数字系统设计

郑海永

中国海洋大学 电子工程系

2014 年 5 月

VHDL

1. 硬件描述语言
2. VHDL 基础
3. VHDL 实例
4. VHDL 入门
5. VHDL 基本语句
6. VHDL 深入

目录

1 顺序语句与并行语句
- 并行语句

内容提要

1 顺序语句与并行语句
- 并行语句

内容提要

1 顺序语句与并行语句
- 并行语句

并行语句

结构体中的语句都是并行语句

- 进程语句
- 块语句
- 并行信号赋值语句
- 并行过程调用语句
- 并行断言语句
- 元件例化语句
- 生成语句

并行语句

结构体中的语句都是并行语句

- 进程语句
- 块语句
- 并行信号赋值语句
- 并行过程调用语句
- 并行断言语句
- 元件例化语句
- 生成语句

① 进程语句

- 进程语句本身是一条并行语句。
- 它可以和其他并行语句一样出现在结构体中。
- 各条进程语句之间也是并行关系，等同于各个电路模块之间是并行工作的。

- 进程内部的各条语句之间是顺序关系。
- 进程内部的所有语句都是用于行为描述的顺序语句。

① 进程语句

- 进程语句本身是一条并行语句。
- 它可以和其他并行语句一样出现在结构体中。
- 各条进程语句之间也是并行关系，等同于各个电路模块之间是并行工作的。

- 进程内部的各条语句之间是顺序关系。
- 进程内部的所有语句都是用于行为描述的顺序语句。

进程状态

- 活动状态 ⇒ 执行。
- 挂起状态 ⇒ 等待事件的发生。

1. 进程永远不会停止，它总是处于执行或挂起状态。
2. 进程在仿真开始（仿真的初始化阶段）进入执行状态，直到它遇到 WAIT 语句或敏感列表的信号没有变化而被挂起。
3. 如果进程没有敏感列表或 WAIT 语句，该进程将陷入无限循环，仿真器永远不会跳出初始化阶段。
4. 敏感列表（隐式的 WAIT 语句）和显式的 WAIT 语句不能同时出现在一个进程中。

进程状态

- 活动状态 ⇒ 执行。
- 挂起状态 ⇒ 等待事件的发生。

1. 进程永远不会停止，它总是处于执行或挂起状态。
2. 进程在仿真开始（仿真的初始化阶段）进入执行状态，直到它遇到 WAIT 语句或敏感列表的信号没有变化而被挂起。
3. 如果进程没有敏感列表或 WAIT 语句，该进程将陷入无限循环，仿真器永远不会跳出初始化阶段。
4. 敏感列表（隐式的 WAIT 语句）和显式的 WAIT 语句不能同时出现在一个进程中。

进程状态

- 活动状态 ⇒ 执行。
- 挂起状态 ⇒ 等待事件的发生。

1. 进程永远不会停止，它总是处于执行或挂起状态。
2. 进程在仿真开始（仿真的初始化阶段）进入执行状态，直到它遇到 WAIT 语句或敏感列表的信号没有变化而被挂起。
3. 如果进程没有敏感列表或 WAIT 语句，该进程将陷入无限循环，仿真器永远不会跳出初始化阶段。
4. 敏感列表（隐式的 WAIT 语句）和显式的 WAIT 语句不能同时出现在一个进程中。

进程状态

- 活动状态 ⇒ 执行。
- 挂起状态 ⇒ 等待事件的发生。

1. 进程永远不会停止，它总是处于执行或挂起状态。
2. 进程在仿真开始（仿真的初始化阶段）进入执行状态，直到它遇到 WAIT 语句或敏感列表的信号没有变化而被挂起。
3. 如果进程没有敏感列表或 WAIT 语句，该进程将陷入无限循环，仿真器永远不会跳出初始化阶段。
4. 敏感列表（隐式的 WAIT 语句）和显式的 WAIT 语句不能同时出现在一个进程中。

进程状态

- 活动状态 ⇒ 执行。
- 挂起状态 ⇒ 等待事件的发生。

1. 进程永远不会停止，它总是处于执行或挂起状态。
2. 进程在仿真开始（仿真的初始化阶段）进入执行状态，直到它遇到 WAIT 语句或敏感列表的信号没有变化而被挂起。
3. 如果进程没有敏感列表或 WAIT 语句，该进程将陷入无限循环，仿真器永远不会跳出初始化阶段。
4. 敏感列表（隐式的 WAIT 语句）和显式的 WAIT 语句不能同时出现在一个进程中。

进程语句格式

```
[ 进程标号: ] [ POSTPONED ] PROCESS [( 敏感信号表 )] IS
    [ { 声明语句 } ]
BEGIN
    [ { 顺序语句 } ]
END [ POSTPONED ] PROCESS [ 进程标号 ];
```

1. 进程标号是该进程的一个名称标识符，可选项。
2. 敏感信号表也是可选项。
   - 敏感信号表中的一个或多个信号值的变化可以激活该进程。
   - 敏感信号表等价于在该进程的最后一条顺序语句是一条隐含的 WAIT 语句。
   - 在含有敏感信号表的进程中，不能再有显式的 WAIT 语句。
3. 声明语句所声明的数据类型、对象和子程序等都是该进程的局部数据环境，不能在进程中声明信号对象。
4. 顺序语句描述了该进程的行为。
5. 延缓进程（VHDL'93）只在一个仿真周期的最后一个 Δ 时间才被激活，可以有效的降低进程的仿真处理频率。

进程语句格式

```
[ 进程标号: ] [ POSTPONED ] PROCESS [( 敏感信号表 )] IS
    [ { 声明语句 } ]
BEGIN
    [ { 顺序语句 } ]
END [ POSTPONED ] PROCESS [ 进程标号 ];
```

1. 进程标号是该进程的一个名称标识符，可选项。
2. 敏感信号表也是可选项。
   - 敏感信号表中的一个或多个信号值的变化可以激活该进程。
   - 敏感信号表等价于在该进程的最后一条顺序语句是一条隐含的 WAIT 语句。
   - 在含有敏感信号表的进程中，不能再有显式的 WAIT 语句。
3. 声明语句所声明的数据类型、对象和子程序等都是该进程的局部数据环境，不能在进程中声明信号对象。
4. 顺序语句描述了该进程的行为。
5. 延缓进程（VHDL'93）只在一个仿真周期的最后一个 Δ 时间才被激活，可以有效的降低进程的仿真处理频率。

进程语句格式

```
[ 进程标号: ] [ POSTPONED ] PROCESS [( 敏感信号表 )] IS
    [ { 声明语句 } ]
BEGIN
    [ { 顺序语句 } ]
END [ POSTPONED ] PROCESS [ 进程标号 ];
```

1. 进程标号是该进程的一个名称标识符，可选项。
   - 虽然进程标号是一个可选项，但是最好还是给每一个进程加一个标号。
   - 进程标号使得调试一个仿真系统更为容易，因为仿真器通过进程的标号提供了一种识别进程的方法。
   - 如果在进程语句中忽略进程标号，那么大多数仿真器也会给进程产生一个默认的名字。
2. 敏感信号表也是可选项。
   - 敏感信号表中的一个或多个信号值的变化可以激活该进程。
   - 敏感信号表等价于在该进程的最后一条顺序语句是一条隐含的 WAIT 语句。
   - 在含有敏感信号表的进程中，不能再有显式的 WAIT 语句。
3. 声明语句所声明的数据类型、对象和子程序等都是该进程的局部数

进程语句格式

```
[ 进程标号: ] [ POSTPONED ] PROCESS [( 敏感信号表 )] IS
    [ { 声明语句 } ]
BEGIN
    [ { 顺序语句 } ]
END [ POSTPONED ] PROCESS [ 进程标号 ];
```

1. 进程标号是该进程的一个名称标识符，可选项。
   - 虽然进程标号是一个可选项，但是最好还是给每一个进程加一个标号。
   - 进程标号使得调试一个仿真系统更为容易，因为仿真器通过进程的标号提供了一种识别进程的方法。
   - 如果在进程语句中忽略进程标号，那么大多数仿真器也会给进程产生一个默认的名字。
2. 敏感信号表也是可选项。
   - 敏感信号表中的一个或多个信号值的变化可以激活该进程。
   - 敏感信号表等价于在该进程的最后一条顺序语句是一条隐含的 WAIT 语句。
   - 在含有敏感信号表的进程中，不能再有显式的 WAIT 语句。
3. 声明语句所声明的数据类型、对象和子程序等都是该进程的局部数

进程语句格式

```
[ 进程标号: ] [ POSTPONED ] PROCESS [( 敏感信号表 )] IS
    [ { 声明语句 } ]
BEGIN
    [ { 顺序语句 } ]
END [ POSTPONED ] PROCESS [ 进程标号 ];
```

1. 进程标号是该进程的一个名称标识符，可选项。
   - 虽然进程标号是一个可选项，但是最好还是给每一个进程加一个标号。
   - 进程标号使得调试一个仿真系统更为容易，因为仿真器通过进程的标号提供了一种识别进程的方法。
   - 如果在进程语句中忽略进程标号，那么大多数仿真器也会给进程产生一个默认的名字。
2. 敏感信号表也是可选项。
   - 敏感信号表中的一个或多个信号值的变化可以激活该进程。
   - 敏感信号表等价于在该进程的最后一条顺序语句是一条隐含的 WAIT 语句。
   - 在含有敏感信号表的进程中，不能再有显式的 WAIT 语句。
3. 声明语句所声明的数据类型、对象和子程序等都是该进程的局部数

进程语句格式

```
[ 进程标号: ] [ POSTPONED ] PROCESS [( 敏感信号表 )] IS
    [ { 声明语句 } ]
BEGIN
    [ { 顺序语句 } ]
END [ POSTPONED ] PROCESS [ 进程标号 ];
```

1. 进程标号是该进程的一个名称标识符，可选项。
2. 敏感信号表也是可选项。
   - 敏感信号表中的一个或多个信号值的变化可以激活该进程。
   - 敏感信号表等价于在该进程的最后一条顺序语句是一条隐含的 WAIT 语句。
   - 在含有敏感信号表的进程中，不能再有显式的 WAIT 语句。
3. 声明语句所声明的数据类型、对象和子程序等都是该进程的局部数据环境，不能在进程中声明信号对象。
4. 顺序语句描述了该进程的行为。
5. 延缓进程（VHDL'93）只在一个仿真周期的最后一个 Δ 时间才被激活，可以有效的降低进程的仿真处理频率。

进程语句格式

```
[ 进程标号: ] [ POSTPONED ] PROCESS [( 敏感信号表 )] IS
    [ { 声明语句 } ]
BEGIN
    [ { 顺序语句 } ]
END [ POSTPONED ] PROCESS [ 进程标号 ];
```

1. 进程标号是该进程的一个名称标识符，可选项。
2. 敏感信号表也是可选项。
   - 敏感信号表中的一个或多个信号值的变化可以激活该进程。
   - 敏感信号表等价于在该进程的最后一条顺序语句是一条隐含的 WAIT 语句。
   - 在含有敏感信号表的进程中，不能再有显式的 WAIT 语句。
3. 声明语句所声明的数据类型、对象和子程序等都是该进程的局部数据环境，不能在进程中声明信号对象。
4. 顺序语句描述了该进程的行为。
5. 延缓进程（VHDL'93）只在一个仿真周期的最后一个 Δ 时间才被激活，可以有效的降低进程的仿真处理频率。

进程语句格式

```
[ 进程标号: ] [ POSTPONED ] PROCESS [( 敏感信号表 )] IS
    [ { 声明语句 } ]
BEGIN
    [ { 顺序语句 } ]
END [ POSTPONED ] PROCESS [ 进程标号 ];
```

1. 进程标号是该进程的一个名称标识符，可选项。
2. 敏感信号表也是可选项。
   - 敏感信号表中的一个或多个信号值的变化可以激活该进程。
   - 敏感信号表等价于在该进程的最后一条顺序语句是一条隐含的 WAIT 语句。
   - 在含有敏感信号表的进程中，不能再有显式的 WAIT 语句。
3. 声明语句所声明的数据类型、对象和子程序等都是该进程的局部数据环境，不能在进程中声明信号对象。
4. 顺序语句描述了该进程的行为。
5. 延缓进程（VHDL'93）只在一个仿真周期的最后一个 Δ 时间才被激活，可以有效的降低进程的仿真处理频率。

进程语句格式

```
[ 进程标号: ] [ POSTPONED ] PROCESS [( 敏感信号表 )] IS
    [ { 声明语句 } ]
BEGIN
    [ { 顺序语句 } ]
END [ POSTPONED ] PROCESS [ 进程标号 ];
```

1. 进程标号是该进程的一个名称标识符，可选项。
2. 敏感信号表也是可选项。
   - 敏感信号表中的一个或多个信号值的变化可以激活该进程。
   - 敏感信号表等价于在该进程的最后一条顺序语句是一条隐含的 WAIT 语句。
   - 在含有敏感信号表的进程中，不能再有显式的 WAIT 语句。
3. 声明语句所声明的数据类型、对象和子程序等都是该进程的局部数据环境，不能在进程中声明信号对象。
4. 顺序语句描述了该进程的行为。
5. 延缓进程（VHDL'93）只在一个仿真周期的最后一个 Δ 时间才被激活，可以有效的降低进程的仿真处理频率。

D 型触发器——bhv1

```
1  ENTITY dff_1 IS
2    PORT (clk, d: IN Bit;
3          q: OUT Bit);
4  END dff_1;

6  ARCHITECTURE bhv1 OF dff_1 IS
7  BEGIN
8    d1: PROCESS(clk)
9    VARIABLE a: Bit;
10   BEGIN
11     IF clk'Event AND clk='1' THEN
12       a := d;
13       q <= a;
14     END IF;
15   END PROCESS d1;
16 END bhv1;
```

敏感信号表

注意 每个进程的敏感信号表必须完整列出所有敏感信号（包括时钟信号和复位信号）。

- 如果敏感信号列表不完整，可能导致综合前后的仿真结果不一致。
- 如果敏感信号表中含有不必要的信号，则又会降低仿真速度。

```
1 PROCESS(x)
2 BEGIN
3   IF x AND y = '1' THEN
4     z <= '1';
5   ELSE
6     z <= '0';
7   END IF;
8 END PROCESS;
```

行为综合后的 RTL 描述：`z <= x AND y;`

敏感信号表

注意 每个进程的敏感信号表必须完整列出所有敏感信号（包括时钟信号和复位信号）。

- 如果敏感信号列表不完整，可能导致综合前后的仿真结果不一致。
- 如果敏感信号表中含有不必要的信号，则又会降低仿真速度。

```
1 PROCESS(x)
2 BEGIN
3   IF x AND y = '1' THEN
4     z <= '1';
5   ELSE
6     z <= '0';
7   END IF;
8 END PROCESS;
```

行为综合后的 RTL 描述：`z <= x AND y;`

被动进程

- 如果在一个进程的顺序语句中没有对任何信号赋值，则该进程被称为被动进程。
- 被动进程用于各种检查，其功能与断言语句类似，但更加灵活。

② 块语句

```
[ 块标号: ] BLOCK [ (保护表达式) ]
    [ 类属子句 ]
    [ 端口子句 ]
    [ 块声明语句 ]
BEGIN
    { 并行语句 }
END BLOCK [ 块标号 ];
```

- 块语句是将若干个并行语句包装起来形成一个子模块。
- 当块中某些输出端口与其他块的输出端口直接相连时，应当将该端口设计为决断信号，并利用保护表达式在块中关闭该端口的驱动源(相当于向该输出端口赋值'Z')。
- 块语句是体现划分机制的语句，它只体现在结构体的划分形式上。
- 块语句不会改变结构体的功能，所以块语句的作用主要是提高并行描述的可读性，以便于仿真、纠错、程序移植和技术交流等。
- VHDL 综合器对块语句不敏感，综合时将忽略块语句的存在。

② 块语句

```
[ 块标号: ] BLOCK [ (保护表达式) ]
    [ 类属子句 ]
    [ 端口子句 ]
    [ 块声明语句 ]
BEGIN
    { 并行语句 }
END BLOCK [ 块标号 ];
```

- 块语句是将若干个并行语句包装起来形成一个子模块。
- 当块中某些输出端口与其他块的输出端口直接相连时，应当将该端口设计为决断信号，并利用保护表达式在块中关闭该端口的驱动源(相当于向该输出端口赋值'Z')。
- 块语句是体现划分机制的语句，它只体现在结构体的划分形式上。
- 块语句不会改变结构体的功能，所以块语句的作用主要是提高并行描述的可读性，以便于仿真、纠错、程序移植和技术交流等。
- VHDL 综合器对块语句不敏感，综合时将忽略块语句的存在。

③ 并行信号赋值语句

条件信号赋值语句

```
[ POSTPONED ] 信号名 <= [ GUARDED ] [ TRANSPORT ]
        [ { 表达式 [ AFTER 时间表达式 ] WHEN 布尔表达式 ELSE } ]
        表达式 [ AFTER 时间表达式 ];
```

选择信号赋值语句

```
[ POSTPONED ] WITH 选择表达式 SELECT
    信号名 <= [ GUARDED ] [ TRANSPORT ]
        { 表达式 [ AFTER 时间表达式 ] WHEN 值域, }
        表达式 [ AFTER 时间表达式 ] WHEN 值域 |OTHERS;
```

一个延缓的并行信号赋值语句，被映射为一个等价的延缓进程。

③ 并行信号赋值语句

条件信号赋值语句

```
[ POSTPONED ] 信号名 <= [ GUARDED ] [ TRANSPORT ]
        [ { 表达式 [ AFTER 时间表达式 ] WHEN 布尔表达式 ELSE } ]
        表达式 [ AFTER 时间表达式 ];
```

选择信号赋值语句

```
[ POSTPONED ] WITH 选择表达式 SELECT
    信号名 <= [ GUARDED ] [ TRANSPORT ]
        { 表达式 [ AFTER 时间表达式 ] WHEN 值域, }
        表达式 [ AFTER 时间表达式 ] WHEN 值域 |OTHERS;
```

一个延缓的并行信号赋值语句，被映射为一个等价的延缓进程。

③ 并行信号赋值语句

条件信号赋值语句

```
[ POSTPONED ] 信号名 <= [ GUARDED ] [ TRANSPORT ]
        [ { 表达式 [ AFTER 时间表达式 ] WHEN 布尔表达式 ELSE } ]
        表达式 [ AFTER 时间表达式 ];
```

选择信号赋值语句

```
[ POSTPONED ] WITH 选择表达式 SELECT
    信号名 <= [ GUARDED ] [ TRANSPORT ]
        { 表达式 [ AFTER 时间表达式 ] WHEN 值域, }
        表达式 [ AFTER 时间表达式 ] WHEN 值域 |OTHERS;
```

一个延缓的并行信号赋值语句，被映射为一个等价的延缓进程。

条件信号赋值语句

目标信号 <= [波形要素 WHEN 条件 ELSE]
[波形要素 WHEN 条件 ELSE]
.....
波形要素 [WHEN 条件]

```
1 z <= in0 AFTER 10ns WHEN s0='0' AND s1='0' ELSE
2     in1 AFTER 10ns WHEN s0='1' AND s1='0' ELSE
3     in2 AFTER 10ns WHEN s0='0' AND s1='1' ELSE
4     in3 AFTER 10ns;
```

1. 当in0、in1、in2、in3、s0 或s1 上发生事件时，赋值语句立即执行。
2. 第一个条件s0='0' AND s1='0' 被检查；
3. 如果为假，检查第二个条件s0='1' AND s1='0'；
4. 如果仍为假，则检查第三个条件，依次类推。
5. 假设s0='0' AND s1='1'，那么10ns 后，in2 的值赋给信号z。

条件信号赋值语句

目标信号 <= [波形要素 WHEN 条件 ELSE]
[波形要素 WHEN 条件 ELSE]
.....
波形要素 [WHEN 条件]

1. 当任何波形表达式（波形要素中的数值表达）或任何条件中的信号上有事件发生时，条件信号赋值语句会判断一次条件值然后执行一次。
2. 对于第一个为真的条件，相应的波形值赋给目标信号。

```
1 z <= in0 AFTER 10ns WHEN s0='0' AND s1='0' ELSE
2     in1 AFTER 10ns WHEN s0='1' AND s1='0' ELSE
3     in2 AFTER 10ns WHEN s0='0' AND s1='1' ELSE
4     in3 AFTER 10ns;
```

1. 当in0、in1、in2、in3、s0 或s1 上发生事件时，赋值语句立即执行。
2. 第一个条件s0='0' AND s1='0' 被检查；
3. 如果为假，检查第二个条件s0='1' AND s1='0'；
4. 如果仍为假，则检查第三个条件，依次类推。

条件信号赋值语句

目标信号 <= [波形要素 WHEN 条件 ELSE]
[波形要素 WHEN 条件 ELSE]
.....
波形要素 [WHEN 条件]

```
1 z <= in0 AFTER 10ns WHEN s0='0' AND s1='0' ELSE
2     in1 AFTER 10ns WHEN s0='1' AND s1='0' ELSE
3     in2 AFTER 10ns WHEN s0='0' AND s1='1' ELSE
4     in3 AFTER 10ns;
```

1. 当in0、in1、in2、in3、s0 或s1 上发生事件时，赋值语句立即执行。
2. 第一个条件s0='0' AND s1='0' 被检查；
3. 如果为假，检查第二个条件s0='1' AND s1='0'；
4. 如果仍为假，则检查第三个条件，依次类推。
5. 假设s0='0' AND s1='1'，那么10ns 后，in2 的值赋给信号z。

条件信号赋值语句

目标信号 <= [波形要素 WHEN 条件 ELSE]
[波形要素 WHEN 条件 ELSE]
.....
波形要素 [WHEN 条件]

```
1 z <= in0 AFTER 10ns WHEN s0='0' AND s1='0' ELSE
2     in1 AFTER 10ns WHEN s0='1' AND s1='0' ELSE
3     in2 AFTER 10ns WHEN s0='0' AND s1='1' ELSE
4     in3 AFTER 10ns;
```

1. 当in0、in1、in2、in3、s0 或s1 上发生事件时，赋值语句立即执行。
2. 第一个条件s0='0' AND s1='0' 被检查；
3. 如果为假，检查第二个条件s0='1' AND s1='0'；
4. 如果仍为假，则检查第三个条件，依次类推。
5. 假设s0='0' AND s1='1'，那么10ns 后，in2 的值赋给信号z。

条件信号赋值语句与其等效进程语句

- 对于一个给定的条件信号赋值语句，有一个相同语义的等效进程语句与之对应。
- 进程语句中有一个IF 语句和一个WAIT 语句。
- WAIT 语句敏感列表中的信号是所有波形表达式和条件中用到的信号。

```
z <= in0 AFTER 10ns WHEN s0='0' AND s1='0' ELSE
     in1 AFTER 10ns WHEN s0='1' AND s1='0' ELSE
     in2 AFTER 10ns WHEN s0='0' AND s1='1' ELSE
     in3 AFTER 10ns;
```

```
PROCESS
BEGIN
  IF s0='0' AND s1='0' THEN
    z <= in0 AFTER 10ns;
  ELSIF s0='1' AND s1='0' THEN
    z <= in1 AFTER 10ns;
  ELSIF s0='0' AND s1='1' THEN
    z <= in2 AFTER 10ns;
  ELSE
    z <= in3 AFTER 10ns;
  END IF;
  WAIT ON in0,in1,in2,in3,s0,s1;
END PROCESS;
```

条件信号赋值语句与其等效进程语句

- 对于一个给定的条件信号赋值语句，有一个相同语义的等效进程语句与之对应。
- 进程语句中有一个IF 语句和一个WAIT 语句。
- WAIT 语句敏感列表中的信号是所有波形表达式和条件中用到的信号。

```
z <= in0 AFTER 10ns WHEN s0='0' AND s1='0' ELSE
     in1 AFTER 10ns WHEN s0='1' AND s1='0' ELSE
     in2 AFTER 10ns WHEN s0='0' AND s1='1' ELSE
     in3 AFTER 10ns;
```

```
PROCESS
BEGIN
  IF s0='0' AND s1='0' THEN
    z <= in0 AFTER 10ns;
  ELSIF s0='1' AND s1='0' THEN
    z <= in1 AFTER 10ns;
  ELSIF s0='0' AND s1='1' THEN
    z <= in2 AFTER 10ns;
  ELSE
    z <= in3 AFTER 10ns;
  END IF;
  WAIT ON in0,in1,in2,in3,s0,s1;
END PROCESS;
```

条件信号赋值语句与其等效进程语句

- 对于一个给定的条件信号赋值语句，有一个相同语义的等效进程语句与之对应。
- 进程语句中有一个IF 语句和一个WAIT 语句。
- WAIT 语句敏感列表中的信号是所有波形表达式和条件中用到的信号。

```
1 z <= in0 AFTER 10ns WHEN s0='0' AND s1='0' ELSE
2      in1 AFTER 10ns WHEN s0='1' AND s1='0' ELSE
3      in2 AFTER 10ns WHEN s0='0' AND s1='1' ELSE
4      in3 AFTER 10ns;
```

```
1  PROCESS
2  BEGIN
3    IF s0='0' AND s1='0' THEN
4      z <= in0 AFTER 10ns;
5    ELSIF s0='1' AND s1='0' THEN
6      z <= in1 AFTER 10ns;
7    ELSIF s0='0' AND s1='1' THEN
8      z <= in2 AFTER 10ns;
9    ELSE
10     z <= in3 AFTER 10ns;
11   END IF;
12   WAIT ON in0,in1,in2,in3,s0,s1;
13 END PROCESS;
```

选择信号赋值语句

```
WITH 表达式 SELECT
    目标信号 <= 波形要素 WHEN 选择,
                波形要素 WHEN 选择,
                ......
                波形要素 WHEN 选择;
```

```
1 TYPE op IS (add,sub,mul,div);
2 SIGNAL op_code: op;
3 ...
4 WITH op_code SELECT
5   z <= a+b AFTER add_prop_delay WHEN add,
6        a-b AFTER sub_prop_delay WHEN sub,
7        a*b AFTER mul_prop_delay WHEN mul,
8        a/b AFTER div_prop_delay WHEN div;
```

1. 只要信号op_code、a 或b 上有事件发生，语句就会执行。
2. 假设选择表达式的值为sub，计算表达式a-b，其值在sub_prop_delay 延迟后赋给信号z。

选择信号赋值语句

```
WITH 表达式 SELECT
    目标信号 <= 波形要素 WHEN 选择,
                波形要素 WHEN 选择,
                ......
                波形要素 WHEN 选择;
```

1. 当选择表达式或是任意波形表达式中使用的信号上有事件发生时，执行该语句。
2. 根据选择表达式（与定义的选择值相匹配）的值，相应的波形被赋予目标信号。
3. 条件值不会按照顺序来选择。
4. 选择表达式的所有可能值必须都出现，且仅出现一次。
5. 没有显式出现的值都包括在 OTHERS 选项中。

```
1 TYPE op IS (add,sub,mul,div);
2 SIGNAL op_code: op;
3 ...
4 WITH op_code SELECT
5   z <= a+b AFTER add_prop_delay WHEN add,
```

选择信号赋值语句

WITH 表达式 SELECT
目标信号 <= 波形要素 WHEN 选择,
波形要素 WHEN 选择,
.....
波形要素 WHEN 选择;

```
1 TYPE op IS (add,sub,mul,div);
2 SIGNAL op_code: op;
3 ...
4 WITH op_code SELECT
5   z <= a+b AFTER add_prop_delay WHEN add,
6        a-b AFTER sub_prop_delay WHEN sub,
7        a*b AFTER mul_prop_delay WHEN mul,
8        a/b AFTER div_prop_delay WHEN div;
```

1. 只要信号op_code、a 或b 上有事件发生，语句就会执行。
2. 假设选择表达式的值为sub，计算表达式a-b，其值在sub_prop_delay 延迟后赋给信号z。

选择信号赋值语句

```
WITH 表达式 SELECT
    目标信号 <= 波形要素 WHEN 选择,
                波形要素 WHEN 选择,
                ......
                波形要素 WHEN 选择;
```

```
1 TYPE op IS (add,sub,mul,div);
2 SIGNAL op_code: op;
3 ...
4 WITH op_code SELECT
5   z <= a+b AFTER add_prop_delay WHEN add,
6        a-b AFTER sub_prop_delay WHEN sub,
7        a*b AFTER mul_prop_delay WHEN mul,
8        a/b AFTER div_prop_delay WHEN div;
```

1. 只要信号op_code、a 或b 上有事件发生，语句就会执行。
2. 假设选择表达式的值为sub，计算表达式a-b，其值在sub_prop_delay 延迟后赋给信号z。

选择信号赋值语句与其等效进程语句

- 每一个选择信号赋值语句都有一个相同语义的等效进程语句。
- 在等效进程语句中，有一个使用选择分支表达式的CASE 语句。
- WAIT 语句敏感列表信号为波形表达式和选择表达式中所有信号。

```
WITH op_code SELECT
  z <= a+b AFTER add_prop_delay WHEN add,
       a-b AFTER sub_prop_delay WHEN sub,
       a*b AFTER mul_prop_delay WHEN mul,
       a/b AFTER div_prop_delay WHEN div;
```

```
PROCESS
BEGIN
  CASE op_code IS
    WHEN add => z <= a+b AFTER add_prop_delay;
    WHEN sub => z <= a-b AFTER sub_prop_delay;
    WHEN mul => z <= a*b AFTER mul_prop_delay;
    WHEN div => z <= a/b AFTER div_prop_delay;
  END CASE;
  WAIT ON op_code,a,b;
END PROCESS;
```

选择信号赋值语句与其等效进程语句

- 每一个选择信号赋值语句都有一个相同语义的等效进程语句。
- 在等效进程语句中，有一个使用选择分支表达式的CASE 语句。
- WAIT 语句敏感列表信号为波形表达式和选择表达式中所有信号。

```
1  WITH op_code SELECT
2    z <= a+b AFTER add_prop_delay WHEN add,
3         a-b AFTER sub_prop_delay WHEN sub,
4         a*b AFTER mul_prop_delay WHEN mul,
5         a/b AFTER div_prop_delay WHEN div;
```

```
1  PROCESS
2  BEGIN
3    CASE op_code IS
4      WHEN add => z <= a+b AFTER add_prop_delay;
5      WHEN sub => z <= a-b AFTER sub_prop_delay;
6      WHEN mul => z <= a*b AFTER mul_prop_delay;
7      WHEN div => z <= a/b AFTER div_prop_delay;
8    END CASE;
9    WAIT ON op_code,a,b;
10 END PROCESS;
```

选择信号赋值语句与其等效进程语句

- 每一个选择信号赋值语句都有一个相同语义的等效进程语句。
- 在等效进程语句中，有一个使用选择分支表达式的CASE 语句。
- WAIT 语句敏感列表信号为波形表达式和选择表达式中所有信号。

```
1  WITH op_code SELECT
2    z <= a+b AFTER add_prop_delay WHEN add,
3         a-b AFTER sub_prop_delay WHEN sub,
4         a*b AFTER mul_prop_delay WHEN mul,
5         a/b AFTER div_prop_delay WHEN div;
```

```
1  PROCESS
2  BEGIN
3    CASE op_code IS
4      WHEN add => z <= a+b AFTER add_prop_delay;
5      WHEN sub => z <= a-b AFTER sub_prop_delay;
6      WHEN mul => z <= a*b AFTER mul_prop_delay;
7      WHEN div => z <= a/b AFTER div_prop_delay;
8    END CASE;
9    WAIT ON op_code,a,b;
10 END PROCESS;
```

UNAFFECTED 值

- VHDL'93 在并行信号赋值语句中将表达式的值域扩展了一个值 UNAFFECTED。
- 它表示信号没有发生赋值操作，不会引起目标信号驱动器的变化。

发生赋值操作

```
1 y <= y;
```

y'Active=True、y'Transaction 发生了一次变化。

未发生赋值操作

```
1 y <= UNAFFECTED;
```

y'Active=False、y'Transaction 没有发生变化。

```
1  TYPE state_type IS (reset,apply,waits,hold,receive);
2  SIGNAL next_state: state_type;
3  ...
4  WITH next_state SELECT
5    zrx <= B"0001" WHEN apply,
6           B"0010" WHEN waits,
7           B"0100" WHEN reset,
8           UNAFFECTED WHEN OTHERS;
9
10 mark_flag <= bkdet AFTER 5ns WHEN strobe='0' ELSE
```

```
1  PROCESS
2  BEGIN
3    IF strobe='0' THEN
4      mark_flag <= bkdet AFTER 5ns;
5    ELSE
6      NULL;
7    END IF;
8    WAIT ON strobe,bkdet;
```

UNAFFECTED 值

发生赋值操作

```
1 y <= y;
```

y'Active=True、y'Transaction 发生了一次变化。

未发生赋值操作

```
1 y <= UNAFFECTED;
```

y'Active=False、y'Transaction 没有发生变化。

```
1  TYPE state_type IS (reset,apply,waits,hold,receive);
2  SIGNAL next_state: state_type;
3  ...
4  WITH next_state SELECT
5    zrx <= B"0001" WHEN apply,
6           B"0010" WHEN waits,
7           B"0100" WHEN reset,
8           UNAFFECTED WHEN OTHERS;
9
10 mark_flag <= bkdet AFTER 5ns WHEN strobe='0' ELSE
11              UNAFFECTED;
```

```
1 PROCESS
2 BEGIN
3   IF strobe='0' THEN
4     mark_flag <= bkdet AFTER 5ns;
5   ELSE
6     NULL;
7   END IF;
8   WAIT ON strobe,bkdet;
9 END PROCESS;
```

UNAFFECTED 值

发生赋值操作

```
1 y <= y;
```

y'Active=True、y'Transaction 发生了一次变化。

未发生赋值操作

```
1 y <= UNAFFECTED;
```

y'Active=False、y'Transaction 没有发生变化。

```
1  TYPE state_type IS (reset,apply,waits,hold,receive);
2  SIGNAL next_state: state_type;
3  ...
4  WITH next_state SELECT
5    zrx <= B"0001" WHEN apply,
6           B"0010" WHEN waits,
7           B"0100" WHEN reset,
8           UNAFFECTED WHEN OTHERS;
9
10 mark_flag <= bkdet AFTER 5ns WHEN strobe='0' ELSE
11              UNAFFECTED;
```

```
1 PROCESS
2 BEGIN
3   IF strobe='0' THEN
4     mark_flag <= bkdet AFTER 5ns;
5   ELSE
6     NULL;
7   END IF;
8   WAIT ON strobe,bkdet;
9 END PROCESS;
```

④ 并行过程调用语句

[POSTPONED] 过程名（实参表）;

- 可以作为并行语句，用于结构体或块中，等价于一个只有单个顺序过程调用语句的进程。
- 一个延缓的并行过程调用语句，可以被映射为一个等价的延缓进程。

④ 并行过程调用语句

[POSTPONED] 过程名（实参表）;

- 可以作为并行语句，用于结构体或块中，等价于一个只有单个顺序过程调用语句的进程。

```
example(sig1,sig2);
```

```
PROCESS(sig1,sig2)
BEGIN
  example(sig1,sig2);
END PROCESS;
```

- 一个延缓的并行过程调用语句，可以被映射为一个等价的延缓进程。

④ 并行过程调用语句

[POSTPONED] 过程名（实参表）;

- 可以作为并行语句，用于结构体或块中，等价于一个只有单个顺序过程调用语句的进程。

```
example(sig1,sig2);
```

```
PROCESS(sig1,sig2)
BEGIN
  example(sig1,sig2);
END PROCESS;
```

- 一个延缓的并行过程调用语句，可以被映射为一个等价的延缓进程。

④ 并行过程调用语句

[POSTPONED] 过程名（实参表）;

- 可以作为并行语句，用于结构体或块中，等价于一个只有单个顺序过程调用语句的进程。

```
1 example(sig1,sig2);
```

```
1 PROCESS(sig1,sig2)
2 BEGIN
3   example(sig1,sig2);
4 END PROCESS;
```

- 一个延缓的并行过程调用语句，可以被映射为一个等价的延缓进程。

④ 并行过程调用语句

[POSTPONED] 过程名 (实参表);

- 可以作为并行语句，用于结构体或块中，等价于一个只有单个顺序过程调用语句的进程。

```
1 example(sig1,sig2);
```

```
1 PROCESS(sig1,sig2)
2 BEGIN
3   example(sig1,sig2);
4 END PROCESS;
```

- 一个延缓的并行过程调用语句，可以被映射为一个等价的延缓进程。

④ 并行过程调用语句

[POSTPONED] 过程名 (实参表);

- 可以作为并行语句，用于结构体或块中，等价于一个只有单个顺序过程调用语句的进程。

```
1 example(sig1,sig2);
```

```
1 PROCESS(sig1,sig2)
2 BEGIN
3   example(sig1,sig2);
4 END PROCESS;
```

- 一个延缓的并行过程调用语句，可以被映射为一个等价的延缓进程。

⑤ 并行断言语句

[POSTPONED] ASSERT 布尔表达式 [REPORT 信息][SEVERITY 错误等级];

- 可以作为并行语句用于结构体或块中，等价于一个只有单个顺序断言语句的进程。
- 一个延缓的并行断言进程，可以被映射为一个等价的延缓进程。

⑤ 并行断言语句

[POSTPONED] ASSERT 布尔表达式 [REPORT 信息][SEVERITY 错误等级];

- 可以作为并行语句用于结构体或块中，等价于一个只有单个顺序断言语句的进程。

```
ASSERT(sig /= 'X')
  REPORT "Uncertainty value on sig"
  SEVERITY Warning;
```

```
PROCESS(sig)
BEGIN
  ASSERT(sig /= 'X')
    REPORT "Uncertainty value on sig"
    SEVERITY Warning;
END PROCESS;
```

- 一个延缓的并行断言进程，可以被映射为一个等价的延缓进程。

⑤ 并行断言语句

[POSTPONED] ASSERT 布尔表达式 [REPORT 信息][SEVERITY 错误等级];

- 可以作为并行语句用于结构体或块中，等价于一个只有单个顺序断言语句的进程。

```
ASSERT(sig /= 'X')
  REPORT "Uncertainty value on sig."
  SEVERITY Warning;
```

```
PROCESS(sig)
BEGIN
  ASSERT(sig /= 'X')
    REPORT "Uncertainty value on sig."
    SEVERITY Warning;
END PROCESS;
```

- 一个延缓的并行断言进程，可以被映射为一个等价的延缓进程。

⑤ 并行断言语句

[POSTPONED] ASSERT 布尔表达式 [REPORT 信息][SEVERITY 错误等级];

- 可以作为并行语句用于结构体或块中，等价于一个只有单个顺序断言语句的进程。

```
1 ASSERT(sig /= 'X')
2   REPORT "Uncertainty value on sig."
3   SEVERITY Warning;
```

```
1 PROCESS(sig)
2 BEGIN
3   ASSERT(sig /= 'X')
4     REPORT "Uncertainty value on sig."
5     SEVERITY Warning;
6 END PROCESS;
```

- 一个延缓的并行断言进程，可以被映射为一个等价的延缓进程。

⑤ 并行断言语句

[POSTPONED] ASSERT 布尔表达式 [REPORT 信息][SEVERITY 错误等级];

- 可以作为并行语句用于结构体或块中，等价于一个只有单个顺序断言语句的进程。

```
1 ASSERT(sig /= 'X')
2   REPORT "Uncertainty value on sig."
3   SEVERITY Warning;
```

```
1 PROCESS(sig)
2 BEGIN
3   ASSERT(sig /= 'X')
4     REPORT "Uncertainty value on sig."
5     SEVERITY Warning;
6 END PROCESS;
```

- 一个延缓的并行断言进程，可以被映射为一个等价的延缓进程。

⑤ 并行断言语句

[POSTPONED] ASSERT 布尔表达式 [REPORT 信息][SEVERITY 错误等级];

- 可以作为并行语句用于结构体或块中，等价于一个只有单个顺序断言语句的进程。

```
1 ASSERT(sig /= 'X')
2   REPORT "Uncertainty value on sig."
3   SEVERITY Warning;
```

```
1 PROCESS(sig)
2 BEGIN
3   ASSERT(sig /= 'X')
4     REPORT "Uncertainty value on sig."
5     SEVERITY Warning;
6 END PROCESS;
```

- 一个延缓的并行断言进程，可以被映射为一个等价的延缓进程。

⑥ 元件例化语句

- VHDL 中的元件例化语句提供了一种可以重复利用设计库中已有设计的机制，常用于在设计实体的顶层描述中。
- 元件例化语句标识了一个例化元件（子元件），同时也将例化元件的实际参数和与之对应的模板元件（父元件）的形式参数相关联。

```
[ 例化元件名 ] 模板元件名 [ GENERIC MAP (类属实参表) ] PORT MAP (信号实参表);
```

- 例化元件名是例化元件的元件名称（子元件名），模板元件名（父元件名）是在元件声明语句中声明的元件名。
- 在元件例化语句中引用的模板元件，必须先用元件声明语句予以声明。
- 元件声明语句可以在当前结构体中，也可以在程序包中用于声明模板文件。
- 例化元件的实际参数映射既可以用位置关联法也可以用名称关联法，甚至也可以用它们的混合关联，与模板元件相应的形式参数相关联。
- 位置关联法就是在进行类属映射和端口映射时，实参表中的参数映射顺序，与元件声明语句中类属子句和端口自己的参数声明顺序相

⑥ 元件例化语句

[例化元件名] 模板元件名 [GENERIC MAP（类属实参表）] PORT MAP（信号实参表）;

- 例化元件名是例化元件的元件名称（子元件名），模板元件名（父元件名）是在元件声明语句中声明的元件名。
- 在元件例化语句中引用的模板元件，必须先用元件声明语句予以声明。
- 元件声明语句可以在当前结构体中，也可以在程序包中用于声明模板文件。
- 例化元件的实际参数映射既可以用位置关联法也可以用名称关联法，甚至也可以用它们的混合关联，与模板元件相应的形式参数相关联。
- 位置关联法就是在进行类属映射和端口映射时，实参表中的参数映射顺序，与元件声明语句中类属子句和端口自己的参数声明顺序相同。
- 名称关联的格式为：形式参数 => 实际参数；建议采用名称关联法，以使类属映射和端口映射更具可读性。

1位全加器

```
ENTITY half_adder IS
  PORT(a, b: IN Bit;
       sum, carry: OUT Bit);
END half_adder;
ARCHITECTURE behavl OF half_adder
BEGIN
  WITH a & b SELECT
    sum <= '1' WHEN "01"|"10",
           '0' WHEN OTHERS;
  WITH a & b SELECT
    carry <= '1' WHEN "11",
             '0' WHEN OTHERS;
END behavl;
```

```
ENTITY full_adder IS
  PORT(a, b, c_in: IN Bit;
       sum, c_out: OUT Bit);
END full_adder;
ARCHITECTURE toplevel OF full_adder IS
  SIGNAL temp_sum, temp_c1, temp_c2: Bit;
  COMPONENT half_adder
    PORT(a, b: IN Bit;
         sum, carry: OUT Bit);
  END COMPONENT;
BEGIN
  U0: half_adder PORT MAP(a => a, b => b,
       sum => temp_sum, carrry => temp_c1);
  U1: half_adder PORT MAP(a => temp_sum, b => c
       sum => sum, carry => temp_c2);
  c_out <= temp_c1 OR temp_c2;
END toplevel;
```

- 对于 VHDL 标准程序包中声明的枚举类型Bit，在布尔表达式或者并行赋值语句和CASE 语句的选择表达式中，都可以直接使用连接运算符&。
- 但对于其他没有在 VHDL 标准程序包中声明的枚举类型（例如Std_Logic 类型）则不能在布尔表达式或者并行赋值语句和CASE 语句的选择表达式中直接使用连接运算符&，而必须先将连接运算的结果赋给一个中间变量，然后在选择表达式中引用这个中间变量。

VHDL 对元件例化语句的实参与形参之间的关联

- 数据类型必须一致。
- 信号模式必须一致。
- 决断性必须一致，即如果形参是决断信号的话，则实参是与形参相同的决断信号，且具有一个的决断函数。
- 当例化元件的某个端口不存在时，相应的实际参数缺失，则要使用关键字OPEN，使模板元件被例化时该端口处于未连接状态。
- 每条元件例化语句都会产生一个电路模块（即一个元件）。

- 对于 VHDL 标准程序包中声明的枚举类型Bit，在布尔表达式或者并行赋值语句和CASE 语句的选择表达式中，都可以直接使用连接运算符&。
- 但对于其他没有在 VHDL 标准程序包中声明的枚举类型（例如Std_Logic 类型）则不能在布尔表达式或者并行赋值语句和CASE 语句的选择表达式中直接使用连接运算符&，而必须先将连接运算的结果赋给一个中间变量，然后在选择表达式中引用这个中间变量。

VHDL 对元件例化语句的实参与形参之间的关联

- 数据类型必须一致。
- 信号模式必须一致。
- 决断性必须一致，即如果形参是决断信号的话，则实参是与形参相同的决断信号，且具有一个的决断函数。
- 当例化元件的某个端口不存在时，相应的实际参数缺失，则要使用关键字OPEN，使模板元件被例化时该端口处于未连接状态。
- 每条元件例化语句都会产生一个电路模块（即一个元件）。

- 对于VHDL标准程序包中声明的枚举类型Bit，在布尔表达式或者并行赋值语句和CASE语句的选择表达式中，都可以直接使用连接运算符&。
- 但对于其他没有在VHDL标准程序包中声明的枚举类型（例如Std_Logic类型）则不能在布尔表达式或者并行赋值语句和CASE语句的选择表达式中直接使用连接运算符&，而必须先将连接运算的结果赋给一个中间变量，然后在选择表达式中引用这个中间变量。

VHDL对元件例化语句的实参与形参之间的关联

- 数据类型必须一致。
- 信号模式必须一致。
- 决断性必须一致，即如果形参是决断信号的话，则实参是与形参相同的决断信号，且具有一个的决断函数。
- 当例化元件的某个端口不存在时，相应的实际参数缺失，则要使用关键字OPEN，使模板元件被例化时该端口处于未连接状态。
- 每条元件例化语句都会产生一个电路模块（即一个元件）。

OPEN

- 当模板元件的输出端口处于未连接状态时，应当用OPEN 予以关联。
- 在使用名称关联方式时，允许省略未连接状态的端口关联；
- 而使用位置关联方式时，则不允许省略未连接状态的端口关联。

```
1 U1: dff PORT MAP(d => s_in, ck => clk, q => s(1));
2 U1: dff PORT MAP(s_in,clk,s(1),OPEN);
```

- 如果模板元件的某个输入端口未连接时，则不允许使用OPEN 予以关联，因为 VHDL 仿真器和综合器均不支持输入“浮动”状态。
- 在实际电路中，这种状态也是不允许出现的。
- 当模板元件的输入端口处于没有信号予以关联时，必须指定该端口的关联信号值，而不能处于“浮动”（OPEN）状态。

```
1 Vcc <= '1';
2 U0: ex1 PORT MAP(a => sig1, b => Vcc, c => sig2, y1 => sig3, y2 => sig4);
3 U0: ex1 PORT MAP(a => sig1, b => '1', c => sig2, y1 => sig3, y2 => sig4);
4 U0: ex1 PORT MAP(a => sig1, b => OPEN, c => sig2, y1 => sig3, y2 => sig4); ERROR
```

OPEN

- 当模板元件的输出端口处于未连接状态时，应当用OPEN 予以关联。
- 在使用名称关联方式时，允许省略未连接状态的端口关联；
- 而使用位置关联方式时，则不允许省略未连接状态的端口关联。

```
1 U1: dff PORT MAP(d => s_in, ck => clk, q => s(1));
2 U1: dff PORT MAP(s_in,clk,s(1),OPEN);
```

- 如果模板元件的某个输入端口未连接时，则不允许使用OPEN 予以关联，因为 VHDL 仿真器和综合器均不支持输入“浮动”状态。
- 在实际电路中，这种状态也是不允许出现的。
- 当模板元件的输入端口处于没有信号予以关联时，必须指定该端口的关联信号值，而不能处于“浮动”（OPEN）状态。

```
1 Vcc <= '1';
2 U0: ex1 PORT MAP(a => sig1, b => Vcc, c => sig2, y1 => sig3, y2 => sig4);
3 U0: ex1 PORT MAP(a => sig1, b => '1', c => sig2, y1 => sig3, y2 => sig4);
4 U0: ex1 PORT MAP(a => sig1, b => OPEN, c => sig2, y1 => sig3, y2 => sig4); ERROR
```

OPEN

- 当模板元件的输出端口处于未连接状态时，应当用OPEN 予以关联。
- 在使用名称关联方式时，允许省略未连接状态的端口关联；
- 而使用位置关联方式时，则不允许省略未连接状态的端口关联。

```
1 U1: dff PORT MAP(d => s_in, ck => clk, q => s(1));
2 U1: dff PORT MAP(s_in,clk,s(1),OPEN);
```

- 如果模板元件的某个输入端口未连接时，则不允许使用OPEN 予以关联，因为 VHDL 仿真器和综合器均不支持输入“浮动”状态。
- 在实际电路中，这种状态也是不允许出现的。
- 当模板元件的输入端口处于没有信号予以关联时，必须指定该端口的关联信号值，而不能处于“浮动”（OPEN）状态。

```
1 Vcc <= '1';
2 U0: ex1 PORT MAP(a => sig1, b => Vcc, c => sig2, y1 => sig3, y2 => sig4);
3 U0: ex1 PORT MAP(a => sig1, b => '1', c => sig2, y1 => sig3, y2 => sig4);
4 U0: ex1 PORT MAP(a => sig1, b => OPEN, c => sig2, y1 => sig3, y2 => sig4); ERROR
```

OPEN

- 当模板元件的输出端口处于未连接状态时，应当用OPEN 予以关联。
- 在使用名称关联方式时，允许省略未连接状态的端口关联；
- 而使用位置关联方式时，则不允许省略未连接状态的端口关联。

```
1 U1: dff PORT MAP(d => s_in, ck => clk, q => s(1));
2 U1: dff PORT MAP(s_in,clk,s(1),OPEN);
```

- 如果模板元件的某个输入端口未连接时，则不允许使用OPEN 予以关联，因为 VHDL 仿真器和综合器均不支持输入“浮动”状态。
- 在实际电路中，这种状态也是不允许出现的。
- 当模板元件的输入端口处于没有信号予以关联时，必须指定该端口的关联信号值，而不能处于“浮动”（OPEN）状态。

```
1 Vcc <= '1';
2 U0: ex1 PORT MAP(a => sig1, b => Vcc, c => sig2, y1 => sig3, y2 => sig4);
3 U0: ex1 PORT MAP(a => sig1, b => '1', c => sig2, y1 => sig3, y2 => sig4);
4 U0: ex1 PORT MAP(a => sig1, b => OPEN, c => sig2, y1 => sig3, y2 => sig4); ERROR
```

8位移位寄存器

```
1  ENTITY shift IS
2    GENERIC (len: Integer := 8);
3    PORT (s_in, clk: IN Bit;
4          s_out: OUT Bit);
5  END shift;
6  ARCHITECTURE ungen_shift OF shift IS
7    SIGNAL s: Bit_Vector(1 TO (len-1));
8    COMPONENT dff
9      PORT (d, ck: IN Bit; q, q_n: OUT Bit);
10   END COMPONENT;
11 BEGIN
12   U1: dff PORT MAP(d=>s_in,ck=>clk,q=>s(1),q_n=>OPEN);
13   U2: dff PORT MAP(d=>s(1),ck=>clk,q=>s(2),q_n=>OPEN);
14   U3: dff PORT MAP(d=>s(2),ck=>clk,q=>s(3),q_n=>OPEN);
15   U4: dff PORT MAP(d=>s(3),ck=>clk,q=>s(4),q_n=>OPEN);
16   U5: dff PORT MAP(d=>s(4),ck=>clk,q=>s(5),q_n=>OPEN);
17   U6: dff PORT MAP(d=>s(5),ck=>clk,q=>s(6),q_n=>OPEN);
18   U7: dff PORT MAP(d=>s(6),ck=>clk,q=>s(7),q_n=>OPEN);
19   U8: dff PORT MAP(d=>s(7),ck=>clk,q=>s_out,q_n=>OPEN);
20 END ungen_shift;
```

⑦ 生成语句

- 生成语句提供了描述重复结构（或称规则结构）和例外情况的机制。
- 它可对设计中规则的重复结构以循环的形式进行描述，也可对重复结构的不规则边界以选择的形式进行描述。

```
[ 生成标号: ] 生成模式 GENERATE
    { 并行语句 }
END GENERATE [ 生成标号 ];
```

- 生成模式可以是FOR 循环变量 IN 离散范围，用于描述规则的重复结构；
- 也可以是IF 布尔表达式，用于描述重复结构的不规则边界；
- 与顺序循环语句LOOP 类似，在FOR 模式的生成语句中，循环变量无须声明，在生成语句外部循环变量不可见，而在生成语句内部，则只能对循环变量进行读访问，不能对其赋值。
- 与顺序循环语句LOOP 不同，在FOR 模式的生成语句中，没有与NEXT 语句和EXIT 语句相似的并行语句。
- 在IF 模式的生成语句中，也没有ELSIF 和ELSE 分支。

⑦ 生成语句

[生成标号:] 生成模式 GENERATE
  { 并行语句 }
END GENERATE [生成标号];

- 生成模式可以是FOR 循环变量 IN 离散范围，用于描述规则的重复结构；
- 也可以是IF 布尔表达式，用于描述重复结构的不规则边界；
- 与顺序循环语句LOOP 类似，在FOR 模式的生成语句中，循环变量无须声明，在生成语句外部循环变量不可见，而在生成语句内部，则只能对循环变量进行读访问，不能对其赋值。
- 与顺序循环语句LOOP 不同，在FOR 模式的生成语句中，没有与NEXT 语句和EXIT 语句相似的并行语句。
- 在IF 模式的生成语句中，也没有ELSIF 和ELSE 分支。
- 对于重复结构使用生成语句的描述要比不用生成语句的描述简捷得多。

8位移位寄存器

```
1  ARCHITECTURE for_shift OF shift IS
2    SIGNAL s: Bit_Vector (0 TO len);
3  BEGIN
4    s(0) <= s_in;
5    FOR i IN 0 TO (len-1) GENERATE
6      Ui: dff PORT MAP(d=>s(i),ck=>clk,q=>s(i+1),q_n=>OPEN);
7    END GENERATE;
8    s_out <= s(len);
9  END for_shift;
```

```
1  ARCHITECTURE if_shift OF shift IS
2    SIGNAL s: Bit_Vector (1 TO (len-1));
3  BEGIN
4    FOR i IN 0 TO (len-1) GENERATE
5      IF i=0 GENERATE
6        U1: dff PORT MAP(d=>s_in,ck=>clk,q=>s(i+1),q_n=>OPEN);
7      END GENERATE;
8      IF (i>0) AND (i<(len-1)) GENERATE
9        Ui: dff PORT MAP(d=>s(i),ck=>clk,q=>s(i+1),q_n=>OPEN);
10     END GENERATE;
11     IF i=(len-1) GENERATE
12       Ulen: dff PORT MAP(d=>s(i),ck=>clk,q=>s_out,q_n=>OPEN);
13     END GENERATE;
14   END GENERATE;
15 END if_shift;
```

8位移位寄存器

```
1  ARCHITECTURE for_shift OF shift IS
2    SIGNAL s: Bit_Vector (0 TO len);
3  BEGIN
4    s(0) <= s_in;
5    FOR i IN 0 TO (len-1) GENERATE
6      Ui: dff PORT MAP(d=>s(i),ck=>clk,q=>s(i+1),q_n=>OPEN);
7    END GENERATE;
8    s_out <= s(len);
9  END for_shift;
```

```
1  ARCHITECTURE if_shift OF shift IS
2    SIGNAL s: Bit_Vector (1 TO (len-1));
3  BEGIN
4    FOR i IN 0 TO (len-1) GENERATE
5      IF i=0 GENERATE
6        U1: dff PORT MAP(d=>s_in,ck=>clk,q=>s(i+1),q_n=>OPEN);
7      END GENERATE;
8      IF (i>0) AND (i<(len-1)) GENERATE
9        Ui: dff PORT MAP(d=>s(i),ck=>clk,q=>s(i+1),q_n=>OPEN);
10     END GENERATE;
11     IF i=(len-1) GENERATE
12       Ulen: dff PORT MAP(d=>s(i),ck=>clk,q=>s_out,q_n=>OPEN);
13     END GENERATE;
14   END GENERATE;
15 END if_shift;
```