



FPGA 技术入门及项目 化开发实战 教材

[李德明 编著]



2020-9-3

老伙计
广西桂林

目 录

前言.....	5
第一章 FPGA 概述	6
1.1 可编程逻辑器件的发展史.....	6
1.2 FPGA 简介	7
1.3 FPGA 特点及应用领域	9
1.4 FPGA 发展前景.....	12
第二章 开发软件的安装与使用.....	14
2.1 Quartus II 软件安装	14
2.2 Quartus II 软件安装后处理	17
2.3 USB-Blaster 驱动安装	18
第三章 FPGA 原理和结构	21
3.1 FPGA 技术原理	21
3.2FPGA 芯片结构	22
3.3 软核、硬核以及固核的概念.....	26
第四章 FPGA 的开发流程	28
4.1 设计的输入.....	28
4.2. 设计的编译.....	34
4.3. 仿真验证.....	36
4.3.1 Quartus II 仿真验证	36
4.3.2 modelsim 仿真验证.....	39
4.4. 引脚锁定.....	48
4.5. 下载调试验证.....	48
第五章 FPGA 开发板	53
5.1 FPGA 开发板简介	53
5.2 开发板硬件详细配置.....	55
5.3 开发板硬件原理图.....	57
5.3.1 核心板硬件电路.....	57
5.3.2 底板硬件电路.....	60
第六章 Verilog HDL 基础知识.....	70
6.1 Verilog HDL 设计模块的基本结构.....	71
6.1.1 模块端口定义.....	71
6.1.2 模块内容.....	72

6.2 Verilog HDL 的词法.....	76
6.2.1 空白符和注释.....	76
6.2.2 常数.....	76
6.2.3 字符串.....	77
6.2.4 关键字.....	77
6.2.5 标识符.....	78
6.2.6 操作符.....	79
6.2.7 Verilog HDL 数据对象.....	84
6.3 Verilog HDL 的语句.....	88
6.3.1 赋值语句.....	88
6.3.2 条件语句.....	92
6.3.3 循环语句.....	99
6.3.4 结构声明语句.....	104
6.3.5 编译预处理语句.....	111
6.3.6 语句的顺序执行与并行执行.....	115
6.4 不同抽象级别的 Verilog HDL 模型.....	119
6.4.1 Verilog HDL 门级描述.....	120
6.4.2 Verilog HDL 的行为级描述.....	121
6.4.3 用结构描述实现电路系统设计.....	123
6.4.4 Verilog HDL 模型设计总结.....	127
6.5 Verilog HDL 设计流程.....	128
6.5.1 编辑 Verilog HDL 源程序.....	128
6.5.2 设计 BCD 数加法器电路顶层文件	130
6.5.3 编译顶层设计文件.....	131
6.5.4 仿真顶层设计文件.....	131
6.6 Verilog HDL 仿真.....	131
6.6.1 Verilog HDL 仿真支持语句.....	132
6.6.2 Verilog HDL 测试平台软件的设计	135
6.7 代码编写“规范”.....	139
6.8 本章小结.....	142
第七章 数字基础实验	143
7.1 分频器的设计及 quartus II 仿真.....	143
7.2 计数器的设计	151
7.3 D 触发器.....	153

7.4 三态门	155
7.5 8-3 编码器	156
7.6 8-3 优先编码器	158
7.7 3-8 译码器	161
7.8 移位寄存器	163
7.9 多路选择器	165
7.10 串行加法器	166
7.11 简单运算单元 ALU	167
第八章 开发板基础实验	170
8.1 LED 流水灯	171
8.2 按键消抖	175
8.3 PWM 控制灯的亮暗	180
8.4 数码管的动态显示	184
8.5 秒表数码管显示	188
8.6 时钟数码管显示	193
8.7 频率计的设计	198
8.8 蜂鸣器音乐播放器	205
8.9 按键计数器	209
8.10 串口通信	212
8.11 液晶 LCD1602 显示	219
8.12 DDS 与嵌入式逻辑分析仪的调用	229
8.13 步进电机控制实验	243
8.14 矩阵键盘控制实验	250
8.15 旋转编码开关实验	262
第九章 宏功能模块的使用	267
9.1 PLL 的使用	267
9.2 ROM 的使用	274
9.3 FIFO 的使用	274
9.4 RAM 的使用	280
9.5 乘法器的使用	285
第十二章 综合实验	291
12.1 基于 DDS 的任意波形发生器	291
12.2 基于 FPGA 的出租车计费器设计	293
12.3 基于 FPGA 的交通灯设计	296

第十三章 学习 FPGA 技术总结	303

前言

FPGA 是何物？曾经的我们对 FPGA 的认识一片空白，现在 FPGA 对我们来说也是一片空白，它可以说是一张白纸，任凭你在上面挥毫泼墨，只要你的想象够丰富，基础够扎实，相信你定会绘出属于自己的一片蓝图。为什么定位为《FPGA 技术入门与项目化开发实战》？回顾我们的学习之路，缺乏一些连贯的学习资料和系统的学习方法，为了让更多的人走上这一弯路，我们积累总结了很多例程和资料，通过一个个简单的例子以点带面，让你逐步掌握 FPGA 的设计，并通过综合实战将理论与 FPGA 的硬件实现相结合。本书不仅仅是实验手册，更是理论与实践相结合的 FPGA 设计手册，图文并茂，一步步开启你的 FPGA 设计之路。

本书主要讲解FPGA的开发流程和程序设计，以一款资源丰富的FPGA开发板为例，介绍学习FPGA内部结构和设计流程，讲解Verilog HDL硬件描述语言的基础知识，以及FPGA开发板的硬件配置，重点是第七章到第十二章的实例程序，由简单到复杂，最后是FPGA的设计心得。

本书适合电子、通信、自动化等相关专业的本科生研究生以及从事FPGA开发/IC设计/PCB 等相关职业的读者阅读参考。

信客电子--老伙计 FPGA 设计团队

第一章 FPGA 概述

FPGA 与 EDA 技术是目前相当热门的技术，翻遍各大招聘网站电子类招聘信息，类似“精通 FPGA 技术，熟悉 Verilog VHDL 语言”等字眼已经为应聘者所熟悉；甚至，有的时候，“熟悉 FPGA”就意味着高薪！实际上，FPGA 技术已经成为目前电子行业应用最为广泛的技术之一，未来的就业和发展前景相当好！

FPGA 是现场可编程门阵列的简称，FPGA 的应用领域最初为通信领域，但目前，随着信息产业和微电子技术的发展，可编程逻辑嵌入式系统设计技术已经成为信息产业最热门的技术之一，应用范围遍及航空航天、医疗、通讯、网络通讯、安防、广播、汽车电子、工业、消费类市场、测量测试等多个热门领域。并随着工艺的进步和技术的发展，向更多、更广泛的应用领域扩展。越来越多的设计也开始以 ASIC 转向 FPGA，FPGA 正以各种电子产品的形式进入了我们日常生活的各个角落。

FPGA 一般应用在高性能处理、实时要求高的领域，比如高速接口、报文转发、图像处理、视频传输等，还可以应用在芯片前期验证。

1.1 可编程逻辑器件的发展史

起源：可编程逻辑器件(Programmable Logic Device, PLD)起源于 20 世纪 70 年代，是在专用集成电路(ASIC)的基础上发展起来的以后总新型逻辑器件。

主要特点：完全由用户通过软件进行配置和编程，从而完成某种特定的功能，并且可以反复擦写。

常见 PLD 产品：可编程只读存储器(PROM)、现场可编程逻辑阵列(FPLA)、可编程阵列逻辑(PAL)、复杂可编程逻辑器件(CPLD)、和现场可编程门阵列(FPGA)等类型，它们的内部结构和表现方法各不相同。

可编程逻辑器件的发展史(4 个阶段)：

第一阶段：(20 世纪 70 年代初到 70 年代中)只有简单的 PROM、紫外线可擦除只读存储器(EPROM)和电可擦除只读存储器(EEPROM)3 种。只能完成简单的数字逻辑功能。

第二阶段：(20 世纪 70 年代中到 80 年代中)结构上稍微复杂的可编程阵列逻辑(PAL)和通用阵列逻辑(GAL)器件，正式被称为 PLD,能够完成各种逻辑运算功能。

第三阶段：(20 世纪 80 年代中到 90 年代末)Xilinx 和 Altera 公司分别推出了与标准门阵列雷似的 FPGA 以及类似于 PAL 结构的扩展性 CPLD。提高

了逻辑运算速度，逻辑单元灵活、集成度高、适用范围宽、编程灵活。

第四阶段：(20 世纪 90 年代末至今)出现了可编程片上系统(SOPC)和片上系统(SOC)技术。涵盖了实时化数字信号处理、高速数据收发器、复杂计算以及嵌入式系统设计技术的全部内容。

1.2 FPGA 简介

FPGA 是 Field Programmable Gate Array 的缩写，即现场可编程门阵列，它是在 PAL、GAL、CPLD 等可编程器件的基础上进一步发展的产物。它是作为专用集成电路(ASIC)领域中的一种半定制电路而出现的，既解决了定制电路的不足，又克服了原有可编程器件门电路数有限的缺点。

FPGA 是 Ross Freema 于 1985 年发明的,当时第一个 FPGA 采用 $2\mu\text{m}$ 工艺,包含 64 个逻辑模块和 85000 个晶体管,门数量不超过 1000 个,当时他所创造的 FPGA 被认为是一项不切实际的技术,他的同事 Bill Carter 曾说:“这种理念需要很多晶体管,但那时晶体管是非常珍贵的东西。”所以人们认为 Ross 的想法过于脱离现实。但是 Ross 预计:根据摩尔定律(每 18 个月晶体管密度翻一翻),晶体管肯定会越来越便宜,因此它必将成为未来不可或缺的技术。在短短的几年时间内,正如 Ross 所预言的,出现了数十亿美元的现场可编程门阵列(FPGA)市场。但可惜的是,他已经无法享受这一派欣欣向荣的景象,Ross Freeman 在 1989 年已经与世长辞了,但是它的发明却持续不断地促进电子行业的进步与发展。

FPGA 企业状况：目前全球主要的 FPGA 企业数量较少,仅有 Xilinx(赛灵思)、Altera(阿尔特拉)、Lattice(莱迪思)、Microsemi(美高森美)等几家,其中,Xilinx 与 Altera 占有近 90% 的市场份额,专利达到 6000 余项之多,Xilinx 始终保持着全球 FPGA 的龙头地位。

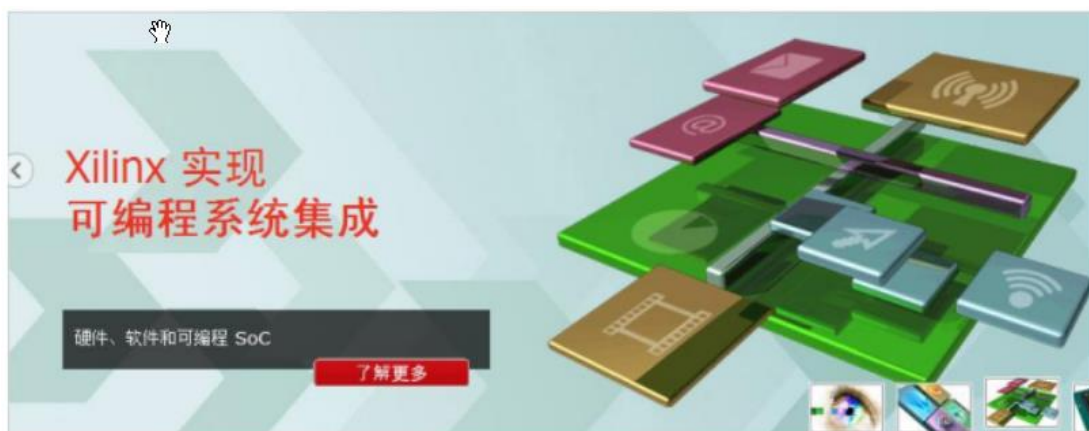


图 1.1 Xilinx 公司宣传图

1、Xilinx 作为 FPGA 的发明者，XilinxFPGA 主要分为两大类，一种侧重低成本应用，容量中等，性能可以满足一般的逻辑设计要求，如 Spartan 系列；还有一种侧重于高性能应用，容量大，性能能满足各类高端应用，如 Virtex 系列，用户可以根据自己实际应用要求进行选择。在性能可以满足的情况下，优先选择低成本器件。Xilinx 官网：<http://china.xilinx.com/>。



图 1.2 Altera 公司宣传图

2、Altera 的主流 FPGA 分为两大类，一种侧重低成本应用，容量中等，性能可以满足一般的逻辑设计要求，如 Cyclone，CycloneII，CycloneIII，最新 Cyclone 系列已经到了 Cyclone V 五代芯片型号了；还有一种侧重于高性能应用，容量大，性能能满足各类高端应用，如 Stratix，StratixII 等，用户可以根据自己实际应用要求进行选择。在性能可以满足的情况下，优先选择低成本器件。Altera 官网：<http://www.altera.com.cn>。

Cyclone (飓风): Altera 中等规模 FPGA，2003 年推出，0.13um 工艺，1.5v 内核供电，与 Stratix 结构类似，是一种低成本 FPGA 系列，其配置芯片也改用全新的产品。

简评：Altera 最成功的器件之一，性价比不错，是一种适合中低端应用的通用 FPGA，推荐使用。

CycloneII: Cyclone 的二代产品，2005 年开始推出，90nm 工艺，1.2v 内核供电，属于低成本 FPGA，性能和 Cyclone 相当，提供了硬件乘法器单元

简评：推出的第二代低成本 FPGA，目前市场零售容易买到，从 2005 年年底开始，逐步取代 Cyclone 器件，成为 Altera 在中低 FPGA 市场中的主力产品。

Stratix: altera 大规模高端 FPGA，2002 年中期推出，0.13um 工艺，1.5v 内核供电。集成硬件乘加器，芯片内部结构比 Altera 以前的产品有很大变化。

简评：Stratix 芯片在 2002 年的推出，改变了 Altera 在 FPGA 市场上的被动局面。该芯片适合高端应用。随着 2005 年新一代 StratixII 器件的推出，被 StratixII 逐渐取代。

StratixII: Stratix 的下二代产品，2004 年中期推出，90nm 工艺，1.2v 内核

供电，大容量高性能 FPGA。

简评：性能超越 Stratix，是 Altera 在高端 FPGA 市场中的主力产品，StratixV 为 altera 目前的高端产品，采用 28-nm 工艺，提供了 28G 的收发器件，适合高端的 FPGA 产品开发。

1.3 FPGA 特点及应用领域

1. FPGA 的基本特点

- 1) 采用 FPGA 设计 ASIC 电路，用户不需要投片生产，就能得到合用的芯片。
- 2) FPGA 可做其它全定制或半定制 ASIC 电路的中试样片。
- 3) FPGA 内部有丰富的基本逻辑门、触发器和 I/O 引脚。
- 4) FPGA 是 ASIC 电路中设计周期最短、开发费用最低、风险最小的器件之一。
- 5) FPGA 采用高速 CHMOS 工艺，功耗低，可以与 CMOS、TTL 电平兼容。可以说，FPGA 芯片是小批量系统提高系统集成度、可靠性的最佳选择之一。

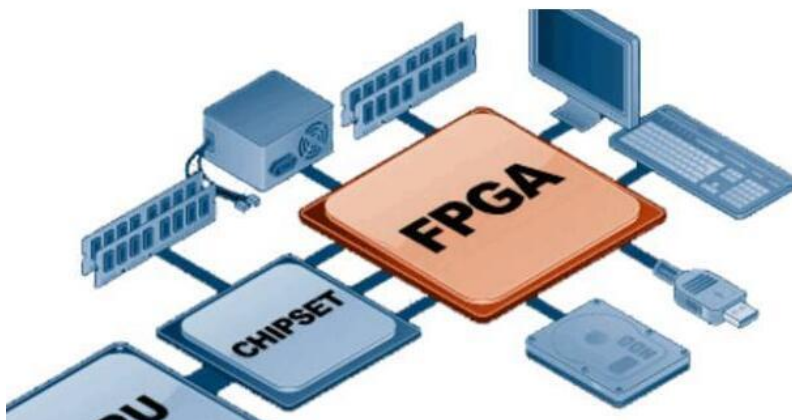


图 1.3 FPGA 应用结构图

2.FPGA 的优势

- 1) 通信高速接口设计。FPGA 可以用来做高速信号处理，一般如果 AD 采样率高，数据速率高，这时就需要 FPGA 对数据进行处理，比如对数据进行抽取滤波，降低数据速率，使信号容易处理，传输，存储。
- 2) 数字信号处理。包括图像处理，雷达信号处理，医学信号处理等。优势是实时性好，用面积换速度，比 CPU 快的多。
- 3) 更大的并行度。这个主要是通过并发和流水两种技术实现。并发是指重

复分配计算资源，使得多个模块之间可以同时独立进行计算。

3.FPGA 的应用领域

一、数据采集和接口逻辑领域

1、FPGA 在数据采集领域的应用

由于自然界的信号大部分是模拟信号，因此一般的信号处理系统中都要包括数据的采集功能。通常的实现方法是利用 A/D 转换器将模拟信号转换为数字信号后，送给处理器，比如利用单片机（MCU）或者数字信号处理器（DSP）进行运算和处理。

对于低速的 A/D 和 D/A 转换器，可以采用标准的 SPI 接口来与 MCU 或者 DSP 通信。但是，高速的 A/D 和 D/A 转换芯片，比如视频 Decoder 或者 Encoder，不能与通用的 MCU 或者 DSP 直接接口。在这种场合下，FPGA 可以完成数据采集的粘合逻辑功能。

2、FPGA 在逻辑接口领域的应用

在实际的产品设计中，很多情况下需要与 PC 机进行数据通信。比如，将采集到的数据送给 PC 机处理，或者将处理后的结果传给 PC 机进行显示等。PC 机与外部系统通信的接口比较丰富，如 ISA、PCI、PCI Express、PS/2、USB 等。

传统的设计中往往需要专用的接口芯片，比如 PCI 接口芯片。如果需要的接口比较多，就需要较多的外围芯片，体积、功耗都比较大。采用 FPGA 的方案后，接口逻辑都可以在 FPGA 内部来实现了，大大简化了外围电路的设计。

在现代电子产品设计中，存储器得到了广泛的应用，例如 SDRAM、SRAM、Flash 等。这些存储器都有各自的特点和用途，合理地选择存储器类型可以实现产品的最佳性价比。由于 FPGA 的功能可以完全自己设计，因此可以实现各种存储接口的控制器。

3、FPGA 在电平接口领域的应用

除了 TTL、COMS 接口电平之外，LVDS、HSTL、GTL/GTL+、SSTL 等新的电平标准逐渐被很多电子产品采用。比如，液晶屏驱动接口一般都是 LVDS 接口，数字 I/O 一般是 LVTTTL 电平，DDR SDRAM 电平一般是 HSTL 的。

在这样的混合电平环境里面，如果用传统的电平转换器件实现接口会导致电路复杂性提高。利用 FPGA 支持多电平共存的特性，可以大大简化设计方案，降低设计风险。



图 1.4 FPGA 应用设计实物图

二、高性能数字信号处理领域

无线通信、软件无线电、高清影像编辑和处理等领域，对信号处理所需要的计算量提出了极高的要求。传统的解决方案一般是采用多片 DSP 并联构成多处理器系统来满足需求。

但是多处理器系统带来的主要问题是设计复杂度和系统功耗都大幅度提升，系统稳定性受到影响。FPGA 支持并行计算，而且密度和性能都在不断提高，已经可以在很多领域替代传统的多 DSP 解决方案。

例如，实现高清视频编码算法 H.264。采用 TI 公司 1GHz 主频的 DSP 芯片需要 4 颗芯片，而采用 Altera 的 StratixII EP2S130 芯片只需要一颗就可以完成相同的任务。FPGA 的实现流程和 ASIC 芯片的前端设计相似，有利于导入芯片的后端设计。

三、其他应用领域

除了上面一些应用领域外，FPGA 在其他领域同样具有广泛的应用。

- (1) 汽车电子领域，如网关控制器/车用 PC 机、远程信息处理系统。
- (2) 军事领域，如安全通信、雷达和声纳、电子战。
- (3) 测试和测量领域，如通信测试和监测、半导体自动测试设备、通用仪表。
- (4) 消费产品领域，如显示器、投影仪、数字电视和机顶盒、家庭网络。
- (5) 医疗领域，如软件无线电、电疗、生命科学。

1.4 FPGA 发展前景

据市场调研近年来全球 FPGA 市场规模基本在 50-60 亿美元左右,未来还将有不断增长的趋势。FPGA 及 PLD 产业发展的最大机遇是替代 ASIC 和专用标准产品(ASSP),由 ASIC 和 ASSP 构成的数字逻辑市场规模大约为 350 亿美元。由于用户可以迅速地对 PLD 进行编程,按照需求实现特殊功能,与 ASIC 和 ASSP 相比,PLD 在灵活性、开发成本、产品快速上市方面更具优势,所以 FPGA 将会是一个非常具有前景的行业。



图 1.5 2013-2017 年全球 FPGA 市场规模

由于 FPGA 结构的特殊性,可以重复编程,开发周期较短,越来越受到人们的青睐,它的特点也更接近 ASIC,ASIC 比 FPGA 最大的优势是低成本,但是 FPGA 的价格现在也越来越低。根据当前发展的趋势,未来的 FPGA 势必将会取代大部分 ASIC 的市场,虽然根据摩尔定律(Moore's Law):每 18 至 24 个月能在相同的单位面积内多集成一倍的晶体管数目,也就意味着每 18 至 24 个月后芯片成本将减半,但这只是指裸晶(Die)的成本,并不表示整个芯片的成本减半,这是由于晶圆制造前端的掩膜(Mask)成本、晶圆制造后端的封装(也称为:构装、包装)成本、人力成本等都不会随摩尔定律而变化,反而芯片的成本有上升的趋势,所以过去许多中、小用量的芯片无法用先进的工艺来生产,对此不是持续使用旧工艺来制造,或是必须改用 FPGA 芯片来生产。

未来的趋势告诉我们,FPGA 将成为 21 世纪最重要的高科技产业之一,特别是国内的 FPGA 市场,更是一个“未完全开垦的处女地”,抓住现在的机遇也就意味着为我们的将来提供更强大的竞争力。我国的 FPGA 市场国产化率非常

低，国产应用率不足 30%，还有很大提升空间，商用市场国产化率更低。从信息、产业和国防安全等方面考虑，中国不仅需要自主 FPGA，而且还需要将其快速国产化。在 AI（人工智能）、IoT（物联网）、5G 快速发展和即将商用的情况下，预计将带来庞大的 FPGA 增量市场，而这也是国内厂商快速切入的时机。国内 FPGA 产品采用 40 纳米制程工艺，拥有超过 2500 万门，Serdes 速率为 6.25Gbps，目前已经量产，并在通信、安全等中高密度市场逐步打开局面。大数据时代，云计算的市场逐步扩大。预测到 2018 年云计算市场规模将达到 1275 亿美金，年复合增长率将达到 22.8%。云计算市场的快速增加必然加大对数据中心服务器的需求。在数据中心成本中，77% 的花费在硬件配置，23% 的费用在软件方面。这也是国内厂商的发展机会。

作为一种可编程逻辑器件，FPGA 在短短二十多年中从电子设计的外围器件逐渐演变为数字系统的核心。伴随半导体工艺技术的进步，FPGA 器件的设计技术取得了飞跃发展及突破。通过 FPGA 器件的发展历程来看，今后仍将朝下以下几个方向发展：高密度、高速度、宽频带、高保密；低电压、低功耗、低成本、低价格；IP 软/硬核复用、系统集成；动态可重构以及单片集群；紧密结合应用需求，多元化发展。此外，集成了 FPGA 架构、硬核 CPU 子系统（ARM/MIPS/MCU）及其他硬核 IP 的芯片已经发展到了一个“关键点”，它将在今后数十年中得到广泛应用，为系统设计人员提供更多的选择。

第二章 开发软件的安装与使用

不同的 FPGA 厂家都有自己独立的开发工具，目前广泛使用的包括 XILINX 公司的 Vivado 软件和 Altera 公司的 Quartus II 软件。Quartus II 软件是 Altera 公司推出的新一代、功能强大的 EDA 工具，至今已发布了几十个版本，为了适应新器件的推出开发，每年都会更新一个软件版本。QuartusII 软件提供了 EDA 设计的综合开发环境，是 EDA 设计的基础。QuartusII 集成环境支持 EDA 设计的设计输入、编译、综合、布局、布线、时序分析、仿真、编程下载等设计过程。下面主要介绍软件的安装和破解过程。

初学者学习 Altea（Altea 已被 Intel 收购）的 FPGA 会有一定优势，因为资料比较多，开发板价钱也便宜，容易上手。而 Xilinx 是世界上的第一块 FPGA 由的生产者，Xilinx 的底蕴很强，很多大公司也在使用它的产品。Xilinx 的开发板主要是太贵了，资料也不像 Altea 那么多。

2.1 Quartus II 软件安装

安装流程：

第一步：解压安装文件

执行 90_Quartus_windows.exe 文件，该过程为解压安装文件，因为安装文件 2G 多，所以要等待几分钟。解压完会弹出下面的窗口：

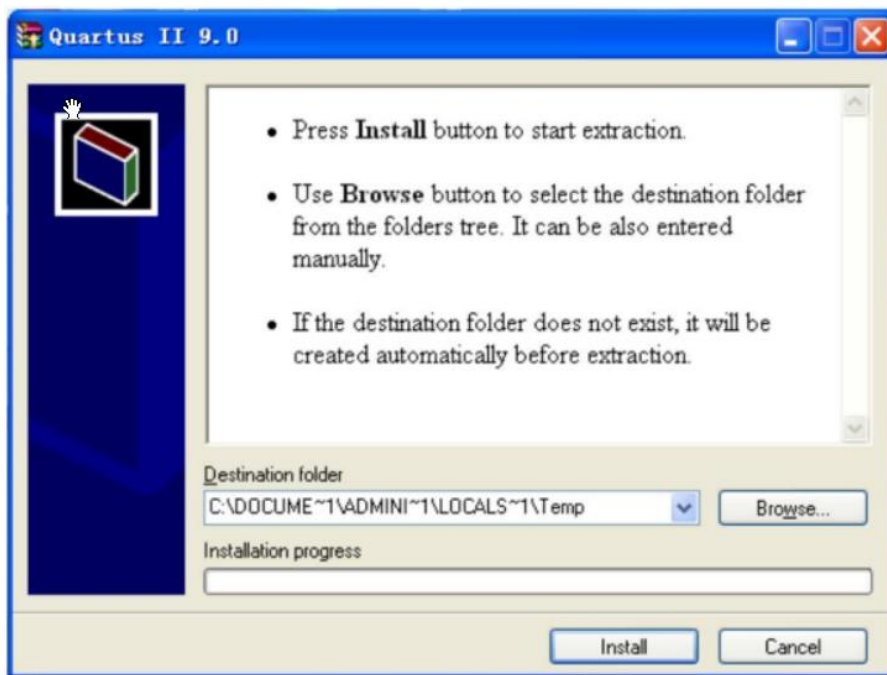


图 2-1 解压路径选择

C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp 这个路径仅仅是一个临时的解压路径，您可以选择的临时其它目录，待安装完毕后方便删除掉此临时目录，接着点击 **Install** 进入第二步。

第二步：安装程序正在解压文件到临时目录中。解压完成后，出现开始安装向导窗口。

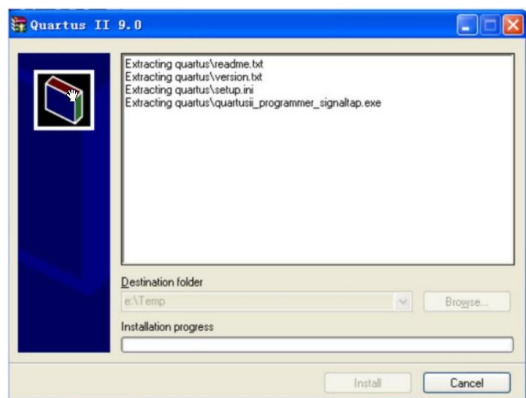


图 2-2 解压文件夹

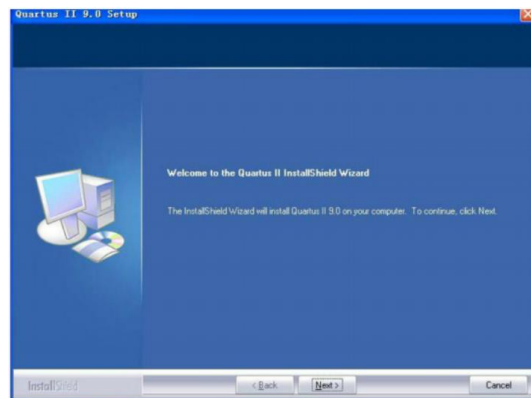


图 2-3 QuartusII 9.0 安装界面

第三步：出现 QuartusII 9.0 安装界面，点击 **next**。

第四步：出现授权窗口点击 **I accept the term of the license agreement.**后，再点击下一步 **next**。

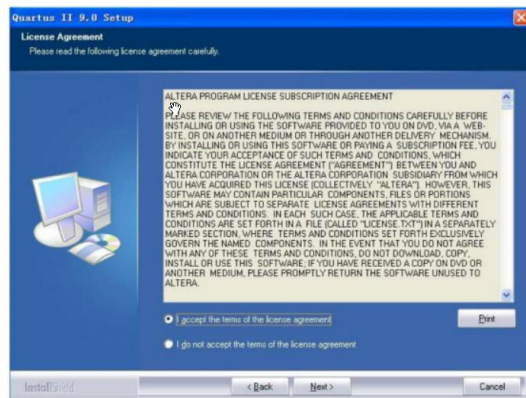


图 2-4 授权窗口

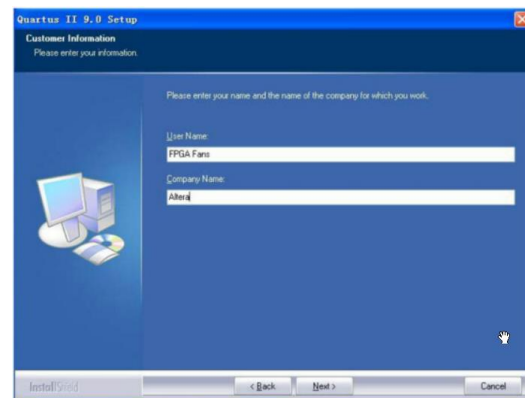


图 2-5 输入用户名和公司名

第五步：接下来输入用户名和公司名，如用户名为 **FPGA Fans**，公司名为 **Altera**，点击下一步 **next**。

第六步：选择安装目录，可以自由选择安装目录（建议目录名为英文）。

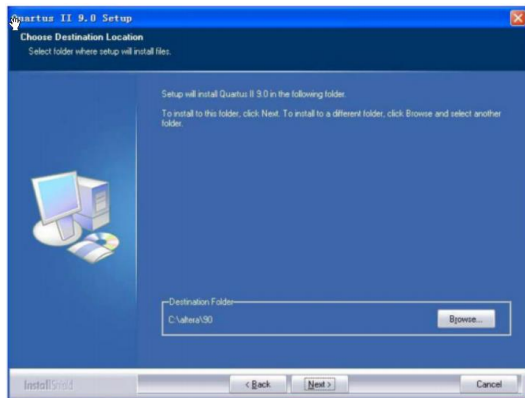


图 2-6 选择安装目录

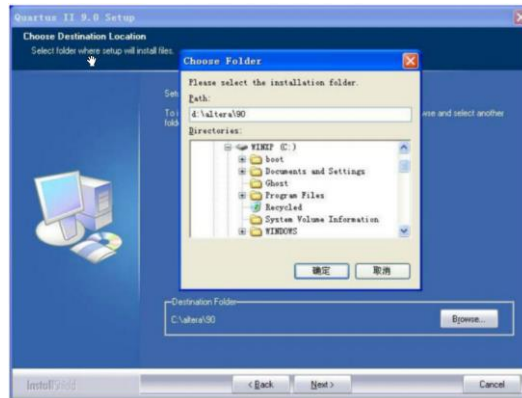


图 2-7 自行选择安装路径

第七步：您可以点击 **Browse**，按钮来自行选择安装路径。选择好路径后，点击确定，再点击 **next** 进入下一步的安装。

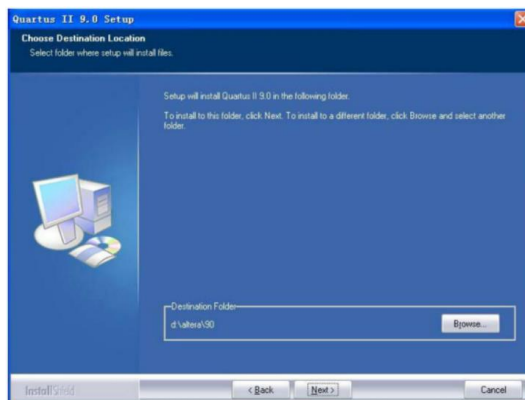


图 2-8 选择好路径

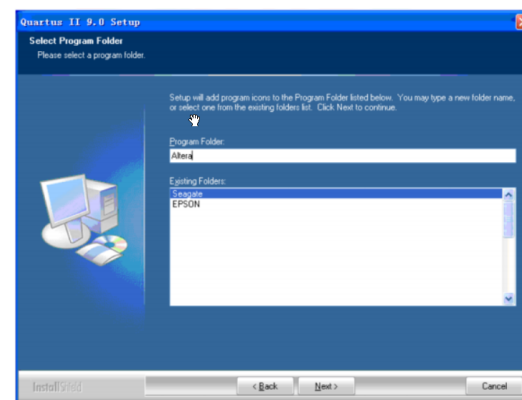


图 2-9 软件名称

第九步：默认开始菜单的软件名称，默认为 **Altera**，点击 **next** 下一步按钮继续安装。

第十步：安装类型的选择，我们推荐采用完全安装，点击 **next** 进入下一步。确认安装目录，点击 **next** 进入下一步。

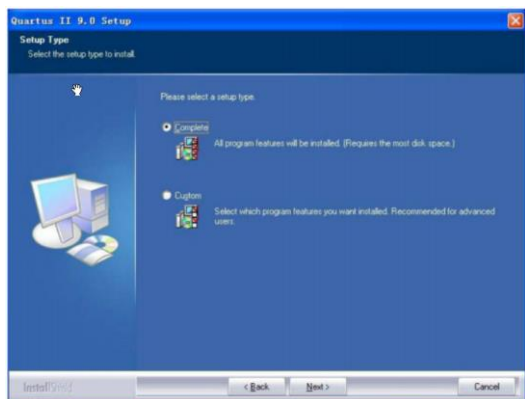


图 2-10 完全安装

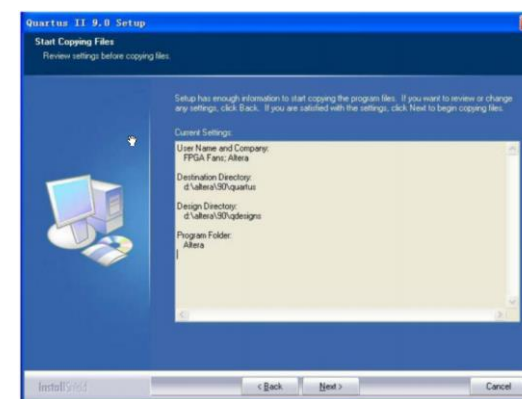


图 2-11 确认安装信息

第十一步：开始安装，此步骤需要的时间最多，请耐心等待。



图 2-12 安装进度



图 2-13 安装确认放置桌面快捷方式

第十二步：安装完成，提示是否要创建快捷方式图标在桌面上，点“是”。最后点击 Finish,完成安装。

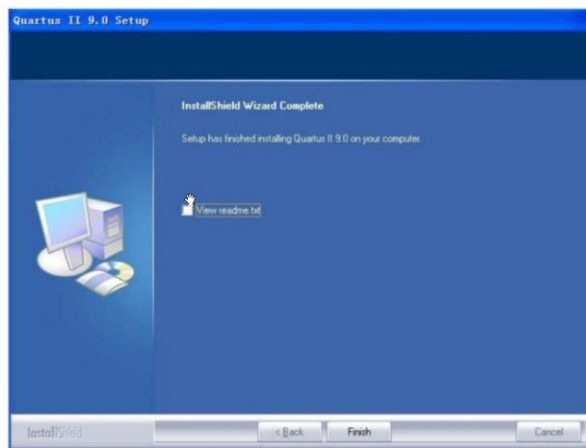


图 2-14 完成安装

2.2 Quartus II 软件安装后处理

软件安装好后，为了方便用于学习目的可以进行“和谐处理”，处理步骤：

(1) 用 Quartus_II_9.0 破解器.exe 替换掉 C:\altera\90\quartus\bin 下的 sys_cpt.dll 和 quartus.exe 文件（运行 Quartus_II_9.0 破解器.exe 后,直接点击“应用补丁”，如果出现“未找到该文件。搜索该文件吗？”，点击“是”，如果直接把该破解器 Copy 到 C:\altera\90\quartus\bin 下，就不会出现这个对话框，而是直接开始破解！）然后选中 sys_cpt.dll，点击“打开”。安装默认的 sys_cpt.dll 路径是在 C:\altera\90\quartus\bin 下）。

(2) 把 license.dat 用记事本的形式打开，把里面的 XXXXXXXXXXXX 用网卡号替换(在 Quartus II 9.0 的 Tools 菜单下选择 License Setup，下面就有

NIC ID)。

(3) 在 Quartus II 9.0 的 Tools 菜单下选择 License Setup，然后选择 Licensefile，最后点击 OK。

注意：license 文件存放的路径名称不能包含汉字和空格，空格可以用下划线代替，其他版本的 Quartus II 软件破解过程基本一样。

2.3 USB-Blaster 驱动安装

USB-Blaster 是 Altera 公司 FPGA/CPLD 的下载程序工具，采用 USB 接口，其驱动程序在 Quartus II 安装目录下已经包含有，Win7 系统下其驱动安装具体步骤如下：

Win7 系统下安装时，插上 usb blaster 后，有可能会提示如下：

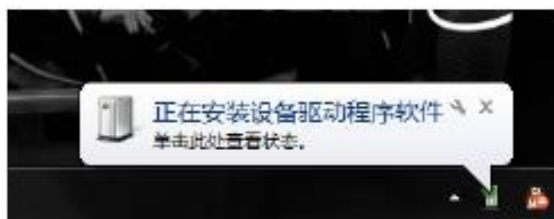


图 2-15 usb blaster 插入提示

有的电脑不提示，不管怎样，我们按下面步骤来操作即可：

- 1、第一步：打开 “开始菜单” 找到“设备和打印机”打开。
- 2、点击未指定的 USB Composite Device，也有的叫“未知设备”，见红色框：



图 2-16 打开设备界面找到未知设备

3、双击打开“未知设备”，或是右击，点硬件>属性>驱动程序>更新驱动程序。

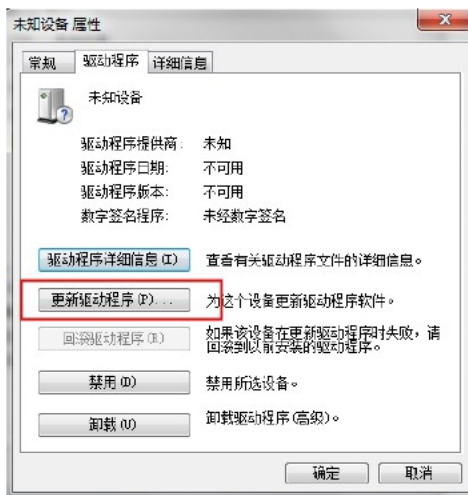


图 2-17 更新驱动程序



图 2-18 选择浏览计算机驱动程序

到这步，要找到驱动程序所在安装目录，根据您的机器上的情况选择，一般驱动程序在 Quartus 安装文件路径下：C:\altera\90\quartus\drivers\usb-blaster，点击下一步。在弹出的 Windows 安全对话框中选择“始终安装此驱动程序软件”。



图 2-19 找到驱动程序目录



图 2-20 点击始终安装此驱动程序软件

到此，等待驱动程序安装，最后弹出驱动安装成功窗口。

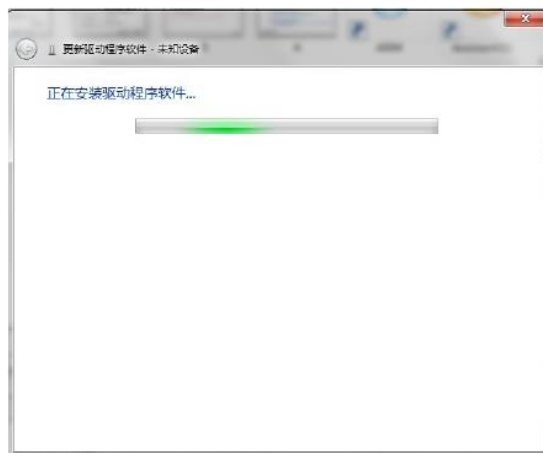


图 2-21 安装驱动程序进程

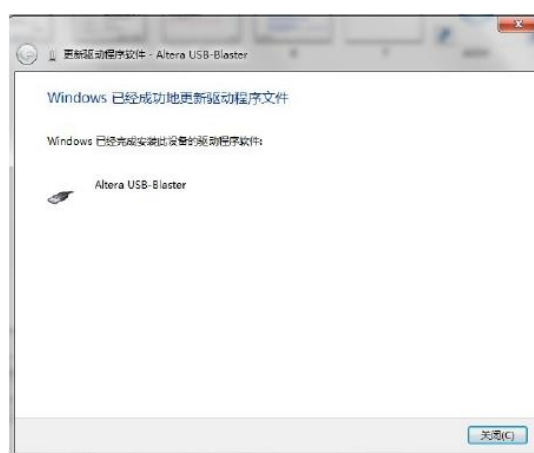


图 2-21 完成安装驱动窗口

安装完成后在设备管理器中找到 Altera USB-Blaster 设备，证明安装驱动成功了。



图 2-22 显示 Altera USB-Blaster 设备

第三章 FPGA 原理和结构

可编程的“万能芯片”FPGA——现场可编程门阵列，是指一切通过软件手段更改、配置器件内部连接结构和逻辑单元，完成既定设计功能的数字集成电路。内部包括可配置逻辑模块 CLB（Configurable Logic Block）、输出输入模块 IOB（InputOutputBlock）和内部连线（Interconnect）三个部分。

3.1 FPGA 技术原理

FPGA 是在 PAL、GAL、EPLD、CPLD 等可编程器件的基础上进一步发展的产物。它是作为 ASIC 领域中的一种半定制电路而出现的，即解决了定制电路的不足，又克服了原有可编程器件门电路有限的缺点。

由于 FPGA 需要被反复烧写，它实现组合逻辑的基本结构不可能像 ASIC 那样通过固定的与非门来完成，而只能采用一种易于反复配置的结构。查找表可以很好地满足这一要求，目前主流 FPGA 都采用了基于 SRAM 工艺的查找表结构，也有一些军品和宇航级 FPGA 采用 Flash 或者熔丝与反熔丝工艺的查找表结构。通过烧写文件改变查找表内容的方法来实现对 FPGA 的重复配置。

根据数字电路的基本知识可以知道，对于一个 n 输入的逻辑运算，不管是与或非运算还是异或运算等等，最多只可能存在 2^n 种结果。所以如果事先将相应的结果存放于一个存储单元，就相当于实现了与非门电路的功能。FPGA 的原理也是如此，它通过烧写文件去配置查找表的内容，从而在相同的电路情况下实现了不同的逻辑功能。

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 可以看成是一个有 4 位地址线的 RAM。当用户通过原理图或 HDL 语言描述了一个逻辑电路以后，PLD/FPGA 开发软件会自动计算逻辑电路的所有可能结果，并把真值表(即结果)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。

下面给出一个 4 与门电路的例子来说明 LUT 实现逻辑功能的原理。

例 1-1：给出一个使用 LUT 实现 4 输入与门电路的真值表。

表 3-1 4 输入与门的真值表

实际逻辑电路		LUT的实现方式	
a, b, c, d输入	逻辑输出	RAM地址	RAM中存储的内容
0000	0	0000	0
0001	0	0001	0
...	...	...	...
1111	1	1111	1

从中可以看到，LUT 具有和逻辑电路相同的功能。实际上，LUT 具有更快的执行速度和更大的规模。

由于基于 LUT 的 FPGA 具有很高的集成度，其器件密度从数万门到数千万门不等，可以完成极其复杂的时序与逻辑组合逻辑电路功能，所以适用于高速、高密度的高端数字逻辑电路设计领域。其组成部分主要有可编程输入/输出单元、基本可编程逻辑单元、内嵌 SRAM、丰富的布线资源、底层嵌入功能单元、内嵌专用单元等，主要设计和生产厂家有 Xilinx、Altera、Lattice、Actel、Atmel 和 QuickLogic 等公司，其中最大的是 Xilinx、Altera、Lattice 三家。

目前，FPGA 市场占有率最高的两大公司 Xilinx 和 Altera 生产的 FPGA 都是基于 SRAM 工艺的，需要在使用时外接一个片外存储器以保存程序。上电时，FPGA 将外部存储器中的数据读入片内 RAM，完成配置后，进入工作状态；掉电后 FPGA 恢复为白片，内部逻辑消失。这样 FPGA 不仅能反复使用，还无需专门的 FPGA 编程器，只需通用的 EPROM、PROM 编程器即可。Actel、QuickLogic 等公司还提供反熔丝技术的 FPGA，只能下载一次，具有抗辐射、耐高低温、低功耗和速度快等优点，在军品和航空航天领域中应用较多，但这种 FPGA 不能重复擦写，开发初期比较麻烦，费用也比较昂贵。

3.2FPGA 芯片结构

目前主流的 FPGA 仍是基于查找表技术的，已经远远超出了先前版本的基本性能，并且整合了常用功能(如 RAM、时钟管理和 DSP)的硬核(ASIC 型)模块。如图 1-1 所示(注：图 1-1 只是一个示意图，实际上每一个系列的 FPGA 都有其相应的内部结构)，FPGA 芯片主要由 6 部分组成，分别为：可编程输入输出单元、基本可编程逻辑单元、完整的时钟管理、嵌入块式 RAM、丰富的布线资源、内嵌的底层功能单元和内嵌专用硬件模块。

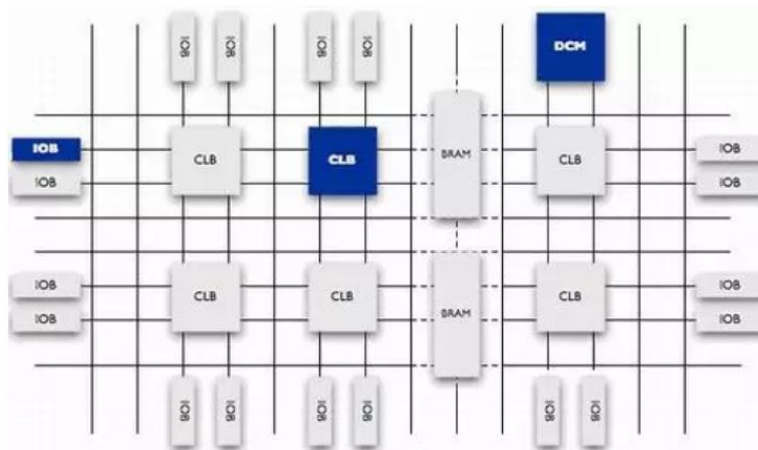


图 3-2 FPGA 内部结构

1. 可编程输入输出单元(IOB)

可编程输入/输出单元简称 I/O 单元，是芯片与外界电路的接口部分，完成不同电气特性下对输入/输出信号的驱动与匹配要求，其示意结构如下图所示。FPGA 内的 I/O 按组分类，每组都能够独立地支持不同的 I/O 标准。通过软件的灵活配置，可适配不同的电气标准与 I/O 物理特性，可以调整驱动电流的大小，可以改变上、下拉电阻。目前，I/O 口的频率也越来越高，一些高端的 FPGA 通过 DDR 寄存器技术可以支持高达 2Gbps 的数据速率。

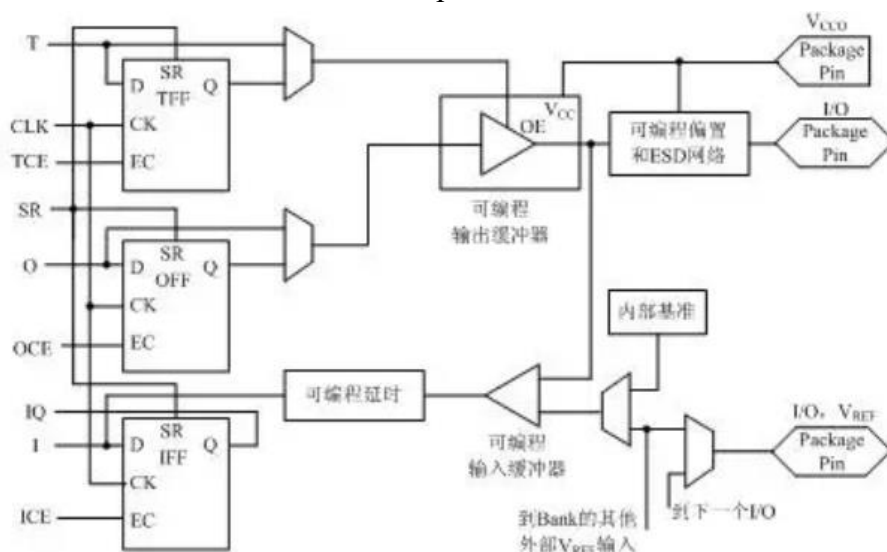


图 3-3 典型的 IOB 内部结构示意图

外部输入信号可以通过 IOB 模块的存储单元输入到 FPGA 的内部，也可以直接输入 FPGA 内部。当外部输入信号经过 IOB 模块的存储单元输入到 FPGA 内部时，其保持时间(Hold Time)的要求可以降低，通常默认为 0。

为了便于管理和适应多种电器标准，FPGA 的 IOB 被划分为若干个组(bank)，每个 bank 的接口标准由其接口电压 VCCO 决定，一个 bank 只能有一种 VCCO，

但不同 bank 的 VCCO 可以不同。只有相同电气标准的端口才能连接在一起，VCCO 电压相同是接口标准的基本条件。

2. 可配置逻辑块(CLB)

CLB 是 FPGA 内的基本逻辑单元。CLB 的实际数量和特性会依器件的不同而不同，但是每个 CLB 都包含一个可配置开关矩阵，此矩阵由 4 或 6 个输入、一些选型电路(多路复用器等)和触发器组成。开关矩阵是高度灵活的，可以对其进行配置以便处理组合逻辑、移位寄存器或 RAM。在 Xilinx 公司的 FPGA 器件中，CLB 由多个(一般为 4 个或 2 个)相同的 Slice 和附加逻辑构成，如图 1-3 所示。每个 CLB 模块不仅可以用于实现组合逻辑、时序逻辑，还可以配置为分布式 RAM 和分布式 ROM。

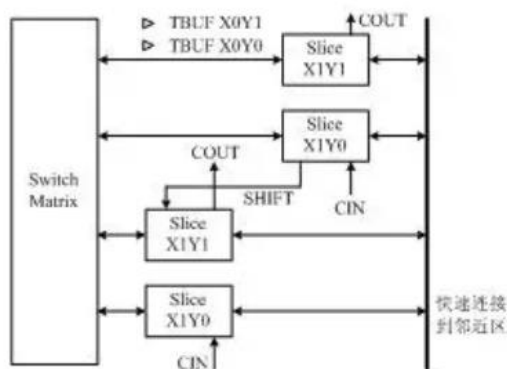


图 3-4 典型的 CLB 结构示意图

3. 数字时钟管理模块(DCM)

业内大多数 FPGA 均提供数字时钟管理(Xilinx 的全部 FPGA 均具有这种特性)。Xilinx 推出最先进的 FPGA 提供数字时钟管理和相位环路锁定。相位环路锁定能够提供精确的时钟综合，且能够降低抖动，并实现过滤功能。

4. 嵌入式块 RAM(BRAM)

大多数 FPGA 都具有内嵌的块 RAM，这大大拓展了 FPGA 的应用范围和灵活性。块 RAM 可被配置为单端口 RAM、双端口 RAM、内容地址存储器(CAM)以及 FIFO 等常用存储结构。RAM、FIFO 是比较普及的概念，在此就不冗述。CAM 存储器在其内部的每个存储单元中都有一个比较逻辑，写入 CAM 中的数据会和内部的每一个数据进行比较，并返回与端口数据相同的所有数据的地址，因而在路由的地址交换器中有广泛的应用。除了块 RAM，还可以将 FPGA 中的 LUT 灵活地配置成 RAM、ROM 和 FIFO 等结构。在实际应用中，芯片内部块 RAM 的数量也是选择芯片的一个重要因素。

单片块 RAM 的容量为 18k 比特，即位宽为 18 比特、深度为 1024，可以根据需要改变其位宽和深度，但要满足两个原则：首先，修改后的容量(位宽 深度)不能大于 18k 比特;其次，位宽最大不能超过 36 比特。当然，可以将多片块 RAM 级联起来形成更大的 RAM，此时只受限于芯片内块 RAM 的数量，而不再受上面两条原则约束。

5. 丰富的布线资源

布线资源连通 FPGA 内部的所有单元，而连线的长度和工艺决定着信号在连线上的驱动能力和传输速度。FPGA 芯片内部有着丰富的布线资源，根据工艺、长度、宽度和分布位置的不同而划分为 4 类不同的类别。第一类是全局布线资源，用于芯片内部全局时钟和全局复位/置位的布线;第二类是长线资源，用以完成芯片 Bank 间的高速信号和第二全局时钟信号的布线;第三类是短线资源，用于完成基本逻辑单元之间的逻辑互连和布线;第四类是分布式的布线资源，用于专有时钟、复位等控制信号线。

在实际中设计者不需要直接选择布线资源，布局布线器可自动地根据输入逻辑网表的拓扑结构和约束条件选择布线资源来连通各个模块单元。从本质上讲，布线资源的使用方法和设计的结果有密切、直接的关系。

6. 底层内嵌功能单元

内嵌功能模块主要指 DLL(Delay Locked Loop)、PLL(Phase Locked Loop)、DSP 和 CPU 等软处理核(Soft Core)。现在越来越丰富的内嵌功能单元，使得单片 FPGA 成为了系统级的设计工具，使其具备了软硬件联合设计的能力，逐步向 SOC 平台过渡。

DLL 和 PLL 具有类似的功能，可以完成时钟高精度、低抖动的倍频和分频，以及占空比调整和移相等功能。Xilinx 公司生产的芯片上集成了 DLL，Altera 公司的芯片集成了 PLL，Lattice 公司的新型芯片上同时集成了 PLL 和 DLL。PLL 和 DLL 可以通过 IP 核生成的工具方便地进行管理和配置。

7. 内嵌专用硬核

内嵌专用硬核是相对底层嵌入的软核而言的，指 FPGA 处理能力强大的硬核(Hard Core)，等效于 ASIC 电路。为了提高 FPGA 性能，芯片生产商在芯片内部集成了一些专用的硬核。例如：为了提高 FPGA 的乘法速度，主流的 FPGA 中都集成了专用乘法器;为了适用通信总线与接口标准，很多高端的 FPGA 内部都集成了串并收发器(SERDES)，可以达到数十 Gbps 的收发速度。

Xilinx 公司的高端产品不仅集成了 Power PC 系列 CPU，还内嵌了 DSP Core 模块，其相应的系统级设计工具是 EDK 和 Platform Studio，并依此提出了片上系统(System on Chip)的概念。通过 PowerPC、Miroblaze、Picoblaze 等平台，能够开

发标准的 DSP 处理器及其相关应用，达到 SOC 的开发目的。

3.3 软核、硬核以及固核的概念

IP(Intelligent Property)核是具有知识产权核的集成电路芯核总称，是经过反复验证过的、具有特定功能的宏模块，与芯片制造工艺无关，可以移植到不同的半导体工艺中。到了 SOC 阶段，IP 核设计已成为 ASIC 电路设计公司和 FPGA 提供商的重要任务，也是其实力体现。对于 FPGA 开发软件，其提供的 IP 核越丰富，用户的设计就越方便，其市场占用率就越高。目前，IP 核已经变成系统设计的基本单元，并作为独立设计成果被交换、转让和销售。

从 IP 核的提供方式上，通常将其分为软核、硬核和固核这 3 类。从完成 IP 核所花费的成本来讲，硬核代价最大；从使用灵活性来讲，软核的可复用使用性最高。

1. 软核

软核在 EDA 设计领域指的是综合之前的寄存器传输级(RTL)模型；具体在 FPGA 设计中指的是对电路的硬件语言描述，包括逻辑描述、网表和帮助文档等。软核只经过功能仿真，需要经过综合以及布局布线才能使用。其优点是灵活性高、可移植性强，允许用户自配置；缺点是对模块的预测性较低，在后续设计中存在发生错误的可能性，有一定的设计风险。软核是 IP 核应用最广泛的形式。

2. 固核

固核在 EDA 设计领域指的是带有平面规划信息的网表；具体在 FPGA 设计中可以看做带有布局规划的软核，通常以 RTL 代码和对应具体工艺网表的混合形式提供。将 RTL 描述结合具体标准单元库进行综合优化设计，形成门级网表，再通过布局布线工具即可使用。和软核相比，固核的设计灵活性稍差，但在可靠性上有较大提高。目前，固核也是 IP 核的主流形式之一。

3. 硬核

硬核在 EDA 设计领域指经过验证的设计版图；具体在 FPGA 设计中指布局和工艺固定、经过前端和后端验证的设计，设计人员不能对其修改。不能修改的原因有两个：首先是系统设计对各个模块的时序要求很严格，不允许打乱已有的物理版图；其次是保护知识产权的要求，不允许设计人员对其有任何改动。IP 硬核的不许修改特点使其复用有一定的困难，因此只能用于某些特定应用，使用范围较窄。

第四章 FPGA 的开发流程

利用 FPGA 技术进行开发设计的大部分工作是在 EDA 软件工作平台上进行的，FPGA 设计开发流程如图 4-1 所示。FPGA 设计流程包括设计准备、设计输入、设计处理和器件编程四个步骤以及相应的功能仿真、时序仿真和器件测试三个设计验证过程。下面以 quartus II 开发软件为平台介绍 FPGA 的开发流程。

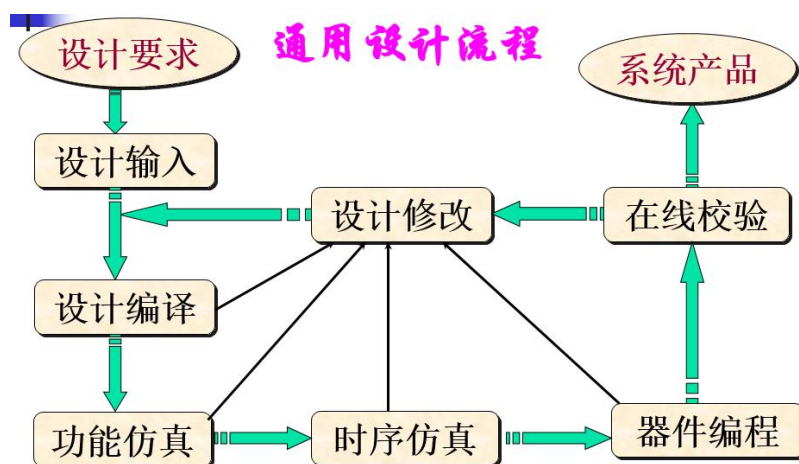


图 4-1 FPGA 设计开发流程

4.1 设计的输入

1. 创建工程

可通过创建工程向导创建一个的工程，在 Quartus II 集成环境下，执行“File”菜单的“New Project Wizard”命令，弹出新建设计工程对话框。设置好工程保存路径、设计工程的名称和顶层文件实体名，如图 4-2 所示。

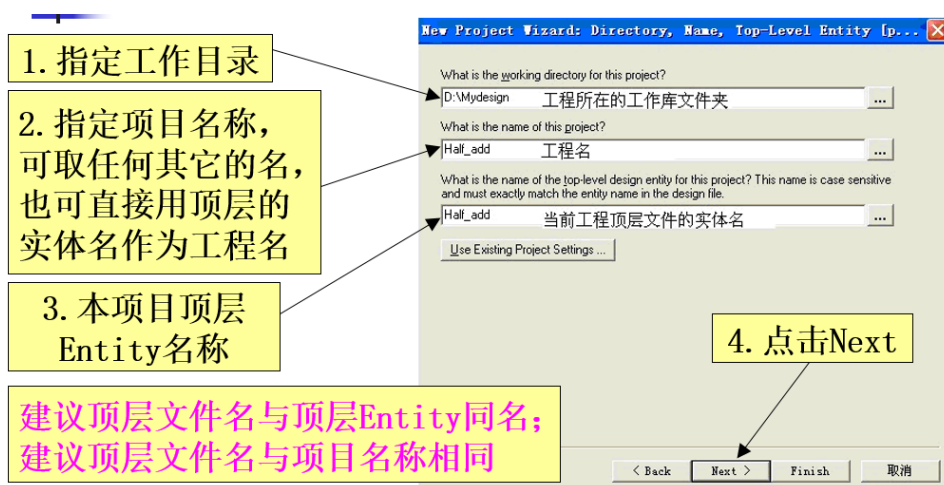


图 4-2 新建项目对话框（第 1 页面）

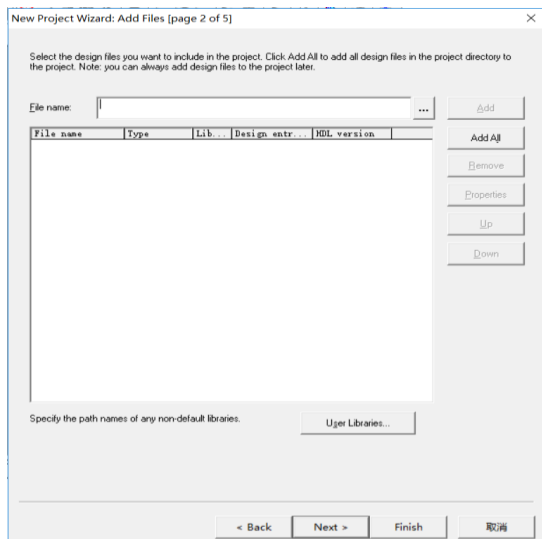


图 4-2 添加文件对话框（第 2 页面）

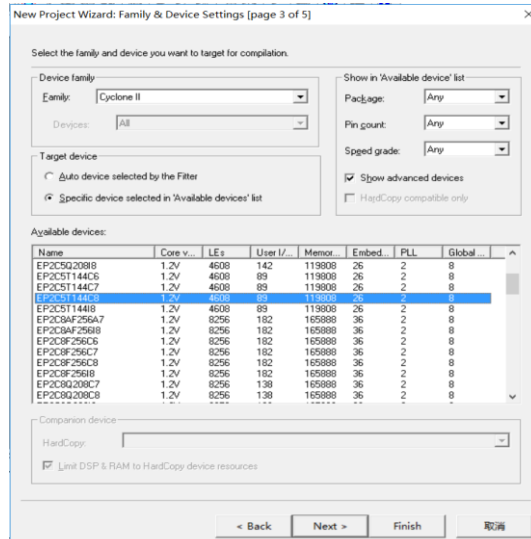


图 4-3 选择器件对话框（第 3 页面）

如果顶层设计文件和其他底层设计文件已经包含在工程文件夹中，则可在图 4-2 添加文件对话框中将这些设计文件增加到新建工程中。在新建工程时应先选择下载的目标芯片，否则系统将以默认的目标芯片为基础完成设计文件的编译。目标芯片选择应根据支持硬件开发和验证的开发板或试验开发系统上提供的可编程逻辑器件来决定。这里选择 Cyclone II 系列的 EP2C5T144C8 芯片，如图 4-3 所示。

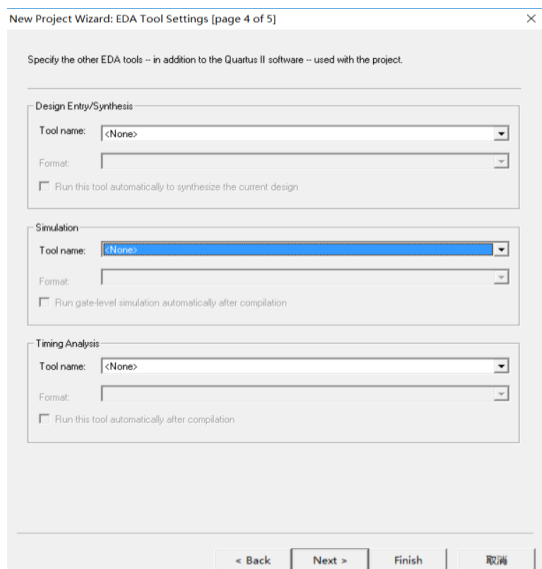


图 4-4 选择设定第三方工具



图 4-5 显示设置完成信息

用户可以选择所用到的第三方工具，比如 ModleSim、Synplify 等。在本例中并没有调用第三方工具，可以都不选，用鼠标单击图 4-4 对话框下方的“Next”

按钮，进入如图 4-5 所示的显示设置完成信息页面。此页面用于显示新设计工程的摘要（Summary）。用鼠标单击此页面下方的“Finish”按钮，完成新设计工程的建立。

2. 原理图输入

原理图输入法也称为图形编辑输入法，用 Quartus II 原理图输入设计法进行数字系统设计时，不需要任何硬件描述语言的知识，在具有数字逻辑电路基本知识的基础上，利用 Quartus II 软件提供的 EDA 平台设计数字电路或系统。

图形输入的简要步骤如下：

1. 选择【File】--【New】选项，打开新建文件类型选择窗口；
2. 选择 Block Diagram/Schematic File 打开图形编辑输入窗口，如图 4-6 所示；（这里以一个半加器为例子）

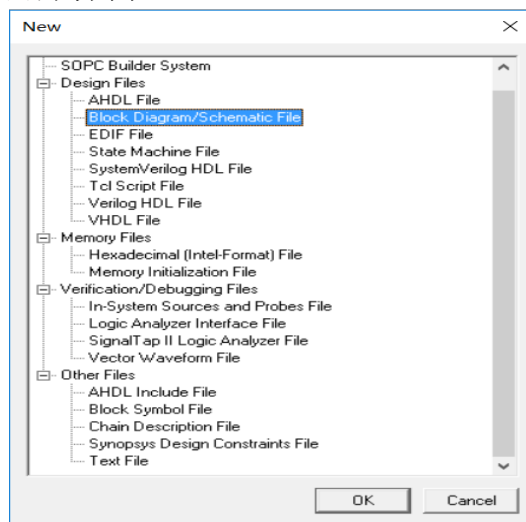


图 4-6 新建图形输入文件

3. 根据半加器的逻辑功能列出真值表。

表 4.1 半加器真值表

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

4. 根据真值表的进行公式化简得出逻辑图。

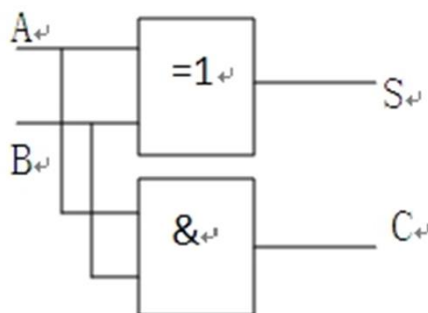


图 4-7 半加器的逻辑图

5. 进入原理图设计界面查找放置器件，打开如图 4-8 所示的图形编辑器对话框，进行设计输入。

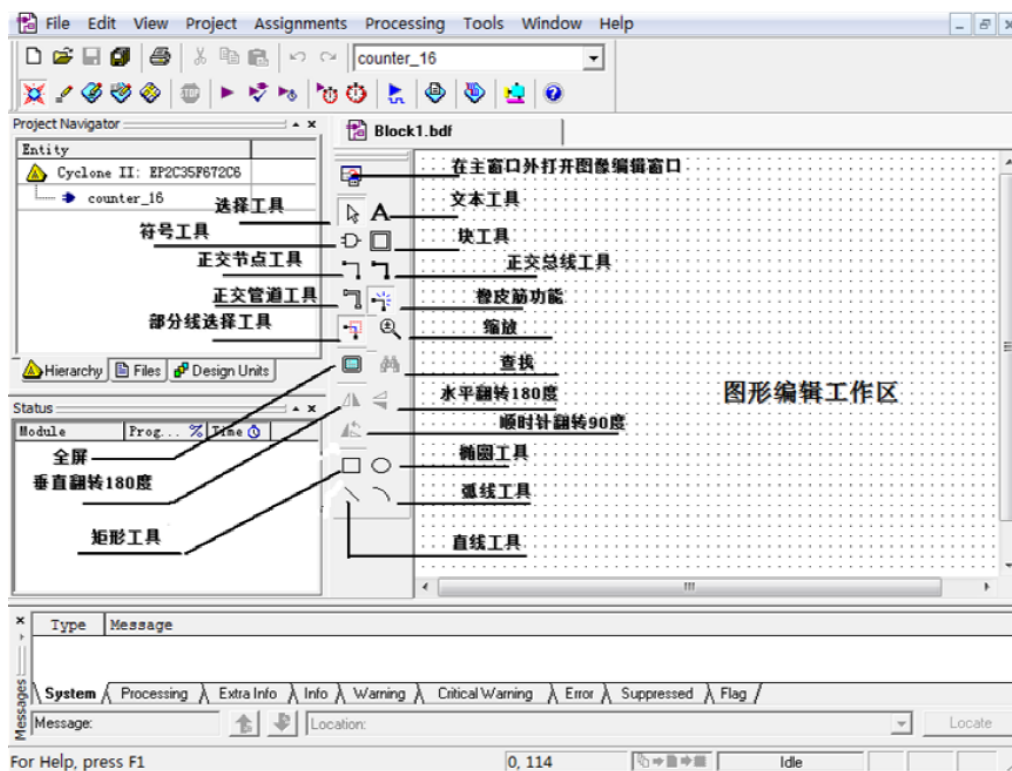


图 4-8 图形编辑器界面

6. 在图形编辑窗口中任一个位置双击鼠标，或点击图中的“符号工具”按钮，或选择 菜单 Edit 下的 Insert Symbol 命令，弹出下图所示的元件选择窗口：

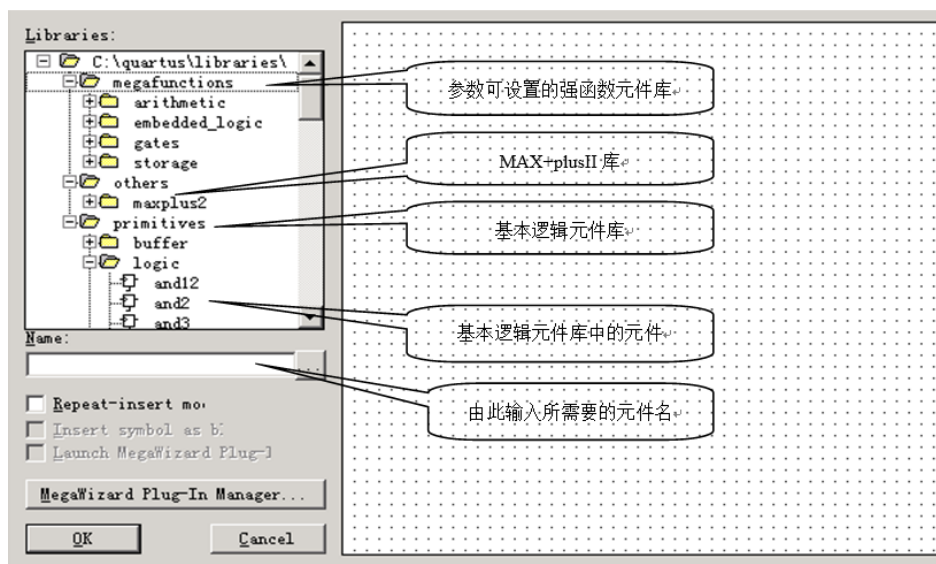


图 4-9 Symbol 对话框

在元件选择窗口的符号库“Libraries”栏目中，用鼠标选择基本逻辑元件库（primitives）文件夹中的逻辑库（logic）后，该库的基本元件的元件名将出现在“Libraries”栏目中。例如 and2（2 输入端的与门）、xor（异或门）、vcc（电源）、input（输入）、output（输出）等。在元件选择窗口的“Name”栏目内直接输入元件名，或者在“Libraries”栏目中，用鼠标单击元件名，可得到相应的元件符号。元件选中后用鼠标单击“OK”键，选中的元件符号将出现在原理图编辑窗口中。

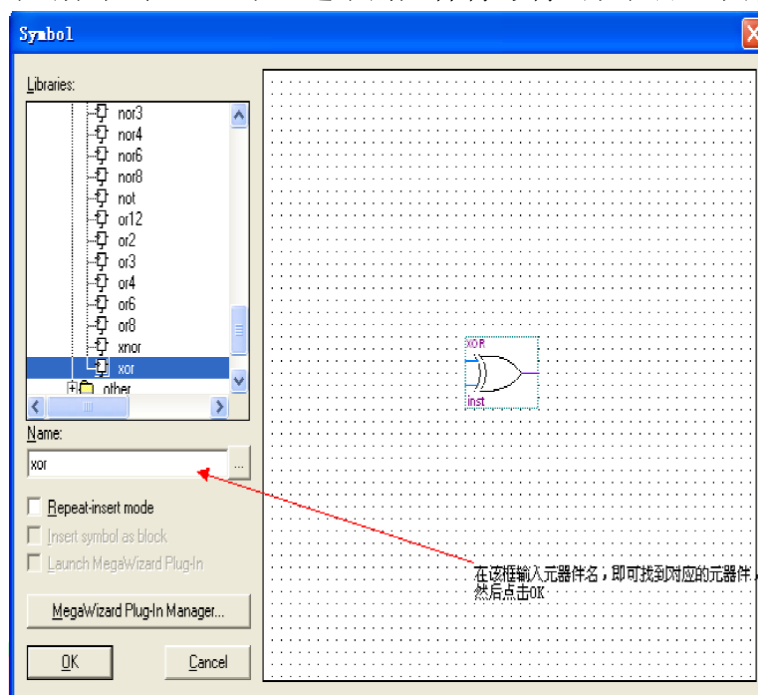


图 4-10 元器件的查找

7. 查找放置元器件、输入输出端口，用鼠标完成电路内部的连接以及与输入、输出端口的连接。

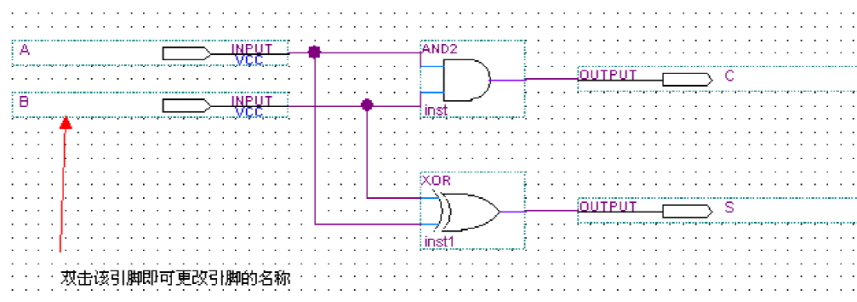


图 4-11 元器件的输入和引脚名的更改

8.原理图输入法—优缺点：

优点：

1) 可以与传统的数字电路设计法接轨，即使用传统设计方法得到电路原理图，然后在 Quartus II 平台完成设计电路的输入、仿真验证和综合，最后下载到目标芯片中。

2) 它将传统的电路设计过程的布局布线、绘制印刷电路板、电路焊接、电路加电测试等过程取消，提高了设计效率，降低了设计成本，减轻了设计者的劳动强度。

缺点：

1) 原理图设计方法没有实现标准化，不同的 EDA 软件中的图形处理工具对图形的设计规则、存档格式和图形编译方式都不同，因此兼容性差，难以交换和管理。

2) 由于兼容性不好，性能优秀的电路模块的移植和再利用非常困难难以实现用户所希望的面积、速度以及不同风格的综合优化

3) 原理图输入的设计方法不能实现真实意义上的自顶向下的设计方案，无法建立行为模型，从而偏离了电子设计自动化最本质的涵义。

3.文本输入

Quartus II 自带文本编辑器，用于程序设计输入。

与图形输入操作类似，打开新建文件类型选择窗口，选择 Verilog HDL File 输入方式，即可打开文本编辑器；如图 4-13 所示，在编辑器中完成程序代码的编写和保存，保存时的文件命名要与模块名一致，否则编译会出错的。

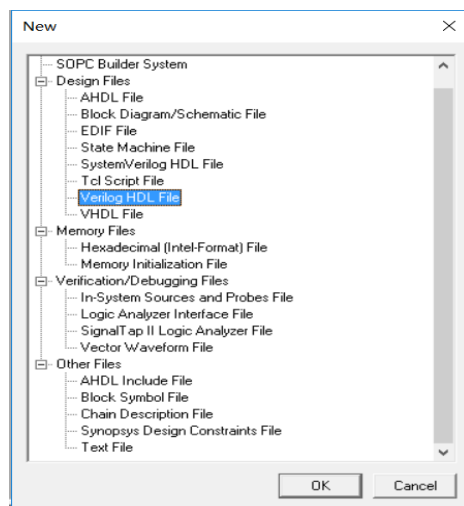


图 4-12 新建文本输入文件

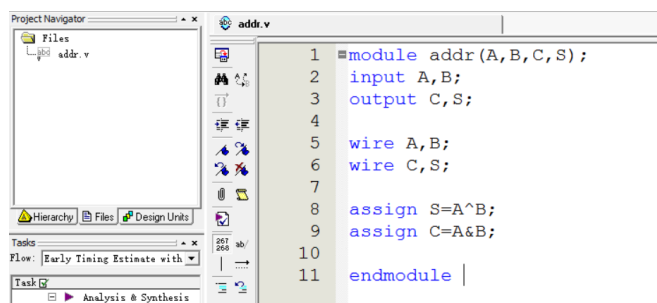


图 4-13 完成代码输入和保存

4.2. 设计的编译

Quartus II 编译器的主要任务是对设计项目进行检查并完成逻辑综合，同时将项目最终设计结果生成器件的下载文件。编译开始前，可以先对工程的参数进行设置。

Quartus II 软件中的编译类型有全编译和分步编译两种。

选择 Quartus II 主窗口 **Process** 菜单下 **Start Compilation** 命令，或者在主窗口的工具栏上直接点击图标  可以进行全编译；

全编译的过程包括分析与综合（**Analysis & Synthesis**）、适配（**Fitter**）、编程（**Assembler**）、时序分析（**Classical Timing Analysis**）这 4 个环节，而这 4 个环节各自对应相应的菜单命令，可以单独分步执行，也就是分步编译。

分步编译就是使用对应命令分步执行对应的编译环节，每完成一个编译环节，生成一个对应的编译报告。分步编译跟全编译一样分为四步：

1、分析与综合（**Analysis & Synthesis**）：设计文件进行分析和检查输入文件是否有错误，对应的菜单命令是 Quartus II 主窗口 **Process** 菜单下 **Start\Start Analysis & Synthesis**，对应的快捷图标是在主窗口的工具栏上的 ；

2、适配（**Fitter**）：在适配过程中，完成设计逻辑器件中的布局布线、选择适

当的内部互连路径、引脚分配、逻辑元件分配等，对应的菜单命令是 Quartus II 主窗口 Process 菜单下 Start\Start Fitter; (注：两种编译方式引脚分配有所区别)

3、编程 (Assembler): 产生多种形式的器件编程映像文件，通过软件下载到目标器件当中去，菜单命令是 Quartus II 主窗口 Process 菜单下 Start\Start Assembler;

4、时序分析(Classical Timing Analyzer): 计算给定设计与器件上的延时，完成设计分析的时序分析和所有逻辑的性能分析，菜单命令是 Quartus II 主窗口 Process 菜单下 Start\Start Classical Timing Analyzer，对应的快捷图标是在主窗口的工具栏上的。

编译完成后，编译报告窗口 Compilation Report 会报告工程文件编译的相关信息，如编译的顶层文件名、目标芯片的信号、引脚的数目等。

全编译操作简单，适合简单的设计。对于复杂的设计，选择分步编译可以及时发现问题，提高设计纠错的效率，从而提高设计效率。

执行 Quartus II 主窗口 “Processing” 菜单的 “Start Compilation” 命令，或者在主窗口上直接用鼠标单击 “开始编译” 命令按钮图标，开始对 addr.bdf 文件进行编译。编译工具的编译进程可以在如图 4-14 所示的状态 (Status) 窗口上看到。编译过程包括分析与综合、适配、编程和时序分析等 4 个环节，都是由软件自动完成。

Status			
Module	Progress %	Time	
Full Compilation	100 %	00:00:07	
- Analysis & Synthesis	100 %	00:00:02	
- Fitter	100 %	00:00:02	
- Assembler	100 %	00:00:02	
- Classic Timing Analyzer	100 %	00:00:01	

图 4-14 Quartus II 编译状态窗口

在设计输入完成后，可以通过选择菜单【Processing】-【Start】-【Start Analysis & Elaboration】选项，对输入进行分析，如果存在错误，信息窗口将出现错误信息。分析完成后，可通过菜单【Tools】-【Netlist viewer】-【RTL Viewer】查看设计对应的寄存器传输级视图。

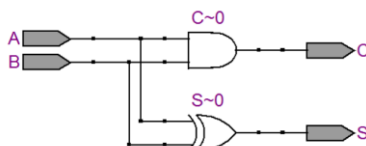


图 4-15 RTL 视图

4.3. 仿真验证

验证设计时序是否符合要求，在设计输入完成后要进行仿真；FPGA 仿真也可在第三方工具软件中完成，工具有 modelsim 、 Activehdl 、NC-Verilog/Vhdl、各厂家自带工具等。常用的仿真有两种方式，一种是用 Quartus II 软件自带的仿真工具，另外一种是采用专业的仿真软件 modelsim，下面分别介绍。

4.3.1 Quartus II 仿真验证

Quartus II 9.0 版本软件自带有仿真功能，仿真的目的就是在软件环境下，验证电路的行为和设想中的是否一致。FPGA/CPLD 中的仿真分为功能仿真和时序仿真。功能仿真着重考察电路在理想环境下的行为和设计构想的一致性，时序仿真则在电路已经映射到特定的工艺环境后，考察器件在延时情况下对布局布线网表文件进行的一种仿真。

仿真一般需要建立波形文件、输入信号节点、编辑输入信号、波形文件的保存和运行仿真器等过程。

1、建立波形文件

波形文件用来为设计产生输入激励信号。利用 QuartusII 波形编辑器可以创建矢量波形文件(.vwf) ， 创建一个新的矢量波形文件步骤如下：

(1) 选择 Quartus II 主界面 File 菜单下的 New 命令，弹出新建对话框；

(2) 在新建对话框中选择 Verification/Debugging Files 文件下的 Vector Waveform File，点击 OK 按钮，则打开一个空的波形编辑器窗口，主要分为信号栏、工具栏和波形栏 。

(3) 添加仿真测试信号，设置输入信号和仿真参数设置。

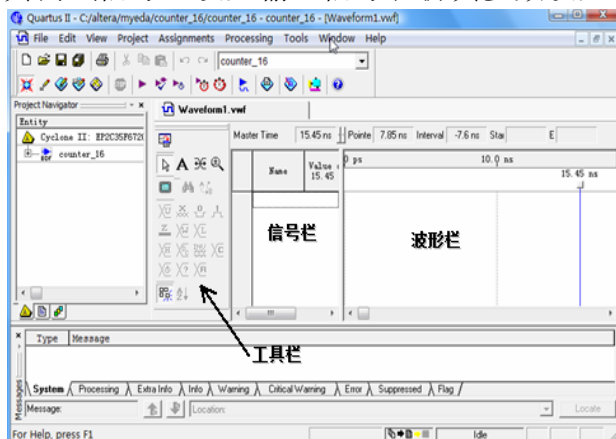


图 4-16 波形编辑器窗口

2、输入信号节点

(1) 在波形编辑方式下，执行 Edit 菜单中的 Insert Node or Bus 命令，或者在波形编辑器左边 Name 列的空白处点击鼠标右键，弹出的 Insert Node or Bus

对话框, 或者在波形编辑器左边 Name 列的空白处双击左键, 弹出的 Insert Node or Bus 对话框。

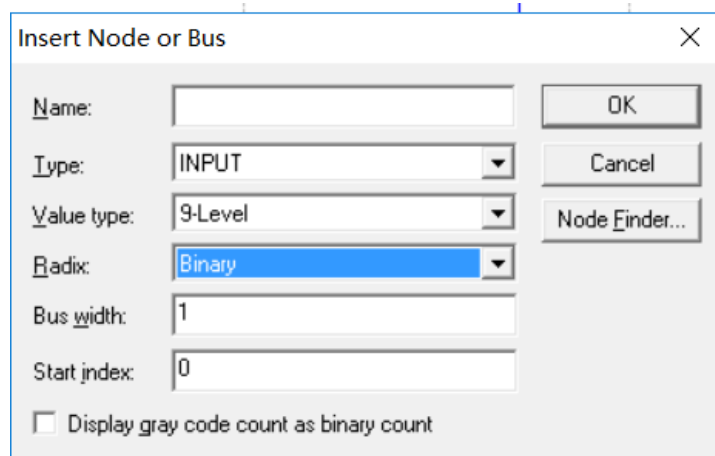


图 4-17 Insert Node or Bus 对话框

(2) 点击 Insert Node or Bus 对话框中的 Node Finder... 按钮, 弹出 Node Finder 窗口, 在此窗口中添加信号节点。

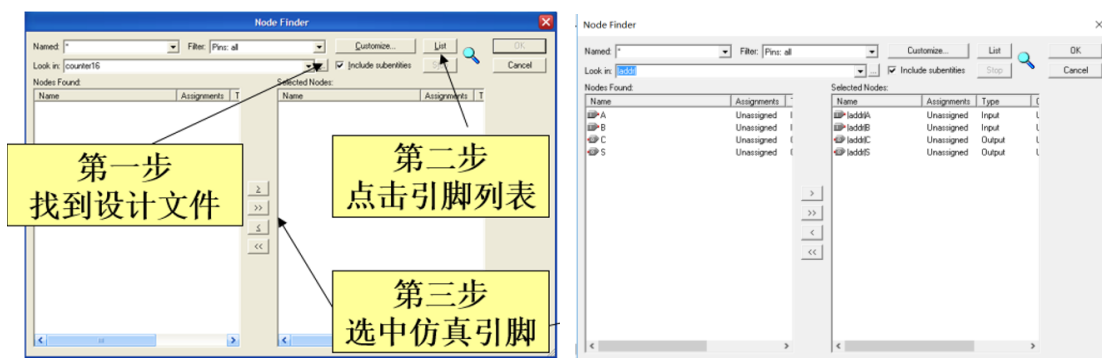


图 4-18 Node Finder 窗口添加信号

3、编辑输入信号

编辑输入信号是指在波形编辑器中指定输入节点的逻辑电平变化, 编辑输入节点的波形。

在仿真编辑窗口的工具栏中列出了各种功能选择按钮, 主要用于绘制、编辑波形, 给输入信号赋值。 具体功能如下:

- : 在波形文件中添加注释;
- : 修改信号的波形值, 把选定区域的波形更改成原值的相反值;
- : 全屏显示波形文件;
- : 放大、缩小波形;
- : 在波形文件信号栏中查找信号名, 可以快捷找到待观察信号;
- : 将某个波形替换为另一个波形;
- : 给选定信号赋原值的反值;

 : 输入任意固定的值;
 : 输入随机值
       : U 给选定的信号赋值, X 表示不定态, 0 表示赋 0, 1 表示赋 1, Z 表示高阻态, W 表示弱信号, L 表示低电平, H 表示高电平, DC 表示不赋值。

 : 设置时钟信号的波形参数, 先选中需要赋值的信号, 然后鼠标右键点击此图标弹出 Clock 对话框, 在此对话框中可以设置输入时钟信号的起始时间(Start Time)、结束时间(End Time)、时钟脉冲周期(Period), 相位偏置(Offset)以及占空比。

 : 给信号赋计数值, 先选中需要赋值的信号, 然后鼠标右键点击此图标弹出如下图所示的 Count Value 对话框, 然后赋值。

设计的半加器仿真输入信号波形设置如下图所示, 用波形编辑工具  设置输入信号。

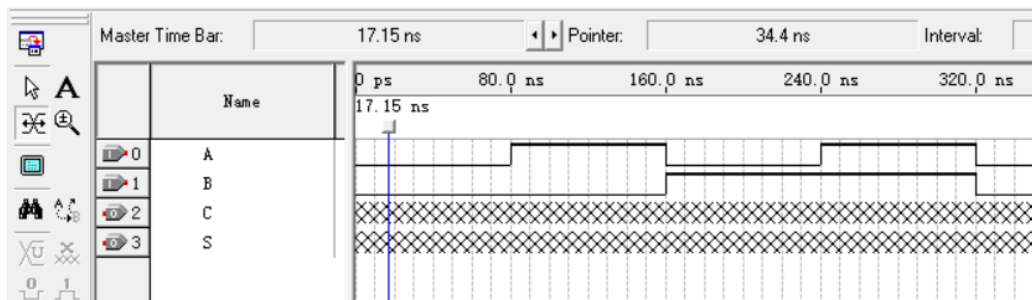


图 4-19 半加器仿真输入信号值

4、仿真波形文件

Quartus II 软件中默认的是时序仿真, 如果进行功能仿真则需要先对仿真进行设置, 步骤如下:

(1) 选择 Quartus II 主窗口 Assignments 菜单下的 Settings... 命令, 可以进入参数设置页面, 如下图所示, 然后单击 Simulation Settings, 在右边的对话框中的 Simulation mode 中选择“Function”;

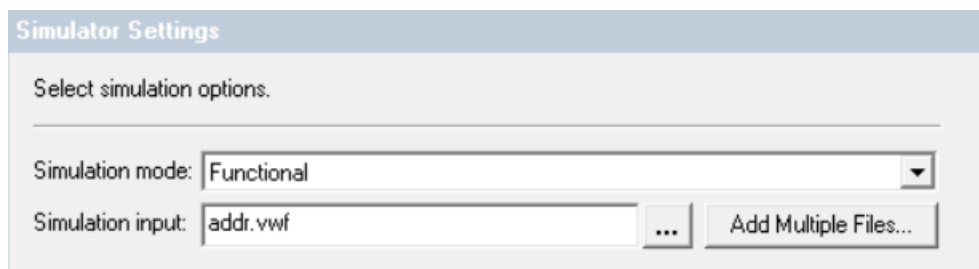


图 4-20 功能仿真模式设置

(2) 选择 Quartus II 主窗口 Processing 菜单下的 Generate Functional

Simulation Netlist 命令，生成功能仿真网表文件；

(3) 选择 Quartus II 主窗口 Processing 菜单下的 Start Simulation 进行功能仿真。

功能仿真满足要求后，还要对设计进行时序仿真，时序仿真可以在编译后直接进行，但是要将上图中的 Simulation mode 中设置为”Timing”，设置好以后直接选择 Start Simulation 命令，执行时序仿真。

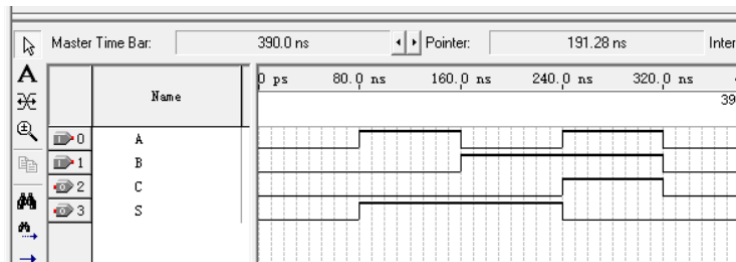


图 4-21 半加器仿真结果

4.3.2 modelsim 仿真验证

Quartus II 9.0 及以前版本集成了仿真功能，但是后面版本不集成仿真功能，因此需要掌握 modelsim 软件仿真技巧。

Quartus II 9.0 版本自带的 Simulation 仿真与 Modelsim 仿真的优缺点比较。

1) Quartus II 自带的 Simulation 比较适合初学者使用，纯粹的 GUI，用鼠标设置即可进行仿真，非常简洁明了。但是相对于比较大的工程，比较复杂的项目，纯粹的手工，死板的设置，或许在设计者面前便是束手无策了，这个时候，无奈之下，移情别恋，开始 Modelsim 仿真。

2) 相对于 Quartus II 自带的 Simulation 而言，Modelsim Simulation 虽然也能简单的进行 GUI 设置激励，但是更强大的在于 testbench 测试语言，来对逻辑设计进行仿真测试。对于强大的项目，往往是测试占去了大部分时间，如若还用 Quartus II 自带的 Simulation，往往得不到满意的结果。Modelsim 仿真复杂的同时意味着强大，Modelsim Simulation 更适合项目的开发，能熟练运用 Modelsim 必然对 fpga 开发项目带来事半功倍的效益。

1、Modelsim_Altera ase 软件安装

Modelsim 软件一般分两个，ase 版本是 altera start edition，即入门版，免费使用的；ae 是 altera edition，需要破解，支持更多功能吧。

但对于一般设计，ase 版本已经足够了。装上就能使用。就不费破解的事了。

如想安装 ae 版本，请参考 Bingo 写过的教程，网页地址如下：

<http://www.cnblogs.com/crazybingo/archive/2011/02/21/1959893.html>

此处以安装 11.0_Altera_Modelsim_ase_windows.exe 为例，其他版本基本一

样操作。具体步骤如下：

(1) 打开安装目录下的 `setup`，一路 `next`，直到选择路径的时候，选择与 `quartus ii` 安装目录相同的路径。如下图所示，我的电脑上装在 E 盘上了，您要根据您的设置来改。

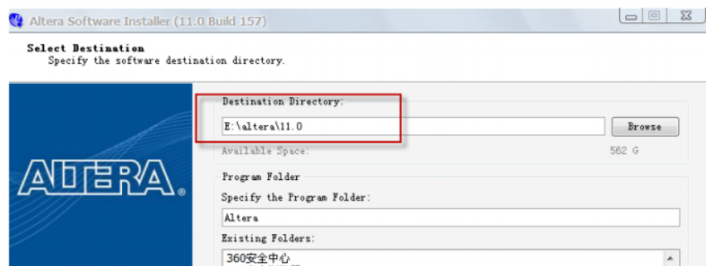


图 4-22 选择安装路径

(2) 继续 `next`，静默，等待安装完毕.....

(3) 安装完毕，出现以下界面：



图 4-23 完成安装

(4) OK, Finish!

(5) 打开 Quartus II，打开菜单 `Tool-Options`，在 `EDA Tool Options` 中的 `Modelsim-Altera`，选择 `Moldelsim-Altera` 应用程序的根目录，配置 `Modelsim-Altera` 应用程序第三方软件路径。如下图所示：在该选项卡中下面的 `ModelSim - Altera` 一项指定安装路径为 `E:/Altera/11.0/modelsim_ase/win32aloem`（其中 `E:/Altera/11.0/modelsim_ase/` 为我电脑中 `ModelSim - Altera 6.5e` 的安装路径）

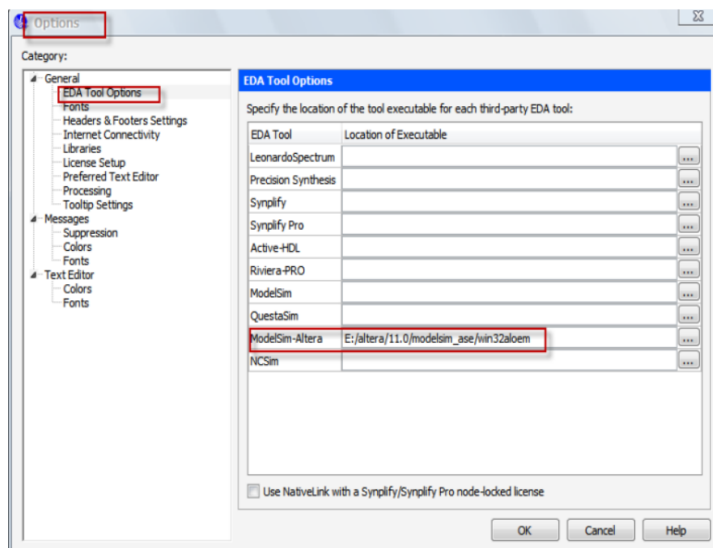


图 4-24 设置第三方软件路径

至此，Altera-Modelsim ase 版本安装完毕。

2、Testbench 的编写

在进行用 ModelSim 进行功能仿真时，需要编写 Testbench 测试文件。由于篇幅的限制，这里只做简要的介绍。

Testbench，其实就是测试平台的意思，一般 Testbench 的结构如下：

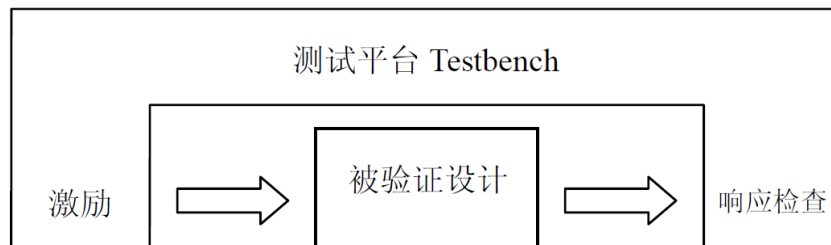


图 4-25 Testbench 的结构图

Testbench 的编写一般使用行为级语法，因为它不需要被综合。而且一般使用元件例化的方法将被验证设计例化至 Testbench 文件中。下面结合 DDS 正弦波发生器的设计列举一个 testbench 的实例。

首先，要得到“被验证设计”文件。如果设计的 DDS 正弦波发生器的顶层文件是一个用原理图方式描述的文件（格式为.bdf），则可以点击 file → Create/Update → **Create HDL Design File for Current File**，得到 verilog 描述方式的顶层文件，因为 testbench 只支持 verilog 描述设计，代码如下：

```

module DDS_generator (inclk, reset, enable, control_word, DDS_DATA);
input inclk;
input reset;
input enable;
input [5:0] control_word;
output [11:0] DDS_DATA;
wire sys_clk;
wire [11:0] SYNTHESIZED_WIRE_0;
wire SYNTHESIZED_WIRE_1;
altpll0 Ualtpll0 (
.inclk0(inclk),
.reset(reset),
.c0(sys_clk));
lpm_rom0 Ulpm_rom0 (
.clock(sys_clk),

```

```

.clken(enable),
.aclr(reset),
.address(SYNTHESIZED_WIRE_0),
.q(DDS_DATA));
assign SYNTHESIZED_WIRE_1 = ~reset;
address_gen Uaddress_gen (
.clk(sys_clk),
.reset(SYNTHESIZED_WIRE_1),
.enable(enable),.control_word(control_word),
.address(SYNTHESIZED_WIRE_0));
endmodule

```

然后设计 testbench，testbench 包括两部分，一是激励，二是被验证设计的元件例化。对于DDS 正弦波发生器，我们共需要时钟inclk，复位reset，使能enable 和频率控制字 control_word 这四个激励源。那么具体的testbench 设计如下：

```

`timescale 1ns/10ps
module Testbench ;
parameter CLK_PERIOD=20 ;
reg clk ;
reg rst ;
reg enable;
reg word;
reg dds_dataout;

initial //时钟激励 （产生周期为20ns，即频率为50MHz 的时钟）
begin
    clk = 0;
    forever begin
        # (CLK_PERIOD/2) clk = ~ clk ;
    end
end

initial //Reset 激励 （产生 5 个周期长度低电平的 reset 信号 ）
begin

```



```

    rst = 1;
    # CLK_PERIOD rst = 0;
    # (5*CLK_PERIOD) rst = 1;
end

initial //enable 激励（等待复位操作结束后，对 enable 信号置位）
begin
    enable=0;
    # (3*CLK_PERIOD) enable=0;
    enable=1;
end

initial //频率控制字激励（先将频率控制字设置为1，在1000 个周期后为3）
begin
    word = 6'b1;
    # (1000*CLK_PERIOD) word = 6'b3;
end

$display ("Simulation Finished!"); // 显示"Simulation Finished!"
$stop; //停止仿真

DDS_generator UDDS_generator (.inclk(clk), // 被验证设计的元件例化
.reset(rst),
.enable(enable),
.control_word(word),
.DDS_DATA(dds_dataout));
endmodule

```

完成后就可以将这两个文件 DDS_generator 和 Testbench 添加至 ModelSim 中进行功能仿真。

涉及到 testbench 的更多编写方法和技巧，可以参加“黑金动力社区”所编写的 testbench 编写教程，大家可以参考（FPGA 那些事儿--Modelsim 仿真技巧 REV6.0）。

3、在 Quartus II 11.0 中调用 ModelSim-Altera

下面以一个简单的实例来描述整个过程：

(1) 打开 QuartusII, 菜单 file---new, 新建一个工程。

(2) 新建一个 verilog HDL File, 代码如下：

```
module modelsim_test(clk,rst_n,div);
input clk;
input rst_n;
output div;
reg div;
always@(posedge clk or negedge rst_n)
if(!rst_n)div<=1'b0;
else div<=~div;
Endmodule
```

这是很简单的一个代码，是实现二分频电路。下面将对这个电路进行 Modelsim 仿真。

(3) 设置仿真相关选项：

在 Quartus II 11.0 界面菜单栏中选择 Assignments ->Settings 。

选中该界面下 EDA Tool settings 中的 Simulation 一项；

Tool name 中选择 ModelSim - Altera ；

Format for output netlist 中选择开发语言的类型 Verilog 或者 VHDL 等；

Time scale 指定时间单位精度级别；

Output directory 指定测试文件模板的输出路径（该路径是工程文件的相对路径）。

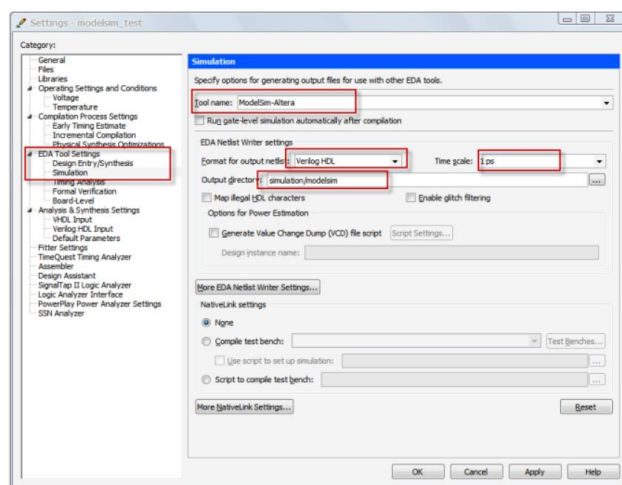


图 4-26 设置仿真选项

(4) 生成仿真测试文件

选择点击 Quartus II 11.0 开发界面菜单栏下 Processing ->Start ->Start Test Bench Template Writer, 提示生成成功。

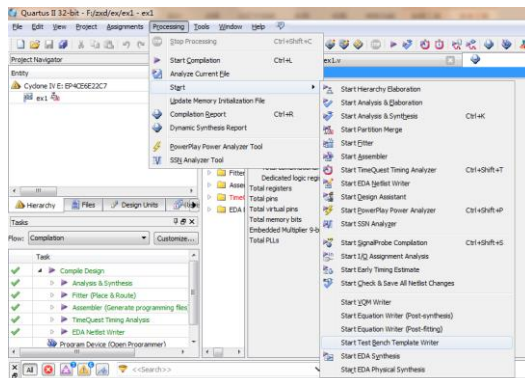


图 4-27 输出测试模块文件菜单

生成的仿真测试文件(modelsim_test 工程文件下 modelsim 目录下找到后缀名为".vt"的文件), 根据自己需要进行编辑修改。下面是生成的文件原样, 还没修改:

```

timescale 1 ps/ 1 ps
module modelsim_test_vlg_tst()
// constants
// general purpose registers
reg eachvec;
// test vector input registers
reg clk;
reg rst_n;
// wires
wire div;

// assign statements (if any)
modelsim_test_i1 (
    .clk(clk),
    .div(div),
    .rst_n(rst_n)
);

initial
begin
    // code that executes only once
    // insert code here --> begin
    // --> end
    $display("Running testbench");
end
always
// optional sensitivity list
// @(event1 or event2 or .... events)
begin
    // code executes for every event on sensitivity list
    // insert code here --> begin
    @eachvec;
    // --> end
end
endmodule

```

图 4-28 仿真测试文件截图

注意: QuartusII 中 testbench 文件的后缀是.vt, 产生的模板文件只是包含了端口映射, 端口声明等, 具体的功能还是需要设计者自己编写, 下面我们在模板上修改, 修改编写分频器的 testbench:

代码如下:

```

timescale 1 ps/ 1 ps
module modelsim_test_vlg_tst();

reg clk;

reg rst_n;

wire div;

```

```

modelsim_test i1(    //例化被测模块
    .clk(clk),
    .div(div),
    .rst_n(rst_n)
);
initial
begin
    clk=0;
    forever
        #100 clk=~clk;    //产生时钟信号
    end
initial
begin
    rst_n=1;
    #1000 rst_n=0; //复位信号
    #1000; rst_n=1;
    $stop;        //停止仿真
end
endmodule

```

测试代码代码产生时钟信号、复位信号，例化被测模块。

(5) 设置测试文件的链接关系

在 Quartus II 11.0 界面菜单栏中选择 Assignments ->Settings。

选中该界面下 EDA Tool settings 中的 Simulation 一项；在 NativeLink settings 中选择 Compile test bench 并点击后面的 Test Benches 按钮。

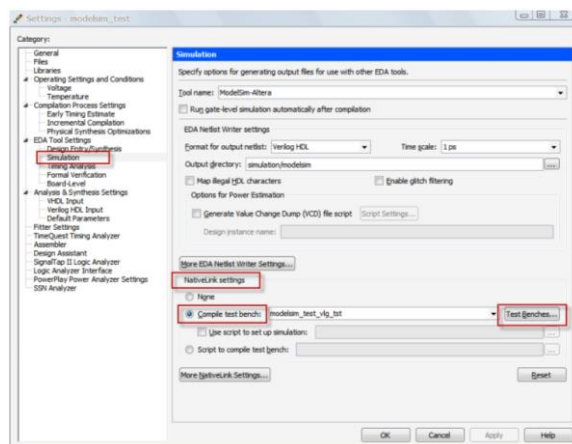


图 4-29 设置测试文件的链接关系

然后在 Test Benches 中点击 New ；

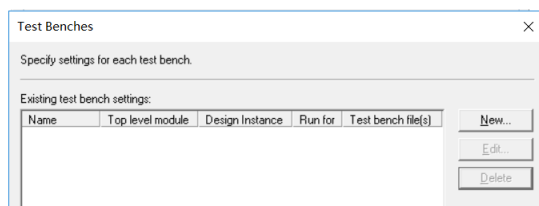


图 4-30 新建 Test Benches

在弹出的对话框中 **Test bench name** 栏要填写刚刚我们创建的 **testbench** 文件的实体名，即 **modelsim_test_vlg_tst**；

在 **Top level module in test bench** 中也填写 **modelsim_test_vlg_tst**；

在 **Design instance name in test bench** 中填写 **i1**，即被测模块例化名称；
这写名称可以直接从 testbench 的文件里直接复制过来，避免手误写错。

可设置仿真结束时间，**End simulation at**。

然后在 **Test bench files** 选项中浏览添加 testbench 文件，然后记住点击 **add** 添加按钮，一步一步进行即可设置完成，如图 4-31 所示。

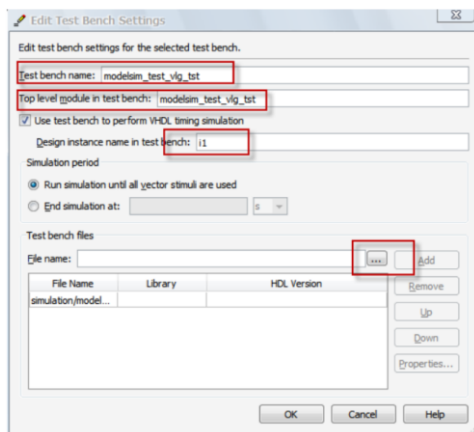


图 4-31 测试平台参数设置

(6) 一切准备就绪，在 Quartus II 11.0 界面菜单栏中选择菜单栏 Tools 中的 Run EDA Simulation Tool ->EDA RTL Simulation 进行行为级仿真，接下来就可以看到弹出 ModelSim - Altera 6.5e 的运行界面，放大观察仿真波形，可以看到 div 信号是 clk 的二分频了。

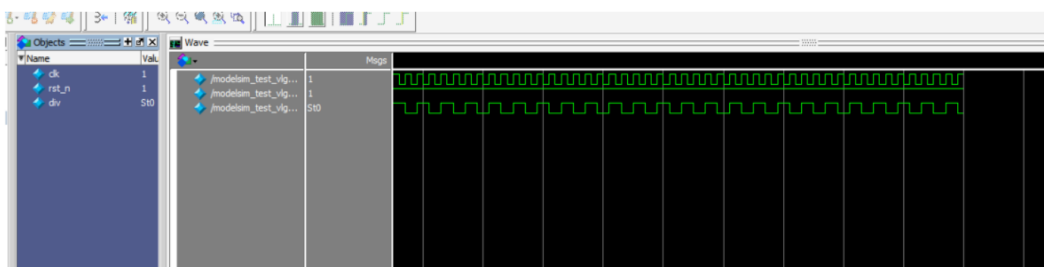


图 4-32 modelSim 仿真波形

4.4. 引脚锁定

引脚锁定是为了对所设计的工程进行硬件测试，将输入/输出信号锁定在器件的某些引脚上。单击“Assignments”菜单下的“Pins”命令，在弹出的对话框如下图所示，在下方的列表中列出了本项目所有的输入/输出引脚名。

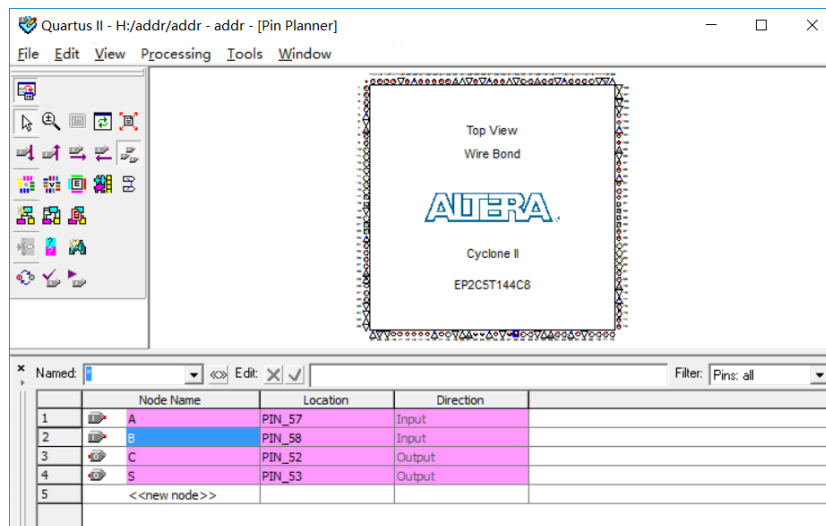


图 4-33 引脚锁定图

在上图中，双击与输入端“A”对应的“Location”选项后弹出引脚列表，从中选择合适的引脚号，则输入“A”的引脚锁定完毕（开发板上对应 PIN_57）。同理完成其它引脚的锁定，如图所示。锁定完管脚后整个工程还需要编译一遍才能下载调试。

4.5. 下载调试验证

对设计进行验证后，即可对目标器件进行编程和配置，下载设计文件到硬件中进行硬件验证。

Quartus II 编程器 Programmer 最常用的编程模式是 JTAG 模式和主动串行编程模式 AS。JTAG 模式主要用在调试阶段，主动串行编程模式用于板级调试无误后将用户程序固化在串行配置芯片 EPCS 中。

1、JTAG 编程下载模式（SOF 文件）

此方式的操作步骤主要分为以下步骤：

选择 Quartus II 主窗口的 Tools 菜单下的 Programmer 命令或点击  图标，进入器件编程和配置对话框。如果此对话框中的 Hardware Setup 后为“No Hardware”，则需要选择编程的硬件。点击 Hardware Setup，进入 Hardware Setup

对话框，下页图所示，在此添加硬件设备。

配置编程硬件后,选择下载模式,在 Mode 中指定的编程模式为 JTAG 模式;

确定编程模式后，单击  Add File... 添加相应的 addr.sof 编程文件，选中 addr.sof 文件后的 Program/Configure 选项，然后点击  Start 图标下载设计文件到器件中，Process 进度条中显示编程进度，编程下载完成后就可以进行目标芯片的硬件验证了。

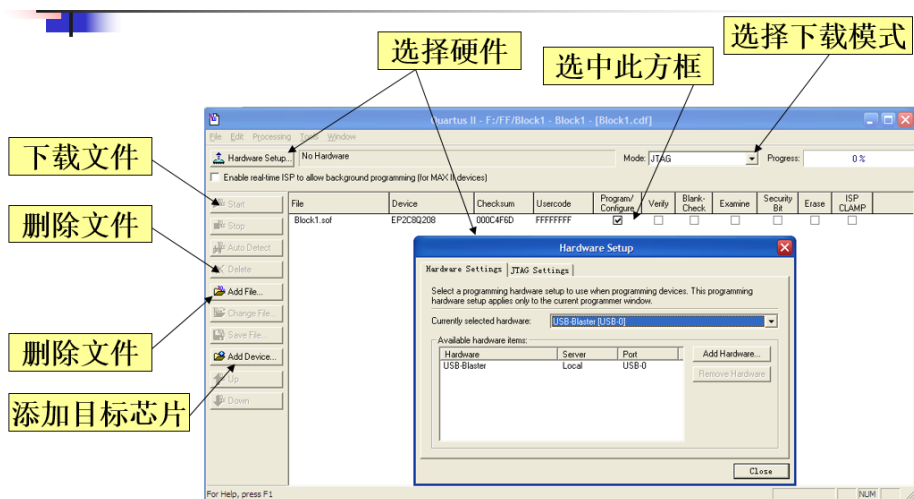


图 4-34 JTAG 编程下载模式

2、(JTAG 模式)下载 EPCS 器件的方法 (JIC 文件)

一般来说 Altera 公司 Cyclone 或者 CycloneII 系列 FPGA 相应的配置器件会选择 EPCS 系列串行 FLASH。一般使用 AS 模式下载 EPCS 系列器件。但有时候可能遇到 AS 模式不能成功下载的案例，原理图以及 PCB 板都是按照推荐电路设计，这时候我们可以通过 Flash Loader 检验 EPCS 器件是否良好。即通过 JTAG 方式下载 EPCS 系列器件。如果这种方式还是不能进行正确的 AS 模式的下载的话，可能 EPCS 器件本身已经不能正常工作了，此时可以建议客户更换配置芯片了:)。

使用 Flash Loader(JTAG 下载模式)下载 EPCS 器件的步骤如下:

- (1) 把需要下载的工程文件生成 `xx.sof`。
- (2) 点击 **File** 菜单，选择 **Convert Programming Files...**，进入文件转换操作对话框，把 **SOF** 文件转换为 **JIC** 文件输出，如下图所示。

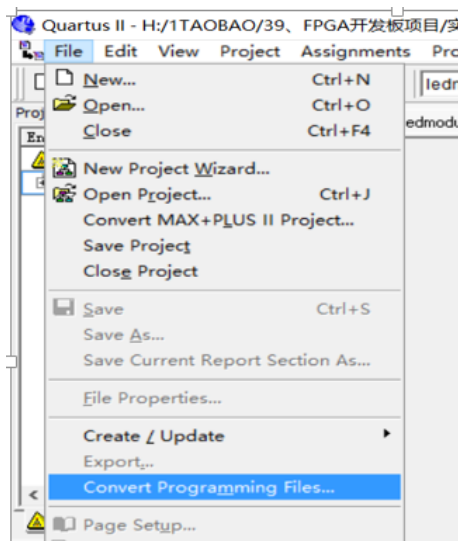


图 4-35 文件转换菜单

(3) 在执行完第 2 步后跳出图三所示的界面。鼠标左键点击 Programming file type 栏目的下拉箭头，选择 JTAG Indirect configuration File (.jic) 项。

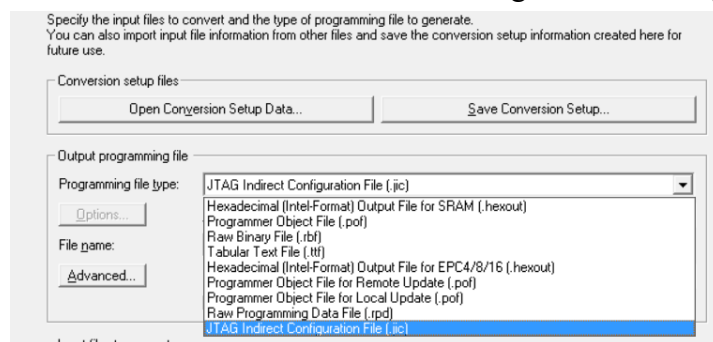


图 4-36 选择转换文件格式

(4) 配置器件选择 EPCS4，根据电路板上实际器件型号选择。

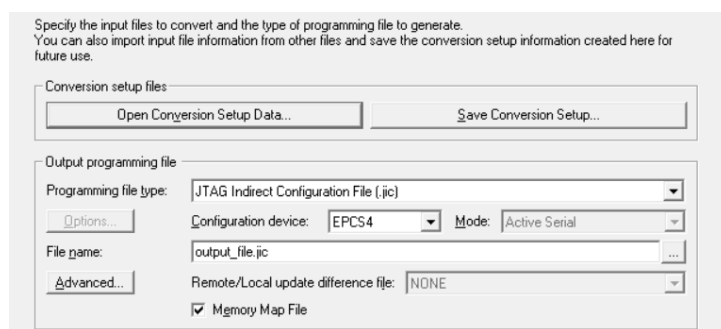


图 4-37 选择配置器件

(5) 完成步骤 3 后选中 Input files to convert 栏目中的 SOF Data，左键点击 Add File 按钮。找到生成的相应的 sof 文件并打开，此时相应的 sof 下载文件出现在相应的界面中，添加进来。

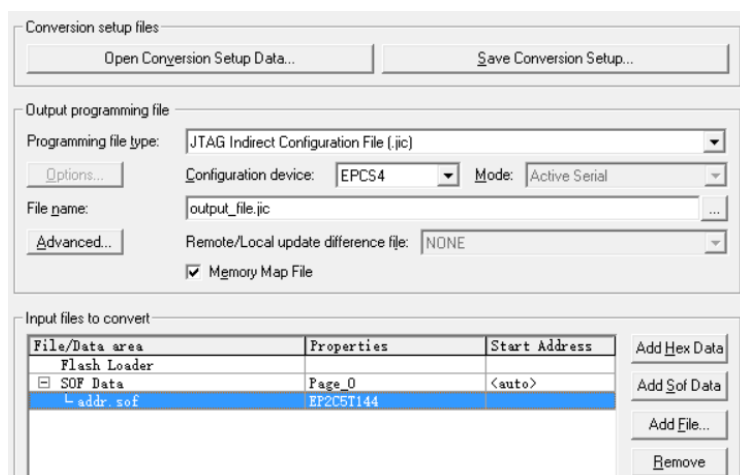


图 4-38 选择 SOF 文件

(6) 选中 Flash Loader，左键点击 Add Device...按钮，在跳出的 Select Devices 界面下选择相应的器件。（这里以 EP2C5 为例）点击 ok，器件选择完毕。

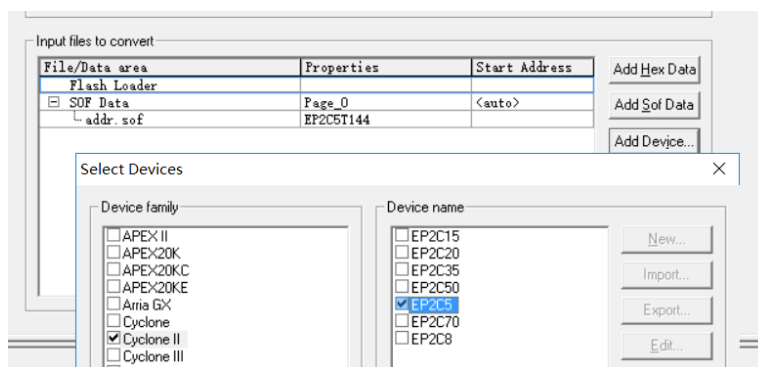


图 4-39 选择 FPGA 芯片型号

(7) 完成以上步骤后就可以转换生成相应的下载文件了。点击 Generate 按钮就可生成 JIC 格式文件。

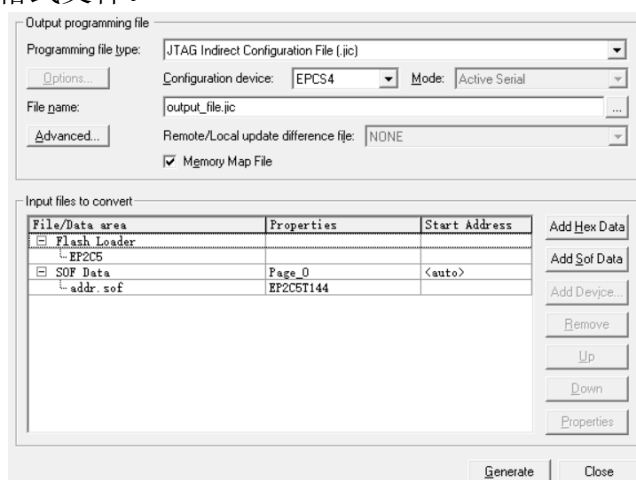


图 4-40 完成设置点击 Generate 按钮

(8) 下载步骤与我们普通的 JTAG 下载过程基本一致了，但需要注意的是这里下载文件必须选择所生成的 jic 文件。打开 Programmer 窗口，按 Add File 按钮找到并打开生成的 jic 文件。最后左键点击 Start 按钮大功告成！

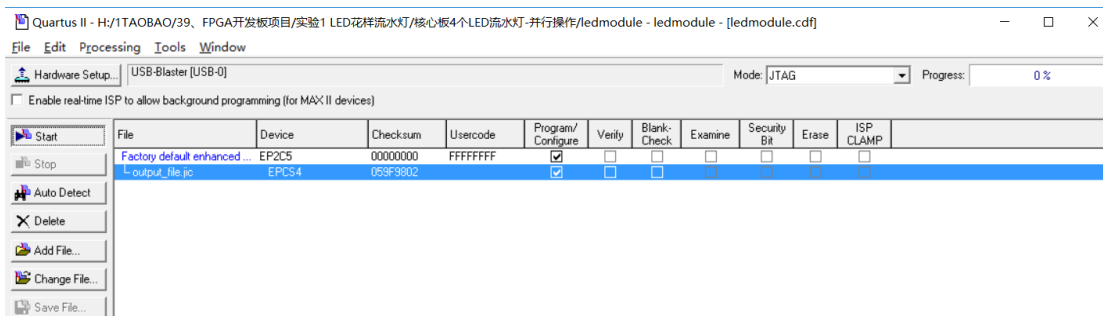


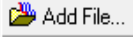
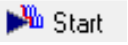
图 4-41 添加 jic 文件下载

3、AS 主动串行编程模式

AS 主动串行编程式的操作步骤如下：

(1) 选择 Quartus II 主窗口 Assignments 菜单下 Device 命令，进入 Settings 对话框的 Device 页面进行设置，见下页图所示。

(2) 选择 Quartus II 主窗口的 Tools 菜单下的 Programmer 命令或点击图标，进入器件编程和配置对话框，添加硬件，选择编程模式为 Active Serial Program；

(3) 单击添加相应的 addr.pof 编程文件，选中文件后的 Program/Configure、Verify 和 Blank Check 项，单击图标下载设计文件到器件中，Process 进度条中显示编程进度。下载完成后程序固化在 EPCS4 中，开发板上电后 EPCS 将自动完成对目标芯片的配置，无须再从计算机上下载程序。

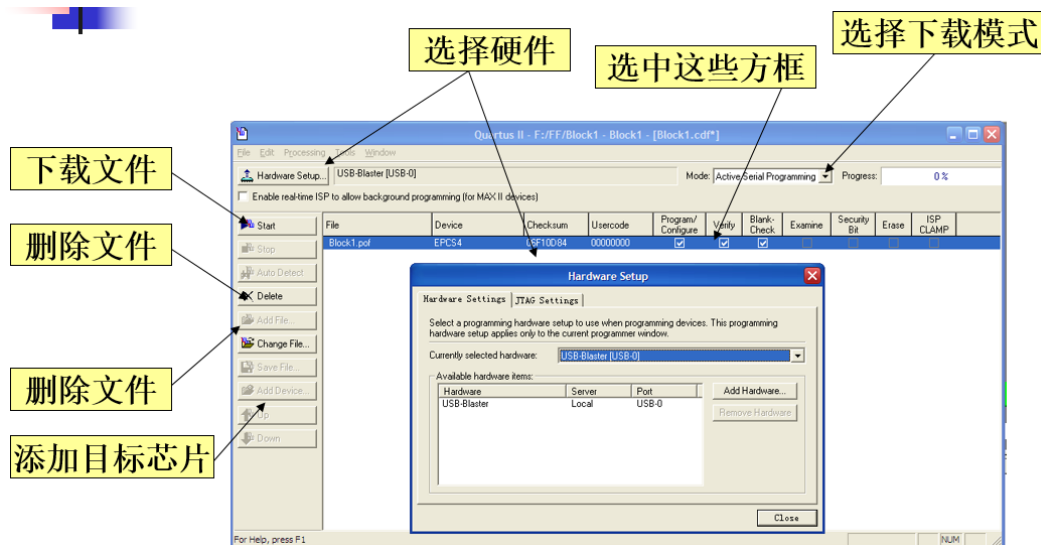


图 4-42 AS 主动串行编程模式

注意：JTAG 和 AS 接口不能带电拔插，不然很容易烧坏配置芯片!!!

第五章 FPGA 开发板

欢迎进入 FPGA 的学习殿堂，也祝贺你踏出学习 EDA 的第一步。在这里我们的目标是零基础也能学会 FPGA 技术，让你领略 FPGA 设计和并行设计的魅力。为了更好轻松入门实践操作，本书配套开发了一块 FPGA 开发板，采用 Cyclone II 系列高性价比的 FPGA 芯片，配置了丰富是硬件资源，其中基础资源有：LED、数码管、蜂鸣器、按键、串口等，进阶实验的资源有：LCD、IIC-AT24C02、RTC 芯片、数模转换芯片 TLC5615、模数转换芯片 TLC549、温度传感器、步进电机接口、超声波接口、红外接口、VGA 接口。为方便进一步方便扩展使用，开发板将所有 IO 引出，可以与其他外部模块连接使用。为了让你更好的学习，本书配套开发了由简单到复杂的相关的例程和资料，让你可以掌握在 FPGA 上进行任何数字电路的设计。相信你在掌握对电路板上的硬件资源编程设计后，就能领略 FPGA 的设计思想，熟练掌握硬件描述语言的编程精髓。

5.1 FPGA 开发板简介

FPGA 开发板实物图如图 5.1 所示：

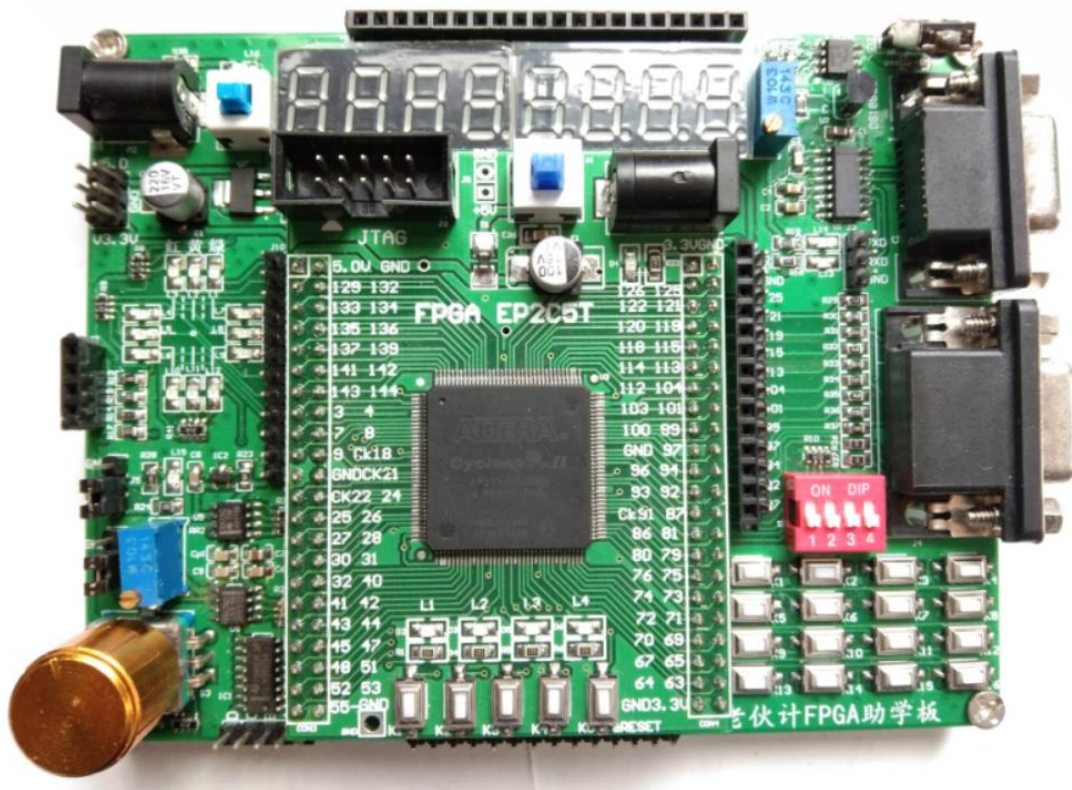


图 5.1 FPGA 开发板实物图

目前有众多朋友热切的期望能学习 FPGA 知识, FPGA 技术实践性极强, 拥有一块适合学习需求的 FPGA 开发板是非常有必要的! 本书配套推出的 FPGA 开发板由笔者精心设计开发, 采用 Altera 公司的 CycloneII 系列芯片 EP2C5T144C8、或者 Cyclone IV 系列芯片 EP4CE6E22C8N 作为核心处理器进行设计, CycloneII 系列芯片可以说是目前市场上性价比较高的芯片, 比第一代芯片设计上、内部的逻辑资源上都有很大的改进, 同时价格也可以被广大用户接受。如图 5.2 和 5.3 所示为开发板资源配置和布局图。

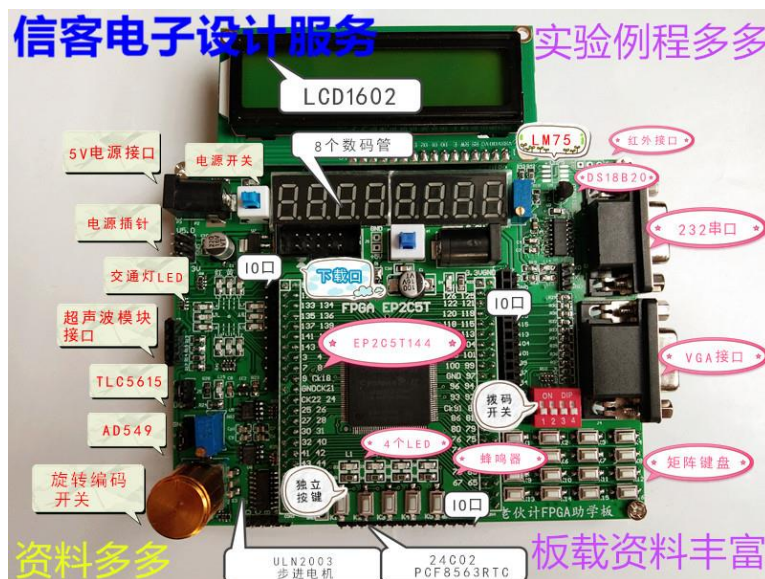


图 5.2FPGA 开发板资源配置位置图

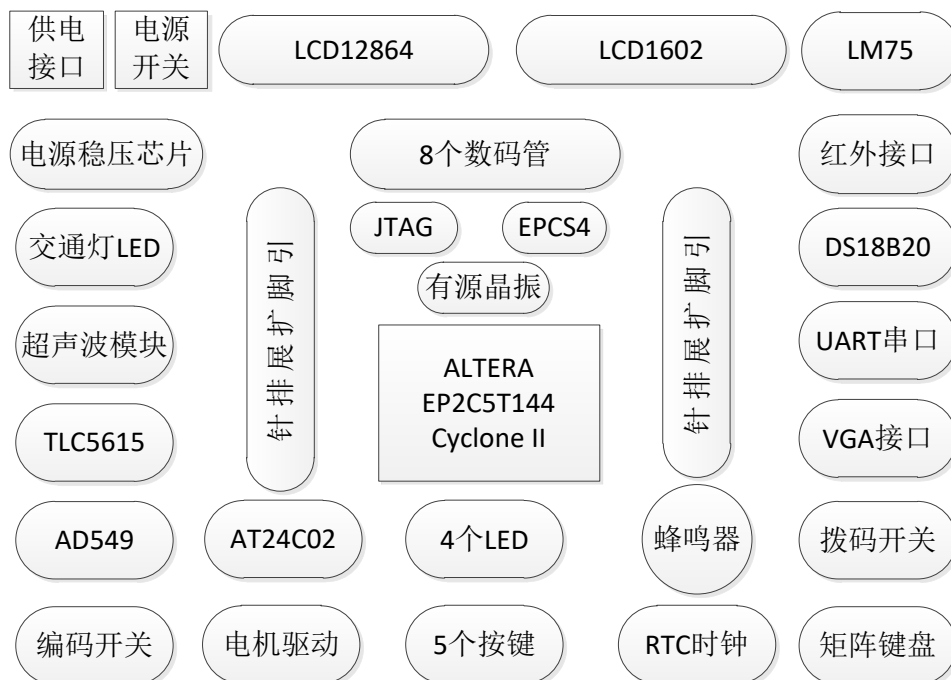


图 5.3 FPGA 开发板资源布局图

5.2 开发板硬件详细配置

FPGA 开发板的硬件资源配置如下表 5-1 所示。

表 5-1 开发板硬件资源配置表

序号	硬件资源名称	说明
1	Cyclone II FPGA	型号是 EP2C5T144C8
2	交通灯模式 LED 灯	底板 12 个红色高亮 LED 灯，
3	核心板上 4 个 LED	红色高亮 LED 灯
4	EPCS4	用于 JTAG 模式下的程序烧录
5	JTAG 下载接口	用于 JTAG 方式下载接口
6	IIC-AT24C02	用 IIC 驱动 EEPROM
7	实时时钟 PCF8563	实时时钟芯片,可换成 DS1302 时钟芯片
8	DAC-TLC5615	10 位 14M 串行数模转换芯片
9	ADC-TLC549	8 位 4M 串行模数转换芯片
10	ULN-2003	步进电机驱动芯片
11	DS18B20	温度传感器 1
12	LM75	温度传感器 2
13	红外遥控接收	作为输入设备
14	核心板 5 个独立按键	根据需要设定按键功能
15	4X4 矩阵键盘	输入设备
16	4 位拨码开关	高低电平输入设备
17	旋转编码开关	作为输入设备，方便调节相关设计参
18	超声波模块接口	接超声波模块
19	蜂鸣器	用户自定义，根据编程可以发出各种
20	VGA 显示接口	用来驱动液晶屏
21	RS232 串口	用来和 PC 通讯
22	8 位共阳数码管	显示数字
23	LCD1602/12864 接口	LCD1602 和 12864 的共用接口

FPGA 开发板采用核心板与接口板分离的方式，核心板上除 FPGA、各类存储器以及用户扩展接口外，还有按键、LED 及电源插座等。因为有用户扩展接口，核心板完全可以脱离接口板而单独使用，通过接口，用于实验设计或电子设计大赛，扩展性极好。接口板上集成了最常用和经典外围电路，所有的外设经过精心分配及设计，不需要进行任何跳线设置，实验时非常方便！此外，对于重点

外设的关键信号都设置了测试点，方便用户使用逻辑分析仪、示波器、万用表等进行信号测量。总之，该款 FPGA 开发板是完全站在用户的角度精心设计开发，简约不简单！同时为读者配套提供相当丰富的实验代码及各种参考文档。

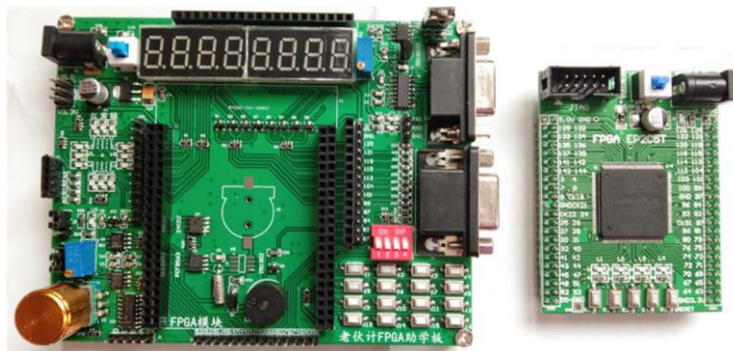


图 5.4 开发板底板与核心板

FPGA 核心板硬件配置如下：

- 1.板载 Altera 公司高性价比 FPGA：CycloneII 系列 EP2C5T144 芯片；
- 2.配置 EPROM 芯片采用 EPCS4，大小为 4Mbit；
- 3.板载 50M 有源贴片晶振(晶振在板子的背面)；
- 4.电源采用大口电源插座，单 5V 供电，有自锁按键开关，操作方便；
- 5.板上有电源指示灯和复位开关；
- 6.板载 4 个贴片 LED，可以做 LED 的测试实验，更多实验利用引线完成；
- 7.板载 4 个贴片 按键，按键按下低电平；
- 8.所有 IO 口和时钟引脚通过排针引出，通过这些接口可以扩展任何存储器和外设；
- 9.采用 1117 - 1.2V 稳压芯片，提供 FPGA 内核电压；采用 1117 - 3.3V 稳压芯片，提供 FPGA IO 口电压；板卡采用高品质钽电容做电源滤波；
- 10.JTAG 下载接口，对应下载的文件是 SOF 文件，速度快，JTAG 将程序直接下载到 FPGA 中，但是掉电程序丢失，平时学习推荐使用 JTAG 方式，最后固化程序的时候再通过 JTAG 下载接口下载 IIC 格式文件。
- 11.支持 NiosII 嵌入式 CPU 的开发。
- 12.所有输入输出精心分配，使用 2 个扩展接口双排插针，通用 2.54mm 间距。

FPGA 开发板用户群体面向广大的高校相关专业学生、电子爱好者、科研单位、企事业单位的开发设计人员，适合于产品原型的快速开发、学生参加各种电子设计大赛、学习 FPGA 技术入门，课程设计以及毕业设计等，亦可用于系统设计前期快速评估设计方案。特别适合 FPGA、NiosII、SOPC 快速入门和产品

开发及验证。淘宝链接如下：<https://item.taobao.com/item.htm>。

5.3 开发板硬件原理图

开发板的硬件资源在表 5-1 资源配置表中列出了，下面对各电路原理进行详细介绍。

5.3.1 核心板硬件电路

1、FPGA 主芯片

核心板上的 FPGA 芯片采用的是 Altera 公司（已经被 Intel 收购）Cyclone II 系列的 EP2C5T144C8，此芯片资源丰富，价格适中，非常适合 FPGA 入门学习使用，它的资源如图：

Table 1-1. Cyclone II FPGA Family Features (Part 1 of 2)							
Feature	EP2C5 (2)	EP2C8 (2)	EP2C15 (1)	EP2C20 (2)	EP2C35	EP2C50	EP2C70
LEs	4,608	8,256	14,448	18,752	33,216	50,528	68,416
M4K RAM blocks (4 Kbits plus 512 parity bits)	26	36	52	52	105	129	250
Total RAM bits	119,808	165,888	239,616	239,616	483,840	594,432	1,152,000
Embedded multipliers (3)	13	18	26	26	35	86	150
PLLs	2	2	4	4	4	4	4

图 5-5 Cyclone II 系列资源图

Cyclone II FPGA 系列简介：

Altera® Cyclone® II 采用全铜层、低 K 值、1.2 伏 SRAM 工艺设计，裸片尺寸被尽可能最小的优化。采用 300 毫米晶圆，以 TSMC 成功的 90nm 工艺技术为基础，Cyclone II 器件提供了 4608 到 68416 个逻辑单元（LE），并具有一整套最佳的功能，包括嵌入式 18 比特 x18 比特乘法器、专用外部存储器接口电路、4kbit 嵌入式存储器块、锁相环（PLL）和高速差分 I/O 能力。

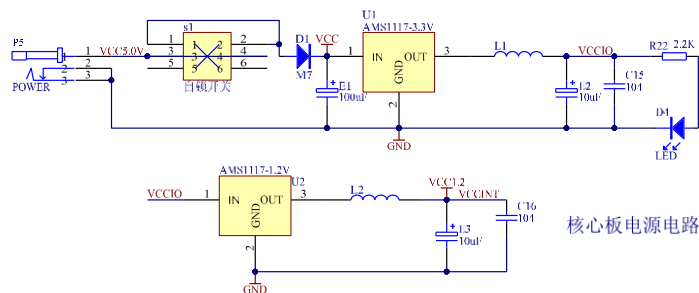
Cyclone II 器件扩展了 FPGA 在成本敏感性、大批量应用领域的影响力，延续了第一代 Cyclone 器件系列的成功。EP2C5T144C8N：“2”：Cyclone II 系列；“5”：逻辑单元 4608 个；“T”：封装类型；“144”：管脚数量；“8”：速度等级为 8。

通常，芯片的逻辑单元和 RAM 的数量是重要的参考指标，对于 EP2C5T144C8 来说，已经足够初学者使用了。核心板上 FPGA 主芯片的原理图如图 5-6 所示，把芯片的可用 IO 口都引出利用。



电源是保证整个开发系统正常工作最重要的部分，电源电路分两部分，一部分是底板电源，一部分是核心板电压。开发板底板外部输入 5V 电源，经过 AMS1117-3.3V 稳压后输出 3.3V 电压，提供给底板上的器件供电。核心板上也通过稳压块输出 3.3V 电压，主要用于给 FPGA 所有 IO 口，核心板存储器电路、串行配置器件、复位电路和 LED 等供电；3.3V 电压再经过 AMS1117-1.2V 稳压输出 1.2V 电压，给 FPGA 的内核和锁相环供电。同时各电源输入及输出部分设计了高质量钽电容用作电源滤波处理，提高系统电源稳定性。电源部分的原理图如图 5-5 所示。





(b)

图 5-7 底板和核心板电源电路

3. 核心板独立按键及 LED 电路

核心板上有 4 个由 IO 口控制的 LED 灯，位于核心板的下方。在单独使用核心板时，可以用来做流水灯等实验项目。核心板上有 5 个独立按键，K1-K4 可以作为操作按键，K5 作为系统复位按键，因为 K5 是接 Pin88 引脚是系统时钟专用输入引脚，作为系统复位输入比较合适，每个独立按键都配有上拉电阻，保证没有按键按下时输入信号是高电平的，电路如图 5-6 所示。

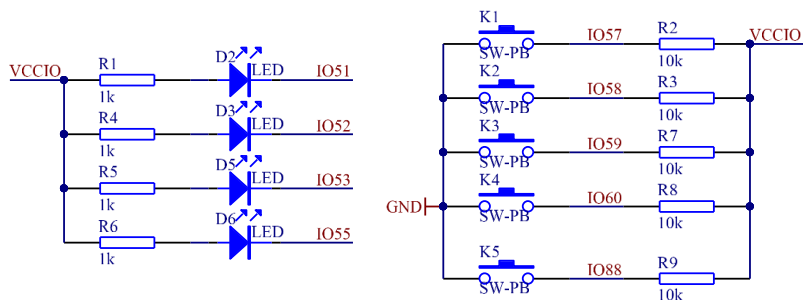


图 5-8 按键和 LED 灯电路

4. JTAG 下载和存储配置电路

电路包括 JTAG 下载接口设计及 EPCS4 存储配置芯片。JTAG 接口可以下载 SOF 文件到 FPGA，但是掉电数据会丢失，将 SOF 文件转换为 JIC 文件格式后可通过 JTAG 接口下载到 EPCS4 存储配置芯片中，掉电程序数据不会丢失，上电后数据从 EPCS4 恢复到 FPGA 中。下载配置相关电路如图 5-7 所示。

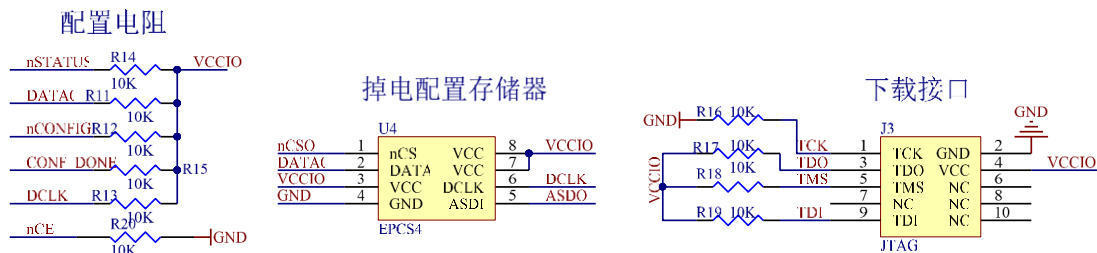


图 5-9 下载和存储配置电路

5. 系统时钟电路

开发板采用 50M 有源贴片晶体为 FPGA 提供运行时钟，时钟部分电路电源经过滤波处理，工作更加稳定可靠。时钟电路如图 5-8 所示，开发板中对 FPGA 时钟引脚处理如下：

- (1) CLK0 用作系统工作时钟，直接接入 50MHz 晶振；
- (2) CLK7 用作系统复位引脚，用户可以通过编程实现复位功能；
- (3) CLK1、CLK2、CLK3、CLK4 为用户输入引脚，引出到扩展座了，供用户使用；
- (4) CLK5，CLK6 未引出。

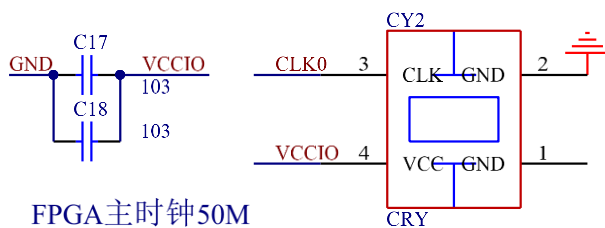


图 5-10 时钟电路

5. 核心板扩展 IO 口电路

开发板采用核心板与接口板分离的形式，用户可以通过核心板外接其他电路，既保证了通用性，又保证了实用性。核心板上的 FPGA IO 口通过 P1A，P1B 两个双排插针引出。插针设置了 VCC5V、3.3V、GND 电源接口，方便核心板与其他电路连接。扩展接口的原理图如图 5-9 所示。

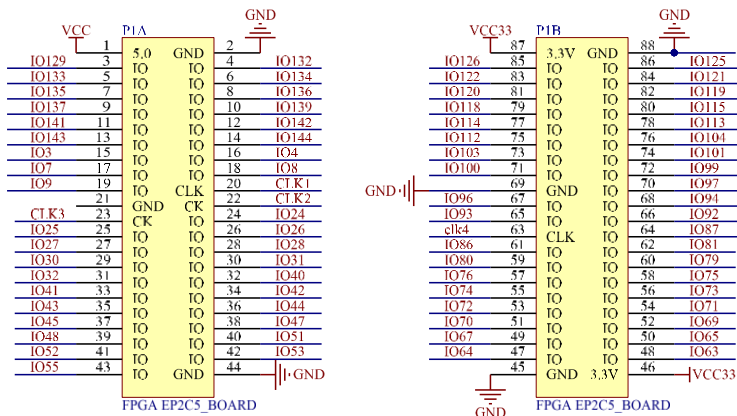


图 5-11 扩展接口电路

5.3.2 底板硬件电路

1. 底板电源电路

为方便使用，在接口底板上配置了 5V 电源插口，输入 5V 电压通过核心板与接口板的互联插口相连，由底板和核心板各自提供 3.3V。因此，核心板的电源

是可以单独使用的，接口板电源在没有核心板接入的情况下也可以单独使用。接口板电源部分如图 5-12 所示。

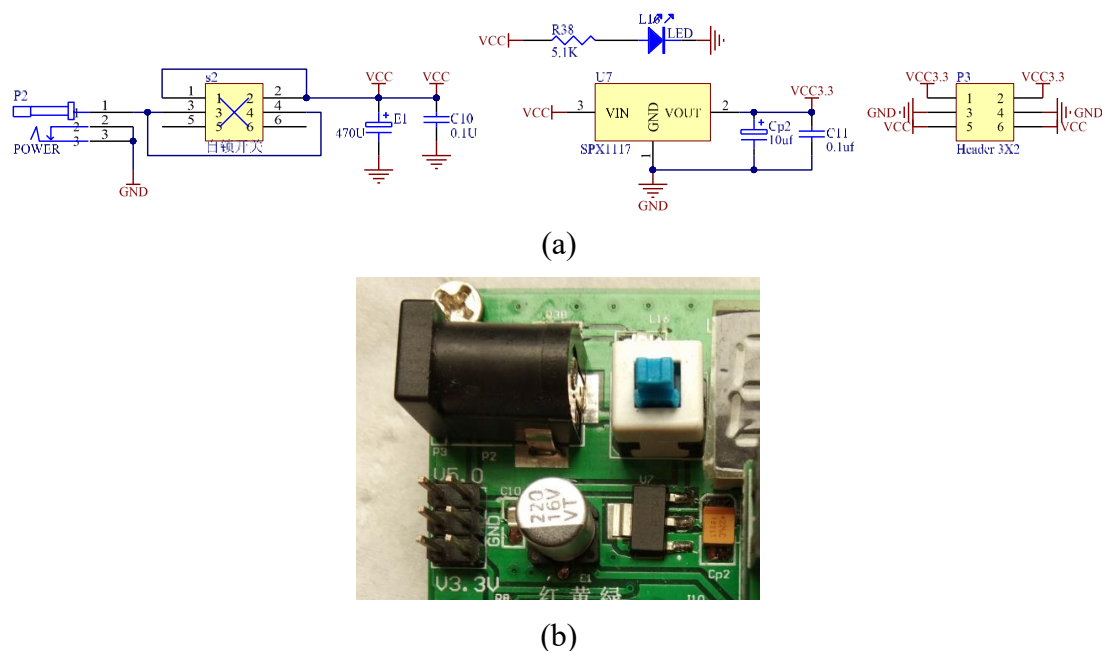


图 5-12 电源电路和实物图

2. 交通灯 LED 电路

接口底板上具有 12 个独立 LED 灯，电路如图 5-13 所示，在 PCB 电路板中把这 12 个 LED 布局成十字路口交通灯的形式，方便设计交通灯的实验，也方便设计流水灯及其他相关实验。

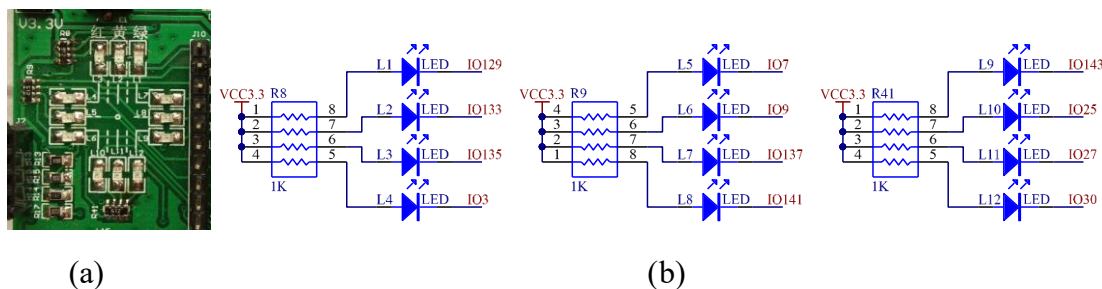


图 5-13 交通灯电路和实物图

3. 矩阵键盘电路

接口底板上具有 4X4 矩阵键盘，矩阵键盘在应用设计中经常需要用到，是常用的输入设备，比如设计密码锁，DDS 信号源等。电路图中的行信号 ROW0-ROW3 作为输入信号需要加上拉电阻，列信号 COL0-COL3 作为输出信号，电路如图 5-14 所示。

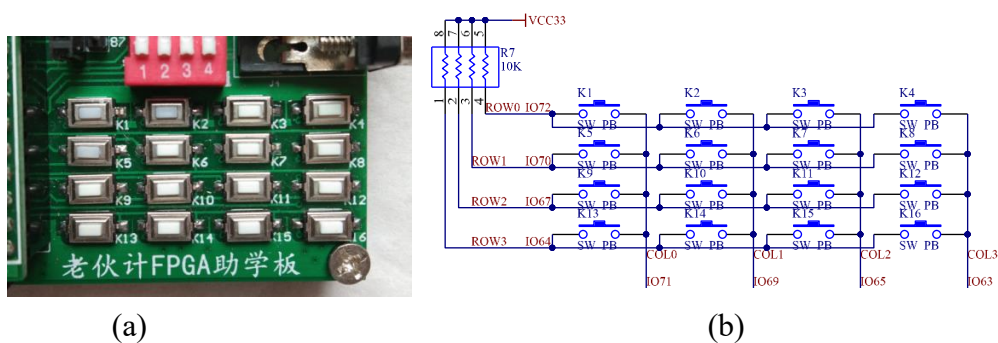


图 5-14 矩阵键盘电路和实物图

4. 拨码开关电路

拨码开关是常用的输入设备，4 位拨码开关电路相对简单，作为输入信号需要配置上拉电阻，如图 5-15 所示。

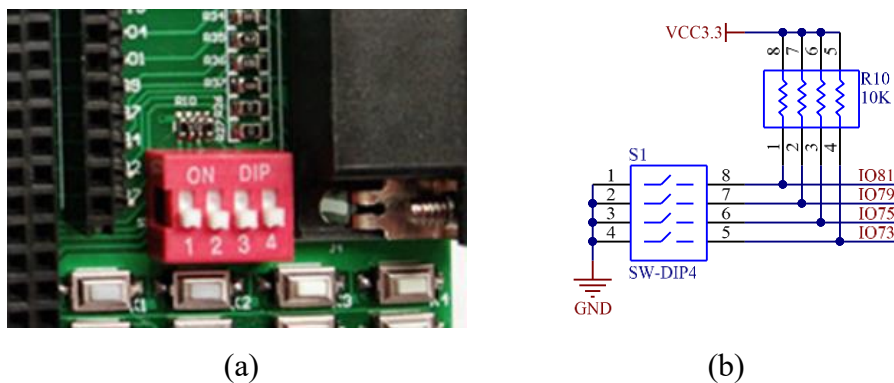


图 5-15 拨码开关电路和实物图

5. 蜂鸣器电路

蜂鸣器使用 PNP 三极管驱动控制，如果在基极输入一定频率脉冲时，蜂鸣器会发出声响，改变输入频率可以改变蜂鸣器的响声，可以设计电子琴、音乐播放器等实验。原理图如图 5-16 所示。

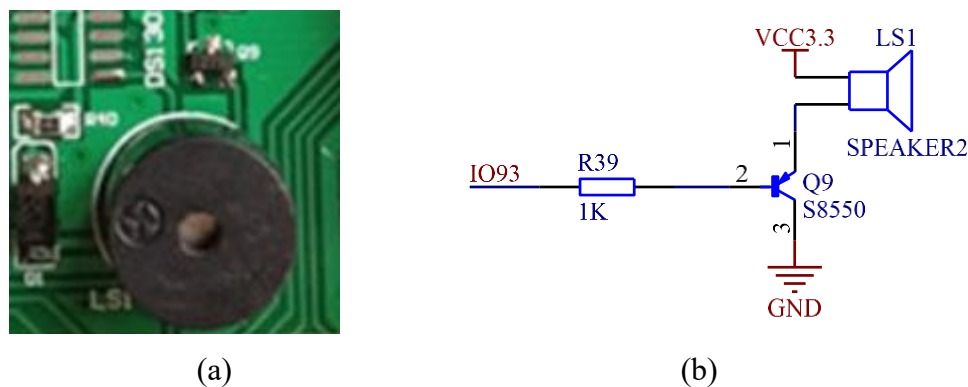


图 5-16 蜂鸣器电路和实物图

6. 数码管显示电路

数码管是作为常见显示器件，使用频率非常高，电路设计采用共阳数码管，当位显示驱动信号控制驱动三极管的基极为 0 低电平时，对应的数码管被点亮；当段显示驱动信号为 0 低电平时，对应的显示段被点亮。由于公共位选信号需要驱动电流较大，采用了 PNP 三极管 S8550 驱动。R1,R2,R3 作为保护限流电阻存在电路中。电路如图 5-17 所示。

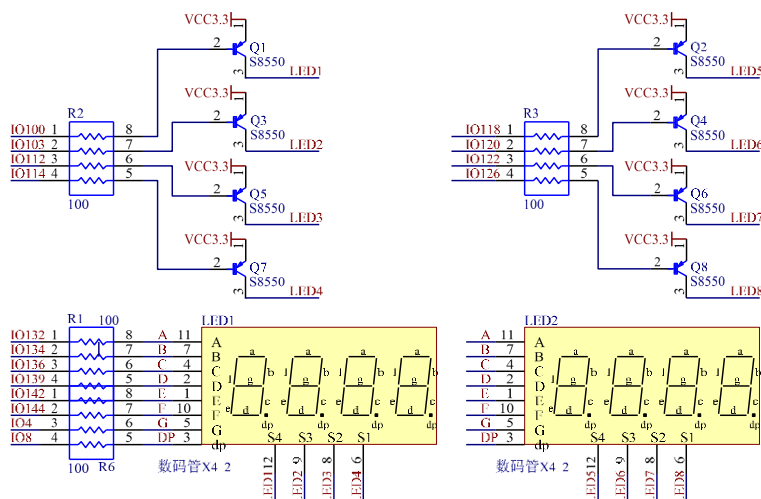


图 5-17 数码管显示电路

7. 液晶显示接口电路

接口底板上设有液晶 LCD1602 与 LCD12864 的接品电路，液晶模块是配置器件，如需要实验，可选配液晶器件，注意安装时，引脚的顺序。图 5-18 为液晶接口连线原理图，其中 1-16 管脚接口是接 LCD1602 液晶的，1-20 管脚接口是接 LCD12864 液晶的。RR1 可调电阻是调节液晶显示灰度的，R4、R5 是选择焊接 0 欧姆电阻，一般在 R4 焊接电阻，这样接 LCD1602 液晶时 15 脚作为背光的 VCC 输入，接 LCD12864 液晶时 15 脚作为并行接口或者串行接口的选择。

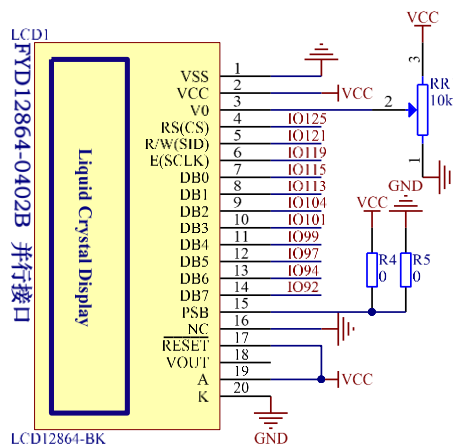


图 5-18 液晶显示接口电路

8.RS232 串口电路

RS232 串口电路如图 5-19 示, 由于是 FPGA 管脚是 3.3V 工作电压, 所以使用了 MAX3232 进行 RS232 电平转换, 同时, 还有两个 LED 用于指示串口的收发工作状态。DB9 接口是用于经过电平转换后与 PC 进行通信; J3 的 SIP3 接口是用于与其他控制器进行通信, 或者用于与 USB 转串口 TTL 电平的转换器通信, 不需要经过电平转换芯片。

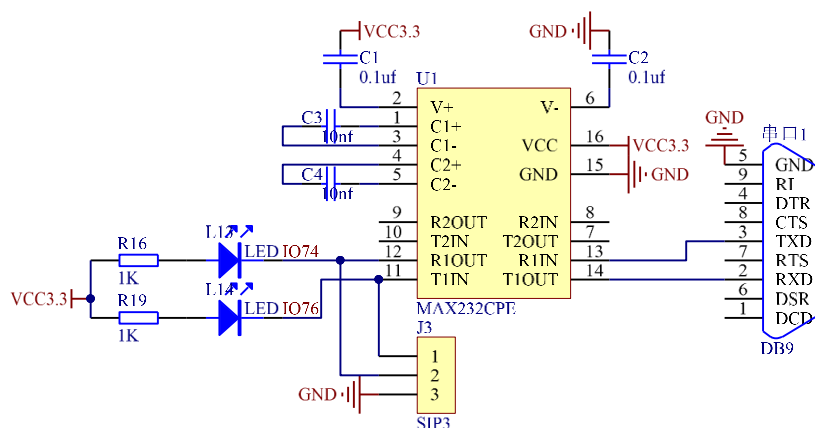


图 5-19 RS232 串口电路

9.VGA 接口电路

VGA 显示接口电路如图 5-20 所示, 本电路采用的是电阻分压网络的方法来产生 VGA 所需要的不同电压信号, 输入端共用了 9 根信号线, 可以产生 512 级颜色。

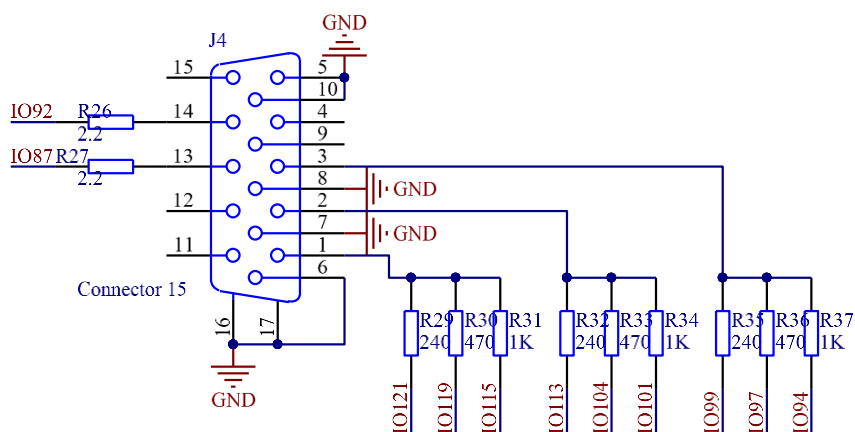


图 5-20 VGA 接口电路

10.串行 D/A、A/D 转换电路

数模转换电路采用串行接口的单通道 10 位 DA 转换芯片 TLC5615，TLC5615 具有半缓冲输出功能，可编程输出量程功能。DA 的参考电压从 REFIN 引脚输入，由 TL431 稳压输出 2.5V 基准电压，DA 转换输出的电压范围是 0-5V。

串行模数转换电路采用单通道 8 位 AD 转换芯片 TLC549C，转换所需的电压基准由 REF+ 输入，电压基准定为 5V，芯片可采集的电压范围是 0-5V。电路图中 RR2 是可调电阻，把 J6 的 1、2 脚用跳线帽短接到可调电阻可作为 AD 的采集电压来源，方便电压测量实验，把 J6 的 2、3 脚连接可从外部接入测量电压。

A/D、D/A 转换电路如图 5-21 所示。

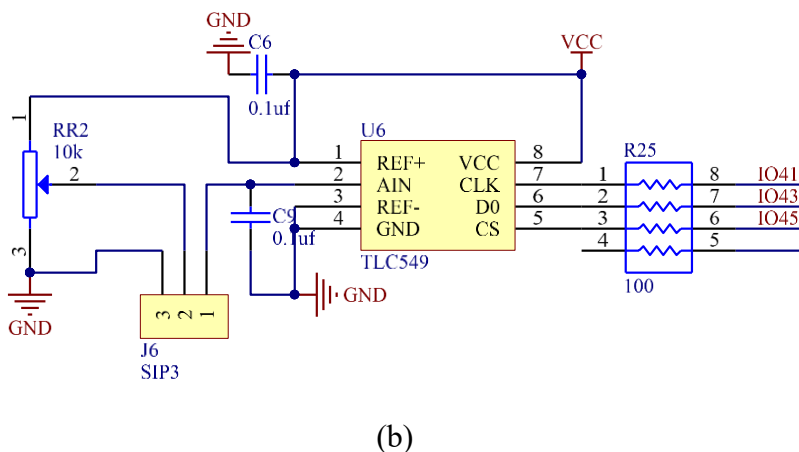
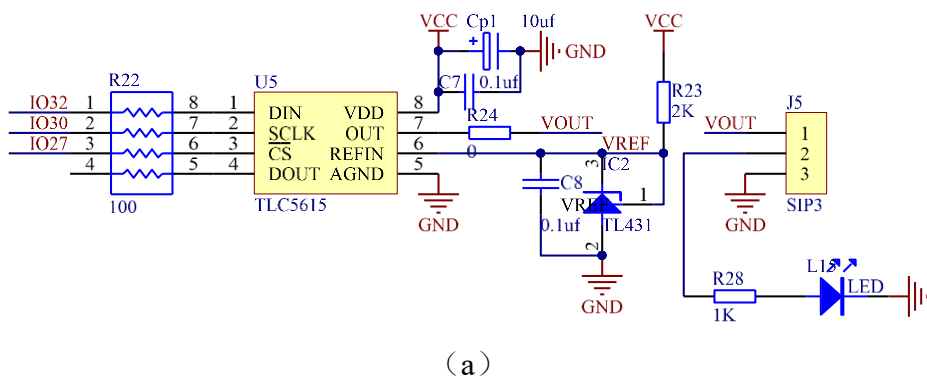
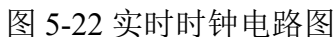


图 5-21 A/D、D/A 转换电路

11.RTC 实时时钟电路

实时时钟芯片采用具有 I2C 接口的低功耗的 CMOS 实时时钟/日历芯片 PCF8563T，它提供一个可编程时钟输出，一个中断输出和掉电检测器，所有的地址和数据通过 I2C 总线接口串行传输。最大总线速度为 400kbps，每次读/写数据后，内嵌的字地址寄存器会自动产生增量。J8 是备用电池，在电路板不供电时可以保证实时时钟还可以正常运转。实时时钟电路图如 5-22 所示。



12.温度传感器电路

13. 存储器 AT24C02 电路

66 / 306

是 I2C 通信接口，AT24C02 内部可以存 256 个字节的数据。原理图如图 5-25 所示。

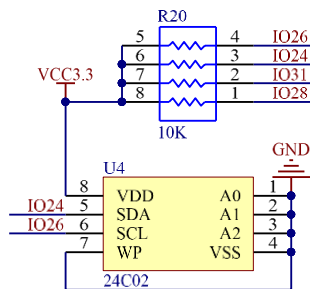


图 5-25 AT24C02 电路

14. 红外接收电路

红外接收电路采用的红外接收头为 HS0038B，红外接收电路比较简单，管脚仅有一个电源、接地和数据信号，红外接收头的数据输出口接 FPGA 的 91 引脚。HS0038B 可以接收红外遥控键盘发出的编码信号，通过 FPGA 解码出来。红外接收电路图和实物图如图 5-26 所示。



图 5-26 红外接收电路和实物图

15. 步进电机驱动电路

由于 FPGA 接口驱动电流不够，不能直接驱动步进电机，需要通过 ULN2003 增大驱动电流再连接到相应的电机接口，FPGA 控制口接到 ULN2003 的 D0\D1\D2\D3 数字端口，控制输出口 Q0\Q1\Q2\Q3 的电平状态，从而实现对步进电机的控制。步进电机驱动电路如图 5-27 所示。

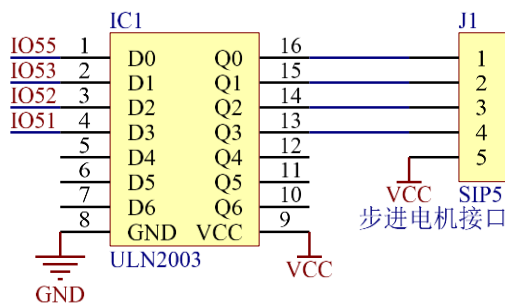


图 5-27 步进电机驱动电路

16. 旋转编码开关电路

旋转编码开关有 5 个引脚，使用时 2、5 脚接地，1、3、4 脚经上拉电阻接到 FPGA 控制 IO 口。三只管脚功能：1 2 3 管脚一般是中间 2 脚接地，1、3 脚上拉电阻后，当左转、右转旋转时，在 1、3 脚就有脉冲信号输出了。另外两只脚：4、5 脚为按压开关，按下时导通，恢复时断开。电路和实物图如图 5-25 所示。

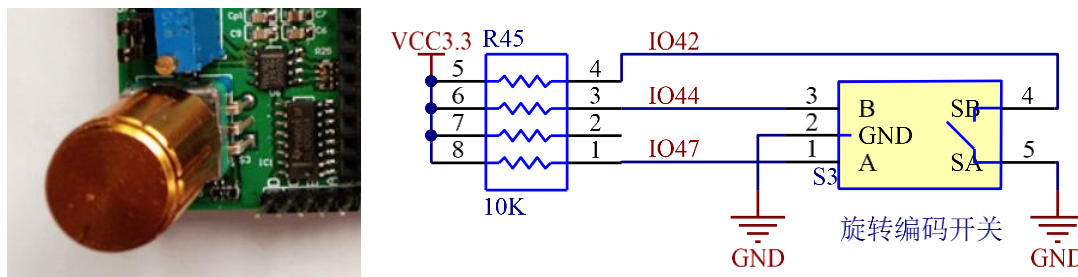


图 5-28 旋转编码开关电路和实物图

17. 超声波模块接口电路

超声波模块采用 HC-SR04 型号，需要将超声波模块的 TRIG 与 FPGA 的一个 IO 连接，FPGA 发送触发脉冲信号到 TRIG，ECHO 是检测回波输出信号，超声波模块采用 5V 供电，因为 FPGA 引脚是 3.3V 电平的，因此采用电阻分压让超声波模块输出信号电压在 FPGA 可接受范围。电路和实物图如图 5-29 所示。

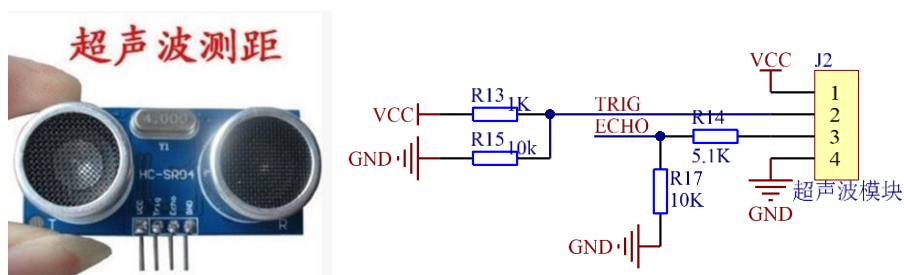


图 5-29 超声波模块接口电路和实物图

18. 插针接口电路

插针接口电路包括与 FPGA 核心板连接的 P1A，P1B 接口，以及引出方便用户使用的接口，包括 J10，J11 是和底板上其他电路有重叠复用关系的接口，在使用时确认不受其他电路影响时可以使用，J7 是和电路板上其他电路没用重叠复用的，可以放心使用。电路如图 5-30 所示。

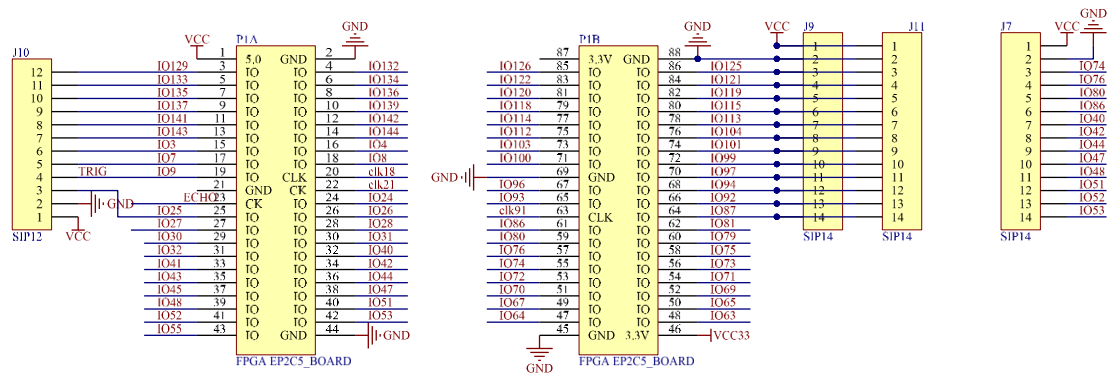


图 5-30 插针接口电路

第六章 Verilog HDL 基础知识

本章介绍硬件描述语言 Verilog HDL 的语言规则、数据类型和语句结构，并介绍最基本、最典型的数字逻辑电路的 Verilog HDL 描述，作为 Verilog HDL 工程设计的基础。

硬件描述语言的发展至今已经有 20 多年的历史。现在主要的语言 VHDL(Very High Speed Integrated Hardware Description Language)和 Verilog HDL (Verilog Hardware Description Language)适应了历史发展的趋势和要求，先后成为 IEEE 标准。硬件描述语言是一种以文本形式来描述数字系统硬件的结构和行为的语言，用它可以表示逻辑电路图、逻辑表达式，还可以表示数字逻辑系统所完成的逻辑功能，任何数字逻辑电路都可以通过硬件描述语言设计实现。硬件描述语言的出现就是用语言描述替代图形化（元件拼凑）设计，把我们要实现的功能和思想用语言的形式写出来，转换成实际电路的工作就交给 EDA 工具去做，从而简化了我们设计的工作，节约了开发的时间。

Verilog HDL 语言是在最广泛的 C 语言的基础上发展起来的一种件描述语言，它是由 GDA(Gateway Design AutomATion)公司的 PhilMoorby 在 1983 年末首创的，最初只设计了一个仿真与验证工具，之后又陆续开发了相关的故障模拟与时序分析工具。1985 年 Moorby 推出它的第三个商用仿真器 Verilog-XL,获得了巨大的成功，从而使得 Verilog HDL 迅速得到推广应用。Verilog HDL 的最大特点就是易学易用，如果有 C 语言的编程经验，可以在一个较短的时间内很快的学习和掌握。

选择学习 VHDL 还是 verilog HDL?

这是一个初学者最常见的问题。其实两种语言的差别并不大，他们的描述能力也是类似的。掌握其中一种语言以后，可以通过短期的学习，较快的学会另一种语言。选择何种语言主要还是看周围人群的使用习惯，这样可以方便日后的学习交流。当然，如果您是集成电路（ASIC）设计人员，则必须首先掌握 verilog，因为在 IC 设计领域，90%以上的公司都是采用 verilog 进行 IC 设计。对于 PLD/FPGA 设计者而言，两种语言可以自由选择。

设计人员通过计算机对 HDL 语言进行逻辑仿真和逻辑综合，方便高效地设计数字电路及其产品。常用的 Verilog HDL 语言开发软件有 Altera 公司的 MAX+PLUS II，Quartus II 和 Xilinx 公司的 Foundation ISE，Mentor Graphics 公司的 ModelSim。

6.1 Verilog HDL 设计模块的基本结构

相信大家都看过芯片吧，它有个名字，有个封装外壳，外壳向外伸出有引脚，各个引脚有对应的名称和功能，然后芯片它可以实现一定的功能。Verilog HDL 设计模块的功能就相当于设计一个芯片的功能，也包含了芯片的相关结构信息。

Verilog HDL 程序设计由模块（module）构成的，设计模块的基本结构如图 6.1 所示。一个完整的 Verilog HDL 设计模块包括端口定义、I/O 声明、变量类型声明和功能描述等 4 个部分。

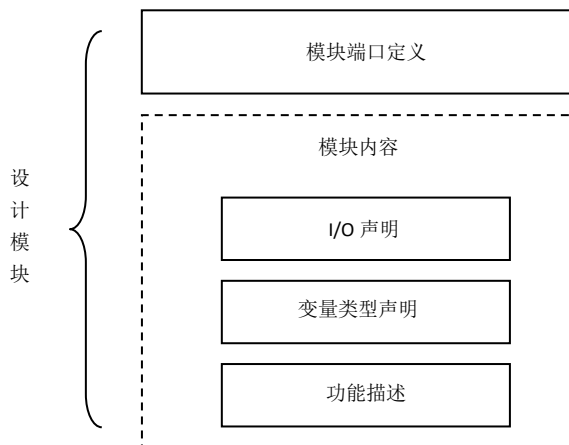


图 6.1 Verilog HDL 程序模块结构图

简单的 Verilog HDL 例子：

```
module block1(a,b,c,d); //模块端口定义
    input a,b,c; // 输出端口声明
    output d; // 输入端口声明
    wire x; //变量类型声明
    assign d = a | x; //功能描述
    assign x = ( b & ~c );
endmodule
```

整个 Verilog HDL 程序嵌套在 module 和 endmodule 声明语句中。

每条语句相对 module 和 endmodule 最好缩进 2 格或 4 格！

6.1.1 模块端口定义

模块端口定义用来声明电路设计模块名称及相应的（输入输出）端口，端口定义格式如下：

module 模块名(端口 1,端口 2,端口 3,...);

模块名是设计电路的名称，它由用户按照标识符规则命名，它也是保存的源文件名称。例如，设计一个 2 输入 4 与非门电路时，可以用 CT7400 作为模块名，并用 CT7400.v（.v 是 Verilog HDL 源文件的属性后缀）保存设计的源文件。在端口定义的圆括弧中，是设计电路模块与外界联系的全部输入和输出端口名称或引脚，它是设计模块对外的一个通信界面，是外界可以看到的部分（不包含电源和接地端），多个端口名之间用“,”分隔。例如，用 adder1 作为 1 位全加器的 Verilog HDL 设计模块名，sum 是求和输出，cout 是向高位的进位输出，a 和 b 是两个加数的输入，cin 是低位进位输入，则 adder1 模块的端口定义为

```
module adder1(sum,cout,a,b,cin);
```

6.1.2 模块内容

模块内容包括 I/O 声明、变量类型声明和功能描述。

1. 模块的 I/O 声明

模块的 I/O 声明用来声明模块端口定义中各端口数据流动方向，包括输入（input）、输出（output）和双向（inout）。I/O 声明格式如下：

```
input  端口 1,端口 2,端口 3,...; //声明输入端口
```

```
output 端口 1,端口 2,端口 3,...; //声明输出端口
```

例如，1 位全加器的 I/O 声明为

```
input  a,b,cin;
```

```
output sum,cout;
```

2. 变量类型声明

变量类型声明用来声明设计电路的功能描述中所用的变量的数据类型和函数。变量的类型主要有连线（wire）、寄存器（reg）、整型（integer）、实型（real）和时间（time）等。例如：

```
wire  a,b,cin;          //声明 a,b,cin 是 wire 变量
```

```
reg   cout;             //声明 cout 是 reg 型变量
```

```
reg [7:0] q;             //声明 q 是 8 位 reg 型变量
```

在 Verilog HDL 的 2001 版本或以上版本，允许将 I/O 声明和变量类型放在一条语句中，例如：

```
output reg[7:0] q;       //声明 q 是 8 位 reg 型输出变量
```

关于变量声明部分内容将在后续的节次中详细介绍。

3. 功能描述

功能描述是 Verilog HDL 程序设计中最主要的部分，用来描述设计模块的内部结构和模块端口间的逻辑关系，在电路上相当于器件的功能或内部电路结构。功能描述可以用 assign 语句、元件例化 (instantiate)、always 块语句、initial 块语句等方法来实现，通常把确定这些设计模块描述的方法称为建模。

(1) 用 assign 语句建模

用 assign 语句建模的方法很简单，只需要在“assign”后面再加一个表达式即可。assign 语句一般适合对组合逻辑进行赋值，称为连续赋值方式。

【例 6.1】1 位全加器的设计。

1 位全加器的逻辑符号如图 6.2 所示，其中 sum 是全加器的和输出端，cout 是进位输出端，a 和 b 是两个加数输入端，cin 是低位进位输入端。

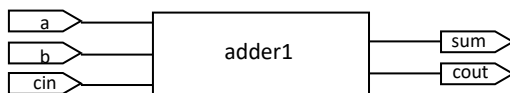


图 6.2 1 位全加器的逻辑符号

全加器设计电路的 Verilog HDL 源程序 adder1.v 如下：

```
module    adder1(sum,cout,a,b,cin);
input    a,b,cin;
output   sum,cout;
assign   {cout,sum} = a+b+cin;
endmodule
```

在例 6.1 中，用语句“assign {cout,sum} = a+b+cin;”实现 1 位全加器的进位 cout 与和输出 sum 的建模。在语句表达式中，用并接运算符“{”将 cout、sum 这两个 1 位操作数并接为一个 2 位操作数，位于左边的操作数的权值高，位于右边的操作数的权值低。

(2) 用元件例化 (instantiate) 方式建模

元件例化方式建模是利用 Verilog HDL 提供的元件库实现的。例如，用与门例化元件定义一个 3 输入端与门可以写为：

```
and myand3(y,a,b,c);
```

其中，and 是 Verilog HDL 元件库中“与门”元件名，属于关键字，myand3 是可选的自定义例化名（可以忽略），y 是与门的输出端，a、b 和 c 是输入端。

注 1：元件例化即是调用 Verilog HDL 提供的元件；

注 2: 元件例化包括门元件例化和模块元件例化;

注 3: 每个实例元件的名字必须唯一!以避免与其它调用元件的实例相混淆。

注 4: 例化元件名也可以省略!

(3) 用 always 块语句建模

always 块语句可以用于设计组合逻辑和时序逻辑,但时序逻辑必须用 always 块语句来建模。一个程序设计模块中,可以包含一个或多个 always 语句。程序运行中,在某种条件满足时,就重复执行一遍 always 结构中的语句。

【例 6.2】8 位二进制加法计数器的设计。

用 Verilog HDL 设计的 8 位二进制加法计数器的元件符号如图 6.3 所示,其中 q 是 8 位二进制计数器的输出端 (8 位向量); cout 是进位输出端 (1 位); d 是并行数据输入端

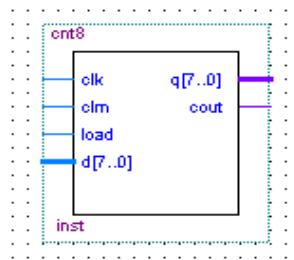


图 6.3 8 位二进制加法计数器的逻辑符号

(8 位向量); clk 是时钟控制输入端,上升沿为有效边沿; clrn 是同步复位输入端,低电平有效,当 clk 的上升沿到来时且 clrn=0,则计数器被复位, q=0。load 是计数器的预置控制输入端,当 clk 的上升沿到来时,若 load=1 (有效), q=d。

8 位二进制加法计数器的 Verilog HDL 源程序 cnt8.v 如下:

```
module cnt8(clk,clrn,load,d,q,cout);
```

```
    input[7:0]      d;
```

```
    input          clk,clrn,load;
```

```
    output reg[7:0] q;
```

```
    output reg      cout;
```

```
always      @(posedge clk)
```

```
begin
```

```
    if (~clrn) q = 0;
```

```
    else if (load) q = d;
```

```
    else      q = q+1;
```

```
        cout = &q;
```

```
    end
```

```
endmodule
```

在例 6.2 的源程序中,用 always 块语句来实现 8 位二进制加法计数器的建模。@(posedge clk)是时间控制敏感函数,表示 clk 上升沿的到来的敏感时刻。每当 clk 的上升沿的到来时, assign 块中的全部语句就执行一遍。另外,在程序的

最后用“`cout = &q`”语句产生进位输出，在语句中“`&q`”是“与”缩减运算中，只有 `q` 中的数字全为 1 时，结果 `cout` 才为 1。

注 1：“`always`”块语句常用于描述时序逻辑，也可描述组合逻辑。

注 2：“`always`”块可用多种手段来表达逻辑关系，如用 `if-else` 语句或 `case` 语句。

注 3：“`always`”块语句与 `assign` 语句是并发执行的，`assign` 语句一定要放在“`always`”块语句之外！

（4）用 `initial` 块语句建模

`initial` 块语句与 `always` 语句类似，不过在程序中它只执行 1 次就结束了，`initial` 块语句常用于设计电路的初始化操作和仿真。

从例 6.1 和例 6.2 可以看出 Verilog HDL 程序设计模块的基本结构：

① Verilog HDL 程序是由模块构成的。每个模块的内容都是嵌在 `module` 和 `endmodule` 两语句之间，每个模块实现特定的功能，模块是可以进行层次嵌套的。

② 每个模块首先要进行端口定义，并声明输入（`input`）、输出（`output`）或双向（`inout`），然后对模块的功能进行逻辑描述。

③ Verilog HDL 程序的书写格式自由，一行可以一条或多条语句，一条语句也可以分为多行写。

④ 除了 `end` 或以 `end` 开头的关键字（如 `endmodule`）语句外，每条语句后必须要有分号“`;`”。程序中的关键字全部用小写字母书写，标点符号符号全部用半角符号。

⑤ 可以用 `/*.....*/` 或 `//.....` 对 Verilog HDL 程序的任何部分作注释。一个完整的源程序都应当加上需要的注释，以加强程序的可读性。

⑥ 每个 Verilog HDL 源文件中只准有一个顶层模块，其他为子模块。

Verilog HDL 模块的模板（仅考虑用于逻辑可综合的部分）。

```
module <顶层模块名> (< 输入输出端口列表>) ;
```

```
    output 输出端口列表;
```

```
    input 输入端口列表;
```

```
//（1）使用 assign 语句定义逻辑功能
```

```
    wire 结果信号名;
```

```
    assign <结果信号名> = 表达式 ;
```

```
//（2）使用 always 块定义逻辑功能
```

```
    always @( <敏感信号表达式> )
```

```

begin
    //过程赋值语句
    //if 语句
    // case 语句
    // while,repeat,for 循环语句
    // task,function 调用
end
// (3) 元件例化
    <module_name> <instance_name> (<port_list>); // 模块元件例化
    <gate_type_keyword> <instance_name> (<port_list>); // 门元件例化
endmodule

```

6.2 Verilog HDL 的词法

Verilog HDL 源程序由空白符号分隔的词法符号流所组成。词法符号包括空白符、注释、操作符、常数、字符串、标识符和关键字。准确无误地理解和掌握 Verilog HDL 的词法的规则和用法，对正确地完成 Verilog HDL 程序设计十分重要。

6.2.1 空白符和注释

Verilog HDL 的空白符包括空格、tab 符号、换行和换页。空白符用来分隔各种不同的词法符号，合理地使用空白符可以使源程序具有一定的可读性和编程风格。空白符如果不是出现在字符串中，编译源程序时将被忽略。

注释分为行注释和块注释两种方式。行注释用符号//（两个斜杠）开始，注释到本行结束。块注释用/*开始，用*/结束。块注释可以跨越多行，但它们不能嵌套。

6.2.2 常数

在 Verilog HDL 中的常数包括数字、未知 x 和高阻 z 三种。数字可以用二进制、十进制、八进制和十六进制等 4 种不同数制来表示，完整的数字格式为：

<位宽>'<进制符号><数字>

其中，位宽表示数字对应的二进制数的位数宽度；进制符号包括 b 或 B（表示二进制数），d 或 D（表示十进制数），h 或 H（表示十六进制数），o 或 O（表示八进制数）。例如：

```
8'b10110001 //表示位宽为 8 位的二进制数 10110001
```

```
8'hf5 //表示位宽为 8 位的十六进制数 f5
```

数字的位宽可以缺省，例如：

```
'b10110001 //表示二进制数
```

```
'hf5 //表示十六进制数
```

十进制数的位宽和进制符号可以缺省，例如：

```
125 //表示十进制数 125
```

另外，用 x（或 X）和 z（或 Z）分别表示未知值和高阻态值，它们可以出现在除十进制数以外的数字形式中。x 和 z 的位数由所在的数字格式决定，在二进制数格式中，一个 x 或 z 表示 1 位未知位或 1 位高阻位；在十六进制数中，一个 x 或 z 表示 4 位未知位或 4 位高阻位；在八进制数中，一个 x 或 z 表示 3 位未知位或 3 位高阻位。例如

```
'b1111xxxx //等价'hfx
```

```
'b1101zzzz //等价'hdz
```

6.2.3 字符串

字符串是用双引号括起来的可打印字符序列，它必须包含在同一行中。例如，“ABC”，“A BOY.”，“A”，“1234”都是字符串。

6.2.4 关键字

关键字是 Verilog HDL 预先定义的单词，它们在程序中有不同的使用目的。例如，module 和 endmodule 来指出源程序模块的开始和结束；用 assign 来描述一个逻辑表达式等。Verilog -1995 的关键词有 97 个，Verilog-2001 增加了 5 个，共 102 个，如表 4.1 所示，每个关键词全部由小写字母组成，少数关键词中包含“0”或“1”数字。

表 4.1 Verilog HDL 关键字

always	and	assign	begin	buf
bufi0	bufu1	case	casex	casez
cmos	deassign	default	defparam	disable
edge	else	end	endcase	endfunction
endmodule	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	nottif1
or	output	pmos	posedge	primitive
pull0	pull1	pulldown	pullup	remos
reg	release	Repeatr	rnmos	rpmos
rtran	rtranif0	reranif1	scalared	small
specify	specparam	strong0	strong1	supply0
supply1	table	task	time	tran
tranif0	tranif1	tri	tri0	tri1
triand	trior	vectored	wait	wand
weak0	weak1	while	wire	wor
xnot	xor			

6.2.5 标识符

标识符是用户编程时为常量、变量、模块、寄存器、端口、连线、示例和 **begin-end** 块等元素定义的名称。标识符可以是字母、数字和下划线“**_**”等符号组成的任意序列。定义标识符时应遵循如下规则

- ① 首字符不能是数字。
- ② 字符数不能多于 1024 个。
- ③ 大小写字母是不同的。
- ④ 不要与关键字同名。

例如，**a**、**b**、**adder8**、**counter**、**my_adder** 都是正确的标识符；而 **123a**、**?b**、**module** 是错误的标识符。

与 VHDL'93 标准支持扩展标识符类似，Verilog HDL 允许使用转义标识符。转义标识符中可以包含任意的可打印字符，转义标识符从空白符号开始，以反斜杠“****”作为开始标记，到下一个空白符号结束，反斜杠不是标识符的一部分。下面是转义标识符的示例

```
\74LS00
```

```
\a+b
```

6.2.6 操作符

操作符也称为运算符，是 Verilog HDL 预定义的函数名字，这些函数对被操作的对象（即操作数）进行规定的运算，得到一个结果。操作符通常由 1~3 个字符组成，例如，“+”表示加操作，“==”（两个=字符）表示逻辑等操作，“===”

（3 个=字符）表示全等操作。有些操作符的操作数只有 1 个，称为单目操作；有些操作符的操作数有 2 个，称为双目操作；有些操作符的操作数有 3 个，称为三目操作。

Verilog HDL 的操作符分为算术操作符、逻辑操作符、关系操作符、等值操作符、缩减操作符、条件操作符、位操作符和并接操作符 9 类。

1. 算术操作符（Arithmetic operators）

常用的算术操作符有：+（加）、-（减）、*（乘）、/（除）、%（求余）和**（乘方）6 种。其中%是求余操作符，在两个整数相除的基础上，取出其余数。例如，5%6 的值为 5；13%5 的值是 3。整除（/）和求余（%）运算符可以方便电路的设计，如将二进制数转换为十进制数（8421BCD 码），但这两种运算符在综合过程中占用很多逻辑单元（LEs），所以一般电路设计最好不要使用。算术操作符的例程如下所示。

```
module arithmetic(clk,a,b,c,d,e);
output [3:0]c,d;
output [7:0]e;
input [3:0]a,b;
input      clk;
reg[3:0]c,d;
reg[7:0]e;
always@(posedge clk)
begin
    c=a/b; //整数除法运算时，结果值只取整数部分
    d=a%b; //求余数
    e=a*b; //乘法
end
endmodule
```

2. 逻辑操作符（Logical operators）

逻辑操作符包括：&&（逻辑与）、||（逻辑或）、！（逻辑非）。例如，A&&B

表示 A 和 B 进行逻辑与运算； $A \parallel B$ 表示 A 和 B 进行逻辑或运算； $!A$ 表示对 A 进行逻辑非运算。该操作符常用于条件语句的判断条件中，示例如下：

```
always@(a or b)
begin
    if(a && b) c=1;
    else c=0;
end
```

3. 位运算 (Bitwise operators)

位运算是将两个操作数按对应位进行逻辑操作。位运算操作符包括： $\sim$ （按位取反）、 $\&$ （按位与）、 $|$ （按位或）、 $\wedge$ （按位异或）、 $\wedge\sim$ 或 $\sim\wedge$ （按位同或）。例如，设 $A=\text{'b11010001}$ ， $B=\text{'b00011001}$ ，则相关位运行结果如下：

```
 $\sim A = \text{'b00101110}$ 
 $A \& B = \text{'b00010001}$ 
 $A | B = \text{'b11011001}$ 
 $A \wedge B = \text{'b11001000}$ 
 $A \wedge\sim B = \text{'b00110111}$ 
```

在进行位运算时，当两个操作数的位宽不同时，计算机会自动将两个操作数按右端对齐，位数少的操作数会在高位用 0 补齐。

位运算与逻辑操作符运算的结果是相同的，因此，逻辑操作运算直接可以用位运算替代，例如， $A \&\& B$ 可以写成 $A \& B$ 。

4. 关系操作符 (Relational operators)

关系操作符用来对两个操作数进行比较。关系操作符有： $<$ （小于）、 $\leq$ （小于等于）、 $>$ （大于）、 $\geq$ （大于等于）。其中， $\leq$ 也是赋值运算的赋值符号。

关系运算的结果是 1 位逻辑值。在进行关系运算时，如果关系是真，则计算结果为 1；如果关系是假，则计算结果为 0；如果某个操作数的值不定，则计算结果不定（未知 x），表示结果是模糊的。该操作符常用于条件语句的判断条件中。如：

```
always@(a or b)
begin
    if(a > b) c=1;
    else c=0;
end
```


5. 等值操作符 (Equality operators)

等值操作符包括：==（等于）、!=（不等于）、===（全等）、!==（不全等）4 种。

等值运算的结果也是 1 位逻辑值，当运算结果为真时，返回值 1；为假则返回值 0。相等操作符（==）与全等操作符（===）的区别是：当进行相等运算时，两个操作数必须逐位相等，其比较结果的值才为 1（真），如果某些位是不定或高阻状态，其相等比较的结果就会是不定值；而进行全等运算时，对不定或高阻状态位也进行比较，当两个操作数完全一致时，其结果的值才为 1（真），否则结果为 0（假）。

例如，设 A = 'b1101xx01，B = 'b1101xx01 则

A == B 运算的结果为 x（未知）；

A === B 运算的结果为 1（真）。

该操作符常用于条件语句的判断条件中。如下计数器例子所示：

```
module count (clk,cnt);
input clk;
output [2:0] cnt;
reg [2:0] cnt;
always@(posedge clk )
begin
    if(cnt==7)
        cnt<=0;
    else
        cnt<=cnt+1;
end
endmodule
```

6. 缩减操作符 (Reduction operators)

缩减操作符包括：&（与）、~&（与非）、|（或）、~|（或非）、^（异或）、^~或~^（同或）。缩减操作运算法则与逻辑运算操作相同，但操作的运算对象只有一个。在进行缩减操作运算时，对操作数进行与、与非、或、或非、异或、同或等缩减操作运算，运算结果有 1 位“1”或“0”。例如，设 A = 'b11010001，则&A=0（在与缩减运算中，只有 A 中的数字全为 1 时，结果才为 1）；|A=1（在或缩减运算中，只有 A 中的数字全为 0 时，结果才为 0）。缩减操作相当于一个逻辑门，与缩减运算相当于一个与门，只有与门的全部输入为“1”时，输出（1 位）

才为“1”，否则输出为“0”。

缩减操作符是对单个操作数进行递推运算,即先将操作数的最低位与第二位进行与、或、非运算，再将运算结果与第三位进行相同的运算，依次类推，直至最高位。运算结果缩减为 1 位二进制数。如下例所示：

```
reg[3:0] a;
b=a;           //等效于 b=(a[0] | a[1]) | a[2]) | a[3]
```

6. 转移操作符 (Shift operators)

转移操作符包括：>>（右移）、<<（左移）。其使用方法为

操作数 >> n; //将操作数的内容右移 n 位，同时从左边开始用 0 来填补移出的位数。

操作数 << n; //将操作数的内容左移 n 位，同时从右边开始用 0 来填补移出的位数。

例如，设 A = 'b11010001，则 A >> 4 的结果是 A = 'b00001101；而 A << 4 的结果是 A = 'b00010000。

8. 条件操作符(Conditional operators)

条件操作符为：?:

条件操作符的操作数有 3 个，其使用格式为

操作数 = 条件 ? 表达式 1:表达式 2;

即当条件为真（条件结果值为 1）时，操作数 = 表达式 1；为假（条件结果值为 0）时，操作数 = 表达式 2。

【例 6.3】用 Verilog HDL 的条件操作符设计三态输出电路。

三态输出电路如图 6.4 所示，其中 a 是 1 位数据输入端，f 是 1 位数据输出端，en 是使能控制输入端，高电平有效，当 en=1 时，电路工作，输出 f=a，当 en=0 时，电路不工作，输出为高阻态（f='bz）。

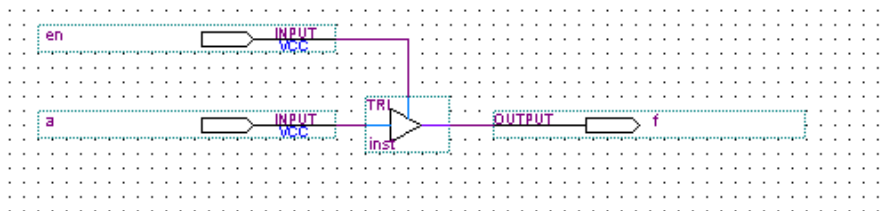


图 6.4 例 4.3 的硬件实现电路

用 Verilog HDL 的条件操作符设计三态输出电路的源程序如下：

```
module tri_v(f,a,en);
input a,en;
```

```
output f;
assign f = en ? a: 'bz;
endmodule
```

在本例中用了含有“?:”条件操作符的“`assign f = en ? a: 'bz;`”语句来描述三态输出电路，`assign`语句的条件是变量 `en`，其值只有 0 和 1 两种。如果 `en` 为 1（真）时 `y=a`，为 0（假）时则 `y='bz`（高阻）。三态输出电路的仿真波形如图 6.5 所示，在波形图中，当 `en=0` 时输出 `f` 为高阻态（高阻态是以在高低电平中部的粗线表示），当 `en=1` 时，输出 `f` 与输入 `a` 的波形相同。

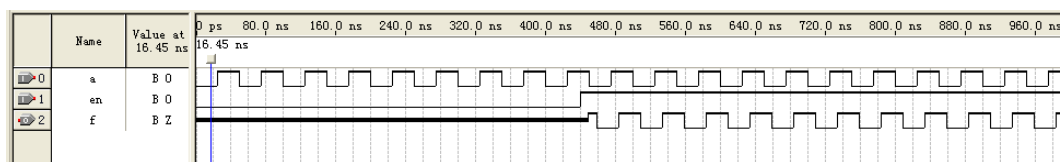


图 6.5 三态输出电路的仿真波形

9. 并接操作符（Concatenation operators）

并接操作符为：{ }

并接操作符的使用格式为

{操作数 1 的某些位,操作数 2 的某些位,...,操作数 n 的某些位};

即将操作数 1 的某些位，与操作数 2 的某些位，与…，与操作数 n 的某些位并接在一起，构成一个由这些数组成的多位数，当做一个整体信号。例如，将 1 位全加器进位 `cont` 与和 `sum` 并接在一起使用，它们的结果由两个加数 `a`、`b` 及低位进位 `cin` 相加决定的表达式为：

[示例 1] `assign {cont,sum}= a+b+cin;`

[示例 2] `{a,b[3:0],w,3'b101} = {a,b[3],b[2],b[1],b[0],w,1'b1,1'b0,1'b1}`

可用重复法简化表达式，如： `{4{w}}` //等同于 `{w,w,w,w}`

还可用嵌套方式简化书写，如：

`{b,{3{a,b}}}` //等同于 `{b,{a,b},{a,b},{a,b}}`，也等同于 `{b,a,b,a,b,a,b}`

在位拼接表达式中，不允许存在没有指明位数的信号，必须指明信号的位数；若未指明，则默认为 32 位的二进制数！如：

`{1,0} = 64'h00000001_00000000,`

注意 `{1,0}` 不等于 `2'b10`。

10. 操作符的优先级

操作符的优先级如表 6.2 所示。表中顶部的操作符优先级最高，底部的最低，

列在同一行的操作符的优先级相同。所有的操作符（?:操作符除外）在表达式中都是从左向右结合的。圆括号可以用来改变优先级，并使运算顺序更清晰，对操作符的优先级不能确定时，最好使用圆括弧来确定表达式的优先顺序，既可以避免出错，也可以增加程序的可读性。

[示例] (a>b)&&(b>c)

(a==b)||((x==y)

(!a)||((a>b)

表 6.2 操作符的优先级

优先级序号	操作符	操作符名称
1	!、~	逻辑非、按位取反
2	*、/、%	乘、除、求余
3	+、-	加、减
4	<<、>>	左移、右移
5	<、<=、>、>=	小于、小于等于、大于、大于等于
6	==、!=、===、!==	等于、不等于、全等、不全等
7	&、~&	缩减与、缩减与非
8	^、^~	缩减异或、缩减同或
9	、~	缩减或、缩减或非
10	&&	逻辑与
11		逻辑或
12	?:	条件操作符

6.2.7 Verilog HDL 数据对象

Verilog HDL 数据对象是指用来存放各种类型数据的容器，包括常量和变量。

1. 常量

常量是一个恒定不变的值数，一般在程序前部定义。常量定义格式为：

parameter 常量名 1 = 表达式,常量名 2 = 表达式,...,常量名 n = 表达式;

其中，parameter 是常量定义关键字，常量名是用户定义的标识符，表达式是为常量赋的值。例如

```
parameter Vcc = 5, fbus = 8'b11010001;
```

上述语句定义了常量 Vcc 的值为十进制数 5，常量 fbus 的值为二进制数 8'b11010001。

每个赋值语句的右边必须为常数表达式，即只能包含数字或先前定义过的符号常量！

```
parameter addrwidth = 16;           //合法格式
```

```
parameter addrwidth = datawidth*2; //非法格式
```

常用参数来定义延迟时间和变量宽度。

可用字符串表示的任何地方,都可以用定义的参数来代替。

参数是本地的,其定义只在本模块内有效。

在模块或实例引用时,可通过参数传递改变在被引用模块或实例中已定义
的参数!

2. 变量

变量是在程序运行时其值可以改变的量。在 Verilog HDL 中,变量分为网络型 (nets type) 和寄存器型 (register type) 两种。

(1) 网络型变量 (nets type)

nets 型变量是输出值始终根据输入变化而更新的变量,它一般用来定义硬件电路中的各种物理连线。Verilog HDL 提供了多种 nets 型变量,如表 6.3 所列。

表 6.3 常用的 nets 型变量及说明

类 型	功能说明
wire、tri	连线类型 (两者功能完全相同)
wor、trior	具有线或特性的连线 (两者功能一致)
wand、triand	具有线与特性的连线 (两者功能一致)
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)

在 nets 型变量中, wire 型变量是最常用的一种。wire 型变量常用来表示以 assign 语句赋值的组合逻辑变量。Verilog HDL 模块中的输入输出变量类型缺省时自动定义为 wire 型。wire 型变量可以作为任何方程式的输入,也可以作为 assign 语句和例化元件的输出。对综合而言, wire 型变量的取值可以是 0、1、x 和 z。

wire 型变量的定义格式如下:

```
wire 变量名 1,变量名 2,...,变量名 n;
```

[示例]

```
wire      a,b,c;           //定义了 3 个 wire 型的变量,位宽均为 1 位
```

```
wire[7:0] databus;        //定义了 1 个 wire 型的数据总线,位宽为 8 位
```

```
wire[15:0] addrbus;        //定义了 1 个 wire 型的地址总线,位宽为 16 位
```

(2) 寄存器型变量 (register type)

register 型变量对应具有状态保持作用的电路元件 (如触发器、寄存器等),常用来表示过程块语句 (如 initial, always, task, function) 内的指定信号。它作为一种数值容器,不仅可以容纳当前值,也可以保持历史值,这一属性与触发器

或寄存器的记忆功能有很好的对应关系。变量也是一种连接线，可以作为设计模块中各器件间的信息传送通道。

register 型变量与 wire 型变量的根本区别在于 register 型变量需要被明确地赋值，并且在被重新赋值前一直保持原值。wire 型变量在 assign 语句和元件例化语句中赋值，而 register 型变量是在 always、initial 等过程语句中赋值。register 型变量必须通过过程赋值语句赋值！不能通过 assign 语句赋值！在过程块内被赋值的每个信号必须定义成 register 型！

Verilog HDL 中的 register 型变量有 4 种，如表 6.4 所列。integer、real 和 time 等 3 种寄存器型变量都是纯数学的抽象描述（不可综合），不对应任何具体的硬件电路，但它们可以描述与模拟有关的计算。例如，可以利用 time 型变量控制经过特定的时间后关闭显示等。

表 6.4 常用的 register 型变量及说明

类 型	功 能 说 明
reg	常用的寄存器型变量
integer	32 位带符号整数型变量
real	64 位带符号实数型变量
time	无符号时间型变量

reg 型变量是可综合的数字系统中存储设备的抽象，常用于的具体的硬件描述，因此是最常用的寄存器型变量，下面重点介绍 reg 型变量。

reg 型变量定义的关键字是 reg，定义格式如下：

reg [位宽] 变量 1,变量 2,...,变量 n;

用 reg 定义的变量有一个范围选项（即位宽），默认的位置是 1。位宽为 1 位的变量称为标量，位宽超过 1 位的变量称为向量。标量的定义不需要加位宽选项，例如

```
reg    a,b;           //定义两个 reg 型变量 a, b
```

向量定义时需要位宽选项，例如

```
reg[7:0]    data;      //定义 1 个 8 位寄存器型变量，最高有效位是 7，最低有效位是 0
```

```
reg[0:7]    data;      //定义 1 个 8 位寄存器型变量，最高有效位是 0，最低有效位是 7
```

向量定义后可以采用多种使用形式（即赋值）。

① 为整个向量赋值的形式为

```
data=8'b00000000;
```

② 为向量的部分位赋值的形式为

`data[5:3]=3'b111;` //将 data 的第 5, 4, 3 位赋值为 “111”

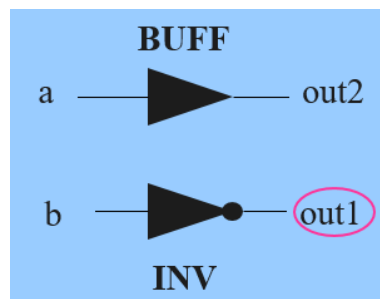
③ 为向量的某一位赋值的形式为

`data[7]=1'b1;`

在过程块中被赋值的 reg 信号，往往代表触发器，但不一定就是触发器（也可以是组合逻辑信号）！reg 型变量既可生成触发器，也可生成组合逻辑；wire 型变量只能生成组合逻辑。

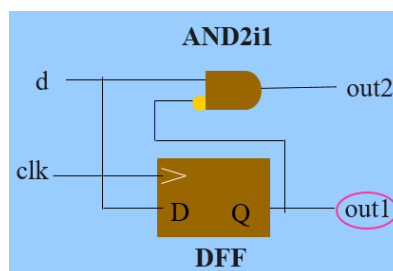
1)用 reg 型变量生成组合逻辑举例：

```
module rw1( a, b, out1, out2 ) ;
    input a, b;
    output out1, out2;
    reg out1;
    wire out2;
    assign out2 = a ; //连续赋值语句
    always @(b)      //电平触发
        out1 <= ~b; //过程赋值语句
endmodule
```



2)用 reg 型变量生成触发器举例：

```
module rw2( clk, d, out1, out2 );
    input clk, d;
    output out1, out2;
    reg out1;
    wire out2;
    assign out2 = d & ~out1 ; //连续赋值语句
    always @(posedge clk) //边沿触发
    begin
        out1 <= d ; //过程赋值语句
    end
endmodule
```



（3）数组

若干个相同宽度的向量构成数组。在数字系统中，reg 型数组变量即为 memory（存储器）型变量。存储器型可以用如下语句定义：

```
reg[7:0]    mymemory[511:0];
```

上述语句定义了一个 512 个字存储器变量 `mymemory`, 每个字的字长为 8 位。在表达式中可以用下面的语句来使用存储器:

```
mymemory[7] = 75; //存储器 mymemory 的第 7 个字被赋值 75
```

memory 型变量与 reg 型变量的区别:

[示例] `reg[n-1:0] rega;` //一个 n 位的寄存器

```
reg mema [n-1:0] ; //由 n 个 1 位寄存器组成的存储器
```

一个 n 位的寄存器可用一条赋值语句赋值; 一个完整的存储器则不行! 若要对某存储器中的存储单元进行读写操作, 必须指明该单元在存储器中的地址!

[示例] `rega = 0;` //合法赋值语句

```
mema = 0 ;      //非法赋值语句
```

```
mema[8] = 1 ;      //合法赋值语句
```

```
mema[1023:0] = 0 ;      //合法赋值语句, 必须指明存储单元的地址!
```

6.3 Verilog HDL 的语句

语句是构成 Verilog HDL 程序不可缺少的部分。Verilog HDL 的语句包括赋值语句、条件语句、循环语句、结构声明语句和编译预处理语句等类型, 每一类语句又包括几种不同的语句。在这些语句中, 有些语句属于顺序执行语句, 有些语句属于并行执行语句。

6.3.1 赋值语句

在 Verilog HDL 中, 赋值语句常用于描述硬件设计电路输出与输入之间的信息传送, 改变输出结果。Verilog HDL 有门基元、连续赋值、过程赋值和非阻塞赋值等 4 种赋值方法 (即语句), 不同的赋值语句使输出产生新值的方法不同。

1. 门基元赋值语句

门基元赋值语句也称为元件例化语句, 其格式为:

基本逻辑门关键字 (门输出, 门输入 1, 门输入 2, ..., 门输入 n);

其中, 基本逻辑门关键字是 Verilog HDL 预定义的逻辑门, 包括 `and`、`or`、`not`、`xor`、`nand`、`nor` 等; 圆括弧中内容是被描述门的输出和输入变量。

例如, 具有 a、b、c、d 四个输入和 y 为输出与非门的门基元赋值语句为:

```
nand mynand4(y,a,b,c,d);
```

//该语句与 `assign y = ~(a & b & c & d)`等效

表 6.5 基本门级元件模型表

元件符号	功能说明	元件符号	功能说明
and	多输入端的与门	nand	多输入端的与非门
or	多输入端的或门	nor	多输入端的或非门
xor	多输入端的异或门	xnor	多输入端的异或非门
buf	多输出端的缓冲器	not	多输出端的反相器
bufif1	控制信号高电平有效的三态缓冲器	notif1	控制信号高电平有效的三态反相器
bufif0	控制信号低电平有效的三态缓冲器	notif0	控制信号低电平有效的三态反相器

基本门级元件按输入输出方式分类如下：

- 1) 多输入门:只允许有一个输出，但可以有多多个输入。

```
and A1 (out, in1, in2, in3);
```

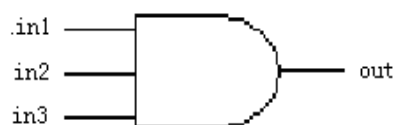


图 6.6 三输入与门元件模型

- 2) 多输出门: 允许有多个输出，但只有一个输入。

```
not N1 (out1, out2, ..., in);
```

```
buf B1 (out1, out2, ..., in);
```

- 3) 三态门: 有一个输出、一个数据输入和一个输入控制。如果输入控制信号无效，则三态门的输出为高阻态 z。

```
bufif1 B1(out, in, ctrl);
```

```
notif1 N1(out, in, ctrl);
```



图 6.7 三态门元件模型
(a) bufif1 (b) notif1

2. 连续赋值语句

连续赋值语句的关键字是 assign，赋值符号是“=”，赋值语句的格式为：

```
assign 赋值变量 = 表达式;
```

例如，具有 a、b、c、d 四个输入和 y 为输出与非门的连续赋值语句为：

```
assign y = ~(a & b & c & d);
```

连续赋值语句的“=”号左边的变量 y 应该是 wire 型变量，而右边的变量可以是 reg 或者 wire 型变量。在执行中，输出 y 的变化跟随输入 a、b、c、d 的变化而变化，反映了信息传送的连续性。连续赋值语句用于逻辑门和组合逻辑电路的描述。

【例 6.4】4 输入端与非门的 Verilog HDL 源程序。

```
module example_4(y,a,b,c,d);
output y;
input a,b,c,d;
assign y = ~(a&b&c&d);
endmodule
```

3. 过程赋值语句

用于对 reg 型变量赋值，有两种方式：

阻塞（blocking）赋值方式：

赋值符号为=，如 b = a ；

非阻塞（non-blocking）赋值方式：

赋值符号为<=，如 b <= a ；

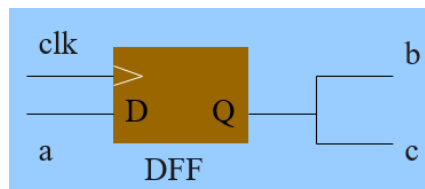
（1）阻塞赋值语句

阻塞赋值语句出现在 initial 和 always 块语句中，赋值符号是“=”，它是顺序语句，语句格式为：

赋值变量 = 表达式；

在阻塞赋值语句中，赋值号“=”左边的赋值变量必须是 reg（寄存器）型变量，其值在该语句结束即可得到。如果一个块语句中包含若干条阻塞赋值语句，那么这些阻塞赋值语句是按照语句编写的顺序由上至下一条一条地执行，前面的语句没有完成，后面的语句就不能执行，就如同被阻塞了一样。因此称为阻塞赋值语句。

```
[示例] always @(posedge clk)
begin
    b = a ;
    c = b;
end
```



//注： b 和 c 的值相同！

（2）非阻塞赋值语句

非阻塞赋值语句也是出现在 `initial` 和 `always` 块语句中，赋值符号是 “`<=`”，语句格式为：

赋值变量 `<=` 表达式；

在非阻塞赋值语句中，赋值号 “`<=`” 左边的赋值变量也必须是 `reg` 型变量，其值不像在阻塞赋值语句那样，语句结束时即刻得到，而在该块语句结束才可得到。

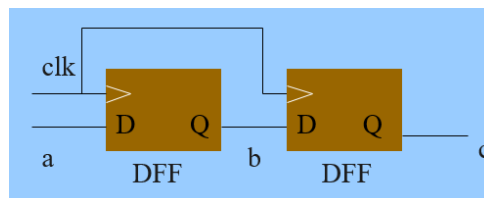
[示例] `always @(posedge clk)`

`begin`

`b <= a;`

`c <= b;`

`end`



//注：c 的值比 b 的值落后一个时钟周期！

（3）阻塞赋值与非阻塞赋值的主要区别

1) 非阻塞（non-blocking）赋值方式（`b <= a`）：

b 的值被赋成新值 a 的操作，并不是立刻完成的，而是在块结束时才完成；块内的多条赋值语句在块结束时同时赋值；硬件有对应的电路。

2) 阻塞（blocking）赋值方式（`b = a`）：

b 的值立刻被赋成新值 a；

完成该赋值语句后才能执行下一句的操作；硬件没有对应的电路，因而综合结果未知。

建议在初学时只使用一种方式，不要混用！

建议在可综合风格的模块中使用非阻塞赋值！

例如，在下面的块语句中包含 4 条赋值语句

`always @(posedge clock)`

`begin`

`m = 3;`

`n = 75;`

`n <= m;`

`r = n;`

`end`

语句执行结束后，r 的值是 75，而不是 3，因为第 3 行是非阻塞赋值语句 “`n <= m`”，该语句要等到本块语句结束时，n 的值才能改变。块语句中的 “`@(posedge clock)`” 是定时控制敏感函数，表示时钟变量 clock 的上升沿到来的敏感时刻。

过程赋值语句和非阻塞赋值语句都是在 `initial` 和 `always` 块语句中使用，因

此都称为过程赋值语句，只是赋值方式不同。过程赋值语句常用语数字系统的触发器、移位寄存器、计数器等时序逻辑电路的描述，也可用于组合逻辑电路的描述。

【例 6.5】上升沿触发的 D 触发器的 Verilog HDL 的源程序。

```
module D_FF(q,d,clock);
input  d,clock;
output q;
reg    q;
always @(posedge clock)
    q = d;
endmodule
```

在源程序中，q 是触发器的输出，属于 reg 型变量；d 和 clock 是输入，属于 wire 型变量（由隐含规则定义）。在 always 块语句中，“posedge clock”是敏感变量，只有 clock 的正边沿（上升沿）到来时，D 触发器的输出 q=d，否则触发器的状态不变（处于保持状态）。

6.3.2 条件语句

条件语句包含 if 语句和 case 语句，它们都是顺序语句，应放在 always 或 initial 块中。

1. if 语句

判定所给条件是否满足，根据判定的结果（真或假）决定执行给出的两种操作之一。完整的 Verilog HDL 的 if 语句结构如下：

```
if(表达式)
    begin    语句; end
else if(表达式)
    begin    语句; end
    .....    //多个条件情况
else
    begin    语句; end
```

根据需要，if 语句可以写为另外两种变化形式：

- ① if(表达式)


```
begin 语句; end
```
- ② if(表达式)

```
begin 语句; end
else
begin 语句; end
```

在 if 语句中，“表达式”一般为逻辑表达式或关系表达式，也可以是位宽为 1 位的变量。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，执行指定的语句。语句可以是多句，多句时用“begin-end”语句括起来；也可以是单句，单句可以省略“begin-end”语句。对于 if 语句嵌套，如果不清楚 if 和 else 的匹配，最好用“begin-end”语句括起来。

if 语句和它的变化形式属于条件语句，在程序中用来改变控制流程。

条件表达式允许一定形式的表达式简写方式，如：

if(expression) 等同于 if(expression == 1)

if(! expression) 等同于 if(expression != 1)

if 语句可以嵌套；若 if 与 else 的数目不一样，注意用“begin_end”语句来确定 if 与 else 的配对关系！

①if（表达式 1） <pre>if（表达式 2）语句 1; else 语句 2; else if（表达式 3）语句 3; else 语句 4;</pre>	②if（表达式 1） <pre>begin if（表达式 2）语句 1; end else 语句 2;</pre>

【例 6.6】8 线-3 线优先编码器的设计。

表 6.8 8 线-3 线优先编码表器的功能表

输 入								输 出		
a0	a1	a2	a3	a4	a5	a6	a7	y2	y1	y0
x	x	x	x	x	x	x	0	1	1	1
x	x	x	x	x	x	0	1	1	1	0
x	x	x	x	x	0	1	1	1	0	1
x	x	x	x	0	1	1	1	1	0	0
x	x	x	0	1	1	1	1	0	1	1
x	x	0	1	1	1	1	1	0	1	0
x	0	1	1	1	1	1	1	0	0	1
0	1	1	1	1	1	1	1	0	0	0

8 线-3 线优先编码器的功能表如表 6.8 所列，a0～a7 是 8 个变量输入端，a7 的优先级最高，a0 的优先级最低。当 a7 有效时（低电平 0），其他输入变量无效，编码输出 y2y1y0=111（a7 输入的编码）；如果 a7 无效（高电平 1），而 a6 有效，

则 $y_2y_1y_0=110$ (a_6 输入的编码); 依此类推。在传统的电路设计中, 优先编码器的设计是一个相对困难的课题, 而采用 Verilog HDL 的 if 语句, 此类难题迎刃而解, 充分体现了硬件描述语言在数字电路设计方面的优越性。

8 线-3 线优先编码器设计电路的 Verilog HDL 源程序 coder8_3.v 如下:

```
module coder8_3 (y,a);
input[7:0]      a;
output[2:0] y;
reg[2:0]      y;
always @(a)
begin
    if(~a[7])    y='b111;
    else if(~a[6])y='b110;
    else if(~a[5])y='b101;
    else if(~a[4])y='b100;
    else if(~a[3])y='b011;
    else if(~a[2])y='b010;
    else if(~a[1])y='b001;
    else if(~a[0])y='b000;
    else        y='b000;
end
endmodule
```

【例 6.7】模为 60 的 BCD 码加法计数器 counter60.v

```
module count60(qout,cout,data,load,cin,reset,clk);
output[7:0]qout;
output cout;
input[7:0]data;
input load,cin,reset,clk;
reg[7:0]qout;
always @(posedge clk)
begin
    if(reset)qout=0;           //同步复位
    else if(load) qout=data;   //同步置数
    else if(cin)               //cin=1 则加计数
        begin
```

```

if(qout[3:0]==9) //低位是否为 9
begin
    qout[3:0]=0;    //是则归 0
    if(qout[7:4]==5) //高位是否为 5
        qout[7:4]=0; //是则归 0
    else
        qout[7:4]=qout[7:4]+1; //高位不是 5 则加 1
    end
else
    qout[3:0]=qout[3:0]+1; //低位不是 9 则加 1
end

end

assign cout=((qout==8'h59)&cin)?1:0; //产生进位输出
endmodule

```

程序仿真图如图 6.9 所示，把置数功能，加计数功能，进位输出功能通过 quartus 软件功能仿真输出。

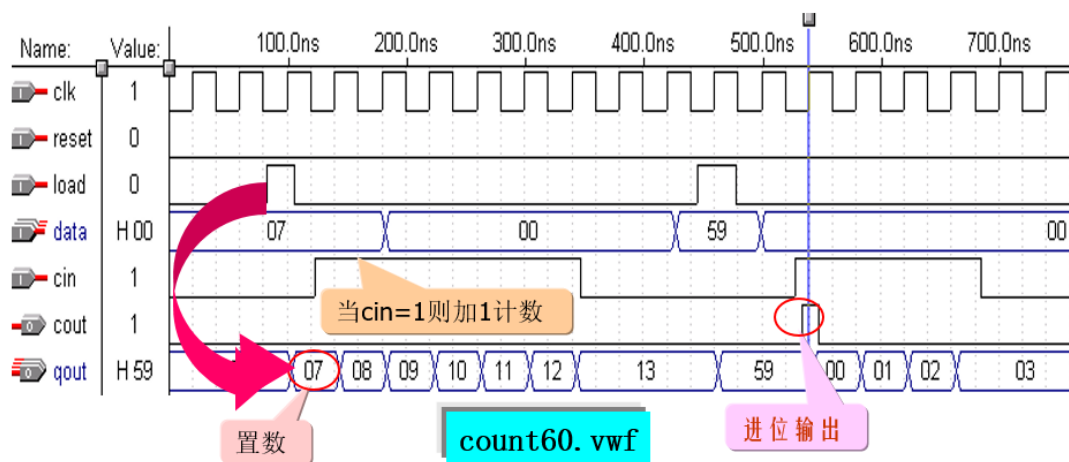


图 6.9 计数器仿真图

在 always 块内的语句是顺序执行的！

always 块语句和 assign 语句是并行执行的！

注意：if (reset)

else if (load)

else if (cin)

不要写成 3 个并列的 if 语句：

if (reset)

if (load)

if (cin)

因为这样写则是同时对 3 个信号 reset、load 和 cin 进行判断，现实中很可能出现三者同时为“1”的情况，即 3 个条件同时满足，则应该同时执行它们对应的执行语句，但 3 条执行语句是对同一个信号 qout 赋不同的值，显然相互矛盾。故编译时会报错！

2. case 语句

case 语句是一种多分支的条件语句，当敏感表达式取不同的值时，执行不同的语句。

功能：当某个（控制）信号取不同的值时，给另一个（输出）信号赋不同的值。常用于多条件译码电路（如译码器、数据选择器、状态机、微处理器的指令译码）！

完整的 case 语句的格式为：

case (敏感表达式)

```

    选择值 1 :    语句 1;
    选择值 2 :    语句 2;
    ...          ;
    选择值 n :    语句 n;
    default :    语句 n+1;
  
```

endcase

说明：

其中“敏感表达式”又称为“控制表达式”，通常表示为控制信号的某些位。

值 1~值 n 称为分支表达式，用控制信号的具体状态值表示，因此又称为常量表达式。

default 项可有可无，一个 case 语句里只能有一个 default 项！

值 1~值 n 必须互不相同，否则矛盾。

值 1~值 n 的位宽必须相等，且与控制表达式的位宽相同。

执行 case 语句时，首先计算表达式的值，然后执行在条件句中找到的“选择值”与其值相同的语句。当所有的条件句的“选择值”与表达式的值不同时，则执行“default”后的语句。当选择值涵盖了表达式的全部结果时（如表达式是 3 位二进制数，而选择值有 8 个），default 语句可以不要，不满足上述条件时，default 语句不可缺省。

【例 6.8】用 case 语句描述 4 选 1 数据选择器。

4 选 1 数据选择器的逻辑符号如图 6.10 所示，其逻辑功能见表 6.9 所列。由表可知，4 选 1 数据选择器的功能是，在控制输入变量 s_1 和 s_2 的控制下，使输入数据变量 a 、 b 、 c 、 d 中的一个被选中传送到输出 y 。 s_1 和 s_2 有 4 种组合值，可以用 `case` 语句实现其功能。

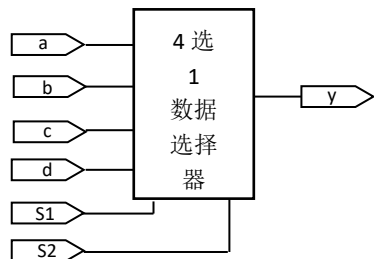


表 6.9 4 选 1 数据选择器功能表

s_1	s_2	y
0	0	a
0	1	b
1	0	c
1	1	d

图 6.10 4 选 1 数据选择器逻辑符号

4 选 1 数据选择器 Verilog HDL 的源程序如下：

```
module mux4_1(y,a,b,c,d,s1,s2);
input      s1,s2;
input      a,b,c,d;
output     y;
reg        y;
always @(s1 or s2 or a or b or c or d)
begin
    case ({s1,s2})
        'b00:  y=a;
        'b01:  y=b;
        'b10:  y=c;
        'b11:  y=d;
    endcase
end
endmodule
```

`case` 语句还有两种变体语句形式，即 `casez` 和 `casex` 语句。`casez` 和 `casex` 语句与 `case` 语句的格式完全相同，它们的区别是：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以考虑，只关注其他位的比较结果。在 `casex` 语句中，把不予以考虑的位扩展到未知 x ，即不考虑值为高阻 z 和未知 x 的那些位，只关注其他位的比较结果。在分支表达式中，可用“?”来标识 x 或 z 。用 `casez` 描述的数据选择器例子如下：

```
module mux_z(out,a,b,c,d,select);
```

```

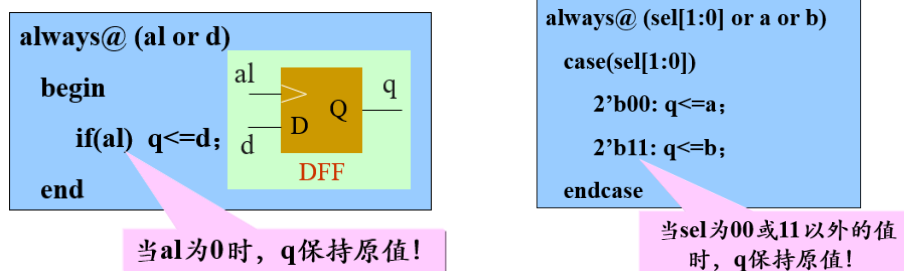
output out;
input a,b,c,d;
input[3:0] select;
reg out; //必须声明
always@ (select[3:0] or a or b or c or d)
begin
    casez (select)
        4'b???1: out = a;
        4'b??1?: out = b;
        4'b? 1?? : out = c;
        4'b 1??? : out = d; //用“?”来表示高阻态 z
    endcase
end
endmodule

```

3. 使用条件语句注意事项

应注意列出所有条件分支，否则当条件不满足时，编译器会生成一个锁存器保持原值！这一点可用于设计时序电路，如计数器：条件满足时加 1，否则保持原值不变。而在组合电路设计中，应避免生成隐含锁存器！有效的方法是在 if 语句最后写上 else 项；在 case 语句最后写上 default 项。

(1) 生成了不想要的锁存器例子：

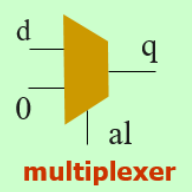


(2) 不会生成锁存器例子：

[例] 设计一个数据选择器

```

always@ (al or d)
begin
    if(al) q<=d;
    else q<=0;
end
        
```



当al为0时，q等于0！

[例] 设计一个数据选择器

```

always@ (sel[1:0] or a or b)
case(sel[1:0])
    2'b00: q<=a;
    2'b11: q<=b;
    default: q<='b0;
endcase
        
```

6.3.3 循环语句

循环语句包含 for 语句、repeat 语句、while 语句和 forever 语句 4 种。

for 语句——通过 3 个步骤来决定语句的循环执行：

- (1) 给控制循环次数的变量赋初值。
- (2) 判定循环执行条件，若为假则跳出循环；若为真，则执行指定的语句后，转到第（3）步。
- (3) 修改循环变量的值，返回第（2）步。

repeat 语句——连续执行一条语句 n 次。

while 语句——执行一条语句，直到循环执行条件不满足；若一开始条件即不满足，则该语句一次也不能被执行！

forever 语句——无限连续地执行语句，可用 disable 语句中断！

1. for 语句

for 语句的语法格式为：

```

for (索引变量 = 初值; 索引变量 < 终值; 索引变量 = 索引变量 + 步长值)
begin
    语句;
end
    
```

for 语句可以使一组语句重复执行，语句中的索引变量、初值、终值和步长值是循环语句定义的参数，这些参数一般属于整型变量或常量。语句重复执行的次数由语句中的参数确定，即

循环重复次数 = (终值 - 初值) / 步长值

【例 6.9】 8 位奇偶校验器的设计。

本例用 a 表示输入变量，它是一个长度为 8 位的向量。在程序中，用 for 语句对 a 的值，逐位进行模 2 加（即异或 XOR）运算，索引变量 n 控制模 2 加的

次数。索引变量的初值为 0，终值为 8，因此，控制循环共执行了 8 次。8 位奇偶校验器的 Verilog HDL 源程序 parity_checker8.v 如下：

```
module parity_checker8(a,out);
input[7:0]      a;
output         out;
reg            out;
integer        n; //定义整形索引变量
always
begin
    out = 0;
    for (n = 0; n < 8; n = n + 1)    out = out ^ a[n];
end
endmodule
```

对于一个具体的电路，可以有多种描述方法，例如，8 位奇偶校验器可以用缩减异或运算来实现，这种设计结果非常简单，其源程序 parity_checker8.v 如下：

```
module parity_checker8(a,out);
input[7:0]      a;
output         out;
reg            out;
assign out = ^ a;
endmodule
```

【例 6.10】用 for 语句描述的 7 人投票表决器：若超过 4 人（含 4 人）投赞成票，则表决通过。

```
module vote7 ( pass,vote );
output pass;
input [6:0] vote;
reg[2:0] sum;          //sum 为 reg 型变量，用于统计赞成的人数
integer i;
reg pass;
always @(vote)
begin
    sum = 0;                //sum 初值为 0
    for(i = 0;i<=6;i = i+1) //for 语句
        if(vote[i])    sum = sum+1; //只要有人投赞成票，则 sum 加 1
end
```

```

        if(sum[2])    pass = 1; //若超过 4 人赞成，则表决通过
        else        pass = 0;

    end

endmodule

```

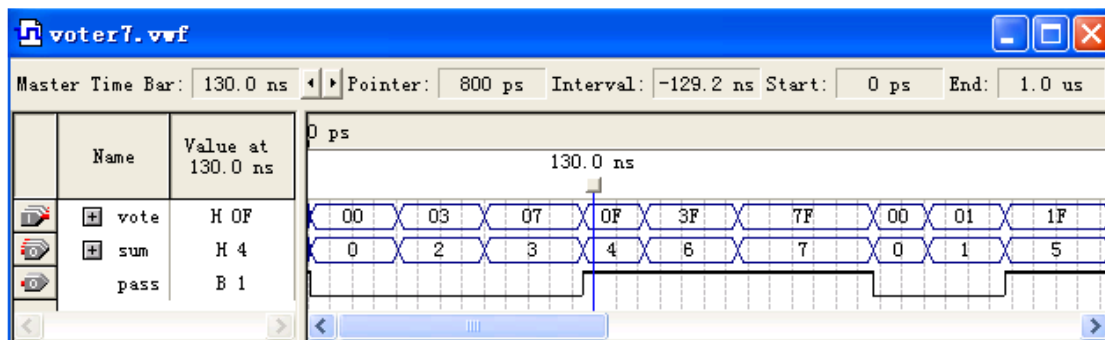


图 6.11 表决器仿真图

2. repeat 语句

repeat 语句的语法格式为：

repeat (循环次数表达式) 语句;

或 repeat (循环次数表达式)

begin

.....

end

【例 6.11】用 repeat 语句实现例 6.9（8 位奇偶校验器）的描述如下：

```
module parity_checker8 (a,out);
```

```
parameter    size = 7;
```

```
input[7:0]    a;
```

```
output        out;
```

```
reg            out;
```

```
integer        n;
```

```
always @(a)
```

```
begin
```

```
    out = 0;
```

```
    n = 0;
```

```
    repeat(size)
```

```
        begin
```

```
            out =out ^ a[n];
```

```
            n = n+1;
```

```

        end
    end
endmodule

```

请读者注意，有的 EDA 工具软件不支持 repeat 语句，因此将 repeat 视为非法语句。

3. while 语句

while 语句的语法格式为：

```

while (循环执行条件表达式)
begin
    重复执行语句;
    修改循环条件语句;
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，则执行其后的语句；若为假，则不执行（表示循环结束）。为了使 while 语句能够结束，在循环执行的语句中必须包含一条能改变循环条件的语句。

注 1：首先判断循环执行条件表达式是否为真，若不为真，则其后的语句一次也不被执行！

注 2：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！

注 3：while 语句只有当循环块有事件控制（即@（posedge clock））时才可综合！

【例 6.12】用 while 语句对一个 8 位二进制数中值为 1 的位进行计数。

```

module count1s_while ( count,rega,clk );
    output[3:0] count;
    input [7:0]   rega;
    input clk;
    reg[3:0]      count;
    always @(posedge clk)
        begin:count1
            reg[7:0] tempreg;           //用作循环执行条件表达式
            count = 0;                  // count 初值为 0
            tempreg = rega;              // tempreg 初值为 rega
            while(tempreg)               // 若 tempreg 非 0，则执行以下语句
                begin

```

```

        if(tempreg[0]) count = count+1;
            //只要 tempreg 最低位为 1，则 count 加 1
        tempreg = tempreg >>1; //右移 1 位，
            //改变循环执行条件表达式的值
    end
end
endmodule

```

思考：如何用 for 语句改写此程序呢？可以找到更简洁的设计方法。

4. forever 语句

forever 语句的语法格式为：

```

forever
begin
    语句;
end

```

forever 是一种无穷循环控制语句，它不断地执行其后的语句或语句块，永远不会结束。一般情况下 forever 语句是不可综合的！常用在测试文件中。forever 语句常用来产生周期性的波形，作为仿真激励变量，常用 disable 语句跳出循环！例如，产生时钟周期为 20 个延迟单位（纳秒）、占空比为 50% 的时钟脉冲 clk 的语句为：

```

initial
begin : Clocking
    clk = 0;
    #10 forever #10 clk = !clk;
end
initial
begin : Stimulus
    .....
    disable Clocking; // 停止时钟
end

```

注：不同于 always 语句，不能独立写在程序中，一般用在 initial 语句块中！

6.3.4 结构声明语句

Verilog HDL 的任何过程模块都是放在结构声明语句中，结构声明语句包括 `always`、`initial`、`task` 和 `function` 等 4 种结构。

`initial` 说明语句——只执行一次。

`always` 说明语句——不断重复执行，直到运行结束。

`task` 说明语句——可在程序模块中的一处或多处调用。

`function` 说明语句——可在程序模块中的一处或多处调用。

1. `always` 块语句

在一个 Verilog HDL 模块（`module`）中，`always` 块语句的使用次数是不受限制的，块内的语句也是不断重复执行的。`always` 块语句的语法结构为：

```
always @(敏感变量表达式)
begin
    // 过程赋值语句;
    // if 语句, case 语句;
    // for 语句, while 语句, repeat 语句;
    // task 语句、function 语句;
end
```

在 `always` 块语句中，敏感变量表达式（`event-expression`）应该列出影响块内取值的所有变量（一般指设计电路的输入变量或者是其他结构声明语句中的 `reg` 型变量），多个变量之间用“`or`”分隔（也可以用逗号“`,`”分隔）。当表达式中任何变量发生变化时，就会执行一遍块内的语句。块内语句可以包括：`过程赋值`、`if`、`case`、`for`、`while`、`repeat`、`task` 和 `function` 等语句。

在 `always` 块中被赋值的只能是 `register` 型变量（如 `reg`、`integer`、`real`、`time`）。

每个 `always` 块在仿真一开始便开始执行，当执行完块中最后一个语句，继续从 `always` 块的开头执行。

在进行时序逻辑电路的描述时中，敏感变量表达式中经常使用“`posedge`”和“`negedge`”这两个关键字来声明事件是由时钟的正边沿（上升沿）或负边沿（下降沿）触发的。若 `clk` 是设计电路的时钟变量，则“`always @(posedge clk)`”表示模块的事件是由 `clk` 的上升沿触发的；而“`always @(negedge clk)`”表示模块的事件是由 `clk` 的下降沿触发的，在 8 位二进制加法计数器（见例 6.2）的模块中，就使用了这类语句。

在进行组合逻辑电路的描述时中，敏感变量表达式中不使用边沿触发

“posedge”和“negedge”这两个关键字，而是使用电平触发，多个变量之间用“or”分隔（也可以用逗号“,”分隔）。

always 块语句是用于综合过程的最有用的语句之一，为得到最好的综合结果，always 块程序应严格按以下模板来编写：

①模板 1：

```
always @ (Inputs) //所有输入信号必须列出，用 or 隔开
begin
    .....           //组合逻辑关系
end
```

②模板 2：

```
always @ (Inputs) //所有输入信号必须列出，用 or 隔开
if (Enable)
begin
    .....           //锁存动作
end
```

③模板 3：

```
always @ (posedge Clock) // Clock only
begin
    .....           // 同步动作
end
```

④模板 4：

```
always @ (posedge Clock or negedge Reset)
// Clock and Reset only
begin
    if (! Reset)      // 测试异步复位电平是否有效
        .....       // 异步动作
    else
        .....       // 同步动作
end
// 可产生触发器和组合逻辑
```

注意：（1）当 always 块有多个敏感信号时，一定要采用 if - else if 语句，而不能采用并列的 if 语句！否则易造成一个寄存器有多个时钟驱动，将出现编译错误。

```
always @ posedge min_clk or negedge reset)
```

```

begin
    if (reset)
        min<=0;
    else if (min=8'h59) //当 reset 无效且 min=8'h59 时，不能写成 if
        begin
            min<=0;h_clk<=1;
        end
    end
end

```

(2) 通常采用异步清零！只有在时钟周期很小或清零信号为电平信号时（容易捕捉到清零信号）采用同步清零，即在敏感列表中不写 `negedge reset` 信号。

2. initial 语句

`initial` 语句的语法格式为：

```

initial
begin
    语句 1;
    语句 2;
    ...;
end

```

`initial` 语句的使用次数也是不受限制的，其特点与 `always` 语句相同，不同之处在于其块内的语句仅执行一次（不重复），因此 `initial` 语句常用于设计电路的初始化数据设置和仿真中。

(1) 利用 `initial` 语句生成激励波形

```

initial
begin
    inputs = 'b000000;
    #10 inputs = 'b011001;
    #10 inputs = 'b011011;
    #10 inputs = 'b011000;
    #10 inputs = 'b001000;
end

```

(2) 对各变量进行初始化

```

.....
parameter size=16;

```

```

reg[3:0] addr;
reg reg1;
reg[7:0] memory[0:15];
initial
begin
    reg1 = 0;
    for(addr=0;addr<size;addr=addr+1);
        memory[addr]=0; //初始化 ROM 存储器
    end
.....

```

3. task 语句

在 Verilog HDL 模块中，task 语句用来定义任务。任务类似高级语言中的子程序，用来单独完成某项具体任务，并可以被模块或其他任务调用。利用任务可以把一个大的程序模块分解成为若干小的任务，使程序清晰易懂，而且便于调试。

当希望能够对一些信号进行一些运算并输出多个结果（即有多个输出变量）时，宜采用任务结构。常常利用任务来帮助实现结构化的模块设计，将批量的操作以任务的形式独立出来，使设计简单明了。

可以被调用的任务必须事先用 task 语句定义，定义格式如下：

```

task 任务名;
    端口声明语句;
    类型声明语句;
begin
    语句;
end
endtask

```

任务定义与模块（module）定义的格式相同，区别在于是任务用 task-endtask 语句来定义，而且没有端口名列表。例如，8 位加法器任务的定义如下：

```

task adder8;
    output[7:0] sum;
    output      cout;
    input[7:0]  ina,inb;
    input       cin;

```

```
{cout,sum}=ina+inb+cin;
```

```
endtask
```

任务调用的格式如下：

任务名 (端口名列表);

例如，8 位加法器任务调用语句如下：

```
adder8 (tsum,tcout,tina,tinb,tcin);
```

【例 6.13】完整的 8 位加法器任务调用的源程序如下：

```
module adder_8(a,b,cin,sum,cout);
    input[7:0]  a,b;
    input      cin;
    output[7:0] sum;
    output     cout;
always
    begin
        adder8(a,b,cin,sum,cout); //任务调用
    end

task adder8;          //没有端口名列表
    input[7:0]  a,b;
    input      cin;
    output[7:0] sum;
    output     cout;
    begin
        {cout,sum}=a+b+cin;
    end
endtask
endmodule
```

使用任务时，需要注意几点：

① 任务的定义和调用必须在同一个 module 模块内，任务调用语句应在 always 或 task-endtask 块中。

② 定义任务时，没有端口名列表，但要在进行端口和数据类型的声明，任务用顺序语句完成功能的描述。

③ 当任务被调用时，任务被激活。任务调用与模块调用一样，通过任务名实现，调用时需列出端口名列表，端口名和类型必须与任务定义中的排序和类型

一致。例如，8 位加法器任务调用时的端口名列表中的 tsum、tcout、tina、tinb、tcin 端口，与任务定义中的端口 sum、cout、ina、inb、cin 排序和类型保持一致。

④ 一个任务可以调用别的任务或函数，可调用的任务和函数的个数不受限制。

任务的作用是方便编程，但模块（module）也具有任务的性质，可以被其他模块调用（调用方法与任务相同），而且模块还可以独立存在（任务只能包含在模块中），因此在电路设计的编程中，可以用模块替代任务。

4. function 语句

在 Verilog HDL 模块中，function 语句用来定义函数。函数类似高级语言中的函数，用来单独完成某项具体操作，并可以作为表达式中的一个操作数，被模块或任务以及其他函数调用，函数调用时返回一个用于表达式的值。

函数的目的是通过返回一个用于某表达式的值，来响应输入信号。适于对不同变量采取同一运算的操作。函数在模块内部定义，通常在本模块中调用，也能根据按模块层次分级命名的函数名从其他模块调用。而任务只能在同一模块内定义与调用！

可以被调用的函数必须事先用定义，函数定义格式如下：

function [最高有效位:最低有效位] 函数名;

端口声明语句;

类型声明语句;

begin

语句;

end

endfunction

在函数定义语句中，“[最高有效位:最低有效位]”是函数调用返回值位宽或类型声明。

【例 6.14】求最大值的函数。

```
function [7:0] max;
```

```
input[7:0] a,b;
```

```
begin
```

```
if (a>=b) max=a;
```

```
else max=b;
```

```
end
```

```
endfunction
```

函数调用的格式如下：

函数名 (关联参数表);

函数调用一般是出现在模块、任务或函数语句中。通过函数的调用来完成某些数据的运算或转换。例如，调用例 6.9 编制的求最大值的函数

`Peak = max(data,peak);`

其中，`data` 和 `peak` 是与函数定义的两个参数 `a`、`b` 关联的关联参数。通过函数的调用，求出 `data` 和 `peak` 中的最大值，并用函数名 `max` 返回。

函数的定义不能包含任何时间控制语句——用延迟#、事件控制@或等待wait标识的语句。函数不能启动（即调用）任务！定义函数时至少要有一个输入参量！且不能有任何输出或输入/输出双向变量。在函数的定义中必须有一条赋值语句，给函数中的一个内部寄存器赋以函数的结果值，该内部寄存器与函数同名。

【例 6.15】 利用函数对一个 8 位二进制数中为 0 的位进行计数

```
module count0s_function(number,rega);
    output[7:0]number;
    input[7:0]rega;

    function[7:0]gefun; //定义函数
        input[7:0]x;
        reg[7:0]count;
        integer i;
        begin
            count=0;
            for(i=0;i<7;i=i+1)
                if(x[i]==1'b0)count=count+1;
            gefun=count; //返回函数值
        end
    endfunction

    assign number=gefun(rega);//调用函数
endmodule
```

函数和任务存在以下几处区别：

① 任务可以有任意不同类型输入/输出变量，函数不能将 `inout` 类型作为输出。

② 任务只可以在过程语句中调用，不能在连续赋值语句 `assign` 中调用；函数可以作为表达式中的一个操作数，在过程赋值语句和连续赋值语句中调用。

③ 任务可以调用其他任务或函数；函数可以调用其他函数，但不能调用任务。

④ 任务不向表达式返回值，函数向调用它的表达式返回一个值。

6.3.5 编译预处理语句

“编译预处理”是 Verilog HDL 编译系统的一个组成部分。编译预处理语句以西文符号“```”开头——注意，不是单引号“`'`”！

在编译时，编译系统先对编译预处理语句进行预处理，然后将处理结果和源程序一起进行编译。

1、``define` 语句

宏定义语句——用一个指定的标志符（即宏名）来代表一个字符串（即宏内容）。

格式：``define` 标志符（即宏名）字符串（即宏内容）

[例] ``define IN ina+inb+inc+ind`

宏展开——在编译预处理时将宏名替换为字符串的过程。

宏定义的作用：

以一个简单的名字代替一个长的字符串或复杂表达式；

以一个有含义的名字代替没有含义的数字和符号。

宏定义使用说明：

① 宏名可以用大写字母，也可用小写字母表示；但建议用大写字母，以与变量名相区别。

② ``define` 语句可以写在模块定义的外面或里面。宏名的有效范围为定义命令之后到源文件结束。

③ 在引用已定义的宏名时，必须在其前面加上符号“```”！

④ 使用宏名代替一个字符串，可简化书写，便于记忆，易于修改。

⑤ 预处理时只是将程序中的宏名替换为字符串，不管含义是否正确。只有在编译宏展开后的源程序时才报错。

⑥ 宏名和宏内容必须在同一行中进行声明！

⑦ 宏定义不是 Verilog HDL 语句，不必在行末加分号！如果加了分号，会连分号一起置换！

[示例] `module test;`

```

reg a,b,c,d,e,out;
`define expression a + b + c + d;
assign out = `expression + e;

.....

```

经过宏展开后，assign 语句为：

```
assign out = a + b + c + d; + e;    //出现语法错误！
```

⑧在进行宏定义时，可引用已定义的宏名，实现层层置换。

[示例] module test;

```

reg a,b,c;
wire out;
`define aa a + b
`define cc c + `aa    //引用已定义的宏名`aa 来定义宏 cc
assign out = `cc;

.....

```

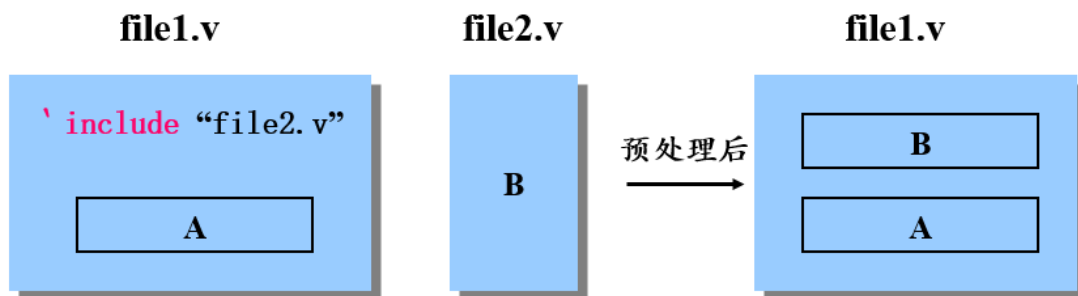
经过宏展开后，assign 语句为：

```
assign out = c + a + b;
```

2、`include 语句

文件包含语句——一个源文件可将另一个源文件的全部内容包含进来。

格式：`include “文件名”



预处理后，将 file2.v 中全部内容复制插入到 `include “file2.v” 命令出现的地方。

使用包含语句的好处：

避免程序设计人员的重复劳动！不必将源代码复制到自己的另一源文件中，使源文件显得简洁。

(1) 可以将一些常用的宏定义命令或任务 (task) 组成一个文件，然后用 `include 语句将该文件包含到自己的另一源文件中，相当于将工业上的标准元件

拿来使用。

(2) 当某几个源文件经常需要被其他源文件调用时，则在其他源文件中用 ``include` 语句将所需源文件包含进来。

[示例] 用 ``include` 语句设计 16 位加法器

```
'include"adder.v"

module adder16(cout,sum,a,b,cin); //16 位加法器
    output    cout;
    parameter my_size=16;
    output[my_size-1:0] sum;
    input[my_size-1:0]  a,b;
    input              cin;
    adder #(my_size) my_adder(cout,sum,a,b,cin); //调用加法器模块
endmodule
```

改变被引用模块 `adder` 中的参数 `size` 为 `my_size`。

adder 模块:

```
module adder(cout,sum,a,b,cin);
    output    cout;
    parameter size=1;
    output[size-1:0] sum;
    input[size-1:0]  a,b;
    input          cin;
    assign{cout,sum}=a+b+cin;
endmodule
```

关于文件包含的说明:

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

``include "aaa.v" "bbb.v" //非法!`

``include "aaa.v"`

``include "bbb.v" //合法!`

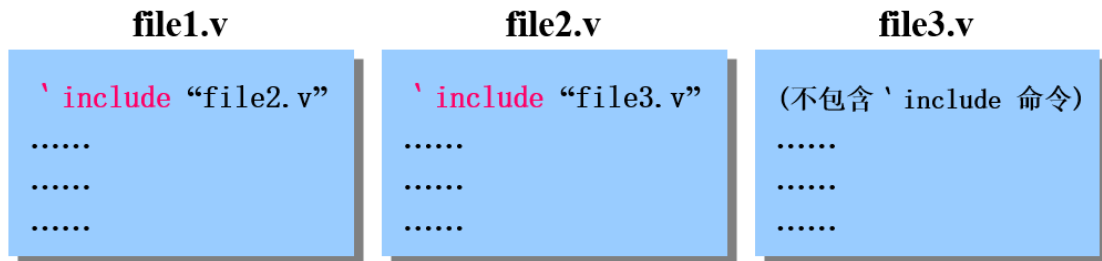
(2) ``include` 语句可出现在源程序的任何地方。被包含的文件若与包含文件不在同一子目录下，必须指明其路径！

``include "parts/count.v" //合法!`

(3) 可将多个 ``include` 语句写在一行；在该行中，只可出现空格和注释行。

``include "aaa.v" `include "bbb.v" //合法！`

(4) 文件包含允许嵌套。



3、`timescale 语句

时间尺度语句——用于定义跟在该命令后模块的时间单位和时间精度。

MAX + PLUS II 和 Quartus II 都不支持！通常用在测试文件中。

格式： ``timescale <时间单位> / <时间精度>`

时间单位——用于定义模块中仿真时间和延迟时间的基准单位。

时间精度——用来声明该模块的仿真时间和延迟时间的精确程度。

在同一程序设计里，可以包含采用不同时间单位的模块。此时用最小的时间精度值决定仿真的时间单位。

时间精度至少要和时间单位一样精确，时间精度值不能大于时间单位值！

``timescale 1ps / 1ns` // 非法！

``timescale 1ns / 1ps` // 合法！

在 ``timescale` 语句中，用来说明时间单位和时间精度参量值的数字必须是整数。其有效数字为 1、10、100。单位为秒 (s)、毫秒 (ms)、微秒 (us)、纳秒 (ns)、皮秒 (ps)、毫皮秒 (fs)。

[示例] ``timescale` 语句应用举例

``timescale 10ns / 1ns` // 时间单位为 10ns，时间精度为 1ns。

.....

`reg sel;`

`initial`

`begin`

`#10 sel = 0; // 在 10ns×10 时刻，sel 变量被赋值为 0`

`#10 sel = 1; // 在 10ns×20 时刻，sel 变量被赋值为 1`

`end`

.....

6.3.6 语句的顺序执行与并行执行

Verilog HDL 中有顺序执行语句和并行执行语句之分。Verilog HDL 的 `always` 块语句与 VHDL 的 `PROCESS`（进程）语句类似，块中的语句是顺序语句，按照程序书写的顺序执行。`always` 块本身却是并行语句，它与其他 `always` 语句以及 `assign` 语句、例化元件语句和 `initial` 语句都是同时执行（即并行）的。由于 `always` 语句的并行行为和顺序行为的双重特性，所以使它成为 Verilog HDL 程序中使用最频繁和最能体现 Verilog HDL 风格的一种语句。

（1）、语句的顺序执行

在“`always`”模块内，逻辑按书写的顺序执行。

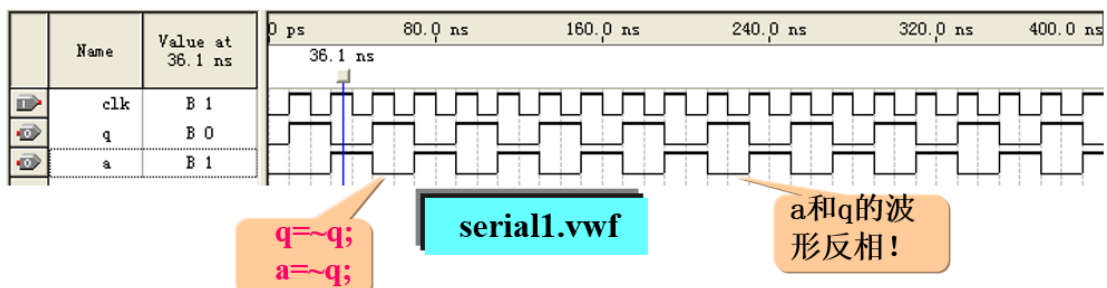
顺序语句——“`always`”模块内的语句。

在“`always`”模块内，若随意颠倒赋值语句的书写顺序，可能导致不同的结果！
注意阻塞赋值语句当本语句结束时即完成赋值操作！

顺序执行模块 1:

```
module serial1(q,a,clk);
    output q,a;
    input clk;
    reg q,a;
    always @(posedge clk)
        begin
            q=~q; //阻塞赋值语句
            a=~q;
        end
endmodule
```

a 和 q 的波形反相！



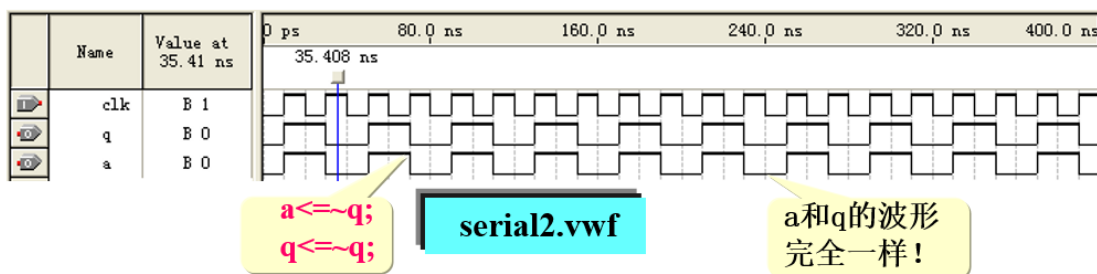
顺序执行模块 2:

```
module serial2(q,a,clk);
    output q,a;
```

```

input clk;
reg q,a;
always @(posedge clk)
begin
    a<=~q; //非阻塞赋值语句
    q<=~q;
end
endmodule
a 和 q 的波形完全相同！

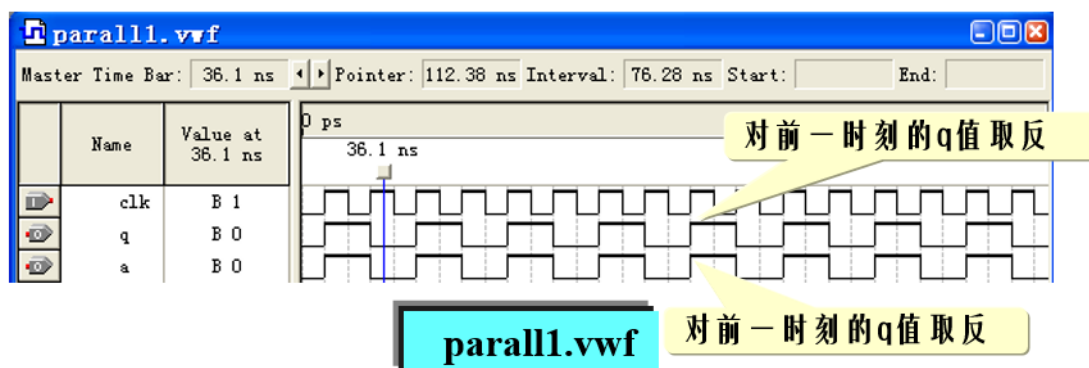
```



(2)、语句的并行执行

“always”模块、“assign”语句、实例元件都是同时（即并行）执行的！它们在程序中的先后顺序对结果并没有影响。

下面将两条赋值语句分别放在两个“always”模块中，尽管两个“always”模块顺序相反，但仿真波形完全相同，q 和 a 的波形完全一样。



并行执行模块1。

```
module parall1(q,a,clk);
    output q,a;
    input clk;
    reg q,a;
    always @(posedge clk)
        begin
            q=~q;
        end
    always @(posedge clk)
        begin
            a=~q;
        end
endmodule
```

并行执行模块2。

```
module parall2(q,a,clk);
    output q,a;
    input clk;
    reg q,a;
    always @(posedge clk)
        begin
            a=~q;
        end
    always @(posedge clk)
        begin
            q=~q;
        end
endmodule
```

always 块语句中有一个敏感变量表，表中列出的任何变量的改变，都将启动 always 块语句，使 always 块语句内相应的顺序语句被执行一次。实际上，用 Verilog HDL 描述的硬件电路的全部输入变量都是敏感变量，为了使 Verilog HDL 的软件仿真与综合和硬件仿真对应起来，应当把 always 块语句中所有输入变量都列入敏感变量表中。在时序逻辑电路的编程中，由于时钟变量（clk）和复位变量（clr）是电路变化的主要条件，因此在敏感变量表中，仅列出 clk 或 clr 就可以了（其他电平型变量可以不列出）。

【例 6.16】同步清除十进制加法计数器的描述。

同步清除是指复位变量有效，而且时钟变量的有效边沿到来时，计数器的状态被清 0。在本例中，复位变量是 clr，高电平有效；时钟变量是 clk，上升沿是有效边沿。当 clk 的上升沿到来时，如果 clr 清除变量有效（为 1），则计数器被清 0；clr 无效时，如果计数器原态是 9，计数器回到 0 态，否则计数器的状态将加 1。同步清除十进制加法计数器的 Verilog HDL 的源程序 example_10.v 如下：

```
module example_10(clr,clk,q,cout);
    input          clr,clk;
    output[3:0]    q;
    output         cout;
    reg[3:0]      q;
    reg           cout;
```

```

always @(posedge clk)
begin
    if (clr) q=0;
    else begin
        if (q==9) q=0;
        else q=q+1;
        if(q==0) cout=1;
        else cout=0;end
    end
endmodule

```

本设计的计数器的仿真波形如图 6.7 所示，从仿真结果可以看到，当 `clk` 的上升沿到来时，若 `clr` 有效（为高电平），计数器被清 0。仿真结果验证了设计的正确性。

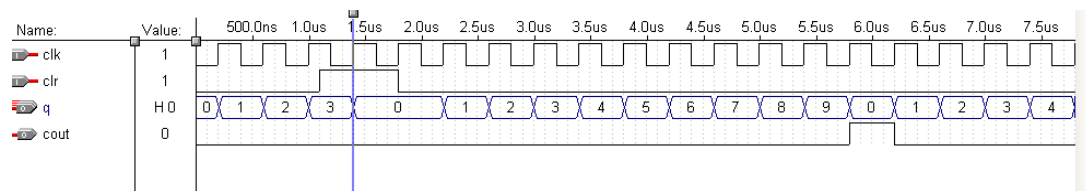


图 6.7 同步清除十进制加法计数器的仿真波形

【例 6.17】异步清除十进制加法计数器的描述。

异步清除是指复位变量有效后，计数器的状态立即被清 0，与时钟变量无关。在本例中，复位变量是 `clr`，上升沿有效；时钟变量是 `clk`，上升沿是有效边沿。当 `clr` 的上升沿到来时，计数器被清 0；`clr` 无效时，当时钟 `clk` 的上升沿到来后，如果计数器原态是 9，计数器回到 0 态，否则计数器的状态将加 1。异步清除十进制加计数器的 Verilog HDL 的源程序 `example_11.v` 如下：

```

module example_11(clr,clk,q,cout);
input      clr,clk;
output[3:0] q;
output      cout;
reg[3:0]    q;
reg         cout;
always @(posedge clk or posedge clr)
begin
    if (clr) q=0;

```

```

else begin
    if (q==9) q=0;
    else q=q+1;
    if(q==0) cout=1;
    else cout=0;end

end

```

`endmodule` 请读者注意同步清除十进制加法计数器的源程序 (`example_10.v`) 与异步清除十进制加法计数器的源程序 (`example_11.v`) 的区别, 在 `example_10.v` 源程序中, 复位变量 `clr` 不包含在 `always` 块的敏感变量表中, 因此只有在时钟 `clk` 的上升沿到来时, 复位语句才能执行, 构成同步复位; 而在 `example_11.v` 源程序中, 复位信号 `clr` 也包含在 `always` 块的敏感变量表中, 因此不受时钟信号的制约, 当 `clr` 的上升沿到来时刻电路立即被清除, 构成异步复位电路。

异步清除十进制加法计数器的仿真波形如图 6.8 所示, 从图中可以看出复位信号与时钟信号无关, 仿真结果验证了设计的正确性。

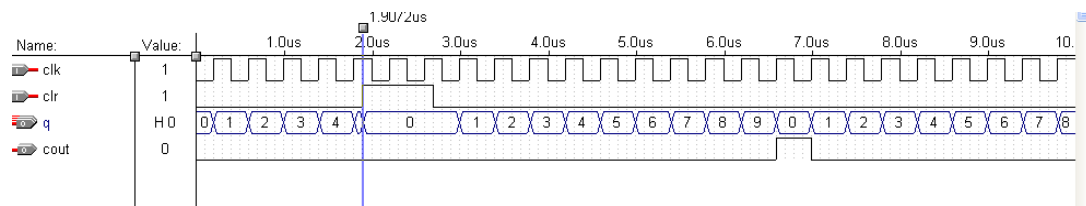


图 6.8 异步清除十进制加法计数器的仿真波形

6.4 不同抽象级别的 Verilog HDL 模型

Verilog HDL 是一种用于逻辑电路设计的硬件描述语言。用 Verilog HDL 描述的电路称为该设计电路的 Verilog HDL 模型。

Verilog HDL 具有行为描述和结构描述功能。行为描述是对设计电路的逻辑功能的描述, 并不用关心设计电路使用哪些元件以及这些元件之间的连接关系。行为描述属于高层次的描述方法, 在 Verilog HDL 中, 行为描述包括系统级 (System Level)、算法级 (Algorithm Level) 和寄存器传输级 (RTL: Register Transfer Level) 等 3 种抽象级别。

结构描述是对设计电路的结构进行描述, 即描述设计电路使用的元件及这些元件之间的连接关系。结构描述属于低层次的描述方法, 在 Verilog HDL, 结构描述包括门级 (Gate Level) 和开关级 (Switch Level) 两种抽象级别。

在 Verilog HDL 的学习中, 应重点掌握高层次的行为描述方法, 但结构描述也可以用来实现电路的系统设计。对于一个实际的数字系统电路, 一般先用行为

描述方法设计底层模块电路，最后用结构描述方法将各模块连接起来，构成顶层文件完成系统电路的设计。

6.4.1 Verilog HDL 门级描述

Verilog HDL 提供了丰富的门类型关键字，用于门级的描述。比较常用的包括：**not**（非门）、**and**（与门）、**nand**（与非门）、**or**（或门）、**nor**（或非门）、**xor**（异或门）、**xnor**（异或非门）、**buf**（缓冲器）以及 **bufif1**、**bufif0**、**notif1**、**notif0** 等各种三态门。

门级描述语句格式为：

门类型关键字 [例化门的名称] (端口列表);

其中，“例化门的名称”是用户定义的标识符，属于可选项；端口列表按：（输出，输入，使能控制端）的顺序列出。例如

`nand nand2(y,a,b);` //2 输入端与非门

`xor myxor(y,a,b);` //异或门

`bufif0 mybuf(y,a,en);` //低电平使能的三态缓冲器

【例 6.18】采用结构描述方式描述图 6.9 所示的硬件电路。

在结构描述中，需要声明电路内部使用的连线，其中 `s1`、`s2`、`s3` 是线型变量连线，用结构描述方式的 Verilog HDL 源程序 `example_12.v` 如下：

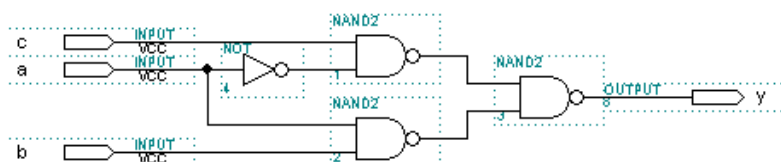


图 6.9 例 6.18 的硬件实现电路

```
module example_18(y,a,b,c);
input  a,b,c;
output y;
wire    s1,s2,s3;
not      (s1,a);
nand     (s2,c,s1);
nand     (s3,a,b);
nand     (y,s2,s3);
endmodule
```


6.4.2 Verilog HDL 的行为级描述

Verilog HDL 的行为级描述是最能体现 EDA 风格的硬件描述方式，它既可以描述简单的逻辑门，也可以描述复杂的数字系统乃至微处理器；既可以描述组合逻辑电路，也可以描述时序逻辑电路。下面再通过几个组合逻辑和时序逻辑电路设计例子，来加深读者对 Verilog HDL 行为级描述方法的理解。

【例 6.19】逻辑功能描述—算法级：用逻辑表达式实现 4 选 1 数据选择器。

```
module mux4_1(out,in1,in2,in3,in4,cntrl1,cntrl2);
    output out;
    input in1,in2,in3,in4,cntrl1,cntrl2;
    assign out=(in1 & ~cntrl1 & ~cntrl2) |
               (in2 & ~cntrl1 & cntrl2) |
               (in3 & cntrl1 & ~cntrl2) |
               (in4 & cntrl1 & cntrl2);
endmodule
```

注：采用算法级描述首先必须根据逻辑功能写出逻辑表达式！再根据表达式进行功能设计。

【例 6.20】条件运算符描述——算法级：用条件运算符描述 4 选 1 数据选择器。只需知道输入与输出间的真值表！

```
module mux4_1(out,in1,in2,in3,in4,cntrl1,cntrl2);
    output out;
    input in1,in2,in3,in4,cntrl1,cntrl2;
    assign out= cntrl1? (cntrl2? in4 :in3) : (cntrl2? in2 :in1);
endmodule
```

注：比调用门原语，采用逻辑表达式或 case 语句描述代码更简单！但也更抽象！且耗用器件资源更多！

【例 6.21】case 语句描述—系统级：用 case 语句描述 4 选 1 数据选择器。

系统级行为描述只需知道输入与输出间的真值表！比调用门级描述和采用逻辑功能描述都简洁！case 语句应放在 always 块内！

```
module mux4_1(out,in1,in2,in3,in4,cntrl1,cntrl2);
    output out;
    input in1,in2,in3,in4,cntrl1,cntrl2;
    reg out;
    always @(in1 or in2 or in3 or in4 or cntrl1 or cntrl2)
```

```

        case ({cntrl1,cntrl2})
            2'b00:out=in1;
            2'b01:out=in2;
            2'b10:out=in3;
            2'b11:out=in4;
            default :out=1'bx;
        endcase
    endmodule

```

【例 6.22】if 语句描述—系统级：3 线-8 线译码器的设计。

3 线-8 线译码器设计电路的元件符号如图 6.10 所示，en 是低电平有效的使能控制输入端，a，b，c 是数据输入端，y 是 8 位数据输出端。

3 线-8 线译码器设计电路的 Verilog HDL 源程序 decoder3_8.v 如下：

```

module decoder3_8 (a,b,c,y,en);
input      a,b,c,en;
output[7:0] y;
reg[7:0]   y;
always     @(en or a or b or c)
begin
    if (en)  y = 'b11111111;
    else
        begin
            case({c,b,a})
                'b000:  y = 'b11111110;
                'b001:  y = 'b11111101;
                'b010:  y = 'b11111011;
                'b011:  y = 'b11110111;
                'b100:  y = 'b11101111;
                'b101:  y = 'b11011111;
                'b110:  y = 'b10111111;
                'b111:  y = 'b01111111;
            endcase
        end
    end
end

```

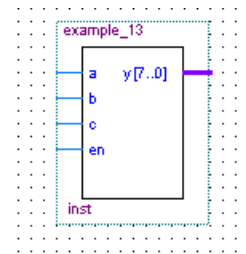


图 6.10 3 线-8 线译码器的元件符号

```

        end
    end
endmodule

```

【例 6.23】if 语句描述—系统级：8D 锁存器的设计。

8D 锁存器设计电路的元件符号如图 6.11 所示，其中 d[7..0] 是 8 位数据输入端，q[7..0] 是 8 位数据输出端，en 是使能控制输入端，当 en=0（无效）时，锁存器的状态不变，en=1（有效）时，q[7..0]=d[7..0]。

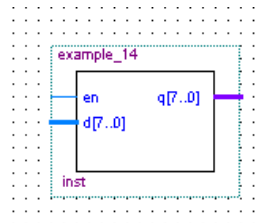


图 6.11 8D 锁存器的元件符号

8D 锁存器设计电路的 Verilog HDL 源程序 latch8.v 如下：

```

module latch8(d,q,en);
input      en;
input[7:0]  d;
output[7:0] q;
reg[7:0]    q;
always      @(en or d)
begin
    if (~en) q = q;
    else
        q=d;
    end
endmodule

```

6.4.3 用结构描述实现电路系统设计

任何用 Verilog HDL 描述的电路设计模块（module），均可用模块例化语句，例化一个元件，来实现电路系统的设计。

模块例化语句格式与逻辑门例化语句格式类似，具体格式如下：

设计模块名 [例化电路名](端口列表);

其中，“设计模块名”是用户设计的电路模块名，相当于设计电路中的一个元件；“例化电路名”是用户为系统设计定义的标识符（为可选项），相当于系统电路板上为插入设计模块元件的插座；而“端口列表”用于描述设计模块元件上的引脚与插座上引脚的连接关系。

端口列表的方法有两种:

① 位置关联法。要求端口列表中的引脚名称应与设计模块的输入/输出端口一一对应。例如,设计模块名为 `cnt10` 的输入/输出端口为 `clk` 和 `cout`,而以 `u1` 为例化电路名的两个引脚名是 `x1` 和 `x2`,那么位置关联法的模块例化语句格式为:

```
cnt10 u1(x1,x2);
```

② 名称关联法。名称关联法的格式如下:

(.设计模块端口名(插座引脚名),.设计模块端口名(插座引脚名),...);

例如,用名称关联法完成 `cnt10` 的模块例化语句格式为:

```
cnt10 u1(.clk(x1),.cout(x2));
```

两种关联法各有特点,位置关联法简单,但没有名称关联法直观。

【例 6.24】用模块例化方式设计 8 位计数译码器电路系统。

在 8 位计数译码系统电路设计中,需要事先设计一个 4 位二进制加法计数器 `cnt4e` 模块和一个七段数码显示器的译码器 `dec7s` 模块,然后用模块例化方式将这两种模块组成计数译码系统电路。

1.4 位二进制加法计数器 `cnt4e` 的设计

`cnt4e` 的元件符号如图 6.12 所示,`clk` 是时钟输入端;`clr` 是复位控制输入端,当 `clr=1` 时计数器被复位,输出 `q[3..0]=0000`; `ena` 是使能控制输入端,当 `ena=1` 时,计数器才能工作; `cout` 是进位输出端,当输出 `q[3..0]='b1111` 时, `cout=1`。

`cnt4e` 的 Verilog HDL 源程序 `cnt4e.v` 如下:

```
module cnt4e (clk,clrn,ena,cout,q);
    input          clk,clrn,ena;
    output reg[3:0] q;
    output reg      cout;
    always @(negedge clrn or posedge clk)
    begin
        if (~clrn)    q = 0;
        else begin
            if (ena) q = q+1;
            if (q==15) cout = 1;
            else    cout=0; end
        end
    end
endmodule
```

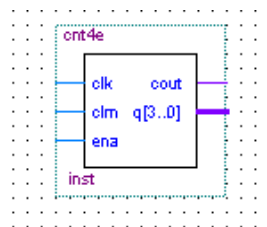


图 6.12 `cnt4e` 的元件符号

2. 七段数码显示器的译码器 dec7s 的设计

dec7s 的元件符号如图 6.13 所示, a[3..0]是 4 数据输入端, 将接至 cnt4e 的输出端 q[3..0]; q[7..0]是译码器的输出端, 提供七段数码显示数据。

dec7s 的 Verilog HDL 源程序 Dec7s.v 如下:

```
module dec7s(a,q);
    input [3:0]    a;
    output reg[7:0] q;
always @(a)
begin
    case(a)
        0: q='b00111111;      1: q='b00000110;
        2: q='b01011011;      3: q='b01001111;
        4: q='b01100110;      5: q='b01101101;
        6: q='b01111101;      7: q='b00000111;
        8: q='b01111111;      9: q='b01101111;
        10: q='b01110111;     11: q='b01111100;
        12: q='b00111001;     13: q='b01011110;
        14: q='b01111001;     15: q='b01110001;
    endcase
end
endmodule
```

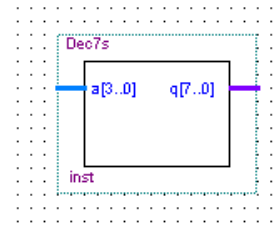


图 6.13 Dec7s 的元件符号

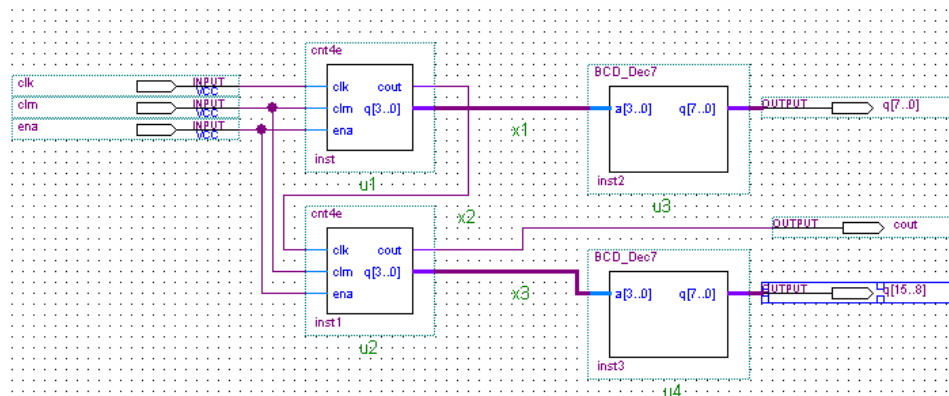


图 6.14 计数译码系统电路的结构图

3. 计数译码系统电路的设计

计数译码系统电路的结构图如图 6.14 所示，它是用 Quartus II 的图形编辑方式设计出来的。其中 u1 和 u2 是两个 cnt4e 元件的例化模块名，相当 cnt4e 系统电路板上的插座；u3 和 u4 是 Dec7s 元件的例化模块名，相当 Dec7s 在系统电路板上的插座。x1、x2、x3 是电路内部的连线。

用模块例化方式将 cnt4e 和 Dec7s 两种模块组成计数译码系统电路，用 Verilog HDL 位置关联法描述的计数译码电路的顶层源程序文件 cnt_dec7s.v 如下：

```
module cnt_dec7s(clk,clrn,ena,q,cout);
    input      clk,clrn,ena;
    output [15:0] q;
    output      cout;
    wire        x1,x2,x3;
    cnt4e u1(clk,clrn,ena,x2,x1);
    cnt4e u2(x2,clrn,ena,cout,x3);
    dec7s u3(x1,q[7:0]);
    dec7s u4(x3,q[15:8]);
endmodule
```

用 Verilog HDL 名称关联法描述的计数译码电路的顶层源程序文件 cnt_dec7s_1.v 如下：

```
module cnt_dec7s_1(clk,clrn,ena,q,cout);
    input      clk,clrn,ena;
    output [15:0] q;
    output      cout;
    wire        x1,x2,x3;
    cnt4e u1(.q(x1),.cout(x2),.clk(clk),.clrn(clrn),.ena(ena));
    cnt4e u2(.q(x3),.cout(cout),.clk(clk),.clrn(clrn),.ena(x2));
    dec7s u3(.a(x1),.q(q[7:0]));
    dec7s u4(.a(x3),.q(q[15:8]));
endmodule
```

计数译码系统电路的仿真波形如图 6.15 所示，其中数据“3F3F”是电路输出端 q[15:0]送给七段数码管显示“00”的数据；“3F06”是显示“01”的数据；

依此类推。仿真结果验证了设计的正确性。

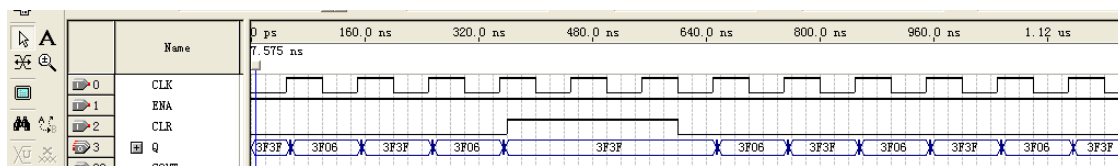


图 6.15 计数译码系统电路的仿真波形

6.4.4 Verilog HDL 模型设计总结

1、小结

采用的描述级别越高,设计越容易,程序代码越简单;但耗用器件资源更多。
对特定综合器,可能无法将某些抽象级别高的描述转化为电路!

基于门级描述的硬件模型不仅可以仿真,而且可综合,且系统速度快。

所有 Verilog HDL 编译软件只是支持该语言的一个子集。

尽量采用编译软件支持的语句来描述设计;或多个软件配合使用。

一般用算法级(写出逻辑表达式)或 RTL 级来描述逻辑功能,尽量避免用门级描述,除非对系统速度要求比较高的场合才采用门级描述。

2、思考

(1) 采用什么描述级别更合适?

系统级描述太抽象,有时无法综合成具体的物理电路;门级描述要求根据逻辑功能画出逻辑电路图,对于复杂的数字系统很难做到;

而算法级和 RTL 级描述级别适中,代码不是很复杂,且一般容易综合成具体的物理电路,故建议尽量采用算法级和 RTL 级来描述。

(2) 怎样减少器件逻辑资源的耗用?

当器件容量有限时,为减少器件逻辑资源的耗用,建议少用 if-else 语句和 case 语句,尽量直接使用逻辑表达式来描述系统的逻辑功能;

或者用 case 语句取代 if-else 语句。

3、建议:

(1) 在进行设计前,一定要仔细分析并熟悉所需设计电路或系统的整个工作过程;合理划分功能模块;并弄清每个模块输入和输出间的逻辑关系!

(2) 在调试过程中,仔细阅读并理解错误信息,随时查阅教材和课件上有关语法,纠正语法错误。

6.5 Verilog HDL 设计流程

Verilog HDL 的设计流程与原理图输入法设计流程基本相同，关于 Quartus II 软件平台的使用方法，在前面章节中已经作过比较详细的介绍，下面仅以 BCD 数加法器电路为例，简要介绍 Verilog HDL 的设计流程。

BCD 数加法器电路的设计包括 BCD_adder.v、BCD_Dec7.v 和 bcd_dec.bdf 三个模块，其中 BCD_adder.v 和 BCD_Dec7.v 是用 Verilog HDL 编写的 BCD 加法器和共阴极七段显示译码器源程序，bcd_dec.bdf 则是以原理图输入法设计的顶层文件。在 bcd_dec.bdf 原理图中以 BCD_adder.v 和 BCD_Dec7.v 作为元件，设计一个 BCD 数加法器电路。设计前应为设计建立一个工程目录（如 D:\myeda\v），用于存放 Verilog HDL 设计文件。

6.5.1 编辑 Verilog HDL 源程序

在 Quartus II 集成环境下，首先 BCD 数加法器设计电路建立一个新工程（New Project Wizard），然后为执行“File”菜单的“New”命令，弹出如图 6.16 所示的打开新文件对话框，选择对话框中的“Verilog HDL File”文件类型，进入 Verilog HDL 文本编辑方式。

1. 编辑 BCD 数加法器的 Verilog HDL 源程序

进入文本编辑方式后，编辑 BCD 数加法器的 Verilog HDL 源程序，并以 BCD_adder.v 为源程序的文件名，保存在 D:\myeda\v 工程目录中，后缀为.v 表示 Verilog HDL 源程序文件。注意：

Verilog HDL 源程序的文件名应与设计模块名相同，否则将是一个错误，无法通过编译。BCD 数加法器的 BCD_adder.v 源程序如下：

```
module BCD_adder(a,b,cin,sum,cout);
    input[3:0]      a,b;
    input           cin;
    output reg[3:0]  sum;
    output reg      cout;
always
    begin
        {cout,sum}=a+b+cin;
```

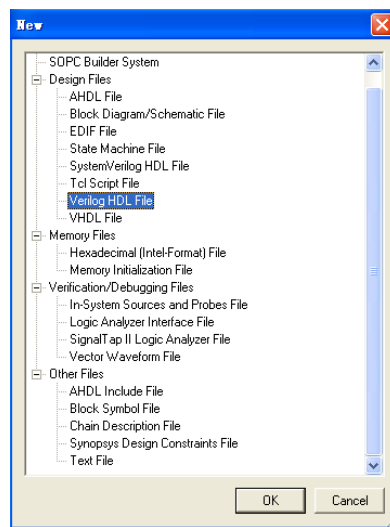


图 6.16 打开新文件对话框


```

    if({cout,sum}>'b01001)
        {cout,sum}=sum+'b01110;
    end
endmodule

```

完成 BCD 数加法器源程序的编辑后, 用 BCD_Adder.v 文件名存盘, 然后在 Quartus II 集成环境下, 对 BCD_adder 进行编译, 然后执行“File”菜单的“Create/Update”项的“Greate Symbol Files for Current File”命令, 为 BCD_adder 设计文件生成元件符号, 如图 6.16

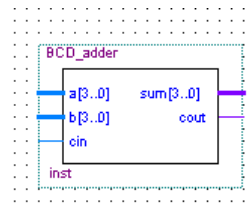


图 6.17 BCD_Adder 元件符号

所示。在元件符号中, 细的输入/输出线表示单变量线, 如 cin 和 cout; 粗的输入输出线表示多变量总线, 如 a[3..0]、b[3..0]和 sum[3..0]。

BCD 数加法器的仿真波形如图 6.18 所示, 仿真结果验证了设计的正确性。

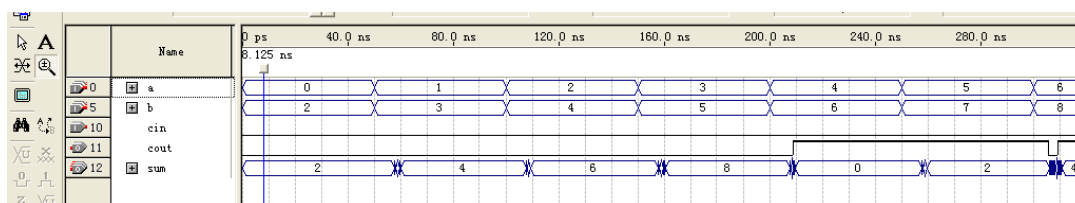


图 6.18 BCD 数加法器的仿真波形

2. 编辑七段显示译码器源程序

首先为七段显示译码器设计建立一个新工程, 然后在 Verilog HDL 文本编辑方式下, 编辑七段显示译码器的源程序, 并以 BCD_Dec7.v 为程序名, 保存在工程目录中。BCD_Dec7.v 源程序如下:

```

module BCD_Dec7(a,q);
    input [3:0]    a;
    output [7:0]  q;
    reg  [7:0]  q;
    always @(a)
    begin
        case(a)
            'b0000: q = 'b00111111;    'b0001: q = 'b00000110;
            'b0010: q = 'b01011011;    'b0011: q = 'b01001111;
            'b0100: q = 'b01100110;    'b0101: q = 'b01101101;

```

```

'b0110: q = 'b01111101;    'b0111: q = 'b00000111;
'b1000: q = 'b01111111;    'b1001: q = 'b01101111;
'b1010: q = 'b01110111;    'b1011: q = 'b01111100;
'b1100: q = 'b00111001;    'b1101: q = 'b01011110;
'b1110: q = 'b01111001;    'b1111: q = 'b01110001;

endcase

end

endmodule

```

为了使 BCD_Dec.v 源程序也能作为十六进制译码器，所以将 A~F 十六进制数的译码输出也包括在内。BCD_Dec.v 通过编译后，生成显示译码器的元件符号如图 6.19 所示。在元件符号中，a[3..0]是译码器的输入端，将与 BCD 加法器的输出端 sum[3..0]连接；q[7..0]是译码器的输出端，为七段数码显示器提供显示数据。

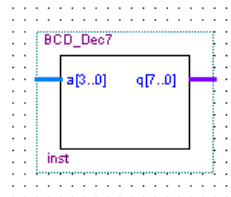


图 6.19 显示译码器的元件符号

6.5.2 设计 BCD 数加法器电路顶层文件

生成的 BCD_adder 和 BCD_Dec7 设计电路的元件图形符号只是代表两个分立的电路设计结果，并没有形成系统。顶层设计文件就是调用 BCD_adder 和 BCD_Dec 两个功能元件，将它们组装起来，成为一个完整的设计。bcd_dec.bdf 是本例的顶层设计文件，在 Quartus II 集成环境下，首先为顶层设计文件建立一个新工程（bcd_dec），然后打开一个新文件，并进入图形编辑方式（Block Diagram/Schematic File）。在图形编辑框中，调出 BCD_adder 和 BCD_Dec7 元件符号以及输入（input）和输出（output）元件符号，如图 6.20 所示。

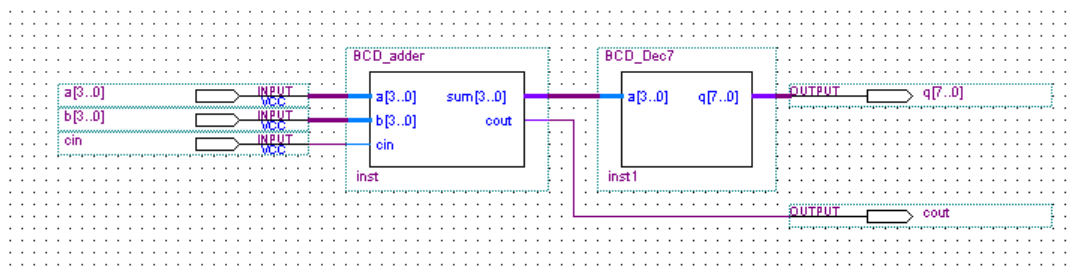


图 6.20 BCD 加法器的顶层设计图

根据 BCD 数加法器电路设计原理，用鼠标按连接关系将它们拉接在一起。具体操作如下：

① 把输入元件 input 与 BCD_adder 的 cin 拉接在一起，并把输入元件的名称改为 cin，作为加法器低位进位输入端。

② 把输入元件 input 与 BCD_adder 的加数输入拉接在一起，并把输入元件的名称分别改为 a[3..0]和 b[3..0]，作为加法器 4 位加数输入端。

③ 把 BCD_adder 输出 sum[3..0]与 BCD_Dec7 的输入 a[3..0]拉接在一起，把 BCD_Dec7 的输出 q[7..0]与输出元件拉接在一起，并把输出元件的名称改为 q[7..0]，作为译码输出端。

④ 把输出元件 output 与 BCD_Dec7 的 cout 拉接在一起，并把输出元件的名称改为 cout，作为加法器进位输出端。

完成上述操作后，得到 BCD 加法器的顶层设计结果（见图 4-19 所示）。顶层设计图形完成后，用 bcd_dec.bdf 作为文件名存入工程目录中。“bcd_dec”是用户为顶层文件定义的名字，后缀.bdf 表示图形设计文件。

6.5.3 编译顶层设计文件

“Processing”菜单下的“Start Compilation”命令，对顶层设计文件进行编译。完成对图形编辑文件的编译后，系统为 BCD 加法器设计文件生成的元件符号如图 6.21 所示。此元件符号可以作为共享元件，供其他电路或系统设计调用。

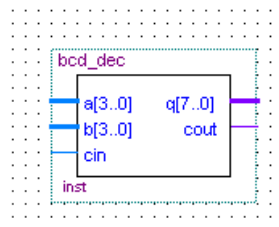


图 6.21 BCD 加法器的元件符

6.5.4 仿真顶层设计文件

BCD 加法器设计电路的仿真波形如图 6.22 所示，其中输出 q[7..0]输出的“7D”是送七段数码管显示“6”（0 加 6）的数据；“07”是显示“7”（1 加 6）的数据；依此类推。仿真结果验证了设计的正确性。

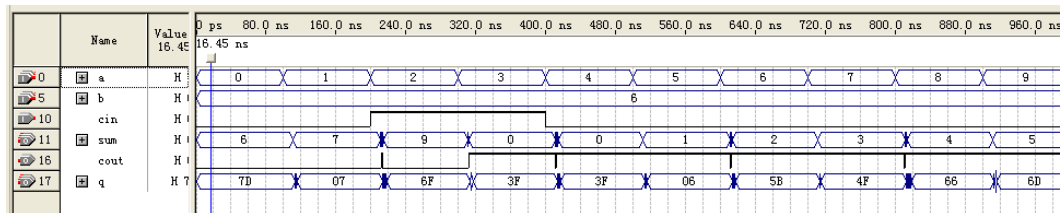


图 6.22 TOP 顶层设计文件的仿真波形

6.6 Verilog HDL 仿真

Verilog HDL 是一种用于设计数字系统电路的硬件描述语言，为了检验设计的正确性，一般需要对设计模块进行仿真验证。几乎所有的 EDA 工具软件都支

持 Verilog HDL 的仿真，而且 Verilog HDL 本身也具有支持仿真的语句。本章介绍 Verilog HDL 仿真支持语句，Verilog HDL 测试平台软件的设计，并给出 ModelSim 软件工具的仿真结果。

6.6.1 Verilog HDL 仿真支持语句

Verilog HDL 仿真支持语句包括 initial 块语句、系统任务、系统函数和编译指令。关于 initial 块语句前面已介绍，下面介绍系统任务、系统函数和编译指令。

1. 系统任务和系统函数

系统任务和系统函数是 Verilog HDL 中预先定义的任务和函数，主要用途是在设计仿真过程中完成信息显示、仿真监视、模拟控制等工作任务。系统任务和系统函数都是以\$为首字符的标识符构成，如\$display、\$write、\$time（注意：\$与其后的保留字单词之间不能有空格）等。下面介绍用途比较广泛的信息显示系统任务（\$display 和\$write）、仿真监视系统任务（\$monitor）、暂停仿真系统任务（\$stop）、结束仿真系统任务（\$finish）和模拟时间函数（\$time 和\$realtime）。

（1）信息显示系统任务（\$display 和\$write）

信息显示系统任务用于仿真过程中在标准输出设备（如显示器、打印机等）显示字符串、表达式或变量的数值。信息显示系统任务包括\$display 或\$write（默认以十进制形式显示数据）、\$displayb 或\$writeb（以二进制形式显示数据）、\$displayh 或\$writeh（以十六进制形式显示数据）和\$displayo 或\$writeo（以八进制形式显示数据）。显示信息语句的格式为：

信息显示系统任务名(显示列表);

其中，显示列表中列出被显示的信息。例如

\$display("hello!"); //显示“hello!”字符串

\$display(\$time); //显示当前时间

\$display 和\$write 功能相同，主要区别在于\$display 在完成一条语句的信息显示后能自动换行，从新的一行开始显示下一条语句的信息；而用\$write 命令完成一条语句的信息显示后不会换行，继续在同一行显示下一条语句的信息。

模拟仿真时信息显示的数据是 64 位二进制数。例如，使用\$displayb 或\$writeb（二进制信息显示格式）显示数据“0”时，将在显示设备（显示器或打印机）上显示 64 个“0”；若使用\$displayh 或\$writeh（十六进制信息显示格式）显示数据“0”时，将在显示设备上显示 16 个“0”；而用若使用\$display 或\$write（十进制信息显示格式）显示数据“0”时，则将在显示设备上显示 1 个“0”；因此在一般情况下，使用\$display 或\$write 命令来显示数据信息比较便于观察。

在显示列表中显示对象的格式还可以用“%”符号来定义，主要包括：

- ① %d 或%D——以十进制格式显示数据。
- ② %b 或%B——以二进制格式显示数据。
- ③ %h 或%H——以十六进制格式显示数据。
- ④ od 或%O——以八进制格式显示数据。
- ⑤ %s 或%S——显示字符串。
- ⑥ %e 或%E——以科学计数法格式显示实数。

例如

```
$display("a=%d",a);           //以十进制格式显示变量 a 的数值
```

(2) 仿真监视系统任务 (\$monitor)

仿真监视系统任务是在仿真过程中，对字符串、表达式或变量的数值的显示进行监视，只要被监视的数据对象的数值发生变化时，立即显示变化后的结果。

\$monitor 任务的格式与\$display 完全相同，但\$display 任务仅对显示列表中的内容执行一次显示，而\$monitor 任务用于激活显示列表中的显示对象，在不同激励语句的触发下不断显示对象的信息，直至执行到\$stop 或\$finish 任务才停止监视。

【例 6.25】用\$display 任务编写 4 输入端与非门的测试程序。

解：编写的 4 输入端与非门测试程序（display_test.v）如下：

```
module display_test;
    wire y;
    reg  a,b,c,d;
    nand #1  g1(y,a,b,c,d);
    initial
    begin
        $display ("hello!",$time,,,"a=%b b=%b c=%b d=%b,y=%b",a,b,c,d,y);
        #10 a=0;b=0;c=0;d=0;
        #10 d=1;
        #10 a=1;b=1;c=1;d=1;
        #10  $finish;
    end
endmodule
```

程序中使用 initial 模块来完成仿真条件的设置，程序中的“#1”表示延迟 1 个基本单位时间，#10 表示延迟 10 个基本单位时间。程序执行时显示的信息为：

```
# hello! 0  a=x b=x c=x d=x,y=x
```

【例 6.26】用\$monitor 任务编写 4 输入端与非门的测试程序。

解：编写的 4 输入端与非门测试程序（monitor_test.v）如下：

```
module monitor_test;
    wire y;
    reg a,b,c,d;
    nand #1 g1(y,a,b,c,d);
    initial
    begin
        $monitor ("hello!",$time,,,"a=%b b=%b c=%b d=%b,y=%b",a,b,c,d,y);
        #10 a=0;b=0;c=0;d=0;
        #10 d=1;
        #10 a=1;b=1;c=1;d=1;
        #10 $finish;
    end
endmodule
```

程序执行时显示的信息为：

```
# hello! 0  a=x b=x c=x d=x,y=x
# hello! 10  a=0 b=0 c=0 d=0,y=x
# hello! 11  a=0 b=0 c=0 d=0,y=1
# hello! 20  a=0 b=0 c=0 d=1,y=1
# hello! 30  a=1 b=1 c=1 d=1,y=1
# hello! 31  a=1 b=1 c=1 d=1,y=0
```

display_test.v 和 monitor_test.v 两个源程序是完全相同的 4 输入端与非门仿真测试设计程序，a、b、c 和 d 是测试输入端，y 是测试输出端。在 display_test.v 源程序中，用\$display 系统任务显示信息，因此仅一次性地显示该命令的显示列表中的信息（其中的“x”表示未知数据）。用\$monitor 命令显示信息时，不仅显示了显示列表中的信息，而且不断监视显示列表中数据对象的变化，当任何数据对象发生变化后，立即显示变化后的信息，直至遇到\$finish 命令才结束显示。

monitor_test.v 程序执行的显示结果解释如下：

① 显示结果“# hello! 0 a=x b=x c=x d=x,y=x”表示 time=0（单位时间）时，数据对象 a、b、c、d 和 y 的数据未知。

② 显示结果“# hello! 10 a=0 b=0 c=0 d=0,y=x”表示 time=10 时，数据对象 a、b、c、d 分别赋值 0、0、0、0 后的变化数据，由于与非门需要 1 个单位时

间（#1）输出 y 才能出现新的数据，因此此时的 y 仍然是未知数据。

③ 显示结果 “# hello! 11 a=0 b=0 c=0 d=0,y=1” 表示 time=11 时，数据对象 a、b、c、d 和 y 的数据，因为经过 1 个单位时间后输出 y 出现了新的数据。

④ 显示结果 “# hello! 20 a=0 b=0 c=0 d=1,y=1” 表示 time=20 时，数据对象 a、b、c、d 分别赋值 0、0、0、1 后的变化数据（y 输出仍然为 1 不变）。

⑤ 显示结果 “# hello! 30 a=1 b=1 c=1 d=1,y=1” 表示 time=30 时，数据对象 a、b、c、d 和 y 的数据。

⑥ 显示结果 “# hello! 31 a=1 b=1 c=1 d=1,y=0” 表示 time=31 时，数据对象 a、b、c、d 和 y 的数据，因为经过 1 个单位时间后输出 y 出现了新的数据。从仿真结果可以验证与非门的功能，即只有全部输入（a、b、c、d）为 1（高电平 1）时，输出 y 才为 0（低电平），输入的其余组合都使输出 y 为 1。

（3）暂停仿真系统任务（\$stop）

\$stop 系统任务用于暂停仿真，进入仿真软件的命令交互模式。

（4）\$finish 任务

\$finish 系统任务用于结束仿真。

（5）模拟时间函数（\$time 和 \$realtime）

\$time 系统函数执行时根据系统任务（如 \$display、\$write、\$monitor 等）的格式要求，返回一个 64 位整型模拟时间，对小数部分自动进行四舍五入处理。而 \$realtime 系统函数执行时返回一个 32 位实型（十进制数格式）模拟时间，其输出格式与系统任务规定的格式无关。

6.6.2 Verilog HDL 测试平台软件的设计

测试平台（Test Bench）软件是用硬件描述语言编写的程序，在程序中用语句为一个设计电路或系统生成测试条件，如输入的高低电平、时钟信号等，在 EDA 工具的支持下，直接运行程序（不需要再设计输入条件），就可以得到仿真结果。下面介绍基于 Verilog HDL 的测试平台软件的设计。

测试平台软件的结构如图 6.23 所示，被测元件是一个已经设计好的电路或系统，测试平台软件用元件例化语句将其嵌入程序中。Verilog HDL 测试平台软件是一个没有输入输出端口的设计模块，被测元件的输入定义为 reg（寄存器）型变量，在 always 块或 initial 块中赋值（产生测试条件），被测元件的输出端口定义为 wire（网线）型变量，产生相应输入变化的输出结果（波形）。

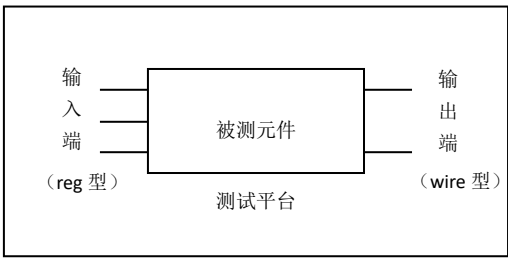


图 6.23 测试平台软件的结构

下面介绍组合逻辑电路、时序逻辑电路和系统电路的测试平台软件的设计，并以 ModelSim 为 EDA 工具，验证这些测试软件。

1. 组合逻辑电路测试平台软件的设计

组合逻辑电路的设计验证，主要是检查设计结果是否符合该电路真值表的功能，因此在组合逻辑电路测试平台软件编写时，用 initial 块把被测电路的输入按照真值表提供的数据变化，作为测试条件就能实现软件的设计。

【例 6.27】编写全加器电路的测试平台软件。

表 6.7 全加器真值表

A B CI	SO CO
0 0 0	0 0
0 0 1	1 0
0 1 0	1 0
0 1 1	0 1
1 0 0	1 0
1 0 1	0 1
1 1 0	0 1
1 1 1	1 1

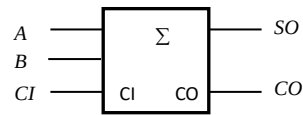


图 6.24 全加器电路的逻辑符号

解：全加器的逻辑符号如图 6.24 所示，真值表如表 4.7 所示。 A 、 B 是两个 1 位二进制加数的输入端， CI 是低位来的进位输入端， SO 是和数输出端， CO 是向高位的进位输出端。

用 Verilog HDL 编写的全加器源程序（adder1.v）如下：

```
module adder1(a,b,ci,so,co);
    input      a,b,ci;
    output     so,co;
    assign {co,so} = a+b+ci;
endmodule
```

根据全加器的真值表（见表 6.7 所示），编写的全加器测试程序（adder1_tb.v）

如下：

```
module adder1_tb;
    wire so,co;
    reg a,b,ci;
    adder1 u1(a,b,ci,so,co); //嵌入 adder1 元件
    initial                //产生测试条件
    begin
        #20 a=0;b=0;ci=0;
        #20 a=0;b=0;ci=1;
        #20 a=0;b=1;ci=0;
        #20 a=0;b=1;ci=1;
        #20 a=1;b=0;ci=0;
        #20 a=1;b=0;ci=1;
        #20 a=1;b=1;ci=0;
        #20 a=1;b=1;ci=1;
        #200 $finish;
    end
endmodule
```

在源程序中，把全加器的输入 a、b 和 ci 定义为 reg 型变量；把输出 so 和 co 定义为 wire 型变量；用元件例化语句“adder1 u1(a,b,ci,so,co);”把全加器设计电路嵌入测试平台软件中；用 initial 块语句来改变输入的变化而生成测试条件，输入的变化语句完全根据全加器的真值表编写的。

在 ModelSim 为 EDA 工具平台，完成全加器源程序 adder1.v 和其测试软件 adder1_tb.v 的编写，并通过编译。测试平台软件的仿真过程与波形仿真相同，包括装载设计文件、设置仿真激励信号和执行仿真 3 个操作。全加器（Adder1_tb.v 文件）的仿真结果如图 6.25 所示

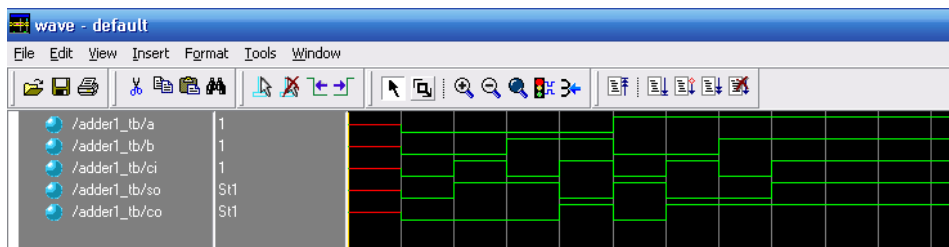


图 6.25 全加器的仿真波形

2. 时序逻辑电路测试平台软件的设计

时序逻辑电路测试平台软件设计的要求与组合逻辑基本相同，主要区别是在于时序逻辑电路测试平台软件中，需要用 `always` 块语句生成时钟信号。

【例 6.28】编写十进制加法计数器的测试软件。

解：用 Verilog HDL 编写的 4 位二进制加法计数器源程序 `cnt4e.v` 已在例 6.24 中给出，其测试软件（`cnt4e_tb.v`）如下：

```
module cnt4e_tb;
    reg      clk, clr, ena;
    wire [3:0] q;
    wire      cout;
    cnt4eu1(clk, clr, ena, cout, q);    //嵌入 cnt4e 元件
    always
        begin #50 clk = ~clk; end      //生成 clk 时钟
    initial begin
        clk=0; clr =0; ena=1;
        #1200 clr =1;
        #120 clr =0;
        #2000 ena=0;
        #200 ena=1;
        #20000 $finish;
    end
endmodule
```

在源程序中，用元件例化语句“`cnt4e u1(clk,clr,ena,cout,q);`”把十进制计数器设计元件嵌入测试软件中；在 `always` 中用语句“`#50 clk = ~clk;`”产生周期为 100（标准时间单位）的时钟（方波）；用 `initial` 块生成复位信号 `clr` 和使能控制信号 `ena` 的测试条件。

请读者注意，时钟 `clk` 只能用 `always` 块语句才能生成，但要在 `initial` 块中设置时钟的初始值（如 `clk=0` 或 `clk=1`），如果不设置时钟的初值，则在仿真是时钟输出端是一个未知 `x`（不变）结果。另外，用 `always` 块生成时钟后，一定要用“`$finish`”语句结束仿真，否则仿真执行将不会结束。

6.7 代码编写“规范”

“规范”这个富含专业气息的词汇（个人感觉），其实规范这种东西，就是大家都约定熟成的东西，一旦你不遵守这个东西，专业人士就会觉得你不够专业，特别是程序开发方面的问题。

为什么要规范呢？一方面能体现你足够专业，另一方面也是最重要的一方面，代码的规范性有利于开发交流，让代码的可读性大大增强，也有利于降低代码的出错率。

如果以前开发的时候都没有养成一个很好的代码规范，在大的项目工程中，这方面的弊端开始慢慢显现，对后续的修改和维护造成很大的影响，重新去编写又浪费很多的时间，所以请记住一句话“千里之行始于足下”，不要到走了很远之后才发现导向错了。

Verilog 代码编写的一些规范总结如下：

1. 文件名必须体现出设计模块的功能；

在 verilog 设计中，模块名就是文件名，这是由于软件编译的问题而限制的。所以设置模块名的时候也决定了文件名。因此，模块名的设定要体现出模块的功能，这样设定对于大的工程项目设计很有用处，便于查找分析。

2. 时钟信号及低电平有效信号的命名：

时钟信号一般的命名为：clk；

请回忆一下，你所编写的程序中低电平复位信号是如何编写的，目前我所见过的版本有：**reset, rst, reset_n, rst_n**；普遍就是这四种。前两种体现出该信号是高电平有效的，而后两种体现该复位信号是低电平有效的，在设计规范中就是采用*_n 来表示低电平有效的。所以低电平有效的信号一律采用*_n 表示，增强代码的可读性。

3. 参数 parameter 和宏定义`define

在定义参数和宏定义时，名字要采用大写进行表示，如下：

```
parameter DATA_WIDTH = 32;
`define AHB_TRANS_SEQ 2'b11;
```

4. 信号、端口、模块、例化

有关这四个方面的代码设计最好采用小写进行设计。

5. 命名字符的长度

一般字符命名的长度不要超过 32 个；

6. 有关模块调用的实体名的设定，参考如下：

```
mux4 u_mux4_1 (...);
mux4 u_mux4_2 (...);
```

7. 位宽的描述

在 Verilog 的设计中一般采用的为[x:0]; 在 VHDL 中一般采用的是 x downto 0; 采用的都是从高在左低位在右符合人的阅读顺序的方式。

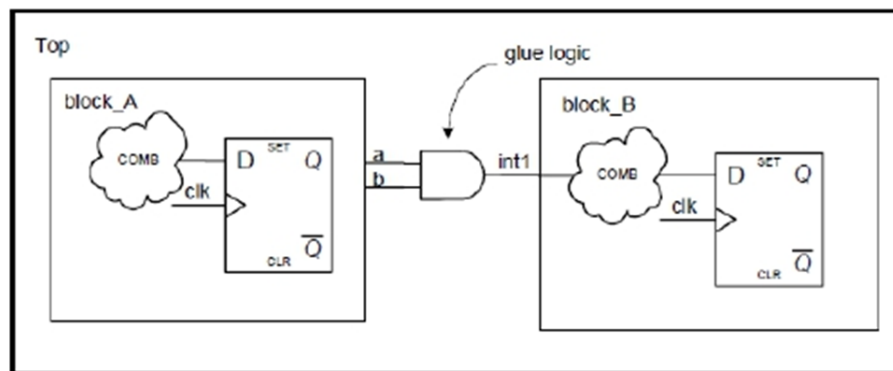
8. Begin 和 end、case 和 endcase、if 和 else 的配对对应上下对准, 语句应注意缩进, 方便阅读代码。可添加必要的注释!

9. 到这里所说到的都是关于文本输入形式的, 那么在原理图输入方式中有没有相关的规范问题呢? 答案是肯定的!!!所以经常采用原理图输入的好好注意了!!!

原理图输入方式中, 有时会将自己所写的文本生成一个原理图的模块, 然后导入到原理图的设计中, 这时该原理图也被设置为顶层模块, 所以有时问题就产生在这里。当有多个模块导入原理图时, 这时就需要将这些模块连接起来, 如果在连接的时候用一些分立的逻辑元件将模块与模块之间连接起来, 这就产生了一个所谓“胶合逻辑”的问题, 因为该逻辑元件在综合时是没有加入到任何一个模块里面的, 所以就无法进行整体的最优化综合, 所以就无法产生一个最优化的电路, 这个逻辑元件将两个模块无缝地连接起来, 这样的电路在同步中又怎么会稳定呢?

所以请记住一点, 有文本输入和原理图输入的工程中, 原理图作为顶层文件时, 原理图最好是页水平的分层结构, 即原理图中的模块都是采用文本输入之后生成的, 这样就避免了“胶合逻辑”, 让综合器更好地综合我们设计的电路。

“胶合逻辑”实例: 是否在你的原理图中出现过呢?



10. 非阻塞和阻塞问题, 这个问题在 FPGA 设计中是老生常谈的问题, 注意两种赋值模式产生的硬件电路不一样的, 从时间上进行分析的话, 非阻塞赋值就是同时赋值, 而阻塞赋值就是顺序赋值, 软件综合时也会帮你简化。

在一个源程序中, 要么都采用阻塞赋值语句, 要么都采用非阻塞赋值语句, 最好不要混合使用, 否则可能逻辑关系出错!

为易于综合, 建议均采用非阻塞赋值语句!

11. 信号敏感列表

什么是信号敏感列表呢?就是在需要信号输入的模块中, 如 always @ () --括

号内包含的信号就是敏感信号，也就是说在括号内的信号都将会影响模块的工作或者是模块内某些信号的输出。

那什么信号是敏感信号呢？凡是影响模块输出和工作状态的信号都将是敏感信号，一般有：时钟信号、复位信号、置数信号、某些赋值语句的右边信号还有一些条件信号。

12. 复位&初始化

相信你看过的经典代码中，一般都会包含有异步复位信号 `rst_n` 吧！

为什么要复位呢？

一个是为了初始化，另一个是在系统工作异常时重新进入初始状态（就像电脑有时死机重启一样）。

为什么要初始化呢？

在 FPGA 中经常要用到 `reg`，不是吗？而且是经常要用到，没有 `reg` 就没有 FPGA。而 `reg` 在上电后里面所寄存的是 1 还是 0 是不一定的，这将会造成很大的问题，会造成输出不确定、会造成状态机进入与确定的状态……所以就需要在上电的时候，利用复位信号进行初始化，就 `reg` 的值进行初始化，使一切都是确定的，防止不确定状态的出现。实例如下：

```
always @(posedge clk or negedge rst_n)
begin
    if(!rst_n)
        data_out<=1'b0;
    else if(Y==3'b001)
        data_out<=1'b1;
    else
        data_out<=1'b0;
end
```

13. 一些设计技巧

①一个变量不能在多个 `always` 块中被赋值！这个问题一定要注意！否则编译不能通过。当某个变量有多个触发条件时，最好将它们放在一个 `always` 块中，并用 `if-else` 语句描述在不同触发条件下应执行的操作！

②在 `always` 块语句中，当敏感信号为两个以上的时钟边沿触发信号时，应注意不要使用多个 `if` 语句！以免因逻辑关系描述不清晰而导致编译错误。

```
always@(posedge start or posedge reset)
begin
```

```

    if (reset) enable <=0;
    if (start) enable<=1;  //错误写法
end

```

变量 enable 不能被分配新的值!在最初开始, start 和 reset 均为 0, 导致 enable 为不定态。

```

always@(posedge start or posedge reset)
    if (reset) enable <=0;
    else enable<=1;  //正确写法

```

语句 “else enable<=1;”隐含了 reset 无效、且 start 有效的意思, 因此与 else if(start) enable<=1;效果一样!

③当输出信号为总线信号时, 一定要在 I/O 说明中指明其位宽! 否则在生成逻辑符号时, 输出信号被误认为是单个信号, 而没有标明位宽, 就不会当成总线信号。

④当要用到计数器时, 一定要根据计数最大值事先计算好所需的位宽! 若位宽不够, 则计数器不能计到你设定的最大值, 当该计数器用作分频时, 则输出时钟始终为 0, 所设计电路将不能按预定功能正常工作!

6.8 本章小结

Verilog HDL 是 EDA 技术的重要组成部分。本章介绍 Verilog HDL 的语法结构, 包括变量、语句、模块和不同级别的电路设计和描述。

Verilog HDL 具有行为描述和结构描述功能, 可以对系统级 (System Level)、算法级 (Algorithm Level) 和寄存器传输级 (RTL: Register Transfer Level) 等高层次抽象级别进行电路设计和描述, 也可以对门级 (Gate Level) 和开关级 (Swicth Level) 等低层次的抽象级别进行电路设计和描述。由于 Verilog HDL 在数字电路设计领域的先进性和优越性, 使之成为 IEEE 标准的硬件描述语言, 得到多种 EDA 设计平台工具软件的支持。

第七章 数字基础实验

7.1 分频器的设计及 quartus II 仿真

一、分频器简介

所谓“分频”，就是把输入信号的频率变成成倍数的低于输入频率的输出信号。文献资料上所谓用计数器的方法做“分频器”的方法，只是众多方法中的一种。它的原理是：把输入的信号作为计数脉冲，由于计数器的输出端口是按一定规律输出脉冲的，所以对不同的端口输出的信号脉冲，就可以看作是对输入信号的“分频”。至于分频频率是怎样的，由选用的计数器所决定。如果是十进制的计数器那就是十分频，如果是二进制的计数器那就是二分频，还有四进制、八进制、十六进制等等。以此类推。

在数字系统的设计中，分频器是一种应用十分广泛的电路，其功能就是对频率较高的信号进行分频。本质上，分频电路是加法计数器的“变异”，其计数值有分频叙述决定，分频器输出不是一般计数器的计数结果，而是根据分频常数对输出信号的高、低电平进行控制。通常来说，分频器常用于对数字电路中的时钟信号进行分频，从而得到较低频率的时钟信号、选通信号、中断信号等。

二、分频系数为 10 的偶数分频器

偶分频意思是时钟模块设计最为简单。首先得到分频系数 M 和计数器值 N 。

$M = \text{时钟输入频率} / \text{时钟输出频率}$;

$N = M / 2$;

如输入时钟为 $50M$ ，输出时钟为 $25M$ ，则 $M=2$ ， $N=1$ 。偶分频则意味着 M 为偶数。

以 $M=4$ ， $N=2$ 为例，输入时钟为 $50M$ ，输出时钟为 $12.5M$ ，我们希望得到的输出时钟时序如下：

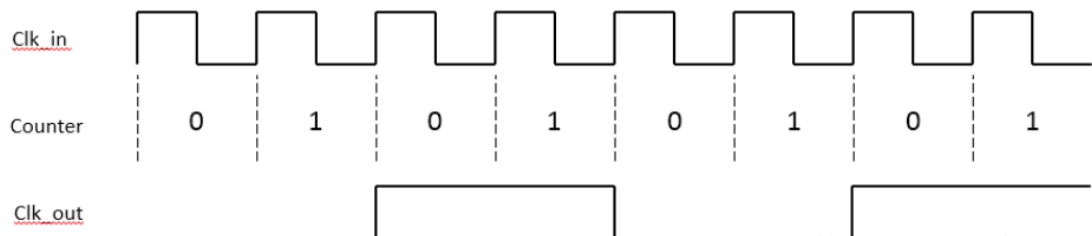


图 8-1 分频原理示意图

对于分频系数为 $M=10$ 的分频器，本例的输入时钟系统 50M 时钟(`clk_50M`)，输出为十分频时钟 (`f_10`)。设计方法为，通过设置一个 3 位的计数寄存器 (`cnt`) 来实现，每种系统时钟周期的上升沿计数一次，当计数寄存器数到 4 的时候，将输出分频信号取反即可得到 10 分频的输出。

本例程的 verilog 程序为：

```
module fengping_2(clk_50M, f_10);
input  clk_50M;           //系统输入时钟，50M，周期20ns
output f_10;              //10分频输出，5M

reg f_10;                 //输出寄存器
reg[2:0] cnt;             //计数寄存器

always@(posedge clk_50M) //每个时钟周期的上升沿触发，执行begin_end中的语句
begin
if(cnt==3'b100)          //判断cnt是否为4, 是的话执行以下程序
begin
    f_10<=~f_10; //把f_10取反
    cnt<=3'b0;   //计数寄存器清零
end
else                      //cnt没到4，执行以下程序
begin
    cnt<=cnt+3'b1; //计数寄存器自加一
end
end

endmodule
```

三、分频器波形仿真

关于 QuartuesII 的波形仿真操作过程，在本例程笔者作简要以下介绍。

1. 首先建立波形仿真的文件：

在 File 中选中 New 选项，选中并新建 **Vector Waveform File** 文件，如下图所示。

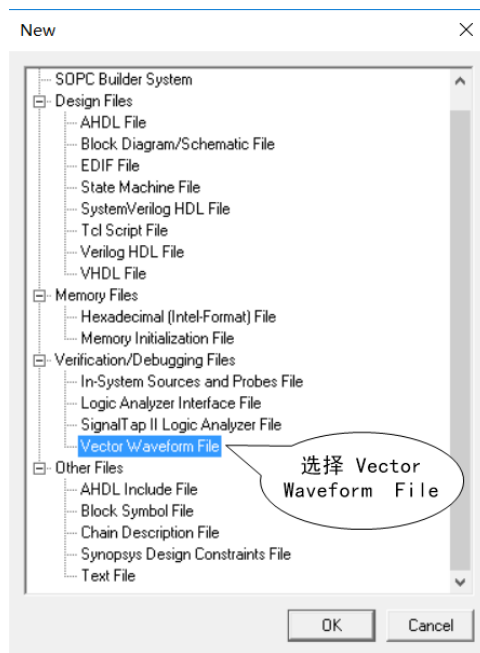


图 8-2 选择新建 Vector Waveform File 文件

2. 对建立的仿真文件进行保存：

按软件左上脚的保存按钮，在弹出的窗口中按保存，如下图所示。



图 8-3 保存文件

3. 在新建的仿真文件中导入需要仿真的信号，在 new 栏目的空白处双击鼠标左键，按下图所示：

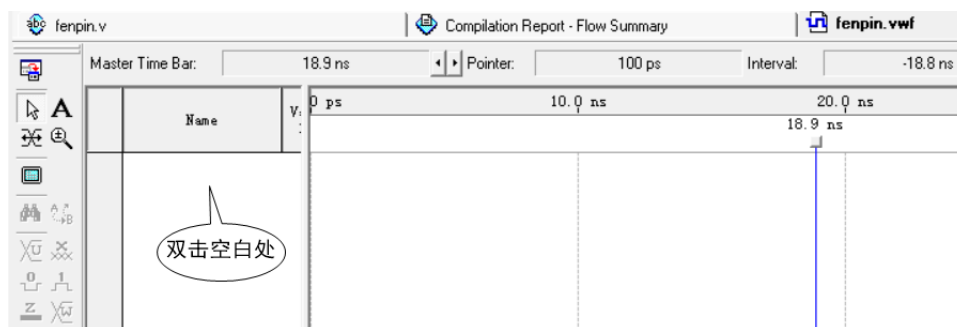


图 8-4 导入信号

4. 在弹出的对话框中，选中 Node Finder 选项，如下图所示：

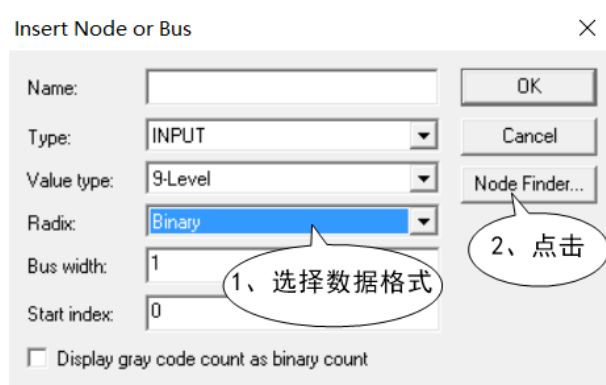


图 8-5 选中 Node Finder 选项

5. 在 Filter 栏目中选中 Pin:all 选项，然后点击 List 按钮，如下图所示：



图 8-5 Node Finder 设置

6. 将左边的端口导入到右边的窗口，按如图所示按键，然后点击 OK 按钮，如下图所示：

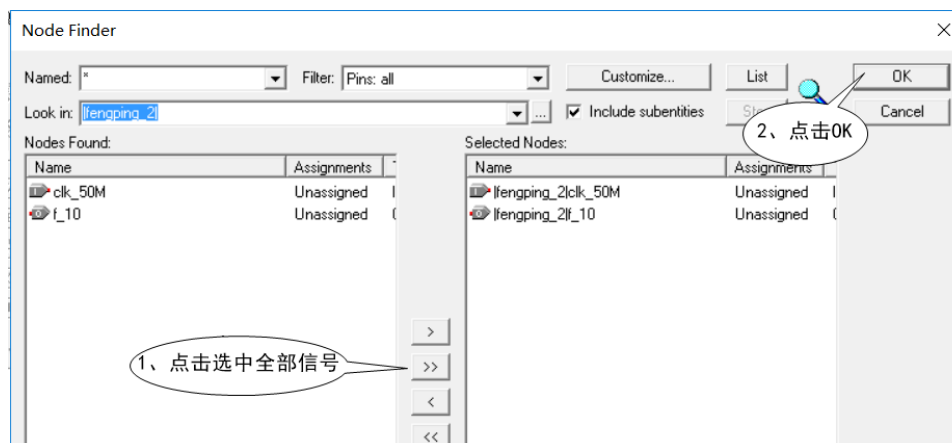


图 8-7 选中信号

7. 在弹出的对话框中选中 OK 的按钮，如下图所示：

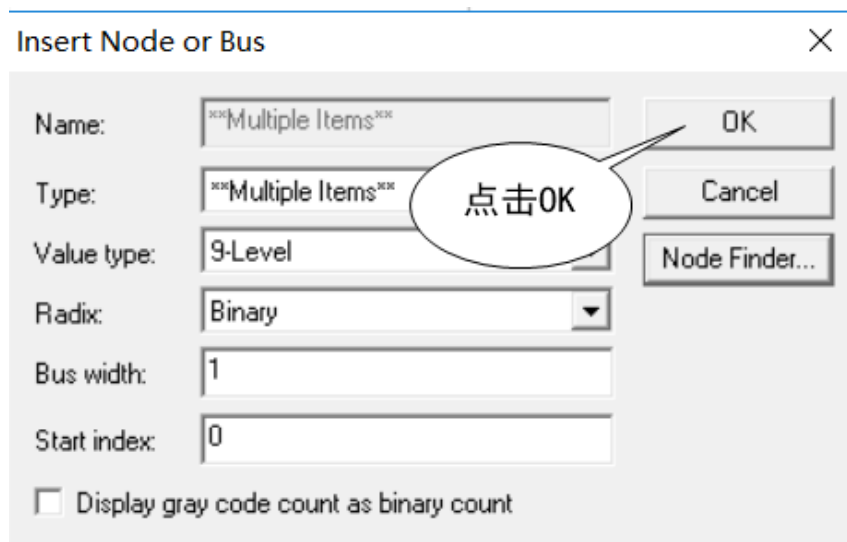


图 8-8 完成信号设置选中 OK 按钮

8. 用鼠标左键选中 clk_50M 的那一栏，然后点击左下角的时钟工具，设置系统的仿真时钟，如下图所示：

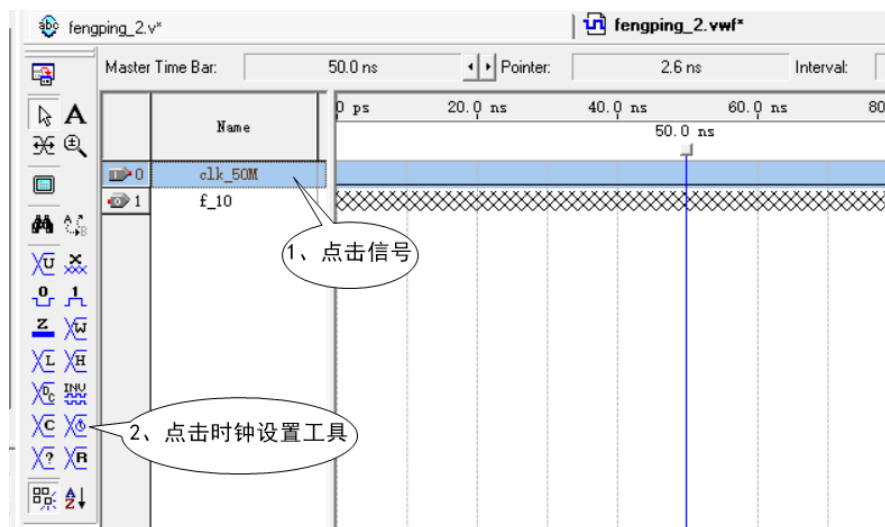


图 8-9 时钟信号设置工具选择

9. 在弹出的对话框中，可以设置系统时钟的占空比以及时钟周期，此处我们的占空比设置为 50%，时钟周期为 10ns，然后点击 ok 即可，如下图所示：

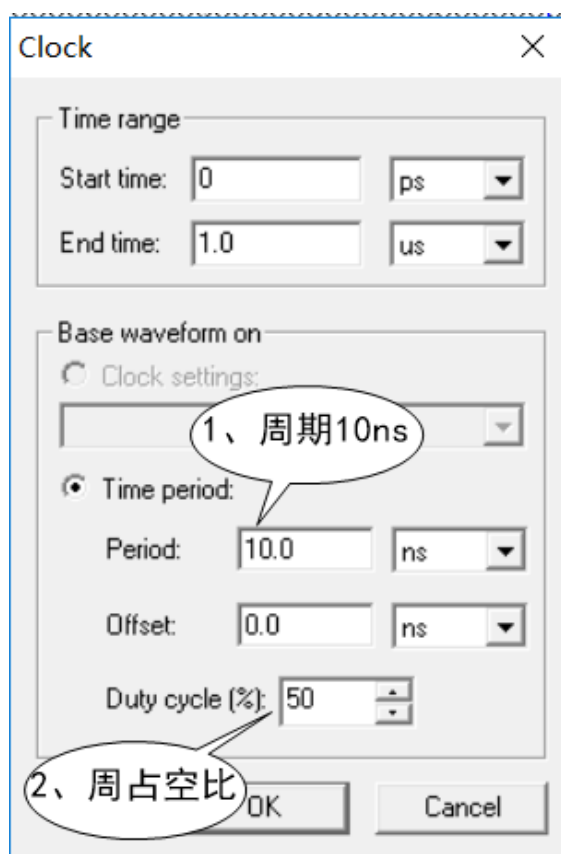


图 8-10 时钟参数设置

10. 首先，进行功能仿真，什么是功能仿真呢，功能仿真简单来说既是理想的运行情况，不会存在任何的逻辑器件的延时（实际运行时是不存在的）。首先在软件上方工具栏点击红色框所示的设置按钮（Settings），对所建立的工程参数进行设

置:



图 8-11 工程参数设置按钮

11. 在弹出的对话框中选中 Simulator Settings 选项, 然后在箭头所示地方选中 Funtional 选项, 既功能仿真, 最后按 ok 选项, 如下图所示:

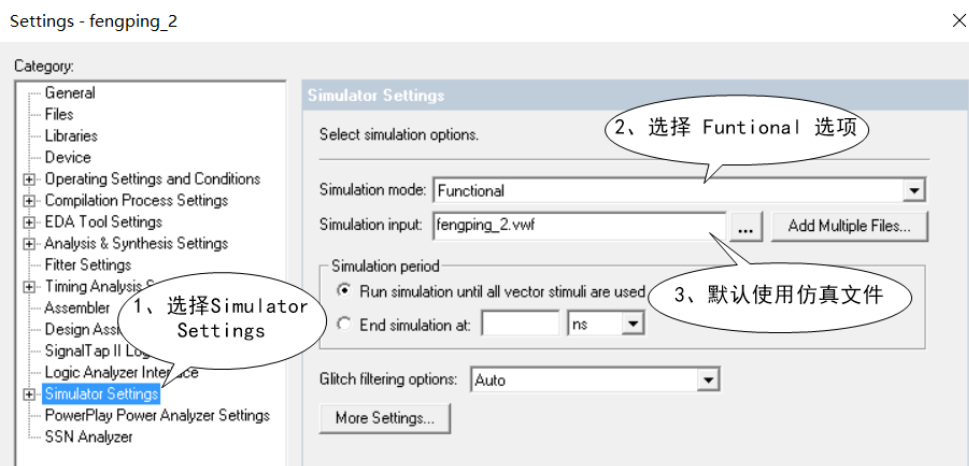


图 8-12 功能仿真设置

12. 在 Processing 栏目中选中 Generate Functional Simulation Netlist 选项, 从而产生仿真网表文件, 如下图所示:

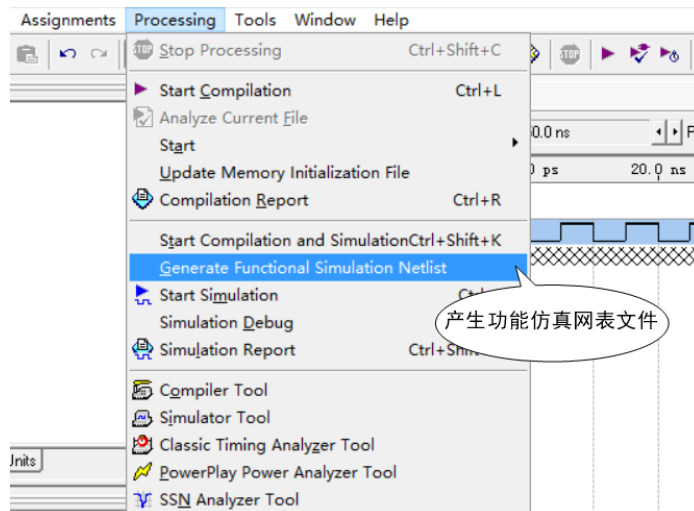


图 8-13 产生功能仿真网表文件

13. 点击软件上方的仿真按钮, 如下图所示:

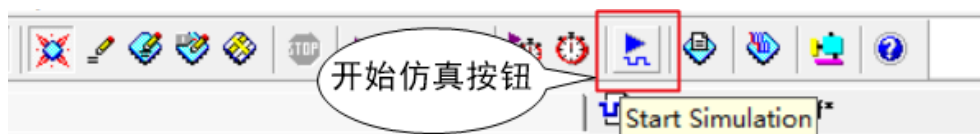


图 8-14 仿真按钮

14. 等待软件运行后，功能仿真波形即可出来：10 个 clk_50M 周期产生一个 f_10 周期，证明设计的分频系数为 $M=10$ 的分频器正确。

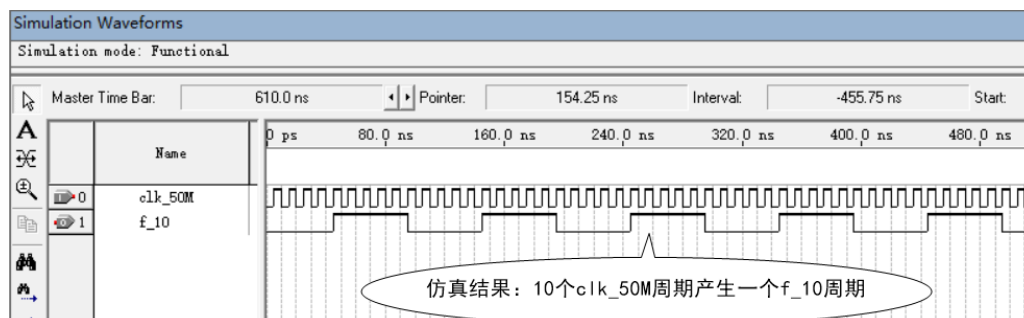


图 8-15 功能仿真波形

15. 下面进行另外一种仿真，时序仿真，时序仿真先对与功能仿真而言其仿真结果比较接近实际情况，即是存在逻辑器件的时间延迟。

首先在软件上方点击红色箭头所示的设置项，对所建立的工程进行设置：



图 8-16 工程参数设置按钮

16. 在弹出的对话框中选 Simulador Settings 选项，然后在箭头所示地方选中 Timing 选项，既功能仿真，最后按 ok 选项，如下图所示：

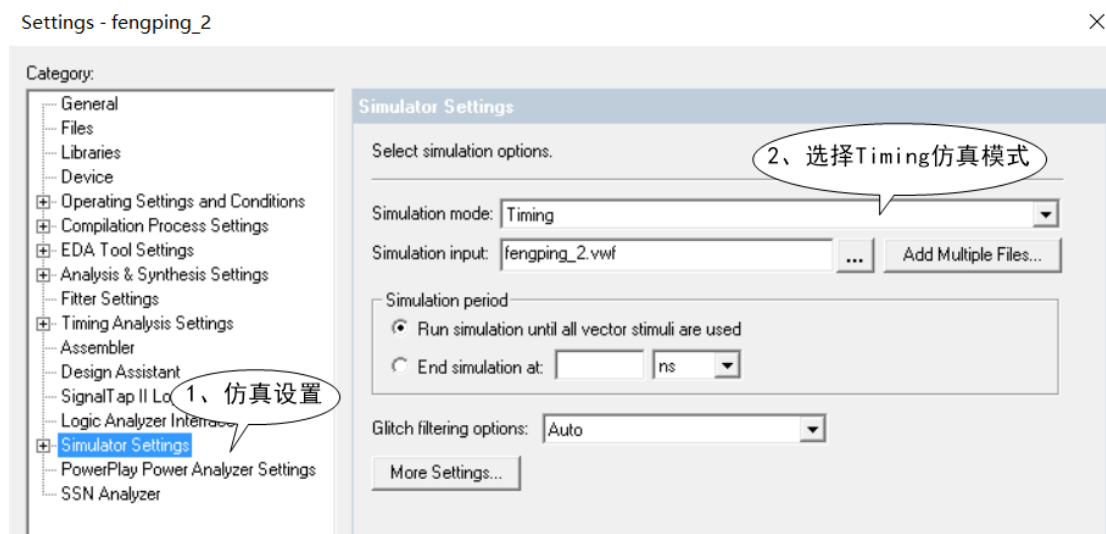


图 8-17 Timing 时序仿真模式选择

17. 对工程进行全编译，按全编译按钮，如下图所示：

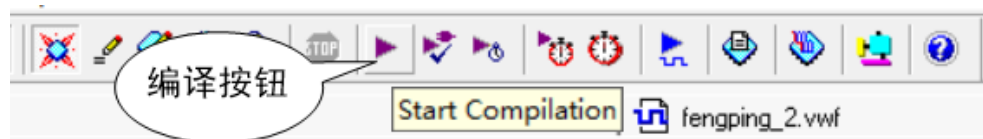


图 8-18 编译按钮

18. 点击软件上方的仿真按钮，如下图所示：

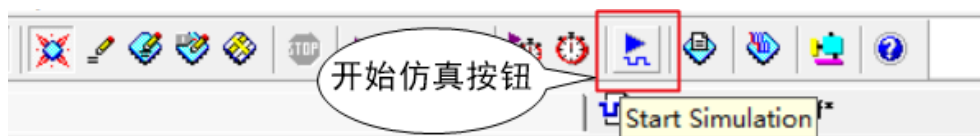


图 8-19 仿真按钮

19. 等待软件运行后，波形仿真波形即可出来，大家仔细看不难发现，时序仿真在每个信号电平变化处都出现了相应的时序延时，因为时序仿真是模拟 FPGA 实际运行时的状态，信号传输有十几纳秒的时间延时，如下图所示。

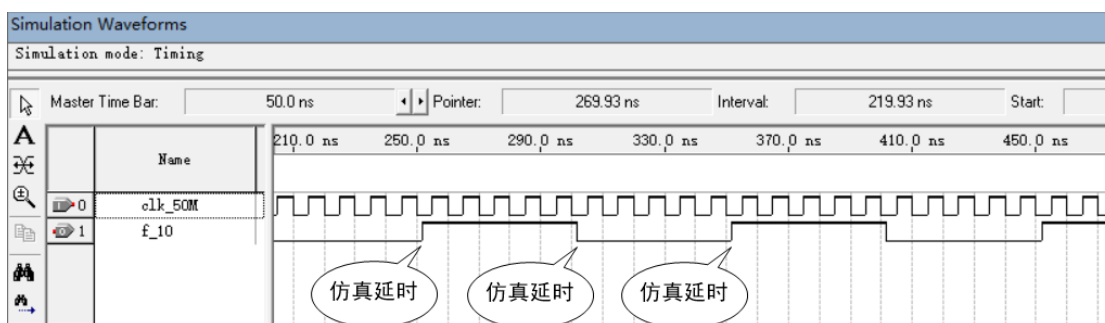


图 8-20 时序仿真结果

7.2 计数器的设计

一、计数器简介

计数器的逻辑功能是用于记忆时钟脉冲的具体个数,通常计数器的最多能够记忆时钟的最大数目为 m 成为计数器的模,即计数器的范围为 $0 \sim m-1$ 或则 $m-1 \sim 0$ 。其基本原理就是将几个触发器按照一定的顺序链接起来,然后根据触发器的组合状态,按照一定的计数归路随着时钟脉冲的变化来记忆时钟脉冲的个数。

二、同步 4 位 2 进制计数器的设计

同步 4 位 2 进制计数器是数字电路中数值电路中广泛使用的计数器。本例程介绍一种具有异步清零,同步置数功能的 4 位 2 进制计数器的设计方法。其中输入时钟为: clk 、置数端: s 、清零端: r 、使能端: en 、置数端: $d[3:0]$ 、计数输出端: $q[3:0]$ 、进位端: $c0$ 。其中同步 4 位 2 进制计数器的状态表如下:

表 7.1 同步 4 位 2 进制计数器的状态表

clk	r	s	en	工作状态
X	1	X	X	置零
↑	0	1	X	预置数
↑	0	0	1	计数
X	0	0	0	保持不变

本例程的 verilog 程序为:

```
module counter(clk,co,q,r,s,en,d);
input  clk,r,s,en;           //时钟,清零端,置数端,使能端
input[3:0] d;                //置数输入端
output co;                   //进位端
output[3:0] q;               //计数输出端

reg[3:0] q;                  //4位的计数寄存器
reg co;                      //1位进位寄存器

always@(posedge clk)        //时钟上升沿触发
if(r)                        //判断清零端是否为1
    begin q=0;end           //是的话把计数寄存器清0
else
begin
    if(s)                    //判断置数端是否为1
    begin q=d;end//是的话 把置数输入端的值赋予计数寄存器
    else
    if(en)                    //判断使能端是否为1
    begin
        q=q+4'b1;           //是的话q自加1
        if(q==4'b1111)      //判断q是否计满
        begin co=1;end      //是的话进位端置1
        else
        begin co=0;end      //否的话进位端置0
    end
    else
    begin    q=q;end         // q保持原值
end
endmodule
```

三、计数器的波形仿真:

a) 功能仿真结果:

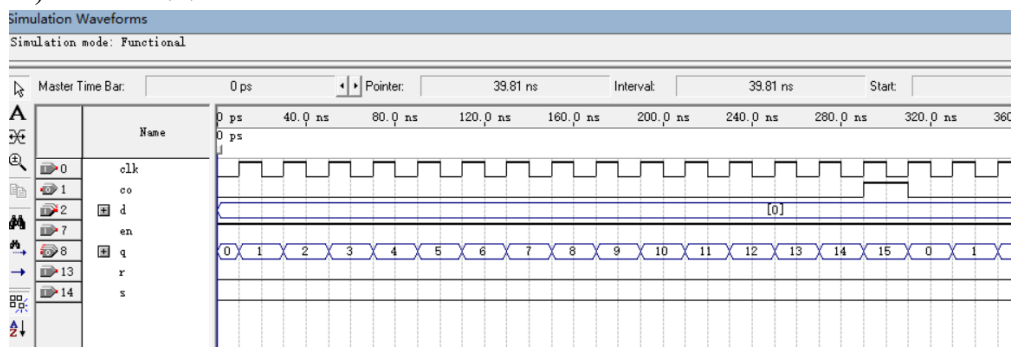


图 8-21 计数器功能仿真图

b) 时序仿真结果:

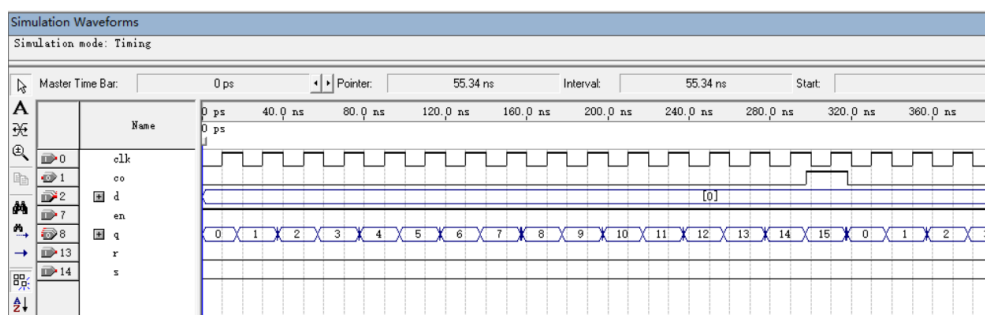


图 8-21 计数器时序仿真图

7.3 D 触发器

一、D 触发器简介:

D 触发器是最常用的触发器之一。对于上升沿触发 D 触发器来说, 其输出 Q 只在 CLK 由 L 到 H 的转换时刻才会跟随输入 D 的状态而变化, 其他时候 Q 则维持不变。

二、D 触发器的设计:

D 触发器的真值表如下, 其中输入信号有: 信号输入端 d、时钟信号 cp、清零端 r、置数端 s; 输出信号有: q、qn。

表 7.2 D 触发器的真值表

输入				输出	
CP	R	S	D	Q	$\sim Q$
X	0	1	X	0	1
X	1	0	X	1	0
0	1	1	X	保持	保持
$\uparrow$	1	1	0	0	1
$\uparrow$	1	1	1	1	0

本例程的 Verilog 程序为:

```

module D(q, qn, d, cp, r, s);
output q, qn;      //D触发器的两个输出
input d, cp, r, s;  //D触发器的四个输入
reg q, qn;          //输出寄存器
always@(posedge cp) //在cp的上升沿触发
begin
    if({r, s}==2'b01) //判断是否有r=0, s=1
    begin
        q=1'b0;
        qn=1'b1;
    end
end

```

```

end
else if({r,s}==2'b10)    //判断是否有r=1,s=0
begin
    q=1'b1;
    qn=1'b0;
end
else if({r,s}==2'b11)    //判断是否有r=1,s=1
begin
    q=d;
    qn=~d;
end
end
endmodule

```

三、D 触发器的波形仿真：

a) 功能仿真结果：

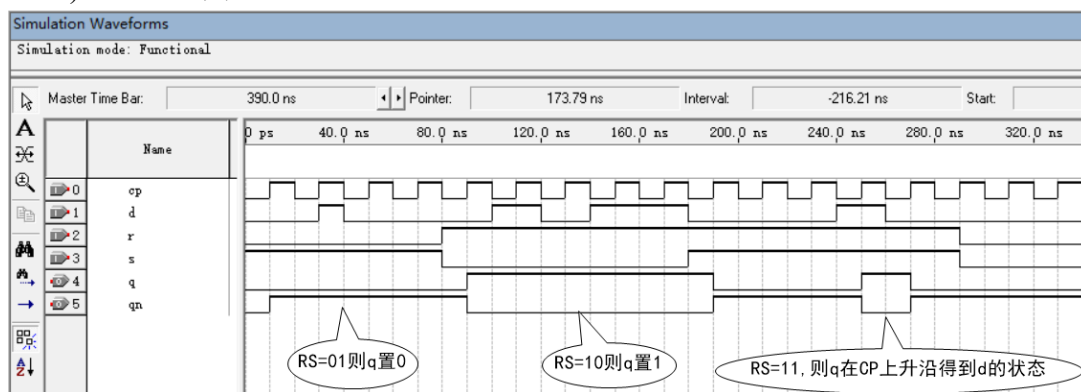


图 8-22 D 触发器功能仿真图

b) 时序仿真结果：

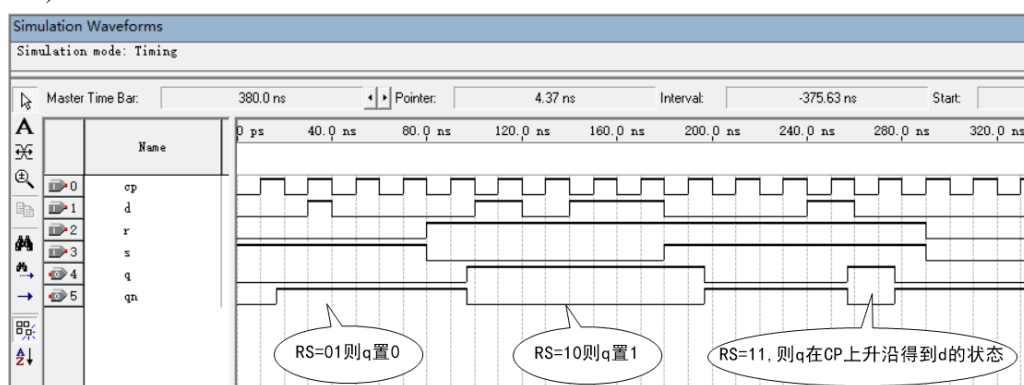


图 8-23 D 触发器时序仿真图

7.4 三态门

一、三态门简介:

三态门是指逻辑门的输出除了高低电平两种状态外,还有第三种状态——高阻状态的门电路。高阻状态相当于隔离状态。三态门都有一个 EN 使能控制端,来控制门电路的通断。

三态门有三种输出状态:输出高电平、输出低电平和高阻状态,前两种状态为工作状态,后一种状态为禁止状态。值得注意的是,三态门不是具有三种逻辑值。在工作状态下,三态门的输出可为逻辑‘0’或者逻辑‘1’;在禁止状态下,其输出呈现高阻态,相当于开路。

三态门有广泛的应用,利用三态门可以实现线与,也被广泛应用于总线传送。总线传送时,为了保证数据传送的准确性,任意时刻,n 个三态门的控制端只能有一个为 1,其余均为 0,而三态门利用高阻态可以很好的实现这一特性。

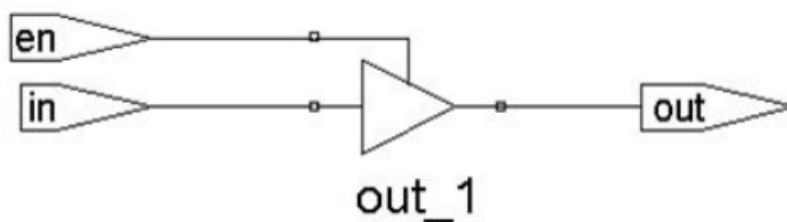


图 8-24 三态门符号示意图

二、三态门的设计:

采用条件操作语句进行设计, $E=0$ 时,输出端处于开路状态,也称高阻状态,这种状态下,无论输入端是高电平还是低电平对输出都无影响; $E=1$ 时,为工作状态,输入和输出为与非关系。

表 7.3 三态门的真值表

输入	输入	输出
EN	DIN	DOUT
0	0	Z
0	1	Z
1	0	0
1	1	1

本例程的 Verilog 程序为：

```
module tri_gata(dout,din,en);
input din,en;           //三态输入端，使能输入
output dout;           //三态输出端
assign dout=en?din:'bz; //判断en是否为1，是就把din赋予dout，否则把dout置高阻
endmodule
```

三、三态门的波形仿真：

a) 功能仿真结果：

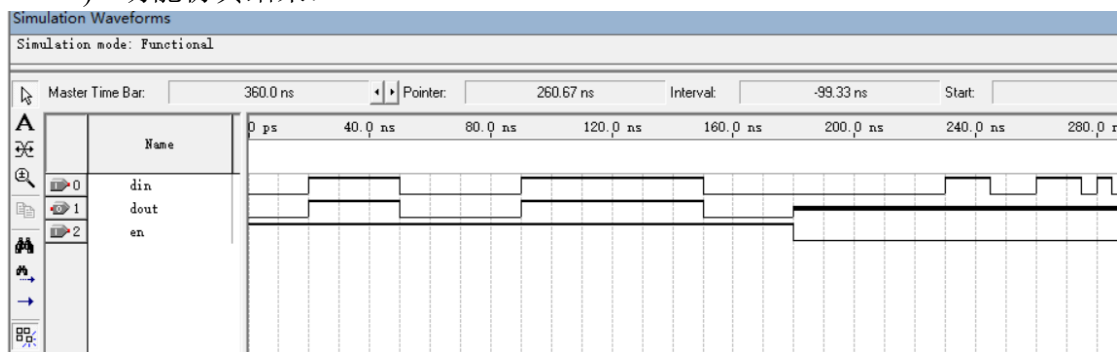


图 8-25 三态门功能仿真图

b) 时序仿真结果：

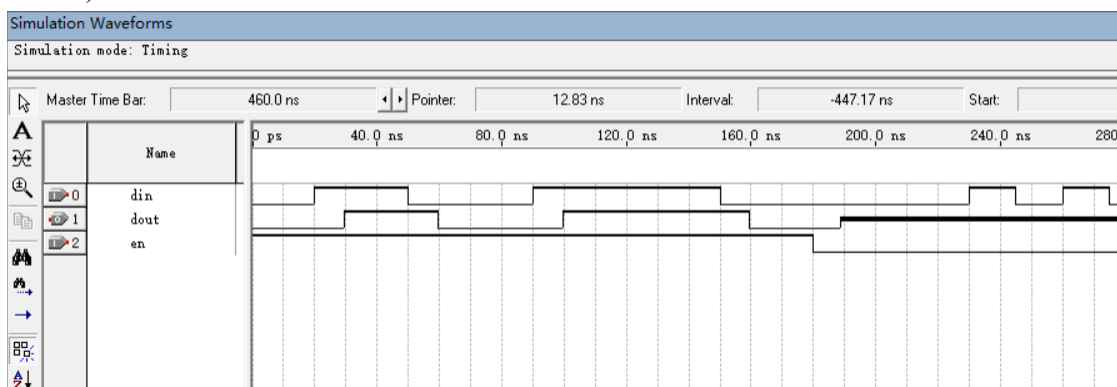


图 8-26 三态门时序仿真图

7.5 8-3 编码器

一、8-3 编码器简介：

在数字系统里,此次需要将某一信息变换成某一特定的代码。把二进制码按照一定的规律编排，如 8421 嘛/格雷码等，使每组代码具有一特定的含义称为编码。具有编码功能的路基电路称为编码器。

编码器是将 2 的 n 次方个分离的信息代码以 n 个二进制码来表示。如 8-3 编码器有 8 个输入、3 个二进制输出。其真值表如下：

表 7.4 8-3 编码器的真值表

输入								输出		
10	11	12	13	14	15	16	17	Y2	Y1	Y0
1	0	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	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

二、8-3 编码器的设计:

输入信号: 信号输入端 x[7:0]

输出信号: 输出信号端 y[2:0]

本例程的 Verilog 程序:

```

module mb_83(x, y);
    input[7:0] x;           //输入变量x
    output[2:0] y;          //输出变量y

    reg[2:0] y;             //输出变量y寄存器

    always@(x)              //电平触发, 若x改变则执行以下操作
    begin
        case(x)             //判断x的取值
            8'b00000001:y=3'b000;    //当x=8'b00000001, 则y输出为3'b000
            8'b00000010:y=3'b001;    //当x=8'b00000010, 则y输出为3'b001
            8'b00000100:y=3'b010;    //当x=8'b00000100, 则y输出为3'b010
            8'b00001000:y=3'b011;    //当x=8'b00001000, 则y输出为3'b011
            8'b00010000:y=3'b100;    //当x=8'b00010000, 则y输出为3'b100
            8'b00100000:y=3'b101;    //当x=8'b00100000, 则y输出为3'b101
            8'b01000000:y=3'b110;    //当x=8'b01000000, 则y输出为3'b110
            8'b10000000:y=3'b111;    //当x=8'b10000000, 则y输出为3'b111
            default: y=3'b000;    //其他情况下, 则y输出为3'b000
        endcase
    end
endmodule

```

三、8-3 编码器的波形仿真:

a) 功能仿真结果:

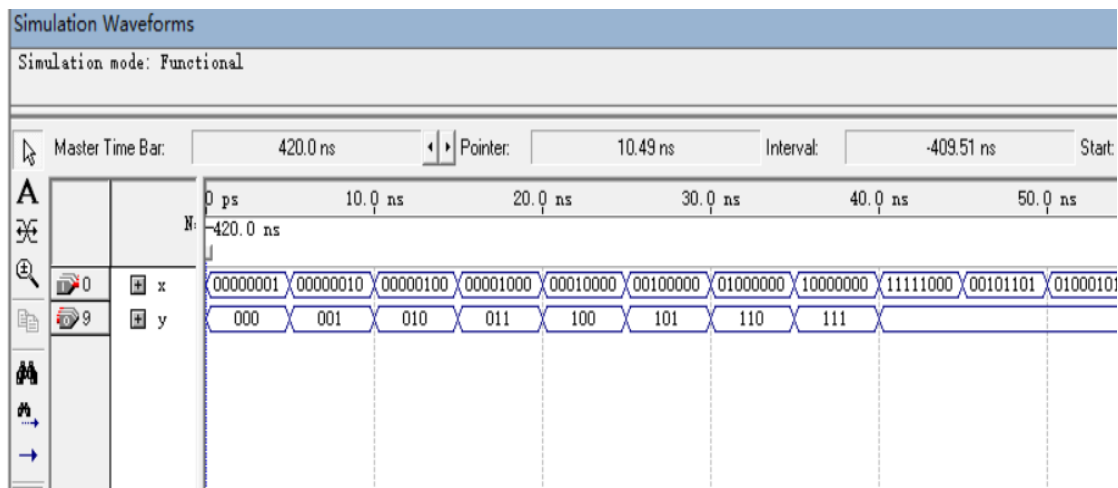


图 8-27 8-3 编码器功能仿真图

b) 时序仿真结果:

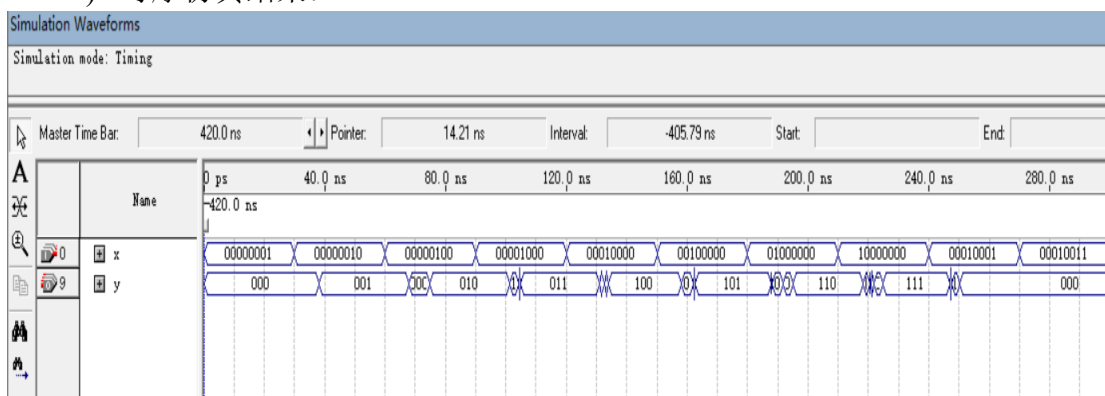


图 8-28 8-3 编码器时序仿真图

7.6 8-3 优先编码器

一、8-3 优先编码器简介:

普通编码器有一个缺点,即在某一时刻只允许有一个有效的输入吸纳后,若同时有两个或者两个以上的输入信号要求编码,输出端一定会发生混乱,出现错误。为了解决这一问题,人们设计了优先编码器。优先编码器的功能是允许同时在几个输入端有输入信号,编码器按照输入信号预先排定的优先顺序,只对同时输入的几个信号中优先权高位的一个信号编码。

下面以 8-3 优先编码器为例,介绍优先编码器的设计方法。8-3 优先编码器的真值表如下:

表 7.5 8-3 优先编码器的真值表

输入									输出				
E1	10	11	12	13	14	15	16	17	Y2	Y1	Y0	EO	GS
1	X	X	X	X	X	X	X	X	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1	1	1	0	1
0	X	X	X	X	X	X	X	0	0	0	0	1	0
0	X	X	X	X	X	X	0	1	0	0	1	1	0
0	X	X	X	X	X	0	1	1	0	1	0	1	0
0	X	X	X	X	0	1	1	1	0	1	1	1	0
0	X	X	X	0	1	1	1	1	1	0	0	1	0
0	X	X	0	1	1	1	1	1	1	0	1	1	0
0	X	0	1	1	1	1	1	1	1	1	0	1	0
0	0	1	1	1	1	1	1	1	1	1	1	1	0

二、8-3 优先编码器的设计:

输入信号: 信号输入端 i[7:0];

输入使能端 ei;

输出信号: 输出信号端 y[2:0];

输出使能端 eo;

优先标志端 gs。

本例程的 Verilog 程序:

```

module yxbm_83(y, eo, gs, i, ei);
input [7:0] i;           //8位输入i
input ei;                //使能输入端ei
output eo, gs;           //使能输出端eo, 优先标志端gs
output[2:0] y;           //3位输出y
reg[2:0] y;              //3位输出寄存器y
reg eo, gs;              //使能输出寄存器,
always@(i, ei)           //电平触发方式, 当i跟ei有改变的时候, 执行以下操作
begin
    if(ei==1'b1) //当ei为1的时候
    begin
        y<=3'b111;
        gs<=1'b1;
        eo<=1'b1;
    end
    else
    begin
        if(i[7]==1'b0) //当i的第8为0时候

```

```
begin
    y<=3'b000;
    gs<=1'b0;
    eo<=1'b1;
end
else if(i[6]==1'b0) //当i的第七位为0时候
begin
    y<=3'b001;
    gs<=1'b0;
    eo<=1'b1;
end
else if(i[5]==1'b0) //当i的第6位为0时候
begin
    y<=3'b010;
    gs<=1'b0;
    eo<=1'b1;
end
else if(i[4]==1'b0) //当i的第5位为0时候
begin
    y<=3'b011;
    gs<=1'b0;
    eo<=1'b1;
end
else if(i[3]==1'b0) //当i的第4位为0时候
begin
    y<=3'b100;
    gs<=1'b0;
    eo<=1'b1;
end
else if(i[2]==1'b0) //当i的第3位为0时候
begin
    y<=3'b101;
    gs<=1'b0;
    eo<=1'b1;
end
else if(i[1]==1'b0) //当i的第2位为0时候
begin
    y<=3'b110;
    gs<=1'b0;
    eo<=1'b1;
end
end
```



```

else if(i[0]==1'b0)//当i的第1位为0时候
begin
    y<=3'b111;
    gs<=1'b0;
    eo<=1'b1;

end
else if(i==8'b11111111)//当i为8'b11111111时候
begin
    y<=3'b111;
    gs<=1'b1;
    eo<=1'b0;

end
end
endmodule

```

三、8-3 优先编码器的功能仿真波形：

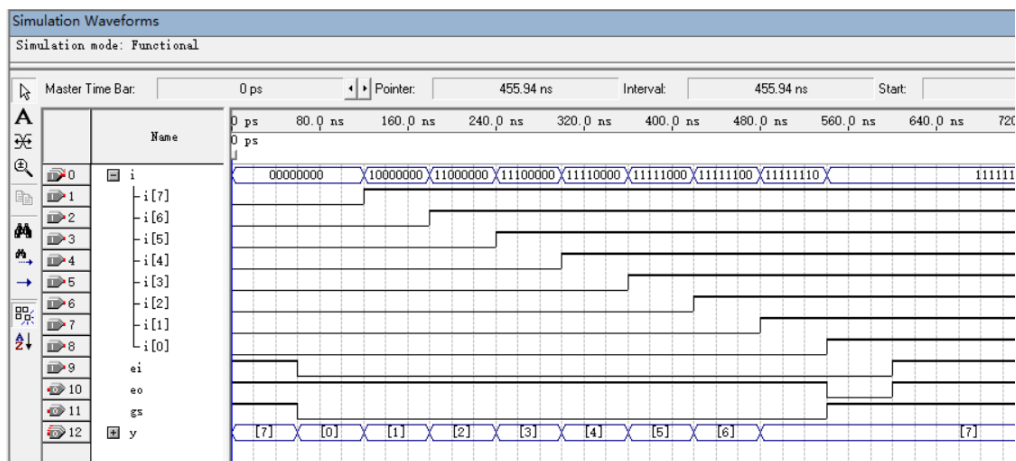


图 8-29 8-3 优先编码器功能仿真图

7.7 3-8 译码器

一、3-8 译码器简介：

译码是编码的逆过程，它的功能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，具有译码功能的逻辑电路称为译码器。

本例程中以 3-8 译码器为例子。如果输入为 n 个二进制选择线，则最多可译码转换成 2 的 n 次方个数据。其真值表如下：

表 7.6 3-8 译码器的真值表

输入						输出							
G1	G2	G3	A2	A1	A0	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
X	1	X	X	X	X	1	1	1	1	1	1	1	1
X	X	1	X	X	X	1	1	1	1	1	1	1	1
0	X	X	X	X	X	1	1	1	1	1	1	1	1
1	0	0	0	0	0	1	1	1	1	1	1	1	0
1	0	0	0	0	1	1	1	1	1	1	1	0	1
1	0	0	0	1	0	1	1	1	1	1	0	1	1
1	0	0	0	1	1	1	1	1	1	0	1	1	1
1	0	0	1	0	0	1	1	1	0	1	1	1	1
1	0	0	1	0	1	1	1	0	1	1	1	1	1
1	0	0	1	1	0	1	0	1	1	1	1	1	1
1	0	0	1	1	1	0	1	1	1	1	1	1	1

二、3-8 译码器的设计:

本例程的以具有 3 个使能端的 3-8 译码器为例。

其中输入信号有:

3 位编码输入端 a[2:0]、使能输入端 g1, g2, g3;

输出信号有:

8 位编码输出端 y[7:0]。

本例程的 Verilog 程序为:

```

module ym_3_8(a, g1, g2, g3, y);
input [2:0] a;           //3位2进制编码输入端
input  g1, g2, g3;       //3个使能输入端
output [7:0] y;          //8位编码输出端
reg [7:0] y;
always@(a or g1 or g2 or g3) //电平触发方式
begin
    if(g1==0) y=8'b11111111; //如果g1为0, 则y输出为11111111
    else if(g2==1) y=8'b11111111; //如果g2为1, 则y输出为11111111
    else if(g3==1) y=8'b11111111; //如果g3为1, 则y输出为11111111
    else
        case(a[2:0]) //判断a的值, 并通过a的值来给y设置输出值
            3'b000:y[7:0]=8'b11111110;
            3'b001:y[7:0]=8'b11111101;
            3'b010:y[7:0]=8'b11111011;
            3'b011:y[7:0]=8'b11110111;
            3'b100:y[7:0]=8'b11101111;
            3'b101:y[7:0]=8'b11011111;
        endcase
    end
end

```

```

3'b110:y[7:0]=8'b10111111;
3'b111:y[7:0]=8'b11111111;
endcase
end
endmodule

```

三、3-8 译码器的功能仿真波形：

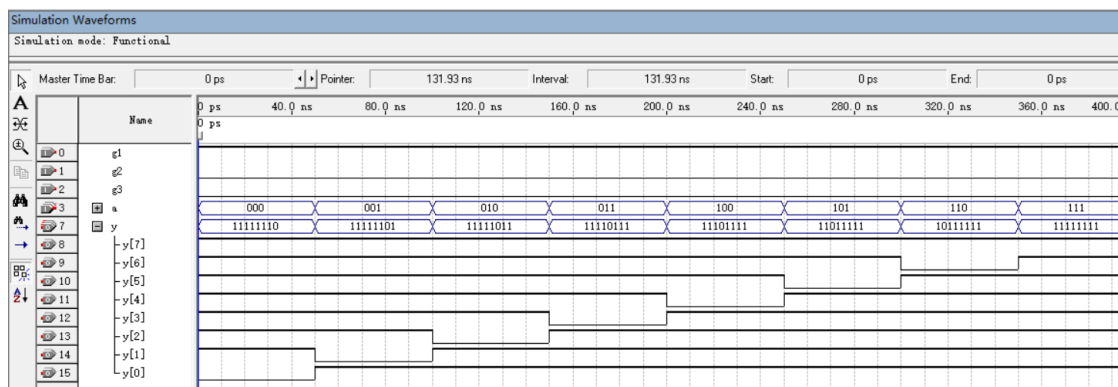


图 8-30 3-8 译码器功能仿真图

7.8 移位寄存器

一、移位寄存器简介：

移位寄存器就是指寄存器里面存储的二进制数据能够在时钟信号的控制下进行一次左移或者右移，在数字电路中通常用于数据的串/并转换、数值运算等。移位寄存器按照不同的分类方法可以分为不同的类型。如果按照移位寄存器的移位方向来进行分类，可以分为左移移位寄存器、右移移位寄存器和双向移位寄存器等；如果按照工作方式来分类，可以分为串入/串出移位寄存器、串入/并出移位寄存器和并入/串出移位寄存器等。

二、移位寄存器的设计：

本例程以异步清零的 4 位并入串出移位寄存器为例介绍一种并入串出寄存器的设计方法。所谓的并入串出寄存器是指输入为并行数据，而输出为串行数据。

本例程的输入信号有：

时钟输入端 `clk`、

清零端 `clr`、

四位并行数据输入端：`din[3:0]`、

一位串行数据输出端：`dout`。

本例程的 Verilog 程序为：

```

module reg_bc(clk, clr, din, dout);
input clk, clr;           //输入时钟端，清零端（高电平有效）
input[3:0] din;           //数据输入端
output dout;              //数据输出端
reg[1:0] cnt;
reg[3:0] q;
reg dout;
always@(posedge clk)      //时钟上升沿触发
begin
    cnt<=cnt+1;           //cnt自加1
    if(clr)                //判断清零信号是否有效
    begin
        q<=4'b0000;       //q置0
    end
    else
    begin
        if(cnt>0)          //判断cnt是否大于0
        begin
            q[3:1]<=q[2:0]; //q中的值向左移1位
        end
        else if(cnt==2'b00) //判断cnt是否为0
        begin
            q<=din;         //把din的值赋予q
        end
        dout<=q[3];         //把q的最高位输出
    end
end
endmodule

```

三、移位寄存器的功能仿真波形：

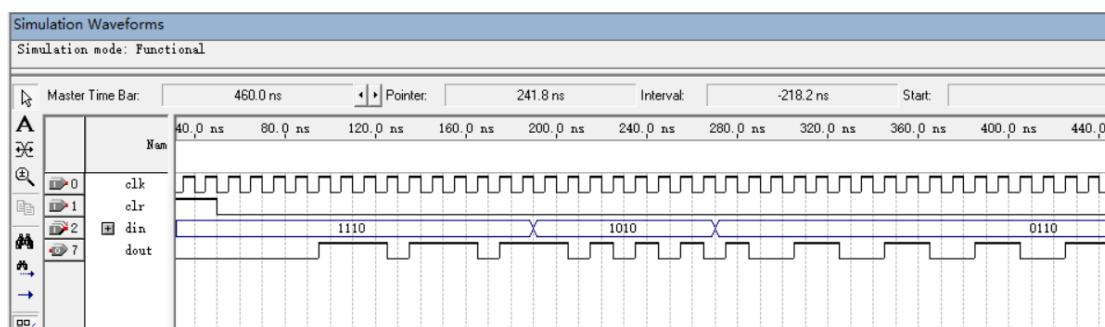


图 8-31 移位寄存器功能仿真图

7.9 多路选择器

一、多路数据选择器简介:

数据选择器是指经过选择,把多个通道的数据传到唯一的公共数据通道上。实现数据选择功能的数字逻辑电路称为数据选择器,它的作用相当于多个输入的单刀多掷开关。

二、多路数据选择器的设计:

本例程以四选一数据选择器为例。四选一数据选择器是对 4 个数据源进行选择,使用两位地址 A1A0 产生 4 个地址信号,由 A1A0 等于“00”、“01”、“10”、“11”来选择输出,四选一数据选择器真值表如下:

表 7.7 四选一数据选择器的真值表

输入			输出
使能	地址		Y
G	A1	A0	
0	X	X	0
1	0	0	D0
1	0	1	D1
1	1	0	D2
1	1	1	D3

本例程的输入信号为:

数据源 d0、d1、d2、d3;

两位地址码 a[1:0];

使能端 g;

输出信号: 选择输出端 y。

本例程的 Verilog 程序为:

```

module mux4(d0, d1, d2, d3, y, a, g);
    input d0, d1, d2, d3;           //输入四个数据
    input g;                        //输入使能端
    input[1:0] a;                   //输入的选择端
    output y;                       //输出数据
    reg y;                          //输出数据寄存器
    always@(d0 or d1 or d2 or d3 or g or a) //电平触发, 当d0, d1, d2, d3, a有变化就触发
    begin
        if(g==1'b0)                 //当g为0时候
            y=1'b0;                 //y置0
        else
            case(a)                  //判断a的取值, 并作出相应的赋值

```

```

2'b00:y=d0;
2'b01:y=d1;
2'b10:y=d2;
2'b11:y=d3;
default:y=1'b0;           //a为其他值得条件下，y赋予0
endcase

end

endmodule

```

三、多路数据选择器的功能仿真波形：

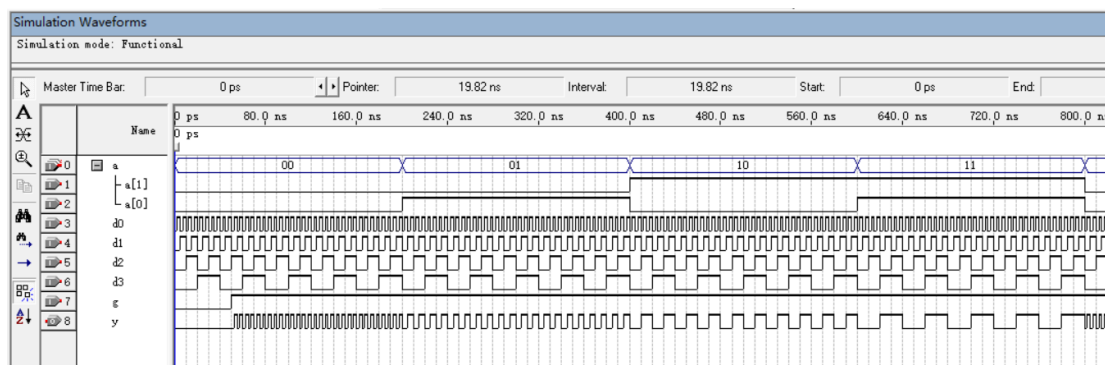


图 8-32 四选一数据选择器功能仿真图

7.10 串行加法器

一、串行加法器简介：

加法器是一种较为常见的算术运算电路，更是计算机中不可或缺的组成成分，包括半加器、全加器、多位全加器等。而全加器器即是指能进行加数、被加数和低位来的进位信号相加，并根据求和结果给出该进位的信号。

二、串行加法器的设计：

本例程以四位全加器为例，介绍一下串行加法器的设计方法。

其中输入信号有：

被加数；

加数 b；

低位进位 ci。

输出信号有：

和数 s；

进位 co。

本例程的 Verilog 程序为：

```

module add4(a,b,ci,s,co);
input [3:0] a,b;           //输入四位数据a, b
input ci;                  //输入进位ci
output [3:0] s;            //输出四位数据 s
output co;                 //输出进位co
assign {co,s}=a+b+ci;      //把相加后的结果赋予co、s，其中co放最高位，s放低三位
endmodule

```

三、串行加法器的功能仿真波形：

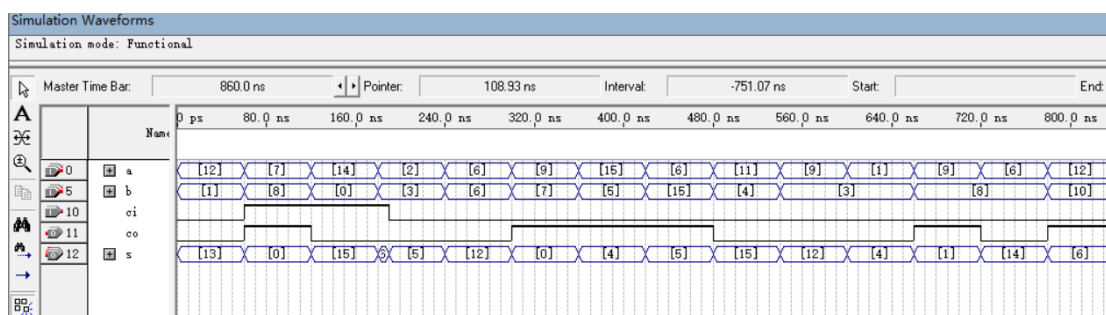


图 8-33 四位全加器功能仿真图

7.11 简单运算单元 ALU

简单运算单元 ALU 简介：

一、运算器：arithmetic unit，计算机中执行各种算术和逻辑运算操作的部件。运算器的基本操作包括加、减、乘、除四则运算，与、或、非、异或等逻辑操作，以及移位、比较和传送等操作，亦称算术逻辑部件（ALU）。

二、简单运算单元 ALU 设计：

本例程以具有 11 种简单运算单元的电路为例，设计一种通过选择端来选择运算功能的简单运算单元 ALU。

其中输入信号有：

数据输入 in1[7:0]、

数据输入 in2[7:0]、

运算操作选择 op[3:0]、

输出信号有：数据输出 out[15:0]。

本例程的 Verilog 程序如下：

```

module ALU(in1,in2,op, out);
input [7:0] in1,in2;           //两个8位输入数据
input [3:0] op;                //运算操作操作选择
output [15:0] out;             //一个16位输出数据

```

```

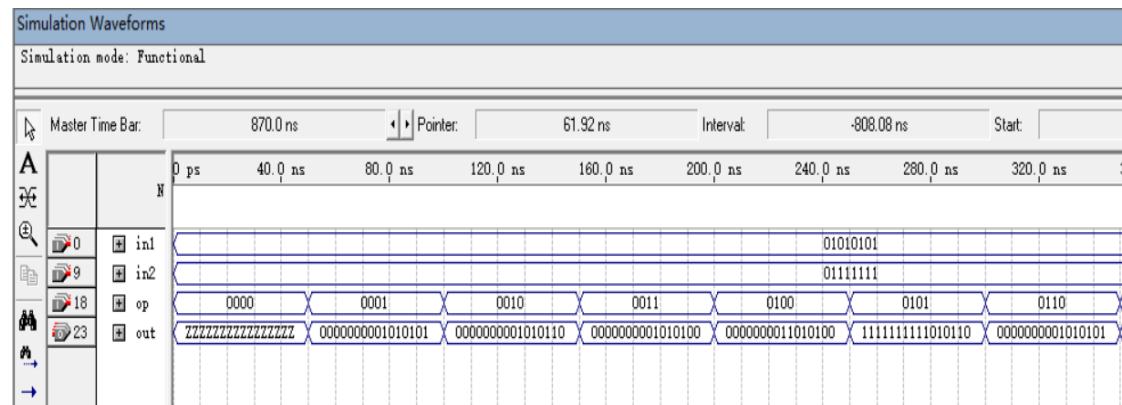
wire [7:0] in1,in2;
wire [3:0] op;
reg [15:0] out;
//将操作符定义为参数
parameter transfer = 4'b0001,
            increase = 4'b0010,
            decrease = 4'b0011,
            addition = 4'b0100,
            subtraction = 4'b0101,
            AND = 4'b0110,
            OR = 4'b0111,
            XOR = 4'b1000,
            NOT = 4'b1001,
            shift_left = 4'b1010,
            shift_right = 4'b1011;

always @(in1 or in2 or op)      //电平触发方式
begin
    case (op)                  //case语句选择运算操作
        transfer:             out=in1;      //赋值操作
        increase:             out=in1+1'b1;  //in1自加一运算
        decrease:             out=in1-1'b1;  //in1自减一运算
        addition:             out=in1+in2;  //in1与in2相加运算
        subtraction:         out=in1-in2;  //in1与in2相减运算
        AND:                  out=in1&in2;  //相与运算
        OR:                   out=in1|in2;  //相或运算
        XOR:                   out=in1^in2;  //异或运算
        NOT:                   out=~ in1;    //取非运算
        shift_left:           out=in1<<1;   //左移运算
        shift_right: out=in1>>1;           //右移运算
        default:              out=16'bz;    //如果op为其他值的话，out设置为高阻态
    endcase
end

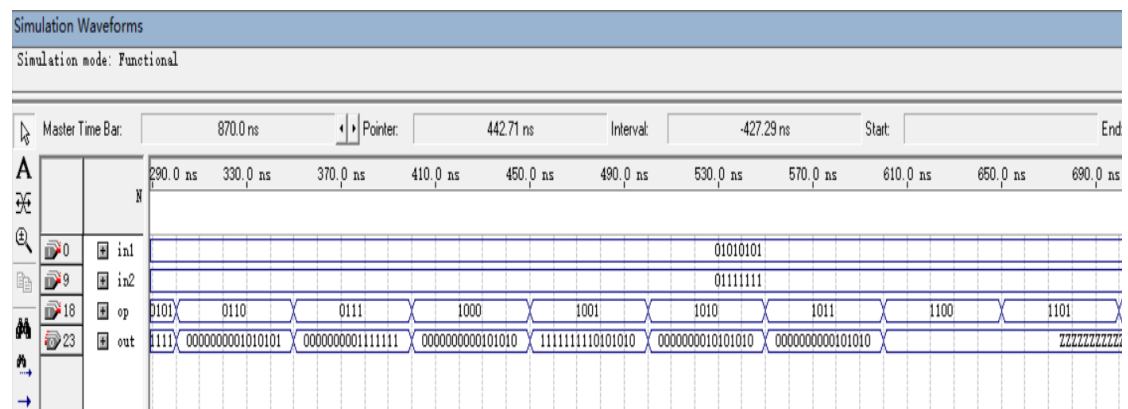
endmodule

```

三、简单运算单元 ALU 功能仿真波形：



(a)



(b)

图 8-34 简单运算单元 ALU 功能仿真图

第八章 开发板基础实验

在数字电路基础实验中，因为设计功能简单，只需要一个设计模块文件就可以实现，但是在往后进行复杂一点的设计时只用一个文件是不好实现的，需要划分模块进行设计，一般一个复杂系统设计可以划分多个模块，包括功能模块、控制模块、存储模块，最后构成系统组合模块。模块划分技巧非常重要，好的模块结构，能极大地精简各模块的结构，从而能用最少的代码实现所需的功能；使各模块顺畅运行，保证系统更加稳定。

模块划分的一般原则：模块内部联系紧密，各个模块的功能尽量独立，模块间连接尽量简单。按功能划分模块是模块划分最基本的指导思想，当我们拿到一个项目后，最直接的想法就是先确定项目需要多少小功能为之服务，然后把一个个功能实现，最后通过顶层模块的例化，完成项目要求。

在确定模块划分后，需要明确模块的端口，及模块与模块之间的数据交互信号。端口命名主要原则是要求从命名中看出端口的主要作用，从而增加程序的可读性。笔者在实际项目经验总结得到了一般模块端口信号命名规范，读者在完成项目模块划分后，确定端口及数据流向时可参考使用。

表 8.1 端口信号命名规范

信号名称	作用	举例说明
CLK	系统时钟	
RSTn	系统复位信号	
XXX_Data	两位以上数据	SMG_Data、 Write_Data、 ROM_Data、
XXX_Addr	地址数据	ROM_Addr、
XXX_Sig	一位标志信号	Read_Sig 、 Write_Sig 、 Done_Sig 、 Start_Sig、 EN_Sig、 Empty_Sig、 Full_Sig
XXX_Out	输出信号	Data_Out、 LED_Out
XXX_IN	输入信号	Key_IN、 Data_IN

8.1 LED 流水灯

实验硬件说明：

LED(Light Emitting Diode) 发光二极管，是一种能够将电能转化为可见光的固态的半导体器件，它可以直接把电转化为光；它是正向导通(正极到负极)反向截止(负极到正极)的器件。对于直插式 LED，长脚为正极，短脚为负极；对于贴片式 LED，有绿点的一端为负极。在正常工作情况下，常规的绿，蓝，白，暖白 LED 灯的导通电压为 3.0-3.5V，红，黄 LED 灯为 2.5-2.8V；电流一般 15 毫安为宜，一般高亮 LED 采用的限流电阻是 400-500 欧姆，普通的采用 1K 左右的就行了，加大限流电阻可以降低 LED 的亮度，可根据需要配置。发光二极管的反向击穿电压约 5 伏，LED 的特点非常明显，寿命长、光效高、低辐射与低功耗。白光 LED 的光谱几乎全部集中于可见光频段，其发光效率可超过 150lm/W。

实验的原理图：

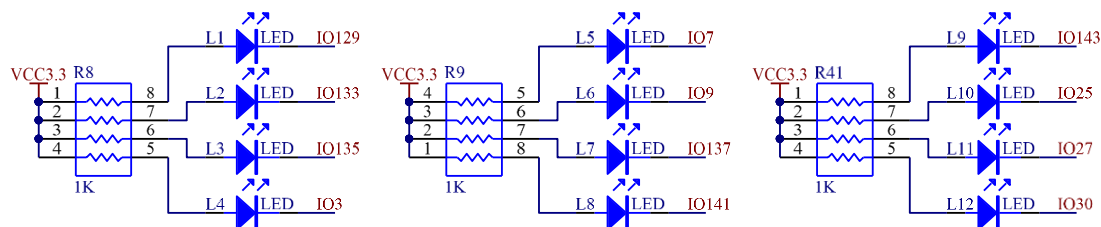


图 8-1 底板发光二极管电路原理图

通过原理图可以看出，当 LED 的负极为低电平时 LED 导通，否则截至，通过这我们便知道 LED 的驱动原理。

1. 实验设计原理

LED 流水灯，无非就是控制 12 个 LED 灯亮灭的顺序，通常采用的方法有移位、循环赋值等等。

移位方式：首先产生移位时钟，该时钟的频率不能够高，否则 LED 的显示速度过快，达不到预期效果，通过移位时钟控制 led_temp 循环右移。

循环赋值：首先产生显示时钟，该时钟的频率不能够高，否则 LED 的显示速度过快，达不到预期效果，通过显示时钟，在每个时钟的上升沿向 LED 赋值。

2. 实验设计要求

控制开发板底板 12 个 LED 灯轮流点亮，实现流水灯效果；控制 12 个 LED 按照一定模式点亮，实现花样流水灯效果。

3. 模块划分：

分频器模块：产生低频信号，控制 LED 流水灯亮灭速度；

流水灯功能模块：实现 LED 灯亮灭。

4. 实验程序

1) 移位方式：

a) 顶层原理图组合模块如下图。

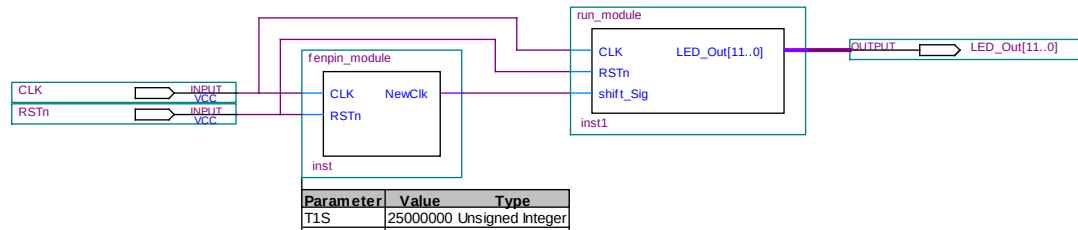


图 8-1 流水灯顶层原理图组合模块

b) 分频器模块 verilog 程序：

```

module fcpin_module
(
    CLK, RSTn, NewClk
);
input CLK;
input RSTn;
output NewClk;

/*****/
parameter T1S = 26'd50000_000;
/*****/
reg[25:0]Count1;
always @(posedge CLK or negedge RSTn)
    if(!RSTn)
        Count1 <= 26'd0;
    else if(Count1 == T1S)
        Count1 <= 26'd0;
    else
        Count1 <=Count1 +1'b1;
/*****/
reg rNewClk;

always @(posedge CLK or negedge RSTn)
    if(!RSTn)
        rNewClk <= 1'b0;
    else if(Count1 == T1S)
        rNewClk <= 1'b1;
    else

```

```

    rNewClk <= 1'b0;
    /*******//
assign NewClk =rNewClk;
endmodule

```

c) 流水灯功能模块模块 verilog 程序:

```

module run_module
(
    CLK, RSTn, shift_Sig, LED_Out
);
input CLK;
input RSTn;
input shift_Sig;
output [11:0] LED_Out;
reg[11:0] rLED_Out;
always @(posedge CLK or negedge RSTn)
begin
    if(!RSTn)
        rLED_Out <= 12'b1111_11111110;
    else if(shift_Sig)
        begin
            rLED_Out <= {rLED_Out[10:0], rLED_Out[11]};
            //rLED_Out <= rLED_Out<<1; //实现循环移位操作
            //rLED_Out[0]<=rLED_Out[7];
        end
    end
end

assign LED_Out =rLED_Out;
endmodule

```

2) 循环赋值方式:

a) 顶层原理图组合模块如下图。

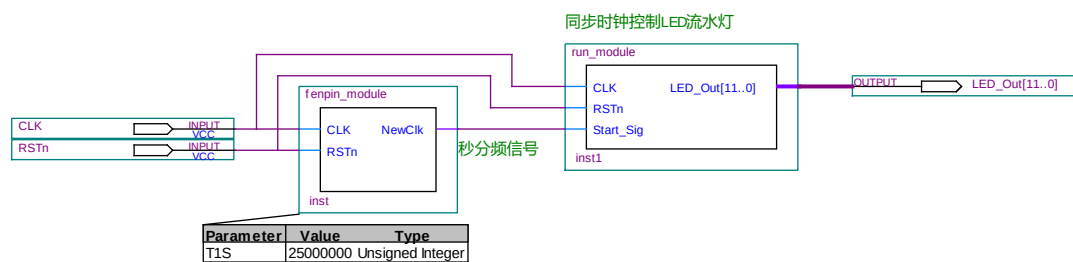


图 8-2 循环赋值方式顶层原理图组合模块

b) 分频器模块 verilog 程序：与上面移位方式程序相同（略）。

c) 流水灯功能模块 verilog 程序，采用 case 语句实现：

```
module run_module( CLK, RSTn, Start_Sig, LED_Out);
input CLK;
input RSTn;
input Start_Sig;
output [11:0] LED_Out;
reg[11:0] rLED_Out;
reg[5:0] i;

always @(posedge CLK or negedge RSTn)
begin
    if(!RSTn)
    begin
        rLED_Out <= 12'b1111_11111110;
        i <= 6'd0;
    end
    else if(Start_Sig)
    begin
        i <= i+1'b1; //i计数器，用于切换流水灯状态
        case(i)
            6'd0: rLED_Out <= 12'b1111_11111110; //实现花样流水灯
            6'd1: rLED_Out <= 12'b1111_111111101;
            6'd2: rLED_Out <= 12'b1111_11111011;
            6'd3: rLED_Out <= 12'b1111_11110111;
            6'd4: rLED_Out <= 12'b1111_11101111;
            6'd5: rLED_Out <= 12'b1111_11011111;
            6'd6: rLED_Out <= 12'b1111_10111111;
            6'd7: rLED_Out <= 12'b1111_01111111;
            6'd8: rLED_Out <= 12'b1110_11111111;
            6'd9: rLED_Out <= 12'b1101_11111111;
            6'd10: rLED_Out <= 12'b1011_11111111;
            6'd11: rLED_Out <= 12'b0111_11111111;
            6'd12: rLED_Out <= 12'b1011_11111111;
            6'd13: rLED_Out <= 12'b1101_11111111;
            6'd14: rLED_Out <= 12'b1110_11111111;
            6'd15: rLED_Out <= 12'b1111_01111111;
            6'd16: rLED_Out <= 12'b1111_10111111;
            6'd17: rLED_Out <= 12'b1111_11011111;
            6'd18: rLED_Out <= 12'b1111_11101111;
            6'd19: rLED_Out <= 12'b1111_11110111;
```

```

6' d20: rLED_Out<=12' b1111_11111011;
6' d21: rLED_Out<=12' b1111_11111101;
6' d22: rLED_Out<=12' b1111_11111110;
6' d23: rLED_Out<=12' b1111_11111100;
6' d24: rLED_Out<=12' b1111_11111000;
6' d25: rLED_Out<=12' b1111_11110000;
6' d26: rLED_Out<=12' b1111_11100000;
6' d27: rLED_Out<=12' b1111_11000000;
6' d28: rLED_Out<=12' b1111_10000000;
6' d29: rLED_Out<=12' b1111_00000000;
6' d30: rLED_Out<=12' b1110_00000000;
6' d31: rLED_Out<=12' b1100_00000000;
6' d32: rLED_Out<=12' b1000_00000000;
6' d33: begin rLED_Out<=12' b0000_00000000; i<=6' d0;end
endcase
end

end

/*****
assign LED_Out =rLED_Out;
endmodule

```

8.2 按键消抖

1. 实验硬件说明

按键原理图:

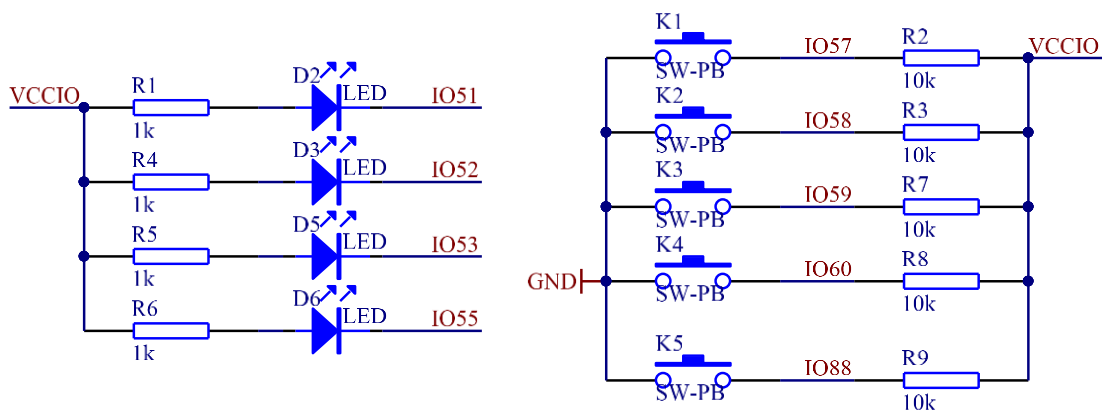


图 8-2 FPGA 核心板上 LED 与按键原理图

2. 实验设计原理

按键按下的示意图如图 8-3 所示，高电为平按键默认状态，按键一旦按下，“按下事件”就发生了，电平随之发生抖动，抖动周期大约为 10ms。事后，如果按键依然按着不放，那么电平便会处于低电平。换之，如果按键此刻被释放，那么

“释放事件”发生了，电平随之由低变高，然后发生抖动，抖动周期大约为 10ms。

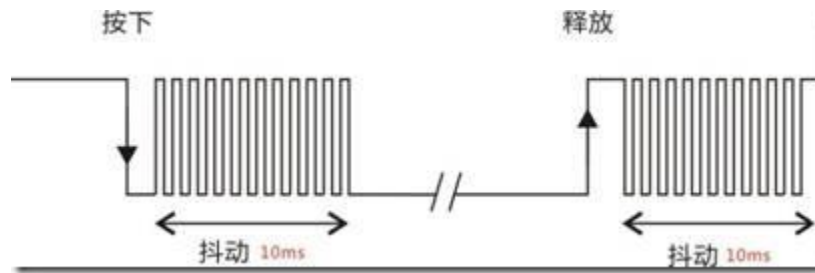


图 8-3 按键活动的时序示意图。

按键消抖处理一般有 3 个工作要做，亦即检测电平变化，过滤抖动，还有产生有效按键。

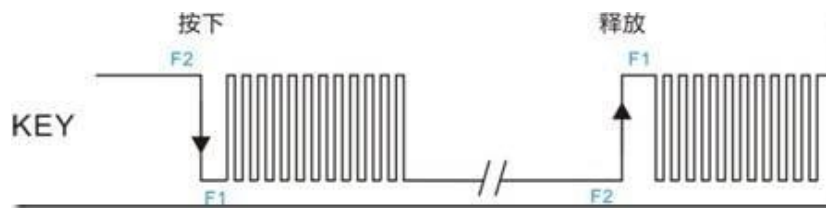


图 8-4 按键电平变化，按下事件与释放事件。

a)检测电平变化就是用来察觉“按下事件”还有“释放事件”，或者监控电平的状态变化。如图 8-4 所示，可以建立一组寄存器 F1~F2，F1 暂存当前的电平状态，F2 则暂存上一个时钟的电平状态。Verilog 语言可以如下代码所示：

```
reg F2,F1;
always @( posedge CLK )
    { F2,F1 } <= { F1,Pin_In };//暂存当前的电平状态
```

根据图 8-4 的显示，按下事件是 F2 为 1 值，F1 为 0 值；释放事件则是 F2 为 0 值，F1 为 1 值。为了不要错过电平变化的黄金时间，“按下事件”还有“释放事件”必须作为“即时“，为此 Verilog 语言可以如下代码所示：

```
wire H2L_Sig = ( F2 == 1 && F1 == 0 );//判断按键下降沿信号
wire L2H_Sig = ( F2 == 0 && F1 == 1 );//判断按键上升沿信号
```

b) 过滤抖动就是也可以称为延迟抖动，常规又廉价的机械按键，抖动时间大约是 10ms 之内，如图 8-3 所示。抖动一般都发生在“按下事件”或者“释放事件”以后，过滤抖动就是延迟若干时间即可。Verilog 语言可以如下代码所示：

```
4'd1:
    if(Count_MS == 8'd10)begin isCount<=1'b0;i <=i+1'b1;end //延时
    else isCount <= 1'b1;
```


c) 产生有效按键: 产生有效按键亦即有效的按键事件, 如图 2.4 所示, 按下有效 isPress 信号, 还有释放有效 isRelease 信号, 两个信号分别都是拉高一个时钟。

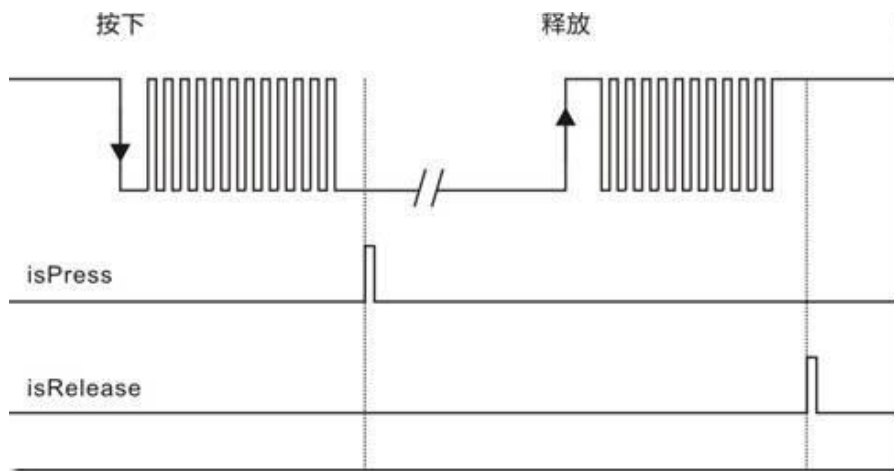


图 8-5 产生有效按键图

除了常见的按下有效或者释放有效以外, 根据设计要求, 有效按键按也有其它, 例如: 按键按下两下有效 (双击), 按键按下一段时间有效 (长击)。本例程根据最常用的设计只产生按下有效脉冲信号, 稍作修改就可以产生释放有效脉冲信号或其他信号。Verilog 语言可以如下代码所示:

```
4'd2: //一个时钟周期的高电平脉冲
begin rPin_Out <= 1'b1; i <= i+1'b1; end
4'd3:
begin rPin_Out <= 1'b0; i <= i+1'b1; end
```

3. 实验设计要求

设计按键消抖模块, 验证按键消抖, 每按一次按键 FPGA 核心板上 4 个 LED 灯轮流点亮一个, 实现按键控制流水灯效果。

4. 模块划分:

按键消抖模块: 按键按下, 产生按下脉冲信号;

流水灯功能模块: 实现 LED 灯亮灭。

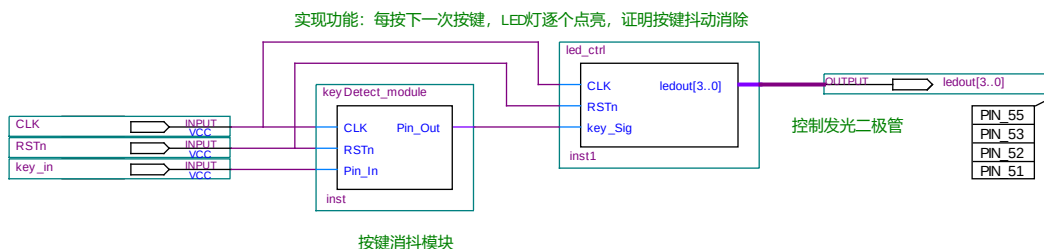


图 8-6 顶层原理图组合模块系统连接图

5. 实验程序

a) 按键消抖模块 verilog 程序:

```

module keyDetect_module(CLK, RSTn, Pin_In, Pin_Out);
input  CLK;
input  RSTn;
input  Pin_In;
output Pin_Out;

/*****/
reg F2,F1;
always @ ( posedge CLK )
    { F2,F1 } <= { F1,Pin_In };

/*****/
wire H2L_Sig = ( F2 == 1 && F1 == 0 );//判断按键下降沿信号
wire L2H_Sig = ( F2 == 0 && F1 == 1 ); //判断按键上升沿信号
/*****/
reg[15:0]Count1;
always @(posedge CLK or negedge RSTn)
begin
    if(!RSTn)
        Count1 <= 16'd0;
    else if(isCount && Count1 == 16'd49999) //毫秒计数
        Count1 <= 16'd0;
    else if(isCount)
        Count1 <= Count1 + 1'b1;
    else if(!isCount)
        Count1 <= 16'd0;
end

/*****/

reg [7:0] Count_MS;

always @(posedge CLK or negedge RSTn)
begin
    if(!RSTn)
        Count_MS <= 8'd0;
    else if(isCount && Count1 == 16'd49999)//毫秒计时
        Count_MS <= Count_MS + 1'b1;
    else if(!isCount)
        Count_MS <= 11'd0;
end

```

```

    /*****/
    reg isCount; //计时使能信号
    reg rPin_Out;
    reg[3:0] i;
    always @(posedge CLK or negedge RSTn)
    begin
        if(!RSTn)
        begin
            isCount <= 1'b0;
            rPin_Out <= 1'b0;
            i <= 4'd0;
        end
        else
        case(i)
            4'd0:
                if(H2L_Sig) i <= 4'd1;    //按键按下脉冲
                else if(L2H_Sig) i <= 4'd3;    //按键释放脉冲
            4'd1:
                if(Count_MS == 8'd10)begin isCount<=1'b0;i <=i+1'b1;end //消抖延时
                else isCount <= 1'b1;
            4'd2:
                begin rPin_Out <= 1'b1; i <=i+1'b1; end
                //一个时钟周期的高电平脉冲
            4'd3:
                begin rPin_Out <= 1'b0; i <=i+1'b1; end
            4'd4:
                if(Count_MS ==8'd50)begin isCount<=1'b0; i <= 4'd0;end //消抖延时
                else isCount <= 1'b1;

        endcase
    end
    /*****/
    assign Pin_Out = rPin_Out; //按键脉冲输出
    /*****/
endmodule

```

b) 流水灯功能模块 verilog 程序:

```

module led_ctrl(CLK, RSTn, key_Sig, ledout);
    input CLK, RSTn;
    input key_Sig;
    output reg[3:0] ledout;

```

```

reg[1:0]couter;
always@(posedge CLK or negedge RSTn)
begin
    if(!RSTn)begin ledout='b1110;end
    else if(key_Sig)
    begin
        couter=couter+1'b1;
        case(couter)
            2'd0:begin ledout='b1110;end
            2'd1:begin ledout='b1101;end
            2'd2:begin ledout='b1011;end
            2'd3:begin ledout='b0111;end
            default:begin ledout='b1111;end
        endcase
    end
end
endmodule

```

8.3 PWM 控制灯的亮暗

实验硬件说明:

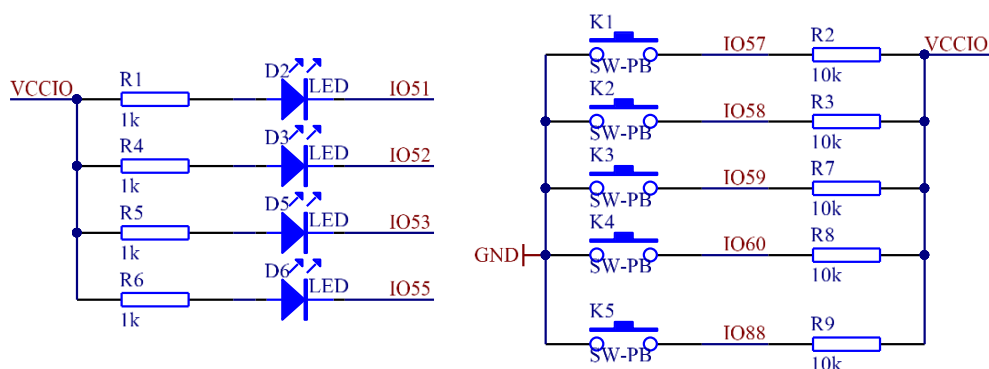


图 8-3 FPGA 核心板上 LED 与按键原理图

通过原理图可以看出，当 LED 的负极为低电平时 LED 导通，否则截至，通过这我们便知道 LED 的驱动原理。

1. 实验设计原理

PWM 的全称为 Pulse-Width Modulation（脉冲宽度调制），实际就是调节脉冲的占空比。当输出的脉冲频率一定时，输出脉冲的占空比越大，相当于输出的有效电平越大，这样也就简单实现了由 FPGA 来控制模拟量。

LED 灯的亮暗是靠平均电压来控制的，举个例子，假设把一个时钟平均分成 10 块，如果只有 1 块的时间 LED 是导通的，剩下的 9 块 led 是截止的。假设

PWM 信号的幅值是 5V，那么在导通的 1 块时间里 LED 电压的确是 5V 的，但是在剩下 9 块时间里，它的电压却是 0V。也就是说从一个周期整体看来，LED 的平均电压只有 $5 \times 0.1 + 0 \times 0.9 = 0.5V$ ，得出计算公式为：输出电压 = (接通时间/脉冲周期) * 最大电压值。

又由于 PWM 信号频率较高（100Hz 以上），我们无法通过肉眼来观察到每一个周期 LED 灯亮灭的变化过程，所以人眼看到的亮暗即为通过平均电压这样一种方式显示。

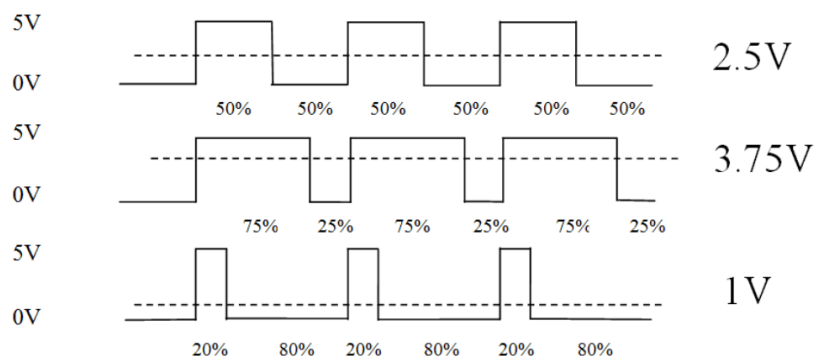


图 8-4 PWM 控制 LED 平均电压示意图

2. 实验设计要求

大家都知道 PWM 波是脉冲宽度调制，用来控制一个周期中高低电平的时间（就是我们经常说的占空比），但是具体用 Verilog 语言来实现可能还比较陌生，下面要求设计一个 PWM 的例子：LED 特效呼吸灯。

其原理是采用 pwm 的方式，在固定的频率下，采用占空比的方式来实现 LED 亮度的变化。占空比为 0，LED 灯不亮，占空比为 100%，则 LED 灯最亮。所以将占空比从 0 到 100%，再从 100%到 0 不断变化，就可以实现 LED 灯实现特效呼吸。

3. 模块划分：

分频器模块：产生 1000HZ 信号，用于 PWM 模块计数信号；

PWM 模块：实现 LED 亮度调节。

4. 实验程序

a) 顶层原理图组合模块如下图。

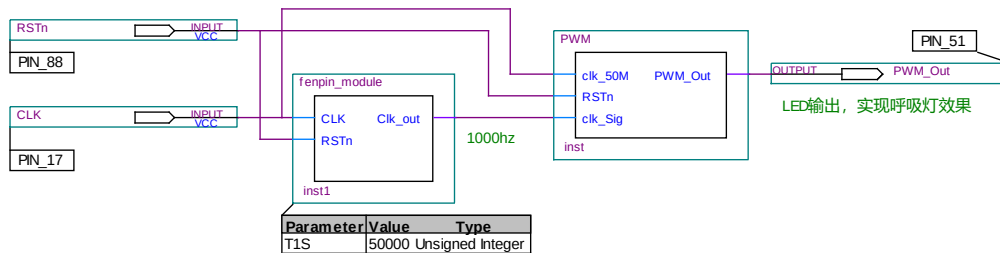


图 8-5 顶层原理图组合模块

b) 分频器模块 verilog 程序:

```
module fenpin_module(CLK, RSTn, Clk_out);
input CLK;
input RSTn;
output Clk_out;
/*****/
parameter T1S = 26'd50000;
/*****/
reg[25:0]Count1;
always @(posedge CLK or negedge RSTn)
    if(!RSTn)
        Count1 <= 26'd0;
    else if(Count1 == T1S)
        Count1 <= 26'd0;
    else
        Count1 <=Count1 +1'b1;
/*****/
reg rNewClk;
always @(posedge CLK or negedge RSTn)
    if(!RSTn)
        rNewClk <= 1'b0;
    else if(Count1 == T1S)
        rNewClk <= 1'b1;
    else
        rNewClk <= 1'b0;
/*****//
assign Clk_out =rNewClk;
endmodule
```

C) PWM 模块程序:

```
module PWM (clk_50M, RSTn, clk_Sig, PWM_Out);
input clk_50M, RSTn; // 系统输入时钟, 50M , 周期 20ns
input clk_Sig; //1000HZ信号, 1ms
output PWM_Out; //PWM输出,
reg PWM_Out; // 输出寄存器
reg[24:0] cnt; // 计数寄存器
reg [24:0] pwm_counter;
reg flag;
```

```

always@(posedge clk_50M or negedge RSTn)
begin
    if(!RSTn)
    begin
        pwm_counter = 25'd2500;
        flag=1'b0;
    end
    else if(clk_Sig) //1000HZ信号脉冲到来, 1ms
    begin
        if(flag==1'b0)//标志位灯逐渐变亮
        begin
            pwm_counter=pwm_counter+2'd2;
            if(pwm_counter>5000) flag=1'b1;//pwm_counter>5000
        end
        else if(flag==1'b1)//标志位灯逐渐变暗
        begin
            pwm_counter=pwm_counter-2'd2;
            if(pwm_counter==0) flag=1'b0;//pwm_counter=25'd0;
        end
    end
end
always@(posedge clk_50M or negedge RSTn) // 每个时钟周期的上升沿触发,
begin// 执行 begin_end 中的语句
    if(!RSTn)
    begin
        cnt = 25'd0;
        PWM_Out=1'b0;
    end
    else
    begin
        if(cnt==25'd5000) // 判断 cnt 是否为 5000, 是的话执行以下程序
        begin
            cnt = 25'd0; // 计数寄存器清零
        end
        else //cnt 没到 5000 , 执行以下程序
        begin
            cnt = cnt+1'b1;// 计数寄存器自加一
            if( pwm_counter>cnt) PWM_Out=1'b0;//PWM低电平
            else PWM_Out=1'b1; //PWM高电平
        end
    end
end

```

```
end  
endmodule
```

8.4 数码管的动态显示

1、数码管动态扫描简介

学习过单片机的朋友，多多少少都对数码管有些认识吧？关于这些基础知识，笔者就不再重复了，因为网上已经有一大堆这样的资料。

动态扫描就是轮流向各位数码管送出段码和相应的位选，只要扫描显示速度足够快，利用发光管的余辉和人眼视觉暂留作用，使人感觉好像各位数码管都在显示。动态显示的特点是将所有位数码管的段选线并联在一起，由位选线控制是哪一位数码管有效。这样一来，就没有必要每一位数码管配一个锁存器，从而大大地简化了硬件电路。

数码管不同位显示的时间间隔可以通过调整延时程序的延时长短来完成。数码管显示的时间间隔还能够确定数码管显示时的亮度，若显示的时间间隔长，则显示时数码管的亮度将亮些；若显示的时间间隔短，发光二极管的电流导通时间也就短，则显示时数码管的亮度将暗些；若显示的时间间隔过长，则数码管显示时将产生闪烁现象。所以，在调整显示的时间间隔时，既要考虑显示时数码管的亮度，又要考虑数码管显示时不闪烁的现象。动态扫描显示时刷新频率最好大于 50HZ，即显示一轮的时间不超过 20ms，一般控制在 1ms 左右最佳，本实验设计开发板的刷新时间为 1ms。

2、实验设计要求

在开发板 8 个数码管上显示 12345678，实现动态扫描程序设计。

3、硬件电路原理图

由于 LED 静态显示需要占用较多的 I/O 口，且功耗较大，因此在大多数场合通常不采用静态显示，而采用动态扫描的方法来控制 LED 数码管的显示。动态显示的特点是将 8 位数码管的段选线并联在一起，由位选线控制是哪一位数码管有效。点亮数码管采用动态扫描显示。

开发板上共有 8 个共阳数码管，采用 PNP 三极管来驱动并且控制数码管选通点亮，数码管段码共用 8 段控制信号，100 欧电阻是限流保护电阻。

数码管动态扫描实验原理图如图 8-6 所示。对于数码管动态扫描，需要动态选通 LED1—LED8 位选信号，每个位选通时根据该位需要显示的数字送入相应的段码(A-B-C-D-E-F-G-H-DP)，依次轮询扫描显示，即可显示出动态效果。

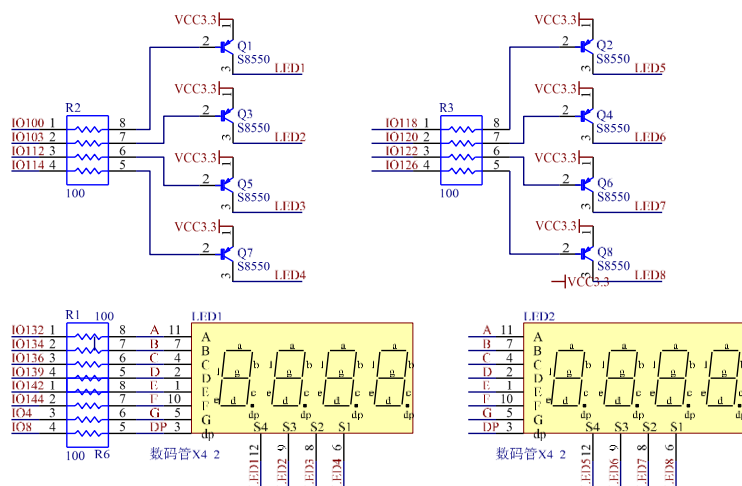


图 8-6 数码管电路原理图

4. 实验程序设计

1) 设计思路

数码管模块顶层原理图如图 8-7 所示, 输入信号有 CLK, RSTn, Number_Sig 显示的数据输入; 输出信号有位选 Scan_Sig 和段码 SMG_Data。模块对系统时钟 CLK 进行计数, 每次计数到 50000 次就是 1ms 时间, 时间到了就改变扫描位选信号和显示的数字, 数字再经过译码后转换为数码管段码从 SMG_Data 端口输出。

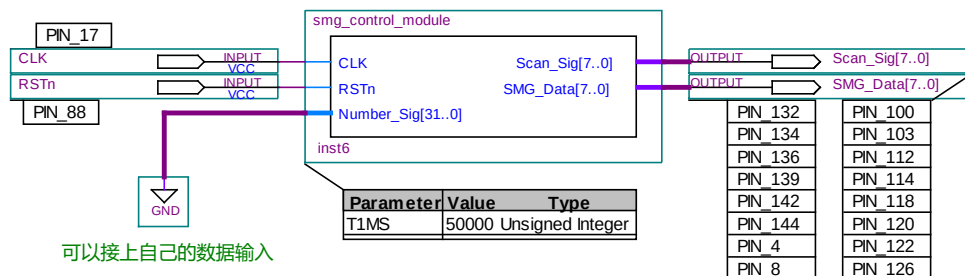


图 8-7 数码管模块顶层原理图

2) 数码管动态扫描模块 verilog 程序:

```

module smg_control_module
(
    input CLK,
    input RSTn,
    input [31:0]Number_Sig,
    output[7:0] Scan_Sig,
    output [7:0]SMG_Data
);
parameter T1MS = 16'd49999;//1ms计数值
reg [15:0]Cl;

```

```

always @(posedge CLK or negedge RSTn)
begin
    if(!RSTn)
        C1<=15'd0;
    else if(C1 == T1MS)    //1ms时间计时，时间节拍器
        C1 <=15'd0;
    else
        C1 <= C1 +1'b1;
end
reg[3:0]i;
reg[3:0]rNumber;
reg[7:0]rScan;

always @(posedge CLK or negedge RSTn)    //扫描切换显示的数据
begin
    if(!RSTn)
    begin
        i<=4'd0;
        rNumber <= 4'd0;
        rScan<= 8'b11111111;
    end
    else
    case(i)
        0:if(C1 == T1MS) i<=i+1'b1; //每个数码管显示1ms，更新位选和显示数字
        else begin rScan<= 8'b11111110; rNumber <=4'd1;end//Number_Sig[31:28];
        1:if(C1 == T1MS) i<=i+1'b1;
        else begin rScan<= 8'b11111101; rNumber <=4'd2;end//Number_Sig[27:24];
        2:if(C1 == T1MS) i<=i+1'b1;
        else begin rScan<= 8'b11111011; rNumber <=4'd3;end//Number_Sig[23:20];
        3:if(C1 == T1MS) i<=i+1'b1;
        else begin rScan<= 8'b11110111; rNumber <=4'd4;end// Number_Sig[19:16];
        4:if(C1 == T1MS) i<=i+1'b1;
        else begin rScan<= 8'b11101111; rNumber <=4'd5;end// Number_Sig[15:12];
        5:if(C1 == T1MS) i<=i+1'b1;
        else begin rScan<= 8'b11011111; rNumber <=4'd6;end//Number_Sig[11:8];
        6:if(C1 == T1MS) i<=i+1'b1;
        else begin rScan<= 8'b10111111; rNumber <=4'd7;end// Number_Sig[7:4];
        7:if(C1 == T1MS) i<=4'd0;
        else begin rScan<= 8'b01111111; rNumber <=4'd8;end// Number_Sig[3:0];
    endcase
end
end

```

```

reg[7:0]rSMG;
always @(posedge CLK or negedge RSTn)//数码管译码模块
begin
    if(!RSTn)
        begin
            rSMG <= 8'b1100_0000;//0
        end
    else
        case(rNumber)
            4'd0 : rSMG <= 8'b1100_0000;//0-9显示数字译码
            4'd1 : rSMG <= 8'b1111_1001;
            4'd2 : rSMG <= 8'b1010_0100;
            4'd3 : rSMG <= 8'b1011_0000;
            4'd4 : rSMG <= 8'b1001_1001;
            4'd5 : rSMG <= 8'b1001_0010;
            4'd6 : rSMG <= 8'b1000_0010;
            4'd7 : rSMG <= 8'b1111_1000;
            4'd8 : rSMG <= 8'b1000_0000;
            4'd9 : rSMG <= 8'b1001_0000;
            default:rSMG <= 8'b1100_0000;//
        endcase
    end
assign SMG_Data = rSMG; //段码输出
assign Scan_Sig = rScan; //位选输出
endmodule

```

5、实验现象



图 8-8 数码管显示实验现象图

8.5 秒表数码管显示

1、秒表功能简介

秒表的逻辑结构较简单，他主要由 100 进制计数器，60 进制计数器，分频器、数码管显示模块、按键控制模块等组成。在整个设计中关键是如何获得 0.01 秒（100HZ）的计时脉冲信号，另外秒表还需要一个启动信号和时间清零信号，以便秒表能随时停止、启动以及清零复位。

秒表需要 8 个数码管显示，其中 6 个显示时间数据，分别是 00-99 百分之一秒、00-60 秒、00-99 分，另外两个显示间隔符 ‘-’，显示格式是 XX-XX-XX。采用数码管动态扫描方式显示，在上一节详细讲解了显示原理和程序设计。

2、实验设计要求

在开发板上用 6 个数码管上显示 XX 分 XX 秒 XX 百分之一秒的秒表设计，能实现清零、开始计时、暂停计时功能。

3、硬件电路原理图

实验需要用到数码管电路和 FPGA 核心板上的独立按键，如图 8-8 所示。

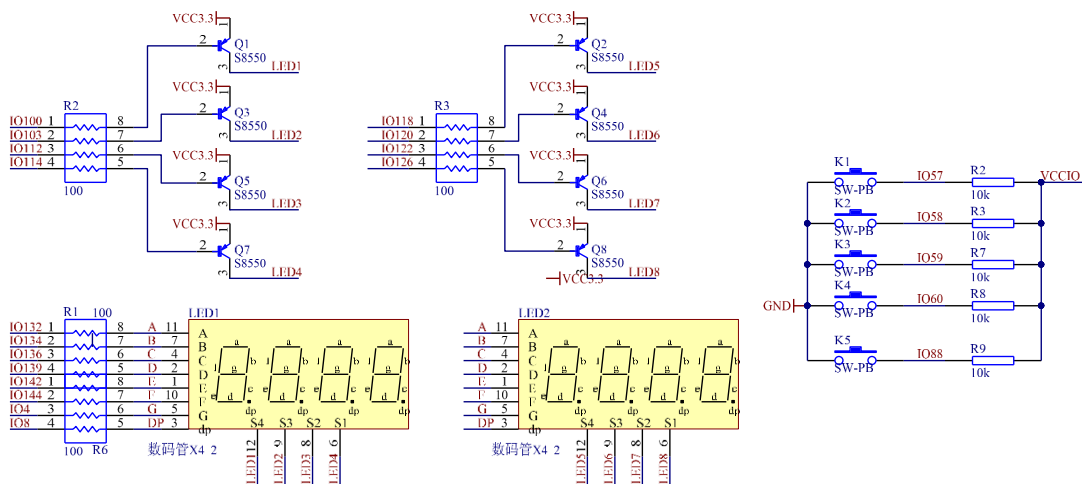


图 8-9 数码管与独立按键电路原理图

4. 实验程序设计

1) 设计思路

设计的原理框图：

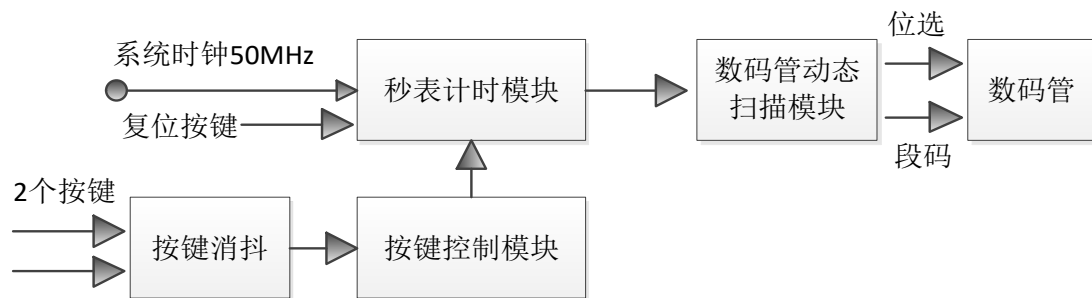


图 8-10 秒表设计原理框图

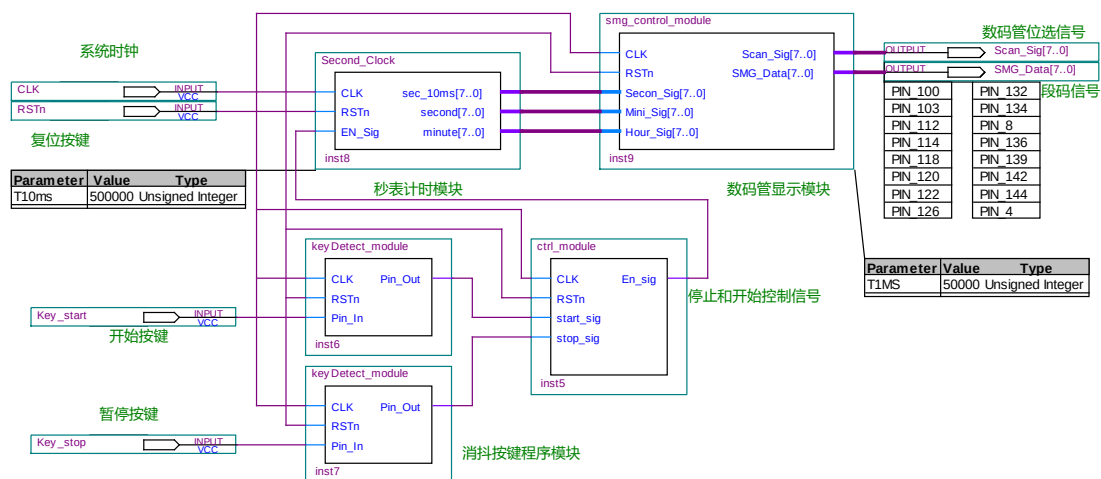


图 8-11 秒表顶层设计原理图

秒表设计顶层原理图如图 8-10 所示，输入信号有 CLK，RSTn，Key_start，Key_stop；输出信号有位选 Scan_Sig 和段码 SMG_Data。秒表计时模块对系统时钟 CLK 进行计数，每次计数到 500000 次就是 10ms 时间，内部设计计时器进行秒表计时，时间数字送到数码管模块经过译码后转换为数码管段码从 SMG_Data 端口输出显示。

2)、实验程序

按键消抖模块程序和数码管显示模块程序前面章节已经给出，下面主要给出秒表计时模块程序和按键控制模块程序。

a)秒表计时模块 verilog 程序：

```
module Second_Clock(CLK, RSTn, EN_Sig, sec_10ms, second, minute);
input CLK;
input RSTn;
input EN_Sig;
output reg [7:0]sec_10ms;
output reg [7:0]second;
output reg [7:0]minute;
reg sec_10msEn; //10ms进位使能
reg secondEn;  //秒进位使能
```

```

reg MinuEn;    //分进位使能
reg HourEn;    //小时进位使能
reg[25:0] counter;
parameter T10ms = 26'd499999;

always @(posedge CLK or negedge RSTn)
begin
    if(!RSTn)
    begin
        counter<=26'd0;
        sec_10msEn<=1'b0;
    end
    else
    begin
        counter<=counter+1'b1;
        if(counter==T10ms)
        begin
            counter<=0;
            sec_10msEn<=1'b1;//产生10ms信号
        end
        else
            sec_10msEn<=1'b0;
        end
    end
end

always @(posedge CLK or negedge RSTn)//00-99百分之一秒计时
begin
    if (!RSTn) sec_10ms=8'd0;
    else
    begin
        if(sec_10msEn && EN_Sig) //计数使能信号
        begin
            if (sec_10ms=='h99) sec_10ms=8'd0;
            else sec_10ms=sec_10ms+1'b1;
            if (sec_10ms[3:0]==4'ha)
            begin
                sec_10ms[3:0]=4'd0;
                sec_10ms[7:4]=sec_10ms[7:4]+1'b1;
            end
            if (sec_10ms=='h00) secondEn=1'b1; //秒进位使能
        end
    end
end

```

```
        else secondEn=1'b0;
    end
end

always @(posedge CLK or negedge RSTn)//60秒计时
begin
    if (!RSTn) second=8'd0;
    else
        begin
            if(secondEn && EN_Sig)//计数使能信号
                begin
                    if (second==h59) second=8'd0;
                    else second=second+1'b1;
                    if (second[3:0]==4'ha)
                        begin
                            second[3:0]=4'd0;
                            second[7:4]=second[7:4]+1'b1;
                        end
                    if (second==h00) MinuEn=1'b1; //分进位使能
                end
            else MinuEn=1'b0;
        end
    end

always @(posedge CLK or negedge RSTn)//00-99分计时
begin
    if (!RSTn) minute=8'd0;
    else
        begin
            if(MinuEn && EN_Sig)//计数使能信号
                begin
                    if (minute==h99) minute=8'd0;
                    else minute=minute+1'b1;
                    if (minute[3:0]==4'ha)
                        begin
                            minute[3:0]=4'd0;
                            minute[7:4]=minute[7:4]+1'b1;
                        end
                    if (minute==h00) HourEn=1'b1; //小时进位使能
                end
            else HourEn=1'b0;
        end
    end
```

```

        end
    end
endmodule

```

b) 按键控制模块 verilog 程序:

```

module ctrl_module
(
    input CLK,
    input RSTn,
    input start_sig,
    input stop_sig,
    output En_sig
);
    reg rEn_sig;
    reg[4:0] i;
    always @(posedge CLK or negedge RSTn)
    begin
        if(!RSTn)
        begin
            rEn_sig<=1'b0;
            i<=5'd0;
        end
        else
        case(i)
            5'd0:
                if(start_sig) i<=5'd1; //开始按键按下
                else if(stop_sig) i<=5'd2; //暂停按键按下
            5'd1:
                begin rEn_sig <=1'b1; i<=5'd0; end //使能有效
            5'd2:
                begin rEn_sig <=1'b0; i<=5'd0; end //使能无效
        endcase
    end
    assign En_sig=rEn_sig; //输出
endmodule

```

5、实验现象

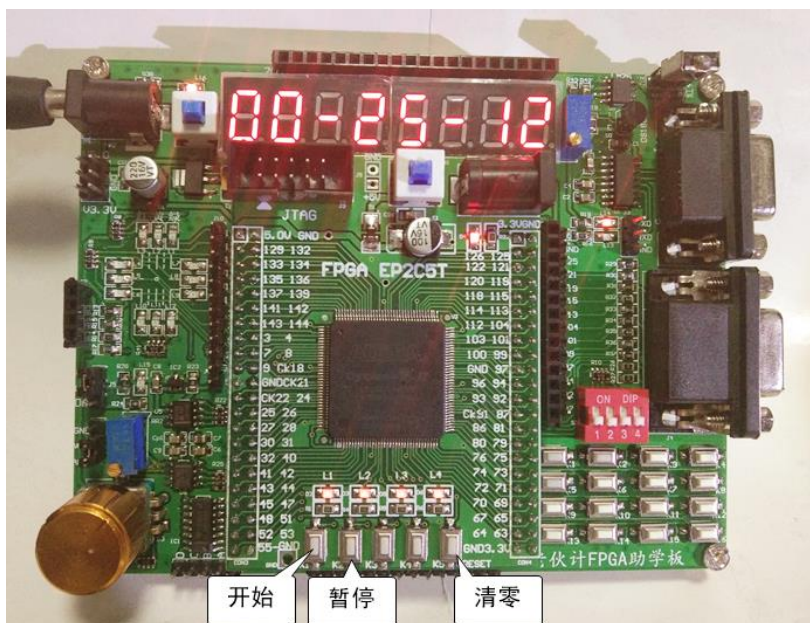


图 8-12 秒表硬件实验现象

8.6 时钟数码管显示

1、时钟功能简介

时钟的设计原理：首先通过分频器产生一个频率为 1Hz 的时钟作为计数源，设置 3 个寄存器分别用来存储计数值，时分秒各用 1 个寄存器。当秒计数达到 59 时便向分进一，同时秒清零；当分计数达到 59 时便向时进一，同时分清零；当时计数达到 24 时便将时分秒计数寄存器同时清零。

数字时钟主要由 60 进制计数器，24 进制计数器，分频器、数码管显示模块、按键消抖模块等组成。在整个设计中关键是如何获得 1 秒的计时脉冲信号，另外数字时钟还需要时间清零信号，以及三个校时按键。

数字时钟需要 8 个数码管显示，其中 6 个显示时间数据，分别是 00-59 秒、00-59 分、00-23 小时；另外两个显示间隔符 ‘-’，显示格式是 XX-XX-XX。采用数码管动态扫描方式显示，在前面章节有详细讲解了显示原理和程序设计。

2、实验设计要求

在开发板上用 6 个数码管上显示 XX 小时 XX 分 XX 秒的数字时钟设计，能实现清零、校时、校分、校秒的功能。

3、硬件电路原理图

实验需要用到数码管电路和 FPGA 核心板上的独立按键，如图 8-13 所示。

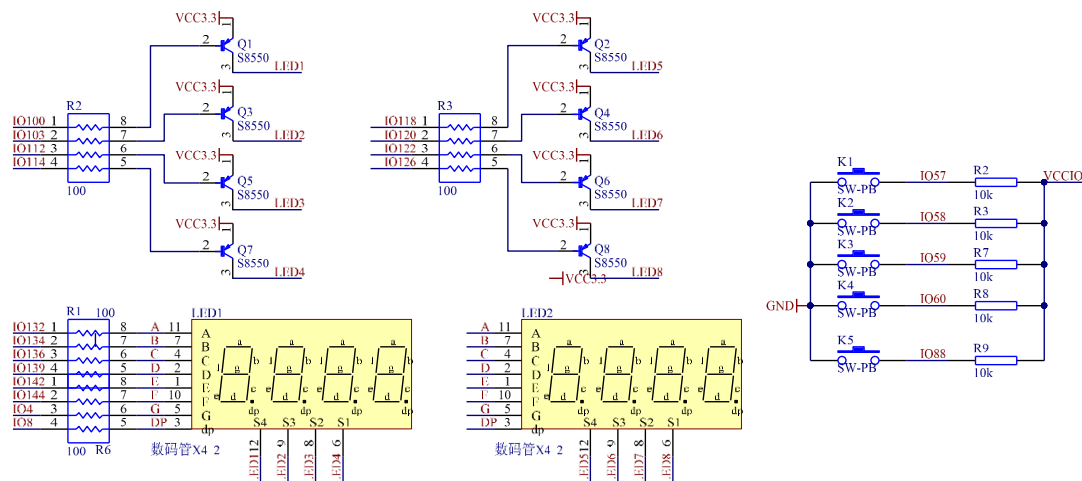


图 8-13 数码管与独立按键电路原理图

4. 实验程序设计

1) 设计思路

设计的原理框图：

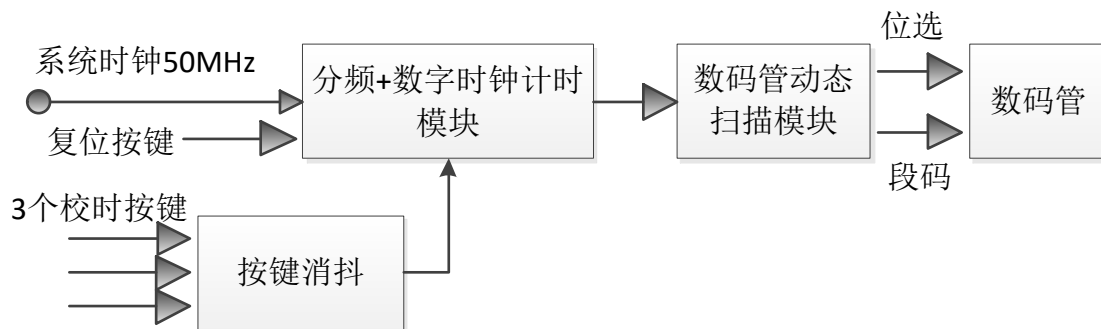


图 8-14 数字时钟设计原理框图

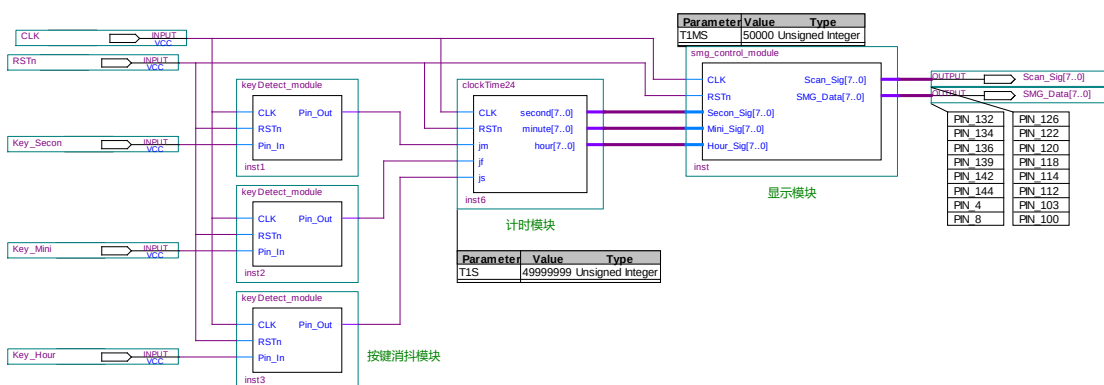


图 8-15 数字时钟顶层设计原理图

数字时钟设计顶层原理图如图 8-15 所示，输入信号有 CLK，RSTn，Key_Secon，Key_Mini，Key_Hour；输出信号有位选 Scan_Sig 和段码 SMG_Data。计时模块对系统时钟 CLK 进行计数，每次计数到 50000000 次就是 1s 时间，内

部设计计时器进行时钟计时，时间数字送到数码管模块经过译码后转换为数码管段码从 SMG_Data 端口输出显示。

2)、实验程序

按键消抖模块程序和数码管显示模块程序前面章节已经给出，下面主要给出数字时钟计时模块程序。

a) 数字时钟计时模块 verilog 程序：

```
module clockTime24(CLK, RSTn, jm, jf, js, second, minute, hour);
input CLK, RSTn;
input jm, jf, js; //校时信号
output reg [7:0]second;
output reg [7:0]minute;
output reg [7:0]hour;
reg secondEn;//秒信号使能
reg MinuEn;//分信号使能
reg HourEn;//时信号使能
reg DateEn;
reg[25:0] counter;

parameter T1S = 26'd49999999;
always @(posedge CLK or negedge RSTn)//产生秒时钟
begin
    if(!RSTn)begin
        counter<=26'd0;
        secondEn<=1'b0;
    end
    else begin
        counter<=counter+1'b1;
        if(counter==T1S)
        begin
            counter<=0;
            secondEn<=1'b1;//秒信号使能
        end
        else
            secondEn<=1'b0;
    end
end

always @(posedge CLK or negedge RSTn)//60秒计时
begin
    if (!RSTn) second=8'd0;
```

```
    else
    begin
    if(secondEn || jm) //计数使能信号
    begin
    if (second=='h59) second=8'd0;
    else second=second+1'b1;
    if (second[3:0]==4'd10)
    begin
    second[3:0]=4'd0;
    second[7:4]=second[7:4]+1'b1;
    end
    if (second=='h00) MinuEn=1'b1;
    end
    else MinuEn=1'b0;
    end
    end

always @(posedge CLK or negedge RSTn)//60分计时
begin
    if (!RSTn) minute=8'd0;
    else
    begin
    if(MinuEn || jf)//计数使能信号
    begin
    if (minute=='h59) minute=8'd0;
    else minute=minute+1'b1;
    if (minute[3:0]==4'd10)
    begin
    minute[3:0]=4'd0;
    minute[7:4]=minute[7:4]+1'b1;
    end
    if (minute=='h00) HourEn=1'b1;
    end
    else HourEn=1'b0;
    end
    end

always @(posedge CLK or negedge RSTn)//24小时计时
begin
    if (!RSTn) hour=8'd0;
    else
```

```
begin
if(HourEn || js)//计数使能信号
begin
if (hour=='h23') hour=8'd0;
else hour=hour+1'b1;
if (hour[3:0]==4'd10)
begin
hour[3:0]=4'd0;
hour[7:4]=hour[7:4]+1'b1;
end
if (hour=='h00) DateEn=1'b1;
end
else DateEn=1'b0;
end
end
endmodule
```

5、实验现象



图 8-16 数字时钟硬件实验现象

8.7 频率计的设计

1、频率计原理简介

所谓“频率”，就是周期性信号在单位时间(秒)内变化的次数。若在一定的时间间隔 T 内计数，计得某周期性信号的重复变化次数为 N ，则该信号的频率可表达为：

$$f = N / T$$

所以测量频率就要分别知道 N 和 T 的值，由此，测量频率的方法一般有三种：测频方法、测周方法和等精度测量。

1) 测频方法：这种方法即已知时基信号（频率或周期确定）做门控信号， T 为已知量，然后在门控信号有效的时间段内进行输入脉冲的计数，原理波形图如下图所示：

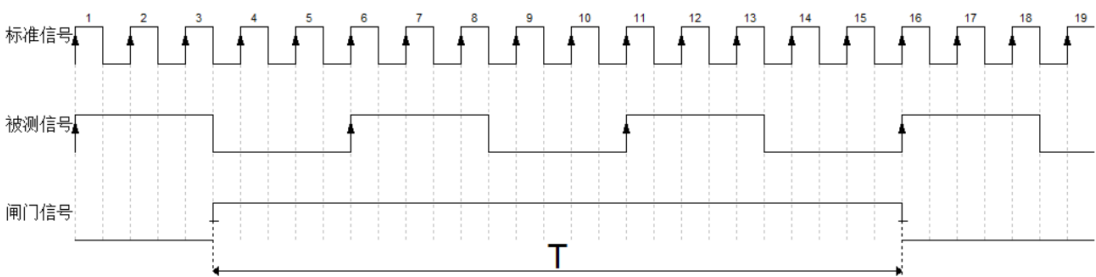


图 8-17 测频原理波形图

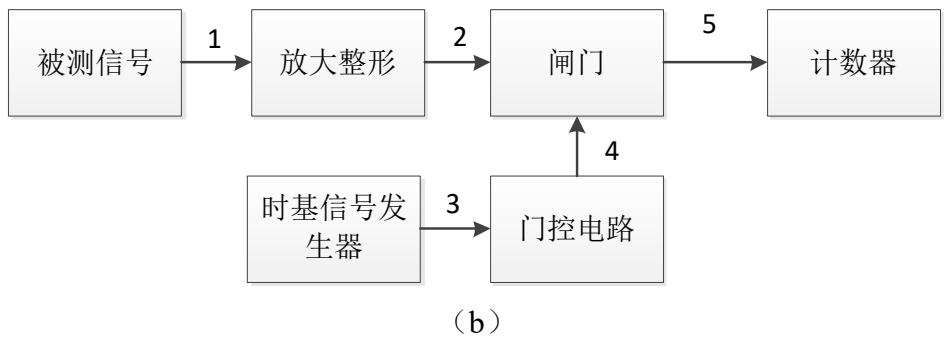
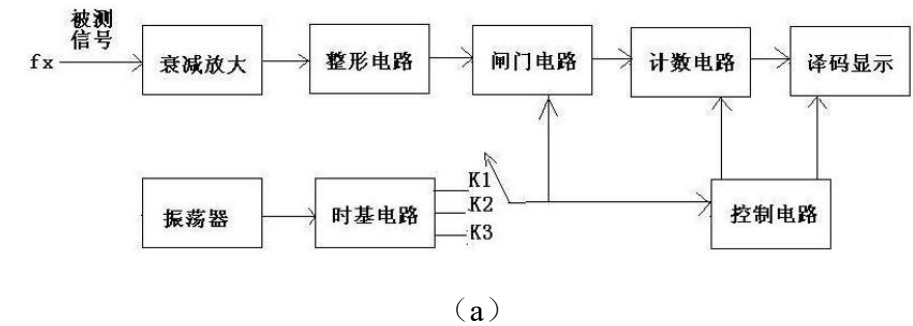


图 8-18 测频原理框图

测频方法原理框图说明：

首先，被测信号①(以正弦波为例)经过放大整形后转变成方波脉冲②，其重复

频率等于被测信号频率。把方波脉冲②加到闸门的输入端。由一个高稳定的石英振荡器和一系列数字分频器组成了时基信号发生器，它输出时间基准(或频率基准)信号③去控制门控电路形成门控信号④，门控信号的作用时间 T 是非常准确的(由石英振荡器决定)。门控信号控制闸门的开与闭，只有在闸门开通的时间内，方波脉冲②才能通过闸门成为被计数的脉冲⑤由计数器计数。闸门开通的时间称为闸门时间，其长度等于门控信号作用时间 T 。比如，时间基准信号的重复周期为 1S ，加到闸门的门控信号作用时间 T 亦准确地等于 1S ，即闸门的开通时间——“闸门时间”为 1S 。在这一段时间内，若计数器计得 $N=100000$ 个数，根据公式 $f = N / T$ ，那么被测频率就是 100000Hz 。如果计数式频率计的显示器单位为“ KHz ”，则显示 100.000KHz ，即小数点定位在第三位。不难设想，若将闸门时间设为 $T=0.1\text{S}$ ，则计数值为 10000 ，这时，显示器的小数点只要根据闸门时间 T 的改变也随之自动往右移动一位(自动定位)，那么，显示的结果为 100.00KHz 。在计数式数字频率计中，通过选择不同的闸门时间，可以改变频率计的测量范围和测量精度。

2) 测周方法:

被测信号(频率或周期待测)做门控信号， T 为未知量，做门控信号 T ，然后在门控信号有效的时间段内对时基信号脉冲计数，原理图如下图所示：

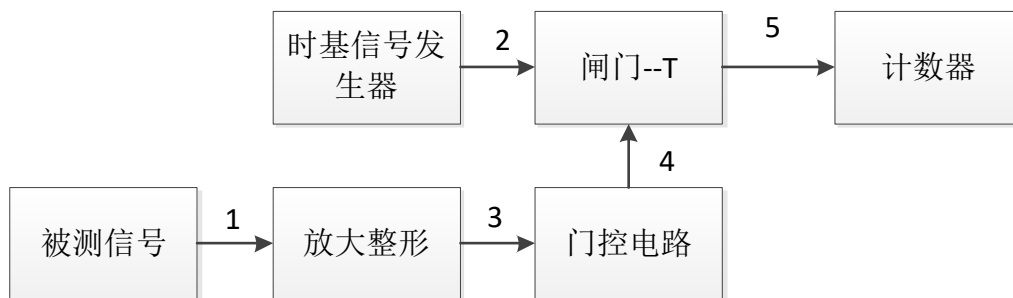


图 8-18 测周发测频原理框图

计数器测周的基本原理刚好与测频相反，即由被测信号控制主门开门，而用时钟脉冲进行计数，所以实质上也是一种比较测量方法。

3) 等精度测量法:

等精度测量法的核心思想是通过闸门信号与被测信号同步，将闸门时间 τ 控制为被测信号周期长度的整数倍。测量时，先打开预置闸门，当检测到被测信号脉冲沿到达时，标准信号时钟开始计数。预置闸门关闭时，标准信号并不立即停止计数，而是等检测到被测信号脉冲沿到达时才停止，完成被测信号整数个周期的测量。测量的实际闸门时间可能会与预置闸门时间不完全相同，但最大差值不会超过被测信号的一个周期。

在等精度测量法中，相对误差与被测信号本身的频率特性无关，即对整个测量

域而言，测量精度相等，因而称之为“等精度测量”。标准信号的计数值越大则测量相对误差越小，即提高门限时间 τ 和标准信号频率 f_c 可以提高测量精度。在精度不变的情况下，提高标准信号频率可以缩短门限时间，提高测量速度。

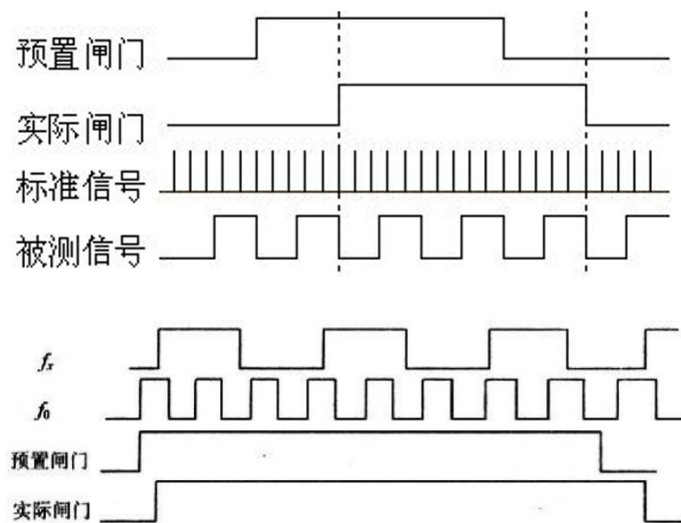


图 8-19 等精度测量法波形图

2、实验设计要求

采用测频法测频信号频率值，通过数码管显示出来。信号从信号发生器产生，幅度 3V，频率 1Hz-10MHz，信号从 IO80 引脚接入，调试程序，观察结果。

3、电路原理图

实验设计要求实验需要用到数码管电路和 FPGA 引出的 IO 口，如图 9-20 所示。

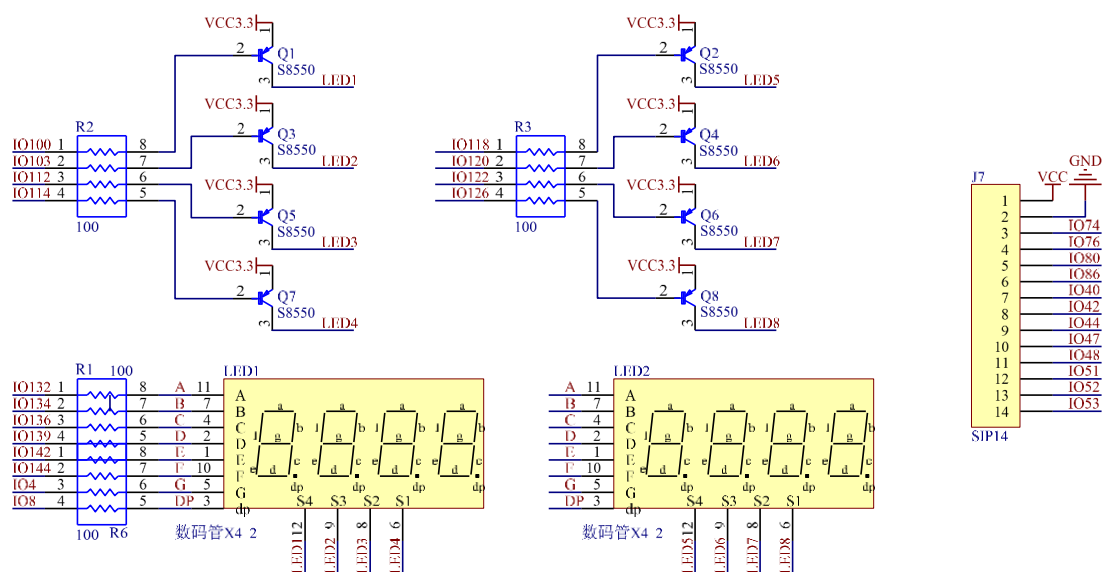


图 8-20 数码管与独立按键电路原理图

4. 实验程序设计

1) 设计思路

设计的原理框图:

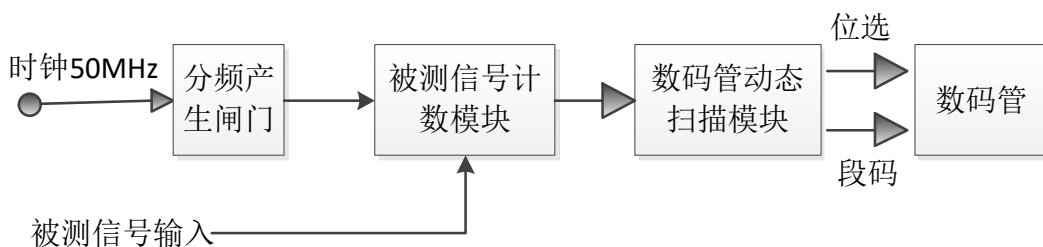


图 8-21 测频法程序设计框图

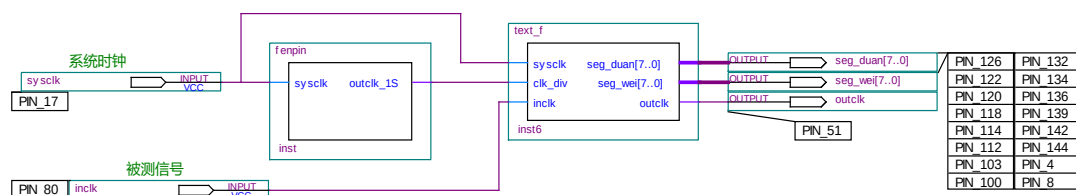


图 8-22 频率计顶层设计原理图

频率计设计顶层原理图如图 8-22 所示，输入信号有 sysclk, inclk；输出信号有位选 seg_wei 和段码 seg_duan。分频模块对系统时钟 sysclk 进行计数，每次计数到 50000000 次就是 1s 时间，产生 1 秒高电平计数闸门，1 秒低电平输出处理，内部设计计数器对被测信号进行计数，计数数字送到数码管模块经过译码后转换为数码管段码从 seg_duan 端口输出显示。

2)、实验程序

a) 分频闸门模块 verilog 程序:

```

module fenpin(sysclk,outclk_1S);
input  sysclk;//系统时钟
output outclk_1S;
reg [25:0] clk_counter;//时钟计数
reg clk_div;//分频后的时钟
/*****产生1HZ的时钟*****/
always @(posedge sysclk)
begin
    if(clk_counter==26'd50000000)
    begin
        clk_div<=~clk_div;
        clk_counter<=26'd0;
    end
    else
        clk_counter<=clk_counter+1'b1;
    end
end

```

```
assign outclk_1S= clk_div;
endmodule
```

b) 被测信号计数模块 verilog 程序:

```
module text_f(sysclk, clk_div, inclk, seg_duan, seg_wei, outclk);
input    sysclk, inclk; //系统时钟
input    clk_div;
output [7:0] seg_duan, seg_wei; //输入时钟
output outclk;
wire     sysclk, inclk; //系统时钟
reg [7:0] rseg_duan, rseg_wei; //输入时钟
reg [25:0] clk_counter; //时钟计数
reg [7:0] data;
reg [3:0]
counter1, counter2, counter3, counter4, counter5, counter6, counter7, counter8;
reg [3:0] count1, count2, count3, count4, count5, count6, count7, count8;
//*****测试待测信号*****/
always @(posedge inclk)
begin
    if(clk_div)
    begin
        if(counter1==4'b1001)
        begin
            counter1<=4'b0;
            counter2<=counter2+1'b1;
            if(counter2==4'b1001)
            begin
                counter2<=4'b0;
                counter3<=counter3+1'b1;
                if(counter3==4'b1001)
                begin
                    counter3<=4'b0;
                    counter4<=counter4+1'b1;
                    if(counter4==4'b1001)
                    begin
                        counter4<=4'b0;
                        counter5<=counter5+1'b1;
                        if(counter5==4'b1001)
                        begin
                            counter5<=4'b0;
                            counter6<=counter6+1'b1;
                            if(counter6==4'b1001)
```

```

        begin
            counter6<=4'b0;
            counter7<=counter7+1'b1;
            if(counter7==4'b1001)
                begin
                    counter7<=4'b0;
                    counter8<=counter8+1'b1;
                    if(counter8==4'b1001)
                        begin
                            counter8<=4'b0;
                        end
                    end
                end
            end
        end
    end
end
end
else
    counter1<=counter1+1'b1;
end
else
    /*******测试结果寄存*****/
    begin
        if(counter1!=4'b0000|counter2!=4'b0000!=4'b0000|counter3!=4'b0000|counter4
        !=4'b0000|counter5!=4'b0000|counter6!=4'b0000|counter7!=4'b0000|counter8!=
        4'b0000)
            begin
                count1<=counter1;count2<=counter2;count3<=counter3;count4<=counter4;
                count5<=counter5;count6<=counter6;count7<=counter7;count8<=counter8;
                counter1<=4'b0000; counter2<=4'b0000; counter3<=4'b0000;
                counter4<=4'b0000; counter5<=4'b0000; counter6<=4'b0000;
                counter7<=4'b0000; counter8<=4'b0000;
            end
        end
    end
    /*******测试结果数码管显示*****/
    always @(clk_counter , count1 , count2 , count3 , count4 , count5 , count6 ,
    count7 , count8,data)
    begin
        case(clk_counter[15:13])//数码管位扫描

```

```

3'b000:begin rseg_wei<=8'b1111_1110;data<=count1;end
3'b001:begin rseg_wei<=8'b1111_1101;data<=count2;end
3'b010:begin rseg_wei<=8'b1111_1011;data<=count3;end
3'b011:begin rseg_wei<=8'b1111_0111;data<=count4;end
3'b100:begin rseg_wei<=8'b1110_1111;data<=count5;end
3'b101:begin rseg_wei<=8'b1101_1111;data<=count6;end
3'b110:begin rseg_wei<=8'b1011_1111;data<=count7;end
3'b111:begin rseg_wei<=8'b0111_1111;data<=count8;end
default:begin rseg_wei<=8'bx;data<=4'bx;end
endcase
case(data[3:0])//数码管显示
4'b0000:begin rseg_duan<=8'b1100_0000;end//0
4'b0001:begin rseg_duan<=8'b1111_1001;end//1
4'b0010:begin rseg_duan<=8'b1010_0100;end//2
4'b0011:begin rseg_duan<=8'b1011_0000;end//3
4'b0100:begin rseg_duan<=8'b1001_1001;end//4
4'b0101:begin rseg_duan<=8'b1001_0010;end//5
4'b0110:begin rseg_duan<=8'b1000_0010;end//6
4'b0111:begin rseg_duan<=8'b1111_1000;end//7
4'b1000:begin rseg_duan<=8'b1000_0000;end//8
4'b1001:begin rseg_duan<=8'b1001_0000;end//9
default:rseg_duan<=8'bx;
endcase
end
assign seg_duan=rseg_duan;
assign seg_wei=rseg_wei;
endmodule

```

5、实验现象

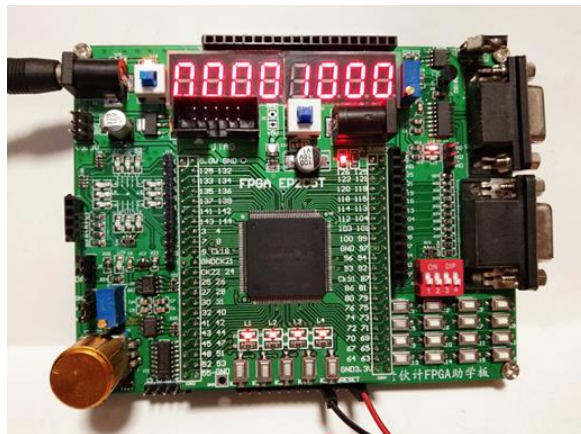


图 8-23 频率计硬件实验现象

8.8 蜂鸣器音乐播放器

1、蜂鸣器原理简介

蜂鸣器结构原理：

1) 压电式蜂鸣器：压电式蜂鸣器主要由多谐振荡器、压电蜂鸣片、阻抗匹配器及共鸣箱、外壳等组成。有的压电式蜂鸣器外壳上还装有发光二极管。

多谐振荡器由晶体管或集成电路构成。当接通电源后（1.5~15V 直流工作电压），多谐振荡器起振，输出 1.5~2.5kHz 的音频信号，阻抗匹配器推动压电蜂鸣片发声。

压电蜂鸣片由锆钛酸铅或铌镁酸铅压电陶瓷材料制成。在陶瓷片的两面镀上银电极，经极化和老化处理后，再与黄铜片或不锈钢片粘在一起。

2) 电磁式蜂鸣器：电磁式蜂鸣器由振荡器、电磁线圈、磁铁、振动膜片及外壳等组成。接通电源后，振荡器产生的音频信号电流通过电磁线圈，使电磁线圈产生磁场。振动膜片在电磁线圈和磁铁的相互作用下，周期性地振动发声。



图 8-24 无源蜂鸣器

有源蜂鸣器和无源蜂鸣器，教你区分有源蜂鸣器和无源蜂鸣器，现在市场上出售的一种小型蜂鸣器因其体积小(直径只有 11mm)、重量轻、价格低、结构牢靠，而广泛地应用在各种需要发声的电器设备、电子制作和单片机等电路中。

从外观上看，两种蜂鸣器好像一样，但仔细看，两者的高度略有区别，

有源蜂鸣器高度为 9mm，而无源蜂鸣器的高度为 8mm。如将两种蜂鸣器的引脚都朝上放置时，可以看出有绿色电路板的一种是无源蜂鸣器，没有电路板而用黑胶封闭的一种是有源蜂鸣器。

进一步判断有源蜂鸣器和无源蜂鸣器，还可以用万用表电阻档 R_{x1} 档测试：用黑表笔接蜂鸣器 "+" 引脚，红表笔在另一引脚上来回碰触，如果触发出咔、咔声的且电阻只有 8Ω(或 16Ω)的是无源蜂鸣器；如果能发出持续声音的，且电阻在几百欧以上的，是有源蜂鸣器。

有源蜂鸣器直接接上额定电源(新的蜂鸣器在标签上都有注明)就可连续发声；而无源蜂鸣器则和电磁扬声器一样，需要接在音频输出电路中才能发声。

2、乐曲演奏实验设计原理

乐曲演奏的原理是：由于组成乐曲的每个音符的频率值（音调）及其持续时间（音长）是乐曲演奏的基本数据，因此需要控制输出到扬声器的激励信号的频率高低和该频率持续的时间。频率的高低决定了音调的高低，而乐曲的简谱与各音名的频率对应关系如下表所示。

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	1045.5
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

图 8-25 音符对应的频率值

所有不同频率的信号都是从同一基准频率分频而来的。由于音节频率多为非整数，而分频系数又不能为小数，故必须计算得到的分频系数进行四舍五入取整，并且其基准频率和分频系数应综合加以选择，从而保证音乐不会走调。开发板的晶振为 50MHZ，故在 50M HZ 时钟下，中音 1（对应的频率值为 523.3Hz）的分频系数为 $50000000/(2*523.3)=47774$ ，这样只需对系统时钟进行 47774 次分频即可得到所要的中音 1。可利用同样方法求出其他音符对应的分频系数，这样利用程序可以很轻松得到相应的乐声。

根据蜂鸣器的发声原理，程序中设置了一个状态机，同时产生一个时钟作为音长的控制，时钟周期为 250ms，每 250ms 改变一个状态（即一个节拍），组成乐曲的每个频率值（音调）相对于状态机的每一个状态。只要让状态机的状态按顺序转换，就可以自动播放音乐了！

3、硬件电路原理图

通过 FPGA 管脚 IO93 输出音符频率，蜂鸣器由 PNP 三极管 8550 驱动，FPGA 管脚驱动三极管的基极控制其导通或者截止，从而控制蜂鸣器发出声音。

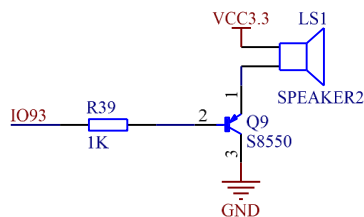


图 8-26 蜂鸣器原理图

4. 实验程序

//音高与频率的对应关系

```
//-----
//|      | 1      | 2      | 3      | 4      | 5      | 6      | 7      |
//|低音 | 261.6Hz | 293.7Hz | 329.6Hz | 349.2Hz | 392Hz  | 440Hz  | 493.9Hz |
//|中音 | 523.3Hz | 587.3Hz | 659.3Hz | 698.5Hz | 784Hz  | 880Hz  | 987.8Hz |
//|高音 | 1045.5Hz| 1174.7Hz| 1318.5Hz| 1396.9Hz| 1568Hz | 1760Hz | 1975.5Hz|
//-----
```

```
module song(clk, beep);           //模块名称song
input  clk;                       //系统时钟50MHz
output beep;                     //蜂鸣器输出端
reg    beep_r;                   //寄存器
reg[7:0] state;                  //乐谱状态机
reg[15:0] count, count_end;
reg[23:0] count1;
//乐谱参数:D=F/2K (D: 参数, F: 时钟频率, K: 音高频率)
parameter L_3 = 16'd75850,       //低音3
           L_5 = 16'd63776,       //低音5
           L_6 = 16'd56818,       //低音6
           L_7 = 16'd50618,       //低音7
           M_1 = 16'd47774,       //中音1
           M_2 = 16'd42568,       //中音2
           M_3 = 16'd37919,       //中音3
           M_5 = 16'd31888,       //中音5
           M_6 = 16'd28409,       //中音6
           H_1 = 16'd23912;       //高音1
parameter TIME = 6000000;        //控制每一个音的长短(250ms)

assign beep = beep_r;            //输出音乐

always@(posedge clk)
begin
    count <= count + 1'b1;       //计数器加1
```

```

if(count == count_end/2)           //24MHZ
begin
    count <= 16'h0;                 //计数器清零
    beep_r <= !beep_r;              //输出取反
end
end

always @(posedge clk)
begin
    if(count1 < TIME)                //一个节拍250mS
        count1 = count1 + 1'b1;
    else
    begin
        count1 = 24'd0;
        if(state == 8'd66)
            state = 8'd0;
        else
            state = state + 1'b1;
        case(state)
            8'd0, 8'd1, 8'd2, 8'd3:    count_end = L_3; //低音"3", 持续4个节拍
            8'd4, 8'd5, 8'd6:        count_end = L_5; //低音"5", 持续3个节拍
            8'd7:                    count_end = L_6; //低音"6", 持续1个节拍
            8'd8, 8'd9, 8'd10:       count_end = M_1; //中音"1", 持续3个节拍
            8'd11:                   count_end = M_2; //中音"2", 持续1个节拍
            8'd12:                   count_end = L_6; //低音"6", 持续1个节拍
            8'd13:                   count_end = M_1; //中音"1", 持续1个节拍
            8'd14, 8'd15:            count_end = L_5; //低音"5", 持续2个节拍
            8'd16:                   count_end = M_1; //中音"1", 持续1个节拍
            8'd17, 8'd18:            count_end = L_5; //低音"5", 持续2个节拍
            8'd19, 8'd20, 8'd21:     count_end = M_5; //中音"5", 持续3个节拍
            8'd22:                   count_end = H_1; //高音"1", 持续1个节拍
            8'd23:                   count_end = M_6; //中音"6", 持续1个节拍
            8'd24:                   count_end = M_5; //中音"5", 持续1个节拍
            8'd25:                   count_end = M_3; //中音"3", 持续1个节拍
            8'd26:                   count_end = M_5; //中音"5", 持续1个节拍
            8'd27, 8'd28, 8'd29, 8'd30, 8'd31, //中音"2", 持续11个节拍
            8'd32, 8'd33, 8'd34, 8'd35, 8'd36, 8'd37: count_end = M_2;
            8'd38:                   count_end = M_3; //中音"3", 持续1个节拍
            8'd39, 8'd40:            count_end = L_7; //低音"7", 持续2个节拍
            8'd41, 8'd42:            count_end = L_6; //低音"6", 持续2个节拍
            8'd43, 8'd44, 8'd45:     count_end = L_5; //低音"5", 持续3个节拍

```



```

8' d46:                count_end = L_6; //低音"6", 持续1个节拍
8' d47, 8' d48:        count_end = M_1; //中音"1", 持续2个节拍
8' d49, 8' d50:        count_end = M_2; //中音"2", 持续2个节拍
8' d51, 8' d52:        count_end = L_3; //低音"3", 持续2个节拍
8' d53, 8' d54, 8' d55: count_end = M_1; //中音"1", 持续3个节拍
8' d56, 8' d57:        count_end = L_5; //低音"5", 持续2个节拍
8' d58:                count_end = M_1; //中音"1", 持续1个节拍
8' d59, 8' d60, 8' d61, 8' d62, 8' d63,
8' d64, 8' d65, 8' d66: count_end = L_5; //低音"5", 持续8个节拍
endcase
end
end
endmodule

```

8.9 按键计数器

1、按键计数器原理简介

按键计数器顾名思义,就是对按键按下进行次数,然后将计数的结果显示在数码管上。因此,首先必须对按键进行消抖,由于 FPGA 开发板的时钟为 50MHz,并且 FPGA 为硬件设计,所以对毛刺十分敏感,在该工程中采用识别按键按下后输出脉冲信号和延时方法对按键进行消抖,消抖后对按键进行计数,最后将计数结构显示在数码管上。

按键消抖和数码管显示相关内容请复习前面相关章节。

2、实验设计要求:

采用 FPGA 核心板上的一个按键,每按一下按键计数器加一,在数码管上显示,计数值是 00-99。

3、硬件设计原理图:

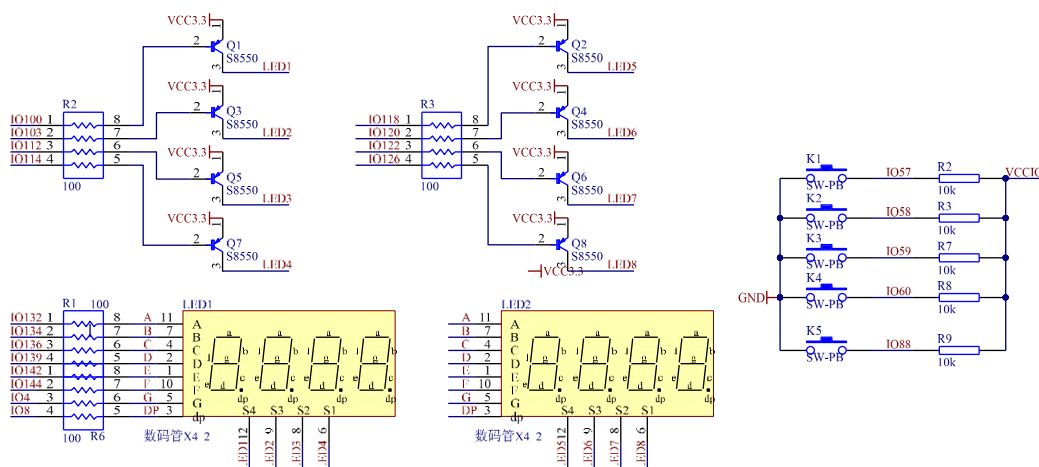


图 8-27 数码管与独立按键电路原理图

4. 实验程序设计

1) 设计思路

设计的原理框图:

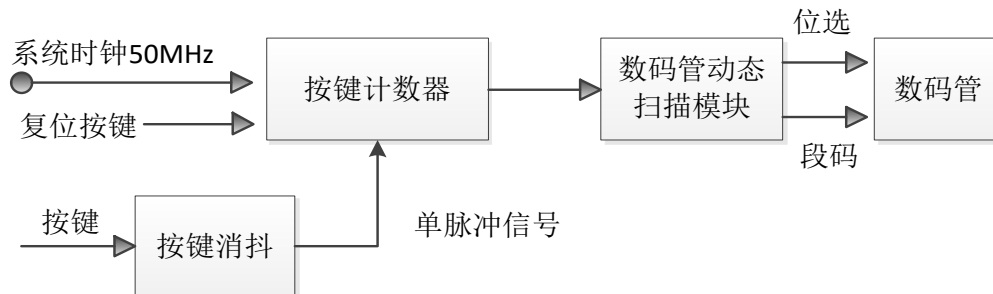


图 8-28 按键计数器设计原理框图

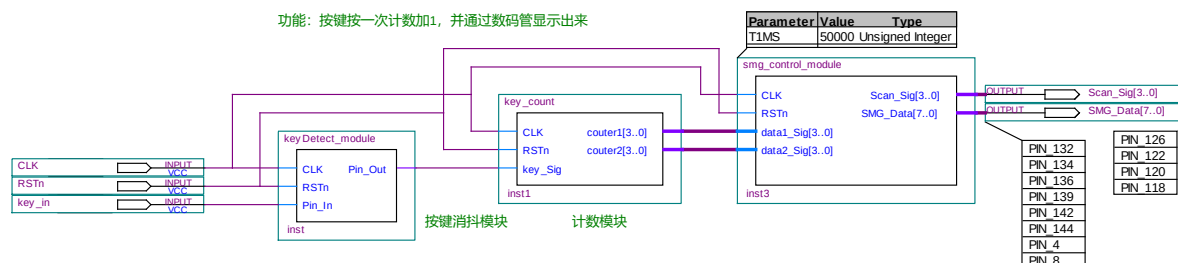


图 8-29 计数器顶层设计原理图

频率计设计顶层原理图如图 8-29 所示，输入信号有 CLK，RSTn，key_in；输出信号有位选 Scan_Sig 和段码 SMG_Data。按键经过消抖处理后输出单脉冲信号进入计数模块，计数模块对按键脉冲信号进行计数，计数数字送到数码管模块经过译码后转换为数码管段码从 SMG_Data 端口输出显示。

2)、实验程序

按键消抖和数码管显示程序请复习前面相关章节，下面给出的是计数模块的程序。

```
module key_count(CLK,RSTn,key_Sig,couter1,couter2);
input CLK,RSTn;
input key_Sig;
output reg[3:0] couter1;
output reg[3:0] couter2;

always@(posedge CLK or negedge RSTn)
begin
    if(!RSTn)begin couter1<=4'd0; couter2<=4'd0;end
    else if(key_Sig)
    begin
```

```
couter1<=couter1+1'b1;  
if(couter1==9)  
begin  
    couter1<=4'd0;  
    couter2<=couter2+1'b1;  
    if(couter2==9)  
        couter2<=4'd0;  
    end  
end  
end  
endmodule
```

5. 实验现象

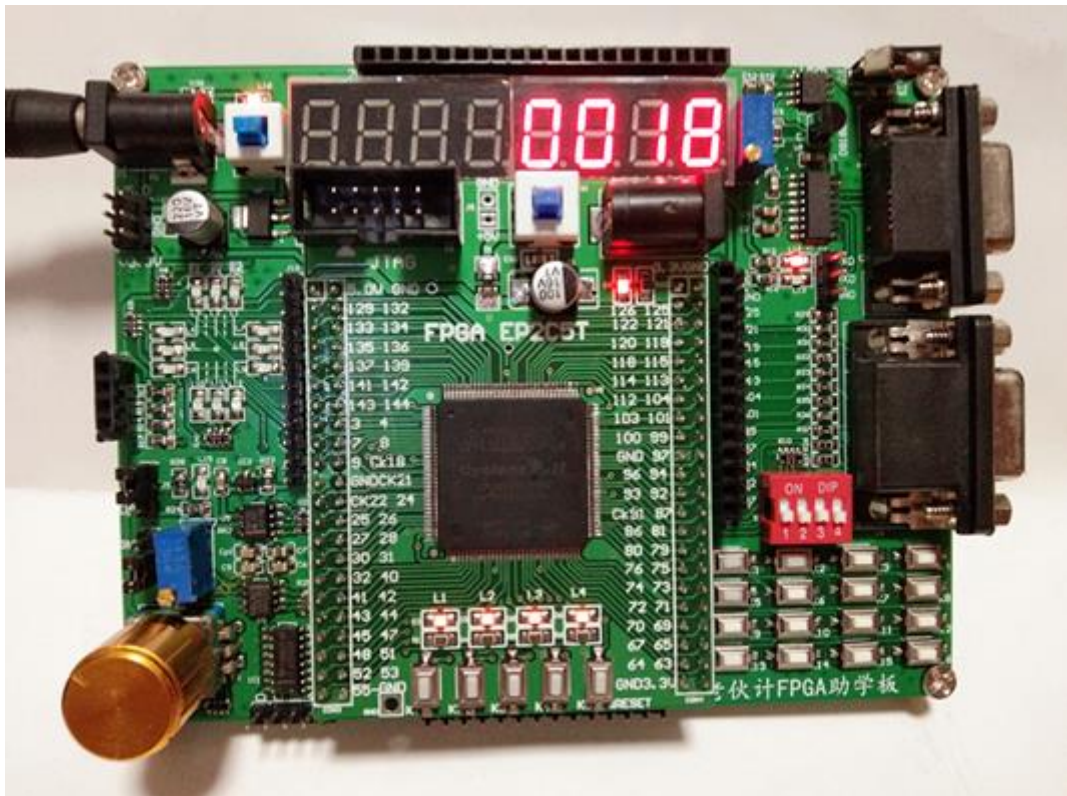


图 8-30 按键计数器实验图

8.10 串口通信

1、串口通信原理说明

1) 硬件结构: FPGA 和上位机电脑之间的通信采用 UART 串口通信方式, 首先介绍下九针串口硬件的结构:

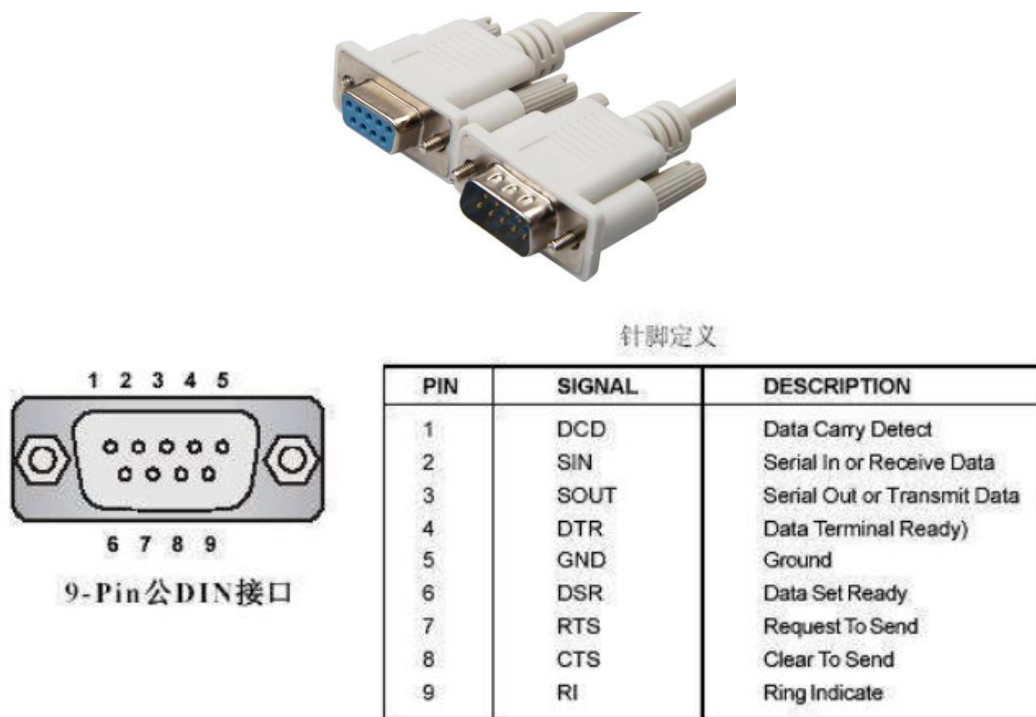


图 8-31 串口实物及引脚图

它一共有 9 个引脚, 但是最重要的 3 个引脚是:

引脚 2: RxD (接收数据);

引脚 3: TxD (发送数据);

引脚 5: GND (地);

仅使用 3 跟电缆, 你就可以发送和接收数据。

2) 串口的通信协议:

首先要说波特率的概念: 含义是每秒传送的二进制位数。通信双方的波特率一定要相等, 比如常用的波特率有 4800bps, 9600bps, 19200bps, 115200bps 等。

波特率(Baud rate), 指的是信号被调制以后在单位时间内的变化, 即单位时间内载波参数变化的次数, 如每秒钟传送 240 个字符, 而每个字符格式包含 10 位 (1 个起始位, 1 个停止位, 8 个数据位), 这时的波特率为 240Bd, 比特率为 10 位*240 个/秒=2400bps。

数据格式: 1 位起始位+8 位数据位+1 位校验位(可选)+1 位停止位, RS-232 是

使用异步通讯协议。也就是说数据的传输没有时钟信号。接收端必须有某种方式，使之与接收数据同步。

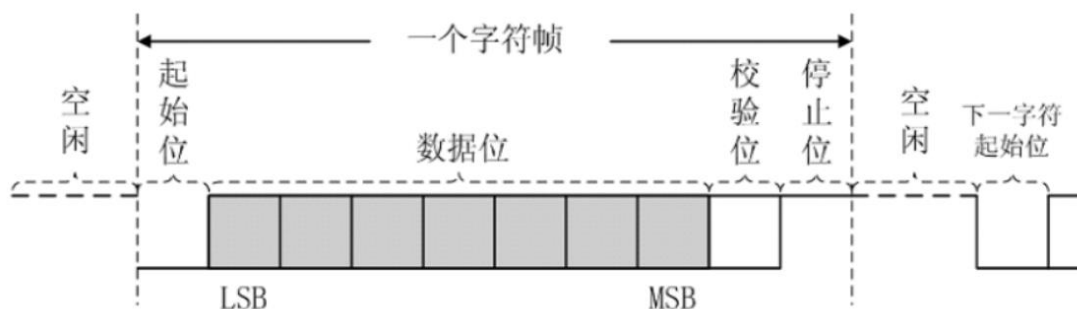


图 8-31 串口数据帧格式

在串口的总线上“高电平”是默认的状态，当一帧数据的开始传输必须先拉低电平，这就是第 0 位的作用。第 0 位过后就是 8 个数据位，这八个数据位才是一帧数据中最有意义的东西。最后两位是校验位和停止位，作用如同命名般一样，基本上是没有重要意义。

3) 数据发送处理过程：

串行线缆的两端事先约定好串行传输的参数（传输速度（既是波特率）、传输格式等），当没有数据传输的时候，发送端向数据线上发送“1”，每传输一个字节之前，发送端先发送一个“0”来表示传输已经开始。这样接收端便可以知道有数据到来了。开始传输后，数据以约定的速度和格式传输，所以接收端可以与之同步每次传输完成一个字节之后，都在其后发送一个停止位（“1”）。

让我们来看看 0x55 是如何传输的：

0x55 的二进制表示为：01010101。

但是由于先发送的是最低有效位，所以发送序列是这样的：

1-0-1-0-1-0-1-0。

2 、实验设计功能

实现串口通信_FPGA 和上位机通信，波特率：9600bps,10 个 bit 是 1 位起始位，8 个数据位，1 个结束位。

按动 key1，FPGA 向 PC 发送“UART_T”一次，用串口调试助手显示字符串（串口调试工具设成字符格式接受和发送方式），PC 发送 0-9 数字字符到 FPGA，FPGA 接受（0 到 9）后显示在数码管上显示，实现串口收发数据功能。

3、实验的原理图：

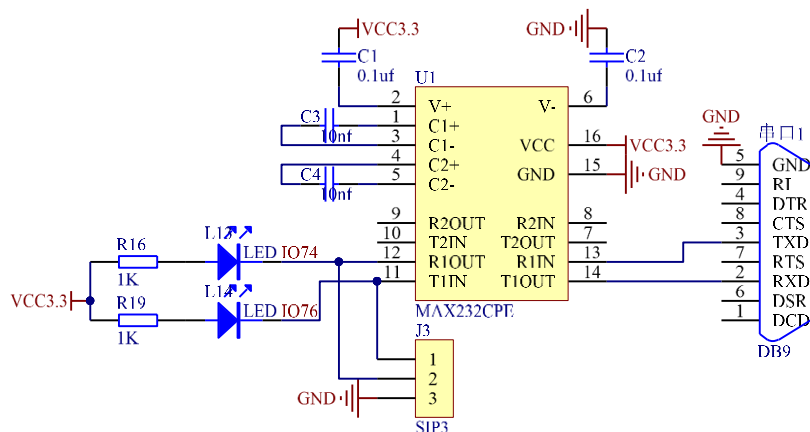


图 8-32 串口设计原理图

图中 MAX232 是电平转换芯片，因为 RS232 电平：采用 -12V 到 +3V，等价于逻辑 "0"，+3V 到 +12V 的逻辑电平，等价于逻辑 "1"，但是 FPGA 的电平属于 TTL 电平，因此通信双方不能直接通信，需经过电平转换。

4、实验程序设计

1) 设计思路

实验设计原理（设计需要包括发送模块，接收模块，波特率发生模块）。

波特率产生模块：原理就是对时钟分频得到波特率时钟，本程序是对 50MHz 时钟进行 651 分频可近似得到 9600bps*8 倍的时钟来控制发送和接收模块， $651 \times 8 = 5208$ 个 MCLK。

串口发送/接收的比特率的时钟计算：串口的比特率为 9600bps 时， $dt = 1s/9600 = 104.1667\mu s$ 。即每 bit 数据发送的时间间隔为 $dt = 104.1667\mu s$ 。当主系统时钟 MCLK = 50MHz，其周期 $T = 0.02\mu s$ 。则要计数 $dt/T = 104.1667/0.02 = 5208$ 个 MCLK 才发送一位 (bit) 数据。则在 RS232 (串口) 的时钟周期 $rsT = dt$ ，要计数 5208 个 MCLK。当时钟为方波时，只要计数 $5208/2 = 2604$ 个 MCLK，就翻转 IO 口产生需要的时钟信号。

发送模块：在波特率时钟的驱动下用状态机把要发送的数据传送给上位机

接收模块：在波特率时钟驱动下一旦收到起始位启动接收状态机接收上位机的数据并通过数码管显示。

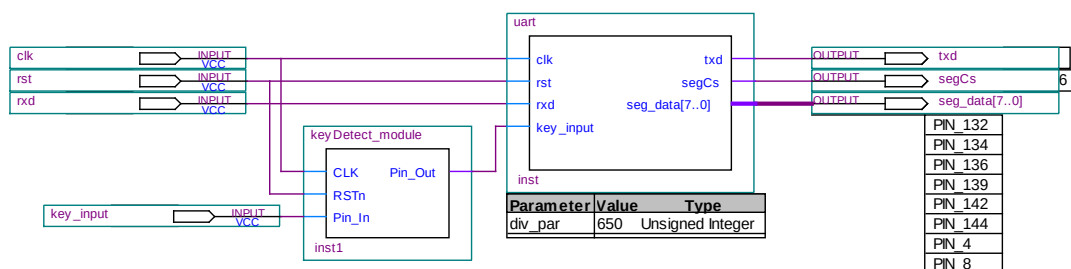


图 8-33 串口通信顶层设计原理图

串口通信设计顶层原理图如图 8-33 所示，输入信号有 `clk`, `rst`，串口接收 `rxd`，按键 `key_input`；输出信号有位选 `seg_Cs` 和段码 `seg_data`，串口发送 `txd`。

2)、实验程序

按键消抖程序在前面章节已经给出，下面给出是串口通信模块的程序。

```
module uart(clk,rst,rxd,txd,segCs,seg_data,key_input);
input clk,rst;
input rxd;           //串行数据接收端
input key_input;     //按键输入
output segCs;
output [7:0] seg_data;
reg [7:0] seg_data;
output txd;          //串行数据发送端
////////////////////inner reg////////////////////////////////////
reg [15:0] div_reg; //分频计数器，分频值由波特率决定。分频后得到频率8倍波特率的时钟
reg [2:0] div8_tras_reg; //该寄存器的计数值对应发送时当前位于的时隙数
reg [2:0] div8_rec_reg; //该寄存器的计数值对应接收时当前位于的时隙数
reg [3:0] state_tras; //发送状态寄存器
reg [3:0] state_rec; //接受状态寄存器
reg clkbaud_tras; //以波特率为频率的发送使能信号
reg clkbaud_rec; //以波特率为频率的接受使能信号
reg clkbaud8x; //以8倍波特率为频率的时钟，它的作用是将发送或接受一个bit的时钟周期分为8个时隙
reg recstart; //开始发送标志
reg recstart_tmp;
reg trasstart; //开始接受标志
reg rxd_reg1; //接收寄存器1
reg rxd_reg2; //接收寄存器2，因为接收数据为异步信号，故用两级缓存
reg txd_reg; //发送寄存器
reg [7:0] rxd_buf; //接受数据缓存
reg [7:0] txd_buf; //发送数据缓存
reg [3:0] send_state; //这是发送状态寄存器
reg key_entry; //确定有键按下标志
parameter div_par=16'd651;
//分频参数，其值由对应的波特率计算而得，按此参数分频的时钟频率是波特率的8
//倍，此处值对应9600的波特率，即分频出的时钟频率是9600*8 (CLK 50M)
assign txd=txd_reg;
assign segCs=0; //7段数码管使能信号赋值
//////////////////////////////////////
always@(posedge clk) //div_reg 计数
begin
    if(!rst)
        div_reg<=0;
    else begin
        if(div_reg==div_par-1'b1)
            div_reg<=0;
        else
            div_reg<=div_reg+1'b1;
    end
end
end
```



```

always@(posedge clk)//分频得到8倍波特率的时钟
begin
    if(!rst)
        clkbaud8x <=0;
    else if(div_reg==div_par-1)
        clkbaud8x <=1'b1;
    else
        clkbaud8x <=1'b0;
end
////////////////////////////////////
always@(posedge clk or negedge rst)
begin
    if(!rst)
        div8_rec_reg<=0;
    else if(recstart & clkbaud8x)//接收开始标志, 开始计数
        div8_rec_reg<=div8_rec_reg+1'b1;//接收开始后, 时隙数在8倍波特率的时钟
        下加1循环
    end
always@(posedge clk)
begin
    if(div8_rec_reg==7)
        clkbaud_rec=1;//在第7个时隙, 接收使能信号有效, 将数据打入
    else
        clkbaud_rec=0;
end
////////////////////////////////////
always@(posedge clk or negedge rst)
begin
    if(!rst)
        div8_tras_reg<=0;
    else if(trasstart & clkbaud8x)//发送开始标志, 开始计数产生发送时钟
        div8_tras_reg<=div8_tras_reg+1'b1;//发送开始后, 时隙数在8倍波特率的时
        钟下加1循环
    end
always@(posedge clk)
begin
    if(div8_tras_reg==7)
        clkbaud_tras=1;//在第7个时隙, 发送使能信号有效, 将数据发出
    else
        clkbaud_tras=0;
end
////////////////////////////////////
always@(posedge clk or negedge rst)
begin
    if(!rst) begin
        txd_reg<=1;
        trasstart<=0;
        txd_buf<=0;
        state_tras<=0;
        send_state<=0;
        key_entry<=0;
    end
    else begin
        if(key_input) begin
            key_entry<=1;
            send_state<=0;
            txd_buf<="U"; //"D"
        end
        else if(clkbaud8x & key_entry) begin
            case(state_tras)
                4'b0000: begin //发送起始位
                    if(!trasstart && send_state<7)
                        trasstart<=1; //发送开始标志, 开始计数产生发送时钟
                    else if(send_state<7) begin
                        if(clkbaud_tras) begin
                            txd_reg<=0; //发送开始标志,
                            state_tras<=state_tras+1'b1;
                        end
                    end
                end
            endcase
        end
    end
end

```



```

        end
    else begin
        key_entry<=0;
        state_tras<=0;
    end
end
4'b0001: begin //发送第1位
    if(clkbaud_tras) begin
        txd_reg<=txd_buf[0];
        txd_buf[6:0]<=txd_buf[7:1];
        state_tras<=state_tras+1'b1;
    end
end
4'b0010: begin //发送第2位
    if(clkbaud_tras) begin
        txd_reg<=txd_buf[0];
        txd_buf[6:0]<=txd_buf[7:1];
        state_tras<=state_tras+1'b1;
    end
end
4'b0011: begin //发送第3位
    if(clkbaud_tras) begin
        txd_reg<=txd_buf[0];
        txd_buf[6:0]<=txd_buf[7:1];
        state_tras<=state_tras+1'b1;
    end
end
4'b0100: begin //发送第4位
    if(clkbaud_tras) begin
        txd_reg<=txd_buf[0];
        txd_buf[6:0]<=txd_buf[7:1];
        state_tras<=state_tras+1'b1;
    end
end
4'b0101: begin //发送第5位
    if(clkbaud_tras) begin
        txd_reg<=txd_buf[0];
        txd_buf[6:0]<=txd_buf[7:1];
        state_tras<=state_tras+1'b1;
    end
end
4'b0110: begin //发送第6位
    if(clkbaud_tras) begin
        txd_reg<=txd_buf[0];
        txd_buf[6:0]<=txd_buf[7:1];
        state_tras<=state_tras+1'b1;
    end
end
4'b0111: begin //发送第7位
    if(clkbaud_tras) begin
        txd_reg<=txd_buf[0];
        txd_buf[6:0]<=txd_buf[7:1];
        state_tras<=state_tras+1'b1;
    end
end
4'b1000: begin //发送第8位
    if(clkbaud_tras) begin
        txd_reg<=txd_buf[0];
        txd_buf[6:0]<=txd_buf[7:1];
        state_tras<=state_tras+1'b1;
    end
end
4'b1001: begin //发送停止位
    if(clkbaud_tras) begin
        txd_reg<=1;
        txd_buf<=8'h55;
        state_tras<=state_tras+1'b1;
    end
end

```

```

end
4'b1111:begin
    if(clkbaud_tras) begin
        state_tras<=state_tras+1'b1;
        send_state<=send_state+1'b1;
        trasstart<=0;
        case(send_state)
            3'b000:
                txd_buf<="A"; //"a"
            3'b001:
                txd_buf<="R"; //"x"
            3'b010:
                txd_buf<="T"; //"i"
            3'b011:
                txd_buf<=" "; //"g"
            3'b100:
                txd_buf<="T"; //"u"
            3'b101:
                txd_buf<="\n"; //"a"
            default:
                txd_buf<=0;
        endcase
    end
end
default: begin
    if(clkbaud_tras) begin
        state_tras<=state_tras+1'b1;
        trasstart<=1;
    end
end
endcase
end
end
end

always@(posedge clkbaud8x or negedge rst)//接受PC机的数据
begin
    if(!rst) begin
        rxd_reg1<=0;rxd_reg2<=0;rxd_buf<=8'h30;
        state_rec<=0;recstart<=0;recstart_tmp<=0;
    end
    else begin
        rxd_reg1<=rxd;
        rxd_reg2<=rxd_reg1;
        if(state_rec==0) begin
            if(recstart_tmp==1) begin//有数据进来，准备启动接收
                recstart<=1;
                recstart_tmp<=0;
                state_rec<=state_rec+1'b1;
            end
            else if(!rxd_reg1&& rxd_reg2) //检测到起始位的下降沿，进入接受状态
                recstart_tmp<=1;
        end
        else if(state_rec>=1&&state_rec<=8) begin//接收8位数据
            if(clkbaud_rec) begin
                rxd_buf[7]<=rxd_reg2;
                rxd_buf[6:0]<=rxd_buf[7:1];
                state_rec<=state_rec+1'b1;
            end
        end
        else if(state_rec==9) begin//接收停止位结束
            if(clkbaud_rec) begin
                state_rec<=0;
                recstart<=0;
            end
        end
    end
end
end
end

```

```

always@(rx_d_buf) //将接受的数据用数码管显示出来. 也可以改成用发光二极管表示
begin
    case (rx_d_buf)
        8'h30:seg_data=8'b11000000;//0
        8'h31:seg_data=8'b11111001;//1
        8'h32:seg_data=8'b10100100;//2
        8'h33:seg_data=8'b10110000;//3
        8'h34:seg_data=8'b10011001;//4
        8'h35:seg_data=8'b10010010;//5
        8'h36:seg_data=8'b10000010;//6
        8'h37:seg_data=8'b11111000;//7
        8'h38:seg_data=8'b10000000;//8
        8'h39:seg_data=8'b10010000;//9
        8'h41:seg_data=8'b10001000;//a
        8'h42:seg_data=8'b10000011;//b
        8'h43:seg_data=8'b11000110;//c
        8'h44:seg_data=8'b10100001;//d
        8'h45:seg_data=8'b10000110;//e
        8'h46:seg_data=8'b10001110;//f
        default:seg_data=8'b10111111;
    endcase
end
endmodule

```

8.11 液晶 LCD1602 显示

1、LCD1602 显示原理说明

1602LCD 分为带背光和不带背光两种，基控制器大部分为 HD44780，带背光的比不带背光的厚，是否带背光在应用中并无差别，具体鉴别办法可参考图 8-34 所示的器件尺寸示意图：

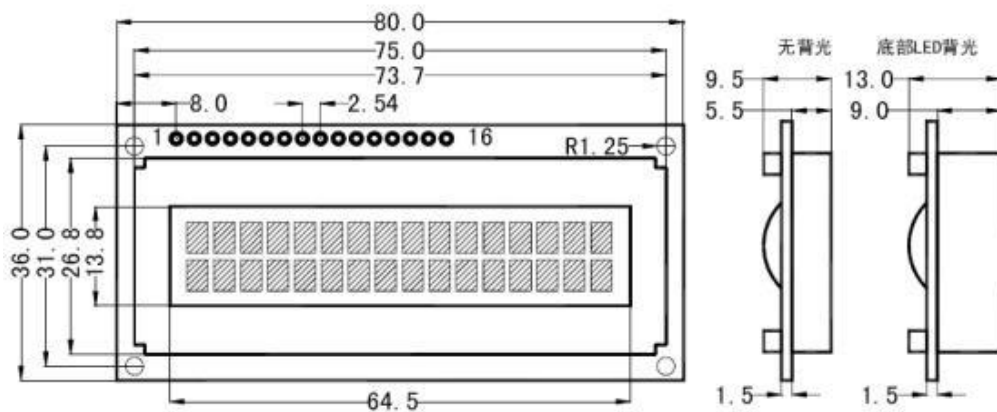


图 8-34 LCD1602 液晶显示器尺寸图

1) LCD1602 常用于显示数字和字母，显示容量为 16*2 个字符，内部有 80 个字节的 RAM 缓冲区，当要向 1602 送数显示时，要先发送显示地址指针 80H+地址码。内部地址码分布如下：

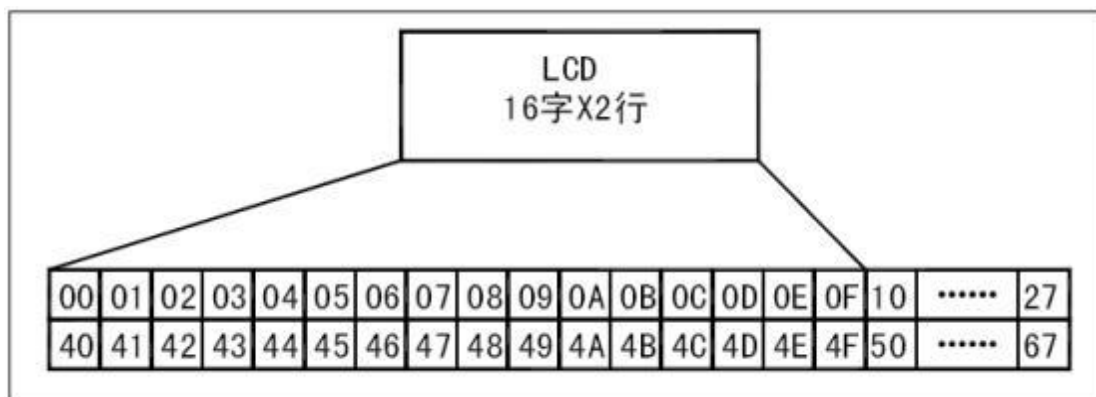


图 8-35 地址分布图

2) LCD1602 有 16 个管脚，其中用于编程操作的有 RS（数据命令），RW（读写），E（使能端），D0-D7（并行数据端口）。

表 1 引脚功能说明

编号	符号	引脚说明	编号	符号	引脚说明
1	VSS	电源地	9	D2	数据
2	VDD	电源正极	10	D3	数据
3	VL	液晶显示偏压	11	D4	数据
4	RS	数据/命令选择	12	D5	数据
5	R/W	读/写选择	13	D6	数据
6	E	使能信号	14	D7	数据
7	D0	数据	15	BLA	背光源正极
8	D1	数据	16	BLK	背光源负极

3) LCD1602 的指令说明及时序

1602 液晶模块内部的控制器共有 11 条控制指令，如表 2 所示：

表 2 控制指令表

序号	指令	RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
1	清显示	0	0	0	0	0	0	0	0	0	1
2	光标返回	0	0	0	0	0	0	0	0	1	*
3	置输入模式	0	0	0	0	0	0	0	1	I/D	S
4	显示开/关控制	0	0	0	0	0	0	1	D	C	B
5	光标或字符移位	0	0	0	0	0	1	S/R	R/	*	*

								C	L		
6	置功能	0	0	0	0	1	D L	N	F	*	*
7	置字符发生存储器地址	0	0	0	1	字符发生存储器地址					
8	置数据存储器地址	0	0	1	显示数据存储器地址						
9	读忙标志或地址	0	1	BF	计数器地址						
10	写数到 CGRAM 或 DDRAM)	1	0	要写的数内容							
11	从 CGRAM 或 DDRAM 读数	1	1	读出的数内容							

4) HD44780 相兼容的芯片时序表如表 3 所示:

表 3 控制芯片时序表

读状态	输入	RS=L, R/W=H, E=H	输出	D0—D7=状态字
写指令	输入	RS=L, R/W=L, D0—D7=指令码, E=高脉冲	输出	无
读数据	输入	RS=H, R/W=H, E=H	输出	D0—D7=数据
写数据	输入	RS=H, R/W=L, D0—D7=数据, E=高脉冲	输出	无

对应的读写操作时序如图 8-36 和 9-37 所示:

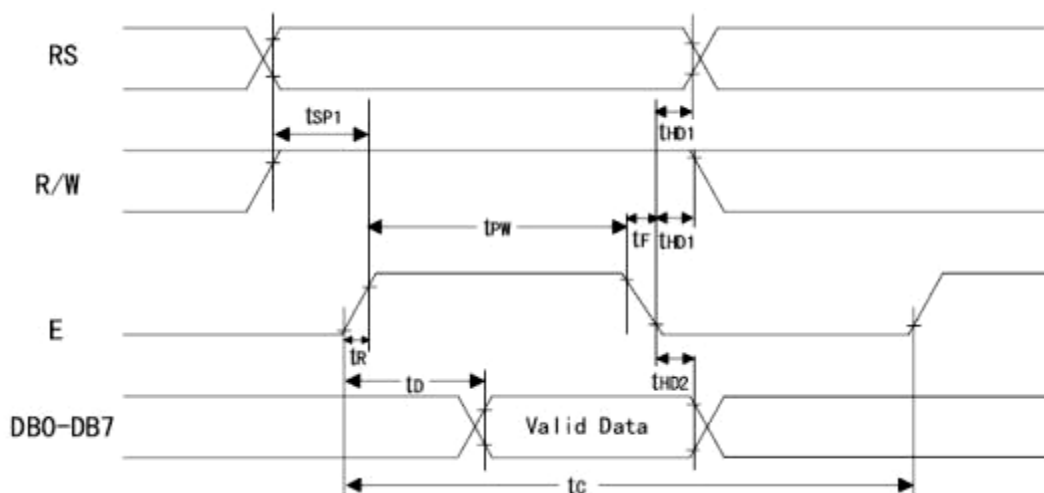


图 8-36 读操作时序

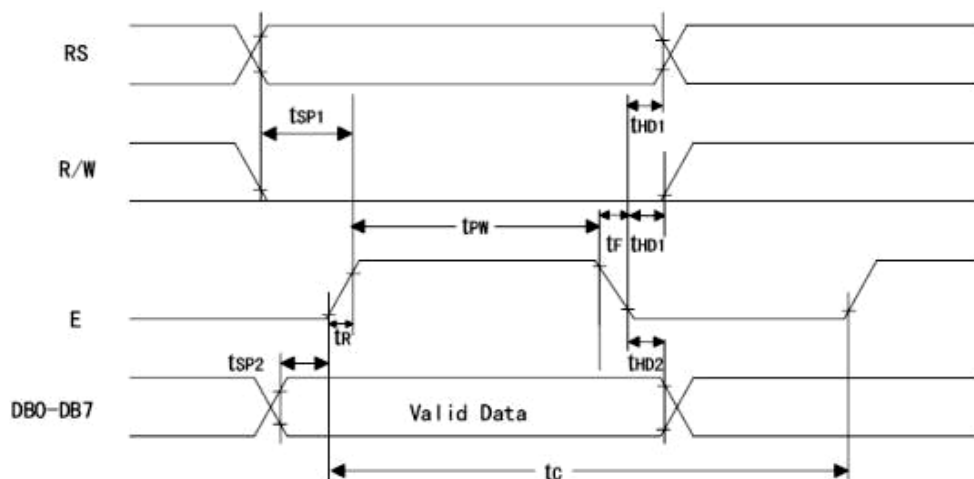


图 8-37 写操作时序

5) LCD1602 液晶屏的一般初始化（复位）过程

按照常规编程操作，LCD1602 液晶屏的一般初始化步骤如下列举：

- ① 延时 10mS，写指令 38H（不检测忙信号）；
- ② 之后每次写指令、读/写数据操作均需要检测忙信号；
- ③ 写指令 08H：显示关闭；
- ④ 写指令 01H：显示清屏；
- ⑤ 写指令 38H：显示模式设置；
- ⑥ 写指令 06H：显示光标移动设置；
- ⑦ 写指令 0CH：显示开及光标设置。

（其指令说明请详细查看 1602 数据手册）。

2、实验设计要求

在开发板上 LCD1602 液晶上显示英文字符和数字。

3、硬件电路原理图

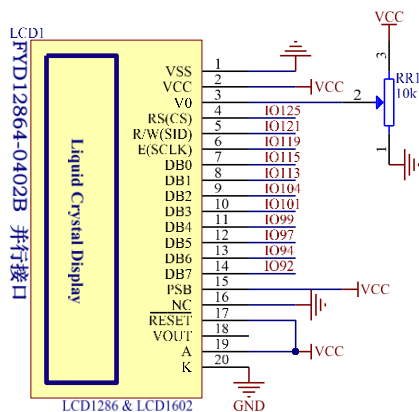


图 8-38 液晶显示原理图

该插口是 1602 和 12864 共用的插口,编号是 1-16 引脚是用来接 LCD1602 的, 15 引脚 PSB 是 LCD1602 背光源正极因此要输出高电平, 16 引脚是背光接地。RR1 可调电阻作用是调整液晶屏幕对比度, 改变电位器则可改变屏幕对比度, 使之显示效果最优。

4、实验程序设计

1) 设计思路

Verilog 的编程只要遵循时序图描写出时序图那样的信号和满足其时序参数就可以实现 FPGA 控制 1602 显示, 并且可用状态机实现。

其中由于是往 1602 写数据因此 RW 可一直置 0 电平, 使能信号 E 可以做成时钟信号用于控制数据的传输。

为了以后方调用液晶显示程序, 设计采用分模块设计程序, 分显示控制模块和读写操作模块, 顶层设计原理图如下图所示。

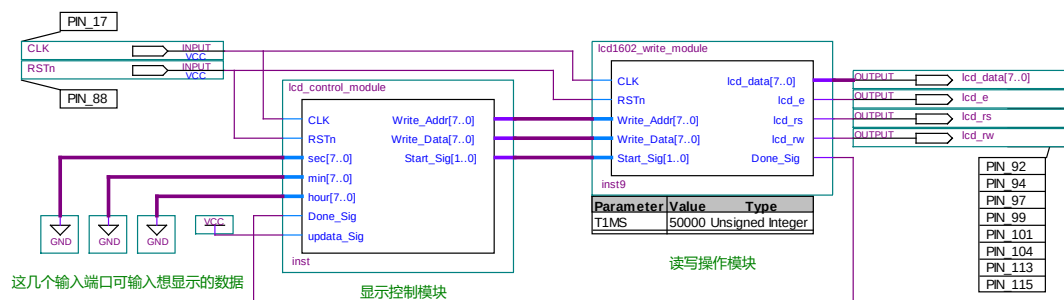


图 8-39 液晶显示顶层设计原理图

顶层设计原理图输入信号有 CLK, RSTn, 三个预留的显示内容输入端; 输出信号有液晶数据口 lcd_data, 读写控制端 lcd_rw, 命令数据控制端 lcd_rs, 读写使能控制端 lcd_e。

2) 实验代码

a. 显示控制模块代码

```

1. module lcd_control_module
2. (
3.     CLK, RSTn,
4.     sec,
5.     min,
6.     hour,
7.     Done_Sig,
8.     Write_Addr,
9.     Write_Data,
10.    updata_Sig,
11.    Start_Sig

```

```

12. );
13. input CLK;
14. input RSTn;
15. input Done_Sig;
16. input[7:0] sec;
17. input[7:0] min;
18. input[7:0] hour;
19. input updata_Sig;
20. output[7:0] Write_Addr;
21. output[7:0] Write_Data;
22. output[1:0] Start_Sig;
23. /*****/
24. reg [4:0] i;
25. reg [1:0] isStart;
26. reg [7:0] rWrite_Addr;
27. reg [7:0] rWrite_Data;
28.
29. always @(posedge CLK or negedge RSTn)
30. begin
31.     if(!RSTn)
32.     begin
33.         i <= 5'd0;
34.         isStart <= 2'b00;
35.         rWrite_Addr <= 8'd0;
36.         rWrite_Data <= 8'd0;
37.     end
38.     else
39.     case(i)
40.     0:
41.         if(Done_Sig)begin isStart <= 2'b00;i <=i+1'b1;end
42.         else isStart <= 2'b10;//液晶初始化控制端口
43.     1:
44.         if(Done_Sig)begin isStart <= 2'b00;i <=i+1'b1;end//显示字符
45.         else begin isStart <= 2'b01;rWrite_Addr<=8'h80; rWrite_Data<="L"; end
46.     2:
47.         if(Done_Sig)begin isStart <= 2'b00;i <=i+1'b1;end
48.         else begin isStart <= 2'b01;rWrite_Addr<=8'h81; rWrite_Data<="c"; end
49.     3:
50.         if(Done_Sig)begin isStart <= 2'b00;i <=i+1'b1;end
51.         else begin isStart <= 2'b01;rWrite_Addr<=8'h82; rWrite_Data<="d"; end
52.     4:

```



```

53.     if(Done_Sig)begin isStart <= 2'b00;i <=i+1'b1;end
54.     else begin isStart <= 2'b01;rWrite_Addr<=8'h83; rWrite_Data<="_"; end
55.     5:
56.     if(Done_Sig)begin isStart <= 2'b00;i <=i+1'b1;end
57.     else begin isStart <= 2'b01;rWrite_Addr<=8'h84; rWrite_Data<="c"; end
58.     6:
59.     if(Done_Sig)begin isStart <= 2'b00;i <=i+1'b1;end
60.     else begin isStart <= 2'b01;rWrite_Addr<=8'h85; rWrite_Data<="l"; end
61.     7:
62.     if(Done_Sig)begin isStart <= 2'b00;i <=i+1'b1;end
63.     else begin isStart <= 2'b01;rWrite_Addr<=8'h86; rWrite_Data<="o"; end
64.     8:
65.     if(Done_Sig)begin isStart <= 2'b00;i <=i+1'b1;end
66.     else begin isStart <= 2'b01;rWrite_Addr<=8'h87; rWrite_Data<="c"; end
67.     9:
68.     if(Done_Sig)begin isStart <= 2'b00;i <=i+1'b1;end
69.     else begin isStart <= 2'b01;rWrite_Addr<=8'h88; rWrite_Data<="k"; end
70.     10:
71.     if(updata_Sig) i <=i+1'b1; //data updata
72.     11:
73.     if(Done_Sig)begin isStart <= 2'b00;i <=i+1'b1;end //显示外部引进来的数字
74.     else begin isStart <= 2'b01;rWrite_Addr<=8'hc0;
75.         rWrite_Data<= {4'd0, hour[7:4]}|8'h30; end
76.     12:
77.     if(Done_Sig)begin isStart <= 2'b00;i <=i+1'b1;end
78.     else begin isStart <= 2'b01; rWrite_Addr<=8'hc1;
79.         rWrite_Data<= {4'd0, hour[3:0]}|8'h30; end
80.     13:
81.     if(Done_Sig)begin isStart <= 2'b00;i <=i+1'b1;end
82.     else begin isStart <= 2'b01; rWrite_Addr<=8'hc2;rWrite_Data<=":";end
83.     14:
84.     if(Done_Sig)begin isStart <= 2'b00;i <=i+1'b1;end
85.     else begin isStart <= 2'b01;rWrite_Addr<=8'hc3;
86.         rWrite_Data<= {4'd0, min[7:4]}|8'h30; end
87.     15:
88.     if(Done_Sig)begin isStart <= 2'b00;i <=i+1'b1;end
89.     else begin isStart <= 2'b01;rWrite_Addr<=8'hc4;
90.         rWrite_Data<= {4'd0, min[3:0]}|8'h30; end
91.     16:
92.     if(Done_Sig)begin isStart <= 2'b00;i <=i+1'b1;end
93.     else begin isStart <= 2'b01; rWrite_Addr<=8'hc5;rWrite_Data<=":"; end

```

```

94.     17:
95.     if(Done_Sig)begin isStart <= 2'b00;i <=i+1'b1;end
96.     else begin isStart <= 2'b01;rWrite_Addr<=8'hc6;
97.         rWrite_Data<= {4'd0,sec[7:4]}|8'h30; end
98.     18:
99.     if(Done_Sig)begin isStart <= 2'b00;i <=i+1'b1;end
100.    else begin isStart <= 2'b01;rWrite_Addr<=8'hc7;
101.        rWrite_Data<= {4'd0,sec[3:0]}|8'h30; end
102.    19:
103.    i <=5'd10;
104.    endcase
105. end
106. /*****/
107. assign Write_Addr = rWrite_Addr ;
108. assign Write_Data = rWrite_Data;
109. assign Start_Sig = isStart;
110. /*****/
111. Endmodule

```

b. 读写操作模块代码

```

1. module lcd1602_write_module
2. (
3.     CLK,RSTn,Write_Addr, Write_Data,Start_Sig,
4.     lcd_data,lcd_e,lcd_rs,lcd_rw, Done_Sig
5. );
6. input CLK,RSTn;
7. input[7:0]Write_Addr;
8. input[7:0]Write_Data;
9. input[1:0] Start_Sig;
10. output [7:0]lcd_data;
11. output lcd_e,lcd_rs,lcd_rw;
12. output Done_Sig;
13. /*****/
14. parameter T1MS = 16'd50_000;
15. reg[15:0]Count1;
16. always @(posedge CLK or negedge RSTn)
17. begin
18.     if(!RSTn)
19.         Count1 <= 15'd0;
20.     else if(Count1 == T1MS)
21.         Count1 <= 15'd0;
22.     else if(Start_Sig[0] == 1'b1 || Start_Sig[1]==1'b1)

```

```

23.         Count1 <= Count1 + 1'b1;
24.     else
25.         Count1 <= 15'd0;
26.     end
27. //*****
28.     reg[9:0]Count_MS;
29. always @(posedge CLK or negedge RSTn)
30.     if(!RSTn)
31.         Count_MS <= 10'd0;
32.     else if(isCount && Count1 == T1MS)
33.         Count_MS <=Count_MS +1'b1;
34.     else if(!isCount)
35.         Count_MS <= 10'd0;
36. //*****
37.     reg [3:0]i;
38.     reg [7:0]rlcd_data;
39.     reg  rlcd_e;
40.     reg  rlcd_rs;
41.     reg  rlcd_rw;
42.     reg  isDone;
43.     reg  isCount;
44. always @(posedge CLK or negedge RSTn)
45.     begin
46.         if(!RSTn)
47.             begin
48.                 i <=4'd0;
49.                 rlcd_data <= 8'd0;
50.                 rlcd_e <= 1'b0;
51.                 rlcd_rs <= 1'b0;
52.                 rlcd_rw <= 1'b0;
53.                 isDone <= 1'b0;
54.                 isCount <=1'b0;
55.             end
56.         else if(Start_Sig[1])//初始化液晶部分
57.             begin
58.                 case(i)
59.                     0: if(Count1 == T1MS)begin rlcd_e <= 1'b1; rlcd_rs <= 1'b0;
60.                         rlcd_rw <= 1'b0; rlcd_data <= 8'h38; i <=i+1'b1; end
61.                     1: if(Count1 == T1MS)begin rlcd_e <= 1'b0; i <= i+1'd1;end
62.                     2: if(Count1 == T1MS)begin rlcd_e <= 1'b1; rlcd_data <= 8'h08;
63.                         i <= i+1'd1;end

```

```

64.      3: if(Count1 == T1MS)begin rlcd_e <= 1'b0; i <= i+1'd1; end
65.      4: if(Count1 == T1MS)begin rlcd_e <= 1'b1; rlcd_data <= 8'h01;
66.          i <= i+1'd1;end
67.      5: if(Count1 == T1MS)begin rlcd_e <= 1'b0; i <= i+1'd1; end
68.      6: if(Count1 == T1MS)begin rlcd_e <= 1'b1; rlcd_data <= 8'h06;
69.          i <= i+1'd1;end
70.      7: if(Count1 == T1MS)begin rlcd_e <= 1'b0; i <= i+1'd1; end
71.      8: if(Count1 == T1MS)begin rlcd_e <= 1'b1; rlcd_data <= 8'h0c;
72.          i <= i+1'd1;end
73.      9: if(Count1 == T1MS)begin rlcd_e <= 1'b0; i <= i+1'd1; end
74.     10: if(Count1 == T1MS)begin rlcd_e <= 1'b1; isDone <= 1'b1;
75.          i <= i+1'd1;end
76.     11: begin isDone <= 1'b0;i <= 4'd0;end
77. endcase
78. end
79. else if(Start_Sig[0]) //显示数据部分
80.     begin
81.         case(i)
82.         0: if(Count_MS == 10'd10)begin isCount <= 1'b0; i<=i+1'b1;end
83.             else begin isCount <= 1'b1;end//延时处理
84.         1: if(Count1 == T1MS)begin rlcd_e <= 1'b1; rlcd_rs <= 1'b0;
85.             rlcd_rw <= 1'b0; rlcd_data <= Write_Addr; i <=i+1'b1;end
86.         2: if(Count1 == T1MS)begin rlcd_e <= 1'b0; i <= i+1'd1;end
87.         3: if(Count1 == T1MS)begin rlcd_e <= 1'b1; rlcd_rs <= 1'b1;
88.             rlcd_rw <= 1'b0; rlcd_data <= Write_Data; i <=i+1'b1; end
89.         4: if(Count1 == T1MS)begin rlcd_e <= 1'b0; i <= i+1'd1;end
90.         5: if(Count1 == T1MS)begin rlcd_e <= 1'b1;isDone <= 1'b1;
91.             i <= i+1'd1;end
92.         6: begin isDone <= 1'b0;; i <= 4'd0;end
93.     endcase
94. end
95. end
96. /*****/
97. assign lcd_data=rlcd_data;
98. assign lcd_e=rlcd_e;
99. assign lcd_rs=rlcd_rs;
100. assign lcd_rw=rlcd_rw;
101. assign Done_Sig=isDone;
102. /*****/
103. endmodule

```

5、实验现象

液晶显示字符: Lcd_clock, 显示数字 00:00:00, 可用来作为显示时钟做准备。

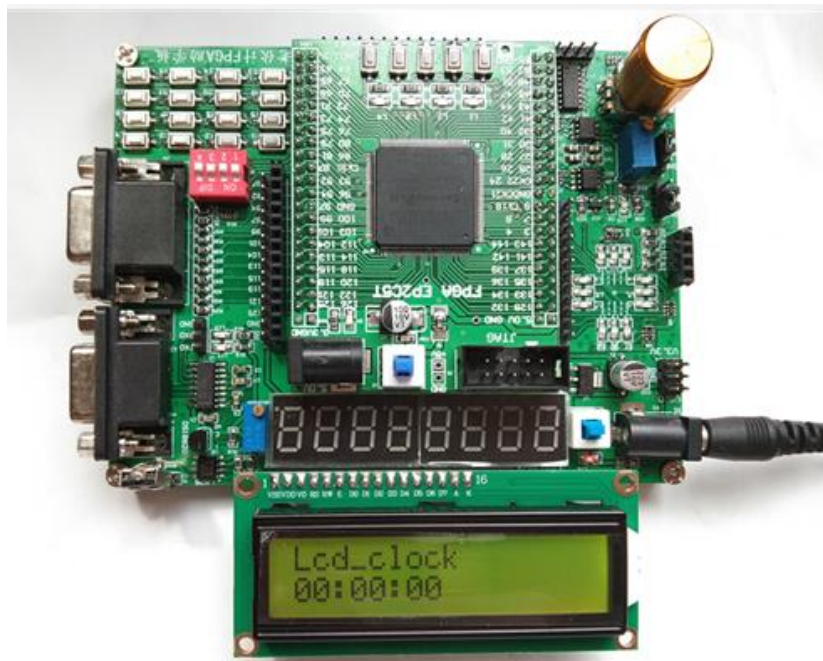


图 8-40 液晶显示实验现象

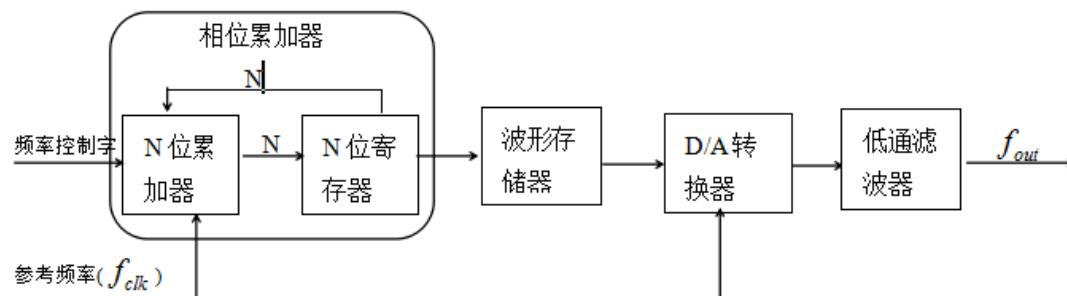
8.12 DDS 与嵌入式逻辑分析仪的调用

1. DDS 的原理

DDS(Direct Digital Frequency Synthesizer) 直接数字频率合成器, 也可叫 DDFS。DDS 是从相位的概念直接合成所需波形的一种频率合成技术。不仅可以产生不同频率的正弦波, 而且可以控制波形的初始相位。

(1) DDS 的主要组成部分: DDS 的结构主要由相位累加器、波形存储器、数模(D/A)转换器和低通滤波器等四个大的结构组成。其结构框图如下

DDS 原理框图如下所示:



(a)

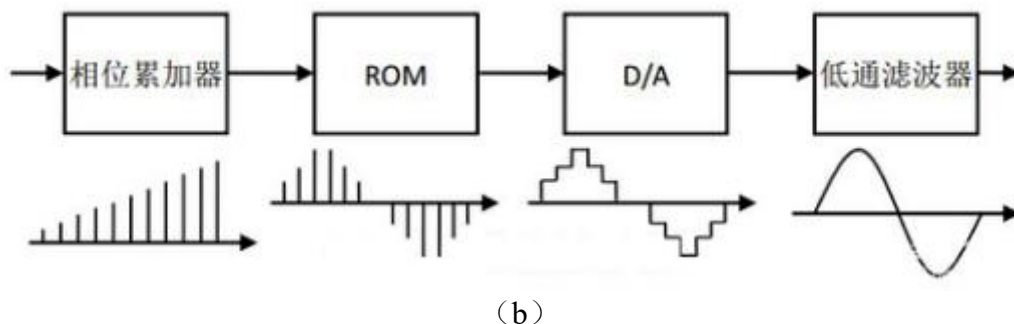


图 8-41 DDS 原理框图

相位累加器是由 N 位加法器与 N 位累加寄存器构成，它是 DDS 模块中一个极其重要的部分。在参考频率时钟的驱动下，DDS 模块开始工作；当每来一个参考时钟时，累加器就把频率控制字 K 与寄存器输出的值进行累加，将相加后的结果再输入到寄存器中，而累加寄存器就将在上一个参考时钟作用时产生的数据通过反馈的方式输送到累加器中。这样，在时钟的作用下，就可以不停的对频率控制字进行累加。

此时 ROM 中要存放好要显示的一个周期的波形数据，用相位累加器输出的数据作为地址在波形存储器中通过查找地址所对应的数据表，然后 D/A 转换器根据 ROM 输出的数据输出相应的电压值，完成其从数字量到模拟波形之间的转化。但是由于数字的离散性，DA 输出的也将是离散的电压值，也就构成了不平滑的正弦波，这时，只需在后级加一个低通滤波器就可以输出完美的正弦波。

(2) 工作过程：

- a) 将存于 ROM 中的数字波形，经 DAC，形成模拟量波形。
- b) 改变寻址的步长来改变输出信号的频率。步长即为对数字波形查表的相位增量。由累加器对相位增量进行累加，累加器的值作为查表地址。
- c) DAC 输出的阶梯形波形，经低通滤波，成为模拟波形。

(3) 频率控制原理：

由于相位累加器位宽的限制，相位累加器累加到一定值后，其输出将会溢出，这样波形存储器的地址就会循环一次，即意味着输出波形循环一周。故改变频率控制字即相位增量，就可以改变相位累加器的溢出时间，在时钟频率不变的条件下就可以改变输出频率。改变查表寻址的时钟频率，同样也可以改变输出波形的频率。

为了获得较高的频率分辨率，则只有增加相位累加器的字长 N ，故一般 N 都取值较大。但是受存储器容量的限制，存储器地址线的位数 W 不可能很大，一般都要小于 N 。这样存储器的地址线一般都只能接在相位累加器输出的高 W 位，

而相位累加器输出余下的(N-W)个低位则只能被舍弃，这就是相位截断误差的来源。

首先我们要考虑以下两个问题：

a) 相位累加器（计数器）的位宽 N 是多少？

b) ROM 的数据位宽 W 和深度（深度： $2^{\text{地址位宽}}$ ）是多少？

DDS 模块的输出频率 F_o 是系统工作频率 F_s 、相位累加器位数 n 及频率控制字 K 满足如下关系：

$$F_o = K * F_s / 2^n$$

假设相位累加器为 24 位，系统时钟是 50MHz，那么输出信号的最低频率就是 $F = 50\text{MHz} / 2^{24} = 2.98\text{Hz}$ ，就是没一个系统时钟累加器加 1。如果我们想要其他的频率，再在这个基础上乘上频率控制字就行了。怎么乘上频率控制字？根据计算频率输出计算公式（ $F_o = K * F_s / 2^n$ ），如果要输出频率 $f_o = 1000\text{Hz}$ 的，系统时钟 $F_s = 50\text{MHz}$ ，则频率控制字 $K = 1000 / 2.98 = 335.54$ ，累加器在每个系统时钟到来时累加 336，这样就可以实现比较灵活的调频。

我们可以通过反推去理解分析 DDS 怎么得到输出频率 $f_o = 1000\text{Hz}$ ，因为 24 位地址累加器，每次从 0 到累加到满（溢出）就可以产生一个周期的信号，1000Hz 就是在 1 秒时间内累加器要累加满 1000 次。因为系统时钟是 50MHz，就是每秒时间有 50000000 个时钟周期，那么累加器累加满一个周期需要的时钟个数是 $50000000 / 1000 = 50000$ 。在 50000 个时钟周期内累加器要累加溢出一次，则每个时钟累加的数据是 $2^{24} / 50000 = 336$ ；要输出其他频率就可以依此类推，从而证明公式（ $F_o = K * F_s / 2^n$ ）计算出频率控制字 K 是正确的。

（4）相位控制：

调相相对来说比较简单，改变波形输出数据的初始相位，原理就是改变地址累加器的偏移地址，实现方法就是定义一个调相控制字，当复位赋初值的时候将想要的初始相位赋给地址初值就可以了，实现起来相对简单。

调相计算公式：假设是地址累加器是 24 位的则有

$$\text{调相控制字} = 2^{24} * (\text{初始相位} / 360)$$

（5）波形存储

在 FPGA 内部有 ROM 存储器。把几种已制作好的波形数据预先存入 ROM 里面，再调用。这里有 4 个 ROM，可以输出 4 个波形，正弦波，方波，三角波，锯齿波，如下图 4 所示。根据需要添加或者减少。

ROM 的地址 $\text{address}[7..0]$ 是 8 位的，说明里面有 256 个存储位置（ 2^8 ），ROM

里面存放的是一种波形的一个周期的数据，比如方波，一半是对应高电平最大值是 0xFF，一半是对应低电平最小值是 0x00，这些数字在地址（由累加器提供地址）和时钟的控制下不断从 q[7..0] 输出，然后送到 DA 转换器进行转换，这样就可以把 0x00----0xFF 这些数字量转换为模拟量，通过示波器观察可以看到波形的输出。

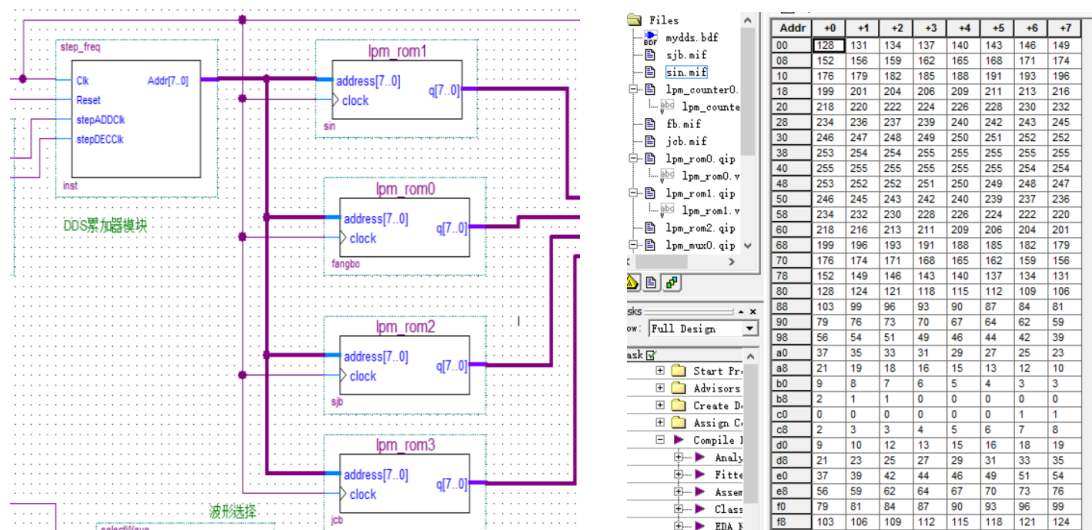


图 8-42 调用出来的 ROM 模块和 ROM 中存放的波形数据 mif 文件

2. MIF 文件的生成

(1) 为了将数据装入 ROM 中，在加入并设置 ROM 之前，应首先建立一个存储器初值设定文件（或称为 .mif 格式文件）。建立存储器初值设定文件的操作如下：

① 在 Quartus II 集成环境下，执行“File”菜单的“New”命令，打开一个新的“Memory initialization file”（存储器初值设定文件）编辑窗口，在弹出如图 8-43 所示的存储器参数设置对话框中输入存储器的字数（Number of words）为 256，字长（Word size）为 8 位。

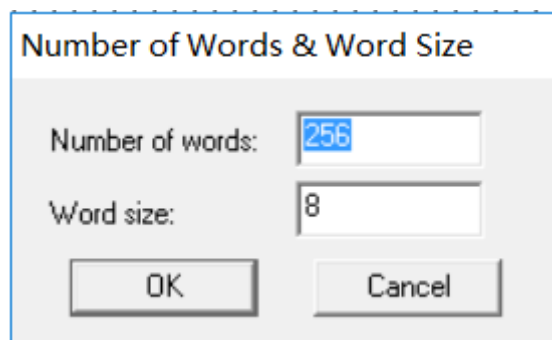


图 8-43 存储器参数设置对话框

② 存储器的参数设置结束后用鼠标单击“OK”按钮，弹出如图 8-44 所示存储器初值设定文件的界面，将此文件以.mif 为类型属性（如 mydds.mif）保存在工程目录中。在存储器初值设定文件的界面中，用鼠标右键点击存储器的某个地址（如 248），弹出 Address Radix（地址基数）和 Memory Radix（存储器基数）选择快捷菜单。执行 Address Radix 项则可对存储器的地址基数进行选择，地址有 Binary（二进制）、Decimal（十进制）、Octal（八进制）和 Hexadecimal（十六进制）4 种基数数制选择。执行 Memory Radix 项则可对存储器单元中的数据基数进行设置，存储器数据有 Binary（二进制）、Hexadecimal（十六进制）、Octal（八进制）、Signed Decimal（带符号十进制）和 Unsigned Decimal（无符号十进制）5 种基数选择，本设计选择“Unsigned Decimal”。

Addr	+0	+1	+2	+3	+4	+5	+6	+7
0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0
32	0	0	0	0	0	0	0	0
40	0	0	0	0	0	0	0	0
48	0	0	0	0	0	0	0	0
56	0	0	0	0	0	0	0	0
64	0	0	0	0	0	0	0	0
72	0	0	0	0	0	0	0	0
80	0	0	0	0	0	0	0	0
88	0	0	0	0	0	0	0	0
96	0	0	0	0	0	0	0	0
104	0	0	0	0	0	0	0	0
112	0	0	0	0	0	0	0	0
120	0	0	0	0	0	0	0	0
128	0	0	0	0	0	0	0	0
136	0	0	0	0	0	0	0	0
144	0	0	0	0	0	0	0	0
152	0	0	0	0	0	0	0	0
160	0	0	0	0	0	0	0	0
168	0	0	0	0	0	0	0	0
176	0	0	0	0	0	0	0	0
184	0	0	0	0	0	0	0	0
192	0	0	0	0	0	0	0	0
200	0	0	0	0	0	0	0	0
208	0	0	0	0	0	0	0	0
216	0	0	0	0	0	0	0	0
224	0	0	0	0	0	0	0	0
232	0	0	0	0	0	0	0	0
240	0	0	0	0	0	0	0	0
248	0	0	0	0	0	0	0	0

图 8-44 存储器初值设定文件的界面

③ 将数据加入存储器初值设定文件中。新建的存储器初值设定文件中的数据全部为 0，在存储器初值设定文件的界面可以直接输入每个存储器字的数据，也可以用鼠标右键点击文件界面，在弹出如图 8-45 所示的格式文件操作快捷菜单提示下，完成数据输入。

例如，在弹出的格式文件操作快捷菜单中选择“Custom Fill Cells”（块填充）项，弹出如图 8-46 所示的“Custom Fill Cells”对话框。在对话框的“Starting address”栏目内输入起始地址（如 00），在“Ending address”栏目内输入结束地址（如 ff）；将“Incrementing/Decrementing”选中后，在“Starting Value”栏目中输入起始值（如 0），在“Increment by”（或 Decrement by）栏目中输入增加（或减少）值（如

2)。完成上述操作后用鼠标单击“OK”按钮，结束.mif 格式文件中的数据填充。数据填充的结果为：从 00 地址开始到 ff 地址结束，数据由 00 值开始存储于 00 地址单元，并将此值递增 2 后填入下一个存储单元；当递增的值大于 8 位二进制数的最大值（即 255）后，数据又从 00 值开始重新填写，直至结束地址为止。用上述方法产生的存储器初始数据实际是一个锯齿波发生器的数据。

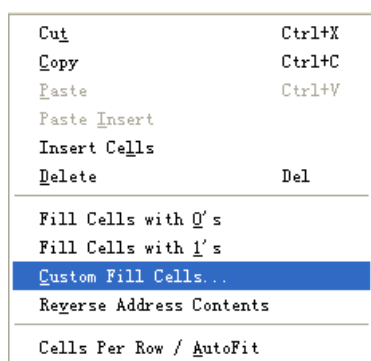


图 9-45 格式文件操作快捷菜单

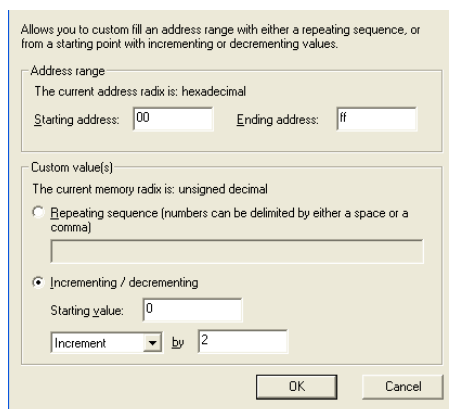
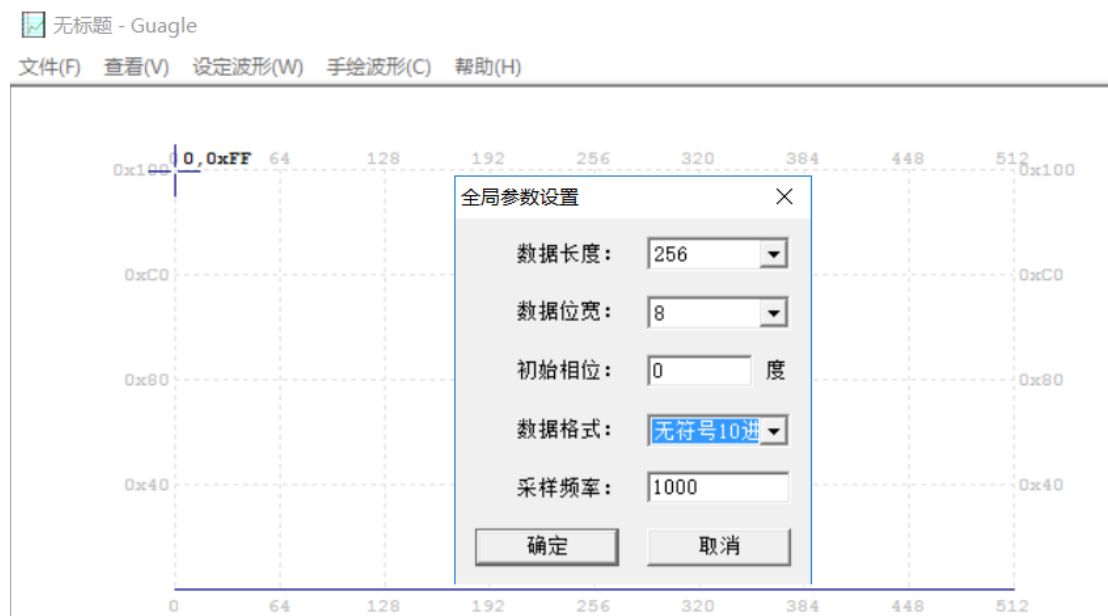


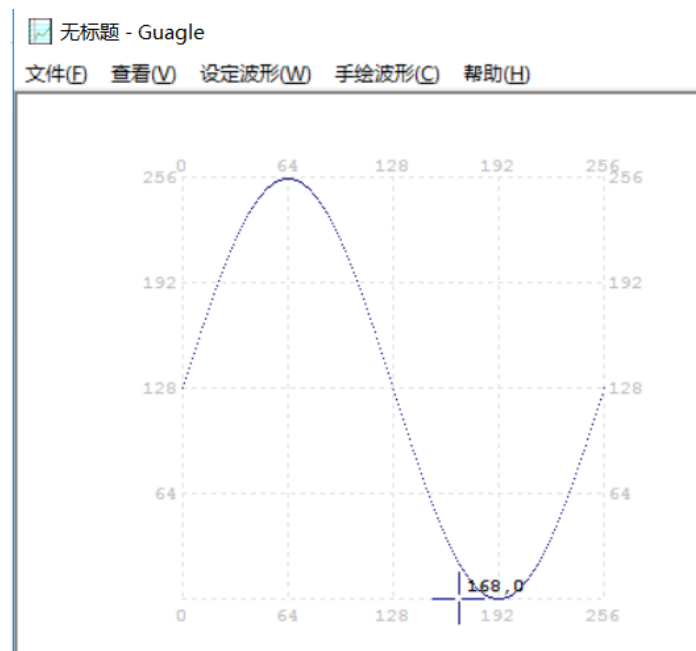
图 9-46 “Custom Fill Cells”对话框

(2) 另外一种比较方便的方法是采用 Mif_Maker2010 软件进行生成。步骤如下：

1) 打开 Mif_Maker2010 软件，点击菜单【设定波形】---【全局参数的设置】--【确定】--【设定波形】--【正弦波】，即可得到相应的波形；



(a) 设置参数



(b) 选择生成波形

图 8-47 采用 Mif_Maker2010 软件生成波形

2) 点击【文件】—【保存】，选择保存途径并进行命名即可得到生成的 MIF 文件。

3. ROM 的生成

用鼠标双击原理图编辑窗，在弹出的元件选择窗的“Libraries”栏目中选择“storage”的“lpm_rom”（只读存储器 ROM）LPM 元件，如图 8-48 所示。用鼠标单击“OK”按钮后弹出如图 8-49 所示的“MegaWizard Plug-In Manager[page 2c]”对话框页面。在该对话框页面中，选择 VHDL（或 Verilog HDL 或 AHDL）作为输出文件的类型，并将生成的只读存储器名称及保存的文件夹（如 D:\myeda_2\lpm_rom0）输入到“What name do you want for the output file？”栏目中。

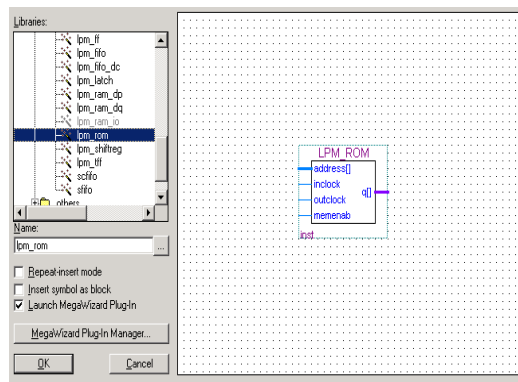


图 8-48 ROM 元件选择窗

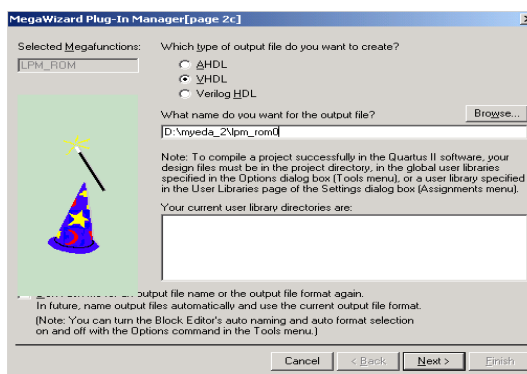


图 8-49 MegaWizard Plug-In Manager

完成上述操作后，用鼠标单击“Next”按钮，进入 ROM 参数设置的下一个对话框页面“MegaWizard Plug-In Manager–LPM_ROM[page 3 of 7]”，如图 8-50 所示。在此页面中设置 ROM 的 q 输出位数为 8bit，字数为 256，采用单时钟控制方式。

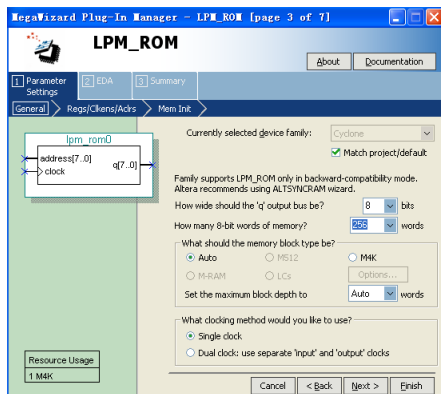


图 8-50 LPM_ROM[page 3 of 7]

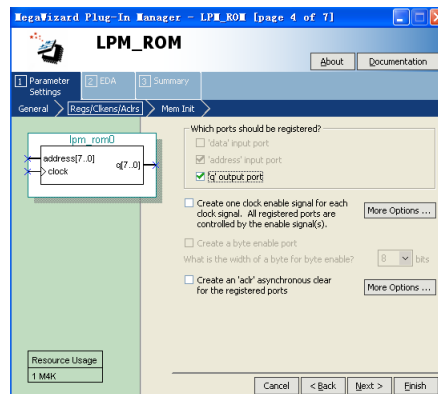


图 8-51 LPM_ROM[page 3 of 7]

完成页的参数设置后用鼠标单击“Next”按钮，进入 ROM 参数设置的“MegaWizard Plug-In Manager–LPM_ROM[page 4 of 7]”对话框页面，如图 8-51 所示。此对话框页面用于选择 ROM 的 clken（时钟使能）和 aclr（清除）输入控制端（本设计不选择）。

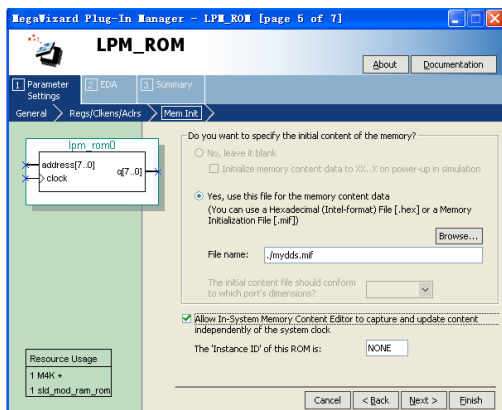


图 8-52 LPM_ROM[page 5 of 7]

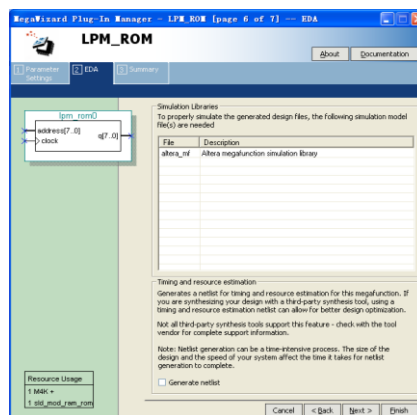


图 8-53 LPM_ROM[page 6 of 7]

完成此页的参数设置后用鼠标单击“Next”按钮，进入 ROM 参数设置的“MegaWizard Plug-In Manager–LPM_ROM[page 5 of 7]”对话框页面，如图 8-52 所示。在此页面的“Do you want to...”栏目中选中“Yes, use this file for the memory content data”项，并输入初始化数据文件名（如 mydds.mif）。另外，将“Allow In-System Memory ...”项选中，表示允许 Quartus II 能通过 JTAG 口对下载于 FPGA 中的 ROM 进行在系统测试和读写。

完成此对话框的参数设置后用鼠标单击“Next”按钮，进入计数器参数设置

的“MegaWizard Plug-In Manager –LPM_COUNTER[page 6 of 7]”对话框，如图 8-53 所示。此对话框是仿真库的列表文件，保持默认后用鼠标单击“Next”按钮，进入 ROM 参数设置的“MegaWizard Plug-In Manager –LPM_ROM[page 7 of 7]”对话框页面。这是 ROM 参数设置的最后一个页面，此页面主要用于选择生成 ROM 的输出文件。至此，ROM 参数设置完成，用鼠标单击“Finish”按钮结束设置，在工程管理窗口可以看到生成的文件。

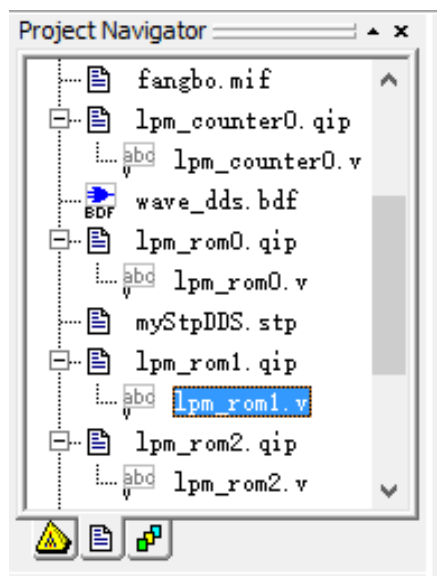


图 8-54

4. DDS 系统实验程序设计

DDS 设计的顶层原理图如下所示：

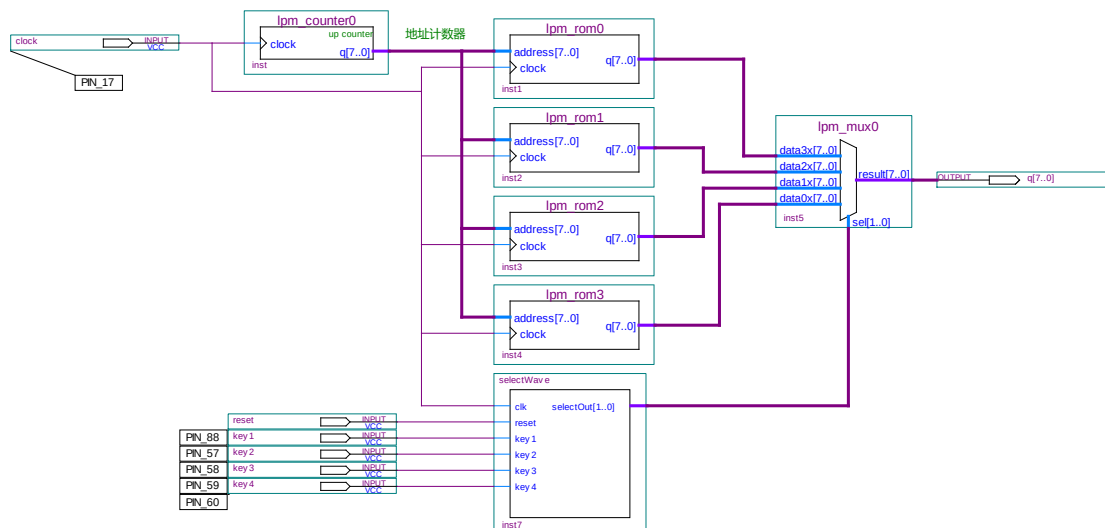


图 8-55 DDS 设计的顶层原理图

其中, `lpm_counter0` 是 LPM 计数器, 生成过程是在弹出的元件选择窗的“Libraries”栏目中选择“arithmetic”的“`lpm_counter`”(计数器) LPM 元件, 按照对话框提示一步一步设置, 只需要时钟输入 `clock` 和数据输出 `q[]` 端口。

`lpm_rom0` 到 `lpm_rom3` 是 LPM 只读存储器 (ROM)。ROM 中保存的是某种波形信号 (如锯齿波或正弦波) 的数据, 其地址由计数器 `lpm_counter0` 提供。`lpm_counter0` 是一个 8 位加法计数器, 在时钟的控制下计数器的输出 `q[7..0]` 由 00000000 到 11111111 循环变化, 使 ROM 输出周期性的波形信号的数据。`selectWave` 模块是波形选择模块, 由 FPGA 核心板上 4 个选择按钮选择不同的波形输出, `lpm_mux0` 是四选一数据选择器, 根据 `selectWave` 模块的选择值确定输出哪个波形。

5. 嵌入式逻辑分析仪的使用

Quartus II 的嵌入式逻辑分析仪 SignalTap II 是一种高效的硬件测试手段, 它可以随设计文件一并下载到目标芯片中, 捕捉目标芯片内部系统信号节点处的信息或总线上的数据流, 而又不影响原硬件系统的正常工作。在实际监测中, SignalTap II 将测得的样本信号暂存于目标芯片的嵌入式 RAM 中, 然后通过器件的 JTAG 端口将采到的信息传出, 送到计算机进行显示和分析。在复杂的设计中, 不能从外部的输入输出引脚上观察内部端口之间 (如模块与模块之间) 的信号波形是否正确, 这就可以使用 SignalTap II 逻辑分析仪来进行观察。

下面以波形发生器 (`mydds`) 为例, 介绍嵌入式逻辑分析仪 SignalTap II 的使用方法。

嵌入式逻辑分析仪 SignalTap II 的设置分为打开 SignalTap II 编辑窗口、调入节点信号、SignalTap II 参数设置、文件存盘、编译、下载和运行分析等操作过程。

(1) 打开 SignalTap II 编辑窗口

在波形发生器工程 (`mydds`) 完成引脚锁定并通过编译后, 执行 Quartus II 主窗口的 File→New 命令, 在弹出的 New 窗口中 (参见图 2.18 所示), 选择打开 SignalTap II Logic Analyzer File 文件, 弹出如图 2.54 所示的 SignalTap II 编辑窗口。SignalTap II 编辑主窗口中包含实例 (Instance)、信号观察、顶层文件观察、数据日志观察等窗口, 另外还有一些命令按钮与工作栏, 主要命令按钮和工作栏在图中加有注解, 其用途说明如下 (根据按钮与工作栏排列自左至右、由上到下说明):

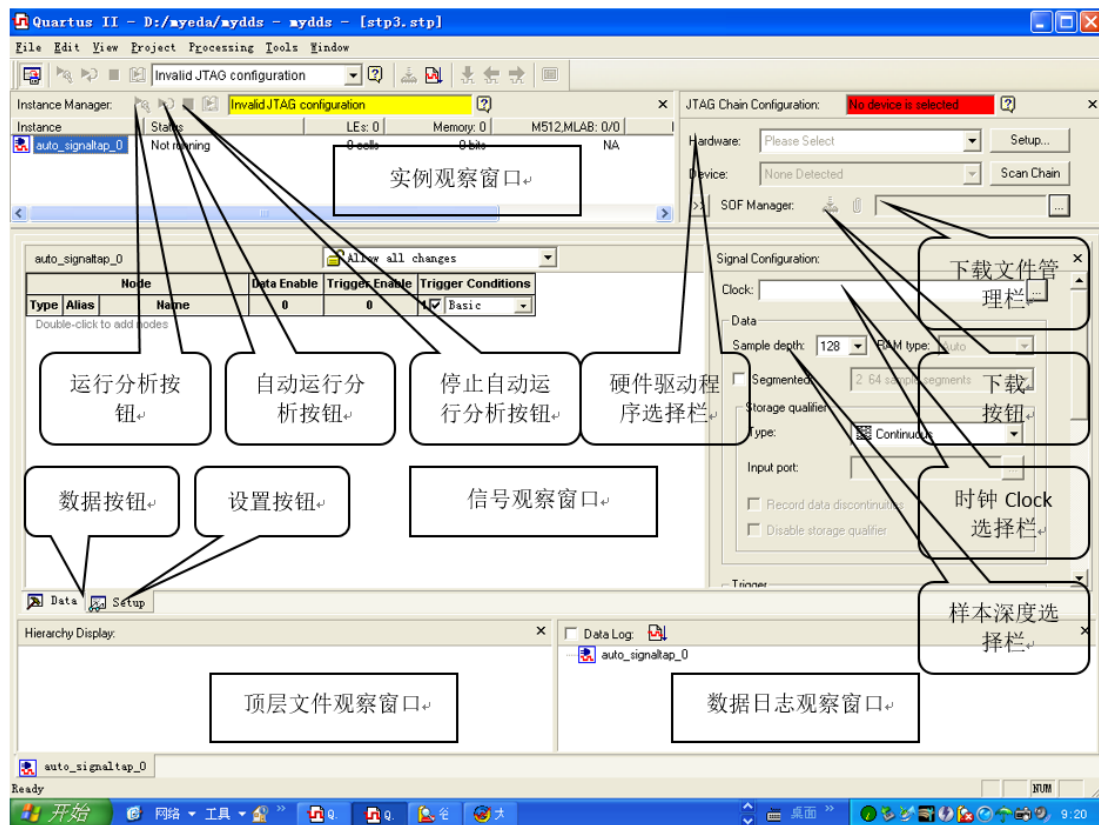


图 8-56 SignalTap II 编辑窗口

①运行分析 (Run Analysis) 按钮。在 SignalTap II 完成节点调入、参数设置、存盘、编译与下载后, 点击该按钮则运行一个样本深度结束, 并在数据窗口显示分析结果。

②自动运行分析（Autorun Analysis）按钮。该按钮有两种功能，其一是在完成 SignalTap II 的节点调入、参数设置与存盘后，点击此按钮则对工程进行编译。其二是完成下载后，点击此按钮开始自动运行分析，并在数据窗口实时显示分析结果。

③停止自动运行分析（**Stop Analysis**）按钮。在自动运行分析时，点击此按钮，结束自动运行分析，并在数据窗口显示结束时刻的分析结果。

④硬件驱动程序（Hardware）选择栏。该栏目用于选择目标芯片下载的硬件驱动程序。

⑤下载文件（Sof）管理（Sof Manager）栏。该栏目用选择目标芯片的下载程序。

⑥下载按钮。点击该按钮完成设计文件到目标芯片的下载。

⑦时钟（Clock）选择栏。该栏用于选择设计文件的时钟信号。

⑧样本深度（Sample Depth）选择栏。该栏目用于选择占用目标芯片中的嵌入式 RAM 的容量，从 0B（Byte，字节）到 128KB，选择容量越大，则存储的分析数据越多。例如，波形产生器（mydds）工程中的存储器容量为 256B，如果样本深度为 2KB，则可以存放（ $2 \times 1024\text{B} / 256\text{B} =$ ）8 个周期的波形。但样本深度的选择不能超过目标芯片中的嵌入式 RAM 的容量。例如，目标芯片是 Cyclone 系列的 EP1C6Q240C8，其内部嵌入式 RAM 的容量是 92160b（bit，位），则其容量为（ $92160 / 8 / 1024 =$ ）11.25KB。

⑨数据按钮（Data）。点击该按钮则打开分析数据窗口。

⑩设置按钮（Setup）。点击该按钮则打开设置窗口。

（2）调入节点信号

在实例观察窗口，默认的实例名为“auto_sigtap_0”，双击该实例名即可以更改，也可以保持默认。运行时该窗口还显示运行的状态、设计文件占用的逻辑单元数（Les）和占用的嵌入式 RAM 的位数（bit）。用鼠标双击信号观察窗口，弹出节点发现（Node Finder）对话框，如图 8.57 所示，在过滤对话框的 Filter 栏目中选择“SignalTap II Per-Synthesis”项后，点击“List”按钮，在 Nodes Found 栏目内列出了设计工程全部节点，（参照仿真方法）用鼠标选择设计电路需要观察的输出端口（如 q）后关闭此对话框，选中的节点出现在信号观察窗口中。

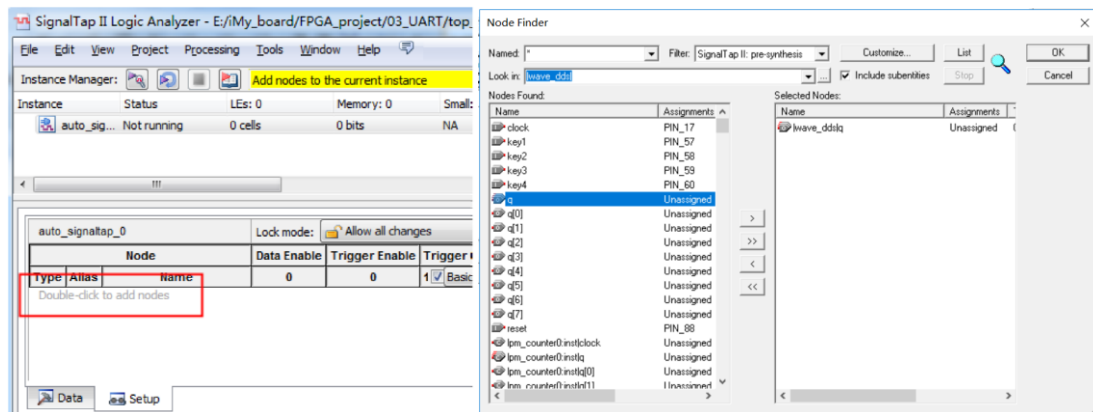


图 8-57 信号节点发现对话框

（3）参数设置

参数设置包含以下几个操作：

- ① 点击硬件驱动程序选择栏（Hardware）该栏目右边的“Setup”按钮，在对话框中选择编程下载的硬件驱动程序，选择“USB_Blaster”。

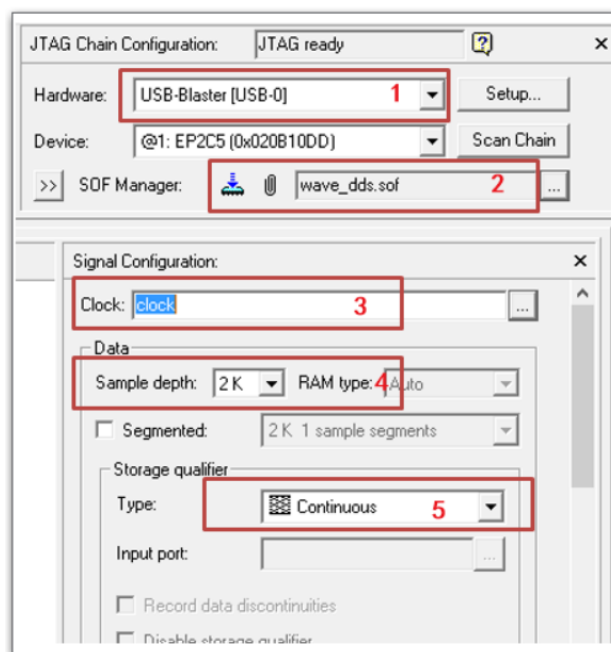


图 8-58 参数设置

② 点击下载文件管理栏（SofManager）右边的查阅按钮，在对话框中选择工程的下载文件（如 wave_dds.sof）。

③ 点击时钟（Clock）栏右边的查阅按钮，弹出节点发现者对话框（参见图 8-57 所示），在对话框中将设计工程文件的时钟信号选中（如 clock）。

④ 展开样本深度（Sample Depth）选择栏的下拉菜单，将样本深度选择为 2K（或其他深度）。

⑤ 选择触发方式，这里选择连续触发方式(Continuous)。

（4）文件存盘

完成上述的加入节点信号和参数设置操作后，执行 File→Save 命令，将 SignalTap II 文件存盘，默认的存盘文件名是“stp1.stp”，为了便于记忆，可以用“mydds_stp.stp”名字存盘。

（5）编译与下载

点击自动运行分析（Autorun Analysis）按钮，编译 SignalTap II 文件。编译完成后点击下载文管理（SofManager）栏中的下载按钮，完成设计工程文件到目标芯片的下载。

（6）运行分析

点击数据按钮，展开信号观察窗口。用鼠标右击被观察的信号名（如 q），弹出

如图 8-59 所示的选择信号显示模式的快捷菜单，在快捷菜单中选择“Bus Display Format”（总线显示方式）中的“Unsigned Line Chart”，将输出 q 的设置为无符号线形图显示模式。

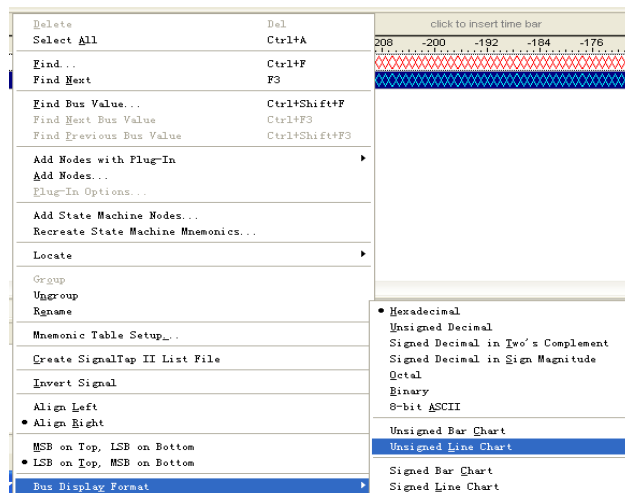
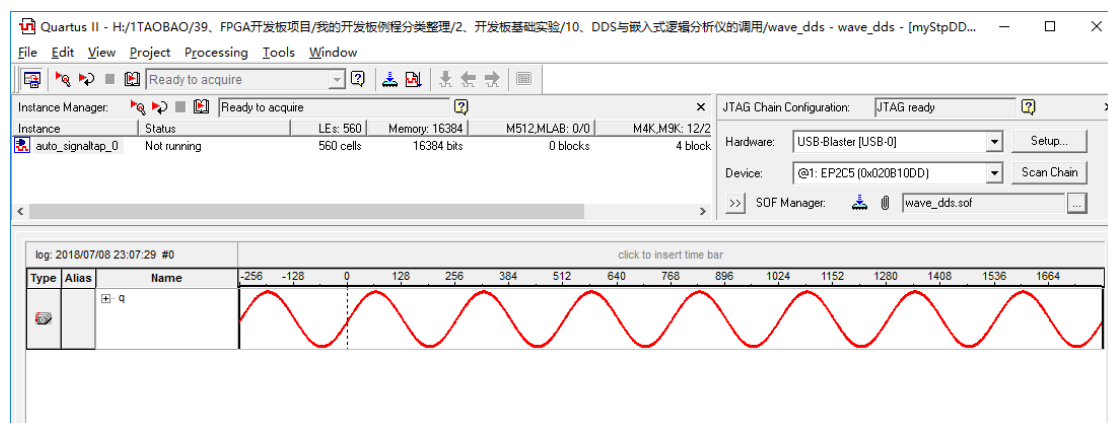
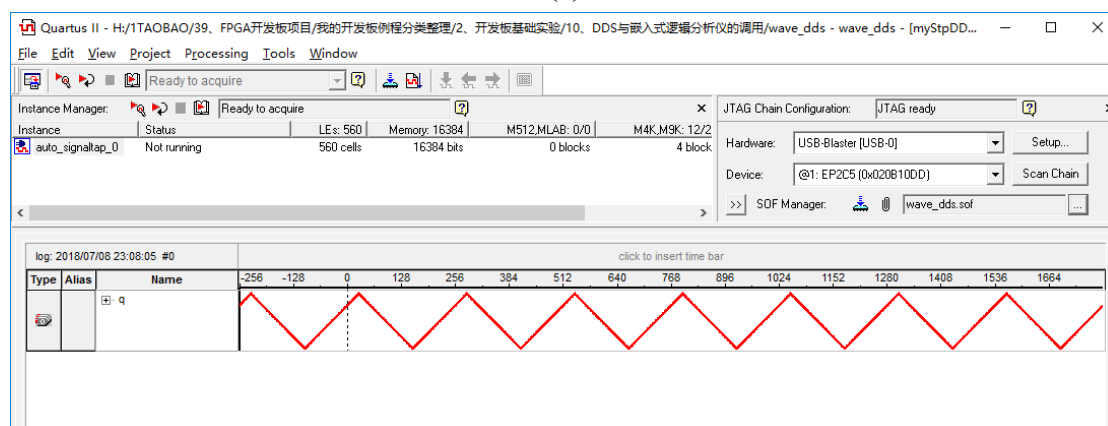


图 8-59 选择信号显示模式的快捷菜单

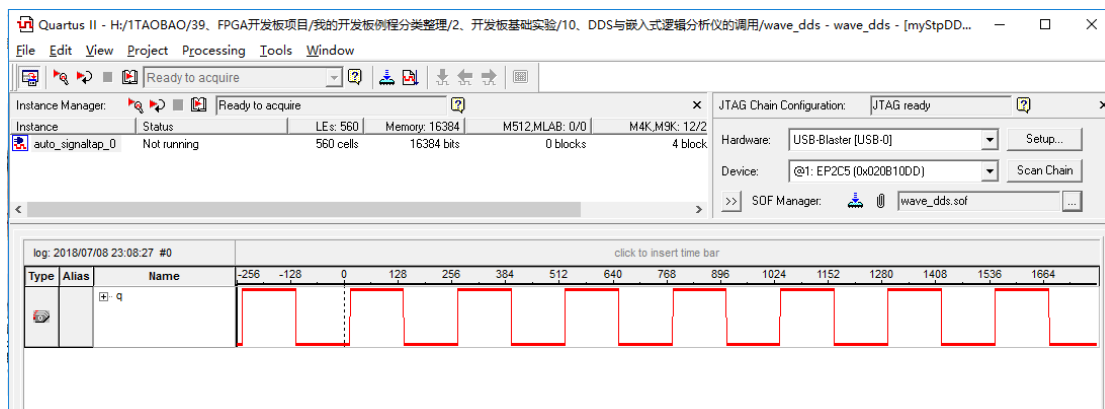
点击运行分析（Run Analysis）按钮或自动运行分析（Autorun Analysis）按钮，在信号观察窗口上可以见到波形发生器设计（mydds）的输出信号 q 的波形，通过按键可以切换波形，如图 8-60 所示，由于本例的样本深度为 2K，所以采样到的最大数据位 8 个周期。



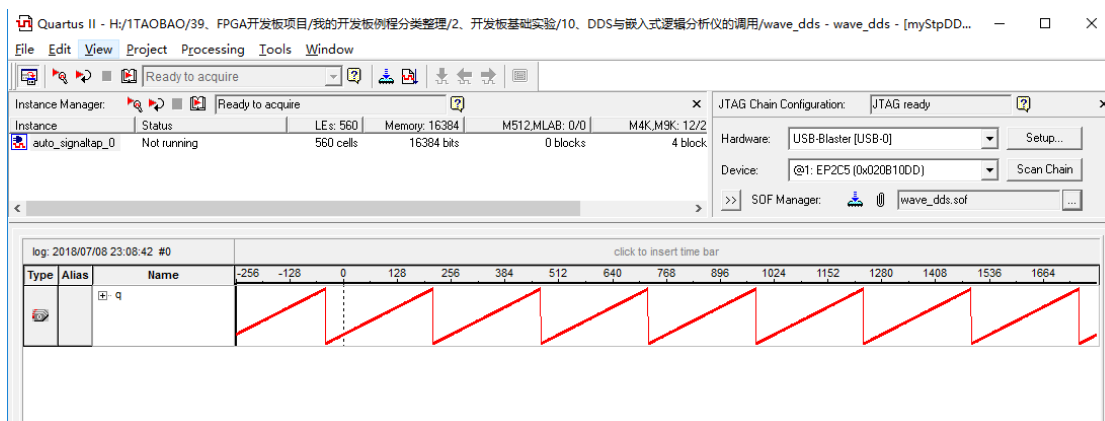
(a)



(b)



(c)



(d)

图 8-60 SignalTap II 采集波形回放实时显示

8.13 步进电机控制实验

1、步进电机相关知识原理

步进电机是一种将电脉冲转化为角位移的执行机构。通俗一点讲：当步进驱动器接收到一个脉冲信号，它就驱动步进电机按设定的方向转动一个固定的角度（及步进角）。您可以通过控制脉冲个数来控制角位移量，从而达到准确定位的目的；同时您可以通过控制脉冲频率来控制电机转动的速度和加速度，从而达到调速的目的。

实验使用的电机如图 8-60 所示，5V 减速步进电机（5 线 4 相），型号：28BYJ-48，其主要参数如下：

尺寸：28mm（直径）；

电压：5V；

步进角度：5.625 x 1/64；

减速比：1/64；

应用：适用于 51/AVR/ARM 等各种平台。



图 8-60 小型步进电机实物图

步进电机 28BYJ48 型四相八拍电机，电压为 DC5V—DC12V。当对步进电机施加一系列连续不断的控制脉冲时，它可以连续不断地转动。每一个脉冲信号对应步进电机的某一相或两相绕组的通电状态改变一次，也就对应转子转过一定的角度（一个步距角）。当通电状态的改变完成一个循环时，转子转过一个齿距。四相步进电机可以在不同的通电方式下运行，常见的通电方式有单（单相绕组通电）单四拍（A-B-C-D-A。。。），双（双相绕组通电）四拍（AB-BC-CD-DA-AB-。。。），四相八拍（A-AB-B-BC-C-CD-D-DA-A。。。）。

导线颜色	1 步	2 步	3 步	4 步
5 红	5V	5V	5V	5V
4 橙	-	+	+	+
3 黄	+	-	+	+
2 粉	+	+	-	+
1 蓝	+	+	+	-

图 8-61 步进电机单四拍控制表

2、实验设计要求

能通过按键控制步进电机正转、反转、加减速等控制，用 LED 指示相应状态。

3、电路原理图

由于 FPGA 接口驱动电流不够，需要通过 ULN2003 放大再连接到相应的电机接口，如图 8-62 步进电机驱动电路所示。FPGA 控制口接到 ULN2003 的 D0\D1\D2\D3 端口，控制输出口 Q0\Q1\Q2\Q3 的电平状态。

步进电机有 5 根接线，红线接电源 5V，橙色电线，黄色电线，粉色电线，蓝色电线，依次接入 J1 排针的对应端口。

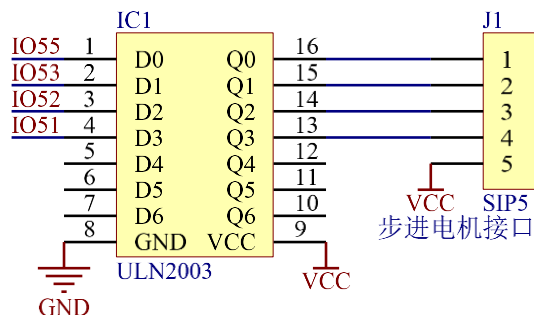


图 8-62 步进电机驱动电路

4、实验程序的设计

1) 设计思路

我们实验中所使用的步进电机为四相步进电机。转子小齿数为 64。系统中采用四路 I/O 进行并行控制，FPGA 直接发出多相脉冲信号，在通过功率放大后，进入步进电机的各相绕组。

步进电机控制设计包括几个模块：按键消抖模块、步进电机工作模式控制模块、步进电机驱动模块。

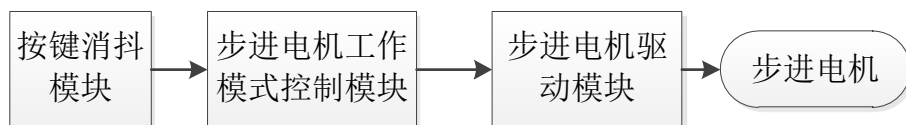


图 8-62 步进电机控制原理框图

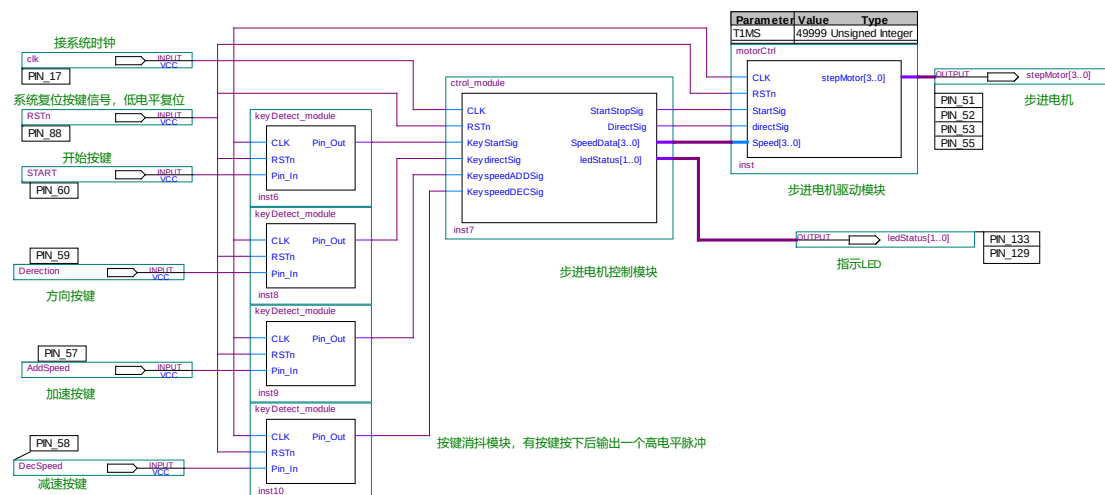


图 8-63 步进电机控制顶层设计原理图

步进电机控制顶层原理图如图 8-63 所示，输入信号有 CLK, RSTn, START, Derection, AddSpeed, DecSpeed；输出信号有步进电机 4 个控制端口 stepMotor，LED 状态指示端口 ledStatus。相关控制按键经过消抖模块后输出单脉冲信号，步

进电机控制模块识别控制信号后，输出对应的控制信号给步进电机驱动模块，包括启动停止信号、方向信号、速度信号以及 LED 状态指示；步进电机驱动模块根据控制信号驱动步进电机工作，本设计采用单四拍控制模式。

2)、实验程序

按键消抖模块程序前面章节已经给出，下面主要给出步电机控制模块程序和步进电机驱动模块程序。

a) 步电机控制模块 verilog 程序：

```

1. module ctrol_module
2. ( input CLK,
3.   input RSTn,
4.   input KeyStartSig,
5.   input KeydirectSig,
6.   input KeyspeedADDSig,
7.   input KeyspeedDECSig,
8.   output StartStopSig,
9.   output DirectSig,
10.  output[3:0] SpeedData,
11.  output[1:0] ledStatus
12. );
13. reg[3:0] i;
14. reg rStartStopSig;
15. reg rDirectSig;
16. reg[3:0] rSpeedData;
17. always @(posedge CLK or negedge RSTn)
18. begin
19.   if(!RSTn)
20.     begin
21.       i<=4'd0;
22.       rStartStopSig <= 1'b0;
23.       rDirectSig <= 1'b0;
24.       rSpeedData <= 4'd5;
25.     end
26.   else
27.     begin
28.       if(KeyStartSig)//启动停止按键信号
29.         rStartStopSig<=rStartStopSig+1'b1;
30.       else if(KeydirectSig)//方向按键信号
31.         rDirectSig<=rDirectSig+1'b1;
32.       else if(KeyspeedADDSig)//加速按键信号
33.         rSpeedData<=rSpeedData+4'd1;

```

```

34.     else if(KeyspeedDECSig)//减速按键信号
35.         rSpeedData<=rSpeedData-4'd1;
36.     end
37. end
38. assign StartStopSig=rStartStopSig;
39. assign DirectSig=rDirectSig;
40. assign SpeedData=rSpeedData;
41. assign ledStatus={rStartStopSig, rDirectSig};
42. endmodule

```

b) 步电机驱动模块 verilog 程序:

```

1. module motorCtrl
2. (
3.     CLK, RSTn, StartSig, directSig, Speed, stepMotor
4. );
5.     input CLK;
6.     input RSTn;
7.     input StartSig;
8.     input directSig;
9.     input [3:0] Speed;
10.    output [3:0] stepMotor;
11.    /*****
12.    parameter T1MS = 16'd49999;
13.    *****/
14.    reg[15:0] Count1;
15.    reg isCount;
16.    always @(posedge CLK or negedge RSTn)
17.    begin
18.        if(!RSTn)
19.            Count1 <= 16'd0;
20.        else if(isCount && Count1 == T1MS)//毫秒计数 Count1 + 1'b1;
21.            Count1 <= 16'd0;
22.        else if(isCount)
23.            Count1 <= Count1 + 1'b1;
24.        else if(!isCount)
25.            Count1 <= 16'd0;
26.    end
27.    /*****
28.    reg [7:0] Count_MS;
29.    always @(posedge CLK or negedge RSTn)
30.    begin

```

```

31.   if(!RSTn)
32.       Count_MS <= 8'd0;
33.   else if(isCount && Count1 == T1MS)//毫秒计数
34.       Count_MS <= Count_MS + 1'b1;
35.   else if(!isCount)
36.       Count_MS <= 8'd0;
37.   end
38.   /*****/
39.   reg [3:0]rstepMotor;
40.   reg[3:0]i;
41.   always @(posedge CLK or negedge RSTn)
42.   begin
43.       if(!RSTn)
44.       begin
45.           isCount <= 1'b0;
46.           rstepMotor <= 4'b0000;
47.           i <= 4'd0;
48.       end
49.   else if(StartSig==1'b1)// 慢速模式
50.       case(i)
51.           4'd0:
52.               if(directSig==1'b1)i<=4'd1;    //正转运行
53.               else if(directSig==1'b0)i<=4'd5; //反转运行
54.               else begin i=4'd0;rstepMotor <= 4'b0000;end //停止
55.           4'd1: //正转运行
56.               if(Count_MS == (Speed+1'b1))begin isCount<=1'b0; i <= i+1'b1;end
57.               else begin isCount <= 1'b1; rstepMotor <= 4'b0001;end//控制信号
58.           4'd2:
59.               if(Count_MS == Speed)begin isCount<=1'b0; i <= i+1'b1;end
60.               else begin isCount <= 1'b1; rstepMotor <= 4'b0010;end//控制信号
61.           4'd3:
62.               if(Count_MS == (Speed+1'b1))begin isCount<=1'b0; i <= i+1'b1;end
63.               else begin isCount <= 1'b1; rstepMotor <= 4'b0100;end//控制信号
64.           4'd4:
65.               if(Count_MS == Speed)begin isCount<=1'b0; i <= 4'd0;end
66.               else begin isCount <= 1'b1; rstepMotor <= 4'b1000;end//控制信号
67.
68.           4'd5: //反转运行
69.               if(Count_MS == (Speed+1'b1))begin isCount<=1'b0; i <= i+1'b1;end
70.               else begin isCount <= 1'b1; rstepMotor <= 4'b1000;end//控制信号
71.           4'd6:

```



```

72.         if(Count_MS == Speed)begin isCount<=1'b0; i <= i+1'b1;end
73.     else begin isCount <= 1'b1; rstepMotor <= 4'b0100;end//控制信号
74.     4'd7:
75.         if(Count_MS == (Speed+1'b1))begin isCount<=1'b0; i <= i+1'b1;end
76.     else begin isCount <= 1'b1; rstepMotor <= 4'b0010;end//控制信号
77.     4'd8:
78.         if(Count_MS == Speed)begin isCount<=1'b0; i <= 4'd0;end
79.     else begin isCount <= 1'b1; rstepMotor <= 4'b0001;end//控制信号
80.     default:
81.         begin isCount<=1'b0; i <= 4'd0;end
82.     endcase
83. end
84. /*****/
85. assign stepMotor = rstepMotor;
86. /*****/
87. endmodule

```

5、实验现象图

下载程序后，通过 FPGA 核心板上的 4 个独立按键 K1\K2\K3\K4\K5 控制步进电机，对应功能是加速、减速、方向、启动停止、复位。

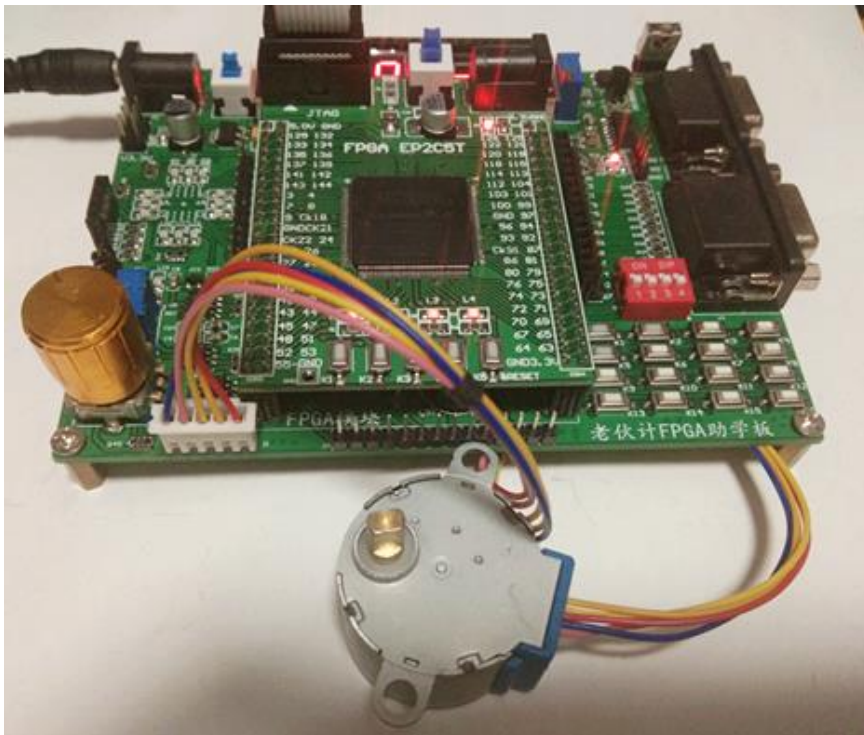


图 8-64 步进电机实验图

8.14 矩阵键盘控制实验

1、4X4 矩阵键盘相关原理

为了减少键盘与控制器接口时所占用 I/O 口线的数目，在键数较多时，通常都将键盘排列成行列矩阵式，行列式键盘又叫矩阵式键盘。用带有 I/O 口的线组成行列结构，按键设置在行列的交点上。例如用 2*2 的行列结构可以构成 4 个键的键盘，4*4 的行列结构可以构成有 16 个键的键盘。这样，当按键数量平方增长时，I/O 口线只是线性增长，这样就可以节省 I/O 口线。

FPGA 与矩阵键盘链接示意图如下图所示。

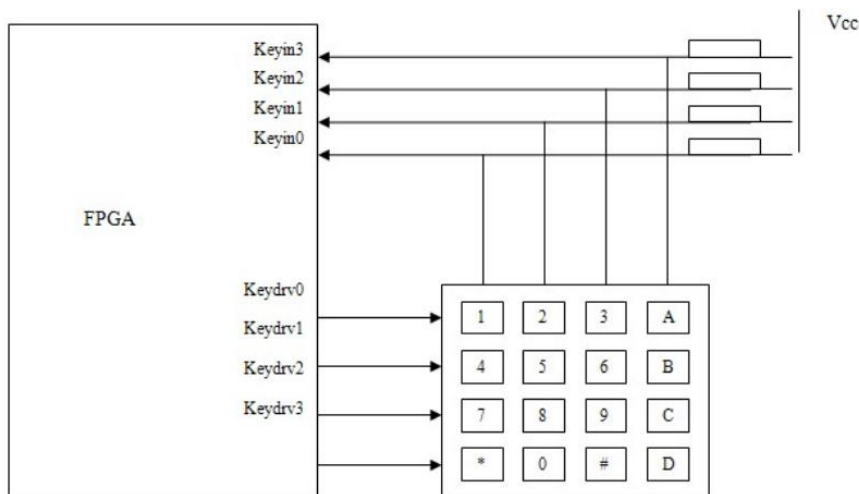


图 8-65 矩阵键盘链接示意图

按键设置在行列线交叉点，行、列线分别连接到按键开关的两端。列线通过上拉电阻接+5V 的电压，即列线的输出被钳位到高电平状态。行线与按键的一个引脚相连，列线与按键的另一个引脚相连。判断键盘中有无按键按下是通过行线送入扫描信号，然后从列线读取状态得到的。其方法是依次给行线送低电平，检查列线的输入。如果列线信号全为高电平，则代表低电平信号所在的行中无按键按下；如果列线有输入为低电平，则低电平信号所在的行和出现低电平的列的交点处有按键按下。

设行扫描信号为 keydrv3~keydrv0，列线按键输入信号 keyin3~keyin0 与按键位置的关系如下表所示。

由行列式键盘的原理可以知道，要正确地完成按键输入工作必须有按键扫描电路产生 keydrv3~keydrv0 信号。同时还必须有按键译码电路从 keydrv3~keydrv0 信号和 keyin3~keyin0 信号中译码出按键的键值。此外，一般还需要一个按键发生标志信号用于和其他模块接口，通知其它模块键盘上有按键动作发生，并可以从键盘模块中读取按键键值。

表 1 行扫描信号、列按键输入信号与按键位置的关系

keydrv3~keydrv0	keyin3~keyin0	对应的按键
1110	1110	1
	1101	2
	1011	3
	0111	A
1101	1110	4
	1101	5
	1011	6
	0111	B
1011	1110	7
	1101	8
	1011	9
	0111	C
0111	1110	*
	1101	0
	1011	#
	0111	D

行列式键盘电路的 FPGA 实现主要解决三个问题，一是如何检测是否有按键按下并防止采集到干扰信号；二是在按键闭合时如何防止抖动；三是如何判断为哪一个按键位动作，并对其进行译码。因此，为了解决这些问题，程序中使用不同的进程分别实现键盘扫描信号的产生、键盘去抖以及键盘的译码。

2、矩阵键盘实验设计要求

设计程序实现矩阵键盘上不同按键的识别，当按键按下后在数码管上显示对应的键值，用 LED 灯的状态指示按键的状态，形成方便调用的矩阵键盘程序模块。

3、矩阵键盘实验电路图

R7 是上拉电阻，接行输入信号 IO64\67\70\72，列输出信号是 IO63\65\69\71。

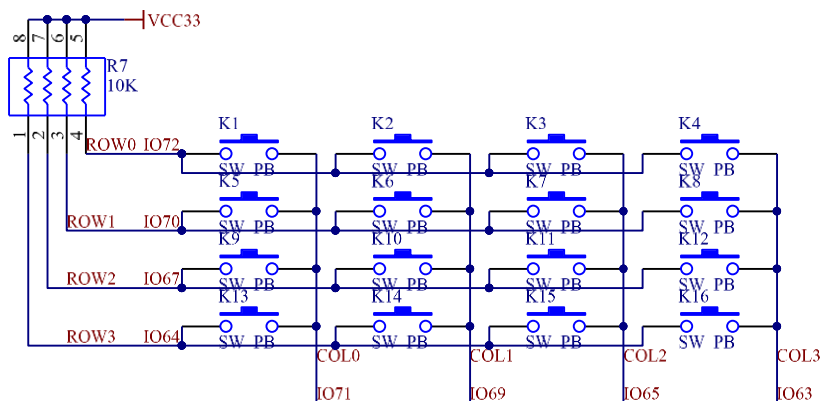


图 8-66 矩阵键盘实验电路图

4、实验程序设计

1) 设计思路

通常我们检测矩阵键盘中某一按键是否被按下，常用的方法是行扫描法。图 8-66 中一共有 8 条控制线，4 条行控制线 (ROW)，4 条列控制线 (COL)，而且可以看到 8 条控制线的初始状态都位高电平。那我们如何去检测某一个按键被

按下，并且还能找到其具体的位置？假如我们让 COL0=0，然后去逐行的进行扫描每一行，如果 ROW0=0 出现低电平通过 20ms 的延迟再次判断该行是否仍然为零，如果仍然为零，那么说明按键 0 被按下。其它按键的检测类似这种方法。例如：如果 COL=4’ b1011 时，当检测到 ROW=4’ b1011, 则说明“K11” 按键被按下；可能这块有些读者会说为啥不是“K6” 被按下？如果有读者提出这个疑惑的话，可能是信号的高低位没有确认好，再仔细思考思考。

总体设计框图如下图所示：

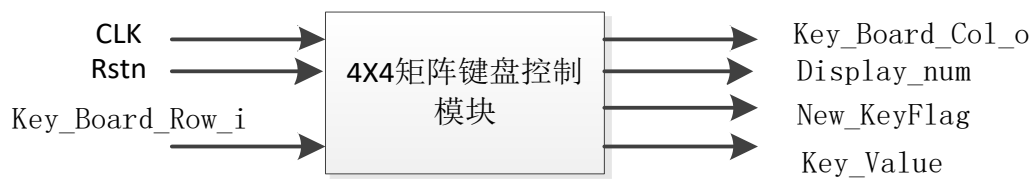


图 8-67 矩阵键盘程序设计总体设计框图

相关控制端口的功能如下表所示。

矩阵键盘控制端口列表

端口类型	端口名	描述
Input	Clk	系统时钟 50MHz
Input	Rst_n	系统复位，低有效
Input	Key_Board_Row_i	矩阵键盘行控制线，按键未按下为高电平
Output	Key_Board_Col_o	矩阵键盘列控制线，驱动列信号为低，实现按键状态扫描，
Output	Display_num	方便用设置参数显示的按键值，包含 4 次按键按下的键值
Output	Key_Value	输出键盘键值
Output	New_KeyFlag	按键检查成功标志信号，每当按键检测成功，产生一个时钟周期的高脉冲

2) 矩阵键盘控制实验顶层原理图如图 8-68 所示，包括矩阵键盘扫描模块和数码管显示模块，矩阵键盘扫描模块的控制端口与设计框图一致。

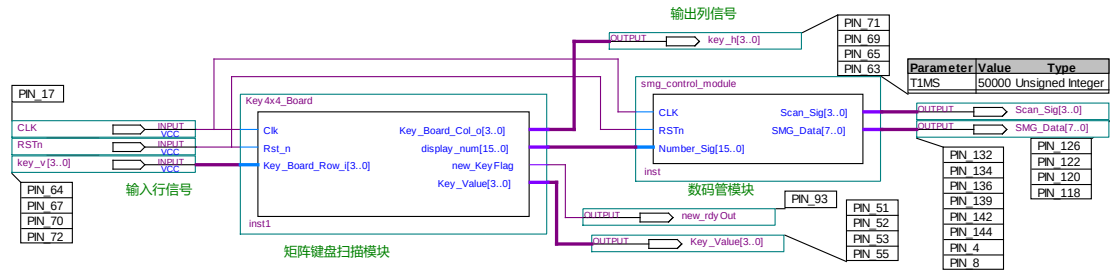


图 8-68

3) 实验程序

比较复杂的程序设计在进行程序设计之前,一般需要进行状态转移图的设计。可能会有读者说,这个东西有什么用处?给大家举个例子,“如何去建筑一座楼房”。在建造前肯定是工程师设计建筑的建造图纸,然后工人们按照图纸进行分工建造房屋。今天作者讲的状态转移图就像是工程师设计图纸没有图纸,直接编代码,肯定会来回修改多次。下面进入正题:

矩阵键盘编程的状态转移图包括如下几个状态:

1、IDLE 状态:主要是用来检测是否有按键被按下,无按键按下时,行控制线 Key_Board_Row_i 等于 15,有按键按下时 Key_Board_Row_i 不等于 15,则进入 P_FILTER 状态;

2、P_FILTER 状态:主要是进行按键消抖检测,判断是否有按键真实被按下,如果有按键真实被按下则跳转至下一个状态 READ_ROW_P 获取行状态的值,否则不跳转;

3、READ_ROW_P 状态:读取行状态并将存入的寄存器中 Key_Board_Row_r(ps:存入寄存器是因为按键按下的时间比较短暂,所以要将其状态缓存起来),并将第 0 列拉低,进入 SCAN_C0 状态进行行扫描;

4、SCAN_C0 状态:进行列扫描,来判断该列是否有按键被按下(ps:通俗地讲就是确定按键位置),判断行输入是否等于 15,如果不等于则说明该列有按键被按下,否则将第 1 列拉低,状态跳转到 SCAN_C1 状态进行行扫描;

5、SCAN_C1 状态:进行列扫描,来判断该列是否有按键被按下(ps:通俗地讲就是确定按键位置),判断行输入是否等于 15,如果不等于则说明该列有按键被按下,否则将第 2 列拉低,状态跳转到 SCAN_C3 状态进行行扫描;

6、SCAN_C2 状态:进行列扫描,来判断该列是否有按键被按下(ps:通俗地讲就是确定按键位置),判断行输入是否等于 15,如果不等于则说明该列有按键被按下,否则将第 3 列拉低,状态跳转到 SCAN_C3 状态进行行扫描;

7、SCAN_C3 状态:进行列扫描,来判断该列是否有按键被按下(ps:通俗地讲就是确定按键位置),判断行输入是否等于 15,如果不等于则说明该列有按键被按下,否则状态跳转到 PRESS_RESULT 状态输出扫描结果;

8、PRESS_RESULT 状态:输出扫描结果。这块给大家教一种比较特别的方法来确定整个键盘上只有一个按键被按下:4 位行控制线如果 4 位行线值加起来如果等于 3 则说明有一行被按下:

$(Key\_Board\_Row\_i[0] + Key\_Board\_Row\_i[1] + Key\_Board\_Row\_i[2] + Key\_Board\_Row\_i[3] = 3)$,同样如果 4 位列控制线值加起来如果等于 1 则说明有一列被按下:

(Col_Tmp[0]+Col_Tmp[1]+Col_Tmp[2]+Col_Tmp[3]=1)

通过这两个参数的限定，确保了只有一个按键被按下。如果不满足，则不产生检测成功标志信号。状态跳转到 WAIT_R 等待按键释放状态

9、WAIT_R 状态：等待按键释放，如果 Key_Board_Row_i=15 则说明按键被释放，状态跳转到释放消抖状态 R_FILTER

10、R_FILTER 状态：检测按键是否完全被释放如果释放则跳转到 READ_ROW_R 状态读取按键行输入状态，读取行的状态。

11、READ_ROW_R 状态：读取按键行输入状态。

通过这些状态，我们把一个按键按下，到释放的过程完美的进行了一次检测，最终将输出结果用 LED 灯的状态表示，同时用数码管显示出来。接下来，就是照着下面的状态转移图敲写程序。这下读者们可以尝试盖起自己的矩阵键盘“大厦”。

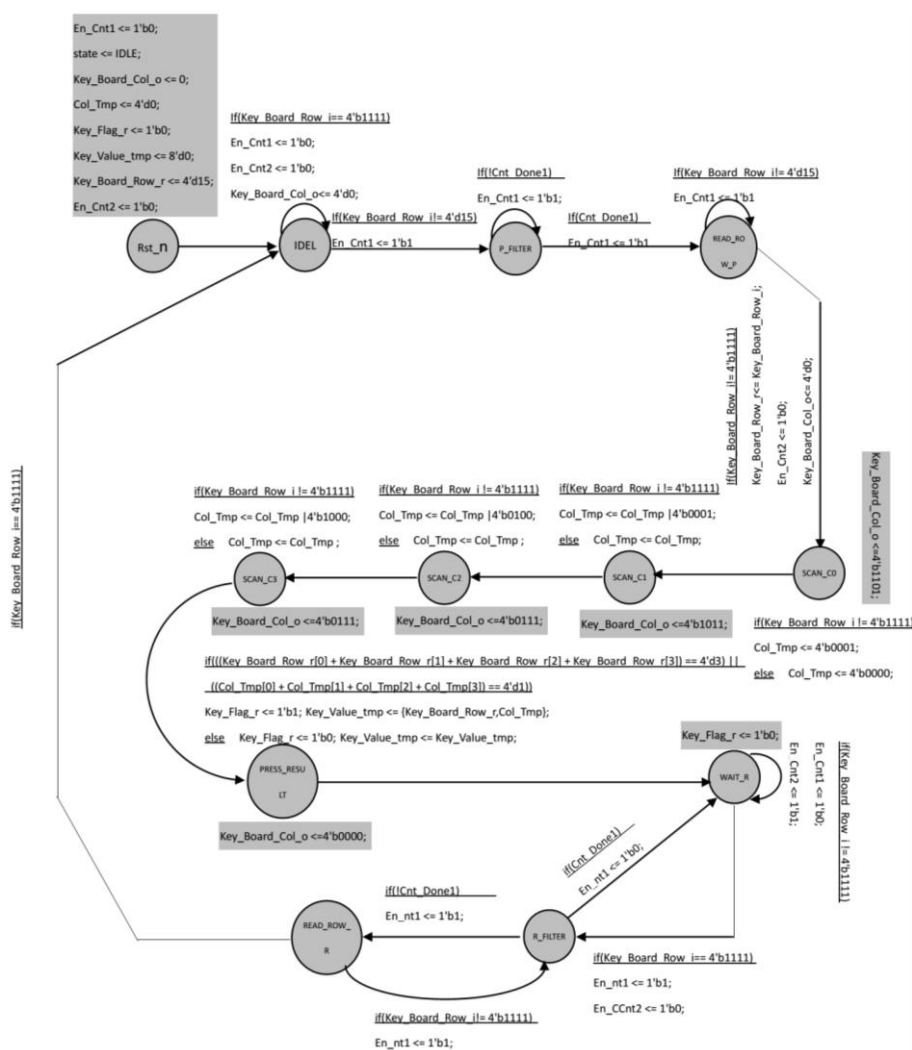


图 8-69 矩阵键盘编程的状态转移图

数码管显示程序前面章节已经给出,下面给出矩阵键盘扫描模块 verilog 程序:

```

1. Module Key4x4_Board
2. (Clk,Rst_n,Key_Board_Row_i,Key_Board_Col_o,display_num,new_KeyFlag,Key_Value);
3.     input Clk; //模块工作时钟,默认为50M
4.     input Rst_n; //模块复位,低电平复位
5.     input [3:0]Key_Board_Row_i; //矩阵键盘行输入
6.     output reg [3:0]Key_Board_Col_o; //矩阵键盘列输出
7.     output reg [15:0] display_num;
8.     output reg new_KeyFlag; //按键检查成功标志信号,高脉冲
9.     output reg [3:0]Key_Value; //键值
10.
11.     reg Key_Flag; //按键检查成功标志信号
12.     reg [19:0] counter1; //延时计数器
13.     reg En_Cnt1; //延时计数器使能控制信号,当进入延时消抖阶段时为高
14.     reg Cnt_Done1; //延时计数器延时完成(计满20ms)信号
15.
16.     reg [25:0] counter2; //连接间隔计数器
17.     reg En_Cnt2; //连接间隔计数器使能控制信号,当进入连接间隔阶段时为高
18.     reg Cnt_Done2; //连接间隔计数器延时完成(计满200ms)信号
19.
20.     reg [10:0]state; //状态寄存器,存储系统的状态
21.     reg [3:0]Key_Board_Row_r; //矩阵键盘行输入寄存器,存储行状态
22.     reg [3:0]Col_Tmp; //
23.     reg [7:0]Key_Value_tmp; //矩阵键盘扫描结果
24.     reg Key_Flag_r; //按键检查成功标志信号
25.
26.     localparam
27.         IDLE = 11'b000000000001, //空闲态,无按键按下
28.         P_FILTER = 11'b000000000010, //按下消抖状态
29.         READ_ROW_P = 11'b000000000100, //读取按下时矩阵键盘行状态
30.         SCAN_C0 = 11'b000000001000, //扫描矩阵键盘第0列(Col0)
31.         SCAN_C1 = 11'b000000010000, //扫描矩阵键盘第1列(Col1)
32.         SCAN_C2 = 11'b000000100000, //扫描矩阵键盘第2列(Col2)
33.         SCAN_C3 = 11'b000001000000, //扫描矩阵键盘第3列(Col3)
34.         PRESS_RESULT = 11'b000100000000, //获得扫描结果
35.         WAIT_R = 11'b001000000000, //等待释放信号到来
36.         R_FILTER = 11'b010000000000, //释放消抖
37.         READ_ROW_R = 11'b100000000000; //读取释放时矩阵键盘行状态
38.
39.     always@(posedge Clk or negedge Rst_n) // 延时计数器,计数以延时20ms
40.         if(!Rst_n)

```



```

41.         counter1 <= 20'd0;
42.     else if(En_Cnt1)begin
43.         if(counter1 == 20'd999999)
44.             counter1 <= 20'd0;
45.         else
46.             counter1 <= counter1 + 1'b1;
47.     end
48.     else
49.         counter1 <= 20'd0;
50.
51. //产生延时完成标志信号，（当延时20ms时产生一个计数完成信号Cnt_Done1）
52.     always@(posedge Clk or negedge Rst_n)
53.     if(!Rst_n)
54.         Cnt_Done1 <= 1'b0;
55.     else if(counter1 == 20'd999999)
56.         Cnt_Done1 <= 1'b1;
57.     else
58.         Cnt_Done1 <= 1'b0;
59.
60. // 连接间隔计数器，计数以延时200ms
61.     always@(posedge Clk or negedge Rst_n)
62.     if(!Rst_n)
63.         counter2 <= 26'd49_999_999; //初始时设定间隔为1s
64.     else if(En_Cnt2)begin
65.         if(counter2 == 26'd0)
66.             counter2 <= 26'd10_999_999; //启动间隔后每200ms发出一次间隔标志
67.         else
68.             counter2 <= counter2 - 1'b1;
69.     end
70.     else
71.         counter2 <= 26'd49_999_999;
72.
73. //产生连接间隔完成标志信号，（当延时20ms时产生一个计数完成信号Cnt_Done2）
74.     always@(posedge Clk or negedge Rst_n)
75.     if(!Rst_n)
76.         Cnt_Done2 <= 1'b0;
77.     else if(counter2 == 26'd0)
78.         Cnt_Done2 <= 1'b1;
79.     else
80.         Cnt_Done2 <= 1'b0;
81. //矩阵键盘扫描状态机

```



```

82.     always@(posedge Clk or negedge Rst_n)
83.     if(!Rst_n)begin
84.         En_Cnt1 <= 1'b0;
85.         state <= IDLE;
86.         //默认需要让列输出为0, 这样, 当有按键按下时, 行列接通, 行输入才能读到有低电平
87.         Key_Board_Col_o <= 4'b0000;
88.         Col_Tmp <= 4'd0;
89.         Key_Flag_r <= 1'b0;
90.         Key_Value_tmp <= 8'd0;
91.         Key_Board_Row_r <= 4'b1111;
92.     end
93.     else begin
94.         case(state)
95.         //空闲态, 持续判断是否有按键被按下 (按键按下时Key_Board_Row_i将不为全1)
96.         IDLE:
97.             if(Key_Board_Row_i != 4'b1111)begin //有按键按下
98.                 En_Cnt1 <= 1'b1;    //启动延時計数器
99.                 state <= P_FILTER; //跳转到按下消抖状态
100.            end
101.            else begin
102.                En_Cnt1 <= 1'b0;
103.                Key_Board_Col_o <= 4'b0000;
104.                state <= IDLE;
105.            end
106.            P_FILTER: //按下消抖状态
107.                if(Cnt_Done1) begin //20ms延时完成
108.                    En_Cnt1 <= 1'b0;    //停止延時計数器
109.                    state <= READ_ROW_P; //跳转到读取矩阵键盘行输入状态
110.                end
111.                else begin //20ms延时还没有完成
112.                    En_Cnt1 <= 1'b1;    //保持延時計数器继续计数
113.                    state <= P_FILTER;
114.                end
115.            READ_ROW_P: //读取矩阵键盘行输入状态
116.                if(Key_Board_Row_i != 4'b1111)begin
117.                    //行输入不全为1, 表明确实有按键按下
118.                    Key_Board_Row_r <= Key_Board_Row_i; //将输入状态存入寄存器
119.                    state <= SCAN_C0; //跳转到扫描矩阵键盘第0列 (Col0) 状态
120.                    Key_Board_Col_o <= 4'b1110; //将第0列列输出置0
121.                end
122.            else begin //否则不再执行扫描, 跳回空闲态

```

```

123.         state <= IDLE;
124.         Key_Board_Col_o <= 4'b0000;
125.     end
126.     SCAN_C0:    //扫描矩阵键盘第0列（Co10）状态
127.     begin
128.         state <= SCAN_C1; //下一个状态为扫描矩阵键盘第1列状态
129.         Key_Board_Col_o <= 4'b1101; //扫描第1列对应的行输出为1101
130.         if(Key_Board_Row_i != 4'b1111) //表明为当前列有按键按下
131.             Col_Tmp <= 4'b0001; //将列值存入列寄存器中
132.         else
133.             Col_Tmp <= 4'b0000;
134.     end
135.     SCAN_C1:    //扫描矩阵键盘第1列（Co11）状态
136.     begin
137.         state <= SCAN_C2; //下一个状态为扫描矩阵键盘第2列状态
138.         Key_Board_Col_o <= 4'b1011; //扫描第1列对应的行输出为1011
139.         if(Key_Board_Row_i != 4'b1111) //表明为当前列有按键按下
140.             Col_Tmp <= Col_Tmp | 4'b0010; //将列值存入列寄存器中
141.         else
142.             Col_Tmp <= Col_Tmp;
143.     end
144.     SCAN_C2:    //扫描矩阵键盘第2列（Co12）状态
145.     begin
146.         state <= SCAN_C3; //下一个状态为扫描矩阵键盘第3列状态
147.         Key_Board_Col_o <= 4'b0111; //扫描第1列对应的行输出为0111
148.         if(Key_Board_Row_i != 4'b1111) //表明为当前列有按键按下
149.             Col_Tmp <= Col_Tmp | 4'b0100; //将列值存入列寄存器中
150.         else
151.             Col_Tmp <= Col_Tmp;
152.     end
153.     SCAN_C3:    //扫描矩阵键盘第3列（Co13）状态
154.     begin
155.         state <= PRESS_RESULT; //下一个状态为得到扫描结果状态
156.         if(Key_Board_Row_i != 4'b1111) //表明为当前列有按键按下
157.             Col_Tmp <= Col_Tmp | 4'b1000; //将列值存入列寄存器中
158.         else
159.             Col_Tmp <= Col_Tmp;
160.     end
161.     PRESS_RESULT: //得到扫描结果状态
162.     begin
163.         state <= WAIT_R; //下一个状态为等待按键释放状态

```

```

164.          //让列输出全0，当有按键按下时，行列接通，行输入才能读到有低电平
165.          Key_Board_Col_o <= 4'b0000;
166.  /*4位行输入值相加为3，表明有且只有一个行输入为0，既保证只有一行中有按键被按
167.  下，4位列扫描结果相加为1，表明有且只有列中有按键被按下，通过这两个条件保证了一
168.  次按键中只有一个按键被按下时才检测有效*/
169.          if(((Key_Board_Row_r[0] + Key_Board_Row_r[1] +
170.              Key_Board_Row_r[2] + Key_Board_Row_r[3]) == 4'd3) &&
171.              ((Col_Tmp[0] + Col_Tmp[1] + Col_Tmp[2] + Col_Tmp[3])
172.               == 4'd1))begin
173.              Key_Flag_r <= 1'b1;//产生检测成功标志信号
174.              Key_Value_tmp <= {Key_Board_Row_r, Col_Tmp};
175.              //将行列结果组合得到按键扫描结果
176.          end
177.          //若不满足只有一个按键被按下，则不产生检测成功标志信号
178.          else begin
179.              Key_Flag_r <= 1'b0;
180.              Key_Value_tmp <= Key_Value_tmp;
181.          end
182.      end
183.      WAIT_R://等待按键释放状态
184.      begin
185.          Key_Flag_r <= 1'b0;//清零检测成功标志信号
186.          if(Key_Board_Row_i == 4'b1111)begin//表明按键被释放
187.              En_Cnt1 <= 1'b1;//启动延时计数器
188.              state <= R_FILTER;//进入释放消抖状态
189.              En_Cnt2 <= 1'b0;    //停止连按间隔计数器
190.          end
191.          else begin
192.              state <= WAIT_R;
193.              En_Cnt1 <= 1'b0;
194.              En_Cnt2 <= 1'b1;//启动连按间隔计数器
195.          end
196.      end
197.      R_FILTER:    //释放消抖状态
198.          if(Cnt_Done1) begin //延时时间到
199.              En_Cnt1 <= 1'b0;
200.              state <= READ_ROW_R;//跳转入读取按键行输入状态
201.          end
202.          else begin
203.              En_Cnt1 <= 1'b1;
204.              state <= R_FILTER;

```

```

205.             end
206.         READ_ROW_R://读取按键行输入状态（按键释放阶段）
207.         if(Key_Board_Row_i == 4'b1111) //表明已经稳定的释放了
208.             state <= IDLE; //跳转回空闲状态
209.         else begin //否则表明此次释放为抖动，回到释放消抖状态继续等待
210.             En_Cnt1 <= 1'b1;
211.             state <= R_FILTER;
212.         end
213.     default::;
214. endcase
215. end
216.
217. //将扫描结果译码输出
218. always@(posedge Clk or negedge Rst_n)
219.     if(!Rst_n)begin
220.         Key_Flag <= 1'd0;
221.         Key_Value <= 4'd0;
222.     end
223.     else begin
224.         //按键检查成功输出标志为扫描成功与连接间隔的逻辑或
225.         Key_Flag <= Key_Flag_r | Cnt_Done2;
226.         case(Key_Value_tmp)
227.             8'b1110_0001 : Key_Value <= 4'd12; //第0行，第0列
228.             8'b1110_0010 : Key_Value <= 4'd13; //第0行，第1列
229.             8'b1110_0100 : Key_Value <= 4'd14; //第0行，第2列
230.             8'b1110_1000 : Key_Value <= 4'd15; //第0行，第3列
231.
232.             8'b1101_0001 : Key_Value <= 4'd9; //第1行，第0列
233.             8'b1101_0010 : Key_Value <= 4'd0; //第1行，第1列
234.             8'b1101_0100 : Key_Value <= 4'd10; //第1行，第2列
235.             8'b1101_1000 : Key_Value <= 4'd11; //第1行，第3列
236.
237.             8'b1011_0001 : Key_Value <= 4'd5; //第2行，第0列
238.             8'b1011_0010 : Key_Value <= 4'd6; //第2行，第1列
239.             8'b1011_0100 : Key_Value <= 4'd7; //第2行，第2列
240.             8'b1011_1000 : Key_Value <= 4'd8; //第2行，第3列
241.
242.             8'b0111_0001 : Key_Value <= 4'd1; //第3行，第0列
243.             8'b0111_0010 : Key_Value <= 4'd2; //第3行，第1列
244.             8'b0111_0100 : Key_Value <= 4'd3; //第3行，第2列
245.             8'b0111_1000 : Key_Value <= 4'd4; //第3行，第3列

```

```

246.         default:begin Key_Value <= Key_Value;Key_Flag <= Key_Flag;end
247.     endcase
248. end
249. //-----
250. //产生最新键值,
251. always @(posedge Clk or negedge Rst_n)
252.     if(!Rst_n)begin display_num <= 16'h0000;end
253. //移动输出输出4个键值
254.     else if(Key_Flag)begin display_num <={display_num[11:0],Key_Value};
255.         new_KeyFlag<=1'b1;
256.         end//new_rdyOut提示下级模块键值有更新
257.     else new_KeyFlag<=1'b0;
258. endmodule

```

5、实验现象

开发板上 4 个数码管显示 4 次按键按下的键值，每次有新的按键按下，则显示的键值左移一位。核心板上的 4 个 LED 等用来显示当前键值的状态，比如最新的是 4，对应的二进制数是 0100，对应的 LED 灯状态是（亮-灭-亮-亮）。

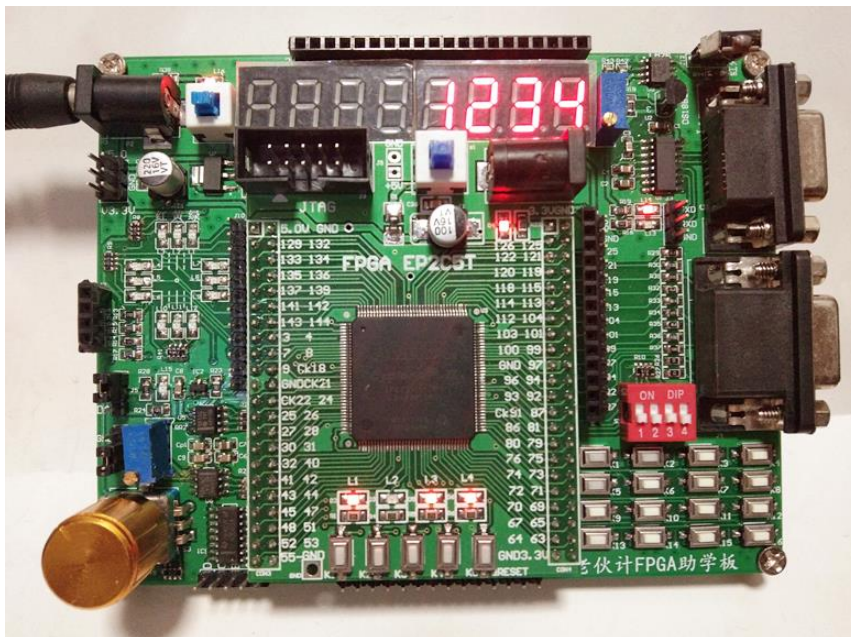


图 8-70 矩阵键盘实验图

8.15 旋转编码开关实验

1、旋转编码开关相关知识原理

旋转编码开关又称旋转编码器、数码电位器等，在电子产品中经常使用。旋转编码器具有 360 度旋转、调节范围广、调节速度可变、可控制、噪音小、寿命长的优点，旋转编码器功能在一定程度上类似于电位器，但其结构、工作原理和使用方法与普通电位器完全不同，在数字电路中的应用有很多，如改变音量的大小，电机的转速等。

旋转编码开关是一种可用于取代模拟电位器的器件，常用的旋转编码开关旋转一周输出 20 个脉冲，每个脉冲代表编码开关旋转了一定的角度，旋转编码开关有 3 个和 5 个引脚的，5 脚的比 3 脚的仅多 2 个按键引脚，另外 3 个引脚的功能与只有 3 个引脚的旋转编码开关功能相同，分别实现旋转时输出脉冲，在对编码器进行左旋和右旋时，其输出波形如图 8-71 (a) 和图 8-71 (b) 所示。

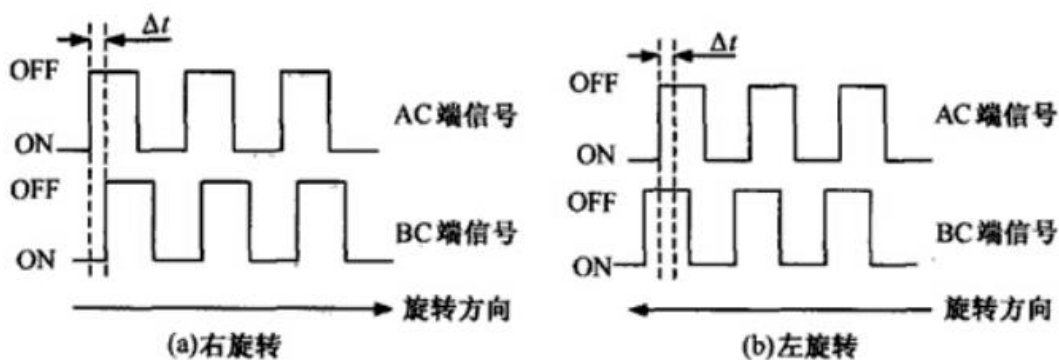


图 8-71 编码器左旋和右旋脉冲波形图

由图 8-71 分析可知，如果输出 AC 端信号由低电平变高电平是，这个时刻输出 BC 端出现是低电平，这时开关就是右旋转；如果输出 AC 端信号由低电平变高电平是，这个时刻输出 BC 端出现是高电平，这时开关就是左旋转；所以，在 FPGA 编程时只需要判断当输出 AC 端出现上升沿时，输出 BC 端当时的状态就可以判断出是左旋转或是右旋转了。

以 EC11 旋转编码开关为例：如图 8-72：三只脚：1 2 3 脚一般是中间 2 脚接地，1、3 脚上拉电阻后，当左转、右转旋转时，在 1、3 脚就有脉冲信号输出了。另外两只脚为按压开关，按下时导通，回复时断开。在编程时，左转和右转的判别是难点，用示波器观察这种开关左转和右转时两个输出脚的信号有个相位差。



图 8-72 EC11 旋转编码开关

2、实验设计内容

通过编码旋转开关控制开发板上的 12 个 LED 轮流点亮，左旋转时在旋转脉冲的作用下 LED 就逆时针轮流点亮，右旋转时在旋转脉冲的作用下 LED 就顺时针轮流点亮，没有旋转则 LED 保持状态。

编码开关按下可以改变 LED 流水灯工作方式，一是在旋转开关控制下手动控制 LED 轮流点亮，二是在时钟控制下 LED 自动轮流点亮。

3、实验原理图

需要使用到 12 个 LED 灯，旋转编码开关有 5 个引脚，2、5 脚接地，1、3、4 脚经上拉电阻接到 FPGA 控制 IO 口。

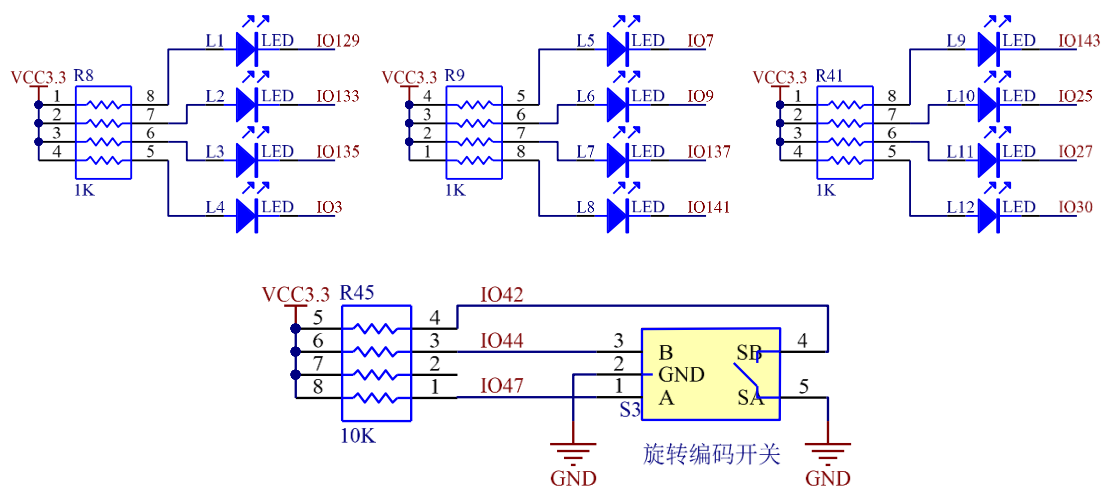


图 8-73 旋转编码开关实验原理图

4、实验程序的设计

1) 设计思路

旋转编码开关在操作和按键的操作有相似之处，都会存在抖动干扰，因此需要经过抖动处理后才能稳定可靠地工作，编程采用的总体设计框图如图 8-74 所示，AB 两端信号只需要做 A 端消抖即可，判断 A 端的边沿信号，再根据 B 端的高低电平确定是左旋转还是右旋转。

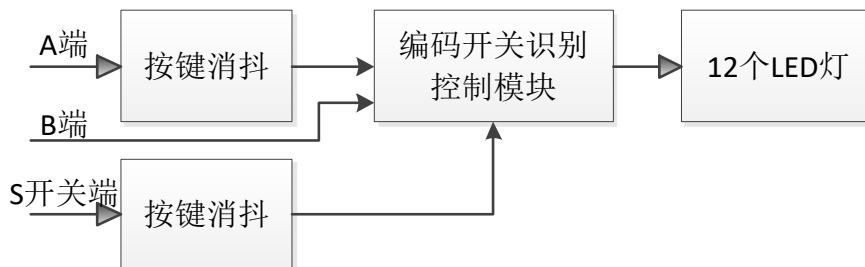


图 8-74 编码开关程序设计总体设计框图

2) 编码开关控制实验顶层原理图如图 8-75 所示，包括按键消抖模块和控制模块。

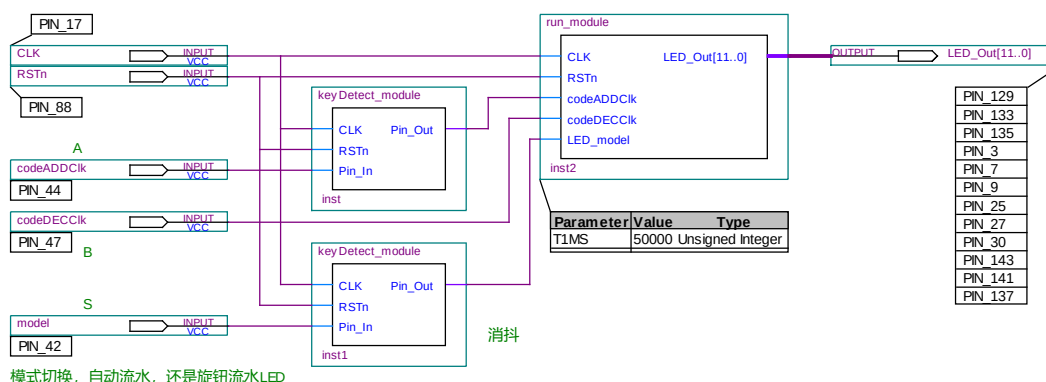


图 8-75 编码开关实验顶层原理图

3) 实验代码

按键消抖程序前面章节已经给出，下面给出选择编码开关识别判断的控制模块 verilog 程序：

```

1. module run_module
2. (
3.     CLK, RSTn, codeADDClk, codeDECClk, LED_model, LED_Out
4. );
5. input CLK;
6. input RSTn;
7. input codeADDClk; //A端口
8. input codeDECClk; //B端口
9. input LED_model; //S端口
10. output [11:0] LED_Out;
11. /*****

```



```

12. parameter T1MS = 16'd50_000;
13. /*****
14. reg[15:0]Count1;
15. always @(posedge CLK or negedge RSTn)
16.     if(!RSTn)
17.         Count1 <= 16'd0;
18.     else if(Count1 == T1MS)//1ms计时
19.         Count1 <= 16'd0;
20.     else
21.         Count1 <=Count1 +1'b1;
22. /*****
23. reg[9:0]Count_MS;
24. always @(posedge CLK or negedge RSTn)
25.     if(!RSTn)
26.         Count_MS <= 10'd0;
27.     else if(Count_MS == 10'd500)//500ms
28.         Count_MS <= 10'd0;
29.     else if(Count1 == T1MS)
30.         Count_MS <=Count_MS +1'b1;
31. /*****
32. reg[11:0] rLED_Out;
33. always @(posedge CLK or negedge RSTn)
34. begin
35.     if(!RSTn)
36.         rLED_Out <= 12'b111111111110;
37.     else if( model) //手动旋转控制LED流水灯
38.     begin
39.         if(codeDECClk==1'b1 && codeADDClk) //判断旋转方向，右旋
40.         begin
41.             rLED_Out <= {rLED_Out[10:0],rLED_Out[11]};
42.         end
43.     else if(codeDECClk==1'b0 && codeADDClk) //判断旋转方向，左旋
44.     begin
45.         rLED_Out <= {rLED_Out[0],rLED_Out[11:1]};
46.     end
47. end
48. else
49. begin
50.     if(Count_MS == 10'd500)//自动旋转控制LED流水灯
51.         rLED_Out <= {rLED_Out[10:0],rLED_Out[11]};
52. end

```

```

53.
54. end
55. reg model;
56. always @(posedge CLK or negedge RSTn)
57. begin
58.     if(!RSTn)
59.         model <= 1'b0;
60.     else if(LED_model)
61.         begin
62.             model <= ~ model;//工作模式切换
63.         end
64. end
65. /*****/
66. assign LED_Out =rLED_Out;
67. endmodule

```

5、实验现象

开发板上左下角黄色旋钮就是旋转编码开关，下载程序后旋转旋钮就可以看到LED灯随旋钮转动的变化，旋钮旋转输出一个脉冲信号，流水点亮下一个LED，按下旋钮可以改变手动-自动工作模式。

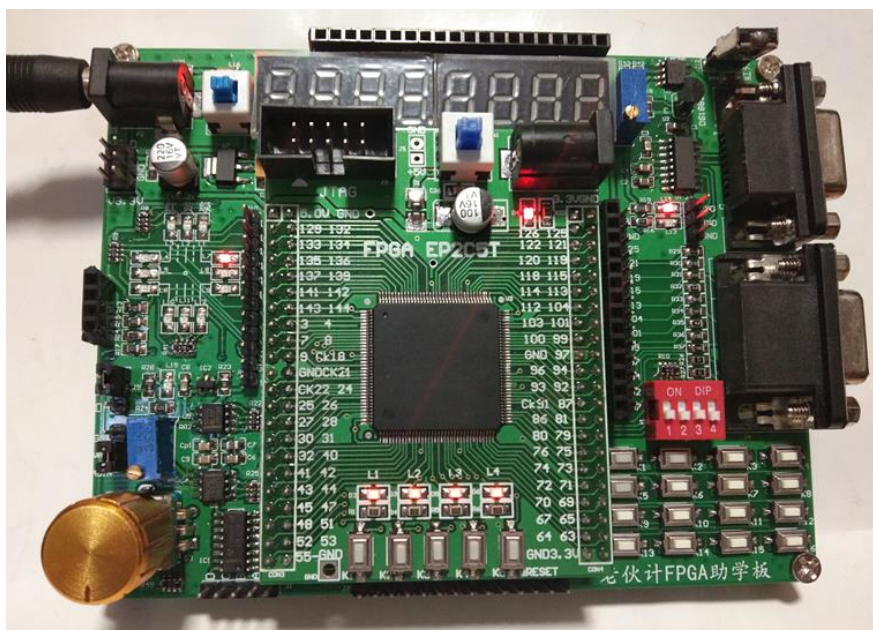


图 8-76 旋转编码开关实验图

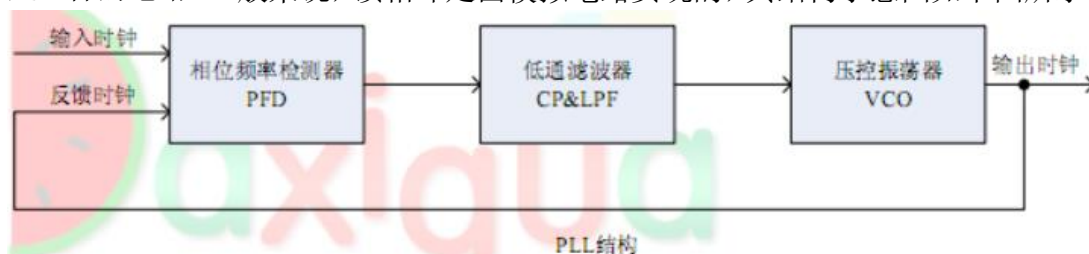
第九章 宏功能模块的使用

9.1 PLL 的使用

FPGA 中 PLL 的使用

一、PLL 简介

随着系统时钟频率逐步提升，I/O 性能要求也越来越高。在内部逻辑实现时，往往需要多个频率和相位的时钟，于是在 FPGA 内部出现了一些时钟管理元件，PLL 就是这样的电路。一般来说，锁相环是由模拟电路实现的，其结构示意图如下图所示。



PLL 工作原理：压控振荡器（VCO）通过自振输出一个时钟，同时反馈给输入端的相位频率检测器（PFD），PFD 根据比较输入时钟和反馈时钟的相位来判断 VCO 输出的快慢，同时输出 Pump-up 和 Pump-down 信号给环路低通滤波器（LPF），LPF 把这些信号转换成电压信号，再用来控制 VCO 的输出频率，当 PFD 检测到输入时钟和反馈时钟边沿对齐时，锁相环就锁定了。

二、下面我们来简单使用 FPGA 的 PLL 宏模块。

例子：我们开发板的输入时钟频率是 50M，但如果我们想等到一个 100M 的稳定时钟，我们可调用 PLL 宏模块来产生 2 倍频时钟（ $100\text{M}=2\times 50\text{M}$ ）；

先写 Verilog.v 程序

```

module PLL (rst,clk,led);
input clk,rst;
output led;
//reg rst;
wire c0;
//-----调用 PLL
PLL_100M PLL_ctrl_inst (
.inclk0 ( clk ),// inclk0 接的是时钟
.c0 ( c0 )// c0 是输出你想要分频的时钟信号
);
reg [28:0] cnt;
  
```

```
always @(posedge c0 or negedge rst)
begin
if(!rst)
cnt<=0;
else
cnt<=cnt+1'b1;
end
assign led=cnt[26];
endmodule
```

大家可能还看不懂红色字体的意思，这是一个实例化，就是说用户自己定义或者是直接引用软件自带的模块（类似于 C 语言的函数调用）。

下面我们来看看怎么配置 PLL 宏模块。

PLL 配置详细说明：

PLL 的配置需求，假定设计者已经新建了一个工程，然后需要配置一个 PLL。该 PLL 的输入时钟为 FPGA 外部的 50MHz 晶振，希望得到一个 100MHz（输入时钟的 2 倍频）的系统时钟供 FPGA 内部使用。该 PLL 的输入输出接口如表 1 所示。

表 1 PLL 的接口定义

信号名	方向	功能描述
inclk0	input	PLL 输入时钟
areset	input	PLL 复位信号，高电平有效
c0	output	PLL 输出时钟
locked	output	该信号用于指示 PLL 处理后的时钟已经稳定输出，高有效

PLL 的配置步骤

① 如图 1 所示，在 Quartus II 的菜单栏选择“Tools—>MegaWizardPlug-In Manager,”。



图 1 选择 MegaWizard

② 如图2所示,使用默认选项“Create a new custom megafunction variation”,点击“Next>”。

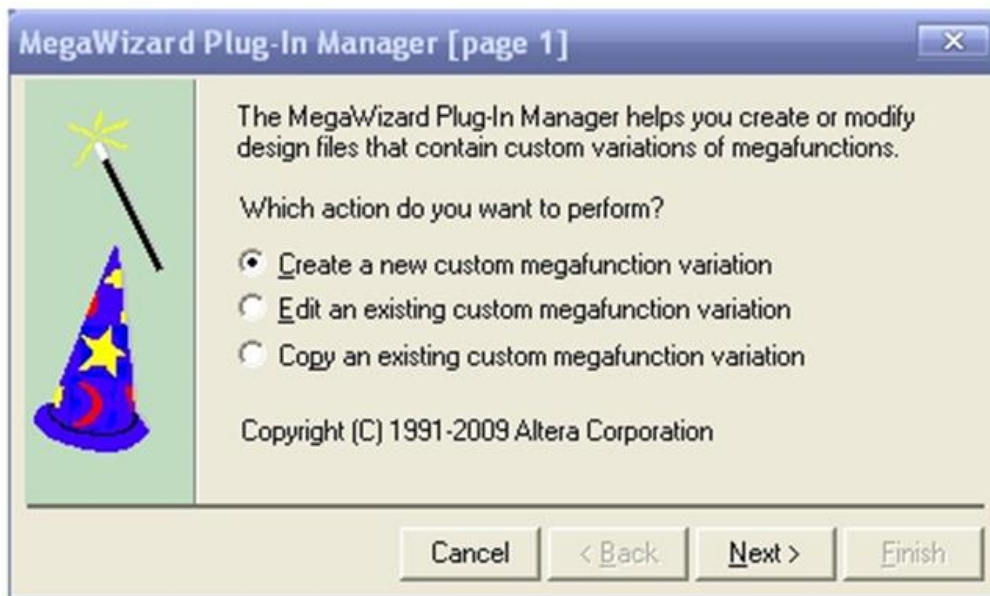


图 2

③如图 3 所示,进行以下配置:

◆在“Select a megafunction from the list below”窗口内打开“I/O”下拉框，选择“ALTPLL”。

◆在“Which type of output file do you want to create?”下选择“Verilog HDL”，这是配置的 PLL 内核使用的语言，一般选择此项。

◆在“What name do you want for the output file?”里默认会出现当前设计的工程路径，需要设计者在最后面手动输入例化的 PLL 的名字，这里输入了“PLL_100M”。

完成以上配置，点击“Next>”

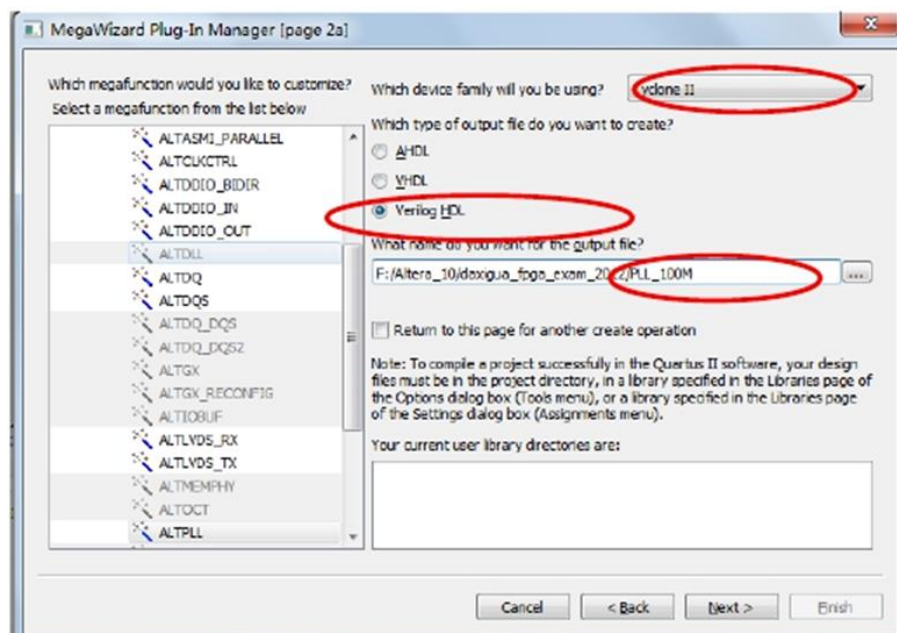
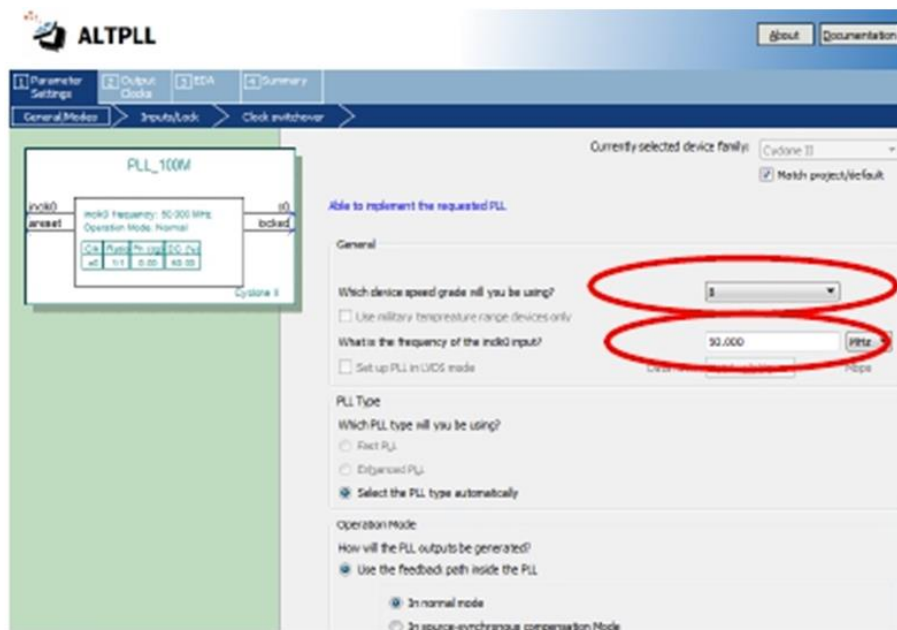


图 3 新建 PLL

④如图 4 所示，进行以下配置：

◆在“General”一栏内的“Which device speed grade will you be using?”选则该工程所使用器件的速度等级，我们的 FPGA 速度等级是 8。

◆在“What is frequency of the inclock0 input?”内选择 PLL 输入时钟的频率（50M）。其他选项使用默认即可。点击“Next>”。

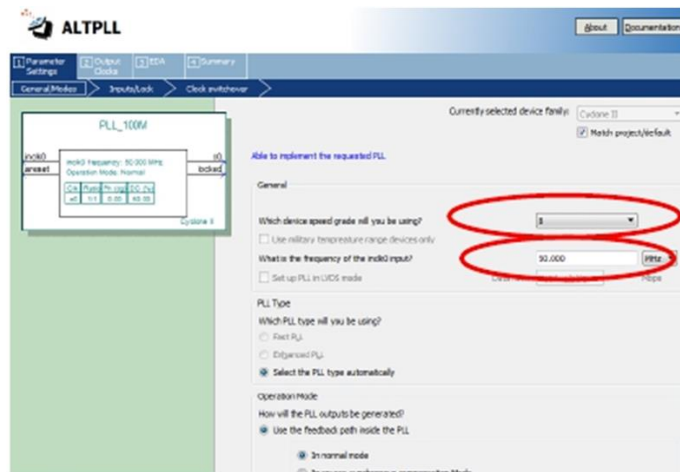


图 4 输入时钟配置

⑤如图 5 所示，配置如下：

◆在“Option input”一栏内不勾选“Creat an ‘areset’ input to asynchronously reset the PLL”（这里是复位输入，我们先不用）。

◆在“Lock output”中不勾选“Creat ‘locked’ output”。（这里是时钟稳定输出信号，我们先不用）其他选项使用默认即可。点击“Next>”。

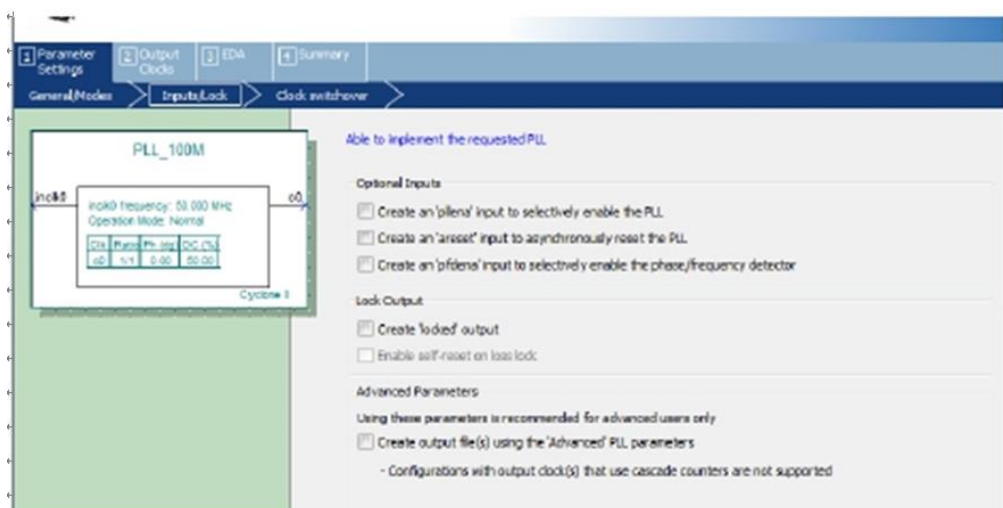


图 5 配置控制信号

⑥配置输出时钟 c0 相关参数，如图 6 所示。

❖设计者可以在“Enter output clock frequency?”后面输入希望得到的 PLL 输出时钟的频率。

设计者也可以在“Enter output clock parameter?”后面设置相应的输出时钟和输入时钟的频率关系。“Clock Multiplication factor”后输入倍频系数，

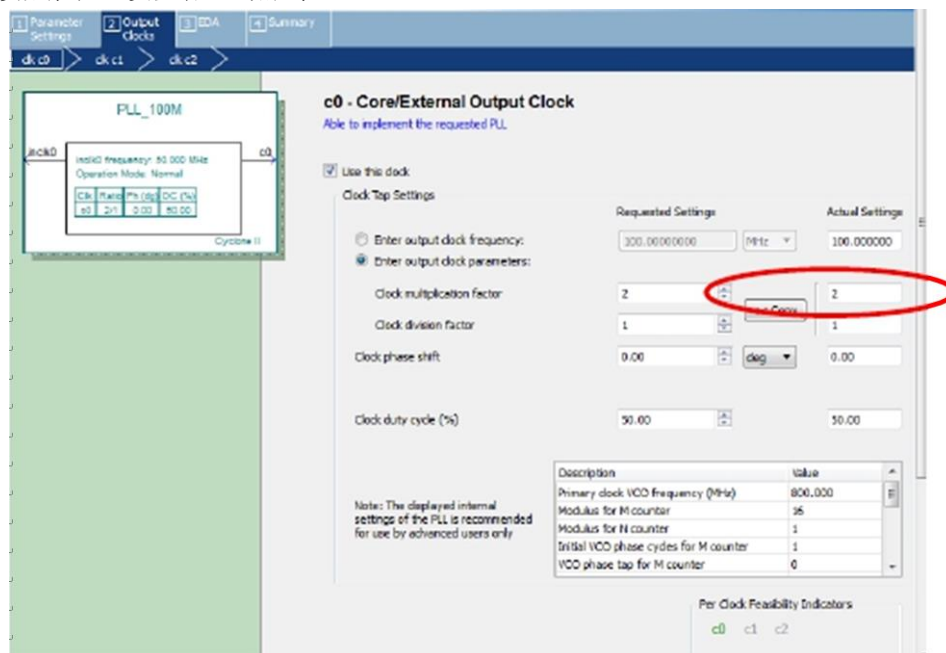
“Clock division factor”后输入分频系数，二者决定了输出时钟频率。

❖在“Clock phase shift”中可以设置相位偏移。

❖在“Clock duty cycle”中可以设置输出时钟占空比。

（设定最后看看是否输出 100M）

按照图 6 设置后，点击“Next>”。



⑦“clk c1”选项是可选的，用户需要第二个输出时钟时可以开启该输出时钟，相应勾选“Use the clock”后和上一步类似进行配置即可。（我们这里只用一个就可以了）图 7。

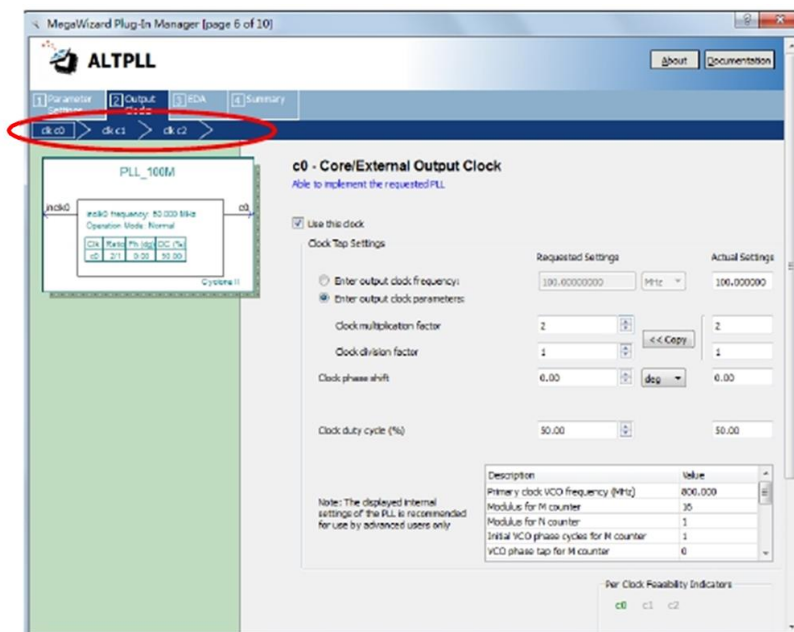
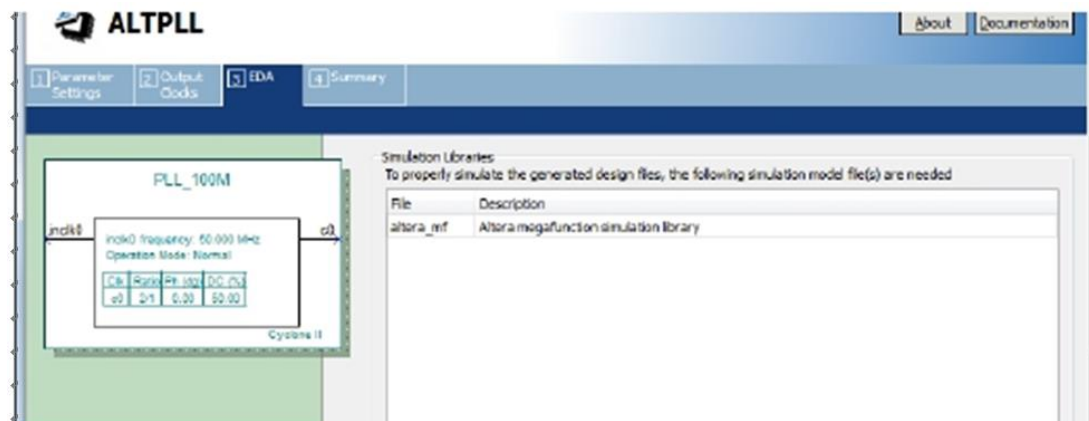


图 7 配置输出时钟 c1

8 如图 8 所示，“EDA”中列了用户在对例化了 PLL 模块的工程仿真时，需要添加的仿真库文件，用户可以到 Quartus II 安装文件夹下可以找到。点击“Next>”。



9 如图 9 所示，“Summary”中罗列了该 PLL 核最终的输出文件。对主要的一些输出文

件说明如下：

◆ PLL_100M.v，字面翻译是“变异文件”，是 PLL 内部的控制 IP 核。

◆ PLL_100M_inst.v 是一个模板的例化文件，用户可以直接复制这个文件里的例化来用。

◆ PLL_100M_wave.jpg 里是用户所配置的 PLL 的波形示例，勾选后可以在工程目录下找到，大家可以就我们的工程去看看波形是否符合预定的要求。或者用它和仿真后的波形对比一下，它们应该是一致的。

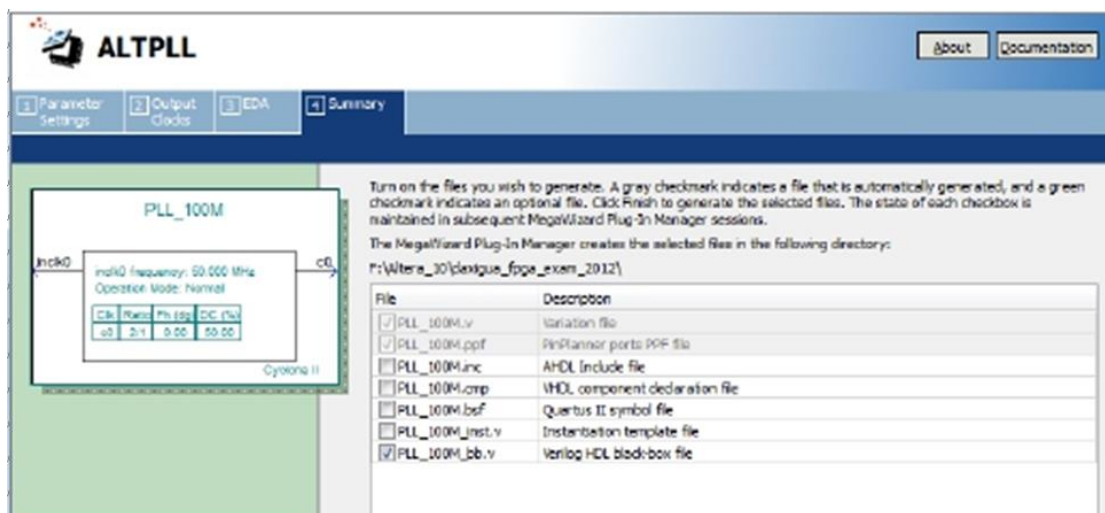


图 10 输出文件

PLL 的例化

PLL 配置完成后，需要将 PLL 例化到工程中。

最后，设计者完善代码的其他部分，编译工程即可。

9.2 ROM 的使用

具体请参考第九章 9.12.2 节

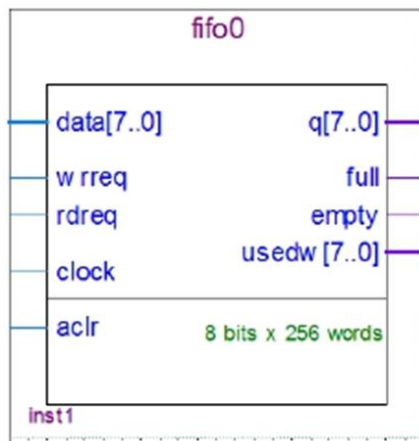
9.3 FIFO 的使用

FPGA 中的 FIFO 的使用

一、FIFO 简介

FIFO 是英文 First In First Out 的缩写，是一种先进先出的数据缓存器，他与普通存储器的区别是没有外部读写地址线，这样使用起来非常简单，但缺点就是只能顺序写入数据，顺序的读出数据，其数据地址由内部读写指针自动加 1 完成，不能像普通存储器那样可以由地址线决定读取或写入某个指定的地址。

二、下面以一个简单的例子来说明 FIFO 的使用。如图



输入信号：时钟信号 clock;

异步清零端 aclr;

写信号 wrreq;

读信号 rdreq;

数据输入端 data[7..0];

输出信号：数据读出端 q[7..0];

存储器为空信号 empty;

存储器为满信号 full.

接下来我们要设计一个从数据输入端输入数字 0~9 十个数，并通过仿真验证输出。（这里我们不用清零端）。

先写 Verilog.v 程序

```
module FIFO(clk,reset,dout);
    //----输入端口
    input clk,reset;
    //----输出端口
    output [7:0]dout;
    //----寄存器定义
    reg [7:0]data_buf;//数据寄存器
    reg [1:0] wr_buf;//读写寄存器
    //----信号线定义
    wire empty,full;
    //----FIFO 宏模块实例化
    FIFO_EXAM fi(
        .clock(clk),
        .data(data_buf),
        .rdreq(wr_buf[0]),
        .wrreq(wr_buf[1]),
        .empty(empty),
        .full(full),
```

```
.q(dout)
);
//---时序控制部分
always@(posedge clk or negedge reset)
begin
  if(!reset) begin //系统复位
    data_buf[7:0]<=8'b0; //数据寄存器清零
    wr_buf[1:0]<=2'b0; //读写寄存器清零
  end
  else
  begin
    if((data_buf<8'h9))
    begin
      data_buf<=data_buf+1'b1;
      if(!full) wr_buf[1]<=1'b1;
      else wr_buf[1]<=1'b0;
      if(!empty)wr_buf[0]<=1'b1;
      else wr_buf[0]<=1'b0;
    end
    else
    begin
      data_buf<=8'h0;
    end
  end

end
endmodule
```

FIFO 配置说明

FIFO 配置需求

假定设计者已经新建了一个工程，然后需要配置一个 FIFO。该 FIFO 的引脚为：

输入信号：	时钟信号 clock
	写信号 wrreq
	读信号 rdreq
	数据输入端 data[7..0]
输出信号：	数据读出端 q[7..0]
	存储器为空信号 empty
	存储器为满信号 full

FIFO 的配置步骤:

① 如图 1 所示, 在 Quartus II 的菜单栏选择 “Tools—>MegaWizard Plug-In Manager,, ”。



图 1 选择 MegaWizard

② 如图 2 所示, 使用默认选项 “Create a new custom megafunction variation”,
点击 “Next>”。

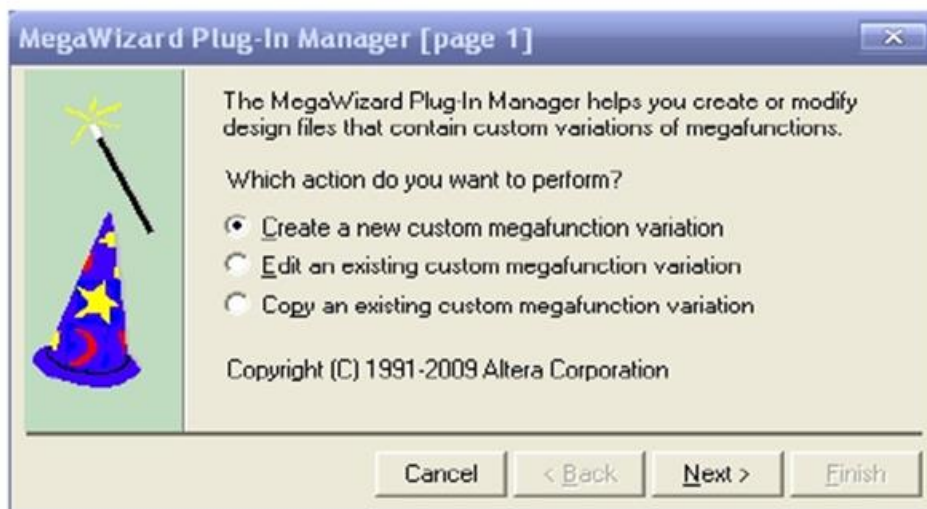


图 2

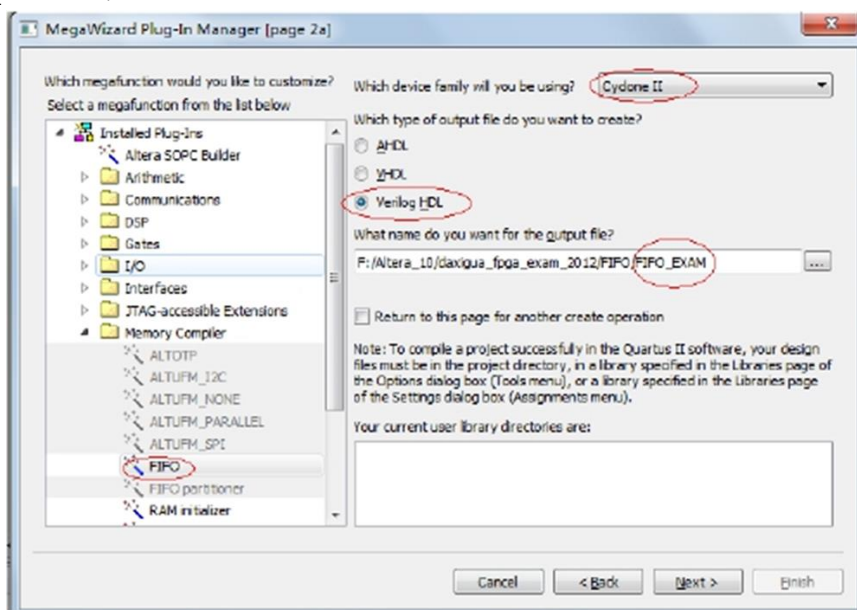
③ 如图 3 所示，进行以下配置：

◆ 在 “Select a megafunction from the list below” 窗口内打开 “Memory Compiler”

选择 “FIFO”

◆ 在 “Which type of output file do you want to create?” 下选择 “Verilog HDL”，这是配置的 FIFO 的语言，一般选择此项。

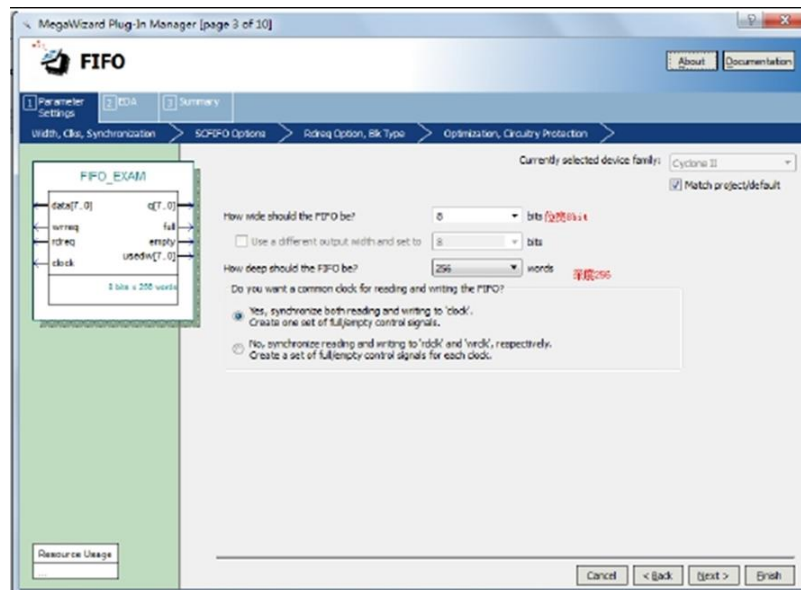
◆ 在 “What name do you want for the output file?” 里默认会出现当前设计的工程路径，需要设计者在最后面手动输入例化的 FIFO 这里输入了 “FIFO_EXAM”完成以上配置，点击 “Next>”。



④ 如图 4 所示，进行以下配置：

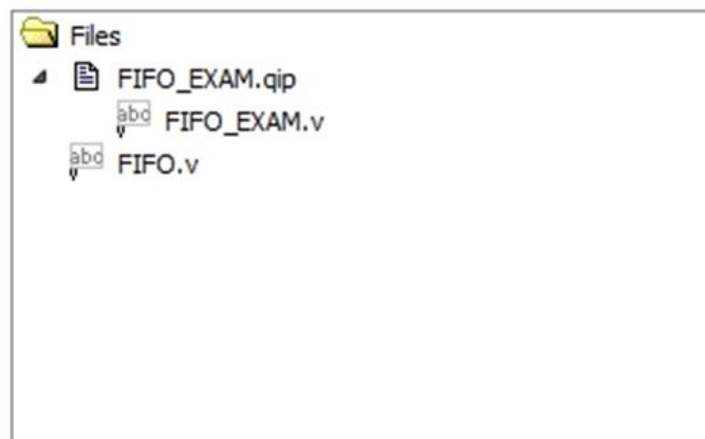
我们不用改变任何配置，直接用默认的配置。其中各选项的功能如图 4 所示。

直接点击 finish.

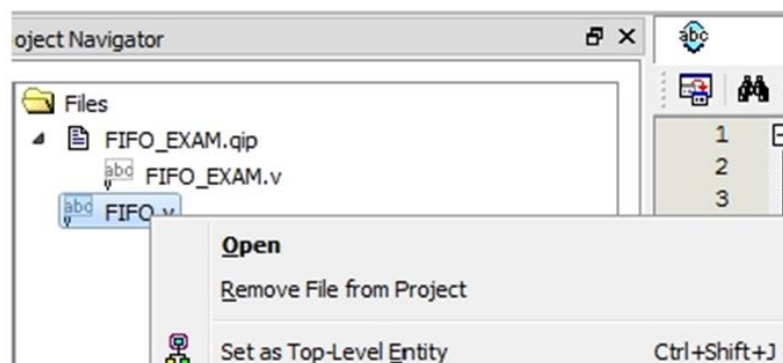


⑤ 回到 quatusII 中, 如图 5

出现了 FIFO_EXAM.v 文件, 这就是我们用来实例化的 verilog 文件。



⑥ 右键点击 FIFO.V



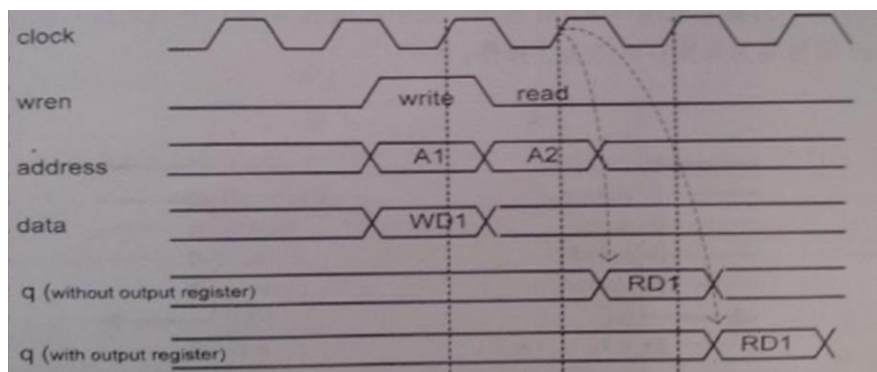
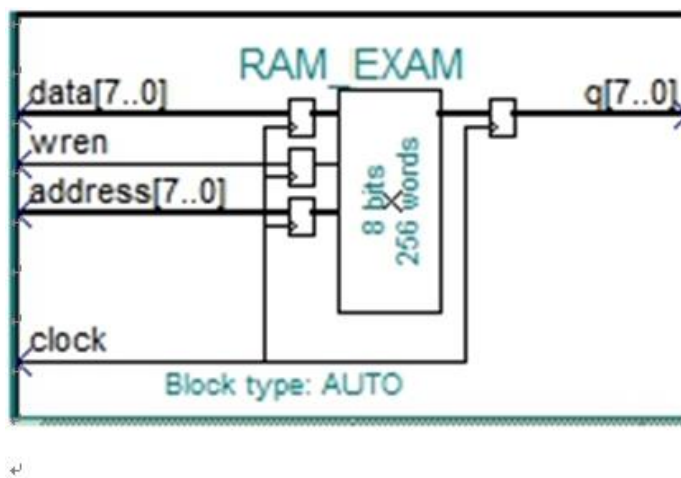


图 9.4.2 Altera 同步 RAM 读写时序

二、下面以一个简单的例子来说明 FIFO 的使用。如图



输入信号：

时钟信号 clock;

读写信号 wren;

数据输入端 data[7..0];

地址输入端 address[7..0];

输出信号：

数据读出端 q[7..0];

接下我们要设计一个从数据输入端输入数字 0~9 十个数，并通过仿真验证输出。

先写 verilog.v 程序

```
module RAM(clk,reset,dout);
input clk,reset;
output[7:0]dout;
//---寄存器定义
reg Forward;//高表示正向计时
reg [7:0]counter;//0~9
//---信号线定义
```

```
wire [7:0]data_buf;
wire wren;
wire [7:0]address_buf;
//---RAM 宏模块的实例化
RAM_EXAM (
.address(address_buf),
.clock(clk),
.data(data_buf),
.wren(wren),
.q(dout));
//---信号线的控制
assign address_buf=counter;
assign wren=(Forward)?1'b1:1'b0;//正向就写，反向读
assign data_buf=(Forward)?address_buf:8'hzz;//正向写入数据
//----主时序控制
always@(posedge clk or negedge reset)
begin
if(!reset) begin
counter<=8'h0;
Forward<=1'b1;
end
else if(Forward)
begin
if(counter<8'h9)
begin
counter<=counter+1'b1;
end
else
begin
Forward<=1'b0;//反向，读数据
counter<=8'h0;
end
end
else
begin
if(counter<8'h9)
begin
counter<=counter+1'b1;
end
else
begin
Forward<=1'b1;//正向，写入数据
counter<=8'h0;
end
end
end
end
```

```
end
end
end
endmodule
```

RAM 配置说明

ARM 配置需求

假定设计者已经新建了一个工程，然后需要配置一个 **FIFO**。该 **FIFO** 的引脚为：

输入信号：	时钟信号 clock
	读写信号 wren
	地址输入端 address[7..0]
	数据输入端 data[7..0]
输出信号：	数据读出端 q[7..0]

RAM 的配置步骤：

① 如图 1 所示，在 Quartus II 的菜单栏选择“Tools—>MegaWizard Plug-In Manager,”。



图 1 选择 MegaWizard

② 如图 2 所示，使用默认选项“Create a new custom megafunction variation”,点击“Next>”。



图 2

③ 如图 3 所示，进行以下配置：

◆ 在“Select a megafunction from the list below”窗口内打开“Memory Compiler”，选择“RAM: 1-PORT”

◆ 在“Which type of output file do you want to create?”下选择“Verilog HDL”，这是配置的 RAM 的语言，一般选择此项。

◆ 在“What name do you want for the output file?”里默认会出现当前设计的工程路径，需要设计者在最后面手动输入例化的 RAM 这里输入了“RAM_EXAM”完成以上配置，点击“Next>”

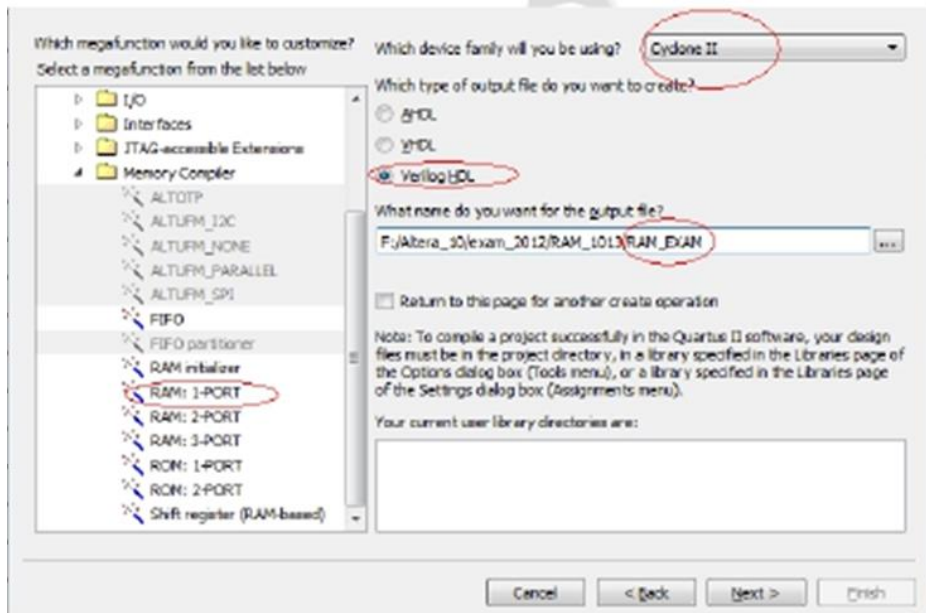
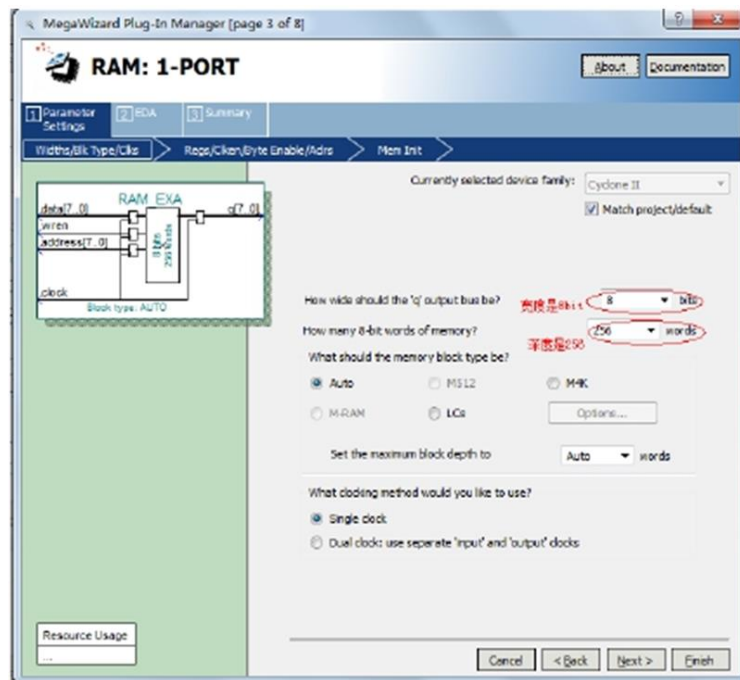


图 3

④ 如图 4 所示，进行以下配置：

我们不用改变任何配置，直接用默认的配置。其中各选项的功能如图 4 所示。直接点击 finish.



⑤ 回到 quatusII 中，如图 5

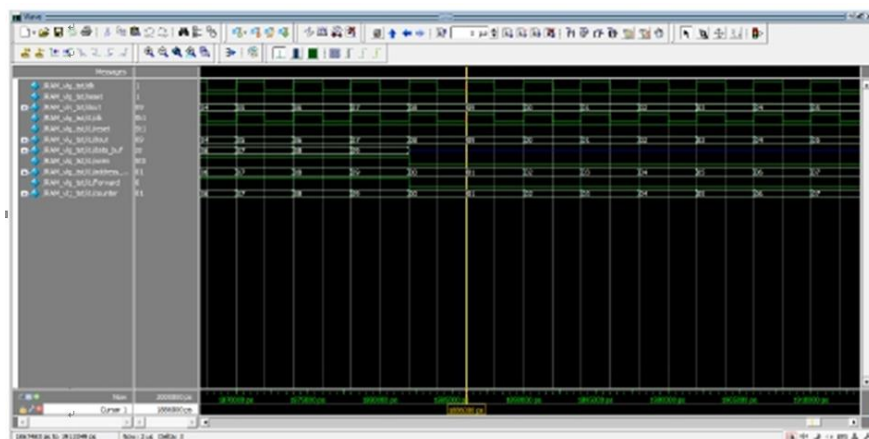
出现了 RAM_EXAM.v 文件，这就是我们用来实例化的 verilog 文件。

⑥ 右键点击 RAM.v

选择 Set as Top-Level Entity

⑦ 再点击编译按钮进行语法检测，通过后就可以进行仿真测试了。测试波形图

如下:



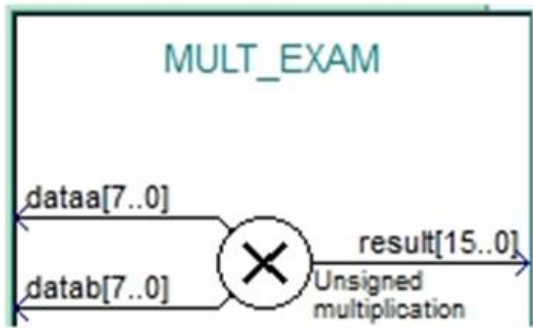
9.5 乘法器的使用

FPGA 中的乘法器的使用

一、简介

FPGA 内部带有乘法器，比使用 FPGA 的 LE 资源构建成的乘法器性能更加优越。乘法器简单来说就是实现乘法运算。

二、下面以一个简单的例子来说明乘法器的使用。如图



输入信号：数据输入端 dataa[7..0];
数据输入端 datab[7..0];
输出信号：数据读出端 result[15..0];
接下我们要设计两个 8bit 数相乘的电路。
先写 verilog.v 程序。

```
module mult(dataa,datab,result);
input [7:0]dataa,datab;
output [15:0]result;
MULT_EXAM mul(
.dataa(dataa),
.datab(datab),
.result(result));
endmodule
```

MULT 配置说明

MULT 配置需求

假定设计者已经新建了一个工程，然后需要配置一个 FIFO。该 FIFO 的引脚为：

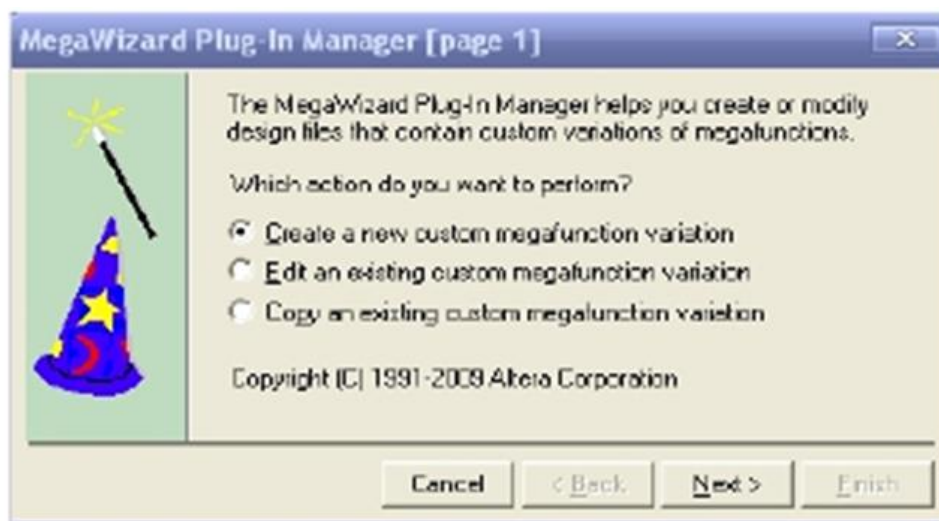
输入信号：	数据输入端 dataa[7..0];
	数据输入端 datab[7..0];
输出信号：	数据读出端 result[15..0]

FIFO 的配置步骤：

① 如图 1 所示，在 Quartus II 的菜单栏选择 “Tools—>MegaWizard Plug-In Manager, ”。



② 如图 2 所示，使用默认选项“Create a new custom megafunction variation”，点击“Next>”。



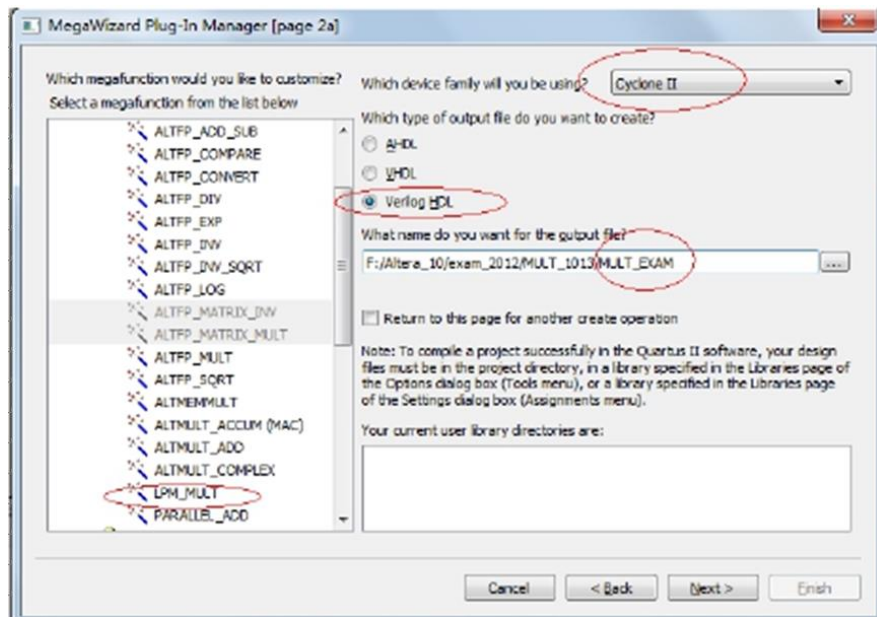
③ 如图 3 所示，进行以下配置：

◆ 在“Select a megafunction from the list below”窗口内打开“Memory Compiler”，选择“LPM_MULT”

◆ 在“Which type of output file do you want to create?”下选择“Verilog HDL”，这是配置的 MULT 的语言，一般选择此项。

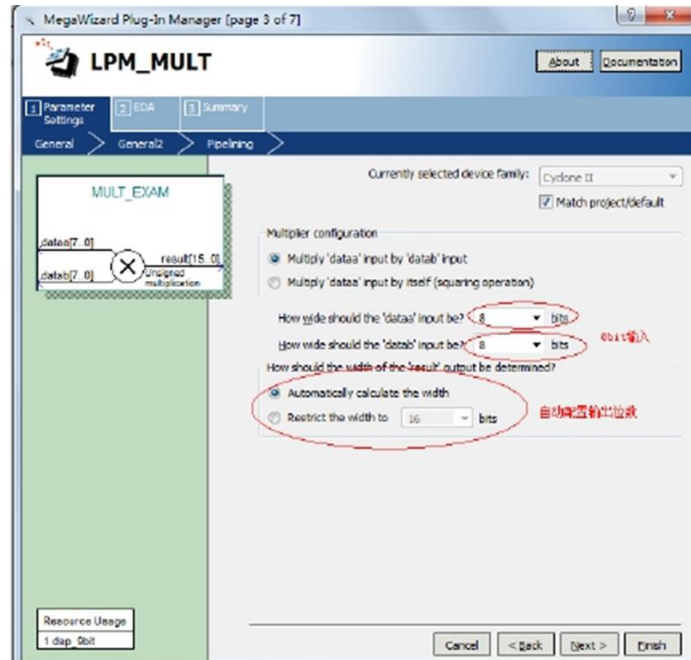
◆ 在“What name do you want for the output file?”里默认会出现当前设计的工程

路径,需要设计者在最后面手动输入例化的 MULT 这里输入了“MULT_EXAM”完成以上配置,点击“Next>”。



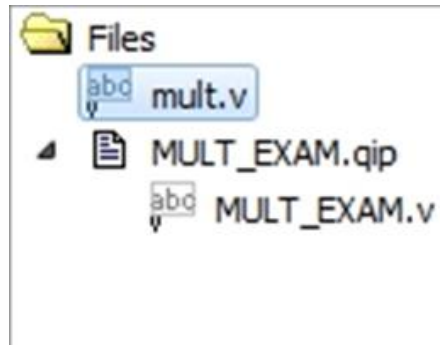
④ 如图 4 所示,进行以下配置:

我们不用改变任何配置,直接用默认的配置。其中各选项的功能如图 4 所示。直接点击 finish.



⑤ 回到 quatusII 中,如图 5

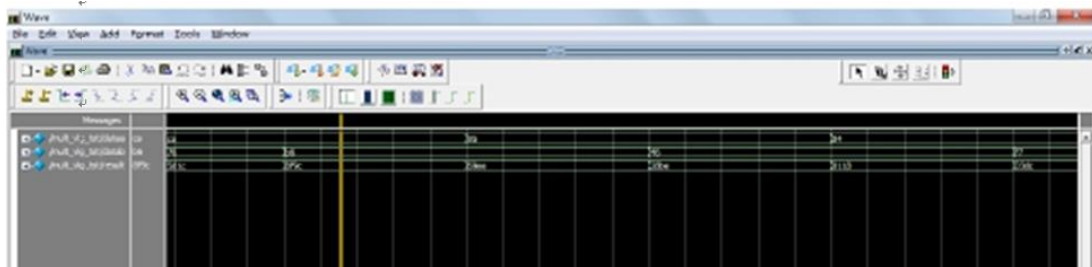
出现了 MULT_EXAM.v 文件,这就是我们用来实例化的 verilog 文件。



⑥ 右键点击 mult.V

选择 Set as Top-Level Entity

⑦ 再点击编译按钮进行语法检测，通过后就可以进行仿真测试了。测试波形图如下：



第十二章 综合实验

12.1 基于 DDS 的任意波形发生器

一、DDS 的原理

DDS(Direct Digital Frequency Synthesizer)直接数字频率合成器,也可叫 DDFS。

DDS 是从相位的概念直接合成所需波形的一种频率合成技术。不仅可以产生不同频率的正弦波,而且可以控制波形的初始相位、幅度等参数。

DDS 的设计原理在 9.12 章节已经有比较详细的介绍了,在这里不做过多讲解。

DDS 原理框图:

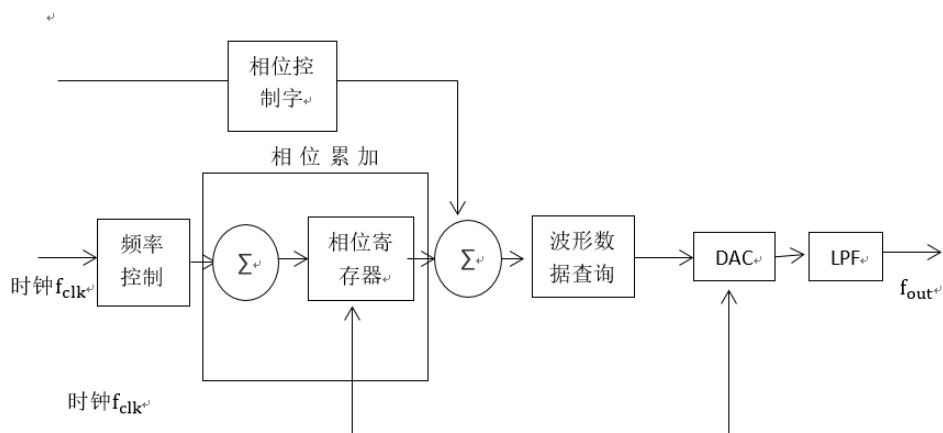


图 13-1

二、整体框架及其说明

波形选择、幅值、频率、相位按键输入。

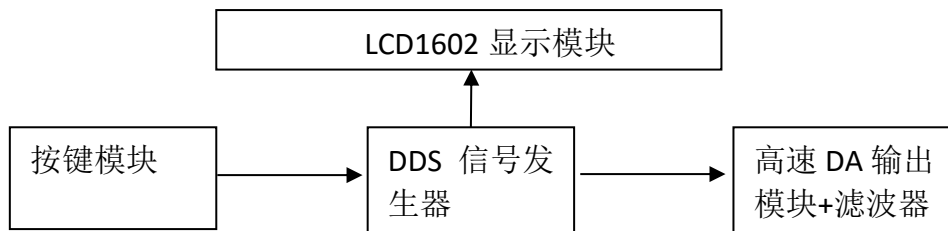


图 13-2

框架说明: 该模块的主要功能为产生任意信号, 这里的任意信号为: 正弦波、三角波、方波、锯齿波这四种波形。整个工程主要分为四个模块: 按键模块、DDS 信号发生器(在做这个实验前一定要先把 DDS 的原理看懂!!!)、AD9708 高速 DA 驱动、低通滤波器(硬件板上已经有了)。按键模块: 由于需要控制波形、幅值、频率、相位三个参数, 所以该模块需要按键, 由于按键为机械按键, 在按下的时候会产生抖动, 为了达到控制比较稳定, 在按键的输入端我们采用软件对按键进行消抖, 具体的过程

可以看教程的矩阵键盘那个章节。

三、设计要求和顶层设计原理图

设计的 DDS 实现用矩阵键盘实现信号参数的设置，包括设置频率、幅度、相位、波形种类。下图是顶层设计原理图，采用分模块设计思想，各模块生成器件符号，然后再原理图把各模块组装连接。

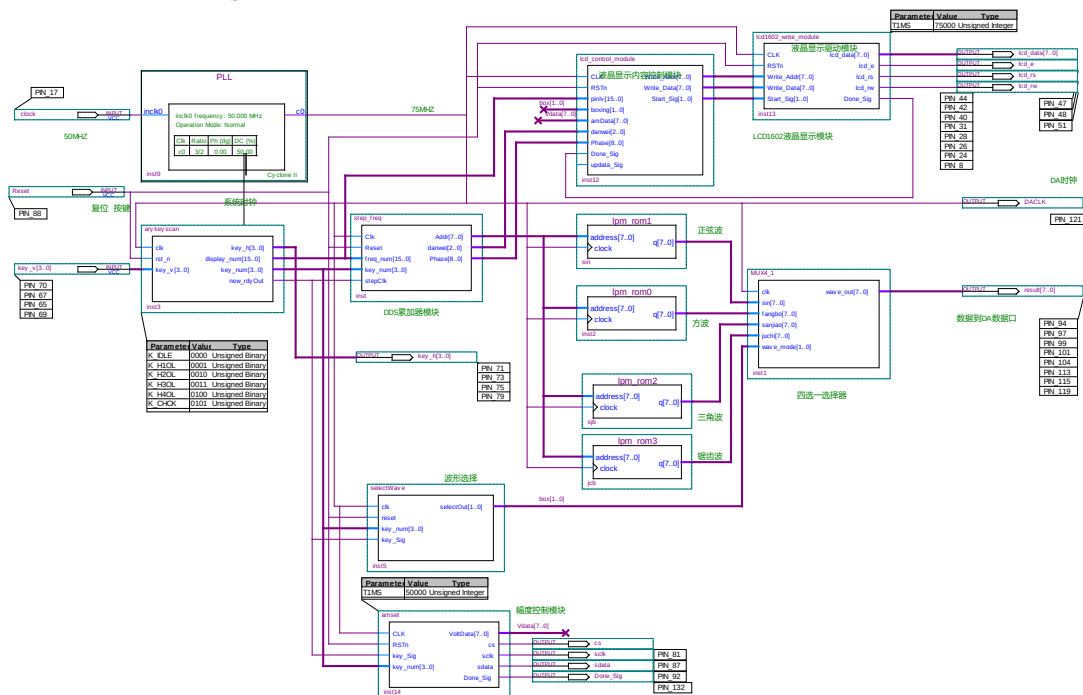


图 13-3

DDS 信号发生器：这里的采用 DDS 来产生四种波形：正弦波、三角波、方波、锯齿波，采用的位数为 8 位，因为我们的 DAC 芯片 AD9708 是 8 位分辨率的。这四种波形通过一个按键进行控制，同时我们需要调用 4 个 ROM 来存放这四个波形。而这四 ROM 里面的数据通过寻址来调用，这里需要一个加法器和一个累加器来产生 ROM 的地址。同过不断让地址累加，从而不断地从 ROM 中读取波形数据，然后将数据送往 DA 驱动模块中，这样最终便输出模拟的波形。**DDS 的具体过程**这里不详讲，具体看“9.12 DSS 与嵌入式逻辑分析仪的调用”。频率累加器和相位累加器，通过改变频率累加器的频率控制字就可以控制输出的波形频率，通过改变相位累加器的相位控制字就可以控制输出波形的相位。而幅值的改变是通过控制 DA 芯片 TLC5615 输出外部基准电压提供给波形转换的高速 DA 转换芯片 AD9708，从而控制波形输出幅度，而改变的过程是通过外部的按键进行改变。

DAC TLC5615 驱动：具体看“12.2 DA_TLC5615 电压输出”这一章节。

矩阵键盘驱动：具体看“9.14 矩阵键盘控制实验”这一章节。

LCD1602 驱动：具体看“9.11 液晶 LCD1602 显示”这一章节。

四、实验原理图

幅度控制部分采用双 D/A 技术，由 FPGA 控制 TLC5615 数模转换，改变其输出电压转换后作为高速 D/A 转换器 AD9708 的基准电压，由此即可由 FPGA 控制输出波形的幅度。该方法能准确实现 0.1V 小幅度的步进，调幅效果良好。DA 模块电路如下所示。

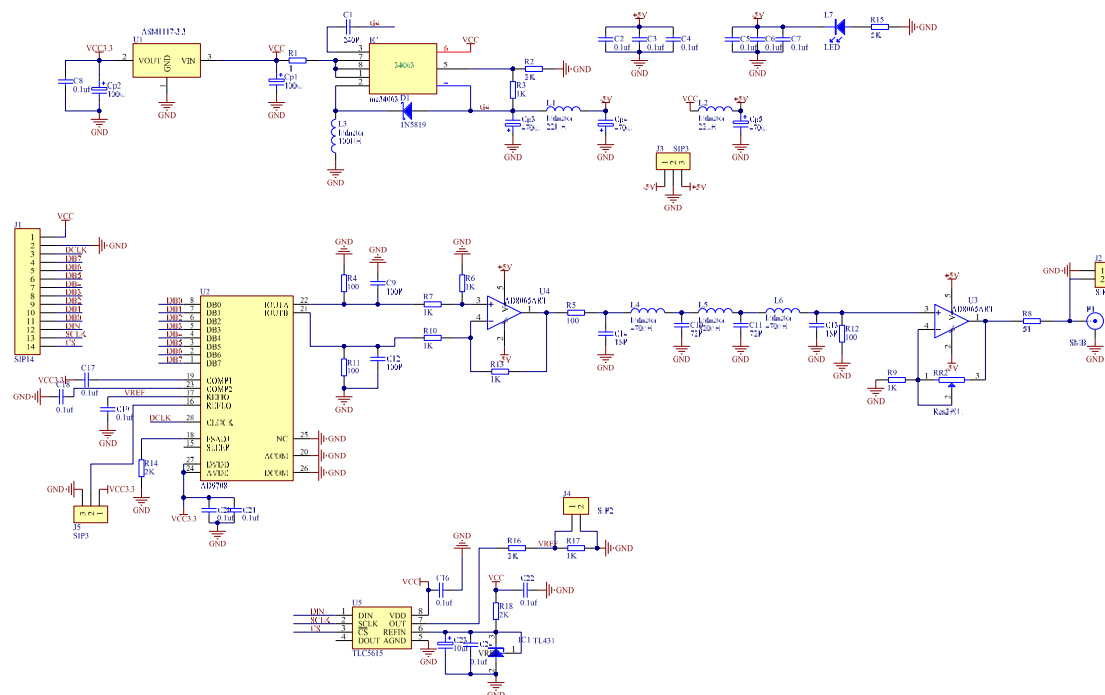


图 13-4

12.2 基于 FPGA 的出租车计费器设计

一、设计原理

由于各地出租车计价情况的不同，本设计主要依据桂林市出租车计价情况作为参考依据。先来了解一下某市出租车是如何计费的：

表 3.1 某市计费标准

起步 8 元 (3km)
2 元/km(超过 3km) 白天
3 元/km(超过 3km) 深夜
1 元/1 分钟 等待时间

从上面表中可以看出，目前出租车行业由于调价、路桥、夏季电调、燃油费波动等诸多因素要求出租车计费器在价格调节方面做到灵活可靠，易于操作等特点。

出租车计费器系统是硬件描述语言的实际应用，利用 verilog 语言设计出

来的出租车计费器系统将实现计程模块、计时模块以及动态扫描模块等设计。计程模块将用计数器来完成，计数器对脉冲数计数，然后提供给程序数据。通过不同的信号，然后用比较器可以让我们确定出租车是在车行计程还是车停计时。再将数据传输到计费模块，通过多种条件判定，最后确定输出值，然后相加确定最后费用，并显示出来。

出租车载客后，启动计费器，整个系统开始运行，里程计数器和时间计数器从 0 开始计数，费用计数器从 8 开始计算。再根据行驶里程或停止等待的时间按以上的标准计费。若在行驶状态，则计程器开始加计数，超过两公里后每公里 1 元。若出租车停止等待状态，则计时器开始加计数，当时间超过三分钟后，计费器以每 3 分钟 1 元累加。出租车到达目的地停止后，停止计费器，显示总费用。

二、整体框架及其说明

根据出租车计费器的工作过程，本系统采用分层次、分模块的方式设计，其本系统组成框图如下所示。其中行驶路程计数模块、等待时间计数模块和计费模块，用来统计路程、等待时间和总费用，控制模块是用来控制开始和停止计费，实现白天和晚上计费模式切换，模拟等待时间等功能。

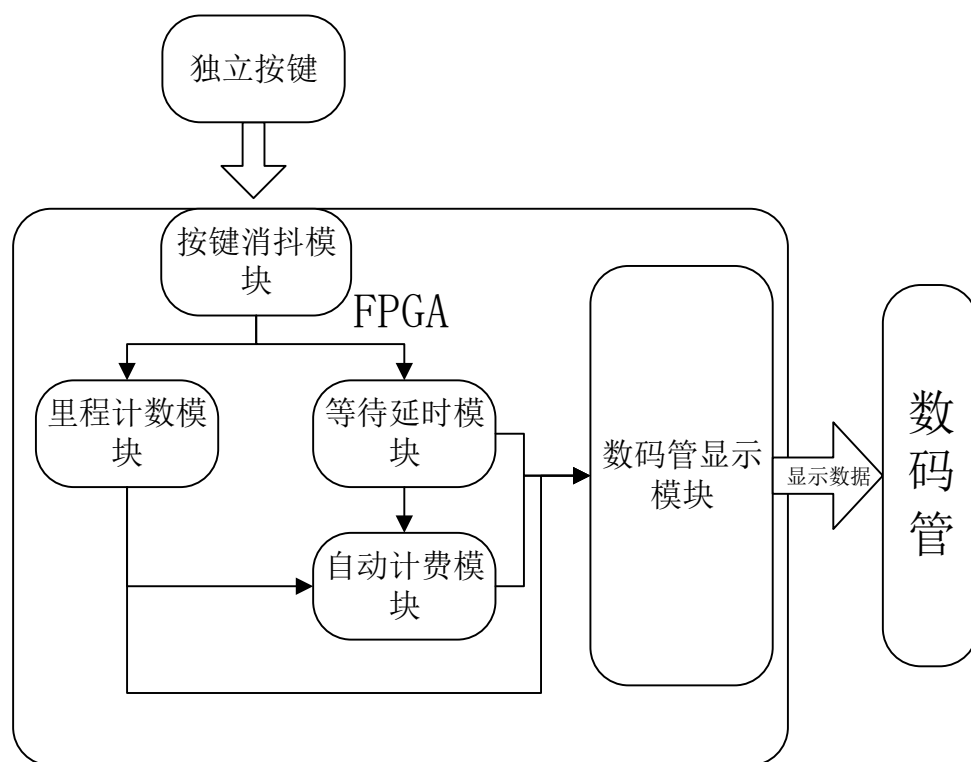


图 13-5

三、设计要求和顶层设计原理图

基于 FPGA 设计一出租车计费系统。该系统包括硬件和软件，系统硬件有核心芯片 FPGA 和相应配置电路、外围电路；系统软件有分频模块，控制模块，里程模块，等待计时模块，计费模块和显示模块等。系统在 quartus 平台完成编程、仿真、下载和调试，实现白天、夜间不同计费标准的计费系统。

(1) 设计一个出租车计费系统硬件电路；

(2) 该计价器的计费系统：

白天 (5:00-23:00)：行程 3 公里内，起步费为 8 元；3 公里外以每 100 米 0.2 计费。深夜 (23:00-5:00)：行程 3 公里内，起步费为 10 元；3 公里外以每 100 米 0.3 元计费。等待时间每分钟一元钱。

(3) 能以十进制显示行驶公里数、等待累计时间、总费用。xxx.x 元 xx 公里 xx 分钟。

设计的出租车计费器需要独立按键来模拟操作，包括停止按键、启动按键、白天晚上模式切换按键、等待模式按键，需要设计按键消抖模块；如下图 13-6 所示，还需要设计按键工作模式控制模块，分频产生里程信号模块，里程计数模块，等待时间计时模块，费用计算模块和数码管显示模块。采用分模块设计思想，各模块生成器件符号，然后再原理图把各模块组装连接。

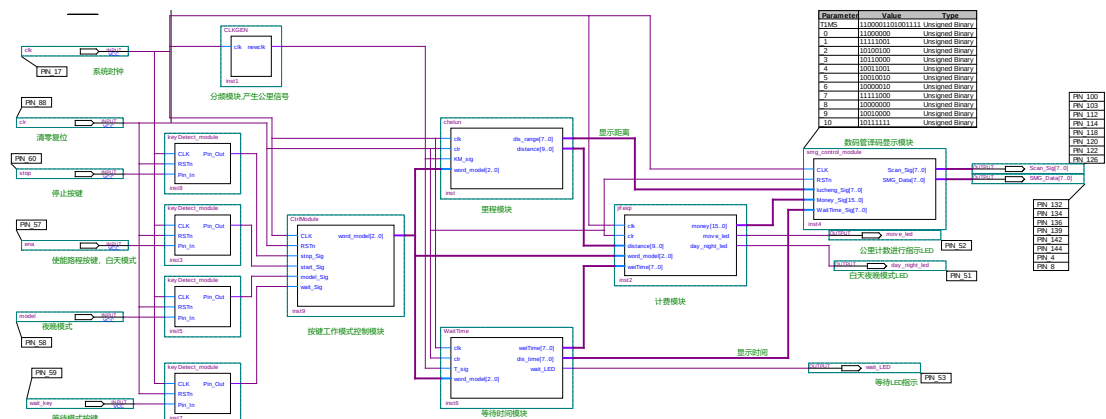


图 13-6

四、实验原理图

本实验需要用到的硬件电路包括独立按键、数码管显示电路，都是比较常用的电路。如图 13-7 所示。

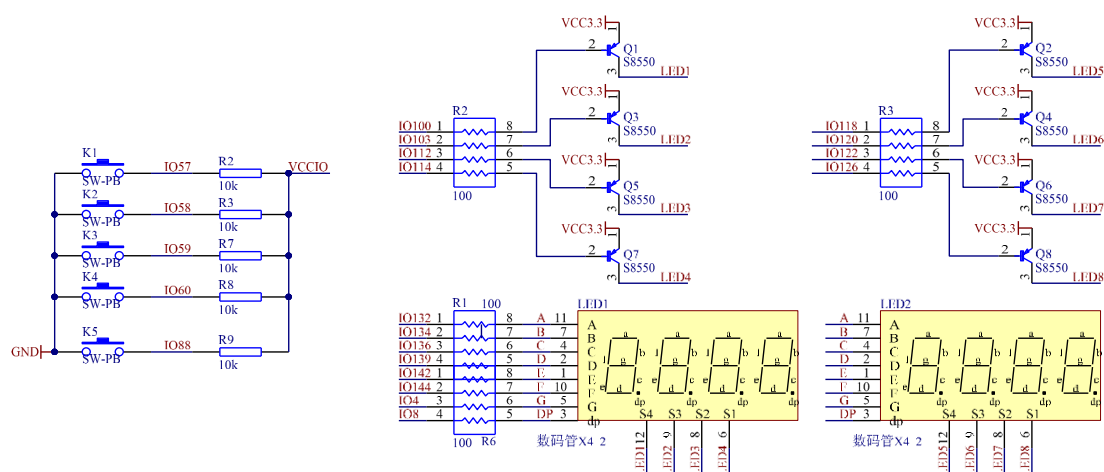


图 13-7

12.3 基于 FPGA 的交通灯设计

一、设计原理

十字路口车辆穿梳，行人熙攘，车行道，人行道，有条不紊。那么靠什么来实现这井然有序呢？靠的是交通信号灯的自动指挥系统。

交通灯通常指由红、黄、绿三种颜色灯组成用来指挥交通的信号灯。绿灯亮时，准许车辆通行，黄灯亮时，已越过停止线的车辆可以继续通行；红灯亮时，禁止车辆通行。

模拟十字路口交通信号灯的工作过程，利用交通信号灯上的四组红，黄，绿 LED 发光二极管作为交通信号灯，设计一个交通信号灯控制器。

模拟两条公路，一条交通主干道，一条交通支干道，在主干道和支干道的交叉路口上设置红，绿，蓝灯进行交通管理。

应用 verilog 硬件描述语言编写程序，利用软件仿真出结果。

二、整体框架及其说明

交通信号灯控制方式很多，本实验设计一个十字路口交通灯控制电路，要求东西、南北两条干道的红、绿、黄交通灯按要求循环变化，并以倒计时方式指示干道通行或禁止的维持时间。并在 FPGA 实验板上实现所设计电路的功能。

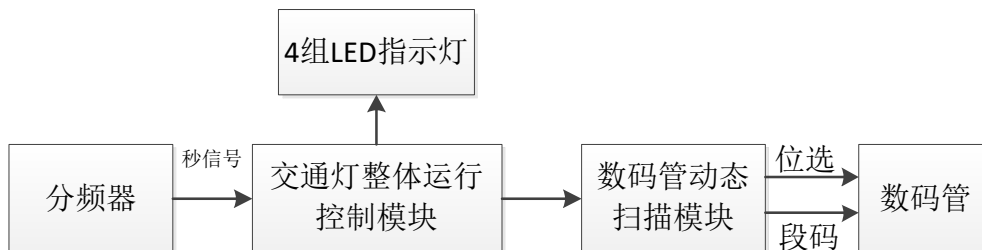


图 13-8 交通灯系统设计框图

如图 13-8 交通灯系统设计框图所示,系统设计包括分频器模块、控制模块、数码管动态扫描模块构成。分频器产生秒脉冲信号,作为控制模块的倒计时驱动信号,控制模块实现十字路口交通灯控制规则的 LED 指示灯状态切换和倒计时计数,数码管模块显示倒计时时间。

三、设计要求

控制系统完成对十字路口交通信号灯的控制,十字路口由一条东西方向的主干道(简称 A 道)和南北方向的支干道(简称 B 道)构成。十字路口交通灯控制规则为:

- (1) 初始状态为 4 个方向的红灯全亮,时间 1s。
- (2) 东西方向绿灯亮,南北方向红灯亮。东西方向通车,时间 30s。
- (3) 东西方向黄灯亮,南北方向红灯亮,时间 3s。
- (4) 东西方向红灯亮,南北方向绿灯亮。南北方向通车,时间 20s。
- (5) 东西方向红灯亮,南,北方向黄灯亮,时间 3s。
- (6) 返回(2),继续运行。
- (7) 如果发生紧急事件,例如救护车,警车通过,则按下单脉冲按钮,使得东,南,西,北四个方向红灯亮。紧急事件结束后,松开单脉冲按钮,将恢复到被打断的状态继续运行。

四、详细设计及实现

此设计即为一个典型的时序状态机,共有六个大的状态,现设定:

S0: 四个方向的红灯全亮

S1: 东、西方向绿灯亮,南、北方向红灯亮

S2: 东、西方向黄灯闪烁,南、北方向红灯亮

S3: 东、西方向红灯亮,南、北方向绿灯亮

S4: 东、西方向红灯亮,南、北方向黄灯闪烁

S5: 东、西、南、北四个方向的红灯亮

则状态图如下:

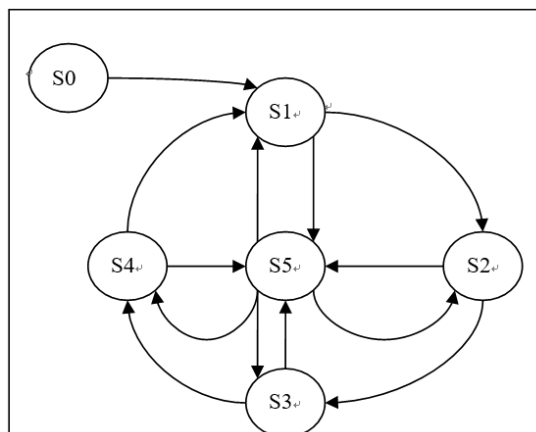


图 3.1 状态机的状态图

此状态机是该设计的核心模块，由其控制着东、西、南、北方向的交通。仔细分析设计要求可知，东西方向交通灯状态变化相同，南北方向亦相同，故实际上只需控制两组相同的交通灯，开发板上有 **12 个 LED 按照交通灯样式进行布局**。状态机有六个输出变量，分别控制东西、南北方向的红、绿、黄灯的亮灭状态。另外，状态机还有三个输入变量，分别为复位信号、时钟信号、紧急状况控制信号、白天夜晚模式切换信号。

由于在 verilog HDL 语言中，同一个进程内只能有一个时钟，所以紧急信号宜以电平状态来触发，低电平时进入紧急状态，红灯都亮起；白天夜晚模式切换信号也以电平状态来识别。

为了控制各个状态持续的时间，需在状态机中设置倒计时装置。可将输入状态机的时钟信号的频率设计为 1Hz, 即一个脉冲周期为 1s。开发板上晶振采用 50MHz, 这样需要计数器来分频产生 1HZ 时钟，此外，复位信号利用高低电平来控制，当电平为低时，状态机保持东、西、南、北四个方向红灯均亮的状态，若为高则由初始状态进入正常状态循环。

综上所述，可得总体设计顶层设计原理图如下：

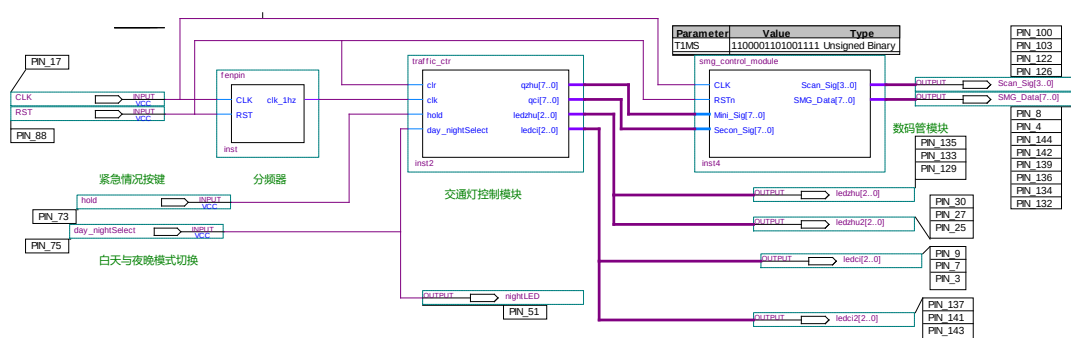


图 13-9 交通灯系统顶层设计原理图

设计的出交通灯顶层设计原理图如下图 13-9 所示，系统设计包括分频器模块、控制模块、数码管动态扫描模块构成。采用分模块设计思想，各模块生成器件符号，然后再原理图把各模块组装连接。

CLK 为输入系统时钟 50MHz (用于计时的 1HZ 由分频器分频后提供)；RST 为复位信号，几个模式复位信号连接在一起，由一个按键开关控制；hold 为紧急信号(由紧急信号发生器控制)；day_nightSelect 为白天夜晚模式切换信号；输出变量 ledzhu 控制东西方向红、绿、黄交通灯的亮灭状态；南北方向则由 ledci 控制；另外倒计时计数输出到数码管模块。

五、状态机核心程序分析

状态机用于控制十字路口交通灯的状态变化。此模块中设计的是异步复位和异步等待(即进入紧急状态)，复位信号是低电平有效，紧急信号是低电平有效。另外，为了控制各个状态的持续时间，此状态机中还设计了一个计数装置，用于计时。交通灯控制模块的源代码如下：

```

1.  module
        traffic_ctr(clr,clk,hold,day_nightSelect,qzhu,qci,ledzhu,ledci);
2.  input clr,clk;
3.  input    hold;
4.  input    day_nightSelect;
5.  output [7:0] qzhu,qci;
6.  output [2:0]ledzhu,ledci;//r y g LED对应顺序：红黄绿
7.  reg [2:0]ledzhu,ledci;
8.  reg [7:0] qtempZ,qtempC;
9.  reg [3:0] i;
10.
11. always    @(posedge clk or negedge clr)
12.     begin
13.         if (~clr)
14.             begin
15.                 qtempZ=8'd30;qtempC=8'd33;
16.                 i=4'd4;
17.                 ledzhu=3'b111;
18.                 ledci=3'b111;
19.             end
20.         else if(hold)
21.             begin
22.                 case(i)
23.                     4'd0:
24.                         begin

```

```

25.         if (qtempZ==8'd1) //切换到主干道黄灯亮3秒状态,
26.         begin i=i+1'b1;
27.             qtempZ=8'd3;
28.             qtempC=8'd3;
29.             ledzhu=3'b101; //LED对应顺序: 红黄绿
30.             ledci=3'b011;
31.         end
32.         else
33.         begin
34.             qtempZ=qtempZ-1;
35.             qtempC=qtempC-1'b1;
36.             ledzhu=3'b110; //主干道通行30秒倒计时,
37.             ledci=3'b011;
38.         end
39.     end
40. 4'd1:
41.     begin
42.         if (qtempZ==8'd1)
43.         begin
44.             i=i+1'b1;
45.             if(day_nightSelect)begin qtempZ=8'd23;qtempC=8'd20;end
46.             else begin qtempZ=8'd18;qtempC=8'd15;end
47.             ledzhu=3'b011;
48.             ledci=3'b110;
49.         end
50.         else
51.         begin
52.             qtempZ=qtempZ-1'b1;
53.             qtempC=qtempC-1'b1;
54.             ledzhu=3'b101; //主干道黄灯亮3秒倒计时
55.             ledci=3'b011;
56.         end
57.     end
58. 4'd2:
59.     begin
60.         if (qtempC==1)
61.         begin i=i+1'b1;
62.             qtempC=8'd3;
63.             qtempZ=8'd3;
64.             ledzhu=3'b011;
65.             ledci=3'b101; //切换到次干道黄灯亮3秒状态,

```

```

66.         end
67.     else
68.         begin
69.             qtempC=qtempC-1'b1;
70.             qtempZ=qtempZ-1'b1;
71.             ledzhu=3'b011;
72.             ledci=3'b110;    //次干道通行20秒倒计时,
73.         end
74.     end
75. 4'd3:
76.     begin
77.         if (qtempC==8'd1)
78.             begin
79.                 i=4'd0;
80.                 if(day_nightSelect)begin qtempZ=8'd30;qtempC=8'd33;end
81.                 else begin qtempZ=8'd20;qtempC=8'd23;end
82.                 ledzhu=3'b110;
83.                 ledci=3'b011;
84.             end
85.         else
86.             begin
87.                 qtempC=qtempC-1'b1;
88.                 qtempZ=qtempZ-1'b1;
89.                 ledzhu=3'b011;
90.                 ledci=3'b101;    //次干道黄灯亮3秒倒计时
91.             end
92.         end
93.     default:
94.         Begin
95.             if(day_nightSelect)begin qtempZ=8'd30;qtempC=8'd33;end
96.             else begin qtempZ=8'd20;qtempC=8'd23;end
97.                 i=4'd0;
98.                 ledzhu=3'b111;
99.                 ledci=3'b111;
100.         end
101.     endcase
102. end
103.
104. else
105.     begin

```

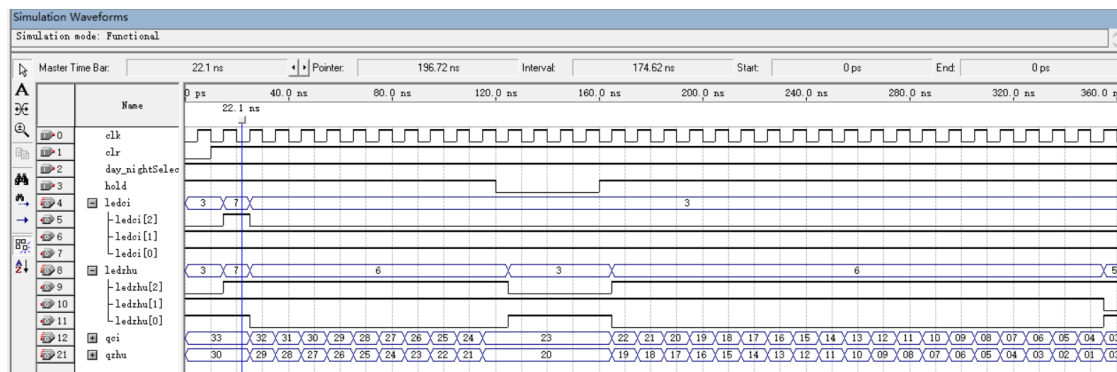
```

106.      ledzhu=3'b011;//紧急情况红灯都亮
107.      ledci=3'b011;
108.  end
109. end
110.
111. assign qzhu[7:4]= qtempZ/10;//送数码管显示计时时间
112. assign qzhu[3:0]= qtempZ%10;
113. assign qci[7:4]= qtempC/10;
114. assign qci[3:0]= qtempC%10;
115. endmodule

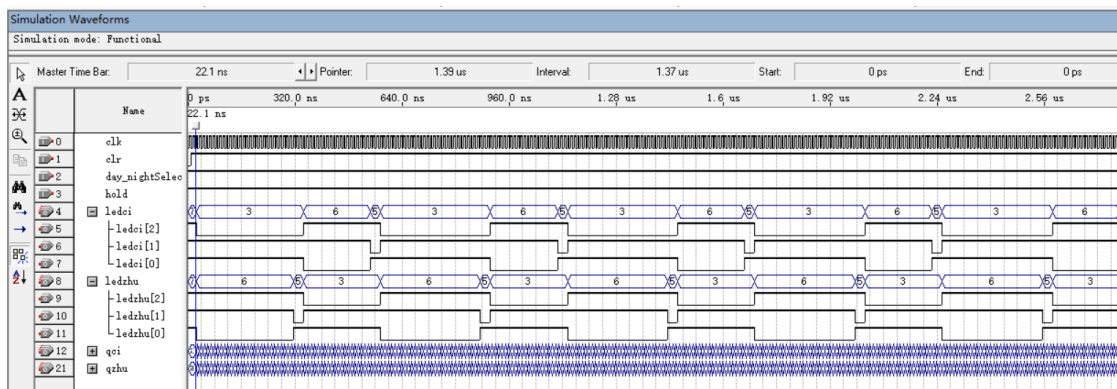
```

六、功能仿真图

仿真波形图如下：



(a)



(b)

图 13-10 交通灯仿真图

初始状态为四个方向的红灯全亮，时间 1 秒。然后东、西方向绿灯亮，南、北方向红灯亮。东、西方向通车，时间 30 秒。但在此过程中，有一个紧急的信号出现，所以在大概在倒计时 23 秒左右，出现紧急信号，东西南北方向红灯都亮，倒计时停止。在解除紧急信号后，仿真图中恢复 S1 的状态继续倒计时。经验证，图中均符合题目要求。

第十三章 学习 FPGA 技术总结

1、如何学习 FPGA？

FPGA 对我们如此重要，那么对于初学者来说，到底该如何学习 FPGA 呢？学习一门技术最好有合适的指导老师，这样对掌握 FPGA 技术更容易，可惜的是大部分的学校还未开设相关的课程，也缺少具有实践经验的老师，那么如何才能找到一种捷径帮助初学者快速学会如此具有竞争力的技术呢？

2、掌握 FPGA 的编程语言

在学习一门技术之前我们往往从它的编程语言开始，学习 FPGA 的编程语言有两种：VHDL 和 [Verilog](#)，这两种语言都适合用于 FPGA 的编程，VHDL 在欧洲的应用较为广泛，而 Verilog 在中国、美国、日本、台湾等地应用较为广泛，建议学习 Verilog，因为它非常易于学习，很类似于 C 语言，如果具有 C 语言基础的人，只需要花很少的时间便能迅速掌握 Verilog，而 VHDL 则较为抽象，学习的时间较长。

3、FPGA 实验尤为重要

除了学习编程语言以外，更重要的是实践，将自己设计的程序能够在真正的 FPGA 里运行起来，这时我们需要选一块板子进行实验，一般的红色飓风的板子基本上可以满足大家的需求，大家感兴趣的不妨买一块做做实验。

4、FPGA 培训不可忽视

在有条件的情况下，参加 FPGA 的培训可以在短时间内大幅提升自己的水平，因为有老师带着可以省去了很多弯路。笔者在网上发现国内第一家大学 EDA 实验室创始人之一的夏宇闻教授和未名芯锐搞了一个 FPGA 培训班，感兴趣的朋友可以去看看，网上也有很多视频资源，也可下下来看看。

我想只要大家想学 FPGA，想从事 FPGA 工作，总会有办法找到适合自己的方法。

良好代码编写风格可以满足信、达、雅的要求。在满足功能和性能目标的前提下，增强代码的可读性、可移植性，首要的工作是在项目开发之前为整个设计团队建立一个命名约定和缩略语清单，以文档的形式记录下来，并要求每位设计人员在代码编写过程中都要严格遵守。良好代码编写风格的通则概括如下：

(1) 对所有的信号名、变量名和端口名都用小写，这样做是为了和业界的习惯保持一致；对常量名和用户定义的类型用大写；

(2) 使用有意义的信号名、端口名、函数名和参数名；

(3) 信号名长度不要太长；

(4) 对于时钟信号使用 **clk** 作为信号名，如果设计中存在多个时钟，使用 **clk** 作为时钟信号的前缀；

(5) 对来自同一驱动源的信号在不同的子模块中采用相同的名字，这要求在芯片总体设计时就定义好顶层子模块间连线的名字，端口和连接端口的信号尽可能采用相同的名字；

(6) 对于低电平有效的信号，应该以一个下划线跟一个小写字母 **b** 或 **n** 表示。注意在同一个设计中要使用同一个小写字母表示低电平有效；

(7) 对于复位信号使用 **rst** 作为信号名，如果复位信号是低电平有效，建议使用 **rst_n**；

(8) 当描述多比特总线时，使用一致的定义顺序，对于 **verilog** 建议采用 **bus_signal[x:0]** 的表示；

(9) 尽量遵循业界已经习惯的一些约定。如 ***_r** 表示寄存器输出，***_a** 表示异步信号，***_pn** 表示多周期路径第 **n** 个周期使用的信号，***_nxt** 表示锁存前的信号，***_z** 表示三态信号等；

(10) 在源文件、批处理文件的开始应该包含一个文件头、文件头一般包含的内容如下例所示：文件名，作者，模块的实现功能概述和关键特性描述，文件创建和修改的记录，包括修改时间，修改的内容等；

(11) 使用适当的注释来解释所有的 **always** 进程、函数、端口定义、信号含义、变量含义或信号组、变量组的意义等。注释应该放在它所注释的代码附近，要求简明扼要，只要足够说明设计意图即可，避免过于复杂；

(12) 每一行语句独立成行。尽管 **VHDL** 和 **Verilog** 都允许一行可以写多个语句，当时每个语句独立成行可以增加可读性和可维护性。同时保持每行小于或等于 72 个字符，这样做都是为了提高代码得可读性；

(13) 建议采用缩进提高续行和嵌套语句得可读性。缩进一般采用两个空格，如果空格太多则在深层嵌套时限制行长。同时缩进避免使用 **TAB** 键，这样可以避免不同机器 **TAB** 键得设置不同限制代码得可移植能力；

(14) 在 RTL 源码的设计中任何元素包括端口、信号、变量、函数、任务、模块等的命名都不能取 Verilog 和 VHDL 语言的关键字;

(15) 在进行模块的端口申明时, 每行只申明一个端口, 并建议采用以下顺序: 输入信号的 `clk`、`rst`、`enables other control signals`、`data and address signals`。然后再申明输出信号的 `clk`、`rst`、`enables other control signals`、`data signals`;

(16) 在例化模块时, 使用名字相关的显式映射而不要采用位置相关的映射, 这样可以提高代码的可读性和方便 debug 连线错误;

(17) 如果同一段代码需要重复多次, 尽可能使用函数, 如果有可能, 可以将函数通用化, 以使得它可以复用。注意, 内部函数的定义一般要添加注释, 这样可以提高代码的可读性;

(18) 尽可能使用循环语句和寄存器组来提高源代码的可读性, 这样可以有效地减少代码行数;

(19) 对一些重要的 `always` 语句块定义一个有意义的标号, 这样有助于调试。注意标号名不要与信号名、变量名重复;

(20) 代码编写时的数据类型只使用 IEEE 定义的标准类型, 在 VHDL 语言中, 设计者可以定义新的类型和子类型, 但是所有这些都必须基于 IEEE 的标准;

(21) 在设计中不要直接使用数字, 作为例外, 可以使用 0 和 1。建议采用参数定义代替直接的数字。同时, 在定义常量时, 如果一个常量依赖于另一个常量, 建议在定义该常量时用表达式表示出这种关系;

(22) 不要在源代码中使用嵌入式的 `dc_shell` 综合命令。这是因为其他的综合工具并不认得这些隐含命令, 从而导致错误的或较差的综合结果。即使使用 Design Compiler, 当综合策略改变时, 嵌入式的综合命令也不如放到批处理综合文件中易于维护。这个规则有一个例外的综合命令, 即编译开关的打开和关闭可以嵌入到代码中;

(23) 在设计中避免实例化具体的门级电路。门级电路可读性差, 且难于理解和维护, 如果使用特定工艺的门电路, 设计将变得不可移植。如果必须实例化门电路, 我们建议采用独立于工艺库的门电路, 如 SYNOPSYS 公司提供的 GTECH 库包含了高质量的常用的门级电路;

(24) 避免冗长的逻辑和子表达式;

(25) 避免采用内部三态电路, 建议用多路选择电路代替内部三态电路。

规则 #1: 建立时序逻辑模型时, 采用非阻塞赋值语句。

规则 #2: 建立 latch 模型时, 采用非阻塞赋值语句。

规则 #3: 在 always 块中建立组合逻辑模型时, 采用阻塞赋值语句。

规则 #4: 在一个 always 块中同时有组合和时序逻辑时时, 采用非阻塞赋值语句。

规则 #5: 不要在一个 always 块中同时采用阻塞和非阻塞赋值语句。

规则 #6: 同一个变量不要在多个 always 块中赋值。

规则 #7: 调用 \$strobe 系统函数显示用非阻塞赋值语句赋的值。

规则 #8: 不要使用 #0 延时赋值。