数字系统设计

郑海永

中国海洋大学 电子工程系

2014 年 6 月

VHDL

目录

内容提要

VHDL 基本术语

设计单元

- 实体声明
- 结构体
- 配置声明
- 程序包声明
- 程序包体

程序包 体现了 VHDL 共享资源的运用。

设计库 VHDL 各种结构的集合，包含实体、结构体、程序包和配置等。

配置 用于建立设计库中实体与元件或者实体与结构体之间的连接关系。

子程序 程序包中的一个重要成分。

VHDL 基本术语

设计单元

- 实体声明
- 结构体
- 配置声明
- 程序包声明
- 程序包体

程序包 体现了 VHDL 共享资源的运用。

设计库 VHDL 各种结构的集合，包含实体、结构体、程序包和配置等。

配置 用于建立设计库中实体与元件或者实体与结构体之间的连接关系。

子程序 程序包中的一个重要成分。

内容提要

子程序

程序包

- 设计者在一个设计实体中声明的数据类型、对象、子程序、元件声明和属性等，对于其他实体而言是不可见的，不能被利用。
- 为了使 VHDL 代码具有可重用性，VHDL 提供了将共享资源封装在程序包中单独编译，并可以为不同的设计实体所利用的共享机制——程序包。
- 程序包中的一个重要成分就是子程序。

自定义子程序

- 可以在结构体的声明部分或者在进程语句的声明部分中声明子程序，可以在该结构体或者该进程中调用子程序。
- 如果在某个设计实体的多个结构体都要调用子程序，或者在不同的设计实体中要调用子程序，则应当在程序包中声明该子程序。
- 子程序是 VHDL 的重要共享资源之一，子程序可被看作设计者自定义的运算符。
- VHDL 提供两种子程序：函数和过程。

子程序

程序包

- 设计者在一个设计实体中声明的数据类型、对象、子程序、元件声明和属性等，对于其他实体而言是不可见的，不能被利用。
- 为了使 VHDL 代码具有可重用性，VHDL 提供了将共享资源封装在程序包中单独编译，并可以为不同的设计实体所利用的共享机制——程序包。
- 程序包中的一个重要成分就是子程序。

自定义子程序

- 可以在结构体的声明部分或者在进程语句的声明部分中声明子程序，可以在该结构体或者该进程中调用子程序。
- 如果在某个设计实体的多个结构体都要调用子程序，或者在不同的设计实体中要调用子程序，则应当在程序包中声明该子程序。
- 子程序是 VHDL 的重要共享资源之一，子程序可被看作设计者自定义的运算符。
- VHDL 提供两种子程序：函数和过程。

子程序

程序包

- 设计者在一个设计实体中声明的数据类型、对象、子程序、元件声明和属性等，对于其他实体而言是不可见的，不能被利用。
- 为了使 VHDL 代码具有可重用性，VHDL 提供了将共享资源封装在程序包中单独编译，并可以为不同的设计实体所利用的共享机制——程序包。
- 程序包中的一个重要成分就是子程序。

自定义子程序

- 可以在结构体的声明部分或者在进程语句的声明部分中声明子程序，可以在该结构体或者该进程中调用子程序。
- 如果在某个设计实体的多个结构体都要调用子程序，或者在不同的设计实体中要调用子程序，则应当在程序包中声明该子程序。
- 子程序是 VHDL 的重要共享资源之一，子程序可被看作设计者自定义的运算符。
- VHDL 提供两种子程序：函数和过程。

子程序

程序包

- 设计者在一个设计实体中声明的数据类型、对象、子程序、元件声明和属性等，对于其他实体而言是不可见的，不能被利用。
- 为了使 VHDL 代码具有可重用性，VHDL 提供了将共享资源封装在程序包中单独编译，并可以为不同的设计实体所利用的共享机制——程序包。
- 程序包中的一个重要成分就是子程序。

自定义子程序

- 可以在结构体的声明部分或者在进程语句的声明部分中声明子程序，可以在该结构体或者该进程中调用子程序。
- 如果在某个设计实体的多个结构体都要调用子程序，或者在不同的设计实体中要调用子程序，则应当在程序包中声明该子程序。
- 子程序是 VHDL 的重要共享资源之一，子程序可被看作设计者自定义的运算符。
- VHDL 提供两种子程序：函数和过程。

子程序

程序包

- 设计者在一个设计实体中声明的数据类型、对象、子程序、元件声明和属性等，对于其他实体而言是不可见的，不能被利用。
- 为了使 VHDL 代码具有可重用性，VHDL 提供了将共享资源封装在程序包中单独编译，并可以为不同的设计实体所利用的共享机制——程序包。
- 程序包中的一个重要成分就是子程序。

自定义子程序

- 可以在结构体的声明部分或者在进程语句的声明部分中声明子程序，可以在该结构体或者该进程中调用子程序。
- 如果在某个设计实体的多个结构体都要调用子程序，或者在不同的设计实体中要调用子程序，则应当在程序包中声明该子程序。
- 子程序是 VHDL 的重要共享资源之一，子程序可被看作设计者自定义的运算符。
- VHDL 提供两种子程序：函数和过程。

子程序

程序包

- 设计者在一个设计实体中声明的数据类型、对象、子程序、元件声明和属性等，对于其他实体而言是不可见的，不能被利用。
- 为了使 VHDL 代码具有可重用性，VHDL 提供了将共享资源封装在程序包中单独编译，并可以为不同的设计实体所利用的共享机制——程序包。
- 程序包中的一个重要成分就是子程序。

自定义子程序

- 可以在结构体的声明部分或者在进程语句的声明部分中声明子程序，可以在该结构体或者该进程中调用子程序。
- 如果在某个设计实体的多个结构体都要调用子程序，或者在不同的设计实体中要调用子程序，则应当在程序包中声明该子程序。
- 子程序是 VHDL 的重要共享资源之一，子程序可被看作设计者自定义的运算符。
- VHDL 提供两种子程序：函数和过程。

子程序

程序包

- 设计者在一个设计实体中声明的数据类型、对象、子程序、元件声明和属性等，对于其他实体而言是不可见的，不能被利用。
- 为了使 VHDL 代码具有可重用性，VHDL 提供了将共享资源封装在程序包中单独编译，并可以为不同的设计实体所利用的共享机制——程序包。
- 程序包中的一个重要成分就是子程序。

自定义子程序

- 可以在结构体的声明部分或者在进程语句的声明部分中声明子程序，可以在该结构体或者该进程中调用子程序。
- 如果在某个设计实体的多个结构体都要调用子程序，或者在不同的设计实体中要调用子程序，则应当在程序包中声明该子程序。
- 子程序是 VHDL 的重要共享资源之一，子程序可被看作设计者自定义的运算符。
- VHDL 提供两种子程序：函数和过程。

子程序

程序包

- 设计者在一个设计实体中声明的数据类型、对象、子程序、元件声明和属性等，对于其他实体而言是不可见的，不能被利用。
- 为了使 VHDL 代码具有可重用性，VHDL 提供了将共享资源封装在程序包中单独编译，并可以为不同的设计实体所利用的共享机制——程序包。
- 程序包中的一个重要成分就是子程序。

自定义子程序

- 可以在结构体的声明部分或者在进程语句的声明部分中声明子程序，可以在该结构体或者该进程中调用子程序。
- 如果在某个设计实体的多个结构体都要调用子程序，或者在不同的设计实体中要调用子程序，则应当在程序包中声明该子程序。
- 子程序是 VHDL 的重要共享资源之一，子程序可被看作设计者自定义的运算符。
- VHDL 提供两种子程序：函数和过程。

函数和过程

函数（FUNCTION）

- 函数被表达式所调用，只返回一个值。
- 函数常常被用来计算一个单一的值。
- 执行函数的仿真时间为零。

过程（PROCEDURE）

- 过程被过程调用语句启动，可以不返回任何值，也可以返回多个结果。
- 过程可以被用来划分大规模的行为级描述。
- 执行一个过程的仿真时间可以为零，也可以不为零，这取决于它是否有一个WAIT 语句。

函数和过程

函数（FUNCTION）

- 函数被表达式所调用，只返回一个值。
- 函数常常被用来计算一个单一的值。
- 执行函数的仿真时间为零。

过程（PROCEDURE）

- 过程被过程调用语句启动，可以不返回任何值，也可以返回多个结果。
- 过程可以被用来划分大规模的行为级描述。
- 执行一个过程的仿真时间可以为零，也可以不为零，这取决于它是否有一个WAIT 语句。

① 函数

当设计者自定义的“运算”只有一个返回值时，常用函数来描述该运算。

```
FUNCTION 函数名（形参表）RETURN 返回值类型 IS
    { 声明语句 }
BEGIN
    { 顺序语句 }
END [ FUNCTION ] [ 函数名 ];
```

- 在声明函数的形参表中，形式参数可以是常量和模式为IN的信号，如果没有声明形式参数的对象种类，则其缺省种类为常量。
- 在声明语句部分，可以声明函数内部要用到的数据类型和局部对象，但不能是信号。
- 函数中的语句都是顺序语句。
- 在函数内部，不允许包含WAIT语句和信号赋值语句（因为参数模式被限定为IN）。

① 函数

当设计者自定义的“运算”只有一个返回值时，常用函数来描述该运算。

```
FUNCTION 函数名 (形参表) RETURN 返回值类型 IS
    { 声明语句 }
BEGIN
    { 顺序语句 }
END [ FUNCTION ] [ 函数名 ];
```

- 在声明函数的形参表中，形式参数可以是常量和模式为IN的信号，如果没有声明形式参数的对象种类，则其缺省种类为常量。
- 在声明语句部分，可以声明函数内部要用到的数据类型和局部对象，但不能是信号。
- 函数中的语句都是顺序语句。
- 在函数内部，不允许包含WAIT语句和信号赋值语句（因为参数模式被限定为IN）。

① 函数

当设计者自定义的“运算”只有一个返回值时，常用函数来描述该运算。

```
FUNCTION 函数名 (形参表) RETURN 返回值类型 IS
    { 声明语句 }
BEGIN
    { 顺序语句 }
END [ FUNCTION ] [ 函数名 ];
```

- 在声明函数的形参表中，形式参数可以是常量和模式为IN 的信号，如果没有声明形式参数的对象种类，则其缺省种类为常量。
- 在声明语句部分，可以声明函数内部要用到的数据类型和局部对象，但不能是信号。
- 函数中的语句都是顺序语句。
- 在函数内部，不允许包含WAIT语句和信号赋值语句（因为参数模式被限定为IN）。

① 函数

当设计者自定义的“运算”只有一个返回值时，常用函数来描述该运算。

```
FUNCTION 函数名 (形参表) RETURN 返回值类型 IS
    { 声明语句 }
BEGIN
    { 顺序语句 }
END [ FUNCTION ] [ 函数名 ];
```

- 在声明函数的形参表中，形式参数可以是常量和模式为IN 的信号，如果没有声明形式参数的对象种类，则其缺省种类为常量。
- 在声明语句部分，可以声明函数内部要用到的数据类型和局部对象，但不能是信号。
- 函数中的语句都是顺序语句。
- 在函数内部，不允许包含WAIT语句和信号赋值语句（因为参数模式被限定为IN）。

① 函数

当设计者自定义的“运算”只有一个返回值时，常用函数来描述该运算。

```
FUNCTION 函数名 (形参表) RETURN 返回值类型 IS
    { 声明语句 }
BEGIN
    { 顺序语句 }
END [ FUNCTION ] [ 函数名 ];
```

- 在声明函数的形参表中，形式参数可以是常量和模式为IN 的信号，如果没有声明形式参数的对象种类，则其缺省种类为常量。
- 在声明语句部分，可以声明函数内部要用到的数据类型和局部对象，但不能是信号。
- 函数中的语句都是顺序语句。
- 在函数内部，不允许包含WAIT语句和信号赋值语句（因为参数模式被限定为IN）。

① 函数

当设计者自定义的“运算”只有一个返回值时，常用函数来描述该运算。

```
FUNCTION 函数名 (形参表) RETURN 返回值类型 IS
    { 声明语句 }
BEGIN
    { 顺序语句 }
END [ FUNCTION ] [ 函数名 ];
```

- 在声明函数的形参表中，形式参数可以是常量和模式为IN 的信号，如果没有声明形式参数的对象种类，则其缺省种类为常量。
- 在声明语句部分，可以声明函数内部要用到的数据类型和局部对象，但不能是信号。
- 函数中的语句都是顺序语句。
- 在函数内部，不允许包含WAIT语句和信号赋值语句（因为参数模式被限定为IN）。

求解函数和类型转换函数

类型转换函数

1. Std_Logic 与Bit 类型之间。
2. Std_Logic_Vector 与Bit_Vector 类型之间。
3. Std_ULogic 与Bit 类型之间。
4. Std_ULogic_Vector 与Bit_Vector 类型之间。

```
1 FUNCTION To_Bit (s: Std_ULogic; xmap: Bit := '0') RETURN Bit;
2 FUNCTION To_StdULogic (b: Bit) RETURN Std_ULogic;
```

求解函数

```
1 FUNCTION largest (total_no: Integer; set: PATTERN) RETURN Real;
```

求解函数和类型转换函数

类型转换函数

1. Std_Logic 与Bit 类型之间。
2. Std_Logic_Vector 与Bit_Vector 类型之间。
3. Std_ULogic 与Bit 类型之间。
4. Std_ULogic_Vector 与Bit_Vector 类型之间。

```
1 FUNCTION To_Bit (s: Std_ULogic; xmap: Bit := '0') RETURN Bit;
2 FUNCTION To_StdULogic (b: Bit) RETURN Std_ULogic;
```

求解函数

```
1 FUNCTION largest (total_no: Integer; set: PATTERN) RETURN Real;
```

求解函数

```
1  FUNCTION largest (total_no: Integer; set: PATTERN) RETURN Real IS
2    VARIABLE: return_value: Real := 0.0;
3  BEGIN
4    FOR k IN set'Range LOOP
5      IF set(k) > return_value THEN
6        return_value := set(k);
7      END IF;
8    END LOOP;
9    RETURN return_value;
10 END largest;
```

- 变量return_value 在每次函数被调用的时候出现，初始值为0.0。
- 当函数终止时，它就消失了。

类型转换函数

```
1  FUNCTION To_BitVector (s: Std_Logic_Vector;xmap: Bit := '0') RETURN Bit_Vector IS
2    ALIAS sv: Std_Logic_Vector (s'Length-1 DOWNTO 0) IS s;
3    VARIABLE result: Bit_Vector (s'Length-1 DOWNTO 0);
4  BEGIN
5    FOR i IN result'Range LOOP
6      CASE sv(i) IS
7        WHEN '0'|'L' => result(i) := '0';
8        WHEN '1'|'H' => result(i) := '1';
9        WHEN OTHERS => result(i) := xmap;
10     END CASE;
11   END LOOP;
12   RETURN result;
13 END To_BitVector;
```

ALIAS

类型转换函数

```
1  FUNCTION To_BitVector (s: Std_Logic_Vector;xmap: Bit := '0') RETURN Bit_Vector IS
2    ALIAS sv: Std_Logic_Vector (s'Length-1 DOWNTO 0) IS s;
3    VARIABLE result: Bit_Vector (s'Length-1 DOWNTO 0);
4  BEGIN
5    FOR i IN result'Range LOOP
6      CASE sv(i) IS
7        WHEN '0'|'L' => result(i) := '0';
8        WHEN '1'|'H' => result(i) := '1';
9        WHEN OTHERS => result(i) := xmap;
10     END CASE;
11   END LOOP;
12   RETURN result;
13 END To_BitVector;
```

ALIAS

- ALIAS 别名语句通常在ENTITY、ARCHITECTURE、PROCESS、PACKAGE、程序包体和子程序的声明部分作为声明语句出现。
- ALIAS 为已经声明的对象再次声明一个“别名”。

```
1 SIGNAL address: Bit_Vector(11 DOWNTO 0);
2 ALIAS page_addr: Bit_Vector(3 DOWNTO 0) IS address(11 DOWNTO 8);
3 ALIAS interior_addr: Bit_Vector(7 DOWNTO 0) IS address(7 DOWNTO 0);
```

- 在信号对象的赋值操作中使用别名是很方便的。
- 但对于常量对象，则要注意由于不能对常量赋值，所以也不能对常量的别名赋值。
- VHDL'87 只允许为对象声明别名，VHDL'93 则放宽了限制，允许为子程序和数据类型等项目声明别名。

ALIAS

- ALIAS 别名语句通常在ENTITY、ARCHITECTURE、PROCESS、PACKAGE、程序包体和子程序的声明部分作为声明语句出现。
- ALIAS 为已经声明的对象再次声明一个“别名”。

```
1 SIGNAL address: Bit_Vector(11 DOWNTO 0);
2 ALIAS page_addr: Bit_Vector(3 DOWNTO 0) IS address(11 DOWNTO 8);
3 ALIAS interior_addr: Bit_Vector(7 DOWNTO 0) IS address(7 DOWNTO 0);
```

- 在信号对象的赋值操作中使用别名是很方便的。
- 但对于常量对象，则要注意由于不能对常量赋值，所以也不能对常量的别名赋值。
- VHDL'87 只允许为对象声明别名，VHDL'93 则放宽了限制，允许为子程序和数据类型等项目声明别名。

ALIAS

- ALIAS 别名语句通常在ENTITY、ARCHITECTURE、PROCESS、PACKAGE、程序包体和子程序的声明部分作为声明语句出现。
- ALIAS 为已经声明的对象再次声明一个“别名”。

```
1 SIGNAL address: Bit_Vector(11 DOWNTO 0);
2 ALIAS page_addr: Bit_Vector(3 DOWNTO 0) IS address(11 DOWNTO 8);
3 ALIAS interior_addr: Bit_Vector(7 DOWNTO 0) IS address(7 DOWNTO 0);
```

- 在信号对象的赋值操作中使用别名是很方便的。
- 但对于常量对象，则要注意由于不能对常量赋值，所以也不能对常量的别名赋值。
- VHDL'87 只允许为对象声明别名，VHDL'93 则放宽了限制，允许为子程序和数据类型等项目声明别名。

② 过程

当设计者自定义了某种运算，而这种运算将产生多个返回值或多个结果时，通常使用过程来描述这种运算。

- 当设计者需要使用已定义好的运算时，可以使用过程调用语句，并用实际参数代替定义运算时声明的形式参数，即可完成这种运算。
- 可以在一个结构体甚至一个进程语句中多次调用这种运算。
- 定义一种运算实际上就是声明一个过程。

```
PROCEDURE 过程名 (形参表) IS
    { 声明语句 }
BEGIN
    { 顺序语句 }
END [ PROCEDURE ] [ 过程名 ];
```

② 过程

当设计者自定义了某种运算，而这种运算将产生多个返回值或多个结果时，通常使用过程来描述这种运算。

- 当设计者需要使用已定义好的运算时，可以使用过程调用语句，并用实际参数代替定义运算时声明的形式参数，即可完成这种运算。
- 可以在一个结构体甚至一个进程语句中多次调用这种运算。
- 定义一种运算实际上就是声明一个过程。

```
PROCEDURE 过程名（形参表）IS
    { 声明语句 }
BEGIN
    { 顺序语句 }
END [ PROCEDURE ] [ 过程名 ];
```

② 过程

当设计者自定义了某种运算，而这种运算将产生多个返回值或多个结果时，通常使用过程来描述这种运算。

- 当设计者需要使用已定义好的运算时，可以使用过程调用语句，并用实际参数代替定义运算时声明的形式参数，即可完成这种运算。
- 可以在一个结构体甚至一个进程语句中多次调用这种运算。
- 定义一种运算实际上就是声明一个过程。

```
PROCEDURE 过程名 (形参表) IS
    { 声明语句 }
BEGIN
    { 顺序语句 }
END [ PROCEDURE ] [ 过程名 ];
```

过程

- 在声明过程的形参表中，形式参数可以是常量、信号和变量，对象模式可以是IN、OUT 或INOUT。
- 如果没有声明形式参数的对象模式，则其缺省模式为IN；
- 如果没有声明形式参数的对象种类，则其缺省种类为：IN 模式的形式参数是常量，OUT 和INOUT 模式的形式参数是变量。
- 对于并行调用语句所调用的过程，由于 VHDL'87 的限制，形式参数不能为变量。
- 在声明语句部分，可以声明过程内部所需的数据类型和局部对象，但不能是信号。
- 过程中的语句都是顺序语句，如果顺序过程调用语句所在的进程没有敏感信号表，则在所调用的过程中可以包含WAIT 语句，否则不能包含WAIT 语句。
- 不正确的使用过程，可能导致出现意想不到的结果。

过程

- 在声明过程的形参表中，形式参数可以是常量、信号和变量，对象模式可以是IN、OUT 或INOUT。
- 如果没有声明形式参数的对象模式，则其缺省模式为IN；
- 如果没有声明形式参数的对象种类，则其缺省种类为：IN 模式的形式参数是常量，OUT 和INOUT 模式的形式参数是变量。
- 对于并行调用语句所调用的过程，由于 VHDL'87 的限制，形式参数不能为变量。
- 在声明语句部分，可以声明过程内部所需的数据类型和局部对象，但不能是信号。
- 过程中的语句都是顺序语句，如果顺序过程调用语句所在的进程没有敏感信号表，则在所调用的过程中可以包含WAIT 语句，否则不能包含WAIT 语句。
- 不正确的使用过程，可能导致出现意想不到的结果。

过程

- 在声明过程的形参表中，形式参数可以是常量、信号和变量，对象模式可以是IN、OUT 或INOUT。
- 如果没有声明形式参数的对象模式，则其缺省模式为IN；
- 如果没有声明形式参数的对象种类，则其缺省种类为：IN 模式的形式参数是常量，OUT 和INOUT 模式的形式参数是变量。
- 对于并行调用语句所调用的过程，由于 VHDL'87 的限制，形式参数不能为变量。
- 在声明语句部分，可以声明过程内部所需的数据类型和局部对象，但不能是信号。
- 过程中的语句都是顺序语句，如果顺序过程调用语句所在的进程没有敏感信号表，则在所调用的过程中可以包含WAIT 语句，否则不能包含WAIT 语句。
- 不正确的使用过程，可能导致出现意想不到的结果。

过程

- 在声明过程的形参表中，形式参数可以是常量、信号和变量，对象模式可以是IN、OUT 或INOUT。
- 如果没有声明形式参数的对象模式，则其缺省模式为IN；
- 如果没有声明形式参数的对象种类，则其缺省种类为：IN 模式的形式参数是常量，OUT 和INOUT 模式的形式参数是变量。
- 对于并行调用语句所调用的过程，由于 VHDL'87 的限制，形式参数不能为变量。
- 在声明语句部分，可以声明过程内部所需的数据类型和局部对象，但不能是信号。
- 过程中的语句都是顺序语句，如果顺序过程调用语句所在的进程没有敏感信号表，则在所调用的过程中可以包含WAIT 语句，否则不能包含WAIT 语句。
- 不正确的使用过程，可能导致出现意想不到的结果。

过程

- 在声明过程的形参表中，形式参数可以是常量、信号和变量，对象模式可以是IN、OUT 或INOUT。
- 如果没有声明形式参数的对象模式，则其缺省模式为IN；
- 如果没有声明形式参数的对象种类，则其缺省种类为：IN 模式的形式参数是常量，OUT 和INOUT 模式的形式参数是变量。
- 对于并行调用语句所调用的过程，由于 VHDL'87 的限制，形式参数不能为变量。
- 在声明语句部分，可以声明过程内部所需的数据类型和局部对象，但不能是信号。
- 过程中的语句都是顺序语句，如果顺序过程调用语句所在的进程没有敏感信号表，则在所调用的过程中可以包含WAIT 语句，否则不能包含WAIT 语句。
- 不正确的使用过程，可能导致出现意想不到的结果。

过程

- 在声明过程的形参表中，形式参数可以是常量、信号和变量，对象模式可以是IN、OUT 或INOUT。
- 如果没有声明形式参数的对象模式，则其缺省模式为IN；
- 如果没有声明形式参数的对象种类，则其缺省种类为：IN 模式的形式参数是常量，OUT 和INOUT 模式的形式参数是变量。
- 对于并行调用语句所调用的过程，由于 VHDL'87 的限制，形式参数不能为变量。
- 在声明语句部分，可以声明过程内部所需的数据类型和局部对象，但不能是信号。
- 过程中的语句都是顺序语句，如果顺序过程调用语句所在的进程没有敏感信号表，则在所调用的过程中可以包含WAIT 语句，否则不能包含WAIT 语句。
- 不正确的使用过程，可能导致出现意想不到的结果。

过程

- 在声明过程的形参表中，形式参数可以是常量、信号和变量，对象模式可以是IN、OUT 或INOUT。
- 如果没有声明形式参数的对象模式，则其缺省模式为IN；
- 如果没有声明形式参数的对象种类，则其缺省种类为：IN 模式的形式参数是常量，OUT 和INOUT 模式的形式参数是变量。
- 对于并行调用语句所调用的过程，由于 VHDL'87 的限制，形式参数不能为变量。
- 在声明语句部分，可以声明过程内部所需的数据类型和局部对象，但不能是信号。
- 过程中的语句都是顺序语句，如果顺序过程调用语句所在的进程没有敏感信号表，则在所调用的过程中可以包含WAIT 语句，否则不能包含WAIT 语句。
- 不正确的使用过程，可能导致出现意想不到的结果。

4 位数值比较器的运算

```
1  LIBRARY IEEE;
2  USE IEEE.Std_Logic_1164.ALL;
3  PROCEDURE comparator(SIGNAL a, b: IN Std_Logic_Vector(3 DOWNTO 0);
4                       SIGNAL a_greater_than_b: OUT Std_Logic;
5                       SIGNAL a_equal_to_b: OUT Std_Logic;
6                       SIGNAL a_less_than_b: OUT Std_Logic) IS
7    VARIABLE y: Std_Logic_Vector(2 DOWNTO 0);
8  BEGIN
9    IF a > b THEN
10     y := "100";
11   ELSIF a = b THEN
12     y := "010";
13   ELSIF a < b THEN
14     y := "001";
15   ELSE
16     y := "000";
17   END IF;
18   a_greater_than_b <= y(2);
19   a_equal_to_b <= y(1);
20   a_less_than_b <= y(0);
21 END PROCEDURE comparator;
```

```
1  comparator(x, y, x_g_y, x_e_y, x_l_y);
```

4 位数值比较器的运算

```
1  LIBRARY IEEE;
2  USE IEEE.Std_Logic_1164.ALL;
3  PROCEDURE comparator(SIGNAL a, b: IN Std_Logic_Vector(3 DOWNTO 0);
4                       SIGNAL a_greater_than_b: OUT Std_Logic;
5                       SIGNAL a_equal_to_b: OUT Std_Logic;
6                       SIGNAL a_less_than_b: OUT Std_Logic) IS
7    VARIABLE y: Std_Logic_Vector(2 DOWNTO 0);
8  BEGIN
9    IF a > b THEN
10     y := "100";
11   ELSIF a = b THEN
12     y := "010";
13   ELSIF a < b THEN
14     y := "001";
15   ELSE
16     y := "000";
17   END IF;
18   a_greater_than_b <= y(2);
19   a_equal_to_b <= y(1);
20   a_less_than_b <= y(0);
21 END PROCEDURE comparator;
```

```
1  comparator(x, y, x_g_y, x_e_y, x_l_y);
```

函数和过程

- 函数被表达式所调用，只返回一个值；而过程则被过程调用语句启动，可以返回多个结果。
- 函数的形式参数只能是IN 模式；而过程的形式参数可以是IN、OUT 或者INOUT 模式。
- 函数的形式参数只允许使用常量和信号类的对象；而过程的形式参数不仅允许使用常量和信号类的对象，还允许使用变量类的对象。
- 如果在形式参数中没有声明对象的种类，则对于函数而言，默认的对象种类是常量；对于过程，IN 模式的形式参数默认对象种类是常量，OUT 和INOUT 模式的形式参数默认对象种类是变量。
- 在函数体中不允许出现WAIT 语句和信号赋值语句；而过程则无此限制。

函数和过程

- 函数被表达式所调用，只返回一个值；而过程则被过程调用语句启动，可以返回多个结果。
- 函数的形式参数只能是IN 模式；而过程的形式参数可以是IN、OUT 或者INOUT 模式。
- 函数的形式参数只允许使用常量和信号类的对象；而过程的形式参数不仅允许使用常量和信号类的对象，还允许使用变量类的对象。
- 如果在形式参数中没有声明对象的种类，则对于函数而言，默认的对象种类是常量；对于过程，IN 模式的形式参数默认对象种类是常量，OUT 和INOUT 模式的形式参数默认对象种类是变量。
- 在函数体中不允许出现WAIT 语句和信号赋值语句；而过程则无此限制。

函数和过程

- 函数被表达式所调用，只返回一个值；而过程则被过程调用语句启动，可以返回多个结果。
- 函数的形式参数只能是IN 模式；而过程的形式参数可以是IN、OUT 或者INOUT 模式。
- 函数的形式参数只允许使用常量和信号类的对象；而过程的形式参数不仅允许使用常量和信号类的对象，还允许使用变量类的对象。
- 如果在形式参数中没有声明对象的种类，则对于函数而言，默认的对象种类是常量；对于过程，IN 模式的形式参数默认对象种类是常量，OUT 和INOUT 模式的形式参数默认对象种类是变量。
- 在函数体中不允许出现WAIT 语句和信号赋值语句；而过程则无此限制。

函数和过程

- 函数被表达式所调用，只返回一个值；而过程则被过程调用语句启动，可以返回多个结果。
- 函数的形式参数只能是IN 模式；而过程的形式参数可以是IN、OUT 或者INOUT 模式。
- 函数的形式参数只允许使用常量和信号类的对象；而过程的形式参数不仅允许使用常量和信号类的对象，还允许使用变量类的对象。
- 如果在形式参数中没有声明对象的种类，则对于函数而言，默认的对象种类是常量；对于过程，IN 模式的形式参数默认对象种类是常量，OUT 和INOUT 模式的形式参数默认对象种类是变量。
- 在函数体中不允许出现WAIT 语句和信号赋值语句；而过程则无此限制。

函数和过程

- 函数被表达式所调用，只返回一个值；而过程则被过程调用语句启动，可以返回多个结果。
- 函数的形式参数只能是IN 模式；而过程的形式参数可以是IN、OUT 或者INOUT 模式。
- 函数的形式参数只允许使用常量和信号类的对象；而过程的形式参数不仅允许使用常量和信号类的对象，还允许使用变量类的对象。
- 如果在形式参数中没有声明对象的种类，则对于函数而言，默认的对象种类是常量；对于过程，IN 模式的形式参数默认对象种类是常量，OUT 和INOUT 模式的形式参数默认对象种类是变量。
- 在函数体中不允许出现WAIT 语句和信号赋值语句；而过程则无此限制。

内容提要

主要内容

1. 程序包以及编译好的设计单元是如何存储在设计库中的。
2. 存储在不同库中的设计单元的内容是如何由几个设计单元共享的。

共享资源

- 当设计者想让某些设计项目（比如数据类型、对象、子程序、元件声明和属性等）成为共享资源能够被其他设计实体使用时，可以利用 VHDL 提供的程序包机制。
  - VHDL 的程序包、实体和结构体都被集成在设计库中。

使用程序包共享资源的方法：

```
LIBRARY { 设计库名 };
USE 设计库名. 程序包名.{ 标识符 }|ALL;
```

```
1 LIBRARY IEEE;
2 USE IEEE.Std_Logic_1164.ALL;
3 USE IEEE.Std_Logic_Arith.ALL;
```

共享资源

- 当设计者想让某些设计项目（比如数据类型、对象、子程序、元件声明和属性等）成为共享资源能够被其他设计实体使用时，可以利用 VHDL 提供的程序包机制。
- VHDL 的程序包、实体和结构体都被集成在设计库中。

使用程序包共享资源的方法：

```
LIBRARY { 设计库名 };
USE 设计库名. 程序包名.{ 标识符 }|ALL;
```

```
1 LIBRARY IEEE;
2 USE IEEE.Std_Logic_1164.ALL;
3 USE IEEE.Std_Logic_Arith.ALL;
```

共享资源

- 当设计者想让某些设计项目（比如数据类型、对象、子程序、元件声明和属性等）成为共享资源能够被其他设计实体使用时，可以利用 VHDL 提供的程序包机制。
- VHDL 的程序包、实体和结构体都被集成在设计库中。

使用程序包共享资源的方法：

```
LIBRARY { 设计库名 };
USE 设计库名. 程序包名.{ 标识符 }|ALL;
```

```
1 LIBRARY IEEE;
2 USE IEEE.Std_Logic_1164.ALL;
3 USE IEEE.Std_Logic_Arith.ALL;
```

共享资源

- 当设计者想让某些设计项目（比如数据类型、对象、子程序、元件声明和属性等）成为共享资源能够被其他设计实体使用时，可以利用 VHDL 提供的程序包机制。
- VHDL 的程序包、实体和结构体都被集成在设计库中。

使用程序包共享资源的方法：

```
LIBRARY { 设计库名 };
USE 设计库名. 程序包名.{ 标识符 }|ALL;
```

```
1 LIBRARY IEEE;
2 USE IEEE.Std_Logic_1164.ALL;
3 USE IEEE.Std_Logic_Arith.ALL;
```

共享资源

- 当设计者想让某些设计项目（比如数据类型、对象、子程序、元件声明和属性等）成为共享资源能够被其他设计实体使用时，可以利用 VHDL 提供的程序包机制。
- VHDL 的程序包、实体和结构体都被集成在设计库中。

使用程序包共享资源的方法：

```
LIBRARY { 设计库名 };
USE 设计库名. 程序包名.{ 标识符 }|ALL;
```

```
1 LIBRARY IEEE;
2 USE IEEE.Std_Logic_1164.ALL;
3 USE IEEE.Std_Logic_Arith.ALL;
```

程序包

- 程序包由**程序包声明**和**程序包体**两部分组成。
- 这两部分可以作为独立的库单元，分别编译并插入设计库中。
- 程序包声明属于设计库的**初级单元**，而程序包体则属于设计库的**次级单元**。

```
PACKAGE 程序包名 IS
    { 程序包声明 }
END [ PACKAGE ][ 程序包名 ];
PACKAGE BODY 程序包名 IS
    { 程序包体声明 }
END [ PACKAGE BODY ][ 程序包名 ];
```

程序包

- 程序包由**程序包声明**和**程序包体**两部分组成。
- 这两部分可以作为独立的库单元，分别编译并插入设计库中。
- 程序包声明属于设计库的**初级单元**，而程序包体则属于设计库的**次级单元**。

```
PACKAGE 程序包名 IS
    { 程序包声明 }
END [ PACKAGE ][ 程序包名 ];
PACKAGE BODY 程序包名 IS
    { 程序包体声明 }
END [ PACKAGE BODY ][ 程序包名 ];
```

程序包声明

```
PACKAGE 程序包名 IS
    { 程序包声明 }
END [ PACKAGE ][ 程序包名 ];
```

- 程序包声明用来声明程序包中包含的共享资源，比如数据类型、对象、子程序、元件声明和属性等，但不能是信号。
- 在程序包声明中声明的标识符是共享资源，它们在程序包之外是可见的。

```
1  PACKAGE synth_pack IS
2    CONSTANT low2high: TIME :=20ns;
3    TYPE alu_op IS (add,sub,mul,div,eql);
4    ATTRIBUTE pipeline: Boolean;
5    TYPE mvl IS ('U','0','1','Z');
6    TYPE mvl_vector IS ARRAY (NATURAL range <>) OF mvl;
7    SUBTYPE my_alu_op IS alu_op RANGE add to div;
8    COMPONENT nand2
9      PORT(a,b:IN mvl; c:OUT mvl);
10   END COMPONENT;
11 END synth_pack;
```

程序包声明

PACKAGE 程序包名 IS
    { 程序包声明 }
END [PACKAGE][程序包名];

- 程序包声明用来声明程序包中包含的共享资源，比如数据类型、对象、子程序、元件声明和属性等，但不能是信号。
- 在程序包声明中声明的标识符是共享资源，它们在程序包之外是可见的。

```
1  PACKAGE synth_pack IS
2    CONSTANT low2high: TIME :=20ns;
3    TYPE alu_op IS (add,sub,mul,div,eql);
4    ATTRIBUTE pipeline: Boolean;
5    TYPE mvl IS ('U','0','1','Z');
6    TYPE mvl_vector IS ARRAY (NATURAL range <>) OF mvl;
7    SUBTYPE my_alu_op IS alu_op RANGE add to div;
8    COMPONENT nand2
9      PORT(a,b:IN mvl; c:OUT mvl);
10   END COMPONENT;
11 END synth_pack;
```

程序包体

```
1 USE work.synth_pack.ALL;
2 PACKAGE program_pack IS
3   CONSTANT prop_delay: TIME;
4   FUNCTION "and" (l,r: mvl) RETURN mvl;
5   PROCEDURE load (SIGNAL array_name: INOUT mvl_vector;
6     start_bit,stop_bit,int_value: IN Integer);
7 END PACKAGE program_pack;
```

```
1 PACKAGE BODY program_pack IS
2   USE work.tables.ALL;
3   CONSTANT prop_delay: TIME := 15ns;
4   FUNCTION "and" (l,r: mvl) RETURN mvl;
5   BEGIN
6     RETURN table_end(l,r);
7   END "and";
8   PROCEDURE load (SIGNAL array_name: INOUT mvl_vector;
9     start_bit,stop_bit,int_value: IN Integer) IS
10  BEGIN
11    ... ...
12  END load;
13 END program_pack;
```

程序包体

```
1 USE work.synth_pack.ALL;
2 PACKAGE program_pack IS
3   CONSTANT prop_delay: TIME;
4   FUNCTION "and" (l,r: mvl) RETURN mvl;
5   PROCEDURE load (SIGNAL array_name: INOUT mvl_vector;
6     start_bit,stop_bit,int_value: IN Integer);
7 END PACKAGE program_pack;
```

- 当程序包声明中包含子程序声明或者延迟常数声明时，必须要有一个程序包体。
- 程序包体主要包含了在程序包声明中的子程序的行为和延迟常数的值。
- 所谓延缓常量，是指在程序包声明中描述了该常量的名称和类型，但没有指定该常量的具体值。
- 在程序包体中声明的标识符，在程序包之外是不可见的，保护知识产权。

```
1 PACKAGE BODY program_pack IS
```

程序包体

```
1  USE work.synth_pack.ALL;
2  PACKAGE program_pack IS
3    CONSTANT prop_delay: TIME;
4    FUNCTION "and" (l,r: mvl) RETURN mvl;
5    PROCEDURE load (SIGNAL array_name: INOUT mvl_vector;
6      start_bit,stop_bit,int_value: IN Integer);
7  END PACKAGE program_pack;
```

```
1  PACKAGE BODY program_pack IS
2    USE work.tables.ALL;
3    CONSTANT prop_delay: TIME := 15ns;
4    FUNCTION "and" (l,r: mvl) RETURN mvl;
5    BEGIN
6      RETURN table_end(l,r);
7    END "and";
8    PROCEDURE load (SIGNAL array_name: INOUT mvl_vector;
9      start_bit,stop_bit,int_value: IN Integer) IS
10   BEGIN
11     ... ...
12   END load;
13 END program_pack;
```

程序包声明和程序包体

- 程序包声明用来声明程序包中包含的共享资源，比如数据类型、对象、子程序、元件声明和属性等，但不能是信号；而程序包体中则包含对延缓常量赋值和对子程序的描述。
- 当程序包声明中不包含延缓常量和子程序时，则不需要程序包体。
- 在程序包声明中声明的标识符是共享资源，它们在程序包之外是可见的（公共声明）；而程序包体声明中声明的标识符，在程序包之外是不可见的（私有声明）。
- 一个程序包声明只能有一个程序包体，并且两者的名字要相同。

程序包声明和程序包体

- 程序包声明用来声明程序包中包含的共享资源，比如数据类型、对象、子程序、元件声明和属性等，但不能是信号；而程序包体中则包含对延缓常量赋值和对子程序的描述。
- 当程序包声明中不包含延缓常量和子程序时，则不需要程序包体。
- 在程序包声明中声明的标识符是共享资源，它们在程序包之外是可见的（公共声明）；而程序包体声明中声明的标识符，在程序包之外是不可见的（私有声明）。
- 一个程序包声明只能有一个程序包体，并且两者的名字要相同。

程序包声明和程序包体

- 程序包声明用来声明程序包中包含的共享资源，比如数据类型、对象、子程序、元件声明和属性等，但不能是信号；而程序包体中则包含对延缓常量赋值和对子程序的描述。
- 当程序包声明中不包含延缓常量和子程序时，则不需要程序包体。
- 在程序包声明中声明的标识符是共享资源，它们在程序包之外是可见的（公共声明）；而程序包体声明中声明的标识符，在程序包之外是不可见的（私有声明）。
  实体声明中声明的条目对于其他设计单元来说是可见的，而结构体中的声明在其范围外是不可见的。
- 一个程序包声明只能有一个程序包体，并且两者的名字要相同。

程序包声明和程序包体

- 程序包声明用来声明程序包中包含的共享资源，比如数据类型、对象、子程序、元件声明和属性等，但不能是信号；而程序包体中则包含对延缓常量赋值和对子程序的描述。
- 当程序包声明中不包含延缓常量和子程序时，则不需要程序包体。
- 在程序包声明中声明的标识符是共享资源，它们在程序包之外是可见的（公共声明）；而程序包体声明中声明的标识符，在程序包之外是不可见的（私有声明）。
- 一个程序包声明只能有一个程序包体，并且两者的名字要相同。一个实体可以有多个不同名字的结构体。

设计文件

- 设计文件是一个包含 VHDL 源代码的 ASCII 文件，它可以包含一个或多个**设计单元**。
- 每一个设计单元也可以被放在一个独立的文件中。
- 设计单元的形式有实体声明、结构体、配置声明、程序包声明、程序包体。

编译过程

设计文件 ⟹ VHDL 分析器 ⟹ 中间形式 ⟹ 设计库（工作库）

1. 设计文件被一个 VHDL 分析器处理。
2. 该分析器在验证了源代码语法和语义的正确性后，将设计文件中的每一个设计单元编译成某种中间形式。
3. 这些中间形式存储在一个被指定为工作库的设计库中。

设计文件

- 设计文件是一个包含 VHDL 源代码的 ASCII 文件，它可以包含一个或多个**设计单元**。
- 每一个设计单元也可以被放在一个独立的文件中。
- 设计单元的形式有实体声明、结构体、配置声明、程序包声明、程序包体。

编译过程

设计文件 ⟹ VHDL 分析器 ⟹ 中间形式 ⟹ 设计库（工作库）

1. 设计文件被一个 VHDL 分析器处理。
2. 该分析器在验证了源代码语法和语义的正确性后，将设计文件中的每一个设计单元编译成某种中间形式。
3. 这些中间形式存储在一个被指定为工作库的设计库中。

设计库

- 一个编译好的设计单元被存储在一个设计库中，设计库是主机环境文件系统中的一块存储区域。
- 每个设计库有一个逻辑名，在 VHDL 描述里，根据这个逻辑名来引用相应的设计库。
- 逻辑名和物理存储名之间的映射是由主机环境来进行的。
- 有一个设计库，VHDL 将其逻辑名预定义为STD，这个库包含了两个预定义程序包STANDARD 和TEXTIO 编译后的文件。
- 一定要有一个设计库被指定为工作库，其逻辑名为WORK，当一个设计文件被编译后，设计文件中设计单元的中间形式都被存放在工作库中。
- 因此，在编译开始之前，逻辑名WORK 必须指向某一个设计库。
- VHDL 源代码出现在一个设计文件中，一个设计文件可以包含一个或多个设计单元，设计单元可以进一步分类为初级单元和次级单元。
- 初级单元不直接依赖于其他设计单元，包括实体声明、程序包声明和配置声明。
- 次级单元总是依赖于一个初级单元，且不允许将在它们内部声明的条目提供给本设计单元以外的地方（不能被其他设计单元引用），包括结构体和程序包体。

预定义程序包

- 在所有设计实体的开头，都隐含有下面两条语句：

```
1 LIBRARY Work, Std;
2 USE Std.Standard.ALL;
```

标准（STDANDARD）程序包中的所有资源对设计实体都是可见的。

- 如果要使用TEXTIO 程序包中的任何共享资源，则必须使用USE 子句：

```
1 USE Std.Textio.ALL;
```

- STANDARD 程序包声明了若干数据类型、子类型和函数；TEXTIO 程序包则声明了支持 ASCII I/O 操作的若干数据类型、子类型、过程和文件。
- 隐含声明的另一个设计库WORK 是设计者的现行工作库，用来放置当前设计实体和设计者自定义程序包。

预定义程序包

- 在所有设计实体的开头，都隐含有下面两条语句：

```
1 LIBRARY Work, Std;
2 USE Std.Standard.ALL;
```

标准（STDANDARD）程序包中的所有资源对设计实体都是可见的。

- 如果要使用TEXTIO 程序包中的任何共享资源，则必须使用USE 子句：

```
1 USE Std.Textio.ALL;
```

- STANDARD 程序包声明了若干数据类型、子类型和函数；TEXTIO 程序包则声明了支持 ASCII I/O 操作的若干数据类型、子类型、过程和文件。
- 隐含声明的另一个设计库WORK 是设计者的现行工作库，用来放置当前设计实体和设计者自定义程序包。

预定义程序包

- 在所有设计实体的开头，都隐含有下面两条语句：

```
1 LIBRARY Work, Std;
2 USE Std.Standard.ALL;
```

标准（STDANDARD）程序包中的所有资源对设计实体都是可见的。

- 如果要使用TEXTIO 程序包中的任何共享资源，则必须使用USE 子句：

```
1 USE Std.Textio.ALL;
```

- STANDARD 程序包声明了若干数据类型、子类型和函数；TEXTIO 程序包则声明了支持 ASCII I/O 操作的若干数据类型、子类型、过程和文件。
- 隐含声明的另一个设计库WORK 是设计者的现行工作库，用来放置当前设计实体和设计者自定义程序包。

预定义程序包

- 在所有设计实体的开头，都隐含有下面两条语句：

```
1 LIBRARY Work, Std;
2 USE Std.Standard.ALL;
```

标准（STDANDARD）程序包中的所有资源对设计实体都是可见的。

- 如果要使用TEXTIO 程序包中的任何共享资源，则必须使用USE 子句：

```
1 USE Std.Textio.ALL;
```

- STANDARD 程序包声明了若干数据类型、子类型和函数；TEXTIO 程序包则声明了支持 ASCII I/O 操作的若干数据类型、子类型、过程和文件。
- 隐含声明的另一个设计库WORK 是设计者的现行工作库，用来放置当前设计实体和设计者自定义程序包。

预定义程序包

- Std_Logic_1164 程序包已经预先在 IEEE VHDL 库中编译，声明了若干常用的数据类型、子类型和函数。
- 在一些 VHDL 仿真器中，Numeric_Std 程序包和Numeric_Bit 程序包也已经预先编译在 IEEE VHDL 库中，声明了用于综合的数值类型和算术函数。
- 如果要使用这些程序包中的任何共享资源，则必须先用USE 子句来使他们对设计实体可见。

```
1 LIBRARY IEEE;
2 USE IEEE.Std_Logic_1164.ALL;
```

预定义程序包

- Std_Logic_1164 程序包已经预先在 IEEE VHDL 库中编译，声明了若干常用的数据类型、子类型和函数。
- 在一些 VHDL 仿真器中，Numeric_Std 程序包和Numeric_Bit 程序包也已经预先编译在 IEEE VHDL 库中，声明了用于综合的数值类型和算术函数。
- 如果要使用这些程序包中的任何共享资源，则必须先用USE 子句来使他们对设计实体可见。

```
1 LIBRARY IEEE;
2 USE IEEE.Std_Logic_1164.ALL;
```

预定义程序包

- Std_Logic_1164 程序包已经预先在 IEEE VHDL 库中编译，声明了若干常用的数据类型、子类型和函数。
- 在一些 VHDL 仿真器中，Numeric_Std 程序包和Numeric_Bit 程序包也已经预先编译在 IEEE VHDL 库中，声明了用于综合的数值类型和算术函数。
- 如果要使用这些程序包中的任何共享资源，则必须先用USE 子句来使他们对设计实体可见。

```
1 LIBRARY IEEE;
2 USE IEEE.Std_Logic_1164.ALL;
```

3-8 译码器 74LS138

- 74LS138 三个输入使能端g1、$g2a\_n$、$g2b\_n$。
  1. g1 高电平有效
  2. $g2a\_n$ 低电平有效
  3. $g2b\_n$ 低电平有效
- 只有在所有使能端都为有效电平（$g1\,g2a\_n\,g2b\_n = 100$）时，才进行相应译码，输出信号低电平有效；否则，译码器停止译码，输出无效电平（高电平）。

3-8 译码器 74LS138

$g1$	$g2a\_n$	$g2b\_n$	c	b	a	y_n_7	y_n_6	y_n_5	y_n_4	y_n_3	y_n_2	y_n_1	y_n_0
0	x	x	x	x	x	1	1	1	1	1	1	1	1
x	1	x	x	x	x	1	1	1	1	1	1	1	1
x	x	1	x	x	x	1	1	1	1	1	1	1	1
1	0	0	0	0	0	1	1	1	1	1	1	1	0
			0	0	1	1	1	1	1	1	1	0	1
			0	1	0	1	1	1	1	1	0	1	1
			0	1	1	1	1	1	1	0	1	1	1
			1	0	0	1	1	1	0	1	1	1	1
			1	0	1	1	1	0	1	1	1	1	1
			1	1	0	1	0	1	1	1	1	1	1
			1	1	1	0	1	1	1	1	1	1	1

```
1  LIBRARY IEEE;
2  USE IEEE.Std_Logic_1164.ALL;
3  ENTITY decoder_3_8 IS
4    PORT(g1, g2a_n, g2b_n, a, b, c: IN Std_Logic;
5         y_n: OUT Std_Logic_Vector(7 DOWNTO 0));
6  END decoder_3_8;
7  ARCHITECTURE behavl_decoder OF decoder_3_8 IS
8  BEGIN
9    PROCESS(g1, g2a_n, g2b_n, a, b, c)
10     VARIABLE temporary: Std_Logic_Vector(2 DOWNTO 0);
11   BEGIN
12     temporary := g1 & g2a_n & g2b_n;
13     IF temporary = "100" THEN
14       temporary := c & b & a;
15       CASE temporary IS
16         WHEN "000" => y_n <= B"1111_1110";
17         WHEN "001" => y_n <= B"1111_1101";
18         WHEN "010" => y_n <= B"1111_1011";
19         WHEN "011" => y_n <= B"1111_0111";
20         WHEN "100" => y_n <= B"1110_1111";
21         WHEN "101" => y_n <= B"1101_1111";
22         WHEN "110" => y_n <= B"1011_1111";
23         WHEN "111" => y_n <= B"0111_1111";
24         WHEN OTHERS => y_n <= (OTHERS => '1');
25       END CASE;
26     ELSE
27       y_n <= (OTHERS => '1');
28     END IF;
29   END PROCESS;
30 END behavl_decoder;
```

3-8 译码器 74LS138

```
1  LIBRARY IEEE;
2  USE IEEE.Std_Logic_1164.ALL;
3  USE IEEE.Std_Logic_Unsigned.ALL;
4  ENTITY decoder_3_8 IS
5    PORT(g1, g2a_n, g2b_n, a, b, c: IN Std_Logic;
6         y_n: OUT Std_Logic_Vector (7 DOWNTO 0));
7  END decoder_3_8;
8  ARCHITECTURE behavl_decoder OF decoder_3_8 IS
9  BEGIN
10   PROCESS(g1, g2a_n, g2b_n, a, b, c)
11     VARIABLE temporary: Std_Logic_Vector (2 DOWNTO 0);
12     VARIABLE y: Std_Logic_Vector (7 DOWNTO 0);
13   BEGIN
14     temporary := g1 & g2a_n & g2b_n;
15     y := (OTHERS => '1');
16     IF temporary = "100" THEN
17       y(conv_integer(c & b & a)) := '0';
18     END IF;
19     y_n <= y;
20   END PROCESS;
21 END behavl_decoder;
```

内容提要

什么是重载？

```
1 FUNCTION count (oranges: Integer) RETURN Integer;
2 FUNCTION count (apples: Bit) RETURN Bit;
```

```
1 count(20)
2 count('1')
```

什么是重载？

```
1 FUNCTION count (oranges: Integer) RETURN Integer;
2 FUNCTION count (apples: Bit) RETURN Bit;
```

```
1 count(20)
2 count('1')
```

什么是重载？

```
1 FUNCTION count (oranges: Integer) RETURN Integer;
2 FUNCTION count (apples: Bit) RETURN Bit;
```

```
1 count(20)
2 count('1')
```

重载

当多个子程序、多个运算符或者多个枚举类型中的值使用同一个名称（标识符）时，就称之为重载。

举例

- 在 VHDL 预定义的标准程序包 STANDARD 中，字符文字'0' 和'1' 同时被声明为枚举类型Bit 和Character 中的值。
- 字符文字'0' 和'1' 还在Std_Logic_1164 程序包中被声明为枚举类型Std_Logic 中的值。
- 子程序重载可以使被声明的同一名称的子程序对不同类型的对象进行操作。
- 运算符重载可以使同一个运算符来对不同类型的对象进行相似的运算。

重载

当多个子程序、多个运算符或者多个枚举类型中的值使用同一个名称（标识符）时，就称之为重载。

举例

- 在 VHDL 预定义的标准程序包 STANDARD 中，字符文字'0' 和'1' 同时被声明为枚举类型Bit 和Character 中的值。
- 字符文字'0' 和'1' 还在Std_Logic_1164 程序包中被声明为枚举类型Std_Logic 中的值。
- 子程序重载可以使被声明的同一名称的子程序对不同类型的对象进行操作。
- 运算符重载可以使同一个运算符来对不同类型的对象进行相似的运算。

重载

当多个子程序、多个运算符或者多个枚举类型中的值使用同一个名称（标识符）时，就称之为重载。

举例

- 在 VHDL 预定义的标准程序包 STANDARD 中，字符文字'0' 和'1' 同时被声明为枚举类型Bit 和Character 中的值。
- 字符文字'0' 和'1' 还在Std_Logic_1164 程序包中被声明为枚举类型Std_Logic 中的值。
- 子程序重载可以使被声明的同一名称的子程序对不同类型的对象进行操作。
- 运算符重载可以使同一个运算符来对不同类型的对象进行相似的运算。

① 子程序重载

1. 声明为同一名称的多个子程序，其操作参数类型不同。
2. 声明为同一名称的多个子程序，其操作参数类型虽然相同，但参数个数不同。

对于声明为同一名称的多个子程序，必须通过其形参表中参数的差异来区分。

```
1  LIBRARY IEEE;
2  USE IEEE.Std_Logic_1164.ALL;
3  USE IEEE.Std_Logic_Arith.ALL;
4  PACKAGE minimum IS
5    FUNCTION min(a, b: Integer) RETURN Integer;
6    FUNCTION min(a, b, c: Integer) RETURN Integer;
7    FUNCTION min(a, b, c, d: Integer) RETURN Integer;
8    FUNCTION min(a, b: Real) RETURN Real;
9    FUNCTION min(a, b, c: Real) RETURN Real;
10   FUNCTION min(a, b, c, d: Real) RETURN Real;
11   FUNCTION min(a, b: Std_Logic_Vector) RETURN Std_Logic_Vector;
12   FUNCTION min(a, b, c: Std_Logic_Vector) RETURN Std_Logic_Vector;
13   FUNCTION min(a, b, c, d: Std_Logic_Vector) RETURN Std_Logic_Vector;
14 END minimum;
```

① 子程序重载

1. 声明为同一名称的多个子程序，其操作参数类型不同。
2. 声明为同一名称的多个子程序，其操作参数类型虽然相同，但参数个数不同。

对于声明为同一名称的多个子程序，必须通过其形参表中参数的差异来区分。

```
1  LIBRARY IEEE;
2  USE IEEE.Std_Logic_1164.ALL;
3  USE IEEE.Std_Logic_Arith.ALL;
4  PACKAGE minimum IS
5    FUNCTION min(a, b: Integer) RETURN Integer;
6    FUNCTION min(a, b, c: Integer) RETURN Integer;
7    FUNCTION min(a, b, c, d: Integer) RETURN Integer;
8    FUNCTION min(a, b: Real) RETURN Real;
9    FUNCTION min(a, b, c: Real) RETURN Real;
10   FUNCTION min(a, b, c, d: Real) RETURN Real;
11   FUNCTION min(a, b: Std_Logic_Vector) RETURN Std_Logic_Vector;
12   FUNCTION min(a, b, c: Std_Logic_Vector) RETURN Std_Logic_Vector;
13   FUNCTION min(a, b, c, d: Std_Logic_Vector) RETURN Std_Logic_Vector;
14 END minimum;
```

不同参数类型的子程序重载

```
1  FUNCTION min(x,y,z: Integer)
2           RETURN Integer IS
3    VARIABLE i: Integer;
4  BEGIN
5    IF x < y THEN
6      i := x;
7    ELSE
8      i := y;
9    END IF;
10   IF z < i THEN
11     i := z;
12   END IF;
13   RETURN i;
14 END min;
```

```
1  FUNCTION min(x,y,z: Std_Logic_Vector)
2           RETURN Std_Logic_Vector IS
3    VARIABLE i: Std_Logic_Vector;
4  BEGIN
5    IF x < y THEN
6      i := x;
7    ELSE
8      i := y;
9    END IF;
10   IF z < i THEN
11     i := z;
12   END IF;
13   RETURN i;
14 END min;
```

不同参数个数的子程序重载

```
1 FUNCTION min(a, b: Integer)
2          RETURN Integer IS
3 BEGIN
4   IF a < b THEN
5     RETURN a;
6   ELSE
7     RETURN b;
8   END IF;
9 END min;
```

```
1  FUNCTION min(a, b, c: Integer)
2             RETURN Integer IS
3    VARIABLE i: Integer;
4  BEGIN
5    IF a < b THEN
6      i := a;
7    ELSE
8      i := b;
9    END IF;
10   IF c < i THEN
11     i := c;
12   END IF;
13   RETURN i;
14 END min;
```

② 运算符重载

- 在 VHDL 中，运算符从广义上讲也是一种函数，因此运算符也可以被重载。
- 例如在 IEEE1076 中，运算符+ 被预定义为用于标量类型中的整数、实数和物理量的加法，如果需要进行Bit_Vector 类型的加法运算，则必须声明一个进行Bit_Vector 类型加法运算的+ 函数，然后再进行重载调用。

```
1  PACKAGE math IS
2    FUNCTION "+"(l, r: Bit_Vector) RETURN Bit_Vector;
3  END math;
4  PACKAGE BODY math IS
5    FUNCTION "+"(l, r: Bit_Vector) RETURN Bit_Vector IS
6      ... ...
7    BEGIN
8      ... ...
9    END FUNCTION;
10 END math;
```

② 运算符重载

- 在 VHDL 中，运算符从广义上讲也是一种函数，因此运算符也可以被重载。
- 例如在 IEEE1076 中，运算符+ 被预定义为用于标量类型中的整数、实数和物理量的加法，如果需要进行Bit_Vector 类型的加法运算，则必须声明一个进行Bit_Vector 类型加法运算的+ 函数，然后再进行重载调用。

```
1  PACKAGE math IS
2    FUNCTION "+"(l, r: Bit_Vector) RETURN Bit_Vector;
3  END math;
4  PACKAGE BODY math IS
5    FUNCTION "+"(l, r: Bit_Vector) RETURN Bit_Vector IS
6      ... ...
7    BEGIN
8      ... ...
9    END FUNCTION;
10 END math;
```

内容提要

多驱动源信号

- VHDL 多个驱动源驱动同一个信号的机制——决断信号。
- 一个决断信号必须具有与之相关联的决断函数。
- 决断信号具有多个驱动源和一个决断函数。

决断信号的声明

- 如果一个信号没有被声明为决断信号，而这个信号却具有多个驱动源，则会出现错误。
- 一个具有多个驱动源的信号必须被声明为决断信号，声明一个决断信号有两种方法。
- 一种方法是在信号声明中直接包含决断函数；

```
SIGNAL multi_driver_sig: decide four_val;
```

- 另一种方法是先声明一个决断子类型，然后用这个子类型声明一个信号，这种方法通常用于多个决断信号使用同一个决断函数的情况。

```
SUBTYPE multi_driver IS decide four_val;
SIGNAL multi_driver_sig: multi_driver;
```

- 使用三态总线会给设计和制造带来诸多麻烦，如果不是绝对必要，尽量不使用决断信号。

决断信号的声明

- 如果一个信号没有被声明为决断信号，而这个信号却具有多个驱动源，则会出现错误。
- 一个具有多个驱动源的信号必须被声明为决断信号，声明一个决断信号有两种方法。
- 一种方法是在信号声明中直接包含决断函数；

```
SIGNAL multi_driver_sig: decide four_val;
```

- 另一种方法是先声明一个决断子类型，然后用这个子类型声明一个信号，这种方法通常用于多个决断信号使用同一个决断函数的情况。

```
SUBTYPE multi_driver IS decide four_val;
SIGNAL multi_driver_sig: multi_driver;
```

- 使用三态总线会给设计和制造带来诸多麻烦，如果不是绝对必要，尽量不使用决断信号。

决断信号的声明

- 如果一个信号没有被声明为决断信号，而这个信号却具有多个驱动源，则会出现错误。
- 一个具有多个驱动源的信号必须被声明为决断信号，声明一个决断信号有两种方法。
- 一种方法是在信号声明中直接包含决断函数；

```
1 SIGNAL multi_driver_sig: decide four_val;
```

- 另一种方法是先声明一个决断子类型，然后用这个子类型声明一个信号，这种方法通常用于多个决断信号使用同一个决断函数的情况。

```
1 SUBTYPE multi_driver IS decide four_val;
2 SIGNAL multi_driver_sig: multi_driver;
```

- 使用三态总线会给设计和制造带来诸多麻烦，如果不是绝对必要，尽量不使用决断信号。

决断信号的声明

- 如果一个信号没有被声明为决断信号，而这个信号却具有多个驱动源，则会出现错误。
- 一个具有多个驱动源的信号必须被声明为决断信号，声明一个决断信号有两种方法。
- 一种方法是在信号声明中直接包含决断函数；

```
1 SIGNAL multi_driver_sig: decide four_val;
```

- 另一种方法是先声明一个决断子类型，然后用这个子类型声明一个信号，这种方法通常用于多个决断信号使用同一个决断函数的情况。

```
1 SUBTYPE multi_driver IS decide four_val;
2 SIGNAL multi_driver_sig: multi_driver;
```

- 使用三态总线会给设计和制造带来诸多麻烦，如果不是绝对必要，尽量不使用决断信号。

决断信号的声明

- 如果一个信号没有被声明为决断信号，而这个信号却具有多个驱动源，则会出现错误。
- 一个具有多个驱动源的信号必须被声明为决断信号，声明一个决断信号有两种方法。
- 一种方法是在信号声明中直接包含决断函数；

```
1 SIGNAL multi_driver_sig: decide four_val;
```

- 另一种方法是先声明一个决断子类型，然后用这个子类型声明一个信号，这种方法通常用于多个决断信号使用同一个决断函数的情况。

```
1 SUBTYPE multi_driver IS decide four_val;
2 SIGNAL multi_driver_sig: multi_driver;
```

- 使用三态总线会给设计和制造带来诸多麻烦，如果不是绝对必要，尽量不使用决断信号。

决断函数

- 决断函数的功能是对多驱动源的信号值的冲突进行仲裁，即对输入的所有驱动值进行判定，挑选一个竞争力最强的值返回，作为决断信号的最终值。
- 输入决断函数的驱动值必须是元素类型与决断信号类型相同的一维非限定性数组，其返回值的类型与决断信号类型相同。
- 决断函数是在对决断信号进行赋值操作的时候，被VHDL模拟器隐含调用的，设计者不能像调用其他函数那样，在表达式中用显式方法来调用决断函数。

```
1 TYPE four_val IS ('X','0','1','Z');
```

- 'X' 表示不确定值，'0' 表示逻辑0值，'1' 表示逻辑1值，'Z' 表示高阻值。
- 为了判定一个four_val 类型的决断信号的最终值，必须先规定four_val 类型中不同信号值的竞争力，当两值竞争时，竞争力强的值胜出。
- 'X' 竞争力最强，'0'、'1' 竞争力中等，'Z' 竞争力最弱。

决断函数

- 决断函数的功能是对多驱动源的信号值的冲突进行仲裁，即对输入的所有驱动值进行判定，挑选一个竞争力最强的值返回，作为决断信号的最终值。
- 输入决断函数的驱动值必须是元素类型与决断信号类型相同的一维非限定性数组，其返回值的类型与决断信号类型相同。
- 决断函数是在对决断信号进行赋值操作的时候，被 VHDL 模拟器隐含调用的，设计者不能像调用其他函数那样，在表达式中用显式方法来调用决断函数。

```
1 TYPE four_val IS ('X','0','1','Z');
```

- 'X' 表示不确定值，'0' 表示逻辑0 值，'1' 表示逻辑1 值，'Z' 表示高阻值。
- 为了判定一个four_val 类型的决断信号的最终值，必须先规定four_val 类型中不同信号值的竞争力，当两值竞争时，竞争力强的值胜出。
- 'X' 竞争力最强，'0'、'1' 竞争力中等，'Z' 竞争力最弱。

标准逻辑

```
1 TYPE Std_ULogic IS ('U','X','0','1','Z','W','L','H','-');
2 TYPE Std_ULogic_Vector IS ARRAY(Natural RANGE <>) OF Std_ULogic;
3 SUBTYPE Std_Logic IS Resolved Std_ULogic;
4 FUNCTION Resolved(S: Std_ULogic_Vector) RETURN Std_ULogic;
```

- 'U' 未初始化值，系统电压建立（上电）时的初始值；
- 'X' 强未知值，强逻辑 0 遭遇强逻辑 1 的结果；
- '0' 强逻辑 0，简称逻辑 0；
- '1' 强逻辑 1，简称逻辑 1；
- 'Z' 高阻值，三态门处于高阻态；
- 'W' 弱未知值，弱逻辑 0 遭遇弱逻辑 1 的结果；
- 'L' 弱逻辑 0，下拉；
- 'H' 弱逻辑 1，上拉；
- '-' 无关，不可能值。

内容提要

为什么要用配置？

1. 有些时候，对同一个实体采用多种描述方式并用其中的任一种来仿真是十分方便的。
   对每一种描述方式指定一个结构体，并用一个配置来绑定所需要的结构体，就可以很容易实现。
2. 可能需要将一个元件和一个实体集合中的某一个联系起来。

配置主要进行绑定

1. 一个结构体和它的实体声明。
2. 一个元件和一个实体。

- 配置语句对实体或者元件的硬件实现不产生影响，只是在仿真阶段指定某种连接关系。
- 综合器将忽略配置语句。
- 利用配置语句可以将复杂的多层次结构描述清晰化，从而增强可读性。

为什么要用配置？

1. 有些时候，对同一个实体采用多种描述方式并用其中的任一种来仿真是十分方便的。
2. 可能需要将一个元件和一个实体集合中的某一个联系起来。
   元件声明可以有它自己的名字，并且端口类属的名字、类型和数量可以与对应的实体不同。

配置主要进行绑定

1. 一个结构体和它的实体声明。
2. 一个元件和一个实体。

- 配置语句对实体或者元件的硬件实现不产生影响，只是在仿真阶段指定某种连接关系。
- 综合器将忽略配置语句。
- 利用配置语句可以将复杂的多层次结构描述清晰化，从而增强可读性。

为什么要用配置？

1. 有些时候，对同一个实体采用多种描述方式并用其中的任一种来仿真是十分方便的。
2. 可能需要将一个元件和一个实体集合中的某一个联系起来。

配置主要进行绑定

1. 一个结构体和它的实体声明。
2. 一个元件和一个实体。

- 配置语句对实体或者元件的硬件实现不产生影响，只是在仿真阶段指定某种连接关系。
- 综合器将忽略配置语句。
- 利用配置语句可以将复杂的多层次结构描述清晰化，从而增强可读性。

为什么要用配置？

1. 有些时候，对同一个实体采用多种描述方式并用其中的任一种来仿真是十分方便的。
2. 可能需要将一个元件和一个实体集合中的某一个联系起来。

配置主要进行绑定

1. 一个结构体和它的实体声明。
2. 一个元件和一个实体。

- 配置语句对实体或者元件的硬件实现不产生影响，只是在仿真阶段指定某种连接关系。
- 综合器将忽略配置语句。
- 利用配置语句可以将复杂的多层次结构描述清晰化，从而增强可读性。

为什么要用配置？

1. 有些时候，对同一个实体采用多种描述方式并用其中的任一种来仿真是十分方便的。
2. 可能需要将一个元件和一个实体集合中的某一个联系起来。

配置主要进行绑定

1. 一个结构体和它的实体声明。
2. 一个元件和一个实体。

- 配置语句对实体或者元件的硬件实现不产生影响，只是在仿真阶段指定某种连接关系。
- 综合器将忽略配置语句。
- 利用配置语句可以将复杂的多层次结构描述清晰化，从而增强可读性。

默认连接

实体与元件

- 如果在当前的Work 设计库中描述了元件；
- 或者在某个设计库中描述了元件，并且在当前实体声明之前用LIBRARY 子句和USE 子句声明了该设计库。

就使得Work 库或者被LIBRARY 子句和USE 子句声明的设计库中的元件，与当前设计实体相连接。

在元件例化时，仿真器会将Work 库或者被声明的设计库中的元件作为模板元件进行例化。

由于这种连接没有使用配置语句，所以称之为默认连接。

实体与结构体

如果为一个设计实体描述了多个结构体，在仿真时仿真器会将最后编译的结构体与实体声明默认连接，即默认的仿真最后一个编译的结构体。

默认连接

实体与元件

- 如果在当前的Work 设计库中描述了元件；
- 或者在某个设计库中描述了元件，并且在当前实体声明之前用LIBRARY 子句和USE 子句声明了该设计库。

就使得Work 库或者被LIBRARY 子句和USE 子句声明的设计库中的元件，与当前设计实体相连接。
在元件例化时，仿真器会将Work 库或者被声明的设计库中的元件作为模板元件进行例化。
由于这种连接没有使用配置语句，所以称之为默认连接。

实体与结构体

如果为一个设计实体描述了多个结构体，在仿真时仿真器会将最后编译的结构体与实体声明默认连接，即默认的仿真最后一个编译的结构体。

默认连接

实体与元件

- 如果在当前的Work 设计库中描述了元件；
- 或者在某个设计库中描述了元件，并且在当前实体声明之前用LIBRARY 子句和USE 子句声明了该设计库。

就使得Work 库或者被LIBRARY 子句和USE 子句声明的设计库中的元件，与当前设计实体相连接。
在元件例化时，仿真器会将Work 库或者被声明的设计库中的元件作为模板元件进行例化。
由于这种连接没有使用配置语句，所以称之为默认连接。

实体与结构体

如果为一个设计实体描述了多个结构体，在仿真时仿真器会将最后编译的结构体与实体声明默认连接，即默认的仿真最后一个编译的结构体。

默认连接

实体与元件

- 如果在当前的Work 设计库中描述了元件；
- 或者在某个设计库中描述了元件，并且在当前实体声明之前用LIBRARY 子句和USE 子句声明了该设计库。

就使得Work 库或者被LIBRARY 子句和USE 子句声明的设计库中的元件，与当前设计实体相连接。
在元件例化时，仿真器会将Work 库或者被声明的设计库中的元件作为模板元件进行例化。
由于这种连接没有使用配置语句，所以称之为默认连接。

实体与结构体

如果为一个设计实体描述了多个结构体，在仿真时仿真器会将最后编译的结构体与实体声明默认连接，即默认的仿真最后一个编译的结构体。

默认配置

当要指定仿真某一个结构体（不一定是最后编译的结构体）时，则需要用配置语句来指定实体声明与某个结构体的连接关系——默认配置。

```
CONFIGURATION 配置名称 OF 实体名称 IS
    FOR 结构体名称
    END FOR;
END 配置名称;
```

```
1 CONFIGURATION config_rtl OF full_adder IS
2   FOR rtl
3   END FOR;
4 END config_rtl;
```

默认配置

当要指定仿真某一个结构体（不一定是最后编译的结构体）时，则需要用配置语句来指定实体声明与某个结构体的连接关系——默认配置。

```
CONFIGURATION 配置名称 OF 实体名称 IS
    FOR 结构体名称
    END FOR;
END 配置名称;
```

```
1 CONFIGURATION config_rtl OF full_adder IS
2   FOR rtl
3   END FOR;
4 END config_rtl;
```

默认配置

当要指定仿真某一个结构体（不一定是最后编译的结构体）时，则需要用配置语句来指定实体声明与某个结构体的连接关系——默认配置。

```
CONFIGURATION 配置名称 OF 实体名称 IS
    FOR 结构体名称
    END FOR;
END 配置名称;
```

```
1 CONFIGURATION config_rtl OF full_adder IS
2   FOR rtl
3   END FOR;
4 END config_rtl;
```

元件配置

如果在某个结构体中做元件例化时不使用默认连接，而需要指定某个元件作为模板元件与当前例化元件相连接时，则应当使用元件配置语句予以指定。

```
CONFIGURATION 配置名称 OF 实体名称 IS
    FOR 结构体名称
        {FOR {例化元件名称 |OTHERS|ALL}: 模板元件名称 | 实体名称
            USE CONFIGURATION 设计库名称. 配置名称 |
            USE ENTITY 设计库名称. 实体名称 (结构体名称);
          [GENERIC MAP(类属映射表)]
          [PORT MAP(端口映射表)]
        END FOR;
    END FOR;
END 配置名称;
```

- 当出现元件例化语句中的映射端口名称与元件声明语句中的端口名称不一致时，需要选用配置端口映射子句。
- 当需要向元件传递参数时，可以在元件例化语句中通过类属映射传递参数，但必须重新编译元件例化语句所在的实体和结构体才可以生效；如果通过元件配置中的类属映射子句传递参数，则不需重新编译元件例化语句所在的实体和结构体。

元件配置

如果在某个结构体中做元件例化时不使用默认连接，而需要指定某个元件作为模板元件与当前例化元件相连接时，则应当使用元件配置语句予以指定。

```
CONFIGURATION 配置名称 OF 实体名称 IS
    FOR 结构体名称
        {FOR {例化元件名称 |OTHERS|ALL}: 模板元件名称 | 实体名称
            USE CONFIGURATION 设计库名称. 配置名称 |
            USE ENTITY 设计库名称. 实体名称 (结构体名称);
          [GENERIC MAP(类属映射表)]
          [PORT MAP(端口映射表)]
        END FOR;
    END FOR;
END 配置名称;
```

- 当出现元件例化语句中的映射端口名称与元件声明语句中的端口名称不一致时，需要选用配置端口映射子句。
- 当需要向元件传递参数时，可以在元件例化语句中通过类属映射传递参数，但必须重新编译元件例化语句所在的实体和结构体才可以生效；如果通过元件配置中的类属映射子句传递参数，则不需重新编译元件例化语句所在的实体和结构体。

元件配置实例

```
1  LIBRARY IEEE;
2  USE IEEE.Std_Logic_1164.ALL;
3  ENTITY inverter IS
4    PORT(a: IN Std_Logic;
5         b: OUT Std_Logic);
6  END inverter;
7  ARCHITECTURE inv_arch OF inverter IS
8  BEGIN
9    b <= NOT a;
10 END inv_arch;
11 CONFIGURATION config_inverter OF inverter IS
12   FOR inv_arch
13   END FOR;
14 END config_inverter;
```

```
1  LIBRARY IEEE;
2  USE IEEE.Std_Logic_1164.ALL;
3  ENTITY nand2 IS
4    PORT(a, b: IN Std_Logic;
5           c: OUT Std_Logic);
6  END nand2;
7  ARCHITECTURE nand2_arch OF nand2 IS
8  BEGIN
9    c <= a NAND b;
10 END nand2_arch;
11 CONFIGURATION config_nand2 OF nand2 IS
12   FOR nand2_arch
13   END FOR;
14 END config_nand2;
```

元件配置实例

```
1  LIBRARY IEEE;
2  USE IEEE.Std_Logic_1164.ALL;
3  ENTITY mux IS
4    PORT(a, b, c: IN Std_Logic;
5           out1: OUT Std_Logic);
6  END mux;
7  ARCHITECTURE mux_arch1 OF mux IS
8    SIGNAL a_n, b_n, c_not: Std_Logic;
9    COMPONENT inverter
10     PORT(a: IN Std_Logic; b: OUT Std_Logic);
11   END COMPONENT;
12   COMPONENT nand2
13     PORT(a, b: IN Std_Logic; c: OUT Std_Logic);
14   END COMPONENT;
15 BEGIN
16   U0: inverter PORT MAP(a=>c, b=>c_not);
17   U1: nand2 PORT MAP(a=>a, b=>c_not, c=>a_n);
18   U2: nand2 PORT MAP(a=>b, b=>c, c=>b_n);
19   U3: nand2 PORT MAP(a=>a_n, b=>b_n, c=>out1);
20 END mux_arch1;
```

元件配置实例

```
1 CONFIGURATION config1_mux OF mux IS
2   FOR mux_arch1
3     FOR U0: inverter USE CONFIGURATION Work.config_inverter;
4     END FOR;
5     FOR ALL: nand2 USE CONFIGURATION Work.config_nand2;
6     END FOR;
7   END FOR;
8 END config1_mux;
```

```
1 CONFIGURATION config2_mux OF mux IS
2   FOR mux_arch1
3     FOR U0: inverter USE ENTITY Work.inverter(inv_arch);
4     END FOR;
5     FOR U1,U2,U3: nand2 USE ENTITY Work.nand2(nand2_arch);
6     END FOR;
7   END FOR;
8 END config2_mux;
```

元件配置实例

```
1 CONFIGURATION config1_mux OF mux IS
2   FOR mux_arch1
3     FOR U0: inverter USE CONFIGURATION Work.config_inverter;
4     END FOR;
5     FOR ALL: nand2 USE CONFIGURATION Work.config_nand2;
6     END FOR;
7   END FOR;
8 END config1_mux;
```

```
1 CONFIGURATION config2_mux OF mux IS
2   FOR mux_arch1
3     FOR U0: inverter USE ENTITY Work.inverter(inv_arch);
4     END FOR;
5     FOR U1,U2,U3: nand2 USE ENTITY Work.nand2(nand2_arch);
6     END FOR;
7   END FOR;
8 END config2_mux;
```

元件配置实例

参数名称不一致时的元件配置

```
1  CONFIGURATION config3_mux OF mux IS
2    FOR mux_arch1
3      FOR U0: inverter USE ENTITY Work.inverter(inv_arch);
4        PORT MAP(x=>a,y=>b);
5      END FOR;
6      FOR U1,U2,U3: nand2 USE ENTITY Work.nand2(nand2_arch);
7        PORT MAP(x=>a,y=>b,z=>c);
8      END FOR;
9    END FOR;
10 END config3_mux;
```

结构体声明中的元件配置

- 在元件配置中，配置声明是独立于结构体的。
- 有另一种简单的元件配置声明形式，可以在结构体的声明语句部分直接声明配置。

```
FOR {例化元件名称 |OTHERS|ALL}: 模板元件名称 | 实体名称
    USE CONFIGURATION 设计库名称. 配置名称 |
    USE ENTITY 设计库名称. 实体名称 (结构体名称);
[PORT MAP(端口映射表);]
```

结构体声明中的元件配置

- 在元件配置中，配置声明是独立于结构体的。
- 有另一种简单的元件配置声明形式，可以在结构体的声明语句部分直接声明配置。

```
FOR {例化元件名称 |OTHERS|ALL}: 模板元件名称 | 实体名称
    USE CONFIGURATION 设计库名称. 配置名称 |
    USE ENTITY 设计库名称. 实体名称 (结构体名称);
[PORT MAP(端口映射表);]
```

结构体中声明的元件配置实例

```
1  LIBRARY IEEE;
2  USE IEEE.Std_Logic_1164.ALL;
3  ENTITY mux IS
4    PORT(a, b, c: IN Std_Logic;
5           out1: OUT Std_Logic);
6  END mux;
7  ARCHITECTURE mux_arch1 OF mux IS
8    SIGNAL a_n, b_n, c_not: Std_Logic;
9    COMPONENT inverter
10     PORT(a: IN Std_Logic; b: OUT Std_Logic);
11   END COMPONENT;
12   COMPONENT nand2
13     PORT(a, b: IN Std_Logic; c: OUT Std_Logic);
14   END COMPONENT;
15   FOR U0: inverter USE CONFIGURATION Work.config_inverter;
16   FOR ALL: nand2 USE ENTITY Work.nand2(nand2_arch);
17 BEGIN
18   U0: inverter PORT MAP(a=>c, b=>c_not);
19   U1: nand2 PORT MAP(a=>a, b=>c_not, c=>a_n);
20   U2: nand2 PORT MAP(a=>b, b=>c, c=>b_n);
21   U3: nand2 PORT MAP(a=>a_n, b=>b_n, c=>out1);
22 END mux_arch1;
```

块的配置

如果在设计实体的结构描述中使用了块语句，则可以用块配置来指定不同块中的各个元件在例化时与不同的模板元件如何连接。

```
CONFIGURATION 配置名称 OF 实体名称 IS
    FOR 结构体名称
        {FOR 块名称
            {FOR {例化元件名称 |OTHERS|ALL}: 模板元件名称 | 实体名称
                USE CONFIGURATION 设计库名称. 配置名称 |
                USE ENTITY 设计库名称. 实体名称 (结构体名称);
              [PORT MAP(端口映射表);]
            END FOR;}
        END FOR;}
    END FOR;
END 配置名称;
```

块的配置

如果在设计实体的结构描述中使用了块语句，则可以用块配置来指定不同块中的各个元件在例化时与不同的模板元件如何连接。

```
CONFIGURATION 配置名称 OF 实体名称 IS
    FOR 结构体名称
        {FOR 块名称
            {FOR {例化元件名称 |OTHERS|ALL}: 模板元件名称 | 实体名称
                USE CONFIGURATION 设计库名称. 配置名称 |
                USE ENTITY 设计库名称. 实体名称 (结构体名称);
              [PORT MAP(端口映射表);]
            END FOR;}
        END FOR;}
    END FOR;
END 配置名称;
```

小结

VHDL 的共享机制、可重用设计和多层次设计。

- 子程序 函数声明、过程声明。
- 程序包 程序包声明、程序包体。
- 重载 枚举类型的值重载、子程序重载、运算符重载。
- 决断函数 决断信号声明、决断函数声明。
- 配置 默认连接与默认配置、元件配置和块配置。

Stay Hungry, Stay Foolish!