

# 数字系统设计

郑海永

上课时间地点：周 2/9 10 节/3308

电子信息科学与技术 2010 级和 2011 级

中国海洋大学 电子工程系

2013 年 6 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 VHDL 深入

- 子程序
- 程序包和设计库
- 重载
- 决断信号与决断函数
- 配置



# 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** 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** 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。



















## 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。

















# 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;
```













## 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VHDL87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VHDL87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VHDL87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。









# 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

## 1 VHDL 深入

- 子程序
- 程序包和设计库
- 重载
- 决断信号与决断函数
- 配置

# 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;
```



# 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。



# 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  $\Rightarrow$  VHDL 分析器  $\Rightarrow$  中间形式  $\Rightarrow$  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



- 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，都隐含有下列两条语句：

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

标准（STANDARD）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（STANDARD）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（STANDARD）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;
```





## 3-8 译码器 74LS138

$g1$	$g2a\_n$	$g2b\_n$	$c$	$b$	$a$	$y\_n7$	$y\_n6$	$y\_n5$	$y\_n4$	$y\_n3$	$y\_n2$	$y\_n1$	$y\_n0$
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



## 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

## 1 VHDL 深入

- 子程序
- 程序包和设计库
- 重载
- 决断信号与决断函数
- 配置

# 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;
```



## 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

## 1 VHDL 深入

- 子程序
- 程序包和设计库
- 重载
- 决断信号与决断函数
- 配置

## 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；

```
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。



# 内容提要

## 1 VHDL 深入

- 子程序
- 程序包和设计库
- 重载
- 决断信号与决断函数
- 配置

# 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。























## 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);]
```









