VHDL Quick Reference
Draft Revision 1.0

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131

1 Predefined Language Environment

1.0 Lexical Elements

General

VHDL is a case insensitive language. It is also a form free language.

VHDL Text

VHDL text contains a sequence of reserved words, identifiers, abstract literals (used to represent numbers and physical entities such as time), character literals, bitstring literals, string literals and comments. These lexical elements are separated from each other using a set of delimiters or filed separator.

Comments

Any text sequence beginning with two hyphens (--) and terminating at the end of the line is treated as a comment. VHDL does not support comment sequences that span across multiple lines, although a comment can start anywhere on the line. All literals and delimiters loose their special meanings within the comment.

Delimiters

In addition to a space, tab and end-of-line character, lexical elements of the VHDL file can also be separated with the following set of delimiters.

Simple delimiters: & ' ( ) * + , - . / : ; < = > | [ ]
Compound delimiters: => ** := /= >= <= <>

Identifiers

A sequence of characters beginning with a letter followed by one or more letter or digit denotes an identifier[1]. Identifiers are used to name objects, ports, blocks, configurations, statements, processes, blocks etc in the VHDL design.

SIGNAL **carry0, sum0** : std_logic;
ENTITY **dff** IS PORT (**D**,**Q**,**CLK**,**RST** : IN std_logic; **Q** : OUT std_logic)
END dff;

[1] VHDL LRM makes references to an extended identifier which is a token of graphic character enclosed in \ and \. No synthesis and simulation software tool seem to be supporting extended identifier.

Abstract Literals:

Abstract literals are used to define numbers and physical types. Numbers can be real or integers expressed in a decimal system or in other numeric system. VHDL treats a numeric literal containing dot (.) as a real number.

Note: Underscores can be inserted anywhere except in the beginning of the numeric literal without changing its value.

Decimal Literals: A base of 10 is assumed. Decimal literals can denote a real number or an integer.

```
10          10_000          1009E+10        -- integer literals
10.7        17.5_009E-1_10  67.7E-5          -- real literals
```

Based Literals: Based literals are identical to decimal literals except the base is specified explicitly. Base literals contain base, mantissa and exponent parts. These three parts can be separated either by a pound character (#) or colon character (:).

```
2#101000    2#1010_100      2#1010#E+10     -- integer literals in binary
F#10EFA     F#69D_FFF       F#11CAE#EF      -- integer literals in hexadecimal
F:10E.FA    F:69.D_FFF      F:11C.AE:EF     -- real literals in hexadecimal
```

Character Literals:

A character literal is a graphic character (any character that can be seen by the eye on computer's screen) enclosed in single quotes (').

constant backslash : **character** := '\';

String Literals:

A string literal is denoted by a sequence of graphic characters enclosed in either double quotes(") or in a percentage character (%).

constant allVowells : string := **"aeiou"** ; alternatively,
constant allVowells : string := **%aeiou%;**

Bitstring Literals:

Bitstring literals are used to denote base qualified numeric values of the integers in a string format. Characters B, X and O are used to denote binary, hexadecimal and octal bases.

```
constant const1   :   bit_vector   := B"1101110";
constant const2   :   bit_vector   := X"EE";
constant const3   :   bit_vector   := O%777%;
```

1.2 Reserved Words

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

1.3 Predefined Data-types

A list of predefined data types supported in VHDL is given below. These data types are defined in package *standard* and in package *textio*. Note that VHDL supports a mechanism for creating user defined data types using TYPE and SUBTYPE constructs. Consequently, users can create their own data-types and use them in their programs. In any case, all VHDL programs are guaranteed to have access to data types defined in its standard environment.

```
-- Enumerated
type boolean is (FALSE,TRUE);
type bit is ('0','1');
type character is (all graphic characters are included here);
type severity_level is (NOTE, WARNING, ERROR, FAILURE);
type universal_integer is range implementation_specific;
type universal_real is range implementation_specific;

-- numeric
type integer is range implementation_specific;
type real is range implementation_specific;
subtype natural is integer range 0 to integer'high;
subtype positive is integer range 1 to integer'high;

-- physical
type time is range implementation_specific
        units

          fs;                   -- 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 time'high;

-- Arrays
type string is array (positive range <>) of character;
type bit_vector is array(natural range <>) of bit;

-- File types
type file_open_kind is (
        READ_MODE,              -- Resulting access mode is read-only.
        WRITE_MODE,             -- Resulting access mode is write-only.
        APPEND_MODE);           -- Resulting access mode is write-only; information
                                -- is appended to the end of the existing file.
type file_open_status is (
        OPEN_OK,                -- File open was successful.
        STATUS_ERROR,           -- File object was already open.
        NAME_ERROR,             -- External file not found or inaccessible.
        MODE_ERROR);            -- Could not open file with requested access mode.

--  Type definitions for text I/O:
        type line is access string;     -- A LINE is a pointer to a STRING value.
        type text is file of string;    -- A file of variable-length ASCII records.
        type side is (right, left);     -- For justifying output data within fields.
        subtype width is natural;       -- For specifying widths of output fields.
```

1.4 Predefined Operators

The operators are listed in their precedence table given below. The operators are grouped with increasing level of precedence. If the expression contains multiple operations having same precedence, the operators are evaluated from left to right.

logical operators(B)	and	or	nand	nor	xor	xnor
relational operators(B)	=	/=	<	<=	>	>=
shift operators(B)	sll	srl	sla	sra	rol	ror
addition operators(B)	+	-	&			
unary sign operators(U)	+	-				
multiplication operator(B)	*	/	mod	rem		
miscellaneous operators	** (B)	abs(U)	not(U)			

Legend: **B** : Binary operator
U : Unary operator
Note: operator *not* is a logical operator. However, its precedence is higher than other operators in the logical group.

The detailed explanation of the shift operators is given in the table below.

Operator	Operation	Left operand type	Right operand type	Result type
sll	Shift left logical	Any one-dimensional array type whose element type is BIT or BOOLEAN	INTEGER	Same as left
srl	Shift right logical	Any one-dimensional array type whose element type is BIT or BOOLEAN	INTEGER	Same as left
sla	Shift left arithmetic	Any one-dimensional array type whose element type is BIT or BOOLEAN	INTEGER	Same as left
sra	Shift right arithmetic	Any one-dimensional array type whose element type is BIT or BOOLEAN	INTEGER	Same as left
rol	Rotate left logical	Any one-dimensional array type whose element type is BIT or BOOLEAN	INTEGER	Same as left
ror	Rotate right logical	Any one-dimensional array type whose element type is BIT or BOOLEAN	INTEGER	Same as left
&	Concatenation	Any array type	Same array type	Same as array
		Any array type	The element type	Same as array
		The element type	Any array type	Same as array
		The element type	The element type	Same as array

1.5 Operator functions

The behaviour of the operators listed in section 1.4 is defined for in package *standard* . Note that VHDL allows operator overloading, where the meaning of the operators can be re-defined for different types of operands. Consequently, there can be more than one-operator functions corresponding to a single operator. Users are requested to refer to VHDL LRM for additional details.

Two tables below list all operators and the types of the arguments for which they are defined.

	Logical	Relational	Addition	Sign	Multiplication	Miscellaneous
boolean	all	all				not
bit	all	all				not
character		all				
severity_level		all				
universal_integer		all	+ , -	all	all	abs
universal_real		all	+,-	all	*,/	abs
integer		all	+,-	all	all	abs,**
real		all	+,-	all	*,/	abs
natural		all				
positive		all	+,-	all	all	abs,**
time		all	+,-	all		abs
delay_length		all	+,-	all	*,/	abs
string		all	&			
bit_vector	all	all				
file_open_kind		all				
file_open_status		all				

Table: Operators functions defined on built-in data types. These operators take arguments of the same types and return the value of the same type except for relational operator, which returns type boolean.

Operator	Left Operand	Right Operand	Return Type
*	universal_real	universal_integer	universal_real
*	universal_integer	universal_real	universal_real
/	universal_real	universal_integer	universal_real
**	universal_integer	integer	universal_integer
**	universal_real	integer	universal_real
**	real	integer	real
*	integer	time	time
*	time	integer	time
*	real	time	time
*	time	real	time
/	time	integer	time
/	time	real	time
/	time	time	time
&	string	character	string
&	character	string	string
&	character	character	string
all shift operators	bit_vector	integer	bit_vector

Table: Overloaded operators' functions defined for built-in data types. These operators take arguments of the different types.

1.6 Predefined attributes

For details on the restrictions used on using the attributes, please refer to VHDL LRM.

Attribute Name	*Kind*	*Prefix*	*Parameter*	*Result Type*	*Result*
T'BASE	Type	Any type or subtype T			The base type of T
T'LEFT	Value	Any scalar type or subtype T		Same type as T	The left bound of T
T'RIGHT	Value	Any scalar type or subtype T		Same type as T	The right bound of T
T'HIGH	Value	Any scalar type or subtype T		Same type as T	The upper bound of T
T'LOW	Value	Any scalar type or subtype T		Same type as T	The lower bound of T
T'ASCENDING	Value	Any scalar type or subtype T		Type Boolean	TRUE if T is defined with an ascending range.FALSE otherwise
T'IMAGE(X)	Function	Any scalar type or subtype T	An expression whose type is the base type of T	Type String	The string representation of the parameter value, without leading or trailing whitespace
T'VALUE(X)	Function	Any scalar type or subtype T	An expression of type String	The base type of T	The value of T whose string representation is given by the parameter
T'POS(X)	Function	Any discrete or physical type or subtype T	An expression whose type is the base type of T	universal_integer	The position number of the value of the parameter
T'VAL(X)	Function	Any discrete or physical type or subtype T	An expression of any integer type	The base type of T	The value whose position number is the universal_integer value corresponding to X
T'SUCC(X)	Function	Any discrete or physical type or subtype T	An expression whose type is the base type of T	The base type of T	The value whose position number is one greater than that of the parameter
T'PRED(X)	Function	Any discrete or physical type or subtype T	An expression whose type is the base type of T	The base type of T	The value whose position number is one less than that of the parameter
T'LEFTOF(X)	Function	Any discrete or physical type or subtype T	An expression whose type is the base type of T	The base type of T	The value that is to the left of the parameter in the range of T
T'RIGHTOF(X)	Function	Any discrete or physical type or subtype T	An expression whose type is the base type of T	The base type of T	The value that is to the right of the parameter in the range of T
T'RIGHTOF(X)	Function	Any discrete or physical type or subtype T	An expression whose type is the base type of T	The base type of T	The value that is to the right of the parameter in the range of T
A'LEFT[(N)]	Function	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes a constrained array subtype	A locally static expression of type universal_integer, the value of which must not exceed the dimensionality of A(If omitted, it defaults to 1)	Type of the left bound of the Nth index range of A	Left bound of the Nth index range of A
A'RIGHT[(N)]	Function	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes a constrained array subtype	A locally static expression of type universal_integer, the value of which must not exceed the dimensionality of A, if omitted, it defaults to 1	Type of the Nth index range of A	Right bound of the Nth index range of A

A'HIGH[(N)]	Function	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes a constrained array subtype	A locally static expression of type universal_integer, the value of which must not exceed the dimensionality of A, if omitted, it defaults to 1	Type of the Nth index range of A	Upper bound of the Nth index range of A
A'LOW[(N)]	Function	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes a constrained array subtype	A locally static expression of type universal_integer, the value of which must not exceed the dimensionality of A. If omitted, it defaults to 1	Type of the Nth index range of A	Lower bound of the Nth index range of A
A'RANGE[(N)]	Range	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes a constrained array subtype	A locally static expression of type universal_integer, the value of which must not exceed the dimensionality of A, If omitted, it defaults to 1	The type of the Nth index range of A	The range A'LEFT(N) to A'RIGHT(N) if the Nth index range of A is ascending. The range A'LEFT(N) downto A'RIGHT(N) if the Nth index range of A is descending
A'REVERSE_RANGE[(N)]	Range	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes a constrained array subtype	A locally static expression of type universal_integer, the value of which must not exceed the dimensionality of A, if omitted, it defaults to 1	The type of the Nth index range of A	The range A'RIGHT(N) downto A'LEFT(N) if the Nth index range of A is ascending. The range A'RIGHT(N) to A'LEFT(N) if the Nth index range of A is descending
A'LENGTH[(N)]	Value	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes a constrained array subtype	A locally static expression of type universal_integer, the value of which must not exceed the dimensionality of A, if omitted, it defaults to 1	universal_integer	Number of values in the Nth index range. i.e, if the Nth index range of A is a null range, then the result is 0. Otherwise, the result is the value of T'POS(A'HIGH(N)) - T'POS(A'LOW(N)) + 1, where T is the subtype of the Nth index of A
A'ASCENDING [(N)]	Value	Any prefix A that is appropriate for an array object, or an alias thereof, or that denotes a constrained array subtype	A locally static expression of type universal integer, the value of which must be greater than zero and must not exceed the dimensionality of A, if omitted, it defaults to 1	Boolean	TRUE if the Nth index range of A is defined with an ascending range. FALSE otherwise.
S'DELAYED[(T)]	Signal	Any signal denoted by the static signal name S	A static expression of type TIME that evaluates to a nonnegative value, if omitted, it defaults to 0 ns	The base type of S	A signal equivalent to signal S delayed T units of time
S'STABLE [(T)]	Signal	Any signal denoted by the static signal name S	A static expression of type TIME that evaluates to a nonnegative value, if omitted, it defaults to 0 ns	Type Boolean	A signal that has the value TRUE when an event has not occurred on signal S for T units of time, and the value FALSE otherwise
S'QUIET[(T)]	Signal	Any signal denoted by the static signal name S	A static expression of type TIME that evaluates to a nonnegative value, if omitted, it defaults to 0 ns	Type Boolean	A signal that has the value TRUE when the signal has been quiet for T units of time, and the value FALSE otherwise

S'TRANSACTION	Signal	Any signal denoted by the static signal name S	Type Bit	A signal whose value toggles to the inverse of its previous value in each simulation cycle in which signal S becomes active
S'EVENT	Function	Any signal denoted by the static signal name S	Boolean	A value that indicates whether an event has just occurred on. For a scalar signal S, S'EVENT returns the value TRUE if an event has occurred on S during the current simulation cycle. Otherwise, it returns the value FALSE. For a composite signal S, S'EVENT returns TRUE if an event has occurred on any scalar sub-element of S during the current simulation cycle. Otherwise, it returns FALSE
S'ACTIVE	Function	Any signal denoted by the static signal name S	Boolean	For a scalar signal S, S'ACTIVE returns the value TRUE if signal S is active during the current simulation cycle. Otherwise, it returns the value FALSE. For a composite signal S, S'ACTIVE returns TRUE if any scalar sub-element of S is active during the current simulation cycle. Otherwise, it returns FALSE
S'LAST_EVENT	Function	Any signal denoted by the static signal name S	Time	The amount of time that has elapsed since the last event For a signal S, S'LAST_EVENT returns the smallest value T of type TIME such that S'EVENT = True during any simulation cycle at time NOW - T, if such value exists. Otherwise, it returns TIME'HIGH
S'LAST_ACTIVE	Function	Any signal denoted by the static signal name S	Time	The amount of time that has elapsed since the last time at. For a signal S, S'LAST_ACTIVE returns the smallest value T of type TIME such that S'ACTIVE = True during any simulation cycle at time NOW - T, if such value exists. Otherwise, it returns TIME'HIGH
S'LAST_VALUE	Function	Any signal denoted by the static signal name S	The base type of S	The previous value of S, immediately before the last change of S
S'DRIVING	Function	Any signal denoted by the static signal name S	Boolean	If the prefix denotes a scalar signal, the result is False if the current value of the driver for S in the current process is determined by the null transaction. True otherwise If the prefix denotes a composite signal, the result is True if and only if R'DRIVING is True for every scalar sub-element R of S; False otherwise. If the prefix denotes a null slice of a signal, the result is True

S'DRIVING_VALUE	Function	Any signal denoted by the static signal name S		The base type of S	If S is a scalar signal S, the result is the current value of the driver for S in the current process. If S is a composite signal, the result is the aggregate of the values of R'DRIVING_VALUE for each element R of S. If S is a null slice, the result is a null slice
E'SIMPLE_NAME	Value	Any named entity		Type String	The simple name, character literal, or operator symbol of the named entity, without leading or trailing white space or quotation marks but with apostrophes (in the case of a character literal) and both a leading and trailing reverse solidus (backslash) (in the case of an extended identifier). In the case of a simple name or operator symbol, the characters are converted to their lowercase equivalents
E'INSTANCE_NAME	Value	Any named entity other than the local ports and generics of a component declaration		String	Please refer to LRM for more details.
E'PATH_NAME	Value	Any named entity other than the local ports and generics of a component declaration		String	A string describing the hierarchical path starting at the root of the design hierarchy and descending to the named entity, excluding the name of instantiated design entities

1.8 Standard I/O Functions

The procedure definitions for data input and output routines are defined in package *textio*. They are summarized below.

```
-- This procedure reads data from a file into a line
procedure readline(file F:  text;L: out line)

-- This procedure reads a data from a line into value
procedure read (L: inout line; value : in <anyBuitdInDataType>)

-- This procedure reads a data from a line into value and indicates its conformance to
-- the data type
procedure read (L: inout line; value : in <anyBuitdInDataType>; good:boolean)

-- This procedure writes a line from a file into a line
procedure writeline(file F: text;L: in line)

-- This procedure writes a value on to a line with the specified justification and inserts
-- field width indicated by width

procedure write (L: inout line; value : in <anyBuitdInDataType>; justified: in side :=
right; field: in width :=0)

-- a functions that tests the end of file condition
function endfile (file  F: text) return boolean;
```

2 Standard IEEE Environment

2.1 Package std_logic_1164

2.1.1 std_logic and std_logic_vector: An industry standard logic system

The *bit* and *bit_vector* data types defined in the standard environment are inadequate to handle simulation of designs containing tri state resources and addressing initialization and X propagation conditions. Consequently, industry standard data types std_logic and std_logic_vector are defined in package std_logic_1164. Notice that std_logic is a *resolved* data type of the std_ulogic. std_ulogic is unresolved 9 valued logic system.

```
-- Unresolved logic state system, std_ulogic

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 for use with the resolution function

TYPE std_ulogic_vector IS ARRAY ( NATURAL RANGE <> ) OF std_ulogic;

-- resolution function (Look for details on the resolution function below)

FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic;

--  industry standard logic type std_logic and std_logic_vector

SUBTYPE std_logic IS resolved std_ulogic;
TYPE std_logic_vector IS ARRAY ( NATURAL RANGE <>) OF std_logic;
```

2.1.2 Resolution Function For std_ulogic

The following two-dimensional table is used to determine effective value of a net having std_ulogic type is driven by multiple drivers.

As an example, if a net is driven by three drivers having transactions 1,L and Z, the effective value of the net will be calculated as follows.

Two drivers forcing signal 1 and L will yield a value of 1. This value is resolved against Z to yield 1.

	U	**X**	**0**	**1**	**Z**	**W**	**L**	**H**	**-**
U	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'
X	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
0	'U'	'X'	'0'	'X'	'0'	'0'	'0'	'0'	'X'
1	'U'	'X'	'X'	'1'	'1'	'1'	'1'	'1'	'X'
Z	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'X'
W	'U'	'X'	'0'	'1'	'W'	'W'	'W'	'W'	'X'
L	'U'	'X'	'0'	'1'	'L'	'W'	'L'	'W'	'X'
H	'U'	'X'	'0'	'1'	'H'	'W'	'W'	'H'	'X'
-	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'

Table: std_ulogic resolution table

2.1.2 Other useful types defined in package std_logic_1164

In addition to *std_logic* and *std_logic_vector* types, std_logic_1164 also defines other useful data-types and conversion functions between them. These types are described below.

```
subtype X01     is resolved std_ulogic RANGE 'X' TO '1';   -- ('X','0','1')
subtype X01Z    is resolved std_ulogic RANGE 'X' TO 'Z';   -- ('X','0','1','Z')
subtype UX01    is resolved std_ulogic RANGE 'U' TO '1';   -- ('U','X','0','1')
subtype UX01Z   is resolved std_ulogic RANGE 'U' TO 'Z';   -- ('U','X','0','1','Z')
```

For details on the conversion functions among these and other data types, readers are requested to refer to std_logic_1164 package listing attached in the appendix.

2.2 Package std_logic_arith

Package std_logic_arith defines two additional data types *signed* and *unsigned*. It also redefines operator definitions and provides conversion functions for these data types. Note that all operators defined in this package take signed and unsigned numbers as their arguments. Operators that take standard_logic_vectors as their arguments are redefined in std_logic_signed and std_logic_unsigned packages.

type **unsigned** is array (NATURAL range <>) of std_logic;
type **signed** is array (NATURAL range <>) of std_logic;

Left Operand	Right Operand	Result	Operations
UNSIGNED	UNSIGNED	UNSIGNED	+, -,*, SHL,SHR
SIGNED	SIGNED	SIGNED	+, -,*
UNSIGNED	SIGNED	SIGNED	+, -,*
SIGNED	UNSIGNED	SIGNED	+, -,*,SHL,SHR
UNSIGNED	INTEGER	UNSIGNED	+, -
INTEGER	UNSIGNED	UNSIGNED	+, -
SIGNED	INTEGER	SIGNED	+, -
INTEGER	SIGNED	SIGNED	+, -
UNSIGNED	STD_ULOGIC	UNSIGNED	+, -
STD_ULOGIC	UNSIGNED	UNSIGNED	+, -
SIGNED	STD_ULOGIC	SIGNED	+, -
STD_ULOGIC	SIGNED	SIGNED	+, -
UNSIGNED	UNSIGNED	STD_LOGIC_VECTOR	+, -,*
SIGNED	SIGNED	STD_LOGIC_VECTOR	+, -,*
UNSIGNED	SIGNED	STD_LOGIC_VECTOR	+, -,*
SIGNED	UNSIGNED	STD_LOGIC_VECTOR	+, -*
UNSIGNED	INTEGER	STD_LOGIC_VECTOR	+, -
INTEGER	UNSIGNED	STD_LOGIC_VECTOR	+, -
SIGNED	INTEGER	STD_LOGIC_VECTOR	+, -
INTEGER	SIGNED	STD_LOGIC_VECTOR	+, -
UNSIGNED	STD_ULOGIC	STD_LOGIC_VECTOR	+, -
STD_ULOGIC	UNSIGNED	STD_LOGIC_VECTOR	+, -
SIGNED	STD_ULOGIC	STD_LOGIC_VECTOR	+, -
STD_ULOGIC	SIGNED	STD_LOGIC_VECTOR	+, -

Table: Arithmetic operators' functions defined in std_logic_arith package

Left Operand	Right Operand	Operation
UNSIGNED	UNSIGNED	< , <=, > , >=,=, /=
SIGNED	SIGNED	< , <=, > , >=,=, /=
UNSIGNED	SIGNED	< , <=, > , >=,=, /=
SIGNED	UNSIGNED	< , <=, > , >=,=, /=
UNSIGNED	INTEGER	< , <=, > , >=,=, /=
INTEGER	UNSIGNED	< , <=, > , >=,=, /=
SIGNED	INTEGER	< , <=, > , >=,=, /=
INTEGER	SIGNED	< , <=, > , >=,=, /=

Table: Relational operators' functions defined in std_logic_arith package

Operand	Return Type	Operation
UNSIGNED	UNSIGNED	+
UNSIGNED	STD_LOGIC_VECTOR	+,ABS
SIGNED	SIGNED	-
SIGNED	STD_LOGIC_VECTOR	-, ABS

Table: Unary operators' functions defined in std_logic_arith package

Operand	Return Type	Conversion Function
INTEGER	INTEGER	CONV_INTEGER
UNSIGNED	INTEGER	CONV_INTEGER
SIGNED	INTEGER	CONV_INTEGER
STD_ULOGIC	INTEGER	CONV_INTEGER

Table: Conversion functions defined in std_logic_arith package

Left Operand	Right Operand	Result	Conversion Function
INTEGER	INTEGER	UNSIGNED	CONV_UNSIGNED
UNSIGNED	INTEGER	UNSIGNED	CONV_UNSIGNED
SIGNED	INTEGER	UNSIGNED	CONV_UNSIGNED
STD_ULOGIC	INTEGER	UNSIGNED	CONV_UNSIGNED
INTEGER	INTEGER	SIGNED	CONV_SIGNED
UNSIGNED	INTEGER	SIGNED	CONV_SIGNED
SIGNED	INTEGER	SIGNED	CONV_SIGNED
STD_ULOGIC	INTEGER	SIGNED	CONV_SIGNED
INTEGER	INTEGER	STD_LOGIC_VECTOR	CONV_STD_LOGIC_VECTOR
UNSIGNED	INTEGER	STD_LOGIC_VECTOR	CONV_STD_LOGIC_VECTOR
SIGNED	INTEGER	STD_LOGIC_VECTOR	CONV_STD_LOGIC_VECTOR
STD_ULOGIC	INTEGER	STD_LOGIC_VECTOR	CONV_STD_LOGIC_VECTOR

Table: Conversion functions defined in std_logic_arith package(right operand specifies the size of the vector returned by the function)

2.3 Package std_logic_signed and std_logic_unsigned

These two packages support operator definitions and conversion functions that are specific to signed and unsigned numbers. While these operators are already defined in *std_logic_arith* package, their implementation in these two packages is in terms of standard logic system defined using *std_logic* and *std_logic_vector* types.

Left Operand	Right Operand	Return Type	Operators
STD_LOGIC_VECTOR	STD_LOGIC_VECTOR	See Note Below	+,-,*,<,<=,>,>=,/=,SHL,SHR
STD_LOGIC_VECTOR	INTEGER	See Note Below	+,-,<,<=,>,>=
INTEGER	STD_LOGIC_VECTOR	See Note Below	+,-,<,<=,>,>=
STD_LOGIC	STD_LOGIC_VECTOR	See Note Below	+,-
STD_LOGIC_VECTOR	STD_LOGIC	See Note Below	+,-

Table: Binary operator functions defined in std_logic_unsigned and std_logic_signed packages

Note: Arithmetic operators return std_logic_vector, relational operator return boolean.

Operand	Return Type	Operators
STD_LOGIC_VECTOR	STD_LOGIC_VECTOR	+,-,ABS

Table: Unary operator's functions defined in std_logic_unsigned and std_logic_signed packages

Operand	Return Type	Conversion Function
STD_LOGIC_VECTOR	INTEGER	CONV_INTEGER

Table: Conversion functions defined in std_logic_unsigned and std_logic_signed packages

3 VHDL Statements

3.1 Design Units

3.1.1 Entity Declaration

Entity declaration describes an interface aspect of a design. The entity declaration part contains entity header, entity declarative region and entity statement part in that order.

```
entity regbank is
        -- entity header
        generic (width : integer := 8);                          -- generic clause
        port (    d : in std_logic_vector(WIDTH-1 downto 0),     -- port clause
                  q : out std_logic_vector(WIDTH-1 downto 0),
                  clock,reset : in std_logic);
        -- entity declarations
        constant Thold : time := 5 ns;
        -- entity statements part
        begin
                  process (clk)
                  begin
                            if(clock = '1' and clock'event) then
                                      if(d'stable(Thold) = FALSE) then
                                                report "Hold violation"
                                                          severity WARNING;
                                      end if;
                            end if;
                  end process;
end entity;
```

3.1.1.1 Entity header

Entity header contains a port list with which it communicates with the outside world and a set of generic values that are passed on to design from the outside world.

3.1.1.3 Generic

- Useful for module parameterization
- Useful for supplying runtime environment such as timing values, speed grades etc.

3.1.1.2 Ports

Ports can have type

- In – input port
- Out – Output port
- Buffer – A signal associated with port is allowed to have at most one driver
- Linkage – the direction is unknown

3.1.1.4 Entity declarative part

The declarations in entity can be one of the following

- Subprogram declaration
- subprogram body
- type declaration
- subtype declaration
- constant declaration
- signal declaration
- shared variable declaration
- file declaration
- alias declaration
- attribute declaration
- attribute specification
- disconnection specification
- use clause
- group template declaration
- group declaration

3.1.1.5 Entity statement part

The statements in entity statement part can be one of the following statements

- concurrent assertion statement
- passive concurrent procedure call
- passive process statement

3.1.2 Package declarations

Packages encapsulate information that can be shared between more than one design unit.

```
package mem is
        -- package declarative part
        type nibble is array (3 downto 0) of BIT;
        type byte is array (7 downto 0) of BIT;
        type word is array(15 downto 0) of BIT;
        type mem8x32 is array (natural range <>) of byte;
        function writeMem (mem : mem8x32;data : byte; address : natural) return natural;
end mem;
```

Package declarative item can be one of the following statements.

- subprogram declaration
- type declaration
- declaration
- constant declaration
- signal declaration
- shared variable declaration
- file declaration
- alias declaration
- component declaration
- attribute declaration
- attribute specification
- disconnection specification
- use clause
- group template declaration
- group declaration

3.1.2.1 Package Bodies

The package body specifies the behavior for the items declared in the package declaration.

```
package body mem is
        -- package body
        function writeMem (mem : mem8x32;data : byte; address : natural) return natural is
        begin
                        mem(address) <= data;
                        address <= address + 1;
                        return address ;
        end writeMem;
end mem;
```

The package body can contain following statements.

- subprogram declaration
- subprogram body
- type declaration
- subtype declaration
- constant declaration
- shared variable declaration
- file declaration
- alias declaration
- use clause
- group template declaration
- group declaration

3.1.3 Configuration Declaration

Configurations are used

- bind the entity to desired architecture implementation (3.1.3.1)
- bind the component instantiations to their desired entity-architecture or configurations(3.1.3.2)
- redefine the port names of the components at the place of instantiation(3.1.3.3)
- relink the component instantiation to different entity other than the one for which component is declared (3.1.3.4)
- late binding of parameters which have to be passed on to design (e.g. delay values) (3.1.3.5)

3.1.3.1 Default configuration

If an entity has two architectures and if the configuration is not specified, then the architecture that *is seen last* is bound to that entity. Otherwise, a default configuration statement for each architecture can be specified:

```
configuration configurationName of entityName is
for architecture
end for;
end configurationName;
```

See example below.

3.1.3.2 Component configuration

Component configurations are used to bind component instantiations to their implementations. Component configurations can be specified using one of the following ways.

- Indirect configuration – specifying component configuration in terms of another configuration. This configuration statement must be specified as a design unit, outside the scope where components are instantiated (case I).
- Direct configuration - specifying component configuration using entity and architecture pair directly in the configuration statement. This configuration statement must be specified as a design unit, outside the scope where components are instantiated (case II).
- Architecture configuration – specifying configuration for components in the architecture declarative part (case III)

Using **all**, **others** and **instance name** as named instances, configuration specification can be simplified. See examples below.

```
--
-- inverter interface
--
entity inv is
        generic (delay : time := 2ns);
        port (a : in bit; q : out bit);
end inv;

--
-- inverter behavioural implementation
--
architecture behv of inv is
begin
        q <= '0'after delay  when a = '1' after delay else '1';
end behv;
--
-- inverter data flow implementation
--
architecture dataflow of inv is
        q <= not (a) after delay;
end dataflow;

--
-- default configuration for data flow architecture
--
configuration inv_dataflow_conf is
for dataflow
end for;

--
-- default configuration for behv architecture
--
configuration inv_behv_conf is
for behv
end for;

entity buffer is
        port (i_pin: in bit; o_pin : out bit);
end entity;

architecture structural of buffer is
component inv port (a : in bit; q : out bit);
end component;
signal bit : tmp;
begin
        I1: inv port map (a => I_pin, q => tmp);
        I2: inv port map(a => tmp, q => o_pin);
end structural;
```

```
--
-- case I
--

configuration case_I of buffer is
for structural
        for I1 : inv use configuration work.inv_dataflow_conf;
        end for;
        for I1 : inv use configuration work.inv_behv_conf;
        end for;
end for;
end case_I;

Alternatively; for all inverter instances

configuration case_I of buffer is
for structural
        for all : inv use configuration work.inv_dataflow_conf;
        end for;
end for;
end case_I;
```

Alternatively, dataflow configuration for I1 can be used explicitly and configuration for remaining instances can be specified using **other** clause.

```
configuration case_I of buffer is
for structural
        for I1 : inv use configuration work.inv_dataflow_conf;
        end for;
        for other : inv use configuration work.inv_behv_conf;
        end for;
end for;
end case_I;

--
-- case II
--
configuration case_II of buffer is
for structural
        for all : inv use entity work.inv(dataflow);
        end for;
end for;
end case_II;
```

```
--
-- case III
--

architecture structural of buffer is
component inv port (a : in bit; q : out bit);
end component;

for I1 : inv use configuration work.inv_behv_conf;
for I2 : inv use configuration entity work.inv(dataflow);

signal bit : tmp;
begin
        I1: inv port map (a => I_pin, q => tmp);
        I2: inv port map(a => tmp, q => o_pin);
end structural;
```

3.1.3.3 Port Maps

The names of the ports can be redefined using configuration statement.

```
architecture structural of buffer is
-- The original inv entity has a and q as pin names
-- These are changed here and their correspondence
-- is established in the configuration statement
component inv port (newA : in bit; newQ : out bit);
end component;

signal bit : tmp;
begin
        I1: inv port map (newA => I_pin, newQ => tmp);
        I2: inv port map(newA  => tmp, newQ  => o_pin);
end structural;

configuration renamePortMaps of buffer is
for structural
        -- substitute inv with invNew
        for all : inv use entity work.inv(dataflow);
        port map (a => newA, q => newQ);
        end for;
end for;
end renamePortMaps ;
```

3.1.3.4 Mapping components to different entity

The name of the entity can be changed using configuration statement as shown below.

```
entity inv is
        port(a : in bit; q : out bit);
end entity;

--
-- some inv architecture
--

entity invNew is
        port(a : in bit; q : out bit);
end entity;

--
-- some invNew architecture
--

architecture structural of buffer is
--
-- used inv and substituted invNew in the configuration
--

component inv port (newInput : in bit; newOutput : out bit);
end component;
signal bit : tmp;
begin
        I1: inv port map (newInput => I_pin, newOutput => tmp);
        I2: inv port map(newInput  => tmp, newOutput  => o_pin);
end structural;

configuration renameEntityName of buffer is
for structural
        for all : invNew use entity work.inv(dataflow);
        end for;
end for;
end renameEntityName ;
```

3.1.3.5 Generics in the configuration statement

Default generic parameters defined in the entity can be overridden in the configuration.

```
configuration config of buffer is
for structural
        for all : inv use entity work.inv(dataflow);
        generic map (delay => 20 ns);
        end for;
end for;
end config;
```

3.1.4 Block configuration

If the architecture definition contains blocks, configuration statement should account for the block hierarchy.

```
architecture structural of buffer is
component inv port (a : in bit; q : out bit);
end component;

signal bit : tmp;
begin
        blk1 : block
                I1: inv port map (a => I_pin, a => tmp);
        end block blk1;

        I2: inv port map(a  => tmp, q  => o_pin);
end structural;

configuration blockConf of buffer is
for structural
        for blk1
                for I1  : inv use entity work.inv(dataflow);
                end for;
        end for;
        for I2  : inv use entity work.inv(dataflow);
        end for;

end for;
end blockConf;
```

3. 1.5 Architecture Bodies

Architecture body provides an implementation for the interface declared in the entity part. There can be multiple architectures for the single entity.

```
architecture behv of andinv is
-- architecture declarative part
signal andOutput;
begin
-- architecture statements part

andOut <= a and b;
and <= andOut;
nand <= not andOut;
end behv;
```

3.1.5.1 Architecture declarative part

The following statements are allowed in architecture declarative part

- 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
- disconnection specification
- use clause
- group template declaration
- group declaration

3.1.5.2 Architecture statement part

Only concurrent statements are allowed in architecture statement part.

3.2 Libraries

Library is an implementation dependent database in which previously analyzed design units are placed. Library is declared using library keyword following a list of library names.

library ieee, constants;

The library in which user supplied design units are being analyzed is called a work library.

Use clause is used to provide visibility to the components defined in the library.

use ieee.std_logic_1164.all; -- make visible all information declared in unit std_logic_1164
use processor.busFunctions.”dma”-- provide an access to only function dma defined in unit busFunction

3.3 Concurrent Statements

3.3.1 Block statement

Block is a part of the design that is enclosed in the architecture. Blocks form the partition of the design and can be construed as a schematic sheet.

```
architecture blockOriented of latch is
begin
        latch : block
                -- block header
                -- block declarative part
                port (en,d : in std_logic; q: out std_logic);                -- block interface
                port map (en => latchEnable, d => data, q => output); -- block connection to
                                                                             -- outside world
                begin
                        -- block statements
                        process (en)
                        begin
                                if(en) then
                                        q <= d;
                                end if;
                        end process;
        end block latch;
end blockOriented;
```

Following statements are allowed in block header.

- generic clause
- generic map aspect
- port clause
- port map aspect

The list of statements allowed in block declarative part is same as that allowed in architecture declaration part and is given in section 3.1.5.1.

Only concurrent statements are allowed in block statement part.

3.3.2 Process statement

Processes encapsulate a set of sequential statements. A process is activated when a signal given in its sensitivity list changes a value. If a process does not have a sensitivity list, it must have a wait statement in which process waits till the wait expression is evaluated true.

```
ff : process (clk, rst)
--process declarative part
begin
-- process statements part
        if(rst = '0') then
                q <= '0';
        else if(clk'event and clk = '1') then
                1 <= d;
        end if;
end process ff;
```

Equivalently,

```
ff : process
begin
        if(reset = '1') then
```

```
            q <= '0';
        else if(clk'event and clk = '1') then
            q <= d;
        end if;
        wait on (clk , rst);
end process;
```

The following statements are allowed in process declarative part.

- subprogram declaration
- subprogram body
- type declaration
- subtype declaration
- constant declaration
- variable declaration
- file declaration
- alias declaration
- attribute declaration
- attribute specification
- use clause
- group type declaration
- group declaration

Only sequential statements are allowed in statements part.

3.3.3 Concurrent procedure call statements

Concurrent procedures provide a convenience of defining a sequential procedure (say in a package or in a declarative region of the enclosing scope) and calling it as a concurrent statement. There is always a corresponding process statement for every concurrent procedure.

```
CheckTiming (tPLH, tPHL, Clk, D, Q);          -- A concurrent procedure called statement.

process                                       -- The equivalent process.
begin
        CheckTiming (tPLH, tPHL, Clk, D, Q);
        wait on Clk, D, Q;
end process;
```

3.3.4 Concurrent assertion statements

Concurrent assertion statement can be written as:

e.g. assert (now – last_clk_change = 20 ns) report “Glitch on the clk signal” severity error;

3.3.6 Conditional signal assignments

Conditional signal assignment statements are illustrated with following examples.

```
--
-- mux
--

q <= d0 when s = “00” else
       d1 when s = “01” else
       d2 when s = “10” else
       d3 when s = “11” else
       ‘x’;

--
-- demonstrates use of unaffected keyword
--

S <= unaffected when Input_pin = S'DrivingValue else
      Input_pin after Buffer_Delay;
```

3.3.7 Selected signal assignments

The selected signal assignment statement is illustrated with the examples given below.

```
--
-- with statement
--

with sel select
          q <= a after 5 ns when 0,
                b after 5 ns when 1,
                ‘x’ after 0 ns when others;
```

3.3.8 Component instantiation statements

A component instantiation statement is used to build well-partitioned hierarchical designs. VHDL supports instantiation of a component, instantiation of a design entity and an instantiation of configuration.

3.3.8.1 Instantiating component

Instantiating a component consists of declaring a component including its port and generic maps and using it in the architecture by supplying port map aspect and generic map aspect.

architecture structural of dp is

```
component regbank
                        generic (width : integer := 16);
                        port(clk,r : in std_logic;
                        d : in std_logic_vector(width-1 downto 0);
                        q : out std_logic_vector(width-1 downto 0) );
end component;

begin
        b8 : regbank
                        generic map (width => 16),
                        port map ( clk => sysclk, r =>  r, d => datain, q => dataout);
```

3.3.8.2 Instantiating entity

An entity can be directly instantiated into its enclosing scope as shown in the example below.

b8: entity Work.regbank (behv) port map (clk => sysclk, r => sysrst, d => datain, q => dataout);

Unlike in the case of component instantiation, an entity that is being instantiated must be previously analyzed design unit.

3.3.8.3 Instantiating configuration

b8: configuration work.config_structural port map (clk => sysclk, r => sysrst, d => datain, q => dataout);

config_structural is the name of the configuration that binds the entity architecture pair of the design unit.

3.3.9 Generate statements

Generate statement is used to instantiate components that display regularity between their interconnections. The example shows interconnections between full adders to implement a ripple carry adder.

Two generate schemes are available. IF GENERATE clause is used to generate a slice conditionally. This scheme is useful for creating hardware on the edges. The FOR GENERATE scheme is used for replicating the hardware slices.

Any block declarative statements listed in section 3.1.5.1 can be used in the block declarative region of generate statement. Only concurrent statements are allowed in the statements part.

```
g1 : for i in 0 to (times - 1) generate
        -- block declarative region (optional, if present statement begin should start statements part)
        begin – (optional, should be used only if block declarative statements are present)
        -- statements
                g2: if( i = 0) generate
                fullAdder :   inst port map  (
                            A => ain(i),
                            B => bin(i),
                            CI => CarryIn,
                            CO => temp(i);
                        );
                end generate;

        g3: if( i> 0 and i < (times - 1) ) generate
                             fullAdder :   inst port map  (
                            A => ain(i),
                            B => bin(i),
                            CI => temp(i-1),
                            CO => temp(i);

        end generate;

        g4: if( i = (times - 1)) generate
                            fullAdder :   inst port map  (
                            A => ain(i),
                            B => bin(i),
                            CI => temp(i-1),
                            CO => carryOut;
        end generate;
        end generate;
```

3.4 Sequential Statements

3.4.1 Wait statement

Wait statement is used to suspend the execution of the process. Different flavors of the wait statement are listed below.

```
wait for 10 ns;              -- wait for 10 ns
wait until (reset = '1');    -- wait till the expression is true
wait on (clk,rst)            -- wait for signals clk and rst to change
```

3.4.2 Assertion statement

Sequential assertion statement appears in a process.

```
assert (now – last_clk_change = 20 ns) report "Glitch on the clk signal" severity error;
```

3.4.3 Report statement

Report statement is used to report textual message to users.

```
report "Can't open file:in.dat" severity error;
```

3.4.4 Signal assignment statement

Various signal assignment statements are illustrated with the examples below.

```
--  Assignments using inertial delay:

    --  The following three assignments are equivalent to each other:

        Output_pin <= Input_pin after 10 ns;
        Output_pin <= inertial Input_pin after 10 ns;
        Output_pin <= reject 10 ns inertial Input_pin after 10 ns;

    --  Assignments with a pulse rejection limit less than the time expression:

        Output_pin <= reject 5 ns inertial Input_pin after 10 ns;
        Output_pin <= reject 5 ns inertial Input_pin after 10 ns, not Input_pin after 20 ns;

--  Assignments using transport delay:

        Output_pin <= transport Input_pin after 10 ns;
        Output_pin <= transport Input_pin after 10 ns, not Input_pin after 20 ns;

    --  Their equivalent assignments:

        Output_pin <= reject 0 ns inertial Input_pin after 10 ns;
        Output_pin <= reject 0 ns inertial Input_pin after 10 ns, not Input_pin after 10 ns;
```

3.4.5 Variable assignment statement

Variable assignment statements are illustrated with examples below.

```
I := 10;          -- constant
J := I * 3;       -- expression
```

3.4.6 Array variable assignments

The array variable assignments are illustrated with the example below.

```
variable intArray : array (7 downto 0) of integer;
variable bitArray : array (7 downto 0) of bit;

intArray(1) := 20;
bitArray := "10101011";
```

3.4.7 Procedure call statement

A procedure call statement is illustrated with the example shown below.

```
process
begin
        CheckTiming (tPLH, tPHL, Clk, D, Q);
        wait on Clk, D, Q;
end process;
```

3.4.8 If statement

The if statement is illustrated with the example below.

```
--
-- 4 to 1 Mux
--

if(s = "00") then
        q <= d0;
elsif (s = "01") then
        q <= d1;
elsif(s = "10") then
        q <= d2;
elsif(s = "11") then
        q <= d3;
else
        q <= 'X';
end if;
```

3.4.9 Case statement

Case statement is illustrated with the following example.

```
--
-- 4 to 1 mux
--

case s is
        when “00” =>
                q <= d0;
        when “01” =>
                q <= d1;
        when “10” =>
                q <= d2;
        when “11” =>
                q <= d3;
        when others =>
                q <= ‘X’;
end case;
```

3.4.10 Loop statements

There are two loop statements, loop and while. These statements are illustrated using examples below.

```
--
-- loop indefinately
--

loop
        checkTiming(CLK,RESET,D)
        wait on(clk,reset);
end loop;

--
-- loop fixed number of times
--
for I in 1 to 5 loop
        sum := sum + a(I);
end loop;

--
-- loop until some condition is satisfied
--
while (a(i) = ‘x’) loop
        i := i + 1;
end loop;
```

3.4.11 Next statement

Next statement is used to skip the iteration of the loop.

```
for I = 0 to 10 loop
        if(a(I) < 0) then
                next;      -- skip this iteration
        end if;
        sum :=  sum + a(I);
end loop;
```

3.4.12 Exit statement

Exit statement is used to exit out of the loop.

```
for I = 0 to 10 loop
        if(a(I) < 0) then
                exit;      -- exit from the loop
        end if;
        sum :=  sum + a(I);
end loop;
```

3.4.13 Return statement

Return statement is used to return from a function or the procedure.

```
function swap ( vec : in std_logic_vector(1 downto 0) ) return std_logic_vector is
variable tmp : std_logic;
begin
        tmp := vec(0);
        vec(0) := vec(1);
        vec(1) := tmp;
        return vec;
end function;
```

3.4.14 Null statement

Null statement is used to convey explicitly that no action is to be taken.

```
--
-- 4 to 1 mux (for illustration only)
--

case s is
        when “00” =>
                q <= d0;
        when “01” =>
                q <= d1;
        when “10” =>
                q <= d2;
        when “11” =>
                q <= d3;
        when others =>
                null;
end case;
```

3.5 Specifications

3.5.1 Attribute specification

Attribute declaration

An attribute is a value, function, type, range, signal, or constant that may be associated with one or more named entities in a description. There are two categories of attributes: predefined attributes and user-defined attributes. Predefined attributes are described in section 2.

Attribute declarations are illustrated with examples given below.

```
attribute pad : boolean;          -- declare an attribute pad which is either true or false
attribute optimize : boolean;     -- declare an attribute optimize which is either true or false
```

Attribute Specifications

Attribute specifications associate attribute name and the value to the entities in the VHDL design. Attributes can be specified for the following entities.

entity	architecture	configuration
procedure	function	package
type	subtype	constant
signal	variable	component
label	literal	units
group	file	

Attributes are illustrated with following examples.

```
attribute pad of AO: signal is true;                -- AO is a pad pin
attribute optimize of add8: component is true;      -- add8 is to be optimized away
attribute optimize of others: component is false;   -- all other components are to be preserved
```

3.5.2 Disconnection specification

3.6. Type declarations

VHDL allows declarations of user defined types. Type declarations are illustrated with examples below.

```
-- short integer
type short is integer range 0 to 255;

-- decimal number system using enumerated type
type decimal is (zero,one,two,three,four,five,six,seven,eight,nine);

-- frequently used bit sizes
type byte is array (7 downto 0) of std_logic;
type word is array (15 downto 0) of std_logic;

-- kilobyte
type kbyte is array (0 to 1024) of byte;

-- two dimentional array indexed by non integers
type delayKind is (min,typ,max)
type scalingFactor is array (delayKind range <>) of real;

-- unconstrained array
type bv is array (natural range <>) of bit;

-- type containing records
type instruction is
        record
                opcode : optype;
                src : integer;
                des : integer;
        end record;

-- incomplete types
type node is
        record
                data : integer;
                next  : ptr;
        end record;
type ptr is access node;

-- file types
type IntegerFile is file of INTEGER;
```

3.6.1 Subtype declarations

Subtypes are the subsets of the original types. Subtype declarations are illustrated using examples below.

```
subtype wholeNumber is integer range 0 to 2,147,483,647;
subtype alpha is character range ‘A’ to ‘Z’;
```

3.6.2 Objects

An object is a named entity that contains (has) a value of a given type. An object is one of the following:

- An object declared by an object declaration (signal,variable,constant or file)
- A loop or generate parameter
- A formal parameter of a subprogram
- A formal port
- A formal generic
- A local port
- A local generic
- An implicit signal GUARD defined by the guard expression of a block statement

In addition, the following are objects, but are not named entities:

- An implicit signal defined by any of the predefined attributes 'DELAYED,'STABLE, 'QUIET, and 'TRANSACTION
- An element or slice of another object
- An object designated by a value of an access type

3.6.3 Object declarations

Objects can be one of the four types defined below.

- constant declaration
- signal declaration
- variable declaration
- file declaration

3.6.4 Constant declarations

A constant declaration declares a constant of a specified type.

constant pi : real := 3.14

3.6.5 Signal declarations

A signal declarations declare signals.

```
signal A : std_logic_vector (1 to 10) ;
-- signal having an initial value
signal reset : std_logic := 0 ;

-- A signal having resolution function
signal output : wired_or mvl;
```

3.6.6 Variable declarations

Variables are declared using variable declaration statement.

```
variable I : integer := 0;
variable A : std_logic := '1';
```

3.6.7 File declarations

File declaration is used to declare files of specified types.

```
type IntegerFile is file of INTEGER;

file F1: IntegerFile;       -- No implicit FILE_OPEN is performed
                            -- during elaboration.

file F2: IntegerFile is "test.dat";   -- At elaboration, an implicit call is performed:
                                      -- FILE_OPEN (F2, "test.dat");
                                      -- The OPEN_KIND parameter defaults to
                                      -- READ_MODE.

file F3: IntegerFile open WRITE_MODE is "test.dat";
                                      -- At elaboration, an implicit call is performed:
                                      -- FILE_OPEN (F3, "test.dat", WRITE_MODE);
```

3.6.8 Interface declarations

3.6.9 Interface lists

3.6.10 Association lists

3.6.11 Alias declarations

Aliases are the alternate names given to objects or non objects.

3.6.12 Object aliases

Object aliases are illustrated with the example below.

```
signal signedNumber : std_logic_vector(7 downto 0);
alias sign : std_logic is signedNumber(7);
```

3.6.13 Non object aliases

Aliases to entities belonging to non object categories are illustrated below.

```
alias INDUSTRY_STANDARD_LOGIC   is IEEE.STD_LOGIC_1164.STD_LOGIC;
```

3.6.14 Component declarations

3.6.15 Group template declarations

3.6.16 Group declarations

3.7 Scalar types

3.7.1 Enumeration types
3.7.1 Predefined enumeration types
3.7.2 Integer types
3.7.3 Physical types
3.7.4 Floating point types
3.7.5 Composite types
3.7.6 Array types
3.7.7 Index constraints and discrete ranges
3.7.8 Record types
3.7.9 Access types
3.7.10 Incomplete type declarations
3.7.11 File types

3.8 Sub-programs

Subprograms are the reusable units of computations. Subprograms are divided into functions and procedures. Functions take arguments and can return only one value via return statement. The procedure does not have return statement but it can return multiple values to calling area via its arguments.

3.8.1 Subprogram declarations

procedure declaration

procedure prodecureName (optional parameter list)

e.g. procedure accumulate (acc : inout word; val : in byte; flag : out boolean);

function declaration

function functionname (parameter list) return returnType;

e.g. function add (a : in byte; b: in byte) return word;

The procedure names and function names can be predefined operator names to support operator overloading.

3.8.2 Formal parameters

Formal parameters to a subprogram can be constants, variables, signals, or files. Parameter of type constants, variables and signals has modes either in, out or inout associated with them. Parameter of type file does not have a mode associated with it.

A procedure can have parameters of in, out or inout mode. If the mode is in and no object class is explicitly specified, constant is assumed. If the mode is inout or out, and no object class is explicitly specified, variable is assumed.

Functions can have parameters having in mode. The object class must be constant, signal, or file. If no object class is explicitly given, constant is assumed.

In a call to a sub program, a formal arguments and actual arguments of type signal and variable and file must match in type. If formal is a constant, then the actual argument can be an expression.

3.8.2.1 Constant and variable parameters

3.8.2.2 Signal parameters

3.8.2.3 File parameters

3.8.3 Subprogram bodies

The implementation of the subprogrms is supplied in the subprogram body.

```
procedure accumulate (acc : inout word; val : in byte; flag : out boolean) is
-- subprogram declarative part
begin
-- subprogram statements part
acc <= acc + byte;
flag <= '0' when acc < 0 else
        '1';
end procedure accumulate;
```

Following statements are allowed in subprogram declarative region.

- subprogram declaration
- subprogram body
- type declaration
- subtype declaration
- constant declaration
- variable declaration
- file declaration
- alias declaration
- attribute declaration
- attribute specification
- use clause
- group template declaration
- group declaration

Only concurrent statements are allowed in the statements part.

3.8.4 Subprogram overloading

3.8.5 Operator overloading

3.8.6 Signatures

3.8.7 Resolution functions