VHDL
硬件描述语言与数字逻辑电路设计

修订版

—电子工程师必备知识

侯伯亨　顾新　编著

西安电子科技大学出版社

封面设计／电脑制作：傅化群

VHDL 硬件描述语言与数字逻辑电路设计

—— 电子工程师必备知识

ISBN 7-5606-0534-6

9 787560 605340 >

ISBN 7-5606-0534-6/TP・0264

定 价:20.80元

VHDL 硬件描述语言与数字逻辑电路设计

——电子工程师必备知识

（修订版）

侯伯亨　顾新　编著

西安电子科技大学出版社

1999

内 容 简 介

本书系统地介绍了一种硬件描述语言，即 VHDL 语言设计数字逻辑电路和数字系统的新方法。这是电子电路设计方法上一次革命性的变化，也是迈向 21 世纪的电子工程师所必须掌握的专门知识。本书共分 12 章，第 1 章～第 8 章主要介绍 VHDL 语言的基本知识和使用 VHDL 语言设计简单逻辑电路的基本方法；第 9 章和第 10 章分别以定时器和接口电路设计为例，详述了用 VHDL 语言设计复杂电路的步骤和过程；第 11 章简单介绍了 VHDL 语言 93 版和 87 版的主要区别；第 12 章介绍了 MAX＋plus Ⅱ 的使用说明。

本书以数字逻辑电路设计为主线，用对比手法来说明数字逻辑电路的电原理图和 VHDL 语言程序之间的对应关系，并列举了众多的实例。另外，还对设计中的有关技术，如仿真、综合等作了相应说明。本书简明扼要，易读易懂。它可作为大学本科和研究生的教科书，也可以作为一般从事电子电路设计工程师的自学参考书。

VHDL 硬件描述语言与数字逻辑电路设计
——电子工程师必备知识
（修订版）

侯伯亨　顾新　编著

责任编辑　徐德源
出版发行　西安电子科技大学出版社
（西安市太白南路 2 号）
邮　编　710071
电　话　(029)8227828
经　销　新华书店
印　刷　陕西省富平印刷有限责任公司
版　次　1997 年 9 月第 1 版
1999 年 1 月第 2 版
1999 年 1 月第 2 次印刷
开　本　787 毫米×1092 毫米　1/16　印张 20.75
字　数　485 千字
印　数　4 001～10 000 册
定　价　20.80 元
ISBN 7－5606－0534－6/TP・0264

修订版说明

本书出版后承蒙众多读者厚爱，对本书的编排及有关内容提出了许多宝贵建议。另外，经近一年的教学实践作者也深深感到，为了进一步适应国内 EDA 技术的发展需要与更便于教学和读者自学，确实需要对本书的部分内容进行删节和增订。

本书修订版对第 9 章进行了修改，使其便于用 MAX+plus Ⅱ 工具来进行验证；原第 10 章的内容改写为微处理器接口芯片设计实例；增加了第 11 章 93 版和 87 版 VHDL 语言的主要区别及第 12 章 MAX+plus Ⅱ 使用说明。在此基础上还对原书已发现的错误进行了订正。书中的主要实例都已用 MAX+plus Ⅱ 工具进行了验证。

需要说明的是，在本书的例程中多处采用了位矢量算术运算。这一点在 SYNOPSYS 公司的 VHDL · compiler 是行得通的，因为它对算术运算包进行了扩充。但是，在使用 MAX+plus Ⅱ 编译工具时却不行。此时只能先进行整数运算，然后再将整数结果转换成位矢量输出，正如本书中第 9 章中所示例的那样。

尽管作者对全书进行了认真修订，但是，由于作者水平有限，书中可能仍有错误和不当之处，敬请广大读者多提宝贵意见。

作者

1998 年 8 月 18 日于西安

前　言

随着电子技术的发展，当前数字系统的设计正朝着速度快、容量大、体积小、重量轻的方向发展。推动该潮流迅猛发展的引擎就是日趋进步和完善的 ASIC 设计技术。目前数字系统的设计可以直接面向用户需求，根据系统的行为和功能要求，自上至下地逐层完成相应的描述、综合、优化、仿真与验证，直到生成器件。上述设计过程除了系统行为和功能描述以外，其余所有的设计过程几乎都可以用计算机来自动地完成，也就是说做到了电子设计自动化(EDA)。这样做可以大大地缩短系统的设计周期，以适应当今品种多、批量小的电子市场的需求，提高产品的竞争能力。

电子设计自动化(EDA)的关键技术之一是要求用形式化方法来描述数字系统的硬件电路，即要用所谓硬件描述语言来描述硬件电路。所以硬件描述语言及相关的仿真、综合等技术的研究是当今电子设计自动化领域的一个重要课题。

硬件描述语言的发展至今已有几十年的历史，并已成功地应用到系统的仿真、验证和设计综合等方面。到本世纪 80 年代后期，已出现了上百种的硬件描述语言，它们对设计自动化起到了促进和推动作用。但是，它们大多各自针对特定设计领域，没有统一的标准，从而使一般用户难以使用。广大用户所期盼的是一种面向设计的多层次、多领域且得到一致认同的标准的硬件描述语言。80 年代后期由美国国防部开发的 VHDL 语言(VHSIC Hardware Description Language)恰好满足了上述这样的要求，并在 1987 年 12 月由 IEEE 标准化(定为 IEEE std 1076—1987 标准，1993 年进一步修订，被定为 ANSI/IEEE std 1076—1993 标准)。它的出现为电子设计自动化(EDA)的普及和推广奠定了坚实的基础。据 1991 年有关统计资料表明，VHDL 语言业已被广大设计者所接受，据称已有 90%的设计者使用或即将使用 VHDL 语言来设计数字系统。另外，众多的 CAD 厂商也纷纷使自己新开发的电子设计软件与 VHDL 语言兼容。由此可见，使用 VHDL 语言来设计数字系统是电子设计技术的大势所趋。

作者编写此书的目的就在于向广大电子设计人员介绍 VHDL 语言的基本知识和使用它来设计数字系统硬件电路的方法，从而使读者摆脱传统的人工设计方法的框框，使数字系统设计的水平上升到一个新的阶段。

本书共分 10 章，第 1 章～第 8 章主要介绍 VHDL 语言的基本知识和使用 VHDL 语言设计简单逻辑电路的基本方法。第 9 章和第 10 章分别以定时器和虚拟处理器电路设计为例，详述了如何用 VHDL 语言设计复杂逻辑电路的步骤和过程。全书采用对照说明的方法来叙述逻辑电原理图和 VHDL 语言描述的对应关系，并且安排了大量的程序实例。读者只要通读全书，就可以基本掌握用 VHDL 语言设计一般逻辑电路的方法。由于受条件和环境限制，本书对仿真和综合只作了概略性的介绍，因为这些工具的具体使用方法随各厂商不同而有较大差别。另外，有些 VHDL 语言的程序实例取材于不同版本的资料，本书在编写时虽作了一些统一，但某些书写格式上还会存在个别的差异，敬请读者谅解。

本书在编写过程中引用了诸多学者和专家的著作和论文中的研究成果，在这里向他们表示衷心的感谢。同时，也向一贯热情支持和关心作者的西安电子科技大学出版社的领导和编辑及工作人员表示深深的谢意。

由于作者水平有限，错误和不当之处在所难免，敬请各位读者不吝赐教。

编者

1997 年 2 月 21 日元霄节于西安

目 录

第 1 章 数字系统硬件设计概述

第 2 章 VHDL 语言程序的基本结构

第 3 章 VHDL 语言的数据类型及运算操作符

第 4 章 VHDL 语言构造体的描述方式

第 5 章 VHDL 语言的主要描述语句

第 6 章 数值系统的状态模型

第 7 章 基本逻辑电路设计

第 8 章 仿真与逻辑综合

第 9 章 计时电路设计实例

第 10 章 微处理器接口芯片设计实例

第 11 章 93 版和 87 版 VHDL 语言的主要区别

第 12 章 MAX+plus Ⅱ使用说明

第 1 章

数字系统硬件设计概述

自计算机诞生以来，数字系统设计历来存在两个分枝，即系统硬件设计和系统软件设计。同样，设计人员也因工作性质不同，被分成两群：硬件设计人员和软件设计人员。他们各自从事各自的工作，很少涉足对方的领域。特别是软件设计人员更是如此。但是，随着计算机技术的发展和硬件描述语言 HDL(Hardware Description Language)的出现，这种界线已经被打破。数字系统的硬件构成及其行为完全可以用 HDL 语言来描述和仿真。这样，软件设计人员也同样可以借助 HDL 语言，设计出符合要求的硬件系统。不仅如此，利用 HDL 语言来设计系统硬件与利用传统方法设计系统硬件相比，还带来了许多突出的优点。它是硬件设计领域的一次变革，对系统的硬件设计将产生巨大的影响。在本章将详细介绍这种硬件设计方法的变化。

1.1 传统的系统硬件设计方法

在计算机辅助电子系统设计出现以前，人们一直采用传统的硬件电路设计方法来设计系统的硬件。这种硬件设计方法主要有以下几个主要特征。

1.1.1 采用自下至上(Bottom Up)的设计方法

自下至上的硬件电路设计方法的主要步骤是：根据系统对硬件的要求，详细编制技术规格书，并画出系统控制流图；然后根据技术规格书和系统控制流图，对系统的功能进行细化，合理地划分功能模块，并画出系统的功能框图；接着就是进行各功能模块的细化和电路设计；各功能模块电路设计、调试完成后，将各功能模块的硬件电路连接起来再进行系统的调试，最后完成整个系统的硬件设计。

自下至上的设计方法在各功能模块的电路设计中的体现大概最能说明问题。下面以一个六进制计数器设计为例作一说明。

要设计一个六进制计数器，其方案是多种多样的，但是摆在设计者面前的一个首要问题是，如何选择现有的逻辑元、器件构成六进制计数器。那么，设计六进制计数器将首先从选择逻辑元、器件开始。

第一步，选择逻辑元、器件。由数字电路的基本知识可知，可以用与非门，或非门，D 触发器，JK 触发器等基本逻辑元、器件来构成一个计数器。设计者根据电路尽可能简单，

价格合理，购买和使用方便及各自的习惯来选择构成六进制计数器的元、器件。本例中我们选择 JK 触发器和 D 触发器作为构成六进制计数器的主要元、器件。

第二步，进行电路设计。假设六进制计数器采用约翰逊计数器。3 个触发器连接应该产生 8 种状态，现在只使用 6 个状态，将其中的 010 和 101 两种状态禁止掉。这样六进制计数器的状态转移图如图 1 - 1 所示。

从这个状态转移图可以看到，在计数过程中计数器中的 3 个触发器的状态是这样转移的：首先 3 个触发器状态均为 0，即 $Q_2Q_1Q_0=000$，以后每来一个计数脉冲，其状态变化情况为 000→001→011→111→110→100→000→001→……。

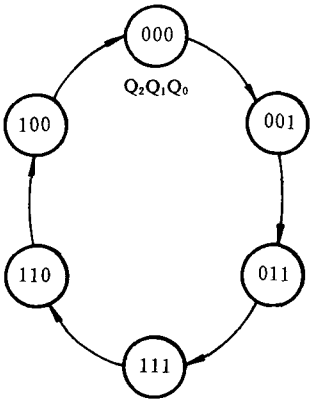


图 1 - 1　六进制计数器状态转移图

在知道六进制计数器的状态变化规律以后，就可以列出每个触发器的前一个状态和后一个状态变化的状态表，如表 1 - 1 所示。从表中可以发现，Q_2 当前的输出是 Q_1 前一个状态的输出，而 Q_1 当前的输出就是 Q_0 前一个状态的输出。这样，如 Q_2 和 Q_1 采用 D 触发器，只要将 Q_0 输出与 D_1 触发器的 d 输入端相连接，将 D_1 输出(Q_1)与 D_2 触发器的 d 输入端相连接就行了。Q_0 输出关系复杂一些，就必须选用 JK 触发器，并且利用 Q_1，Q_2 输出作为约束条件，经组合逻辑电路作为 Q_0 的 J 和 K 的输入。Q_2，Q_1 输出和 Q_0 的 J，K 输入关系如表 1 - 2 所示。从表 1 - 2 就很容易写出以 Q_2，Q_1 为输入，J，K 为输出的两个真值表。该真值表实际上就是或非门的真值表和与门的真值表。那么，将 Q_2，Q_1 分别连到或非门的输入端，将或非门的输出连到 Q_0 的 J 输入端；再将 Q_2，Q_1 分别连接到与门的输入端，将与门的输出端与 Q_0 的 K 输入端相连。这样，一个六进制计数器的硬件电路设计就完成了，如图 1 - 2 所示。当然，触发器的时钟端应和计数脉冲端相连接，系统复位信号应和触发器的置“0”端相连接，这样就可以保证实际电路的正常工作。

表 1 - 1　触发器状态变化表

触发器状态 / 计数脉冲	Q_2		Q_1		Q_0	
	前一状态	当前状态	前一状态	当前状态	前一状态	当前状态
1	0	0	0	0	0	1
2	0	0	0	1	1	1
3	0	1	1	1	1	1
4	1	1	1	1	1	0
5	1	1	1	0	0	0
6	1	0	0	0	0	0

表 1－2　Q_2，Q_1 输出和 Q_0 的 J，K 输入关系表

触发器状态 / 计数脉冲	Q_2	Q_1	Q_0			
	前一状态	前一状态	J	K	前一状态	当前状态
1	0	0	1	0	0	1
2	0	0	1	0	1	1
3	0	1	0	0	1	1
4	1	1	0	1	1	0
5	1	1	0	1	0	0
6	1	0	0	0	0	0

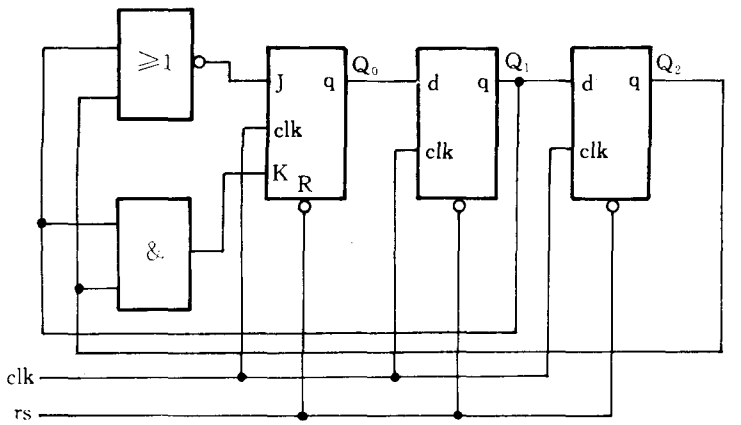


图 1－2　六进制约翰逊计数器原理图

与六进制计数器模块设计一样，系统的其它模块也按此方法进行设计。在所有硬件模块设计完成以后，再将各模块连接起来，进行调试，如有问题则进行局部修改，直至整个系统调试完毕为止。

从上述设计过程我们可以看到，系统硬件的设计是从选择具体元、器件开始的，并用这些元、器件进行逻辑电路设计，完成系统各独立功能模块设计，然后再将各功能模块连接起来，完成整个系统的硬件设计。上述过程从最底层开始设计，直至到最高层设计完毕，故将这种设计方法称为自下至上的设计方法。

1.1.2　采用通用的逻辑元、器件

在传统的硬件电路设计中，设计者总是根据系统的具体需要，选择市场上能买到的逻辑元、器件，来构成所要求的逻辑电路，从而完成系统的硬件设计。尽管随着微处理器的出现，在由微处理器及其相应硬件构成的系统中，许多系统的硬件功能可以用软件功能来实现，从而在较大程度上简化了系统硬件电路的设计；但是，这种选择通用的元、器件来构成系统硬件电路的方法并未改变。

1.1.3 在系统硬件设计的后期进行仿真和调试

在传统的系统硬件设计方法中，仿真和调试通常只能在后期完成系统硬件设计以后，才能进行。因为进行仿真和调试的仪器一般为系统仿真器、逻辑分析仪和示波器等。因此只有在硬件系统已经构成以后才能使用。系统设计时存在的问题只有在后期才能较容易发现。这样，传统的硬件设计方法对系统设计人员有较高的要求。一旦考虑不周，系统设计存在较大缺陷，那么就有可能要重新设计系统，使得设计周期也大大增加。

1.1.4 主要设计文件是电原理图

在用传统的硬件设计方法对系统进行设计并调试完毕后，所形成的硬件设计文件，主要是由若干张电原理图构成的文件。在电原理图中详细标注了各逻辑元、器件的名称和互相间的信号连接关系。该文件是用户使用和维护系统的依据。对于小系统，这种电原理图只要几十张至几百张就行了。但是，如果系统比较大，硬件比较复杂，那么这种电原理图可能要有几千张、几万张，甚至几十万张。如此多的电原理图给归档、阅读、修改和使用都带来了极大的不方便。

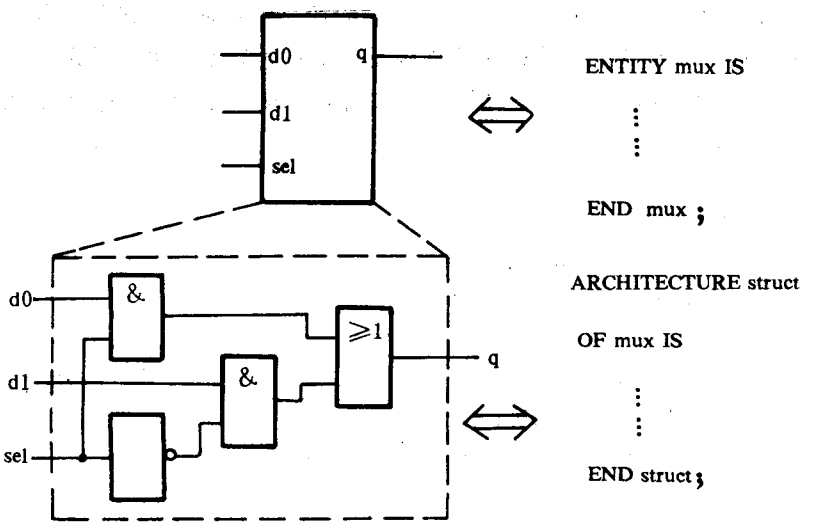
传统的硬件电路设计方法已经沿用几十年，是目前广大电子工程师所熟悉和掌握的一种方法。但是，随着计算机技术、大规模集成电路技术的发展，这种传统的设计方法已大大落后于当今技术的发展。一种崭新的，采用硬件描述语言的硬件电路设计方法已经兴起，它的出现将给硬件电路设计带来一次重大的变革。

1.2 利用硬件描述语言(HDL)的硬件电路设计方法

在硬件电路设计中采用计算机辅助设计技术(CAD)，一般来说到 80 年代才得到了普及和应用。在一开始，仅仅是利用计算机软件来实现印刷板的布线，以后慢慢地才实现了插件板级规模的电子电路的设计和仿真。在我国所使用的工具中，最有代表性的设计工具是 Tango 和早期的 ORCAD。它们的出现，使得电子电路设计和印刷板布线工艺实现了自动化。但是，就其设计方法而言，仍是自下至上的设计方法，利用已有的逻辑元、器件来构成硬件电路。

随着大规模专用集成电路(ASIC)的开发和研制，为了提高开发的效率，增加已有开发成果的可继承性以及缩短开发时间，各 ASIC 研制和生产厂家相继开发了用于各自目的的硬件描述语言。其中最有代表性的是美国国防部开发的 VHDL 语言(VHSIC Hardware Description Language)，Verilog 公司开发的 Verilog HDL 以及日本电子工业振兴协会开发的 UDL/I 语言。

所谓硬件描述语言，就是可以描述硬件电路的功能，信号连接关系及定时关系的语言。它能比电原理图更有效地表示硬件电路的特性。例如，一个二选一的选择器的电原理图如图 1 - 3(*a*)所示，而用 VHDL 语言描述的二选一的选择器如图 1 - 3(*b*)所示。利用硬件描述语言编程来表示逻辑器件及系统硬件的功能和行为，这是该设计方法的一个重要特征。



(*a*)

```
ENTITY mux IS
PORT(d0, d1, sel: IN BIT;
                q: OUT BIT);
END mux;
ARCHITECTURE connect OF mux IS
BEGIN
  calc: PROCESS(d0, d1, sel)
    VARIABLE tmp1, tmp2, tmp3: BIT;
  BEGIN
    tmp1:=d0 AND sel;
    tmp2:=d1 AND (NOT sel);
    tmp3:=tmp1 OR tmp2;
    q<=tmp3;
  END PROCESS;
END connect;
```

(*b*)

图 1-3　二选一选择器描述

(*a*) 电原理图表示；(*b*) 用 VHDL 语言描述

利用 HDL 语言设计系统硬件的方法，归纳起来有以下几个特点。

1.2.1　采用自上至下(Top Down)的设计方法

所谓自上至下的设计方法，就是从系统总体要求出发，自上至下地逐步将设计内容细化，最后完成系统硬件的整体设计。在利用 HDL 的硬件设计方法中，设计者将自上至下分

成 3 个层次对系统硬件进行设计。

第一层次是行为描述。所谓行为描述，实质上就是对整个系统的数学模型的描述。一般来说，对系统进行行为描述的目的是试图在系统设计的初始阶段，通过对系统行为描述的仿真来发现设计中存在的问题。在行为描述阶段，并不真正考虑其实际的操作和算法用什么方法来实现。考虑更多的是系统的结构及其工作过程是否能达到系统设计规格书的要求。下面仍以六进制计数器为例，说明一下如何用 VHDL 语言，以行为方式来描述它的工作特性，其实例如例 1 - 1 所示。

【例 1 - 1】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY counter IS
PORT(
      clk: IN STD_LOGIC;
       rs: IN STD_LOGIC;
      count_out: OUT STD_LOGIC_VECTOR(2 DOWNTO 0));
END counter;
ARCHITECTURE behav OF counter IS
   SIGNAL next_count: STD_LOGIC_VECTOR(2 DOWNTO 0);
BEGIN
   count_proc: PROCESS(rs, clk)
BEGIN
  IF rs='0' THEN
     mext_count<="000";
  ELSIF (clk'EVENT AND clk='1') THEN
     CASE next_count IS
     WHEN "000"=>next_count<="001";
     WHEN "001"=>next_count<="011";
     WHEN "011"=>next_count<="111";
     WHEN "111"=>next_count<="110";
     WHEN "110"=>next_count<="100";
     WHEN "100"=>next_count<="000";
     WHEN OTHERS=>next_count<="XXX";
   END CASE;
  END IF;
   count_out<=next_count AFTER 10 ns;
  END PROCESS;
END behav;
```

从例 1 - 1 可以看出，该段 VHDL 语言程序勾画出了六进制计数器的输入输出引脚和内部计数过程的计数状态变化时序和关系。这实际上是计数器工作模型的描述。当该程序仿真通过以后，说明六进制计数器模型是正确的。在此基础上再改写该程序，使其语句表达式易于用逻辑元件来实现。这是第二层次所要做的工作。

第二层次是 RTL 方式描述。这一层次称为寄存器传输描述(又称数据流描述)。如前所述，用行为方式描述的系统结构的程序，其抽象程度高，是很难直接映射到具体逻辑元件结构的硬件实现的。要想得到硬件的具体实现，必须将行为方式描述的 VHDL 语言程序改写为 RTL 方式描述的 VHDL 语言程序。也就是说，系统采用 RTL 方式描述，才能导出系统的逻辑表达式，才能进行逻辑综合。当然，这里所说的"可以"进行逻辑综合是有条件的，它是针对某一特定的逻辑综合工具而言的。与例 1 - 1 行为方式描述所等价的六进制计数器的 RTL 描述，如例 1 - 2 所示。

【例 1 - 2】

```
LIBRARY IEEE;
USE IEEE. STD_LDGIC_1164. ALL;
USE WORK. NEW. ALL;
ENTITY counter IS
PORT(clk, rs: IN STD_LOGIC;
     q1, q2, q3: OUT STD_LOGIC);
END counter;
ARCHITECTURE rtl OF counter IS
COMPONENT dff
 PORT(d, rs, clk: IN STD_LOGIC;
              q: OUT STD_LOGIC);
END COMPONENT;
COMPONENT djk
PORT(j,k,rs,clk: IN STD_LOGIC;
            q: OUT STD_LOGIC);
END COMPONENT;
COMPONENT and2
  PORT(a,b: IN STD_LOGIC;
        c: OUT STD_LOGIC);
END COMPONENT;
COMPONENT nor2
  PORT(a,b: IN STD_LOGIC;
        c: OUT STD_LOGIC);
END COMPONENT;
SIGNAL jin, kin, q1_out, q2_out, q3_out:
       STD_LOGIC;
BEGIN
  u1: nor2
    PORT MAP(q3_out, q2_out, jin);
  u2: and2
    PORT MAP(q3_out, q2_out, kin);
  u3: djk
    PORT MAP(jin, kin, rs, clk, q1_out);
  u4: dff
```

```
    PORT MAP(q1_out, rs, clk, q2_out);
  u5: dff
    PORT MAP(q2_out, rs, clk, q3_out);
  q1<=q1_out;
  q2<=q2_out;
  q3<=q3_out;
END rtl;
```

在该例中，JK 触发器、D 触发器、与门和或非门都已在库 WORK. NEW. ALL 中定义了，这里可以直接引用。例中的构造体直接描述了它们之间的连接关系。与例 1 - 1 相比，例 1 - 2 更趋于实际电路的描述。

在把行为方式描述的程序改写为 RTL 方式描述的程序时，编程人员必须深入了解逻辑综合工具的详细说明和具体规定，这样才能编写出合格的 RTL 方式描述的程序。

在完成编写 RTL 方式的描述程序以后，再用仿真工具对 RTL 方式描述的程序进行仿真。如果通过这一步仿真，那么就可以利用逻辑综合工具进行综合了。

第三层次是逻辑综合。逻辑综合这一阶段是利用逻辑综合工具，将 RTL 方式描述的程序转换成用基本逻辑元件表示的文件(门级网络表)。此时，如果需要，可以将逻辑综合结果，以逻辑原理图方式输出。也就是说，逻辑综合的结果相当于在人工设计硬件电路时，根据系统要求画出了系统的逻辑电原理图。此后对逻辑综合结果在门电路级上再进行仿真，并检查定时关系。如果一切都正常，那么系统的硬件设计就基本结束。如果在 3 个层次的某个层次上发现有问题，都应返回上一层，寻找和修改相应的错误，然后再向下继续未完的工作。

由逻辑综合工具产生门级网络表后，在最终完成硬件设计时，还可以有两种选择。第一种是由自动布线程序将网络表转换成相应的 ASIC 芯片的制造工艺，做出 ASIC 芯片。第二种是将网络表转换成 FPGA(现场可编程门阵列)的编程码点，利用 FPGA 完成硬件电路设计。

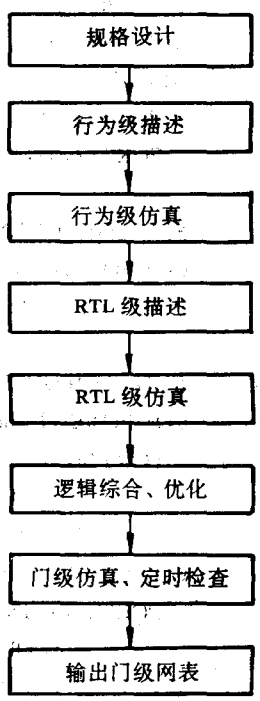


图 1 - 4　自上至下设计系统硬件的过程

在用 HDL 语言设计系统硬件时，无论是设计一个局部电路，还是设计由多块插件板组成的复杂系统，上述自上至下的 3 个层次的设计步骤是必不可少的。利用自上至下设计系统硬件的过程如图 1 - 4 所示。

由自上至下的设计过程可知，从总体行为设计开始到最终逻辑综合，形成网络表为止，每一步都要进行仿真检查，这样有利于尽早发现系统设计中存在的问题，从而可以大大缩短系统硬件的设计周期。这是用 HDL 语言设计系统硬件的最突出的优点之一。

1.2.2 系统中可大量采用 ASIC 芯片

由于目前众多的制造 ASIC 芯片的厂家，它们的工具软件都可支持 HDL 语言的编程，因此，硬件设计人员在设计硬件电路时，无须受只能使用通用元、器件的限制，而可以根据硬件电路设计需要，设计自用的 ASIC 芯片或可编程逻辑器件。这样最终会使系统电路设计更趋合理，体积也可大为缩小。

1.2.3 采用系统早期仿真

从自上至下的设计过程可以看到，在系统设计过程中要进行三级仿真，即行为层次仿真、RTL 层次仿真和门级层次仿真。也就是说进行系统数学模型的仿真、系统数据流的仿真和系统门电路电原理的仿真。这 3 级仿真贯穿系统硬件设计的全过程，从而可以在系统设计早期发现设计中存在的问题。与自下至上设计的后期仿真相比可大大缩短系统的设计周期，节约大量的人力和物力。

1.2.4 降低了硬件电路设计难度

在采用传统的硬件电路设计方法时，往往要求设计者在设计电路前应写出该电路的逻辑表达式或真值表(或时序电路的状态表)。这一工作是相当困难和繁杂的，特别是在系统比较复杂时更是如此。例如，在设计六进制计数器时，必须编写输入和输出的真值表和状态表。根据表中的关系，写出逻辑表达式，并用相应的逻辑元件来实现。

在用 HDL 语言设计硬件电路时，就可以使设计者免除编写逻辑表达式或真值表之苦。如图 1－1 和例 1－1 所示，只要知道六进制计数器的 6 个计数状态就行了，而无须写出相关电路的逻辑表达式。这样使硬件电路的设计难度有了大幅度的下降，从而也缩短了硬件电路的设计周期。据有关资料估计，仅此一项可使设计周期大约缩短 1/3～1/2。

1.2.5 主要设计文件是用 HDL 语言编写的源程序

在传统的硬件电路设计中，最后形成的主要文件是电原理图，而采用 HDL 语言设计系统硬件电路时，主要的设计文件是用 HDL 语言编写的源程序。如果需要也可以转换成电原理图形式输出。用 HDL 语言的源程序作为归档文件有很多好处。其一是资料量小，便于保存。其二是可继承性好。当设计其它硬件电路时，可以使用文件中的某些库、进程和过程等描述某些局部硬件电路的程序。其三是阅读方便。阅读程序比阅读电原理图要更容易一些。阅读者很容易在程序中看出某一硬件电路的工作原理和逻辑关系。而阅读电原理图，推知其工作原理却需要较多的硬件知识和经验，而且看起来也不那么一目了然。

1.3 利用 VHDL 语言设计硬件电路的优点

当前各 ASIC 芯片制造商都相继开发了用于各自目的的 HDL 语言，但是大多都未标准化和通用化。唯一已被公认的是美国国防部开发的 VHDL 语言，它已成为 IEEE STD-1076标准。另外，从近期 HDL 语言发展的动态来看，许多公司研制的硬件电路设计工具也都逐渐向 VHDL 语言靠拢，使得它们的硬件电路设计工具也能支持 VHDL 语言。

Verilog 公司开发的 Verilog_HDL 语言目前使用也较广泛，据资料介绍近期也有可能成为另一种标准的 HDL 语言。但是，从整体来看 Verilog_HDL 语言不如 VHDL 语言。

VHDL 语言和其它 HDL 语言相比到底有哪些特色呢？下面作一简要说明。

1.3.1 设计技术齐全、方法灵活、支持广泛

VHDL 语言可以支持自上至下(Top Down)和基于库(Library-Based)的设计方法，而且还支持同步电路、异步电路、FPGA 以及其它随机电路的设计。其范围之广是其它 HDL 语言所不能比拟的。例如，SFL 语言和 UDL/ I 语言，它们只能描述同步电路。另外，由于 VHDL 语言早在 1987 年 12 月已作为 IEEE_STD_1076 标准公开发布。因此，目前大多数 EDA 工具几乎在不同程度上都支持 VHDL 语言。这样给 VHDL 语言进一步推广和应用创造了良好的环境。

1.3.2 系统硬件描述能力强

如前所述，VHDL 语言具有多层次描述系统硬件功能的能力，可以从系统的数学模型直到门级电路。另外，高层次的行为描述可以与低层次的 RTL 描述和结构描述混合使用。例如，在 PC 机扩展槽上要设计一块接口卡，该接口卡的硬件设计应满足主机的接口要求。此时，主机部分功能可以用行为方式描述，而接口卡可以采用 RTL 方式描述。在系统仿真时就可以验证接口卡的工作是否正确。这样，在接口卡设计出来以前就可以知道接口卡的工作是否满足系统要求。

VHDL 语言能进行系统级的硬件描述，这是它的一个最突出的优点。其它 HDL 语言，如 UDL/ I 、Verilog 等只能进行 IC 级、PCB 级描述，而不能对系统级的硬件很好地进行描述。

再如，VHDL 语言可以自定义数据类型，这样也给编程人员带来了较大的自由和方便。

1.3.3 VHDL 语言可以与工艺无关编程

在用 VHDL 语言设计系统硬件时，没有嵌入与工艺有关的信息。当然，这样的信息是可以用 VHDL 语言来编写的。与大多数 HDL 语言的不同之处是，当门级或门级以上层次的描述通过仿真检验以后，再用相应的工具将设计映射成不同的工艺(如 MOS、CMOS 等)。这样，在工艺更新时，就无须修改原设计程序，只要改变相应的映射工具就行了。由此可见，无论修改电路还是修改工艺相互之间不会产生什么不良影响。

1.3.4 VHDL 语言标准、规范，易于共享和复用

由于 VHDL 语言已作为一种 IEEE 的工业标准，这样，设计成果便于复用和交流，反过来就能更进一步推动 VHDL 语言的推广及完善。另外，VHDL 语言的语法比较严格，其风格类似于 Ada 语言，给阅读和使用都带来了极大的好处。

第 2 章

VHDL 语言程序的基本结构

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。前 4 种是可分别编译的源设计单元。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；库存放已经编译的实体、构造体、包集合和配置。库可由用户生成或由 ASIC 芯片制造商提供，以便于在设计中为大家所共享。本章将对上述 VHDL 设计的主要构成作一详细介绍。

2.1 VHDL 语言设计的基本单元及其构成

所谓 VHDL 语言设计的基本单元(Design Entity)，就是 VHDL 语言的一个基本设计实体。一个基本设计单元，简单的可以是一个与门(AND Gate)，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。它们都由实体说明(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操作(行为)。图 2 - 1 示出了作为一个设计单元的二选一电路的 VHDL 描述。由图 2 - 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。

```
ENTITY mux IS
GENERIC(m:TIME := 1 ns);
PORT(d0, d1, sel: IN BIT;
                 q: OUT BIT);
END mux;
ARCHITECTURE connect OF mux IS
SIGNAL tmp: BIT;
BEGIN
  cale: PROCESS(d0, d1, sel)
    VARIABLE tmp1, tmp2, tmp3: BIT;
  BEGIN
```

```
    tmp1:=d0 AND sel;
    tmp2:=d1 AND (NOT sel);
    tmp3:=tmp1 OR tmp2;
    tmp<=tmp3;
      q<=tmp AFTER m;
  END PROCESS;
END connect;
```

图 2-1　一个基本设计单元的构成

下面以二选一器件描述为例，说明一下这两部分的具体书写规定。

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。例如在图 2-1 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。小写字母是设计者添写的部分，随设计单元不同而不同。实际上，对 VHDL 而言，大写或小写都一视同仁，不加区分。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，例如图 2-1 中的 GENERIC (m:TIME:=1 ns)。该语句指定了构造体内 m 的值为1 ns。这样语句

```
tmp1:=d0 AND sel AFTER m;
```

表示 d0和 sel 两个输入信号相与后，经1 ns 延迟才送到 tmp1。在这个例子中，GENERIC 利用类属参数为 tmp1建立一个延迟值。

2. 端口说明

端口说明是对基本设计实体(单元)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。其一般书写格式如下：

```
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。例如图 2-1 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，图 2-1 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。“LINKAGE”用以说明该端口无指定方向，可以与任何方向的信号相连接。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

方向定义	含　义
IN	输入
OUT	输出(构造体内部不能再使用)
INOUT	双向
BUFFER	输出(构造体内部可再使用)
LINKAGE	不指定方向，无论哪一个方向都可连接

注：OUT 允许对应多个信号，而 BUFFER 只允许对应一个信号

表 2－1 中“OUT”和“BUFFER”都可以定义输出端口，但是它们之间是有区别的，如图 2－2 所示。

在图 2－2(*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 2－2(*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。

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：BIT 和 BIT_VECTOR。

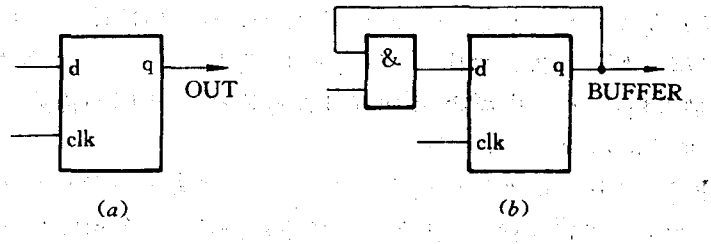


图 2－2　OUT 和 BUFFER 的区别

(*a*) OUT；(*b*) BUFFER

当端口被说明为 BIT 数据类型时，该端口的信号取值只可能是“1”或“0”。注意，这里的“1”和“0”是指逻辑值。所以 BIT 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 BIT_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 BIT_VECTOR。总线端口上的值由 8 位二进制位的值所确定。较完整的端口说明如例 2 - 1 所示。

【例 2 - 1】

```
PORT(d0, d1, sel: IN BIT;
                q: OUT BIT;
              bus: OUT BIT_VECTOR(7 DOWNTO 0));
```

该例中 d0,d1,sel,q 都是 BIT 数据类型，而 bus 是 BIT_VECTOR 类型，(7 DOWNTO 0)表示该 bus 端口是一个 8 位端口，由 B_7～B_0 8 位构成。位矢量长度为 8 位。

在某些 VHDL 语言的程序中，数据类型的说明符号有所不同。仍以例 2 - 1 为例进行说明。

【例 2 - 2】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY mu IS
  PORT(d0, d1, sel: IN STD_LOGIC;
                 q: OUT STD_LOGIC;
               bus: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END mu;
```

该例中 BIT 类型用 STD_LOGIC 说明，而 bus 则用 STD_LOGIC_VECTOR (7 DOWNTO 0)说明。上述两例的描述实际上是完全等效的。在 VHDL 语言中存在一个库，该库有一个包集合，专门对数据类型作了说明，其作用像 C 语言中的 include 文件一样。这样做主要为了标准和统一。但是，在用 STD_LOGIC 和 STD_LOGIC_VECTOR 说明时，在实体说明以前必须增加例中所示的两个语句，以便在对 VHDL 语言程序编译时，从指定库的包集合中寻找数据类型的定义。

2.1.2 构造体

构造体是一个基本设计单元的实体，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。

构造体的名称可以由设计者自由命名。但是在大多数的文献和资料中，通常把构造体的名称命名为 behavioral(行为)，dataflow(数据流)或者 structural(结构)。如前所述，这 3 个名称实际上是 3 种构造体描述方式的名称。当设计者采用某一种描述方式来描述构造体时，该构造体的结构名称就命名为那一个名称。这样，使得阅读 VHDL 语言程序的人能直接了解设计者所采用的描述方式。例如，使用结构描述方式来描述二选一电路，那么二选一电路的构造体就可以这样命名：

```
ARCHITECTURE structural OF mux IS
```

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 nes1:BIT;
    ⋮
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。例如，二选一的数据流方式描述可以写为：

【例 2 - 3】

```
ENTITY mux IS
  PORT (d0, d1:IN BIT;
        sel:IN BIT;
        q:OUT BIT);
END mux;
ARCHITECTURE dataflow OF mux IS
BEGIN
  q<=(d0 AND sel)OR(NOT sel AND d1);
END dataflow;
```

在该程序的构造体中所使用的语句，实际上是二选一的逻辑表达式的描述语句。它正确地反映了二选一器件的行为。这种语句和其它高级语言是相当类似的，读者只要有一点基本的高级语言知识就可以读懂。在语句中，符号“<=”表示传送(或代入)的意思，即将逻辑运算结果送 q 输出。

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 语言构造体的子结构描述

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非常不方便的。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，一个构造体可以用几个子结构，即相对比较独立的几个模块来构成。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。

下面就上述 3 种子结构作一说明。

2.2.1 BLOCK 语句结构描述

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 2-4】

```
ENTITY mux IS
PORT(d0, d1, sel: IN BIT;
                 q: OUT BIT);
END mux;
ARCHITECTURE connect OF mux IS
   SIGNAL tmp1, tmp2, tmp3: BIT;
BEGIN
  cale:
BLOCK
  BEGIN
    tmp1<=d0 AND sel;
    tmp2<=d1 AND (NOT sel);
    tmp3<=tmpI OR tmp2;
    q<=tmp3;
  END BLOCK cale;
END connect;
```

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

2. BLOCK 块和子原理图的关系

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一一对应的。一个具体实例如图 2－3 所示。

在图 2－3 的左边，有一张电原理图，它被分成 4 个子原理图。在图 2－3 的右边是用 VHDL 语言书写的电路设计程序。该程序中的构造体由 4 个 BLOCK 语句构成，它们分别对应被分割的子原理图。

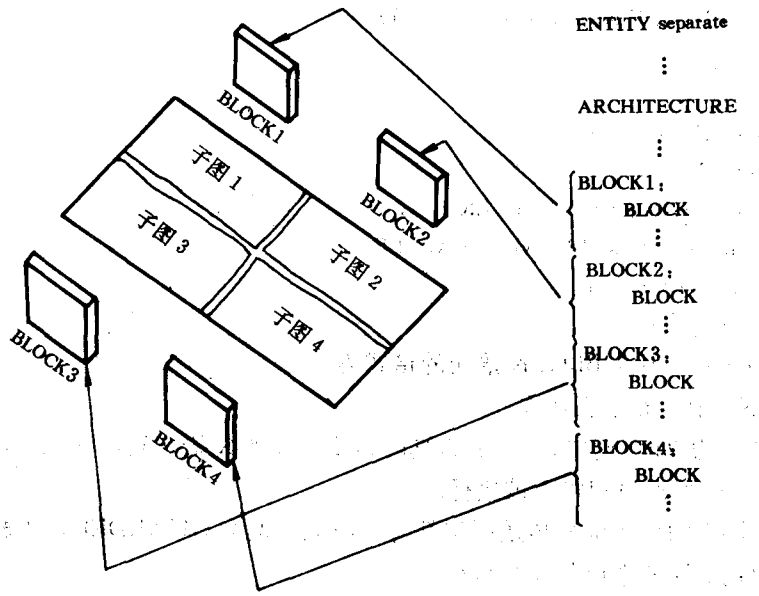


图 2－3　BLOCK 块和子原理图关系

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。

3. BLOCK 中语句的并发性

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行执行的，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，这一点将在后面再介绍。

4. 卫式 BLOCK(Guarded BLOCK)

在图 2－3 中使用 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。如前所述，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，往往会碰到这样情况，当某一种条件得到满足时，BLOCK语句才可以被执行；而条件不满足时该BLOCK语句将不能执行。这就是卫式BLOCK，它可以实现BLOCK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，用卫式BLOCK语句描述该锁存器结构的VHDL语言程序可以书写为：

【例2-5】

```
ENTITY latch IS
  PORT(d, clk: IN BIT;
       q, qb: OUT BIT);
END latch;
ARCHITECTURE latch_guard OF latch IS
BEGIN
    G1:
    BLOCK(clk='1')
    BEGIN
      q<=GUARDED d AFTER 5 ns;
      qb<=GUARDED NOT(d) AFTER 7 ns;
    END BLOCK G1;
END latch_guard;
```

如上述程序所示，卫式BLOCK语句的格式为：

BLOCK [卫式布尔表达式]

当卫式布尔表达式为真时(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块中的两个信号传送语句都写有前卫关键词GUARDED，这表明只有卫式布尔表达式为真时，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，卫式布尔表达式为真。d端的输入值经5 ns延迟以后从q端输出，并且对d端的值取反，经7 ns后从qb端输出。当端口clk的值为"0"时，d端到q，qb端的信号传递通道将被切断，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(信号1, 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 2－6】

```
ENTITY mux IS
PROT(d0, d1, sel: IN BIT;
                q: OUT BIT);
END mux;
ARCHITECTURE connect OF mux IS
BEGIN
  cale: PROCESS(d0, d1, sel)
      VARIABLE tmp1, tmp2, tmp3: BIT;
  BEGIN
      tmp1:=d0 AND sel;
      tmp2:=d1 AND (NOT sel);
      tmp3:=tmp1 OR tmp2;
      q<=tmp3;
  END PROCESS;
END connect;
```

程序中 tmp1，tmp2 和 tmp3 是变量，变量只在进程中定义，详细说明后面再述。

2. 进程(PROCESS)中语句的顺序性

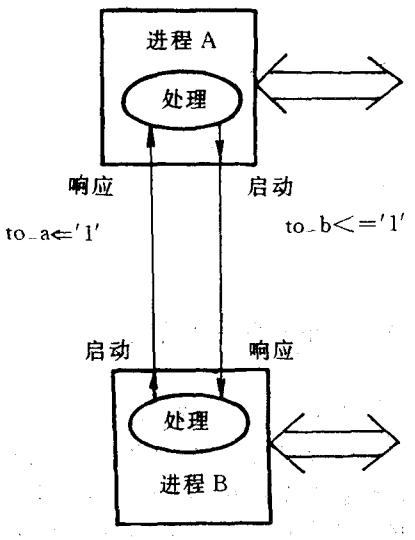
在 VHDL 中，与 BLOCK 语句一样，某一个功能独立的电路，在设计时也可以用一个 PROCESS 语句结构来描述。与 BLOCK 语句不同的是，在系统仿真时，PROCESS 结构中的语句是按顺序一条一条向下执行的，而不像 BLOCK 中的语句可以并行执行。这一点与单处理机上执行 C 语言和 Pascal 语言的语句是完全一样的。在后面还会提到，在 VHDL 语言中，这种顺序执行的语句只在 PROCESS 和 SUBPROGRAMS 的结构中使用。

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，在书写时跟在“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 语言中也称敏感量。这些信号无论哪一个发生变化(如由“0”变“1”或者由“1”变“0”)都将启动该 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。

4. 进程(PROCESS)的同步描述

在上例用 PROCESS 结构描述的二选一电路的程序中，构造体内部只存在一个 PROCESS。但是在实际的程序设计中，同一个构造体中可以有多个进程存在，而且各 PROCESS 之间还可以一边进行通信，一边并行地同步执行。下面以图 2－4 为例作一说明。




```
ENTITY pros_com IS
PORT(event_a: IN BIT);
END pros_com;
ARCHITECTURE catch_ball OF
        pros_com IS
SIGNAL to_a, to_b: BIT:='0';
BEGIN
A:
PROCESS(event_a, to_a)
BEGIN
  IF (event_a' EVENT AND event_a='1')
       OR
    (to_a' EVENT AND to_a='1')
  THEN
    to_b<='1' AFTER 20 ns,
          '0' AFTER 30 ns;
  END IF;
END PROCESS A;
B:
PROCESS(to_b)
BEGIN
  IF(to_b' EVENT AND to_b='1')
  THEN
    to_a<='1' AFTER 10 ns,
          '0' AFTER 20 ns;
  END IF;
END PROCESS B;
END catch_ball;
```

图 2-4　进程的同步描述

如图 2-4 左边框图所示，在一个构造体中存在着两个进程 A 和 B。A 进程处理结束或者有了一个 B 进程启动所需要的数据，就使信号量 to_b='1'。to_b 是 B 进程的输入信号，当 B 进程敏感到 to_b 有变化，且 to_b='1'时，则 B 进程被启动。同样，B 进程处理结束或者有了一个 A 进程启动所需的数据，就使信号量 to_a='1'。to_a 是 A 进程的输入信号，当 A 进程敏感到 to_a 有变化，且 to_a='1'时，则 A 进程被启动。如此循环工作，使 A 进程和 B 进程能并行地同步工作。

2.2.3　子程序(SUBPROGRAM)语句结构描述

所谓子程序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。

在 VHDL 中子程序有两种类型：

- 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

1) 过程语句的结构

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。

【例 2 - 7】

```
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

【例 2 - 8】

```
PROCEDURE shift(
```

```
        din: IN STD_LOGIC_VECTOR;
    SIGNAL dou: OUT STD_LOGIC_VECTOR);
            ⋮
END shift;
```

2）过程结构中语句的顺序性

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

1）函数语句的结构

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。例如：

【例 2 - 9】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
PACKAGE bpac IS
  FUNCTION max(a: STD_LOGIC_VECTOR;
               b: STD_LOGIC_VECTOR)
               RETURN STD_LOGIC_VECTOR;
END bpac
PACKAGE BODY bpac IS
  FUNCTION max(a: STD_LOGIC_VECTOR;
               b: STD_LOGIC_VECTOR) RETURN STD_LOGIC_VECTOR IS
  VARIABLE tmp: STD_LOGIC_VECTOR(a' RANGE);
  BEGIN
    IF (a>b) THEN
       tmp:=a;
    ELSE
       tmp:=b;
    END IF;
```

```
    RETURN tmp;
  END max;
END bpac;
```

2）函数调用及结果的返回

在 VHDL 语言中，函数语句可以在构造体的语句中直接调用。例2 - 10给出，用 FUNCTION 语句描述最大值检出的程序。

【例 2 - 10】

```
LIBRARY IEEE.NEWLIB;
USE IEEE.STD_LOGIC_1164.ALL;
USE NEWLIB.bpac.ALL;
ENTITY peakdetect IS
  PORT (data: IN STD_LOGIC_VECTOR(5 DOWNTO 0);
        clk, set: IN STD_LOGIC;
        dataout: OUT STD_LOGIC_VECTOR(5 DOWNTO 0));
END peakdetect
ARCHITECTURE rtl OF peakdetect IS
  SIGNAL peak: STD_LOGIC_VECTOR(5 DOWNTO 0);
BEGIN
  dataout<=peak;
  PROCESS(clk)
    BEGIN
      IF(clk'EVENT AND clk='1') THEN
      IF(set='1') THEN
        peak<=data;
      ELSE
        peak<=MAX(data, peak);
      END IF;
    END IF;
  END PROCESS;
END rtl;
```

在上述程序中，peak<=MAX(data, peak)就是调用 FUNCTION 的语句。在包集合中的参数 a 和 b，在这里用 data 和 peak 所替代。函数的返回值 tmp 被赋予 peak。在 MAX (a, b)函数的定义中，返回值 tmp 可以赋予信号或者变量，在本例中被赋予信号 peak。

上面详细叙述了子程序中过程、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将在下一节中介绍。但是，需要指出的是，不同公司发布的包集合和库的登记方法是各不相同的。

2.3 包集合、库及配置

除了实体和构造体之外，包集合、库及配置是在 VHDL 语言中另外 3 个可以各自独立

进行编译的源设计单元。

2.3.1 库

库(Library)是经编译后的数据的集合，它存放包集合定义、实体定义、构造体定义和配置定义。

库的功能类似于 UNIX 和 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。由此可见，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，但是库和库之间是独立的，不能互相嵌套。

1. 库的种类

当前在 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、用户定义的库和 WORK 库。

1) IEEE 库

在 IEEE 库中有一个“STD_LOGIC_1164”的包集合，它是 IEEE 正式认可的标准包集合。现在有些公司，如 SYNOPSYS 公司也提供一些包集合“STD_LOGIC_ARITH”、“STD_LOGIC_UNSIGNED”，尽管它们没有得到 IEEE 的承认，但是仍汇集在 IEEE 库中。

2) STD 库

STD 库是 VHDL 的标准库，在库中存放有称为“STANDARD”的包集合。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 库中还包含有称作“TEXTIO”的包集合。在使用“TEXTIO”包集合中的数据时，应先说明库和包集合名，然后才可使用该包集合中的数据。例如，

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

该包集合在测试时使用。

3) 面向 ASIC 的库

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。为了使用面向 ASIC 的库，对库进行说明是必要的。

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，将都存放在 WORK 库中。在使用该库时无需进行任何说明。

5) 用户定义库

用户为自身设计需要所开发的共用包集合和实体等，也可以汇集在一起定义成一个库，这就是用户定义库或称用户库。在使用时同样要首先说明库名。

2. 库的使用

1) 库的说明

前面提到的 5 类库除 WORK 库和 STD 库之外，其它 3 类库在使用前都首先要作说明，第一个语句是“LIBRARY 库名”，表明使用什么库。另外还要说明设计者要使用的是

库中哪一个包集合以及包集合中的项目名(如过程名、函数名等)。这样第二个语句的格式如：

```
USE LIBRARY_name. package_name. ITEM. name
```

所以，一般在使用库时首先要用两条语句对库进行说明。例如：

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

上述表明，在该 VHDL 语言程序中要使用 IEEE 库中 STD_LOGIC_1164包集合的所有项目。这里，项目名为 ALL，表示包集合的所有项目都要用。

2) 库说明作用范围

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、配置为止。当一个源程序中出现两个以上的实体时，两条作为使用库的说明语句应在每个实体说明语句前重复书写。例如：

【例 2 - 11】

```
LIBRARY IEEE;
USE IEEE. STD_LOGIC_1164. ALL;      } 库使用说明
ENTITY and1 IS
        ⋮
END and1;
ARCHTECTURE rtl of and1 IS
        ⋮
END rtl;
CONFIGURATION s1 OF and1 IS
        ⋮
END s1;
LIBRARY IEEE;
USE IEEE. STD_LOGIC_1164. ALL;      } 库使用说明
ENTITY or1 IS
        ⋮
        ⋮
CONFIGURATION s2 OF or1 IS
        ⋮
END s2;
```

2.3.2 包集合

包集合(Package)说明像 C 语言中 include 语句一样，用来单纯地罗列 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。要使用包集合时可以用 USE 语句说明。例如：

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

该语句表示在 VHDL 程序中要使用名为 STD_LOGIC_1164的包集合中的所有定义或说

明项。

包集合的结构如下所示：

```
PACKAGE  包集合名  IS          ┐
  [说明语句];                  ├ 包集合标题
END  包集合名;                 ┘
PACKAGE BODY  包集合名  IS     ┐
  [说明语句];                  ├ 包集合体
END BODY;                      ┘
```

一个包集合由两大部分组成：包集合标题(Header)和包集合体。包集合体(Package Body)是一个可选项，也就是说，包集合可以只由包集合标题构成。一般包集合标题列出所有项的名称，而包集合体具体给出各项的细节。例如：

【例 2 - 12】

```
LIBRARY STD;
USE STD.STD_LOGIC.ALL;
PACKAGE math IS                                          ┐
  TYPE tw16 IS ARRAY(0 TO 15)OF T_WLOGIC;                ├ 包集合标题
  FUNCTION add(a, b: IN tw16) RETURN tw16;               ├
  FUNCTION sub(a, b: IN tw16) RETURN tw16;               ┘
END math;
PACKAGE BODY math IS                                     ┐
FUNCTION vect_to_int (s: tw16)                           │
      RETURN INTEGER IS                                  │
    VARIABLE result: INTEGER:=0;                         │
BEGIN                                                    │
    FOR i IN 0 TO 15 LOOP                                │
      result:=result * 2;                                │
      IF s(i)='1' THEN                                   │
        result:=result+1;                                │
      END IF;                                            │
    END LOOP;                                            │
    RETURN result;                                       ├ 包集合体
END vect_to_int;                                         │
FUNCTION int_to_tw16(s: INTEGER)                         │
RETURN tw16 IS                                           │
    VARIABLE result: tw16;                               │
    VARIABLE digit: INTEGER:=2 * * 15;                   │
    VARIABLE local: INTEGER;                             │
BEGIN                                                    │
    local:=s;                                            │
    FOR i IN 0 TO 15 LOOP                                │
      IF local/digit>=1 THNE                             │
        result(i):=1;                                    ┘
```

```
        local:=local-digit;
      ELSE
        result(i):=0;
      END IF;
        digit:=digit/2;
      END LOOP
      RETURN result;
    END int_to_tw16;
  FUNCTION add(a, b: IN tw16)
          RETURN tw16 IS
    VARIABLE result:  INTEGER;
  BEGIN
    result:=vect_to_int(a)+vect_to_int(b);
    RETURN int_to_tw16(result);
  END add;
  FUNCTION sub(a, b: IN tw16)
        RETURN tw16 IS
    VARIABLE result:INTEGER;
  BEGIN
    result:=vect_to_int(a)-vect_to_int(b);
    RETURN int_to_tw16 (result);
  END sub;
END math;
```

上面例子的包集合由包集合标题和包集合体两部分组成。在包集合标题中，定义了数据类型和函数的调用说明，而在包集合体中才具体地描述实现该函数功能的语句和数据的赋值。这种分开描述的好处是，当函数的功能需要作某些调整或数据赋值需要变化时，只要改变包集合体的相关语句就行了，而无须改变包标题的说明，这样可以使重新编译的单元数目尽可能少。

包集合也可以只有一个包集合标题说明，因为在包集合标题中也允许使用数据赋值和有实质性的操作语句。例如：

【例 2 - 13】

```
LIBRARY IEEE;
USE IEEE STD_LOGIC_1164. ALL;
PACKAGE upac IS
    CONSTANT k: INTEGER:=4;
    TYPE instruction IS(add, sub, adc, inc, srf, slf);
    SUBTYPE cpu_bus IS STD_LOGIC_VECTOR(k-1 DOWNTO 0);
END upac;
```

上述的包集合是用户自定义的。在该包集合中定义了 CPU 的指令这一数据类型和 cpu_bus 为一个 4 位的位矢量。由于它是用户自己定义的，因此编译以后就会自动地加到 WORK 库中，如要使用该包集合，则可用如下格式调用：

```
USE WORK. upac. inctruction;
```

2.3.3 配置

配置(Configuration)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构造体，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构造体，进行性能对比试验以得到性能最佳的构造体。例如，要设计一个二输入四输出的译码器。如果一种结构中的基本元件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。它们各自的构造体是不一样的，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体的选择。

配置语句的基本书写格式如下：

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

配置语句根据不同情况，其说明语句有简有繁，下面举几个例子作一些说明。

【例 2-14】 最简单的缺省配置格式结构：

```
CONFIGURATION  配置名  OF  实体名  IS
    FOR  选配构造体名
    END FOR;
END  配置名;
```

这种配置用于选择不包含块(BLOCK)和元件(COMPONENTS)的构造体。在配置语句中只包含有实体所选配的构造体名，其它什么也没有。典型的例子是对计数器实现多种形式的配置：

```
LIBRARY STD;
USE STD.STD_LOGIC.ALL;
ENTITY counter IS
    PORT(load, clear, clk: IN T_WLOGIC;
         data_in: IN INTEGER;
         data_out: OUT INTEGER);
END counter;
ARCHITECTURE count_255 OF counter IS
BEGIN
    PROCESS(clk)
        VARIABLE count: INTEGER:=0;
    BEGIN
        IF clear='1' THEN
          count:=0;
        ELSIF load='1' THEN
          count:=data_in;
        ELSIF(clk'EVENT) AND (clk='1') AND
                (clk'LAST_VALUE='0') THEN
          IF(count=255) THEN
```

```
            count:=0;
          ELSE
            count:=count+1;
          END IF;
        END IF;
      data_out<=count;
    END PROCESS;
  END count_255;
  ARCHITECTURE count_64K OF counter IS
  BEGIN
    PROCESS(clk)
      VARIABLE count: INTEGER:=0;
    BEGIN
      IF (clear='1') THNE
        count:=0;
      ELSIF load='1' THEN
        count:=data_in;
      ELSIF(clk'EVENT) AND (clk='1') AND
            (clk'LAST_VALUE='0') THEN
          IF(count=65535) THEN
            count:=0;
          ELSE
            count:=count+1;
          END IF;
    END IF;
  data_out<=count;
END PROCESS;
END count_64K;
  CONFIGURATION small_count OF counter IS
      FOR count_255
      END FOR;
  END small_count;
  CONFIGURATION big_count OF counter IS
      FOR count_64K
    END FOR;
  END big_count;
```

在例 2-14 中，一个计数器实体可以实现两个不同构造体的配置。需要注意的是，为达到这个目的，计数器实体中，对装入计数器和构成计数器的数据位宽度不应作具体说明，只将输入和输出数据作为 INTEGER(整型)数据来对待。这样就可以支持多种形式的计数器(如例中的 8 位计数器和 16 位计数器)，以便在宿主机上方便地进行仿真。

下面再举一个构造体内含有元件的配置。仍是设计一个二输入四输出的译码器。译码器的电原理图如图 2-5 所示，它由反相器和三输入与门构成。

【例 2 - 15】反相器和三输入与门电路描述

```
LIBRARY STD;
USE STD.STD_LOGIC.ALL;
USE STD.STD_TTL.ALL;
ENTITY inv IS
    PORT(a: IN T_WLOGIC;
         b: OUT T_WLOGIC);
END inv;
ARCHITECTURE behave OF inv IS
BEGIN
    b<=NOT(a) AFTER 5 ns;
END behave;
CONFIGURATION invcon OF inv IS
    FOR behave
    END FOR;
END invcon;
USE STD.STD_LOGIC.ALL;
USE STD.STD_TTL.ALL;
ENTITY and3 IS
    PORT(a1, a2, a3: IN T_WLOGIC;
         o1: OUT T_WLOGIC);
END and3;
ARCHITECTURE behave OF and3 IS
BEGIN
    o1<=a1 AND a2 AND a3 AFTER 5 ns;
END behave;
CONFIGURATION and3con OF and3 IS
  FOR behave
  END FOR;
END and3con;
```

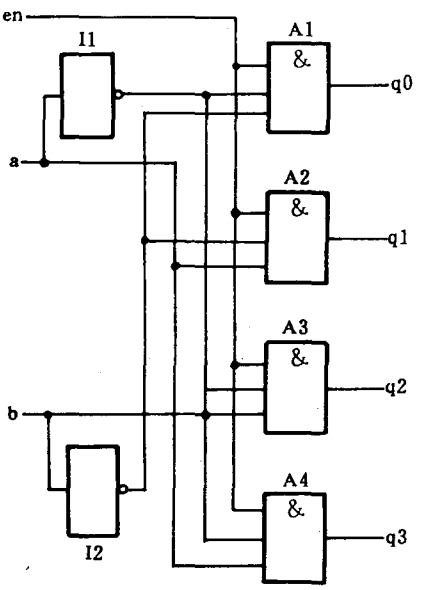


图 2 - 5　二输入四输出译码电路

下面就是用反相器和与非门构成译码器的程序实例，例中使用了 COMPONENT 语句和 PORT MAP(　)语句(这两个语句的含义在后面章节中详述)。

【例 2 - 16】构成译码器的程序

```
LIBRARY STD;
USE STD.STD_LOGIC.ALL;
ENTITY decode IS
    PORT(a, b, en: IN T_WLOGIC;
         q0, q1, q2, q3: OUT T_WLOGIC);
END decode;
ARCHITECTURE structural OF decode IS
    COMPONENT inv
      PORT(a: IN T_WLOGIC;
```

```
            b: OUT T_WLOGIC);
    END COMPONENT;
    COMPONENT and3
      PORT(a1, a2, a3: IN T_WLOGIC;
           o1: OUT T_WLOGIC);
    END COMPONENT;
    SIGNAL nota, notb: T_WLOGIC;
BEGIN
  I1: inv
    PORT MAP(a, nota);
  I2: inv
    PORT MAP(b, notb);
  A1: and3
    PORT MAP(nota, en, notb, q0);
  A2: and3
    PORT MAP(a, en, notb, q1);
  A3: and3
    PORT MAP(nota, en, b, q2);
  A4: and3
    PORT MAP(a, en, b, q3);
END structural;
```

根据上面对反相器和与门的结构描述，以及译码器结构的描述，可以利用不同层次的连接关系实现不同的译码器的配置。

【例 2 - 17】 低层次配置的选择配置

```
CONFIGURATION decode_llcon OF decode IS
    FOR structural1
      FOR I1: inv USE CONFIGURATION WORK.invcon;
    END FOR ;
    FOR I2: inv USE CONFIGURATIOR WORK.invcon;
    END FOR ;
    FOR ALL: and3 USE CONFIGURATION WORK.and3con;
    END FOR;
END decode_llcon;
```

【例 2 - 18】 实体与构造体对应的配置

```
CONFIGURATION decode_eacon OF decode IS
    FOR structural
      FOR I1: inv USE ENTITY WORK.inv(behave);
    END FOR;
    FOR OTHERS: inv USE ENTITY
                WORK.inv(behave);
    END FOR;
    FOR A1: and3 USE ENTITY WORK.and3(behave);
```

```
    END FOR;
    FOR OTHERS: and3 USE ENTITY
                    WORK.and3(behave);
    END FOR;
  END structural;
END decode_eacon;
```

上面例 2 - 14～例 2 - 18 是配置实际应用的几个例子，内部的语句也有多种多样的书写格式，但是不管怎样变，基本格式和实现的功能是完全相同的。

第 3 章

VHDL 语言的数据类型及运算操作符

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，两者是一致的。但是也有某些区别，如 VHDL 语言中可以由用户自己定义数据类型，这一点在其它高级语言中是做不到的。读者在阅读时务请多加注意。

3.1 VHDL 语言的客体及其分类

在 VHDL 语言中凡是可以赋予一个值的对象就称为客体(Object)。客体主要包括以下 3 种：信号、变量、常数(Signal、Variable、Constant)。在电子电路设计中，这 3 类客体通常都具有一定的物理含义。例如，信号对应地代表物理设计中的某一条硬件连接线；常数对应地代表数字电路中的电源和地等。当然，变量对应关系不太直接，通常只代表暂存某些值的载体。3 类客体的含义和说明场合如表 3-1 所示。

表 3-1 VHDL 语言 3 类客体含义和说明场合

客体类别	含　义	说 明 场 合
信　号	信号说明全局量	ARCHITECTURE, PACKAGE, ENTITY
变　量	变量说明局部量	PROCESS, FUNCTION, PROCEDURE
常　数	常数说明全局量	上面两种场合下，均可存在

3.1.1 常数(Constant)

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 DALY: TIME := 100 ns;
CONSTANT FBUS: BIT_VECTOR := "0101";
```

常数一旦被赋值就不能再改变。上面 V_{cc}被赋值为 5.0 V，那么在所有的 VHDL 语言程序中 V_{cc}的值就固定为 5.0 V，它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。例如：

CONSTANT V_{CC}:REAL ：＝"0101"；

这样的常数说明显然是错误的。

3.1.2 变量(Variable)

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：＝表达式；

例如：

VARIABLE x,y: INTEGER;

VARIABLE count: INTEGER RANGE 0 TO 255：＝10;

变量在赋值时不能产生附加延时。例如，tmp1,tmp2,tmp3 都是变量，那么下式产生延时的方式是不合法的：

tmp3：＝tmp1＋tmp2 AFTER 10 ns;

3.1.3 信号(Signal)

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：

SIGNAL　信号名:数据类型　约束条件：＝表达式；

例如：

SIGNAL sys_clk: BIT：＝'0';

SIGNAL ground: BIT：＝'0';

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。

一般来说，在 VHDL 语言中对信号赋值是按仿真时间来进行的。信号值的改变也需按仿真时间的计划表行事。

3.1.4 信号和变量值代入的区别

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。

如图 3－1 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。如图3-2所示，为了进行仿真，需要让代入和处理交替地反复进行。

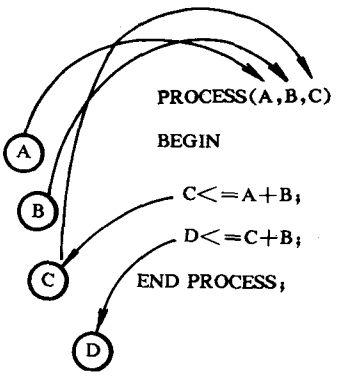


图3-1　信号代入值的取得

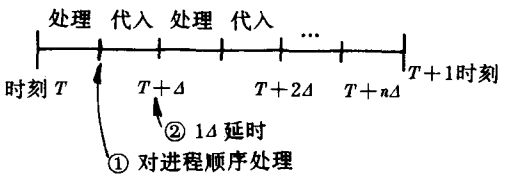


图3-2　进程语句的顺序处理

现在来看一下例3-1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3-1】

```
PROCESS(A,B,C,D)
BEGIN
  D<=A;
  X<=B+D;
  D<=C;
  Y<=B+D;
END PROCESS;

结果
  X<=B+C;
  Y<=B+C;
PROCESS(A,B,C)
VARIABLE D:STD_LOGIC_VECIOR(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 3-1 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 Δ 的延时，但是，在代入时由于不进行处理，因此仿真时认为是时间 0 值延时。因此 D 的最终值应为 C，这样 X 和 Y 的内容都为 B+C。

在例 3-1 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。

与此相反，VHDL 语言的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，必须很好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。

下面对各数据类型作一简要说明。

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－(2^31－1)到(2^31－1)。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。整数的例子如下：

＋136，＋12 456，－457

表 3-2 标准数据类型

数据类型	含　义
整　数	整数 32 位，－2 147 483 647～2 147 483 647
实　数	浮点数，－1.0E＋38～＋1.0E＋38
位	逻辑"0"或"1"
位矢量	位矢量
布尔量	逻辑"假"或逻辑"真"
字　符	ASCII 字符
时　间	时间单位 fs，ps，ns，μs，ms，sec，min，hr
错误等级	NOTE，WARNING，ERROR，FAILURE
自然数，正整数	整数的子集(自然数：大于等于 0 的整数；正整数：大于 0 的整数)
字符串	字符矢量

尽管整数值在电子系统中可能是用一系列二进制位值来表示的，但是整数不能看作是位矢量，也不能按位来进行访问，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。目前在有的CAD厂商所提供的工具中，对此规定已有所突破，允许对有符号和无符号的整型量进行算术逻辑运算。

在电子系统的开发过程中，整数也可以作为对信号总线状态的一种抽象手段，用来准确地表示总线的某一种状态。

2. 实数(Real)

在进行算法研究或者实验时，作为对硬件方案的抽象手段，常常采用实数四则运算。实数的定义值范围为－1.0E＋38～＋1.0E＋38。实数有正负数，书写时一定要有小数点。例如：

－1.0，＋2.5，－1.0E38

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'(将值放在单引号中)表示之。位与整数中的1和0不同，'1'和'0'仅仅表示一个位的两种取值。有时也可以用显式说明之，例如：

BIT '('1')

位数据可以用来描述数字系统中总线的值。位数据不同于布尔数据，当然也可以用转换函数进行转换。

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"

在这里，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。在以后的VHDL程序中将会经常遇到。

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，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、小写字符则认为是不一样的。例如，'B'不同于'b'。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。字符'1'与整数1和实数1.0都是不相同的。当要明确指

出 1 的字符数据时，则可写为

CHARACTER '('1')

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 μs，100 ns，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(Severity Level)

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。

10. 大于等于零的整数(Natural)，正整数(Positive)

这两类数据是整数的子类，Natural 类数据只能取值 0 和 0 以上的正整数；而 Positive 则只能为正整数。

上述 10 种数据类型是 VHDL 语言中标准的数据类型，在编程时可以直接引用。如果用户需使用这 10 种以外的数据类型，则必须进行自定义。但是，大多数的 CAD 厂商已在包集合中对标准数据类型进行了扩展。例如，数组型数据等，这一点请读者注意。

由于 VHDL 语言属于强类型语言，在仿真过程中，首先要检查赋值语句中的类型和区间，任何一个信号和变量的赋值均须落入给定的约束区间中，也就是说要落入有效数值的范围中。约束区间的说明通常跟在数据类型说明的后面。例如：

```
INTEGER RANGE 100 DOWNTO 1
BIT_VECTOR(3 DOWNTO 0)
REAL RANGE 2.0 TO 30.0
```

这里 DOWNTO 表示下降；而 TO 表示上升。

一个 BCD 数的比较器，利用约束区间说明的端口说明语句可以写为：

```
ENTITY bcd_compare IS
PORT (a,b:IN INTEGER RANGE 0 TO 9 := 0;
      c:OUT BOOLEAN);
END bcd_compare;
```

3.2.2 用户定义的数据类型

在 VHDL 语言中，使用户最感兴趣的一个特点是，可以由用户自己来定义数据类型。

由用户定义的数据类型的定义书写格式为：

TYPE 数据类型名 {,数据类型名} 数据类型定义；

在 VHDL 语中还存在不完整的用户定义的数据类型的书写格式：

TYPE 数据类型名 {,数据类型名}；

这种由用户做的数据类型定义是一种利用其它已定义的说明所进行的“假”定义，因此它不能进行逻辑综合。

可由用户定义的数据类型有：

- 枚举(Enumerated)类型；
- 整数(Integer)类型；
- 实数(Real)、浮点数(Floating)类型；
- 数组(Array)类型；
- 存取(Access)类型；
- 文件(File)类型；
- 记录(Recode)类型；
- 时间(Time)类型(物理类型)。

下面对常用的几种用户定义的数据类型作一举例说明。

1. 枚举(Enumerated)类型

在逻辑电路中，所有的数据都是用“1”或“0”来表示的，但是人们在考虑逻辑关系时，只有数字往往是不方便的。在 VHDL 语言中，可以用符号名来代替数字。例如，在表示一周每一天状态的逻辑电路中，可以假设“000”为星期天，“001”为星期一。这对阅读程序是颇不方便的。为此，可以定义一个叫“week”的数据类型。

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

由于上述的定义，凡是用于代表星期二的日子都可以用 tue 来代替，这比用代码“010”表示星期二直观多了，使用时也不易出错。

枚举类型数据的定义格式为：

TYPE 数据类型名 IS (元素，元素，…)；

这类用户定义的数据类型应用相当广泛，例如在包集合“STD_LOGIC”和 “STD_LOGIC_1164”中都有此类数据的定义。如：

```
TYPE STD_LOGIC IS
  ('U','X','0','1','Z','W','L','H','-');
```

2. 整数类型，实数类型(Integer，Real)

整数类型在 VHDL 语言中已存在，这里所说的是用户所定义的整数类型，实际上可以认为是整数的一个子类。例如，在一个数码管上显示数字，其值只能取 0～9 的整数。如果由用户定义一个用于数码显示的数据类型，那么就可以写为：

TYPE digit IS INTEGER RANGE 0 TO 9；

同理实数类型也如此，例如：

TYPE current IS REAL RANGE −1E4 TO 1E4；

据此，可以总结出整数或实数用户定义数据类型的格式为：

TYPE 数据类型名 IS 数据类型定义 约束范围；

3. 数组(Array)

数组是将相同类型的数据集合在一起所形成的一个新的数据类型。它可以是一维的也可以是二维或多维的。

数组定义的书写格式为：

TYPE 数据类型名 IS ARRAY 范围 OF 原数据类型名；

在这里如果范围这一项没有被指定，则使用整数数据类型。例如：

```
TYPE word IS ARRAY (1 TO 8) OF STD_LOGIC;
```

若范围这一项需用整数类型以外的其它数据类型时，则在指定数据范围前应加数据类型名。例如：

```
TYPE word IS ARRAY (INTEGER 1 TO 8) OF STD_LOGIC;
TYPE instruction IS (ADD,SUB,INC,SRL,SRF,LDA,LDB,XFR);
SUBTYPE digit IS INTEGER 0 TO 9;
TYPE insflag IS ARRAY (instruction ADD TO SRF) OF digit;
```

数组在总线定义及 ROM,RAM 等的系统模型中使用。“STD_LOGIC_VECTOR”也属于数组数据类型，它在包集合“STD_LOGIC_1164”中被定义：

```
TYPE STD_LOGIC_VECTOR IS ARRAY
    (NATURAL RANGE< >) OF STD_LOGIC;
```

这里范围由“RANGE< >”指定，这是一个没有范围限制的数组。在这种情况下，范围由信号说明语句等确定。例如：

```
SIGNAL aaa:STD_LOGIC_VECTOR (3 DOWNTO 0);
```

在函数和过程的语句中，若使用无限制范围的数组时，其范围一般由调用者所传递的参数来确定。

多维数组需要用两个以上的范围来描述，而且多维数组不能生成逻辑电路，因此只能用于生成仿真图形及硬件的抽象模型。例如：

```
TYPE memarray IS ARRAY (0 TO 5,7 DOWNTO 0) OF STD_LOGIC;
CONSTANT romdata:memarray := 
            (('0','0','0','0','0','0','0','0'),
             ('0','1','1','1','0','0','0','1'),
             ('0','0','0','0','0','1','0','1'),
             ('1','0','1','0','1','0','1','0'),
             ('1','1','0','1','1','1','1','0'),
             ('1','1','1','1','1','1','1','1'));
SIGNAL data_bit:STD_LOGIC;
          :
data_bit<=romdata(3,7);
```

上述例子是二维的。在三维情况下要用 3 个范围来描述。

在代入初值时，各范围最左边所说明的值为数组的初始位脚标。在上例中(0,7)是起始位，接下去右侧范围向右移一位变为(0,6)，以后顺序为(0,5)，(0,4)直至(0,0)。然后，左侧范围向右移一位变为(1,7)，此后按此规律移动得到最后一位(5,0)。

4. 时间(Time)类型(物理类型)

表示时间的数据类型，在仿真时是必不可少的，其书写格式为：

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

例如：

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

这里基本单位是“fs”，其 1 000 倍是“ps”等等。时间是物理类型的数据，当然对容量、阻抗值等也可以作定义。

5. 记录(Recode)类型

数组是同一类型数据集合起来形成的，而记录则是将不同类型的数据和数据名组织在一起而形成的新客体。记录数据类型的定义格式为：

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

在从记录数据类型中提取元素数据类型时应使用“.”。例如：

```
TYPE bank IS RECORD
      addr0: STD_LOGIC_VECTOR (7 DOWNTO 0);
      addr1: STD_LOGIC_VECTOR (7 DOWNTO 0);
      r0: INTEGER;
      inst: instruction;
END RECORD;
SIGNAL addbus1,addbus2: STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL result: INTEGER;
SIGNAL alu_code: instruction;
SIGNAL r_bank: bank := ("00000000", "00000000", 0,add);
addbus1<=r_bank.addr1;
r_bank.inst<=alu_code;
```

用记录描述 SCSI 总线及通信协议是比较方便的。记录数据类型在生成逻辑电路时应

将它分解开来才行。因此，它比较适用于系统仿真。

3.2.3 用户定义的子类型

用户定义的子类型是用户对已定义的数据类型，作一些范围限制而形成的一种新的数据类型。子类型的名称通常采用用户较易理解的名字。子类型定义的一般格式为：

SUBTYPE 子类型名 IS 数据类型名[范围]；

例如，在“STD_LOGIC_VECTOR”基础上所形成的子类：

```
SUBTYPE iobus IS STD_LOGIC_VECTOR(7 DOWNTO 0);
SUBTYPE digit IS INTEGER RANGE 0 TO 9;
```

子类型可以对原数据类型指定范围而形成，也可以完全和原数据类型范围一致。例如：

```
SUBTYPE abus IS STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL aio: STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL bio: STD_LOGIC_VECTOR (15 DOWNTO 0);
SIGNAL cio: abvs;
    aio<=cio; 正确操作
    bio<=cio; 错误操作
```

除上述外，子类型还常用于存贮器阵列等的数组描述的场合。新构造的数据类型及子类型通常在包集合中定义，再由 USE 语句装载到描述语句中。

3.2.4 数据类型的转换

在 VHDL 语言中，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。这就是所谓类型变换。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 3-3 所示的数据类型变换函数。下面举一个数据类型转换的例子。

表 3-3 类型变换函数

函 数 名	功 能
• STD_LOGIC_1164 包集合	
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 转换成 STD_LOGIC
TO_BIT(A)	由 STD_LOGIC 转换成 BIT
• STD_LOGIC_ARITH 包集合	由 INTEGER,UNSDGNED,SIGNED 转换成
CONV_STD_LOGIC_VECTOR(A,位长)	STD_LOGIC_VECTOR
CONV_INTEGER(A)	由 UNSIGNED,SIGNED 转换成 INTEGER
• STD_LOGIC_UNSIGNED 包集合	
CONV_INTEGER(A)	由 STD_LOGIC_VECTOR 转换成 INTEGER

【例 3-1】 由“STD_LOGIC_VECTOR”变换成“INTEGER”的实例。

```
LIBRARY IEEE;
USE IEEE STD_LOGIC_1164. ALL;
USE IEEE STD_LOGIC_UNSIGNED. ALL;
ENTITY add5 IS
    PORT (num:IN STD_LOGIC_VECTOR (2 DOWNTO 0);
                ⋮
          );
END add5;
ARCHITECTURE rtl OF add5 IS
    SIGNAL in_num:INTEGER RANGE 0 TO 5;
          ⋮
BEGIN
    in_num<=CON_INTEGER (num); (变换式)
          ⋮
END rtl;
```

此外，由“BIT_VECTOR”变换成“STD_LOGIC_VECTOR”也非常方便。代入“STD_LOGIC_VECTOR”的值只能是二进制数，而代入“BIT_VECTOR”的值除二进制数以外，还可能是十六进制及八进制数。不仅如此，“BIT_VECTOR”还可以用“_”来分隔数值位。下面的几个语句表示了“BIT_VECTOR”和“STD_LOGIC_VECTOR”的赋值语句。

```
SIGNAL a: BIT_VECTOR (11 DOWNTO 0);
SIGNAL b: STD_LOGIC_VECTOR (11 DOWNTO 0);
a<=X"A8"; 十六进制值可赋予位矢量
b<=X"A8"; 语法错，十六进制值不能赋予位矢量
b<=TO_STDLOGICVECTOR (X"AF7");
b<=TO_STDLOGICVECTOR (O"5177"); 八进制变换
b<=TO_STDLOGICVECTOR (B"1010_1111_0111");
```

3.2.5 数据类型的限定

在 VHDL 语言中，有时可以用所描述的文字的上下关系来判断某一数据的数据类型。例如：

```
SIGNAL a: STD_LOGIC_VECTOR (7 DOWNTO 0);
a<="01101010";
```

联系上下文关系，可以断定“01101010”不是字符串(String)，也不是位矢量(Bit_Vector)，而是“STD_LOGIC_VECTOR”。但是，有时也有判断不出来的情况。例如：

```
CASE (a & b & c) IS
  WHEN "001"=>Y<="01111111";
  WHEN "010"=>Y<="10111111";
    ⋮
END CASE;
```

在该例中 a&b&c 的数据类型如果不确定就会发生错误。在这种情况下就要对数据进行类型限定(这类似于 C 语言中的强制方式)。数据类型限定的方式是在数据前加上“类型名'”。例如:

```
a<=STD_LOGIC_VECTOR'("01101010");
SUBTYPE STD3BIT IS STD_LOGIC_VECTOR (0 TO 2);
CASE STD3BIT'(a & b & c) IS
    WHEN "000"=>Y<="01111111";
    WHEN "001"=>Y<="10111111";
    ⋮
```

类型限定方式与数据类型变换很相似,这一点应引起读者注意。

3.2.6 IEEE 标准“STD_LOGIC”、“STD_LOGIC_VECTOR”

在上面的数据类型介绍中,曾讲到 VHDL 的标准数据类型“BIT”,它是一个逻辑型的数据类型。这类数据取值只能是“0”和“1”。由于该类型数据不存在不定状态'X',故不便于仿真。另外,由于它也不存在高阻状态,因此也很难用它来描述双向数据总线。为此 IEEE 在 1993 年制订出了新的标准(IEEE STD1164),使得“STD_LOGIC”型数据可以具有如下的 9 种不同的值:

'U'——初始值;
'X'——不定;
'0'——0;
'1'——1;
'Z'——高阻;
'W'——弱信号不定;
'L'——弱信号 0;
'H'——弱信号 1;
'—'——不可能情况。

“STD_LOGIC”和“STD_LOGIC_VECTOR”是 IEEE 新制订的标准化数据类型,也是在 VHDL 语法以外所添加的数据类型,因此将它归属到用户定义的数据类型中。当使用该类型数据时,在程序中必须写出库说明语句和使用包集合的说明语句。

3.3 VHDL 语言的运算操作符

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 3-4 示出了所有操作符的优先次序。

表 3-4 操作符的优先级

优先级顺序	运算操作符类型	操 作 符	功 能
低	逻辑运算符	AND	逻辑与
		OR	逻辑或
		NAND	逻辑与非
		NOR	逻辑或非
		XOR	逻辑异或
	关系运算符	=	等号
		/=	不等号
		<	小于
		>	大于
		<=	小于等于
		>=	大于等于
	加、减、并置运算符	+	加
		−	减
		&	并置
	正、负运算符	+	正
		−	负
	乘法运算符	*	乘
		/	除
		MOD	求模
		REM	取余
高		* *	指数
		ABS	取绝对值
		NOT	取反

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“BIT”等的逻辑型数据、“STD_LOGIC_VECTOR”逻辑型数组及布尔型数据进行逻辑运算。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);

当然也有例外，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;

a<=b XOR c XOR d XOR e;

a<=((b NAND c) NAND d) NAND e;(必须要括号)

a<=(b AND c) OR (d AND e);(必须要括号)

在所有逻辑运算符中 NOT 的优先级最高。

3.3.2 算术运算符

VHDL 语言有 10 种算术运算符，它们分别是：

+——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(只有在左操作数是实数时，右操作数才可以是负整数)。

实际上能够真正综合逻辑电路的算术运算符只有“+”、“-”、“*”。在数据位较长的情况下，在使用算术运算符进行运算，特别是使用乘法运算符“*”时，应特别慎重。因为对于 16 位的乘法运算，综合时逻辑门电路会超过 2000 个门。对于算术运算符“/”、“MOD”、“REM”，分母的操作数为 2 乘方的常数时，逻辑电路综合是可能的。

若对“STD_LOGIC_VECTOR”进行“+”(加)、“-”(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)等枚举类型以及位矢量(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。注意在使用时必须首先说明调用该包集合。当然，此时位矢量还可以和整数进行关系运算。

在关系运算符中小于等于符“<=”和代入符“<=”是相同的，在读 VHDL 语言的语句时，应按照上下文关系来判断此符号到底是关系符还是代入符。

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。图 3-3 就是使用并置运算符的实例。

在图 3-3 中 en 是 b(0)～b(3)的允许输出信号，而 y(0)～y(7)中存在如下关系：

```
y(0)=b(0)        y(1)=b(1)
y(2)=b(2)        y(3)=b(3)
y(4)=a(0)        y(5)=a(1)
y(6)=a(2)        y(7)=a(3)
```

这种逻辑关系用并置运算符就很容易表达出来：

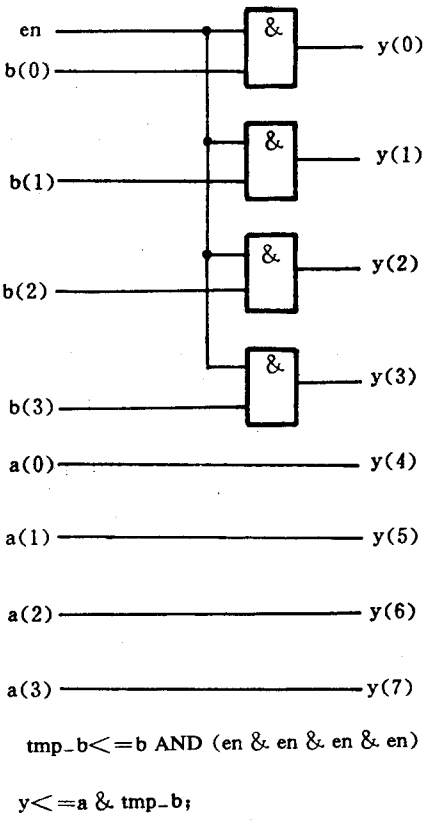


图 3-3 并置运算符使用实例

```
tmp_b<=b AND (en&en&en&en);
y<=a&tmp_b;
```

第一个语句表示 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。第二个语句表示 4 位位矢量 a 和 4 位位矢量 b 再次连接(并置)构成 8 位的位矢量 y 输出。

位的连接也可使用集合体的方法，即将并置符换成逗号就可以了。例如：

```
tmp_b<=(en,en,en,en);
```

但是，这种方法不适用于位矢量之间的连接。如下的描述方法是错误的：

```
a<=(a,tmp_b);
```

集合体也能指定位的脚标，例如上一个语句可表示为：

```
tmp_b<=(3=>en,2=>en,1=>en,0=>en);
```

或

```
tmp_b<=(3 DOWNTO 0=>en);
```

在指定位的脚标时，也可以用“OTHERS”来说明：

```
tmp_b<=(OTHERS=>en);
```

要注意，在集合体中“OTHERS”只能放在最后。假若 b 位矢量的脚标 b(2)的选择信号为“0”，其它位的选择信号均为 en。那么此时表达式可写为：

```
tmp_b<=(2=>'0', OTHERS=>en);
```

第 4 章

VHDL 语言构造体的描述方式

在第 1 章中已经提到，对硬件系统进行描述，可以采用 3 种不同风格的描述方式，即行为描述方式、寄存器传输(或数据流)描述方式和结构化的描述方式。这 3 种描述方式从不同的角度对硬件系统进行行为和功能的描述。在当前情况下，采用后两种描述方式的 VHDL 语言程序可以进行逻辑综合，而采用行为描述的 VHDL 语言程序，大部分只用于系统仿真，少数的也可以进行逻辑综合。本章针对这 3 种不同风格的描述方式作一介绍。

4.1 构造体的行为描述方式

什么样的描述属于行为描述方式，这一点目前还没有确切的定义。所以在不同的书刊中，对相同或相似的某些用 VHDL 语言描述的逻辑电路的程序，有不同的说明。有的说明为行为描述方式，有的说明为寄存器传输描述方式。但是，有一点是明确的，行为描述方式是对系统数学模型的描述，其抽象程度比寄存器传输描述方式和结构化描述方式的更高。在行为描述方式的程序中大量采用算术运算、关系运算、惯性延时、传输延时等难于进行逻辑综合和不能进行逻辑综合的 VHDL 语句。一般来说，采用行为描述方式的 VHDL 语言程序主要用于系统数学模型的仿真或者系统工作原理的仿真。

在 VHDL 语言中存在一些专门用于描述系统行为的语句，它们是 VHDL 语言为什么能在高层次上对系统硬件进行行为描述的原因所在。这些语句与一般的高级语言的语句有较大差别。

4.1.1 代入语句

代入语句是 VHDL 语言中进行行为描述的最基本的语句。例如，

```
a<=b;
```

该语句的功能是 a 得到 b 的值。当该语句有效时，现行信号 b 的值将代入到信号 a，只要 b 的值有一个新的变化，那么该语句将被执行。所以，b 是该代入语句的一个敏感量。

代入语句最普遍的格式为：

```
信号量<=敏感信号量表达式;
```

例如：

```
z<=a NOR (b NAND c);
```

上式有 3 个敏感量 a,b,c。无论哪一个敏感量发生新的变化，该代入语句将被执行。

具有延时时间的代入语句如下所示：

```
a<=b AFTER 5 ns;
```

该语句表示，当 b 发生新的变化 5 ns 以后才被代入到信号 a。

众所周知，一个二输入的与门，由于与门的固有延时，当输入端发生变化以后，与门的输出端的新的输出总要比输入端的变化延时若干时间，例如延时 5 ns。与门的这种输出特性就可以用具有延时时间的代入语句来描述。

【例 4 - 1】

```
ENTITY and2 IS
    PORT (a,b:IN BIT;
          c:OUT BIT);
END and2;
ARCHITECTURE and2_behav OF and2 IS
BEGIN
    c<=a AND b AFTER 5 ns;
END and2_behav;
```

下面再举一个如何用行为描述方式描述四选一电路的例子。四选一电路的逻辑原理图如图 4 - 1 所示。描述四选一电路的 VHDL 语言程序如例 4 - 2 所示。

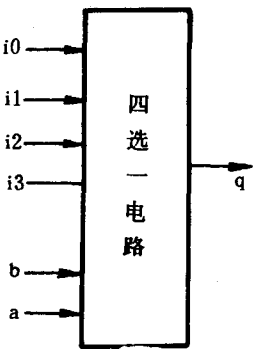


图 4 - 1　四选一电路

【例 4 - 2】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY mux4 IS
    PORT (i0,i1,i2,i3,a,b:IN STD_LOGIC;
                q:OUT STD_LOGIC);
END mux4;
ARCHITECTURE behav OF mux4 IS
    SIGNAL sel:INTEGER;
BEGIN
    WITH sel SELECT
        q<=i0 AFTER 10 ns WHEN 0,
             i1 AFTER 10 ns WHEN 1,
             i2 AFTER 10 ns WHEN 2,
             i3 AFTER 10 ns WHEN 3,
             'X' AFTER 10 ns 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;
```

在四选一电路的构造体中有 6 个输入端口和一个输出端口。a 和 b 是选择信号输入端口。在正常情况下，a 和 b 共有 4 种取值 0～3。a 和 b 的取值将确定 i0～i3 的哪一个输入端信号可以通过四选一电路从输出端 q 输出，其真值表如表 4－1 所示。

表 4－1　四选一电路真值表

b	a	q
0	0	i0
0	1	i1
1	0	i2
1	1	i3

在例 4－2 中，用了两个语句，第一个语句是选择语句；第二个语句是代入语句。这两个语句是条件代入类型的语句。也就是说，只有 WHEN 后面所指定的条件得到满足时，指定的代入值将被代入信号量 sel 或输出量 q。

当第一个语句执行时，将使用选择信号。根据选择信号 sel 的当前值，后跟的 5 种状态下的值 i0～i3、'X'中的一种值将通过输出端口 q 输出。在正常情况下，q 端将选择 i0～i3 的之一输出，在非正常情况下将输出'X' 值。

第二个语句执行时，根据 a 和 b 的具体状态，将 0～4 的值代入信号量 sel。正常情况下，代入 sel 的值为 0～3，非正常情况下代入 4。

上述两个语句都存在敏感信号量。在第二个语句中，a 和 b 是敏感信号量，当 a 和 b 任何一个值有变化，该语句将被执行。第一个语句的信号敏感量为 sel，只要 sel 值有新的变化，第一个语句就会执行。在该构造体中，上述这两个语句是可以并发执行的。有关并发执行的概念，将在后面章节中进一步介绍。

4.1.2　延时语句

在 VHDL 语言中存在两种延时类型：惯性延时和传输延时。这两种延时常用于 VHDL 语言的行为描述方式。

1. 惯性延时

在 VHDL 语言中，惯性延时是缺省的，即在语句中如果不作特别说明，产生的延时一定是惯性延时。这是因为大多数器件在行为仿真时都会呈现这种惯性延时。

在惯性模型中，系统或器件输出信号要发生变化必须有一段时间的延时，这段延时时间常称为系统或器件的惯性或称惯性延时。惯性延时有一个重要的特点，即当一个系统或器件，它的输入信号变化周期小于系统或器件的惯性(或惯性延时)时，其输出将保持不变。如图 4－2 所示，有一个门电路，其惯性延时时间为 20 ns，当该门电路的输入端 a 输入一个 10 ns 的脉冲信号时，其输出端 b 的输出仍维持低电平，没有发生变化。对于惯性时间等于 20 ns 的门电路，为使其实现正常的功能，输入信号的变化周期一定要大于 20 ns。

几乎所有器件都存在惯性延时，因此，硬件电路的设计人员为了逼真地仿真硬件电路的实际工作情况，在代入语句中总要加上惯性延时时间的说明。例如：

```
b<=a AFTER 10 ns;
```

惯性延时说明只在行为仿真时有意义，逻辑综合时将被忽略，或者在逻辑综合前必须去掉延时说明。

2. 传输延时

在 VHDL 语言中，传输延时不是缺省的，必须在语句中明确说明。传输延时常用于描述总线延时、连接线的延时及 ASIC 芯片中的路径延时。

如果图 4 - 2 的门电路的惯性延时用传输延时来替代，那么就可以得到如图 4 - 3 的波形结果。从图 4 - 3 的波形可以看到，同样门电路，当有 10 ns 的脉冲波形输入时，经 20 ns 传输延时以后，在输出端就产生 10 ns 的脉冲波形。也就是说，输出端的信号除延时规定时间外，将完全复现输入端的输入波形，而不管输入波形的形状和宽窄如何。

具有传输延时的代入语句如下所示：

```
b<=TRANSPORT a AFTER 20 ns;
```

语句中“TRANSPORT”是专门用于说明传输延时的前置词。

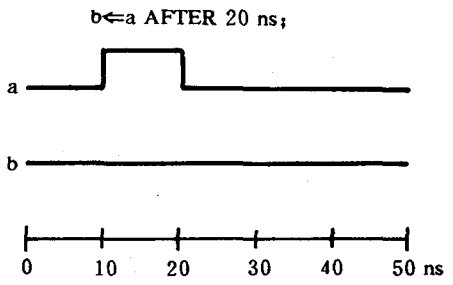


图 4 - 2　惯性延时示例

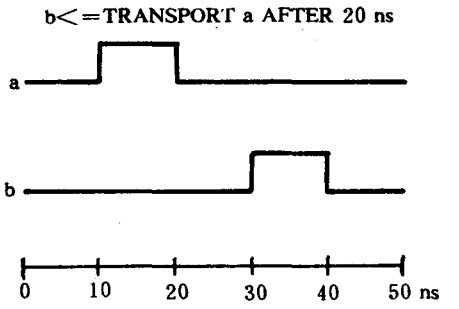


图 4 - 3　传输延时示例

4.1.3　多驱动器描述语句

在 VHDL 语言中，创建一个驱动器可以由一条信号代入语句来实现。当有多个信号并行输出时，在构造体内部必须利用代入语句，对每个信号创建一个驱动器。这样在构造体内部就会有多个代入语句。在设计逻辑电路时，有时会碰到这样情况，多个驱动器的输出将连接到同一条信号线上。考虑这种情况，多驱动器的构造体应按如下方式描述：

```
ARCHITECTURE sample OF sample IS
BEGIN
    a<=b AFTER 5 ns;
    a<=d AFTER 5 ns;
END sample;
```

在上述的 sample 的结构中，信号 a 由两个驱动源 b 和 d 驱动。每一个并发的信号代入语句都将创建一个驱动器，它们的输出共同驱动信号 a。第一条语句创建一个驱动器，其输出值为 b，经 5 ns 延时驱动信号 a；第二个语句创建一个驱动器，其输出值为 d，经 5 ns 延时驱动信号 a。在这种情况下，信号 a 的值将取决于两个驱动器的输出 b 和 d。那么信号 a 到底应该取何值。这一点在标准的数据类型中是没有定义的。为了解决多个驱动器同时驱动一个信号的信号行为描述，在包集合 STD_LOGIC_1164 中专门定义了一种描述判决函数的数据类型，称判决函数子类型。所谓判决函数就是在多个驱动器同时驱动一个信号

时，定义输出哪一个值的函数。例 4 - 3 是包集合 STD_LOGIC_1164 中关于判决函数描述的部分源程序。

【例 4 - 3】

```
PACKAGE STD_LOGIC_1164 IS
    ⋮
TYPE STD_ULOGIC IS ('U','X','0','1','Z','W','L','H','-');
TYPE STD_ULOGIC_VECTOR IS ARRAY (NATURAL RANGE<>)
                                    OF STD_ULOGIC;  - - - 判决函数说明
FUNCTION resolved(s:STD_ULOGIC_VECTOR) RETURN STD_ULOGIC;
SUBTYPE STD_LOGIC IS resolved STD_ULOGIC; - - -子类型数据说明
TYPE STD_LOGIC_VECTOR IS ARRAY(NATURAL RANGE<>) OF STD_LOGIC;
    ⋮
END STD_LOGIC_1164;
PACKAGE BODY STD_LOGIC_1164 IS
    ⋮
  CONSTANT resolution_table:STDLOGIC_TABLE := (
--| U   X   0   1   Z   W   L   H   -        |  |
  ('U','U','U','U','U','U', 'U','U','U'),- -  |U |
  ('U','X','X','X','X','X', 'X','X','X'),- -  |X |
  ('U','X','0','X','0','0', '0','0','X'),- -  |0 |
  ('U','X','X','1','1','1', '1','1','X'),- -  |1 |
  ('U','X','0','1','Z','W', 'L','H','X'),- -  |Z |
  ('U','X','0','1','W','W', 'W','W','X'),- -  |W |
  ('U','X','0','1','L','W', 'L','W','X'),- -  |L |
  ('U','X','0','1','H','W', 'W','H','X'),- -  |H |
  ('U','X','X','X','X','X', 'X','X','X'),- -  |- |
);
FUNCTION resolved (s:STD_ULOGIC_VECTOR) RETURN STD_ULOGIC IS - -判决函数本体
  VARIABLE result:STD_ULOGIC := 'Z'; 高阻状态
BEGIN
  IF (s'LENGTH=1) THEN RETURN s(s'LOW);
  ELSE
    FOR i IN s'RANGE LOOP
        result := resolution_table(result,s(i));
    END LOOP;
  END IF;
  RETURN result;
END resolved;
    ⋮
END STD_LOGIC_1164;
```

在例 4 - 3 中定义了判决函数，当系统要确定多驱动器输出的状态时，可调用该函数，

例如：

FUNCTION resolved (s:STD_ULOGIC_VECTOR) RETURN STD_ULOGIC;

语句中 s 是位矢量，其位长度就是多驱动器输出的信号数。

```
⋮
若 s 取值为('0','1','X')
s1 := resolved(s); --s1 为 STD_ULOGIC
⋮
```

上述两条语句表示，有 3 个驱动器，其输出值分别为“0”,“1”和“X”。s1 是这 3 个驱动器输出共同驱动的信号。那么在这种情况下，s1 应该处于什么状态呢。调用判决函数 resolved(s)得到的返回值应为“X”，那么此时 s1 的状态应为“X”。若 s=('0','Z','Z')，调用 resolved(s)可得到返回值“0”，那么此时 s1 的状态应为“0”。这样，使用判决函数就可以正确地描述多驱动器输出时的信号行为。

4.1.4 GENERIC 语句

GENERIC 语句常用于不同层次之间的信息传递。例如，在数据类型说明上用于位矢量长度、数组的位长以及器件的延时时间等参数的传递。该语句所涉及的数据除整数类型以外，如涉及其它类型的数据则不能进行逻辑综合。因此，该语句主要用于行为描述方式。

使用 GENERIC 语句易于使器件模块化和通用化。例如，要描述二输入与门的行为。二输入与门的逻辑关系是明确的，但是由于在集成时材料不同和工艺不同，不同类型的二输入与非门的上升沿、下降沿等参数是不一致的。为简化设计和供其它设计人员方便地调用，需要开发一个通用的二输入与门的程序模块。在该模块中某些参数是待定的，在仿真或逻辑综合时，只要用 GENERIC 语句将待定参数初始化后，即可实现各种类型二输入与门的仿真或逻辑综合。在例 4－4 中就是采用 GENERIC 语句的一个实例。

【例 4－4】

```
ENTITY and2 IS
    GENERIC (rise,fall:TIME);
    PORT (a,b: IN BIT;
          c: OUT BIT);
END and2;
ARCHITECTURE behav OF and2 IS
    SIGNAL internal:BIT;
BEGIN
    internal<=a AND b;
    c<=internal AFTER (rise)  WHEN internal='1' ELSE
        internal AFTER (fall);
END behav;
```

例 4－4 是一个通用的二输入与门的实体。如果现在要构成一个如图 4－4 所示的电路，那么尽管图 4－4 中的各二输入与非门的上升和下降的时间不同，如使用 GENERIC 和 GENERIC MAP 语句，仍能调用通用的二输入与非门模块，以简化电路的设计。利用例 4－4 通用二输入与门模块，构成图 4－4 逻辑电路的 VHDL 语言程序如例 4－5 所示。

【例 4 - 5】

```
ENTITY sample IS
GENERIC (rise, fall:TIME);
PORT (ina,inb,inc,ind:IN BIT;
                    q:OUT BIT);
END sample;

ARCHITECTURE behav OF sample IS
    COMPONENT and2
    GENERIC (rise,fall:TIME);
    PORT (a,b:IN BIT;
          c:OUT BIT);
    END COMPONENT;

SIGNAL U0_C,U1_C:BIT
BEGIN
    U0:and2 GENERIC MAP (5 ns, 5 ns)
          PORT MAP (ina,inb,U0_C);
    U1:and2 GENERIC MAP (8 ns, 10 ns)
          PORT MAP (inc,ind,U1_C);
    U2:and2 GENERIC MAP (9 ns, 11 ns)
          PORT MAP (U0_C,U1_C,q);
END behav;
```

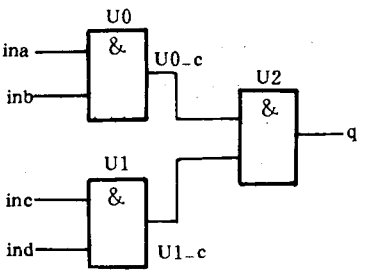


图 4 - 4　三个二输入与门构成的电路

由例 4 - 5 可以看到，利用 GENERIC MAP 语句的功能，在使用同一个 and2 实体的情况下，可使得 U0,U1,U2三个与门的上升时间和下降时间具有不同的值。U0的上升时间为5 ns，U1的上升时间为8 ns，U2的上升时间为9 ns；而 U0的下降时间为 5 ns，U1的下降时间为10 ns，U2的下降时间为11 ns。如此灵活地改变参数就可以完全满足实际设计中的要求。

还有其它许多语句也用于构造体的行为描述方式，如 GUARDED BLOCK 等。VHDL 语言之所以优于目前已开发的各种硬件描述语言，其主要的一个优点是，它具有丰富的语句和语法，能在高层次上对系统的行为进行描述和仿真。

4.2　构造体的寄存器传输(RTL)描述方式

如前一节所述，采用行为描述方式的 VHDL 语言程序，在一般情况下只能用于行为层次的仿真，而不能进行逻辑综合。对于用行为描述方式的 VHDL 语言程序只有改写为 RTL 描述方式才能进行逻辑综合，也就是说 RTL 描述方式才是真正可以进行逻辑综合的描述方式。在某些书刊中也把 RTL 描述方式称为数据流描述方式。

4.2.1 RTL 描述方式的特点

RTL 描述方式，是一种明确规定寄存器描述的方法。由于受逻辑综合的限制，在采用 RTL 描述方式时，所使用的 VHDL 语言的语句有一定限制，其限制情况如附录 A 所示。在 RTL 描述方式中要么采用寄存器硬件的一一对应的直接描述，要么采用寄存器之间的功能描述。例如，上节中的四选一电路，如采用 RTL 描述方式，其 VHDL 语言描述程序如例4 - 6所示。

【例 4 - 6】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGEND.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
    y<=input(0) WHEN sel="00" ELSE
       input(1) WHEN sel="01" ELSE
       input(2) WHEN sel="10" ELSE
       input(3);
END rtl;
```

例 4 - 6 其实是对四选一电路的功能进行描述而得到的 RTL 描述实体。

下面再举一个二选一电路的例子，二选一电路的电原理图如图 4 - 5 所示。下面用两种不同的方法来描述该电路。例 4 - 7 是用功能描述的 RTL 描述方式，例 4 - 8 是采用硬件一一对应的 RTL 描述方式。

【例 4 - 7】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGEND.ALL;
ENTITY mux2 IS
    PORT (input:IN STD_LOGIC_VECTOR (1 DOWNTO 0);
          sel:IN STD_LOGIC;
          y:OUT STD_LOGIC);
END mux2;

ARCHITECTURE rtl OF mux2 IS
BEGIN
    y<=input(0) WHEN sel='1' ELSE
```

```
        input(1);
    END rtl;
```

【例 4 - 8】

```
    LIBRARY IEEE;
    USE IEEE.STD_LOGIC_1164.ALL;
    USE IEEE.STD_LOGIC_UNSIGEND.ALL
      ENTITY mux2 IS
          PORT (in0,in1,sel:IN STD_LOGIC;
                        y:OUT STD_LOGIC);
      END mux2;

      ARCHITECTURE rtl OF mux2 IS
          SIGNAL tmp1,tmp2,tmp3:STD_LOGIC;
      BEGIN
          tmp1<=in0 AND sel;
          tmp2<=in1 AND (NOT sel);
          tmp3<=tmp1 OR tmp2;
          y<=tmp3;
      END rtl;
```

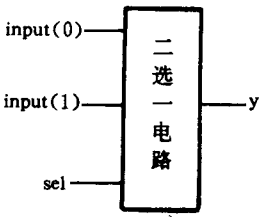


图 4 - 5　二选一电路原理图

对于例 4 - 7，是将二选一电路看成一个黑框，编程者无需了解二选一电路内部的细节，只要知道外部特性和功能就可以进行正确的描述。而对例 4 - 8，编程者就必须了解二选一电路是怎样构成的，内部采用了哪些门电路。只有了解了这样一些细节，才能用 VHDL 语言进行正确的描述。所以从编程效率及编程难度上来看，应该选择例 4 - 7 的这种编程方法，来编写 RTL 描述方式的程序。

随着 CAD 技术的发展，人们也正在探讨如何对用行为描述方式的程序进行逻辑综合，如能做到这一点，这将会大大提高 CAD 技术的水平。

4.2.2　使用 RTL 描述方式应注意的几个问题

1. “X”状态的传递

在目前的 RTL 设计中要对所设计的程序进行仿真检验，而且在逻辑电路综合以后还有必要对综合的结果进行仿真。之所以要进行二次仿真，这是因为在仿真过程中存在“X”传递的影响。它可以使得 RTL 仿真和门级电路仿真产生不一致的结果。

所谓“X”状态的传递，实质上是不确定信号状态的传递，它将使逻辑电路产生不确定的结果，不确定“状态”在 RTL 仿真时是允许出现的，但是在逻辑综合后的门级电路仿真中是不允许出现的。

例 4 - 9 是一个二值输入的器件的 RTL 描述。当 sel=1 时，其输出 y 为“0”；而当 sel=0 时，其输出 y 为“1”。如果在这里 sel 的状态为“X”，那么，因为“X”不是“1”，故程序执行 ELSE 项，使输出为“1”。这样“X”状态就从前一段传递到后一段，在仿真时认为电路是正确的。现在将例 4 - 9 的描述顺序改变一下，如例 4 - 10 所示。

【例 4 - 9】

```
PROCESS (sel)
BEGIN
  IF (sel='1') THEN
    y<='0';
  ELSE
    y<='1';
  END IF;
END PROCESS;
```

【例 4 - 10】

```
PROCESS (sel)
BEGIN
  IF (sel='0') THEN
    y<='1';
  ELSE
    y<='0';
  END IF;
END PROCESS;
```

同样当 sel='X'时，输出的 y 值将变为“0”。为了防止这种不合理的结果，在例 4 - 9 增加一项 y<='X'输出项，如下例所示。

```
PROCESS (sel)
BEGIN
  IF (sel='1') THEN
    y<='0';
  ELSIF (sel='0') THEN
    y<='1';
  ELSE
    y<='X';
  END IF;
END PROCESS;
```

在上例中 ELSE 项以前，将 sel 所有的可能取值都作了明确的约束，当 sel='X'时，其输出 y 也将变为“X”，就不会出现不合理的结果。在逻辑综合时，ELSE 项是被忽略的，这样 RTL 仿真结果就和逻辑综合的仿真结果是一样的。

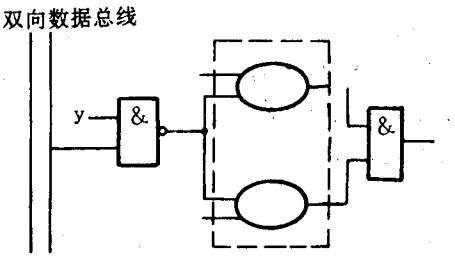


图 4 - 6　双向总线与逻辑电路的连接

在使用双向总线(如数据总线)时，其信号取值总是会出现高阻状态“Z”。当双向总线的信号去驱动逻辑电路时，就有可能出现“X”状态的传递。为了保证逻辑电路的正常工作，高阻状态“Z”是应该禁止的，如图 4 - 6 所示。在图 4 - 6 中，用与非门来禁止，不使

“Z”状态变为“X”状态，而被传递。禁止信号 EN 保证在双向总线出现“Z”状态时，其取值为“0”，正常时取值为“1”。

2. 寄存器 RTL 描述的限制

由 RTL 描述所生成的逻辑电路中，一般来说寄存器的个数和位置与 RTL 描述的情况是一致的。但是，寄存器 RTL 描述不是任意的，而是有一定限制的。

1）禁止在一个进程中存在两个寄存器描述

RTL 描述规定，在一个进程中只能描述一个寄存器。像例 4－11 那样对两个寄存器进行描述是不允许的。

【例 4－11】

```
PROCESS (clk1,clk2)
BEGIN
    IF (clk1'EVENT AND clk1='1') THEN
            y<=a;
    END IF;
    IF (clk2'EVENT AND clk2='1') THEN
            z<=b;
    END IF;
END PROCESS;
```

2）禁止使用 IF 语句中的 ELSE 项

在用 IF 语句描述寄存器功能时，禁止采用 ELSE 项。例 4－12 这样的描述是应该禁止使用的。

【例 4－12】

```
PROCESS (clk)
BEGIN
    IF (clk'EVENT AND clk='1') THEN
        y<=a;
    ELSE              --禁止使用
        y<=b;
    END IF;
END PROCESS;
```

3）寄存器描述中必须代入信号值

在寄存器描述中，必须将值代入信号，如例 4－13 所示那样。

【例 4－13】

```
PROCESS (clk)
VARIABLE tmp:STD_LOGIC;
BEGIN
    IF (clk'EVENT AND clk='1') THEN
        tmp:=a;
    END IF;
        y<=tmp;
END IF;
```

3. 关联性强的信号应放在一个进程中

在设计“与”及“或”这样部件时，如果在原理图上是并行放置的，那么通常进程和部件是一一对应的。但是，在许多较复杂的电路中，它有多个输入和输出，有些信号互相的关联度很高，而有些信号互相的关联度就很低，在这种情况下，为了在逻辑综合以后，使其电路的面积和速度指标更高，通常将关联度高的信号放在一个进程中，将电路分成几个进程来描述。如图 4 - 7 这样的逻辑电路，可以用一个进程描述如例 4 - 14 所示，也可以采用多进程描述如例 4 - 15 所示。

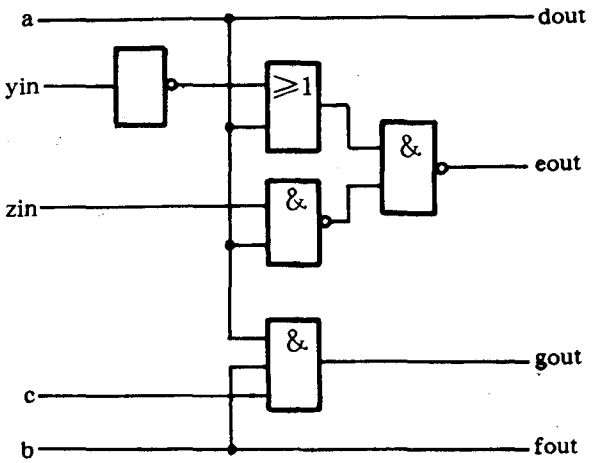


图 4 - 7　多进程描述的电路

【例 4 - 14】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY ex1 IS
    PORT (a,b,c,zin,yin:IN STD_LOGIC;
          dout,eout,fout,gout:OUT STD_LOGIC);
END ex1;
ARCHITECTURE rtl OF ex1 IS
BEGIN
    PROCESS (a,b,c,zin,yin)
        BEGIN
            IF (a='1' AND b='0') THEN
                dout<='1';
                eout<=zin;
                fout<='0';
                gout<='0';
            ELSIF (a='0' AND b='0') THEN
                    dout<='0';
                    eout<=yin;
```

```
            fout<='0';
            gout<='0';
        ELSIF (a='0' AND b='1') THEN
            dout<='0';
            eout<=yin;
            fout<='1';
            gout<='0';
        ELSIF (c='1' ) THEN
            dout<='1';
            eout<=zin;
            fout<='1';
            gout<='1';
        ELSE
            dout<='1';
            eout<=zin;
            fout<='1';
            gout<='0';
        END IF;
    END PROCESS;
END rtl;
```

【例 4 - 15】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY ex2 IS
    PORT (a,b,c,zin,yin:IN STD_LOGIC;
          dout,eout,fout,gout:OUT STD_LOGIC);
END ex2;
ARCHITECTURE rtl OF ex2 IS
BEGIN
    PROCESS (a,zin,yin)
      BEGIN
        IF (a='1') THEN
            dout<='1';
            eout<=zin;
        ELSE
            dout<='0';
            eout<=yin;
        END IF;
      END PROCESS;

  PROCESS (b)
    BEGIN
        IF (b='1') THEN
```

```
            fout<='1';
        ELSE
            fout<='0';
        END IF;
    END PROCESS;

    PROCESS (a,b,c)
      BEGIN
        IF (a='1' AND b='1' AND c='1') THEN
            gout<='1';
        ELSE
            gout<='0';
        END IF;
  END PROCESS;
END rtl;
```

由上面几个例子可以看出，在用 RTL 描述时，要想使这些描述都能正确地进行逻辑综合，并使综合结果具有较佳的性能，就必须要注意 RTL 描述的一些具体规定和相应的技巧。

4.3 构造体的结构描述方式

所谓构造体的结构描述方式，就是在多层次的设计中，高层次的设计模块调用低层次的设计模块，或者直接用门电路设计单元来构成一个复杂的逻辑电路的描述方法。结构描述方式最能提高设计效率，它可以将已有的设计成果，方便地用到新的设计中去。例如，某一个逻辑电路是由 AND 门、OR 门和 XOR 门构成的，而 AND 门、OR 门和 XOR 门的逻辑电路都已有现成的设计单元。那么，用这些现成的设计单元(AND 的 ENTITY、OR 的 ENTITY 和 XOR 的 ENTITY)经适当连接就可以构成新的设计电路的 ENTITY。这样的描述，其结构非常清晰，且能做到与电原理图中所画的器件一一对应。当然，如要用结构描述方式，则要求设计人员有较多的硬件设计知识。

4.3.1 构造体结构描述的基本框架

一个二选一电路用结构化描述方式描述的构造体如例 4 - 16 所示。二选一电路的逻辑电路如图 4 - 8 所示。

【例 4 - 16】

```
ENTITY mux2 IS
PORT (d0,d1,sel:IN BIT;
          q:OUT BIT);
END mux2;

ARCHITECTURE struct OF mux2 IS
COMPONENT and2
```

```
    PORT (a,b:IN BIT;
          c:OUT BIT);
END COMPONENT;
COMPONENT or2
    PORT (a,b:IN BIT;
          c:OUT BIT);
END COMPONENT;
COMPONENT inv
    PORT (a:IN BIT;
          c:OUT BIT);
END COMPONENT;
SIGNAL aa,ab,nsel:BIT;
BEGIN
    u1:inv PORT MAP (sel,nsel);
    u2:and2 PORT MAP (nsel,d1,ab);
    u3:and2 PORT MAP (d0,sel,aa);
    u4:or2 PORT MAP (aa,ab,q);
END struct;
```

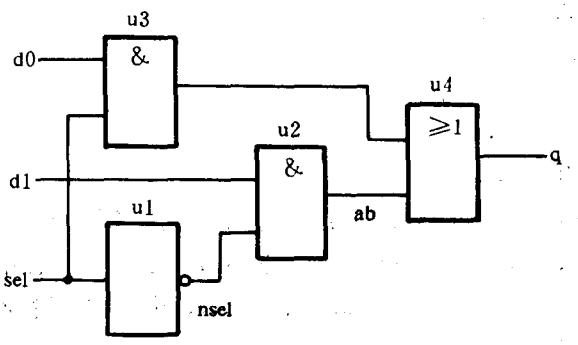


图 4-8 二选一逻辑电路

从例 4-16 中可以看出，在二选一电路的构造体中用 COMPONENT 语句指明了在该电路中所使用的已生成的模块(在这里是 AND,OR,NOT 门电路)，供本构造体调用。用 PORT MAP( )语句将生成模块的端口与所设计的各模块(在这里为 u1,u2,u3,u4)的端口联系起来，并定义相应的信号，以表示所设计的各模块的连接关系。

这种结构描述方式，较方便地能进行多层次的结构设计。例如，某系统由若干块插件板组成，每个插件块又由若干块专用的 ASIC 电路组成，各专用的 ASIC 电路又由若干个已生成的基本单元电路组成。这样 3 个层次构成的系统可以用 3 个层次的结构来描述。

1. ASIC 级结构描述

假设该系统中的 ASIC 电路的基本结构是由与、或和非门 3 种基本逻辑电路构成的。那么 ASIC 级的结构描述如例 4-17 所示。

【例 4 - 17】

```
asic1: BLOCK
    PORT (…);
    COMPONENT and2
        ⋮
    END COMPONENT;
    COMPONENT or2
        ⋮
    END COMPONENT;
    COMPONENT inv
        ⋮
    END COMPONENT;
    SIGNAL …;
    FOR u1:and2 USE ENTITY WORK.and2;
    FOR u2:or2 USE ENTITY WORK.or2;
BEGIN
    u1:and2 PORT MAP (…);
    u2:or2 PORT MAP (…);
        ⋮
END BLOCK asic1;
        ⋮
asicn:BLOCK
    ⋮
END BLOCK asicn;
```

在例 4 - 17 中对 n 种 ASIC 芯片的结构作了描述，不同的 ASIC 芯片是由不同个数和连接关系的与门、或门和非门构成的。对这些 ASIC 芯片进行逻辑综合就可以得到现成的ASIC芯片。如在其它的逻辑电路中要使用这些ASIC芯片，即可在库中调用。

2. 插件板级结构描述

每种插件板是由若干块不同的 ASIC 芯片构成的，如要描述插件板的逻辑电路，则仍可采用结构描述方式来进行描述，如例4 - 18 所示。

【例 4 - 18】

插件板1

```
ENTITY printed_board1 IS
    PORT (…);
END printed_board1;
ARCHITECTURE board1 OF printed_board1 IS
    signal …;
BEGIN
    asic1:BLOCK
        ⋮
    END BLOCK asic1;
```

```
        ⋮
    asic5:BLOCK
        ⋮
    END BLOCK asic5;
        ⋮
END board1;
⋮                ⋮
ENTITY printed_boardm IS
    PORT (…);
END printed_boardm;
ARCHITECTURE boardm OF printed_boardm IS
        ⋮
END boardm;
```

插件板 *m*

在例 4 - 18 中描述了 *m* 块插件板的每一块是由哪些 ASIC 芯片组成的，且其连接关系是什么。这样就得到了插件板级的逻辑电路的结构描述。

3. 系统级的结构描述

若一个系统是由 *m* 块插件板连接而成的，通过插件板级描述，认为它们是可以供系统设计逻辑电路时能任意调用的已设计好的模块。此时，系统级的结构描述实例如例 4 - 19 所示。

【例 4 - 19】

```
ENTITY system IS
    PORT (…);
        ⋮
END system;
ARCHITECTURE struct OF system IS
    COMPONENT printed_board1
    PORT (…);
    END COMPONENT;
        ⋮
    COMPONENT printed_boardm
    PORT (…);
    END COMPONENT;
    FOR B_1:printed_board1 USE ENTITY WORK.printed_board1;
            ⋮
    FOR B_m:printed_boardm USE ENTITY WORK.printed_boardm;
    SIGNAL …;
BEGIN
    B_1:printed_board1    --插件板1
        PORT MAP (…);
            ⋮
    B_m:printed_boardm    --插件板 m
        PORT MAP (…);
END struct;
```

由此三级结构描述，可以清晰地看到系统的结构和它们的连接关系，而且这些现成的模块可为以后的设计带来很大的方便。

4.3.2 COMPONENT 语句

在构造体的结构描述中，COMPONENT 语句是基本的描述语句。该语句指定了本构造体中所调用的是哪一个现成的逻辑描述模块。例如在例 4－16 的二选一电路的结构描述程序中，使用了 3 个 COMPONENT 语句，分别引用了现成的 3 种门电路的描述。这 3 种门电路在库中已生成，在任何设计中如用到这 3 种门电路，只要用 COMPONENT 语句调用就行了，而无需在构造体中再对这些门电路进行定义和描述。COMPONENT 语句的基本书写格式如下：

```
COMPONENT 元件名
    GENERIC 说明；  --参数说明
    PORT 说明；     --端口说明
END COMPONENT；
```

COMPONENT 语句可以在 ARCHITECTURE，PACKAGE 及 BLOCK 的说明部分中使用。在 COMPONENT 和 END COMPONENT 之间可以有参数传递的说明 GENERIC 语句和端口说明的 PORT 语句。

GENERIC 通常用于该元件的可变参数的代入或赋值，而 PORT 则说明该元件的输入输出端口的信号规定。

4.3.3 COMPONENT_INSTANT 语句

COMPONENT_INSTANT 语句是结构化描述中不可缺少的一个基本语句。该语句将现成元件的端口信号映射成高层次设计电路中的信号。例如在例 4－16 中将二输入与门的 a,b,c 三个端口信号映射成图 4－8 中与门 u2的 nsel，d1和 ab 三条连接线的信号。图 4－8 中的各门电路之间的连接关系就是通过该语句信号映射关系来实现连接的。COMPONENT_INSTANT 语句的书写格式为：

```
标号名:元件名  PORT MAP (信号,…);
```

例如：

```
u2:and2 PORT MAP (nse1,d1,ab);
```

标号名加在元件名的前面，在该构造体的说明中该标号名一定是唯一的。下一层元件的端口信号与实际连接的信号用 PORT MAP 的映射关系联系起来。映射方法有两种：一种是位置映射；一种是名称映射。

1. 位置映射方法

所谓位置映射方法就是在下一层中元件端口说明中的信号书写顺序位置和 PORT MAP( )中指定的实际信号书写顺序位置一一对应。例如，在二输入与门中端口的输入输出定义为：

```
PORT (a,b:IN BIT;
      c:OUT BIT);
```

在设计的引用中与门 u2的信号对应关系描述为：

u2:and2 PORT MAP (nsel,d1,ab);

也就是说在图 4 - 8 中，u2 的 nse1 对应 a，d1对应 b，ab 对应 c。

2. 名称映射方法

所谓名称映射就是将已经存于库中的现成模块的各端口名称，赋予设计中模块的信号名。例如：

u2:and2 PORT MAP (a=>nsel,b=>d1,c=>ab);

在输出信号没有连接的情况下，对应端口的描述可以省略。

第 5 章

VHDL 语言的主要描述语句

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)描序语句和并发(Concurrent)描述语句。例如，进程语句(Process Statement)是一个并发语句。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，是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。

5.1 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；
- 断言语句；
- 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；
- 过程调用语句；
- NULL 语句。

空(NULL)语句表示只占位置的一种空处理操作，但是它可以用来为所对应信号赋一个空值，表示该驱动器被关闭，该语句在下面不作介绍，其余语句将通过具体实例，一一作详细介绍。

5.1.1 WAIT 语句

进程在仿真运行中总是处于下述两种状态之一：执行或挂起。进程状态的变化受等待语句的控制，当进程执行到等待语句时，就将被挂起，并设置好再次执行的条件。WAIT 语句可以设置 4 种不同的条件：无限等待、时间到、条件满足以及敏感信号量变化。这几类条件可以混用，其书写格式为：

```
WAIT          ——无限等待
WAIT ON       ——敏感信号量变化
WAIT UNTIL    ——条件满足
WAIT FOR      ——时间到
```

1. WAIT ON

WAIT ON 语句的完整书写格式为：

```
WAIT ON 信号[,信号];
```

WAIT ON 语句后面跟着的是一个或多个信号量，例如：

```
WAIT ON a,b;
```

该语句表明，它等待信号量 a 或 b 发生变化。a 或者 b 中只要有一个信号量发生变化，进程将结束挂起状态，而继续执行 WAIT ON 语句后继的语句。WAIT ON 可以再次启动进程的执行，其条件是指定的信号量必须有一个新的变化。从这一点来看，与进程指定的敏感信号量有新的变化时，也会启动进程的情况相类似，如例 5－1 所示。

【例 5－1】

```
PROCESS(a, b)                PROCESS
  BEGIN                        BEGIN
    y<=a AND b;                  y<=a AND b;
END PROCESS;                     WAIT ON a,b;
                             END PROCESS;
```

例 5－1 所示的两个进程的描述是完全等价的，只是 WAIT ON 和 PROCESS 中所使用的敏感信号量的书写方法有区别。在使用 WAIT ON 语句的进程中，敏感信号量应写在进程中的 WAIT ON 语句后面；而在不使用 WAIT ON 语句的进程中，敏感信号量只应在进程开头的 PROCESS 后跟的括号中说明。需要注意的是，如果 PROCESS 语句已有敏感信号量说明，那么在进程中再不能使用 WAIT ON 语句。例如，例 5－2 的描述是非法的。

【例 5－2】

```
PROCESS(a, b)
  BEGIN
    y<=a AND b;
    WAIT ON a, b; --错误语句
END PROCESS;
```

2. WAIT UNTIL

WAIT UNTIL 语句的完整书写格式为：

```
WAIT UNTIL 表达式;
```

WAIT UNTIL 语句后面跟的是布尔表达式，当进程执行到该语句时将被挂起，直到表达

式返回一个“真”值，进程才被再次启动。

该语句在表达式中将建立一个隐式的敏感信号量表，当表中的任何一个信号量发生变化时，就立即对表达式进行一次评估。如果评估结果使表达式返回一个“真”值，则进程脱离等待状态，继续执行下一个语句。例如：

```
WAIT UNTIL((x * 10)<100);
```

在这个例子中，当信号量 x 的值大于或等于 10 时，进程执行到该语句将被挂起；当 x 的值小于 10 时进程再次被启动，继续执行 WAIT 语句的后继语句。

3. WAIT FOR 语句

WAIT FOR 语句的完整书写格式为：

```
WAIT FOR 时间表达式;
```

WAIT FOR 语句后面跟的是时间表达式，当进程执行到该语句时将被挂起，直到指定的等待时间到时，进程再开始执行 WAIT FOR 语句后继的语句。例如：

```
WAIT FOR 20 ns;
WAIT FOR (a * (b+c));
```

在上例的第一个语句中，时间表达式是一个常数值 20 ns，当进程执行到该语句时将等待 20 ns。一旦 20 ns 时间到，进程将执行 WAIT FOR 语句的后继语句。

在上述的第二个语句中，FOR 后面是一个时间表达式，a * (b+c)是时间量。WAIT FOR 语句在等待过程中，要对表达式进行一次计算，计算结果返回的值就作为该语句的等待时间。例如，a=2，b=50 ns，c=70 ns。那么 WAIT FOR(a * (b+c))这个语句将等待 240 ns，也就是说该语句和 WAIT FOR 240 ns 是等价的。

4. 多条件 WAIT 语句

在前面已叙述的 3 个 WAIT 语句中，等待的条件都是单一的，要么是信号量，要么是布尔量，要么是时间量。实际上 WAIT 语句还可以同时使用多个等待条件。例如：

```
WAIT ON nmi, interrupt UNTIL((nmi=TRUE)
        OR (interrupt=TRUE)) FOR 5 μs;
```

上述语句等待的是 3 个条件：

第一，信号量 nmi 和 interrupt 任何一个有一次新的变化；

第二，信号量 nmi 或 interrput 任何一个取值为“真”；

第三，该语句已等待 5 μs。

只要上述 3 个条件中一个或多个条件满足，进程将再次启动，继续执行 WAIT 语句的后继语句。

应该注意的是，在多条件等待时，表达式的值至少应包含一个信号量的值，例如：

```
WAIT UNTIL (interrupt=TRUE) OR
                (old_clk='1');
```

如果该语句的 interrupt 和 old_clk 两个都是变量，而不是信号量，那么，即使两个变量的值有新的改变，该语句也不会对表达式进行评估和计算(事实上，在挂起的进程中变量的值是不可能改变的)。这样，该等待语句将变成无限的等待语句，包含该等待语句的进程就不能再启动。在多种等待条件中，只有信号量变化才能引起等待语句表达式的一次新的评价和计算。

5. 超时等待

往往存在这样一种情况，在你所设计的程序模块中，等待语句所等待的条件，在实际执行时不能保证一定会碰到。在这种情况下，等待语句通常要加一项超时等待项，以防止该等待语句进入无限期的等待状态。但是，如果采用这种方法，应作适当的处理，否则就会产生错误的行为，如例 5 - 3 所示。

【例 5 - 3】

```
ARCHITECTURE wait_example OF wait_example IS
  SIGNAL sendB, sendA: STD_LOGIC;
BEGIN
    sendA<='0';
    A: PROCESS
    BEGIN
      WAIT UNTIL sendB='1';
      sendA<='1' AFTER 10 ns;
      WAIT UNTIL sendB='0';
      sendA<='0' AFTER 10 ns;
    END PROCESS A;
    B: PROCESS
    BEGIN
      WAIT UNTIL sendA='0';
      sendB<='0' AFTER 10 ns;
      WAIT UNTIL sendA='1';
      sendB<='1' AFTER 10 ns;
    END PROCESS B;
END wait_example;
```

在例 5 - 3 中，一个构造体内包含有两个进程。这两个进程通过两个信号量 sendA 和 sendB 进行通信。尽管该例子实际上并不做任何事情，但是它可以说明为什么等待语句会处于无限期的等待状态，也就是通常所说的“死锁”状态。

在仿真的最初阶段，所有的进程都将会执行一次。进程通常在仿真启动的执行点得到启动。在本例中进程 A 在仿真启动点启动，而在下述执行语句被挂起：

```
WAIT UNTIL sendB='1';
```

此时，进程 B 同样在启动点被启动，而在执行到下述这一行语句时被挂起：

```
WAIT UNTIL sendA='1';
```

B 进程启动以后不会停留在第一条等待语句 WAIT UNTIL sendA='0'。这是因为该构造体中的第一条语句是 sendA<='0'。它使 B 进程中的第一条等待语句，已满足了等待条件，可以继续执行后继的语句。此后 B 进程向下执行将“0”代入 sendB，而后停留在 B 进程的第二条等待语句上。这样，两个进程就处于相互等待状态，两个进程都不能继续执行。因为两个进程各自等待的条件都需要对方继续执行。如果在每一个等待语句中插入一个超时等待项，那么就可以允许进程继续执行，而不至于进入死锁状态。为了检测出进程在没有遇到等待条件而继续向下执行的情况，在等待语句后面可以加一条 ASSERT(断言)语句。加有超时等待项的例程如例 5 - 4 所示。

【例 5 - 4】

```
ARCHITECTURE wait_timeout OF wait_example IS
    SIGNAL sendA, sendB: STD_LOGIC;
    BEGIN
      A:PROCESS
      BEGIN
        WAIT UNTIL (sendB='1') FOR 1 μs;
        ASSERT (sendB='1')
          REPORT "sendB timed out at '1'"
          SEVERITY ERROR;
        sendA<='1' AFTER 10 ns;
        WAIT UNTIL (sendB='0') FOR 1 μs;
        ASSERT (sendB='0')
          REPORT "sendB timed out at '0'"
          SEVERITY ERROR;
        sendA<='0' AFTER 10 ns;
    END PROCESS A;
    B: PROCESS
    BEGIN
        WAIT UNTIL (sendA='0') FOR 1 μs;
        ASSERT (sendA='0')
          REPORT "sendA timed out at '0'"
          SEVERITY ERROR;
        sendB<='0' AFTER 10 ns;
        WAIT UNTIL (sendA='1') FOR 1 μs;
        ASSERT (sendA='1')
          REPORT "sendA timed out at '1'"
          SEVERITY ERROR;
        sendB<='1' AFTER 10 ns;
      END PROCESS B;
    END wait_timeout;
```

在例 5 - 4 中每个等待语句的超时表达式用 1 μs 说明。如果等待语句的等待时间超过了 1 μs，进程将执行下一条 ASSERT 语句。ASSERT 语句判断条件为“假”，就向操作人员提供错误信息输出，从而有助于操作人员了解在进程中发生了超时等待。

5.1.2 断言(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。SEVER-

ITY 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，在例 5－4 A 进程中的第一个等待语句后面跟的 ASSERT 语句：

```
ASSERT (sendB='1')
  REPORT "sendB timed out at '1'"
  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。

5.1.3 信号代入语句

信号代入语句的情况在第 4 章中已有详述。这里只作归纳性的介绍。

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“<＝”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。

5.1.4 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、范围及初值在事先应已给出过。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:=3.0;
c:=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。

5.1.5 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 5－5 所示。

【例 5－5】

```
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号量 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。例 5－6 就是 D 触发器的 VHDL 语言描述。

【例 5－6】

```
LIBRARY IEEE;
USE IEEE. STD_LOGIC_1164. ALL;
ENTITY dff IS
    PORT (clk, d: IN STD_LOGIC;
                q: OUT STD_LOGIC);
END dff;
ARCHITECTURE rtl OF dff IS
BEGIN
    PROCESS(clk)
        BEGIN
            IF(clk'EVENT AND clk='1') THEN
              q<=d;
            END IF;
    END PROCESS;
END rtl;
```

在例 5－6 中 IF 语句的条件是时钟信号 clk 发生变化，且时钟信号 clk='1'。只是在这个时候 q 端输出复现 d 端输入的信号值。当该条件不满足时，q 端维持原来的输出值。

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 之间所界定的顺序处理语句。当 IF 语句所指定的条件不满足时，将执行 ELSE 和 END IF 之间所界定的顺序处理语句。也就是说，用条件来选择两条不同程序执行的路径。

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 5 - 7 所示。

【例 5 - 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 5 - 8 所示。

【例 5 - 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。

需要注意的是，IF 语句的条件判断输出是布尔量，即是“真”(TRUE)或“假”(FALSE)。因此在 IF 语句的条件表达式中只能使用关系运算操作(=，/=，<，>，<=，>=)及逻辑运算操作的组合表达式。

5.1.6 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，程序的阅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 5 - 9 所示。

【例 5 - 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: INTEGER RANGE 0 TO 3;
BEGIN
  B: PROCESS(a,b,i0,i1,i2,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 5 - 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。但是它们两者还是有区别的。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。

带有 WHEN OTHERS 项的三一八译码器的行为描述的例子，如例 5 - 10 所示。

【例 5 - 10】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY decode_3to8 IS
  PORT(a,b,c,G1,G2A,G2B: IN STD_LOGIC;
       y: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END decode_3to8;
ARCHITECTURE rtl OF decode_3to8 IS
```

```
SIGNAL indata: STD_LOGIC_VECTOR(2 DOWNTO 0);
BEGIN
  indata<=c & b & a;
  PROCESS(indata,G1,G2A,G2B)
    BEGIN
    IF(G1='1' AND G2A='0' AND G2B='0') THEN
      CASE indata IS
        WHEN "000"=>y<="11111110";
        WHEN "001"=>y<="11111101";
        WHEN "010"=>y<="11111011";
        WHEN "011"=>y<="11110111";
        WHEN "100"=>y<="11101111";
        WHEN "101"=>y<="11011111";
        WHEN "110"=>y<="10111111";
        WHEN "111"=>y<="01111111";
        WHEN OTHERS=>y<="XXXXXXXX";
      END CASE;
    ELSE
      y<="11111111";
    END IF;
  END PROCESS;
END rtl;
```

在例 5 - 10 中，indata 是矢量型数据，除了取值为“0”和“1”之外，还有可能取值为“X”，“Z”和“U”。尽管这些取值在逻辑电路综合时没有用，但是，在 CASE 中却必须把所有可能取的值都要描述出来，故在本例中应加一项 WHEN OTHERS 项，使得它包含了 y 输出的所有缺省值。当 WHEN 后跟的值不同，但是输出相同时，则可以用“1”符号来描述。例如，本例中 WHEN OTHERS 项也可以写成：

```
WHEN "UZX"|"ZXU"|"UUZ"|…|"UUU"
      =>y<="XXXXXXXX";
```

所有“U”，“Z”，“X”三种状态的排列，表示了不同的取值。但是，所有这些排列，使其三—八译码器的输出值是一致的。因此 WHEN 后面可以用 OTHERS 符号来列举所有可能的取值。

同样，当输入值在某一个连续范围内，其对应的输出值是相同的，此时在用 CASE 语句时，在 WHEN 后面可以用“TO”来表示一个取值的范围。例如，对自然数取值范围为 1～9，则可表示为 WHEN 1 TO 9 =>…。

应该再次提醒的是 WHEN 后跟的“=>”符号不是关系运算操作符，它在这里仅仅描述：值和对应执行语句的对应关系。

在进行组合逻辑电路设计时，往往会碰到任意项，即在实际正常工作时不可能出现的那些输入状态。在利用卡诺图对逻辑进行化简时，可以把这些项看作“1”或者“0”，从而可以使逻辑电路得到简化。

现在来看一下，怎样用 CASE 语句来描述这种逻辑设计时的任意项。

例 5 - 10 是一个三一八译码电路。将三一八译码电路的输入变为输出，输出变为输入，它就变为二一十进制的编码电路。该电路的功能描述如例 5 - 11 所示。

【例 5 - 11】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY encodey IS
  PORT(input: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
             y: OUT STD_LOGIC_VECTOR (2 DOWNTO 0));
END encoder;
ARCHITECTURE rtl OF encoder IS
BEGIN
  PROCESS(input)
    BEGIN
      CASE input IS
        WHEN "01111111"=>y<="111";
        WHEN "10111111"=>y<="110";
        WHEN "11011111"=>y<="101";
        WHEN "11101111"=>y<="100";
        WHEN "11110111"=>y<="011";
        WHEN "11111011"=>y<="010";
        WHEN "11111101"=>y<="001";
        WHEN "11111110"=>y<="000";
        WHEN OTHERS=>y<="XXX";
      END CASE;
    END PROCESS;
END rtl;
```

在例 5 - 11 中的 WHEN OTHERS 语句和例 5 - 10 中的 WHEN OTHERS 语句尽管最后都将使“X”值代入 y。但是其含义是不一样的。在例 5 - 10 中，在正常情况下的所有的输入状态从 000～111 都在 CASE 语句的 OTHERS 之前罗列出来了。因此在逻辑综合时就不会有什么不利影响。而在例 5 - 11 中输入的所有状态并未在 CASE 语句的 OTHERS 之前都罗列出来。例如，当某一项输入同时出现两个或两个以上的“0”时，y 输出值就将变为“X”(可能是“0”也可能是“1”)。如果逻辑综合时，可以认为这些是不可能的输出项。那么就可以大大简化逻辑电路的设计。在仿真时如果出现了不确定的“X”值就可以检查是否出现了不正确的输入。

如果要想用 CASE 语句描述具有两个以上的“0”的情况，并使它们针对某一特定的 y 输出，例如 OTHERS 改写为：

```
WHEN OTHERS=>y<="111";
```

那么，在逻辑电路综合时，就会使电路的规模和复杂性大大增加。

在目前 VHDL 语言的标准中还没有能对输入任意项进行处理的方法。例如优先级编码器的真值表如表 5 - 1 所示。其中，标有“－”符号的输入项为任意项。也就是说标有“－”的位，其值可以取“1”也可以取“0”。如果想用 CASE 语句来描述优先级编码电路就必须用

到下述这样的语句：

```
WHEN "XXXXXXX0"=>y<="111";
WHEN "XXXXXX01"=>y<="110";
        ⋮
```

表 5-1 优先级编码器的真值表

输入								输出		
b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	b_2	b_1	b_0
—	—	—	—	—	—	—	0	1	1	1
—	—	—	—	—	—	0	1	1	1	0
—	—	—	—	—	0	1	1	1	0	1
—	—	—	—	0	1	1	1	1	0	0
—	—	—	0	1	1	1	1	0	1	1
		0	1	1	1	1	1	0	1	0
	0	1	1	1	1	1	1	0	0	1
—	1	1	1	1	1	1	1	0	0	0

显然，这样的描述语句在 VHDL 语言中还未制订出来，因此不能使用这种非法的语句。此时利用 IF 语句则能正确地描述优先级编码器的功能，如例 5-12 所示。

【例 5-12】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY priorityencoder IS
  PORT(input: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
             y: OUT STD_LOGIC_VECTOR (2 DOWNTO 0));
END priorityencoder;
ARCHITECTURE rtl OF priorityencoder IS
BEGIN
  PROCESS(input)
    BEGIN
      IF(input(0)='0') THEN
        y<="111";
      ELSIF(input(1)='0') THEN
        y<="110";
      ELSIF(input(2)='0') THEN
        y<="101";
      ELSIF(input(3)='0') THEN
        y<="100";
      ELSIF(input(4)='0') THEN
        y<="011";
```

```
        ELSIF(input(5)='0') THEN
          y<="010";
        ELSIF(input(6)='0') THEN
          y<="001";
        ELSE
          y<="000";
        END IF;
    END PROCESS;
END rtl;
```

在例 5－12 中 IF 语句首先判别 input(0)是否为“0”，然后再依顺序判别下去。如果该程序中首先判别 input(6)是否为“0”，然后再判 input(5)是否为“0”，这样一直判别到 input(0)是否为“0”。尽管每种情况所使用的条件是一样的，而且每种条件也只用到一次，但是其结果却是不一样的。例 5－12 中所采用的判别顺序是正确的，它正确地反映了优先级编码器的功能；而后者由 input(6)到 input(0)进行判别的顺序是错误的，它不能正确反映优先级编码器的功能，其原因请读者自己思考。

通常在 CASE 语句中，WHEN 语句可以颠倒次序而不致于发生错误，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。这一点希望读者切记。

在大多数情况下，能用 CASE 语句描述的逻辑电路，同样也可以用 IF 语句来描述。例如，例 5－8 用 IF 语句描述的四选一电路和例 5－9 用 CASE 语句描述的四选一电路。

目前 IEEE 正在对任意项描述的 VHDL 语言标准进行深入探讨，相信在不久的将来，像优先级编码器那样的逻辑电路也完全可以用 CASE 语句进行描述。

5.1.7 LOOP 语句

LOOP 语句与其它高级语句中的循环语句一样，使程序能进行有规则的循环，循环的次数受迭代算法控制。在 VHDL 语言中常用来描述位片逻辑及迭代电路的行为。

LOOP 语句的书写格式一般有两种：

1. FOR 循环变量

这样 LOOP 语句的书写格式如下：

```
[标号]: FOR  循环变量  IN  离散范围  LOOP
          顺序处理语句;
          END LOOP [标号];
```

LOOP 语句中的循环变量的值在每次循环中都将发生变化，而 IN 后跟的离散范围则表示循环变量在循环过程中依次取值的范围。例如：

```
ASUM: FOR i IN 1 TO 9 LOOP
          sum=i+sum; --sum 初始值为 0
END LOOP ASUM;
```

在该例子中 i 是循环变量，它可取值 1，2，…，9 共 9 个值，也就是说 sum=i+sum 的算式应循环计算 9 次。该程序对 1～9 的数进行累加计算。

例 5－13 是 8 位的奇偶校验电路的 VHDL 语言描述的实例。

【例 5 - 13】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY parity_check IS
  PORT(a: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
       y: OUT STD_LOGIC);
END parity_check IS
ARCHITECTURE rtl OF parity_check IS
BEGIN
  PROCESS(a)
  VARIABLE tmp: STD_LOGIC;
  BEGIN
    tmp:='0';
    FOR i IN 0 TO 7 LOOP
      tmp:=tmp XOR a(i);
    END LOOP;
    y<=tmp;
  END PROCESS;
END rtl;
```

在例 5 - 13 中有几点需要说明：tmp 是变量，它只能在进程内部说明，因为它是一个局部量。FOR－LOOP 语句中的 i 无论在信号说明和变量说明中都未涉及，它是一个循环变量。如前例所述，它是一个整数变量。信号和变量都不能代入到此循环变量中。tmp 是变量，如果该变量值要从进程内部输出就必须将它代入信号量，信号量是全局的，可以将值带出进程。在例 5 - 13 中 tmp 的值通过信号 y 带出进程。

2. WHILE 条件

这种 LOOP 语句的书写格式如下：

```
[标号]: WHILE  条件  LOOP
          顺序处理语句;
    END LOOP [标号];
```

在该 LOOP 语句中，如果条件为“真”，则进行循环；如果条件为“假”，则结束循环。例如：

```
i:=1;
sum:=0;
sbcd:WHILE (i<10) LOOP
     sum:=i+sum;
     i:=i+1;
END LOOP sbcd;
```

该例和 FOR－LOOP 语句例的行为是一样的，都是对 1～9 的数求累加和的运算。这里利用了 i<10 的条件使程序结束循环，而循环控制变量 i 的递增是通过算式 i:=i+1 来实现的。

例 5 - 13 的 8 位奇偶校验电路的行为如果用 WHILE 条件的 LOOP 语句来描述，即可

写为如例 5 - 14 所示的程序。

【例 5 - 14】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY parity_check IS
  PORT(a: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
       y: OUT STD_LOGIC);
END parity_check;
ARCHITECTURE behav OF parity_check IS
BEGIN
  PROCESS(a)
  VARIABLE tmp: STD_LOGIC;
  BEGIN
    tmp:='0';
    i:=0;
    WHILE (i<8) LOOP
      tmp:=tmp XOR a(i);
      i:=i+1;
    END LOOP;
    y<=tmp;
  END PROCESS;
END behav;
```

虽然 FOR－LOOP 和 WHILE－LOOP 语句都可以用来进行逻辑综合，但是一般都不太采用 WHILE－LOOP 语句来进行 RTL 描述。

5.1.8 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 5 - 15 所示。

【例 5 - 15】

```
PROCESS(a,b)
  CONSTANT max_limit: INTEGER:=255;
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;
```

当 LOOP 语句嵌套时，通常 NEXT 语句应标有“标号”和“WHEN 条件”。例如，有一个 LOOP 嵌套的程序如下所示：

```
L1: WHILE i<10 LOOP
L2: WHILE j<20 LOOP
      ⋮
    NEXT L1 WHEN i=j;
      ⋮
END LOOP L2;
END LOOP L1;
```

在上例中，当 i=j 时，NEXT 语句被执行，程序将从内循环跳出，而再从下一次外循环开始执行。

由此可知，NEXT 语句实际上是用于 LOOP 语句的内部循环控制。

5.1.9 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，而从 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 5 - 16 所示。

【例 5 - 16】

```
PROCESS(a)
  VARIABLE int_a: INTEGER;
BEGIN
  int_a:=a;
  FOR i IN 0 TO max_limit LOOP
    IF(int_a<=0) THEN
      EXIT;
    ELSE
      int_a:=int_a-1;
      q(i)<=3.1416/REAL(a*i);
    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。

EXIT 语句具有 3 种基本的书写格式。第一种书写格式是 EXIT 语句没有“循环标号”或“WHEN 条件”。当条件为“真”执行 EXIT 语句时，程序将按如下顺序执行：执行 EXIT，程序将仅仅从当前所属的 LOOP 语句中退出。如果 EXIT 语句位于一个内循环 LOOP 语句中，即该 LOOP 语句嵌在任何其它的一个 LOOP 语句中。那么执行 EXIT，程序仅仅退出内循环，而仍然留在外循环的 LOOP 语句中。

第二种书写格式是 EXIT 语句后跟 LOOP 语句的标号。此时，执行 EXIT 语句时，程序将跳至所说明的标号。

第三种书写格式是 EXIT 语句后跟“WHEN 条件”语句。当程序执行到该语句时，只有所说明的条件为“真”的情况下，才跳出循环的 LOOP 语句。此时，不管 EXIT 语句是否有标号说明，都将执行下一条语句。如果有标号说明，下一条要执行的语句将是标号所说明的语句。如果无标号说明，下一条要执行的语句是循环外的下一条语句。

EXIT 语句是一条很有用的控制语句。当程序需要处理保护、出错和警告状态时，它能提供一个快捷、简便的方法。

5.2 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。下面介绍一下各种并发语句的使用。

5.2.1 进程(PROCESS)语句

PROCESS 语句在前面已多次提到，并在众多实例中得到了广泛的使用。进程语句是一种并发处理语句，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：

- 它可以与其它进程并发运行，并可存取构造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，在进程结构中必须包含一个显式的敏感信号量表或者包含一个 WAIT 语句；
- 进程之间的通信是通过信号量传递来实现的。

后面要提到的一些并发语句，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。

5.2.2 并发信号代入(Concurrent Signal Assignment)语句

在 4.1 节中已详述了代入语句的功能和相关的问题。在这里重提代入语句，并且冠以“并发信号”的词句，主要是强调该语句的并发性。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缩写。例如：

```
ARCHITECTURE behav OF a_var IS
BEGIN
    output<=a(i);
END behav;
```

可以等效于

```
ARCHITECTURE behav 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，使其值有了一个新的改变，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。

5.2.3 条件信号代入(Conditional Signal Assignment)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。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 5 - 17 就是利用条件信号代入语句来描述的四选一逻辑电路。

【例 5 - 17】

```
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="11" 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。

5.2.4 选择信号代入(Selective Signal Assignment)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 5 - 18 所示。

【例 5 - 18】

```
LIBRARY IEEE;
USE IEEE.STD_LDGIC_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(例如 sel)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 5 - 19 所示。

【例 5 - 19】

```
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: STD_LOGIC);
END mux4;
```

```
ARCHITECTURE rtl OF mux4 IF
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 5-18 和例 5-19 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。

5.2.5 并发过程调用(Concurrent Procedure Call)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在 2.2 节中已详述，这里仅就调用时应注意的几个问题作一说明。

· 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；

· 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；

· 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 q。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。如 2.2 节所述，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。

另外，过程还存在这样一个问题，尽管某一个目标量并未编入过程的自变量表中，但是过程中的目标量的值却发生了变化。例如，过程中某一语句的代入操作可能使构造体中的一个信号量的值发生变化，而这个信号量并未编入过程的自变量表中。再例如，如果有两个信号并未在过程的自变量表中说明，但是在现过程的过程调用中发生了代入操作，这样的信号量的代入操作都会带来问题。因此在编写过程语句时应很好地注意这个问题，不要使类似的问题发生。

5.2.6 块(BLOCK)语句

在 2.2 节中已经介绍，BLOCK 语句是一个并发语句，而它所包含的一系列语句也是并发语句，而且块语句中的并发语句的执行与次序无关。为便于 BLOCK 语句的使用，这里再详细介绍一下 BLOCK 语句的书写格式。BLOCK 语句的书写格式一般为：

```
标号：BLOCK
      块头
      {说明语句};
      BEGIN
      {并发处理语句};
      END BLOCK 标号名;
```

在这里，块头主要用于信号的映射及参数的定义，通常通过 GENERIC 语句、GENERIC_MAP 语句以及 PORT 语句和 PORT_MAP 语句来实现。

说明语句与构造体的说明语句相同，主要是对该块所要用到的客体加以说明。可说明的项目有：

- USE 子句；
- 子程序说明及子程序体；
- 类型说明；
- 常数说明；
- 信号说明；
- 元件说明。

BLOCK 语句常用于构造体的结构化描述。为了更好地了解 BLOCK 语句的使用方法，这里再举一个使用实例。

如果想设计一个 CPU 芯片，为简化起见，假设这个 CPU 只由 ALU 模块和 REG8(寄存器)模块组成。ALU 模块和 REG8 模块的行为分别由两个 BLOCK 语句来描述。每个模

块相当于 CPU 电原理图中的子原理图(REG8 模块又由 8 个 REG1，REG2，…，REG8 子块构成)。在每个块内能够有局部信号、数据类型、常数等说明。任何一个客体可以在构造体中说明，也可以在块中说明，如例 5 - 20 所示。

【例 5 - 20】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
PACKAGE BIT 32 IS
  TYPE tw32 IS ARRAY(31 DOWNTO 0)
            OF STD_LOGIC;
END BIT 32;
USE IEEE.STD_LOGIC_1164.ALL;
USE WORK.BIT32.ALL;
ENTITY CPU IS
  PORT(clk, interrupt: IN STD_LOGIC;
       addr: OUT tw32; data: INOUT tw32);
END CPU
ARCHITECTURE cpu_blk OF cpu IS
  SIGNAL ibus, dbus: tw32;
BEGIN
  ALU: BLOCK
    SIGNAL qbus: tw32;
  BEGIN
      --ALU 行为描述语句
  END BLOCK ALU;
  REG8: BLOCK
    SIGNAL zbus: tw32;
  BEGIN
    REG1:BLOCK
      SIGNAL qbus: tw32;
    BEGIN
      --REG1 行为描述语句
    END BLOCK REG1;
      --其它 REG8 行为描述语句
    END BLOCK REG8;
END cpu_blk;
```

在例 5 - 20 中，CPU 模块有 4 个端口用作与外面的接口。其中 clk，interrupt 是输入端口；addr 是输出端口；data 是双向端口。在该实体的构造体中的所有 BLOCK，对这些信号都是用显式说明的，全都可以在 BLOCK 内使用。

信号 ibus 和 dbus 是构造体 cpu_blk 中的局部信号量，它只能在构造体 cpu_blk 中使用，在构造体 cpu_blk 之外不能使用。只要是在 cpu_blk 构造体内，无论在哪一个 BLOCK 块中这些信号量都是可以使用的。另外，由于 BLOCK 块是可以嵌套的，内层 BLOCK 块能够使用外层 BLOCK 块所说明的信号，而外层 BLOCK 块却不能够使用内层 BLOCK 块中

所说明的信号。

例如，例 5－20 中的 qbus 信号，只在 ALU 块中说明。因此，它是 ALU 块的局部信号量，所有 ALU 块中的语句都可以使用 qbus 信号，但是在 ALU 块之外则不能使用该信号，如 REG8 块就不能使用 qbus 信号。再譬如，zbus 信号是 REG8 块所说明的局部信号量，REG1 块是嵌套在 REG8 块中的内层块，所以 REG1 块可以使用 zbus 信号。

在 REG1 块的信号说明项中也有一个称为 qbus 的信号，该信号与 BLOCK 块中所说明的信号具有相同名字。那么，这样做会不会引起问题呢？事实上，编译器将分别对这两个信号进行处理。在语法上虽然是合法的，但是容易引起混淆。两个信号分别在各自的说明区域中加以说明，同样也仅仅在这些范围中有效。因此，可以这样认为，它们是具有相同信号名的各自独立的信号。每个 qbus 只能在所说明的 BLOCK 块区域中使用。

还有一种应该注意的情况如下所示：

```
BLK1: BLOCK
  SIGNAL qbus: tw32;
BEGIN
  BLK2: BLOCK
    SIGNAL qbus: tw32;
  BEGIN
      --BLK2 语句
  END BLOCK BLK2;
      --BLK1 语句
END BLOCK BLK1;
```

在上述的实例中，信号 qbus 在两个块中都作了说明。应注意的是，这两个块是嵌套关系，一个块包含另一个块。现在先来看一下 BLK2 块对信号 qbus 的操作。第一种类型的操作是 BLK2 中的语句对 BLK2 中所说明的 BLK2 局部信号量 qbus 进行的操作；第二种类型的操作是 BLK2 中的语句对 BLK1 中所说明的局部信号量 qbus 进行的操作(由于 BLK1 包含 BLK2，因此这种操作是允许的)。由此可见，BLK1 块所说明的信号可以看作是 BLK2 块内部说明的信号。如果名字相同，可以在信号名字前面加块名字前缀。例如，在本例中 BLK1 块的 qbus 可以用 BLK1_qbus 来表示。

通常，这种同名信号会使编程发生混乱。出现这个问题的起因是，在有限时间内，如果不对信号说明进行详细分析，就不能保证正确地使用 qbus 信号。

以上所提到的只是块本身范围所涉及的问题。但是，块是一个独立的子结构，它可以包含 PORT 和 GENERIC 语句。这样就允许设计者通过这两个语句将块内的信号变化传递给块的外部信号，同样也可以将块外部的信号变化传递给块的内部。

PORT 和 GENERIC 语句的这种性能，将允许在一个新的设计中可重复使用 BLOCK 块。例如，在上例的 CPU 的设计中，如果需要扩展 ALU 部分的功能，设计一个新的 ALU 模块，使其完成新的所需要的功能。在新的 CPU 模块中 PORT 名和 GENERIC 名与原来的不一致，此时，如果在块中采用 PORT 和 GENERIC 映射就可以顺利解决这个问题。也就是说，在上例的基础上，在设计中映射信号名和产生参数，就可以建立一个新的 ALU 模块，如例 5－21 所示。

【例 5 - 21】

```
PACKAGE math IS
  TYPE tw32 IS ARRAY(31 DOWNTO 0)
        OF STD_LOGIC;
  FUNCTION tw_add(a,b: tw32)
        RETURN tw32;
  FUNCTION tw_sub(a,b:tw32)
        RETURN tw32;
END math;
USE WORK.math.ALL;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY cpu IS
  PORT(clk, interrupt:IN STD_LOGIC;
       add:OUT tw32; comt:IN INTEGER;
       data:INOUT tw32);
ARCHITECTURE cpu_blk OF cpu IS
  SIGNAL ibus,dbus:tw32;
BEGIN
  ALU: BLOCK
    PORT(abus,bbus:IN tw32;
         d_out:OUT tw32;
         ctbus:IN INTEGER);
    PORT MAP(abus=>ibus,bbus=>dbus,
       d_out=>data,ctbus=>comt);
    SIGNAL qbus:tw32;
BEGIN
  d_out<=tw_add(abus,bbus) WHEN
          ctbus=0 ELSE
        tw_sub(abus,bbus) WHEN
          ctbus=1 ELSE
        abus;
  END BLOCK ALU;
END;
```

从例 5 - 21 可以看出，除了端口和端口映射语句之外，ALU 的说明部分和前面例子中所述的是一样的。端口语句说明了端口号和方向，并且还说明了端口的数据类型。端口映射语句映射了带有信号的新的端口，或者映射了 BLOCK 块外部的端口。例如，本例中端口 abus 被映射到 cpu_blk 构造体内说明的局部信号 ibus；端口 bbus 被映射到 dbus；端口 d_out 和 ctbus 被映射到实体外部的端口。

映射实现了端口和外部信号之间的连接，使连接到端口的信号值发生变化，由原来的值变成一个新的值。如果这种变化发生在 ibus 上，则 ibus 上出现的新的值，将被传送到 ALU 块内，使得 abus 端口得到新的值。当然，其它有映射关系的端口也应如此。

5.3 其它语句和有关规定的说明

在 VHDL 语言中除了顺序描述语句和并发描述语句之外，还有一些说明语句，定义语句和一些具体的规定，在这一节中将逐一对未在前面介绍过的内容作一说明。

5.3.1 命名规则和注解的标记

在 VHDL 语言中大写字母和小写字母是没有区别的，这一点在前面的说明和程序实例中已经可以看到。也就是说，在所有的语句中写大写字母也可以，写小写字母也可以，混合起来写也可以。但是，有两种情况是例外，这就是用单引号括起来的字符常数和用双引号括起来的字符。这时大写字母和小写字母是有区别的。例如，在 STD_LOGIC 和 STD_LOGIC_VECTOR 代入不定值'X'时应注意。

```
SIGNAL a: STD_LOGIC;
SIGNAL b: STD_LOGIC_VECTOR(3 DOWNTO 0);
a<='X'; --X 用小字字母是错误的
b<="XXXX"; --X 用小写字母是错误的
```

在 VHDL 语言中所使用的名字(名称)，如信号名、实体名、构造体名、变量名等，在命名时应遵守如下规则：

(1) 名字的最前面应该是英文字母；

(2) 能使用的字符只有英文字母、数字和'－'；

(3) 不能连续使用'－'符号，在名字的最后也不能使用'－'符号。

例如：

```
SIGNAL a_bus:STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL 302_bus; … --数字开头的名字是错误的
SIGNAL b_@bus; … --@符号不能作为名称的字母，是错误的
SIGNAL a__bus; … --'－'符号在名称中不能连着使用，故是错误的
SIGNAL b_bus_; … --'－'符号不能在名称最后使用，故是错误的。
```

像其它高级语言一样，VHDL 语言的程序有注释栏目，可以对所编写的语句进行注释。注释从"--"符号开始到该项末尾(回车、换行符)结束。注释文字虽然不作为 VHDL 的语句予以处理，但是有时也用于其它工具和接口。

5.3.2 ATTRIBUTE(属性)描述与定义语句

VHDL 语言有属性预定义功能，该功能有许多重要的应用，例如检出时钟边沿，完成定时检查，获得未约束的数据类型的范围等。ATTRIBUTE 语句可以从所指定的客体中获得关心的数据或信息。譬如：

```
TYPE number IS INTEGER RANGE 9 DOWNTO 0;
```

如果 i 为一个整数变量，那么利用属性描述语句就可以得到设计者感兴趣的数据。

例如，要想得到 number 的最大值，那么就可以用下述变量赋值语句得到：

```
i:=number'HIGH; --i=9
```

如果想要得到 number 的最小值，同样可以利用下述的变量赋值语句得到：

```
i:=number'LOW; --i=0
```

通过预定义属性描述语句，可以得到客体的有关值、功能、类型和范围(区间)。预定义的属性类型有以下几种：

- 数值类；
- 函数类；
- 信号类；
- 数据类型类；
- 数据范围类。

下面对各类属性的具体应用作一说明。

1. 数值类属性

数值类属性用来得到数组、块或者一般数据的有关值。例如，可以用来得到数组的长度、数据的最低限制等。在数值类属性中还可以再分成 3 个子类，它们是：

- 一般数据的数值属性；
- 数组的数值属性；
- 块的数值属性。

1) 一般数据的数值属性

一般数据的数值属性有以下 4 种：

- T'LEFT——得到数据类或子类区间的最左端的值；
- T'RIGHT——得到数据类或子类区间的最右端的值；
- T'HIGH——得到数据类或子类区间的高端值；
- T'LOW——得到数据类或子类区间的低端值。

一般数据的数值属性的书写格式如下：

```
客体'属性名
```

上述的 T'LOW 表示：T 为客体，T 代表一般数据类或子类的名称，符号"'"紧跟客体的后面，符号"'"的后面是属性名，本例中属性名为 LOW。

LEFT 表示数据类或子类区间的左端，也就是说，它表示约束区间最左的入口点。例如，区间左端为 0，区间右端为 9。

```
TYPE number IS 0 To 9;
```

显然，RIGHT 表示数据类或子类的区间的右端，也就是约束区的最右端的入口点。

HIGH 表示数据类或子类区间的高端，也就是约束区间的最大值；LOW 表示数据类或子类区间的低端，也就是约束区间的最低值。按此表示方法，就可以写出如下关系式：

```
i:=number'LEFT; --i=0
i:=number'RIGHT; --i=9
i:=number'HIGH; --i=9
i:=number'LOW; --i=0
```

需要注意的是，变量 i 的数据类型应与赋值的区间的数据类型相同。例如，上例的 number 是正整数，那么 i 也应该是正整数。

下面来看一下用 DOWNTO 所表示的区间的例子，如例 5 - 22 所示。

【例 5 - 22】

```
PROCESS(a)
TYPE bit_range IS ARRAY(31 DOWNTO 0) OF BIT;
VARIABLE left_range,right_range, uprange,lowange:INTEGER;
BEGIN
  left_range:=bit_range'LEFT; --得到 31
  right_range:=bit_range'RIGHT; --得到 0
  uprange:=bit_range'HIGH; --得到 31
  lowrange:=bit_range'LOW; --得到 0
END PROCESS;
```

从例 5 - 22 可以看出，不同的属性可以得到不同的关于数据类的信息。例如，当数据类的区间用(a TO b)来定义时，那么 b>a，此时'LEFT 属性的值通常等于'LOW 属性的值。相反，如果数据类的区间用(b DOWNTO a)来定义时，那么 b>a，此时'LEFT 属性的值则与'HIGH 的属性值相对应。

数值类属性不光适用于数字类型，而且该属性还可以适用于任何标量类型。用于枚举类型的情况如例 5 - 23 所示。

【例 5 - 23】

```
ARCHITECTURE time1 OF time IS
  TYPE tim IS(sec,min,hous,day,moth,year);
  SUBTYPE reverse_tim IS
      tim RANGE month DOWNTO min;
  SIGNAL tim1,tim2,tim3,tim4,tim5,tim6,tim7,tim8:TIME;
BEGIN
  tim1<=tim'LEFT; --得到 sec
  tim2<=tim'RIGHT; --得到 year
  tim3<=tim'HIGH; --得到 year
  tim4<=tim'LOW; --得到 sec
  tim5<=reverse_tim'LEFT; --得到 min
  tim6<=reverse_tim'RIGHT; --得到 month
  tim7<=reverse_tim'HIGH; --得到 month
  tim8<=reverse_tim'LOW; --得到 min
END time1;
```

在例 5 - 23 中，信号 tim1 和 tim2 代入的是 sec 和 year，分别是区间的左端值和右端值。这一点很容易在类型说明中得到验证。但是，如何来说明，用'HIGH 和'LOW 属性来得到枚举类数据的数值属性呢？实际上，这里的'HIGH 和'LOW 表示的是数据类的位置序号值的大小。对于整数和实数来说，数值的位置序号值与数本身的值相等，而对于枚举类型的数据来说，在说明中较早出现的数据，其位置序号值低于较后说明的数据。例如，在例 5 - 23 中 sec 的位置序号为 0，因为它最先在类型说明中说明；同样，min 位置序号为 1，hous 为 2 等等。这样，位置序号大的，其属性为'HIGH；而位置序号小的，其属性为'LOW。

信号 tim5 到 tim8 代入的是 reverse_tim 类数据的属性值。该类数据的区间用 DOWN-

TO 来加以说明。此时，用属性'HIGH 和'RIGHT 得到的将不是同一个值(在用 TO 来说明区间时，两者的属性值是相同的)，其原因就在于区间内的数据说明颠倒了。在例 5 - 23 中，对 reverse_tim 数据类型来说，month 的位置序号大于 min 的位置序号。

2) 数组的数值属性

数组的数值属性只有一个，即'LENGTH。在给定数组类型后，用该属性将得到一个数组的长度值。该属性可用于任何标量类数组和多维的标量类区间的数组。例 5 - 24 就是一个简单应用的示例。

【例 5 - 24】

```
PROCESS(a)
  TYPE bit4 IS ARRAY (0 TO 3) of BIT;
  TYPE bit_strange IS ARRAY(10 TO 20) OF BIT;
  VARIABLE len1,len2: INTEGER;
BEGIN
  len1:=bit4'LENGTH; --len1=4
  len2:=bit_strange'LENGTH; --len2=11
END PROCESS;
```

在例 5 - 24 中，len1 代入的是数组 bit4 的元素个数；len2 代入的是数组 bit_strange 的元素个数。

该属性同样也可以用于枚举类型的区间，如例 5 - 25 所示。

【例 5 - 25】

```
PACKAGE p_4val IS
  TYPE t_4val IS('X','0','1','z');
  TYPE t_4valx1 IS ARRAY(t_4val'LOW TO
       t_4val'HIGH) OF t_4val;
  TYPE t_4valx2 IS ARRAY (t_4val'LOW TO
       t_4val'HIGH) OF t_4valx1;
  TYPE t_4valmd IS ARRAY
       (t_4val'LOW TO t_4val'HIGH,
        t_4val'LOW TO t_4val'HIGH) OF t_4val;
  CONSTANT andsd:t_4valx2:=
    (('X',    --XX
      '0',    --X0
      'X',    --X1
      'X'),   --XZ
      ('0',   --0X
      '0',    --00
      '0',    --01
      '0'),   --0Z
      ('X',   --1X
      '0',    --10
      '1',    --11
```

```
        'X'),    --1Z
        ('X',    --ZX
        '0',     --Z0
        'X',     --Z1
        'X'));   --ZZ
    CONSTANT andmd: t_4valmd:=
      (('X',     --XX
        '0',     --X0
        'X',     --X1
        'X'),    --XZ
        ('0',    --0X
        '0',     --00
        '0',     --01
        '0'),    --0Z
        ('X'     --1X
        '0',     --10
        '1',     --11
        'X'),    --1Z
        ('X',    --ZX
        '0',     --Z0
        'X',     --Z1
        X));     --ZZ
  END p_4val;
```

例 5－25 中的 andsd 和 andmd 是两个复合型常数，它们是 t_4val 类型数据的“与”函数的真值表。第一个常数 andsd 用数组的数组来表示其相“与”的值。第二个常数 andmd 用多维数组来表示它的取值。在 andsd 中“X”和“X”相“与”为“X”，“X”和“0”相“与”为“0”，“X”和“1”相“与”为“X”，“X”和“Z”相“与”为“X”，其它状态值也是根据逻辑“与”的功能得到的。

如果现在将属性'LENGTH 用于这些类型的数据，那么就可以得到例 5－26 注解中所注明的数值。

【例 5－26】

```
  PROCESS(a)
    VARIABLE len1,len2,len3,len4: INTEGER;
  BEGIN
    len1:=t_4valx1'LENGTH; --得到 4
    len2:=t_4valx2'LENGTH; --得到 4
    len3:=t_4valmd'LENGTH(1); --得到 4
    len4:=t_4valmd'LENGTH(2); --得到 4
  END PROCESS;
```

在例 5－26 中，t_4valx1是一个包含 4 个元素的数组，数组的区间用 t_4val 类型数据的'LOW 和'HIGH 属性来说明，因此 t_4valx1 的长度值应为 4。同理，len2 也将得到 4。这

是因为 t_valx2的区间是从数组元素 t_4valx1 的'LOW 到'HIGH，共有 4 个元素(一个数组为另一个数组的元素)。

代入 len3 和 len4 的是多维数组 t_4valmd 的'LENGTH 的属性值。由于多维数组有多个区间，因此在对某个区间取属性值时，在属性'LENGTH 后面应标注区间号，如例 5 - 26 中的'LENGTH(1)和 LENGTH(2)。如果不作特别说明，那么属性'LENGTH 得到的将是第一个区间的长度值。

3) 块的数值属性

块的数据属性有两种：'STRUCTURE 和'BEHAVIOR。这两种属性用于块(BLOCK)和构造体，通过它们可以得到块和构造体是怎么样的一个设计模块的信息。如果块有标号说明，或者构造体有构造体名说明，而且在块和构造体中不存在 COMPONENT 语句，那么用属性'BEHAVIOR 将得到"TRUE"的信息；如果在块和构造体中只有 COMPONENT 语句或被动进程，那么用属性'STRUCTURE 将得到"TRUE"的信息。

下面来看一个如例 5 - 27 所示的具体实例。

【例 5 - 27】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY shifter IS
  PORT(clk, left: IN STD_LOGIC;
       right: OUT STD_LOGIC);
END shifter;
ARCHITECTURE structural OF shifter IS
  COMPONENT dff
    PORT(d,clk: IN STD_LOGIC;
         q: OUT STD_LOGIC);
  END COMPONENT;
  SIGNAL i1,i2,i3: STD_LOGIC;
BEGIN
  u1:dff
    PORT MAP(d=>left,clk=>clk,q=>i1);
  u2:dff
    PORT MAP(d=>i1,clk=>clk,q=>i2);
  u3:dff
    PORT MAP(d=>i2,clk=>clk,q=>i3);
  u4:dff
    PORT MAP(d=>i3,clk=>clk,q=>right);
  checktime: PROCESS(clk)
    VARIABLE last_time: time:=time'LEFT;
  BEGIN
    ASSERT(NOW-last_time=20 ns)
      REPORT "spike on clock"
      SEVERITY WARNING;
```

```
        last_time := now;
    END PROCESS checktime;
END structural;
```

在例 5 - 27 中，移位寄存器模块由 4 个 D 触发器基本单元串联而成。在对应于 shifter 实体的构造体中，还包含有一个用于检出时钟 clk 跳变的被动进程 checktime。现在对这样的构造体施加属性'BEHAVIOR 和'STRUCTURE，那么就可以得到如下所描述的信息。

structural'BEHAVIOR - - 得到“假”

structural'STRUCTURE - - 得到“真”

由上述描述可知，实际上属性'BEHAVIOR 和'STRUCTURE 用来验证所说明的块或构造体是用结构描述方式来描述的模块还是用行为描述方式来描述的模块。这对设计人员检查程序是非常有用的。另外，上述中的 checktime 是被动进程，所谓被动进程，可以这么认为，它是一个无源的进程。如果在进程中包含有代入语句，那么该进程就不是被动进程了，它变成了一个有源进程或者称主动进程。如果 checktime 进程包含有代入语句，那么用属性'STRUCTURE得到的信息将不是“真”而是“假”了。

2. 函数类属性

所谓函数类属性是指属性以函数的形式，让设计人员得到有关数据类型、数组、信号的某些信息。当函数类属性以表达式形式使用时(例如'POS(x))，首先应指定一个输入的自变量值(如 x)，函数调用后将得到一个返回的值。该返回的值可能是枚举数据的位置序号，也可能是信号有某种变化的指示，还可能是数组区间中的某一个值。

函数类属性有以下 3 种：

- 数据类型属性函数
- 数组属性函数
- 信号属性函数

1) 数据类型属性函数

用数据类型属性函数可以得到有关数据类型的各种信息。例如，给出某类数据值的位置，那么利用位置函数属性就可以得到该位置的数值。另外，利用其它相应属性还可以得到某些值的左邻值和右邻值等等。

对数据类型属性函数再进行细分，可以有以下 6 种属性函数：

- 'POS(x)——得到输入 x 值的位置序号；
- 'VAL(x)——得到输入位置序号 x 的值；
- 'SUCC(x)——得到输入 x 值的下一个值；
- 'PRED(x)——得到输入 x 值的前一个值；
- 'LEFTOF(x)——得到邻接输入 x 值左边的值；
- 'RIGHTOF——得到邻接输入 x 值右边的值。

数据类型属性函数的一个典型应用是将枚举或物理类型的数据转换成整数。例 5 - 28 就是将物理量 μA，μV，ohm 转换成整数的实例。

【例 5 - 28】

```
PACKAGE ohms_law IS
  TYPE current IS RANGE 0 TO 1000000
```

```
    UNITS
      μA;
      mA=1000 μA;
       A=1000 mA;
    END UNITS;
  TYPE voltage IS RANGE 0 TO 1000000
    UNITS
      μV;
      mV=1000 μV;
       V=1000 mV;
    END UNITS;
  TYPE resistance IS RANGE 0 TO 1000000
    UNITS
      ohm;
      kohm=1000 ohm;
      mohm=1000 kohm;
    END UNITS;
END ohms_law;
USE work.ohms_law.ALL;
ENTITY calc_resistance IS
  PORT(i: IN current; e: IN voltage;
       r: OUT resistance);
END calc_resistance;
ARCHITECTURE behav OF calc_resistance IS
BEGIN
  ohm_proc: PROCESS(i,e)
    VARIABLE covi,cove,int_r: INTEGER;
  BEGIN
    convi:=current'pos(i);   --以微安为单位的电流值
    conve:=voltage'pos(e);   --以微伏为单位的电压值
    int_r:=conve/convi;      --以欧姆为单位的电阻值
    r<=resistance'VAL(int_r);
  END PROCESS;
END behav;
```

包集合 ohms_law 定义了 3 种物理类型的数据，即 current(电流)、voltage(电压)和 resistance (电阻)。例 5 - 28 的作用是将物理类型的数据转换成整数(conve,convi→int_r)，而后再由整数转换成物理类型的数据(int_r→r)。从这个转换和再转换的过程可以看出，它实际完成了由电压和电流值，计算电阻值的运算过程。当端口 i 和 e 中的任何一个发生变化时，ohm_proc 进程就被启动，根据新的电流(i)和电压(e)值计算得到新的电阻(r)值。

进程的第一条语句将输入电流值(i)的位置序号赋予变量 convi。例如，输入电流值为 10 μA，那么赋予变量 convi 的值为 10。

进程的第二条语句将输入电压值(e)的位置序号赋予变量 conve。电压的基本单位是

μV，因此，电压值的位置序号与输入电压的 μV 数相等。

进程的第三条语句是计算整数 conve 和 convi 的商，得到的是一个 int_r 整数值。该整数值与要得到的电阻的阻值是相等的，但是 int_r 不是物理量数据，要转换成物理量数据还需进行一次整数至物理量的转换，这就是进程中的第四条语句。

进程的第四条语句是将位置序号转换成数值的语句，即利用属性'VAL，将位置序号 int_r 转换成用欧姆表示的电阻值。

前面详述了属性'POS 和'VAL 的使用方法，下面再举例说明一下属性'SUCC，'PRED，'RIGHTOF和'LEFTOF。

例如有一个 t_time 的包集合，它定义了两类枚举型数据，如例 5 - 29 所示。

【例 5 - 29】

```
PACKAGE t_time IS
  TYPE time IS( sec, min, hous, day, month, year);
  TYPE reverse_time IS time RANGE
        year DOWNTO sec;
END t_time;
```

利用余下的 4 个属性，就可以得到 time 的不同值。现罗列如下：

time'SUCC(hous)——得到 day；

time'PRED(day)——得到 hous；

reverse_time'SUCC(hous)——得到 min；

reverse_time'PRED(day)——得到 month；

time'RIGHTOF(hous)——得到 day；

time'LEFTOF(day)——得到 hous；

reverse_time'RIGHTOF(hous)——得到 min；

reverse_time'LEFTOF(day)——得到 month。

由上述可知，对于递增区间来说，下面的等式将成立：

'SUCC(x)='RIGHTOF(x)；

'PRED(x)='LEFTOF(x)。

而对于递减区间来说，则与上述等式相反，下面两个等式将成立：

'SUCC(x)='LEFTOF(x)；

'PRED(x)='RIGHTOF(x)。

需要注意的是，如果一个枚举类型数据的极限值被传递给属性'SUCC 和'PRED 时，如本例中假设：

```
y:=sec;
x:=time'PRED(y);
```

第二个表达式将引起运行错误。这是因为，在枚举数据 time 中，最小的值是 sec，time'PRED(y)要求提供比 sec 更小的值，已超出了定义范围。

2）数组属性函数

利用数组属性函数可得到数组的区间。在对数组的每一个元素进行操作时，必须知道数组的区间。数组属性函数可分以下 4 种：

·'LEFT(n)——得到索引号为 n 的区间的左端位置号。在这里 n 实际上是多维数组中所定义的多维区间的序号。当 n 缺省时，就代表对一维区间进行操作。

·'RIGHT(n)——得到索引号为 n 的区间的右端位置号。

·'HIGH(n)——得到索引号为 n 的区间的高端位置号。

·'LOW(n)——得到索引号为 n 的区间的低端位置号。

上述属性与数值数据类属性一样，在递增区间和递减区间存在着不同的对应关系。

在递增区间，存在如下关系：

·'LEFT='LOW　数组'LEFT=数组'LOW

·'RIGHT='HIGHT　数组'RIGHT=数组'HIGHT

在递减区间，存在如下关系：

·'LEFT='HIGHT

·'RIGHT='LOW

下面举一个描述随机存贮器的例子，如例 5 - 30 所示。

【例 5 - 30】

```
PACKAGE p_ram IS
  TYPE ram_data IS ARRAY(0 TO 511)
        OF INTEGER;
  CONSTANT x_val: INTEGER:=-1;
  CONSTANT z_val: INTEGER:=-2;
END p_ram;
USE WORK.p_ram.ALL;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY ram IS
  PORT(data_in: IN INTEGER;
       addr: IN INTEGER;
       data: OUT INTEGER;
       cs: IN STD_LOGIC;
       r_wb: IN STD_LOGIC);
END ram;
ARCHITECTURE behav_ram OF ram IS
BEGIN
  main_proc: PROCESS(cs,addr,r_wb)
    VARIABLE ram_data1: ram_data;
    VARIABLE ram_init: boolean:=FALSE;
  BEGIN
    IF NOT(ram_init) THEN
      FOR i IN ram_data1'LOW TO
            ram_data1'HIGH LOOP
        ram_data1(i):=0;
      END LOOP;
      ram_init:=TRUE;
```

```
    END IF;
    IF(cs='X')OR (r_wb='X') THEN
      data<=x_val;
    ELSIF(cs='0') THEN
      data<=z_val;
    ELSIF(r_wb='1') THEN
      IF (addr=x_val) OR (addr=z_val) THEN
      data<=x_val;
    ELSE
      data<=ram_data1(addr);
      END IF;
  ELSE
    IF(addr=x_val) OR (addr=z_val) THEN
       ASSERT FALSE REPORT
         "WRITING TO UNKNOWN ADDRESS"
         SEVERITY ERROR;
       data<=x_val;
    ELSE
      ram_data1(addr):=data_in;
      data<=ram_data1(addr);
    END IF;
  END IF;
 END PROCESS;
END behav_ram;
```

例 5 - 30 描述的是一个输入输出整数的随机存贮器。该 RAM 有 512 个整数单元，由两条控制线进行数据输入输出控制。一条是片选线 cs，另一条是读/写线 r_wb。

该程序包含有一条 IF 语句，用来将 RAM 的各单元初始化为“0”值。布尔量 ram_init 用来指示 RAM 是否已被初始化。如果 ram_init 为“假”，则表明 RAM 未被初始化；如果 ram_init 为“真”，则表明 RAM 已被初始化。

在进程首次执行时，变量 ram_init 将是“假”状态，因此 IF 语句被执行。IF 语句内部的 LOOP 语句循环对每一个单元进行初始化，使用函数数组属性'LOW 和'HIGH 来控制初始化的循环区间。

该循环语句只要被执行一次，所有 RAM 单元将被初始化，并将 ram_init 置为“真”。设置变量 ram_init 为“真”，可防止程序再次对 RAM 进行初始化。

程序中的其它语句用于描述 RAM 的读写功能，并检查在输入端口的值是否是正确的值。

3) 信号属性函数

信号属性函数用来得到信号的行为信息。例如，信号的值是否有变化；从最后一次变化到现在经过了多长时间；信号变化前的值为多少等。

信号属性函数共有 5 种，它们是：

· s'EVENT——如果在当前一个相当小的时间间隔内，事件发生了，那么，函数将返

回一个为“真”的布尔量；否则就返回“假”。

• s'ACTIVE——如果在当前一个相当小的时间间隔内，信号发生了改变，那么，函数将返回一个为“真”的布尔量；否则就返回“假”。

• s'LAST_EVENT——该属性函数将返回一个时间值，即从信号前一个事件发生到现在所经过的时间。

• s'LAST_VALUE——该属性函数将返回一个值，即该值是信号最后一次改变以前的值。

• s'LAST_ACTIVE——该属性函数返回一个时间值，即从信号前一次改变到现在的时间。

(1) 属性'EVENT 和'LAST_VALUE

属性'EVENT 通常用于确定时钟信号的边沿，用它可以检查信号是否处于某一个特殊值，以及信号是否刚好已发生变化。下面就是一个用属性'EVENT 检出 D 触发器时钟脉冲上升沿的描述实例。

【例 5 - 31】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY dff IS
  PORT(d,clk:IN STD_LOGIC;
            q: OUT STD_LOGIC);
END dff;
ARCHITECTURE dff OF dff IS
BEGIN
  PROCESS(clk)
  BEGIN
    IF(clk='1') AND (clk'EVENT) THEN
          q<=d;
    END IF;
  END PROCESS;
END dff;
```

在例 5 - 31 中描述了 D 触发器的工作原理，当 D 触发器的时钟脉冲的上升沿到来时，其 D 输入端的值就被传送到输出端 Q。为了检出时钟脉冲的上升沿，就用到了属性'EVENT。上升沿的发生是由两个条件来约束的，即时钟脉冲目前处于“1”电平，而且时钟脉冲刚刚从其它电平变为“1”电平。

在上例中，如果原来的电平为“0”，那么逻辑是正确的。但是，如果原来的电平是“X”(不定状态)，那么上例的描述同样也被认为出现了上升沿，显然这种情况是错误的。为了避免出现这种逻辑错误，最好使用属性'LAST_VALUE。这样上例中的 IF 语句可以作如下改写：

```
IF(clk='1') AND (clk'EVENT)
      AND (clk'LAST_VALUE='0') THEN
    q<=d;
END IF;
```

该语句保证时钟脉冲在变成“1”电平之前一定是处于“0”状态。

值得注意的是，在上面的两种应用场合使用属性'EVENT 并不是必需的。因为该进程中只有一个敏感信号量 clk，该进程启动的条件是敏感信号量发生变化，其作用和'EVENT 的说明是一致的。但是，如果进程中有多个敏感信号量，那么用'EVENT 来说明哪一个信号发生变化是必需的。

(2) 属性'LAST_EVENT

用属性'LAST_EVENT 可得到信号上各种事件发生以来所经过的时间。该属性常用于检查定时时间，如检查建立时间、保持时间和脉冲宽度等。用于检查建立时间和保持时间的示例如图 5 - 1 所示。

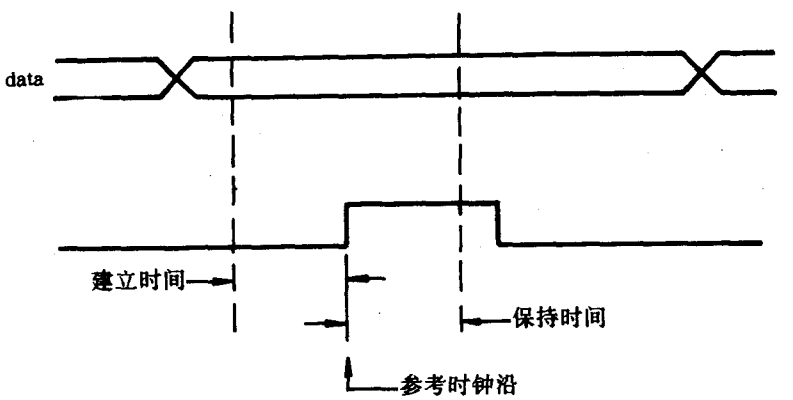


图 5 - 1　建立时间和保持时间示例

图 5 - 1 中的信号 clk，其上升沿是所有时间检查的参考沿。建立时间检查将保证数据输入信号在建立时间内不发生变化；而保持时间检查将保证在参考沿后面的一段规定的保持时间内数据输入信号不发生变化。通过这些检查就可以确保 D 触发器正常工作。

利用'LAST_EVENT 属性对建立时间进行检查的实例如例 5 - 32 所示。

【例 5 - 32】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL
ENTITY dff IS
  GENERIC(setup_time,hold_time: TIME);
  PORT(d,clk: IN STD_LOGIC
           q: OUT STD_LOGIC);
BEGIN
  setup_check: PROCESS(clk)
  BEGIN
    IF(clk='1') AND (clk'EVENT) THEN
       ASSERT(d'LAST_EVENT>=setup_time)
         REPORT "SETUP VIOLATION"
         SEVERITY ERROR;
```

```
    END IF;
  END PROCESS setup_check;
END dff;
ARCHITECTURE dff_behav OF dff IS
BEGIN
  dff_process: PROCESS(clk)
  BEGIN
    IF(clk='1') AND (clk'EVENT) THEN
        q<=d;
    END IF;
  END PROCESS dff_process;
END dff_behav;
```

由例 5 - 32 可以看到，建立时间的检查进程是一个无源进程，它被放在实体 dff 的模块中，检查当然也可以放在 dff 构造体的模块中。但是，在实体中的检查可以被该实体所属的所有构造体所共享，这一点是需要充分注意的。

信号 clk 每发生一次变化，都将执行一次该无源进程。在 clk 的上升沿时 ASSERT 语句将执行，并对建立时间进行检查。

ASSERT 语句将检查数据输入端 D 的建立时间是否大于或等于规定的建立时间，属性 d'LAST_EVENT 将返回一个信号 d 自最近一次变化以来到现在(clk 上升沿)clk 事件发生时为止所经过的时间。如果得到的时间小于规定的建立时间，那么就会发出错误警告。

(3) 属性'ACTIVE 和'LAST_ACTIVE

属性'ACTIVE 和'LAST_ACTIVE 在信号发生转换或事件发生时被触发。当一个模块的输入或输入输出端口发生某一事件时，将启动该模块执行，从而使信号发生转换。其转换后的值应与'ACTIVE 所指定的值相同，这样'ACTIVE 将返回一个“真”值，否则就会返回一个“假”值。属性'LAST_ACTIVE 将返回一个时间值，这个时间值就是所加信号发生转换或发生某一个事件开始到当前时刻的时间间隔。上述两个属性与'EVENT 和'LAST_EVENT 提供相对应的事件发生行为的描述。

3. 信号类属性

信号类属性用于产生一种特别的信号，这个特别的信号是以所加属性的信号为基础而形成的。也就是说，在这个特别的信号中包含了所加属性的有关信息。用这种信号类属性得到的有关信息非常类似于用函数类属性所得到的信号。所不同的是前者可以用于任何一般的信号，也包括敏感信号量表中所指定的信号。

信号类属性有 4 种，它们是：

· s'DELAYED[(time)]——该属性将产生一个延时的信号，其信号类型与该属性所加的信号相同，即以属性所加的信号为参考信号，经括号内时间表达式所确定的时间延时后所得的延迟信号。

· s'STABLE[(time)]——该属性可建立一个布尔信号，在括号内的时间表达式所说明的时间内，若参考信号没有发生事件，则该属性可得到“真”的结果。

· s'QUIET[(time)]——该属性可建立一个布尔信号，在括号内的时间表达式所说明的时间内，若参考信号没有发生转换或其它事件，则属性可以得到“真”的结果。

· s'TRANSACTION——该属性可以建立一个 BIT 类型的信号，当属性所加的信号发生转换或事件时，其值都将发生改变。

需要注意，上述的信号类属性不能用于子程序中，否则程序在编译时会出现编译错误信息。

(1) 属性'DELAYED

属性'DELAYED 可建立一个所加信号的延迟版本。为实现同样的功能，也可以用传送延时赋值语句(Transport delay)来实现。两者不同的是，后者要求编程人员用传送延时赋值的方法记入程序中，而且带有传送延时赋值的信号是一个新的信号，它必须在程序中加以说明。

下面来看一下属性'DELAYED 实际应用的例子。在建立 ASIC 器件模型时，有一种方法采用器件输入引脚的通路相关延时的模型，如图 5 - 2 所示。

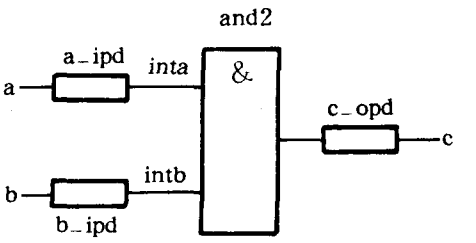


图 5 - 2　ASIC 器件的通路相关延时模型

在设计以前，要估计每一个输入的延时。在设计以后，反过来要注明实际的延时值，并且再次对实际延时情况进行仿真。提供实际延时值的方法之一，是在器件的配置(Confingurations)中用 GENERIC 产生所说明的延时值。一个对如图 5 - 2 中 and2 进行描述的典型模块，如例 5 - 33。

【例 5 - 33】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY and2 IS
  GENERIC(a_ipd,b_ipd,c_opd:
          TIME);
  PORT(a,b:IN STD_LOGIC;
          c: OUT STD_LOGIC);
END and2;
ARCHITECTURE int_signals OF and2 IS
  SIGNAL inta,intb: STD_LOGIC;
BEGIN
  inta<=TRANSPORT a AFTER a_ipd;
  intb<=TRANSPORT b AFTER b_ipd;
    c<=inta AND intb AFTER c_opd;
END int_signals;
ARCHITECTURE attr OF and2 IS
BEGIN
  c<=a'DELAYED(a_ipd) AND b'DELAYED (b_ipd)
       AFTER c_opd;
END attr;
```

在例 5 - 33 中采用两种不同的方法来描述信号输入通道的延时。第一种方法采用传送延时描述，它重新定义两个中间信号作为延时后的信号，两个中间信号相“与”以后经延时再赋予输出端 c，从而完成整个器件的通道延时描述。第二种方法使用信号属性'DELAYED。输入信号 a 和 b 分别被已定义的延时时间 a_ipd 和 b_ipd 所延时，延时后的两个信号相“与”后再经 c_opd 延时时间而被赋予输出端口 c。

在使用'DELAYED 属性时，如果所说明的延时时间事先未加定义，那么实际的延时时间就被赋为 0 ns。

属性'DELAYED 还用于保持检查。

在前面的章节中已经讨论了建立时间和保持时间，并且举了利用属性'LAST_EVENT 实现建立时间检查的例子。现在为了实现保持时间的检查，需要使用延时后的 clk 信号版本，如例 5 - 34 所示。

【例 5 - 34】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL
ENTITY dff IS
  GENERIC(setup_time,hold_time: TIME);
  PORT(d,clk: IN STD_LOGIC;
           q: OUT STD_LOGIC);
BEGIN
  setup_check: PROCESS(clk)
  BEGIN
    IF(clk='1') AND (clk'EVENT) THEN
      ASSERT(d'LAST_EVENT<=setup_time)
      REPORT"setup violation"
      SEVERITY ERROR;
    END IF;
  END PROCESS setup_check;
  hold_check: PROCESS(clk'DELAYED(hold_time))
  BEGIN
    IF (clk'DELAYED(hold_time)='1') AND
      (clk'DELAYED(hold_time)'EVENT) THEN
      ASSERT(d'LAST_EVENT=0 ns) OR
           d'LAST_EVENT<hold_time)
        REPORT "hold violation"
        SEVERITY ERROR;
    END IF;
  END hold_check;
END dff;
ARCHITECTURE dff_behave OF dff IS
BEGIN
    dff_process: PROCESS(clk)
```

```
    BEGIN
      IF(clk='1') AND (clk'EVENT) THEN
          q<=d;
      END IF;
    END PROCESS dff_process;
END dff_behave
```

在例 5 - 34 中，clk 输入的延时版本将触发保持时间的检查，clk 输入信号延时了相当于保持检查所要求的时间。如果数据输入信号在要求的保持时间内发生了改变，d'LAST_EVENT 将返回一个低于要求保持时间的值。如果数据输入信号与被延时的 clk 信号是同时发生改变，那么由 d'LAST_EVENT 返回的是 0 ns。这是一种特殊情况，它是正确的，必须作特殊处理。

(2) 属性'STABLE

属性'STABLE 用来确定信号对应的有效电平，即它可以在一个指定的时间间隔中，确定信号是否正好发生改变或者没有发生改变。属性返回的值就是信号本身的值，用它可以触发其它的进程。例 5 - 35 就是一个使用属性'STABLE 的例子。

【例 5 - 35】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY pulse_gen IS
  PROT (a: IN STD_LOGIC;
        b: OUT STD_LOGIC);
END pulse_gen;
ARCHITECTURE pulse_gen OF pulse_gen IS
BEGIN
  b<=a'STABLE(10 ns);
END pulse_gen;
```

如图 5 - 3 所示，当波形 a 加到本模块时，即可得到输出波形 b。图中的波形说明，每次信号 a 电平有一次改变，信号 b 的电平将从高电平变成低电平(即由“真”变为“假”)，其

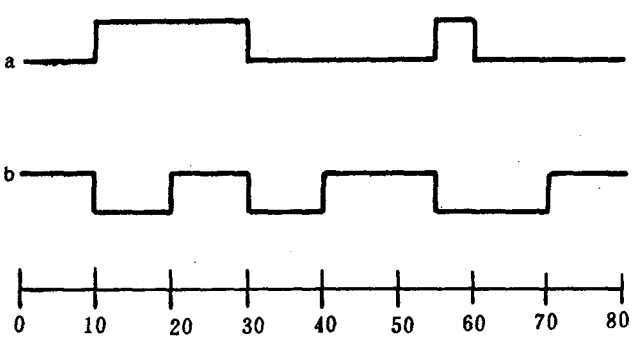


图 5 - 3　使用'STABLE 时的输入输出波形关系

持续时间为 10 ns(该值由属性括号内的时间值确定)。信号 a 在 10 ns 和 30 ns 处各有一次改变，因而对应的信号 b 在 10 ns 和 30 ns 处各有 10 ns 的低电平时间。在 55 ns 处和 60 ns 处信号 a 又各有一次改变。但是，由于改变的间隔小于 10 ns，因此信号 b 从 55 ns 处开始到 70 ns 处结束，将变为低电平。

如果属性'STABLE 后跟括号中的时间值被说明为 0 ns 或者未加说明，那么当信号 a 发生改变时，输出信号 b 在对应的时间位置将产生 Δ 宽度的低电平，如图 5 - 4 所示。

从图 5 - 4 可以看到，当波形 a 发生变化时，其输出波形 b，在对应时刻就会出现一个宽度为 Δ 的负向脉冲。

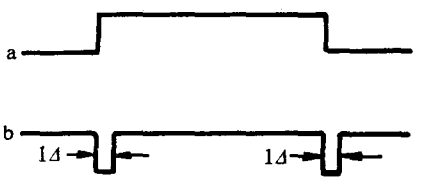


图 5 - 4 'STABLE 属性时间为 0 ns 时的输入输出波形关系

该属性与'EVENT 一样也可以检出信号的上升沿，例如：

```
IF((clk'EVENT)AND(clk='1')AND
        (clk'LAST_VALUE='0')) THEN
          ⋮
END IF;
IF((NOT(clk'STABLE) AND (clk='1') AND
        (clk'LAST_VALUE='0')) THEN
END IF;
              ⋮
```

上述两种情况用 IF 语句都可以检出上升沿，但是，在'EVENT 情况下，在内存有效利用及速度方面将更加有效。这是因为属性'STABLE 需要建立一个额外的信号，这将使其使用更多的内存。另外，不管对新的信号来说，是否需要该值都要求对其进行刷新。

(3) 属性'QUIET

属性'QUIET 具有与'STABLE 相同的功能，但是，属性'QUIET 由所加信号上的电平值的改变所触发(事件除外)。属性'QUIET 将建立一个布尔信号，当所加的信号没有改变，或者在所说明的时间内没有发生事件时，利用该属性可得到一个“真”的结果。

该属性常用于描述较复杂的一些信号值的变化，如开关电平或者器件的值的解析。例如：

【例 5 - 36】

```
ARCHITECTURE teset OF test IS
  TYPE t_int IS (int1, int2, int3, int4, int5);
  SIGNAL int, intsig1, intsig2, intsig3: t_int;
  SIGNAL lock_out: BOOLEAN;
```

```
BEGIN
  int1_proc:PROCESS
  BEGIN
    ⋮
    WAIT ON trigger1; --输出端触发信号
    WAIT UNTIL clk='1';
    IF NOT(lock_out) THEN
      intsig1<=int1;
    END IF;
  END PROCESS int1_proc;
  int2_proc: PROCESS
  BEGIN
    ⋮
    WAIT ON trigger2; --输出端触发信号
    WAIT UNTIL clk='1';
    IF NOT(lock_out) THEN
      intsig2<=int2;
    END IF;
  END PROCESS int2_proc;
  int3_proc: PROCESS
  BEGIN
    ⋮
    WAIT ON trigger3; --输出端触发信号
    WAIT UNTIL clk='1';
    IF NOT(lock_out) THEN
      intsig3<=int3;
    END IF;
  END PROCESS int3_proc;
  int<=intsig1 WHEN NOT(intsig1'QUIET) ELSE
       intsig2 WHEN NOT(intsig2'QUIET) ELSE
       intsig3 WHEN NOT(intsig2'QUIET) ELSE
       int;
  int_handle: PROCESS
  BEGIN
    WAIT ON int'TRANSACTION; --后述讨论
    lock_out<=TRUE;
    WAIT FOR 10 ns;
    CASE int IS
      WHEN int1=>
        ⋮
      WHEN int2=>
        ⋮
      WHEN int3=>
```

```
        ⋮
      WHEN int4=>
        ⋮
      WHEN int5=>
        ⋮
    END CASE;
    lock_out<=FALSE;
  END PROCESS;
END test;
```

例 5 - 36 描述了一个具有优先级的机制，能处理多级的外部中断。进程 int1_proc 优先级最高，进程 int3_proc 优先级最低。无论哪一个进程被触发，相应的中断处理字将放在信号 int 上，并且根据该中断处理字将调用相应的中断过程。

上述模块由 3 个进程组成，这 3 个进程将驱动中断信号 int，而且任何进程都可以调用各自的中断过程。信号 int 不是一个判决函数信号(resolved)，因此不能支持多个驱动器的输出。如果将信号 int 设定成判决函数信号，那么驱动器的序列就不能用来确定优先级。因此，在要求有优先级的情况下，只能采用上述的方法。

在上述的模块中，内部信号 intsig1，intsig2和 intsig3 分别由各进程驱动。这些信号由一个状态信号赋值语句组合起来，在所涉及的驱动器的信号发生改变时，利用所定义的属性'QUIET的赋值语句来确定状态信号的值。信号的改变应在内部信号中进行检测，这是因为，进程通常将赋予相同的值，以使事件仅仅在第一条赋值语句上发生。

优先级的机制是通过状态信号赋值语句来进行控制的。当 intsig1，intsig2 或 intsig3 发生转换时，赋值语句将给信号 int 赋一个相对应的值。如果只有信号 intsig2 发生转换(改变)，那么 intsig2'QUIET 将返回“假”，从而执行状态赋值语句，将 intsig2 的值赋予信号 int。但是，如果 intsig2 和 intsig3 同时发生了信号转换，此时状态赋值语句将进行判断，第一级 WHEN 表达式判别结果将返回一个“假”，此后继续对第二级 WHEN 表达式进行判别。这时由于 intsig2 发生了转换，WHEN 表达式判别结果将返回一个“真”，因此 intsig2 的值将赋予 int。这样，优先级的编排是由状态赋值语句中的 WHEN 语句的前后次序不同来实现的。

(4) 属性'TRANSACTION

在例 5 - 36 中还可以看到，在中断处理的进程中，利用 WAIT 语句中的属性'TRANSACTION 来实现中断处理的实例。属性'TRANSACTIN 将建立一个数据类型为 BIT 的信号，当属性所加的信号每次从“1”或“0”发生改变时，就触发该 BIT 信号翻转。该属性常用于进程调用。

在上述的中断处理进程的实例中，中断处理进程应在信号 int 发生改变时启动。由于同样的中断可以多次发生，所以，如果当一个信号转换发生，而不能在信号 int 上产生一个事件时，那么等待语句 WAIT 就会一直处于等待状态。用属性'TRANSACTION 和 int'TRANSACTION 触发一个事件发生，从而将 WAIT 激活，中断处理程序就被启动。

4. 数据类型类属性

利用该属性可以得到数据类型的一个值。它仅仅是一种类型属性，而且必须使用数值类或函数类属性的值来表示。例如：

t'BASE

用该属性可以得到数据 t 的类型或子类型，它仅仅作为其它属性的前缀来使用，具体如例 5 - 37 所示。

【例 5 - 37】

```
do_nothing: PROCESS(x)
  TYPE color IS (red,blue,green,yellow,brown,black);
  SUBTYPE color_gun IS color RANGE red TO green;
  VARIABLE a: color;
BEGIN
  a:=color_gun'BASE'RIGHT; --a=black
  a:=color'BASE'LEFT; --a=red
  a:=color_gun'BASE'SUCC(green); --a=yellow
END PROCESS do_nothing;
```

在例 5 - 37 的第一条对变量 a 进行赋值的语句中，color_gun'BASE 将返回 color 的数据类型，而后跟的'RIGHT，则将 color 的数据值 black 赋予变量 a。此时变量 a 的数据类型与 color_gun 的相同，且其值为 color 的最后一个值 black。同理对变量 a 的第二条赋值语句，将 color 的 red 的取值赋予变量 a。

在 VHDL 语言中只有一种可供编程人员使用的属性变量，这一点应特别注意。

5. 数据区间类属性

在 VHDL 语言中有两类数据区间类属性，这两类属性仅用于受约束的数组类型数据，并且可返回所选择输入参数的索引区间。这两个属性是：

- a'RANGE[(n)]
- a'REVERSE_RANGE[(n)]

属性'RANGE[(n)]将返回一个由参数 n 值所指出的第 n 个数据区间，而'REVERSE_RANGE 将返回一个次序颠倒的数据区间。

属性'RANGE 和'REVERSE 循环语句的循环次数如例 5 - 38 所示。

【例 5 - 38】

```
FUNCTION vector_to_int(vect: STD_LOGIC_VECTOR)
    RETURN INTEGER IS
  VARIABLE result: INTEGER:=0;
BEGIN
  FOR i IN vect'RANGE LOOP
    result:=result * 2;
    IF Vect(i)='1' THEN
      result:=result+1;
    END IF;
  END LOOP;
  RETURN result;
END vector_to_int;
```

例 5 - 38 是一个将位矢量转换成整数的函数，程序中的循环次数应由输入参数 vect 的位数来确定。在该函数被调用时，输入不能被赋予没有约束的值。这样，属性' RANGE 就

可以用来确定输入矢量的区间。

属性'REVERSE_RANGE 类似于属性'RANGE，所不同的仅仅是返回区间的次序是颠倒的。例如，若属性'RANGE 返回的区间 0 TO 15，那么使用'REVERSE_RANGE 返回的区间将是 15 TO 0。

6. 用户自定义的属性

除了上面在 VHDL 语言中所定义的属性以外，还可以有由用户自定义的属性，用户自定义属性的书写格式为：

ATTRIBUTE 属性名：数据子类型名；

ATTRIBUTE 属性名 OF 目标名：目标集合 IS 公式；

在对要使用的属性进行说明以后，接着就可以对数据类型、信号、变量、实体、构造体、配置、子程序、元件、标号进行具体的描述，例如：

```
ATTRIBUTE max_area: REAL;
ATTRIBUTE max_area OF fifo: ENTITY IS 150.0;

ATTRIBUTE capacitance: cap;
ATTRIBUTE capacitance OF clk, reset: SIGNAL
        IS 20 pF;
```

用户自定义属性的值在仿真中是不能改变的，也不能用于逻辑综合。用户自定的属性主要用于从 VHDL 到逻辑综合及 ASIC 的设计工具、动态解析工具的数据的过渡。

5.3.3 GENERATE 语句

GENERATE 语句用来产生多个相同的结构，它有 FOR－GENERATE 和 IF－GENERATE 两种使用形式，如下所示：

```
标号: FOR 变量 IN 不连续区间 GENERATE
        <并发处理语句>;
END GENERATE [标号名];

标号: IF 条件 GENERATE
        <并发处理语句>;
END GENERATE [标号名];
```

FOR－GENERATE 和 FOR－LOOP 的语句不同，在 FOR－GENERATE 结构中所列举的是并发处理语句。因此，在结构内部的语句不是按书写顺序执行的，而是并发执行的。这样，结构中就不能使用 EXIT 语句和 NEXT 语句。

IF－GENERATE 语句在条件为“真”时才执行结构内部的语句，语句同样是并发处理的。与 IF 语句不同的是该结构中没有 ELSE 项。

该语句的典型应用场合是生成存贮器阵列和寄存器阵列等。另一种应用像在其它语句中那样(如 C 语言)，用于仿真状态编译机。下面举一个利用多个 D 触发器构成移位寄存器的例子。

图 5－5 是一个由 4 个 D 触发器组成的移位寄存器的原理框图，例 5－39 是利用 GENERATE 语句来描述该 4 位移位寄存器的一个程序模块。

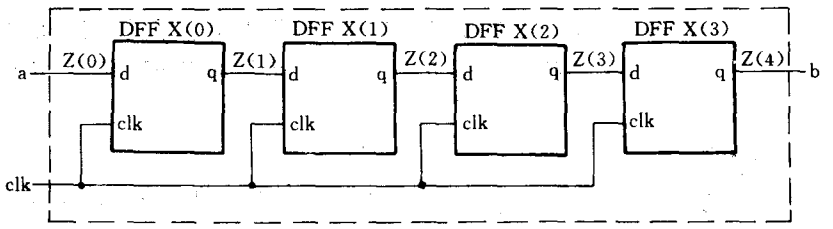


图 5－5　4 位移位寄存器原理框图

【例 5－39】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY shift IS
  PORT(a,clk: IN STD_LOGIC;
           b: OUT STD_LOGIC);
END shift;
ARCHITECTURE gen_shift OF shift IS
  COMPONENT dff
    PORT(d,clk: IN STD_LOGIC;
             q: OUT STD_LOGIC);
  END COMPONENT;
  SIGNAL z: STD_LOGIC_VECTOR(0 TO 4);
BEGIN
  z(0)<=a;
  g1: FOR i IN 0 TO 3 GENERATE
    dffx: dff PORT MAP(z(i), clk, z(i+1));
  END GENERATE;
  b<=z(4);
END gen_shift;
```

例 5－39 是 4 位移位寄存器的行为描述，端口 a 是移位寄存器的输入端，端口 b 为输出端，端口 clk 为时钟输入端。

在构造体 gen_shift 中有两条并发的信号赋值语句和一条 GENERATE 语句。信号赋值语句将内部信号 z 和输入端口 a 和输出端口 b 连接起来。GENERATE 语句(FOR－GENERATE)产生 4 个 D 触发器元件。

在 FOR－GENERATE 语句中，FOR 的作用和一般顺序语句中的 FOR－LOOP 相像，变量 i 不需要事先定义，i 在 GENERATE 语句中是不可见的，而且在 GENERATE 语句内部也是不能赋值的。

为了说明 GENERATE 语句的特点，这里再举一个一般生成 4 位移位寄存器模块的实例，以进行对比参考。

【例 5－40】

```
ARCHITECTURE long_way_shift OF shift IS
```

```
  COMPONENT dff
    PORT(d,clk: IN STD_LOGIC;
             q: OUT STD_LOGIC);
  END COMPONENT;
  SIGNAL z: STD_LOGIC_VECTOR(0 TO 4);
BEGIN
  z(0)<=a;
  dff1: dff PORT MAP(z(0), clk, z(1));
  dff2: dff PORT MAP(z(1), clk, z(2));
  dff3: dff PORT MAP(z(2), clk, z(3));
  dff4: dff PORT MAP(z(3), clk, z(4));
  b<=z(4);
END long_way_shift;
```

从例 5 - 39 和例 5 - 40 比较可以看到，两者的区别仅仅在于，前者用一条 FOR－GENERATE 语句替代了后者的 4 条 PORT MAP 语句，使程序更加简练了，而且改变 i 的取值范围可以描述任意长度的移位寄存器。

从例 5 - 39 可以发现，在移位寄存器的输入端和输出端的信号连接无法用 FOR－GENERATE 语句来实现，只能用两条信号代入语句来完成。也就是说，FOR－GENERATE 语句只能处理规则的构造体。但是，在大多数情况下，电路的两端(输入端和输出端)总是具有不规则性，无法用同一种结构表示。为解决这种不规则电路的统一描述方法，就可以采用 IF－GENERATE 语句。下面仍以一个任意长度的移位寄存器描述模块为例来加以说明。假设移位寄存器的输入信号为 a，输出信号为 b，时钟信号为 clk，共有 len 位。那么该移位寄存器描述的模块如例 5 - 41 所示。

【例 5 - 41】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY shift IS
  GENERIC(len: INTEGER);
  PORT(a,clk: IN STD_LOGIC;
           b: OUT STD_LOGIC);
END shift;
ARCHITECTURE if_shift OF shift IS
  COMPONENT dff
    PORT(d,clk: IN STD_LOGIC;
             q: OUT STD_LOGIC);
  END COMPONENT;
  SIGNAL z: STD_LOGIC_VECTOR(1 TO (len-1));
BEGIN
  g1: FOR i IN 0 TO (len-1) GENERATE
    IF i=0 GENERATE
      dffx: dff PORT MAP(a,clk,z(i+1));
```

```
    END GENERATE;
    IF i=(len-1) GENERATE
      dffx: PORT MAP (z(i), clk, b);
    END GENERATE;
    IF(i/=0) AND (i/=(len-1)) GENERATE
      dffx: PORT MAP(z(i),clk,z(i+1));
    END GENERATE;
  END GENERATE;
END if_shift;
```

在例 5-41 中使用了一个可配置长度的移位寄存器。len 是移位寄存器长度，也是信号数组 z 的长度，它应由 GENERIC 语句事先说明。

在 FOR-GENERATE 语句结构中，IF-GENERATE 语句首先检查 i=0，或者 i=len-1。也就是说所产生的 D 触发器是移位寄存器最前面一级还是最后面一级。因为在程序中都用 PORT MAP 语句来生成 D 触发器。如果是第一级，那么 POR MAP 语句中的输入应用信号 a 来代入；而如果是最后一级，那么 PORT MAP 语句的输出信号应用 b 来取代。这样引入条件语句后，用 PORT MAP 语句就可以生成任意长度的移位寄存器。

下面再来举一个根据不同状态控制生成的语句，如例 5-42 所示。

【例 5-42】

```
PACKAGE gen_cond IS
  TYPE t_checks IS(onn,off);
END gen_cond;
LIBRARY IEEE;
USE WORK.gen_cond.ALL;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY dff IS
  GENERIC(timing_checks: t_checks;
        setup,qrise,qfall,qbrise, qbfall: TIME);
  PORT(din,clk: IS STD_LOGIC;
          q,qb: OUT STD_LOGIC);
  END dff;
ARCHITECTURE condition OF dff IS
BEGIN
  G1: IF (timing_checks=onn) GENERATE
    ASSERT(din'LAST_EVENT>setup)
      REPROT "SETUP VIOLATION"
      SEVERITY ERROR;
  END GENERATE;
  PROCESS(clk)
    VARIABLE int_qb: STD_LOGIC;
  BEGIN
    IF(clk='1') AND (clk'EVENT) AND
       (clk'LAST_VALUE='0') THEN
```

```
        int_qb:=NOT din;
        q<=din AFTER
            f_delay(din,qrise,qfall);
        qb<=int_qb AFTER
            f_delay(int_qb,qbrise,qbfall);
    END IF;
  END PROCESS;
END condition;
```

在例 5-42 中 D 触发器元件用 IF—GENERATE 语句来建立其模型，并用它来控制是否需要生成定时检查语句。当状态值为 onn 时，生成语句将生成一个并发的赋值语句；如果状态为 off 则不生成赋值语句。这种功能可以仿真状态编译机，正如在某些编程语言中(如 C 语言和 PASCAL 语言中)所见到的那样。

5.3.4 TEXTIO

在 VHDL 语言中提供了一个预先定义的包集合是文本输入输出包集合(TEXTIO)，在该 TEXTIO 中包含有对文本文件进行读写的过程和函数。这些文本文件都是 ASCII 码文件，其格式可根据编程人员需要设定(VHDL 语言对文件格式不作任何限制，但是主机对此往往有一定限制)。TEXTIO 按行对文件进行处理，一行为一个字符串，并以回车、换行符作为行结束符。TEXTIO 中提供了读、写一行的过程及检查文件结束的函数。

TEXTIO 也对用于处理文本文件的数据类型作了具体说明。数据类型 line(行)是读写文本文件时要用的，line 的结构是 TEXTIO 对文件进行操作的基本单位。例如，对文件进行读操作时，首先读一行字符，并将它放到 line 数据类型的结构中，而后再按字段进行处理。与此相反，当要写一个文件时，首先在行数据暂存区按字段建立 line 结构，然后再将 line 的数据写到文件中去。

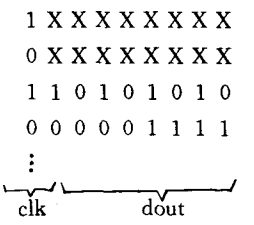


图 5-6 文件 testp.in 中数据排列示意图

下面讲述在 TEXTIO 中读、写文件的语句的书写格式。

(1) 从文件中读一行

READLINE(文件变量，行变量)；

READLINE 用于从指定的文件中读一行的语句。例如，在一个文件中数据按行排列的情况如图 5-6 所示，利用 READLINE 就可以读一行数据，如例 5-43 所示。

【例 5-43】

```
SIGNAL clk: STD_LOGIC;
SIGNAL lin:STD_LOGIC_VECTOR(7 DOWNTO 0);
VARIABLE li: LINE;
FILE invector: TEXT IS IN "testp.in";
READLINE(invector,li); -- 读一行数据
```

例 5-43 表明，利用 READLINE 语句，从文件变量 invector 所指定的文件中(testp.in)读一行数据，将它放到 li 的行变量中。

(2) 从一行中读一个数据

READ(行变量,数据变量);- - 从一行中读取

- - 一个数据

利用 READ 语句可以从一行中取出一个字符，放到所指定的数据变量(信号)中。

例如，在上例中要从某一行中取出第一位数据的值，并赋予信号 clk 时，即可使用如下的读语句。

READ(li,clk); - - 取 li 行的第一位数据，并赋给 clk

READ(li,din); - - 取后续的 8 位数据，并赋给 din

(3) 写一行到输出文件

WRITELINE(文件变量，行变量);

该行写语句与行读语句相反，将行变量中存放的一行数据写到文件变量所指定的文件中去。

(4) 写一个数据至行

WRITE(行变量，数据变量);

该写语句将一个数据写到某一行中。在某些 CAD 公司的 VHDL 语言的版本中，对该写语句有些扩充。例如按十六进制值写时，写语句应以 H 为前缀：HWRITE；如按八进制值写时，则应冠以 O 前缀：OWRITE 等。另外，写语句的格式也有相应变化：

WRITE(行变量，数据变量，起始位置，字符数);

在这里，起始位置可以有两种选择：

LEFT——从行的最左边对齐；

RIGHT——从行的最右边对齐。

而字符个数则表示要写的起始位置，例如：

WRITE(lo，dout，left，9);

则表示 dout 的数据从输出行 lo 的左边对齐输出9个字符。例如，某一个输出文件 testp.out 如图 5 - 7 所示。对文件 testp.out 的写操作的语句如例 5 - 44 所示。

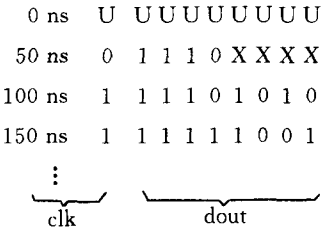


图 5 - 7 文件 testp.out 数据排列示意图

【例 5 - 44】

```
SLGNAL dout: STD_LOGIC_VECTOR(7 DOWNTO 0);
VARIABLE lo: LINE;
file outvector: TEXT IS OUT "testp.out";
WRITE(lo, now, right, 6); - - 写一个数据
WRITE(lo, e, left, 1); - - 写一个数据
```

```
WRITE(lo, dout, left, 9); --写一个数据
WRITELINE(outvector, lo); --写一行数据
```

(5) 文件结束检查

```
ENDFILE(文件变量);
```

该语句检查文件是否结束，如果检出文件结束标志，则返回“真”值，否则返回“假”值。

TEXTIO 常用于测试图的输入和输出。在使用 TEXTIO 的包集合时，首先要进行必要的说明，例如：

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

在 VHDL 语言的标准格式中，TEXTIO 只能使用“BIT”和“BIT_VECTOR”两种数据类型。如果要使用“STD_LOGIC”和“STD_LOGIC_VECTOR”，就要调用“STD_LOGIC_TEXTIO”，即

```
USE IEEE.STD_LOGIC_TEXTIO.ALL;
```

正如在例 5 - 43 中所示，TEXTIO 在读写文件时先要对文件进行说明，其说明格式如下：

FILE 文件变量：TEXT IS 方向“文件名”;

在这里，方向是指读还是写，读为 IN，写为 OUT。

一个完整的 TEXTIO 的描述实例如例 5 - 45 所示。

【例 5 - 45】

```
LIBRARY IEEE.STD;
USE STD.TEXTIO.ALL;
USE IEEE STD_LOGIC_1164.ALL;
USE IEEE STD_LOGIC_TEXTIO.ALL;
ENTITY simtop IS
END simtop;
ARCHITECTURE sim OF simtop IS
COMPONENT cn8 PORT(clk,reset:IN STD_LOGIC;
  count: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END COMPONENT;
FILE inv: TEXT IS IN "bar.in";
FILE outv: TEXT IS OUT "bar.out";
SIGNAL clkin,resetin: STD_LOGIC;
SIGNAL count: STD_LOGIC_VECTOR(7 DOWNTO 0);
CONSTANT clk_cycle: TIME:=10 ns;
CONSTANT stb: TIME:=2 ns;
BEGIN
  U1: cn8 PORT MAP(clk=>clkin,reset=>resetin,count=>count);
  PROCESS
  VARIBLE li,lo: LINE;
  VARIBLE clk,reset: STD_LOGIC;
  BEGIN
```

```
    READLINE(inv,li);
    READ(li,clk);
    READ(li,reset);
    clkin<=clk;
    resetin<=reset;
    WAIT FOR clk_cycle_stb;
    WRITE(lo,now,left,8);
    HWRITE(lo,count,right,3);
    WRITELINE(outv,lo);
    WAIT FOR stb;
    if(ENDFILE(inv)) THEN
      WAIT;
    END IF;
  END PROCESS;
END sim;
```

第 6 章

数值系统的状态模型

在设计数值系统时，必须要事先知道系统所规定的几种逻辑状态。在以往的数字电路的设计中，二态逻辑系统和三态逻辑系统已为一般的工程设计人员所熟知。但是，随着大规模集成电路技术的发展，在数值系统设计时往往需要用到混合技术，将 ECL、TTL、CMOS、MOS 等不同的器件连接起来。这些器件之间的逻辑电平是不一致的，为了描述这些器件的逻辑电平，前面已经提到的用二态和三态来描述数值系统的逻辑电平显然是不够了，而是需要增加某些状态。另外，建立双向开关电平及处理未知状态等也需要引入其它状态。

下面概略介绍一下随着硬件设计技术和仿真技术而发展起来的四十六态数值系统。

6.1 二态数值系统

在对数字系统进行初级仿真时，一般采用二态数值系统，逻辑“1”(或者“真”)和逻辑“0”(或者“假”)就是系统的两种状态。信号的状态只可能取二者之一。在 VHDL 语言中，通常用 BIT 数据类型来描述这两种状态。例如：

```
TYPE BIT IS('0','1');
```

最简单的数值系统是一个信号源的系统，用二态数值系统就能很好地描述这样的系统。例如，由一个反相器构成的数值系统，当输入为“0”时，其输出为“1”；而当输入为“1”时，其输出为“0”。系统的输入和输出，在任何时候其值只能取此两种状态之一。

在数字电路和计算机原理的有关书籍中经常可以看到这样的一个概念，即总线竞争或者称总线冲突。在某一条总线上，如果有多个信号源，以相同的强度值对它进行驱动时，就会产生总线竞争(或总线冲突)，此时总线上的信号电平可能是一个不能具体确定的逻辑电平。这样的系统，如果要想用二态数值系统来描述就不行了，因为二态中的“0”和“1”都无法正确地描述其输出。在图

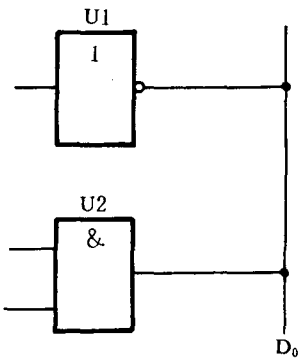


图 6-1 总线冲突电路实例

6－1 中，如果某一条数据总线 D_0 由一块反相器 U1 和一块“与”门 U2 所驱动，U1 的输出为“0”，而 U2 的输出为“1”。在一条总线上出现了两个不同的逻辑电平，这样 D_0 数据线上到底是“1”还是“0”就很难确定了，也就是说出现了不确定的值“X”。这种状态是一种错误的状态，仅仅利用二态数值系统就不能表示信号的输出错误状态。

6.2 三态数值系统

为了避免在二态数值系统中所发生的问题，人们认识到，在二态数值系统的基础上需要再加一个新的状态，这就是未知值状态。这种状态在 VHDL 语言中通常用字符“X”来表示。未知状态可以取值为“1”，也可以取值为“0”，但是当前到底取值是“1”还是“0”是不确定的。对三态数值系统状态，可以用数据类型定义语句来描述：

TYPE threestate IS ('X', '0', '1');

未知状态值可以在不同情况下表示不同的行为。例如，用未知状态值可以表示 0～5 V 之间的电压值；另一种情况下，也可以表示“1”和“0”的值。在数值系统中，未知状态将表示“1”或者“0”，但是具体是何值却不能确定。

在系统设计的仿真中，引入“X”值有许多方便之处。首先，用“X”值可以表示信号的初始状态值，在系统启动时所有信号都被置为“X”值，此后这个值可以被电路元件的后继状态所改写。在系统开始仿真时，系统中的每一个信号均将赋以“X”值。在外部输入信号值加到电路的输入端以后，通过电路的信号传递，就会改写初始启动值“X”。

在仿真时产生“X”值的另一个原因是总线冲突(总线竞争)，也就是前面所述的，有多种输出信号线连接在一起，且它们的逻辑值是相反的，如图 6－1 所示。此时，电路的输出值将为“X”。将两个输出信号连接在一起，这可能是有意的，但也可能是无意的设计错误。不管哪一种情况，仿真器必须预测出正确的输出值。在图 6－1 所示的例子中，如果不给出两个信号的有关各自强度的信息，那么仿真器就不能确定输出值到底应该取值为“0”还是为“1”。此时，其输出值只能用“X”值来表示。有关输出信号值的强度问题，在后面将会做更详细的解释。这里，为便于读者理解“X”值的状态，先对信号值的强度作一概略介绍。在第 4 章的例 4－3 中，介绍了一种判决函数表。在该表中定义了 9 种逻辑状态，其中：

“0”——强逻辑低电平；

“1”——强逻辑高电平；

“L”——弱逻辑低电平；

“H”——弱逻辑高电平。

当两个强度相同，而逻辑不同的信号同时出现在一个输出端时，其输出端的值是不确定的。例如“0”和“1”及“L”和“H”同时出现在信号的输出端时，输出端的取值应为“X”。如果不同强度的信号出现在输出端时，输出端的最终取值应由强信号逻辑状态确定。例如，当“1”和“L”出现在输出端时，输出端取值为“1”；当“0”和“H”出现在输出端时，输出端取值为“0”，其它情况依次类推。

由此可见，在系统或电路仿真时，给出信号强度的信息是至关重要的。

6.3 四态数值系统

在当前的计算机系统中常常要用到双向数据总线，数据总线驱动器的输出需要有一个特殊的状态，即高阻状态。这是无法用二态数值系统和三态数值系统进行正确描述的。利用这个高阻状态，可以使总线被多个设备所共享，并且可以方便地实现数据总线的双向操作。高阻状态通常用集电极开路门来实现，为表示这种状态，需要引入另一种状态，通常称为“Z”状态。这样就形成了四态数值系统，在 VHDL 语言中常用如下数据类型来描述这 4 种状态：

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

“Z”状态是三态驱动器的一种输出状态，与一般的门电路不同，它除了具有输入和输出端之外，还有一个允许端，如图 6 - 2 所示的 en 端。

当 en 端为“0”(低电平)时，无论输入端 a 的信号值是“0”还是“1”，其输出端 b 均呈现高阻状态；而当 en 端为“1”(高电平)时，输出端 b 的信号值就随输入端 a 的信号值变化而变化。当 a＝“1”时，b 就为“0”；当 a＝“0”时，b 就为“1”。

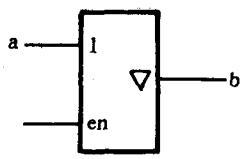


图 6 - 2　三态驱动器

高阻状态“Z”的引入，解决了多个信号源驱动一条信号线以及信号线的双向驱动等问题，图 6 - 3 是一个利用三态门实现总线的双向操作的实例。

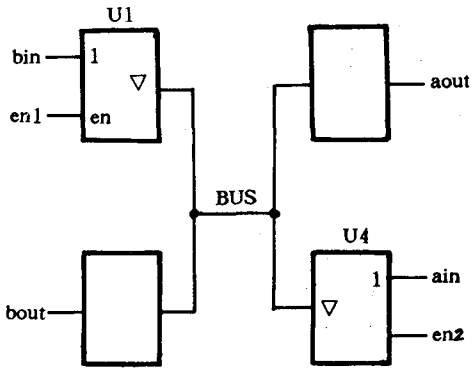


图 6 - 3　利用三态门实现总线双向操作实例

现假设 U1 的输入 bin＝'1'，U4 的输入 ain＝'0'。当 en1＝'1'时，U1 的输出值“1”将加到总线上。如果此时 en2＝'0'，则 U4 的输出值“Z”也同时加到总线上，这样在总线上就存在两个逻辑值“1”和“Z”。由于“1”和“0”的强度都大于“Z”，故总线上的最终状态值应为“1”。总线上数据的流向便为自左至右。

如果现在 en1＝'0'，en2＝'1'，那么 U4 就将输出“0”值加到总线上；而 U1 则将输出“Z”值加到总线上。这样总线上同样存在两个逻辑值“0”和“Z”。与上述理由相同，此时总线上的最终状态值为“0”。总线上的数据流向便为自右至左。

如果某一时刻 en1 和 en2 同时为“1”，也就是说，U1 输出的“1”值和 U4 输出的“0”值将同时加到总线上。由于该两个值的强度相同，故最终总线上的状态应为“X”。在一般情况下，这种状态是不希望出现的，在系统正常工作时，通常要求驱动总线的三态门在某一时刻只允许有一个被选通。

表 6－1 示出了总线上两种不同状态施加时，其最终取值的结果。这种列表与例 4－3 中所述的判决函数是一致的。利用查表方法可以验证，上述所列举的 3 种情况，其结果是完全一致的，即“1”和“Z”将取值“1”，“0”和“Z”将取值“0”，“1”和“0”将取值“X”。

表 6－1　总线状态值关系表

U2 输出值 \ U1 输出值	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	Z

表 6－1 仅仅示出了两个三态驱动器驱动总线时，总线取值情况。如果有多个三态驱动器驱动总线时，总线上的状态值又如何来确定呢？实际上也很简单，仍然可以使用该表。经查表得到两个驱动器驱动总线时的取值。然后由这个总线取值和下一个总线驱动器的输出值，再经查表可得到下一个总线取值。如此循环查表，相当于一个迭代过程，最后即可得到由多个驱动器驱动时，总线的最终取值。

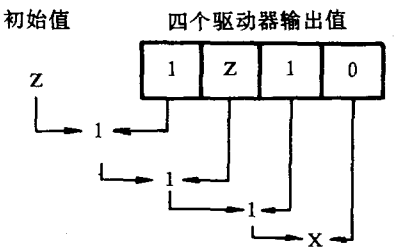


图 6－4　4 个驱动器时总线取值的查表过程的示意图

例如，有 4 个驱动器同时驱动一条总线，总线的初始状态为“Z”，4 个驱动器的输出分别为“1”，“Z”，“1”，“0”。那么其总线的最终取值应为“X”，其具体的查表过程如图 6－4 所示。从图 6－4 可以看出，在这种情况下总线上的取值应为“X”。

6.4　九态数值系统

四态数值系统能较精确地描述 TTL 器件的工作过程。但是，随着 MOS 技术的不断发展，四态数值系统已不能正确地反映系统的实际工作过程了。在 MOS 电路中“1”和“0”具有不同的强度。在 NMOS 电路中，“0”值的强度比“1”值强；而在 PMOS 电路中“1”值的强度又比“0”值强。另外，在 NMOS 和 PMOS 电路中，如果某一个节点处于三态时，电荷将被贮存，节点将维持原来的逻辑值(当然，这只是在某一工作周期中有效)。

为了表示数值系统的所有状态，人们开发了九态数值系统。该数值系统由 3 种强度和 3 种逻辑值组成。3 种强度分别为高阻“Z”，电阻“R”及强强度“F”。9 个状态值的对应关系如表 6－2 所示。在 VHDL 语言中常用如下数据类型来描述九态数值系统：

表 6-2 九态数值关系表

强度值 \ 逻辑值	0	1	X
Z	Z0	Z1	ZX
R	R0	R1	RX
F	F0	F1	FX

TYPE ninestate IS(Z0、Z1、ZX、R0、R1、RX、F0、F1、FX);

上述定义的状态将表示如下的系统状态：

Z0——高阻强度的逻辑“0”；

Z1——高阻强度的逻辑“1”；

ZX——高阻强度的逻辑“X”；

R0——电阻强度的逻辑“0”；

R1——电阻强度的逻辑“1”；

RX——电阻强度的逻辑“X”；

F0——强强度的逻辑“0”；

F1——强强度的逻辑“1”；

FX——强强度的逻辑“X”。

在这里，强强度的 F0、F1、FX 很像在三态数值系统中所讨论的“0”，“1”，“X”3 种状态，所不同的仅仅是现在加了一个强度值。在四态数值系统中所讨论的状态值“Z”，现在被扩展成 3 种状态，“Z”作为强度值来进行表示。这样，Z0，Z1，ZX 分别表示电荷所存贮的逻辑值。第三种电阻强度值用来处理 NMOS 的“1”值和 PMOS 的“0”值。

强强度在 3 种强度描述中表示强度最强的一个强度，它相当于供电电源提供的强度。例如，5 V电源的电平可以用 F1 来表示；电源地可以用 F0 来表示。电阻强度，其强度值低于强强度的值，它可以由强强度逻辑电平经过一个电阻后得到。如图 6-5 所示，电阻的左端若加一个强强度逻辑电平，那么在电阻的另一端即可得到电阻强度的逻辑电平。例如，5 V电源是强强度逻辑电平 F1，如果在电路中加一个上拉电阻，那么在上拉电阻的另一端即可得到电阻强度的逻辑电平 R1。高阻强度是 3 种强度中最弱的一种强度。它所描述的是，NMOS，PMOS，CMOS 器件的门电路断开时，在分布电容上所存贮的电荷数量。例如，图 6-6 就是一个电荷存贮电路的实例。图中 en=0 时，门被断开，信号 b 失去了驱动源，存贮于电容上的电荷就加在 U2 的输入端。这个存贮电荷的逻辑值将一直保持，直到下一个时间周期再对信号 b 进行驱动为止。此时信号 b 就处于高阻态，至于是 Z0 还是 Z1，则取决于开关断开时的最后逻辑电平是“0”还是“1”。

图 6-5 F0 和 R0 的关系

下面再来看一下用于构成 NMOS 和 PMOS 电路的电阻强度。图 6-7 所示是一个 NMOS 的反相器电路。它由两个晶体管器件组成，分别为 Q1 和 Q2。Q2 是一个标准的增强型 NMOS 开关器件。Q1 是一个耗尽型器件，它的作用像一个电阻，电压经过它将产生压降。

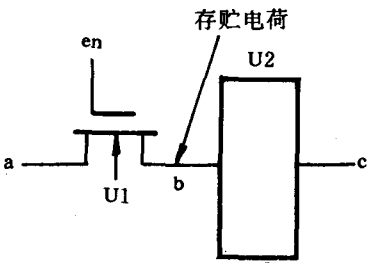


图 6-6 高阻态的电荷存贮电路实例

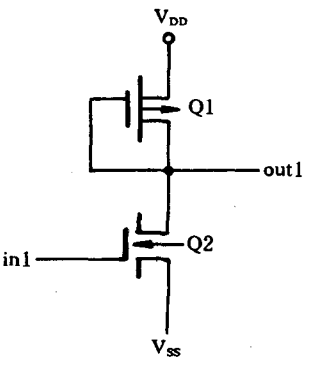


图 6-7 NMOS 的反相电路

当输入 in1 加“1”电平时，Q2 将导通，信号 out1 将被拉至地电平(F0)。此时虽然 Q1 也导通，但是呈现较高的阻抗。该反相器的输出将变成 F0。当输入 in1 加“0”电平时，Q2 将不导通，并呈现很高的阻抗。此时，Q1 呈现中等大小的阻抗，因此输出 out1 由 Q1 驱动，使其呈现“1”电平，它的输出强度不是强强度，而是 R1 强度。在这里 Q1 可以看成是一个带有中等大小阻抗值的电阻。这样，该反相器的真值表就如表 6-3 所示。

表 6-3 NMOS 反相器真值表

输入 in1	输出 out1
0	R1
1	F0
X	FX

在表 6-3 中，为什么在输入 in1 为“X”时，输出 out1 被置为“FX”值呢？这可以从两方面来考虑：如果输入“X”值是“0”，则输出值为 R1；如果输入值为“1”，则输出值为“F0”。从最坏情况来考虑，输入为“X”值，输出也将是“X”值，而且从强度上来考虑，强强度也是一种最坏的情况。因此，在这里用“FX”来表示“R1”和“F0”的两种不确定状态。

现在再来看一下如图 6-8 所示的 NMOS 开关，用作闸门的电路。正如前面所讨论的那样，U2 的作用像一个开关。当 en 为“1”时，信号 b 的值通过开关加到信号 C 上。当 en 为“0”时，信号 C 将存贮前一个逻辑的电荷。该电荷将存贮在门的结及 U3 和信号 C 的线电容中。由该开关实现的真值表如表 6-4 所示。

表 6-4 NMOS 开关真值表

门控值	输 出 值
0	前一个状态值+Z 强度
1	输出=输入

X	IF(逻辑值=前一个状态逻辑值)
	THEN
	输出值=逻辑值+输入强度
	ELSE
	输出=X+输入强度

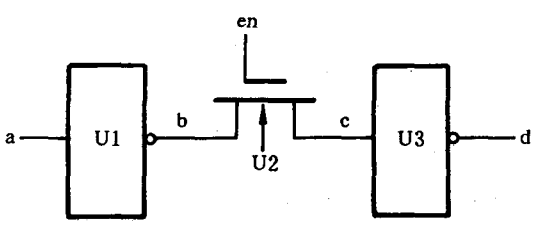


图 6-8 NMOS 开关用作闸门电路

门控值为“X”这种情况应特别注意，在九态数值系统中，没有更好的值来表示确切的输出状态，假设开关所处状态如图 6－9 所示。从图中可以看到，源极输入为 F0 值，漏极存贮着“Z1”值；门控制为“R0”值。这是一个稳定状态，没有任何状态值发生变化。如果现在门控值变成“X”，那么门将输出什么新的值？在 U2 的控制端加一个“RX”值，那么根据“RX”值实际上是“R0”还是“R1”，可能有两种输出值。

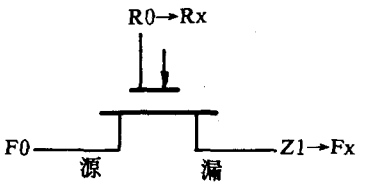


图 6－9　开关的一种状态实例

如果“RX”值是“R0”，U2 将继续输出“Z1”值；如果“RX”值是“R1”，那么 U2 将输出 F0。但是在一般情况下，当编程人员不可能预知是什么强度和什么逻辑值时，通常按最坏情况考虑，即按强强度和“X”逻辑值来处理。

6.5　十二态数值系统

输出“X”值，存在着通过电路扩散或传递的问题，这会引起非常棘手的结果。但是，在大多数情况下，采取某些措施，这种结果是可以避免的。第一种办法是在数值系统中再另加强度种类，即加一个强度 U。这样，数值系统就变成十二态，如表 6－5 所示。

表 6－5　十二态数值状态表

强度＼逻辑值	0	1	X
Z	Z0	Z1	ZX
R	R0	R1	RX
F	F0	F1	FX
U	U0	U1	UX

U 表示一个未知强度，它可以表示 R，或 Z，也可以表示 F 强度。正像“X”可以表示为“1”或“0”一样。U 通常用来表示开关门控值为“X”时的输出强度。

十二态数值系统的状态，在 VHDL 语言中的数据类型是这样定义的：

```
TYPE twelvestate IS (Z0、Z1、ZX
                     R0、R1、RX
                     F0、F1、FX
                     U0、U1、UX);
```

能产生 U 强度值的电路实例，如图 6－10 所示。

如图所示，三态缓冲器输入端加 F0 值，输出端为 Z0。假设该三态缓冲器是用 TTL 工艺制造的器件，在门控值为“1”(允许)时其输出强度为强强度 F。现在开关的门控值从 F0 向 FX 转换，看输出如何变化。

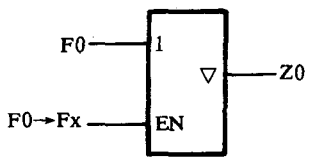


图 6－10　产生 U 强度的电路实例

如果开关的门控值为 F0，则其输出将为 Z0，如果开关的门控值为 F1，则其输出为 F0。因此，当门控值

为“X”时，其可能的输出值为{F0，Z0}。由此可见，该三态门的输出逻辑值始终为“0”，但是其输出强度值的取值可以是F，也可以是Z。为了表示这种状态，现重新定义一个强度值U。在上述情况下，三态缓冲器的输出为U0。

十二态数值系统仍有许多不能表示系统正常工作的状态，例如初始化状态和错误状态。在用“与非”门构成CMOS的RS触发器时，任何时候电路都不会进行初始化。为进行初始化，需在电路中对全局信号进行初始化，但是这是一个非常冗长的处理，因此很少采用。有一个非常棘手的实例如图6-11所示。

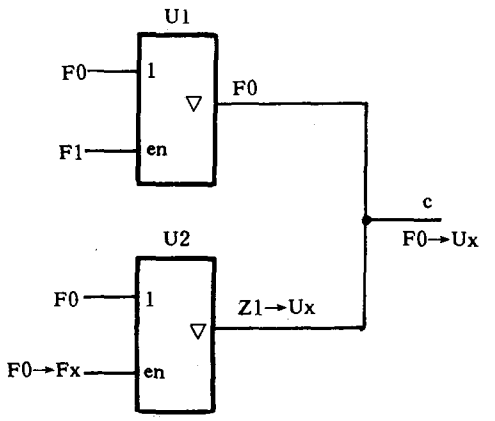


图6-11　一个非常棘手的实例

在这个实例中，由两个三态缓冲器去驱动一个信号。在正常情况下，两个驱动器不会同时去驱动一个信号。但是，在初始状态期间有可能出现图中所示的情况。

开始时，由于U1所有的输入值是知道的，因此U1加给信号c的值将是F0。而U2所有的输入值不是都知道的，故其输出值为UX。

调用判决函数可以分析出信号c的值。如前所述，U1加给信号c的值是F0，而U2加给信号c的值为UX。对这两个值进行判决，可得到信号c的最终值应为FX。这当然不是一个所希望的结果，但是根据现在数值系统所携带的信息变量，这是可能做到的最好结果。

在电路中FX不是一个正确的值。当两个三态门各自的输入都是F0时，无论U2门控输入是“0”还是“1”，都无关紧要，信号c上的正确输出值应为F0。

如果这种状态仅仅发生在系统初始化期间，那么为什么还要讨论它呢？这是因为，一个初始化期间不确定的信号可以导致其它信号的不确定，而其它信号的不确定又可以导致另一些相关信号的不确定，如此循环扩散，其最终结果是不能对系统进行相应的初始化。这就是在4.2节中所提到的“X”状态的传递。

为了解决上述问题，必须定义更多能够表示每种信号状态的信息，这就是下节要介绍的四十六态数值系统。

6.6 四十六态数值系统

在四十六态数值系统中，每个信号值将用“区间标识”的方法来表示，这样使得人们在解析时可以使用更多的当前和预估的信息。该方法用非常紧凑和简单的手段来表示信号取值的范围，而这些范围就是系统的值。例如，现在状态值为F0的信号能够用F0到F0来表示，其它常用的范围或区间从图6-12中可以查到。

应注意，跨越Z0到Z1区间的范围被称为ZX。该状态值的取值可以是Z0，也可以是Z1，只要在这两者之间就行。同理，WX，RX和FX也是这种含义。

在本章前面所叙述的是常用的基本状态值变量。可以看到，大多数状态值是非常雷同的。在本节将再引出新的状态值W0,W1和D。W是另一种强度值，它介于高阻和电阻强度之间，被称为弱电阻强度。这种强度通常用于存贮器和弱上拉电阻等的建模。

状态值D是一个新的概念，它表示该结点没有电容，且不能存贮电荷的值。从另一方面来说，D值表示网络被切断。在STD_LOGIC包集合中对基本的四十六状态值的类型、判决函数等都作了详细说明，读者如有兴趣可参阅有关书刊和手册。

现在再来看一下四十六态数值系统最常见的一些值，下面仍以图6-11两个三态门的输出连接在一起的情况为例作一说明。

U1的工作状态如前所述，输出给信号c的值为F0，而U2仍如前所述，处于不确定的状态。在“区间标识”方法中，结点的大多数信息将传递给判决函数。用“区间标识”法可以用值的范围来表示U2的输出值。如果U2的门控值是“0”，其输出值将维持Z1。当然，如果门控值为“1”，那么U2输出将为F0。

在这种状态下，U2的输出值可用F0到Z1的区间来表示。由图6-12所示，这个状态的名称为FZX。第一个字母F表示逻辑“0”的强度值；第二个字符Z表示逻辑“1”的强度值；字符X表示跨越“0”值和“1”值的范围。

FZX状态将向判决函数传递两个信息，一个是强度最强的逻辑“0”信息，一个是强度为Z的逻辑“0”信息。根据这些信息判决函数将把这两个分量看作是影响输出信号的因素。

在一种情况下，U1输出分量为F0，U2输出分量为Z1。F0强度比Z1的强，故输出最终值为F0。在另一种情况下，U1和U2输出分量都为F0，故输出最终值为F0。这样，判决函数将对这种情况进行判决：信号c最终所处的状态应为F0。这种处理正是人们所希望的，它将制止不需要的“X”值的传递。

“区间标识”的实际价值就在于，它为判决函数提供了一个传送结点状态信息的方法，并可灵活地处理未知状态。四十六态数值系统具有9个基本值，当所有可能的分量都加起来，结果状态值为：

$$9+8+7+6+5+4+3+2+1=45$$

那么46个值是从何而来呢？还有一个值U，它是永远不可能被赋值的，只表示信号的未初始化的状态。这样整个数值系统就变为46个状态值了。

有了46个状态，几乎可以表示所有的逻辑状态了。但是，大多数的设计人员都不想使用这种麻烦的四十六态的设计。数值系统也支持根据实际所需的子集，而忽略某些不同的状态，数值系统现在可支持5种不同类型的工艺技术。

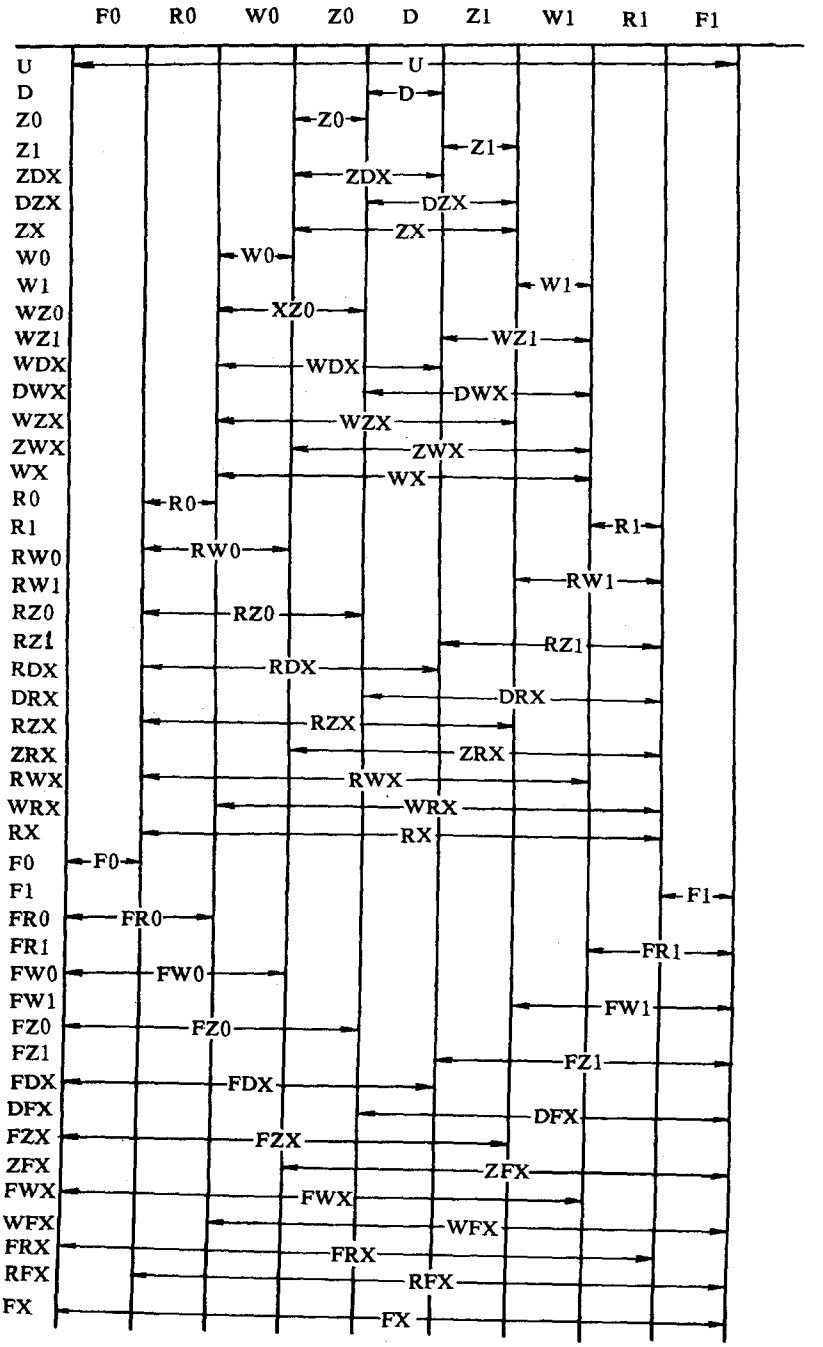


图 6-12　四十六态数值系统的状态值规定

数值系统所支持的不同的工艺技术是 TTL，CMOS，NMOS，ECL，TTLOC。在 TTL 工艺技术中，逻辑值“1”用 F1 表示；逻辑值“0”用 F0 表示。其它几种工艺技术的逻辑值表示如表 6－6 所示。

表 6－6　各工艺技术的逻辑值表示

工艺技术类 / 逻辑值	TTL	ECL	NMOS	CMOS	TTLOC
0	F0	R0	F0	F0	F0
1	F1	F1	R1	F1	ZX
X	FX	RFX	FRX	FX	FZX

在本章中，对数值系统、逻辑值和状态的强度等基本概念作了简要的介绍。在实际所使用的系统中，不同厂家对系统状态的定义及符号的使用都有所不同。这里以 STD_LOGIC_1164 中定义的逻辑状态系统为例加以说明。

在 PACKAGE STD_LOGIC_1164 中，逻辑状态是这样定义的：

```
TYPE STD_ULOGIC IS ('U'——初始状态
                    'X'——强强度不确定状态
                    '0'——强强度 0
                    '1'——强强度 1
                    'Z'——高阻
                    'W'——弱不确定
                    'L'——弱强度 0
                    'H'——弱强度 1
                    '—'——任意项
                    );
```

在上述的 9 个状态中，除'—'外，其它状态的概念和前面所介绍的是完全一致的，只是所使用的符号略有差别而已。'--'状态是任意项，表示系统不可能出现的状态。

第 7 章

基本逻辑电路设计

在前面几章中，对 VHDL 语言的语句、语法及利用 VHDL 语言设计逻辑电路的基本方法作了详细介绍。为了使读者能深入理解使用 VHDL 语言设计逻辑电路的具体步骤和方法，在本章以常用的基本逻辑电路设计为例，再次对其进行详细介绍，以使读者初步掌握用 VHDL 语言描述基本逻辑电路的方法。

7.1 组合逻辑电路设计

在本节所要叙述的组合逻辑电路有简单门电路、选择器、译码器、三态门等。下面逐一地对它们作一介绍。

7.1.1 简单门电路

简单门电路包括 2 输入“与非”门、集电极开路的 2 输入“与非”门、2 输入“或非”门、反相器、集电极开路的反相器、3 输入“与门”、3 输入“与非”门、2 输入“或门”和 2 输入“异或”门等，它们是构成所有逻辑电路的基本电路。

1. 2 输入“与非”门电路

2 输入“与非”门电路的逻辑表达式为：

$$y=\overline{(a \wedge b)}$$

其逻辑电路图如图 7-1 所示。

利用 VHDL 语言描述 2 输入“与非”门有多种形式，现举两个例子加以说明。

【例 7-1】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY nand2 IS
PORT (a,b:IN STD_LOGIC;
      y:OUT STD_LOGIC);
END nand2;
ARCHITECTURE nand2_1 OF nand2 IS
BEGIN
```

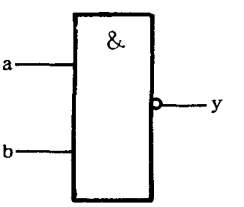


图 7-1 2 输入“与非”门电路

```
    y<=a NAND b;
  END nand2_1;
```

【例 7-2】

```
  LIBRARY IEEE;
  USE IEEE.STD_LOGIC_1164.ALL;
  ENTITY nand2 IS
  PORT (a,b:IN STD_LOGIC;
        y:OUT STD_LOGIC);
  END nand2;
  ARCHITECTURE nand2_2 OF nand2 IS
  BEGIN
    t1:
    PROCESS (a,b)
      VARIABLE comb:STD_LOGIC_VECTOR (1 DOWNTO 0);
    BEGIN
      comb := a & b;
      CASE comb IS
        WHEN "00"=>y<='1';
        WHEN "01"=>y<='1';
        WHEN "10"=>y<='1';
        WHEN "11"=>y<='0';
        WHEN OTHERS=>y<='X';
      END CASE;
    END PROCESS t1;
  END nand2_2;
```

从上面两个例子可以看出，例 7-1 的描述更简洁，更接近于 2 输入“与非”门的行为描述，因此也更易于阅读。例 7-2 的描述是以 2 输入“与非”门的真值表为依据来编写的，它罗列了 2 输入“与非”门的每种输入状态及其对应的输出结果。

集电极开路的 2 输入“与非”门和一般 2 输入的“与非”门在 VHDL 语言的描述上没有什么差异，所不同的只是从不同元件库中提取相应的电路而已。例如：

```
LIBRARY STD;
USE STD.STD_LOGIC.ALL;
USE STD.STD_ttl.ALL;
ENTITY nand2 IS
        ⋮
END nand2;
LIBRARY STD;
USE STD.STD_LOGIC.ALL;
USE STD.STD_ttloc.ALL;
ENTITY nand2 IS
        ⋮
```

```
END nand2;
```

在第一个例子中要生成的是一般 TTL 的 2 输入“与非”门，而在第二个例子中要生成的是 TTL 集电极开路的 2 输入“与非”门。这里所叙述的情况对其它门电路同样适用。因此，在本节里对不同类型门电路的集电极开路输出门将不再赘述。

2. 2 输入“或非”门电路

2 输入“或非”门电路的逻辑表达式为：

$$y=\overline{a \vee b}$$

其逻辑电路图如图 7 - 2 所示。

现举两个用 VHDL 语言描述 2 输入“或非”门电路的例子。

【例 7 - 3】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY nor2 IS
PORT (a,b:IN STD_LOGIC;
      y:OUT STD_LOGIC);
END nor2;

ARCHITECTURE nor2_1 OF nor2 IS
BEGIN
    y<=a NOR b;
END nor2_1;
```

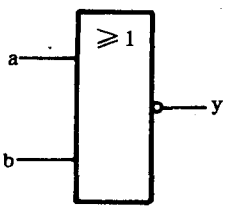


图 7 - 2　2 输入“或非”门电路

【例 7 - 4】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL
ENTITY nor2 IS
PORT (a,b:IN STD_LOGIC;
      y:OUT STD_LOGIC);
END nor2;

ARCHITECTURE nor2_2 OF nor2 IS
BEGIN
  t2:
  PROCESS (a,b)
     VARIABLE comb:STD_LOGIC_VECTOR (1 DOWNTO 0);
  BEGIN
     comb:=a & b;
     CASE comb IS
        WHEN "00"=>y<='1';
        WHEN "01"=>y<='0';
        WHEN "10"=>y<='0';
        WHEN "11"=>y<='0';
        WHEN OTHERS=>y<='X';
```

```
        END CASE;
    END PROCESS t2;
END nor2_2;
```

3. 反相器

反相器电路的逻辑表达式为：

$$y=\overline{a}$$

其逻辑电路图如图 7－3 所示。

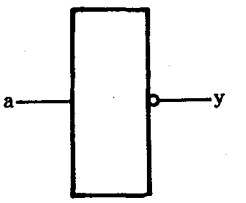


图 7－3　反相器电路

VHDL 语言对反相器的描述如例 7－5 所示。

【例 7－5】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY inverter IS
    PORT (a:IN STD_LOGIC;
          y:OUT STD_LOGIC);
END inverter;
ARCHITECTURE inverter_1 OF inverter IS
BEGIN
    y<=NOT a;
END inverter_1;
```

【例 7－6】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY inverter IS
  PORT (a:IN STD_LOGIC;
        y:OUT STD_LOGIC);
END inverter;
ARCHITECTURE inverter_2 OF inverter IS
BEGIN
  t3:
  PROCESS (a)
      BEGIN
          IF (a='1') THEN
              y<='0'
          ELSE
              y<='1'
          END IF
  END PROCESS;
END inverter_2;
```

4. 3 输入“与非”门电路

3 输入“与非”门电路的逻辑表达式为：

$$y=\overline{(a \wedge b \wedge c)}$$

其逻辑电路如图 7 - 4 所示。

3 输入“与非”门和 2 输入“与非”门的差异仅在于多了一个输入引脚，在用 VHDL 语言编程时，在端口说明中应加一个输入端口。例如，原来的输入端口为 a,b 两个，现在应变为 a,b,c 三个。当然，根据逻辑表达式，该输入端口的信号 c 应与 a 和 b 一样，一起参与逻辑运算，以得到最后的输出 y。

【例 7 - 7】

```
 LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY nand3 IS
  PORT (a,b,c:IN STD_LOGIC;
              y:OUT STD_LOGIC);
END nand3;

ARCHITECTURE nand3_1 OF nand3 IS
BEGIN
  y<=NOT (a AND b AND c);
END nand3_1;
```

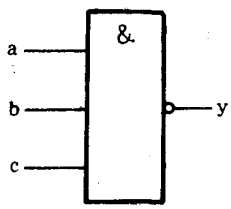


图 7 - 4　3 输入“与非”门电路

【例 7 - 8】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY nand3 IS
  PORT (a,b,c:IN STD_LOGIC;
              y:OUT STD_LOGIC);
END nand3;

ARCHITECTURE nand3_2 OF nand3 IS
BEGIN
  t4:
  PROCESS (a,b,c)
      VARIABLE comb:STD_LOGIC_VECTOR (2 DOWNTO 0);
  BEGIN
      comb:=a & b & c;
      CASE comb IS
         WHEN "000"=>y<='1';
         WHEN "001"=>y<='1';
         WHEN "010"=>y<='1';
         WHEN "011"=>y<='1';
         WHEN "100"=>y<='1';
         WHEN "101"=>y<='1';
         WHEN "110"=>y<='1';
         WHEN "111"=>y<='0';
         WHEN OTHERS=>y<='X';
      END CASE;
  END PROCESS;
END nand3_2;
```

5. 2 输入“异或”门电路

2 输入“异或”门电路的逻辑表达式为：

$$y=a\oplus b$$

其逻辑电路如图 7 - 5 所示。

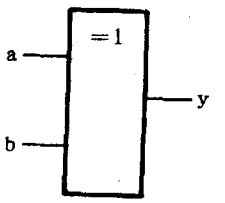


图 7 - 5　2 输入“异或”门电路

用 VHDL 语言描述 2 输入“异或”门的程序实例如例 7 - 9、例 7 - 10 所示。

【例 7 - 9】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY xor2 IS
  PORT (a,b:IN STD_LOGIC;
        y:OUT STD_LOGIC);
END xor2;
ARCHITECTURE xor2_1 OF xor2 IS
BEGIN
  y<=a XOR b;
END xor2_1;
```

【例 7 - 10】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY xor2 IS
  PORT (a,b:IN STD_LOGIC;
        y:OUT STD_LOGIC);
END xor2;
ARCHITECTURE xor2_2 OF xor2 IS
BEGIN
  t5:
  PROCESS (a,b)
     VARIABLE comb:STD_LOGIC_VECTOR (1 DOWNTO 0);
  BEGIN
     comb:=a & c;
     CASE comb IS
        WHEN "00"=>y<='0';
        WHEN "01"=>y<='1';
        WHEN "10"=>y<='1';
        WHEN "11"=>y<='0';
        WHEN OTHERS=>y<='X';
      END CASE;
  END PROCESS;
END xor2_2;
```

上述简单的门电路大多用两种不同型式的 VHDL 语言程序来描述，其行为和功能是完全一样的。事实上还可以运用 VHDL 语言中所给出的语句来描述这些门电路，这就给编程人员提供了较大的编程灵活性。但是，一般来说，无论是编程人员还是阅读这些程序的

人员，都希望能一目了然，因此尽可能采用 VHDL 语言中所提供的语言和符号，用简洁的语句描述其行为，总是首选的描述方式。

7.1.2 编、译码器与选择器

编、译码器和选择器是组合电路中较简单的 3 种通用电路。它们可以由简单的门电路组合连接而构成。例如，如图 7－6 所示，它是一个3—8译码器电路(74LS138)。由有关手册可知，该译码器由 8 个 3 输入“与非”门，4 个反相器和一个 3 输入“或非”门构成。如果事先不作说明，只给出电路，让读者来判读该电路的功能，那么毋庸置疑，要读通该电路就要花较多的时间。如果采用 VHDL 语言，从行为、功能来对 3—8 译码器进行描述，不仅逻辑设计变得非常容易，而且阅读也会很方便。

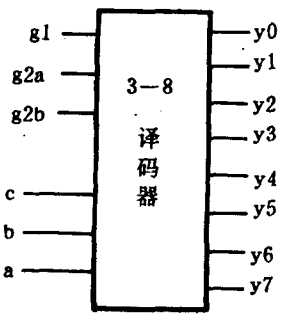


图 7－6　3—8 译码器电路

1. 3—8 译码器

3—8 译码器是最常用的一种小规模集成电路，它有 3 个二进制输入端 a、b、c 和 8 个译码输出端 y0～y7。对输入 a、b、c 的值进行译码，就可以确定输出端 y0～y7 的哪一个输出端变为有效(低电平)，从而达到译码的目的。

3—8 译码器的真值表如表 7－1 所示。3—8 译码器还有 3 个选通输入端 g1、g2a 和 g2b。只有在 g1＝1，g2a＝0，g2b＝0时，3—8 译码器才进行正常译码，否则 y0～y7 输出将均为高电平。

用 VHDL 语句描述的 3—8 译码器如例 7－11 所示。

表 7－1　3—8 译码器的真值表

选通输入			二进制输入端			译码输出端							
g1	g2a	g2b	c	b	a	y0	y1	y2	y3	y4	y5	y6	y7
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
0	X	X	X	X	X	1	1	1	1	1	1	1	1
1	0	0	0	0	0	0	1	1	1	1	1	1	1
1	0	0	0	0	1	1	0	1	1	1	1	1	1
1	0	0	0	1	0	1	1	0	1	1	1	1	1
1	0	0	0	1	1	1	1	1	0	1	1	1	1
1	0	0	1	0	0	1	1	1	1	0	1	1	1
1	0	0	1	0	1	1	1	1	1	1	0	1	1
1	0	0	1	1	0	1	1	1	1	1	1	0	1
1	0	0	1	1	1	1	1	1	1	1	1	1	0

【例 7 - 11 】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY decoder_3_to_8 IS
  PORT (a,b,c,g1,g2a,g2b:IN STD_LOGIC;
        y:OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END decoder_3_to_8;
ARCHITECTURE rtl OF decoder_3_to_8 IS
SIGNAL indata:STD_LOGIC_VECTOR (2 DOWNTO 0);
BEGIN
  indata<=c & b & a;
  PROCESS (indata,g1,g2a,g2b)
    BEGIN
      IF (g1='1' AND g2a='0' AND g2b='0' ) THEN
        CASE indata IS
          WHEN "000"=>y<="11111110";
          WHEN "001"=>y<="11111101";
          WHEN "010"=>y<="11111011";
          WHEN "011"=>y<="11110111";
          WHEN "100"=>y<="11101111";
          WHEN "101"=>y<="11011111";
          WHEN "110"=>y<="10111111";
          WHEN "111"=>y<="01111111";
          WHEN OTHERS=>y<="XXXXXXXX";
        END CASE;
      ELSE
        y<="11111111";
      END IF;
    END PROCESS;
END rtl;
```

在例 7 - 11 中，y(0)对应真值表中的 y0，y(1)对应 y1，依次类推。

2. 优先级编码器

优先级编码器常用于中断的优先级控制，例如，74LS148是一个 8 输入，3 位二进制码输出的优先级编码器。当其某一个输入有效时，就可以输出一个对应的 3 位二进制编码。另外，当同时有几个输入有效时，将输出优先级最高的那个输入所对应的二进制编码。

图 7 - 7 就是优先级编码器的引脚图，它有 8 个输入 input(0)～input(7)和 3 位二进制码输出 y(0)～y(2)。

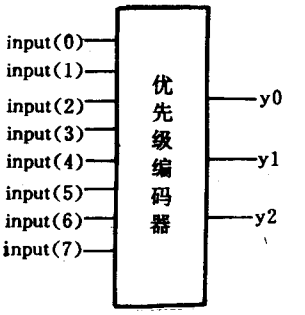


图 7 - 7　优先级编码器电路

该优先级编码器的真值表如表 7 - 2 所示。表中的“X”项表示任意项，它可以是“0”，也可以是“1”。input(0)的优先级最高，input(7)的优先级最低。

表 7 - 2 优先级编码器真值表

输入								二进制编码输出		
input(7)	input(6)	input(5)	input(4)	input(3)	input(2)	input(1)	input(0)	y2	y1	y0
X	X	X	X	X	X	X	0	1	1	1
X	X	X	X	X	X	0	1	1	1	0
X	X	X	X	X	0	1	1	1	0	1
X	X	X	X	0	1	1	1	1	0	0
X	X	X	0	1	1	1	1	0	1	1
X	X	0	1	1	1	1	1	0	1	0
X	0	1	1	1	1	1	1	0	0	1
X	1	1	1	1	1	1	1	0	0	0

用 VHDL 语言描述优先级编码器的程序实例如例 7 - 12 所示。

【例 7 - 12 】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY priorityencoder IS
  PORT (input:IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        y:OUT STD_LOGIC_VECTOR (2 DOWNTO 0));
END priorityencoder;

ARCHITECTURE rtl OF priorityencoder IS
BEGIN
  PROCESS (input)
    BEGIN
      IF(input(0)='0') THEN
          y<="111";
      ELSIF(input(1)='0') THEN
          y<="110";
      ELSIF(input(2)='0') THEN
          y<="101";
      ELSIF(input(3)='0') THEN
          y<="100";
      ELSIF(input(4)='0') THEN
          y<="011";
      ELSIF(input(5)='0') THEN
          y<="010";
      ELSIF(input(6)='0') THEN
          y<="001";
```

```
        ELSIF
              y<="000";
    END IF;
  END PROCESS;
END rtl;
```

由于 VHDL 语言中目前还不能描述任意项，所以不能用前面一贯采用的 CASE 语句来描述，而是采用了 IF 语句。

3. 四选一选择器

选择器常用于信号的切换，四选一选择器可以用于 4 路信号的切换。四选一选择器有 4 个信号输入端 input(0)～input(3)，两个信号选择端 a 和 b 及一个信号输出端 y。当 a,b 输入不同的选择信号时，就可以使 input(0)～input(3)中某个相应的输入信号与输出 y 端接通。例如，当 a=b="0" 时，input(0)就与 y 接通。其逻辑电路如图 7－8 所示。

四选一电路的真值表如表 7－3 所示。现用 VHDL 语言对它进行描述，就可以得到如例 7－13 所示的程序。

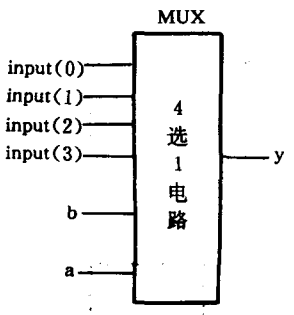


图 7－8　四选一电路

表 7－3　四选一电路真值表

选择输入		数　据　输　入				数据输出
b	a	input(0)	input(1)	input(2)	input(3)	y
0	0	0	X	X	X	0
0	0	1	X	X	X	1
0	1	X	0	X	X	0
0	1	X	1	X	X	1
1	0	X	X	0	X	0
1	0	X	X	1	X	1
1	1	X	X	X	0	0
1	1	X	X	X	1	1

【例 7－13】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY mux4 IS
  PORT (input:IN STD_LOGIC_VECTOR (3 DOWNTO 0);
        a,b:IN STD_LOGIC;
          y:OUT STD_LOGIC);
END mux4;
ARCHITECTURE rtl OF mux4 IS
SIGNAL sel:STD_LOGIC_VECTOR (1 DOWNTO 0);
BEGIN
```

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

例 7-13 的四选一选择器是用 IF 语句描述的，程序中的 ELSE 项作为余下的条件，将选择 input(3)从 y 端输出，这种描述比较安全。当然，不用 ELSE 项也可以，这时必须列出 sel 的所有可能出现的情况，加以一一确认。

7.1.3 加法器、求补器

1. 加法器

加法器有全加器和半加器之分，全加器可以用两个半加器构成，因此下面先以半加器为例加以说明。

半加器有两个二进制一位的输入端 a 和 b 以及一位和的输出端 s 和一位进位位的输出端 co。半加器的真值表如表 7-4 所示，其电路符号如图 7-9 所示。

表 7-4 半加器真值表

二进制输入		和输出	进位输出
b	a	s	co
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

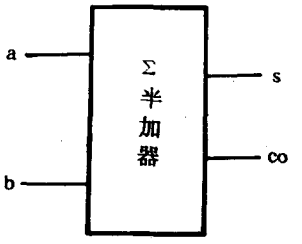


图 7-9 半加器电路

用 VHDL 语言描述半加器的程序如例 7-14 所示。

【例 7-14】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY half_adder IS
  PORT (a, b:IN STD_LOGIC;
        s,co:OUT STD_LOGIC);
```

```
END half_adder;
ARCHITECTURE half1 OF half_adder IS
SIGNAL c,d:STD_LOGIC;
BEGIN
    c<=a OR b;
    d<=a NAND b;
    co<=NOT d;
    s<=c AND d;
END half1;
```

用两个半加器可以构成一个全加器，全加器的电路如图 7 - 10 所示。

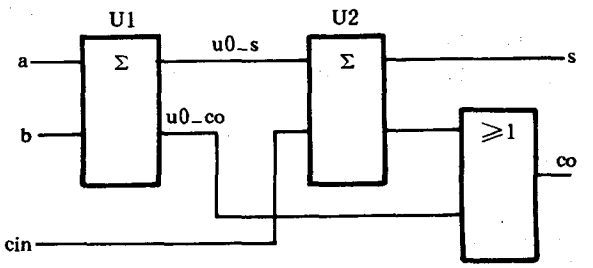


图 7 - 10　用两个半加器构成的全加器

基于半加器的描述，若采用 COMPONENT 语句和 PORT MAP 语句就很容易编写出描述全加器的程序。

【例 7 - 15】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY full_adder IS
  PORT (a,b,cin:IN STD_LOGIC;
        s,co:OUT STD_LOGIC);
END full_adder;
ARCHITECTURE full1 OF full_adder IS
COMPONENT half_adder
  PORT (a,b:IN STD_LOGIC;
        s,co:OUT STD_LOGIC);
END COMPONENT;
SIGNAL u0_co,u0_s,u1_co:STD_LOGIC;
BEGIN
    u0:half_adder PORT MAP (a,b,u0_s,u0_co);
    u1:half_adder PORT MAP (u0_s,cin,s,u1_co);
    co<=u0_co OR u1_co;
END full1;
```

2. 求补器

二进制运算经常要用到求补的操作，8 位二进制数的求补电路符号如图 7 - 11 所示。

求补电路的输入为 a(0)～a(7)，补码输出为 b(0)～b(7)，其中 a(7)和 b(7)为符号位。该电路较复杂，如果像半加器那样，对每个门进行描述和连接是可以做到的，但是那样做就太繁锁了。这里采用 RTL 描述就显得更加简洁、清楚。

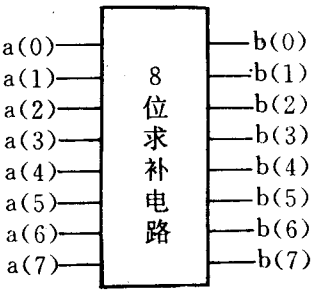


图 7 - 11　8 位二进制数的求补电路符号

【例 7 - 16】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY hosuu IS
    PORT (a: IN STD_LOGIC_VECTOR (7
          DOWNTO 0);
          b:OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END hosuu;
ARCHITECTURE rtl OF hosuu IS
BEGIN
    b<=NOT a+'1';
END rtl;
```

7.1.4　三态门及总线缓冲器

三态门和双向缓冲器是接口电路和总线驱动电路经常用到的器件。它们虽然不属于组合电路，为简化章节，也列于此进行叙述。

1. 三态门电路

三态门的电原理图如图 7 - 12 所示。它具有一个数据输入端 din，一个数据输出端 dout 和一个控制端 en。当 en='1'时，dout=din；当 en='0'时，dout='Z'(高阻)，三态门的真值表如表7 - 5 所示。

表 7 - 5　三态门真值表

数据输入	控制输入	数据输出
din	en	dout
X	0	Z
0	1	0
1	1	1

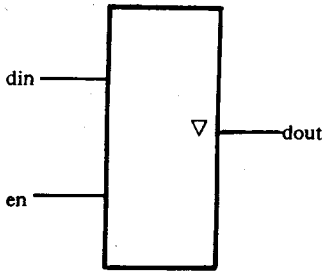


图 7 - 12　三态门电路

用 VHDL 语言描述的三态门的程序实例如例 7 - 17 所示。

【例 7 - 17】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY tri_gate IS
  PORT (din,en:IN STD_LOGIC;
        dout:OUT STD_LOGIC);
END tri_gate;
ARCHITECTURE zas OF tri_gate IS
BEGIN
  tri_gate1:PROCESS (din,en)
    BEGIN
      IF (en='1') THEN
        dout<=din;
      ELSE
        dout<='Z';
      END IF;
    END PROCESS;
END zas;
```

在第 2 章中读者已经知道，一个实体可以对应多种构造体，例 7 - 18 和例 7 - 19 就是用不同的 VHDL 语句描述的三态门的结构。

【例 7 - 18】

```
ARCHITECTURE blk OF tri_gate IS
BEGIN
  tri_gate2:BLOCK (en='1')
    BEGIN
      dout<=GUARDED din;
    END BLOCK;
END blk;
```

该例中采用卫式块语句结构来表示三态门。卫式块语句结构的特点是，只有块语句的条件满足时，块中所含的语句才会被执行。在这里只有 en='1'的条件满足时，dout<=GUARDED din 语句才会被执行。

【例 7 - 19】

```
ARCHITECTURE nas OF tri_gate IS
BEGIN
  tri_gate3: PROCESS (din,en)
    BEGIN
      CASE en IS
        WHEN '1'=>dout<=din;
        WHEN OTHERS=>dout<='Z';
      END CASE;
    END PROCESS;
```

```
END nas;
```

在该例中，当 en='1' 时，使 dout 和 din 的信号保持一致，否则就将“Z”波形赋予 dout，也就是说 dout 不受 din 的影响。

3. 单向总线缓冲器

在微型计算机的总线驱动中经常要用单向总线缓冲器，它通常由多个三态门组成，用来驱动地址总线和控制总线。一个 8 位的单向总线缓冲器如图 7-13 所示。8 位的单向总线缓冲器由 8 个三态门组成，具有 8 个输入和 8 个输出端。所有三态门的控制端连在一起，由一个控制输入端 en 控制。

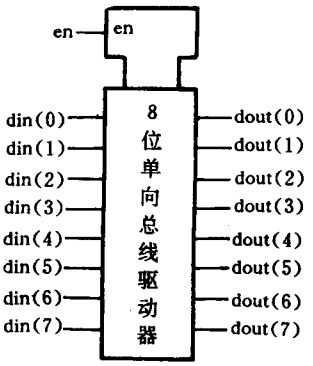


图 7-13 单向总线驱动器

用 VHDL 语言描述的 8 位单向总线缓冲器的程序实例如例 7-20、例7-21 和例 7-22 所示。

【例 7-20】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY tri_buf8 IS
  PORT ( din:IN STD_LOGIC_VECTOR
              (7 DOWNTO 0);
         dout:OUT STD_LOGIC_VECTOR
              (7 DOWNTO 0) BUS;
          en:IN STD_LOGIC);
END tri_buf8;

ARCHITECTURE zas OF tri_buf8 IS
BEGIN
  tri_buff:PROCESS (en,din)
    BEGIN
      IF (en='1' ) THEN
          dout<=din;
      ELSE
          dout<="ZZZZZZZZ";
      END IF
    END PROCESS;
END zas;
```

【例 7-21】

```
ARCHITECTURE blk OF tri_buf8 IS
BEGIN
  tri_buff:BLOCK (en='1')
    BEGIN
      dout<=GUARDED STD_LOGIC_VECTOR (din);
    END BLOCK;
```

```
END blk;
```

【例 7 - 22】

```
ARCHITECTURE nas OF tri_buf8 IS
BEGIN
  tri_buff:PROCESS (en,din)
    BEGIN
      CASE en IS
        WHEN '1'=>dout<=din;
        WHEN OTHERS=>dout <="ZZZZZZZZ";
      END CASE;
    END PROCESS;
END nas;
```

在编写上述程序时应注意，不能将“Z”值赋予变量，否则就不能进行逻辑综合。另外，对信号赋值时“Z” 和“0”或“1”不能混合使用，如：

```
dout<="Z001ZZZZ";
```

这样的语句是不允许出现的。但是变换赋值表达式，分开赋值是可以的。例如：

```
dout(7)<="Z";
dout(6 DOWNTO 4)<="001";
dout(3 DOWNTO 0)<="ZZZZ";
```

4. 双向总线缓冲器

双向总线缓冲器用于对数据总线的驱动和缓冲，典型的双向总线缓冲器的电路图如图 7 - 14 所示。图中的双向缓冲器有两个数据输入输出端 a 和 b，一个方向控制端 dr 和一个选通端 en。当 en=1 时双向总线缓冲器未被选通，a 和 b 都呈现高阻；en=0 时，双向总线缓冲器被选通，如果 dr=0，那么 a=b；如果 dr=1，那么 b=a。双向总线缓冲器的真值表如表 7 - 6 所示。

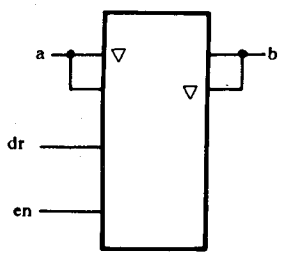


图 7 - 14　双向总线驱动器

表 7 - 6　双向总线缓冲器真值表

en	dr	功　能
0	0	a=b
0	1	b=a
1	X	三　态

用 VHDL 语言描述的双向总线缓冲器的实例如例 7 - 23 所示。

【例 7 - 23】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY tri_bigate IS
  PORT (a,b:INOUT STD_LOGIC_VECTOR (7 DOWNTO 0);
        en:IN STD_LOGIC;
        dr:IN STD_LOGIC);
END tri_bigate;
ARCHITECTURE rtl OF tri_bigate
SIGNAL aout,bout:STD_LOGIC_VECTOR (7 DOWNTO 0);
BEGIN
  PROCESS (a,dr,en)
    BEGIN
      IF ((en='0') AND (dr='1')) THEN
          bout<=a;
      ELSE
          bout<="ZZZZZZZZ";
      END IF;
      b<=bout;
    END PROCESS;
    PROCESS (b,dr,en)
      BEGIN
        IF ((en='0') AND (dr='0')) THEN
            aout<=b;
        ELSE
            aout<="ZZZZZZZZ";
        END IF
        a<=aout;
      END PROCES;
END rtl;
```

从例 7 - 23 可以看出，双向总线缓冲器由两组三态门组成，利用信号 aout 和 bout 将两组三态门连接起来。由于在实际工作过程中 a 和 b 都不可能同时出现“0”和“1”，故在这里没有使用判决函数。

7.2 时序电路设计

在本节的时序电路设计中主要介绍触发器、寄存器和计数器。在介绍这些电路以前，先说明一下时钟信号和复位信号的描述。

7.2.1 时钟信号和复位信号

1. 时钟信号描述

众所周知，任何时序电路都以时钟信号为驱动信号，时序电路只是在时钟信号的边沿

到来时，其状态才发生改变。因此，时钟信号通常是描述时序电路的程序的执行条件。另外，时序电路也总是以时钟进程形式来进行描述的，其描述方式一般有两种：

1）进程的敏感信号是时钟信号

在这种情况下，时钟信号应作为敏感信号，显式地出现在 PROCESS 语句后跟的括号中，例如 PROCESS(clock_signal)。时钟信号边沿的到来，将作为时序电路语句执行的条件，如例 7 - 24 所示。

【例 7 - 24】

```
PROCESS (clock_signal)
BEGIN
  IF (clock_edge_condition) THEN
     signal_out<=signal_in;
              ⋮
     其它时序语句
              ⋮
  END IF;
END PROCESS;
```

例 7 - 24 程序说明，该进程在时钟信号 clock_signal 发生变化时被启动，而在时钟边沿的条件得到满足后才真正执行时序电路所对应的语句。

2）用进程中的 WAIT ON 语句等待时钟

在这种情况下，描述时序电路的进程将没有敏感信号，而是用 WAIT ON 语句来控制进程的执行。也就是说，进程通常停留在 WAIT ON 语句上，只有在时钟信号到来，且满足边沿条件时，其余的语句才能执行，如例 7 - 25 所示。

【例 7 - 25】

```
PROCESS
BEGIN
  WAIT ON (clock_signal) UNTIL (clock_edge_condition);
          signal_out<=signal_in;
                   ⋮
          其它时序语句
                   ⋮
END PROCESS;
```

在编写上述两个程序时应注意：

· 无论 IF 语句还是 WAIT ON 语句，在对时钟边沿说明时，一定要注明是上升沿还是下降沿(前沿还是后沿)，光说明是边沿是不行的。

· 当时钟信号作为进程的敏感信号时，在敏感信号的表中不能出现一个以上的时钟信号，除时钟信号以外，像复位信号等是可以和时钟信号一起出现在敏感表中的。

· WAIT ON 语句只能放在进程的最前面或者最后面。

3）时钟边沿的描述

为了描述时钟边沿，一定要指定是上升沿还是下降沿，这一点可以使用时钟信号的属性描述来达到。也就是说，时钟信号的值是从“0”到“1”变化；还是从“1”到“0”变化。由

此可以得知是时钟脉冲信号的上升沿还是下降沿。

(*a*) 时钟脉冲的上升沿描述

时钟脉冲上升沿波形与时钟信号属性的描述关系如图 7 - 15 所示。

从图中可以看到，时钟信号起始值为“0”，故其属性值 clk'LAST_VALUE='0'；上升沿的到来表示发生了一个事件，故用 clk'EVENT 表示；上升沿以后，时钟信号的值为“1”，故其当前值为 clk='1'。这样，表示上升沿到来的条件可写为：

IF clk='1' AND clk'LAST_VAULE ='0' AND clk'EVENT

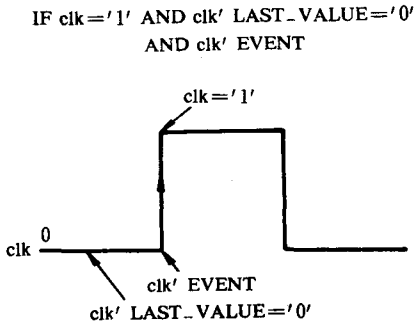


图 7 - 15　时钟脉冲上升沿波形和时钟信号属性描述关系

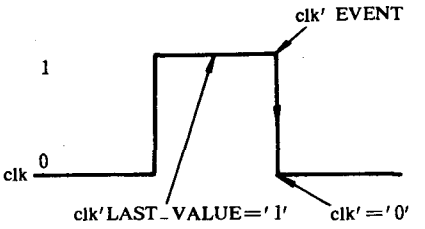


图 7 - 16　时钟脉冲下降沿和时钟信号属性描述关系

(*b*) 时钟脉冲下降沿描述

时钟脉冲下降沿波形与时钟信号属性的描述关系如图 7 - 16 所示。其关系与图 7 - 15 类同，此时 clk'LAST_VALUE='1'；时钟信号当前值为 clk='0'；下降沿到来的事件为 clk'EVENT。这样表示下降沿到来的条件可写为：

IF clk='0' AND clk'LAST_VALUE ='1' AND clk'EVENT

根据上面的上升沿和下降沿的描述，时钟信号边沿检出条件可以统一描述如下：

```
IF clock_signal=current_value AND
   clock_signal'LAST_VALUE AND
   clock_signal'EVENT
```

在某些书刊中边沿检出条件也可简写为：

IF clock_signal=clock_signal'EVENT AND current_value

2. 触发器的同步和非同步复位

触发器的初始状态应由复位信号来设置，复位信号对触发器复位的操作不同，使其可以分为同步复位和非同步复位两种。所谓同步复位，就是当复位信号有效且在给定的时钟边沿到来时，触发器才被复位；而非同步复位则是，一旦复位信号有效，触发器就被复位。

1) 同步复位

在用 VHDL 语言描述时，同步复位一定在以时钟为敏感信号的进程中定义，且用 IF 语句来描述必要的复位条件。下面两个例子就是同步复位方式的描述实例。

【例 7 - 26】

```
PROCESS (clock_signal)
BEGIN
  IF (clock_edge_condition) THEN
    IF (reset_condition) THEN
       signal_out<=reset_value;
    ELSE
       signal_out<=signal_in;
            ⋮
       其它时序语句
            ⋮
    END IF;
  END IF;
END PROCESS;
```

【例 7 - 27】

```
PROCESS
BEGIN
  WAIT ON (clock_signal) UNTIL (clock_edge_condition)
    IF (reset_condition) THEN
       signal_out<=reset_value;
    ELSE
       signal_out<=signal_in;
            ⋮
       其它时序语句
            ⋮
    END IF;
END PROCESS;
```

2) 非同步复位

非同步复位又称异步复位，在描述时与同步方式不同：首先在进程的敏感信号中除时钟信号以外，还应加上复位信号；其次是用 IF 语句描述复位条件；最后在 ELSIF 段描述时钟信号边沿的条件，并加上 EVENT 属性。其描述方式如例 7 - 28 所示。

【例 7 - 28】

```
PROCESS (reset_signal,clock_signal)
BEGIN
  IF (reset_condition) THEN
     signal_out<=reset_value;
  ELSIF(clock_event AND clock_edge_condition) THEN
       signal_out<=signal_in;
            ⋮
       其它时序语句
            ⋮
```

```
    END IF;
  END PROCESS;
```

从例 7 - 28 可以看到，非同步时的信号和变量的代入和赋值必须在时钟信号边沿有效的范围内进行，如在例 7 - 28 中的 ELSIF 后进行的那样。

另外，添加 clock_event 是为了防止没有时钟事件发生时的误操作。譬如，现在时钟事件没有发生而是发生了复位事件，这样该进程就得到了启动。在此情况下，若复位条件没有满足，而时钟边沿条件却是满足的，那么与时钟信号有关的那一段程序(ELSIF 段)就会得到执行，从而造成错误操作。

7.2.2 触发器

触发器种类很多，这里仅举常用的几种加以说明。

1. 锁存器

锁存器根据触发边沿、复位和预置的方式以及输出端多少的不同也可以有多种不同形式的锁存器。

1) D 锁存器

正沿触发的 D 锁存器的电路符号如图 7- 17 所示。它是一个正沿触发的 D 触发器，有一个数据输入端 d，一个时钟输入端 clk 和一个数据输出端 q。D 锁存器的真值表如表 7 - 7 所示。从表中可以看到，D 锁存器的输出端只有在正沿脉冲过后，输入端 d 的数据才传递到输出端 q。用 VHDL 语言描述该 D 锁存器的程序实例如例 7 - 29 和 7 - 30 所示。

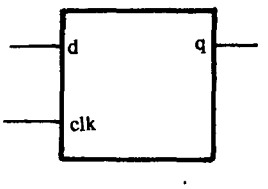


图 7 - 17 D 锁存器

表 7 - 7 D 锁存器真值表

数据输入端	时钟输入端	数据输出端
d	clk	q^{n+1}
X	0	不 变
X	1	不 变
0	↑	0
1	↓	1

【例 7 - 29】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY dff1 IS
  PORT (clk,d:IN STD_LOGIC;
        q:OUT STD_LOGIC);
END dff1;
ARCHITECTURE rtl OF dff1 IS
BEGIN
  PROCESS (clk)
    BEGIN
```

```
    IF (clk'EVENT AND clk='1') THEN
        q<=d;
    END IF;
  END PROCESS;
END rtl;
```

【例 7-30】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY dff1 IS
  PORT (clk,d:IN STD_LOGIC;
        q:OUT STD_LOGIC);
END dff1;
ARCHITECTURE rtl OF dff1 IS
BEGIN
  PROCESS
    BEGIN
      WAIT UNTIL clk'EVENT AND clk='1';
      q<=d;
    END PROCESS;
END rtl;
```

例 7-29 和 7-30 是对时钟信号边沿利用前述的不同方法描述时所得到的两个不同的程序。程序中描述的是正沿触发，如果要改成是下降沿触发，只要对条件作如下改动就行了：

```
IF (clk'EVENT AND clk='0')
```

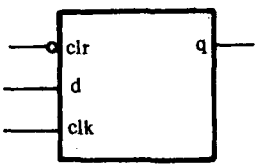


图 7-18 非同步复位的 D 锁存器

2）非同步复位的 D 锁存器

非同步复位的 D 锁存器的电路符号如图 7-18 所示。它和一般的 D 锁存器的区别是多了一个复位输入端 clr。当 clr='0'时，其 q 端输出被强迫置为“0”。clr 又称清零输入端。

用 VHDL 语言描述的非同步复位的 D 锁存器如例 7-31所示。

【例 7-31】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY dff2 IS
  PORT (clk,d,clr:IN STD_LOGIC;
        q:OUT STD_LOGIC);
END dff2;
ARCHITECTURE rtl OF dff2 IS
BEGIN
  PROCESS (clk,clr)
    BEGIN
```

```
        IF (clr='0') THEN
            q<='0';
        ELSIF (clk'EVENT AND clk='1') THEN
            q<=d;
        END IF;
    END PROCESS;
END rtl;
```

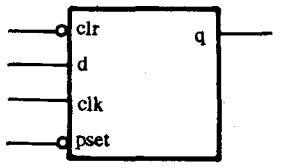


图 7-19 非同步复位/置位 D 锁存器

3) 非同步复位/置位 D 锁存器

非同步复位/置位 D 锁存器的电路符号如图 7-19 所示。除了前述的 d,clk 和 q 端外，还有 clr 和 pset 的复位、置位端。当 clr='0' 时复位，使 q='0'；当 pset='0'时置位，使 q='1'。

用 VHDL 语言描述的非同步复位/置位锁存器的程序实例如例 7-32 所示。

【例 7-32】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY dff3 IS
  PORT (clk,d,clr,pset:IN STD_LOGIC;
                q:OUT STD_LOGIC);
END dff3;
ARCHITECTURE rtl OF dff3 IS
BEGIN
  PROCESS (clk,pset,clr)
    BEGIN
      IF (pset='0') THEN
          q<='1';
      ELSIF (clr='0') THEN
          q<='0';
      ELSIF (clk'EVENT AND clk='1') THEN
          q<=d;
      END IF;
    END PROCESS;
END rtl;
```

从例 7-32 可以看到，事件的优先级置位最高，复位次之，而时钟最低。这样，当 pset=0 时，无论 clr 和 clk-是什么状态，q 一定被置为“1”。

4) 同步复位的 D 锁存器

同步复位的 D 锁存器的电路如图 7-20 所示。与非同步方式不同的是，当复位信号 clr 有效(clr='1') 以后，只是在有效时钟边

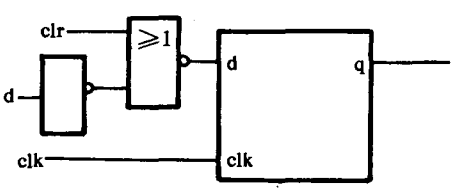


图 7-20 同步复位的 D 锁存器

沿到来时才能进行复位操作。图中 clr='1'以后，在 clk 的上升沿到来时，q 输出才变为“0”。另外，从图中还可以看出复位信号的优先级比 d 端的数据输入高，也就是说当 clr='1' 时，无论 d 端输入什么信号，在 clk 的上升沿到来时，q 输出总是为“0”。

用 VHDL 语言描述的同步复位 D 锁存器的程序实例如例 7 - 33 所示。

【例 7 - 33】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY dff4 IS
  PORT (clk,clr,d:IN STD_LOGIC;
                q:OUT STD_LOGIC);
END dff4;
ARCHITECTURE rtl OF dff4 IS
BEGIN
  PROCESS (clk)
    BEGIN
      IF (clk'EVENT AND clk='1') THEN
        IF (clr='1') THEN
            q<='0';
        ELSE
            q<=d;
        END IF;
      END IF;
    END PROCESS;
END rtl;
```

2. JK 触发器

带有复位/置位功能的 JK 触发器电路符号如图 7 - 21 所示。JK 触发器的输入端有置位输入 pset，复位输入 clr，控制输入 j 和 k，时钟信号输入 clk；输出端有正向输出端 q 和反向输出端 qb。JK 触发器的真值表如表 7 - 8 所示。表中 q0 表示原状态不变，翻转表示改变原来状态。如原来为“0”则变成“1”；原来为“1”则变成“0”。

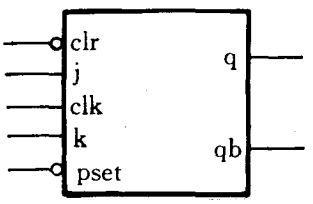


图 7 - 21　JK 触发器

表 7 - 8　JK 触发器真值表

输入端					输出端	
pset	clr	clk	j	k	q	qb
0	1	X	X	X	1	0
1	0	X	X	X	0	1
0	0	X	X	X	X	X
1	1	↓	0	1	0	1
1	1	↓	1	1	翻	转
1	1	↓	0	0	q0	NOT q0
1	1	↓	1	0	1	0
1	1	0	X	X	q0	NOT q0

用 VHDL 语言描述 JK 触发器的程序如例 7 - 34 所示。

【例 7 - 34】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY jkdff IS
  PORT (pset,clr,clk,j,k:IN STD_LOGIC;
        q,qb:OUT STD_LOGIC);
END jkdff;

ARCHITECTURE rtl OF jkdff IS
  SIGNAL q_s, qb_s: STD_LOGIC;
BEGIN
  PROCESS (pset,clr,clk,j,k)
    BEGIN
      IF (pset='0') THEN
         q_s<='1';
         qb_s<='0';
      ELSIF (clr='0') THEN
         q_s<='0';
         qb_s<='1';
      ELSIF (clk' EVENT AND clk='1') THEN
          IF (j='0') AND (k='1') THEN
                q_s<='0';
                qb_s<='1';
          ELSIF (j='1') AND (k='0') THEN
                q_s<='1';
                qb_s<='0';
          ELSIF (j='1') AND (k='1') THEN
                q_s<=NOT q_s;
                qb_s<=NOT qb_s;
          END IF;
        END IF;
        q<=q_s;
        qb<=qb_s;
    END PROCESS;
END rtl;
```

例 7 - 34 中的复位和置位显然也是非同步的，且 pset 的优先级比 clr 高，也就是说当 pset='0'，且 clr='0'时，q 将输出"1"；qb 输出为"0"。这种结果和表 7 - 8 的真值表是不一致的。为了避免这种情况，程序可以改写成例 7 - 35 所示。

【例 7 - 35】

```
ARCHITECTURE rtl OF jkdff IS
  SIGNAL q_s, qb_s:STD_LOGIC;
```

```
BEGIN
  PROCESS (pset,clr,clk,j,k)
    BEGIN
      IF (pset='0') AND (clr='1') THEN
          q_s<='1';
          qb_s<='0';
      ELSIF (pset='1') AND (clr='0') THEN
          q_s<='0';
          qb_s<='1';
      ELSIF (clk'EVENT AND clk='1') THEN
          IF (j='0') AND (k='1') THEN
              q_s<='0';
              qb_s<='1';
          ELSIF (j='1') AND (k='0') THEN
              q_s<='1';
              qb_s<='0';
          ELSIF (j='1') AND (k='1') THEN
              q_s<=NOT q_s;
              qb_s<=NOT qb_s;
          END IF;
        END IF;
      q<=q_s;
      qb<=qb_s;
  END PROCESS;
END rtl;
```

在例 7 - 35 中 pset='0'，clr='0'这种情况未加考虑，那么在逻辑综合时，其输出是未知的。

7.2.3 寄存器

寄存器一般由多位触发器连接而成，通常有锁存寄存器和移位寄存器等。下面主要介绍一些移位寄存器的实例。

1. 串行输入、串行输出移位寄存器

串行输入、串行输出移位寄存器的电原理图如图 7 - 22 所示。它具有两个输入端：数据输入端 a 和时钟输入端 clk；一个数据输出端 b。图中所示的是 8 位的串行移位寄存器，在时钟信号作用下，前级的数据向后级移动。该 8 位移位寄存器由 8 个 D 触发器构成。正如第 5 章所述，利用 GENERATE 语句和 D 触发器的描述就很容易写出 8 位移位寄存器的 VHDL 语言程序，如例 7 - 36 所示。

【例 7 - 36】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY shift8 IS
    PORT (a,clk:IN STD_LOGIC;
```

```
                b:OUT STD_LOGIC);
END shift8;

ARCHITECTURE sample OF shift8 IS
    COMPONENT dff
      PORT (d,clk:IN STD_LOGIC;
                q:OUT STD_LOGIC);
    END COMPONENT;
    SIGNAL z:STD_LOGIC_VECTOR (0 TO 8);
BEGIN
    z(0)<=a;
    g1:FOR i IN 0 TO 7 GENERATE
       dffx:dff PORT MAP (z(i),clk,z(i+1));
    END GENERATE;
    b<=z(8);
END sample;
```

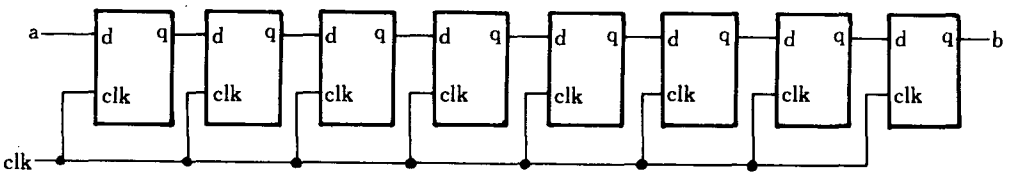


图 7-22　串行输入、串行输出的 8 位移位寄存器

在例 7-36 中把 dff 看作已经生成的元件，然后利用 GENERATE 来循环生成串行连接的 8 个 D 触发器。

8 位移位寄存器直接利用信号来连接也是可以进行描述的，如例 7-37 所示。

【例 7-37】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY shift8 IS
  PORT (a,clk:IN STD_LOGIC;
            b:OUT STD_LOGIC);
END shift8;

ARCHITECTURE rtl OF shift8 IS
SIGNAL dfo_1,dfo_2,dfo_3,dfo_4,dfo_5,dfo_6,dfo_7,dfo_8:STD_LOGIC;
BEGIN
  PROCESS (clk)
    BEGIN
      IF (clk'EVENT AND clk='1') THEN
         dfo_1<=a;
```

```
        dfo_2<=dfo_1;
        dfo_3<=dfo_2;
        dfo_4<=dfo_3;
        dfo_5<=dfo_4;
        dfo_6<=dfo_5;
        dfo_7<=dfo_6;
        dfo_8<=dfo_7;
        b<=dfo_8;
    END IF;
  END PROCESS;
END rtl;
```

在第 3 章里已经提到了变量赋值和信号代入的区别，其中特别强调了，即使执行了信号代入语句，被代入的信号量的值在当时并没有发生改变，直到进程结束，代入过程才同时发生。因此，例 7 - 37 这样描述是正确的。如果将例 7 - 37 中的信号量改成变量，代入符“<=”改成赋值符“:=”，那么该程序所描述的是否仍是一个 8 位移位寄存器？这一点请读者根据已学知识进行思考。

2. 循环移位寄存器

在计算机的运算操作中经常用到循环移位，它可以用硬件电路来实现。一个 8 位循环左移的寄存器的电路符号如图 7 - 23 所示。该电路有 8 个数据输入端 din(0)～din(7)，移位和数据输出控制端 enb，时钟信号输入端 clk，移位位数控制输入端 s(0)～s(2)，8 位数据输出端 dout(0)～dout(7)。循环左移操作的示意图如图 7 - 24 所示。

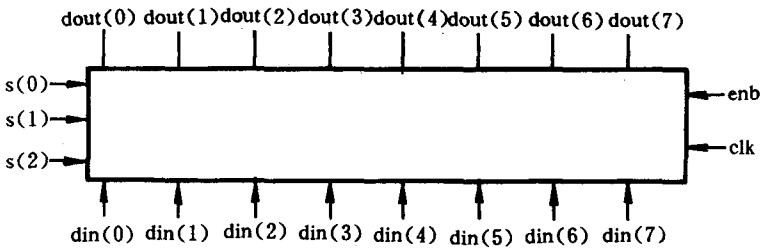


图 7 - 23　8 位循环移位寄存器

当 enb=1 时，根据 s(0)～s(2)输入的数，确定在时钟脉冲作用下，循环左移几位。图 7 - 24 所示是循环左移了 3 位。当 enb=0 时，din 直接输出至 dout。

为了生成 8 位循环左移位的寄存器，在对其进行描述时要调用包集合 CPAC 中的循环左移过程。在 CPAC 中该过程的描述如例 7 - 38 所示。

【例 7 - 38】

```
LIBARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

USE IEEE.STD_LOGIC_ARITH.ALL;
```

```
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
PACKAGE CPAC IS
  PROCEDURE shift(
          din,s:IN STD_LOGIC_VECTOR;
          SIGNAL dout:OUT STD_LOGIC_VECTOR);
END CPAC;
PACKAGE BODY CPAC IS
    PROCEDURE shift(
          din,s:IN STD_LOGIC_VECTOR;
          SIGNAL dout:OUT STD_LOGIC_VECTOR) IS
    VARIABLE sc:INTEGER;
    BEGIN
      sc:=CONV_INTEGER(s);
      FOR i IN din'RANGE LOOP
        IF (sc+i<=din'LEFT) THEN
          dout(sc+i)<=din(i);
        ELSE
          dout(sc+i-din'LEFT)<=din(i);
        END IF;
      END LOOP;
    END shift;
END CPAC;
```

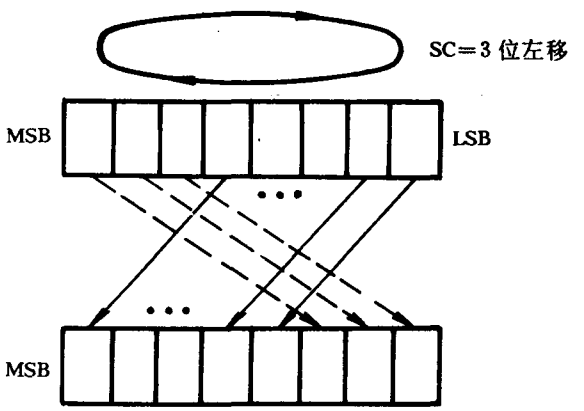


图 7-24　循环左移操作示意图

利用该移位过程就可以来描述 8 位的循环左移的寄存器了，具体如例 7-39 所示。

【例 7-39】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE WORK.CPAC.ALL;
ENTITY bsr IS
  PORT (din:IN STD_LOGIC_VECTOR (7 DOWNTO 0);
```

```
        s:IN STD_LOGIC_VECTOR (2 DOWNTO 0);
        clk,enb:IN STD_LOGIC;
        dout:OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END bsr;

ARCHITECTURE rtl OF bsr IS
BEGIN
  PROCESS (clk) BEGIN
    IF (clk'EVENT AND clk='1') THEN
      IF (enb='0') THEN
          dout<=din;
      ELSE
          shift(din,s,dout);
      END IF;
    END IF;
  END PROCESS;
END rtl;
```

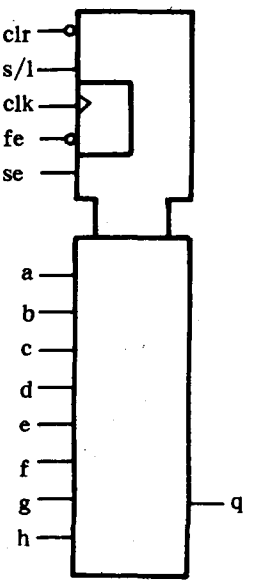


图 7-25 带清零端的 8 位并行装载移位寄存器

该程序经逻辑综合所生成的电路较复杂，读者如有兴趣可参阅参考文献[2]。

3. 带清零端的 8 位并行装载移位寄存器

该移位寄存器就是 TTL 手册中的74166，其引脚图如图 7-25 所示。

图中各引脚名称及功能如下：

a，b～h——8 位并行数据输入端；

se——串行数据输入端；

q——串行数据输出端；

clk——时钟信号输入端；

fe——时钟信号禁止端；

s/l——移位装载控制端；

clr——清零端。

其真值表如表 7-9 所示。

表 7-9 带清零端的 8 位并行装载移位寄存器真值表

输入						内部输出		输出
clr	s/l	fe	clk	a～h	se	qa	qb～qh	qh=q
0	X	X	X	X	X	0	0	0
1	X	0	0	X	X	不改变		不改变
1	X	1	X	X	X	不改变		不改变
1	0	0	↑		X	a～h 装入 qa～qh		
1	1	0	↑	X		se	右移一位	

从表 7－9 可以看到，当清零输入端 clr 为“0”时，8 位寄存器的输出均为“0”，从而使 q 输出也为“0”。fe 是时钟禁止端，当它为“1” 时将禁止时钟，即不管时钟信号如何变化，移位寄存器的状态不发生改变。另外，时钟信号只在上升沿时才有效，即使 fe＝0。如果时钟信号的上升沿未到来，移位寄存器的状态仍不会发生变化。s/l 是移位/装载控制信号。当 s/l＝1 时是移位状态，在时钟信号上升沿的控制下，向右移一位，串行输入端 se 的信号将移入 qa 位，而 q 的输出将是移位前的内部 qg 输出。当 s/l＝0 时是装载状态，在时钟脉冲上升沿的作用下，数据输入端 a～h 的信号就装载到移位寄存器的 qa～qh。根据上述描述的功能，就可以用 VHDL 语言编出描述74166功能的程序，如例 7－40 所示。

【例 7－40】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY sreg8parlwclr IS
    PORT (clr,sl,fe,clk,se,a,b,c,d,e,f,
          g,h:IN STD_LOGIC;
            q:OUT STD_LOGIC);
END sreg8parlwclr;
ARCHITECTURE behav OF sreg8parlwclr IS
SIGNAL tmpreg8:STD_LOGIC_VECTOR (7 DOWNTO 0);
BEGIN
    PROCESS (clr,sl,fe,clk)
      IF (clr='0') THEN
         tmpreg8<="00000000";
         q<=tmpreg8(7);
      ELSE (clk'EVENT) AND (clk='1') AND (fe='0') THEN
        IF (sl='0') THEN
           tmpreg8(0)<=a;
           tmpreg8(1)<=b;
           tmpreg8(2)<=c;
           tmpreg8(3)<=d;
           tmpreg8(4)<=e;
           tmpreg8(5)<=f;
           tmpreg8(6)<=g;
           tmpreg8(7)<=h;
           q<=tmpreg8(7);
        ELSIF (sl='1') THEN
        FOR i IN tmpreg8'HIGH DOWNTO tmpreg8'LOW+1 LOOP
            tmpreg8(i)<=tmpreg8(i-1);
        END LOOP;
        tmpreg8(tmpreg8'LOW)<=se;
        q<=tmpreg8(7);
        END IF;
    END IF;
```

```
    END PROCESS;
END behav;
```

7.2.4 计数器

计数器分同步计数器和异步计数器两种，如果按工作原理和使用情况来分那就更多了。计数器是一个典型的时序电路，分析计数器就能更好地了解时序电路的特性。

1. 同步计数器

所谓同步计数器，就是在时钟脉冲(计数脉冲)的控制下，构成计数器的各触发器状态同时发生变化的那一类计数器。

(1) 带允许端的十二进制计数器

该计数器由 4 个触发器构成，clr 输入端用于清零，en 端用于控制计数器工作，clk 为时钟脉冲(计数脉冲)输入端，qa,qb,qc,qd 为计数器的 4 位二进制计数值输出端。该计数器的真值表如表 7 - 10 所示。

表 7 - 10 带允许端的十二进制计数器真值表

输 入 端			输 出 端			
clr	en	clk	qd	qc	qb	qa
1	X	X	0	0	0	0
0	0	X	不变	不变	不变	不变
0	1	↑	计数值加1			

该十二进制计数器用 VHDL 语言描述的程序如例 7 - 41 所示。

【例 7 - 41】

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

ENTITY count12en IS
    PORT (clk,clr,en:IN STD_LOGIC;
          qa,qb,qc,qd:OUT STD_LOGIC);
END count12en;

ARCHITECTURE rtl OF count12en IS
SIGNAL count_4:STD_LOGIC_VECTOR (3 DOWNTO 0);
BEGIN
    qa<=count_4(0);
    qb<=count_4(1);
    qc<=count_4(2);
    qd<=count_4(3);
  PROCESS (clk,clr)
    BEGIN
      IF (clr='1') THEN
          count_4<="0000";
      ELSIF (clk'EVENT AND clk='1') THEN
```

```
        IF (en='1') THEN
          IF (count_4="1011")THEN
              count_4<="0000";
          ELSE
              count_4<=count_4+'1';
          END IF;
        END IF;
      END IF;
    END PROCESS;
END rtl;
```

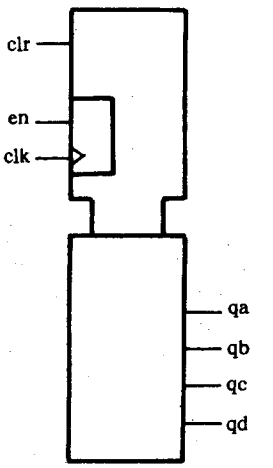


图 7-26 带允许端的十二进制计数器电路

该程序对应的电路的引脚图如图 7-26 所示。

(2) 可逆计数器

所谓可逆计数器，就是根据计数控制信号的不同，在时钟脉冲作用下，计数器可以进行加 1 操作或者减 1 操作的一种计数器。

可逆计数器有一个特殊的控制端，这就是 updn 端。当 updn='1' 时，计数器进行加 1 操作，当 updn='0'时，计数器就进行减 1 操作。一种 6 位二进制可逆计数器的真值表如表 7-11 所示。

表 7-11 6 位二进制可逆计数器真值表

输入端			输出端					
clr	updn	clk	qf	qe	qd	qc	qb	qa
1	X	X	0	0	0	0	0	0
0	1	↑	计数器加 1 操作					
0	0	↑	计数器减 1 操作					

根据该真值表，用 VHDL 语言所描述的 6 位二进制可逆计数器的程序如例 7-42 所示。

【例 7-42】

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

ENTITY updncount64 IS
  PORT (clk,clr,updn: STD_LOGIC;
        qa,qb,qc,qd,qe,qf:STD_LOGIC);
END updncount64;

ARCHITECTURE rtl OF updncount64 IS
SIGNAL count_6:STD_LOGIC_VECTOR (5 DOWNTO 0);
BEGIN
    qa<=count_6(0);
    qb<=count_6(1);
```

```
    qc<=count_6(2);
    qd<=count_6(3);
    qe<=count_6(4);
    qf<=count_6(5);
    PROCESS (clr,clk) BEGIN
      IF (clr='1') THEN
          count_6<=(OTHERS=>'0');
      ELSIF (clk'EVENT AND clk='1') THEN
          IF (updn='1') THEN
              count_6<=count_6+'1';
          ELSE
              count_6<=count_6-'1';
          END IF;
      END IF;
    END PROCESS;
END rtl;
```

根据该程序所合成的电路引脚图如图 7 - 27 所示。

(3) 六十进制计数器

众所周知，用一个 4 位二进制计数器可以构成 1 位十进制计数器，也就是说可以构成 1 位 BCD 计数器，而 2 位十进制计数器连接起来可以构成一个六十进制的计数器。六十进制计数器常用于时钟计数。

一个六十进制计数器的电路引脚图如图 7 - 28 所示。

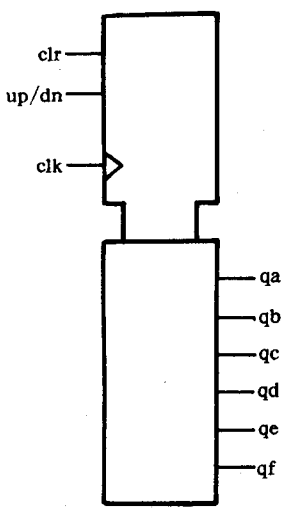


图 7 - 27　6 位二进制可逆计数器电路

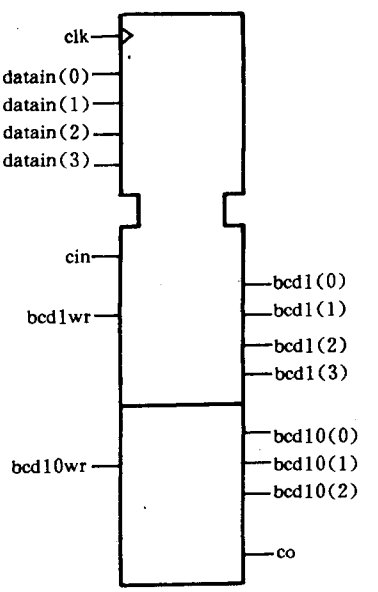


图 7 - 28　六十进制计数器电路

六十进制计数器的输入和输出端的名称和功能说明如下：

clk——时钟输入端；

bcd1wr——个位写控制端；

bcd10wr——十位写控制端；

cin——进位输入端；

co——进位输出端；

datain——数据输入端，共有 4 条输入线 datain(0)～datain(3)；

bcd1——计数值个位输出，共有 4 条输出线 bcd1(0)～bcd1(3)；

bcd10——计数值十位输出，共有 3 条输出线 bcd10(0)～bcd10(2)。

在该六十进制计数器的电路中，bcd1wr 和 bcd10wr 与 datain 配合，以实现对六十进制计数器的个位和十位的装载操作，也就是说可以实现对个位和十位值的预置操作。应注意，在对个位和十位进行预置操作时，datain 输入端是公用的，因而个位和十位的预置操作必定要串行进行。

利用 VHDL 语言描述六十进制计数器的程序如例 7 - 43 所示。

【例 7 - 43】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY bcd60count IS
    PORT (clk,bcd1wr,bcd10wr,cin:STD_LOGIC;
          co:OUT STD_LOGIC;
          datain:IN STD_LOGIC_VECTOR (3 DOWNTO 0);
          bcd1:OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
          bcd10:OUT STD_LOGIC_VECTOR (2 DOWNTO 0));
END bcd60count;

ARCHITECTURE rtl OF bcd60count IS
SIGNAL bcd1n:STD_LOGIC_VECTOR (3 DOWNTO 0);
SIGNAL bcd10n:STD_LOGIC_VECTOR (2 DOWNTO 0);
BEGIN
    bcd1<=bcd1n;
    bcd10<=bcd10n;
    PROCESS (clk,bcd1wr)
      BEGIN
        IF (bedlwr='1') THEN
          bedln<=datain;
        ELSIF (clk'EVENT AND clk='1') THEN
          IF (cin='1') THEN
            IF (bedln=9) THEN
                bcdln<="0000";
            ELSE
```

```
                bcdln<=bcdln+1;
            END IF;
          END IF;
        END IF;
      END PROCESS;

  PROCESS (clk,bcd10wr)
      BEGIN
        IF (bcd10wr='1') THEN
          bcd10n<=datain (2 DOWNTO 0);
        ELSIF (clk'EVENT AND clk='1') THEN
          IF (cin='1' AND bcdln=9) THEN
            IF (bcd10n=5) THEN
                bcd10n<="000";
            ELSE
                bcd10n<=bcd10n+1;
            END IF;
          END IF;
        END IF;
  END PROCESS;

  PROCESS (bcd10n,bcdln,cin)
      BEGIN
        IF (cin='1' AND bcdln=9 AND bcd10n=5) THEN
          co<='1';
        ELSE
          co<='0';
        END IF;
      END PROCESS;
  END rtl;
```

在例 7 - 43 中第一个进程处理个位计数；第二个进程处理十位计数；第三个进程处理进位输出 co 的输出值。应注意，个位和十位的计数条件是不一样的。

2. 异步计数器

异步计数器又称行波计数器，它的下一位计数器的输出作为上一位计数器的时钟信号，这一级一级串行连接起来就构成了一个异步计数器。

异步计数器与同步计数器不同之处就在于时钟脉冲的提供方式，除此之外就没有什么不同，它同样可以构成各种各样的计数器。但是，由于异步计数器采用行波计数，从而使计数延迟增加，在要求延迟小的领域受到了很大限制。尽管如此，由于它的电路简单，仍有广泛的应用。

用 VHDL 语言描述异步计数器，与上述同步计数器不同之处主要表现在对各级时钟脉冲的描述上，这一点请读者在阅读例程时多加注意。

一个由 8 个触发器构成的行波计数器的程序如例 7 - 44 所示，其综合以后的电原理图如图 7 - 29 所示。

【例 7 - 44】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY dffr IS
    PORT (clk,clr,d:IN STD_LOGIC;
          q,qb:OUT STD_LOGIC);
END dffr;

ARCHITECTURE rtl OF dffr IS
SIGNAL q_in:STD_LOGIC;
BEGIN
    qb<=NOT q_in;
    q<=q_in;
  PROCESS (clk,clr)
    BEGIN
      IF (clr='1') THEN
          q_in<='0';
      ELSIF (clk'EVENT AND clk='1') THEN
          q_in<=d;
      END IF
    END PROCESS;
END rtl;

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY rplcont IS
  PORT (clk,clr:IN STD_LOGIC;
        count:OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END rplcont;

ARCHITECTURE rtl OF rplcont IS
SIGNAL count_in_bar:STD_LOGIC_VECTOR (8 DOWNTO 0);
COMPONENT dffr
  PORT (clk,clr,d:IN STD_LOGIC;
        q,qb:OUT STD_LOGIC);
END COMPONENT;
BEGIN
    count_in_bar(0)<=clk;
    gen1:FOR i IN 0 TO 7 GENERATE
      U:dffr PORT MAP (clk=>count_in_bar(i),
        clr=>clr,d=>count_in_bar(i+1),
        q=>count(i),qb=>count_in_bar(i+1));
  END GENERATE;
END rtl;
```

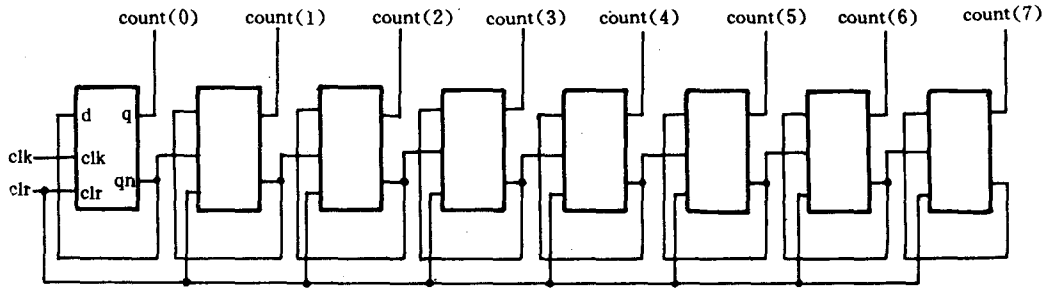


图 7－29　8 位行波计数器电原理图

7.3 存　贮　器

存贮器按其类型可分为只读存贮器和随机存贮器，它们的功能有较大的区别，因此在描述上也有诸多不同。尽管如此，它们仍有许多相同之处，在分别详述它们的各自特性以前，先就共性的一些问题作一说明。

7.3.1　存贮器描述中的一些共性问题

1. 存贮器的数据类型

存贮器是众多存贮单元的一个集合体，按单元号顺序排列。每个单元由若干个二进制位构成，以表示单元中存放的数据值。这种结构和数组的结构是非常相似的。不妨认为某一个存贮器可以用一个数组来代表，每个单元代表数组中的一个元素，数组中的元素序号和存贮器中的单元序号一致。这样，用一个数组就能很好地描述存贮器存放数据的结构了。

每个存贮单元所存放的数可以用不同的、由 VHDL 语句所定义的数的类型来描述，例如用整数或位矢量来描述：如

```
TYPE memory IS ARRAY (INTEGER RANEG<>) OF INTEGER;
```

这是一个元素用整数表示的数组，用它来描述存贮器存贮数据的结构。

```
SUBTYPE wrod IS STD_LOGIC_VECTOR (k－1 DOWNTO 0);
TYPE memory IS ARRAY (0 TO 2＊＊w－1) OF word;
```

这是一个元素用位矢量表示的数组，用它来描述存贮器存贮数据的结构。这里 k 表示存贮单元二进制位数，w 表示数组的元素个数。

2. 存贮的初始化

在用 VHDL 语言描述 ROM 时，ROM 的内容应该在仿真时事先读到 ROM 中，这就是所谓存贮器的初始化。存贮器的初始化要依赖于外部文件的读取，也就是说也依赖于 TEXTIO。下面是对 ROM 进行初始化的实例。

```
变量说明
VARIABLE  startup:BOOLEAN:=TRUE;
VARIABLE  l:LINE;
VARIABLE  j:INTEGER;
```

```
VARIABLE  rom:memory;
FILE romin:TEXT IS IN "rom24s10.in";

初始化程序
IF startup THEN
   FOR j IN rom'RANGE LOOP
        READLINE (romin,l);
        READ (l,rom(j));
   END LOOP;
END IF;
```

一般，ROM 初始化在系统加电之后只执行一次。如果在仿真时，RAM 也要事先赋值的情况下，也可以采用上述同样的方法。

7.3.2 ROM(只读存贮器)

一种容量为 256×4 的 ROM 存贮器的引脚图如图 7-30 所示。该 ROM 有 8 位地址线 adr(0)～adr(7)、4 位数据输出线 dout(0)～dout(3)及 2 位选择控制输入 g1 和 g2。当 g1＝1，g2＝1时，由 adr(0)～adr(7)选中某一 ROM 单元，该单元中的 4 位数据就从 dout(0)～dout(3)输出；否则 dout(0)～dout(3)将呈现高阻状态。据此就可以用 VHDL 语言写出对 ROM 的描述程序，如例 7-45 所示。

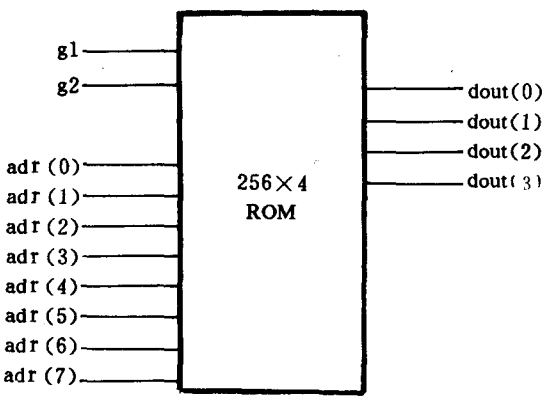


图 7-30　ROM 存贮器引脚图

【例 7-45】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY rom24s10 IS
  PORT (g1,g2:IN STD_LOGIC;
        adr:IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        dout:OUT STD_LOGIC_VECTOR (3 DOWNTO 0));
END rom24s10;

ARCHITECTURE behav OF rom24s10 IS
```

```
SUBTYPE word IS STD_LOGIC_VECTOR (3 DOWNTO 0);
TYPE memory IS ARRAY (0 TO 255) OF word;
SIGNAL adr_in:INTEGER RANGE 0 TO 255;
VARIBLE rom:memory;
VARIBLE startup:BOOLEAN:=TRUE;
VARIBLE l:LINE;
VARIBLE j:INTEGER;
FILE romin:TEXT IS IN "rom24s10.in";
BEGIN
  PROCESS (g1,g2,adr)
    IF startup THEN
      FOR j IN rom'RANGE LOOP
          READLINE (romin,l);
          READ (l,rom(j));
          END LOOP;
          startup:=FALSE;
    END IF;
    adr_in<=CONV_INTEGER (adr);
    IF (g1='1' AND g2='1') THEN
        dout<=rom (adr_in);
    ELSE
        dout<="ZZZZ";
    END IF
  END PROCESS
END behav;
```

例 7 - 45 中的 CONV_INTEGER( )是一个将位矢量转换成整数的函数，在 IEEE 的标准包集合中可以找到，在这里直接引用了该函数。

7.3.3 RAM(随机存贮器)

RAM 和 ROM 的主要区别，在于其描述上有读和写两种操作，而且在读、写上对时间有较严格的要求。一种容量为 8×8 位的 SRAM 的引脚框图如图 7 - 31 所示。

如图 7 - 31 所示，它有 8 条地址线 adr(0)～adr(7)、8 条数据输入线 din(0)～din(7)、8条数据输出线 dout(0)～dout(7)。另外，wr 为写控制线，rd 为读控制线以及 cs 为片选控制线。当 cs＝1、wr 信号由低变高(上升沿)时，din 上的数据将写入由 adr 所指定的单元；当 cs＝1、rd＝0时，由 adr 所指定单元的内容将从 dout 的数据线上输出。

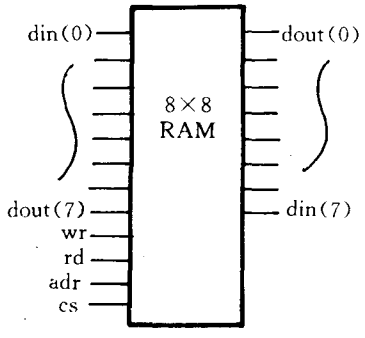


图 7 - 31 SRAM 引脚框图

由 VHDL 语言描述的 SRAM 的程序实例如例 7 -

46 所示。

【例 7－46】

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

ENTITY sram64 IS
    GENERIC (k:INTEGER:=8;
             w:INTEGER:=3);
    PORT (wr,rd,cs:IN STD_LOGIC;
          adr:IN STD_LOGIC_VECTOR (w－1 DOWNTO 0);
          din:IN STD_LOGIC_VECTOR (k－1 DOWNTO 0);
          dout:OUT STD_LOGIC_VECTOR (k－1 DOWNTO 0));
END sram64;

ARCHITECTURE behav OF sram64 IS
SUBTYPE word IS STD_LOGIC_VECTOR (k－1 DOWNTO 0);
TYPE memory IS ARRAY (0 TO 2**w－1) OF word;
SIGNAL adr_in:INTEGER RANGE 0 TO 2**w－1;
SIGNAL sram:memory;
SIGNAL din_change,wr_rise:TIME:=0 ps;

BEGIN
    adr_in<=CONV_INTEGER (adr); --位矢量变换成整数
    PROCESS (wr)
      BEGIN
        IF (wr'EVENT AND wr='1') THEN
          IF (cs='1' AND wr='1') THEN
              sram (adr_in)<=din AFTER 2 ns;
          END IF;
        END IF;
        wr_rise<=NOW; --wr 上升边时间
        ASSERT (NOW－din_change>=800 ps)
        REPORT "SETUP ERROR din(sram)"
        SEVERITY WARNING; --din 建立时间检查
      END PROCESS;

    PROCESS (rd,cs)
      BEGIN
        IF (rd='0' AND cs='1') THEN
            dout<=sram(adr_in) AFTER 3 ns;
        ELSE
            dout<="ZZZZZZZZ" AFTER 4 ns;
        END IF;
      END PROCESS;
```

```
    PROCESS (din)
      BEGIN
          din_chang<=NOW;
          ASSERT (NOW-wr_rise>=300 ps)
          REPORT "HOLD ERROR din (sram)"
          SEVERITY WARNING; --din 保持检查
      END PROCESS;
END behav;
```

在例 7-46 中加了一些信号传送的延迟时间和错误检查的语句。它们是在仿真时检查 RAM 定时关系所必须的，程序将保证实际进行逻辑综合后的电路能满足 RAM 的定时要求。

7.3.4 FIFO(先进先出堆栈)

FIFO 是先进先出堆栈，作为数据缓冲器，通常其数据存放结构是完全和 RAM 一致的，只是存取方式有所不同。图 7-32 是容量为 8×4 位的 FIFO 的引脚框图和原理框图。

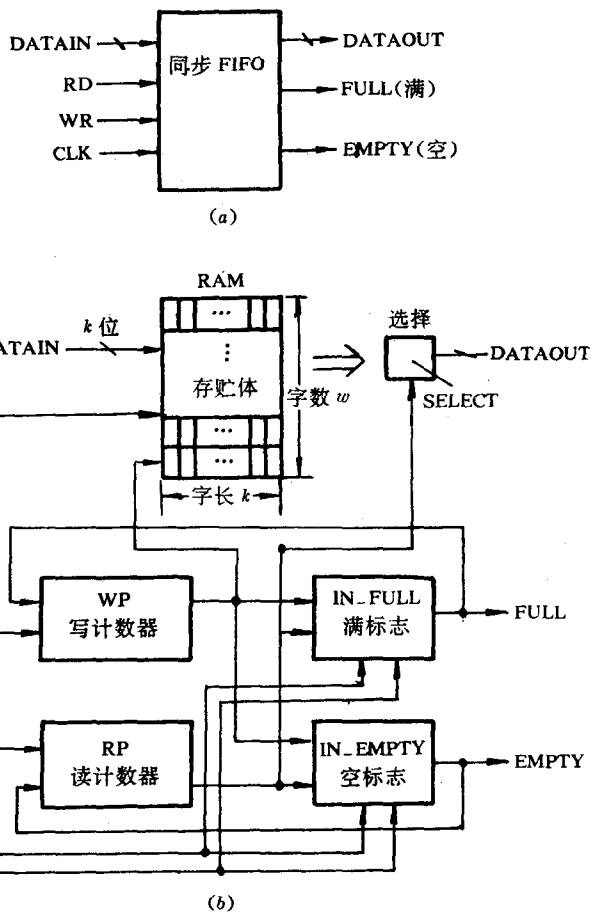


图 7-32 FIFO 引脚框图和原理框图

(a) 引脚框图；(b) 原理框图

图 7－32 中的 FIFO 有 4 条数据输入线 DIN，4 条数据输出线 DOUT，一条读控制线 RD，一条写控制线 WR，一条时钟输入线 CLK 及两条状态信号线，即满信号和空信号线 (FULL,EMPTY)。

FIFO 由 6 个功能块组成，它们是存贮体、写指示器(WP)、读指示器(RP)、满逻辑 IN_FULL、空逻辑 IN_EMPTY 和选择逻辑 SELECT。这是一个同步的 FIFO。在时钟脉冲的上升沿作用下，当 WR＝0 且 FULL＝0 时，DIN 的数据将压入 FIFO 堆栈。在通常情况下，RP 指示器所指出的单元内容总是放于 DOUT 的输出数据线上，只是在 RD＝0 且 EMPTY＝0 时，RP 指示器内容才改变而指向 FIFO 的下一个单元，下一个单元的内容替换当前内容并从 DOUT 输出。应注意，在任何时候 DOUT 上有一个数据输出，而不像 RAM 那样，只有在读有效时才有数据输出，平时为三态输出。

FIFO 的存贮器实际上是一个环形数据结构，由 WP 和 RP 分别指示数据写和读的对应单元。在这里 WP 指示的是新数据待写入的单元地址，发一个 WR 有效信号(WR＝0)，就可将 DIN 上的数据写入该单元；而 RP 指示的是已读出数据的单元地址，要想读下一个数据就要发一个 RD 有效信号(RD＝'0') 使 RP＝RP＋1，这时就可以读出下一个新的数据了。RP 和 WP 之间的信号关系如图 7－33 所示。

FIFO 在复位以后就处于初始状态，WP＝0，RP＝7，此时 FIFO 处于空状态，数据第

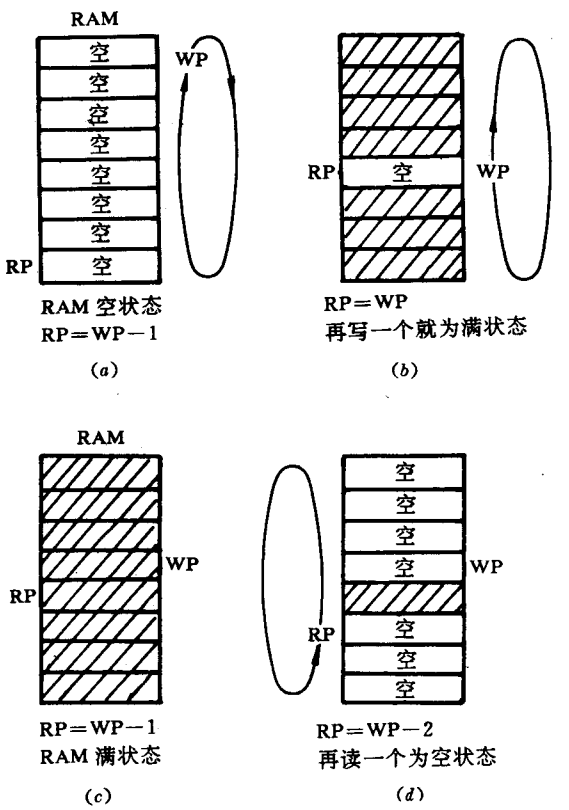


图 7－33　RP 和 WP 的关系

(a) 初始状态或空状态；(b) RP＝WP 状态；(c) 满状态；(d) RP＝WP－2 状态

一次写入的单元应是 0 号单元。RP 和 WP 之间应满足 RP=WP−1。RP=WP 状态是 FIFO 只要进行一次写操作就会变成满的状态。如图 7 - 33(*a*)连续写 7 个数据以后就会变成 RP=WP=7，此时若再写一个数据就会使 FIFO 变成满状态。RP=WP−2 状态时只要再读一次就会使 FIFO 变为空的状态，如图 7 - 33(*d*)所示。由图 7 - 33(*a*)和图 7 - 33(*c*)可以看出，满状态和空状态的 RP 和 WP 的关系是一致的，均为 RP=WP−1。但是，稍加分析即可知道，满或空状态出现之前的一个状态是各不相同的。在 RP=WP 时，由于写一个数据而使其进入满状态(RP=WP−1)，而在 RP=WP−2 时，由于读一个数据而使其进入空状态(RP=WP−1)。据此，即可得到满或空信号产生的条件。

FIFO 的 VHDL 描述如例 7 - 47 所示。

【例 7 - 47】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY fifo IS
  GENERIC (w:INTEGER:=8;
           k:INTEGER:=4);
  PORT (clk,reset,wr,rd:IN STD_LOGIC;
        din:IN STD_LOGIC_VECTOR (k-1 DOWNTO 0);
        dout:OUT STD_LOGIC_VECTOR (k-1 DOWNTO 0);
        full,empty:OUT STD_LOGIC);
END fifo;

ARCHITECTURE behav OF fifo IS
TYPE memory IS ARRAY (0 TO w-1) OF STD_LOGIC_VECTOR(k-1 DOWNTO 0);
SIGNAL ram:MEMORY;
SIGNAL wp,rp:INTEGER RANGE 0 TO w-1;
SIGNAL in_full,in_empty:STD_LOGIC;

BEGIN
full<=in_full;
empty<=in_empty;
dout<=ram(rp);
PROCESS (clk)                                          ┐
  BEGIN                                                │
    IF (clk'EVENT AND clk='1') THEN                    │
      IF (wr='0' AND in_full='0') THEN                 ├ 数据写堆栈
        ram (wp)<=din;                                 │
      END IF;                                          │
    END IF;                                            │
  END PROCESS;                                         ┘
```

```
PROCESS (clk,reset)
  BEGIN
    IF (reset='1') THEN
          wp<=0;
    ELSIF (clk'EVENT AND clk='1') THEN
      IF (wp='0' AND in_full='0') THEN
        IF (wp=w-1) THEN
            wp<=0;
        ELSE
            wp<=wp+1;
        END IF;
      END IF;
    END IF;
END PROCESS;
```

wp 修改描述

```
PROCESS (clk,reset)
        BEGIN
          IF (reset='1') THEN
              rp<=w-1;
          ELSIF (clk'EVENT AND clk='1') THEN
            IF (rd='0' AND in_empty='0') THEN
              IF (rp=w-1) THEN
                  rp<=0;
              ELSE
                  rp<=rp+1;
              END IF;
            END IF;
          END IF;
      END PROCESS;
```

rp 修改描述

```
    PROCESS (clk,reset)
    BEGIN
      IF (reset='1') THEN
          in_empty<='1';
      ELSIF (clk'EVENT AND clk='1') THEN
        IF ((rp=wp-2 OR (rp=w-1 AND wp=1)
            OR (rp=w-2 AND wp=0)) AND (rd='0'
            AND wr='1')) THEN
              in_empty<='1';
        ELSIF(in_empty='1' AND wr='0') THEN
              in_empty<='0';
        END IF;
      END IF;
    END PROCESS;
```

empty 标志产生描述

```
    PROCESS (clk,reset)                                          ┐
      BEGIN                                                      │
        IF (reset='1') THEN                                      │
          in_full<='0';                                          │
        ELSIF (clk'EVENT AND clk='1')THEN                        │
          IF (rp=wp AND wr='0' AND rd='1') THEN                  │
                in_full<='1';                                    ├ full 标志产生描述
          ELSIF (in_full='1' AND rd='0') THEN                    │
                in_full<='0';                                    │
          END IF;                                                │
        END IF;                                                  │
    END PROCESS;                                                 ┘
END behav;
```

例 7－47 由 3 条代入语句和4个进程语句描述了 FIFO 的工作原理。3 条代入语句反映了当前满或空的状态及当前 FIFO 所输出的数据。第一个进程描述 FIFO 的数据压入操作；第二个进程描述写数据地址指示器的数值修改；第三个进程描述读数据地址指示器的数值修改；第四、第五进程描述 FIFO 的“空”、“满”标志的产生。

第 8 章

仿真与逻辑综合

本章对利用 VHDL 语言设计硬件系统的两个重要步骤(仿真和逻辑综合)进行了详细介绍。

8.1 仿　　真

在前面几章已经详细地介绍了 VHDL 语言的基本语句和它们的使用方法，同时还列举了许多利用 VHDL 语言设计一般逻辑电路的实例。为了验证这些设计模块是否正确，还需对这些设计模块进行仿真。在目前，各国的相关公司和厂商已为设计者提供了众多的仿真工具，如：Synopsys 公司的 VHDL System Symulator，model Technology 公司的 SYNARIO VHDL Simulator，VEDA Design Automation 公司的 VULCAL 等。通过这些仿真工具，设计者可对各设计层次的设计模块进行仿真，以确定这些设计模块的功能，逻辑关系及定时关系是否满足设计要求。所以，仿真是利用 VHDL 语言进行硬件设计的一个必不可少的步骤，它贯穿设计的整个过程。

如第 1 章所述，在硬件系统设计过程中一般要进行 3 次仿真：行为级仿真、RTL 级仿真和门级仿真。各级所要达到的仿真目的是不一样的，同时对 VHDL 语言的描述要求也有所不同。下面就仿真中的几个主要问题作一介绍。

8.1.1 仿真输入信息的产生

硬件系统通常是通过输入信号来驱动的，在不同输入信号情况下其行为表现是产生不同的输出结果。因此仿真输入信息的产生是对系统进行仿真的重要前提和必须进行的步骤。仿真信息的产生通常有两种方法：程序直接产生方法和读 TEXIO 的方法。

1. 程序直接产生法

所谓程序直接产生法就是由设计者设计一段 VHDL 语言程序，由该程序直接产生仿真的输入信息。例如要对第 7 章的例 7 - 40 带允许端的十二进制计数器进行仿真。该计数器有 3 个输入端，仿真时要产生 clr、en 和 clk 3 个输入信号，如图 8 - 1 所示。3 个输入信号之间有严格的定时关系。这些定时波形可以用进程来产生。

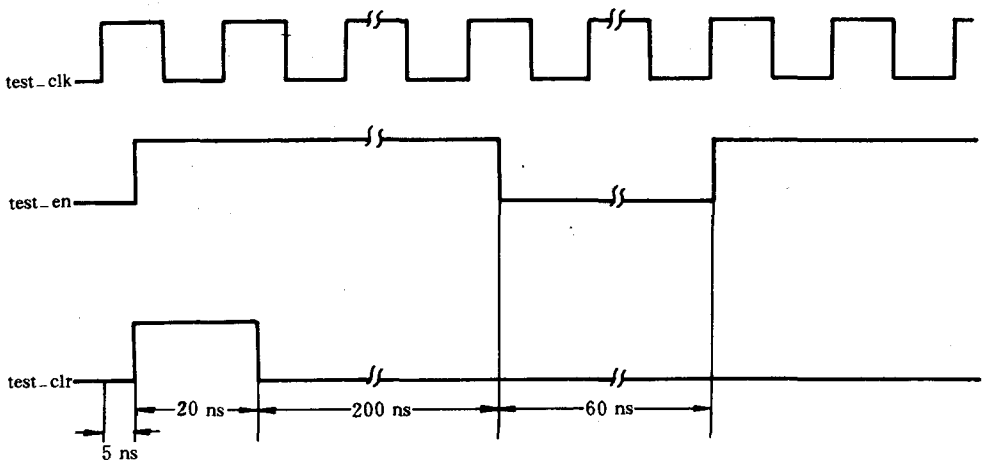


图 8-1　带允许端的十二进制计数器的仿真输入信号

例如：

```
        ⋮
CONSTANT clk_cycle:TIME := 20 ns;
        ⋮
PROCESS                                          ┐
  BEGIN                                          │
    test_clk<='1';                               │
    WAIT FOR clk_cycle/2;                        ├ 产生周期为 20 ns 的时钟信号
    test_clk<='0';                               │
    WAIT FOR clk_cycle/2;                        │
END PROCESS;                                     ┘
PROCESS                                          ┐
  BEGIN                                          │
    test_clr<='0';                               │
    test_en<='1';                                │
    WAIT FOR clk_cycle/4;                        │
    test_clr<='1';                               │
    WAIT FOR clk_cycle;                          ├ 产生复位和允许信号
    test_clr<='0';                               │
    WAIT FOR clk_cycle * 10;                     │
    test_en<='0';                                │
    WAIT FOR clk_cycle * 3;                      │
    test_en<='1';                                │
    WAIT;                                        │
  END PROCESS;                                   ┘
```

例中的第一个进程产生周期为 20 ns 的时钟脉冲 test_clk。开始，test_clk='1'，保持 10 ns。然后，test_clk='0'，再保持 10 ns，得到一个时钟周期。该进程没有指定敏感量，因此当进程执行到最后一条语句以后又返回到最前面，开始执行进程的第一条语句。如此循环往复就能产生出一串周期为 20 ns 的时钟脉冲。

例中的第二个进程用来产生初始的复位(清除)信号和计数允许信号。该进程可产生宽 20 ns 的复位信号，复位以后 260 ns 再使 test_en 有效(置为"1")，从而使计数器进入正常的计数状态。该进程的最后一条语句是 WAIT 语句，它表明该进程只执行一次，进程在 WAIT 语句上处于无限制的等待状态。

利用程序直接产生输入信号的实例如例 8 - 1 所示。

【例 8 - 1】

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

ENTITY test_count12en IS
END test_count12en;
ARCHITECTURE siml OF test_count12en IS
COMPONENT count12en
  PORT (clk,clr,en:IN STD_LOGIC;
        qa,qb,qc,qd:OUT STD_LOGIC);
END COMPONENT;
CONSTANT clk_cycle:TIME := 20 ns;
SIGNAL test_clk,test_clr,test_en:STD_LOGIC;
SIGNAL t_qa,t_qb,t_qc,t_qd:STD_LOGIC;
BEGIN
  U0:count12en PORT MAP (clk=>test_clk,
     clr=>test_clr,en=>test_en,qa=>t_qa,
     qb=>t_qb,qc=>t_qc,qd=>t_qd);
PROCESS
  BEGIN
    test_clk<='1';
    WAIT FOR clk_cycle/2;
    test_clk<='0';
    WAIT FOR clk_cycle/2;
  END PROCESS;

PROCESS
  BEGIN
    test_clr<='0';
    test_en<='1';
    WAIT FOR clk_cycle/4;
    test_clr<='1';
    WAIT FOR clk_cycle;
```

```
        test_clr<='0';
        WAIT FOR clk_cycle * 10;
        test_en<='0';
        WAIT FOR clk_cycle * 3;
        test_en<='1';
        WAIT;
    END PROCESS;
END siml;
```

根据例 8 - 1 的仿真程序可以得到仿真波形如图 8 - 2 所示。

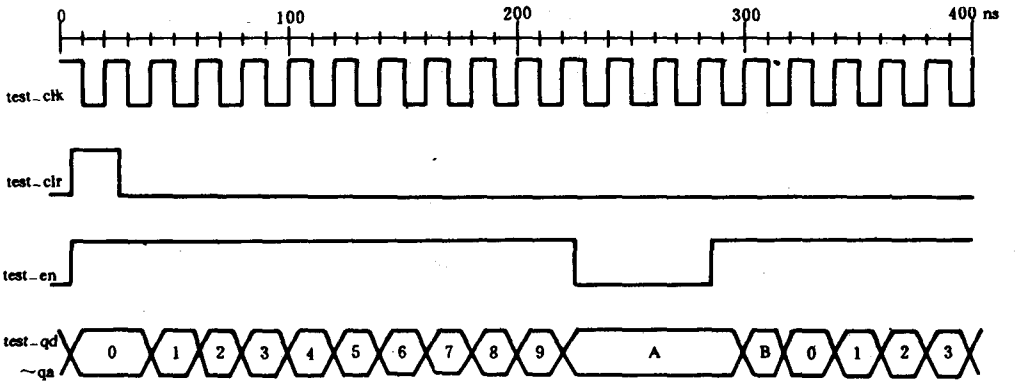


图 8 - 2　带允许端的十二进制计数器的仿真波形

2. 读 TEXTIO 文件产生法

在由程序直接产生输入信号的方法中，仿真模块的编程人员必须了解输入信号的详细状态和它们与时间的关系，这对编程人员似乎提出了太高的要求。为此，人们设计了一种用数据文件输入仿真的办法，也就是说，仿真输入数据按定时要求按行存于一个文件中(即 TEXTIO 文件)。在仿真时，根据定时要求按行读出，并赋予相应的输入信号。例如图 8 - 3 就是根据图 8 - 1 的仿真输入信号要求所设计的 TEXTIO 文件 test1. in。

test1. in

	clk	clr	en
0 ns	1	0	0
5 ns	1	1	1
10 ns	0	1	1
15 ns	0	1	1
20 ns	1	1	1
25 ns	1	0	1
30 ns	0	0	1
35 ns	0	0	1
40 ns	1	0	1
	⋮		
220 ns	1	0	1
225 ns	1	0	0
230 ns	0	0	0.

```
235 ns    0          0          0
240 ns    1          0          0
                ⋮
280 ns    1          0          0
285 ns    1          0          1
290 ns    0          0          1
295 ns    0          0          1
300 ns    1          0          1
                ⋮
```

图 8 - 3　TEXTIO 文件 test1. in 的文件格式

在 test. in 文件中每行包含 3 位数据，第 1 位为 clk，第 2 位为 clr，第 3 位为 en。每行数据之间的定时间隔为 10 ns。如果在程序中每隔 10 ns 读入一行数据，并将读入值赋予对应的 clk，clr 和 en，那么就产生了图 8 - 1 所示的仿真输入信号。这一点利用 TEXTIO 中的 READLINE 和 READ 语句就很容易实现了。例如：

```
        ⋮
FILE intest:TEXT IS IN "test1. in";
SIGNAL test_clk,test_clr,test_en:STD_LOGIC;
CONSTANT clk_cycle:TIME := 20 ns;
      ⋮
PROCESS
    VARIABLE li:LINE;
    VARIABLE clk,clr,en:STD_LOGIC;
    BEGIN
      READLINE (intest,li);
      READ (li,clk);
      READ (li,clr);
      READ (li,en);
      test_clk<=clk;
      test_clr<=clr;
      test_en<=en;
      WAIT FOR clk_cycle/2;
      IF (ENDFILE (intest)) THEN
            WAIT;
      END IF;
END PROCESS;
      ⋮
```

在该例中就描述了每隔 10 ns 从 test1. in 文件中读入一行数据，并将其对应值赋予 test_clk，test_clr 和 test_en 的情况。该进程除非碰到了 test1. in 文件的末尾标志，否则该进程中的语句将循环执行。这样也就产生了图 8 - 1 所示的仿真输入信号。

利用该产生仿真输入信号的具体仿真模块的实例如例 8 - 2 所示。

【例 8 - 2】

```
LIBRARY IEEE.STD;
USE STD.TEXTIO.ALL;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_TEXTIO.ALL;
ENTITY test_count12en IS
END test_count12en;
ARCHITECTURE sim2 OF test_count12en;
COMPONENT count12en PORT (clk,clr,en:IN STD_LOGIC;
                          qa,qb,qc,qd:OUT STD_LOGIC);
END COMPONENT;
FILE intest:TEXT IS IN "test.in";
SIGNAL test_clk,test_clr,test_en:STD_LOGIC;
SIGNAL t_qa,t_qb,t_qc,t_qd:STD_LOGIC;
CONSTANT clk_cycle:TIME := 20 ns;
BEGIN
    U0:count12en PORT MAP (clk=>test_clk,
        clr=>test_clr,en=>test_en,qa=>t_qa,
        qb=>t_qb,qc=>t_qc,qd=>t_qd);
    PROCESS
        VARIABLE li:LINE;
        VARIABLE clk_v,clr_v,en_v:STD_LOGIC;
        BEGIN
          READLINE (intest,li);
          READ (li,clk_v);
          READ (li,clr_v);
          READ (li,en_v);
          test_clk<=clk_v;
          test_clr<=clr_v;
          test_en<=en_v;
          WAIT FOR clk_cycle/2;
          IF (ENDFILE (intest)) THEN
              WAIT;
          END IF;
        END PROCESS;
END sim2;
```

输入仿真信号产生时还应该注意的一点是，输入控制信号和时钟信号最好不要在同一仿真时刻发生变化。例如图 8 - 1 中的 test_clr、test_en 变化时间与时钟变化沿错开了四分之一的时钟周期。这样做的好处是，防止仿真中因判别二者变化的先后不同而出现相反的结果，使仿真结果具有唯一性。

8.1.2 仿真 Δ

仿真 Δ(即仿真中的 Δ 延时)对仿真来说是至关重要的。它能使那些零延时事件得到适

当的排队次序，以便在仿真过程中得到一致的结果。众所周知，用 VHDL 语言程序来描述系统的硬件，它所描述的仅仅是系统的行为和构造，最终表现为门电路之间的连接关系。因此，在处理中对某些部分先处理，对另外一些部分后处理并不要求有非常严格的顺序关系。例如，某一个组合逻辑电路，其输入为 a 和 b，其输出为 q，它由一个反相器，一个与非门和一个与门构成，其连接关系如图 8－4 所示。用 VHDL 语言所描述的对应程序模块如例 8－3 所示。

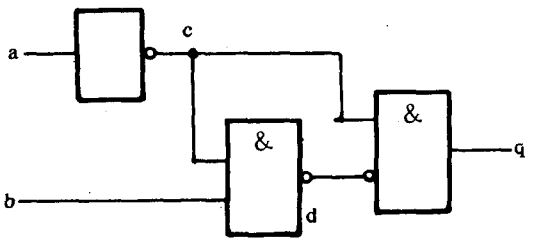


图 8－4　一个组合逻辑电路例

【例 8－3】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY sample IS
    PORT (a,b:IN STD_LOGIC;
          q:OUT STD_LOGIC);
ARCHITECTURE behav OF sample IS
    SIGNAL c,d:STD_LOGIC;
BEGIN
    c<=NOT (a);
    d<=NOT (b AND c);
    q<=c AND d;
END behav;
```

在该模块的构造体中，3 条语句都是信号代入语句，因此它们都是并发语句，只要其敏感量有变化，该语句将被启动执行一次。现在假设信号 b 为“1”，端口 a 的信号有一个变化，即从“1”变成“0”。第一条信号代入语句的敏感量是 a，因此，该语句启动执行一次，使信号量 c 由“0”变为“1”。第二、第三条语句都含有敏感量 c，因此，这两条语句都将启动执行一次。在仿真中第二、第三条语句既然是并发语句，按理谁先执行，谁后执行，其结果是一样的，但是事实并非如此。下面分析一下两种不同情况。

若第三条语句先执行，由于 d='1'，c='1'，故 q=1；接着执行第二条语句，由于 b='1'，c='1'，故 d='0'。d 由“1”到“0”将再次启动第三条语句执行，此时 d='0'，c='1'，故 q=0。这样，输出端口 q 就有由“0”变“1”，又由“1”变“0”的一个正跳变化。

若第二条语句先执行，由于 b='1'，c='1'，故 d='0'。由于 d 和 c 的变化使第三条语句执行，此时 d='0'，c='1'，故 q=0。即使 a 值发生由“1”变“0”的变化，q 值将始终维持为“0”。两种不同情况的 q 输出波形如图 8－5 所示。

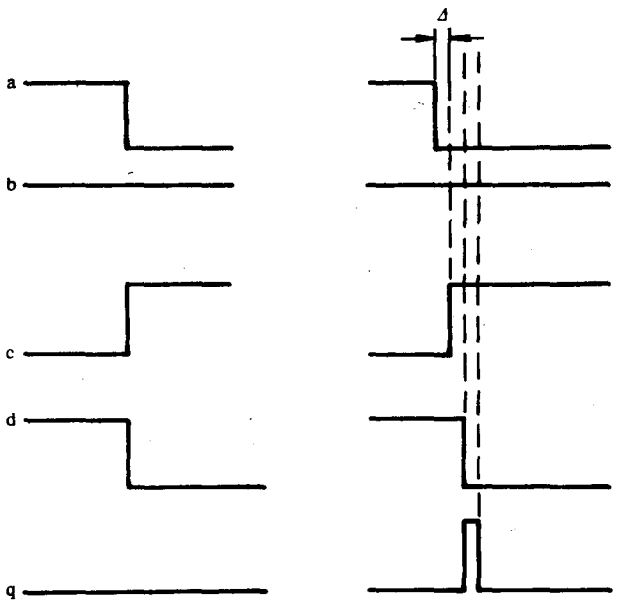


图 8－5　两种不同情况的 q 输出波形

由上面分析可知，在仿真过程中，仿真次序不一致就会产生不同的仿真结果，这当然是不允许的。为了取得与硬件动作一致的仿真结果，必须引入一个适当的仿真同步机制，使仿真结果和处理次序先后无关。这种仿真同步机制就是 Δ 延时同步机制或称仿真 Δ 机制。

所谓 Δ 延时同步机制就是对那些零延时的事件，在仿真中加一个无限小的时间量，例如在 VHDL 语言中时间量的最小单位为 1 fs(10^{-15} s)，那么 Δ 延时就不能超过这个值。也就是说即使加有限个 Δ 延时的时间值，也决不会使其超过仿真时间的最小分辨率。下面再以例 8－4 采用 Δ 延时为例作一说明，其仿真过程如图 8－6 所示。

如图 8－6 所示，输入端 a 的一个信号变化，在输出端要出现新的值需要有 3 个仿真 Δ 延时时间。如果 a 信号变化时刻为 0 ns，那么 q 输出端出现新的值的时刻为 0 ns＋3Δ。前面假设 Δ 是一个无限小量，也就是说，从仿真角度来看，有限个 Δ 的延时是可以忽略的，相当于 a 有一个自“1”至“0”的跳变，立即使信号 c 产生一个自“0”到“1”，使信号 d 产生一个自“1”到“0”的跳变，而输出 q 维持原值不变。应注意这些跳变被认为是发生在同一仿真时刻 0 ns 处。引入 Δ 延时的目的仅仅是为了便于排出仿真计算的次序，在仿真波形中是不反映计算过程的，而只反映最终的计算结果。这样处理以后，使得仿真结果和硬件动作就完全一致起来了。

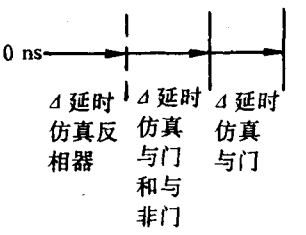


图 8－6　采用 Δ 延时的仿真过程

总而言之，在那些零延时的语句中，如例 8－3 的 3 条代入语句，在仿真时都将加上 Δ 的延时，这样就解决了仿真时由于计算顺序不同所带来的不一致性。

8.1.3 仿真程序模块的书写

为了进行正确的仿真，对仿真程序的书写也有一定的要求。例如，程序应包含仿真输入信号的处理部分，下面就几个问题作一说明。

1. 可简化实体描述

例 8－1 所示是一个带允许端的十二进制计数器的仿真模块，在仿真过程中要输出的是仿真信号。这些仿真信号通常在仿真模块中定义，如例中的 test_clk，test_clr 和 test_en 等。因此，在仿真模块的实体中可以省略有关端口的描述。如例中的实体描述为：

```
ENTITY test_count12en IS
END test_count12en;
```

2. 程序中应包含输出错误信息的语句

在仿真中往往要对波形、定时关系进行检查，如不满足要求，应输出仿真错误信息，以引起设计人员的注意。在 VHDL 语言中 ASSERT 语句就专门用于错误验证及错误信息的输出。该语句的书写格式如下：

```
ASSERT  条件  [REPORT 输出错误信息]
              [SEVERITY 出错级别]
```

ASSERT 后跟的是条件，也就是检查的内容，如果条件不满足，则就输出错误信息和出错级别。出错信息将指明具体出错内容或原因。出错级别表示错误的程序。在 VHDL 语言中出错级别分为：NOTE，WARNING，ERROR 和 FAILURE 共 4 个级别，这些都将由编程人员在程序中指定。

在例 7－45 sram64 的程序中用 ASSERT 语句对数据建立时间进行检查的实例如下所示：

```
PROCESS (wr)
  BEGIN
    IF (wr'EVENT AND wr='1') THEN
        IF (cs='1' AND wr='1') THEN
          sram (adr_in)<=din AFTER 2 ns;
        END IF;
    END IF;
  wr_rise<=NOW; wr 上升边时间
  ASSERT (NOW-din_change>=800 ps)
  REPORT "SETUP ERROR din (sram)"
  SEVERITY WARNING; --din 建立时间检查
END PROCESS;
```

该段程序描述，当 wr 信号由低到高跳变时，也就是要将 SRAM 数据输入端的数据写入存贮单元时，要对数据线上数据的建立时间进行检查，NOW 表示当前的仿真时刻，也就是 wr 上跳的时间。din_change 表示该数据出现的时刻。ASSERT 语句检查的条件为 NOW－din_change>=800 ps。如果这个条件不满足，表明数据建立时间不够长，写入的数据就

不可靠。REPORT 后跟的出错内容信息“SETUP ERROR din(sram)”，表明是 sram 数据输入的建立时间不够而产生的错误。SEVERITY 后跟的是出错级别“WARING”，也就是警告级别。设计人员可以根据这些出错信息，适当地修改相应部分的程序，以使其满足原先提出的要求。

3. 用配置语句选择不同仿真构造体

在 2.3 节中已详细介绍了 CONFIGRATION 这一配置语句的功能和使用实例。在编写仿真程序模块时，为了方便，也经常要使用该语句。设计者为了获得较佳的系统性能，总要采用不同方法，设计不同结构的系统进行对比仿真，以寻求最佳的系统结构。在这种情况下，系统的实体只有一个，而对应构造体可以有多个。仿真时可以用 CONFIGRATION 语句进行选配。例如，用该语句可以选配例 8 - 1 的 siml 构造体：

```
CONFIGRATION cfg_test OF test_count12en IS
   FOR sim1
   END FOR;
END cfg_test;
```

同样，仿真时也可以选配其它构造体，如选配例 8 - 2 的 sim2 构造体：

```
CONFIGRATION cfg_test OF test_count12en IS
   FOR sim2
   END FOR;
END cfg_test;
```

由此可知，在仿真程序模块中使用配置语句会给仿真带来极大的便利。

4. 不同级别或层次的仿真有不同要求

正如前面所述，系统仿真通常由 3 个阶段组成：行为级仿真、RTL 级仿真和门级仿真。它们的仿真目的和仿真程序模块的书写要求都各不相同。对此，设计者必须充分注意。

(1)行为级仿真

行为级仿真的目的是验证系统的数学模型和行为是否正确，因而对系统的抽象程度较高。由于有这个前提，对行为级仿真程序模块的书写没有太多限制，凡是 VHDL 语言中的语句和数据类型都可在程序中使用。在书写时应尽可能使用抽象程度高的描述语句，以使程序更简洁明了。

另外，除了某些系统规定的定时关系以外，一般的电路延时及传输延时在行为级仿真中都不予以考虑。

(2)RTL 级仿真

通过行为级仿真以后，下一步就是要将行为级描述的程序模块改写为 RTL 描述的程序模块。RTL 级仿真是为了使仿真模块符合逻辑工具的要求，使其能生成门级逻辑电路。

如前所述，根据目前逻辑综合工具的情况，有些 VHDL 语言中所规定的语句是不能使用的，例如，ATTRIBUTE，带有卫式(GUARDED)的语句等(具体请参见附录 A、B)。另外，在程序中绝对不能使用浮点数，尽可能少用整数，最好使用 STD_LOGIC 和 STD_LOGIC_VECTOR 这两种类型来表示数据(不同逻辑工具有不同要求)。在 RTL 仿真中尽管可以不考虑门电路的延时，但是像传输延时等那样一些附加延时还应该加以考虑，并用

TRANSPORT 和 AFTER 语句在程序中体现出来。

(3)门级电路仿真

RTL 程序模块经逻辑综合以后就生成了门级电路。既然 RTL 程序模块已经通过仿真，为何还要对门级电路进行仿真呢？这主要有以下几个原因。

第一，在 RTL 仿真中一般不考虑门的延时，也就是说进行零延时仿真。在这种情况下系统的工作速度不能得到正确的验证。不仅如此，由于门延时的存在还会对系统内部工作过程及输入输出带来意想不到的影响。

第二，正如在 4.2 节所述那样，在 RTL 描述中像"Z"和"X"那样的状态，在描述中是可以将其屏蔽的，但是利用逻辑综合工具，根据不同的约束条件，对电路进行相应变动时，这种状态就有可能发生传播。在门级电路仿真中出现这种状态是不允许的。

RTL 描述经逻辑综合生成门电路的过程中，需对数据类型进行转换。一般情况下，输入输出端口只限定使用 STD_LOGIC 和 STD_LOGIC_VECTOR 数据类型。

8.2 逻 辑 综 合*

所谓逻辑综合就是将较高抽象层次的描述自动地转换到较低抽象层次描述的一种方法。就现有的逻辑综合工具而言，所谓逻辑综合就是将 RTL 级的描述转换成门级网表的过程。当前适用于 VHDL 语言的逻辑综合工具主要有：Cadence Design Systems 公司的 Synergy，Synopsys 公司的 Design Compiler Family，Mentar Graphics 公司的 Autologic Ⅱ 等十几种。设计人员只要正确地使用这些工具就可以得到系统的门级网络表。应该说，对于系统的硬件设计人员，他们并不需要详细地了解逻辑综合的细节，只要知道逻辑综合工具的使用方法和大概情况就行了。本节就是根据这样的目的，简单介绍一下逻辑综合的一般概念和有关的基本知识。

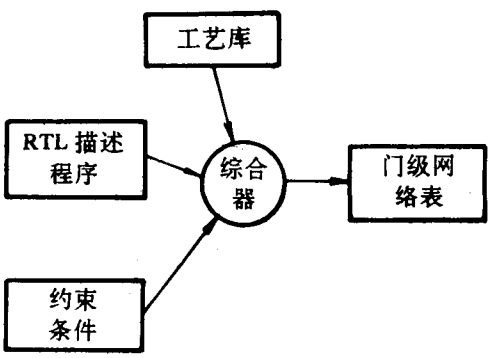


图 8-7 逻辑综合过程示意图

一般逻辑综合的过程如图 8-7 所示。逻辑综合过程要求的输入为：RTL 描述的程序模块；约束条件，如面积、速度、功耗、可测性；支持工艺库，如 TTL 工艺库、MOS 工艺库、CMOS 工艺库等。输出的是门级网表。RTL 描述的程序模块在前面已详细介绍，下面仅就约束条件、工艺库、门级网表的基本知识作一介绍。

8.2.1 约束条件

在逻辑综合过程中为优化输出和工艺映射的需要，一定要有相应的约束条件，以实现对所设计结构的控制，也就是说，采用不同的约束条件如面积、延时、功耗和可测性等，对

* 该部分主要参考文献[7]。

于同样的一个系统，其实现的系统结构是不一样的。

1. 面积约束条件

在将设计转换成门级电路时通常要加面积的约束条件。这是一个设计目标，也是逻辑综合过程中进行优化的依据之一。多数的逻辑综合工具允许设计者按工艺库中门级宏单元所用的单位来指定面积的约束条件。例如，如果用等效门作为测量单位，那么面积约束条件即可以用门的个数来描述。如：

```
max_area 1200
```

一旦确定了面积约束条件，在逻辑综合时就将该条件通知逻辑综合工具。逻辑综合工具利用各种可能的规则和算法，尽可能地减少该设计的面积。优化过程将反复地进行，直到最终所设计的面积小于或等于 1 200 个门单位为止。

2. 时间延时约束条件

时间延时约束条件最常用的描述方法是指定输入输出的最大延时时间。用延时约束条件来引导优化和映射，对设计电路来说是一个相当困难的任务。在某些条件下，无论逻辑综合工具采用什么样的优化手段，最终达不到预期目标的情况时有发生。一种典型的时间延时约束条件的描述如下：

```
max_delay 1.7 data_out
```

这种描述规定信号 data_out 的最大延时应小于或等于 1.7 个库单位时间。

有时为了对所设计的每个节点进行延时计算，还应进行静态分析。也就是说，根据网表中每个连接元件的延时模型，对节点进行定时分析，给出最好和最坏的延时情况。然后检查电路，看所有的延时限制条件是否满足。如果满足则进行优化和工艺映射，否则就要更换优化方案。

8.2.2 属性描述

用属性描述来界定设计的环境，例如由属性来规定所设计电路的负载数、驱动能力和输入信号定时等。

1. 负载

每个输出引脚都要规定一个驱动能力，由它确定在一个特定的时间范围内驱动多少负载，每个输入也要有一个指定的负载值。通过负载的计算就可以推算出，因负载的轻重而使输出波形的变坏程度。

负载属性将指明某一信号的输出负载能力，在工艺库中按库单位计算。例如：

```
set_load 6 xbus
```

该属性规定输出信号 xbus 可带动 6 个库单位负载信号。

2. 驱动

驱动属性规定驱动器电阻的大小，也即控制驱动电流的大小。它同样也按工艺库的单位来指定。例如：

```
set_drive 2 ybus
```

该属性规定输出信号 ybus 有 2 个库单位的驱动能力。

3. 到达时间

逻辑综合工具通常用静态时间分析器来检查正在综合的逻辑是否满足用户规定的延时

限制条件。在特定的节点设置到达时间，以便进行指定的定时分析，这一点有时是非常重要的。例如，某逻辑电路的所有输入信号中有一路信号比其它信号要迟到达，而逻辑电路的输出又要满足所给定的延时限制的要求。这就给逻辑综合提出了严格的要求，也就是说，该路信号的迟到时间加上在该电路中的延时时间不应该超过用户对该电路的延时限制。这种要求表明，逻辑综合结果要求该路迟到信号在本逻辑模块中的延迟要比其它路信号更小，也即要尽可能减少该信号从输入到输出通路上的门的级数。

8.2.3 工艺库

在根据约束条件进行逻辑综合时，工艺库将持有综合工具所必要的全部信息，即工艺库不仅仅含有 ASIC 单元的逻辑功能，而且还有该单元的面积、输入到输出的定时关系、输出的扇出限制和对单元所需的定时检查。例如，一个 2 输入与门的工艺库描述如下所示：

```
LIBRARY (xyz) {
CELL (and2) {
        area:5
        pin (a1,a2) {
            direction:input;
            capacitance:1;
        }
        pin (o1) {
            direction:output;
            functon:"a1 * a2";
            timing ( ) {
            intrinsic_rise:0.37;
            intrinsic_fall:0.56;
            rise_resistance:0.1234;
            fall_resistance:0.4567;
            related_pin:"a1,a2";
            }
        }
    }
}
```

该例描述了一个名称为 xyz 的工艺库中的一个单元，库单元名为 and2，它有 2 个输入 a1 和 a2，一个输出 o1。该 and2 单元的面积为 5 个库单位，要用一个库单位的负载电容能力的信号才能驱动它的一个输入引脚。输出引脚 o1 的固有上升和下降延时规定为不带负载时的输出延时。器件输出 o1 是输入 a1 和 a2 的函数，在计算延时时应从 a1、a2 输入到 o1 输出这样一个通道的延时。

多数逻辑综合工具都有一个完整而复杂的模型，该模型能计算通过一个 ASCI 单元的延时。这类模型不仅包括固有的上升和下降时间，而且还包括输出负载、输入级波形的斜

度延时和估计的引线延时。这样，某一电路的总延时就可写为：

总延时＝固有延时＋负载延时＋引线延时＋输入级波形斜度延时

在这里，

固有延时(惯性延时)——不带任何负载的门延时；

负载延时——驱动输出时因负载电容所产生的附加延时；

引线延时——信号在引线上传送的延时，它和单元的物理特征有关；

输入级波形斜度延时——由于输入波形不够陡所引起的延时。

工艺库还包括如何用有关的工艺参数和工作条件换算延时信息的数据，其中工作条件是器件工作温度和加在器件上的供电电压。

8.2.4 逻辑综合的基本步骤

应用逻辑综合工具将 RTL 描述转换至门级描述一般应有 3 步。

第一步：将 RTL 描述转换成未优化的门级布尔描述(如与门、或门、触发器、锁存器等)；

第二步：执行优化算法，产生优化的布尔描述；

第三步：按照目的工艺要求，采用相应工艺库把优化的布尔描述映射成实际的逻辑门。

上述 3 个步骤的执行过程如图 8－8 所示。

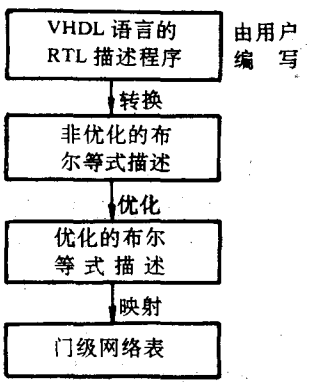


图 8－8 逻辑综合的主要步骤示意图

1. RTL 描述至未优化的布尔描述的转换

从 RTL 描述转换到布尔描述是由逻辑综合工具来实现的，该过程不受用户控制。其最终的转换结果是一种中间结果，格式随不同逻辑综合工具而异，且对用户是不透明的。

按照转换的规则和算法，将 RTL 描述中的 IF，CASE，LOOP 语句以及条件信号代入和选择信号代入等语句转换成中间的布尔表达式，要么装配组成，要么由推论形成触发器和锁存器。

2. 布尔优化描述

布尔优先过程是将一个非优化的布尔描述转化成一个优化的布尔描述的过程。这个工作是逻辑综合过程中的一个重要的工作，它采用了大量的算法和规则。优化的一种方法是，先将非优化的布尔描述转换到最低级描述(pla 格式)，然后再优化这种描述(用 pla 优化技术)，最后再用共享公共项(包括引入中间变量)去简化逻辑，减少门的个数。

将非优化的布尔描述转换成一种 pla 格式的过程称为展平设计，它将所有的逻辑关系都转换成简单的 AND(与)和 OR(或)的表达式。这种转换的目的是，使非优化的布尔描述格式转换成能执行优化算法的布尔描述格式。例如非优化的布尔描述如下：

```
a＝b AND c;
b＝x OR (y AND z);
c＝q OR w;
```

输出 a 用 3 个功能方程式描述，其中 b 和 c 为中间变量。展平的功能是将中间变量 b 和 c 置换掉，完全用不带中间变量的布尔式来表示。展平过程实际上是一个消元过程：

a = (x OR (y AND z)) AND (q OR w)

= x AND (q OR w) OR (y AND z) AND (q OR w)

= (x AND q) OR (x AND w) OR (y AND z AND q) OR (w AND y AND z)

这样 a 的布尔描述就消去了中间节点或中间变量，其结构是一种二级逻辑门。

这种逻辑设计方法是非常快，也是非常容易的。貌似这种逻辑结构的速度一定比较高，因为它含的级数较少。但是，实际上这种逻辑结构可能比有更多级的逻辑结构的速度更慢。其原因是，某个输入信号要与多个逻辑门的输入端相连接，这样就大大增加了该信号的扇出负载，从而使延时增加。而且其面积也非常大，因为它没有共享项，每项必须对应独立的门电路。另外，也存在大量很难展平的电路，这是因为产生的项数非常之多，一个只含 AND 函数的方程只产生一个积项，而一个含有大型异或功能的函数能产生成百上千个积项。2 输入异或门就有(a AND $\bar{b}$) OR ($\bar{a}$ AND b)，一个 N 输入异或将包含 2^{N-1}个积项。例如，16 个输入的异或含有 32 768 项，而 32 位输入的异或会超过 20 亿个项。很显然，对这类功能的布尔描述是很难展平的。

尽管如此，展平毕竟可以使设计去除隐含结构。设计者如想减少工作量，最好用一小块随机控制逻辑去做展平工作，以便与提取公因数部分连接，并去产生一种小型的逻辑描述。

提取公因数是把附加的中间项加到结构描述中去的一种过程，它与展平过程恰好是一个相反的运算过程。如前所述，展平设计通常会使设计变得非常之大，并且展平过程可能比提公因数过程在速度上慢得多。

提公因数的设计将使输入到输出之间的逻辑级数增加，从而使延时增加，但净结果的设计面积会更小，是一个速度较慢的设计。

通常，设计者想得到一个接近展平设计那样速度快的设计只需要用驱动能力大的驱动器，但面积却不能像提公因子的设计那样小。理想的情况将是，就速度而言，在设计中对延时小的通道应采用展平设计；而对延时要求不那么高的地方应采用提公因数设计以减小设计面积。

3. 门级映射

该映射过程是取出经优化后的布尔描述，并利用从工艺库中得到的逻辑和定时上的信息去做网表，网表是对用户所提出的面积和速度目标的一种体现。工艺库中存有大量的网表，它们在功能上相同，但在速度和面积上却有一个很宽的选择范围。某些网表速度快，但实现起来要花费更多的库单元；而另一些花费库的单元少，但速度则要慢一些。

映射过程根据优化的布尔描述、工艺库和用户提出的约束条件，将输出一个优化的网表，该网表的结构是以工艺库单元为基础而建成的。

第 9 章

计时电路设计实例

在前面几章对 VHDL 语言的基本知识和使用 VHDL 语言设计系统硬件的方法作了详细介绍。本章将以较简单的计时控制器，1/100 s 计时控制电路的设计为例，说明一下用 VHDL 语言设计硬件的具体过程。由于该电路较简单，因此省略了行为描述这一步，而是直接用 RTL 描述来设计该电路。

9.1 1/100 s 计时器的功能要求和结构

1/100 s 计时器常用于体育竞赛及各种要求有较精确定时的各领域。以往利用常用的中小规模集成电路也可以设计出这种 1/100 s 的计时器，但是其体积通常都较大，携带和使用都很不方便。为此，要求设计一块专用的 ASIC 芯片，除开关、时钟和显示功能以外，它包括 1/100 s 计时器所有的控制和定时功能，其体积应和机械式计时器的大致相同。

9.1.1 1/100 s 计时器的功能要求

(1) 精度应大于 1/100 s

计时器能显示 1/100 s 的时间，故提供给计时器内部定时的时钟脉冲频率应大于 100 Hz，可选 1 kHz。

(2) 计时器的最长计时时间为 1 h

在一般的短时计时应用中，1 h 是足够了。为此需要一个 6 位的显示器，显示的最长时间为59 分 59.99 秒。

(3) 设置复位和启/停开关

复位开关用来使计时器清零，并作好计时准备。启/停开关的使用方法应与传统的机械式计时器相同，即按一下启/停开关，启动计时器开始计时，再按一启/停开关计时终止。复位开关可以在任何情况下使用，即使在计时过程中，只要一按复位开关，计时进程应立刻终止，并对计时器清零。

9.1.2 1/100 s 计时器的结构设想

1/100 s 计时器的结构可如图 9 - 1 所示。从图 9 - 1 可以看到，1/100 s 计时器由复位开关、启/停开关、系统电源复位电路、时钟脉冲发生器、7 段 LED 显示器和 1/100 s 计时

控制芯片组成。复位开关 reset_sw 和启/停开关 start_stop_sw 都是高电平有效。另外，开关的消抖动电路应放在控制芯片部分考虑。系统复位输入 sysres 是计时器加电复位的输入端，其复位电路是外加的，不包含在芯片中。时钟输入端 clk 是由外加时钟脉冲发生器的输出提供的，本设计中要求输入一个频率稳定的 1 kHz 时钟脉冲。6 位显示器需要 6 个 7 段 LED，控制芯片的 7 条段输出线 segment(6 TO 0)与 7 段 LED 的对应段相连。控制芯片的 6 条 common(5 TO 0)输出线分别接到各个 LED，用来选择显示的 LED。common 以166 Hz 的频率使 6 个 LED 按次序循环点亮，从而可以得到一个人眼观察无闪烁感觉的稳定的显示输出。

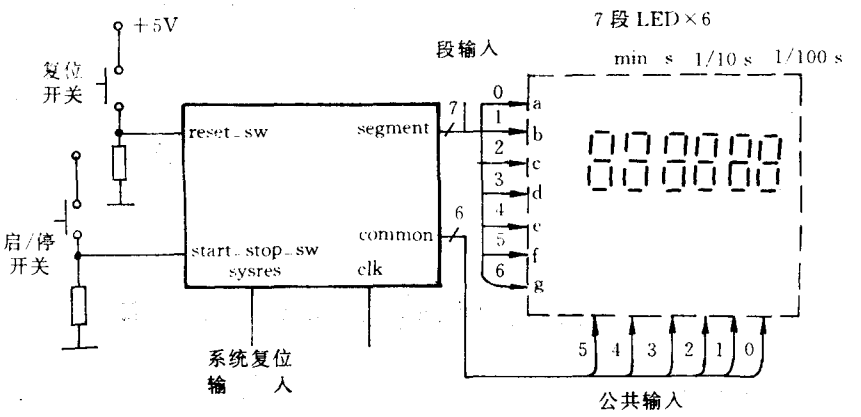


图 9-1 1/100 s 计时器设想结构

9.2 1/100 s 计时控制芯片设计

上一节已经介绍了 1/100 s 计时器结构的设想，其核心是要设计一片用于计时控制的 ASIC 芯片，该芯片的输入和输出信号已有明确定义：

(1) 输入信号

- 复位输入——reset_sw；
- 启/停输入——start_stop_sw；
- 系统复位输入——sysres；
- 时钟输入——clk。

(2) 输出信号

- LED7 段输出——Segment(6 TO 0)，共 7 条输出线；
- LED 公共端输出——common(5 TO 0)，共 6 条输出线。

知道了输入输出，下一步根据其功能就可以设计该芯片了。

9.2.1 计时控制芯片的结构

为了便于描述，现将整个计时控制芯片分成 5 大子模块：键输入子模块(keyin)、时钟产生子模块(clkgen)、控制子模块(ctrl)、定时计数子模块(cntblk)和显示子模块(disp)。各

子模块之间的信号连接关系如图 9 - 2 所示。

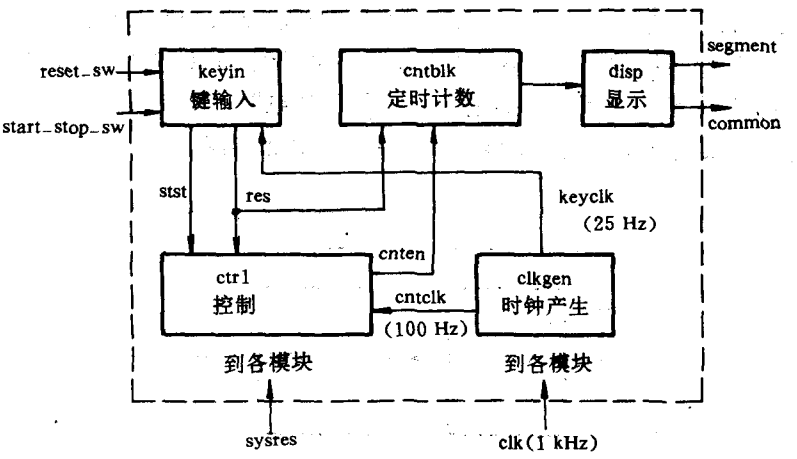


图 9 - 2　计时控制芯片各子模块之间的信号连接关系

1. 键输入子模块(keyin)

该子模块的输入信号是 reset_sw、start_stop_sw 和 keyclk。输出信号是 res(复位脉冲输出)和 stst(启/停脉冲输出)。每按一下 reset_sw 开关的按钮，res 输出端将输出一个脉冲宽度为一个时钟周期(1 ms)的复位脉冲。每按一下 start_stop_sw 开关的按钮，stst 输出端将输出一个脉冲宽度为 1 ms 的启/停控制脉冲。这两种脉冲产生电路的结构是完全一致的，唯一应注意的是两种电路都应采取防抖动措施。一般键抖动的时间小于 20 ms，因此为了去除键的机械抖动对输出的影响，各输入输出信号应保持如图 9 - 3 所示的定时关系。

在图 9 - 3 中 keyclk 是一个脉冲宽度为 1 ms，周期为 10 ms 的时钟脉冲，rs0 和 rs1 是为去除抖动而引入的状态控制信号。图 9 - 3 所示的是复位开关输入信号 reset_sw 和复位输出信号 res 的定时关系，如果用 start_stop_sw 替代 reset_sw，用 stst 替代 res，用 stst0 替代 res0，用 stst1 替代 res1，这样就可以立刻得到启/停开关输入和启/停脉冲输出之间的定时关系。

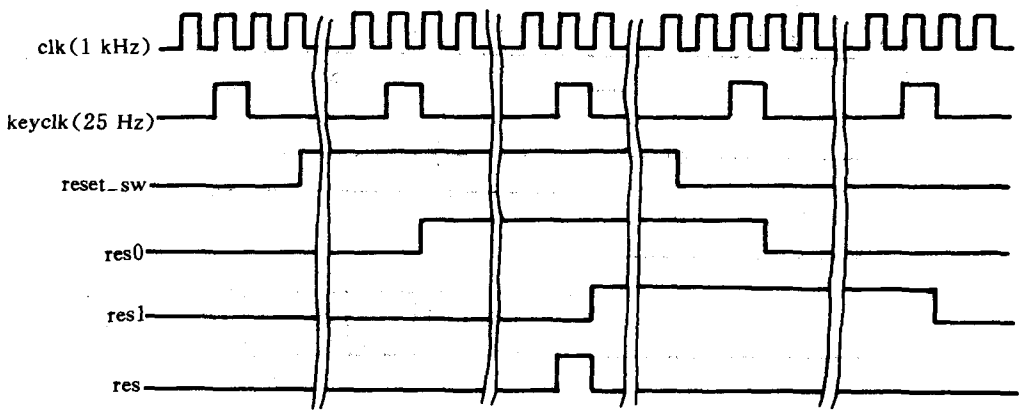


图 9 - 3　去除抖动的输入、输出信号关系

2. 时钟产生子模块(clkgen)

时钟产生子模块的输入信号是 1 kHz 的时钟信号；输出信号是 25 Hz 的 keyclk 和用于定时计数的 100 Hz 计数脉冲 cntclk。1 kHz 时钟信号经 10 次分频后得到作为计数脉冲 cntclk 输出的 100 Hz 时钟脉冲信号，再经 4 次分频即可得到 25 Hz 的 Keyclk 输出。由此可知，clkgen 子模块实际上是一个用计数器进行分频的分频电路，其结构和输入输出信号关系如图 9 - 4 所示。为实现严格的同步，该模块采用同步计数电路。

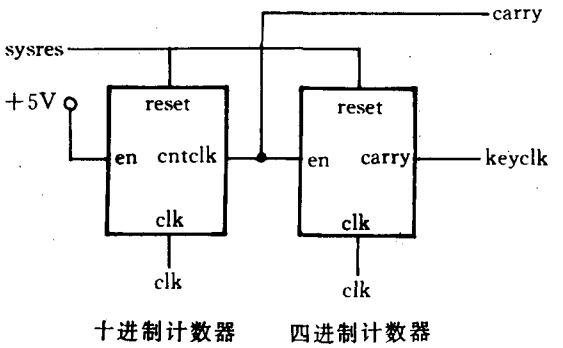


图 9 - 4　时钟产生子模块结构和输入输出信号关系

3. 控制子模块(ctrl)

控制子模块的输入信号是键输入子模块输出的复位脉冲信号 res、启/停控制脉冲 stst 和时钟产生子模块输出的 cntclk；其输出信号是计数允许信号 cnten，它用于控制计数子模块的计数工作。计时器工作时，cnten 端输出 100 Hz 的计数允许脉冲，计时器停止工作时 cnten 端输出低电平。由此可见，控制子模块是根据计时器的工作状态，控制是否输出计数允许脉冲的电路，其输入信号和输出信号的定时关系如图 9 - 5 所示。

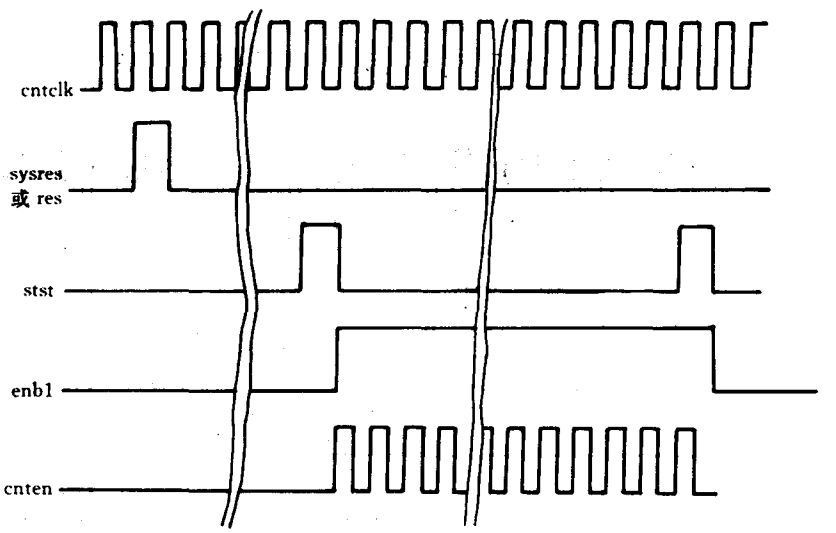


图 9 - 5　控制子模块输入信号和输出信号的实时关系

图 9－5 中的 enb1 信号是一个闸门控制信号，用来选通定时允许计数信号 cntclk。

4. 定时计数子模块(cntblk)

该子模块的输入信号是复位脉冲信号 sysres、res 和定时计数脉冲 clk 和计数允许信号 cnten，其输出信号是：

min10——分十位信号；

min——分个位信号；

sec10——秒十位信号；

sec——秒个位信号；

secl_10——1/10 s 位信号；

secl_100——1/100 s 位信号；

该模块是一个定时计数器，用来产生要显示的 6 位计时信息。

定时计数器的结构如图 9－6 所示。它由 6 个十进制计数器和两个六进制计数器串联连接而组成，用来产生 6 位时间信息。同理，这些计数器也都是同步计数器。该计数器由 sysres 和 res 信号复位。

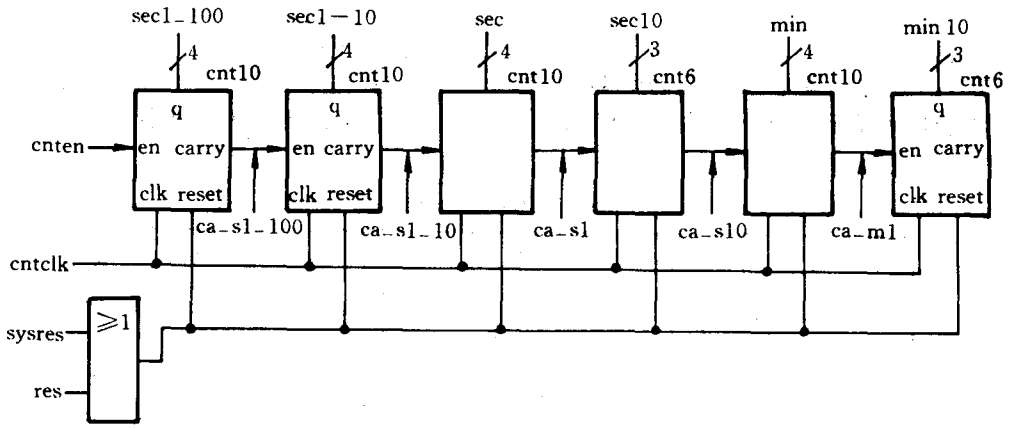


图 9－6　定时计数器结构

5. 显示子模块

显示子模块的输入信号来自定时计数子模块的输出，它们是 min10，min，sec10，sec，secl_10，secl_100 以及 clk 和 sysres。输出信号是 segment 和 common，用来驱动 6 个 7 段 LED 数码显示管。显示子模块的结构框图及信号连接关系如图 9－7 所示。

clk 是六进制计数器的计数脉冲，在该计数脉冲驱动下，六进制计数器的 6 个状态 000B～101B 按顺序循环变化，其输出为 comcnt。该 comcnt 经译码电路 com_dec 译码产生选通 6 位数码管之一的 common 信号。与此同时，comcnt 还作为数据选择电路 digit_set 的选择信号，选择对应位的数据。例如，当 comcnt＝000B时，common 的输出为 000001B，经 com_dec 译码，最低位(1/100 秒位)的 7 段 LED 数码管被选通。同时，经选择器选择，数据 secl_100(1/100 s 位数据)再经段译码将数据送至 7 段数码管的 7 个段上，从而在 secl_100 位上显示出一个对应的数字。由于六进制计数器循环计数，因此每个位的显示时

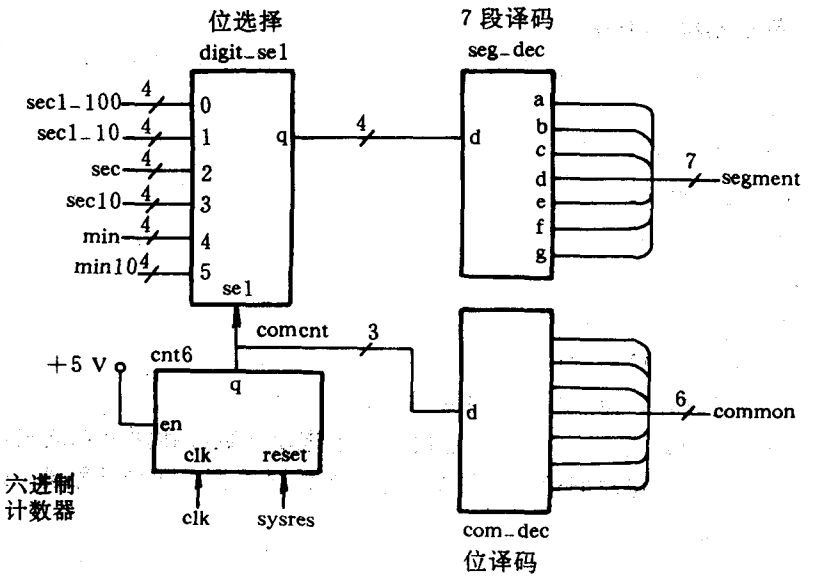


图 9－7　显示子模块结构框图及信号连接关系

间约为 0.166 ms，刷新频率为 166 Hz。这样在 6 个 7 段 LED 数码管上就可以看到一个稳定的数字时间显示。comcnt 的取值与 com_dec 和 digit_set 输出的关系如表 9－1 所示。7 段显示关系表如图 9－8 所示。

表 9－1　com_dec 和 digit_set 输出与 comcnt 关系

comcnt	common	digit_set 输出 (q)
000B	000001B	secl_100
001B	000010B	secl_10
010B	000100B	sec
011B	001000B	sec10
100B	010000B	min
101B	100000B	min10

上面介绍了计时控制芯片的各个子模块的构成及其承担的功能，并规定了相互连接的信号关系。需要注意的是，计时控制芯片的输入信号 sysres 和 clk 都将作为 5 个子模块的输入信号，这一点在图 9－4 已标明，而在各子模块中均省略了说明。

9.2.2　计时控制芯片的包集合 Package_p_stop_watck

在用 VHDL 语言设计硬件系统时，首先要定义对象的各种类型，特别是用户自定义的一些数据类型，都应事先作出定义。另外，在后面设计中所要用到的公共函数，过程等也应事先给出，以备具体设计各模块时能方便地调用。

数字 \ 段	g	f	e	d	c	b	a
0	0	1	1	1	1	1	1
1	0	0	0	0	1	1	0
2	1	0	1	1	0	1	1
3	1	0	0	1	1	1	1
4	1	1	0	0	1	1	0
5	1	1	0	1	1	0	1
6	1	1	1	1	1	0	1
7	0	1	0	0	1	1	1
8	1	1	1	1	1	1	1
9	1	1	0	1	1	1	1

图 9-8　7段显示关系表

这些内容都应包含在计时控制芯片的包集合中。下面分析一下计时控制芯片的结构，以便用函数 FUNCTION 和 PROCEDURE 来描述其中某些专用的或公用的功能。

1. 可用函数描述的功能

由函数描述语句 FUNCTION 的说明可知，在用函数语句描述电路时可以有多个输入参数，但只能有一个输出参数。本例中满足这样要求的功能模块为：显示数据选择模块、7段译码模块和显示公用端输出译码模块。这样就可以分别编写出 digit_sel，seg_dec，com_dec 三个函数。

2. 可用过程描述的功能

由过程描述语句 PROCEDURE 的说明可知，在用过程语句描述电路时，可以有多个输入参数和多个输出参数。只要在调用过程时按规定输入和输出参数赋值，就可以实现参数的输入和输出。用过程描述的电路部分通常是各模块中可共用的部分或重复使用的部分，这样可以简化编程。

在本例中，十进制计数器和六进制计数器是公用的或重复使用的电路，因此这两个电路可以分别用过程来描述。但是，考虑到时序关系和实际仿真结果，十进制计数器和六进制计数器都用基本实体来描述，在使用这些计数器的地方用 COMPONENT 语句和 PORT MAP 语句进行映射连接。

3. 各函数输入参数说明

(1) 函数 digit_sel 的输入参数

如图 9-7 所示，digit_sel 电路的输入参数为：

secl_100；

secl_10；

sec；

min；

min10。

(2) 函数 com_dec 的输入参数

如图 9－7 所示，com_dec 电路的输入参数为 comcnt。

(3) 函数 seg_dec 的输入参数

如图 9－7 所示，seg_dec 电路的输入参数为 digit。

4. 包集合 Package p_stop_watck 描述

包集合 package p_stop_watck 的清单如例 9－1 所示。由第 2 章介绍已经知道，包集合由包集合定义部分和包集合体两大部分构成。在包集合定义部分对函数 digit_sel，seg_dec 和 com_dec 进行了定义，而在包集合体部分对各函数的功能进行了详细的描述。这样，该包集合中的函数就可以在实体所对应的构造体设计中引用。但是，在引用前应用一条说明语句 USE WORK. p_stop_watck. ALL 进行说明。

【例 9－1】

```
LIBRARY IEEE;
USE IEEE. STD_LOGIC_1164. ALL;
USE IEEE. STD_LOGIC_ARITH. ALL;
PACKAGE p_stop_watch IS

  FUNCTION digit_sel
          (comcnt:STD_ULOGIC_VECTOR;
            min10:STD_ULOGIC_VECTOR;
              min:STD_ULOGIC_VECTOR;
          sec10:STD_ULOGIC_VECTOR;
            sec:STD_ULOGIC_VECTOR;
          secl1:STD_ULOGIC_VECTOR;
          secl2:STD_ULOGIC_VECTOR;
          )
                RETURN STD_ULOGIC_VECTOR;
  FUNCTION seg_dec(digit:STD_ULOGIC_VECTOR)
                RETURN STD_ULOGIC_VECTOR;
  FUNCTION com_dec(comcnt:STD_ULOGIC_VECTOR)
                RETURN STD_ULOGIC_VECTOR;

END p_stop_watch;
PACKAGE BODY p_sotp_watch IS
  FUNCTION digit_sel
    (comcnt:STD_ULOGIC_VECTOR;
          min10:STD_ULOGIC_VECTOR;
            min:STD_ULOGIC_VECTOR;
```

```
        sec10:STD_ULOGIC_VECTOR;
          sec:STD_ULOGIC_VECTOR;
        secl1:STD_ULOGIC_VECTOR;
        secl2:STD_ULOGIC_VECTOR
    )
        RETURN STD_ULOGIC_VECTOR IS
VARIABLE digit_sel_tmp:STD_ULOGIC_VECTOR(3 DOWNTO 0):="0000";
VARIABLE comcnt_s:STD_ULOGIC_VECTOR(2 DOWNTO 0);
BEGIN
  comcnt_s:=comcnt;
  CASE comcnt_s IS;
      WHEN"000"=>digit_sel_tmp:=sec12;
      WHEN"001"=>digit_sel_tmp:=secl1;
      WHEN"010"=>digit_sel_tmp:=sec;
      WHEN"011"=>digit_sel_tmp(2 DOWNTO 0):=sec10;
      WHEN"100"=>digit_sel_tmp:=min;
      WHEN"101"=>digit_sel_tmp(2 DOWNTO 0):=min10;
      WHEN OTHERS=>digit_sel_tmp:="XXXX";
END CASE;
      RETURN digit_sel_tmp;
END digit_sel;
FUNCTION seg_dec(digit:STD_ULDGIC_VECTOR)
      RETURN STD_ULOGIC_VECTOR IS
VARIABLE seg_dec_tmp:STD_ULOGIC_VECTOR(6 DOWNTO 0);
VARIABLE digit_s:STD_ULOGIC_VECTOR(3 DOWNTO 0);
BEGIN
  digit_s:=digit;
  CASE digit_s IS
      WHEN"0000"=>seg_dec_tmp:="0111111";
      WHEN"0001"=>seg_dec_tmp:="0000110";
      WHEN"0010"=>seg_dec_tmp:="1011011";
      WHEN"0011"=>seg_dec_tmp:="1001111";
      WHEN"0100"=>seg_dec_tmp:="1100110";
      WHEN"0101"=>seg_dec_tmp:="1101101";
      WHEN"0110"=>seg_dec_tmp:="1111101";
      WHEN"0111"=>seg_dec_tmp:="0100111";
      WHEN"1000"=>seg_dec_tmp:="1111111";
      WHEN"1001"=>seg_dec_tmp:="1101111";
      WHEN OTHERS=>seg_dec_tmp:="XXXXXXX";
  END CASE;
      RETURN seg_dec_tmp;
END seg_dec;
```

```
FUNCTION com_dec(comcnt:STD_ULOGIC_VECTOR)
    RETURN STD_ULOGIC_VECTOR IS
VARIABLE com_dec_tmp:STD_ULOGIC_VECTOR(5 DOWNTO 0);
VARIABLE comcnt_s:STD_ULOGIC_VECTOR(2 DOWNTO 0);
BEGIN
  comcnt_s:=comcnt;
  CASE comcnt_s IS
      WHEN"000"=>com_dec_tmp:="000001";
      WHEN"001"=>com_dec_tmp:="000010";
      WHEN"010"=>com_dec_tmp:="000100";
      WHEN"011"=>com_dec_tmp:="001000";
      WHEN"100"=>com_dec_tmp:="010000";
      WHEN"101"=>com_dec_tmp:="100000";
      WHEN OTHERS=>com_dec_tmp:="XXXXXX";
  END CASE;
      RETURN com_dec_tmp;
  END com_dec;
END P_stop_watch;
```

5. 各函数描述说明

(1) 函数 digit_sel

该函数功能是根据六进制计数器的 6 个状态的输出值，来选择对应显示位的数据，作为送至 7 段译码器的输入端数据。该函数的返回值即是所选择显示位的数据。例如，当 comcnt＝000 时，所选择的数据为 secl_100；而当 comcnt＝101 时，所选择的数据为 min10。函数中使用 CASE 语句来实现多输入单输出，这是典型的选择器描述。因此，函数 digit_sel 是实现一个6输入单输出的选择器功能的描述。

在函数语句中 CASE 后跟的量应该是变量，故此，在程序中要用一条“comcnt_s:＝comcnt;”这样一条语句。在函数语句中只能定义变量而不能定义信号量。另外，由图 9－7 可以看到选择器的输出是 4 位，而选择器的输入既有 4 位的数据也有 3 位的数据(如 sec10，min10 均为 3 位)，怎样实现将 3 位数据值正确地赋给4位的数据输出端呢？在这里使用“digit_sel_tmp(2 DOWNTO 0):＝sec10;”这样一条语句。如果直接使用了“digit_sel_tmp:＝sec10”语句，那就会产生错误的结果，其原因请读者自己分析。

(2) 函数 seg_dec

该函数实现段译码功能，也就是将显示的数据值转换成 7 段 LED 数码管上应辉亮的显示图形。在该函数中同样使用 CASE 语句来实现 7 段译码。众所周知，选择器同样也可以用作译码器，所不同的是其输入都是事先确定的固定数值。该函数返回值是译码得到的 7 段图形码。

(3) 函数 com_dec

该函数根据六进制计数器状态，将其译码成 6 位显示位选择码。例如，当 comcnt＝000 时，其译码值为000001，也就是选中最低位 1/100 秒位的数码管，允许显示该位的数据值；当 comcnt＝101 时，其译码值为 100000，也就是选中最高位——分十位的数码管显

示数据。该函数中仍采用 CASE 语句来实现对译码器的描述。实际上，无论是译码器还是选择器，同样也可以用 IF 语句来描述。这样，函数 com_dec 可以改成例 9 - 2 形式。

【例 9 - 2】

```
FUNCTION com_dec(comcnt:STD_LOGIC_VECTOR)
   RETURN STD_LOGLC_VECTOR IS
        VARIABLE com_dec_tmp:STD_LOGIC_VECTOR(5 DOWNTO 0)
BEGIN
          IF(comcnt:="000")  THEN
            com_dec_tmp:="000001";
          ELSIF(comcnt="001")  THEN
            com_dec_tmp:="000010";
          ELSIF(comcnt="010")  THEN
            com_dec_tmp:="000100";
          ELSIF(comcnt="011")  THEN
            com_dec_tmp:="001000";
          ELSIF(comcnt="100")  THEN
            com_dec_tmpo:="010000";
          ELSIF(comcnt="101")  THEN
            com_dec_tmp:="100000";
          ELSE
            com_dec_tmp:="XXXXXX";
          END IF;
          RETURN com_dec_tmp;
END com_dec;
```

9.2.3 基本单元电路描述

前面已经提到，十进制、六进制、四进制计数器是本设计的基本单元电路，在计时控制芯片的各子模块中都有重复应用。为此，将这几个计数器设计成标准的基本设计单元，以元件形式(Component)供各子模块使用。

1. 十进制计数器 cnt10

如图 9 - 6 所示，cnt10 电路是一个十进制计数器电路，它的输入端口为：

en——允许计数；

clk——时钟；

reset——复位。

输出端口为：

q——计数值输出；

carry——进位输出。

这是一个同步先行进位的计数电路，计数时钟 clk 为1 kHz 周期脉冲信号，只有当 en 有效时，计数器才进行计数。cnt10 的描述如例 9 - 3。

【例 9 - 3】

```
LIBRARY IEEE;
```

```
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
ENTITY cnt10 IS
    PORT(reset, en, clk:IN STD_ULOGIC;
         carry:OUT STD_ULOGIC;
         q:OUT STD_ULOGIC_VECTOR(3 DOWNTO 0));
END cnt10;

ARCHITECTURE rtl OF cnt10 IS
        SIGNAL qs:STD_LOGIC_VECTOR(3 DOWNTO 0);
        SIGNAL ca:STD_ULOGIC;
        BEGIN
          PROCESS(clk)
                    VARIABLE q10:INTEGER;
                BEGIN
                  IF(clk'EVENT AND clk='1') THEN
                     IF(reset='1') THEN
                        q10:=0;
                     ELSIF(en='1') THEN
                       IF(q10=9) THEN
                         q10:=0
                         ca<='0';
                       ELSIF(q10=8) THEN
                         q10:=q10+1;
                         ca<='1';
                       ELSE
                         q10:=q10+1;
                         ca<='0';
                       END IF;
                     END IF;
                  END IF;
               qs<=CONV_STD_LOGIC_VECTOR(q10, 4);
               q<=TO_STDULOGICVECTOR(qs);
            END PROCESS;
          PROCESS(ca, en)
          BEGIN
               carry<=ca AND en;
          END PROCESS;
END rtl;
```

描述该十进制计数器的构造体由两个进程组成。从逻辑上来看，第二个进程是可以省去的，只要将该进程中的“carry<=ca AND en;”语句放到第一个进程中去执行就行了。但是这样做的话，在多级十进制计数器级联时定时会发生错误，故分成两个进程来描述。另

外，在该描述中还采用了一个标准的转换函数，其中：

CONV_STD_LOGIC_VECTOR(q10,4)是从整数转换成 STD_LOGIC_VECTOR 类型数据的函数；

TO_STDULOGICVECTOR(qs)是从 STD_LOGIC_VECTOR 转换成 STD_ULOGIC_VECTOR 类型数据的函数。

在 STD_LOGIC_1164(附录 C)中有如下类型说明：

```
FUNCTION Resolved(S:Std_Ulogic_vector)
                        RETURN STD_ULOGIC;
SUBTYPE Std_logic IS Resolved Std_Ulogic;
```

由此可知，STD_ULOGIC_VECTOR 是不包含判决功能的数据类型，而 STD_LOGIC_VECTOR 是含判决功能的数据类型，即它能描述多源连接。

2. 六进制计数器 cnt6 和四进制计数器 cnt4

六进制计数器 cnt6 和四进制计数器 cnt4 的描述结构完全相同，所不同的仅仅是计数器的状态数。cnt6 和 cnt4 的描述如例 9－4 和例 9－5 所示。

【例 9－4】

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

ENTITY cnt6 IS
   PORT (reset, en, clk:IN STD_ULOGIC;
         carry:OUT STD_ULOGIC;
         q:OUT STD_ULOGIC_VECTOR(2 DOWNTO 0);
END cnt6;

ARCHITECTURE rtl OF cnt6 IS
 SIGNAL qs:STD_LOGIC_VECTOR(2 DOWNTO 0);
 SIGNAL ca:STD_ULOGIC;
 BEGIN
PROCESS(clk)
     VARIABLE q6:INTEGER;
BEGIN
 IF (clk'EVENT ADN clk='1')  THEN
     IF(reset='1')  THEN
        q6:=0;
     ELSIF (en='1')  THEN
        IF(q6=5)  THEN
          q6:=0;
          ca<='0';
        ELSIF(q6=4)  THEN
          q6:=q6+1;
```

```
            ca<='1';
          ELSE
            q6:=q6+1;
            ca<='0';
          END IF;
      END IF;
  END IF;
      qs<=CONV_STD_LOGIC_VECTOR(q6, 3);
      q<=TO_STDULOGICVECTOR(qs);
  END PROCESS;
       PROCESS(ca,en)
         BEGIN
              carry< =ca AND en;
         END PROCESS;
  END rtl;
```

【例 9-5】

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

  ENTITY cnt4 IS
     PORT (reset, en, clk:IN STD_ULOGIC;
           carry:OUT STD_ULOGIC;
           q:OUT STD_ULOGIC_VECTOR(1 DOWNTO 0);
  END cnt4;
  ARCHITECTURE rtl OF cnt4 IS
         SIGNAL qs:STD_LOGIC_VECTOR(1 DOWNTO 0);
         SIGNAL ca:STD_ULOGIC;
         BEGIN
         PROCESS(clk)
                 VARIABLE q4:INTEGER;
                 BEGIN
                   IF (clk'EVENT AND clk='1')  THEN
                     IF(reset='1')  THEN
                         q4:=0;
                     ELSIF(en='1')  THEN
                         IF(q4=3)  THEN
                           q4:=0;
                           ca<='0';
                         ELSIF(q4=2)  THEN
                           ca<'1';
                           q4:=q4+1;
```

```
                    ELSE
                      q4:=q4+1;
                      ca<='0';
                    END IF;
                  END IF;
              END IF;
            qs<=CONV_STD_LOGIC_VECTOR(q4, 2);
            q<=TO_STDULOGICVECTOR(qs);
          END PROCESS;
          PROCESS(clk)
            BEGIN
                IF (clk'EVENT AND clk='1')  THEN
                  carry<=ca AND en;
              END IF;
          END PROCESS;

END rtl;
```

9.2.4 计时控制芯片实体 stop_watch 描述

在 VHDL 语言中，系统接口是由实体来描述的。系统接口包括输入端口信号、输出端口信号、输入输出端口的双向信号以及需要传入实体的参数等。系统描述的第一步是建立系统的实体。根据计时控制芯片的输入输出信号要求及实体描述规定，计时控制芯片的实体描述如例 9 - 6 所示。

【例 9 - 6】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE WORK.P_stop_watch.all;
ENTITY stop_watch IS
   PORT(sysres, reset_sw, start_stop_sw, clk,
   dispen, enclk:IN STD_ULOGIC;
   segment:OUT STD_ULOGIC_VECTOR(6 DOWNTO 0);
   common:OUT STD_ULOGIC_VECTOR(5 DOWNTO 0));
END stop_watch;
```

计时控制芯片的实体名为 stop_watch，实体对输入端口 clk，sysres，reset_sw，start_stop_sw 和输出端口 segment，common 进行了定义和说明。另外还对输入端口 dispen 和 enclk 作了定义和说明，实际上这两个输入端口是可以不要的，只要将这两个输入在内部接高电平(Vcc)就可以了。这里将它们作为输入端口的目的是为了在仿真时可以在外部进行自由控制而设置的。实体定义前的 4 条说明语句表明，在该实体设计中引用了 IEEE 的标准库和工作库中的包集合 P_stop_watck。

9.2.5 计时控制芯片的构造体描述

如前所述，对于一般系统的设计，其构造体应先进行行为描述和仿真。但是，根据实际工作情况和经验，电路规模超过 10 万门的，应先进行行为描述和仿真，而电路规模小于 10 万门的，通常直接采用 RTL 描述方式，以简化设计步骤和缩短设计时间，故本例的构造体就直接采用 RTL 描述。

由计时控制芯片的结构分析得知，该芯片分成 5 个子模块。这 5 个子模块由 5 个独立的元件构成。在形成 stop_watch 构造体时，将这 5 个元件按图 9 - 2 的信号关系连接起来就行了。

整个系统的描述如例 9 - 7。

【例 9 - 7】

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

ENTITY stop_watch IS

  PORT(sysres, reset_sw, start_stop_sw, clk:IN STD_ULOGIC;
       dispen, enclk:IN STD_ULOGIC;
       segment:OUT STD_ULOGIC_VECTOR(6 DOWNTO 0);
       common:OUT STD_ULOGIC_VECTOR(5 DOWNTO 0);

END stop_watch;
ARCHITECTURE rtl OF stop_watch IS
  COMPONENT clkgen
      PORT(sysres, en1, clk:IN STD_ULOGIC;
           cntclk.keyclk:OUT STD_ULOGIC);
  END COMPONENT;
  COMPONENT keyin
     PORT (reset_sw, start_stop_sw, keyclk, clk:IN STD_ULOGIC;
          res, stst: OUT STD_ULOGIC);
  END COMPONENT;
  COMPONENT ctrl
      PORT(sysres, res, stst, cntclk:IN STD_ULOGIC;
                              cnten: OUT STD_ULOGIC);
  END COMPONENT;
  COMPONENT cntblk
     PORT (sysres, cnten, clk, res: IN STD_ULOGIC;
          secl2:OUT STD_ULOGIC_VECTOR(3 DOWNTO 0);
          secl1:OUT STD_ULOGIC_VECTOR(3 DOWNTO 0);
          sec:OUT STD_ULOGIC_VECTOR(3 DOWNTO 0);
          sec10:OUT STD_ULOGIC_VECTOR(2 DOWNTO 0);
```

```
            min:OUT STD_ULOGIC_VECTOR(3 DOWNTO 0);
            min10:OUT STD_ULOGIC_VECTOR(2 DOWNTO 0));
    END COMPONENT;
    COMPONENT disp
        PORT (secl2:IN STD_ULOGIC_VECTOR(3 DOWNTO 0);
            secl1:IN STD_ULOGIC_VECTOR(3 DOWNTO 0);
              sec:IN STD_ULOGIC_VECTOR(3 DOWNTO 0)
            sec10:IN STD_ULOGIC_VECTOR(3 DOWNTO 0)
              min:IN STD_ULOGIC_VECTOR(3 DOWNTO 0);
            min10:IN STD_ULOGEC_VECTOR(3 DOWNTO 0);
            sysres, clk, dispen:IN STD_ULOGIC;
            segment:OUT STD_ULOGIC_VECTOR(6 DOWNTO 0);
            common:OUT STD_ULOGIC_VECTOR(5 DOWNTO 0);
    END COMPONENT;
    SIGNAL cntclk_s, keyclk_s, stst_s, cnten_s, res_s:STD_ULOGIC;
    SIGNAL secl2_s:STD_ULOGIC_VECTOR(3 DOWNTO 0);
    SIGNAL secl1_s:STD_ULOGIC_VECTOR(3 DOWNTO 0);
    SIGNAL sec_s:STD_ULOGIC_VECTOR(3 DOWNTO 0);
    SIGNAL min_s:STD_ULOGIC_VECTOR(3 DOWNTO 0);
    SIGNAL sec10_s:STD_ULOGIC_VECTOR(3 DOWNTO 0);
    SIGNAL sec10_sl:STD_ULOGIC_VECTOR(2 DOWNTO 0);
    SIGNAL min10_s:STD_ULOGIC_VECTOR(3 DOWNTO 0);
    SIGNAL min10_sl:STD_ULOGIC_VECTOR(2 DONTO 0);
    BEGIN
        u0:clkgen PORT MAP(sysres, enclk, clk, cntclk_s, keyclk_s);
        u1:keyin PORT MAP(reset_sw, start_stop_sw, keyclk_s, clk,
                          res_s, stst_s);
        u2:ctrl PORT MAP(sysres, res_s, stst_s, cntclk_s, cnten_s);
        u3:cntblk PORT MAP(sysres, cnten_s, clk, res_s, secl2_s, secl1_s,
                           sec_s, sec10_sl, min_s, min10_sl);
        u4:disp PORT MAP(secl2_s, secl1_s, sec_s, sec10_s, min_s,
                         min10_s, sysres, clk, dispen, segmet, common);
        sec10_s<='0'& sec10_sl;
        min10_s<='0'& min10_sl;
END rtl;
```

需要说明的是，在程序中 secl_100 用 secl2 说明，secl_10 用 secl1 说明。

9.2.6 各子模块描述说明

各子模块用独立实体构成，作为元件被系统所引用。

1. Keyin 子模块

该模块的描述是为了产生如图 9－3 所示的单个复位脉冲 res 和启/停脉冲 stst。整个功能模块用两个进程语句描述，如图 9－8 所示。

【例 9 - 8】

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

ENTITY keyin IS
   PORT(reset_sw, start_stop_sw, keyclk, clk:IN STD_ULOGIC;
        res, stst:OUT STD_ULOGIC);
END keyin;

ARCHITECTURE rtl OF keyin IS
   SIGNAL res0, res1, stst0, stst1:STD_ULOGLC;
BEGIN
   PROCESS(keyclk)
        BEGIN
            IF (keyclk 'EVENT AND keyclk='0') THEN
                res1<=res0;
                res0<=reset_sw;
                stst1<=stst0;
                stst0<=start_stop_sw;
            END IF;
     END PROCESS;
  PROCESS(res0, res1, stst0, stst1)
        BEGIN
            res<=keyclk AND res0 AND (NOT res1);
            stst<=keyclk AND stst0 AND (NOT stst1);
     END PROCESS;
END rtl;
```

第一个进程的敏感量为 keyclk，现以复位开关为例说明一下 res 的形成过程。参照图 9 - 3，在复位开关按下以前，reset_sw 为“0”，在 keyclk 下降沿有效时进程被启动。此时 res0 为“0”，res1也为“0”。现在 reset_sw 开关按下，reset_sw 变为“1”在 keyclk 的下降沿有效时 res0='1'，res1='1'。在出现第二个 keyclik 脉冲时，进程再次被执行，从而使 res0='1'，res1='1'。

第二个进程的敏感量为 clk，在每个 clk 的上升沿到来时，keyclk，res0和$\overline{res1}$相“与”后传送给 res，由图 9 - 3 定时关系可知，reset_sw 变为“1”后的第二个 keyclk 将被提取作为 res 的复位脉冲，每按一次 reset_sw 开关只能产生一个 res 复位脉冲。

同理，每按一次 start_sotp_sw 也只能产生一个宽度为 1 ms 的 stst 脉冲。

该模块的硬件电路实现如图 9 - 9 所示。

2. ctkgen 子模块

该模块的功能是产生100 Hz 的计时允许信号 cntclk 和 25 Hz 的宽度为 1 ms 的键输入时钟信号 keycek。

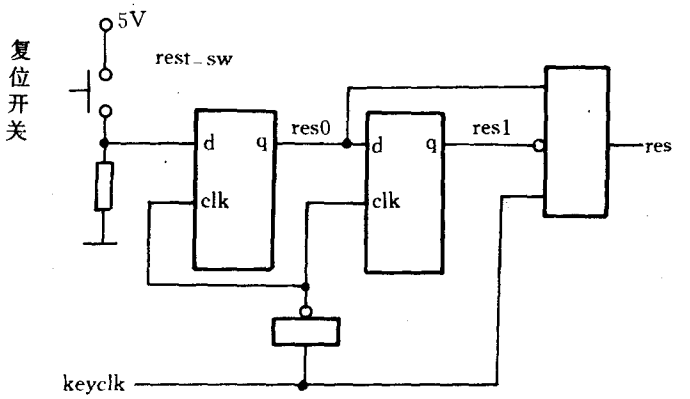


图 9－9　keyin 模块的硬件电路实现

该模块由两个元件十进制计数器 cnt10 和四进制计数器 cnt4 串接构成。计数时钟脉冲为 1 kHz 的 clk 信号，经 10 次分频得到 100 Hz 的 cntclk，再经4次分频得到 25 Hz 的 keyclk 信号。ctkgen 的 VHDL 描述如例 9－9 所示。

【例 9－9】

```
LBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;

ENTITY clkgen IS
 PORT(sysres, en1, clk:IN STD_ULOGIC);
      cntclk, keyclk:OUT STD_ULOGIC);
END clkgen;

ARCHITECTURE rt1 OF clkgen IS
      COMPONENT cnt10
       PORT(reset, en, clk:IN STD_ULOGIC;
                  carry :OUT STD_ULOGIC;
                      q:OUT STD_ULOGIC_VECTOR(3 DOWNTO 0));
      END COMPONENT;
      COMPONENT; cnt4
       PORT(reset, en, clk:IN STD_ULOGIC;
                  carry :OUT STD_ULOGIC;
                      q:OUT STD_ULOGIC_VECTOR(1 DOWNTO 0));
      END COMPONENT;
      SIGNAL cntclk_s:std_ULOGIC;
BEGIN
 u0:cnt10 PORT MAP(sysres, en1, clk, cntclk_s);
```

```
    u1:cnt4 PORT MAP(sysres, cntclk_s, clk, keyclk);
    cntclk<=cntclk_s;
END rtl;
```

3. ctrl 子模块

该模块的功能是产生计时计数模块的计数允许信号 cnten。用一个进程语句来描述，敏感量为 stst。模块中定义了一个信号量 enb1，它起开关作用。当 enb1='1'时，允许 cntclk 从 cnten 端输出；当 enb1='0'时，禁止 cntclk 从 cnten 端输出，从而对计时计数实现启/停控制。另外，enb1是一个乒乓开关，stst 每一次有效，其状态就会向相反状态转换一次。ctrl 的 VHDL 语言描述如例 9 - 10 所示。

【例 9 - 10】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
ENTITY ctrl IS
   PORT(sysres, res, stst, cntclk:IN STD_ULOGIC;
                cnten:OUT STD_ULOGIC);
END ctrl;

ARCHITECTURE rtl OF ctrl IS
SIGNAL enb1:STD_ULOGIC;
BEGIN
    PROCESS(stst, sysres, res)
    BEGIN
            IF (sysres='1' OR res='1')  THEN
                enb1<='0';
            ELSIF( stst'EVENT AND stst='1')  THEN
                enb1<=NOT enb1;
            END IF;
    END PROCESS;
    cnten<=enb1 AND cntclk;
END rtl;
```

4. cntblk 子模块

该模块的功能是实现计时计数，它由 4 个十进制计数器和两个六进制计数器串接而成。该模块将输出 1/100 秒位，1/10 秒位，秒个位，秒十位，分个位，分十位的计时数值。十进制计数器和六进制计数器，以元件形式被 cntblk 模块所使用，如例 9 - 11 所示。

【例 9 - 11】

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

ENTITY cntblk IS
   PORT (sysres, clk, cnten, res:IN STD_ULOGIC;
```

```
        secl2:OUT STD_ULOGIC_VECTOR(3 DOWNTO 0);
        secl1:OUT STD_ULOGIC_VECTOR(3DOWNTO 0);
        sec:OUT STD_ULOGIC_VECTOR(3 DOWNTO 0);
        secl0:OUT STD_ULOGIC_VECTOR(2 DOWNTO 0);
        min:OUT STD_ULOGIC_VECTOR(3 DOWNTO 0);
        min10:OUT STD_ULOGIC_VECTOR(2 DOWNTO 0);
END cntblk;

ARCHITECTURE rtl OF cntblk IS
    COMPONENT cnt6
    PORT (reset, en, clk:IN STD_ULOGIC;
          carry:OUT STD_ULOGIC;
          q:OUT STD_ULOGIC_VECTOR(2 DOWNTO 0));
    END COMPONENT;
    COMPONENT cnt10
        PORT (reset, en, clk:IN STD_ULGOIC;
              carry:OUT STD_ULOGIC;
              q:OUT STD_ULOGIC_VECTOR(3 DOWNTO 0));
    END COMPONENT;
  SIGNAL ca_sl_10, ca_sl, ca_s10:STD_ULOGIC;
  SIGNAL clk_s, cnten_s, res_s, :STD_ULOGIC;
  SIGNAL ca_ml, ca_m10, ca_sl_100:STD_ULOGIC;
BEGIN
  res_s<=sysres OR res;
  cnten_s<=cnten;
  u0:cnt10 PORT MAP(res_s, cnten_s, clk, ca_sl_100, secl2);
  u1:cnt10 PORT MAP(res_s, ca_sl_100, clk, , ca_sl_10, secl1);
  u2:cnt10 PORT MAP(res_s, ca_sl_10, clk, ca_sl, sec);
  u3:cnt6 PORT MAP(res_s, ca_sl, clk, ca_s10, secl0);
  u4:cnt10 PORT MAP(res_s, ca_s10, clk, ca_ml, min);
  u5:cnt6 PORT MAP(res_s, ca_ml, clk, ca_m10, min10);
  END rtl;
```

5. disp 子模块

该模块的功能是为 6 个 7 段 LED 显示数码管提供显示数据和对应位显示的同步控制，使得在 LED 数码管上获得稳定、正确的计时时间显示。同样，它由一个进程和 3 个函数语句组成。

在 disp 构造体中，首先是一个六进制计数进程，进程的敏感量为 clk。其输出作为 6 个显示位数据输入选择控制和显示位译码器的控制输入。这样六进制循环计数，就可以使 6 个显示数码管循环显示对应的时间值。计时计数模块输出的计时数据，为了在 7 段 LED 上进行显示需要段译码，这一功能通过调用段译码函数 seg_dec 来实现。公共端的位显示译码输出则调用 com_dec 函数来实现；对应显示位显示的数据则通过调用 digit_sel 函数来

实现。具体描述参见例 9 - 12。

【例 9 - 12】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE WORK.p_stop_watch.ALL;
ENTITY disp IS
  PORT(secl2:IN STD_ULOGIC_VECTOR(3 DOWNTO 0);
       secl1:IN STD_ULOGIC_VECTOR(3 DOWNTO 0);
         sec:IN STD_ULOGIC_VECTOR(3 DOWNTO 0);
       sec10:IN STD_ULOGIC_VECTOR(3 DOWNTO 0);
         min:IN STD_ULOGIC_VECTOR(3 DOWNTO 0);
       min10:IN STD_ULOGIC_VECTOR(3 DOWNTO 0);
       sysres, clk, dispen:IN STD_ULOGIC;
       segment:OUT STD_ULOGIC_VECTOR(6 DOWNTO 0);
         common:OUT STD_ULOGIC_VECTOR(5 DOWNTO 0));
END disp;

ARCHITECTURE rtl OF disp IS
  SIGNAL selq:STD_ULOGIC_VECTOR(2 DOWNTO 0);
  SIGNAL comcnt_sl:STD_LOGIC_VECTOR(2 DOWNTO 0);
  SIGNAL comcnt:STD_ULOGIC_VECTOR(2 DOWNTO 0);
BEGIN
PROCESS(clk)
    VARIABLE q6:INTEGER;
  BEGIN
    IF (clk'EVENT AND clk='1')THEN
      IF(sysres='1')  THEN
        q6:=0;
      ELSIF( dispen='1')  THEN
        IF(q6=5)THEN
          q6:=0;
        ELSE
          q6:=q6+1;
        END IF;
      END IF;
    END IF;
        comcnt_sl<=CONV_STD_LOGIC_VECTOR(q6, 3);
        comcnt<=TO_STDULOGICVECTOR(comcnt_sl);
 END PROCESS;
  selq<=digit_sel (comcnt, min10, min, sec10, sec,
                   secl1, secl2);
  segment<=seg_dec(selq);
```

```
    common<=com_dec(comcnt);
END rtl;
```

值得注意的是，MAX+plus Ⅱ 并不支持独立的包集合编译单元，因此那些使用包集合体的实体，在编译前应将包集合体与使用包集合的实体编辑在一起作为一个 VHDL 语言文件。例如，在本例中 p_stop_watch 包集就是和实体 disp 编辑在一起，作为 disp 文件的一个组成部分。这样做了以后，MAX+plus Ⅱ 的编译工具才能正常进行编译并得到最后的结果。

第 10 章

微处理器接口芯片设计实例

在第 9 章举了一个计时电路的设计实例，为了便于读者进一步掌握用 VHDL 语言设计实际数字电路的方法，这里再举 3 个较简单的微处理器接口芯片的设计实例。

10.1 可编程并行接口芯片设计实例

凡是学习过微型计算机原理的读者都知道，8255 是典型的可编程并行接口芯片，它广泛地用于各种接口电路中。为使设计的程序不要过于复杂，这里所设计的芯片仅适用于 8255 的“0”型工作模式，即基本的输入输出方式。

10.1.1 8255 的引脚及内部结构

1. 外部引脚

8255 的引脚如图 10－1 所示。它共有 40 条引脚，其中

$D_0 \sim D_7$——双向数据总线，用来传送数据和控制字。

$\overline{RD}$——读信号线，与其它信号线一起实现对 8255 接口的读操作。

$\overline{WR}$——写信号线，与其它信号线一起实现对 8255 的写操作。

$\overline{CS}$——片选信号线，当它为低电平(有效)时，才能选中该 8255 芯片，也才能对 8255 进行操作。

$A_0 \sim A_1$——端口地址选择信号线。8255 有 4 个端口：其中 3 个为输入输出口，一个为控制寄存器端口，具体规定如下：

A_1A_0	选择端口
00	A 口
01	B 口

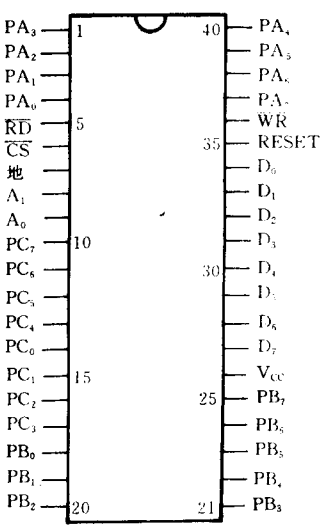


图 10－1 8255 引脚图

10　　C 口

11　　控制寄存器

通常 A_0、A_1 与 CPU 的地址总线 A_0 和 A_1 相连接。

RESET——复位信号输入，高电平有效。复位后，8255 的 A 口、B 口、C 口均被定义为输入。

PA_0～PA_7、PB_0～PB_7、PC_0～PC_7——3 个输入输出端口的引脚，输入输出方向由软件来设定。

2. 内部结构

8255 的内部结构框图如图 10－2 所示。

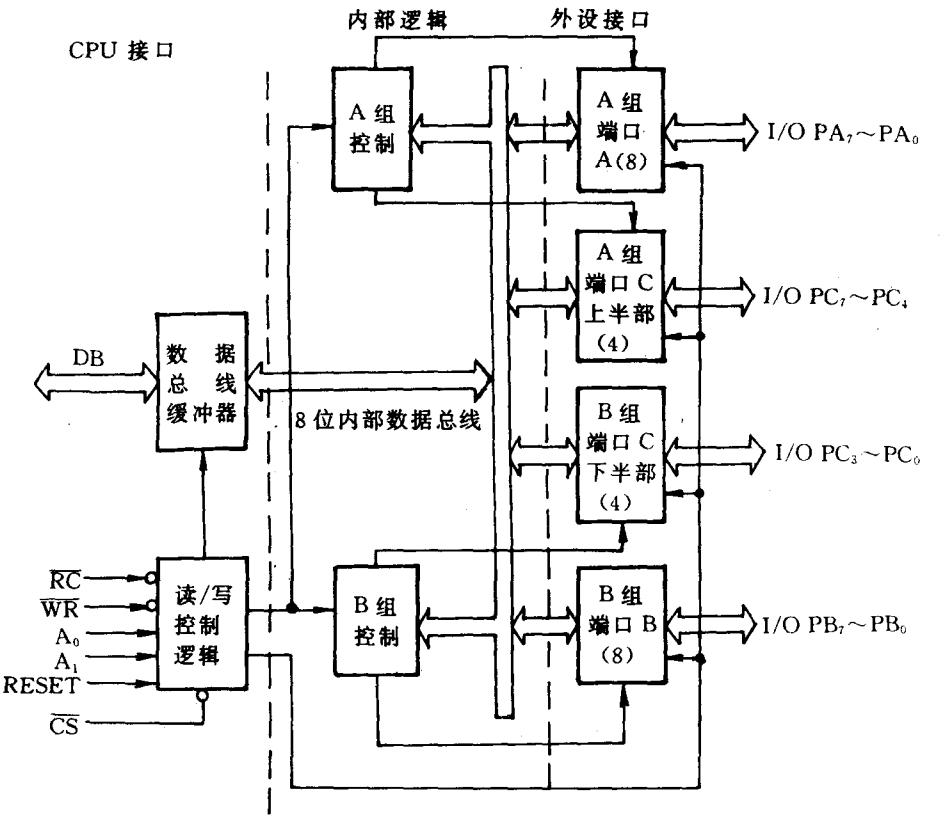


图 10－2　8255 内部结构框图

从图 10－2 可以看到，左边的信号与 CPU 总线相连；而右边的则与外设相连。A、B、C 口的输出均有锁存能力，而输入都没有锁存能力(这一点与原 8255 芯片略有区别)。

为了控制方便，将 8255 的 3 个口分成 A、B 两组。其中 A 组包括 A 口的 8 条线 PA_0～PA_7 和 C 口的高 4 位 PC_4～PC_7；B 组包括 B 口的 8 条线 PB_0～PB_7 和 C 口的低 4 位 PC_0～PC_3。A 组和 B 组都分别由软件编程来加以控制。

10.1.2　8255 的工作方式及其控制字

8255 有 3 种工作方式：即方式 0，方式 1 和方式 2。前面已经提到，为简化设计这里只

设定为方式 0。

1. 8255 的方式 0

在此方式下，A 口的 8 条线，B 口的 8 条线，C 口的高 4 位对应的 4 条线和 C 口低 4 位对应的 4 条线，这 4 部分可分别定义为输入或输出。因为上述 4 部分的输入或输出是可以互相独立来定义的，故它们的输入输出组合可有 16 种。另外，在方式 0 情况下，C 口还有按位的置位和复位的能力，这一点在后述的控制字中详述。

2. 控制字

8255 有很强的功能，其不同功能的实现是通过对控制器写不同控制字来实现的。

8255 有两种控制字：即方式控制字和 C 口位操作控制字。

(1) 方式控制字

8255 方式控制字格式如图 10 - 3 所示。方式控制字的标志是控制字的最高位为“1”，

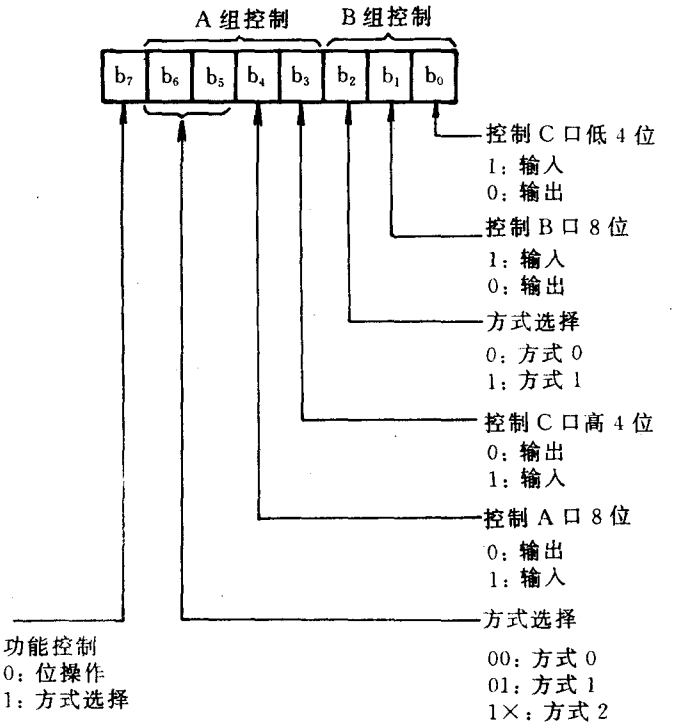


图 10 - 3　8255 的方式控制字

即图中的 b_7 位为“1”。如果现在设定 A 口为输入口、B 口为输出口，C 口的低 4 位输入，C 口的高 4 位为输出，那么控制字的格式应为：

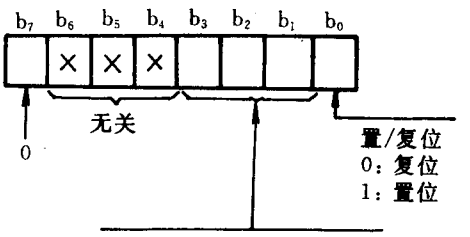
1	0	0	1	0	0	0	1
b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

控制字

控制字的值为 91H。那么将该控制字写向控制寄存器，就会使 8255 处于所设定的工作方式。

(2) C 口位操作控制字

C 口位操作控制字的格式如图 10 - 4 所示。



C 口的位选择

b_3	b_2	b_1	选择
0	0	0	PC_0
0	0	1	PC_1
0	1	0	PC_2
0	1	1	PC_3
1	0	0	PC_4
1	0	1	PC_5
1	1	0	PC_6
1	1	1	PC_7

图 10 - 4　C 口位操作控制字

该控制字和方式控制字的区别在于，此时控制字的最高位(b_7 位)为“0”。用此位作为软开关将控制字写入不同的控制寄存器。例如，当 PC 口作为输出口时，用如下控制字：

0	×	×	×	1	1	0	1

其值为 0DH。将此值送控制寄存器地址就可以使 PC_6 置“1”。

10.1.3　8255 的结构设计

由图 10 - 2 的 8255 内部结构框图可知，该芯片应由 3 种逻辑电路构成：锁存器、组合逻辑电路和三态缓冲器。

1. 锁存器

锁存器用于锁存数据。在 8255 的结构中应定义 7 个锁存器，它们是：

pa_latch——A 口输出锁存器，8 位；

pb_latch——B 口输出锁存器，8 位；

pcl_latch——C 口低 4 位输出锁存器；

pch_latch——C 口高 4 位输出锁存器；

ctrreg——方式控制字寄存器；

bctrreg——C 口位控制字寄存器，4 位；

ctrregF——选择标志寄存器，1 位。

当该标志寄存器为“1”时，数据存入方式控制字寄存器，当它为“0”时，数据的低 4 位存入 C 口控制字寄存器。

2. 三态缓冲器

在 8255 芯片中数据线 D_0～D_7，端口 PA、PB、PC 都可以是双向的。因此，在设计该部

分逻辑与外部接口时，必须是三态的，即这些引脚都应为三态双向引脚。

3. 组合逻辑电路

除上述两类电路外，余下的基本上是选择电路或译码电路。

10.1.4 8255 芯片的 VHDL 语言描述

由于该 VHDL 语言描述的程序模块将在 MAX+plus Ⅱ 的工具上进行编译、综合和仿真，故程序中应采用 RTL 描述方式。

8255 芯片的 VHDL 语言描述程序如例 10 - 1 所示。

【例 10 - 1】

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

ENTITY ppi IS
   PORT(reset, rd, wr, cs, a0, a1:IN STD_ULOGIC;
        pa:INOUT STD_ULOGIC_VECTOR(7 DOWNTO 0);
        pb:INOUT STD_ULOGIC_VECTOR(7 DOWNTO 0);
        pcl:INOUT STD_ULOGIC_VECTOR(3 DOWNTO 0);
        pch:INOUT STD_ULOGIC_VECTOR(3 DOWNTO 0);
        d:INOUT STD_ULOGIC_VECTOR(7 DOWNTO 0));
END ppi;

ARCHITECTURE rtl OF ppi IS
   SIGNAL internal_bus_out:STD_ULOGIC_VECTOR(7 DOWNTO 0);
   SIGNAL internal_bus_in:STD_ULOGIC_VECTOR(7 DOWNTO 0);
   SIGNAL st, ad, flag:STD_ULOGIC_VECTOR(1 DOWNTO 0);
   SIGNAL ctrreg:STD_ULOGIC_VECTOR(7 DOWNTO 0);
   SIGNAL pa_latch, pb_latch, pc_latch:STD_ULOGIC_VECTOR(7 DOWNTO 0);
BEGIN
  PROCESS(rd, cs)
  BEGIN
      st<=ctrreg(3) & ctrreg(0);
      IF (cs='0' AND rd='0') THEN
        IF(a0='0' AND a1='0' AND ctrreg(4)='1') THEN
            internal_bus_in<=(pa);
        ELSIF(a0='1' AND a1='0' AND ctrreg(1)='1') THEN
            internal_bus_in<=pb;
        ELSIF(a0='0' AND a1='1' AND st="01") THEN
            internal_bus_in(3 DOWNTO 0)<=pcl(3 DOWNTO 0);
        ELSIF(a0='0' AND a1='1' AND st="10") THEN
            internal_bus_in(7 DOWNTO 4)<=pch(3 DOWNTO 0);
        ELSIF(a0='0' AND a1='1' AND st="11" AND ctrreg(7)='1') THEN
```

```
            internal_bus_in(3 DOWNTO 0)<=pcl(3 DOWNTO 0);
            internal_bus_in(7 DOWNTO 4)<=pch(3 DOWNTO 0);
        END IF;
      ELSE
        internal_bus_in<="ZZZZZZZZ";
      END IF;
      d<=internal_bus_in;
  END PROCESS;

PROCESS(cs, wr, reset)
    VARIABLE ctrregF:STD_ULOGIC;
    VARIABLE bctrreg_v:STD_ULOGIC_VECTOR(3 DOWNTO 0);
  BEGIN
    IF (cs='0' AND wr='0') THEN
      ad<=a1 & a0;
      ctrregF:=d(7);
      internal_bus_out<=d;
    END IF;
  IF(reset='1')      THEN
    pa_latch<="00000000";
    pb_latch<="00000000";
    pc_latch<="00000000";
    ctrreg<="10011011";
    bctrreg_v:="0000";
    ctrregF:='0';
  ELSIF (wr'EVENT AND wr='1') THEN
      IF(ctrregF='1' AND ad="11" AND cs='0') THEN
        ctrreg<=internal_bus_out;
      ELSIF(ctrreg(7)='1' AND cs='0' AND ad="00") THEN
        pa_latch<=internal_bus_out;
      ELSIF(ctrreg(7)='1' AND cs='0' AND ad="01") THEN
        pb_latch<=internal_bus_out;
      ELSIF(ctrreg(7)='1' AND cs='0' AND ad="10") THEN
        pc_latch<=internal_bus_out;
      END IF;
      IF(ctrregF='0' AND cs='0' AND ad="11") THEN
        bctrreg_v:=internal_bus_out(3 DOWNTO 0);
        CASE bctrreg_v IS
          WHEN "0000" => pc_latch(0)<='0';
          WHEN "0010" => pc_latch(1)<='0';
          WHEN "0100" => pc_latch(2)<='0';
          WHEN "0110" => pc_latch(3)<='0';
          WHEN "1000" => pc_latch(4)<='0';
```

```
                WHEN "1010" => pc_latch(5)<='0';
                WHEN "1100" => pc_latch(6)<='0';
                WHEN "1110" => pc_latch(7)<='0';
                WHEN "0001" => pc_latch(0)<='1';
                WHEN "0011" => pc_latch(1)<='1';
                WHEN "0101" => pc_latch(2)<='1';
                WHEN "0111" => pc_latch(3)<='1';
                WHEN "1001" => pc_latch(4)<='1';
                WHEN "1011" => pc_latch(5)<='1';
                WHEN "1101" => pc_latch(6)<='1';
                WHEN "1111" => pc_latch(7)<='1';
                WHEN OTHERS => flag<="11";
            END CASE;
        END IF;
        END IF;
 END PROCESS;
PROCESS(pa_latch)
   BEGIN
   IF(ctrreg(4)='0') THEN
      pa<=(pa_latch);
   ELSE
      pa<="ZZZZZZZZ";
   END IF;
   END PROCESS;
      PROCESS(pb_latch)
      BEGIN
        IF(ctrreg(1)='0') THEN
           pb<=pb_latch;
        ELSE
           pb<="ZZZZZZZZ";
        END IF;
   END PROCESS;
   PROCESS(pc_latch)
   BEGIN
     IF (ctrreg(0)='0') THEN
        pcl<=pc_latch(3 DOWNTO 0);
     ELSE
        pcl<="ZZZZ";
     END IF;
END PROCESS;
PROCESS(pc_latch)
  BEGIN
```

```
        IF(ctrreg(3)='0') THEN
            pch<=pc_latch(7 DOWNTO 4);
        ELSE
            pch<="ZZZZ";
        END IF;
    END PROCESS;
END rtl;
```

本程序清单的前 4 条说明语句用于所引用的 IEEE 库。接着 8 行是 8255 的实体描述，其中输入输出引脚与 8255 定义相同。在这里只是将 PC 口的 8 条线分成高 4 位和低4位而已。由于 pa，pb，pc 和 d 都是双向的，故在这里也定义成双向。

8255的构造体由 5 个进程构成，它们是读进程、写进程和形成 pa、pb、pc 三态输出的三个进程。下面对构造体中的有关问题作一说明。

1. 构造体中各信号定义说明

(1) 内部总线

在构造体中定义了两条内部总线 internal_bus_in 和 internal_bus_out，所有 8 位数据的输入或输出，在 8255 芯片内部都是通过这两条总线实现的。

(2) 锁存器和寄存器输出

构造体中信号 pa_latch、pb_latch 及 pc_latch 是 8255 芯片中 A 口、B 口及 C 口锁存器的输出。信号 ctrreg 是方式控制寄存器的输出。

其它信号是为了内部连接而引入的，请读者自行理解。

2. 写进程

8255在"0"型方式下写进程的流程图如图 10 - 5所示。

在写进程最前面是将写 8255 时的最高数据位送标志寄存器保存，以便以后在判别是方式控制字还是位控制字时使用。这里的标志寄存器采用的是变量 ctrregF，而没有采用信号量。

当复位信号有效时(reset='1')，对 8255 芯片进行初始化。前面提到 8255 芯片复位后所有端口都处于输入方式，故方式控制寄存器初始化值应为 9BH，其它均设置为"0"。

如果是写状态，则根据数据线 D_0～D_7送来的不同数据及地址线 A_0～A_1的不同状态，将数据写入控制寄存器或 pa，pb，pc 各输出锁存器。程序中的 CASE 语句用来实现 C 口的位控功能。当写控制寄存器的控制字为位控字($b_7=0$)时，b_3～b_0的值就写入位控寄存器，位控寄存器输出经译码使 C 口的某一位置位或复位。

这里的位控寄存器 bctrreg_v 定义成一个变量。如果将其定义成信号量是否可以，会出现什么结果请读者自行思考。

3. 读进程

读进程的程序如例 10 - 1 所示。其工作过程是当选片信号有效(cs='0')和读信号有效(rd='0')时，从 A 口或 B 口或 C 口读入外部设备提供的数据。注意，在本设计中所有端口输入都是不锁存的。

由于读进程程序比较简单，这里不再用流程图说明。要注意的是，该读进程还描述了最终送数据总线 D_0～D_7的数据是通过三态缓冲器来实现的。

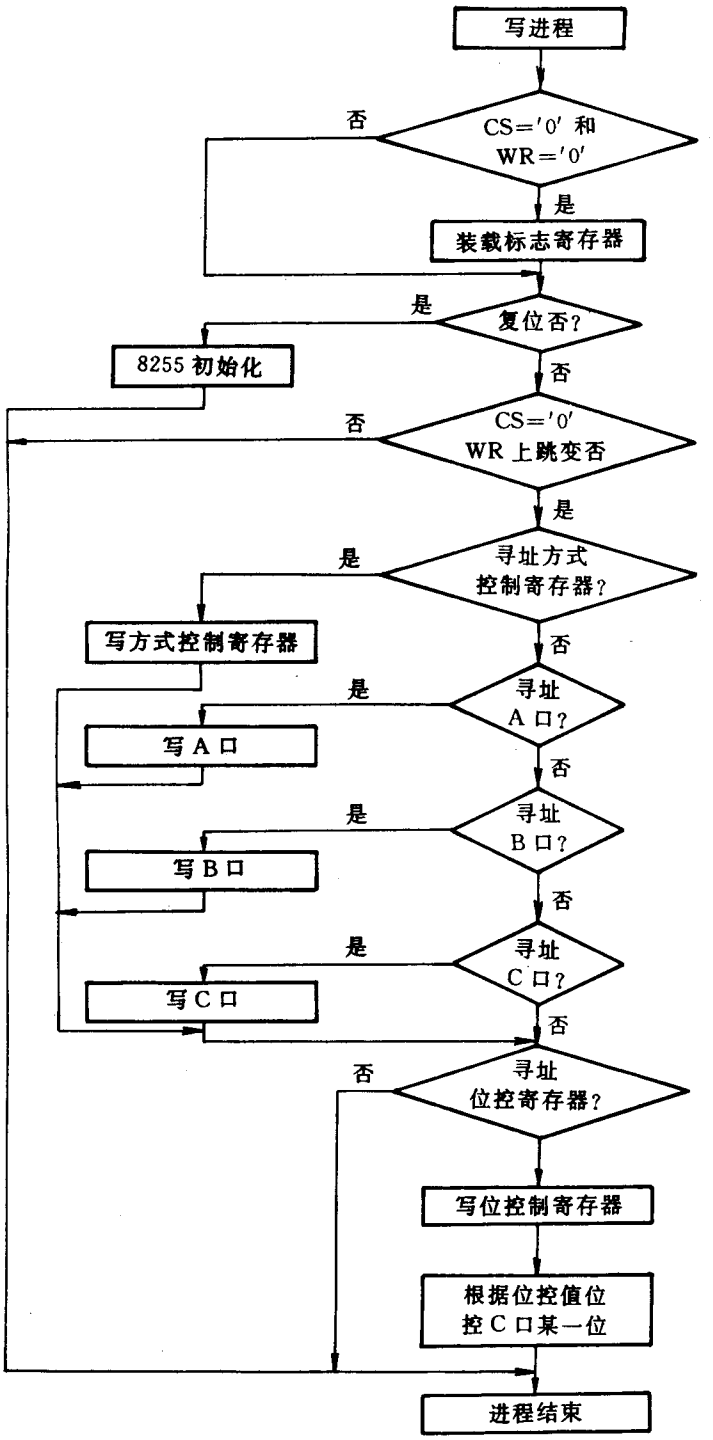

写进程
否
CS='0' 和
WR='0'
是
装载标志寄存器
是
复位否?
8255 初始化
否
否
CS='0'
WR 上跳变否
是
是
寻址方式
控制寄存器?
写方式控制寄存器
否
是
寻址
A 口?
写 A 口
否
是
寻址
B 口?
写 B 口
否
是
寻址
C 口?
写 C 口
否
否
寻址
位控寄存器?
写位控制寄存器
根据位控值位
控 C 口某一位
进程结束

图 10－5　写进程流程图

10.1.5 8255芯片 VHDL 语言描述模块仿真

前面详细地解释了8255芯片 VHDL 语言描述程序的各组成部分。为证明该程序模块的正确性，我们用 MAX+puls Ⅱ 的仿真器进行了实际仿真，其仿真波形如图 10－6 所示。

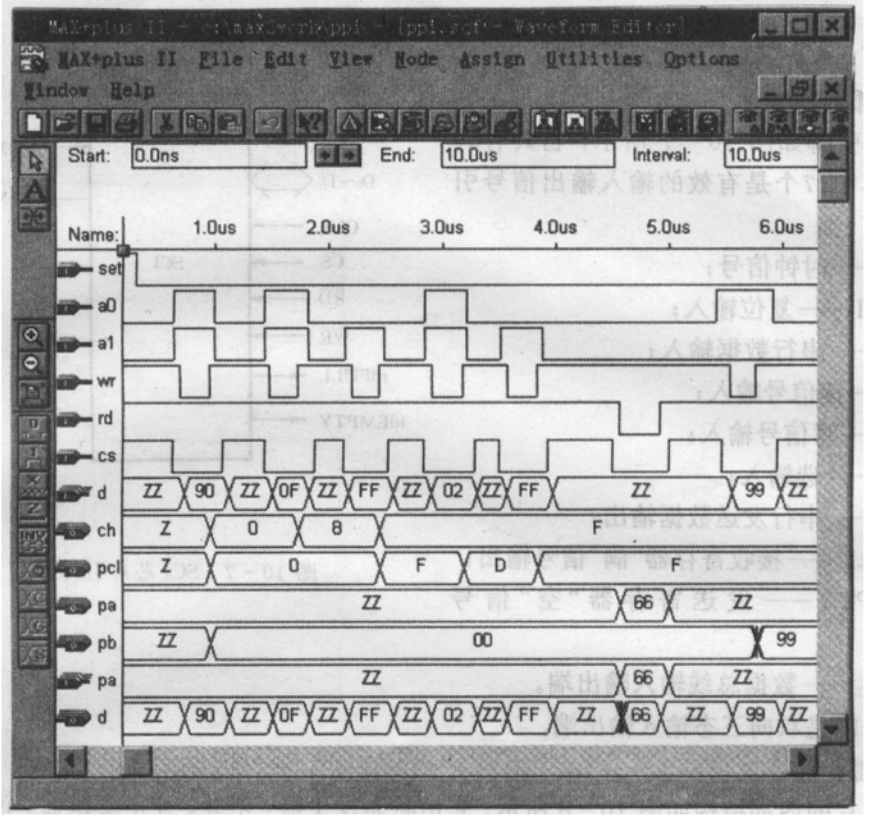


图 10－6　8255芯片 VHDL 语言模块仿真波形

下面参见图 10－6 叙述一下仿真过程。开始 RESET 施加高电平("1")，使8255芯片复位，所有端口都处于输入方式。接着向方式控制寄存器写控制字90H，使得 A 口变为输入口；B 口、C 口变为输出口。然后向控制寄存器写0FH 数据，从仿真波形可以看到，写完后 pch 输出变为8H。0FH 是位控控制字，它将使 C 口的最高位置"1"。将数据总线 D_0～D_7的值改为 FFH，再将该数据写向 C 口，在写操作完成后 C 口变为 FFH。下面再向控制寄存器写位控控制字02H。该字表明要将 PC_1位清零。在写操作完成后，pcl 变为 DH，表明 PC_1位已清零。再改变 A 口的输入数据值，使其成为66H，读 A 口就从 A 口得到66H 值并从 D_0～D_7输出。改变 D_0～D_7数据上的值，使其成为99H，并将该值写向 B 口，那末在该写操作完成后，在 B 口输出端即得到99H 输出值。

由上述分析可知，该8255芯片 VHDL 语言描述模块的仿真结果是正确的，完全符合8255"0"型方式的输入输出功能。

10.2 SCI 串行接口芯片设计实例

目前最常见的串行接口芯片有8251和8250，同样为了简化设计，这里只举一个固定信号格式的串行接口芯片 SCI 的设计。

10.2.1 SCI 的引脚及内部结构

1. 外部引脚

SCI 的引脚如图 10－7 所示，它共有20个引脚，其中17个是有效的输入输出信号引脚。如：

CLK——时钟信号；

RESET——复位输入；

RXD——串行数据输入；

RD——读信号输入；

WR——写信号输入；

CS——片选输入；

TXD——串行发送数据输出；

rdFULL——接收寄存器“满”信号输出；

tdEMPTY——发送寄存器“空”信号输出；

D_0～D_7——数据总线输入输出端。

其中，D_0～D_7是双向三态输入输出端。

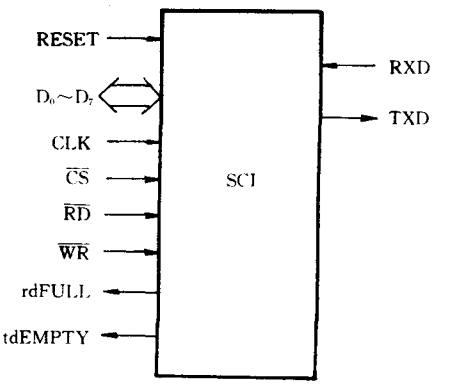


图 10－7　SCI 芯片引脚图

2. 内部结构

SCI 芯片的内部结构如图 10－8 所示。它由状态发生器，串并、并串变换器，锁存器和三态缓冲器组成。由图 10－8 可知，该 SCI 芯片的功能和性能是固定的，而不是程序可编的。

10.2.2 串行数据传送格式及同步控制机构

1. 串行数据传送格式

SCI 芯片以固定的串行数据传送格式传送数据。传送一个数据或一个字符共需 10 位，即 1 位启动位，8 位数据位或一个字符加 1 个校验位，一个停止位。为了能对位进行正确的操作，选取每位数据位应包含 4 个时钟(CLK)周期。为了得到串行数据传送的波特率，例如9600 bps，那么外部时钟应选取为38.4 kHz。

2. 串行数据传送的控制机构

在异步串行数据传送时，由于没有专门提供同步信号，因此只能从所传送的信号中提取同步信息，例如数据的启动位就为 SCI 的串并变换提供启动信号。

(1) 串行数据接收的同步控制

在串行数据接收同步控制器中设置了一个 6 位的计数器，高 4 位为 sh_r，低 2 位为

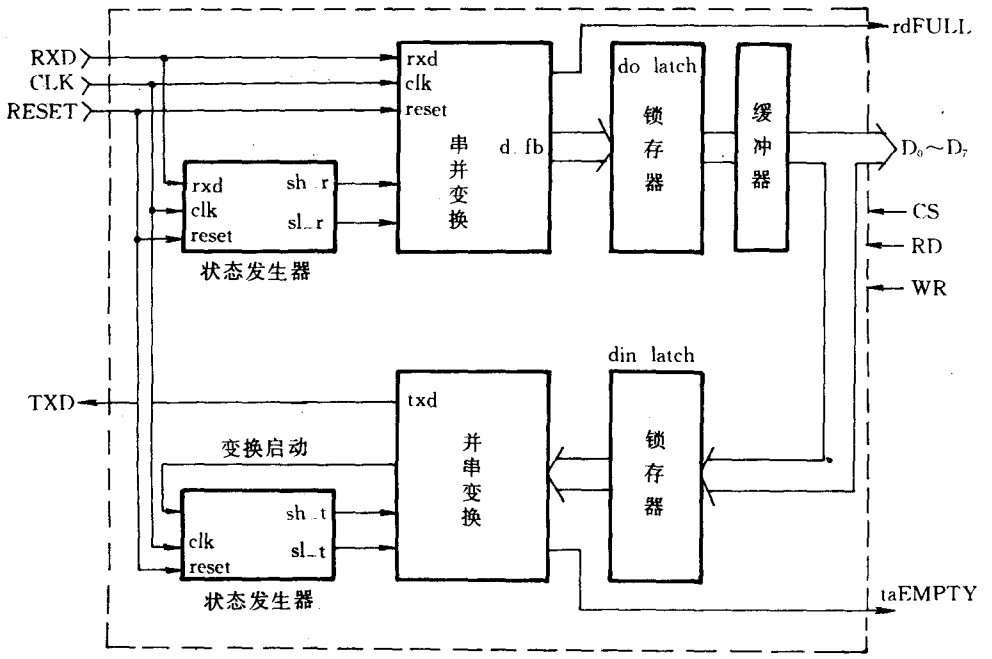


图 10－8　SCI 芯片内部结构框图

sl_r。利用该计数器的计数状态，实现串行数据接收的同步控制。计数器的状态与串行数据接收过程的波形关系如图 10－9 所示。

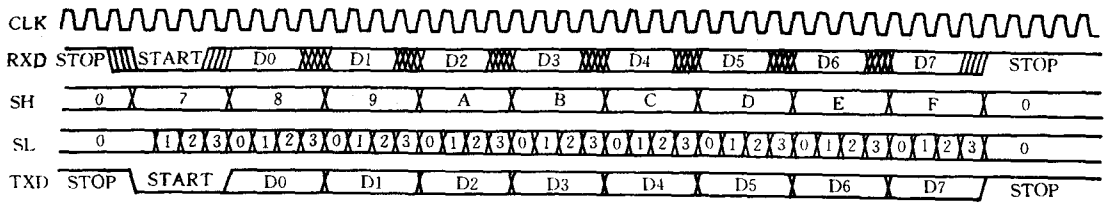


图 10－9　计数器的状态与串行数据接收、发送过程的关系

从图 10－9 中可以看到，在 RXD 端的启动位未到达以前 sh_r 和 sl_r 都保持为“0”。当同步控制机构检测到启动位以后就立即将 sh_r 置为7H(0111B)，sl_r 置为0(00B)。此后计数器启动，对 CLK 进行计数。当计数到 sh_r＝FH，sl_r＝3H 时，一个数据接收过程结束，计数器又处于 0 状态，等待下一个启动位的到来。根据此接收一个数据的过程，即可画出其状态转移图如图 10－10 所示。

从图中可以看到，sh_r 从0H～6H 的值都处于空闲状态，只有出现启动位后 sh_r 才跳到7H 状态，此后每 4 个时钟周期转移一个状态(即接收 1 位数据)，直到 sh_r 为 FH，8 位数据结束为止。

(2) 串行数据发送的同步控制

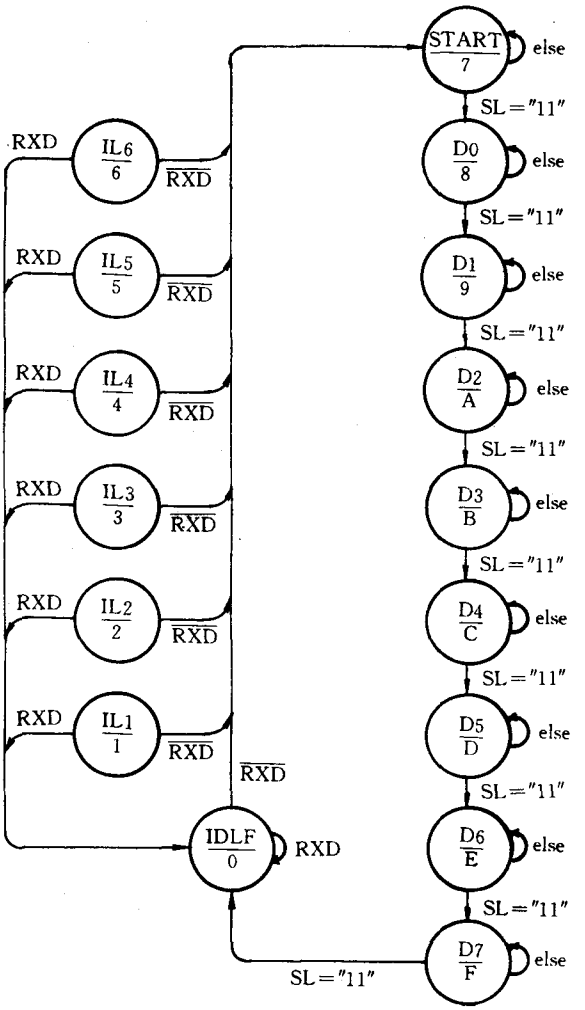


图 10－10　接收一个数据的状态转移图

串行数据发送的同步控制与串行数据接收的同步控制类同，所不同的是启动发送数据的条件不是启动信号，而是“空”信号 tdEMPTY。当发送锁存器 din_latch 空时，tdEMPTY ='1'，当 CPU 向发送锁存器写入一个发送数据以后，tdEMPTY='0'。该信号变“0”，将启动发送计数器，在 CLK 的同步下使 sh_t 置为7H，sl_t 置为0H。在 CLK 时钟驱动下 sh_t 从7H 到 FH 逐个进行计数，完成一个数据的发送过程。发送一个数据的状态转移图请读者自行画出来。

10.2.3　SCI 芯片的 VHDL 语言描述

SCI 芯片的 VHDL 语言描述程序如例 10－2 所示。

【例 10 - 2】

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

ENTITY sci IS
   PORT(clk, reset, rxd, rd, wr, cs:IN STD_ULOGIC;
        txd, rdFULL, tdEMPTY:OUT STD_ULOGIC;
        data:INOUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END sci;

ARCHITECTURE rtl OF sci IS
   SIGNAL scir:STD_ULOGIC_VECTOR(5 DOWNTO 0);
   SIGNAL scit:STD_ULOGIC_VECTOR(5 DOWNTO 0);
   SIGNAL sh_r :STD_ULOGIC_VECTOR(3 DOWNTO 0);
   SIGNAL sl_r :STD_ULOGIC_VECTOR(1 DOWNTO 0);
   SIGNAL sh_t :STD_ULOGIC_VECTOR(3 DOWNTO 0);
   SIGNAL sl_t :STD_ULOGIC_VECTOR(1 DOWNTO 0);
   SIGNAL d_fb:STD_ULOGIC_VECTOR(7 DOWNTO 0);
   SIGNAL din_latch:STD_ULOGIC_VECTOR(7 DOWNTO 0);
   SIGNAL do_latch:STD_ULOGIC_VECTOR(7 DOWNTO 0);
   SIGNAL txdF, rxdF:STD_ULOGIC;
   SIGNAL tdEMPTY_s:STD_ULOGIC:='1';
   SIGNAL rdFULL_s:STD_ULOGIC:='0';

BEGIN
  sh_r<=scir(5 DOWNTO 2);
  sl_r<=scir(1 DOWNTO 0);
  sh_t<=scit(5 DOWNTO 2);
  sl_t<=scit(1 DOWNTO 0);

  tdEMPTY<=tdEMPTY_s;
  rdFULL<=rdFULL_s;
  PROCESS(clk, rd, cs)
    BEGIN
       IF(rd='0' AND cs='0') THEN
          rdFULL_s<='0';
       ELSIF(clk'EVENT AND clk='1') THEN
          IF((rxdF='1') AND (sh_r="1111") AND (sl_r="11")) THEN
             do_latch<=d_fb;
             rdFULL_s<='1';
          END IF;
       END IF;
  END PROCESS;
```

```
PROCESS(wr, cs)
 VARIABLE data_v:STD_LOGIC_VECTOR(7 DOWNTO 0);
 BEGIN
     IF(wr'EVENT AND wr='1') THEN
       IF(cs='0') THEN
          data_v:=data;
          din_latch<=TO_STDULOGICVECTOR(data_v);
       END IF;
     END IF;
END PROCESS;

 PROCESS(clk)
  BEGIN
      IF(clk'EVENT AND clk='1') THEN
        IF(rxd='0') THEN
           rxdF<='1';
        ELSIF((rxdF='1') AND (sh_r="1111") AND (sl_r="11")) THEN
           rxdF<='0';
        END IF;
      END IF;
END PROCESS;

PROCESS(wr, clk)
 BEGIN
     IF (wr='0' AND cs='0') THEN
       txdF<='0';
       tdEMPTY_s<='0';
     ELSIF(clk'EVENT AND clk='1') THEN
       IF(((txdF='0') AND (sh_t="1111") AND (sl_t="11"))
                          OR reset='0') THEN
          tdEMPTY_s<='1';
          txdF<='1';
       END IF;
     END IF;
END PROCESS;

PROCESS(rd, cs)
  VARIABLE do_latch_v:STD_ULOGIC_VECTOR(7 DOWNTO 0);
 BEGIN
       do_latch_v:=do_latch;
       IF(rd='0' AND cs='0') THEN
        data<=TO_STDLOGICVECTOR(do_latch_v);
       ELSE
        data<="ZZZZZZZZ";
       END IF;
```

```
END PROCESS;

 PROCESS(clk, reset)
  VARIABLE scir_v:INTEGER RANGE 0 TO 63;
  VARIABLE scir_s:STD_LOGIC_VECTOR(5 DOWNTO 0);
  BEGIN
      IF (reset='0') THEN
        scir_v:=0;--"000000";
      ELSIF(clk'EVENT AND clk='1') THEN
          IF((scir_v<=27) AND (rxd='0')) THEN --sci_v="011011"
              scir_v:=28;--sci_v="011100";
          ELSIF ((scir_v<=27) AND (rxd='1')) THEN
              scir_v:=0;
          ELSE
              scir_v:=scir_v+1;
          END IF;
      END IF;
      scir_s:=CONV_STD_LOGIC_VECTOR(scir_V, 6);
      scir<=TO_STDULOGICVECTOR(scir_s);
 END PROCESS;

PROCESS(clk, reset)
   VARIABLE scit_v:INTEGER RANGE 0 TO 63;
   VARIABLE scit_s:STD_LOGIC_VECTOR(5 DOWNTO 0);
   BEGIN
       IF(reset='0') THEN
         scit_v:=0;--"000000";
       ELSIF(clk'EVENT AND clk='1') THEN
           IF(scit_v<=27) THEN --sci_v="011011"
              IF (tdEMPTY_s='0' AND wr='1') THEN
                 scit_v:=28;--sci_v="011100";
              ELSE
                 scit_v:=0;
              END IF;
           ELSE
               scit_v:=scit_v+1;
           END IF;
       END IF;
       scit_s:=CONV_STD_LOGIC_VECTOR(scit_V, 6);
       scit<=TO_STDULOGICVECTOR(scit_s);
  END PROCESS;

  PROCESS(clk, reset)
   BEGIN
       IF (reset='0') THEN
```

```
            d_fb<="00000000";
          ELSIF(clk'EVENT AND clk='0') THEN
            IF((sh_r>="1000") AND (sh_r<="1111") AND (sl_r="01")) THEN
               d_fb(7)<=rxd;
               FOR i IN 0 TO 6 LOOP
               d_fb(i)<=d_fb(i+1);
            END LOOP;
          END IF;
        END IF;
    END PROCESS;

    PROCESS(sh_t)
     BEGIN
        CASE sh_t IS
          WHEN "0111" => txd<='0';
          WHEN "1000" => txd<=din_latch(0);
          WHEN "1001" => txd<=din_latch(1);
          WHEN "1010" => txd<=din_latch(2);
          WHEN "1011" => txd<=din_latch(3);
          WHEN "1100" => txd<=din_latch(4);
          WHEN "1101" => txd<=din_latch(5);
          WHEN "1110" => txd<=din_latch(6);
          WHEN "1111" => txd<=din_latch(7);
          WHEN OTHERS => txd<='1';
        END CASE;
    END PROCESS;
END rtl;
```

程序清单的前 4 条语句用于所引用的 IEEE 库。后续的 5 行语句是对 SCI 的实体进行描述，其输入输出引脚定义与图 10－7所示相同。需要指出的是，D_0～D_7是三态双向数据总线。

1. 构造体上各信号定义说明

(1) 计数器输出

在 SCI 芯片描述程序中定义了两个计数器输出。其中一个是发送计数器输出 scit，一个是接收计数器输出 scir。另外还将 scit 输出分为高 4 位输出 sh_t 和低 2 位输出 sl_t；同理 scir 输出分为 sh_r 和 sl_r 两个输出。

(2) 移位寄存器输出

接收串行数据的移位寄存器输出定义为 d_fb。在一个数据接收结束后，它输出的就是一个完整的 8 位数据值。

(3) 锁存器输出

SCI 芯片有两个锁存器 do_latch 和 din_latch。其中 do_latch 锁存移位寄存器输出的数据，而 din_latch 则锁存要发送的数据。

(4) 标志寄存器输出

为了实现与计算机的读/写同步，这里设置了两条状态线 rdFULL_s 和 tdEMPTY_s。当接收数据装入 do_latch 时，rdFULL_s 变“1”，而当计算机读走该数据时则置“0”。当计算机写入一个发送数据至 din_latch 时，tdEMPTY_s 变“0”，而一个数据发送完毕时则置“1”。txdF 和 rxdF 是发送和接收过程的中间标志。

2. 内部各进程描述

第 1 个进程是对 rdFULL 标志进行置“0”、置“1”操作和在串行数据接收结束时将接收数据送给 do_latch 锁存器输出的进程。当接收结束，移位寄存器 d_fb 将数据送入 do_latch，同时将标志信号 rdFULL_s 置“1”，向计算机表示数据已准备好，可以来读取。当 rd ='0'且 cs='0'时，表示计算机读接收数据，此时使 rdFULL_s 变“0”，表明数据已读走，为下一次读作准备。

第 2 个进程是写数据进程，计算机将数据总线上的一个数据写发送寄存器 din_latch。

第 3 个进程是对标志 rxdF 进行操作的进程。当 rxd='0'，表明启动位到来时，rxdF 置“1”；而当一个数据接收完毕时则将其置“0”。

第 4 个进程是对 txdF 和 tdEMPTY_s 进行置“0”，置“1”操作的进程。当写发送寄存器时 txdF 和 tdEMPTY_s 置“0”，而一个数据发送结束时它们都被置“1”。

第 5 个进程是读进程，当 rd='0'和 cs='0'时变换好的数据从接收锁存器 do_latch 输出到数据总线 data 上。

第 6 个进程是数据接收控制进程，当数据接收端 rxd 出现低电平，一个数据启动位到来时，接收控制计数器 scir 将其置成“011100”(28)，即 sh_r=“0111”(7)，一个接收计数周期开始。当计数到 63 时(sh_r=“1111”，sl_r=“11”)，scir 清零等待下一个启动位的到来。

第 7 个进程是数据发送控制进程，其原理同数据接收进程。所不同的是，当 CPU 写一个数据到发送寄存器 din_latch 时，其控制计数器 scit 开始一个发送的计数周期，当一个数据发送结束，即 scit 计数到 63 后，就清零，等待下一个发送数据到来。

第 8 个进程是接收数据移位控制进程，进行数据的串并变换。

第 9 个进程是发送数据的并串变换进程。

10.2.4 SCI 芯片 VHDL 语言描述模块仿真

SCI 芯片 VHDL 模块仿真波形如图 10－11 所示。它也是用 MAX+plus Ⅱ 的仿真器进行仿真的结果。

如图 10－7 所示，SCI 复位以后，发送数据控制计数器 scit 和接收数据控制计数器 scir 都被清零，tdEMPTY 为“1”，rdFULL 为“0”。SCI 芯片进入初始状态。为了仿真接收和发送过程，在 rxd 信号线上设置一个“0011110001”一串数据，表示一个 8 位的1EH 值的数据。在数据总线上放一个 36H 值的数据。在 data 的值为 36H 时刻，使 cs 和 wr 出现一个为“0”的负向选通和写入脉冲。在串行数据的停止位之后，在 cs 和 rd 上设置一个负向选通脉冲和读信号，然后启动仿真器。那么在 txd 上就可以看到一串“0011011001”的发送脉冲，在 data 的数据线上的与读脉冲对应位置得到 1EH 的接收数据值。该仿真结果表明，SCI 芯片的工作过程是正确的。

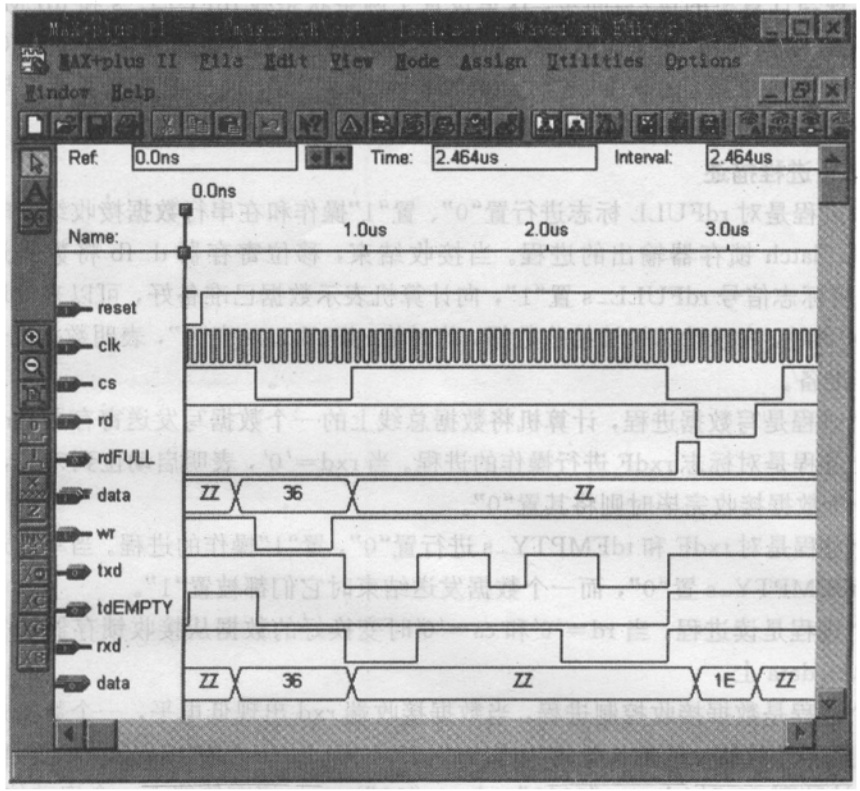


图 10 - 11　SCI 芯片 VHDL 语言模块的仿真波形

10.3　键盘接口芯片 KBC 设计实例

这里介绍一个基于 RS232C 总线的键盘接口芯片设计。在一般的 PC 机中键盘与主机的接口几乎都采用这种方式。但是，我们对该芯片的功能作了相应的简化，以便更容易理解。

10.3.1　KBC 的引脚及内部结构

1. 外部引脚

KBC 芯片的引脚如图 10 - 12 所示。它有 23 条有效的输入输出引脚，其中

$SCAN_0$～$SCAN_7$——键盘扫描输出线，列线。

$RETN_0$～$RETN_7$——键盘扫描输入线，行线。

LED_0～LED_2——LED 显示输出端。

RESET——复位输入端。

RXD——串行数据接收端。

TXD——串行数据发送端。

CLK——时钟输入端。

与 SCI 芯片一样，CLK 时钟输入信号应为串行数据传送波特率的 4 倍。

2. 工作原理与内部结构

KBC 芯片与键盘的连接图如图 10－13 所示。

图中 $SCAN_0$～$SCAN_7$与键盘矩阵的列线相连接，而键盘矩阵的行线与键盘扫描输入线 $RETN_0$～$RETN_7$相连接。LED_0～LED_2与 3 个发光二极管相连接。在正常工作时，SCAN 按列进行扫描。例如，当 SCAN＝“11111110”，对键盘矩阵的第0列进行扫描。如果该列上有一个键按下，$RETN_0$～$RETN_7$的某一行就会变成“0”。例如，第 0 列，第 0 行的那个键被按下，那时 RETN＝“11111110”。如果无键按下，那么 RETN＝“11111111”。键盘向下一列扫描，使列扫描输出 SCAN＝“11111101”。这样一直循环工作。当有键按下时，从当时的 SCAN 值和 RETN 值就可以判别是哪一个键按下。

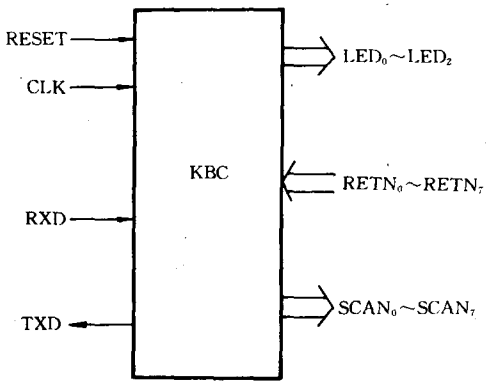


图 10－12　KBC 芯片引脚图

该键盘共有 64 个按键，每个键所对应的 SCAN 值和 RETN 值如表 10－1 所示。当某一键按下时，根据当时的 SCAN 值和 RETN 值，查阅表 10－1就可以确定是哪一个键按下，同时得到的键 ASCII 值就可以通过 TXD 串行数据发送线，按预定格式发向主机，从而实现键盘的输入。

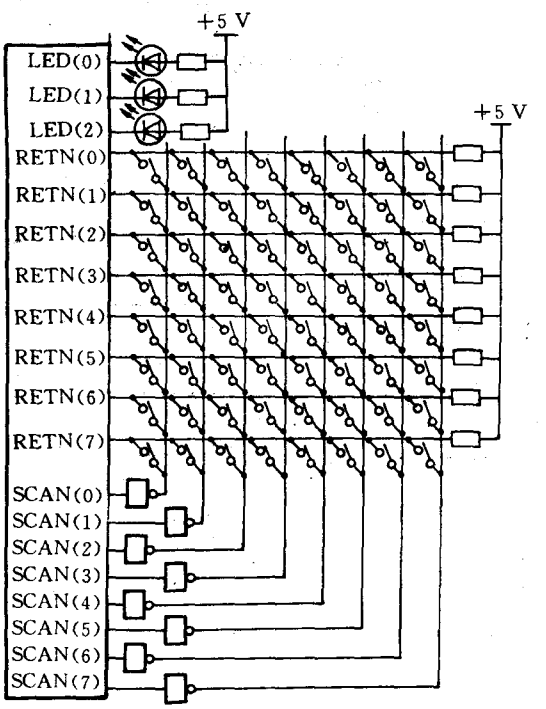


图 10－13　KBC 芯片与键盘的连接图

表 10－1　键和 ASCII 码的关系

RETN输入 \ SCAN输出	0	1	2	3	4	5	6	7
0	空格	(	0	8	@	H	P	X
1	!	)	1	9	A	I	Q	Y
2	"	*	2	:	B	J	R	Z
3	#	+	3	;	C	K	S	[
4	$	,	4	<	D	L	T	¥
5	%	—	5	=	E	M	U	]
6	&	.	6	>	F	N	V	^
7	'	/	7	?	G	O	W	_
对应 ASCII 码高 4 位	2		3		4		5	

例如，当 0 行 0 列的空格键按下时，SCAN＝“11111110”，RETN＝“11111110”。根据表 10－1 则生成20H 的键值，并从 TXD 发向主机。

LED_0～LED_2用来显示数据接收端 RXD 的每个数据的低 3 位值。当每个数据的低 3 位相同时，显示状态就不变化，否则就会出现显示闪烁，用它可以检查芯片的工作情况。

根据上述 KBC 芯片的工作原理，就可以画出该芯片内部结构框图，如图 10－14 所示。

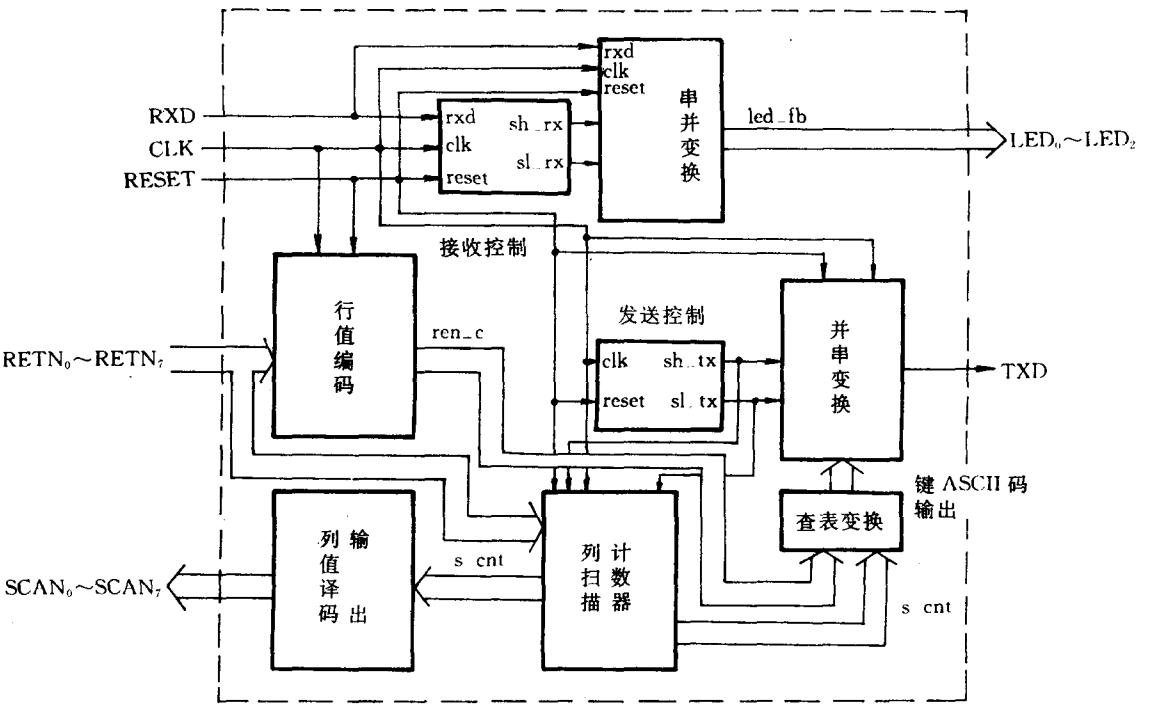


图 10－14　KBC 芯片内部结构框图

由图 10－14可知，KBC 芯片有如下几部分构成：

接收控制机构——控制串行数据接收；

发送控制机构——控制串行数据的发送；

串并变换——将串行格式信号变成并行格式信号输出；

并串变换——将并行格式数据变成串行格式信号输出；

行值编码——将 8 位行输入状态变成 3 位二进制码；

列值译码——将 3 位列计数器状态译成 8 位二进制码；

列扫描计数器——进行列扫描计数；

查表变换——将 r_enc 的值和 s_cnt 的值变换成对应的 ASCII 码值。

从 KBC 芯片内部结构框图可以发现，它是在串行接口芯片基础上，加上有关键盘矩阵接口部分而形成的。

10.3.2 同步控制机构和查表变换

1. 同步控制机构

考虑到键输入的特点，这里的同步机构要比 SCI 芯片稍稍复杂一些。

(1) 列扫描计数器

键盘矩阵的列扫描是通过列扫描计数器的循环计数来实现的。当没有键按下时，计数器的状态将以时钟频率的四分之一的速度进行转移，其状态转移图如图 10－15 所示。

(2) 串行数据发送的同步控制

在上一节 SCI 芯片设计时已提到，为了实现串行数据的发送，设置了一个 6 位计数器 sci_tx，其中 sh_tx 是它的高 4 位。在这里只有键按下时才发送数据。因此，启动发送的条件是有键按下(即 sh_tx＝1，retn＝FFH)。另外，一个对应按下键的 ASCII 码被发送以后，就要等待下一次有键按下时才能发送数据。所以，同步机构应使被按下的键离开以后才能进入下一个数据的发送准备状态，也就是说，在按下的键抬起以前应使 sci_tx 处于“000000”的闲置状态。一旦键抬起后，再将 sci_tx 置为“000100”状态，即使 sh_tx＝1。sh_tx 的状态转移图如图 10－16 所示。

值得注意的是，键的机械抖动将会破坏同步而出现错误。本芯片中并没有采用任何去抖动措施。因此，为了正确工作应尽量选择质量高的按键。例如，其抖动时间可以小于 1 ms。另外尽可能降低串行发送数据的波特率。使发送一个数据的时间稍长于键触点抖动的时间。用这种方法来增加键盘工作的可靠性。当然，一次按下多个键的情况也是不允许的，这样同样会导致出错。

2. 查表变换

如表 10－1 所示，当有一个键按下时，根据当时的 SCAN 和 RETN 值即可以确定哪一个键按下。但是，要将 SCAN 和 RETN 的状态变成该键对应的 ASCII 码值，却要进行相应的变换运算，为了得到对应的变换表达式，将表 10－1 的表示形式稍作一些变化，以便更易找出变换的规律，如表 10－2 所示。对照表 10－1 和表 10－2 可以看到。

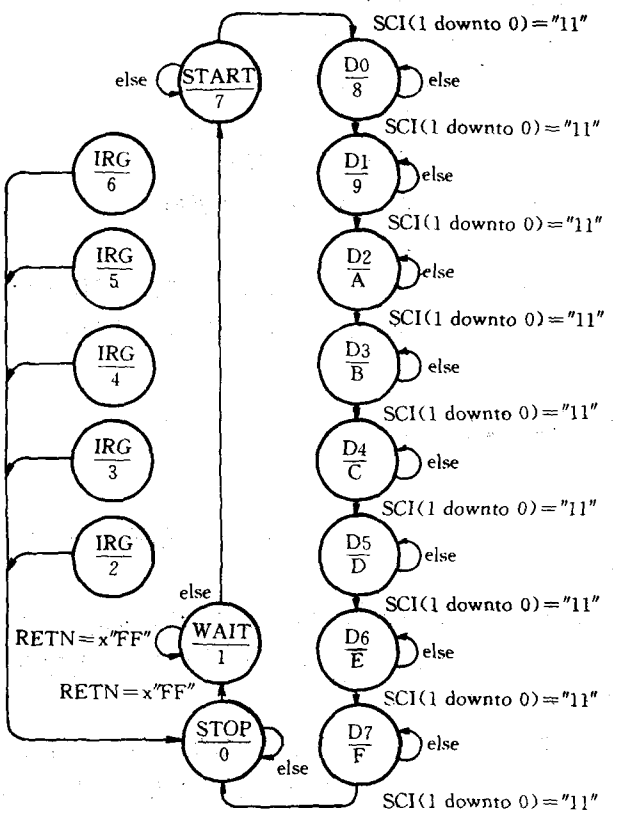


图 10－15　列扫描计数器状态转移图

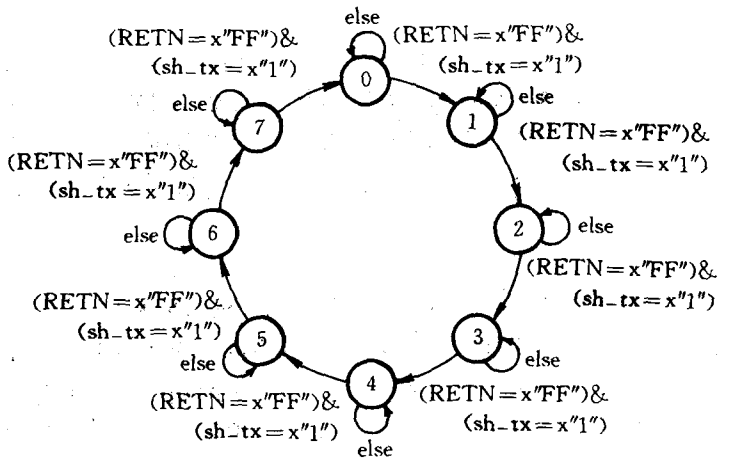


图 10－16　sh_tx 的状态转移图

表 10－2　ASCII 码变换关系表

SCAN 输出	ASCII 码字符	ASCII 码值		s_cnt		
		高位	低位			
0	空格～'	0010	0××××	0	0	0
1	(～/	0010	1××××	0	0	1
2	0～7	0011	0××××	0	1	0
3	8～?	0011	1××××	0	1	1
4	@～G	0100	0××××	1	0	0
5	H～O	0100	1××××	1	0	1
6	P～W	0101	0××××	1	1	0
7	X～_	0101	1××××	1	1	1

按下键的对应 ASCII 码值的低 3 位与 RETN 的扫描行号一致，即与 RETN 编码输出 R_ENC(0)－R_ENC(2)的值一致。第 3 位值与列计数器的第 0 位一致。第 4 位与列计数器的第1位一致。第 5 位与列计数器的第 2 位取反值一致。第 6 位与列计数器的第 2 位一致。这样，按下键的 ASCII 码值可以表示为

按下键的 ASCII 码值＝'0'&s_cnt(2)& $\overline{\text{s\_cnt(2)}}$&s_cnt(1)&s_cnt(0)
&r_enc(2)&r_enc(1)&r_enc(0)

变换运算就变得非常简单。

10.3.3　KBC 芯片的 VHDL 语言描述

KBC 芯片的 VHDL 语言描述程序如例 10－3所示。

【例 10－3】

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

ENTITY kbc IS
    PORT(reset, clk, rxd:IN STD_ULOGIC;
         txd:OUT STD_ULOGIC;
         retn:IN STD_ULOGIC_VECTOR(7 DOWNTO 0);
         scan:OUT STD_ULOGIC_VECTOR(7 DOWNTO 0);
         led:OUT STD_ULOGIC_VECTOR(2 DOWNTO 0));
END kbc;

ARCHITECTURE rtl OF kbc IS
    SIGNAL sci_rx:STD_ULOGIC_VECTOR(5 DOWNTO 0);
    SIGNAL sh_rx:STD_ULOGIC_VECTOR(3 DOWNTO 0);
    SIGNAL sl_rx:STD_ULOGIC_VECTOR(1 DOWNTO 0);
    SIGNAL sci_tx:STD_ULOGIC_VECTOR(5 DOWNTO 0);
    SIGNAL sh_tx:STD_ULOGIC_VECTOR(3 DOWNTO 0);
    SIGNAL s_cnt:STD_ULOGIC_VECTOR(2 DOWNTO 0);
    SIGNAL r_enc:STD_ULOGIC_VECTOR(2 DOWNTO 0);
```

```
    SIGNAL led_fb:STD_ULOGIC_VECTOR(2 DOWNTO 0);
BEGIN
    sh_rx<=sci_rx(5 DOWNTO 2);
    sl_rx<=sci_rx(1 DOWNTO 0);
    sh_tx<=sci_tx(5 DOWNTO 2);

        PROCESS(reset, clk)
          VARIABLE sci_rx_v:INTEGER RANGE 0 TO 63;
          VARIABLE sh_rx_v:INTEGER RANGE 0 TO 15;
          VARIABLE sci_rx_s:STD_LOGIC_VECTOR(5 DOWNTO 0);
          VARIABLE sh_rx_s:STD_LOGIC_VECTOR(3 DOWNTO 0);
        BEGIN

            IF(reset='0') THEN
               sci_rx_v:=0;
            ELSIF(clk'EVENT AND clk='1') THEN
                IF(sci_rx_v<=27) THEN
                   IF(rxd='0') THEN
                      sci_rx_v:=28;
                   ELSE
                       sci_rx_v:=0;
                   END IF;
                ELSE
                        sci_rx_v:=sci_rx_v+1;
                END IF;
            END IF;
            sci_rx_s:=CONV_STD_LOGIC_VECTOR(sci_rx_v, 6);
            sci_rx<=TO_STDULOGICVECTOR(sci_rx_s);
        END PROCESS;

        PROCESS(reset, clk)
        BEGIN
            IF(reset='0') THEN
               led_fb<="000";
            ELSIF(clk'EVENT AND clk='1') THEN
                IF(sl_rx="01") THEN
                   CASE sh_rx IS
                         WHEN "1000" => led_fb(0)<=rxd;
                         WHEN "1001" => led_fb(1)<=rxd;
                         WHEN "1010" => led_fb(2)<=rxd;
                         WHEN OTHERS => led_fb<=led_fb;
                   END CASE;
                ELSE
                    led_fb<=led_fb;
                END IF;
```

```
        END IF;
    END PROCESS;
    led<=led_fb;

    PROCESS(reset, clk)
    BEGIN
        IF(reset='0') THEN
            r_enc<="000";
        ELSIF(clk'EVENT AND clk='0') THEN
            IF(retn="11111111") THEN
                r_enc<=r_enc;
            ELSE
                IF    (retn(0)='0') THEN r_enc<="000";
                ELSIF(retn(1)='0') THEN r_enc<="001";
                ELSIF(retn(2)='0') THEN r_enc<="010";
                ELSIF(retn(3)='0') THEN r_enc<="011";
                ELSIF(retn(4)='0') THEN r_enc<="100";
                ELSIF(retn(5)='0') THEN r_enc<="101";
                ELSIF(retn(6)='0') THEN r_enc<="110";
                ELSE                    r_enc<="111";
                END IF;
            END IF;
        END IF;
    END PROCESS;

    PROCESS(reset, clk)
    VARIABLE s_cnt_v:INTEGER RANGE 0 TO 7;
    VARIABLE s_cnt_s:STD_LOGIC_VECTOR(2 DOWNTO 0);
    BEGIN
        IF(reset='0') THEN
            s_cnt_v:=0;
        ELSIF(clk'EVENT AND clk='0') THEN
            IF(retn="11111111" AND sh_tx="0001") THEN
                s_cnt_v:=s_cnt_v+1;
            ELSE
                s_cnt_v:=s_cnt_v;
            END IF;
        END IF;
        s_cnt_s:=CONV_STD_LOGIC_VECTOR(s_cnt_v, 3);
        s_cnt<=TO_STDULOGICVECTOR(s_cnt_s);
    END PROCESS;

    PROCESS(s_cnt)
    BEGIN
```

```
    CASE s_cnt IS
        WHEN "000" => scan<="11111110";
        WHEN "001" => scan<="11111101";
        WHEN "010" => scan<="11111011";
        WHEN "011" => scan<="11110111";
        WHEN "100" => scan<="11101111";
        WHEN "101" => scan<="11011111";
        WHEN "110" => scan<="10111111";
        WHEN OTHERS => scan<="01111111";
    END CASE;
END PROCESS;

PROCESS(reset, clk)
    VARIABLE sci_tx_v:INTEGER RANGE 0 TO 63;
    VARIABLE sci_tx_s:STD_LOGIC_VECTOR(5 DOWNTO 0);
BEGIN
    IF(reset='0') THEN
        sci_tx_v:=0;
    ELSIF (clk'EVENT AND clk='0') THEN
        IF(sci tx_v>=28 ) THEN
            sci_tx_v:=sci_tx_v+1;
        ELSIF(sh_tx="0001") THEN
            IF (retn="11111111") THEN
              sci_tx_v:=4;
            ELSE
              sci_tx_v:=28;
            END IF;
        ELSIF(sh_tx="0000") THEN
            IF (retn="11111111") THEN
              sci_tx_v:=4;
            ELSE
              sci_tx_v:=0;
            END IF;
        ELSE
            sci_tx_v:=0;
        END IF;
    END IF;
    sci_tx_s:=CONV_STD_LOGIC_VECTOR(sci_tx_v, 6);
    sci_tx<=TO_STDULOGICVECTOR(sci_tx_s);
END PROCESS;

PROCESS(sh_tx, r_enc, s_cnt)
BEGIN
    CASE sh_tx IS
```

```
            WHEN "0111" => txd<='0';
            WHEN "1000" => txd<=r_enc(0);
            WHEN "1001" => txd<=r_enc(1);
            WHEN "1010" => txd<=r_enc(2);
            WHEN "1011" => txd<=s_cnt(0);
            WHEN "1100" => txd<=s_cnt(1);
            WHEN "1101" => txd<=NOT s_cnt(2);
            WHEN "1110" => txd<=s_cnt(2);
            WHEN "1111" => txd<='0';
            WHEN OTHERS => txd<='1';
        END CASE;
    END PROCESS;
END rtl;
```

程序清单的前 3 条语句用于所引用的 IEEE 库，后继 7 行语句是对 KBC 的实体进行描述，其输入输出引脚定义与图 10－8所示相同。

1. 构造体上各信号定义说明

(1) 接收、发送计数器输出

此两个信号 sci_rx 和 sci_tx 与 SCI 芯片中的一样，是 6 位的接收计数器和 6 位发送计数器输出。同样 sh_rx 为 sci_rx 的高 4 位输出；sl_rx 为 sci_rx 的低 2 位输出；sh_tx 为 sci_tx 的高 4 位输出。

(2) 列计数器输出

s_cnt 是 3 位列计数器输出，对该 3 位二进制数译码就产生8位的列扫描输出 scan。

(3) 行编码器输出

对键盘矩阵的行输入信号 retn(0)～retn(7) 8 位数据进行编码，得到 3 位编码输出信号 r_enc。

(4) 低 3 位接收寄存器输出

led_fb 是低 3 位接收寄存器输出，用来驱动 LED_0～$LED_2$3个外接发光二极管。

2. 内部各进程描述

第 1 个进程是串行数据接收控制进程，也是接收计数器状态控制和计数控制进程。

第 2 个进程是串行数据接收和串并变换进程，其输出驱动 LED_0～$LED_2$3 个发光二极管。

第 3 个进程是行编码进程，对从引脚 retn(0)～retn(7)送来的行扫描输入信号进行编码，形成 3 位行编码输出信号。该编码工作在其输入不是为全"1"时进行。由进程的语句可知，如果有两个键同时按下，那么所处行号低的将被编码，也就是行号低的优先。

第 4 个进程是列计数进程，该进程对列计数器进行状态和计数控制。列计数器输出用来产生列扫描输出 scan(0)～scan(7)。在没有键按下时该计数器应循环计数，故 s_cnt 的计数受 retn 控制，当 retn="11111111"时(即无键按下时)，计数器进行加 1 计数操作。一旦有键按下，s_cnt 值将保持不变。

第 5 个进程是对列扫描计数器输出进行译码的进程。3 位列计数器输出，产生 8 位译

码输出。

第 6 个进程是发送串行数据的控制进程，也是发送计数器状态和计数控制进程。这个进程与 SCI 芯片的控制进程不同的地方是插入了判断等待键输入和等待键抬起的控制。等待键输入用进程中的第 8 行和第 9 行语句进行控制。等待键抬起用进程中的第 14 和 15 行语句进行控制。

第 7 个进程是查表变换和发送串行数据的进程。变换关系在前面已叙述，其它与 SCI 芯片的串行数据发送进程相同。

10.3.4 KBC 芯片 VHDL 语言描述模块仿真

KBC 芯片 VHDL 模块的仿真波形如图 10－17 所示。

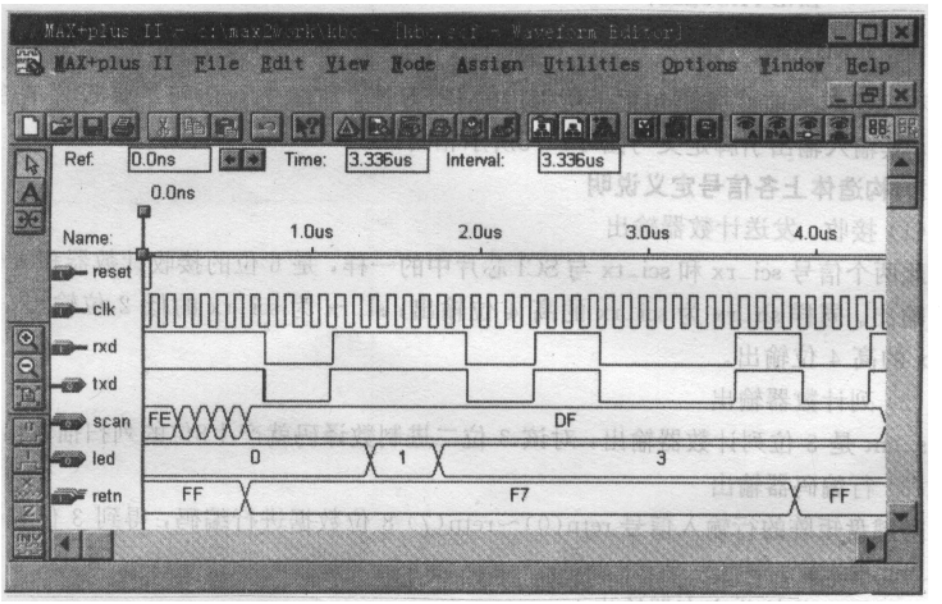


图 10－17　KBC 芯片 VHDL 语言模块的仿真

该仿真波形是用 MAX＋plus Ⅱ 的仿真器进行仿真的结果。如图 10－17 所示，复位之后所有寄存器、计数器都被清零。将 retn 置为 FFH(初始态)，然后在 scan 输出为 DFH 处，将 retn 值改为 F7H，其 F7H 数据保持时间可持续若干个 scan 的计数周期，如图所示。这样的数据输入是模拟大写字母 K 键按下。当 K 键按下时，

scan＝"11011111"

retn＝"11110111"

然后启动仿真器，在 txd 上即可读得4BH 的串行数据。它表明仿真结果正确。

同样，在串行数据接收线上置上一个4BH 的数据，在 led 端输出即可得到 1 和 3 的数据。它表明仿真结果也是正确的。

第 11 章

93 版和 87 版 VHDL 语言的主要区别

本书以 VHDL 语言的 87 版本为基础编写而成的，但是随着 VHDL 语言的使用，发现 87 版本存在许多缺陷和局限性，因此 IEEE 对 87 版本进行了修订，推出了较完善的 93 版本。93 版本的某些特性是特有的，为了编写出与 87 版兼容的模块，就必须避免使用这些特性。当然用 93 版本的 VHDL 语言编写程序就会更灵活更方便。在一般使用 EDA 工具进行编译时，都要事先指定版本号，这一点务请读者引起充分注意。

11.1 VHDL 语言 93 版本的特点

为了便于读者了解和查阅，下面将详细罗列出 93 版本 VHDL 语言所引入的几十种重要的变化特征，并加上适当例子加以说明。

11.1.1 文件是 VHDL 语言新的客体

如前所述，87 版中有 3 类基本客体，即变量、常量和信号。93 版中将文件作为新增的客体，在 VHDL 语言中使用。

1. 文件说明语句

文件说明语句的格式为：

```
FILE  文件名：数据子类型说明
  [OPEN 打开文件类型]IS 路径表达式；
```

例如，

```
TYPE bv_ftype IS FILE OF BIT_VECTOR;
FILE vec_file: bv_ftype IS "usr/home/jb/vec.in";
```

该例子说明：vec.file 是一个以位矢量存贮的文件，由于没有打开文件类型(属缺省)，故它处于 READ_MODE 状态，是一个输入文件。该文件的读取路径及物理文件名为：usr/home/jb/vec.in。

又如，

```
FILE in_file:TEXT OPEN READ_MODE IS
                    "post.dat";
FILE out_file:TEXT OPEN WRITE_MODE IS
```

"fir3_out.data";

在上述两个文件说明语句中，in_file 是输入文件，读入的是当前目录的 post.dat 文件；另外，out_file 是输出文件，它将 out_file 的内容写到当前目录的 fir3_out.data 文件名中存放起来。

2. 文件类型说明语句

文件类型说明语句用于文件数据类型的说明。每个文件类型说明都隐含定义了对所说明文件的操作。这些操作用过程 FILE_OPEN，FELE_CLOSE，READ 及 WRITE 和函数 ENDFILE 来描述。

文件类型定义的格式为：

TYPE 类型名 IS FILE OF 类型/子类型名；

例如：

```
TYPE idex IS RANGE 0 TO 15;
TYPE int_ftype IS FILE OF index;
```

该例说明，int_ftype 是一个 index 值的文件类型，同时，该文件类型还隐含地说明了如下操作：

```
PROCEDURE FILE_OPEN(FILE F:int_ftype;
        EXTERNAL_NAME:IN STRING;
        OPEN_KIND:IN FILE_OPEN_KIND:=READ_MODE);
PROCEDURE FILE_OPEN(STATUS:OUT FILE_OPEN_STATUS;
            FILE F:int_ftype;EXTERNAL_NAME:IN STRING;
            OPEN_KIND:IN FILE_OPEN_KIND:=READ_MODE);
```

第一个过程是以所说明的打开类型打开由外部名所说明的文件，返回文件指示器 F。第二个过程是返回过程的状态。

```
PROCEDURE FILE_CLOSE(FILE F:int_ftype);
```

该过程关闭所说明的文件。

```
PROCEDURE READ(FILE F:int_ftype;
                    VALUE:OUT index);
```

该过程从文件 F 读一个数据类型为 index 的值。

```
PROCEDURE WRITE(FILE F:int_ftype;
                    VALUE:IN index);
```

该过程将一个 index 类型的数据写到文件 F 中去。

```
FUNTION ENDFILE(FILE F:int_ftype)
                    RETURN BOOLEAN;
```

该函数检测文件 F，如果到了文件末尾，则返回一个 TRUE 值。

上面仅仅给出了 93 版本在文件说明中所涉及的一些基本问题，在实际编程时应以此为索引参考有关例程才行。

11.1.2 在端口映射中使用常量表达式

在本书的 4.3.3 节中叙述了名称映射方法，例如：

```
u2:and2 PORT MAP(a=>nsel, b=>dl, c=>ab);
```

其中a，b是“与门”的输入端，c是输出端。nsel，dl和ab是信号量或输入端口名。映射的对象都是信号量。但是，在93版本中这种情况已有了拓展，映射的对象可以是一个常量表达式。例如：

```
M1:mux PORT MAP(sel=>TO_MVL(code),
    d0=>TO_MVL(bus(0)), d1=>TO_MVL(bus(1)),
    TO_BIT(2)=>ctrl);
```

该例说明二选一的选择器的输入端为sel，d0，d1。这里映射的是函数表达式如：sel=>TO_MVL(code)、d0=>TO_MVL(bus(0))等。实际上选择器选择输入端sel代入的是函数TO_MVL(code)返回的值，其它各端也类同。

11.1.3 定义了共享变量

前面已经提到信号量和变量的重要区别是信号可以是全局量，只要在构造体中已定义，那么构造体内的所有地方都可以使用。而变量是局部量，只能在进程及子程序内部定义和使用。如想将结果带出外部，必须将变量值赋给某一个信号量才行。

但是，实际使用过程中希望进程或子程序中的结果以变量形式进行数据传递。为此，在93版本中定义了共享变量。共享变量的说明格式为：

```
SHARED VARIABLE 变量名：子类型名[:=初始值];
```

例如：

```
ARCHITECTURE sample OF tests IS
    SHARED VARIABLE notclk:STD_LOGIC;
    SIGNAL clk:STD_LOGIC;
BEGIN
    p1:PROCESS(clk)IS
        BEGIN
            IF(clk' EVENT AND clk='1') THEN
                notclk:='0';
            END IF;
        END PROCESS p1;
    p2:PROCESS(clk)IS
        BEGIN
            IF(clk'EVENT AND clk='0') THEN
                notclk:='1';
            END IF;
        END PROCESS p2;
    END ARCHITECTURE sample;
```

p1进程在时钟上升沿时将共享变量notclk置为“0”，而p2进程在时钟下降沿时将notclk置为“1”，从而使notclk和clk在任何时刻，其值正好相反。共享变量除在进程和子程序的说明域中不能使用外，其它任何地方都可以使用。

11.1.4 定义了 GROUP

这里引入了组的概念，一个组是一些已定义项目的集合。一个组模块说明用于定义一个组模块，也就是将一些已定义的项目构成了一个组。一个组说明对应于一个组模块。组说明的格式为：

GROUP 组名：组属性名(项目，项目……)；

例如：

```
GROUP ml:mark(alu3, compo5, mux32);
GROUP k1:keep(rst, rdy, sdrd);
GROUP q_grp:equivalent('A', 'a');
```

由上例可知，组内的项目可以是一个标号，可以是信号名，也可以是简单的值。组名不同，但属性名相同的那些组，其组构成的结构是一致的。

11.1.5 定义了新的属性 FOREIGN

该属性用于构造体和子程序中以连接非 VHDL 模块。下面就是使用属性 FOREIGN 的几种基本格式。

1. 用于构造体

```
ENTITY nand IS
    GENERIC(N:positive:=2);
    PORT(input:IN BIT_VECTOR(N DOWNTO 1);
         output:OUT BIT);
    END ENTITY nand;

    ARCHITECTURE NonVHDL OF nand IS
      ATTRIBUTE FOREIGN OF NonVHDL:ARCHITECTURE
      IS "NonVHDL_nand(A, B, C)";
BEGIN

END ARCHITECTURE NonVHDL;
```

2. 用于子程序

(1) 用于过程

```
PROCEDURE print_line(a:STRING)IS
    ATTRIBUTE FOREIGN OF print_line:PROCEDURE
    IS "putline(a)";
BEGIN

END PROCEDURE print_line;
```

(2) 用于函数

```
PACKAGE p IS
    FUNCTION atoi(s:STRING)
        RETURN INTEGER;
```

```
ATTRIBUTE FOREIGN OF atoi:
    FUNCTION IS "/bin/sh atoi";
```

11.1.6 语句描述上的区别

93 版本对 87 版本上的语法进行统一，使语句格式更加规范。

1. COMPONENT 语句

```
COMPONENT c IS
    ⋮
END COMPONENT  c;
```

2. PROCESS 语句

```
PROCESS(…)  IS
    ⋮
END PROCESS;
```

3. 构造体描述

```
ARCHITECTURE a OF e IS
    ⋮
END ARCHITECTURE  a;
```

4. PROCEDURE 语句

```
PROCEDURE p IS
    ⋮
END PROCEDURE  p;
```

5. FUNCTION f(…) IS

```
    ⋮
END FUNCTION  f;
```

6. 实体描述

```
ENTITY e IS
    ⋮
END ENTITY  e;
```

7. 配置描述

```
CONFIGURATION c OF e IS
    ⋮
END CONFIGURATION  c;
```

8. 包集合描述

```
PACKAGE pk IS
    ⋮
END PACKAGE  pk;
```

11.1.7 扩展标号标注

在任何顺序语句前面都可以加标号进行标注。例如：

```
L1:IF a=b THEN
   c:=d;
END IF L1;
L2:sum:=(a XOR b)XOR c;
```

11.1.8 纯函数和非纯函数

在 93 版中函数可以设计成纯函数和非纯函数。一个纯函数指的是，在所带参量值相同时，调用后应返回一个相同值；而在非纯函数情况下，即使函数所带的参量值相同，其调用后可能会返回不同值。例如：

- 纯函数

```
PURE FUNCTION "AND"(L, R:X01Z)
          RETURN X01Z IS
BEGIN
    RETURN TABLE_AND(L, R);
END FUNCTION "AND";
```

- 非纯函数

```
IMPURE FUNCTION RANDOM RETURN REAL;
```

这是一个非纯函数描述，RANDOM 是产生随机值的函数，不同次调用该函数就会得到不同的返回值。当然，该函数没有输入参数。

11.1.9 “标识”(Signature)

它能显式地识别子程序和枚举字符的越界。该“标识”能显式地说明参数和结果的概略情况。例如：

```
ATTRIBUTE BUILT_IN OF "_"[STD_LOGIC_
   VECTOR, STD_LOGIC_VECTOR RETURN STD_LOGIC_
   VECTOR]:FUNCTION IS TRUE;
```

上例中方括号[ ]内是“标识”部分，它限定了参加操作的数是位矢量，返回结果也是位矢量。

11.1.10 文件操作定义

在一个文件类型说明之后，隐式定义的文件操作将使文件被再次定义。这些操作是：

```
FILE_OPEN;
FILE_CLOSE;
READ;
WRITE;
ENDFILE
```

上述 4 个操作为过程，最后一个操作用函数来实现。

例如：

```
TYPE index IS RANGE 0 TO 15;
```

```
TYPE int_ftype IS FILE OF index;
    PROCEDURE file_open(FILE f:int_ftype;
        external_name:IN STRING;
        open_kind:IN file_open_kind:=READ_MODE);
    PROCEDURE file_open(STATUS:OUT file_open_status;FILE f:int_
        ftype;external_name:IN STRING;
        open_kind:IN file_open_kind:=REAN_MODE);
    PROCEDURE file_close(FILE f:int_ftype);
    PROCEDURE read(FILE f:int_ftype;
                   value:out index);
    PROCEDURE write(FILE f:int_ftype;
                    value:IN index);
    FUNCTION endfile(FILE f:int_ftype)
                   RETURN BOOLEAN;
```

11.1.11 扩大了属性使用范围

在 93 版本中文字、单元、组和文件都可以使用属性，例如：

```
GROUP pinzpin IS(SIGNAL, SIGNAL);
GROUP clk2q:pinzpin(clk, q);
ATTRIBUTE tpLH:DELAY_LENGTH;
ATTRIBUTE tpLH OF clk2q:GROUP IS 12ns;
        ⋮
q⇐GUARDED d AFTER clkzq'tpLH;
```

11.1.12 增加了逻辑操作

增加了 XNOR 操作、移位操作和循环操作。它们是：

SLL——逻辑左移；
SRL——逻辑右移；
SLA——算术左移；
SRA——算术右移；
ROL——逻辑循环左移；
ROR——逻辑循环右移；
这些操作的含义与汇编语言中所定义的内容是一致的。

11.1.13 Report 语句(报告语句)

它与断言语句相似，但没有断言表达式，如：

```
R1:REPORT"This code should not have
          been entered!"
   SEVERITY NOTE;
```

该语句在仿真时使用。

11.1.14 信号延时可指定脉冲宽度限制

在信号延迟表达式中 REJECT 用来限制脉冲宽度。如：

```
dout1<=a AND b AFTER 5 ns;
dout2<=REJECT 3ns INERTIAL a AND b
        AFTER 5 ns;——脉冲宽度限制为3 ns
```

11.1.15 可对信号赋无效值

可以对信号赋一个无效值，以表明不改变当前驱动器的输出值。例如：

```
a<=NULL;
```

执行该条语句，a 的信号值将不发生变化。

11.1.16 延迟过程

一个过程可以标识为延迟过程，该过程仅仅在某一时段结束时才执行，也就是在一个时段的所有 Δ 延时之后才执行。延迟过程的标识符是“POSTPONED”，例如：

```
p1:POSTPONED nand_1(a, b:IN BIT;
                    c:OUT BIT);
```

实际上，并发断言语句、并发过程调用和并发的信号代入语句都可以标识为延迟执行的语句。延迟进程机制的典型应用是延迟断言语句。在一个仿真时刻不作多次检测，不但避免了误警，而且也节省了仿真时间。

11.1.17 COMPONENT 语句、实体——构造体或配置的直接说明

文件的直接说明可简化描述方法。在结构化描述时通常需要文件说明或配置，而直接说明则可省略。一个“与非”门的直接说明的构造体描述可写为：

```
ARCHITECTURE direct_Nand OF Nand_2 IS
BEGIN
  G:ENTITY work.Nand_2
        GENERIC MAP(tpLH=>13 ns, tpHL=>14 ns)
        PORT MAP(I1=>a, I2=>b, o=>c);
  END ARCHITECTURE direct_Nand;

  G:CONFIGURATION Nand_2_Final GENERIC MAP(tpLH=>13 ns,
      tpHL=>14 ns)
        PORT MAP(I1=>a, I2=>b, o=>c);
```

11.1.18 GENERATE 语句可含端口说明部分

在 GENERATE 语句中可以包含说明的端口，例如：

```
Label:IF N MOD 2=1 GENERATE
```

```
imstance:COMPONENT_NAME PORT MAP
        (t1, t2);
END GENERATE;
```

11.1.19 扩展了字符集

93版本的字符集扩展成256个字符的字符集，具体定义请参见文献[8]。

11.1.20 定义了扩展标识符

扩展的标识符是写在两个黑斜杠之间的一系列字符，例如：

```
\VHDL ENTITY\, \a AND b\
\ADDER\, \adder\
```

扩展标识符和一般标识符使用目的相同。但是，一般标识符受 VHDL 语言的保留字限制，而扩展标识符则不受这一限制。扩展标识符中的大小写字母是不一致的。如上例中\ADDER\和\adder\是两个不同的标识符。

11.1.21 位串

位串是用双引号括起来的扩展的数字序列，例如：

```
B"001_101_010"——9位二进制位串
X"A_F0_FC"   ——20位十六进制位串
O"3701"      ——12位八进制位串
X" "         ——空位串
```

11.1.22 增加了预定义属性

在 93 版中增加了如下的预定义属性：

```
'ASCENDING;
'IMAGE;
'VALUE;
'DRIVING_VALUE;
'SIMPLE_NAME;
'INSTANCE_NAME;
'PATH_NAME.
```

1. 'ASCENDING 属性

例如：

```
VARIABLE axe:BIT_VECTOR(0 TO 63);
CONSTANT max:POSITIVE:=12;
TYPE two_d_arr IS ARRAY(i TO max,
      63 DOWNTO 0) OF STD_LOGIC;
SIGNAL box:two_d_arr;
```

如果 box 索引范围是升序排列的，那么 box'ASCENDING 返回值为 TRUE；否则返回

值为 FALSE。例如：

box'ASCENDING——一维范围升序，故返回 TRUE

box'ASCENDING(2)——二维范围为降序，故返回 FALSE

axe'ASCENDING——返回 TRUE

2. 'IMAGE 和'VALUE 属性

'IMAGE 属性是取一个标量值并产生一个串表示；而'VALVE 属性是取一个标量值的串表示并产生它的等价值，例如：

```
TYPE test IS(A, \A\, 'A');          ——枚举型
TYPE numeric IS RANGE 1 TO 16;      ——整型
TYPE cap IS 0 TO 5000
    UNITS
      Pf;
      nf=1000Pf;
    END UNITS;                      ——物理型
test'IMAGE(A)                       ——"A"
test'IMAGE('A')                     ——"'A'"
numeric'IMAGE(12)                   ——"12"
cap'IMAGE(5nf)                      ——"5000Pf"
cap'VALUE("2000Pf")                 ——2nf 或2000Pf
test'VALUE("A")                     ——A
numeric'VALUE("13")                 ——13
```

3. DRIVING_VALUE 属性

DRIVING_VALUE 属性将取得当前进程中的信号值。例如：

```
PROCESS
BEGIN
      ⋮
  IF a='0' THEN
    car<=NULL;
  ELSE
      car<='1';
  END IF;
      ⋮
END PROCESS;
```

car'DRIVING_VALUE——如果当前时刻 car='0'，则 car'DRIVING 取得'0'；如果 car='1'，则取得'1'。

4. 'SIMPLE_NAME 属性

该属性将取得所指定命名项的名字，如标号名、变量名、信号名、实体名和文件名等，例如：

```
SIGNAL clk:BIT;
```

```
TYPE mc_state IS (READY, WAITING, HOLD,
                  RUNNING);
VARIABLE \wait\:STD_LOGIC;
TYPE abc IS('A', 'B', 'C');
clk'SIMPLE_NAME——"clk"
ready'SIMPLE_NAME——"ready"
\wait\'SIMPLE_NAME——"\wait\"
'c''SIMPLE_NAME——"'c' "
```

5. 'INSTANCE_NAME 属性

该属性将给出指定项的路径，例如：

```
BIT_ARITH'INSTANCE_NAME——":att:bit_arith:"
```

该属性指明了 BIT_ARITH 这个包集合，在设计库 att 中提供。

6. 'PATH_NAME 属性

'PATH_NAME 属性和'INSTANCE 属性非常类同，但是它们有一点区别，'PATH_NAME 属性将不显示说明的设计单元，例如：

```
full_adder'INSTANCE_NAME——":full_adder(dataflow):"
full_adder'PATH_NAME——"full_adder:"
```

除增加上述预定义属性外，93版本删除了87版中的'STRUCTURE 和'BEHAVIR 预定义属性。

11.1.23 扩充了标准包集合(STANDARD)

在标准包集合中增加了如下的一些内容：

- DELAY_LENGTH 物理子类型；
- FILE_OPEN_KIND 枚举类型；
- FILE_OPEN_STATUS 枚举类型；
- 'FOREIGN 属性说明。

11.2 87 版到 93 版的移植问题

下面仅就移植过程中应注意的几个问题作一说明。

- 枚举类型 CHARACTER 成为一个更大的集合，根据类型 CHARACTER'HIGH 和 CHARACTER'RIGHT 所得到的代码会发生某些改变。
- 由并置操作符所执行的操作更复杂了。
- 重新定义了文件类型说明，与之相应的隐式文件操作也重新进行了定义。
- 取消了'STRUCTURE 和'BEHAVIOR 属性。因此源代码中存在此属性的地方要去掉。
- 增加了新的保留字，因此使用这些保留字作标识的地方应改掉。

前面详述了 93 版本和 87 版本的主要不同点，其实还存在许多细微的差别，这里无法一一详述和列举。欲知详情的读者请参阅 93 版手册。

第 12 章

MAX+plus Ⅱ使用说明

12.1 MAX+plus Ⅱ概况

MAX+plus Ⅱ开发工具是美国 Altera 公司自行设计的一种 CAE 软件工具，其全称为 Multiple Array Matrix and Programmable Logic User Systems。它具有原理图输入和文本输入(采用硬件描述语言)两种输入手段，利用该工具所配备的编辑、编译、仿真、综合、芯片编程等功能，将设计电路图或电路描述程序变成基本的逻辑单元写入到可编程的芯片中(如 FPGA 芯片)，做成 ASIC 芯片。它是 EDA 设计中不可缺少的一种有用工具，目前在国内使用较为普遍。但是，由于该工具是针对可编程芯片而设计的，因此它不支持系统行为级的描述和仿真，某些 VHDL 语言中的语句如 WAIT 语句等将得不到支持，这一点务请使用该工具的读者注意。

12.1.1 系统安装

MAX+plus Ⅱ最低硬件平台为 486 微机、16 MB 内存、500 MB 硬盘。在 PC 机价格不断下降的今天，为了工程设计需要，建议硬件平台最好设置为 586 微机、64 MB 内存、4.3 GB 硬盘。

目前 MAX+plus Ⅱ有 3 种版本：商业版、基本版和学生版。

商业版——可以进行各种文件和图像的输入，并可进行功能分析、时序分析、下载其厂家的各种已设计芯片。商业版软件为避免盗版，在并行口上需加一个硬件“狗”，出售的系统除一张光盘外，还应带一个“狗”。该“狗”中存有一长串由英文大、小写字符和数字组成的密码。每次系统启动都要通过“狗”来核对用户的合法性。

基本版——在商业版上对功能加以限制，时序分析、VHDL 语言综合等功能不能使用。系统启动不需要硬件“狗”支持，只要向 Altera 公司申请一个基本版受权码就可以工作。

学生版——在商业版基础上再加以限制。若想安装学生版可以向 Altera 公司大学项目部申请一个学生版授权码就可以工作。

下面以商业版安装为例，简述一下具体的安装过程：

(1) 首先将 MAX+plus Ⅱ光盘插入光驱。

(2) 在 Windows 95 界面下，将光标移至 Windows 资源管理器并单击鼠标左键，屏幕显示各驱动器的文件名称。此时双击光驱所对应的驱动器号(例如现在光驱号为 E 盘)，屏幕刷新显示光盘的所有文件夹。

(3) 双击屏幕上显示的 PC 文件夹，屏幕刷新再显示 PC 文件夹的内容。接着双击 maxplus2 文件夹，找到 install 应用程序，双击该应用程序图标后就进入 install 界面。

(4) 屏幕显示版本号、日期、安装和不安装按钮，如：

Version 8.1 9/12/97

[install] [uninsall]

把光标移至 install 按钮，单击鼠标左键。

(5) 屏幕显示默认的系统目录和用户指定的目录。如：

System Directory

[C：\maxplus2]

Working Directory

[C：\max2work]

User—Specified Directory

C：\maxplus2

This directory does not exist OK to create it?

[Yes] [No]

Working Directory：

C：\max2work

[Continue] [Select Directory]

如果按默认信息工作，系统和用户目录均在 C 盘。若想装入其它盘，只需在屏幕上进行修改就行了。用鼠标单击 Yes 按钮和 Continue 按钮，安装程序向下执行。

(6) 屏幕显示 Purchased、Custom 和 Full 三个按钮，供用户选择安装内容。

Purchased——购买的基本系统；

Custom——通常使用的系统；

Full——系统全部内容。

如果磁盘容量足够大，可选择 Full。

(7) 单击 Full 按钮后，屏幕显示要求用户输入授权码：

MAX+plus Ⅱ Authorization Code

[ ]

输入与硬件"狗"对应的授权码后，屏幕显示安装信息及 Yes 和 No 按钮。单击 Yes 按钮以后，安装工作就立即开始。

(8) 安装完毕后屏幕上出现 MAX+plus Ⅱ 8.1 图标，关闭安装画面，将该图标移至合适位置(将光标移至图标处，然后按住鼠标左键就可以实现图标的拖动)。

至此，整个系统安装完毕。以后只要启动 Windows 95，并双击该图标即可进入 MAX+plus Ⅱ 系统。

12.1.2 MAX+plus Ⅱ 对 VHDL 的支持

MAX+plus Ⅱ 早期版本并不支持 VHDL 语言，当前版本如 MAX+plus Ⅱ Version 8.1 已较好地支持 VHDL 语言。

1. 支持 VHDL 语言的 87 版本和 93 版本

MAX+plus Ⅱ 支持 VHDL 语言的 87 版本和 93 版本。在对编辑好的 VHDL 语言程序模块进行编译时，应首先设置版本选择。进入编译状态后，单击屏幕上方的 Interfaces 选项，此时出现下拉菜单，然后单击其中的 VHDL Netlist Reader Settings 选项，屏幕就出现

VHDL Version

⊙ VHDL 1987　　○ VHDL 1993

点击圆圈，使中心出现实心点，那么选中的将是后跟的版本，上例选中的是 87 版本。点击 VHDL 1993 项前面的圆圈就可以选择 93 版本。应注意，程序编写的格式和语句应和所选择的版本一致。

2. 支持 VHDL 语言的库

在 MAX+plus Ⅱ 系统中有 3 个库能支持 VHDL 语言。它们是 IEEE 库，Altera 库和 Lpm 库。

IEEE 库提供基本的逻辑运算函数及数据类型转换函数等，它在本书附录 A 中已有说明。

Altera 库和 Lpm 库提供了一些基本的函数、宏函数及常用电路的基本设计单元供编程人员使用，具体请查阅 MAX+plus Ⅱ 的 Help 文件。

3. 对 VHDL 语句的支持

MAX+plus, Ⅱ 的编译器支持基本的 VHDL 语句，由于 MAX+plus Ⅱ 是针对能生成写入可编程器件的网表而设计的，也就是它只支持 RTL 描述方式，而不支持行为描述方式。例如，WAIT 语句可在进程中的第一条语句使用，其它地方使用将不支持，具体支持细节请参阅 MAX+plus Ⅱ 的 Help 文件。

12.1.3 MAX+plus Ⅱ 系统的启动

当 PC 机引导 Windows 95 完毕后，屏幕将出现 MAX+plus Ⅱ 的图标，用鼠标双击该图标就使系统进入 MAX+plus Ⅱ 的主菜单，如图 12-1 所示。在该主菜单上有 5 个选项，点击各选项即可进入不同的功能。

各选项下还有子菜单，某些重要的主菜单选项以图标形式列在屏幕上，如打开文件、现行编辑文件存盘等。

屏幕的标题条显示 MAX+plus Ⅱ Manager 和它的子目录，如果用户自己建立了工程目录，那么在子目录位置将显示驱动器到工程目录名的一串对应字符。图 12-1 显示的是：

MAX+plus Ⅱ Manager c:\max2work\counter

在这里由用户建立的工程文件名是 counter。

图 12-1 MAX+plus Ⅱ 主菜单

12.2 建立和编辑一个 VHDL 语言的工程文件

12.2.1 新文件的编辑

建立和编辑一个 VHDL 语言的工程文件，是数字系统或数字逻辑电路设计的第一步。下面以第 1 章中的约翰逊计数器设计为例，叙述一下在 MAX+plus Ⅱ 系统中怎样来建立一个工程文件的步骤。

(1) 在屏幕上用鼠标点击 File 选项，此时出现子菜单如图 12-2 所示。由于是输入新

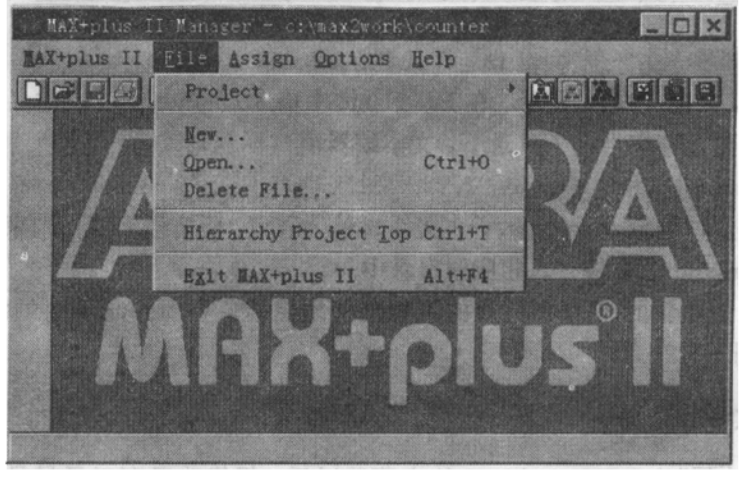


图 12-2 File 选项子菜单

文件，故点击子菜单中的 New 项，接着屏幕上出现 New 的对话框。在对话框内可供用户选择的 4 种编辑方式为：图形编辑、符号编辑、文本编辑和波形编辑，如图 12 - 3 所示。

(2) VHDL 语言的工程文件编辑属于文本编辑，那么应该点击 New 对话框中的第三项，当该项字符被虚线框框住，并在该项前面的圆圈中出现一个实心点时，表明该项已被选中，系统将进入文本编辑状态。

(3) 点击 New 对话框的 OK 按钮，屏幕上将出现一个无名的编辑窗口，如图 12 - 4 所示。

(4) 在无名的文本编辑窗口下任意输入一个字符后，点击图标的第 3 项或点击 File→Save(File→Save As)，在屏幕上会弹出一个

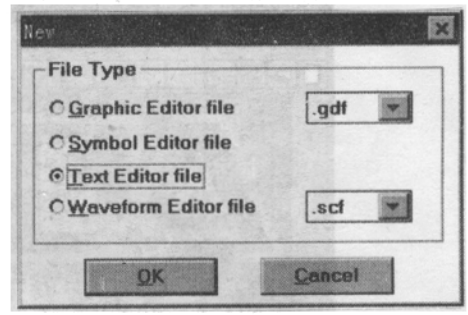


图 12 - 3　New 对话框

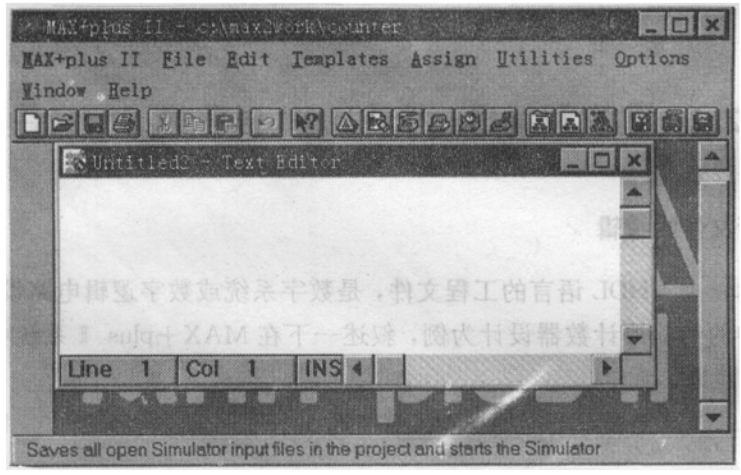


图 12 - 4　无名的文本编辑窗

Save As 对话框，如图 12 - 5 所示。在 File Name 中输入 counter. vhd 文件名，点击 OK 按钮，该文件名即可保存在当前子目录下，文本编辑窗口标题则改为：

counter. vhd－Text Editor

(5) 在编辑窗口中输入 counter. vhd 文件的对应语句，所有语句输入结束，点击第 3 项图标，把 counter. vhd 源文件存入相应子目录中。

到此为止，一个新的 VHDL 语言的工程文件编辑结束，counter. vhd 的源程序清单请参见第 1 章的例 1 - 2。

12. 2. 2　文件的修改

如果系统启动进入 MAX＋plus Ⅱ 的主菜单以后，想修改已存于子目录中的 counter. vhd 文件，那么应按如下的步骤工作：

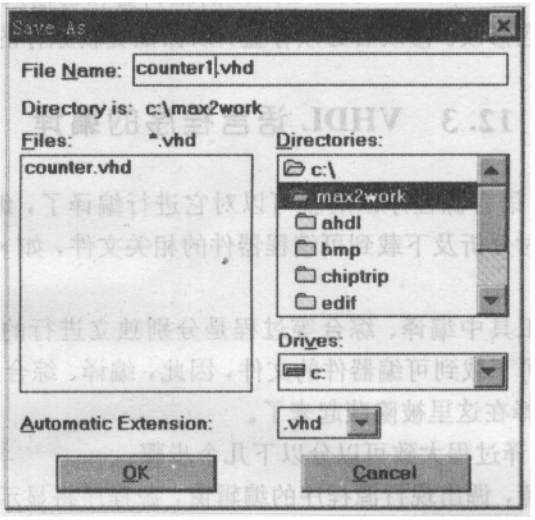


图 12－5　Save As 对话框

(1) 点击 File 选项，再点击子菜单中的 Open 项，屏幕就出现 Open 对话框，如图 12－6 所示。

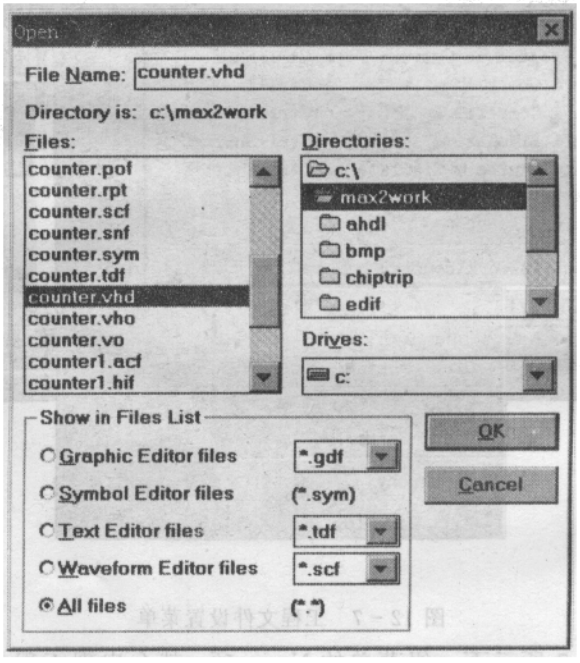


图 12－6　Open 对话框

(2) 点击左下角的显示文件框中的文本编辑文件项(第 3 项)，并不断点击该项后面的下拉标志，直到 *.vhd 项出现为止。再点击框中的 *.vhd 项，在 Open 对话框的左上角的

文件框中即可显示出 counter. vhd 文件名。点击该框中所列出的 counter. vhd，并点击 OK 按钮，屏幕上即可显示 counter. vhd－Text Editor 的编辑窗和已输入的源程序清单。如果要进行修改，则可以作相应修改。修改后必须存盘，以保证更新原存的文件。

12.3 VHDL 语言程序的编译

在建立新的 VHDL 语言源程序以后就可以对它进行编译了，编译最终的目的是为了生成可以进行仿真、定时分析及下载到可编程器件的相关文件，如 *.cnf，*.rpt，*.snf，*.pof 等。

在某些 EDA 设计工具中编译、综合等过程是分别独立进行的。而在 MAX＋plus Ⅱ中，由于其目的是形成可下载到可编器件的文件，因此，编译、综合过程是一气呵成的。综合过程中的某些参数选择在这里被隐蔽起来了。

MAX＋plus Ⅱ的编译过程大致可以分以下几个步骤：

(1) 按前面所述步骤，调出现行源程序的编辑窗，源程序将显示在编辑窗中。

(2) 点击 File 选项，光标移至子菜单的 Project 项停留几秒钟，屏幕上又会出现下一级菜单，如图 12－7 所示。这时可以对编译的工程文件进行设置。

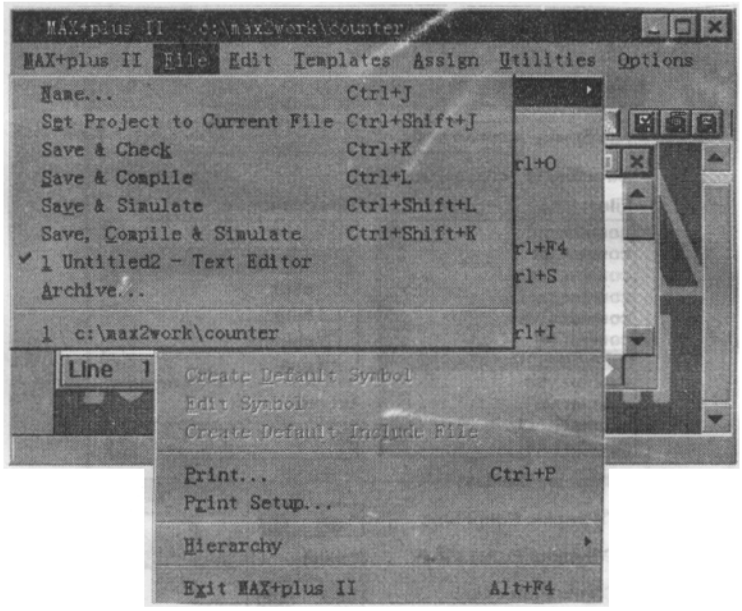


图 12－7　工程文件设置菜单

(3) 点击图 12－7 所示下一级菜单的 Name 项，就会出现工程名(Project name)对话框，输入程序名，如 counter 或用光标点击左下角框内的现有工程文件 counter，就可以确定当前的工程文件名，然后点击 OK 按钮结束工程文件名设置。

(4) 点击 File，按(3)方法再进入下一级菜单，再点击 Set project 选项，使得编译器指向已设定的现行工程文件 counter。

(5) 点击主菜单 MAX+plus Ⅱ 选项，出现子菜单，如图 12 - 8 所示。再点击 Compiler 选项，屏幕上就出现编译对话框，如图 12 - 9 所示。

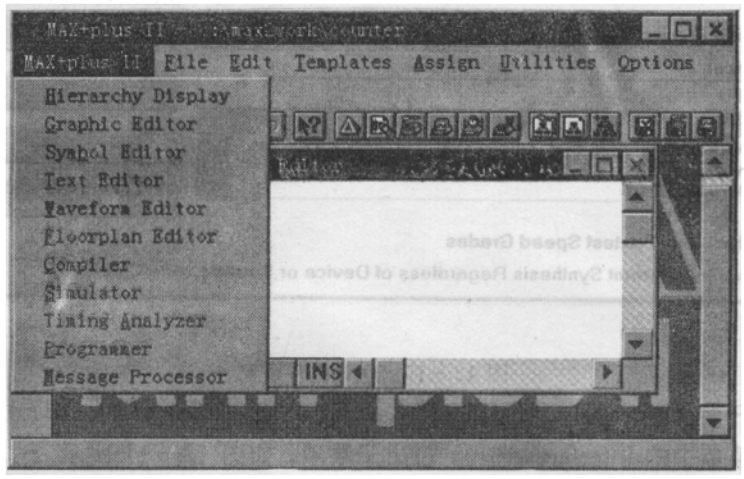


图 12 - 8 MAX+plus Ⅱ 选项子菜单

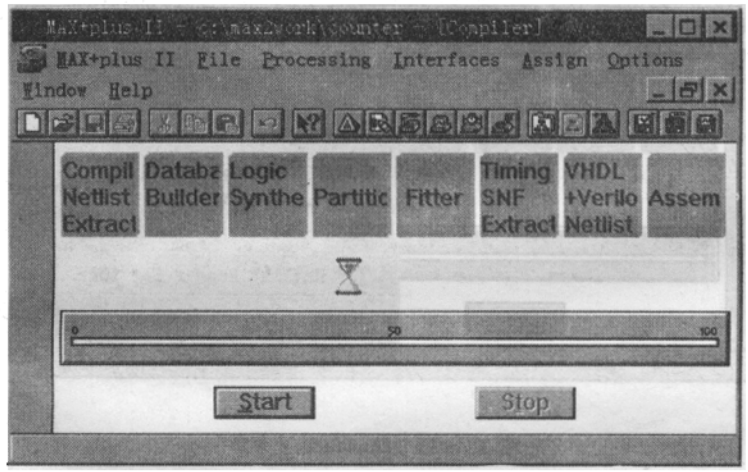


图 12 - 9 编译对话框

(6) 由于编译、综合结果要生成适于写可编程器件的文件，所以在编译以前应选定最终要下载的芯片。在 MAX+plus Ⅱ 的工具中，可以支持的下载芯片有多种，根据所设计的逻辑电路规模，用户可以自由地进行选择。

点击主菜单的 Assign 选项，再点击子菜单中的 Device 项得到 Device 对话框，如图 12 - 10 所示。

利用 Device Family 框中的上移和下移标志，寻找系统中适用的芯片名，找到以后用鼠标点击确定。图 12 - 10 中选取的是 MAX 7000，单击 OK 按钮，此后系统指定的下载器件被指定为 MAX 7000。

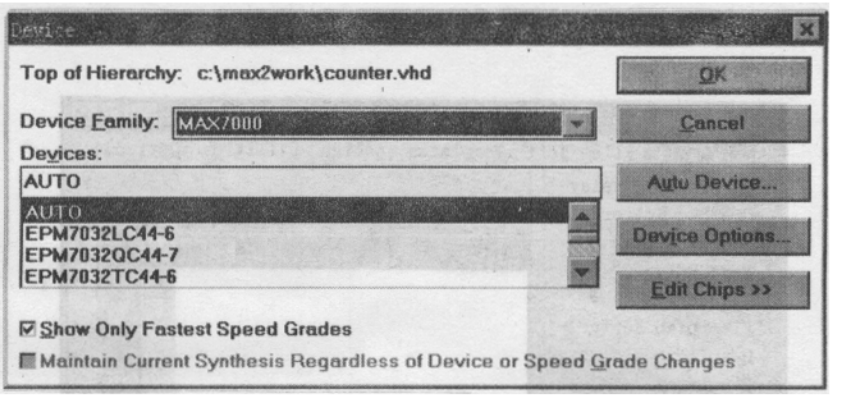


图 12 - 10　Device 对话框

(7) 编译开始前还应确定源程序的版本及用户的附加库。此时在编译对话框已打开情况下，点击主菜单中的 Interfaces 选项，拉出子菜单，如图 12 - 11 所示。

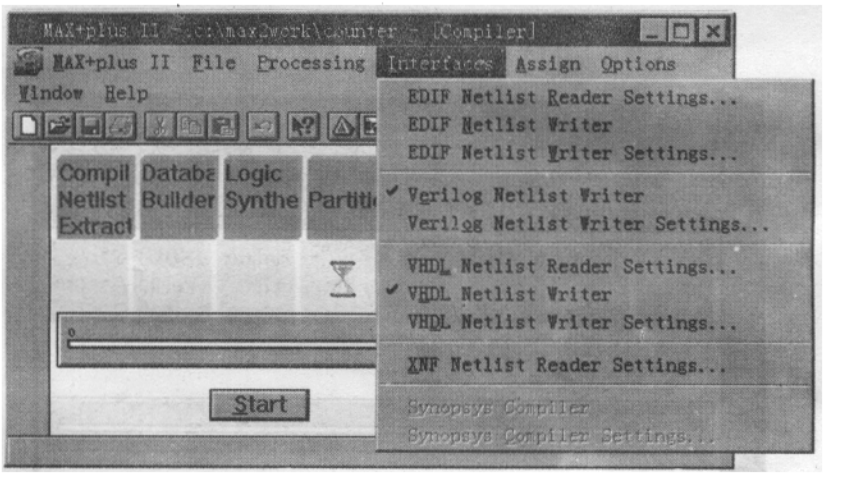


图 12 - 11　Interface 子菜单

点击 EDIF Netlist Reader Settings ... 项，就可在屏幕上显示 VHDL Netlist Reader Settings 对话框，如图 12 - 12 所示。

点击 VHDL Version 框中的 VHDL 1987 或 VHDL 1993 就可选择相应的版本编译器，图中选择的是 1987 版本。

如果要加工程库，则可作相应添加。点击 OK 按钮即可完成 VHDL 网表阅读器的设置。

(8) 前面 7 步完成了编译前的准备及必要的设置工作。此时只要点击编译对话框中的 Start 按钮，编译器即可启动。在编译过程中如果编译器发现源程序语法有错或设置有错就会自动弹出错误信息窗进行提示。如果没有错，编译结束会通过 MAX+plus Ⅱ -Compiler 信息窗提示：

Project compilation was successful

0 errors

0 warnings

接着点击框中的 OK 键，整个编译即可结束。

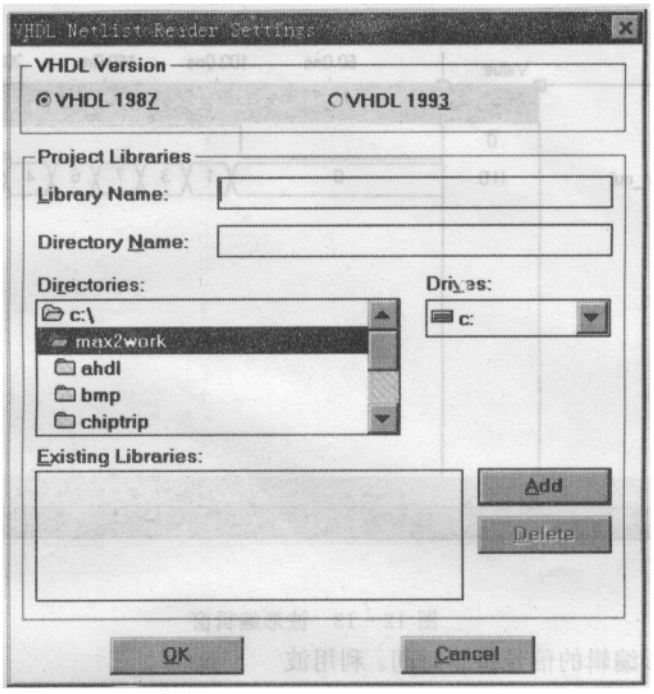


图 12－12　VHDL Netlist Reader Settings 对话框

12.4　VHDL 语言程序的仿真

如第 8 章所述，仿真是为了验证所编写的 VHDL 语言程序的功能是否正确。在 MAX＋plus Ⅱ的工具中仿真输入信号是通过波形编辑器生成 *.scf 文件的。*.scf 文件和编译生成的文件连接就可以实现程序的仿真。VHDL 语言程序的仿真过程大致可以分生成仿真波形文件、仿真和定时分析 3 个步骤。

12.4.1　生成仿真波形文件

生成仿真波形文件的具体方法如下：

(1) 与编译时一样，应设置当前工程文件名，如当前仿真的工程文件名为 counter。

(2) 点击主菜单的 MAX＋plus Ⅱ选项，弹出子菜单，如图 12－8 所示。点击 Waveform Editor 选项，就可在屏幕上显示波形编辑窗口。与文本编辑时一样，利用 File 选项设置 counter.scf 文件名。设置完成以后，编辑窗就出现 counter.scf－Waveform Editor，如图 12－13 所示。在未输入信号名以前整个窗口是空白的。

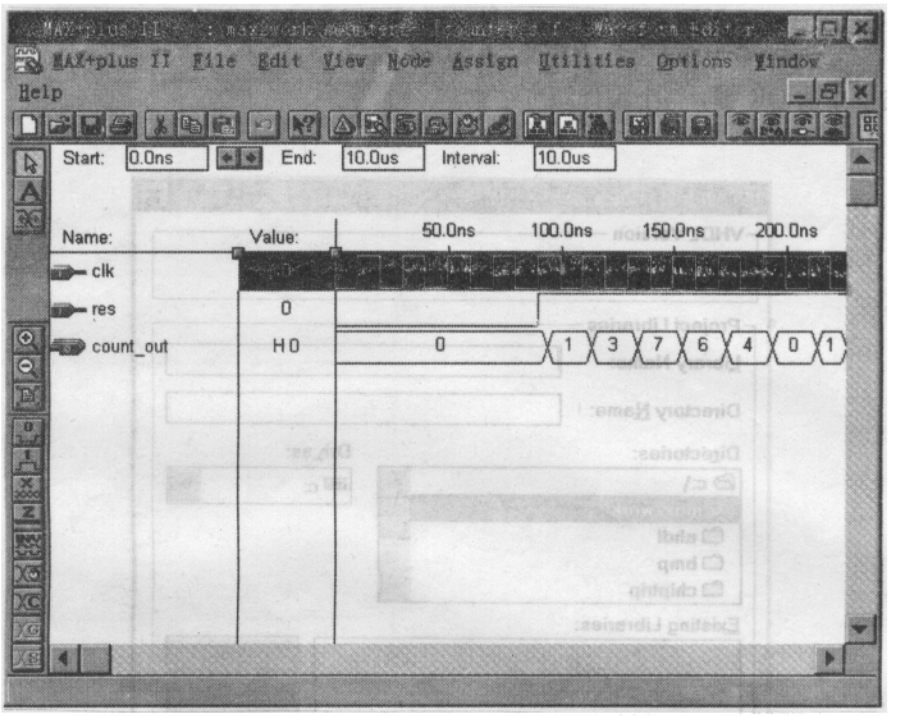


图 12 - 13　波形编辑窗

(3) 确定波形编辑的信号持续时间。利用波形编辑器所生成的信号持续时间是有限的，它可以根据用户需要进行相应调整。

在屏幕显示波形编辑窗情况下，点击主菜单 File 选项，再点击子菜单的 End Time 项，屏幕就弹出一个 End Time 对话框，如图 12 - 14 所示。修改 Time 框中的时间，并点击 OK 按钮就可以设置波形编辑器生成信号的持续时间。图 12 - 14 中信号持续时间设置为 10 μs。

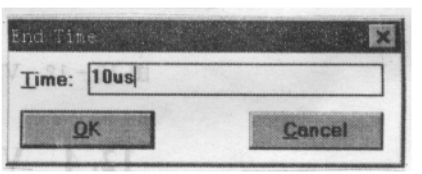


图 12 - 14　End Time 对话框

(4) 信号波形编辑。counter 有两个输入信号 clk 和 res，一个输出信号 count_out。

• 生成 clk 信号。光标指向 Name 项的第 1 行位置，点击鼠标左键，在第 1 行出现一条黑条。再双击黑条就会弹出 Insert Node 对话框，如图 12 - 15 所示。首先输入信号名(clk)，然后在 I/O Type 框中选择信号的性质是输入、输出还是双向。点击后，对应项的圆圈中出现一个实心黑点。点击 OK 按钮后在编辑窗的第 1 行位置就会出现输入信号标志和信号名(clk)。将光标移至 Value 项的第 1 行位置，点击左键，第 1 行位置自 Value 开始向后出现一条黑条。点击编辑窗左边倒数第 4 个图标(红色图标)，就会弹出一个 Overwrite Clock 对话框，如图 12 - 16 所示。图中提示时钟的基本周期为 2 ns。如果需要增加周期长度，则可在 Multiplied By 框中填入倍乘的数字，如图中填为 20，那么设定时间周期为 40 ns。点击 OK 按钮完成时钟的编辑，此时在窗口的第 1 行就会出现周期为 40 ns 的时钟脉冲波形。

• 生成 res 信号。将光标移至第 2 行位置，采用与生成时钟信号同样方法，在 Name 项

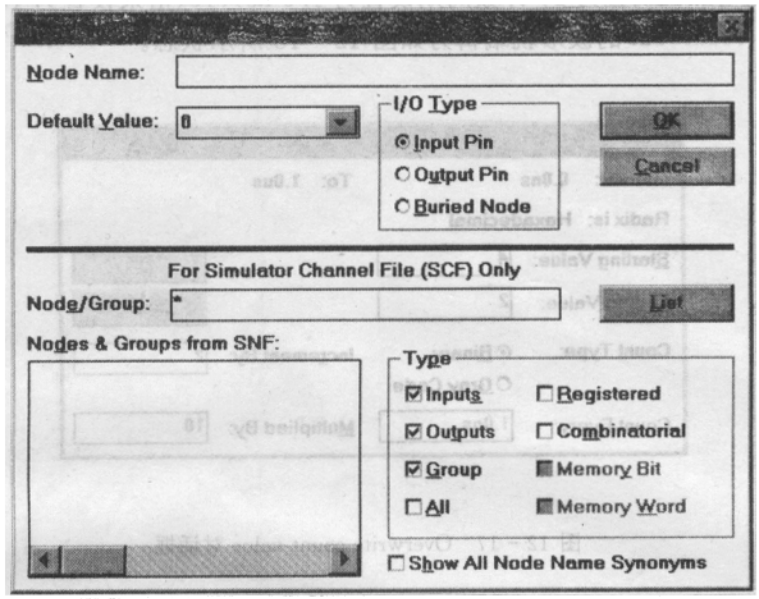


图 12 - 15　Insert Node 对话框

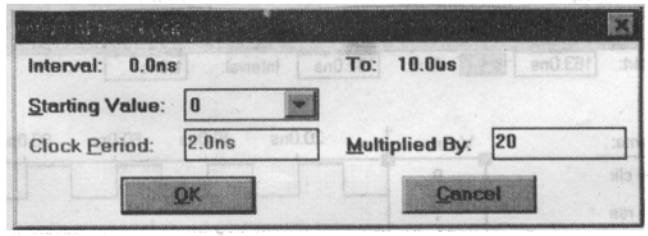


图 12 - 16　Overwrite Clock 对话框

显示输入标志和 res 信号名。由于 res 是低电平有效("0"有效)，故为了编辑方便，在 Insert Node 对话框中的 Default Value 框中的值应选择为"1"。此时 res 显示的波形是一个为"1"的高电平。为了对 counter 进行复位应该使 res 在开始一段时间保持为低电平，如图 12 - 13 所示。此时将光标移至 res 项的波形生成起始位置(0.0 ns)，然后按住鼠标左键并向右拖动，所经空间被黑条填满，一直拖到 res 应由低变高(由"0"变"1")的时间坐标为止，即可放开鼠标左键。将光标移至左边"0"图标位置(自上至下第 4 个图标)，并点击左键，res 被涂黑的那一段就会变成低电平("0")。

• 用同样方式可以编辑出 count_out 信号。输出波形最终由仿真结果填写，用户可以设置缺省值，例如设置为高阻。

上面的输入信号都是单个信号，如果是总线信号，即由多位二进制位组成的信号时，应按下面的方法进行编辑。这里以对 count_out 信号编辑为例加以说明。

• count_out 编辑成周期为 10 ns，每个周期总线信号值以加 2 递增变化。点击 count_out 的 Value 位置，使其出现一条黑条。接着点击左边倒数第 3 个图标(C)，这时屏幕弹出

Overwrite Count Value 对话框，如图 12 - 17 所示。启动值设置为 4；计数类型设置为二进制计数；计数增加值设置为 2；计数周期设置为 1.0 ns×10(倍乘设置为 10)。设置完毕后点击 OK 键，count_out 的波形就编辑为如图 12 - 18 所示状态。

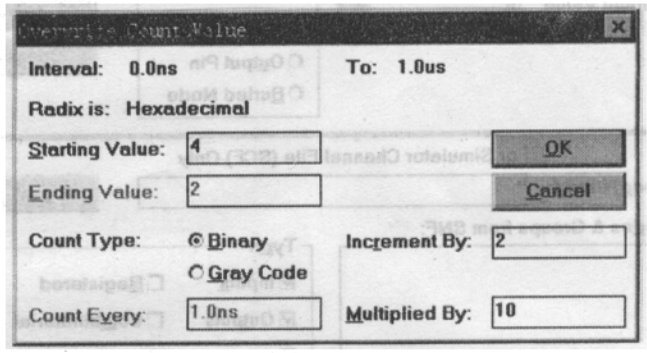


图 12 - 17　Overwrite count value 对话框

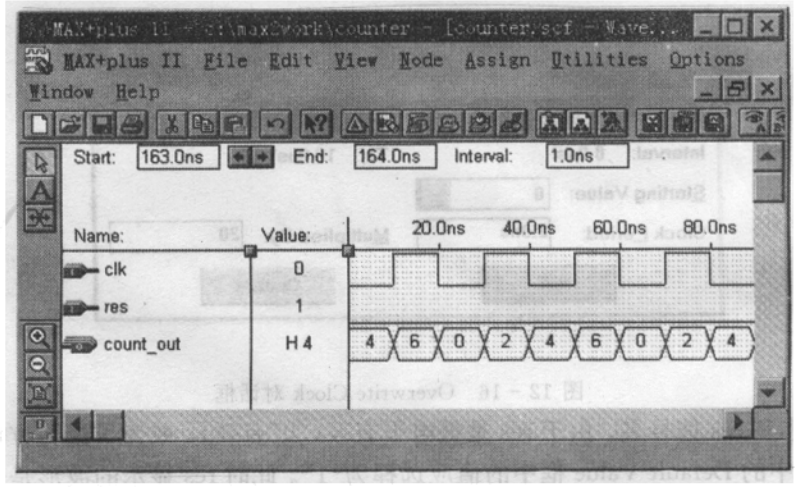


图 12 - 18　对应图 12 - 17 设置的 count_out 波形

• count_out 的某一时段总线值设置成某一个固定值。例如要使 count_out 从 0 ns 到 90 ns 这一段时间编辑取值为 5 的波形。此时将光标移至 count_out 的 0.0 ns 位置，按下鼠标的左键并向左边拖动至 90 ns 处，放开左键后就见到从 0.0 ns 到 90 ns 长的一条黑条。点击左边倒数第 2 个图标(G)，屏幕就弹出 Overwrite Group Value 对话框，如图 12 - 19 所示。在

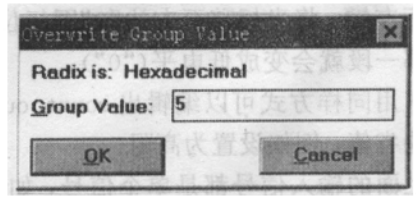


图 12 - 19　Overwrite Group 对话框

Group Value 框中填入 5 并点击 OK 按钮，即可得到如图 12 - 20 的波形。

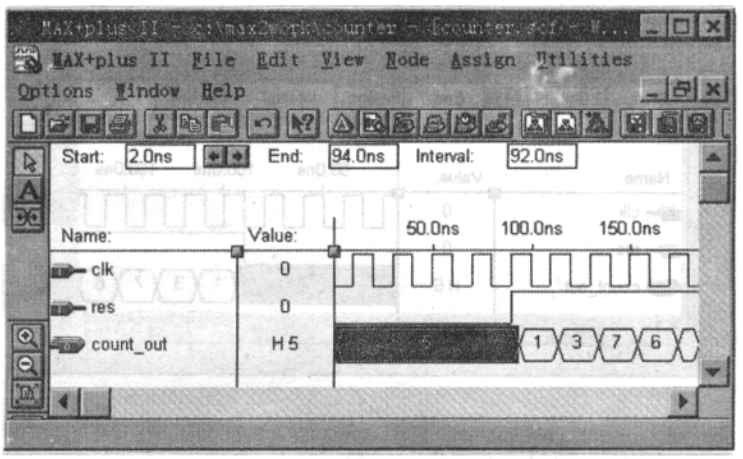


图 12 - 20　对应图 12 - 19 设置的 count_out 波形

值得注意的是，count_out 本是输出波形，其最终值应由仿真结果确定。这里仅仅是为了简单起见将它看作为输入波形，用编辑工具进行编辑示例而已。

(5) 波形的显示和观察。为了在编辑窗口中显示各种长度仿真时间的波形，并有利于观察。在波形编辑窗口左边有放大和缩小两个图标(＋和－)，点击一次对应图标，波形将放大一次或缩小一次。以此将波形缩放成便于观察的程度。例如，图 12 - 21 的(*a*)是正常波形；那么(*b*)是缩小波形；(*c*)是放大波形。

另外，拖动波形框下方的游标即可观察整个时段的波形。

按上面叙述将波形编辑文件编辑完成以后点击存盘图标即可将 count.scf 文件存入盘中以备仿真时使用。

12.4.2　仿真

在生成仿真波形文件以后，即可以开始进行仿真了。

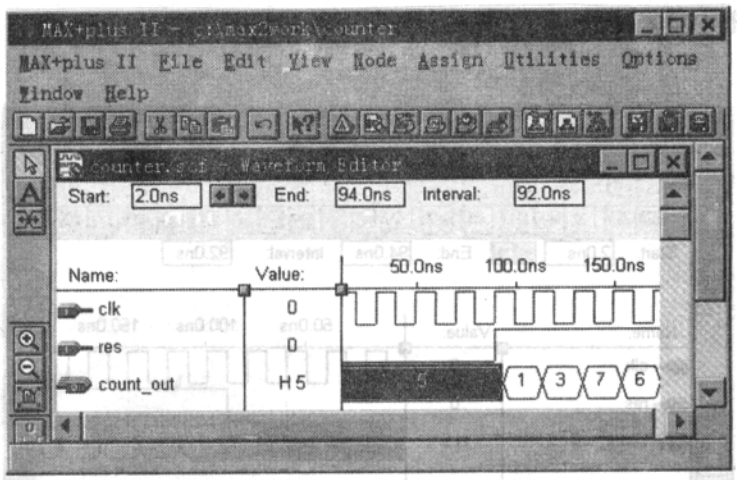
(1) 点击主菜单 MAX＋plus Ⅱ 选项，拉出子菜单并点击 Simulator 项，此时弹出 Simulator 对话框，如图 12 - 22 所示。在 End Time 应设置仿真结束时间，仿真结束时间应小于或等于波形编辑长度时间。另外还有几个仿真检查选项可根据需要进行选择。

(2) 设置完毕后，点击 Start 按钮，仿真开始进行，在仿真结束后就会弹出一个信息窗，说明有无错误信息，点击确定键以后，回到仿真对话框。再点击 Open SCF 按钮即可以显示仿真结果。

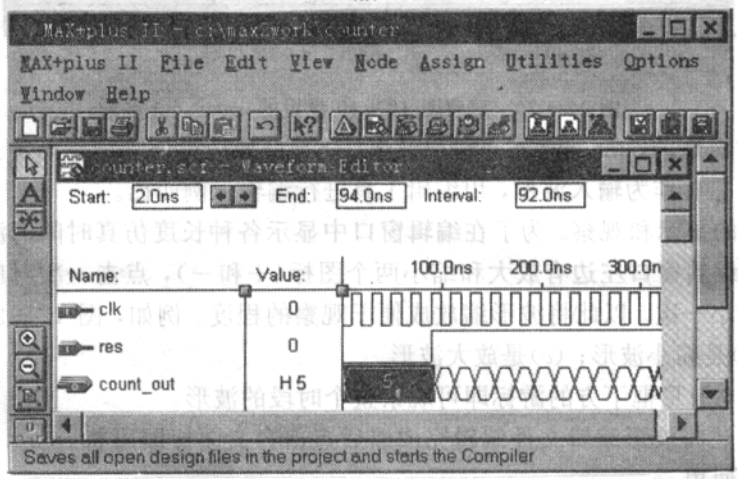
12.4.3　定时分析

仿真结果从波形上来看，很难给出定量的信号延迟关系，这一点定时分析却能直观地用表来进行显示。

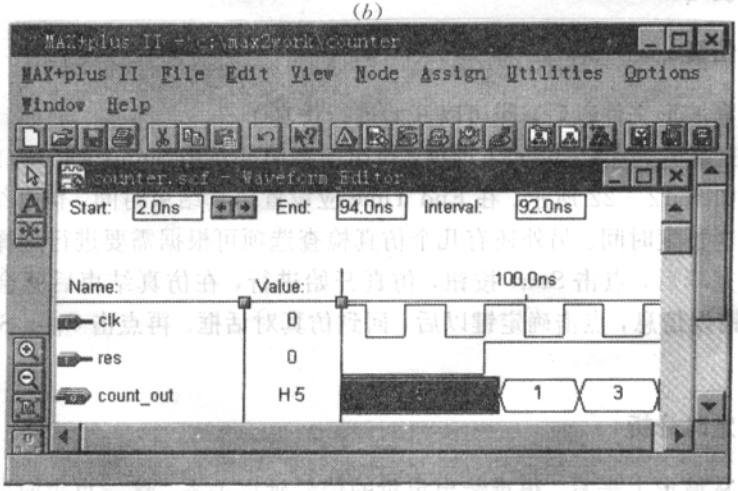
点击主菜单的 Timing Analyzer 选项，屏幕上即可显示出如图 12 - 23 的 Timing Analyzer 对话框。在对话框弹出时，表中是空白的。点击 Start 按钮后，定时分析器被启动。定



(a)



(b)



(c)

图 12 - 21　波形缩放实例

(a) 正常；(b) 缩小；(c) 放大

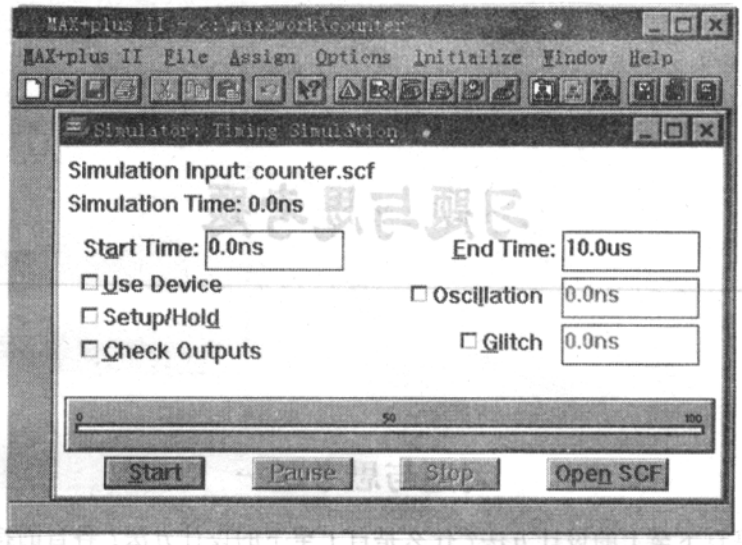


图 12 - 22　Simulator 对话框

时分析结束后弹出一个结束提示框，点击确定按钮即可返回定时分析对话框。此时表中将显示有关数据。例如图 12 - 23 的表中显示时钟触发沿与计数器的一次新的输出变化，其延迟时间为 4.0 ns。

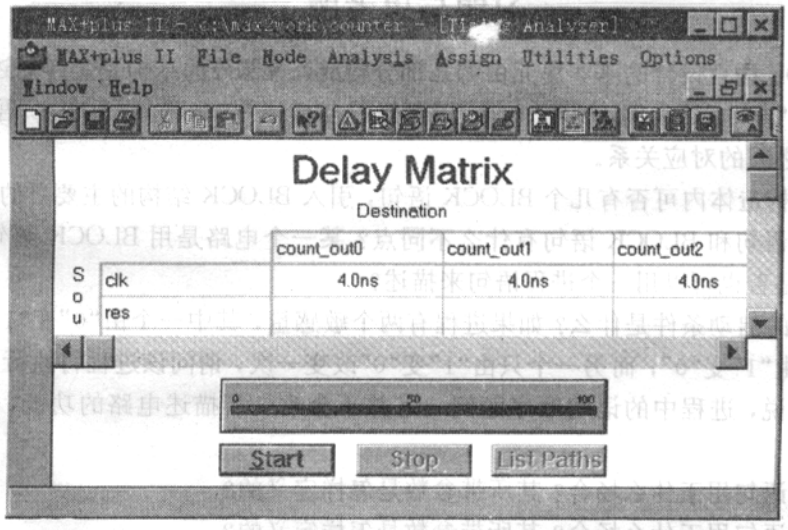


图 12 - 23　Timing Analyaer 对话框

这样，经 VHDL 语言的文本编辑、编译、仿真波形编辑、仿真和定时分析一系列设计步骤，设计出了符合要求的数字系统和数字逻辑电路。这几个步骤有时要反复循环多次才能达到设计要求。当完全满足要求后就可以通过编程器下载到指定的芯片中去，以生成 ASIC 芯片。

有关 MAX＋plus Ⅱ 编程器的使用及其它一些相关功能这里不再作介绍，另外该工具功能很强，许多选项也未作详细说明，如要深入了解请参见 MAX＋plus Ⅱ 使用说明书和工具中所携带的帮助信息。

习题与思考题

习题与思考题一

1. 什么是自下至上的设计方法？什么是自上至下的设计方法？各自的特点是什么？

2. 什么是硬件描述语言，它和一般的高级语言有什么相同点和不同点？

3. 用 VHDL 语言设计数字系统有些什么优点？

4. 用 VHDL 语言设计数字系统与传统的设计方法没有太大区别，因为它们都是为了设计出目的逻辑电路。这种说法全面吗？为什么？

习题与思考题二

1. VHDL 语言设计的基本单元由哪几部分构成？各部分的结构是怎样描述的？

2. 试以“与非”门为例，说明“与非”门逻辑符与描述“与非”门的 VHDL 语言基本设计单元各部分之间的对应关系。

3. 一个构造体内可否有几个 BLOCK 语句，引入 BLOCK 结构的主要目的是什么？

4. 进程语句和 BLOCK 语句有什么不同点？某一个电路是用 BLOCK 语句来描述的，那么其功能是否也可以用一个进程语句来描述？

5. 进程的启动条件是什么？如果进程有两个敏感量，其中一个由“0”变“1”，等待一段时间以后再由“1”变“0”；而另一个只由“1”变“0”改变一次，请问该进程将执行几遍。

6. 有人说，进程中的语句顺序颠倒一下并不会改变所描述电路的功能，这种说法对吗？为什么？

7. 过程语句用于什么场合？其所带参数是怎样定义的？

8. 函数语句用于什么场合？其所带参数是怎样定义的？

9. 库由哪些部分组成？在 VHDL 语言中常见的有几种库？编程人员怎样使用现有的库？

10. 一个包集合由哪两大部分组成？包集合体通常包含哪些内容？

11. 在 VHDL 语言中配置的主要功能是什么？试举例说明之。

12. 在 VHDL 语言中哪些部分是可以单独编译的源设计单元？

习题与思考题三

1. VHDL 语言中 3 类客体常数、变量和信号的实际物理含义是什么？

2. 信号和变量在描述和使用时有哪些主要区别？

3. 在 VHDL 语言中标准数据类型有哪几类？用户可以自己定义的数据有哪几类？

4. 若一个十二进制计数器输出 count12_out 要用整数来描述，试问应怎样进行定义？

5. 试问下面数据类型定义和操作是否正确？

```
SIGNAL  atmp: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL  btmp: STD_LOGIC_VECTOR(0 TO 7);
SIGNAL  cint: INTEGER;
SIGNAL  dtmp: STD_LOGIC_VECTOR(15 DOWNTO 0);
    atmp<=cint;
    atmp<=btmp;
    btmp<=dtmp;
```

6. 为什么要进行数据类型转换？查阅附录 C，列出几种主要转换函数的名称。

7. 在题 5 中为了实现将 cint 的值代入 atmp，应怎样使用转换函数？

8. 参阅附录 C，说明一下 STD_LOGIC、STD_ULOGIC、STD_LOGIC_VECTOR、STD_ULOGIC_VECTOR 之间的关系。下列操作是否正确？

```
SIGNAL  a: STD_LOGIC;
SIGNAL  b: STD_ULOGIC;
SIGNAL  abus: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL  bbus: STD_ULOGIC_VECTOR(7 DOWNTO 0);
    a<=b;
    abus<=bbus;
```

9. BIT 类型数据和 STD_LOGIC 类型数据有什么区别？

10. VHDL 语言有哪几类主要运算，在一个表达式中有多种运算符时应按怎样的准则进行运算？

如下 3 个表达式是否等效：

```
    a<=NOT b AND c OR d;
    a<=(NOT b AND c) OR d;
    a<=NOT b AND (c OR d);
```

11. 并置运算应用于什么场合？下面的并置运算是否正确？

```
SIGNAL  a: STD_LOGIC;
SIGNAL  eb: STD_LOGIC;
SIGNAL  b: STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL  d: STD_LOGIC_VECTOR(7 DOWNTO 0);
    b<=a&a&eb&eb;
    d<=b&eb&eb&eb&eb;
```

习题与思考题四

1. 什么是构造体的行为描述方式？它应用于什么场合？用行为描述方式所编写的VHDL语言程序是否都可以进行逻辑综合？

2. 什么是惯性延时？什么是传输延时？若图4－2中门的惯性延时时间为5 ns，试问该图应怎样进行修改？

3. 为什么要引入判决函数resolved(s)，若某一条引线上有3个输出信号作用，其状态分别为'1'，'Z'和'0'，试问最终该引线所呈现的状态应是什么？

4. 什么是RTL描述方式？它和行为描述方式的主要区别在哪里？用RTL描述方式所编写的程序是否都可以进行逻辑综合？

5. 在用RTL方式描述寄存器时应注意哪些问题？

6. 什么是构造体的结构描述方式？实现结构描述方式的主要语句是哪两个？

习题与思考题五

1. WAIT语句有几种书写格式？哪一种格式可以进行逻辑综合？

2. 试用IF语句设计一个四—十六译码器。

3. 试用CASE语句设计一个四—十六译码器。

4. 在CASE语句中在什么情况下可以不要WHEN OTHERS语句？在什么情况下一定要WHEN OTHERS语句？

5. FOR－LOOP语句应用于什么场合？循环变量怎样取值？是否需要事先在程序中定义？

6. 进程语句和并发代入语句之间有什么关系？进程之间的通信是通过什么方法来实现的？

7. 试用条件代入语句和选择信号代入语句来描述四—十六译码器。

8. 试用'EVENT属性描述一种用时钟clk上升触发的D触发器及一种用时钟clk下降沿触发的触发器。

9. 用GENERATE语句构造一个串行的十六进制计数器(使用什么触发器读者自定)。

10. 在VHDL语言中TEXTIO的作用是什么？其主要内容有哪一些？

习题与思考题六

1. 各种多值系统是针对什么样的实际情况而提出来的？

2. 在MOS电路中是否“0”值的强度都比“1”值的强。

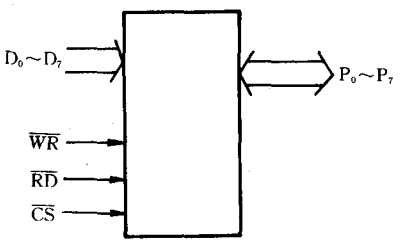
习题与思考题七

1. 试设计一个两位二进制的加法器。

2. 试设计一个两位的BCD计数器。

3. 同步计数器和异步计数器在设计时有哪些区别，试用一个六进制计数器和一个十进制计数器构成一个六十进制同步计数器。

4. 某一输入输出接口电路的引脚如题图7－1所示。



题图 7-1　某一输入输出接口引脚图

当$\overline{WR}$='0'，CS='0'有效时将 $D_0 \sim D_7$ 的 8 位数据写入输出锁存器锁存，$P_0 \sim P_7$ 输出 $D_0 \sim D_7$ 的数据。当$\overline{RD}$='0'，$\overline{CS}$='0'时，$P_0 \sim P_7$ 的 8 位数据经缓冲器从 $D_0 \sim D_7$ 输入，输入数据不被锁存。试用 VHDL 语言设计该接口电路。

5. 例 7-46 是 SRAM 的行为描述实例，如果要转换成 RTL 描述，程序应作哪些改变？

习题与思考题八

1. 仿真输入信息产生有哪几种方法？当前许多 EDA 工具所携带的波形编辑器也能产生仿真的输入信息，那么它应属于哪一类？

2. 对照习题七的第 2 题，试编写出一个仿真程序(参数自定)。

3. 为什么要引入仿真 Δ？对照例 8-3 写出每个仿真 Δ 时刻的 c，d，q 值。

4. 在仿真计数器时，时钟 clk 的周期应怎样选择？是否可以比实际的周期小？周期大小受到哪些条件限制？

5. 行为仿真、RTL 仿真和门级仿真这 3 个不同层次仿真的目的和内容有哪些区别？

6. 什么是逻辑综合？逻辑综合主要步骤有哪一些？

习题与思考题九

1. 本章例中包集合 p_stop_watch 包含哪些函数？如果取消包集合，这些函数功能应用什么样的模块来替代？

2. 什么是同步先行进位计数器？为什么本章例中要采用同步先行进位计数器？这样做对计数器设计带来哪些困难？

3. 本章例中的去抖动电路是怎样实现的？它的引入对计时精度会产生什么影响？

习题与思考题十

1. 在 8255 芯片设计中内部总线为什么要分成输入(internal_bus_in)和输出(internal_bus_out)两种，采用一条总线行否？为什么？

2. 如果 A 口的输入也要进行锁存，试问在“0”型方式下能否实现？为什么？

3. 为什么本章例中分成读进程和写进程？如果按端口划分进程来编写程序行不行？为什么？

4. 为什么在写进程中要设置一个 ctrregF 变量？把它设置成信号量行不行？

5. 串行数据的接收和发送是怎样进行控制的？接收寄存器和发送寄存器的“空”和“满”是怎样来实现同步的？

6. 如果 SCI 芯片在接收时要进行奇偶校验，试问该功能应怎样加上去？

7. KBC 芯片是怎样实现键盘扫描的？消除键抖动的措施加在什么地方？

8. KBC 的查表变换是怎样实现的？如要扩展键盘，这种变换规则仍适用否？

附录 A

VHDL 语言文法一览表

· **ALIAS(替换名)说明**

功能：定义已有目标的替换名

逻辑综合：不能

文法：ALIAS 替换名：子类型表示符　IS　目标名；

使用场所：ARCHITECTURE 说明部分，ENTITY 说明部分，PROCESS 说明部分，PACKAGE 说明，PACKAGE BODY 语句，SUBPROG RAM 说明部分

ALIAS 说明与子类型定义不同，只更改名称。

ALIAS bit1：STD_LOGIC IS L_VECTOR(1)；

ALIAS aaa：STD_LOGIC_VECTOR(0 TO 4) IS L_VECTOR(5 DOWNTO 1)；

· **ARCHITECTURE 说明**

功能：定义构造体(2.1 节中详述)

逻辑综合：可能

文法：ARCHITECTURE　构造体名　OF　实体名 IS

{说明语句}——说明部分

BEGIN

{并行处理语句}——本体

END{构造体名}；

说明语句::=SUBPROGRAM 说明|SUBPROGRAM 本体|TYPE 说明|SUBTYPE 说明|CONSTANT 说明|SIGNAL 说明|FILE 说明|ALIAS 说明|COMPONENT 说明|ATTRIBUTE 说明|ATTRIBUTE 定义|USE 语句|CONFIGURATION 定义|DISCONNECTION 定义

并行处理语句::=BLOCK 语句|PROCESS 语句|PROCEDURE 调用语句|ASSERT语句|代入语句|GENERATE 语句|COMPONENT_INSTANCE 语句

· **ASSERT 语句**

功能：信息输出(详见 8.1 节)

逻辑综合：忽略

文法：ASSERT　条件 [REPORT 输出信息] [警告级别];

级别::=NOTE|WARNING|ERROR|FAILURE

使用场所：ENTITY 本体，ARCHITECTURE 本体，PROCESS 本体，SUBPROGRAM 本体，BLOCK 本体，IF 语句，CASE 语句，LOOP 语句

· **ATTRIBUTE 说明**

功能：属性说明(详见 5.3 节)

逻辑综合：不能(只一部分属性可使用)

文法：ATTRIBUTE　属性名：子类型说明

使用场所：ARCHITECTURE 说明部分，ENTITY 说明部分，PROCESS 说明部分，PACKAGE 说明部分，SUBPROGRAM 说明部分

· **ATTRIBUTE 定义**

功能：详见 5.3 节

逻辑综合：不能(只一部分可使用)

文法：ATTRIBUTE　属性名　OF　目标名：目标级　IS　表示式;

级::=ENTITY|ARCHITECTURE|CONFIGURATION|LABEL|PROCEDURE|FUNCTION|P ACKAGE|TYPE|SUBTYPE|CONSTANT|VARIABLE|COMPONENT

使用场所：ARCHITECTURE 说明部分，ENTITY 说明部分，PROCESS 说明部分，PACKAGE 说明，SUBPROGRAM 说明部分，BLOCK 说明部分

```
ATTRIBUTE capacitance: cap;
ATTRIBUTE capacitance OF clk, reset: SIGNAL IS 20 pF;
```

· **ATTRIBUTE 名称**

功能：详见 5.3 节

逻辑综合：不能(只有 EVENT 和 STABLE 可以用)

文法：对象′ 属性名[(固定表示式)]

使用场所：信号代入语句，并行处理信号代入语句，变量代入语句，类型定义，接口清单，表示式，固定表示式

所谓固定表示式是可由文法判断的信号名，值，数值等。

```
r<=resistance'VAL(int_v);
```

· **BLOCK_CONFIGURATION**

功能：详见 2.2 节

逻辑综合：不能(只能选择 ARCHITECTURE)

文法：FOR　块定义
　　{USE 语句}
　　{BLOCK_ CONFIGURATION 语句 | COMPONENT_ CONFIGURATION 语句}
　　END FOR；
　　块定义::＝ARCHITECTURE 名|BLOCK 标号名|GENERATE 标号名
使用场所：CONFIGURATION 说明，BLOCK_CONFIGURATION 语句，COMPONENT_CONFIGURATION 语句

- **BLOCK 名称**

功能：ARCHITECTURE 的内部块(可嵌套)
逻辑综合：可能(卫式块不行)
文法：标号名：BLOCK [(卫式)]
　　块头
　　{说明语句}——说明部分
　　BEGIN
　　{并行处理语句}——本体
　　END BLOCK [标号名]；
　　块头::＝[GENERIC 语句 [GENERIC_ MAP 语句；]] [PORT 语句 [PORT_MAP 语句]]
　　说明语句：与构造体说明语句相同
　　并行处理语句：与构造体本体相同
使用场所：ARCHITECTURE 本体，BLOCK 本体
BLOCK 语句定义构造体内的子模块。与进程(PROCESS)语句不同的是，在内部各语句为并行处理。

- **CASE 语句**

功能：详见 5.1 节
逻辑综合：可能
文法：CASE　式　IS
　　条件式{条件式}
　　END CASE；
　　条件式::＝WHEN 选择项{|选择项}＝>{顺序处理语句}
　　选择项::＝式|不连续范围|名称|OTHERS
注意，WHEN OTHERS 只能用在最后一项上。
使用场所：PROCESS 本体，SUBPROGRAM 本体，IF 语句，CASE 语句，LOOP 语句

· **COMPONENT_CONFIGURATION 语句**

功能：详见 2.3 节

逻辑综合：不能

文法：FOR 样品清单：元件名

{USE 语句}

{BLOCK_CONFIGURATION 语句 | COMPONENT_CONFIGURATION 语句}

END FOR；

样品清单::=标号名{，标号名}|OTHERS|ALL

使用场所：CONFIGURATION 说明，BLOCK_CONFIGURATION 语句，COMPONENT_CONFIGURATION 语句

· **COMPONENT 说明**

功能：详见 4.3 节

逻辑综合：可能

文法：COMPONENT　元件名

[类属语句]

[端口语句]

END COMPONENT；

使用场所：ARCHITECTURE 说明部分，PACKAGE 说明部分

· **COMPONENT_INSTANCE 语句**

功能：详见 4.3 节

逻辑综合：可能

文法：标号名：元件名 [GENERIC_MAP 语句] [PORT_MAP 语句]；

使用场所：ARCHITECTURE 本体，BLOCK 本体，GENERATE 语句

· **CONFIGURATION 说明**

功能：详见 2.3 节

逻辑综合：不能(只可用于 ARCHITECTURE 选择)

文法：CONFIGURATION　配置名　OF　实体名　IS

{USE 语句|ATTRIBUTE 定义}

BLOCK_CONFIGURATION 语句

END FOR；

· **CONFIGURATION 定义**

功能：详见 2.3 节

逻辑综合：忽略

文法：FOR 样品清单：元件名　USE　对应对象；

使用场所：ARCHITECTURE 说明部分，BLOCK 说明部分

- **CONSTANT 说明**

功能：详见 3.1 节

逻辑综合：可能

文法：CONSTANT　常数名{，常数名}：子类型符 [：＝初始值]；

使用场所：ARCHITECTURE 说明部分，ENTITY 说明部分，PROCESS 说明部分，PACKAGE 说明，PACKAGE 本体，SUBPROGRAM 说明部分，BLOCK 说明部分

- **DISCONNECTION 定义**

功能：卫式驱动器默认关断时间定义

逻辑综合：不能

文法：DISCONNECT　卫式信号定义　AFTER　关断时间；

使用场所：ARCHITECTURE 说明部分，ENTITY 说明部分，PACKAGE 说明，BLOCK 说明部分

- **ENTITY 说明**

功能：详见 2.1 节

逻辑综合：可能

文法：ENTITY　实体名　IS

[类属语句；]

[PORT 语句；]

BEGIN

{断言语句|被调用过程|被调用进程语句}

END[实体名]；

说明语句::＝SUBPROGRAM 说明|SUBPROGRAM 本体|TYPE 说明|SUBTYPE 说明|CONSTANT 说明|SIGNAL 说明|FILE 说明|ALIAS 说明|ATTRIBUTE 说明|ATTRIBUTE 定义|USE 语句|CONFIGURATION 定义语句|DISCONNECTION 定义

- **FILE 说明**

功能：详见 5.3 节

逻辑综合：不能

文法：FILE 文件变量：子类型符　IS　方向　“文件名”；

使用场所：ARCHITECTURE 说明部分，ENTITY 说明部分，PROCESS 说明部分，PACKAGE 说明，PACKAGE 本体，SUBPROGRAM 说明部分，BLOCK 说明部分

- **GENERATE 语句**

  功能：详见 5.3 节

  逻辑综合：可能

  文法：标号名：FOR　产生变量　IN　不连续范围　GENERATE
  　　{并行处理语句}
  　END GENERATE [标号名]；
  　标号名：IF　条件　GENERATE
  　　{并行处理语句}
  　　END　GENERATE [标号名]；

  使用场所：ARCHITECTURE 本体，BLOCK 本体，GENERATE 语句

- **GENERIC 语句**

  功能：详见 4.1 节

  逻辑综合：只有整数型数据可以综合

  文法：GENERIC(端口名{，端口名}：[IN]
  　　子类型符[：＝ 初始值]
  　　{；端口名{，端口名}：[IN] 子类型符
  　　[：＝初始值]})

  使用场所：ENTITY 说明部分，BLOCK 说明部分

- **GENERIC_MAP 语句**

  功能：详见 4.1 节

  逻辑综合：只有整数型数据可以综合

  文法：GENERIC_MAP([形式＝＞]实体{,[形式＝＞]实体})
  　形式::＝端口名|类型变换函数名(端口名)
  　实体::＝式|SIGNAL 名|OPEN|类型变换函数名(实体)

  使用场所：COMPONENT_INSTANCE，BLOCK 说明部分，CONFIGURATION
  　　连接符

- **IF 语句**

  功能：详见 5.1 节

  逻辑综合：可能

  文法：IF　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；

  使用场所：PROCESS 语句，SUBPROGRAM 本体，IF 语句，CASE 语句，LOOP
  　　语句

· **LIBRARY 说明**

功能：详见 2.3 节

逻辑综合：可能

文法：LIBRARY 库名{，库名}；

· **LOOP 语句**

功能：详见 5.1 节

逻辑综合：可能

文法：[标号：] [循环次数限定] LOOP {顺序处理语句}

END LOOP [标号]；

循环次数限定::=FOR 循环变量 IN 不连续范围|WHILE 条件

在 LOOP 语句内部 NEXT [标号][WHEN 条件]；

：跳出本次

EXIT [标号] [WHEN 条件]；

：跳出环外

使用场所：PROCESS 语句，SUBPROGRAM 本体，IF 语句，CASE 语句，LOOP 语句

· **PACKAGE 说明**

功能：详见 2.3 节

逻辑综合：可能

文法：PACKAGE 包集合名 IS

{说明语句}

END [包集合名]；

说明语句::=SUBPROGRAM 说明|TYPE 说明|SUBTYPE 说明|CONSTANT 说明|SIGNAL 说明|FILE 说明|ALIAS 说明|COMPONENT 说明|ATTRIBUTE 说明|ATTRIBUTE 定义|USE 语句|DISCONNECTION 定义

· **PACKAGE_BODY 语句**

功能：详见 2.3 节

逻辑综合：可能

文法：PACKAGE BODY 包集合名 IS

{说明语句}

END [包集合名]；

说明语句::=SUBPROGRAM 本体|TYPE 说明|SUBTYPE 说明|CONSTANT 说明|FILE 说明|ALIAS 说明|USE 语句

· **PORT 语句**

功能：详见 2.1 节

逻辑综合：可能

文法：PORT(端口名{，端口名}：[方向] 子类型符 [BUS] [：=初始值]
{；端口名{，端口名}：[方向] 子类型符 [BUS] [：=初始值]})

方向::=IN|OUT|INOUT|BUFFER|LINKAGE

使用场所：ENTITY 说明部分，BLOCK 说明部分

· **PORT_MAP 语句**

功能：详见 4.3 节

逻辑综合：可能

文法：PORT_MAP([形式=>]实体{，形式=>]实体})

形式::=端口名|类型变换函数名(端口名)

实体::=式|SIGNAL 名|OPEN|类型变换函数名(实体)

使用场所：COMPONENT_INSTANCE 语句，BLOCK 说明部分，CONFIGURATION
连接符

· **PROCESS 语句**

功能：详见 2.2 节

逻辑综合：可能

文法：标号名：PROCESS [(敏感量清单)]
{说明语句} ——说明部分
BEGIN
{顺序处理语句}——本体
END PROCESS [标号名]；

说明语句::=SUBPROGRAM 说明|SUBPROGRAM 本体|TYPE 说明|SUBTYPE 说明|CONSTANT 说明|VARIABLE 说明|FILE 说明|ALIAS 说明|ATTRIBLE 说明|ATTRIBLE 定义|USE 语句

顺序处理语句::=WAIT 语句|PROCEDURE 调用|ASSERT 语句|信号代入语句|变量代入语句|IF 语句|CASE 语句|LOOP 语句|NULL 语句

使用场所：ARCHITECTURE 本体，BLOCK 本体，GENERATE 语句

· **PROCEDURE 调用**

功能：详见 2.2 节

逻辑综合：可能

文法：过程名[(接口清单)]

并行处理过程调用时可加标号

使用场所：ARCHITECTURE 本体，BLOCK 本体，PROCESS 本体，SUBPROGRAM 本体

- **SIGNAL 说明**

功能：详见 3.1 节

逻辑综合：可能

文法：SIGNAL 信号名{，信号名}：子类型符 [REGISTER|BUS] [：=初始值]；

若指定 REGISTER，BUS 则为卫式信号(可关断信号)。

REGISTER 保持关断时的信号值，而 BUS 则不保持。

卫式信号代入"NULL"，则由卫式关断信号。

使用场所：ARCHITECTURE 说明部分，ENTITY 说明部分，PACKAGE 说明部分

- **SUBPROGRAM 说明**

功能：详见 2.2 节

逻辑综合：可能

文法：PROCEDURE 过程名 [(输入，输出参数)]；

|FUNCTION 函数名 [(输入，输出参数)] RETURN 数据类型名；

输入，输出参数::=[SIGNAL|VARIABLE|CONSTANT 端口名{，端口名}：[方向]子类型符 [BUS] [：=初始值]；

使用场所：ARCHITECTURE 说明部分，ENTITY 说明部分，PROCESS 说明部分

- **SUBPROGRAM 本体**

功能：详见 2.2 节

逻辑综合：可能

文法：子程序定义 IS

{说明语句}

BEGIN

{顺序处理语句}

END {子程序名}；

使用场所：ARCHITECTURE 说明部分，ENTITY 说明部分，PROCESS 说明部分，PACKAGE 说明部分，PACKAGE 本体，SUBPROGRAM 说明部分，BLOCK 说明部分

- **TYPE 说明**

功能：详见 3.2 节

逻辑综合：一部分不可能

文法：不完整类型说明|完整类型说明

使用场所：ARCHITECTURE 说明部分，ENTITY 说明部分，PROCESS 说明部分，PACKAGE 说明部分，PACKAGE 本体，SUBPROGRAM 说明部分

不完整类型说明::＝TYPE 类型名{，类型名}；

→利用其它类型的假类型不能进行逻辑综合。

完整类型说明::＝TYPE 类型名{，类型名}　类型定义；

类型定义::＝标量类型定义|复合类型定义|存取类型定义|文件类型定义

标量类型定义::＝枚举类型定义|INTEGER 类型定义|FLOATING 类型定义|物理量类型定义

枚举类型定义::＝(元素{，元素})

→可进行逻辑综合

INTEGER 类型定义，FLOATING 类型定义::＝[简单式[TO|DOWNTO]简单式|属性名]；

→可进行逻辑综合

物理量类型定义::＝[RANGE[简单式[TO|DOWNTO] 简单式|属性名]]
UNITS 基本单位
{单位}
END UNITS

→不能进行逻辑综合

复合类型定义::＝数组类型|记录类型

数组类型::＝ARRAY(范围|RANGE<>{，范围|RANGE<>}) OF　子类型符

→只有一维的可以进行逻辑综合

记录类型::＝RECODE 元素{，元素} END RECODE

→可以进行逻辑综合(但是记录类型不能作元素)

存取定义::＝ACCESS 子类型符

→指向目标的类型(类似于 C 语言指示器)，不能进行逻辑综合

文件定义::＝FILE　OF　子类型符

→指定文件能读的数据类型，不能进行逻辑综合

- **USE 语句**

功能：使可视的说明变为直接可视

逻辑综合：可能

文法：USE{选择名{，选择名}；

选择名::＝实体{.实体}

使用场所：Design_unit，CONFIGURATION 说明，ARCHITECTURE 说明部分，BLOCK 说明部分，BLOCK_CONFIGURATION 语句，COMPONENT_CONFIGURATION 语句

· **VARIABLE 说明**

功能：详见 3.1 节

逻辑综合：可能

文法：VARIABLE　变量名{，变量名}：子类型符[：＝初始值]；

使用场所：PROCESS 说明部分，SUBPROGRAM 本体

· **WAIT 语句**

功能：详见 5.1 节

逻辑综合：只有 UNTIL 可能，其它忽略

文法：WAIT [ON 信号名{，信号名}]

[UNTIL 条件] [FOR 时间表达式]；

使用场所：PROCESS 语句，PROCEDURE 本体，IF 语句，CASE 语句，LOOP 语句

· **并行处理信号代入语句**

功能：详见 5.2 节

逻辑综合：可能(卫式不行，TRANSPORT 忽略)

文法：[标号名：]　条件信号代入语句|[标号名：]　选择信号代入语句

条件信号代入语句::＝目标<＝[GUARDED] [TANSPORT]条件波形

条件波形::＝{波形　WHEN 条件 ELSE}波形

选择信号代入语句::＝WITH 式 SELECT

目标⇐[GUARDED] [TRANSPORT]选择波形；

选择波形::＝{波形 WHEN 选择项，}波形 WHEN 选择项

使用场所：ARCHITECTURE 本体，BLOCK 本体，GENERATE 语句

· **信号代入语句**

功能：详见 5.1 节

逻辑综合：可能(忽略 TRANSPORT)

文法：目标<＝[TRANSPORT]波形；

使用场所：PROCESS 语句，SUBPROGRAM 本体，IF 语句，CASE 语句，LOOP 语句

· **波形**

功能：详见 3.1 节

逻辑综合：可能(忽略 AFTER，NULL)

文法：波形::＝式 [AFTER 时间表达式　|NULL [AFTER 时间表达式]{，式 [AFTER 时间表达式]|NULL[AFTER 时间表达式]}

如果代入 NULL，将关断卫式信号。

附录 B

属性说明

· 数组(信号，变量，常数)属性

N 表示二维数组行序号

A'LEFT [(N)]	左限值
A'RIGHT [(N)]	右限值
A'HIGH [(N)]	上限值
A'LOW [(N)]	下限值
A'RANGE [(N)]	范围
A'REVERSE_RANGE [(N)]	逆向范围
A'LENGTH [(N)]	范围个数

【例】

```
SIGNAL A: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL B: STD_LOGIC_VECTOR(0 TO 8);
TYPE C IS ARRAY (0 TO 5, 0 TO 8) OF STD_LOGIC;
A'LEFT→7          B'LEFT→0
A'RIGHT→0         B'RIGHT→8
                  C'RIGHT(2)→8
A'HIGH→7          B'HIGH→8
                  B'HIGH(1)→5
A'RANGE→7 DOWNTO 0
A'REVERSE_RANGE→0 TO 7
A'LENGTH→8        B'LENGTH→9
```

· 数据类型属性

T'BASE	T 的基本类型，只和其它属性并用。例：T'BASE'LEFT
T'LEFT	左限值
T'RIGHT	右限值
T'HIGH	上限值
T'LOW	下限值
T'POS(x)	参数(x)的位序号

T'VAL(x)　　x 的位置值
T'SUCC(x)　　比 x 的位序号大的一位置值
T'PRED(x)　　比 x 的位序号小的一位置值
T'LEFTOF(x)　在 x 左边位置的值
T'RIGHTOF(x) 在 x 右边位置的值

【例】

```
TYPE STD_LOGIC IS('U','X','0','1','Z','W','L','H','-');
SUBTYPE bar IS STD_LOGIC
bar'BASE'LFFT          →'U'
STD_LOGIC'LEFT         →'U'
STD_LOGIC'RIGHT        →'-'
STD_LOGIC'LOW          →'U'
STD_LOGIC'POS('Z')     →4
STD_LOGIC'SUCC('Z')    →'W'
STD_LOGIC'PRED('Z')    →'1'
```

· **信号属性**

S'DELAYED[(T)]　　延时 T 时间的值
S'EVENT　　表示 S 事件是否发生的值(发生=TRUE)
S'LAST_EVENT　　表示从事件发生后所经过的时间
S'LAST_VALUE　　表示最后事件发生前的值
S'STABLE[(T)]　　表示在 T 时间内是否发生事件的值(发生="FALSE")
S'QUIET[(T)]　　表示在 T 时间内信号值是否保持的值(保持="TRUE")
S'ACTIVE　　表示 S 是否有效的值(有效="TRUE")
S'LAST_EVENT　　从最后有效开始所经过的时间
S'TRANSACTION　　每次 S 有效重复返回 0 和 1 的值

附录 C

VHDL 标准包集合文件

● std_logic_1164(std_logic_1164: Draft Standard Version 4.2)

```
--  ------------------------------------------------------
--    Title       : std_logic_1164 multi_value logic system
--    Library     : This package shall be compiled into a library
--                : symbolically named IEEE.
--
--    Developers  : IEEE model standards group (par 1164)
--    Purpose     : This packages defines a standard for designers
--                : to use in describing the interconnection data types
--                : used in vhdl modeling.
--                :
--    Limitation  : The logic system defined in this package may
--                : be insufficient for modelign switched transistors,
--                : since such a requirement is out of the scope of this
--                : effort. Furthermore, mathematics, primitives,
--                : timing standards, etc. are considered orthogonal
--                : issues as it relates to this package and are therefore
--                : beyond the scope of this effort.
--                :
--    Note        : No declarations or definitions shall be included in,
--                : or excluded from this package. The "package declaration"
--                : defines the types, subtypes and declarations of
--                : std_logic_1164. The std_logic_1164 package body shall be
--                : considered the formal definition of the semantics of
--                : this package. Tool developers may choose to implement
--                : the package body in the most efficient manner available
--                : to them.
--                :
--  ------------------------------------------------------
--  modification history :
```

```
--  ------------------------------------------------
--  version | mod. date:|
--  v4.200 | 01/02/92 |
--  ------------------------------------------------

PACKAGE std_logic_1164 IS
    --  ------------------------------------------------
    --  logic state system (unresolved)
    --  ------------------------------------------------
    TYPE std_ulogic IS ('U', --  Uninitialized
                        'X', --  Forcing Unknown
                        '0', --  Forcing 0
                        '1', --  Forcing 1
                        'Z', --  High Impedance
                        'W', --  Weak     Unknown
                        'L', --  Weak     0
                        'H', --  Weak     1
                        '-', --  Don't care
                       );
    --  ------------------------------------------------
    --  unconstrained array of std_ulogic for use with the resolution function
    --  ------------------------------------------------
    TYPE std_ulogic_vector IS ARRAY (NATURAL RANGE <>) OF std_ulogic;
    --  ------------------------------------------------
    --  resolution function
    --  ------------------------------------------------
    FUNCTION resolved (s : std_ulogic_vector ) RETURN std_ulogic;
    --  ------------------------------------------------
    --  * * * industry standard logic type * * *
    --  ------------------------------------------------
    SUBTYPE std_logic IS resolved std_ulogic;
    --  ------------------------------------------------
    --  unconstrained array of std_logic for use in declaring signal arrays
    --  ------------------------------------------------
    TYPE std_logic_vector IS ARRAY (NATURAL RANGE <>) OF std_logic;
    --  ------------------------------------------------
    --  common subtypes
    --  ------------------------------------------------
    SUBTYPE X01   IS resolved std_ulogic RANGE 'X' TO '1'; -- ('X','0','1')
    SUBTYPE X01Z  IS resolved std_ulogic RANGE 'X' TO 'Z'; -- ('X','0','1','Z')
    SUBTYPE UX01  IS resolved std_ulogic RANGE 'U' TO '1'; -- ('U','X','0','1')
    SUBTYPE UX01Z IS resolved std_ulogic RANGE 'U' TO 'Z'; -- ('U','X','0','1','Z')
    --  ------------------------------------------------
```

```
-- overloaded logical operators
-- ---------------------------------------------
FUNCTION "and"  (l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "nand" (l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "or"   (l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "nor"  (l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "xor"  (l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "xnor" (l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "not"  (l : std_ulogic                 ) RETURN UX01;

-- ---------------------------------------------
-- vectorized overloaded logical operators
-- ---------------------------------------------
FUNCTION "and"  (l, r : std_logic_vector)  RETURN std_logic_vector;
FUNCTION "and"  (l, r : std_ulogic_vector) RETURN std_ulogic_vector;
FUNCTION "nand" (l, r : std_logic_vector)  RETURN std_logic_vector;
FUNCTION "nand" (l, r : std_ulogic_vector) RETURN std_ulogic_vector;
FUNCTION "or"   (l, r : std_logic_vector)  RETURN std_logic_vector;
FUNCTION "or"   (l, r : std_ulogic_vector) RETURN std_ulogic_vector;
FUNCTION "nor"  (l, r : std_logic_vector)  RETURN std_logic_vector;
FUNCTION "nor"  (l, r : std_ulogic_vector) RETURN std_ulogic_vector;
FUNCTION "xor"  (l, r : std_logic_vector)  RETURN std_logic_vector;
FUNCTION "xor"  (l, r : std_ulogic_vector) RETURN std_ulogic_vector;

-- ---------------------------------------------
-- Note: The declaration and implementation of the "xnor"function is
-- specifically commented until at which time the VHDL language has been
-- officially adopted as containing such a function. At such a point,
-- the following comments may be removed along with this notice without
-- further "official" ballotting of this std_logic_1164 package. It is
-- the intent of this effort to provide such a function once it becomes
-- available in the VHDL standard.
-- ---------------------------------------------
-- function "xnor" (l, r: std_logic_vector)  return std_logic_vector;
-- function "xnor" (l, r: std_ulogic_vector) return std_ulogic_vector;
FUNCTION "not" (l : std_logic_vector) RETURN std_logic_vector;
FUNCTION "not" (l : std_ulogic_vector)RETURN std_ulogic_vector;

-- ---------------------------------------------
-- conversion functions
-- ---------------------------------------------
FUNCTION To_bit       (s : std_ulogic; xmap : BIT := '0') RETURN BIT;
FUNCTION To_bitvector (s : std_logic_vector; xmp : BIT := '0') RETURN BIT_VECTOR;
FUNCTION To_bitvector(s : std_ulogic_vector; xmap: BIT := '0') RETURN BIT_VECTOR;
FUNCTION To_StdULogic (b : BIT ) RETURN std_ulogic;
```

```
    FUNCTION To_StdLogicVector  (b : BIT_VECTOR   ) RETURN std_logic_vector;
    FUNCTION To_StdLogicVector  (s : std_ulogic_vector) RETURN std_logic_vector;
    FUNCTION To_StdULogicVector (b : BIT_VECTOR   ) RETURN std_ulogic_vector;
    FUNCTION To_StdULogicVector (s : std_logic_vector) RETURN std_ulogic_vector;

    -- ------------------------------------------------------

    --  strength strippers and type convertors
    -- ------------------------------------------------------

    FUNCTION To_X01   (s : std_logic_vector    ) RETURN std_logic_vector;
    FUNCTION To_X01   (s : std_ulogic_vector   ) RETURN std_ulogic_vector;
    FUNCTION To_X01   (s : std_ulogic          ) RETURN X01;
    FUNCTION To_X01   (b : BIT_VECTOR          ) RETURN std_logic_vector;
    FUNCTION To_X01   (b : BIT_VECTOR          ) RETURN std_ulogic_vector;
    FUNCTION To_X01   (b : BIT                 ) RETURN X01;
    FUNCTION To_X01Z  (s : std_logic_vector    ) RETURN std_logic_vector;
    FUNCTION To_X01Z  (s : std_ulogic_vector   ) RETURN std_ulogic_vector;
    FUNCTION To_X01Z  (s : std_ulogic          ) RETURN X01Z;
    FUNCTION To_X01Z  (b : BIT_VECTOR          ) RETURN std_logic_vector;
    FUNCTION To_X01Z  (b : BIT_VECTOR          ) RETURN std_ulogic_vector;
    FUNCTION To_X01Z  (b : BIT                 ) RETURN X01Z;
    FUNCTION To_UX01  (s : std_logic_vector    ) RETURN std_logic_vector;
    FUNCTION To_UX01  (s : std_ulogic_vector   ) RETURN std_ulogic_vector;
    FUNCTION To_UX01  (s : std_ulogic          ) RETURN UX01;
    FUNCTION To_UX01  (b : BIT_VECTOR          ) RETURN std_logic_vector;
    FUNCTION To_UX01  (b : BIT_VECTOR          ) RETURN std_ulogic_vector;
    FUNCTION To_UX01  (b : BIT                 ) RETURN UX01;
    -- ------------------------------------------------------

    --  edge detection
    -- ------------------------------------------------------

    FUNCTION rising_edge  (SIGNAL s: std_ulogic) RETURN BOOLEAN;
    FUNCTION falling_edge (SIGNAL s: std_ulogic) RETURN BOOLEAN;
    -- ------------------------------------------------------

    --  object contains an unknown
    -- ------------------------------------------------------

    FUNCTION Is_X (s: std_ulogic_vector ) RETURN BOOLEAN;
    FUNCTION Is_X (s: std_logic_vector  ) RETURN BOOLEAN;
    FUNCTION Is_X (s: std_ulogic        ) RETURN BOOLEAN;

END std_logic_1164;

PACKAGE BODY std_logic_1164 IS

    -- ------------------------------------------------------

    --  local types
    -- ------------------------------------------------------

    TYPE stdlogic_ld IS ARRAY (std_ulogic) OF std_ulogic;
```

```
TYPE stdlogic_table IS ARRAY (std_ulogic, std_ulogic) OF std_ulogic;
-----------------------------------------------
-- resolution function
-----------------------------------------------
CONSTANT resolution_table: stdlogic_table:=(
--  ---------------------------------------------
--  | U  X  0  1  Z  W  L  H  -       | |
--  ---------------------------------------------
    ( 'U','U','U','U','U','U','U','U','U'),   --|U|
    ( 'U','X','X','X','X','X','X','X','X'),   --|X|
    ( 'U','X','0','X','0','0','0','0','X'),   --|0|
    ( 'U','X','X','1','1','1','1','1','X'),   --|1|
    ( 'U','X','0','1','Z','W','L','H','X'),   --|Z|
    ( 'U','X','0','1','W','W','W','W','X'),   --|W|
    ( 'U','X','0','1','L','W','L','W','X'),   --|L|
    ( 'U','X','0','1','H','W','W','H','X'),   --|H|
    ( 'U','X','X','X','X','X','X','X','X')    --|-|
   );
FUNCTION resolved (s:std_ulogic_vector) RETURN std_ulogic IS
   VARIABLE result:std_ulogic:='Z' ; -- weakest state default
BEGIN
  -- the test for a single driver is essential otherwise the
  -- loop would return 'X' for a single driver of '-' and that
  -- would conflict with the value of a single driver unresolved
  -- signal.
  IF  (s'LENGTH=1) THEN  RETURN s(s'LOW);
  ELST
    FOR i IN s'RANGE LOOP
       result:=resolution_table(result, s(i));
    END LOOP;
  END IF;
  RETURN result;
END resolved;
-----------------------------------------------
-- tables for logical operations
-----------------------------------------------
-- truth table for "and" function
CONSTANT and_table : stdlogic_table:=(
--  ---------------------------------------------
--  | U  X  0  1  Z  W  L  H  -       | |
--  ---------------------------------------------
    ( 'U','U','U','U','U','U','U','U','U'),   --|U|
```

```
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X'),   -- |X|
        ( '0', '0', '0', '0', '0', '0', '0', '0', '0'),   -- |0|
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X'),   -- |1|
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X'),   -- |Z|
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X'),   -- |W|
        ( '0', '0', '0', '0', '0', '0', '0', '0', '0'),   -- |L|
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X'),   -- |H|
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X'),   -- |-|
);

-- truth table for "or" function
CONSTANT or_table: stdlogic_table := (
--    ---------------------------------------------
--    |  U   X   0   1   Z   W   L   H   -      | |
--    ---------------------------------------------
        ( 'U', 'U', 'U', '1', 'U', 'U', 'U', '1', 'U'), -- |U|
        ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X'), -- |X|
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X'), -- |0|
        ( '1', '1', '1', '1', '1', '1', '1', '1', '1'), -- |1|
        ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X'), -- |Z|
        ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X'), -- |W|
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X'), -- |L|
        ( '1', '1', '1', '1', '1', '1', '1', '1', '1'), -- |H|
        ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X'), -- |-|
);

-- truth table for "xor" function
CONSTANT xor_table: stdlogic_table := (
--    ---------------------------------------------
--    |  U   X   0   1   Z   W   L   H   -      | |
--    ---------------------------------------------
        ( 'U', 'U', 'U', '1', 'U', 'U', 'U', 'U', 'U'), -- |U|
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'), -- |X|
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X'), -- |0|
        ( 'U', 'X', '1', '0', 'X', 'X', '1', '0', 'X'), -- |1|
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'), -- |Z|
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'), -- |W|
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X'), -- |L|
        ( 'U', 'X', '1', '0', 'X', 'X', '1', '0', 'X'), -- |H|
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'), -- |-|
);

-- truth table for "not" function
CONSTANT not_table: stdlogic_ld :=
-----------------------------------------
```

```
--    |  U   X   0   1   Z   W   L   H   -   |
-----------------------------------------------
      (  'U', 'X', '1', '0', 'X', 'X', '1', '0', 'X');
-----------------------------------------------
-- overloaded logical operators(with optimizing hints)
-----------------------------------------------
FUNCTION "and" (l:std_ulogic; r:std_ulogic) RETURN UX01 IS
BEGIN
    RETURN (and_table(l, r));
END "and";
FUNCTION "nand"(l:std_ulogic; r:std_ulogic) RETURN UX01 IS
BEGIN
    RETURN (not_talbe (and_table(l, r)));
END "nand";
FUNCTION "or"(l:std_ulogic; r:std_ulogic) RETURN UX01 IS
BEGIN
    RETURN(or_table(l, r));
END "or";
FUNCTION "nor"(l:std_ulogic; r:std_ulogic) RETURN UX01 IS
BEGIN
    RETURN(not_table(or_table(l, r)));
END "nor";
FUNCTION "xor"(l:std_ulogic; r: std_ulogic) RETURN UX01 IS
BEGIN
    RETURN(xor_table(l, r));
END "xor";
--  function "xnor"(l: std_ulogic; r: std_ulogic) return ux01 is
--  begin
--    return not_table(xor_table(l, r));
--  end "xnor";
  FUNCTION "not" (l: std_ulogic) RETURN UX01 IS
  BEGIN
     RETURN(not_table(l));
  END "not";
-----------------------------------------------
--  and
-----------------------------------------------
FUNCTION "and"(l, r: std_logic_vector) RETURN std_Logic_vector IS
    ALIAS lv: std_logic_vector(1 TO l'LENGTH) IS l;
    ALIAS rv: std_logic_vector(1 TO r'LENGTH) IS r;
    VARIABLE result: std_logic_vector(1 TO l'LENGTH);
BEGIN
    IF(l'LENGTH /=r'LENGTH) THEN
```

```
        ASSERT FALSE
REPORT "arguments of overloaded 'and' operator are not of the same length"
        SEVERITY FAILURE;
    ELSE
        FOR i IN result'RANGE LOOP
            result(i):=and_table(lv(i), rv(i));
        END LOOP;
    END IF;
    RETURN result;
END "and";
-------------------------------------------------------
FUNCTION "and"(l, r : std_ulogic_vector) RETURN std_ulogic_vector IS
    ALIAS lv: std_ulogic_vector(1 TO l'LENGTH) IS l;
    ALIAS rv: std_ulogic_vector(1 TO r'LENGTH) IS r;
    VARIABLE result: std_ulogic_vector(1 TO l'LENGTH);
BEGIN
    IF (l'LENGTH /=r'LENGTH) THEN
        ASSERT FALSE
 REPORT"arguments of overloaded 'and' operator are not of the same length"
        SEVERITY FAILURE;
  ELSE
        FOR i IN result'RANGE LOOP
            result(i):=and_table(lv(i), rv(i));
        END LOOP;
  END IF;
  RETURN result;
END "and";
-------------------------------------------------------
--  nand
-------------------------------------------------------
FUNCTION "nand"(l, r: std_logic_vector) RETURN std_logic_vector IS
    ALIAS lv: std_logic_vector(1 TO l'LENGTH) IS l;
    ALIAS rv: std_logic_vector(1 TO r'LENGTH) IS r;
    VARIABLE result: std_logic_vector(1 TO 1'LENGTH);
BEGIN
    IF(1'LENGTH /=r'LENGTH) THEN
        ASSERT FALSE
REPORT "arguments of overloaded 'nand' operator are not of the same length"
        SEVERITY FAILURE;
    ELSE
        FOR i IN result' RANGE LOOP
            result(i) :=not_table(and_table(lv(i), rv(i)));
        END LOOP;
```

```
    END IF;
    RETURN result;
END "nand";
-------------------------------------------------
FUNCTION "nand"(l, r: std_ulogic_vector) RETURN std_ulogic_vector IS
    ALIAS lv: std_ulogic_vector(1 TO l'LENGTH) IS l;
    ALIAS rv: std_ulogic_vector(1 TO r'LENGTH) IS r;
    VARIABLE result: std_ulogic_vector(1 TO l'LENGTH);
BEGIN
    IF (l'LENGTH /=r'LENGTH) THEN
        ASSERT FALSE
 REPORT "arguments of overloaded 'nand' operator are not of the same length"
        SEVERITY FAILURE;
    ELSE
        FOR i IN result' RANGE LOOP
            result(i):=not_table(and_table(lv(i), rv(i)));
        END LOOP;
    END IF;
    RETURN result;
END "nand";
-------------------------------------
--  or
-------------------------------------
FUNCTION "or"(l, r: std_logic_vector) RETURN std_logic_vector IS
    ALIAS lv: std_logic_vector(1 TO l'LENGTH) IS l;
    ALIAS rv: std_logic_vector(1 TO r'LENGTH) IS r;
    VARIABLE result: std_logic_vector(1 TO 1'LENGTH);
BEGIN
    IF(l'LENGTH /=r'LENGTH) THEN
        ASSERT FALSE
REPORT "arguments of overloaded 'or' operator are not of the same length"
        SEVERITY FAILURE;
    ELSE
        FOR i IN result' RANGE LOOP
            result(i) :=or_table(lv(i), rv(i));
        END LOOP;
    END IF;
    RETURN result;
END "or";
-------------------------------------------------
FUNCTION "or"(l, r : std_ulogic_vector) RETURN std_ulogic_vector IS
    ALIAS lv: std_ulogic_vector(1 TO l'LENGTH) IS l;
    ALIAS rv: std_ulogic_vector(1 TO r'LENGTH) IS r;
```

```
    VARIABLE result: std_ulogic_vector(1 TO 1'LENGTH);
BEGIN
    IF (l'LENGTH /=r'LENGTH) THEN
        ASSERT FALSE
 REPORT "arguments of overloaded 'or' operator are not of the same length"
        SEVERITY FAILURE;
    ELSE
        FOR i IN result' RANGE LOOP
            result(i):=or_table(lv(i), rv(i));
        END LOOP;
    END IF;
    RETURN result;
END "or";
-------------------------------------------------------
--  nor
-------------------------------------------------------
FUNCTION "nor"(l, r: std_logic_vector) RETURN std_logic_vector IS
    ALIAS lv: std_logic_vector(1 TO l'LENGTH) IS l;
    ALIAS rv: std_logic_vector(1 TO r'LENGTH) IS r;
    VARIABLE result: std_logic_vector(1 TO 1'LENGTH);
BEGIN
    IF(l'LENGTH /=r'LENGTH) THEN
        ASSERT FALSE
REPORT "arguments of overloaded 'nor' operator are not of the same length"
        SEVERITY FAILURE;
    ELSE
        FOR i IN result'RANGE LOOP
            result(i) :=nor_table(or_table (lv(i), rv(i)));
        END LOOP;
    END IF;
    RETURN result;
END "nor";
-------------------------------------------------------
FUNCTION "nor"(l, r : std_ulogic_vector) RETURN std_ulogic_vector IS
    ALIAS lv: std_ulogic_vector(1 TO l'LENGTH) IS l;
    ALIAS rv: std_ulogic_vector(1 TO r'LENGTH) IS r;
    VARIABLE result: std_ulogic_vector(1 TO l'LENGTH);
BEGIN
    IF (l'LENGTH /=r'LENGTH) THEN
        ASSERT FALSE
 REPORT "arguments of overloaded 'nor' operator are not of the same length"
        SEVERITY FAILURE;
 ELSE
```

```
        FOR i IN result' RANGE LOOP
            result(i):=nor_table(or_table (lv(i), rv(i)));
        END LOOP;
    END IF;
    RETURN result;
END "nor";
-----------------------------------------
--   xor
-----------------------------------------
FUNCTION "xor"(l, r: std_logic_vector) RETURN std_logic_vector IS
    ALIAS lv: std_logic_vector(1 TO l'LENGTH) IS l;
    ALIAS rv: std_logic_vector(1 TO r'LENGTH) IS r;
    VARIABLE result: std_logic_vector(l TO l'LENGTH);
BEGIN
    IF(l'LENGTH /=r'LENGTH) THEN
        ASSERT FALSE
REPORT "arguments of overloaded 'xor' operator are not of the same length"
        SEVERITY FAILURE;
    ELSE
        FOR i IN result' RANGE LOOP
            result(i):=xor_table (lv(i), rv(i));
        END LOOP;
    END IF;
    RETURN result;
END "xor";
-----------------------------------------------
FUNCTION "xor"(l, r : std_ulogic_vector) RETURN std_ulogic_vector IS
    ALIAS lv: std_ulogic_vector(1 TO l'LENGTH) IS l;
    ALIAS rv: std_ulogic_vector(1 TO r'LENGTH) IS r;
    VARIABLE result: std_ulogic_vector(1 TO l'LENGTH);
BEGIN
    IF (l'LENGTH /=r'LENGTH) THEN
        ASSERT FALSE
 REPORT "arguments of overloaded 'xor' operator are not of the same length"
        SEVERITY FAILURE;
  ELSE
        FOR i IN result'RANGE LOOP
            result(i):=xor_table (lv(i), rv(i));
        END LOOP;
  END IF;
  RETURN result;
END "xor";
-----------------------------------------
```

```
--  xnor
-------------------------------------------
--  Note: The declaration and implementation of the "xnor" function is
--  specifically commented until at which time the VHDL language has been
--  officially adopted as containing such a function. At such a point,
--  the following comments may be removed along with this notice without
--  further "official" ballotting of this std_logic_1164 package. It is
--  the intent of this effort to provide such a function once it becomes
--  available in the VHDL standard.
--  ---------------------------------------------------
--  function "xnor"(l, r: std_logic_vector)return std_logic_vector is
--      alias lv: std_logic_vector(1 to l'length) is l;
--      alisa rv: std_logic_vector(1 to r'length) is r;
--      variable result: std_logic_vector(1 to l'length);
--  begin
--      if(l'length /=r'length) then
--        assert false
--  report "arguments of overloaded 'xnor' operator are not of the same length"
--        severity failure;
--      else
--        for i in result'range loop
--            result(i):=not_table(xor_table(lv(i), rv(i)));
--        end loop;
--      end if;
--    return result;
--  end "xnor";
--  function "xnor"(l, r: std_ulogic_vector) return std_ulogic_vector is
--      alias lv: std_ulogic_vector(1 to l'length) is l;
--      alias rv: std_ulogic_vector(1 to r'length) is r;
--      variable result: std_ulogic_vector(1 to l'length);
--  begin
--      if(l'length /=r'length) then
--        assert false
--  report "arguments of overloaded 'xnor' operator are not of the same length"
--        severity failure;
--      else
--        for i in result'range loop
--            result(i):=not_talbe(xor_table(lv(i), rv(i)));
--        end loop;
--    end if;
--    return result;
--  end "xnor";
    -----------------------------------------
```

```
--   not
-------------------------------------------
FUNCTION "not" (l: std_logic_vector) RETURN std_logic_vector IS
    ALIAS lv: std_logic_vector(1 TO l'LENGTH) IS l;
    VARIABLE result: std_logic_vector(1 TO l'LENGTH):=(OTHERS=>'X');
BEGIN
    FOR i IN result'RANGE LOOP
        result(i):=not_table(lv(i));
    END LOOP;
    RETURN result;
END;
-------------------------------------------------
FUNCTION "not" (l: std_ulogic_vector) RETURN std_ulogic_vector IS
    ALIAS lv: std_ulogic_vector(1 TO l'LENGTH) is l;
    VARIABLE result: std_ulogic_vector(1 TO l'LENGTH):=(OTHERS=>'X');
BEGIN
    FOR i IN result'RANGE LOOP
        result(i):=not_table(lv(i));
    END LOOP;
    RETURN result;
END;
-------------------------------------------------
--conversion tables
-------------------------------------------------
TYPE logic_x01_table IS ARRAY(std_ulogic' LOW TO std_ulogic'HIGH)OF X01;
TYPE logic_x01z_table IS ARRAY(std_ulogic' LOW TO std_ulogic'HIGH)OF X01Z;
TYPE logic_ux01_table IS ARRAY(std_ulogic' LOW TO std_ulogic'HIGH)OF UX01;
-------------------------------------------------
--table name : cvt_to_x01
--parameters :
--        in  : std_ulogic     --some logic value
--returns     : x01            --state value of logic value
--purpose     : to convert state-strength to state only
--example     : if(cvt_to_x01(input_signal)='1') then…
-------------------------------------------
CONSTANT cvt_to_x01 : logic_x01table:=(
                      'X',      ----'U'
                      'X',      ----'X'
                      '0',      ----'0'
                      '1',      ----'1'
                      'X',      ----'Z'
                      'X',      ----'W'
                      '0',      ----'L'
```

```
                    '1',        ----'H'
                    'X'         ----'-');
-------------------------------------------------
-- table name : cvt_to_x01z
-- parameters :
--       in   : std_ulogic     -- some logic value
-- returns    : x01z           -- state value of logic value
-- purpose    : to convert state - strength to state only
-- example    : if(cvt_to_x01z(input_signal)='1') then...
-------------------------------------------------
CONSTANT cvt_to_x01z : logic_x01z_table:=(
                    'X',    --'U'
                    'X',    --'X'
                    '0',    --'0'
                    '1',    --'1'
                    'Z',    --'Z'
                    'X',    --'W'
                    '0',    --'L'
                    '1',    --'H'
                    'X'     --'-');
-------------------------------------------------
-- table name : cvt_to_ux01
-- parameters :
--       in   : std_ulogic -- some logic value
-- returns    : ux01           -- state value of logic value
-- purpose    : to convert state - strength to state only
--
-- example    : if(cvt_to_ux01(input_signal)='1') then...
-------------------------------------------------
CONSTANT cvt_to_ux01 : logic_ux01_table:=(
                    'U',        --'U'
                    'X',        --'X'
                    '0',        --'0'
                    '1',        --'1'
                    'X',        --'Z'
                    'X',        --'W'
                    '0',        --'L'
                    '1',        --'H'
                    'X'         --'-');
-------------------------------------------------
--   conversion functions
-------------------------------------------------
FUNCTION To_bit(s : std_ulogic; xmap : BIT:='0' ) RETURN BIT IS
```

```
BEGIN
      CASE s IS
          WHEN '0' | 'L' =>RETURN('0');
          WHEN '1' | 'H' =>RETURN('1');
          WHEN OTHERS=>RETURN xmap;
      END CASE;
END;
-------------------------------------------------
FUNCTION To_bitvector(s:std_logic_vector; xmap:BIT:='0')RETURN BIT_VECTOR IS
    ALIAS sv : std_logic_vector(s'LENGTH-1 DOWNTO 0) IS s;
    VARIABLE result : BIT_VECTOR(s'LENGTH-1 DOWNTO 0);
BEGIN
    FOR i IN result'RANGE LOOP
        CASE sv(i) IS
            WHEN '0' | 'L'=>result(i) :='0';
            WHEN '1' | 'H' =>result(i) :='1';
            WHEN OTHERS=>result(i) := xmap;
        END CASE;
END LOOP;
     RETURN result;
END;
-------------------------------------------------
FUNCTION To_bitvector(s : std_ulogic_vector; xmap:BIT:='0')RETURN BIT_VECTOR IS
    ALIAS sv : std_ulogic_vector(s'LENGTH-1 DOWNTO 0) IS s;
    VARIABLE result : BIT_VECTOR(s'LENGTH-1 DOWNTO 0);
BEGIN
    FOR i IN result'RANGE LOOP
        CASE sv(i) IS
            WHEN '0' | 'L' =>result(i) :='0';
            WHEN '1' | 'H' =>result(i) :='1';
            WHEN OTHERS=>result(i) := xmap;
        END CASE;
    END LOOP;
        RETURN result;
END;
-------------------------------------------------
FUNCTION To_StdULogic(b : BIT) RETURN std_ulogic IS
BEGIN
      CASE b IS
          WHEN '0' =>RETURN'0';
          WHEN '1' =>RETURN'1';
      END CASE;
END;
```

```
-------------------------------------------------
FUNCTION To_StdLogicVector(b : BIT_VECTOR)RETURN std_logic_vector IS
    ALIAS bv : BIT_VECTOR(b'LENGTH -1 DOWNTO 0) IS b;
    VARIABLE result : std_logic_vector(b'LENGTH -1 DOWNTO 0);
BEGIN
    FOR i IN result'RANGE LOOP
        CASE bv(i) IS
            WHEN '0' =>result(i) :='0';
            WHEN '1' =>result(i) :='1';
        END CASE;
    END LOOP;
    RETURN result;
END;
-------------------------------------------------
FUNCTION To_StdULogicVector(s : std_ulogic_vector)RETURN std_logic_vector IS
    ALIAS sv : std_ulogic_vector(s'LENGTH -1 DOWNTO 0) IS s;
    VARIABLE result : std_logic_vector(s'LENGTH -1 DOWNTO 0);
BEGIN
    FOR i IN result'RANGE LOOP
        result(i):=sv(i);
    END LOOP;
    RETURN result;
END;
-------------------------------------------------
FUNCTION To_StdULogicVector(b : BIT_VECTOR)RETURN std_ulogic_vector IS
    ALIAS bv : BIT_VECTOR(b'LENGTH -1 DOWNTO 0) IS b;
    VARIABLE result : std_ulogic_vector(b'LENGTH -1 DOWNTO 0);
BEGIN
    FOR i IN result'RANGE LOOP
        CASE bv(i) IS
            WHEN '0' => result(i) :='0';
            WHEN '1' => result(i) :='1';
        END CASE;
    END LOOP;
    RETURN result;
END;
-------------------------------------------------
FUNCTION To_StdULogicVector(s : std_logic_vector)RETURN std_ulogic_vector IS
    ALIAS sv : std_logic_vector(s'LENGTH -1 DOWNTO 0) IS s;
    VARIABLE result : std_ulogic_vector(s'LENGTH -1 DOWNTO 0);
BEGIN
    FOR i IN result'RANGE LOOP
        result(i):=sv(i);
```

```
    END LOOP;
    RETURN result;
END;
-----------------------------------------------
-- strength strippers and type convertors
-----------------------------------------------
-- to_x01
-----------------------------------------------
FUNCTION To_X01(s : std_logic_vector)RETURN std_logic_vector IS
    ALIAS sv : std_logic_vector(1 TO s'LENGTH) IS s;
    VARIABLE result : std_logic_vector(1 TO s'LENGTH);
BEGIN
    FOR i IN result'RANGE LOOP
        result(i):=cvt_to_x01 (sv(i));
    END LOOP;
    RETURN result;
END;
-----------------------------------------------
FUNCTION To_X01(s : std_ulogic_vector)RETURN std_ulogic_vector IS
    ALIAS sv : std_ulogic_vector(1 TO s'LENGTH) IS s;
    VARIABLE result : std_ulogic_vector(1 TO s'LENGTH);
BEGIN
    FOR i IN result' RANGE LOOP
        result(i):=cvt_to_x01 (sv(i));
    END LOOP;
    RETURN result;
END;
-----------------------------------------------
FUNCTION To_X01(s : std_ulogic)RETURN X01 IS
BEGIN
    RETURN(cvt_to_x01(s));
END;
-----------------------------------------------
FUNCTION To_X01(b : BIT_VECTOR)RETURN std_logic_vector IS
    ALIAS bv : BIT_VECTOR(1 TO b'LENGTH) IS b;
    VARIABLE result : std_logic_vector(1 TO b'LENGTH);
BEGIN
    FOR i IN result' RANGE LOOP
        CASE bv(i) IS
            WHEN '0' =>result(i) :='0';
            WHEN '1' =>result(i) :='1';
        END CASE;
    END LOOP;
```

```
    RETURN result;
END;
-------------------------------------------------
FUNCTION To_X01(b : BIT_VECTOR)RETURN std_ulogic_vector IS
    ALIAS bv : BIT_VECTOR(1 TO b'LENGTH) IS b;
    VARIABLE result : std_ulogic_vector(1 TO b'LENGTH);
BEGIN
    FOR i IN result 'RANGE LOOP
        CASE bv(i) IS
            WHEN '0' =>result(i) :='0';
            WHEN '1' =>result(i) :='1';
        END CASE;
    END LOOP;
    RETURN result;
END;
-------------------------------------------------
FUNCTION To_X01(b : BIT)RETURN X01 IS
BEGIN
        CASE b IS
                WHEN '0' =>RETURN('0');
                WHEN '1' =>RETURN('1');
        END CASE;
END;
-------------------------------------------------
--to_x01z
-------------------------------------------------
FUNCTION To_X01Z(s:std_logic_vector)RETURN std_logic_vector IS
    ALIAS sv:std_logic_vector(1 TO s'LENGTH) IS s;
    VARIABLE result : std_logic_vector(1 TO s'LENGTH);
BEGIN
    FOR i IN result'RANGE LOOP
        result(i):=cvt_to_x01z (sv(i));
    END LOOP;
    RETURN result;
END;
-------------------------------------------------
FUNCTION To_X01Z  (s : std_ulogic_vector)RETURN std_ulogic_vector IS
    ALIAS sv:std_ulogic_vector(1 TO s'LENGTH) IS s;
    VARIABLE result:std_ulogic_vector(1 TO s'LENGTH);
BEGIN
    FOR i IN result' RANGE LOOP
        result(i):=cvt_to_x01z (sv(i));
    END LOOP;
```

```
    RETURN result;
END;
-----------------------------------------------------
FUNCTION To_X01Z(s:std_ulogic)RETURN X01Z IS
BEGIN
    RETURN(cvt_to_x01z(s));
END;
-----------------------------------------------------
FUNCTION To_X01Z(b: BIT_VECTOR)RETURN std_logic_vector IS
    ALIAS bv: BIT_VECTOR(1 TO b'LENGTH) IS b;
    VARIABLE result:std_logic_vector(1 TO b'LENGTH);
BEGIN
    FOR i IN result' RANGE LOOP
        CASE bv(i) IS
            WHEN '0' =>result(i) :='0';
            WHEN '1' =>result(i) :='1';
        END CASE;
    END LOOP;
    RETURN result;
END;
-----------------------------------------------------
FUNCTION To_X01Z  (b: BIT_VECTOR)  RETURN std_ulogic_vector IS
    ALIAS bv: BIT_VECTOR(1 TO b'LENGTH) IS b;
    VARIABLE result : std_ulogic_vector(1 TO b'LENGTH);
BEGIN
    FOR i IN result' RANGE LOOP
        CASE bv(i) IS
            WHEN '0' => result(i) :='0';
            WHEN '1' => result(i) :='1';
        END CASE;
    END LOOP;
    RETURN result;
END;
-----------------------------------------------------
FUNCTION To_X01Z(b: BIT)RETURN X01Z IS
BEGIN
        CASE b IS
                WHEN '0' => RETURN('0');
                WHEN '1' => RETURN('1');
        END CASE;
END;
-----------------------------------------------------
--to_ux01
```

```
-------------------------------------------------------
FUNCTION To_UX01(s : std_logic_vector)RETURN std_logic_vector IS
    ALIAS sv : std_logic_vector(1 TO s'LENGTH) IS s;
    VARIABLE result : std_logic_vector(1 TO s'LENGTH);
BEGIN
    FOR i IN result' RANGE LOOP
        result(i):=cvt_to_ux01 (sv(i));
    END LOOP;
    RETURN result;
END;
-------------------------------------------------------
FUNCTION To_UX01  (s:std_ulogic_vector)  RETURN std_ulogic_vector IS
    ALIAS sv:std_ulogic_vector(1 TO s'LENGTH) IS s;
    VARIABLE result:std_ulogic_vector(1 TO s'LENGTH);
BEGIN
    FOR i IN result' RANGE LOOP
        result(i):=cvt_to_ux01 (sv(i));
    END LOOP;
    RETURN result;
END;
-------------------------------------------------------
FUNCTION To_UX01(s:std_ulogic)RETURN UX01 IS
BEGIN
    RETURN(cvt_to_ux01(s));
END;
-------------------------------------------------------
FUNCTION To_UX01(b:BIT_VECTOR)RETURN std_logic_vector IS
    ALIAS bv:BIT_VECTOR(1 TO b'LENGTH) IS b;
    VARIABLE result:std_logic_vector(1 TO b'LENGTH);
BEGIN
    FOR i IN result' RANGE LOOP
        CASE bv(i) IS
            WHEN '0' => result(i) :='0';
            WHEN '1' => result(i) :='1';
        END CASE;
    END LOOP;
    RETURN result;
END;
-------------------------------------------------------
FUNCTION To_UX01(b:BIT_VECTOR)RETURN std_ulogic_vector IS
    ALIAS bv:BIT_VECTOR(1 TO b'LENGTH) IS b;
    VARIABLE result : std_ulogic_vector(1 TO b'LENGTH);
BEGIN
```

```
    FOR i IN result' RANGE LOOP
        CASE bv(i) IS
            WHEN '0' => result(i) :='0';
            WHEN '1' => result(i) :='1';
        END CASE;
    END LOOP;
    RETURN result;
END;
-----------------------------------------------
FUNCTION To_UX01(b : BIT)RETURN UX01 IS
BEGIN
        CASE b IS
                WHEN '0' => RETURN('0');
                WHEN '1' => RETURN('1');
        END CASE;
END;
-----------------------------------------------
--edge detection
-----------------------------------------------
FUNCTION rising_edge(SIGNAL s:std_ulogic) RETURN BOOLEAN IS
BEGIN
    RETURN(s'EVENT AND (TO_X01(s)='1') AND
                        (TO_X01(s'LAST_VALUE)='0'));
END;
FUNCTION falling_edge(SIGNAL s:std_ulogic) RETURN BOOLEAN IS
BEGIN
    RETURN(s'EVENT AND (TO_X01(s)='0') AND
                        (TO_X01(s'LAST_VALUE)='1'));
END;
-----------------------------------------------
--object contains an unknown
-----------------------------------------------
FUNCTION Is_X(s:std_ulogic_vector) RETURN BOOLEAN IS
BEGIN
    FOR i IN s'RANGE LOOP
        CASE s(i) IS
            WHEN 'U'|'X'|'Z'|'W'|'-'|=>RETURN TRUE;
            WHEN OTHERS=>NULL;
        END CASE;
    END LOOP;
    RETURN FALSE;
END;
-----------------------------------------------
```

```
    FUNCTION ls_X(s : std_logic_vector) RETURN BOOLEAN IS
    BEGIN
        FOR i IN s'RANGE LOOP
            CASE s(i) IS
                WHEN 'U'|'X'|'Z'|'W'|'-'|=>RETURN TRUE;
                WHEN OTHERS=>NULL;
            END CASE;
        END LOOP;
        RETURN FALSE;
    END;
    ------------------------------------------------
    FUNCTION ls_X(s:std_ulogic) RETURN BOOLEAN IS
    BEGIN
            CASE s IS
                WHEN 'U'|'X'|'Z'|'W'|'-'|=>RETURN TRUE;
                WHEN OTHERS=>NULL;
            END CASE;
        RETURN FALSE;
    END;

    END std_logic_1164;
```

● std_logic_arith(Synopsys 公司提供)

```
------------------------------------------------------------
--Copyright (c) 1990, 1991, 1992 by Synopsys, Inc. All rights reserved. --
--This source file may be used and distributed without restriction      --
--provided that this copyright statement is not removed from the file   --
--and that any derivative work contains this copyright notice.          --
--                                                                      --
--    Package name: STD_LOGIC_ARITH                                     --
--                                                                      --
--    Purpose:                                                          --
--    A set of arithemtic, conversion, and comparison functions         --
--    for SIGNED, UNSIGNED, SMALL_INT, INTEGER,                         --
--    STD_ULOGIC, STD_LOGIC, and STD_LOGIC_VECTOR.                      --
------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
package std_logic_arith is
    type UNSIGNED is array(NATURAL range <>)of STD_LOGIC;
    type SIGNED is array(NATURAL range <>)of STD_LOGIC;
    subtype SMALL_INT is INTEGER range 0 to 1;
    function "+"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED;
```

```
function "+"(L: SIGNED; R: SIGNED) return SIGNED;
function "+"(L: UNSIGNED; R: SIGNED) return SIGNED;
function "+"(L: SIGNED; R: UNSIGNED) return SIGNED;
function "+"(L: UNSIGNED; R: INTEGER) return UNSIGNED;
function "+"(L: INTEGER; R: UNSIGNED) return UNSIGNED;
function "+"(L: SIGNED; R: INTEGER) return SIGNED;
function "+"(L: INTEGER; R: SIGNED) return SIGNED;
function "+"(L: UNSIGNED; R: STD_ULOGIC) return UNSIGNED;
function "+"(L: STD_ULOGIC; R: UNSIGNED) return UNSIGNED;
function "+"(L: SIGNED; R: STD_ULOGIC) return SIGNED;
function "+"(L: STD_ULOGIC; R: SIGNED) return SIGNED;
function "+"(L: UNSIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "+"(L: UNSIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: UNSIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
function "+"(L: INTEGER; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
function "+"(L: INTEGER; R: SIGNED) return STD_LOGIC_VECTOR;
function "+"(L: UNSIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
function "+"(L: STD_ULOGIC; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
function "+"(L: STD_ULOGIC; R:SIGNED) return STD_LOGIC_VECTOR:
function "-"(L: UNSIGNED; R: NSIGNED) return UNSIGNED;
function "-"(L: SIGNED; R: SIGNED) return SIGNED;
function "-"(L: UNSIGNED; R: SIGNED) return SIGNED;
function "-"(L: SIGNED; R: UNSIGNED) return SIGNED;
function "-"(L: UNSIGNED; R: INTEGER) return UNSIGNED;
function "-"(L: INTEGER; R: UNSIGNED) return UNSIGNED;
function "-"(L: SIGNED; R: INTEGER) return SIGNED;
function "-"(L: INTEGER; R: SIGNED) return SIGNED;
function "-"(L: UNSIGNED; R: STD_ULOGIC) return UNSIGNED;
function "-"(L: STD_ULOGIC; R: UNSIGNED) return UNSIGNED;
function "-"(L: SIGNED; R: STD_ULOGIC) return SIGNED;
function "-"(L: STD_ULOGIC; R: SIGNED) return SIGNED;
function "-"(L: UNSIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "-"(L: SIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "-"(L: UNSIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "-"(L: SIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "-"(L: UNSIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
function "-"(L: INTEGER; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "-"(L: SIGNED; R: INTEGER) return STD_LOGIC_VECTOR;
function "-"(L: INTEGER; R: SIGNED) return STD_LOGIC_VECTOR;
```

```
function "-"(L: UNSIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
function "-"(L: STD_ULOGIC; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "-"(L: SIGNED; R: STD_ULOGIC) return STD_LOGIC_VECTOR;
function "-"(L: STD_ULOGIC; R: SIGNED) return STD_LOGIC_VECTOR;
function "+"(L: UNSIGNED) return UNSIGNED;
function "+"(L: SIGNED) return SIGNED;
function "-"(L: SIGNED) return SIGNED;
function "ABS"(L: SIGNED) return SIGNED;
function "+"(L: UNSIGNED) return STD_LOGIC_VECTOR;
function "+"(L: SIGNED) return STD_LOGIC_VECTOR;
function "-"(L: SIGNED) return STD_LOGIC_VECTOR;
function "ABS"(L: SIGNED) return STD_LOGIC_VECTOR;
function "*"(L: UNSIGNED; R: SNSIGNED) return UNSIGNED;
function "*"(L: SIGNED; R: SIGNED) return SIGNED;
function "*"(L: SIGNED; R: UNSIGNED) return SIGNED;
function "*"(L: UNSIGNED; R: SIGNED) return SIGNED;
function "*"(L: UNSIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "*"(L: SIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "*"(L: SIGNED; R: UNSIGNED) return STD_LOGIC_VECTOR;
function "*"(L: UNSIGNED; R: SIGNED) return STD_LOGIC_VECTOR;
function "<"(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function "<"(L: SIGNED; R: SIGNED) return BOOLEAN;
function "<"(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function "<"(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function "<"(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function "<"(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function "<"(L: SIGNED; R: INTEGER) return BOOLEAN;
function "<"(L: INTEGER; R: SIGNED) return BOOLEAN;
function "<="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function "<="(L: SIGNED; R: SIGNED) return BOOLEAN;
function "<="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function "<="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function "<="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function "<="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function "<="(L: SIGNED; R: INTEGER) return BOOLEAN;
function "<="(L: INTEGER; R: SIGNED) return BOOLEAN;
function ">"(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function ">"(L: SIGNED; R: SIGNED) return BOOLEAN;
function ">"(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function ">"(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function ">"(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function ">"(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function ">"(L: SIGNED; R: INTEGER) return BOOLEAN;
```

```
function ">"(L: INTEGER; R: SIGNED) return BOOLEAN;
function ">="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function ">="(L: SIGNED; R: SIGNED) return BOOLEAN;
function ">="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function ">="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function ">="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function ">="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function ">="(L: SIGNED; R: INTEGER) return BOOLEAN;
function ">="(L: INTEGER; R: SIGNED) return BOOLEAN;
function "="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function "="(L: SIGNED; R: SIGNED) return BOOLEAN;
function "="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function "="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function "="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function "="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function "="(L: SIGNED; R: INTEGER) return BOOLEAN;
function "="(L: INTEGER; R: SIGNED) return BOOLEAN;
function "/="(L: UNSIGNED; R: UNSIGNED) return BOOLEAN;
function "/="(L: SIGNED; R: SIGNED) return BOOLEAN;
function "/="(L: UNSIGNED; R: SIGNED) return BOOLEAN;
function "/="(L: SIGNED; R: UNSIGNED) return BOOLEAN;
function "/="(L: UNSIGNED; R: INTEGER) return BOOLEAN;
function "/="(L: INTEGER; R: UNSIGNED) return BOOLEAN;
function "/="(L: SIGNED; R: INTEGER) return BOOLEAN;
function "/="(L: INTEGER; R: SIGNED) return BOOLEAN;
function SHL(ARG: UNSIGNED; COUNT: UNSIGNED) return UNSIGNED;
function SHL(ARG: SIGNED; COUNT: UNSIGNED) return SIGNED;
function SHL(ARG: UNSIGNED; COUNT: UNSIGNED) return UNSIGNED;
function SHL(ARG: SIGNED; COUNT: UNSIGNED) return SIGNED;

function CONV_INTEGER(ARG: INTEGER) return INTEGER;
function CONV_INTEGER(ARG: UNSIGNED) return INTEGER;
function CONV_INTEGER(ARG: SIGNED) return INTEGER;
function CONV_INTEGER(ARG: STD_ULOGIC) return SMALL_INT;
function CONV_UNSIGNED(ARG: INTEGER: SIZE: INTEGER) return UNSIGNED;
function CONV_UNSIGNED(ARG: UNSIGNED: SIZE: INTEGER) teturn UNSIGNED;
function CONV_UNSIGNED(ARG: SIGNED: SIZE: INTEGER) return UNSIGNED;
function CONV_UNSIGNED(ARG: STD_ULOGIC: SIZE: INTEGER) return UNSIGNED;
function CONV_SIGNED(ARG: INTEGER: SIZE: INTEGER) return SIGNED;
function CONV_SIGNED(ARG: UNSIGNED: SIZE: INTEGER) return SIGNED;
function CONV_SIGNED(ARG: SIGNED: SIZE: INTEGER) return SIGNED;
function CONV_SIGNED(ARG: STD_ULOGIC: SIZE: INTEGER) return SIGNED;
function CONV_STD_LOGIC_VECTOR(ARG: INTEGER: SIZE: INTEGER)
```

```
                                    return STD_LOGIC_VECTOR;
    function CONV_STD_LOGIC_VECTOR(ARG: UNSIGNED; SIZE: INTEGER)
                                    return STD_LOGIC_VECTOR;
    function CONV_STD_LOGIC_VECTOR(ARG: SIGNED; SIZE: INTEGER)
                                    return STD_LOGIC_VECTOR;
    function CONV_STD_LOGIC_VECTOR(ARG: STD_ULOGIC; SIZE: INTEGER)
                                    return STD_LOGIC_VECTOR;
    -- zero extend STD_LOGIC_VECTOR(ARG) to SIZE,
    -- SIZE <0 is same as SIZE=0
    -- returns STD_LOGIC_VECTOR(SIZE -1 downto 0)
    function EXT(ARG: STD_LOGIC_VECTOR; SIZE: INTEGER) return STD_LOGIC_VECTOR;
    -- sign extend STD_LOGIC_VECTOR(ARG) to SIZE,
    -- SIZE <0 is same as SIZE=0
    -- returns STD_LOGIC_VECTOR(SIZE -1 downto 0)
    function SXT(ARG: STD_LOGIC_VECTOR; SIZE: INTEGER) return STD_LOGIC_VECTOR;
end Std_logic_arith;
```

● std_logic_unsigned(Synopsys 公司提供)

```
------------------------------------------------------------
-- Copyright (c) 1990, 1991, 1992 by Synopsys, Inc.        --
                                    All rights reserved.    --
-- This source file may be used and distributed without restriction --
-- provided that this copyright statement is not removed from the file --
-- and that any derivative work contains this copyright notice. --
--                                                          --
--     Package name: STD_LOGIC_UNSIGNED                     --
--       Date: 09/11/92     KN                              --
--             10/08/92     AMT                             --
--     Purpose:                                             --
--     A set of unsigned arithemtic, conversion,            --
--              and comparison functions of STD_LOGIC_VECTOR. --
--     Note: comparision of same length discrete arrays is defined --
--              by the LRM. This package will "overload" those --
--              definitions                                 --
------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

package STD_LOGIC_UNSIGNED is

    function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return STD_
LOGIC_VECTOR;
```

```
    function "+"(L: STD_LOGIC_VECTOR; R: INTEGER) return STD_LOGIC_VECTOR;
    function "+"(L: INTEGER; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC) return STD_LOGIC_VECTOR;
    function "+"(L: STD_LOGIC; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;

    function "-"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return STD_
LOGIC_VECTOR;
    function "-"(L: STD_LOGIC_VECTOR; R: INTEGER) return STD_LOGIC_VECTOR;
    function "-"(L: INTEGER; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC_VECTOR; R: STD_LOGIC) return STD_LOGIC_VECTOR;
    function "-"(L: STD_LOGIC; R: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;

    function "+"(L: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;

    function "*"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return STD_
LOGIC_VECTOR;

    function "<"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "<"(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "<"(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "<="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "<="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "<="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;

    function ">"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function ">"(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function ">"(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;

    function ">="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function ">="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function ">="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;

    function "="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;

    function "/="(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function "/="(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "/="(L: INTEGER; R: STD_LOGIC_VECTOR) return BOOLEAN;
    function SHR(ARG: STD_LOGIC_VECTOR; COUNT: STD_LOGIC_VECTOR) return STD_
LOGIC_VECTOR;
    function SHR(ARG: STD_LOGIC_VECTOR; COUNT: STD_LOGIC_VECTOR) return STD_
LOGIC_VECTOR;

    function CONV_INTEGER(ARG: STD_LOGIC_VECTOR) return INTEGER;

-- remove this since it is already in std_logic_arith
--     function CONV_STD_LOGIC_VECTOR(ARG: INTEGER; SIZE: INTEGER) return STD_
LOGIC_VECTOR;
end STD_LOGIC_UNSIGNED;
```

主要参考文献

[1] Douglasl L. Perry. VHDL, New York: McGraw - Hill,1991

[2] 長谷川裕恭. VHDLによるハードウエア設計入門. 東京: CQ 出版社, 1995年

[3] 長谷川裕恭. 例解 VHDLプログラミング. トランジスタ技術, 1993年3月号～1994年5月号

[4] 中林　祥惠/江森　玲/今井　正治. VHDL によるマイクロプロセッサ設計, インターフユース, 1994年2月号①～1995年7月号⑩

[5] 小林優. 入門 Verilog - HDL 記述, インターフユース, 1994年4月号 p135～p154

[6] David R. Coelho. The VHDL Handbook. Boston: Vantage Analysis Systems, INC, 1993年.

[7] 周祖成译. 电子设计硬件描述语言 VHDL, 北京学苑出版社, 1994年

[8] J. BHASKER. A Guide to VHDL syntax Based on the new IEEE std 1076—1993. AT&T Bell Laboratories, Allentown, PA.