Functions and Procedures

FUNCTIONS and PROCEDURES are collectively called subprograms. From a construction point of view, they are very similar to a PROCESS.

However, from the applications point of view, there is a fundamental difference between a PROCESS and a FUNCTION or PROCEDURE. While the first is intended for immediate use in the main code, the others are intended mainly for LIBRARY allocation.

FUNCTION

A FUNCTION is a section of sequential code.

Its syntax is shown below.

```
FUNCTION function_name [<parameter list>] RETURN data_type IS
    [declarations]
BEGIN
    (sequential statements)
END function_name;
```

There can be any number of parameters (even zero), can only be CONSTANT (default) or SIGNAL (VARIABLES are not allowed).

Their types can be any of the synthesizable data types studied (BOOLEAN, STD_LOGIC, INTEGER, etc.).

However, no range specification should be included (for example, do not enter RANGE when using INTEGER, or TO/ DOWNTO when using STD_LOGIC_VECTOR). On the other hand, there is only one return value, whose type is specified by data_type.

Example:

```
FUNCTION f1 (a, b: INTEGER; SIGNAL c: STD_LOGIC_VECTOR)
RETURN BOOLEAN IS
BEGIN
(sequential statements)
END f1;
```

The function below, named f1, receives three parameters (a, b, and c). a and b are CONSTANTS (notice that the word CONSTANT can be omitted, for it is the default object), while c is a SIGNAL. a and b are of type INTEGER, while c is of type STD_LOGIC_VECTOR.

Notice that neither RANGE nor DOWNTO was specified. The output parameter (there can be only one) is of type BOOLEAN.

Function Call

A function is called as part of an expression. The expression can obviously appear by itself or associated to a statement (either concurrent or sequential).

Example of function:

```
x <= conv_integer(a); -- converts a to an integer
```

Example: Function positive_edge( )

The FUNCTION below detects a positive (rising) clock edge. It is similar to the IF(clk'EVENT and clk ='1') statement. This function could be used, for example, in the implementation of a DFF.

------ Function body: ------------------------------

```
FUNCTION positive_edge(SIGNAL s: STD_LOGIC) RETURN BOOLEAN IS
BEGIN
        RETURN (s'EVENT AND s='1');
END positive_edge;
```

------ Function call: ------------------------------

```
...
IF positive_edge(clk) THEN...
...
```

Example: Function conv_integer( )

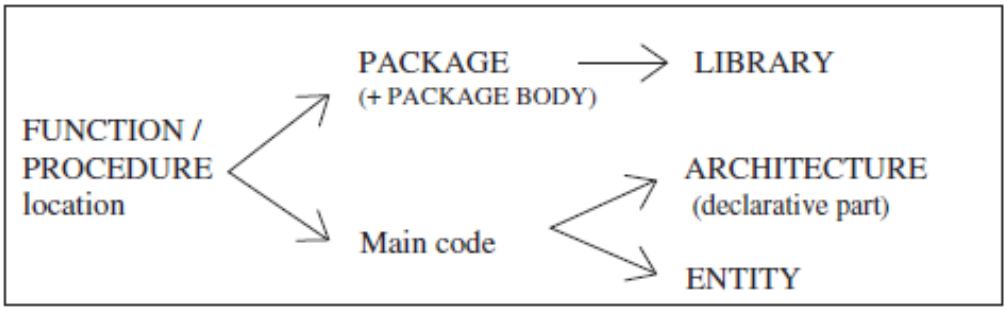
The FUNCTION presented next converts a parameter of type STD_LOGIC_VECTOR into an INTEGER. Notice that the code is generic, that is, it works for any range or order (TO/DOWNTO) of the input STD_LOGIC_VECTOR parameter. A typical call to the function is also shown.

```
------ Function body: -------------------------------
FUNCTION conv_integer (SIGNAL vector: STD_LOGIC_VECTOR)
        RETURN INTEGER IS
        VARIABLE result: INTEGER RANGE 0 TO 2**vector'LENGTH-1;
BEGIN
        IF (vector(vector'HIGH)='1') THEN result:=1;
        ELSE result:=0;
        END IF;
        FOR i IN (vector'HIGH-1) DOWNTO (vector'LOW) LOOP
                result:=result*2;
                IF(vector(i)='1') THEN result:=result+1;
                END IF;
        END LOOP;
RETURN result;
END conv_integer;
------ Function call: -------------------------------
...
y <= conv_integer(a);
...
-------------------------------------------------
```

Function Location

Though a FUNCTION is usually placed in a PACKAGE (for code partitioning, code reuse, and code sharing purposes), it can also be located in the main code (either inside the ARCHITECTURE or inside the ENTITY).

When placed in a PACKAGE, then a PACKAGE BODY is necessary, which must contain the body of each FUNCTION (or PROCEDURE) declared in the declarative part of the PACKAGE.



Example: FUNCTION Located in the Main Code

As mentioned above, when installed in the main code itself, the function can be located either in the ENTITY or in the declarative part of the ARCHITECTURE. In the present example, the function appears in the latter, and is used to construct a DFF.

```
1 ----------------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ----------------------------------------
5 ENTITY dff IS
6       PORT ( d, clk, rst: IN STD_LOGIC;
7                   q: OUT STD_LOGIC);
8 END dff;
9 ----------------------------------------
```

```
10 ARCHITECTURE my_arch OF dff IS
11 ----------------------------------------
12     FUNCTION positive_edge(SIGNAL s: STD_LOGIC)
13     RETURN BOOLEAN IS
14             BEGIN
15                     RETURN s'EVENT AND s='1';
16             END positive_edge;
17 ----------------------------------------
18 BEGIN
19     PROCESS (clk, rst)
20     BEGIN
21             IF (rst='1') THEN q <= '0';
22             ELSIF positive_edge(clk) THEN q <= d;
23             END IF;
24     END PROCESS;
25 END my_arch;
26 ----------------------------------------
```

Example: FUNCTION Located in a PACKAGE

```
1 ------- Package: ----------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ----------------------------------------
5 PACKAGE my_package IS
6       FUNCTION positive_edge(SIGNAL s: STD_LOGIC) RETURN
BOOLEAN;
7 END my_package;
8 ----------------------------------------
9 PACKAGE BODY my_package IS
10     FUNCTION positive_edge(SIGNAL s: STD_LOGIC)
11     RETURN BOOLEAN IS
12             BEGIN
13                     RETURN s'EVENT AND s='1';
14             END positive_edge;
15 END my_package;
```

```
16 ---------------------------------------------

1 ------ Main code: ---------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 USE work.my_package.all;
5 ---------------------------------------------
6 ENTITY dff IS
7               PORT ( d, clk, rst: IN STD_LOGIC;
8                           q: OUT STD_LOGIC);
9 END dff;
10 ---------------------------------------------
11 ARCHITECTURE my_arch OF dff IS
12      BEGIN
13              PROCESS (clk, rst)
14              BEGIN
15                      IF (rst='1') THEN q <= '0';
16                      ELSIF positive_edge(clk) THEN q <= d;
17                      END IF;
18              END PROCESS;
19 END my_arch;
20 ---------------------------------------------
```

Example: Function conv_integer( )

The conv_integer( ) function was placed in a PACKAGE (plus PACKAGE BODY). A call to this function appears in the main code that follows the function implementation.

```
1 --------- Package: ----------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ---------------------------------------------
5 PACKAGE my_package IS
6       FUNCTION conv_integer (SIGNAL vector: STD_LOGIC_VECTOR)
7 RETURN INTEGER;
8 END my_package;
9 ---------------------------------------------
```

```
10 PACKAGE BODY my_package IS

11      FUNCTION conv_integer (SIGNAL vector: STD_LOGIC_VECTOR)
12      RETURN INTEGER IS

13       VARIABLE result: INTEGER RANGE 0 TO 2**vector'LENGTH-1;
14      BEGIN
15              IF (vector(vector'HIGH)='1') THEN result:=1;
16              ELSE result:=0;
17              END IF;
18              FOR i IN (vector'HIGH-1) DOWNTO (vector'LOW) LOOP
19                      result:=result*2;
20                      IF(vector(i)='1') THEN result:=result+1;
21                      END IF;
22              END LOOP;
23      RETURN result;
24 END conv_integer;
25 END my_package;
26 ---------------------------------------------

1 -------- Main code: -------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 USE work.my_package.all;
5 ---------------------------------------------
6 ENTITY conv_int2 IS
7       PORT ( a: IN STD_LOGIC_VECTOR(0 TO 3);
8                 y: OUT INTEGER RANGE 0 TO 15);
9 END conv_int2;
10 ---------------------------------------------
11 ARCHITECTURE my_arch OF conv_int2 IS
12 BEGIN
13      y <= conv_integer(a);
14 END my_arch;
15 ---------------------------------------------
```

Example: Overloaded "+" Operator

```
1 -------- Package: ----------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 ------------------------------------------
5 PACKAGE my_package IS
6       FUNCTION "+" (a, b: STD_LOGIC_VECTOR)
7       RETURN STD_LOGIC_VECTOR;
8 END my_package;
9 ------------------------------------------
10 PACKAGE BODY my_package IS
11     FUNCTION "+" (a, b: STD_LOGIC_VECTOR)
12             RETURN STD_LOGIC_VECTOR IS
13     VARIABLE result: STD_LOGIC_VECTOR;
14     VARIABLE carry: STD_LOGIC;
15     BEGIN
16             carry := '0';
17             FOR i IN a'REVERSE_RANGE LOOP
18                     result(i) := a(i) XOR b(i) XOR carry;
19                     carry := (a(i) AND b(i)) OR (a(i) AND carry) OR
20                                     (b(i) AND carry);
21             END LOOP;
22 RETURN result;
23 END "+";
24 END my_package;
25 ------------------------------------------
```

```
1 --------- Main code: -------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 USE work.my_package.all;
5 ------------------------------------------
6 ENTITY add_bit IS
7       PORT ( a: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
8                 y: OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
9 END add_bit;
10 ------------------------------------------
```

```
11 ARCHITECTURE my_arch OF add_bit IS
12       CONSTANT b: STD_LOGIC_VECTOR(3 DOWNTO 0) := "0011";
13       CONSTANT c: STD_LOGIC_VECTOR(3 DOWNTO 0) := "0110";
14 BEGIN
15        y <= a + b + c; -- overloaded "+" operator
16 END my_arch;
17 ---------------------------------------------
```

PROCEDURE

A PROCEDURE is very similar to a FUNCTION and has the same basic purposes. However, a procedure can return more than one value.

Like a FUNCTION, two parts are necessary to construct and use a PROCEDURE: the procedure itself (procedure body) and a procedure call.

Procedure Body

```
PROCEDURE procedure_name [<parameter list>] IS
        [declarations]
BEGIN
        (sequential statements)
END procedure_name;
```

A PROCEDURE can have any number of IN, OUT, or INOUT parameters, which can be SIGNALS, VARIABLES, or CONSTANTS. For input signals (mode IN), the default is CONSTANT, whereas for output signals (mode OUT or INOUT) the default is VARIABLE.

WAIT, SIGNAL declarations, and COMPONENTS are not synthesizable when used in a FUNCTION. The same is true for a PROCEDURE, with the exception that a SIGNAL can be declared, but then the PROCEDURE must be declared in a PROCESS. Moreover, besides WAIT, any other edge detection is also not synthesizable with a PROCEDURE (that is, contrary to a function, a synthesizable procedure should not infer registers).

Example: The PROCEDURE below has three inputs, a, b, and c (mode IN).

a is a CONSTANT of type BIT, while b and c are SIGNALS, also of type BIT. Notice that the word CONSTANT can be omitted for input parameters, for it is the default object (recall, however, that for outputs the default object is VARIABLE). There are also two return signals, x (mode OUT, type BIT_VECTOR) and y (mode INOUT, type INTEGER).

```
PROCEDURE my_procedure ( a: IN BIT; SIGNAL b, c: IN BIT;
SIGNAL x: OUT BIT_VECTOR(7 DOWNTO 0);
SIGNAL y: INOUT INTEGER RANGE 0 TO 99) IS
BEGIN
...
END my_procedure;
```