电子设计竞赛·课程设计·毕业设计 指导丛书

VHDL入门·解惑·经典实例·经验总结

黄 任 编著

北京航空航天大学出版社

责任编辑：王　鸿
封面设计：runsign

ISBN 7-81077-589-8
9 787810 775892 >

ISBN 7-81077-589-8
定价：22.00元

电子设计竞赛·课程设计·毕业设计 指导丛书

VHDL 入门·解惑·经典实例·经验总结

黄 任 编著

北京航空航天大学出版社

内 容 简 介

本书分 4 部分对 VHDL 进行了系统的介绍。第 1 部分为入门篇，介绍了 VHDL 的常用语法及利用 VHDL 实现系统层次化设计的方法；第 2 部分为解惑篇，详细解答了 VHDL 初学者常见的一些问题；第 3 部分为实例篇，包括多个常用单元电路的 VHDL 程序和包括数字频率计在内的多个经典入门题目的参考程序，并给出了 2002 年北京市大学生电子设计竞赛(EDA 专项)的参考设计；第 4 部分为经验篇，对一些 VHDL 语句的可综合性进行探讨，并给出了 6 种可综合的进程语句的模板。

本书内容详实，语言通俗、易懂，附带大量经典入门练习题及其参考设计，可以帮助初学者在短时间内入门，可作为全国大学生电子设计竞赛的培训材料、电子科技活动的参考资料，也可供电子爱好者及高等院校的师生在进行数字电路设计实验时参考。

图书在版编目(CIP)数据

VHDL 入门·解惑·经典实例·经验总结/黄任编著.
北京：北京航空航天大学出版社，2005.1
ISBN 7-81077-589-8

Ⅰ.V… Ⅱ.黄… Ⅲ.硬件描述语言，VHDL
Ⅳ.TP312

中国版本图书馆 CIP 数据核字(2004)第 132708 号

VHDL 入门·解惑·经典实例·经验总结

黄 任 编著

责任编辑 王 鸿

*

北京航空航天大学出版社出版发行

北京市海淀区学院路 37 号(100083) 发行部电话：010－82317024 传真：010－82328026

http://www.buaapress.com.cn E-mail：bhpress@263.net

涿州市新华印刷有限公司印装 各地书店经销

*

开本：787×960 1/16 印张：16.75 字数：375 千字

2005 年 1 月第 1 版 2005 年 1 月第 1 次印刷 印数：5 000 册

ISBN 7-81077-589-8 定价：22.00 元

献给敬爱的郭莉老师，我的父亲黄禄河、母亲周欣和妹妹黄正及所有曾经教过我的老师，感谢他(她)们一直以来给予我的支持和无限关怀。

前 言

本书的特点

市面上关于 VHDL 的书汗牛充栋，为什么作者还要不辞辛苦地添砖加瓦呢？如果你抱有这种怀疑态度，不妨将本书与其他的 VHDL 教材比较一下，会发现本书有以下特点：

(1) 真正面向初学者

与其他入门的书不同，本书不求博大精深，只求深入浅出。本书并未覆盖 VHDL 的所用语法，因为有些语法不但艰深，而且一般的设计根本不会用到。学习这些语法不但无助于初学者设计水平的提高，反而容易挫伤初学者学习 VHDL 的积极性。很多 VHDL 门外汉都渴望有一本能把他们带入 VHDL 设计大门的真正的入门教程。为了保证初学者能看懂本书，作者每写完一章就请没有学过 VHDL 语言的学生试读，并根据他们的意见对书稿进行修改，因此本书虽然无法做到老妪能懂，但对有一定数字设计基础的读者来说是浅显易懂的。

(2) 可综合的 VHDL 语言

VHDL 语句大体上可以分为两种：一种是只可仿真，不可综合(即不能用硬件实现)的语句；另一种则是可综合的语句。对初学者来说，学习 VHDL 语言大都是为了在 CPLD/FPGA 上实现数字电路设计，因此语句的可综合性就十分重要。本书是一本面向实际应用的入门教程，书中所用语句均为可综合的 VHDL 语句，所有程序均在实验仪上调试通过。

(3) 内容全面，学、练结合

本书分为入门篇、解惑篇、实例篇和经验篇 4 部分，这几部分相辅相成，可以帮助读者在最短的时间内掌握 VHDL 语言。值得一提的是，实例篇中所用的题

目均是北京邮电大学信息工程学院的学生在准备2002年北京市大学生电子设计竞赛所做的练习。实践证明，这些题目对提高初学者的CPLD/FPGA开发水平是很有帮助的(经过两个月的训练，参赛学生有一半获得一等奖，而且成绩在获一等奖的学生中名列前茅)。本书给出的例程均为获一等奖学生的作品，具有参考价值。

本书的结构

本书分为4部分，各部分的内容简介如下：

(1) 入门篇

简要介绍VHDL语言的特点和设计流程；通过大量的简单实例深入浅出地讲解了VHDL的基本结构、数据类型和语法；介绍了Moore状态机和Mealy状态机在VHDL中的实现方法；介绍如何在MAX+plus II中采用图形法与文本法结合的混合输入方法实现元件重用与系统的层次化设计；最后，介绍了元件例化、程序包及类属映射等VHDL进阶语句，以及利用这些语句实现系统层次化设计的方法。

(2) 解惑篇

对初学者常见的一些问题(包括设计方法、仿真与综合以及MAX+plus II的使用等方面)进行详细的解答。

(3) 实例篇

给出了分频电路、七段数码管驱动电路、SRAM读/写电路等常用单元电路的VHDL程序；详细讲解了交通灯控制器、乒乓球游戏机、数字频率计、WCDMA短码生成器、Franaszek码编解码器等系统的设计方法，并给出了源程序，同时给出了2002年北京市大学生电子设计竞赛题——自动打铃系统的系统框图和参考程序；此外，在每章后附加了练习题，读者可以通过这些题目的训练，迅速提高VHDL设计水平。

(4) 经验篇

对LOOP语句和进程语句的可综合性进行了探讨，并给出了6种可综合的进程语句的模板，最后介绍了VHDL可综合编程的一般规则。

配套书籍

如果读者学完本书后想进一步了解CPLD/FPGA与单片机的综合应用，那么请参考北京航空航天大学出版社出版的本丛书中的《AVR单片机与CPLD/

FPGA综合应用入门》。该书系统论述了单片机与 CPLD/FPGA 之间的通信和合作方法，并全面阐述了高精度频率计、数控波形发生器、简易逻辑分析仪等单片机与 CPLD/FPGA 综合应用系统的设计思想和具体实现方法。《VHDL 入门·解惑·经典实例·经验总结》既是该书第 3 章的扩展，又可作为该书的基础教程，二者相辅相成，可帮助读者在最短时间内掌握单片机和 CPLD/FPGA 的应用。

致　谢

本书在编写过程中，得到了北京邮电大学信息工程学院创新实验室主任郭莉副教授的大力帮助与支持。郭莉老师带领创新实验室的学生参加了多次电子设计竞赛，取得令人瞩目的成绩，参赛学生获奖率为 100%，其中一半的学生获得了一等奖。郭莉老师在百忙之中仍仔细审阅了全书，并提出了许多宝贵的意见，在此向她表示衷心的感谢。

本书在正式出版前，曾作为北京邮电大学信息工程学院创新实验室的内部教材，用以指导学生参加各种电子设计竞赛。本书得到了学生的肯定，也从参赛学生那里得到了很多反馈意见。在此要特别感谢杨芳芳、张万能、赵荣华、于光炜、刘阳、赵莹、王志勇、胡子明等同学，他们的意见对本书的改进起到了十分关键的作用。

特别值得一提的是，掌宇公司(http://www.kandh.com.cn)为作者提供了实验仪。本书中所有程序均在该实验仪上调试通过，在此对于掌宇公司的大力支持表示感谢。

此外，本书的练习题部分选自北京邮电大学信息工程学院田宝玉副院长的《VHDL》课程练习和电路中心的《数字逻辑电路实验》课程讲义，在此一并致谢。

本书的相关网站

本书的相关网站地址为 http://www.ele-contests.com。本网站提供本书中部分源程序以及一些 VHDL 相关资料的下载。

如果在阅读本书的过程中发现任何错误，或是有任何改进本书的建议，请通过 huangren@ieee.org 与作者联系，最新的勘误表将会在本书的相关网站上登出。

作　者

于韩国国立汉城大学 SOC Center

2004 年 12 月

目 录

入门篇

第 1 章 VHDL 初探

第 2 章 VHDL 入门

实例篇

第 7 章 常用电路的 VHDL 程序

第 8 章 交通灯控制器

門篇

第1章

VHDL 初探

1.1 VHDL 简介

VHDL 是 Very high speed integrated circuit Hardware Description Language 的缩写，即“甚高速集成电路硬件描述语言”，最初由美国国防部和 Intermetrics、IBM、TI 公司联合开发，1987 年成为 IEEE 标准，即 IEEE1076 标准(俗称 87 版 VHDL)。此后，美国国防部要求官方的与高速集成电路设计相关的所有文档必须用 VHDL 描述，因此 VHDL 在电子设计领域得到了广泛的应用，渐渐成为工业界的标准。1993 年，IEEE 对 VHDL 进行了修订，公布了新的 VHDL 标准，即 IEEE1076—1993 版(俗称 93 版 VHDL)。

1.1.1 VHDL 的特点

VHDL 的主要特点包括：

1. 支持“自顶向下”的设计方法

设计可按层次分解，采用结构化开发手段，可实现多人、多任务的并行工作方式，使系统的设计效率大幅提高。

2. 系统硬件描述能力强

可以同时支持“行为描述”、“数据流描述”和“结构描述”3种描述方式，并可混用。其中，强大的“行为描述”能力使设计者可以避开具体的器件结构，从逻辑行为上描述和设计大规模电子系统。这一特点使VHDL成为系统设计领域中最佳的硬件描述语言。

3. 系统仿真能力强

VHDL最初是作为一种仿真标准问世的，因此VHDL具有丰富的仿真语句和库函数。另外，VHDL强大的“行为描述”能力也使其十分适用于系统级的仿真。

4. 工艺无关性

在使用VHDL设计系统硬件时，没有嵌入与工艺相关的信息。正因为VHDL的硬件描述与具体工艺无关，因而其程序的硬件实现目标器件有广阔的选择范围，其中包括各种CPLD、FPGA及ASIC等。

5. 其 他

VHDL的特点当然不只上面列出的这几点，但作为一本VHDL的入门书，本书无意(也不可能)将VHDL的所有特点一一列出。关于VHDL的好处，还是请读者在实践中仔细体会吧。

1.1.2 设计流程

VHDL设计的基本流程如图1-1所示，这一流程基本可适用于任何基于硬件描述语言的设计。

下面对这一流程中的关键步骤进行简要说明：

(1) 系统层次划分/画出系统框图(Hierarchy/Block Diagram)

按照“自顶向下”的设计方法对系统进行划分(确定系统由哪些模块构成，各模块又由哪些子模块构成)。

(2) 编码(Coding)

写出VHDL代码。原则上此工作可在任何文本编辑器内完成，但大多数集成开发环境

(如 MAX+plus II 等)都集成了针对 VHDL 的编辑器。这些编辑器一般都具有 VHDL 关键词的高亮显示等特点,有的还内嵌了常用的 VHDL 程序模板等。

(3) 编译(Compilation)

编译器会对 VHDL 程序进行语法检查,还会产生用于仿真的一些内部信息。这一步骤通常由编译器自动完成,毋须我们干预。如果 VHDL 语法有错误,编译无法通过,则需要修改程序,即回到第(2)步。事实上,在 VHDL 设计过程中,常常根据需要往后退一步,甚至更多,这也是流程图中出现很多"往回走"的箭头的原因。

(4) 功能仿真(Functional Simulation)

VHDL 仿真器允许定义输入并应用到设计中,不必生成实际电路就可以观察输出。此仿真主要用于检验系统功能设计的正确性,不涉及具体器件的硬件特性。

(5) 综合(Synthesis)

利用综合器对 VHDL 代码进行综合优化处理,生成门级描述的网表文件,这是将 VHDL 语言描述转化为硬件电路的关键步骤。这一步通常由综合器自动完成,但设计者可以设定一些技术上的约束条件(如限定逻辑层次的最大数等)来"帮助"综合器。

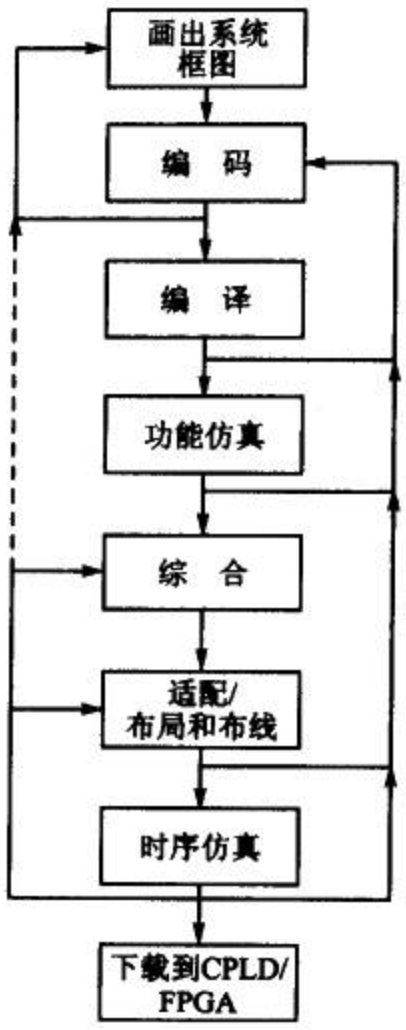


图 1-1 VHDL 设计流程

(6) 适配(Fitting)

利用适配器将综合后的网表文件针对某一具体的目标器件进行逻辑映射操作,包括底层器件配置、逻辑分割、逻辑优化、布局布线等。此步骤将产生多项设计结果:① 适配报告,包括芯片内部资源的利用情况、设计的布尔方程描述情况等;② 适配后的仿真模型;③ 器件编程文件。

(7) 时序仿真(Timing Simulation)

根据适配后的仿真模型,可以进行时序仿真。因为这时已经得到目标器件的实际硬件特性(如时延特性等),所以仿真结果能比较精确地预期芯片的实际性能。如果仿真结果达不到设计要求,就需要修改 VHDL 源代码或选择不同的目标器件,甚至要重构整个系统(看到图 1-1里那个虚线的箭头了吗? 这是所有设计者极力避免出现的情况)。

(8) 下载到 CPLD/FPGA(Programming)

如果时序仿真通过,那么可以将"适配"时产生的器件编程文件下载到 CPLD 或 FPGA 中

(FPGA的编程通常被称为“配置”)。虽然流程图中未标出从此步“往回走”的箭头,但事实上,实际的结果有可能与仿真结果有差异(可能是设计时未考虑到外部硬件的实际情况;也可能是由于仿真时测试的条件不够多,没有发现其中隐藏的错误),这时,必须再回头重新找出问题所在。

不断地回头查错、改错是痛苦的,而到最后一步才发现整个系统必须推倒重来更是令人难以忍受。此处引用John F. Wakerly说的一句话作为本节的结束语:“That's worth remembering—excellent tools are still no substitute for careful though at the outset of a design.(值得记住的是:出色的工具并不能代替设计之初的仔细思考。)”

1.1.3 初学者如何学好VHDL

初学者想要尽快跨入VHDL设计的大门,需要注意以下几点:

1. 掌握时钟的概念

CPLD/FPGA的一大优势就在于它能实现复杂的时序,而时序电路的心脏就是时钟。在用VHDL进行时序电路设计时,脑子里一定要有时钟的概念。只有清楚地意识到每个语句将在时钟沿产生一个什么样的结果,才能写出正确的语句;否则,通过不断的仿真来修正错误,既耗时又难以保证结果的正确。

2. 注意VHDL编程与软件编程的差别

初学者,特别是习惯于PC机上软件编程和单片机编程的人,刚开始很难适应VHDL中的“并发执行”语句,往往在这上面犯错。下面的章节中会对这个问题进行讨论。

3. 语法学习贵精不贵多,靠练不靠背

有人说30%的基本VHDL语句可以实现95%的电路设计,可谓一矢中的。初学者学习VHDL语言,千万不要贪多,应该把精力放在常用的语句上;不要死记硬背语法,因为这样会使学习变得很枯燥。不妨大体了解一下VHDL语言有些什么语句,在实际应用时需要哪个语句,可以现查现用,多用几次以后,自然能将这个语句牢牢记住,这比死记硬背效果要好得多。VHDL语言和任何程序语言一样,必须靠练习来积累应用经验,光记住语法不够,还应该知道如何应用。

4. 注意VHDL语句的可综合性

VHDL的主要作用有两个:一为系统仿真;二为硬件实现。VHDL的所有语句均可以用于仿真,但只有一部分可以用硬件实现(可综合)。初学者学习VHDL,往往是为了在CPLD/FPGA上实现某一个具体的功能,而不是为了系统级的仿真。因此在学习时,要特别注意哪些语句是可以综合的,哪些语句是不可被综合的。虽然各种综合工具支持的可综合语句数量不

同,但基本的语句(如 IF_ELSE 语句等)在所有综合工具中都得到很好的支持。本书中的所有程序均可在包括 MAX+plusII 的综合器在内的大部分综合工具中被综合。

1.2 VHDL 与 MAX+plus II 的初体验

看到 1.1 节中关于 VHDL 的介绍,你是不是跃跃欲试,想立即编一个 VHDL 试试? 下面,就在 Max+plusII 中编写、仿真并综合我们的第 1 个 VHDL 程序——2 分频电路。

在开始使用 Max+plusII 之前,有必要对此软件及其开发流程进行简要的介绍。

MAX+plus II 是世界最大的可编辑逻辑器件供应商之一的 Altera 公司推出的一款 CPLD/FPGA 开发平台。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,具备编辑、编译、仿真、综合、下载等功能。用户可以在此软件中完成从源代码输入到最后烧录到芯片的全部开发工作。虽然此软件支持的 VHDL 语句比专业的仿真和综合工具要少,但是由于它的简便易用和 All-in-one(全部功能都集成在一个软件中)特性,在国内高等院校教学中得到了很广泛的应用。

使用 MAX+plus II 进行 CPLD/FPGA 设计的流程如图 1-2 所示,下面就按这个流程来设计并实现一个 2 分频电路。

1.2.1 设计输入

MAX+plus II 支持的设计输入方式主要有 4 种:图形输入、AHDL(Altera 公司自定义的 HDL)、Verilog HDL 以及 VHDL。这里只介绍 VHDL 设计输入方式,关于图形输入的内容,在第 4 章中会再补充。

(1) 打开 Text Editor

在主菜单最左边的 MAX+plus II 子菜单中选择 Text Editor,出现如图 1-3 所示的窗口。

(2) 存储文件

在 File 菜单中选择 Save,打开如图 1-4 所示的对话框,选择欲存放的路径并输入文件名 FreDevider.vhd。

注意:存储路径中不得出现中文字符,不得将文件存于根目录下,否则无法编译。

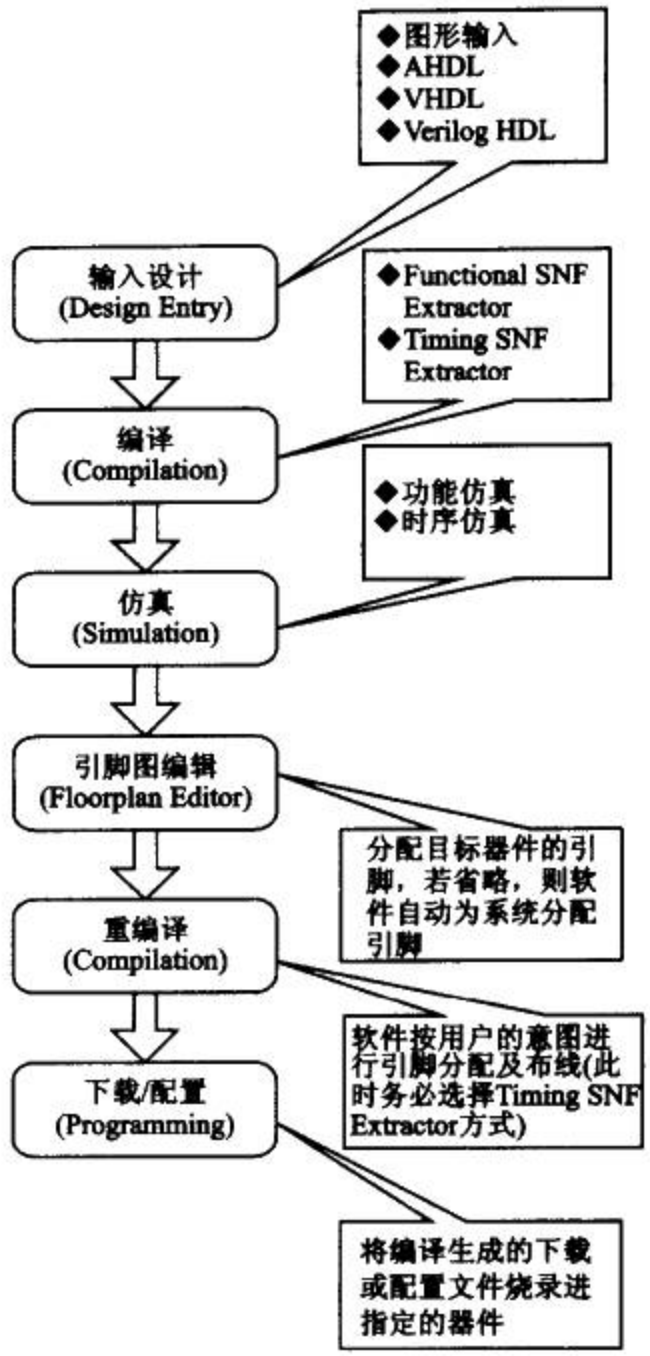


图 1-2　用 MAX+plus II 进行 CPLD/FPGA 设计的基本流程

(3) 输入 VHDL 程序

在 Text Editor 中输入例程 1-1 的程序。在此给一个小小的建议：MAX+plus II 默认的字体太难看，不妨将字体改为 Courier(在 Option 菜单中的 Font 一栏内可修改字体)。

提示：在输入代码较多的情况下，要及时存储。存储文件的快捷键与一般的文字编辑软件一样，都是 Ctrl+S。

注意：VHDL 程序不区别大、小写，因此没有必要像例程 1-1 这样把所有关键字都用大写字母表示。

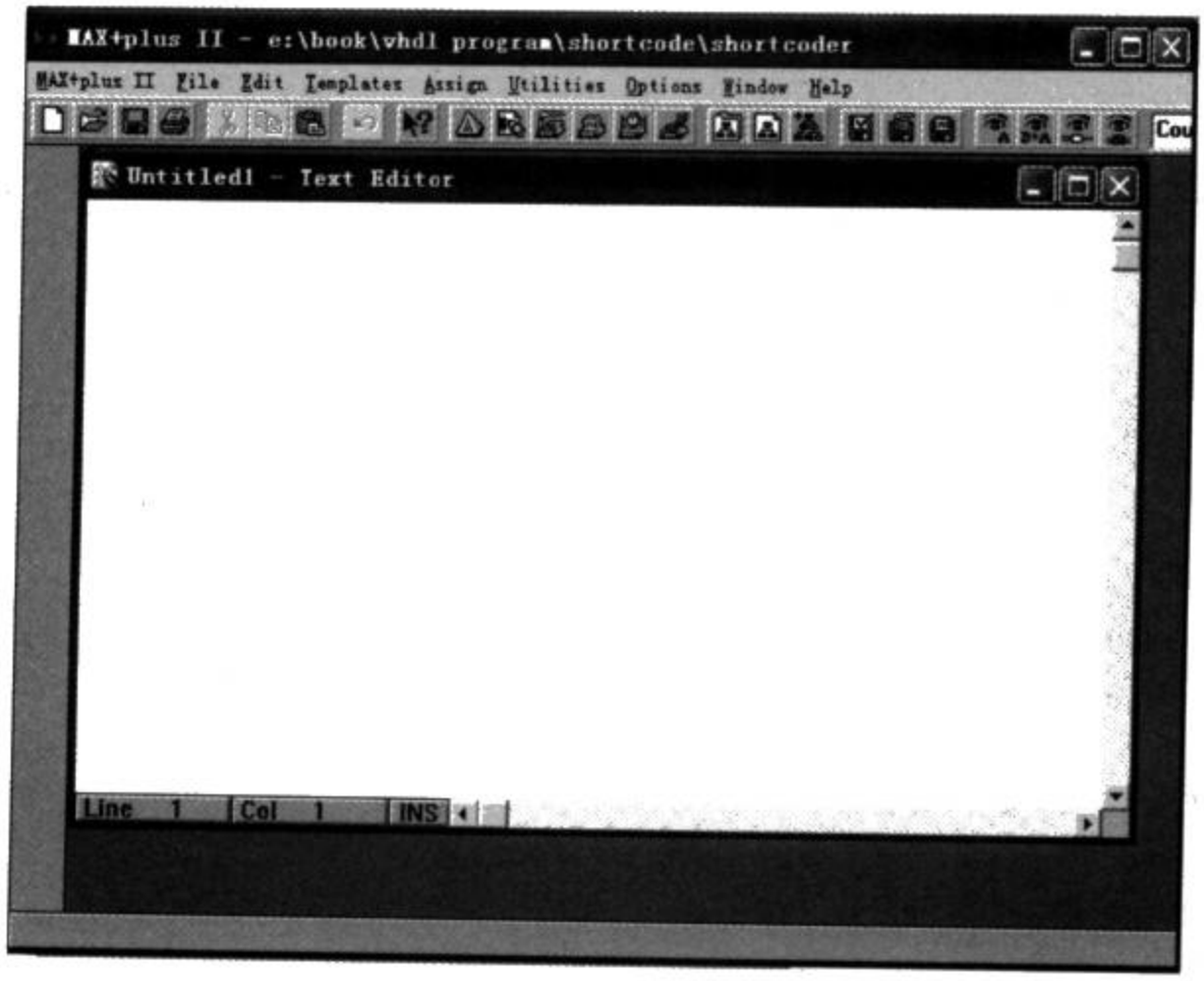


图 1-3　Text Editor 界面

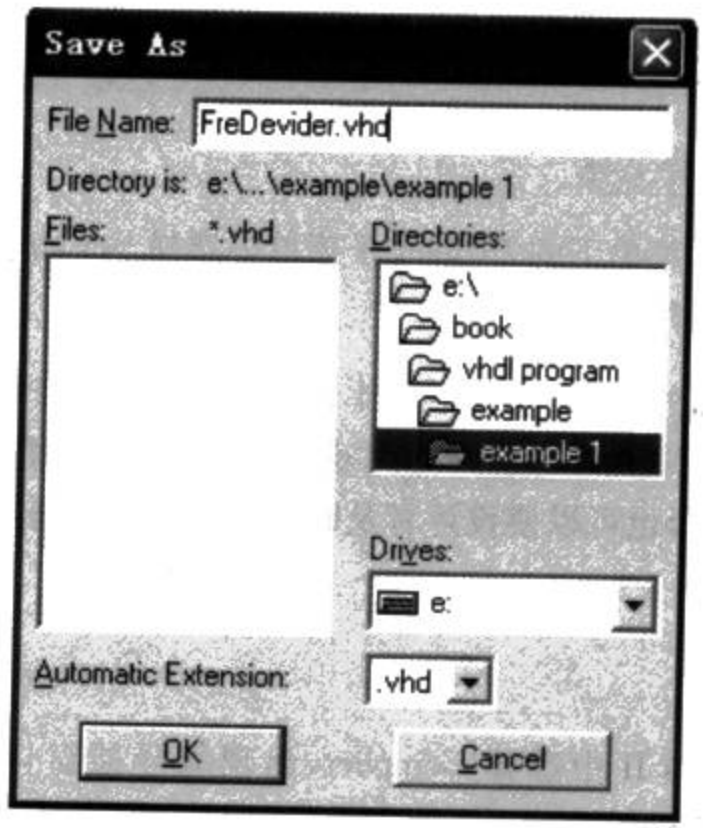


图 1-4　存储文件对话框

例程1-1 2分频电路

```
LIBRARY IEEE;
USE IEEE.Std_Logic_1164.All;

ENTITY FreDevider Is
PORT
( Clock:IN Std_Logic;
  Clkout:OUT Std_Logic
);
END;

ARCHITECTURE Behavior OF FreDevider IS
SIGNAL Clk:Std_Logic;
BEGIN
  PROCESS(Clock)
  BEGIN
    IF rising_edge(Clock) THEN
      Clk<=NOT Clk;
    END IF;
  END PROCESS;
  Clkout<=Clk;
END;
```

(4) 将当前文件设为工作文件

如图1-5所示,在File菜单下的Project子菜单中选择Set Project to Current File,即将FreDevider.vhd设为工作文件。

可能有读者会觉得前面的步骤有问题:为什么先存储一次文件再输入代码呢?这是因为,**只有把文件存成.vhd格式后,MAX+plus II才会对VHDL的关键字进行"变色"显示。这样使整个程序看起来比较清晰,也可以使设计者及时发现关键字的拼写错误。**

1.2.2 编 译

在主菜单的MAX+plus II中选择Compiler,出现编译器窗口。此时,可以在主菜单的Processing中选择Functional SNF Extractor或Timing SNF Extractor,前者用于功能仿真,后者用于时序仿真。这里先试试功能仿真的功能,因此选择Functional SNF Extractor,如图1-6所示。

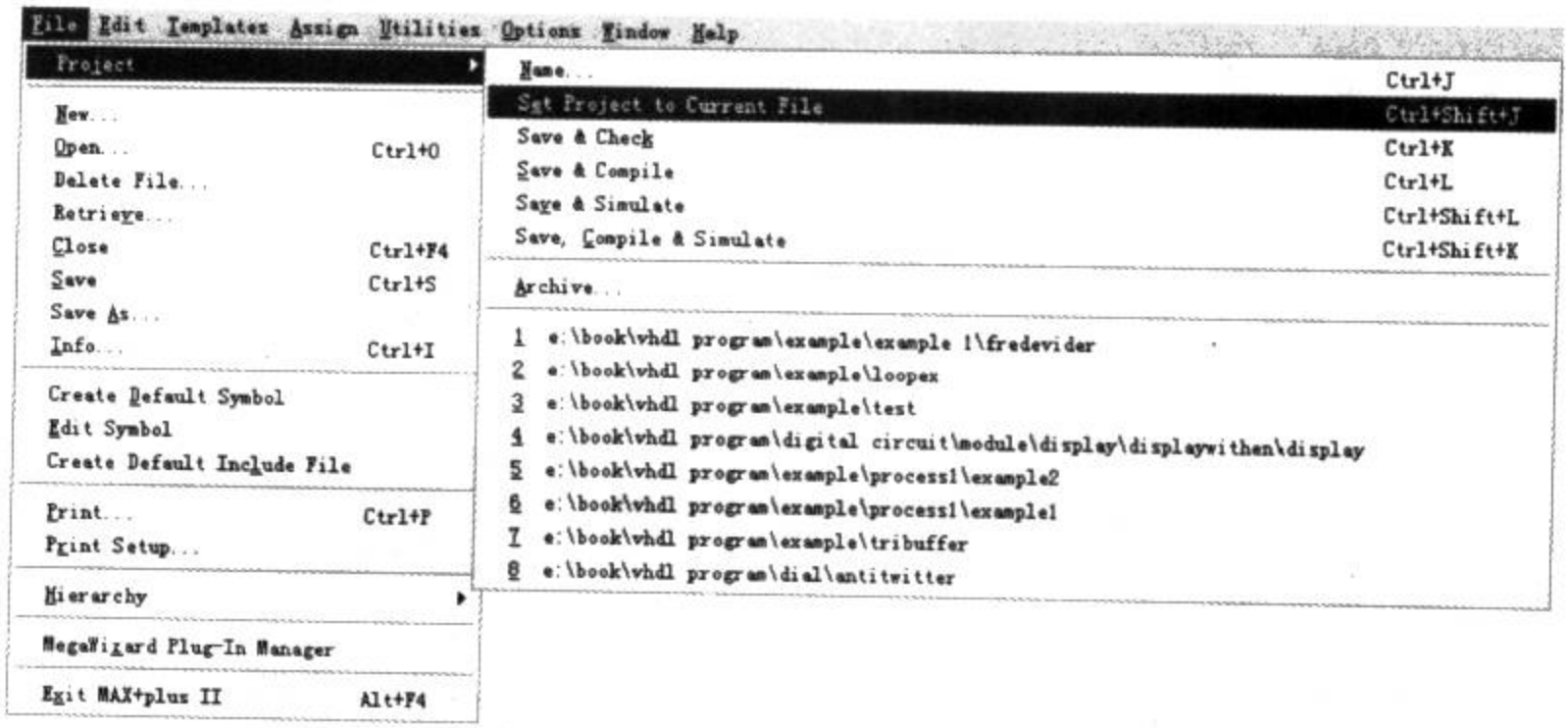


图 1-5　将当前文件设为工作文件

单击 Start 按钮开始编译，如果出错，会出现错误提示；否则会有信息窗口提示编译成功。

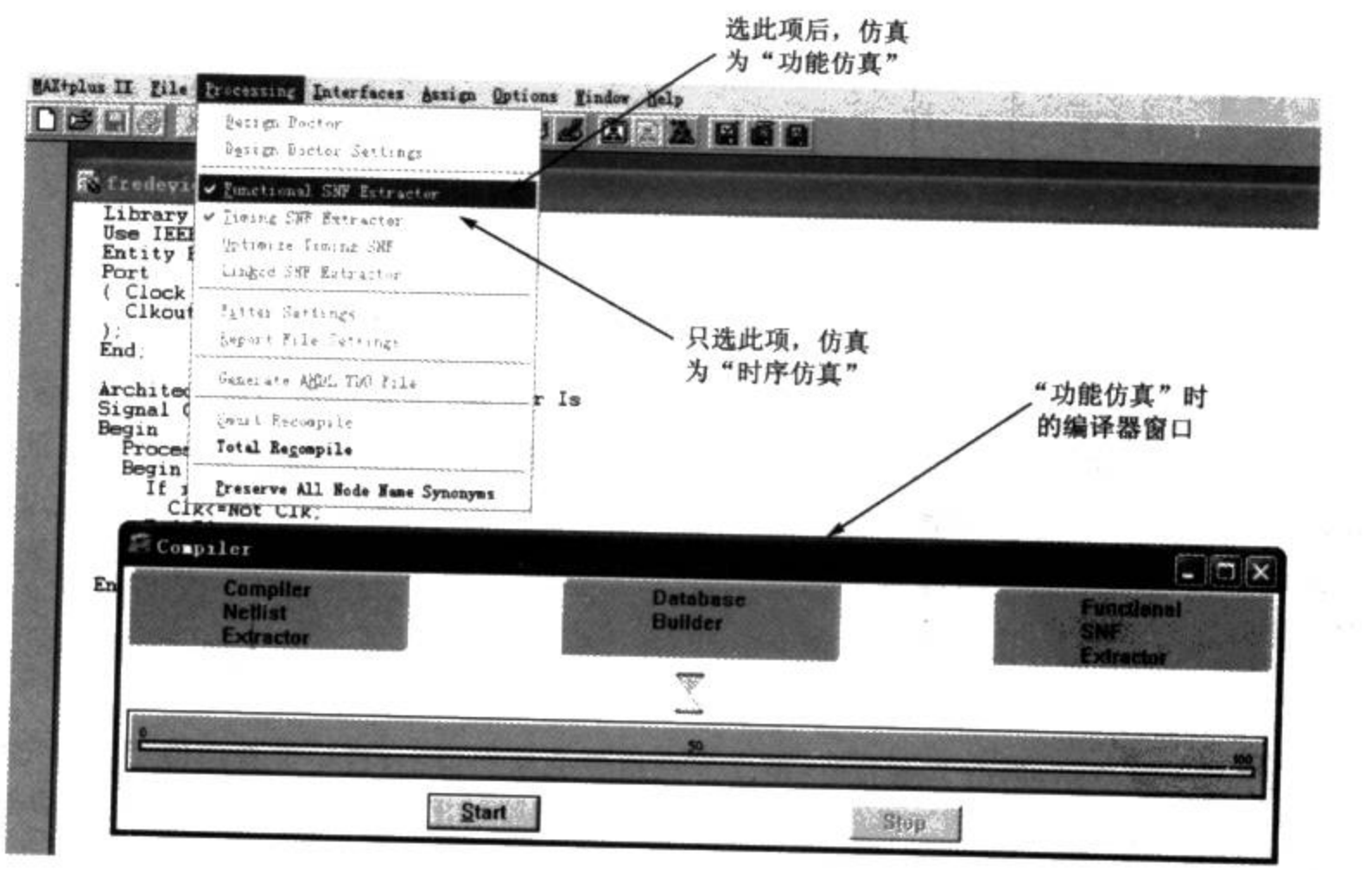


图 1-6　选择 Functional SNF Extractor 时的编译器窗口

1.2.3 仿 真

(1) 打开波形编辑器中的“输入节点”对话框

在主菜单中的 MAX+plus II 子菜单中选择 Waveform Editor,打开波形编辑器。然后在 Node 菜单中选择 Enter Nodes from SNF,如图 1-7 所示。

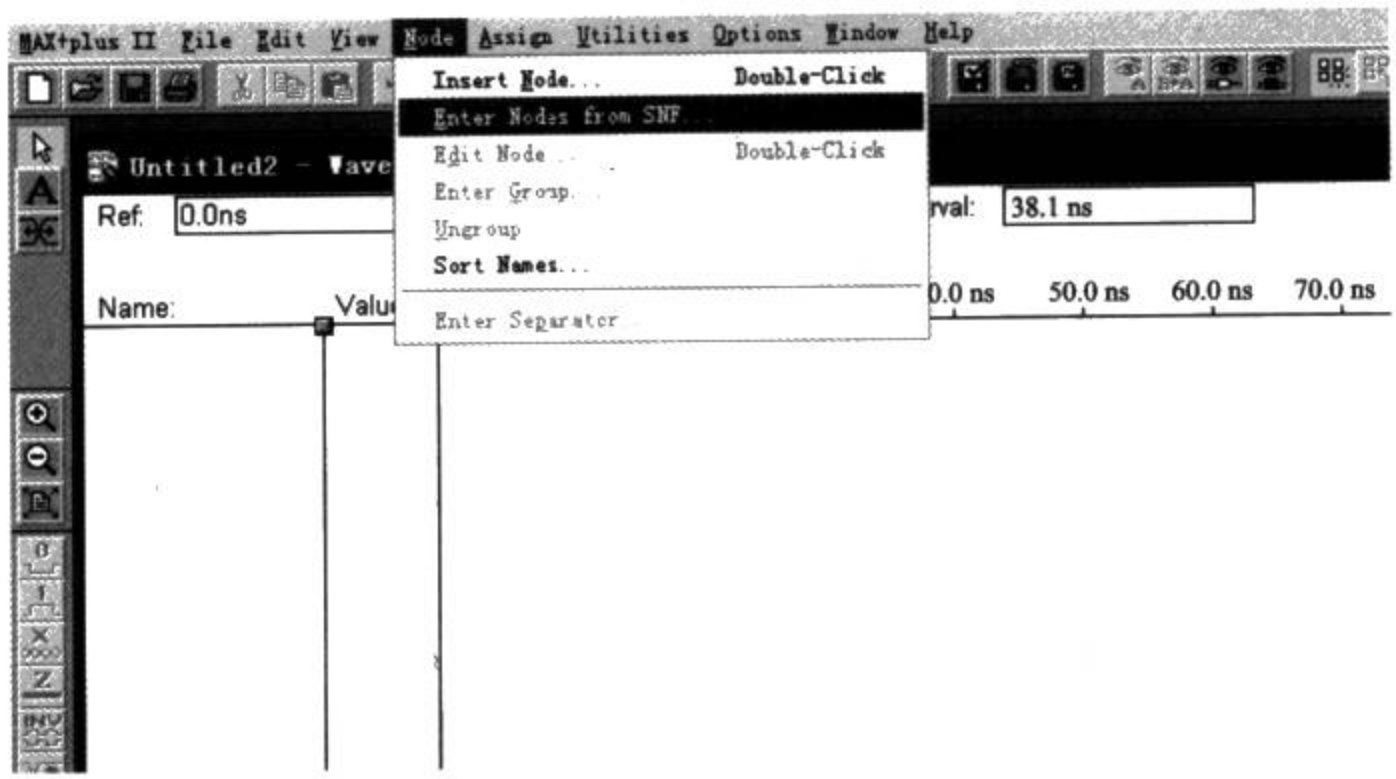


图 1-7 打开输入节点对话框

(2) 选择要观察的节点

在如图 1-8 所示的对话框中,单击右上角的 List 按钮,在左边的 Available Nodes & Groups 中就会出现与左下角的 Type 对应的节点(不同 Type 对应的节点数是不同的,这里选择的是输入/输出节点)。选择所需观测的节点,然后单击“=>”,这些节点就会出现在右边的 Selected Nodes & Groups 中。这就完成了对节点的选择,单击 OK,选择的节点波形即出现在波形编辑器中(如图 1-9 所示)。

(3) 对 Clock 进行赋值

如图 1-10 所示,单击节点名后,整个节点信号会反白显示;再单击左边的工具栏中的 Clock 赋值工具(对其他赋值工具的使用,请读者自己试验),出现如图 1-11 所示的对话框。在此对话框中,可以修改时钟起始值和实际时钟周期倍数等值。Clock Period 与 Option 中的 Grid Size 的设置值有关,这里 Grid Size 设为 10 ns,所以 Clock Period 为 20 ns(总是 2 倍的关

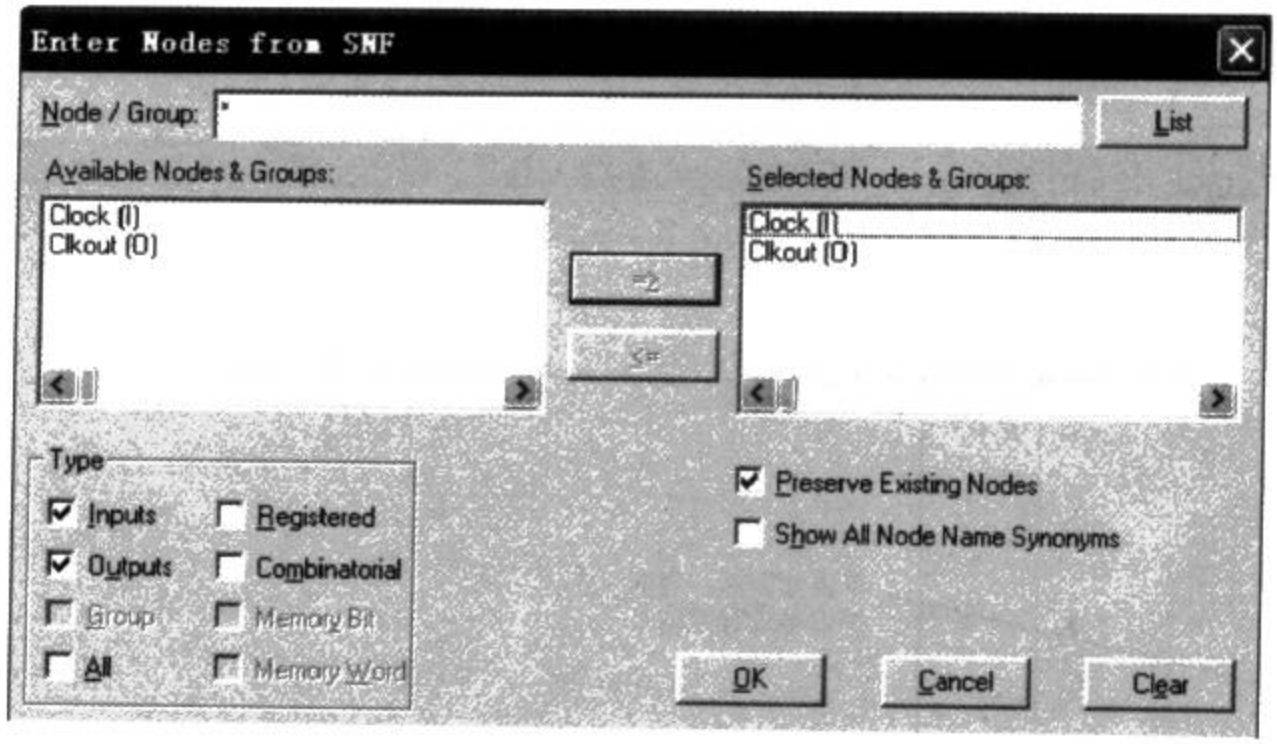


图 1-8　选择欲观测的节点

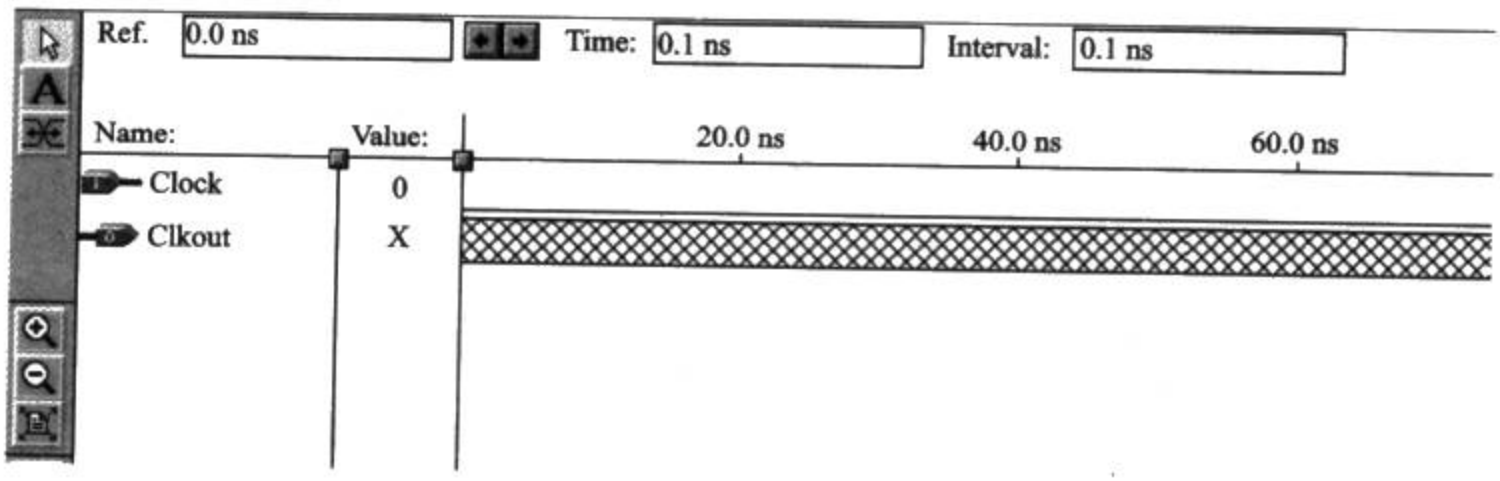


图 1-9　选择的节点波形出现在波形编辑器中

系)。这里按图 1-11 进行设置后,就将时钟设为周期为 20 ns 的周期性信号(如图 1-12 所示)。

(4) 进行仿真

先将波形存盘,然后调用 MAX+plus II 菜单下的 Simulator,出现图 1-13 所示的对话框。单击 Start 开始仿真,仿真结果如图 1-14 所示。这里可以很直观地看出,Clkout 的周期为 40 ns,是 Clock 周期的 2 倍,即频率为 Clock 的 1/2,满足了 2 分频的要求。

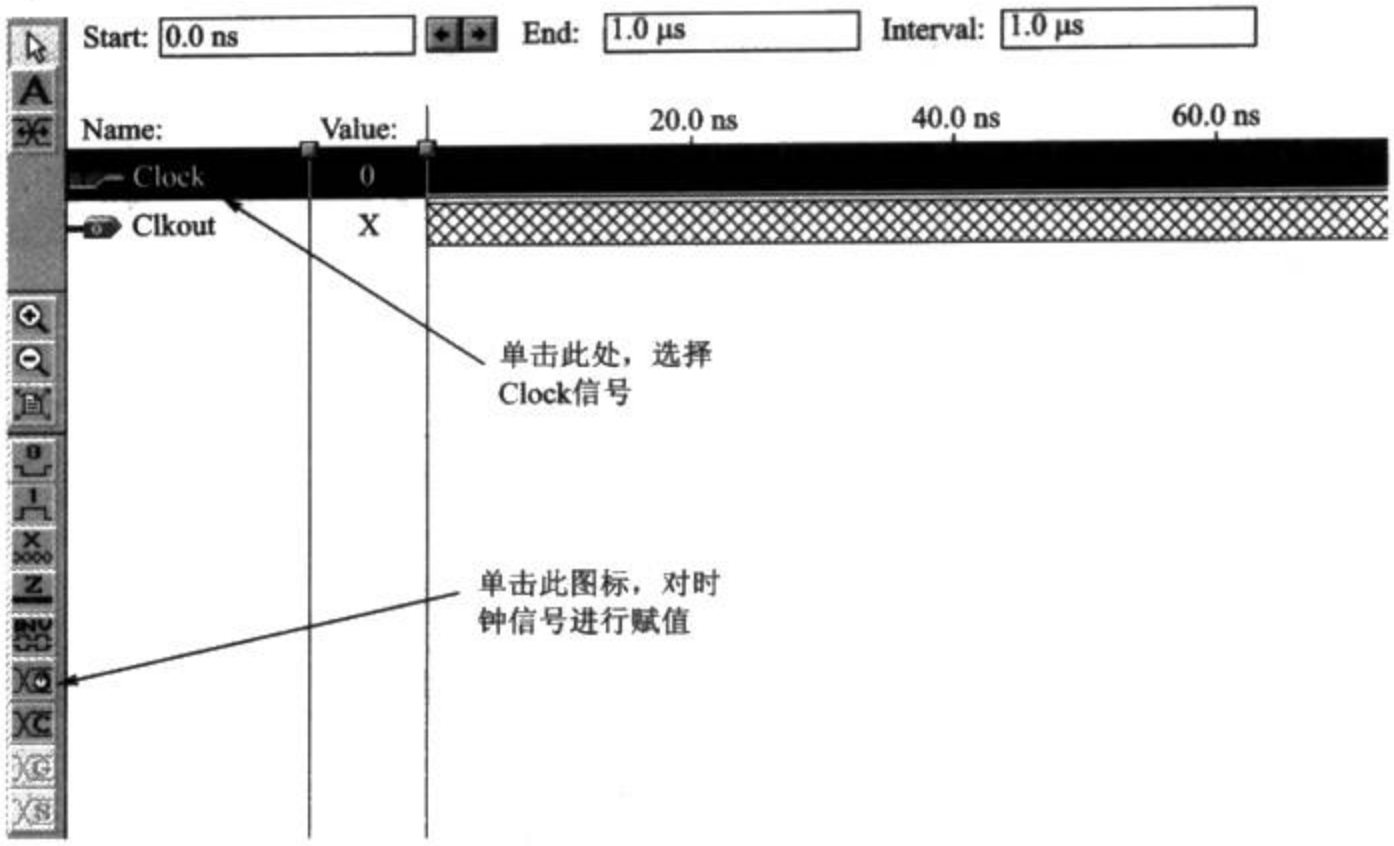


图 1－10　对时钟信号进行赋值的方法

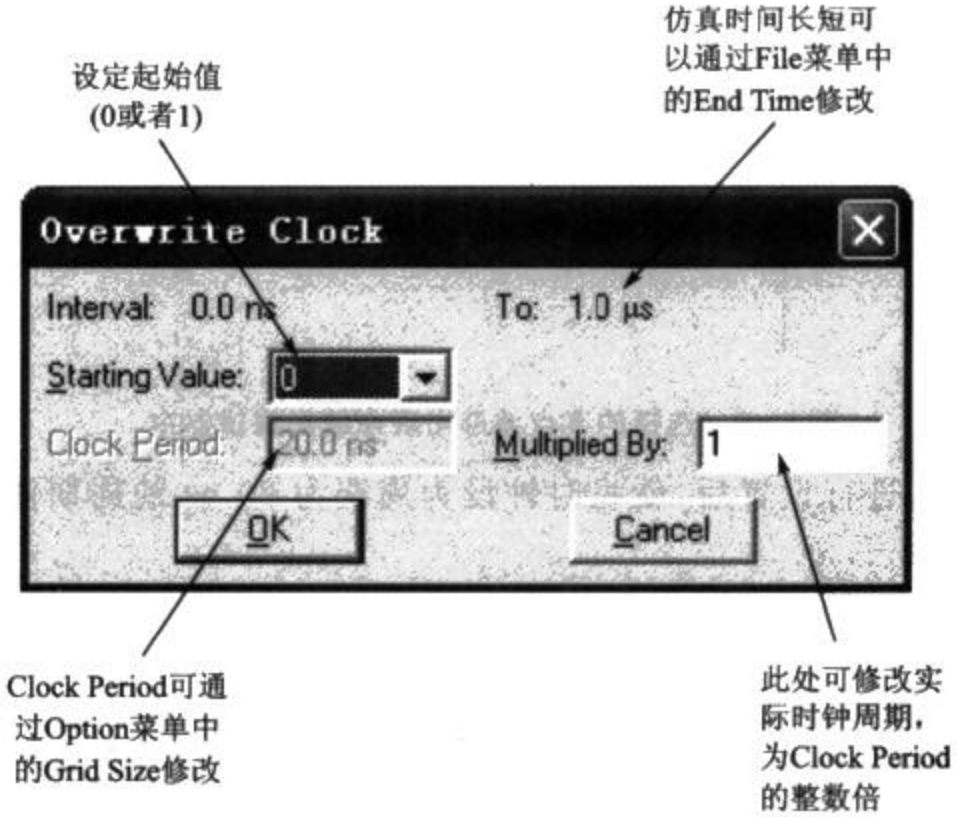


图 1－11　时钟赋值工具对话框

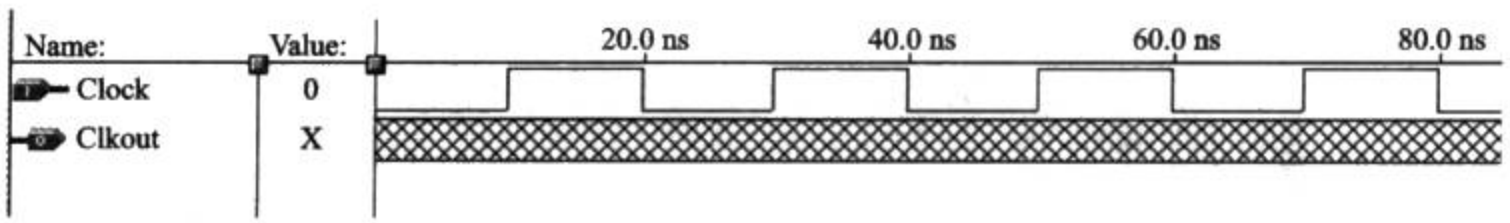


图 1－12　将时钟设为周期为 20 ns 的信号

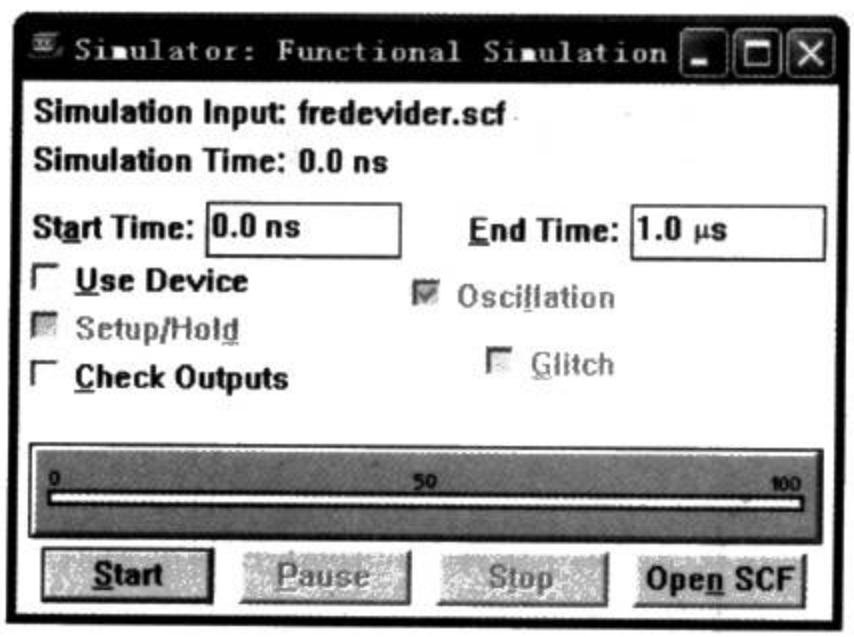


图 1－13　仿真器对话框

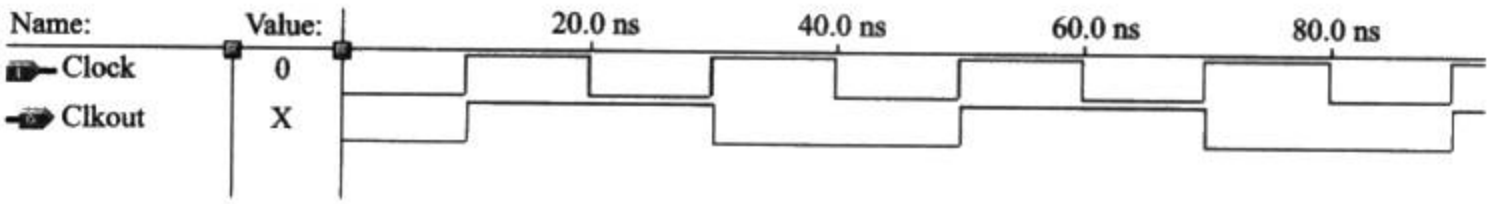


图 1－14　例程 1－1 的仿真波形

1.2.4　引脚图编辑

接下来的工作就是用硬件来实现用 VHDL 设计的 2 分频电路了。首先要做的就是选择一个目标器件。

(1) 选择目标器件

在主菜单的 Assign 子菜单中选择 Device，出现图 1－15 所示的对话框。首先选择 Device Family，然后再确定目标器件。我们发现每个器件名后都跟一条横线，横线后有一个数字，此数字表征所选器件的速度，例如，“－6”表示所选器件的传输时延为 6 ns。这里选择

EPM7128SLC84－6作为目标器件。

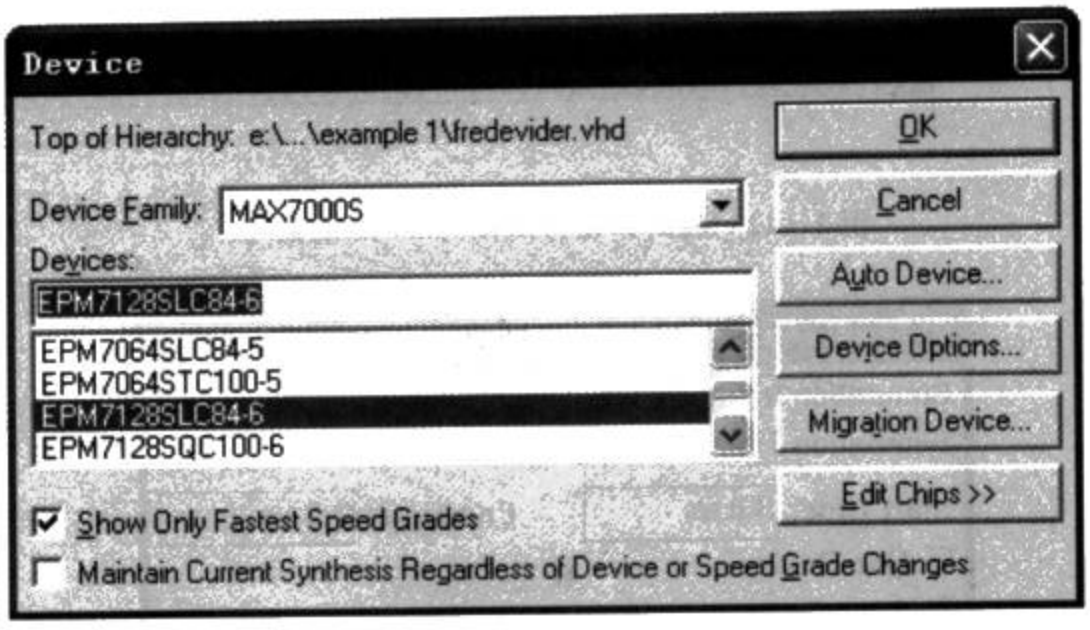


图1－15 选择器件对话框

(2) 分配引脚

在主菜单的MAX+plus II子菜单中选择Floorplan Editor，然后在Floorplan Editor中将Layout子菜单中的选项设置为Device View和Current Assignments Floorplan，结果如图1－16所示。将Clock拖到83引脚(此引脚在硬件上应该接晶振，用以提供稳定的时钟)，把Clkout拖到任意一个标着(I/O)的引脚。此时，引脚分配即告完成。

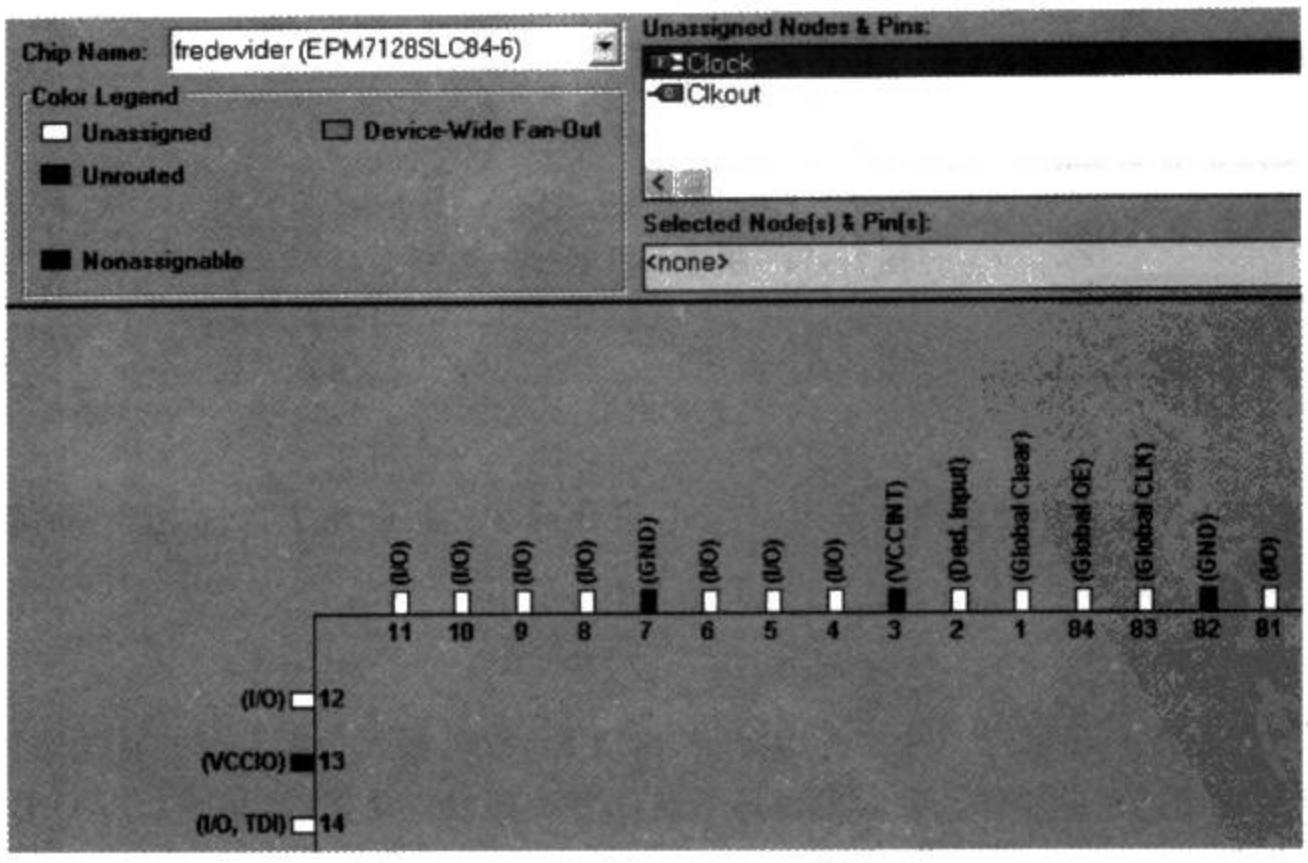


图1－16 Floorplan Editor的界面

1.2.5　重编译与时序仿真

分配完引脚后，必须再次进行编译。这一次编译不可选 Functional SNF Extractor，而只能选 Timing SNF Extractor，编译时的窗口如图 1－17 所示。

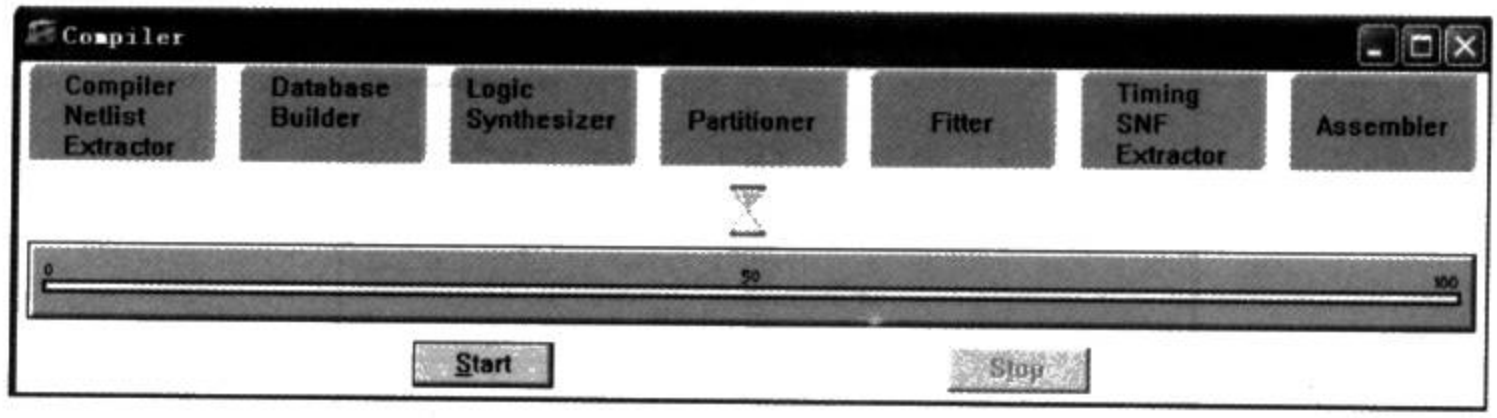


图 1－17　选择 Timing SNF Extractor 时的编译器窗口

我们很容易发现，此时的编译器窗口比之前多了以下几项：① Logic Synthesizer(逻辑综合器)；② Partitioner(分割器)；③ Fitter(适配器)；④ Assembler(装配器)。另外，仿真器网表文件生成器也由之前的 Functional SNF Extractor 变成 Timing SNF Extractor。综合器与适配器的作用在前面介绍 VHDL 设计流程时已经提及，此处不再多作解释。Partitioner 的作用是：如果器件在编译时尚未选择目标器件，那么分割器会自动选择合适的器件；如果项目对硬件资源要求过多而使目标器件不满足要求，那么分割器将把逻辑综合的结果分割成几部分，便于用户采取芯片级联的方法实现设计(初学者往往用不到此功能，此处不再介绍)。而 Assembler则用于生成用于器件下载/配置的文件。

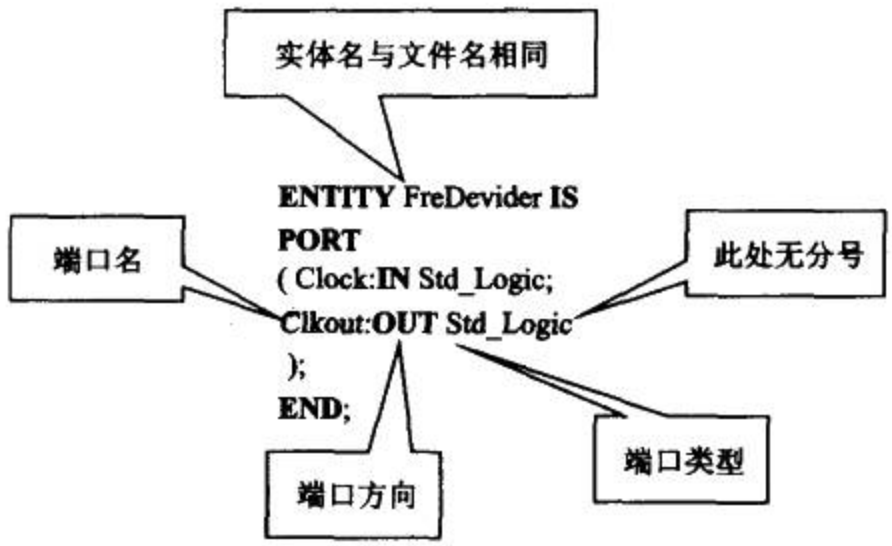
在此次编译后，可以进行时序仿真。时序仿真与功能仿真的步骤完全一样，因此这里就不再重复了，读者可以自己试验一下。需要注意的是，在 Waveform Editor 中，时钟周期不可比所选器件的传输时延短。例如，选择传输时延为 15 ns 的器件一定不可以把时钟周期设为 10 ns，至于其中的道理，读者一试便知。

1.2.6　下　载

在主菜单的 MAX+plus II 子菜单中选择 Programmer 来启动编程器。如果Programmer 是第 1 次启动，那么会出现如图 1－18 所示的窗口，提示用户对下载线进行设置。此处选择的是并口下载线，所以用图中的参数进行设置(对于其他类型下载线的设置，请参考所购下载线的说明。一般高等院校中使用的均为并口下载线，因此图 1－18 中的配置具有代表性)。

② 最后一条端口声明语句后不可加分号。

下面是例程2-1的实体部分与实体声明格式的对应关系，望能藉此牢记实体声明的格式。



2.1.2 结构体

结构体描述实体内部的结构或功能。一个实体可对应多个结构体，每个结构分别代表该实体功能的不同实现方案或不同描述方式。在同一时刻，只有一个结构体起作用，可以通过配置(CONFIGURATION)来决定使用哪一个结构体进行仿真或综合。

结构体的语法格式如下：

```
ARCHITECTURE 结构体名 OF 实体名 IS
[声明语句]
BEGIN
功能描述语句
END [结构体名];
```

实体名必须与实体声明部分所取的名字相同，而结构体名则可由设计者自由选择，但当一个实体具有多个结构体时，各结构体的取名不可相同。

声明语句用于声明该结构体将用到的信号、数据类型、常数、子程序和元件等。值得注意的是，在一个结构体内声明的数据类型、常数、子程序(包括函数和过程)和元件只能用于该结构体中。如果希望在其他的实体或结构体中引用这些定义，那么需要将其作为程序包(PACKAGE)来处理。

1.2.7 实际验证

如果你身边有示波器或者数字频率计，不妨测试一下，Clkout 所在引脚的输出频率是否为 83 脚晶振频率的 1/2。

做完了第一个小实验，感觉如何？是不是很有成就感呢？那就让我们继续努力吧，相信你在学完本书的内容之后，能掌握 VHDL 设计的基本要领，并能够进行门数在 1～5 万门的 FPGA设计工作。

第2章

VHDL 入门

2.1 VHDL 程序结构

一个完整的 VHDL 程序的结构以及各部分说明如图 2-1 所示。

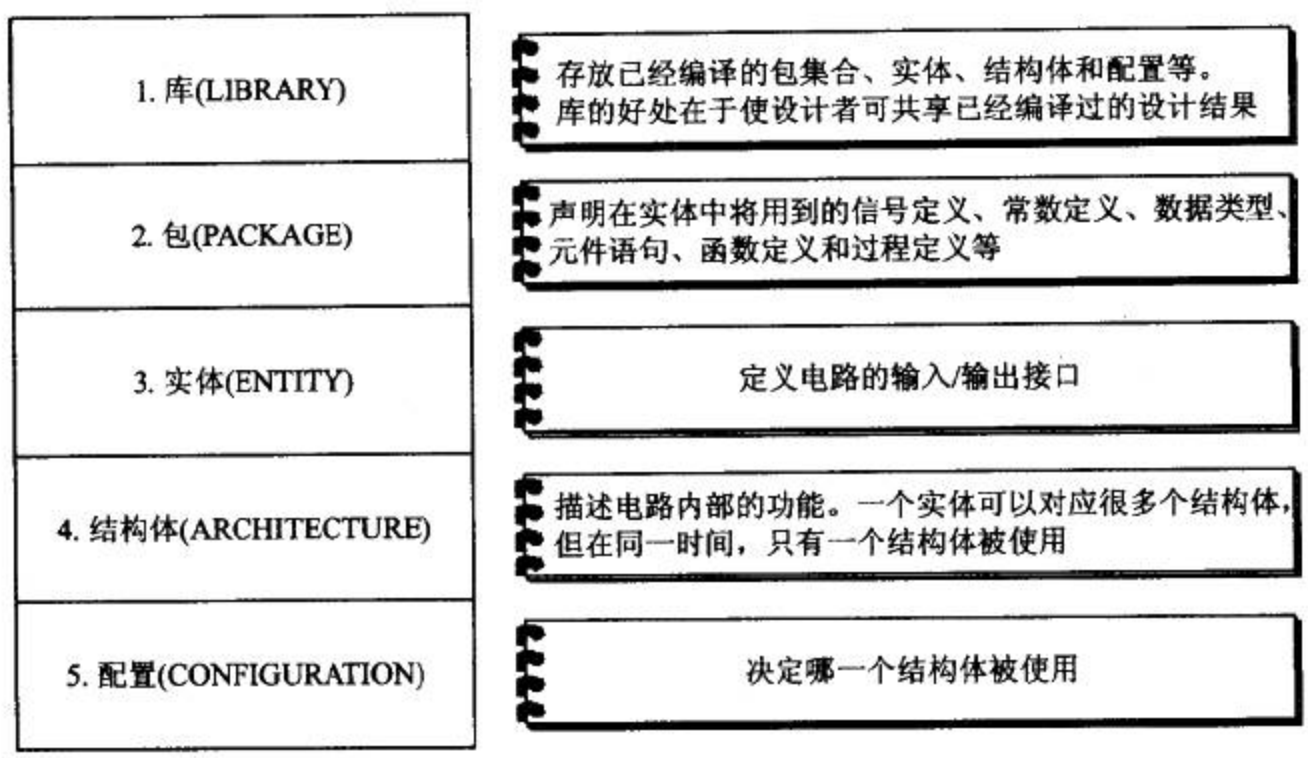


图 2-1 VHDL 程序结构

第2章

VHDL入门

2.1 VHDL程序结构

一个完整的VHDL程序的结构以及各部分说明如图2-1所示。

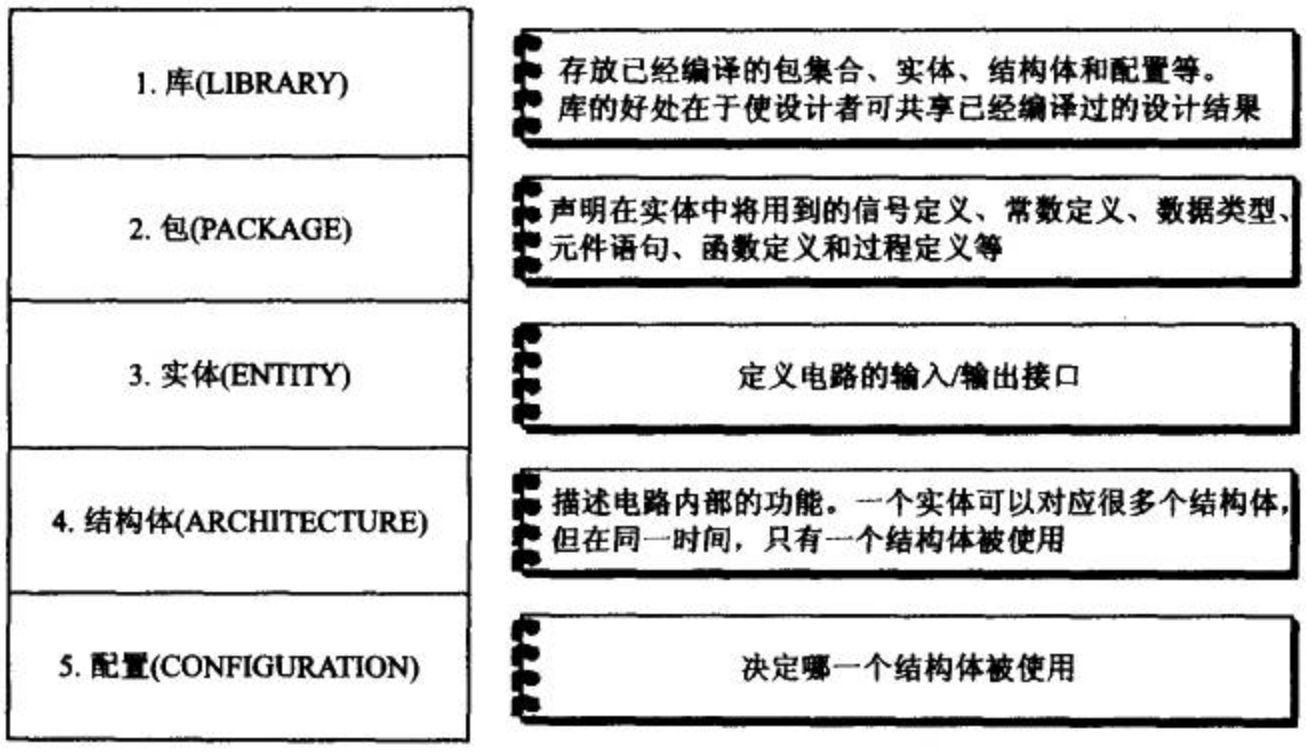


图2-1 VHDL程序结构

响程序的可读性，而且常常会影响整个程序的编译。因此，建议读者在写程序注释时，尽量用英文(本书中所有例程的注释原为英文，考虑到读者的习惯，因此在编入书中时都译成了中文)。

下面，将详细讲解 VHDL 程序基本结构的各组成部分。为了让读者容易理解这些内容，讲解时并不完全按照图 2-2 的顺序，而是从最重要和最简单的概念开始。

2.1.1 实 体

实体(ENTITY)是 VHDL 设计中最基本的组成部分之一。实体用于定义电路的输入/输出引脚，但并不描述电路的具体构造和实现的功能。

例如，例程 2-1 的实体声明(Entity Declaration)对应的是图 2-3 中的电路外观图。从这个外观图只能看出这个实体有一个输入端(Clock)，有一个输出端(Clkout)。但这个电路究竟实现什么功能，却无法从实体声明中得知。

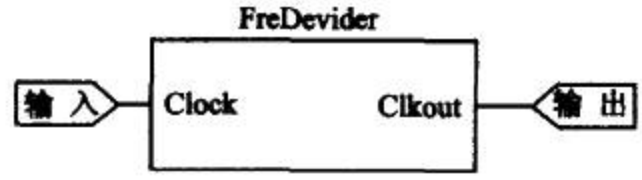


图 2-3 例程 2-1 的实体对应的电路外观图

实体声明的格式是：

```
ENTITY  实体名  IS
[GENERIC (常数名:数据类型:设定值)]  --类属参数说明,"[]"中内容为可选项
PORT
( 端口名 1:端口方向  端口类型;          --端口声明语句用分号隔开
  端口名 2:端口方向  端口类型;
  ⋮
  端口名 n:端口方向  端口类型           --最后一个端口声明语句后不加分号
);
END [实体名];                          --可以只用 END 结束实体声明,不一定加实体名
```

格式说明：

(1) 实体名

实体名必须与文件名相同，否则编译时会出错。

(2) 类属参数

类属参数为实体声明中的可选项，常用来规定端口的大小、信号的定时特性等。本书将在

第5章中介绍GENERIC语句。

(3) 端口名

端口名是设计者赋予每个外部引脚的名称，如例程2-1中，输入引脚的名字是Clock，而输出引脚为Clkout。

(4) 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(或者同时可以作输入与输出)。说明端口方向的标识符以及含义如表2-1所列，端口方向示意图如图2-4所示。

表2-1 端口方向说明

端口方向	含 义
IN	输入
OUT	输出(结构体内部不能读取)
INOUT	双向(输入/输出)
BUFFER	输出(结构体内部可读取)

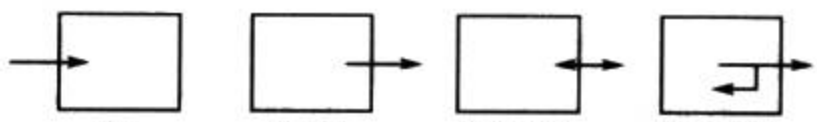


图2-4 端口方向示意图

如果读者一时无法理解OUT与BUFFER的区别也没有关系，在以后的章节中有许多用到BUFFER的例程，可通过这些例子来学习BUFFER的用法。

(5) 端口类型

定义端口的数据类型。常用的数据类型有Std_Logic、Std_Logic_Vector和Integer等。VHDL是一种强类型语言，即对语句中的所有端口信号、内部信号和操作数的数据类型有严格规定，只有相同数据类型的端口信号和操作数才能相互作用。VHDL中常用的数据类型将在2.2节中介绍。

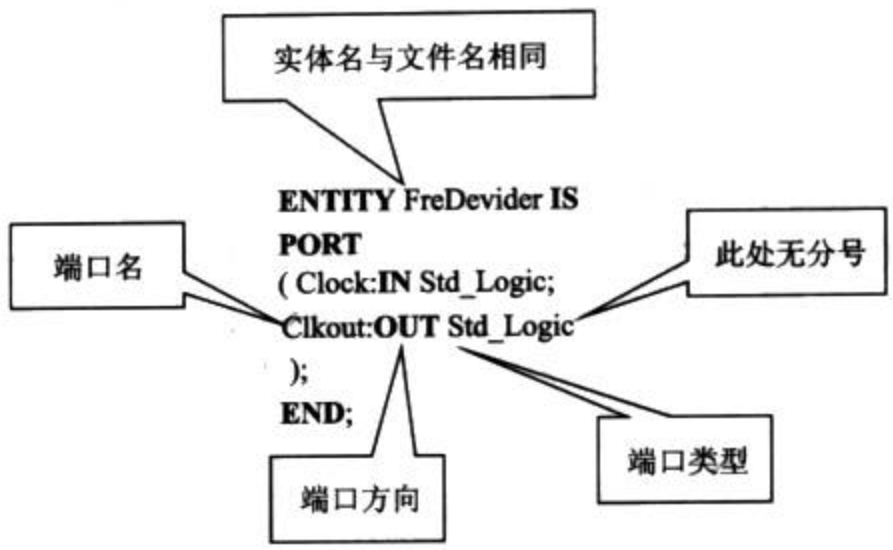
补充说明：

① 在端口说明部分，可将几个同样方向、同样类型的端口放在同一个说明语句里。例如，端口A和端口B都是输入端口，且数据类型都是Std_Logic，那么可用下面这种格式来定义端口A和端口B：

```
A,B:IN Std_Logic;
```

② 最后一条端口声明语句后不可加分号。

下面是例程2-1的实体部分与实体声明格式的对应关系，望能藉此牢记实体声明的格式。



2.1.2 结构体

结构体描述实体内部的结构或功能。一个实体可对应多个结构体，每个结构分别代表该实体功能的不同实现方案或不同描述方式。在同一时刻，只有一个结构体起作用，可以通过配置(CONFIGURATION)来决定使用哪一个结构体进行仿真或综合。

结构体的语法格式如下：

```
ARCHITECTURE 结构体名 OF 实体名 IS
[声明语句]
BEGIN
功能描述语句
END [结构体名];
```

实体名必须与实体声明部分所取的名字相同，而结构体名则可由设计者自由选择，但当一个实体具有多个结构体时，各结构体的取名不可相同。

声明语句用于声明该结构体将用到的信号、数据类型、常数、子程序和元件等。值得注意的是，在一个结构体内声明的数据类型、常数、子程序(包括函数和过程)和元件只能用于该结构体中。如果希望在其他的实体或结构体中引用这些定义，那么需要将其作为程序包(PACKAGE)来处理。

功能描述语句具体描述了结构体的功能和行为。功能描述语句可含有 5 种不同类型的以并行方式工作的语句结构，这几个语句结构又被称为结构体的子结构。各子结构的作用如图 2－5所示。

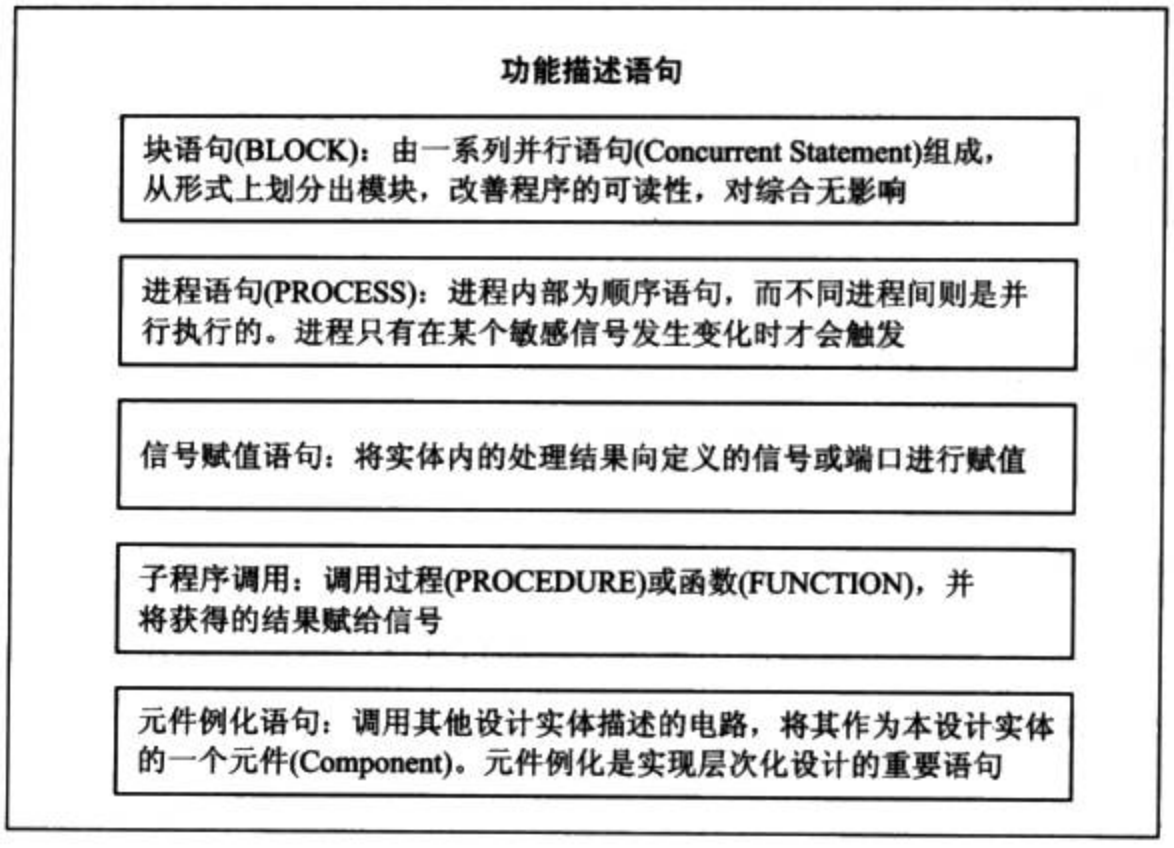


图 2－5　功能描述语句结构

作者并不建议读者花很多时间来背这些概念，因为这将使学习过程变得枯燥乏味，甚至痛苦。读者此刻只需要对这几个子结构有一个基本的认识即可，这些子结构并不都很常用，而常用的子结构会在以后的章节中多次出现，所以只须跟着本书循序渐进地学习，就可轻松地掌握它们。但此处要特别提醒读者，进程语句(PROCESS)在 VHDL 程序中起着不可或缺的重要作用，以后遇到关于进程的讲解，务必要用心学习，此时仅须记住：① 进程内部为顺序语句；② 只有在某个敏感信号发生变化时，进程才会被执行。

例程 2－1 的结构体及其各部分说明如下：

```
ARCHITECTURE Behavior OF FreDevider IS

SIGNAL Clk:Std_Logic;                    --信号的声明

BEGIN
  PROCESS(Clock)
```

```
BEGIN
    IF rising_edge(Clock) THEN
        Clk<=NOT Clk;
    END IF;
END PROCESS;
Clkout<=Clk;                    --信号赋值语句
END;
```

顺序语句 进程 功能描述语句

2.1.3 库与包的调用

当需要引用一个库时，首先要对库名进行说明，其格式为：

```
LIBRARY 库名;    --如例程2-1中的"LIBRARY IEEE;"即调用IEEE标准库
```

对库名进行说明后，就可以使用库中已编译好的设计了。而对库中程序包的访问，则必须通用USE语句实现，其格式为：

```
USE 库名.程序包名.项目名;--如例程2-1中的USE IEEE.Std_Logic_1164.ALL;
```

其中，关键字ALL表示本设计实体可引用此程序包中的所有资源。

初学者往往搞不清楚什么时候应该调用什么程序包，但这并不会给编程造成很大影响。一般来说，IEEE库中的3个程序包Std_Logic_1164、Std_Logic_Arith、Std_Logic_Unsigned足以应付大部分的VHDL程序设计。调用库和程序的语句本身在综合时并不消耗更多资源，因此，初学者不妨在每个设计中将这3个程序包的调用语句都写上，即在每个程序的开始处写上如下代码：

```
Library IEEE;
Use IEEE.Std_Logic_1164.All;
Use IEEE.Std_Logic_Arith.All;
Use IEEE.Std_Logic_Unsigned.All;
```

关于程序包的使用，将在第5章中深入讨论，目前只须掌握本节中的内容。

2.2 VHDL基本数据类型

与C语言不同，VHDL是一种强类型语言(Strong Typed Language)。也就是说，VHDL对每个常数、变量、信号等的数据类型都有严格要求，只有相同数据类型的量，才能互相传递。

“事物都具有两面性”这一老生常谈，用在VHDL的强类型特征上是十分贴切的。强类型的好处是，它使程序更可靠也更容易调试，因为在它的“监控”下，很难犯一些低级错误，例如进行错误的类型赋值，或者大小超过范围的赋值。另一方面，强类型的“死板”有时也很令人厌烦。即使是一些简单的操作，也必须调用类型转换函数才能完成(在第12章中会遇到这样的例子，此处不再举例)。

2.2.1 VHDL预定义数据类型

VHDL中常见的预定义数据类型及其简要说明如表2-2所列。

表2-2 常见的预定义数据类型及说明

数据类型	简要说明
布尔量(Boolean)	取值为FALSE(伪)和TRUE(真)，用于逻辑(关系)运算
位(Bit)	取值为0和1，用于逻辑运算
位矢量(Bit_Vector)	基于Bit类型的数组，用于逻辑运算
整数(Integer)	整数的取值范围是$-(2^{31}-1)\sim(2^{31}-1)$，可用32位有符号的二进制数表示，用于数值运算
实数(Real)	实数的取值范围是-1.0E38～+1.0E38，仅用于仿真，不可综合
时间(Time)	完整的时间类型包括整数和物理量单位两部分，整数与单位之间至少留1个空格，如20 ms、30 μs等。整数部分取值范围与Integer相同。此类型仅用于仿真，不可综合

1. 布尔数据类型

布尔(Boolean)数据类型实际上是一个二值枚举型数据类型，取值为FALSE和TRUE。这里先来看一个关于Boolean类型的小程序(例程2-2)，程序对应的仿真图如图2-6所示。

例程2-2 关于Boolean类型的小程序

```
ENTITY TEST_Boolean IS
PORT
(   A:IN Bit;
```

```
    B,C:OUT Boolean
);
END;

ARCHITECTURE TEST OF TEST_Boolean IS
BEGIN
  B<=True WHEN A='1' ELSE          --A=1 时,B 为 TURE,否则为 FALSE
     False;
  C<=False WHEN A='1' ELSE         --A=1 时,C 为 FLASE,否则为 TRUE
     True;
END;
```

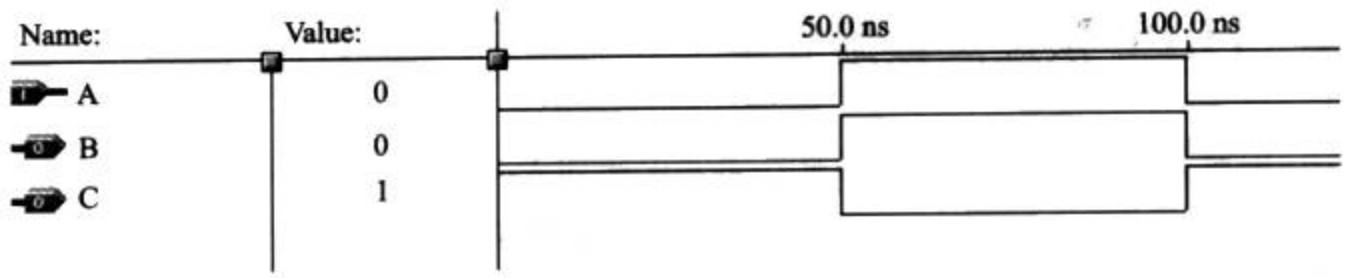


图 2-6　例程 2-2 的仿真波形

从图中可以看出,实际上 FALSE 对应的是逻辑 0,而 TRUE 对应的是逻辑 1,即综合器用一个二进制位(Bit)来表示 Boolean 型变量或信号。由于 VHDL 是强类型语言,即使 Boolean 与 Bit 在本质上是一样的两种类型的变量、信号之间也不可直接赋值。

在例程 2-2 中,为了便于说明问题,将输出引脚赋予 Boolean 类型。在实际的设计中,一般不会将与外界的接口设为 Boolean 类型,Boolean 类型通常以“隐含定义”的方式出现在程序中,如例程 2-3。

例程 2-3　隐式定义的 Boolean 类型

```
ENTITY TEST_Boolean IS
PORT
( A,B:IN Integer RANGE 0 TO 7;
  C:OUT Integer RANGE 0 TO 7
);
END;

ARCHITECTURE TEST OF TEST_Boolean IS
BEGIN
```

```
    PROCESS(A,B)
    BEGIN
      IF (A>B) THEN              --当A>B时,将A值赋给C,否则将B值赋给C
        C<=A;
      ELSE
        C<=B;
      END IF;
    END PROCESS;
END;
```

在例程2-3中,表面上没有出现Boolean类型的数据,其实IF语句中的关系运算表达式(A>B)的结果就是Boolean值。当A>B时,此关系式的值为TRUE,否则为FALSE。

2. 位数据类型

位(Bit)与布尔一样,同属二值枚举型数据类型。取值为0或者1。对应于实际电路中的低电平与高电平。Bit类型的数据对象可以进行“与”、“或”、“非”等逻辑运算,结果仍为Bit类型。

3. 位矢量数据类型

位矢量(Bit_Vector)是基于位类型的数组。使用Bit_Vector时,必须注明数组中的元素个数和排列方向。例如:

```
SIGNAL a:Bit_Vector(0 TO 7);
```

信号a被定义成一个具有8个元素的数组,而且它的最高位为a(0),而最低位为a(7)。

若希望这个数组的排列符合日常使用的顺序,即最高位为a(7),而最低位为a(0),则应将该信号声明语句改写成:

```
SIGNAL a:Bit_Vector(7 DOWNTO 0);
```

关键字TO表示数组从左到右是升序排列,而DOWNTO则是降序排列。

4. 整数数据类型

整数(Integer)类型的数包括正整数、负整数和零。在VHDL中,整数的取值范围为$-(2^{31}-1)\sim(2^{31}-1)$。Integer类型的数常用于加、减、乘、除四则运算(除法运算对运算数有特殊要求,2.4.1小节会给出解释)。在使用整数时,必须用RANGE…TO…限定整数的范围,综合器将根据所限定的范围来决定此信号或变量的二进制数的位数。若所设计的整数范围包括负数,则该数将以二进制补码的形式出现(关于二进制补码的基本概念和计算方法,在

任何一本数字逻辑设计的教材中都能找到)。

例程 2-4 给出一个简单实现整数加、减运算的程序,图 2-7 为对应的仿真波形。

例程 2-4　实现整数加、减运算程序

```
ENTITY TEST_Integer IS
PORT
( A,B:IN Integer RANGE -16 TO 15;          --用 RANGE…TO…限定整数范围
  C,D:OUT Integer RANGE -32 TO 31
);
END;

ARCHITECTURE TEST OF TEST_Integer IS
BEGIN
  C<=A-B;
  D<=A+B;
END;
```

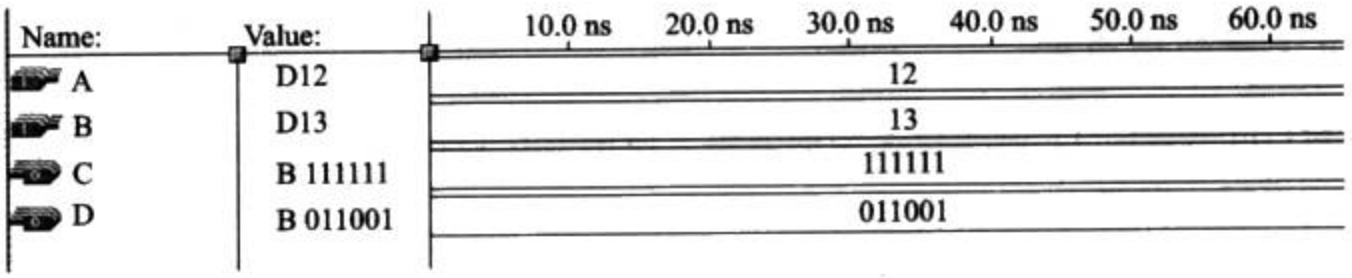


图 2-7　例程 2-4 对应的仿真波形

从图 2-7 可以看出,12−13 的结果为 111111(二进制),而 12+13 的结果为 011001(二进制)。根据补码的计算规则,111111 为−1 的补码,而 011001 则是 25 的补码(亦是原码,正数的补码形式与原码同)。读者若有兴趣,则可修改例程 2-4 中整数的范围,看看仿真结果是如何随整数范围的变化而变化的,这样可以对整数的二进制表示法有一个较深刻的认识。本书中所有整数范围均为非负数,因此,这里就不对数制问题多加讨论了。

5. 其他数据类型

VHDL 的预定义类型还有错误等级、实数和时间等。但这些数据类型只用于仿真,不可被综合。而本书主要讨论的是可综合的 VHDL 语句,且本书中未用到上述数据类型,所以此处略去不表(请读者不必担心学的知识不够多,本书略去的内容都是初学者基本上不可能用到的)。

2.2.2 IEEE预定义标准逻辑位与矢量

在IEEE库的程序包Std_Logic_1164中，定义了两个十分重要的数据类型，即标准逻辑位Std_Logic和标准逻辑矢量Std_Logic_Vector。

1. 标准逻辑位数据类型

标准逻辑位(Std_Logic)数据类型共定义了9种信号状态，如表2-3所列。

表2-3 Std_Logic中的信号值及其定义

信号值	定 义
U	Uninitialized，未初始化的
X	Forcing Unkown，强未知的
0	Forcing 0，强0
1	Forcing 1，强1
Z	High Impedance，高阻态
W	Weak Unknown，弱未知的
L	Weak 0，弱0
H	Weak 1，弱1
—	Don't care，忽略

从表中可以看出，Std_Logic的信号定义比Bit类型对数字电路的逻辑特性描述更完整，更真实。Std_Logic中的X态和Z态可以使设计者模拟一些未知的和高阻态的线路情况，“—”态常用于一些Boolean表达式的化简。但就综合而言，只有4种状态可被综合，即0、1、“—”和Z。其他态虽然不可综合，但对行为仿真仍有十分重要的意义。

本书仅在三态缓冲器的设计中用到了高阻态Z，其他情况下只使用0和1。目前，在设计中一般只使用Std_Logic类型，而很少使用Bit类型，因此本书中所有例程(除讲解Boolean类型的例程2-2外)，对于逻辑信号的定义，均采用Std_Logic类型。

2. 标准逻辑位矢量数据类型

标准逻辑位矢量(Std_Logic_Vector)是基于Std_Logic类型的数组。简而言之，Std_Logic_Vector和Std_Logic的关系就像Bit_Vector与Bit的关系。读者复习一下2.2.1小节中关于Bit_Vector的内容，就很容易理解Std_Logic_Vector的用法了。

需要强调的是，使用Std_Logic和Std_Logic_Vector时，一定要调用IEEE库中的

Std_Logic_1164的程序包。关于库和程序包的调用方法，请读者参考2.1.3小节，这里不再给出例程。

2.2.3 用户自定义的数据类型

用户自定义的数据类型主要有枚举类型(Enumerated Types)和数组类型(Array Types)等，前者常用于状态机描述，而后者常用于ROM和RAM的描述等。

1. 枚举类型

枚举类型的语法格式如下：

```
TYPE 数据类型名 IS (元素1,元素2,…);
```

在状态机描述中，常常使用枚举类型为每一状态命名(本来表征状态的是二进制数)，使程序更具可读性。例如：

```
TYPE State_type IS (Start,Step1,Step2,Final);
SIGNAL State:State_type;
```

上面这个例子为状态机定义了4个状态：Start、Step1、Step2、Final。表征当前状态的信号State就在这4个状态中取值。有关状态机的设计，将在第3章中论述。

2. 数组类型

数组类型常用于组合同样数据类型的元素，其语法格式如下：

```
TYPE 数组名 IS ARRAY(范围) OF 数据类型;
```

下面是几个数组定义的例子：

```
TYPE Byte IS Array (7 DOWNTO 0) OF Bit;          --1 Byte=8 Bits
TYPE Word IS Array (31 DOWNTO 0) OF Bit;         --1 Word=32 Bits
```

2.3 VHDL数据对象

在VHDL中，数据对象(Data Objects)有3类：信号(Signal)、变量(Variable)和常量(Constant)。

VHDL中的变量和常量与软件高级语言中的变量和常量相似，而信号则具有更多的硬件特征，是硬件描述语言所特有的数据对象。

2.3.1 信 号

信号是用来描述实体内部节点的重要数据类型。这里很难给信号下一个完整的定义，因为它已经成为VHDL语言中最基本的概念之一。图2-8可以帮助读者理解信号的概念和其在系统中的作用。

从图2-8中可以看出，信号(Signal)与端口(Port)之间的相似之处和差异点。信号与端口都描述了电路中实际存在的节点(Node)，只是信号描述的是实体内部的节点，而端口则描述实体与外界的接口。在语法上，信号的声明与端口的声明很相似，下面是信号声明的语法格式：

```
SIGNAL 信号名：数据类型[:=初始值];      --初始值仅在仿真时有意义，综合时将忽略此值
```

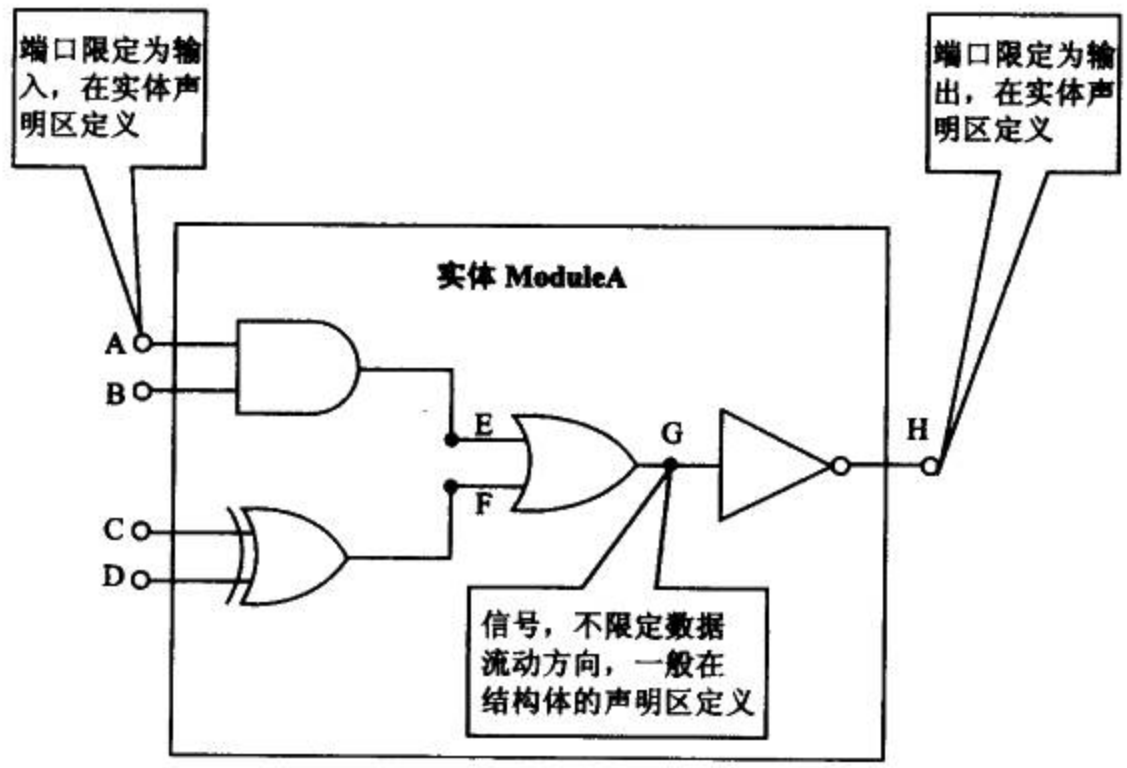


图2-8 信号与端口的区别

对比信号声明与端口声明的格式可以发现，除了端口声明中规定的方向之外，二者无任何差别(虽然信号声明比端口声明多了初始值的赋值，但是这一赋值仅在仿真时有意义，综合器会忽略这一赋值。因此在实际应用中，基本不使用初始值赋值语句)。换句话说，可以将信号理解为“实体内部不限定数据流动方向的端口”，或者将端口理解为“限定数据流动方向的信

号”。因此，信号赋值语句同样适用于端口。

信号赋值语句的格式如下：

```
目标信号名<=表达式
```

图 2-8 对应的 VHDL 程序如例程 2-5 所示。

例程 2-5　图 2-8 对应的 VHDL 程序

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

ENTITY ModuleA IS
PORT
(A,B,C,D:IN Std_Logic;
  H:OUT Std_Logic
);
END;

ARCHITECTURE Dataflow OF ModuleA IS
SIGNAL E,F,G:Std_Logic;
BEGIN
  E<=A AND B;          --方向为 IN 的端口只能出现在赋值符号“<=”的右边
  F<=C XOR D;
  G<=E OR F;           --信号既可以出现在赋值符号“<=”右边，也可以出现在其左边
  H<=NOT G;            --方向为 OUT 的端口只能出现在赋值符号“<=”的左边
END;
```

信号赋值语句同样适用于位矢量(Bit_Vector)和标准逻辑矢量(Std_Logic_Vector)，只要赋值符号左、右两边的位数相同即可，见例程 2-6。

例程 2-6　矢量信号赋值

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

ENTITY Assignment IS
PORT
( A:IN Std_Logic_Vector(7 DOWNTO 0);
  B:IN Std_Logic_Vector(0 TO 7);
```

```
    C:OUT Std_Logic_Vector(0 TO 7)
);
END;

ARCHITECTURE Dataflow OF Assignment IS
SIGNAL Temp:Std_Logic_Vector(7 DOWNTO 0);
BEGIN
    Temp(3 DOWNTO 0)<=A(7 DOWNTO 4);
    Temp(7 DOWNTO 4)<=B(0 TO 3);
    C<=NOT Temp;                    -- 若两个矢量的位数相同,则整体赋值时可不必限定范围
END;
```

例程 2-6 中矢量信号赋值情况如图 2-9 所示。读者将程序与示意图进行比较后，应能了解矢量赋值的特点，也能对升序(…TO…)和降序(…DOWNTO…)排列的概念有更深刻的理解。

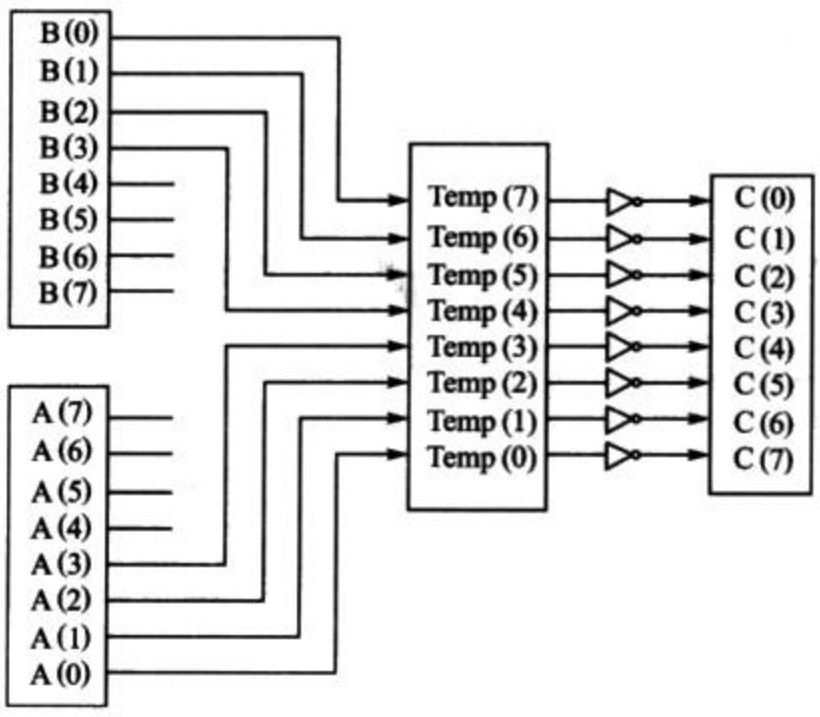


图 2-9　例程 2-6 中矢量信号赋值示意图

最后，要特别强调指出，信号的赋值具有“非立即性”，即会有延时。这与实际硬件的传播延迟特性十分吻合。初学者可能难以理解信号赋值的延时特性，这没关系，因为以后会常常和信号打交道，用久了自然就能摸清它的路数了。在第 6 章中，对信号这个极具硬件特色的数据对象再作研讨。

2.3.2 变 量

变量只能在进程和子程序中使用，主要用于描述算法和方便程序中的数值运算。

定义变量的语法格式如下：

```
VARIABLE 变量名：数据类型[:=初始值];
```

定义变量与定义信号的语法格式十分相似，只是将关键字 SIGNAL 变成 VARIABLE。与信号一样，变量的初始值赋值只在仿真中有用，综合时将被忽略，因此在实际应用中很少对变量赋初值。虽然二者语法格式十分相似，但在程序中的位置却不同，下面这个例子比较了变量与信号声明语句的所在位置：

```
ARCHITECTURE …OF… IS
SIGNAL 信号名 1:数据类型;          --信号的声明在结构体内，进程外部
 ⋮
SIGNAL 信号名 n:数据类型;
BEGIN
  PROCESS(…)
  VARIABLE 变量名 1: 数据类型;   --变量的声明在进程内部，进程中的 BEGIN 之前
   ⋮
  VARIABLE 变量名 n: 数据类型;
  BEGIN
   ⋮
  END PROCESS;
END;
```

变量赋值语句的语法格式如下：

```
目标变量名:=表达式;
```

表达式可以是一个数值，也可以是一个与目标变量数据类型相同的变量，或者是运算表达式。例如：

```
PROCESS(…)
VARIABLE a,b,c: Integer RANGE 0 TO 31;
BEGIN
   ⋮
```

```
    a:=5;          --表达式为数值
    b:=a;          --表达式为与目标变量数据类型相同的变量
    c:=a+b+2;      --表达式为运算表达式
     :
END;
```

变量与信号的区别不仅仅在于声明与赋值语句的格式，最重要的区别在于信号与实际电路的某个节点或信号线对应，因为硬件具有传播延迟特性，所以信号的赋值具有延时特性；而变量是一个抽象的值，它不与任何实际电路连线对应，因此它的赋值是立即生效的。

注意：变量不与任何实际电路连线对应，这并不代表变量赋值行为不产生与之对应的硬件结构。变量赋值语句既然是可综合的，就一定会对硬件结构产生影响。

在实际应用中，信号的行为更接近硬件的实际情况，因此将更多使用信号进行电路内部数据传递。只有在描述一些算法时，才用到变量。当然，有些情况下（如作矢量的索引值等）只能使用变量，在“解惑篇”中将会对变量和信号的应用范围做更详细的讨论。此时，读者只须记住信号赋值的“非即时性”与变量赋值的“即时性”之间的区别即可。

2.3.3 常 数

VHDL中的常数（Constant）与软件高级语言中的常数十分相似，作用如下：

- 保证该常数描述的那部分数据在程序中不会因误操作被改变；
- 对程序中的某些关键数值进行命名，可以提高程序的可读性；
- 将出现次数较多的关键数值用常数表示，可以使程序易于修改：只须修改常数就可以替换所有相关数值。

定义常数的语法格式如下：

```
CONSTANT 常数名：数据类型：=设置值
```

例程2-7为偶数倍分频电路的VHDL程序。分频倍数为$2\times(N+1)$，要改变分频倍数，只须修改常数N即可。图2-10为例程2-7的仿真波形，读者可修改程序中的N值，观看仿真波形结果变化。

例程2-7 偶数倍分频电路程序

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

```
ENTITY FreDevider IS
PORT
  (Clkin: IN Std_Logic;
  Clkout:OUT Std_Logic
  );
END;

ARCHITECTURE Devider OF FreDevider IS
CONSTANT N:Integer:=3;                          --定义常数 N
SIGNAL Counter:Integer RANGE 0 TO N;            --引用常数 N
SIGNAL Clk:Std_Logic;
BEGIN
  PROCESS(Clkin)
  BEGIN
    IF rising_edge(Clkin) THEN
      IF Counter=N THEN                         --第 2 次引用常数 N
        Counter<=0;
        Clk<=Not Clk;
      ELSE
        Counter<=Counter+1;
      END IF;
    END IF;
  END PROCESS;
  Clkout<=Clk;
END;
```

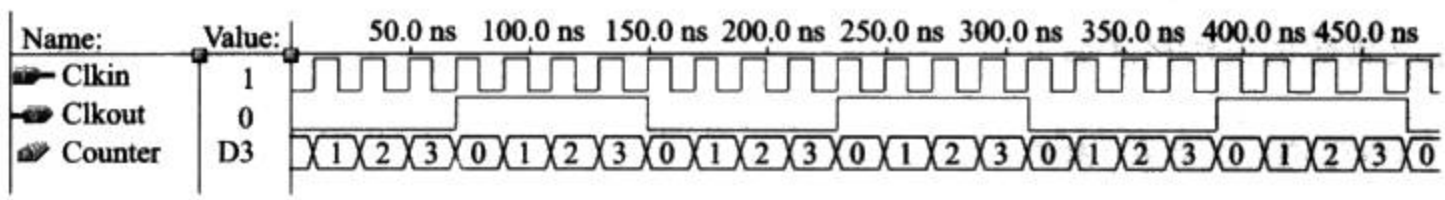


图 2-10　例程 2-7 的仿真波形

2.4　VHDL 运算符

VHDL 的运算符主要有 4 种：算术运算符、并置运算符、关系运算符和逻辑运算符。

2.4.1 算术运算符

VHDL预定义的常见算术运算符及其说明如表2-4所列。

表2-4 常见算术运算符及其说明

运算符	含 义	备 注
+	加	一般情况下,"+"号两边只能是整型信号(变量)。但若事先调用了IEEE库中的Std_Logic_1164和Std_Logic_Unsigned(或Std_Logic_Signed)程序包,则"+"号两边可以是:① Std_Logic_Vector + Std_Logic_Vector; ② Std_Logic_Vector + Integer; ③ Integer + Std_Logic_Vector; ④ Integer + Integer
−	减	同上
*	乘	一般情况下,"*"号两边只能是整型信号(变量)。但若事先调用IEEE库中的Std_Logic_1164和Std_Logic_Unsigned(或Std_Logic_Signed)程序包,则"*"两边可以是:① Std_Logic_Vector * Std_Logic_Vector; ② Integer * Integer **注意**:Std_Logic_Vector与Integer的混合乘法是不被支持的
/	除	MAX+plus II要求"/"号右边的数为2的N次幂(2^N)
* *	乘方	MAX+plus II不支持
MOD	求模	MAX+plus II不支持,一般综合器会要求MOD右边的数为2的N次幂
REM	求余	MAX+plus II不支持,一般综合器会要求REM右边的数为2的N次幂
ABS	求绝对值	MAX+plus II不支持

2.4.2 并置运算符

并置运算符"&"用于将多个元素或矢量连接成新的矢量。例如:

```
Signal A:Std_Logic_Vector(3 Downto 0);
Signal B:Std_Logic_Vector(1 Downto 0);
Signal C:Std_Logic_Vector(5 Downto 0);
Signal D:Std_Logic_Vector(4 Downto 0);
Signal E:Std_Logic_Vector(2 Downto 0);
 ⋮
C<=A&B;                                    --矢量与矢量并置
D<=A(1 Downto 0)&B(1 Downto 0)&'1';        --矢量与元素并置
E<=B(0)&A(1)&'0';                          --元素与元素并置
 ⋮
```

上例中矢量*A*、*B*、*C*、*D*的关系如图2-11所示。将此图与上例中的并置语句对比,应很容

易明白并置运算符的使用方法。矢量 *E* 与 *A*、*B* 的关系图留给读者自己推导。

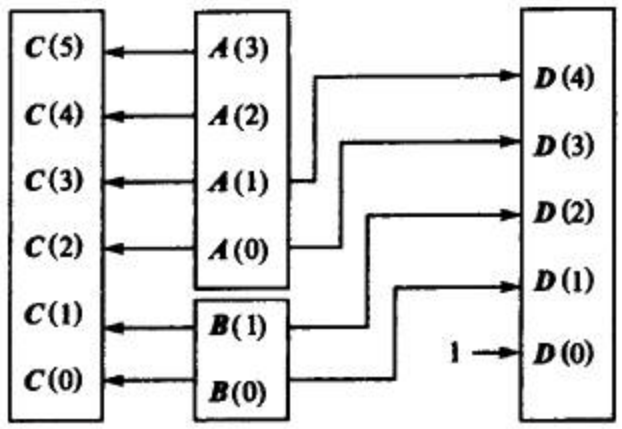


图 2-11 矢量 *A*、*B*、*C*、*D* 的关系图

2.4.3 关系运算符

VHDL 预定义的关系运算符如表 2-5 所列。

表 2-5 关系运算符及其说明

关系运算符	=	/=	<	<=	>	>=
含 义	等于	不等于	小于	小于或等于	大于	大于或等于

关系运算符的作用是，将相同数据类型的数据对象进行数值比较或关系排序判断，并将结果以 Boolean 类型的数据表示，即 TRUE 或 FALSE(参见 2.2.1 小节)。

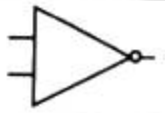
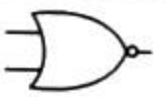
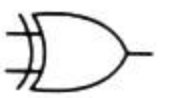
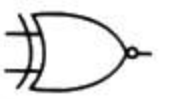
VHDL 规定，“=”和“/=”的操作对象可以是 VHDL 中任何数据类型构成的操作数；其余关系运算符的操作对象，则仅限于整数数据类型、枚举数据类型以及由整数型或枚举型数据类型元素构成的一维数组。但作者建议，初学者在使用除“=”和“/=”之外的关系运算符时，只进行整数数据类型的比较，否则很容易出错。

值得注意的是，“小于或等于”关系运算符“<=”的形式与信号赋值操作符一模一样。判别二者的关键在于其使用环境：在条件语句(如 IF_THEN_ELSE、WHEN 等)中的条件式(即条件判断语句)中出现的“<=”是关系运算符，其他情况则是信号赋值操作符。

2.4.4 逻辑运算符

VHDL 共定义了 7 种逻辑运算符，其含义与对应逻辑门如表 2-6 所列。

表2-6 逻辑运算符及其说明

逻辑运算符	含 义	对应逻辑门
AND	与	
OR	或	
NOT	非	
NAND	与非	
NOR	或非	
XOR	异或	
XNOR	同或	

逻辑操作符的操作对象一般为以下5种数据类型之一：Boolean、Bit、Bit_Vector、Std_Logic和Std_Logic_Vector。

建议：虽然NOT比其他逻辑运算符的优先级高，但为了避免犯错，在写程序时仍应用括号将NOT与其对应的操作数括起来。其他逻辑运算符也应照此处理。例如：

```
A<=B AND (NOT C);
A<=(B AND C) XOR (C AND D);
A<=(NOT (B AND C)) NAND (C XOR D);
```

这样可使整个逻辑表达式层次清楚，提高程序的可读性，同时方便查错。

2.5 VHDL并行语句

并行语句(Concurrent Statements)是硬件描述语言区别于一般软件程序语言的最显著的特点之一。所有并行语句在结构体中的执行都是同时进行的，即它们的执行顺序与语句书写的顺序无关。

所谓“并行”，指的是这些并行语句之间没有执行顺序的先、后之分，但并不意味着并行语

句内部也一定是以并行方式运行的。事实上，并行语句内部的语句运行可以是并行的(如块语句)，也可以是顺序的(如进程)。图 2-12 是一个结构体中各种并行语句运行的示意图。从图中可以看出，“并行执行”并不是“孤立执行”的代名词，事实上，并行语句之间是可以通过信号交换信息的。

对初学 VHDL 的人来说，上面这些说明可能太抽象了。如果读者觉得难以理解，不妨先跳过去，等积累了一定的编程经验后，再回过头来看这段说明，自然就会明白了。

如图 2-12 所示，VHDL 的并行语句主要有以下 6 种：

① 进程语句；
② 并行信号赋值语句；
③ 并行过程调用语句；
④ 元件例化语句；
⑤ 生成语句；
⑥ 块语句。

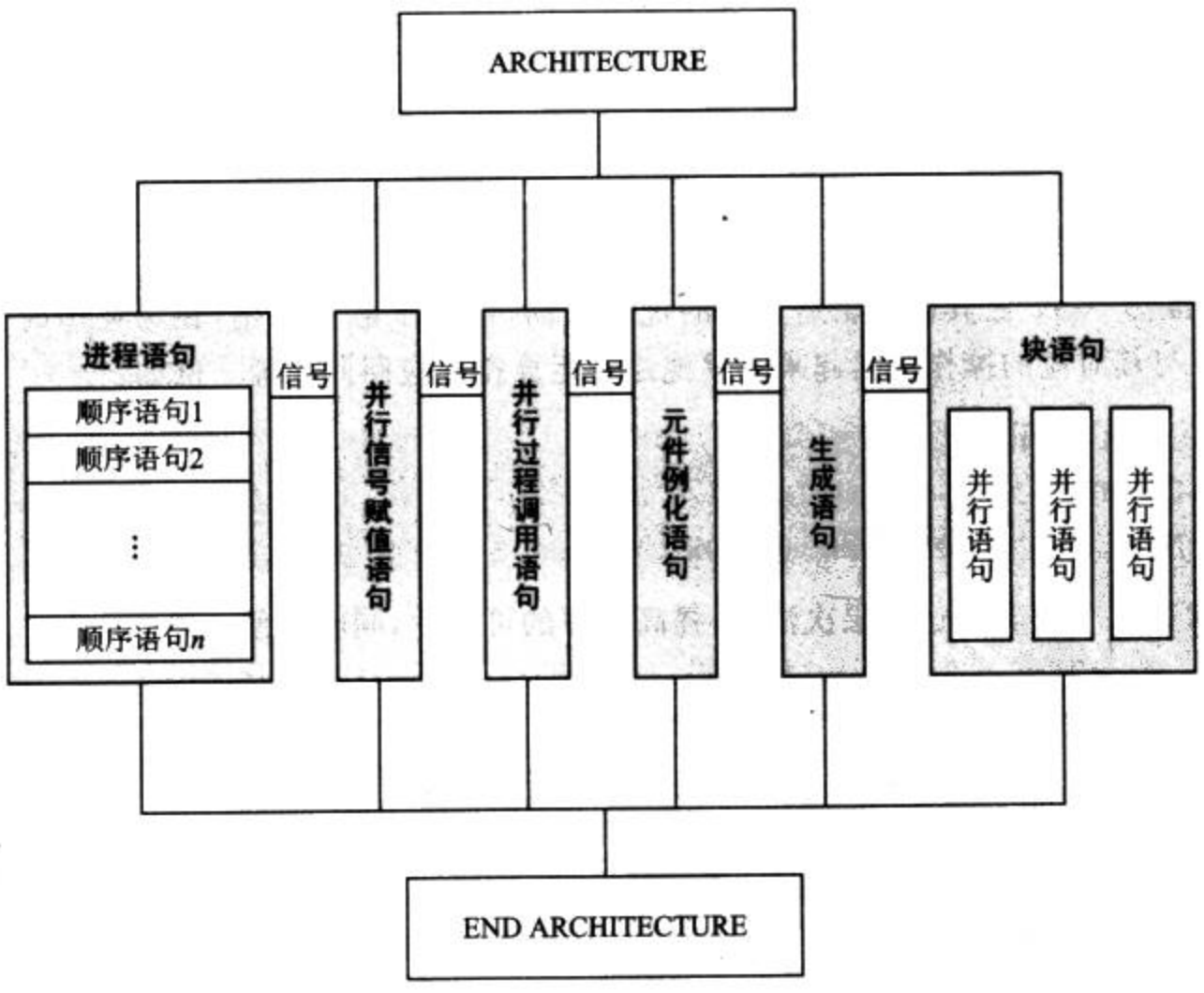


图 2-12 并行语句运行示意图

本节并未覆盖以上所有语句，这是因为：上面这些语句中有的对初学者而言太难(如元件例化语句)，有的则没有太大的实用价值(如块语句)。本节主要讲述并行信号赋值语句和进程语句这两种贯穿本书始末的重要语句，其余语句将在后续章节中讲解。

2.5.1 并行信号赋值语句

并行信号赋值语句又分为以下3种类型：

① 简单信号赋值语句；
② 选择信号赋值语句；
③ 条件信号赋值语句。

这3种信号赋值语句的赋值目标都必须是信号。下面分别介绍这几种并行信号赋值语句。

1. 简单信号赋值语句

简单信号赋值语句已在2.3.1小节中介绍信号概念时介绍过了，它的语句格式如下：

```
目标信号名<=表达式
```

因为VHDL是强类型语言，所以目标信号的数据类型必须与赋值符号“<=”右边表达式的数据类型一致。

前面的章节中已多处出现简单信号赋值语句，此处不再举例。

2. 选择信号赋值语句

选择信号赋值语句的格式如下：

```
WITH 选择表达式 SELECT
  赋值目标信号 <= 表达式1  WHEN 选择值1,
                  表达式2  WHEN 选择值2,
                  ⋮
                  表达式n  WHEN OTHERS;
```

从WITH_SELECT语句的格式不难猜出它的用法：当“选择表达式”等于某一个“选择值”时，就将其对应的表达式的值赋给目标信号；若“选择表达式”与任何一个“选择值”均不相

等,则将 WHEN OTHERS 前的表达式的值赋给目标信号。

由 WITH_SELECT 语句的用法很容易联想到“数据选择器(也称为多路复用器,Multiplexer)”。表 2-7 为一个 8 路 4 选 1 的多路复用器的工作模式,例程 2-8 用 WITH_SELECT 语句描述了这一工作模式,图 2-13 为例程 2-8 的仿真波形。

表 2-7 4X1 多路复用器工作模式

地址选择线 Sel	输出 DOUT
00	Data0
01	Data1
10	Data2
11	Data3

例程 2-8 8 路 4 选 1 复用器

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

ENTITY MUX IS
PORT
( Data0,Data1,Data2,Data3:IN Std_Logic_Vector(7 DOWNTO 0);
  Sel:IN Std_Logic_Vector(1 DOWNTO 0);
  DOUT:OUT Std_Logic_Vector(7 DOWNTO 0)
);
END;

ARCHITECTURE Dataflow OF MUX IS
BEGIN
  WITH Sel SELECT
    DOUT<=Data0 WHEN "00",
          Data1 WHEN "01",
          Data2 WHEN "10",
          Data3 WHEN "11",
          "00000000" WHEN OTHERS;
END;
```

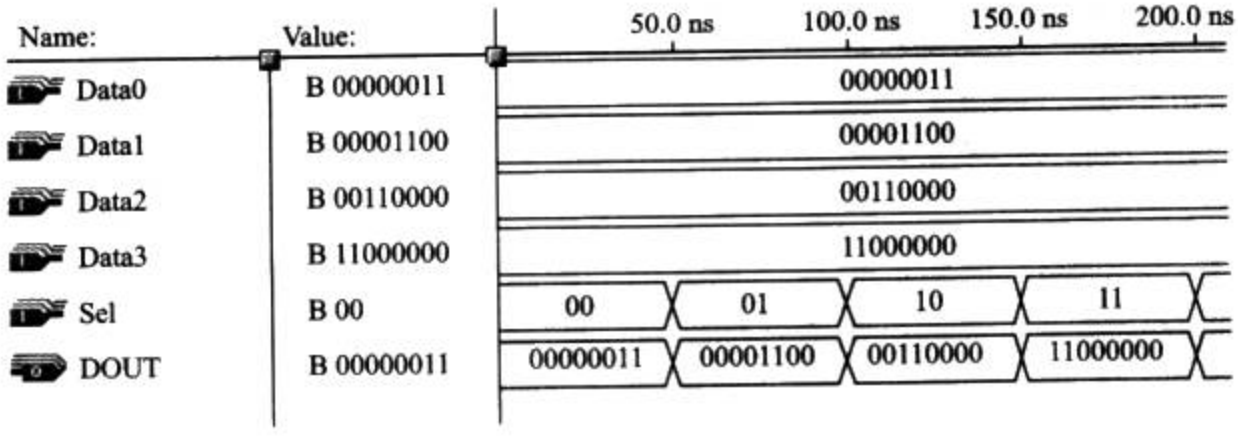


图 2-13 例程 2-8 的仿真波形

从图2-13中可以看出，数据输出端DOUT随着地址线Sel的变化而变化，变化规律与表2-7一致。

读者可能会奇怪，在例程2-8中，已经确定了Sel为00、01、10和11这4种情况时的输出，按理说已将表2-7中的情况完全覆盖了，为什么还要在最后加入WHEN OTHERS指定其他情况的输出呢？还有什么"其他情况"呢？

读者应该还记得，Std_Logic数据类型的预设信号值总共有9种，00、01、10、11这4种选项远远不足以覆盖所有可能的情况，因此要为"其他情况"(即使遇到的可能性几乎为零)的信号赋值指定一个"出口"，否则综合器会提示"Error：choices … not specified(有些选项下没有指定输出)"。类似的情况在以后的学习中还会遇到(如CASE语句等)，所以在此先提个醒：使用Std_Logic数据类型时，如果要覆盖所有可能的情况，那么别忘了OTHERS。

最后总结一下使用WITH_SELECT语句的注意事项：

①"选择值"要覆盖所有可能的情况，若不可能一一指定，则要借助OTHERS为其他情况找一个"出口"；

②"选择值"必须互斥，不能出现条件重复或重叠的情况。

3. 条件信号赋值语句

选择信号赋值语句简单、易用，但它仅对某一特定信号进行选择值的判断(所以叫"选择信号赋值语句")，当需要对较多信号条件进行判断时，它就无能为力了。这时，则需要用到条件信号赋值语句。

条件信号赋值语句的格式如下：

```
赋值目标信号<=表达式1 WHEN 赋值条件1 ELSE
              表达式2 WHEN 赋值条件2 ELSE
               ⋮
              表达式n WHEN 赋值条件n ELSE
              表达式;
```

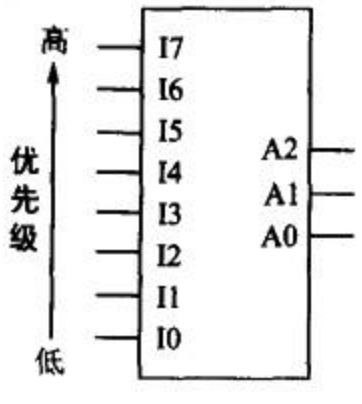


图2-14 8输入优先编码器

在执行WHEN_ELSE语句时，赋值条件按书写的先后顺序逐项测试，一旦发现某一赋值条件得到满足，即将相应表达式的值赋给目标信号，并不再测试下面的赋值条件。换言之，各赋值子句有优先级(Priority)的差别，按书写先后顺序从高到低排列。

说到优先级，很容易想到优先编码器(Priority Encoder)。下面，用WHEN_ELSE语句设计一个简单的优先编码器。图2-14给出了一个简单的8输入优先编码器的元件外观图，

其中输入为 I7～I0(I7 具有最高编码优先级),编码输出为 A2～A0。

例程 2－9 用 WHEN_ELSE 语句实现了一个 8 输入优先编码器,图 2－15 为相应的仿真波形。

例程 2－9　8 输入优先编码器

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

ENTITY Priority_Encoder IS
PORT
(I:IN Std_Logic_Vector(7 DOWNTO 0);
 A:OUT Std_Logic_Vector(2 DOWNTO 0)
);
END;

ARCHITECTURE Dataflow OF Priority_Encoder IS
BEGIN
  A<="111" WHEN I(7)='1' ELSE
   "110" WHEN I(6)='1' ELSE
   "101" WHEN I(5)='1' ELSE
   "100" WHEN I(4)='1' ELSE
   "011" WHEN I(3)='1' ELSE
   "010" WHEN I(2)='1' ELSE
   "001" WHEN I(1)='1' ELSE
   "000" WHEN I(0)='1' ELSE
   "111";
END;
```

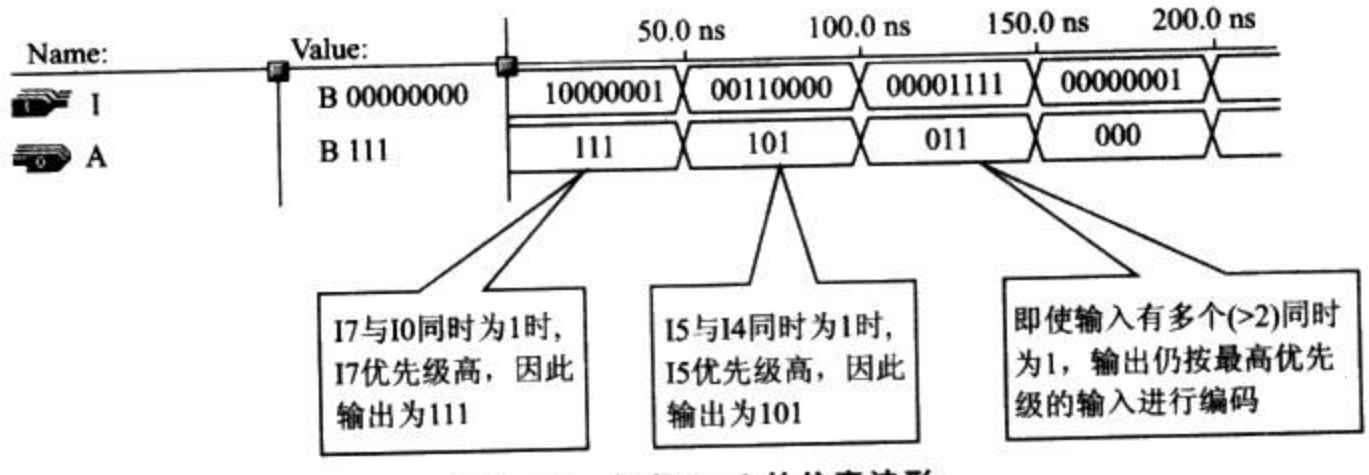


图 2－15　例程 2－9 的仿真波形

从例程2-9和图2-15可得出以下结论：因为WHEN_ELSE语句各赋值子句具有优先级差别，所以各赋值条件可重叠。

2.5.2 进程语句

进程语句PROCESS可以说是VHDL语言中最重要的语句之一，它的特点如下：

- 进程本身是并行语句，但其内部则为顺序语句；
- 进程只有在特定的时刻(敏感信号发生变化)才会被激活。

1. 进程语句的语法格式

PROCESS语句的语法格式有如下两种：

PROCESS语法格式1：

```
[进程标号:] PROCESS (敏感信号参数表)
[声明区];
BEGIN
  顺序语句
END PROCESS [进程标号];
```

PROCESS语法格式2：

```
[进程标号:] PROCESS
[声明区];
BEGIN
  WAIT UNTIL (激活进程的条件);
  顺序语句
END PROCESS [进程标号];
```

上面这两种语法格式是等价的，但一般只采用第1种语法格式，而避免使用WAIT语句(原因请读者参见14章)。本书中所有PROCESS语句均遵从语法格式1。

下面对语法格式1中的各项进行说明：

- 进程标号

简单地说，就是给进程起名。这个标号不是必需的，在大型的多个进程并存的程序中，标号可提高程序的可读性。

● 敏感信号参数表

如前所述，进程只在敏感信号发生变化的情况下被激活，而这些敏感信号就包括在敏感信号参数表中。

注意：一个进程可有多个敏感信号，任一敏感信号发生变化都会激活进程，各敏感信号间以逗号隔开。

● 声明区

定义一些仅在本进程中起作用的局部量，最常在此处定义的是变量。

注意：信号是全局量，不可在此处声明。

● 顺序语句

按书写顺序执行的语句，如 IF_THEN_ELSE 和 CASE 语句。

注意：所谓“顺序执行”是指在仿真意义上具有一定的顺序性（并不意味着这些语句对应的硬件结构也有相同的顺序性），详见 2.6 节。

2. 进程的工作原理

图 2-16 说明了进程工作的基本原理。

从图 2-16 中可知：

① 当进程的敏感信号参数表中的任一敏感信号发生变化时，进程被激活，开始从上而下按顺序执行进程中的顺序语句；当最后一个语句执行完毕，进程挂起，等待下一次敏感信号的变化。从系统上电开始，这个过程就周而复始地进行，就像软件（或单片机程序）中的死循环。

② 虽然进程内部的语句是顺序执行的，但进程与进程之间则是并行的关系。例如，一个 ARCHITECTURE 中有若干个 PROCESS，颠倒各 PROCESS 在程序中的顺序并不会造成仿真与综合结果的改变。

3. 进程与时钟

虽然进程可用来描述组合逻辑电路，但最重要的还是用它来设计时序电路（或是时序电路与组合逻辑电路的综合电路）。对于组合逻辑电路的设计，用前面所提到的一些语句和关系符就可以实现，而时序电路的设计则必须借助 PROCESS 的力量。

如果设计时序电路，就一定要对时钟有所了解，因为大多数时序电路的正常工作依赖于时钟。这里不准备研究时钟的概念及其与时序电路设计的关系，因为这超出了本书的范围，而且这些知识可以在任何一本数字逻辑设计的教科书中找到。这里仅讨论以下两个问题：① 时钟

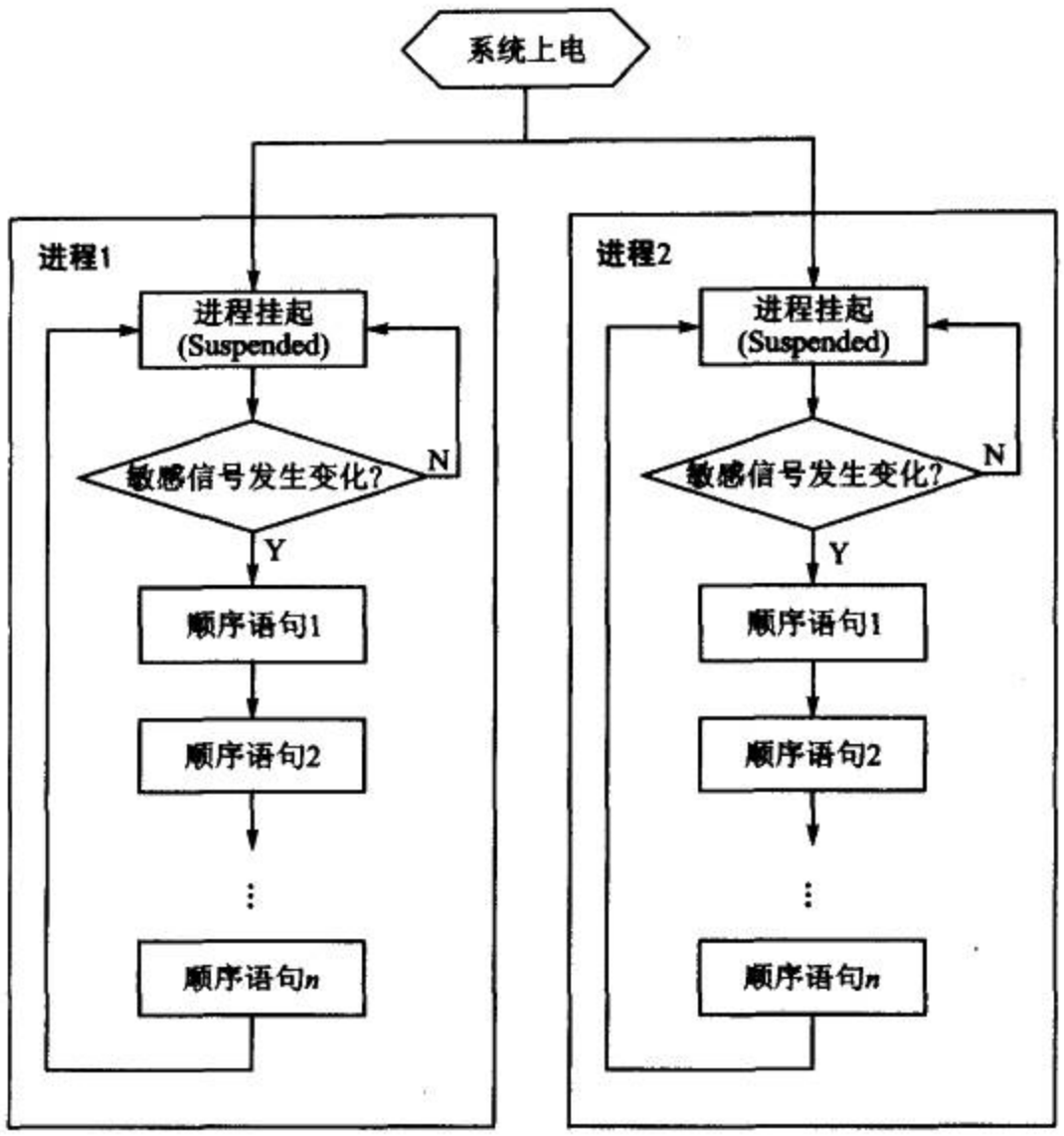


图 2-16　进程的工作原理示意图

与进程的关系；② 时钟沿在 VHDL 中的描述方法。

(1) 时钟与进程的关系

进程是由敏感信号的变化来启动的，因此可将时钟作为敏感信号，用时钟的上升沿或下降沿来驱动进程语句的执行。图 2-17 给出了时钟上升沿驱动进程语句的示意图。

从图 2-17 中可以看出，在每个时钟上升沿，进程都被激活，进程中的语句被执行。有一点要特别强调：是在每个上升沿启动一次进程(执行进程内所有语句)，而不是在每个上升沿执行一条语句。这一点与计算机软件程序和单片机程序有很大差异。

(2) 时钟沿的 VHDL 描述法

假设时钟信号为 Clock，且数据类型为 Std_Logic，则时钟沿在 VHDL 中的描述方法如下：

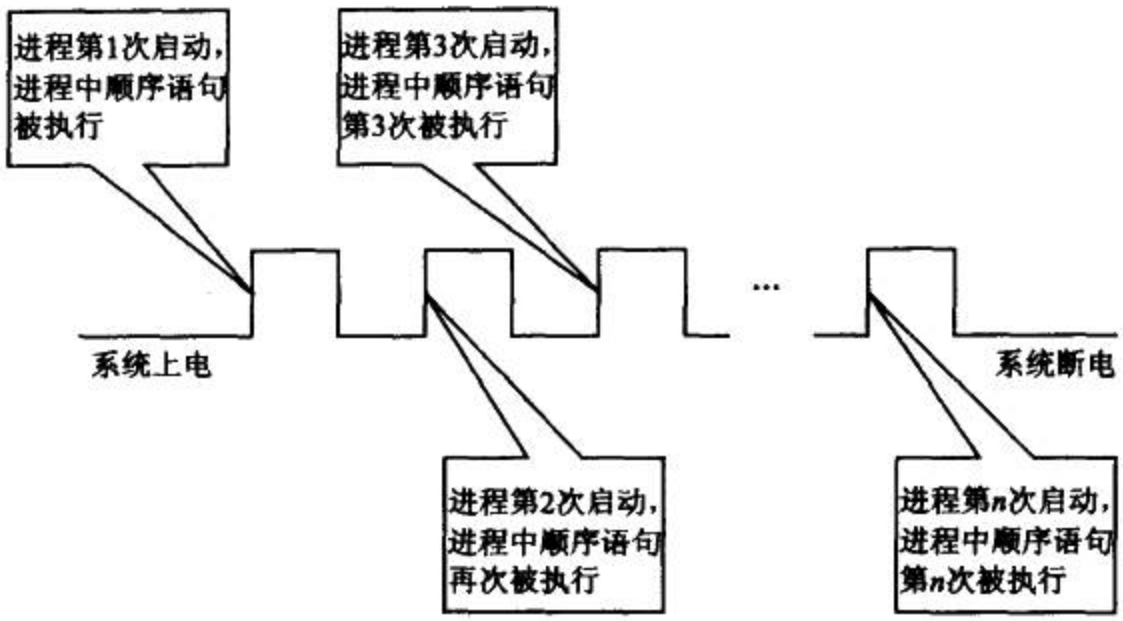


图 2-17 时钟上升沿驱动进程语句示意图

```
上升沿描述：Clock'EVENT AND Clock='1'
下降沿描述：Clock'EVENT AND Clock='0'
```

其中，Clock'EVENT 表示在 Clock 信号上有事件发生(即信号发生变化)。若变化后 Clock 值为'1'，则表示时钟从'0'变成'1'，因此 Clock'EVENT AND Clock='1'就表示时钟上升沿。

除了上面的表示方式外，还有两个预定义的函数来表示时钟沿：

```
上升沿描述：rising_edge(Clock)
下降沿描述：falling_edge(Clock)
```

由于这两个函数比较直观、易懂，本书中所有例程均用这两个函数来描述时钟沿。

4. 进程实例

进程的使用十分灵活，既可以描述时序电路，又可以描述组合逻辑。用进程描述组合逻辑的例子见 2.6.1 小节中的例程 2-12，这里仅给出用进程描述时序电路的例子。

这里给出两个很简单但又极具代表性的进程实例：2 分频电路和计数器。前者无复位信号，而后者有异步复位。请注意这两个程序中进程的格式，因为这是最常用的两种格式，也是本书中大部分进程所遵从的格式。

例程 2-10 为 2 分频电路的 VHDL 程序，其仿真波形见图 2-18。

例程 2-10 2分频电路

```
LIBRARY IEEE;
USE IEEE. Std_Logic_1164. All;

ENTITY FreDevider Is
PORT
( Clock:IN Std_Logic;
  Clkout:OUT Std_Logic
);
END;

ARCHITECTURE Behavior OF FreDevider IS
SIGNAL Clk:Std_Logic;
BEGIN
  PROCESS(Clock)                          --将时钟作为进程的敏感信号
  BEGIN
    IF rising_edge(Clock) THEN            --在时钟上升沿执行 Clk<=NOT Clk
      Clk<=NOT Clk;
    END IF;
  END PROCESS;
  Clkout<=Clk;
END;
```

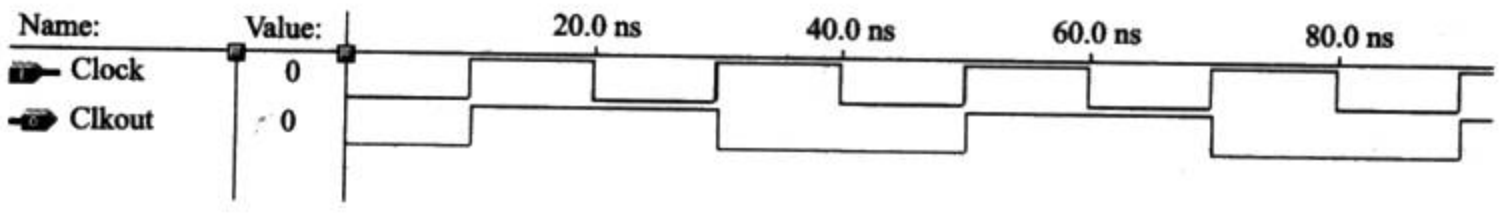


图 2-18 例程 2-10 的仿真波形

例程 2-11 为计数器的 VHDL 程序，相应的仿真波形见图 2-19。

例程 2-11 计数器

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

ENTITY Counter IS
PORT
```

```
( Reset:IN Std_Logic;                    --异步复位信号
  Clock:IN Std_Logic;                    --时钟信号
  Num:BUFFER Integer RANGE 0 TO 3        --计数器输出值
);
END;

ARCHITECTURE Behavior OF Counter IS
BEGIN
  PROCESS(Reset,Clock)                   --RESET 信号必须在敏感信号表中
  BEGIN
    IF Reset='1' THEN                    --注意格式
      Num<=0;                            --复位时 Num 清 0
    ELSIF rising_edge(Clock) THEN
      IF Num=3 THEN                      --如果 Num=3 就清 0
        Num<=0;
      ELSE
        Num<=Num+1;                      --否则自加 1
      END IF;
    END IF;
  END PROCESS;
END;
```

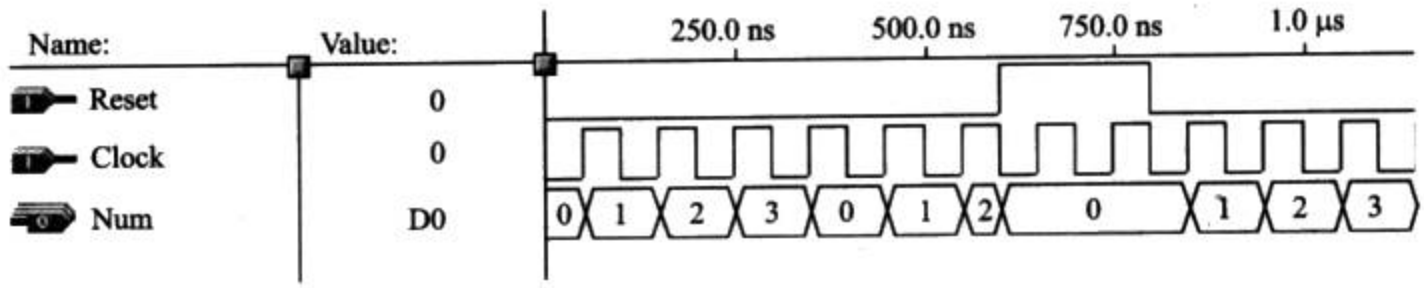


图 2-19　例程 2-11 的仿真波形

关于进程的各种格式及其综合结果的讨论，可参考第 14.2 节，这里就不赘述了(建议初学者不要在这个时候去看较深入的讨论，一下接受太多的信息实在是有害无益，不妨在积累一定的编程经验后，再深入地学习)。

5. 进程要点

进程有如下要点：

① 进程语句(PROCESS)本身是并行语句，但其内部为顺序语句。

② 进程在敏感信号发生变化时被激活。

③ 在同一进程中对同一信号多次赋值，只有最后一次生效。

例如：

```
SIGNAL A,B:Integer RANGE 0 TO 7;
  ⋮
PROCESS(Clock)
BEGIN
  IF rising_edge(Clock) THEN
    ⋮
    B<=A+1;          --忽略
    B<=B+1;
    ⋮
  END IF;
END PROCESS;
```

上例中，在一个进程中对信号进行两次赋值，只有 B<=B+1 是有效的赋值，前一次的赋值被忽略。

④ 在不同进程中，不可对同一信号进行赋值。

⑤ 一个进程不可同时对时钟上、下沿敏感，下面两种格式是错误的：

错误格式 1：

```
PROCESS(Clock)
BEGIN
  IF rising_edge(Clock) THEN
    ⋮
  ELSIF falling_edge(Clock) THEN
    ⋮
  END IF;
END PROCESS;
```

错误格式 2：

```
PROCESS(Clock)
BEGIN
  IF rising_edge(Clock) THEN
    ⋮
```

```
    ELSE
      ⋮
    END IF;
  END PROCESS;
```

⑥ 进程中的信号赋值是在进程挂起时生效的，而变量赋值则是即时生效的(参见2.3.2小节)。

2.6 VHDL顺序语句

顺序语句(Sequential Statements)是与并行语句(Concurrent Statements)相对而言的，其特点是：每一条顺序语句的执行顺序与其书写顺序对应，改变顺序语句的书写顺序有可能(不是必然)改变综合的结果。顺序语句只能出现在进程和子程序中。

前面在介绍进程时提到过，所谓"顺序执行"是指在仿真意义上具有一定的顺序性(或者说在逻辑上有先、后之分)，并不意味着这些语句对应的硬件结构也有相同的顺序性。当顺序语句综合后，映射为实际的门电路，系统一上电，这些门电路就同时开始工作。电路可实现逻辑上的顺序执行，实际上所有门电路是并行地工作，并没有先、后之分。这种以并行的工作方式实现顺序的逻辑是硬件描述语言的一大特点，也是进程可以在被激活的瞬间执行完进程中所有语句(见图2-17)的原因。读者在积累了一定的编程经验后应当体会出VHDL中顺序语句与传统概念中软件编程语言的顺序语句之间的差别。

VHDL中主要的顺序语句有6种：赋值语句、流程控制语句、空操作语句、等待语句、子程序调用语句和返回语句。本章主要介绍前3种语句。

2.6.1 赋值语句

赋值语句包括信号赋值语句和变量赋值语句。信号赋值语句在进程与子程序之外是并行语句，在进程与子程序之内则为顺序语句；而变量赋值语句只存在于进程和子程序中。这两种赋值语句的格式在2.3节已有介绍，这里不再详述。

信号赋值语句在进程中的执行机制比较特殊，下面通过例程2-12及其仿真波形(见图2-20)来说明这一机制，并总结变量与信号的不同之处。

例程 2－12　信号赋值语句在进程中的执行机制

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

ENTITY TEST_Signal IS
PORT
(Reset,Clock:IN Std_Logic;
NumA,NumB:OUT Integer RANGE 0 TO 255
);
END;

ARCHITECTURE TEST OF TEST_Signal IS
SIGNAL A,B:Integer RANGE 0 TO 255;              --声明信号 A、B,注意其位置
BEGIN
  PROCESS(Reset,Clock)
  VARIABLE C:Integer RANGE 0 TO 255;            --声明变量 C,注意其位置
  BEGIN
    IF Reset='1' THEN                           --系统复位时对 A、B、C 赋初值
      A<=0;                                     --对信号赋值用"<="
      B<=2;
      C:=0;                                     --对变量赋值用":="
    ELSIF rising_edge(Clock) THEN               --在每个时钟上升沿执行
      C:=C+1;
      A<=C+1;
      B<=A+2;
    END IF;
  END PROCESS;
  NumA<=A;                                      --信号可在 PROCESS 外使用,变量不行
  NumB<=B;
END;
```

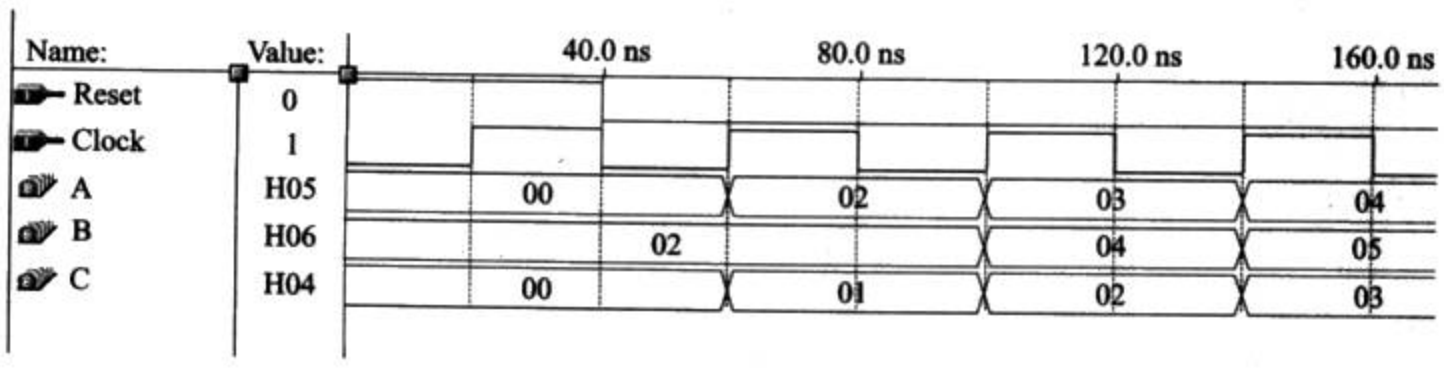


图 2－20　例程 2－12 的仿真波形

从上面的程序和仿真图中，可总结出信号与变量的一些不同之处：

① 声明形式与赋值符号不同。变量声明为 Variable，赋值符号为“:=”；而信号声明为 Signal，代入语句采用“<=”代入符。

② 信号在结构体内、进程外定义，而变量在进程内定义。换句话说，信号的有效域为整个结构体，可在不同进程间传递数值；变量的有效域只是定义该变量的进程，不能为多个进程所用。

③ 操作过程不相同。观察第 2 个时钟上升沿可以看出，当执行 C:=C+1 后 C 的值立即变成 1，A<=C+1 也就相应地变成了 A<=1+1(=2)；但是 B<=A+2 却没有相应地变成(2+2)=4，而是保持着原来的(0+2)=2。这说明，**在进程中，变量赋值语句一旦被执行，目标变量立即被赋予新值，在执行下一条语句时，该变量的值为上一句新赋的值；而信号的赋值语句即使被执行，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(信号是在进程挂起时才发生代入的)。**

2.6.2 流程控制语句

常用的流程控制语句有 3 个：IF 语句、CASE 语句和 LOOP 语句，下面分别介绍这 3 个语句。

1. IF 语句

IF 语句的语法格式有 3 种。

IF 语法格式 1：

```
IF 条件式 THEN
  顺序语句
END IF;
```

IF 语法格式 2：

```
IF 条件式 THEN
  顺序语句
ELSE
  顺序语句
EIND IF;
```

IF 语法格式 3：

```
IF  条件式 1 THEN
  顺序语句
ELSIF  条件式 2 THEN
  顺序语句
  ⋮
ELSE
  顺序语句
END IF;
```

以上几种格式的 IF 语句的执行流程，与软件编程语言中的 IF 语句相差无几。下面逐一进行简单介绍。

格式 1：判断条件式是否成立。若条件成立，则执行 THEN 与 END IF 之间的顺序语句；若条件不成立，则跳过不执行，IF 语句结束。

格式 2：判断条件式是否成立。若条件成立，则执行 THEN 与 ELSE 之间的顺序语句；若条件不成立，则执行 ELSE 与 END IF 之间的顺序语句。

格式 3：自上而下逐一判断条件式是否成立。若条件成立，则执行相应的顺序语句，并不再判断其他条件式，直接结束 IF 语句的执行。其执行流程与 WHEN_ELSE 相似(见 2.5.1 小节)。

使用 IF 语句时的要点：

① IF 语句可以嵌套，但层数不宜过多。

② 上面的 IF 语法格式 3 和 WHEN_ELSE 一样，用于有优先级的条件判断，因此各条件式中的条件可以重叠。如果所需判断的条件没有优先级的差别，且条件之间没有重叠的情况，那么建议使用 CASE 语句。

③ 如例程 2－12 所示，用 IF 语句描述异步复位信号和时钟沿时，只能用 IF_ELSIF_END IF 的格式，不能出现 ELSE。

④ 在进程中用 IF 语句描述组合逻辑电路时，务必覆盖所有的情况；否则综合后将引入锁存器，违背设计初衷。

前 3 个要点比较容易理解，最后一点涉及到 IF 语句与综合结果的映射关系，有一定的难度。这里不准备用大量的篇幅讨论这个问题，只通过一个简单的程序来说明组合进程(用 IF 语句实现)的设计要点。

例程2-13是与例程2-9描述的8输入优先编码器基本相同的电路，只是多加了一个使能端EN和一个无效输入(当所有输入都为0时)的指示端IDLE。

例程2-13 带使能端EN和无效输入指示端IDLE的优先编码器

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

ENTITY Encoder IS
PORT
(En:IN Std_Logic;
I:IN Std_Logic_Vector(7 DOWNTO 0);
A:OUT Std_Logic_Vector(2 DOWNTO 0);
Idle:OUT Std_Logic
);
END;

ARCHITECTURE Behavior OF Encoder IS
BEGIN
  PROCESS(En,I)                    --所有需要读入的信号都必须放在敏感信号参数表中
  BEGIN
  IF En='1' THEN
    IF I(7)='1' THEN
      A<="111";
      Idle<='0';
    ELSIF I(6)='1' THEN
      A<="110";
      Idle<='0';
    ELSIF I(5)='1' THEN
      A<="101";
      Idle<='0';
    ELSIF I(4)='1' THEN
      A<="100";
      Idle<='0';
    ELSIF I(3)='1' THEN
      A<="011";
      Idle<='0';
    ELSIF I(2)='1' THEN
```

```
        A<="010";
        Idle<='0';
      ELSIF I(1)='1' THEN
        A<="001";
        Idle<='0';
      ELSIF I(0)='1' THEN
        A<="000";
        Idle<='0';
      ELSE
        A<="000";
        Idle<='1';
      END IF;
    ELSE                              --此处必须有 ELSE 语句,否则综合器将引入锁存器
      A<="000";
      Idle<='1';
    END IF;
  END PROCESS;
END;
```

从例程 2-13 可以总结出设计组合进程的两个要点：

① 所有在该进程中被读取的信号都必须加入敏感变量表；否则，当没有被包括在敏感变量表中的信号发生变化时，进程不能即时生成新的输出，综合器可能会误以为设计者希望存储数据而引入锁存器。这样就违背了组合逻辑的原则：当前输出只与当前输入有关系。

② 必须在所有条件下指定输出值。在没有指定输出值的条件下，综合器同样可能误以为设计者希望存储数据而引入锁存器。

在第 14 章中将对综合结果有更深入的讨论，目前只须记住以上提到的几个要点即可。

2. CASE 语句

CASE 语句根据满足的条件直接选择多项顺序语句中的一项执行，其语法格式如下：

```
CASE 表达式 IS
   WHEN 选择值[|选择值] => 顺序语句;     --若有多个选择值,则用"|"间隔。2|3 表示 2 或 3
   WHEN 选择值[|选择值] => 顺序语句;     --"=>"相当于 IF 语句中的 THEN
      ⋮
   WHEN OTHERS=>顺序语句;
END CASE;
```

CASE语句与WITH_SELECT很相似，下面将例程2-8(用WITH_SELECT语句实现的8路4选1复用器)改用CASE语句实现(见例程2-14)。

例程2-14 8路4选1复用器

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

ENTITY MUX4 IS
PORT
(Data0,Data1,Data2,Data3:IN Std_Logic_Vector(7 DOWNTO 0);
 Sel:IN Std_Logic_Vector(1 DOWNTO 0);
 DOUT:OUT Std_Logic_Vector(7 DOWNTO 0));
END;

ARCHITECTURE Dataflow OF MUX4 IS
BEGIN
  PROCESS(Sel,Data0,Data1,Data2,Data3)          --所有在进程中被读取的信号均为敏感信号
  BEGIN
    CASE Sel IS
      WHEN "00"=>DOUT<=Data0;
      WHEN "01"=>DOUT<=Data1;
      WHEN "10"=>DOUT<=Data2;
      WHEN "11"=>DOUT<=Data3;
      WHEN OTHERS=>DOUT<="00000000";  --覆盖所有条件
    END CASE;
  END PROCESS;
END;
```

CASE语句使用要点：

① 选择值不可重复或重叠。例如不可同时出现两次WHEN "00"(重复)，也不可同时出现WHEN "00"| "01"和WHEN "00"| "11"(选择值"00"重叠)。

② 当CASE语句的选择值无法覆盖所有情况时，要用OTHERS指定未能列出的其他所有情况的输出值。

③ 用CASE语句设计组合进程的要点与用IF语句设计组合进程的要点相同。

3. LOOP语句

LOOP语句是循环语句，它有3种常见格式(LOOP_EXIT、FOR_LOOP和

WHILE_LOOP),但 MAX+plus II 只支持最常用的 FOR_LOOP 语句,其语法格式如下:

```
[LOOP 标号:] FOR 循环变量 IN 循环次数范围 LOOP
                顺序语句
             END LOOP [LOOP 标号];
```

下面对 FOR_LOOP 语句的各组成部分进行说明:

① 循环变量 这是一个临时的变量,仅在此 LOOP 语句中有效,因此不需要事先定义。但要注意,在 PROCESS 的声明区不要定义与此同名的变量。

② 循环次数范围 主要有两种格式,即"…TO …"和"… DOWNTO …"。循环变量从循环次数范围的初值开始,每执行完一次顺序语句后递增(或递减)1,直到达到循环次数范围的终值为止。

③ LOOP 标号不是必需的,可以省略。

下面这个例子实现了 0～9 的累加计算:

```
Sum:=0;
FOR i IN 0 TO 9 LOOP
   Sum:=Sum+i;
END LOOP;
```

使用 LOOP 语句时,要注意循环次数范围只能用具体数值表示。下面这种描述方式是不可综合的:

```
VARIABLE Length: Integer RANGE 0 TO 15;
    ⋮
FOR i IN 0 to Length LOOP
    ⋮
END LOOP;
```

4. NULL 语句

NULL 语句即空操作语句。顾名思义,它不执行任何操作,只是让程序接着往下执行。NULL 语句一般用在 CASE 语句中,用于表示在某些情况下对输出不作任何改变。所谓"不作任何改变",实质上隐含了锁存信号的意思,所以有些综合器(如 MAX+plus II)在遇到 NULL 语句时将引入锁存器。因此,如果要设计纯组合逻辑电路,就不要使用 NULL 语句。

典型的 NULL 语句的应用如下:

```
PROCESS(Clock)
BEGIN
  IF rising_edge(Clock) THEN
     ⋮
    CASE Sel IS
      WHEN …
      WHEN OTHERS=>NULL;
    END CASE;
       ⋮
  END IF;
END PROCESS;
```

此处是在时序电路中使用 NULL，即有意识地引入锁存器。

从上面这个例子可以看出，VHDL 与一般的软件设计程序语言有很大差异，一定不能用纯软件的思维方式看待硬件编程。

一个很简单的 NULL 语句就包含了不少学问，其他语句就更不用说了。虽然作者很希望将自己的编程经验浓缩在薄薄的一本书里介绍给读者，但是这个愿望是很难实现的。所谓“运用之妙，存乎一心”，想将所有经验形成文字实在是不容易。而且“纸上得来终觉浅”，如果没有实践，谈再多的经验又有什么用呢？所以作者希望各位读者一定要多练习，在不断的实践中总结经验，这样才能真正地掌握一门语言。

第3章

状态机在 VHDL 中的实现

数字电路分为组合逻辑和时序逻辑两大类。组合逻辑电路的输出只与当前的输入有关；而时序逻辑电路的输出则不仅与当前的输入有关，而且还与过去的一系列输入有关。

组合逻辑电路就像老式的按钮式风扇，按 1 输出弱风，按 2 输出中等强度的风，按 3 则输出强风。输出风力的大小(输出值)完全取决于当前按下的按钮值(输入值)。

时序逻辑电路则更像空调遥控器中靠“+”键和“−”键调节温度的控制电路，无法单凭当前按下的键(输入值)来判断将设定的温度(输出值)，显然必须知道遥控器当前设定的温度，即遥控器目前的状态。

状态机(State Machine)可以说是一个广义时序电路，触发器、计数器、移位寄存器等都算是它的特殊功能类型的一种。实际时序电路中的状态数是有限的，因此又叫做有限状态机(FSM，Finite State Machine)。用 VHDL 设计状态机不必知道其电路具体实现的细节，而只需要在逻辑上加以描述。因此本书在提到状态机一词时，更多地是指对状态及状态转移的总体描述，而不是指一个具体的时序电路。

状态机又分为 Moore 型与 Mealy 型，前者的输出仅取决于其所处状态；而后者的输出则不仅与当前所处状态有关，同时也与当前的输入有关。下面分别介绍这两种状态机在 VHDL 中的实现。

3.1 Moore 状态机的 VHDL 描述

Moore 状态机的输出仅由状态决定，一个典型的 Moore 状态机的状态转移图如图 3 - 1

所示，其对应的 VHDL 程序(带异步复位信号)见例程 3－1。例程 3－1 的仿真波形如图 3－2 所示。

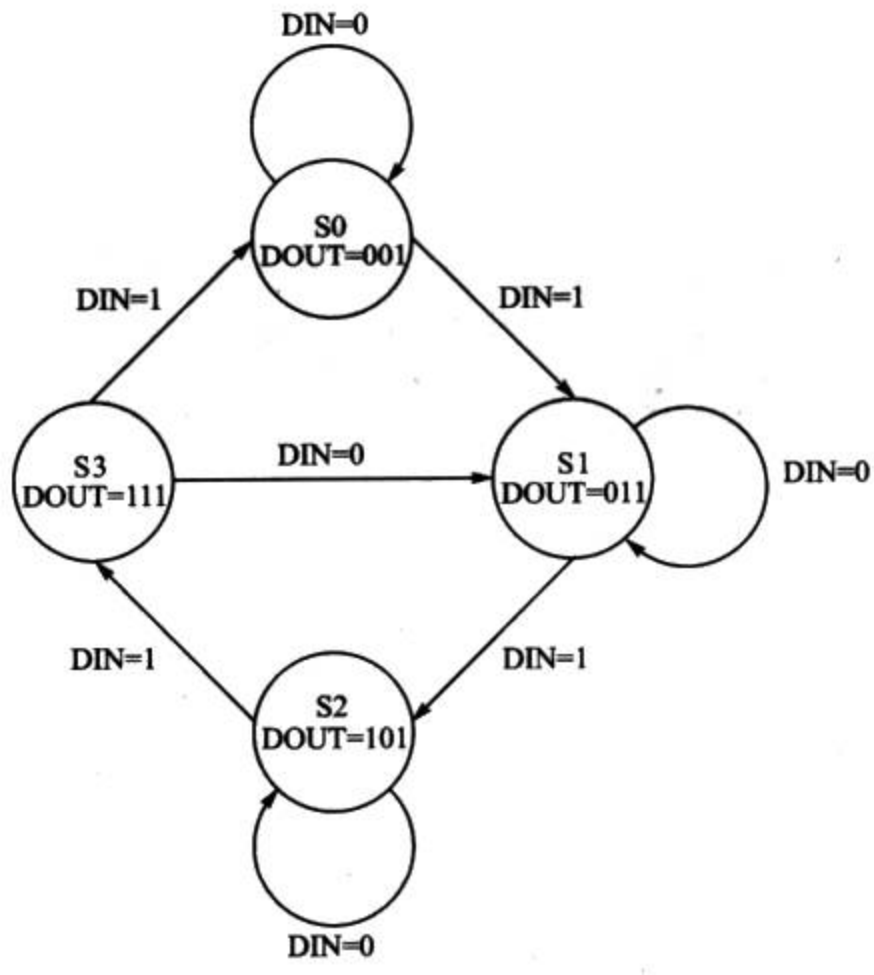


图 3－1　Moore 状态机的状态转移图

例程 3－1　实现 Moore 状态机的状态转移

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

ENTITY Moore IS
PORT
(Reset:IN Std_Logic;
 Clock:IN Std_Logic;
 DIN:IN Std_Logic;
 DOUT:OUT Std_Logic_Vector(2 DOWNTO 0)
);
END;

ARCHITECTURE Mooremachine OF Moore IS
TYPE State_type IS (S0,S1,S2,S3);        --用枚举类型来表示状态,使程序更具可读性
```

```
SIGNAL State:State_type;
BEGIN

Change_State:                          --此进程用以确定状态的转移
  PROCESS(Reset,Clock)
  BEGIN
    IF Reset='1' THEN
      State<=S0;
    ELSIF rising_edge(Clock) THEN
      CASE State IS
        WHEN S0=>
          IF DIN='1' THEN
            State<=S1;
          END IF;
        WHEN S1=>
          IF DIN='1' THEN
            State<=S2;
          END IF;
        WHEN S2=>
          IF DIN='1' THEN
            State<=S3;
          END IF;
        WHEN S3=>
          IF DIN='1' THEN
            State<=S0;
          ELSE
            State<=S1;
          END IF;
      END CASE;
    END IF;
  END PROCESS;

Output_Process:                        --此进程决定输出值
  PROCESS(State)
  BEGIN
    CASE State IS
      WHEN S0=>DOUT<="001";
```

```
        WHEN S1=>DOUT<="011";
        WHEN S2=>DOUT<="101";
        WHEN S3=>DOUT<="111";
    END CASE;
  END PROCESS;

END;
```

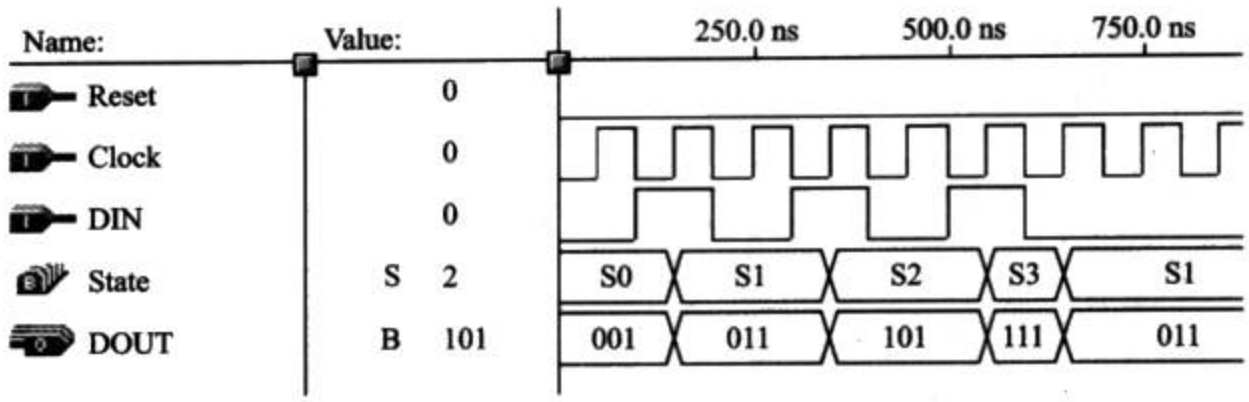


图 3-2　例程 3-1 仿真波形图

例程 3-1 是 MAX+plus II 默认的状态机描述格式之一，因此其仿真波形图中将 State 以状态名的形式表示出来，使仿真结果十分直观、易懂。从图 3-2 中可以看出，状态机上电后默认的初始状态为定义的第 1 个状态(本例中为 S0)。

决定状态转移的进程 Change_State 的格式是相对固定的，而决定输出值的进程 Output_Process则可以用 WITH_SELECT 语句来代替，或者，如果输出值需要同步，那么也可以在输出进程中引入时钟。实例篇中的程序几乎都用到了状态机，读者如果需要实际应用的例子，可参考该篇。

需要说明的是，状态机描述的格式并不是只有例程 3-1 中给出的那种。事实上，更多的综合器建议将状态机写成例程 3-2 中的格式，该格式是严格按图 3-3 所示的时钟同步状态机结构写的。

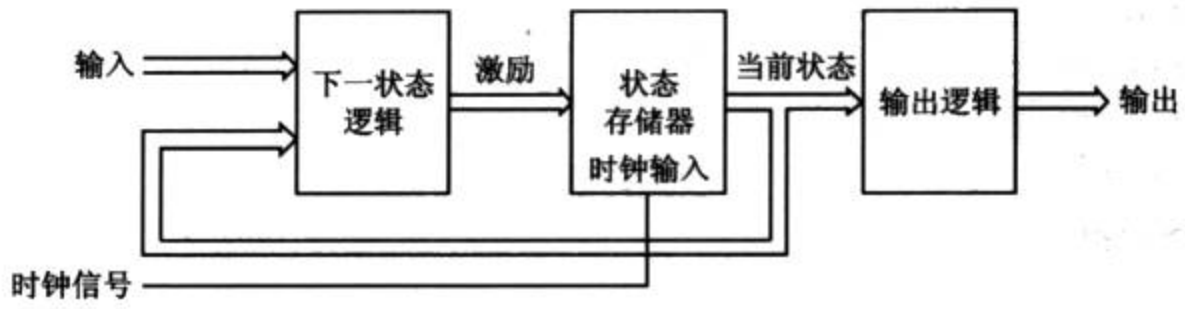


图 3-3　时钟同步 Moore 状态机结构图

例程 3-2　图 3-3 对应的 Moore 状态机程序

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

ENTITY Moore IS
PORT
(Reset:IN Std_Logic;
 Clock:IN Std_Logic;
 DIN:IN Std_Logic;
 DOUT:OUT Std_Logic_Vector(2 DOWNTO 0)
);
END;

ARCHITECTURE Mooremachine OF Moore IS
TYPE State_type IS (S0,S1,S2,S3);
SIGNAL PresentState:State_type;
SIGNAL NextState:State_type;
BEGIN
State_Reg:
  PROCESS(Reset,Clock)                    --相当于图 3-3 中的"状态存储器"
  BEGIN
    IF Reset='1' THEN
      PresentState<=S0;
    ELSIF rising_edge(Clock) THEN
      PresentState<=NextState;
    END IF;
  END PROCESS;

Change_State:                             --相当于图 3-3 中的"下一状态逻辑"
                                          --和"输出逻辑"的结合
                                          --也可以将输出逻辑像例程 3-1 一样独立出来
  PROCESS(PresentState,DIN)
  BEGIN
    CASE PresentState IS
      WHEN S0=>
        IF DIN='1' THEN
          NextState<=S1;
```

```
    ELSE
      NextState<=S0;
    END IF;
    DOUT<="001";
  WHEN S1=>
    IF DIN='1' THEN
      NextState<=S2;
    ELSE
      NextState<=S1;
    END IF;
    DOUT<="011";
  WHEN S2=>
    IF DIN='1' THEN
      NextState<=S3;
    ELSE
      NextState<=S2;
    END IF;
    DOUT<="101";
  WHEN S3=>
    IF DIN='1' THEN
      NextState<=S0;
    ELSE
      NextState<=S1;
    END IF;
    DOUT<="111";
  END CASE;
END PROCESS;

END;
```

例程 3-2 中的状态机用两个信号来表示状态：PresentState 表示当前状态，NextState 表示下一状态。此状态机由两个进程构成(若将输出逻辑独立出来，则可以写成 3 个进程)：State_Reg 和 Change_State。进程 State_Reg 负责在每个时钟上升沿将 NextState 的值赋给 PresentState，即将下一状态值赋给当前状态，但它并不决定下一状态的取值。决定下一状态取值(或者说状态转移)的是 Change_State 进程，这是一个组合进程，它根据外部输入的控制信号和当前状态确定下一状态的取向。

目前介绍 VHDL 的大部分书籍，都使用例程 3－2 中的格式来描述状态机，并宣称这样描述状态机比较容易理解。作者认为，例程 3－1 中的格式更符合一般人的逻辑思维，特别是那些没有受过数字电路设计专业训练的人。虽然很多综合器都建议用户将状态机描述为例程 3－2中的格式，但 MAX＋plus II 却不支持该格式，而只承认例程 3－1 描述的状态机，因此，例程 3－2 的仿真波形（见图 3－4）中 PresentState 并没有以状态名显示，而是显示其编码。

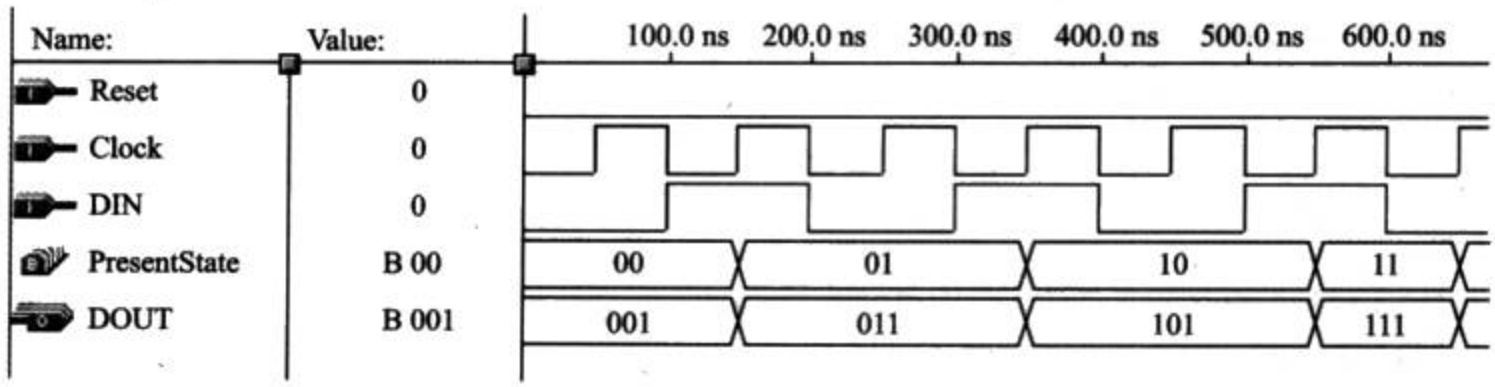


图 3－4　例程 3－2 仿真波形

从图 3－4 中可以看出，虽然 MAX＋plus II 没有将例程 3－2 当成状态机处理，但其工作仍是正常的，差别仅在状态名的显示。

综上所述，建议使用 MAX＋plus II 的读者按例程 3－1 的格式来描述 Moore 状态机，这里给出例程 3－2 只是为了使读者开拓眼界，以便阅读其他 VHDL 书籍时能适应其状态机的描述方式。本书中所有状态机程序均是基于例程 3－1 的格式。

3.2　Mealy 状态机的 VHDL 描述

Mealy 状态机的输出是由当前的状态与输入共同决定的。图 3－5 是一个典型的 Mealy 状态机的状态转移图，例程 3－3 为此状态机的 VHDL 程序，图 3－6 为相应的仿真波形。相信读者有了 3.1 节的基础，理解本程序应该没有问题，因此程序未加注释。

例程 3－3　Mealy 状态机程序

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

ENTITY MealyMachine IS
PORT
(Reset:IN Std_Logic;
 Clock:IN Std_Logic;
```

```
 DIN:IN Std_Logic;
 DOUT:OUT Std_Logic_Vector (2 DOWNTO 0)
);
END;

ARCHITECTURE Statemachine OF MealyMachine IS
TYPE State_type IS  (S0,S1,S2,S3);
SIGNAL State:State_type;

BEGIN

Change_State:
  PROCESS(Reset,Clock)
  BEGIN
    IF Reset='1' THEN
      State<=S0;
    ELSIF rising_edge(Clock) THEN
      CASE State IS
    WHEN S0=>
      IF DIN='1' THEN
        State<=S1;
      END IF;
    WHEN S1=>
      IF DIN='1' THEN
        State<=S2;
      END IF;
    WHEN S2=>
      IF DIN='1' THEN
        State<=S3;
      END IF;
    WHEN S3=>
      IF DIN='1' THEN
        State<=S0;
      ELSE
        State<=S1;
      END IF;
END CASE;
```

```
    END IF;
  END PROCESS;

Output_Process:
  PROCESS(State,DIN)
  BEGIN
    CASE State IS
      WHEN S0=>
        IF DIN='0' THEN
          DOUT<="000";
        ELSE
          DOUT<="001";
        END IF;
      WHEN S1=>
        IF DIN='0' THEN
          DOUT<="010";
        ELSE
          DOUT<="011";
        END IF;
      WHEN S2=>
        IF DIN='0' THEN
          DOUT<="100";
        ELSE
          DOUT<="101";
        END IF;
      WHEN S3=>
        IF DIN='0' THEN
          DOUT<="110";
        ELSE
          DOUT<="111";
        END IF;
    END CASE;
  END PROCESS;

END;
```

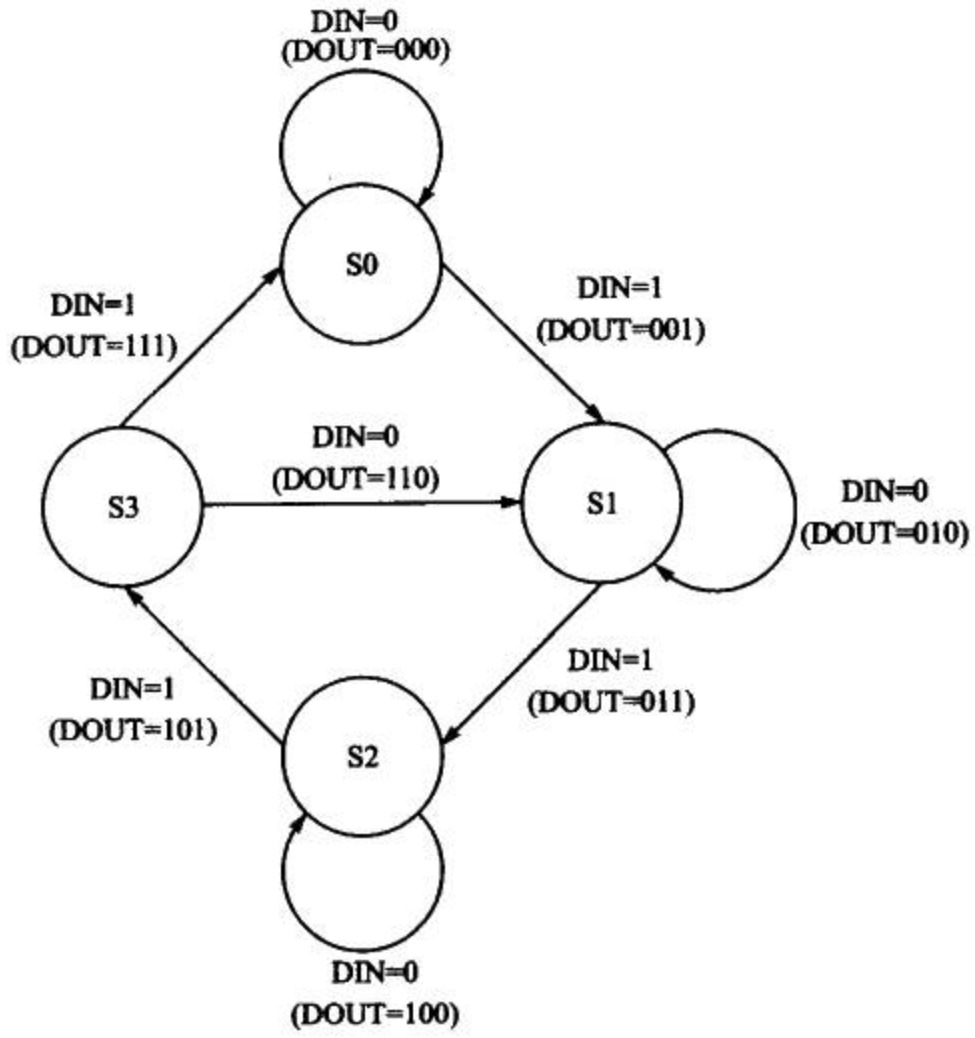


图 3-5　Mealy 状态机的状态转移图

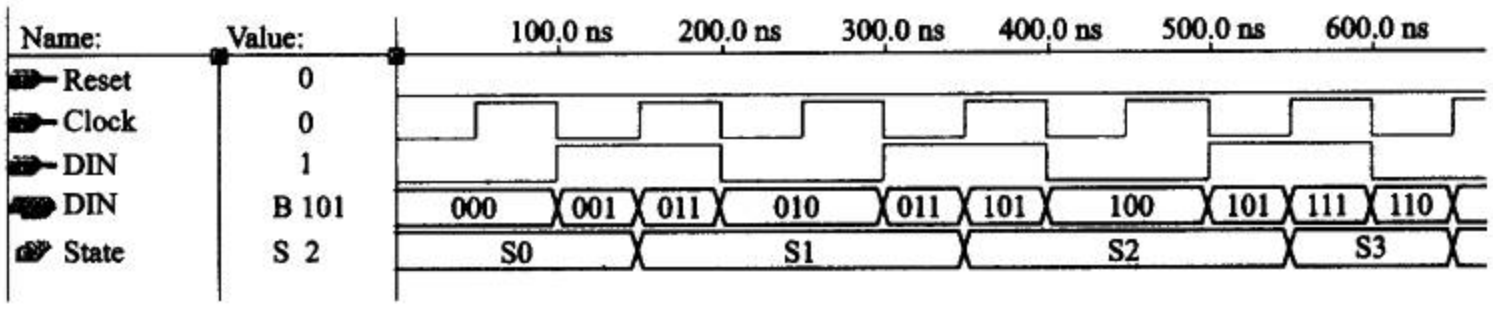


图 3-6　例程 3-3 仿真波形式

为节省篇幅，本节只给出 MAX+plus II 可以识别的状态机格式。读者若有兴趣，可参照例程 3-2 用另一种格式来描述状态机。

3.3　状态机的容错设计

设计一个高效而又稳定的状态机不是一件容易的事(特别是在高速应用中)。为了提高状态机的容错性，不少论文都提出了一些行之有效的方法，主要是在改进结构和状态编码上作文

章。本书作为一本 VHDL 的入门书，不准备深入探讨这个问题，仅给出最简单的容错设计方法。这一方法对初学者来说，既容易理解，又完全够用。读者若想深入研究状态机的问题，可以在网上搜索相关的论文研读。

状态机的容错设计主要是针对未定义状态(或称剩余状态)。状态机的每一状态即对应着一个用二进制表示的状态编码，因此，若状态机中定义的状态数不足 2^N(N 为状态编码的二进制位数)，则必然会有一些状态编码未使用，即存在剩余状态。例如，在状态机中定义了 3 个状态，那么它们对应的编码是用 2 位二进制数表示的，分别是 00、01、10，而 11 则未使用，这就是剩余状态了。系统一旦意外进入此状态，状态机的行为将出错。

为了使系统进入未定义状态后能够回复正常工作，那么只需要在 CASE 语句中加上 WHEN OTHERS语句，指明在出现其他未知状态时的处理方法。一般用以下两种处理方法：

① 单独设计一个状态(如 Error)，用以处理状态机出错时的情况，然后用

```
WHEN OTHERS=>State<=Error;
```

使状态机从未定义状态中跳转到处理出错情况的状态。

② 直接回复到其他已设定的状态。如果系统对状态机的容错性要求不是很高，那么可以不对状态机出错进行事后的处理，直接回复到某一确定状态即可(通常是回复到初始状态)。

第4章

系统层次化设计

层次化设计(Hierarchical Design)在数字系统设计中被广泛地应用,因为它有下面两个主要的优点:① 一些常用的模块可以被单独创建并存储,在以后的设计中可以直接调用该模块,而无需重新设计;② 它使整个设计更结构化,程序也具有更高的可读性:顶层文件只将一些小模块整合在一起,这使整个系统的设计思想比较容易被理解。

本章将介绍如何在 MAX+plus II 中采用图形法与文本法结合的混合输入方法实现元件重用(Reuse)与系统的层次化设计。

4.1 层次化设计的概念

层次化设计的示意图如图 4-1 所示。

从图中可以看出,层次化设计的核心思想有两个,一是模块化,二是元件重用(Reuse)。

① 模块化:可以将一个大系统划分为几个子模块,而这些子模块又分别由更小的模块组成,如此往下,直至不可再分。这也正是自顶向下(Top-down)的设计方法。图 4-1 中,顶层文件所描述的实体 A 由 B、C、D 三个实体组成,而实体 B 和实体 C 又分别由实体

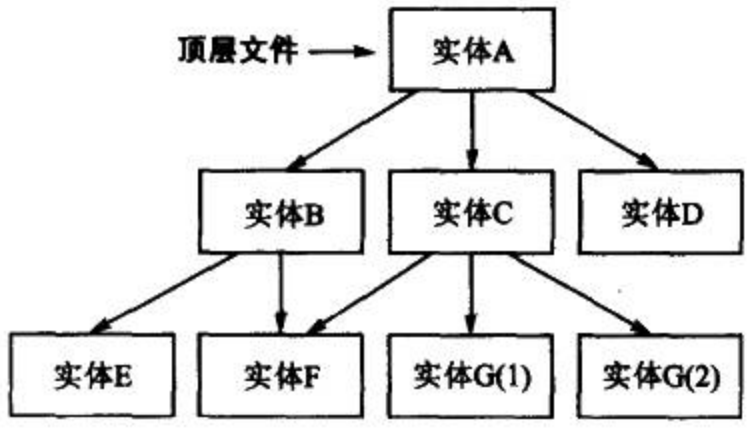


图 4-1 层次化设计的示意图

E、F 和实体 F、G 构成。每个实体都可以看成上一层实体中的一个模块或元件(Component)，系统就像搭积木一样一层一层地构建。

② 元件重用(Reuse)：同一个元件可以被不同的设计实体调用，也可以被同一个设计实体多次调用。图 4－1 中，实体 F 分别被实体 B 和实体 C 调用，而实体 G 则被实体 C 调用了 2 次。元件重用不但大大减轻了设计者的工作量，而且使程序更结构化和具有更高的可读性。

4.2　在 MAX＋plus II 中实现层次化设计

4.2.1　元件重用

这里先看一个简单的问题：假设系统中有一个 200 kHz 的时钟，系统要求将其分为 100 kHz、50 kHz 和 25 kHz，并在这 4 个频率的时钟中选择一个作为输出(如图 4－2 所示)，如何从 CLk1(200 kHz)生成其他频率的时钟信号呢？

很容易就可以想到两种方案：

① 设计一个 2 分频电路、一个 4 分频电路和一个 8 分频电路，直接从 200 kHz 分频得到所需的几个频率的时钟信号(如图 4－3 所示)；

② 只设计一个 2 分频电路，用 3 个 2 分频电路级联的方式，从 200 kHz 逐级分出所需的时钟信号(如图 4－4 所示)。

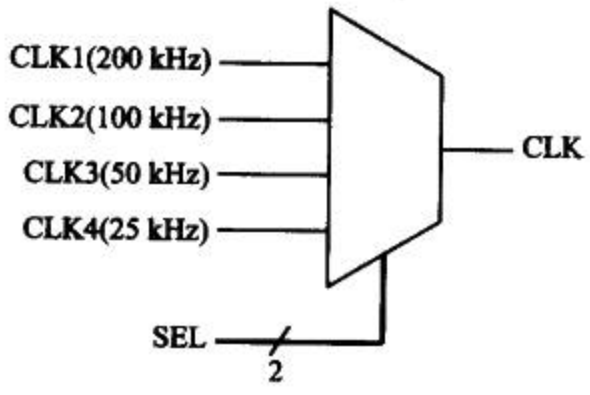


图 4－2　时钟选择器示意图

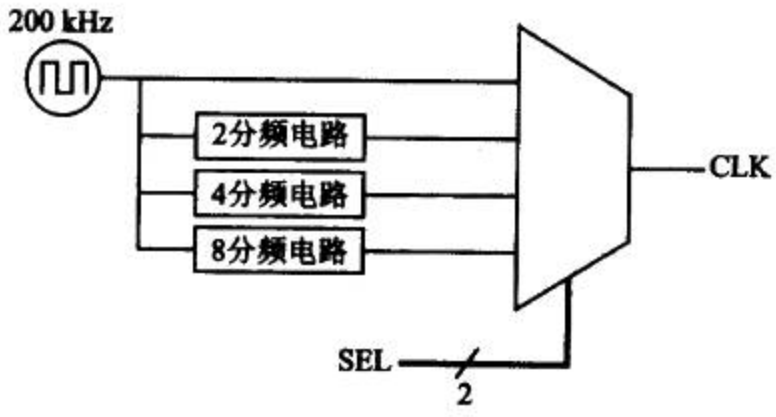


图 4－3　方案①示意图

从编程的工作量、对已有资源的再利用和节省系统资源这 3 个角度来看，方案②显然要优于方案①。下面介绍如何在 MAX＋plus II 中，利用已有的 2 分频电路(参见例程 1－1)和 4X1 多路复用器(参见例程 2－7)实现方案②的电路。

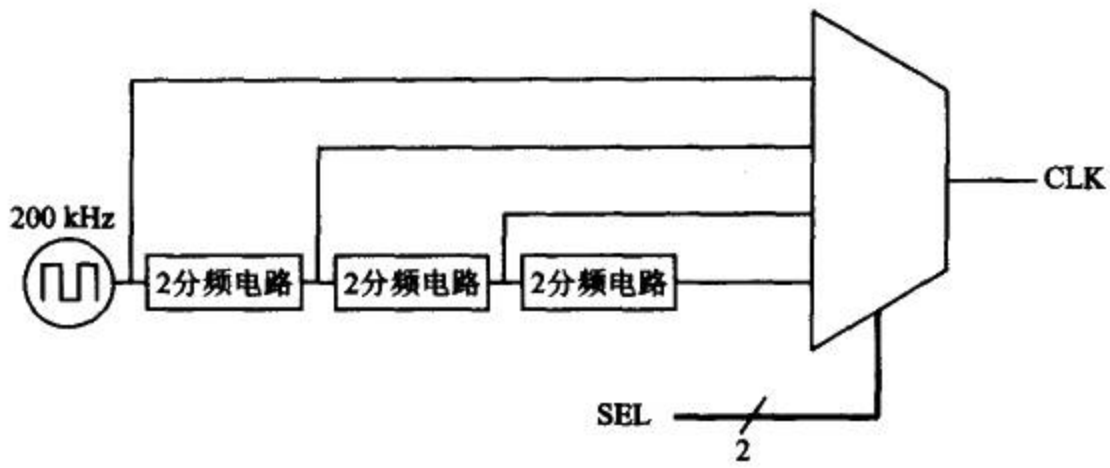


图 4-4　方案②示意图

1. 生成元件符号

如果想把例程 1-1 描述的 2 分频电路作为上一层设计实体中的元件，就必须先生成元件符号(Symbol)。生成元件符号很简单，只要编译这个程序就可以了，MAX+plus II 会在编译时自动为此实体生成一个后缀名为 sym 的元件符号文件，这样就可以在图形编辑器(Graphic Editor)中调用此元件符号了。

特别值得注意的是：**MAX+plus II 只在第 1 次成功编译程序时会自动生成元件符号**。这意味着：① 如果此程序没有通过编译(例如因为语法错误等)，那么将得不到它的元件符号，也就无法调用它；② 在第 1 次成功编译后(即自动生成元件符号后)，如果修改这个设计实体的端口(包括端口名、端口数等)，就必须更新元件符号(编译时不会自动更新)。

更新元件符号十分简单，将当前文件设为工作文件后，在 File 菜单中选择 Create Default Symbol 选项，会出现对话框，问用户是否要将现有的 Symbol 文件覆盖，选择“确定”后，元件符号就得到了更新。

2. 调用元件符号

在 MAX+plus II 菜单中选择 Graphic Editor 就启动了图形编辑器。在图形编辑器中调用元件有 3 种方法：① 在 Symbol 菜单中选择 Enter Symbol；② 双击“图形编辑区”的空白处；③ 右击后选择 Enter Symbol。执行完上面的操作后，会出现如图 4-5 所示的对话框。

在 Symbol Libraries 处列出了很多元件库的目录，这里选择用户工作目录。这时在左下角的 Symbol Files 处会出现所有用户生成的元件，选择所需元件，单击 OK 键就完成了元件的调用。这时图形编辑器的“图形编辑区”上就出现了刚才生成的 2 分频电路的元件符号(见图 4-6)。

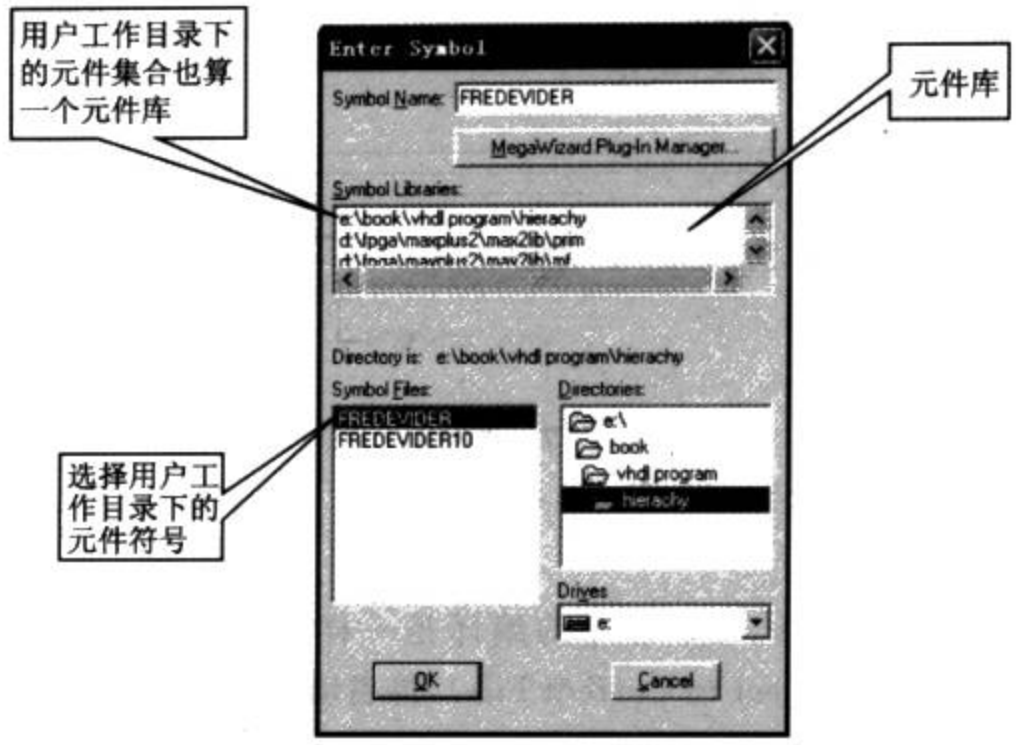


图 4-5　调用用户生成的元件

从图 4-6 中可以看出，这个元件符号有两个接口，CLOCK 为输入端，CLKOUT 为输出端，这与在 2 分频电路中实体的声明部分定义的端口是一致的。

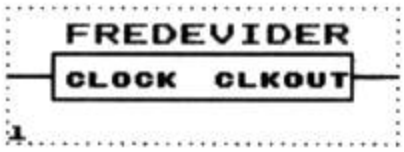


图 4-6　2 分频电路的元件符号

重复调用 3 次这个元件，同时调用 4X1 复用器的元件，接下来就是连线和定义输入/输出端口的工作了。

3. 定义输入/输出端口与连线

添加输入/输出引脚的方法是：在 Enter Symbol 对话框（见图 4-5）中的 Symbol Name 里输入 Input 即可添加输入引脚，输入 Output 即可添加输出引脚。

连线的方法是：如果需要连接两个元件的端口，那么将鼠标移到其中一个端口上，这时鼠标指示符会自动变为“+”形；然后拖动鼠标到第 2 个端口，松开鼠标左键后即画好一条线。

删除连线的方法是：单击点中该线，被点中的线会变成红色，按 Delete 键即可删除。

将调用的元件按图 4-7 连好线，然后存盘。

注意：存储的文件名不可与调用的任一元件名相同，即本设计不可命名为 FreDevider.gdf 也不可命名为 MUX4.gdf。

对比一下图 4-7 与图 4-4，是不是很像呢？这就是图形输入法的优点所在：十分直观、形象，很容易理解。本章以后的例程，如果需要用到层次化设计，一般均用 VHDL 设计底层模

块，然后在上层文件中以图形方式连接。

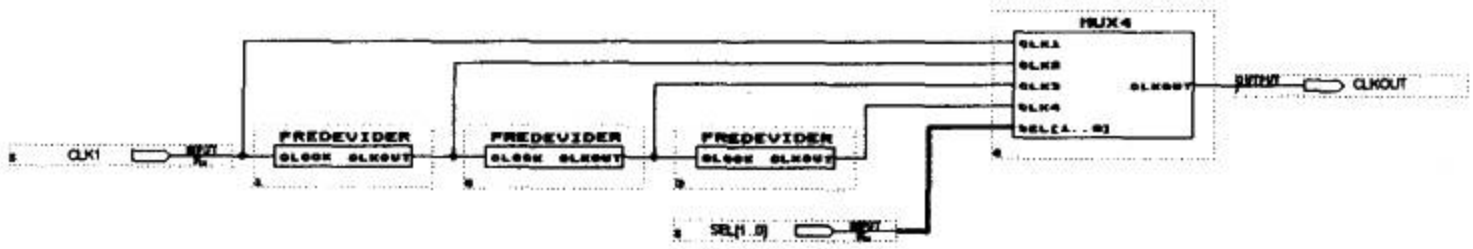


图 4-7　连接图

4. 编　译

其实，图形文件的编译和 VHDL 程序的编译是一样的，只要把当前文件设成工作文件(File=>Project=>Set Project to Current File)，再启动编译器进行编译就可以了，请读者参考第 1 章，此处不再赘述。

5. 图形编辑器注意事项

下面总结一些使用图形编辑器的注意事项和技巧，希望能帮助读者用好这个工具。

(1) 可以对连线命名，同名连线在电气上是连接的。例如，图 4-8 与图 4-7 的电路是一样的。对连线命名的方法是：单击要命名的连线，连线会变成红色，并有闪烁的黑点，此时键入的文字即为连线名。

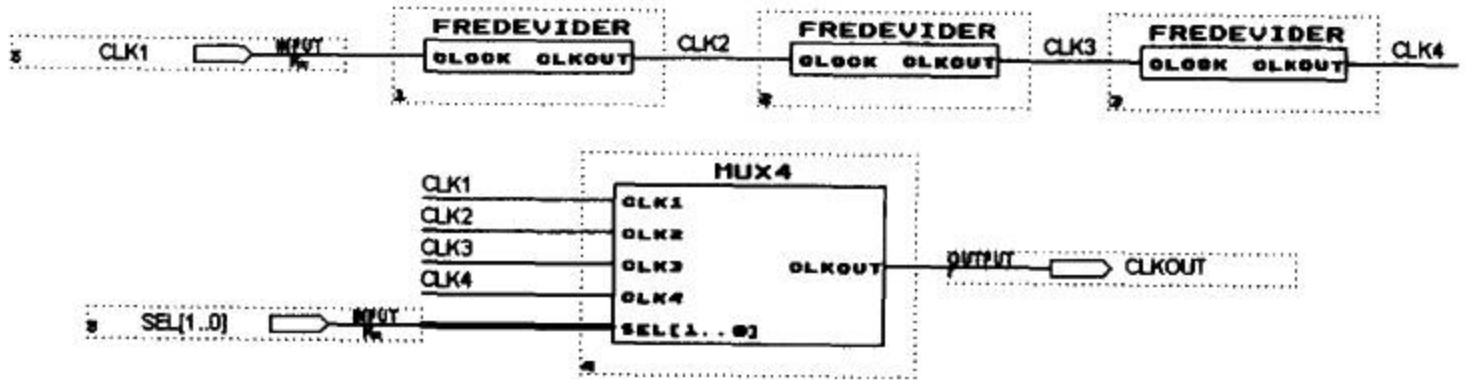


图 4-8　同名连线连接图

(2) 在进行总线(BUS)与输入/输出端口的连接时，必须从元件的端口处开始引线，不能从输入/输出端口开始引线(见图 4-9)。为什么这么做，读者自己一试便知。

(3) 当 VHDL 程序的元件符号更新后，必须在图形编辑器中也进行相应的更新。在 Symbol 菜单中，选择 Update Symbol 可对本图形文件中的元件符号进行更新(如图 4-10 所示)。更新方式有两种：① 选定一个或几个元件，在图 4-10 所示的对话框中选择

All Occurrences of Selected Symbol(s)，这样只更新选择选定的元件；② 选择 All Symbols in the File，更新本图形文件中所有用到的元件符号。

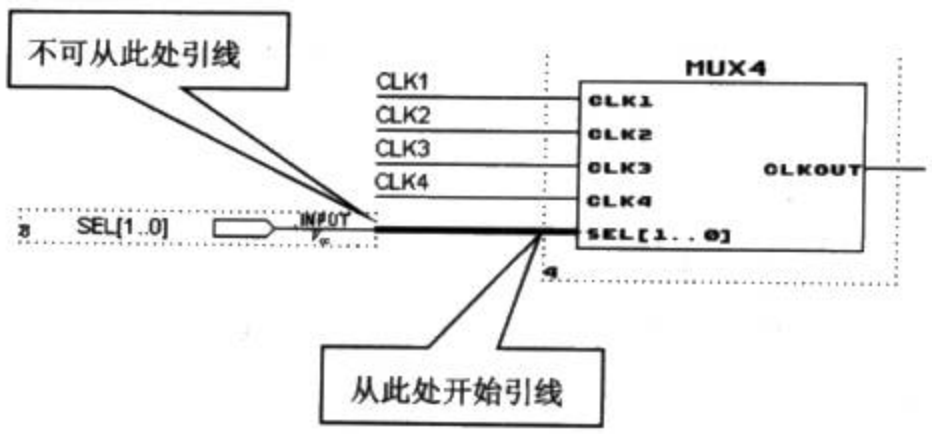


图 4-9　BUS 引线的特殊之处

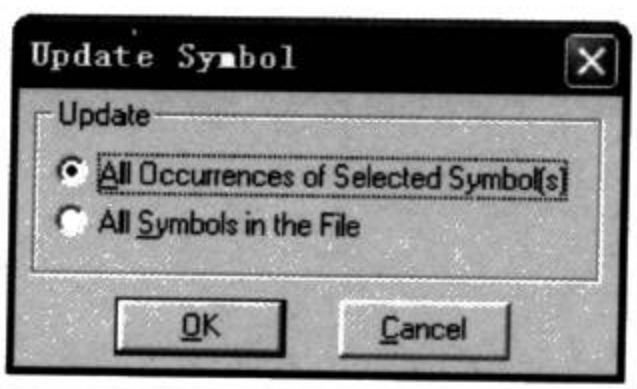


图 4-10　更新元件符号对话框

4.2.2　多层次设计的实现

4.2.1 小节在讨论元件重用的实现时，就用了层次化设计的方法：顶层文件调用 3 次名为 FreDevider 底层实体和 1 次名为 MUX4 的底层实体。本小节在此基础上，介绍多层次设计的实现。

其实，多层次设计在 MAX+plus II 中的实现十分简单，归结起来就一句话：每个设计实体(包括用 HDL 设计的和用图形编辑器画成的)都可以生成元件符号，供上一层实体调用。

和文本编辑器中生成元件符号一样，在图形编辑器中生成元件符号的操作步骤是：① 将当前文件存储并设为工作文件(File=>Project=>Set Project to Current File)；② 创建元件符号(File=>Create Default Symbol)。图 4-7 对应的元件符号见图 4-11(假设将图 4-7 所在的图形文件存为 HierachyTest.gdf)，那么可以创建一个新图形文件(.gdf 文件)，并在其中调用这个元件符号。如此一层一层地生成元件符号并供上一层设计实体调用，即实现了

图 4-1中所示的多层次设计模型。

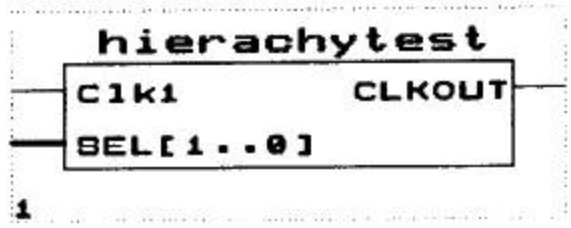


图 4-11　图 4-7 对应的元件符号

经过本章的学习可以发现，在 MAX+plus II 中用图形编辑的方法实现层次化设计，实际上是结合了自顶向下和自底向上两种设计方法。自顶向下即在设计之初画出框图，把所需的子模块以及实现此子模块所需的更底层的模块确定下来，然后用自底向上的方法，先将最底层的模块设计并调试好，然后一层一层向上生成元件符号并供上层文件调用，一直到最顶层文件为止。

第5章

系统层次化设计进阶

第4章介绍了在MAX+plus II中用图形法实现系统层次化设计的方法，本章则主要侧重于用VHDL语言实现系统层次化设计的方法。本章以图5-1的结构为例，介绍元件例化、程序包和类属映射这几个层次化设计中最重要的语句。本章没有深入讨论各语句的语法结构，而是着眼于实际的应用，读者不但可以从实例中掌握各语句的语法结构，而且能深刻体会层次化设计的设计方法及其优点。

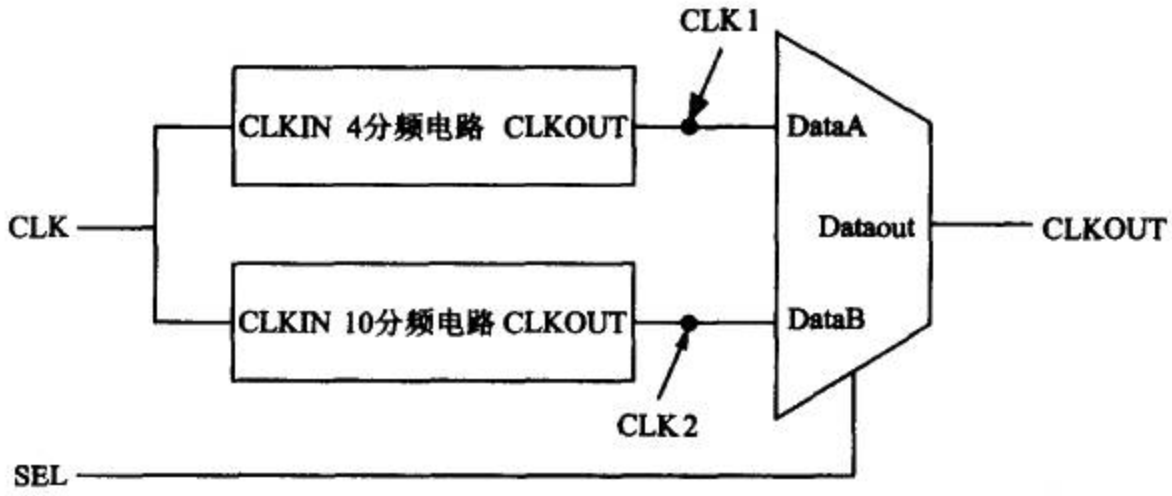


图5-1 时钟选择器结构图

5.1 元件例化

简单地说，元件例化就是将以前设计的实体当作本设计的一个元件，然后用VHDL语句

将各元件之间的连接关系描述出来。

元件的例化语句由两部分组成，第1部分是元件定义，即将现成的设计实体定义为本设计的元件；第2部分是元件连接关系映射语句，即描述各元件之间的连接关系。

元件例化语句的格式如下：

```
COMPONENT 元件名 IS                                              }元件定义部分
  GENERIC(类属表);      --类属映射不是必需的，本节先略过不提
  PORT (端口名表);
END COMPONENT 元件名;
例化名1: 元件名1 PORT MAP( 元件端口名=>连接端口名,…);           }元件映射部分
  ⋮
例化名n: 元件名n PORT MAP( 元件端口名=>连接端口名,…);
```

假设之前已经设计过4分频电路、10分频电路和2路复用器，它们的实体名及输入/输出端口名如表5-1所列。

表5-1　图5-1中各子电路的实体名与输入/输出端口名

电　路	实体名	输入端口	输出端口
4分频电路	FreDevider4	CLKIN	CLKOUT
10分频电路	FreDevider10	CLKIN	CLKOUT
2路复用器	MUX2	DataA、DataB、Sel	Dataout

例程5-1给出了用元件例化语句实现图5-1的电路结构的参考程序，请读者注意元件例化语句的位置及其使用方法。

例程5-1　图5-1电路结构的参考程序

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

ENTITY Hierarchy_Eg IS
PORT
(Clk:IN Std_Logic;
 Sel:IN Std_Logic;
 Clkout:OUT Std_Logic
);
END;
```

```
ARCHITECTURE Structure OF Hierarchy_Eg IS

COMPONENT FreDevider4 IS                    --定义元件 FreDevider4
PORT
(Clkin:IN Std_Logic;
 Clkout:OUT Std_Logic
);
END COMPONENT FreDevider4;

COMPONENT FreDevider10 IS                   --定义元件 FreDevider10
PORT
(Clkin:IN Std_Logic;
 Clkout:OUT Std_Logic
);
END COMPONENT FreDevider10;

COMPONENT MUX2 IS                           --定义元件 MUX2
PORT
(DataA,DataB,Sel:IN Std_Logic;
 Dataout:OUT Std_Logic
);
END COMPONENT MUX2;

SIGNAL Clk1,Clk2:Std_Logic;

BEGIN                                       --元件映射语句在结构体的 BEGIN 和 END 之间使用
  u1:FreDevider4 PORT MAP(Clkin=>Clk,Clkout=>Clk1);
  u2:FreDevider10 PORT MAP(Clkin=>Clk,Clkout=>Clk2);
  u3:MUX2 PORT MAP(DataA=>Clk1,DataB=>Clk2,Sel=>Sel,Dataout=>Clkout);
END;
```

元件例化语句实现了系统的层次化和结构化设计，但有一个缺点，那就是：如果有 N 个上层实体用到了同一个下层实体，那么在这 N 个上层实体的程序中，都必须对该下层实体进行元件定义。另外，如果一个程序中用到了很多元件，那么元件定义语句要占很大篇幅，使程序显得臃肿，降低程序的可读性。

解决上述两个问题的办法就是用程序包 Package。

5.2　程序包

在 VHDL 中，在某一设计实体中定义的数据类型、子程序、数据对象和元件定义对于其他设计实体来说是不可见的。为了使已定义的数据类型、元件定义等能被更多的设计实体共享，避免重复劳动，可将这些定义收集到一个 VHDL 程序包中。这样，只要在设计实体中用 USE 语句调用该程序包，就可以使用这些预定义的数据类型和元件定义等元素了。

定义程序包的语法结构如下：

```
PACKAGE  程序包名 IS
    程序包首说明部分;
END  程序包名;

PACKAGE BODY  程序包名 IS
    程序包体说明部分;
END   程序包名;
```

程序包首的说明部分收集多个不同 VHDL 设计所需的常用公共信息，例如数据类型、元件定义、子程序说明等；而程序包体则包括已在程序包首定义了的子程序的子程序体，没有子程序的程序包体可以省略。

因为本书中未涉及子程序的知识，且本章只关注元件定义的重用，所以下面未给出程序包体说明部分的实例。

例程 5－2 给出了在程序包中进行元件定义的参考程序。

例程 5－2　在程序名中进行元件定义

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

PACKAGE pac_devider IS

    COMPONENT FreDevider4 IS
    PORT
    (Clkin:IN Std_Logic;
     Clkout:OUT Std_Logic
```

```
    );
    END COMPONENT FreDevider4;

    COMPONENT FreDevider10 IS
    PORT
    (Clkin:IN Std_Logic;
     Clkout:OUT Std_Logic
    );
    END COMPONENT FreDevider10;

    COMPONENT MUX2 IS
    PORT
    (DataA,DataB,Sel:IN Std_Logic;
     Dataout:OUT Std_Logic
    );
    END COMPONENT MUX2;
END pac_devider;
```

注意，本程序包应存为 pac_devider. vhd，并编译。只有编译过的程序包才能被其他设计实体调用。

例程 5－3 给出了调用 pac_devider 程序包的时钟选择器程序。

例程 5－3　调用 pac_devider 程序包的时钟选择器

```
LIBRARY IEEE;
USE IEEE. Std_Logic_1164. ALL;
USE Work. Pac_Devider. ALL;      --调用 pac_devider 程序包
ENTITY Hierarchy_Eg IS
PORT
(Clk:IN Std_Logic;
 Sel:IN Std_Logic;
 Clkout:OUT Std_Logic
);
END;

ARCHITECTURE Structure OF Hierarchy_Eg IS
                                 --此处无需再进行元件定义
SIGNAL Clk1,Clk2:Std_Logic;
BEGIN
```

```
    u1:FreDevider4 PORT MAP(Clkin=>Clk,Clkout=>Clk1);
    u2:FreDevider10 PORT MAP(Clkin=>Clk,Clkout=>Clk2);
    u3:MUX2 PORT MAP(DataA=>Clk1,DataB=>Clk2,Sel=>Sel,Dataout=>Clkout);
END;
```

值得注意的是，例程 5－3 调用 pac_devider 程序包的语句是

```
USE Work.Pac_Devider.All;
```

显然 Work 是一个库，那么这个库到底有些什么呢？为什么事先没有用 Library 语句调用呢？事实上，从库名就可以推断出，这是一个用户工作库，其路径一般默认为当前工作目录。此库无需“显式调用”(即不需要用 Library 语句调用)，存在当前工作目录中的一切程序包均默认为用户工作库的一部分。

5.3　类属映射

从分频器的工作原理来说，4 分频电路与 10 分频电路的程序基本相同，只是计数器的上限不同。在前面的设计中，分别设计了一个 4 分频电路和一个 10 分频电路，如果还需要 16 分频、24 分频之类的偶数倍分频电路，难道还要把每个电路都做成一个实体？显然，需要一个更好的办法，那就是设计一个参数化元件。所谓“参数化元件”，顾名思义，就是该元件有某些参数是可调的。通过调整这些参数，可利用一个实体实现结构相似但功能不同的电路。

例程 5－4 给出了参数可调的偶数倍分频器的程序，其中的 N 为分频参数，分频倍数为 $2\times(N+1)$。请读者注意该程序中 GENERIC 语句的位置及其用法。

例程 5－4　参数可调的偶数倍分频器的程序

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

ENTITY FreDevider IS
GENERIC (N:Integer:=4);        --此处定义了一个默认值 N=4,即电路为 10 分频电路;
                               --但当此实体作为上层实体的一个参数化元件时,
                               --N 可由上层实体指定,此默认值失效
PORT
(Clkin:IN Std_Logic;
 Clkout:OUT Std_Logic
);
```

```
END;

ARCHITECTURE Behavior OF FreDevider IS
SIGNAL Counter:Integer RANGE 0 TO N;
SIGNAL Clk:Std_Logic;
BEGIN
  PROCESS(Clkin)
  BEGIN
    IF rising_edge(Clkin) THEN
      IF Counter=N THEN
        Clk<=NOT Clk;
        Counter<=0;
      ELSE
        Counter<=Counter+1;
      END IF;
    END IF;
  END PROCESS;
  Clkout<=Clk;
END;
```

由于此实体使用了类属映射语句 GENERIC，那么在元件定义时，也必须有 GENERIC 定义，如例程 5-5 所示。

例程 5-5　元件定义

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

PACKAGE pac_devider IS
  COMPONENT FreDevider IS
  GENERIC (N:Integer);                    --注意 GENERIC 的位置，此处不必加默认值
  PORT
  (Clkin:IN Std_Logic;
   Clkout:OUT Std_Logic
  );
END COMPONENT FreDevider;
```

```
COMPONENT MUX2 IS
  PORT
  (DataA,DataB,Sel:IN Std_Logic;
   Dataout:OUT Std_Logic
  );
  END COMPONENT MUX2;
END pac_devider;
```

在元件例化时，必须进行类属映射(GENERIC MAP)，见例程 5－6。

例程 5－6　类属映射

```
LIBRARY IEEE;
USE IEEE. Std_Logic_1164. ALL;
USE Work. Pac_Devider. ALL;
ENTITY Hierarchy_Eg IS
PORT
  (Clk:IN Std_Logic;
   Sel:IN Std_Logic;
   Clkout:OUT Std_Logic
  );
END;

ARCHITECTURE Structure OF Hierarchy_Eg IS
SIGNAL Clk1,Clk2:Std_Logic;
BEGIN
  u1:FreDevider GENERIC MAP(N=>1) PORT MAP(Clkin=>Clk,Clkout=>Clk1);
  u2:FreDevider GENERIC MAP(N=>4) PORT MAP(Clkin=>Clk,Clkout=>Clk2);
  u3:MUX2 PORT MAP(DataA=>Clk1,DataB=>Clk2,Sel=>Sel,Dataout=>Clkout);
END;
```

请读者注意例程 5－6 中 GENERIC MAP 的位置及其格式。事实上，因为本例中的类属参数只有一个(*N*)，所以 u1 和 u2 的类属映射可以直接写成：

```
GENERIC MAP (1) 和 GENERIC MAP(4);
```

但若一个参数化元件中存在多个参数，则应将映射关系写清楚。


惑篇

第6章

初学者常见问题解答

万事开头难，VHDL 的学习也不例外。初学者常常对一些问题感到困惑，这种困惑会在长期的实践中逐渐淡去，但这个过程往往是痛苦而漫长的。本章以问答的形式解答 VHDL 初学者的一些常见问题，希望能减少初学者的困惑，帮助刚入门的 VHDL 使用者较快地理解和掌握 VHDL 的编程思想。此外，本章的 6.5 节对初学者使用 MAX+plus II 时的几个常见问题进行了解答。

6.1 关于设计方法

问：用元件例化方式完成系统的层次化设计好，还是用图形连接方式好？

答：图 6－1 是 3 个 VHDL 模块在 Max+PlusII 中的连线图，例程 6－1 则是此连线图对应的 VHDL 程序（用 Component 语句将 3 个模块元件例化后调用）。哪种方式更直观呢？显然，图形连接比元件例化更直观，可读性强。

如果系统较大，初学者在使用元件例化时很容易将各模块之间连线的关系搞混，不但使设计系统的速度变慢，而且也不利于查错。因此建议初学者在设计系统时，各个功能模块的设计用 VHDL 实现，各个模块之间的连接用图形方式。本书中所有实例均采用这种方式。

```
        ScanSignal:OUT Std_Logic_Vector(0 TO 3));
    END COMPONENT;
  COMPONENT AntiTwitter
    PORT(Clock:IN Std_Logic;
         Numin:IN Std_Logic_Vector(3 DOWNTO 0);
         Numout:OUT Std_Logic_Vector(3 DOWNTO 0));
    END COMPONENT;
  SIGNAL Clk:Std_Logic;
  SIGNAL NumTemp:Std_Logic_Vector(3 DOWNTO 0);
  BEGIN
    u0:Frequency PORT MAP(Clock,Clk);
    u1:Keyboard PORT MAP(Clk,KIN,NumTemp,ScanSignal);
    u2:AntiTwitter PORT MAP(Clk,Numtemp,Num);
  END;
```

6.2　关于信号与变量

问:信号与变量有什么区别?

答:信号与变量的区别在第2章中已有详细论述,这里对前面的内容进行概括,并进行适当补充。

① 声明形式与赋值符号不同。变量声明为Variable,赋值符号为“:=”;而信号声明为Signal,代入语句采用“<=”代入符。

② 有效域不同。信号在结构体内、进程外定义,而变量在进程内定义。换句话说,信号的有效域为整个结构体,可在不同进程间传递数值;变量的有效域只是定义该变量的进程,不能为多个进程所用。

③ 赋值操作的执行不同。**在进程中,变量赋值语句一旦被执行,目标变量立即被赋予新值,在执行下一条语句时,该变量的值为上一句新赋的值;而信号的赋值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(信号是在进程挂起时才发生代入的)。**

④ 应用场合不同。在实际应用中,信号的行为更接近硬件的实际情况,因此应该更多地使用信号进行电路内部数据传递,只有在描述一些用信号很难描述的算法时才用到变量(也有人建议在同一进程内部传输的数据均采用变量表示,只有在需要进程间传递数据时才使用信号)。但在几种情况下,则必须采用变量,下面介绍两种最典型的情况。

● 在LOOP语句中，若在一个循环体内需要对某一个数据进行多次操作，则必须用变量，因为对信号的多次赋值只有最后一次会生效。例如：

```
VARIABLE Sum:Integer RANGE 0 TO 127;
      ⋮
FOR i IN 1 To 9 LOOP
   Sum:=Sum+i;    --此处千万不要写成 Sum<=Sum+i;否则只有当 i=9 时,才执行赋值操作
END LOOP;
```

上面这个程序中的Sum如果用信号，那么是实现不了1～9的累加计算的。因为对同一信号进行多次赋值时，只有最后一次生效，也就是说，只有在 $i=9$ 的时候才生效，最后的结果是Sum=9。

下面这个程序则是可行的：

```
For i IN 1 To 15 LOOP
  Data(i-1)<=Data(i);
End LOOP;
```

这是因为在 i 不断自加的时候，赋值对象Data($i-1$)是不断改变的，并不会出现对同一个信号被多次赋值的情况。

● 数组的索引(Index)只能用变量。例程6-2是一个简单的并-串转换程序，请注意该程序中的数组索引。

例程6-2　简单的并-串转换程序(PISO-Parallel In Serial Out)

```
LIBRARY IEEE;
USE IEEE. Std_Logic_1164. All;

ENTITY PISO IS
PORT
(Reset,Clock:IN Std_Logic;
 ParallelNum:IN Std_Logic_Vector(31 DOWNTO 0);
 SerialOut:OUT Std_Logic);
END;

ARCHITECTURE Convert OF PISO IS
BEGIN
  PROCESS(Clock)
  VARIABLE i:Integer RANGE 0 TO 31;
```

```
BEGIN
  IF Reset='1' THEN
    i:=0;
  ELSIF rising_edge(Clock) THEN
    SerialOut<=ParallelNum(i);        --变量 i 作数组 ParallelNum 的 index
                                      --i 如果为信号,那么编译时会报错
    IF i<31 THEN
      i:=i+1;
    END IF;
  END IF;
END PROCESS;
END;
```

问:关于信号的赋值,还有什么需要注意的?

答:最重要的一点就是,可在不同的进程中读取一个信号,**但不能在不同的进程中对同一个信号进行赋值操作**。此外,在同一进程中对同一信号进行多次赋值操作时,只有最后一次会生效。如果对一个信号进行有条件地赋值(IF_THEN_ELSE),要注意条件的互斥性。

例如:

```
IF A=4 THEN
  S<=1;
END IF;
IF A=5 THEN
  S<=2;
END IF;
```

这样写后,Max+PlusII 在编译时会报警,因为这种写法不够严格。那么把上面的程序修改如下:

```
IF A=4 THEN
  S<=1;
ELSIF A=5 THEN
  S<=2;
END IF;
```

用 ELSIF 将(A=4)和(A=5)这两个条件互斥后,编译正常通过。从这个小例子可以看出,VHDL 语法严格到了苛刻的地步,但苛刻的好处就是,让使用者养成严谨的思维习惯,尽量减少犯错。

6.3　关于顺序语句的顺序性

问：顺序语句与并行语句如何区分？为什么书上说有些语句是顺序语句，可是执行以后的结果却表明它们是并发执行的？

答：所谓顺序语句，即从仿真的角度看，每一条语句的执行是按书写顺序进行的。顺序语句只能在进程和子程序（函数和过程）内部使用。

VHDL有6类基本顺序语句：① （变量）赋值语句；② 流程控制语句；③ 等待语句；④ 子程序调用语句；⑤ 返回语句；⑥ 空操作语句。

常用的顺序语句有IF_THEN_ELSE、CASE_WHEN和LOOP_FOR等。

并行语句是硬件描述语言与一般软件程序的最大区别所在，所有并行语句在结构体中的执行都是同时进行的，即它们的执行顺序与语句书写的先后顺序无关。这种并行性是由硬件本身的并行性所决定的，即一旦电路接通电源，它的各部分就会按照事先设计好的方案同时工作。

VHDL有以下几种主要并行语句：① 进程语句；② 块语句；③ 并行信号赋值语句；④ 元件例化语句；⑤ 生成语句；⑥ 并行过程调用语句。

应该注意的是，把进程语句（Process）当作并行语句，是因为进程与进程间的执行是并行的（一个Architecture里的不同Process是同时执行的）。但进程内部是顺序执行的，读者千万不要因为进程语句是并行语句就误以为其内部也是并行执行的。

有个问题常常困扰初学者，那就是有些顺序语句在实际使用时却表现出并行特性。我们先来看下面这两个进程。

例程6-3　表现出并行特性的顺序语句

```
--注意，进程1和进程2并不在同一个程序中，
--此处只是为了方便比较，将其放在一起

进程1：
Process(Reset,Clock)
Begin
  If Reset='1' then                    --对A、B赋初值
    A<=1;
    B<=0;
```

```
  Elsif rising_edge(Clock) then
   If A=1 then                    --语句(1)
      B<=4;
   End If;
   If B=4 then                    --语句(2)
      A<=5;
   End If;
  End If;
End Process;

进程 2:
Process(Reset,Clock)
Begin
  If Reset='1' then
    A<=1;
    B<=0;
  Elsif rising_edge(Clock) then
    If B=4 then
      A<=5;
    End If;
    If A=1 then
      B<=4;
    End If;
  End If;
End Process;
```

很容易看出，进程 2 和进程 1 几乎一样，只是将进程 1 的语句(1)和语句(2)颠倒了。由于 IF_THEN_ELSE 语句是顺序语句，按照定义，其语句的书写顺序会影响执行的结果，即进程 1 与进程 2 应当产生不一样的结果。但是经过仿真(如图 6-2 所示)会发现，进程 1 与进程 2 的结果是一模一样的(如图 6-3 所示)。

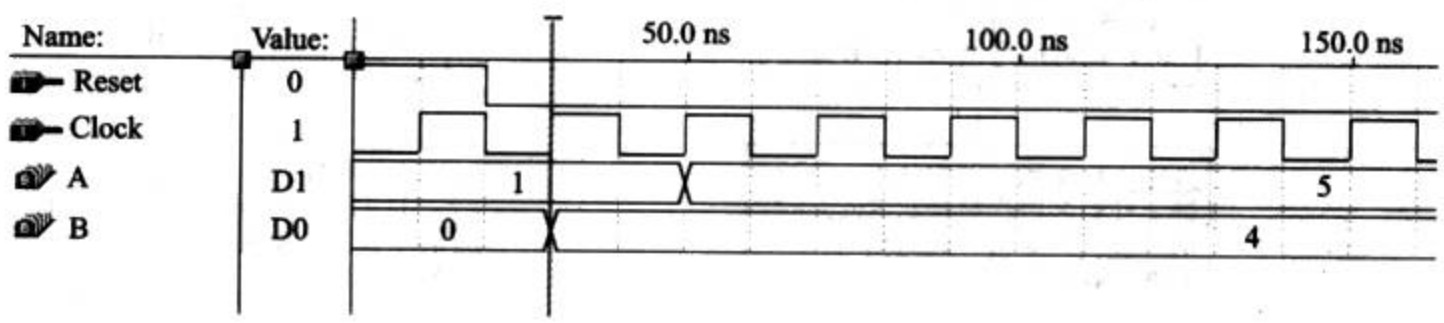
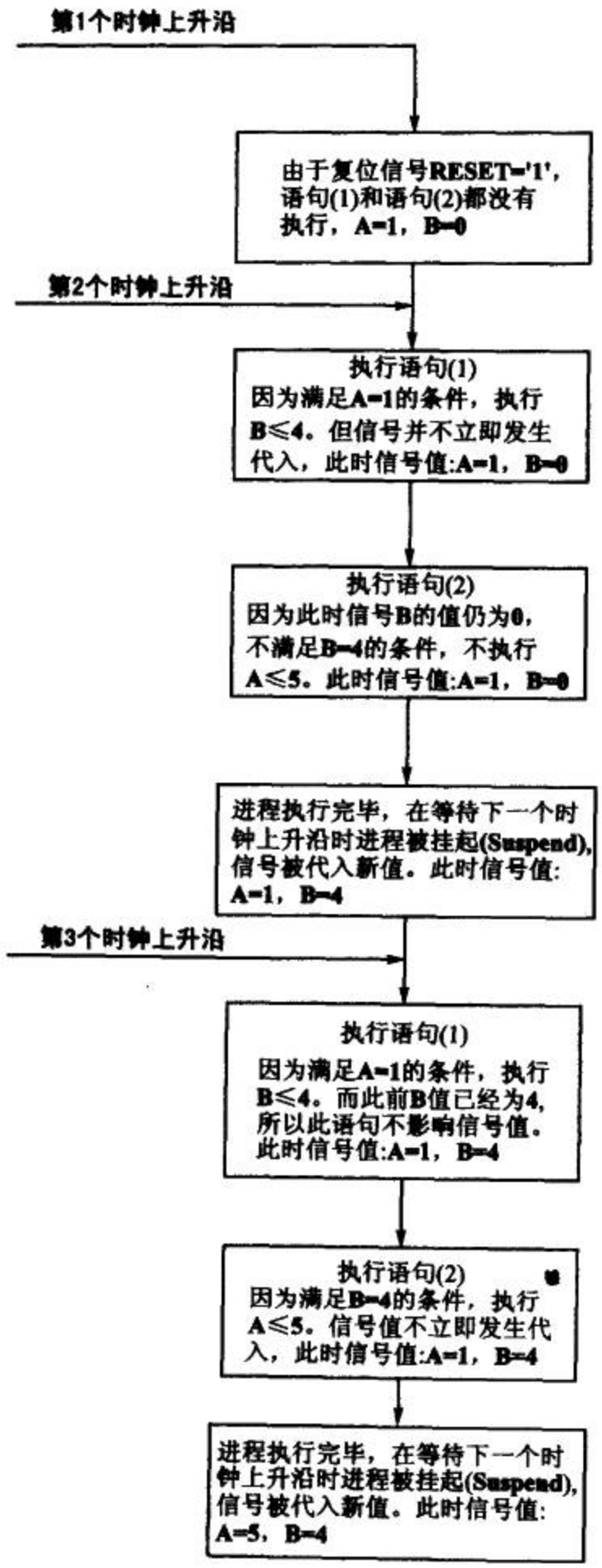
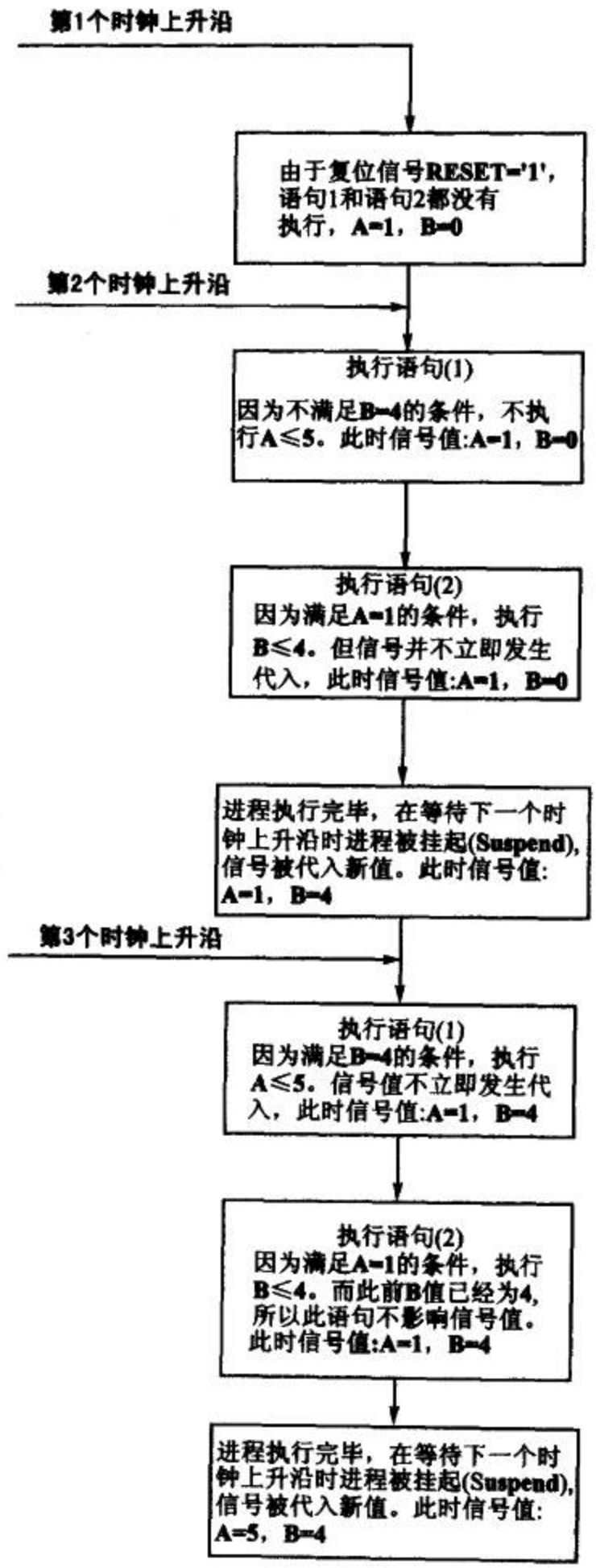


图 6-2　例程 6-3 的仿真图



(a) 进程1执行流程



(b) 进程2执行流程

图 6－3　例程 6－3 的执行过程

这是为什么呢？在讨论信号与变量的区别时曾提到：**信号的赋值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。**

图 6-3 的流程图说明了信号赋值语句的这种特性，请对照程序、仿真图与流程图仔细地体会信号这一硬件描述语言特有的数据对象。

从大量的实践中可以总结出一条规律：一个进程(Process)中，如果没有变量的存在，那么仍可将进程内的语句视为并行执行，也就是说，语句的书写顺序并不影响结果；如果进程存在变量，那么进程呈现顺序性，但信号的代入语句仍是在进程被挂起(Suspend)时同时执行(并行执行)的。这一点请读者牢记。读者可以自己写几个变量与信号混杂的进程进行仿真，通过仿真结果来理解变量与信号的关系。

6.4　关于仿真与综合

问：书上说有些 VHDL 语句是不可综合的，那么可综合的语句有哪些？

答：不同的综合器支持的语句不同，一般在该综合器软件的 HELP 中可找到它的可综合语句集。对 MAX+plus II 的用户，可以在 HELP=>VHDL=>MAX+PLUS II VHDL Support 中看到此软件对 VHDL 的支持情况。

问：在定义信号和变量时都给它们赋了初值，为什么结果与预期的不同？

答：信号和变量的初值对综合器而言是没有意义的，会被忽略掉，大部分综合器会把信号和变量的初值都设为数值 0 或逻辑 0(有例外，见下一问)。如果需要一个初值非 0 的信号或变量，那么需要写一段程序对信号或变量进行初始化(可以参考“实例篇”中自动打铃系统的打铃长度设定模块的设计)。

问：有些书上说信号不管赋什么初值，综合后都会被清 0。为什么有时候会出现初值非 0 的情况？

答：这跟综合器的设置有关系。若在 MAX+plus II 中出现这种情况，则可按以下步骤修改综合器选项：Assign=>Global Project Logic Synthesis=>Define Synthesis Style=>Advanced Options，将 NOT Gate Push-Back 选项去掉。当然，更好的做法是在每个程序中都对信号进行显式初始化。

问：为什么系统的仿真结果是正确的，但在实际电路中总是出错？

答：这种情况往往是由于初学者对外围电路的处理不当引起的。作者不止一次看见学生未在按键上接上拉电阻或下拉电阻，这导致 CPLD/FPGA 读取的键值不稳定而引起整个系统的不

稳定。在处理外围电路时，一定要注意上、下拉电阻的作用，只在程序中查错是远远不够的。

6.5　关于 MAX+plus II

问：在使用并口下载线时，为什么在 Windows98 下工作正常，在 Windows2000 或 XP 中却不能下载程序？

答：在 Windows2000 和 XP 中需要添加驱动程序。默认驱动程序路径为\maxplus2\Drivers\Win2000。安装此驱动的步骤如下：

① 在控制面板中选择“添加硬件”；
② 在随后出现的添加硬件向导的对框话中选择“是，硬件已连接好”；
③ 选择“添加新的硬件设备”；
④ 选择“安装我手动从列表选择的硬件(高级)”；
⑤ 选择“声音、视频和游戏控制器”；
⑥ 选择“从磁盘安装”，并指明驱动程序所在的路径。

问：为什么在 MAX+plus Ⅱ 中找不到所使用的器件？

答：MAX+plus II 支持的器件有限，有些最新的芯片(如 Cyclone 等)只能在 Quartus II 中使用。若找到了所要的芯片型号，但是速度不匹配(例如需要速度标识为－15 的 MAX7128，但可选项只有－6 的芯片)，则在选择芯片时，将 Show Only Fastest Speed Grades 复选框中的勾号去掉即可。

问：用 MAX+plus Ⅱ 仿真时，如何将仿真结果从默认的二进制改为十进制显示？

答：如图 6－4 所示，在 Value 一栏双击，就会出现图 6－5 所示的窗口，BIN 为二进制，DEC 为十进制，OCT 为八进制，HEX 为十六进制。值得注意的是，当所选信号为 Std_Logic 或 Bit 时，此选项是无效的。

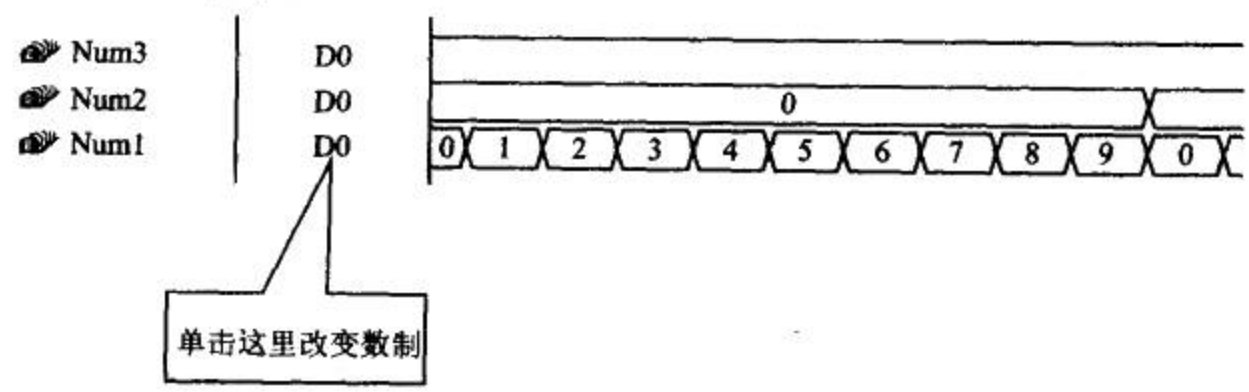


图 6－4　改变显示结果的数制

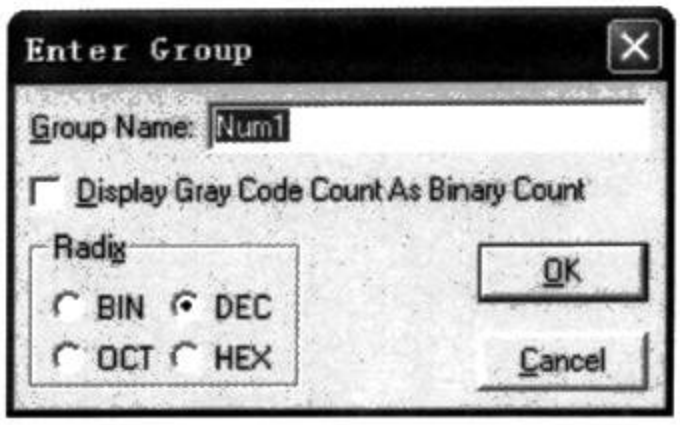


图 6-5　改变数制对话框

问：对 MAX 系列的 CPLD，在综合设置上有什么需要注意的？

答：在 Global Project Logic Synthesis（在 Assign 菜单中）的 MAX Device Synthesis Options中，默认设置为 Multi-Level Synthesis for MAX 9000 Devices（见图 6-6），而一般高等院校使用的是 MAX7128S，所以需要将其改为 Multi-Level Synthesis for MAX 3000/5000/7000 Devices，否则综合后占用的资源将大大增加。

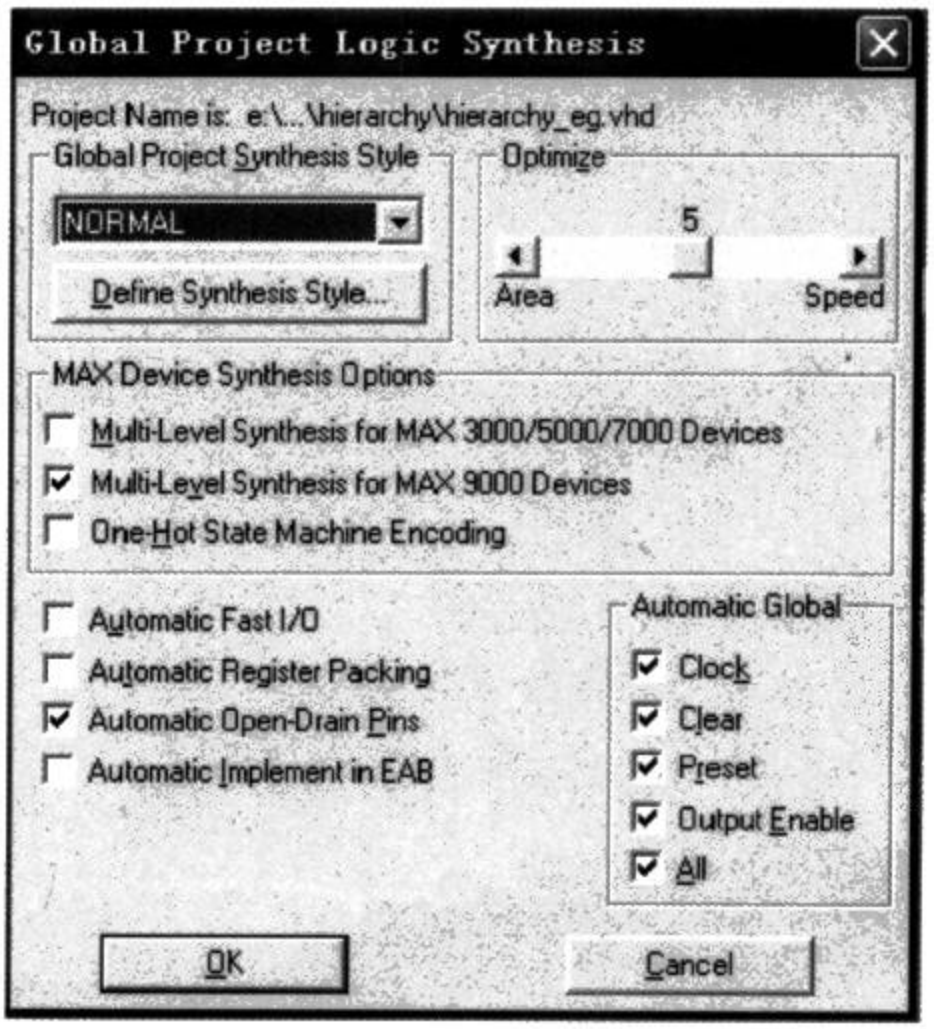


图 6-6　逻辑综合参数设置对话框


例篇

第7章

常用电路的 VHDL 程序

本章给出了一些常用电路的 VHDL 程序，这些电路在后面的章节中可能会经常用到。初学者亦可通过这些简单的练习，熟悉 VHDL 程序的基本结构和一些常用的 VHDL 语句。

7.1 分频电路

CPLD/FPGA 与单片机相比，一个非常明显的优势就在于它的高速性。但是因为很多外围器件的驱动需要低频的时钟(若时钟频率太高，则键盘扫描容易出错，七段数码管会闪烁和不稳定等)，所以常常需要用到分频电路。常用的分频电路又分为偶数倍分频和奇数倍分频两种。

7.1.1 偶数倍分频

偶数倍分频的原理十分简单，相信读者在看完例程 7-1 和相应的仿真波形(图 7-1)后就能明白其中的道理，此处不再详述。

例程 7-1　偶数倍分频电路(8 分频)

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

ENTITY FreDevider IS
PORT
  (Clkin: IN Std_Logic;
   Clkout:OUT Std_Logic
  );
END;

ARCHITECTURE Devider OF FreDevider IS
CONSTANT N:Integer:=3;
SIGNAL Counter:Integer RANGE 0 TO N;
SIGNAL Clk:Std_Logic;
BEGIN
  PROCESS(Clkin)
  BEGIN
    IF rising_edge(Clkin) THEN
      IF Counter=N THEN      --每计到4个(0~3)上升沿,输出信号翻转一次
        Counter<=0;
        Clk<=NOT Clk;
      ELSE
        Counter<=Counter+1;
      END IF;
    END IF;
  END PROCESS;
  Clkout<=Clk;
END;
```

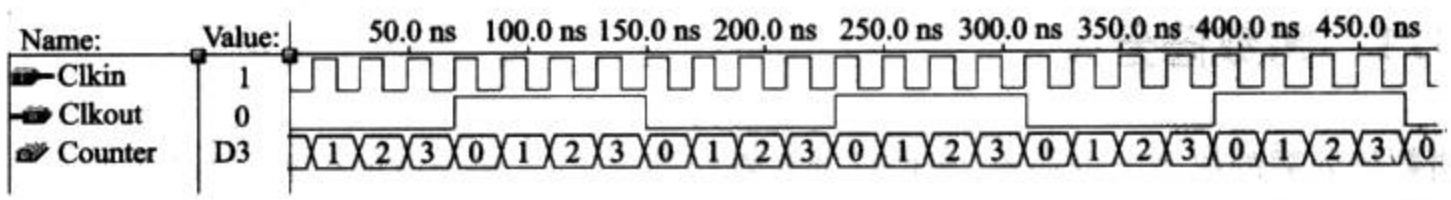


图 7-1　8 分频电路(上升沿翻转)的仿真波形

例程7-1给出的是一个8分频电路，其他倍数的分频电路可以通过修改Counter的上限值N得到。一般的计算规则是：对一个$2x$分频的电路来说，Counter上限值$N=x-1$(从0计到$x-1$恰好为x次，每x个上升沿翻转一次就实现了$2x$分频)。

上面给出的是上升沿翻转的分频电路，如果希望产生一个下降沿翻转的分频电路，只须将例程中的rising_edge(Clkin)改成falling_edge(Clkin)即可。图7-2是下降沿翻转的8分频电路的仿真波形。

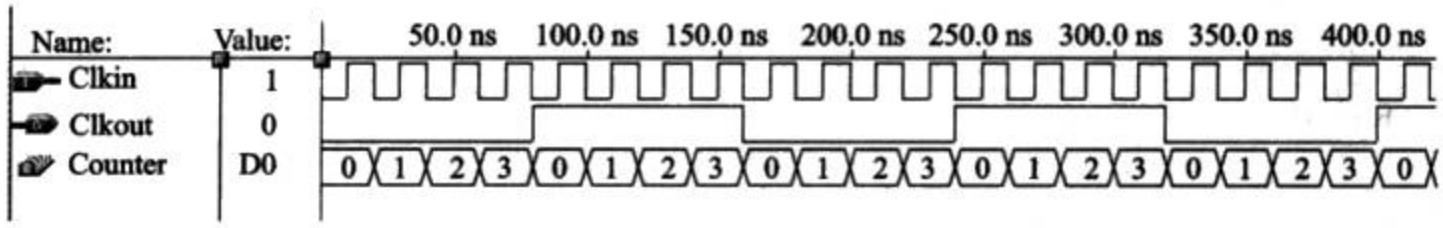


图7-2　8分频电路(下降沿翻转)的仿真波形

7.1.2　奇数倍分频

奇数倍分频要比偶数倍分频复杂，实现奇数倍分频的方法不是惟一的，但最简单的是错位"异或"法，请读者通过例程7-2和图7-3理解这种方法的思想，并试着将其修改为其他倍数(奇数倍)的分频电路。

例程7-2　奇数倍分频电路(3分频)

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

ENTITY Fredevider3 IS
PORT
(Clock:IN Std_Logic;
 Clk:OUT Std_Logic
);
END;

ARCHITECTURE Devider OF Fredevider3 IS
SIGNAL Counter:Integer RANGE 0 TO 2;
SIGNAL Temp1,Temp2:Std_Logic;

BEGIN
```

```
PROCESS(Clock)
BEGIN
    IF rising_edge(Clock) THEN
      IF Counter=2 THEN
        Counter<=0;
        Temp1<=NOT Temp1;
      ELSE
        Counter<=Counter+1;
      END IF;
    END IF;

    IF falling_edge(Clock) THEN
      IF Counter=1 THEN
        Temp2<=NOT Temp2;
      END IF;
    END IF;
END PROCESS;
Clk<=Temp1 XOR Temp2;
END;
```

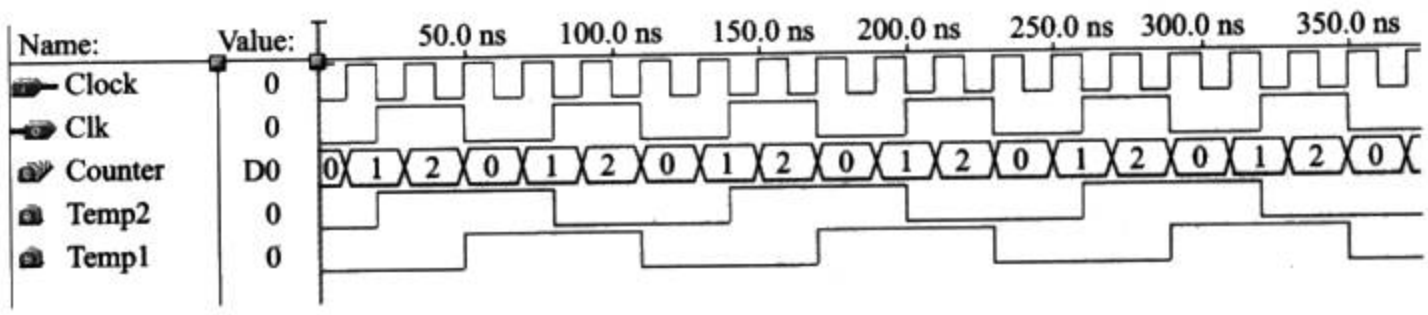


图7-3　3分频电路的仿真波形

7.2　七段数码管驱动电路

七段数码管分为共阴极和共阳极两种。简而言之，对共阴极来说，公共引脚要接地，想要点亮某段数码管，就在相应的引脚加上高电平；对共阳极来说刚好相反，公共引脚提高电平，想要点亮某段数码管，就在相应的引脚加上低电平。七段数码管的分段与相应的引脚图见图7-4和图7-5。

如图7-5所示，七段数码管共有10个引脚，分别对应七段发光二极管(a～g)和一个指示

小数点的发光二极管(dp),另外还有两个公共端K(对共阴极数码管来说该端接地,而对共阳极来说则该端接高电平)。

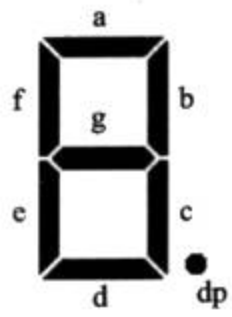


图7-4　七段数码管各段对应位置

图7-5　七段数码管引脚图

七段数码管的驱动电路有并行和串行两种接法。

7.2.1　并行连接的七段数码管驱动程序

并行连接,即每个数码管都由单独的译码电路控制,各数码管之间除地端GND接在一起外,其余引脚各不相关。并行法的优点是控制简单,有几个数码管就用几个译码电路,不必修改程序,十分简便。但当系统所需数码管较多时,这种方法既耗资源,又占用较多的I/O口,N个数码管需占用$7N$个引脚(若需要小数点,则是$8N$个引脚)。因此,此接法适合于系统中数码管数量不多的应用场合。

例程7-3为最基本的译码电路的VHDL程序,用选择信号赋值语句描述,将综合成组合逻辑电路(以下的数码管驱动程序中,均未使用小数点dp,读者若在应用中需要使用小数点,则请参考例程自行修改)。

例程7-3　最基本的译码电路程序

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

ENTITY Display IS
PORT
(Num:IN Integer RANGE 0 TO 15;
Display:OUT Std_Logic_Vector(6 DOWNTO 0)
);
END;

ARCHITECTURE Encoder OF Display IS
```

```
BEGIN
  WITH Num SELECT
    Display<="1111110" WHEN 0,
             "0110000" WHEN 1,
             "1101101" WHEN 2,
             "1111001" WHEN 3,
             "0110011" WHEN 4,
             "1011011" WHEN 5,
             "1011111" WHEN 6,
             "1110000" WHEN 7,
             "1111111" WHEN 8,
             "1111011" WHEN 9,
             "0000000" WHEN OTHERS;
END;
```

读者会发现，上面这个例程预留了一种情况，即当 Num 不是 0～9 中的数值时，数码管的各段发光二极管将熄灭，数码管无任何显示。与译码电路输入端相接的模块可以利用此特点来控制数码管的亮、灭。但有时为了方便其他模块对译码电路的控制（如图 7－6），希望在译码电路上加上使能端，例程 7－4 给出了带使能端的译码器的 VHDL 程序。

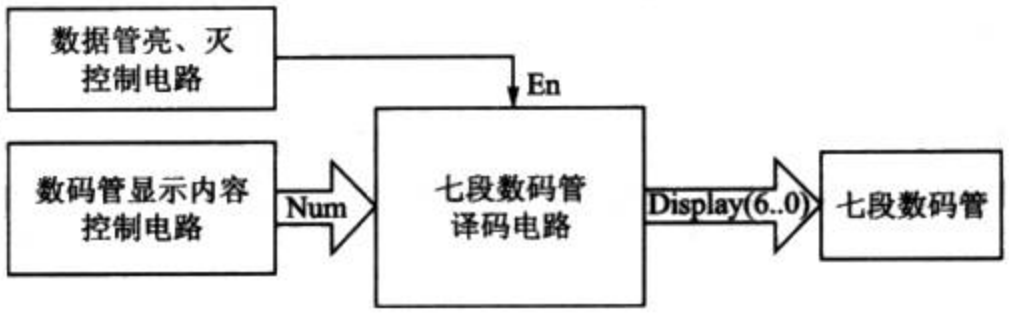


图 7－6　多个模块共同控制七段数码管译码电路示意图

例程 7－4　带使能端的译码器程序

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

ENTITY Display IS
PORT
(En:IN Std_Logic;
 Num:IN Integer RANGE 0 TO 15;
 Display:OUT Std_Logic_Vector(0 TO 6)
);
END;
```

```
ARCHITECTURE Decoder OF Display IS
BEGIN

  PROCESS(En,Num)
  BEGIN
    IF En='1' THEN
      CASE Num IS
        WHEN 0=>Display<="1111110";
        WHEN 1=>Display<="0110000";
        WHEN 2=>Display<="1101101";
        WHEN 3=>Display<="1111001";
        WHEN 4=>Display<="0110011";
        When 5=>Display<="1011011";
        WHEN 6=>Display<="0011111";
        WHEN 7=>Display<="1110000";
        WHEN 8=>Display<="1111111";
        WHEN 9=>Display<="1110011";
        WHEN OTHERS=>Display<="0000000";
      END CASE;
    ELSE
      Display<="0000000";
    END IF;
  END PROCESS;
END;
```

例程 7－4 中，信号 En 为使能信号。En 为高电平时，译码器正常工作；En 为低电平时，译码器输出 0000000，表示数码管无显示。值得注意的是，例程 7－4 综合的结果是组合逻辑电路，Process 的敏感信号参数表中一定要有 Num；否则编译时会提示如下出错信息："Else Clause following a Clock edge must hold the state of signal 'Display'"。

出现此提示信息的原因是：综合器将 En 误判为时钟信号，并试图将程序综合成时序逻辑电路，但该程序的格式又不附合综合器对时钟信号描述的要求，因此无法综合。

关于 Process 综合结果的讨论，请参见第 14 章。

在某些场合下希望数码管带有闪烁的效果，例程 7－5 给出了带闪烁功能的译码电路的 VHDL 程序。

例程 7－5　带闪烁功能的译码电路程序

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

ENTITY Display IS
PORT
( Clock:IN Std_Logic;
 Flash:IN Std_Logic;
 Qin:IN Std_Logic_Vector(3 DOWNTO 0);
 Display:OUT Std_Logic_Vector(0 TO 6)
);
END;

ARCHITECTURE Decoder OF Display IS
SIGNAL Timeout:Integer RANGE 0 TO 63;
BEGIN
  PROCESS(Clock)
  BEGIN
    IF rising_edge(Clock) THEN
      IF (Flash='0') THEN
        Timeout<=0;
    ELSE
      IF (Timeout=63) THEN      --调整此值即可调整数码管闪烁的频率
        Timeout<=0;
      ELSE
        Timeout<=Timeout+1;
      END IF;
    END IF;
    IF (Timeout<31) THEN        --调整此值即可调整数码管亮、灭的时间比
      CASE Qin IS
        WHEN "0000"=>Display<="1111110";
        WHEN "0001"=>Display<="0110000";
        WHEN "0010"=>Display<="1101101";
        WHEN "0011"=>Display<="1111001";
        WHEN "0100"=>Display<="0110011";
        WHEN "0101"=>Display<="1011011";
        WHEN "0110"=>Display<="0011111";
```

```
            WHEN "0111"=>Display<="1110000";
            WHEN "1000"=>Display<="1111111";
            WHEN "1001"=>Display<="1110011";
            WHEN OTHERS=>Display<="0000000";
        END CASE;
      ELSE
        Display<="0000000";
      END IF;
    END IF;
  END PROCESS;
END;
```

例程7-5中,数码管闪烁的频率由Clock的频率与计数器Timeout的值共同决定。此程序中输入数字用Std_Logic_Vector(3 Downto 0)表示,与前两个例程中的Integer Range 0 To 15是等价的(看一看这几个例程生成的Symbol文件,会发现Num和Qin的接口其实是一样的)。

用Integer表示的好处在于其直观、明了,但编程时不容易看出实际使用数字的范围;而用Std_Logic_Vector表示则更接近实际电路,也可以使编程者对实际使用数字的范围有清楚的认识。例如,Integer Range 0 To 10和Integer Range 0 To 15实际上对应的都是Std_Logic_Vector(3 Downto 0),然而初学者往往忽视这一点,导致在条件判断语句中对条件覆盖不全等情况。因此,有不少VHDL的教科书都告诫使用者,尽量少用Integer类型。事实上,使用Integer可能更符合人们的计数习惯,使程序有更好的可读性。只要在使用时,注意其与实际电路的对应关系,就可避免犯错。

7.2.2 串行连接的七段数码管驱动程序

串行连接,即每个数码管对应的引脚都接在一起(如每个数码管的a引脚都接到一起,然后再接到CPLD/FPGA上的一个引脚上),通过控制公共端(图7-5中的K端)控制相应数码管的亮、灭(共阴极数码管的公共端为高电平时,LED不亮;共阳极的公共端为低电平时,LED不亮)。

串行法的优点在于消耗的系统资源少,占用的I/O口少,N个数码管只需要$(7+N)$个引脚(如果需要小数点,则是$(8+N)$个引脚)。其缺点是控制起来不如并行法容易。

下面给出一个串行连接的七段数码管驱动程序,此例中使用了6个数码管。

例7-6 串行连接的七段数码管驱动程序

```
LIBRARY IEEE;
USE IEEE.Std_Logic_1164.ALL;
USE IEEE.Std_Logic_Unsigned.ALL;

ENTITY Display IS
PORT( Clock:IN Std_Logic;
      NumA,NumB,NumC,NumD,NumE,NumF:IN Integer RANGE 0 TO 9;
      En:OUT Std_Logic_Vector(0 TO 5);      --分别接6个数码管的公共端
      Display:OUT Std_Logic_Vector(0 TO 6)  --接数码管的7个控制端
    );
END;

ARCHITECTURE Decoder OF Display IS
SIGNAL Counter:Integer RANGE 0 TO 5;
BEGIN
  PROCESS(Clock)
  VARIABLE Num:Integer RANGE 0 TO 9;
  BEGIN
 IF rising_edge(Clock) THEN
  IF Counter=5 THEN
    Counter<=0;
  ELSE
    Counter<=Counter+1;
  END IF;
  CASE Counter IS
    WHEN 0=>
      En<="011111";          --点亮第1个数码管,屏蔽其他5个数码管
      Num:=NumA;             --显示第1个数
    WHEN 1=>
      En<="101111";          --点亮第2个数码管
      Num:=NumB;             --显示第2个数
    WHEN 2=>
      En<="110111";
      Num:=NumC;
    WHEN 3=>
      En<="111011";
```

```
            Num:=NumD;
          WHEN 4=>
            En<="111101";
            Num:=NumE;
          WHEN 5=>
            En<="111110";
            Num:=NumF;
          WHEN OTHERS=>
            En<="000000";
            Num:=0;
        END CASE;
        CASE Num IS
          WHEN 0=>Display<="1111110";
          WHEN 1=>Display<="0110000";
            ⋮（译码部分略）
          WHEN 9=>Display<="1110011";
          WHEN OTHERS=>Display<="0000000";
        END CASE;
      END IF;
    END PROCESS;
END;
```

带闪烁功能的串行连接的七段数码管的驱动程序留给读者作为练习。

7.3　键盘扫描电路

数字系统中，常用的按键有直接式和矩阵式两种。直接式按键十分简单，一端接 V_{cc}，一端接 CPLD/FPGA 或单片机的 I/O 口(设为输入)。当按键按下时，此接口为高电平，通过对 I/O 口电平的检测就可知按键是否按下。其优点是简单、易行，连接方便，但每个按键要占用一个 I/O 口，如果系统中需要很多按键，那么用这种方法会占用大量的 I/O 口。而矩阵式键盘控制比直接式按键要麻烦得多，但其优点也是很明显的，即节省 I/O 口。设矩阵式键盘有 m 行 n 列，则键盘上有(mn)个按键，而它只需要占用($m+n$)个 I/O 口。当需要很多按键时，用矩阵式键盘显然比直接式按键要合理得多。

CPLD/FPGA 系统中矩阵式键盘的接法通常有两种(其余的接法一般都是由这两种接法稍加改变而得，基本原理是完全一样的)，下面将一一介绍。

从未接触过矩阵键盘的读者，初看图7-7可能觉得有点乱，其实这个接法总共就4行4列共8条线(4条行线接PC7～PC4，4条列线接PC3～PC0)。经过认真观察，发现每个按键都连着一条行线和一条列线，可以选按键0作为研究对象。按键0的行线连着PC4，而且通过一个下拉电阻接地(下拉就是将不确定的信号通过一个电阻钳位在低电平，上拉反之)；其列线接着PC3。将PC3～PC0设为输出，PC7～PC4设为输入。试想，如果PC2～PC0输出0，而PC3输出1，当按键0按下时，PC7～PC4会读到什么值呢？很显然，因为PC7～PC5没有输入，会由于下拉电阻的下拉作用稳定在低电平，而PC4则由于与PC3接通而呈现高电平。也就是说，当一个按键的行线为1时，如果此按键按下，则列线读到的值为1，否则为0。换句话说，当PC3～PC0为1000时，当按键0按下时，PC7～PC4读到的电平值为0001。如果是按键1按下呢？那么PC7～PC4读到的电平值就是0010。以此类推，可以列出一张行列电平值与按键的对应关系表(见表7-1)。

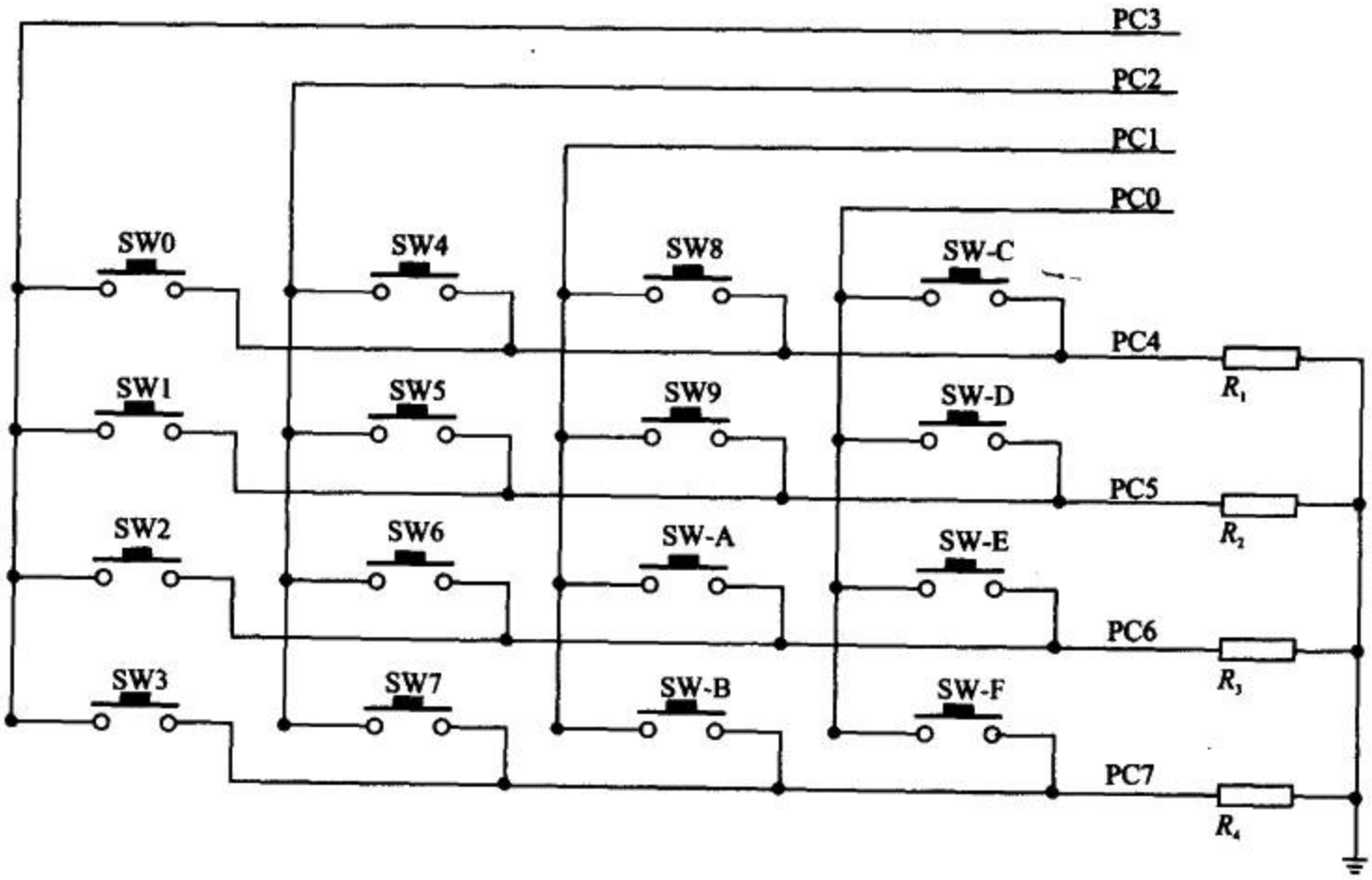


图7-7　矩阵键盘接法A

表7-1　行列电平值与按键的对应关系

列/PC3～PC0(输出)	行/PC7～PC4(输入)	按　键
1000	0001	0
1000	0010	1
1000	0100	2

续表 7-1

列/PC3～PC0(输出)	行/PC7～PC4(输入)	按 键
1000	1000	3
0100	0001	4
0100	0010	5
0100	0100	6
0100	1000	7
0010	0001	8
0010	0010	9
0010	0100	A
0010	1000	B
0001	0001	C
0001	0010	D
0001	0100	E
0001	1000	F

从表 7-1 看出,PC3～PC0 的输出在任意时刻总是只有一条线为 1,其他线均为 0。这很容易理解,若 PC3～PC0 的输出在同一时刻有多于一条线为 1,则无法准确判断究竟是哪个键被按下。例如 PC3 和 PC2 同时为 1,当按键 0 或者按键 4 被按下,PC7～PC4 读到的值都是 0001,这样根本无法判断究竟是按键 0,还是按键 4 被按下。

因为无法预计什么时候有键按下,也无法预测究竟是哪一列上的键被按下,所以只能对键盘的列线(PC3～PC0)进行扫描,同时读取键盘行线(PC7～PC0)的电平值。如表 7-2 所列,PC3～PC0 按下述的 4 种组合依次输出,不断循环:

表 7-2 列线扫描输出组合

PC3	1	0	0	0
PC2	0	1	0	0
PC1	0	0	1	0
PC0	0	0	0	1

再来看矩阵键盘接法 B(图 7-8)。很容易看出,此接法与接法 A 的原理类似,只是将下拉电阻变成上拉电阻。在没有键被按下时,PC7～PC4 被上拉电阻稳定在高电平。如果某一键被按下,而该键对应的列线为低电平,则对应的 CPLD/FPGA 输入口读到的电平值为 0。与接法 A 同理,任一时刻的 PC3～PC0 的输出只能有一个为 0。

只要将扫描信号变成表 7-3 所列的 4 种组合就行了。

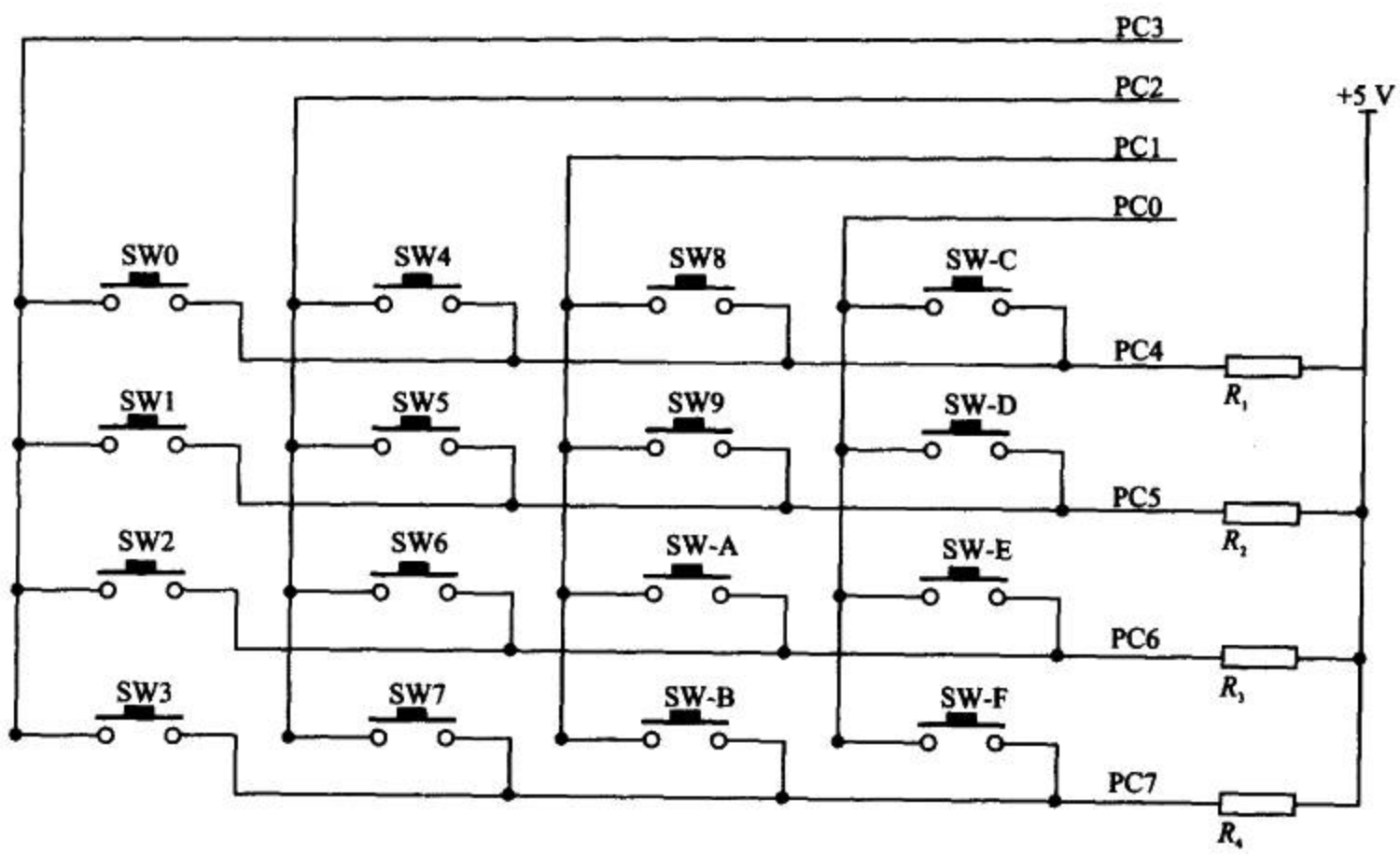


图7-8　矩阵键盘接法B

表7-3　列线扫描信号组合

PC3	0	1	1	1
PC2	1	0	1	1
PC1	1	1	0	1
PC0	1	1	1	0

其行列电平值与按键的对应关系如表7-4所列：

表7-4　行列电平值与按键的对应关系

列/PC3～PC0(输出)	行/PC7～PC4(输入)	按　键
0111	1110	0
0111	1101	1
0111	1011	2
0111	0111	3
1011	1110	4
1011	1101	5

续表 7-4

列/PC3～PC0(输出)	行/PC7～PC4(输入)	按　键
1011	1011	6
1011	0111	7
1101	1110	8
1101	1101	9
1101	1011	A
1101	0111	B
1110	1110	C
1110	1101	D
1110	1011	E
1110	0111	F

例程 7-7 给出了 4×4 矩阵键盘(接法 A)的扫描程序。在这个程序中,只采用了 0～E 共 15 个键,即键值 Num 为 0～14 中的值,把 Num=15 当成无效值,用以指示无按键按下的情况。如果读者所需用键少于 15,或者需要 16 个键,请自行修改程序,以满足实际需要。

例程 7-7　4×4 矩阵键盘的扫描程序

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

ENTITY Keyboard IS
PORT
(Clock:IN Std_Logic;                          --扫描时钟频率不宜过高,一般在 1 kHz 以下
 KIN:IN Std_Logic_Vector(0 TO 3);             --读入行码
 ScanSignal:OUT Std_Logic_Vector(0 TO 3);     --输出列码(扫描信号)
 Num:OUT Integer RANGE 0 TO 15                --输出键值,15 表示无键按下
);
END;

ARCHITECTURE Scan OF Keyboard IS
SIGNAL ScanS:Std_Logic_Vector(0 TO 7);
SIGNAL SCN:Std_Logic_Vector(0 TO 3);
SIGNAL Counter:Integer RANGE 0 TO 3;          --用以计数产生扫描信号
SIGNAL CounterB:Integer RANGE 0 TO 3;         --用以计算
BEGIN
```

```
PROCESS(Clock)
BEGIN
  IF rising_edge(Clock) THEN
    IF Counter=3 THEN
      Counter<=0;
    ELSE
      Counter<=Counter+1;
    END IF;
    CASE Counter IS                          --产生扫描信号
      WHEN 0=>SCN<="1000";
      WHEN 1=>SCN<="0100";
      WHEN 2=>SCN<="0010";
      WHEN 3=>SCN<="0001";
    END CASE;
  END IF;
END PROCESS;

PROCESS(Clock)
BEGIN
  IF falling_edge(Clock) THEN                --上升沿产生扫描信号,下降沿读入行码
    IF KIN="0000" THEN                       --如果连接检测到4次"0000",表示无键按下
      IF CounterB=3 THEN
        Num<=15;                             --15为无效值,用以指示无键按下
        CounterB<=0;
      ELSE
        CounterB<=CounterB+1;
    END IF;
  ELSE
    CounterB<=0;
    CASE ScanS IS
      WHEN "10000001"=>Num<=0;
      WHEN "10000010"=>Num<=1;
      WHEN "10000100"=>Num<=2;
      WHEN "10001000"=>Num<=3;
      WHEN "01000001"=>Num<=4;
      WHEN "01000010"=>Num<=5;
      WHEN "01000100"=>Num<=6;
```

```
            WHEN "01001000"=>Num<=7;
            WHEN "00100001"=>Num<=8;
            WHEN "00100010"=>Num<=9;
            WHEN "00100100"=>Num<=10;
            WHEN "00101000"=>Num<=11;
            WHEN "00010001"=>Num<=12;
            WHEN "00010010"=>Num<=13;
            WHEN "00010100"=>Num<=14;
            WHEN OTHERS=>Num<=Num;
          END CASE;
        END IF;
      END IF;
      END PROCESS;
    ScanS<=SCN&KIN;
    ScanSignal<=SCN;
END;
```

7.4　键盘消抖电路

键盘的按键闭合与释放瞬间，输入的信号会有毛刺。如果不进行消抖处理，系统会将这些毛刺误以为是用户的另一次输入，导致系统的误操作。防抖电路有很多种，最简单、最容易理解的就是计数法。其原理是对键值进行计数，当某一键值保持一段时间不改变时(计数器达到一定值后)，才确认它为有效键值；否则将其判为无效键值，重新对键值进行计数。

例程 7－8 是基于计数法的防抖电路(此为 4×4 矩阵键盘的防抖电路，单个按键的防抖电路原理和此电路是基本相同的，请参考此电路自行设计)。

例程 7－8　基于计数法的防抖电路程序

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

ENTITY Antitwitter IS
PORT
(Clock:IN Std_Logic;
 Numin:IN Integer RANGE 0 TO 15;
 Numout:OUT Integer RANGE 0 TO 15
```

```
);
END;

ARCHITECTURE Behavior OF Antitwitter IS
SIGNAL TempNum:Integer RANGE 0 TO 15;
SIGNAL Counter:Integer RANGE 0 TO 31;
SIGNAL Start:Std_Logic;
BEGIN
  PROCESS(Clock)
  BEGIN
    IF rising_edge(Clock) THEN
      IF Start='0' THEN                    --上电后立即对输出的键值赋予无效值
        TempNum<=15;                       --此处沿用例程7-7的作法,将15作为无效值
        Numout<=15;                        --此无效值务必随实际情况改变
        Start<='1';
      ELSE
        IF Numin/=TempNum THEN             --上一键值与此键值不同
          TempNum<=Numin;                  --记录此键值
          Counter<=0;                      --并对计数器清0,准备对此键值计时
        ELSE
        IF Counter=31 THEN                 --当键值保持31个时钟周期不变时
          NumOut<=Numin;                   --即确认为有效键值,并输出
          Counter<=0;
        ELSE
          Counter<=Counter+1;
        END IF;
      END IF;
    END IF;
  END IF;
 END PROCESS;
END;
```

7.5　同步整形电路

在实际应用中,外部输入的异步信号常常需要进行同步化(与系统时钟同步)和整形(将输入信号整形为一个时钟周期长的信号脉冲),如图7-9所示。

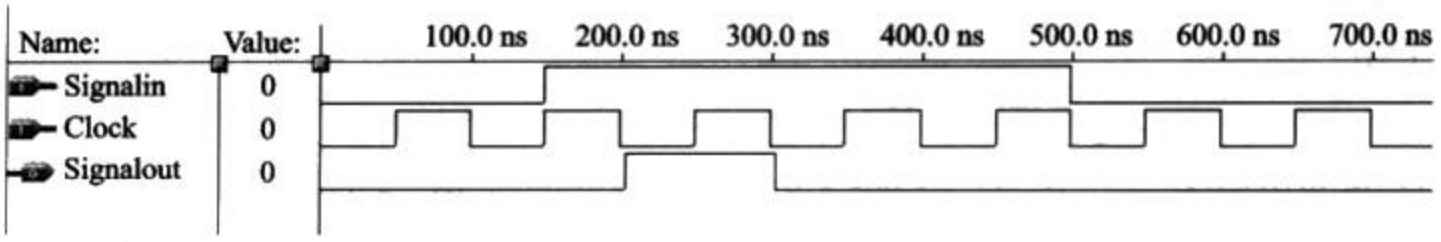


图 7-9　同步整形电路仿真波形

下面给出两个程序，这两个程序的功能一样，都是将外部信号进行同步整形，只是例程 7-9是在系统时钟的上升沿处输出脉冲信号，而例程 7-10 则是在下降沿处输出。之所以给出两个程序是因为这两个程序各具特色。例程 7-9 十分简洁，但要求读者具有较高的数字电路分析能力；例程 7-10 则稍繁琐，但其编程思想很容易被理解，读者很容易读懂这个程序，而且此程序用到的进程间通信方法可供读者借鉴。

此处不对这两个程序加注释，请仔细推敲这两个程序的编程思想和每个语句的作用。同步整形电路在此后的章节中多有应用，望勿轻视之。

同步整形电路 1：

例程 7-9　上升沿输出脉冲信号

```
LIBRARY IEEE;
USE IEEE. Std_Logic_1164. All;

ENTITY SignalLatch IS
PORT
  (Clock: IN Std_Logic;
   SignalIn:IN Std_Logic;
   SignalOut:OUT Std_Logic);
END;

ARCHITECTURE Dataflow OF SignalLatch IS
SIGNAL Q1,Q0:Std_Logic;
BEGIN
  SignalOut<=Q0 AND (NOT(Q1));
  PROCESS(Clock)
  BEGIN
    IF rising_edge(Clock) THEN
      Q0<=SignalIn;
      Q1<=Q0;
```

```
    END IF;
  END PROCESS;
END;
```

同步整形电路2：

例程7-10　下降沿处输出脉冲信号

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

ENTITY Signallatch IS
PORT
(Clock:IN Std_Logic;
 Signalin:IN Std_Logic;
 Signalout:OUT Std_Logic
);
END;

ARCHITECTURE Slatch OF Signallatch IS
SIGNAL Clear,S:Std_Logic;
BEGIN
  PROCESS(Signalin)
  BEGIN
    IF Clear='1' THEN
      S<='0';
    ELSIF rising_edge(Signalin) THEN
      S<='1';
    END IF;
  END PROCESS;

  PROCESS(Clock)
  BEGIN
    IF falling_edge(Clock) THEN
      IF S='1' THEN
        Signalout<='1';
        Clear<='1';
      ELSE
        Signalout<='0';
```

```
        Clear<='0';
      END IF;
    END IF;
  END PROCESS;
END;
```

7.6　三态缓冲器

逻辑输出有两个正常态:低电平状态(对应逻辑 0)和高电平状态(对应逻辑 1)。另外,电路还有不属于 0 和 1 的第 3 种状态,称为高阻态(Hi - Z)。所谓高阻,即输出端处于浮空状态(只有很小的漏电流流动),其电平随外部电平高低而定,即门电路放弃对输出端电路的控制。或者可以简单地理解为:输出与电路是断开的。具有 3 种可能状态的输出称为三态输出,具有三态输出的器件称为三态器件。

最基本的三态器件是三态门,也称为三态缓冲器。图 7 - 10 为最基本的三态缓冲器的逻辑符号。当 OE 为高电平时,Dataout 与 Datain 相连;而 OE 为低电平时,Dataout 为高阻态,相当于和 Datain 之间的连线断开。

三态缓冲器有许多实际应用,例如 CPU 设计中的数据总线和地址总线的构建,以及外部 SRAM 电路的读/写(见 7.7 节)等。

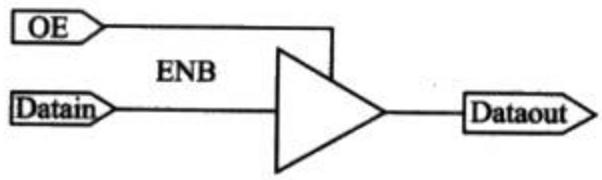


图 7 - 10　三态缓冲器的逻辑符号图

从理论上说,用 VHDL 描述三态缓冲器的实现方法有很多种,但实际上各种综合器对三态缓冲器的描述都有比较严格的要求,例程 7 - 11 和例程 7 - 12 为 MAX+plus II 默认的两种三态门的描述方式。

例程 7 - 11　MAX+plus II 默认的三态门的描述方式 1

```
LIBRARY IEEE;
USE IEEE.Std_Logic_1164.All;

ENTITY TriBuffer IS
PORT
( OE:IN Std_Logic;
  Datain:IN Std_Logic;
  Dataout:OUT Std_Logic
);
```

```
END;

ARCHITECTURE TriState Of TriBuffer IS
BEGIN
  PROCESS (OE,Datain)
  BEGIN
      IF OE='0' THEN
      Dataout<='Z';
      ELSE
          Dataout<=Datain;
    END IF;
  END PROCESS;
END TriState;
```

例程 7-12　MAX+plus II 默认的三态门的描述方式 2

```
LIBRARY IEEE;
USE IEEE.Std_Logic_1164.All;

ENTITY TriBuffer IS
PORT
( OE:IN Std_Logic;
  Datain:IN Std_Logic;
  Dataout:OUT Std_Logic
);
END;

ARCHITECTURE TriState OF TriBuffer IS
BEGIN
  Dataout<=Datain WHEN OE='1' ELSE
            'Z';
END TriState;
```

7.7　SRAM 控制电路

SRAM(Static Random Access Memory,静态随机存储器)是数字系统中最常用的存储器,它具有速度快、接口简单、读/写操作简便等优点。下面以 UT62256 为例,介绍 SRAM 的

读/写时序及用 CPLD/FPGA 控制 SRAM 进行读/写的方法。

UT62256 是一款 32K×8 Bit(256 Kbits)的 SRAM，图 7－11 所示为 UT62256 的芯片引脚图(DIP 封装)，图 7－12 则是其内部的功能框图。从图中可以看出，它有 A14～A0 共 15 位地址线，用于选中片内的某个存储单元；I/O1～I/O8 共 8 位双向数据线；另外还有 3 位控制线，分别为 CS#(片选信号，低电平表示该芯片被选中)、WE#(写使能信号，低电平时数据被写入 SRAM)和 OE#(输出使能信号，通常用于读操作控制，低电平时可从 SRAM 读出数据)。

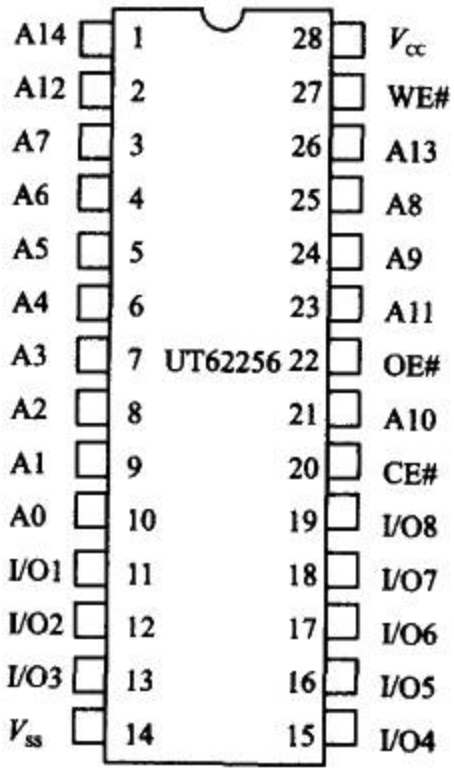


图 7－11　62256 引脚图

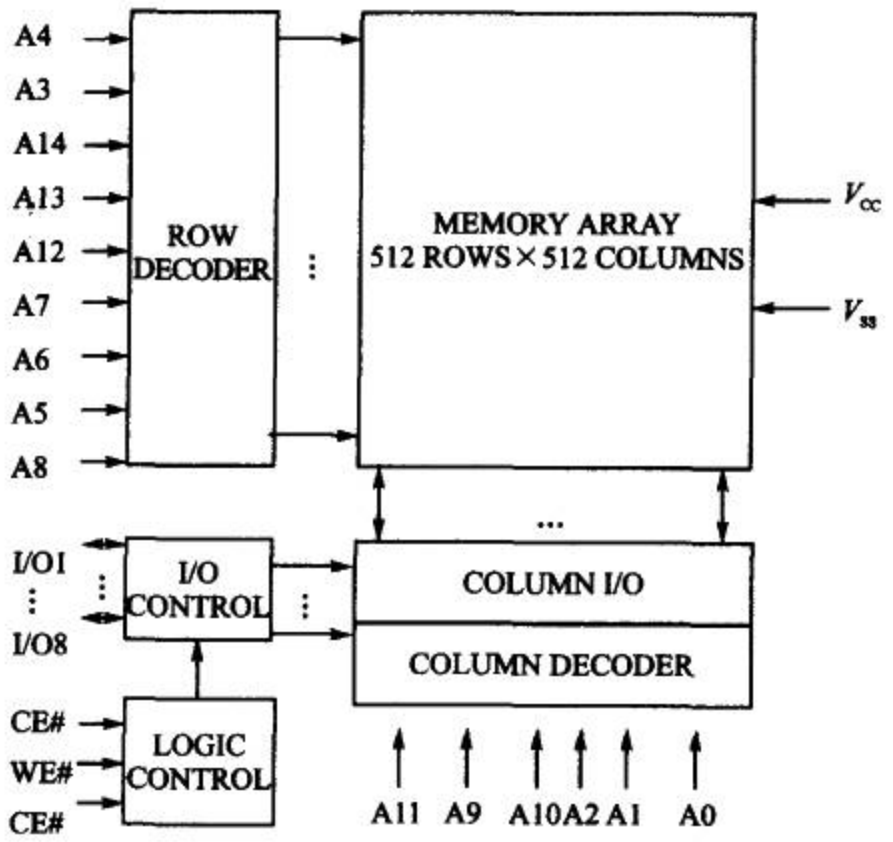


图 7－12　SRAM 内部功能结构图

7.7.1　SRAM UT62256 的读/写时序说明

从 UT62256(及大部分 SRAM)中读取数据有两种方法，分别介绍如下：

(1) 读时序 1(通过地址线控制)

在该模式下，将写使能(WE#)始终置为高电平，片选信号(CE#)和输出使能(OE#)置为低电平。当 SRAM 的地址改变后，输出的数据也随之改变，但刚开始输出数据不稳定，需要经过地址载入时间(t_{AA})后，SRAM 才开始输出有效数据，此时可以读取数据。这种方法十分方便，无需控制信号线的介入，如图 7－13 所示。

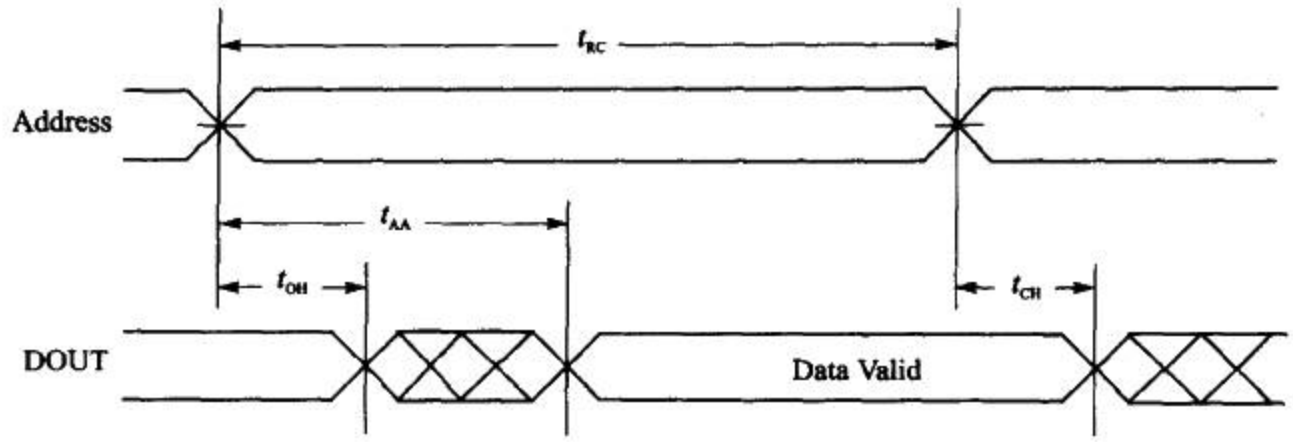


图 7-13　SRAM 读时序图示 1

(2) 读时序 2(通过片选信号 CE# 和输出使能 OE# 控制)

读取数据的第 2 种方法如图 7-14 所示。读者如果不习惯看时序图，十有八九会被这个时序图弄得不知所措。其实，图 7-14 总结起来就是：

确定地址 => 拉低 CE# => 拉低 OE# => 读取数据

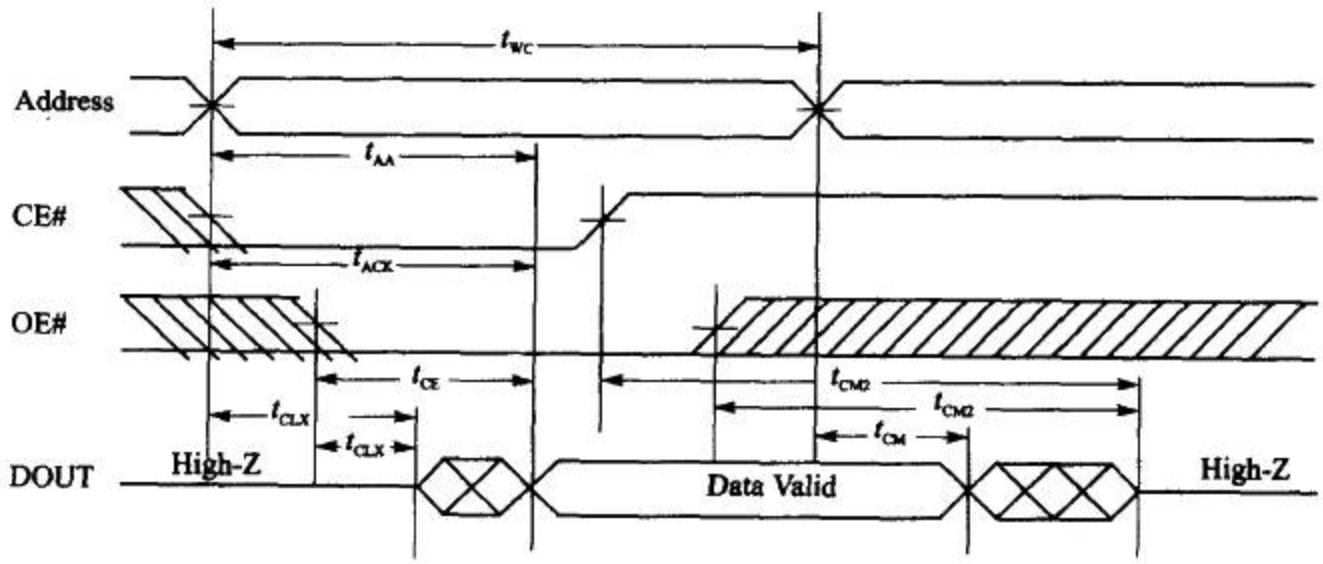


图 7-14　SRAM 读时序图示 2

在各步操作之间都要间隔一定的时间，至于间隔时间要多长，不同 SRAM 的要求是不同的，请参考 Datasheet。其实，只要保守一点，将等待时间适当延长，就可让一个程序几乎适用于所有 SRAM。

往 SRAM 中写数据也有两种方法，分别介绍如下：

(1) 写时序 1(通过写使能 WE# 控制)

如图 7-15 所示，该模式下的时序图可以总结为：

确定地址=>拉低 CE#=>拉低 WE#=>将要写入的 8 位数据置于数据线上=>

拉高WE#

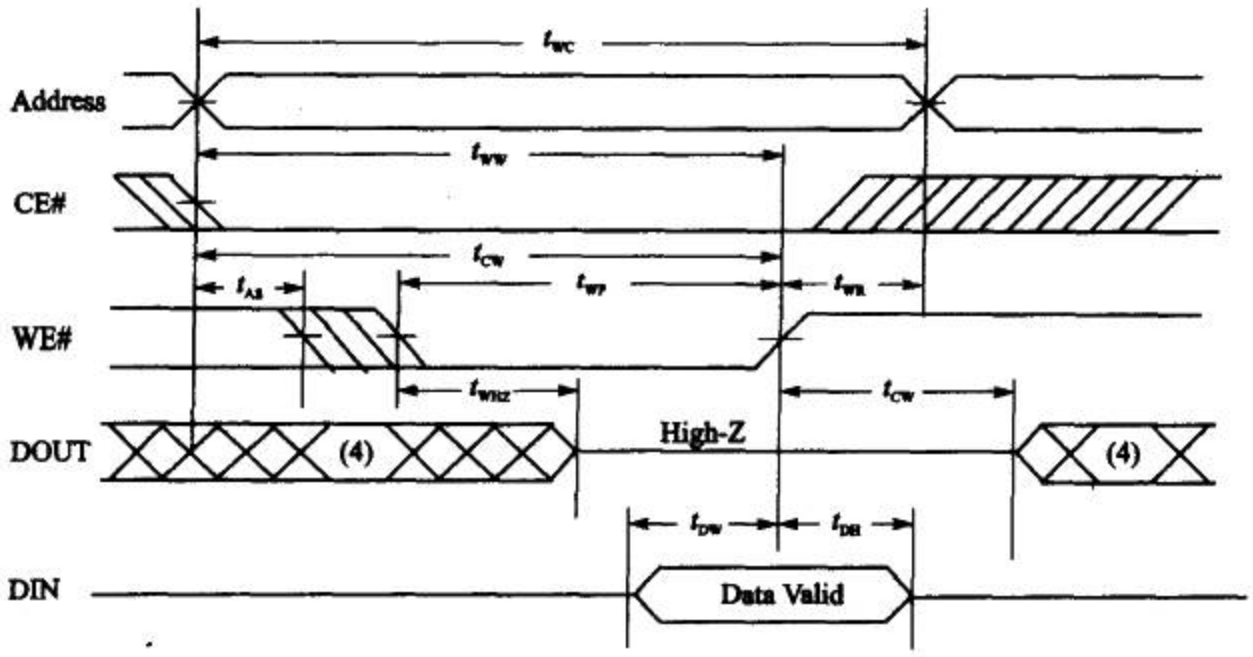


图 7-15 SRAM 写时序图 1

在写模式下,输出使能信号 OE# 的值对写操作不产生影响。

(2) 写时序 2(通过片选信号 CE# 控制)

如图 7-16 所示,该模式下的时序图可以总结为:

确定地址=>拉低 WE#=>拉低 CE#=>将要写入的 8 位数据置于数据线上=>拉高CE#

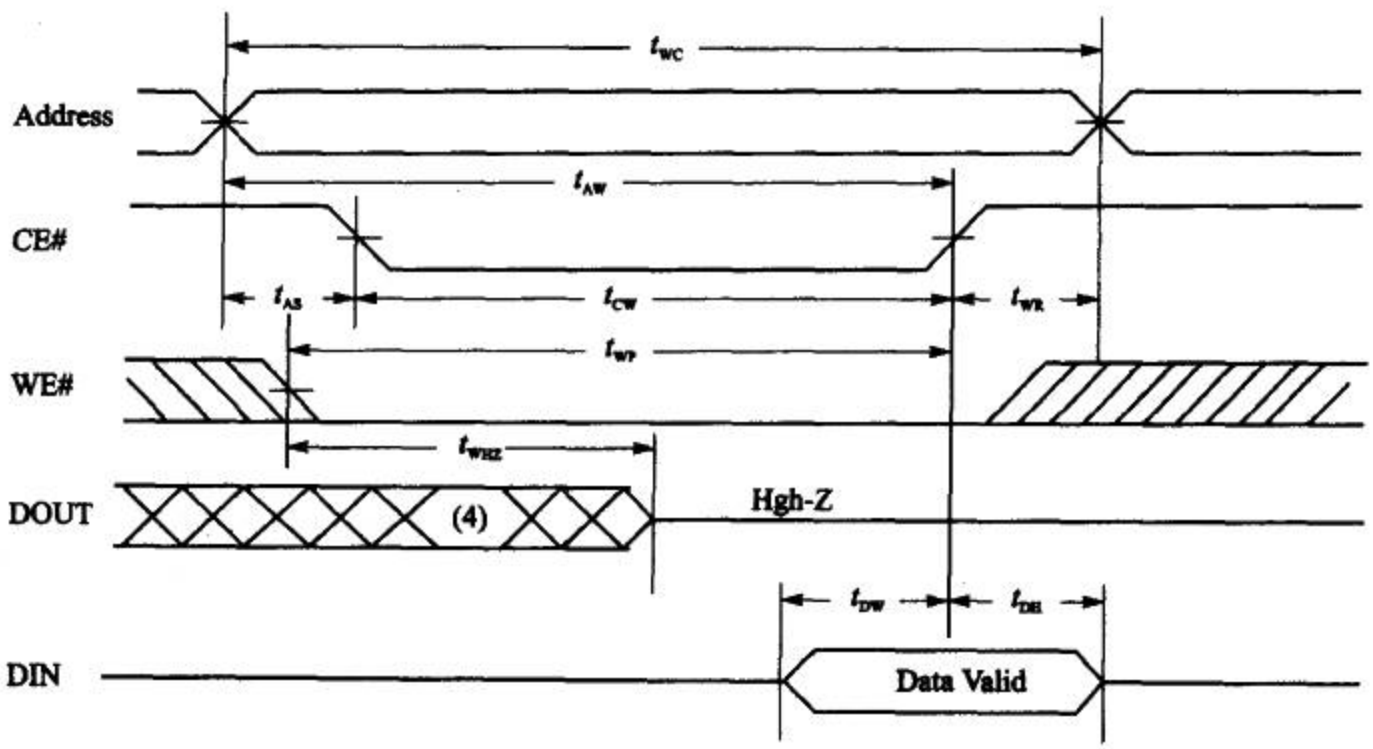


图 7-16 SRAM 写时序图 2

7.7.2　用CPLD/FPGA控制SRAM的读/写

此处采用原理图与VHDL语言结合的方法对SRAM进行读/写测试(对SRAM的前2 KB写入数据,写满后再读出,并输出到特定的引脚)。

图7-17中有一点需要注意,接到SRAM的8位数据线上的MemData[7..0]必须设成BIDIR(双向输入/输出口),而SRAM控制器上的MemDataIn[7..0](从SRAM读入的数据)和MemDataOut[7..0](写入SRAM的数据)不能直接接在一起,而应该通过三态总线进行隔离。当SRAM控制器从SRAM中读取数据时,OE为低电平,则MemDataOut[7..0]相当于与MemData[7..0]断开,这样就避免了数据冲突。当SRAM控制器往SRAM中写入数据时,OE为高电平,MemDataOut[7..0]相当于与MemData[7..0]直接相连(此时MemDataOut[7..0]与MemDataIn[7..0]也相连,但并不会产生数据冲突)。

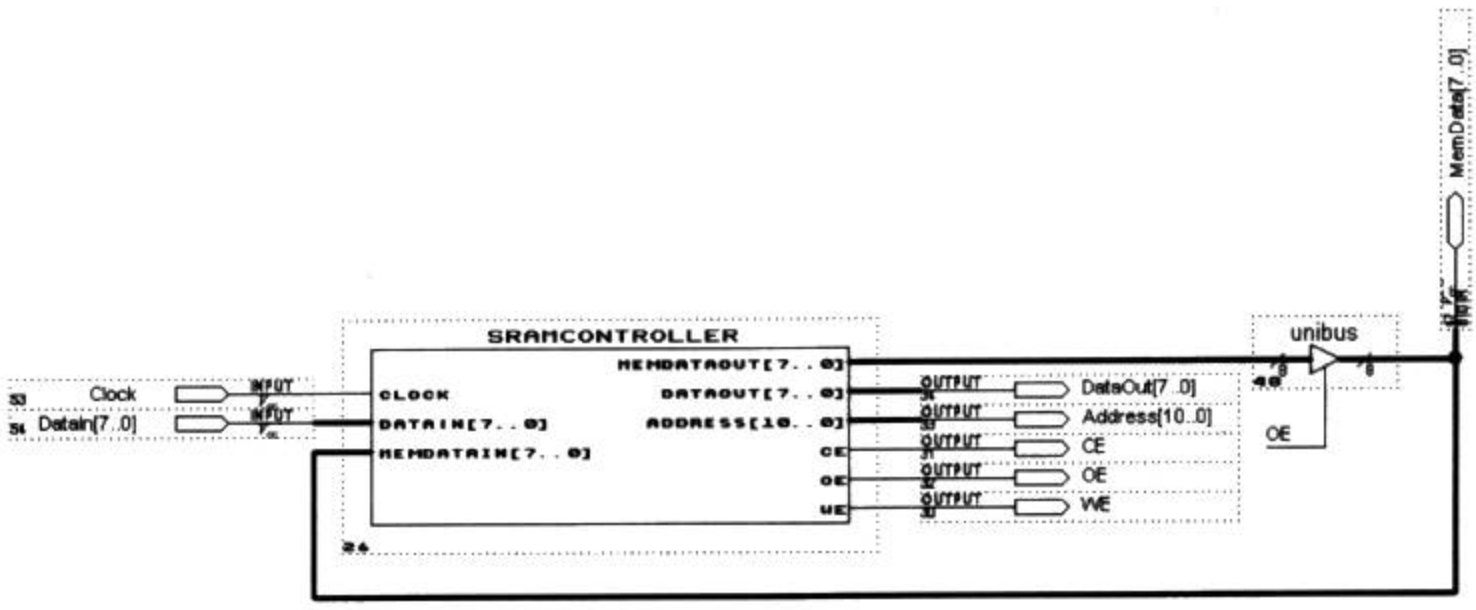


图7-17　SRAM读/写电路

例程7-13仅用于验证SRAM与CPLD/FPGA是否连接正确,若在设计系统时需要用到SRAM,应该根据自己的要求,按照前面所说的时序关系进行编程,不能照搬例程。

例程7-13　验证SRAM与CPLD/FPGA是否连接正确

```
LIBRARY IEEE;
USE IEEE. Std_Logic_1164. ALL;
USE IEEE. Std_Logic_Unsigned. ALL;

ENTITY SramController IS
```

```
PORT
(Clock:IN Std_Logic;
 DataIn:IN Std_Logic_Vector(7 DOWNTO 0);          --准备写入 SRAM 的数据
 MemDatain:IN Std_Logic_Vector(7 DOWNTO 0);       --从 SRAM 读取的数据
 MemDataout:OUT Std_Logic_Vector(7 DOWNTO 0);     --写入 SRAM 的数据
 DataOut:OUT Std_Logic_Vector(7 DOWNTO 0);
                                                  --输出从 SRAM 读取的数据到指定引脚
                                                  --以验证程序是否正确
 Address:BUFFER Integer RANGE 0 TO 2047;          --11 位地址线
 CE:OUT Std_Logic;                                --连接到 CE#
 OE:OUT Std_Logic;                                --连接到 OE#
 WE:OUT Std_Logic                                 --连接到 WE#
);
END;

ARCHITECTURE Controller OF SramController IS
SIGNAL Counter:Integer RANGE 0 TO 9;
SIGNAL Read:Std_Logic;
BEGIN
  PROCESS(Clock)
  BEGIN
    IF rising_edge(Clock) THEN
      IF Counter=9 THEN
        Counter<=0;
      ELSE
        Counter<=Counter+1;
      END IF;
    END IF;
  END PROCESS;

  PROCESS(Clock)
  BEGIN
    IF rising_edge(Clock) THEN
      IF Counter=0 THEN                           --也可以改用 CASE 语句
        CE<='1';
        IF Read='0' THEN
          MemDataout<=DataIn;
```

```
        END IF;
      ELSIF Counter=1 THEN
        CE<='0';
      ELSIF Counter=2 THEN
        IF Read='1' THEN
          Dataout<=MemDatain;
        END IF;
      ELSIF Counter=3 THEN
        CE<='1';
      ELSIF Counter=4 THEN
        IF Address=2047 THEN    --如果写满 2 KB,就转为读模式
          Address<=0;
          Read<='1';
        ELSE
          Address<=Address+1;--如果未满 2 KB,就继续写下一单元
        END IF;
      END IF;
    END IF;
  END PROCESS;
  OE<=Not Read;                    --Read='0'(写模式下,OE# 为高电平,表示 SRAM 不输出数据)
    WE<=Read;                      --Read='0'(写模式下,WE# 为低电平;否则为高电平)
  END;
```

7.8　Flex10K10 内部 RAM 的读/写

Flex10K10 中共有 3 块 EAB,每块大小为 2 Kbits,可构成 2 048×1、1 024×2、512×4、256×8 等 4 种类型的 RAM/ROM 中任意一种。较常使用的是 256×8 这种类型的 ROM/RAM。Flex10K 中的 RAM 有几种配置方法,如图 7-18 所示。在 mega_lpm 库里,有几个关于 RAM 的 Symbol Files:lpm_ram_dp、lpm_ram_dq、lpm_ram_io 等。本节以 lpm_ram_dq 为例,简要介绍内部 RAM 的读/写方法。这个 RAM 是一个输入数据口和输出数据口分开的 RAM,这样可以不用三态门进行总线隔离,减少编程上的麻烦。

LPM_RAM_DQ 的逻辑是很简单的,当 WE 为高电平时,将输入的数据(data[])写入当前地址的 RAM 中;当 WE 为低电平时,读出当前地址的数据(q[])。内部 RAM 读/写控制器与 LPM_RAM_DQ 的接法如图 7-19 所示。

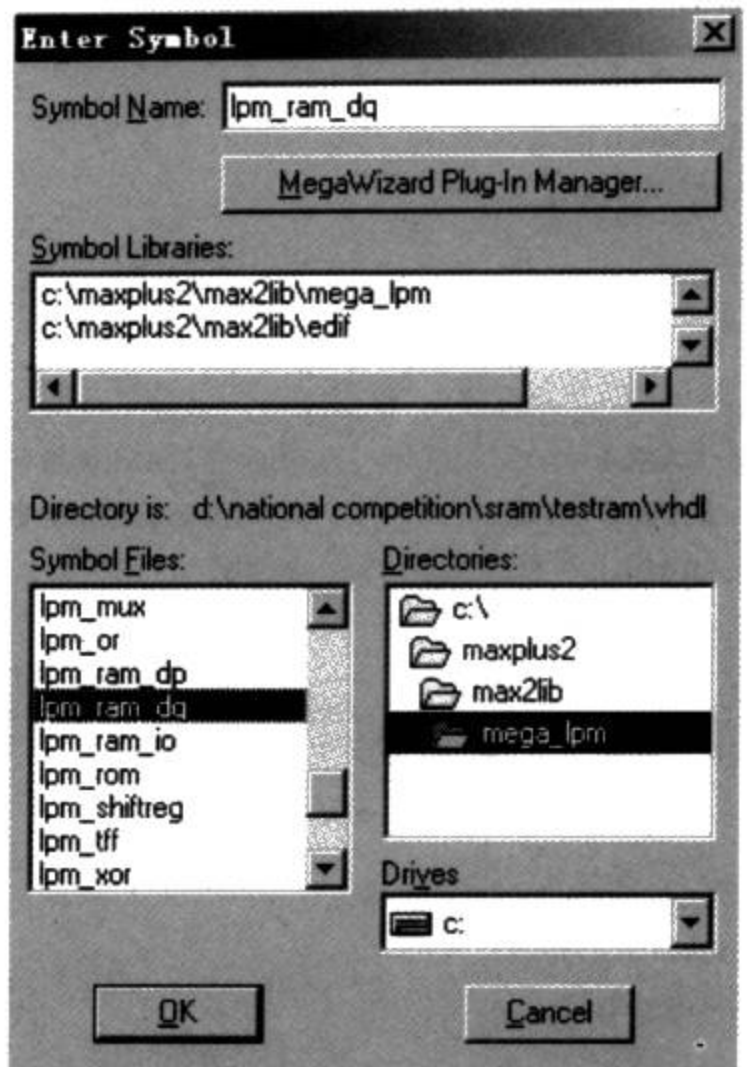


图 7－18　LPM 库中的 RAM

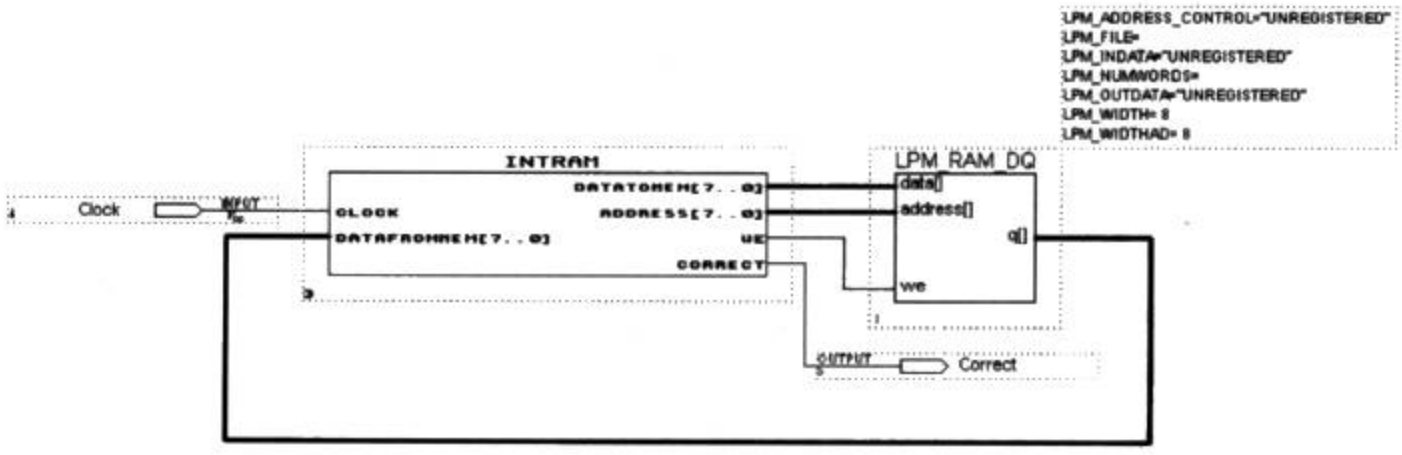


图 7－19　内部 RAM 读/写电路图

例程 7－14 是一个简单的 LPM_RAM_DQ 读/写电路的 VHDL 程序，它仅用于验证 LPM_RAM_DQ的读/写是否正常，读者需要根据具体情况修改程序。

例程 7－14　LPM_RAM_DQ 读/写电路程序

```
LIBRARY IEEE;
USE IEEE. Std_Logic_1164. ALL;
USE IEEE. Std_Logic_Unsigned. ALL;

ENTITY INTRAM IS
PORT
(Clock:IN Std_Logic;
 DatafromMem:IN Integer RANGE 0 TO 255;      --从内部 RAM 读取的数据
 DatatoMem:OUT Integer RANGE 0 TO 255;       --写入内部 RAM 的数据
 Address:BUFFER Integer RANGE 0 TO 255;      --地址线
 WE:OUT Std_Logic;                           --写使能
 Correct:OUT Std_Logic                       --在仿真中检测读/写是否正确的信号
);
END;

ARCHITECTURE RAMController OF INTRAM IS
SIGNAL Counter:Integer RANGE 0 TO 9;
SIGNAL Data:Integer RANGE 0 TO 255;
BEGIN
  PROCESS(Clock)
  BEGIN
    IF rising_edge(Clock) THEN
      IF Counter=9 THEN
        Counter<=0;
      ELSE
        Counter<=Counter+1;
      END IF;
    END IF;
  END PROCESS;
  PROCESS(Clock)
  BEGIN
    IF rising_edge(Clock) THEN
      IF Counter=0 THEN
        IF Address=255 THEN
          Address<=0;
        ELSE
```

```
            Address<=Address+1;
          END IF;
          ELSIF Counter=1 THEN
            DatatoMem<=Address;                    --写入与地址相同的数据
            WE<='1';                               --WE为高电平时写入数据
          ELSIF Counter=2 THEN
            WE<='0';                               --WE为低电平时读取数据
          ELSIF Counter=3 THEN
            IF DatafromMem=Address THEN            --如果读取的数据与地址相同
              Correct<='1';                        --Correct信号线输出高电平
            ELSE
              Correct<='0';
            END IF;
          END IF;
        END IF;
      END PROCESS;
END;
```

第8章

交通灯控制器

8.1 任务书

(1) 简要说明

在十字路口，每条道路各有一组红、黄、绿灯和倒计时显示器，用以指挥车辆和行人有序地通行。其中，红灯(R)亮，表示该条道路禁止通行；黄灯(Y)亮，表示停车；绿灯(G)亮，表示可以通行。倒计时显示器是用来显示允许通行或禁止通行的时间。交通灯控制器就是用来自动控制十字路口的交通灯和计时器，指挥各种车辆和行人安全通行。

(2) 任务和要求

① 在十字路口的两个方向上各设一组红、绿、黄灯，显示顺序为其中一方向(东西方向)是绿灯、黄灯、红灯；另一方向(南北方向)是红灯、绿灯、黄灯。

② 设置一组数码管，以倒计时的方式显示允许通行或禁止通行的时间，其中绿灯、黄灯、红灯的持续时间分别是 20 s、5 s 和 25 s。

③ 当各条路上任意一条上出现特殊情况时，如当消防车、救护车或其他需要优先放行的车辆通过时，各方向上均是红灯亮，倒计时停止，且显示数字在闪烁。当特殊运行状态结束后，控制器恢复原来状态，继续正常运行。

④ 选做：用两组数码管实现双向倒计时显示。

(3) 训练目标

具备用 VHDL 设计数字系统的初步能力，熟悉开发环境和流程，掌握计数器用法。

8.2　参考设计

8.2.1　系统框图

本题逻辑简单，用到的外围器件也不多，作为初学者入门设计的第 1 个系统非常合适。从题目中很容易看出，本题的重点在计时上，因此计数器是必不可少的。对于这个系统，即可以只用计数器实现，也可以用计数器配合状态机来实现。前者实现起来比较简单，占用资源较少，但程序的可读性可能不如状态机好。本设计采用计数器完成，未使用状态机。

计数器的计数值与交通灯亮灭的关系如图 8－1 所示。

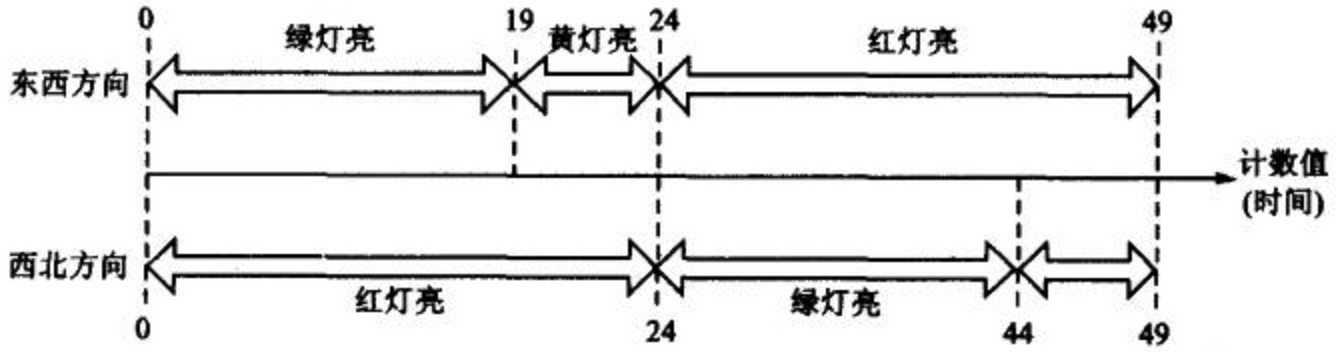


图 8－1　计数值与交通灯亮灭的关系

显然，本课题的核心是一个计数范围为 0～49(共 50 s)的计数器和一个根据计数值做出规定反应的控制器。另外，作者所用实验仪配备的晶振为 20 MHz，因此还需要一个分频电路。最后，要驱动七段数码管，显然还需要一个译码电路。

根据上面的分析，可以画出如图 8－2 所示的系统框图。

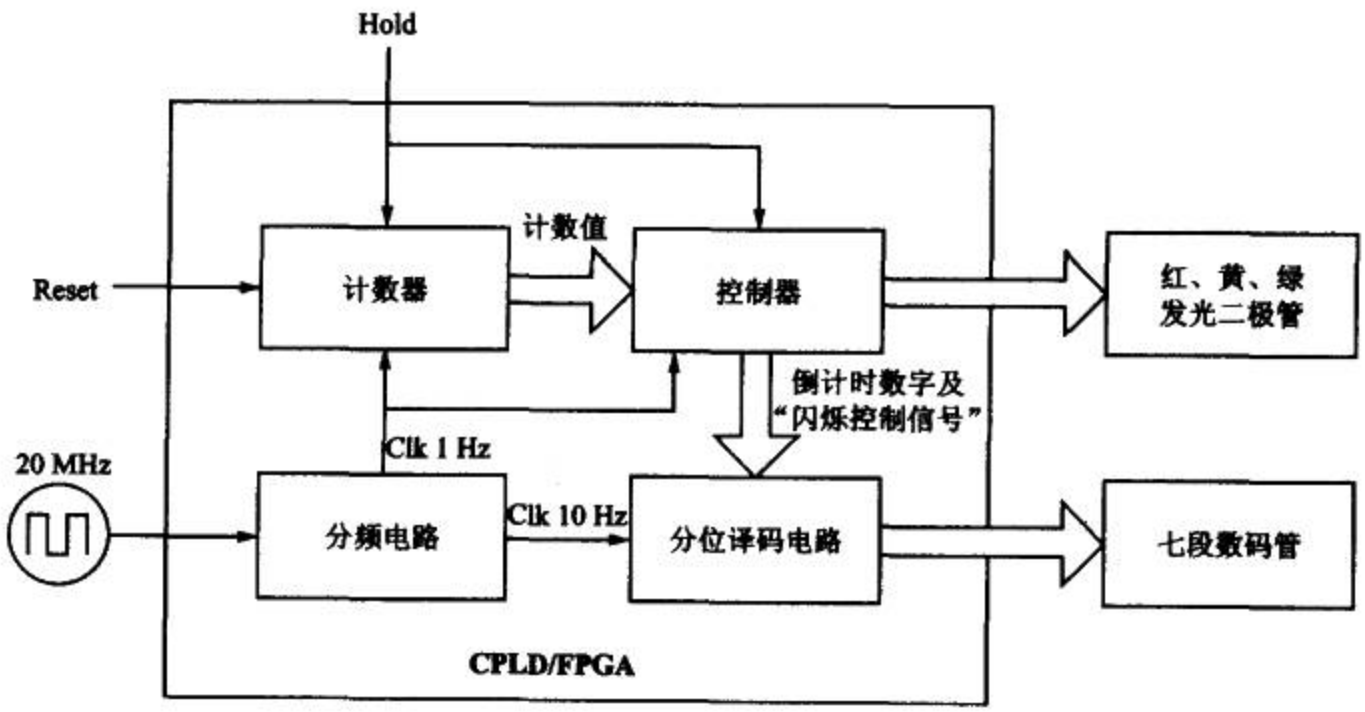


图 8－2　交通灯控制器系统框图

8.2.2　计数器的设计

这里需要的计数器的计数范围为0～49。计到49后，下一个时钟沿回复到0，开始下一轮计数。此外，当检测到特殊情况(Hold='1')发生时，计数器暂停计数，而系统复位信号Reset则使计数器异步清0。例程8-1为计数器的参考设计。

例程8-1　计数器程序

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

ENTITY Counter IS
PORT
(Clock:IN Std_Logic;
 Reset:IN Std_Logic;
 Hold:IN Std_Logic;
 CountNum:BUFFER Integer RANGE 0 TO 49
);
END;

ARCHITECTURE Behavior OF Counter IS
BEGIN
  PROCESS(Reset,Clock)
  BEGIN
  IF Reset='1' THEN
    CountNum<=0;
  ELSIF rising_edge(Clock) THEN
    IF Hold='1' THEN                    --当出现紧急情况时,计数器暂停计数
      CountNum<=CountNum;
    ELSE
      IF CountNum=49 THEN
        CountNum<=0;
      ELSE
        CountNum<=CountNum+1;
      END IF;
    END IF;
  END IF;
```

```
END PROCESS;
END;
```

8.2.3　控制器的设计

控制器的作用是根据计数器的计数值控制发光二极管的亮、灭，以及输出倒计时数值给七段数译管的分位译码电路。此外，当检测到特殊情况(Hold='1')发生时，无条件点亮红色的发光二极管。

由于控制器要对计数值进行判断，很容易想到用 IF 语句来实现。本控制器可以有两种设计方法，一种是利用时钟沿的下降沿读取前级计数器的计数值，然后作出反应；另一种则是将本模块设计成纯组合逻辑电路，不需要时钟驱动。这两种方法各有所长，必须根据所用器件的特性进行选择：比如有些 FPGA 有丰富的寄存器资源，而且可用于组合逻辑的资源则相对较少，那么使用第 1 种方法会比较节省资源；而有些 CPLD 的组合逻辑资源则相对较多，用第 2 种方法可能会更好。这个话题的讨论已超出了“入门”的范畴，因此这里不准备继续深入下去。读者可尝试两种方法，比较一下哪种方法所用资源较少，然后在最后的方案中采用这个方法。例程 8-2 为控制器的参考设计(采用上述第 1 种方法)。

例程 8-2　控制器程序

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

ENTITY Controller IS
PORT
(Clock:IN Std_Logic;
 Hold:IN Std_Logic;
 CountNum:IN Integer RANGE 0 TO 49;          --前级计数器的计数值
 NumA,NumB:OUT Integer RANGE 0 TO 25;        --倒计时数值输出
 RedA,GreenA,YellowA:OUT Std_Logic;          --控制东西方向红、黄、绿灯的亮、灭
 RedB,GreenB,YellowB:OUT Std_Logic;          --控制南北方向红、黄、绿灯的亮、灭
 Flash:OUT Std_Logic                         --用以指示七段数码管显示数字的闪烁
);
END;

ARCHITECTURE Behavior OF Controller IS
BEGIN
  PROCESS(Clock)
```

```
BEGIN
  IF falling_edge(Clock) THEN                    --计数器是上升沿改变计数值,此处用下降沿读取
    IF Hold='1' THEN
      RedA<='1';
      RedB<='1';
      GreenA<='0';
      GreenB<='0';
      YellowA<='0';
      YellowB<='0';
      Flash<='1';
    ELSE
      Flash<='0';
      IF CountNum<=19 THEN
        NumA<=20-CountNum;                       --计算东西方向倒计时数值
        RedA<='0';
        GreenA<='1';
        YellowA<='0';
      ELSIF (CountNum<=24) THEN
        NumA<=25-CountNum;
        RedA<='0';
        GreenA<='0';
        YellowA<='1';
      ELSE
        NumA<=50-CountNum;
        RedA<='1';
        GreenA<='0';
        YellowA<='0';
      END IF;

      IF CountNum<=24 THEN
        NumB<=25-CountNum;                       --计算南北方向倒计时数值
        RedB<='1';
        GreenB<='0';
        YellowB<='0';
      ELSIF CountNum<=44 THEN
        NumB<=45-CountNum;
        RedB<='0';
```

```
            GreenB<='1';
            YellowB<='0';
          ELSE
            NumB<=50-CountNum;
            RedB<='0';
            GreenB<='0';
            YellowB<='1';
          END IF;
        END IF;
      END IF;
    END PROCESS;
END;
```

8.2.4　分位译码电路的设计

因为控制器输出的倒计时数值可能是 1 位或者 2 位十进制数，所以在七段数码管的译码电路前要加上分位电路(即将其分为 2 个 1 位的十进制数，如 25 分为 2 和 5，7 分为 0 和 7)。

与控制器一样，分位电路同样可以由时钟驱动，也可以设计成纯组合逻辑电路。控制器中，引入了寄存器。为了让读者开拓眼界，分位电路就用组合逻辑电路实现。例程 8－3 是分位电路的参考程序。

例程 8－3　分位电路程序

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

ENTITY Fenwei IS
PORT
(Numin:IN Integer RANGE 0 TO 25;
 NumA,NumB:OUT Integer RANGE 0 TO 9
);
END;

ARCHITECTURE Behavior OF Fenwei IS
BEGIN
  PROCESS(Numin)
  BEGIN
```

```
    IF Numin>=20 THEN
      NumA<=2;
      NumB<=Numin-20;
    ELSIF Numin>=10 THEN
      NumA<=1;
      NumB<=Numin-10;
    ELSE
      NumA<=0;
      NumB<=Numin;
    END IF;
  END PROCESS;
END;
```

七段数码管的译码电路在前面的章节中已经给出，请参考前面的程序自己设计。因为要实现闪烁显示数字的功能，所以此电路必须引入时钟(见图 8-2 和图 8-3)。

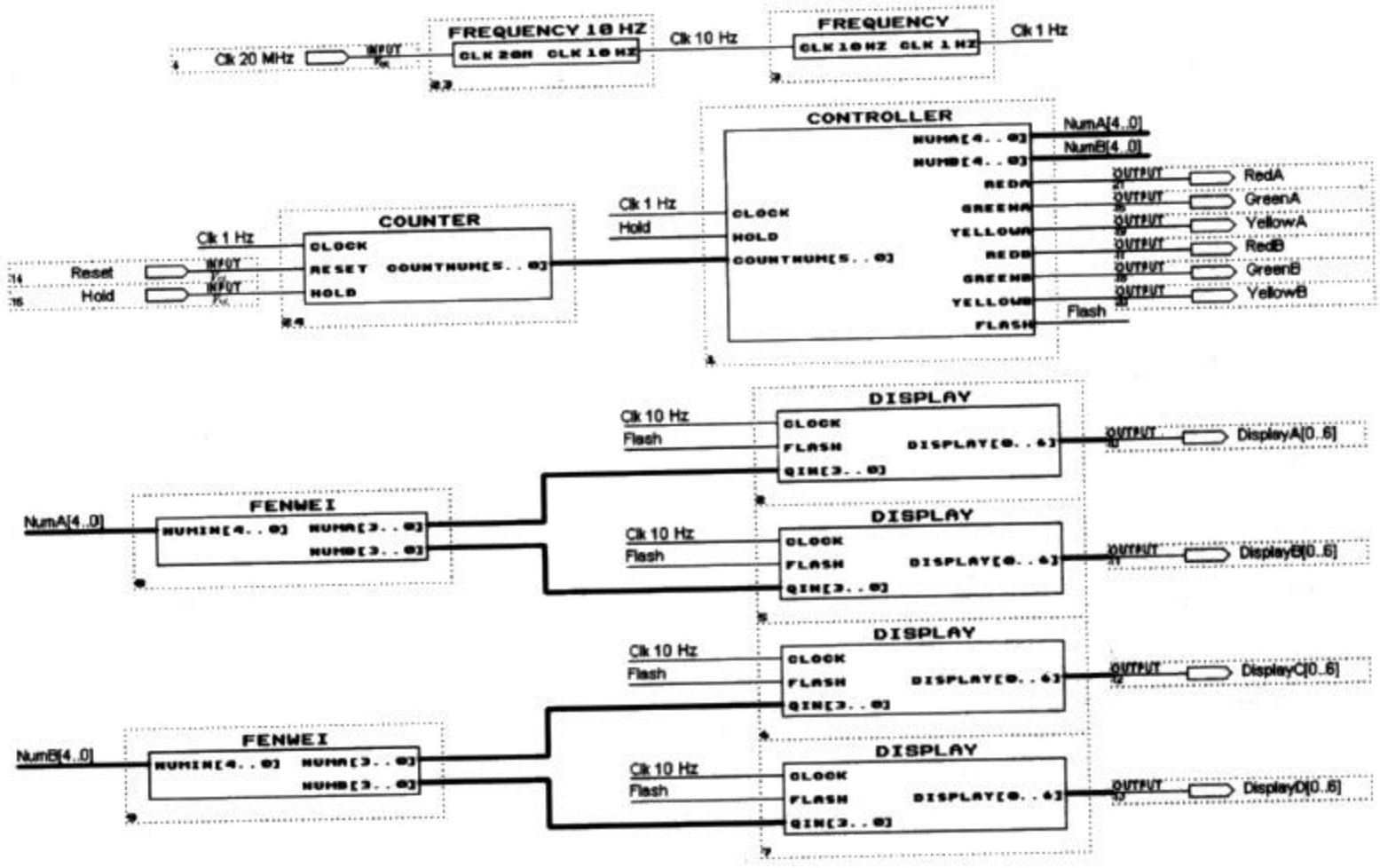


图 8-3　交通灯控制器顶层文件原理图

8.2.5　顶层文件元件连接图

图8-3为交通灯控制器的顶层文件连接图。由于本课题比较简单，此处为节省篇幅就不给出仿真波形了。

8.3　练习题

8.3.1　用状态机实现交通灯控制器

用状态机实现本课题，可以用两个状态机分别控制两个方向上的交通灯和倒数计时器的显示，可以将红灯亮、绿灯亮和黄灯亮分别对应一个状态，通过计时来进行状态转移。但要注意两个状态机之间并不是完全独立的，关于如何作好同步的工作，就留给读者思考了。

8.3.2　计时秒表

在学完本章之后，读者应当对计数器的应用有所了解了，这里给出另一个计数器的经典应用——计时秒表。

设计一个计时秒表，具有以下功能：

① 有启/停开关，用于开始/结束计时操作；

② 秒表计时长度为59分59.99秒，超过计时长度，有溢出则报警，计时长度可手动设置；

③ 设置复位开关，在任何情况下，只要按下复位开关，秒表都要无条件地进行复位清0操作。

第9章

乒乓游戏机

9.1 任务书

图 9-1 是乒乓游戏机的组成示意图。

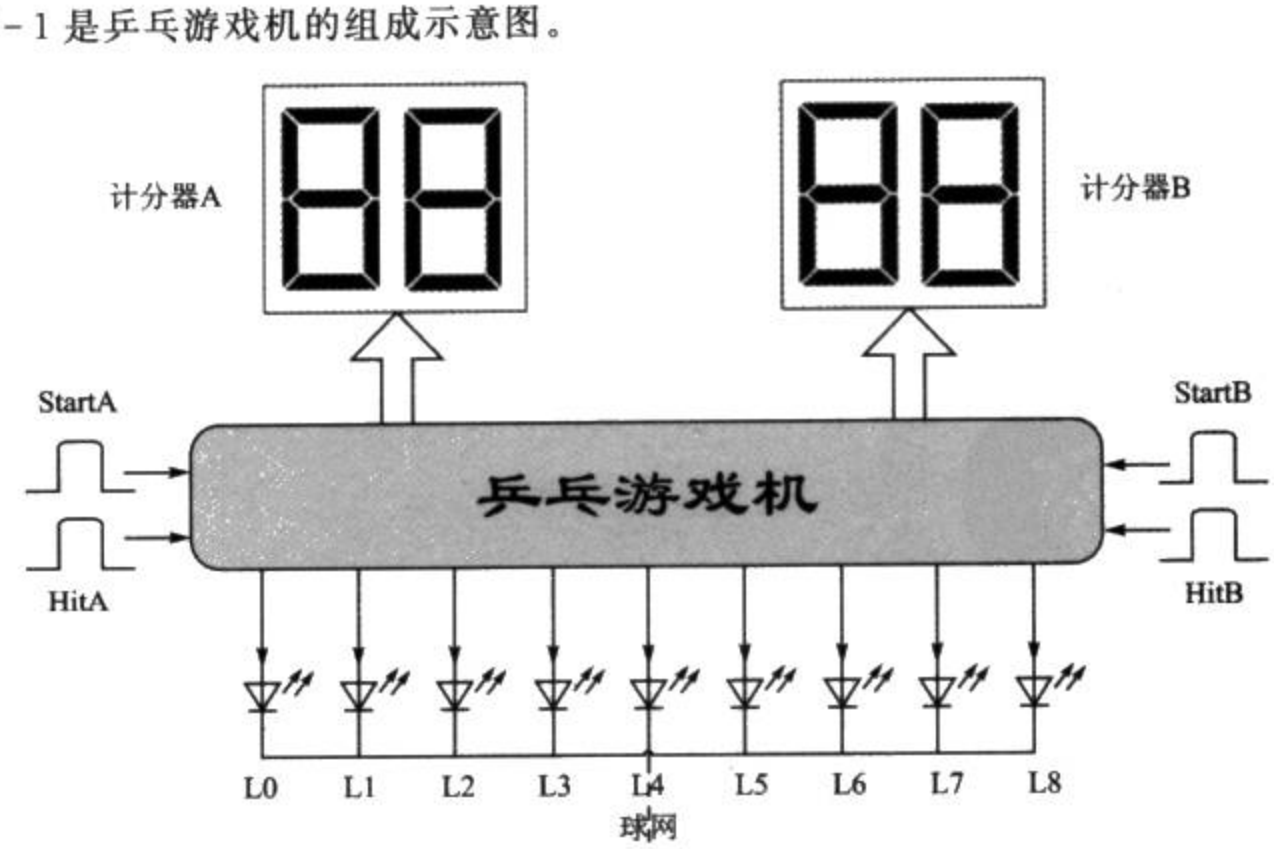


图 9-1 乒乓游戏机的组成

(1) 简要说明

两人乒乓游戏机是用9个发光二极管代表乒乓球台,中间的发光二极管兼作球网,用点亮的发光二级管按一定的方向移动来表示球的运动。在游戏机的两侧各设置两个开关,一个是发球开关StartA、StartB;另一个是击球开关HitA、HitB。甲、乙二人按乒乓球比赛规则来操作开关。当甲方按动发球开关StartA时,靠近甲方的第1个发光二级管亮,然后发光二极管由甲向乙依次点亮,代表乒乓球的移动。当球过网后按设计者规定的球位,乙方就可击球。若乙方提前击球或没击着球,则判乙方失分,甲方的记分牌自动加分。然后重新发球,比赛继续进行。比赛一直要进行到一方记分牌达到21分,该局结束。

(2) 任务和要求

设计一个乒乓游戏机,该机模拟乒乓球比赛的基本过程和规则,并能自动裁判和记分。要求如下:

① 使用乒乓游戏机的甲、乙双方各在不同的位置发球或击球。

② 乒乓球的位置和移动方向由灯亮及依次点亮的方向决定。球移动的速度为0.1~0.5 s移动1位(读者可据实际情况自定)。球过网,接球方即可击球,提前击球或没击着球均判失分。

③ 比赛按21分为一局进行,甲、乙双方都应设置各自的记分牌,任何一方先记满21分,该方就算胜出。按Reset键将记分牌清0后,可开始新的一局比赛。

(3) 训练目标

学会用VHDL设计简单的状态机,掌握用状态机控制其他模块并处理其反馈信息的经典用法。

9.2 参考设计

9.2.1 系统框图

此系统的逻辑划分框图如图9-2所示,其中译码显示器与按键去抖等模块在前面的章节中已经给出例程,此处不再重复,仅给出状态机/球台控制器与记分器的参考设计。

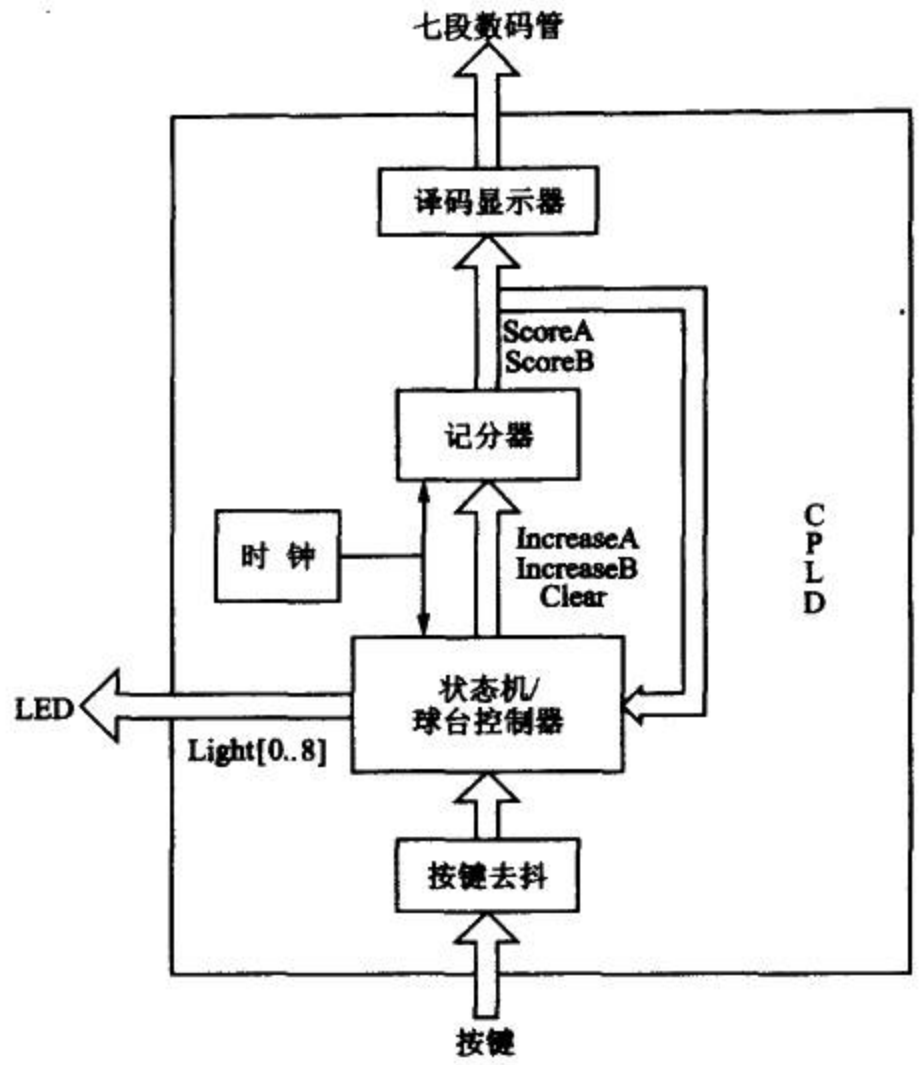


图9-2　系统的逻辑划分图

9.2.2　状态机的设计

根据题意，可以将系统状态机的状态数确定为6种，如表9-1所列。

表9-1　状态机的6种状态及其含义

状　态	含　义
WaitState	等待状态，等待A或B方的开球
AtoB	球从A向B方移动
BtoA	球从B向A方移动
AScore	A得1分
BScore	B得1分
FinalResult	比赛结束(最终判分)，在此状态下需按复位键，才能开始下一轮比赛

根据题目要求很容易画出如图9-3所示的状态转移图。

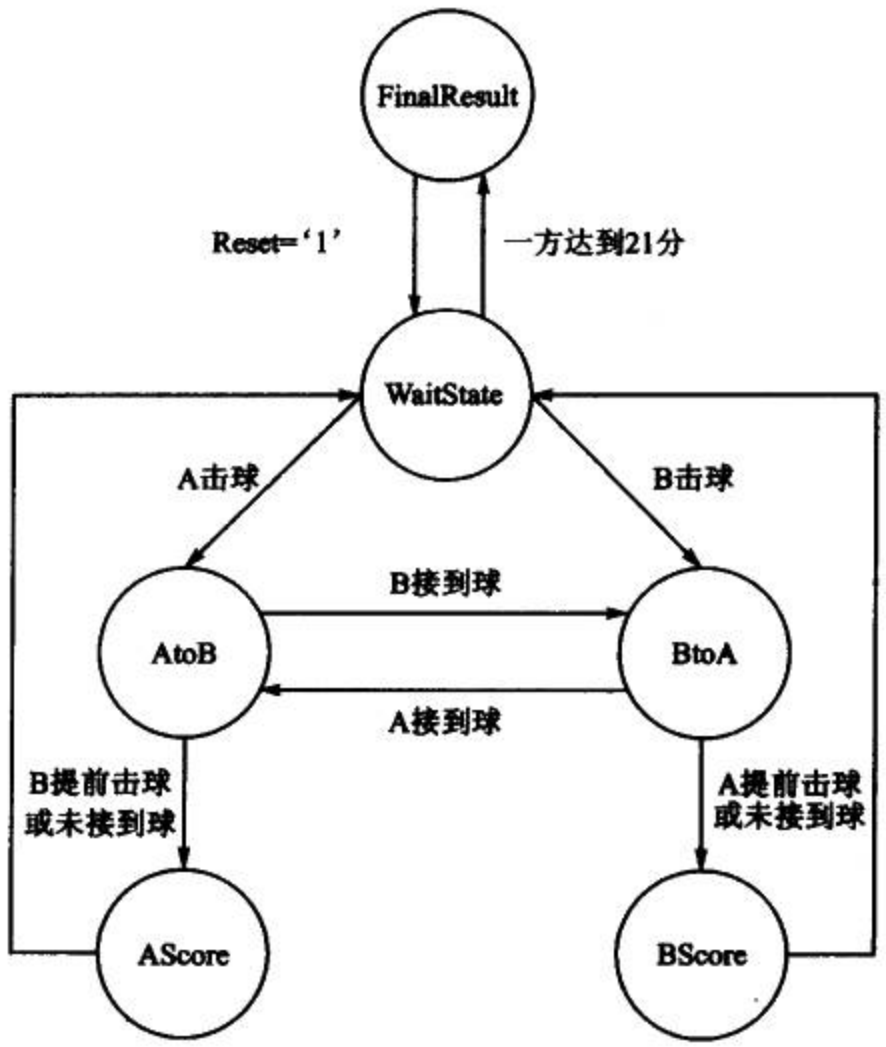


图9-3　乒乓游戏机状态转移图

有了状态转移图，要写状态机的程序就容易多了。由于本状态机的输入/输出较多，下面给出状态机的元件符号(见图9-4)及其输入/输出引脚的作用(见表9-2)。例程9-1为状态机的参考设计。

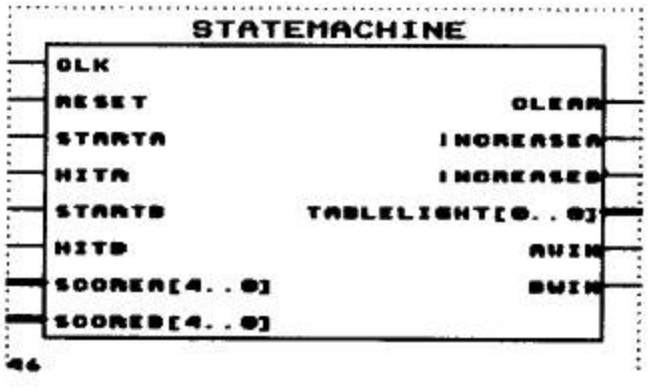


图9-4　状态机元件符号

表 9-2　输入/输出引脚的作用

Clk	10 Hz 的时钟，可由系统时钟分频得到。此时钟决定了球移动的速度，可根据实际需要调整
Reset	复位键，比赛重新开始，记分器清 0
StartA、StartB	A 和 B 双方的开球键
HitA、HitB	A 和 B 双方的击球键(可以将其与开球键合并)
Clear	将记分器清 0(给记分器的控制信号)
IncreaseA、IncreaseB	分别为 A、B 双方的加分信号(给记分器的控制信号)
ScoreA[4..0]、ScoreB[4..0]	A、B 双方的分数(由记分器给出)
Light[0..8]	接 9 个发光二极管
AWin、BWin	分别接发光二极管，表示 A 或 B 方胜出

例程 9-1　状态机/球台控制器参考程序

```
LIBRARY IEEE;
USE IEEE.Std_Logic_1164.ALL;
USE IEEE.Std_Logic_Unsigned.ALL;

ENTITY Statemachine IS
PORT
(Clk: IN Std_Logic;
 Reset:IN Std_Logic;
 StartA,HitA,StartB,HitB:IN Std_Logic;
 ScoreA,ScoreB:IN Integer RANGE 0 TO 21;
 Clear,IncreaseA,IncreaseB:OUT Std_Logic;
 TableLight:OUT Std_Logic_Vector(0 TO 8);
 AWin,Bwin:OUT Std_Logic
);
END;

ARCHITECTURE Behavior OF Statemachine IS
TYPE State_type IS (WaitState,AtoB,BtoA,AScore,BScore,FinalResult);
SIGNAL State:State_type;
SIGNAL TableState:Integer Range 0 To 8;
BEGIN
  PROCESS(Clk,Reset)
  BEGIN
    IF Reset='1' THEN                    --按下复位键，比赛重新开始
```

```
    State<=WaitState;                    --进入等待状态
    Clear<='1';                          --记分器清0
    AWin<='0';
    BWin<='0';
  ELSIF rising_edge(Clk) THEN
    CASE State IS
      WHEN WaitState=>
        Clear<='0';
        IncreaseA<='0';
        IncreaseB<='0';
        IF ((ScoreA=21) OR (ScoreB=21)) THEN
                                         --如果1方先达到21分,比赛结束
          State<=FinalResult;
        ELSE
          IF StartA='1' THEN             --如果A开球
            State<=AtoB;                 --球从A向B方移动
            TableState<=0;               --A方第1个灯点亮
          ELSE
            IF StartB='1' THEN           --如果B开球(A、B开球有一定的优先级区别)
              State<=BtoA;               --球从B向A方移动
              TableState<=8;             --B方第1个灯点亮
            ELSE
              State<=WaitState;
            END IF;
          END IF;
        END IF;
      WHEN AtoB=>                        --球从A向B移动的过程
        IF HitB='1' THEN                 --如果检测到B方击球
          IF TableState<=4 THEN          --若未过网提前击球
            State<=AScore;               --判为A胜
          ELSE
            State<=BtoA;                 --若过了网击球,球从B向A移动
          END IF;
        ELSE                             --若未检测到B方击球
          IF TableState=8 THEN           --如果离B方最近的灯已经亮了
            State<=AScore;               --判为A胜
          ELSE
            TableState<=TableState+1;    --否则球继续移动
          END IF;
```

```
    END IF;
  WHEN BtoA=>                    --球从B向A移动的过程,与AtoB状态相似,不再加注释
    IF HitA='1' THEN
      IF TableState>=4 THEN
        State<=BScore;
      ELSE
        State<=AtoB;
      END IF;
    ELSE
      IF TableState=0 THEN
        State<=BScore;
      ELSE
        TableState<=TableState-1;
      END IF;
    END IF;
  WHEN AScore=>                  --如果A胜
    IncreaseA<='1';              --A方加1分
    State<=WaitState;            --回到等待开球状态
  WHEN BScore=>
    IncreaseB<='1';
    State<=WaitState;
  WHEN FinalResult=>             --最后结果
    IF (ScoreA=21) then          --若A方达到21分
      AWin<='1';                 --表示A方胜出的灯亮
    ELSE
      BWin<='1';
    END IF;
  WHEN OTHERS=>
    State<=WaitState;
  END CASE;
 END IF;
END PROCESS;

PROCESS(Clk)                     --此进程控制9个发光二极管的亮灭
BEGIN
 IF falling_edge(Clk) THEN       --注意,此处是下降沿触发(为什么?)
  IF ((State=AtoB) OR (State=BtoA)) THEN
    CASE TableState IS
```

```
            WHEN 0=>TableLight<="100000000";
            WHEN 1=>TableLight<="010000000";
            WHEN 2=>TableLight<="001000000";
            WHEN 3=>TableLight<="000100000";
            WHEN 4=>TableLight<="000010000";
            WHEN 5=>TableLight<="000001000";
            WHEN 6=>TableLight<="000000100";
            WHEN 7=>TableLight<="000000010";
            WHEN 8=>TableLight<="000000001";
            WHEN OTHERS=>TableLight<="00000000";
          END CASE;
        ELSE
          TableLight<="000000000";
        END IF;
      END IF;
    END PROCESS;
END;
```

图 9-5 和图 9-6 为例程 9-1 的仿真波形，其中图 9-5 为 A 开球后 B 未在规定时刻击球的情况；而图 9-6 则是 B 在规定时刻反击而 A 却未接到球的情况。图中有清楚的标注，这里就不加详细说明了。

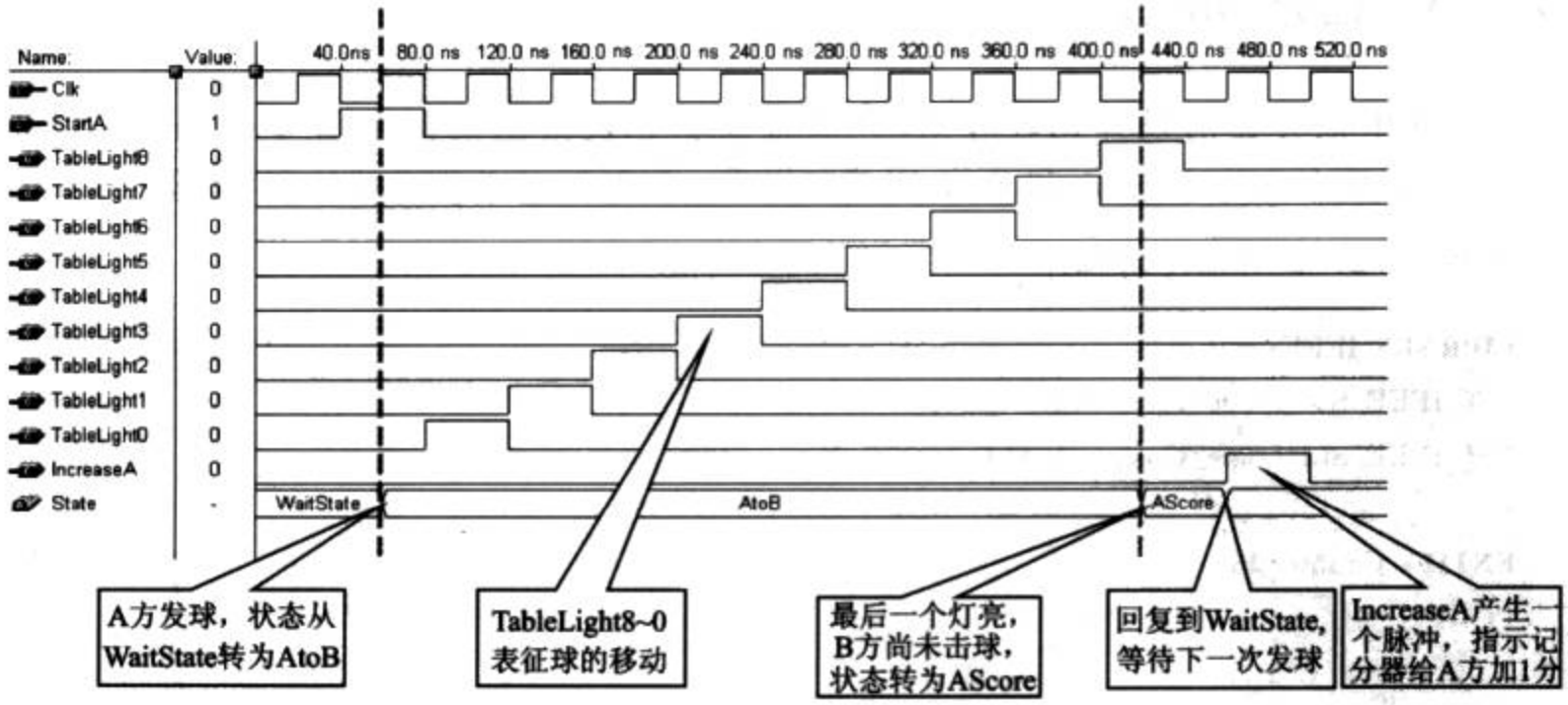


图 9-5　例程 9-1 的仿真波形 1

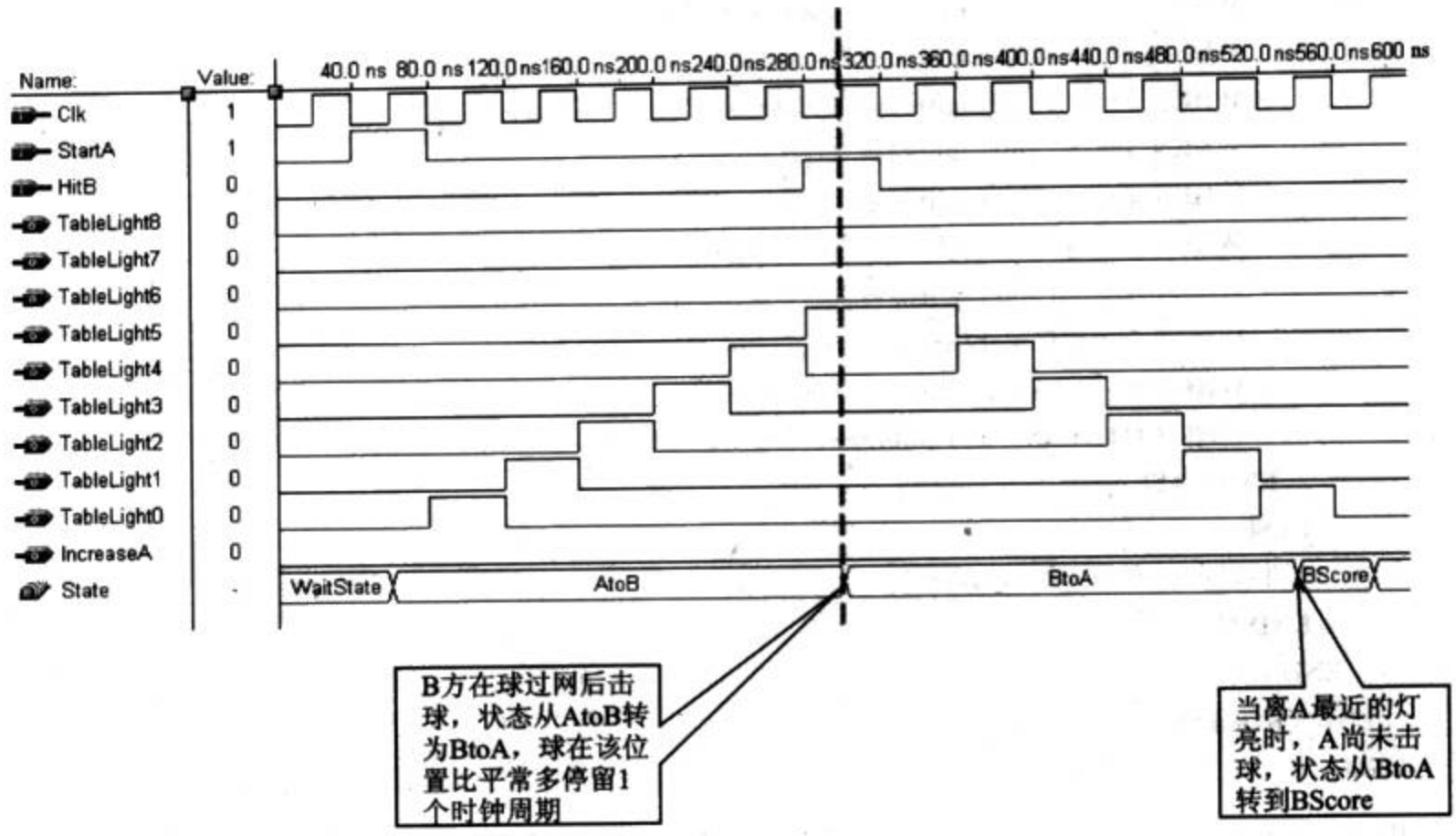


图9-6　例程9-1的仿真波形2

9.2.3　记分器的设计

本课题中记分器的设计比较容易，只须根据状态机给出的两个信号(IncreaseA 和 IncreaseB)对两个分数(ScoreA 和 ScoreB)进行操作。例程9-2为记分器的参考设计。

例程9-2　记分器参考程序

```
LIBRARY IEEE;
USE IEEE. Std_Logic_1164. ALL;
USE IEEE. Std_Logic_Unsigned. ALL;

ENTITY Counter IS
PORT
(Clk:IN Std_Logic;
 Clear:IN Std_Logic;
 IncreaseA,IncreaseB:IN Std_Logic;
 ScoreA,ScoreB:BUFFER Integer RANGE 0 TO 21
);
```

```
END;

ARCHITECTURE Count OF Counter IS
BEGIN
  PROCESS(Clk,Clear)
  BEGIN
    IF Clear='1' THEN                    --清0
      ScoreA<=0;
      SCoreB<=0;
    ELSIF falling_edge(Clk) THEN
      IF IncreaseA='1' THEN              --A方加1分
        ScoreA<=ScoreA+1;
      ELSIF IncreaseB='1' THEN           --B方加1分
        ScoreB<=ScoreB+1;
      END IF;
    END IF;
  END PROCESS;
END;
```

9.2.4　顶层文件元件连接图

图9-7给出了这个系统的顶层文件元件连接图(状态机与记分器部分)。

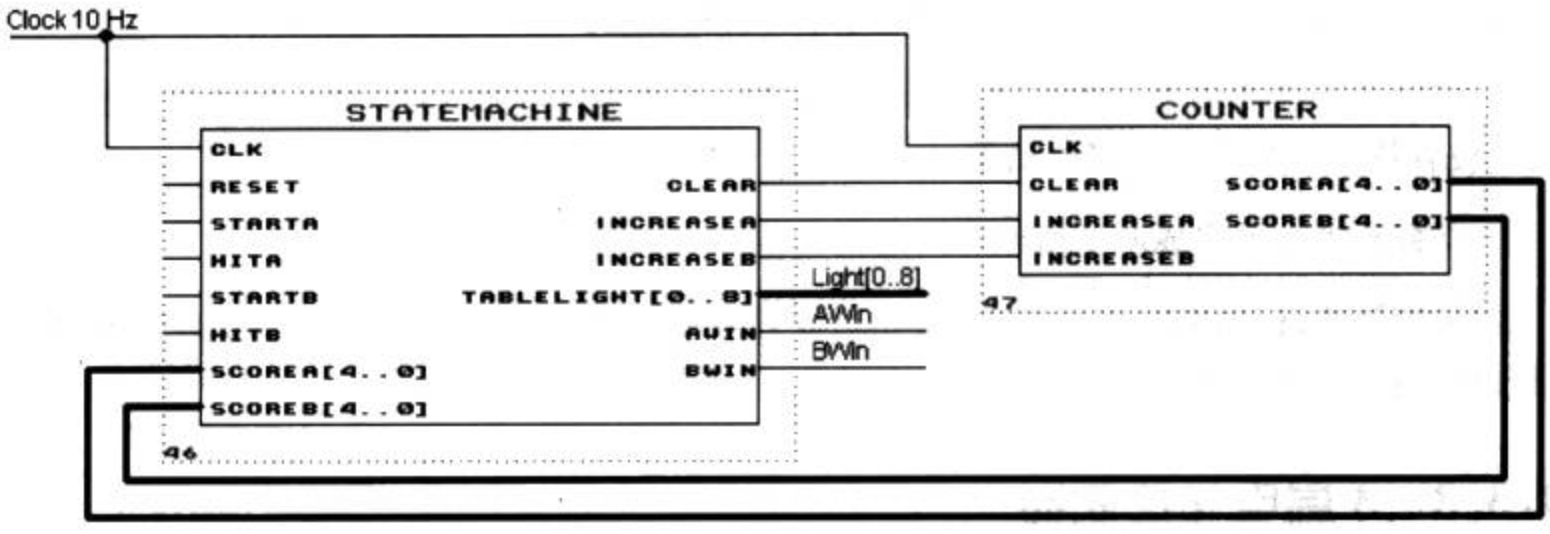


图9-7　乒乓球游戏机顶层文件元件连接图(状态机与记分器)

值得注意的是，此系统虽然简单，但其状态机的用法比较具有代表性，即状态机控制其他模块，而被控制的模块将某些信息反馈给状态机，状态机根据这些反馈信息，再做进一步的处理。

此外，还有一点需要提醒读者，在按键与 CPLD 相连的引脚需要加下拉电阻(300～1 000 Ω即可)，以便在没有按键按下时将输入也稳定在低电平，否则系统会不稳定。

9.3 练习题

9.3.1 乒乓游戏机功能扩展要求

在原有要求上扩充以下功能：

① 五局三胜制，能记录和显示双方赢得的局数；

② 发球权，双方按乒乓球比赛规则获得发球权，没有发球权的一方，发球开关无效；

③ 其他，读者可自行扩展其他功能。

9.3.2 经典数学游戏——过河

请把一个人将 1 只狗、1 只猫和 1 只老鼠渡过河的经典数学游戏用 VHDL 描述并实现。该游戏要求：独木舟 1 次只能装载人和 1 只动物，且猫和狗不能单独在一起，而猫和老鼠也不能友好相处，若能将 3 只动物安全渡过河，则游戏胜利结束。

① 用发光二极管以适当的方式表示各种动物、河和船以及过河的动作(从此岸到彼岸的发光二极管依次点亮表示过河的动作)；

② 以数码管显示完成游戏所经历的过河次数；

③ 调置复位键，当游戏失败后，以发光二极管或数码管显示 E 等方式来表示此时程序挂起，按复位键可重新开始一轮游戏。

9.3.3 3 层电梯控制器

(1) 简要说明

电梯控制器是控制电梯按顾客的要求自动上、下的装置。

(2) 任务和要求

① 每层电梯入口处设有上、下请求开关,电梯内设有乘客至所达楼层的停站请求开关。

② 设有电梯所处位置指示装置及电梯运行模式(上升或下降)指示装置。

③ 电梯每秒升(降)1层楼。

④ 电梯到达有停站请求的楼层后,经过1 s电梯门打开,开门指示灯亮,开门4 s后,电梯门关闭(开门指示灯灭),电梯继续运行,直至执行完最后一个请求信号后停在当前层。

⑤ 能记忆电梯内、外的所有请求信号,并按照电梯运行规则按顺序响应,每个请求信号保留至执行后消除。

⑥ 电梯运行规则:当电梯处于上升模式时,只响应比电梯所在位置高的上楼请求信号,由下而上逐个执行,直到最后一个上楼请求执行完毕;若更高层有下楼请求,则直接升到有下楼请求的最高层,然后便进入下降模式。当电梯处于下降模式时,则与上升模式相反。

⑦ 电梯初始状态为一层开门。

⑧ 选做:到达各层时有音响提示、故障报警、更多层的设计。

(3) 训练目标

此系统的难点在于复杂的状态机设计,如果读者能独立完成此题,将对状态机的概念以及VHDL实现有一个较深刻的理解。状态机是数字系统中的一个十分重要的概念,希望读者能通过设计3层电梯控制器切实地掌握它。

第10章

数字频率计

10.1 任务书

(1) 简要说明

根据频率计的测频原理，按照测频量程需要，选择合适的时基信号即闸门时间，对输入被测信号进行计数，实现测频的目的。

(2) 任务和要求

设计一个3位十进制数字显示的数字式频率计，其频率测量范围在1 MHz内。量程分为10 kHz、100 kHz和1 MHz三档，即最大读数分别为9.99 kHz、99.9 kHz和999 kHz。这里要求量程能够自动转换，具体要求如下：

① 当读数大于999时，频率计处于超量程状态，下一次测量时，量程自动增大1档。

② 当读数小于099时，频率计处于欠量程状态，下一次测量时，量程自动减小1档。

③ 当超过频率测量范围时，显示器显示溢出。

④ 采用记忆显示方式，即计数过程中不显示数据，待计数过程结束后，显示测频结果，并将此显示结果保持到下次计数结束。显示时间不短于1 s。

⑤ 小数点位置随量程变化自动移位。

⑥ 选做：增加测周期功能(量程分为1 ms、10 ms、100 ms三档，即最大读数为9.99 ms、99.9 ms和999 ms)。

(3)训练目标

掌握测频、测周的基本原理，会根据系统的实际情况划分状态，进一步掌握状态机的设计与应用。

10.2　频率与周期的测量原理

10.2.1　测频的原理

测频的原理归结成一句话，即在单位时间内对被测信号进行计数。图 10－1 说明了测频的原理及误差产生的原因。

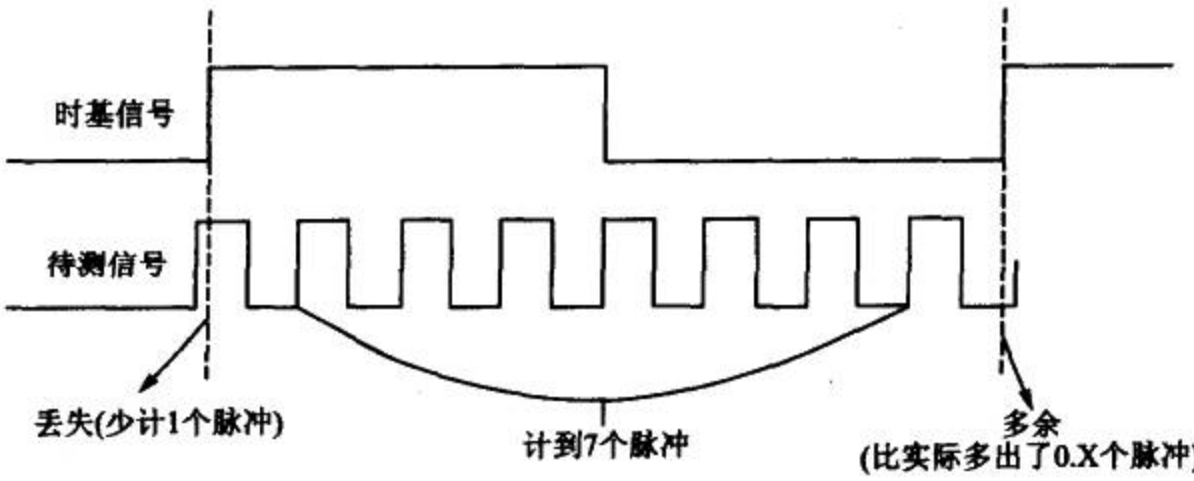


图 10－1　测频原理

在图 10－1 中，假设时基信号为 1 kHz，则用此法测得的待测信号为 1 kHz×7＝70 kHz。但从图中可以看出，待测信号应该在 77 kHz 左右，误差约有 7/77～9.1%。这个误差是比较大的，实际上，测量的脉冲个数的误差会在±1 之间。假设所测得的脉冲个数为 N，则所测频率的最大误差为 $\delta=\frac{1}{N-1}\times 100\%$。显然，减小误差的方法就是增大 N。本频率计要求测量结果以 3 位数表示，则测频误差应为 1%～0.1%，则 N 的取值范围为：

$$100 \leqslant N < 1\ 000$$

即时基信号的频率为量程中最低频率的 1/100(同时约为最高频率的 1/1 000)。通过计算，得出表 10－1中的数据。

表 10－1　待测信号与时基信号的关系

待测信号/kHz	时基信号/Hz
100～999	1 000
10～99.9	100
1～9.99	10

10.2.2　测周期的原理

测量周期法用一句话概括，就是在被测信号周期时间内对某一个基准脉冲进行计数。图 10－2说明了测周期的原理及误差的产生的原因。

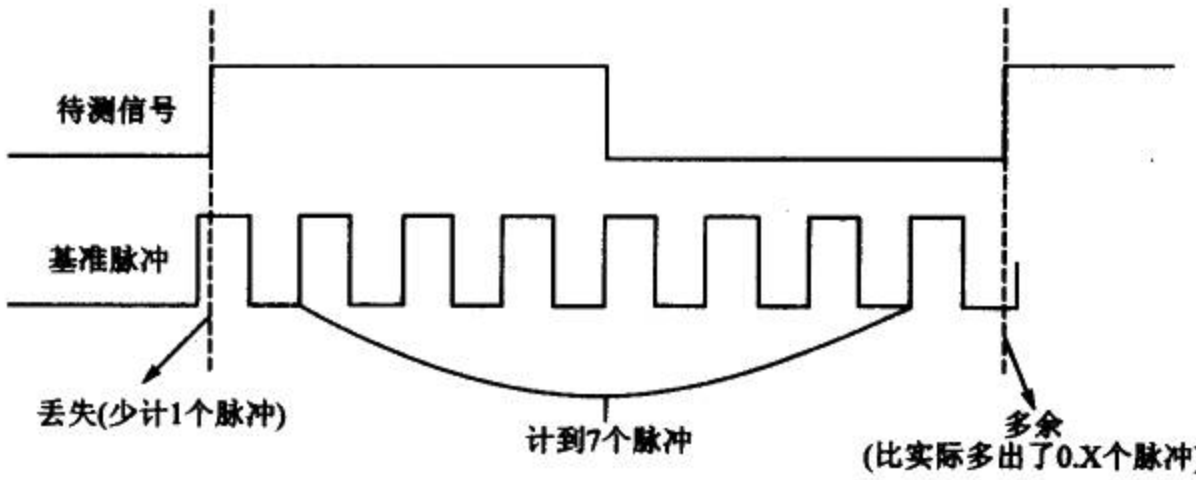


图 10－2　测周期原理

读者们肯定发现了，图 10－2 和图 10－1 的差别仅仅是待测信号与时基信号颠倒了位置。事实上，测频和测周的惟一区别就在于计数的究竟是待测信号，还是系统提供的时基信号(基准脉冲)。

以图 10－2 为例，假设基准脉冲为 1 ms，在一个待测信号周期内计到 7 个基准脉冲，则测得的待测信号周期为 70 ms。与测频一样，此法同样存在误差，而最大误差也同样为 $\delta=\frac{1}{N-1}\times 100\%$。若要求测量误差在 1%以下，则待测信号与基准脉冲的关系如表 10－2 所列。

表 10－2　待测信号与基准脉冲的关系

待测信号/ms	基准脉冲/kHz
100～999	1
10～99.9	10
1～9.99	100

10.3　参考设计

10.3.1　系统框图

根据题意，可以大概划分出系统需要的几个功能模块。

① 分频器：由于测频、测周时不同量程档需要不同的时基信号，分频模块是必不可少的。

系统通过分频模块从晶振时钟(例如 20 MHz)分出系统所需的几个时基信号,其中包括状态机所需时钟。

② 计数器:从 10.2 节的介绍可以看出,测频、测周本质上即是计数,所以计数器也是系统中不可或缺的模块。为实现量程的自动转换,本计数器需输出指示超量程和欠量程状态的信号。

③ 状态机:题目要求本系统可以自动切换量程,如果把每个档量程看成一个状态,那么量程的切换实际上就是状态的转换。显然,用状态机来实现是最方便的。

④ 时钟选择器:从 10.2 节中很容易看出,测量周期和测量频率的区别仅在于系统的基准脉冲是用来计数的还是用来被计数的。显然,不必为测频和测周各写一个计数器,只需用一个时钟选择器就可以完成测频和测周的切换了。

⑤ 其他:如锁存器(用来使显示数字稳定)、同步整形电路(用来处理超量程信号和欠量程信号,避免状态机进行多次状态转移)和七段数码管译码电路等。

系统框图如图 10-3 所示。

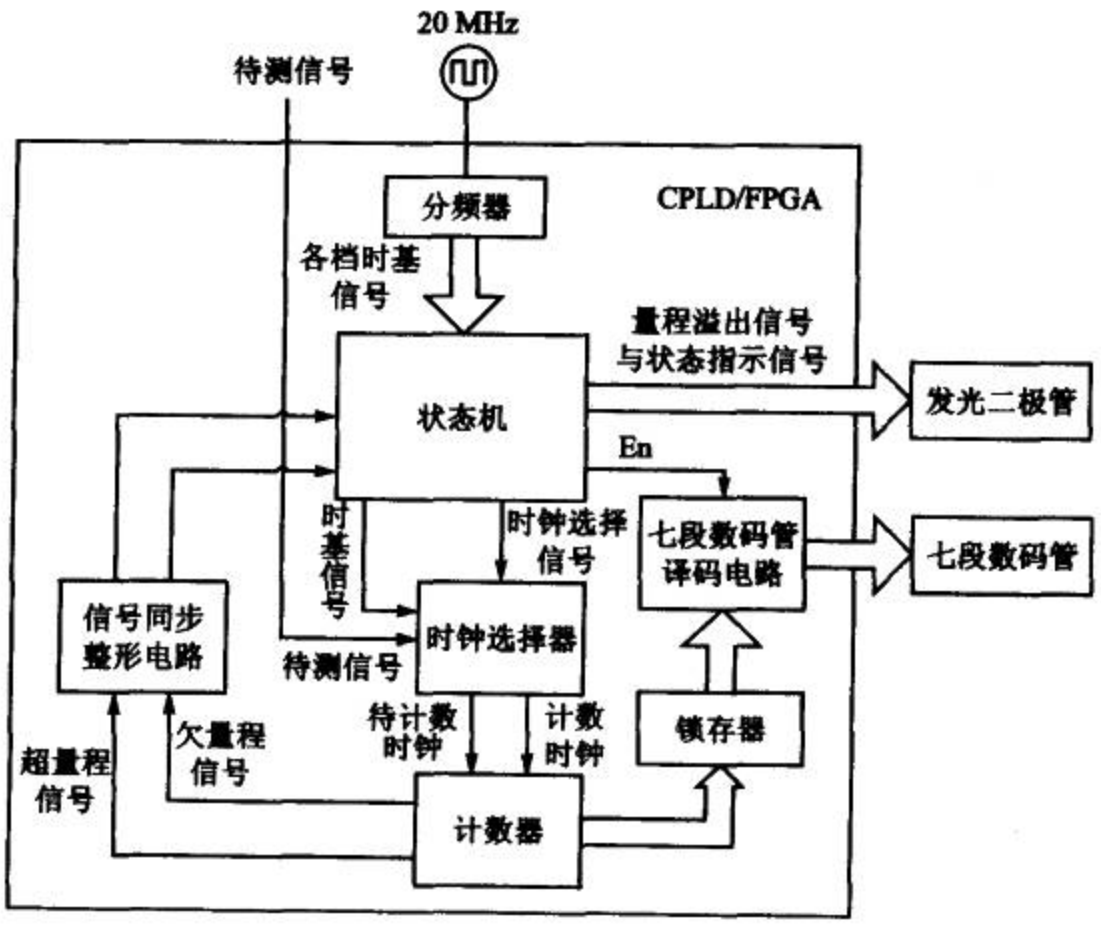


图 10-3　数字频率计系统框图

10.3.2　状态机的设计

如前所述，系统的状态机是用来自动切换量程的。根据题目要求，我们可以设立 8 个状态，状态名及其对应的量程如表 10－3 所列。

表 10－3　状态名及其对应的量程

状态名	对应量程	时基信号/Hz
F1M	100～999 kHz(测频)	1 000
F100K	10～99.9 kHz(测频)	100
F10K	1～9.99 kHz(测频)	10
P1ms	1～9.99 ms(测周)	1 000
P10ms	10～99.9 ms(测周)	100
P100ms	100～999 ms(测周)	10
OverError	超过数字频率计最高量程	1 000
LowError	低于数字频率计最低量程	10

假设计数器输出的超量程信号为 Over，欠量程信号为 Low，那么可以画出如图 10－4 所示的状态转移图。根据此状态转移图，很容易就可以写出对应的状态机程序(例程 10－1)。

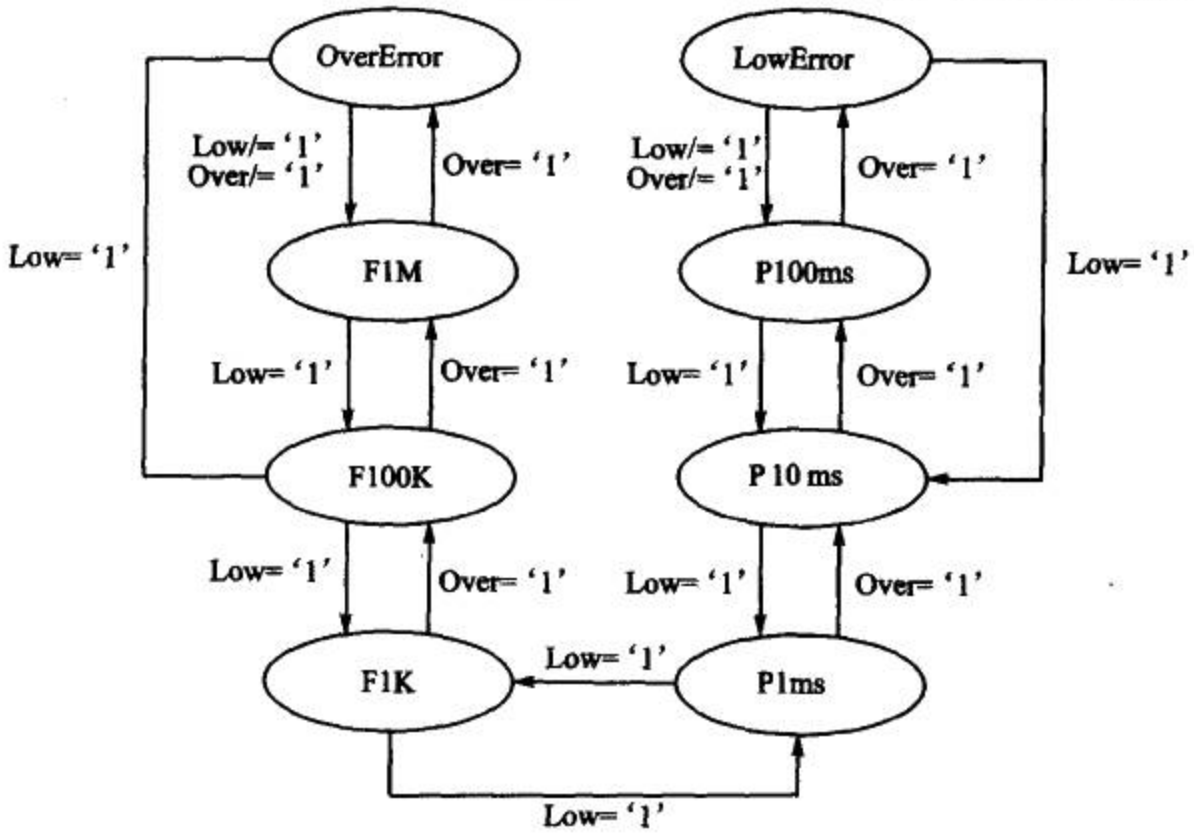


图 10－4　状态转换图

例程 10－1　状态机程序

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

ENTITY Statemachine IS
PORT
( Clock: IN Std_Logic;
 Clock10Hz,Clock100Hz,Clock1kHz,Clock10kHz,Clock100kHz:IN Std_Logic;
 Low,Over:IN Std_Logic;                    --欠量程信号和超量程信号
 Reset:IN Std_Logic;
 En:OUT Std_Logic;                         --控制七段数码管的显示(待测信号超过总量程时不显示数字)
 Dp1,Dp2:OUT Std_Logic;                    --Dp1 为百位的小数点,Dp2 为十位的小数点
 Overlight,Lowlight:OUT Std_Logic;         --待测信号频率或周期超过总量程的指示灯
 Period:OUT Std_Logic;                     --用以指示测频和测周期状态,1 为测周
 Outclock:OUT Std_Logic                    --输出时基信号
);
END;

ARCHITECTURE MooreMachine OF Statemachine IS
TYPE State_type Is (F1M,F100K,F10K,P1ms,P10ms,P100ms,Overerror,Lowerror);
SIGNAL State:State_type;
BEGIN
  PROCESS(Clock100kHz,Reset)
  BEGIN
    IF (Reset='1') THEN
      State<=F10K;
    ELSIF rising_edge(Clock100kHz) THEN
      CASE State IS
      WHEN F1M=>
        IF Over='1' THEN
          State<=Overerror;
        ELSIF Low='1' THEN
          State<=F100K;
        ELSE
          State<=F1M;
        END IF;
      WHEN F100K=>
```

```
    IF Over='1' THEN
      State<=F1M;
    ELSIF Low='1' THEN
      State<=F10K;
    ELSE
      State<=F100K;
    END IF;
  WHEN F10K=>
    IF Over='1' THEN
      State<=F100K;
    ELSIF Low='1' THEN
      State<=P1ms;
    ELSE
      State<=F10K;
    END IF;
  WHEN P1ms=>
    IF Over='1' THEN
      State<=P10ms;
    ELSIF Low='1' THEN
      State<=F10K;
    ELSE
      State<=P1ms;
    END IF;
  WHEN P10ms=>
    IF Over='1' THEN
      State<=P100ms;
    ELSIF Low='1' THEN
      State<=P1ms;
    ELSE
      State<=P10ms;
    END IF;
  WHEN P100ms=>
    IF Over='1' THEN
      State<=Lowerror;
    ELSIF Low='1' THEN
      State<=P10ms;
    ELSE
```

```
        State<=P100ms;
      END IF;
    WHEN Overerror=>
      IF Low='1' THEN
        State<=F100K;
      ELSIF Over='1' THEN
        State<=Overerror;
      ELSE
        State<=F1M;
      END IF;
    WHEN Lowerror=>
      IF Low='1' THEN
        State<=P10ms;
      ELSIF Over='1' THEN
        State<=Lowerror;
      ELSE
        State<=P100ms;
      END IF;
  END CASE;
END IF;
END PROCESS;

PROCESS(State)                          --原则上,这里应将所有在进程中可能被读取的信号
                                        --都加入敏感信号表,但 Max+plus II 自动将本格式
                                        --综合为组合进程,为了书写和阅读的方便,这里
                                        --仅将 State 列入敏感信号表。理论与实践之间的差别,
                                        --在此可见一斑
BEGIN
  CASE State IS
    WHEN F1M=>
      En<='1';                          --允许七段数码管显示数字
      Outclock<=Clock1kHz;              --时基信号为 1 kHz
      Overlight<='0';                   --待测信号频率未超过总量程
      Lowlight<='0';
      Period<='0';                      --表示系统处于测频状态
      Dp1<='0';                         --百位上小数点不亮
      Dp2<='0';                         --十位上小数点不亮
```

```
WHEN F100K=>
  Outclock<=Clock100Hz;
  Overlight<='0';
  Lowlight<='0';
  Period<='0';
  En<='1';
  Dp1<='0';
  Dp2<='1';
WHEN F10K=>
  Outclock<=Clock10Hz;
  Overlight<='0';
  Lowlight<='0';
  En<='1';
  Dp1<='1';
  Dp2<='0';
  Period<='0';
WHEN P1ms=>
  En<='1';
  Outclock<=Clock100kHz;
  Overlight<='0';
  Lowlight<='0';
  Period<='1';
  Dp1<='1';
  Dp2<='0';
WHEN P10ms=>
  En<='1';
  Outclock<=Clock10kHz;
  Overlight<='0';
  Lowlight<='0';
  Period<='1';
  Dp1<='0';
  Dp2<='1';
WHEN P100ms=>
  En<='1';
  Outclock<=Clock1kHz;
  Overlight<='0';
  Lowlight<='0';
```

```
        Period<='1';
        Dp1<='0';
        Dp2<='0';
      WHEN Overerror=>
        En<='0';
        Outclock<=Clock1kHz;
        Overlight<='1';
        Lowlight<='0';
        Period<='0';
      WHEN Lowerror=>
        Outclock<=Clock1kHz;
        En<='0';
        Lowlight<='1';
        Overlight<='0';
        Period<='1';
    END CASE;
  END PROCESS;
END;
```

10.3.3　计数器的设计

计数器的原理很简单，计数值大于 999，就将超量程信号 Over 置 1；若计数值小于 100，则将欠量程信号 Low 置 1。

本计数器在待计数信号的两个时钟周期内完成计数与控制信号(Over 与 Low)的传输，在量程合适的情况下，将计数值输出。在这两个时钟周期内，第 1 个时钟周期完成计数，第 2 个时钟周期完成控制信号的传输与计数值输出。这样做的好处是稳定，将计数与控制信号传输分开进行，避免了时钟跳变。例程 10－2 给出计数器程序。

例程 10－2　计数器程序

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

ENTITY Counter IS
PORT
( Clock1,Clock2:IN Std_Logic;                    --Clock1 为待计数信号,Clock2 为计数时钟
 Result1,Result2,Result3:OUT Integer RANGE 0 TO 9;
```

```
 Over,Low:OUT Std_Logic                    --超量程、欠量程信号,传给状态机
);
END;

ARCHITECTURE Behavior OF Counter IS
SIGNAL En,EnTransfer:Std_Logic;
SIGNAL Num1,Num2,Num3:Integer RANGE 0 TO 9;
SIGNAL Overmode:Std_Logic;

BEGIN
  PROCESS(Clock1)
  BEGIN
    IF rising_edge(Clock1) THEN           --En 信号其实是 Clock1 的二分频,
      En<=NOT En;                         --En 为低电平时计数,En 为高电平时传输信号与计数值
    END IF;
  END PROCESS;

  PROCESS(Clock2)
  BEGIN
    IF rising_edge(Clock2) THEN
      IF En='1' THEN                      --En='1'时传输数据
        IF EnTransfer='1' THEN            --EnTransfer 用以保证数值与信号只传输一次
          EnTransfer<='0';
          IF Overmode='1' THEN            --超量程的情况
            Low<='0';
            Over<='1';
            Overmode<='0';
          ELSIF Num3=0 THEN               --欠量程的情况
            Low<='1';
            Over<='0';
          ELSE
            Low<='0';
            Over<='0';
            Result1<=Num1;
            Result2<=Num2;
            Result3<=Num3;
          END IF;
```

```
        Num1<=0;
        Num2<=0;
        Num3<=0;
        END IF;
      ELSE                                    --En='0'时计数
        EnTransfer<='1';
        Low<='0';
        Over<='0';
        IF Num1=9 THEN
      IF Num2=9 THEN
        IF Num3=9 THEN
          Overmode<='1';
        ELSE
          Overmode<='0';
          Num3<=Num3+1;
          Num2<=0;
          Num1<=0;
        END IF;
      ELSE
          Overmode<='0';
          Num2<=Num2+1;
          Num1<=0;
        END IF;
      ELSE
          Overmode<='0';
          Num1<=Num1+1;
        END IF;
      END IF;
    END IF;
  END PROCESS;
END;
```

计数器的仿真波形图如图 10－5 所示。

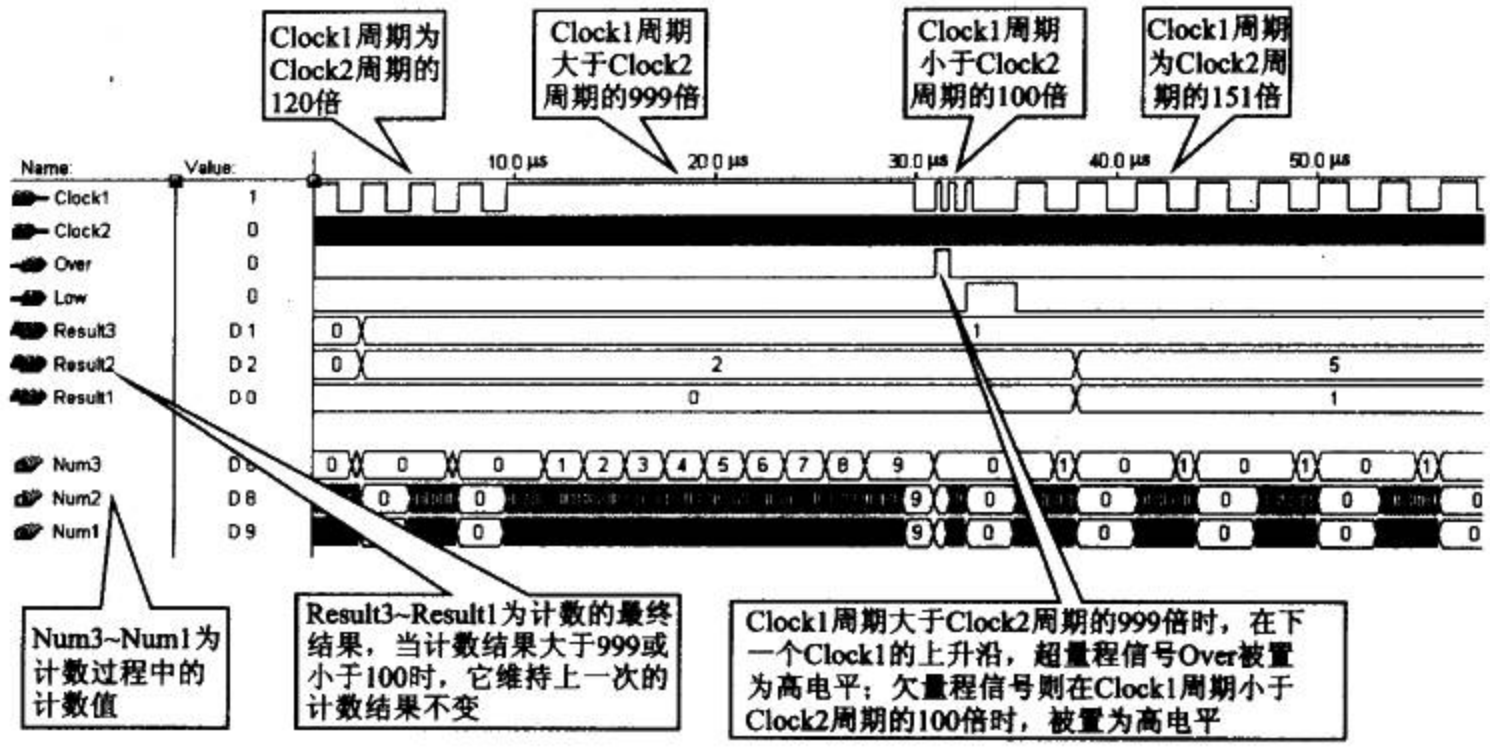


图 10－5　计数器的仿真波形

10.3.4　顶层文件元件连接图

图 10－6 为数字频率计的顶层文件图。其中 ClkProduct 是分频模块，Statemachine 为状态机，ClockMux 为时钟选择器，Counter 为计数器，SignalLatch 为同步整形电路，NumLatch 为锁存器，ShowNum 为七段数码管译码器。

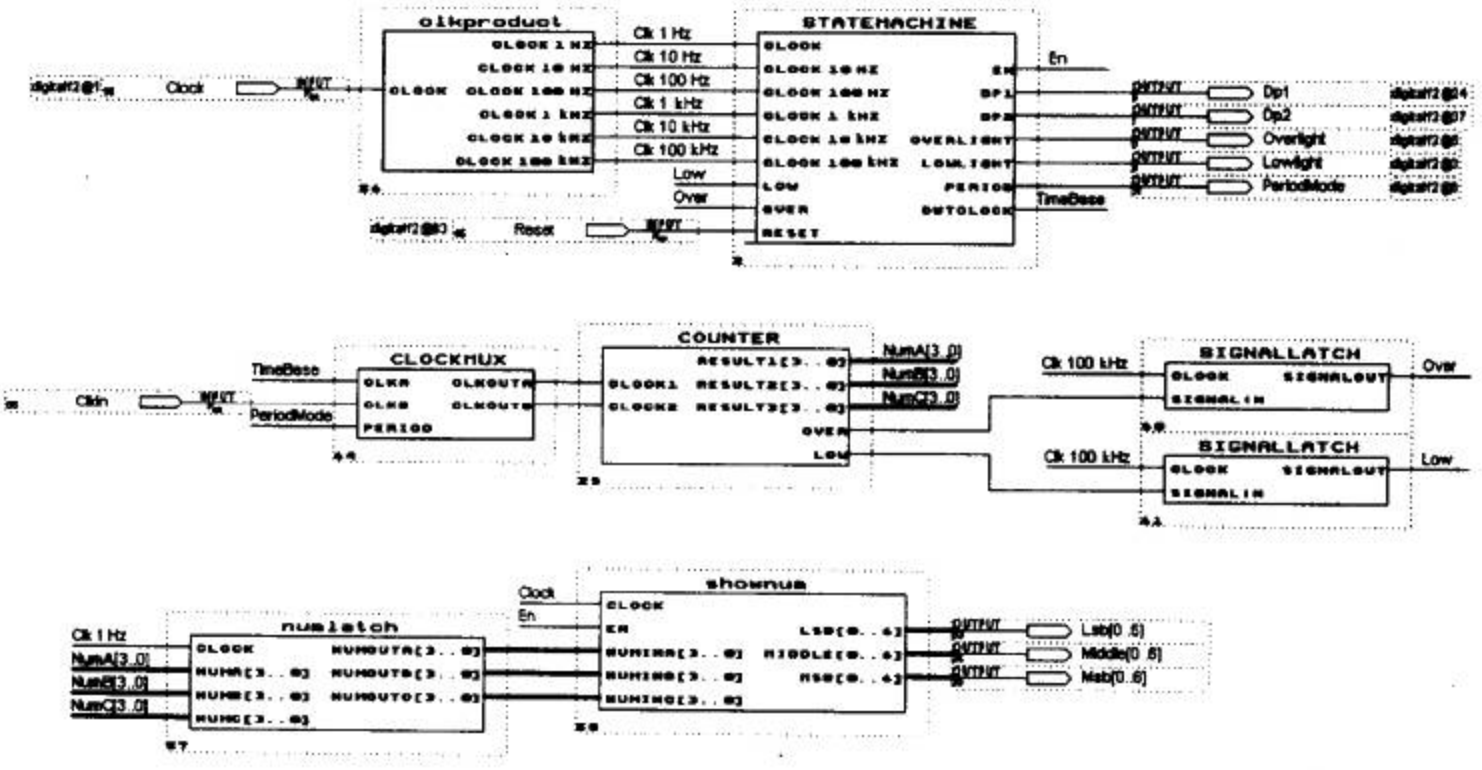


图 10－6　数字频率计顶层文件图

图中的输入/输出引脚的作用如表 10－4 所列。

表 10－4　输入/输出引脚的作用

引脚名	作　用
Clock	接晶振时钟(20 MHz)
Clkin	接待测信号
Dp1	接百位的小数点
Dp2	接十位的小数点
OverLight	接表征高溢出的发光二极管
LowLight	接表征低溢出的发光二极管
PeriodMode	接表征测周期状态的发光二极管
Lsb[0..6]	接个位七段数码管
Middle[0..6]	接十位七段数码管
Msb[0..6]	接百位七段数码管

上面给出了状态机和计数器的例程，对于其他的模块，有的在前面章节中已经给出程序，有的则相对简单，为节省篇幅，此处不再给出例程。

10.4　练习题

10.4.1　洗衣机控制器

设计、制作一个简易全自动洗衣机控制器：

① 洗衣机的动作有洗涤、漂洗和脱水，每个动作持续的时间分别为 20 s、15 s 和 10 s；

② 用一个按键实现洗衣程序的手动选择：

A 单洗涤　　B 单漂洗　　C 单脱水　　D 漂洗和脱水

E 洗涤、漂洗和脱水的全过程；

③ 无论选择何种洗衣服的程序，在所选择的程序完成之后，控制器应处于暂停状态；

④ 用一个按键实现暂停洗衣和继续洗衣的控制，暂停后继续洗衣应回到暂停之前的状态；

⑤ 用发光二极管指示状态；

⑥ 用数码管以倒计数的方式显示当前状态的剩余时间。

10.4.2　数字钟

设计一个简单的能显示时、分、秒的数字钟，具体要求如下：

① 计时可选十二进制计时和二十四进制计时；

② 可手动校时，能分别进行时、分的校正；

③ 带闹钟功能(用发光二极管表示，或接扬声器)，当计时计到闹铃时间时，发光二极管点亮，闹铃时间为 1 min(分)，可用按键提前终止闹铃；

④ 带秒表功能；

⑤ 选做：带日历显示，可显示月、日等。

第11章

自动打铃系统

11.1 任务书

(1) 基本要求

① 基本计时和显示功能(用 12 小时制显示):包括上下午标志,时、分的数字显示,秒信号显示。

② 能设置当前时间(含上、下午,时、分)。

③ 能实现基本打铃功能,规定:

上午 06:00 起床铃,打铃 5 s,停 2 s,再打铃 5 s;
下午 10:00 熄灯铃,打铃 5 s,停 2 s,再打铃 5 s。

铃声可用 LED 灯光显示,如果实验装置没有 LED 发光管,那么可以用七段显示管的小数点显示,也可以用显示小时的十位数码管的多余段显示。凡是用到铃声功能的均可如此处理。

(2) 发挥部分

① 增加整点报时功能,整点时响铃 5 s。要求有控制启动和关闭功能。
② 增加调整起床铃、熄灯铃时间的功能。
③ 增加调整打铃时间长短和间歇时间长短的功能。
④ 增设上午 4 节课的上、下课打铃功能,规定:

7:30 上课,8:20 下课;8:30 上课,9:20 下课;9:40 上课,10:30 下课;10:40 上课,11:30 下课。每次响铃 5 s。

⑤ 特色与创新(自选)。

11.2　参考设计

本题是 2002 年北京市大学生电子设计竞赛题,本节给出的参考设计为当时参加竞赛获得一等奖的作品(由于原作品加入不少创新功能,所以程序很长,此处为了方便理解程序,将创新部分功能去掉,以缩短代码长度)。本节不准备用很大篇幅来讲设计思想,读者有了前面几章的基础,应该具备自己划分模块并具体实现这些模块的能力了,因此本节只给出系统框图、各模块 VHDL 程序及顶层文件元件连接图,希望读者通过阅读这些程序及其注释来体会其中的设计思想。

11.2.1　系统框图

按照题目的要求,可以大致将系统划分为以下几个模块:

① 状态机,系统有多种显示模式(如显示当前时间,显示上午打铃时间,显示下午打铃时间等),如果把每种模式当成一种状态,那么用状态机来进行模式的切换是最方便的,因此可以采用状态机作为中心控制模块。

② 计时模块,本系统中有一部分功能类似于数字钟,因此计时模块是必不可少的。

③ 打铃时间设定模块,系统要求打铃时间可调,由于此功能相对独立,可以单独用一个模块来实现。

④ 打铃长度设定模块,用以设定打铃时间的长短。

⑤ 显示控制及打铃控制模块,根据当前时间(由计时模块输出)、打铃时间(由打铃时间设定模块输出)等信息决定当前的输出数字和打铃信号灯的亮、灭情况。

⑥ 其他,如分频模块、七段数码管译码电路等,如果实验板上没有按键消抖电路,那么必须再加上按键消抖电路。

系统框图如图 11 - 1 所示(为方便起见,图中未标明时钟)。

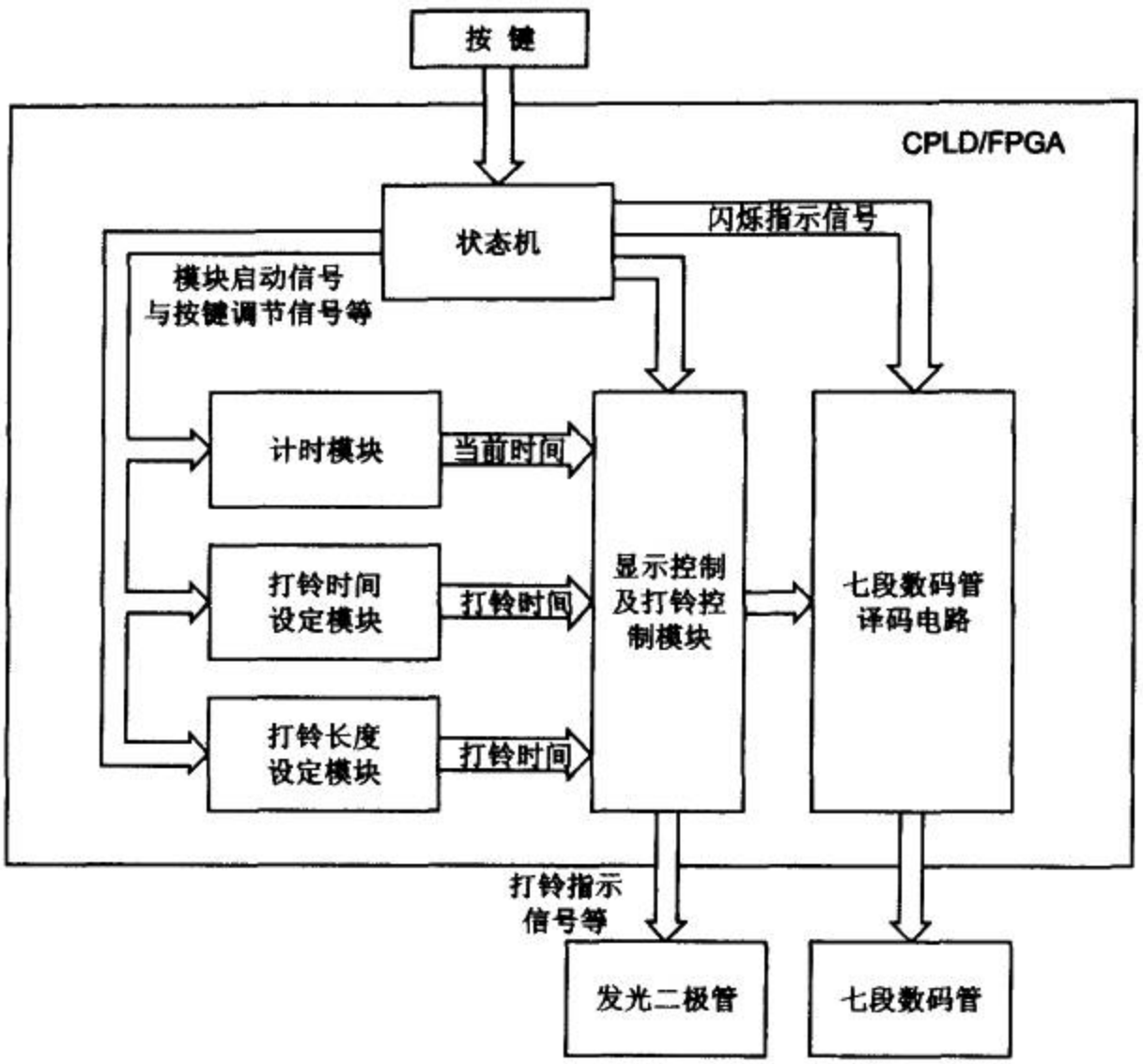


图 11-1　自动打铃系统框图

11.2.2　状态机的设计

如前所述，将每个功能模式当作一种状态，因此设定的状态如表 11-1 所列。

表 11-1　状态与功能模式对照表

状态名	功　能
Timer	正常计时
Adj_Time	调整当前时间
Adj_MorningTime	调整起床时间
Adj_EveningTime	调整熄灯时间
Adj_RingLength	调整打铃时间长度
Adj_12or24	切换时制(12 小时或 24 小时)

从图 11－1 中可以看出，状态机通过读入按键值进行状态切换并执行某些特定的操作。那么可以模仿电子表来设计这个系统，采用 4 个按键来完成全部操作。这里，状态机只使用 3 个按键（另一个按键用于直接切换是否整点报时，接到显示控制及打铃控制模块）。这 3 个按键的信号名及其作用如表 11－2 所列。

表 11－2　按键名及其作用

按键名	作　用
ChangeMode	切换当前功能模式（即切换状态）
AdjPosition	当调整时间（包括打铃时间等）时，用以切换当前位置（时位、分位或秒位）
AdjVal	调整当前所在位置的数字

这里将 6 个七段数码管分为 3 部分，分别代表时、分和秒，在 VHDL 程序中用一个信号 Pos 来表示（如图 11－2 所示）。当 AdjPosition＝1 时，通过 Pos 的自加来改变当前的闪烁位置（用闪烁来表示当前正在调整的是时、分还是秒）。

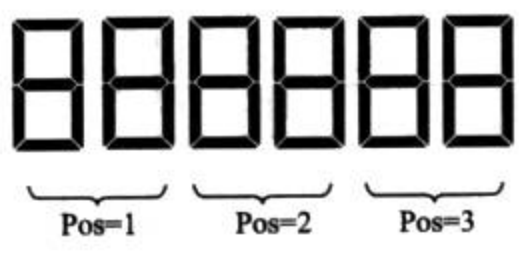


图 11－2　七段数码管位置与 Pos 的对应关系图

例程 11－1 为完整的状态机参考程序。

例程 11－1　状态机参考程序

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

ENTITY Statemachine IS
PORT
(Clock:IN Std_Logic;
 ChangeMode:IN Std_Logic;
 AdjPosition:IN Std_Logic;
 AdjVal:IN Std_Logic;
 Command:BUFFER Std_Logic_Vector(4 DOWNTO 0);    --用以指示当前所处状态
 AdjBotton:OUT Std_Logic_Vector(2 DOWNTO 0);      --AdjBotton(2)为调时信号
                                                  --AdjBotton(1)为调分信号
```

```
                                                    --AdjBotton(0)为调秒信号
  Flash:OUT Std_Logic_Vector(2 DOWNTO 0)            --指示七段数码管进行闪烁的信号
                                                    --Flash(2)指示时位闪烁
                                                    --Flash(1)指示分位闪烁
                                                    --Flash(0)指示秒位闪烁
);
END;

ARCHITECTURE Mealymachine OF Statemachine IS
TYPE State_type IS (Timer,Adj_Timer,Adj_MorningTime,Adj_EveningTime,
                    Adj_RingLength,Adj_12or24);
SIGNAL State:State_type;
SIGNAL Pos:Integer RANGE 0 TO 3;

BEGIN
  PROCESS(Clock)                                    --此进程用以进行状态转换
  BEGIN
    IF rising_edge(Clock) THEN
      CASE State IS
    WHEN Timer=>
      IF ChangeMode='1' THEN
        State<=Adj_Timer;
      END IF;
  WHEN Adj_Timer=>
    IF ChangeMode='1' THEN
      State<=Adj_MorningTime;
    END IF;
  WHEN Adj_MorningTime=>
    IF ChangeMode='1' THEN
      State<=Adj_EveningTime;
    END IF;
  WHEN Adj_EveningTime=>
    IF ChangeMode='1' THEN
      State<=Adj_RingLength;
    END IF;
  WHEN Adj_RingLength=>
    IF ChangeMode='1' THEN
```

```
        State<=Adj_12or24;
      END IF;
    WHEN Adj_12or24=>
      IF ChangeMode='1' THEN
        State<=Timer;
      END IF;
    WHEN Others=>
      State<=Timer;
    END CASE;
  END IF;
END PROCESS;

PROCESS(Clock)
BEGIN
  IF rising_edge(Clock) THEN
    CASE State IS
      WHEN Timer=>
        Command<="00000";          --表示不启动任何调整参数(时间,打铃长度等)的
                                   --模块
        Pos<=0;                    --Pos=0 表示不闪烁,Pos=1、2、3 表示闪烁的位置
      WHEN Adj_Timer=>
        IF Command<="00000" then   --将当前位置初始化为时位
          Pos<=1;
        ELSE
          IF AdjPosition='1' THEN  --调整当前位置(时、分或秒)
            IF Pos=3 THEN
              Pos<=1;
            ELSE
              Pos<=Pos+1;
            END IF;
          END IF;
        END IF;
        Command<="00001";          --启动调时模块
      WHEN Adj_MorningTime=>
        IF Command="00001" THEN
          Pos<=1;
        ELSE
```

```
      IF AdjPosition='1' THEN          --调整当前位置(只调时和分)
        IF Pos=1 THEN
          Pos<=2;
        ELSE
          Pos<=1;
        END IF;
      END IF;
    END IF;
    Command<="00010";                  --启动调整起床时间的模块
  WHEN Adj_EveningTime=>
    IF Command="00010" THEN
      Pos<=1;
    ELSE
      IF AdjPosition='1' THEN
        IF Pos=1 THEN
          Pos<=2;
        ELSE
          Pos<=1;
        END IF;
      END IF;
    END IF;
    Command<="00100";                  --启动调整熄灯时间的模块
  WHEN Adj_RingLength=>
    IF Command="00100" THEN
      Pos<=1;
    ELSE
      IF AdjPosition='1' THEN
        IF Pos=1 THEN                  --Pos=1 时调整打铃时间
          Pos<=3;                      --Pos=3 时调整间歇时间
        ELSE
          Pos<=1;
        END IF;
      END IF;
    END IF;
    Command<="01000";                  --启动调整打铃长度的模块
  WHEN Adj_12or24=>
    Pos<=2;
```

```
        Command<="10000";                    --启动调整时制的模块
    WHEN OTHERS=>
        Pos<=0;
        Command<="00000";
END CASE;

CASE Pos IS
    WHEN 0=>
        Flash<="000";
        AdjBotton<="000";
    WHEN 1=>
        Flash<="100";                        --指示时位闪烁
        IF AdjVal='1' THEN                   --调整时位数值
        AdjBotton<="100";
    ELSE
        AdjBotton<="000";
    END IF;
WHEN 2=>
    Flash<="010";                            --指示分位闪烁
    IF AdjVal='1' THEN                       --调整分位数值
        AdjBotton<="010";
    ELSE
        AdjBotton<="000";
    END IF;
WHEN 3=>
    Flash<="001";                            --指示秒位闪烁
    IF AdjVal='1' THEN                       --调整秒位数值
        AdjBotton<="001";
    ELSE
        AdjBotton<="000";
    END IF;
WHEN OTHERS=>
    Flash<="000";
    AdjBotton<="000";
END CASE;

END IF;
```

```
    END PROCESS;
END;
```

11.2.3　计时/调时模块的设计

计时/调时模块在本系统中起着十分重要的作用，但其设计却相对简单，这里只须根据状态机给出的几个信号进行相应的操作即可(见例程 11－2)。

例程 11－2 计时/调时模块程序

```
LIBRARY IEEE;
USE IEEE.STd_Logic_1164.ALL;

ENTITY DigitalClock IS
PORT
(Clock:IN Std_Logic;                                    --接 10 Hz 时钟
 En:IN Std_Logic;                                       --En=1 表调时,En=0 表正常计时
 AdjustH,AdjustM,AdjustS:IN Std_Logic;                  --AdjustH 为调时信号
                                                        --AdjustM 为调分信号
                                                        --AdjustS 为调秒信号
 AdjustPm:IN Std_Logic;                                 --调上、下午
 Second,Minute:BUFFER Integer RANGE 0 TO 59;            --当前秒、分
 Hour:BUFFER Integer RANGE 0 TO 23                      --当前时
);
END;

ARCHITECTURE Behavior OF DIGITALCLOCK IS
SIGNAL Timecount:Integer RANGE 0 TO 9;
BEGIN
  PROCESS(Clock)
  BEGIN
    IF rising_edge(Clock) THEN
      IF En='1' THEN
        IF AdjustPm='1' THEN                            --切换上、下午
          IF Hour<12 THEN
            Hour<=Hour+12;
          ELSE
            Hour<=Hour-12;
```

```
    END IF;
  ELSIF AdjustH='1' THEN          --调时
    IF Hour=23 THEN
      Hour<=0;
    ELSE
      Hour<=Hour+1;
    END IF;
  ELSIF AdjustM='1' THEN          --调分
    IF Minute=59 THEN
      Minute<=0;
    ELSE
      Minute<=Minute+1;
    END IF;
  ELSIF AdjustS='1' THEN          --调秒(即清 0)
    Second<=0;
  END IF;
ELSE                              --正常计时
  IF (Timecount=9) THEN           --之所以用 10 Hz 时钟,是为了让此模块反应速度变快
    Timecount<=0;                 --缺点时,此处需多加一个计数器,消耗一定资源
    IF (Second=59) THEN
      Second<=0;
      IF (Minute=59) THEN
        Minute<=0;
        IF (Hour=23) THEN
          Hour<=0;
        ELSE
          Hour<=Hour+1;
        END IF;
      ELSE
        Minute<=Minute+1;
      END IF;
    ELSE
      Second<=Second+1;
    END IF;
  ELSE
    Timecount<=Timecount+1;
  END IF;
```

```
        END IF;
      END IF;
    END PROCESS;
END;
```

11.2.4　打铃时间设定模块的设计

打铃时间设定模块的设计思想与调时模块的基本相同(见例程 11－3)。

例程 11－3　打铃时间设定模块程序

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

ENTITY SetRingTime IS
PORT
(Clock:IN Std_Logic;
 En1,En2:IN Std_Logic;     --En1=1 时调整起床时间,En2=1 时调整熄灯时间
 AdjustH:IN Std_logic;     --调时信号
 AdjustM:IN Std_Logic;     --调分信号
 GetUpHour:BUFFER Integer RANGE 0 TO 11;          --起床时间(时)
 GetUpMinute:BUFFER Integer RANGE 0 TO 59;        --起床时间(分)
 SleepHour:BUFFER Integer RANGE 12 TO 23;         --熄灯时间(时)
 SleepMinute:BUFFER Integer RANGE 0 TO 59         --熄灯时间(分)
);
END;

ARCHITECTURE Behavior OF SetRingTime IS

BEGIN
  PROCESS(Clock)
  VARIABLE FirstTime:Integer RANGE 0 TO 1;        --用以判断是否需要初始化
  BEGIN
    IF rising_edge(Clock) THEN
      IF FirstTime=0 THEN                         --初始化
        GetUpHour<=6;                             --初始起床时间为 6:00
        GetUpMinute<=0;
        SleepHour<=22;                            --初始熄灯时间为 22:00
```

```
    SleepMinute<=30;
    FirstTime:=1;                                    --表示初始化完毕
  ELSE
    IF En1='1' THEN                                  --处于调整起床时间模式下
      IF AdjustH='1' THEN                            --调整起床时间(时)
        IF GetUpHour=11 THEN
          GetUpHour<=0;
        ELSE
          GetUpHour<=GetUpHour+1;
        END IF;
      ELSIF AdjustM='1' THEN                         --调整起床时间(分)
        IF GetUpMinute=59 THEN
          GetUpMinute<=0;
        ELSE
          GetUpMinute<=GetUpMinute+1;
        END IF;
      END IF;
    END IF;
    IF En2='1' THEN                                  --调整熄灯时间
      IF AdjustH='1' THEN
        IF SleepHour=23 THEN
          SleepHour<=12;
        ELSE
          SleepHour<=SleepHour+1;
        END IF;
      ELSIF AdjustM='1' THEN
        IF SleepMinute=59 THEN
          SleepMinute<=0;
        ELSE
          SleepMinute<=SleepMinute+1;
        END IF;
      END IF;
    END IF;
  END IF;
 END PROCESS;
END;
```

11.2.5　打铃长度设定模块的设计

打铃长度设定模块可以调整打铃时间长度和间隔时间长度，这两个时间分别在时位和秒位上显示，默认为 5 s 和 2 s(见例程 11－4)。

例程 11－4　打铃长度设定模块程序

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

ENTITY SetRingLength IS
PORT
(Clock:IN Std_Logic;
 En:IN Std_Logic;                          --本模块启动信号
 AdjustH:IN Std_Logic;                     --调整打铃时间长度的信号
 AdjustS:IN Std_Logic;                     --调整间隔时间长度的信号
 RingTime:BUFFER Integer RANGE 1 TO 20;    --打铃时间长度
 PauseTime:BUFFER Integer RANGE 1 TO 10    --间隔时间长度
);
END;

ARCHITECTURE Behavior OF SetRingLength IS
BEGIN
  PROCESS(Clock)
  VARIABLE FirstTime:Integer RANGE 0 TO 1;
  BEGIN
    IF rising_edge(Clock) THEN
      IF FirstTime=0 THEN
        RingTime<=5;
        PauseTime<=2;
        FirstTime:=1;
      ELSE
        IF En='1' THEN
          IF AdjustH='1' THEN                --调整打铃时间长度
            IF RingTime=20 THEN
              RingTime<=1;
            ELSE
```

```
            RingTime<=RingTime+1;
          END IF;
        ELSIF AdjustS='1' THEN              --调整间隔时间长度
          IF PauseTime=10 THEN
            PauseTime<=1;
          ELSE
            PauseTime<=PauseTime+1;
          END IF;
        END IF;
      END IF;
    END IF;
  END IF;
END PROCESS;
END;
```

11.2.6　显示控制及打铃控制模块的设计

显示控制及打铃控制模块根据前面几个模块输出的信息(如当前时间和打铃时间等)确定当前的显示数字以及打铃信号。此外,12/24 时制的切换及整点报时的使能切换也在此模块中完成。相关程序请参见例程 11-5。

例程 11-5　显示控制及打铃控制模块程序

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

ENTITY DisplayController IS
PORT
(Clock:IN Std_Logic;
 EnChange:IN Std_Logic;                          --时制调整模块的启动信号
 Change:IN Std_Logic;                            --接状态机的 AdjBotton[1]
 EnRing:IN Std_Logic;                            --允许整点报时
 Command:IN Std_Logic_Vector(4 DOWNTO 0);        --状态机的输出,用以指示当前状态
 Hour:IN Integer RANGE 0 TO 23;                  --当前时间的时
 GetUpHour:IN Integer RANGE 0 TO 11;             --起床时间的时
 SleepHour:IN Integer RANGE 12 TO 23;            --熄灯时间的时
 Minute,GetUpMinute,SleepMinute:IN Integer RANGE 0 TO 59;
                                                 --当前时间、起床时间和熄灯时间的分
```

```
  Second:IN Integer RANGE 0 TO 59;                    --当前时间的秒
  RingTime:IN Integer RANGE 1 TO 20;                  --打铃时间长度
  PauseTime:IN Integer RANGE 1 TO 10;                 --间隔时间长度
  NumA,NumB,NumC:OUT Integer RANGE 0 TO 60;           --时、分、秒位上的数字
  Pm:BUFFER Std_Logic;                                --指示上、下午的信号
  Ring:OUT Std_Logic;                                 --表征打铃的信号
  EnAlarm:OUT Std_Logic                               --指示是否整点报时的信号
);
END;

ARCHITECTURE Behavior OF DisplayController IS
SIGNAL Display24:Std_Logic;
SIGNAL EnableAlarm:Std_Logic;
SIGNAL Time1,Time2:Integer RANGE 0 TO 59;
BEGIN
  PROCESS(Clock)
  BEGIN
    IF rising_edge(Clock) THEN                        --调整时制
      IF EnChange='1' THEN
        IF Change='1' THEN
          Display24<=Not Display24;
        END IF;
      END IF;
      IF (EnRing='1' AND Command="00000") THEN        --决定是否整点报时
        EnableAlarm<=NOT EnableAlarm;
      END IF;
    END IF;
  END PROCESS;

  PROCESS(Clock)
  BEGIN
    IF falling_edge(Clock) THEN
      CASE Command IS
        WHEN "00000"|"00001"=>                        --正常计时和调整当前时间的状态下
          IF Display24='0' THEN                       --如果是 12 时制
            IF Hour>=12 THEN
              NumA<=Hour-12;
```

```
      Pm<='1';
    ELSE
      NumA<=Hour;
      Pm<='0';
    END IF;
  ELSE                                          --如果是 24 时制
    NumA<=Hour;
    Pm<='0';
  END IF;
    NumB<=Minute;
    NumC<=Second;
  WHEN "00010"=>                                --在调整起床时间状态下
    Pm<='0';
    NumA<=GetUpHour;
    NumB<=GetUpMinute;
    NumC<=0;
  WHEN "00100"=>                                --在调整熄灯时间状态下
    IF Display24='0' THEN
      NumA<=SleepHour-12;
      Pm<='1';
    ELSE
      NumA<=SleepHour;
      Pm<='0';
    END If;
    NumB<=SleepMinute;
    NumC<=0;
  WHEN "01000"=>                                --在调整打铃时间长度状态下
    Pm<='0';
    NumA<=RingTime;
    NumB<=60; --no display
    NumC<=PauseTime;
  WHEN "10000"=>                                --在调整时制状态下
    Pm<='0';
    NumA<=60;
    IF Display24='0' THEN
      NumB<=12;
    ELSE
```

```
            NumB<=24;
          END IF;
          NumC<=60;
        WHEN OTHERS=>
          NULL;
    END CASE;
  END IF;

END PROCESS;
PROCESS(Clock)                                    --此进程用来控制打铃信号
BEGIN
  IF falling_edge(Clock) THEN
    IF ((Hour=GetUpHour) AND (Minute=GetUpMinute)) OR
      ((Hour=SleepHour) AND (Minute=SleepMinute)) THEN
      IF Second<RingTime THEN
        Ring<='1';
      ELSIF Second<Time1 THEN
        Ring<='0';
      ELSIF Second<Time2 THEN
        Ring<='1';
      ELSE
        Ring<='0';
      END IF;
    ELSIF ((Hour=7 AND Minute=30) OR (Hour=8 AND Minute=20) OR
          (Hour=8 AND Minute=30) OR (Hour=9 AND Minute=20) OR
          (Hour=9 AND Minute=40) OR (Hour=10 AND Minute=30) OR
          (Hour=10 AND Minute=40) OR (Hour=11 AND Minute=30)) THEN
      IF Second<5 THEN
        Ring<='1';
      ELSE
        Ring<='0';
      END IF;
    ELSIF (EnableAlarm='1' AND Minute=0 AND Second<5) THEN
      Ring<='1';
    ELSE
      Ring<='0';
    END IF;
```

```
    END IF;
  END PROCESS;
  Time1<=RingTime+PauseTime;
  Time2<=RingTime*2+PauseTime;
  EnAlarm<=EnableAlarm;
END;
```

11.2.7　其他模块的设计

在进行七段数码管译码前，需要将 **DisplayController** 输出的数分解成单独的十进制数，分位模块的程序如例程 11－6 所示。

例程 11－6　分位模块程序

```
LIBRARY IEEE;
USE IEEE.Std_Logic_1164.ALL;
USE IEEE.Std_Logic_Unsigned.ALL;

ENTITY Fenwei IS
PORT
(Numin:IN Integer RANGE 0 TO 60;
 NumA,NumB:OUT Integer RANGE 0 TO 10
);
END;

ARCHITECTURE Fen OF Fenwei IS
SIGNAL Shi:Integer RANGE 0 TO 15;
BEGIN
  Shi<=10 WHEN Numin=60 ELSE
       5 WHEN Numin>=50 ELSE
       4 WHEN Numin>=40 ELSE
       3 WHEN Numin>=30 ELSE
       2 WHEN Numin>=20 ELSE
       1 WHEN Numin>=10 ELSE
       0;
  NumB<=10 WHEN (Numin=60) ELSE
       Numin-Shi*10;
  NumA<=Shi;
END;
```

七段数码管的译码程序如例程 11－7 所示。值的注意的是，这是一个带闪烁功能的以串行扫描方式工作的七段数码管译码电路，因为 6 个数码管如果以并行方式工作，那么将占用太多资源，甚至可能超过 Flex10K10 的容限。

例程 11－7　七段数码管的译码程序

```
LIBRARY IEEE;
USE IEEE. Std_Logic_1164. ALL;
USE IEEE. Std_Logic_Unsigned. ALL;

ENTITY DisplayScan IS
PORT
(Clock:IN Std_Logic;
 Flash:IN Std_Logic_Vector(2 DOWNTO 0);
 NumA,NumB,NumC,NumD,NumE,NumF:IN Integer RANGE 0 TO 10;
 ScanSignal:OUT Std_Logic_Vector(0 TO 5);
 Display:OUT Std_Logic_Vector(0 TO 6)
);
END;

ARCHITECTURE Scaner OF DisplayScan IS
SIGNAL Timecount:Integer RANGE 0 TO 5;
SIGNAL Timeout:Integer RANGE 0 TO 511;
SIGNAL Number:Integer RANGE 0 TO 10;
BEGIN
  PROCESS(Clock)
  BEGIN
    IF rising_edge(Clock) THEN
      IF Timecount=5 THEN
        Timecount<=0;
      ELSE
        Timecount<=Timecount+1;
      END IF;
      IF Timeout=511 THEN
        Timeout<=0;
      ELSE
        Timeout<=Timeout+1;
      END IF;
```

```
CASE Timecount IS
  WHEN 0=>
    ScanSignal<="100000";
    IF ((Flash(2)='1')AND (Timeout>300)) THEN
      Number<=10;
    ELSE
      Number<=NumA;
    END IF;
  WHEN 1=>
    ScanSignal<="010000";
    IF ((Flash(2)='1') AND (Timeout>300)) THEN
      Number<=10;
    ELSE
      Number<=NumB;
    END IF;
  WHEN 2=>
    ScanSignal<="001000";
    IF ((Flash(1)='1') AND (Timeout>300)) THEN
      Number<=10;
    ELSE
      Number<=NumC;
    END IF;
  WHEN 3=>
    ScanSignal<="000100";
    IF ((Flash(1)='1') AND (Timeout>300)) THEN
      Number<=10;
    ELSE
      Number<=NumD;
    END IF;
  WHEN 4=>
    ScanSignal<="000010";
    IF ((Flash(0)='1') AND (Timeout>300)) THEN
      Number<=10;
    ELSE
      Number<=NumE;
```

```
        END IF;
      WHEN 5=>
        ScanSignal<="000001";
        IF ((Flash(0)='1') AND (Timeout>300)) THEN
          Number<=10;
        ELSE
          Number<=NumF;
        END IF;
      END CASE;
    END IF;
  END PROCESS;
  WITH Number SELECT
    Display<="1111110" WHEN 0,
             "0110000" WHEN 1,
             "1101101" WHEN 2,
             "1111001" WHEN 3,
             "0110011" WHEN 4,
             "1011011" WHEN 5,
             "1011111" WHEN 6,
             "1110000" WHEN 7,
             "1111111" WHEN 8,
             "1111011" WHEN 9,
             "0000000" WHEN OTHERS;
  END;
```

11.2.8　顶层文件元件连接图

顶层文件如图 11-3 所示。其中，Clock 为晶振时钟，Botton1～Botton4 为 4 个按键的接口(图中未连接 INPUT 端)，PM 接表示上、下午的发光二极管，Ring 接表示打铃的发光二极管，EnAlarm 接表示是否整点报时的发光二极管，ScanSignal 和 Display 则用于串行扫描七段数码管。

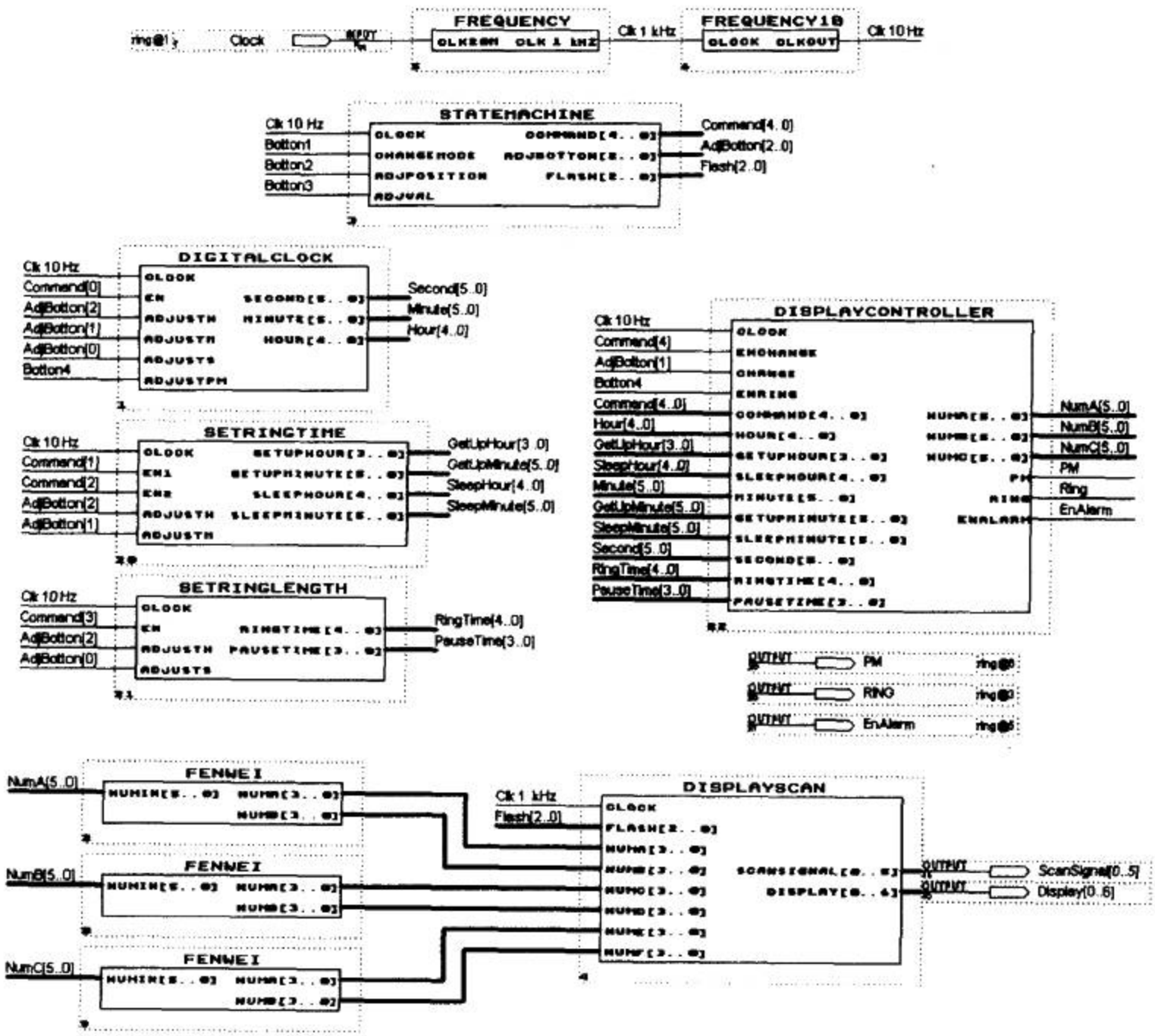


图 11-3　自动打铃系统顶层文件图

11.3　练习题

设计一个通用计数器，具有以下功能：

① 8 位数(二进制，下同)与 8 位数的加、减法；

② 4 位数与 4 位数的乘、除法；

③ 计算器具有 3 个七段数码管作为显示屏，输入数字时，从右往左依次显示(输入 1 个

数，则显示在最右边的数码管上，左边两个数码管不显示；再输入第 2 个数，第 1 个数显示在中间的数码管上，第 2 个数显示在右边的数码管上，最左边的数码管不显示，以此类推）。

④ 计算器采用 4×4 矩阵键盘，0～9 为数字键，A 为加，B 为减，C 为乘，D 为除，E 为等于，F 为 BackSpace 键（功能和 PC 机上的 BackSpace 键同）。

第12章

WCDMA 短码生成器

12.1 任务书

WCDMA 系统中短码用于区别信道，采用正交可变扩频因子码(OVSF，也称树码)作为短码，能保证不同速率的扩频因子的信道间的正交性。该码的树如图 12-1 所示，其中码型(扩频因子)*SF* 等于码长。每一个码都和一个扩频因子(码型)和在此码型中的序号 *SN* 一一对应。设 *SF* 最大为 128，编写产生这种短码的 VHDL 程序，并进行仿真。

要求如下：

① 输入：时钟、异步复位信号、扩频因子 *SF*、序号 *SN*、编码器启动信号；

② 输出：编码序列、同步信号(对编码位置起始和终止位置对齐的正电平)；

③ 特殊要求：要求在接收到编码器启动信号后一个时钟周期内即开始输出生成的短码，不可使用查找表方式生成，程序需使用在 Max+PlusII 中可综合的 VHDL 语句。

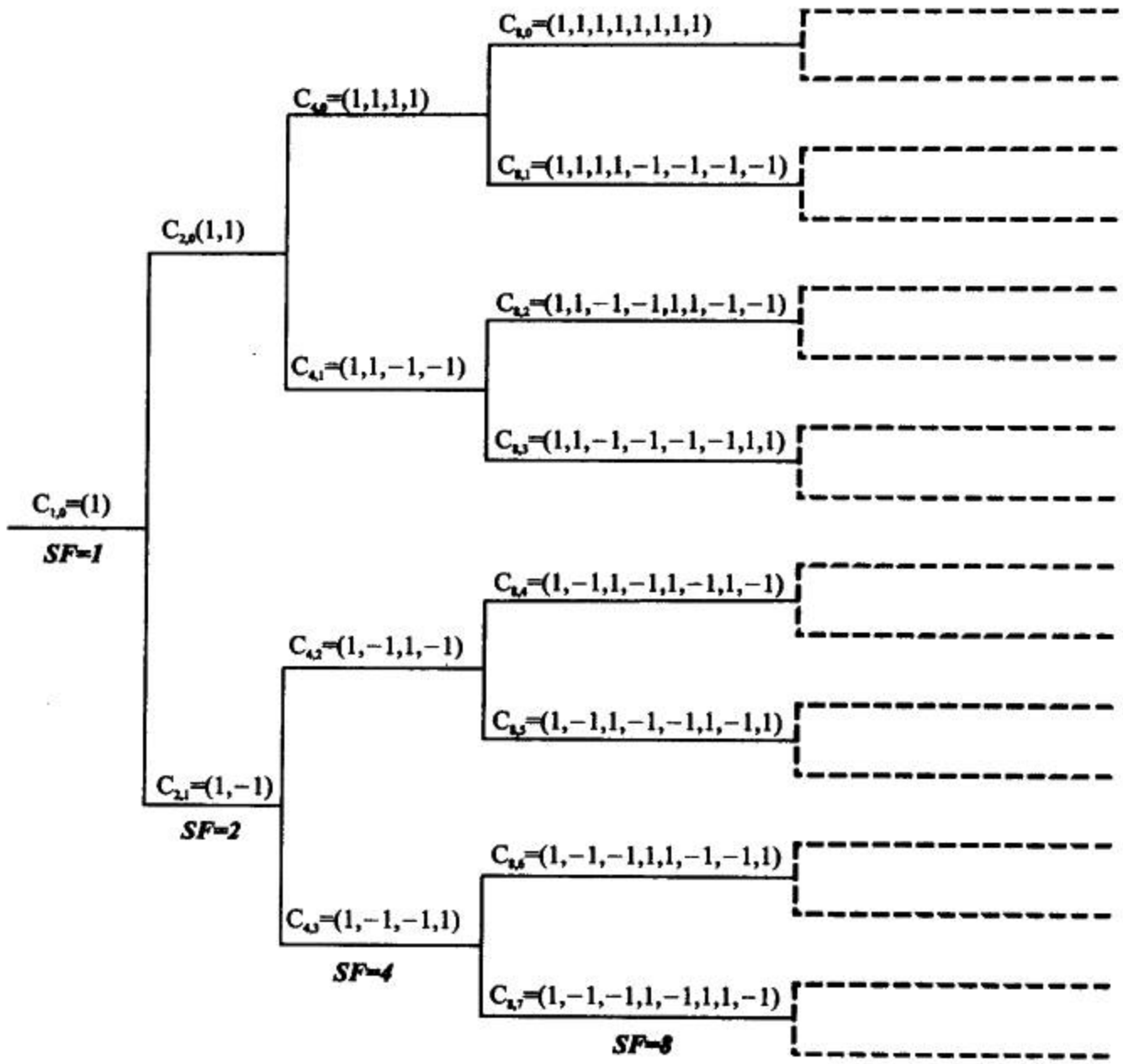


图 12-1　WCDMA 短码码树

12.2　参考设计

本题既要求编码器快速反应，又要求不使用查找表，这使编程难度大大增加。这里可以采用边生成码字边输出的办法来解决这一问题。

为了方便下面的讲解，那么先定义一下层与级的概念：

① 第 1 级码字即第 1 层码字，为 1；

② 第 $N(N>1)$层的码字，指 $SF=N$ 的码字，码字长度为 2^{N-1}；

③ 第 $N(N>1)$级的码字，指第 N 层码字的后半部分(即第 N 层码字与第 $N-1$ 层码字相异的部分)，码字长度为 2^{N-2}。

举个例子，第 4 层上的第 3 个码字为 $C_{8,2}=(1,1,-1,-1,1,1,-1,-1)$，它的码字构成有 4 级：第 1 级为 1，第 2 级为 1，第 3 级为(−1，−1)，第 4 级为(1，1，−1，−1)。

可以看出，各级码字级连后的结果就是该层上一个完整的码字。

从图 12－1 的码树可以看出，序号 *SN* 为偶数(包括 0)时，本级码字为前一层码字的重复；*SN* 为奇数时，本级码字则为前一层码字取反的结果。

因为本级码字由前面所有生成的码字所决定，所以只能依次生成各级码字，要一次输出所有码字是很困难的。

这里可以用状态机来实现码字的生成与输出。设状态有 Stage1～Stage8 共 8 个状态，Stage*N* 对应码树的第 *N* 层，完成此层码字的输出(事实上只须输出第 *N* 级码字)。比如，需要输出 $C_{8,2}=(1,1,-1,-1,1,1,-1,-1)$，在 Stage1 输出 1 同时生成第 2 级码字(1)，在 Stage2 输出第 2 级码字同时生成第 3 级码字(−1，−1)，在 Stage3 输出第 3 级码字同时生成第 4 级码字(1，1，−1，−1)，在 Stage4 则输出第 4 级码字。

分级生成和输出码字可以使编码器具有很高的反应速度(第 1 层码字无需计算，即是 1，因此可在收到启动信号后立即开始输出)，同时简化了编程难度(输出第 *N* 级码字的同时生成第 *N*＋1 级码字，将同级码字的计算和输出分在两个状态中进行，避免了可能遇到的同步等问题)。

例程 12－1 为 WCDMA 短码生成器的参考程序。

例程 12－1　WCDMA 短码生成器的参考程序

```
LIBRARY IEEE;
USE IEEE. Std_Logic_1164. ALL;
USE IEEE. Std_Logic_Unsigned. ALL;
USE IEEE. Std_Logic_Arith. ALL;

ENTITY ShortCoder IS
PORT
(Clock:IN Std_Logic;
 Reset:IN Std_Logic;
 Synin:IN Std_Logic;                    --编码器启动信号
 SF:IN Integer RANGE 1 TO 128;          --扩频因子(同时也是码长)
 SN:IN Integer RANGE 0 TO 127;          --序号，此处为了使程序直观、易懂，
                                        --用 Integer 类型，下面的 SN_Route
                                        --为 SN 对应的 Std_Logic_Vector 类型
```

```
 SynOut:OUT Std_Logic;                                    --输出码字的同步信号
 CodeOut:OUT Std_Logic                                    --编码输出
);
END;

ARCHITECTURE OVSF OF ShortCoder IS
TYPE State_type IS
(Stage1,Stage2,Stage3,Stage4,Stage5,Stage6,Stage7,Stage8,Finished);
    --状态机共有 9 个状态,Stage1～Stage8 分别代表码树的 8 个层

SIGNAL State:State_type;
SIGNAL CodeStage1:Std_Logic;                              --第 1 级码字(即 1)
SIGNAL CodeStage2:Std_Logic;                              --CodeStage2～CodeStage8 为
                                                          --第 2～8 级码字
SIGNAL CodeStage3:Std_Logic_Vector(1 DOWNTO 0);
SIGNAL CodeStage4:Std_Logic_Vector(3 DOWNTO 0);
SIGNAL CodeStage5:Std_Logic_Vector(7 DOWNTO 0);
SIGNAL CodeStage6:Std_Logic_Vector(15 DOWNTO 0);
SIGNAL CodeStage7:Std_Logic_Vector(31 DOWNTO 0);
SIGNAL CodeStage8:Std_Logic_Vector(63 DOWNTO 0);
SIGNAL Length:Integer RANGE 0 TO 7;                       --路径长度(亦即码树的深度)
SIGNAL Route:Std_Logic_Vector(6 DOWNTO 0);                --记录生成码字的路径信息
SIGNAL Clear:Std_Logic;                                   --和 Start 信号一起用来处理编码器启动信号
SIGNAL Start:Std_Logic;
SIGNAL SN_Route:Std_Logic_Vector(6 DOWNTO 0);             --与 SN 对应,见 SN 的注释

BEGIN
  SN_Route<=CONV_STD_LOGIC_VECTOR(SN,7);                  --将 SN 转为 Std_Logic_Vector 类型
  PROCESS(Clear,Synin)
  BEGIN
    IF Clear='1' THEN                                     --Clear 用来清除编码器的开始信号
      Start<='0';                                         --Clear 本身由另一个 Process 控制
    ELSIF rising_edge(Synin) THEN
      Start<='1';                                         --Synin 上升沿触发编码器的开始信号
      CASE SF IS                                          --计算路径长度及生成路径信息
        WHEN 2=>                                          --扩频因子(码长)为 2 时
          Length<=1;                                      --路径长度为 1,即在第 1 级码后只须经过 1
                                                          --步就可生成所需码字
```

```
    Route<=SN_Route(0)&"000000";          --有效路径信息只有 1 位,其余为 0
  WHEN 4=>                                --扩频因子(码长)为 4 时
    Length<=2;                            --路径长度为 2,即在第 2 级码后须经 2 步才
                                          --能生成所需码字
    Route<=SN_Route(1 Downto 0)&"00000";  --有效路径信息为 2 位
  WHEN 8=>
    Length<=3;
    Route<=SN_Route(2 Downto 0)&"0000";
  WHEN 16=>
    Length<=4;
    Route<=SN_Route(3 Downto 0)&"000";
  WHEN 32=>
    Length<=5;
    Route<=SN_Route(4 Downto 0)&"00";
  WHEN 64=>
    Length<=6;
    Route<=SN_Route(5 Downto 0)&'0';
  WHEN 128=>
    Length<=7;
    Route<=SN_Route;
  WHEN OTHERS=>
    NULL;
  END CASE;
 END IF;
END PROCESS;

PROCESS(Reset,Clock)
VARIABLE i3:Integer RANGE 0 TO 1;
VARIABLE i4:Integer RANGE 0 TO 3;
VARIABLE i5:Integer RANGE 0 TO 7;
VARIABLE i6:Integer RANGE 0 TO 15;
VARIABLE i7:Integer RANGE 0 TO 31;
VARIABLE i8:Integer RANGE 0 TO 63;
BEGIN
 IF Reset='1' THEN
   State<=Stage1;
   SynOut<='0';
```

```
    Clear<=1';
  ELSIF falling_edge(Clock) THEN      --时钟下降沿工作
    CASE State IS
      WHEN Stage1=>
        Clear<=0';
        IF Start=1' THEN              --接收到开始编码的命令
          CodeStage1<=1';             --1 级码字为1'
          CodeOut<=1';                --直接输出第 1 级码字
          IF Route(6)=0' THEN         --根据路径信息生成第 2 级码字
            CodeStage2<=1';           --0 表示本级码字与前面所有码字级联后的码字相同
        ELSE
            CodeStage2<=0';           --1 表示本级码字为前面所有码字级联后取反
        END IF;
        State<=Stage2;
        SynOut<=1';                   --输出同步信号
      END IF;
    WHEN Stage2=>
      Clear<=1';                      --Clear 信号为 1,表示将 Start 信号清 0,
                                      --同时屏蔽 Synin 信号(编码完成前不再接收新的启动信号)
      CodeOut<=CodeStage2;            --输出第 2 级码字
      IF Length=1 THEN                --所有码字均已输完
        State<=Finished;              --转到 Finished 状态
      ELSE
        IF Route(5)=0' THEN           --生成第 3 级码字
          CodeStage3<=CodeStage1&CodeStage2;
        ELSE
          CodeStage3<=NOT(CodeStage1&CodeStage2);
        END IF;
        State<=Stage3;
        i3:=1;
      END IF;
    WHEN Stage3=>
      IF i3=1 THEN
        CodeOut<=CodeStage3(1);
        i3:=0;
      ELSE
        CodeOut<=CodeStage3(0);
```

```
        IF Length=2 THEN
          State<=Finished;
        ELSE

       IF Route(4)='0' THEN
          CodeStage4<=CodeStage1&CodeStage2&CodeStage3;
       ELSE
          CodeStage4<=Not(CodeStage1&CodeStage2&CodeStage3);
        END IF;
        i4:=3;
        State<=Stage4;
      END IF;
    END IF;
  WHEN Stage4=>                          --在码树的第 4 层上
    IF i4>0 THEN                         --输出第 4 级码字的前 3 位
      CodeOut<=CodeStage4(i4);
      i4:=i4-1;
    ELSE
      CodeOut<=CodeStage4(0);            --输出第 4 级码字的最低位
      IF Length=3 THEN                   --同时判断码字是否已经输完并生成第 5 级码字
                                         --下面的程序大同小异,不再加注释
        State<=Finished;
      ELSE
        IF Route(3)='0' THEN
          CodeStage5<=CodeStage1&CodeStage2
                      &CodeStage3&CodeStage4;
        ELSE
          CodeStage5<=NOT (CodeStage1&CodeStage2
                          &CodeStage3&CodeStage4);
        END IF;
        i5:=7;
        State<=Stage5;
      END IF;
    END IF;
  WHEN Stage5=>
    IF i5>0 THEN
      CodeOut<=CodeStage5(i5);
```

```
    i5:=i5-1;
  ELSE
    CodeOut<=CodeStage5(0);
    IF Length=4 THEN
      State<=Finished;
    ELSE
      IF Route(2)='0' THEN
        CodeStage6<=CodeStage1&CodeStage2&CodeStage3
                    &CodeStage4&CodeStage5;
      ELSE
        CodeStage6<=NOT (CodeStage1&CodeStage2&CodeStage3
                         &CodeStage4&CodeStage5);
      END IF;
      i6:=15;
      State<=Stage6;
    END IF;
  END IF;
WHEN Stage6=>
  IF i6>0 THEN
    CodeOut<=CodeStage6(i6);
    i6:=i6-1;
  ELSE
    CodeOut<=CodeStage6(0);
    IF Length=5 THEN
      State<=Finished;
    ELSE
      IF Route(1)='0' THEN
        CodeStage7<=CodeStage1&CodeStage2&CodeStage3
                    &CodeStage4&CodeStage5&CodeStage6;
      ELSE
        CodeStage7<=NOT (CodeStage1&CodeStage2&CodeStage3
                         &CodeStage4&CodeStage5&CodeStage6);
        END IF;
        i7:=31;
        State<=Stage7;
      END IF;
    END IF;
```

```
WHEN Stage7=>
  IF i7>0 THEN
    CodeOut<=CodeStage7(i7);
    i7:=i7-1;
    ELSE
      CodeOut<=CodeStage7(0);
      IF Length=6 THEN
        State<=Finished;
      ELSE
        IF Route(0)='0' THEN
          CodeStage8<=CodeStage1&CodeStage2&CodeStage3
                      &CodeStage4&CodeStage5&CodeStage6&CodeStage7;
        ELSE
          CodeStage8<=Not(CodeStage1&CodeStage2&CodeStage3
                      &CodeStage4&CodeStage5&CodeStage6&CodeStage7);
        END IF;
        i8:=63;
        State<=Stage8;
      END IF;
    END IF;
  WHEN Stage8=>
    IF i8>0 THEN
      CodeOut<=CodeStage8(i8);
      i8:=i8-1;
    ELSE
      CodeOut<=CodeStage8(0);
      State<=Finished;
    END IF;
  WHEN Finished=>
    SynOut<='0';                    --将输出同步信号清 0
    State<=Stage1;                  --回到最初状态
  WHEN OTHERS=>
    NULL;
  END CASE;
 END IF;
END PROCESS;
END;
```

图 12－2 为 $SF=16$、$SN=7$ 时的仿真波形(图中的 0 表示－1,下同),得到的码字为 1100001100111100,从图 12－1 中的码树图很容易推断出,此码字确实为 $SF=16$、$SN=7$ 时的码字。

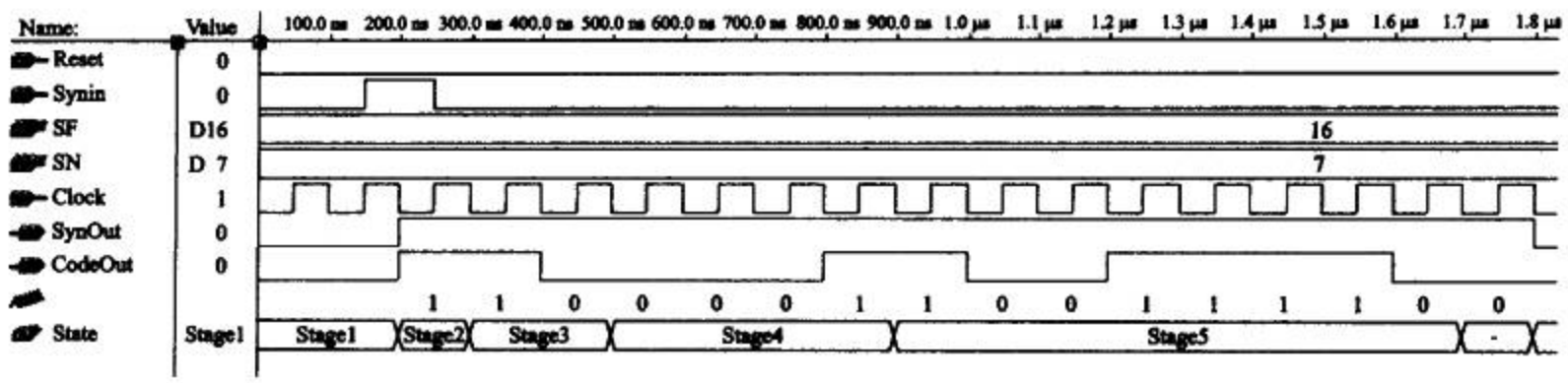


图 12－2　$SF=16$、$SN=7$ 时的仿真波形

图 12－3 为 $SF=8$、$SN=5$ 时的仿真波形,得到的码字为 10100101,这与图 12－1 的码树中的 $C_{8,5}$ 是一致的。

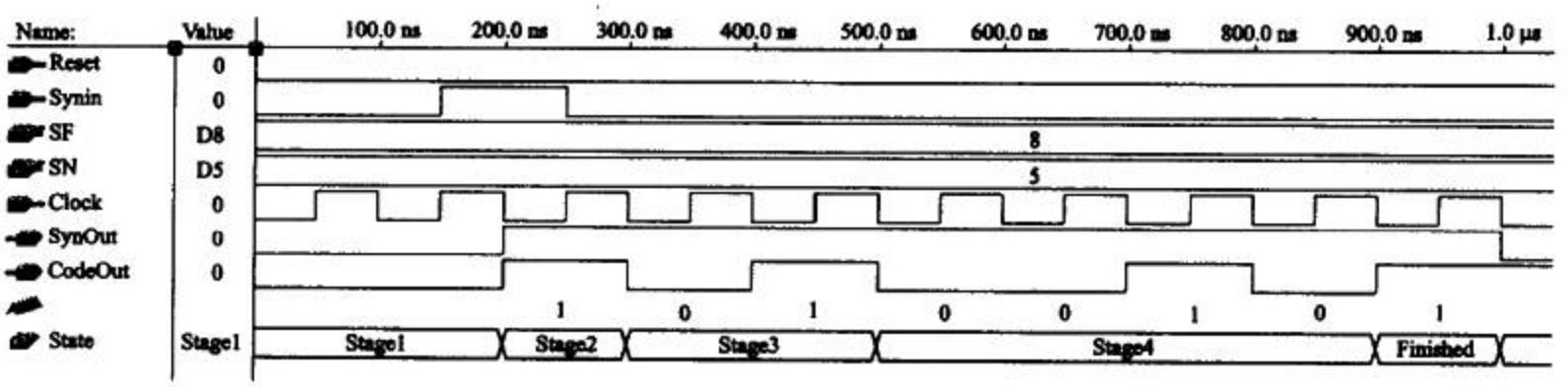


图 12－3　$SF=8$、$SN=5$ 时的仿真波形

图 12－4 为简单的容错性测试的仿真波形。图中 Synin(输入同步信号)出现了两次,而 SF 也改变了两次(从 8 到 16,再从 16 到 8)。这些改变都是在码字输出尚未完成的情况下发生的,但码字输出没有受到影响。这说明本程序能够避免码字输出过程中因同步信号的重复出现、码型的改变等引起的一些错误。

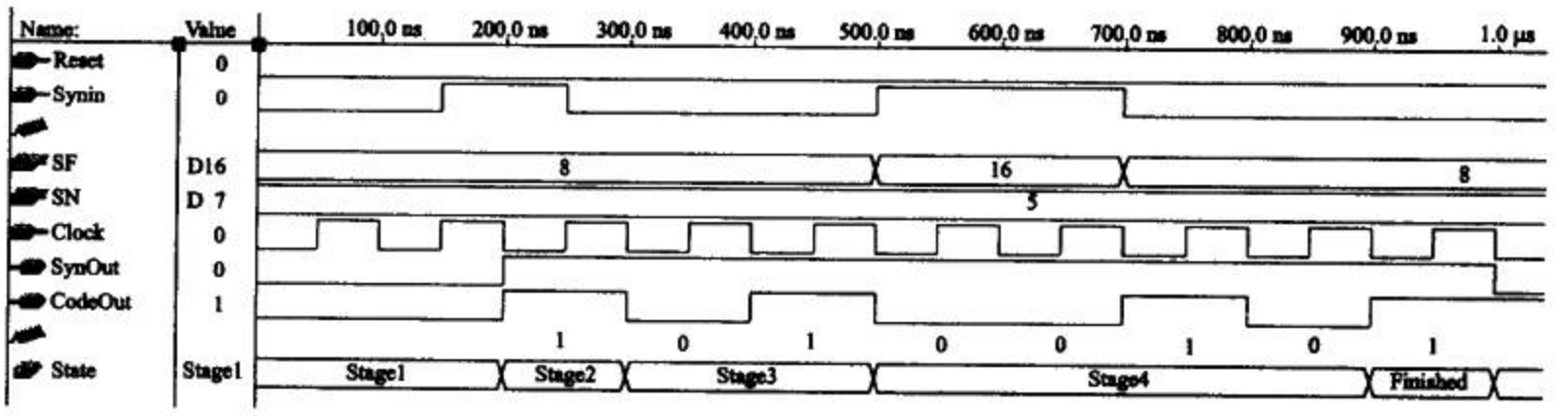


图 12－4　简单容错性测试的仿真波形

12.3　练习题

码率 2/3 $(d,k)=(1,7)$码，该码利用前看(look - ahead)编码技术。如果一个码当前码字的编码与译码可能依赖将来的输入符号，那么该码称为前看(look - ahead)编码。

该码有两个码表：基本码表和替代码表，如表 12 - 1 所列。其中，基本码表为 4 个基本信源符号的编码。在对当前信源符号编码时，要提前看下一个信源字，如果属于替代码表中的数据组合，就按替代码表编码。可以看到，基本码表中的码字本身满足 $d=1$ 的约束，但这些码字相互连接时，d 可能受到破坏(以 1 结尾的码字与以 1 开头的码字相连)，所以需要一个 000 替代码字，防止连续 1 的出现。这样，序列的 0 游程长度最大为 7。

(1,7)码基本码表和替代码表如表 12 - 1 所列。

表 12 - 1　(1,7)码基本码表和替代码表

基本码表(1,7)码		替代码表(1,7)码	
数　据	码　字	数　据	码　字
00	101	00.00	101.000
01	100	00.01	100.000
10	001	10.00	001.000
11	010	10.01	010.000

实际上，该码在编码时还依赖前一个数据或码字。例如，如果当前数据是 00，那么还要看前一个码字是 000 还是基本码表中的码字。如果是 000 就按基本码表编码，否则要看是否为 101 或 001，是则编为 000，否则编为 101。

当对当前码字进行译码时，也要根据前一个码字和下一个码字。例如，当前为 000，只有依据前一个码字才能判定译码输出，而当接收 100 时，只有依据下一个码字才能判定译码输出是 01 还是 00，所以该码的译码窗长为 3 个码字。根据编码表可以导出译码器输出的布尔表达式，从而证明一个比特的接收差错可影响 5 个连续数据比特错误。

要求：给定如表 12 - 1 所列码率 2/3 的$(d,k)=(1,7)$码，设 a、b 分别表示 2 比特的信源字，x、y、z 表示 3 比特的码字，下标 -1，0，1 表示前一个、当前和下一个比特。

① 证明编码函数为：

$$x_0 = \bar{a}_0\bar{b}_0\bar{z}_{-1} + \bar{a}_0 b_0(b_{-1} + \bar{x}_{-1}\bar{y}_{-1})$$

$$y_0 = a_0(b_0 + \bar{a}_1 b_1)$$

$$z_0 = \bar{b}_0(a_1 + \bar{b}_1)(a_0 + \bar{z}_{-1})$$

② 证明译码函数为：

$$a_0 = \bar{x}_0(y_0 + z_0)$$

$$b_0 = \bar{x}_0\bar{y}_0\bar{z}_0\bar{z}_{-1} + (x_0 + y_0)(x_1 + y_1 + z_1)\bar{z}_0$$

③ 用 VHDL 语言设计编译码器，设输入序列为：

00 11 01 00 00 00 00 10 01 11 01 01 00 10 00 01 10

第13章

Franaszek 编/译码器

13.1 任务书

在使用峰值检测技术的磁盘驱动中的一种标准编码是码率为 1/2 的 $(d, k) = (2, 7)$ 码，也称 Franaszek 码。该码的编码表如表 13-1 所列。

要求：

① 设计编码器，要求信息序列能连续输入，编码序列连续输出；

编码器输入序列为 1 1 0 1 0 0 1 1 0 0 1 0 1 1 0 0 0 0 0 1 1 1 0 1 0 1 0 0 0 1 1 0 1 0 0 0 0 0 1 0 0 0 1 1 0 1 0 0 1 1 1 1 1 0 0 0 1 0 1 0 1 0 1 1;

② 设计译码器，要求信息序列能连续输入，编码序列连续输出，译码器输入使用编码器的输出序列；

③ 将码字存在 ROM 中；

④ 输入有一个指示数据起始的同步脉冲；

⑤ 系统使用的最高时钟频率只能是输入信息序列时钟的 2 倍；

⑥ 特殊要求是程序必须使用 Max+plusII 能综合的语句。

表 13-1 Franaszek 码的编码表

信源字	码 字
11	0100
10	1000
000	000100
010	001000
011	100100
0010	00001000
0011	00100100

13.2　参考设计

13.2.1　总体方案设计

本题目属于变长编解码的范畴，从信息论中学到的知识可知，变长编码有很多优点，例如压缩率大，平均码长短，其中最著名的就是经常用到的哈夫曼编码。下面主要讨论编码器的设计。

一般来说，变长码的编码器要比定长码的编码器难，主要有以下两个难点：

(1) 如何识别信源字

信源字并不是固定长度的，不能像定长码一样简单地按码长来判断，而必须边接收信息边识别，如何最简单、有效地检测出信源字是本题首先要解决的一个问题。

(2) 速率匹配问题

假设输入的信源码存在这样的组合：2 位信源字后紧接着 4 位信源字。如果编码器在检测到 2 位信源字后立即输出编码结果，将出现什么结果呢？聪明的读者肯定想到了，当这 2 位信源字的编码输出结束后，4 位信源字尚未接收完毕，自然也就无法进行编码。也就是说，2 个信源字的编码之间出现了空档，造成了输出码字的不连续，这是不可接收的。造成这个现象的原因是：变长编码编成的码字长度是不相等的，这必然导致信源输出速率是变化的，然而在实际信道中传送的信息率是固定不变的，也就是说信源给出的是变速的，而信道传送的则是恒速的，因此信源与信道之间必然存在一个速率匹配问题。

解决第 1 个问题的方法有很多，但最简单、易懂而且容易编程的方法非状态机莫属。图 13－1为识别 000、010、10 和 11 这 4 个信源字的状态转移图。图中 0、1、00、01 这 4 个状态为中间状态，而 000、010、10 和 11 这 4 个状态为信源识别状态，进入信源识别状态即表示识别出了信源字。在信源识别状态下，若读到信息位为 0，则重新进入 0 状态；否则进入 1 状态，开始识别下一个信源字。事实上，这里还应该再加上一个初始状态，即开始编码之前的状态。在此状态下，如果接收到开始编码的信号，程序就根据读到的信息位选择进入 0 状态或 1 状态。完整的状态定义与状态转移将在 13.2.2 小节的程序中得到体现，此处不再赘述。

如果要解决第 2 个问题(即速率匹配问题)，那么通常采用缓冲存储器(Buffer)来实现变速入，恒速出的功能。缓存就像水库一样，就算往水库注水时快时慢，抽水机输出水流的速率仍是一样的(前提是抽水的平均速率不能比注水的平均速率高，否则早晚会把水全抽完)。

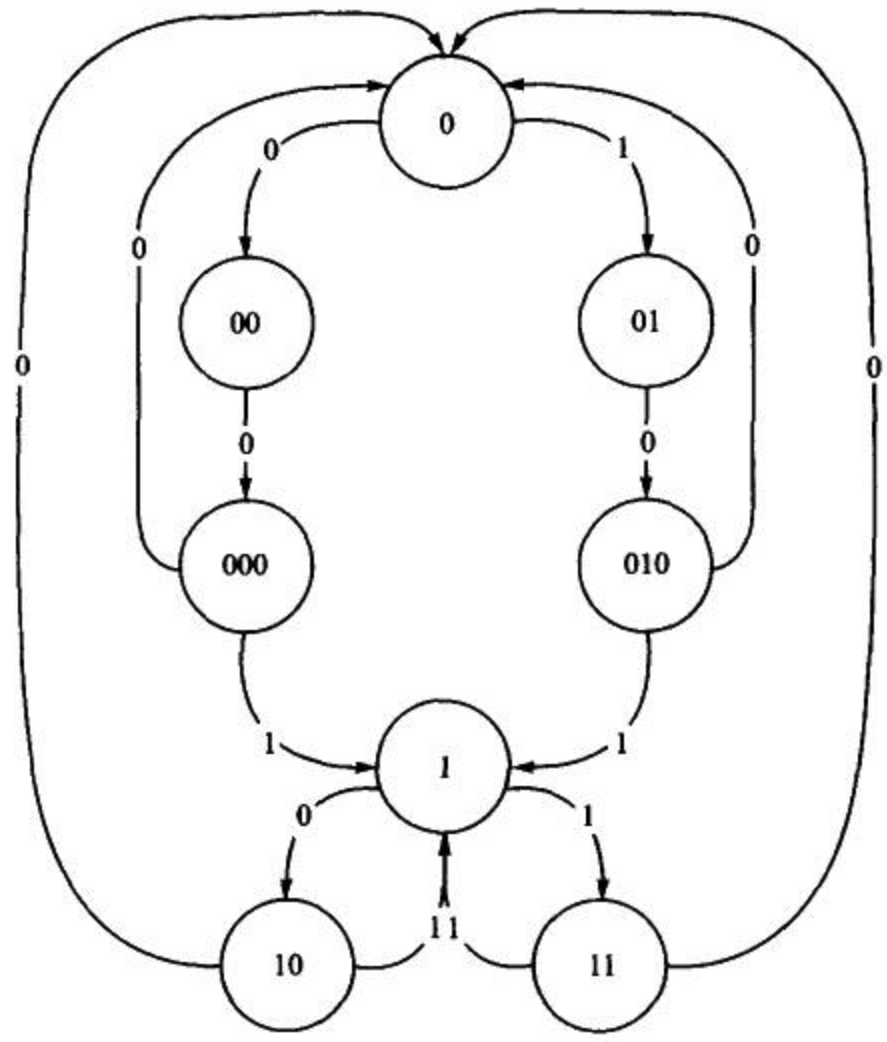


图 13－1　信源字识别过程的状态转移图(部分)

根据上面的分析,初步定下了整体的设计方案(系统框图见图 13－2):

① 整个编码器由状态机、缓存控制模块和 ROM 组成;

② 用状态机来进行信源字的识别,并将其相应的存储地址传给存有编码规则的 ROM;

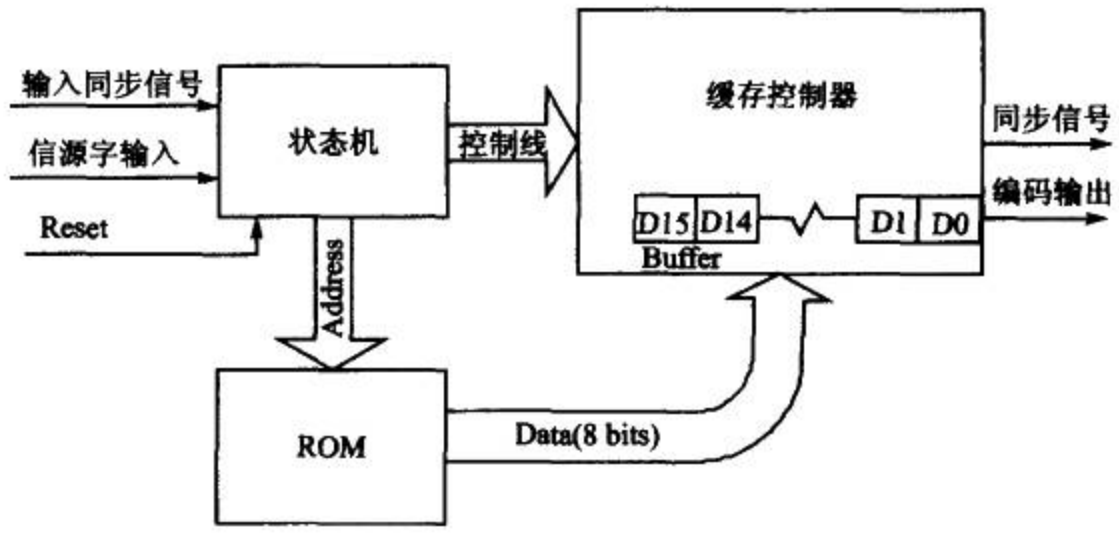


图 13－2　Franaszek 编码器系统框图

③ 缓存控制模块读取存于 ROM 中的编码，置于缓存中，当缓存中的数据达到一定的数量(足以避免出现空档现象)即开始对外串行输出编码；

④ ROM 采用 Max+plusII 中的可调参数宏模块 LPM_ROM，既充分利用片内资源，又减少编程量。

13.2.2　状态机的设计

关于用状态机来实现信源字识别的原理，13.2.1 小节已经讲得很清楚了，此处不再赘述。为了与后级的缓存控制器配合，本模块设立了两个与缓存控制器相连的接口信号：Flag 和 Length。每识别出一个信源字，Flag 就从低电平变为高电平，用以指示缓存控制器从 ROM 中读取相应编码；而 Length 则用来表征编码的长度，即指示缓存控制器应从 ROM 中读取的数值的位数(ROM 的数据线长度是固定的，因此必须借助 Length 信号完成变长编码)。例程 13－1是状态机程序。

例程 13－1　状态机

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

ENTITY Statemachine IS
PORT
(Clkin:IN Std_Logic;                          --时钟
 Reset:IN Std_Logic;                          --复位信号
 Datain:IN Std_Logic;                         --输入的信息序列
 Synin:IN Std_Logic;                          --输入的同步信号(启动编码器信号)
 Flag:OUT Std_Logic;                          --每识别出一个信源字就产生一个脉冲
 Length:OUT Integer RANGE 0 TO 7;             --表征编码的长度
 Address:OUT Std_Logic_Vector(2 DOWNTO 0)  --信源字对应编码在 ROM 中的地址
);
END;

ARCHITECTURE MealyMachine OF Statemachine IS
TYPE State_type IS (S_Init,S_0,S_1,S_00,S_01,S_10,S_11,S_000,
                    S_001,S_010,S_011,S_0010,S_0011);
SIGNAL State:State_type;
SIGNAL Clear:Std_Logic;                       --Clear 和 Start 信号用于捕获同步信号的进程
SIGNAL Start:Std_Logic;
```

```
BEGIN
  PROCESS(Clear,Synin)                    --此进程用以检测输入同步信号
  BEGIN
    IF Clear='1' THEN
      Start<='0';
    ELSIF rising_edge(Synin) THEN
      Start<='1';
    END IF;
  END PROCESS;

  PROCESS(Reset,Clkin)                    --此进程用以控制状态转移
  BEGIN
    IF Reset='1' THEN                     --复位时
      State<=S_Init;                      --状态回复到初始状态
      Clear<='0';
    ELSIF rising_edge(Clkin) THEN
      CASE State IS
        WHEN S_Init=>                     --初始状态
          IF Start='1' THEN               --检测到启动编码器的信号
      Clear<='1';                         --将启动编码器信号清 0
      IF Datain='0' THEN                  --输入的信息位为 0
        State<=S_0;                       --转到 S_0 状态
      ELSE
        State<=S_1;                       --否则转到 S_1 状态
      END IF;
    END IF;
  WHEN S_0=>                              --以下的状态转移很容易理解,原理见 13.2.1 小节,此
                                          --处不再加注释
    IF Datain='0' THEN
      State<=S_00;
    ELSE
      State<=S_01;
    END IF;
  WHEN S_1=>
    IF Datain='0' THEN
      State<=S_10;
    ELSE
```

```
        State<=S_11;
      END IF;
    WHEN S_00=>
      IF Datain='0' THEN
        State<=S_000;
      ELSE
        State<=S_001;
      END IF;
    WHEN S_01=>
      IF Datain='0' THEN
        State<=S_010;
      ELSE
        State<=S_011;
      END IF;
    WHEN S_001=>
      IF Datain='0' THEN
        State<=S_0010;
      ELSE
        State<=S_0011;
      END IF;
    WHEN OTHERS=>
      IF Datain='0' THEN
        State<=S_0;
      ELSE
        State<=S_1;
      END IF;
    END CASE;
  END IF;
END PROCESS;

PROCESS(Reset,State)                    --此进程用以输出与后级缓存控制器相关的控制信号
                                        --和与 ROM 相关的地址信号
BEGIN
  IF Reset='1' THEN
    Flag<='0';
    Address<="000";
    Length<=0;
  ELSE
```

```
CASE State IS
  WHEN S_11=>
    Flag<='1';                    --每次识别出一个信源字,Flag 都置 1
                                  --用以指示缓存控制器读取 ROM 的值
    Address<="001";               --ROM 的地址
    Length<=3;                    --指示缓存控制器,编码长度为 4(0 为
                                  --第 1 个数,所以此处 3 表示长度为 4)
  WHEN S_10=>
    Flag<='1';
    Address<="010";
    Length<=3;
  WHEN S_000=>
    Flag<='1';
    Address<="011";
    Length<=5;
  WHEN S_010=>
    Flag<='1';
    Address<="100";
    Length<=5;
  WHEN S_011=>
    Flag<='1';
    Address<="101";
    Length<=5;
  WHEN S_0010=>
    Flag<='1';
    Address<="110";
    Length<=7;
  WHEN S_0011=>
    Flag<='1';
    Address<="111";
    Length<=7;
  WHEN OTHERS=>
    Flag<='0';
    Address<="000";
    Length<=0;
  END CASE;
 END IF;
END PROCESS;
END;
```

例程 13－1 的仿真图如图 13－3 所示。从图中可以清楚地看到识别信源字 10 和 011 的状态转移过程。值得注意的是，在 Flag 从低电平变为高电平的一刻，Length 和 Address 也同时改变；Flag 变为低电平后，Length 和 Address 也同时归 0；换句话说，当 Flag 为高电平时，Length 和 Address 为有效值。后级缓存控制器的编程中利用了这一特点，请在阅读 13.2.4 小节的时候注意。

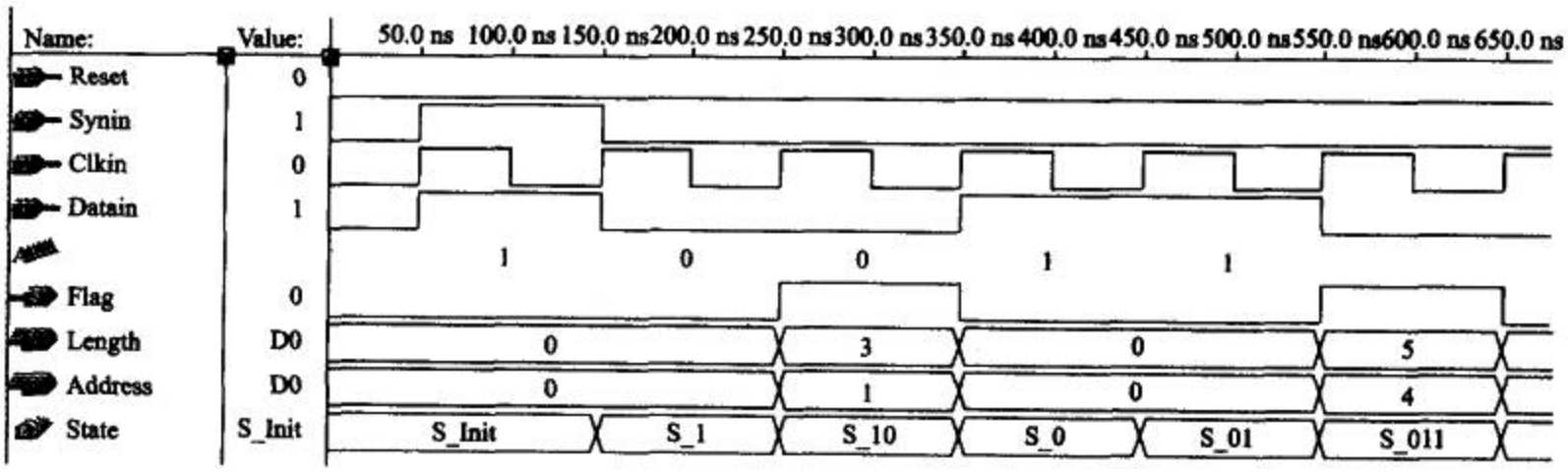


图 13－3　状态机的仿真图

13.2.3　LPM_ROM 的配置

图 13－4 为调用 LPM_ROM 时的配置图，这里将 ROM 配置成异步方式，输出的数据根据地址的变化而变化。例程 13－2 为 ROM 的初始化文件 Code.mif 中的内容，请将此文件与编码表(表 13－1)进行对比，找出存储的规律。

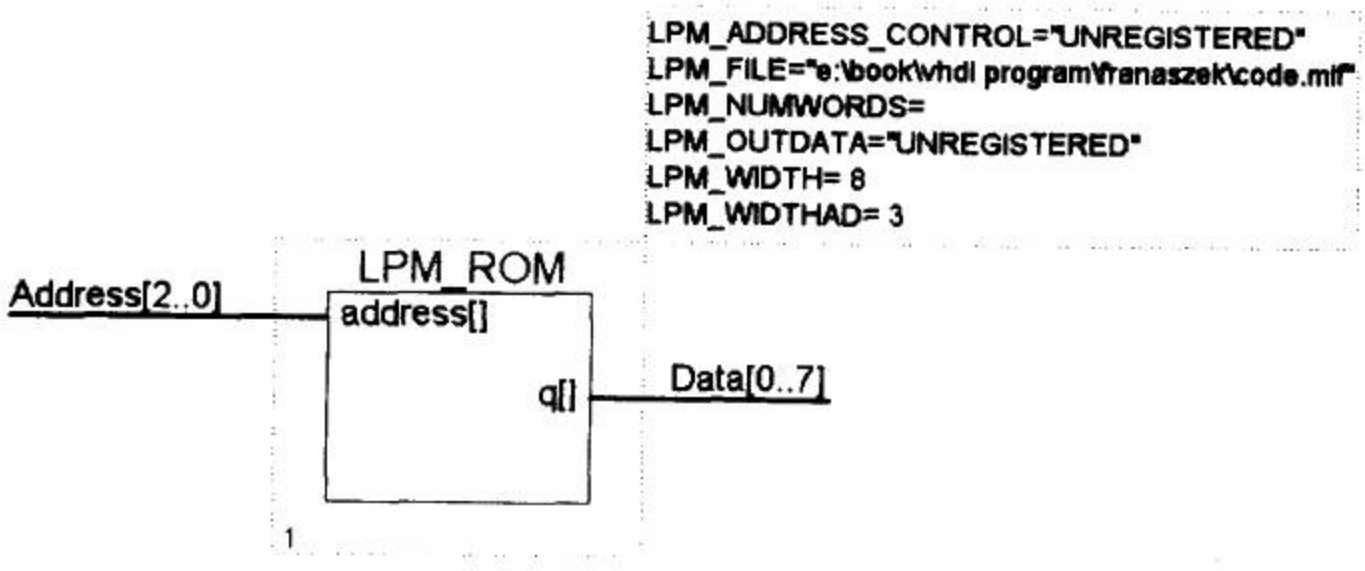


图 13－4　LPM_ROM 的配置图

例程 13－2　ROM 的初始化文件 Code. mif 中的内容

```
--CODE. mif

WIDTH = 8;
DEPTH = 8;

ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;

CONTENT BEGIN
    0    :    00000000;
    1    :    01000000;
    2    :    10000000;
    3    :    00010000;
    4    :    00100000;
    5    :    10010000;
    6    :    00001000;
    7    :    00100100;
END;
```

13.2.4　缓存控制器的设计

缓存的设计是本题的难点之一，主要是因为：

① 缓存控制器不但要将 ROM 中的数据存入缓存，而且同时要实现串、并转换；

② 系统的最高时钟为信息序列输入的 2 倍，即与编码输出的时钟相同，因此要在同一个时钟沿解决存与出的问题，时序上比较复杂；

③ Max＋plusII 支持的仅是 VHDL 的一个子集，题目要求程序能在 Max＋plusII 得到综合，这就意味着可用的语句较少，增加了编程的难度。

在设计时首先要解决以下几个问题：

① 采用多大的缓存？理论上需要的缓存最大为 14 位(当第 1 个编码为 6 位，而第 2 个编码为 8 位时)，再加上有可能要存储一些暂时的变量，所以这里采用 16 位的缓冲区。

② 采用什么数据结构？这里采用了如图 13－5 所示的这么一个类似于队列的数据结构。TempData(0 to 15)为 16 位的 Std_Logic_Vector；Pos 是类型为 Integer Range 0 To 15 的变量，用以指示第 1 个尚未存储数据的单元；Code 为指出的编码，一直指向队列首 TempData(0)。

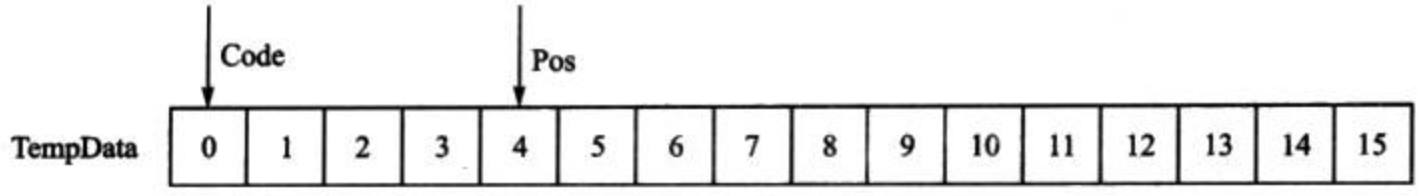


图 13-5　缓冲存储器示意图

③ 何时开始输出编码？当缓存存储的数据第 1 次达到 8 位或者多于 8 位(即 Pos＞8)时，就可以开始输出了(为什么？留给读者自己思考)。

④ 用什么语句实现数据的存储与移位？Max＋plusII 不支持诸如 A(1 to N)＜＝B(2 to N＋1)这种语句，即段下标必须以常量来表示而不能含有变量，因此就不能简单地用信号赋值语句来实现不定位数的数据存储。可以考虑用 LOOP 语句来实现，但 Max＋plusII 对 LOOP 语句的支持十分有限，因此在写程序时要注意语句的可综合性(关于 LOOP 语句的使用技巧等方面的经验总结请参见第 14 章)。

在此仅讨论以上几个方向性问题，关于缓存控制器编程上的细节问题，请读者自己在程序中体会。例程 13-3 给出了缓存控制器的程序。

例程 13-3　缓存控制器

```
LIBRARY IEEE;
USE IEEE.Std_Logic_1164.ALL;
USE IEEE.Std_Logic_Unsigned.ALL;

ENTITY Buffer_Ps IS
PORT
(Clkout:IN Std_Logic;                    --编码输出所用时钟,是信息序列输入时钟的两倍
 Reset:IN Std_Logic;
 Flag:IN Std_Logic;                      --状态机用以指示新的信源字被识别出来的信号
 Length:IN Integer RANGE 0 TO 7;         --当前信源字对应编码的长度
 Data:IN Std_Logic_Vector(0 TO 7);       --当前信源字对应的编码
 Synout:OUT Std_Logic;                   --输出同步信号
 CodeOut:OUT Std_Logic                   --编码输出
);
END;

ARCHITECTURE Behavior OF Buffer_Ps IS

SIGNAL OutputEn:Std_Logic;               --用以指示是否允许编码开始输出
```

```
SIGNAL Clear:Std_Logic;
SIGNAL Start:Std_Logic;
BEGIN
  PROCESS(Clear,Flag)
  BEGIN
    IF Clear='1' THEN
      Start<='0';
    ELSIF rising_edge(Flag) THEN
      Start<='1';
    END IF;
  END PROCESS;

  PROCESS(Reset,Clkout)
  VARIABLE Pos:Integer RANGE 0 TO 15;          --指示缓存中第 1 个尚未使用的数据单元
  VARIABLE k:Integer RANGE 0 TO 7;
  VARIABLE TempData:Std_Logic_Vector(0 TO 15); --缓冲存储器

  BEGIN
    IF Reset='1' THEN
      TempData:="0000000000000000";
      Pos:=0;
      OutputEn<='0';
      Synout<='0';
    ELSIF rising_edge(Clkout) THEN
      IF OutputEn='0' THEN                     --不允许编码输出时(初始状态)
        IF Start='1' THEN                      --级状态机识别出新的信源字
          Clear<='1';                          --将 Start 信号清 0
          FOR i IN 0 TO 7 LOOP                 --将 ROM 中对应编码存入缓存
            IF i<=Length THEN
              TempData(Pos):=Data(i);
              Pos:=Pos+1;                      --Pos 重定位,指向第 1 个尚未存储数据的单元
            ELSE
              NULL;
            END IF;
          END LOOP;
          IF (Pos>8) THEN                      --当缓存中数据量大于或等于 8 时
            OutputEn<='1';                     --允许开始输出编码
```

```
    END IF;
  ELSE
    Clear<='0';
  END IF;
ELSE                                          --允许编码输出时
  CodeOut<=TempData(0);                       --输出队列首位
  Synout<='1';
  IF Start='1' THEN                           --前级状态机识别出新信源字
    Clear<='1';
    FOR i IN 1 TO 15 LOOP
  IF i<Pos THEN                               --将原有的数据进行移位
    TempData(i-1):=TempData(i);
    k:=0;
  ELSIF k<=Length THEN                        --在原有的数据后存入新编码
    TempData(i-1):=Data(k);
    k:=k+1;
  ELSE
    NULL;
  END IF;
    END LOOP;
    Pos:=Pos+Length;                          --Pos 重定义,指向第 1 个尚未存储数据的单元
  ELSE                                        --如果没有检测到的新的信源字
    Clear<='0';
    FOR i IN 1 TO 15 LOOP                     --只须简单地进行移位即可
      IF i<Pos THEN
        TempData(i-1):=TempData(i);
      ELSE
        NULL;
      END IF;
    END LOOP;
    Pos:=Pos-1;                               --Pos 重定义,指向第 1 个尚未存储数据的单元
  END IF;
 END IF;
END IF;
END PROCESS;
END;
```

事实上，缓冲控制器从总体上来说，逻辑还是比较简单的，只是在具体实现上需要一定的技巧。例如，Max+plusII 不支持 For i IN 1 To Length LOOP 这样的语句，即 i 的范围(循环范围)必须以常量表示而不能含有变量，因此只好在 LOOP…END LOOP 之间加入判断语句；而 Max+plusII 又不支持在 LOOP 循环体中使用 EXIT 语句，只能用 IF 语句对变量 i 的大小进行判断，以决定是否执行移位或存储数据的操作。关于 LOOP 语句的用法与注意事项会在第 14 章中再作讨论。

13.2.5　顶层文件波形仿真

编码器的顶层文件(gdf 文件)及波形仿真图如图 13－6 及图 13－7 所示。仿真时输出信息序列 1101001100101100 0，对应输出的编码为 0100001000100100000010000100，与编码表一一对应。限于篇幅，此处仅附上 1 个仿真图，读者可以多试几种情况，也可将相关的其他变量和信号加入一起仿真，以便更好地了解整个系统的时序关系。

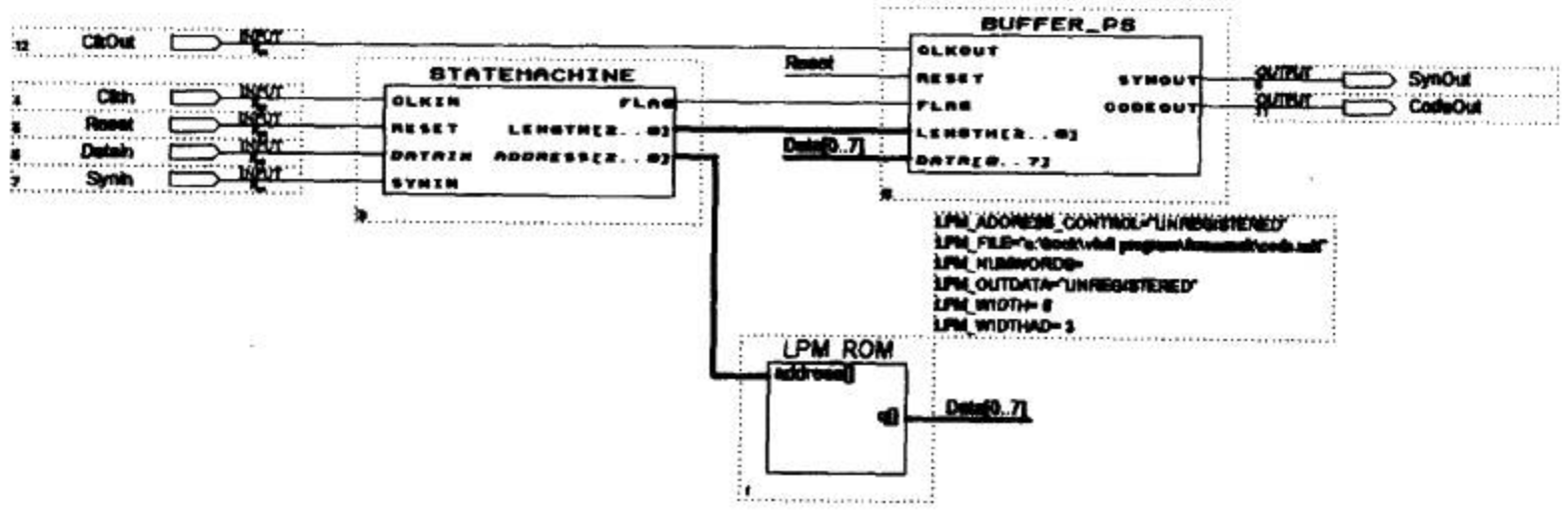


图 13－6　Franaszek 编码器顶层文件

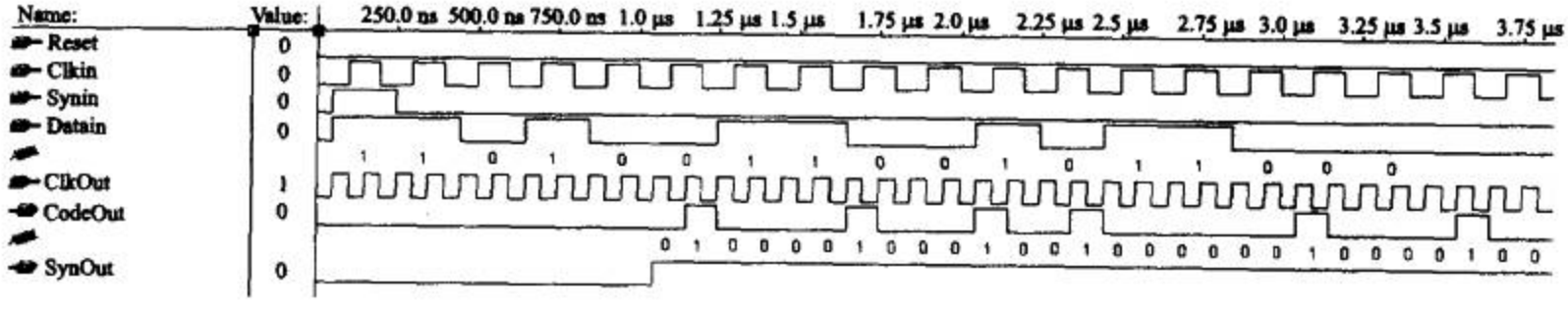


图 13－7　Franaszek 编码器仿真图

13.3　练习题

本章仅给出了编码器的范例，事实上，本题的编码器与译码器实现的原理基本一样，因此读者可以参考编码器的例程自行设计与之相对应的译码器。最终在验证时，把编码器与译码器级联，看看输出的结果是否与输入的信源字一致。

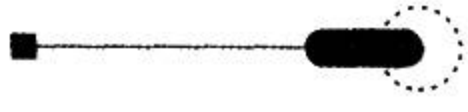
实验篇

第14章

VHDL的综合

简单地说，综合即是将 VHDL 语言转化为相应的硬件电路（准确的说，应该是转化为门级的网表文件），就像软件编译时将高级程序语言转化为机器码一样。VHDL 最初是作为一种电路模型描述语言和电路行为仿真语言被开发出来的，后来有越来越多的 VHDL 综合工具面市，才使 VHDL 从单纯的描述与仿真语言变为描述、仿真、综合三位一体的语言。但“先有标准，后有工具”这一特点，也使 VHDL 有很多的语句是不可被综合的（事实上，有一些语句根本没有综合的必要）。可综合的 VHDL 语句的集合称为可综合子集，虽然此子集已被 IEEE 标准化，但是不同的综合工具支持的语句仍不尽相同。常见的综合工具有 Synopsys 公司的 FPGA Express 和 FPGA Compiler II、Synplicity 公司的 Synplify 以及 Mentor Graphics 公司的 Leonardo Spectrum 等。当然，MAX+plus II 中也集成了综合器，但 MAX+plus II 综合器支持的 VHDL 语句较少，综合的结果也不如上述几种专业的综合器好。在读完本书之后，读者如果希望更上一层楼，那么应学会使用上述几种专业的综合器。

本章讨论的关于 VHDL 综合的问题主要基于 FPGA Compiler II 和 MAX+plus II。因为 FPGA Compiler II 提供电路原理图浏览功能，可以查看综合后生成的电路原理图，所以文中关于综合结果的讨论主要基于 FPGA Compiler II，但其结论对 MAX+plus II 综合器基本适用。由于 MAX+plus II 是本书主要针对的 EDA 软件，本章对 MAX+plus II 综合器的一些特点和局限性作了探讨，并提出一些突破其局限性的技巧。



这里特别要说明的是，如果初学者因为时间或者基础薄弱的关系不愿意花时间了解语句与综合结果的关系，那么不妨跳过 14.3.2 小节。但仍需对其他小节中提出的一些原则有所了解，并在编程中遵循这些原则，以保证 VHDL 程序是可综合的，且综合的结果与初衷相符。

14.1　LOOP 语句的综合

MAX+plus II 支持的 LOOP 语句格式只有一种，即：

```
[LOOP 标号:] FOR  循环变量 IN  循环次数范围 LOOP
                    顺序语句
                END LOOP [LOOP 标号]
```

其中，循次次数范围必须以具体数值表示，而不能出现变量。换句话说，LOOP 语句的循环边界必须是固定的。下面这个 LOOP 语句是可综合的：

```
FOR i IN  0 TO 15 LOOP
  ⋮
END LOOP;
```

而这种格式的 LOOP 语句则是不可综合的：

```
VARIABLE Length: Integer RANGE  0 TO  15;
    ⋮
FOR i IN  0 TO Length LOOP
    ⋮
END LOOP;
```

LOOP 语句不支持可变的循环边界，那么，如果需要一个次数可变的循环怎么办？下面提出两种解决方案，其中一种适用于支持 EXIT 语句的综合器，另一种则适用于不支持 EXIT 语句的综合器(如 MAX+plus II)。

例程 14－1 与例程 14－2 的编程思想其实是一样的，即先判断循环次数是否已经满足需要，若不满足，则继续执行循环体中的顺序语句；否则就跳出循环体(例程 14－1)或者不进行任何操作(例程 14－2)。如果综合器支持 EXIT 语句，建议使用例程 14－1 中的方案。

在第 13 章 Franaszek 编/译码器中的缓冲器设计一节(见 13.2.4 小节)中，使用了例程 14－2的方案(因为题目要求程序能够被 MAX+plus II 综合)，由于篇幅所限，此处就不再给

出完整实例了。

例程 14－1　跳出循环程序

```
FOR i IN 0 TO MAX LOOP          --MAX 为常数,表示可能需要用到的最大循环次数
   EXIT WHEN i=Length;          --Length 表示实际需要的循环次数
   ⋮                            --顺序语句
END LOOP;
```

例程 14－2　不进行任何操作

```
FOR i IN 0 TO MAX LOOP--MAX 为常数,表示可能需要用到的最大循环次数
  IF (i<Length) THEN--Length 表示实际需要的循环次数
     ⋮--顺序语句
  ELSE
  NULL;
  END IF;
END LOOP;
```

14.2　进程的综合

进程语句可以说是 VHDL 中最常用、最重要的语句之一,在前面的例程中,几乎找不到没有进程的程序。进程又分为组合进程和时序进程两种类型,组合进程综合后生成组合逻辑电路,而时序进程则会产生时序逻辑电路(或者时序与组合逻辑相结合的电路)。因此不但要注意进程的可综合性,而且要对进程综合的结果有清楚的认识;否则,综合的结果可能与设计初衷相违背。

14.2.1　可综合的进程格式

进程的使用十分灵活(既可以描述组合逻辑,又可以描述时序逻辑),但其格式却相对固定。一般来说,一个进程必须符合下面 5 种模板中某一种的格式才能被综合。

模板 1:

```
PROCESS(输入信号)              --所有输入信号都必须列入敏感信号表
BEGIN
   ⋮                           --必须定义所有输入条件下的输出,而且不可有反馈信号
END PROCESS;                   --此进程将综合成纯粹的组合逻辑电路
```

模板 2：

```
PROCESS(输入信号)                    --所有输入信号(包括 En)都必须列入敏感信号表
BEGIN
  IF En='1' THEN
    ⋮
  END IF;
END PROCESS;                         --此进程将综合成锁存器+组合逻辑
```

模板 3：

```
PROCESS(Clock)                       --敏感信号表中只有时钟
BEGIN
  IF rising_edge(Clock) THEN         --监测时钟上升沿,若用 falling_edge(Clock)
                                     --则监测时钟下降沿
    ⋮
  END IF;
END PROCESS;                         --此进程将综合成沿触发的触发器+组合逻辑
```

模板 4：

```
PROCESS(Clock,Reset)                 --敏感信号表中只有时钟和复位信号
BEGIN
  IF Reset='1' THEN                  --监测异步复位信号的电平值
    ⋮
  ELSIF rising_edge(Clock) THEN      --监测时钟沿
    ⋮
  END IF;
END PROCESS;                         --此进程将综合成沿触发的触发器+组合逻辑
```

模板 5：

```
PROCESS                              --无敏感信号表
BEGIN
  WAIT UNTIL rising_edge(Clock);     --此语句必须放在首位
    ⋮
END PROCESS;                         --此进程将综合成沿触发的触发器+组合逻辑
```

以上 5 种格式的进程，基本上能被所有综合器综合。但能被综合，并不代表综合的结果与设计初衷吻合。在 14.2.2 小节中将对进程的综合结果进行讨论。

14.2.2　进程综合结果的讨论

14.2.1 小节给出了进程的 5 种模板，其中模板 3 和模板 4 中的格式在前面的章节中多有应用，其综合结果也是显而易见的(对时钟沿敏感的 rising_edge()和 falling_edge()语句，在绝大多数综合器中，都会被综合成沿触发的触发器，这应该很容易理解)，在此就不多做讨论。至于模板 5，这里不建议初学者使用，因为关于 WAIT 的语句不少(如 WAIT、WAIT FOR、WAIT ON、WAIT UNTIL 等)，但可综合的只有 WAIT UNTIL，而且 WAIT UNTIL 仅在模板 5 所示的这种格式中才可综合。为了避免混淆各种 WAIT 语句，造成不必要的麻烦，建议初学者不使用 WAIT UNTIL，而采用 PROCESS(敏感信号表)的方式来检测时钟。

本小节主要讨论模板 1 和模板 2 中的进程格式综合的结果。下面给出了 4 个例程，它们表面看起来相差不多，但实际综合的结果却相距甚远。请读者仔细体会这几个例程的细微差别与它们综合结果的联系。例程 14－3 是生成纯组合逻辑电路的进程。

例程 14－3　生成纯组合逻辑电路的进程

```
PROCESS(Sel,A)                    --所有输入信号都包括在敏感信号表中
BEGIN
  IF (Sel='1') THEN               --Sel=1 时的输出
     Data<=A;
  ELSE
     Data<='0';                   --其他情况下的输出(覆盖所有输入条件)
  END IF;
END PROCESS;                      --生成纯组合逻辑电路
```

例程 14－3 综合后的结果是一个纯粹的组合逻辑电路，读者应很容易推导出 Data 的逻辑表达式为 Data=Sel AND A。FPGA Compiler II 综合的结果正是一个“与”门，而 Quartus II 综合的结果是一个多路选择器。无论如何，可以确定的是此例程生成的是组合逻辑电路。例程 14－4 是生成锁存器的进程。

例程 14－4　生成锁存器的进程

```
PROCESS(Sel,A)--所有输入信号都包括在敏感信号表中
BEGIN
  FI (Sel='1') THEN--遵从模板 2 的格式
     Data<=A;
```

```
  END IF;
END PROCESS;--生成锁存器
```

例程 14－4 与例程 14－3 相比少了一个 Else 条件下的赋值，结果生成的是锁存器。这是因为，当 Sel＝0 时，程序未对 Data 进行赋值，综合器就认为，当 Sel＝0 时电路需要锁住输出值，使之不随 A 值变化而变化，结果就生成锁存器。

通过两个程序及其综合结果的比较可以得到一个初步的结论：**组合进程中的 IF 语句中的条件必须被完全覆盖，否则综合的结果可能会引入锁存器。**这里还可以作出一个推论（此处不再给出例程）：**组合进程中的 CASE 语句中的条件也必须被完全覆盖。**

现在应该可以理解模板 1（组合进程格式）要求定义所有输入条件下的输出的原因了。例程 14－5 是敏感变量表参数不全的进程。

例程 14－5　敏感变量表参数不全的进程

```
PROCESS(Sel)--敏感变量表中未包括 A,其余与例程 14-3 相同
BEGIN
  IF (Sel='1') THEN
      Data<=A;
  ELSE
      Data<='0';
END IF;
END PROCESS;
```

例程 14－5 与例程 14－3 相比，差别仅在于敏感变量表中是否包括 A。各综合器对这个程序的综合情况各不相同：MAX＋plus II 在编译时提示出错信息；而 FPGA Compiler II 和 Quartus II 仍将其综合为“与”门和多路选择器，但出现警告信息。

MAX＋plus II 提示的出错信息是“Else Clause following a Clock edge must hold the state of signal 'Data'”。出现此提示信息的原因是：综合器将 Sel 误判为时钟信号，并试图将程序综合成时序逻辑电路，但该程序的格式又不符合综合器对时钟信号描述的要求（不可以有 ELSE 语句，参看模板 3 和模板 4），因此无法综合。

通过例程 14－5 和例程 14－3 及其在 MAX＋plus II 中的比较，可以得到另一个结论：**组合进程中所有输入信号，包括赋值符号右边的所有信号和条件表达式中所有信号，都必须包括在此进程的敏感信号表中。**这是因为，如果有输入信号没被包括在敏感信号表中，当这些信号发生变化时，进程中的输出信号不能按照组合逻辑的要求得到即时的新的信号，综合器会将进程误判为时序进程，并引入锁存器或者触发器，这样就违背了设计组合逻辑电路的初衷。

虽然 FPGA Compiler II 和 Quartus II 能综合例程 14－3，但会警告 A 不在敏感信号中。

显然，这两个综合器在分析程序时，认为设计者未将 A 包括在敏感信号表中是一种疏忽，并非想设计时序逻辑电路，所以将 A 自动添加入敏感信号表。但不能指望每一种综合器每时每刻都这么“聪明”，为了使写的 VHDL 程序能够得到大多数综合器的支持，还是将程序写得规范一些。例程 14－6 给出了上升沿描述不规范的进程。

例程 14－6　上升沿描述不规范的进程

```
PROCESS(Clock)
BEGIN
  IF (Clock='1') THEN
      Data<=A;
  END IF;
END PROCESS;
```

例程 14－6 的本意是生成一个触发器，即在 Clock 的上升沿将 A 值赋予 Data。从逻辑上说，这种写法是没有错的，因为敏感信号表中只有 Clock 信号，当 Clock 信号值发生变化且 Clock 为高电平时(这种情况只可能发生在时钟上升沿)，将 A 值赋于 Data，即描述了一个触发器。并不是所有综合器都这么认为，这个程序在 MAX＋plus II 和 Quartus II 中综合的结果是触发器，在 FPGA Complier II 中却和例程 14－4 一样生成了锁存器，并且出现警告信息：A 不在敏感信号表中。由此可知，用例程 14－6 中的方式描述时钟上升沿是不规范的(虽然从逻辑上说它应该是对的)，从而得到结论：**描述时钟沿应该采用固定的格式，即用模板 3 和模板 4 的格式，或者用 Clock'EVENT AND Clock＝'1'来描述，只有这样才能保证综合生成触发器。**

这里可以将以上的讨论总结成以下两条规则：

① 组合进程必须严格按照模板 1 的格式写，进程中的 IF 和 CASE 语句中的条件必须被完全覆盖。进程中没有反馈信号(赋值符号右边的信号不会出现在左边)，且所有输入信号(包括赋值符号右边的所有信号和条件表达式中所有信号)都必须包括在此进程的敏感信号表中。

② 时序进程可按模板 2(生成锁存器)或模板 3 和模板 4(生成触发器)写。生成锁存器的进程和组合进程一样，必须将所有输入信号都包括在进程的敏感信号表中，但条件不完全覆盖。描述时钟沿必须采用带 EVENT 的语句或按照模板 3 和模板 4 的格式写。

14.3　VHDL 可综合编程的一般规则

一个 VHDL 程序是否可以(在目标器件中)用硬件实现，主要取决于以下 3 个因素。

① VHDL 程序使用的语句是否可综合：这是必要条件，只有程序中所有的语句都被综合器支持，才有可能实现 VHDL 程序中描述的电路。

② VHDL 程序中描述的硬件元素是否被目标器件支持：有些程序原则上可综合，但目标器件不支持程序中所描述的硬件元素，比如含内部三态门的 VHDL 程序在原则上是可综合的，但有些器件（如 Lattice 1000 系列 ispLSI）不支持在内部引入三态门，因此一个 VHDL 程序是否可以用硬件实现，还取决于其目标器件的结构。所以，虽然与硬件描述语言相关的开发软件的已经发展得相当完善，甚至不必详细了解 CPLD/FPGA 的结构就可以用原理图输入或 HDL 来完成中型甚至是大型的数字系统设计，但是想成为真正优秀的设计人员，还是应该对目标器件的内部结构有所了解。

③ 目标器件的资源是否满足程序综合后对硬件资源的要求：若程序综合后所需的硬件资源（如门数、内部 RAM 容量等）超过目标器件的资源，则此程序不可在目标器件中用硬件实现。想在一个 1 万门的 FPGA（如 Flex10K10）中实现 1 024 点的 FFT（快速傅立叶变换），正如“挟泰山以超北海”，“非不为也，是不能也”！因此，面向综合的 VHDL 程序设计中必须注意硬件资源的占用问题。

由于篇幅所限，在此仅讨论第 1 个因素，即如何写出可综合的 VHDL 程序。如前所述，不同的综合器支持的语句是不同的，因此很难在可综合语句和不可综合语句之间划一条明显的界限。一般来说，在综合器的 HELP 文件中都可以找到该软件所支持的可综合子集。所以，下面不准备将所有 VHDL 语句分门别类，仅总结初学者在使用中容易出错的一些常见语句的可综合性。

(1) AFTER 语句将被忽略

由于没有对应的硬件原型可以实现固定、准确且时间可调的延时，AFTER 语句在综合时会被忽略。因此，A<=not A after 15ns 这种语句不会被综合成时延为 15 ns 的反相器。大部分综合器会将其综合成一个反相器，但不会刻意去实现 15 ns 的时延。

(2) 对时钟上、下沿同时敏感的 PROCESS 不可综合

关于 PROCESS 的综合，在 14.2 节中已经花了大量篇幅讨论，得出的结论是：只有按照 14.2.1 小节中给出的模板格式写程序，才能得到正确的结果。然而，初学者往往容易写出下面这种语句：

```
PROCESS(Clock)
BEGIN
  IF rising_edge(Clock) THEN
    ⋮
```

```
    ELSIF falling_edge(Clock) THEN
      ⋮
    END IF;
END PROCESS;
```

这种对时钟上、下沿同时敏感的 PROCESS 是不可综合的。但在同一 ARCHITECTURE 中，同时存在对时钟上升沿敏感的 PROCESS 和对时钟下降沿敏感的 PROCESS 则是允许的。如：

```
P0:PROCESS(Clock)
BEGIN
  IF rising_edge(Clock) THEN
    ⋮
  END IF;
END PROCESS P0;

P1:PROCESS(Clock)
BEGIN
  IF falling_edge(Clock) THEN
    ⋮
  END IF;
END PROCESS P1;
```

(3) LOOP 语句中循环边界范围不定的语句不可综合

如前所述，LOOP 语句中的循环边界必须用具体数值表示，而不能出现变量。请参考 14.1节。

(4) 不要对变量与信号赋初值

多数综合器会忽略变量与信号赋初值的语句。一般综合后的变量初值为数值 0 或逻辑 0，而信号则根据综合器所设条件的不同而改变，有些综合器会将信号的初始逻辑值赋为 1。为了确保程序按设计者的意图运行，最好能在进程中对信号进行赋初值的操作。

(5) 不要使用段下标可变的信号赋值

一般综合器不支持如 Data(N to N+1)<=Data(N+1 to N+2)这种段下标用变量表示的赋值语句，换句话说，赋值语句中若使用段下标元素赋值目标，则段下标必须用具体数值表示。

(6) Integer 类型须指明范围

虽然有些综合器会根据程序自动计算出 Integer 的范围，但正如前面所说的，并不是所有综合器都很“聪明”，所以用 Integer Range …To …指明整数类型的范围还是很有必要的，否则可能导致程序的不可综合或者综合后占用的资源过多。

以上即是初学者容易遇到的关于可综合性问题的总结，难免挂一漏万，希望读者在实践中不断总结，以期尽快掌握面向综合的 VHDL 编程方法与技巧。

附录 A　VHDL 保留字

abs	access	after	alias
all	and	architecture	array
assert	attribute	begin	block
body	buffer	bus	case
component	configuration	constant	disconnect
downto	else	elsif	end
entity	exit	file	for
function	generate	generic	group *
guarded	if	impure *	in
inertial *	inout	is	label
library	linkage	literal *	loop
map	mod	nand	new
next	nor	not	null
of	on	open	or
others	out	package	port
postponded *	procedure	process	pure *
range	record	register	reject *
rem	report	return	rol *
ror *	select	severity	signal
shared *	sla *	sll *	sra *
srl *	subtype	then	to
transport	type	unaffected *	units
until	use	variable	wait
when	while	with	xnor *
xor			

加 * 号的为 VHDL－1993 新加入的关键字

附录 B　常用语法

声明与赋值

格　式

```
SIGNAL 信号名：数据类型；                    --声明信号
VARIABLE 变量名：数据类型；                  --声明变量
CONSTANT 常数名：数据类型：=设置值；
目标信号名<=表达式；                         --信号赋值
目标信号名：=表达式；                        --变量赋值
```

范　例

```
SIGNAL Data_S: Integer Range 0 To 1023;    --声明信号
VARIABLE Data_V: Integer Range 0 To 1023;
CONSTANT Data_C: Integer:=15;
Data_S<=5;                                 --信号赋值
Data_C:=15;                                --变量赋值
```

使用要点

- 信号名、变量名和常量名不可与 VHDL 保留字(附录 A)相同；
- 注意变量赋值的即时性与信号赋值的延时效应；
- 常用数据类型与端口类型相同，参见本附录中的 ENTITY 项。

CASE

格　式

```
CASE 表达式 IS
    WHEN   选择值[|选择值] =>  顺序语句;
                              --若有多个选择值用"|"间隔;
    WHEN   选择值[|选择值] =>  顺序语句;
        ⋮
    WHEN OTHERS=>顺序语句;
END CASE;
```

范　例

```
CASE Sel IS
    WHEN  "00"=>DOUT<=Data0;
    WHEN  "01"=>DOUT<=Data1;
    WHEN  "10"=>DOUT<=Data2;
    WHEN  "11"=>DOUT<=Data3;
    WHEN OTHERS=>DOUT<=Data4";
```

使用要点

- 选择值不可以重复或重叠，例如不可以同时出现两次 WHEN "00"(重复)，也不可以同时出现 WHEN "00"|"01"和 WHEN "00"|"11"(选择值"00"重叠)；
- 当 CASE 语句的选择值无法覆盖所有情况时，要用 OTHERS 指定未能列出的其他所有情况的输出值；
- CASE 语句一般被综合成多路复用器(Multiplexer)。

ENTITY

格 式

```
ENTITY  实体名  IS
[GENERIC (常数名:数据类型:设定值)] --类属参数说明
PORT
( 端口名1:端口方向  端口类型;
  端口名2:端口方向  端口类型;
     ⋮
  端口名n:端口方向  端口类型        --最后一个端口声明语句后不加分号
);
END [实体名];
```

范 例

```
ENTITY FreDevider IS
GENERIC (N:Integer:=4);
Port
(Clkin:IN Std_Logic;
 Clkout:OUT Std_Logic
);
END;
```

使用要点

- 常用端口方向：IN(输入)，OUT(输出)，IN/OUT(输入/输出)，BUFFER(输出，但结构体内部可读取)；
- 常用端口类型：Boolean(布尔量：FALSE/TRUE)，Bit(位：'0'/'1')，Bit_Vector(位矢量：基于 Bit 类型的数组)，Std_Logic(标准逻辑位：'U'/'X'/'0'/'1'/'Z'/'W'/'L'/'H'/'－')，Std_Logic_Vector(标准逻辑矢量：基于 Std_Logic 类型的数组)，Integer(整数量：须用 Range…To…指定数值范围)。

IF

格　式

```
IF  条件式 1 THEN
  顺序语句
ELSIF  条件式 2 THEN
  顺序语句
ELSE
  顺序语句
END IF;
```

范　例

```
IF S1='1' AND S2='0' THEN
 P:=NOT P;
 Q:=R;
 IF S3='1' THEN
     DATA<=A;
 ELSIF S4='1' THEN
     DATA<=B;
 ELSE
     DATA<=C;
 END IF;
END IF;
```

使用要点

- IF 语句可以嵌套，但层数不宜过多；
- 可以用几个 ELSIF 共同实现具有优先级差别的条件判断，如果所需判断的条件没有优先级的差别，且条件之间没有重叠的情况，那么建议使用 CASE 语句；
- 在进程中用 IF 语句描述组合逻辑电路时，务必要覆盖所有的情况，否则综合后将引入锁存器，违背设计初衷。

LOOP

格　式

```
[LOOP 标号:]  FOR 循环变量 IN 循环次数范围 LOOP
                顺序语句
              END LOOP [LOOP 标号];
```

范　例

```
FOR i IN 0 TO 7 LOOP
    Data_out(i)<=Data_in(i)
    Sum:=Sum+i;
END LOOP;
```

使用要点

- 循环变量是一个临时的变量，仅在此 LOOP 语句中有效，因此不需要事先定义。但要注意，在 PROCESS 的声明区不要定义与此同名的变量。
- 循环次数范围只能用常数表示，不可出现变量。

PROCESS

格　式

```
[进程标号:] PROCESS (敏感信号参数表)
[声明区];
BEGIN
顺序语句
END PROCESS [进程标号];
```

```
[进程标号:] PROCESS
[声明区];
BEGIN
  WAIT UNTIL (激活进程的条件);
  顺序语句
END PROCESS [进程标号];
```

范　例

```
PROCESS(Reset,Clock)
BEGIN
  IF Reset='1' THEN
    Num<=0;
  ELSIF rising_edge(Clock) THEN
    IF Num=3 THEN
      Num<=0;
    ELSE
      Num<=Num+1;
    END IF;
  END IF;
END PROCESS;
```

关于进程的各种格式及其使用要点，在第 14 章有详细讨论，此处不再重复。

WHEN_ELSE

格　式

```
赋值目标信号<=  表达式 1 WHEN 赋值条件 1 ELSE
                表达式 2 WHEN 赋值条件 2 ELSE
                ⋮
                表达式 n WHEN 赋值条件 n ELSE
                表达式;
```

范　例

```
A<="111" WHEN I(7)='1' ELSE
   "110" WHEN I(6)='1' ELSE
   "101" WHEN I(5)='1' ELSE
   "100" WHEN I(4)='1' ELSE
   "011" WHEN I(3)='1' ELSE
   "010" WHEN I(2)='1' ELSE
   "001" WHEN I(1)='1' ELSE
   "000" WHEN I(0)='1' ELSE
   "111";
```

使用要点

- 赋值子句有优先级的差别，按书写先后顺序从高到低排列；
- 某一赋值条件得到满足后即执行赋值操作，不再测试下面的赋值条件。

WITH_SELECT

格　式

```
WITH 选择表达式 SELECT
  赋值目标信号 <=  表达式 1   WHEN 选择值 1,
                   表达式 2   WHEN 选择值 2,
                   ⋮
                   表达式 n   WHEN OTHERS;
```

范　例

```
WITH Sel SELECT
    DOUT<=A WHEN "000",
           B WHEN "001",
           C WHEN "010"|"011",
           D WHEN "100",
           E WHEN OTHERS;
```

使用要点

- 选择值要覆盖所有可能的情况；
- 各选择值必须互斥，不能出现条件重复或重叠的情况。

附录 C　IEEE 库中 ARITH 程序包声明

IEEE 库中的 ARITH 程序包，包含了 3 个类型的定义（即 UNSIGNED、SIGNED、SMALL_INT）、四则运算符号的重载定义和一些常用的类型转换函数。第 2 章中说过，VHDL 是一种“强类型语言”，虽然 INTEGER 的本质是 STD_LOGIC_VECTOR，但是它们之间不能直接进行赋值，所以本程序包中的类型转换函数就显得十分重要。

为方便读者查阅 ARITH 程序包中所包括的函数，本附录给出了 ARITH 程序包的声明部分，但为节省篇幅，没有给出程序包体（body）部分，感兴趣的读者可以在 Max+PLUS II 的＜maxplus2\vhdl93\ieee＞目录下的“arithb. vhd”文件中找到 ARITH 程序包体的描述。

```
------------------------------------------------------------------
--                                                              --
-- Copyright (c) 1994, 1990 - 1993 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                               --
--                                                              --
--  Description:  This package contains a set of arithmetic operators  --
--                and functions.                                --
--                                                              --
--                                                              --
--                                                              --
------------------------------------------------------------------
```

```
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: 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) 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: UNSIGNED) 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 SHR(ARG: UNSIGNED; COUNT: UNSIGNED) return UNSIGNED;
function SHR(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 INTEGER;

function CONV_UNSIGNED(ARG: INTEGER; SIZE: INTEGER) return UNSIGNED;
function CONV_UNSIGNED(ARG: UNSIGNED; SIZE: INTEGER) return 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;
------------------------------------------------------------
-- attributes for conversion functions
------------------------------------------------------------
attribute Synth_Conversion_Function of CONV_INTEGER: function is "INTEGER";
attribute Synth_Conversion_Function of CONV_UNSIGNED: function is "UNSIGNED";
attribute Synth_Conversion_Function of CONV_SIGNED: function is "SIGNED";
attribute Synth_Conversion_Function of CONV_STD_LOGIC_VECTOR:
                                        function is "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
-- return STD_LOGIC_VECTOR(SIZE-1 downto 0)
function SXT(ARG: STD_LOGIC_VECTOR; SIZE: INTEGER) return STD_LOGIC_VECTOR;

end std_logic_arith;
```

附录 D　IEEE 库中 SIGNED/UNSIGNED 程序包声明

一般来说，四则运算的运算符和关系运算符的左、右两边的操作数类型应该相同，但若该程序调用此程序包，则可实现 STD_LOGIC_VECTOR 和 INTEGER 的混合运算。值得注意的是，此程序包中的 STD_LOGIC_VECTOR 是以补码形式表示的带符号的二进制数。

读者可以从本附录的 SIGNED 程序包声明中了解各运算符能够进行怎样的混合运算，其中 *L* 表示运算符左边的操作数(若是一元运算符，则是惟一操作数)；而 *R* 表示运算符右边的操作数。

IEEE 库中还有一个 UNSIGNED 程序包，除其中的 STD_LOGIC_VECTOR 是以原码形式表示的不带符号的二进制数外，其余与 SIGNED 程序包相同(其程序包声明是相同的，故此处不再重复)。为节省篇幅，本附录没有给出 SIGNED 和 UNSIGNED 的程序包体(body)部分，感兴趣的读者可以在 Max＋PLUS II 的＜maxplus2\vhdl93\ieee＞目录下的“signedb.vhd”和“unsignedb.vhd”文件中找到这两个程序包体的描述。

```
------------------------------------------------------------------
-- Copyright (c) 1994, 1990 - 1993 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_signed                               --
--                                                                --
--   Description:  This package contains a set of signed arithemtic --
```

```
--                    operators and functions. --
-----
------------------------------------------------------------

library IEEE;
use IEEE. std_logic_1164. all;
use IEEE. std_logic_arith. all;

package std_logic_signed is

    function "+"(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return STD_LOGIC_VEC-
TOR;
    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_VEC-
TOR;
    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) return STD_LOGIC_VECTOR;
    function "ABS"(L: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;


    function " * "(L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR) return STD_LOGIC_VEC-
TOR;

    function "<"(L: STD_LOGIC_VECTOR; R: INTEGER) return BOOLEAN;
    function "<"(L: INTEGER; 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: INTEGER) return BOOLEAN;
    function ">"(L: INTEGER; 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: INTEGER) return BOOLEAN;
    function "="(L: INTEGER; 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 CONV_INTEGER(ARG: STD_LOGIC_VECTOR) return INTEGER;

end std_logic_signed;
```

参 考 文 献

1 肖冰，郭莉，安德宁. 数字电路与逻辑设计实验技术. 北京：北京邮电大学出版社，2000

2 黄任. AVR 单片机与 CPLD/FPGA 综合应用入门. 北京：北京航天航空大学出版社，2004

3 潘松，王国栋. VHDL 实用教程. 成都：电子科技大学出版社，2001

4 黄正谨，徐坚，章小丽，熊明珍，等. CPLD 系统设计技术入门与应用. 北京：电子工业出版社，2002

5 侯伯亨，顾新. VHDL 硬件描述语言与数字逻辑电路设计(修订版). 西安：西安电子科技大学出版社，1999

6 卢毅，赖杰. VHDL 与数字电路设计. 北京：科学出版社，2001

7 褚振勇. FPGA 设计与应用. 西安：西安电子科技大学出版社，2002

有关此电子图书的说明

本人由于一些便利条件，可以帮您提供各种中文电子图书资料，且质量均为清晰的 PDF 图片格式，质量要高于网上大量传播的一些超星 PDG 的图书。方便阅读和携带。只要图书不是太新，文学、法律、计算机、人文、经济、医学、工业、学术等方面的图书，我都可以帮您找到电子版本。所以，当你想要看什么图书时，可以联系我。我的 QQ 是：85013855，大家可以在 QQ 上联系我。

此 PDF 文件为本人亲自制作，请各位爱书之人尊重个人劳动，敬请您不要修改此 PDF 文件。因为这些图书都是有版权的，请各位怜惜电子图书资源，不要随意传播，否则，这些资源更难以得到。