Chapter 6.1~4 Additional Topics in VHDL

Version: 2023/12/19

Chapter 6 Additional Topics in VHDL

6.1 VHDL Functions (函数)

	Functions	Procedures
Return value	A single value	Any number of values
Return method	Use **return** statement	Use **output** parameter

VHDL provides **functions** and **procedures(过程)** to easily express repeated invocation of the same functionality or the repeated use of structures

Functions, procedures: **subprograms(子程序)**

6.1 VHDL Functions (函数)

A simple example

```
function rotate_right (reg: bit_vector)
  return bit_vector is
begin
  return reg ror 1;
end rotate_right;
```

The function returns a bit-vector equal to input bit-vector (reg) rotated one position to the right

```
B <= rotate_right(A);
```

If A = "10010101", it would set B = " 11001010 ", and leave A unchanged

Function call can be used anywhere that an expression can be used

6.1 VHDL Functions (函数)

A simple example

```
function rotate_right (reg: bit_vector)
  return bit_vector is
begin
  return reg ror 1;
end rotate_right;
```

- A function is a section of sequential code
- Its purpose is to create new functions to deal with commonly encountered problems (data type conversions, logical operations, arithmetic computations, new operators, attributes)

- By writing such code as a function, it can be shared and reused, also propitiating the main code to be shorter and easier to understand

6.1 VHDL Functions (函数)

A simple example

```
function rotate_right (reg: bit_vector)
  return bit_vector is
begin
  return reg ror 1;
end rotate_right;
```

- The same statements that can be used in a process (**if**, **case**, **loop**) can also be used in a function, with the exception of **wait**
- Other two prohibitions in a function are **signal declarations** and **component instantiations**

To construct and use a function, two parts are necessary:

- the function itself (function-body)
- a call to the function

6.1 VHDL Functions (函数)

General form of function declaration

```
function function-name (formal-parameter-list)
  return return-type is
[declarations]
begin
  sequential statements
end function-name;
```

Variables can be declared here

Signal **cannot** be declared in a function

6.1 VHDL Functions (函数)

General form of function declaration

```
function function-name (formal-parameter-list)
  return return-type is
[declarations]
begin
  sequential statements
end function-name;
```

return type

must include **return return-value**

Note: Function body may **not** contain a **wait** statement or a signal assignment

General form of function call

```
   function-name(actual-parameter-list)
```

acutal-parameter-list must match the formal-parameter-list

6.1 VHDL Functions (函数)

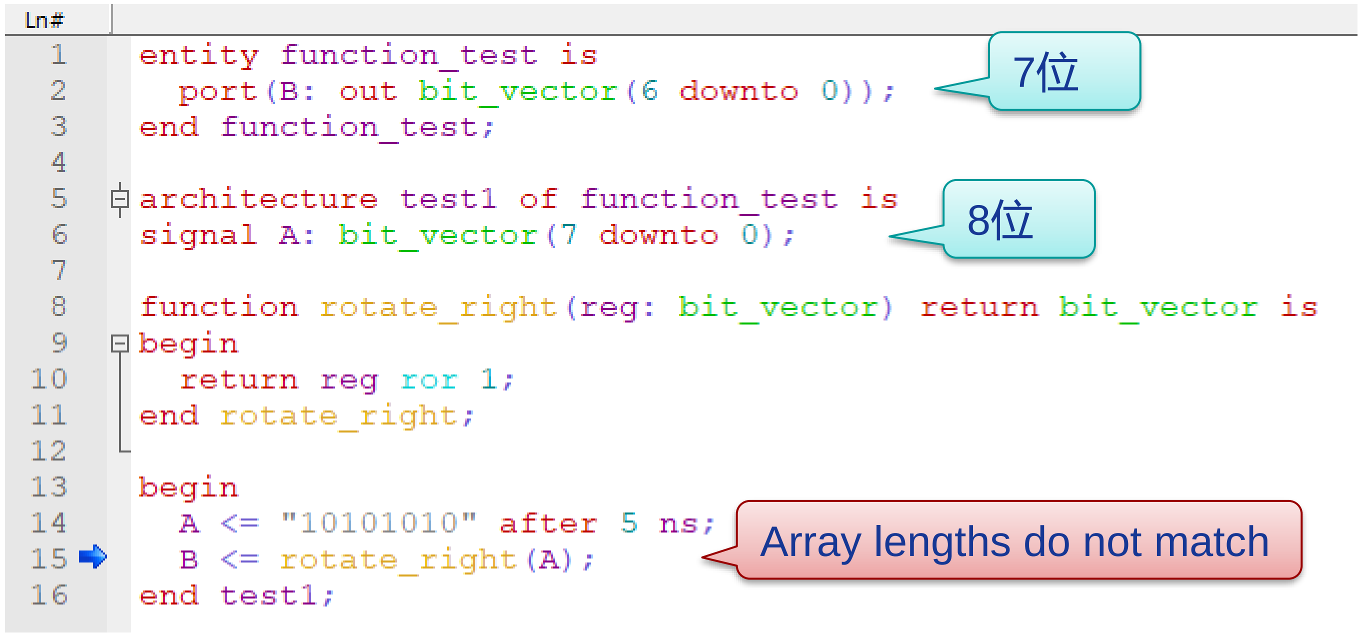
Actual parameters are treated as input values and cannot be changed during the execution of the function

Example

```
entity function_test is
  port(B: out bit_vector(7 downto 0));
end function_test;

architecture test1 of function_test is
signal A: bit_vector(7 downto 0);

function rotate_right(reg: bit_vector) return bit_vector is
begin
  reg <= reg ror 1;
  return reg;
end rotate_right;

begin
  A <= "10101010" after 5 ns;
  B <= rotate_right(A);
end test1;
```

Error: Function body may **not** a signal assignment

6.1 VHDL Functions (函数)

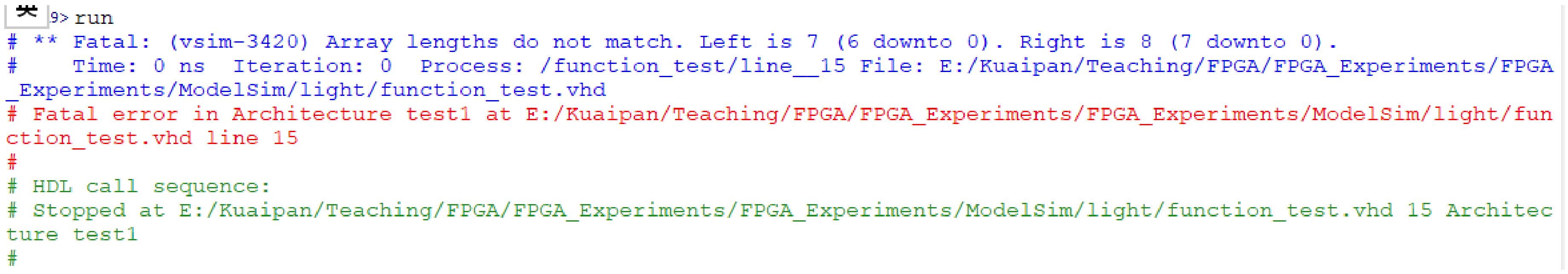


```vhdl
entity function_test is
  port(B: out bit_vector(6 downto 0));
end function_test;

architecture test1 of function_test is
signal A: bit_vector(7 downto 0);

function rotate_right(reg: bit_vector) return bit_vector is
begin
  return reg ror 1;
end rotate_right;

begin
  A <= "10101010" after 5 ns;
  B <= rotate_right(A);
end test1;
```

7位

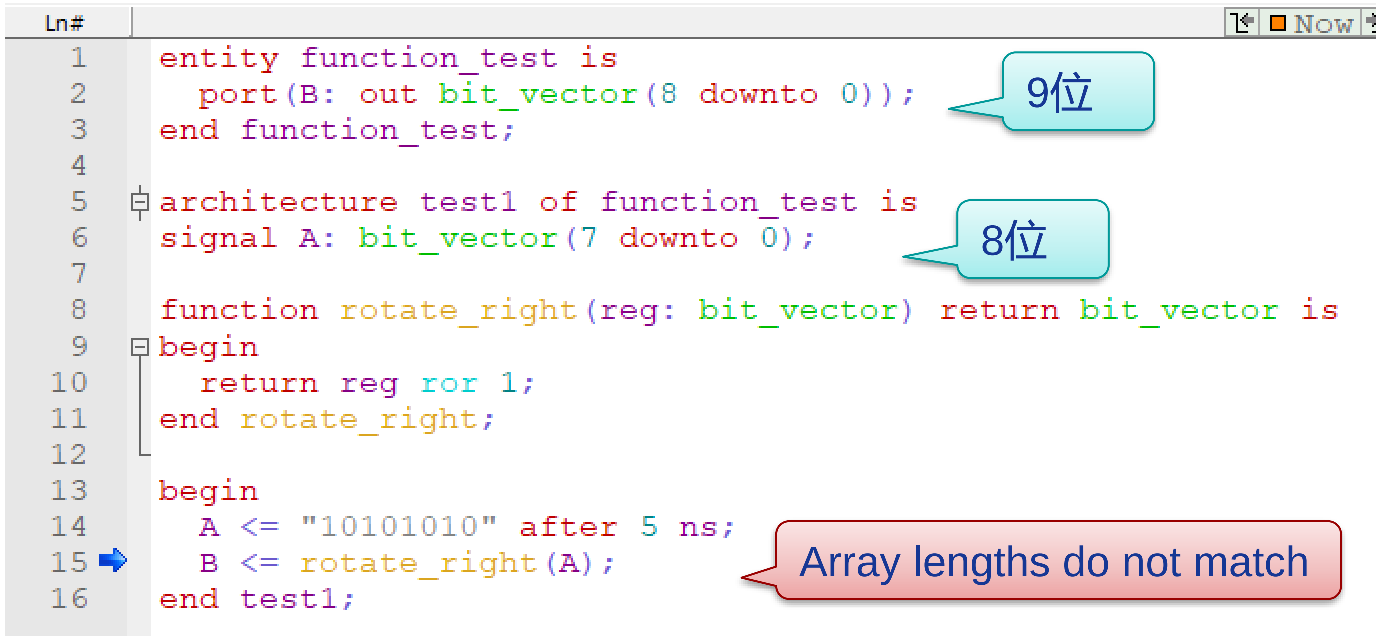
8位

Array lengths do not match

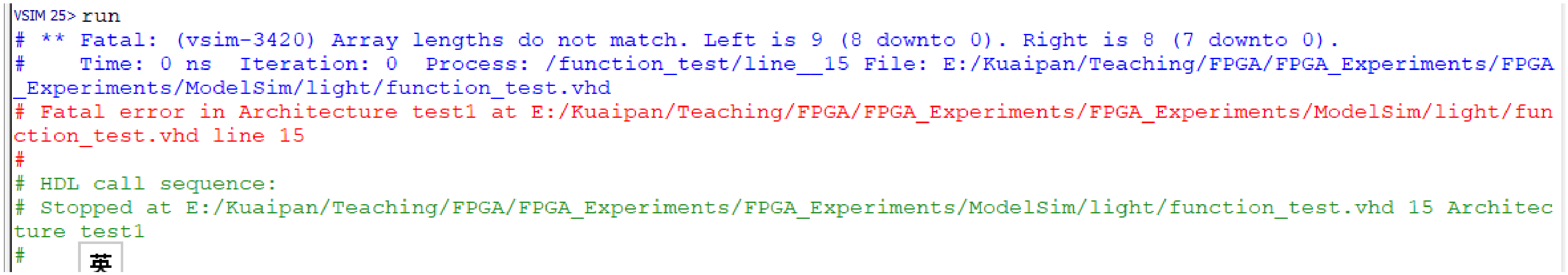


```
9> run
# ** Fatal: (vsim-3420) Array lengths do not match. Left is 7 (6 downto 0). Right is 8 (7 downto 0).
#    Time: 0 ns  Iteration: 0  Process: /function_test/line__15 File: E:/Kuaipan/Teaching/FPGA/FPGA_Experiments/FPGA
_Experiments/ModelSim/light/function_test.vhd
# Fatal error in Architecture test1 at E:/Kuaipan/Teaching/FPGA/FPGA_Experiments/FPGA_Experiments/ModelSim/light/fun
ction_test.vhd line 15
#
# HDL call sequence:
# Stopped at E:/Kuaipan/Teaching/FPGA/FPGA_Experiments/FPGA_Experiments/ModelSim/light/function_test.vhd 15 Architec
ture test1
#
```

6.1 VHDL Functions (函数)



```vhdl
entity function_test is
  port(B: out bit_vector(8 downto 0));
end function_test;

architecture test1 of function_test is
signal A: bit_vector(7 downto 0);

function rotate_right(reg: bit_vector) return bit_vector is
begin
  return reg ror 1;
end rotate_right;

begin
  A <= "10101010" after 5 ns;
  B <= rotate_right(A);
end test1;
```



```
VSIM 25> run
# ** Fatal: (vsim-3420) Array lengths do not match. Left is 9 (8 downto 0). Right is 8 (7 downto 0).
#    Time: 0 ns  Iteration: 0  Process: /function_test/line__15 File: E:/Kuaipan/Teaching/FPGA/FPGA_Experiments/FPGA
_Experiments/ModelSim/light/function_test.vhd
# Fatal error in Architecture test1 at E:/Kuaipan/Teaching/FPGA/FPGA_Experiments/FPGA_Experiments/ModelSim/light/fun
ction_test.vhd line 15
#
# HDL call sequence:
# Stopped at E:/Kuaipan/Teaching/FPGA/FPGA_Experiments/FPGA_Experiments/ModelSim/light/function_test.vhd 15 Architec
ture test1
#
```

6.1 Function

Example: Even Parity generation for a 4-bit number

```
  function parity (A:  bit_vector(3 downto 0))
     return bit_vector is
   variable parity:  bit;
   variable B:  bit_vector(4 downto 0);
   begin
     parity := a(0) xor a(1) xor a(2) xor a(3);
     B := A & parity;
     return B;
   end parity;
```

If parity circuits are used in several parts in a system, we could call the function each time it is desired

6.1 Function

Example: Add function

```
C <= A + B
```

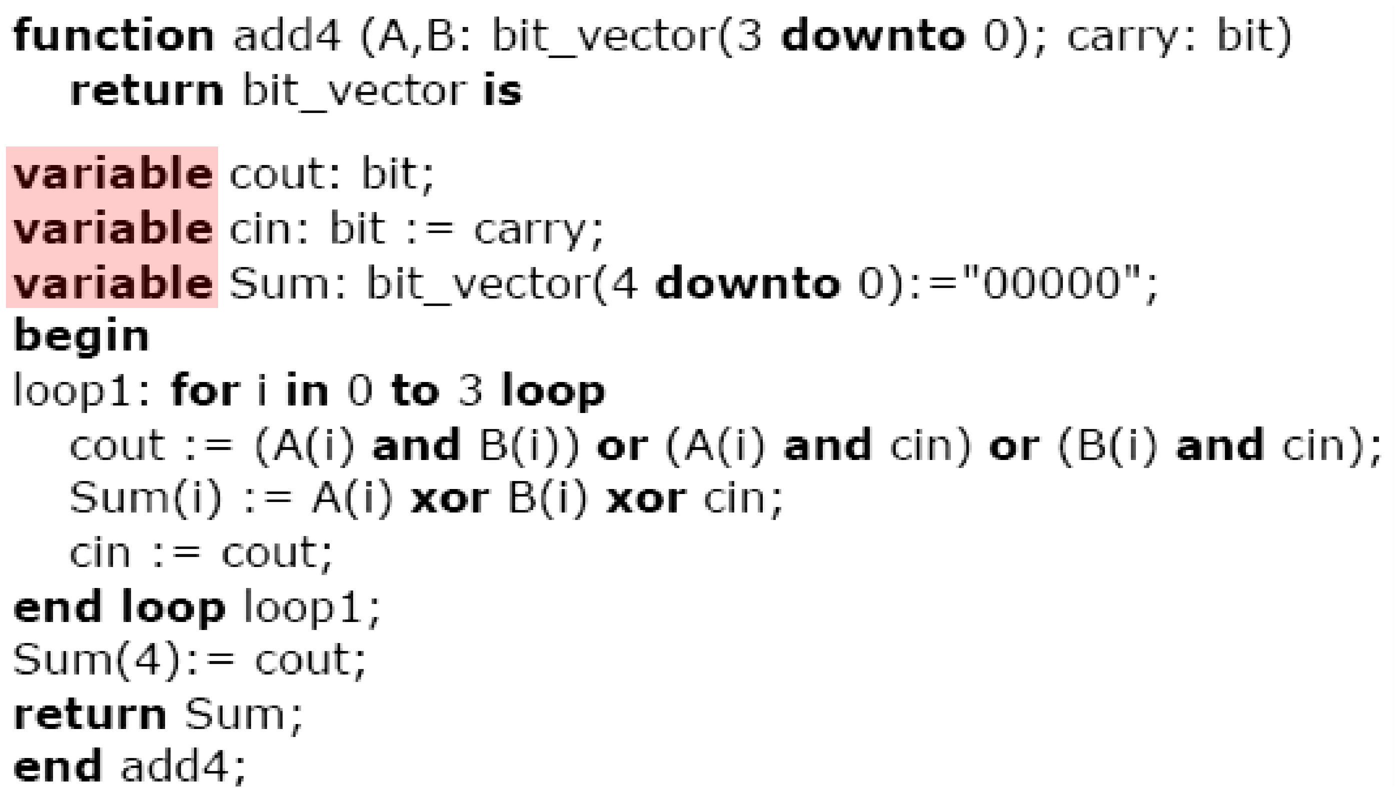
If A, B, and C are integers, the statement C <= A + B will set C equal to the sum of A and B

or if numeric_bit package is used, and A, B, and C are unsigned

- If A, B, and C are bit-vectors, this statement will not work
- “+” operation is not defined for bit-vectors

6.1 Function

Example: Add function



```
function add4 (A,B: bit_vector(3 downto 0); carry: bit)
    return bit_vector is

variable cout: bit;
variable cin: bit := carry;
variable Sum: bit_vector(4 downto 0):="00000";
begin
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;
```

6.1 Function

Example: Add function

```
function add4 (A,B: bit_vector(3 downto 0); carry: bit)
   return bit_vector is

variable cout: bit;
variable cin: bit := carry;
variable Sum: bit_vector(4 downto 0):="00000";
begin
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;
```

; is used

When add4() is called, cin will be initialized to the value of carry

- Variables **cout** and **cin** are defined to hold intermediate values during the calculation
- Variable **sum** is used to store the value to be returned

6.1 Function

Example: Add function

```
function add4 (A,B: bit_vector(3 downto 0); carry: bit)
   return bit_vector is

variable cout: bit;
variable cin: bit := carry;
variable Sum: bit_vector(4 downto 0):="00000";
begin
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;
```

What is the total simulation time required to execute the add4()?

- Not even Δ time is required
- All the computation are done using variables, and variables are updated instantaneously

6.1 Function

Example: Add function call

```
add4(A, B, carry);
```

- A and B may be replaced with any expression that evaluate to bit-vector with dimensions 3 downto 0
- carry may be replaced with any expression that evaluates to a bit

```
Z <= add4(X, not Y, '1');
```

- Parameters A, B, and carry are set equal to the values of X, not Y and '1', respectively
- X and Y must be bit-vectors dimensioned 3 downto 0

- The function computes

$$\text{Sum} = A + B + \text{carry} = X + \text{not } Y + '1'$$

and return this value
- Since sum is a variable, computation of sum requires zero time
- After delta time, Z is set equal to the returned value of Sum

Example: Squares function

```
library IEEE;
use IEEE.numeric_bit.all;
entity test_squares is
  port(CLK:  in bit);
end test_squares;
architecture test of test_squares is
  type FourBitNumbers is array (0 to 6) of unsigned (3 downto 0);
  type squareNumbers is array (0 to 6) of unsigned (7 downto 0);
  constant FN: FourBitNumbers := ("0001", "1000", "0011", "0010", "0101", "0000", "1111");
  signal answer:  squareNumbers;
  signal length:  integer :  = 6;
function squares (Number_arr:  FourBitNumbers; length:  positive)
  return squareNumbers is
variable SN:  squareNumbers;
begin
loop1:  for i in 0 to length loop
  SN(i) :  = Number_arr(i) * Number_arr(i);
end loop loop1;
return SN;
end squares;
begin
  process(CLK)
  begin
    if CLK = '1' and CLK'EVENT then
     answer <= squares(FN,  length);
    end if;
  end process;
end test;
```

Functions can be used to return an **array**

squares( ) inputs an array of numbers and returns an array

Example: Squares function

```
library IEEE;
use IEEE.numeric_bit.all;
entity test_squares is
  port(CLK:  in bit);
end test_squares;
architecture test of test_squares is
  type FourBitNumbers is array (0 to 6) of unsigned (3 downto 0);
  type squareNumbers is array (0 to 6) of unsigned (7 downto 0);
  constant FN: FourBitNumbers := ("0001", "1000", "0011", "0010", "0101", "0000", "1111");
  signal answer:  squareNumbers;
  signal length:  integer :  = 6;
function squares (Number_arr:  FourBitNumbers; length:  positive)
  return squareNumbers is
variable SN:  squareNumbers;
begin
loop1:  for i in 0 to length loop
  SN(i) :  = Number_arr(i) * Number_arr(i);
end loop loop1;
return SN;
end squares;
begin
  process(CLK)
  begin
    if CLK = '1' and CLK'EVENT then
     answer <= squares(FN,  length);
    end if;
  end process;
end test;
```

Length of input numbers is provided as a parameter

Functions are frequently used to do **type conversions**

In IEEE numeric_bit library:

```
to_integer(A)
```

convert an unsigned-vector to an integer

```
to_unsigned(B, N)
```

convert an integer to an unsigned-vector with N bits

Chapter 6 Additional Topics in VHDL

6.2 VHDL Procedures (过程)

Procedures can return any number of values

```
procedure procedure_name (formal-parameter-list) is
[declarations]
begin
  sequential statements
end procedure_name;
```

formal-parameter-list specifies **inputs** and **outputs** to procedure and their types

```
procedure_name(actual-parameter-list);
```

Procedure call can be a sequential or concurrent statement

2、VHDL Procedures(过程)

Example: Addvec

add two N-bit vectors and a carry, and return an N-bit sum and a carry

```
procedure Addvec
   (Add1,Add2: in bit_vector;
      Cin: in bit;
      signal Sum: out bit_vector;
      signal Cout: out bit;
      n:in positive) is
      variable C: bit;
begin
   C := Cin;
   for i in 0 to n-1 loop
      Sum(i) <= Add1(i) xor Add2(i) xor C;
      C := (Add1(i) and Add2(i)) or (Add1(i) and C) or (Add2(i) and C);
   end loop;
   Cout <= C;
end Addvec;
```

Procedure call: A, B, and Sum are N-bit vector

```
Addvec(A, B, Cin, Sum, Cout, N);
```

sequential or concurrent statement

2、VHDL Procedures(过程)

```
procedure Addvec
    (Add1,Add2: in bit_vector;
        Cin: in bit;
        signal Sum: out bit_vector;
        signal Cout: out bit;
        n:in positive) is
        variable C: bit;
begin
    C := Cin;
    for i in 0 to n-1 loop
        Sum(i) <= Add1(i) xor Add2(i) xor C;
        C := (Add1(i) and Add2(i)) or (Add1(i) and C) or (Add2(i) and C);
    end loop;
    Cout <= C;
end Addvec;
```

- C is a variable, since new value of C is needed each time through the loop

2、VHDL Procedures(过程)

```
procedure Addvec
   (Add1,Add2: in bit_vector;
      Cin: in bit;
      signal Sum: out bit_vector;
      signal Cout: out bit;
      n:in positive) is
      variable C: bit;
begin
   C := Cin;
   for i in 0 to n-1 loop
      Sum(i) <= Add1(i) xor Add2(i) xor C;
      C := (Add1(i) and Add2(i)) or (Add1(i) and C) or (Add2(i) and C);
   end loop;
   Cout <= C;
end Addvec;
```

- Sum can be a signal since Sum is not used within the loop

- After N times through the loop, all the values of the signal Sum have been computed
- But Sum is not updated until a delta time after exiting from the loop

2、VHDL Procedures(过程)

```
procedure Addvec
  (Add1,Add2: in bit_vector;
    Cin: in bit;
    signal Sum: out bit_vector;
    signal Cout: out bit;
    n:in positive) is
    variable C: bit;
begin
  C := Cin;
  for i in 0 to n-1 loop
    Sum(i) <= Add1(i) xor Add2(i) xor C;
    C := (Add1(i) and Add2(i)) or (Add1(i) and C) or (Add2(i) and C);
  end loop;
  Cout <= C;
end Addvec;
```

Within procedure declaration:

- class (**constant/signal/variable**)
- mode
- type

of each parameter are specified

2、VHDL Procedures(过程)

```
procedure Addvec
  (Add1,Add2: in bit_vector;
    Cin: in bit;
    signal Sum: out bit_vector;
    signal Cout: out bit;
    n:in positive) is
```

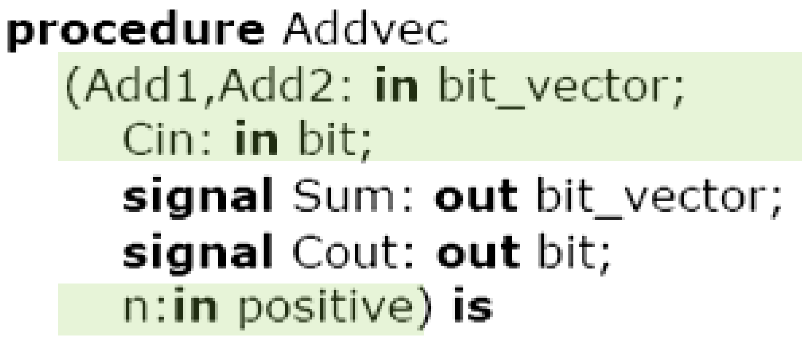
For **in** mode, **constant** is default class: Add1, Add2, Cin, n

For **out** and **inout** modes, **variable** is default class

=

```
procedure Addvec
     (constant Add1,Add2 : in bit_vector;
      constant Cin       : in bit;
      signal   Sum       : out bit_vector;
      signal   Cout      : out bit;
      constant n         : in positive)  is
```

2、VHDL Procedures(过程)



```
procedure Addvec
   (Add1,Add2: in bit_vector;
      Cin: in bit;
      signal Sum: out bit_vector;
      signal Cout: out bit;
      n:in positive) is
```

For a **constant** formal parameter, actual parameter can be any expression that evaluates to a constant of the proper type

Constant is used inside the procedure and **cannot** be changed

=

```
procedure Addvec
     (constant Add1,Add2  : in bit_vector;
      constant Cin        : in bit;
      signal   Sum        : out bit_vector;
      signal   Cout       : out bit;
      constant n          : in positive)  is
```

2、VHDL Procedures(过程)

```
procedure Addvec
  (Add1,Add2: in bit_vector;
    Cin: in bit;
    signal Sum: out bit_vector;
    signal Cout: out bit;
    n:in positive) is
```

Signals and variables can be mode **in**, **out**, or **inout**

- Parameters of modes **out** and **inout** can be changed in the procedure
- They are used to return values to the caller

‖

```
procedure Addvec
     (constant Add1,Add2 : in bit_vector;
      constant Cin       : in bit;
      signal   Sum       : out bit_vector;
      signal   Cout      : out bit;
      constant n         : in positive)  is
```

Parameters for Subprogram Calls

		Actual Parameter	
Mode	Class	Procedure Call	Function Call
In[1]	Constant[2]	Expression	Expression
	Signal	Signal	Signal
	Variable	Variable	n/a
Out/inout	Signal	Signal	n/a
	Variable[3]	Variable	n/a

1 Default mode for functions
2 Default for in mode
3 Default for out/inout mode
NOTE: n/a = "not applicable"

Function call: variable are not allowed

Parameters for Subprogram Calls

		Actual Parameter	
Mode	**Class**	**Procedure Call**	**Function Call**
In[1]	Constant[2]	Expression	Expression
	Signal	Signal	Signal
	Variable	Variable	n/a
Out/inout	Signal	Signal	n/a
	Variable[3]	Variable	n/a

1 Default mode for functions
2 Default for in mode
3 Default for out/inout mode
NOTE: n/a = "not applicable"

Function call: All parameters of function must be of mode **in** (default)

Parameters for Subprogram Calls

		Actual Parameter	
Mode	Class	Procedure Call	Function Call
In[1]	Constant[2]	Expression	Expression
	Signal	Signal	Signal
	Variable	Variable	n/a
Out/inout	Signal	Signal	n/a
	Variable[3]	Variable	n/a

1 Default mode for functions
2 Default for in mode
3 Default for out/inout mode
NOTE: n/a = "not applicable"

For function, **out** or **inout** are not allowed

Parameters for Subprogram Calls

		Actual Parameter	
Mode	**Class**	**Procedure Call**	**Function Call**
In[1]	Constant[2]	Expression	Expression
	Signal	Signal	Signal
	Variable	Variable	n/a
Out/inout	Signal	Signal	n/a
	Variable[3]	Variable	n/a

1 Default mode for functions
2 Default for in mode
3 Default for out/inout mode
NOTE: n/a = "not applicable"

A procedure can have output parameters (**signal/variable**, **out/inout**)

Process, Function, Procedure

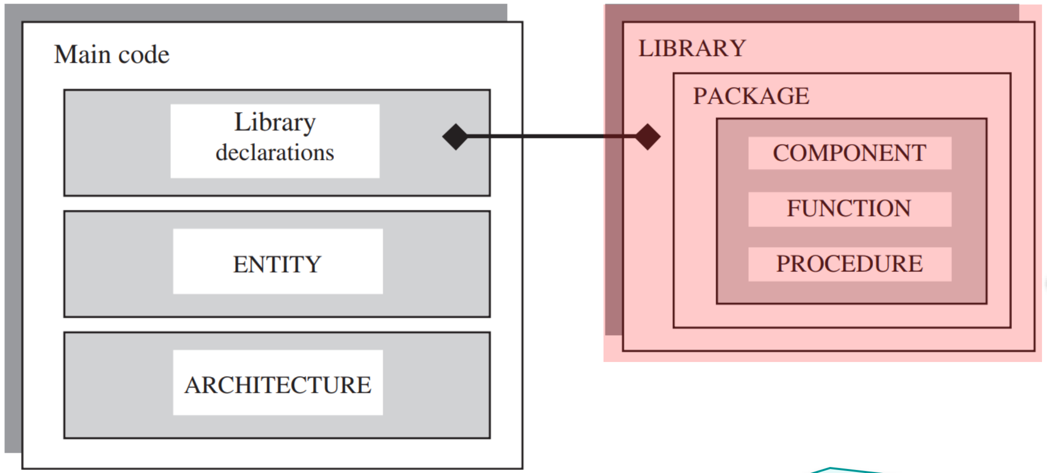
From a construction point of view, they are very similar:

- The only pieces of sequential code
- Employ same sequential statements (if, case, loop; wait not allowed in function)

From an application point of view, they are very different:

- Process: intended for immediate use in main code
- Function, procedure: mainly for library (can be used in main code)

Process, Function, Procedure



Main purpose of **component**, **function**, and **procedure** is to allow common pieces of code to be reused and shared

- They can be defined in the main code directly
- However, it is more usual to place them in a **library**

Frequently used pieces of code can be written in the form of component, function, or procedures, then placed in a package, which is finally compiled into the destination library

Structure of a package

Besides components, functions, and procedures, package can also contain type and constant definitions, among others

```
package package_name is
    (declarations)
end package_name;

[package body package_name is
    (function and procedure descriptions)
end package_name;]
```

Package part is mandatory and contains all declarations

package and package body must have the same name

Package body is necessary only when one or more subprograms are declared in the package part, in which case it must contain the descriptions of the subprograms

Example: Simple package

```
-------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
-------------------------------------------
package my_package is
       type state is (st1, st2, st3, st4);
       type color is (red, green, blue);
       constant vec: std_logic_vector(7 downto 0) := "11111111";
end my_package;
```

- Example my_package contains only type and constant declarations
- In this case, package body is not necessary

Example: Package with a Function

```
-------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
-------------------------------------------
package my_package is
       type state is (st1, st2, st3, st4);
       type color is (red, green, blue);
       constant vec: std_logic_vector(7 downto 0) := "11111111";
       function positive_edge(signal s: std_logic) return boolean;
end my_package;
-------------------------------------------
package body my_package is
       function positive_edge(signal s: std_logic) return boolean is
       begin
              return (s'event and s='1');
       end positive_edge;
end my_package;
-------------------------------------------
```

Example: Use my_package

```
-------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use work.my_package.all;
-------------------------------------------
entity ...
...
architecture ...
...
-------------------------------------------
```

Component

Component declaration

```
component component_name is
    port (
      port_name : signal_mode signal_type;
      port_name : signal_mode signal_type;
      ...)
end component;
```

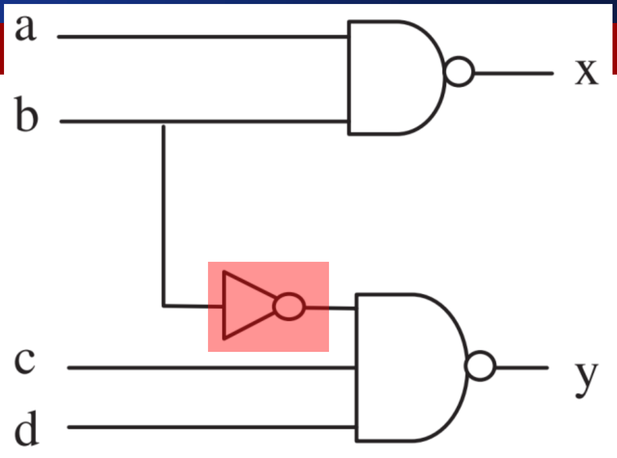
To use (instantiate) a component, it must first be declared

Component instantiation

```
label: component_name port map (port_list);
```

A label is required

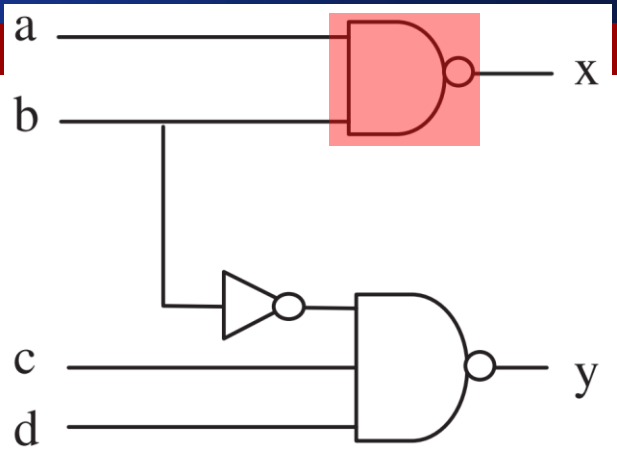
Component



Example 1: Components declared in the main code

```
-------------File inverter.vhd:-----------------
library ieee;
use ieee.std_logic_1164.all;
------------------------------------------------
entity inverter is
       port(a: in std_logic; b: out std_logic);
end inverter;
------------------------------------------------
architecture inverter of inverter is
begin
       b <= not a;
end inverter;
------------------------------------------------
```

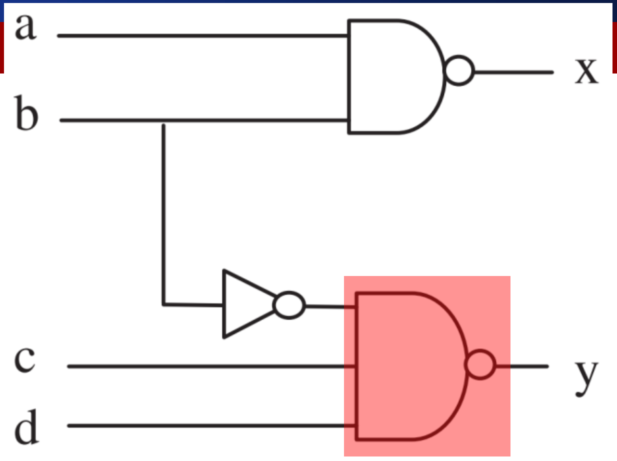
Component



Example 1: Components declared in the main code

```
---------------File nand_2.vhd:--------------------
library ieee;
use ieee.std_logic_1164.all;
---------------------------------------------------
entity nand_2 is
       port(a, b: in std_logic; c: out std_logic);
end nand_2;
---------------------------------------------------
architecture nand_2 of nand_2 is
begin
       c <= not (a and b);
end nand_2;
---------------------------------------------------
```

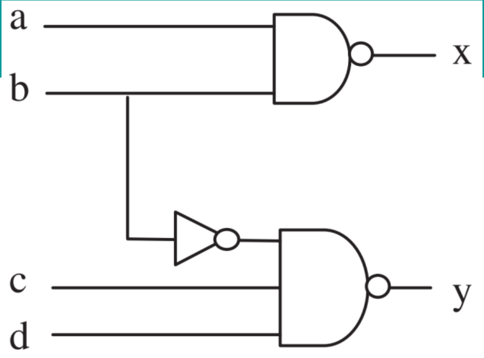
Component



Example 1: Components declared in the main code

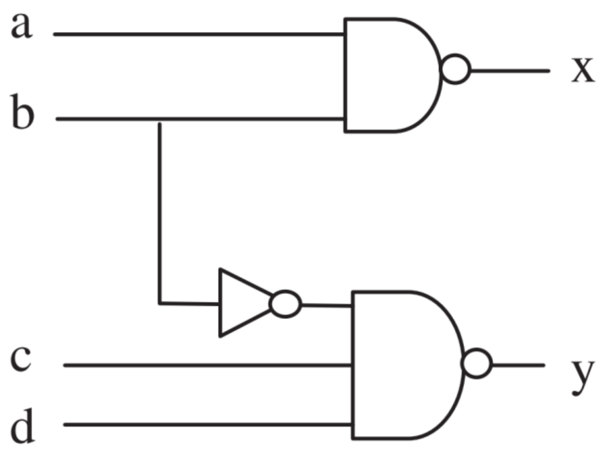
```
-----------------File nand_3.vhd:--------------------
library ieee;
use ieee.std_logic_1164.all;
------------------------------------------------------
entity nand_3 is
       port(a, b, c: in std_logic; d: out std_logic);
end nand_3;
------------------------------------------------------
architecture nand_3 of nand_3 is
begin
       d <= not (a and b and c);
end nand_3;
------------------------------------------------------
```

Example 1: Components declared in the main code



```
-----------------File project.vhd:---------------------
library ieee;
use ieee.std_logic_1164.all;
------------------------------------------------------
entity project is
        port(a, b, c, d: in std_logic; x, y: out std_logic);
end project;
------------------------------------------------------
architecture structural of project is
        --------------
        component inverter is
                port(a: in std_logic; b: out std_logic);
        end component;
        --------------
        component nand_2 is
                port(a, b: in std_logic; c: out std_logic);
        end component;
        --------------
        component nand_3 is
                port(a, b, c: in std_logic; d: out std_logic);
        end component;
        --------------
        signal w: std_logic;
begin
        U1: inverter port map (b, w);
        U2: nand_2 port map (a, b, x);
        U3: nand_3 port map (w, c, d, y);
end structural;
------------------------------------------------------
```

Component

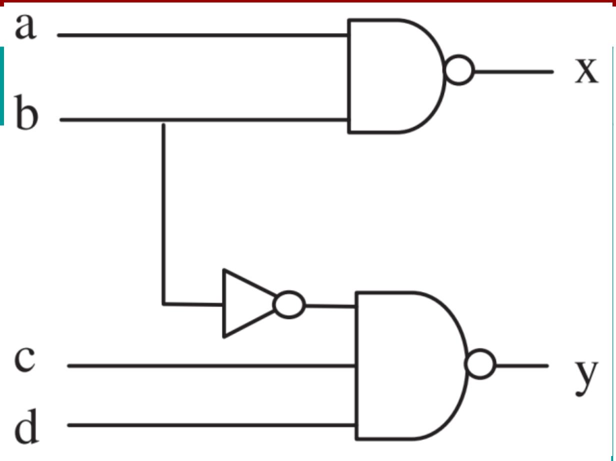


Example 2: Components declared in a package

```
--File inverter.vhd
--File nand_2.vhd
--File nand_3.vhd

--The above three file is needed.
--They are the same as those in the previous example.
```

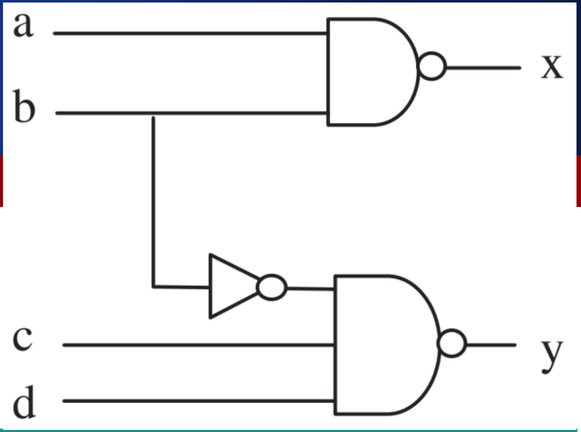
Component

Example 2: Components declared in a package



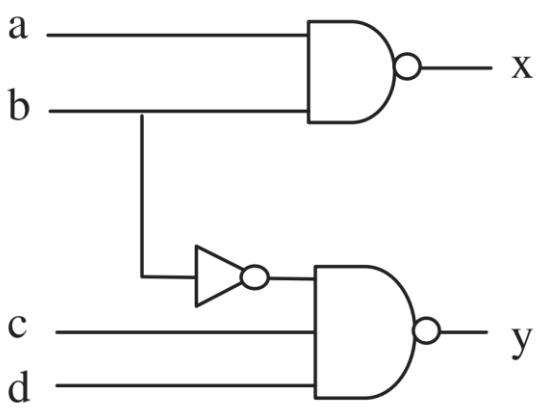
```
----- File my_components.vhd: -----
library ieee;
use ieee.std_logic_1164.all;
----------------------------
package my_components is
       ----- inverter: -----
       component inverter is
              port(a: in std_logic; b: out std_logic);
       end component;
       ----- 2-input nand: ---
       component nand_2 is
              port(a, b: in std_logic; c: out std_logic);
       end component;
       ----- 3-input nand: ---
       component nand_3 is
              port(a, b, c: in std_logic; d: out std_logic);
       end component;
       -----------------------
end my_components;
------------------------------------
```

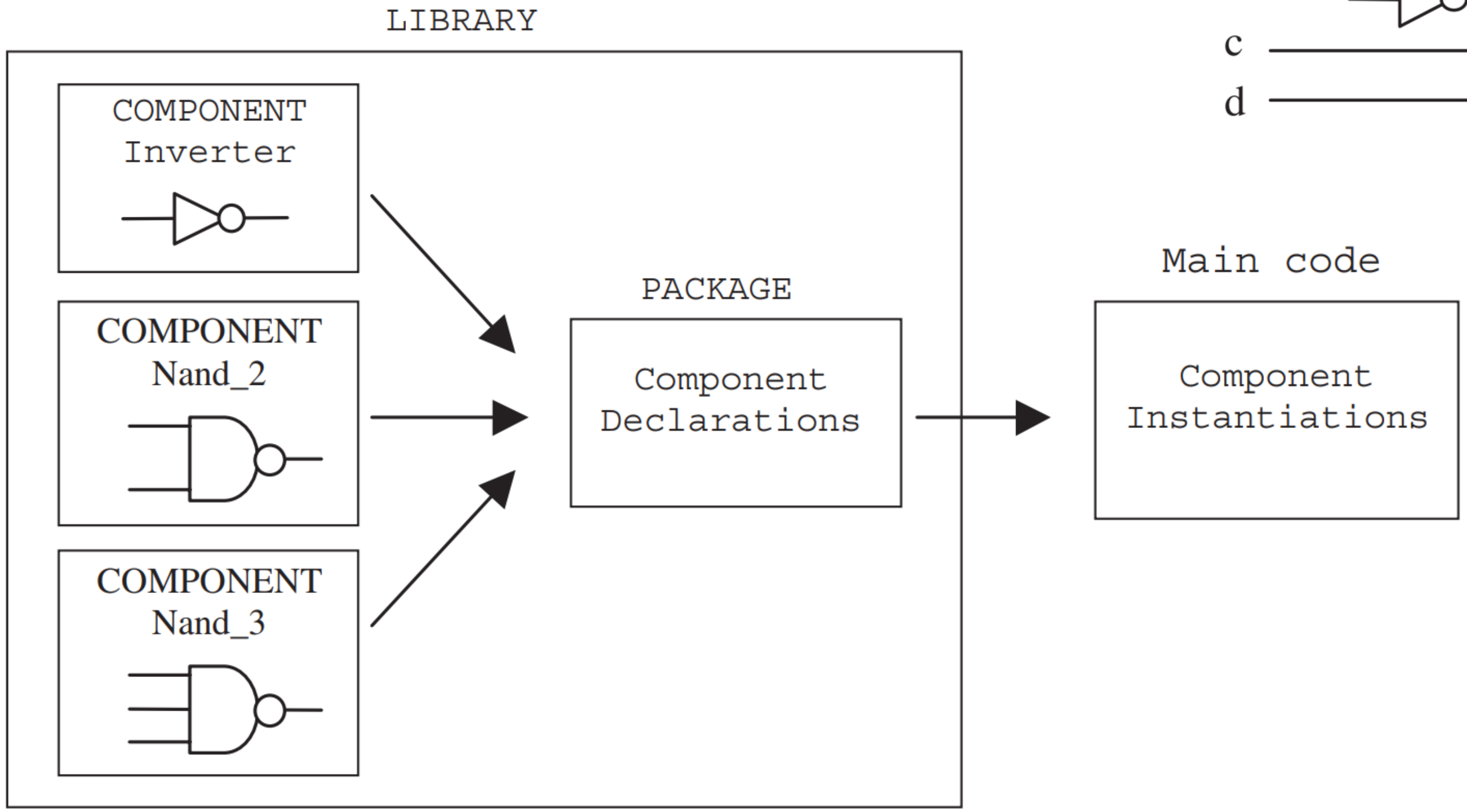
Component



Example 2: Components declared in a package

```
-----------------File project.vhd:---------------------
library ieee;
use ieee.std_logic_1164.all;
use work.my_components.all;
------------------------------------------------------
entity project is
       port(a, b, c, d: in std_logic; x, y: out std_logic);
end project;
------------------------------------------------------
architecture structural of project is
       signal w: std_logic;
begin
       U1: inverter port map (b, w);
       U2: nand_2 port map (a, b, x);
       U3: nand_3 port map (w, c, d, y);
end structural;
------------------------------------------------------
```


a
b
c
d
x
y


LIBRARY
COMPONENT
Inverter
COMPONENT
Nand_2
COMPONENT
Nand_3
PACKAGE
Component
Declarations
Main code
Component
Instantiations

Function in package

Example: Function located in a package

```
------ Package: ------------------------------
library ieee;
use ieee.std_logic_1164.all;
----------------------------------------------
package my_package is
       function positive_edge(signal s: std_logic) return boolean;
end my_package;
----------------------------------------------
package body my_package is
       function positive_edge(signal s: std_logic)
              return boolean is
       begin
              return s'event and s='1';
       end positive_edge;
end my_package;
```

Function in package

Example: Function located in a package

```
------ Main code: -----------------------------------
library ieee;
use ieee.std_logic_1164.all;
use work.my_package.all;
-----------------------------------------------------
entity dff is
       port( d, clk, rst: in std_logic;
             q: out std_logic);
end dff;
-----------------------------------------------------
architecture my_arch of dff is
begin
       process (clk, rst)
       begin
              if (rst='1') then q <= '0';
              elsif positive_edge(clk) then q<=d;
              end if;
       end process;
end my_arch;
```

Procedure in package

Example: Procedure located in a package

```
--------- Package: --------------------
library ieee;
use ieee.std_logic_1164.all;
---------------------------------------
package my_package is
        constant limit: integer := 255;
        procedure sort (signal in1, in2: in integer range 0 to limit;
                signal min, max: out integer range 0 to limit);
end my_package;
---------------------------------------
package body my_package is
        procedure sort (signal in1, in2: in integer range 0 to limit;
                signal min, max: out integer range 0 to limit) is
        begin
                if (in1 > in2) then
                        max <= in1;
                        min <= in2;
                else
                        max <= in2;
                        min <= in1;
                end if;
        end sort;
end my_package;
---------------------------------------
```

Procedure in package

Example: Procedure located in a package

```
----------- Main code:------------------
library ieee;
use ieee.std_logic_1164.all;
use work.my_package.all;
----------------------------------------
entity min_max is
    generic (limit: integer := 255);
    port ( ena: in bit;
           inp1, inp2: in integer range 0 to limit;
           min_out, max_out: out integer range 0 to limit);
end min_max;
----------------------------------------
architecture my_architecture of min_max is
begin
    process(ena)
        begin
            if (ena='1') then sort(inp1, inp2, min_out, max_out);
        end if;
    end process;
end my_architecture;
----------------------------------------
```

Chapter 6 Additional Topics in VHDL

6.3.1 Signal Attributes

Attributes can be associated with **signals** and **arrays**

```
Example:

    CLOCK'EVENT
```

'EVENT returns **TRUE** if a change in signal CLOCK has just occurred

VHDL has two types of attributes:

- ☐ attributes that return a **value**
- ☐ attributes that return a **signal**

6.3.1 Attributes （属性语句）

Event & Transaction

	occurs means:
Event (事件)	a change in the signal
Transaction （事务）	the signal is evaluated, regardless of whether the signal changes or not

```
Example:

  A <= B and C
```

- If B = 0, a transaction occurs on A every time C changes (A is recomputed every time C changes)
- no event occurs

- If B = 1, an event and a transaction occur on A every time C changes (A changes when C changes)

Signal Attributes That Return a Value

Attribute	Returns a Value
S'EVENT	True if an event occurred dring the current delta, else false
S'ACTIVE	True if a transaction occurred during the current delta, else false
S'LAST_EVENT	Time elapsed since the previous event on S
S'LAST_VALUE	Value of S before the previous event on S
S'LAST_ACTIVE	Time elapsed since previous transaction on S

If S changes at time T, then S'EVENT is true at T but false at T+Δ

Signal Attributes That Create a Signal

Attribute	Creates a Signal
S'DELAYED[(time)]	Signal same as S delayed by specified time
S'STABLE[(time)]	Boolean signal that is true if S had no events for the specified time
S'QUIET[(time)]	Boolean signal that is true if S had no transactions for the specified time
S'TRANSACTION	Signal of type bit that changes for every transaction on S

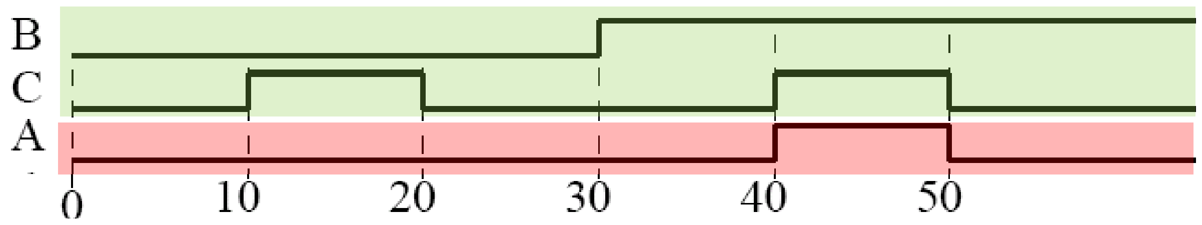
- [(time)]: time is optional
- If (time) is omitted, one **Δ** is used

6.3 Attributes （属性语句）

```
entity attr_ex is
    port (B,C : in bit);
end attr_ex;

architecture test of attr_ex is
    signal A, C_delayed5, A_trans : bit;
    signal A_stable5, A_quiet5 : boolean;
begin
    A <= B and C;
    C_delayed5 <= C'delayed(5 ns);
    A_trans <= A'transaction;
    A_stable5 <= A'stable(5 ns);
    A_quiet5 <= A'quiet(5 ns);
end test;
```

Note: the types

Waveforms for Attribute Test

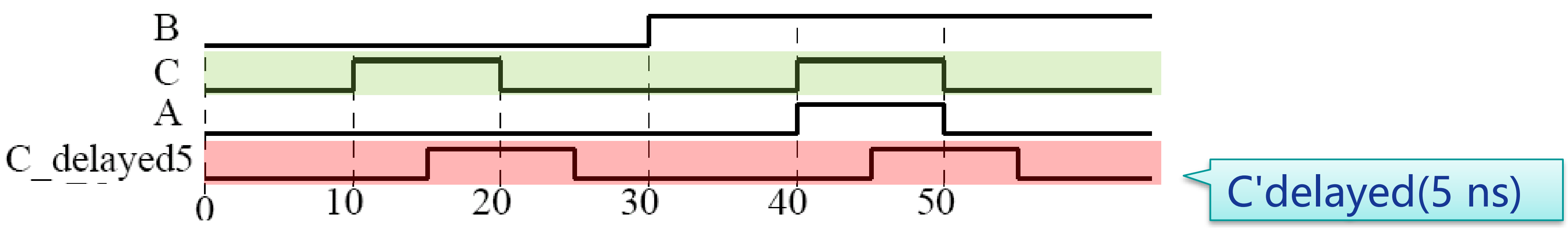


A <= B and C;

6.3 Attributes （属性语句）

```
entity attr_ex is
    port (B,C : in bit);
end attr_ex;

architecture test of attr_ex is
    signal A, C_delayed5, A_trans : bit;
    signal A_stable5, A_quiet5 : boolean;
begin
    A <= B and C;
    C_delayed5 <= C'delayed(5 ns);
    A_trans <= A'transaction;
    A_stable5 <= A'stable(5 ns);
    A_quiet5 <= A'quiet(5 ns);
end test;
```

Waveforms for Attribute Test

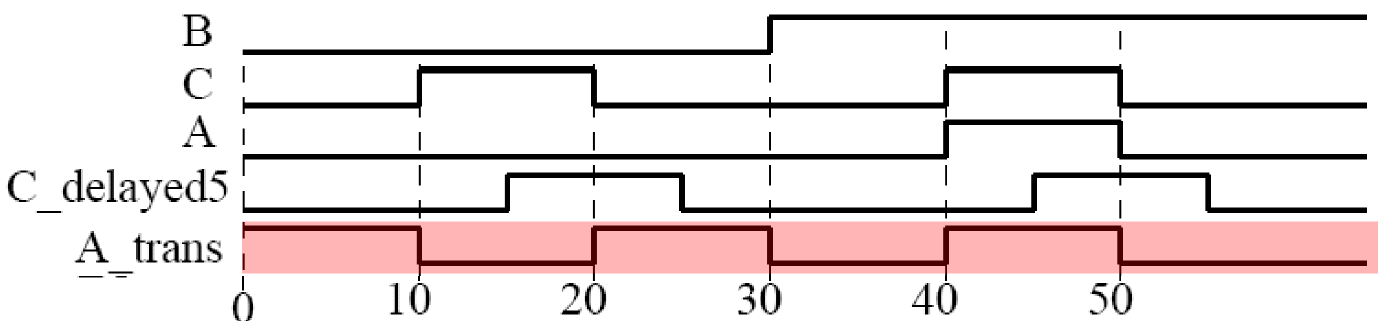


6.3 Attributes （属性语句）

```
entity attr_ex is
   port (B,C : in bit);
end attr_ex;

architecture test of attr_ex is
   signal A, C_delayed5, A_trans : bit;
   signal A_stable5, A_quiet5 : boolean;
begin
   A <= B and C;
   C_delayed5 <= C'delayed(5 ns);
   A_trans <= A'transaction;
   A_stable5 <= A'stable(5 ns);
   A_quiet5 <= A'quiet(5 ns);
end test;
```

- The initial computation produces a transaction on A at time = delta
- A_trans changes to '1' at that time

Waveforms for Attribute Test



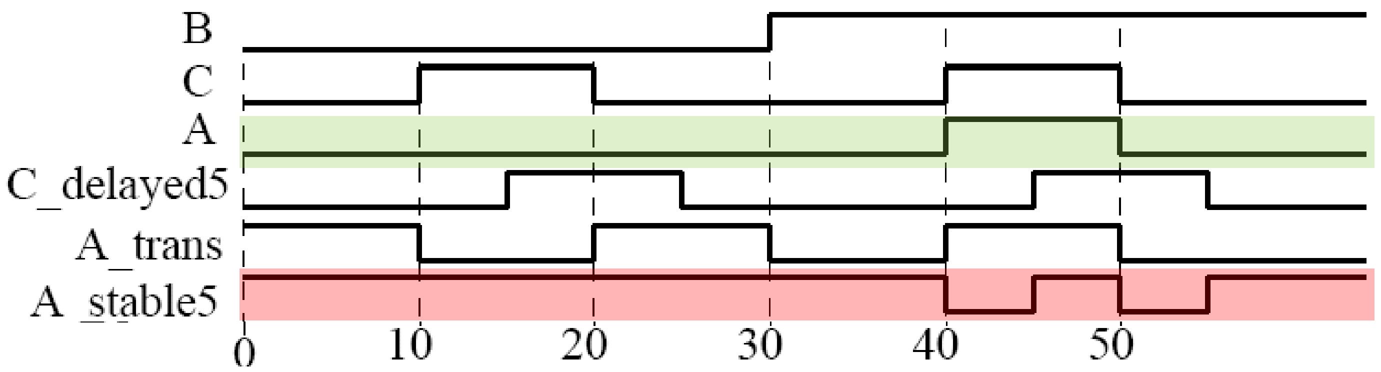
A'transaction

6.3 Attributes （属性语句）

```
entity attr_ex is
    port (B,C : in bit);
end attr_ex;

architecture test of attr_ex is
    signal A, C_delayed5, A_trans : bit;
    signal A_stable5, A_quiet5 : boolean;
begin
    A <= B and C;
    C_delayed5 <= C'delayed(5 ns);
    A_trans <= A'transaction;
    A_stable5 <= A'stable(5 ns);
    A_quiet5 <= A'quiet(5 ns);
end test;
```

A'STABLE(time) is true if A has not changed during the preceding interval of length (time)

Waveforms for Attribute Test



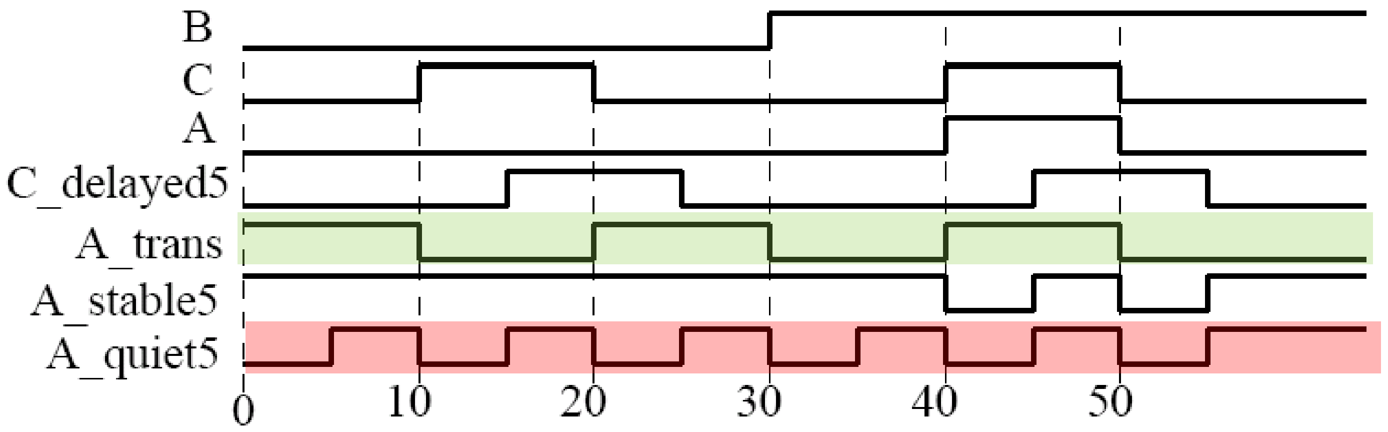
A'stable(5 ns)

6.3 Attributes （属性语句）

```
entity attr_ex is
   port (B,C : in bit);
end attr_ex;

architecture test of attr_ex is
   signal A, C_delayed5, A_trans : bit;
   signal A_stable5, A_quiet5 : boolean;
begin
   A <= B and C;
   C_delayed5 <= C'delayed(5 ns);
   A_trans <= A'transaction;
   A_stable5 <= A'stable(5 ns);
   A_quiet5 <= A'quiet(5 ns);
end test;
```

A'QUIET(time) is true if A has no transactionss during the preceding interval of length (time)

Waveforms for Attribute Test



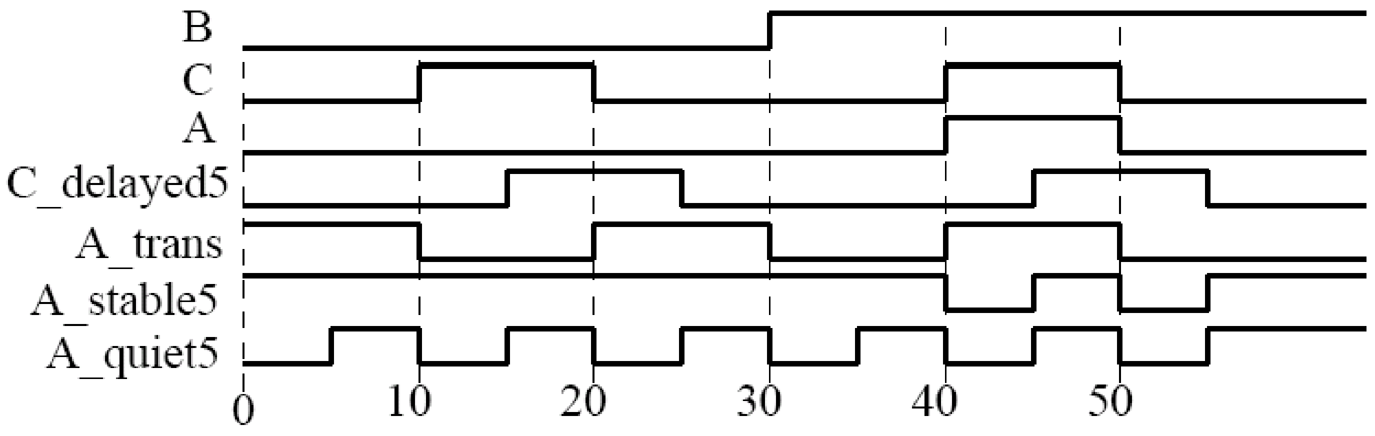
A'quiet(5 ns)

6.3 Attributes （属性语句）

```
entity attr_ex is
   port (B,C : in bit);
end attr_ex;

architecture test of attr_ex is
   signal A, C_delayed5, A_trans : bit;
   signal A_stable5, A_quiet5 : boolean;
begin
   A <= B and C;
   C_delayed5 <= C'delayed(5 ns);
   A_trans <= A'transaction;
   A_stable5 <= A'stable(5 ns);
   A_quiet5 <= A'quiet(5 ns);
end test;
```

S'STABEL and S'QUIET creates **boolean** signals

Waveforms for Attribute Test



6.3.2 Array Attributes

Attribute	Returns	Examples
A'LEFT(N)	left bound of Nth index range	
A'RIGHT(N)	right bound of Nth index range	
A'HIGH(N)	largest bound of Nth index range	
A'LOW(N)	smallest bound of Nth index range	
A'RANGE(N)	Nth index range	
A'REVERSE_RANGE(N)	Nth index range reversed	
A'LENGTH(N)	size of Nth index range	

A can either be an array name or an array type

6.3.2 Array Attributes

```
Type ROM is array (0 to 15, 7 downto 0) of bit;
Signal ROM1 : ROM;
```

Attribute	Returns	Examples
A'LEFT(N)	left bound of Nth index range	
A'RIGHT(N)	right bound of Nth index range	
A'HIGH(N)	largest bound of Nth index range	
A'LOW(N)	smallest bound of Nth index range	
A'RANGE(N)	Nth index range	
A'REVERSE_RANGE(N)	Nth index range reversed	
A'LENGTH(N)	size of Nth index range	

6.3.2 Array Attributes

```
Type ROM is array (0 to 15, 7 downto 0) of bit;
Signal ROM1 : ROM;
```

Attribute	Returns	Examples
A'LEFT(N)	left bound of Nth index range	ROM1'LEFT(1)=0 ROM1'LEFT(2) = 7
A'RIGHT(N)	right bound of Nth index range	ROM1'RIGHT(1) = 15 ROM1'RIGHT(2) = 0
A'HIGH(N)	largest bound of Nth index range	ROM1'HIGH(1) = 15 ROM1'HIGH(2) = 7
A'LOW(N)	smallest bound of Nth index range	ROM1'LOW(1) = 0 ROM1'LOW(2) = 0
A'RANGE(N)	Nth index range	ROM1'RANGE(1) = 0 to 15 ROM1'RANGE(2) = 7 downto 0
A'REVERSE_RANGE(N)	Nth index range reversed	ROM1'REVERSE_RANGE(1) = 15 downto 0 ROM1'REVERSE_RANGE(2) = 0 to 7
A'LENGTH(N)	size of Nth index range	ROM1'LENGTH(1) = 16 ROM1'LENGTH(2) = 8

6.3.2 Array Attributes

```
Type ROM is array (0 to 15, 7 downto 0) of bit;
Signal ROM1 : ROM;
```

Array attributes also work with array **constants** and array **variables**

Attribute	Retur	
A'LEFT(N)	left bound of Nth index range	ROM1'LEFT(1)=0 ROM1'LEFT(2) = 7
A'RIGHT(N)	right bound of Nth index range	ROM1'RIGHT(1) = 15 ROM1'RIGHT(2) = 0
A'HIGH(N)	largest bound of Nth index range	ROM1'HIGH(1) = 15 ROM1'HIGH(2) = 7
A'LOW(N)	smallest bound of Nth index range	ROM1'LOW(1) = 0 ROM1'LOW(2) = 0
A'RANGE(N)	Nth index range	ROM1'RANGE(1) = 0 to 15 ROM1'RANGE(2) = 7 downto 0
A'REVERSE_RANGE(N)	Nth index range reversed	ROM1'REVERSE_RANGE(1) = 15 downto 0 ROM1'REVERSE_RANGE(2) = 0 to 7
A'LENGTH(N)	size of Nth index range	ROM1'LENGTH(1) = 16 ROM1'LENGTH(2) = 8

6.3.2 Array Attributes

```
Type ROM is array (0 to 15, 7 downto 0) of bit;
Signal ROM1 : ROM;
```

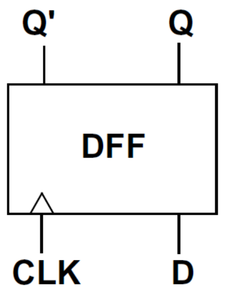
Attribute	Returns	Examples
A'LEFT(N)	left bound of Nth index range	ROM1'LEFT(1)=0 ROM1'LEFT(2) = 7
A'RIGHT(N)	right bound of Nth index range	ROM1'RIGHT(1) = 15 ROM1'RIGHT(2) = 0
A'HIGH(N)	largest bound of Nth index range	ROM1'HIGH(1) = 15 ROM1'HIGH(2) = 7
A'LOW(N)	smallest bound of Nth index range	ROM1'LOW(1) = 0 ROM1'LOW(2) = 0
A'RANGE(N)	Nth index range	ROM1'RANGE(1) = 0 to 15 ROM1'RANGE(2) = 7 downto 0
A'REVERSE_RANGE(N)	Nth index range reversed	ROM1'REVERSE_RANGE(1) = 15 downto 0 ROM1'REVERSE_RANGE(2) = 0 to 7
A'LENGTH(N)	size of Nth index range	ROM1'LENGTH(1) = 16 ROM1'LENGTH(2) = 8

If N is 1 (one-dimensional array) , it can be omitted

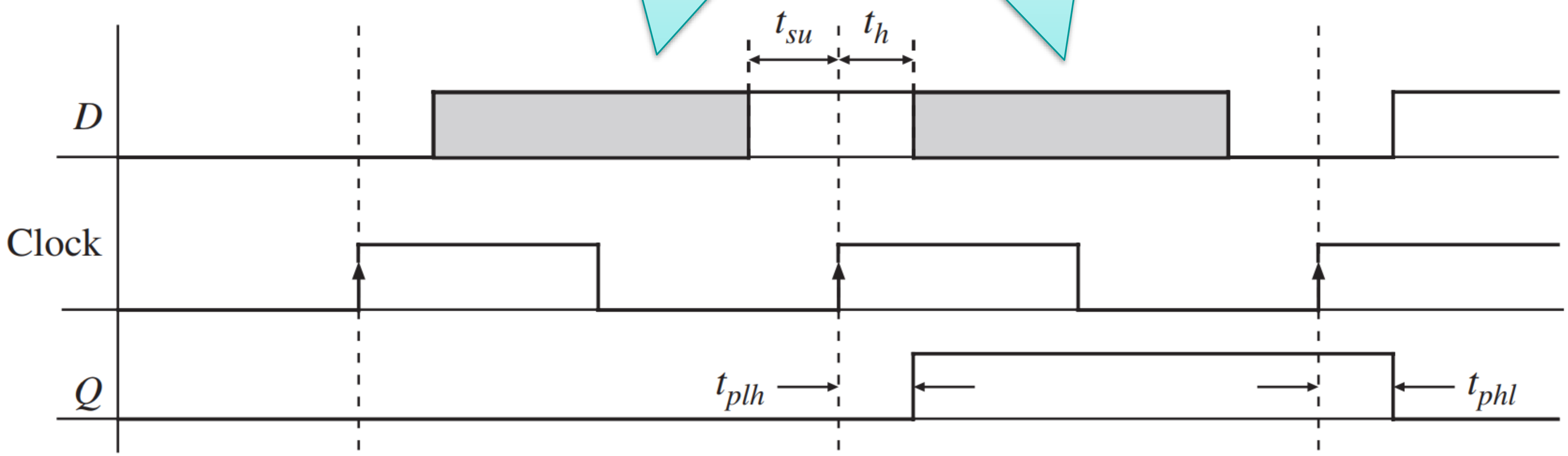
- Attributes are often used together with assert statements for error checking
- The assert statement checks to see if a certain condition is true and , if not, causes an error message to be displayed

We present two examples:
- one illustrating use of signal attributes
- one illustrating use of array attributes

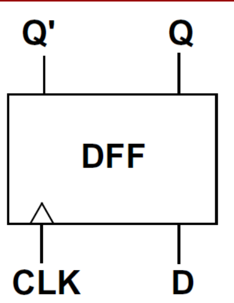
6.3.3 Use of Attributes



D can changes at any time during the shaded region

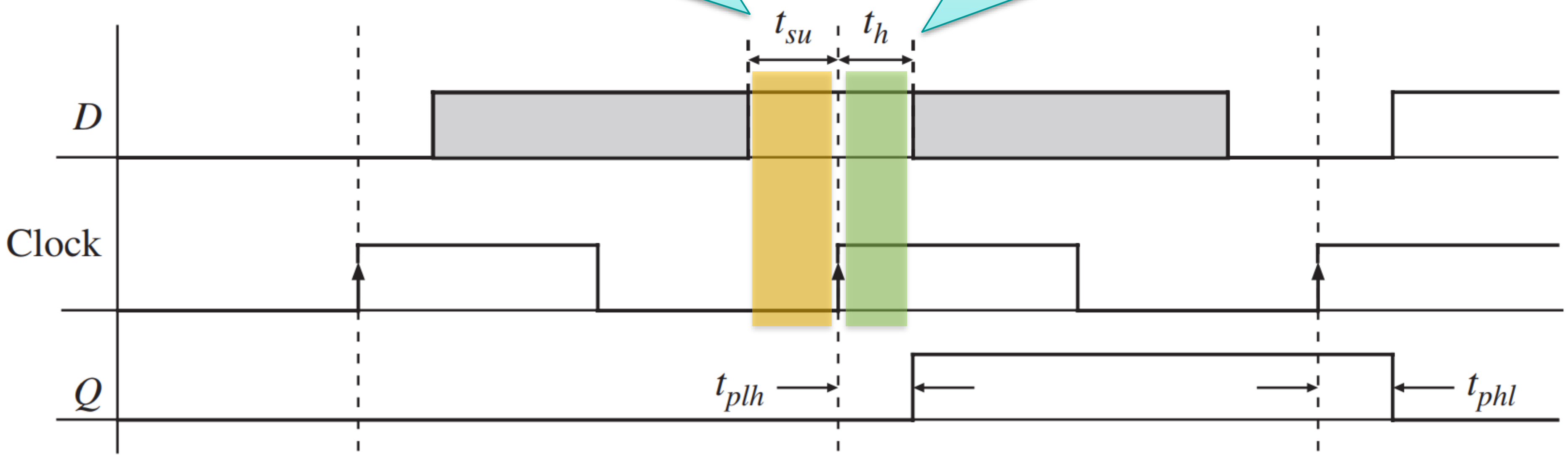


6.3.3 Use of Attributes



Setup time (建立时间): D input must be stable for a certain amount of time before the active edge of the clock

Hold time(保持时间): D must be stable for a certain amount of time after the active edge of the clock



6.3.3 Use of Attributes

Example: Use of Signal Attributes

```
check: process
begin
      wait until rising_edge(Clk);
      assert (D’stable(setup_time))
            report (“setup time violation”)
            severity error;
      wait for hold_time;
      assert (D’stable(hold_time))
            report (“Hold time violation”)
            severity error;
end process check;
```

The process checks to see if the setup and hold time are satisfied for a D flip-flop

Example: Use of Signal Attributes

```
check: process
begin
      wait until (Clk'event and CLK = '1');
      assert (D’stable(setup_time))
            report (“setup time violation”)
            severity error;
      wait for hold_time;
      assert (D’stable(hold_time))
            report (“Hole time violation”)
            severity error;
end process check;
```

- After active edge of the clocks occurs, D input is check to see if it has been stable for setup_time
- **If not**, a violation is reported as an error

Example: Use of Signal Attributes

```
check: process
begin
      wait until (Clk'event and CLK = '1');
      assert (D’stable(setup_time))
            report (“setup time violation”)
            severity error;
      wait for hold_time;
      assert (D’stable(hold_time))
            report (“Hole time violation”)
            severity error;
end process check;
```

- Then, after waiting for hold_time, D is checked to see if it has been stable during hold-time
- **If not**, a violation is reported as an error

6.3 Attributes （属性语句）

Example: Use of Array Attributes in Vector Addition

```
procedure Addvec
   (Add1,Add2: in bit_vector;
      Cin: in bit;
      signal Sum: out bit_vector;
      signal Cout: out bit;
      n:in positive) is
      variable C: bit;
```

In Section 6.2, Addvec( ) need parameter **n** to specify length of Add1 and Add2

6.3 Attributes （属性语句）

Example: Use of Array Attributes in Vector Addition

```
procedure Addvec2
    (Add1,Add2: in bit_vector;
    Cin: in bit;
    signal Sum: out bit_vector;
    signal Cout: out bit) is
    variable C: bit := Cin;
    alias n1 : bit_vector(Add1'length-1 downto 0) is Add1;
    alias n2 : bit_vector(Add2'length-1 downto 0) is Add2;
    alias S : bit_vector(Sum'length-1 downto 0) is Sum;

begin
    assert ((n1'length = n2'length) and (n1'length = S'length))
        report "Vector lengths must be equal!"
        severity error;
    for i in s'reverse_range loop
        S(i) <= n1(i) xor n2(i) xor C;
        C := (n1(i) and n2(i)) or (n1(i) and C) or (n2(i) and C);
    end loop;
    Cout <= C;
end Addvec2;
```

- Here Addvec2( ) does not required to pass length of arrays
- The inputs to procedure include only Add1, Add2, Cin

6. 3 Attributes （属性语句）

Use of Array Attributes in Vector Addition

```
procedure Addvec2
   (Add1,Add2: in bit_vector;
   Cin: in bit;
   signal Sum: out bit_vector;
   signal Cout: out bit) is
   variable C: bit := Cin;
   alias n1 : bit_vector(Add1'length-1 downto 0) is Add1;
   alias n2 : bit_vector(Add2'length-1 downto 0) is Add2;
   alias S : bit_vector(Sum'length-1 downto 0) is Sum;

begin
   assert ((n1'lengt
      report "Vector
      severity error;
   for i in s'reverse_range loop
      S(i) <= n1(i) xor n2(i) xor C;
      C := (n1(i) and n2(i)) or (n1(i) and C) or (n2(i) and C);
   end loop;
   Cout <= C;
end Addvec2;
```

- Addvec2() creates aliases n1, n2, and S
- n1, n2, S have same length as Add1, Add2, and Sum, respectively

The ranges for n1, n2, and S are defined in a uniform manner to faciliate further computation

6.3 Attributes （属性语句）

Use of Array Attributes in Vector Addition

```
procedure Addvec2
   (Add1,Add2: in bit_vector;
   Cin: in bit;
   signal Sum: out bit_vector;
   signal Cout: out bit) is
   variable C: bit := Cin;
   alias n1 : bit_vector(Add1'length-1 downto 0) is Add1;
   alias n2 : bit_vector(Add2'length-1 downto 0) is Add2;
   alias S : bit_vect

begin
   assert ((n1'length = n2'length) and (n1'length = S'length))
      report "Vector lengths must be equal!"
      severity error;
   for i in s'reverse_range loop
      S(i) <= n1(i) xor n2(i) xor C;
      C := (n1(i) and n2(i)) or (n1(i) and C) or (n2(i) and C);
   end loop;
   Cout <= C;
end Addvec2;
```

Addvec2() also uses array attributes and checks whether the lengths are equal

6.3 Attributes （属性语句）

Use of Array Attributes in Vector Addition

```
procedure Addvec2
   (Add1,Add2: in bit_vector;
   Cin: in bit;
   signal Sum: out bit_vector;
   signal Cout: out bit) is
   variable C: bit := Cin;
   alias n1 : bit_vector(Add1'length-1 downto 0) is Add1;
   alias n2 : bit_vector(Add2'length-1 downto 0) is Add2;
   alias S : bit_vector(Sum'length-1 downto 0) is Sum;

begin
   assert ((n1'length = n2'length) and (n1'length = S'length))
      report "Vector lengths must be equal!"
      severity error;
   for i in s'reverse_range loop
      S(i) <= n1(i) xor n2(i) xor C;
      C := (n1(i) and n2(i)) or (n1(i) and C) or (n2(i) and C);
   end loop;
   Cout <= C;
end Addvec2;
```

Since this loop must start with i=0, the range of i is the reverse of the range for S

Chapter 6 Additional Topics in VHDL

6.4 Creating Overloading Operators

- Operator overloading means that we will extend the definition of the operator to other data types in addition to the default data types that have already been defined
- The operator will **implicitly** call an appropriate function, which eliminates the need for an explicit function or procedure call
- Overloading can also be applied to procedures and functions

6.4 Creating Overloading Operators

The VHDL arithmetic operators, + and -, are defined to operate on integers, but not on bit-vectors

```
function add4 (A,B: bit_vector(3 downto 0); carry: bit)
   return bit_vector is
```

```
  Z <= add4(X, not Y, '1');
```

In Section 6.1, Add4() should be called explicitly

6.4 Creating Overloading Operators

```
package bit_overload is
  function "+" (Add1, Add2: bit_vector)
    return bit_vector;
end bit_overload;

package body bit_overload is
  function "+" (Add1, Add2: bit_vector)
    return bit_vector is
  variable sum: bit_vector(Add1'length-1 downto 0);
  variable c: bit := '0';              -- no carry in
  alias n1: bit_vector(Add1'length-1 downto 0) is Add1;
  alias n2: bit_vector(Add2'length-1 downto 0) is Add2;
  begin
    for i in sum'reverse_range loop
      sum(i) := n1(i) xor n2(i) xor c;
      c := (n1(i) and n2(i)) or (n1(i) and c) or (n2(i) and c);
    end loop;
    return (sum);
  end "+";
end bit_overload;
```

" " indicates this function is an operator overloading function

assumes the lengths of Add1 and Add2 are the same

sum's range reversed