Lucrare de laborator nr. 11
Funcţii, proceduri şi package-uri

1. Scopul lucrării

Însuşirea noţiunilor privind definirea şi utilizarea funcţiilor şi procedurilor în VHDL; declaraţia de package şi corpul package-ului.

2. Funcţii şi proceduri

În VHDL funcţiile şi procedurile fac parte din categoria subprogramelor. O procedură poate returna una sau mai multe valori, în schimb o funcţie returnează întotdeauna o singură valoare. Pe de altă parte, în timp ce toţi parametrii unei funcţii sunt parametri de intrare, în cazul unei proceduri parametri pot fi de intrare, de ieşire sau de intrare-ieşire (inout).

În funcţie de locul în care sunt apelate, funcţiile şi procedurile pot fi: *concurente* – dacă sunt apelate în afara unei instrucţiuni *process*, sau *secvenţiale* – dacă sunt apelate în cadrul unei instrucţiuni *process*.

Toate instrucţiunile din definiţia funcţiilor şi procedurilor sunt instrucţiuni secvenţiale. Aceste instrucţiuni sunt aceleaşi cu cele din cadrul unui proces, cu excepţia instrucţiunii *wait* care nu poate fi folosită în cazul funcţiilor.

În ceea ce priveşte apelarea subprogramelor, funcţiile sunt apelate în instrucţiuni de atribuire sau în expresii, în timp ce procedurile sunt apelate ca instrucţiuni distincte de apel de procedură fie într-o arhitectură sau într-un proces.

Funcţiile şi procedurile pot fi definite în trei locuri:

- Într-un *package*. În acest caz, în declaraţia de package se află linia de antet a subprogramelor, în timp ce în corpul package-ului se află linia de antet împreună cu corpul subprogramelor respective.
- În *partea declarativă* a unei *arhitecturi*.
- În *partea declarativă* a unui *proces*.

Uzual, subprogramele se definesc în cadrul package-urilor deoarece, în acest caz, pot fi utilizate la diverse proiecte.

Sintaxa de definire a unei funcţii:

function <nume_functie> ([clasa_obiect] <nume_param>{, nume_param}:
  [in] tip [(interval_index)], …)] **return** tip **is**
        [declaraţii de constante]
        [declaraţii de variabile]
        [declaraţii de tip]
begin
        instrucţiuni secvenţiale
        return(expresie);
end [<nume_functie>];

Argumentele (parametrii) unei funcţii sunt din clasa constant sau signal iar modul este numai **in** (intrare). Dacă pentru un parametru al funcţiei nu este specificată clasa sau modul, implicit se consideră că este din clasa constant, respectiv modul in.

Sintaxa pentru definirea unei proceduri:

procedure <nume_procedura>([clasa_obiect] <nume_param>{, nume_param}:
  [mod] tip [(interval_index)], …)] **is**
        [declaraţii de constante]
        [declaraţii de variabile]

[declarații de tip]
begin
instrucțiuni secvențiale
end [<nume_procedura>];

O procedură poate modifica valorile argumentelor sale. Argumentele fac parte din clasa constant, variable sau signal şi pot avea trei moduri diferite: **in**, **out** sau **inout**. Dacă modul nu este specificat atunci implicit acesta este considerat in. Dacă modul este out sau inout atunci argumentul trebuie să aparțină clasei variable sau signal. În instrucțiunile concurente de apel de procedură argumentele nu pot fi din clasa variabilelor.

3. Package-uri

Principalul scop al unui package (pachet) este de a stoca elemente care pot fi utilizate în diverse proiecte.

Un package constă din două părți: *declarația de package* şi *corpul package-ului*. Declarația de package defineşte interfața pentru package, aşa cum declarația de entitate definește interfața pentru model.

În declarația de package pot fi întâlnite următoarele declarații:

- declarații de subprograme
- declarații de tipuri şi subtipuri
- declarații de semnale globale
- declarații de fişier
- declarații de componente
- declarații şi specificații de atribute, etc.

Sintaxa pentru declarația de package:
package <nume_package> **is**
declarații
end [nume_package];

Dacă în declarația de package sunt declarate funcții sau proceduri atunci package-ul este insoțit de un corp al package-ului (*package body*). În corpul package-ului se află declarațiile subprogramelor împreună cu corpul acestora.

Toate elementele declarate într-un package pot fi făcute vizibile si utilizate în orice proiect folosind clauza **use**. Astfel, dacă un package este compilat într-o biblioteca numita pack_lib, atunci acesta poate fi făcut vizibil într-un proiect dacă în fişierul acestuia înainte de declarația de entitate se adaugă următoarele două linii:

```
library pack_lib;
use pack_lib.nume_package.all;
entity .....
```

3.1 Package-uri standardizate

Limbajul VHDL este însoțit implicit de două package-uri compilate în biblioteca *std*: package-ul `standard`, care conține declarații pentru tipurile predefinite ale limbajului şi package-ul *textio*.

Package-ul IEEE std_logic_1164

Pentru a reflecta mai bine proprietățile electrice ale circuitelor digitale, IEEE a dezvoltat şi standardizat un nou tip de date pentru a extinde tipurile predefinite bit şi bit_vector. Astfel, în package-ul std_logic_1164 cele mai utile tipuri de date declarate sunt

std_ulogic şi std_ulogic_vector cu 9 stări. Deoarece aceste tipuri de date sunt nerezolvate, în cadrul package-ului este definită o funcție de rezoluție numită *resolved* cu ajutorul căreia s-au declarat subtipurile std_logic şi std_logic_vector. Acestea sunt subtipuri ale tipului std_ulogic însoțit de funcția de rezoluție. Totodată, în cadul package-ului sunt definite funcții pentru operatorii care să lucreze cu tipul std_logic. Package-ul std_logic_1164, la fel ca toate celelalate package-uri IEEE este compilat în biblioteca *ieee*.

Mai jos este prezentat o parte din codul VHDL corespunzător package-ului std_logic_1164:

```
PACKAGE std_logic_1164 IS
    ------------------------------------------------------------------
    -- logic state system  (unresolved)
    ------------------------------------------------------------------
    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
    ------------------------------------------------------------------
    FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic;

    ------------------------------------------------------------------
    -- *** industry standard logic type ***
    ------------------------------------------------------------------
    SUBTYPE std_logic IS resolved std_ulogic;

    ------------------------------------------------------------------
    -- unconstrained array of std_logic for use in declaring signal arrays
    ------------------------------------------------------------------
    TYPE std_logic_vector IS ARRAY ( NATURAL RANGE <>) OF std_logic;

    ------------------------------------------------------------------
    -- common subtypes
    ------------------------------------------------------------------
    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
    ------------------------------------------------------------------

    FUNCTION "and"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "nand" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "or"   ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "nor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    FUNCTION "xor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
    function "xnor" ( l : std_ulogic; r : std_ulogic ) return ux01;
    FUNCTION "not"  ( l : std_ulogic                 ) RETURN UX01;
.................
end std_logic_1164;
```

Funcţiile “and”, „nand”, etc din package-ul std_logic_1164 supradefinesc operatorii logici and, nand, etc. În VHDL se pot utiliza acelaşi nume de funcţii sau operatori pentru date de tipuri diferite. Această posibilitate este cunoscută ca supradefinirea funcţiilor sau operatorilor.

Package-ul IEEE numeric_std

În plus faţă de operatorii logici, la implementarea sistemelor digitale sunt necesare şi operaţii aritmetice. În package-ul std_logic_1164 operatorii aritmetici sunt definiţi numai pentru tipul predefinit integer.

Pentru a putea efectua operaţii aritmetice cu semnale pe mai mulţi biţi, IEEE a standardizat package-ul *numeric_std* în care sunt declarate două noi tipuri de date: *signed* (cu semn) şi *unsigned* (fără semn]. Ambele tipuri de date sunt definite ca un tablou (array) de elemente de tip std_logic. Pentru tipul de date *unsigned*, tabloul este interpretat ca un număr binar fără semn în care bitul din stânga este interpretat ca MSB. Pentru tipul de date *signed*, tabloul este interpretat ca un număr binar cu semn în format complement faţă de 2.

Deoarece scopul package-ului numeric_std este de a suporta operaţii aritmetice, în cadrul acestuia s-au supradefinit operatorii aritmetici abs, *, /, mod, rem, *, + şi -. De asemenea, s-au definit cîteva funcţii pentru operaţiile de rotire şi deplasare precum şi de conversie.

Mai jos este prezentat o parte din codul package-ului IEE numeric.std:

```
      library IEEE;
      use IEEE.STD_LOGIC_1164.all;
      package NUMERIC_STD is
        constant CopyRightNotice: STRING
            := "Copyright 1995 IEEE. All rights reserved.";
        attribute builtin_subprogram: string;
        --
============================================================================
        -- Numeric array type definitions
============================================================================

        type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
        type SIGNED is array (NATURAL range <>) of STD_LOGIC;
        --
============================================================================
        -- Arithmetic Operators:
        --
============================================================================

        -- Id: A.1
        function "abs" (ARG: SIGNED) return SIGNED;
          attribute builtin_subprogram of
              "ABS"[SIGNED return SIGNED]: function is "numstd_abs_ss";
        -- Result subtype: SIGNED(ARG'LENGTH-1 downto 0).
        -- Result: Returns the absolute value of a SIGNED vector ARG.

      ..................
end NUMERIC_STD.
```

Întrucât tipurile *signed* şi *unsigned* sunt tablouri, declaraţiile lor sunt similare cu cele ale tipului std_logic_vector. De ex:

```
            signal x, y, z, w: signed(7 downto o);
```

Pe de altă parte sunt corecte următoarele instrucţiuni de atribuire:

```
            z <= x+y;
            w <= x+1;
```

4. Exemplu de package cu funcţii şi proceduri

În continuare este prezentat un package care conține declarații pentru o procedură şi două funcții de conversie. Ca urmare package-ul este însoțit de corpul său.

(fisier Pack_Utils.vhd)

```
3   package Utils is
5     procedure Clock ( signal C: out Bit; HT, LT :Time );
7     function Convert ( N, L : Natural ) return Bit_Vector ;
8     function Convert ( B : Bit_Vector ) return Natural;
10  end Utils;

14  package body Utils is
16    procedure Clock ( signal C : out Bit; HT, LT : Time ) is
17    begin
18          loop
19                C <= '1' after LT, '0' after LT+HT;
20                wait for LT+HT;
21          end loop;
22    end;
23
24    function Convert ( N, L : Natural ) return Bit_Vector is
25      variable Temp : Bit_Vector ( L-1 downto 0);
26      variable Value : Natural := N;
27    begin
28      for i in Temp'Right to Temp'Left loop
29          Temp (i) := Bit'Val ( Value mod 2);
30          Value := Value /2 ;
31       end loop;
32    return Temp;
33  end;

35  function Convert ( B: Bit_Vector ) return Natural is
36    variable Temp : Bit_Vector ( B'Length - 1 downto 0) :=B;
37    variable Value: Natural := 0 ;
38  begin
39    for i in Temp'Right to Temp'Left loop
40          if Temp (i) = '1' then
41                Value := Value + ( 2 ** i) ;
42          end if;
43    end loop;
44    return Value;
45  end;
47  end Utils;
```

Declaratia de pachet *("package*") descrie acele parti care vor fi accesibile in exteriorul pachetului si cuprinde declaratia unei proceduri si a doua functii. In limbajul VHDL procedurile si functiile sunt denumite subprograme. In consecinta, declaratia unui subprogram reprezinta declaratia unei functii sau proceduri.

O funcție întoarce întotdeauna doar un singur rezultat. Intr-o funcție toți parametrii sunt parametri de mod intrare, în timp ce la proceduri pot fi parametri de mod intrare, parametri de mod ieşire, sau parametri de mod intrare-ieşire.

Procedurile şi funcțiile concurente există în afara specificațiilor proceselor sau ale altor subprograme. Procedurile şi funcțiile secvențiale există numai în specificațiile proceselor sau ale altor subprograme. Toate specificațiile din interiorul unui subprogram sînt secvențiale.

Declaratia de la linia 5 declara o procedura "Clock" care poate fi apelată de un numar de ori pentru a genera semnalul de ceas cu perioada si factor de umplere variabile. Procedura

are trei parametri: “C” care este un semnal de tip bit comandat de procedura ca semnal de clock, ultimii doi parametri “HT” si “LT” reprezentind durata de timp cit semnalul de clock este in ‘1’, respectiv ‘0’ logic. Procedura este apelata concurent.

Declaratiile de la liniile 7 si 8 declara doua functii ce au acelasi nume “*Convert*” si realizeaza conversia intre tipurile standard Bit_Vector si Integer. Limbajul VHDL permite ca subprogramele sa fie supradefinite in anumite conditii.

Functia “Convert” de la linia 7 accepta doi parametri, ambii de tip Integer si returneaza o valoare de tip Bit_Vector. Aceasta functie converteste intregul nenegativ N intr-un vector de tip Bit_Vector de lungime L, ai carui biti reprezinta numarul N in formatul fara semn. Functia “Convert” de la linia 8 realizeaza conversia in sens invers.

Supraincarcarea functiilor este posibila atita timp cit compilatorul VHDL face distinctie macar intre una din caracteristicile urmatoare:tip returnat, numar de parametri si tipul parametrilor functiilor ce se supraincarca.

Deoarece declaratia de pachet contine declaratii ale unor subprograme este necesar si un corp al pachetului care sa contina corpul subprogramelor. Mai poate cuprinde si alte declaratii utilizabile in interiorul corpului pachetului.

Fiecare subprogram declarat in declaratia de pachet trebuie sa aiba un corp corespunzator in corpul pachetului.

Liniile 16-21 introduc un asemenea corp pentru procedura “Clock”. Prima linie a corpului trebuie sa fie identica cu declaratia de procedura din declaratia de pachet. Acest lucru este necesar pentru compilatorul VHDL pentru a identifica corect declaratia de subprogram cu corpul corespunzator.

Specificatiile din corpului procedurii (liniile 18-21) realizeaza o bucla infinita. Reamintim ca aceasta procedura este apelata concurent. Asemenea proceduri sint apelate o singura data, specificatiile din corpul lor fiind executate in mod repetat; daca ar trebui sa revina la apelator ele nu vor mai fi apelate.

Specificatiile din bucla implementeaza semnalul de tact. In linia 19 semnalul de tact este comandat in ‘1’ logic dupa intervalul LT apoi in ‘0’ dupa intervalul LT+HT. Specificatia *wa*it de la linia urmatoare are rolul de permite stabilirea modificarilor pe semnalul de tact. Sfirsitul buclei produce reexecutarea specificatiilor din corpul subprogramului.

Liniile 24-33 si 35-45 furnizeaza corpul celor doua functii “Convert”.

In corpul primei functii este declarat un vector de biti de lungime corespunzatoare (L) si facuta o copie a lui N (se impune decrementarea lui N, dar limbajul VHDL nu permite acest lucru). Bucla de la liniile 28-31 parcurge elementele variabilei Temp de la dreapta la stinga utilizind atributele “Right” si “Left”.

Pentru fiecare bit se verifica daca valoarea curenta de convertit este para sau impara. Daca este para, bitul curent ar trebui sa fie ‘0’, altfel este ’1’.

Se genereaza valorile de ‘0’ si ‘1’ utilizind un atribut predefinit al tipurilor:”Val”. Acesta se comporta in felul urmator: expresia “***Value mod 2***” este zero daca variabila *Valu*e este para, altfel expresia este unu.

*“**Bit'Val(0)*** “ returneaza valoarea bitului al carui numar de pozitie este 0. **“BitVal(1)”** returneaza valoara bitului al carui numar de pozitie este 1.

Numarul de pozitie al unui tip scalar reprezinta distanta fara de cel mai din stinga element . Tipurile enumerat si numeric sint tipuri scalare.

Odata bucla terminata se iese din ea si se reia executia de la linia 32, ceea ce produce intoarcerea valorii memorate in variabila *“Tem*p” ca rezultat al apelului functiei. O functie poate avea mai multe specificatii *return* dar toate trebuie sa intoarca rezultate de acelasi tip .

Functia “Convert”, al carei corp incepe la linia 35 lucreaza putin diferit. Mai intii este creata o copie a lui *B* in variabila *Temp* (linia 36) determinind astfel directia si ultima valoare a indexului.

Reamintim ca un vector de biti poate avea, ca si valorile extreme ale indexul sau, orice intregi nenegativi si ca domeniul vectorului poate fi crescator sau descrescator. Este creata de asemenea o variabila acumulator la linia 37 a carei valoare initiala este zero.

Pentru fiecare bit al variabilei *Temp* se testeaza daca este '1', caz in care se adauga 2^i in acumulator, unde *i* reprezinta distanta dintre bitul curent si bitul cel mai din dreapta. Cind bucla se incheie variabila Value contine valoarea dorita care este intoarsa la linia 44.

Operatorul de ridicare la putere ****** admite un operand sting intreg sau virgula mobila dar trebuie sa aiba un operand intreg pozitiv in dreapta, daca operandul din stinga este intreg, sau intreg daca operandul din stinga este in virgula flotanta.

5 Testarea funcțiilor şi procedurii:

(fisier Test_Utils.vhd)

```
52  entity Test_Utils is end;
56  use work.Utils.all ;
57  architecture Driver of Test_Utils is
59    signal N: Natural;
60    signal B: Bit_Vector ( 5 downto 0);
61    signal C: Bit;
62    use work.Utils.all;
64  begin
66    CLK : Clock (C, 10 ns, 20 ns);
68    process
69          use work.Utils.all;
70    begin
71          for i in 0 to 31 loop
72          B <= Convert ( i, B'Length ) after 10 ns;
73          wait for 10 ns;
74          end loop;
75          wait;
76    end process;
78    N <= Convert (B) after 1 ns;
80 end;
```

Deoarece pachetul nu contine o pereche entitate-arhitectura nu vom declara si nici nu vom instantia a componenta pentru a crea o unitate supusa testarii, ci vom utiliza pachetul cu o clauza *use*. Vom apela apoi functiile si procedurile in programul de test.

Clauza *use* apare de trei ori (liniile 56,62,69). Oricare doua dintre acestea sint redundante, ele arata locul unde pot fi introduse .

Expresia "**use work.Utils.all"** este interpretata in modul urmator: in biblioteca WORK este cautat un pachet deja compilat denumit *"Utils*". Sint importate declaratiile din interfata pachetului (dar nu si acelea din corpul sau). Atentie! Dupa numele bibliotecii de lucru **work** inserati numele pachetului –cel de dupa cuvintul rezervat *package*- nu al fisierului in care declarati pachetul.

Procedura "*Clock"* si functiile "*Convert"* din testbench vor referi la subprogramele declarate in pachetul Utils. Clauza *use* este asemanatoare directivei *"#include < >"* din ***C***.

Linia 66 testeaza procedura *"Clock"* cu un apel concurent de procedura. Reamintim ca specificatiile intr-o arhitectua sint executate concurent. Aceasta procedura este executata concurent cu un proces (liniile 68-76) si cu o specificatie de atribuire de semnal la linia 78.

Corpul procedurii *"Clock"* este o bucla infinita. Nu poate fi compilata pina cind unitatea supusa testului nu se stabilizeaza, iar semnalul de clock, prin definitie, nu este stabil niciodata! Este necesara o comanda pentru o pauza, dupa o perioada timp expirata.

Functia *"Convert"* ce intoarce un rezultat de tip *Bit_Vector* este testata la linia 72. La fiecare zece nanosecunde este convertit un intreg in domeniul 0=<i=<31. Rezultatul este memorat in semnalul *B*. Cind acesta isi schimba valoarea, specificatia de atribuire de la linia 78 apeleaza functia de conversie *Convert* al carei rezultat este atribuit semnalului N dupa o nanosecunda .

Functia *Convert* apelata la linia 78 are ca parametru un tip *Bit_Vector* iar ca tip returnat are un tip intreg. Compilatorul VHDL apeleaza functia *Convert* corecta dupa regulile corespunzatoare supradefinirii functiilor. Daca nu exista nici o functie cu acest nume este emis un mesaj eroare.

Existenta mai multor functii cu acelasi nume care au acelasi tip returnat si aceeasi parametri genereaza eroare.

6. Aplicatii

1. Vizualizați codul sursă al package-urilor IEEE std_logic_1164 şi numeric_std. Urmăriți funcțiile definite în aceste package-uri.
2. Vizualizați codul sursă al package-urilor std_logic_unsigned şi std_logic_signed. Care sunt diferențele față de package-ul numeric_std?
3. Lecturați cu atenție explicațiile referitoare la implementarea funcțiilor Convert şi a procedurii Clock, precum şi a entității de test.
4. Copiați fişierele sursă Pack_Utils.vhd şi Test_Utils.vhd în directorul de lucru, compilați, efectuați simularea şi vizualizați rezultatele.
5. În fişierul Test_Utils, în arhitectura Driver completați codul VHDL astfel încât după simulare să puteți vizualiza valoarea indexului i utilizat în instrucțiunea for…loop.
6. În fişierul Pack_Utils.vhd, creați alte două funcții de conversie denumite Convert_slogic similare cu funcțiile Convert în care tipul bit este înlocuit cu tipul std_logic.
7. Completați arhitectura Driver astfel încat să testați şi funcțiile Convert_slogic.