## 第二章 VHDL语言基础

一个完整的VHDL语言程序通常包含实体（Entity）、构造体（Architecture）、配置（Configuration）、包集合（Package）和库（Library）5个部分。实体用于描述所设计的系统的外部接口信号；构造体用于描述系统内部的结构和行为；包集合存放各设计模块都能共享的数据类型、常数和子程序等；配置用于从库中选取所需单元来组成系统设计的不同版本；库存放已经编译好的实体、构造体、包集合和配置。本篇对上述VHDL设计的主要构成做一详细介绍。

第一节概述了VHDL的相关知识。第二节介绍了其基本结构——实体、构造体、子结构以及库、包集合等。第三节讲述了数据类型和运算操作符。第四节举例说明了VHDL语言的基本语句，包括并发的和顺序的描述语句。

### 第一节 概述

硬件描述语言（Hardware Description Language），顾名思义，就是描述硬件电路功能、信号连接关系及定时关系的语言。它能比电路原理图更有效地表示硬件电路的特性。利用硬件描述语言可以进行数字电子系统的设计。

国外硬件描述的语言种类很多，在我国比较有影响的大概有3种硬件描述语言：VHDL语言、Verilog HDL语言和AHDL语言。前两种已成为IEEE标准语言。

电子设计自动化（Electronic Design Automatic，EDA）技术的理论基础、设计工具、设计器件的关系是这样的：设计师用硬件描述语言HDL描绘出硬件的结构和硬件的行为，用设计工具将这些描述综合映射成与半导体工艺有关的硬件工艺文件，半导体器件FPGA则是这些硬件工艺文件的载体。当这些FPGA器件加载，配置上不同的工艺文件时，这个器件便具有了相应的功能。在这一系列的设计、综合、仿真、验证、配置的过程中，现代设计理论和现代设计方法始终贯穿其中。

以HDL语言表达设计意图，FPGA作为硬件载体，计算机为设计开发工具，EDA软件为开发环境的现代电子设计方法日趋成熟。软件硬件协同开发缩短了电子设计周期，加速了电子产品更新换代步伐。毫不夸张地说：“EDA是电子产业的心脏起搏器，是电子产业飞速发展的原动力。”

**1.1 VHDL硬件描述语言的诞生**

VHDL语言的英文全称是Very High Speed Integrated Circuit Hardware Description Language，即超高速集成电路硬件描述语言。在利用EDA工具进行电子设计时，逻辑图、分立电子原件作为整个越来越复杂的电子系统的设计已不适应。任何一种EDA工具，都需要一种硬件描述语言来做为 EDA工具的工作语言。这些众多的EDA工具软件开发者，各自推出了自己的HDL语言。

由美国政府牵头，VHDL工作小组于1981年6月成立，提出了一个满足电子设计各种要求的能够作为工业标准的HDL。1983年第三季度，由IBM公司、TI公司、Intermetrics公司签约，组成开发小组，工作任务是提出语言版本和开发软件环境。1986年IEEE标准化组织开始工作，讨论VHDL语言标准，历时一年有余，于1987年12月通过标准审查，并宣布实施，即IEEE STD 1076－1987。1993年VHDL重新修订，形成新的标准即IEEE STD 1076－1993。从此以后，美国国防部实施新的技术标准，要求电子系统开发商的合同文件一律采用VHDL文档。即第一个官方VHDL标准得到推广、实施和普及。

**1.2 电子器件的发展**

**1.2.1 CPLD或EPLD芯片**

这类器件的使用特点是：芯片一经上电加载即已完成编程，不必在每次上电时重新进行加载，也就是当程序烧入芯片后，只有下一次需要再修改程序时才需要对芯片进行重新加载，否则将不会改变先前所烧入的代码，类似于大的GAL芯片。例如，Lattice的ispGAL芯片可以在系统进行编程加载程序，也可以与该公司的ispLSI芯片在同一个电路中加载程序。CPLD在结构上主要由可编程逻辑宏单元LMC（Logic Macro Cell）围绕中心的可编程互连矩阵单元组成，其中LMC的逻辑结构比较复杂，并具有复杂的I/O单元互连结构，可以由用户根据设计的需要生成特定的电路结构，完成一定的功能。

**1.2.2 FPGA芯片**

这类器件不像CPLD或EPLD，每次上电使用时不管是否改变程序都需要对芯片进行加载。现在FPGA芯片的容量一般比CPLD或EPLD的容量大得多，更适合于做较大系统的复杂设计。FPGA芯片通常包含3类可编程资源：可编程的逻辑功能块、可编程I/O块和可编程的内部互连。可编程的逻辑功能块是实现用户功能的基本单元，它们通常排成一个阵列，遍布于整个芯片中；可编程I/O块完成芯片上的逻辑与外部封装管脚的接口，常围绕着阵列排列于芯片四周；可编程的内部互连包括各种长度的连接线段和一些可编程连接开关，它们将各个可编程逻辑块或I/O块连接起来，构成特定功能的电路。

Xilinx的XC系列器件采用的是现场可编程门阵列FPGA，现场可编程门阵列是一种类似门阵列的结构，它的基本单元以阵列的形式排列在芯片上，但它不像门阵列那样由连线掩膜确定其最终的逻辑功能，而是将规则的连线阵列也已做好，其逻辑功能由各连线节点的控制开关的通断来确定。这些节点的控制开关的值有各种不同的控制方法，如静态随机存储器控制（SRAM）、反熔丝（antifuse based）控制以及由闪存（Flash）控制等几大类。

SRAM控制的FPGA器件是用静态随机存储器中存储的数值来控制芯片中可编程节点的通断，以实现芯片的设计功能。这种FPGA在使用时，需要在系统加电时首先进行功能初始化，将存储器的内容加载到芯片的控制器中。

反熔丝控制的FPGA器件是用反熔丝单元来控制可编程器件内的可编程节点的通断，使芯片中每一部分具有应有的逻辑功能，以实现器件的设计功能。反熔丝单元是一个被动的两端器件，通常情况下处于开路状态，在施加充足的电压后，能够永久的导通。用于现场可编程器件的反熔丝其占用芯片面积非常小，速度也很高，这方面的性能几乎可以与门阵列相比。但这种FPGA在编程时需要专用的编程设备，而且芯片功能一经编程确定后，不能再修改，反熔丝控制的FPGA器件价格最低。

Flash控制的FPGA器件是用闪存的数值来控制FPGA节点的通断，实现现场可编程目的。这种FPGA的工作特性与SRAM-FPGA相似，但与SRAM-FPGA相比，它的单元面积小。由于Flash具有不挥发性，使用时不必在每次上电时都重新进行功能加载，所以使用较方便。

### 第二节 VHDL语言的基本结构

一个VHDL程序包含实体（Entity）、结构体（Architecture）、配置（Configuration）、包集合（Package）、库（Library）五部分。

其中实体是一个VHDL程序的基本单元，由实体说明和结构体两部分组成。实体说明用于描述设计系统的外部接口信号；结构体用于描述系统的行为、系统数据的流程或者系统的组织结构形式。配置用于从库中选取所需单元来组成系统设计的不同规格的不同版本，使被设计系统的功能根据需要发生变化。包集合存放各设计模块能共享的数据类型、常数、子程序等。库用于存放已编译的实体、构造体、包集合、配置。总的来说有两种库，一种是用户自行生成的IP库，有些集成电路设计中心开发了大量的工程软件，有不少好的设计范例，可以重复引用，所以允许用户自行建库是专业EDA公司的重要任务之一。另一类是PLD，ASIC芯片制造商提供的库。比如常用的74系列芯片，RAM、ROM控制器，Counter计数器等标准模块。用户可以直接引用，不必从头编写。

**2.1 VHDL语言的基本单元**

VHDL语言设计的基本单元就是VHDL语言的一个基本设计实体（Entity）。一个实体，简单的可以是一个与门，复杂点的可以是一个微处理器或一个系统。但是，不管是简单的数字电路，还是复杂的数字电路，其基本构成是一致的，都由实体说明（Entity Declaration）和构造体（Architecture Body）两部分构成。如前所述，实体说明部分规定了设计单元的输入输出接口信号或引脚，而构造体部分定义了设计单元的具体构造和行为。例1示出了二选一电路的VHDL描述。由例1可以看出，实体说明是二选一器件外部引脚的定义；而构造体则描述了二选一器件的逻辑电路和逻辑关系。

例1 二选一电路的VHDL描述

LIBRARY IEEE；

USE IEEE.STD\_LOGIC\_1164.ALL；

-- ENTITY DECLARATION

ENTITY mux IS

PORT (d0，d1，sel：IN STD\_LOGIC；

q：OUT STD\_LOGIC)；

END mux；

-- ARCHITECTURE BODY

ARCHITECTURE connect OF mux IS

SIGNAL tmp：STD\_LOGIC；

BEGIN

PROCESS(d0，d1，sel)

BEGIN

tmp <= (d0 AND sel) OR (d1 AND (NOT sel))；

q <= tmp；

END PROCESS；

END connect；

下面以本段程序为例，说明一下实体和构造体的书写规定。

**2.1.1 实体说明**

任何一个基本设计单元的实体说明都具有如下的结构：

ENTITY 实体名 IS

【类属参数说明】；

【端口说明】；

END 实体名；

一个基本设计单元的实体说明以“ENTITY 实体名 IS”开始至“END 实体名”结束。例如在上面例子中从“ENTITY mux IS”开始至“END mux”结束。这里大写字母表示实体说明的框架，即每个实体说明都应这样书写，是不可缺少和省略的部分。小写字母是设计部分，随设计单元不同而不同。实际上，VHDL不区分大小写，这里仅仅是为了阅读方便而加以区分的。

1．类属参数说明

类属参数说明必须放在端口说明之前，用于指定参数，例1中没体现类属说明语句，具体书写范例如下：

GENERIC ( m: TIME: = 1 ns)

该语句指定了m的值为1。

2．端口说明

端口说明是对基本设计实体与外部接口的描述，也可以说是对外部引脚信号的名称、数据类型和输入、输出方向的描述。其书写格式如下：

PORT（端口名｛，端口名｝：方向 数据类型名；

┇

端口名｛，端口名｝：方向 数据类型名）；

1）端口名

端口名是赋予每个外部引脚的名称，通常用一个或几个英文字母，或者用英文字母加数字命名之。例如例1中的外部引脚为d0，d1，sel，q。

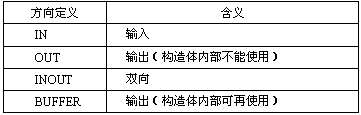
2）端口方向

端口方向用来定义外部引脚的信号方向是输入还是输出。例如，例1中的d0，d1，sel为输入引脚，故用方向说明符“IN”说明之，而q则为输出引脚，用方向说明符“OUT”说明之。

凡是用“IN”进行方向说明的端口，其信号自端口输入到构造体，而构造体内部的信号不能从该端口输出。相反，凡是用“OUT”进行方向说明的端口，其信号将从构造体内经端口输出，而不能通过该端口向构造体内输入信号。

另外，“INOUT”用以说明该端口是双向的，可以输入也可以输出；“BUFFER”用以说明该端口可以输出信号，且在构造体内部也可以利用该输出信号。表示方向的说明符及其含义如表2.1所示。

表2.1 端口方向说明



注：OUT允许对应多个信号，BUFFER只允许对应一个

表2.1中“OUT”和“BUFFER”都可以定义为输出端口，但是它们之间是有差别的，如图2.1所示。



（a） （b）

图2 .1 OUT和BUFFER的区别

在图2.1（a）中，锁存器的输出端口被说明为“OUT”，而在（b）中，锁存器的输出被说明为“BUFFER”。从图中可以看到，如果构造体内部要使用该信号，那么锁存器的输出端必须说明为“BUFFER”，而不能用“OUT”说明。

图2.1（b）说明了，当一个构造体用“BUFFER”说明输出端口时，与其连接的另一个构造体的端口也要用“BUFFER”说明。对于“OUT”则没有这样的要求。

如下两例说明了端口方向的使用方法。

例2：并行口通信芯片8255，其端口定义为：

ENTITY i8255 IS

PORT（

reset，cs：IN STD\_LOGIC；

rd，wr：IN STD\_LOGIC；

a1，a0：IN STD\_LOGIC；

pa，pb：INOUT STD\_LOGIC\_VECTOR（7 DOWNTO 0）；

pcl，pch：INOUT STD\_LOGIC\_VECTOR（3 DOWNTO 0）；

d：INOUT STD\_LOGIC\_VECTOR（7 DOWNTO 0））；

END i8255；

根据电路定义，所有的控制信号和地址信号都是输入模式IN，而其3个端口以及与总线相连的数据端口都是双向的，因此定义为双向模式INOUT。

例3：定时/计数器芯片8253，其端口定义为：

ENTITY i8253 IS

PORT（

rd，wr，cs，a1，a0：IN STD\_LOGIC；

clk0，clk1，clk2：IN STD\_LOGIC；

gate0，gate1，gate2：IN STD\_LOGIC；

out0，out1，out2：BUFFER STD\_LOGIC；

d：INOUT STD\_LOGIC\_VECTOR（7 DOWNTO 0））；

END i8253；

与8255一样，8253的控制与地址信号是输入模式，而连接总线的数据端口是双向模式。由于计数器的输出out要被用来决定下一个状态，而这个输出又不能连接别的输出或双向瑞口，因此将其定义成缓冲端口。

3）数据类型

在VHDL语言中有10种数据类型，但是在逻辑电路设计中只用到两种：STD\_LOGIC和STD\_LOGIC\_VECTOR。

当端口被说明为STD\_LOGIC数据类型时，该端口的信号取值只可能是“l”或“0”。这里的“1”和“0”是指逻辑值，也就是说STD\_LOGIC数据类型是位逻辑数据类型，其取值只能是两个逻辑值（“1”和“0”）中的一个。

当端口被说明为STD\_LOGIC\_VECTOR数据类型时，该端口的取值可能是一组二进制位的值。例如，某一数据总线输出端口，具有8位的总线宽度。那么这样的总线端口的数据类型可以被说明成STD\_LOGIC\_VECTOR。总线端口上的值由8位二进制位的值所确定。较完整的端口说明下例所示。

例4：PORT（d0，d1，sel：IN STD\_LOGIC；

q：OUT STD\_LOGIC；

bus：OUT STD\_LOGIC\_VECTOR（7 DOWNTO 0）；

该例中d0，d1，sel，q都是STD\_LOGIC数据类型，而bus是STD\_LOGIC\_VECTOR类型，（7 DOWNTO 0）表示该bus端口是一个8位端口，由B7－B0 8位构成。位矢量长度为8位。

**2.1.2 构造体**

构造体是一个基本设计单元，它具体地指明了该基本设计单元的行为、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。构造体对其基本设计单元的逻辑关系可以用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，而构造体的结构是完全一样的。由于构造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构造体进行编译。如果实体需要重新编译，那么相应构造体也应重新进行编译。

一个构造体的具体结构描述如下：

ARCHITECTURE 构造体名 OF 实体名 IS

[定义语句] 内部信号，常数，数据类型，函数等的定义；

BEGIN

[并行处理语句]；

END 构造体名；

一个构造体从“ARCHITECTURE 构造体名 OF 实体名 IS”开始，至“END 构造体名”结束。下面对构造体的有关内容和书写方法作一说明。

1．构造体名称的命名

构造体的名称是对本构造体的命名，它是该构造体的唯一名称。OF后面紧跟的实体名表明了该构造体所对应的是哪一个实体，用IS来结束构造体的命名。

2．定义语句

定义语句位于ARCHITECTURE和BEGIN之间，用于对构造体内部所使用的信号、常数、数据类型和函数进行定义。例如：

ARCHITECTURE behav OF mux IS

SIGNAL nesl：STD\_LOGIC；

┇

BEGIN

┇

END behav；

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。因它是内部连接用的信号，故没有也不需有方向的说明。

3．并行处理语句

并行处理语句处于语句BEGIN和END之间，这些语句具体地描述了构造体的行为及其连接关系。在构造体中的语句都是可以并行执行的，也就是说，语句的执行不以书写的语句顺序为执行顺序。

**2.2 VHDL构造体的子结构**

在规模较大的电路设计中，全部电路都用唯一的一个模块来描述是不方便的。为此，电路设计者总希望将整个电路分成若干个相对比较独立的模块来进行电路的描述。这样，一个构造体可以用若干个子结构，即相对比较独立的模块来构成。VHDL语言可以有以下3种形式的子结构描述语句：

★ BLOCK语句结构

★ PROCESS语句结构

★ SUBPROGRAMS结构

其中，process语句结构最常用。下面对上述3种子结构逐一说明。

**2.2.1 BLOCK语句结构描述**

人们在用计算机电路辅助设计工具输入电原理图时，往往将一个大规模的电原理图分割成多张子原理图，进行输入和存档。同样在VHDL语言中也不例外，电路的构造体对应整个电原理图，而构造体可以由多个BLOCK块构成，每一个BLOCK块对应一张子原理图。这样电原理图的分割关系和VHDL语言程序中用BLOCK分割构造体的关系是—一对应的。

在用其它高级语言编程时，总希望程序模块小一点，以利于编程和查错，也利于实现积木化结构。同理，在VHDL语言中采用BLOCK 语言对编程、查错、仿真及再利用都会带来莫大的好处。

1．BLOCK语句的结构

采用BLOCK语句描述局部电路的书写格式如下所示：

块结构名：

BLOCK

BEGIN

┇

END BLOCK块结构名；

如果采用BLOCK语句来描述二选一电路，那么用VHDL语言就可以书写为：

例4：

LIBRARY IEEE；

USE IEEE.STD\_LOGIC\_1164.ALL；

-- ENTITY DECLARATION

ENTITY mux IS

PORT (d0，d1，sel：IN STD\_LOGIC；

q：OUT STD\_LOGIC)；

END mux；

-- ARCHITECTURE BODY

ARCHITECTURE connect OF mux IS

SIGNAL tmp：STD\_LOGIC；

BEGIN

blk：BLOCK

BEGIN

tmp <= (d0 AND sel) OR (d1 AND (NOT sel))；

q <= tmp；

END BLOCK blk；

END connect；

上述程序的构造体中只有一个BLOCK块，如果电路复杂时可以由几个BLOCK块组成。

2．BLOCK内语句的并发性

在对程序进行仿真时，BLOCK语句中所描述的各个语句是可以并发执行的，它和书写顺序无关。在VHDL语言中将这样可以并行执行的语句称为并发语句（Concurrent Statement）。当然在构造体内直接书写的语句也是并发的。在VHDL语言中也存在只能顺序执行的语句，如process语句。

3．Guarded BLOCK

前面所述的BLOCK语句仅仅是将构造体划分成几个独立的程序模块，这和执行控制没有直接关系,在系统仿真时BLOCK语句将被无条件地执行。但是，在实际电路设计中，往往会碰到这样的情况，当满足条件时，BLOCK语句才可以被执行；条件不满足时BLOCK语句不能执行。这就是Guarded BLOCK，它可以实现BLOCK语句的执行控制。

例如，现在用BLOCK语句来描述一个锁存器的结构。该锁存器是一个D触发器，具有一个数据输入端d、时钟输入端clk、输出端q和反相输出端qb。众所周知，只有clk有效时（clk＝“1”），输出端q和qb才会随D端输入数据变化而变化。此时，用Guarded BLOCK语句描述该锁存器结构的VHDL语言程序可以书写为：

ENTITY latch IS

PORT（ d，clk：IN STD\_LOGIC；

q，qb： OUT STD\_LOGIC）；

END latch；

ARCHITECTURE latch\_guard OF latch IS

BEGIN

G：

BLOCK（cIk ＝＝’1’）

BEGIN

q <= GUARDED d；

qb <= GUARDED NOT（d）；

END BLOCK G；

END latch\_guard；

如上述程序所示，Guarded BLOCK语句的格式为：

BLOCK [布尔表达式]

当布尔表达式为真时（例中clk=’1’为真时），该BLOCK语句被启动执行；而当卫式表达式为假时，该BLOCK语句将不被执行。

在BLOCK块中的两个信号赋值语句都写有关键词GUARDED，这表明只有布尔表达式为真时，这两个语句才被执行。

现在根据程序描述一下锁存器的工作过程。当端口clk的值为“1”时，布尔表达式为真。d端的输入值从q端输出，并且对d端的值取反后从qb端输出。当端口clk的值为“0”时，d端到q、qb端的信号传递通道被切断，q端和qb端的输出保持原状，不随d端值的变化而改变。

**2.2.2 进程（PROCESS）语句结构描述**

1．PROCESS语句的结构

采用PROCESS语句描述电路结构的书写格式如下：

[进程名] PROCESS（敏感信号1，敏感信号2，…）

BEGIN

┇

END PROCESS；

进程名可以有，也可以省略。PROCESS语句从PROCESS开始至END PROCESS结束。执行PROCESS语句时，通常带有若干个敏感信号量。这些信号量将在 PROCESS结构的语句中被使用。用PROCESS语句结构描述的程序如下所示：

例5：

ENTITY mux IS

PORT（d0，d1，sel：IN STD\_LOGIC；

q：OUT STD\_LOGIC）；

END mux；

ARCHITECTURE connect OF mux IS

BEGIN

PROCESS（d0，d1，sel）

VARIABLE tmp1，tmp2，tmp3：STD\_LOGIC；

BEGIN

tmp1:= d0 AND sel；

tmp2:= dl AND（NOT sel）；

tmp3:= tmp1 OR tmp2；

q <= tmp3；

END PROCESS；

END connect；

程序中，tmp1、tmp2和tmp3是变量，变量只在进程内部定义，详见后面阐述。

2．进程（PROCESS）中语句的顺序性

在VHDL中，与BLOCK语句一样，某个功能独立的电路，在设计时也可以用一个PROCESS语句结构来描述。与BLOCK语句不同的是，PROCESS结构中的语句是按顺序一条一条向下执行的，而不像BLOCK中的语句可以并行执行。在VHDL语言中，这种顺序执行的语句只在PROCESS和SUBPROGRAM的结构中使用。

3．PROCESS的启动

在PROCESS的语句中总是带有1个或几个信号量。这些信号量是PROCESS的输入信号，书写时跟在“PROCESS”后面的括号中。例如PROCESS（d0，d1，sel）。该语句中d0、d1、sel都是信号量，在VHDL语言中也称敏感信号量。这些信号无论哪一个发生变化（如由“1”变为“0”或者由“0”变为“1”）都将启动PROCESS语句。一旦启动以后， PROCESS中的语句将从上到下逐句执行一遍。当最后一个语句执行完毕以后，就返回到开始的PROCESS语句，等待信号量下一次变化的出现。这样，只要PROCESS中指定的信号变化一次，该PROCESS语句就会执行一遍。

**2.2.3 SUBPROGRAM语句结构描述**

所谓SUBPROGRAM（过程和函数）就是在主程序调用它以后能将处理结果返回主程序的程序模块，其含义和其它高级语言中的子程序概念相当。它可以反复调用，使用非常方便。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。因此予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。

在VHDL中SUBPROGRAM有两种类型：

★ 过程（Procedure）

★ 函数（Function）

其中“过程”与其它高级语言中的子程序相当；而“函数”与其它高级语言中的函数相当。

1．过程语句

在VHDL语言中，过程语句的书写格式如下：

PROCEDURE 过程名（参数1，参数2，…） IS

［定义语句］； （变量等定义）

BEGIN

［顺序处理语句］； （过程的语句）

END过程名；

在PROCEDURE结构中，参数可以是输入也可以是输出。也就是说，过程中的输入输出参数都应列在紧跟过程名的括号内。例如，在VHDL语言中，将位矢量转换为整数的程序可以由一个过程语句来实现。

PROCEDURE vector＿to＿Int

（z：IN STD­\_LOGIC\_VECTOR；

x\_flag：OUT BOOLEAN；

q：INOUT INTEGER） IS

BEGIN

q:=0；

x\_flag:=FALSE；

FOR i IN z’ RANGE LOOP

q:=q\*2；

IF（z（i）= 1） THEN

q:=q+1；

ELSIF（z（i）／= 0） THEN

x\_flag:=TRUE；

END IF；

END LOOP；

END vector\_to\_int；

该过程调用后，如果x\_flag=TRUE，则说明转换失败，不能得到正确的转换整数值。

在上例中，z是输入，x\_flag是输出，q为输入输出。在没有特别指定的情况下，“IN”作为常数；而“OUT”和“INOUT”则看作“变量”进行拷贝。当过程的语句执行结束以后，在过程内所传递的输出和输入输出参数值，将拷贝到调用者的信号或变量中。此时输入输出参数如没有特别指定则按变量对待，将值传递给变量。如果调用者需要将输出和输入输出作为信号使用，则在过程参数定义时要指明是信号。例如

PROCEDURE shift（

din：IN STD\_LOGIC\_VECTOR；

SIGNAL dout：OUT STD\_LOGIC\_VECTOR）；

END shift；

前面已经提到，PROCESS结构中的语句是顺序执行的，那么在过程结构中的语句也是顺序执行的。调用者在调用过程前应先将初始值传递给过程的输入参数。然后过程语句启动，按顺序自上至下执行过程结构中的语句，执行结束，将输出值拷贝到调用者的“OUT”和“INOUT”所定义的变量或信号中。

2．函数语句

在VHDL语言中，函数语句的书写格式如下：

FUNCTION 函数名（参数1，参数2，…）

RETURN 数据类型名 IS

［定义语句］；

BEGIN

[顺序处理语句]；

RETURN [返回变量名]；

END［函数名］；

在VHDL语言中，FUNCTION语句中括号内的所有参数都是输入参数或称输入信号。因此在括号内指定端口方向的“IN”可以省略。FUNCTION的输入值由调用者拷贝到输入参数中，如果没有特别指定，在FUNCTION语句中按常数处理。

通常各种功能的FUNCTION语句的程序都被集中在包集合（Package）中。

上面详细叙述了过程、函数的结构和使用方法。为了能重复使用这些过程和函数，这些程序通常组织在包集合、库中。它们与包集合和库具有这样的关系：即多个过程和函数汇集在一起构成包集合（Package），而几个包集合汇集在一起就形成一个库（Library）。有关包集合和库的详细内容在下一节中介绍。但是，需要指出的是，不同公司发布的包集合和库的使用方法各不相同。

**2.3 库和包集合**

除了实体和构造体之外，包集合和库是在VHDL语言中另外3个可以各自独立进行编译的设计单元。

**2.3.1 库**

库（Library）是经编译后的数据的集合，存放包集合定义、实体定义以及构造体定义。库的功能类似于MS－DOS操作系统中的目录，库中存放设计的数据。在VHDL语言中，库的说明总是放在设计单元的最前面：

Library 库名；

这样，在设计单元内的语句就可以使用库中的数据。因此，库的好处就在于使设计者可以共享已经编译过的设计结果。在VHDL语言中可以存在多个不同的库，但库与库之间是独立的，不能互相嵌套。

1．库的种类

VHDL语言中的库分为设计库和资源库两类。

1）设计库

所谓设计库，即由VHDL标准规定的，对所有项目是默认可见的，不需要用Library、USE子句声明的库。STD库和WORK库是设计库。

STD：这个库中称为Standard的程序包中所有预定义的数据类型和函数都可以使用，为所有设计单元所共享、隐含定义、默认和“可见”。STD库中包含预定义的程序包STANDARD和TEXTIO，这两个程序包是使用VHDL语言时必须用到的工具。

WORK库：是VHDL语言的工作库，用户在设计项目中已经设计好的，正在校验、未仿真的中间件等都放在工作库中。实际上WORK是一个临时的仓库，用来存放成品或半成品。

也就是说，在每段VHDL程序中都隐含有下面的不可见的代码：

Library work；

Library std；

Use std.standard.all；

2）资源库

除了STD和WORK库以外，其他的库都为资源库。资源库用来存放常规元件和标准模块，使用时要预先声明。在有些库中，存放的元件、函数都是IEEE标准化组织认可的，称为IEEE库。VHDL工具厂商与EDA工具专业公司都有自己的资源库。

常用的IEEE资源库包含的程序包如下：

STD\_LOGIC\_1164

NUMERIC\_BIT

MATH\_REAL

2．库的使用

所有被编译的元件都存在设计库（Library）中，程序包通常也存在设计库中，不同库中程序包的个数也不相同，一个程序包中可以包含若干个子程序，子程序中又包含函数、过程、元件等基础的设计单元。尽管每个文件中可以存放多个单元，但为了更便于组织和管理，通常在一个文件中只存放一个实体说明和结构体对。用VHDL写程序时必须首先指定库和包的名字。在每个设计项目的开头声明选用的库名，用USE语句声明选中的逻辑单元。一个设计单元可以访问多个设计库，有些库被IEEE标准化组织认可，成为IEEE库，IEEE库存放了IEEE标准1076中的标准设计单元。

库语句和USE语句的语法形式是：

Library ＜库名＞

Use ＜库名＞．＜包名＞．a11；

例如：

LIBRARY IEEE；

USE IEEE.STD\_LOGIC\_1164.ALL；

**2.3.2 包集合**

数据类型、常量与子程序可以在实体说明部分和结构体部分加以说明，而且实体说明部分所定义的类型、常量及子程序在相应的结构体中是可见的（可以被使用）。但是，在一个实体的说明部分与结构体部分中定义的数据类型、常量及子程序对于其他实体的说明部分与结构体部分是不可见的。为了使一组类型说明、常量说明和子程序说明对多个设计实体都成为可见的，VHDL提供了包结构。包由包头和包体两部分组成。包头以保留字Packape开头，包体则以Package Body识别。实际上，标准库中存在的也是一些程序包。

程序包的语法是：

Package ＜包名＞ is

［外部子程序说明］

［外部常量说明］

［外部元件模板］

［外部类型说明］

［属性说明］

［属性指定］

end ［＜包名＞］

Package body ＜包名＞ is

［外部子程序体］

［内部子程序说明］

［内部子程序体］

［内部常量说明］

［内部类型说明］

end ［＜包名＞］

例如：

PACKAGE Logic IS

TYPE three\_level\_loglc IS（‘0’，‘1’，‘Z’）；

CONSTANT unknown\_value： three\_level\_logic：＝‘0’；

FUNCTION invert（input：three\_level\_loglc）

RETURN three\_level\_logic；

END logic；

PACKAGE BODY logic IS

FUNCTION invert（input：three\_level\_logic）

RETURN three\_level\_logic IS

BEGIN

CASE input IS

WHEN ‘0’＝＞ RETURN ‘1’；

WHEN ‘1’＝＞ RETURN ‘0’；

WHEN ‘ Z’＝＞ RETURN ‘ Z’；

END CASE；

END invert；

END logic；

一个程序包所定义的项对另一个单元并不是自动可见的。如果在某个VHDL单元之前加上USE语句，则可以使得程序包说明中的定义项在该单元中可见。

假定上述程序包logic的说明部分已经存在，下面的USE语句使得three\_level\_logic和invert对实体说明成为可见。

USE logic.three\_level\_logic，logic.invert；

ENTITY inverter IS

PORT （ x：IN three\_level\_logic；

y：OUT three\_level\_logic）；

END inverter；

ARCHITECTURE inverter\_body OF inverter IS

BEGIN

PROCESS

BEGIN

y <= invert(x)；--一个函数调用

END PROCESS；

END inverter\_body；

结构体部分继承了实体说明部分的可见性，所以不必再使用USE语句。USE语句后跟保留字ALL，表示使用库或程序包中的所有定义。

### 第三节 数据类型、运算操作符

VHDL语言像其它高级语言一样，具有多种数据类型。对大多数数据类型的定义是一致的。

**3.1 VHDL语言的对象及其分类**

在VHDL语言中凡是可以赋予一个值的对象就称为object。object主要包括以下3种：信号、变量、常数（signal、variable、constant）。在电子电路设计中，这3类object通常都具有一定的物理含义。例如，信号对应的代表物理设计中的某一条硬件连接线；常数对应的代表数字电路中的电源和地等；变量对应关系不太直接，通常只代表暂存某些值的载体。3类object的含义和说明场合如表3.1所示。

表3.1 3类object含义和说明场合



**3.1.1 常数**

常数是一个固定的值。所谓常数说明就是对某一常数名赋予一个固定的值。通常赋值在程序开始前进行，该值的数据类型则在说明语句中指明。常数说明的一般格式如下：

CONSTANT 常数名：数据类型：= 表达式；

例如：

CONSTANT VCC：REAL：= 5.0；

CONSTANT DELAY：TIME：=100 ns；

常数一旦被赋值就不能再改变。上面VCC被赋值为 5.0 V，那么在所有的VHDL语言程序中VCC的值就固定为5.0 V，不像后面所提到的信号和变量那样，可以任意代入不同的数值。另外，常数所赋的值应和定义的数据类型一致。

**3.1.2 变量**

变量只能在进程语句、函数语句和过程语句结构中使用，它是一个局部量。在仿真过程中，它不像信号那样，到了规定的仿真时间才进行赋值，变量的赋值是立即生效的。变量说明语句的格式为：

VARIABLE 变量名：数据类型约束条件；

例如：

VARIABLE x，y： INTEGER；

VARIABLE count： INTEGER；

变量在赋值时不能产生附加延时。例如，tmpl，tmp2，tmp3都是变量，那么下式产生延时的方式是不合法的：

tmp3 ：= tmpl ＋ tmp2 AFTER 10 us ；

**3.1.3 信号**

信号是电子电路内部硬件连接的抽象。它除了没有数据流动方向说明以外，其它性质几乎和前面所述的“端口”概念一致。信号通常在构造体、包集合和实体中说明。信号说明语句格式为：

VARIABLE 信号名：数据类型；

例如：

SIGNAL sysclk： STD\_LOGIC；

SIGNAL ground： STD\_LOGIC；

在程序中，信号值的代入采用“<=”代入符，而不是像变量赋值时用“：=”符。而且信号代入时可以附加延时。例如，s1和s2都是信号，且s2的值经10 ns延时以后才被代入s1。此时信号传送语句可书写为：

s1 <= s2 AFTER 10 ns；

信号是一个全局量，它可以用来进行进程之间的通信。

**3.1.4 信号和变量的区别**

信号和变量值的代入不仅形式不同，而且其操作过程也不相同。在变量的赋值语句中，该语句一旦被执行，其值立即被赋予变量。在执行下一条语句时，该变量的值就为上一句新赋的值。变量的赋值符为“：=”。信号代入语句采用“<=”代入符，该语句即使被执行也不会使信号立即发生代入。下一条语句执行时，仍使用原来的信号值。由于信号代入语句是同时进行处理的，因此，实际代入过程和代入语句的处理是分开进行的。

如例1所示，信号c和d的代入值（a+b）和（c+b）将由PROCESS外部通过进程的敏感信号a，b，c取得。进程执行时，只从信号所对应的实体取值，只要不碰到WAIT语句或进程执行结束，进程执行过程中信号值是不进行代入的。

例1： PROCESS （a，b，c）

BEGIN

c <= a + b；

d <= c + b；

END PROCESS；

现在来看一下例2中两个进程描述的语句。首先，由于信号A发生变化使进程语句开始启动执行。这样一来，仿真器对进程中的各语句自上至下地进行处理。当进程所有语句执行完毕，或者中途碰到WAIT语句时，该进程执行结束，信号代入过程被执行。代入同样应按顺序自上至下地进行。

例2： PROCESS（ a，b，c，d）

BEGIN

d <= a；

x <= b + d；

d <= c；

y <= b + d；

END PRO0CESS；

结果

x <= b + c；

y <= b + c；

PROCESS（a，b，c）

VARIABLE d：STD\_LOGIC\_VECTOR（3 DOWNTO 0）；

BEGIN

d := a；

x <= b + d；

d := c；

y <= b + d；

END PROCESS；

结果

x <= b + a；

y <= b + c；

在例2的第一个进程中，d中最初代入的值是a，接着又代入c值。尽管d中先代入a值，后代入c值，在时间上有一个的延时，但是，在代入时由于不进行处理，因此仿真时认为是没有延时。因此d的最终值应为c，这样x和y的内容都为b + c。

在例2的第二个进程中，d是变量。在执行“d ：=a”语句以后，a的值就被赋给d，所以x为b + a。此后又执行“d ：= c”，从而使y为b + c。从这里可以看出，信号量的值将进程语句最后所代入的值作为最终代入值。而变量的值一经赋值就变成新的值。这就是变量赋值和信号代入在操作上的区别。

**3.2 VHDL语言的数据类型**

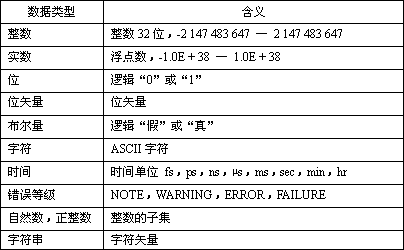
如前所述，在VHDL语言中信号、变量、常数都要指定数据类型。为此，VHDL提供了多种标准的数据类型。另外，为使用户设计方便，还可以由用户自定义数据类型。这样使语言的描述能力及自由度更进一步提高，从而为系统高层次的仿真提供了必要手段。

同时，VHDL语言数据类型的定义相当严格，不同类型之间的数据不能直接代入，而且，即使数据类型相同，而位长不同时也不能直接代入。这样，为了熟练地使用VHDL语言编写程序，必须准确理解各种数据类型的定义。

**3.2.1 标准的数据类型**

标准的数据类型共有10种，如表3.2所示。

表3.2 标准数据类型



1．整数（integer）

整数与数学中的整数的定义相同。在VHDL中，整数的表示范围为－2 147 483 647～

2 147 483 647，即从一（231－l）到（231一1）。千万不要把一个实数（含小数点的数）赋予一个整数变量，这是因为VHDL是一个强类型语言，它要求在赋值语句中的数据类型必须匹配。整数的例子如 ＋136，＋12 456，一457。

尽管整数值在电子系统中可能是用一系列二进制位来表示的，但是整数不能看作是位矢量，也不能按位来进行运算，对整数不能用逻辑操作符。当需要进行位操作时，可以用转换函数，将整数转换成位矢量。

2．实数（real）

实数的定义值范围为一1.0E＋38 ～ ＋1.0E＋38。实数有正负数，书写时一定要有小数点。例如：

一1.0，十 2.5，－1.0E3 8

有些数可以用整数表示也可以用实数表示。例如，数字1的整数表示为1，而用实数表示则为1.0。两个数的值是一样的，但数据类型却不一样。

3．位（bit）

在数字系统中，信号值通常用一个位来表示。位值的表示方法是，用字符‘0’或者‘1’表示之。位与整数中的1和0不同，‘0’和‘1’仅表示一个位的两种取值。另外，位不能用来描述三态信号。

4．位矢量（bit\_vector）

位矢量是用双引号括起来的一组位数据。例如：“001100”，X“00BB”。这里，位矢量最前面的X表示是十六进制。用位矢量数据表示总线状态最形象也最方便。

5．布尔量（boolean）

一个布尔量具有两种状态，“真”或者“假”。虽然布尔量也是二值枚举量，但它和位不同，布尔量没有数值的含义，也不能进行算术运算。它能进行关系运算。例如，它可以在IF语句中被测试，测试结果产生一个布尔量TRUE或者FALSE。

一个布尔量常用来表示信号的状态或者总线上的情况。如果某个信号或者变量被定义为布尔量，那么在仿真中将自动地对其赋值进行核查。一般这一类型的数据的初始值总为 FALSE。

6．字符（character）

字符也是一种数据类型，所定义的字符量通常用单引号括起来，如‘A’。一般情况下VHDL对大小写不敏感，但是对字符量中的大、小写字符VHDL则认为是不一样的。例如，‘A’不同于‘a’。字符量中的字符可以是a～z中的任一个字母，0～9中的任一个数以及空白或者特殊字符，如＄，＠，％等等。包集合STANDARD中给出了预定义的128个ASCII码字符类型，不能打印的用标识符给出。

7．字符串（string）

字符串是由双引号括起来的一个字符序列，它也称字符矢量或字符串数组。例如：

“ Integer range”

字符串常用于程序的提示和说明。

8．时间（time）

时间是一个物理量数据。完整的时间量数据应包含整数和单位两部分，而且整数和单位之间至少应留一个空格的位置。例如，55 sec，2 min等。在包集合STANDARD中给出了时间的预定义，其单位为fs，ps，ns，µs，ms，sec，min，hr。下面是时间数据的例子：

20 fs，100 ps，3 sec

在系统仿真时，时间数据特别有用，用它可以表示信号延时，从而使模型系统能更逼近实际系统的运行环境。

9．错误等级

错误等级类型数据用来表征系统的状态，它共有4种：NOTE（注意），WARNING（警告），ERROR（出错），FAILURE（失败）。在系统仿真过程中可以用这4种状态来提示系统当前的工作情况。这样可以使操作人员随时了解当前系统工作的情况，并根据系统的不同状态采取相应的对策。

**3.2.2 用户定义的数据类型**

用户自定义的数据类型包括枚举类型、整数类型、实数类型、数组类型、记录类型。存取类型、文件类型等。对于自定义的数据类型的一般书写格式是：

TYPE 数据类型名｛，数据类型名｝ 数据类型定义；

1．枚举类型（enumerated type）

定义：把类型中的各个元素列表出来。

说明：方便、直观，提高程序的可读性。

格式：TYPE 数据类型名 IS（元素1，元素2，……）；

例如：TYPE states is（ state0，state1，state2，state3）；

2．整数类型、实数类型

定义：将数规定在某个范围内。

说明：在标准的数据类型中已经有整数和实数的数据类型，用户再自定义一般是为了

电路设计的特殊要求。

格式：TYPE 数据类型名 IS 数据类型约束范围

例如：TYPE data IS INTEGER range 0 to 7；

3．数组类型（array type）

定义：把两个以上的同一类型的数据组织在一起而形成的新的数据类型称为数组。

说明：数组中的元素可以是VHDL语言的任何一种数据类型；可以是一维或多维数组，多维数组要用多个范围来描述。但多维数组不能生成逻辑电路，EDA工具不能用多维数组进行设计综合，只能用于设计仿真和系统建模。

格式：TYPE数据类型名 IS ARRAY （INTEGER 0 TO 9） OF STD\_LOGIC；

例如：

TYPE d IS ARRAY（INTEGER range 0 to 3）of STD\_LOGIC\_VECTOR（7 downto 0）

SIGNAL data：d；

BEGIN

d（0）<= “01010110”；

d（1）<= “10101000”；

d（2）<= “01100011”；

d（3）<= “00010110”；

4．记录类型（record type）

定义：由不同类型的数据对象组织在一起而形成的单一的对象组称为记录。

说明：记录中的每个元素通过字段名访问；记录中的元素可以是相同或不同的类型；适用于仿真，不能用于综合；从记录中提取元素的数据类型应该用“．”。

格式：TYPE 数据类型 IS RECORD

元素名：数据类型名；

元素名：数据类型名；

END RECORD；

例如：TYPE date IS RECORD

year：INTEGER range 2000 to 2009;

month：INTEGER range 1 to 12；

date：INTERGER range 1 to 31；

END RECORD；

SIGNAL d：date；

BEGIN

d.year <= 2004；

d.month <= 2；

d.date <= 15；

END；

5．存取类型（access type）

定义：存储类型也称为寻址类型，用来在对象之间建立联系，或者给新对象分配或释放存储空间，与高级语言中使用的指针类似。在VHDL’ 93语言标准IEEE std\_1076的程序包TEXTIO中定义了一个存取类型。

TYPE line IS ACESS string；

表示类型为line的变量，它的值是指向字符值的指针。只有变量才能定义为存取类型。

6．文件类型（files type）

定义：文件类型是在系统环境中定义为代表文件的数据对象。文件对象的值是系统文件中值的序列。文件类型允许对象有二个文件类型的说明。文件对象由规定的两个过程读出（read）、写入（write）和函数（endfile）检查文件的结尾。

格式：TYPE文件类型名 IS FILE；

**3.3 VHDL的运算操作符**

在VHDL语言中共有4类操作符，可以分别进行逻辑运算（logical）、关系运算（relational）、算术运算（Arithmetic）和并置运算（Concatenation）。需要注意的是，被操作符所操作的对象是操作数，且操作数的类型应该和操作符所要求的类型相一致。另外，运算操作符是有优先级的，例如逻辑运算符NOT，在所有操作符中其优先级最高。表3.3列出了所有操作符的优先次序。

**3.3.1 逻辑运算符**

在VHDL语言中逻辑运算符共有6种，它们分别是：

NOT——取反；

AND——与；

OR——或；

NAND——与非；

NOR——或非；

XOR——异或。

这6种逻辑运算符可以对“STD\_LOGIC”和“STD\_LOGIC\_VECTOR”等数据进行逻辑运算。必须注意，运算符的左边和右边，以及代入的信号的数据类型必须是相同的。

当一个语句中存在两个以上的逻辑表达式时，在C语言中运算有自左至右的优先级顺序的规定，而在VHDL语言中，左右没有优先级差别。例如，在下例中，如去掉式中的括号，那么从语法上来说是错误的：

x <=（a AND b）OR（NOT c AND d）；

当然，如果一个逻辑表达式中只有“AND”，“OR”，“XOR”运算符，那么改变运算顺序将不会导致逻辑的改变。此时，括号是可以省略的。例如：

a <= b AND c AND d AND e；

a <= b OR c OR d OR e；

在所有逻辑运算符中NOT的优先级最高。

表3.3 操作符的优先级



**3.3.2 算术运算符**

VHDL语言有10种算术运算符，它们分别是：

十 ——加；

— ——减；

\* ——乘；

／ ——除；

MOD ——求模；

REM ——取余；

＋ ——正；（一元运算）

— ——负；（一元运算）

\*\* ——指数；

ABS ——取绝对值。，

在算术运算中，对于一元运算的操作数（正、负）可以为任何数值类型（整数、实数、物理量）。加法和减法的操作数也和上面一样，具有相同的数据类型，而且参加加、减运算的操作数的类型也必须相同。乘除法的操作数可以同为整数和实数。物理量可以被整数或实数相乘或相除，其结果仍为一个物理量。物理量除以同一类型的物理量即可得到一个整数量。求模和取余的操作数必须是同一整数类型数据。一个指数的运算符的左操作数可以是任意整数或实数，而右操作数应为一整数。

若对“STD\_LOGIC\_VECTOR”进行“＋”（加）、“一”（减）运算时，两边的操作数和代入的变量位长如不同，则会产生语法错误。另外，“\*”运算符两边的位长相加后的值和要代入的变量的位长不相同时，同样也会出现语法错误。

**3.3.3 关系运算符**

VHDL语言中有6种关系运算符，它们分别是：

＝ ——等于；

／＝ ———不等于；

＜ ——小于；

＜＝ ———小于等于；

＞ ——大于；

＞＝ ——大于等于。

在关系运算符的左右两边是运算操作数，不同的关系运算符对两边的操作数的数据类型有不同的要求。其中等号“＝”和不等号“／＝”可以适用所有类型的数据。其它关系运算符则可使用于整数（INTEGER）和实数（REAL）、位（STD\_LOGIC）等校举类型以及位矢量（STD\_LOGIC\_VECTOR）等数组类型的关系运算。在进行关系运算时，左右两边的操作数的数据类型必须相同，但是位长度不一定相同。当然也有例外的情况，在利用关系运算符对位矢量数据进行比较时，比较过程是从最左边的位开始。自左至右按位进行比较的。在位长不同的情况下，只能按自左至右的比较结果作为关系运算的结果。例如，对3位和4位的位矢量进行比较：

SIGNAL a：STD\_LOGIC\_VECTOR（3 DOWNTO 0）；

SIGNAL b：STD\_LOGIC\_VECTOR（2 DOWNTO 0）；

a <= “1010”； －－10

b <= “111”； －－7

IF（ a ＞ b） THEN

┇

ELSE

┇

上例a的值为10，而b的值为7，a应该比b大。但是，由于位矢量是从左至右按位比较的，当比较到次高位时，a的次高位为“0”而b的次高位为“1”，故比较结果b比a大。这样的比较结果显然是不符合实际情况的。

为了能使位矢量进行关系运算，在包集合“STD\_LOGIC\_UNSIGNED”中对“STD\_LOGIC\_VECTOR”关系运算重新作了定义，使其可以正确的进行关系运算。注意在用时必须首先说明调用该包集合。

**3.3.4 并置运算符**

并置运算符“＆”用于位的连接。例如，将4个位用并置运算符”＆”连接起来就可以构成一个具有4位长度的位矢量。两个4位的位矢量用并置运算符“＆”连接起来就可以构成8位长度的位矢量。

如en是b（0）～b（3）的允许输出信号，

tmp\_b <= b AND （en & en & en & en）

这个语句表示b的4位位矢量由en进行选择得到一个4位位矢量的输出。

### 第四节 VHDL语言的基本描述语句

在用VHDL语言描述系统硬件行为时，按语句执行顺序对其进行分类，可以分为顺序（Sequential）描述语句和并发（Concurrent）描述语句。例如，进程语句（Process Statement）是并发语句。在一个构造体内可以有几个进程语句同时存在，各进程语句是并发执行的。但是，在进程内部所有语句应是顺序描述语句，也就是说按书写的顺序自上至下，一个语句一个语句地执行的。例如，IF语句、LOOP语句等都属于此类顺序描述语句。灵活运用这两类语句就可以正确地描述系统的并发行为和顺序行为。

**4.1 并发描述语句**

在VHDL语言中能进行并发处理的语句有：进程（ PROCESS）语句，并发信号代入（Concurrent Signal Assignment）语句，条件信号代入（Conditional Signal Assignment）语句，选择信号代入（Selective Signal Assignment）语句，并发过程调用（Concurrent Procedure Call）语句和块（BLOCK）语句。由于硬件描述语言所描述的实际系统，其许多操作是并发的，所以在对系统进行仿真时，这些系统中的元件在定义的仿真时刻应该是并发工作的。并发语句就是用来表示这种并发行为的。并发描述可以是结构性的也可以是行为性的。在并发语句中最关键的语句是进程。

**4.1.1 进程语句**

PROCESS语句是并发处理语句，在一个构造体中多个PROCESS语句可以同时并发运行。因此，PROCESS语句是VHDL语言中描述硬件系统并发行为的最基本的语句。PROCESS语句归纳起来有如下几个特点：

1．可以与其它进程并发运行，并可存取构造体或实体中所定义的信号；

2．进程结构中的所有语句都是顺序执行的；

3．要启动进程，在进程结构中必须包含一个显式的敏感信号量表；

4．进程之间的通信是通过信号量传递来实现的。

并发信号代入语句、条件信号代入语句等并发执行语句，实质上是一种进程的缩写形式，它们仍可以归属于进程语句。

**4.1.2 并发信号代入语句**

代入语句（信号代入语句）可以在进程内部使用，此时它作为顺序语句形式出现；代入语句（并发信号代入语句）也可以在构造体的进程之外使用，此时它作为并发语句形式出现。一个并发信号代入语句实际上是一个进程的编写。例如：

ARCHITECTURE behav OF a\_var IS

BEGIN

Output <= a（i）；

END behav；

可以等效于

ARCHITECTURE behave OF a\_ var IS

BEGIN

PROCESS（a，i）

BEGIN

Output <= a（i）；

END PROCESS；

END behav；

由信号代入语句的功能可以知道，当代入符号“<=”右边的信号值发生任何变化时，代入操作就会立即发生，新的值将赋予代入符号“<=”左边的信号。从进程语句描述来看，在PROCESS语句的括号中列出了敏感信号量表，例中是a和i。由PROCESS语句的功能可 知，在仿真时进程一直在监视敏感信号量表中的敏感信号量a和i。一旦任何一个敏感信号量发生新的变化，进程将得到启动，代入语句将被执行，新的值将从Output信号量输出。

由上面叙述可知，并发信号代入语句和进程语句在这种情况下确实是等效的。

并发信号代入语句在仿真时刻同时运行，它表征了各个独立器件的各自的独立操作。例如：

a <= b + c；

d <= e \* f；

第一个语句描述了一个加法器的行为，而第二个语句描述了一个乘法器的行为。在实际硬件系统中，加法器和乘法器是独立并行工作的。现在第一个语句和第二个语句都是并发信号代入语句，在仿真时刻，两个语句是并发处理的，从而真实地模拟了实际硬件系统中的加法器和乘法器的工作。

并发信号代入语句可以仿真加法器、乘法器、除法器、比较器及各种逻辑电路的输出。因此，在代入符号“<=”的右边可以用算术运算表达式，也可以用逻辑运算表达式，还可以用关系操作表达式来表示。

**4.1.3 条件信号代入语句**

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，其书写格式为：

目的信号量 < = 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

┇ ELSE

表达式n；

在每个表达式后面都跟有用“WHEN”所指定的条件，如果满足该条件，则该表达式的值代入目的信号量；如果不满足条件，则再判别下一个表达式所指定的条件。最后一个表达式可以不跟条件表达式。它表明，在上述表达式所指明的条件都不满足时，则将该表达式的值代入目标信号量。

例1就是利用条件信号代入语句来描述的四选一逻辑电路。

例1：

ENTITY mux4 IS

PORT（i0，i1，i2，i3，a，b：IN STD\_LOGIC；

q：OUT STD\_LOGIC）；

END mux4；

ARCHITECTURE rtl OF mux4 IS

SIGNAL sel：STD\_LOGIC\_VECTOR（1 DOWNTO 0）；

BEGIN

sel <= b ＆ a；

q <= i0 WHEN sel==“00”ELSE

i1 WHEN sel==“01”ELSE

i2 WHEN sel==“10”ELSE

i3 WHEN sel==“01”ELSE

‘X’；

END rtl；

条件信号代入语句与前述的IF语句的不同之处就在于，后者只能在进程内部使用（因为它们是顺序语句），而且与IF语句相比，条件信号代入语句中的ELSE是一定要有的，而IF语句则可以有也可以没有。另外，与IF语句不同的是，条件信号代入语句不能进行嵌套，因此，受制于没有自身值代入的描述，不能生成锁存电路。用条件信号代入语句所描述的电路，与逻辑电路的工作情况比较贴近，这样，往往要求设计者具有较多的硬件电路知识，从而使一般设计者难于掌握。一般来说，只有当用进程语句、IF语句和CASE语句难于描述时，才使用条件信号代入语句。

**4.1.4 选择信号代入语句**

选择信号代入语句类似于CASE语句，它对表达式进行测试，当表达式取值不同时，将使不同的值代入目的信号量。选择信号代入语句的书写格式如下：

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n WHEN 条件n；

下面仍以四选一电路为例说明一下该语句的使用方法，具体如例2所示。

例2：

LIBRARY IEEE；

USE IEEE.STD\_ LOGIC\_1164.ALL；

ENTITY mux IS

PORT（i0，i1，i2，i3，a，b：IN STD\_LOGIC；

q：OUT STD\_LOGIC）；

END mux；

ARCHITECTURE behav OF mux IS

SIGNAL sel：INTEGER；

BEGIN

WITH sel SELECT

q <= i0 WHEN 0，

i1 WHEN 1，

i2 WHEN 2，

i3 WHEN 3，

‘X’WHEN OTHERS；

sel <= 0 WHEN a=‘0’AND b=‘0’ELSE

1 WHEN a=‘1’AND b=‘0’ELSE

2 WHEN a=‘0’AND b=‘1’ELSE

3 WHEN a=‘1’AND b=‘1’ELSE

4；

END behav；

上例中的选择信号代入语句，根据sel的当前不同值来完成i0，i1，i2，i3及剩余情况的选择功能。选择信号代入语句在进程外使用。当被选择的信号发生变化时，该语句就会启动执行。由此可见，选择信号的并发代入，可以在进程外实现CASE语句进程的功能。例如，四选一电路用CASE语句进程所描述的程序如例3所示。

例3：

LIBRARY IEEE；

USE IEEE.STD\_LOGIC\_1164. ALL；

ENTITY mux4 IS

PORT（input： IN STD\_LOGIC\_VECTOR（1 DOWNTO 0）；

i0，i1，i2，i3：IN STD＿LOGIC；

q：OUT STD\_LOGIC）；．

END inllX4 alal；

ARCHITECTURE rtl OF mux4 IS

BEGIN

PROCESS（input）

BEGIN

CASE input IS

WHEN “00” => q <= i0；

WHEN “01” => q <= i1；

WHEN “10” => q <= i2；

WHEN “11” => q <= i3；

WHEN OTHERS => q <=‘X’；

END CASE；

END PROCESS；

END rtl；

对照例2和例3可以看到，两者功能是完全一样的，所不同的仅仅是描述方法有区别而已。

**4.1.5 并发过程调用语句**

并发过程调用语句可以出现在构造体中，而且是一种可以在进程之外执行的过程调用语句。有关过程的结构及书写方法不再详述，这里仅就调用时应注意的几个问题作一说明。

1．并发过程调用语句是一个完整的语句，在它的前面可以加标号；

2．并发过程调用语句应带有IN，OUT或者INOUT的参数，它们应列于过程名后跟的括号内；

3．并发过程调用可以有多个返回值，但这些返回值必须通过过程中所定义的输出参数带回。

在构造体中采用并发过程调用语句的实例如下所示：

ARCHITECTURE …

BEGIN

vector\_to\_int（z，x.flag，q）；

┇

END；

例中的vector\_to\_int并发过程调用是对位矢量z进行数制转换，使之变成十进制的整数。x.flag是标志位，当标志位为“真”表明转换失败，为“假”表明转换成功。

这种并发过程调用语句实际上是一个过程调用进程的简写。过程调用语句可以出现在进程语句中，如果该进程的作用就是进行过程调用，完成该过程的操作功能，那么两者是完全等效的。由此可知，上例的并发过程调用语句和下面的过程调用进程是完全等效的，因为两者都是为了完成位矢量至整数的转换。

ARCHITECTURE …

BEGIN

PROCESS（z，q）

BEGIN

vector\_to\_int（z，x.flag，q）；

┇

END PROCESS：

END；

在构造体中的并发过程调用语句也由过程信号敏感量的变化而得到启动，例如上例的位矢量z的变化将使vector\_to\_int语句得到启动，并执行之。执行结果将拷贝到x.flag和q中，构造体中的其它语句就可以使用该结果。

**4.2 顺序描述语句**

顺序描述语句只能出现在进程或子程序中，由它定义进程或子程序所执行的算法。语句中所涉及到的系统行为有时序流、控制、条件和迭代等；语句的功能操作有算术、逻辑运算、信号和变量的赋值、子程序调用等。顺序描述语句像在一般高级语言中一样，其语句是按出现的次序加以执行的。在VHDL语言中顺序描述语句有以下几种：

★ WAIT语句

★ ASSERT语句

★ 信号代入语句

★ 变量赋值语句

★ IF语句

★ CASE语句

★ LOOP语句

★ NEXT语句；

★ EXIT语句；

**4.2.1 WAIT 语句**

WAIT语句通常在进程中使用。敏感信号表和WAIT都是进程的启动、触发条件。为避免进程产生误触发，在进程语句中，敏感信号表和WAIT不能共存于一个进程中。

WAIT语句的书写格式有以下几种：

1．WAIT

WAIT后不带任何信息，表示无限等待。

2．WAIT ON信号表

暂停进程，直到敏感信号表中的某个信号值发生变化。

3．WAIT UNTIL条件表达式

暂停进程，直到条件表达式成立时进程启动。

4．WAIT FOR时间表达式

进程暂停由时间表达式所规定的时间，时间到时进程启动。

下面举两个例子来说明这两种方法。

例4：

ENTITY reg\_logic IS

PORT（d：IN STD\_LOGIC；

clk：IN STD\_LOGIC；

q：OUT STD\_LOGIC）；

ARCHITECTURE example OF reg\_logic IS

BEGIN

PROCESS（clk）

BEGIN

IF （clk’EVENT AND clk=’1’） THEN q <= d；

END IF；

END PROCESS；

END example；

例5：

ENTITY xor\_gate IS

PORT （in1：IN STD\_LOGIC；

in2：IN STD\_LOGIC；

q：OUT STD\_LOGIC）；

END xor\_gate；

ARCHITECTURE example OF xor\_gate IS

BEGIN

PROCESS

BEGIN

q <= in1 XOR in2 AFTER 10 ns；

END PROCESS；

END example；

**4.2.2 ASSERT语句**

ASSERT语句主要用于程序仿真、调试中的人机会话，它可以给出一个文字串作为警告和错误信息。ASSERT语句的书写格式为：

ASSERT条件［REPORT输出信息］［SEVERITY级别］；

当执行ASSERT语句时，就会对条件进行判别。如果条件为“真”，则向下执行另一个语句；如果条件为“假”，则输出错误信息和错误严重程度的级别。在REPORT后面跟的是设计者所写的文字串，通常是说明错误的原因。文字串应用双引号“”将它们括起来。 SEVERITY后面跟的是错误严重程度的级别。在VHDL语言中错误严重程度分为4个级别：FAILURE，ERROR，WARNING，NOTE。

例如ASSERT语句：

ASSERT（sendB＝’ l’）

REPORT ’’sendB timed out at’ l’”;

SEVERITY ERROR；

该断言语句的条件是信号量sendB＝‘1’。如果执行到该语句时，信号量sendB＝‘0’，说明条件不满足，就会输出REPORT后跟的文字串。该文字串说明，出现了超时等待错误。SEVERITY后跟的错误级别告诉操作人员，其出错级别为ERROR。ASSERT语句为程序的仿真和调试带来了极大的方便。

**4.2.3 信号代入语句**

信号代入语句前面已有详述，这里只做简单介绍。

信号代入语句的书写格式为：

目的信号量 <= 信号量表达式；

该语句表明，将右边信号量表达式的值赋予左边的目的信号量。例如：a <= b；该语句表示将信号量b的当前值赋予目的信号量a。需要再次指出的是，代入语句的符号“<=”和关系操作的小于等于符号“<=”非常相似，要正确判别不同的操作关系，应注意上下文的含义和说明。另外，代入符号两边信号量的类型和位长度应该是一致的。

**4.2.4 变量赋值语句**

变量赋值语句的书写格式为：

目的变量 := 表达式；

该语句表明，目的变量的值将由表达式所表达的新值替代。但是两者的类型必须相同。目的变量的类型、范围事先应给出。右边的表达式可以是变量、信号或字符。该变量和一般高级语言中的变量是类似的。例如：

a := 2；

b := d + e；

变量值只在进程或子程序中使用，它无法传递到进程之外。因此，它类似于一般高级语言的局部变量，只在局部范围内有效。

**4.2.5 IF语句**

IF语句是根据所指定的条件来确定执行哪些语句的，其书写格式通常可以分成3种类型。

1．IF语句的门闩控制

用作门闩控制的IF语句的书写格式为：

IF条件 THEN

顺序处理语句

END IF；

当程序执行到该IF语句时，就要判断IF语句所指定的条件是否成立。如果条件成立，则IF语句所包含的顺序处理语句将被执行；如果条件不成立，程序将跳过IF语句所包含的顺序处理语句，而向下执行IF语句后继的语句。这里的条件起到门闩的控制作用，如例6所示。

例6：

IF（a=’1’）THEN

c <= b；

END IF；

该IF语句所描述的是一个门闩电路。a是门闩控制信号量；b是输入信号量；c是输出信号量。当门闩控制信号量a为‘1’时，输入信号量b的任何值的变化都将被赋予输出信量c。也就是说，c值与b是永远相等的。当 a≠‘ 1’时c <= b语句不被执行，c将维持原始值，而不管信号量b的值发生什么变化。

这种描述经逻辑综合，实际上可以生成D触发器。

2．IF语句的二选择控制

当IF语句用作二选择控制时的书写格式为：

IF条件 THEN

顺序处理语句；

ELSE

顺序处理语句；

END IF；

在这种格式的IF语句中，当IF语句所指定的条件满足时，将执行THEN和ELSE之间的顺序处理语句。当IF语句所指定的条件不满足时，将执行ELSE和END IF之间的顺序处理语句。也就是说，用条件来选择两条不同程序的执行路径。

这种描述的典型逻辑电路实例是二选一电路。例如，二选一电路的输入为a和b，选择控制端为sel，输出端为c。那么用IF语句来描述该电路行为的程序如例7所示。

例7：

ARCHITECTURE rtl OF mux2 IS

BEGIN

PROCESS（a，b，sel）

BEGIN

IF（sel=’1’） THEN

c <= a；

ELSE

c <= b；

END IF；

END PROCESS；

END rtl；

3．IF语句的多选择控制

IF语句的多选择控制又称IF语句的嵌套，在这种情况下，它的书写格式为：

IF 条件 THEN

顺序处理语句；

ELSIF 条件 THEN

顺序处理语句；

ELSIF 条件 THEN

顺序处理语句；

ELSE

顺序处理语句；

END IF；

在这种多选择控制的IF语句中，设置了多个条件，当满足所设置的多个条件之一时，就执行该条件后跟的顺序处理语句。如果所有设置的条件都不满足，则执行ELSE和END IF之间的顺序处理语句。

这种描述的典型逻辑电路实例是多选一电路。例如，四选一电路的描述如例8所示。

例8：

LIBRARY IEEE；

USE IEEE.STD\_LOGIC\_1164.ALL；

ENTITY mux4 IS

PORT（input：IN STD\_LOGIC\_VECTOR（3 DOWNTO 0）；

sel：IN STD\_LOGIC\_VECTOR（1 DOWNTO 0）；

y：OUT STD\_LOGIC）；

END mux4；

ARCHITECTURE rtl OF mux4 IS

BEGIN

PROCESS（input，sel）

BEGIN

IF（sel=”00”）THEN

y <= input（0）；

ELSIF（sel=”01”）THEN

y <= input（1）；

ELSIF（sel=”10”）THEN

y <= input（2）；

ELSE

Y <= input（3）；

END IF；

END PROCESS；

END rtl；

IF语句不仅可以用于选择器的设计，而且还可以用于比较器、译码器等凡是可以进行条件控制的逻辑电路设计。

**4.2.6 CASE语句**

CASE语句用来描述总线或编码、译码的行为，从许多不同语句的序列中选择其中之一执行之。虽然IF语句也有类似的功能，但是CASE语句的可读性比IF语句要强得多，程序的读者很容易找出条件式和动作的对应关系。CASE语句的书写格式如下所示：

CASE表达式 IS

WHEN条件表达式 => 顺序处理语句；

END CASE；

上述CASE语句中的条件表达式可以有如下4种不同的表示形式：

WHEN值 => 顺序处理语句；

WHEN值∣值∣值∣…∣值 => 顺序处理语句；

WHEN值TO值 => 顺序处理语句；

WHEN OTHERS => 顺序处理语句；

当CASE和IS之间的表达式的取值满足指定的条件表达式的值时，程序将执行后跟的

由符号 => 所指的顺序处理语句。条件表达式的值可以是一个值；或者是多个值的“或”关系；或者是一个取值范围；或者表示其它所有的缺省值。

当条件表达式取值为某一值时的CASE语句的使用实例如例9所示。

例9：

LIBRARY IEEE；

USE IEEE.STD\_LOGIC\_1164.ALL；

ENTITY mux4 IS

PORT（a，b，i0，i1，i2，i3：IN STD\_LOGIC；

q：OUT STD\_LOGIC）；

END mux4；

ARCHITECTURE mux4\_behave OF mux4 IS

SIGNAL sel：NTEGER RANGE 0 TO 3；

BEGIN

PROCESS（a，b，ic，if，iZ，i3）

BEGIN

sel <= ‘0’；

IF（a＝’1’）THEN

sel <= sel + 1；

END IF；

IF（b＝’1’）THEN

sel <= sel + 2；

END IF；

CASE sel IS

WHEN 0 => q <= i0；

WHEN 1 => q <= i1；

WHEN 2 => q <= i2；

WHEN 3 => q <= i3；

END CASE；

END PROCESS；

END mux4\_behave；

例9表明，选择器的行为描述不仅可以用IF语句，而且也可以用CASE语句。两者的区别在于：首先在IF语句中，先处理最起始的条件；如果不满足，再处理下一个条件；而在CASE语句中，没有值的顺序号，所有值是并行处理的。因此，在WHEN项中已用过的值，如果在后面WHEN项中再次使用，那在语法上是错误的。也就是说，值不能重复使用。另外，应该将表达式的所有取值都—一列举出来，如果不列举出表达式的所有取值，在语法上也是错误的。

通常在CASE语句中，WHEN语句可以颠倒次序而不至于发生错误，而在IF语句中，颠倒条件判别的次序往往会使综合的逻辑功能发生变化。

**4.2.7 LOOP语句**

LOOP语句使程序能进行有规则的循环，循环次数受迭代算法控制。常用来描述迭代电路的行为，通常使用在循环语句中。一个LOOP语句中所包含的一组顺序语句要重复执行若干次。LOOP语句有3种重复方式：FOR模式、WHILE模式和无限循环模式。

1．FOR模式

格式：［标号］：FOR 循环变量 IN 范围 LOOP

顺序处理语句

END LOOP［标号］；

上述格式中，循环变量的值在每次循环中都会发生变化；循环变量在信号说明、变量说明中不能使用，信号、变量不能代入到循环变量中；范围表示循环变量在循环过程中的取值范围；只要循环变量还在范围内，循环将一直继续下去；局部变量只能在进程中出现，值不能被带出进程，只有全局变量、信号才可以将局部变量的值带出进程。

例10：

ARCHITECTURE rtl OF ex IS

BEGIN

PROCESS（a，b，c）

BEGIN

FOR i IN 0 TO 4 LOOP

IF a（i）=’1’ THEN

q（i）<= b（i）；

ELSE

q（i）<= c（i）；

END IF；

END LOOP；

END PROCESS；

END rtl；

2．WHILE模式

格式：［标号］：WHILE 条件 LOOP

顺序处理语句

END LOOP［标号］；

上述格式中，当条件为“真”（true）时，执行紧跟着的顺序语句；反之，如果条件为“假”（false）就结束循环。

例10：

PROCESS（a，b，reset）

VARIABLE i：INTEGER RANGE 0 TO 31；

BEGIN

IF reset=’0’ THEN

i:=0；

q <=（OTHERS => ‘0’）；

ELSE

WHILE q < 12 LOOP

IF i=31 THEN EXIT；

ELSE

i <= i + 1；

q <= a（i）+b（i）；

END IF；

END LOOP；

END IF；

END PROCESS；

**4.2.8 NEXT语句**

在LOOP语句中NEXT语句用来跳出本次循环，其书写格式为：

NEXT［标号］［WHEN条件］；

NEXT语句执行时将停止本次迭代，而转入下一次新的迭代。NEXT后跟的“标号”表明下一次迭代的起始位置，而“WHEN条件”则表明 NEXT语句执行的条件。如果NEXT语句后面既无“标号”也无“WHEN条件”说明，那么只要执行到该语句就立即无条件地跳出本次循环，从 LOOP语句的起始位置进入下一次循环，即进行下一次迭代，如例11所示。

例11：

PROCESS（a，b）

CONSTANT max：INTEGER := 255；

BEGIN

FOR i IN 0 TO max LOOP

IF（done（i）＝TRUE） THEN

NEXT；

ELSE

done（i）:= TRUE；

END IF；

q（i） <= a（i）AND b（i）；

END LOOP；

END PROCESS；

当LOOP语句嵌套时，通常NEXT语句应标有“标号”和“WHEN条件”，也就是说NEXT语句实际上是用于LOOP语句的内部循环控制。

**4.2.9 EXIT语句**

EXIT语句也是LOOP语句中使用的循环控制语句，与NEXT语句不同的是，执行EXIT语句将结束循环状态，从LOOP语句中跳出，结束LOOP语句的正常执行。EXIT语句的书写格式为：

EXIT［标号］［WHEN条件］；

如果EXIT后面没有跟“标号”和“ WHEN条件”，则程序执行到该语句时就无条件地从LOOP语句中跳出，结束循环状态，继续执行LOOP语句后继的语句，如例12所示。

例12：

PROCESS（a）

VARIABLE int\_a：INTEGER；

BEGIN

Int\_a := a；

FOR i IN 0 TO max LOOP

IF（int\_a < 0）THEN

EXIT；

ELSE

Int\_a := int\_a - 1；

END IF；

END LOOP；

y <= q；

END PROCESS；

在该例中int\_a通常代入大于0的正数值。如果int\_a的取值为负值或零将出现错误状态，算式就不能计算，也就是说int\_a小于或等于0时，IF语句将返回“真”值，EXIT语句得到执行，LOOP语句执行结束。程序将向下执行LOOP语句后继的语句。

**4.3 命名规则和注解**

为使程序结构清晰，具有更好的可读性，提出如下建议：

1）对VHDL语言的保留字，习惯上用大写，其他应小写。但有一种情况需要注意，代表不定状态的“X”和高阻态的“Z”要求必须大写。

2）单词、信号名的含义要明确，命名时不要与保留字相同，以免造成混乱。

3）要求段落分明、含义确切，嵌套关系一目了然。

4）应辅以适量的程序注释。

1．命名规则

所使用的名字，如信号名、实体名、结构体名、变量名、各种进程标记、块标记等进行命名时，应遵守如下规则：

1）名字的最前面应使用英文字母；

2）构成名字的字符只能用英文字母、数字和连字符“－”；

3）不能连续使用“－”，名字的最后一个字符也不能用连字符“－”。

2．注释

1）VHDL语言中使用的注释符是“--”，从注释符号“--”开始到该行末尾结束；

2）所注释的文字不作为语句来处理，不产生硬件电路结构，不描述电路硬件行为；

3）在编译、仿真过程中，对于中文的注释有的软件可能会产生错误。

**4.4 VHDL设计举例**

这一节，举一个8位寄存器的简单VHDL程序（详见参考文献8）。

LIBRARY IEEE；

USE IEEE.STD\_LOGIC\_1164.ALL；

USE IEEE.STD\_LOGIC\_UNSIGNED.ALL；

ENTITY shifter8 IS

PORT（in1，in2：IN STD\_LOGIC； --input port

clrn：IN STD\_LOGIC； --clear signal

clk：IN STD\_LOGIC； --clokck signal

q0，q1，q2，q3，q4，q5，q6，q7：OUT STD\_LOGIC）；

END shifter8； --output port

ARCHITECTURE rtl Of shifter8 IS

SIGNAL shift\_reg：STD\_LOGIC\_VECTOR（7 DOWNTO 0）；

BEGIN

PROCESS（clrn，clk）

BEGIN

IF clrn=’0’ THEN

shift\_reg <=（ OTHERS => ‘0’）；

ELSIF clk’s EVENT AND clk=’1’ THEN

shift\_reg（7 DOWNTO 1）<= shift\_reg（6 DOWNTO 0）；

shift\_reg（0）<= in1 AND in2；

END IF；

END PROCESS；

q0 <= shift\_reg（0）；

q1 <= shift\_reg（1）；

q2 <= shift\_reg（2）；

q3 <= shift\_reg（3）；

q4 <= shift\_reg（4）；

q5 <= shift\_reg（5）；

q6 <= shift\_reg（6）；

q7 <= shift\_reg（7）；

END rtl；