Subprograms

Lecture 6

Subprograms

- A subprogram defines a sequential algorithm that performs some computations.
- Subprograms can be:
  - 1. functions
  - 2. procedures
- Functions:
  - Compute a single value
  - Execute in zero simulation time => **do not contain WAIT statements**.
- Procedures:
  - May return zero or more values
  - May contain WAIT statements
    - In that case do not execute in zero simulation time
    - NOT RECOMMENDED !!
  - May be called sequentially or concurrently.

Subprograms: specification

Subprogram-specification:

```
subprogram-specification IS
        subprogram-item-declarations
BEGIN
        subprogram-statements
END [FUNCTION | PROCEDURE] [subprogram-name];
```

Subprogram-specification:

- gives the name of the subprogram
- defines the interface of the subprogram, i.e. the formal parameters.

Subprogram-item-declarations: the declarations from the subprogram

Subprogram-statements: sequential statements and the RETURN statement

Subprogram-name at the end: if it appears, is the same as the name from subprogram-specification

Symbol ; at the end is mandatory.

Specification: parameters

- Formal parameters have:
  - name
  - Type
  - class: constant, variable, signal, FILE
  - mode: IN, OUT, INOUT
- The mode of the formal parameters:
  - IN – may be read, but not modified
  - OUT –may be modified, but may not be read
  - INOUT: may be both read and modified
  - Files don’t have mode, they may be read or written, depending on how they are opened.
  - Functions may have only parameters of mode IN, procedures may have parameters of any mode.

Subprogram parameters

- Subprogram calls use actual parameters.
- Associations:
  - To a formal parameter of class signal it may be associated only an actual parameter of class signal.
  - To a formal parameter of class variable it may be associated only an actual parameter of class variable.
  - To a formal parameter of class file it may be associated only an actual parameter of class file.
  - To a formal parameter of class constant it may be associated an expression.
  - The type of the formal and actual parameter must be the same
  - If a formal parameter is an unconstrained array, its size will be given by the size of the actual parameter
- Signals:
  - In functions we MAY NOT assign values to signals
  - If inside a procedure there is a signal assignment statement, then the signal driver is affected immediately, no matter if the procedure ends or not.
  - In procedures (and in functions) it is not allowed to use attributes that generate new signals: ‘STABLE, ‘QUIET, ‘TRANSACTION, ‘DELAYED

Subprograms: declarations

- Subprogram-item-declarations:
  - May be type declarations or object declarations (constants, variables, but NOT signals!)
  - May not be component declarations !
  - These declarations are visible only inside subprograms and become effective only at the subprogram call.
  - Hence, a variable declared inside a subprogram
    - is initialized at every call of the subprogram and
    - It exists only during the execution of the subprogram.
    - (A variable declared inside a process is initialized at the beginning of the simulation, exists for the entire duration of the simulation and it maintains its value from one execution of the process to another)

Specification: statements

- The statements from subprograms can be:
  - Sequential statements (like in processes)
  - RETURN statement:
    - Syntax: RETURN [expression];
    - It may exist only in subprograms
    - When it is executed the subprogram ends and the control is given back to the program that called the subprogram.
    - All functions must contain RETURN expression;
      - Value of the expression is returned to the program that called the function
    - In procedures the OUT and INOUT parameters return values, too.
- Procedures may have side effects, i.e. they can affect objects that are declared globally to the procedure -> **NOT recommended !!**

Procedures

Procedures permit the de-composing of long code sequences into sections. Unlike a function, a procedure may return zero or more values.

Syntax for procedure specification:

PROCEDURE *procedure_name*(*parameter_list*)

where parameter_list specifies the list of formal parameters.

Parameter classes: constant, variable, signals (and files)

Parameter mode: IN, OUT, INOUT.

If the parameter class is not specified, then it will be constant for the parameters of mode IN and variable for the parameters of mode OUT or INOUT.

It it is not specified the mode of a parameter, then implicitly it will be IN.

Example of procedure:

```
TYPE op_code IS (add, sub, mul, div, lt, le, eq);
```

…

Procedures

```
PROCEDURE alu_proc IS (a,b: IN INTEGER; op: op_code; z:
OUT INTEGER; zcomp: OUT BOOLEAN) IS

BEGIN

      CASE op IS

            WHEN add => z:=a+b;

            WHEN sub=> z:=a-b;

            WHEN mul=> z:=a*b;

            WHEN div=> z:=a/b;

            WHEN lt=> zcomp := a<b;

            WHEN le=> zcomp:= a<=b;

            WHEN eq=> zcomp:=a=b;

END CASE;

END PROCEDURE alu_proc;
```

Procedures

Procedure call can be sequential or concurrent. Syntax:

[label:] procedure_name(list_of_actuals);

The concurrent procedure call is equivalent with a process that contains the sequential call and that has a sensitivity list containing the parameters belonging to class signal and having the mode IN or INOUT.

Actual parameters can be specified by positional or named association.

Example:

alu_proc(d1,d2,add, sum, comp); -- positional association

alu_proc(z=>sum, b=>d2, a=>d1, op=>add, zcomp=>comp);-- named association

If a procedure contains WAIT statements, then it may not be called from a process that has a sensitivity list.

A procedure may be postponed, being declared in this way:

POSTPONED PROCEDURE a_procedure(list_of_parameters) IS

…

Declaration versus specification

A procedure may be only declared, without specifying its body. The same is true for functions:

- for example in PACKAGE DECLARATION

-it is useful at a recursive call (see example)

Syntax for a subprogram declaration:

subprogram_specification

Example with two subprograms:

```
PROCEDURE p(...) IS

      VARIABLE a, b:...

BEGIN

      a:=q(b);

...

END p;
```

Declaration versus specification

```
FUNCTION q(…)
BEGIN
…
      p();
…
END q;
```

The example is wrong, it will not compile.

The correct form is:

```
PROCEDURE p();
FUNCTION q();
PROCEDURE p() IS
…
END p;
FUNCTION q() IS
…
END FUNCTION q;
```

Functions

Specification:

```
[PURE|IMPURE] FUNCTION function_name (parameter_list)
RETURN return_type
```

A function is *pure* if it returns the same value every time when it is called with the same set of actual parameters.

An *impure* function may return different values when it is called with the same set of actual parameters.

Implicitly a function is pure. Example of impure functions:

- functions that generate random numbers
- The function NOW (returns the current simulation time).

Parameter_list describes the list of formal parameters of the function. They can be only of mode IN and may belong to constant or signal classes, being implicitly constants.

Function call can be done only in expressions, according to the following syntax:

```
function_name(list_of_actuals)
```

Functions

For a constant, the actual parameter may be a constant or an expression, for a signal, the actual may be only a signal.

Association between formal and actual parameters can be positional or by name.

Functions MAY NOT contain WAIT statements and may not assign values to signals.

In general functions do not have side effects.

Example of a function that detects the rising edge of a signal:

```
FUNCTION rising_edge (SIGNAL s: BIT) RETURN BOOLEAN IS

BEGIN

      RETURN (s=‘1’ AND s’EVENT AND s’LAST_VALUE=‘0’);

END FUNCTION rising_edge;
```

Most frequent utilization of functions:

- for conversions
- resolution functions

Conversion functions

In VHDL there are few cases when we can use cast-type conversion, like (INTEGER) x or (REAL) y. In general conversion functions are needed.

Example:

```
TYPE mvl IS (‘X’, ‘0’, ‘1’, ‘Z’);

TYPE fourval IS (X, L, H, Z);

FUNCTION mvl_to_fourval(v: mvl) RETURN fourval IS

BEGIN

      CASE v IS

            WHEN ‘X’ => RETURN X;

            WHEN ‘0’ => RETURN L;

            WHEN ‘1’ => RETURN H;

            WHEN ‘Z’ => RETURN Z;

      END CASE;

END FUNCTION;
```

Conversion with look-up table

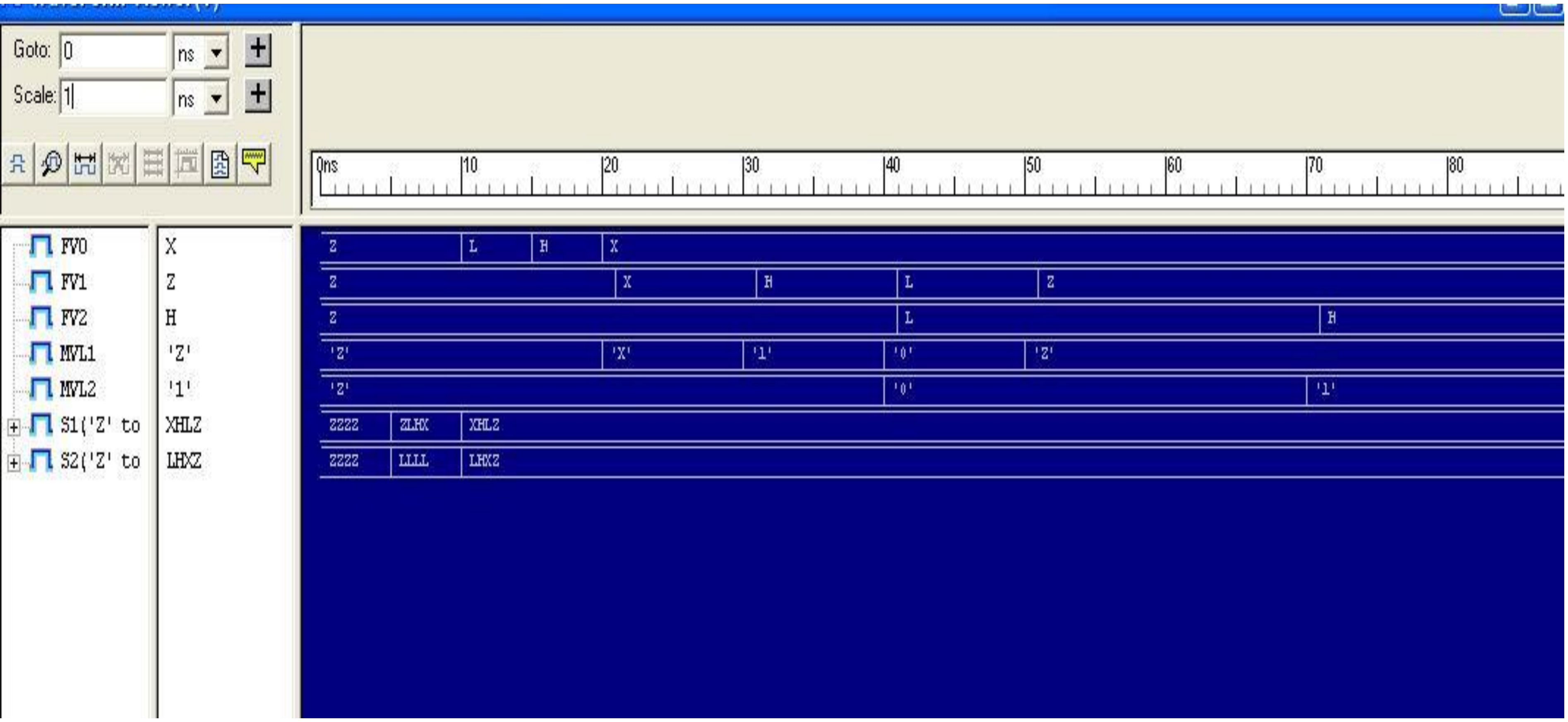
The conversion is more efficient if look-up tables are used instead of functions (the function calls introduce an overhead), like in the following example:

```
entity look_up_table_2 is
        generic (tp: time:=5ns);
end;
architecture a of look_up_table_2 is
        type mvl is ('Z', '0', '1', 'X');
        type fourval is (Z, L, H, X);
        type look_up is array(mvl) of fourval;

        CONSTANT to_fourval: look_up:=('Z'=>Z, '0'=>L,'1'=>H,'X'=>X);

        SIGNAL fv0,fv1,fv2: fourval;
        SIGNAL mvl1,mvl2: mvl;

        SIGNAL s1, s2: look_up;
BEGIN
        Process(mvl1)
        Begin
                fv1 <= to_fourval(mvl1) after 1ns;
        end process;
```

Conversion with look-up table

```
        process(mvl2)
        begin
                fv2<= to_fourval(mvl2) after 1ns;
        end process;

fv0 <= to_fourval(mvl'('Z')) after tp, to_fourval('0') after 2*tp,
to_fourval('1') after 3*tp, to_fourval('X') after 4*tp;

mvl1 <= 'X' after 20ns, '1' after 30ns, '0' after 40ns, 'Z' after 50ns;
mvl2 <= '0' after 40ns, '1' after 70ns;

s1 <= ('X'=>X, '1'=>H, '0'=>L, 'Z'=> Z) after tp, ('X'=>Z, 'Z'=>X,
'0'=>H, others=>L) after 2*tp;

s2 <= (others=>L) after tp, (L,H,X,Z) after 2*tp;

end architecture;
```

Signal diagram


Goto: 0 ns

Scale: 1 ns

```
FV0          X
FV1          Z
FV2          H
MVL1         'Z'
MVL2         '1'
S1('Z' to    XHLZ
S2('Z' to    LHXZ
```

Fig 10. Signal diagram for the previous example of conversion with look-up table.

Resolution functions

- Are used in order to resolve the value of a signal when the signal has more than one source (driver).
- In VHDL is illegal if a signal with more than one driver is not resolved.
- The resolution function:
  - Is written by the programmer
  - Is called by the simulator:
    - When at least one of the signal’s driver has an event.
    - The resolution function will return a value obtained from the values of all signal’s drivers.
- Unlike other HDLs, in VHDL there aren’t predefined resolution functions.

Overloading

- Sometimes we want to have two subprograms with the same name, case when we say that the subprograms are overloaded
  - For example a subprogram named *count,* which counts bits, and another subprogram, named also *count*, which counts integers
- A call to such subprograms is ambiguous, hence **wrong**, if we cannot identify the called subprogram.
- Subprogram identification can be based on:
  - Subprograms names
  - The number of actual parameters
  - Type and order of the actual parameters
  - The name of formal parameters (if the call uses the association by name)
  - The type of result (at functions)

Operator overloading

- When a standard VHDL operator is made to behave differently we say that we have an operator overloading
  - Usually we want to extend the operator for operands of different types than in the case of the standard operator
  - When the new operator is declared, its name is put between citation marks, in order to be able to call it as an operator
    - The operator may be called as a function, too.

Example of operator overloading

```
--Operator definition
FUNCTION “+” (op1, op2: BIT_VECTOR) RETURN BIT_VECTOR;
FUNCTION “-” (op1, op2: BIT_VECTOR) RETURN BIT_VECTOR;
TYPE mvl IS (X, L, H, Z);
FUNCTION “OR”(l, r: mvl) RETURN mvl;
FUNCTION “AND”(l, r: mvl) RETURN mvl;
…
--Operator call
VARIABLE x,y,z1,z2: BIT_VECTOR(3 DOWNTO 0);
VARIABLE a1, b1, c1,c2: mvl;
z1:=x+y;-- standard notation for operators
z2:=“-”(x,y);-- standard notation for functions
c1:=a1 AND b1;-- standard notation for operators
c2:= “OR”(a1,b1);--standard notation for functions
```