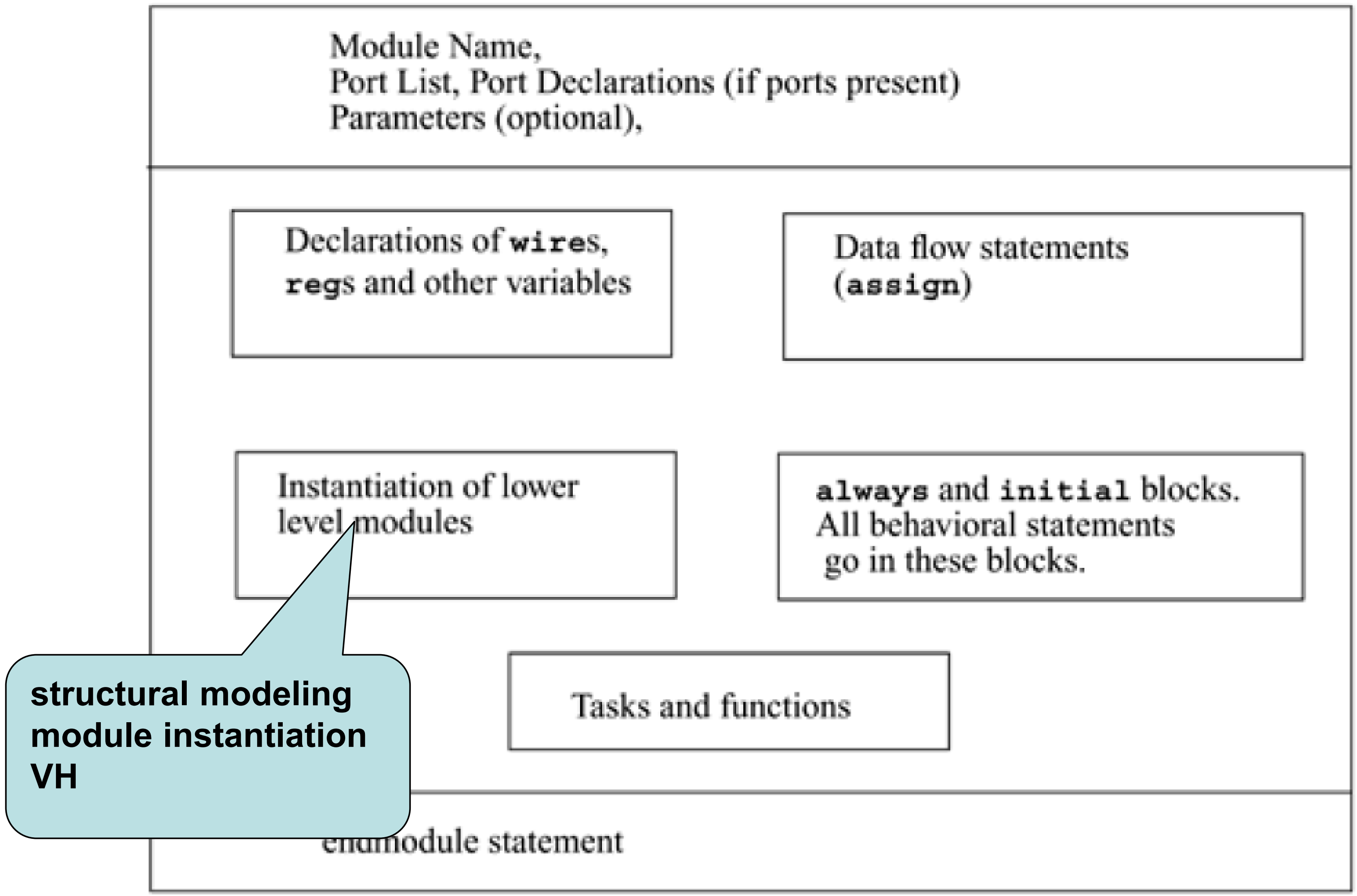
VHDL
packages and components

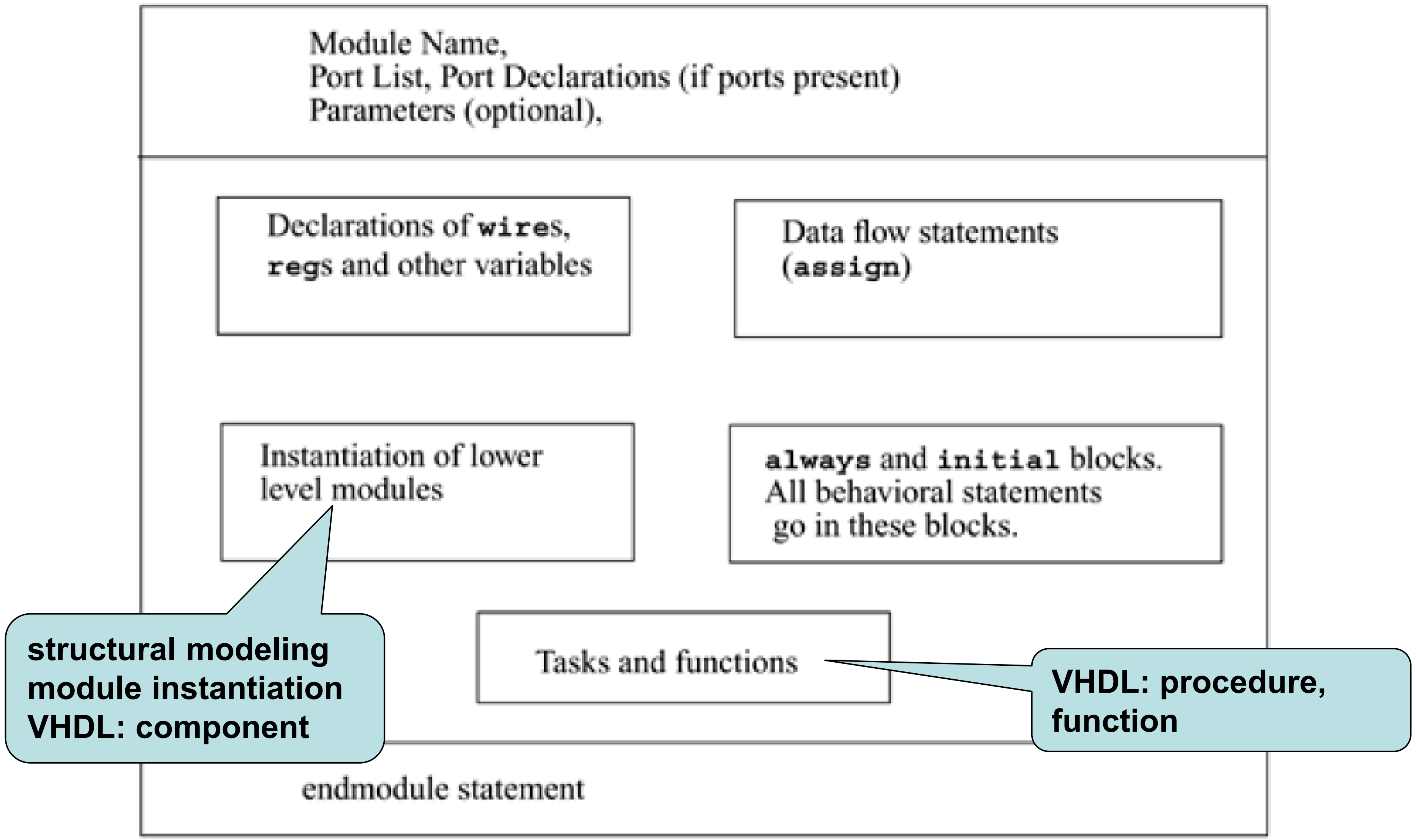
outlines

- package and component
- function and procedure
- configuration

Verilog Module (Structure-Level)


Module Name,
Port List, Port Declarations (if ports present)
Parameters (optional),
Declarations of wires, regs and other variables
Data flow statements (assign)
Instantiation of lower level modules
always and initial blocks. All behavioral statements go in these blocks.
Tasks and functions
endmodule statement
structural modeling
module instantiation
VH

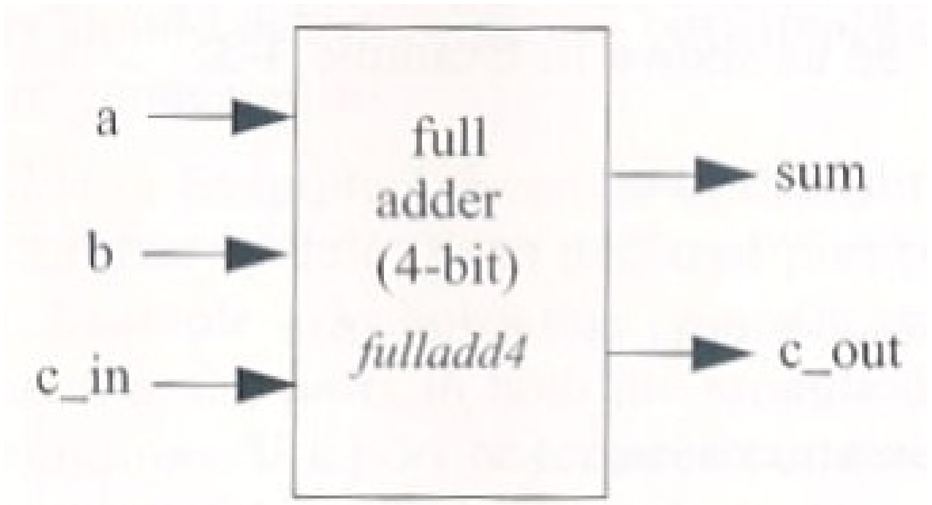
Verilog Module (Structure-Level)


Module Name,
Port List, Port Declarations (if ports present)
Parameters (optional),
Declarations of wires, regs and other variables
Data flow statements (assign)
Instantiation of lower level modules
always and initial blocks. All behavioral statements go in these blocks.
Tasks and functions
endmodule statement
structural modeling module instantiation VHDL: component
VHDL: procedure, function

Verilog Port Declaration

```
// IEEE 1364-1995 (old)
module fulladd4(sum, c_out, a, b, c_in);
parameter width = 4;
output [width-1:0] sum;
output c_out;
input [width-1:0] a, b;
input c_in;
reg [width-1:0] sum;
reg c_out;
…
endmodule
```

```
// ANSI C style (IEEE 1364-2001)
module
      # (parameter width =4)
      fulladd4 (
      output reg [width-1:0] sum,
      output reg c_out;
      input [width-1:0] a, b;
      input c_in
);
…
endmodule
```



Verilog: Connecting Ports to External Signals

- Module instantiation
- connecting by order list

```
 // instantiate   fulladd4 (sum, c_out, a, b, c_in)
 // call it fa_ordered, signals are connected to ports in order (by position)

  fulladd4   fa_ordered  (EXT_SUM, EXT_C_OUT, EXT_A, EXT_B, EXT_C_IN);
```

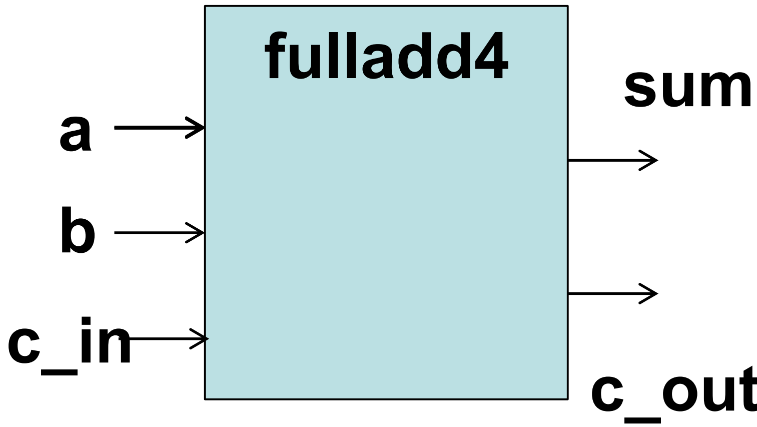
- connecting by name

```
 // instantiate module fa_byname and connect signals to ports by name

  fulladd4   fa_byname
  (.c_out(EXT_C_OUT), .sum(EXT_SUM),  .b(EXT_B), .c_in(EXT_C_IN), .a(EXT_A));
```

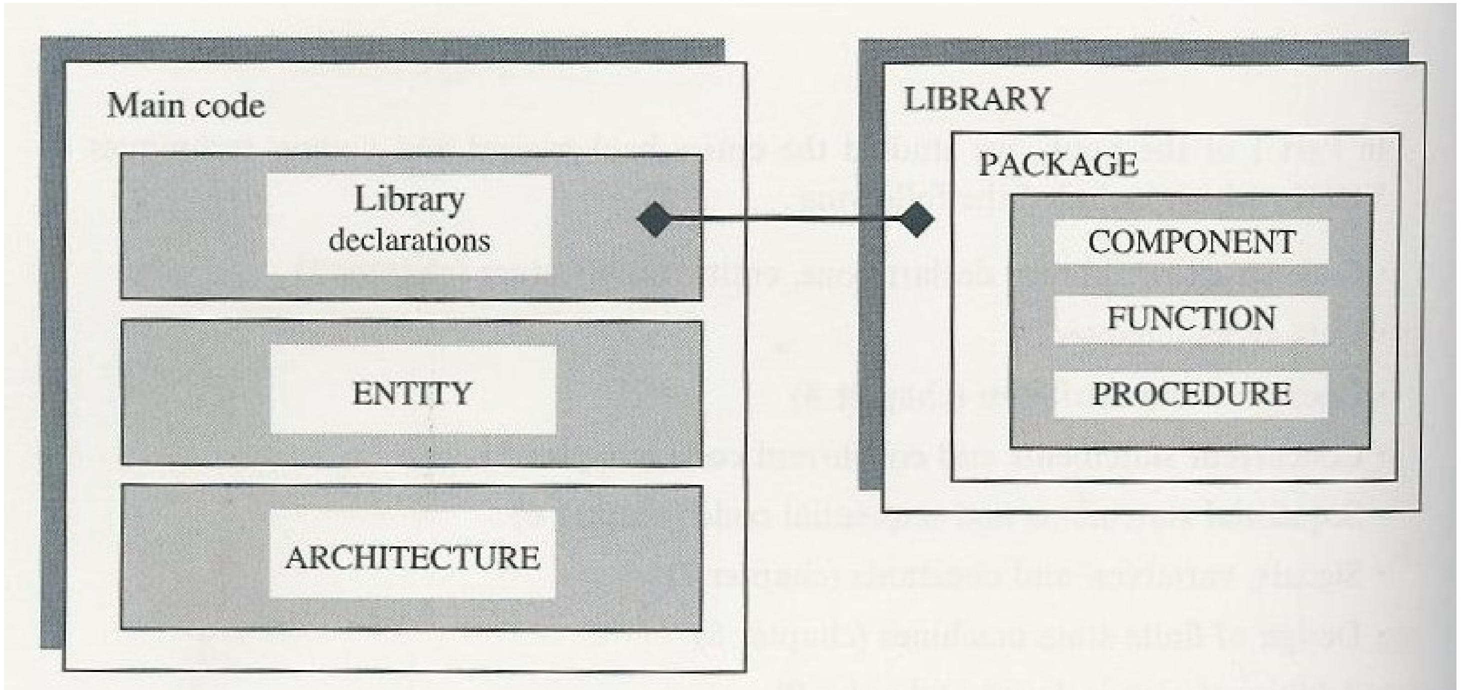
```
module fulladd4 (sum, c_out, a, b, c);

output [3:0] sum;
output c_cout;
input [3:0] a, b;
input c_in;
....
endmodule
```



Fundamental units of VHDL

- Library
  - Package
    - **component, function, procedure**
- Entity
- Architecture



VHDL PACKAGE

- **PACKAGE**
  - Contains all declarations
- **PACKAGE BODY**
  - Necessary only when FUNCTION, or PROCEDURE are declared in PACKAGE

```
PACKAGE package_name IS
        (declarations)
END package_name;

PACKAGE BODY package_name IS
        (FUNCTION and PROCEDURE descriptions)
END package_name;
```

Example : simple package: define types and constants

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
--------------------------------------------------------------------------------
--- my_package contains three declarations: states, color, vec
PACKAGE my_package IS
            TYPE states IS (st1, st2, st3, st4);
            TYPE colors IS (red, green, blue);
            CONSTANT vec: STD_LOGIC_VECTOR(7 DOWNTO 0) := “11111111”;
END my_package;
```

Example : package with a function (need package body)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------------------
PACKAGE my_package IS
        TYPE states IS (st1, st2, st3, st4);
        TYPE colors 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;
```

VHDL component

- hierarchical design, code sharing and reuse
- Declaration

```
COMPONENT component_name IS
PORT (
            port_name : signal_mode  signal_type;
            port_name : signal_mode  signal_type; …);
END COMPONENT;
```

- Instantiation

```
Label: component_name  PORT MAP (port_list);
```

- example

```
------------ COMPONENT declaration -------------------
COMPONENT inverter IS
           PORT ( a: IN STD_LOGIC;    b: OUT  STD_LOGIC);
END COMPONENT;

------------ COMPONENT instantiation -------------------
U1: inverter PORT MAP (x, y);
-- Verilog module instantiation
-- inverter U1 (x,y);
```

COMPONENT

- declaration in PACKAGE or main code
- instantiation in ARCHITECTURE, PACKAGE, or BLOCK

COMPONENT declaration:

```
COMPONENT component_name [IS]
   [GENERIC ( ... );]
   PORT ( ... );
END COMPONENT [component_name];
```

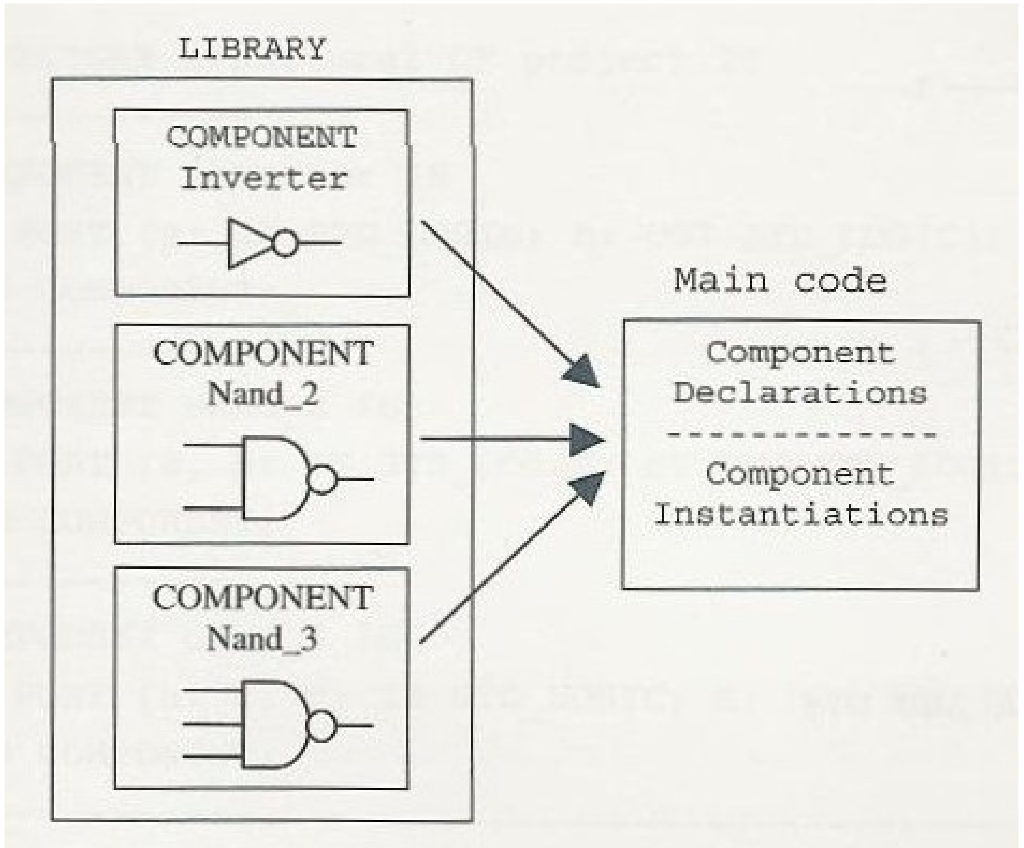
A copy of the entity

COMPONENT instantiation:

```
label: [COMPONENT] component_name
       [GENERIC MAP (generic_list)]
       PORT MAP (port_list);
```

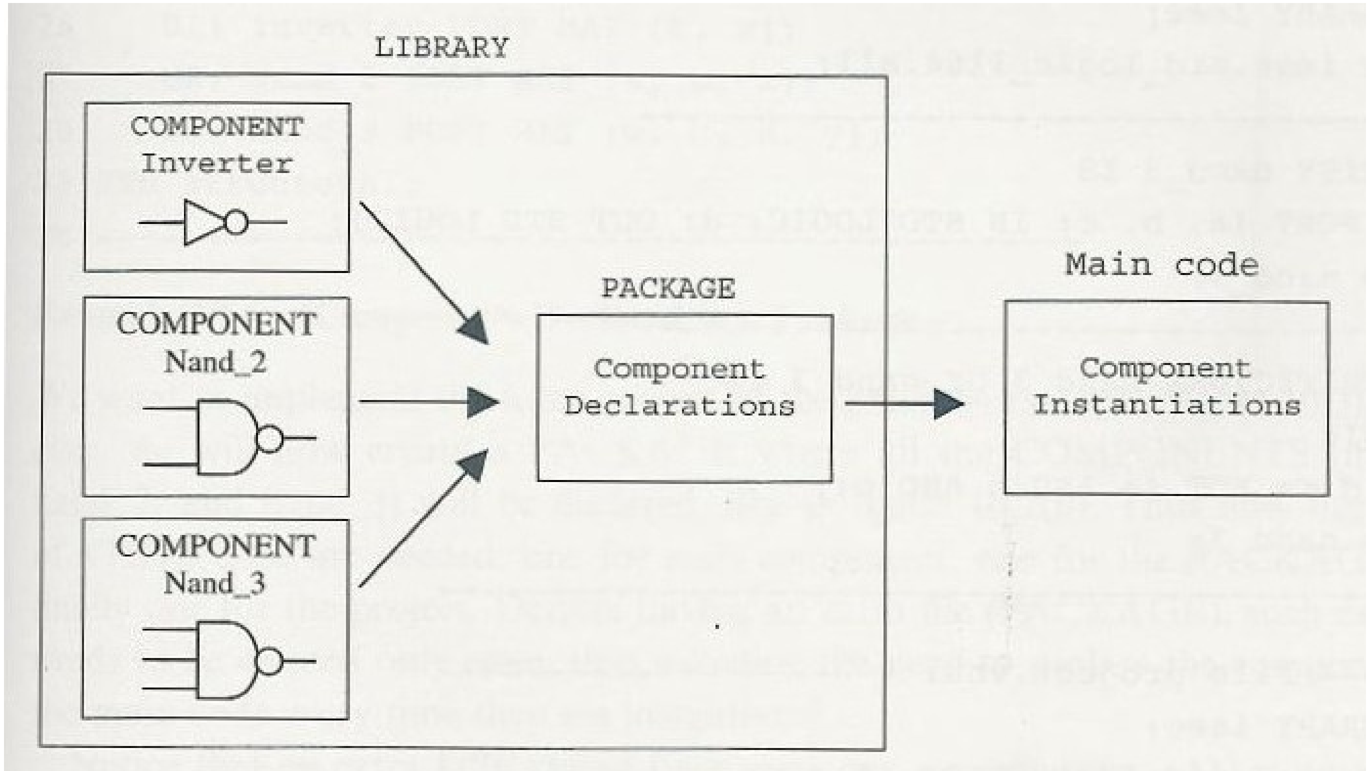
component declaration in main code

- component *declaration* and component *instantiation* can be in the main code
  - cannot be shared outside the main code



component declaration in package

- component declarations are in a package
  - allows reuse of the components
- the main code just include the library/package

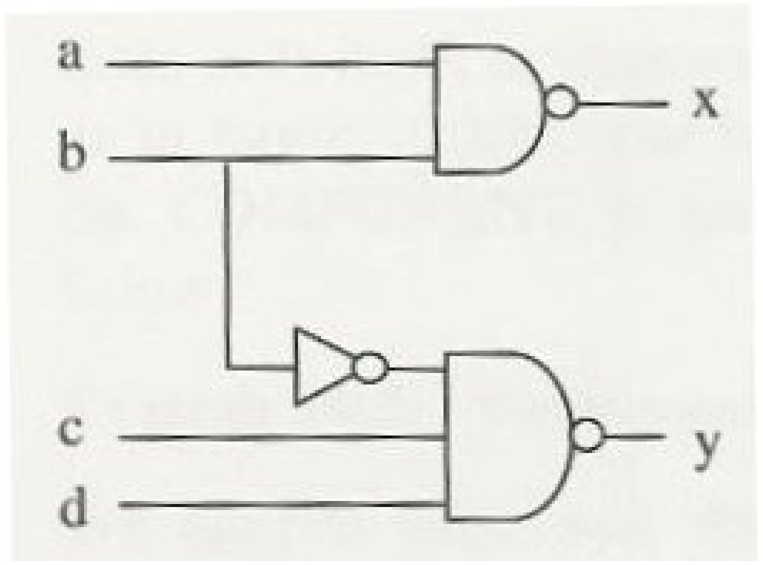


Example : components defined in three files

```
-- Part 1: file file1_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;
```

```
-- Part 2: file file2_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;
```

```
-- Part 3: file file3_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: component declaration/instantiation in the main code

```
-- Part 4: file file4_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      -- positional port mapping   versus   nominal (named) port mapping
  U1: inverter PORT MAP (b, w);        -- inverter PORT MAP (a=>b, b=>w);
  U2: nand_2 PORT MAP (a, b, x);       -- nand_2 PORT MAP (a=>a, b=>b, c=>x);
  U3: nand_3 PORT MAP (w, c, d, y); -- nand_3 PORT MAP (a=>w, b=>c, c=>d, d=>w);
END structural;
```

Example: component declaration in a package

```
-- Part 4: file file5_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;
```

Example: component instantiation in the main code

```
-- part 5: file file4_alternative_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;
```

PORT MAP

```
COMPONENT inverter IS
          PORT (a: IN STD_LOGIC;  b: OUT STD_LOGIC)
END COMPONENT;
```

- *Positional* mapping (by order)

```
U1: inverter PORT MAP (x, y);  -- VHDL
```

```
inverter U1 (x, y);                    // Verilog
```

- *Nominal* mapping (by name)

```
U2: inverter PORT MAP (b=>y, a=>x);  -- VHDL
```

```
inverter U2 (.b(y), .a(x));                    // Verilog
```

GENERIC MAP

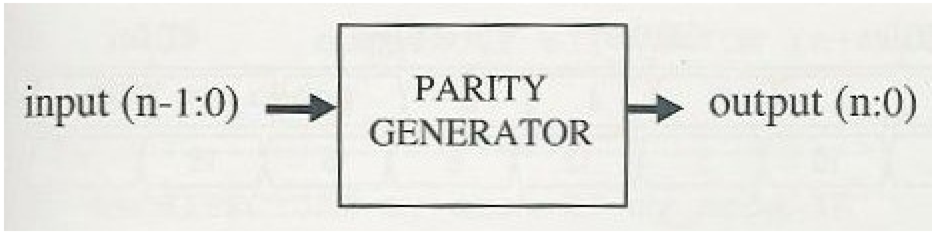
- Used in the COMPONENT instantiation
  - Pass information to GENERIC parameters

```
-- VHDL
Label: component_name
        GENERIC MAP (parameter list)
        PORT MAP (port list);
```

```
// Verilog
component_name # (parameter list) Label (port list);
parameter …
```

Example : generic parity generator

```
-- Part 1: file parity_gen.vhd (component)
LIBRARY ieee; USE ieee.std_logic_1164.all;
-----------------------------------
ENTITY parity_gen IS
GENERIC (n : INTEGER := 7);        -- default is 7
PORT (input: IN BIT_VECTOR (n DOWNTO 0);
output: OUT BIT_VECTOR (n+1 DOWNTO 0));
END parity_gen;
-----------------------------------
ARCHITECTURE parity OF parity_gen IS
BEGIN
PROCESS (input)
VARIABLE temp1: BIT;
VARIABLE temp2: BIT_VECTOR (output'RANGE);
BEGIN
temp1 := '0';
FOR i IN input'RANGE LOOP    temp1 := temp1 XOR input(i);    temp2(i) := input(i);
END LOOP;
temp2(output'HIGH) := temp1;
output <= temp2;
END PROCESS;
END parity;
```

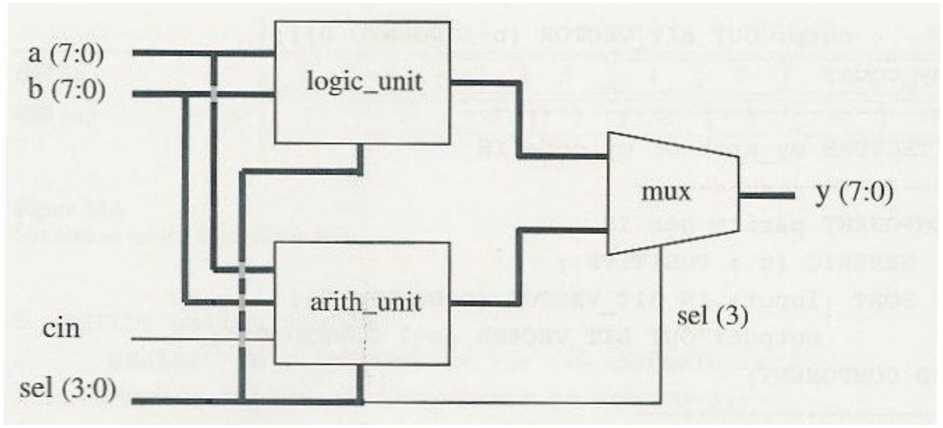


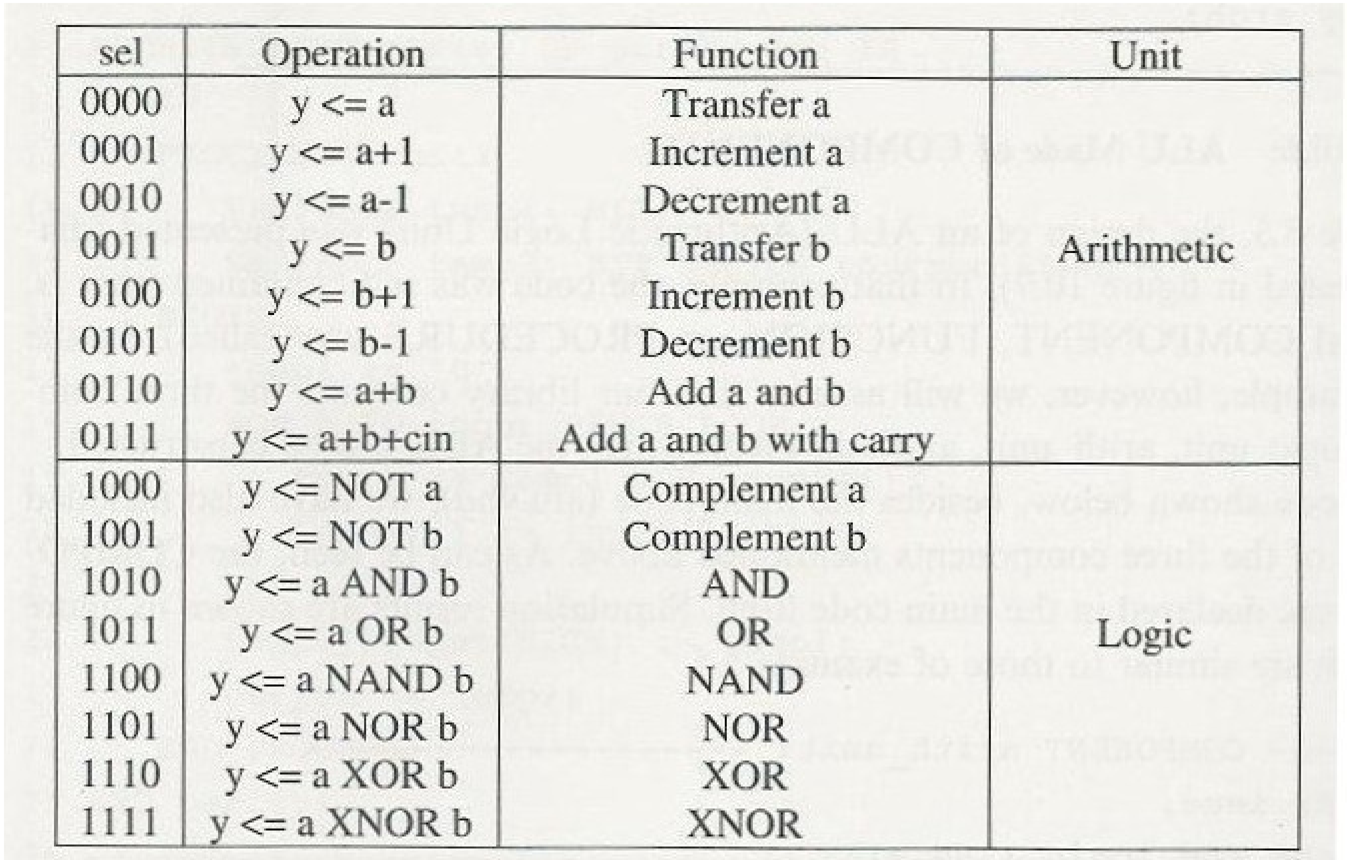
Example : project

```
-- Part 2: file my_code.vhd (actual project)
--------------------------------------------
LIBRARY ieee;    USE ieee.std_logic_1164.all;
----------------------------------
ENTITY my_code IS
        GENERIC (n : POSITIVE := 2);  -- 2 will overwrite 7
        PORT (inp: IN BIT_VECTOR (n DOWNTO 0);
                outp: OUT BIT_VECTOR (n+1 DOWNTO 0));
END my_code;
----------------------------------
ARCHITECTURE my_arch OF my_code IS
        -----------------------
        COMPONENT parity_gen IS     -- declaration of defined component
        GENERIC (n : POSITIVE);
        PORT (     input: IN     BIT_VECTOR (n DOWNTO 0);
                output: OUT BIT_VECTOR (n+1 DOWNTO 0));
        END COMPONENT;
        -----------------------
BEGIN    -- component instantiation
        C1: parity_gen
             GENERIC MAP (n)    -- n=2 will overwrite the original value
             PORT MAP (inp, outp);
END my_arch;
```

Example : ALU

- 16 ALU operations

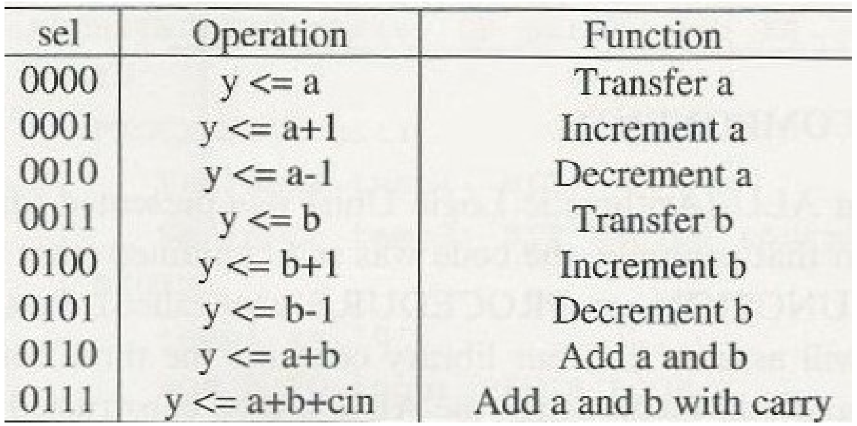




sel	Operation	Function	Unit
0000	y <= a	Transfer a	Arithmetic
0001	y <= a+1	Increment a	
0010	y <= a-1	Decrement a	
0011	y <= b	Transfer b	
0100	y <= b+1	Increment b	
0101	y <= b-1	Decrement b	
0110	y <= a+b	Add a and b	
0111	y <= a+b+cin	Add a and b with carry	
1000	y <= NOT a	Complement a	Logic
1001	y <= NOT b	Complement b	
1010	y <= a AND b	AND	
1011	y <= a OR b	OR	
1100	y <= a NAND b	NAND	
1101	y <= a NOR b	NOR	
1110	y <= a XOR b	XOR	
1111	y <= a XNOR b	XNOR	

Example : ALU (component "arith_unit")

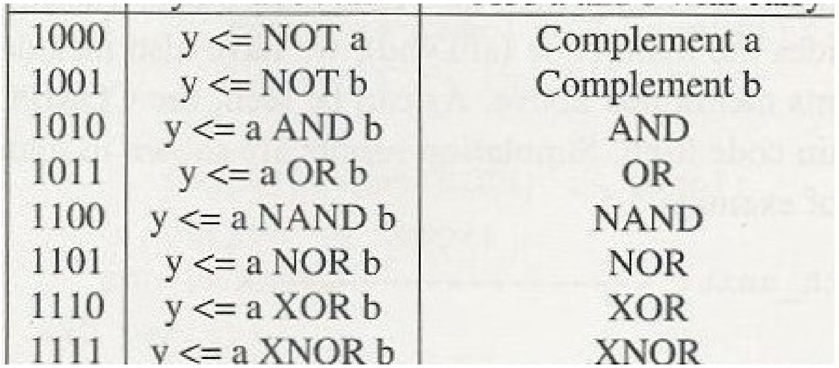
```
-- Part 1: COMPONENT arith_unit
LIBRARY ieee;   USE ieee.std_logic_1164.all;    USE ieee.std_logic_unsigned.all;
----------------------------------------
ENTITY arith_unit IS  PORT
           (a, b: IN     STD_LOGIC_VECTOR (7 DOWNTO 0);
              sel: IN     STD_LOGIC_VECTOR (2 DOWNTO 0);
              cin: IN     STD_LOGIC;
                x: OUT STD_LOGIC_VECTOR (7 DOWNTO 0)); END arith_unit;
----------------------------------------
ARCHITECTURE arith_unit OF arith_unit IS
           SIGNAL arith, logic: STD_LOGIC_VECTOR (7 DOWNTO 0); BEGIN
WITH sel SELECT
x <=       a          WHEN "000",
           a+1        WHEN "001",
           a-1        WHEN "010",
           b          WHEN "011",
           b+1        WHEN "100",
           b-1        WHEN "101",
           a+b        WHEN "110",
           a+b+cin WHEN OTHERS;
END arith_unit;
```



sel	Operation	Function
0000	y <= a	Transfer a
0001	y <= a+1	Increment a
0010	y <= a-1	Decrement a
0011	y <= b	Transfer b
0100	y <= b+1	Increment b
0101	y <= b-1	Decrement b
0110	y <= a+b	Add a and b
0111	y <= a+b+cin	Add a and b with carry

Example : ALU (component "logic_unit")

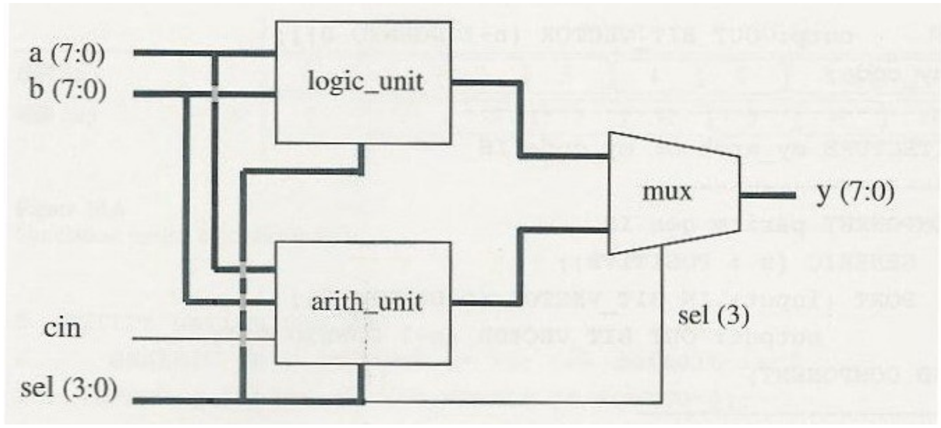
```
-- Part 2: COMPONENT logic_unit
LIBRARY ieee;   USE ieee.std_logic_1164.all;
-----------------------------------------
ENTITY logic_unit IS
          PORT (a, b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                    sel: IN STD_LOGIC_VECTOR (2 DOWNTO 0);
                    x: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END logic_unit;
-----------------------------------------
ARCHITECTURE logic_unit OF logic_unit IS
BEGIN
WITH sel SELECT
x <=      NOT a              WHEN "000",
          NOT b              WHEN "001",
          a AND b            WHEN "010",
          a OR b             WHEN "011",
          a NAND b           WHEN "100",
          a NOR b            WHEN "101",
          a XOR b            WHEN "110",
          NOT (a XOR b)  WHEN OTHERS;
END logic_unit;
```



1000	y <= NOT a	Complement a
1001	y <= NOT b	Complement b
1010	y <= a AND b	AND
1011	y <= a OR b	OR
1100	y <= a NAND b	NAND
1101	y <= a NOR b	NOR
1110	y <= a XOR b	XOR
1111	y <= a XNOR b	XNOR

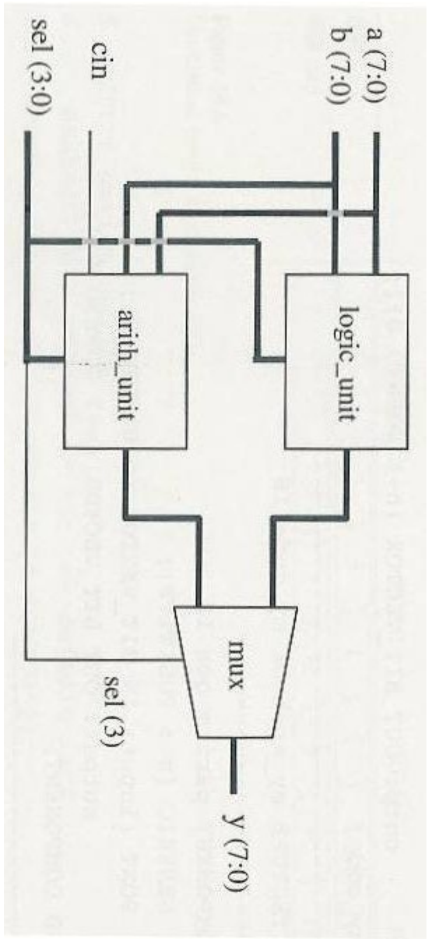
Example : ALU (component “mux”)

```
-- Part 3: COMPONENT mux
--------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------
ENTITY mux IS
        PORT ( a, b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                sel: IN STD_LOGIC;
                  x: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END mux;
-----------------------------------------
ARCHITECTURE mux OF mux IS
BEGIN
WITH sel SELECT
        x <=    a WHEN '0',
                b WHEN OTHERS;
END mux;
```



Example : ALU (main code)

```
LIBRARY ieee;   USE ieee.std_logic_1164.all;
----------------------------------------
ENTITY alu IS PORT
(a, b: IN STD_LOGIC_VECTOR(7 DOWNTO 0); cin: IN STD_LOGIC;
sel: IN STD_LOGIC_VECTOR(3 DOWNTO 0); y: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));    END alu;
----------------------------------------
ARCHITECTURE alu OF alu IS
----------------------
COMPONENT arith_unit IS PORT
(a, b: IN STD_LOGIC_VECTOR(7 DOWNTO 0); cin: IN STD_LOGIC;
sel: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
x: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));   END COMPONENT;
----------------------
COMPONENT logic_unit IS PORT
(a, b: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
sel: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
x: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));  END COMPONENT;
----------------------
COMPONENT mux IS PORT
(a, b: IN STD_LOGIC_VECTOR(7 DOWNTO 0); sel: IN STD_LOGIC;
x: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));  END COMPONENT;
----------------------
SIGNAL x1, x2: STD_LOGIC_VECTOR(7 DOWNTO 0);
----------------------
BEGIN
        U1: arith_unit PORT MAP (a, b, cin, sel(2 DOWNTO 0), x1);
        U2: logic_unit PORT MAP (a, b, sel(2 DOWNTO 0), x2);
        U3: mux        PORT MAP (x1, x2, sel(3), y);
END alu;
```



function and procedure

Verilog Tasks vs. Functions (1/2)

- task is similar to subroutine in FORTRAN
- function is similar to function in FORTRAN

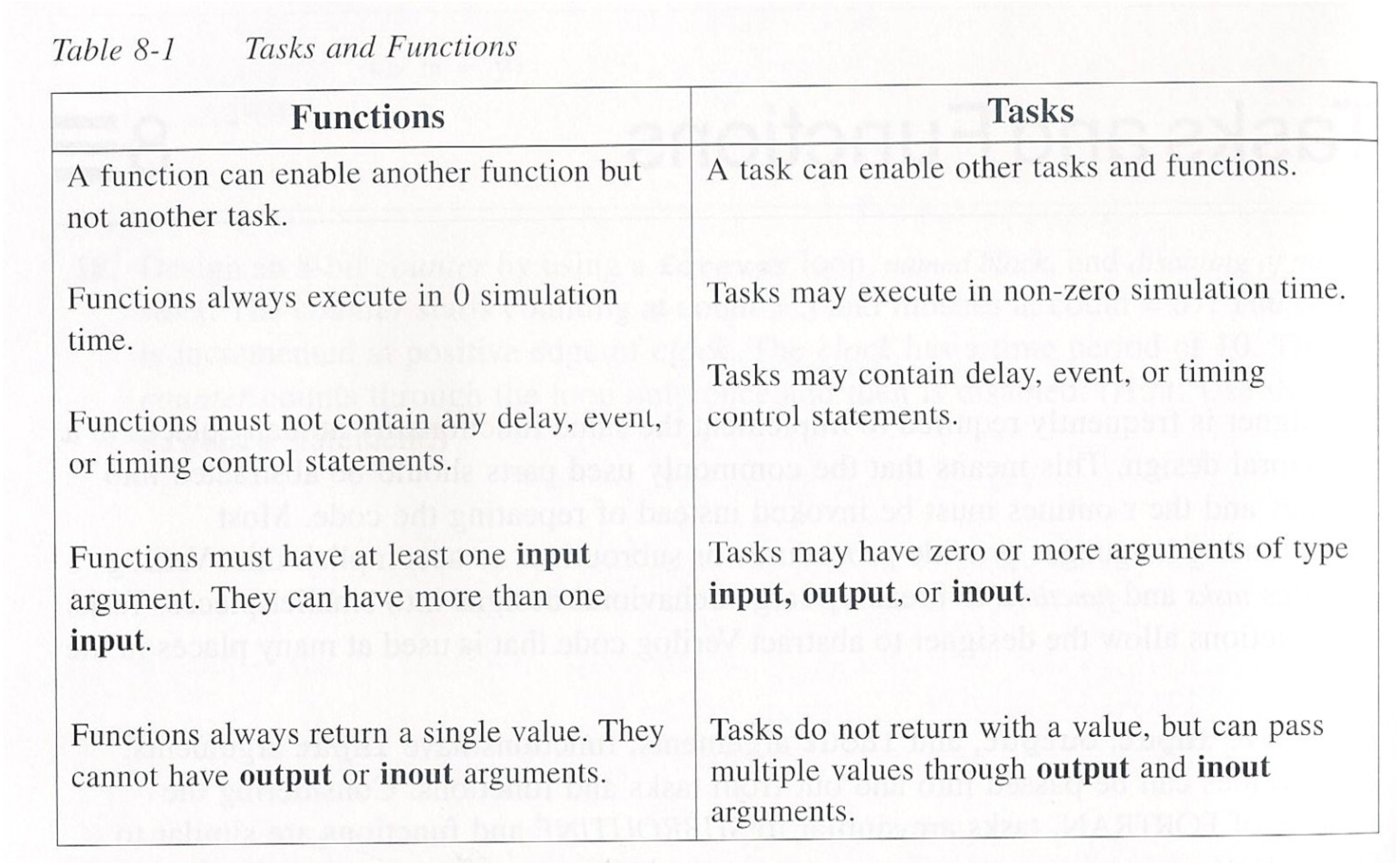


Table 8-1 Tasks and Functions

Functions	Tasks
A function can enable another function but not another task.	A task can enable other tasks and functions.
Functions always execute in 0 simulation time.	Tasks may execute in non-zero simulation time.
Functions must not contain any delay, event, or timing control statements.	Tasks may contain delay, event, or timing control statements.
Functions must have at least one **input** argument. They can have more than one **input**.	Tasks may have zero or more arguments of type **input, output**, or **inout**.
Functions always return a single value. They cannot have **output** or **inout** arguments.	Tasks do not return with a value, but can pass multiple values through **output** and **inout** arguments.

Verilog Tasks vs. Functions (2/2)

- defined in a module and local to the module
  - contain *behavioral statements only*
  - called from the **always** or **initial** blocks, or other tasks and functions
- tasks are used for commonly Verilog code
  - contain delays, timing, event constructs, or multiple output arguments
  - ***can have input, output and inout arguments***
- functions are used for code that
  - is purely **combinational**
  - executes in zero simulation time
  - provides ***exactly one output***
  - can have input arguments

Verilog Task Examples

```
module operation;
…
parameter delay=10;
reg [15:0] A, B;
reg [15:0] AB_AND, AB_OR, AB_XOR;
always @ (A or B)
begin
bitwise_oper(AB_AND, AB_OR, AB_XOR,
     A, B);
end
…
task bitwise_oper;
output [15:0] ab_and, ab_or, ab_xor;
input [15:0] a, b;
begin
   #delay ab_and = a & b;
   ab_or = a | b;
   ab_xor = a ^ b;
end
end task
…
endmodule
```

task could have
(1) multiple outputs
(2) delay/timing constructions

```
// another task declaration
//
task bitwise_oper (
output [15:0] ab_and, ab_or, ab_xor,
input [15:0] a, b);
begin
   #delay ab_and = a & b;
   ab_or = a | b;
   ab_xor = a ^ b;
end
endtask
```

Verilog Function Example

```
module parity;
reg [31:0] addr;
reg parity;
always @ (addr)
   parity = calc_parity(addr);

function calc_parity;
input [31:0] address;
begin
   calc_parity = ^address;
end
endfunction

endmodule
```

function could have
(1) only one returned output
(2) no delay/timing constructions
(3) purely combinational

```
// alternate function definition
function calc_parity
(input [31:0] address);
begin
   calc_parity = ^address;
end
endfunction
```

VHDL FUNCTION

- Create new functions such as
  - Data type conversions
  - Logical operations
  - Arithmetic computations
  - New operators and attributes
- Available statements
  - **IF**, **CASE**, **LOOP**, (WAIT is not allowed in function)
- **Prohibited**
  - **SIGNAL declaration, COMPONENT instantiation**

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

examples of VHDL function call

- x <= conv_integer (*a*);
  - – convert *a* to an integer
- y <= maximum (*a,b*);
  - – returns the largest of *a* and *b*
- IF x > maximum(a,b) THEN …
  - – function call associated to a statement

Example : positive_edge()

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

-------------------------- functional call ----------------
…
IF positive_edge (clk) THEN ….
…
```

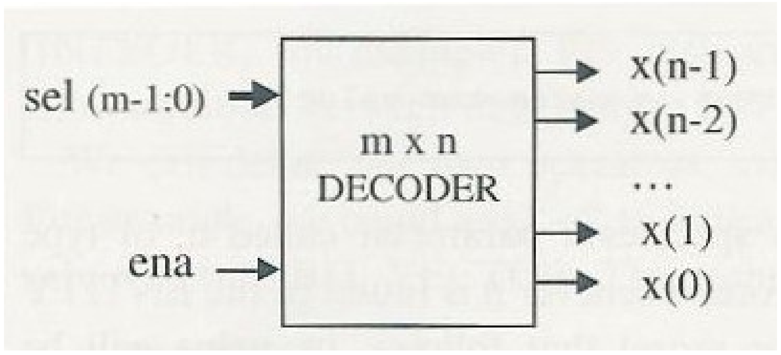
Example : conv_integer()

```
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);
…
```

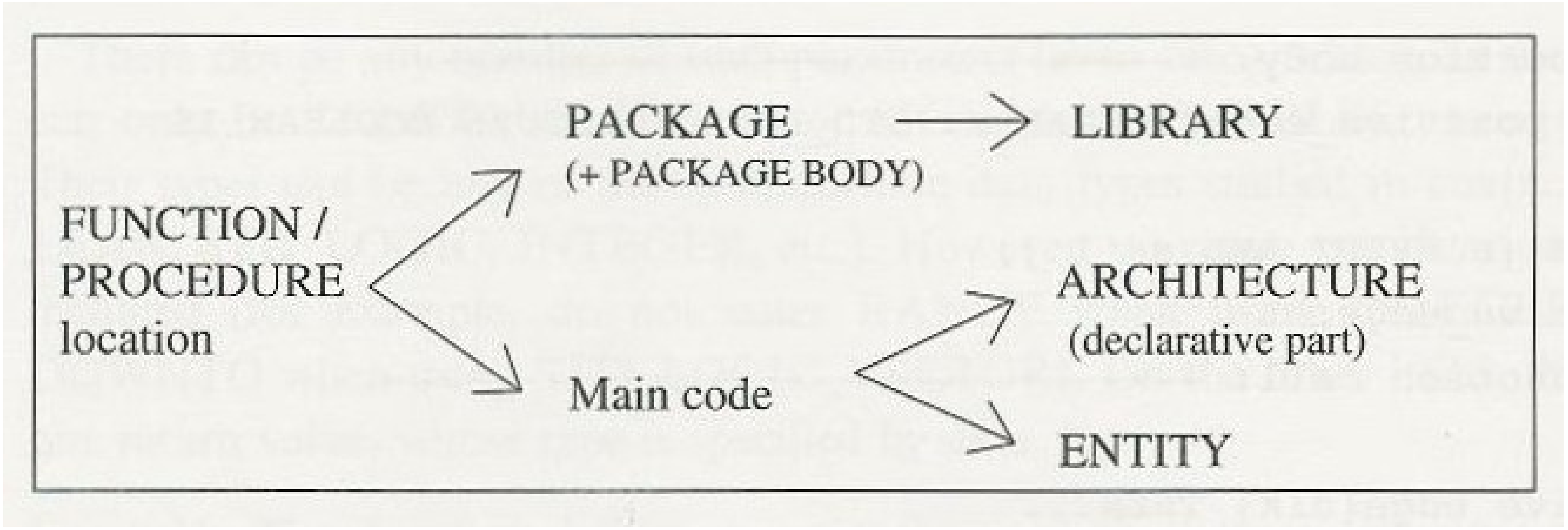
review of integer conversion in previous 3-to-8 decoder

```
ENTITY decoder IS
PORT (ena : IN      STD_LOGIC; sel : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
           x : OUT  STD_LOGIC_VECTOR (7 DOWNTO 0)  );      END decoder;
---------------------------------------------
ARCHITECTURE generic_decoder OF decoder IS BEGIN
PROCESS (ena, sel)
  VARIABLE temp1 : STD_LOGIC_VECTOR (x'HIGH DOWNTO 0);
  VARIABLE temp2 : INTEGER RANGE 0 TO x'HIGH;
BEGIN
temp1 := (OTHERS => '1');  -- initialize all output bits to be '1'
temp2 := 0;
IF (ena='1') THEN
           FOR i IN sel'RANGE LOOP   -- sel range is 2 downto 0
                     IF (sel(i)='1') THEN  -- binary-to-Integer conversion
                               temp2:=2*temp2+1;
                     ELSE
                               temp2 := 2*temp2;
                     END IF;
           END LOOP;
           temp1(temp2):='0'; -- active low
END IF;
x <= temp1; -- assign variable to signal
END PROCESS;   END generic_decoder;
```



function location

- in a package
  - **package body** is required
- in the main code


FUNCTION / PROCEDURE location
PACKAGE (+ PACKAGE BODY)
LIBRARY
Main code
ARCHITECTURE (declarative part)
ENTITY

Example : function in main code

```
LIBRARY ieee;  USE ieee.std_logic_1164.all;
---------------------------------------------
ENTITY dff IS PORT (d, clk, rst: IN STD_LOGIC; q: OUT STD_LOGIC); END dff;
---------------------------------------------

ARCHITECTURE my_arch OF dff IS
----------- function declared and called inside architecture -----------------------------
FUNCTION positive_edge(SIGNAL s: STD_LOGIC) RETURN BOOLEAN IS
BEGIN
        RETURN s'EVENT AND s='1';
END positive_edge;
------------------------------------------------------------------------------------------
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;
```

Example : function in package

```
-- Part 1: package
LIBRARY ieee; USE ieee.std_logic_1164.all;
--------------- function declared inside package -----------
PACKAGE my_package IS
              FUNCTION positive_edge(SIGNAL s: STD_LOGIC) RETURN BOOLEAN;
END my_package;
--------------- function defined inside package body ------
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;
 ------------------------------------------------------------------------------------------
-- Part 2: 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;
-------------------------- function called insided architecture -------------------
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;
```

Example : conv_integer (in package)

```
-- Part 1: package
LIBRARY ieee; USE ieee.std_logic_1164.all;
-----------------------------------------------
PACKAGE my_package IS
FUNCTION conv_integer (SIGNAL vector: STD_LOGIC_VECTOR) RETURN
INTEGER;
END my_package;
-----------------------------------------------
PACKAGE BODY my_package IS
      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;
END my_package;
```

Example : conv_integer (cont.)

```
-- Part 2: main code
---------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.my_package.all;
---------------------------------------------
ENTITY conv_int2 IS
        PORT (a: IN STD_LOGIC_VECTOR(0 TO 3);
                        y: OUT INTEGER RANGE 0 TO 15);
END conv_int2;
---------------------------------------------
ARCHITECTURE my_arch OF conv_int2 IS
BEGIN
        y <= conv_integer(a);
END my_arch;
```

Example : overloaded "+" operator with ripple carry (declared in package)

```
-- Part 1: package
----------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
----------------------------------------------
PACKAGE my_package IS
   FUNCTION "+" (a, b: STD_LOGIC_VECTOR) RETURN STD_LOGIC_VECTOR;
END my_package;
----------------------------------------------
PACKAGE BODY my_package IS
FUNCTION "+" (a, b: STD_LOGIC_VECTOR) RETURN STD_LOGIC_VECTOR IS
VARIABLE result: STD_LOGIC_VECTOR;
VARIABLE carry: STD_LOGIC;
BEGIN
carry := '0';
FOR i IN a'REVERSE_RANGE LOOP
   result(i) := a(i) XOR b(i) XOR carry;
   carry := (a(i) AND b(i)) OR (a(i) AND carry) OR (b(i) AND carry);
END LOOP;
RETURN result;
END "+";
END my_package;
```

Example : overloaded "+" operator (cont.)

```
-- Part 2: main code
---------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.my_package.all;
---------------------------------------------
ENTITY add_bit IS
        PORT (a: IN     STD_LOGIC_VECTOR(3 DOWNTO 0);
                 y: OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END add_bit;
---------------------------------------------
ARCHITECTURE my_arch OF add_bit IS
        CONSTANT b: STD_LOGIC_VECTOR(3 DOWNTO 0) := "0011";
        CONSTANT c: STD_LOGIC_VECTOR(3 DOWNTO 0) := "0110";
BEGIN
        y <= a + b + c;                         -- overloaded "+" operator
END my_arch;
```

Example : shift left with duplicated LSB (in main code)

```
ENTITY shift_left IS
GENERIC (size: INTEGER := 4);
PORT (      a: IN     STD_LOGIC_VECTOR(size-1 DOWNTO 0);
       x, y, z: OUT STD_LOGIC_VECTOR(size-1 DOWNTO 0));     END shift_left;
---------------------------------------------
ARCHITECTURE behavior OF shift_left IS
------------------------------------------
FUNCTION slar (arg1: STD_LOGIC_VECTOR (size-1 DOWNTO 0); arg2: NATURAL)
RETURN STD_LOGIC_VECTOR IS
VARIABLE input: STD_LOGIC_VECTOR(size-1 DOWNTO 0) := arg1;
CONSTANT size : INTEGER := arg1'LENGTH;
VARIABLE copy: STD_LOGIC_VECTOR(size-1 DOWNTO 0) := (OTHERS => arg1(arg1'RIGHT));
VARIABLE result: STD_LOGIC_VECTOR(size-1 DOWNTO 0);
BEGIN
          IF (arg2 >= size-1) THEN result := copy;
          ELSE result := input(size-1-arg2 DOWNTO 1) & copy(arg2 DOWNTO 0);
          END IF;
          RETURN result;
END slar;
------------------------------------------
BEGIN
          x <= slar (a, 0);
          y <= slar (a, 1);
          z <= slar (a, 2);
END behavior;
```

Example : fully CL bit-serial multiplier (in package)

```
-- Part 1: package
LIBRARY ieee; USE ieee.std_logic_1164.all; USE ieee.std_logic_arith.all;
---------------------------------------------
PACKAGE pack IS
              FUNCTION mult (a, b: UNSIGNED) RETURN UNSIGNED;
END pack;
---------------------------------------------
PACKAGE BODY pack IS
FUNCTION mult(a, b: UNSIGNED) RETURN UNSIGNED IS
CONSTANT length: INTEGER := a'LENGTH + b'LENGTH ;
VARIABLE aa: UNSIGNED(length-1 DOWNTO 0)
                            := (length-1 DOWNTO a'LENGTH => '0') & a(a'LENGTH-1 DOWNTO 0);
VARIABLE prod: UNSIGNED(length-1 DOWNTO 0) := (OTHERS => '0');
BEGI
FOR i IN 0 TO b'LENGTH-1 LOOP
              IF (b(i)='1') THEN prod := prod + aa; END IF;
              aa := aa(length-2 DOWNTO 0) & '0';
END LOOP;
RETURN prod;
END mult;
END pack;
```

Example : bi-srial multiplier (cont.)

```
-- Part 2: main code
---------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE work.pack.all;
---------------------------------------------
ENTITY multiplier IS
        GENERIC (size: INTEGER := 4);
        PORT (a, b: IN UNSIGNED(size-1 DOWNTO 0);
                y: OUT UNSIGNED(2*size-1 DOWNTO 0));
END multiplier;
---------------------------------------------
ARCHITECTURE behavior OF multiplier IS
BEGIN
        y <= mult (a,b);
END behavior;
```

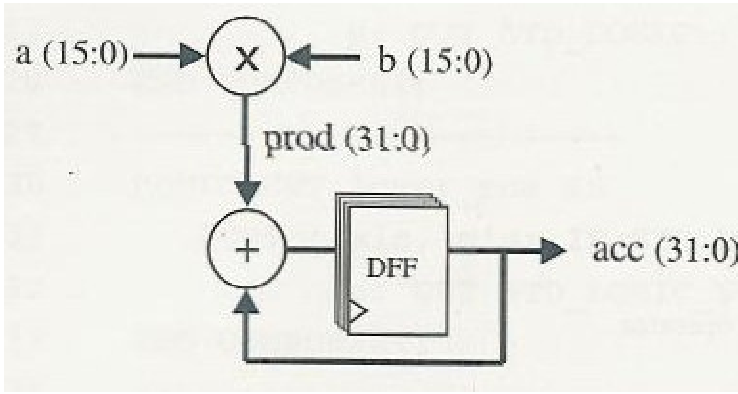
Quiz: sequential bit-serial multiplier

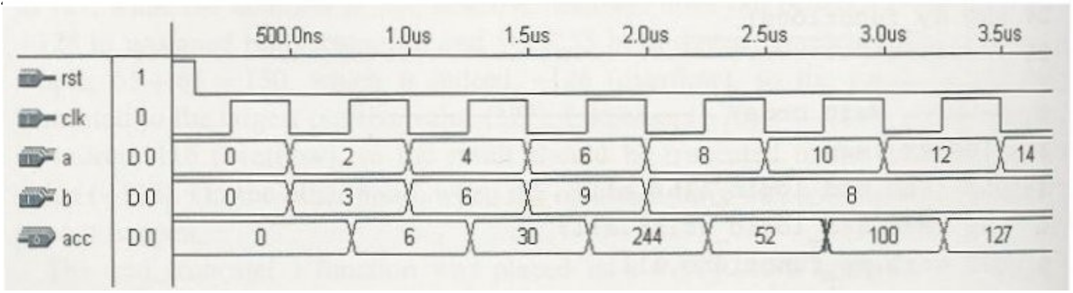
- How to implement a sequential bit-serial multiplier where each shift-add operation is executed in one cycle?
  - the previous example is a fully-combinational circuit where all the recursive shift-add operations are combined together
  - signal declaration and component instantiation are NOT allowed in function
  - function usually syntheizes combination logic
  - hint: use for loop inside process block

MAC (with sat. adder)

```
-- Part 1: my_functions (package)
----------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
----------------------------------------------------------
PACKAGE my_functions IS
    FUNCTION add_truncate (SIGNAL a, b: SIGNED; size: INTEGER)
               RETURN SIGNED;
END my_functions;
----------------------------------------------------------
PACKAGE BODY my_functions IS
    FUNCTION add_truncate (SIGNAL a, b: SIGNED; size: INTEGER)
                              RETURN SIGNED IS
               VARIABLE result: SIGNED (7 DOWNTO 0);
    BEGIN
               result := a + b;
               IF (a(a'left)=b(b'left)) AND (result(result'LEFT)/=a(a'left)) THEN
                              result := (result'LEFT => a(a'LEFT), OTHERS => NOT a(a'left));
               END IF;
               RETURN result;
    END add_truncate;
END my_functions;
----------------------------------------------------------

-- Part 2: main code
----------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE work.my_functions.all;
----------------------------------------------
ENTITY mac IS
    PORT (a, b: IN SIGNED(3 DOWNTO 0);
                              clk, rst: IN STD_LOGIC;
                              acc: OUT SIGNED(7 DOWNTO 0));
END mac;
----------------------------------------------
ARCHITECTURE rtl OF mac IS
    SIGNAL prod, reg: SIGNED(7 DOWNTO 0);
BEGIN
    PROCESS (rst, clk)
               VARIABLE sum: SIGNED(7 DOWNTO 0);
    BEGIN
               prod <= a * b;
               IF (rst='1') THEN
                              reg <= (OTHERS=>'0');
               ELSIF (clk'EVENT AND clk='1') THEN
                              sum := add_truncate (prod, reg, 8);
                              reg <= sum;
               END IF;
    acc <= reg;
    END PROCESS;
END rtl;
----------------------------------------------
```



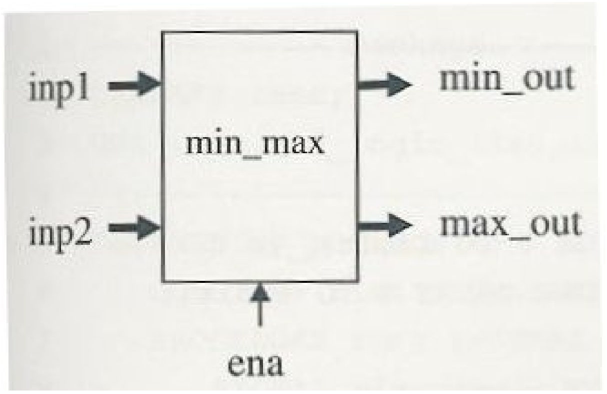


VHDL PROCEDURE

- Can return more than one value

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

Example : Procedure in main code



```
LIBRARY ieee;
USE ieee.std_logic_1164.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
-------------------------
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;
-------------------------
BEGIN
  PROCESS (ena)
  BEGIN
    IF (ena='1') THEN sort (inp1, inp2, min_out, max_out); END IF;
  END PROCESS;
END my_architecture;
```

Example: procedure in package

```
-- Part 1: 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;
```

Example: procedure in package (cont.)

```
-- Part 2: 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;
```

FUNCTION vs. PROCEDURE

- FUNCTION
  - Has zero or more input parameters
  - Input parameters can only be
    - CONSTANT (default)
    - SIGNAL (VARIABLES are not allowed)
  - return a single output
  - Called as part of an expression
- PROCEDURE
  - Has any number of IN, OUT, INOUT parameters
  - Parameters can be
    - SIGNAL
    - VARIABLE (default for mode OUT, or INOUT parameters)
    - CONSTANT (default for mode IN parameters)
- can be in PACKAGE or in the main code
  - PACKAGE BODY is required if they are placed in the PACKAGE

ASSERT

- Non-synthesizable, only for simulation

```
ASSERT condition
[REPORT “message”]
[SEVERITY severity_level];
```

- Severity levels:
  - Note, Warning, Error (default), Failure
- Message written when condition is FALSE

```
ASSERT a’LENGTH = b’LENGTH
REPORT “Error: vector do not have same length”
SEVERITY failure;
```

configuration

Verilog configuration

- top.v wants to use adder.v for instance a1 and adder.vg for instance a2

```
file top.v

module top();
adder a1(...);
adder a2(...);
endmodule
```

```
file adder.v

module adder(...);
// rtl adder
// description
...
endmodule
```

```
file adder.vg

module adder(...);
// gate-level adder
// description
...
endmodule
```

- use configuration to bind instances with implementations
  - **design** statement in config cfg1 specifies the top-level module in the design
  - **default** statement coupled with the **liblist** clause specifies , by default, all subinstances of top (i.e., top.a1, top.a2)
  - **instance** statement specifies, for the particular instance top.a2, the source description shall be from gateLib.

```
config cfg1; // specify rtl adder for top.a1, gate-level adder for top.a2
    design rtlLib.top;
    default liblist rtlLib;
    instance top.a2 liblist gateLib;
endconfig
```

```
library rtlLib *.v;          // matches all files in the current directory with a .v suffix
library gateLib ./*.vg;      // matches all files in the current directory with a .vg suffix
```

VHDL configuration

- multiple architectures might exist for a single entity
  - for example, one architecture might be behavioral model, while another might be structural model
- configuration specifies the architecture to be used (***bind*** a specific architecture to entity)
- **no need for single entity-architecture pair**
  - that is, no configuration is needed if only one architecture exists for an entity

Two Binding Methods

- Direct binding

```
CONFIGURATION config_name OF entity_name IS
  FOR arch_name
  END FOR;
END [CONFIGURATION] [entity_name]
```

- Binding in component instantiation

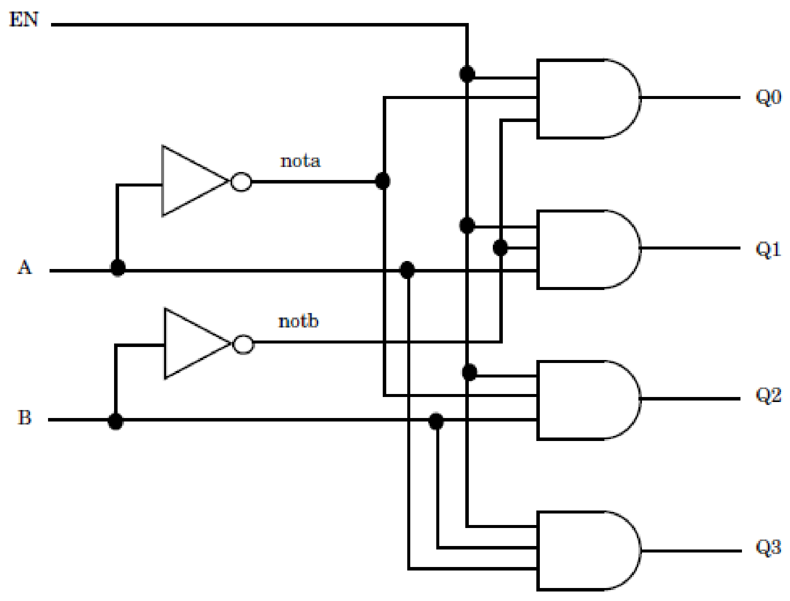
```
CONFIGURATION config_name OF entity_name IS
  FOR arch_name
      FOR label: component_name
--- or FOR OTHERS/ALL: component_name
        USE ENTITY entity_name [(arch_name)];
      END FOR;   -- end of component_name
  END FOR;  -- end of arch_name
END [CONFIGURATION] [entity_name]
```

configuration

```
entity counter is … end counter;
…
-- two different architectures for the same entity "counter"
architecture count_255 of counter is … end count_255;
architecture count_64k of counter is … end count_64k;
…
-- select the architecture "count_255" for the entity "counter"
configuration small_count of counter is
   for count_255     -- architecture for entity being configured
   end for;          -- or names of blocks of components
end small_count;
…
-- select the architecture "count_64k" for the entity "counter"
configuration big_count of counter is
   for count_64k
   end for;
end big_count;
```

component configuration (bind architecture to entity)

```
entity  inv is … end inv;
architecture behave of inv is … end behave;
configuration invcon of inv is     for behave  end for;     end invcon;
…
entity and3 is … end and3;
architecture behave of and3 is … end behave;
configureation and3con of and3 is     for behave end for;     end and3con;
…
entity decode is … end decode;
architecture structural of decode is
component inv … end component;
component and3 … end component;
begin
I1   : inv     port map (…);
I2   : inv     port map (…)
A1 : and3  port map (…);
A2 : and3  port map (…);
A3 : and3  port map (…);
A4 : and3  port map (…);
end structural;
```



using lower-level configuration

```
configuration decode_lower_level_config of decode is

for structural  -- for architecture “structural” of entity “decode”
   for  I1 : inv use configration work.invcon; end for;
-- use configuration invcon (defined in previous slide)
   for  I2 : inv use configuration work.invcon; end for;
   for all : and3 use configuration work.and3con; end for;
-- use configuration and3con (defined in previous slide)
end for;

end decode_lower_level_config;
```

using *entity-architecture* pair

```
configuration decode_entity_arch_pair of decode is

for structural
   for        I1 : inv    use entity work.inv(behave); end for;
-- use entity inv, and architecture behave, in work directory
   for others : inv    use entity work.inv(behave); end for;
   for        a1 : and3 use entity work.and3(behave); end for;
-- use entity and3, architecture behave, in work directory
   for others : and3 use entity work.and3(behave); end for;
end for;

end decode_entity_arch_pair;
```

block configuration

```
entity cpu is … end cpu;

architecture fragment of cpu is
        component int_reg … end component;
        component       alu … end component;
        …
        reg_array : block   -- block “reg_array”
        begin  R1 : int_reg port map (…);
                R2 : int_reg port map (…);
        end block reg_array;
        …
        shifter : block   -- block “shifter”
        begin A1 : alu port map (…);
               A2 : alu port map (…);
               shift_reg : block   -- block “shift_reg”
                 begin   R1: int_reg port map (…);  end block shift_reg;
        end block shifter;
        …
end fragment;

configuration cpu_con of cpu is
  for fragment   -- for architecture “fragment”
    for reg_array  -- for block “reg_array”
      for all: int_reg use configuration work.int_reg_con; end for;
    end for;
    for shifter  -- for block “shifter”
      for A1 : alu use configuration work.alu_con; end for;
      for shift_reg  -- for block “shift_reg”
        for R1 : int_reg use configuration work.int_reg_con; end for;
      end for;
    end for;
  end for;
end cpu_con;
```