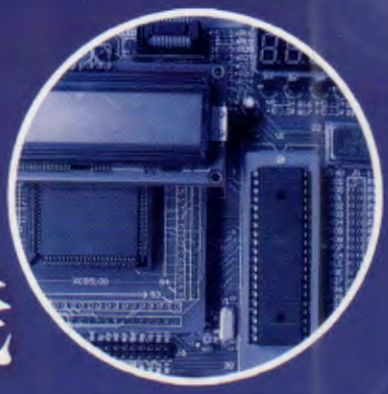
CPLD 入门与实践

周兴华 编著

中国电力出版社
CHINA ELECTRIC POWER PRESS

CPLD 入门与实践



本书以通俗的语言、翔实的实例，教会读者从零开始学会Xilinx公司的CPLD设计。当然由于设计语言Verilog HDL的通用性，读者也可以快速地将从本书学到的设计知识应用到其他公司的CPLD上。

本书的实验芯片是基于Xilinx公司的XC95108，通过阅读本书及进行实践后，读者可以发现，理论与实践的紧密结合是本书的最大特色，这样能够由浅入深、循序渐进地引导读者学习、实践，再学习、再实践，一步一步地掌握CPLD的设计。

ISBN 978-7-5123-1496-2
9 787512 314962 >
定价：35.00元(1CD)
上架建议：计算机/CPLD

周兴华

1986年毕业于西安交通大学电子技术专业，多年来一直从事工业自动化控制的设计及应用推广，有20多年的电子产品设计制造经验及嵌入式系统设计经验。早在1979年就自行组装调试成功高灵敏中、短波收音机，1987年获《无线电》KD、NS音响电路设计制作竞赛鼓励奖，1990年获《电子世界》电子电路设计制作竞赛一等奖，1992年获第五届上海市“星火杯”发明创造竞赛四等奖。著有《AVR单片机C语言高级程序设计》、《实用遥控控制线路200例》、《实用遥控专用器件速查手册》等专著。现主要从事嵌入式智能化电子产品的研发、教学与推广。2010年创建的“周兴华单片机培训中心”已培训出10余个班级的近百名学员，在各行各业的科研生产中发挥着重要的作用，得到各界的好评。

责任编辑　刘　炽
联系电话　010-63412395
电子信箱　liuchi1030@163.com

CPLD 入门与实践

周兴华　编著

中国电力出版社
CHINA ELECTRIC POWER PRESS

内 容 简 介

本书以通俗的语言、翔实的实例，教会读者从零开始学会 Xilinx 公司的 CPLD 设计。当然由于设计语言 Verilog HDL 的通用性，读者也可以快速地将从本书学到的设计知识应用到其他公司的 CPLD 上。

本书的实验芯片是基于 Xilinx 公司的 XC95108，通过阅读本书及进行实践后，读者可以发现，理论与实践的紧密结合是本书的最大特色，这样能够由浅入深、循序渐进地引导读者学习、实践，再学习、再实践，一步一步地掌握 CPLD 的设计。

本书附有光盘，含本书所有的程序设计文件。本书适合用作高职高专或中等职业技术学校、电视大学、培训中心等的教学用书，也非常适合广大电子爱好者作为 CPLD 入门的自学用书。

图书在版编目（CIP）数据

CPLD入门与实践／周兴华编著. —北京：中国电力出版社，2011.3

ISBN 978-7-5123-1496-2

Ⅰ. ①C… Ⅱ. ①周… Ⅲ. ①可编程序逻辑器件 Ⅳ. ①TP332.1

中国版本图书馆CIP数据核字（2011）第044354号

中国电力出版社出版、发行

（北京市东城区北京站西街 19 号 100005 http://www.cepp.sgcc.com.cn）

汇鑫印务有限公司印刷

各地新华书店经售

*

2011 年 5 月第一版 2011 年 5 月北京第一次印刷

710 毫米 ×980 毫米 16 开本 16 印张 257 千字

印数 0001—3000 册 定价 **35.00** 元（含 1CD）

前　言

当前，可编程逻辑器件的发展非常迅速，已出现高达数百万门的 FPGA 和数十万门的 CPLD。就目前的应用而言，集成度为数千门的 CPLD 应用最广泛。

CPLD 一般主要应用于高频数字逻辑领域及单片机外围配置。例如，各种门电路、计数器、触发器、锁存器以及更复杂的逻辑控制器等，甚至还可以直接构造出 CPU 内核。单片机的特长是使用方便、运算精确灵活、控制能力强，将 CPLD 与单片机结合起来设计应用系统之后，充分发挥了它们各自的特长，使其优势互补。这样设计的系统结构简单、功能强大、性价比非常高。因此，CPLD 与单片机的结合使用将使系统以前所未有的优势赢得市场。

本书以初学者为对象，从零开始，实践为主，循序渐进地讲解当前热门的 CPLD 设计技术。本书的特点是使用入门难度浅、程序长度短且又能立竿见影的初级实例，通过实例操作，详细介绍了如何迅速掌握使用 Xilinx 公司的 XC95108，帮助初学者快速掌握 CPLD 的高效设计。

本书的学习成本较低，整套实验器材的配置在 400 元左右。

本书所配的实验器材如下：

- Keil C51、Xilinx ISE 集成开发环境。
- MCU & CPLD DEMO 试验母板及 XC95108 子板（配 AT89S51 及 XC95108）。
- Xilinx CPLD JTAG 并口程序下载器。
- 单片机 USB 程序下载器。
- 16×2 字符型液晶显示模组。
- 9V/800mA 专用电源。

随书所附的光盘中提供了本书的所有软件设计程序文件，可供读者朋友参考学习。

参与本书编写的主要工作人员有周兴华、吕超亚、傅飞峰、周济华、沈惠

莉、周渊、周国华、丁月妹、周晓琼、钱真、付毛仙、吕丁才、唐群苗、吕亚波等，全书由周兴华统稿并审校。

限于作者水平，必定还存在不少缺点或漏洞，诚挚欢迎广大读者提出意见并不吝赐教。

如果读者朋友自制或购买书中介绍的学习、实验器材有困难时，可与作者联系，咨询购买事宜。

联系方式如下：

地址：上海市闵行区莲花路 2151 弄 57 号 201 室

邮编：201103

联系人：周兴华

电话：021-64654216　13774280345　13044152947

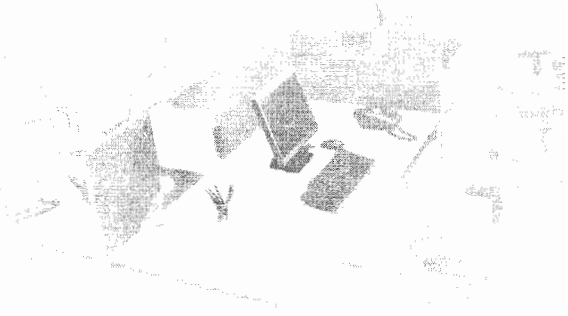
技术支持 E-mail：zxh2151@sohu.com

zxh2151@yahoo.com.cn

培训中心网站：http://www.zxhmcu.com　http://www.hlelectron.com

编　者

目　录



第1章

可编程逻辑器件简介

可编程逻辑器件（Programmable Logic Device，PLD）是20世纪70年代发展起来的一种新型器件，是用于数字逻辑系统设计的主要硬件基础。可编程逻辑器件的出现给数字系统的设计方式带来了革命性的变化。多年来，人们设计数字电路系统都是使用标准的数字集成电路芯片，如 74/54 系列（TTL）、4000/4500系列（CMOS）等，根据设计的功能从这些标准的芯片中进行选择，然后搭建成一个完整的数字电路应用系统。使用这样的方法设计出的系统，不仅芯片数量多、印板面积大，而且可靠性差，并且毫无设计的灵活性可言。

可编程逻辑器件PLD出现后，改变了人们的传统设计方法，人们可以直接使用PLD芯片进行数字电路系统的设计。例如，可以直接设计芯片内部的数字逻辑并定义输入/输出引脚等，使得设计过程从原来的印板级设计上升到主要是芯片级设计。由于PLD设计时引脚定义非常灵活，不仅减轻了电路原理和印板设计的难度，提高了设计效率，而且大大减少了芯片的数量和种类，缩小了印板面积，降低了功耗，并极大地提高了系统工作的可靠性。

1.1 可编程逻辑器件的发展历程

PLD 器件的最早原型是 20 世纪 70 年代中期出现的可编程逻辑阵列 PLA（Programmable Logic Array），PLA 在结构上由可编程的与阵列和可编程的或阵列构成，阵列规模比较小，编程也很烦琐，它并没有得到广泛应用。随后出现了可编程阵列逻辑 PAL（Programmable Array Logic），PAL 由可编程的与阵列

和固定的或阵列组成，采用熔丝编程方式，它的设计比较灵活，器件速度快，因而成为第一个得到普遍应用的 PLD 器件。

后来 Lattice 公司发明了通用阵列逻辑 GAL（Generic Array Logic）。GAL 器件采用了输出逻辑宏单元（OLMC）的结构和 EEPROM 工艺，具有可编程、可擦除、可长期保持数据的优点，使用灵活，所以 GAL 得到了极为广泛的应用。

80 年代中期，Altera 公司推出了一种新型的可擦除、可编程的逻辑器件 EPLD（Erasable Programmable Logic Device），EPLD 采用 CMOS 和 UVEPROM 工艺制成，集成度更高，设计也更灵活，但它的内部连线功能并不是很强。

EPLD 经 Lattice 公司改进后就成为 CPLD（Complex Programmable Logic Device），即复杂可编程逻辑器件，采用 EEPROM 工艺制作，与 EPLD 相比，CPLD 增强了内部连线，对逻辑宏单元和 I/O 单元也有重大的改进，它的性能更好，使用更方便。并且，现在的大部分 CPLD 都具备在系统编程（ISP）功能。CPLD 是当前的主流 PLD 器件之一。

1985 年，Xilinx 公司推出了现场可编程门阵列 FPGA（Field Programmable Gate Array），这是一种采用单元型结构的新型 PLD 器件。它采用 CMOS 的 SRAM 工艺制作，在结构上和阵列型 PLD 不同，它的内部由许多独立的可编程逻辑单元构成，各逻辑单元之间可以灵活地相互连接，具有密度高、速度快、编程灵活、可重新配置等优点，FPGA 也是当前主流的 PLD 器件之一。

1.2 可编程逻辑器件的基本结构

可编程逻辑器件的基本结构如图 1-1 所示。

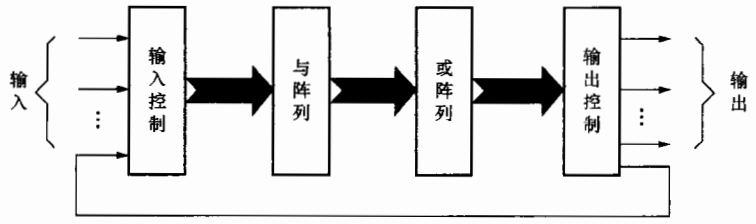


图 1-1 可编程逻辑器件的基本结构

由输入控制电路、与阵列、或阵列以及输出控制电路组成。在输入控制电路中，输入信号经过输入缓冲单元产生每个输入变量的原变量和反变量，并作

为与阵列的输入项。与阵列由若干个与门组成，输入缓冲单元提供的各输入项被有选择地连接到各个与门输入端，每个与门的输出则是部分输入变量的乘积项。各与门输出又作为或阵列的输入，这样或阵列的输出就是输入变量的与或形式。输出控制电路将或阵列输出的与或式通过三态门、寄存器等电路，一方面产生输出信号，另一方面作为反馈信号送回输入端，以便实现更复杂的逻辑功能。因此，利用可编程逻辑器件可以方便地实现各种逻辑功能。

1.3　可编程逻辑器件的特点及分类

可编程逻辑器件按照不同的类型和标准，可以有许多种分类方法。

如果按器件的集成度划分，可分为低密度可编程逻辑器件（LDPLD）和高密度可编程逻辑器件(HDPLD)。常见的低密度可编程逻辑器件有 PROM、PLA、PAL 和 GAL 等，通常称为简单 PLD 器件；常见的高密度可编程逻辑器件有 CPLD 以及 FPGA 等，称为复杂 PLD 器件。图 1-2 为其分类示意图。

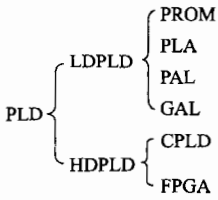


图 1-2　可编程逻辑器件按集成度分类示意

根据可编程逻辑器件的与阵列和或阵列的编程情况以及输出形式，低密度可编程逻辑器件（LDPLD）通常可分为四类。

（1）与阵列固定、或阵列可编程的 PLD 器件。

这类 PLD 器件以可编程只读存储器 PROM 为代表。可编程只读存储器 PROM 是组合逻辑阵列，它包含一个固定的与阵列和一个可编程的或阵列。PROM 中的与阵列是全译码形式，它产生 n 个输入变量的所有最小项。PROM 的每个输出端通过或阵列将这些最小项有选择地进行或运算，即可实现任何组合逻辑函数。由于与阵列能够产生输入变量的全部最小项，所以用 PROM 实现组合逻辑函数不需要进行逻辑化简。但是随着输入变量数的增加，与阵列的规模会迅速增大，其价格也随之大大提高。而且与阵列越大，译码开关时间就越长，相应的工作速度也越慢。因此，实际上只有规模较小的 PROM 可以有效地实现组合逻辑函数，而大规模的 PROM 价格高，工作速度低，一般只作存储器使用。

（2）与阵列和或阵列均可编程的 PLD 器件。

以可编程逻辑阵列 PLA 为代表。PLA 和 PROM 一样也是组合型逻辑阵列，

与 PROM 不同的是它的两个逻辑阵列均可编程。PLA 的与阵列不是全译码形式，它可以通过编程控制只产生函数最简与或式中所需要的与项。因此 PLA 器件的与阵列规模减小，集成度相对高一些。

但是，由于 PLA 只产生函数最简与或式中所需要的与项，因此 PLA 在编程前必须先进行函数化简。另外，PLA 器件需要对两个阵列进行编程，编程难度较大。而且，PLA 器件的开发工具应用不广泛，编程一般只能由生产厂家完成，目前基本上已不使用。

（3）以可编程阵列逻辑 PAL 为代表的与阵列可编程、或阵列固定的 PLD 器件。

这种器件的每个输出端是若干个乘积项之或，其中乘积项的数目固定。通常 PAL 的乘积项数允许达到 8 个，而一般逻辑函数的最简与或式中仅需要完成 3~4 个乘积项或运算。因此，PAL 的这种阵列结构可以满足大多数逻辑函数的设计要求。

PAL 有几种固定的输出结构，如专用输出结构、可编程 I/O 结构、带反馈的寄存器输出结构以及异或型输出结构等。一种输出结构只能实现一种类型的逻辑函数，因此，PAL 的通用性较差，目前基本上也已不使用。

（4）具有可编程输出逻辑宏单元的通用 PLD 器件。

以通用型可编程阵列逻辑 GAL 器件为主要代表，GAL 器件的阵列结构与 PAL 相同，都是采用与阵列可编程而或阵列固定的形式，两者的主要区别是输出结构不同：PAL 的输出结构是固定的，一种结构对应一种类型芯片，如果系统中需要几种不同的输出形式，就必须选择多种芯片来实现；GAL 器件的每个输出端都集成有一个输出逻辑宏单元 OLMC（Out Logic Macro Cell），输出逻辑宏单元是可编程的，通过编程可以决定该电路是完成组合逻辑还是时序逻辑，是否需要产生反馈信号，并能实现输出使能控制以及输出极性选择等。因此，GAL 器件通过对输出逻辑宏单元 OLMC 的编程可以实现 PAL 的各种输出结构，使芯片具有很强的通用性和灵活性，目前使用很广泛。

高密度可编程逻辑器件（HDPLD）主要包括 CPLD 和 FPGA 两类器件，这两类器件也是当前 PLD 的主流应用器件。

（1）CPLD 是基于乘积项（Product-Term）技术，采用 Flash（或 EEPROM）工艺制作的 PLD 器件，配置数据掉电后不会丢失，一般多用于 5000 门以下的中小规模设计，适合做复杂的组合逻辑，如译码器等。

（2）FPGA 采用静态存储器（SRAM）结构，属于单元型的 PLD 器件，它的基本结构是可编程逻辑块，由许多这样的逻辑块排列成阵列状，逻辑块之间由水平连线和垂直连线通过编程连通。FPGA 器件采用查找表（Look-Up Table）技术及 SRAM 工艺，配置数据易失，需要外挂非易失性器件进行配合。FPGA 的集成度高（其密度远高于 CPLD），触发器多，多用于较大规模的设计，适合做复杂的时序逻辑，如数字信号处理和各种算法等。

1.4　可编程逻辑器件的逻辑约定方法

由于可编程逻辑器件的阵列结构特点，现在广泛采用如下的逻辑约定方法。

1. PLD 输入缓冲单元

PLD 的输入缓冲单元由若干个缓冲器组成，每个缓冲器产生该输入变量的原变量和反变量，其逻辑表示方法如图 1-3 所示，表 1-1 是它所对应的真值表。

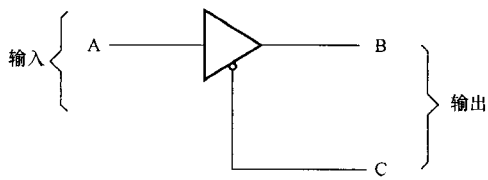


图 1-3　PLD 输入缓冲单元逻辑表示方法

表 1-1　　PLD 输入缓冲单元真值表

A	B	C
0	0	1
1	1	0

2. PLD 与门

以三输入与门为例，其 PLD 表示法如图 1-4 所示。A、B、C 为输入项，D 为输出项。其中 D=A×B×C。

3. PLD 或门

以三输入或门为例，其 PLD 表示法如图 1-5 所示。A、B、C 为输入项，D 为输出项。其中 D=A+B+C。

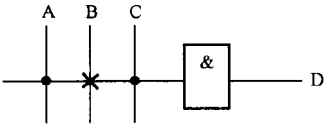


图 1-4　PLD 三输入与门

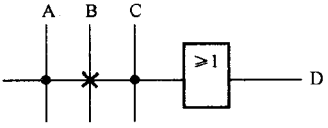


图 1-5　PLD 三输入或门

4. PLD 连接方式

PLD 有三种不同的连接方式：固定连接、可编程连接和断开，其表示方法如图 1-6 所示。

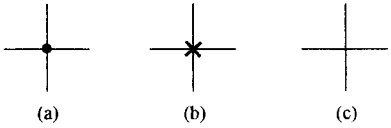
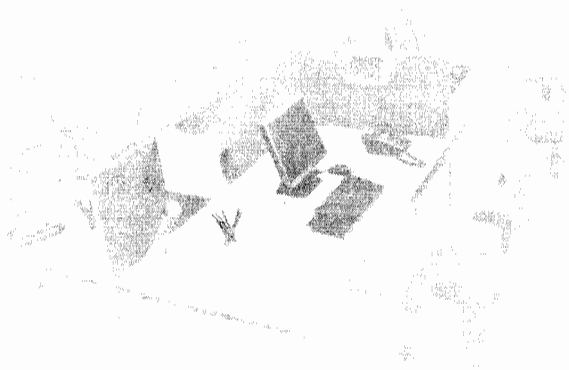


图 1-6　PLD 的连接方式

（a）固定连接；（b）可编程连接；（c）断开

图 1-6（a）表示的固定连接是厂家在生产芯片时连好的，是不可改变的；图 1-6（b）表示可编程连接，在熔丝编程工艺的 PLD 中（如 PAL），接通对应于熔丝未熔断，断开对应于熔丝已熔断，又称该单元被编程；图 1-6（c）表示断开，这种断开，一种可能是该点原本就没有连接，另一种可能是由于熔丝熔断而形成的可编程断开。



第2章

CPLD/FPGA的结构与特性

2.1 CPLD结构简介

CPLD是在PAL、GAL的基础上发展起来的阵列型的PLD器件，具有高密度、高速度的优点。从结构上来看，CPLD大都包含了三种结构：宏单元、可编程I/O单元和可编程内部连线。

2.1.1 宏单元

宏单元是CPLD器件的基本单元，宏单元内部主要包括“与或”阵列、触发器和多路选择器等电路，能独立地配置为组合或者时序工作方式。在GAL器件中，逻辑宏单元与I/O单元做在一起，称为输出逻辑宏单元（OLMC）。但高密度CPLD的逻辑宏单元都做在内部，称为内部逻辑宏单元。

逻辑宏单元具有以下的特点：

（1）多触发器和“隐埋”触发器结构。

GAL器件每个输出宏单元只有一个触发器，而CPLD的宏单元内一般有多个触发器，其中只有一个触发器是与输出端相连的。其余触发器的输出不与输出端相连，但可以反馈到与阵列，构成更复杂的时序电路，这些触发器称为“隐埋”触发器。这种结构对于I/O口有限的CPLD器件来说，增加了其内部资源的利用率。

（2）乘积项（Product Terms）共享结构。

大多数逻辑函数能够用每个宏单元中的乘积项来实现，但某些逻辑函数比

较复杂，要实现它们的话，需要附加乘积项。为提供所需要的逻辑资源，可以借助可编程开关将同一宏单元（或其他宏单元）中的未使用的乘积项联合起来使用，称为乘积项共享。

Altera 的 CPLD 无一例外地采用了乘积项共享结构，利用乘积扩展项可保证在实现逻辑综合时，用尽可能少的逻辑资源，得到尽可能快的工作速度。

2.1.2 可编程 I/O 单元

输入输出单元（I/O 单元）必须考虑下面的要求：

（1）能够兼容 TTL 和 CMOS 多种接口电压和接口标准。

（2）可配置为输入、输出、双向 I/O、集电极开路和三态门等各种组态。

（3）能够提供适当的驱动电流，以直接驱动小功率器件（如 LED）。

（4）降低功率消耗，防止过冲和减少电源噪声。

I/O 单元必须考虑能够支持多种接口电压。随着半导体工艺线宽的不断缩小，从器件功耗的要求出发，器件的内核必须采用更低的电压。比如当工艺线宽为 1.2～0.5μm 时，器件一般采用 5V 电压供电；当工艺线宽为 0.35μm 时，器件的供电电压为 3.3V；当工艺线宽为 0.25μm 时，I/O 单元与芯片内核的供电电压不再相同，内核的电压一般为 2.5V，I/O 单元的工作电压为 3.3V，并且能兼容 5V 和 3.3V 的器件；当工艺线宽为 0.18μm 时，器件内核一般采用 1.8V 的电压，I/O 单元则要能够兼容 2.5V 和 3.3V 的电压。

2.1.3 可编程连线阵列（PIA）

可编程连线阵列的作用是在各逻辑宏单元之间以及逻辑宏单元和 I/O 单元之间提供互连网络。在 FPGA 中基于通道布线方案的布线延时是累加的、可变的和与路径有关的。而在 CPLD 器件中，一般采用固定长度的线段来进行连接，这种连线的好处是有固定的延时，使得时间性能更加容易预测。

2.2 基于乘积项的 CPLD 原理与结构

基于乘积项的 CPLD 芯片有 Altera 的 MAX7000 系列，MAX3000 系列（EEPROM 工艺）、Xilinx 的 XC9500 系列（Flash 工艺）和 Lattice、Cypress 的大部分产品（EEPROM 工艺）。

以典型的 CPLD 芯片 MAX7000 为例，图 2-1 是这种 CPLD 的结构示意图，主要包括宏单元、可编程连线（PIA）和 I/O 控制块。宏单元是 CPLD 的基本结构，由它来实现基本的逻辑功能。可编程连线 PIA 负责信号传递，连接所有的宏单元。I/O 控制块负责输入输出的特性控制，比如可以设定集电极开路输出、摆率控制、三态输出等。图中上部的 INPUT/GCLK1，INPUT/GCLRn，INPUT/OE1，INPUT/OE2 是全局时钟、清零和输出使能信号，这几个信号有专用连线与 CPLD 中每个宏单元相连，信号到每个宏单元的延时相同并且延时最短。MAX7000 中宏单元的具体结构如图 2-2 所示。

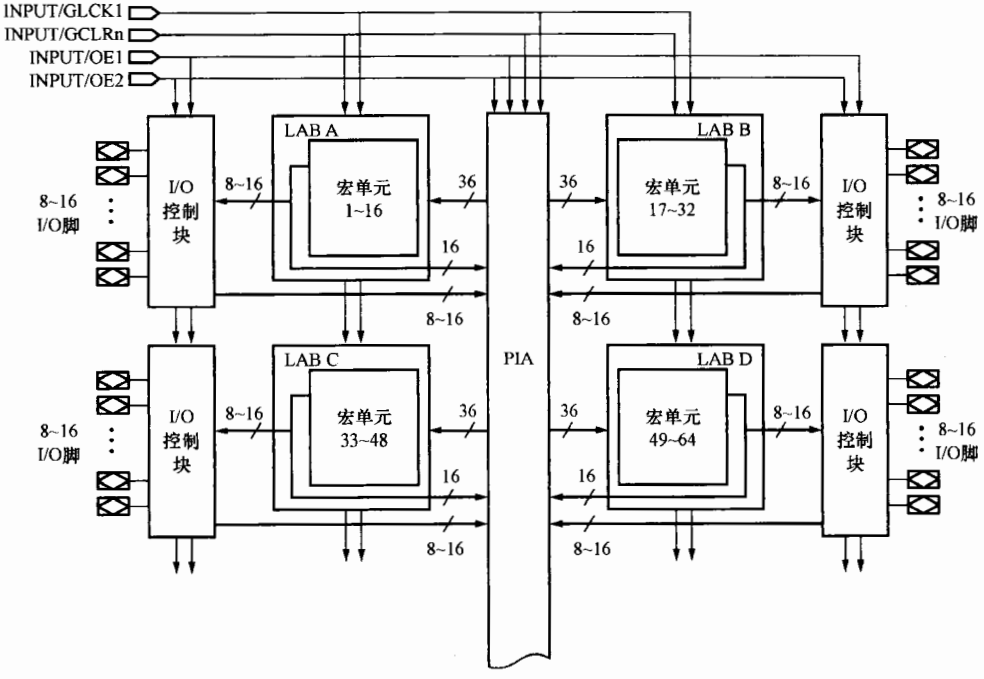


图 2-1　MAX7000 的结构

图 2-2 的左侧是乘积项阵列，实际上是一个“与或”阵列，每个交叉点都是一个可编程熔丝，如果导通就实现与逻辑，后面的乘积项选择矩阵是一个“或”阵列。两者一起完成组合逻辑。最右侧是一个可编程 D 触发器，它的时钟、清零输入都可以编程选择，可以使用专用的全局清零和全局时钟，也可以使用内部逻辑（乘积项阵列）产生的时钟和清零。如果不需要触发器，也可以将此触发器旁路，信号直接输给 PIA 或输出到 I/O 脚。

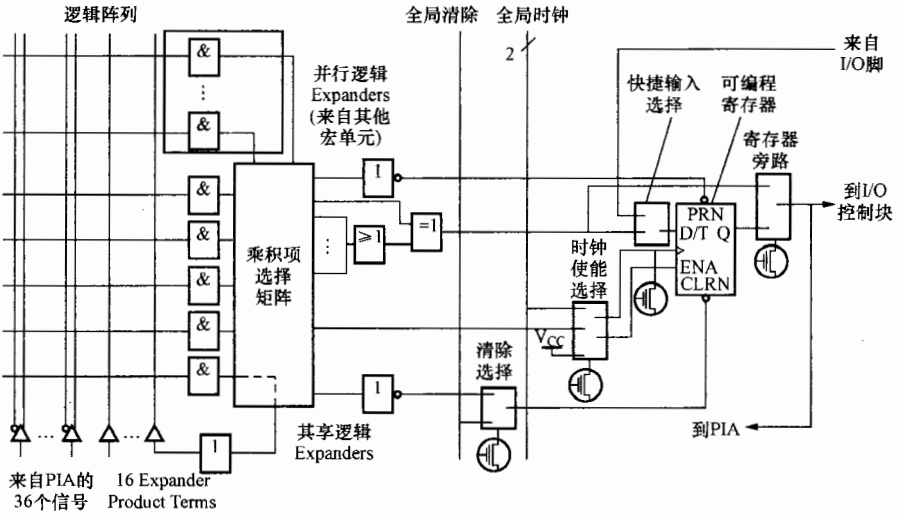


图 2-2　MAX7000 中宏单元的具体结构

2.3　基于乘积项的 CPLD 逻辑实现方式

我们通过一个简单的实例电路，来具体说明 CPLD 是如何利用以上结构实现逻辑的，电路如图 2-3 所示。

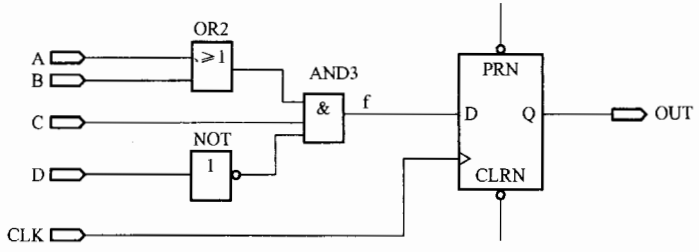


图 2-3　一个简单的实例电路

假设组合逻辑的输出（AND3 的输出）为 f，则 f=(A+B)*C*(~D)=A*C*(~D)+B*C*(~D)。

CPLD 将以如图 2-4 所示的电路来实现组合逻辑 f。

A、B、C、D 由 CPLD 芯片的引脚输入后进入可编程连线阵列（PIA），在内部会产生 A、~A、B、~B、C、~C、D、~D 共八个输出。每一个“*”点表示相连（可编程熔丝导通），所以得到：f=f1+f2=A*C*(~D)+B*C*(~D)。这样组合逻辑就实现了。

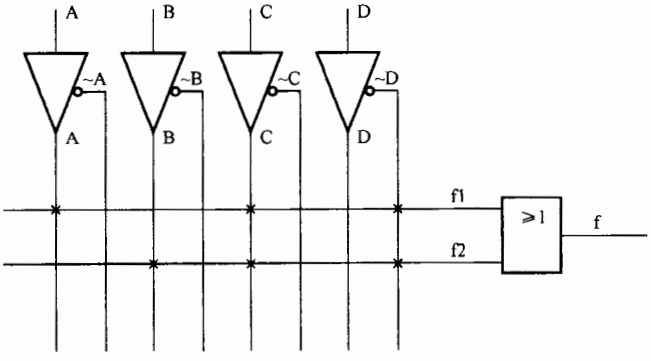


图 2-4　CPLD 实现组合逻辑

图 2-3 电路中 D 触发器的实现比较简单，直接利用宏单元中的可编程 D 触发器来实现。时钟信号 CLK 由 I/O 脚输入后进入芯片内部的全局时钟专用通道，直接连接到可编程触发器的时钟端。可编程触发器的输出与 I/O 脚相连，把结果输出到芯片引脚。这样 CPLD 就完成了图 2-3 所示电路的功能。以上这些步骤都是由设计软件自动完成的。

图 2-3 的例子比较简单，只需要一个宏单元就可以完成。但对于一个复杂的设计，一个宏单元是不能实现的，这时就需要通过并联扩展项和共享扩展项将多个宏单元相连，宏单元的输出也可以连接到可编程连线阵列，再作为另一个宏单元的输入。这样 CPLD 就可以实现更复杂逻辑。

这种基于乘积项的 CPLD 基本都是由 EEPROM 和 Flash 工艺制造的，上电后就可以工作，无需其他芯片配合。

2.4　基于查找表的 FPGA 原理与结构

采用查找表结构的 PLD 芯片称为 FPGA，如 Altera 的 ACEX、APEX 系列，Xilinx 的 Spartan、Virtex 系列等。

查找表（Look-Up-Table）简称为 LUT，LUT 本质上就是一个 RAM。目前 FPGA 中多使用 4 输入的 LUT，所以每一个 LUT 可以看成一个有 4 位地址线的 16X1 的 RAM。

当用户通过原理图或硬件描述语言描述了一个逻辑电路以后，FPGA 设计软件会自动计算出逻辑电路所有可能的结果，并把结果事先写入 RAM，这样，每输入一个信号进行逻辑运算就等于输入一个地址进行查表，找出地址对应的

内容，然后输出即可。

图 2-5 是 Altera 的 ACEX 1K 器件的结构。主要包括嵌入式阵列块（EAB）、逻辑阵列块（LAB）、快速通道（Fast Track）互连、I/O 单元（IOE）等。由一组 LE 组成一个 LAB，LAB 按行和列排成一个矩阵，并且在每一行中放置了一个嵌入式阵列块（EAB）。在器件内部，信号的互连及信号与器件引脚的连接由快速通道（Fast Track）提供，在每行（或每列）Fast Track 互连线的两端连接着若干个 I/O 单元（IOE）。Altera 其他系列，如 APEX 的结构与此基本相同。

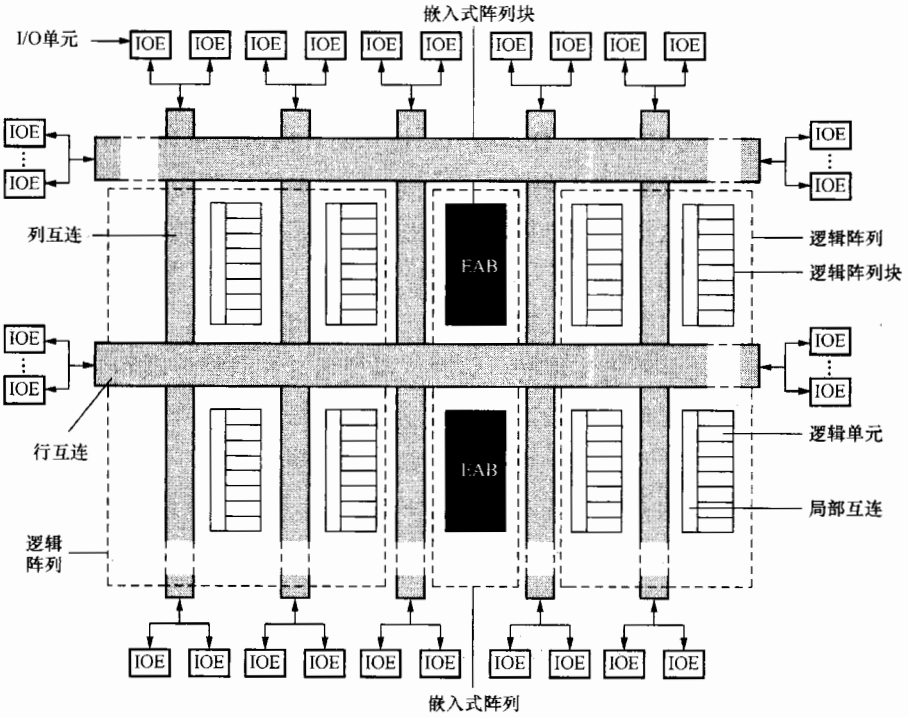


图 2-5　Altera 的 ACEX 1K 器件的结构

2.5　基于查找表结构的 FPGA 逻辑实现方式

以图 2-3 所示的逻辑电路为例，说明查找表结构 FPGA 逻辑实现的方式。

A、B、C、D 由 FPGA 芯片的引脚输入后进入可编程连线，然后作为地址线连接到 LUT，LUT 中已经事先写入了所有可能的逻辑结果，通过地址查找到

相应的数据然后输出，这样组合逻辑就实现了。该电路中 D 触发器是直接利用 LUT 后面 D 触发器来实现。时钟信号 CLK 由 I/O 脚输入后进入芯片内部的时钟专用通道，直接连接到触发器的时钟端。触发器的输出与 I/O 脚相连，把结果输出到芯片引脚。这样 FPGA 就完成了图 2-3 所示电路的功能。以上这些步骤都是由设计软件自动完成的。

这只是一个很简单的例子，只需要一个 LUT 加上一个触发器就可以完成。对于一个 LUT 无法完成的电路，就需要通过进位逻辑将多个单元相连，这样 FPGA 就可以实现更复杂的逻辑。

2.6　CPLD 与 FPGA 的区别

FPGA 和 CPLD 都是可编程的逻辑器件，都能设计组成数字逻辑系统，具有很多共同特点，但由于 CPLD 和 FPGA 结构上的差异,它们又各有其特点。

CPLD 和 FPGA 都是由逻辑单元、I/O 单元和互连三部分组成。其中 I/O 单元的功能基本一致，两者的逻辑单元和互连法则各不相同。另外，两类器件的编程工艺也有很大的差别，这些区别决定了应用范围的差别，下面从几个方面介绍两者的差别之处。

2.6.1　逻辑单元的区别

CPLD 中的逻辑单元是大单元，其输入变量数可以多达 20 个，通常称之为粗粒度结构。因为变量多，所以只能采用 PAL（即乘积项结构）。由于这样的单元功能强大，一般的逻辑在单元内均可实现，因而其互连关系简单，通过总线即可实现。电路的延时通常就是逻辑单元本身和总线的延时（在数 ns 到十几 ns 之间），但芯片内的触发器数量相对较少。CPLD 较适合控制器等逻辑型系统，这种系统的逻辑关系复杂，输入变量多，对触发器的需要量少。

FPGA 逻辑单元是小单元，每个单元有 1～2 个触发器，其输入变量通常只有几个，因此采用 LUT。这样的工艺结构占用芯片面积小，速度高（延时只有 1～2ns），每块芯片上能集成的单元数多，但逻辑单元的功能弱，因此，也把 FPGA 称为细粒度结构。实现一个复杂的逻辑函数，需要用到多个逻辑单元，输入到输出的延时大，互连关系比较复杂。FPGA 较适合信号处理等数据型系统，这种系统的逻辑关系简单，输入变量少，对触发器的需要量多。

结论：CPLD 更适合完成各种算法和组合逻辑，FPGA 更适合于完成时序逻辑。换句话说，FPGA 更适合于触发器丰富的结构，而 CPLD 更适合于触发器有限而乘积项丰富的结构。

2.6.2 互连方式的区别

CPLD 逻辑单元大，单元数量少，互连使用的是总线，其互连特点是总线上任意一对输入与输出之间的延时相等，而且是可预测的。

FPGA 因逻辑单元小，单元数量多，所以互连关系复杂，使用的互连方式较多，主要有分段总线、长线和直连等方式。因此，FPGA 属于分段式布线结构。分段总线分布在各单元之间，通过配置将不同位置的单元连接起来，但速度慢。长线有水平长线和垂直长线两种，贯穿芯片内部，使用频率较高，速度快。直连是速度最快的一种互连方式，但只限于单元与其四周的 4 个单元之间。由于一对单元之间的互连路径可以有多种，其速度不同，传输延迟也不好确定。应用 FPGA 时，除了逻辑设计外还要进行延时设计，通常需经数次设计和仿真，才能找出最佳设计方案。

结论：CPLD 的连续式布线结构决定了它的时序延迟是均匀的和可预测的，而 FPGA 的分段式布线结构决定了其延迟的不可预测性。

2.6.3 编程方式的区别

在编程上 FPGA 比 CPLD 具有更大的灵活性。

CPLD 通过修改具有固定内连电路的逻辑功能来编程，FPGA 主要通过改变内部连线的布线来编程。

FPGA 可在逻辑门下编程，而 CPLD 是在逻辑块下编程。

2.6.4 编程方式及次数的区别

在编程方式上，CPLD 主要是基于 EPROM、EEPROM 和 FlashROM 存储器编程，编程次数可达 1 万次，优点是系统断电时编程信息也不丢失。CPLD 又可分为在编程器上编程和在系统编程两类。FPGA 大部分是基于 SRAM 编程，编程信息在系统断电时丢失，每次上电时，需从器件外部的存储器将编程数据重新写入 SRAM 中。其优点是可以编程任意次，可在工作中快速编程，从而实现板级和系统级的动态配置。

2.6.5　集成度的区别

FPGA 的集成度比 CPLD 高，具有更复杂的布线结构和逻辑实现。

2.6.6　使用方便性的区别

CPLD 比 FPGA 使用起来更方便。CPLD 的编程采用 EEPROM 或快闪（FastFlash）技术，无需外部存储器芯片，使用简单。而 FPGA 的编程信息需存放在外部存储器上，使用方法复杂。

2.6.7　工作速度的区别

CPLD 的速度比 FPGA 快，并且具有较好的时间可预测性。这是由于 FPGA 是门级编程，并且 CLB 之间采用分布式互联；而 CPLD 是逻辑块级编程，并且其逻辑块之间的互联是集总式的。

2.6.8　功耗的区别

一般情况下，CPLD 的功耗要比 FPGA 大，且集成度越高越明显。

2.6.9　保密性的区别

CPLD 的保密性要比 FPGA 好。

FPGA 是一种高密度的可编程逻辑器件，自从 Xilinx 公司 1985 年推出第一片 FPGA 以来，FPGA 的集成密度和性能提高很快，其集成密度最高达 500 万门/片以上，系统性能可达 200MHz。由于 FPGA 器件集成度高,方便易用，开发和上市周期短，在数字设计和电子生产中得到迅速普及和应用，并一度在高密度的可编程逻辑器件领域中独占鳌头。

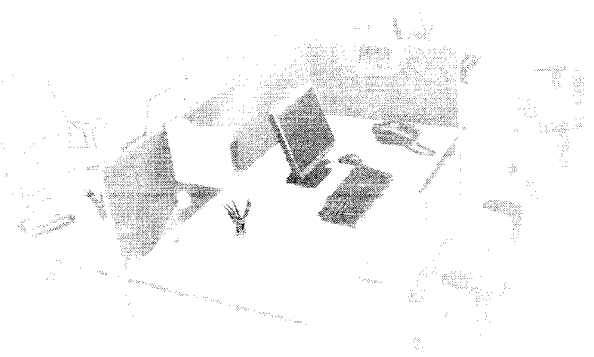
CPLD 是由 GAL 发展起来的，其主体结构仍是与或阵列，自从 20 世纪 90 年代初 Lattice 公司开发出高性能的具有在系统可编程 ISP（In System Programmable）功能的 CPLD 以来，CPLD 发展迅速。具有 ISP 功能的 CPLD 器件由于具有同 FPGA 器件相似的集成度和易用性，在速度上还有一定的优势，使其在可编程逻辑器件技术的竞争中与 FPGA 并驾齐驱，成为两支领导可编程器件技术发展的力量之一。

过去，由于受到 CPLD 密度的限制，对于较复杂的设计，只好转向 FPGA 和 ASIC（专用集成电路）。随着 CPLD 密度的提高，许多设计人员已经感受到

CPLD 容易使用、时序可预测和速度高等优点。现在，设计人员可以深切体会到密度高达数十万门的 CPLD 所带来的方便之处。

CPLD 结构在一个逻辑路径上采用 1～16 个乘积项，因而大型复杂设计的运行速度可以预测。因此，原有设计的运行可以预测，也很可靠，而且修改设计也很容易。CPLD 在本质上很灵活、时序简单、路由性能极好，用户可以改变他们的设计同时保持引脚输出不变。与 FPGA 相比，CPLD 的 I/O 更多，尺寸更小。

尽管 CPLD 与 FPGA 存在着一些差别，但开发它们的过程，所使用的工具软件、编程语言几乎是完全相同的，因此，学会了开发 CPLD，也就学会了开发 FPGA。



第3章

Xilinx公司的XC9500系列CPLD

Xilinx公司成立于1984年，总部位于美国加州的圣荷塞。Xilinx公司是领先的可编程逻辑解决方案公司，其解决方案包括：高级集成电路、软件设计工具、以核心形式提供的预先设定系统功能，以及独一无二的现场工程技术支持。

Xilinx公司的XC9500系列CPLD是目前市场占有率较高的产品，它具有先进的在系统编程及测试能力，编程/擦除次数大于10000次。该系列的所有器件都支持IEEE1149.1（JTAG）边界扫描。

XC9500系列CPLD的资源范围为800~6400个逻辑门（见表3-1）。XC9500系列CPLD采用多种封装形式，图3-1为84引脚的PLCC封装图。表3-2所列为各种封装选项和器件引脚数。

表3-1　　XC9500系列CPLD的资源范围

器件型号	XC9536	XC9572	XC95108	XC95144	XC95216	XC95288
宏单元	36	72	108	144	216	288
逻辑门	800	1600	2400	3200	4800	6400
寄存器	36	72	108	144	216	288

表3-2　　XC9500系列CPLD的各种封装选项和器件引脚数

器件型号 / 引脚封装	XC9536	XC9572	XC95108	XC95144	XC95216	XC95288
44脚VQFP	34					
44脚PLCC	34	34				
48脚CSP	34					

续表

器件型号 / 引脚封装	XC9536	XC9572	XC95108	XC95144	XC95216	XC95288
84 脚 PLCC		69	69			
100 脚 TQFP		72	81	81		
100 脚 PQFP		72	81	81		
160 脚 PQFP			108	133	133	
208 脚 HQFP					166	168
352 脚 BGA					166	192

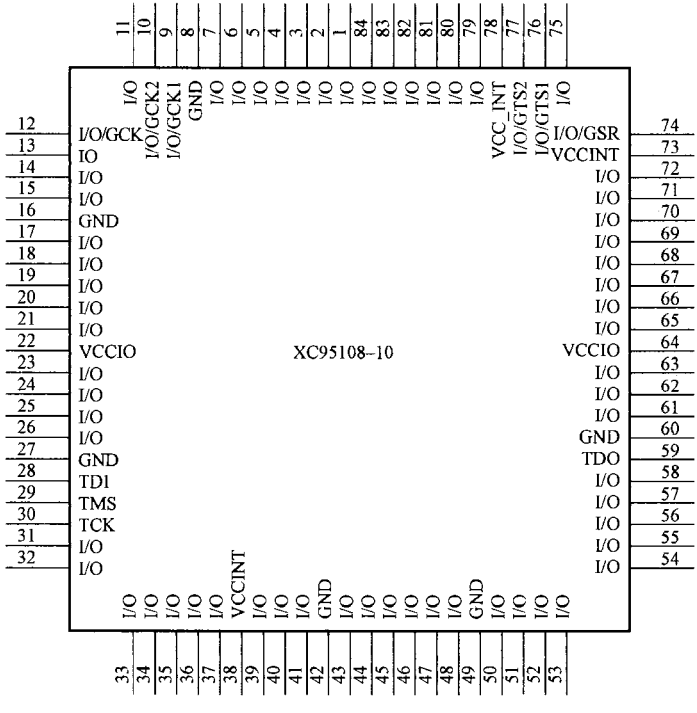


图 3-1　84 引脚的 PLCC 封装图

3.1　XC9500 系列 CPLD 结构及特性简介

图 3-2 为 XC9500 系列 CPLD 结构图，包括多个功能模块（FB）和 I/O 模块（IOB），并通过 FastCONNECT（快速连接）开关矩阵实现内部连接。IOB

提供器件输入和输出的缓冲。每个 FB 提供 36 个输入和 18 个输出的可编程逻辑。FastCONNECT 开关矩阵将所有 FB 输出和输入信号连接到 FB 输入。对于每一个 FB，12～18 个输出（取决于封装引脚数）和相关的输出使能信号直接驱动 IOB。

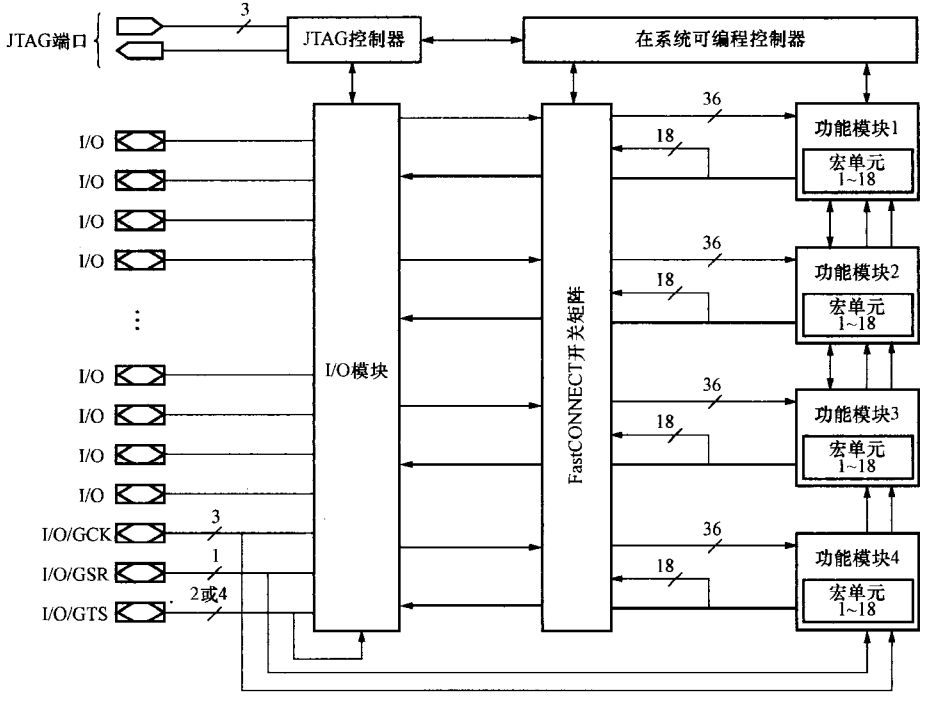


图 3-2　XC9500 系列 CPLD 结构图

3.1.1　功能模块（FB）

图 3-3 为功能模块框图，每个功能模块都包含 18 个独立的宏单元，都可通过组合或寄存器实现功能。FB 还可接收全局时钟、输出使能和设置/复位信号。FB 产生 18 个输出，用于驱动 FastCONNECT 开关矩阵。这 18 个输出和它们对应的输出使能信号也驱动 IOB。

FB 内的逻辑通过乘积表达式的集合来实现。36 个输入提供 72 个“真”与“互补”信号，输入到可编程“与”阵列来构成 90 个乘积项。这些乘积项的任意一个或全部都可通过乘积项定位器定位到每个宏单元。

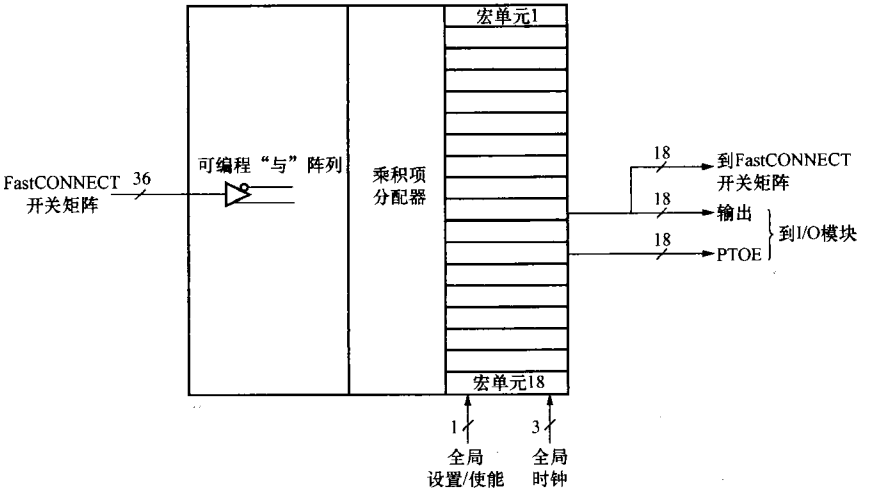


图 3-3　功能模块框图

3.1.2　宏单元

XC9500 系列 CPLD 的宏单元都可独立配置为通过组合或寄存器实现功能。宏单元和相关的 FB 逻辑如图 3-4 所示。

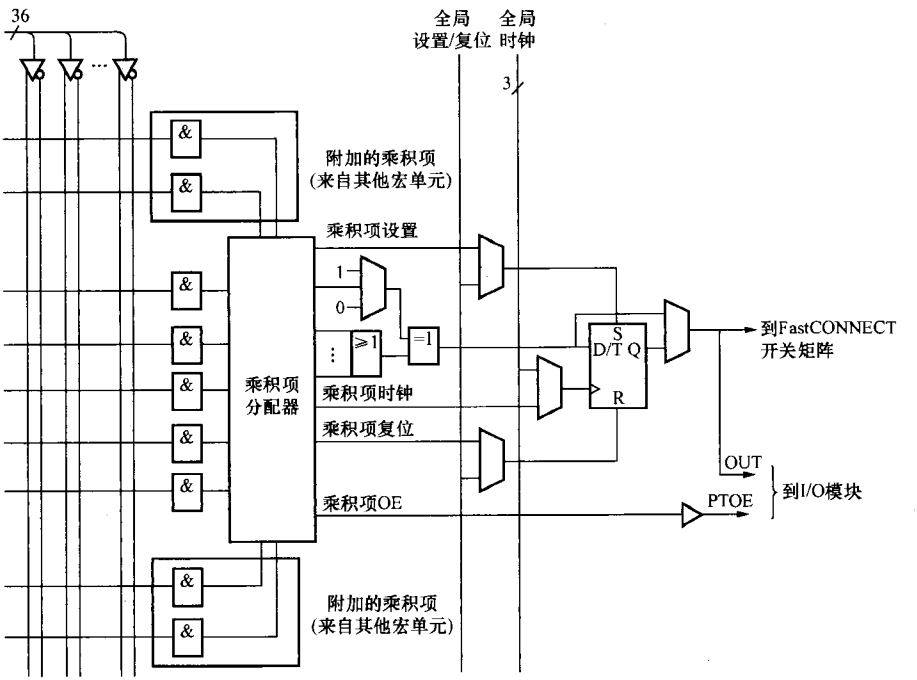


图 3-4　宏单元和相关的 FB 逻辑

“与”数组中的 5 个直接乘积项用于主要的数据输入（输入到“或”和“异或”门），以实现组合功能，或者作为控制输入信号（包括时钟、设置/复位和输出使能信号）。与每个宏单元相关的乘积项分配器用于选择如何使用这 5 个直接乘积项。

宏单元寄存器可配置为 D 型或 T 型触发器，或者被旁路，以实现组合操作。每个寄存器都支持异步设置和复位操作。上电时，所有用户寄存器都初始化为用户预定义的状态（如果未定义，则默认为 0）。

所有的全局控制信号都可用于单个宏单元，包括时钟、设置/复位和输出使能信号。如图 3-5 所示，宏单元寄存器时钟来自 3 个全局时钟之一或乘积项时钟。GCK 引脚的“真”和“补”极性可在器件内部使用。它还提供 GSR 输入，使用户寄存器可以设置成用户定义的状态。

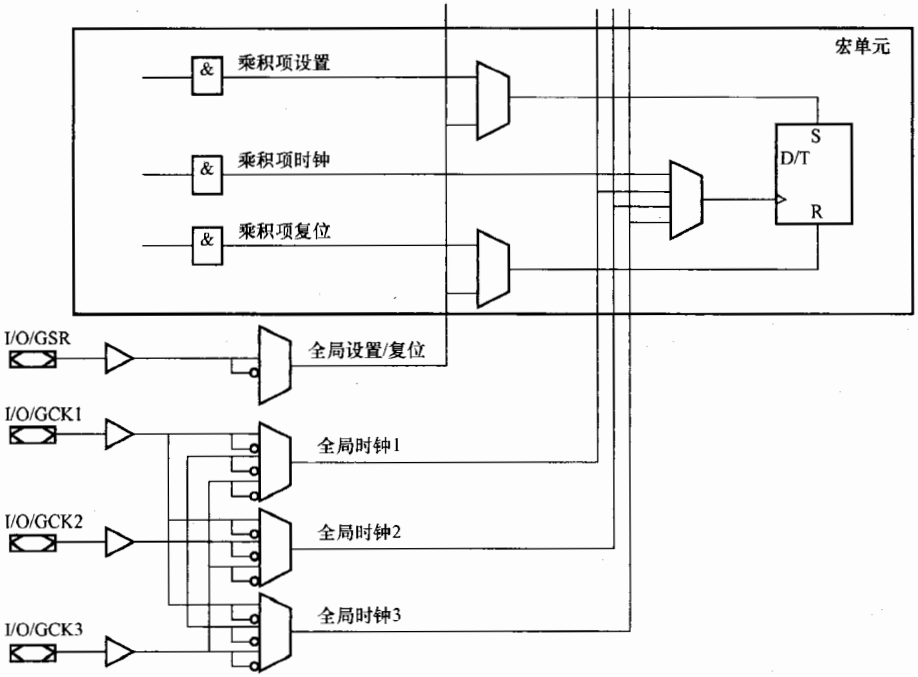


图 3-5　宏单元的时钟及设置/复位

3.1.3　乘积项分配器

乘积项分配器控制如何将 5 个直接乘积项分配到每个宏单元。例如，所有 5 个直接乘积项可驱动如图 3-6 所示的“或”功能。

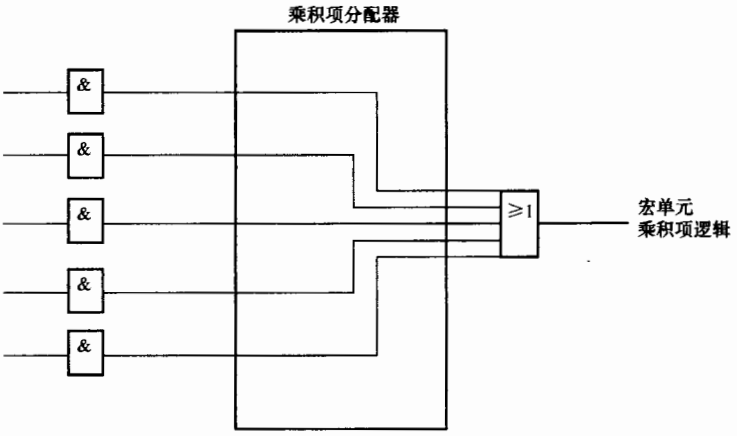


图 3-6　使用直接乘积项的宏单元逻辑

乘积项分配器可在 FB 内重新分配其他乘积项，这样可提高宏单元在 5 个直接乘积项之外的逻辑性能。在 FB 内，任何一个需要额外乘积项的宏单元都可访问其他宏单元中未使用的乘积项。一个宏单元最多可有 15 个乘积项，但单个宏单元的延迟时间增加很少。

3.1.4　FastCONNECT 开关矩阵

FastCONNECT 开关矩阵将信号连接到 FB 输入端（见图 3-7），所有 IOB 输出（对应于用户引脚输入）和所有 FB 输出驱动 FastCONNECT 矩阵。可选择上述任意输出（一个 FB 的最大扇入限值为 36）通过用户编程以相同的延迟来驱动每个 FB。

FastCONNECT 开关矩阵可以在驱动目标 FB 之前，将多个内部连接组合成单个“线与”输出。这样提供了额外的逻辑能力，并且在没有任何额外时序延迟的情况下，提高了目标 FB 的有效逻辑扇入。该特性只适合于 FB 输出端的内部连接，它由开发软件自动调用。

3.1.5　I/O 模块

内部逻辑和器件用户 I/O 引脚之间通过 I/O 模块（IOB）接口。每个 IOB 都包括一个输入缓冲器、输出驱动器、输出使能选择和用户可编程接地控制，如图 3-8 所示。

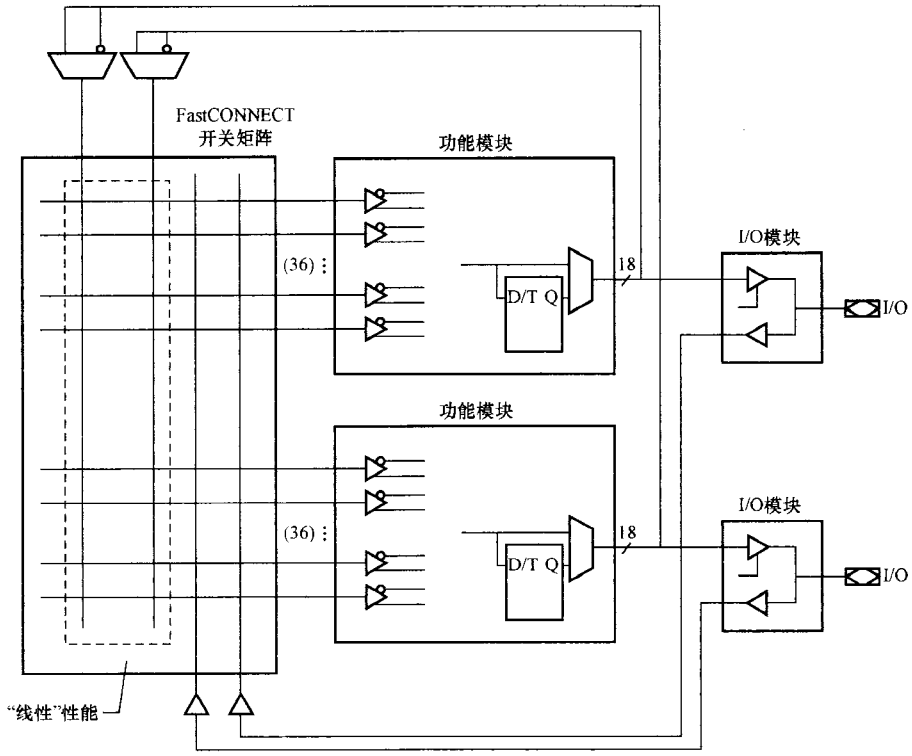


图 3-7　FastCONNECT 开关矩阵

输入缓冲器兼容标准 5V CMOS、5V TTL 和 3.3V 信号电平。输入缓冲器使用内部 5V 电源（V_{CCINT}）以确保内部门槛保持恒定，不随 V_{CCIO} 的电压变化。输出使能可由下列 4 个选项之一产生：一个宏单元的乘积项信号、任意全局输出使能 OE 信号的任意一个（总是为“1”或总是为“0”）。输出使能信号对 144 个宏单元的器件有 2 个全局输出使能，对 180 个或更多宏单元的器件有 4 个全局输出使能。任意一个全局 3 态控制（GTS）引脚的两个极性可以在全局内被利用。

每个输出都由独立的斜率控制。输出边沿斜度可以通过编程降低，以减少系统噪声。每个 IOB 都提供用户可编程接地脚性能，这使器件的 I/O 引脚可以配置到额外的地。通过将可编程接地脚和外部地相连接，众多输出同时开/关所产生的系统噪声因此减小。

当器件不处于正常用户操作中时，连接到每个器件 I/O 引脚的控制上拉电阻（典型值 10kΩ）可防止引脚处于悬浮状态。该电阻在器件编程模式、上电和擦除芯片时有效。在正常操作时无效。

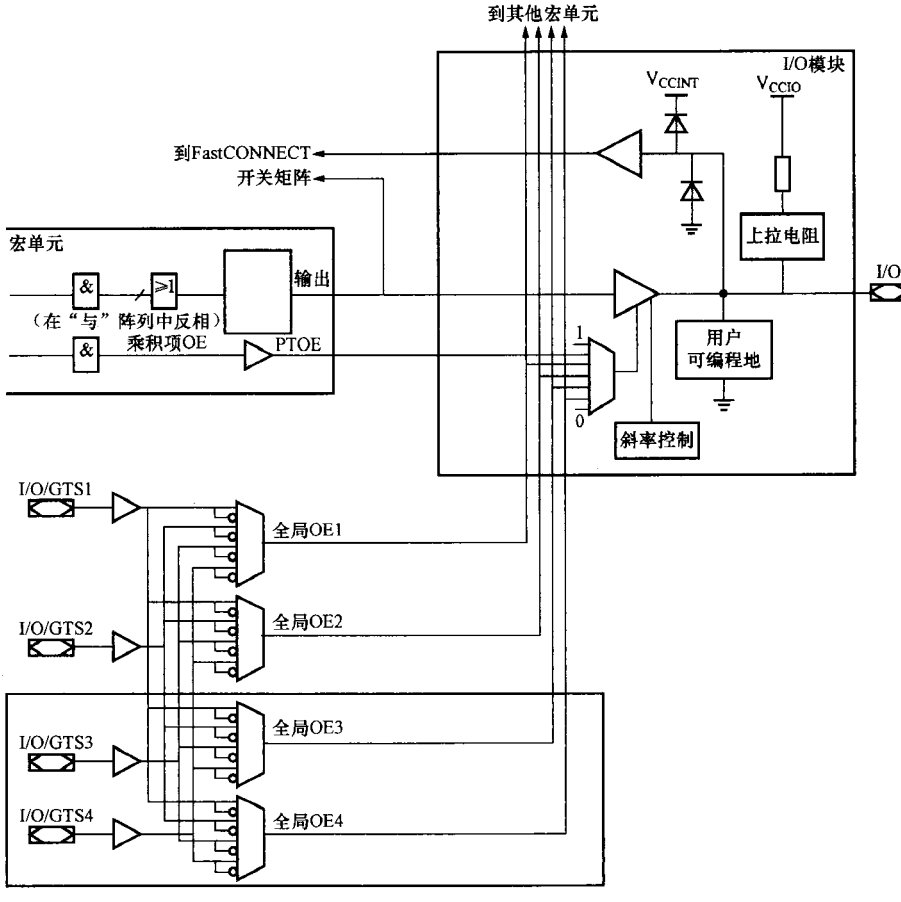


图 3-8　I/O 模块

输出驱动器具有 24mA 的驱动能力。通过将器件输出电源（V_{CCIO}）连接到 5V 或 3.3V，器件内的所有输出驱动器可配置为 5V TTL 电平或 3.3V 电平。

3.1.6　其他特性

1. 引脚锁定功能

当设计变更时，锁定用户定义的引脚分配取决于器件的结构是否能适应意外的变更。XC9500 系列 CPLD 的结构特性增强了这一适应能力。XC9500 的结构在 FastCONNECT 开关矩阵内提供了最大数目的路由，并集成了一个灵活的功能模块来实现可用乘积项的分配。

2. 在系统编程及耐久性

XC9500 系列 CPLD 通过标准 4 脚 JTAG 协议实现在系统编程。在系统编程时由 IOB 电阻上拉为高电平。如果在该时间内一个特殊信号必须保持低电平，那么可以在该引脚增加一个下拉电阻。所有的 XC9500 系列 CPLD 都可承受最少 10000 次在系统编程/擦除周期。

3. 保密设计

XC9500 系列 CPLD 集成了先进的数据保密特性，可完全防止数据在未经授权的情况下被读出，或因为疏忽对器件进行擦除/重新编程。

4. 低功耗模式

XC9500 系列 CPLD 器件的每个宏单元都提供低功耗模式。该特性使器件可以显著地降低功耗。用户可将任意一个宏单元编程为低功耗模式。对性能有严格要求的应用部分可保持标准功耗模式，而其他部分可以编程为低功耗模式，以降低整个系统的功率损耗。宏单元编程为低功耗模式时，会增加引脚间组合延迟时间和寄存器建立时间。乘积项时钟到输出和乘积项输出使能的延迟不受功耗模式设定的影响。

5. 上电特性

XC9500 系列 CPLD 器件在所有操作条件下都可良好工作。在上电时，XC9500 的内部电路使器件保持静止状态，直到 V_{CCINT} 处于安全的电压时（约为 3.8V）为止。在此期间，所有器件引脚和 JTAG 引脚都被禁止。当电源电压到达安全值时，所有用户寄存器开始初始化（XC9536~XC95144 在 100μs 以内，XC95216 为 200μs，XC95288 为 300μs），器件立即进入可操作状态。

如果器件处于擦除状态，器件输出保持禁止，IOB 上拉电阻使能。JTAG 引脚使能，以允许对器件编程。如果器件已被编程，则器件的输入和输出按照它们的配置状态执行正常操作。JTAG 引脚使能，则允许对器件进行擦除或边界扫描测试。

3.2　XC95108 CPLD 的主要特点

XC95108 是一款高性能的 CPLD，为通用的数字逻辑集成提供了先进的在系统编程及测试能力。它包含 6 个 36V18 功能模块，提供 2400 个可用门，传输延迟时间为 7.5ns。图 3-9 为 XC95108 的总体结构。

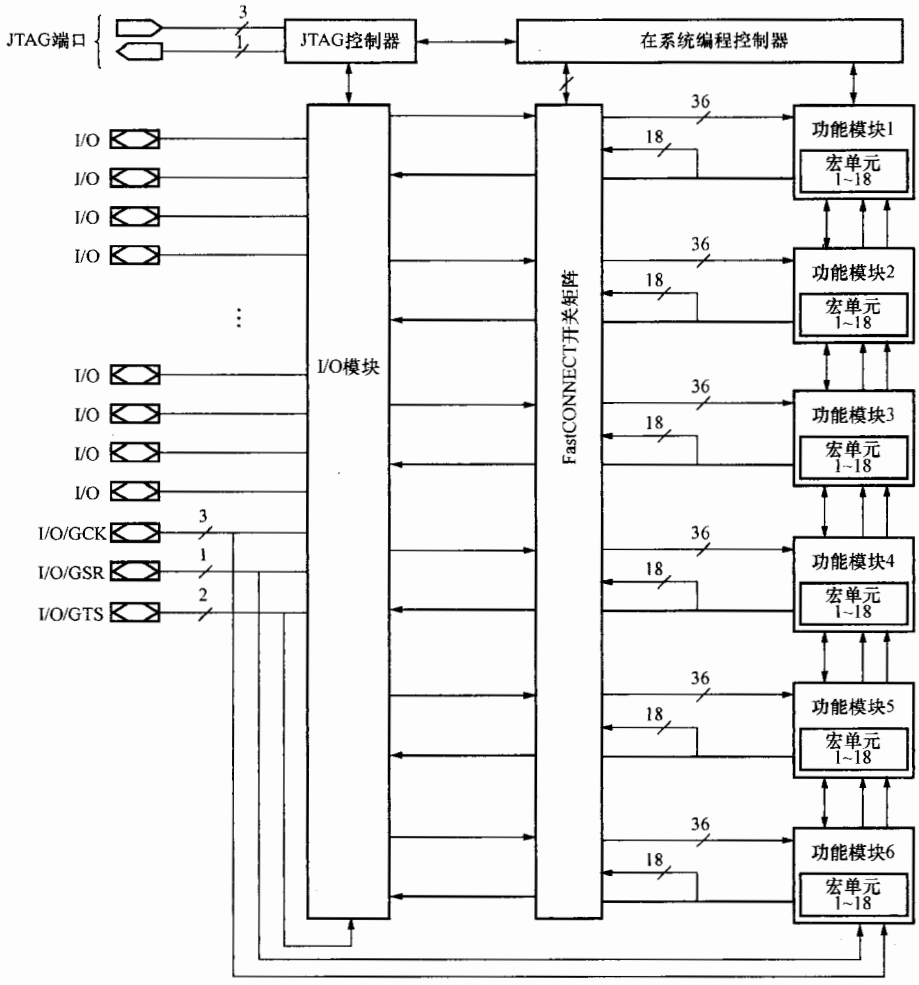


图 3-9　XC95108 的总体结构

XC95108 主要特点有：

（1）7.5ns 引脚间逻辑延迟。

（2）f_{CNT} 可达 125MHz。

（3）108 个宏单元带有 2400 个门。

（4）最多 108 个用户 I/O 口。

（5）5V 在系统编程。确保 10000 次编程/擦除周期。

（6）增强引脚锁定结构。

（7）灵活的 36V18 功能模块。

（8）IEEE Std 1149.1 边界扫描（JTAG）支持。

（9）每个宏单元都有可编程低功耗模式。

（10）单个输出的斜率控制。

（11）用户可编程接地引脚。

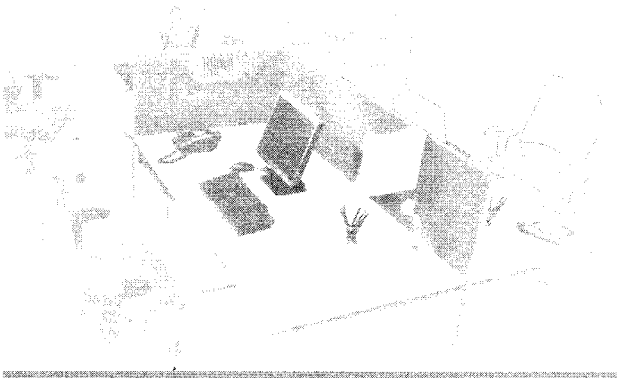
（12）扩展模式的保密特性，用于设计的保护。

（13）24mA 输出驱动能力。

（14）相容 3.3V 或 5V 的 I/O 口。

（15）先进的 CMOS 高速 Flash 工艺。

（16）支持对多个 XC95108 器件同时进行编程。



第4章

CPLD的设计流程与设计语言

CPLD的设计流程如图4-1所示，主要包括设计输入、综合、CPLD器件适配、仿真和编程下载等。实际上，这个设计流程同样也适用于FPGA。

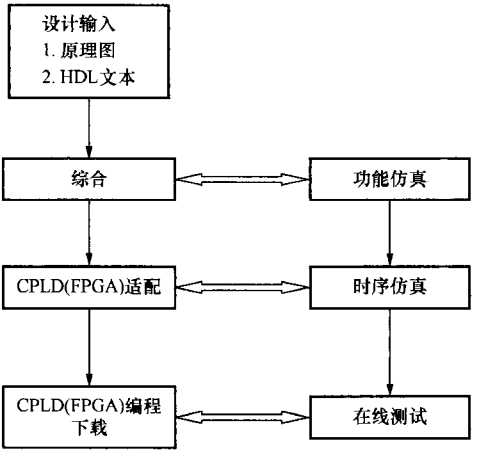


图4-1　可编程逻辑器件的设计流程

4.1 设 计 输 入

设计输入是将设计者将所设计的数字电路以开发软件要求的形式进行表达，然后输入到软件中去。设计输入有多种方式，最常用的是原理图输入方式和HDL文本输入方式两种。

原理图输入设计方式（图形方式）是设计规模较小的电路时经常采用的方法，这种方法直接把设计的电路用原理图方式表现出来（在开发软件中画原理图），具有直观、形象的优点，尤其对表现层次结构、模块化结构更为方便。

原理图输入设计方式要求设计工具提供必要的元件库，以供调用。它比较适合于描述连接关系和接口关系，而对于描述逻辑功能就不如文本方式方便了，同时，如果所设计系统的规模比较大，或设计软件不能提供设计者所需的库单元时，这种方法就会受到很大的限制。而且用原理图表示的设计，通用性、可移植性也弱一些，所以在现今的设计中，越来越多地采用基于硬件描述语言 HDL 的设计方式。

硬件描述语言 HDL（Hardware Description Language）是一种用文本形式来描述和设计电路的语言。设计者可利用 HDL 语言来描述自己的设计，然后利用 EDA 工具进行综合和仿真，最后变成某种目标文件，再用 ASIC（专用集成电路）或 CPLD（FPGA）具体实现。现在这种设计方法已被普遍采用。

用 HDL 文本来描述设计，其逻辑描述能力强，但描述接口和连接关系则不如图形方式直观。流行的硬件描述语言有 Verilog HDL 和 VHDL 等，Verilog HDL 和 VHDL 的功能比较强，属于行为描述语言，能描述和仿真复杂的逻辑设计，它们都已被采用为国际标准，被绝大多数的 EDA 工具所支持。

对于比较复杂的可编程逻辑器件的设计，往往采用层次化的设计方法，即分模块、分层次地进行设计描述（称为“Top-down”设计）。描述器件总功能的模块放置在最上层，称为顶层设计（Top）；描述器件最基本功能的模块放置在最下层，称为底层设计（down）。顶层和底层之间的关系类似于软件中的主程序和子程序的关系。层次化设计的方法比较自由，可以在任何层次使用原理图或硬件描述语言进行描述。目前一般做法是：在顶层设计中，使用图形法表达连接关系和芯片内部逻辑到引脚的接口；在底层设计中，使用硬件描述语言描述各个模块的逻辑功能。当然在顶层设计中也可使用硬件描述语言来描述各个模块的逻辑功能及连接关系，这样就实现了完全文本设计，方便程序在各种不同器件之间的移植，但各个模块的连接关系看上去没有图形表达法来得清晰明了。

硬件描述语言的发展只有 20 年左右的历史。20 世纪 80 年代后，硬件描述语言向着标准化、集成化的方向发展。最终，VHDL 和 Verilog HDL 适应了这种趋势的要求，先后成为 IEEE 国际标准。

4.1.1 原理图设计方式

原理图（Schematic）是图形化的表达方式，使用元件符号和连线来描述设计。其特点是适合描述连接关系和接口关系，而描述逻辑功能则比较烦琐。原理图输入对用户来讲很直观，尤其对表现层次结构、模块化结构更方便。但它要求设计工具提供必要的元件库或逻辑宏单元。如果输入的是较为复杂的逻辑或者元件库中不存在的模型，采用原理图输入方式往往很不方便。因此原理图方式只适用于规模比较小、电路比较简单的数字逻辑系统。此外，原理图方式的设计可重用性、可移植性也较差。

4.1.2 VHDL 语言设计方式

VHDL（Very High Speed Integrated Circuit Hardware Description Language）即超高速集成电路硬件描述语言。众所周知，美国国防部电子系统项目有着众多的承包商，但他们各自建立并使用自己的硬件描述语言，这就使得各公司之间的设计不能被重复利用，造成了信息交换和维护方面的困难。因此，20 世纪 80 年代初美国国防部制定了 VHDL，以作为各承包商提交复杂电路设计文档的一种标准方案。1987 年 12 月，VHDL 被正式接受为国际标准，编号为 IEEE Std1076—1987，即 VHDL'87。1993 年被更新为 IEEE Std1164—1993，即 VHDL'93。目前 VHDL 已被广泛应用。

VHDL 具有以下主要特点：

（1）功能强大，描述力强。可用于门级、电路级甚至系统级的描述、仿真和设计。

（2）可移植性好。对于设计和仿真工具采用相同的描述，对于不同的平台也采用相同的描述。

（3）研制周期短，成本低。这主要是由于 VHDL 支持对大规模设计的分解和对已有设计的利用，因此加快了设计流程。

（4）可以延长设计的生命周期。因为 VHDL 的硬件描述与工艺技术无关，不会因工艺变化而使描述过时。

目前，在大规模复杂电路与系统的设计中，VHDL 等标准化硬件描述语言已经逐步取代门级描述、逻辑电路图和布尔方程等级别较低的硬件描述语言，从而成为主要的硬件描述工具之一。

4.1.3　Verilog HDL 语言设计方式

Verilog HDL 是目前应用最为广泛的硬件描述语言之一，它允许设计者用其来进行各种级别的逻辑设计，以及数字逻辑系统的仿真验证、时序分析和逻辑综合。

Verilog HDL 语言最初是于 1983 年由 Gateway Design Automation 公司为其模拟器产品开发的硬件建模语言，那时它只是一种专用语言。由于模拟仿真器产品的广泛使用，Verilog HDL 作为一种便于使用且实用的语言逐渐为众多设计者所接受，1990 年 Cadence 公司成立了 OVI（Open Verilog International）组织来负责促进 Verilog HDL 语言的发展。基于 Verilog HDL 语言的优越性，1995 年，Verilog HDL 语言成为 IEEE 标准，称为 IEEE1364—1995。2001 年，在原标准的基础上经过改进和补充，发布了 Verilog HDL IEEE1364—2001 新标准。

4.1.4　Verilog HDL 与 VHDL 的比较

在硬件描述语言 HDL 的设计方式中，Verilog HDL 和 VHDL 又各有自己的优势和特点。Verilog HDL 早在 1983 年就已推出，至今已有近 20 多年的历史，因而 Verilog HDL 拥有更广泛的设计群体，其设计资源也远比 VHDL 丰富。与 VHDL 相比，Verilog HDL 的最大优点是：它是一种比较容易掌握的硬件描述语言，只要有 C 语言的编程基础，通过一段时间的学习和实验操作，可以在较短的时间内掌握这种设计技术；而掌握 VHDL 设计技术则比较困难。这是因为 VHDL 不是很直观，需要有 Ada 编程基础，一般需要至少半年以上的专业培训及实验，才能掌握 VHDL 的基本设计技术。因此，Verilog HDL 作为学习可编程逻辑器件的设计入门是再合适不过了，我们这里也是以 Verilog HDL 语言为主进行 CPLD 的入门学习及设计进阶的。

4.2　综　　合

综合（Synthesis）是一个很重要的步骤，它指的是将较高层次的设计描述自动转化为较低层次描述的过程。综合有下面几种形式：

（1）将算法表示、行为描述转换到寄存器传输级（RTL），即从行为描述到结构描述，称为行为综合。

（2）RTL 级描述转换到逻辑门级（可包括触发器），称为逻辑综合。

（3）将逻辑门表示转换到版图表示，或转换到PLD器件的配置网表表示，称为版图综合或结构综合。根据版图信息能够进行ASIC生产，有了配置网表可完成基于PLD器件的系统实现。

综合器就是能够自动实现上述转换的软件工具。或者说，综合器是能够将原理图或HDL语言表达和描述的电路功能转化为具体的电路结构网表的工具。

硬件综合器和软件程序编译器有着本质的区别。软件程序编译器是将C语言或汇编语言等编写的程序编译为0、1代码流，而硬件综合器则将用硬件描述语言编写的程序代码转化为具体的电路网表结构。

4.3 器 件 适 配

适配器（Fitter）有时也称为结构综合器，它的功能是将由综合器产生的网表文件配置于指定的目标器件中，并产生最终的可下载文件。如对CPLD器件而言，产生熔丝图文件，即JEDEC文件；对FPGA器件则产生Bitstream位流数据文件。

利用适配器可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。映射是把设计分为多个适合器件内部逻辑资源实现的逻辑小块的过程。

适配器会产生以下一些重要的文件：

（1）适配报告：包括芯片内部资源耗用情况，设计的布尔方程描述情况等。

（2）面向其他EDA工具的输出文件，如EDIF文件等。

（3）适配后的仿真模型，包括延时信息等，以便于进行精确的时序仿真。因为已经得到器件的实际硬件特性（如时延特性），所以仿真结果能精确地预测未来芯片的实际性能。如果仿真结果达不到设计要求，就需要修改源代码或选择不同速度的器件，直至满足设计要求。

（4）器件编程文件：如用于CPLD编程的JEDEC、POF等格式的文件；用于FPGA配置的SOF、JAM、BIT等格式的文件。

4.4 仿 真

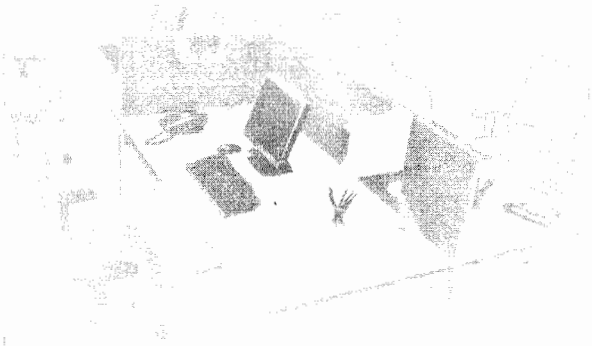
仿真（Simulation）是对所设计电路进行功能验证的过程。用户可以在设

计过程中对整个系统和各个模块进行仿真，即在计算机上用软件验证功能是否正确，各部分的时序配合是否准确。如果有问题可以随时进行修改，从而避免了逻辑错误。高级的仿真软件还可以对整个系统的设计性能进行检验。规模越大的设计，越需要进行仿真。

仿真包括功能仿真和时序仿真。不考虑信号时延等因素的仿真，称为功能仿真，又叫前仿真；时序仿真又称后仿真，它是在选择了具体器件并完成了布局布线后进行的包含定时关系的仿真。由于不同器件的内部时延不一样，不同的布局、布线方案也给延时造成了很大的影响，因此，在设计实现后，要对网络和逻辑块进行时延仿真，分析定时关系，评估设计性能。

4.5 编 程 下 载

把适配后生成的编程文件装入到 PLD 器件中的过程称为下载。通常将基于 EEPROM 工艺的非易失结构 CPLD 器件的下载称为编程（Program），而将基于 SRAM 工艺结构的 FPGA 器件的下载称为配置（Configure）。编程需要满足一定的条件，如编程电压、编程时序和编程算法等。一般有两种常用的编程方式：在系统编程 ISP（In System Programmable）和专用的编程器编程。现在的 PLD 器件一般都支持在系统编程，因此在设计数字系统和 PCB 时，应预留器件的下载接口。



第5章

CPLD学习开发器材介绍

学习 CPLD 的设计，必须要进行大量的实验，俗话说“实践出真知”，否则只能是“纸上谈兵”。这里我们使用下面介绍的实验器材进行 CPLD 的学习设计。

（1）Xilinx 的集成开发软件 Xilinx ISE。

（2）Keil C51 Windows 集成开发环境。

（3）MCU & CPLD DEMO 综合试验板。

（4）Xilinx 并口下载器，用于 CPLD 的程序下载。

（5）USB 下载器，用于单片机的程序下载。

（6）9V 高稳定专用稳压电源。

下面对这些实验工具及器材进行介绍。

5.1 Xilinx 的集成开发软件 Xilinx ISE

Xilinx ISE 是 Xilinx 公司推出的一个功能强大的 CPLD/FPGA 开发软件，具有从设计到仿真调试的全部功能。图 5-1 为 Xilinx ISE 的界面。

5.2 Keil C51 Windows 集成开发环境

Keil C51 是目前世界上最优秀、最强大的 51 单片机开发应用平台之一，它集编辑、编译、仿真于一体，支持汇编、PL/M 语言和 C 语言的程序设计，界面友好，易学易用。它内嵌的仿真调试软件可以让用户采用模拟仿真和实时在

线仿真两种方式对目标系统进行开发。软件仿真时，除了可以模拟单片机的 I/O 口、定时器、中断外，甚至可以仿真单片机的串行通信。图 5-2 为 Keil C51 的界面。

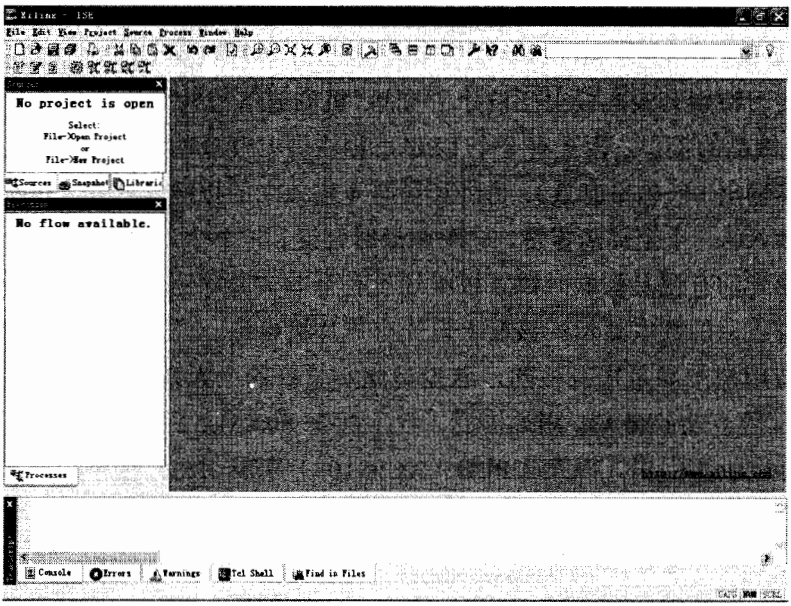


图 5-1　Xilinx ISE 的界面

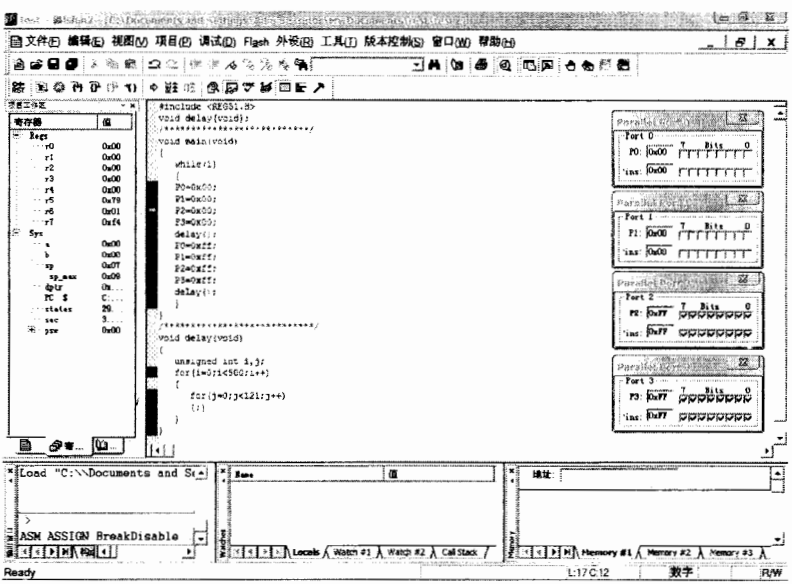


图 5-2　Keil C51 的界面

5.3 MCU & CPLD DEMO 综合试验板

MCU & CPLD DEMO 试验板为多功能的 51 单片机和 CPLD 开发试验板，对入门实习特别有效，板上已设计了与 PC 机的通信与电平转换电路及驱动 16×2 字符液晶的接口。板上有 8 个 LED 可独立做 CPLD 的输出实验，用发光二极管指示输出（低电平有效）。另外还设有 4 位独立的按键输入和 4 位独立的拨码开关输入。一个全局清零键、一个全局时钟键、一个全局输出使能键，CPLD 使用高稳定的有源晶振做时钟。板上还设有音响实验电路。8 位高亮度数码管可做多种用途的数字显示。该板上设计有单片机电路，对于学习设计较高级的智能化应用型产品是很有效的，这也是此实验板的一大特色。MCU & CPLD DEMO 试验板功能强大、用途广泛，板上标有 89X51/52 系列单片机引脚标准标识及标准引脚引出，以及 CPLD（XC95108）引脚引出，便于用户实验时识别及进行扩展使用。MCU & CPLD DEMO 试验板使用 9V 电源供电，板上带有 5V 及 3.3V 的可选稳压电路。

图 5-3 为 MCU & CPLD DEMO 试验板电路原理示意图。

板上资源简介如下：

（1）U1 为单片机 AT89S51，可进行单片机与 CPLD 的联合设计。

（2）子板上 U2 为 CPLD，我们使用 Xilinx 公司的 XC95108，烧写次数大于 10000 次，非常适合初学者学习实验。

（3）U3 为 24MHz 的高稳定有源晶振。

（4）U4 为可调三端稳压器，产生 5V 或 3.3V 电压供 CPLD 或单片机工作（通过 J11 跳针选择）。

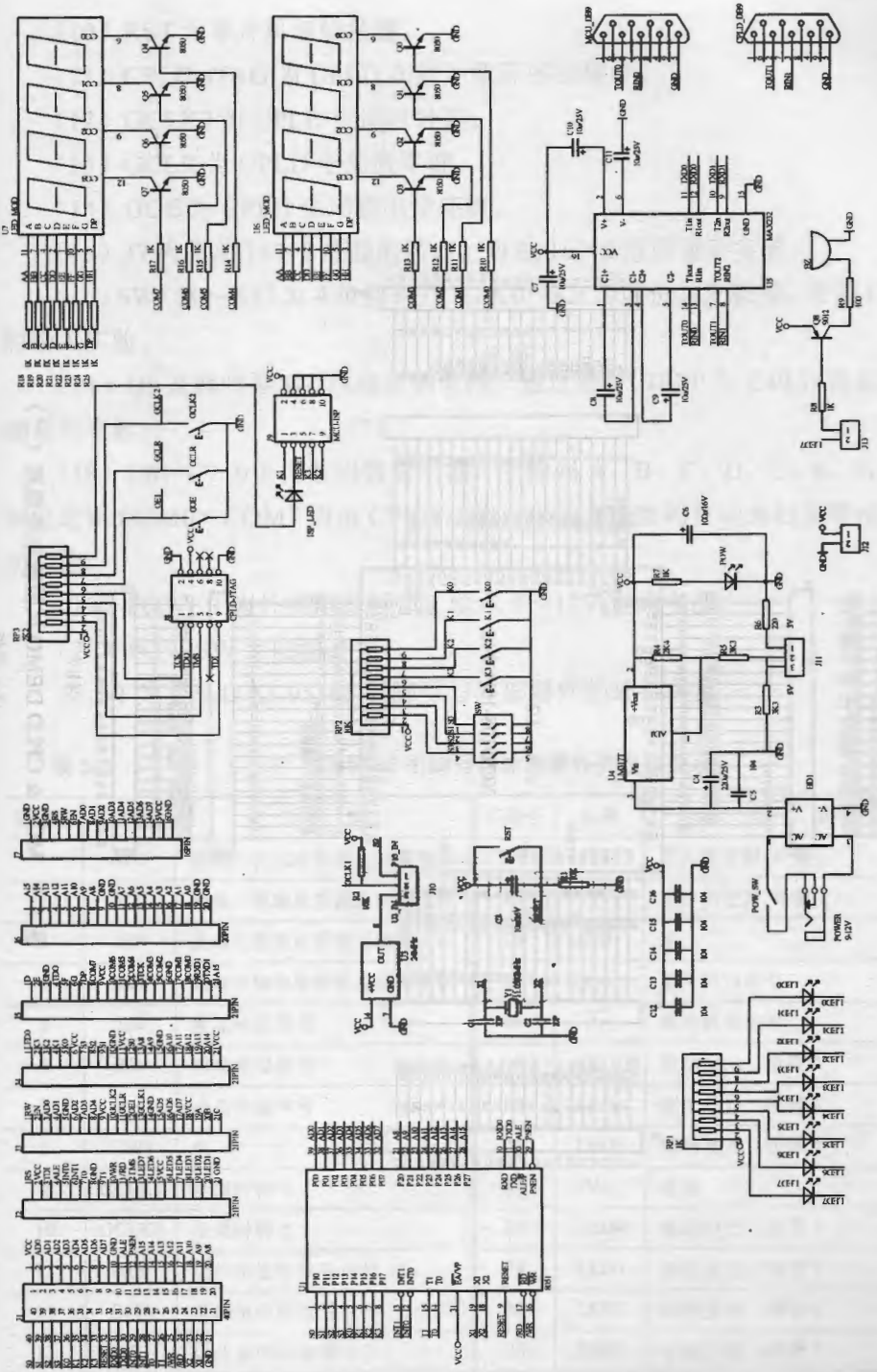
（5）U5 为 232 通信芯片，便于单片机或 CPLD 与 PC 机进行 RS232 通信实验。

（6）J1 为双排针，它将单片机的 40PIN 引出，便于单片机外扩其他器件。

（7）CPLD 子板上的 J2~J5 为单排针，它将 XC95108 的 84PIN 引出，便于实验或外扩其他器件。

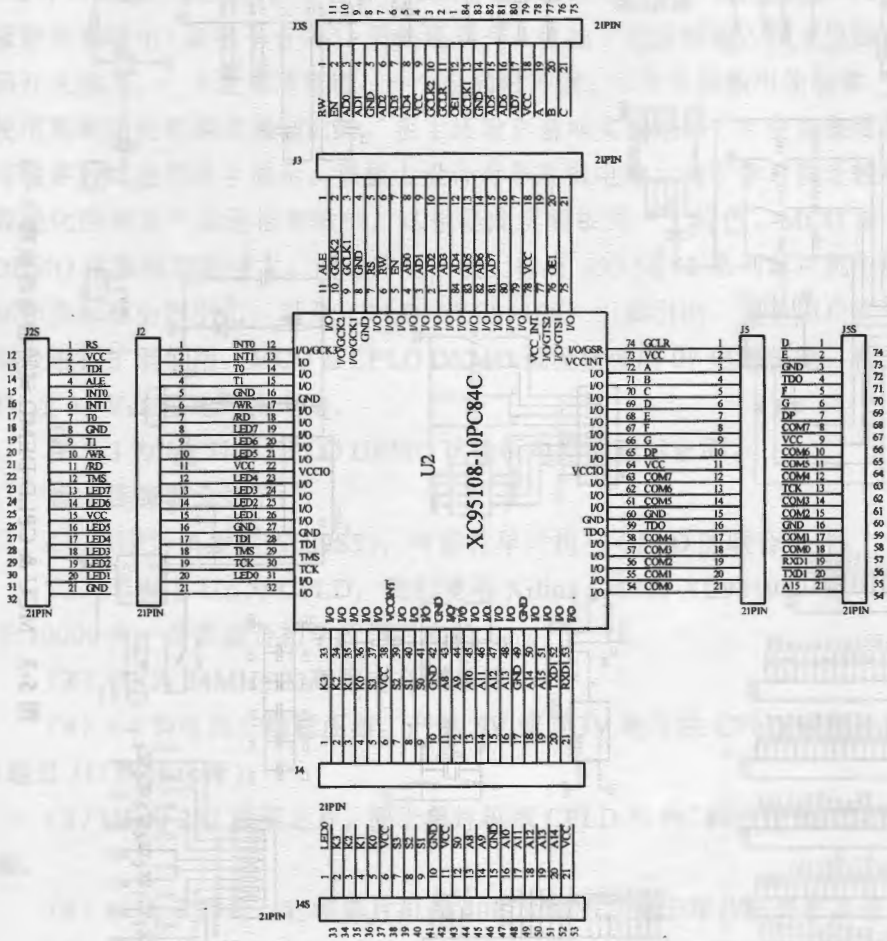
（8）LED0～LED7 为 8 个发光二极管，直接与 CPLD 连接，可作开关量输出的指示。

（9）MCU～ISP 为单片机在线下载程序的接口。



（a）母板

图 5-3　MCU & CPLD DEMO 试验板电路原理（一）



（b）

图 5-3 MCU & CPLD DEMO 试验板电路原理（二）

（b）子板

（10）RST 为单片机复位按键。

（11）CPLD-JTAG 为 CPLD 在线下载程序的接口。

（12）GCLK2 为 CPLD 全局时钟键。

（13）GCLR 为 CPLD 全局清零键。

（14）GOE 为 CPLD 全局输出使能键。

（15）J7 为驱动 16×2 液晶的接口，可做 16×2 液晶驱动实验。

（16）SW（S0～S3）为 4 位拨码开关，K0～K3 为四位独立按键，可做 CPLD 的输入实验。

（17）Q8 及蜂鸣器 BZ 组成音响电路，通过排针 BEEP 与 CPLD 连接，可做音响实验。

（18）U6、U7 为 8 位数码管显示器，字段码 A、B、C、D、E、F、G、DP 和位选码 COM0～COM7 均由 CPLD 送出，可做 8 位数码管动态扫描输出及驱动实验。

（19）POWER 为外接电源插口，输入 9～12V 直流电压。

（20）POW SW 为电源开关。

表 5-1 为 CPLD XC95108 引脚号与外部器件的连接关系。

表 5-1　　CPLD XC95108 引脚号与外部器件的连接关系

引脚号	名称	功　能	引脚号	名称	功　能
1	AD3	接单片机地址数据总线第 3 位	14	T0	单片机定时/计数 0
2	AD2	接单片机地址数据总线第 2 位	15	T1	单片机定时/计数 1
3	AD1	接单片机地址数据总线第 1 位	16	GND	地
4	AD0	接单片机地址数据总线第 0 位	17	$\overline{WR}$	单片机写信号
5	EN	液晶使能信号	18	$\overline{RD}$	单片机读信号
6	RW	液晶读写信号	19	LED7	驱动发光二极管 7
7	RS	液晶控制信号	20	LED6	驱动发光二极管 6
8	GND	地	21	LED5	驱动发光二极管 5
9	GCLK1	全局时钟 1	22	V_{CC}	电源
10	GCLK2	全局时钟 2	23	LED4	驱动发光二极管 4
11	ALE	单片机地址锁存信号	24	LED3	驱动发光二极管 3
12	INT0	单片机中断信号 0	25	LED2	驱动发光二极管 2
13	INT1	单片机中断信号 1	26	LED1	驱动发光二极管 1

续表

引脚号	名称	功　能	引脚号	名称	功　能
27	GND	地	56	COM2	数码管 2 位驱动
28	TDI	JTAG 数据输入	57	COM3	数码管 3 位驱动
29	TMS	JTAG 模式	58	COM4	数码管 4 位驱动
30	TCK	JTAG 时钟	59	TDO	JTAG 数据输出
31	LED0	驱动发光二极管 0	60	GND	地
32	—	—	61	COM5	数码管 5 位驱动
33	K3	按键输入 3	62	COM6	数码管 6 位驱动
34	K2	按键输入 2	63	COM7	数码管 7 位（最高位）驱动
35	K1	按键输入 1	64	V_{CC}	电源
36	K0	按键输入 0	65	DP	数码管段位
37	S3	拨码输入 3	66	G	数码管段位
38	V_{CC}	电源	67	F	数码管段位
39	S2	拨码输入 2	68	E	数码管段位
40	S1	拨码输入 1	69	D	数码管段位
41	S0	拨码输入 0	70	C	数码管段位
42	GND	地	71	B	数码管段位
43	A8	外部扩展地址总线第 8 位	72	A	数码管段位
44	A9	外部扩展地址总线第 9 位	73	V_{CC}	电源
45	A10	外部扩展地址总线第 10 位	74	GCLR	全局清零
46	A11	外部扩展地址总线第 11 位	75	—	—
47	A12	外部扩展地址总线第 12 位	76	OE1	输出使能 1
48	A13	外部扩展地址总线第 13 位	77	—	—
49	GND	地	78	V_{CC}	电源
50	A14	外部扩展地址总线第 14 位	79	—	—
51	A15	外部扩展地址总线第 15 位	80	—	—
52	TXD1	串口发送 1	81	AD7	接单片机地址数据总线第 7 位
53	RXD1	串口接收 1	82	AD6	接单片机地址数据总线第 6 位
54	COM0	数码管 0 位（最低位）驱动	83	AD5	接单片机地址数据总线第 5 位
55	COM1	数码管 1 位驱动	84	AD4	接单片机地址数据总线第 4 位

图 5-4 为 MCU & CPLD DEMO 试验板的元件排列布局。

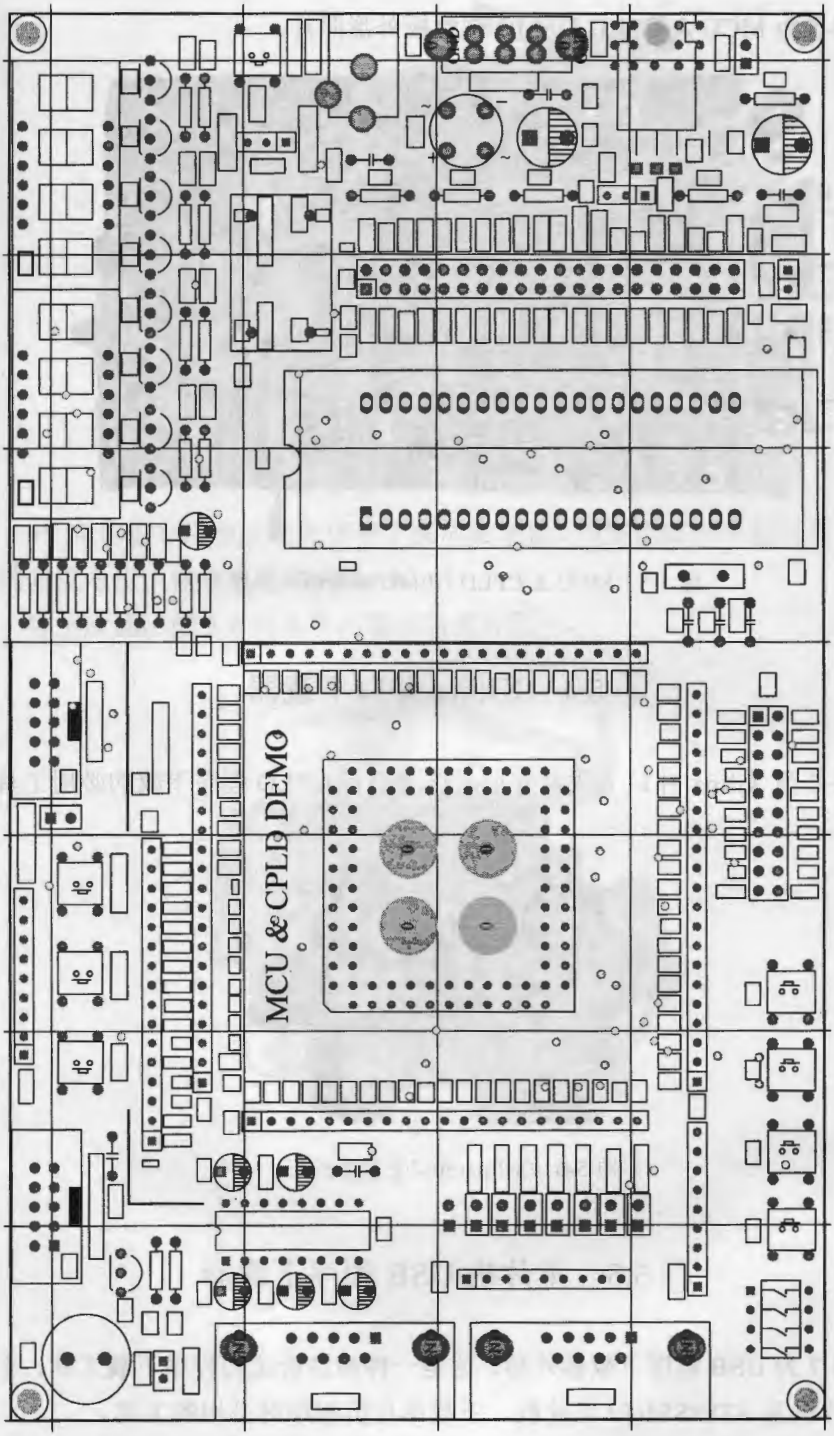


图 5-4　MCU & CPLD DEMO 试验板的元件排列布局

图 5-5 为 MCU & CPLD DEMO 试验板外形照片。

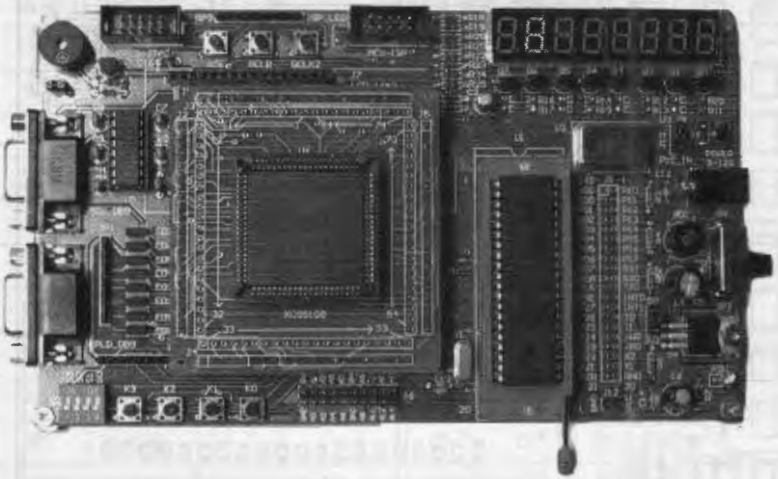


图 5-5　MCU & CPLD DEMO 试验板外形照片

5.4　Xilinx 并口下载器

图 5-6 为 Xilinx 并口下载器外形，这是进行 CPLD 程序下载的必用工具。

图 5-6　Xilinx 并口下载器外形

5.5　单片机 USB 程序下载器

图 5-7 为 USB 程序下载器外形，这是一种高性价比的程序下载工具。支持 AVR 单片机及 AT89S51/52 单片机。下载单片机程序时必用的工具。

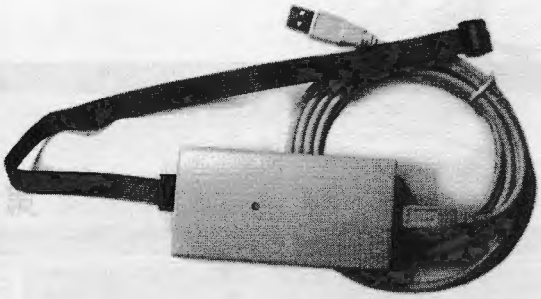


图 5-7　USB 程序下载器外形

5.6　9V 高稳定专用稳压电源

9V 高稳定专用稳压电源使用了集成稳压器，可输出纹波系数很小、非常纯净的直流电压，输出电流达 800mA。

综上所述，图 5-8 为本学习器材的套件照片。

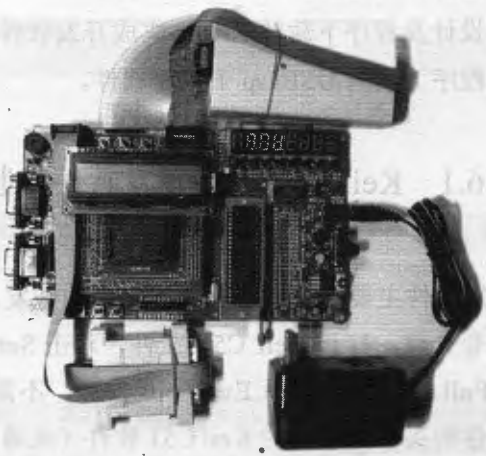
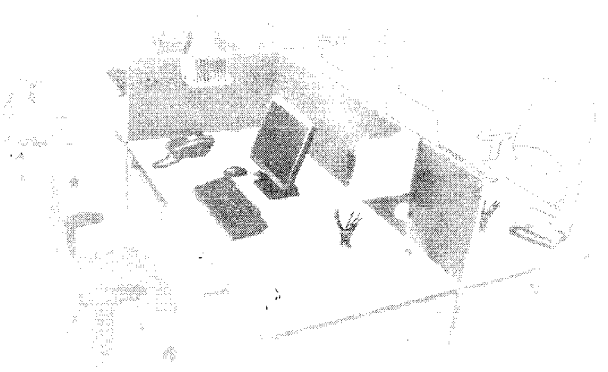


图 5-8　学习套件照片



第6章

开发软件 Keil C51 及 Xilinx ISE 的安装

我们在学习开发中需要用到的软件有：

（1）用于单片机设计的 Keil C51 集成开发软件。

（2）用于 CPLD 设计及程序下载的 Xilinx 集成开发软件 Xilinx ISE9.1i。

（3）用于单片机程序下载的 USBasp 下载器软件。

6.1 Keil C51 集成开发软件安装

Keil C51 集成开发软件主要用于 MCS-51 单片机的开发。

在电脑中放入配套光盘，打开 Keil C51 文件，双击 Setup.exe 进行安装，在提示选择 Eval 或 Full 方式时，选择 Eval 方式安装，不需注册码，但有 2K 大小的代码限制。如你购买了完全版的 Keil C51 软件（或通过其他途径得到），则选择 Full 方式安装，代码量无限制。安装结束后，如果您想在中文环境使用，可安装 Keil C51 汉化软件，双击 KEIL707 应用程序进行安装，安装完成后在桌面上会出现 Keil uVision2（汉化版）图标，双击该图标便可启动程序，启动后的界面如图 6-1 所示。

Keil C51 集成开发环境主要由菜单栏（见图 6-2）、工具栏（见图 6-3）、源文件编辑窗口、工程窗口和输出窗口 5 部分组成。

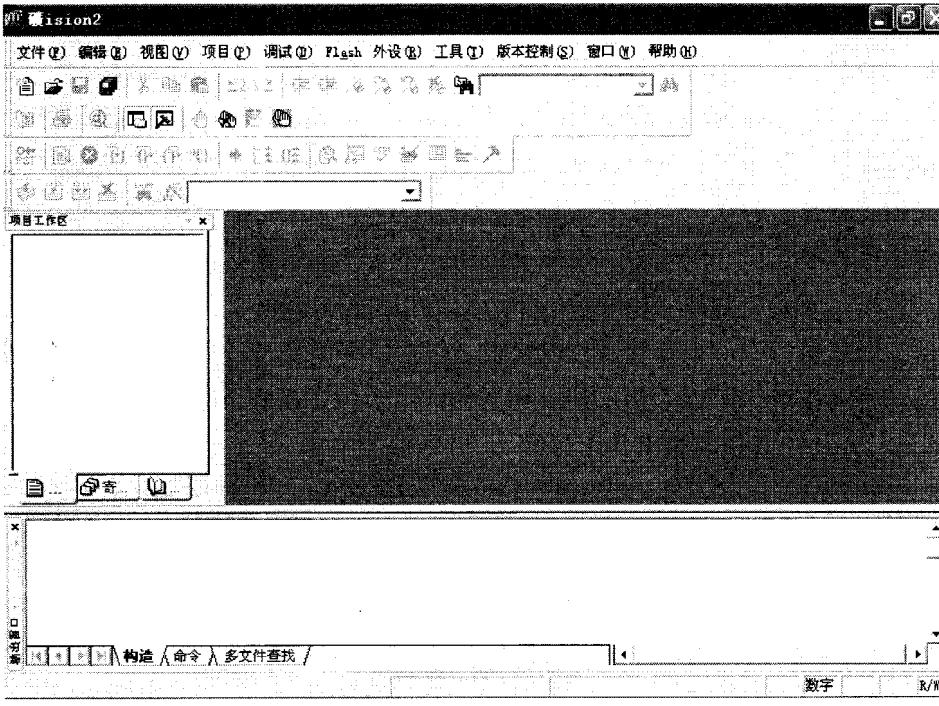


图 6-1　Keil C51 启动后界面

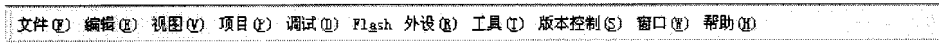


图 6-2　Keil C51 菜单栏

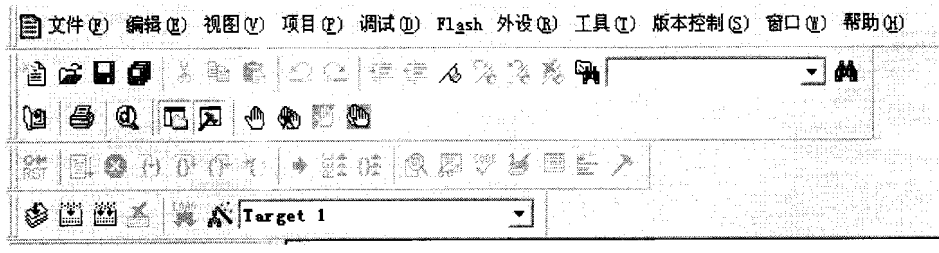


图 6-3　Keil C51 工具栏

工具栏为一组快捷工具图标，主要包括基本文件工具栏、建造工具栏和调试工具栏，基本文件工具栏包括新建、打开、拷贝、粘贴等基本操作。建造工具栏主要包括文件编译、目标文件编译连接、所有目标文件编译连接、目标选项和一个目标选择窗口。调试工具栏位于最后，主要包括一些仿真调试源程序的基本操作，如单步、复位、全速运行等。在工具栏下面，默认有三个窗口。左边的工程窗口包含一个工程的目标（target）、组（group）和项目文件。右边为源文件编辑窗口，编辑窗口实质上就是一个文件编辑器，我们可以在这里对源文件进行编辑、修改、粘贴等。下边的为输出窗口，源文件编译之后的结果显示在输出窗口中，会出现通过或错误（包括错误类型及行号）的提示。如果通过，则会生成“HEX”格式的目标文件，用于仿真或烧录芯片。

MCS-51 单片机软件 Keil C51 开发步骤为：

（1）建立一个工程项目，选择芯片，确定选项。

（2）建立汇编源文件或 C 源文件。

（3）用项目管理器生成各种应用文件。

（4）检查并修改源文件中的错误。

（5）编译连接通过后进行软件模拟仿真或硬件在线仿真。

（6）编程操作。

（7）应用。

如果读者对 Keil C51 集成开发环境的使用不熟悉，可以先阅读关于使用 Keil C51 集成开发环境的相关书籍。

6.2 Xilinx 集成开发软件 Xilinx ISE9.1i 的安装

由于 Xilinx ISE 集成开发软件在使用时对电脑的要求比较高，因此建议电脑的配置要高一些，内存也要足够大，起码也要达 512M 以上。

在电脑中放入 Xilinx ISE9.1i 安装光盘，双击 **setup.exe**，进行安装，这时弹出一个安装的欢迎界面，我们点击 Next 进入安装（见图 6-4）。当弹出安装许可协议时，点击接受（见图 6-5～图 6-7）。安装目录可使用默认方式将其安装在 C 盘中（见图 6-8）。安装选项也可以使用默认方式（见图 6-9～图 6-11）。当出现图 6-12 的界面后，选择“Install”进入安装，图 6-13 为正在拷贝文件，直到安装结束（见图 6-14）。

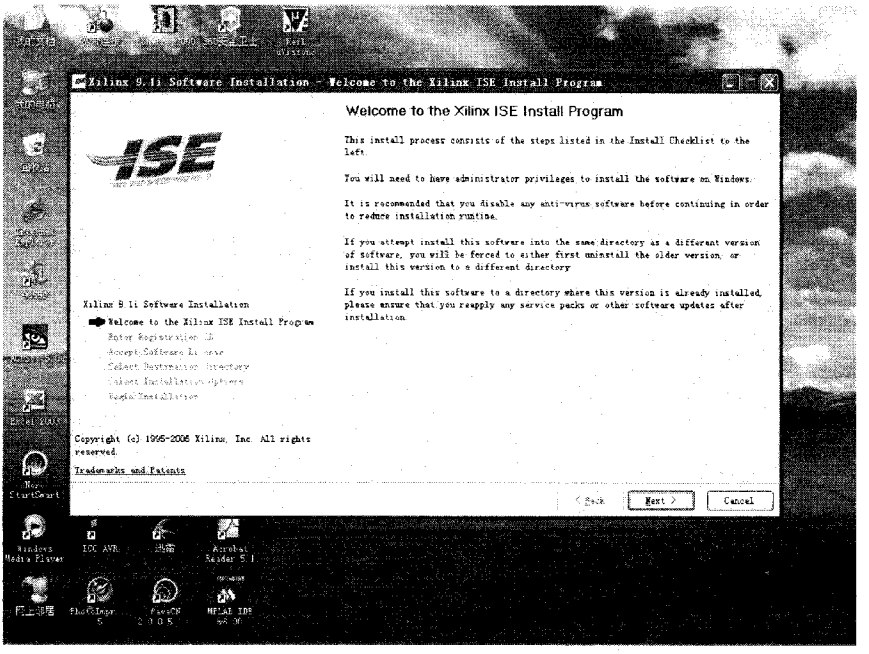


图 6-4　点击 Next 进入安装

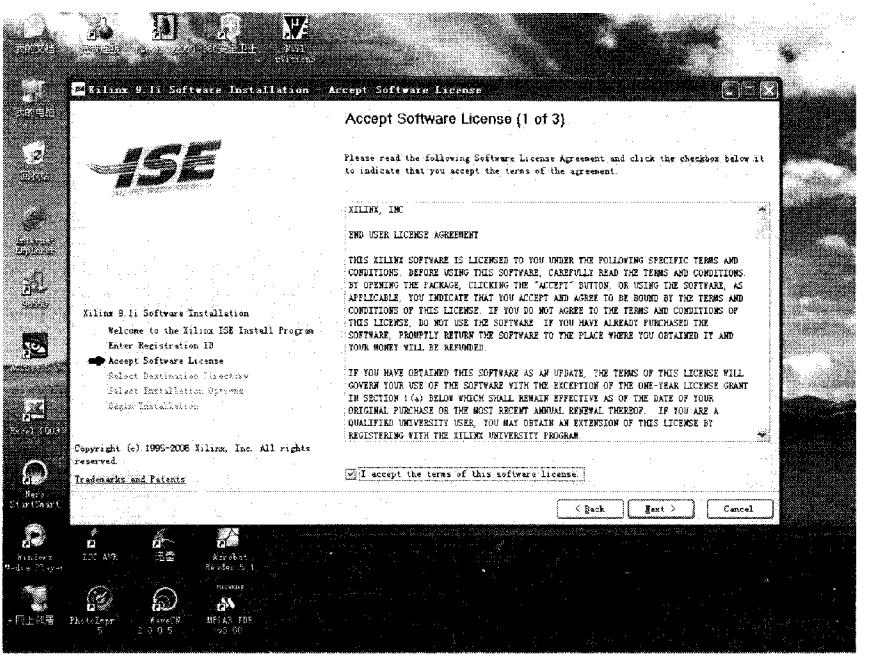


图 6-5　接受安装许可协议（一）

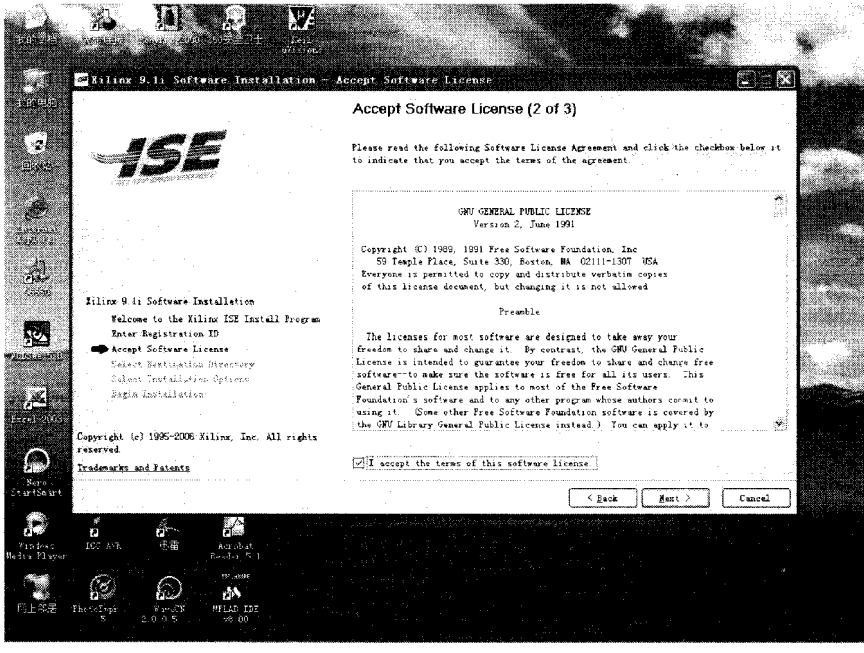


图 6-6　接受安装许可协议（二）

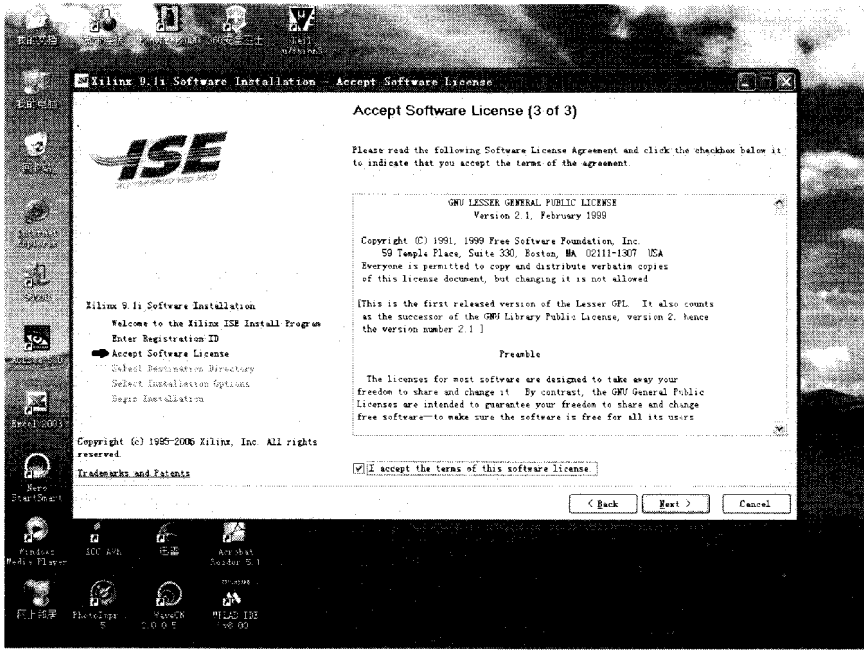


图 6-7　接受安装许可协议（三）

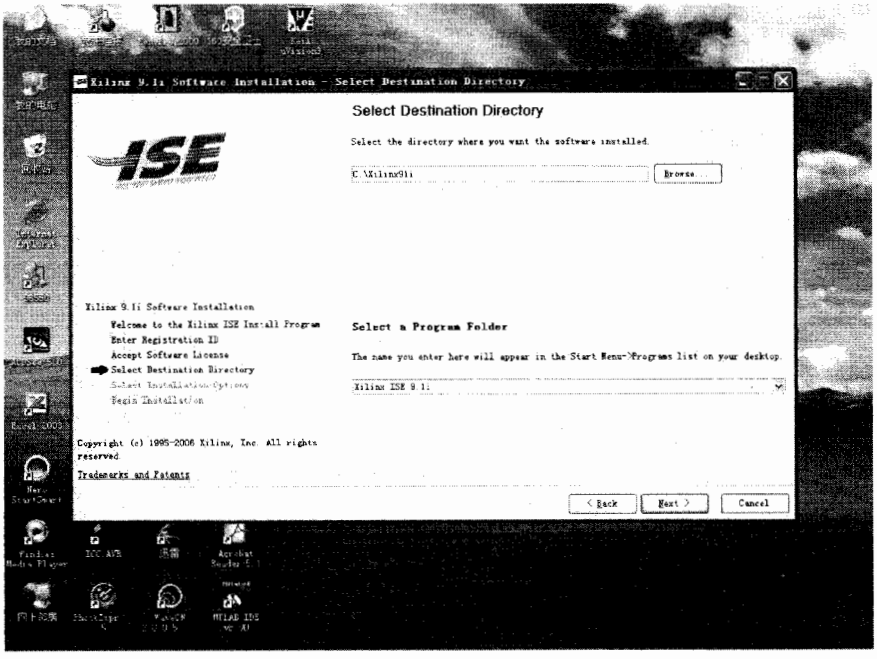


图 6-8　使用默认方式将其安装在 C 盘中

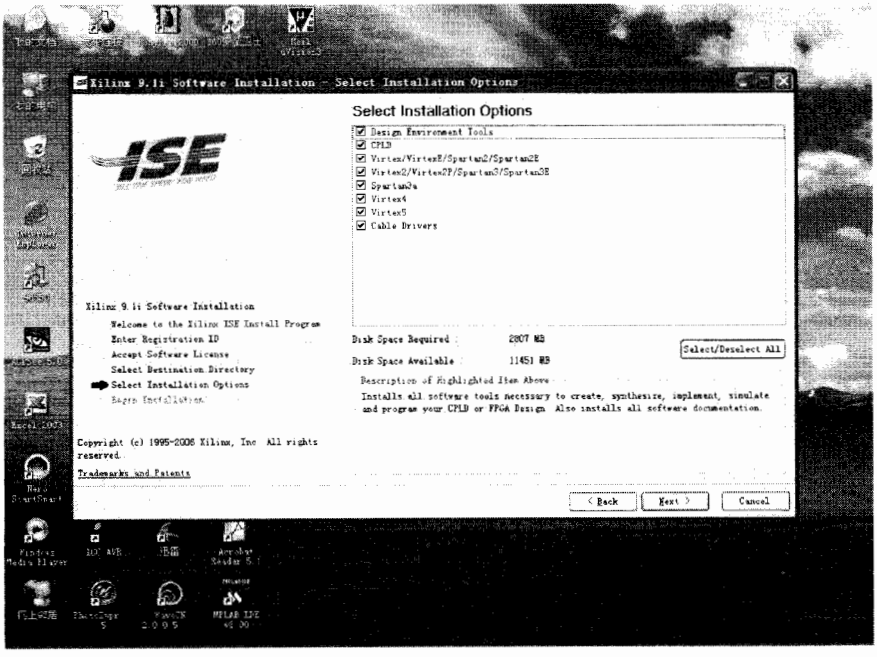


图 6-9　安装选项使用默认方式（一）

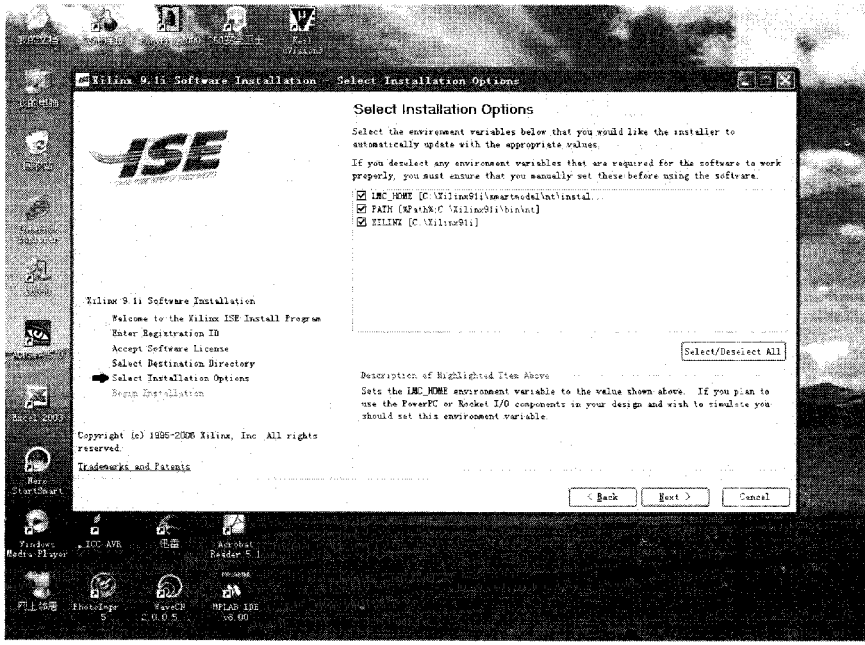


图 6-10　安装选项使用默认方式（二）

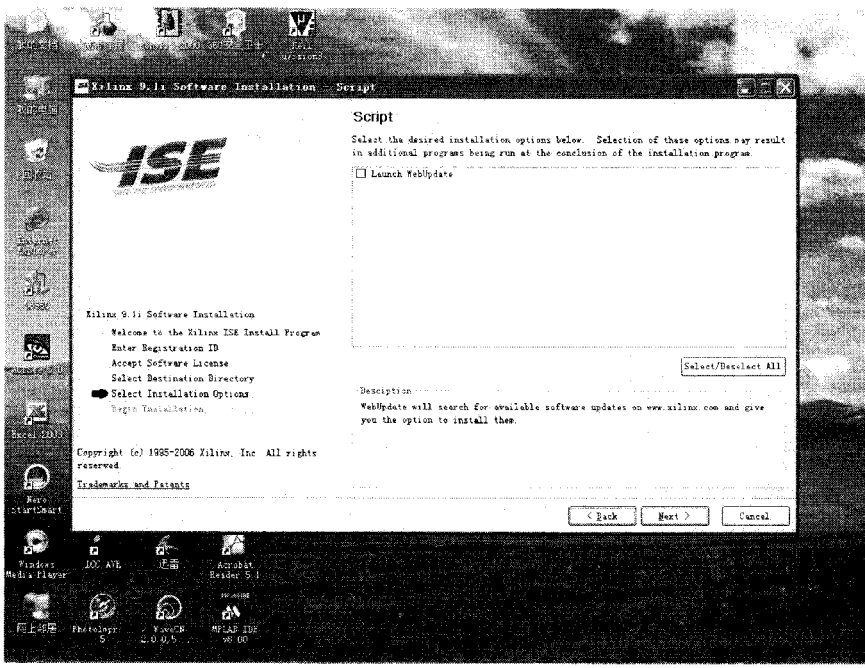


图 6-11　安装选项使用默认方式（三）

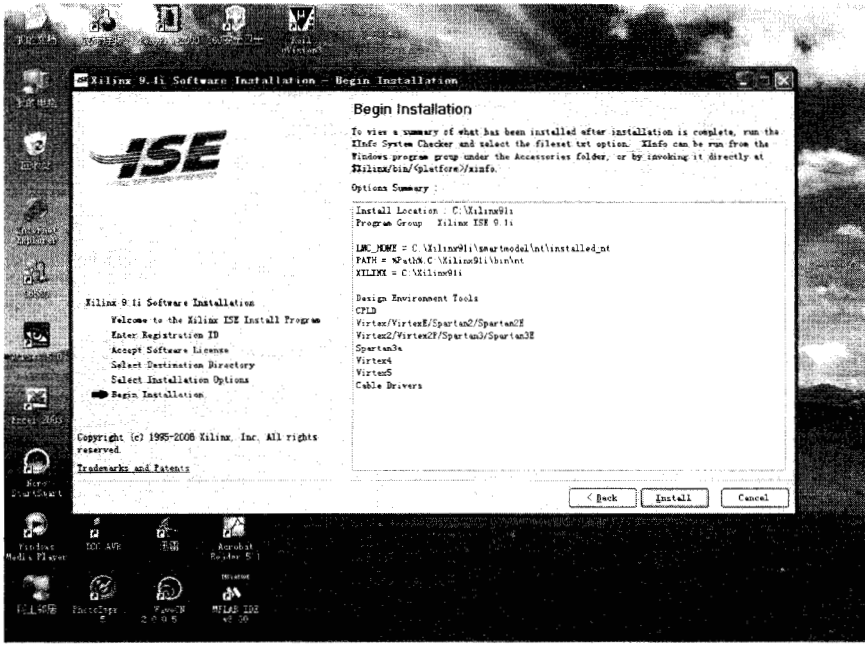


图 6-12　选择“Install”进入安装

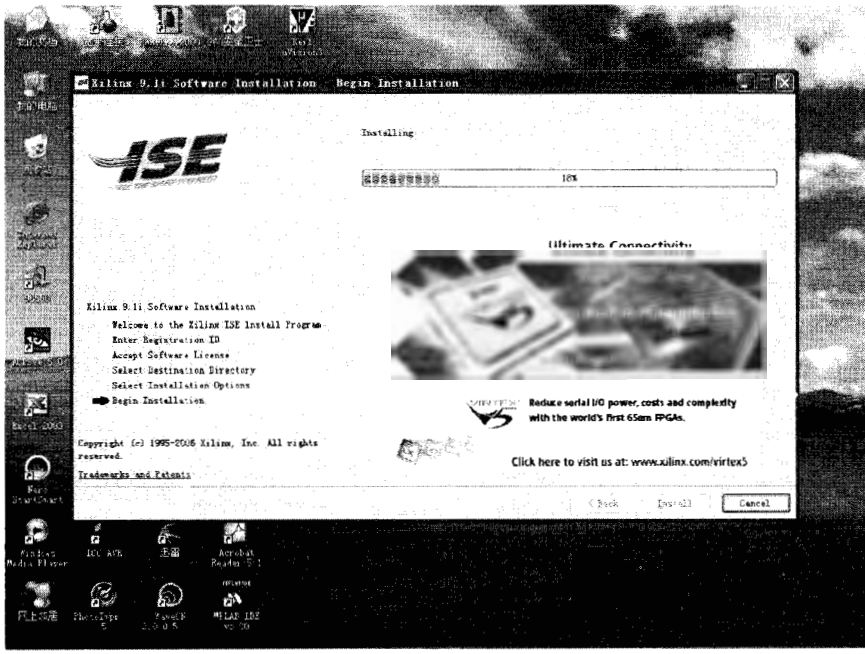


图 6-13　正在拷贝文件

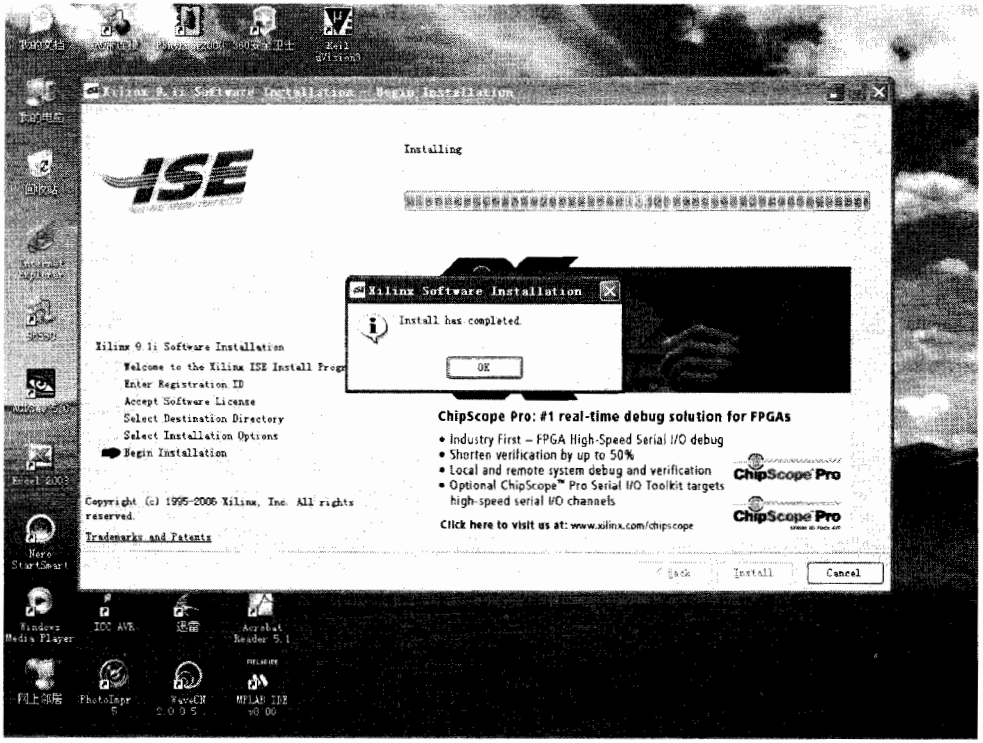


图 6-14　安装结束

6.3　USBasp 下载器软件的安装及使用

6.3.1　USBasp 下载器软件的安装

将配套软件中的“USBasp 下载器的配套软件”文件夹复制到电脑硬盘上（例如复制到 D 盘上）。使用 USB 电缆，一端（方口）插 USB 下载器，另一端（扁口）插电脑的 U 口。电脑会出现“发现新硬件”的提示（见图 6-15）。

随后出现“找到新的硬件向导”的界面（见图 6-16）。

我们选择“从列表或指定位置安装（高级）（S）”（见图 6-17）。

图 6-15　电脑会出现“发现新硬件”的提示

图 6-16　出现“找到新的硬件向导”的界面

图 6-17　我们选择“从列表或指定位置安装（高级）（S）”

使用“浏览”按键找到刚才复制到硬盘的“USBasp 下载器的配套软件\USBasp 驱动”，然后点击“下一步”（见图 6-18）。

图 6-18　找到刚才复制到硬盘的“USBasp 下载器的配套软件\USBasp 驱动”

电脑进行 USB 驱动程序的安装（见图 6-19）。

图 6-19　电脑进行 USB 驱动程序的安装

USB 驱动程序安装完毕后，点击“完成”（见图 6-20）。在桌面的右下方也会出现“新硬件已安装并可以使用了”的提示（见图 6-21）。

图 6-20　USB 驱动程序安装完毕后，点击“完成”

图 6-21　桌面的右下方也会出现“新硬件已安装并可以使用了”的提示

6.3.2　USBasp 下载器的使用

打开“USBasp 下载器的配套软件\USBasp 下载软件”，双击“progisp.exe”图标，出现图 6-22 的 PROGISP 下载软件界面。为了方便以后的使用，我们也可右击“progisp.exe”图标，然后在桌面上生成一个快捷图标。

在 PROGISP 下载软件界面中，“编程器及接口”栏我们选择“USBASP”和“usb”。“选择芯片”栏我们可根据要求进行选择。例如：如果使用 AT89S51，就选择“AT89S51”。

将 10 芯的扁平编程电缆，一端插 USB 下载器的 10 芯座，另一端插 MCU & CPLD DEMO 综合试验板的 MCU-ISP 下载口（注意不要插错）。

提示：如果单片机的晶振频率小于 1.5MHz，我们必须用一个短路块插到 USB 下载器的 10 芯座右侧的双芯针上（见图 6-23），使 USB 下载器能适应单片机较低的频率。当然，如果单片机的晶振频率大于 1.5MHz，我们应该撤下短路块，这样可以达到最快的下载速度。

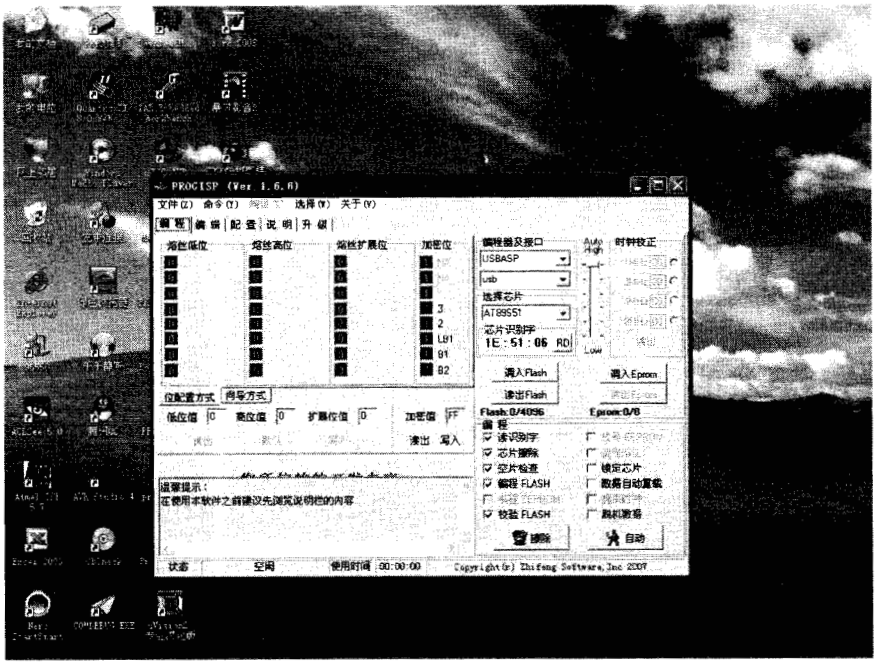


图 6-22　双击“progisp.exe”图标，出现下载软件界面

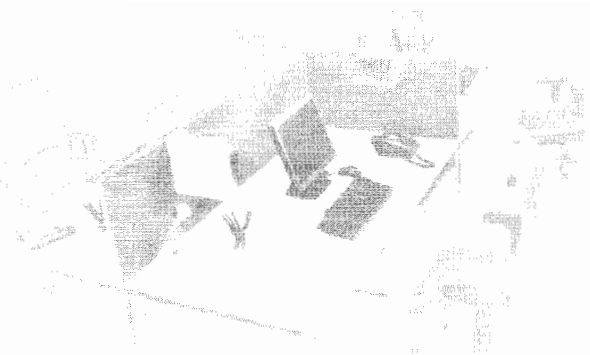
图 6-23　用一个短路块插到 USB 下载器的 10 芯座右侧的双芯针上

程序的下载我们以 AT89S51 为例。选择好芯片后，点击“调入 Flash”，找到我们需要烧写的 hex 文件。在编程框中，“读识别字”、“芯片擦除”、“空片检查”、“编程 FLASH”、“校验 FLASH”前选中打钩。点击“自动”进行编程（见图 6-24）。

图 6-24　点击“自动”进行编程

编程成功后，下方的窗口会出现“Successfully done”的成功提示（见图6-25）。

图 6-25　下方的窗口会出现“Successfully done”的成功提示



第7章

入门的第一个实验程序

使用 Xilinx ISE 集成开发软件进行开发的过程为：

（1）新建项目。

（2）设计输入。

（3）锁定引脚。

（4）编译项目。

（5）仿真。

（6）编程下载。

（7）应用。

我们先将单片机从 MCU & CPLD DEMO 试验板上取下，而仅留 CPLD 进行实验。

7.1 新 建 项 目

打开 Xilinx ISE9.1i 集成开发软件，点击“File→New Project”（见图 7-1），将该新项目保存在 E 盘中建立一个文件名为“decoder2_4”的文件夹中，项目名命名为“decoder2_4”，编程语言栏选择 HDL 硬件描述语言（如果我们需要使用原理图设计，那么选择“Schematic”），如图 7-2 所示。

随后点击“Next”选择目标器件，这里我们选择使用“XC95108”，封装为 PLCC84 脚，速度等级为 10，如图 7-3 所示。点击“Next”后创建新的源文件（见图 7-4）。由于我们使用 Verilog HDL 语言进行设计，因此选择右上角的“New Source”，出现语言选择向导后，选择“Verilog Module”，文件名命名为“decoder2_4”，如图 7-5 所示。

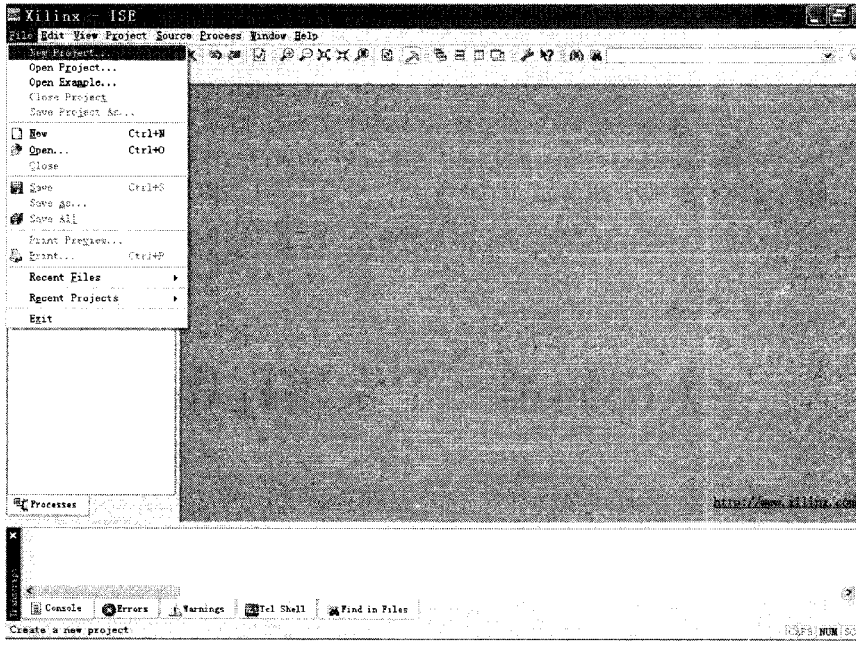


图 7-1　点击“File”→“New Project”新建项目

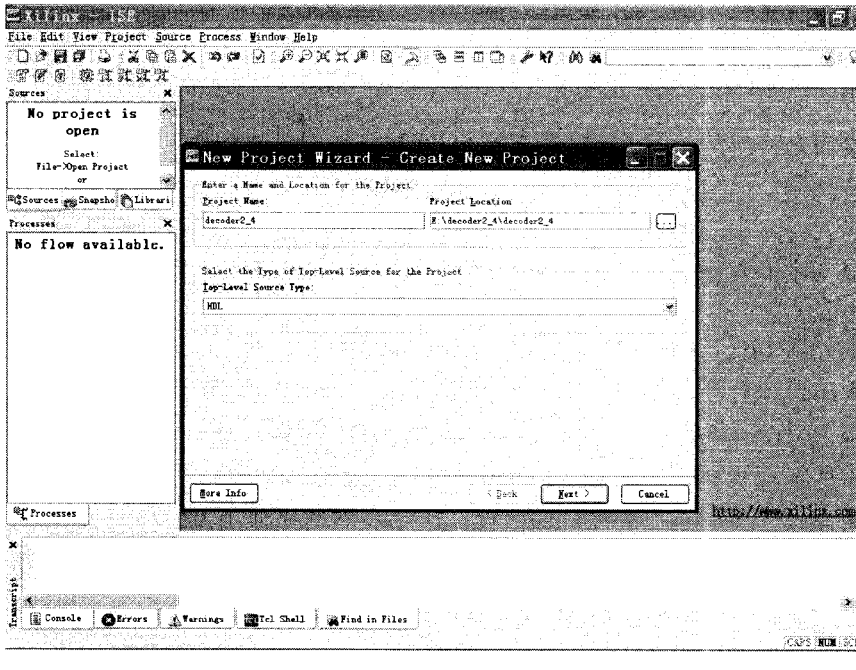


图 7-2　项目名命名为“decoder2_4”

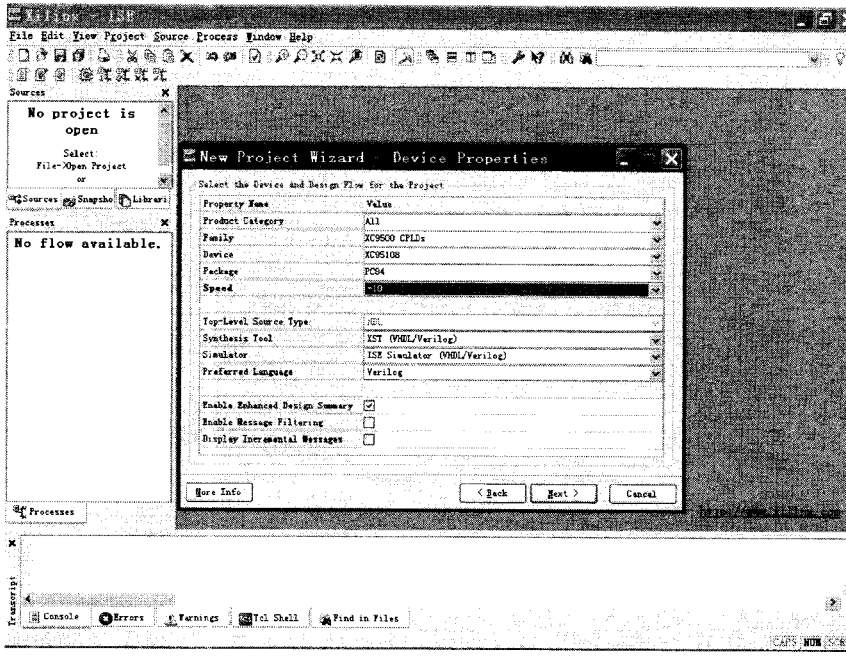


图 7-3　点击“Next”选择目标器件

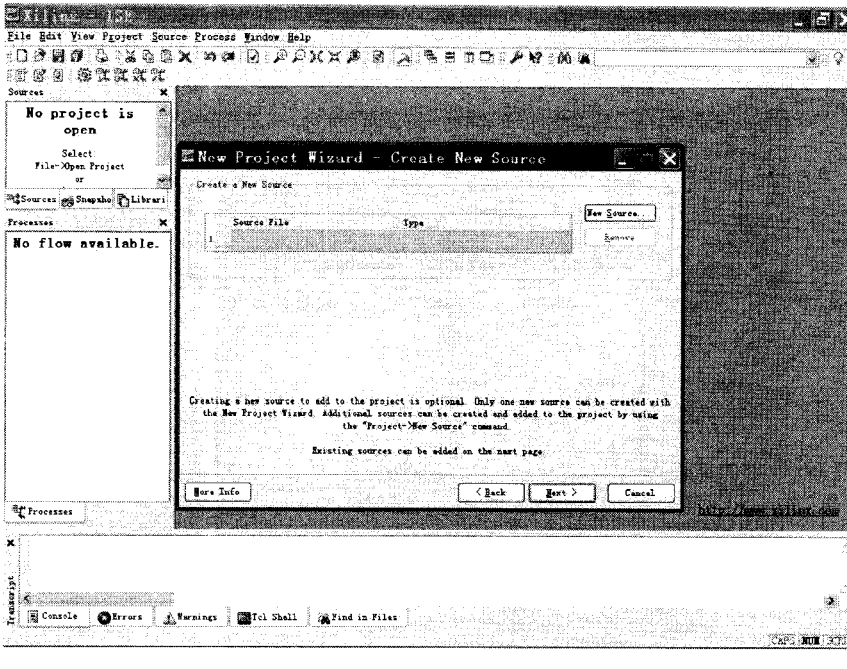


图 7-4　点击“Next”后创建新的源文件

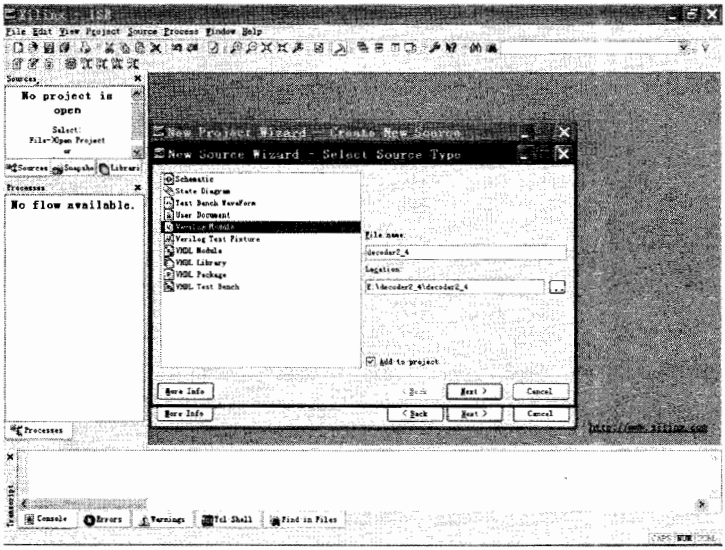


图 7-5　选择“Verilog Module”,文件名命名为“decoder2_4”

再点击“Next”后出现如图 7-6 所示的引脚定义窗口，要求定义您所用到的输出、输出引脚。可以现在进行定义，也可以到后面再定义。我们选择到后面再定义，因此继续点击“Next”。随后出现源文件向导综述界面（见图 7-7），点击“Finish”关闭此界面。

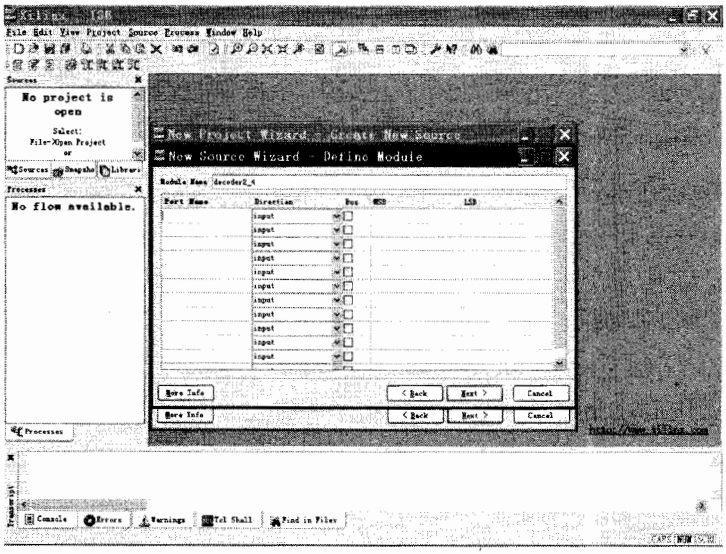


图 7-6　出现引脚定义窗口

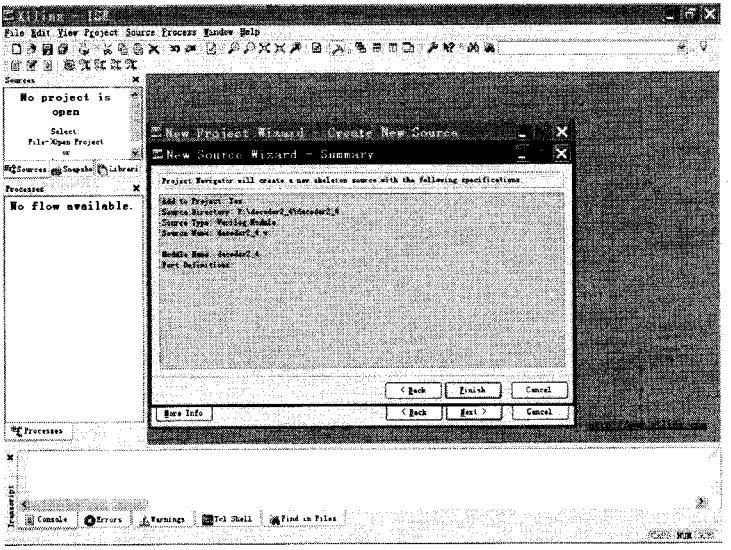


图 7-7　出现源文件向导综述界面

接下来出现创建新源文件的提示（见图 7-8），点击“Next”后，出现添加外部源文件的提示（见图 7-9）。由于不需要添加外部源文件，因此点击“Next”。最后出现一个项目综述的界面（见图 7-10），点击“Finish”后完成项目的新建，同时出现软件自动生成的 Verilog HDL 源文件的框架（见图 7-11）。

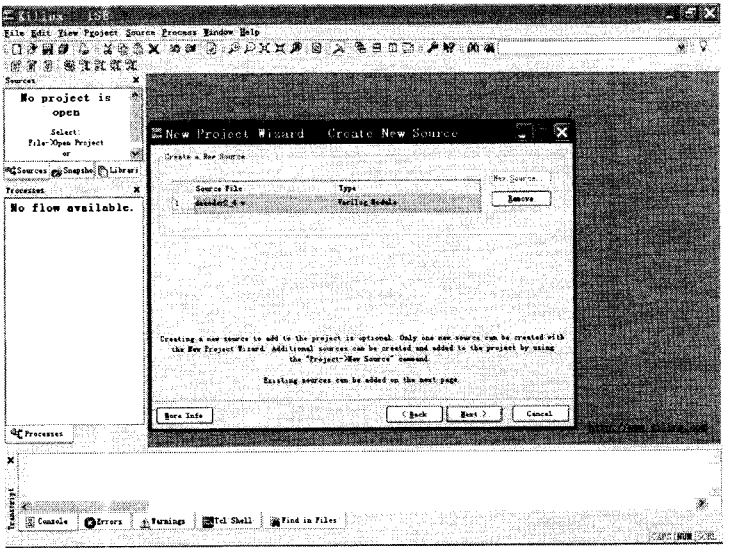


图 7-8　出现创建新源文件的提示

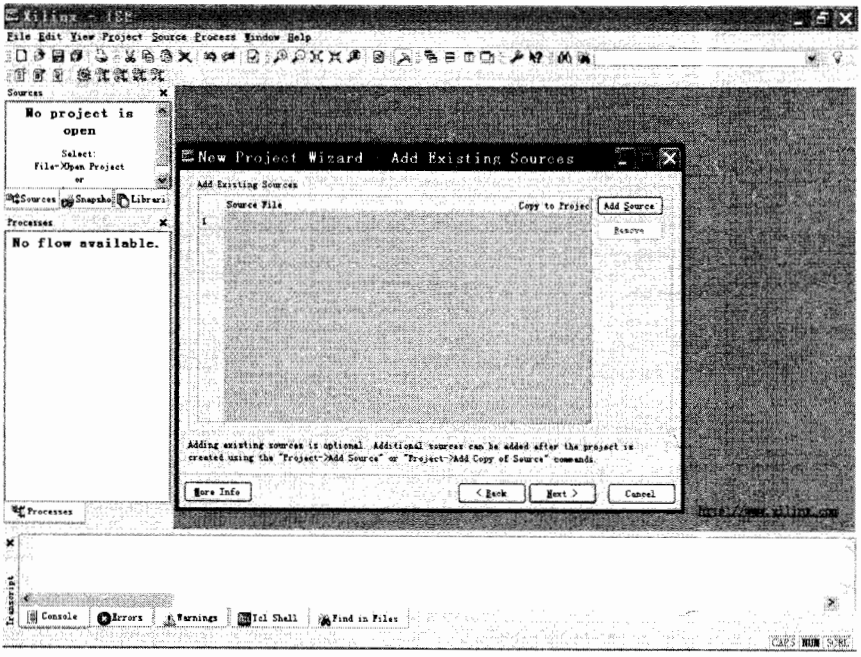


图 7-9　出现添加外部源文件的提示

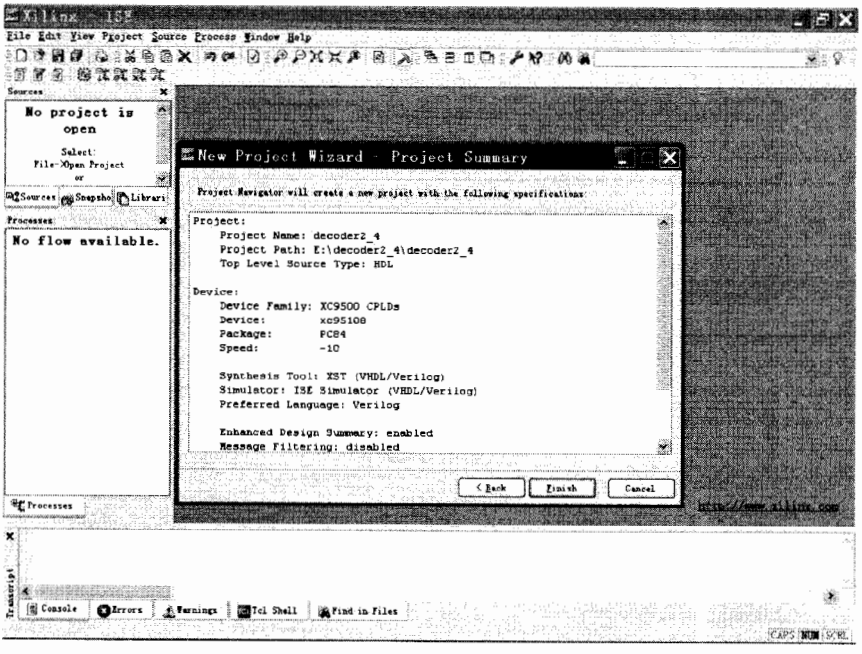


图 7-10　最后会出现一个项目综述的界面

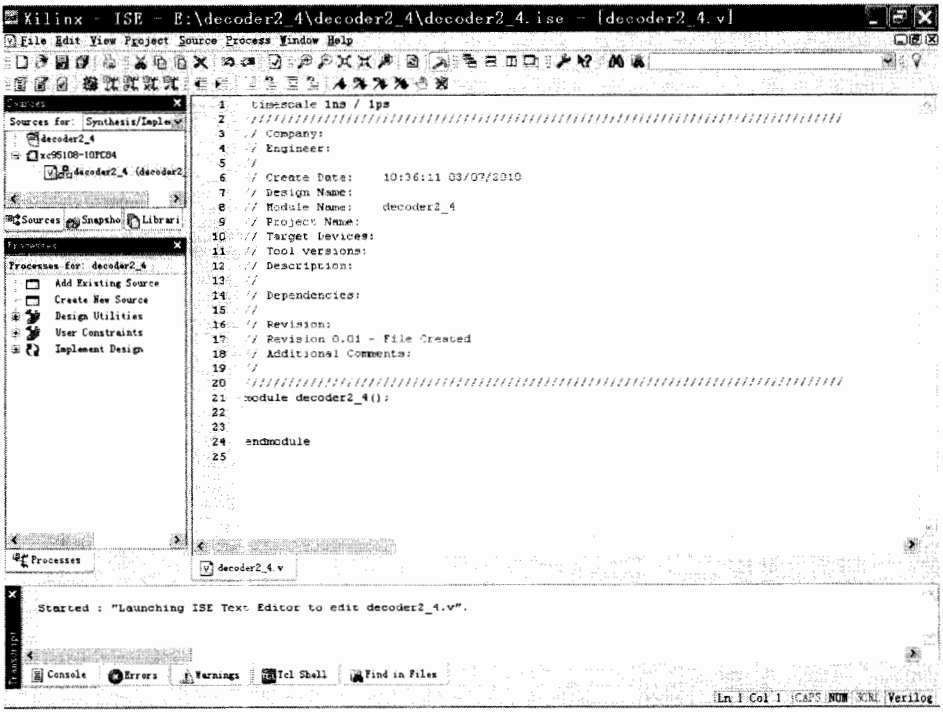


图 7-11　软件自动生成的 Verilog HDL 源文件的框架

7.2　设　计　输　入

在如图 7-11 所示的程序编辑窗口中完成输入如下 Verilog HDL 代码（见图 7-12），最后点击工具栏内的“保存”按钮保存整个工程项目。

输入以下的程序代码：

```
module decoder2_4(LED,S);
output [3:0]LED;
input [1:0]S;
reg [3:0]LED;
always@(S)
begin
case(S)
2'b000:LED=4'b0001;
2'b001:LED=4'b0010;
```

```
2'b010:LED=4'b0100;
2'b011:LED=4'b1000;
endcase
end
endmodule
```

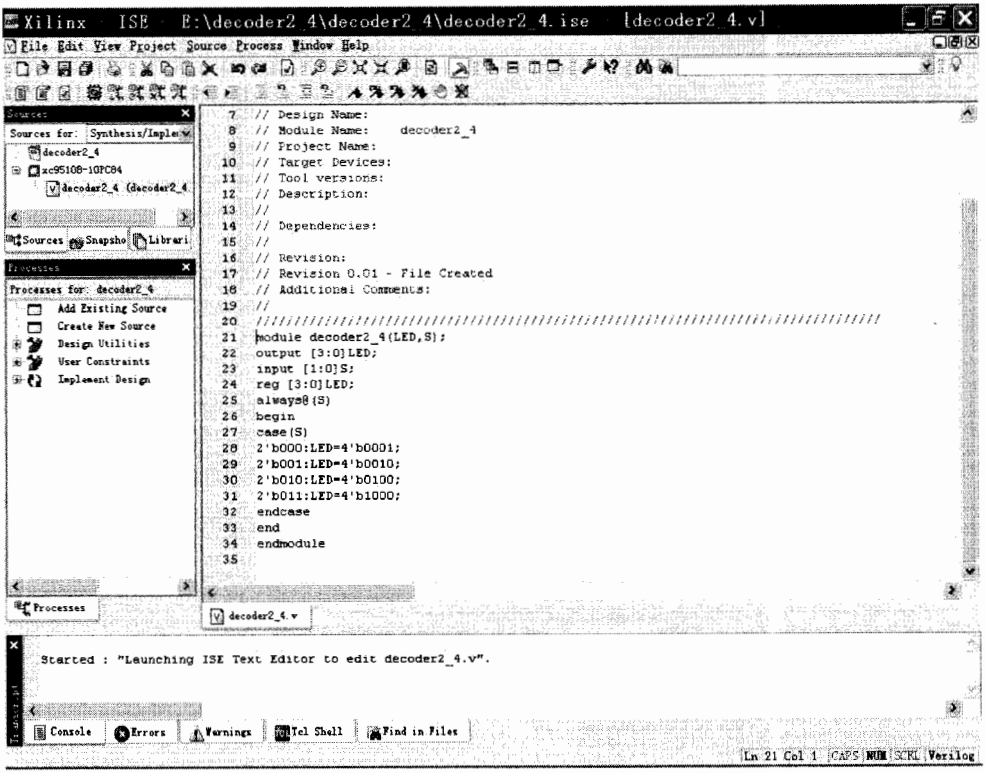


图 7-12　完成输入 Verilog HDL 代码

7.3　锁　定　引　脚

在开发软件左面的进程窗口中，单击“User Constrains”前的+号展开（见图 7-13），展开后选中“Assign Package Pins”，并双击它，软件会弹出一个引脚锁定的对话框（见图 7-14）。

根据 MCU & CPLD DEMO 试验板电路原理，得到本例引脚锁定表（见表 7-1）。

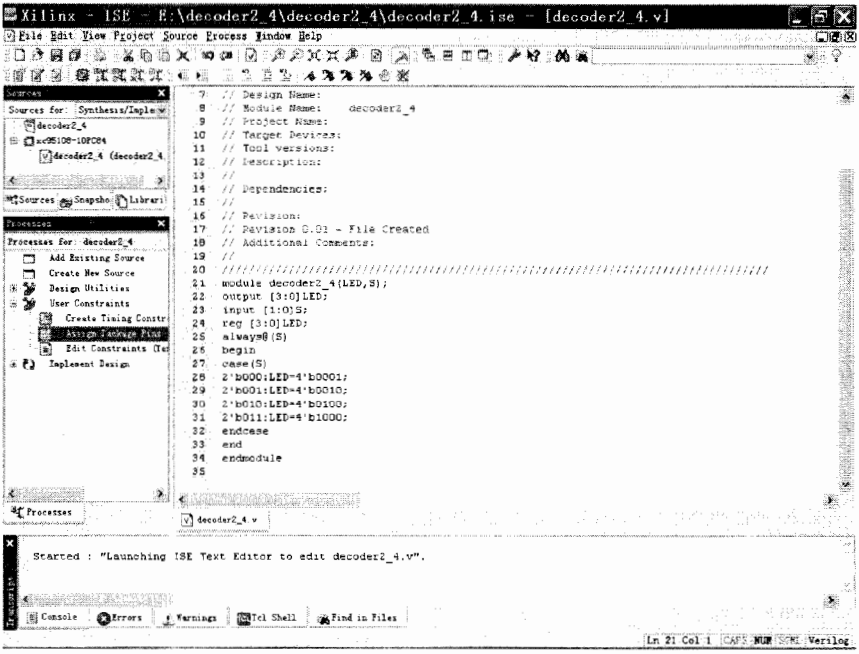


图 7-13　单击“User Constrains”前的+号进行展开

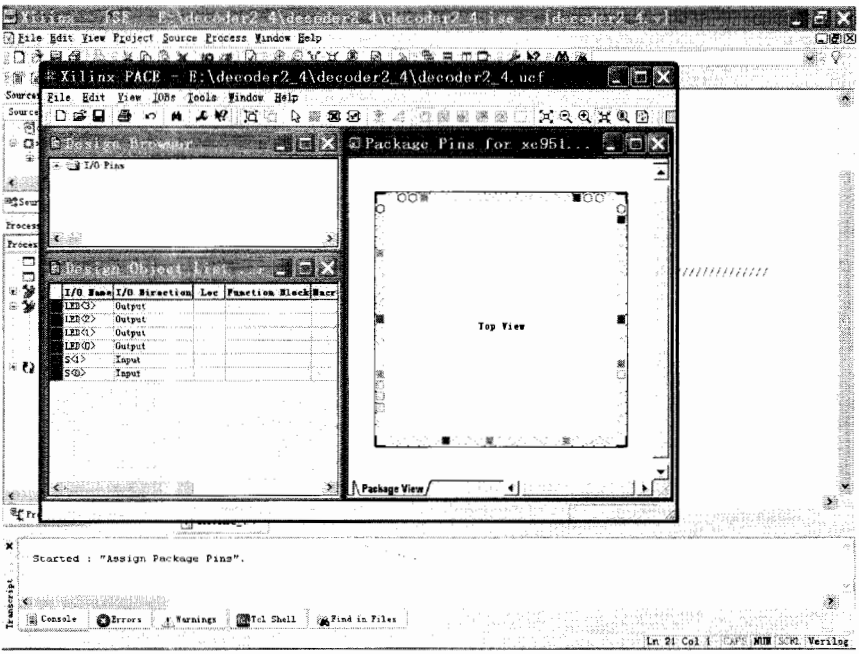


图 7-14　软件会弹出一个引脚锁定的对话框

表 7-1　　引 脚 锁 定 表

引脚名	引脚号	输入或输出	板上丝印符号
S0	41	Input	S0
S1	40	Input	S1
LED0	31	Output	LED0
LED1	26	Output	LED1
LED2	25	Output	LED2
LED3	24	Output	LED3

我们进行输入和输出引脚的锁定：首先在“Design Object List...”框内，点击选择 LED<3>左边的蓝色小方框，然后再次点击并保持不放，拖动鼠标，将此小方块移到芯片的 24 号引脚上，鼠标在空白处点一下，24 号引脚变成深蓝色，同时，LED<3>左边的蓝色小方框消失，这样就把 LED<3>锁定好了（见图 7-15）。以此类推，锁定好每一个引脚，锁定完毕后再保存，然后退出引脚锁定对话框。

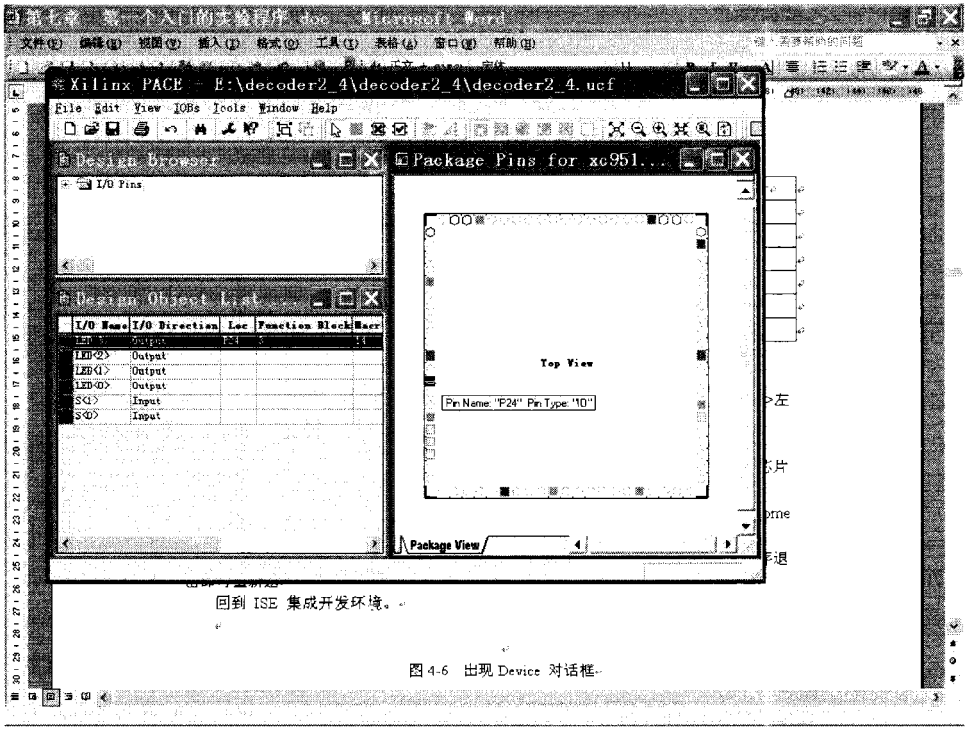


图 7-15　把 LED<3>锁定好

7.4 编 译 项 目

在进程窗口中双击“Implement Design”命令，运行相关进程。其中打钩标记的标示该进程已经成功运行，而打惊叹号的则表示该进程已运行，但是包含有系统给出的警告，如果没有问题，底下的输出窗口会提示已生成可供下载的.Jed 文件，如图 7-16 所示。

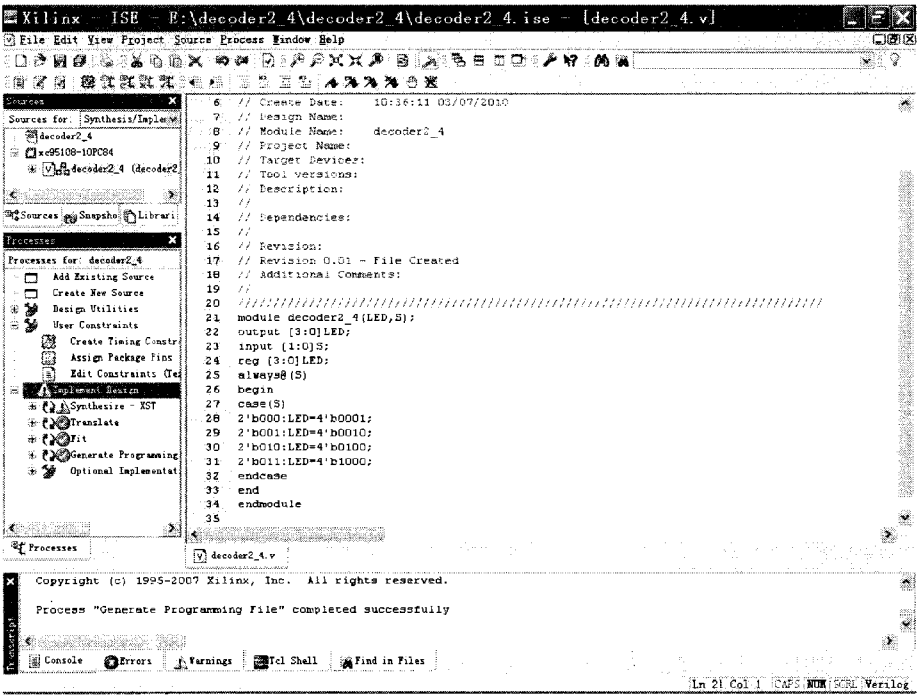


图 7-16　输出窗口会提示已生成可供下载的.Jed 文件

7.5 软 件 仿 真

在左侧的“Source”窗口内，单击右键，在出现的快捷菜单中选择“New Source”（见图 7-17）。在弹出的对话框中选择“Test Bench Waveform”，在右侧 file 下输入文件名，这里取名为“decoder2_4_test”，如图 7-18 所示。随后一直点击“Next”至结束（见图 7-19）。最后出现一个时序及时钟仿真的初始化向导界面（见图 7-20）。

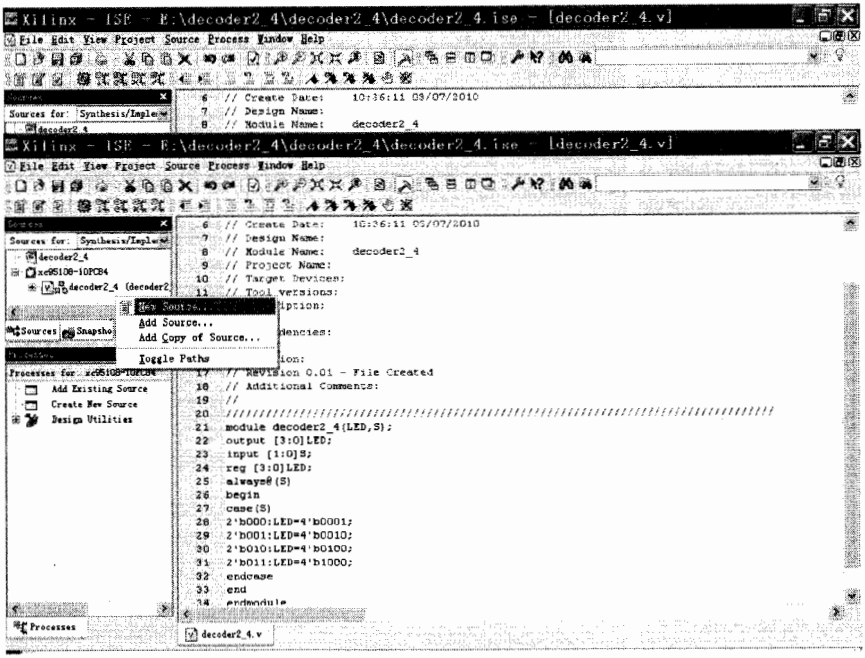


图 7-17　在出现的快捷菜单中选择“New Source”

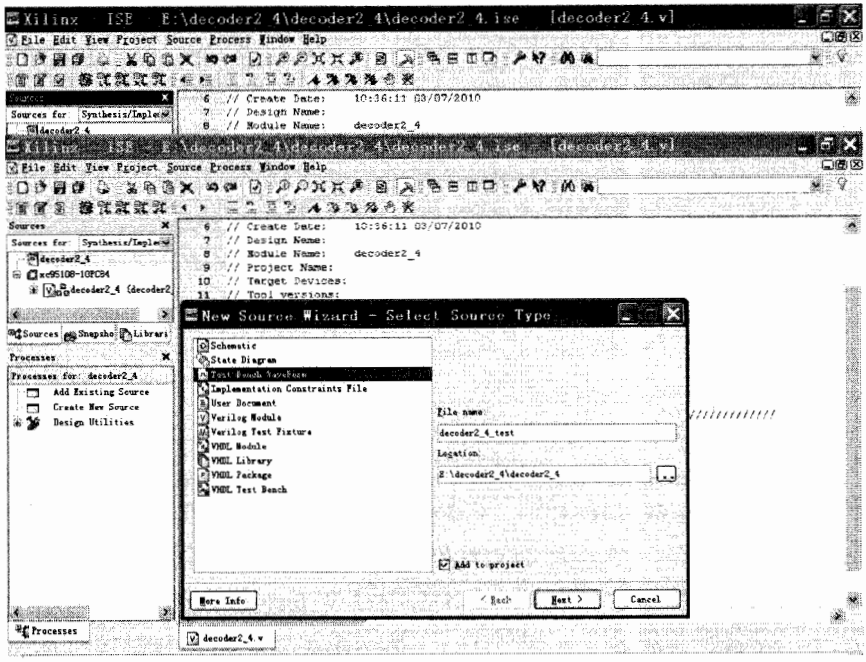


图 7-18　选择“Test Bench Waveform”并输入文件名

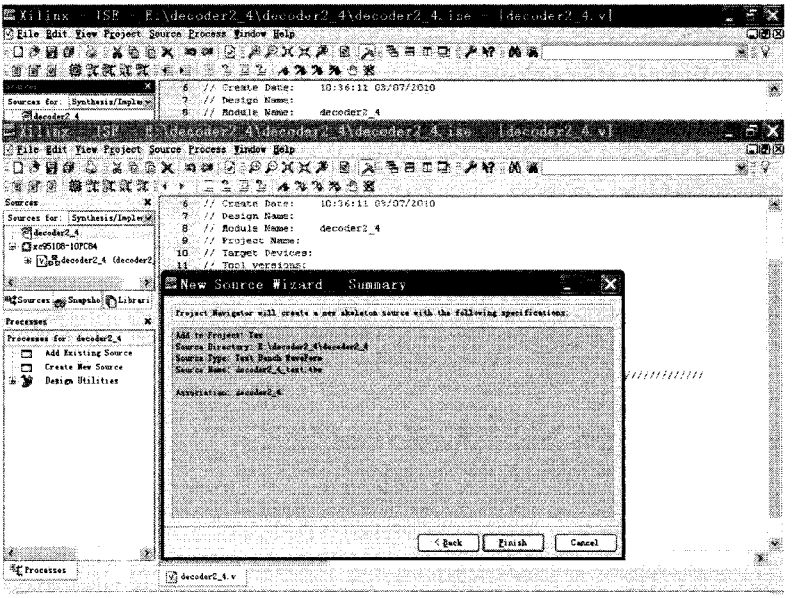


图 7-19　一直单击“Next”至结束

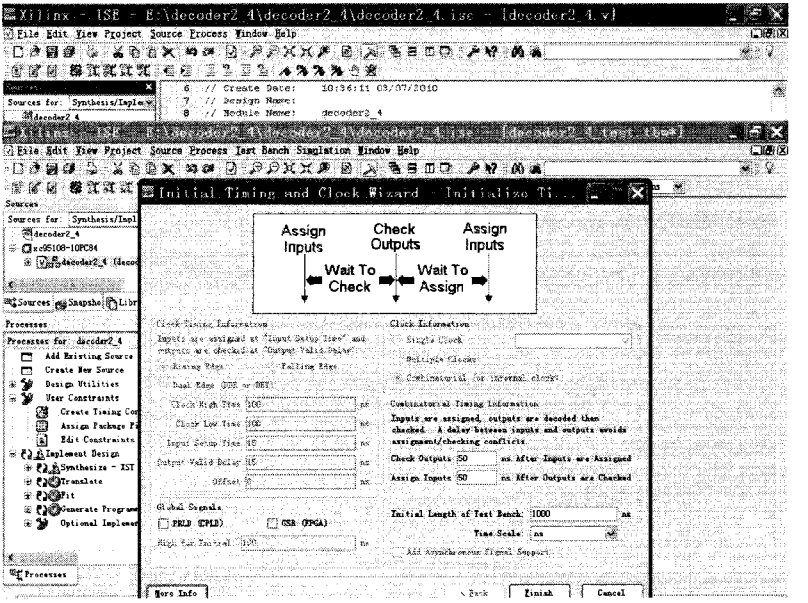


图 7-20　最后出现一个时序及时钟仿真的初始化向导界面

我们的第一个实验是一个简单的组合逻辑电路，所以可以取默认值。点击“Finish”后进入波形设置界面，如图 7-21 所示。

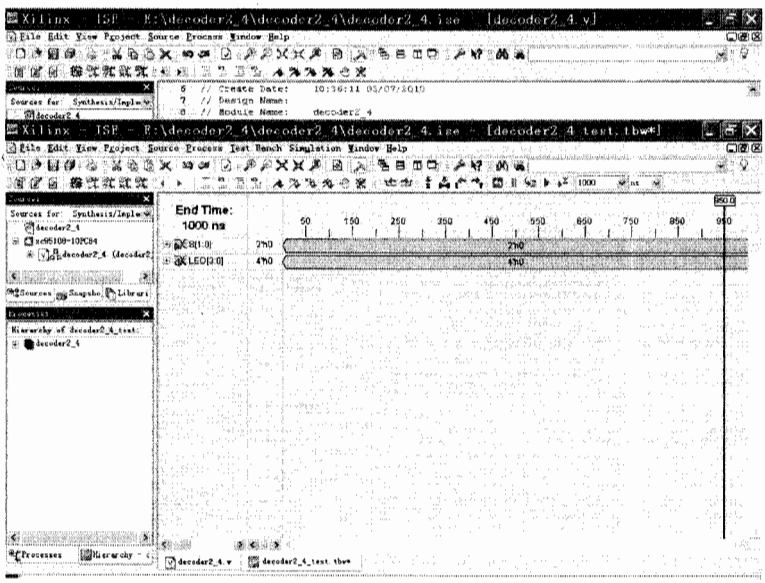


图 7-21　进入波形设置界面

将 S[1:0]及 LED[3:0]展开，如图 7-22 所示。根据设计的要求，设置输入信号波形，我们这里设计的是一个 2-4 译码器，因此 S[1:0]的波形设置如图 7-23 所示，设置完成后点击保存。

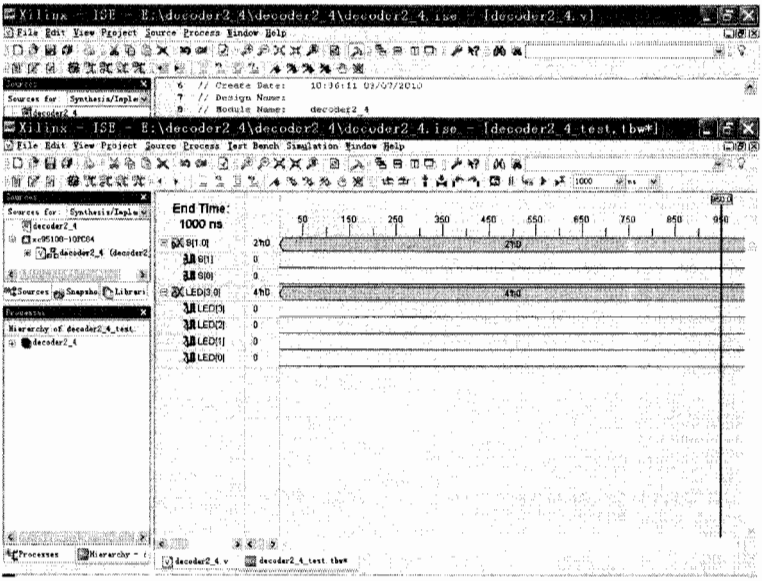


图 7-22　将 S[1:0]及 LED[3:0]展开

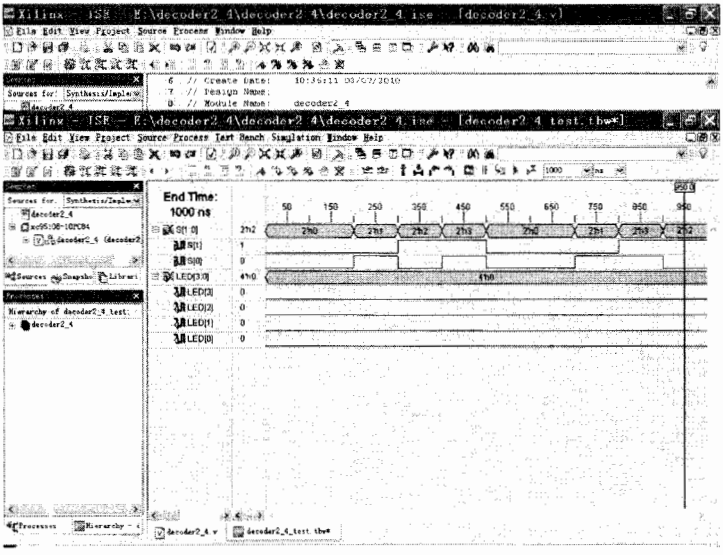


图 7-23　S[1:0]的波形设置

在源窗口的右侧，选择“Behavioral Simulation”，如图 7-24 所示。然后选中“decoder2_4_test.tbw”（见图 7-25），在进程窗口中选择“Simulate Behavioral Model”并双击（见图 7-26），这时软件进行仿真处理，仿真波形的输出如图 7-27 所示，完全符合 2-4 译码器的输出。

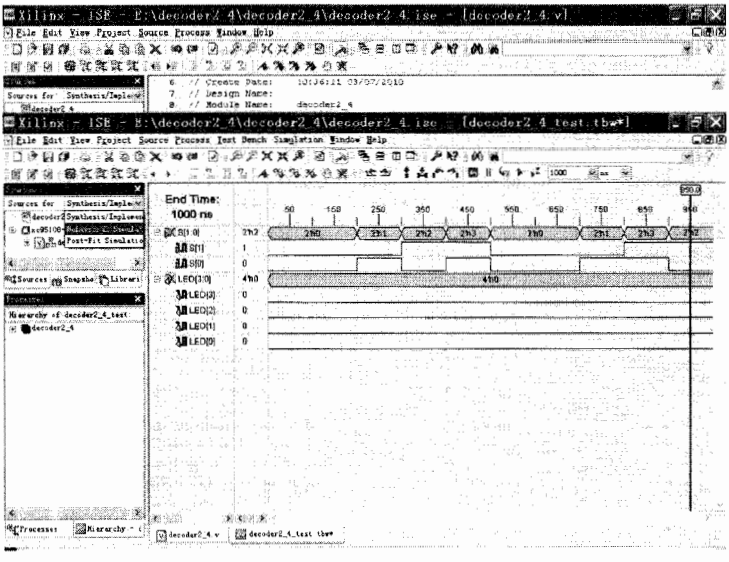


图 7-24　选择“Behavioral Simulation”

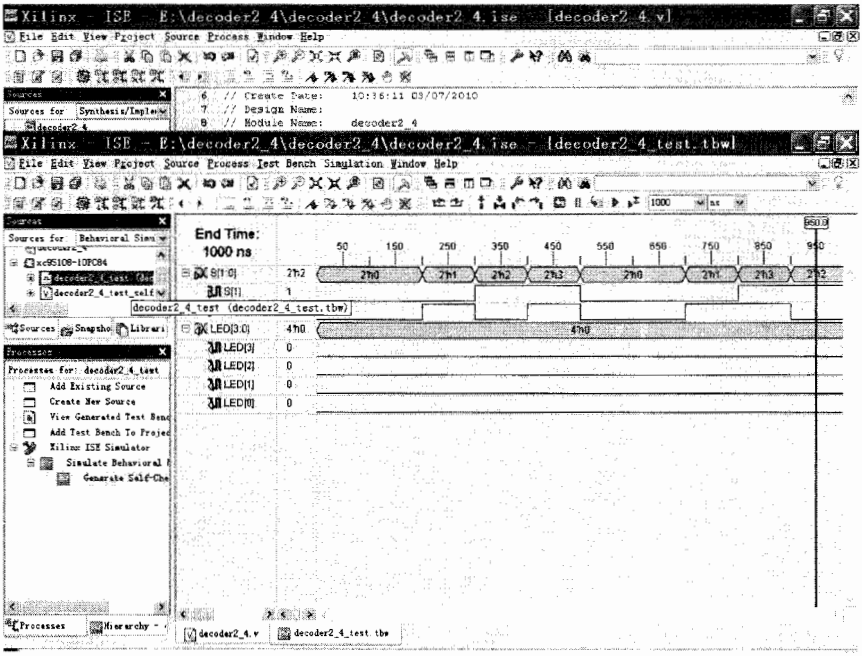


图 7-25　选中“decoder2_4_test.tbw”

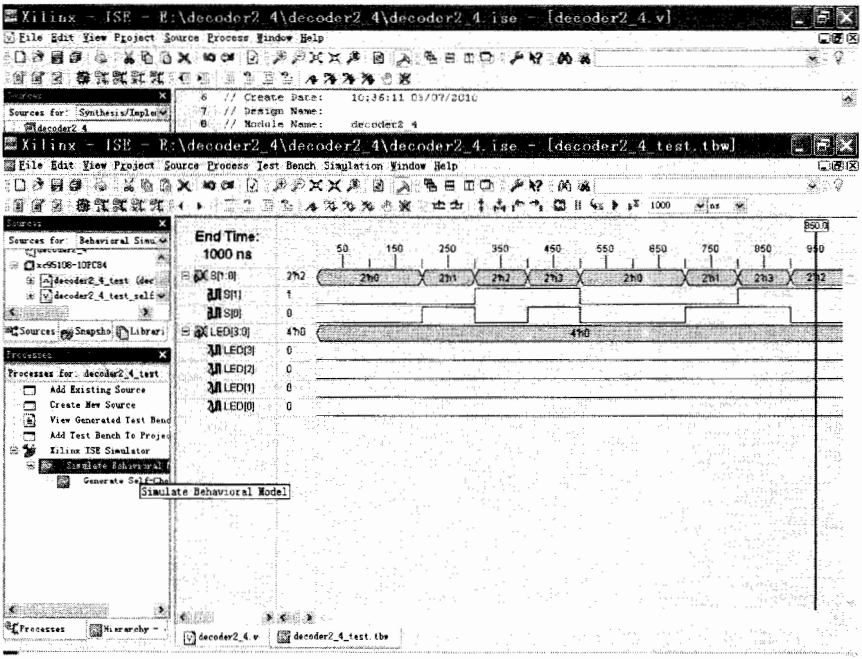


图 7-26　选择“Simulate Behavioral Model”并双击

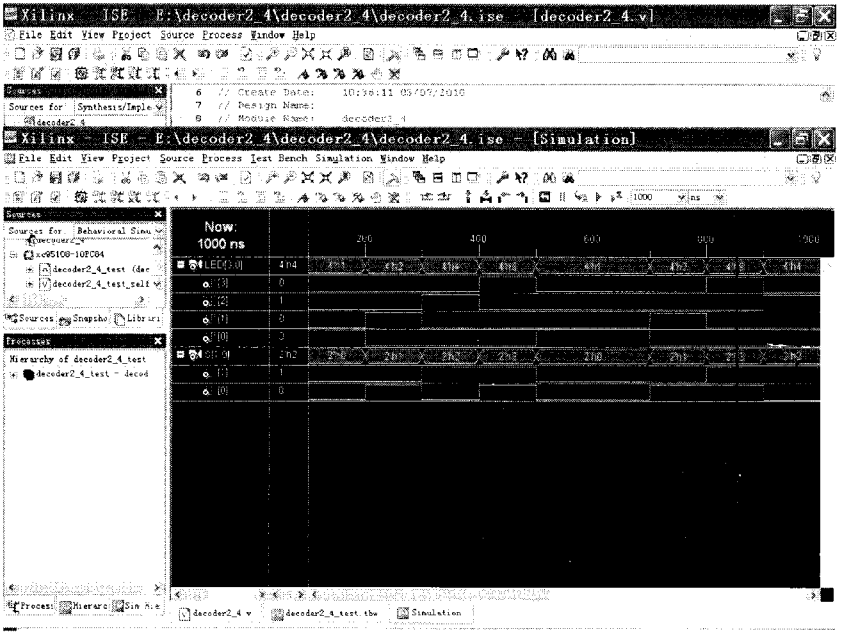


图 7-27　仿真波形的输出

7.6 编　程　下　载

在源窗口的右侧，选择“Synthesis/Implementation”，如图 7-28 所示。然后选中“decoder2_4(decoder2_4.v)”，如图 7-29 所示。在进程窗口中选择“Implement Design→Generate Programming→Configure Device (iMAPCT)”，如图 7-30 所示，并进行双击。这时弹出检测硬件的对话框，我们选择默认的方式由软件自动检测（见图 7-31）。随后软件又弹出为器件添加 Jed 文件的画面，我们选择“decoder2_4.jed”文件并双击（见图 7-32）。图 7-33 为已经添加“decoder2_4.jed”文件后的界面，由此可见，右侧的 XC95108 芯片已经加载了 decoder2_4.jed 文件。

将 Xilinx 的 JTAG 下载器插在电脑的 25 针打印口上，下载电缆的 10 脚插头插入 MCU & CPLD DEMO 试验板的 CPLD-JTAG 下载口中。MCU & CPLD DEMO 试验板通入 9~12V 的工作电源，打开电源开关。

在开发界面中，鼠标选中 XC95108 芯片并右击，弹出一个编程步骤的选择界面，我们使用默认的方式，即编程前先擦除（见图 7-34），点 OK 后开始下载程序，下载完成后，软件会出现成功的提示（见图 7-35）。

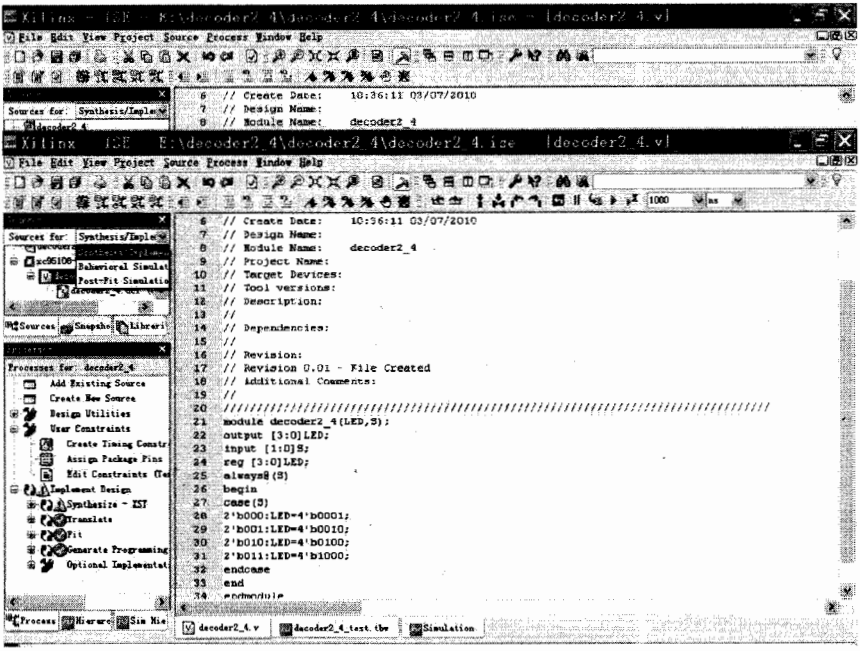


图 7-28　选择“Synthesis/Implementation”

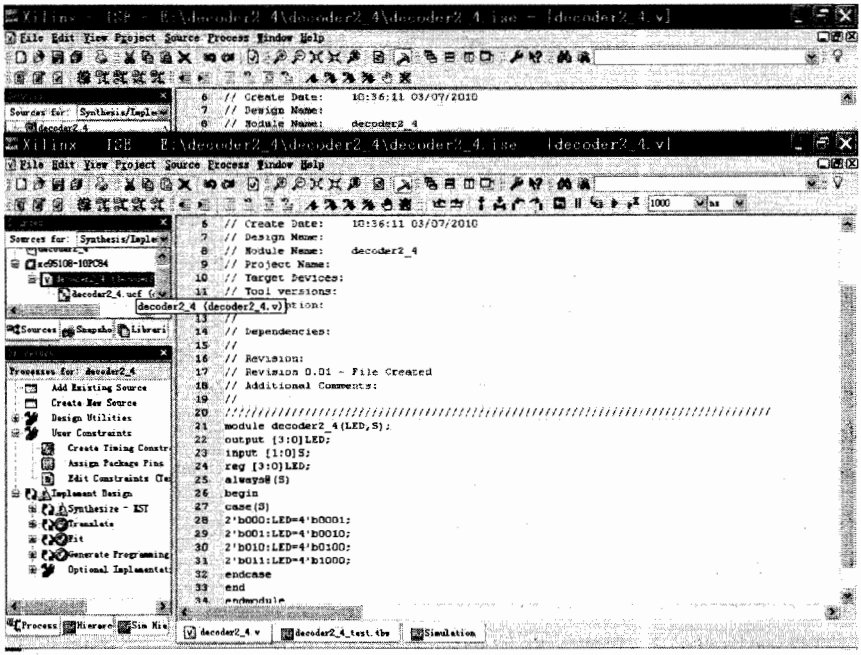


图 7-29　然后选中“decoder2_4(decoder2_4.v)”

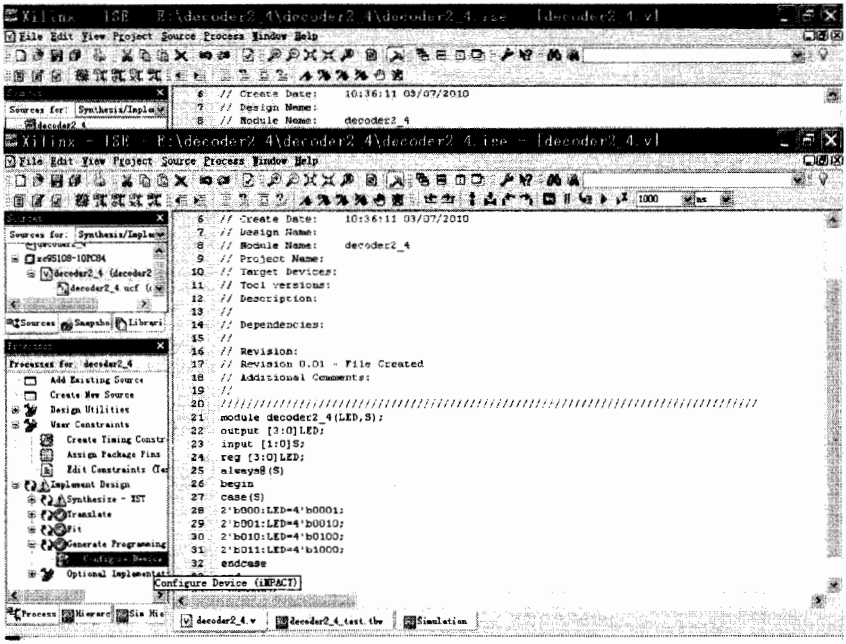


图 7-30　选择“Implement Design”→“Generate Programming”→“Configure Device (iMAPCT)”

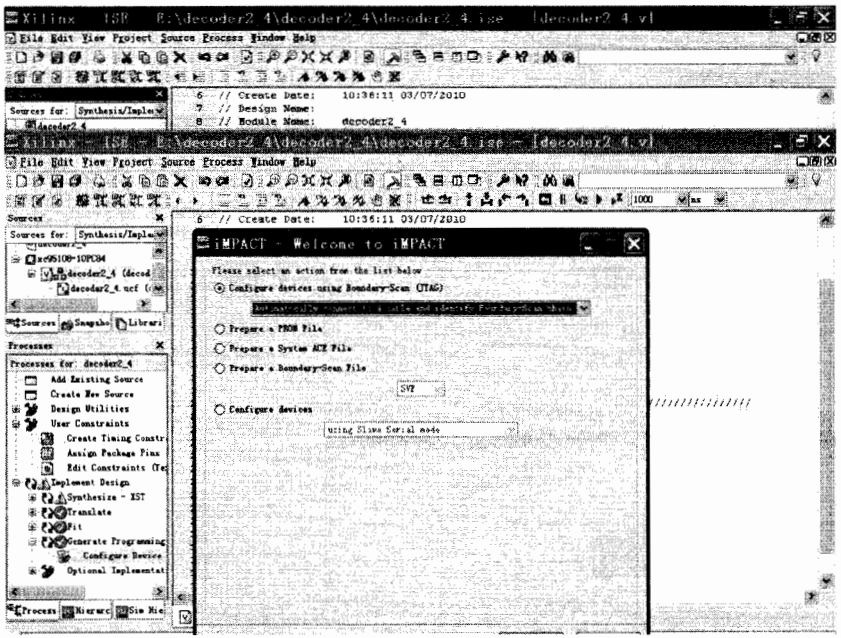


图 7-31　选择默认的方式由软件自动检测

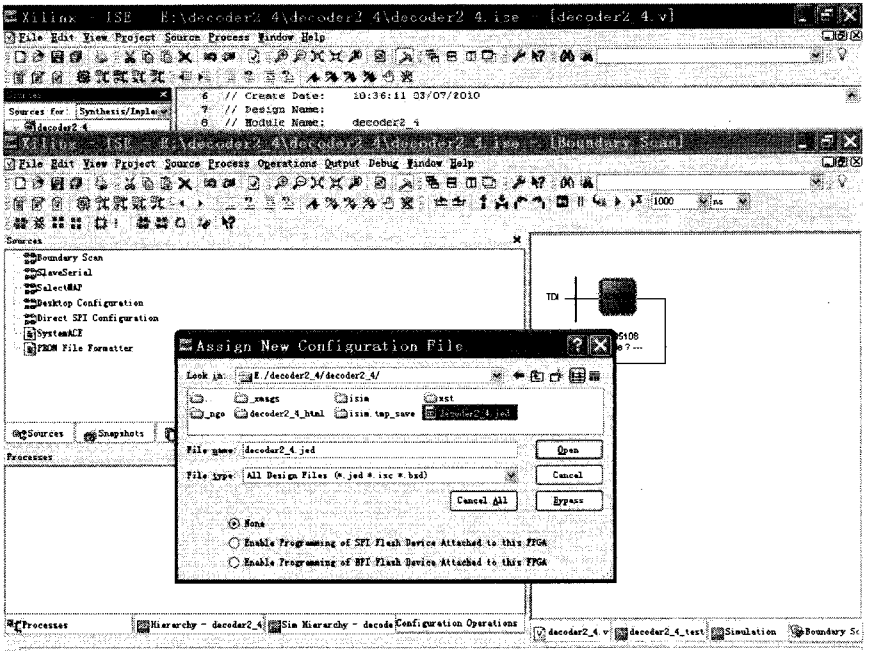


图 7-32 选择“decoder2_4.jed”文件并双击

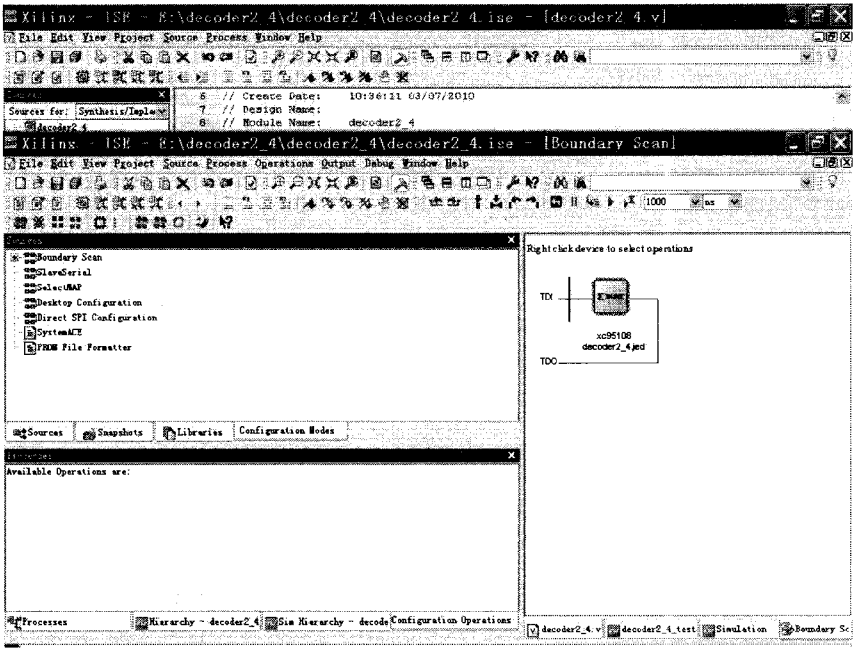


图 7-33 已经添加“decoder2_4.jed”文件后的界面

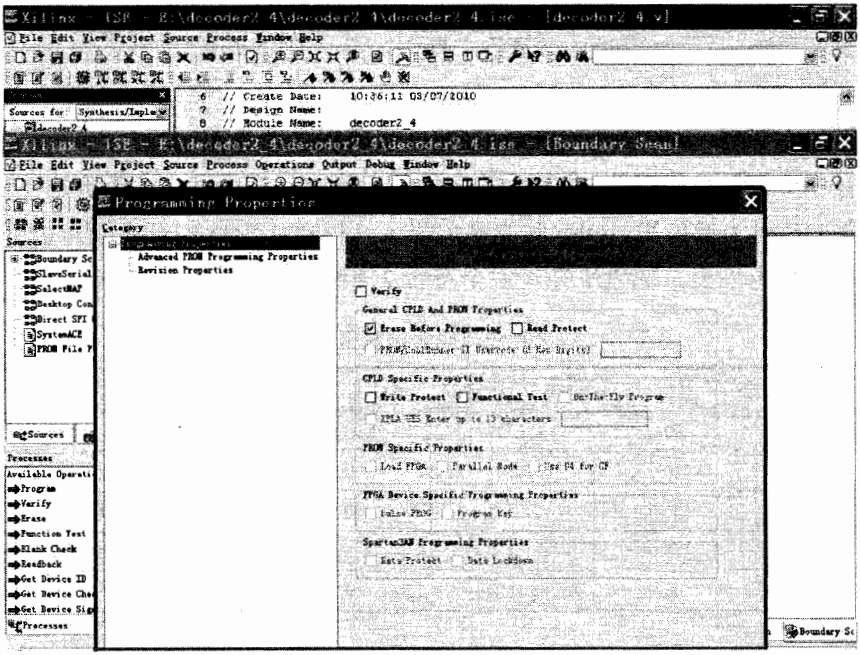


图 7-34　使用默认的方式，即编程前先擦除

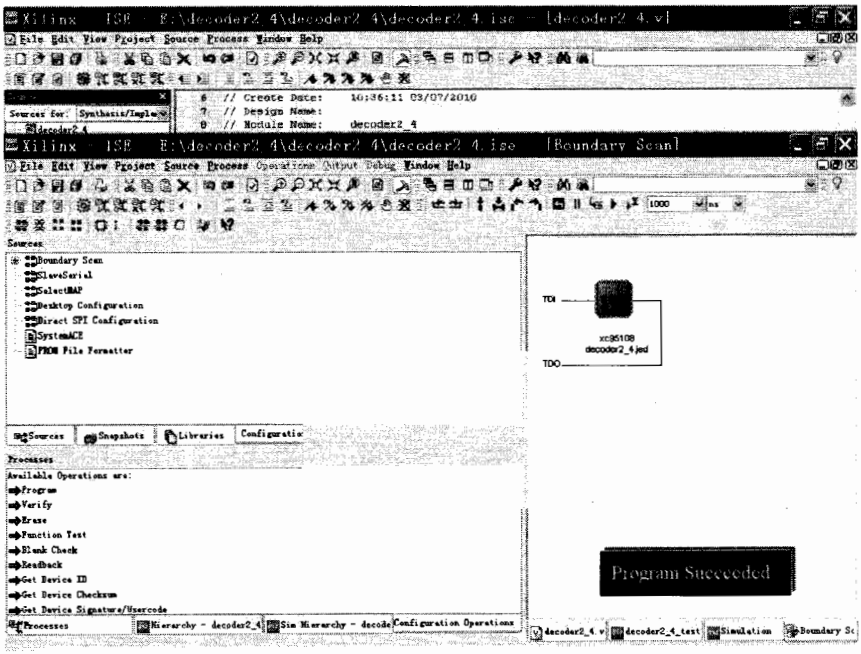


图 7-35　下载完成后，软件会出现成功的提示

7.7 应　　用

下载完毕后，MCU & CPLD DEMO 试验板自动进入运行状态，板子的左下角有一个 SW 拨码开关，我们看到上面标示有 S0~S3 的字样。拨动 S0~S1（拨码开关 ON 时为低电平，OFF 时为高电平），LED0~LED3 的灯光输出信号会发生变化，高电平灭，低电平亮，完全符合 2-4 线译码器的真值表（见表 7-2）。

表 7-2　　2-4 线译码器的真值表

输　出					
S1	S0	LED3	LED2	LED1	LED0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

第 8 章

Verilog HDL 硬件描述语言

Verilog HDL 是一种用于数字系统设计的硬件描述语言，它可用来进行各种级别的逻辑设计，以及数字逻辑系统的仿真验证、时序分析和逻辑综合。Verilog HDL 是目前应用最广泛的一种硬件描述语言。

8.1 Verilog HDL 模块的基本结构

Verilog HDL 程序是由模块构成的。每个模块的内容部位于 module 和 endmodule 这 2 个关键字之间，Verilog HDL 模块的基本结构如图 8-1 所示。每个 Verilog HDL 模块包括 4 个主要部分：模块声明、端口定义、信号类型说明和逻辑功能描述，每个模块用于实现特定的功能。

module 模块声明（<输入输出端口列表>）
端口定义 output 输出端口 input 输入端口 inout 端口
信号类型说明 wire reg parameter
逻辑功能描述 assign always function task ……
endmodule

图 8-1 Verilog HDL 模块的基本结构

8.1.1 模块声明

模块声明包括模块名字，模块输入、输出端口列表。模块定义格式如下：

```
module 模块名(端口名1,端口名2,……, 端口名n);
```

模块的端口表示的是模块的输入和输出口名，也就是它与别的模块联系端口的标识。在模块被引用时，该模块有的信号要输入到被引用的模块中，有的信号需要从被引用的模块中取出。

8.1.2 端口定义

对模块的输入输出端口要明确说明，其格式为：

```
input 端口名1,端口名2,……端口名n;          //输入端口
output 端口名1,端口名2,……端口名n;         //输出端口
inout 端口名1,端口名2,……端口名n;          //输入输出端口
```

端口是模块与外界或其他模块连接和通信的信号线，一个模块的端口如图8-2所示。有三种端口类型，分别是输入端口（input）、输出端口（output）和输入/输出端口（inout）。

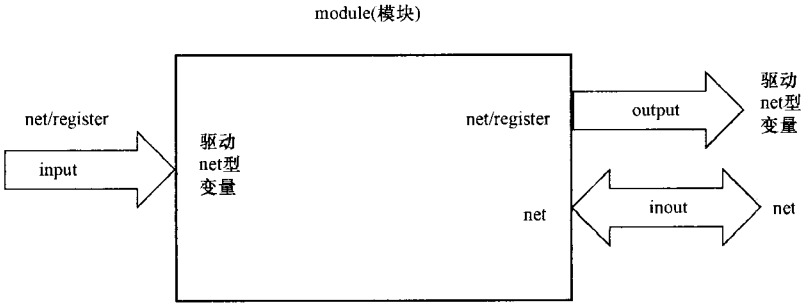


图 8-2 Verilog HDL 模块的端口

对于端口应注意以下事项：

（1）每个端口除了要声明是输入、输出、还是双向端口外，还要声明其数据类型，是连线型（wire），还是寄存器型（reg），如果没有声明，则综合器将其默认为是 wire 型。

（2）输入和双向端口不能声明为寄存器型。

（3）在测试模块中不需要定义端口。

8.1.3　信号类型说明

对模块中所用到的所有信号（包括端口信号、节点信号等）都必须进行数据类型的定义。Verilog HDL 语言提供了各种信号类型，分别模拟实际电路中的各种物理连接和物理实体。信号类型声明为在模块内用到的和与端口有关的 wire 及 reg 类型变量的声明。

下面是定义信号数据类型的几个例子：

```
reg bout;              //定义信号 bout 的数据类型为 reg 型
reg[3:0] dataout;      //定义信号 dataout 的数据类型为 4 位 reg 型
wire A,B,C,D;          //定义信号 A,B,C,D 为 wire(连线)型
```

如果信号的数据类型没有定义，则综合器将其默认为是 wire 型。

8.1.4　逻辑功能描述

模块中最核心的部分是逻辑功能定义。有多种方法可在模块中描述和定义逻辑功能，还可以调用函数（function）和任务（task）来描述逻辑功能。下面介绍定义逻辑功能的几种基本方法。

1. 用“assign”持续赋值语句定义

【例】 `assign out=a&b;`

“assign”语句是描述组合逻辑最常用的方法之一，称为持续赋值方式。这种方法简单，只需将逻辑表达式放在关键字“assign”后即可。

2. 调用实例元件

调用实例元件的方法类似于在电路图输入方式下调入图形符号来完成设计，输入元件的名字和相连的引脚即可，这种方法侧重于电路的结构描述。在 Verilog HDL 语言中，可通过调用如下元件的方式来描述电路的结构。

（1）调用 Verilog HDL 内置门元件（门级结构描述）。

（2）调用开关级元件（开关级结构描述）。

在多层次结构电路设计中，不同模块的调用也可以认为是结构描述。下面是元件调用的例子：

```
and a2(out,in1,in2);   //调用门元件,定义了一个二输入的与门,名字为 a2
```

3. 用“always”过程块赋值

“always”过程块既可用于描述组合逻辑，也可描述时序逻辑。

【例】 有以下代码：

```
always @(posedge clk)        //每当 clk 上升沿到来时执行一遍 begin-end
                               块内的语句
begin
     if(clr) out<=0;
     else out<=out+1;
end
```

上面的代码用“always”块来描述逻辑功能，一般称为行为描述方式，这种方法一般多用于描述时序逻辑。

归纳起来，Verilog HDL 模块的模板如下：

```
module <顶层模块名> (<输入输出端口列表>);
output 输出端口列表;                    //输出端口声明
input 输入端口列表;                     //输入端口声明

/*定义数据,信号类型,函数声明,用关键字 wire,reg,task,function 等定义*/
wire 信号名;
reg 信号名;

/*逻辑功能定义*/
assign <结果信号名>=<表达式>;      //使用 assign 语句定义逻辑功能

/*用 always 块描述逻辑功能*/
always @(<敏感信号表达式>)
    begin
    //过程赋值
    //case 语句
    //while,repeat,for 循环语句
    //task,function 调用
    end

/*调用其他模块*/
<调用模块名>  <实例化模块名>  (<端口列表>);

/*门元件实例化*/
门元件关键字 <实例化门元件名>  (<端口列表>);
endmodule
```

8.2　Verilog HDL 语法要素

Verilog HDL 程序是由各种符号流构成的，这些符号包括空白符、操作符、数字、字符串、注释、标识符和关键字等。

8.2.1　标识符与关键字

1. 标识符

标识符是用来标识源程序中某个对象的名字，这些对象可以是变量、语句、数据类型和函数等。Verilog HDL 中的标识符可以是任意一组字母、数字以及符号“$”和“_”（下划线）的组合，但标识符的第一个字符必须是字母或者是下划线。另外，标识符是区分大小写的。

以下是几个合法的标识符的例子：

```
time
TIME                    //time 与 TIME 是不同的
_A1_b2
S12_6
TOW
```

而下面几个例子则是不正确的：

```
30c                     //非法：标识符不允许以数字开头
out*                    //非法：标识符中不允许包含字符*
```

标识符在命名时，应当简单，含义清晰，并且尽量为每个标识符取一个有意义的名字，这样有助于阅读及理解程序。

2. 关键字

Verilog HDL 语言内部已经使用的词称为关键字或保留字，在程序编写中不允许用户的标识符与关键字相同。以下是 Verilog HDL 语言中所有的关键字，所有关键字都是小写的。

always、and、assign、begin、buf、bufif0、bufif1、case、casex、casez、cmos、deassign、default、defparam、disable、edge、else、end、endcase、endmodule、endfunction、endprimitive、endspecify、endtable、endtask、event、for、force、forever、fork、function、highz0、highz1、if、initial、inout、input、integer、

join、large、macromodule、medium、module、nand、negedge、nmos、nor、not、notif0、notif1、or、output、parameter、pmos、posedge、primitive、pull0、pull1、pullup、pulldown、rcmos、reg、releses、repeat、mmos、rpmos、rtran、rtranif0、rtranif1、scalared、small、specify、specparam、strength、strong0、strong1、supply0、supply1、table、task、time、tran、tranif0、tranif1、tri、tri0、tri1、triand、trior、trireg、vectored、wait、wand、weak0、weak1、while、wire、wor、xnor、xor。

8.2.2 常量、变量及数据类型

1. 常量

在程序运行过程中，其值不能被改变的量称为常量，Verilog HDL 中的常量主要有 3 种类型：整数、实数、字符串，最常用的常量有数字和字符串。

Verilog HDL 有以下 4 种逻辑值状态。

（1）0：低电平、逻辑 0 或逻辑非。

（2）1：高电平、逻辑 1 或逻辑真。

（3）x 或 X：不确定或未知的逻辑状态。

（4）z 或 Z：高阻态。

Verilog HDL 中的常量在上述 4 种逻辑状态中取值，其中 x 和 z 都不区分大小写。

在 Verilog HDL 语言中，用 parameter 来定义符号常量，即用 parameter 来定义一个标志符，代表一个常量。其定义格式如下：

```
parameter  参数名 1=表达式 1，参数名 2=表达式 2，…… 参数名 n=表达式 n;
```

【例】
```
parameter select=5,code=8'h56;
//分别定义参数 select 代表常数 5(十进制),参数 code 代表常量 56(十
//六进制)
```

2. 变量

在程序运行过程中，其值可以被改变的量称为变量，Verilog HDL 中的变量主要分为连线型（net）和寄存器型（register）。连线型中常用的有 wire 型，寄存器型包括 reg 型、integer 型和 time 型等。

（1）连线型。

连线型变量相当于硬件电路中的各种物理连接，其特点是输出的值紧跟输入值的变化而变化。对连线型有两种驱动方式，一种方式是在结构描述中将其

连接到一个门元件或模块的输出端；另一种方式是用持续赋值语句 assign 对其进行赋值。

连线型变量包括多种类型，见表 8-1。

表 8-1　　连 线 型 变 量

类　型	功能说明	可综合性
wire,tri	连线类型	√
wor,trior	具有线或特性的连线	
wand,triand	具有线与特性的连线	
tri1,tri0	分别为上拉电阻和下拉电阻	
supply1,supply0	分别为电源（逻辑 1）和地（逻辑 0）	√

wire 型变量常用来表示以 assign 语句赋值的组合逻辑信号。Verilog HDL 模块中的输入/输出信号类型默认时自动定义为 wire 型。wire 型信号可以用作任何表达式的输入，也可以用做“assign”语句和实例元件的输出。其取值可为 0、1、x、z。

wire 型变量的定义格式如下：

```
wire 数据名 1,数据名 2,……,数据名 n;
```

【例】 `wiar a,b;`

定义了两个宽度为一位的 wire 型变量 a 和 b。

若需要定义一个多位的 wire 型数据（如总线），可按如下方式定义：

```
wire[n-1:0] 数据名 1,数据名 2,……,数据名 n;
```

或

```
wire[n:1] 数据名 1,数据名 2,……,数据名 n;
```

【例】 我们定义 8 位宽的数据总线，16 位宽的地址总线：

```
wire[7:0] databus;          //databus 的宽度是 8 位
wire[15:0] addrbus;         //addrbus 的宽度是 16 位
```

或

```
wire[8:1] databus;          //databus 的宽度是 8 位
```

```
wire[16:1] addrbus;          //addrbus 的宽度是 16 位
```

（2）寄存器型。

寄存器型变量对应的是具有状态保持作用的电路元件，如触发器、寄存器等。

寄存器型变量与连线型变量的根本区别在于：register 型变量需要被明确地赋值，并且 register 型变量在被重新赋值前一直保持原值。在设计中必须将寄存器型变量放在过程语句（如 initial、always）中，通过过程赋值语句赋值。另外，在 always、initial 等过程块内被赋值的信号都必须定义成寄存器型。

寄存器型变量包括 4 种类型，见表 8-2。

表 8-2　　寄 存 器 型 变 量

类　型	功能说明	可综合性
reg	常用的寄存器变量	√
integer	32 位带符号整型变量	√
real	64 位带符号实型变量	
time	无符号时间变量	

integer、real 和 time 三种寄存器型变量都是纯数学的抽象描述，不对应任何具体的硬件电路，real 和 time 型变量不能被综合。

reg 型变量是最常用的一种寄存器型变量，reg 型变量的定义格式类似 wire 型，如下所示：

```
reg 数据名 1,数据名 2, ……,数据名 n;
```

【例】 `reg a,b;`

定义了两个宽度都是 1 位的 reg 型变量 a，b。

若需要定义一个多位的 reg 型向量，可按以下方式定义：

```
reg[n-1:0] 数据名 1,数据名 2, …,数据名 n;
```

或

```
reg[n:1] 数据名 1,数据名 2, …,数据名 n;
```

如下面的语句定义了 8 位宽的 reg 型向量：

```
reg[7:0] cout;
reg[8:1] cout;
```

3. 数据类型

在硬件描述语言中，数据类型是用来表示数字电路中的物理连线、数据存储和传送单元等物理量的。

Verilog HDL 中共有 19 种数据类型，包括 wire 型、reg 型、integer 型、parameter 型、large 型、medium 型、scalared 型、time 型、small 型、tri 型、trio 型、tril 型、triand 型、trior 型、trireg 型、real 型、vectored 型、wand 型和 wor 型。

最常用的是 reg 型、wire 型、integer 型和 parameter 型。

8.2.3　运算符

Verilog HDL 语言提供了丰富的运算符，有许多与 C 语言很类似，但也有许多则是完全不同的，例如拼接运算符、阻塞和非阻塞赋值运算符等。

按功能分的话，包括：算术运算符、逻辑运算符、位运算符、关系运算符、等式运算符、缩减运算符、移位运算符、条件运算符和位拼接运算符等 9 类。

按运算符所带操作数的个数来区分，可分为单目运算符、二目运算符和三目运算符。

（1）单目运算符：运算符可带一个操作数。

（2）双目运算符：运算符可带两个操作数。

（3）三目运算符：运算符可带三个操作数。

1. 算术运算符

算术运算符包括：

+ 加

– 减

* 乘

/ 除

% 求模

算术运算符都是二目运算符。符号“+”、“–”、“*”、“/”分别表示常用的加、减、乘、除四则运算；%是求模运算符，或称为求余运算符，比如“4%2”

的值为0，“4%3”的值为1。在进行整数除法运算时，结果值要略去小数部分。在进行取模运算时，结果值的符号位采用模运算式里第一个操作数的符号位。

2. 逻辑运算符

逻辑运算符包括：

&& 逻辑与

|| 逻辑或

! 逻辑非

“&&”与“||”是二目运算符，有2个操作数。“!”是单目运算符，只有1个操作数。例如，A和B的与表示为：A&&B；A和B的或表示为A||B；A的非表示为：!A。

如果操作数是1位，则逻辑运算的真值表见表8-3。

表8-3 1位的逻辑运算真值表

a b	a&&b	a||b	!a !b
1 1	1	1	0 0
1 0	0	1	0 1
0 1	0	1	1 0
0 0	0	0	1 1

如果操作数不止1位的话，则应将操作数作为一个整体来对待。即如果操作数是全0，则相当于逻辑0，但只要某一位是1，则操作数就应该整体看作逻辑1。

逻辑运算符的操作结果是1位的，要么为逻辑1，要么为逻辑0。

3. 位运算符

位运算符的作用是将两个操作数按对应位分别进行逻辑运算。Verilog HDL共有5种位运算符，包括：

~ 按位取反

& 按位与

| 按位或

^ 按位异或

^~ 、 ~^ 按位同或（符号^~与~^是等价的）

按位取反的真值表见表8-4。

按位与的真值表见表8-5。

表 8-4　　按位取反的真值表

~	结果
1	0
0	1
x	x

表 8-5　　按位与的真值表

&	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

按位或的真值表见表 8-6。

表 8-6　　按位或的真值表

|	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

按位异或的真值表见表 8-7。

表 8-7　　按位异或的真值表

^	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

按位同或的真值表见表 8-8，按位“同或”就是将 2 个操作数的相应位先进行“异或”运算，再进行“非”运算。

表 8-8　　按位同或的真值表

^~	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

位运算符中除“~”是单目运算符外，其他均为二目运算符。位运算符中的二目运算符要求对 2 个操作数的相应位进行运算操作。

4. 关系运算符

关系运算符包括：

< 小于

<= 小于或等于（其中<=操作符也用于表示信号的一种赋值操作）

> 大于

>= 大于或等于

在进行关系运算时，如果声明的关系是假，则返回值是 0；如果声明的关系是真，则返回值是 1；如果某个操作数的值不定，则关系的结果是模糊的，返回值是不定值。

5. 等式运算符

等式运算符包括：

== 等于

!= 不等于

=== 全等

!== 不全等

这 4 种运算符都是双目运算符，得到的结果是 1 位的逻辑值。如果得到 1，说明声明的关系为真；如果得到 0，说明声明的关系为假。

相等运算符（==）进行判别时：参与比较的两个操作数必须逐位相等，其相等比较的结果才为 1；如果某些位是不定态或高阻值，其相等比较得到的结果是不定值。

全等运算符（===）进行判别时：参与比较的两个操作数必须逐位相等，其相等比较的结果才为 1；如果某些位是不定态或高阻值，则对这些不定态或高阻值的位也进行比较，两个操作数必须完全一致，其结果才是 1，否则结果是 0。

例如：a=4'b10x1，b=4'10x1，那么 a==b 的结果是 x，a===b 的结果是 1。

相等运算符（==）的真值表见表 8-9。全等运算符（===）的真值表见表 8-10。

表 8-9 相等运算符（==）的真值表

==	0	1	x	Z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
Z	x	x	x	x

表 8-10　　全等运算符（===）的真值表

===	0	1	x	z
0	1	0	0	0
1	0	1	0	0
x	0	0	1	0
z	0	0	0	1

6. 缩减运算符

缩减运算符是单目运算符，它包括下面 6 种：

& 与

~& 与非

| 或

~| 或非

^ 异或

^~，~^ 同或

缩减运算符与位运算符的逻辑运算法则一样，但其运算过程不同。位运算是对操作数的相应位进行逻辑运算，操作数是几位数，则运算结果也是几位数。但缩减运算是对单个操作数进行与、或、非递推运算的，最后的运算结果是 1 位的二进制数。

【例】
```
reg[3:0] a;
b=&a;      //等效于 b=((a[0]&a[1])&a[2])&a[3];
```

【例】 若 A=5'b11001，则有

```
&A=0;      //只有 A 的各位都为 1 时,其与缩减运算的值才为 1
|A=1;      //只有 A 的各位都为 0 时,其或缩减运算的值才为 0
```

7. 移位运算符

移位运算符只有 2 位：

>> 右移

<< 左移

表示把操作数右移或左移。

【例】 若 A=4'b1100，则：

```
A>>2 的值为 4'b0011;      //将 A 右移 2 位,用 0 添补添补移出的位
```

```
A<<2 的值为 4'b0000;        //将 A 左移 2 位,用 0 添补添补移出的位
```

8. 条件运算符

条件运算符与 c 语言中的定义一样，这是一个三目运算符，对 3 个操作数进行运算，方式如下：

```
信号=条件?表达式 1:表达式 2;
```

当条件成立时，信号取表达式 1 的值，反之取表达式 2 的值。

【例】 对于图 8-3 的 2 选 1 数据选择器，可用条件运算符描述为：

```
F=SEL? A:B;   //SEL 为 1 时 F=A;SEL=0 时 F=B
```

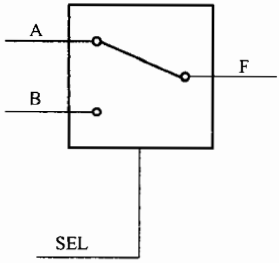


图 8-3 2 选 1 数据选择器

9. 位拼接运算符

Verilog HDL 中有一个特殊的运算符，即位拼接运算符：{}。位拼接运算符用于将两个或多个信号的某些位拼接起来。使用方式如下：

```
{信号 1 的某几位,信号 2 的某几位,……,信号 n 的某几位}
```

【例】 在进行加法运算时，可将和、进位输出拼接在一起使用。

```
output[3:0] sum;                  //sum 代表和
output cout;                      //cout 为进位输出
input[3:0] ina,inb;               //ina,inb 为两个加数
input cin;                        //cin 为进位输入
assign {cout,sum}=ina+inb+cin;    //进位、和拼接在一起
```

位拼接可以嵌套使用，还可以用重复法来简化书写。

【例】 {b,{3{a,b}}}

等同于

{b,a,b,a,b,a,b}。

8.2.4　运算符的优先级

运算符的优先级见表 8-11。我们在进行软件开发时，尽量用括号()来控制运算的优先级，这样程序的可读性好，也能有效地避免错误发生。

表 8-11　　运 算 符 的 优 先 级

运　算　符	优　先　级	运　算　符	优　先　级
! ~	高优先级	& ~&	↓
* / %	↓	^ ^~	↓
+ -	↓	| ~|	↓
<< >>	↓	&&	↓
< <= > >=	↓	||	↓
== != === !==	↓	?:	低优先级

8.3　Verilog HDL 的行为语句

Verilog HDL 有许多的行为语句，使其成为结构化和行为性的语言。Verilog HDL 语句包括：赋值语句、过程语句、块语句、条件语句、循环语句、编译预处理等，见表 8-12。

表 8-12　　Verilog HDL 语　句

类　　别	语句	可综合性
过程语句	initial	
	always	√
块语句	串行块 begin-end	√
	并行块 fork-join	
赋值语句	持续赋值 assign	√
	过程赋值=、<=	√
条件语句	if-else	√
	case	√

续表

类　别	语句	可综合性
循环语句	for	√
	repeat	
	while	
	forever	
编译预处理	`define	√
	`include	√
	`ifdef、`else、`endif	√

注　符号“√”表示该语句能够为综合工具所支持，是可综合的。

8.3.1　赋值语句

赋值语句包括持续赋值语句与过程赋值语句。

1. 持续赋值语句

assign 为持续赋值语句，主要用于对 wire 型（连线型）变量赋值。

【例】 `assign c=~(a&b);`

在上面的赋值中，a、b、c 三个变量皆为 wire 型变量，a 和 b 信号的任何变化，都将随时反映到 c 上来。

2. 过程赋值语句

过程赋值语句多用于对 reg 型变量进行赋值。过程赋值有阻塞赋值和非阻塞赋值两种方式。

（1）非阻塞赋值方式。

非阻塞赋值符号为“<=”，例如：b<=a;

非阻塞赋值在整个过程块结束时才完成赋值操作，即 b 的值并不是立刻就改变的。

（2）阻塞赋值方式。

阻塞赋值符号为“=”，例如：b=a;

阻塞赋值在该语句结束时就立即完成赋值操作，即 b 的值在该语句结束时立刻改变。

如果在一个块语句中（例如 always 块语句），有多条阻塞赋值语句，那么在前面的赋值语句没有完成之前，后面的语句就不能被执行，仿佛道路被阻塞了一样，因此称为阻塞赋值方式。

8.3.2　过程语句

Verilog HDL 中的多数过程模块都从属于以下两种过程语句：initial 和 always。

在一个模块（module）中，使用 initial 和 always 语句的次数是不受限制的。initial 语句常用于仿真中的初始化，initial 过程块中的语句仅执行一次；always 块内的语句则是不断重复执行的。

1. initial 过程语句

initial 过程语句使用格式如下：

```
initial
    begin
    语句 1;
    语句 2;
    ⋮
    语句 n;
end
```

intial 语句不带触发条件，initial 过程中的块语句沿时间轴只执行一次。initial 语句通常用于仿真模块中对激励向量的描述，或用于给寄存器变量赋初值，它是面向模拟仿真的过程语句，通常不能被逻辑综合工具所接受。

2. aIways 过程语句

always 过程语句使用格式如下：

```
always  @ (<敏感信号表达式>)
begin
//过程赋值
//if-else,case,casex,casez 选择语句
//while,repeat,for 循环
//task,function 调用
end
```

always 过程语句通常是带有触发条件的，触发条件写在敏感信号表达式中。只有当触发条件满足敏感信号表达式时，其后的“begin-end”块语句才能被执行。

3. 敏感信号表达式

所谓敏感信号表达式又称事件表达式，即当该表达式中变量的值改变时，

就会引发块内语句的执行，因此敏感信号表达式中应列出影响块内取值的所有信号。若有两个或两个以上信号时，它们之间用“or”连接。

敏感信号可以分为两种类型：一种为边沿敏感型，另一种为电平敏感型。

对于边沿敏感信号，posedge 表示时钟信号的上升沿作为触发条件，而 negedge 表示时钟信号的下降沿作为触发条件。

【例】
```
@ (a)                                  //当信号 a 的值发生改变
@ (a or b)                             //当信号 a 或信号 b 的值发生改变
@ (posedge clock)                      //当 clock 的上升沿到来时
@ (negedge clock)                      //当 clock 的下降沿到来时
@ (posedge clock or negedge reset)
                                       //当 clock 的上升沿到来或 reset
                                       //信号的下降沿到来时
```

每一个 always 过程最好只由一种类型的敏感信号来触发，而最好不要将边沿敏感型和电平敏感型信号列在一起，如下面的例子。

```
always @ (posedge clk or posedge clr)    //两个敏感信号都是边沿型
always @ (a or b)                        //两个敏感信号都是电平敏
                                         //感型
always @ (posedge clk or clr)            //不建议这样做，最好不要
                                         //将边沿敏感型和电平敏感
                                         //型信号列在一起
```

4. always 过程块实现比较复杂的组合逻辑电路

always 过程语句通常用来对寄存器类型的数据进行赋值，但 always 过程语句也可以用来设计组合逻辑。在比较复杂的情况下，使用 assign 来实现组合逻辑电路，会显得冗长且效率低下，而适当地采用 always 过程语句来设计组合逻辑，则显得简洁明了，效果也更好。

【例】 通过对下面一个简单的指令译码电路的设计分析，我们不但了解了设计者的设计思路，而且因为使用了 always 过程语句，代码看起来整齐有序，便于理解。

```
`define  add    3'd0   //常量定义
`define  minus  3'd1
`define  band   3'd2
`define  bor    3'd3
`define  bnot   3'd4
```

```
module  alu(out,opcode,a,b);
output[7:0]  out;
reg[7:0]     out;
input[2:0]   opcode;        //操作码
input[7:0]   a,b;           //操作数
always @ (opcode  or  a  or  b)   //电平敏感的 always 块
    begin
    case (opcode)
    `add:  out=a+b;         //加操作
    `minus:  out=a-b;       //减操作
    `band:  out=a&b;        //按位与
    `bor:  out=a|b;         //按位或
    `bnot:  out=~a;         //按位取反
    default:out=8'hx;       //未收到指令时,输出任意态
    endcase
end
endmodule
```

8.3.3　块语句

块语句通常用来将两条或多条语句组合在一起，用块标志 begin-end 或 fork-join 界定的一组语句。当块语句只包含一条语句时，块标志可以缺省。

1. 串行块 begin-end

串行块定义格式如下：

```
begin
    语句 1;
    语句 2;
    ⋮
    语句 n;
end
```

begin-end 串行块中的语句按串行方式顺序执行，即只有上一条语句执行完成后才执行下面的语句。全部语句执行完后才跳出该语句块。

【例】
```
begin
        regb=rega;
        regc=regb;
    end
```

第一条语句先执行 regb=rega;然后流程转到执行第二条语句 regc=regb;因为这两条语句之间没有任何时间延迟，所以 regc 的值实际就是 rega 的值。当然我们也可在语句之间控制延迟时间。

【例】 下面就是利用延迟时间产生波形。

```
parameter d=100     //声明 d 为延迟参数,延迟 100 个时间单位
reg[7:0] r;         //声明 r 为 8 位的寄存器变量
begin               //由系列延迟产生波形
    #d r=`h50;
    #d r=`hE1;
    #d r=`h00;
    #d r=`hFF;
    #d ->end_wave; //
end
```

2. 并行块 fork-join

并行块定义格式如下：

```
fork
    语句 1;
    语句 2;
    ⋮
    语句 n;
join
```

并行块 fork-join 中的所有语句是并发执行的。

【例】
```
fork
    regb=rega;
    regc=regb;
    join
```

由于 fork-join 并行块中的语句是并发执行的，在上面的 regb=rega;语句执行完成后，regb 更新为 rega 的值，而 regc 的值更新为没有改变前的 regb 值，regb 与 rega 的值是不同的。

在 Verilog HDL 中，还可以给每个块取一个名字，只需将名字加在关键字 begin 或 fork 后面即可（例如，begin test）。这样做，可以在块内定义局部变量，也允许块被其他语句调用，如被 disable 语句调用。

8.3.4　条件语句

条件语句有 if-else 语句和 case 语句两类，它们都是顺序语句，应放在 always 块内。

1. if-else 语句

if-else 语句的使用方法有以下 3 种。

```
(1) if (表达式)    语句 1;
(2) if (表达式)    语句 1;
    else                  语句 2;
(3) if (表达式 1)  语句 1;
    else  if (表达式 2)  语句 2;
    else  if (表达式 3)  语句 3;
    ⋮
    else  if (表达式 n)  语句 n;
    else  语句 n+1;
```

这 3 种方式中，“表达式”一般为逻辑表达式或关系表达式。系统对表达式的值进行判断，若为 0，x，z，按“假”处理；若为 1，按“真”处理，执行指定语句。语句如果是多句时应用“begin-end”块语句括起来。对于 if 语句的嵌套，若不清楚 if 和 else 的匹配，最好用“begin-end”语句括起来。

2. case 语句

if-else 语句只有两个分支，而处理复杂问题时往往需要多分支选择。Verilog HDL 的 case 语句是一种多分支语句，故 case 语句可用于多条件选择电路，如译码器、数据选择器、状态机及微处理器的指令译码等。case 语句有 case、casez、casex 三种。

(1) case 语句。

case 语句的格式如下：

```
case (敏感表达式)
    值 1:  语句 1;    // case 分支项
    值 2:  语句 2;
    ⋮
    值 n:  语句 n;
    default:语句 n+1;
endcase
```

当敏感表达式的值为值 1 时，执行语句 1；为值 2 时，执行语句 2，……如果敏感表达式的值与上面列出的值都不符的话，则执行 default 后面的语句。如果前面已列出了敏感表达式所有可能的取值，则 default 语句可以省略。

（2）casez 与 casex 语句。

case 语句中，敏感表达式与值 1～n 间的比较是一种全等比较，必须保证两者的对应位全等。casez 与 casex 语句是 case 语句的两种变体，在 casez 语句中，如果分支表达式某些位的值为高阻 z，那么对这些位的比较就不予考虑，因此只需关注其他位的比较结果。而在 casex 语句中，如果比较的双方有一方的某些位的值为高阻 z 或不定值 x，那么这些位的比较也都不予考虑。表 8-13 中是 case、casez 和 casex 在进行比较时的比较真值表。此外，还有另外一种标识 x 或 z 的方式，即用表示无关值的符号“?”来表示。

表 8-13　　case、casez 和 casex 的比较真值表

case	0 1 x z	casez	0 1 x z	casex	0 1 x z
0	1 0 0 0	0	1 0 0 1	0	1 0 1 1
1	0 1 0 0	1	0 1 0 1	1	0 1 1 1
x	0 0 1 0	x	0 0 1 1	x	1 1 1 1
z	0 0 0 1	z	1 1 1 1	z	1 1 1 1

3. 条件语句使用注意事项

在使用条件语句时，应注意列出所有条件分支，否则，编译器认为条件不满足时，会引进一个触发器保持原值。这一点可用于设计时序电路，例如在计数器的设计中，条件满足则加 1，否则保持不变；而在组合电路设计中，应避免这种隐含触发器的存在。当然，一般不可能列出所有分支，因为每一个变量至少有 4 种取值 0，1，z，x。为了包含所有分支，可以在 if 语句最后加上 else；在 case 语句的最后加上 default 语句。

8.3.5　循环语句

在 Verilog HDL 中存在四种类型的循环语句，用来控制语句的执行次数，这四种语句分别为：

（1）forever。连续地执行语句；多用在“initial”块中。

（2）repeat。连续执行一条语句 n 次。

（3）while。执行一条语句直到某个条件不满足。如一开始条件不满足，则一次也不执行。

（4）for。有条件的循环语句。

1. forever 语句

forever 语句的使用格式如下：

```
forever   语句;
```

或

```
forever  begin
         ⋮
         end
```

forever 循环语句连续不断地执行后面的语句或语句块，常用来产生周期性的波形，作为仿真激励信号。forever 语句一般用在 intial 过程语句中。

2. repeat 语句

repeat 语句的使用格式为：

```
repeat (循环次数表达式)  语句;
```

或

```
repeat (循环次数表达式)
begin
⋮
end
```

3. while 语句

while 语句的使用格式如下：

```
while (循环执行条件表达式)    语句;
```

或

```
while (循环执行条件表达式)
begin
⋮
end
```

while 语句在执行时，首先判断循环执行条件表达式是否为真。若为真，执行后面的语句或语句块，然后再回头判断循环执行条件表达式是否为真；若为

真的话，再执行一遍后面的语句，如此不断，直到条件表达式不为真。因此在执行语句中，必须有一条改变循环执行条件表达式的值的语句。

4. for语句

for语句的使用格式如下所示：

```
for (循环变量赋初值;循环结束条件;循环变量增值)  语句;
```

for语句执行过程如下：

（1）循环变量赋初值。

（2）判断循环结束条件。若为“真”，执行语句，然后转到c；若为“假”则退出for语句。

（3）执行循环变量增值语句，转回到b继续执行。

8.3.6 编译预处理

Verilog HDL语言和C语言一样，也提供了编译预处理功能。在编译时，通常先进行“预处理”，然后再将预处理的结果和源程序一起进行编译。

为了与其他语句区别开来，编译预处理语句以符号“`”开头。Verilog HDL提供了二十几条编译预处理语句，常用的有：`define、`include、`ifdef、`else、`endif等。

1. 宏替换`define

`define语句用于将一个指定的标识符（或称为宏名）来代替一个复杂的字符串，其使用格式为：

```
`define  宏名(标识符)  字符串
```

【例】 `define sum ina+inb

在上面的语句中，用简单的宏名sum来代替了一个复杂的表达式“ina+inb”。采用了这样的定义形式后，在后面的程序中，就可以直接用sum来代替表达式“ina+inb”。

2. 文件包含`include

文件包含是指一个源文件将另一个源文件的全部内容包含进来。`include用来实现文件包含的操作。其格式为：

```
`include  "文件名"
```

使用`include语句时应注意以下几点：

（1）一个`include 语句只能指定一个被包含的文件。如要包含 n 个文件，就要用 n 个`include 语句。

（2）`include 语句可以出现在源程序的任何地方。被包含的文件若与包含文件不在同一个子目录下，必须指明其路径名。

（3）文件包含允许多重包含，比如文件 1 包含文件 2，文件 2 又包含文件 3 等。

3. 条件编译`ifdef、`else、`endif

条件编译命令`ifdef、`else、`endif 可以仅对程序中指定的部分内容进行编译，这 3 个命令的使用形式如下：

（1）
```
`ifdef  宏名(标识符)
    语句块
    `endif
```

这种表达式的意思是：如果宏名在程序中被定义过（用`define 语句定义），则下面的语句块参与源文件的编译，否则，该语句块将不参与源文件的编译。

（2）
```
`ifdef 宏名(标识符)
        语句块 1
    `else  语句块 2
    `endif
```

这种表达式的意思是：如果宏名在程序中被定义过（用`define 语句定义），则语句块 1 将被编译到源文件中，否则，语句块 2 将被编译到源文件中。

8.3.7　任务和函数

任务和函数的关键字分别是 task 和 function，利用任务和函数可以把一个大的程序模块分解成许多小的任务和函数，以方便调试，并且能使写出的程序结构更清晰。

1. 任务

任务定义的格式如下：

```
task <任务名>;              //注意无端口列表
端口及数据类型声明语句;
其他语句;
endtask
```

任务调用的格式如下：

```
<任务名> (端口 1, 端口 2, ……端口 n)
```

注意：任务调用时和任务定义时的端口变量应一一对应。

【例】 任务定义时：

```
task test;
input in1,in2;
output out1,out2;
#1  out1=in1&in2;
#1  out2=in1|in2;
endtask
```

任务调用时：

```
test(data1,data2,code1,code2);
```

调用任务时 test 时，变量 data1 和 data2 的值赋给 in1 和 in2，而任务完成后，out1 和 out2 的值赋给了 code1 和 code2。

在使用任务时，应特别注意以下几点：

（1）任务的定义与调用必须在同一个 module 模块内。

（2）定义任务时，没有端口名列表，但需要紧接着进行输入、输出端口和数据类型的说明。

（3）当任务被调用时，任务被激活。任务的调用与模块调用一样通过任务名调用实现，调用时，需列出端口名列表，端口名的排序和类型必须与任务定义时相一致。

（4）一个任务可以调用别的任务和函数，可以调用的任务和函数个数不受限制。

2. 函数

函数的目的是返回一个值，以用于表达式的计算。

函数的定义格式为：

```
function <返回值位宽或类型说明>  (函数名);
    端口声明;
    局部变量定义;
    其他语句;
endfunction
```

上面的定义格式中，<返回值位宽或类型说明>是一个可选项，如果缺省，则返回值为 1 位寄存器类型的数据。

3. 函数的调用

函数的调用是通过将函数作为表达式中的操作数来实现的。

调用格式如下：

```
<函数名> (<表达式> , <表达式>);
```

4. 任务与函数的区别

任务与函数的区别见表 8-14。

表 8-14　　任务与函数的区别

比较项目	任务（task）	函数（function）
输入与输出	可有任意个各种类型的参数	至少有一个输入，不能将 inout 类型作为输出
调用	任务只可在过程语句中调用，不能在持续赋值语句 assign 中调用	函数可作为表达式中的一个操作数来调用，在过程赋值和持续赋值语句中均可以调用
定时事件控制（#、@ 和 wait）	任务可以包含定时和事件控制语句	函数不能包含这些语句
调用其他任务和函数	任务可调用其他任务和函数	函数可调用其他函数，但不可以调用其他任务
返回值	任务不能向表达式返回值	函数向调用它的表达式返回一个值

8.4　Verilog HDL 数字逻辑单元结构的设计

Verilog HDL 是一种专门用于数字逻辑设计的语言，它既是一种行为描述语言，也是一种结构描述语言，也就是说，既可以用电路的功能描述，也可以用元器件和它们之间的连接来建立所设计电路的 Verilog HDL 模型。Verilog HDL 程序有三种描述设计的方法，分别是结构描述方式、数据流描述方式和行为描述方式。

8.4.1　结构描述方式

在 Verilog HDL 程序设计中可通过以下方式来描述电路的结构：

（1）调用 Verilog HDL 内置门元件（门级结构描述）。

（2）调用开关级元件（开关级结构描述）。

（3）用户自定义元件 UDP（也在门级）。

除此之外，在多层次结构电路的设计中，不同模块间的调用也可以认为是结构描述的。

1. Verilog HDL 内置门元件

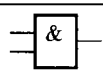
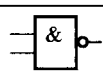
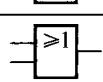
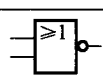
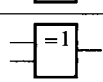
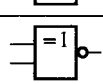
Verilog HDL 内置 26 个基本元件，其中 14 个是门级元件，12 个为开关级元件，见表 8-15。

表 8-15　　Verilog HDL 内置 26 个基本元件

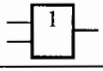
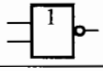
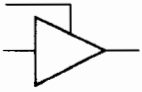
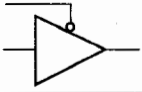
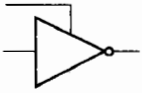
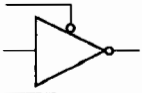
类　别		元　件
基本门	多输入门	and,nand,or,nor,xor,xnor
	多输出门	buf,not
三态门	允许定义驱动强度	bufif0, bufif1, notif0, notif1
MOS 开关	无驱动强度	nmos,pmos,cmos,mmos,rpmos,rcmos
双向开关	无驱动强度	tran,tranif0,tranif1
	无驱动强度	rtran,rtranif0,rtranif1
上拉、下拉电阻	允许定义驱动强度	pullup,pulldown

这里我们重点介绍门级元件及其结构描述。Verilog HDL 内置的门元件见表 8-16。

表 8-16　　Verilog HDL 内置的门元件

类　型	关 键 字	符　号	门 名 称
多输入门	and		与门
	nand		与非门
	or		或门
	nor		或非门
	xor		异或门
	xnor		异或非门

续表

类　型	关 键 字	符　号	门 名 称
多输出门	buf		缓冲器
	not		非门
三态门	bufif1		高电平使能三态缓冲器
	bufif0		低电平使能三态缓冲器
	notif1		高电平使能三态非门
	notif0		低电平使能三态非门

为了方便理解与查找，我们将各种基本的逻辑门真值表进行了分类：表 8-17 是与门的真值表；表 8-18 是与非门的真值表；表 8-19 是或门的真值表；表 8-20 是或非门的真值表；表 8-21 是异或门的真值表；表 8-22 是异或非门的真值表；表 8-23 是三态门 bufif1 的真值表；表 8-24 是三态门 bufif0 的真值表；表 8-25 是三态门 notif1 的真值表；表 8-26 是三态门 notif0 的真值表。

表 8-17　　与 门 的 真 值 表

and	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

表 8-18　　与 非 门 的 真 值 表

nand	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x

表 8-19　　或门的真值表

or	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

表 8-20　　或非门的真值表

nor	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

表 8-21　　异或门的真值表

xor	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

表 8-22　　异或非门的真值表

xnor	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

表 8-23　　三态门 bufif1 的真值表

bufif1		使能端			
		0	1	x	z
输入端	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	x	x	x

注　表中 L 代表 0 或 z，H 代表 1 或 z。

表 8-24　　三态门 bufif0 的真值表

bufif0		使能端			
		0	1	x	z
输入端	0	0	z	L	L
	1	1	z	H	H
	x	x	z	x	x
	z	x	z	x	x

注　表中 L 代表 0 或 z，H 代表 1 或 z。

表 8-25　　三态门 notif1 的真值表

notif1		使能端			
		0	1	x	z
输入端	0	z	1	H	H
	1	z	0	L	L
	x	z	x	x	x
	z	z	x	x	x

注　表中 L 代表 0 或 z，H 代表 1 或 z。

表 8-26　　三态门 notif0 的真值表

notif0		使能端			
		0	1	x	z
输入端	0	1	z	H	H
	1	0	z	L	L
	x	x	z	x	x
	z	x	z	x	x

注　表中 L 代表 0 或 z，H 代表 1 或 z。

2. 门元件的调用

调用门元件的格式为：

门元件名字 <例化的门名字> (< 端口列表>)

其中，普通门的端口列表按下面的顺序列出：

(输出,输入 1,输入 2,输入 3,……);

【例】 nand U1 (out,in1,in2); //二输入端与非门,名字为U1

对于三态门，则按如下顺序列出输入、输出端口：

(输出,输入,使能控制端);

【例】 bufif1 U2 (out,in,enable); //高电平使能的三态门

对于三态门 buf 和 not 两种元件的调用时，需要注意：它们允许有多个输出，但只能有一个输入。

3. 门级结构描述

门级结构描述是指采用 Verilog HDL 内置门实例语句的描述方法进行设计。

【例】 对于二输入端的与非门，其设计描述如下：

```
module NAND2_G(A,B,F);
input A,B;
output F;
nand U1(F,A,B);
endmodule
```

8.4.2 数据流描述方式

一般使用持续赋值语句描述数据流程的运动路径、运动方向和运动结果的设计方法，称为数据流描述方法。

【例】
```
module NAND2_G(A,B,F);          //模块声明及输入输出端口列表
input A,B;                      //定义输入端口
output F;                       //定义输出端口
assign F=~(A&B);                //数据流描述
endmodule                       //模块结束
```

对于表达式 assign F=~(A&B)；右边的操作数 A、B 无论何时发生变化，都会引起表达式值的重新计算，并将重新计算后的值赋予左边的网线变量 F。

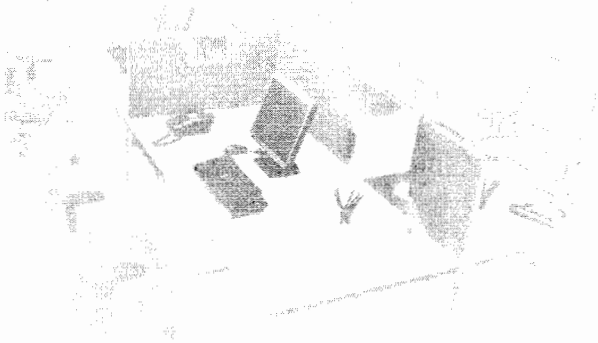
8.4.3 行为描述方式

前面我们介绍的硬件电路的结构描述主要侧重于表示一个电路由哪些基本元件组成，以及这些基本元件的相互连接关系。硬件电路的行为描述则主要反映该电路输入、输出信号间的相互关系。行为描述方式一般采用 initial 语句（此

语句只执行一次，一般用于初始化）或 always 语句（此语句重复执行）来描述逻辑功能。行为描述方式既适合于设计时序逻辑电路，也适合于设计组合逻辑电路的设计。

结构级的描述在进行仿真时要优于行为描述，而行为描述在进行综合时则更优越一些。在电路的规模较大或者需要描述复杂的时序关系时，使用行为描述方式会更有效。

例如，我们入门的第一个实验程序，就是采用行为描述方式设计的。



第9章

基本逻辑门电路的实践

9.1 缓 冲 器 实 践

9.1.1 数据流描述设计的缓冲器

缓冲器电路符号如图 9-1 所示。真值表见表 9-1。

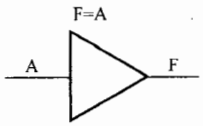


图 9-1 缓冲器电路符号

表 9-1 缓 冲 器 真 值 表

输 入	输 出
A	F
0	0
1	1

在 E 盘中建立一个文件名为“BUF_S1”的文件夹，然后新建一个“BUF_G”的新项目，输入以下的源代码并保存为“BUF_S1.v”。

```
module BUF_S1(A,F);
output F;
```

```
input A;
assign F=A;
endmodule
```

根据第 5 章“表 5-1　XC95108 引脚号与外部器件的连接关系”，得到表 9-2 所示的引脚分配表。项目编译通过后，可根据需要进行仿真，图 9-2 为仿真波形。最后将*.jed 文件下载到 XC95108 芯片中。

表 9-2　　缓 冲 器 引 脚 分 配

引脚名	引脚号	输入或输出	板上丝印符号
A	41	Input	S0
F	31	Output	LED0

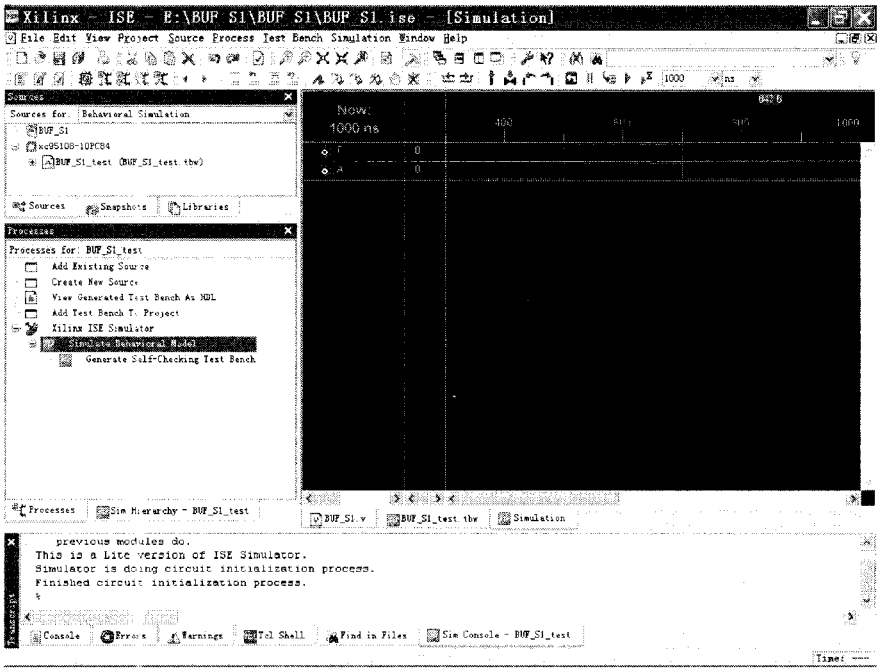


图 9-2　缓冲器仿真波形

在 MCU&CPLD DEMO 试验板的左下角，有一个 4 位的拨码开关 SW，标示 S0 的开关拨上时（ON），缓冲器输入端（XC95108 的 41 脚）为低电平，这时缓冲器输出端（XC95108 的 31 脚）也为低电平，LED0 亮；标示 S0 的开关拨下时（OFF），缓冲器输入端为高电平，这时缓冲器输出端也为高电平，LED0

```
input A;
assign F=A;
endmodule
```

根据第 5 章“表 5-1 XC95108 引脚号与外部器件的连接关系”，得到表 9-2 所示的引脚分配表。项目编译通过后，可根据需要进行仿真，图 9-2 为仿真波形。最后将*.jed 文件下载到 XC95108 芯片中。

表 9-2 缓 冲 器 引 脚 分 配

引脚名	引脚号	输入或输出	板上丝印符号
A	41	Input	S0
F	31	Output	LED0

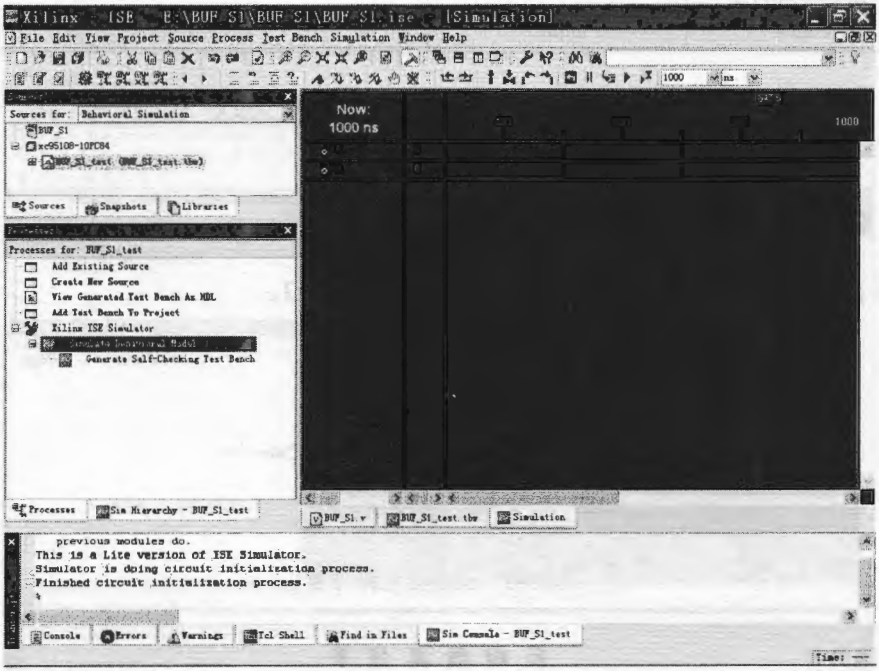


图 9-2 缓冲器仿真波形

在 MCU&CPLD DEMO 试验板的左下角，有一个 4 位的拨码开关 SW，标示 S0 的开关拨上时（ON），缓冲器输入端（XC95108 的 41 脚）为低电平，这时缓冲器输出端（XC95108 的 31 脚）也为低电平，LED0 亮；标示 S0 的开关拨下时（OFF），缓冲器输入端为高电平，这时缓冲器输出端也为高电平，LED0

灭。从而实现了缓冲器的控制逻辑。

9.1.2 门级结构描述设计的缓冲器

除了采用数据流描述方法设计缓冲器外，我们也可以使用门级结构描述来进行设计。除源代码不同外，设计的其他过程完全相同，这里不再赘述（以下同）。

源代码为：

```
module BUF_S2(A,F);
output F;
input A;
buf U1(F,A);
endmodule
```

9.2 反相器（非门）实践

9.2.1 数据流描述设计的反相器

反相器电路符号如图 9-3 所示。真值表见表 9-3。

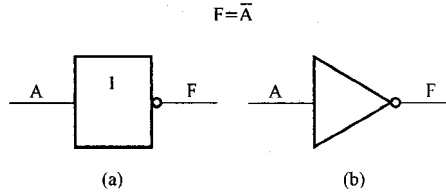


图 9-3 反相器电路符号

表 9-3 反 相 器 真 值 表

输 入	输 出
A	F
0	1
1	0

在 E 盘中先建立一个文件名为“NOT_S1”的文件夹，然后建立一个“NOT_S1”

的新项目，输入以下的源代码并保存为“NOT_S1.v”。

```
module NOT_S1(A,F);
output F;
input A;
assign F=~A;
endmodule
```

反相器的引脚分配见表 9-4。项目编译通过后，可根据需要进行仿真，图 9-4 为仿真波形。最后将*.jed 文件下载到 XC95108 芯片中。

表 9-4　　反 相 器 引 脚 分 配

引脚名	引脚号	输入或输出	板上丝印符号
A	41	Input	S0
F	31	Output	LED0

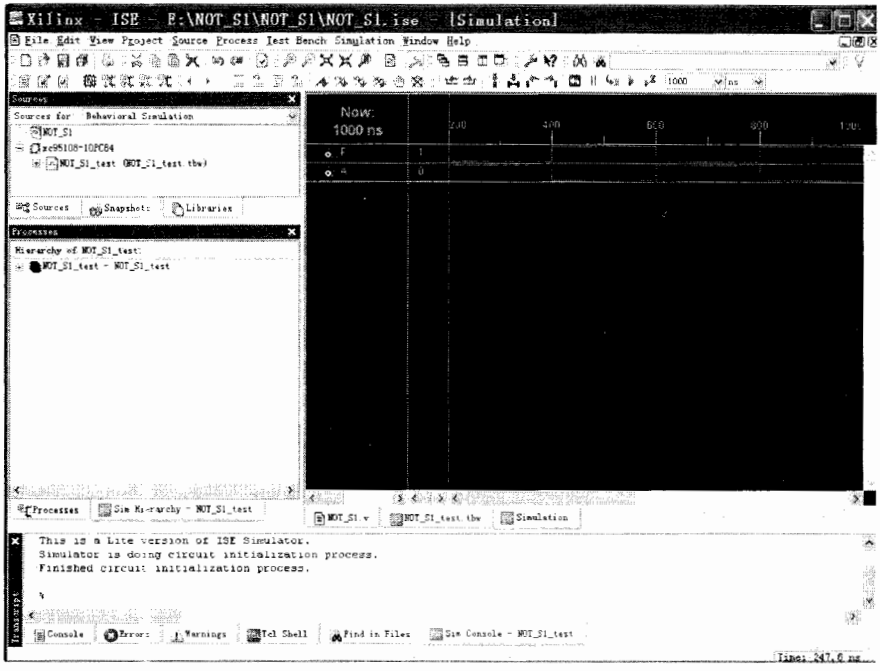


图 9-4　反相器仿真波形

在 MCU&CPLD DEMO 试验板上，我们将 4 位拨码开关的 S0 的开关拨上时（ON），XC95108 的 41 脚输入低电平，这时 LED0 灭（XC95108 的 31 脚输

的新项目，输入以下的源代码并保存为“NOT_S1.v”。

```
module NOT_S1(A,F);
output F;
input A;
assign F=~A;
endmodule
```

反相器的引脚分配见表 9-4。项目编译通过后，可根据需要进行仿真，图 9-4 为仿真波形。最后将*.jed 文件下载到 XC95108 芯片中。

表 9-4　　反相器引脚分配

引脚名	引脚号	输入或输出	板上丝印符号
A	41	Input	S0
F	31	Output	LED0

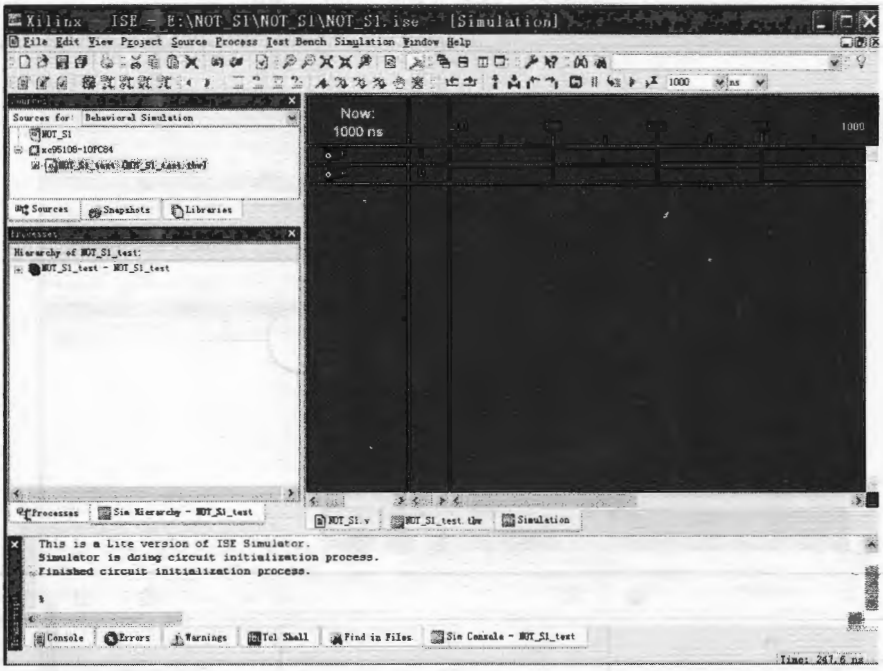


图 9-4　反相器仿真波形

在 MCU&CPLD DEMO 试验板上，我们将 4 位拨码开关的 S0 的开关拨上时（ON），XC95108 的 41 脚输入低电平，这时 LED0 灭（XC95108 的 31 脚输

出高电平)；S0 的开关拨下时(OFF)，XC95108 的 41 脚输入高电平，这时 LED0 亮（XC95108 的 31 脚输出低电平)。实现了反相器（非门）的控制逻辑。

9.2.2 门级结构描述设计的反相器

除了采用数据流描述方法设计反相器外，我们也可以使用门级结构描述来进行设计。除源代码不同外，设计的其他过程完全相同。

源代码为：

```
module NOT_S2(A,F);
output F;
input A;
not U1(F,A);
endmodule
```

9.3 与 门 实 践

9.3.1 数据流描述设计的与门

与门电路符号如图 9-5 所示。真值表见表 9-5。

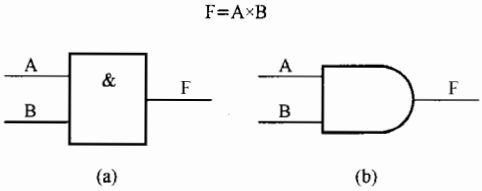


图 9-5 与门电路符号

表 9-5 与 门 真 值 表

输 入		输 出
A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

在 E 盘中先建立一个文件名为“AND2_S1”的文件夹，然后建立一个“AND2_S1”的新项目，输入以下的源代码并保存为“AND2_S1.v”。

```
module AND2_S1(A,B,F);
output F;
input A,B;
assign F=A&B;
endmodule
```

与门的引脚分配见表 9-6。项目编译通过后，可根据需要进行仿真，图 9-6 为仿真波形。最后将*.jed 文件下载到 XC95108 芯片中。

表 9-6　　与 门 引 脚 分 配

引脚名	引脚号	输入或输出	板上丝印符号
A	41	Input	S0
B	40	Input	S1
F	31	Output	LED0

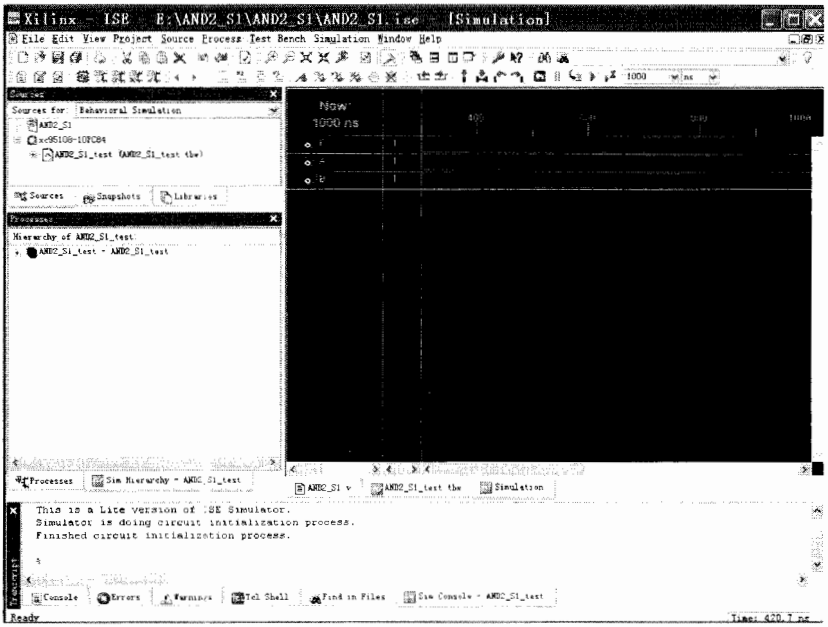


图 9-6　与门仿真波形

在 MCU&CPLD DEMO 试验板上，我们将 4 位拨码开关的 S1S0 的开关拨

在 E 盘中先建立一个文件名为“AND2_S1”的文件夹，然后建立一个“AND2_S1”的新项目，输入以下的源代码并保存为“AND2_S1.v”。

```
module AND2_S1(A,B,F);
output F;
input A,B;
assign F=A&B;
endmodule
```

与门的引脚分配见表 9-6。项目编译通过后，可根据需要进行仿真，图 9-6 为仿真波形。最后将*.jed 文件下载到 XC95108 芯片中。

表 9-6　　与 门 引 脚 分 配

引脚名	引脚号	输入或输出	板上丝印符号
A	41	Input	S0
B	40	Input	S1
F	31	Output	LED0

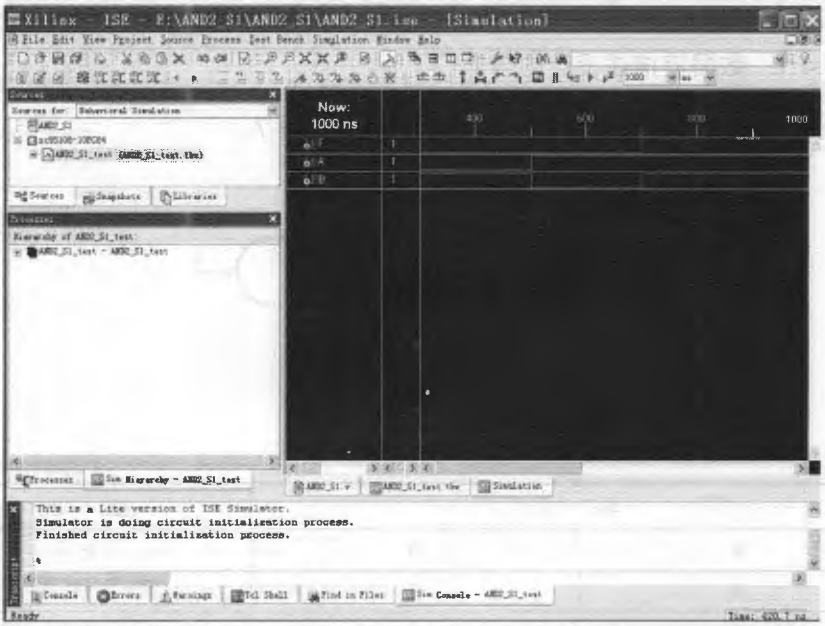


图 9-6　与门仿真波形

在 MCU&CPLD DEMO 试验板上，我们将 4 位拨码开关的 S1S0 的开关拨

下时（OFF），XC95108的41、40脚输入高电平，这时LED0灭（XC95108的31脚输出高电平）；S1S0的任一位或两位同时拨上时（ON），XC95108的41、40脚有一脚或两脚输入低电平，这时LED0亮（XC95108的31脚输出低电平）。实现了与门的控制逻辑。

9.3.2 门级结构描述设计的与门

除了采用数据流描述方法设计与门外，我们也可以使用门级结构描述来进行设计。除源代码不同外，设计的其他过程完全相同。

源代码为：

```
module AND2_S2(A,B,F);
output F;
input A,B;
and U1(F,A,B);
endmodule
```

9.4 与非门实践

9.4.1 数据流描述设计的与非门

与非门电路符号如图9-7所示。真值表见表9-7。

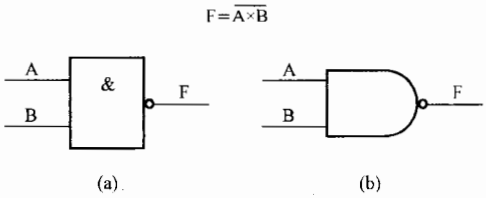


图9-7 与非门电路符号

表9-7 与非门真值表

输入		输出
A	B	F
0	0	1
0	1	1
1	0	1
1	1	0

在 E 盘中先建立一个文件名为“NAND2_S1”的文件夹，然后建立一个“NAND2_S1”的新项目，输入以下的源代码并保存为“NAND2_S1.v”。

```
module NAND2_S1(A,B,F);
output F;
input A,B;
assign F=~(A&B);
endmodule
```

与非门的引脚分配见表 9-8。项目编译通过后，可根据需要进行仿真，图 9-8 为仿真波形。最后将*.jed 文件下载到 XC95108 芯片中。

表 9-8　　与非门引脚分配

引脚名	引脚号	输入或输出	板上丝印符号
A	41	Input	S0
B	40	Input	S1
F	31	Output	LED0

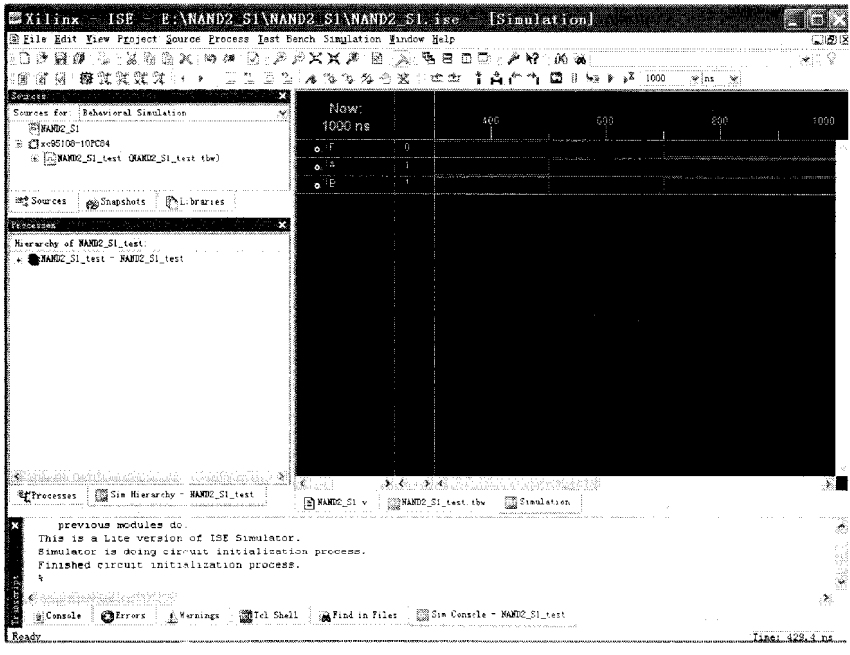


图 9-8　与非门仿真波形

在 MCU&CPLD DEMO 试验板上，我们将 4 位拨码开关的 S1S0 的开关拨

在 E 盘中先建立一个文件名为“NAND2_S1”的文件夹，然后建立一个“NAND2_S1”的新项目，输入以下的源代码并保存为“NAND2_S1.v”。

```
module NAND2_S1(A,B,F);
output F;
input A,B;
assign F=~(A&B);
endmodule
```

与非门的引脚分配见表 9-8。项目编译通过后，可根据需要进行仿真，图 9-8 为仿真波形。最后将*.jed 文件下载到 XC95108 芯片中。

表 9-8　　与非门引脚分配

引脚名	引脚号	输入或输出	板上丝印符号
A	41	Input	S0
B	40	Input	S1
F	31	Output	LED0

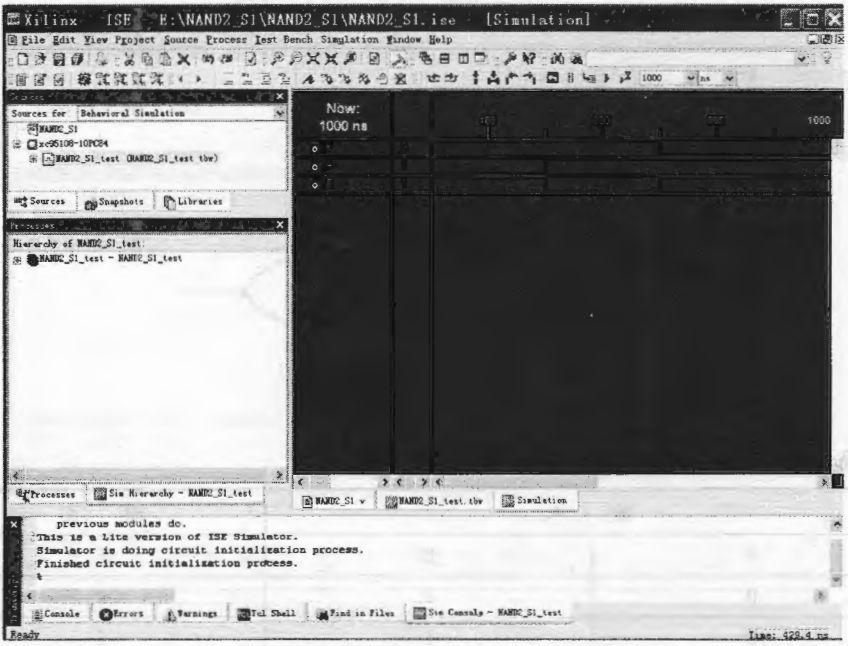


图 9-8　与非门仿真波形

在 MCU&CPLD DEMO 试验板上，我们将 4 位拨码开关的 S1S0 的开关拨

下时（OFF），XC95108的41、40脚输入高电平，这时LED0亮（XC95108的31脚输出低电平）；S1S0的任一位或两位同时拨上时（ON），XC95108的41、40脚有一脚或两脚输入低电平，这时LED0灭（XC95108的31脚输出高电平）。实现了与非门的控制逻辑。

9.4.2 门级结构描述设计的与非门

除了采用数据流描述方法设计与非门外，我们也可以使用门级结构描述来进行设计。除源代码不同外，设计的其他过程完全相同。

源代码为：

```
module NAND2_S2(A,B,F);
output F;
input A,B;
nand U1(F,A,B);
endmodule
```

9.5 或 门 实 践

9.5.1 数据流描述设计的或门

或门电路符号如图9-9所示。真值表见表9-9。

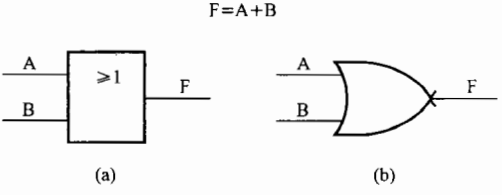


图9-9 或门电路符号

表9-9 或 门 真 值 表

输 入		输 出
A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

在 E 盘中先建立一个文件名为“OR2_S1”的文件夹，然后建立一个“OR2_S1”的新项目，输入以下的源代码并保存为“OR2_S1.v”。

```
module OR2_S1(A,B,F);
output F;
input A,B;
assign F=A|B;
endmodule
```

或门的引脚分配见表 9-10。项目编译通过后，可根据需要进行仿真，图 9-10 为仿真波形。最后将*.jed 文件下载到 XC95108 芯片中。

表 9-10　　或门引脚分配

引脚名	引脚号	输入或输出	板上丝印符号
A	41	Input	S0
B	40	Input	S1
F	31	Output	LED0

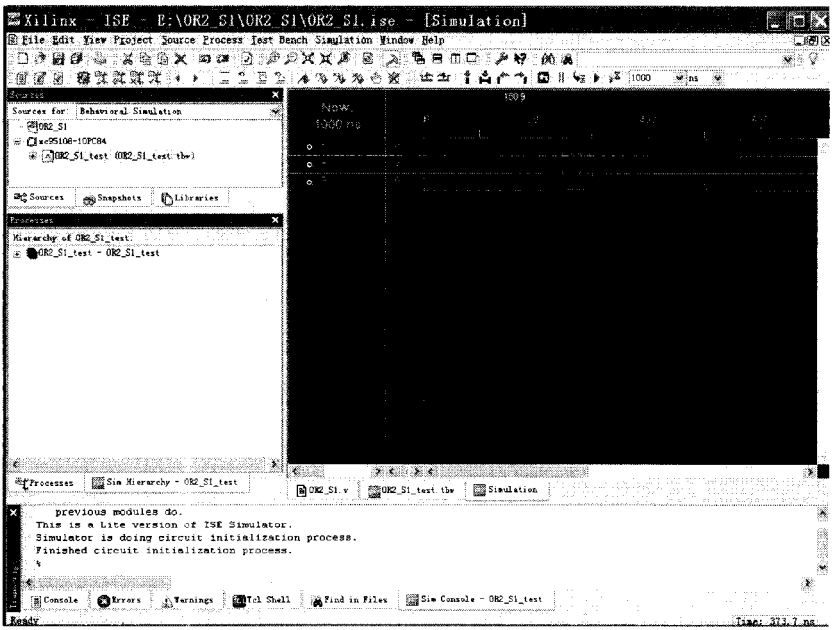


图 9-10　或门仿真波形

在 MCU&CPLD DEMO 试验板上，我们将 4 位拨码开关的 S1S0 的开关拨下时（OFF），XC95108 的 41、40 脚输入高电平，这时 LED0 灭（XC95108 的

在 E 盘中先建立一个文件名为“OR2_S1”的文件夹，然后建立一个“OR2_S1”的新项目，输入以下的源代码并保存为“OR2_S1.v”。

```
module OR2_S1(A,B,F);
output F;
input A,B;
assign F=A|B;
endmodule
```

或门的引脚分配见表 9-10。项目编译通过后，可根据需要进行仿真，图 9-10 为仿真波形。最后将*.jed 文件下载到 XC95108 芯片中。

表 9-10　　或门引脚分配

引脚名	引脚号	输入或输出	板上丝印符号
A	41	Input	S0
B	40	Input	S1
F	31	Output	LED0

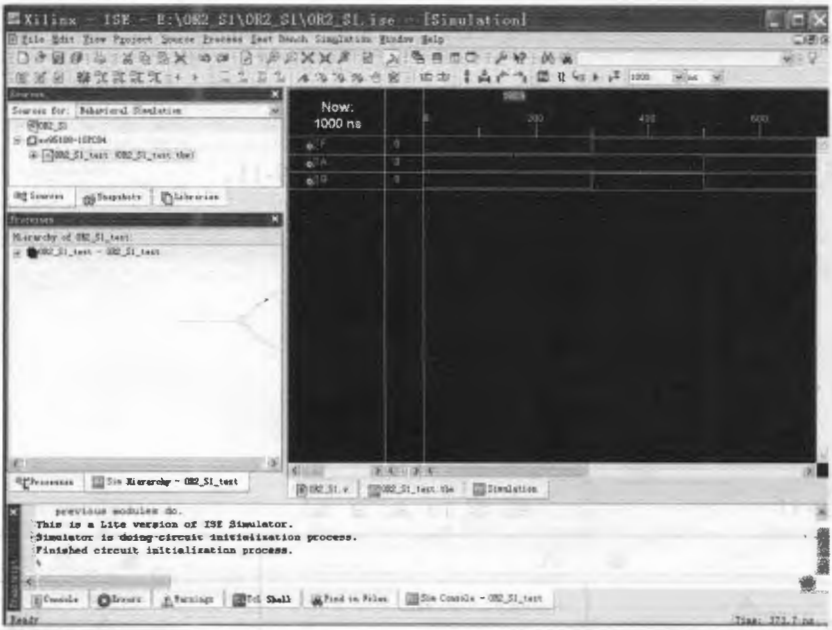


图 9-10　或门仿真波形

在 MCU&CPLD DEMO 试验板上，我们将 4 位拨码开关的 S1S0 的开关拨下时（OFF），XC95108 的 41、40 脚输入高电平，这时 LED0 灭（XC95108 的

31 脚输出高电平)；S1S0 的任一位拨上时（ON），XC95108 的 41、40 脚有一脚输入低电平，LED0 还是灭（XC95108 的 31 脚输出高电平)；S1S0 两位同时拨上时(ON)，XC95108 的 41、40 脚同时输入低电平，这时 LED0 亮(XC95108 的 31 脚输出低电平)。实现了或门的控制逻辑。

9.5.2 门级结构描述设计的或门

除了采用数据流描述方法设计或门外，我们也可以使用门级结构描述来进行设计。除源代码不同外，设计的其他过程完全相同。

源代码为：

```
module OR2_S2(A,B,F);
output F;
input A,B;
or U1(F,A,B);
endmodule
```

9.6 或非门实践

9.6.1 数据流描述设计的或非门

或非门电路符号如图 9-11 所示。真值表见表 9-11。

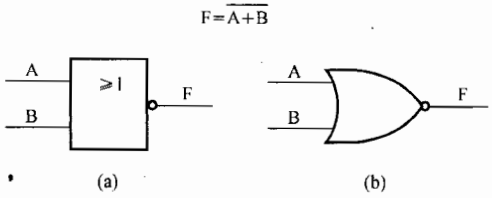


图 9-11 或非门电路符号

表 9-11 或非门真值表

输入		输出
A	B	F
0	0	1
0	1	0
1	0	0
1	1	0

在 E 盘中先建立一个文件名为“NOR2_S1”的文件夹，然后建立一个“NOR2_S1”的新项目，输入以下的源代码并保存为“NOR2_S1.v”。

```
module NOR2_S1(A,B,F);
output F;
input A,B;
assign F=~(A|B);
endmodule
```

或非门引脚分配见表 9-12。项目编译通过后，可根据需要进行仿真，图 9-12 为仿真波形。最后将*.jed 文件下载到 XC95108 芯片中。

表 9-12　　或 非 门 引 脚 分 配

引脚名	引脚号	输入或输出	板上丝印符号
A	41	Input	S0
B	40	Input	S1
F	31	Output	LED0

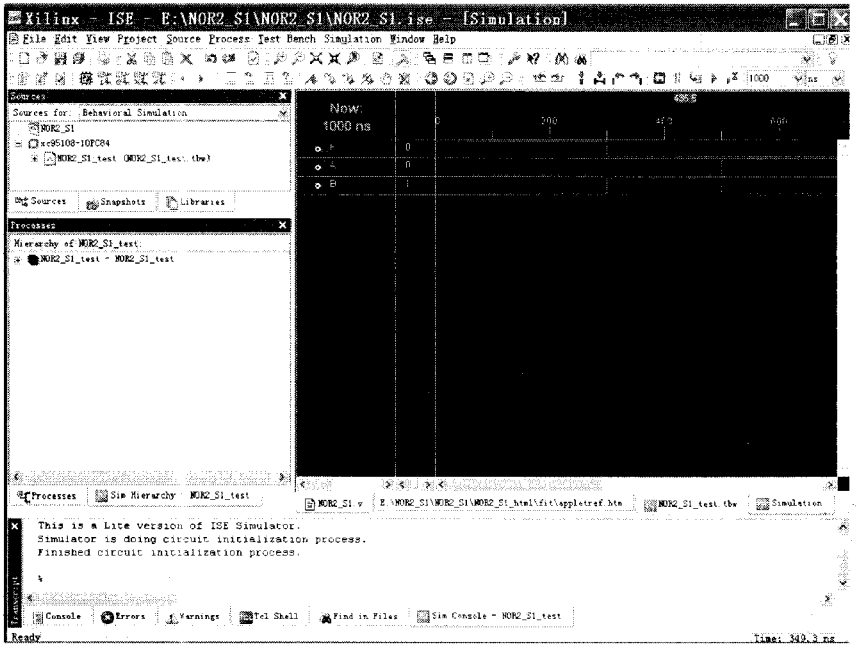


图 9-12　或非门仿真波形

在 MCU&CPLD DEMO 试验板上，我们将 4 位拨码开关的 S1S0 的开关拨

在 E 盘中先建立一个文件名为“NOR2_S1”的文件夹，然后建立一个“NOR2_S1”的新项目，输入以下的源代码并保存为“NOR2_S1.v”。

```
module NOR2_S1(A,B,F);
output F;
input A,B;
assign F=~(A|B);
endmodule
```

或非门引脚分配见表 9-12。项目编译通过后，可根据需要进行仿真，图 9-12 为仿真波形。最后将*.jed 文件下载到 XC95108 芯片中。

表 9-12　　或非门引脚分配

引脚名	引脚号	输入或输出	板上丝印符号
A	41	Input	S0
B	40	Input	S1
F	31	Output	LED0

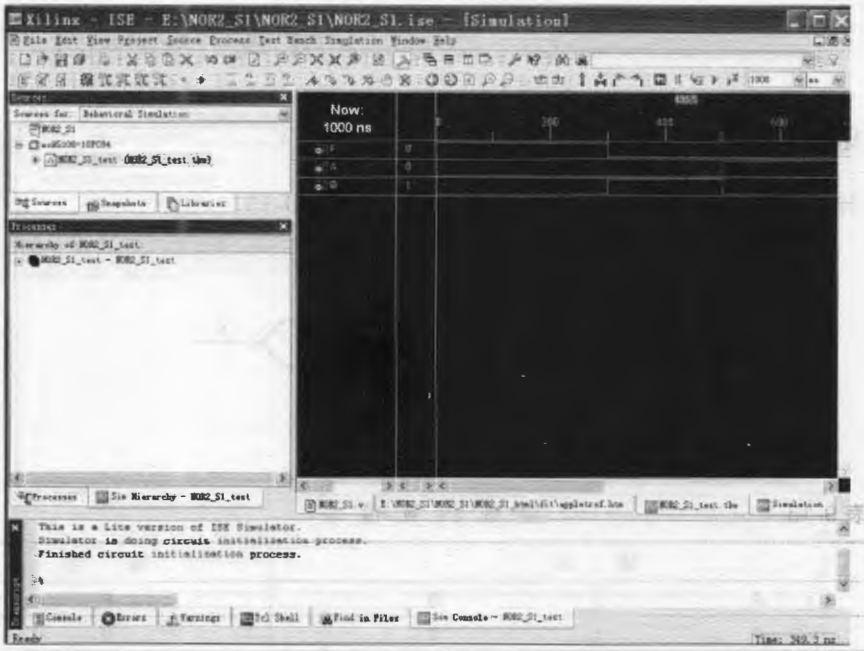


图 9-12　或非门仿真波形

在 MCU&CPLD DEMO 试验板上，我们将 4 位拨码开关的 S1S0 的开关拨

下时（OFF），XC95108的41、40脚输入高电平，这时LED0亮（XC95108的31脚输出低电平）；S1S0的任一位拨上时（ON），XC95108的41、40脚有一脚输入低电平，LED0还是亮（XC95108的31脚输出低电平）；S1S0两位同时拨上时（ON），XC95108的41、40脚同时输入低电平，这时LED0灭（XC95108的31脚输出高电平）。实现了或非门的控制逻辑。

9.6.2 门级结构描述设计的或非门

除了采用数据流描述方法设计或非门外，我们也可以使用门级结构描述来进行设计。除源代码不同外，设计的其他过程完全相同。

源代码为：

```
module NOR2_S2(A,B,F);
output F;
input A,B;
nor U1(F,A,B);
endmodule
```

9.7 异或门实践

9.7.1 数据流描述设计的异或门

异或门电路符号如图9-13所示。真值表见表9-13。

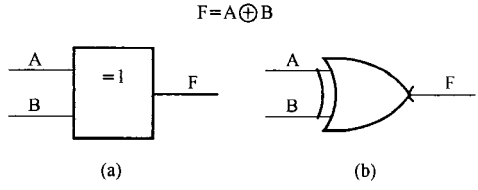


图9-13 异或门电路符号

表9-13 异或门真值表

输入		输出
A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

在 E 盘中先建立一个文件名为“XOR2_S1”的文件夹，然后建立一个“XOR2_S1”的新项目，输入以下的源代码并保存为“XOR2_S1.v”。

```
module XOR2_S1(A,B,F);
output F;
input A,B;
assign F=A^B;
endmodule
```

异或门引脚分配见表 9-14。项目编译通过后，可根据需要进行仿真，图 9-14 为仿真波形。最后将*.jed 文件下载到 XC95108 芯片中。

表 9-14　　异或门引脚分配

引脚名	引脚号	输入或输出	板上丝印符号
A	41	Input	S0
B	40	Input	S1
F	31	Output	LED0

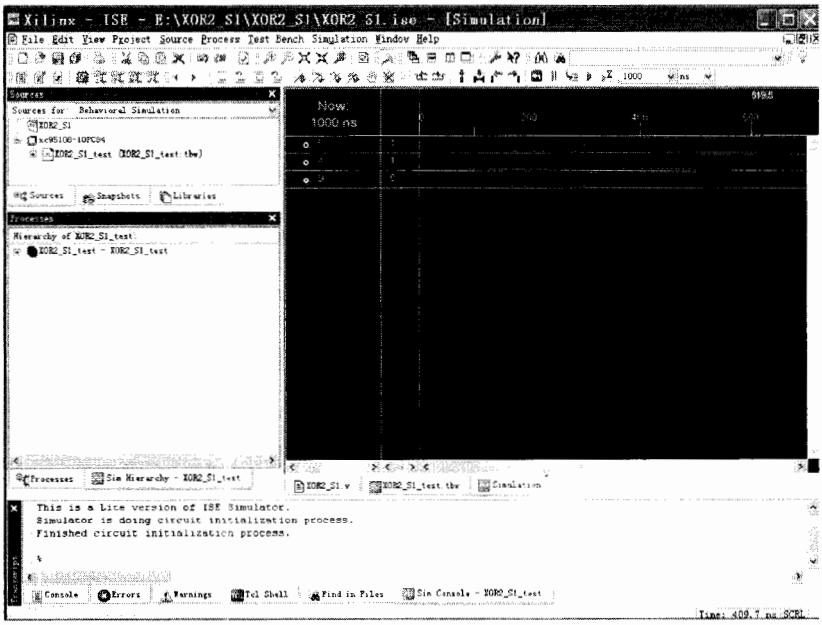


图 9-14　异或门仿真波形

在 MCU&CPLD DEMO 试验板上，我们将 4 位拨码开关的 S1S0 的开关同

在 E 盘中先建立一个文件名为“XOR2_S1”的文件夹，然后建立一个“XOR2_S1”的新项目，输入以下的源代码并保存为“XOR2_S1.v”。

```
module XOR2_S1(A,B,F);
output F;
input A,B;
assign F=A^B;
endmodule
```

异或门引脚分配见表 9-14。项目编译通过后，可根据需要进行仿真，图 9-14 为仿真波形。最后将*.jed 文件下载到 XC95108 芯片中。

表 9-14　　异或门引脚分配

引脚名	引脚号	输入或输出	板上丝印符号
A	41	Input	S0
B	40	Input	S1
F	31	Output	LED0

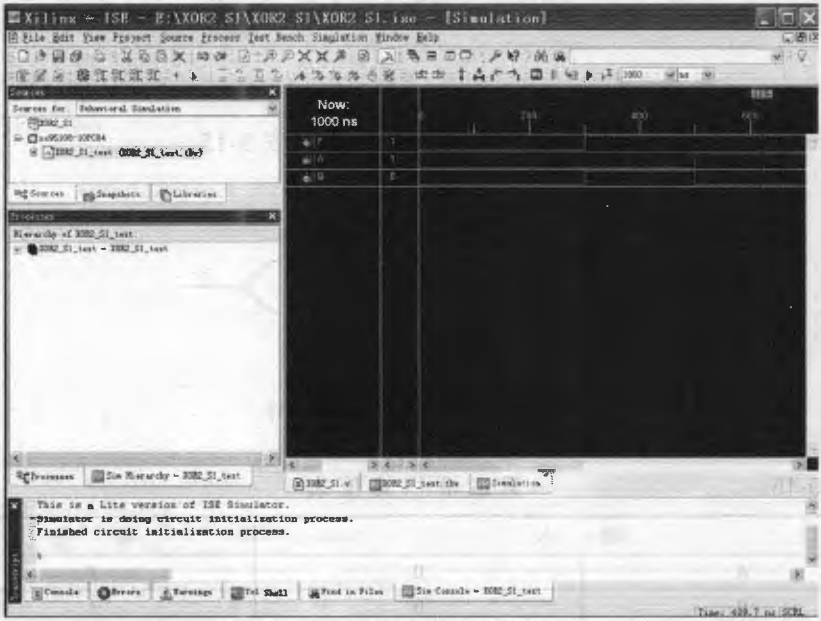


图 9-14　异或门仿真波形

在 MCU&CPLD DEMO 试验板上，我们将 4 位拨码开关的 S1S0 的开关同

时拨下（OFF）或同时拨上时（ON），XC95108的41、40脚同时输入高电平或低电平，这时LED0灭（XC95108的31脚输出高电平）；S1S0的一位拨上、一位拨下时，XC95108的41、40脚有一脚输入低电平、另一脚输入高电平，LED0亮（XC95108的31脚输出低电平）。实现了异或门的控制逻辑。

9.7.2 门级结构描述设计的异或门

除了采用数据流描述方法设计异或门外，我们也可以使用门级结构描述来进行设计。除源代码不同外，设计的其他过程完全相同。

源代码为：

```
module XOR2_S2(A,B,F);
output F;
input A,B;
xor U1(F,A,B);
endmodule
```

9.8 异或非门实践

9.8.1 数据流描述设计的异或非门

异或非门电路符号如图9-15所示。真值表见表9-15。

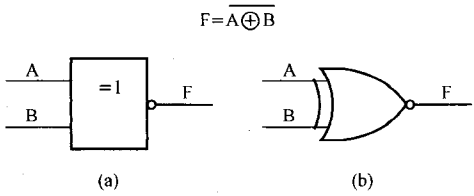


图9-15 异或非门电路符号

表9-15 异或非门真值表

输入		输出
A	B	F
0	0	1
0	1	0
1	0	0
1	1	1

在 E 盘中先建立一个文件名为“NXOR2_S1”的文件夹，然后建立一个“NXOR2_S1”的新项目，输入以下的源代码并保存为“NXOR2_S1.v”。

```
module NXOR2_S1(A,B,F);
output F;
input A,B;
assign F=~(A^B);
endmodule
```

异或非门引脚分配见表 9-16。项目编译通过后，可根据需要进行仿真，图 9-16 为仿真波形。最后将*.jed 文件下载到 XC95108 芯片中。

表 9-16　　异或非门引脚分配

引脚名	引脚号	输入或输出	板上丝印符号
A	41	Input	S0
B	40	Input	S1
F	31	Output	LED0

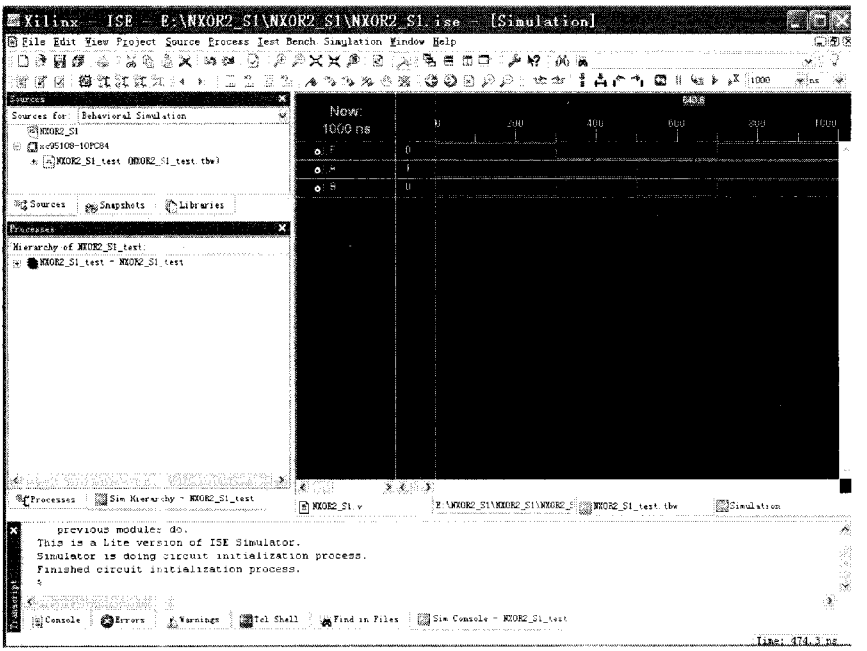


图 9-16　异或非门仿真波形

在 MCU&CPLD DEMO 试验板上，我们将 4 位拨码开关的 S1S0 的开关同

在 E 盘中先建立一个文件名为“NXOR2_S1”的文件夹，然后建立一个“NXOR2_S1”的新项目，输入以下的源代码并保存为“NXOR2_S1.v”。

```
module NXOR2_S1(A,B,F);
output F;
input A,B;
assign F=~(A^B);
endmodule
```

异或非门引脚分配见表 9-16。项目编译通过后，可根据需要进行仿真，图 9-16 为仿真波形。最后将*.jed 文件下载到 XC95108 芯片中。

表 9-16　　异或非门引脚分配

引脚名	引脚号	输入或输出	板上丝印符号
A	41	Input	S0
B	40	Input	S1
F	31	Output	LED0

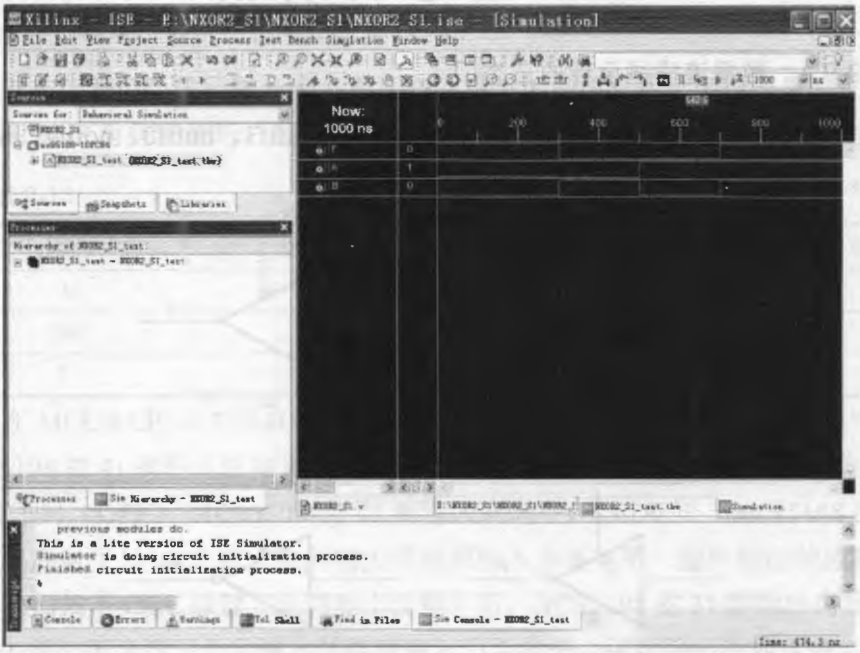


图 9-16　异或非门仿真波形

在 MCU&CPLD DEMO 试验板上，我们将 4 位拨码开关的 S1S0 的开关同

时拨下（OFF）或同时拨上时（ON），XC95108 的 41、40 脚同时输入高电平或低电平，这时 LED0 亮（XC95108 的 31 脚输出低电平）；S1S0 的一位拨上、一位拨下时，XC95108 的 41、40 脚有一脚输入低电平、另一脚输入高电平，LED0 灭（XC95108 的 31 脚输出高电平）。实现了异或非门的控制逻辑。

9.8.2 门级结构描述设计的异或非门

除了采用数据流描述方法设计异或非门外，我们也可以使用门级结构描述来进行设计。除源代码不同外，设计的其他过程完全相同。

源代码为：

```
module XNOR2_S2(A,B,F);
output F;
input A,B;
xnor U1(F,A,B);
endmodule
```

9.9 三 态 门 实 践

9.9.1 数据流描述设计的三态门

三态门的电路符号如图 9-17 所示，主要有 bufif1、bufif0、notif1、notif0 四种。

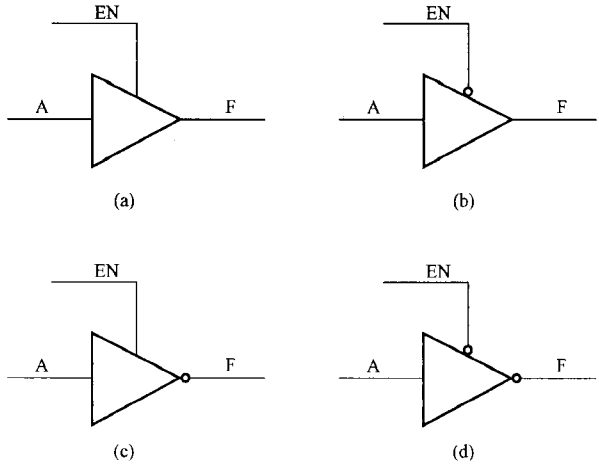


图 9-17 三态门电路符号

（a）bufif1；（b）bufif0；（c）notif1；（d）notif0

我们以 bufif1 为例进行实验，其真值表见表 9-17。

表 9-17　　三态门 bufif1 真值表

buif1		EN（使能端）			
		0	1	x	z
输入端	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	x	x	x

注　表中 L 代表 0 或 z，H 代表 1 或 z。

在 E 盘中先建立一个文件名为“BUFEN_S1”的文件夹，然后建立一个“BUFEN_S1”的新项目，输入以下的源代码并保存为“BUFEN_S1.v”。

```
module BUFEN_S1(A,EN,F);
output F;
input A,EN;
assign F=EN?A:1'bZ;
endmodule
```

三态门 bufif1 的引脚分配见表 9-18。项目编译通过后，可根据需要进行仿真，图 9-18 为仿真波形。最后将*.jed 文件下载到 XC95108 芯片中。

表 9-18　　三态门 bufif1 引脚分配

引脚名	引脚号	输入或输出	板上丝印符号
A	41	Input	S0
EN	76	Input	GOE
F	31	Output	LED0

在 MCU&CPLD DEMO 试验板上，将拨码开关的 S0 的开关拨上（ON），XC95108 的 41 脚输入低电平，这时 LED0 亮（XC95108 的 31 脚输出低电平）；S0 拨下时（OFF），XC95108 的 41 脚输入高电平，LED0 灭（XC95108 的 31 脚输出高电平）。以上由于全局输出使能端输入为高电平，因此输出使能有效。

我们按下 GOE 按键（全局输出使能）后，XC95108 的 31 脚输出高阻抗，LED0 灭。因为这时全局输出使能端输入为低电平，因此输出使能无效，输出高阻抗。

这样就实现了三态门 bufif1 的控制逻辑。

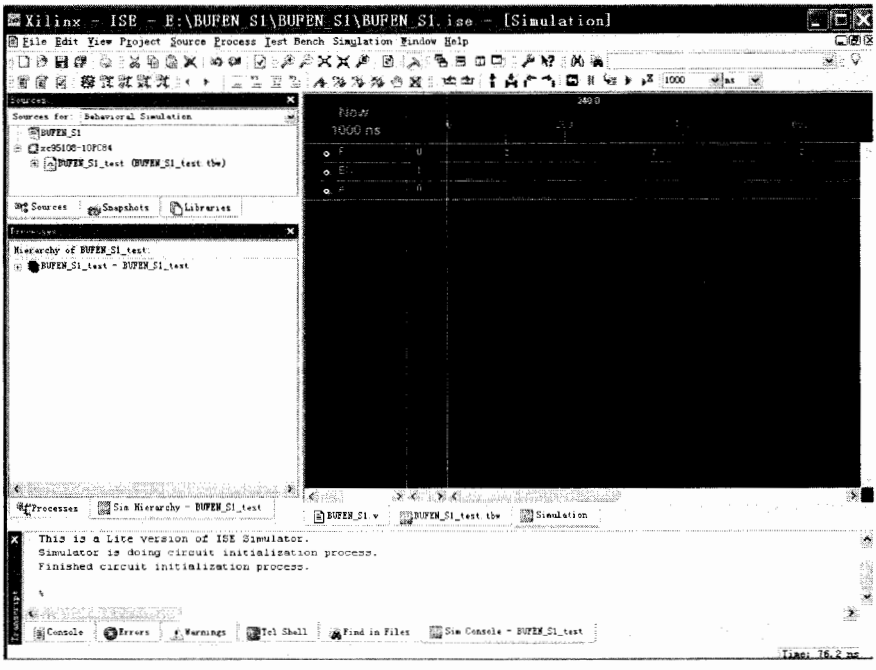


图 9-18　三态门 bufif1 仿真波形

9.9.2　门级结构描述设计的三态门

除了采用数据流描述方法设计三态门外，我们也可以使用门级结构描述来进行设计，还是以 bufif1 为例进行实验。除源代码不同外，设计的其他过程完全相同。

源代码为：

```
module BUFEN_S2(A,EN,F);
output F;
input A,EN;
bufif1 U1(F,A,EN);
endmodule
```

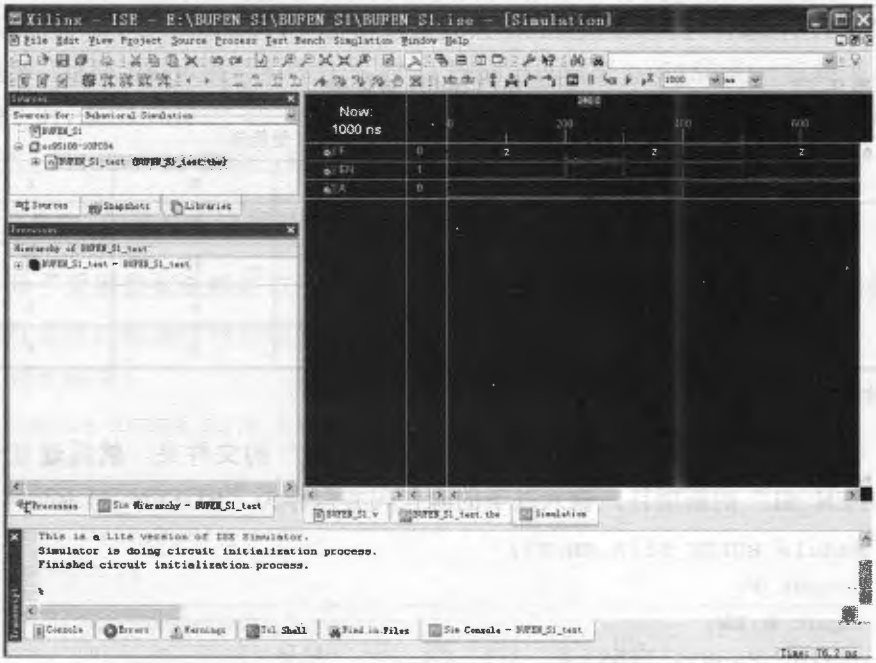


图 9-18　三态门 bufif1 仿真波形

9.9.2　门级结构描述设计的三态门

除了采用数据流描述方法设计三态门外，我们也可以使用门级结构描述来进行设计，还是以 bufif1 为例进行实验。除源代码不同外，设计的其他过程完全相同。

源代码为：

```
module BUFEN_S2(A,EN,F);
output F;
input A,EN;
bufif1 U1(F,A,EN);
endmodule
```

第 10 章

组合逻辑电路的设计实验

数字逻辑电路系统按功能的不同，可以分为组合逻辑电路和时序逻辑电路两大类。组合逻辑电路在任意时刻产生的输出只取决于该时刻的输入，而与电路过去的输入无关。常见的组合逻辑电路有数据选择器、编码器、译码器、加法器等。

10.1 2 选 1 数据选择器

10.1.1 2 选 1 数据选择器简介

数据选择器又称为多路开关，它的逻辑功能是在地址选择信号的控制下，从多路输入数据中选择某一路数据作为输出。2 选 1 数据选择器的逻辑功能是在地址选择信号的控制下，从 2 路输入数据中选择其中 1 路数据作为输出。

图 10-1 为 2 选 1 数据选择器的电路框图，输入端为 A、B，输出端为 F，SEL 为控制端。真值表见表 10-1。引脚分配见表 10-2。

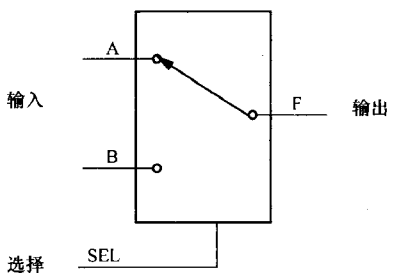


图 10-1 2 选 1 数据选择器电路框图

可见当控制信号 SEL 为 0 时，输出端 F 输出 A 信号；当 SEL 为 1 时，F 输出 B 信号。

表 10-1　　2 选 1 数据选择器真值表

输　入			输　出
SEL	B	A	F
0	X	X	A
1	X	X	B

表 10-2　　2 选 1 数据选择器引脚分配表

引脚名	引脚号	输入或输出	板上丝印符号
A	36	Input	K0
B	35	Input	K1
SEL	41	Input	S0
F	31	Output	LED0

10.1.2　数据流描述方式设计的源代码

```
module SEL2_1_S1 (A,B,SEL,F);          //模块声明及输入输出端口列表
input A,B,SEL;                         //定义输入端口
output F;                              //定义输出端口
assign F=~SEL&A|SEL&B;                 //数据流描述
endmodule                              //模块结束
```

10.1.3　行为描述方式设计的源代码

```
module SELE2_1_S2 (A,B,SEL,F);         //模块声明及输入输出端口列表
input A,B,SEL;                         //定义输入端口
output F;                              //定义输出端口
assign F=SELE2_1_FUN(A,B,SEL);         //调用 SELE2_1_FUN 函数

function SELE2_1_FUN;                  //定义函数
input A,B,SEL;                         //定义输入端口
if(SEL==0) SELE2_1_FUN=A;              //行为描述方式,当 SEL 为 0 时,输出 A
else SELE2_1_FUN=B;                    //否则当 SEL 为 1 时,输出 B
endfunction                            //函数模块结束

endmodule                              //模块结束
```

10.2　4 选 1 数据选择器

10.2.1　4 选 1 数据选择器简介

4 选 1 数据选择器的逻辑功能是在地址选择信号的控制下，从 4 路输入数据中选择其中 1 路数据作为输出。

图 10-2 为 4 选 1 数据选择器的电路框图，输入端为 A、B、C、D，输出端为 F，SEL1、SEL0 为控制端。真值表见表 10-3，可见当控制信号 SEL1 SEL0 为 00 时，输出端 F 输出 A 信号；当 SEL1 SEL0 为 01 时，F 输出 B 信号；当 SEL1 SEL0 为 10 时，F 输出 C 信号；当 SEL1 SEL0 为 11 时，F 输出 D 信号。引脚分配见表 10-4。

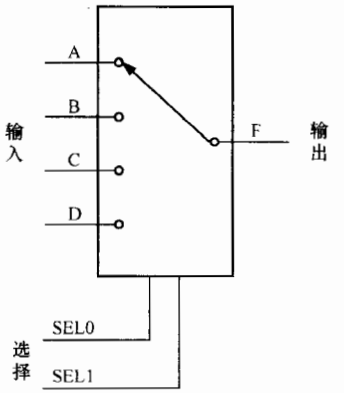


图 10-2　4 选 1 数据选择器电路框图

表 10-3　　4 选 1 数据选择器真值表

输　入						输　出
SEL1	SEL0	D	C	B	A	F
0	0	X	X	X	X	A
0	1	X	X	X	X	B
1	0	X	X	X	X	C
1	1	X	X	X	X	D

表 10-4　　4 选 1 数据选择器引脚分配表

引脚名	引脚号	输入或输出	板上丝印符号
A	36	Input	K0
B	35	Input	K1
C	34	Input	K2
D	33	Input	K3
SEL0	41	Input	S0
SELI	40	Input	S1
F	31	Output	LED0

10.2.2　数据流描述方式设计的源代码

```
module SEL4_1_S1 (A,B,C,D,SEL,F);  //模块声明及输入输出端口列表
input A,B,C,D;                     //定义输入端口
input [1:0]SEL;                    //定义输入端口
output F;                          //定义输出端口
assign                             //数据流描述
F=(~SEL[1]&~SEL[0]&A)|(~SEL[1]&SEL[0]&B)|(SEL[1]&~SEL[0]&C)|
(SEL[1]&SEL[0]&D);
endmodule                          //模块结束
```

10.2.3　行为描述方式设计的源代码

```
module SEL4_1_S2(A,B,C,D,SEL,F);   //模块声明及输入输出端口列表
input A,B,C,D;                     //定义输入端口
input [1:0] SEL;                   //定义输入端口
output F;                          //定义输出端口
assign F=SELE4_1_FUN(A,B,C,D,SEL); //数据流描述

function SELE4_1_FUN;              //定义函数
input A,B,C,D;                     //定义输入端口
input [1:0] SEL;                   //定义输入端口
case (SEL)                         //case 语句,根据 SEL 的值,产生散转分支
2'b00:SELE4_1_FUN=A;               //SEL 为 00 时,输出 A
2'b01:SELE4_1_FUN=B;               //SEL 为 01 时,输出 B
2'b10:SELE4_1_FUN=C;               //SEL 为 10 时,输出 C
2'b11:SELE4_1_FUN=D;               //SEL 为 11 时,输出 D
```

```
endcase                          //case 语句结束
endfunction                      //函数模块结束

endmodule                        //模块结束
```

10.3　2 位二进制编码器（4–2 编码器）

10.3.1　2 位二进制编码器简介

在数字逻辑电路系统里，把二进制码按一定的规律编排，使每组代码具有一特定的含义（代表某个数字或控制信号）称为编码，具有编码功能的电路称为编码器。编码器有若干个输入，但在某一时刻只有一个输入信号被转换成为二进制码。2 位二进制编码器也叫 4-2 编码器，它的逻辑功能是将 4 位输入转换为 2 位二进制码输出。

图 10-3 为 2 位二进制编码器的电路框图，输入端为 I0~I3，输出端为 F0、F1。真值表见表 10-5，引脚分配见表 10-6。

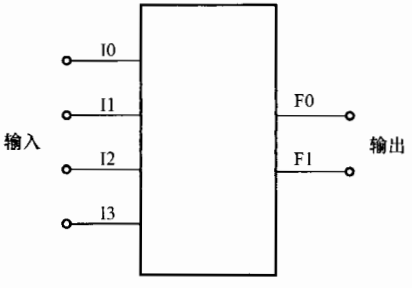


图 10-3　2 位二进制编码器电路框图

表 10-5　　2 位二进制编码器真值表

输　入				输　出	
I3	I2	I1	I0	F1	F0
0	X	X	X	1	1
1	0	X	X	1	0
1	1	0	X	0	1
1	1	1	0	0	0

表 10-6　　2 位二进制编码器引脚分配表

引脚名	引脚号	输入或输出	板上丝印符号
I0	36	Input	K0
I1	35	Input	K1
I2	34	Input	K2
I3	33	Input	K3
F0	31	Output	LED0
F1	26	Output	LED1

10.3.2　行为描述方式设计的源代码

```
module CODER4_2_S1 (I,F);          //模块声明及输入输出端口列表
input [3:0]I;                      //定义输入端口
output [1:0] F;                    //定义输出端口
assign F=code(I);                  //数据流描述

function [1:0] code;               //定义函数
input [3:0] I;                     //定义输入端口
case(I)
4'b0111: code=2'b11;               //如果 I[3]为低电平,输出 11
4'b1011: code=2'b10;               //如果 I[2]为低电平,输出 10
4'b1101: code=2'b01;               //如果 I[1]为低电平,输出 01
4'b1110: code=2'b00;               //如果 I[0]为低电平,输出 00
endcase                            //case 语句结束
endfunction                        //函数模块结束

endmodule                          //模块结束
```

10.4　3 位二进制优先编码器（8–3 优先编码器）

10.4.1　3 位二进制优先编码器简介

3 位二进制优先编码器也叫 8-3 优先编码器，它的逻辑功能是将 8 位输入转换为 3 位二进制码输出。

图 10-4 为 3 位二进制优先编码器的电路框图，I0～I7 是要进行优先编码的 8 个输入信号，F0~F3 是用来进行优先编码的 3 位二进制代码。I0～I7 八个输

入信号中，假定 I7 优先级别最高，I0 最低。根据优先级的排列，我们可列出 3 位二进制优先编码器的真值表，见表 10-7。引脚分配见表 10-8。

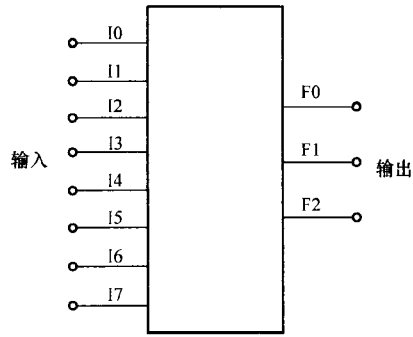


图 10-4　3 位二进制优先编码器电路框图

表 10-7　　3 位二进制优先编码器真值表

输　入								输　出		
I7	I6	I5	I4	I3	I2	I1	I0	F2	F1	F0
0	X	X	X	X	X	X	X	1	1	1
1	0	X	X	X	X	X	X	1	1	0
1	1	0	X	X	X	X	X	1	0	1
1	1	1	0	X	X	X	X	1	0	0
1	1	1	1	0	X	X	X	0	1	1
1	1	1	1	1	0	X	X	0	1	0
1	1	1	1	1	1	0	X	0	0	1
1	1	1	1	1	1	1	0	0	0	0

表 10-8　　3 位二进制优先编码器引脚分配表

引脚名	引脚号	输入或输出	板上丝印符号
I0	36	Input	K0
I1	35	Input	K1
I2	34	Input	K2
I3	33	Input	K3
I4	41	Input	S0

续表

引脚名	引脚号	输入或输出	板上丝印符号
I5	40	Input	S1
I6	39	Input	S2
I7	37	Input	S3
F0	31	Output	LED0
F1	26	Output	LED1
F2	25	Output	LED2

10.4.2 行为描述方式设计的源代码

```
module CODER8_3_S1 (I,F);          //模块声明及输入输出端口列表
input [7:0]I;                      //定义输入端口
output [2:0] F;                    //定义输出端口
assign F=code(I);                  //数据流描述

function [2:0] code;               //定义函数
input [7:0] I;                     //定义输入端口
if(!I[7]) code=3'b111;             //如果 I[7]为低电平,输出 111
else  if(!I[6]) code=3'b110;       //如果 I[6]为低电平,输出 110
else  if(!I[5]) code=3'b101;       //如果 I[5]为低电平,输出 101
else  if(!I[4]) code=3'b100;       //如果 I[4]为低电平,输出 100
else  if(!I[3]) code=3'b011;       //如果 I[3]为低电平,输出 011
else  if(!I[2]) code=3'b010;       //如果 I[2]为低电平,输出 010
else  if(!I[1]) code=3'b001;       //如果 I[1]为低电平,输出 001
else  if(!I[0]) code=3'b000;       //如果 I[0]为低电平,输出 000
endfunction                        //函数模块结束

endmodule                          //模块结束
```

10.5 3位二进制译码器（3-8线译码器）

10.5.1 3位二进制译码器简介

译码是编码的逆过程，它是将具有特定含义的二进制码进行辨别，并转换

成控制信号输出。具有译码功能的数字逻辑电路系统称为译码器。

3 位二进制译码器的逻辑功能是将 3 位二进制输入转换成 8 位的信号输出，8 位的信号输出在任意时刻只能有一位输出有效。

图 10-5 为 3 位二进制译码器的电路框图，输入端为 A0、A1、A2，可输入 3 位二进制码，共有 2^3=8 种组合状态；输出端有 F0~F7 共 8 条线。表 10-9 为真值表。引脚分配见表 10-10。

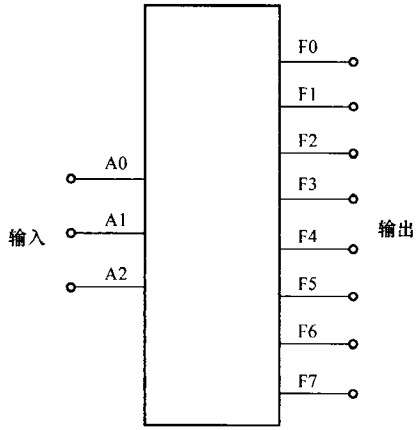


图 10-5　3 位二进制译码器电路框图

表 10-9　　3 位二进制译码器真值表

输　入			输　出							
A2	A1	A0	F7	F6	F5	F4	F3	F2	F1	F0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

表 10-10　　3 位二进制译码器引脚分配表

引脚名	引脚号	输入或输出	板上丝印符号
A0	41	Input	S0
A1	40	Input	S1

续表

引脚名	引脚号	输入或输出	板上丝印符号
A2	39	Input	S2
F0	31	Output	LED0
F1	26	Output	LED1
F2	25	Output	LED2
F3	24	Output	LED3
F4	23	Output	LED4
F5	21	Output	LED5
F6	20	Output	LED6
F7	19	Output	LED7

10.5.2 行为描述方式设计的源代码

```
module DECOD3_8_S1 (F,A);  //模块声明及输入输出端口列表
input [2:0]A;              //定义输入端口
output [7:0] F;            //定义输出端口
reg [7:0] F;               //定义 F 为寄存器类型的 8 位变量
always @(A)       //每当输入 A 发生变化时,执行一遍 begin_end 块内的语句
begin                      //begin_end 块开始
case (A)                   //case 语句,根据 A 的值,产生散转分支
3'b000:F=8'b00000001;      //A 为 000 时,F 输出 00000001
3'b001:F=8'b00000010;      //A 为 001 时,F 输出 00000010
3'b010:F=8'b00000100;      //A 为 010 时,F 输出 00000100
3'b011:F=8'b00001000;      //A 为 011 时,F 输出 00001000

3'b100:F=8'b00010000;      //A 为 100 时,F 输出 00010000
3'b101:F=8'b00100000;      //A 为 101 时,F 输出 00100000
3'b110:F=8'b01000000;      //A 为 110 时,F 输出 01000000
3'b111:F=8'b10000000;      //A 为 111 时,F 输出 10000000
endcase                    // case 语句结束
end                        // begin_end 块结束
endmodule                  //模块结束
```

10.6　4 位二进制译码器（4–16 线译码器）

10.6.1　4 位二进制译码器简介

4 位二进制译码器的逻辑功能是将 4 位二进制输入转换成 16 位的信号输出，16 位的信号输出在任意时刻只能有一位输出有效。

图 10-6 为 4 位二进制译码器的电路框图，输入端为 A0、A1、A2、A3，可输入 4 位二进制码，共有 2^4=16 种组合状态；输出端有 F0~F15 共 16 条线。表 10-11 为 4 位二进制译码器的真值表。由于我们的 MCU&CPLD DEMO 试验板上只有 8 个发光二极管作指示，不能满足 16 个发光管的指示要求，因此这里就不进行实验了，但应当掌握设计方法。4 位二进制译码器真值表见表 10-11。

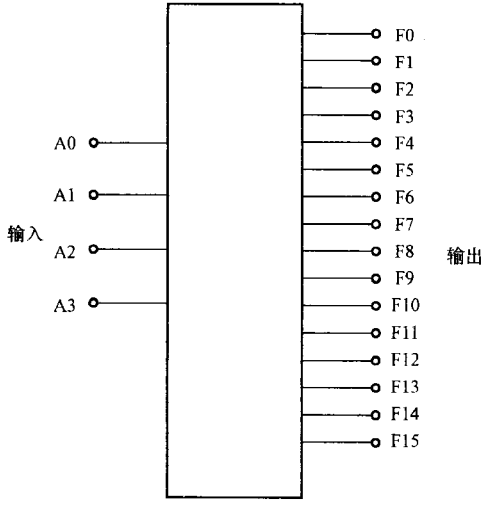


图 10-6　4 位二进制译码器电路框图

表 10-11　　4 位二进制译码器真值表

输　入				输　出															
A3	A2	A1	A0	F15	F14	F13	F12	F11	F10	F9	F8	F7	F6	F5	F4	F3	F2	F1	F0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0

续表

输入				输出															
A3	A2	A1	A0	F15	F14	F13	F12	F11	F10	F9	F8	F7	F6	F5	F4	F3	F2	F1	F0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

10.6.2 行为描述方式设计的源代码

```
module DECOD4_16_S1 (F,A); //模块声明及输入输出端口列表
input [3:0]A;              //定义输入端口
output [15:0] F;           //定义输出端口
reg [15:0] F;              //定义 F 为寄存器类型的 16 位变量
always @(A)       //每当输入 A 发生变化时,执行一遍 begin_end 块内的语句
begin                      //begin_end 块开始
case (A)                   //case 语句,根据 A 的值,产生散转分支
4'b0000:F=16'b0000000000000001;   //A 为 0000 时,F 输出 0000000000
                                  //000001
4'b0001:F=16'b0000000000000010;   //A 为 0001 时,F 输出 0000000000
                                  //000010
4'b0010:F=16'b0000000000000100;   //A 为 0010 时,F 输出 0000000000
                                  //000100
4'b0011:F=16'b0000000000001000;   //A 为 0011 时,F 输出 0000000000
                                  //001000

4'b0100:F=16'b0000000000010000;   //A 为 0100 时,F 输出 0000000000
                                  //010000
4'b0101:F=16'b0000000000100000;   //A 为 0101 时,F 输出 0000000000
                                  //100000
```

```
4'b0110:F=16'b0000000001000000;   //A为0110时,F输出0000000001
                                  //000000
4'b0111:F=16'b0000000010000000;   //A为0111时,F输出0000000010
                                  //000000

4'b1000:F=16'b0000000100000000;   //A为1000时,F输出0000000100
                                  //000000
4'b1001:F=16'b0000001000000000;   //A为1001时,F输出0000001000
                                  //000000
4'b1010:F=16'b0000010000000000;   //A为1010时,F输出0000010000
                                  //000000
4'b1011:F=16'b0000100000000000;   //A为1011时,F输出0000100000
                                  //000000

4'b1100:F=16'b0001000000000000;   //A为1100时,F输出0001000000
                                  //000000
4'b1101:F=16'b0010000000000000;   //A为1101时,F输出0010000000
                                  //000000
4'b1110:F=16'b0100000000000000;   //A为1110时,F输出0100000000
                                  //000000
4'b1111:F=16'b1000000000000000;   //A为1111时,F输出1000000000
                                  //000000
endcase                  //case语句结束
end                      //begin_end块结束
endmodule                //模块结束
```

10.7　4–10 线译码器（BCD 译码器）

10.7.1　4-10 线译码器简介

4-10 线译码器的逻辑功能是将 4 位二进制输入转换成 10 位的信号输出，10 位的信号输出在任意时刻只能有一位输出有效。

图 10-7 为 4-10 线译码器的电路框图，输入端为 A0、A1、A2、A3，可输入 4 位二进制码，共有 2^4=16 种组合状态；输出端有 F0~F9 共 10 条线。表 10-11 为真值表。由于我们的 MCU&CPLD DEMO 试验板上只有 8 个发光二极管作指示，不能满足 10 个发光管的指示要求，因此这里也不进行实验了，但应当掌握

设计方法。4-10线译码器真值表见表10-12。

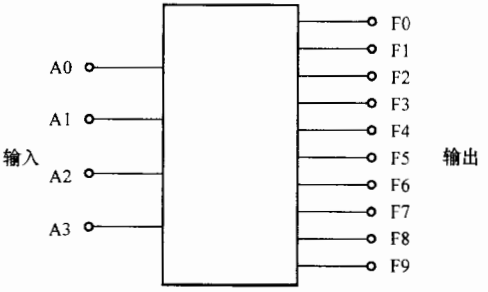


图10-7　4-10线译码器电路框图

表10-12　　4-10线译码器真值表

输　入				输　出									
A3	A2	A1	A0	F9	F8	F7	F6	F5	F4	F3	F2	F1	F0
0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	0	0	0	1	0	0
0	0	1	1	0	0	0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	0	0	1	0	0	0	0
0	1	0	1	0	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	0	1	0	0	0	0	0	0
0	1	1	1	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	1	1	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	X	X	X	X	X	X	X	X	X	X
1	1	0	0	X	X	X	X	X	X	X	X	X	X
1	1	0	1	X	X	X	X	X	X	X	X	X	X
1	1	1	0	X	X	X	X	X	X	X	X	X	X
1	1	1	1	X	X	X	X	X	X	X	X	X	X

注　X为任意态。

10.7.2　行为描述方式设计的源代码

```
module DECOD4_10_S1 (F,A);    //模块声明及输入输出端口列表
```

```
input [3:0]A;                    //定义输入端口
output [9:0] F;                  //定义输出端口
reg [9:0] F;                     //定义 F 为寄存器类型的 10 位变量
always @(A)                      //每当输入 A 发生变化时,执行
                                 //一遍 begin_end 块内的语句
begin                            //begin_end 块开始
case (A)                         //case 语句,根据 A 的值,产生散转分支
4'b0000:F=10'b0000000001;        //A 为 0000 时,F 输出 0000000001
4'b0001:F=10'b0000000010;        //A 为 0001 时,F 输出 0000000010
4'b0010:F=10'b0000000100;        //A 为 0010 时,F 输出 0000000100
4'b0011:F=10'b0000001000;        //A 为 0011 时,F 输出 0000001000

4'b0100:F=10'b0000010000;        //A 为 0100 时,F 输出 0000010000
4'b0101:F=10'b0000100000;        //A 为 0101 时,F 输出 0000100000
4'b0110:F=10'b0001000000;        //A 为 0110 时,F 输出 0001000000
4'b0111:F=10'b0010000000;        //A 为 0111 时,F 输出 0010000000

4'b1000:F=10'b0000000001;        //A 为 1000 时,F 输出 0100000000
4'b1001:F=10'b0000000010;        //A 为 1001 时,F 输出 1000000000
endcase                          //case 语句结束
end                              //begin_end 块结束
endmodule                        //模块结束
```

10.8　BCD–7 段译码器实践

10.8.1　BCD-7 段译码器简介

BCD-7 段译码器用于将 BCD 码译成 LED 数码管可直接显示的数字，根据数码管的不同（有共阴或共阳极的数码管），共有两种不同的 BCD-7 段译码器。分析 MCU&CPLD DEMO 试验板电路原理图，使用的是共阴极数码管，因此我们必须设计成驱动共阴极数码管的 BCD-7 段译码器。表 10-13 为共阴极数码管的段位与显示字型码及 BCD 码关系。BCD-7 段译码器真值表见表 10-14，引脚分配见表 10-15。

表 10-13　　共阴极数码管的段位与显示字型码及 BCD 码关系

输入					输出								
Data				显示	dp	g	f	e	d	c	b	a	十六进制
0	0	0	0	0	0	0	1	1	1	1	1	1	3f
0	0	0	1	1	0	0	0	0	0	1	1	0	06
0	0	1	0	2	0	1	0	1	1	0	1	1	5b
0	0	1	1	3	0	1	0	0	1	1	1	1	4f
0	1	0	0	4	0	1	1	0	0	1	1	0	66
0	1	0	1	5	0	1	1	0	1	1	0	1	6d
0	1	1	0	6	0	1	1	1	1	1	0	1	7d
0	1	1	1	7	0	0	0	0	0	1	1	1	07
1	0	0	0	8	0	1	1	1	1	1	1	1	7f
1	0	0	1	9	0	1	1	0	1	1	1	1	6f
1	0	1	0	熄灭	高阻	高阻	高阻	高阻	高阻	高阻	高阻	高阻	高阻
1	0	1	1	熄灭	高阻	高阻	高阻	高阻	高阻	高阻	高阻	高阻	高阻
1	1	0	0	熄灭	高阻	高阻	高阻	高阻	高阻	高阻	高阻	高阻	高阻
1	1	0	1	熄灭	高阻	高阻	高阻	高阻	高阻	高阻	高阻	高阻	高阻
1	1	1	0	熄灭	高阻	高阻	高阻	高阻	高阻	高阻	高阻	高阻	高阻
1	1	1	1	熄灭	高阻	高阻	高阻	高阻	高阻	高阻	高阻	高阻	高阻

表 10-14　　BCD-7 段译码器真值表

输入				输出							
Data				dp	g	f	e	d	c	b	a
0	0	0	0	0	0	1	1	1	1	1	1
0	0	0	1	0	0	0	0	0	1	1	0
0	0	1	0	0	1	0	1	1	0	1	1
0	0	1	1	0	1	0	0	1	1	1	1
0	1	0	0	0	1	1	0	0	1	1	0
0	1	0	1	0	1	1	0	1	1	0	1
0	1	1	0	0	1	1	1	1	1	0	1
0	1	1	1	0	0	0	0	0	1	1	1
1	0	0	0	0	1	1	1	1	1	1	1
1	0	0	1	0	1	1	0	1	1	1	1

表 10-15　　BCD-7 段译码器引脚分配表

引脚名	引脚号	输入或输出	板上丝印符号
Data[0]	41	Input	S0
Data[1]	40	Input	S1
Data[2]	39	Input	S2
Data[3]	37	Input	S3
F0（a）	72	Output	由个位数码管进行显示
F1（b）	71	Output	
F2（c）	70	Output	
F3（d）	69	Output	
F4（e）	68	Output	
F5（f）	67	Output	
F6（g）	66	Output	
F7（dp）	65	Output	
SelOutCom0	54	Output	

10.8.2　行为描述方式设计的源代码

```
module DEC_BCD_S1(F,Data,SelOutCom0); //模块声明及输入输出端口列表
input [3:0]Data;                      //定义输入端口
output [7:0] F;                       //定义输出端口
output SelOutCom0;                    //定义输出端口
reg [7:0] F;          //定义 F 为寄存器类型的 8 位变量
reg SelOutCom0;       //定义 SelOutCom0 为寄存器类型变量
always @(Data)//每当输入 Data 发生变化时,执行一遍 begin_end 块内的语句
begin                 //begin_end 块开始
case (Data)           //case 语句,根据 Data 的值,产生散转分支
4'd0:F=8'h3f;         //Data 为 0 时,F 输出 3fH
4'd1:F=8'h06;         //Data 为 1 时,F 输出 06H
4'd2:F=8'h5b;         //Data 为 2 时,F 输出 5bH
4'd3:F=8'h4f;         //Data 为 3 时,F 输出 4fH

4'd4:F=8'h66;         //Data 为 4 时,F 输出 66H
4'd5:F=8'h6d;         //Data 为 5 时,F 输出 6dH
4'd6:F=8'h7d;         //Data 为 6 时,F 输出 7dH
```

```
4'd7:F=8'h07;          //Data 为 7 时,F 输出 07H
4'd8:F=8'h7f;          //Data 为 8 时,F 输出 7fH
4'd9:F=8'h6f;          //Data 为 9 时,F 输出 6fH
default:F=8'hz;        //Data 为大于 9 以上时,F 输出高阻抗
endcase                // case 语句结束
end                    // begin_end 块结束

always                 //无条件执行一遍 begin_end 块内的语句
begin                  //begin_end 块开始
SelOutCom0=1;          //SelOutCom0 输出 1
end                    //begin_end 块结束

endmodule              //模块结束
```

10.9 半加器实践

10.9.1 半加器简介

加法器主要有半加法器和全加法器两种（简称半加器和全加器）。它们都用来实现二进制数中的加法运算。

两个 1 位的二进制数相加，叫做半加。实现两个 1 位二进制数相加运算的电路叫做半加器电路，半加器可完成两个 1 位二进制数的求和运算。根据半加器电路的定义，半加器是实现加数、被加数、和数、向高位进位数组成的运算电路，它仅考虑本位数相加，而不考虑低位来的进位数。半加器具有加数端 A、被加数端 B、和数端 SUM、进位输出端 COUT。

图 10-8 为半加器的电路框图。表 10-16 为真值表，引脚分配见表 10-17。

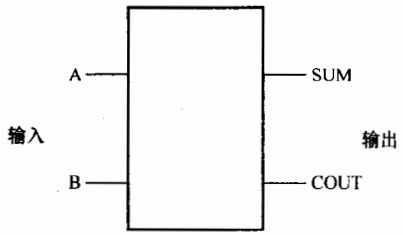


图 10-8　半加器电路框图

表 10-16　　半 加 器 真 值 表

输入端		输出端	
加数端 A	被加数端 B	和数端 SUM	进位输出端 COUT
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

表 10-17　　半加器引脚分配表

引脚名	引脚号	输入或输出	板上丝印符号
A	41	Input	S0
B	40	Input	S1
SUM	31	Output	LED0
COUT	19	Output	LED7

10.9.2　门级结构描述的源代码

```
module ADD_H_S1(A,B,SUM,COUT);          //模块声明及输入输出端口列表
input A,B;                              //定义输入端口
output SUM,COUT;                        //定义输出端口
and (COUT,A,B);                         //行为描述方式
xor (SUM,A,B);                          //行为描述方式
endmodule                               //模块结束
```

10.9.3　数据流描述方式的源代码

```
module ADD_H_S2(A,B,SUM,COUT);          //模块声明及输入输出端口列表
input A,B;                              //定义输入端口
output SUM,COUT;                        //定义输出端口
assign SUM=A^B;                         //数据流描述方式
assign COUT=A&B;                        //数据流描述方式
endmodule                               //模块结束
```

10.9.4　行为描述方式的源代码

```
module ADD_H_S3(A,B,SUM,COUT); //模块声明及输入输出端口列表
```

```
input A,B;                          //定义输入端口
output SUM,COUT;                    //定义输出端口
reg SUM,COUT;                       //定义 SUM、COUT 为寄存器类型的变量
always @(A or B)//每当输入 A 或 B 发生变化时,执行一遍 begin_end 块内的语句
begin                               //begin_end 块开始
case ({A,B})                        //case 语句,根据 AB 的值,产生散转分支
2'b00:begin SUM=0;COUT=0;end        //AB 为 00 时,SUM 输出 0,COUT 输出 0
2'b01:begin SUM=1;COUT=0;end        //AB 为 01 时,SUM 输出 1,COUT 输出 0
2'b10:begin SUM=1;COUT=0;end        //AB 为 10 时,SUM 输出 1,COUT 输出 0
2'b11:begin SUM=0;COUT=1;end        //AB 为 11 时,SUM 输出 0,COUT 输出 1
endcase                             //case 语句结束
end                                 //begin_end 块结束
endmodule                           //模块结束
```

10.10 全加器实践

10.10.1 1位全加器简介

半加器只有两个输入端，不能处理由低位送来的进位数，全加器则能够实现二进制全加运算。全加器在对两个二进制数进行加法运算时，除了能将加数A、被加数B相加外，还要加上低位送来的进位数CIN。所以，全加器比半加器电路多一个输入端，共有三个输入端。全加器与半加器相比只是多了一个低位进位数端CIN。这里的实验主要针对1位全加器为例的。

图10-9为1位全加器的电路框图，表10-18为真值表。引脚分配见表10-19。1位全加器的结构原理如图10-10所示。

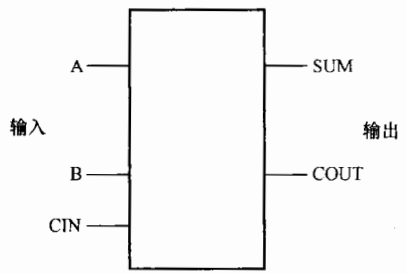


图10-9 1位全加器电路框图

表 10-18　　全 加 器 真 值 表

输入端			输出端	
加数端 A	被加数端 B	进位输入 CIN	和数 SUM	进位输出 COUT
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

表 10-19　　全加器引脚分配表

引脚名	引脚号	输入或输出	板上丝印符号
A	41	Input	S0
B	40	Input	S1
CIN	39	Input	S2
SUM	31	Output	LED0
COUT	19	Output	LED7

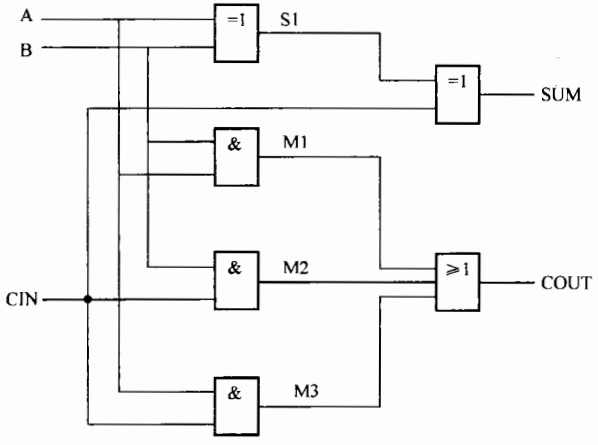


图 10-10　1 位全加器的结构原理

10.10.2　门级结构描述的源代码

```
module ADD1_FU_S1(A,B,CIN,SUM,COUT); //模块声明及输入输出端口列表
```

```
input A,B,CIN;                                    //定义输入端口
output SUM,COUT;                                  //定义输出端口
wire S1,M1,M2,M3;
and (M1,A,B),(M2,B,CIN),(M3,A,CIN);
xor (S1,A,B),(SUM,S1,CIN);
or (COUT,M1,M2,M3);
endmodule                                         //模块结束
```

10.10.3 数据流描述方式的源代码

```
module ADD1_FU_S2(A,B,CIN,SUM,COUT); //模块声明及输入输出端口列表
input A,B,CIN;                                    //定义输入端口
output SUM,COUT;                                  //定义输出端口
assign {COUT,SUM}=A+B+CIN;                        //数据流描述
endmodule                                         //模块结束
```

10.10.4 行为描述方式的源代码

```
module ADD1_FU_S3(A,B,CIN,SUM,COUT); //模块声明及输入输出端口列表
input A,B,CIN;                                    //定义输入端口
output SUM,COUT;                                  //定义输出端口
reg SUM,COUT;
reg M1,M2,M3;
always@(A or B or CIN)
  begin
  M1=A&B;
  M2=B&CIN;
  M3=A&CIN;
  SUM=(A^B)^CIN;
  COUT=(M1|M2)|M3;
  end
endmodule                                         //模块结束
```

第 11 章

触发器的实践

组合逻辑电路，其任意时刻产生的输出仅与当时的输入有关，它没有记忆功能。而触发器是一种具有记忆功能的电路，在任意时刻产生的输出不仅与当时的输入有关，而且还与过去的输入有关。

11.1 RS 触 发 器

11.1.1 RS 触发器简介

图 11-1 为 RS 触发器电路框图，输入端为 R、S、CLK，输出端为 Q、QB，其中时钟 CLK 为输入门控信号，只有 CLK 信号到来时，输入信号 R、S 才能进入触发器。依据 CLK 信号的触发方式不同，RS 触发器可分为上升沿触发和下降沿触发两种。图 11-1 为上升沿触发的 RS 触发器。真值表见表 11-1。引脚分配见表 11-2。

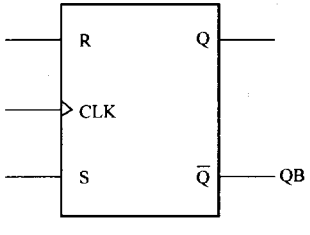


图 11-1 RS 触发器电路框图

表 11-1　　RS 触发器真值表

R	S	Qn+1	说明
0	0	Qn	保持
0	1	1	置 1
1	0	0	置 0
1	1	不定	不使用

表 11-2　　RS 触发器引脚分配表

引脚名	引脚号	输入或输出	板上丝印符号
R	41	Input	S0
S	40	Input	S1
CLK	10	Input	GCLK2
Q	31	Output	LED0
QB	26	Output	LED1

11.1.2　RS 触发器的设计源代码

```
module SYRS_FF(Q,QB,R,S,CLK); //模块声明及输入输出端口列表
output Q,QB;                  //定义输出端口
input R,S,CLK;                //定义输入端口
reg Q;                        //定义 Q 为寄存器类型的变量
assign QB=~Q;                 //数据流描述
always @(posedge CLK)         //每当 CLK 产生上升沿时
case ({R,S})                  //case 语句,根据 RS 的值,产生散转分支
2'b01:Q<=1'b1;                //RS 为 01 时,Q 输出 1
2'b10:Q<=1'b0;                //RS 为 10 时,Q 输出 0
2'b11:Q<=1'bx;                //RS 为 11 时,Q 输出无关值
endcase                       //case 语句结束
endmodule                     //模块结束
```

11.2　JK 触发器

11.2.1　JK 触发器简介

图 11-2 为 JK 触发器电路框图，输入端为 J、K、CLK，输出端为 Q、QB。

其中时钟 CLK 为输入门控信号，只有 CLK 信号到来时，输入信号 J、K 才能进入触发器。依 CLK 信号的触发方式不同，JK 触发器可分为上升沿触发和下降沿触发两种。图 11-2 为上升沿触发的 JK 触发器。表 11-3 为 JK 触发器真值表。引脚分配见表 11-4。

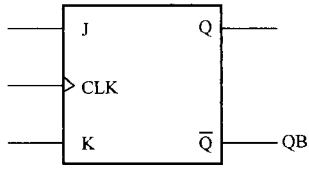


图 11-2　JK 触发器电路框图

表 11-3　　JK 触发器真值表

J	K	Qn	Qn+1	说　明
0	0	0	0	保持
0	0	1	1	
0	1	0	0	同步置 0
0	1	1	0	
1	0	0	1	同步置 1
1	0	1	1	
1	1	0	1	翻转
1	1	1	0	

表 11-4　　JK 触发器引脚分配表

引脚名	引脚号	输入或输出	板上丝印符号
J	41	Input	S0
K	40	Input	S1
CLK	10	Input	GCLK2
Q	31	Output	LED0
QB	26	Output	LED1

11.2.2　JK 触发器的设计源代码

```
module SYJK_FF(Q,QB,J,K,CLK); //模块声明及输入输出端口列表
output Q,QB;                  //定义输出端口
```

```
input J,K,CLK;                    //定义输入端口
reg Q;                            //定义 Q 为寄存器类型的变量
assign QB=~Q;                     //数据流描述

always @(posedge CLK)             //每当 CLK 产生上升沿时,执行一遍
                                    begin_end 块内的语句

begin                             //begin_end 块开始
case ({J,K})                      //case 语句,根据 JK 的值,产生散转分支
2'b00:Q<=Q;                       //JK 为 00 时,Q 无变化
2'b01:Q<=1'b0;                    //JK 为 01 时,Q 输出 0
2'b10:Q<=1'b1;                    //JK 为 10 时,Q 输出 1
2'b11:Q<=~Q;                      //JK 为 11 时,Q 的输出反相
default:Q<=1'bx;                  //默认状态,Q 输出无关值
endcase                           //case 语句结束
end                               //begin_end 块结束

endmodule                         //模块结束
```

11.3 带有复位的 D 触发器

11.3.1 带有复位的 D 触发器简介

图 11-3 为带有复位的 D 触发器电路框图。其中时钟 CLK 为输入门控信号，只有 CLK 信号到来时，输入信号 D 才能进入触发器。依 CLK 信号的触发方式不同，D 触发器可分为上升沿触发和下降沿触发两种。图 11-3 为上升沿触发的 D 触发器。表 11-5 为 D 触发器真值表。引脚分配见表 11-6。

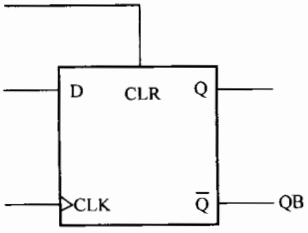


图 11-3 带有复位的 D 触发器电路框图

表 11-5　　带有复位的 D 触发器真值表

CLR	D	Qn	Qn+1	说　明
0	0	0	0	输出状态与 D 端状态相同
0	0	1	0	
0	1	0	1	
0	1	1	1	
1	X	X	0	清 0

表 11-6　　带有复位的 D 触发器引脚分配表

引脚名	引脚号	输入或输出	板上丝印符号
CLR	74	Input	GCLR
D	41	Input	S0
CLK	10	Input	GCLK2
Q	31	Output	LED0
QB	26	Output	LED1

11.3.2　带有复位的 D 触发器设计源代码

```
module CLR_SYD_FF(Q,QB,D,CLK,CLR); //模块声明及输入输出端口列表
output Q,QB;                       //定义输出端口
input D,CLK,CLR;                   //定义输入端口
reg Q,QB;                          //定义 Q、QB 为寄存器类型的变量
//每当 CLK 产生上升沿或 CLR 产生上升沿时,执行一遍 begin_end 块内的语句
always @(posedge CLK or posedge CLR)

begin                              //begin_end 块开始

if(CLR)                            //如果 CLR 为低电平时
begin
Q<=0;                              //Q 输出 0(非阻塞赋值)
QB<=1;                             //QB 输出 1(非阻塞赋值)
end

else                               //否则如果 CLR 为高电平时
begin
Q<=D;                              //Q 输出 D(非阻塞赋值)
QB<=~D;                            //QB 输出 D 的反相信号(非阻塞赋值)
```

```
end

end                                    //begin_end 块结束

endmodule                              //模块结束
```

11.4 带有复位的异步 T 触发器

11.4.1 带有复位的异步 T 触发器简介

所谓 T 触发器就是翻转触发器或计数触发器，当每来一个时钟脉冲（或计数脉冲），触发器就翻转一次。图 11-4 为带有复位的异步 T 触发器电路框图。表 11-7 为带有复位的异步 T 触发器真值表。引脚分配见表 11-8。

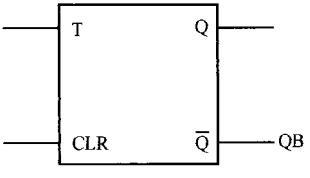


图 11-4 带有复位的异步 T 触发器电路框图

表 11-7 带有复位的异步 T 触发器真值表

R	T	Qn	Qn+1	说 明
0	0	0	0	保持
0	0	1	1	
0	1	0	1	翻转
0	1	1	0	
1	X	X	0	清 0

表 11-8 带有复位的异步 T 触发器引脚分配表

引脚名	引脚号	输入或输出	板上丝印符号
CLR	74	Input	GCLR
T	36	Input	K0
Q	31	Output	LED0
QB	26	Output	LED1

11.4.2　带有复位的异步 T 触发器的设计源代码

```
module CLR_T_FF(Q,QB,T,CLR); //模块声明及输入输出端口列表
output Q,QB;                 //定义输出端口
input T,CLR;                 //定义输入端口
reg Q;                       //定义 Q 为寄存器类型的变量
assign QB=~Q;                //数据流描述
//每当 T 产生上升沿或 CLR 产生上升沿时,执行一遍 begin_end 块内的语句
always @(posedge T or posedge CLR)
begin                        //begin_end 块开始
if(CLR)Q<=0;                 //如果 CLR 为高电平时,Q 输出 0
else if(T)Q<=~Q;             //否则如果 T 为高电平时,Q 的输出反转
end                          //begin_end 块结束

endmodule                    //模块结束
```

11.5　带有复位的同步 T 触发器

11.5.1　带有复位的同步 T 触发器简介

图 11-5 为带有复位的同步 T 触发器电路框图，与带有复位的异步 T 触发器相比，增加了一个时钟端 CLK。时钟 CLK 为输入门控信号，只有 CLK 信号到来时，输入信号 T 才能进入触发器。依 CLK 信号的触发方式不同，带有复位的同步 T 触发器可分为上升沿触发和下降沿触发两种。图 11-5 为上升沿触发的带有复位的同步 T 触发器。引脚分配见表 11-9。

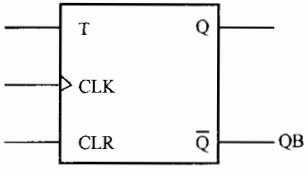


图 11-5　带有复位的同步 T 触发器电路框图

表 11-9　　带有复位的同步 T 触发器引脚分配表

引脚名	引脚号	输入或输出	板上丝印符号
CLK	10	Input	GCLK2

续表

引脚名	引脚号	输入或输出	板上丝印符号
CLR	74	Input	GCLR
T	41	Input	S0
Q	31	Output	LED0
QB	26	Output	LED1

11.5.2 带有复位的同步T触发器设计源代码

```
module CLR_SYT_FF(Q,QB,T,CLK,CLR); //模块声明及输入输出端口列表
output Q,QB;                  //定义输出端口
input T,CLK,CLR;              //定义输入端口
reg Q;                        //定义Q为寄存器类型的变量
assign QB=~Q;                 //数据流描述
//每当CLK产生上升沿或CLR产生上升沿时,执行一遍begin_end块内的语句
always @(posedge CLK or posedge CLR)
begin                         //begin_end块开始
if(CLR)Q<=0;                  //如果CLR为高电平时,Q输出0
else if(T)Q<=~Q;              //否则如果T为高电平时,Q的输出反转
end                           //begin_end块结束

endmodule                     //模块结束
```

第 12 章

时序逻辑电路的设计实验

时序逻辑电路的输出是与时序（时钟）是有关联的，在时钟的作用下将输入信号传输到输出端。

12.1 寄 存 器

12.1.1 寄存器简介

具有将二进制数据寄存起来功能的数字电路称为寄存器。寄存器主要是由具有记忆功能的触发器组合起来构成的。

图 12-1 为 4 位寄存器电路框图，4 位数据输入端为 D0~D3；CLR 为清零端，低电平有效；CLK 为时钟端，上升沿触发；输出端为 Q0~Q3。图 12-2 为由 D 触发器构成的 4 位寄存器内部逻辑电路。4 位寄存器真值表如表 12-1 所示。引脚分配见表 12-2。

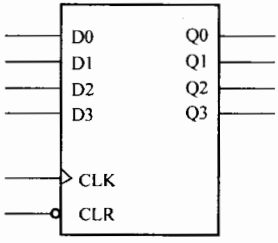


图 12-1 4 位寄存器电路框图

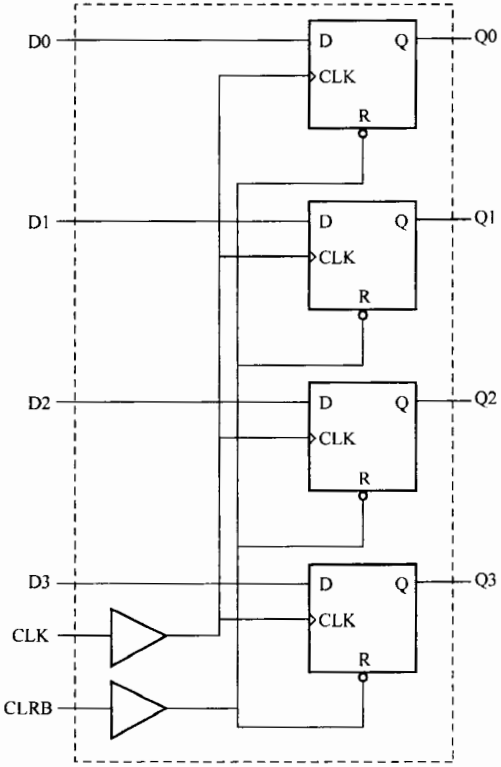


图 12-2　由 D 触发器构成的 4 位寄存器内部逻辑电路

表 12-1　　4 位寄存器真值表

输　入						输　出				说　明
CLRB	CLK	D3	D2	D1	D0	Q3	Q2	Q1	Q0	
0	X	X	X	X	X	0	0	0	0	清零
1	↑	D3	D2	D1	D0	D3	D2	D1	D0	送数
1	1	X	X	X	X	保持				
1	0	X	X	X	X					

表 12-2　　4 位寄存器引脚分配表

引脚名	引脚号	输入或输出	板上丝印符号
CLRB	74	Input	GCLR
CLK	10	Input	GCLK2
D3	37	Input	S3
D2	39	Input	S2

续表

引脚名	引脚号	输入或输出	板上丝印符号
D1	40	Input	S1
D0	41	Input	S0
Q3	24	Output	LED3
Q2	25	Output	LED2
Q1	26	Output	LED1
Q0	31	Output	LED0

12.1.2　寄存器的设计源代码

```
module REG4(CLRB,CLK,D,Q); //模块声明及输入输出端口列表
input CLRB,CLK;            //定义输入端口
input [3:0] D;             //定义输入端口
output [3:0] Q;            //定义输出端口
reg [3:0] Q;               //定义 Q 为寄存器类型的 4 位变量
//每当 CLK 产生上升沿或 CLRB 产生下降沿时,执行一遍 begin_end 块内的语句
always @(posedge CLK or negedge CLRB)
  begin                    //begin_end 块开始
  if(!CLRB)Q<=0;           //如果 CLRB 为低电平,Q 输出 0(非阻塞赋值)
  else Q<=D;               //Q 输出 D 的值(非阻塞赋值)
  end                      //begin_end 块结束
endmodule                  //模块结束
```

12.2　锁　存　器

12.2.1　锁存器简介

锁存器和寄存器都具有数据暂存功能，但两者也有区别：锁存器一般是由电平信号控制的，属于电平敏感型；而寄存器一般由同步时钟信号控制。因此，当数据信号提前于控制信号并要求同步控制时，可使用寄存器；当数据信号滞后于控制信号时，只能使用锁存器了。

4 位锁存器与 4 位寄存器的区别是输入的 4 位数据信号滞后于控制信号 CLK，输出由电平信号 CLK 控制，高电平有效。4 位锁存器真值表见表 12-3。

引脚分配见表 12-4。

表 12-3　　4 位锁存器真值表

输　入						输　出				说　明
CLRB	CLK	D3	D2	D1	D0	Q3	Q2	Q1	Q0	
0	X	X	X	X	X	0	0	0	0	清零
1	1	D3	D2	D1	D0	D3	D2	D1	D0	送数
1	0	X	X	X	X	保持				

表 12-4　　4 位锁存器引脚分配表

引脚名	引脚号	输入或输出	板上丝印符号
CLRB	74	Input	GCLR
CLK	10	Input	GCLK2
D3	37	Input	S3
D2	39	Input	S2
D1	40	Input	S1
D0	41	Input	S0
Q3	24	Output	LED3
Q2	25	Output	LED2
Q1	26	Output	LED1
Q0	31	Output	LED0

12.2.2　锁存器的设计源代码

```
module LATCH4(CLRB,CLK,D,Q); //模块声明及输入输出端口列表
input CLRB,CLK;              //定义输入端口
input [3:0] D;               //定义输入端口
output [3:0] Q;              //定义输出端口
reg [3:0] Q;                 //定义 Q 为寄存器类型的 4 位变量
//每当输入 CLRB 或 CLK 或 D 发生变化时,执行一遍 begin_end 块内的语句
always @(CLRB or CLK or D)
begin                        //begin_end 块开始
   if(!CLRB) Q=0;            //如果 CLRB 为低电平,Q 输出 0(阻塞赋值)
   else if(CLK) Q=D;    //否则 CLK 为高电平时,Q 输出 D 的值(阻塞赋值)
   end                       //begin_end 块结束
endmodule                    //模块结束
```

12.3　移 位 寄 存 器

12.3.1　移位寄存器简介

移位寄存器是一种在时钟脉冲的作用下，将暂存在寄存器内的数据按位左移或右移的数字电路。数据可以采用并行输入、并行输出方式，也可以采用串行输入、串行输出方式，还可以并行输入、串行输出或串行输入、并行输出。因此移位寄存器的使用非常灵活，用途十分广泛。

图 12-3 为 4 位串行输入、并行输出移位寄存器逻辑电路。DATA 为数据输入端；D0~D3 为数据输出端；CLK 为时钟信号，上升沿触发；CLRB 为清零信号，下降沿触发。表 12-5 为移位寄存器真值表。引脚分配见表 12-6。

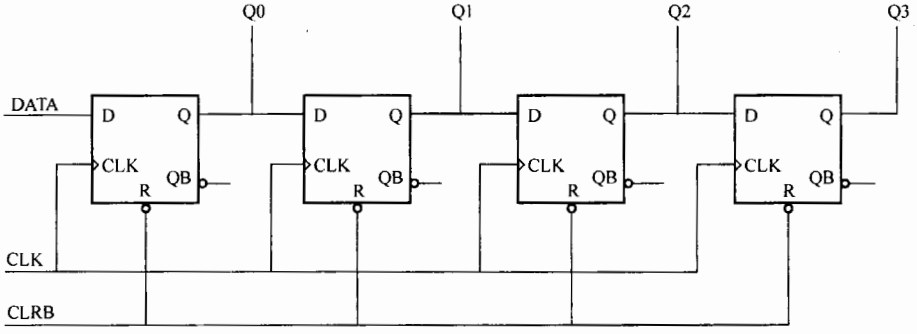


图 12-3　4 位串行输入、并行输出移位寄存器逻辑电路

表 12-5　　移位寄存器真值表

输　入			输　出				说明
CLRB	CLK 脉冲次数↑	DATA	Q3	Q2	Q1	Q0	
1	0	1	0	0	0	0	移位
1	1	1	0	0	0	1	
1	2	1	0	0	1	1	
1	3	1	0	1	1	1	
1	4	1	1	1	1	1	
↓	X	X	0	0	0	0	清零

表 12-6　　4 位串行输入、并行输出移位寄存器引脚分配表

引脚名	引脚号	输入或输出	板上丝印符号
CLRB	74	Input	GCLR
CLK	10	Input	GCLK2
DATA	41	Input	S0
Q3	24	Output	LED3
Q2	25	Output	LED2
Q1	26	Output	LED1
Q0	31	Output	LED0

12.3.2　4 位移位寄存器的设计源代码

```
module SHIFT4(CLRB,CLK,DATA,Q); //模块声明及输入输出端口列表
input CLRB,CLK,DATA;            //定义输入端口
output [3:0] Q;                 //定义输出端口
reg [3:0] Q;                    //定义 Q 为寄存器类型的 4 位变量
//每当 CLK 产生上升沿或 CLRB 产生下降沿时,执行一遍 begin_end 块内的语句
always @(posedge CLK or negedge CLRB)
  begin                         //begin_end 块开始
  if(!CLRB)Q<=0;     //如果 CLRB 为低电平,Q 输出 0(非阻塞赋值)
  else                          //否则
     begin
     Q<=Q<<1;                   //Q 左移一位后输出(非阻塞赋值)
     Q[0]<=DATA;                //Q[0]输出 DATA 的电平
     end
  end                           //begin_end 块结束
endmodule                       //模块结束
```

12.4　计　数　器

12.4.1　二进制异步加法计数器简介

计数器是一种能够将输入的时钟脉冲记忆下来的数字电路。在数字系统中，计数器是一种使用很广泛的器件，它不仅能够记忆输入的时钟脉冲，还可实现分频、定时、产生同步脉冲及脉冲分配等。计数器的分类有好多种，以下是常

见的几种分类方法。

（1）按计数的进制分，可分为二进制计数器、十进制计数器、任意进制计数器。

（2）按计数的加或减，可分为加法计数器、减法计数器、可逆（可加也可减）计数器。

（3）按计数时触发器的翻转是否同步，可分为同步计数器、异步计数器。

图 12-4 为 4 位二进制异步加法计数器逻辑电路，时钟输入端为 CLK，上升沿触发；CLRB 为清零端，下降沿触发；输出端为 Q0~Q3。表 12-7 为 4 位二进制异步加法计数器真值表。引脚分配见表 12-8。

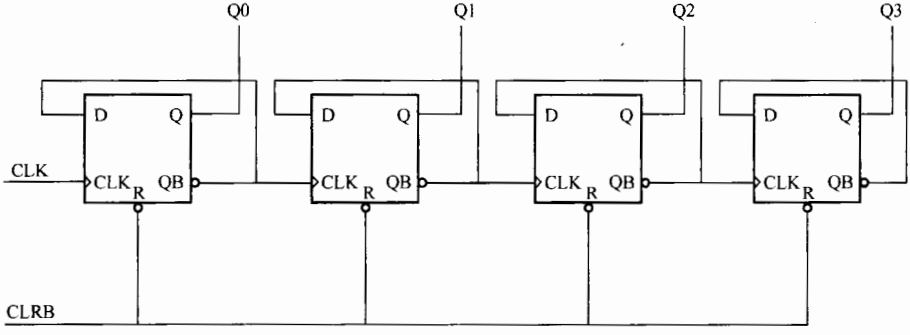


图 12-4　4 位二进制异步加法计数器逻辑电路

表 12-7　　4 位二进制异步加法计数器真值表

输　入		输　出				说　明
CLRB	CLK 脉冲次数↑	Q3	Q2	Q1	Q0	
1	0	0	0	0	0	
1	1	0	0	0	1	
1	2	0	0	1	0	
1	3	0	0	1	1	加
1	4	0	1	0	0	法
1	5	0	1	0	1	计
1	6	0	1	1	0	数
1	7	0	1	1	1	
1	8	1	0	0	0	
1	9	1	0	0	1	

续表

输　入		输　出				说　明
CLRB	CLK 脉冲次数↑	Q3	Q2	Q1	Q0	
1	10	1	0	1	0	
1	11	1	0	1	1	加
1	12	1	1	0	0	法
1	13	1	1	0	1	计
1	14	1	1	1	0	数
1	15	1	1	1	1	
1	16	0	0	0	0	
↓	X	0	0	0	0	清零

表 12-8　　4 位二进制异步加法计数器引脚分配表

引脚名	引脚号	输入或输出	板上丝印符号
CLRB	74	Input	GCLR
CLK	10	Input	GCLK2
Q3	24	Output	LED3
Q2	25	Output	LED2
Q1	26	Output	LED1
Q0	31	Output	LED0

12.4.2　4 位二进制异步加法计数器的设计源代码

```
/**********************CNT4.v**********************/
module CNT4(Q,CLK,CLRB);    //模块声明及输入输出端口列表
input CLK,CLRB;             //定义输入端口
output [3:0]Q;              //定义输出端口
wire [3:0] QB;              //定义 QB 为网线型的 4 位变量
//下面为门级结构描述设计
CLR_SYD_FF CLR_SYD_FF0(Q[0],QB[0],QB[0],CLK,CLRB);
CLR_SYD_FF CLR_SYD_FF1(Q[1],QB[1],QB[1],QB[0],CLRB);
CLR_SYD_FF CLR_SYD_FF2(Q[2],QB[2],QB[2],QB[1],CLRB);
CLR_SYD_FF CLR_SYD_FF3(Q[3],QB[3],QB[3],QB[2],CLRB);
endmodule                   //模块结束
/******************CLR_SYD_FF******************/
module CLR_SYD_FF(Q,QB,D,CLK,CLR);    //模块声明及输入输出端口列表
output Q,QB;                //定义输出端口
input D,CLK,CLR;            //定义输入端口
```

```
reg Q,QB;                          //定义 Q、QB 为寄存器类型的变量
//每当 CLK 产生上升沿或 CLR 产生下降沿时,执行一遍 begin_end 块内的语句
always @(posedge CLK or negedge CLR)

begin                              //begin_end 块开始

if(!CLR)                           //如果 CLR 为低电平

begin
Q<=0;                              //Q 输出 0(非阻塞赋值)
QB<=1;                             //QB 输出 1(非阻塞赋值)
end

else                               //否则如果 CLR 为高电平
begin
Q<=D;                              //Q 输出 D(非阻塞赋值)
QB<=~D;                            //QB 输出 D 的反相值(非阻塞赋值)
end

end                                //begin_end 块结束

endmodule                          //模块结束
```

12.4.3　十进制（任意进制）同步加法计数器简介

表 12-9 为十进制同步加法计数器真值表，根据该表，我们可以取某个中间值（Q3~Q0=1010）来控制输出端清零，例如：在前 9 个脉冲时，计数器作加法输出；当第 10 个脉冲到来时控制计数器清零。引脚分配见表 12-10。

表 12-9　　十进制同步加法计数器真值表

输　入		输　出				说　明
CLRB	CLK 脉冲次数↑	Q3	Q2	Q1	Q0	
1	0	0	0	0	0	加法计数
1	1	0	0	0	1	
1	2	0	0	1	0	
1	3	0	0	1	1	
1	4	0	1	0	0	
1	5	0	1	0	1	

续表

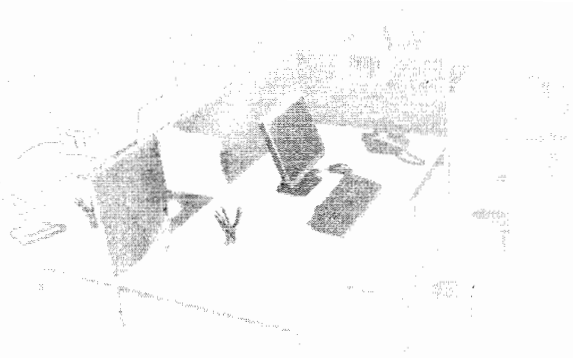
输入		输出				说明
CLRB	CLK 脉冲次数↑	Q3	Q2	Q1	Q0	
1	6	0	1	1	0	加法计数
1	7	0	1	1	1	
1	8	1	0	0	0	
1	9	1	0	0	1	
↓	10	0	0	0	0	清零

表 12-10　　十进制同步加法计数器引脚分配表

引脚名	引脚号	输入或输出	板上丝印符号
CLRB	74	Input	GCLR
CLK	10	Input	GCLK2
Q3	24	Output	LED3
Q2	25	Output	LED2
Q1	26	Output	LED1
Q0	31	Output	LED0

12.4.4　十进制同步加法计数器的设计源代码

```
module SYCNT10(Q,CLK,CLRB); //模块声明及输入输出端口列表
input CLK,CLRB;             //定义输入端口
output [3:0]Q;              //定义输出端口
reg [3:0]Q;                 //定义 Q 为寄存器类型的 4 位变量
//每当 CLK 产生上升沿或 CLRB 产生下降沿时,执行一遍 begin_end 块内的语句
always @(posedge CLK or negedge CLRB)
begin                       //begin_end 块开始
if(!CLRB) Q<=0;             //如果 CLRB 为低电平,Q 输出 0(非阻塞赋值)
else if(Q==9) Q<=0;         //否则如果 Q 为 9 时,Q 输出 0(非阻塞赋值)
else Q<=Q+1;                //否则 Q 作加法计数(非阻塞赋值)
end                         //begin_end 块结束
endmodule                   //模块结束
```



第 13 章

用 XC95108 芯片进行多种设计实验

13.1 发光管 LED0～LED7 闪烁实验

13.1.1 实验要求

我们的眼睛能够清楚地观察到，MCU&CPLD DEMO 试验板上的发光管 LED0~LED7 闪烁。

我们可以设定一个时间（例如 0.7s），让 LED 每隔一定的时间点亮。MCU&CPLD DEMO 试验板上的有源晶振频率为 24MHz，要得到 0.7s 的时间，需要进行 2^{24} 分频，时间 $t=2^{24}/24000000=0.7s$。

13.1.2 程序设计

在 E 盘中先建立一个文件名为“LED_GLITTER”的文件夹，然后建立一个“LED_GLITTER”的新项目，输入以下的源代码并保存为“LED_GLITTER.v”。

```
module LED_GLITTER (LED,CLK); //模块声明及输入输出端口列表
output[7:0] LED;              //定义输出端口
input CLK;                    //定义输入端口
reg[7:0] LED;                 //定义 LED 为寄存器类型的 8 位变量
reg[23:0] BUFFER;             //定义 BUFFER 为寄存器类型的 24 位变量
//-------------------------------------------------------------
```

```
//每当 CLK 产生上升沿时,执行一遍 begin_end 块内的语句
always@(posedge CLK)
 begin                                  //begin_end 块开始
   BUFFER=BUFFER+1;                     //计数缓存器 BUFFER 加 1
     if(BUFFER==24'b111111111111111111111111) //如果 0.7s 到了
         begin
         LED=~LED;                      //LED 输出信号翻转
         end
  end                                   //begin_end 块结束
endmodule                               //模块结束
```

引脚分配关系见表 13-1。器件编译通过后，可根据需要进行仿真。最后将*.jed 文件下载到 XC95108 芯片中。

表 13-1　　LED0~LED7 闪烁实验的引脚分配

引脚名	引脚号	输入或输出	板上丝印符号
CLK	9	Input	—
LED0	31	Output	LED0
LED1	26	Output	LED1
LED2	25	Output	LED2
LED3	24	Output	LED3
LED4	23	Output	LED4
LED5	21	Output	LED5
LED6	20	Output	LED6
LED7	19	Output	LED7

在 MCU&CPLD DEMO 试验板上，我们看到 LED0~LED7 这 8 个发光二极管闪烁起来。由于 MCU&CPLD DEMO 试验板上安装了一个 24MHz 的有源晶振，经过 2^{24} 的分频后，发光二极管的闪烁频率约 1.4Hz。

13.2　发光管 LED0～LED7 的跑马灯实验

13.2.1　实验要求

我们的眼睛能够清楚地观察到，MCU&CPLD DEMO 试验板上的 8 个发光管 LED0~LED7 以跑马灯的方式运行。

同样我们可以设定一个时间（例如 0.35s），让 LED 每隔一定的时间点亮。MCU&CPLD DEMO 试验板上的有源晶振频率为 24MHz，要得到 0.35s 的时间，需要进行 2^{23} 分频，时间 t=2^{23}/24000000=0.35s。为了驱动 8 个 LED，我们还需要建立一个状态变量 status，让 status 每 0.35s 做 1 次加法，status 的范围可以控制在 0~7 之间，这样就可根据 status 的值来扫描点亮 LED 了，看上去就像马在跑一样，故取名为“跑马灯”。

13.2.2　程序设计

在 E 盘中先建立一个文件名为“LED_HORSE”的文件夹，然后建立一个“LED_HORSE”的新项目，输入以下的源代码并保存为“LED_HORSE.v”。

```
module LED_HORSE(LED,CLK);      //模块声明及输入输出端口列表
output[7:0] LED;                //定义输出端口
input CLK;                      //定义输入端口
reg[7:0] LED;                   //定义 LED 为寄存器类型的 8 位变量
reg[22:0] BUFFER;               //定义 BUFFER 为寄存器类型的 23 位变量
reg[2:0] STATUS;                //定义 STATUS 为寄存器类型的 3 位变量
//-----------------------------------------------
//每当 CLK 产生上升沿时,执行一遍 begin_end 块内的语句
always@(posedge CLK)
 begin                          //begin_end 块开始
   BUFFER<=BUFFER+1'b1;         //计数缓存器 BUFFER 加 1
     if(BUFFER==23'b11111111111111111111111) //如果 0.35s 到了
         begin
         STATUS <= STATUS +1'b1;   //状态变量 STATUS 加 1
           if(STATUS ==3'd7) STATUS <=0;  //状态变量 STATUS 在 0~7
                                          之间循环
        end
  end                           // begin_end 块结束
//-----------------------------------------------
//每当 CLK 产生上升沿时,执行一遍 begin_end 块内的语句
always@(posedge CLK)
 begin                          //begin_end 块开始
   case(STATUS)                 //case 语句,根据 STATUS 的值,产生散转分支
   4'd0:LED<=8'b11111110; //STATUS 为 0000 时,LED 输出 11111110
   4'd1:LED<=8'b11111101; //STATUS 为 0001 时,LED 输出 11111101
   4'd2:LED<=8'b11111011; //STATUS 为 0010 时,LED 输出 11111011
```

```
    4'd3:LED<=8'b11110111; //STATUS 为 0011 时,LED 输出 11110111
    4'd4:LED<=8'b11101111; //STATUS 为 0100 时,LED 输出 11101111
    4'd5:LED<=8'b11011111; //STATUS 为 0101 时,LED 输出 11011111
    4'd6:LED<=8'b10111111; //STATUS 为 0110 时,LED 输出 10111111
    4'd7:LED<=8'b01111111; //STATUS 为 0111 时,LED 输出 01111111
    endcase                //case 语句结束
  end                      //begin_end 块结束
//-------------------------------------------
endmodule                  //模块结束
```

引脚分配见表 13-2。器件编译通过后，可根据需要进行仿真。最后将*.jed 文件下载到 XC95108 芯片中。

表 13-2　　跑马灯实验的引脚分配

引脚名	引脚号	输入或输出	板上丝印符号
CLK	9	Input	—
LED0	31	Output	LED0
LED1	26	Output	LED1
LED2	25	Output	LED2
LED3	24	Output	LED3
LED4	23	Output	LED4
LED5	21	Output	LED5
LED6	20	Output	LED6
LED7	19	Output	LED7

通电以后，我们看到，在 MCU&CPLD DEMO 试验板上，LED0~LED7 这 8 个 LED 中，始终有一个点亮的 LED 以一定的速度在跑动，并循环不已。

13.3　数码管动态扫描显示实验

13.3.1　实验要求

我们的眼睛能够清晰地看到，MCU&CPLD DEMO 试验板上的 8 个数码管稳定地显示（不抖动）“76543210”。

我们还是要设定一个较短的时间（例如 0.7ms），每隔一定的时间点亮一位数码管，循环扫描点亮。这样由于扫描的时间很快，8 位数码管的扫描周期还

不到 6ms，远低于人眼视觉暂留特性的限定值，故可以看到稳定的显示。当然为了驱动 8 个数码管，我们也要建立一个状态变量 status，让 status 每 0.7ms 做 1 次加法，status 的范围可以控制在 0~7 之间，这样就可根据 status 的值来扫描点亮数码管了。

13.3.2　程序设计

在 E 盘中先建立一个文件名为“SEG7”的文件夹，然后建立一个“SEG7”的新项目，输入以下的源代码并保存为“SEG7.v”。

```
module SEG7(SEG,SEL,CLK); //模块声明及输入输出端口列表
output[7:0] SEG;          //定义输出端口
output[7:0] SEL;          //定义输出端口
input CLK;                //定义输入端口
reg[7:0] SEG;             //定义 SEG 为寄存器类型的 8 位变量
reg[7:0] SEL;             //定义 SEL 为寄存器类型的 8 位变量
reg[13:0] COUNTER;        //定义 COUNTER 为寄存器类型的 14 位变量
reg[2:0] STATUS;          //定义 STATUS 为寄存器类型的 3 位变量
//------------------------------------------
//每当 CLK 产生上升沿时,执行一遍 begin_end 块内的语句
always@(posedge CLK)
 begin                    //begin_end 块开始
   COUNTER<=COUNTER+1'b1; //计数器 COUNTER 加 1
     if(COUNTER==14'b11111111111111) //如果 0.7ms 到了
         begin
         STATUS<=STATUS+1'b1;        //状态 STATUS 加 1
           if(STATUS==3'd7)STATUS<=0; //状态 STATUS 在 0~7 之间循环
         end
  end                                 //begin_end 块结束
//------------------------------------------
//每当 CLK 产生上升沿时,执行一遍 begin_end 块内的语句
always@(posedge CLK)
 begin                    //begin_end 块开始
   case(STATUS)           //case 语句,根据 STATUS 的值,产生散转分支
   3'd0:SEG<=8'h3f;       //送出“0”的字段码
   3'd1:SEG<=8'h06;       //送出“1”的字段码
   3'd2:SEG<=8'h5b;       //送出“2”的字段码
   3'd3:SEG<=8'h4f;       //送出“3”的字段码
```

```
    3'd4:SEG<=8'h66;          //送出“4”的字段码
    3'd5:SEG<=8'h6d;          //送出“5”的字段码
    3'd6:SEG<=8'h7d;          //送出“6”的字段码
    3'd7:SEG<=8'h07;          //送出“7”的字段码
    endcase                   //case 语句结束
  end                         //begin_end 块结束
//-------------------------------------------
//每当 CLK 产生上升沿时,执行一遍 begin_end 块内的语句
always@(posedge CLK)
  begin                       //begin_end 块开始
    case(STATUS)              //case 语句,根据 STATUS 的值,产生散转分支
    3'd0:SEL<=8'b00000001; //点亮个位数码管
    3'd1:SEL<=8'b00000010; //点亮十位数码管
    3'd2:SEL<=8'b00000100; //点亮百位数码管
    3'd3:SEL<=8'b00001000; //点亮千位数码管
    3'd4:SEL<=8'b00010000; //点亮万位数码管
    3'd5:SEL<=8'b00100000; //点亮十万位数码管
    3'd6:SEL<=8'b01000000; //点亮百万位数码管
    3'd7:SEL<=8'b10000000; //点亮千万位数码管
    endcase                   //case 语句结束
  end                         //begin_end 块结束

endmodule                     //模块结束
```

引脚分配见表 13-3。器件编译通过后，可根据需要进行仿真。最后将*.jed文件下载到 XC95108 芯片中。

表 13-3　　数码管动态扫描显示实验的引脚分配

引脚名	引脚号	输入或输出	板上丝印符号
CLK	9	Input	—
SEG7	65	Output	—
SEG6	66	Output	—
SEG5	77	Output	—
SEG4	68	Output	—
SEG3	69	Output	—
SEG2	70	Output	—
SEG1	71	Output	—
SEG0	72	Output	—

续表

引脚名	引脚号	输入或输出	板上丝印符号
SEL7	63	Output	—
SEL 6	62	Output	—
SEL 5	61	Output	—
SEL 4	58	Output	—
SEL 3	57	Output	—
SEL 2	56	Output	—
SEL 1	55	Output	—
SEL 0	54	Output	—

在 MCU&CPLD DEMO 试验板上，我们看到 8 个数码管稳定地显示“7654-3210”。

13.4　4 人投票表决器实验

13.4.1　实验要求

按下 MCU&CPLD DEMO 试验板上的 K0~K3 中的一个或多个按键，电路能实现 4 人投票表决器的作用，如果有 3 个或以上的人按下键，LED7 点亮，表示表决通过；如果少于 3 个人，LED7 不亮，表明表决未获通过。

这个实验我们要用到 for 语句来进行循环判断，因为是 4 人投票表决器，所以要判断 4 次。然后再根据判断的结果来控制 LED7 的亮灭。

13.4.2　程序设计

在 E 盘中先建立一个文件名为“VOTER4”的文件夹，然后建立一个“VOTER4”的新项目，输入以下的源代码并保存为“VOTER4.v”。

```
module VOTER4(PASS,VOTER);          //模块声明及输入输出端口列表
output PASS;                        //定义输出端口
input[3:0] VOTER;                   //定义输入端口
reg[2:0] SUM;                       //定义 SUM 为寄存器类型的 2 位变量
integer i;                          //定义 i 为整型变量
reg PASS;                           //定义 PASS 为寄存器类型的变量
```

```
    always @(VOTER)        //每当 VOTER 发生变化时,执行一遍 begin_end
                             块内的语句
      begin      //begin_end 块开始
        SUM=0;  //SUM 变量先清零
        for(i=0;i<=3;i=i+1)                    //for 循环
            if(!VOTER[i]) SUM=SUM+1;  //若有键按下,SUM 变量递加
            if((SUM==3)||(SUM==4))PASS=0; //若超过 2 人赞成,则 PASS=0
            else   PASS=1;                     //否则 PASS=1
        end                                    //begin_end 块结束
endmodule                                      //模块结束
```

引脚分配见表 13-4。器件编译通过后，可根据需要进行仿真。最后将*.jed文件下载到 XC95108 芯片中。

表 13-4　　4 人投票表决器实验的引脚分配

引脚名	引脚号	输入或输出	板上丝印符号
VOTER 0	36	Input	K0
VOTER 1	35	Input	K1
VOTER 2	34	Input	K2
VOTER 3	33	Input	K3
PASS	19	Output	LED7

在 MCU&CPLD DEMO 试验板上，我们分别按下 K0~K3 中的一个或多个按键，即可实现 4 人投票表决器的实验。

13.5　蜂鸣器发声实验

13.5.1　实验要求

我们要使 MCU&CPLD DEMO 试验板上的蜂鸣器发出一定频率的声音。

MCU&CPLD DEMO 试验板上使用的是交流蜂鸣器，只要加入一定频率的脉冲信号即可发出音频声响。我们可以设定一个 500μs 时间(通过计数器得到)，每隔 500μs 后取反输出端，经三极管 Q8 电流放大后即以 1kHz 的脉冲驱动蜂鸣器，那么蜂鸣器就会发出 1kHz 的声响。

13.5.2　程序设计

在 E 盘中先建立一个文件名为“BEEP”的文件夹，然后建立一个“BEEP”的新项目，输入以下的源代码并保存为“BEEP.v”。

```
module BEEP(BZ_OUT,CLK);   //模块声明及输入输出端口列表
output BZ_OUT;             //定义输出端口
input CLK;                 //定义输入端口
reg[13:0] COUNTER;         //定义 COUNTER 为寄存器类型的 14 位变量
reg BZ_OUT;                //定义 BZ_OUT 为寄存器类型的 1 位变量
//-------------------------------------
//每当 CLK 产生上升沿时,执行一遍 begin_end 块内的语句
always@(posedge CLK)
 begin                              //begin_end 块开始
   COUNTER<=COUNTER+1'b1;           //计数器 COUNTER 加 1
     if(COUNTER==14'd12000)         //如果 500μs 到了
         begin
         COUNTER<=14'd0;            //计数器清零
         BZ_OUT<=~BZ_OUT;           //输出端取反,驱动蜂鸣器
         end
  end                               //begin_end 块结束

endmodule                          //模块结束
```

引脚分配见表 13-5。器件编译通过后，可根据需要进行仿真。最后将*.jed 文件下载到 XC95108 芯片中。

表 13-5　　蜂鸣器发声实验引脚分配

引脚名	引脚号	输入或输出	板上丝印符号
CLK	9	Input	—
BZ_OUT	19	Output	LED7

在 MCU&CPLD DEMO 试验板上，将一个短路块插到 BEEP 排针上（连通蜂鸣器的驱动电路），我们立刻能听到清脆的音频声。

13.6　报 警 声 实 验

13.6.1　实验要求

上面的实验蜂鸣器会发出连续的声响。下面我们进行蜂鸣器发出间断的

“滴、滴…”声响实验。

13.6.2 程序设计

在 E 盘中先建立一个文件名为“ALARM”的文件夹，然后建立一个“ALARM”的新项目，输入以下的源代码并保存为“ALARM.v”。

```
module ALARM (BZ_OUT,CLK);//模块声明及输入输出端口列表
output BZ_OUT;              //定义输出端口
input CLK;                  //定义输入端口
reg[31:0] COUNTER;          //定义 COUNTER 为寄存器类型的 32 位变量
reg BZ_OUT;                 //定义 BZ_OUT 为寄存器类型的 1 位变量
//---------------------------------------
//每当 CLK 产生上升沿时,执行一遍 begin_end 块内的语句
always@(posedge CLK)
 begin                              //begin_end 块开始
   COUNTER<=COUNTER+1'b1;           //计数器 COUNTER 加 1
 end

always@(COUNTER[8])
 begin
   BZ_OUT<=!(COUNTER[12]& COUNTER[23]&COUNTER[28]) //驱动蜂鸣器
 end                                //// begin_end 块结束

endmodule                           //模块结束
```

引脚分配见表 13-6。器件编译通过后，可根据需要进行仿真。最后将*.jed 文件下载到 XC95108 芯片中。

表 13-6　　报警声实验引脚分配

引脚名	引脚号	输入或输出	板上丝印符号
CLK	9	Input	—
BZ_OUT	I9	Output	LED7

在 MCU&CPLD DEMO 试验板上，将一个短路块插到 BEEP 排针上（连通蜂鸣器的驱动电路），我们就能听到清脆的间断性的“滴、滴…”报警声了。

13.7　简易电子琴实验

13.7.1　实验要求

按下 MCU&CPLD DEMO 试验板上的 K0~K3 键，蜂鸣器能发出简谱的中音 1~中音 4 的声音。

13.7.2　原理设计

前面已经提到过，交流蜂鸣器只要加入不同频率的脉冲，就能发出不同音调的声音。

表 13-7 为简谱中的音名与频率的关系。MCU&CPLD DEMO 试验板上的有源晶振频率为 24MHz。例如，为了发出中音 1 的音调，我们应当进行分频，分频系数为：

```
24000000÷523.3÷2=22931
```

据此，我们可计算出简谱中不同音名的分频系数，见表 13-8。

表 13-7　　简谱中的音名与频率的关系

音　名	频率（Hz）	音　名	频率（Hz）	音　名	频率（Hz）
低音 1	261.6	中音 1	523.3	高音 1	1046.5
低音 2	293.7	中音 2	587.3	高音 2	1174.7
低音 3	329.6	中音 3	659.3	高音 3	1318.5
低音 4	349.2	中音 4	698.5	高音 4	1396.9
低音 5	392	中音 5	784	高音 5	1568
低音 6	440	中音 6	880	高音 6	1760
低音 7	493.9	中音 7	987.8	高音 7	1975.5

表 13-8　　简谱中不同音名的分频系数

音　名	分频系数	音　名	分频系数	音　名	分频系数
低音 1	45872	中音 1	22931	高音 1	11467
低音 2	40858	中音 2	20432	高音 2	10215
低音 3	36408	中音 3	18201	高音 3	9101

续表

音　名	分频系数	音　名	分频系数	音　名	分频系数
低音 4	34364	中音 4	17180	高音 4	8590
低音 5	30612	中音 5	15306	高音 5	7653
低音 6	27273	中音 6	13636	高音 6	6818
低音 7	24296	中音 7	12148	高音 7	6074

我们只需通过计数器分频后得到各音名的驱动频率脉冲，当某个按键按下时用相应频率的脉冲去驱动蜂鸣器，那么蜂鸣器就会发出对应频率的音调。

13.7.3 程序设计

在 E 盘中先建立一个文件名为“SOUND”的文件夹，然后建立一个“SOUND”的新项目，输入以下的源代码并保存为“SOUND.v”。

```
module SOUND(BZ_OUT,KEY,CLK);          //模块声明及输入输出端口列表
output BZ_OUT;                         //定义输出端口
input CLK;                             //定义输入端口
input[3:0] KEY;                        //定义输入端口
//定义 COUNTER 和 COUNTER_END 为寄存器类型的 15 位变量
reg[14:0] COUNTER,COUNTER_END;
reg OUT_FLAG;              //定义 OUT_FLAG 为寄存器类型的 1 位变量
reg BZ_OUT;                //定义 BZ_OUT 为寄存器类型的 1 位变量
reg[3:0] KEY_STATUS;       //定义 KEY_STATUS 为寄存器类型的 4 位变量
//-----------------------------------------
//每当 CLK 产生上升沿时,执行一遍 begin_end 块内的语句
always@(posedge CLK)
 begin                     //begin_end 块开始
    KEY_STATUS=KEY;        //读取键值
    case(KEY_STATUS)       //case 语句,根据 KEY_STATUS 的值,产生散转分支
    //KEY_STATUS 为 0111 时,COUNTER_END 赋值 22931,OUT_FLAG 置 1
    4'b0111:begin COUNTER_END<=15'd22931;OUT_FLAG<=1'b1;end
    //KEY_STATUS 为 1011 时,COUNTER_END 赋值 20432,OUT_FLAG 置 1
    4'b1011:begin COUNTER_END<=15'd20432;OUT_FLAG<=1'b1;end
    //KEY_STATUS 为 1101 时,COUNTER_END 赋值 18201,OUT_FLAG 置 1
    4'b1101:begin COUNTER_END<=15'd18201;OUT_FLAG<=1'b1;end
    //KEY_STATUS 为 1110 时,COUNTER_END 赋值 17180,OUT_FLAG 置 1
    4'b1110:begin COUNTER_END<=15'd17180;OUT_FLAG<=1'b1;end
    //默认情况下,COUNTER_END 赋值 32768(无脉冲输出),OUT_FLAG 清 0
    default:begin COUNTER_END<=15'd32768;OUT_FLAG<=1'b0;end
```

```
    endcase                    // case 语句结束
  end                          // begin_end 块结束
  //-----------------------------------------
//每当 CLK 产生上升沿时,执行一遍 begin_end 块内的语句
always@(posedge CLK)
 begin                                  //begin_end 块开始
   COUNTER<=COUNTER+1'b1;               //计数器 COUNTER 加 1
     if(COUNTER==COUNTER_END)           //计数到分频值时
         begin
         COUNTER<=15'd0;                //计数器清 0
         if(OUT_FLAG==1'b1)             //如果输出标志 OUT_FLAG 等于 1
            BZ_OUT<=~BZ_OUT;            //输出端取反,驱动蜂鸣器
         else                           //否则未计数到分频值时
            BZ_OUT<=1'b1;               //输出端置 1
         end
  end                                   //begin_end 块结束

endmodule                               //模块结束
```

引脚分配见表 13-9。器件编译通过后，可根据需要进行仿真。最后将*.jed 文件下载到 XC95108 芯片中。

表 13-9　　简易电子琴实验引脚分配

引脚名	引脚号	输入或输出	板上丝印符号
CLK	9	Input	—
K0	36	Input	K0
KI	35	Input	K1
K2	34	Input	K2
K3	33	Input	K3
BZ_OUT	I9	Output	LED7

在 MCU&CPLD DEMO 试验板上，将一个短路块插到 BEEP 排针上（连通蜂鸣器的驱动电路），分别按下 K0~K3 键，蜂鸣器即发出中音 1~中音 4 的音频声。

13.8　自动演奏乐曲实验

13.8.1　实验要求

要求设计一个能自动演奏乐曲的设备。能自动演奏歌曲“新年好”中的片段。

下面是歌曲“新年好”中的一段简谱：

1=C 1 1 1 $\underline{5}$ | 3 3 3 1 | 1 3 5 5 | 4 3 2 — |

13.8.2 原理设计

上面一个实验我们已经知道如何发出对应频率的音调了。

实际上，音乐中的音符除了有高低（音调）之分外，还有长短之分。这里引用一个基础的音乐术语——拍子。拍子是表示音符长短的重要概念。

表示音符的长短需要有一个相对固定的时间概念。简谱里将音符分为全音符、二分音符、四分音符、八分音符、十六分音符等。在这几个音符里最重要的是四分音符，它是一个基本参照度量单位，即四分音符为一拍。一拍是一个相对时间度量单位，一拍的长度没有限制，可以是1s，也可以是0.5s或0.25s。如果一拍的长度是0.25s的话，则各节拍的音长见表13-10。

表13-10　　各节拍的音长

节拍符号	X	X	X·	X	X·	X-	X---
名称	十六分音符	八分音符	八分符点音符	四分音符	四分符点音符	二分音符	全音符
拍数	1/4拍	1/2拍	3/4拍	1拍	1又1/2拍	2拍	4拍
音长	0.0625s	0.125s	0.1875s	0.25s	0.375s	0.5s	1s

我们可以对24MHz的有源晶振频率进行3000000分频（24000000/8=3000000），得到8Hz（周期0.125s）的信号，然后再每0.125s取反一次，这样就可得到4Hz（周期0.25s）的信号了。

歌曲“新年好”中的各音名的分频系数及预置数见表13-11。最大分频系数为30612，因此我们使用15位的计数器就足够了（最大计数值=2^{15}-1=32767）。这样也可以求出各预置数了（预置数=32767-音名的分频系数）。

表13-11　　歌曲“新年好”中的各音名的分频系数及预置数

音　名	预置数	分频系数
中音1	9836	22931
中音2	12335	20432
中音3	14566	18201
中音4	15587	17180

续表

音　名	预置数	分频系数
中音 5	17461	15306
低音 5	2155	30612

13.8.3　程序设计

在 E 盘中先建立一个文件名为“SONG”的文件夹，然后建立一个“SONG”的新项目，输入以下的源代码并保存为“SONG.v”。

```
module SONG (BZ_OUT, CLK);          //模块声明及输入输出端口列表
output BZ_OUT;                      //定义输出端口
input CLK;                          //定义输入端口
//定义 COUNTER 和 COUNTER_END 为寄存器类型的 15 位变量,产生音调
reg[14:0] COUNTER, COUNTER_BEGIN;
reg[3:0] H,M,L;                     //高、中、低音寄存器
reg[4:0] LOOP_CNT;                  //循环演奏寄存器
reg CLK_4;                          //4Hz 时钟寄存器
wire CA;                            //进位信号
reg[23:0] CNT;                      //产生 4Hz 信号的计数器
reg BZ_OUT;                         //定义 BZ_OUT 为寄存器类型的 1 位变量
//-----------------------------------------
//每当 CLK 产生上升沿时,执行一遍 begin_end 块内的语句
always@(posedge CLK)
 begin                              //begin_end 块开始
    CNT= CNT+1;
    if(CNT==24'd3000000)
        begin
            CNT=24'd0;
            CLK_4=~ CLK_4;          //产生 4Hz 信号
        end
  end                               //begin_end 块结束
  assign CA=( COUNTER==32767);          //计数满值时,产生进位信号
//-----------------------------------------
//每当 CLK 产生上升沿时,执行一遍 begin_end 块内的语句
always@(posedge CLK)
 begin                                  //begin_end 块开始
   if(CA) COUNTER<= COUNTER_BEGIN;      //计数满值时,重装预置值
```

```
    else COUNTER<= COUNTER+1;              //计数
end                                        // begin_end 块结束
//-------------------------------------------
//每当 CA 产生上升沿时,执行一遍 begin_end 块内的语句
always@(posedge CA)
 begin                                 //begin_end 块开始
    BZ_OUT=~ BZ_OUT;                   //发声
end                                    //begin_end 块结束
//-------------------------------------------
//每当 CLK_4 产生上升沿时,执行一遍 begin_end 块内的语句
always@( posedge CLK_4)
 begin                                 //begin_end 块开始
    case({H,M,L})                      //case 语句,产生散转分支
     12'b000000000101: COUNTER_BEGIN <=2155;  //低音 5 预置数
     12'b000000010000: COUNTER_BEGIN <=9836;  //中音 1 预置数
     12'b000000100000: COUNTER_BEGIN <=12335; //中音 2 预置数
     12'b000000110000: COUNTER_BEGIN <=14566; //中音 3 预置数
     12'b000001000000: COUNTER_BEGIN <=15587; //中音 4 预置数
     12'b000001010000: COUNTER_BEGIN <=17461; //中音 5 预置数
     12'b000000000000: COUNTER_BEGIN <=32767; //休止符
     endcase                           // case 语句结束
end                                    // begin_end 块结束
//-------------------------------------------
//每当 CLK_4 产生上升沿时,执行一遍 begin_end 块内的语句
always@( posedge CLK_4)
 begin                                 //begin_end 块开始
     if(LOOP_CNT==32) LOOP_CNT<=0;     //循环计数,以实现循环演奏
     else LOOP_CNT<= LOOP_CNT+1;
    case(LOOP_CNT)                          //case 语句,产生散转分支
     0:{H,M,L} =12'b000000010000; //中音 1,一个节拍
     1:{H,M,L} =12'b000000010000; //中音 1,一个节拍
     2:{H,M,L} =12'b000000010000; //中音 1,二个节拍
     3:{H,M,L} =12'b000000010000;
     4:{H,M,L} =12'b000000000101; //低音 5,二个节拍
     5:{H,M,L} =12'b000000000101;

     6:{H,M,L} =12'b000000110000; //中音 3,一个节拍
     7:{H,M,L} =12'b000000110000; //中音 3,一个节拍
     8:{H,M,L} =12'b000000110000; //中音 3,二个节拍
     9:{H,M,L} =12'b000000110000;
```

```
    10:{H,M,L} =12'b000000010000; //中音 1,二个节拍
    11:{H,M,L} =12'b000000010000;

    12:{H,M,L} =12'b000000010000; //中音 1,一个节拍
    13:{H,M,L} =12'b000000110000; //中音 3,一个节拍
    14:{H,M,L} =12'b000001010000; //中音 5,二个节拍
    15:{H,M,L} =12'b000001010000;
    16:{H,M,L} =12'b000001010000; //中音 5,二个节拍
    17:{H,M,L} =12'b000001010000;

    18:{H,M,L} =12'b000001000000; //中音 4,一个节拍
    19:{H,M,L} =12'b000000110000; //中音 3,一个节拍
    20:{H,M,L} =12'b000000100000; //中音 2,四个节拍
    21:{H,M,L} =12'b000000100000;
    22:{H,M,L} =12'b000000100000;
    23:{H,M,L} =12'b000000100000;
    default:{H,M,L} =12'b000000000000;//停止
    endcase                          // case 语句结束
end                                  // begin_end 块结束

endmodule                            //模块结束
```

引脚分配见表 13-12。器件编译通过后，可根据需要进行仿真。最后将*.jed 文件下载到 XC95108 芯片中。

表 13-12　　简易电子琴实验引脚分配

引脚名	引脚号	输入或输出	板上丝印符号
CLK	9	Input	—
BZ_OUT	19	Output	LED7

在 MCU&CPLD DEMO 试验板上，将一个短路块插到 BEEP 排针上（连通蜂鸣器的驱动电路），蜂鸣器会自动演奏歌曲“新年好”中的片段。

13.9　D/A 转换器实验

13.9.1　实验要求

用一个 8 位的计数器自动输出 PWM 调宽信号，实现 D/A 转换实验。

13.9.2 程序设计

在 E 盘中先建立一个文件名为“PWM_DA”的文件夹，然后建立一个“PWM_DA”的新项目，输入以下的源代码并保存为“PWM_DA.v”。

```
module PWM_DA (PWM_OUT, CLK);        //模块声明及输入输出端口列表
output PWM_OUT;                      //定义输出端口
input CLK;                           //定义输入端口
reg[21:0] COUNT;    //定义 COUNTER 为寄存器类型的 22 位变量
reg[7:0] PWM_CNT;                    //PWM 信号翻转控制值(数据值)
reg CLK_16;                          //16Hz 时钟寄存器
reg[19:0] CNT;                       //产生 16Hz 信号的计数器
reg PWM_OUT;         //定义 PWM_OUT 为寄存器类型的 1 位变量
//---------------------------------------
//每当 CLK 产生上升沿时,执行一遍 begin_end 块内的语句
always@(posedge CLK)
 begin                          //begin_end 块开始
    COUNT= COUNT +1;            //22 位计数器计数
    //其中的 8 位计数值小于 PWM 信号翻转控制值,输出高电平
    if(COUNT [16:9]< PWM_CNT) PWM_OUT =1;
    else PWM_OUT=0;//否则 8 位计数值大于 PWM 信号翻转控制值,输出低电平
  end                           // begin_end 块结束
//---------------------------------------
//每当 CLK 产生上升沿时,执行一遍 begin_end 块内的语句
always@(posedge CLK)
 begin                          //begin_end 块开始
    CNT= CNT+1;                 //计数器计数
    if(CNT==20'd750000)         //当计数值为 750000 时
        begin
             CNT=20'd0;         //计数清零
             CLK_16=~ CLK_16; //产生 16Hz 信号
        end
  end                           // begin_end 块结束
//---------------------------------------
//每当 CLK_16 产生上升沿时,执行一遍 begin_end 块内的语句
always@(posedge CLK_16)
   begin                        //begin_end 块开始
    PWM_CNT= PWM_CNT+1;         //PWM 信号翻转控制值增加
```

```
    end                                    //begin_end 块结束

endmodule                                  //模块结束
```

引脚分配见表 13-13。器件编译通过后，可根据需要进行仿真。最后将*.jed 文件下载到 XC95108 芯片中。

表 13-13　　D/A 转换器实验引脚分配

引脚名	引脚号	输入或输出	板上丝印符号
CLK	9	Input	
PWM_OUT	19	Output	LED7

在 MCU&CPLD DEMO 试验板上，将一个短路块插到 BEEP 排针上，我们看到发光管 LED7 的亮度会周期性地变化，同时蜂鸣器的音调也发生变化，说明输出信号的 PWM 值在周期性地变化。如果我们在输出端增加一个电容对信号进行积分平滑，那么就可以得到模拟信号了。

13.10　D/F 转换器实验

13.10.1　实验要求

按下按键 K0 及 K1 后，产生调频信号，实现 D/F 转换实验。

13.10.2　程序设计

在 E 盘中先建立一个文件名为“PWM_DF”的文件夹，然后建立一个“PWM_DF”的新项目，输入以下的源代码并保存为“PWM_DF.v”。

```
module PWM_DF(PWM_OUT, CLK,K0,K1);     //模块声明及输入输出端口列表
output PWM_OUT;                        //定义输出端口
input CLK;                             //定义输入端口
input K0,K1;                           //定义输入端口(两个按键)
reg[21:0] COUNT;                       //定义 COUNTER 为寄存器类型的 22 位变量
reg[7:0] PWM_CNT;                      //PWM 信号翻转控制值(数据值)
reg CLK_16;                            //16Hz 时钟寄存器
reg[19:0] CNT;                         //产生 16Hz 信号的计数器
reg PWM_OUT;                           //定义 PWM_OUT 为寄存器类型的 1 位变量
```

```
//-----------------------------------------
//每当 CLK 产生上升沿时,执行一遍 begin_end 块内的语句
always@(posedge CLK)
  begin                          //begin_end 块开始
    COUNT= COUNT +1;             //22 位计数器计数
    //其中的 8 位计数值小于 PWM 信号翻转控制值,输出高电平
    if(COUNT [16:9]< PWM_CNT) PWM_OUT =1;
    else PWM_OUT=0;//否则 8 位计数值大于 PWM 信号翻转控制值,输出低电平
  end                            //begin_end 块结束
//-----------------------------------------
//每当 CLK 产生上升沿时,执行一遍 begin_end 块内的语句
always@(posedge CLK)
 begin                           //begin_end 块开始
    CNT= CNT+1;                  //计数器计数
    if(CNT==20'd750000)          //当计数值为 750000 时
        begin
             CNT=20'd0;          //计数清零
             CLK_16=~ CLK_16; //产生 16Hz 信号
        end
  end                            //begin_end 块结束
//-----------------------------------------
//每当 CLK_16 产生上升沿时,执行一遍 begin_end 块内的语句
always@(posedge CLK_16)
   begin                         //begin_end 块开始
    //当键 K0 按下时,PWM 信号翻转控制值增加
    if(!K0) PWM_CNT= PWM_CNT+1;
    //当键 K1 按下时,PWM 信号翻转控制值减小
    else if(!K1) PWM_CNT= PWM_CNT-1;
    end                          //begin_end 块结束

endmodule                        //模块结束
```

引脚分配见表 13-14。器件编译通过后，可根据需要进行仿真。最后将*.jed文件下载到 XC95108 芯片中。

表 13-14　　D/F 转换器实验引脚分配

引脚名	引脚号	输入或输出	板上丝印符号
CLK	9	Input	
K0	36	Input	K0

续表

引脚名	引脚号	输入或输出	板上丝印符号
K1	35	Input	K1
PWM_OUT	19	Output	LED7

在 MCU&CPLD DEMO 试验板上，将一个短路块插到 BEEP 排针上。按下 K0 键，我们看到发光管 LED7 的亮度会逐渐变暗；按下 K1 键，我们看到发光管 LED7 的亮度会逐渐变亮。在亮度变化时蜂鸣器的音调也同时发生变化，说明输出信号的 PWM 值在发生变化，同时它的频率也在发生变化。

13.11　RS232 收发实验

13.11.1　实验要求

在 PC 机上使用串口调试软件发送一个字节，MCU&CPLD DEMO 试验板收到后驱动发光二极管进行相应的指示，同时将收到的数据发回 PC 机。

13.11.2　原理设计

串口收发器的结构组成方框图如图 13-1 所示。我们以波特率 9600、数据位 8 位、停止位 1 位为例进行设计。因为 MCU&CPLD DEMO 试验板上的有源晶振频率为 24MHz，所以波特率 9600 的分频数是 24000000÷9600=2500，波特率 9600 的分频数一半是 24000000÷9600÷2=1250。

平时，RS232_RX 为高电平。当串口接收电路检测到输入端 RS232_RX 出现下降沿时，将 bps_start_rx 置高，启动接收波特率发生器，产生波特率时钟脉冲 clk_bps_rx，同时开始从 RS_232RX 端接收串行数据，并且同步启动接收移位次数计数器工作，每个接收波特率时钟的高电平接收一位数据。在标准的接收模式下，有 1+8+1（2 或 3）=12 位的有效数据。收到停止位后，接收移位次数计数器清零，并将数据锁存到输出寄存器中，驱动发光管指示。

在串口接收电路收到数据期间时，将接收数据中断信号 rx_int 置高，根据此信号启动发送波特率发生器，产生波特率时钟脉冲 clk_bps_tx，从 RS232_TX 端将接收到的数据发送回去，并且同步启动发送移位次数计数器工作。每个发送波特率时钟的高电平发送一位数据。在标准的发送模式下，有 1+8+1(2)=11

位的有效数据。发送结束后，发送移位次数计数器清零。

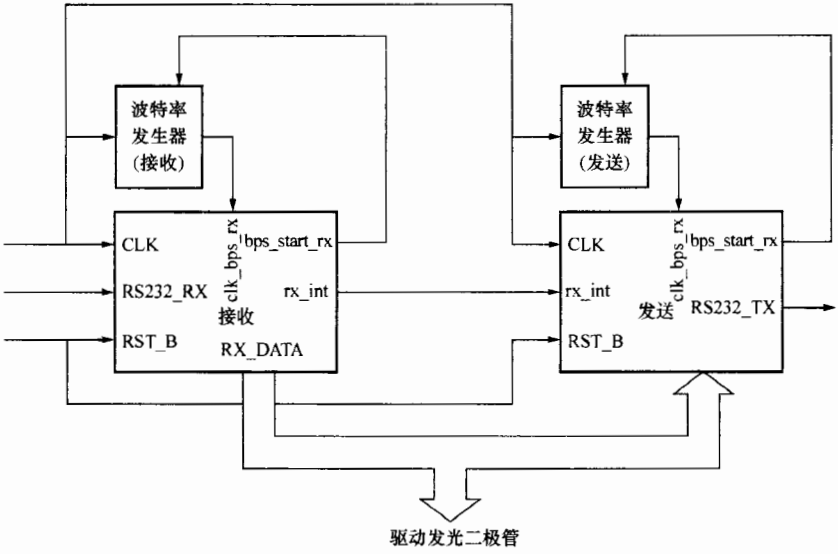


图 13-1　串口收发器的结构组成方框图

13.11.3　程序设计

在 E 盘中先建立一个文件名为“RS232”的文件夹，然后建立一个“RS232”的新项目，输入以下的源代码并保存为“RS232.v”。

```
//模块声明及输入输出端口列表
module RS232(CLK,RST_B,RS232_RX,RX_DATA,RS232_TX);

/***********接收部分***************/
input CLK;                  //定义输入端口,时钟信号
input RST_B;                //定义输入端口,复位信号
input RS232_RX;             //定义输入端口,RS232 接收端
output RS232_TX;            //定义输入端口,RS232 发送端
output[7：0] RX_DATA;       //定义输出端口,发光管驱动端
//----------------------------------------
reg rs232_rx0,rs232_rx1,rs232_rx2,rs232_rx3;   //定义寄存器类型的
                                                1 位变量
wire neg_rs232_rx;          //网线型 1 位变量,表示数据线接收到脉冲下降沿
reg[12：0] cnt_rx;          //定义寄存器类型的 13 位变量
```

```
reg clk_bps_rx;     //定义 clk_bps_rx 为寄存器类型的 1 位变量(接收波特
                      率时钟)
reg bps_start_rx;   //定义 bps_start_rx 为寄存器类型的 1 位变量(接收波
                      特率启动)
reg bps_start_tx;   //定义 bps_start_tx 为寄存器类型的 1 位变量(发送波
                      特率启动)
//----------------------------------------
//每当 CLK 产生上升沿或 RST_B 产生下降沿时
always @ (posedge CLK or negedge RST_B)
    if(!RST_B)          //如果复位端为低电平
       cnt_rx<=13'd0; //计数器清零
    //否则如果计数值为 2500 或波特率关闭,接收计数器清零
    //2500 是波特率为 9600 时的分频计数值
    else if((cnt_rx==2500)||!bps_start_rx)  cnt_rx<=13'd0;
    else  cnt_rx <= cnt_rx+1'b1; //否则接收波特率时钟计数启动
//----------------------------------------
//每当 CLK 产生上升沿或 RST_B 产生下降沿时
always @ (posedge CLK or negedge RST_B)
    if(!RST_B)              //如果复位端为低电平
       clk_bps_rx<=1'b0;  // clk_bps_rx 清零
    //否则如果计数值为 1250,clk_bps_rx 置高一个时钟周期
    //1250 是波特率为 9600 时的分频计数值的一半,用于数据采样
    else if(cnt_rx==1250)  clk_bps_rx<=1'b1;
    else clk_bps_rx <= 1'b0; //否则 clk_bps_rx 清零
//----------------------------------------
//每当 CLK 产生上升沿或 RST_B 产生下降沿时
always @ (posedge CLK or negedge RST_B)
begin
    if(!RST_B)                      //如果复位端为低电平
        begin
            rs232_rx0 <= 1'b0;  //清零
            rs232_rx1 <= 1'b0;  //清零
            rs232_rx2 <= 1'b0;  //清零
            rs232_rx3 <= 1'b0;  //清零
        end
    else begin                      //否则
            rs232_rx0 <= RS232_RX; //接收后滤波(抗干扰)
            rs232_rx1 <= rs232_rx0;//接收后滤波(抗干扰)
```

```
                rs232_rx2 <= rs232_rx1;//接收后滤波(抗干扰)
                rs232_rx3 <= rs232_rx2;//接收后滤波(抗干扰)
            end
    end
    //下降沿检测,如果接收到下降沿后,在第4个CLK后neg_rs232_rx置高一个CLK
    assign neg_rs232_rx=rs232_rx3&rs232_rx2&~rs232_rx1&~rs232_rx0;

    //-----------------------------------------
    reg[3:0] num_rx;    //定义num_rx为寄存器类型的4位变量,用于接收移位次
                          数计数
    reg rx_int;         //接收数据中断信号,接收到数据期间始终为高电平
    //-----------------------------------------
    //每当CLK产生上升沿或RST_B产生下降沿时
    always @ (posedge CLK or negedge RST_B)
        if(!RST_B)                          //如果RST_B为低电平
            begin
                bps_start_rx <= 1'bz;   //置bps_start_rx为高阻
                rx_int <= 1'b0;         //清零接收数据中断信号
            end
        else if(neg_rs232_rx) //否则如果接收到串口接收线RS232_RX的下降
                              沿启动信号
            begin
                bps_start_rx <= 1'b1;   //启动串口的接收波特率准备数
                                          据接收
                rx_int <= 1'b1;         //接收数据中断信号使能
            end
        else if(num_rx==4'd12)              //如果接收完有用数据信息
            begin
                bps_start_rx <= 1'b0;   //数据接收完毕,关闭接收波特
                                          率启动信号
                rx_int <= 1'b0;         //接收数据中断信号关闭
            end
    //-----------------------------------------
    reg[7:0] rx_data_rx;        //定义串口接收数据寄存器,保存直至下一次数
                                  据来到
    //-----------------------------------------
    reg[7:0] rx_temp_data;          //定义当前接收数据寄存器
    //-----------------------------------------
    //每当CLK产生上升沿或RST_B产生下降沿时
```

```
always @ (posedge CLK or negedge RST_B)
    if(!RST_B)                          //如果 RST_B 为低电平
        begin
            rx_temp_data <= 8'd0;       //清空当前接收数据寄存器
            num_rx <= 4'd0;             //接收移位次数计数器清零
            rx_data_rx <= 8'd0;         //清空串口接收数据寄存器
        end
    else if(rx_int)                     //否则如果接收到数据
        //读取并保存数据,接收数据为一个起始位,8 位数据,1~3 个结束位
        begin
        if(clk_bps_rx)                  //当接收波特率时钟为高电平时
            begin
                num_rx <= num_rx+1'b1;//接收移位次数计数器递加
                case (num_rx) //case 语句,根据 num_rx 的值,产生
                                散转分支
                        4'd1:   rx_temp_data[0]<=RS232_RX;
                          //锁存第 1 位
                        4'd2:   rx_temp_data[1]<=RS232_RX;
                          //锁存第 2 位
                        4'd3:   rx_temp_data[2]<=RS232_RX;
                          //锁存第 3 位
                        4'd4:   rx_temp_data[3]<=RS232_RX;
                          //锁存第 4 位
                        4'd5:   rx_temp_data[4]<=RS232_RX;
                          //锁存第 5 位
                        4'd6:   rx_temp_data[5]<=RS232_RX;
                          //锁存第 6 位
                        4'd7:   rx_temp_data[6]<=RS232_RX;
                          //锁存第 7 位
                        4'd8:   rx_temp_data[7]<=RS232_RX;
                          //锁存第 8 位
                        default: ;
                endcase   // case 语句结束
            end
        else if(num_rx == 4'd12) //如果接收移位次数计数值为 12
            begin
                num_rx <= 4'd0; //接收到停止位后结束,接收移位次
                                  数计数器清零
                rx_data_rx <= rx_temp_data;
```

```
                                        //把数据锁存到数据寄存器rx_data_rx中
            end
        end

assign RX_DATA = rx_data_rx;  //接收的数据输出到 RX_DATA 端口上,驱
                                动 LED
/************发送部分**************/
reg[12:0] cnt_tx;//定义 cnt_tx 为寄存器类型的 13 位变量,分频后用于发送
                 波特率计数器
reg clk_bps_tx;  //定义 clk_bps_tx 为寄存器类型的 1 位变量(发送波特率
                 时钟)
//-----------------------------------------
//每当 CLK 产生上升沿或 RST_B 产生下降沿时
always@(posedge CLK or negedge RST_B)
  if(!RST_B) cnt_tx<=13'd0; //如果 RST_B 为低电平,发送波特率计数清零
//否则如果计数值为 2500 或波特率关闭,发送计数器清零
//2500 是波特率为 9600 时的分频计数值
  else if((cnt_tx==2500)||!bps_start_tx)    cnt_tx<=13'd0;
  else    cnt_tx<=cnt_tx+1'b1; //否则发送波特率时钟计数启动
//-----------------------------------------
//每当 CLK 产生上升沿或 RST_B 产生下降沿时
always@(posedge CLK or negedge RST_B)
  if(!RST_B)clk_bps_tx<=1'b0;//如果 RST_B 为低电平,clk_bps_tx 清零
  //否则如果计数值为 1250,clk_bps_tx 置高一个时钟周期
  //1250 是波特率为 9600 时的分频计数值的一半,用于数据采样
  else if(cnt_tx==1250) clk_bps_tx<=1'b1;
  else  clk_bps_tx<=1'b0;      //否则 clk_bps_tx 清零
/************************************************/
reg rx_int0,rx_int1,rx_int2;  //接收数据寄存器,定义为寄存器类型的变
                              //量,滤波用
wire neg_rx_int;              //网线型变量,表示收到了数据
//-----------------------------------------
//每当 CLK 产生上升沿或 RST_B 产生下降沿时
always@(posedge CLK or negedge RST_B)
begin
  if(!RST_B)            //如果 RST_B 为低电平
    begin
    rx_int0<=1'b0;      //清零
    rx_int1<=1'b0;      //清零
```

```
    rx_int2<=1'b0;      //清零
    end
  else                  //否则
    begin
    rx_int0<=rx_int;    //滤波(抗干扰)
    rx_int1<=rx_int0;   //滤波(抗干扰)
    rx_int2<=rx_int1;   //滤波(抗干扰)
    end
end
//接收到数据后,在第 3 个 CLK 后 neg_rs232_rx 置高一个 CLK
assign neg_rx_int=~rx_int1&rx_int2;
//----------------------------------------
reg tx_en;          //定义 tx_en 为寄存器类型的 1 位变量(发送允许)
reg[3:0] num_tx;    //定义 num_tx 为寄存器类型的 4 位变量,用于发送移位次
                      数计数
//----------------------------------------
//每当 CLK 产生上升沿或 RST_B 产生下降沿时
always@(posedge CLK or negedge RST_B)
begin
  if(!RST_B)        //如果 RST_B 为低电平
    begin
    bps_start_tx<=1'bz;    //置 bps_start_tx 为高阻
    tx_en<=1'b0;           //禁止发送
    end
  else if(neg_rx_int)      //否则如果检测到数据输入
    begin
    bps_start_tx<=1'b1;    //启动串口的发送波特率准备发送
    tx_en<=1'b1;           //发送使能
    end
  else if(num_tx==4'd11)   //如果发送移位次数计数值为 11
    begin
    bps_start_tx<=1'b0;    //关闭发送波特率
    tx_en<=1'b0;           //禁止发送
    end
end
//----------------------------------------
reg rs232_tx_r;           //串口发送数据寄存器
//----------------------------------------
//每当 CLK 产生上升沿或 RST_B 产生下降沿时
```

```
always@(posedge CLK or negedge RST_B)
begin
  if(!RST_B)          //如果 RST_B 为低电平
    begin
    num_tx<=4'd0;  //发送移位次数计数器清零
    rs232_tx_r<=1'b1;       //串口发送数据寄存器置位
    end
  else if(tx_en)            //否则如果发送使能
    begin
        if(clk_bps_tx)      //当发送波特率时钟为高电平时
          begin
         num_tx<=num_tx+1'b1; //发送移位次数计数器递加
         case(num_tx) // case 语句,根据 num_tx 的值,产生散转分支
         4'd0:rs232_tx_r<=1'b0;            //发送开始位(低电平)
         4'd1:rs232_tx_r<=rx_data_rx[0]; //发送第 1 位
         4'd2:rs232_tx_r<=rx_data_rx[1]; //发送第 2 位
         4'd3:rs232_tx_r<=rx_data_rx[2]; //发送第 3 位
         4'd4:rs232_tx_r<=rx_data_rx[3]; //发送第 4 位
         4'd5:rs232_tx_r<=rx_data_rx[4]; //发送第 5 位
         4'd6:rs232_tx_r<=rx_data_rx[5]; //发送第 6 位
         4'd7:rs232_tx_r<=rx_data_rx[6]; //发送第 7 位
         4'd8:rs232_tx_r<=rx_data_rx[7]; //发送第 8 位
         4'd9:rs232_tx_r<=1'b1;            //发送停止位(高电平)
         default:rs232_tx_r<=1'b1;    //默认发送停止位(高电平)
         endcase                 //case 语句结束
          end
       //如果发送移位次数计数值为 11,则发送移位次数计数器清零
        else if(num_tx==4'd11) num_tx<=4'd0;
    end
end

assign RS232_TX=rs232_tx_r;    //持续赋值语句输出发送数据

endmodule                          //模块结束
```

引脚分配见表 13-15。器件编译通过后，可根据需要进行仿真。最后将*.jed 文件下载到 XC95108 芯片中。

表 13-15　　RS232 收发实验引脚分配

引脚名	引脚号	输入或输出	板上丝印符号
CLK	9	Input	
RST_B	74	Input	GCLR
RS232_RX	53	Input	
RS232_TX	52	Output	
RX_DATA 7	19	Output	LED7
RX_DATA 6	20	Output	LED6
RX_DATA 5	21	Output	LED5
RX_DATA 4	23	Output	LED4
RX_DATA 3	24	Output	LED3
RX_DATA 2	25	Output	LED2
RX_DATA 1	26	Output	LED1
RX_DATA 0	31	Output	LED0

串口线的一端插在 MCU&CPLD DEMO 试验板上 CPLD_DB9 插座上，另一端插到 PC 机的串口上。为了防止传输混乱而收不到数据，首先按一下 GCLR 键，将发送缓冲区的乱码清除。然后在 PC 机上，打开串口调试器软件，清空发送、接收区内的数据，发送、接收均选择“按 16 进制显示”，然后点击“打开串口”按钮。发送区输入 aa，点击“发送”，我们看到试验板上 LED7-LED0 立刻间隔点亮，与此同时接收区也立刻显示出了收到的“aa”数据。

13.12　数字跑表实验

13.12.1　实验要求

按住K0键后，进行数字跑表计时实验。放开K0键后暂停计时。按下GCKLR键后，清除计时值。数码管的个、十位进行百分秒显示；百、千位进行秒显示；万、十万位进行分的显示。

13.12.2　程序设计

在 E 盘中先建立一个文件名为“STOPWATCH”的文件夹，然后建立一个“STOPWATCH”的新项目，输入以下的源代码并保存为“STOPWATCH.v”。

```
module STOPWATCH (SEG,SEL, CLK,CLR,PAUSE);//模块声明及输入输出端口列表
output[7:0] SEG;              //定义输出端口
output[7:0] SEL;              //定义输出端口
input CLK;                    //定义输入端口
input CLR;                    //定义输入端口
input PAUSE;                  //定义输入端口
reg CLK_100;                  //100Hz 时钟寄存器
reg[7:0] SEG;                 //定义 SEG 为寄存器类型的 8 位变量
reg[3:0] SEG_BUF;             //定义 SEG_BUF 为寄存器类型的 4 位变量
reg[5:0] SEL;                 //定义 SEL 为寄存器类型的 6 位变量
reg[17:0] COUNTER;            //定义 COUNTER 为寄存器类型的 18 位变量
reg[2:0] STATUS;              //定义 STATUS 为寄存器类型的 3 位变量
//分、秒、百分秒的计时寄存器
reg[3:0] MIN_H,MIN_L,SEC_H,SEC_L,MSEC_H,MSEC_L;
//---------------------------------------
//每当 CLK 产生上升沿时
always@(posedge CLK)
 begin
   COUNTER=COUNTER+1'b1;              //计数器 COUNTER 加 1
        if(COUNTER==18'd120000)       //如果 5 毫秒到了
           begin
            CLK_100=~CLK_100;         //产生 100Hz 信号
            COUNTER=18'd0;
           end
 end
//---------------------------------------
//每当 CLK_100 产生上升沿时或 CLR 产生下降沿时
always@(posedge CLK_100 or negedge CLR )
 begin          // begin_end 块开始
   if(!CLR)     //如果 CLR 为低电平
         begin
          MSEC_H=4'd0;MSEC_L=4'd0;    //百分秒清零
          SEC_H=4'd0;SEC_L=4'd0;      //秒清零
          MIN_H=4'd0;MIN_L=4'd0;      //分清零
         end
    else if(!PAUSE)                   //PAUSE 为低电平时计时
         begin
              MSEC_L=MSEC_L+1;        //百分秒个位计数
```

```
                if(MSEC_L==10)              //如果百分秒个位计数到 10
                 begin
                  MSEC_L=0;MSEC_H=MSEC_H+1;//百分秒个位清零、十位计数
                   if(MSEC_H==10)           //如果百分秒十位计数到 10
                       begin
                        MSEC_H=0;SEC_L=SEC_L+1;  //百分秒十位清零、
                                                   秒个位计数
                        if(SEC_L==10)   //如果秒个位计数到 10
                            begin       //秒个位清零、秒十位计数
                              SEC_L=0;SEC_H=SEC_H+1;
                                  if(SEC_H==6)//如果秒十位计数到 6
                               begin    //秒十位清零、分个位计数
                                  SEC_H=0;MIN_L=MIN_L+1;
                                 if(MIN_L==10)//如果分个位计数到10
                                     begin //分个位清零、分十位计数
                                        MIN_L=0;MIN_H=MIN_H+1;
                                          //如果分十位计数到 6,保
                                            持计数到 6
                                         if(MIN_H>=6) MIN_H=6;
                                     end
                               end
                              end
                           end
                       end
      end
 end                          // begin_end 块结束

//---------------------------------------
//每当 COUNTER[11:9]发生变化时
always@(COUNTER[11:9])
 begin
   case(COUNTER[11:9])      //case 语句,根据 COUNTER[11:9]的值,产生
                              散转分支
   3'd0:SEG_BUF<= MSEC_L; //送出百分秒的低位
   3'd1:SEG_BUF<= MSEC_H; //送出百分秒的高位
   3'd2:SEG_BUF<= SEC_L;  //送出秒的低位
   3'd3:SEG_BUF<= SEC_H;  //送出秒的高位
   3'd4:SEG_BUF<= MIN_L;  //送出分的低位
   3'd5:SEG_BUF<= MIN_H;  //送出分的高位
```

```
    endcase                    // case 语句结束
  end
//---------------------------------------
//每当 SEG_BUF 发生变化时
always@(SEG_BUF)
  begin
    case(SEG_BUF)          //case 语句,根据 SEG_BUF 的值,产生散转分支
    4'd0:SEG<=8'h3f;       //送出"0"的字段码
    4'd1:SEG<=8'h06;       //送出"1"的字段码
    4'd2:SEG<=8'h5b;       //送出"2"的字段码
    4'd3:SEG<=8'h4f;       //送出"3"的字段码
    4'd4:SEG<=8'h66;       //送出"4"的字段码
    4'd5:SEG<=8'h6d;       //送出"5"的字段码
    4'd6:SEG<=8'h7d;       //送出"6"的字段码
    4'd7:SEG<=8'h07;       //送出"7"的字段码
    4'd8:SEG<=8'h7f;       //送出"8"的字段码
    4'd9:SEG<=8'h6f;       //送出"9"的字段码
    default:SEG<=8'hzz;    //默认状态下,数据口为高阻
    endcase                //  case 语句结束
  end
//---------------------------------------
//每当 COUNTER[11:9]发生变化时
always@(COUNTER[11:9])
  begin
    case(COUNTER[11:9])        //case 语句,根据 COUNTER[11:9]的值,产生
                               //散转分支
    3'd0:SEL<=6'b000001;       //点亮个位数码管
    3'd1:SEL<=6'b000010;       //点亮十位数码管
    3'd2:SEL<=6'b000100;       //点亮百位数码管
    3'd3:SEL<=6'b001000;       //点亮千位数码管
    3'd4:SEL<=6'b010000;       //点亮万位数码管
    3'd5:SEL<=6'b100000;       //点亮十万位数码管
    3'd6:SEL<=6'b000000;       //关闭显示
    endcase                    //  case 语句结束
  end
endmodule                      //模块结束
```

引脚分配见表 13-16。器件编译通过后，可根据需要进行仿真。最后将*.jed 文件下载到 XC95108 芯片中。

表 13-16　　数字跑表实验引脚分配

引脚名	引脚号	输入或输出	板上丝印符号
CLK	9	Input	
CLR	74	Input	
PAUSE	36	Input	K0
SEG7	65	Output	
SEG6	66	Output	
SEG5	77	Output	
SEG4	68	Output	
SEG3	69	Output	
SEG2	70	Output	
SEG1	71	Output	
SEG0	72	Output	
SEL 5	61	Output	
SEL 4	58	Output	
SEL 3	57	Output	
SEL 2	56	Output	
SEL 1	55	Output	
SEL 0	54	Output	

MCU&CPLD DEMO 试验板上电以后，我们按下 K0 按键，这时数字跑表开始运行起来计时。释放 K0，时间被冻结暂停。按一下 GCLR 键，时间清零，可重新开始计时。

13.13　数字电子钟实验

13.13.1　实验要求

数字电子钟显示的时间范围为：000000~235959，可以循环计时。

说明：　000000 代表 00 时 00 分 00 秒，235959 代表 23 时 59 分 59 秒。时间过 23 时 59 分 59 秒后，又从 00 时 00 分 00 秒开始计时，反复循环。

13.13.2　原理设计

设定一个时间计数器，每隔 0.5 秒产生一个秒标志（即每秒产生一个脉冲），

然后分别用秒、分、时计数器计时。计时时必须遵循时间规律，即秒、分按60进制，时按24进制。另外还要设定一个数码管显示用的扫描计数器，每隔一定时间（例如 0.7ms）点亮一位数码管，循环扫描点亮。这样由于扫描的时间很快，8 位数码管的扫描周期还不到 6ms，远低于人眼视觉暂留特性的限定值，故可以看到稳定的显示。

13.13.3 程序设计

在 E 盘中先建立一个文件名为“CLOCK”的文件夹，然后建立一个“CLOCK”的新项目，输入以下的源代码并保存为“CLOCK.v”。

```
module CLOCK(SEG,SEL,CLK);     //模块声明及输入输出端口列表
output[7:0] SEG;   //定义输出端口
output[5:0] SEL;   //定义输出端口
input CLK;         //定义输入端口
//-------------------------------------------------
reg[7:0] SEG_REG;  //定义 SEG_REG 为寄存器类型的 8 位变量
reg[3:0] SEG_BUF;  //定义 SEG_BUF 为寄存器类型的 4 位变量
reg[5:0] SEL_REG;  //定义 SEL_REG 为寄存器类型的 6 位变量
reg[13:0] DIS_COUNTER;   //定义 DIS_COUNTER 为寄存器类型的 14 位变量
reg[23:0] TIME_COUNTER; //定义 TIME_COUNTER 为寄存器类型的 24 位变量
reg[2:0] DIS_STATUS;     //定义 DIS_STATUS 为寄存器类型的 3 位变量
reg[7:0] HOUR,MIN,SEC;   //定义 HOUR、MIN、SEC 为寄存器类型的 8 位变量
reg SEC_FLAG;            //定义 SEC_FLAG 为寄存器类型的 1 位变量
//-------------------------------------------------
always@(posedge CLK)   //每当 CLK 产生上升沿时，执行一遍 begin_end 块
                       //内的语句
 begin                 //begin_end 块开始
   TIME_COUNTER=TIME_COUNTER+1'b1;    //计数器 TIME_COUNTER 递加
   if(TIME_COUNTER==24'd12000000)     //如果 0.5s 到了
    begin
     TIME_COUNTER=24'd0;              //计数器 TIME_COUNTER 清零
     SEC_FLAG=~SEC_FLAG;              //秒标志取反
    end
end                    //begin_end 块结束
//-------------------------------------------------
//每当 SEC_FLAG 产生上升沿时，执行一遍 begin_end 块内的语句
always@(posedge SEC_FLAG)
```

```
begin                                          //begin_end 块开始
     SEC[3:0]=SEC[3:0]+1'b1;                   //秒递加
        if(SEC[3:0]==4'd10)                    //如果秒的个位计数为 10
        begin
         SEC[3:0]=4'd0;                        //秒个位清零
         SEC[7:4]=SEC[7:4]+1'b1;               //秒十位计数
         //-------------------------------------
           if(SEC[7:4]==4'd6)                  //如果秒的十位计数为 6
            begin
              SEC[7:4]=4'd0;                   //秒十位清零
              MIN[3:0]=MIN[3:0]+1'b1;          //分个位递加
            //-------------------------------------
            if(MIN[3:0]==4'd10)                //如果分的个位计数为 10
              begin
               MIN[3:0]=4'd0;                  //分个位清零
               MIN[7:4]=MIN[7:4]+1'b1;         //分十位递加
                //-------------------------------------
               if(MIN[7:4]==4'd6)              //如果分的十位计数为 6
                 begin
                  MIN[7:4]=4'd0;               //分十位清零
                  HOUR[3:0]=HOUR[3:0]+1'b1;//时个位递加
                   //-----------------------------------
                   //如果时的计数为 24
                   if((HOUR[7:4]==4'd2)&&(HOUR[3:0]==4'd4))
                         begin
                           HOUR[7:4]=4'd0;  //时十位清零
                           HOUR[3:0]=4'd0;  //时个位清零
                         end
                    else if(HOUR[3:0]==4'd10) //否则如果时个位
                                              //为 10
                         begin
                           HOUR[3:0]=4'd0;  //时个位清零
                           HOUR[7:4]=HOUR[7:4]+1'b1;//时十位
                                                    //递加
                         end
                 end
              end
            end
        end
```

```
 end                          //begin_end 块结束
//-------------------------------------------------
//每当 CLK 产生上升沿时,执行一遍 begin_end 块内的语句
always@(posedge CLK)
 begin                        //begin_end 块开始
   DIS_COUNTER<=DIS_COUNTER+1'b1;     //计数器 DIS_COUNTER 递加
     if(DIS_COUNTER==14'b11111111111111) //如果 0.7 毫秒到了
       begin
       DIS_STATUS=DIS_STATUS+1'b1;        //状态 DIS_STATUS 递加
         if(DIS_STATUS==3'd6)DIS_STATUS=0; //状态 DIS_STATUS 在
                                           //0～5 之间循环
       end
  end                     // begin_end 块结束
//-------------------------------------------------
//每当 CLK 产生上升沿时,执行一遍 begin_end 块内的语句
always@(posedge CLK)
 begin                     //begin_end 块开始
   case(DIS_STATUS)        //case 语句,根据 DIS_STATUS 的值,产生散转分支
   3'd0:SEG_BUF<=SEC[3:0];     //送出秒的低位
   3'd1:SEG_BUF<=SEC[7:4];     //送出秒的高位
   3'd2:SEG_BUF<=MIN[3:0];     //送出分的低位
   3'd3:SEG_BUF<=MIN[7:4];     //送出分的高位
   3'd4:SEG_BUF<=HOUR[3:0];    //送出时的低位
   3'd5:SEG_BUF<=HOUR[7:4];    //送出时的高位
   endcase                 //case 语句结束
 end                       //begin_end 块结束
//-------------------------------------------------
//每当 CLK 产生上升沿时,执行一遍 begin_end 块内的语句
always@(posedge CLK)
 begin                     //begin_end 块开始
   case(SEG_BUF)           //case 语句,根据 SEG_BUF 的值,产生散转分支
   4'd0:SEG_REG<=8'h3f;    //送出"0"的字段码
   4'd1:SEG_REG<=8'h06;    //送出"1"的字段码
   4'd2:SEG_REG<=8'h5b;    //送出"2"的字段码
   4'd3:SEG_REG<=8'h4f;    //送出"3"的字段码
   4'd4:SEG_REG<=8'h66;    //送出"4"的字段码
   4'd5:SEG_REG<=8'h6d;    //送出"5"的字段码
   4'd6:SEG_REG<=8'h7d;    //送出"6"的字段码
   4'd7:SEG_REG<=8'h07;    //送出"7"的字段码
```

```
    4'd8:SEG_REG<=8'h7f;          //送出"8"的字段码
    4'd9:SEG_REG<=8'h6f;          //送出"9"的字段码
    default:SEG_REG<=8'hzz;       //默认状态下,数据口为高阻
    endcase                       // case 语句结束
  end                             // begin_end 块结束
//-------------------------------------------------
//每当 CLK 产生上升沿时,执行一遍 begin_end 块内的语句
always@(posedge CLK)
  begin                    //begin_end 块开始
    case(DIS_STATUS)       //case 语句,根据 DIS_STATUS 的值,产生散转分支
    3'd0:SEL_REG<=8'b00000001; //点亮个位数码管
    3'd1:SEL_REG<=8'b00000010; //点亮十位数码管
    3'd2:SEL_REG<=8'b00000100; //点亮百位数码管
    3'd3:SEL_REG<=8'b00001000; //点亮千位数码管
    3'd4:SEL_REG<=8'b00010000; //点亮万位数码管
    3'd5:SEL_REG<=8'b00100000; //点亮十万位数码管
    default:SEL_REG<=6'hzz;    //默认状态下,位选口为高阻
    endcase                    //case 语句结束
  end                          //begin_end 块结束
//-------------------------------------------------
  assign SEG=SEG_REG;          //持续赋值语句,输出到数码管的数据口
  assign SEL=SEL_REG;          //持续赋值语句,输出到数码管的位选口
endmodule                      //模块结束
```

引脚分配见表 13-17。器件编译通过后，可根据需要进行仿真。最后将*.jed 文件下载到 XC95108 芯片中。

表 13-17　　数字电子钟引脚分配

引脚名	引脚号	输入或输出	板上丝印符号
CLK	9	Input	—
SEG7	65	Output	—
SEG6	66	Output	—
SEG5	77	Output	—
SEG4	68	Output	—
SEG3	69	Output	—
SEG2	70	Output	—
SEG1	71	Output	—
SEG0	72	Output	—

续表

引脚名	引脚号	输入或输出	板上丝印符号
SEL 5	61	Output	—
SEL 4	58	Output	—
SEL 3	57	Output	—
SEL 2	56	Output	—
SEL 1	55	Output	—
SEL 0	54	Output	—

MCU&CPLD DEMO 试验板上电以后，我们看到右边 6 个数码管从“000000”开始计时，非常准确。

13.14 交通信号灯实验

13.14.1 实验要求

在一个十字路口，需要安置自动的交通信号灯。其中 A 路为主要的交通干道；B 路为次要的交通干道。要求 A 路的绿灯为 25s，红灯为 10s。B 路的绿灯为 10s，红灯为 25s。绿灯与红灯转换之间有 5s 的黄灯时间。并且，在 A 路口与 B 路口均有两位的数码管进行时间倒数显示。

13.14.2 程序设计

在 E 盘中先建立一个文件名为“TRA_LIGHT”的文件夹，然后建立一个“TRA_LIGHT”的新项目，输入以下的源代码并保存为“TRA_LIGHT.v”。

```
module TRA_LIGHT(CLK,LED,SEG,SEL); //模块声明及输入输出端口列表
input CLK;                  //定义输入端口
output[7:0] LED;            //定义输出端口
output[7:0] SEG,SEL;        //定义输出端口
reg[7:0] LED;               //定义 LED 为寄存器类型的 8 位变量
reg[7:0] SEG,SEL;           //定义 SEG、SEL 为寄存器类型的 8 位变量
reg[23:0] CNT;              //定义 CNT 为寄存器类型的 24 位变量
reg[3:0] DIS_BUFF;          //定义 DIS_BUFF 为寄存器类型的 4 位变量
reg SEC;                    //定义 SEC 为寄存器类型的 1 位变量
reg[2:0] STATUS;            //定义 STATUS 为寄存器类型的 3 位变量
```

```
reg[7:0] TIME;          //定义 TIME 为寄存器类型的 8 位变量

parameter               //常量定义
                        YELLOE_1=0,     //黄灯 1 为 0
                        GREEN=1,        //绿灯为 1
                        RED=2,          //红灯为 2
                        YELLOE_2=3;     //黄灯 2 为 3
//-------------------------------------------------------
//每当 CLK 产生上升沿时
always @ (posedge CLK)
  begin
  CNT=CNT+1;                    //计数器计数
   if (CNT==24'd12000000)       //如果 0.5s 到了
   begin CNT=0;SEC=~SEC;end     //计数器清零,秒标志取反
  end
//-------------------------------------------------------
//每当 SEC 产生上升沿时
always @ (posedge SEC)
  begin
    case (STATUS)       //case 语句,根据 STATUS 的值,产生散转分支
          //-----------------------------------------------
          YELLOE_1:     //如果主干道是黄灯 1 状态
            if((TIME[7:4]==0)&&(TIME[3:0]==0))   //如果时间到 0
               begin
                STATUS<=GREEN;          //主干道将转入绿灯状态
                TIME[7:4]<=2;TIME[3:0]<=5;//置主干道绿灯时间 25s
                LED<=8'b01111110;       //主干道绿灯亮,次干道红灯亮
               end
            else        //否则时间不到 0
               begin
                TIME[3:0]<=TIME[3:0]-1;             //时间递减
                if(TIME[3:0]==0)begin TIME[3:0]<=9;TIME[7:4]<
                   =TIME[7:4]-1;end
               end
          //-----------------------------------------------
          GREEN:        //如果主干道是绿灯状态
           if((TIME[7:4]==0)&&(TIME[3:0]==0))    //如果时间到 0
              begin
```

```
        STATUS<=YELLOE_2;  //主干道将转入黄灯 2 状态
        TIME[7:4]<=0;TIME[3:0]<=5;//置主干道黄灯时间 5s
        LED<=8'b10111101;  //主干道黄灯亮,次干道黄灯亮
       end
     else                        //否则时间不到 0
       begin
        TIME[3:0]<=TIME[3:0]-1;    //时间递减
        if(TIME[3:0]==0)begin TIME[3:0]<=9;TIME[7:4]<=
        TIME[7:4]-1;end
       end
//-------------------------------------------
YELLOE_2:              //如果主干道是黄灯 2 状态
  if((TIME[7:4]==0)&&(TIME[3:0]==0))   //如果时间到 0
      begin
       STATUS<=RED;  //主干道将转入红灯状态
       TIME[7:4]<=1;TIME[3:0]<=0;//置主干道红灯时间 10s
       LED<=8'b11011011; //主干道红灯亮,次干道绿灯亮
      end
  else                 //否则时间不到 0
      begin
       TIME[3:0]<=TIME[3:0]-1;  //时间递减
       if(TIME[3:0]==0)begin  TIME[3:0]<=9;TIME[7:4]
       <=TIME[7:4]-1;end
      end

RED:                   //如果主干道是红灯状态
 if((TIME[7:4]==0)&&(TIME[3:0]==0))    //如果时间到 0
    begin
     STATUS<=YELLOE_1;  //主干道将转入黄灯 1 状态
     TIME[7:4]<=0;TIME[3:0]<=5;//置主干道黄灯时间 5s
     LED<=8'b10111101;  //主干道黄灯亮,次干道黄灯亮
    end
 else                  //否则时间不到 0
   begin
    TIME[3:0]<=TIME[3:0]-1;     //时间递减
    if(TIME[3:0]==0)begin  TIME[3:0]<=9;TIME[7:4]<=
    TIME[7:4]-1;end
   end
```

```
    endcase                          // case 语句结束
  end
//-------------------------------------------------------
//每当 CNT[13:12]有变化时
always @ (CNT[13:12])
 begin
  case (CNT[13:12])    //case 语句,根据 CNT[13:12]的值,产生散转分支
    2'b00:DIS_BUFF=TIME[3:0]; //送出时间的低位
    2'b01:DIS_BUFF=TIME[7:4]; //送出时间的高位
    2'b10:DIS_BUFF=TIME[3:0]; //送出时间的低位
    2'b11:DIS_BUFF=TIME[7:4]; //送出时间的高位
    endcase                          //case 语句结束
 end
//-------------------------------------------------------
//每当 CNT[13:12]有变化时
always @ (CNT[13:12])
 begin
  case (CNT[13:12])    //case 语句,根据 CNT[13:12]的值,产生散转分支
    2'b00:SEL=8'b00000001;     //点亮主干道的数码管个位
    2'b01:SEL=8'b00000010;     //点亮主干道的数码管十位
    2'b10:SEL=8'b01000000;     //点亮次干道的数码管个位
    2'b11:SEL=8'b10000000;     //点亮次干道的数码管十位
    endcase                          // case 语句结束
 end
//-------------------------------------------------------
//每当 DIS_BUFF 有变化时
always @ (DIS_BUFF)
 begin
  case (DIS_BUFF)       //case 语句,根据 DIS_BUFF 的值,产生散转分支
   4'd0:SEG<=8'h3f;     //送出"0"的字段码
   4'd1:SEG<=8'h06;     //送出"1"的字段码
   4'd2:SEG<=8'h5b;     //送出"2"的字段码
   4'd3:SEG<=8'h4f;     //送出"3"的字段码
   4'd4:SEG<=8'h66;     //送出"4"的字段码
   4'd5:SEG<=8'h6d;     //送出"5"的字段码
   4'd6:SEG<=8'h7d;     //送出"6"的字段码
   4'd7:SEG<=8'h07;     //送出"7"的字段码
   4'd8:SEG<=8'h7f;     //送出"8"的字段码
```

```
    4'd9:SEG<=8'h6f;      //送出"9"的字段码
     endcase              //case 语句结束
  end

endmodule               //模块结束
```

引脚分配见表 13-18。器件编译通过后，可根据需要进行仿真。最后将*.jed文件下载到 XC95108 芯片中。

表 13-18　　交通信号灯实验引脚分配

引脚名	引脚号	输入或输出	板上丝印符号
CLK	9	Input	—
SEG7	65	Output	—
SEG6	66	Output	—
SEG5	77	Output	—
SEG4	68	Output	—
SEG3	69	Output	—
SEG2	70	Output	—
SEG1	71	Output	—
SEG0	72	Output	—
SEL 5	61	Output	—
SEL 4	58	Output	—
SEL 3	57	Output	—
SEL 2	56	Output	—
SEL 1	55	Output	—
SEL 0	54	Output	—
LED0	31	Output	LED0
LED1	26	Output	LED1
LED2	25	Output	LED2
LED3	24	Output	LED3
LED4	23	Output	LED4
LED5	21	Output	LED5
LED6	20	Output	LED6
LED7	19	Output	LED7

MCU&CPLD DEMO 试验板上电以后，我们可以观察数码管的显示及 LED 的显示，与道路上交通灯的显示完全吻合。

13.15　4 位数字频率计实验

13.15.1　实验要求

要求设计一个简单的 4 位数字频率计，测频范围 0000~9999Hz。

13.15.2　程序设计

在 E 盘中先建立一个文件名为“FREQUENCY”的文件夹，然后建立一个“FREQUENCY”的新项目，输入以下的源代码并保存为“FREQUENCY.v”。

```
module FREQUENCY(SEG,SEL,CLK,FX); //模块声明及输入输出端口列表
output[7:0] SEG;          //定义输出端口
output[3:0] SEL;          //定义输出端口
input CLK,FX;             //定义输入端口
//-------------------------------------------------------
reg[7:0] SEG;             //定义 SEG_REG 为寄存器类型的 8 位变量
reg[7:0] SEG_BUF;         //定义 SEG_BUF 为寄存器类型的 8 位变量
reg[3:0] SEL;             //定义 SEL_REG 为寄存器类型的 4 位变量
reg[24:0] COUNTER;        //定义 COUNTER 为寄存器类型的 25 位变量
reg[15:0] DISP_CNT;       //显示计数器
reg[15:0] CNT;            //频率计数器
reg SEC_FLAG;             //定义 SEC_FLAG 为寄存器类型的 1 位变量
reg OVER_FLAG;            //定义 OVER_FLAG 为寄存器类型的 1 位变量
reg DATA_FLAG;            //定义 DATA_FLAG 为寄存器类型的 1 位变量
//------------------产生秒脉冲----------------------
always@(posedge CLK)   //每当 CLK 产生上升沿时
 begin
   COUNTER=COUNTER+1'b1;                  //计数器 COUNTER 递加
    if(COUNTER==25'd24000000)             //如果 1s 到了
    begin
     COUNTER=25'd0;                       //计数器 COUNTER 清零
     SEC_FLAG=~SEC_FLAG;                  //秒标志取反
    end
 end
//--------------------频率计数----------------------
always@(posedge FX)    //每当 FX 产生上升沿时
```

```
begin
  if(SEC_FLAG)                          //如果秒脉冲为高电平
   begin
   DATA_FLAG=1;                         //置位数据标志
   CNT[3:0]=CNT[3:0]+1'b1;              //脉冲计数,个位加 1
     if(CNT[3:0]==4'd10)                //如果个位满 10
      begin
       CNT [3:0]=4'd0;                  //个位清零
       CNT [7:4]= CNT [7:4]+1'b1; //十位加 1
            //================================
           if(CNT [7:4]==4'd10)  //如果十位满 10
              begin
              CNT [7:4]=4'd0;    //十位清零
              CNT [11:8]= CNT [11:8]+1'b1; //百位加 1
                 //*******************************
                if(CNT [11:8]==4'd10)       //如果百位满 10
                  begin
                    CNT [11:8]=4'd0;        //百位清零
                    CNT [15:12]= CNT [15:12]+1'b1; //千位加 1
                     //--------如果千位大于 10,置位 OVER_FLAG
                     if(CNT [15:12]>4'd9)   OVER_FLAG=1;
                      else OVER_FLAG=0; //否则清零 OVER_FLAG
                     end
                end
        end
   end
    else if(DATA_FLAG==1)               //如果数据标志为 1
    begin
     DATA_FLAG=0;                       //清除数据标志
     DISP_CNT[15:0]=CNT[15:0];          //将脉冲计数值送入显示缓冲区
     CNT[15:0]=0;                  //然后清除脉冲计数值,准备下一次测试
    end
 end
//-------------------取出计数值---------------------------
always@(COUNTER[12:11])    //每当 COUNTER[12:11]发生变化时
 begin
     case(COUNTER[12:11]) //case 语句,产生散转分支
      2'b00:SEG_BUF<= DISP_CNT [3:0]; //送出个位
      2'b01:SEG_BUF<= DISP_CNT [7:4]; //送出十位
```

```
    2'b10:SEG_BUF<= DISP_CNT [11:8];      //送出百位
    2'b11:SEG_BUF<= DISP_CNT [15:12];     //送出千位
    endcase                  //case 语句结束
 end
//-------------------取出字形码-------------------------
always@(SEG_BUF)             //每当 SEG_BUF 发生变化时
 begin
   if(!OVER_FLAG)            //如果频率计数没有溢出
    begin
     case(SEG_BUF)           //case 语句,根据 SEG_BUF 的值,产生散转分支
     4'd0:SEG <=8'h3f;       //送出"0"的字段码
     4'd1:SEG <=8'h06;       //送出"1"的字段码
     4'd2:SEG <=8'h5b;       //送出"2"的字段码
     4'd3:SEG <=8'h4f;       //送出"3"的字段码
      4'd4:SEG <=8'h66;      //送出"4"的字段码
     4'd5:SEG <=8'h6d;       //送出"5"的字段码
     4'd6:SEG <=8'h7d;       //送出"6"的字段码
     4'd7:SEG <=8'h07;       //送出"7"的字段码
     4'd8:SEG <=8'h7f;       //送出"8"的字段码
     4'd9:SEG <=8'h6f;       //送出"9"的字段码
     default:SEG <=8'hzz;    //默认状态下,数据口为高阻
     endcase                 //case 语句结束
    end
   else  SEG <=8'h40;        //否则有溢出,送出"-"的字段码
 end
//---------------------扫描四位数码管----------------
always@(COUNTER[12:11])      //每当 CNT[12:11]变化时
 begin
   case(COUNTER[12:11])      //case 语句,根据 CNT[12:11]的值进行散转
    2'b00:SEL <=4'b0001;     //点亮个位数码管
    2'b01:SEL <=4'b0010;     //点亮十位数码管
   2'b10:SEL <=4'b0100;      //点亮百位数码管
   2'b11:SEL <=4'b1000;      //点亮千位数码管
   default:SEL <=4'hzz;      //默认状态下,位选口为高阻
   endcase                   //case 语句结束
 end                         //begin_end 块结束
//-----------------------------------------------------

endmodule                    //模块结束
```

引脚分配见表 13-19。器件编译通过后，可根据需要进行仿真。最后将*.jed 文件下载到 XC95108 芯片中。

表 13-19　　4 位数字频率计实验引脚分配

引脚名	引脚号	输入或输出	板上丝印符号
CLK	9	Input	—
FX	10	Input	GCLK2
SEG7	65	Output	—
SEG6	66	Output	—
SEG5	77	Output	—
SEG4	68	Output	—
SEG3	69	Output	—
SEG2	70	Output	—
SEG1	71	Output	—
SEG0	72	Output	—
SEL 3	57	Output	—
SEL 2	56	Output	—
SEL 1	55	Output	—
SEL 0	54	Output	—

我们可以使用一台低频信号发生器，将低频信号加到 MCU&CPLD DEMO 试验板上，看看测试的准确性。

13.16　驱动 1602 字符型液晶显示器实验

13.16.1　实验要求

用 XC95108 设计 1602 液晶的驱动器，直接驱动 1602 液晶显示两行字符。

13.16.2　程序设计

在 E 盘中先建立一个文件名为“LCD_TEST”的文件夹，然后建立一个“LCD_TEST”的新项目，输入以下的源代码并保存为“LCD_TEST.v”。

```
module LCD_TEST(CLK,RS,RW,EN,LCD_DATA);//模块声明及输入输出端口列表
 input CLK;                              //定义输入端口
```

```
output [7:0] LCD_DATA;  //定义输出端口
output  RS,RW,EN;       //定义输出端口
reg RS;                 //定义 RS 为寄存器类型的 1 位变量
reg [7:0] LCD_DATA;     //定义 DAT 为寄存器类型的 8 位变量
reg  [15:0] COUNTER;    //定义 COUNTER 为寄存器类型的 16 位变量
reg [5:0] CURRENT,NEXT; //定义 CURRENT、NEXT 为寄存器类型的 6 位变量
reg CLK_LOW;            //定义 CLK_LOW 为寄存器类型的 1 位变量
reg [1:0] CNT;          //定义 CNT 为寄存器类型的 2 位变量

parameter SET0=6'd0,     //常量定义
          SET1=6'd1,  //常量定义
          SET2=6'd2,  //常量定义
          SET3=6'd3,  //常量定义
          SET4=6'd4,  //常量定义

          ROW1_0=6'd5,       //第 1 行第 1 个字符的地址常量定义
          ROW1_1=6'd6,       //第 1 行第 2 个字符的地址常量定义
          ROW1_2=6'd7,       //第 1 行第 3 个字符的地址常量定义
          ROW1_3=6'd8,       //第 1 行第 4 个字符的地址常量定义
          ROW1_4=6'd9,       //第 1 行第 5 个字符的地址常量定义
          ROW1_5=6'd10,      //第 1 行第 6 个字符的地址常量定义
          ROW1_6=6'd11,      //第 1 行第 7 个字符的地址常量定义
          ROW1_7=6'd12,      //第 1 行第 8 个字符的地址常量定义
          ROW1_8=6'd13,      //第 1 行第 9 个字符的地址常量定义
          ROW1_9=6'd14,      //第 1 行第 10 个字符的地址常量定义
          ROW1_10=6'd15,     //第 1 行第 11 个字符的地址常量定义
          ROW1_11=6'd16,     //第 1 行第 12 个字符的地址常量定义
          ROW1_12=6'd17,     //第 1 行第 13 个字符的地址常量定义
          ROW1_13=6'd18,     //第 1 行第 14 个字符的地址常量定义
          ROW1_14=6'd19,     //第 1 行第 15 个字符的地址常量定义
          ROW1_15=6'd20,     //第 1 行第 16 个字符的地址常量定义

          CH_ROW2=6'd21,     //换第 2 行定义

          ROW2_0=6'd22,      //第 2 行第 1 个字符的常量定义
          ROW2_1=6'd23,      //第 2 行第 2 个字符的常量定义
          ROW2_2=6'd24,      //第 2 行第 3 个字符的常量定义
          ROW2_3=6'd25,      //第 2 行第 4 个字符的常量定义
          ROW2_4=6'd26,      //第 2 行第 5 个字符的常量定义
```

```
                ROW2_5=6'd27,       //第 2 行第 6 个字符的常量定义
                ROW2_6=6'd28,       //第 2 行第 7 个字符的常量定义
                ROW2_7=6'd29,       //第 2 行第 8 个字符的常量定义
                ROW2_8=6'd30,       //第 2 行第 9 个字符的常量定义
                ROW2_9=6'd31,       //第 2 行第 10 个字符的常量定义
                ROW2_10=6'd32,      //第 2 行第 11 个字符的常量定义
                ROW2_11=6'd33,      //第 2 行第 12 个字符的常量定义
                ROW2_12=6'd34,      //第 2 行第 13 个字符的常量定义
                ROW2_13=6'd35,      //第 2 行第 14 个字符的常量定义
                ROW2_14=6'd36,      //第 2 行第 15 个字符的常量定义
                ROW2_15=6'd37,      //第 2 行第 16 个字符的常量定义

                NUL=6'd38;          //停止的常量定义
//-------------------------------------------------------
//每当 CLK 产生上升沿时,执行一遍 begin_end 块内的语句
always @(posedge CLK)
 begin                          //begin_end 块开始
  COUNTER=COUNTER+1;            //计数器 COUNTER 加 1
  if(COUNTER==16'h0000)         //每当计数值等于 0 时
  CLK_LOW=~CLK_LOW;             //CLK LOW 翻转(366Hz)
end                             //begin_end 块结束
//-------------------------------------------------------
always @(posedge CLK_LOW) //每当 CLKR 产生上升沿时
 CURRENT=NEXT;                  //将下一变量 NEXT 当前变量 CURRENT
//-------------------------------------------------------
always                          //执行 always 语句
begin
  case(CURRENT)                 //case 语句,根据 CURRENT 的值进行散转
     SET0:NEXT<=SET1;           //下一状态为 SET1
    SET1:NEXT<=SET2;            //下一状态为 SET
    SET2:NEXT<=SET3;            //下一状态为 SET3
    SET3:NEXT<=SET4;            //下一状态为 SET4
     SET4:NEXT<=ROW1_0;         //下一状态为 ROW1_0

     ROW1_0:NEXT<=ROW1_1;       //下一状态为 ROW1_1
    ROW1_1:NEXT<=ROW1_2;        //下一状态为 ROW1_2
     ROW1_2:NEXT<=ROW1_3;       //下一状态为 ROW1_3
     ROW1_3:NEXT<=ROW1_4;       //下一状态为 ROW1_4
     ROW1_4:NEXT<=ROW1_5;       //下一状态为 ROW1_5
```

```
      ROW1_5:NEXT<=ROW1_6;        //下一状态为 ROW1_6
      ROW1_6:NEXT<=ROW1_7;        //下一状态为 ROW1_7
      ROW1_7:NEXT<=ROW1_8;        //下一状态为 ROW1_8
     ROW1_8:NEXT<=ROW1_9;         //下一状态为 ROW1_9
     ROW1_9:NEXT<=ROW1_10;        //下一状态为 ROW1_10
     ROW1_10:NEXT<=ROW1_11;       //下一状态为 ROW1_11
      ROW1_11:NEXT<=ROW1_12;      //下一状态为 ROW1_12
      ROW1_12:NEXT<=ROW1_13;      //下一状态为 ROW1_13
     ROW1_13:NEXT<=ROW1_14;       //下一状态为 ROW1_14
     ROW1_14:NEXT<=ROW1_15;       //下一状态为 ROW1_15
     ROW1_15:NEXT<=CH_ROW2;       //下一状态为 ROW1_16

      CH_ROW2:NEXT<=ROW2_0;       //下一状态为 ROW2_0

      ROW2_0:NEXT<=ROW2_1;        //下一状态为 ROW2_1
     ROW2_1:NEXT<=ROW2_2;         //下一状态为 ROW2_2
      ROW2_2:NEXT<=ROW2_3;        //下一状态为 ROW2_3
      ROW2_3:NEXT<=ROW2_4;        //下一状态为 ROW2_4
      ROW2_4:NEXT<=ROW2_5;        //下一状态为 ROW2_5
      ROW2_5:NEXT<=ROW2_6;        //下一状态为 ROW2_6
      ROW2_6:NEXT<=ROW2_7;        //下一状态为 ROW2_7
     ROW2_7:NEXT<=ROW2_8;         //下一状态为 ROW2_8
     ROW2_8:NEXT<=ROW2_9;         //下一状态为 ROW2_9
      ROW2_9:NEXT<=ROW2_10;       //下一状态为 ROW2_10
      ROW2_10:NEXT<=ROW2_11;      //下一状态为 ROW2_11
     ROW2_11:NEXT<=ROW2_12;       //下一状态为 ROW2_12
     ROW2_12:NEXT<=ROW2_13;       //下一状态为 ROW2_13
     ROW2_13:NEXT<=ROW2_14;       //下一状态为 ROW2_14
      ROW2_14:NEXT<=ROW2_15;      //下一状态为 ROW2_15
      ROW2_15:NEXT<=NUL;          //下一状态为 NUL

     NUL: NEXT<=NUL;              //下一状态为 NUL

     default:NEXT<=SET0;          //默认下一状态为 SET0
     endcase                      //case 语句结束
   end
   //-------------------------------------------------------------
always @(posedge CLK_LOW)        //每当 CLK_LOW 产生上升沿时
begin
   case(CURRENT)          //case 语句,根据 CURRENT 的值进行散转
```

```
 SET0:RS<=0;              //当前状态 SET0 时,为命令传送
  CH_ROW2:RS<=0;          //当前状态 CH_ROW2 时,为命令传送
  NUL:RS<=0;              //当前状态 NUL 时,为命令传送
  ROW1_0:RS<=1;           //当前状态 ROW1_0 时,为数据传送
  ROW2_0:RS<=1;           //当前状态 ROW2_0 时,为数据传送
endcase                   //case 语句结束
//-----------------------------------------------------------
case(CURRENT)             //case 语句,根据 CURRENT 的值进行散转
   // CURRENT 为 SET0 时,选择液晶为 8 位数据传输,两行显示
   SET0:LCD_DATA<=8'h3c;
   // CURRENT 为 SET1 时,显示屏开启
   SET1:LCD_DATA<=8'h0c;
   // CURRENT 为 SET2 时,字符不移动
   SET2:LCD_DATA<=8'h06;
   // CURRENT 为 SET3 时,清除显示器的内容
   SET3:LCD_DATA<=8'h01;
   // CURRENT 为 SET4 时,从第 1 行第 1 个字符处开始写入内容
   SET4:LCD_DATA<=8'h80;
   //以下从第 1 行第 1 个字符处开始到 16 个字符处,依次写入内容
   ROW1_0:LCD_DATA<="-";
   ROW1_1:LCD_DATA<="X";
   ROW1_2:LCD_DATA<="C";
   ROW1_3:LCD_DATA<="9";
   ROW1_4:LCD_DATA<="5";
   ROW1_5:LCD_DATA<="1";
   ROW1_6:LCD_DATA<="0";
   ROW1_7:LCD_DATA<="8";
   ROW1_8:LCD_DATA<=" ";
   ROW1_9:LCD_DATA<=" ";
   ROW1_10:LCD_DATA<="T";
   ROW1_11:LCD_DATA<="e";
   ROW1_12:LCD_DATA<="s";
   ROW1_13:LCD_DATA<="t";
   ROW1_14:LCD_DATA<="!";
   ROW1_15:LCD_DATA<="-";
   //换行到第 2 行第 1 个字符处
   CH_ROW2:LCD_DATA<=8'hc0;
   //以下从第 2 行第 1 个字符处开始到 16 个字符处,依次写入内容
   ROW2_0:LCD_DATA<=" ";
```

```
        ROW2_1:LCD_DATA<="S";
        ROW2_2:LCD_DATA<="B";
        ROW2_3:LCD_DATA<="S";
        ROW2_4:LCD_DATA<="-";
        ROW2_5:LCD_DATA<="M";
        ROW2_6:LCD_DATA<="C";
        ROW2_7:LCD_DATA<="U";
        ROW2_8:LCD_DATA<=" ";
        ROW2_9:LCD_DATA<="S";
        ROW2_10:LCD_DATA<="t";
        ROW2_11:LCD_DATA<="u";
        ROW2_12:LCD_DATA<="d";
        ROW2_13:LCD_DATA<="i";
        ROW2_14:LCD_DATA<="o";
        ROW2_15:LCD_DATA<=" ";

      NUL:LCD_DATA<=8'h00;   //写完后,清除数据口
       endcase               //case 语句结束
end                          //begin_end 块结束

      assign EN=CLK_LOW;     //持续赋值语句,输出 EN 信号
      assign RW=0;           //持续赋值语句,输出 RW 信号(低电平为写入)

endmodule                    //模块结束
```

引脚分配见表 13-20。器件编译通过后，可根据需要进行仿真。最后将*.jed 文件下载到 XC95108 芯片中。

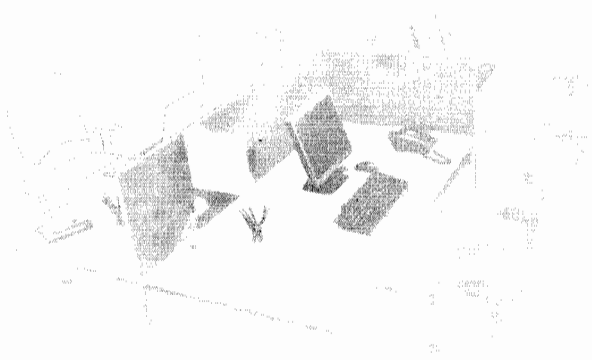
表 13-20　　驱动 1602 字符型液晶显示器实验引脚分配

引脚名	引脚号	输入或输出	板上丝印符号
CLK	9	Input	—
RS	7	Output	—
RW	6	Output	—
EN	5	Output	—
LCD_DATA0	4	Output	—
LCD_DATA1	3	Output	—
LCD_DATA2	2	Output	—
LCD_DATA3	1	Output	—

续表

引脚名	引脚号	输入或输出	板上丝印符号
LCD_DATA4	84	Output	—
LCD_DATA5	83	Output	—
LCD_DATA6	82	Output	—
LCD_DATA7	81	Output	—

将一个 1602 字符型液晶模组正确地插入 LCD16*2 单排座上（注意，液晶模组不要碰到 XC95108 芯片上，必要时可垫一层纸），MCU&CPLD DEMO 试验板上电以后，我们看到屏幕的第一行显示“-XC95108 Test!-”，第二行显示“SBS-MCU Studio”。



第 14 章

CPLD 与单片机的双向数据接口及应用

CPLD 与单片机之间的连接与通信，就相当于两台设备之间的连接与通信，需要遵循一定的格式与规定，这样才能保证实现快速准确高效的通信。MCU&CPLD DEMO 试验板与 AT89S51/52 单片机之间使用高速并行总线的方式进行通信。

14.1 CPLD 与单片机的双向数据接口连接及数据传输实验

14.1.1 实验要求

单片机读取 CPLD 外接的按键开关状态，然后将其状态取反后传回 CPLD 中，并点亮 8 个发光管进行显示。

14.1.2 原理设计

MCU&CPLD DEMO 试验板上 CPLD 与单片机接口连接及数据传输实验的结构组成方框图如图 14-1 所示。这里我们可以将 CPLD 当作单片机的外部 RAM 来进行读写操作。

在 CPLD 内构建 3 个 8 位的寄存器：ADDRESS_REG、KEY_REG、LED_REG。

ADDRESS_REG 内存放用于操作端口的地址指令。单片机使用 MOVX 指令对外部器件进行读写操作时，P0 口上将分时出现低 8 位地址与数据信号，其简明时序如图 14-2 所示。

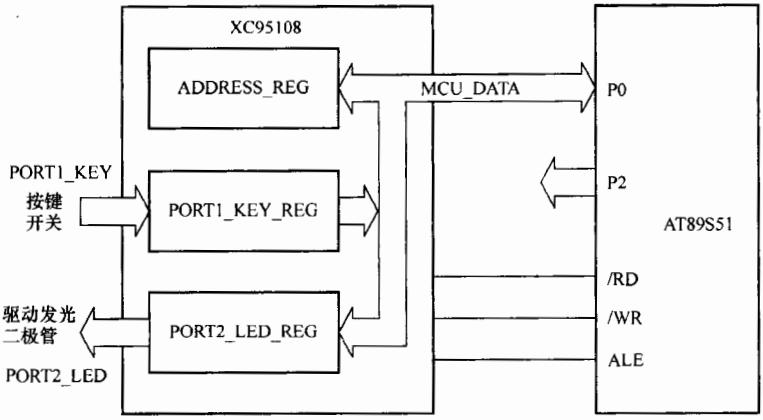


图 14-1　CPLD 与单片机接口连接的组成方框图

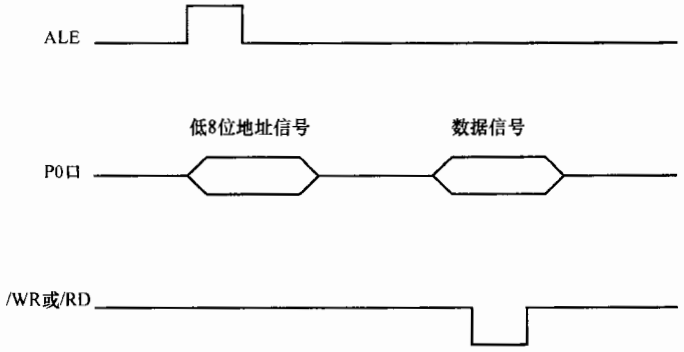


图 14-2　使用 MOVX 指令对外部器件进行读写的简明时序

P0 口出现低 8 位地址信号时，在 ALE 脉冲的下降沿，将低 8 位地址打入 CPLD 内的 ADDRESS_REG 寄存器，我们可以定义：低 8 位地址为 00000000 时，CPLD 读取按键开关的状态到内部的 KEY_REG 寄存器中，并将按键开关的状态数据送到总线上，在单片机随后发出的“/RD”脉冲下降沿，将此状态读入单片机中；低 8 位地址为 00000001 时，单片机将数据送到总线上，在单片机随后发出的“/WR”脉冲下降沿，CPLD 接收数据到其内部的 LED_REG 寄存器中，并驱动发光二极管进行指示。这样，完成了 CPLD 与单片机之间的数据高速双向传输。

14.1.3　CPLD 程序设计

在 E 盘中先建立一个文件名为“CPLD_MCU_RW”的文件夹，然后建立

一个“CPLD_MCU_RW”的新项目，输入以下的源代码并保存为“CPLD_MCU_RW.v”。

```
module CPLD_MCU_RW(DATA_PORT,RD,WR,ALE,KEY,LED);
inout[7:0] DATA_PORT;        //定义输入端口(8 位数据/地址总线)
input RD;                    //定义输入端口(读控制端)
input WR;                    //定义输入端口(写控制端)
input ALE;                   //定义输入端口(地址锁存控制端)
input[7:0] KEY;              //定义输入端口(按键输入端)
output[7:0] LED;             //定义输出端口(发光管输出端)
//-----------------------------------
reg[7:0] ADDRESS_REG;        //定义 ADDRESS_REG 为寄存器类型的 8 位变量
reg[7:0] KEY_REG;            //定义 KEY_REG 为寄存器类型的 8 位变量
reg[7:0] LED_REG;            //定义 LED_REG 为寄存器类型的 8 位变量
//-----------------------------------
always@(negedge ALE)         //每当 ALE 产生下降沿时
begin
ADDRESS_REG=DATA_PORT;//将总线上的低 8 位地址锁入 ADDRESS_REG 寄存器
end
//-----------------------------------
always@(negedge RD)          //每当 RD 产生下降沿时
begin
  if(ADDRESS_REG ==8'b00000000)   //如果地址为 00000000
KEY_REG=KEY;                 //读取按键开关的状态
else                         //否则
KEY_REG=8'bzzzzzzzz;         // KEY_REG 寄存器置高阻
end
//-----------------------------------
always@(negedge WR)          //每当 WR 产生下降沿时
begin
if(ADDRESS_REG ==8'b00000001)      //如果地址为 00000001
LED_REG= DATA_PORT;          //读取总线的数据并点亮发光二极管
end
//-----------------------------------
assign DATA_PORT =RD? 8'bzzzzzzzz:KEY_REG; //持续赋值语句
assign LED=LED_REG;          //持续赋值语句
//-----------------------------------
endmodule
```

引脚分配见表 14-1。器件编译通过后，可根据需要进行仿真。最后将*.jed 文件下载到 XC95108 芯片中。

表 14-1　CPLD 与单片机的双向数据接口连接及数据传输实验引脚分配

引脚名	引脚号	输入或输出	板上丝印符号
CLK	9	Input	—
RD	18	Input	—
WR	17	Input	—
ALE	11	Input	—
KEY7	33	Input	—
KEY6	34	Input	—
KEY5	35	Input	—
KEY4	36	Input	—
KEY3	37	Input	—
KEY2	39	Input	—
KEY1	40	Input	—
KEY0	41	Input	—
LED7	19	Output	LED7
LED6	20	Output	LED6
LED5	21	Output	LED5
LED4	23	Output	LED4
LED3	24	Output	LED3
LED2	25	Output	LED2
LED1	26	Output	LED1
LED0	31	Output	LED0
DATA_PORT 7	81	Inout	—
DATA_PORT 6	82	Inout	—
DATA_PORT 5	83	Inout	—
DATA_PORT 4	84	Inout	—
DATA_PORT 3	1	Inout	—
DATA_PORT 2	2	Inout	—
DATA_PORT 1	3	Inout	—
DATA_PORT 0	4	Inout	—

14.1.4　单片机 C 程序设计

在 E 盘中先建立一个文件名为“MCU_CPLD_RW”的文件夹，然后建立一个“MCU_CPLD_RW”的新项目，输入以下的源代码并保存为“MCU_CPLD_RW.v”。

```
#include<REG51.H>                    //包含头文件
#include<ABSACC.H>                   //包含头文件
#define uchar unsigned char          //数据类型的宏定义
#define uint unsigned int            //数据类型的宏定义
//-------------------------------------------------------
#define RD_CPLD_KEY_REG XBYTE[0x0000]  //读 CPLD 的地址宏定义
#define WR_CPLD_LED_REG XBYTE[0x0001]  //写 CPLD 的地址宏定义
//******************延时子程序**************************
void delay(uint k)                   //延时 k*1ms 的子程序
{
uint i,j;
for(i=0;i<k;i++){
for(j=0;j<120;j++)
{;}}
}
/*******************主程序**************************/
void main(void)                      //定义主函数
{
uchar key_val;                       //定义局部变量
  while(1)                           //无限循环
  {
    key_val= RD_CPLD_KEY_REG;        //读取 CPLD 外接的按键开关状态
    delay(50);                       //延时 50ms
    //将按键开关状态取反后送回 CPLD 点亮 LED
    WR_CPLD_LED_REG =~key_val;
    delay(50);                       //延时 50ms
  }
}                                    //主函数结束
```

在 MCU&CPLD DEMO 试验板上，我们按下 K0～K3 键或拨动 S0～S3 开关，可以观察到 LED0～LED7 这 8 个发光二极管的亮灭会发生变化，说明按键的状态已经被传输到单片机中，经单片机处理后又传送给 XC95108，然后驱动 LED 进行指示。

14.2 长时间多曲自动演奏器设计与实验

14.2.1 实验要求

单片机与CPLD建立双向通信联系后，单片机将已编码的乐曲数据发送给CPLD。CPLD构成一台音频发生器，根据接收的数据驱动蜂鸣器进行各种乐曲的演奏。

14.2.2 原理设计

1. 音调控制的设计

在上一章中，我们已经介绍过简谱的有关知识。为了便于理解，这里我们再将简谱中不同音名的频率及分频系数进行介绍。

表14-2为简谱中的音名与频率的关系。MCU&CPLD DEMO试验板上的有源晶振频率为24MHz。例如，为了发出中音1的音调，我们应当进行直接分频，分频系数为

$$24000000 \div 523.3 \div 2 = 22931$$

但是如果对简谱中的所有音名都直接分频，则XC95108的内部资源是远远不够的。怎么办呢？我们还可以采样别的办法：即先在XC95108内部构建一个预分频器，将24MHz的有源晶振频率降低到1MHz，然后再将简谱中的所有音名对1MHz的频率进行分频，这样就满足了设计要求。计算出的简谱中所有音名对1MHz频率的分频系数见表14-3。

表14-2 简谱中的音名与频率的关系

音　名	频率（Hz）	音　名	频率（Hz）	音　名	频率（Hz）
低音1	261.6	中音1	523.3	高音1	1046.5
低音2	293.7	中音2	587.3	高音2	1174.7
低音3	329.6	中音3	659.3	高音3	1318.5
低音4	349.2	中音4	698.5	高音4	1396.9
低音5	392	中音5	784	高音5	1568
低音6	440	中音6	880	高音6	1760
低音7	493.9	中音7	987.8	高音7	1975.5

表 14-3　　简谱中不同音名的分频系数

音　名	分频系数	音　名	分频系数	音　名	分频系数
低音 1	1911	中音 1	955	高音 1	956
低音 2	1702	中音 2	851	高音 2	426
低音 3	1517	中音 3	758	高音 3	379
低音 4	1432	中音 4	716	高音 4	358
低音 5	1276	中音 5	638	高音 5	319
低音 6	1136	中音 6	568	高音 6	284
低音 7	1012	中音 7	1012	高音 7	253

2.音长控制的设计

在上一章中，我们也讲过，音乐中的音符除了有音调之分外，还有长短之分——拍子，拍子是表示音符长短的重要概念。各节拍的分类见表 14-4。

表 14-4　　简谱中各节拍的分类

节拍符号	X	X	X·	X	X·	X-	X---
名称	十六分音符	八分音符	八分符点音符	四分音符	四分符点音符	二分音符	全音符
拍数	1/4 拍	1/2 拍	3/4 拍	1 拍	1 又 1/2 拍	2 拍	4 拍

一拍是一个相对时间度量单位，一拍的长度没有限制，可以是 1s，也可以是 0.5s 或 0.25s。

3. 单片机与 CPLD 的通信约定

了解了音调、音长的产生外，还需要解决单片机与 CPLD 的通信约定，即 CPLD 如何知道单片机发来的数据是产生何种音调的。这就需要制定音频信号的编码。

这里我们定义：单片机发送的每个字节数据中，高半字节代表低、中、高音，低半字节代表音调。高半字节的 1 代表低音；2 代表中音；3 代表高音。低半字节的 1~7 分别代表音调 1~7。例如：0x15 代表低音 5；0x24 代表中音 4；0x31 代表高音 1。

还有一个音长的问题，不过这个问题比较容易解决。我们可以直接由单片机解决。在定义乐曲的数组时，每两个字节代表一音符，前字节代表音调，传送给 CPLD 发音；后字节代表音长，控制单片机的定时器工作时间。单片机每

次读取两个字节，代表处理一个音符。一个音符完毕后，再读取两个字节，处理下一个音符。直到将整首乐曲处理完成。

音调、音长的编码组成举例如图 14-3 所示。

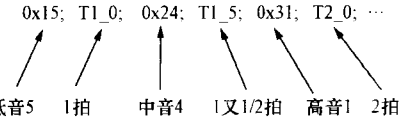


图 14-3 音调、音长的编码组成举例

前面的实验中，我们假设低 8 位地址为 0x00 时，单片机读取 CPLD 的信号；低 8 位地址为 0x01 时，单片机将数据传送到 CPLD 中。这里为了提高抗干扰能力及避开与音调数据可能的冲突，我们规定：低 8 位地址为 0x55 时，单片机读取 CPLD 的应答信号 0xf1；低 8 位地址为 0x88 时，单片机将音调数据传送到 CPLD 中。

4. 歌曲“两只老虎”的简谱及编码举例

下面是歌曲“两只老虎”中的一段简谱：

E4/4 ‖ 1 2 3 1 | 1 2 3 1 | 3 4 5— | 3 4 5 — |
两 只 老 虎 两只 老 虎 跑 得 快 跑 得 快
| 5 6 5 4 3 1 | 5 6 5 4 3 1 | 1 $\dot{5}$ 1— | 1 $\dot{5}$ 1 — ‖
一 只 没 有 耳朵 一只 没有 尾巴 真 奇怪 真奇怪

下面是对歌曲“两只老虎”进行的编码举例：

```
unsigned char code MUSIC1[]={
0x21,T1_0,0x22,T1_0,0x23,T1_0,0x21,T1_0,0x21,T1_0,0x22,T1_0,
0x23,T1_0,0x21,T1_0,0x23,T1_0,0x24,T1_0,0x25,T2_0,0x23,T1_0,
0x24,T1_0,0x25,T2_0,0x25,T0_75,0x26,T0_25,0x25,T0_75,0x24,T0_25,0
x23,T1_0,0x21,T1_0,0x25,T0_75,0x26,T0_25,0x25,T0_75,0x24,T0_25,
0x23,T1_0,0x21,T1_0,0x21,T1_0,0x15,T1_0,0x21,T2_0,0x21,T1_0,
0x15,T1_0,0x21,T2_0,0x00,T0_0};
```

14.2.3 CPLD 程序设计

在 E 盘中先建立一个文件名为“CPLD_MUSIC”的文件夹，然后建立一个“CPLD_MUSIC”的新项目，输入以下的源代码并保存为“CPLD_MUSIC.v”。

```
module CPLD_MUSIC(MUSIC_OUT,CLK,DATA_PORT,RD,WR,ALE);
                              //模块声明及输入输出端口列表
inout[7:0] DATA_PORT;         //定义输入端口(8 位地址/数据总线)
input RD;                     //定义输入端口(读控制端)
input WR;                     //定义输入端口(写控制端)
input ALE;                    //定义输入端口(地址锁存控制端)
output MUSIC_OUT;             //定义音乐输出端口
input CLK;                    //定义输入端口
reg MUSIC_OUT;                //定义 BZ_OUT 为寄存器类型的 1 位变量
//定义 COUNTER 和 COUNTER_END 为寄存器类型的 11 位变量
reg[10:0] COUNTER,COUNTER_BEGIN;
reg[7:0] ADDRESS_REG;         //定义 ADDRESS_REG 为寄存器类型的 8 位变量
reg[7:0] DATA_REG;            //定义 DATA_REG 为寄存器类型的 8 位变量
reg[7:0] ACK_REG;             //定义 ACK_REG 为寄存器类型的 8 位变量
reg[3:0] CNT;                 //定义 CNT 为寄存器类型的 4 位变量
reg CLK_1M;                   //定义 CLK_1M 为寄存器类型的 1 位变量
wire CA;                      //网线型变量

always@(posedge CLK)          //每当 CLK 产生上升沿时
 begin                        //begin_end 块开始
    CNT=CNT+1;                //计数器递加
  if(CNT==4'd12)              //当计数值为 12 时
      begin
           CNT=4'd0;          //计数值清零
           CLK_1M=~CLK_1M;//产生 1MHz 信号
      end
  end                         //begin_end 块结束

assign CA=(COUNTER==2047); //COUNTER 计数器为 2047 时,产生进位信号
//-------------------------------------------------
always@(posedge CLK_1M)     //每当 CLK 产生上升沿时
 begin
 if(CA) COUNTER<= COUNTER_BEGIN;  //如有进位信号,重装计数器的预置值
  else COUNTER<= COUNTER+1;       //否则无进位信号,则作加法计数
end
//-------------------------------------------------
always@(posedge CA)           //每当 CA 产生上升沿时
 begin
   MUSIC_OUT=~MUSIC_OUT;   //驱动蜂鸣器发声
```

```
end
//***************根据音调数据信号,得到预置值*****************
always @ (DATA_REG)              //每当音调数据信号有变化时
   begin
   case(DATA_REG)                //case 语句,产生散转分支
   8'h00:COUNTER_BEGIN=2047;  //音调数据信号为 0x00,是乐曲停止信号

   8'h11:COUNTER_BEGIN=136;   //音调数据信号为 0x11,得到低音 1 预置值
   8'h12:COUNTER_BEGIN=345;   //音调数据信号为 0x12,得到低音 2 预置值
   8'h13:COUNTER_BEGIN=530;   //音调数据信号为 0x13,得到低音 3 预置值
   8'h14:COUNTER_BEGIN=615;   //音调数据信号为 0x14,得到低音 4 预置值
   8'h15:COUNTER_BEGIN=771;   //音调数据信号为 0x15,得到低音 5 预置值
   8'h16:COUNTER_BEGIN=911;   //音调数据信号为 0x16,得到低音 6 预置值
   8'h17:COUNTER_BEGIN=1035;  //音调数据信号为 0x17,得到低音 7 预置值

   8'h21:COUNTER_BEGIN=1092;  //音调数据信号为 0x21,得到中音 1 预置值
   8'h22:COUNTER_BEGIN=1196;  //音调数据信号为 0x22,得到中音 2 预置值
   8'h23:COUNTER_BEGIN=1289;  //音调数据信号为 0x23,得到中音 3 预置值
   8'h24:COUNTER_BEGIN=1331;  //音调数据信号为 0x24,得到中音 4 预置值
   8'h25:COUNTER_BEGIN=1409;  //音调数据信号为 0x25,得到中音 5 预置值
   8'h26:COUNTER_BEGIN=1479;  //音调数据信号为 0x26,得到中音 6 预置值
   8'h27:COUNTER_BEGIN=1541;  //音调数据信号为 0x27,得到中音 7 预置值

   8'h31:COUNTER_BEGIN=1567;  //音调数据信号为 0x37,得到高音 1 预置值
   8'h32:COUNTER_BEGIN=1621;  //音调数据信号为 0x37,得到高音 2 预置值
   8'h33:COUNTER_BEGIN=1668;  //音调数据信号为 0x37,得到高音 3 预置值
   8'h34:COUNTER_BEGIN=1689;  //音调数据信号为 0x37,得到高音 4 预置值
   8'h35:COUNTER_BEGIN=1728;  //音调数据信号为 0x37,得到高音 5 预置值
   8'h36:COUNTER_BEGIN=1763;  //音调数据信号为 0x37,得到高音 6 预置值
   8'h37:COUNTER_BEGIN=1794;  //音调数据信号为 0x37,得到高音 7 预置值
   endcase                       //case 语句结束
   end
//-----------------------------------
always@(negedge ALE)             //每当 ALE 产生下降沿时
begin
ADDRESS_REG= DATA_PORT;   //将总线上的数据锁入 ADDRESS_REG 寄存器
end
//-----------------------------------
always@(negedge RD)              //当 RD 产生下降沿时
```

```
begin
  if(ADDRESS_REG==8'h55)          //如果 ADDRESS_REG 寄存器内容为 0x55
  ACK_REG=8'hf1;                  //向单片机发送应答信号
  else                            //否则
  ACK_REG=8'bzzzzzzzz;            //ACK_REG 寄存器置高阻
end
//-----------------------------------
always@(negedge WR)               //每当 WR 产生下降沿时
begin
  if(ADDRESS_REG==8'h88)          //如果 ADDRESS_REG 寄存器内容为 0x88
  DATA_REG=DATA_PORT;             //读取单片机发送来的音调数据信号
end
//-----------------------------------
assign DATA_PORT =RD? 8'bzzzzzzzz:ACK_REG; //持续赋值语句
endmodule                         //模块结束
```

引脚分配见表 14-5。器件编译通过后，可根据需要进行仿真。最后将*.jed 文件下载到 XC95108 芯片中。

表 14-5　　长时间多曲自动演奏器的引脚分配

引脚名	引脚号	输入或输出	板上丝印符号
CLK	9	Input	
RD	18	Input	
WR	17	Input	
ALE	11	Input	
MUSIC_OUT	19	Output	LED7
DATA_PORT 7	81	Inout	
DATA_PORT 6	82	Inout	
DATA_PORT 5	83	Inout	
DATA_PORT 4	84	Inout	
DATA_PORT 3	1	Inout	
DATA_PORT 2	2	Inout	
DATA_PORT 1	3	Inout	
DATA_PORT 0	4	Inout	

14.2.4　单片机 C 程序设计

在 E 盘中先建立一个文件名为“MCU_MUSIC”的文件夹，然后建立一个

“MCU_MUSIC”的新项目，输入以下的源代码并保存为“MCU_MUSIC.v”。

```
#include<REG51.H>                    //包含头文件
#include<ABSACC.H>                   //包含头文件
#define uchar unsigned char          //数据类型的宏定义
#define uint unsigned int            //数据类型的宏定义
//-------------------------------------------------------
#define RD_CPLD_ACK_REG XBYTE[0x0055]     //读 CPLD 的地址宏定义
#define WR_CPLD_DATA_REG XBYTE[0x0088]    //写 CPLD 的地址宏定义
//-------------------------------------------------------
#define T0_0 0                       //定义乐曲停止
#define T0_25 15                     //音长 0.25 拍定义,75ms
#define T0_5 25                      //音长 0.5 拍定义,125ms
#define T0_75 38                     //音长 0.75 拍定义,190ms
#define T1_0 50                      //音长 1 拍定义,250ms
#define T1_5 75                      //音长 1.5 拍定义,375ms
#define T2_0 100                     //音长 2 拍定义,500ms
#define T3_0 150                     //音长 3 拍定义,750ms
#define T4_0 200                     //音长 4 拍定义,1000ms
//-------------------------------------------------------
uchar music_val;                     //全局变量定义,音调数据
uint time;                           //全局变量定义,音长数据
//--------------“两只老虎”的编码
uchar code MUSIC1[]={
0x21,T1_0,0x22,T1_0,0x23,T1_0,0x21,T1_0,0x21,T1_0,0x22,T1_0,
0x23,T1_0,0x21,T1_0,
0x23,T1_0,0x24,T1_0,0x25,T2_0,0x23,T1_0,0x24,T1_0,0x25,T2_0,
0x25,T0_75,0x26,T0_25,
0x25,T0_75,0x24,T0_25,0x23,T1_0,0x21,T1_0,0x25,T0_75,0x26,
T0_25,0x25,T0_75,0x24,T0_25,
0x23,T1_0,0x21,T1_0,0x21,T1_0,0x15,T1_0,0x21,T2_0,0x21,T1_0,
0x15,T1_0,0x21,T2_0,
0x00,T0_0};
//---------------“新年好”的编码
uchar code MUSIC2[]={
0x21,T0_5,0x21,T0_5,0x21,T1_0,0x15,T1_0,0x23,T0_5,0x23,T0_5,
0x23,T1_0,0x21,T1_0,
0x21,T0_5,0x23,T0_5,0x25,T1_0,0x25,T1_0,0x24,T0_5,0x23,T0_5,
0x22,T2_0,
```

```
0x22,T0_5,0x23,T0_5,0x24,T1_0,0x24,T1_0,0x23,T0_5,0x22,T0_5,
0x21,T1_0,0x23,T1_0,
0x21,T0_5,0x23,T0_5,0x22,T1_0,0x15,T1_0,0x17,T1_0,0x22,T1_0,
0x21,T2_0,0x00,T0_0};
//---------------“世上只有妈妈好”的编码
uchar code MUSIC3[]={
0x26,T1_5,0x25,T0_5,0x23,T1_0,0x25,T1_0,0x31,T1_0,0x26,T0_5,
0x25,T0_5,0x26,T2_0,
0x23,T1_0,0x25,T0_5,0x26,T0_5,0x25,T1_0,0x23,T1_0,0x21,T0_5,
0x16,T0_5,0x25,T0_5,
0x23,T0_5,0x22,T2_0,0x22,T1_5,0x23,T0_5,0x25,T1_0,0x25,T0_5,
0x26,T0_5,0x23,T1_0,
0x22,T1_0,0x21,T2_0,0x25,T1_5,0x23,T0_5,0x22,T0_5,0x21,T0_5,
0x16,T0_5,0x21,T0_5,
0x15,T2_0,0x00,T0_0};
//------------------“橄榄树”的编码
uchar code MUSIC4[]={
0x23,T0_5,0x21,T0_25,0x22,T0_25,0x23,T1_0,0x23,T0_5,0x21,T0_
25,0x22,T0_25,0x23,T0_5,
0x24,T0_25,0x23,T0_25,0x22,T0_5,0x17,T0_25,0x21,T0_25,0x22,
T1_0,0x22,T0_5,0x17,T0_25,
0x21,T0_25,0x22,T0_5,0x17,T0_25,0x21,T0_25,0x21,T0_5,0x16,
T1_0,0x21,T0_5,0x17,T0_5,
0x15,T1_0,0x17,T0_5,0x16,T2_0,0x16,T1_0,
0x00,T0_5,0x26,T0_25,0x26,T0_25,0x23,T0_5,0x25,T0_5,0x24,T0_5,
0x23,T0_25,0x22,T0_25,
0x23,T1_0,0x23,T1_0,0x00,T0_5,0x26,T0_25,0x26,T0_25,0x23,T0_5,
0x24,T0_5,0x24,T0_5,
0x23,T0_25,0x22,T0_25,0x21,T2_0,0x21,T1_0,0x00,T0_5,0x26,T0_5,
0x25,T0_5,0x26,T0_5,
0x23,T0_5,0x21,T0_25,0x22,T0_25,0x23,T2_0,0x00,T0_5,0x22,T0_5,
0x22,T0_5,0x22,T0_5,
0x21,T2_0,0x21,T2_0,0x00,T1_0,0x17,T0_5,0x22,T0_25,0x21,T0_25,
0x16,T2_0,0x16,T1_0,
                                    0x00,T0_5,0x27,T0_5,0x26,T1_5,
0x27,T0_25,0x23,T2_0,0x00,T0_5,0x27,T1_0,0x26,T0_5,0x23,T0_5,
0x25,T0_25,0x25,T0_25,
0x24,T0_5,0x23,T0_25,0x22,T0_25,0x23,T1_0,0x22,T0_5,0x21,T0_5,
0x22,T2_0,0x00,T0_5,
```

```
0x23,T1_0,0x25,T0_5,0x26,T0_5,0x23,T0_5,0x24,T0_5,0x23,T0_25,
0x22,T0_25,0x23,T2_0,
0x00,T0_5,0x21,T1_0,0x22,T0_5,0x23,T0_5,0x25,T1_0,0x23,T0_5,
0x27,T1_5,0x31,T0_25,
0x27,T0_25,0x26,T2_0,0x26,T2_0,0x00,T1_0,0x00,T0_5,0x25,T0_5,
0x25,T1_5,0x25,T0_25,
0x26,T1_0,0x23,T2_0,0x00,T0_5,0x25,T0_5,0x25,T1_5,0x25,T0_25,
0x23,T1_0,0x26,T1_0,
0x27,T1_5,0x31,T0_25,0x27,T0_25,0x26,T2_0,0x26,T2_0,0x25,T1_
0,0x23,T1_0,0x00,T1_0,
0x26,T1_0,0x27,T1_0,0x26,T1_0,0x23,T1_0,0x22,T1_0,0x21,T2_0,
0x21,T2_0,0x00,T1_0,
0x17,T0_5,0x22,T0_25,0x21,T0_25,0x16,T2_0,0x16,T1_0,0x00,T0_
5,0x26,T0_25,0x26,T0_25,
0x23,T0_5,0x25,T0_5,0x24,T0_5,0x23,T0_25,0x22,T0_25,0x23,T2_
0,0x23,T1_0,0x00,T0_5,
0x26,T0_25,0x26,T0_25,0x23,T0_5,0x25,T0_5,0x24,T0_5,0x23,T0_
25,0x22,T0_25,0x21,T2_0,
0x21,T1_0,0x00,T0_5,0x26,T0_5,0x25,T0_5,0x26,T0_5,0x23,T0_5,
0x21,T0_25,0x22,T0_25,
0x23,T2_0,0x00,T0_5,0x22,T0_5,0x22,T0_5,0x22,T0_5,0x21,T2_0,
0x21,T2_0,0x00,T1_0,
0x17,T0_5,0x22,T0_25,0x21,T0_25,0x16,T2_0,0x16,T2_0,0x23,T0_
5,0x21,T0_25,0x22,T0_25,
0x23,T1_0,0x23,T0_5,0x21,T0_25,0x22,T0_25,0x23,T0_5,0x24,T0_
25,0x23,T0_25,0x22,T0_5,
0x17,T0_25,0x21,T0_25,0x22,T1_0,0x22,T0_5,0x17,T0_25,0x21,
T0_25,0x22,T0_5,0x17,T0_25,
0x21,T0_25,0x21,T0_5,0x16,T1_0,0x21,T0_5,0x17,T0_5,0x15,T1_0,
0x17,T0_5,0x16,T2_0,
0x16,T2_0,0x00,T0_0};
//*******************************************
void delay(uint k)              //延时 k*1ms 的子程序
{
uint i,j;
for(i=0;i<k;i++){
for(j=0;j<120;j++)
{;}}
}
```

```
/*********************************************/
void init(void)                    //单片机初始化
{
TMOD=0x01;                         //定时器 0 方式 1
ET0=1;                             //开 T0 中断
TH0=0xee;                          //定时 5ms 初值
TL0=0x00;
}
/*************************************/
void main(void)                    //定义主函数
{
uint i,j,val;                      //局部变量定义
  init();                          //调用初始化子函数
  while(1)                         //无限循环
  {
    val=0x00;val=RD_CPLD_ACK_REG;      //呼叫 CPLD
    if(val==0xf1)                  //如果 CPLD 有应答
    {
       i=0;j=1;
       for(;;)                     //播放乐曲 1
       {
         music_val=MUSIC1[i];      //取出一个音符的音调数据
         time=MUSIC1[j];           //取出一个音符的音长数据
         if((music_val==0x00)&&(time==0))break;
                                   //如遇到乐曲结束符则退出播放
         WR_CPLD_DATA_REG =music_val; //音调数据传送到 CPLD 进行乐
                                        //曲播放
         TR0=1;EA=1;               //启动定时器,开中断,控制音长
         while(time!=0);           //等待当前音符的结束
         TR0=0;EA=0;               //当前音符结束,关定时器,关中断
         i=i+2; j=j+2;             //指向下一个音符
       }
       WR_CPLD_DATA_REG =0x00;delay(3000); //一曲结束,等待 3s
    }
    //*********************************
    val=0x00;val=RD_CPLD_ACK_REG; //呼叫 CPLD
    if(val==0xf1)                      //如果 CPLD 有应答
    {
       i=0;j=1;
```

```
    for(;;)                          //播放乐曲 2
    {
      music_val=MUSIC2[i];           //取出一个音符的音调数据
      time=MUSIC2[j];time=time*2;    //取出一个音符的音长数据,
                                     //根据情况修正音长长度
      if((music_val==0x00)&&(time==0))break; //如遇到乐曲结束
                                             //符则退出播放
      WR_CPLD_DATA_REG=music_val;    //音调数据传送到 CPLD 进行
                                     //乐曲播放
      TR0=1;EA=1;                    //启动定时器,开中断,控制音长
      while(time!=0);                //等待当前音符的结束
      TR0=0;EA=0;                    //当前音符结束,关定时器,关中断
      i=i+2; j=j+2;                  //指向下一个音符
    }
    WR_CPLD_DATA_REG =0x00;delay(3000);  //指向下一个音符
}
//*********************************
val=0x00;val=RD_CPLD_ACK_REG;        //呼叫 CPLD
if(val==0xf1)                        //如果 CPLD 有应答
{
    i=0;j=1;
    for(;;)                          //播放乐曲 3
    {
      music_val=MUSIC3[i];
      time=MUSIC3[j];time=time*2;
      if((music_val==0x00)&&(time==0))break;
      WR_CPLD_DATA_REG =music_val;
      TR0=1;EA=1;
      while(time!=0);
      TR0=0;EA=0;
      i=i+2; j=j+2;
    }
    WR_CPLD_DATA_REG =0x00;delay(3000);
}
//*********************************
val=0x00;val=RD_CPLD_ACK_REG;        //呼叫 CPLD
if(val==0xf1)                        //如果 CPLD 有应答
{
    i=0;j=1;
```

```
        for(;;)                                    //播放乐曲 4
        {
          music_val=MUSIC4[i];
          time=MUSIC4[j];time=time*3;
          if((music_val==0x00)&&(time==0))break;
          WR_CPLD_DATA_REG =music_val;
          TR0=1;EA=1;
          while(time!=0);
          TR0=0;EA=0;
          i=i+2; j=j+2;
        }
        WR_CPLD_DATA_REG =0x00;delay(3000);
    }
    //*********************************
 }
}
/*************T0  5ms 定时中断子函数************/
void time0(void) interrupt 1
{
 TH0=0xee;
 TL0=0x00;
 time--;
}
```

在 MCU&CPLD DEMO 试验板上电以后，我们听到电路自动循环播放的“两只老虎”、“新年好”、“世上只有妈妈好”、“橄榄树”等乐曲，每曲之间停顿 3s。读者朋友也可将自己喜爱的歌曲、乐曲编入单片机，由于 AT89S51/52 单片机的 flash 区达 4KB/8KB，因此可以存入几十首乐曲，这样这台长时间多曲自动演奏器的内容就更丰富多彩了。

参 考 文 献

[1] 王金明编著. 数字系统设计与 Verilog HDL（第 2 版）. 北京：电子工业出版社，2005.

[2] 常晓明编著. Verilog-HDL 实践与应用系统设计. 北京：北京航空航天大学出版社，2003.

[3] J. Bhasker 著. 孙海平等译. Verilog HDL 综合实用教程. 北京：清华大学出版社，2004.

[4] 周立功，夏宇闻等编著. 单片机与 CPLD 综合应用技术. 北京：北京航空航天大学出版社，2003.

[5] 周兴华编著. 手把手教你学单片机 C 程序设计. 北京：北京航空航天大学出版社，2007.