Lecture 3: Subprograms and Packages

The objective of this supplemental material is to introduce you to the concepts of subprograms (functions and procedures) and packages in VHDL.

1. VHDL Functions

A ***function*** executes a sequential algorithm and returns a single value to the calling program. We can think of a function as a generalization of expressions. The syntax rule for a function declaration is:

```
[pure | impure] function identifier [(parameter_interface_list)]
      return type_mark is
      {subprogram declarations}
begin
      {sequential statements}
end [function] [identifier];
```

By default (i.e., if no keyword is given), functions are declared as ***pure***. A pure function does not have access to a shared variable, because shared variables are declared in the declarative part of the architecture and pure functions do not have access to objects outside of their scope. Only parameters of mode 'in' are allowed in function calls and are treated as 'constant' by default.

Functions may be used wherever an expression is necessary within a VHDL statement. Subprograms themselves, however, are executed sequentially like processes. Similar to a process, it is also possible to declare local variables. These variables are initialized with each function call with the leftmost element of the type declaration (boolean: false, bit: '0'). The leftmost value of integers is guaranteed to be at least -(2^31)-1 (i.e. zeros must be initialized to 0 at the beginning of the function body). It's recommended to initialize all variables in order to enhance the clarity of the code.

Example 1: The following VHDL code describes a simple function that adds two 4-bit vectors and a carry in and returns a 5-bit sum:

```
function add4_func(a, b : std_logic_vector(3 downto 0); carry: std_logic)
      return std_logic_vector is
variable cout : std_logic;
variable cin : std_logic;
variable sum : std_logic_vector(4 downto 0);
begin
cin := carry;
sum := "00000";
loop1 : for i in 0 to 3 loop
      cout := (a(i) and b(i)) or (a(i) and cin) or (b(i) and cin);
      sum(i) := a(i) xor b(i) xor cin;
      cin := cout;
end loop loop1;
sum(4) := cout;
return sum;
end add4_func;
```

Question: what is the role of the statement: `cin := cout;` inside loop1?

2. VHDL Procedures

Procedures, in contrast to functions, are used like any other statement in VHDL. Consequently, they do not have a return value, although the keyword 'return' may be used to indicate the termination of the subprogram. Depending on their position within the VHDL code, either in an architecture or in a process, the procedure as a whole is ***executed concurrently or sequentially***, respectively.

Procedures facilitate decomposition of VHDL code into modules. They can return any number of values using output parameters. The default mode of a parameter is 'in', the keyword 'out' or 'inout' is necessary to declare output signals/variables. The syntax rule for a procedure declaration is:

```
procedure identifier [(parameter_interface_list)] is
      {subprogram declarations}
begin
      {sequential statements}
end [procedure] [identifier];
```

Example 2: The following procedure does basically the same thing as the function in the previous example:

```
procedure add4_proc
      (a, b : in std_logic_vector(3 downto 0);
      carry : in std_logic;
      signal sum : out std_logic_vector(3 downto 0);
      signal cout : out std_logic) is
variable c : std_logic;
begin
c := carry;
for i in 0 to 3 loop
      sum(i) <= a(i) xor b(i) xor c;
      c := (a(i) and b(i)) or (a(i) and c) or (b(i) and c);
end loop;
cout <= c;
end add4_proc;
```

3. Packages and libraries

Packages and libraries provide a convenient way of referencing frequently used functions and components. Packages are the only language mechanism to share objects among different design units. Usually, they are designed to provide standard solutions for specific problems (e.g., data types and corresponding subprograms like type conversion functions for a certain bus protocol, procedures and components (macros) for signal processing purposes, etc.).

A ***package*** consists of a package declaration and an optional package body. The package declaration contains a set of declarations, which may be shared by several design units (for example: types, signals, components, and function and procedure declarations). The body package usually contains the functions and procedure bodies.
The syntax rule for a package declaration is:

```
package identifier is
      {package declarations}
begin
      {sequential_statement}
end [package] [identifier];
```

A package is analyzed separately and placed in the working library by the analyzer. Each package declaration that includes function and/or procedure declarations must have a corresponding package body. The syntax rule for a package body is:

```
package body identifier is
      {package body declarations}
end [package body] [identifier];
```

Example 3: Simple package declaration and its corresponding body.

```
library IEEE;
use IEEE.std_logic_1164.all;
package my_package is
function add4_func(a, b: std_logic_vector(3 downto 0); carry : std_logic)
      return std_logic_vector;
procedure add4_proc
      (a, b: in std_logic_vector(3 downto 0);
      carry: in std_logic;
      signal sum: out std_logic_vector(3 downto 0);
      signal cout: out std_logic);
end package my_package;
```

Since the package contains subprogram declarations, we declare also the package body:

```
package body my_package is

function add4_func(a, b: std_logic_vector(3 downto 0); carry: std_logic)
      return std_logic_vector is
variable cout: std_logic;
variable cin: std_logic;
variable sum: std_logic_vector(4 downto 0);
begin
cin := carry;
sum := "00000";
loop1: for i in 0 to 3 loop
      cout := (a(i) and b(i)) or (a(i) and cin) or (b(i) and cin);
      sum(i) := a(i) xor b(i) xor cin;
      cin := cout;
end loop loop1;
sum(4) := cout;
return sum;
end add4_func;

procedure add4_proc
      (a, b: in std_logic_vector(3 downto 0);
      carry: in std_logic;
      signal sum: out std_logic_vector(3 downto 0);
      signal cout: out std_logic) is
variable c: std_logic;
```

```
begin
c := carry;
for i in 0 to 3 loop
      sum(i) <= a(i) xor b(i) xor c;
      c := (a(i) and b(i)) or (a(i) and c) or (b(i) and c);
end loop;
cout <= c;
end add4_proc;

end package body my_package;
```

Suppose the above package and package body declarations are saved as **my_package.vhd**, (i.e., as a VHDL file). Normally, it could be analyzed and placed in any directory, for instance MY_LIBRARY directory. Then, we can write other VHDL files (or library units) in which we instantiate items from the newly created library, (i.e., MY_LIBRARY), using the "selected name". The "selected name" is formed by writing the library name, then the package name, and then the name of the item (or all if you want to use all items), all separated by dots. For example:

```
library MY_LIBRARY;
use MY_LIBRARY.my_package.all;
```

Example 4: Simple 8-bit adder using the above package.

Use ISE WebPack to create a new project. Add to your project the following two VHDL files, and then synthesize and implement the design.

```
-------------------------------------------------------------------
-- First VHDL file: has package declaration and package body.
-- Save it as my_package.vhd
library IEEE;
use IEEE.std_logic_1164.all;
-------------------------------------------------------------------
package my_package is
function add4_func(a, b : std_logic_vector(3 downto 0); carry : std_logic)
      return std_logic_vector;
procedure add4_proc
      (a, b : in std_logic_vector(3 downto 0);
      carry: in std_logic;
      signal sum: out std_logic_vector(3 downto 0);
      signal cout: out std_logic);
end package my_package;
-------------------------------------------------------------------
package body my_package is

function add4_func(a, b : std_logic_vector(3 downto 0); carry: std_logic)
      return std_logic_vector is
variable cout: std_logic;
variable cin: std_logic;
variable sum: std_logic_vector(4 downto 0);
begin
cin := carry;
sum := "00000";
loop1: for i in 0 to 3 loop
      cout := (a(i) and b(i)) or (a(i) and cin) or (b(i) and cin);
      sum(i) := a(i) xor b(i) xor cin;
      cin := cout;
end loop loop1;
sum(4) := cout;
return sum;
end add4_func;

procedure add4_proc
      (a, b : in std_logic_vector(3 downto 0);
      carry: in std_logic;
      signal sum: out std_logic_vector(3 downto 0);
      signal cout: out std_logic) is
variable c: std_logic;
begin
c := carry;
for i in 0 to 3 loop
      sum(i) <= a(i) xor b(i) xor c;
      c := (a(i) and b(i)) or (a(i) and c) or (b(i) and c);
end loop;
cout <= c;
end add4_proc;

end package body my_package;
--------------------------------------------------------------------

--------------------------------------------------------------------
-- Second VHDL file: simple 8-bit adder
-- Uses items from "my_package" created in WORK library directory
-- in your current project directory
-- Save it as bit8_adder.vhd
library IEEE;
use IEEE.std_logic_1164.all;
use WORK.my_package.all;

entity bit8_adder is
port(a, b: in std_logic_vector(7 downto 0);
      ci: in std_logic;
      y: out std_logic_vector(7 downto 0);
      co: out std_logic);
end bit8_adder;

architecture structural of bit8_adder is
signal internal_carry : std_logic;
signal sum1, sum2: std_logic_vector(4 downto 0);
begin
sum1 <= add4_func(a(3 downto 0), b(3 downto 0), ci);
sum2 <= add4_func(a(7 downto 4), b(7 downto 4), sum1(4));
y <= sum2(3 downto 0) & sum1(3 downto 0);
co <= sum2(4);
end;
--------------------------------------------------------------------
```