SECTION 2

Declarations and sequential constructs

Lexical elements

Character set

Letters: A..Z, a..z

Digits: 0..9

Special characters " # & ' () +, -. / : ; < = > _ | ! $ % @ ? [ ] ^ ` {} ~ *

Space

An identifier is

- letter {[underline] letter_or_digit}
- not case sensitive

Literals (examples)

Decimal literals

Integer literals: 12 0 1E6 123_456

Real literals: 2.0 0.0 0.456 3.14159_26

1.34E-12 1.0E+6 6.023E24

Based literals

Integer literals of value 255: 2#1111_1111# 16#FF# 016#FF#

Integer literals of value 224: 16#E#E1 2#1110_0000#

Real literal of value 4095.0: 16#F.FF#E2 2#1.1111_1111_111#E11

Bit string-literal

X"FFF" = B"1111_1111_1111"

O"777" = B"111_111_111"

X"777" = B"0111_0111_0111"

Character and string literals

Character literals:

enclosed between two ' characters: 'A' '*' '"' ' '

String literals:

Text enclosed between two " characters

A string must fit on one line.

Use concatenation operation to obtain longer strings:

"First part of a string" &

"continuation on a second line"

Use concatenation operation to include non graphic characters in a string:

"sequence that includes the " & ACK & "control character"

Comments

```
-- VHDL is a strongly typed language
-- LRM means Language Reference Manual
-- Refer to the LRM for the syntax and semantics of VHDL
-- The LRM contains also  the definition of the standard package,
-- A glossary and a syntax summary using BNF notation
end;                    -- end is a reserved word
                        -- reserved words are in bold face troughout the text
```


Declarations

- Type and subtype
- Constant, variable, signal
- Package
- Entity
- Architecture
- Configuration
- Subprograms

Type and subtype declarations

Scalar types

- Enumeration
- Integer
- Floating
- Physical

Composite types

- Array
- Record

Type and subtype declarations

- A type is characterized by a set of values and a set of operations.
  ex: **type** BYTE_VALUE **is range** 0 **to** 255;
- The set of possible values of a given type may be subjected to a constraint.
- A subtype is a type (base type) plus a constraint.
  ex: **subtype** FIRST_100 **is** INTEGER **range** 1 **to** 100;
  subtype SEVEN_BIT_V **is** BYTE_VALUE **range** 0 **to** 127;
  - The constraint is checked during simulation.
  - A type is a subtype of itself.
  - The base type of a type is the type itself.

Enumeration types

Enumeration type definition

== (enumeration literal { , enumeration literal})

Enumeration literal == identifier | character literal

Encoded as bit vectors

Examples

type TRANSITION **is** (H, L, R, F, U);

Encoding: H:"000", L:"001", R:"010", F:"011", U:"100"

type BIT **is** ('0', '1');

type LOGIC4 **is** ('X', '0', '1', 'Z'); -- '0' and '1' overloaded, type resolved by context

type MODE **is** (NORMAL, SCAN, TEST);

- Order: NORMAL < SCAN < TEST
- Default initialization of signals to the 'LEFT value.

  (Attribute of a type or object, e.g. MODE'LEFT, MODE'RIGHT)

Predefined Enumeration types

- Character

  128 ASCII characters
- BIT

  ('0', '1')
- BOOLEAN

  (FALSE, TRUE)
- SEVERITY_LEVEL

  (NOTE, WARNING, ERROR, FAILURE)

Integer types

Examples

type byte **is range** 0 **to** 255; synthesized to 8 bits

type other **is range** -6 **to** 7; synthesized to 4bits: 2's-complement bit vector

type WIDTH **is range** 31 **downto** 0;

type reverse **is range** WIDTH'LOW **to** WIDTH'HIGH;

subtype AT_WIDTH **is** WIDTH **range** 15 **downto** 0;

Each bound must be a locally static expression of some integer type

Predefined integer types

The only one is the type INTEGER

- The range is implementation dependent
- guaranteed to include the range: -2147483647 to + 2147483647(2**31+1 to 2**31-1)
- The range may be determined from attributes

INTEGER'LOW, INTEGER'HIGH

Physical types

- TIME is the only predefined type
- Appears in package STANDARD
  - Guaranteed to include the range -2**31+1 to +2**31-1
  - All specification of delays must be of type TIME
- Not supported in FPGA Synthesis

```
type TIME is range -1E18 to 1E18                    -- range constraint,
    implementation dependent
units                                               -- keyword
    fs;                                             -- base unit declaration
    ps   =          1000 fs;                        -- secondary unit declaration
    ns   =          1000 ps;                        -- ...
    us   =          1000 ns;
    ms   =          1000 us;
    sec  =          1000 ms;
    min  =          60 sec;
end units;
```

Floating point types

- Approximate real numbers
- A floating point is defined by a range constraint
- Each bound must be a locally static expression
- Each bound must be of some floating point type
- Not supported in FPGA Synthesis

Examples

type coord_x_type **is range** 0.0 **to** 100.0
subtype One_interval **is** coord_x_type **range** 0.0 **to** 1.0;

Predefined floating point types

- REAL is the only predefined type
- Appears in package STANDARD
- Guaranteed to include the range -1E38 to +1E38
- Defined with an ascending range
- The range may be determined from attributes REAL'LOW, REAL'HIGH

Array types

Composite types are: array types and record types

An array type definition may be either:

- constrained
- unconstrained

constrained array definition

array (discrete range {, discrete range})
of element subtype indication

unconstrained array definition

array (subtype name **range** <> {, subtype name **range** <>})
of element subtype indication

Array types (cont'd)

Predefined array types

```
subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH
type STRING is array (POSITIVE range <>) of CHARACTER;
subtype NATURAL is INTEGER range 0 to INTEGER'HIGH
type BIT_VECTOR is array (NATURAL range <>) of BIT;
```

Examples

```
type mu_word is array (48 downto 0) of BIT;
type Memory is array (NATURAL range <>) of mu_word;

-- Possible declarations in a sequential part (in  the body of a process for example)
variable ROM : Memory (0 to 2**N-1);
variable ScanChain : BIT_VECTOR (0 to N*L);

-- Possible declaration in a concurrent part (in an architecture for example)
signal mu_pc : mu_word;
```


Record types

Examples

```
type DATE is
record
    YEAR: POSITIVE range 1 to 3000;
    MONTH: STRING(0 to 8);
    DAY: POSITIVE range 1 to 31;
end record;

type VERSION is
record
    NUMBER:   REAL range 0.0 to 30.0;
    CREATED:  DATE;
    MODIFIED: DATE;
end record;
```

Object declaration

Constant declaration

Variable declaration (in a sequential context)

Signal declaration (in a concurrent context)

Constant declaration

constant identifier {, identifier} : **subtype** indication [:= expression]

The value of a constant cannot be modified after the declaration is elaborated.

Examples:

```
constant normal_weight :weight := 70 kg;
constant cycle  : TIME := 1 us;
constant e : real :=  2.71828 ;
type vect_n is array (Positive range <>) of NATURAL;
constant Five_Ones : vect_n := (1, 1, 1, 1, 1);
                    -- The range of Five_Ones is 5 due to the assignment.
```

Constants of type record are not supported for synthesis
(the initialization of records is not supported).

Variable declaration

variable identifier {, identifier} : **subtype** indication [:= expression]

- A variable is an object with a single current value.
  (no history or future projected values.)
- The expression (if present) specifies an initial value of the variable.
  It is evaluated at each elaboration of the variable declaration.
- The default initial value for a variable of a scalar type T is T'left.
- Can appear only in algorithmic descriptions: inside processes, procedures and functions.

Examples

variable atomic_interval : time; -- initial value -1E18 fs (implementation dependent)
variable vector_5 : bit_vector (1 **to** 5); -- initial value is ('0', '0', '0', '0', '0')
variable index : integer **range** 0 **to** 99 := 10; -- explicit initial value 10

Signal declaration

signal identifier {, identifier} : **subtype** indication [**bus** | **register**] [:= expression]

- A signal is an object with a past history of values.
- It may have one (or many) drivers, each with a current value
  and projected future values.
- The expression (if present) specifies the initial value of the signal at simulation time 0. The expression is evaluated at each elaboration of the signal.
- The default initial value for a signal of a scalar type T is T'left.
- Can be declared in concurrent descriptions only.

Examples

--signals with single drivers
signal Clk1, Clk2 : time;
signal vector_5 : bit_vector (1 **to** 5);
signal index : integer := -127;

Signals can be given default (initial) values. However, these initial values are not used for synthesis.

Entity declaration

entity identifier **is**

entity header

[**generic** (interface constant declaration {; interface constant declaration});]

[**port** (interface signal declaration {; interface signal declaration});]

entity declarative part

subprogram declaration, subprogram body, type and subtype declaration, constant and signal declaration, use clause…, but *no variable declaration.*

All the declarations are visible throughout any architecture of the entity

What is declared in an architecture is not visible to another architecture of the same entity.

[**begin**

concurrent assertion statement

passive concurrent procedure call

passive process statement

]

end [entity simple name] ;

Entity header

The entity header declares objects used for communication between a design entity and its environment

Generics allow static information to be communicated to a block from its environment. They may be used to specify:

- Number of inputs to a component.
- Number of subcomponents within a block.
- Timing characteristics.
- Physical characteristics of a design: temperature, capacitance, etc.

Ports provide channels for communication between an entity (or a block in general) with its environment.

[**signal**] identifier{, identifier}:[mode] subtype indication [**bus**] [:= expression]

- Modes: in (default mode), out, inout, buffer, linkage
- Only signals can be in entity ports (no variables, files or access types).
- The expression defines the default value of the signal.

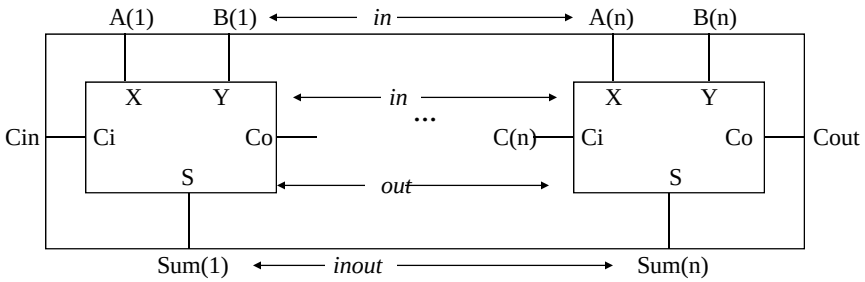
Modes

- in:
  - The value of the signal may only be read.
  - Any attribute may be read.
- out
  - The value of the signal may be updated.
  - Any attribute may be read except: STABLE, QUIET, DELAYED, TRANSACTION, EVENT, ACTIVE, LAST_EVENT, LAST-ACTIVE, and LAST_VALUE.
- inout:
  - The value of the signal may be read and updated.
  - Any attribute may be read.

Mode association

Possible associations

Formal	Actual
in	in, inout
out	out, inout
inout	inout



Entity declaration (examples)

```
entity Full_Adder  is          -- an one-bit adder
port ( X, Y: in Bit; Ci : in Bit := '0' ;
       S, Co :  out Bit );
end Full_Adder;

entity N_BIT_ADDER is        --   an n-stage adder
generic (n: integer := 8);
port ( A, B: in Bit_Vector (1 to n) ;
       Cin : in Bit := '0' ;
       Sum : inout Bit_Vector (1 to n);
       Cout : inout Bit );
begin
       assert (n>1 and n <= 20) report " operands oversized !" severity error;
       assert ( Cout  /= '1') report "Addition ended with a carry "
       severity WARNING;
       Check_Result ( A, B, Cin, Sum, Cout );
end ;
```


Architectures

- A design entity has a unique interface (entity decalaration), and one or more architectures.
- An architecture body defines the body of the design entity.
- It specifies the relationship between the input and the outputs

  of the design entity.

architecture identifier **of** entity name **is**

architecture declarative part

subprogram declaration, subprogram body, type and subtype declaration, constant and signal declaration, use clause…, but *no variable declaration.*

What is declared in an architecture is not visible to another architecture of the same entity.

begin

architecture statement part {concurrent statements}

end [architecture simple name] ;

Expressions

Operators

Classes of operators listed in order of increasing precedence.

- Logical: **and or nand nor xor**
- relational = /= < <= > >=
- addition + - &
- signs + -
- multiplication * / **mod rem**
- miscellaneous ** **abs not**

Operands

- Literals
- Aggregates

Qualified expressions

Type conversions

Operators

Logical operators

Operands:

- BIT or BOOLEAN
- One dimensional arrays which have elements of type BIT or BOOLEAN
- The arrays must have the same length.

Relational operators

The result is BOOLEAN.

For = and /= operators the operands may be of any type.

For < , <= , > , >= operators the operands may be of any scalar type or discrete array type.

Addition operators:

+,-: numeric types

&: concatenation between 2 arrays of the same type or an arrays and an element.

Examples of logical expressions

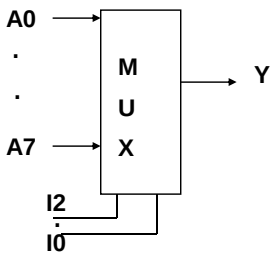
```
Signal A,B,C,D: BIT_VECTOR (7 downto 0);
Signal E,F,G,H: BIT_VECTOR (3 downto 0);
Signal X,Y, Z: BIT;
Signal Alpha,Beta, Gamma: LOGICAL;
A <= B or C;
X <= (Y and Z) nor X
Gamma <= Alpha xor Beta
D <= A and H --illegal
```

Straightforward synthesis

Arithmetic operators

- &, +, -, unary - : supported
- * :supported
- /, mod, rem: right must be power of two
- Indexing:

```
Signal A: bit_vector (0 to 7);
Signal I: integer range 0 to 7;
Signal Y: bit;
…
Y<= A(I);
```



Aggregate

([choice { | choice} =>] expression {,[choice { | choice} =>] expression)

A choice is a simple expression, a discrete range, an element simple name, or others.

Example:

```
type Switch is
record
      X : Integer;
      Y : Time;
end record;

signal S :array (7 downto 1) of Switch :=
      (1 | 4 to 5 | 7 => ( 1,  4 ns),
      others => (X => 2, Y => 6 ns));

signal SP:array (7 downto 1) of Switch :=
      ((1, 4 ns), ( 2, 6 ns), (1, 4 ns), (1, 4 ns), ( 2, 6 ns), ( 2, 6 ns),(1, 4 ns));
```


Qualified expressions

- type mark'(expression)
- type mark'aggregate

- A qualified expression is used to explicitly state the type (or subtype) of an expression or an aggregate.
- The value of the qualified expression is the value of the operand.
- The evaluation checks that the value of the operand belongs to the specified subtype.
- The operand must have the same type as the base type of the type mark.
- Examples:

  POSITIVE '(34*N +M) -- N and M must be integers

  STRING'("Success")
- Useful for alleviating ambiguity, when operators or subprograms are overloaded

Type conversions

- A type conversion provides for explicit conversion between closely related types.
- type conversion
  type mark (expression)
- The operand of a type conversion cannot be:
  - **null**,
  - an allocator (used for dynamic allocation of variables),
  - an aggregate,
  - a string literal.
- If the type mark denotes a subtype, conversion consists of:
  - conversion to the base type, and
  - a check that the result belongs to the subtype.

Closely related types (for conversion)

- Abstract numeric types
  The operand can be of any integer or floating point type.
- Array types
  The conversion is allowed if the target and operand types:
  - have the same dimensionality;
  - for each index position, the index types are:
    - the same or
    - convertible to each other
  - the element types are the same.

Examples of type conversion

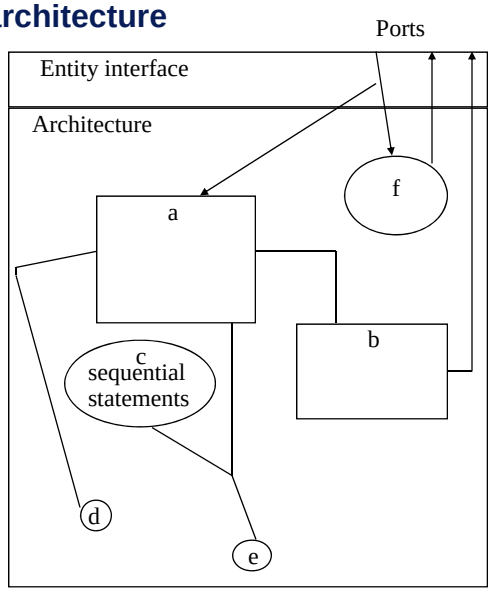
```
INTEGER (3.1459)          -- value 3
INTEGER (3.5)             -- implementation dependent 3 or 4
INTEGER (3.6)             -- value 4.
…
type table  is array(positive range 1 to 8) of bit;
type rev_table is array ( positive range 8 downto 1) of bit;
variable  tabinc: table;
variable tabdec: rev_table;
…
      tabinc <= X"0F";                  -- tabinc (1 to 8) ="0F"
      tabdec <= rev_table(tabinc);      -- tabdec (8 downto 1) = "0F"
 …
```


Sequential statements

- Used to define algorithms for subprograms or processes
- Execute in the order in which they appear.

  - Variable assignment statement
  - If statement
  - Case statement
  - Loop statement
  - Exit statement
  - Next statement
  - Null statement
  - Signal assignment statement
  - Procedure call statement
  - Return statement
  - Wait statement (see Section 3)
  - Assertion statement (see Section 3)

A typical architecture



a,b: instances of components

c,d,e, f: processes

SR Latch
(Example of variable and signal assignments)

```
entity SR_Latch  is
port ( S,R: in Bit; Q, QB: out Bit );
begin
     assert (S=0 or R=0)  report "S = R = '1' " severity error;
end SR_Latch;
```

```
architecture behave of SR_Latch is
begin
     process (S, R)
     variable state: Bit;
     begin
          If  S /= R  then
                state := S;                    -- variable assignment
          end if;
          Q <= state after 2 ns;               -- signal assignment
          QB <= not state after 2 ns;
     end process;
end behave ;
```

Architecture (Example)N_XOR gate (Illustration of loop statement)

```
entity N_XOR_Gate  is
generic
    (N: positive := 2);
 port
    ( Inputs: in Bit_Vector (1 to N);
      Result : out Bit );
end N_XOR_Gate ;
```

```
architecture behave of  N_XOR_Gate  is
begin
    process (inputs)
    variable Temp: Bit;
    begin
        Temp := '0';
        for i in Inputs'Range loop
            Temp := Temp xor Inputs(i);
        end loop;
        Result <= Temp after 10 ns;
    end process;
end behave ;
```


If statement (general)

```
-- Number of ones in the binary representation of X of range 0 to 7.
-- Even parity of X.

parity := 0;
if X = 0 then
      nb_of_ones := 0;
elsif X=7 then
      nb_of_ones := 3;
      parity := 1;
elsif X=1 or X= 2 or X=4 then
      nb_of_ones := 1;
      parity := 1;
else
      nb_of_ones := 2;
end if;
```

Case statement

```
-- Number of ones in the binary representation of X of range 0 to 7.
-- Even parity of X.
parity := 0;
case X is
    when 0          =>  nb_of_ones := 0;
    when 7          =>  nb_of_ones := 3;
                        parity := 1;
    when 1 | 2 | 4  =>  nb_of_ones := 1;
                        parity := 1;
    when others     =>  nb_of_ones := 2;
end case;
```

See also FSM modeling

Loop and Exit statements

[loop label :]
[iteration scheme] **loop**
sequence of statements
end loop [loop label] ;

iteration scheme
- while condition
- for identifier in discrete range

An exit statement is used to complete the execution of the innermost enclosing loop statement.

```
I:=0;
while ((T2(I)/='0') and (I<=131)) loop

    T2 (I) := 2 * I;
    T3 (I) := T3(I) + 1;
    I := I + 1;
end loop;
```

```
I:=0;
loop
    exit  when T2(I)='0' or I>131 ;
    T2 (I) := 2 * I;
    T3 (I) := T3(I) + 1;
    I := I + 1;
end loop;
```

Null statement

The execution of a null statement has no effect

Example: Rendezvous element:

X	RES
0	0
1	no change
2	no change
3	1

```
Signal RES: Bit := 0;     -- initial value
      …
case X is
      when 0          =>   RES <= '0';

      when 3          =>   RES <= '1';
      when others     =>   null;
end case;
```


Projected output waveform

signal assignment statement
target <= [**transport**] waveform element {, waveform element};

waveform element
- value expression [**after** time expression]
- **null** [**after** time expression] -- guarded signal (see session 5)

If after clause missing "after 0 ns" is assumed (one delta delay).

Time expression cannot be negative.

null assignment turns the driver of the signal off.

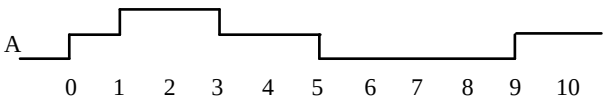
Updating a projected output waveform

- All old transactions that are projected to occur at or after the time at which the earliest new transaction is projected to occur are deleted from the projected output waveform.
- The new transactions are then appended to the projected output waveform.

  if the word transport is absent, inertial delay is assumed, and more processing steps are necessary:
- New transactions are marked.
- An old transaction is marked if:
  - it precedes a marked transaction, and,
  - its value is the same as the marked transaction.
- The transaction that determines the current value is marked.
- All unmarked transaction are deleted.

Examples of projected output waveform

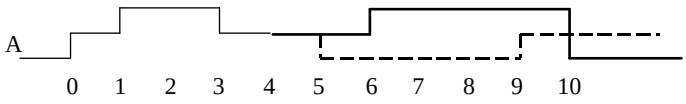
A <= 1, 2 **after** 1 ns, 1 **after** 3 ns, 0 **after** 5 ns, 1 **after** 9 ns;



1*
-
2 at 1 ns
-
1 at 3 ns
-
0 at 5 ns
-
1 at 9 ns

Examples of projected output waveform (cont'd)

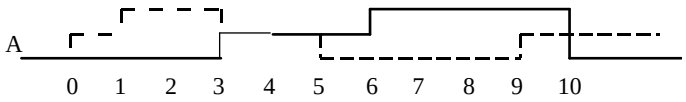
```
process   -- transport delay
begin
    A <= transport 1,  2  after 1 ns, 1 after 3 ns, 0 after 5 ns, 1 after 9 ns;
    A <= transport 1 after 4 ns, 2 after 6 ns, 0 after 10 ns;
    wait;
end process;
```



1*
2 at 1 ns
1 at 3 ns
~~0 at 5 ns~~ 1 at 4 ns
~~1 at 9 ns~~ 2 at 6 ns
0 at 10 ns

Examples of projected output waveform (cont'd)

```
process      -- inertial delay
begin
    A <=    1,  2  after 1 ns, 1 after 3 ns, 0 after 5 ns, 1 after 9 ns;
    A <=    1 after 4 ns, 2 after 6 ns, 0 after 10 ns;
    wait;
end process;
```



~~1*~~
~~2 at 1 ns~~
1 at 3 ns
~~0 at 5 ns~~
~~1 at 9 ns~~
1 at 4ns
2 at 6 ns
0 at 10 ns

Multiple assignments to a signal in a process

```
signal X : Bit_vector (0 to 1);
…
process ;
begin
    X <= "11" after 1 ns;
    X <= "01" after 2 ns;
    X <= "10" after 3 ns;
    X <= "00" after 4 ns;
    wait;
end process;
...
```

All assignments are executed but only the last one takes effect due to inertial delay.
If transport delay used then the waveform is reproduced.

Subprograms

- Subprogram declaration
  Formal parameters
  - Constant and variable parameters
  - Signal parameters
- Subprogram body
- Subprogram call
- Subprogram overloading
- Operator overloading

Subprograms

- **procedure** identifier [(formal parameter list)];
- **function** designator [(formal parameter list)] **return** type mark;
  The designator is either an identifier or an operator symbol
- A procedure call is a statement.
- A function call is an expression and returns a value.
- The definition of a program can be given in two parts:
  - A declaration defining its calling conventions
  - A subprogram body defining its execution.
- All subprograms can be called recursively.

Formal parameters

formal parameters

- may be constants, variables or signals for procedures
- may be only constants or signals for functions
- No global objects can be modified inside a function: the mode is always in.
- Mode: in, out, inout for procedures.
- variable is the default class for modes: out, inout.
- constant is the default class for mode in.

syntax

[**constant**] identifier list: [**in**] subtype indication [:= static expression]

[**signal**] identifier list: [mode] subtype indication [**bus**] [:=static expression]]

[**variable**]identifier list : [mode] subtype indication [:= static expression]

Subprogram body

procedure body
procedure identifier [(formal parameter list)] **is**
{subprogram declarative item} -- no signal declaration
begin
{sequential statement} -- no return statement
end [identifier] ;

function body
function designator [(formal parameter list)] **return** type mark **is**
{subprogram declarative item} -- no signal declaration
begin
{sequential statement} -- must contain: return [expression]
end [designator] ;

Examples of subprograms

```
function W_AND ( INPUT : BIT_VECTOR ) return  BIT is
variable RESULT : BIT := '1';
 -- pragma resolution_method
 wired_and
Begin
-- The code in this function is ignored by RTL synthesis but
parsed for correct syntax
    for I in INPUT'low to INPUT'high loop        -- for I in INPUT'range loop
        if INPUT ( I ) = '0' then
            RESULT := '0';
            exit;
        end if;
    end loop;
    return RESULT ;
end W_AND;
```

Signal parameters

- During the execution of a subprogram, a reference to a formal within an expression is equivalent to a reference to the associated actual signal.
- An assignment to the driver of a formal signal is equivalent to an assignment to the driver of the actual signal.
- Attributes STABLE, QUIET, DELAYED of a formal signal parameter cannot be read within a subprogram, however, EVENT, ACTIVE, LAST_EVENT, and LAST_ACTIVE are available since they are functional attributes of the formal parameters.

Subprogram overloading

- A given subprogram designator is overloaded if it is used in several subprogram specifications.

```
procedure count (C: Character;  S: String; NB: inout INTEGER);
    --    counts the number of occurrences of C in S.
procedure count (B: BIT; V: BIT_vector; NB: inout INTEGER);
    --    counts the number of occurrences of B in V.
procedure check (setup:Time; signal D: Data);
procedure check (hold:Time; signal D: Data);

count ('a',  message, NB);    --    correct implementation determined
count ('1', V1,NB);           --    from parameter types
check (setup => 10 ns, D => DL);
check (10 ns, DL);            -- ambiguous
-- See 6.13-6.14 for other examples of subprogram overloading
```

Operator overloading

```
type logic4 is ('X','0','1','Z');
…
      function "and"(a,b: logic4) return logic4 is
      type tablogic4 is array (logic4,logic4) of logic4;

      constant table:tablogic4:=
            (('X','0','X','X'),
             ('0','0','0','0'),
             ('X','0','1','X'),
             ('X','0','X','Z'));

      begin
         return table(a,b);
      end "and";
…
signal P, Q : logic4;
signal A : Bit;
…
P <= "and" ('0', 'Z');
Q <= '1' and 'Z';    -- invocation of the " logic4" and.
A <= '1' and '0';    -- invocation of the "Bit" and since A is of type BIT.
```

	X	0	1	Z
X	X	0	X	X
0	0	0	0	0
1	X	0	1	X
Z	X	0	X	Z

Adding bit vectors

```
function "+" ( bv1, bv2 : in bit_vector ) return bit_vector;
…
 function "+" ( bv1, bv2 : in bit_vector ) return bit_vector is

   alias op1 : bit_vector(bv1'length - 1 downto 0) is bv1;
   alias op2 : bit_vector(bv2'length - 1 downto 0) is bv2;
   variable result : bit_vector(bv1'length - 1 downto 0);
   variable carry_in : bit;
   variable carry_out : bit := '0';
```

Adding bit vectors (cont'd)

```
begin
    assert(not( bv1'length /= bv2'length))
    report """+"": operands of different lengths"
    severity failure;
  --- else
    for index in result'reverse_range loop
      carry_in := carry_out;  -- of previous bit
      result(index) := op1(index) xor op2(index) xor carry_in;
      carry_out := (op1(index) and op2(index))
                   or (carry_in and (op1(index) xor op2(index)));
    end loop;
  --- end if;
   return result;
 end "+";
```