Filename=”ch3.doc”

# 3.0 PACKAGES AND LIBRARIES

Shareable designs among users are important features of VHDL. That is how can you make a design be visible or available to others users.

## 3.1 Scope and Visibility

### 3.1.1 Scope

Range of identifier's name appearing in declaration.
Extends from beginning of declaration to END of region declaration appears in.
Limits:

design file
entity declaration
architecture
block
process
subprogram

### 3.1.2 Visibility

Establishes meaning of identifiers.
Declarations are visible only within their scope.
Declarations may be hidden by intervening declarations of identifier

Example:

```
ARCHITECTURE
        CONSTANT x:INTEGER:=2;
BEGIN
        PROCESS
                VARIABLE x:INTEGER:=3;
        BEGIN
                x  -- refer to variable x
        END
                x     -- refer to constant x
END
```

### 3.1.3 Selected Names

Declarations which are not visible in a particular region of their scope can still be used.
A selected name establishes the location of the identifier's declaration.
Format:
statement_label.identifier_name

Example1:

```
ARCHITECTURE example OF selected_name IS
        CONSTANT x:INTEGER:=2;
BEGIN
     p1: PROCESS(in_1)
```

```
                VARIABLE x:INTEGER:=3;
        BEGIN
                out_1 <= example.x*in_1;        -- refers to constant x
                out_2 <= p1.x*in_1;             -- refers to variable x
                out_5 <= x*in_1;                -- refers to variable x
        END PROCESS p1;
END example;
```

Example2:

```
        -- a block declaration
    B1: BLOCK
                --containing a signal declaration
                SIGNAL SUM, I1,I2:BIT;
        BEGIN
                --a nested block declaration
            B2: BLOCK
                -- This SUM hides the one in block B1.
                SIGNAL SUM: BIT:
                BEGIN
                        SUM <= I1 AND I2 AFTER 4 ns;  -- refers to SUM within B2,
                END BLOCK B2;
                SUM <= I1 AND I2 AFTER 4 ns;          -- refers to SUM in B1.
        END BLOCK B1;
```

## 3.2 Packages

- Shareable collections of declaration made outside models.
- Allow common use of:
  - Subprograms
  - Types
  - Subtypes
  - Constants
  - Signals
  - Aliases **
  - Attributes
  - Components
  - Disonnection Specification
- Consist of:
  - declaration
  - body

** An object, an indexed part of it, or a slice of it can be given alternative names by using an alias declaration. For example,

```
ALIAS c_flag: BIT IS flag_register(3)
```

### 3.2.1 Package Declaration

- Describe identifiers declared in package for use outside the package.
- Package must be compiled to be accessible.
- Only declarations made in the package declaration are visible outside the package. Declarations made in the package body are not visible unless they also appear in the package declaration.

- Package declarations can exist without a body. This is useful when the package consists only of declarations which in themselves do not require bodies. An example would be package which contains only type declarations.
- Format:

```
PACKAGE package_name IS
        -- declarations
END package_name;
```

- Allowable identifier declarations:
  - Subprograms
  - Types
  - Subtypes
  - Constants
  - Signals
  - Aliases
  - Attribute Specifications
  - Component Specifications
  - Disconnection Specifications
  - USE clause

Example:

```
PACKAGE example IS
        TYPE tj IS (`T','J');
        CONSTANT pi: REAL := 3.14159;
        FUNCTION mean(a,b,c:REAL) RETURN REAL;
END example;
```

### 3.2.2 Package Body

- Describes implementation of subprograms and deferred constants.
- Packages require bodies only if declaring subprograms and/or deferred constants.
- Format:

```
PACKAGE BODY package_name IS
        -- declarations
END package_name;
```

- Package_name must be same as name used in package declaration.
- Declarations within body are not visible unless stated in package declaration.

Example:

```
PACKAGE BODY example IS
        FUNCTION mean(a,b,c:REAL) RETURN REAL IS
        BEGIN
                RETURN (a+b+c)/3.0;
        END
END example;
```

#### 3.2.2.1 Deferred Constants

- Appear only in packages.
- Allow declaration of constant without specification of value.
- Constant appears without value in package declaration.
- Full constant declaration must appear in package body.

- Deferred constants might be useful when one model is used to model several different technologies. In these cases by simply changing the body of the package the constants for the different technology can be implemented. This would save time and effort when different technologies were being examined.
- The value of the constant can be the result derived from the evaluation of an expression.

Example:

```
--package declaration
PACKAGE example IS
        CONSTANT constantly :INTEGER;
END example;
--package body
PACKAGE BODY example IS
        CONSTANT constantly:INTEGER :=5;
END example;
```

### 3.2.3 Using Packages

- Identifiers declared in package are not automatically visible to design entities.
- USE makes identifiers in packages visible.
- Format:
  USE package_name.identifier_name;

Example:
  USE example.tj, example.mean:

- To specify use of all declarations made in package use .ALL for identifier_name.
  USE example.ALL; -- is equivalent to the above USE statement.

## 3.3 Libraries

- Provide storage for compiled design units.
- Units:
  primary:
    entity declarations
    package declarations
    configuration declarations
  secondary
    package bodies
    architecture bodies
- Classifications:
  working library
  resource library
- Primary units must exist and be compiled before their associated secondary unit can be compiled.

### 3.3.1 Working Library

- Corresponds to working directory.
- Named "work".
- Units in working library are directly accessible.
- Identifiers within packages in working library are made visible with USE

```
    USE example.ALL;
Or  USE work.example.ALL;
```

### 3.3.2 Resource Libraries

- Offer additional storage for compiled design units.
- Name is logical name and must be mapped to physical structure.
- Units in resource libraries are not directly accessible.

### 3.3.3 Accessing Libraries

- LIBRARY statement identifies logical name of library to be accessed.
- USE statement makes design units within library accessible.
- Format:
  LIBRARY library_name;
  USE library_name.package_name;
- Identifiers within packages within a library are accessed with a second USE statement.

Example:

```
LIBRARY things;
USE things.func_pkg;
USE func_pkg.ALL;
-- (alternative)
LIBRARY things;
USE things.func_pkg.ALL;
```

- Note that a LIBRARY clause does not have to be included in the model if the package is in the working directory.

Example:
assume the package example is in the working directory

```
USE work.example.ALL;
```

- The package STANDARD is in LIBRARY STD. When we use items from this library (bit, integer, etc.), we do not need to use LIBRARY and USE clauses. Since the package standard is automatically linked to all models, this is a requirement of the IEEE standard.

## 3.4 Subprograms

- Sequence of statements which can be executed from multiple locations in model.
- Provide method for breaking large segments of code into smaller segments.
- Types of subprograms:
  (1) Functions - used to calculate and return one value.
  (2) Procedures - used to define an algorithm which affects many values or no values.

### 3.4.1 Functions

Provides a method of performing complex algorithms.
Produce a single return value.
Invoked by an expression.
May not effect parameters
May (not MUST) consist of two parts:

* Function declaration
* Function Body

The function declaration is required only when the function body appears in a lower structural level than the level it is being called from. For example, if a function body appears in a process, it cannot be called in another process unless a function declaration occurs at the architectural level. If the body occurs at the same level as the function call, the declaration is not needed, and the function may consist of only one part (the body).

### 3.4.1.1 Function Declaration

- Specifies: Function name, Input parameters, Type of return value.
- Must:
  (1) Exist for recursive functions.
  (2) Exist when body occurs at level below calling level.
  (3) Occur at or above level of use.
  (4) Appear in statement declaration areas.
- Format:
  FUNCTION name(parameter_list) RETURN type;

Example:

```
FUNCTION exam(SIGNAL signal_a:BIT) RETURN BIT;
```

### 3.4.1.2 Function Body

- Defines how function behaves.
- Consists of sequential statements.
- Placed in declrative statement areas.
- Format:

```
FUNCTION name(parameter_list) RETURN type IS
  -- declarative_statements
BEGIN
  -- sequential_statements
  -- including RETURN expression
END name;
```

### 3.4.1.3 Funcion Parameter List

- Identifies class, name, mode and type of objects which contain values passed to function.
- Format:
  CLASS object_name: MODE type
- Object CLASS may be SIGNAL or CONSTANT only. CONSTANT is the default class.
- Mode is limited to IN (is the default mode).
- Type of objects must correspond to type of values passed.

Example:

```
FUNCTION equal (a,b:IN BIT) RETURN BOOLEAN IS
BEGIN
  RETURN a = b;
END equal;
```

### 3.4.1.4 Function Calls

- Stated as expressions.
- Cause execution of function body.
- Specify passing parameters.
- Format:
  function_name(passing_parameter_list)

Example:

```
IF transfer_check(array_1,array_2)
```

- A function may be called from concurrent statements areas and sequential statement areas.

### 3.4.1.5 Passing Parameter List

- Specifies objects from which values pass to function
- Positional association:
  Parameters in call match order of parameters in function parameter list.

Example:

```
FUNCTION tran_ck(SIGNAL a:IN BIT;SIGNAL b:IN BIT) RETURN BOOLEAN;

IF tran_ck(signal_a,signal_b) ..
```

- Named association:
  Relates parameters in call by name to parameters in function list

Example:

```
FUNCTION tran_ck(SIGNAL a:IN BIT;SIGNAL b:IN BIT) RETURN BOOLEAN;

IF tran_ck(b => signal_b, a => signal_a)..
```

### 3.6.2 Procedures

- Provide a method of performing complex algorithms.
- May produce multiple output values.
- May affect input parameters.
- Are invoked by a statement.
- May consist of two parts:
  Procedure Declaration
  Procedure Body
- Major difference between functions and procedures:
  Procedures may affect the values of the passing parameters and the functions may not.
  Procedures may produce multiple output values while functions return only one value.

### 3.6.2.1 Procedure Declaration

- Specifies: Procedure name, Parameters
- Must

Exist for recursive procedures.
Exist when body occurs below calling level in structure hierarchy.
Occur at or above level of call.
Appear in statement declaration areas.

- Format:
  PROCEDURE name(parameter_list)

Example:

```
PROCEDURE example(VARIABLE a:IN BIT;SIGNAL b:OUT BIT);
```

### 3.6.2.2 Procedure Body

- Defines how procedure behaves.
- Consists of sequential statements.
- Placed in declrative statement areas.
- May read, create, or modify parameters.
- Must exit at or above the level of use if procedure declaration is not used in model.
- If a procedure is declared in the ENTITY declaration, it can be called from any structural level within the architectural body.
- Format:

```
PROCEDURE name(parameter_list) IS
  -- declarative_statements
BEGIN
  -- sequential_statements
END name;
```

### 3.6.2.3 Procedure Parameter List

- Identifies class, name, mode, and type of objects which contain values passed to and from procedure.
- Mode determines what actions can be taken on parameter.
- Object class may be: CONSTANT(default for MODE IN), SIGNAL, or VARIABLE(default for MODE INOUT or OUT).
- Mode may be: IN(default), OUT, or INOUT.
- Format:
  CLASS object_name: MODE type

Example:

```
PROCEDURE a(VARIABLE a: IN REAL);
PROCEDURE b(SIGNAL a:BIT);
PROCEDURE c(a:BIT);  -- a is a CONSTANT of MODE IN
PROCEDURE d(d:OUT BIT);  -- d is a variable
```

### 3.6.2.4 Procedure Calls

- Cause execution of procedures.
- Can be used in sequential or concurrent statement areas.
- Specify objects whose values are passed to parameters in procedure.
- Concurrent calls are activated by changes in any signal associated with a parameter of mode IN or INOUT. In a sense, the parameter list is like a process sensitivity list.
- Sequential procedure call is executed whenever it is encounter during the execution of the sequential code.
- Format:

```
procedure_name(passing_parameter_list);
```

Example:

```
find_min(array_1,out_1,out_2);
```

### 3.6.2.5 Passing Parameter List

- Specifies objects from and to which values pass.
- Association by:
  - Position
  - Name
- Values associated with IN parameters are "protected" from change.

**Example 1:** inverter

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY invert_example IS
        PORT(in_a:IN std_logic;out_b:OUT std_logic);
END inverter_example;
ARCHITECTURE behav OF invert_example IS
        PROCEDURE inverter (SIGNAL bit_in:IN std_logic;
        SIGNAL bit_out:OUT std_logic) IS
        BEGIN
        IF bit_in = `1' THEN
                bit_out <= `0';
        ELSIF bit_in = `0' THEN
                bit_out <= `1';
        ELSE
                bit_out <= `X';
        END IF;
        END inverter;

BEGIN
        inverter(in_a,out_b);
END behav;
```

**Example 2:** General rotate right procedure for any array size --using unconstrained array

```
PROCEDURE rotate(a: INOUT BIT_VECTOR) IS
BEGIN
                IF a'RIGHT < a'LEFT THEN
                        a<=a(a'RIGHT) & a(a'LEFT DOWNTO a'RIGHT+1)
                ELSE
                        a<=a(a'RIGHT) & a(a'LEFT TO a'RIGHT - 1)
                END IF;
END;
```

**Sample invocations:**

```
SIGNAL a:BIT_VECTOR(7 DOWNTO 0);
```

rotate(a) -- a <= a(0) & a(7 DOWNTO 1)
SIGNAL a: BIT_VECTOR(0 TO 15)
rotate(a) -- a<= a(15) & a(0 TO 14)

## 3.5 Package Examples

| resolution | 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 | - |
| **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 |

| AND | U | X | 0 | 1 | Z | W | L | H | - |
|---|---|---|---|---|---|---|---|---|---|
| **U** | U | U | 0 | U | U | U | 0 | U | U |
| **X** | U | X | 0 | X | X | X | 0 | X | X |
| **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **1** | U | X | 0 | 1 | X | X | 0 | 1 | X |
| **Z** | U | X | 0 | X | X | X | 0 | X | X |
| **W** | U | X | 0 | X | X | X | 0 | X | X |
| **L** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **H** | U | X | 0 | 1 | X | X | 0 | 1 | X |
| **-** | U | X | 0 | X | X | X | 0 | X | X |

| OR | U | X | 0 | 1 | Z | W | L | H | - |
|---|---|---|---|---|---|---|---|---|---|
| **U** | U | U | U | 1 | U | U | U | 1 | U |
| **X** | U | X | X | 1 | X | X | X | 1 | X |
| **0** | U | X | 0 | 1 | X | X | 0 | 1 | X |
| **1** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| **Z** | U | X | X | 1 | X | X | X | 1 | X |
| **W** | U | X | X | 1 | X | X | X | 1 | X |
| **L** | U | X | 0 | 1 | X | X | 0 | 1 | X |
| **H** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| **-** | U | X | X | 1 | X | X | X | 1 | X |

| XOR | 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 | 1 | X | X | 0 | 1 | X |
| **1** | U | X | 1 | 0 | X | X | 1 | 0 | X |
| **Z** | U | X | X | X | X | X | X | X | X |
| **W** | U | X | X | X | X | X | X | X | X |
| **L** | U | X | 0 | 1 | X | X | 0 | 1 | X |
| **H** | U | X | 1 | 0 | X | X | 1 | 0 | X |
| **-** | U | X | X | X | X | X | X | X | X |

| | U | X | 0 | 1 | Z | W | L | H | - |
|---|---|---|---|---|---|---|---|---|---|
| **NOT** | U | X | 1 | 0 | X | X | 1 | 0 | X |

/opt/cds/ldv41/tools.sun4v/inca/files/STD --location of std library
/opt/cds/ldv41/tools.sun4v/inc a/files/IEEE --location of IEEE Libraries

For Example to locate std_logic_arith package the source will be in
/opt/cds/ldv41/tools.sun4v/inca/files/IEEE/std_logic_arith/package/vhdl.vhd

**PACKAGE STD_LOGIC_1164 IS** --partial listing of IEEE STD_LOGIC_1164

```
TYPE std_ulogic IS (    'U'     --Uninitialized
                        'X'     --Forcing Unknown
                        '0'     --Forcing       0
                        '1'     --Forcing       1
                        'Z'     --High Impedance
                        'W'     --Weak          Unknown
                        'L'     --Weak          0
                        'H'     --Weak          1
                        '-'     --Don't care
                        );

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

FUNCTION resolved (s: std_ulogic_vector) RETURN std_ulogic;

SUBTYPE std_logic IS resolved std_ulogic;

TYPE std_logic_vector IS ARRAY ( NATURAL RANGE <>) OF std_logic;

SUBTYPE X01 IS resolved std_ulogic RANGE 'X' TO '1';        --('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')

--overloaded logical operators. All operators are declared identically, illustrated for "and"  only.

FUNCTION "and" (l : std_ulogic; r : std_ulogic) RETURN UX01;
FUNCTION "and" (l, r : std_logic_vector) RETURN std_logic_vector;
FUNCTION "and" (l, r : std_ulogic_vector) RETURN std_ulogic_vector;

--conversion functions between standard types to IEEE types

FUNCTION To_bit( s : std_ulogic; xmap : BIT:='0') RETURN BIT;
FUNCTION To_bitvector( s : std_logic_vector; xmap : BIT :='0') RETURN BIT_VECTOR;
FUNCTION To_bitvector( s : std_ulogic_vector; xmap : BIT :='0') RETURN BIT_VECTOR;

FUNCTION To_StdULogic( b: BIT) RETURN std_ulogic;
FUNCTION To_StdULogicVector ( b : BIT_VECTOR) RETURN std_ulogic_vector;
FUNCTION To_StdULogicVector( b: std_logic_vector) RETURN std_ulogic_vector;

--edge detection

FUNCTION rising_edge (SIGNAL s : std_ulogic) RETURN BOOLEAN;
FUNCTION falling_edge (SIGNAL s : std_ulogic) RETURN BOOLEAN;
```

```
END std_logic_1164;

PACKAGE BODY std_logic_1164 IS

        TYPE std_logic_table IS ARRAY (std_ulogic, std_ulogic) OF std_ulogic;



        CONSTANT and_table : std_logic_table :=(
--          U    X    0    1    Z    W    L    H    -
        (‘U’, ‘U’, ‘0’, ‘U’, ‘U’, ‘U’, ‘0’, ‘U’, ‘U’ ),       --U
        (‘U’, ‘X’, ‘0’, ‘X’, ‘X’, ‘X’,  ‘0’, ‘X’, ‘X’),       --X
        (‘0’,  ‘0’,  ‘0’, ‘0’,  ‘0’,  ‘0’,  ‘0’, ‘0’,  ‘0’),       --0
        (‘U’, ‘X’, ‘0’, ‘1’,  ‘X’, ‘X’,  ‘0’, ‘1’,  ‘X’),       --1
        (‘U’, ‘X’, ‘0’, ‘X’, ‘X’, ‘X’, ‘0’, ‘X’, ‘X’),       --Z
        (‘U’, ‘X’, ‘0’, ‘X’, ‘X’, ‘X’, ‘0’, ‘X’, ‘X’),       --W
        (‘0’,  ‘0’,  ‘0’, ‘0’,  ‘0’,  ‘0’,  ‘0’, ‘0’,  ‘0’),       --L
        (‘U’, ‘X’, ‘0’, ‘1’,  ‘X’, ‘X’,  ‘0’, ‘1’,  ‘X’),       --H
        (‘U’, ‘X’, ‘0’, ‘X’, ‘X’, ‘X’, ‘0’, ‘X’, ‘X’),       -- -
        );

        FUNCTION “and” ( l : std_ulogic; r : std_ulogic) RETURN UX01 IS
        BEGIN
                RETURN(and_table(l, r));
        END “and”;

        FUNCTION “and” (l, r : std_logic_vector) RETURN std_logic_vector IS
                ALIAS lv : std_logic_vector(1 TO l’LENGTH) IS l;
                ALIAS rv : std_logic_vector(1 TO r’LENGTH) IS r;
                VARIABLE result : std_logic_vector( 1 TO l’LENGTH);
        BEGIN
                IF(l’LENGTH /= r’LENGTH) THEN
                        ASSERT FALSE
                        REPORT “arguments of “and” are not of the same length”
                        SEVERITY FAILURE;
                ELSE
                        FOR i IN result’RANGE LOOP
                                result(i) := and_table(lv(i), rv(i));
                        END LOOP;
                END IF;
                RETURN result;
        END “and”;

        FUNCTION “and” (l, r : std_ulogic_vector) RETURN std_ulogic_vector IS
                ALIAS lv : std_ulogic_vector(1 TO l’LENGTH) IS l;
                ALIAS rv : std_ulogic_vector(1 TO r’LENGTH) IS r;
                VARIABLE result : std_ulogic_vector( 1 TO l’LENGTH);
        BEGIN
                IF(l’LENGTH /= r’LENGTH) THEN
                        ASSERT FALSE
                        REPORT “arguments of “and” are not of the same length”
                        SEVERITY FAILURE;
                ELSE
                        FOR i IN result’RANGE LOOP
```

```
                result(i) := and_table(lv(i), rv(i));
            END LOOP;
        END IF;
        RETURN result;
    END "and";

-- resolution_table is entered as in the and_table with entry as shown in resolution table.
    FUNCTION resolved( s : std_ulogic_vector) RETURN std_ulogic IS
        VARIABLE result : std_ulogic := 'Z';        --weakest state
    BEGIN
        IF (s'LENGTH = 1) THEN RETURN s(s'LOW);
        ELSE
            FOR i IN s'RANGE LOOP
                result := resolution_table(result, s(i));
            END LOOP;
        END IF;
        RETURN result;
    END resolved;

    FUNCTION To_bit ( s : std_ulogic; xmap : BIT :='0') RETURN BIT IS
    BEGIN
        CASE s IS
            WHEN '0' | 'L' => RETURN ('0');
            WHEN '1' | 'H' => RETURN ('1');
            WHEN OTHERS => RETURN xmap;
        END CASE;
    END;

    FUNCTION To_bitvector ( s : std_logic_vector; xmap : BIT :='0') RETURN BIT_VECTOR IS
        ALIAS sv : std_logic_vector( s'LENGTH –1 DOWNTO 0) IS s;
        VARIABLE result : BIT_VECTOR( s'LENGTH-1 DOWNTO 0);
    BEGIN
        FOR i IN result'RANGE LOOP
            CASE sv(i) IS
                WHEN '0' | 'L' => result(i) :=  '0';
                WHEN '1' | 'H' => result(i) := '1';
                WHEN OTHERS => result(i) := xmap;
            END CASE;
        END LOOP;
        RETURN result;
    END;

    FUNCTION To_bitvector ( s : std_ulogic_vector; xmap : BIT :='0') RETURN BIT_VECTOR IS
        ALIAS sv : std_ulogic_vector( s'LENGTH –1 DOWNTO 0) IS s;
        VARIABLE result : BIT_VECTOR( s'LENGTH-1 DOWNTO 0);
    BEGIN
        FOR i IN result'RANGE LOOP
            CASE sv(i) IS
                WHEN '0' | 'L' => result(i) :=  '0';
                WHEN '1' | 'H' => result(i) := '1';
                WHEN OTHERS => result(i) := xmap;
            END CASE;
        END LOOP;
        RETURN result;
    END;
```

```
    FUNCTION To_StdULogic ( b : BIT) RETURN std_ulogic IS
    BEGIN
        CASE b IS
            WHEN '0' => RETURN '0';
            WHEN '1' => RETURN '1';
        END CASE;
    END;

    FUNCTION To_StdULogicVector( b : BIT_VECTOR) RETURN std_ulogic_vector IS
        ALIAS bv : BIT_VECTOR(b'LENGTH-1 TO 0) IS b;
        VARIABLE result : std_ulogic_vector (b'LENGTH-1 DOWNTO 0);
    BEGIN
        FOR i IN result'RANGE LOOP
            CASE bv(i) IS
                WHEN '0' => result(i) := '0';
                WHEN '1' => result(i) :='1';
            END CASE;
        END LOOP;
        RETURN result;
    END;

    FUNCTION To_StdULogicVector( s : std_logic_vector) RETURN std_ulogic_vector IS
        ALIAS sv : std_logic_vector(s'LENGTH-1 TO 0) IS s;
        VARIABLE result : std_ulogic_vector (b'LENGTH-1 DOWNTO 0);
    BEGIN
        FOR i IN result'RANGE LOOP
            result(i) := sv(i);
        END LOOP;
        RETURN result;
    END;

    FUNCTION rising_edge(SIGNAL s : std_ulogic) RETURN BOOLEAN IS
    BEGIN
        RETURN(s'EVENT AND (To_X01(s) = '1') AND (To_X01(s'LAST_VALUE)='0'));
    END;

    FUNCTION falling_edge(SIGNAL s : std_ulogic) RETURN BOOLEAN IS
    BEGIN
        RETURN(s'EVENT AND (To_X01(s) = '0') AND (To_X01(s'LAST_VALUE)='1'));
    END;
END std_logic_1164;
```

Example: rising_edge

```
    rising_edge(s);                 --if s: std_ulogic
    rising_edge(s);                 --if s: std_logic ; std_logic is resolved std_ulogic
    rising_edge(To_StdULogic(s));   --if s : BIT
```

Example: signal assignment with different data types.

```
    SIGNAL IO : std_logic_vector(7 DOWNTO 0);
    SIGNAL data : bit_vector(7 DOWNTO 0);

    IO <= To_StdLogicVector(data);
```

## 3.6 Creating your own Library and Package

The standard IEEE package does not have pre-defined operators for the standard data type bit and bit_vector. One can easily write their own package which will facilitate operating on the standard data type directly. The following show a sample of writing your own package. In order that it can be made available to others, one must give it a library name and where physically it is stored in disk directory.

## Creating a Resource Library in CADENCE

1. Add a line in the file : ~/cadence/vhdl/cds.lib or $CDSVHDL/cds.lib
   DEFINE MYLIB ~/cadence/mylib

2. Create a new folder "mylib" under ~/cadence by executing: (FILE>New Folder… enter "mylib"

3. Invoke nclaunch, then on the right window panel select MYLIB. Click RMB and select "**Set as Work Library**"

4. Then compile your package (mypack) source code, the compiled object code will be stored in the current work library ="MYLIB".

The users of this library must include the following lines in their VHDL code.

```
LIBRARY MYLIB;
USE MYLIB.mypack.ALL;
```

**Sample Package (=mypack) source code to be compiled into the resource library name MYLIB**

```
PACKAGE mypack IS
        function TO_INT(a:BIT_VECTOR) RETURN INTEGER;
        function TO_BIT_VEC(a,n:INTEGER) RETURN BIT_VECTOR;
END mypack;

PACKAGE BODY mypack IS

function TO_INT(a:BIT_VECTOR) RETURN INTEGER IS
        VARIABLE val:INTEGER:=0;
        ALIAS b:BIT_VECTOR(a'LENGTH-1 DOWNTO 0) IS a;
BEGIN
        FOR i IN b'HIGH DOWNTO 1 LOOP
                IF(b(i)='1') THEN
                        val:=(val+1)*2;
                ELSE
                        val:=val*2;
                END IF;
        END LOOP;
        IF b(0)='1' THEN
                val:=val+1;
        END IF;
        RETURN val;
END TO_INT;

function TO_BIT_VEC(a, n:INTEGER)RETURN BIT_VECTOR IS
        VARIABLE n1:INTEGER;
        VARIABLE val:BIT_VECTOR(n-1 DOWNTO 0);
BEGIN
        ASSERT a>=0
```

```
                REPORT "function TO_BIT_VEC: input integer cannot be negative"
                SEVERITY ERROR;
        n1=a;
        FOR i IN val'REVERSE_RANGE LOOP
                IF (n1 MOD 2)=1 THEN
                        val(i):='1';
                ELSE
                        val(i):='0';
                END IF;
                n1:=n1/2;
        END LOOP;
        RETURN val;
END TO_BIT_VEC;
END mypack;
```

## Example of using mypack

```
LIBRARY MYLIB;
USE MYLIB.mypack.ALL;

ENTITY binctr IS
        PORT(clk:IN BIT; q:INOUT BIT_VECTOR(3 DOWNTO 0));
END binctr;

ARCHIT
ECTURE arch_binctr OF binctr IS
BEGIN
        PROCESS(clk)
        BEGIN
                IF clk='1' THEN
                        q<=TO_BIT_VEC(TO_INT(q)+1,4);
                END IF;
        END PROCESS;
END arch_binctr;
```