# VHDL 语法学习笔记

# 一、VHDL 简介

## 1.1 VHDL 的历史

VHDL 的英文全名是 Very-High-Speed Integrated Circuit Hardware DescriptionLanguage，诞生于 1982 年。

1987 年底，VHDL 被 IEEE 和美国国防部确认为标准硬件描述语言。自 IEEE 公布了 VHDL 的标准版本 IEEE-1076（简称 87 版）之后，各 EDA 公司相继推出了自己的 VHDL 设计环境，或宣布自己的设计工具可以提供 VHDL 接口。此后 VHDL 在电子设计领域逐步取代了原有的各种非标准硬件描述语言。

1993 年，IEEE 对 VHDL 进行了修订，从更高的抽象层次和系统描述能力上扩展 VHDL 的内容，并公布了新版本的 VHDL，即 IEEE 标准的 1076-1993 版本（简称 93 版）。
现在，VHDL 和 Verilog HDL 作为 IEEE 的工业标准硬件描述语言，在电子工程领域已成为事实上的通用硬件描述语言。

## 1.2 VHDL 的特点

VHDL 主要用于描述数字系统的结构、行为、功能和接口。除了含有许多具有硬件特征的语句外，VHDL 在语言形式、描述风格和句法上与一般的计算机高级语言十分相似。VHDL 的程序结构特点是将一项工程设计，或称设计实体（可以是一个元件、一个电路模块或一个系统）分成外部和内部两部分。

外部也可称为可视部分，它描述了此模块的端口，而内部可称为不可视部分，它涉及到实体的功能实现和算法完成。在对一个设计实体定义了外部端口后，一旦其内部开发完成，其他的设计就可以直接调用这个实体。这种将设计实体分成内外部分的概念是 VHDL 系统设计的基本点。

应用 VHDL 进行工程设计有以下的优点：

1．行为描述

与其他的硬件描述语言相比，VHDL 具有更强的行为描述能力，强大的行为描述能力是避开具体的器件结构，从逻辑行为上描述和设计大规模电子系统的重要保证。

2．仿真模拟

VHDL 丰富的仿真语句和库函数，使得在任何系统的设计早期就能查验设计系统的功能可行性，随时可对设计进行仿真模拟。

3．大规模设计

一些大型的 FPGA 设计项目必须有多人甚至多个开发组共同并行工作才能实现。VHDL

语句的行为描述能力和程序结构决定了它具有支持大规模设计的分解和已有设计的再利用功能。

4．门级网表

对于用 VHDL 完成的一个确定的设计，可以利用 EDA 工具进行逻辑综合和优化，并自动把 VHDL 描述设计转变成门级网表。

5．独立性

VHDL 对设计的描述具有相对独立性，设计者可以不懂硬件的结构，也不必对最终设计实现的目标器件有很深入地了解。

# 二、VHDL 程序基本结构

一般的 VHDL 程序可以由**实体（Entity）、结构体（Architecture）、配置（Configuration）、程序包和程序包体（Package）以及库（Library）**5 个部分组成，它们是 VHDL 程序的设计单元。

其中实体、配置和程序包属于初级设计单元，主要的功能是进行端口、行为、函数等的定义。结构体和程序包体是次级设计单元，包含了所有行为以及函数的实现代码。其中，程序包和程序包体又属于公用设计单元，即它们是被其他程序模块调用的。库则是一批程序包的集合。

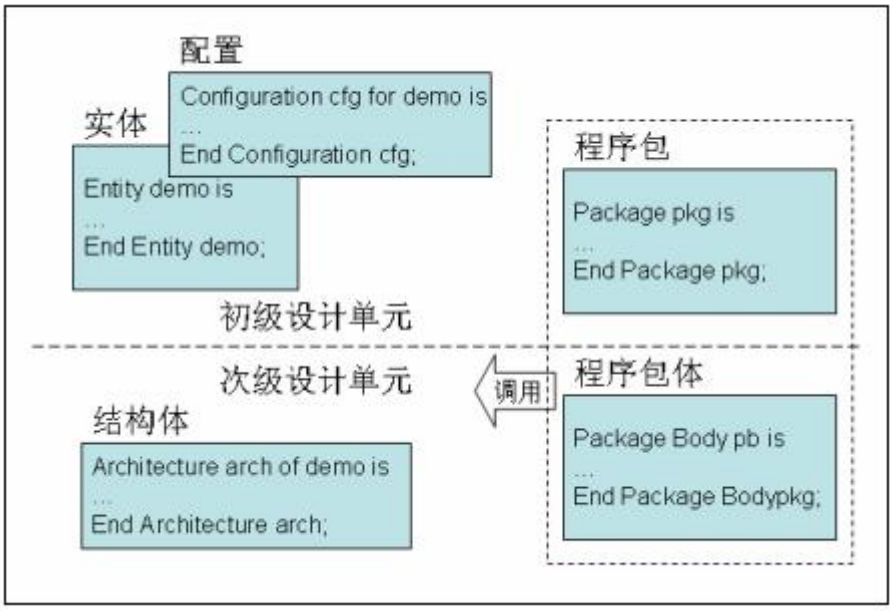



图 1 所示为 VHDL 程序设计单元之间的关系。

无论是复杂的还是简单的数字模块，用 VHDL 来描述都至少需要包括两个部分，即实体申明（Entity Declaration）和结构体（Architecture）。其中实体申明用于说明模块的端口，而结构体用于描述模块的功能。本节下面将详细介绍 VHDL 程序的各个设计单元。

## 2.1 实体（Entity）的申明方法

实体是设计的基本模块和设计的初级单元，在分层次设计中，顶层有顶级实体，含在顶级实体中的较低层次的描述为低级实体，通过配置可把顶层实体和底层实体连接起来。可以将实体理解为电路图设计中的芯片符号（Symbol），符号规定了电路的符号名、接口和数据类型。由连线（或信号）将符号互连建立设计所需的电路图，互连线生成的网表，在设计实现之前一直是设计验证的仿真模型，并在设计验证后，由网表向布线工具提供所需的连接信息和层信息。

图 2 所示是传统设计中 R-S 触发器的符号图，用 VHDL 对其进行描述的代码如下

```
ENTITY rsff IS
PORT (
        Set, Reset : IN BIT;
        Q, QB : BUFFER BIT );
END rsff;
```

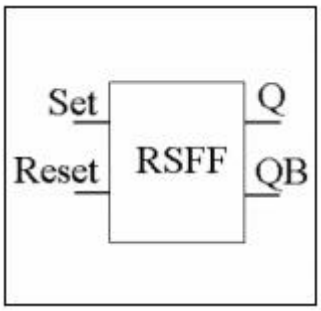



图 2 R-S 触发器的 VHDL 实体描述和符号

实体语句用关键词 ENTITY 开头，实体名 rsff 是描述的符号名，在结束实体语句的 END rsff 之间，实体语句可以用关键词 BEGIN 把实体语句分成两部分：即 BEGIN 之前是实体说明，BEGIN 之后是实体语句。

在 ENTITY 语句的实体说明部分，常用 PORT 付语描述实体对外界连接的端口（数目、方向和数据类型）。实体 rsff 有 4 个端口，Set/Reset 是输入 IN 模式，Q/QB 是输出 BUFFER（缓冲）模式，都为 BIT 类型。实体说明中还可说明数据类型、子程序和常量等数据信息，实体语句常用于描述设计经常用到的判断和检查信息。

实体描述的格式如下：

```
ENTITY 实体名 IS
    [GENERIC(参数表);]
    [PORT(端口表);]
[BEGIN
    实体语句部分;]
END [ENTITY] [实体名];
```

**参数 GENERIC**

用于说明设计实体和其外部环境通信的对象，规定端口的大小、实体中子元件的数目、实体的延时特性等。只能用整数类型表示，如整型、时间型等，其他类型的数据不能逻辑综合。格式如下：

```
GENERIC ([CONSTANT]属性名称:[IN]子类型标识[:=静态表达式],……);
```

**端口 PORT**

port 和 port，port 和 signal 均可同名，但是连接时仍要声明

port 用于定义模块的端口，它的格式如下：

```
PORT( [SIGNAL]  端口名称:[方向]  类型标识[BUS] [:=静态表达式],
      [SIGNAL]  端口名称:[方向]  类型标识[BUS] [:=静态表达式],
        …
      [SIGNAL]  端口名称:[方向]  类型标识[BUS] [:=静态表达式]);
```

- SIGNAL：SIGNAL 是关键字，但是由于 **PORT 之后必须是信号类**，所以一般可以将 SIGNAL 关键字省略。
- 端口名称：是该端口的标识，通常由英文字母和数字组成，但是必须是英文字母打头。
- 方向：定义了端口是输入还是输出，如 IN、OUT。表明端口方向的关键字如表 1 所示。

| 关键字 | 意义 |
|---|---|
| IN | 输入，信号从此端口输入模块 |
| OUT | 输出，信号从模块的此端口输出 |
| INOUT | 双向端口，既可输入也可输出 |
| BUFFER | 输出信号，此信号模块可再用 |
| LINKAGE | 不指定方向，无论哪个方向都可以连接 |

- 类型标识：说明流过该端口的数据类型，常用的数据类型有 BIT（位）、BIT_VECTOR（位向量）、BOOLEAN（布尔型）和 INTEGER（整数型）4 种。
- BUS 关键字：在该端口和多个输出端相连的情况下使用。

## 2.2 结构体（Architecture）的描述方法

结构体描述实体的行为功能，一个实体可以有多个结构体。结构体是一个基本设计单元，它具体地指明了所设计模块的行为、元件及内部的连接关系，也就是说它定义了设计单元具体的功能。结构体对其基本设计单元的输入/输出关系可以用 3 种方式进行描述，即行为描述（基本设计单元的数学模型描述）、寄存器传输描述（数据流描述）和结构描述（逻辑元件连接描述）。

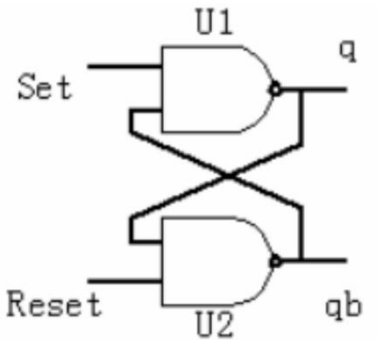
不同的描述方式只体现在描述语句上，而结构体的结构是完全一样的。由于结构体是对实体功能的具体描述，因此它一定要跟在实体的后面。通常，先编译实体之后才能对结构体进行编译。如果实体需要重新编译，那么相应结构体也应重新进行编译。

结构体的格式如下：

```
ARCHITECTURE 结构体名 OF 实体名 IS
[定义语句]
BEGIN
[并行处理语句]
END 结构体名;
```

定义语句用于对结构体内部所使用的信号、常数、数据类型和函数等进行定义。信号定义和端口说明的语句一样，应有信号名和数据类型的说明，但因它是内部连接用的信号，故没有也不需有方向的说明。并行处理语句具体地描述了结构体的行为及其连接关系，它们都是可以并行执行的。

以上面介绍的 R-S 触发器为例。假设已经有一个实现了与非功能的模块 nand2，用它实现 R-S 触发器的原理图如图 3 所示。




对应以上原理图的结构体描述如下：

```
library IEEE;use IEEE.std_logic_1164.all;
ENTITY rsff isPORT (
        set, reset: in bit;
        q, qb: buffer
        );
END ENTITY;

ARCHITECTURE arch_rsff OF rsff IS
        COMPONENT nand2
             PORT (
                        a, b : IN BIT;
                        c: OUT BIT);
         END COMPONENT;

  BEGIN
          U1: nand2
          PORT MAP (set, qb, q);
          U2: nand2
          PORT MAP (reset, q, qb);
  END arch_rsff;
```

# 结构体说明区

ARCHITECTURE 和 BEGIN 之间是结构体说明区，结构体说明区描述**组件（COMPONENT）**和**局部信号（SIGNAL）**。

## 组件（COMPONENT）

上面的代码中，以关键字 ARCHITECTURE 作为结构体的开头，结构体名为 arch_rsff，表示描述 rsff 实体的结构体 arch_rsff。

# 结构体语句区

BEGIN 和 END 之间是结构体语句区。结构体语句中用的具体元件（上例是 nand2）均应在结构体说明中说明接口，以便将描述的信息通知给编辑器。

## 端口映射（PORT MAP）

component port => entity port，=> open 表示断开连接

## 参数映射（generic map）

component generic => entity generic

```
begin
uart_1: COMPONENT mod_m_counter
                GENERIC MAP (n=>div_nbit,m=>div_m)
                PORT MAP (clk=>clk,rst=>rst,max_tick=>tick,q=>open);
end
```

## =>的其他用法

case 语句

```
CASE expression IS
WHEN choice1 =>
        sequence_of_statements
```

状态机

```
case cnt is
                when 0=>
                    if () then
                        ……
                    cnt:=1;
                    end if;

                when 1=>
                    if () then
                        ……
                    cnt:=0;
                    end if;
end case;
```

多位清零

```
count    <= (others => '0');
```

等价于 count（7 downto 0）<=“00000000”

## 子模块语句结构

如果设计者希望将模块分为若干个相对比较独立的子模块进行描述，可以将一个结构体用几个子结构来构成。VHDL 结构体描述常常用到 3 种**语句结构：PROCESS 语句结构、BLOCK 语句结构和子程序结构**。

## 1). PROCESS 语句结构

process 进程语句是一种并发处理语句，在一个结构体中多个 PROCESS 语句可以同时**并行运行**（相当于多个 CPU 同时运作）。PROCESS 语句是 VHDL 语言中描述硬件系统并发行为的最基本语句。

PROCESS 语句归纳起来有如下几个特点：

- 它可以与其他进程并发运行，并可存取结构体或实体号中所定义的信号；
- 进程结构中的所有语句都是按顺序执行的；
- 为启动进程，在进行结构中必须包含一个显式的敏感信号量表或包含一个 WAIT 语句；
- 进程之间的通信是通过信号量传递来实现的。

PROCESS 语句的格式如下：

```
[进程名]:PROCESS(信号 1,信号 2,…)
BEGIN
…
END PROCESS;
```

一般情况下进程名可以被省略。进程申明关键字 PROCESS 后面括号内的信号是此进程的敏感信号，这些信号的变化会激活过程的执行。例如下面的代码就表示过程 main_proc 在信号 clk 和 reset 变化时执行：

```
library IEEE;use IEEE.std_logic_1164.all;
ENTITY counter is
    PORT (
            clk, reset: in bit;
            c: out bit
            );
    END ENTITY;

    ARCHITECTURE arch of counter is
```

```
BEGIN
    main_proc:
    PROCESS(clk, reset)
    BEGIN
    if (reset = '1') then
    …
    end if;
    END PROCESS;
END
```

**变量（VARIABLE）**

在进程中也可以定义一些变量，这些变量是局部量，只能在进程内部使用，它们的赋值是立即生效的。局部变量定义的格式如下：

VARIABLE 变量名:数据类型 [约束条件] [:=表达式];

下面的代码演示了定义一个局部变量并且使用它的方法：

```
library IEEE;use IEEE.std_logic_1164.all;
ENTITY counter is
    PORT (
            a: in bit;
            c: out bit
            );
    END ENTITY;
ARCHITECTURE arch of counter is
    BEGIN
        main_proc:
        PROCESS(a)
        VARIABLE item:array0(7 downto 0); --定义一个变量 item
        BEGIN
            item(7):=a;
            c<=item(7);
        END PROCESS;
    END
```

## 2). BLOCK 语句结构

BLOCK 语句的格式如下：

```
块名:BLOCK(条件)
    [参数 GENERIC 说明; [参数映射;] ]
```

[端口说明; [端口映射;] ]
[块说明语句]
BEGIN
并发语句组;
END BLOCK 块名;

BLOCK 放在结构体的并行语句组中，每一个 BLOCK 相当于一个子电路原理图。和 PROCESS 语句不同，BLOCK 内的语句是并发执行的。只要 BLOCK 右边的条件满足，BLOCK 内的语句就被执行。如果省略条件，表示本 BLOCK 被无条件执行。下面是一个 BLOCK 语句的例子：

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY test is
    PORT (
            a, b: in bit;
            s, c: out bit
        );END ENTITY;

ARCHITECTURE arch of test is
    BEGIN
        block_demo: BLOCK(clk='1')
        BEGIN
            s<=a xor b;
            c<=a and b;
        END BLOCK block_demo;
    END;
```

上面的程序表示当 clk 信号变为 1 时，并行执行 BLOCK 语句内的程序，即将 a 和 b 两个信号的异或结果赋给 s 信号，同时将 a 和 b 信号的与结果赋给 c 信号。

## 3)．子程序结构

所谓子程序结构就是将一部分实现代码放到公用的程序（即程序包 Package）文件中实现。程序包中的代码以子程序的方式提供给 VHDL 程序调用，这样代码可以实现共享，同时还使得 VHDL 程序的结构明了。
子程序在调用时首先要进行初始化，执行结束后子程序就终止，再调用时要再进行初始化。因此子程序内部的值不能保持，子程序返回以后才能被再调用，它是一个非重入的程序。
VHDL 中有两种类型的子程序—**过程（Procedure）和函数（Function）**，下面分别介绍一下它们的格式。

**过程（Procedure）**

```
PROCEDURE 过程名(参数 1;参数 2;……) IS
    定义语句;
    BEGIN
        顺序语句组;
    END 过程名;
```

每个参数的说明格式如下：

参数名:方向 类型

方向一般为 3 种：IN、OUT、INOUT。如果方向为 IN 则可省略方向说明。

**函数（Function）**

函数的格式如下：

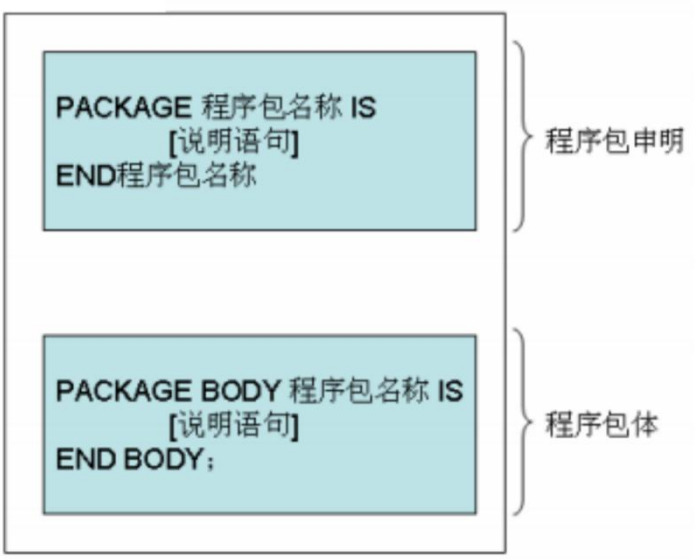
```
FUNCTION 函数名(参数 1;参数 2;……) RETURN 数据类型 IS
    定义语句;
    BEGIN
        顺序语句组;
        RETURN [返回变量名];
    END 函数名;
```

在 VHDL 语言中，函数的所有参数都是输入参数，因此都是 IN 的方向，可以省略方向说明。

在下面介绍程序包（PACKAGE）的时候将会介绍函数、过程定义的例子，在此不再举例说明。

## 2.3 程序包（package）和程序包体（package body）

程序包说明类似 C 语言中的 include 语句，用来罗列 VHDL 语言中所要用到的信号定义、常数定义、数据类型、元件语句、函数定义和过程定义等，它是一个可编译的设计单元，也是库结构中的一个层次。
程序包的结构如图 4 所示。




一个程序包由两大部分组成：程序包申明和程序包体。程序包体是一个可选项，也就是说，程序包可以仅仅由程序包标题构成。一般程序包标题列出所有项的名称，而程序包体具体给出各项的细节。
下面介绍一个包含与非函数的程序包的实现以及调用方法。实现与非函数程序包的代码如下：

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

-- 程序包申明
PACKAGE package_demo is
-- 函数申明
    FUNCTION nand2(a, b : in bit)
    RETURN bit;
END package_demo;

-- 程序包体 PACKAGE BODY package_demo is
        -- 函数实现
        FUNCTION nand2(a, b : in bit)
        RETURN bit IS
        VARIABLE ret: bit;
        BEGIN
            ret = not(a and b);
```

```
            return ret;
        END nand2;
    END BODY;
```

上面的代码在程序包申明中申明了函数 nand2，然后在程序包体中具体实现了此函数的功能。下面举个例子来说明程序包的使用方法，即函数的调用方法。假设要得到如下的逻辑关系式：

$$D = \overline{A1 \bullet B1} \bullet \overline{A2 \bullet B2}$$

可以用下面的代码描述：

```
library IEEE;
use IEEE.std_logic_1164.all;
use WORK.package_demo.all;

-- 调用自定义的程序包
ENTITY test is
    PORT (
            A1, B1, A2, B2: in bit;
            D: out bit
            );
    END ENTITY;

ARCHITECTURE arch of test is
    BEGIN
        -- 调用 nand 函数
        D = nand2(A1, B1) and nand2(A2, B2);
    END;
```

自定义的程序包必须首先进行编译，然后才能够编译调用此程序包的 VHDL 程序。自定义的程序包属于 WORK 库，所以申明调用的代码是：
use WORK.自定义程序包名称.all;
调用程序包中函数或者过程的方法和一般高级语言（如 C 语言）一样直接调用就可以了。

## 2.4 配置（CONFIGURATION）的申明方法

一个实体可以包含**多个结构体**，配置的作用就是根据需要选择实体的结构体。配置语句描述层与层之间的连接关系以及实体与结构之间的连接关系。设计者可以利用这种配置语句来选择不同的结构体，使其与要设计的实体相对应。在仿真某一个实体时，可以利用配置来选择不同的结构体，进行性能对比试验以得到性能最佳的结构体。

例如，设计一个二输入、四输出的译码器。如果一种结构中的基本单元采用反相器和三输入与门，而另一种结构中的基本元件都采用与非门。它们各自的结构体是不一样的，并且都放在各自不同的库中。那么现在要设计的译码器，就可以利用配置语句实现对两种不同构

造的选择。配置的基本格式如下：

```
CONFIGURATION 配置名 OF 实体名 IS
    [语句说明]
END 配置名;
```

如果一个实体仅仅具有一个结构体，也需要定义其配置，但是可以写成一种最为简洁的格式：

```
CONFIGURATION 配置名 OF 实体名 IS
    FOR 所选的构造体名
    END FOR;
END 配置名;
```

如果一个模块比较复杂，含有多个子模块，使用低层次配置可以为每个子模块选择其结构体，代码如下：

```
CONFIGURATION 配置名 OF 实体名 IS
    FOR 所选的构造体名
            FOR 标号 1:元件名 1 USE CONFIGURATION WORK.配置体名 1;
            END FOR;
            FOR 标号 2:元件名 2 USE ENTITY WORK.实体名 2(构造体名 2);
            END FOR;
            …
    END FOR;
END 配置名;
```

其中，低层次构造体名可用 ALL 或 OTHERS，构造体名 2 可以省略。

## 2.5 VHDL 程序的库

库（Library）是经编译后的数据的集合，它存放包集合申明、实体申明、构造体申明和配置定义。它的功能类似于 UNIX 和 MS-DOS 操作系统中的目录，在 VHDL 中，库的说明总是放在设计单元的最前面，这样在设计单元内的语句就可以使用库中的数据了。由此可见，使用库的好处是使设计者可以共享已经编译过的设计结果。在 VHDL 中可以存在多个不同的库，但是库和库之间是独立的，不能互相嵌套。
申明库的格式如下：

```
LIBRARY 库名;
```

在 VHDL 语言中存在的库大致可以归纳为 5 种：IEEE 库、STD 库、ASIC 矢量库、用户定义库和 WORK 库。

1)．IEEE 库
在 IEEE 库中的“STD_LOGIC_1164”包集合是 IEEE 正式认可的标准包集合。现在有些公司提供的包集合如“STD_LOGIC_ARITH”、“STD_LOGIC_UNSIGNED”等，尽管没有得到 IEEE 的承认，但是仍汇集在 IEEE 库中。

2）．STD 库
STD 库是 VHDL 的标准库，在库中存放有“STANDARD”包集合。由于它是 VHDL 的标准配置，因此设计者如要调用“STANDARD”中的数据可以不按标准格式说明。STD 库中还包含有“TEXTIO”包集合，在测试时使用。使用“TEXTIO”包集合中的数据时，应先说明库和包集合名：

```
LIBRARY STD;
USE STD.TEXTIO.ALL;
```

3）．ASIC 矢量库
在 VHDL 中，为了进行门级仿真，各公司可提供面向 ASIC 的逻辑门库。在该库中存放着与逻辑门一一对应的实体。

4）．WORK 库
WORK 库是现行作业库。设计者所描述的 VHDL 语句不需要任何说明，都将存放在 WORK 库中。在使用该库时无需进行任何说明。

5）．用户定义库
将用户自身设计开发的包、实体等汇集在一起定义成一个库，就是用户定义库或称用户库。在使用用户定义库时同样要首先说明库名。

以上各种库中，除 WORK 库外，其他 4 类库在使用前都首先要进行说明，格式为：

USE 库名.包集合名.项目名;

如果项目名为 ALL，则表示包集合中的所有项目都要使用，例如：

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.ALL;
USE WORK.STD_ARITH.ALL;
…
```

库说明语句的作用范围从一个实体说明开始到它所属的构造体、配置结束为止。当一个源程序出现两个或两个以上的实体时，两条作为使用库的说明语句就在每个实体说明语句前重复书写。例如，在一个 VHDL 文件中定义两个实体，库的申明如下：

```
-- 第一个实体的库申明
LIBRARY IEEE;USE IEEE.STD_LOGIC_1644.ALL;
```

```
-- 第一个实体申明
ENTITY ent1 is
…
END ent1;

-- 第一个实体的结构体
ARCHITECTURE arch1 of ent1 is
…
END arch1;

-- 第一个实体的配置
CONFIGURATION cfg1 of ent1 is
…
END cfg1;


-- 第二个实体的库申明
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1644.ALL;

-- 第二个实体申明
ENTITY ent2 is
…
END ent2;

-- 第二个实体的结构体

ARCHITECTURE arch2 of ent2 is
```

# 三、VHDL 语言的数据类型和运算符

VHDL 和其他高级语言一样，具有多种数据类型。对大多数数据类型的定义两者是一致的（例如整数型），但是也有一些数据类型是 VHDL 所独有的。表 2 所示为 VHDL 支持的数据类型和它的数据对象。

表 2 VHDL 数据类型和数据对象表

| 数据类型（数据对象取值的集合或子集） | 数据对象（传递信息的载体） |
|---|---|
| Scalar types（标量数据类型） | Constant（常量） |
| Enumeration（可枚举数据类型） | Variable（变量） |
| Integer （整数数据类型） | Signal（信号） |
| *Physical（物理数据类型） | *File（文件） |
| *Floating point（实数数据类型） | |
| Composite（复合数据类型） | |
| Array（数组数据类型） | |
| Record（记录数据类型） | |
| *Access（寻址数据类型） | |

注意：表 2 中带*号的数据类型表示不可以综合的类型或对象。

## 3.1 VHDL 语言的数据对象

VHDL 对象有 4 种，即**信号（Signal）、变量（Variable）、常量（Constant）和文件（File）**。其中文件（File）是 VHDL’93 标准中新通过的，它是不可以综合的。下面介绍一下常量、信号和变量的申明方法。

### 1）．信号（Signal）

信号用于将元件的装配端口连在一起形成模块，它的作用相当于连接元件的导线。信号是实体间动态数据交换的手段，信号申明格式如下：

SIGNAL signal_name : signal_type [ :=initial_value ] ;

在关键字 SIGNAL 后跟一个或者多个信号名，每个信号名将建立一个新信号，用冒号把信号名和信号的数据类型分隔开，信号数据类型规定信号包含的数据类型信息及初始化信号指定的初值，用 := 对信号/变量/常量 赋值。
实体说明部分（port 定义即为 signal 类型）、结构体说明（signal 定义）和程序包说明都能声明信号，全局信号在程序包中声明，它们被所属的实体分享。

## 2）．变量（Variable）

变量用于存储进程和子程序中的局部数据，变量的赋值是立即执行的，没有延时。变量的申明格式如下：

VARIABLE variable_name ,variable_name : variable_type[:= value];

关键字 VARIABLE 后跟着一个或多个变量名，每个变量名对应建立一个新变量。variable_type 字段定义了变量的数据类型，并且还可以指定一个可选的初值。此外，还需要注意的是只可以在进程说明部分和子程序说明部分声明变量。
和信号相比，变量有以下优点：
• 变量处理起来更快，因为变量赋值是立即发生的，而信号却必须为此事件作相应的处理。
• 变量用很少的存储器，相反为了做一个调度安排和处理信号属性，需要存储更多的信号信息。
• 变量比信号更容易实现同步处理。

## 3）．常量（CONSTANT）

常量是为特定的数据类型值所赋予的名称，如果需要在多个具体元件中存放一个固定值就使用常量。例如可以如下定义常量 PI（π ）：

CONSTANT PI: REAL:= 3.1416;

定义常量的格式如下：

CONSTANT constant_name,constant_name: type_name[:= value];

一般情况下，VHDL 中的常量是在程序包申明中进行申明，而在程序包体中指定具体的值。使用常量需要注意以下几个问题：
• 在程序包中说明的常量被全局化。
• 在实体说明部分的常量被那个实体中任何结构体引用。
• 在结构体中的常量能被其结构体内部任何语句采用，包括为进程语句采用。
• 在进程说明中说明的常量只能在进程中使用。
• 在数组和一些线性运算中经常用常量表，VHDL 的设计描述用常量表特别适于实现 ROM 网络的电路与函数设计。

## 3.2 VHDL 语言的数据类型

VHDL 的数据类型根据使用目的和场合，可以分为标准数据类型和用户定义的数据类型两种。

## 1）. 标准数据类型

VHDL 中定义的标准数据类型如表 3 所示。

| 数据类型名 | 类型符号 | 说明和取值举例 |
|---|---|---|
| 标准逻辑 | STD_LOGIC | 逻辑‘0’或逻辑‘1’、‘Z’ |
| 标准逻辑向量 | STD_LOGIC_VECTOR | "0011ZZ"， X"00BB" |
| 位 | BIT | '0'， '1' |
| 位矢量 | BIT_VECTOR | "001100"， X"00BB" |
| 整数 | INTEGER | 整数 32 位，-（$2^{31}$-1） ~（$2^{31}$-1） |
| 实数 | REAL | 浮点数，-1.0×1038~1.0×1038 |
| 布尔量 | BOOLEAN | TRUE， FALSE |
| 字符 | CHARACTER | 用单引号括起来。'a'， 'A' |
| 字符串 | STRING | 用双引号括起来。"My good" |
| 时间 | TIME | 单位用 fs， ps， ns， us， ms， sec， min， hr |
| 错误等级 | SEVERITY LEVEL | 表征系统的状态：NOTE、WARNING、ERROR、FAILURE |
| 自然数 | NATURAL | 大于或等于 0 的整数 |
| 正整数 | POSITIVE | 大于 0 的整数 |

CHARACTER ：’ ’（字符）

其中，在数据类型后面，可以加上约束区间，比如：

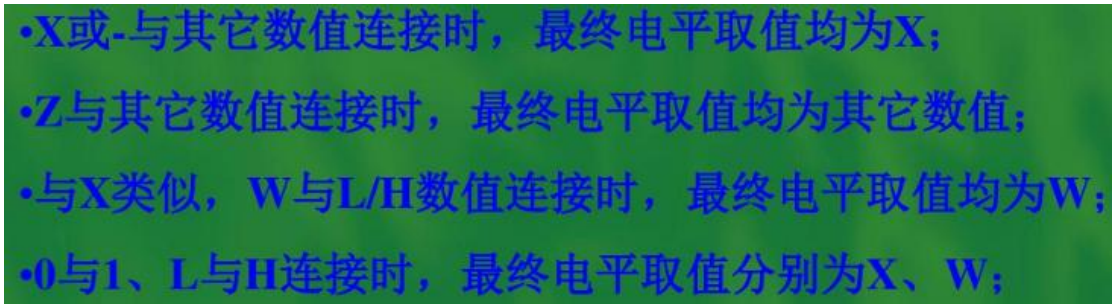
INTEGER RANGE 100 downto 1;
BIT_VECTOR(3 downto 1)
real range 2.0 to 30.0;

Downto 表示高位在前，to 表示低位在前，range to 表示取值范围

STD_LOGIC 和 STD_LOGIC_VECTOR 的逻辑数据取值可以有 9 种状态，如表 4 所示。

| 逻辑值 | 说明 |
|---|---|
| ‘0’ | 逻辑 0，这和 bit 类型相同 |
| ‘1’ | 逻辑 1，这和 bit 类型相同 |
| ‘X’ | 不定（赋值时使用） |
| ‘U’ | 不定（初始值） |
| ‘Z’ | 高阻 |
| ‘W’ | 弱信号不定 |
| ‘L’ | 弱信号 0 |
| ‘H’ | 弱信号 1 |
| ‘- ‘ | 不可能情况 |

表 4 标准逻辑（向量）取值表




线与

| | X | 0 | 1 | Z | W | L | H | - |
|---|---|---|---|---|---|---|---|---|
| X | X | X | X | X | X | X | X | X |
| 0 | X | 0 | X | 0 | 0 | 0 | 0 | X |
| 1 | X | X | 1 | 1 | 1 | 1 | 1 | X |
| Z | X | 0 | 1 | Z | W | L | H | X |
| W | X | 0 | 1 | W | W | L | H | X |
| L | X | 0 | 1 | L | W | L | W | X |
| H | X | 0 | 1 | H | W | W | H | X |
| - | X | X | X | X | X | X | X | X |

## 2）．用户定义的数据类型（TYPE）

用户定义数据类型的格式如下：

TYPE 数据类型名 {,数据类型名} 数据类型定义;

**例：状态机实现**

```
architecture
        type state is (idle,start,data,stop);
        signal pr_state,nx_state: state;

begin

    process behavior of ？？is
    begin
    case pr_state is
```

```
            when idle=>
                if () then
                    ……
                nx_state<=start;
                end if;

            when start=>
                if () then
                    ……
                nx_state<=data;
                end if;

            when data=>
                if () then
                    ……
                nx_state<=stop;
                end if;

            when stop=>
                if () then
                    ……
                nx_state<=idle;
                end if;
        end case;
    end process;
end behavior;
```

数据类型定义放在语句的定义部分中，定义范围为从本定义行开始到本语句作用的最后。一般用户定义的**数据类型**分为以下几种。

- **枚举类型（enumeration）**

枚举类型的格式如下：

TYPE 数据类型名 IS (元素,元素,……)
例如，将一星期七天作为一个枚举，可以如下定义：

```
TYPE week IS (sun, mon, tue, wed, thu, fri, sat);
```

在枚举类型中，元素是有序列性的，第 1 个元素对应逻辑电路状态 000，第 2 个为状态 001，第 3 个为状态 010……后一个逻辑状态为前一个元素逻辑状态加 1。所以，上面的例子中，sun 对应逻辑状态 000，mon 对应逻辑状态 001，……，sat 对应逻辑状态 110。

• **整数类型、实数类型（INTEGER，REAL）**

这里的整数类型和实数类型其实是前面所述的标准整数类型和实数类型的子类，定义的格式如下：

TYPE 数据类型名 IS 数据类型定义 约束范围

TYPE current IS REAL RANGE -1E4 TO 1E4
定义了 current 类型实数的范围是-104 到 104。

• **数组（ARRAY）**
数组定义的格式如下：

TYPE 数据类型名 IS ARRAY 范围 OF 原数据类型名；
注意：如果 范围 这一项没有被指定，则使用整数数据类型。

下面通过例子说明数组的定义方法。

```
TYPE word IS ARRAY (1 TO 8) OF STD_LOGIC;
TYPE tmem IS ARRAY (0 TO 2， 3 DOWNTO 0) OF STD_LOGIC;
```
以上定义了 tmem 一种数组类型，可以定义一个此类新的常数，如下：

```
CONSTANT mem:tmem:= ( ('0', '0', '0', '0'),
                      ('0', '0', '1', '0'),
                      ('1', '1', '0', '0'));
```

当范围这一项需用整数类型以外的其他数据类型时（如枚举类型），则应在指定数据范围前加数据类型名。例如：

TYPE week IS (sun, mon, tue, wed, thu, fri, sat);TYPE workdate IS ARRAY (week mon TO fri) OF STD_LOGIC;

如果要取得数组内的一个元素，格式如下：

数组名(下标)

例如，word（1）区的 word 数组序号为 1 的元素。

当数组类型定义中的范围用“（Natural Range <>）”或“（Positive Range <>）”代替时，表示本**数组类型为非限定的类型，下标范围在信号或变量定义时再具体指定**。
例如下面的代码中定义的 array0 就是非限定类型的数组。

```
library IEEE;
use IEEE.std_logic_1164.all;
entity tmyarray is
```

```
port(a:in std_logic;c:out std_logic);
end ;
architecture myarray of tmyarray is
type array0 is array (natural range <>) of std_logic;
Begin
    process(a)
            variable item:array0(7 downto 0);--定义一个变量 item
        Begin
            item(7):=a;
            c<=item(7);
        end process;
    end;
```

**• 时间（TIME）**

定义时间的格式如下：

```
TYPE 数据类型名 IS 范围
    UNITS   基本单位;
            单位描述;
    END UNITS
```

例如：

```
TYPE time IS RANGE -1E18 TO 1E18
    UNITS fs;
            ps=1000fs;
            ns=1000ps;
            us=1000ns;
            ms=1000us;
            sec=1000ms;
            min=60sec;
            hr=60min;
     END UNITS;
```

**• 记录（Record）**

记录的定义格式是：

```
TYPE 数据类型名 IS RECORD
    元素名:数据类型名;
    元素名:数据类型名;
        …
    元素名:数据类型名;
END RECORD;
```

注意：引用记录数据类型中的元素应使用“.”，而不是数组的括号。

例如，定义一个窗口尺寸的记录，如下：

```
TYPE window IS RECORD
    length:INTEGER;
    width:INTEGER;
END RECORD;
```

当需要使用 window 类型记录的元素时，方法如下：

```
signal win: window;win.length<=10;
```

- **子类型（SUBTYPE）**

用户定义的子类型是用户对已定义的数据类型做一些范围限制而形成的一种数据类型。子类型的名称通常采用用户较容易理解的名字。子类型的定义格式为：

```
SUBTYPE 子类型名 IS 数据类型名[范围];
```

例如：

```
SUBTYPE digit IS INTEGER RANGE 0 TO 9;
SUBTYPE abus IS STD_LOGIC_VECTOR(7 DOWNTO 0);
signal a: STD_LOGIC_VECTOR (7 downto 0);
signal b: STD_LOGIC_VECTOR (15 downto 0);
signal c: abus;
a<=c; --正确
b<=c; --错误
```

## 3）．类型变换

在 VHDL 中，数据类型的定义是相当严格的，不同类型的数据是不能进行运算和直接代入的。为了实现正确的代入操作，必须将要代入的数据进行类型变换。变换函数通常由 VHDL 语言的包集合提供。

例如在“ STD_LOGIC_1164 ”、“STD_LOGIC_ARITH”、STD_LOGIC_UNSIGNED”的包集合中提供了如表 2-5 所示的数据类型变换函数。

| 所属包 | 函数名 | 功能 |
|---|---|---|
| STD_LOGIC_1164 | TO_STDLOGICVECTOR（A） | 由 BIT_VECTOR 转换为 STD_LOGIC_VECTOR |
| | TO_BITVECTOR（A） | 由 STD_LOGIC_VECTOR 转换为 BIT_VECTOR |
| | TO_STDLOGIC（A） | 由 BIT 转换为 STD_LOGIC |
| | TO_BIT（A） | 由 STD_LOGIC 转换为 BIT |
| STD_LOGIC_ARITH | CONV_STD_LOGIC_VECTOR | 由 INTEGER，UNSIGNED，SIGNED 转换成 STD_LOGIC_VECTOR |
| | CONV_INTEGER（A） | 由 UNSIGNED，SIGNED 转换成 INTEGER |
| STD_LOGIC_UNSIGNED | CONV_INTEGER（A） | 由 STD_LOGIC_VECTOR 转换成 INTEGER |

有些数据，从数据本身是断定不出其类型的，如“01010001”，如果没有上下文，VHDL编译器就无法知道它是字串型还是位数组类型。这时就要进行数据类型的限定。类型限定的格式如下：

类型名'(数据)

例如：
a<=std_logic_vector'("01010001");
这样，编译器知道“01010001”肯定是矢量型，而不是别的类型。

## 3.3 VHDL 语言的运算符

在 VHDL 语言中，常用的运算符有**逻辑运算（Logic）、关系运算（Relational）、算术运算（Arithmetic）和移位运算（Shift）**，下面分别对它们进行介绍。

### 1）．逻辑运算符

逻辑运算符可以对 bit 和 boolean 类型的值进行运算，也可对这些类型的一维数组进行运算。对数组型的运算，运算施加于数组中的每个元素，结果与原来数组长度相同。

**逻辑判断的运算为“短路运算”**，也就是说，条件表达式的左边成立时，就不再进行右边的判断。比如，IF （a=0） AND （b/a>2） THEN…这个判断运算，当 a=0 时，后面的判断不再继续，避免出现除数为 0 的运算。
VHDL 的逻辑运算符如表 6 所示。

| 操作符 | 功能 | 操作数据类型 |
|---|---|---|
| AND | 与 | BIT，BOOLEAN，STD_LOGIC |
| OR | 或 | BIT，BOOLEAN，STD_LOGIC |
| NOT | 非 | BIT，BOOLEAN，STD_LOGIC |
| NAND | 与非 | BIT，BOOLEAN，STD_LOGIC |
| NOR | 或非 | BIT，BOOLEAN，STD_LOGIC |
| XOR | 异或 | BIT，BOOLEAN，STD_LOGIC |
| XNOR | 异或非 | BIT，BOOLEAN，STD_LOGIC |

### 2）．关系运算符

关系运算符两边必须为相同的类型，其结果为 boolean 类型。

等号（=）和不等号（/=）两边可以为任意类型的运算对象。其他关系运算符的运算对象必须为标量类型或离散类型的一维数组。对于复杂的运算对象，如数组，两个值相等意味着两个值的所有对应元素相等。VHDL 的关系运算符如表 7 所示。

| 操作符 | 功能 | 操作数据类型 |
|---|---|---|
| = | 相等 | 任何数据类型 |
| /= | 不等 | 任何数据类型 |
| < | 小于 | 枚举与整数，及对应的一维数组 |
| > | 大于 | 枚举与整数，及对应的一维数组 |
| <= | 小于或等于 | 枚举与整数，及对应的一维数组 |
| >= | 大于或等于 | 枚举与整数，及对应的一维数组 |

## 3）．算术运算符

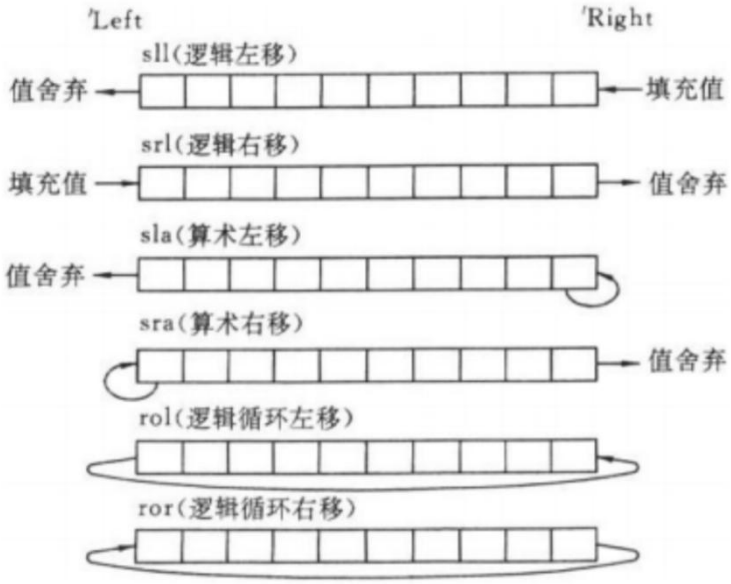
算术运算符包括一些基本的算术运算，使用算术运算符需要注意的是乘方（**）运算的右边必须为整数。VHDL 的算术运算符如表 8 所示。

表 8 VHDL 算术运算符

| 操作符 | 功能 | 操作数据类型 |
|---|---|---|
| + | 加 | 整数 |
| - | 减 | 整数 |
| * | 乘 | 整数和实数（包括浮点数） |
| / | 除 | 整数和实数（包括浮点数） |
| ** | 乘方 | 整数 |
| ABS | 取绝对值 | 整数 |
| MOD | 取模 | 整数 |
| REM | 取余 | 整数 |

## 4）．移位运算符

移位运算符为二元运算符，左边必须为一维数组，且元素类型为 bit 或 boolean 类型。右边运算数为整数，可以为负数，相当于反方向移位。一位移位与循环移位的语义示意如图 5 所示。



VHDL 的移位运算符如表 9 所示。

| 操作符 | 功能 | 操作数据类型 |
| --- | --- | --- |
| SLL | 逻辑左移 | BIT 或布尔型一维数组 |
| SRL | 逻辑右移 | BIT 或布尔型一维数组 |
| SLA | 算术左移 | BIT 或布尔型一维数组 |
| SRA | 算术右移 | BIT 或布尔型一维数组 |
| ROL | 逻辑循环左移 | BIT 或布尔型一维数组 |
| ROR | 逻辑循环右移 | BIT 或布尔型一维数组 |

除了上面介绍的，VHDL 中运算符还包括正号“+”、负号“-” 以及“&”。其中，**连接符号（&）**用于一维数组，这个数组的元素个数可以为 1，运算结果为右边数组连接在左边数组之后形成新数组，例如：

```
sel <= a & b;
```

假设 a 信号为“0”，b 信号为“1”，那么得到的 sel 信号就是“01”。

# 算符优先级

所有的 VHDL 运算符之间都有优先级的关系，各运算符优先级从最高到最低，顺序如表 10 所示（同一行优先级相同）。

表 10 运算符的优先顺序

| 符号 | 优先级 |
| --- | --- |
| ** ABS NOT | 最高优先级 ↑ |
| * / MOD REM | |
| +（正号） -（负号） | |
| + - & | |
| SLL SLA SRL SRA ROL ROR | |
| = /= < <= > >= | |
| AND OR NAND NOR XOR XNOR | 最低优先级 |

# 四、 VHDL 语言的描述语句

使用 VHDL 进行数字电路描述时候，如果按照执行顺序对 VHDL 的程序进行分类，可以分为**顺序（sequential）描述语句和并行（concurrent）描述语句**。顺序语句描述的程序总是按照程序书写的顺序执行；而并行语句都是同时执行的，和程序的书写顺序无关。

## 4.1 VHDL 顺序语句描述方法

VHDL 中的顺序语句一般在进程中出现，或者以函数、过程的方式在进程中被调用。顺序语句所涉及到的系统行为有时序流、控制、条件和迭代等。

VHDL 中的顺序语句有 **WAIT 语句、断言语句、IF 语句、CASE 语句、LOOP 语句、NEXT 语句、过程调用语句和 NULL 语句**，下面就对它们进行详细介绍。

### 1). WAIT 语句

WAIT 语句允许把一个顺序执行的进程或子程序挂起，挂起的进程或子程序恢复的条件由 3 种不同的方法指定。WAIT 语句可以有不同的格式，分别有不同的作用，例如 WAIT ON 表示等待到信号变化，WAIT UNTIL 表示等到一个表达式为真，而 WAIT FOR 表示等待一个固定的事件，如果仅仅写一个 WAIT 的话就表示无限期的等待。

WAIT 语句能用于多种不同的目的，常用于为综合工具指定时钟输入。另一用途是将进程的执行延时一段时间或者是为了动态地修改进程敏感表。

为了避免无休止的等待可以加一个**超时付句**，不管进行到哪儿或是条件有没有满足都允许执行超时处理。下面的代码就演示了 WAIT UNTIL 语句的使用方法和超时处理的方法：

```
WAIT    UNTIL (sendB = '1') FOR 1 ns;
        ASSERT (sendB = '1')
        REPORT "sendB timed out at '1'"
        SEVERITY ERROR;
```

### 2). 断言（ASSERT）语句

断言语句的功能是为设计者报告一个文本字符串。断言语句包含一个布尔表达式，表达式为真，该语句不做任何事；反之，它将输出一用户规定的字符串到标准输出终端。

断言语句规定输出字符串的严重程度为 4 个级别（NOTE、WARNING、ERROR 和 FAILURE），它们的意思分别是注意、警告、错误和失败，严重层次递增。

断言语句的格式如下：

```
ASSERT_STATEMENT ::=
    ASSERT CONDITION
    [REPORT EXPRESSION]
    [SEVERITY EXPRESSION];
```

其中，关键字 ASSERT 后跟 CONDITION 布尔值表达式，它的条件决定 REPORT 付句规定的文字表达式输出不输出，如果是假，文字表达式输出，如果是真，该文字表达式不输出。

**REPORT 付句**
允许设计者指定输出文字表达式的值，如果不指定 REPORT 语句，默认值是 ASSERTION VIOLATION

**SEVERITY 付句**
允许设计者指定断言语句的严重级别，如果没指定 SEVERITY 付句，其默认值是 ERROR。

下面是一个断言语句的使用实例，它表示对输入时钟进行检查，如果其建立时间小于 20ns，则输出 ERROR 信号：

```
PROCESS (clk,din)
    VARIABLE last_d_change :TIME := 0 ns;
    VARIABLE last_d_value :std_logic := 'X';
    VARIABLE last_clk_value :std_logic := 'X';

BEGIN
    IF (last_d_value /= din) THEN    --结束时钟
        last_d_change := NOW;
        last_d_value := din;
    END IF;

    IF (last_clk_value /= clk) THEN
        last_clk_value := clk;

        IF (clk = '1') THEN
            -- 断言语句
            ASSERT (NOW - last_d_change >= 20 ns)
            REPORT "setup violation"
            SEVERITY WARNING;
          END IF;
       END IF;
    END PROCESS;
```

## 3). IF 语句

IF 语句是根据所指定的条件来确定执行哪些语句，其格式如下：

```
IF condition THEN
    sequence_of_statements
ELSIF condition THEN
```

```
    sequence_of_statements
ELSE
    sequence_of_statement
END IF;
```

IF 语句用关键字 IF 开头和用关键字 END IF 结尾，END IF 分开拼写。有两个可选付句（ELSIF 付句和 ELSE 付句），**ELSIF 付句可重复并允许有多个 ELSIF 付句，可选 ELSE 付句但只允许有一个 ELSE 付句**。付句中的条件是一布尔表达式，如条件为真值，则下一语句被执行；如果条件不为真，那么接着执行跟在 ELSE 付句后的顺序语句。
下面举一个 IF 语句的使用例子，如下：

```
IF (day = sunday) THEN
    weekend := TRUE;
ELSIF (day = saturday) THEN
    weekend := TRUE;ELSE
    weekday := TRUE;
END IF;
```

以上代码的意义如下：有两个变量 weekend 和 weekday，每当 day 等于 saturday 或 sunday 时变量 weekend 变为真，执行跟着的下一句并控制转到跟在 END IF 之后的语句，否则转到 ELSIF 语句部分并检查 day 是否为 Saturday；当变量 day 等于 saturday，执行跟着的下一句并再次控制转到跟在 END IF 之后的语句；若 day 并不等于 sunday 或 saturday，执行 ELSE 语句部分。

## 4). CASE 语句

当单个表达式的值在多个起作用的项中选择时用 CASE 语句。CASE 语句的格式如下：

```
CASE expression IS
    WHEN choice1 =>
            sequence_of_statements
    WHEN choice2 | choice3 =>
            sequence_of_statements
        …
    WHEN OTHERS =>
            sequence_of_statements
    END CASE;
```

下面是一个使用 CASE 语句执行处理器指令的例子：

```
CASE instruction IS
    WHEN load_accum =>
            accum <= data;
    WHEN store_out =>
```

```
            data_out <= accum;
    WHEN load|store =>
            process_IO(addr);
    WHEN OTHERS =>
            process_error(instruction);
END CASE
```

## 5). 循环（LOOP）语句

当需要重复操作时用循环语句，或者实现的模块需要很强的迭代能力时用循环语句：

```
[循环标示 :] [循环条件] LOOP
    顺序处理语句
END LOOP[LOOP_label];
```

其中循环条件可以用 WHILE 语句或者 FOR 语句来描述。

**WHLIE 语句**

有一个循环控制的条件 condition，只要条件表达式为真，WHILE 循环语句就一直执行下去，除非要退出循环。例如：

```
WHILE (day = weekday) LOOP
    day := get_next_day(day);
END LOOP
```

**FOR 循环**

for 循环是根据预先的设定进行迭代，所指定的范围并不一定必须为整数值，也可以表示成一个子类型的指示或者一个范围语句，例如：

```
PROCESS (clk)
    TYPE day_of_week IS (sun,mon,tue,wed,thur,fri,sat);
    BEGIN
        FOR i IN day_of_week LOOP
            IF i = sat THEN
                son <= mow_lawn;
            ELSEIF i = sun THEN
                church <= family;
            ELSE
                dad <= go_to_work;
            END IF;
    END LOOP;
END PROCESS;
```

FOR LOOP 语句的指数值（i）由 FOR 语句局部地说明，这和进程、函数和过程中变量 I 不是一会事，它不需要显式地说明，由于 FOR LOOP 语句的虚拟性，循环指数要局部说明之。这样在进程、函数或过程中存在同名变量时，它们会被分别处理并由它们的内含寻址。

此外，关于循环需要特别注意的是，在某些编程语言中循环指数的值可由赋予内部循环值来改变，但是 VHDL 中是不允许对循环指数的任何赋值，这排除了在任何函数返回值中或在过程的输出与双向参量中存在循环指数。

## 6). NEXT 语句

如果必须在这次迭代或循环中停下正在执行的语句，而转向下一个迭代时，用 NEXT 语句。执行 NEXT 语句时，模块处理停在当前点并转到循环语句的开始。随着循环的第一个语句执行，循环变量增加一个迭代值，直到迭代的限制值，循环停止。
下面是一个 NEXT 语句使用的例子：

```
PROCESS(A,B)
    CONSTANT max_limit :INTEGER := 255;
    TYPE d_type IS ARRAY (0 to max_limit) OF BOOLEAN ;
    VARIABLE done : d_type;
BEGIN
    FOR i IN 0 TO max_limit LOOP
    IF (done(i) = TRUE ) THEN
        NEXT;
    ELSE
            done (i) := TRUE;
    END IF;
    q(i) <= a(i) AND b(i);
    END LOOP;
END PROCESS;
```

## 7). EXIT 语句

EXIT 语句提供完全停下循环执行的能力。执行期间发生了明显的错误或者所有的进程已执行完毕就跳出循环，EXIT 语句允许退出或跳出循环语句。执行 EXIT 语句后 EXIT 语句后面的语句暂停执行，去执行循环语句后面的语句。
EXIT 语句的基本书写格式如下：

```
EXIT [循环标号][WHEN 条件]
```

循环标号一般在多重循环中用于标明循环层次，如果 EXIT 语句后面添加循环标号，它将会退出循环标号指定的循环。“WHEN 条件”项用于表明 EXIT 语句执行的条件，此条件为真时才推出循环。
EXIT 语句的使用实例如下：

```
PROCESS (a)
```

```
BEGIN
    first_loop:FOR i IN 0 TO 100 LOOP
        second_loop:FOR j IN 1 TO 10 LOOP
            ......
            EXIT second_loop;
            ......
            EXIT first_loop;
        END LOOP;
    END LOOP;
END PROCESS;
```

## 4.2 VHDL 并行语句描述方法

VHDL 不仅仅提供了一系列的顺序语句，同样也提供了很多并行语句。在 VHDL 中，并行语句主要包括以下几种：

- **进程（PROCESS）语句；**
- **块（BLOCK）语句；**
- **并发信号赋值；**
- **条件信号赋值；**
- **选择信号赋值。**

其中进程语句和块语句已经在结构体的描述方法中介绍过了，在此不再累赘，后面主要介绍余下的 3 种并行语句。

### 1）．并发信号赋值<=

信号赋值就是使用**信号赋值操作符“<=”**修改一个信号的状态，如果此语句是在一个进程中，那么它是一个顺序语句，反之如果它是在进程外面（和进程并列关系），那么它就是一个并行赋值的语句。

下面是一个信号赋值的例子，其中 c1、c2 是顺序赋值的，c2 在 c1 之后赋值；d1 和 d2 是并行赋值的，它们同时被赋值：

```
ARCHITECTURE arch of demo is
BEGIN
-- 并行赋值
d1 <= din
d2 <= din
-- 进程
PROCESS(din)
BEGIN
```

```
-- 顺序赋值
    c1 <= din
    c2 <= din
END PROCESS;
END arch;
```

## 2）．条件信号赋值

条件信号赋值的格式如下：

```
目的信号 <= 表达式 1 WHEN 条件 1 ELSE
            表达式 2 WHEN 条件 2 ELSE
            表达式 3 WHEN 条件 3 ELSE
            …
            表达式 n;
```

最后一个表达式 n 表示以上 n-1 个条件都不满足时自动选用此表达式，如果有条件满足，则条件对应的表达式会计算赋值给目的信号量。条件信号代入语句也是并发描述语句，它可以根据不同条件将不同的多个表达式之一的值代入信号量。

下面通过一个四选一选择器的实现方法来介绍条件信号代入语句的使用方法：

```
ENTITY mux4 IS
PORT (
      din0, din1, din2, din3,sel0,sel1: in bit;
      dout: out bit );
END mux4;

ARCHITECTURE arch of mux4 is
SIGNAL sel : bit_vector(1 downto 0);
BEGIN
    sel   <=   sel1 & sel0;
    dout <= din0 WHEN sel = “00” ELSE
            din1 WHEN sel = “01” ELSE
            din2 WHEN sel = “10” ELSE
            din3 WHEN sel = “11” ELSE
            ‘X’;
END mux4;
```

## 3）．选择信号赋值

选择信号赋值类似于 CASE 语句，它的格式如下：

```
WITH 表达式 SELECT
目的信号量 <= 表达式 1 WHEN 条件 1;
              表达式 2 WHEN 条件 2;
              表达式 3 WHEN 条件 3;
             …
              表达式 n WHEN 条件 n;
```

如果使用选择信号赋值实现上面的四选一选择器，代码如下：

```
ENTITY mux4 IS
PORT (
        din0, din1, din2, din3,sel0,sel1: in bit；
        dout: out bit )；
END mux4；
ARCHITECTURE arch of mux4 is
SIGNAL sel : bit_vector(1 downto 0)；
BEGIN
      sel <= sel1 & sel0；
      WITH sel SELECT
      dout <= din1 when “00”,
      dout <= din1 when “01”,
      dout <= din2 when “10”,
      dout <= din3 when “1”,
      ‘X” WHEN OTHERS；
      END mux4；
```

# 五、 VHDL 语言的预定义属性

在 VHDL 中，属性是指关于设计实体、结构体、类型、信号等项目的制定特征，利用属性可以使得 VHDL 代码更加简明扼要、易于理解。

VHDL 提供了下面 5 类预定义属性：值**类属性、函数类属性、信号类属性、数据类型类属性和数据范围类属性**。

## 5.1 值类预定义属性

值类属性返回有关数组类型、块和常用数据类型的**特定值**，值类属性还用于返回数组的长度或者类型的最低边界，值类属性分成 3 个子类。

### 1）．值类型属性：返回类型的边界

值类型属性用来返回类型的边界，有 4 种预定义属性：

- T'LEFT 用于返回类型或者子类型的左边界；
- T'RIGHT 用于返回类型或者子类型的右边界；
- T'High 用于返回类型或者子类型的上限值；
- T'Low 用于返回类型或者子类型的下限值。

用字符“'”指定属性并后跟属性名，“'”前的对象是所附属性的对象，字首大写“T”指所附属性的对象是类型（TYPE），“'”字符标点符号（tick）是 VHDL 特有的标号。

### 2）．值类数组属性：返回数组长度

值类数组属性只有一个，即 LENGTH，该属性返回指定数组范围的总长度，它用于带某种标量类型的数组范围和带标量类型范围的多维数组。

### 3）．值类块属性：返回块的信息（元件具体装配语句）

用属性'STRUCTURE 和'BEHAVIOR 返回有关在块和结构体中块是如何建模的信息。在块和结构体中如不含**元件具体装配语句**，则属性'BEHAVIOR 将返回真值，如果块或者结构体中只含元件具体装配语句或被动进程，则属性'STRUCTUTE 将返回真值。

## 5.2 函数类预定义属性

函数类属性为设计者返回**类型、数组和信号信息**。用函数类属性时，函数调用由输入变元的值返回一个值，返回值为可枚举值的位置号码、在一个△时间内信号是否改变的指示或者一个数组的边界。函数类属性可细分为 3 个常见的类别。

## 1）．函数类型属性：返回类型值

函数类型属性返回类型内部值的位置号码、返回特定类型输入值的左和右边的值，函数类型属性分为 6 种：

- 'POS（value）返回传入值的位置号码；
- 'VAL（value）返回从该位置号码传入的值；
- 'SUCC（value）返回输入值后类型中的下一个值；
- 'PRED（value）返回输入值前类型中的原先的值；
- 'LEFTOF（value）表示立即返回一个值到输入值的左边；
- 'RIGHTOF（value）表示立即返回一值到输入值的右边。函数类型属性主要用于从可枚举数或物理类型的数转换到整数类型。

## 2）．函数数组属性：返回数组的边界

函数数组类属性返回数组类型的边界，分 4 类：

- 数组'LEFT（n）返回指数范围 n 的左边界；
- 数组'RIGHT（n）返回指数范围 n 的右边界；
- 数组'HIGH（n）返回指数范围 n 的上限值；
- 数组'LOW（n）返回指数范围 n 的下限值。值类数组属性只有一个即 LENGTH，该属性返回指定数组范围的总长度，它用于带某种标量类型的数组范围和带标量类型范围的多维数组。

## 3）．函数信号属性：返回信号历史信息

函数信号属性用来返回有关信号行为功能的信息，例如报告究竟一个信号是否正好有值的变化，报告从上次事件中跳变过了多少时间以及该信号原来的值是什么。

函数信号属性有如下 5 类：

- S'EVENT，如果当前的△时间期间发生了事件返回真，否则返回假（信号是否有值的变化）；
- S'ACTIVE，如果在当前的△时间期间做了事项处理返回真，否则返回假；
- S'LAST_EVENT，返回从信号原先事件的跳变至今所经历的时间；
- S'LAST_VALUE，返回在上一次事件之前 S 的原先值；
- S'LAST_ACTIVE，返回自信号原先一次的事项处理至今所经历的时间。

## 5.3 信号类预定义属性

信号类属性用于根据另一个信号**创建一些专用的信号**，由类专用信号为设计者返回有关所附属性的信号信息（在一指定时间范围内该信号是否已经稳定的信息、在信号上有无事项处理的信息和建立的信号的延迟形式）。

对这类信号是不能在子程序内部使用的，返回的信息和由某种函数属性所提供的功能非常类似，区别是这类专用信号用于正常信号能用的任何场合，包括在敏感表中。有如下的 4 类属性：

- S'DELAYED[（time）] 建立和参考信号同类型的信号，该信号后跟参考信号和延时可选时间表示式的时间。'DELAYED 属性为信号建立延迟的版本并附在该信号上，它和传输延时信号赋值的功能相同，但简单。
- S'STABLE[（time）] 在选择时间表达式指定的时间内参考信号无事件发生时，属性建立为真值的布尔信号。
- S'QUIET[（time）] 参考信号或所选时间表达式指定时间内没事项处理时，属性建立一个为真值的布尔信号。
- S'TRANSACTION 信号上有事件发生或为每个事项处理而翻转它的值时，该属性建立一个 BIT 类型的信号。

## 5.4 数据类型类预定义属性

数据类型类的属性只有一个 t'BASE 类型属性，它必须由另一个值或函数类型属性用该属性。这个属性将返回类型或者子类型的基本类型，这个属性只能作另一属性的前缀。

## 5.5 数据范围类预定义属性

数据范围类属性返回数组类型的范围值，并由所选的输入参数返回指定的指数范围，这种属性标记如下：a'RANGE[（n）];a'REVERSE_RANGE[（n）]。

属性 RANGE 将返回由参数 n 值指明的第 n 个范围和按指定排序的范围，'REVERSE_RANGE 将返回按逆序的范围，属性'RANGE 和'REVERSE_RANGE 也用于控制循环语句的循环次数。

REVERSE_RANGE 属性的用法和 RANGE 属性相类似，只是它按逆序返回一范围而已。比如假设'RANGE 属性是返回 0 到 15，那么'REVERSE_RANGE 属性返回 15 下降到 0。