
"**mod**" [a]	L/*u*	R/*u*	unsigned(R'length-1 **downto** 0)	
	L/*s*	R/*s*	signed(R'length-1 **downto** 0)	
	L/*u*	R/*n*	unsigned(L'length-1 **downto** 0)	result truncated to vector'length
	L/*n*	R/*u*	unsigned(R'length-1 **downto** 0)	*id.*
	L/*s*	R/*i*	unsigned(L'length-1 **downto** 0)	*id.*
	L/*i*	R/*s*	unsigned(R'length-1 **downto** 0)	*id.*

[a] A severity level of ERROR is issued if 2nd argument is zero.

- Comparison operators: ">", "<", "<=", ">=", "=", "/=". Operands are treated as binary integers.

arg/type		*result type*	*note*
L/*u*	R/*u*	boolean	vectors may be of different lengths
L/*s*	R/*s*	boolean	*id.*
L/*u*	R/*n*	boolean	
L/*n*	R/*u*	boolean	
L/*s*	R/*i*	boolean	
L/*i*	R/*s*	boolean	

- Shift and rotate functions and operators:

function	*arg/type*		*result type*	*note*
shift_left [a]	ARG/*u*	COUNT/*n*	unsigned(ARG'length-1 **downto** 0)	shifts left vector ARG COUNT times.
	ARG/*s*	COUNT/*n*	signed(ARG'length-1 **downto** 0)	
shift_right [a]	ARG/*u*	COUNT/*n*	unsigned(ARG'length-1 **downto** 0)	shifts right vector ARG COUNT times.
	ARG/*s*	COUNT/*n*	signed(ARG'length-1 **downto** 0)	
rotate_left	ARG/*u*	COUNT/*n*	unsigned(ARG'length-1 **downto** 0)	left rotates vector ARG COUNT times.
	ARG/*s*	COUNT/*n*	signed(ARG'length-1 **downto** 0)	
rotate_right	ARG/*u*	COUNT/*n*	unsigned(ARG'length-1 **downto** 0)	right rotates vector ARG COUNT times.
	ARG/*s*	COUNT/*n*	signed(ARG'length-1 **downto** 0)	
operator [b]	***arg/type***		***result type***	***note***
"**sll**"	ARG/u	COUNT/n	unsigned(ARG'length-1 **downto** 0)	shifts left vector ARG COUNT times.
	ARG/s	COUNT/n	signed(ARG'length-1 **downto** 0)	
"**slr**"	ARG/u	COUNT/n	unsigned(ARG'length-1 **downto** 0)	shifts right vector ARG COUNT times.
	ARG/s	COUNT/n	signed(ARG'length-1 **downto** 0)	
"**rol**"	ARG/u	COUNT/n	unsigned(ARG'length-1 **downto** 0)	left rotates vector ARG COUNT times.
	ARG/s	COUNT/n	signed(ARG'length-1 **downto** 0)	
"**ror**")	ARG/u	COUNT/n	unsigned(ARG'length-1 **downto** 0)	right rotates vector ARG COUNT times.
	ARG/s	COUNT/n	signed(ARG'length-1 **downto** 0)	

[a] The type of the first argument determines the kind of shift operation: *u* (logical), *s* (arithmetic). Only shifts in one direction are allowed.

[b] Operators "**sla**" and "**sra**" are not overloaded as the choice of the types of the operands (unsigned or signed) for operators "**sll**" and "**srl**" determine whether a logical or an arithmetic shift should be performed. Operators "**sla**" and "**sra**" are available if the NUMERIC_BIT package is used.

- Resize functions:

function	*arg/type*		*result type*	*note*
resize	ARG/*u*	NEW_SIZE/*n*	unsigned(NEW_SIZE-1 **downto** 0)	increasing size zero-extends to the left truncating keeps the rightmost bits
	ARG/*s*	NEW_SIZE/*n*	signed(NEW_SIZE-1 **downto** 0)	increasing size replicates the sign bit truncating keeps the sign & rightmost bits

- Conversion functions:

function	*arg/type*		*result type*
to_integer	ARG/*u*		natural
	ARG/*s*		integer
to_unsigned	ARG/*n*	SIZE/*n*	unsigned(SIZE-1 **downto** 0)
to_signed	ARG/*i*	SIZE/*n*	signed(SIZE-1 **downto** 0)

- Logical operators: bitwise operations

Operator "**not**"

arg/type	*result type*
L/*u*	unsigned(L'length-1 **downto** 0)
L/*s*	signed(L'length-1 **downto** 0)

- Operators "**and**", "**or**", "**nand**", "**nor**", "**xor**", "**xnor**"

arg/type		*result type*
L/*u*	R/*u*	unsigned(L'length-1 **downto** 0)
L/*s*	R/*s*	signed(L'length-1 **downto** 0)

- Edge detection functions: rising_edge, falling_edge. Already defined in package STD_LOGIC_1164 (§15.1). Provided in package NUMERIC_BIT with a single signal argument of type BIT and returning the boolean value TRUE when there is an event on the signal and its new value is '1' (rising) or '0' (falling).

- Match function: std_match. Bitwise compare, treats '-' value (don't care) as matching any other std_ulogic value. Returns TRUE when both arguments are either '0' and 'L' or '1' and 'H'.

arg/type		*result type*	*arg/type*		*result type*	*arg/type*		*result type*
L/*sul*	R/*sul*	boolean	L/*sulv*	R/*sulv*	boolean	L/*u*	R/*u*	boolean
			L/*slv*	R/*slv*	boolean	L/*s*	R/*s*	boolean

- Translation function: to_01. Only defined in package NUMERIC_STD. Bitwise translation '1'|'H' → '1', '0'|'L' → '0'.

arg/type		*result type*	*note*
S/*u*	XMAP/*sl*	unsigned(S'range)	XMAP is optional (default = '0')
S/*s*	XMAP/*sl*	signed(S'range)	*id*.

15.3. Package MATH_REAL

IEEE standard 1076.2. Defines constants and mathematical functions on real numbers.
Stored in library IEEE and requires the following context clause:

```
library ieee;
use ieee.math_real.all;
```

• Constants:

name	*value*	*name*	*value*
math_e	*e*	math_log_of_2	ln 2
math_1_over_e	1/*e*	math_log_of_10	ln 10
math_pi	π	math_log2_of_e	$\log_2 e$
math_2_pi	2π	math_log10_of_e	$\log_{10} e$
math_1_over_pi	$1/\pi$	math_sqrt_2	$\sqrt{2}$
math_pi_over_2	$\pi/2$	math_1_over_sqrt_2	$1/\sqrt{2}$
math_pi_over_3	$\pi/3$	math_sqrt_pi	$\sqrt{\pi}$
math_pi_over_4	$\pi/4$	math_deg_to_rad	$2\pi/360$
math_3_pi_over_2	$3\pi/2$	math_rad_to_deg	$360/2\pi$

• Procedure:

```
procedure uniform (
    variable seed1, seed2: inout positive;
    variable x: out real);
```

Generates successive values in the range [0.0, 1.0[in a pseudo-random number sequence with a uniform distribution.

• Functions (*x*, *y*: real numbers; *n*: integer number; return real numbers):

name	*meaning*
sign(*x*)	sign of *x* (-1.0, 0.0 or +1.0)
ceil(*x*)	least integer $\geq x$
floor(*x*)	greatest integer $\leq x$
round(*x*)	*x* rounded to the nearest integer value (ties rounded away from 0.0)
trunc(*x*)	*x* truncated toward 0.0
"mod"(*x*, *y*)	floating-point modulus of *x*/*y*
realmax(*x*, *y*)	greater of *x* and *y*
realmin(*x*, *y*)	lesser of *x* and *y*

name	*meaning*	*name*	*meaning*
sqrt(*x*)	square root	log(*x*)	ln *x*
cbrt(*x*)	cube root	log2(*x*)	$\log_2 x$
"**"(*n*, *y*)	n^y	log10(*x*)	$\log_{10} x$
"**"(*x*, *y*)	x^y	log(*x*, *y*)	$\log_y x$
exp(*x*)	e^x		
sin(*x*)	sin *x* (1)	arcsin(*x*)	arcsin *x*
cos(*x*)	cos *x* (1)	arccos(*x*)	arccos *x*
tan(*x*)	tan *x* (1)	arctan(*x*)	arctan *x*
	(1) *x* in radians		
arctan(*y*, *x*)	arctan of point (*x*, *y*)		
sinh(*x*)	sinh *x*	arcsinh(*x*)	arcsinh *x*
cosh(*x*)	cosh *x*	arccosh(*x*)	arccosh *x*
tanh(*x*)	tanh *x*	arctanh(*x*)	arctanh *x*

15.4. Package MATH_COMPLEX

IEEE standard 1076.2. Defines types, constants and mathematical functions on complex numbers.
Stored in library IEEE and requires the following context clause:

```
library ieee;
use ieee.math_complex.all;
```

• Types:

```
type complex is record
    re: real; im:real;      -- real part, 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
end record;                 -- –math_pi is illegal
```

• Constants:

name	*value*	*name*	*value*
math_cbase_1	1.0 + *j*0.0	math_czero	0.0 + *j*0.0
math_cbase_j	0.0 + *j*0.0		

• Operators (*r*: real; *pr*: positive_real; *c*: complex; *cp*: complex_polar; → result type):

"="(*p*, *p*) → boolean	"abs"(*c* | *cp*) → *pr*
"/="(*p*, *p*) → boolean	"–"(*c*) → *c* "–"(*cp*) → *cp*
"+", "–", "*", "/"(*c*, *c*) → *c*	"+", "–", "*", "/"(*cp*, *cp*) → *cp*
"+", "–", "*", "/"(*r*, *c*) → *c*	"+", "–", "*", "/"(*r*, *cp*) → *cp*
"+", "–", "*", "/"(*c*, *r*) → *c*	"+", "–", "*", "/"(*cp*, *r*) → *cp*

• Functions(*x*, *y*: real numbers; *z*: complex or complex_polar number; *pv*: principal_value):

name →*result type*	*meaning*
cmplx(*x*, *y*) → *c*	x + *jy*
get_principal_value(*x*) → *pv*	$x + 2k\pi$, *k*: $-\pi <$ result $\leq \pi$ *x* in radians
complex_to_polar(*c*) → *cp*	*c* in polar form
polar_to_complex(*cp*) → *c*	*cp* in Cartesian form
arg(*z*) → same as *z*	angle of *z* in radians
conj(*z*) → same as *z*	complex conjugate of *z*
sqrt(*z*) → same as *z*	square root of *z*
exp(*z*) → same as *z*	e^z
log(*z*) → same as *z*	ln *z*
log2(*z*) → same as *z*	$\log_2 z$
log10(*z*) → same as *z*	$\log_{10} z$
log(*z*, *y*) → same as *z*	$\log_y z$
sin(*z*) → same as *z*	sin *z*
cos(*z*) → same as *z*	cos *z*
sinh(*z*) → same as *z*	sinh *z*
cosh(*z*) → same as *z*	cosh *z*