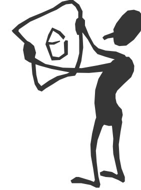
Chapter 8
Structural Modeling

VHDL

Outline

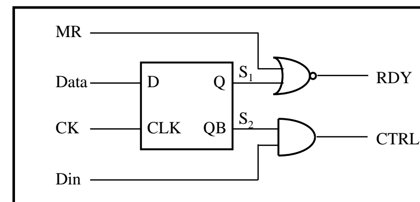
- Basic Example
- Component Declaration
- Component Instantiation
- Resolving Signal Value
- Generate Statement



Structural Modeling

- **This chapter describes the structural style of modeling. An entity is modeled as a set of components connected by signals, that is, as a netlist.**
- **The behavior of the entity is not explicitly apparent from its model. The component instantiation statement is the primary mechanism used for describing such a model of an entity.**
- **COMPONENT & PORT MAP statements are used to implement structural modeling.**
- **The component instantiation statements are concurrent statements, and their order of appearance in the architecture body is therefore not important.**
- **A component can, in general, be instantiated any number of times.**
- **Each instantiation must have a unique component label.**

Basic Example



- **Three components and2, nor2, and dff are used.**
- **The components are instantiated in the architecture body via three component instantiation statements - PORT MAP, and the instantiated components are connected to each other via signals S1 and S2.**

Basic Example (VHDL Code)

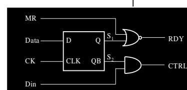
```
ENTITY Gating IS
  PORT (data, ck, mr, din: IN bit; rdy, ctrl: OUT bit);
END Gating;

ARCHITECTURE Structure_View OF Gating IS
  COMPONENT and2
    PORT (x, y: IN bit; z: OUT bit);
  END COMPONENT;

  COMPONENT nor2
    PORT (x, y: IN bit; z: OUT bit);
  END COMPONENT;

  COMPONENT dff
    PORT (d, clk: IN bit; q, qb: OUT bit);
  END COMPONENT;

  SIGNAL s1, s2: bit;
BEGIN
  d1: dff  PORT MAP (data, ck, s1, s2);
  a1: and2 PORT MAP (s2, din, ctrl);
  n1: nor2 PORT MAP (s1, mr, rdy);
END Structure_View ;
```



Component Declaration

- **A component in a structural description must first be declared using a component declaration.**
- **A component declaration declares the name and the interface of a component (similar to the entity).**
- **The interface specifies the mode and the type of ports.**
- **The syntax of a simple form of component declaration is:**

```
COMPONENT Component-Name [IS]
   [PORT(List-of-Interface-Ports);]
END COMPONENT [Component-Name ];
```

Component Declaration (notes)

- The ***component-name*** may or may not refer to the name of an entity already existing in a library. If it does not, it must be explicitly bound to an entity.
- The binding information can be specified using a configuration.
- The ***List-Of-Interface-Ports*** specifies the name, mode, and type for each port of the component in a manner similar to that specified in an entity declaration.
- The names of the ports may also be different from the names of the ports in the entity to which it may be bound (different port names can be mapped in a configuration). For while, we will assume that an entity of the same name as that of the component already exists and that the name, mode, and type of each port matches the corresponding ones in the component.
- Configurations are discussed in the next chapter.

Component & Package

- Component declarations appear in the declarations part of an architecture body.
- Alternately, they may also appear in a package declaration. Items declared in this package can then be made visible within any architecture body by using the library and use clauses.
- For example, consider the entity **GATING** described in the **basic example**. A package such as shown may be created to hold the component declarations.

```
PACKAGE MyCOMP IS
  COMPONENT and2
    PORT (x, y: IN bit; z: OUT bit);
  END COMPONENT;
  COMPONENT nor2
    PORT (x, y: IN bit; z: OUT bit);
  END COMPONENT;
  COMPONENT dff
    PORT (d, clk: IN bit; q, qb: OUT bit);
  END COMPONENT;
END MyCOMP;
```


Component & Package Library

- If the package **MyPackage** has been compiled into library **MyLib**, the architecture body can be as:

```
LIBRARY MyLib;
USE MyLib.MyPackage.All;

ARCHITECTURE Structure_View OF Gating IS

  SIGNAL s1, s2: bit;
BEGIN
  d1: dff  PORT MAP (data, ck, s1, s2);
  a1: and2 PORT MAP (s2, din, ctrl);
  n1: nor2 PORT MAP (s1, mr, rdy);
END Structure_View ;
```

- More on Library and Package in the next chapters.

Component Instantiation

- A component instantiation statement defines a sub-component of the entity in which it appears. It associates the signals in the entity with the ports of that sub-component.
- A format of a component instantiation statement**:**

```
Component-Label: Component-Name PORT MAP (association-list);
```

- The ***Component-Label*** can be any legal identifier and can be considered as the name of the instance.
- The ***Component-Name*** must be the name of a component declared earlier using a component declaration.
- The ***association-list,*** associates signals in the entity, called **actuals**, with the ports of a component, called **formals**.

Actuals and Formals

- An actual may be a signal. An actual for an input port may also be an expression.
- An actual may also be the keyword **open** to indicate a port that is not connected**.**
- There are two ways to perform the association of formals with actuals:
  1. **Positional association**
  2. **Named association**
- In positional association, each actual in the component instantiation is mapped by position with each port in the component declaration. That is, the first port in the component declaration corresponds to the first actual in the component instantiation, the second with the second, and so on.

```
  COMPONENT dff
    PORT (d, clk: IN bit; q, qb: OUT bit);
  END COMPONENT;
-----------------------------------------
  d1: dff  PORT MAP (data, ck, s1, s2);
```


Actuals and Formals

- If a port in a component instantiation is not connected to any signal, the keyword **OPEN** can be used to signify that the port is not connected.
- For example:

```
  COMPONENT dff
    PORT (d, clk: IN bit:='0'; q, qb: OUT bit);
  END COMPONENT;
-----------------------------------------
  d1: dff  PORT MAP (OPEN, ck, s1, OPEN);
```

- The second input port of the **dff** component is not connected to any signal. An **input** port may be left open only if its declaration specifies an initial value. For the previous component instantiation statement to be legal, port **d** of the component declaration for **dff** must have an initial value expression, while the **output** port **qb** not.
- A port of any other mode may be left unconnected as long as it is not an unconstrained array.

Actuals and Formals

- In named association, an association-list is of the form:
  - $formal_1$ => $actual_1$, $formal_2$ => $actual_2$, … $formal_n$ => $actual_n$
- For example:

```
  COMPONENT dff
    PORT (d, clk: IN bit; q, qb: OUT bit);
  END COMPONENT;
----------------------------------------
  d1: dff  PORT MAP (clk => ck, d => data, qb => s2, q => s1);
```

- In named association, the ordering of the associations is not important since the mapping between the actuals and formals is explicitly specified.
- An important point to note is that the scope of the formals is restricted to be within the port map part of the instantiation for that component.

Actuals / Formals Type and Mode

- The types of the formal and actual being associated must be the same.
- The modes of the ports must conform to the rule that if the formal is readable, so must the actual be; and if the formal is writable, so must the actual be.
- Locally declared signal is considered to be both readable and writable, such a signal may be associated with a formal of any mode.
- If an actual is a port of mode **in**, it may not be associated with a formal of mode **out** or **inout**; if the actual is a port of mode **out**, it may not be associated with a formal of mode **in** or **inout**; if the actual is a port of mode **inout**, it may be associated with a formal of mode **in**, **out**, or **inout**.
- It is important to note that an actual of mode **out** or **inout** indicates the presence of a source for that signal, and therefore, it must be resolved if that signal is multiply driven.
- A **buffer** port can never have more than one source; therefore, the only kind of actual that can be associated with a **buffer** port is another **buffer** port or a signal that has at most one source.

Component Model

- Structural models can be simulated and synthesize only after the entities that the components represent are modeled and placed in a design library.
- The lowest-level entities must be behavioral models (or dataflow).
- Consider the components instantiation A1, N1, and D1 in the **basic example**. Assume that those instance is bound to an entity with the same name and identical port names.
- The library must include the model of those components.
- More on Library and Package in the next chapters.

Component Example (Package)

```
 1 library IEEE;
 2 use IEEE.STD_LOGIC_1164.all;
 3 use IEEE.NUMERIC_STD.all;
 4
 5 package MyTimer is
 6 component TimerEnt
 7      generic(TLEN:integer := 16);
 8      port(
 9          Clk        : in  STD_LOGIC;
10          Rst        : in  STD_LOGIC;
11          Enable     : in  STD_LOGIC;
12          TimerOut   : buffer UNSIGNED(TLEN-1 downto 0)
13          );
14 end component;
15 end MyTimer;
16 ------------------------------------------------------------------
```


Component Example (Entity & Arc)

```
17 library IEEE;
18 use IEEE.STD_LOGIC_1164.all;
19 use IEEE.NUMERIC_STD.all;
20
21 entity TimerEnt is
22      generic(TLEN:integer := 16);
23      port(
24          Clk        : in  STD_LOGIC;
25          Rst        : in  STD_LOGIC;
26          Enable     : in  STD_LOGIC;
27          TimerOut   : buffer UNSIGNED(TLEN-1 downto 0)
28          );
29 end TimerEnt;
30
31
32 architecture TimerArc of TimerEnt is
33 ------------------------------------------------------------------
34 begin
35   TimerProcess: process(Clk, Rst)
36   variable vCnt : UNSIGNED(TLEN-1 downto 0);
37     begin
38       if Rst = '1' then
39         TimerOut <= (OTHERS => '0');
40       elsif (clk'event and clk = '1') then
41           if Enable = '1' then
42              TimerOut <= TimerOut+1;
43           end if;
44       end if;
45     end process;
46 ------------------------------------------------------------------
47 end TimerArc;
```

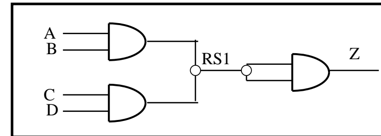

Component Example (Main)

```
52
53 library IEEE;
54 use IEEE.STD_LOGIC_1164.all;
55 use IEEE.NUMERIC_STD.all;
56 use WORK.MyTimer.all;
57
58 entity MyMain is
59      port(
60          XClk       : in  STD_LOGIC;
61          XRst       : in  STD_LOGIC;
62          XEnable    : in  STD_LOGIC;
63          XOutA      : buffer UNSIGNED(7 downto 0);
64          XOutB      : buffer UNSIGNED(15 downto 0)
65          );
66 end MyMain;
67
68 architecture MyMainArc of MyMain is
69 ------------------------------------------------------------------
70 begin
71 Timer1: TimerEnt
72         generic map (8)
73         port map (XClk, XRst, XEnable, XOutA);
74
75 Timer2: TimerEnt
76         generic map (16)
77         port map (XClk, XRst, XEnable, XOutB);
78
79 end MyMainArc;
```

Component Example (Simulation)

Resolving Signal Value

- If outputs of two components drive a common signal, the value of the signal must be **resolved** using a **resolution function**. This is similar to the case of a signal being assigned using more than one concurrent signal assignment statement.
- For example, consider the circuit shown below, which shows two and gates driving a common signal RS1, which is drive to produce the result in Z.
- The RS1 signal must be of resolved type (std_logic for example).



Generate Statement

- **Generate statement is concurrent.**
- **Concurrent statements can be conditionally selected or replicated during the elaboration phase using the generate statement.**
- **There are two forms of the generate statement:**
  1. **for-generation - concurrent statements replicated a predetermined number of times.**
  2. **if-generation - concurrent statements conditionally elaborated.**
- **The generate statement is interpreted during elaboration, and therefore has no simulation semantics. It resembles a macro expansion.**
- **The generate statement provides for a compact description of regular structures such as memories, registers, and counters.**

For Generate Scheme

- The format of a generate statement using the **for-generation** scheme is:

```
Generate-Label: FOR index IN range GENERATE
  [block-declarations
BEGIN]
  concurrent-statements;
END GENERATE [Generate-Label];
```

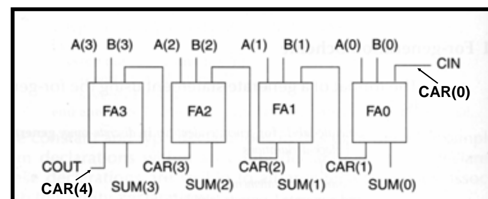
- The values in the discrete range must be globally static, that is, they must be computable at elaboration time.
- During elaboration, the set of concurrent statements are replicated once for each value in the discrete range.
- The statements can use the generate index in their expressions, and its value would be substituted during elaboration for each replication.

For Generate Scheme (notes)

- There is an implicit declaration for the generate index within the generate statement.
- The type of the identifier is defined by the discrete range.
- Declarations, if present, declare items that are visible only within the generate statement.
- The body of the generate statement can also include other concurrent statement (as we will see later).
- For-Generate statements may be nested.

For Generate (example)

- **Consider the following representation of a 4 - bit full-adder:**



For - Generate (example code)

```
ENTITY full_add4 IS
  PORT (a, b: IN bit_vector(3 DOWNTO 0); cin: IN bit;
        sum: OUT bit_vector(3 DOWNTO 0); cout: OUT bit;
END full_add4;

ARCHITECTURE for_generate OF full_add4 IS
  COMPONENT fa
    PORT (pa, pb, pc: IN bit; pcout, psum: OUT bit);
  END COMPONENT;

  SIGNAL car: bit_vector(4 DOWNTO 0);
BEGIN
  car(0) <= cin;
  gk: FOR k IN 3 DOWNTO 0 GENERATE
        f1: fa PORT MAP(car(k),a(k),b(k),car(k+1),sum(k));
      END GENERATE gk;
  cout <= car(4);
END for_generate;
```


For - Generate (without Port-Map)

- The body of the generate statement can also have other concurrent statement.
- For example, in the previous architecture, the component instantiation statement could be replace by concurrent signal assignment:

```
  SIGNAL car: bit_vector(4 DOWNTO 0);
BEGIN
  car(0) <= cin;
  gk: FOR k IN 3 DOWNTO 0 GENERATE
        sum(k) <= a(k) XOR b(k) XOR car(k);
        car(k+1) <= a(k) AND b(k) AND car(k);
      END GENERATE gk;
  cout <= car(4);
END for_generate;
```


If Generate Scheme

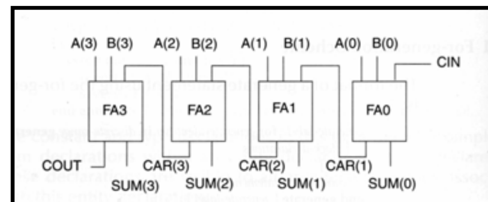
- The format of a generate statement using the **if-generation** scheme is:

```
Generate-Label: IF expression GENERATE
  [block-declarations
BEGIN]
  concurrent-statements;
END GENERATE [Generate-Label];
```

- The if-generate statement allows for conditional selection of concurrent statements based on the value of an expression.
- The expression must be a globally static expression, that is, the value must be computable at elaboration time.
- Any declarations present are local to the generate statement.
- There is no **else** or **elsif** branch.

If Generate (example)

- **Consider the following representation of a 4 - bit full-adder:**



- **This time we use the port directly.**

If - Generate (example code)

```
ARCHITECTURE if_generate OF full_add4 IS
  COMPONENT fa
    PORT (pa, pb, pc: IN bit; pcout, psum: OUT bit);
  END COMPONENT;
SIGNAL car: bit_vector(3 DOWNTO 1);
BEGIN
  gk: FOR k IN 3 DOWNTO 0 GENERATE
        ck0: IF k = 0 GENERATE
          f1: fa PORT MAP(cin,a(k),b(k),car(k+1),sum(k));
        END GENERATE ck0;
        ck1: IF k > 0 AND k < 3 GENERATE
          f1: fa PORT MAP(car(k),a(k),b(k),car(k+1),sum(k));
        END GENERATE ck1;
        ck3: IF k = 3 GENERATE
          f1: fa PORT MAP(car(k),a(k),b(k),cout,sum(k));
        END GENERATE ck3;
     END GENERATE gk;
END if_generate;
```