# FPGA 实战手册

基础实例

综合实战

实例+教程+开发板

# 目录：

# 前言

FPGA 是为何物？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 实战手册》？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。实战手册不仅仅是实验手册，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。

大西瓜 FPGA 设计团队

2013.06.19

QQ:776231646

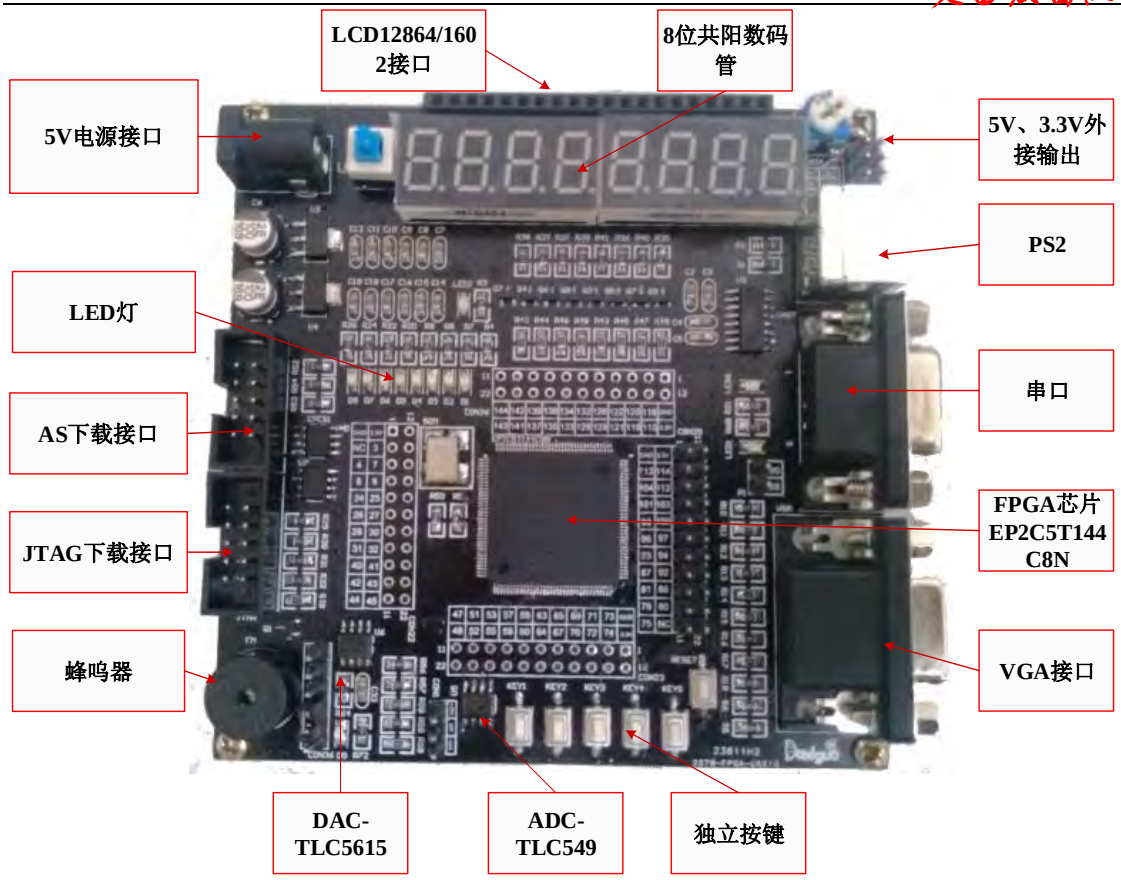
邮箱：776231646@qq.com

| 版本 V.10 | 2013.06.19 |
| --- | --- |
| | |
| | |

# 第一章开发板硬件资源

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EDA技术，让你领略EDA设计和并行设计的魅力。为此，我们的开发板配套的基础资源有：LED、数码管、蜂鸣器、按键、串口等，进阶实验的资源有：LCD、IIC-AT24C04、数模转换芯片TLC5615、模数转换芯片TLC549、PS2接口、VGA接口，同时，采用Cyclone II系列高性价比的FPGA芯片。为方便外扩，我们将所有IO引出。

为了让你更好的学习，开发板配备了相关的例程和电子书籍、资料，以及LPM宏功能模块和IP核的调用，让你掌握在FPGA上跑51核……，熟悉在FPGA上嵌入NiosII软核。相信你在掌握了各个模块后，就能领略FPGA的设计思想，熟练掌握硬件描述语言的设计方法和EDA的设计软件。大西瓜FPGA第一代开发板实物图如下图：




## 各个模块：

| | |
|---|---|
| LED 灯 | 8 个红色高亮 LED 灯，用户自定义显示 |
| AS 下载接口 | FPGA 的 ASP 模式下载接口 |
| EPCS1 | 用于 ASP 模式下的程序烧录 |
| IIC-AT24C04 | 用 IIC 驱动 EPROM |
| JTAG 下载接口 | 用于 JTAG 方式下载接口 |
| 蜂鸣器 | 用户自定义，根据编程可以发出各种声音 |
| DAC-TLC5615 | 14M10 位串行数模转换芯片 |
| ADC-TLC549 | 4M8 位串行模数转换芯片 |
| 独立按键 | 根据需要设定按键功能 |
| VGA | 用来驱动液晶屏 |
| 串口 | 用来和 PC 通讯 |
| PS2 | 用来驱动 PS2 接口的鼠标或者键盘等设备 |
| 8 位共阳数码管 | 用户自定义显示 |
| LCD1602/12864 接口 | LCD1602 和 12864 的共用接口 |

# 第二章 FPGA 的简介

## 2.1 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。Xilinx 和 Altera 公司也推出了相应的 SOC FPGA 产品。

## 2.2 FPGA 概述

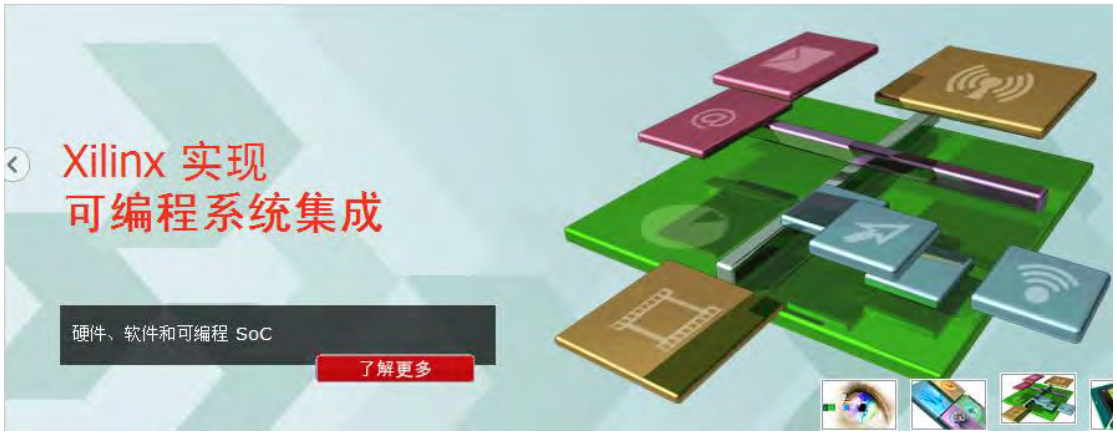
FPGA 是 Field Programmable Gate Array 的缩写，即现场可编程门阵列，它是在 PAL、GAL、CPLD 等可编程器件的基础上进一步发展的产物。它是作为专用集成电路(ASIC)领域中的一种半定制电路而出现的，既解决了定制电路的不足，又克服了原有可编程器件门电路数有限的缺点。

FPGA 是 Ross Freema 于 1985 年发明的，当时第一个 FPGA 采用 2μm 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 状况**

FPGA 市场占有率最高的两大公司 Xilinx 和 Altera。




**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/


**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；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 Startix，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，是目前主流产品，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 的下一代产品，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：刚刚推出的新一代低成本 FPGA，目前市场零售还不容易买到，估计从 2005 年年底开始，将逐步取代 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：Startix 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，将被 StratixII 逐渐取代。

**StratixII**: Stratix 的下一代产品，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，是未来几年中，Altera 在高端 FPGA 市场中的主力产品。StrtratixV 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。

## 2.3 FPGA 运用领域

**汽车电子**

在发现可编程逻辑器件（PLD）非常有利于系统供应商和汽车生产商 （原始设备生产商） 获得成功之后，汽车行业开始大量采用这些器件。PLD 的质量是同类最佳的，成本结构适合大批量消费类应用，系统性能也在不断提高，因此，和其他半导体产品相比，PLD 能够更迅速地进入汽车市场领域。与 ASSP 和微控制器解决方案不同，PLD 的灵活性和产品迅速面市特性越来越成为汽车行业的关键需求。PLD 已经在信息娱乐和通信市场上得到了广泛应用，新兴的汽车辅助驾驶设计也采用了 PLD。在这一领域中，某些应用发展非常迅速，包括道路偏离报警、夜视和胎压监控系统等。PLD 具有较低的芯片成本结构、丰富的知识产权（IP）内核、参考设计以及较长的产品在市时间，是汽车电子市场发展的理想选择。

**消费电子**

发展迅速的消费类电子市场各式新产品层出不穷，让人耳目一新，例如平面显示器、便携式媒体播放器以及家庭联网产品等。这些产品的功能不断丰富，每年都有很大的改进。对采用最新技术的消费类电子生产商而言，如此迅速地发展给他们在时间带来了很大的竞争压力。

**计算机与存储**

计算机存储发展迅速。在传统的 IT 应用中，服务器和存储器直接互联，而现在已经进展为联网存储体系结构，即存储区域网（SAN）。SAN 很容易实现存储扩展，以前受内部存储能力限制的服务器可以在现有条件以外扩容。除了容易扩展存储之外，服务器也发展到能够迅速高效地实现数据处理。

创新的存储和服务器技术需要灵活的平台来迅速实现各种解决方案，而大批量应用市场更需要低成本方案。在计算机和存储设备中使用 Altera 的低成本可编程逻辑和结构化 ASIC 可以保证无风险地快速提高产量。在这些存储网络的推动下，流量和数据处理的迅猛增长导致迫切需要高性能芯片和接口技术 Stratix III FPGA 在体系结构上突出了丰富的存储器和串行接口，非常适合高端存储应用。

**军事与航空航天**

军事商用（COTS）计划的前提是军事项目能够采用商用元件、电路板和系统，充分发挥新技术和规模经济的优势。虽然 COTS 获得了一定的成功，但是还需要进行改进。商用供应商必须能够更好地满足军事和航空航天市场的需求

**医疗**

大部分医疗产品都采用了某种类型的半导体器件。实际上，半导体器件在这些产品中的应用越来越广泛。可编程逻辑器件（PLD）的普及率要远远高于其他类型的半导体

器件。在医疗设备开发中，PLD 是功能强大而且切实可行的 ASIC 和 ASSP 替代方案。在设计过程中，根据需要对 PLD 重新编程，避免了前端流片（NRE）成本，减少了与 ASIC 相关的订量，降低了芯片多次试制的巨大风险。

和 ASSP 相比，PLD 在设计上非常灵活，可实现电路板级集成，从而使产品在众多的竞争医疗设备生产商中脱颖而出。此外，随着标准的发展或者当需求出现变化时，还可以在现场更新 PLD。而且，设计人员能够反复使用公共硬件平台，在一个基本设计基础上，建立不同的系统，支持各种功能，从而大大降低了生成成本。不论是设计 CT 还是病人监控设备，可编程逻辑器件都能够成功实现系统设计，非常灵活，没有风险——和其他医疗设备生产商相比，不但性价比高，而且更能突出产品增值优势。

**无线通信**

近十年来互联网的爆炸性增长导致对大众化高速互联网接入技术的需求越来越大。无线互联网接入技术在家庭和办公室之外提供网络接入，满足了这种不断增长的需求。目前有很多无线应用方案，能够满足各种不同宽带无线接入技术具有很大的市场潜力，使得微波接入全球互通(WiMAX)技术越来越流行。WiMAX 802.16e-2005 支持城域网(MAN)范围内的移动高速互联网接入，使用正交频分复用接入(OFDMA)和多输入多输出(MIMO)技术等高级信号处理方案。WiMAX 802.16e-2005 作为一种固定无线技术，将在新网络实施以及新兴市场上扮演重要角色。随着半导体技术和信号处理技术的进步，无线标准和系统本身也在不断发展。这就需要一个可以提供较宽处理带宽，具有产品及时面市优势的灵活硬件平台来满足这些需求。

**工业**

灵活、可靠，并且能够在同一平台上支持多种标准，Altera FPGA 帮助您在工业自动化和过程控制产品中开发适应性强而且不会过时的设计方法。从可编程逻辑控制器（PLC）到运动/电机控制器、I/O 模块、人机接口（HMI)/操作面板和智能驱动器，在开发过程中甚至在现场，您都可以对产品重新进行配置。而且，您还降低了总成本，进一步提高了效能。Altera 器件在工业温度范围内完全能够正常工作，极冷或者极热都不会影响性能，您还可以将知识产权（IP)/解决方案移植到未来的产品系列中。

## 2.4 FPGA 发展前景

据市场调研公司 Gartner Dataquest 预测，2010 年 FPGA 和其它可编程逻辑器件(PLD)市将从 2005 年的 32 亿美元增长到 67 亿美元，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，所以未来 FPGA 将会是一个非常有前景的行业。

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，例如，Actel 的 Nano 系列更是打破了 FPGA 的价格屏障，提供超过 50 种低于 1 美金的 FPGA，在一定程度上已经可以与 ASIC 相抗衡。

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。

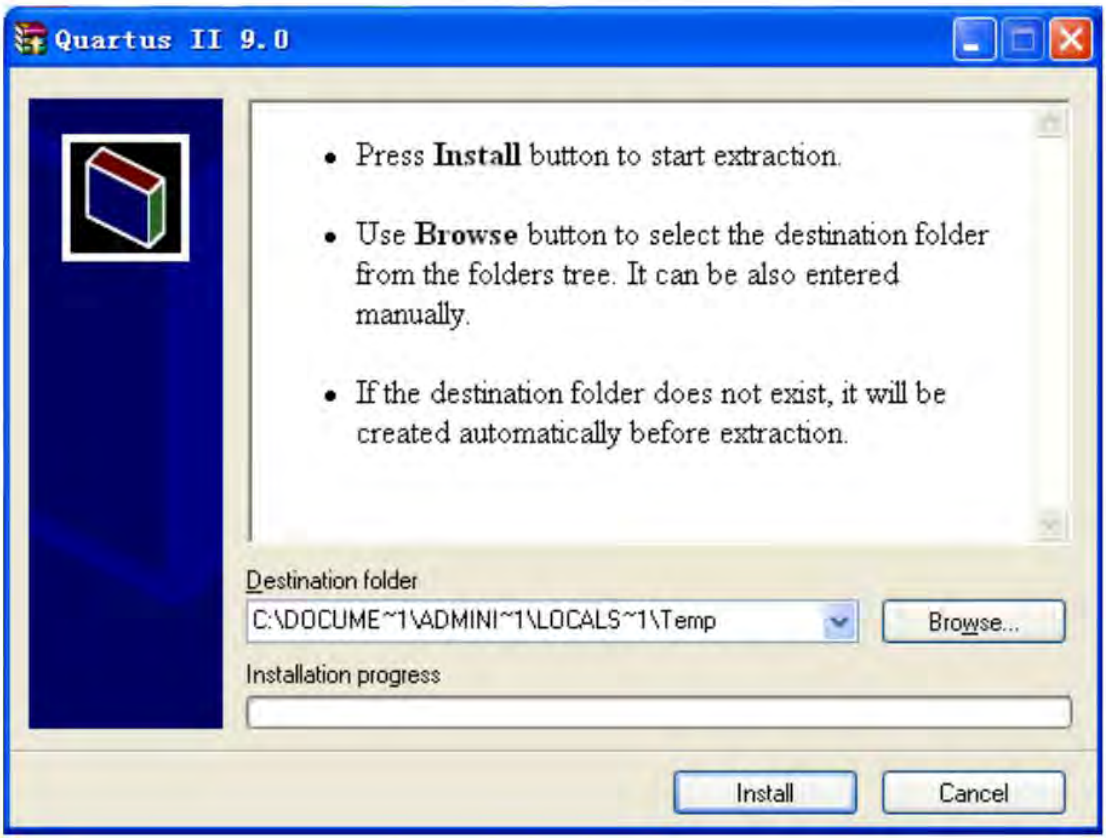
# 第三章配套软件的安装和使用

## 软件安装

安装流程：

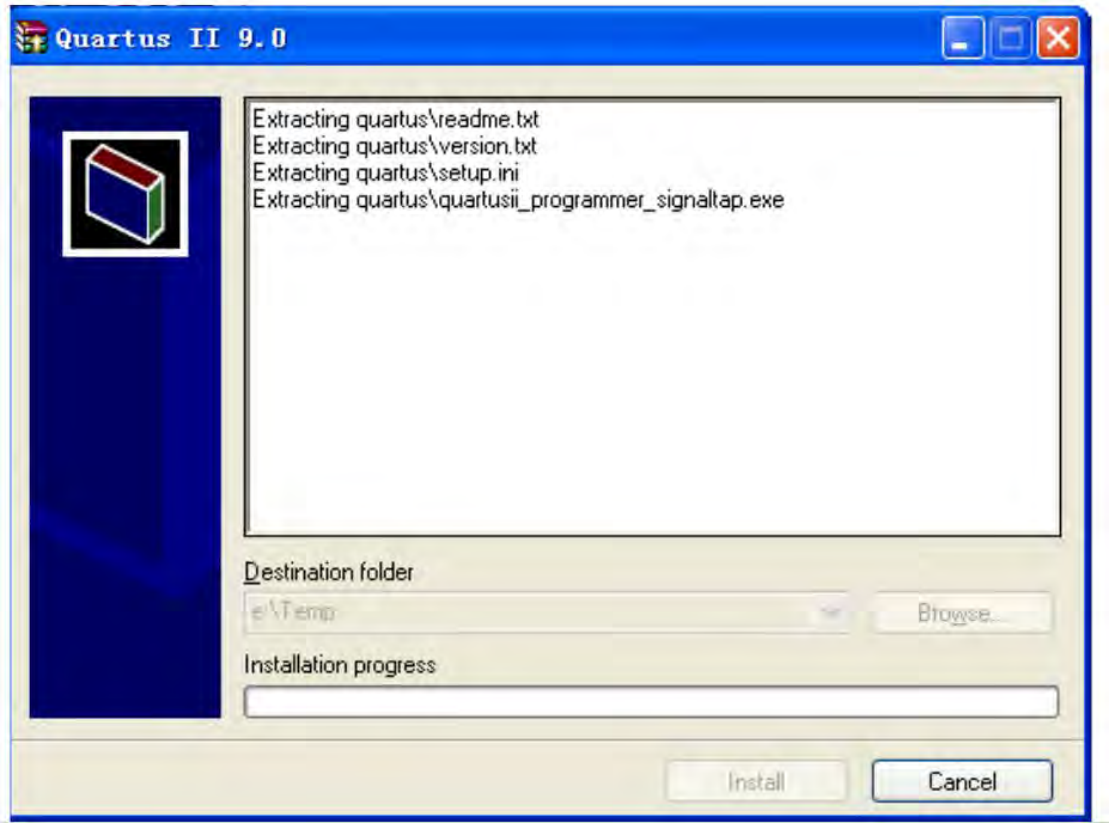
**第一步：解压安装文件**

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：



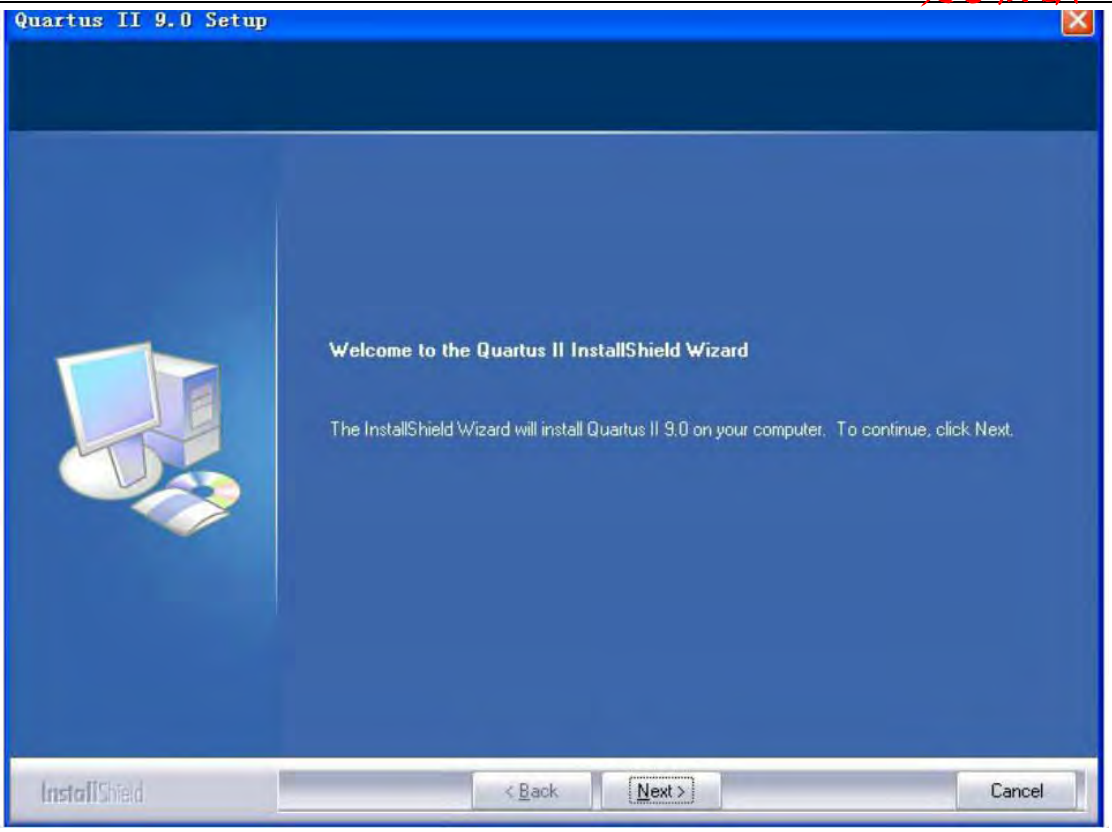

C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp 这个路径仅仅是一个临时的解压路径，您可以选择的临时其它目录，待安装完毕后方便删除掉此临时目录，接着点击 Install 进入第二步

**第二步：安装程序正在解压文件到临时目录中。**



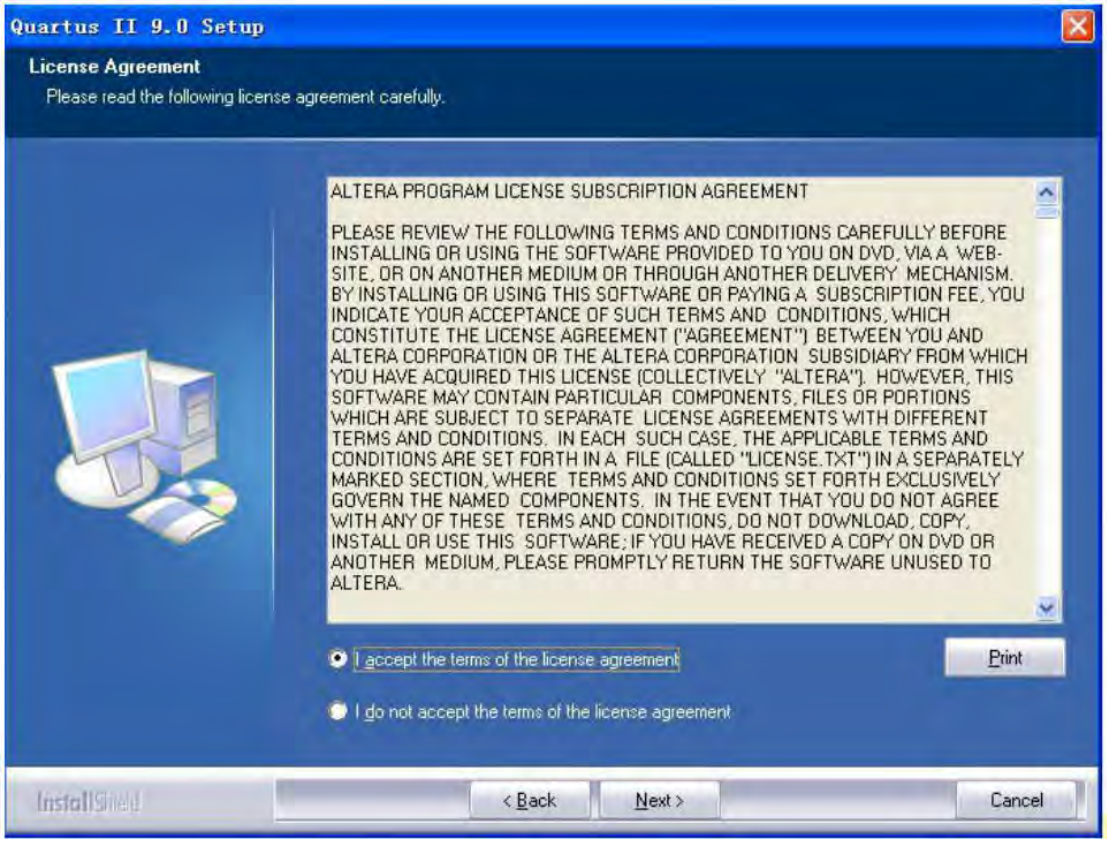

第三步：解压完成后，出现开始安装向导窗口


第四步：出现 QuartusII 9.0 安装界面

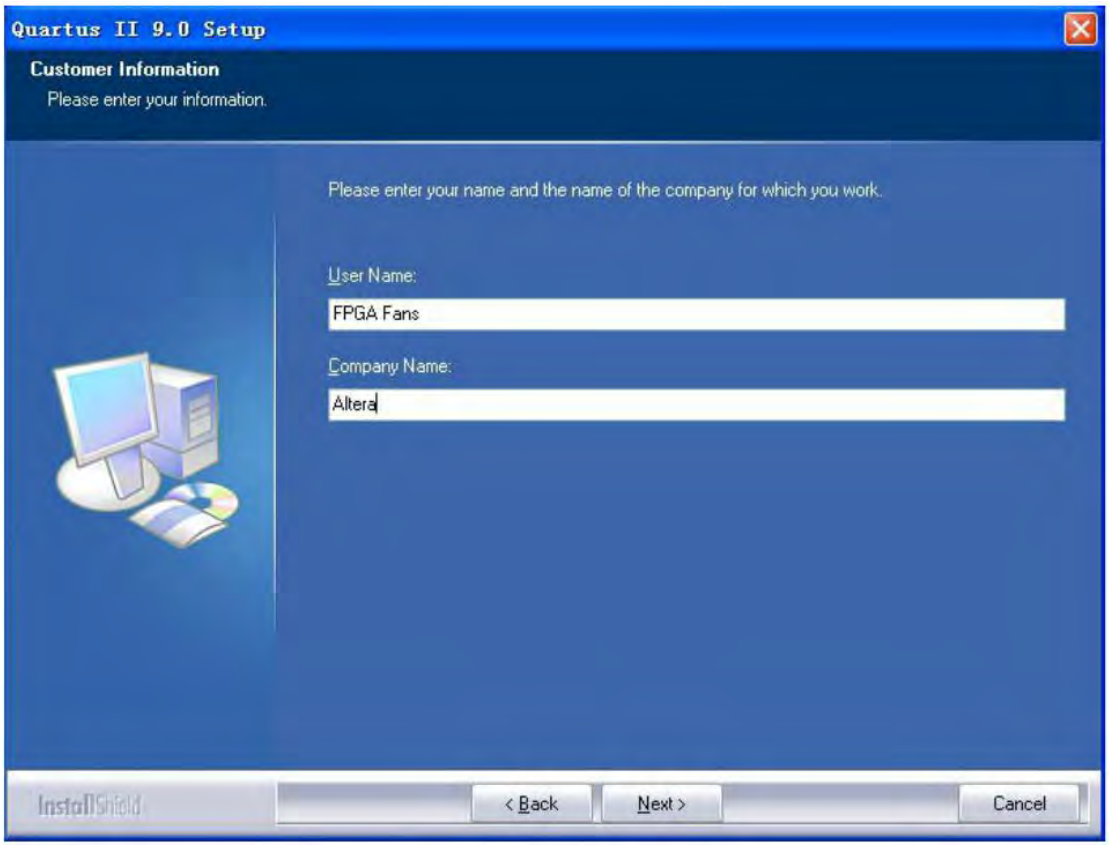



点击 next

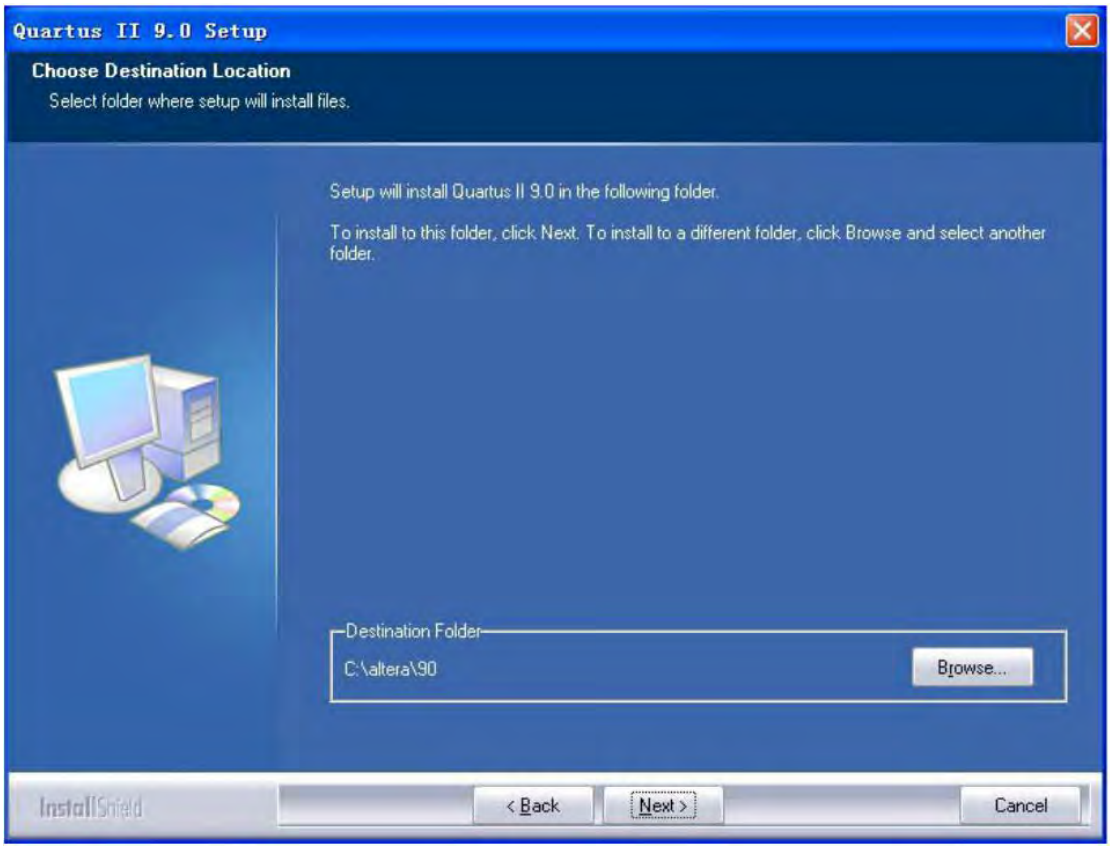
**第五步：出现授权窗口点击 I accept the term of the license agreement.后，再点击下一步 next**



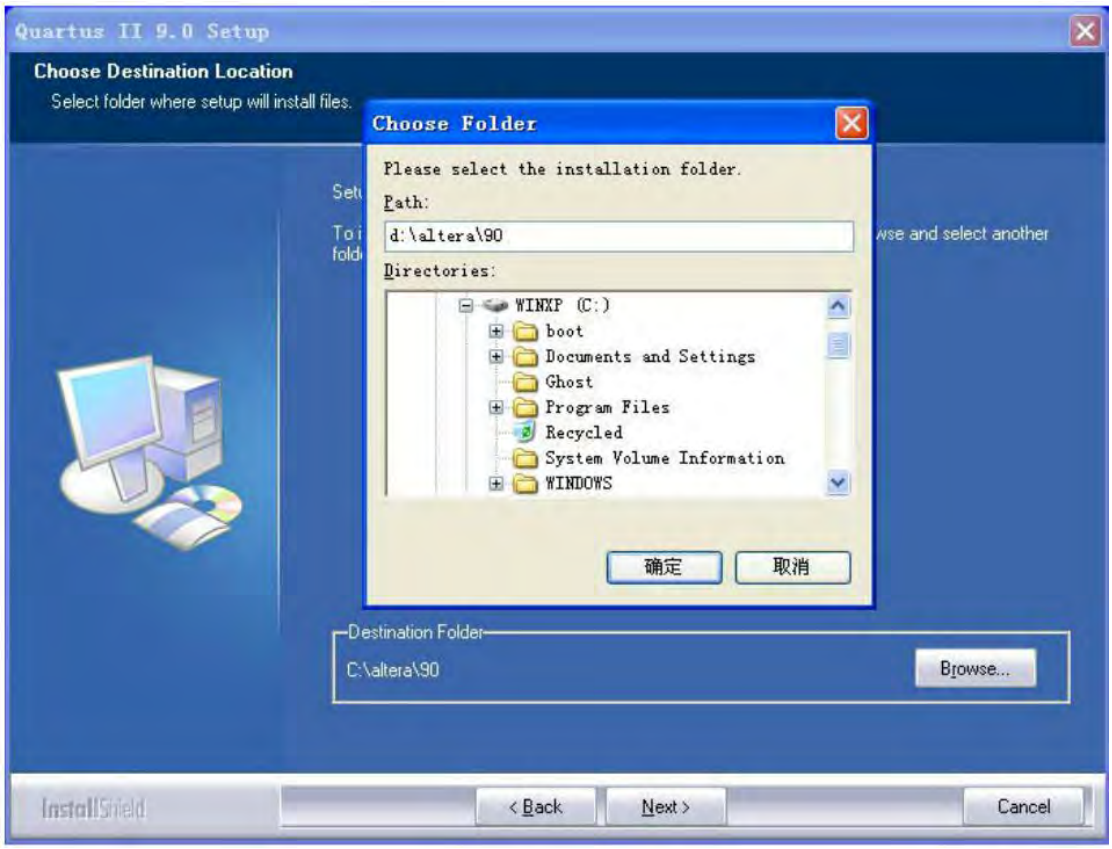
第六步：接下来输入用户名和公司名，如用户名为 **FPGA Fans**，公司名为 **Altera**，点击下一步 **next**



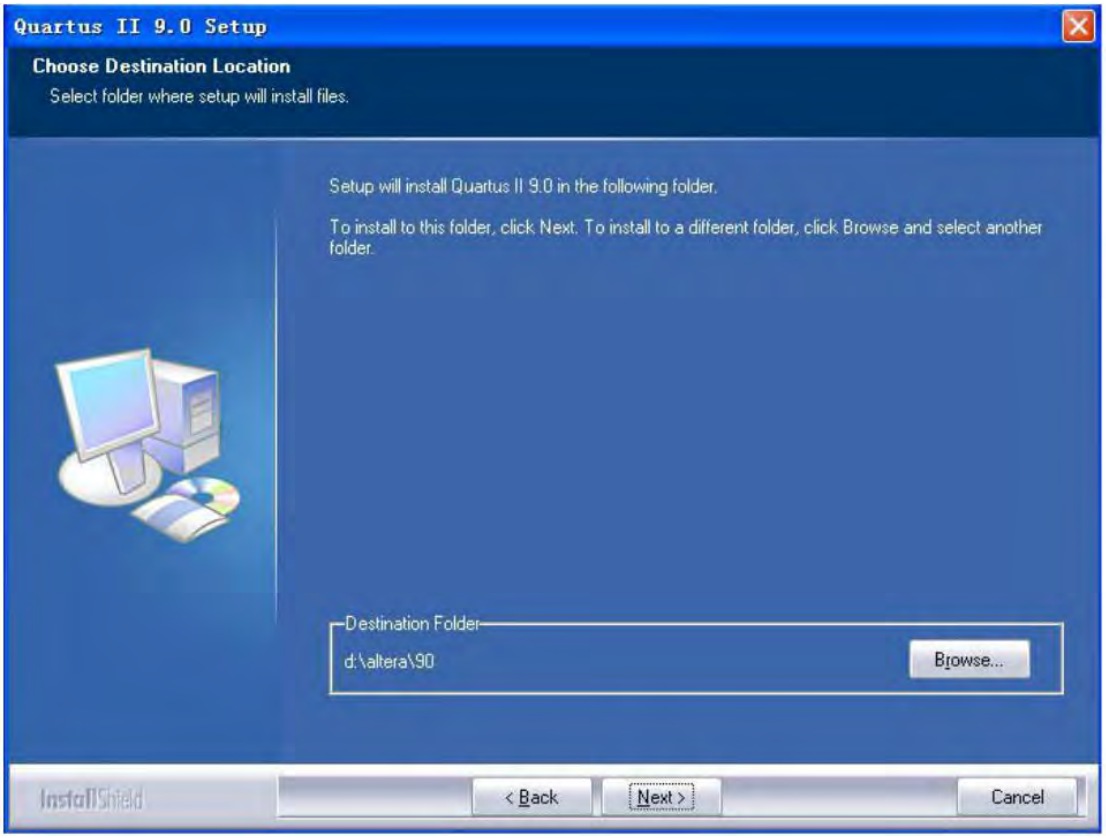

第七步：选择安装目录，可以自由选择安装目录（建议目录名为英文）



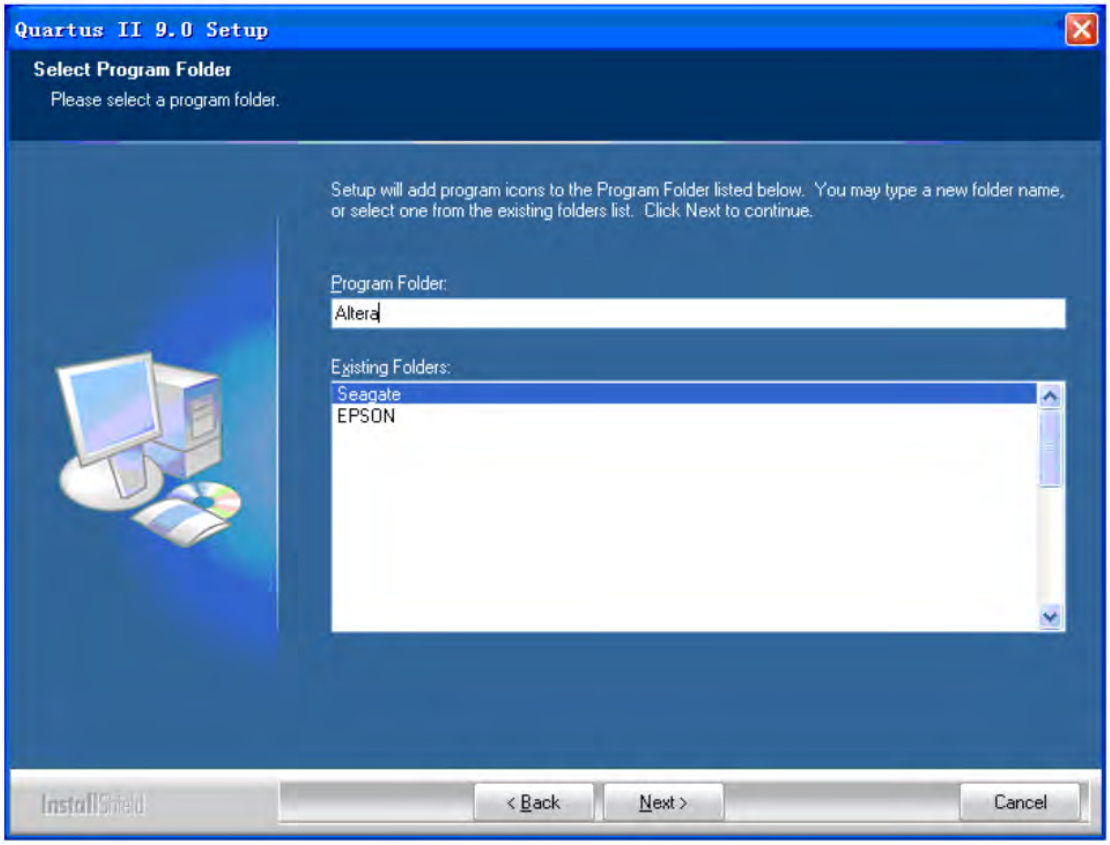

第八步：您可以点击 **Browse…**按钮来自行选择安装路径。



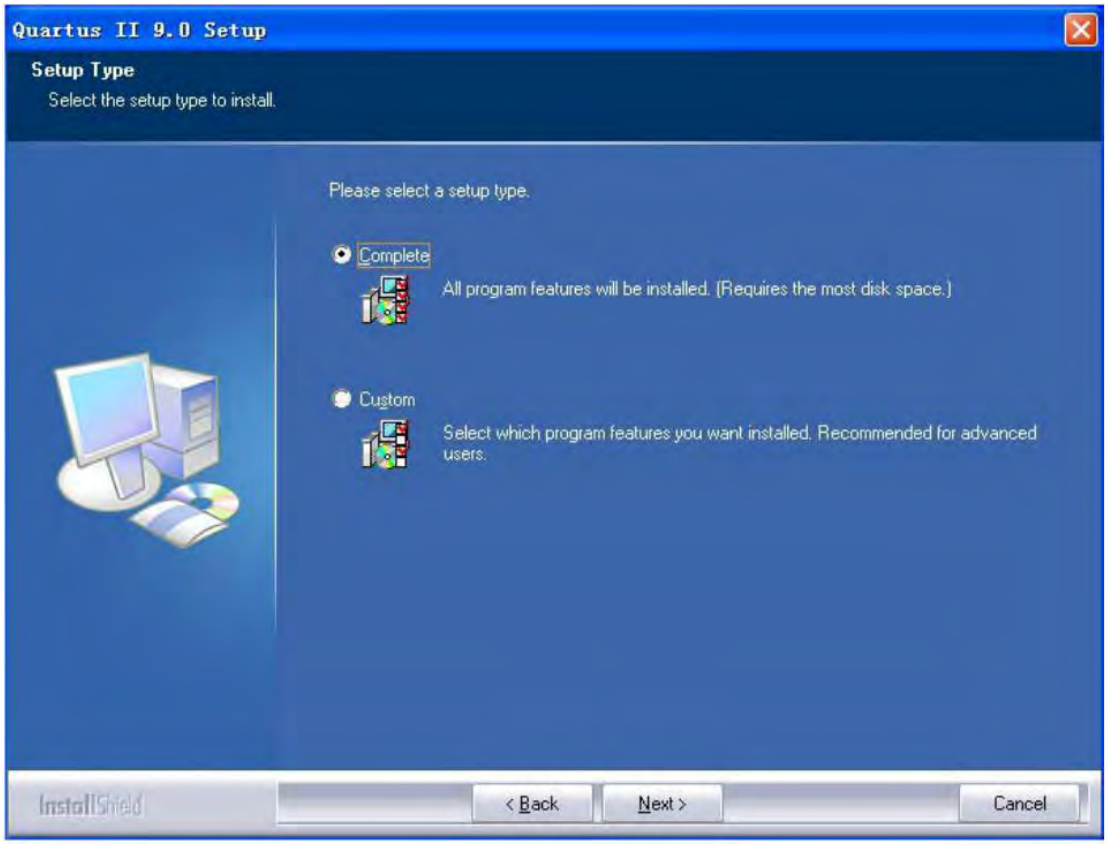
选择好后，点击确定，再点击 next 进入下一步的安装。



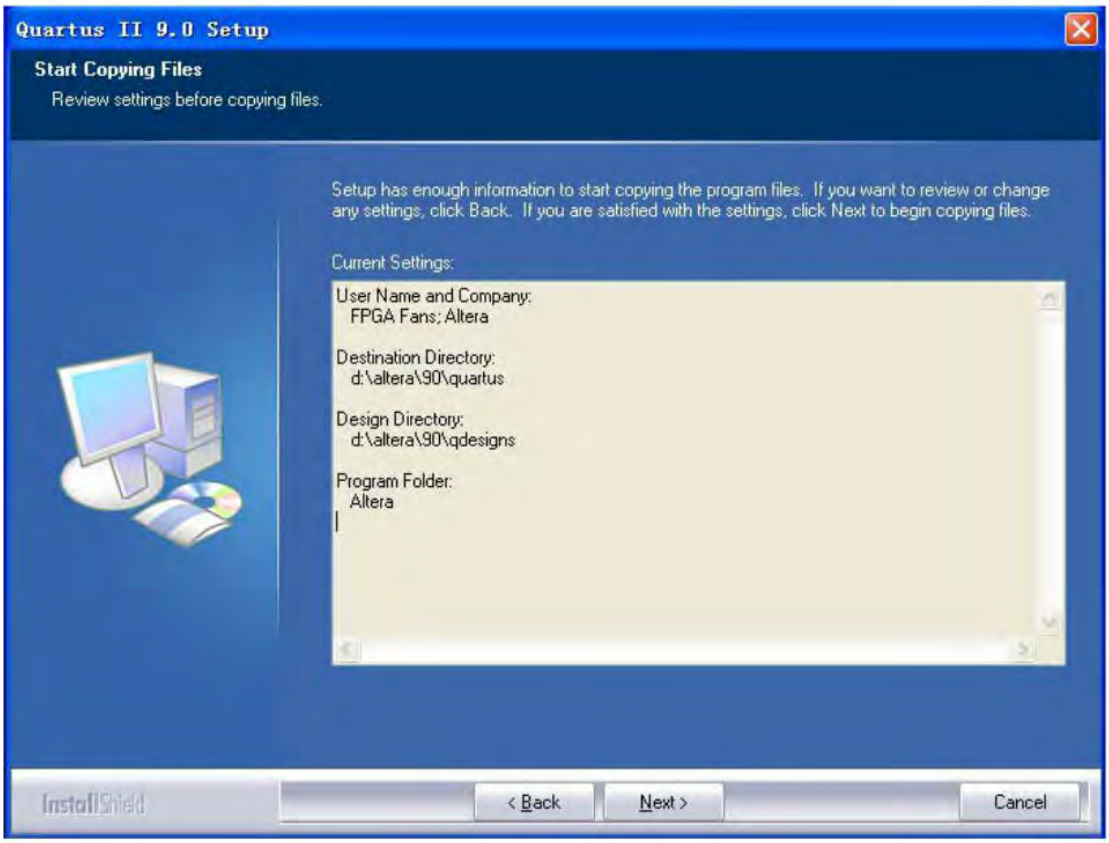

**第九步：默认开始菜单的软件名称，默认为 Altera,点击 next 下一步按钮继续安装**



第十步:安装类型的选择，我们推荐采用完全安装。

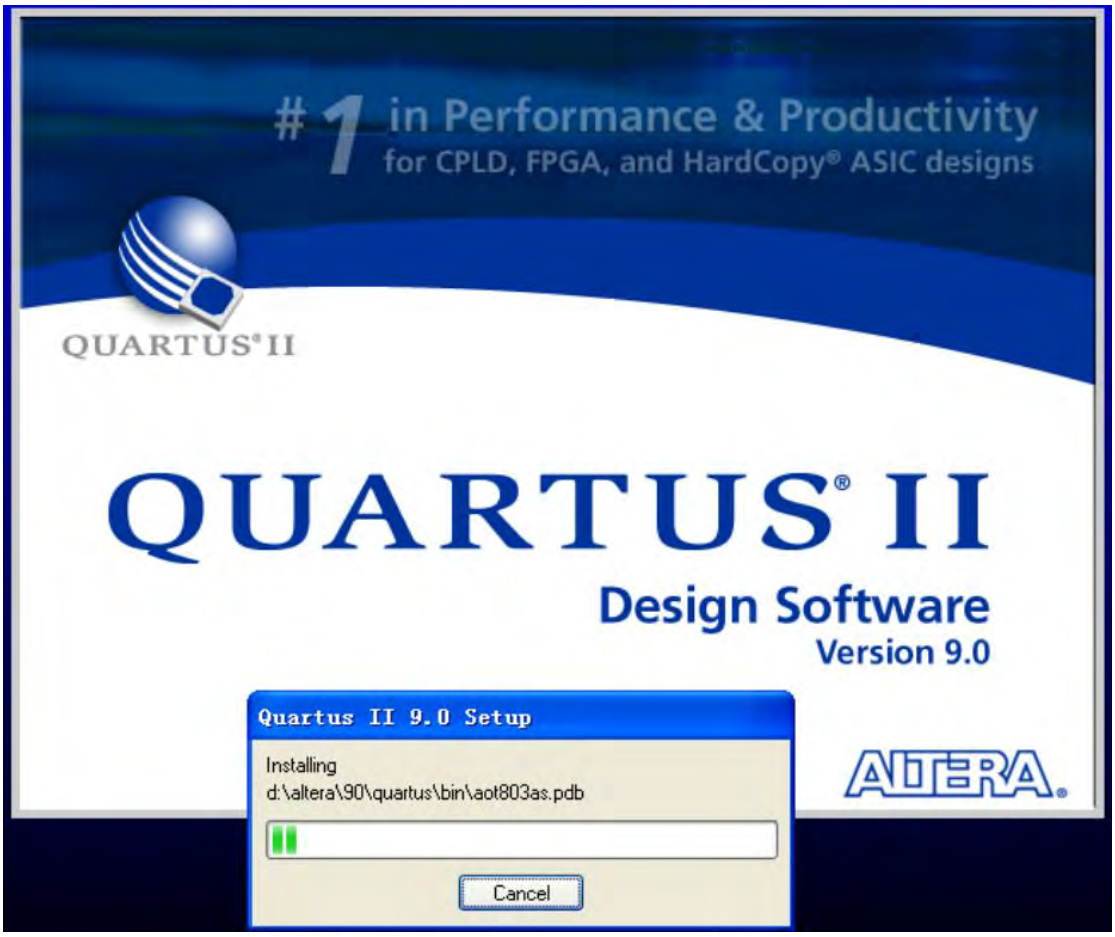



点击 next,进入下一步。

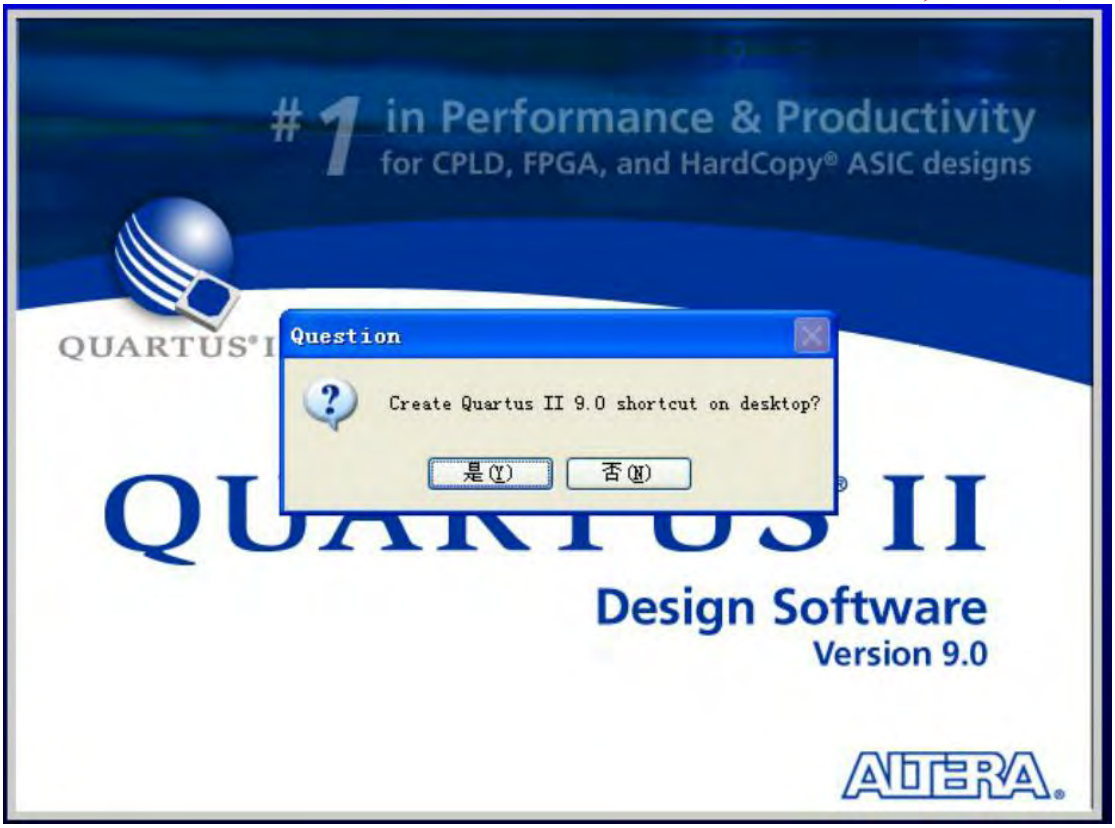


确认安装目录，点击 next 进入下一步。

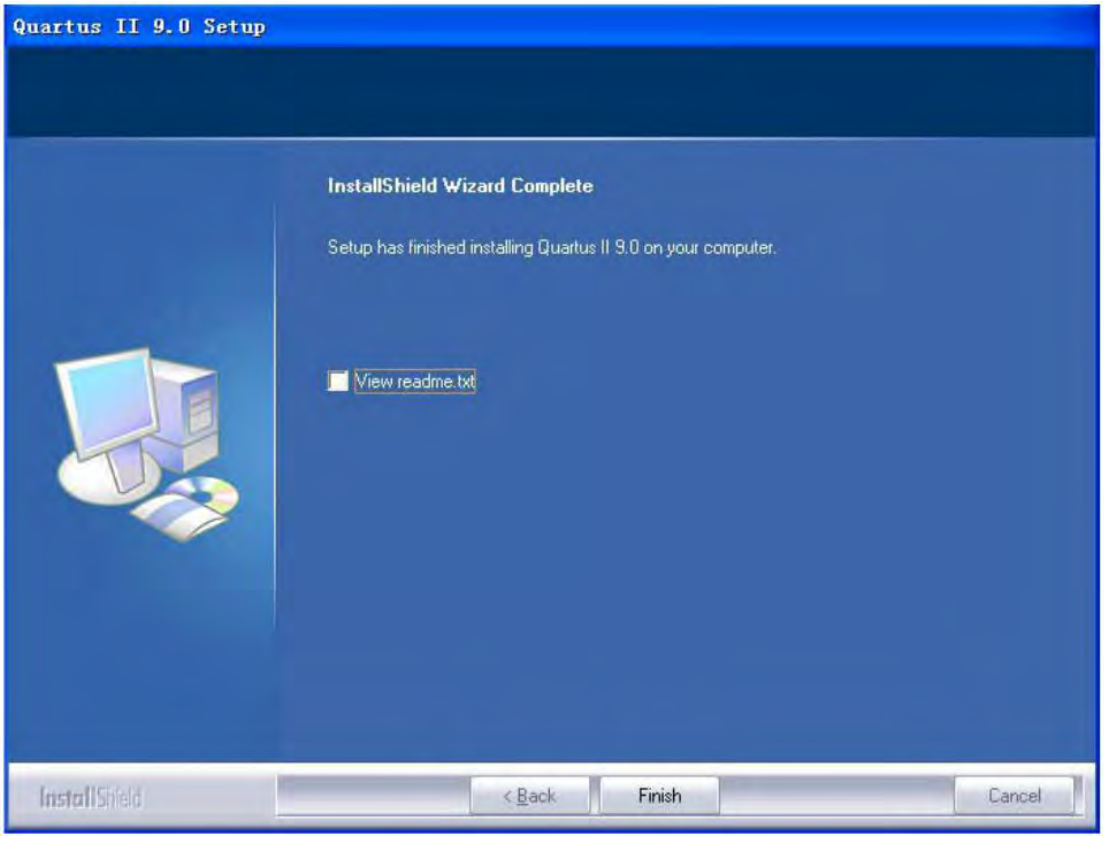
**第十一步：开始安装，此步骤需要的时间最多，请耐心等待。**




**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”**




点击 Finish,完成安装。




破解步骤：

（1） 用 Quartus_II_9.0 破解器.exe 破解 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 XXXXXXXXXXX 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 License file，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。

# 第四章 FPGA 芯片资源的介绍

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 4,608 到 68,416 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。表 1 所示为 Cyclone II FPGA 系列的特性和能力。

EP2C5T1448N:“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。

表一：Cyclone II FPGA 简介:

表 1. Cyclone II FPGA 简介

| 器件 | EP2C5 | EP2C8 | EP2C15 | EP2C20 | EP2C35 | EP2C50 | EP2C70 |
|---|---|---|---|---|---|---|---|
| 逻辑单元 | 4,608 | 8,256 | 14,448 | 18,752 | 33,216 | 50,528 | 68,416 |
| M4K RAM 块（4 k 比特 + 512 校验比特） | 26 | 36 | 52 | 52 | 105 | 129 | 250 |
| 总比特数 | 119,808 | 165,888 | 239,616 | 239,616 | 483,840 | 594,432 | 1,152,000 |
| 嵌入式 18x18 乘法器 | 13 | 18 | 26 | 26 | 35 | 86 | 150 |
| PLLs | 2 | 2 | 4 | 4 | 4 | 4 | 4 |
| 最多用户 I/O 管脚 | 158 | 182 | 315 | 315 | 475 | 450 | 622 |
| 差分通道 | 58 | 77 | 132 | 132 | 205 | 193 | 262 |
| Cyclone II "A" 快速接通型 (1) | 是(3) | 工业汽车电子 | 是(2) | 工业汽车电子 | N/A | N/A | N/A |

表 1 注释：

某些器件的快速接通型号具有较短的上电复位（POR）时间。这一特性使 FPGA 能够比普通 Cyclone II 器件更迅速地进入工作状态。这些器件在订购码（例如，EP2C8 A F256I8N）中以字母“A”标出；

EP2C15 器件只提供 EP2C15A 快速接通型；

EP2C5A 器件提供汽车级版本。

表二：Cyclone II 器件封装和用户 I/O 管脚数：

表 2. Cyclone II 器件封装和最多用户 I/O 管脚

| 封装尺寸 (mm x mm) | EP2C5 | EP2C8 | EP2C15 | EP2C20 | EP2C35 | EP2C50 | EP2C70 |
|---|---|---|---|---|---|---|---|
| 144-Pin TQFP (1) (22 x 22) | 89 | 85 | – | – | – | – | – |
| 208-Pin PQFP (2) (30.6 x 30.6) | 142 | 138 | – | – | – | – | – |
| 240-Pin PQFP (32 x 32) (3) | – | – | – | 142 | – | – | – |
| 256-Pin FineLine BGA (4) (17 x 17) | 158 | 182 | 152 | 152 | – | – | – |
| 484-Pin Ultra FineLine BGA (19 x 19) | – | – | – | – | 322 | 294 | – |
| 484-Pin FineLine BGA (23 x 23) | – | – | 315 | 315 | 322 | 294 | – |
| 672-Pin FineLine BGA (27 x 27) | – | – | – | – | 475 | 450 | 422 |
| 896-Pin FineLine BGA (31 x 31) | – | – | – | – | – | – | 622 |

表二注释：

TQFP = 薄四方扁平封装；

PQFP = 塑封四方扁平封装；

BGA = 球栅阵列封装。

表 3. Cyclone II FPGA 的适用配置器件

| 配置器件 | 支持 Cyclone II 器件 | | | | | | |
|---|---|---|---|---|---|---|---|
| | EP2C5 | EP2C8 | EP2C15 | EP2C20 | EP2C35 | EP2C50 | EP2C70 |
| EPCS1 | X | | | | | | |
| EPCS4 | X | X | X | X | | | |
| EPCS16 | X | X | X | X | X | X | X |
| EPCS64 | X | X | X | X | X | X | X |

表三：Cyclone II 器件的适用配置器件：

Cyclone II 器件系列特性：

Altera 的 Cyclone® II FPGA 系列是低成本 90-nm 解决方案，它前所未有地提高了设计灵活性，实现了高性能系统集成（ 请参见表四）。

表 1. Cyclone II 特性一览

| 特性 | 说明 |
|---|---|
| 成本优化的架构 | Cyclone II 器件架构为最低的成本而优化，提供多达 68,416 个逻辑单元(LE)，密度超过第一代 Cyclone FPGA 的 3 倍。Cyclone II FPGA 内部的逻辑资源可以用来实现复杂的应用。 |
| 高性能 | Cyclone II FPGA 要比竞争低成本 90-nm FPGA 快 60%，是市场上性能最好的低成本 90-nm FPGA。 |
| 低功耗 | Cyclone II FPGA 功耗只有竞争低成本 90-nm FPGA 的一半，大大降低了静态和动态功耗。 |
| 工艺技术 | Cyclone II FPGA 在 300 毫米晶圆的基础上，采用了 TSMC 领先的 90nm 低电介工艺技术而生产。 |
| 嵌入式存储器 | Cyclone II FPGA 基于流行的 M4K 存储器块，提供多达 1.1 兆比特的嵌入式存储器，可以支持配置为广泛的操作模式，包括 RAM、ROM、先入先出（FIFO）缓冲器以及单端口和双端口模式。 |
| 嵌入式乘法器 | Cyclone II FPGA 提供最多 150 个 18x18 比特乘法器，是低成本数字信号处理（DSP）应用的理想方案。这些乘法器可用于实现通用 DSP 功能，如有限冲击响应（FIR）滤波器、快速傅立叶变换、相关器、编/解码器以及数控振荡器（NCO）。 |
| 外部存储器接口 | Cyclone II 器件提供高级外部存储器接口支持，允许开发人员集成外部单倍数据速率（SDR）、双倍数据速率（DDR）、DDR2 SDRAM 器件以及第二代四倍数据速率（QDRII）SRAM 器件，数据速率最高可达 668 Mbps。 |
| 差分 I/O 支持 | Cyclone II 器件提供差分信号支持，包括 LVDS、RSDS、mini-LVDS、LVPECL、SSTL 和 HSTL I/O 标准。LVDS 标准支持接收端最高 805 Mbps 数据速率，发送端最高 622 Mbps。 |
| 单端 I/O 支持 | Cyclone II 器件支持各种单端 I/O 标准，如当前系统中常用的 LVTTL、LVCMOS、SSTL、HSTL、PCI 和 PCI-X 标准。 |
| 接口和协议支持 | Cyclone II 器件支持串行总线和网络接口（如 PCI 和 PCI-X），快速访问外部存储器件，同时还支持大量通讯协议，包括以太网协议和通用接口。 |
| 时钟管理电路 | Cyclone II 器件支持最多达四个可编程锁相环（PLL） |

| | |
|---|---|
| | 和最多 16 个全局时钟线，提供强大的时钟管理和频率合成能力，使系统性能最大化。这些 PLL 提供的高级特性包括频率合成、可编程占空比、外部时钟输出、可编程带宽、输入时钟扩频、锁定探测以及支持差分输入输出时钟信号。 |
| Nios® II 嵌入式处理器 | Cyclone II 器件的 Nios II 嵌入式处理器降低了成本，提高了灵活性，给低成本分立式微处理器提供了一个理想的替代方案。 |
| 片内匹配 | Cyclone II FPGA 支持驱动阻抗匹配和片内串行终端匹配。片内匹配消除了对外部电阻的需求，提高了信号完整性，简化电路板设计。Cyclone II FPGA 通过外部电阻还可支持并行匹配和差分匹配。 |
| 快速接通能力 | Cyclone II FPGA 具有快速接通能力，上电后能够迅速工作，是汽车等需要快速启动的应用的理想选择。在器件订购码中以“ A ”表示具有较短上电复位 (POR) 时间的 Cyclone II FPGA (EP2C5A、 EP2C8A、 EP2C15A 和 EP2C20A) 。 |
| 热插拔及上电顺序 | Cyclone II 器件提供强大的片内热插拔以及上电顺序支持，确保器件正确操作不依赖上电顺序。该特性同时实现了上电之前和上电过程中对器件和三态 I/O 缓冲的保护，使 Cyclone II 器件成为多电压系统以及具高可靠性和冗余需求的应用的理想方案。 |
| 循环冗余码(CRC) | Cyclone II 器件具有 32 比特 CRC 自动校验功能。内置的 CRC 校验电路简化了校验流程，只需在 Quartus II 软件中单击一下即可。这是 FPGA 中对付单事件干扰（SEU）问题最有效的解决方案。 |
| Cyclone II 与 Cyclone FPGA 的差别 | Cyclone II FPGA 提供比 Cyclone FPGA 更新更先进的特性。这些特性包括嵌入式乘法器、支持 DDR2 和 QDR II 存储器件的外部存储器接口、片内串行匹配，以及支持更多的差分和单端 I/O 标准。 |
| 串行配置器件 | Cyclone II 器件可以采用 Altera 的低成本串行配置器件进行配置，这种串行配置器件最大可提供 64 兆比特的 Flash 存储器。 |

# 第五章 FPGA 芯片内部硬件介绍

FPGA（Filed programmable gate device）：现场可编程逻辑器件

FPGA 基于查找表加触发器的结构，采用 SRAM 工艺，也有采用 flash 或者反熔丝工艺；主要应用高速、高密度大的数字电路设计。

FPGA 由可编程输入/输出单元、基本可编程逻辑单元、嵌入式块 RAM、丰富的布线资源（时钟/长线/短线）、底层嵌入功能单元、内嵌专用的硬核等组成；

目前市场上应用比较广泛的 FPGA 芯片主要来自 Altera 与 Xilinx。另外还有其它厂家的一些低端芯片(Actel、Lattice)。

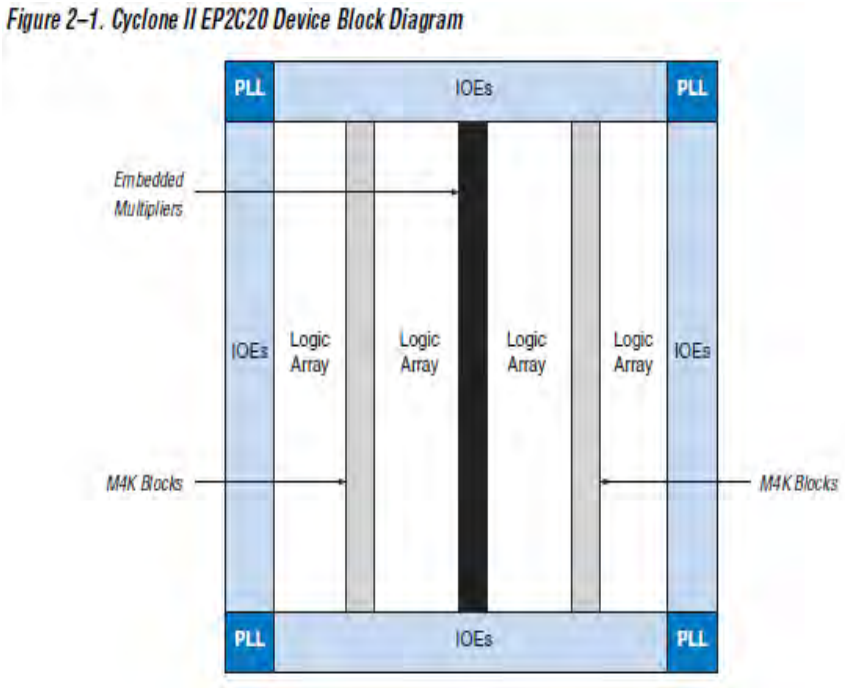
这里主要介绍 AlteraCyclone II 系列 FPGA 的内部硬件结构：

1. FPGA 器件结构
2. 可编程输入/输出单元 IOE
3. 可编程逻辑单元 LE
4. 嵌入式块 RAM
5. 布线资源
6. 底层嵌入功能单元

1、 FPGA 器件结构

Altera cyclone II 器件结构

*Figure 2–1. Cyclone II EP2C20 Device Block Diagram*




FPGA 的内部结构包括：

（1） 可编程逻辑门阵列，由最小单元 LE 组成

（2） 可编程输入输出单元 IOE

（3） 嵌入式 RAM 块，为 M4K 块，每个的存储量为 4K，掉电丢失

（4） 布线网络

（5） PLL 锁相环，EP2C5T144C8N 最大的倍频至 250MHz,这也是该芯片的最大工作频率

2、 可编程输入/输出单元 IOE

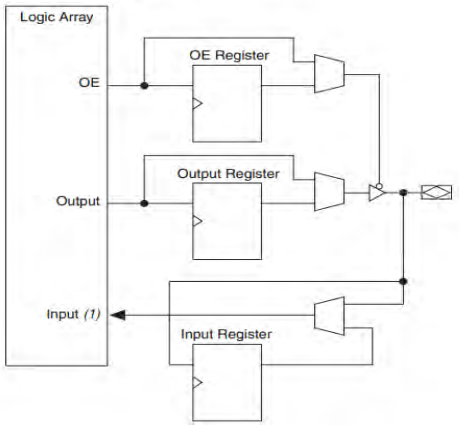
可编程 I/O，可配置成 OC 门、三态门、双向 IO、单端 / 差分等；支持各种不同的 I/O

标准：LVTTL、LVCOMS、SSTL、LVDS、HSTL、PCI 等；

| I/O Standard | Type | $V_{CCIO}$ Level | | Top & Bottom I/O Pins | | Side I/O Pins | | |
|---|---|---|---|---|---|---|---|---|
| | | Input | Output | CLK, DQS | User I/O Pins | CLK, DQS | PLL_OUT | User I/O Pins |
| 3.3-V LVTTL and LVCMOS | Single ended | 3.3 V/ 2.5 V | 3.3 V | ✓ | ✓ | ✓ | ✓ | ✓ |
| 2.5-V LVTTL and LVCMOS | Single ended | 3.3 V/ 2.5 V | 2.5 V | ✓ | ✓ | ✓ | ✓ | ✓ |
| 1.8-V LVTTL and LVCMOS | Single ended | 1.8 V/ 1.5 V | 1.8 V | ✓ | ✓ | ✓ | ✓ | ✓ |
| 1.5-V LVCMOS | Single ended | 1.8 V/ 1.5 V | 1.5 V | ✓ | ✓ | ✓ | ✓ | ✓ |
| SSTL-2 class I | Voltage referenced | 2.5 V | 2.5 V | ✓ | ✓ | ✓ | ✓ | ✓ |
| SSTL-2 class II | Voltage referenced | 2.5 V | 2.5 V | ✓ | ✓ | ✓ | ✓ | ✓ |
| SSTL-18 class I | Voltage referenced | 1.8 V | 1.8 V | ✓ | ✓ | ✓ | ✓ | ✓ |
| SSTL-18 class II | Voltage referenced | 1.8 V | 1.8 V | ✓ | ✓ | (1) | (1) | (1) |
| HSTL-18 class I | Voltage referenced | 1.8 V | 1.8 V | ✓ | ✓ | ✓ | ✓ | ✓ |
| HSTL-18 class II | Voltage referenced | 1.8 V | 1.8 V | ✓ | ✓ | (1) | (1) | (1) |
| HSTL-15 class I | Voltage referenced | 1.5 V | 1.5 V | ✓ | ✓ | ✓ | ✓ | ✓ |
| HSTL-15 class II | Voltage referenced | 1.5 V | 1.5 V | ✓ | ✓ | (1) | (1) | (1) |
| PCI and PCI-X (2) | Single ended | 3.3 V | 3.3 V | | | ✓ | ✓ | ✓ |
| Differential SSTL-2 class I or class II | Pseudo differential (3) | (4) | 2.5 V | | | | ✓ | |
| | | 2.5 V | (4) | ✓ (5) | | ✓ (5) | | |
| Differential SSTL-18 class I or class II | Pseudo differential (3) | (4) | 1.8 V | | | | ✓ (6) | |
| | | 1.8 V | (4) | ✓ (5) | | ✓ (5) | | |

Altera 器件 IOE 结构

*Figure 2–20. Cyclone II IOE Structure*




Altera 器件的输入输出结构：可配置成三态、输入 / 输出、双向 IO

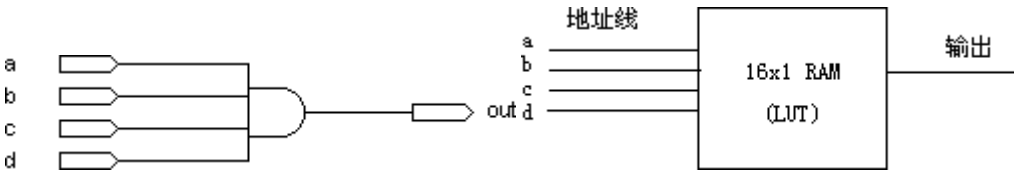
Altera 器件中 cyclone 系列中的 IOE 结构是基本的输入、输出、使能的触发器结构。

3、 可编程逻辑单元 LE

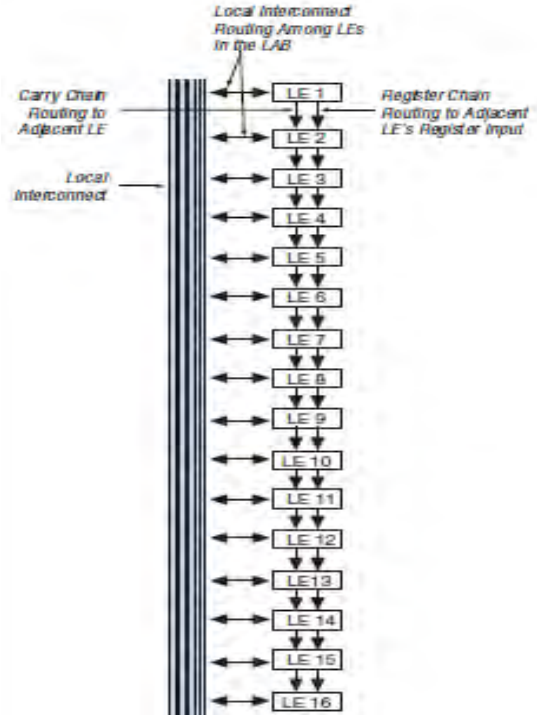
**基本可编程逻辑单元 LE** 由查找表(Look up table)触发器(FF)组成，而 LE 是组成 LAB 的最小单元；

LUT 一般是 4 输入查找表，高端器件（xilinx v5）采用 LUT-6 结构；LUT 可看成 4 位地址线的 16x1 的 RAM 结构。

FF 是可编程的触发器，可配置成同步/异步复位、同步/异步置位、使能、装载等功能触发器。




**可编程逻辑块：**




Altera:LAB

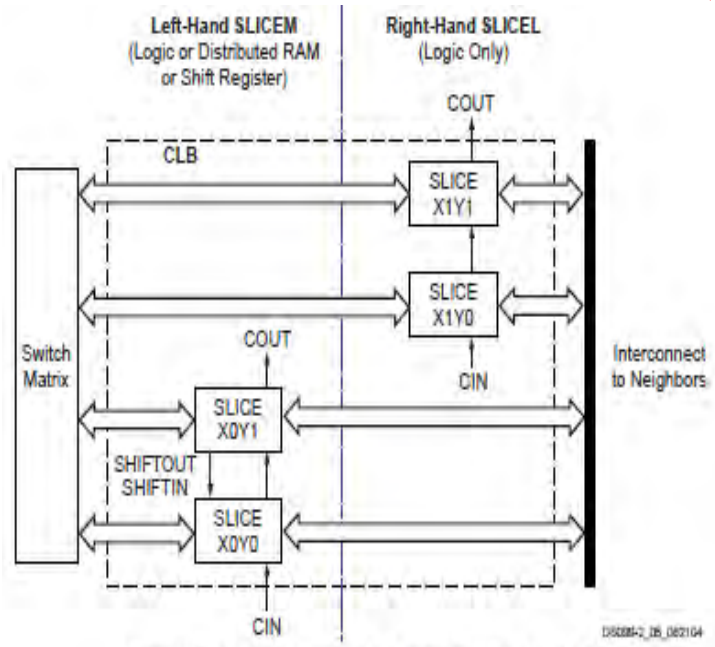
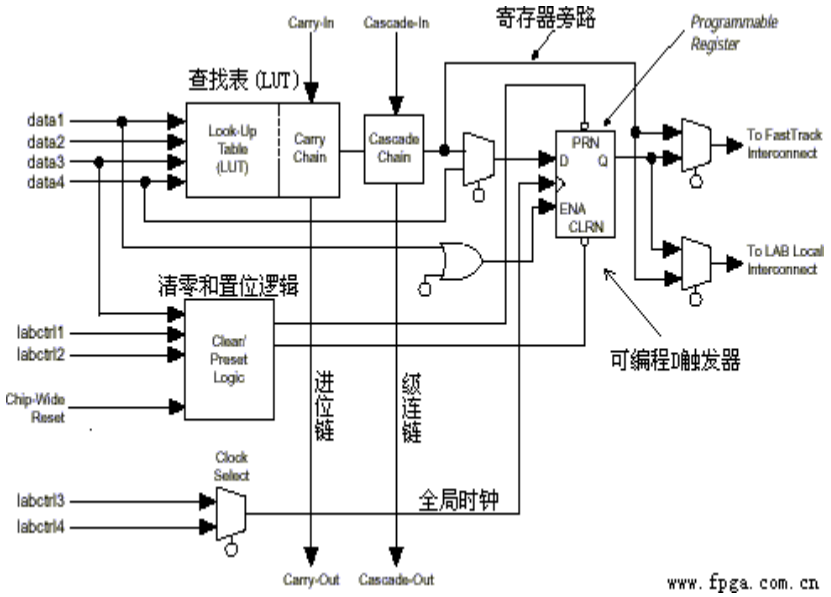



*Figure 9:* **Arrangement of Slices within the CLB**

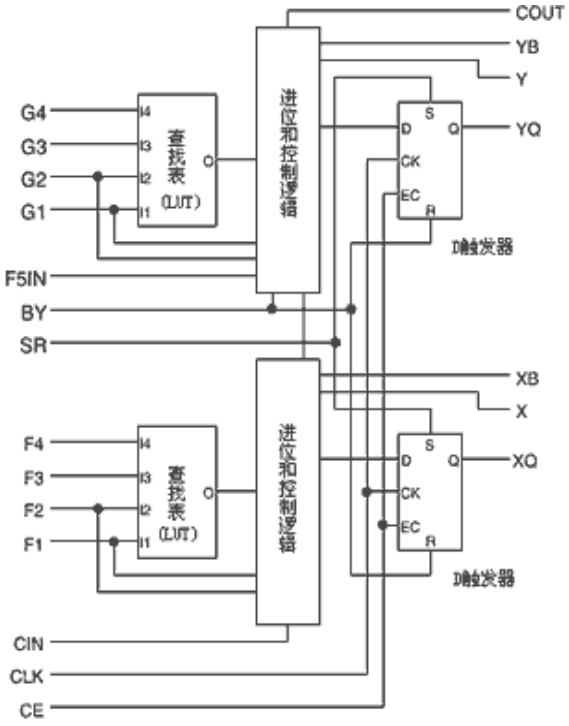
Xilinx:CLB

Xilinx CLB 由四个 SLICE 构成；而 Altera 的 LAB 由 16 个/8 个 LE 构成；

**基本逻辑单元 LE/SLICE：**




Altera:LE




Xilinx:SLICE

xilinx 为 SLICE:包括两个 LUT-4/两个 FF;

altera 为 LE:包括一个 LUT-4/一个 FF;

4、 嵌入式块 RAM

嵌入式块 RAM 可配置单/双端口 RAM、伪双端口 RAM、ROM、FIFO、SHIFT、CAM 等；不同厂家的块 RAM 大小不一样：

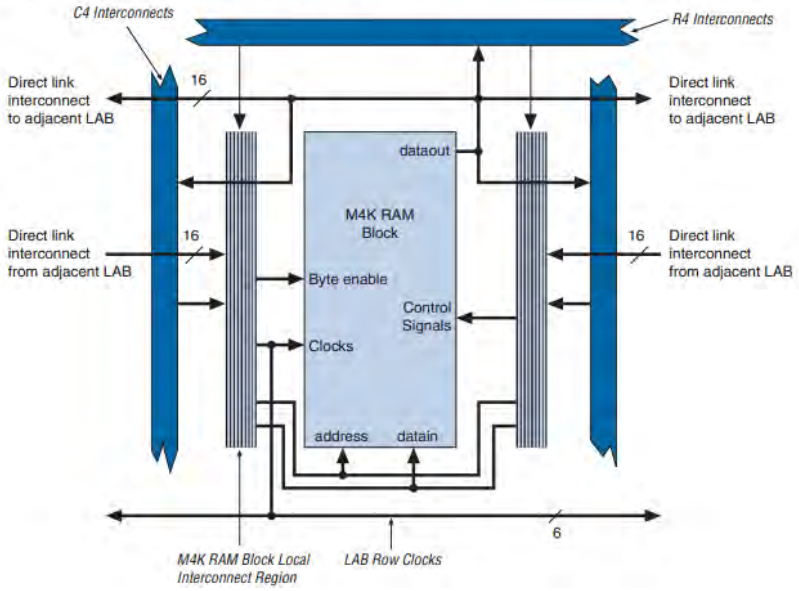
Altera：M512、M4K M4K、M-RAM(512K)；

Xilinx：18kbit；

Lattic：9kbit；

**Altera：M4K：**

Figure 2–17. M4K RAM Block LAB Row Interface



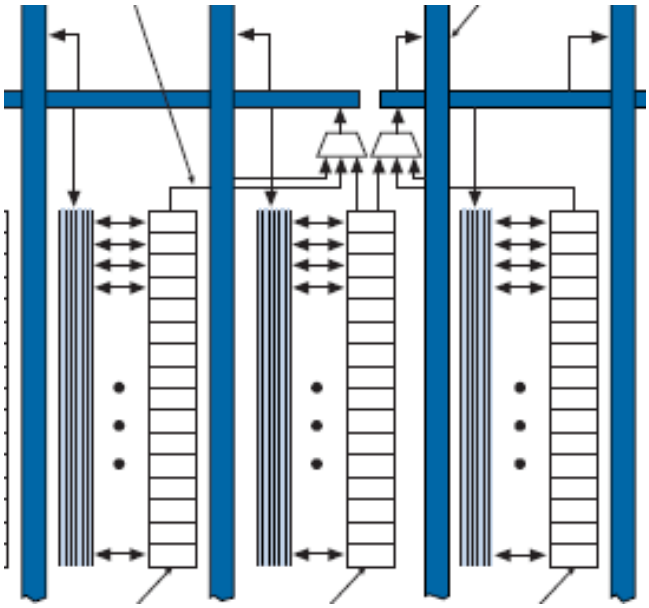

5、 布线资源

全局布线资源：用于全局时钟/全局复位/全局置位布线；

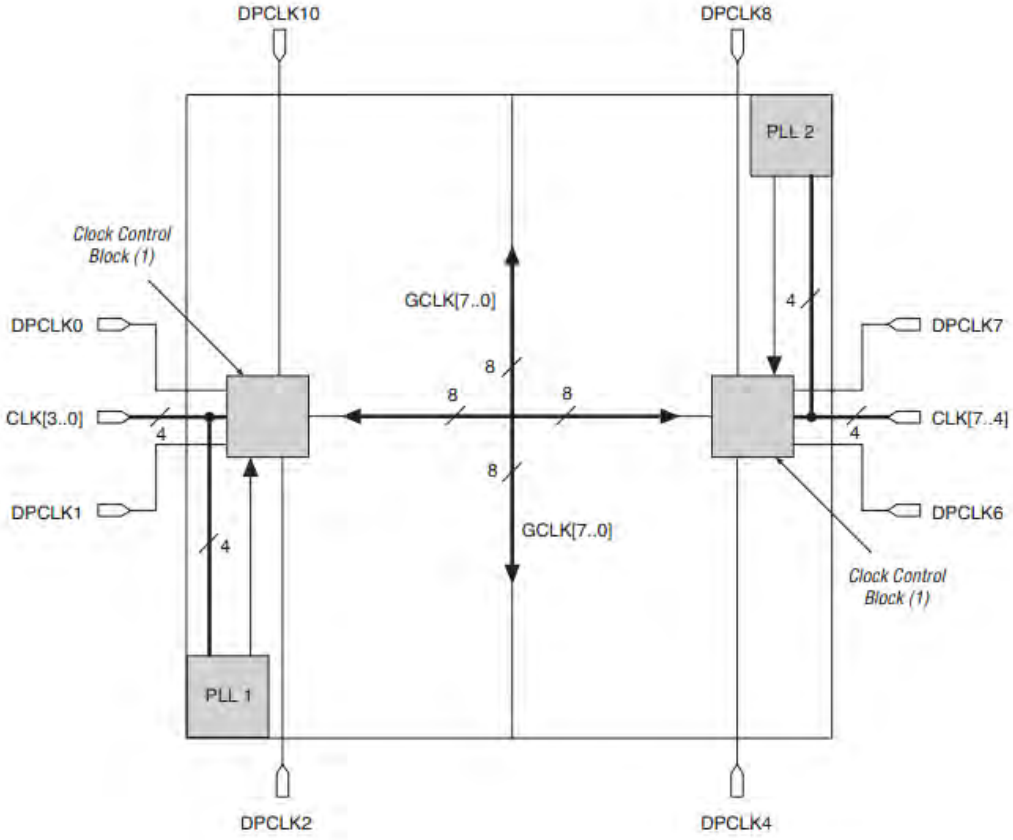
长线资源：用于 BANK 或者嵌入式功能单元的高速信号或者第二全局时钟的布线；

短线资源：用于其中逻辑单元间的逻辑互联与布线；

**全局布线资源：**



**全局时钟树：**

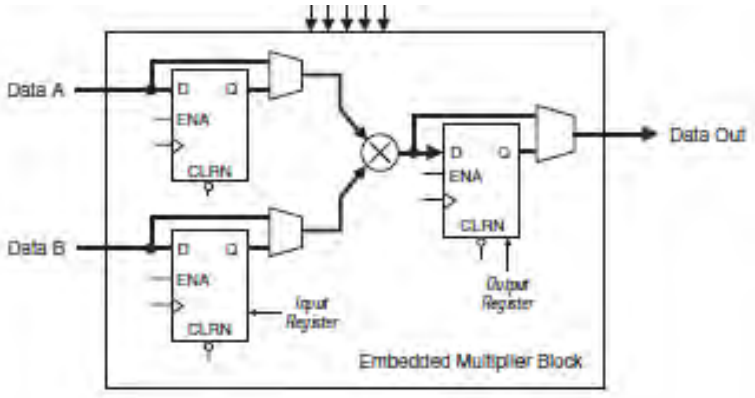



6、 底层嵌入式功能块

主要是指 PLL/DPLL、DCM、DSP48、乘法器、嵌入式硬核/软核；

Xilinx：DCM、DSP48/48E、DPLL、Multiplier 等

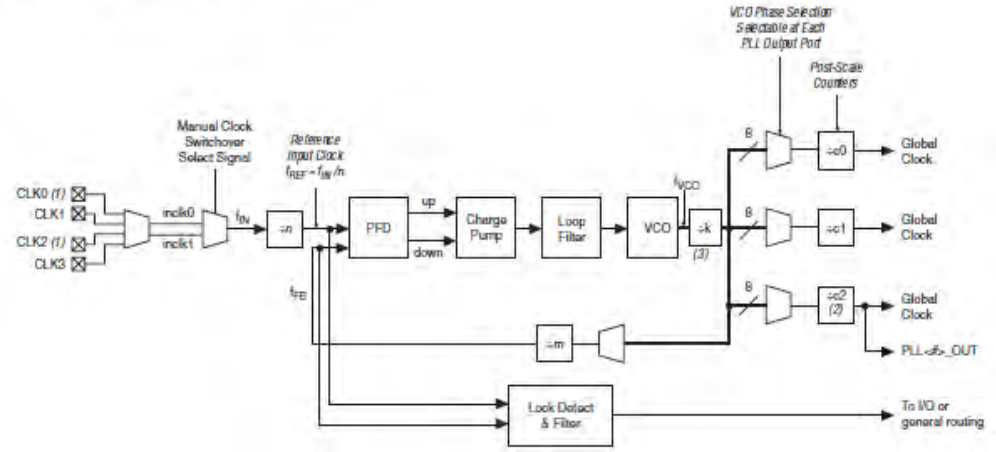
Altera：PLL/EPLL/FPLL、DSPcore 等；



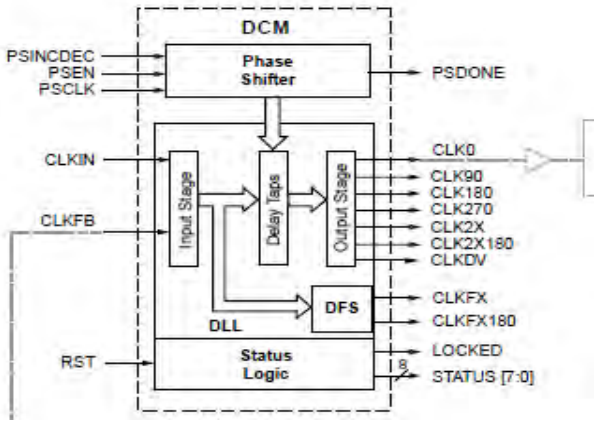

Multiplier 结构

**PLL/DCM：嵌入式锁相环**

Figure 2–16. Cyclone II PLL Note (1)




**Altera:PLL**




**Xilinx:DCM**

Altera 的 Cyclone II 器件最多有四个 PLL，分布在芯片四个角；需要主要的是 Altera 的 PLL 是模拟锁相环，在电源 / 地方面要做考虑。

Xilinx 的 spatan-3 器件最多有四个 DCM，也是分布在芯片四个角。

**两者的区别：**Altera 的 PLL 可支持较低的输入频率，可 Xilinx 的 DCM 支持的最低锁相频率为 24/32MHz；但 Xilinx 的高端器件 Virtex-5 的 DPLL 可达到很低的输入频率。

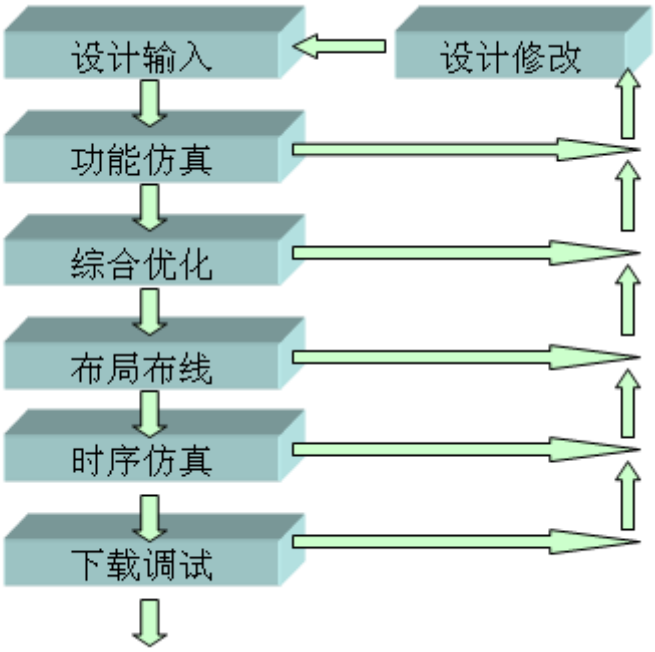
**内嵌专用硬核**

指高速串行收发器；GMAC、SERDES、PCIe 等；

Xilinx：GMAC、SERDES、PCI、GTX、GRX

Atera：GMAC、SERDES、PCIe、SPI.4/SFI.5

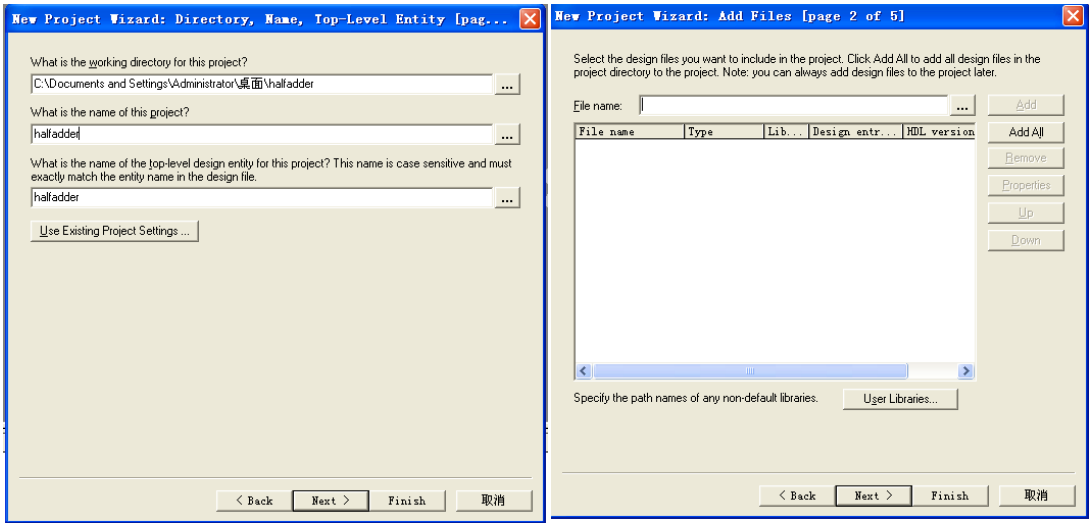
# 第六章 FPGA 的设计流程



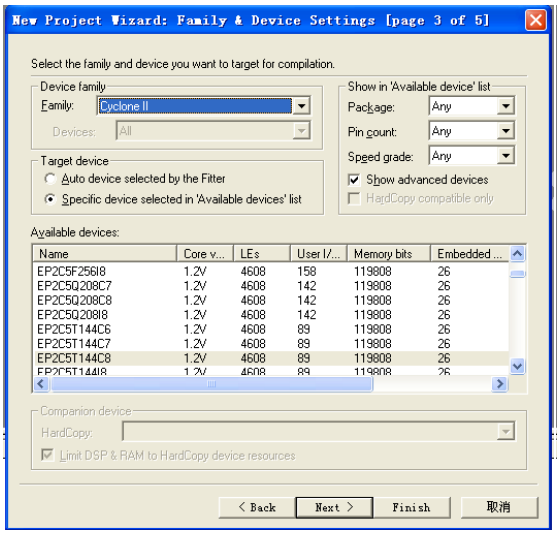

1. 设计的输入

**创建工程**

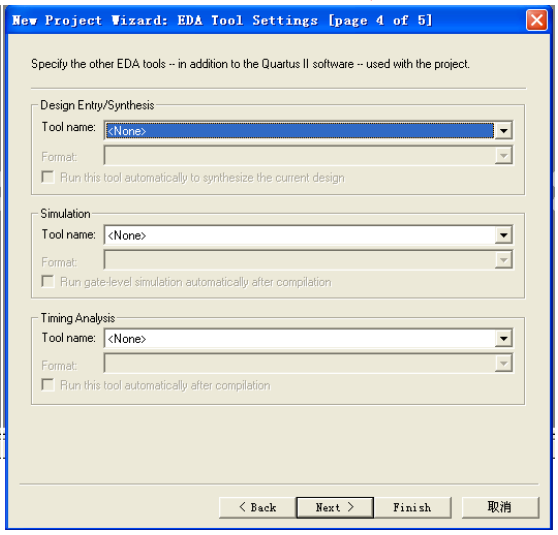
可通过创建工程向导创建一个的工程



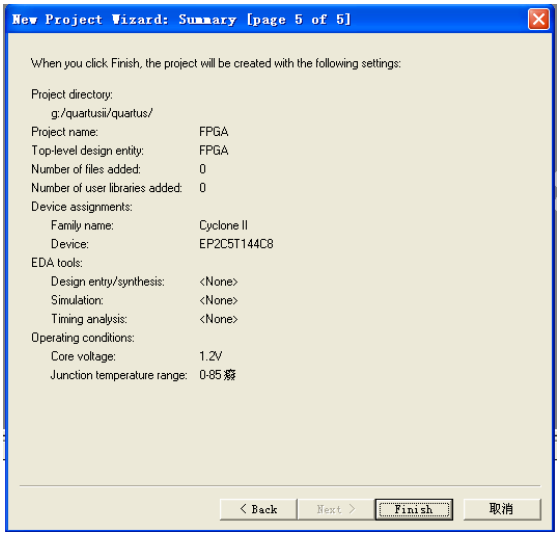

设置工程文件夹及工程名添加已存在文件（可选）




选择器件




选择设定第三方工具




显示设置信息

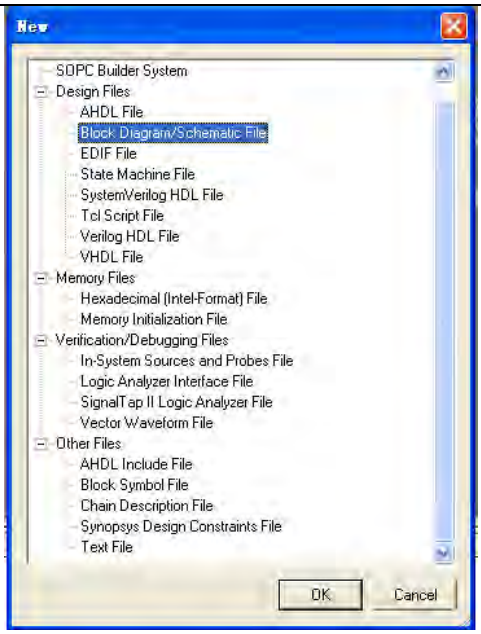
**图形输入**

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

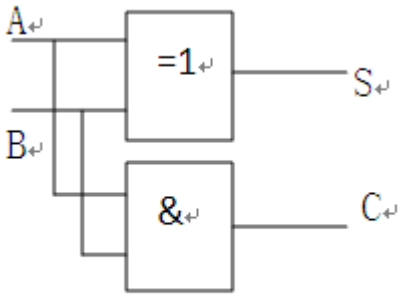
（这里以一个半加器为例子）



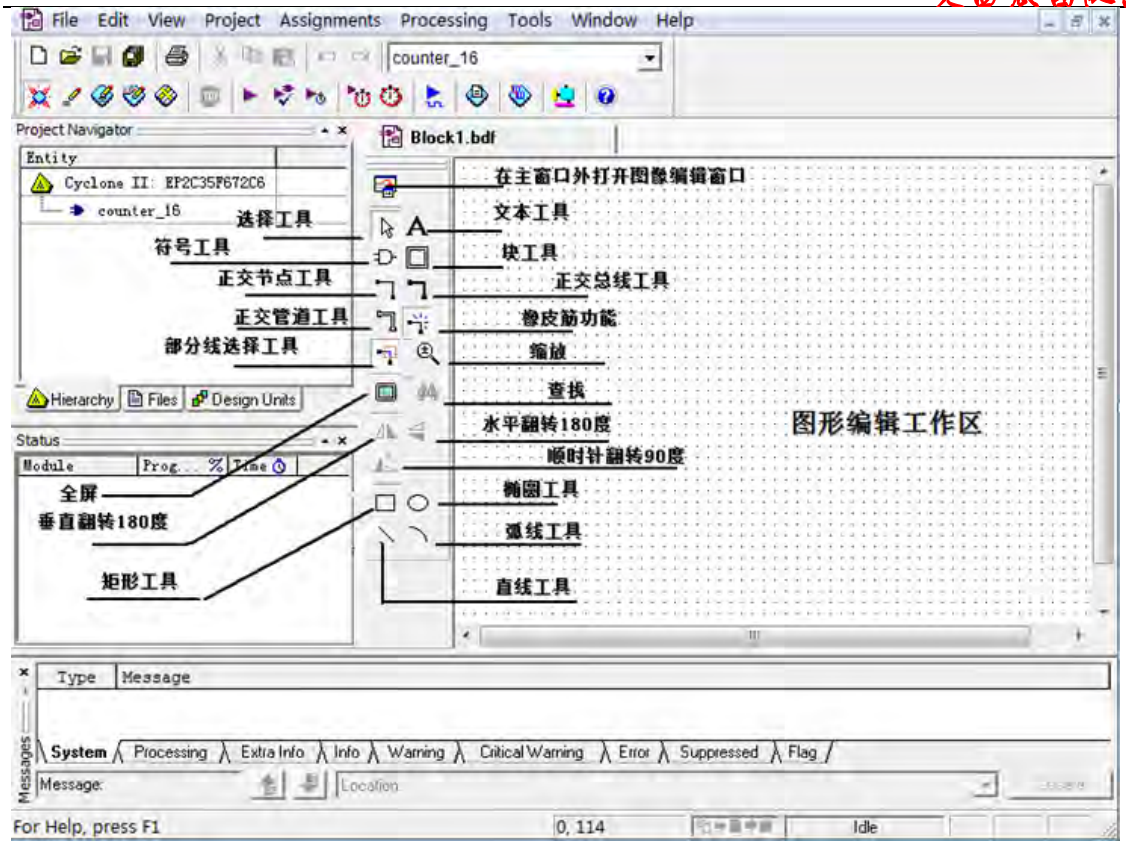

新建图形输入文件

| A | B | S | C |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

半加器真值表

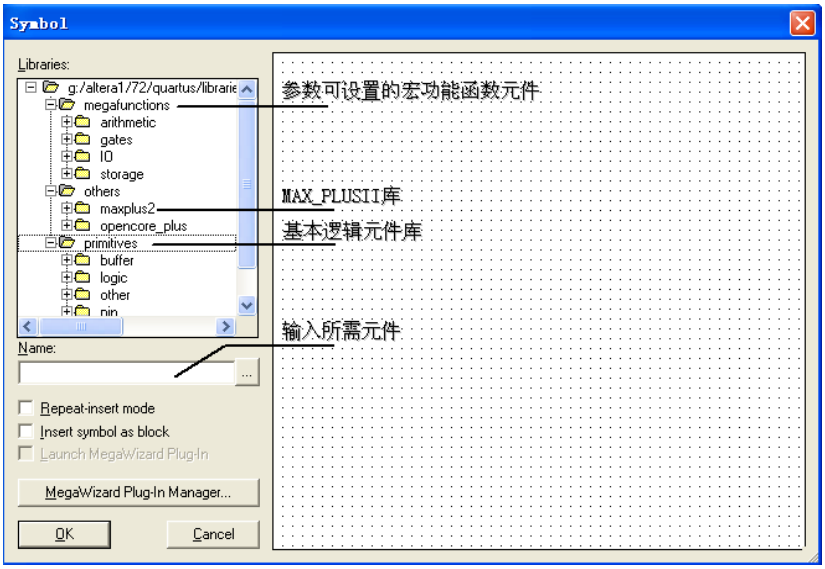



半加器的逻辑图

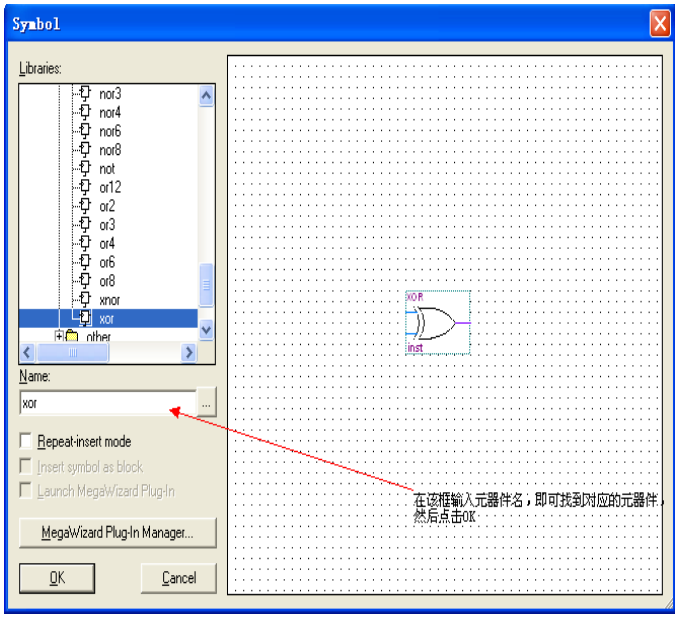



图形编辑器界面

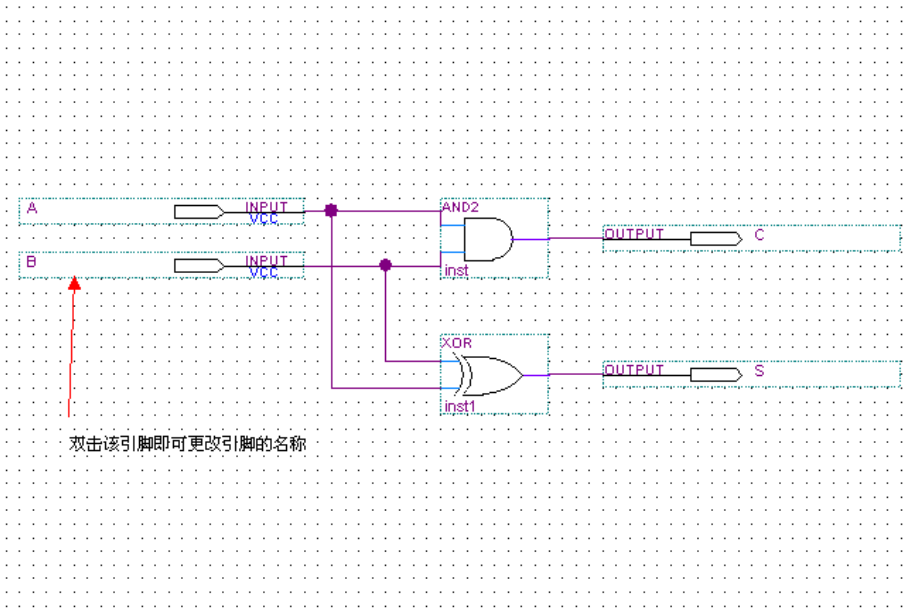
在图形编辑窗口中任一个位置双击鼠标，或点击图中的“符号工具”按钮，或选择 菜单 Edit 下的 Insert Symbol 命令，弹出下图所示的元件选择窗口：




Symbol 对话框




元器件的查找

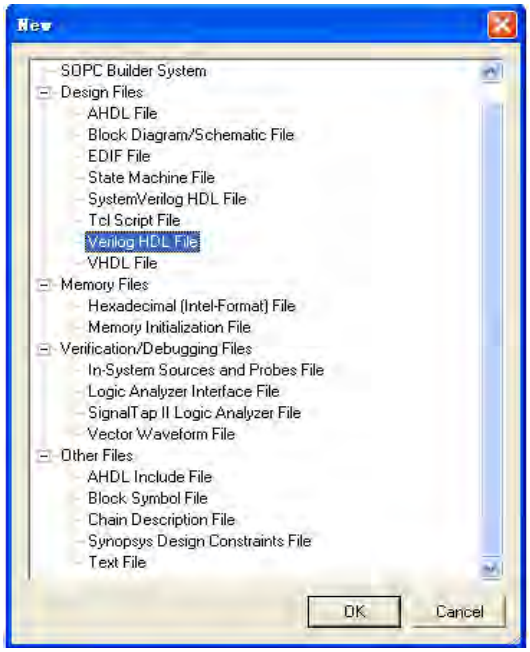



元器件的输入和引脚名的更改

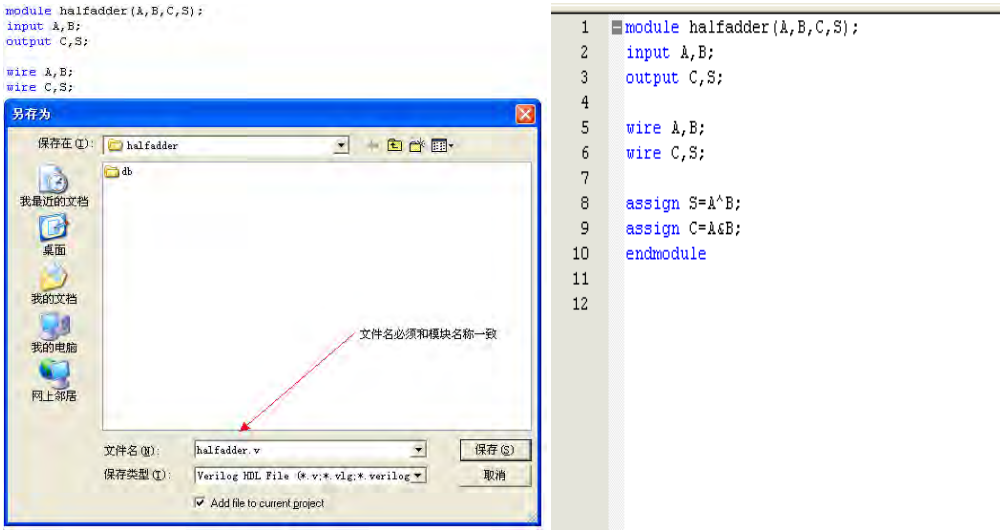
**文本输入**

Quartus II 自带文本编辑器，用于程序设计输入。

与图形输入类似，选择 Verilog HDL File 输入方式，即可打开文本编辑器；如下图所示，在编辑器中完成程序代码的编写。




新建文本输入文件

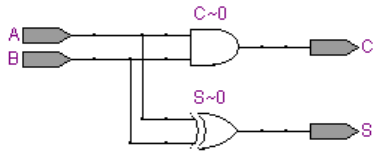



文件保存　　完成代码输入

NOTES

PFGA 开发流程中的设计输入、综合以及仿真步骤也可在第三方工具软件中完成。此时 Quartus II 软件将第三方工 具的输出作为输入文件，继续进行后续流程。

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

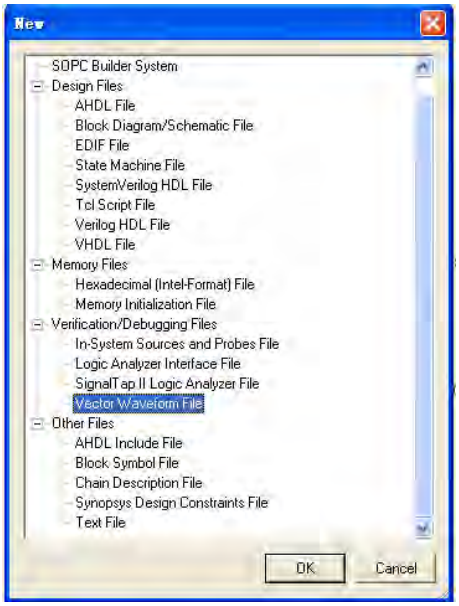


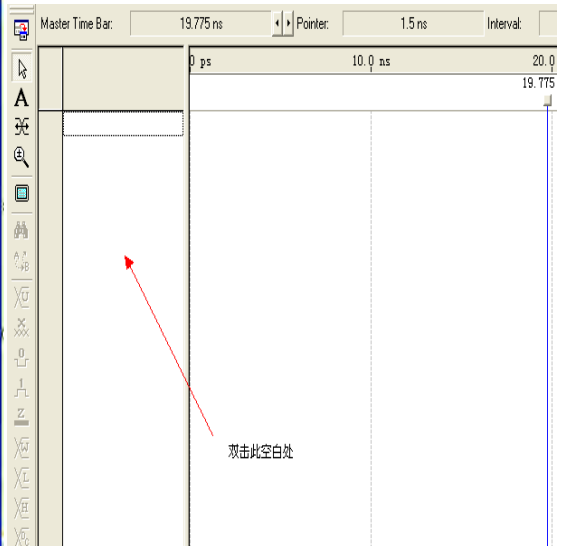

RTL 视图

2. 功能仿真

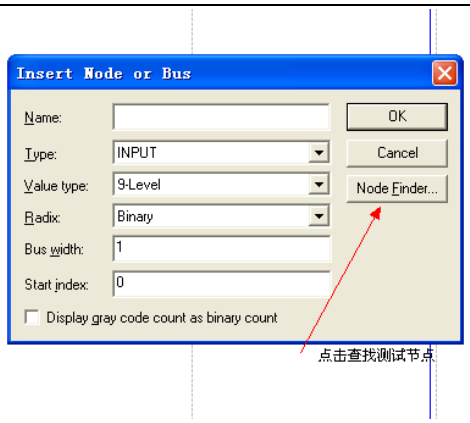
验证设计时序是否符合要求；工具有 modelsim、Activehdl、NC-Verilog/Vhdl、各厂家自带工具等

1. 选择【File】--【New】选项，打开新建文件类型选择窗口
2. 选择 Vector Waveform File 打开波形仿真窗口
3. 添加仿真测试信号，设置输入信号和仿真参数设置

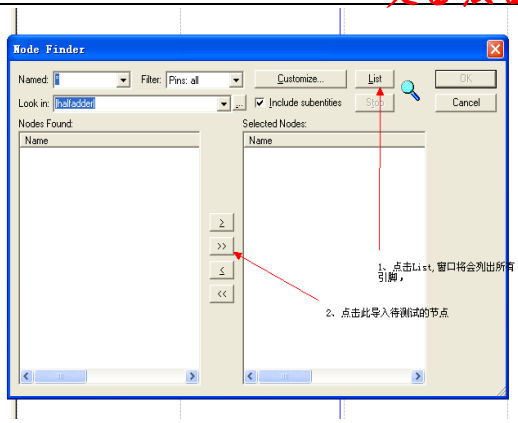


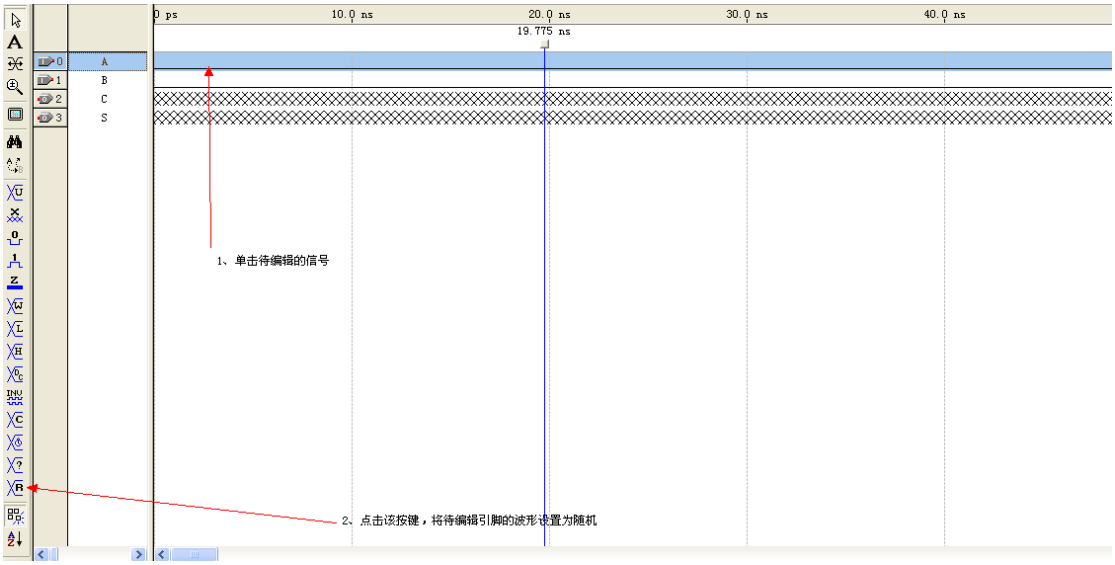

新建波形仿真文件　　添加仿真测试信号



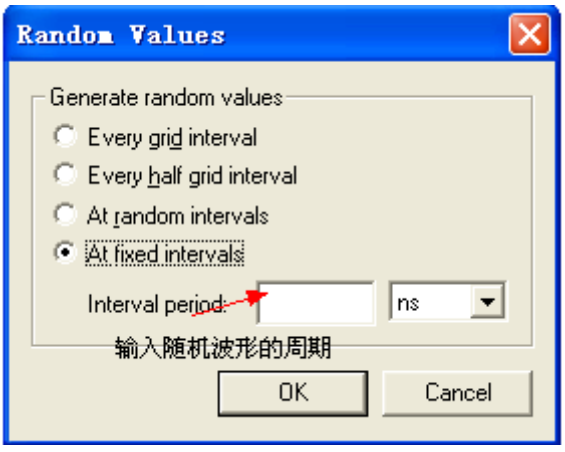

查找节点




添加节点

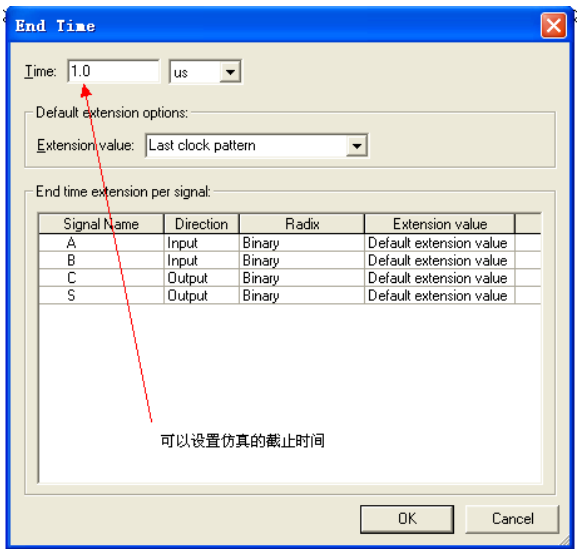



设置待编辑的引脚 1



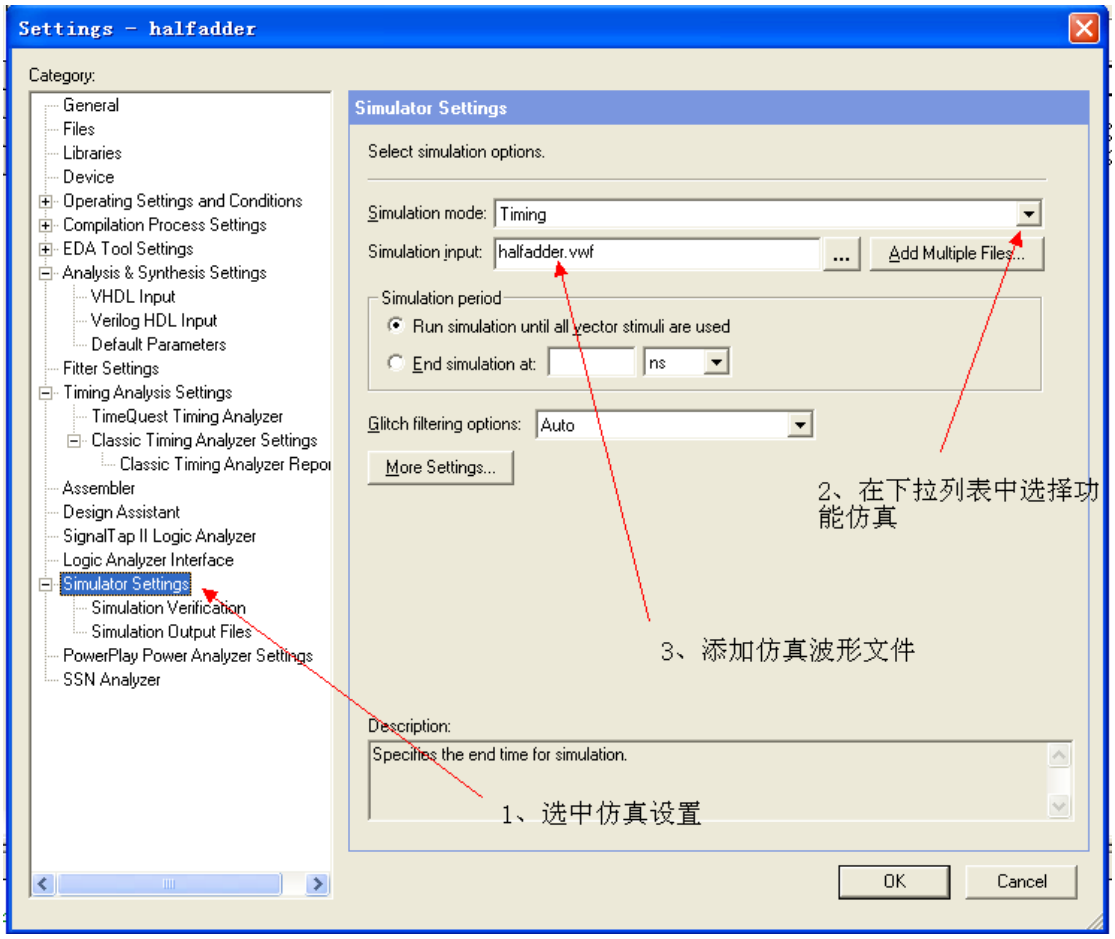

设置待编辑的引脚 2

设置仿真参数：（1）选择【Edit】--【End Time】选项，出现下面的窗口




设置仿真截止时间

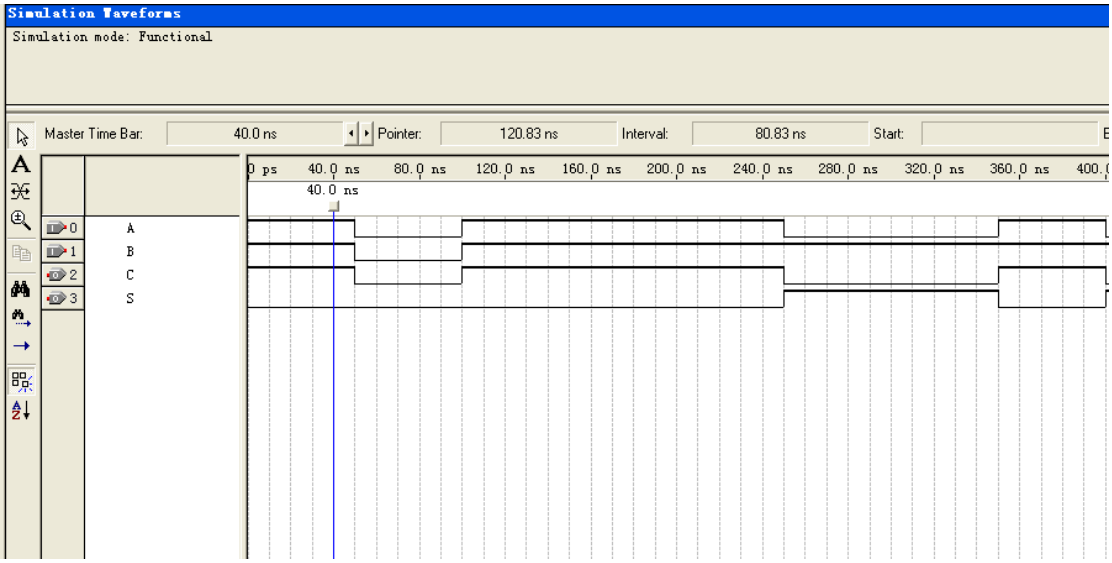
（2）选择【Assignments】--【Setings】选项，出现下面的窗口




(3)点击  进行编译，编译完成后在生成功能仿真网表：【Processing】--【Generate

Functional Simulation Netlist】选项

(4) 点击  进行仿真




3. 综合优化

综合（Synthesis）：综合过程完成对设计文件进行分析，生成门级网表文件。

在 Quartus II 中完成设计的输入后选择【Processing】-【Start】-【Start Analysis & Synthesis】可以启动综合过程或者点击  。

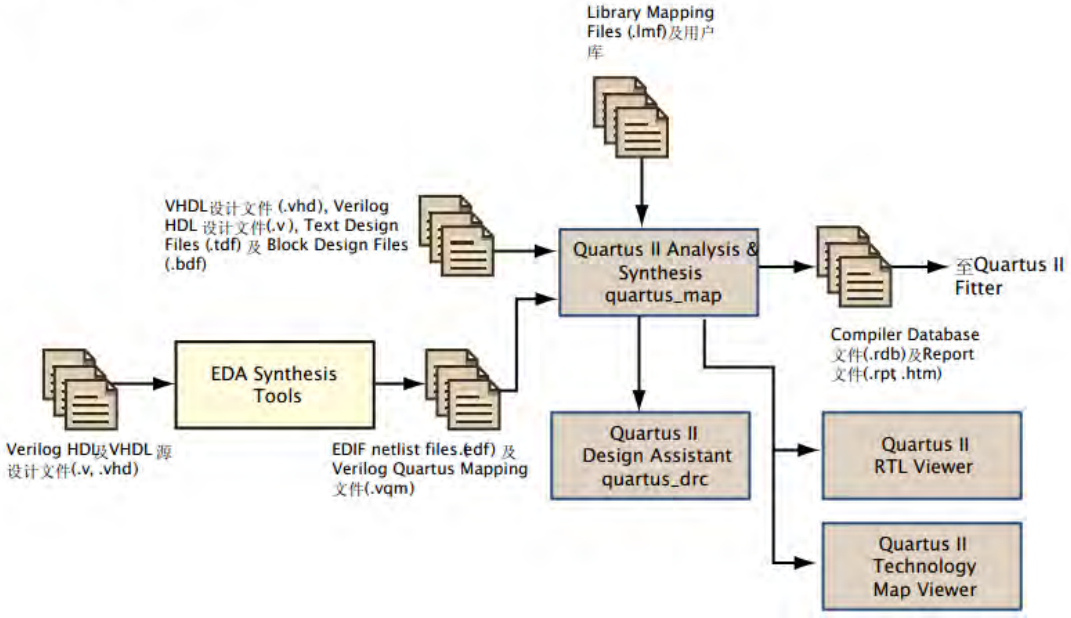
综合过程的输入可以是设计源文件(.v/.vhd/.tdf/.bdf etc.)，也可以是第三方综合工具的输出文件，如 Synplify 综合工具输出的综合结果.edf/.vqm 文件。

综合后形成工程数据库文件，用于后续的布局布线流程。

门级是 RTL 级的更进一步，设计此时已转变以基本逻辑门单位的逻辑网表。

综合设计流程图如下：

图 1. 综合设计流程

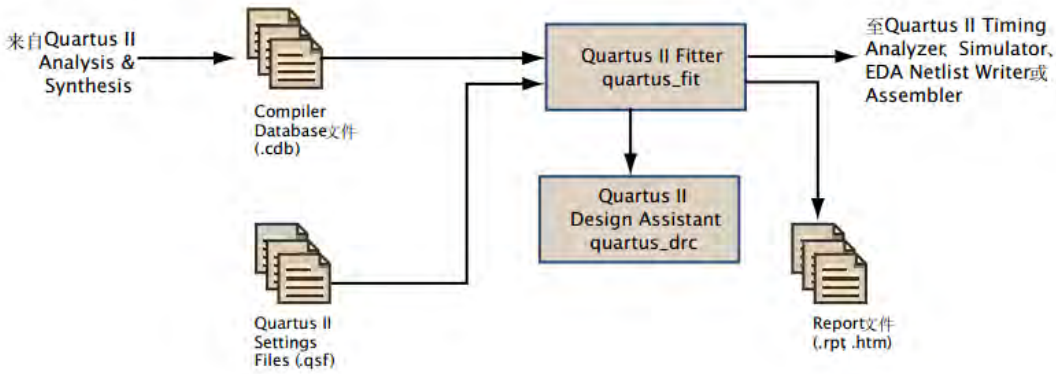



4. 适配

适配过程执行布局布线功能 。适配使用由 Analysis & Synthesis 建立的数据库，将工程的逻辑和时序要求与器件的可用资源相匹配。它将每个逻辑功能分配给最佳逻辑单元位置，进行布线和时序分析，并选定相应的互连路径和引脚分配。

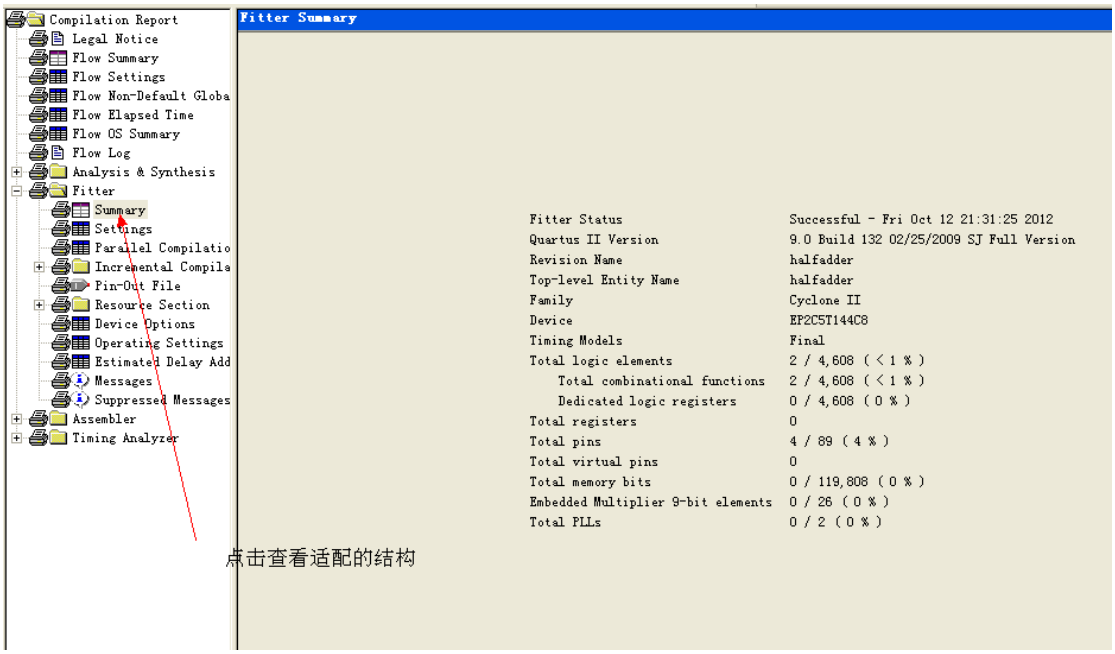
适配完成后，通过 Netlist Writer 生成的标准网表文件(.vo) 以及标准延时(.sdo)文件，用于对设计进行时序仿真（后仿真）。

图 1. 布局布线设计流程



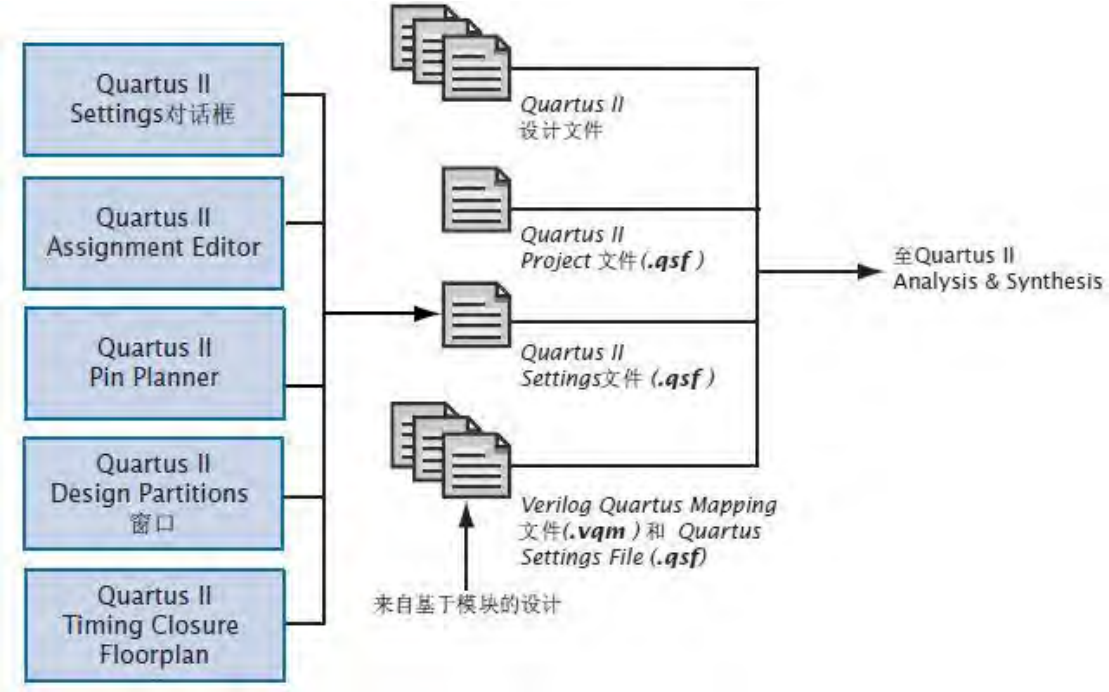

如果在设计中进行了资源分配， Fitter 试图将这些资源分配与器件上的资源相匹配，努力满足您已设置的任何其它约束条件，然后试图优化设计中的其余逻辑。如果尚未对设计设置任何约束条件，Fitter 将自动优化设计。如果适配不成功， Fitter 会终止编译，并给出错误信息。

**使用 Report 窗口或 Report 文件查看适配结果**




5. 约束

建立工程和设计之后，可以使用 Quartus II 软件 Assignment 菜单中的 Settings 对话框、Assignment Editor 、Pin Planner、Design Partitions 窗口和 Timing Closure 布局图指定初始设计约束条件，例如，引脚分配、器件选项、逻辑选项和时序约束条件。约束将对后序的综合与适配过程产生控制与影响




使用【Assignments】菜单下的 Settings 对话框，可以设置一般工程范围的选项以及综合、适配、仿真和时序分析选项。如：

（1）修改工程设置：为工程和修订信息指定和查看当前顶层实体；从工程中添加和删除文件；指定自定义的用户库；为封装、引脚数量和速度等级指定器件选项；指定移植器件。

指定 EDA 工具设置：为设计输入、综合、仿真、时序分析、物理综合以及相关工具选项指定 EDA 工具。

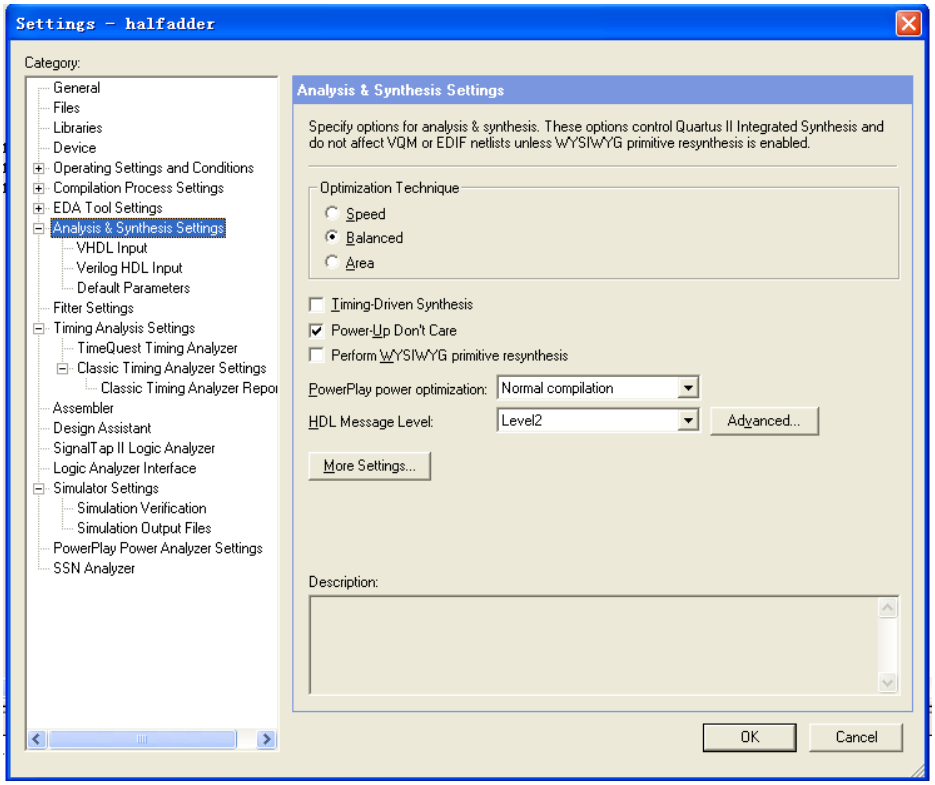
（2）指定编译过程选项：智能编译选项，在编译过程中保留节点名称，运行 Assembler，以及渐进式编译或综合，并且保存节点级的网表，导出版本兼容数据库，显示实体名称，使能或者禁止 OpenCore® Plus 评估功能，还为生成早期时序估算提供选项。

（3）指定时序分析设置：为工程设置默认频率，定义各时钟的设置，延时要求和路径排除选项以及时序分析报告选项。

（4）指定 PowerPlay Power Analyzer 设置：输入文件类型、输出文件类型和默认触发速率，以及结温、散热方案要求、器件特性等工作条件。

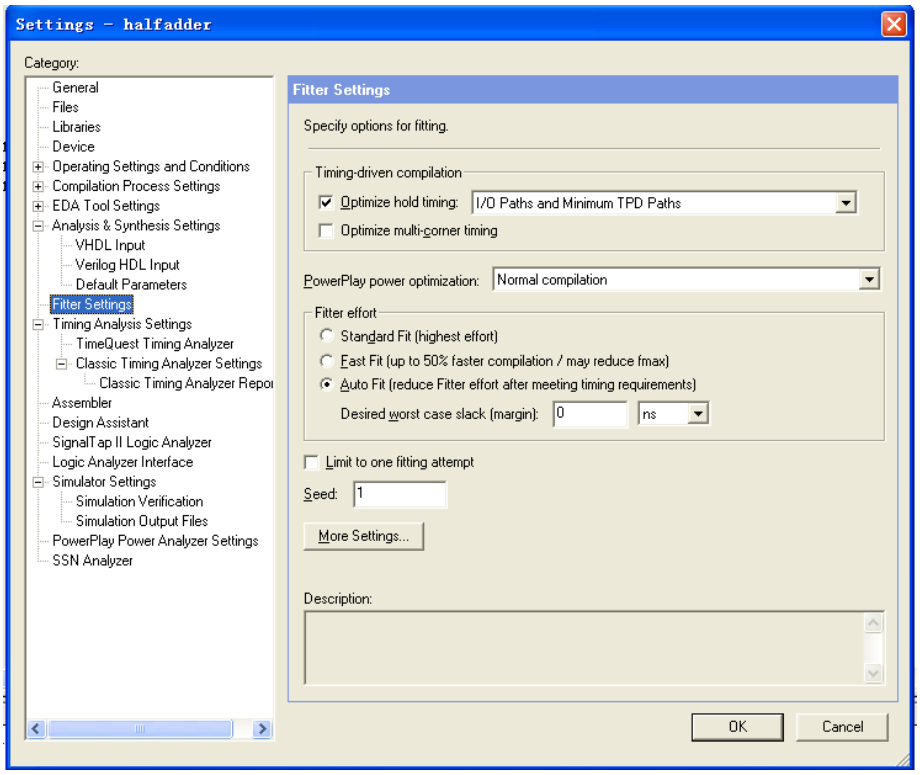
**Settings 对话框 - Analysis & Synthesis 设置**

选择菜单【Assignments】-【Settings】对话框 Category 中的 Analysis & Synthesis settings 选项，则可进行用于 Analysis&Synthesis、HDL 输入、默认设参数和综合网表优化选项工程范围内的设置



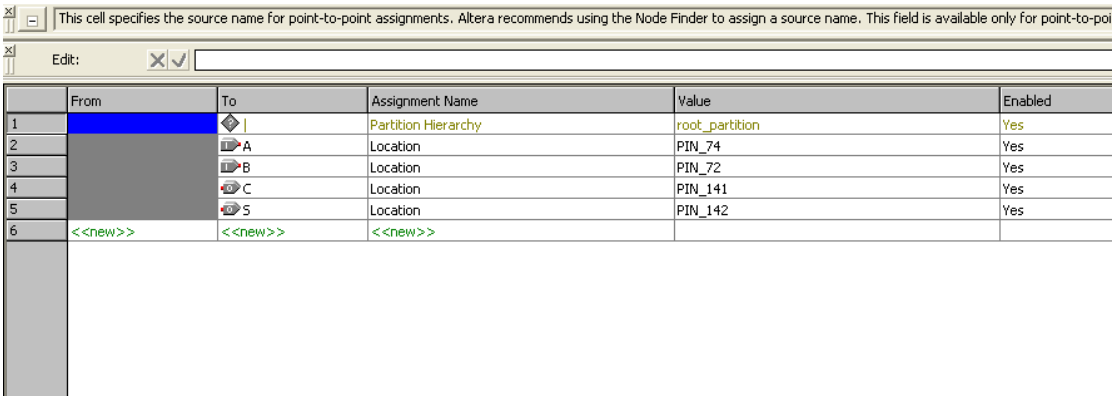

**Settings 对话框 – Fitter 设置**

选择菜单【Assignments】-【Settings】对话框 Category 中的 Fitter Settings 选项，则可进行时序驱动编译选项、Fitter 等级、工程范围的 Fitter 逻辑选项分配，以及物理综合网表优化等设置。




**分配编辑器**

Assignment Editor  界面用于在 Quartus II 软件中建立、编辑节点和实体级别分配。分配用于在设计中为逻辑指定各种选项和设置，包括位置、I/O 标准、时序、逻辑选项、参数、仿真和引脚分配。




**引脚分配**

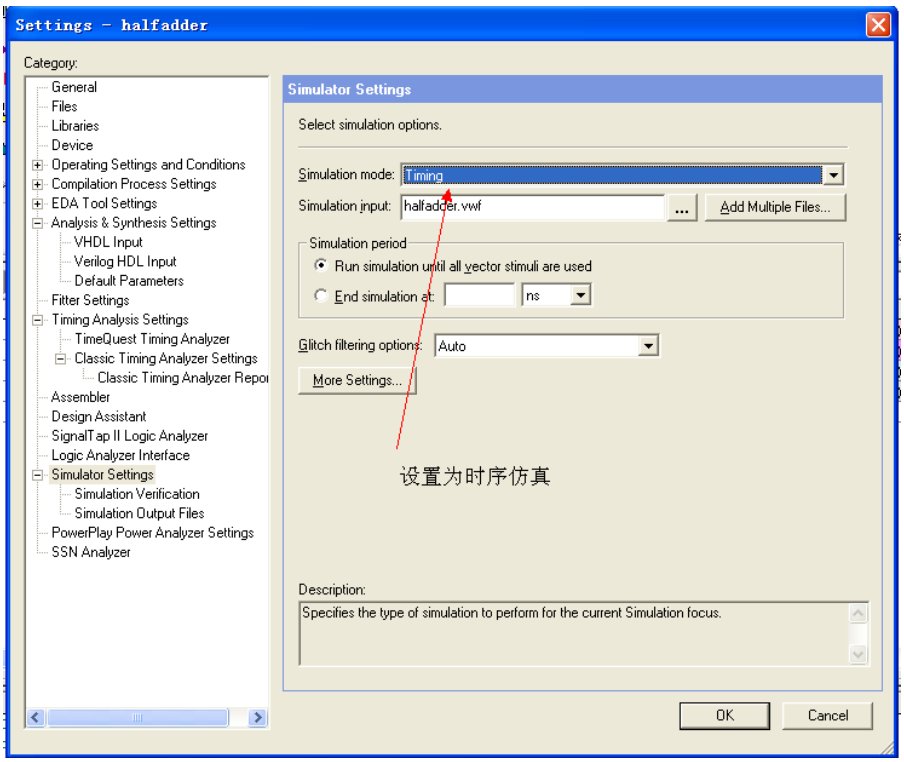
引脚分配可以在 Assign Editer 中进行，也可在【Assignments】-【Pin Planner】中实现，如图：

Named: Edit:

| | Node Name | Direction | Location | I/O Bank | VREF Group | I/O Standard |
|---|---|---|---|---|---|---|
| 1 | A | Input | PIN_74 | 3 | B3_N1 | 3.3-V LVTTL (default) |
| 2 | B | Input | PIN_72 | 4 | B4_N0 | 3.3-V LVTTL (default) |
| 3 | C | Output | PIN_141 | 2 | B2_N1 | 3.3-V LVTTL (default) |
| 4 | S | Output | PIN_142 | 2 | B2_N1 | 3.3-V LVTTL (default) |
| 5 | <<new node>> | | | | | |

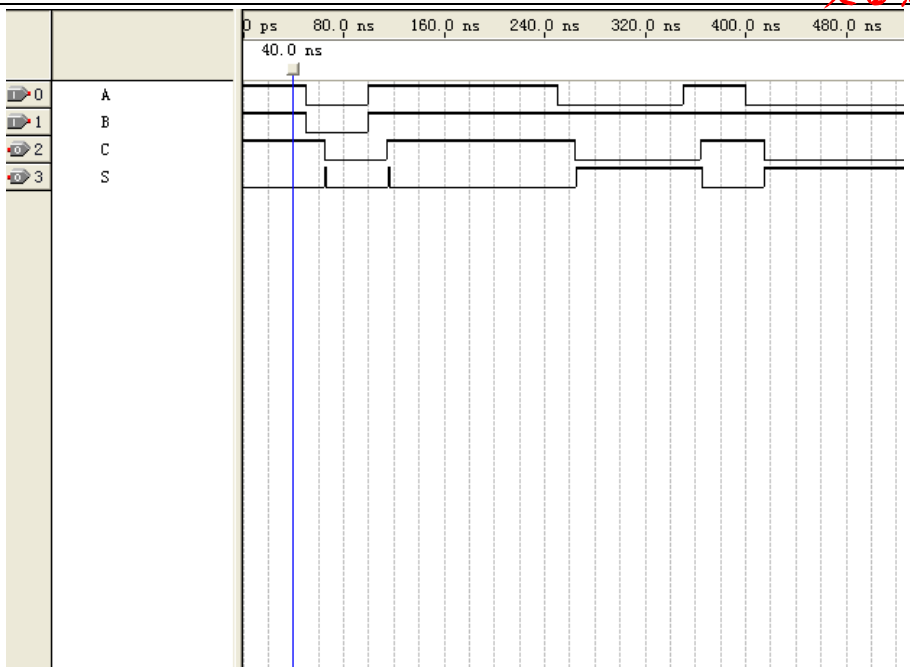
6. 时序仿真

在功能仿真的基础上，将功能仿真设置为时序仿真




点击  进行编译，编译完成后在生成功能仿真网表：【Processing】--【Generate Functional Simulation Netlist】选项

点击  进行仿真



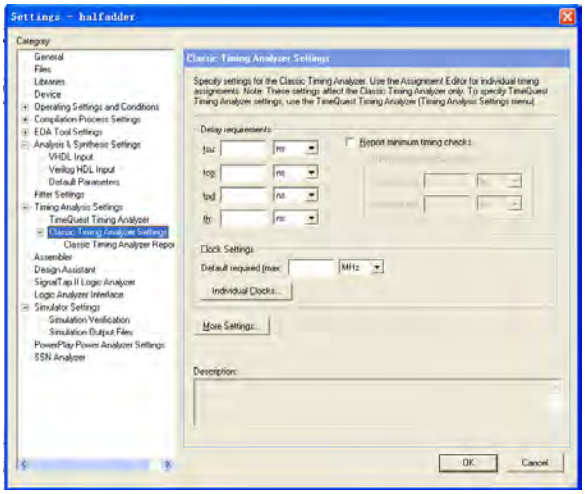

从时序仿真的结果可以看出输入与输出的信号都存在一个延迟。

7. 时序分析

Quartus®IITiming Analyzer 可用于分析设计中的所有逻辑，并有助于指导 Fitter 达到设计中的时序要求。默认情况下，Timing Analyzer 作为完整编译的一部分自动运行，分析、报告时序信息，例如，建立时间(tSU)、保持时间( tH)、时钟至输出延时和最小时钟至输出延时(tCO)、引脚至引脚延时和最小引脚至引脚延时(tPD)、最大时钟频率( fMAX)，以及设计的其它时序特性。当提供时序约束或者默认设置有效时，Timing Analyzer 报告迟滞时间。可以使用 Timing Analyzer 生成的信息分析、调试和验证设计的时序性能。还可以使用快速时序模型，验证最佳情况(最快速率等级的最小延时)条件下的时序。

选择【Assignments】-【Timing Analysis Settings】命令，弹出 Settings 对话框的 Timing Requirements & Opinions 页，在对话框中，可以对逻辑的静态时序要求做出设定。

选择菜单【Processing】-【Start】-【Start Timing Analyzer】可以单独启动静态时序分析进程,这一步骤默认包括在全编译进程中。在编译结束后,通过编译报告(Compilation Report)中的 Timing Analyzer 项可以查看设计中的时序是否满足要求




8. 下载调试

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 方式下载**

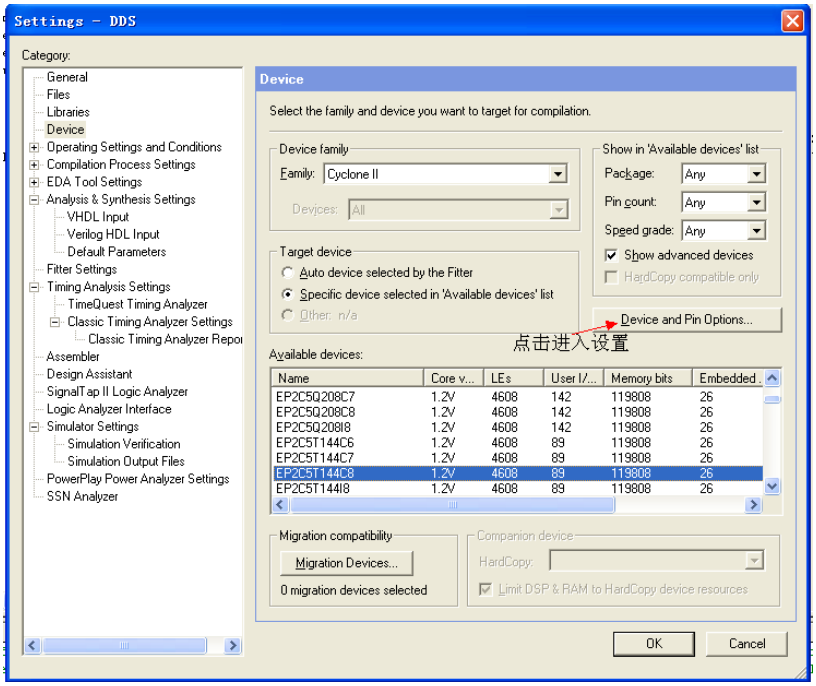
- 选择 Quartus II 主窗口的 Tools 菜单下的 Programmer 命令或点击  图标，进入器件编程和配置对话框。如果此对话框中的 Hardware Setup 后为“No Hardware”，则需要选择编程的硬件。点击 Hardware Setup，进入 Hardware Setup 对话框，下图所示，在此添加硬件设备。
- 配置编程硬件后，选择下载模式，在 Mode 中指定的编程模式为 JTAG 模式；
- 确定编程模式后，单击 

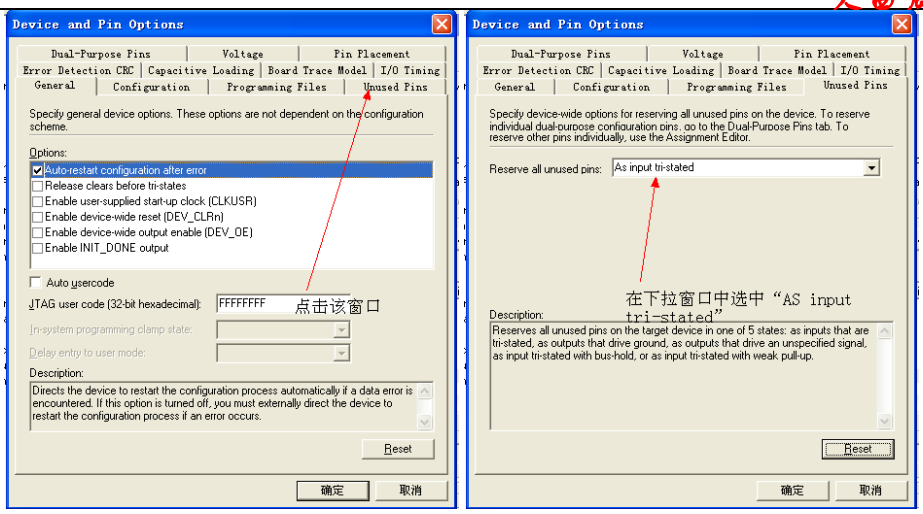
添加相应的 counter.sof 编程文件，选中 counter.sof 文件后的 Program/Configure 选项，然后点击 

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。
- 界面下图所示

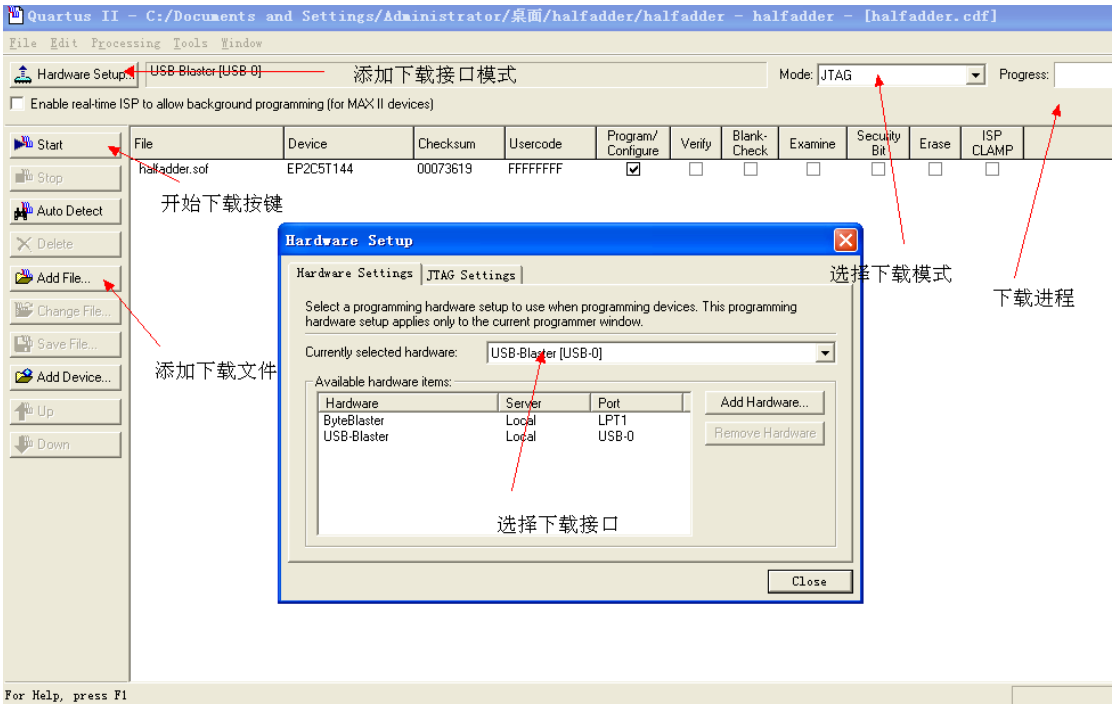
下载前先进行管脚设置，将未用到的管脚设置为输入三态；

【Assignments】--【Device】然后出现以下窗口






下载界面的设置

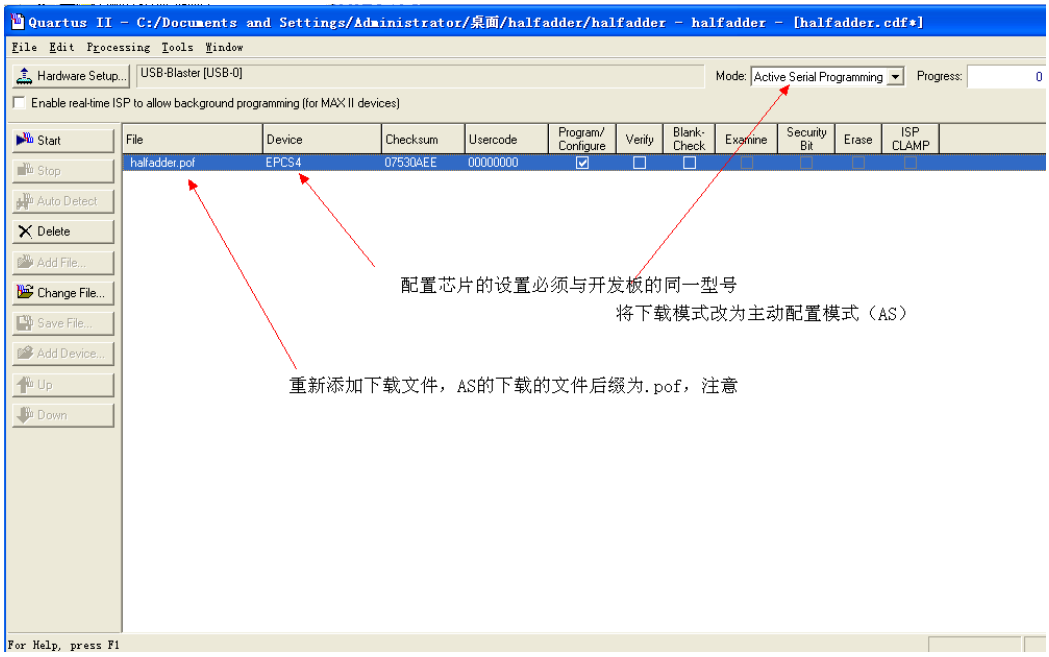



**2、AS 方式下载**

- 选择 Quartus II 主窗口 Assignments 菜 Device 命令，进入 Settings 对话框的 Device 页面进行设置，如下图所示。
- 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；
- 单击 

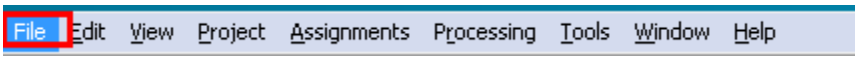
添加相应的 counter.pof 编程文件，选中文件后的 Program/Configure、Verify 和 Blank Check 项，单击图标 

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 EPCS 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。



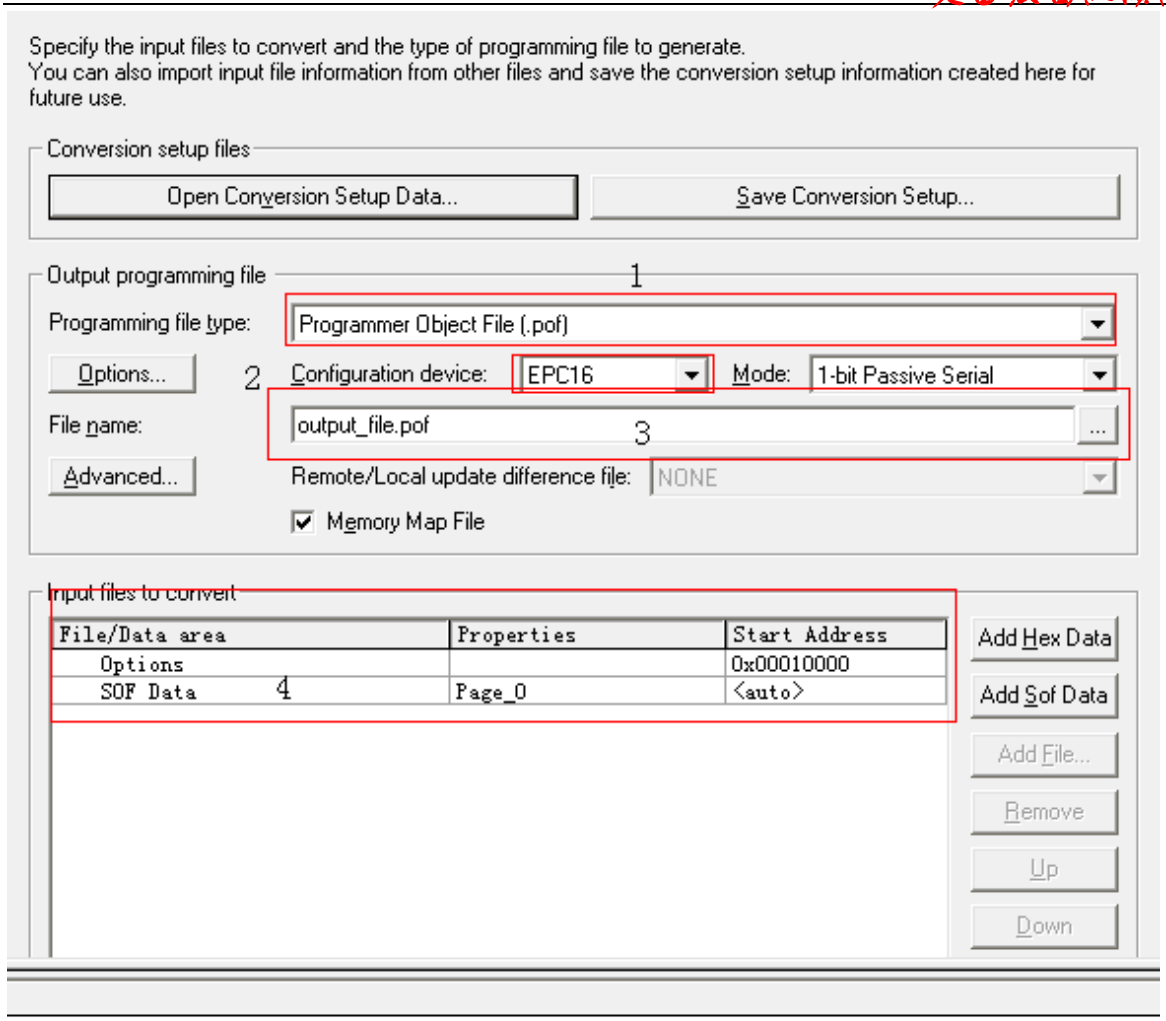

注意：JTAG 和 AS 接口不能带电拔插，不然很容易烧坏配置芯片!!!

### 3、JTAG 配置 EPCS




File→convert programming Files

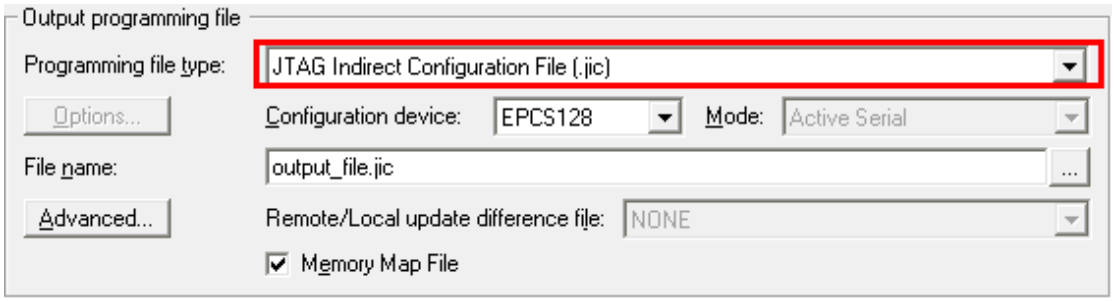
需要设置的地方，有如下几个：




**第一步：**

设置下载文件的类型

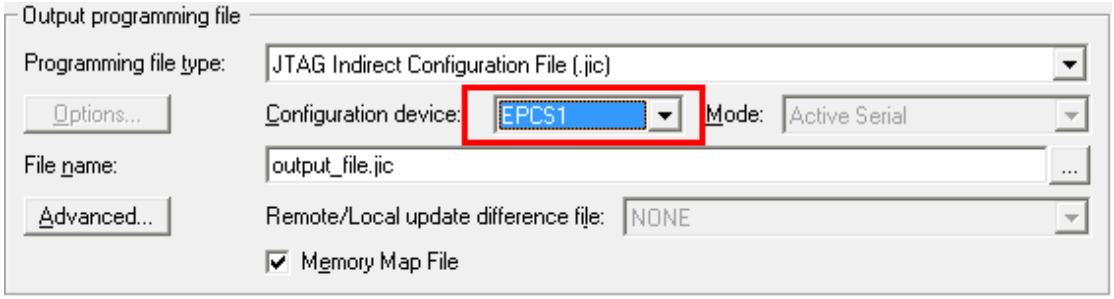
这里要的是后缀为.jic 的文件




**第二步：**

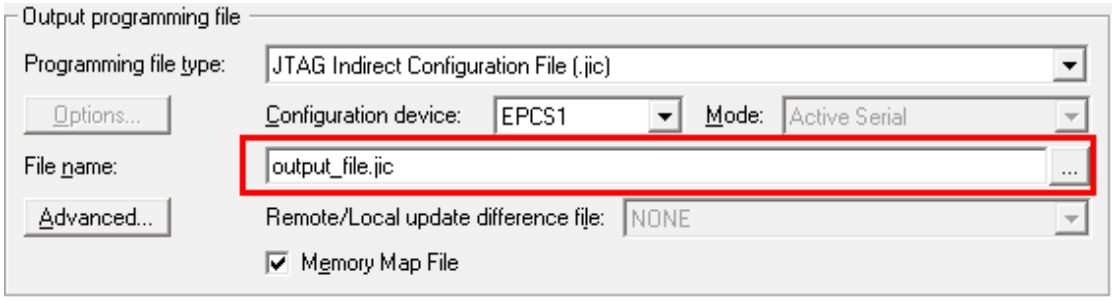
设置配置芯片的类型：

这里选择为 EPCS1 和开发板的配置芯片相匹配



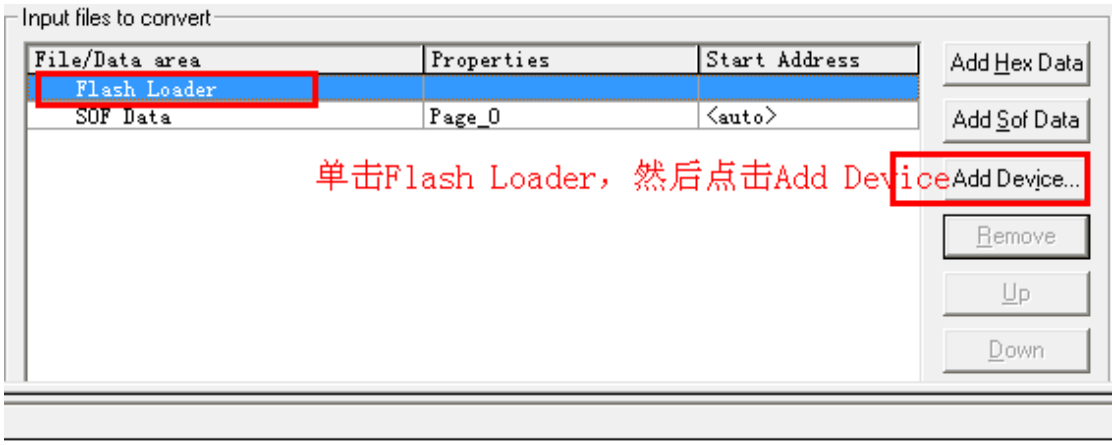

**第三步：**

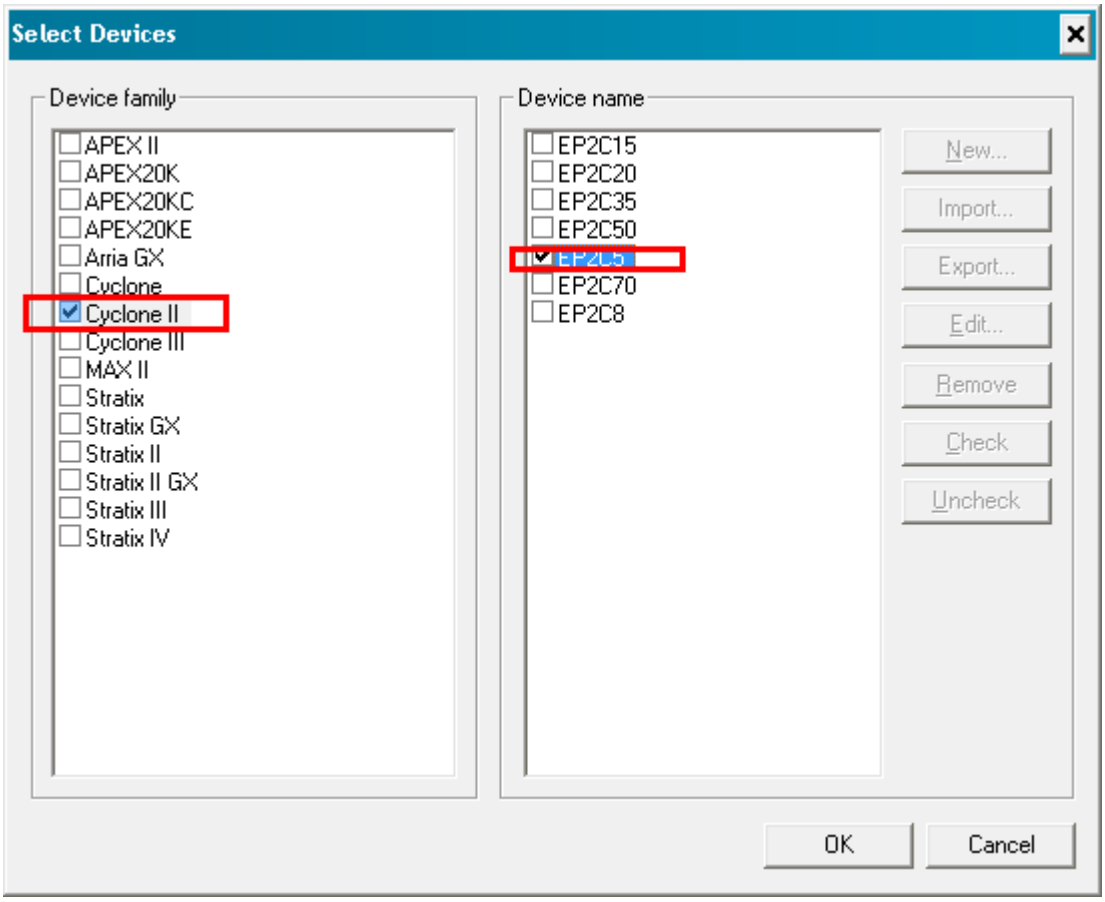
设置输出 jic 文件的名称，这里默认；



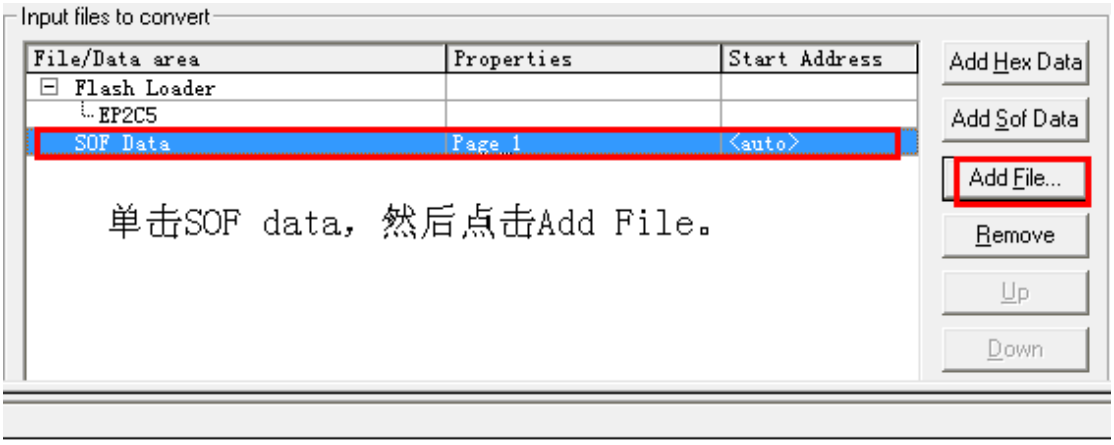

**第四步：**

输入 sof 文件和设置 FPGA 器件类型：



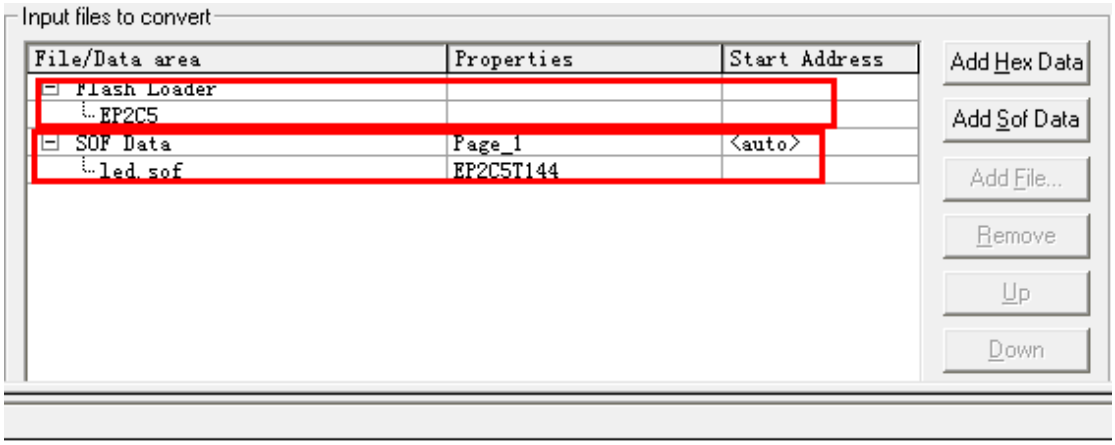



选择和开发板的 FPGA 芯片类型相匹配的芯片。

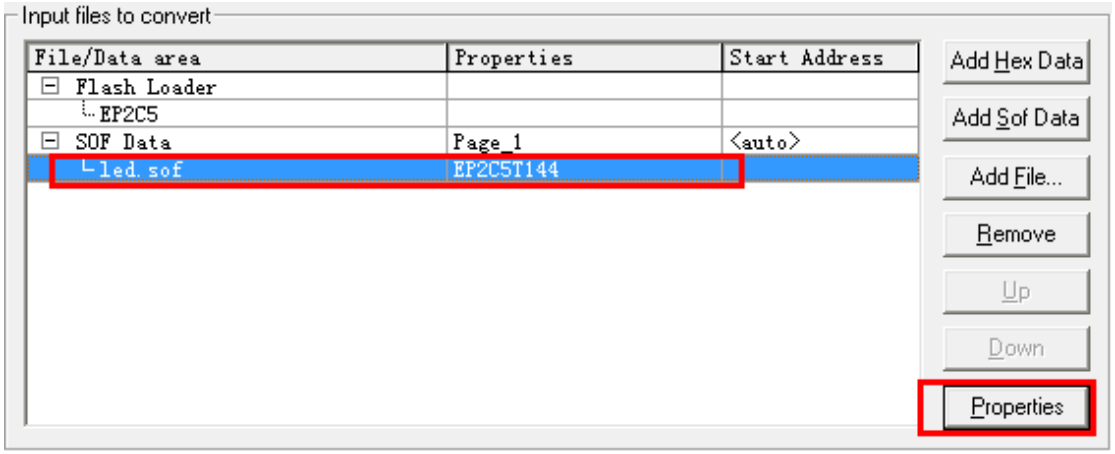


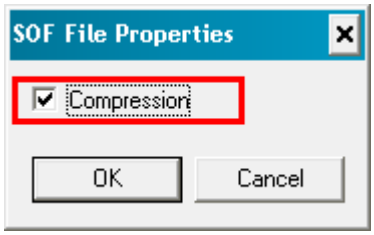

选择.sof 文件

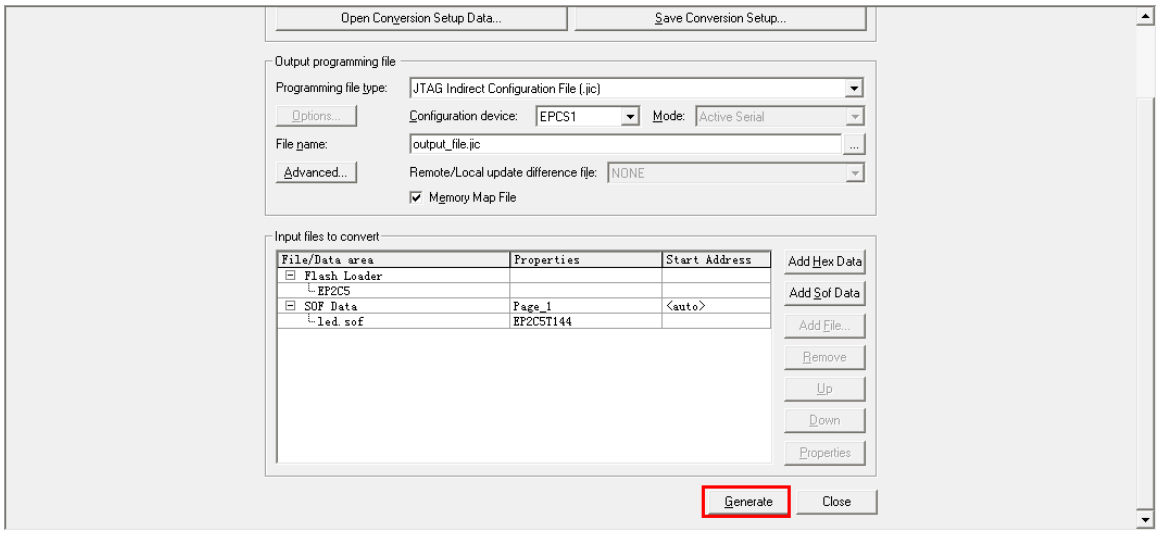
最后窗口如下：




**第五步：**



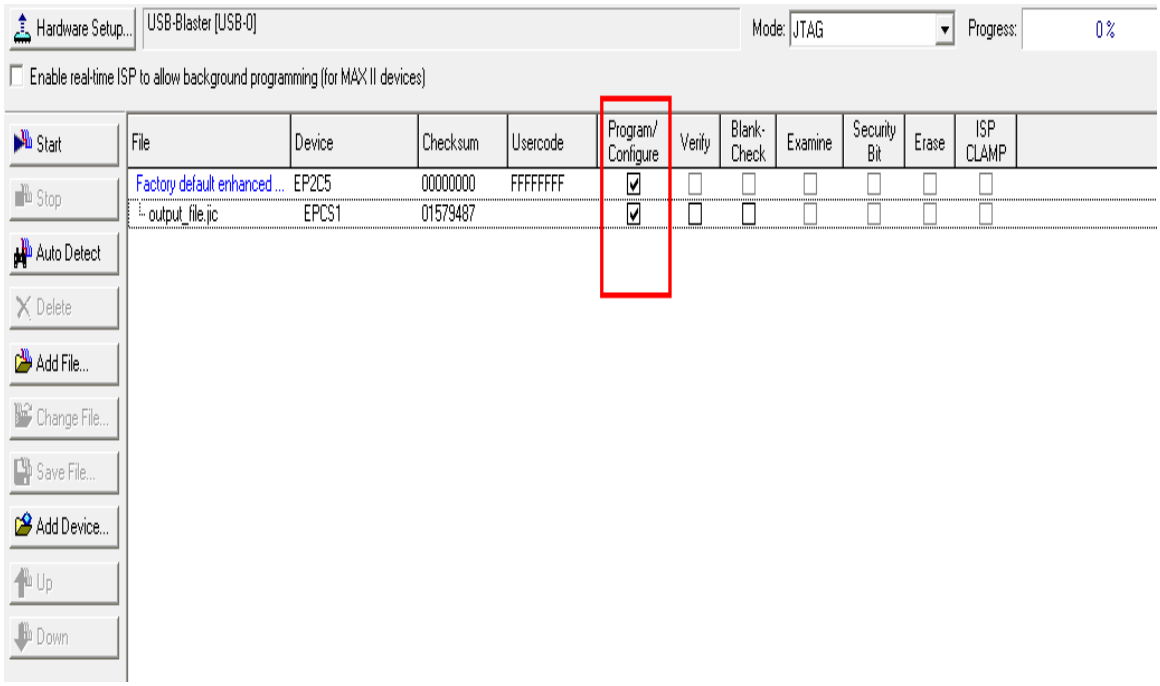



点击生成 jic 文件

**最后一步**

进行下载



# 第七章 Verilog 代码规范 I

## 7.1、“规范”这问题

“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。

网上关于 EDA 设计方面的代码规范问题的资料，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。

## 7.2、Verilog 代码有哪些规范？

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，如下：

- **Ports**

```
input       rst_n;
output      wr_data;
```

- **Signals**

```
reg [7:0]   a_reg;
wire        we_n;
```

- **Constructs**

```
task        busy;
function    call_addr;
```

- **Instance names**

```
block1  u_block1 (…
```

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 的对准问题
   这里采用一个实例进行说明：

```
module SYN3_1 (a, data_out);
input a;
output data_out;
always @ (a)
  begin
    if (a == 1'b1)
      data_out = 1'b0;
  end
endmodule
```

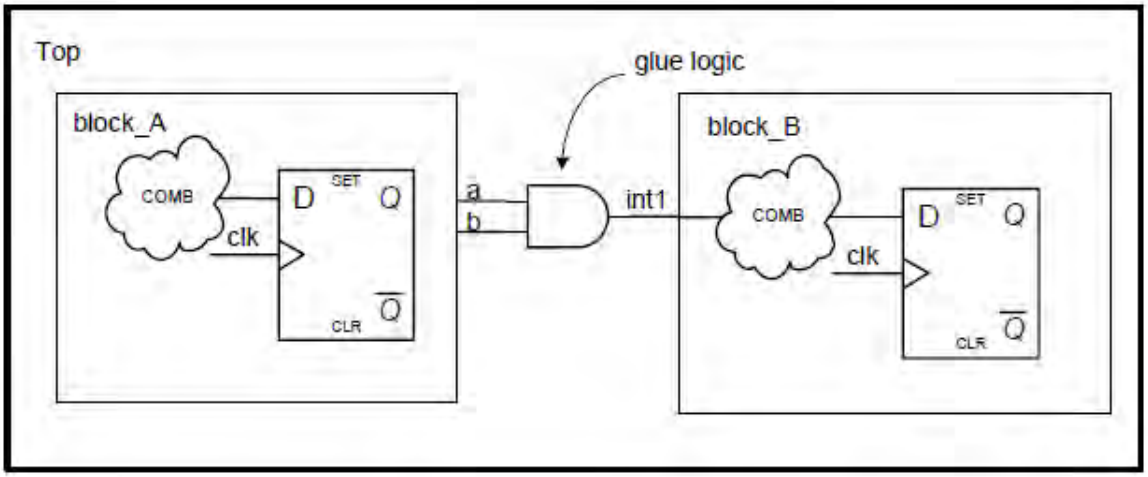
9. 到这里所说到的都是关于文本输入形式的，那么在原理图输入方式中有没有相关的规范

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，这是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有缝地连接起来，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？。。。。。。。。。。。。。

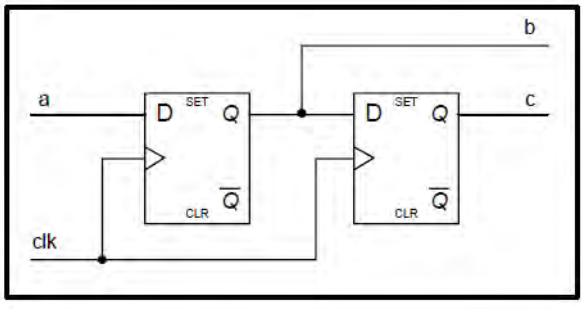


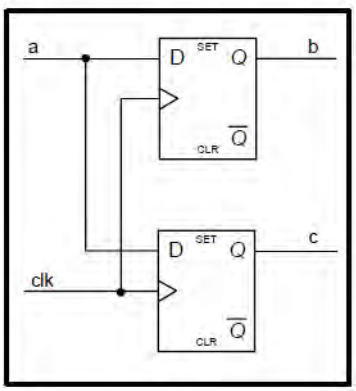

10. 非阻塞和阻塞问题，这个问题在 FPGA 设计中是老生常谈的问题

下面看看一个实例，相信你就懂了：

```
always @(posedge clk)
  begin
    b <= a;
    c <= b;
  end
```

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




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

也会帮你对电路进行简化。

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。

```
always @ (a or b or c)
  begin
    Z = a or b
  end
```

12. 复位&初始化

相信你看过的经典代码中，一般都会包含有复位信号 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
endmodule
```

# 第八章数字基础实验

## 8.1 分频器的设计

一、分频器简介

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,分频电路时加法计数器的”变异”,其计数值有分频叙述决定,器输出不是一般计数器的技术结果,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 的分频器

对于分频系数为 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_50）。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：

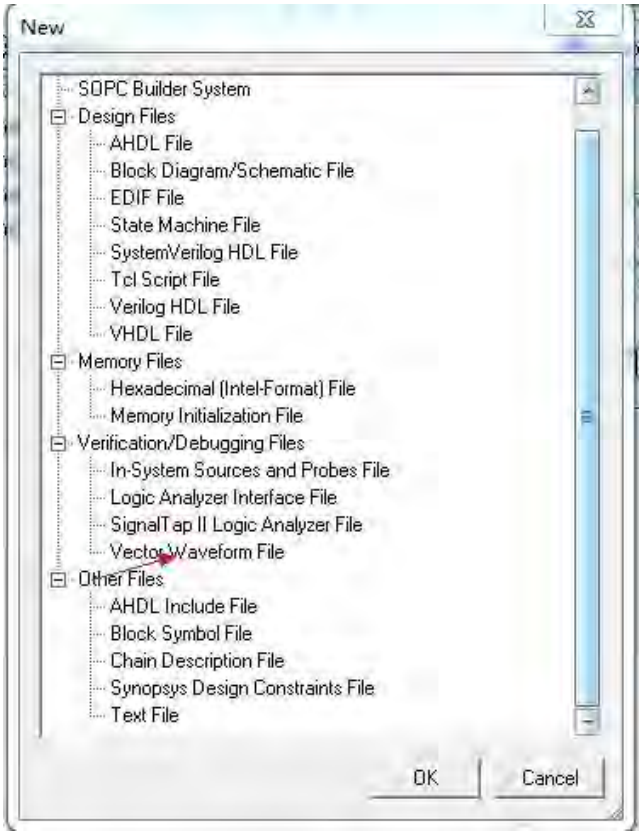
```
module fengping_2(clk_50M,f_10);
input     clk_50M;          //系统输入时钟，50M，周期 20ns
output    f_10;             //10 分频输出，5M

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，在本例程笔者作简要以下介绍。

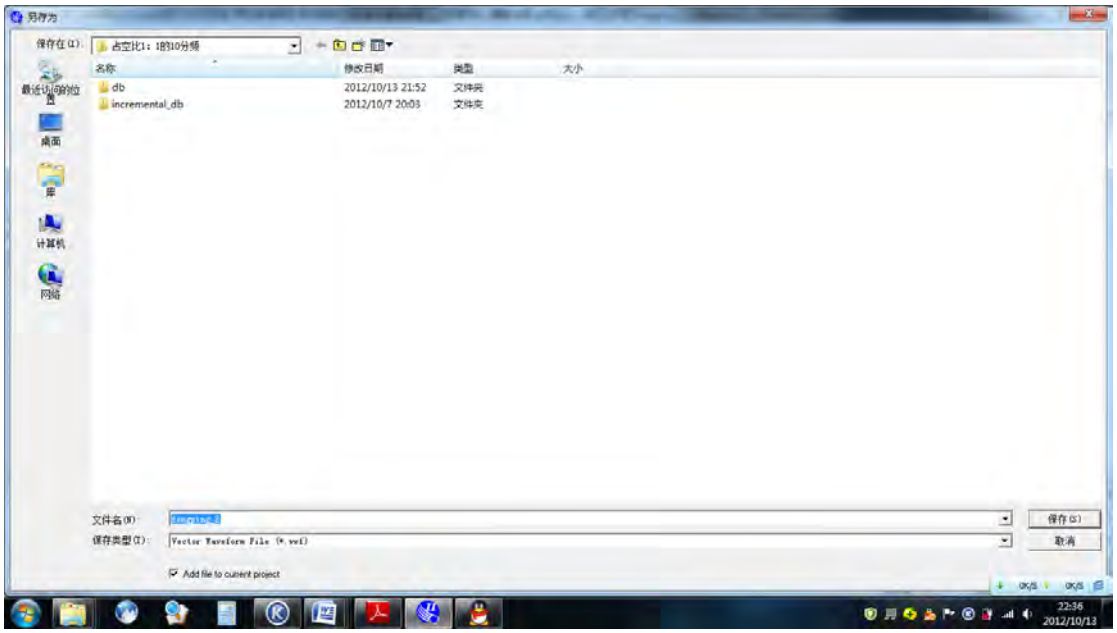
1. 首先建立波形仿真的文件：

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

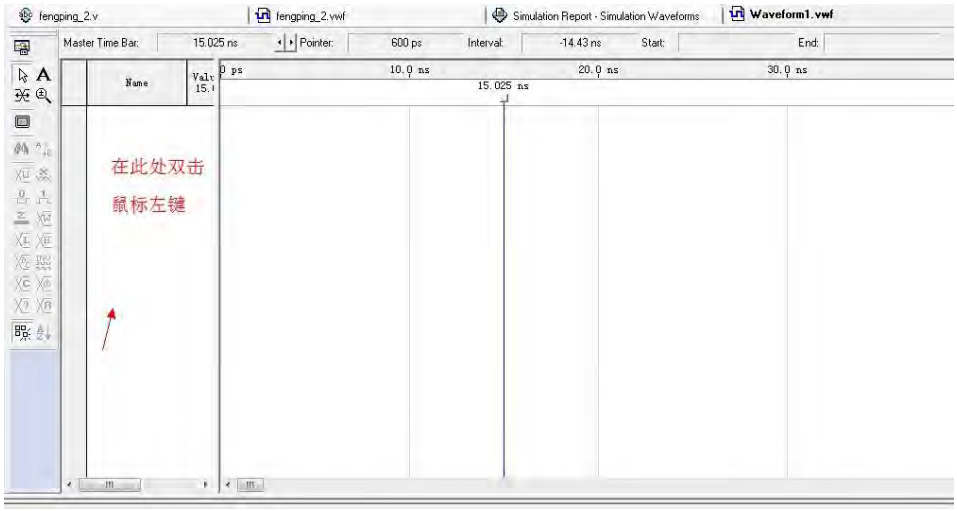



2. 对建立的仿真文件进行保存：

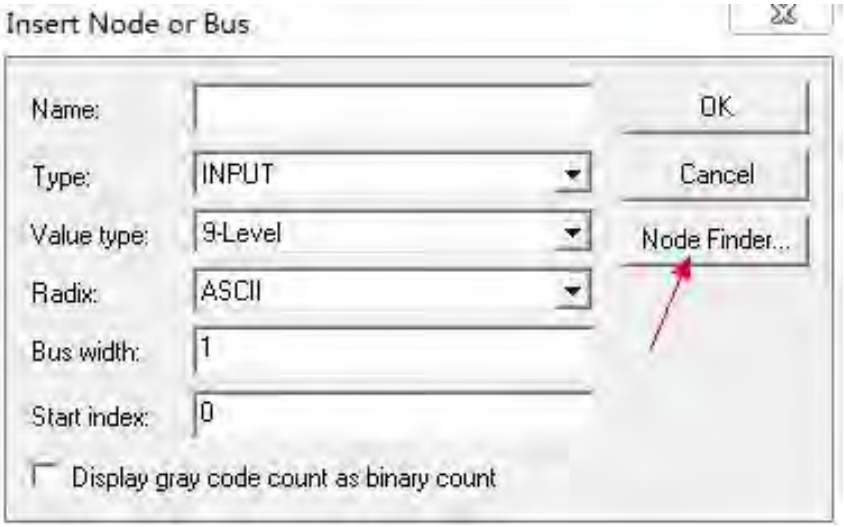
按软件左上脚的保存  ，在弹出的窗口中按保存，如下图所示。



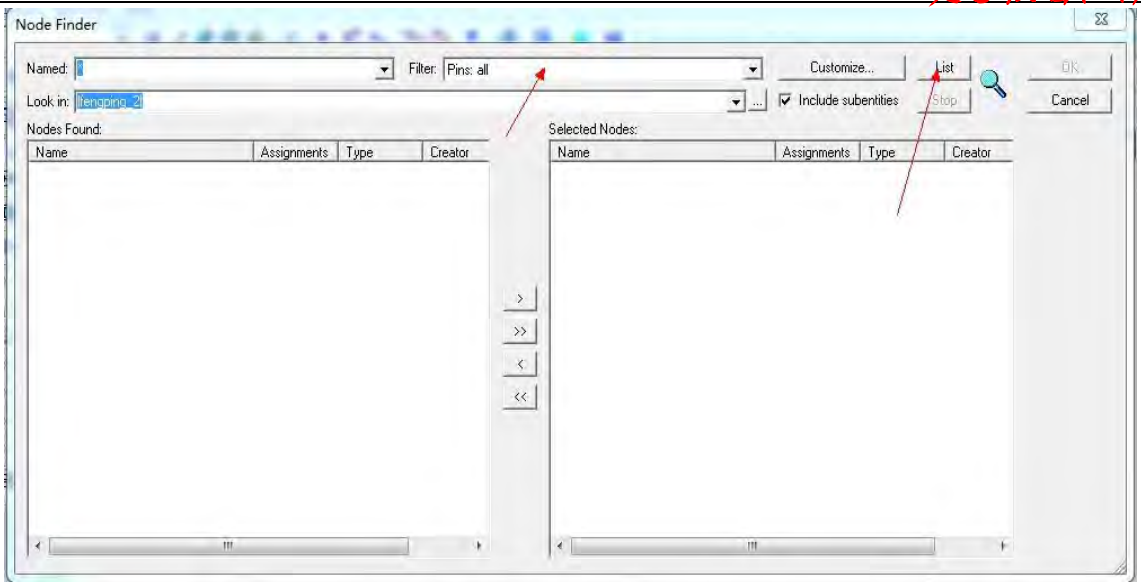
3. 在新建的仿真文件中导入需要仿真的信号，在 new 栏目的空白处双击鼠标左键，按下图所示：



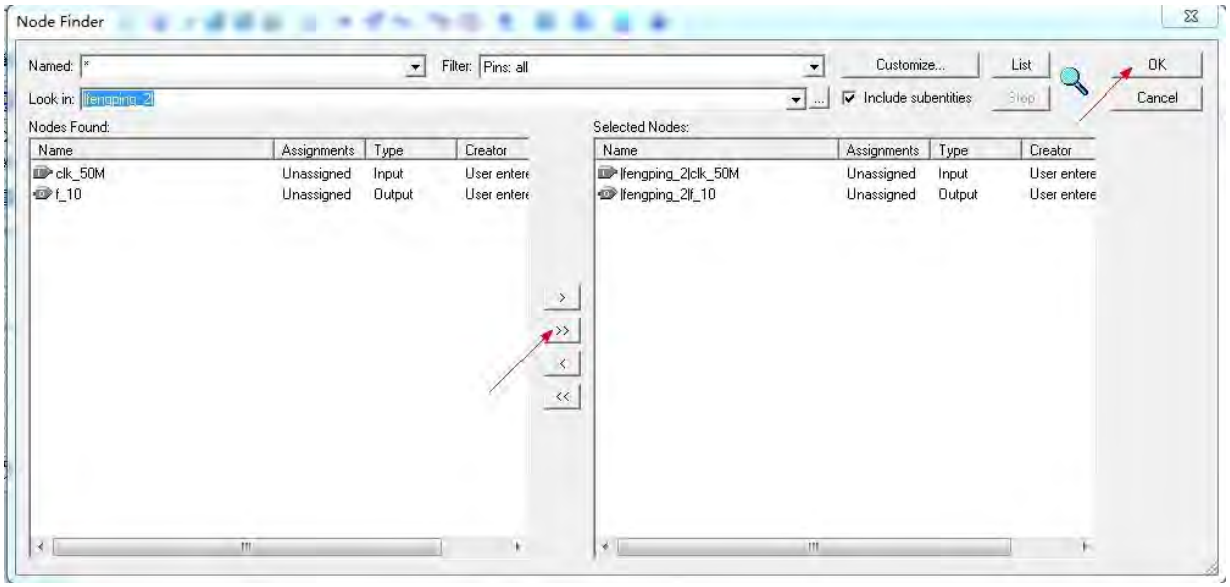

4. 在弹出的对话框中，选中 Node Finder 选项，如下图所示：



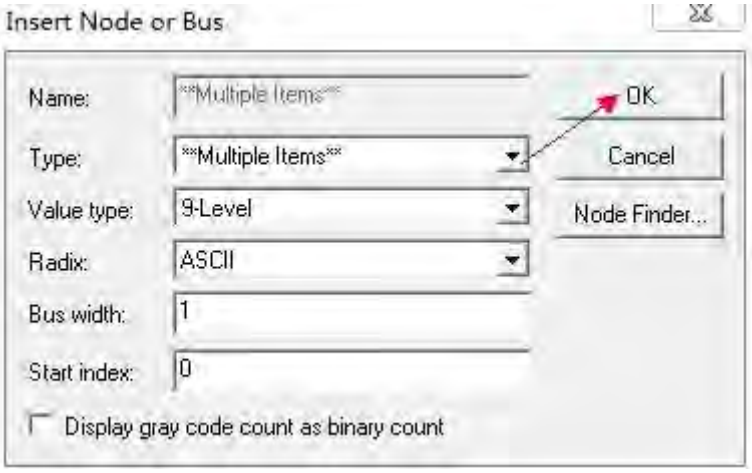

5. 在 Fiter 栏目中选中 Pin:all 选项，然后点击 List 按钮，如下图所示：



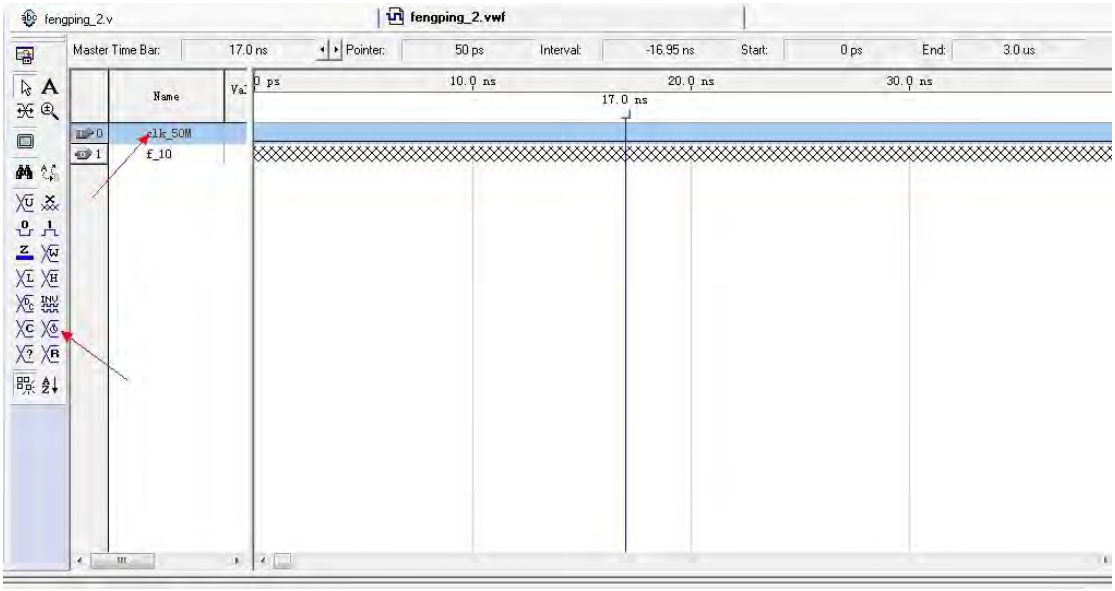

6. 将左边的端口导入到右边的窗口，按如图所示按键，然后点击 OK 按钮，如下图所示：



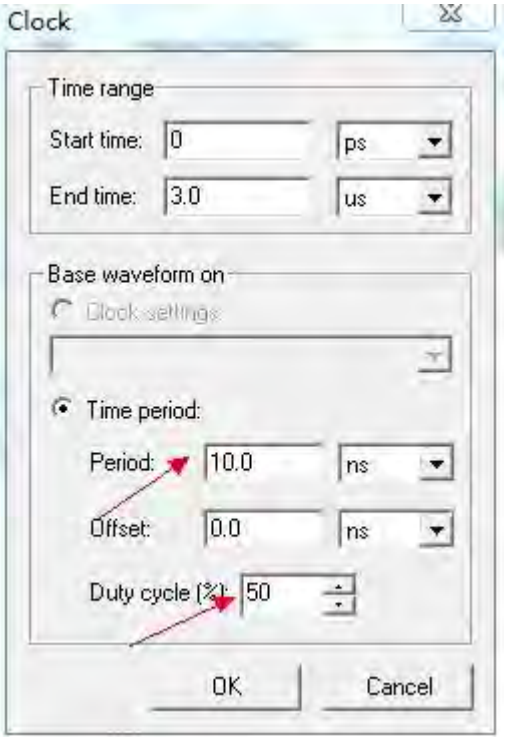

7. 在弹出的对话框中选中 OK 的按钮，如下图所示：



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

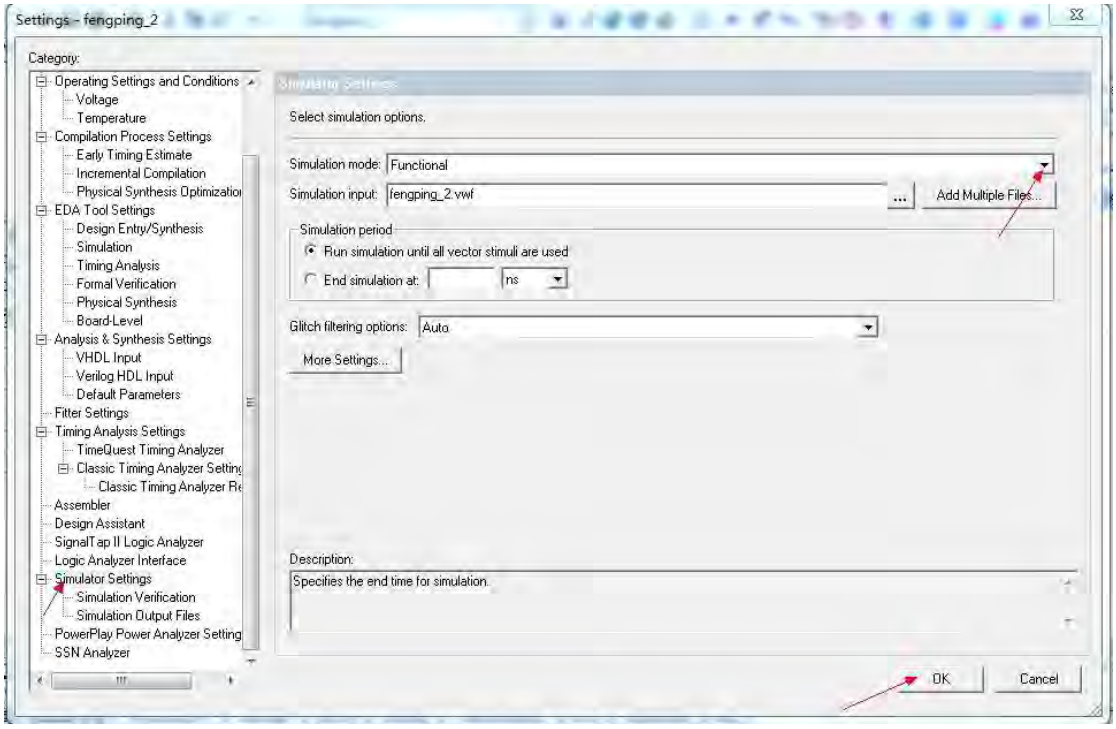



9. 在弹出的对话框中，可以设置系统时钟的占空比以及时钟周期，此处我们的占空设置为 50%，时钟周期为 10ns，然后点击 ok 即可，如下图所示：

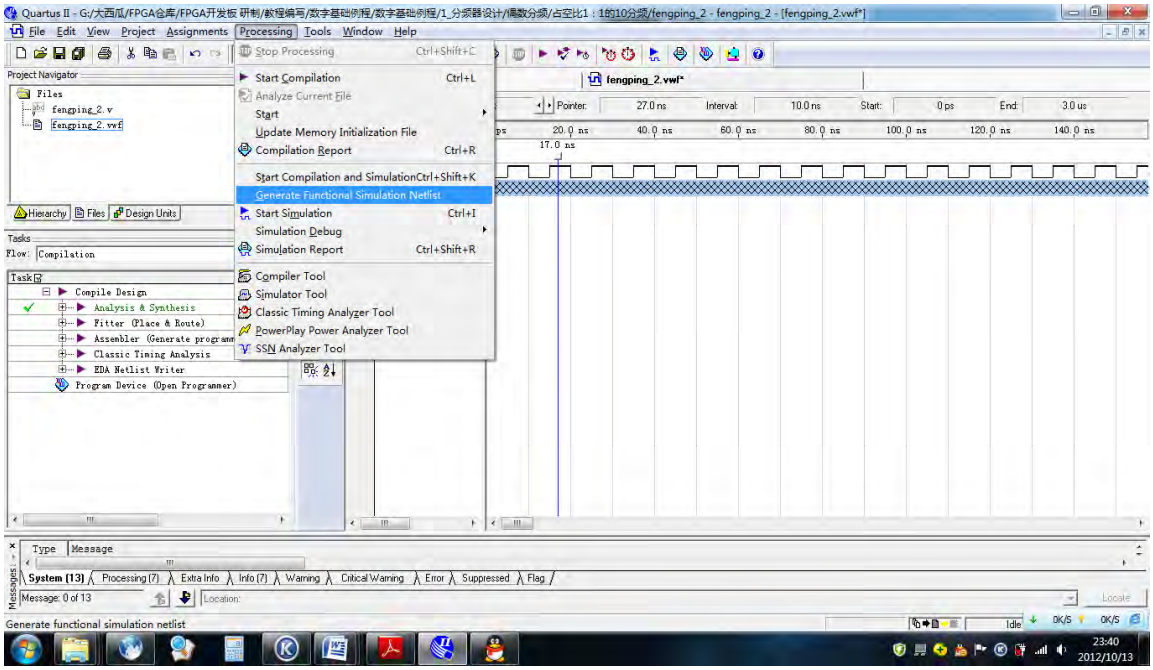



10. 首先，进行功能仿真，什么是功能仿真呢，功能仿真简单来说既是理想的运行情况，不会存在任何的逻辑器件的延时（事实上是不存在的）。首先在软件上方点击红色箭头所示的设置项，对所建立的工程进行设置：

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：

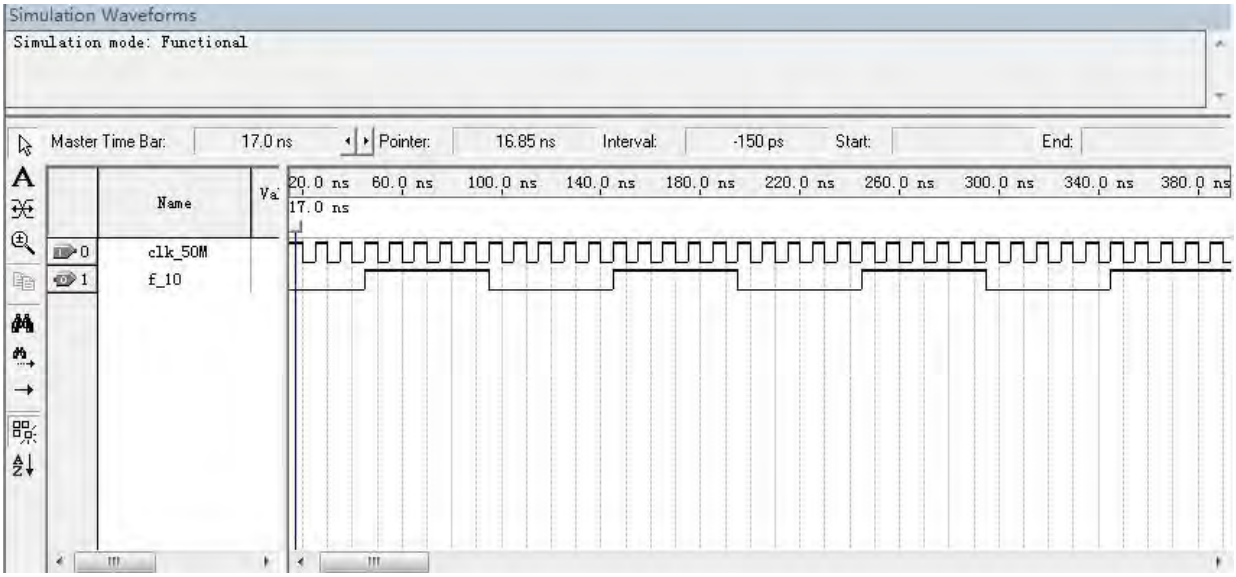



12. 在 Processing 栏目中选中 Generate Functional Simulation Netlist 选项，从而产生仿真网表文件，如下图所示：



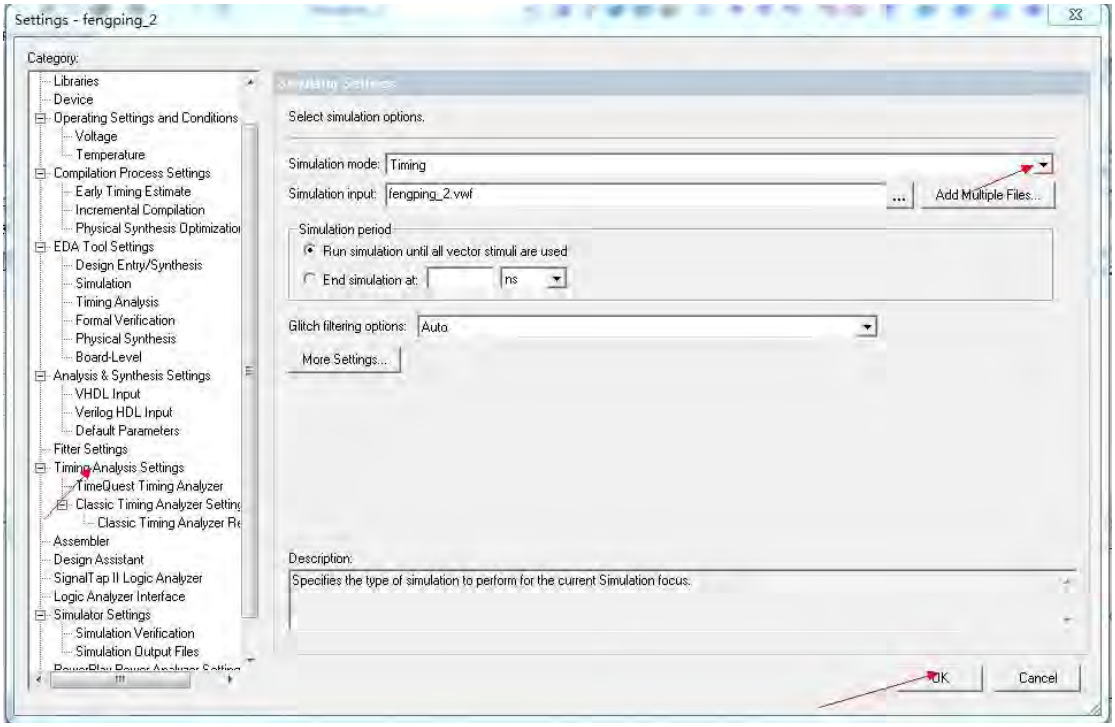

13. 点击软件上方的仿真按钮  ，如下图所示：

14. 等待软件运行后，功能仿真波形即可出来：

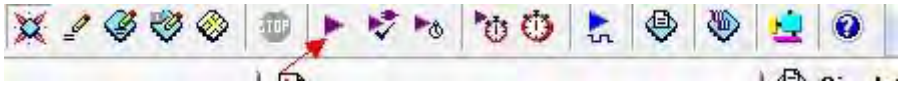



15. 下面进行另外一种仿真，时序仿真，时序仿真先对与功能仿真而言其仿真结果比较接近实际情况，即是存在逻辑器件的时间延迟。首先在软件上方点击红色箭头所示的设置项，对所建立的工程进行设置：

16. 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 Timing 选项，既功能仿真，最后按 ok 选项，如下图所示：

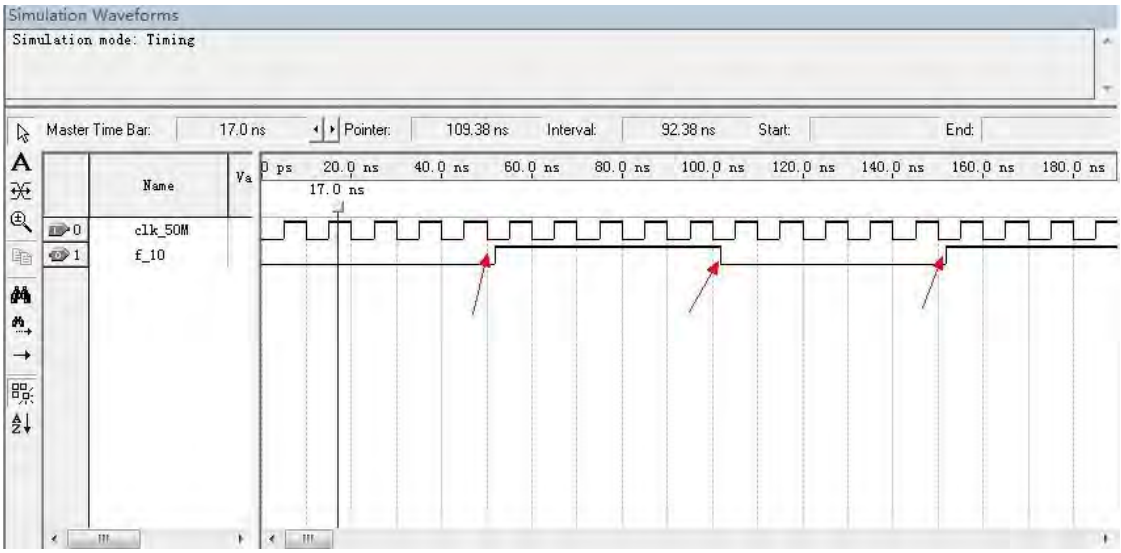



17. 对工程进行全编译，按全编译按钮  ，如下图所示：



18. 点击软件上方的仿真按钮  ，如下图所示：

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，时序反正在每个红色箭头处都出现了相应的时序延时：




## 8.2 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~m-1 或则 m-1~0。其基本原理就是将几个触发器按照一定的顺序链接起来，然后更具触发器的组合状态，按照一定的技术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：

| 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 |

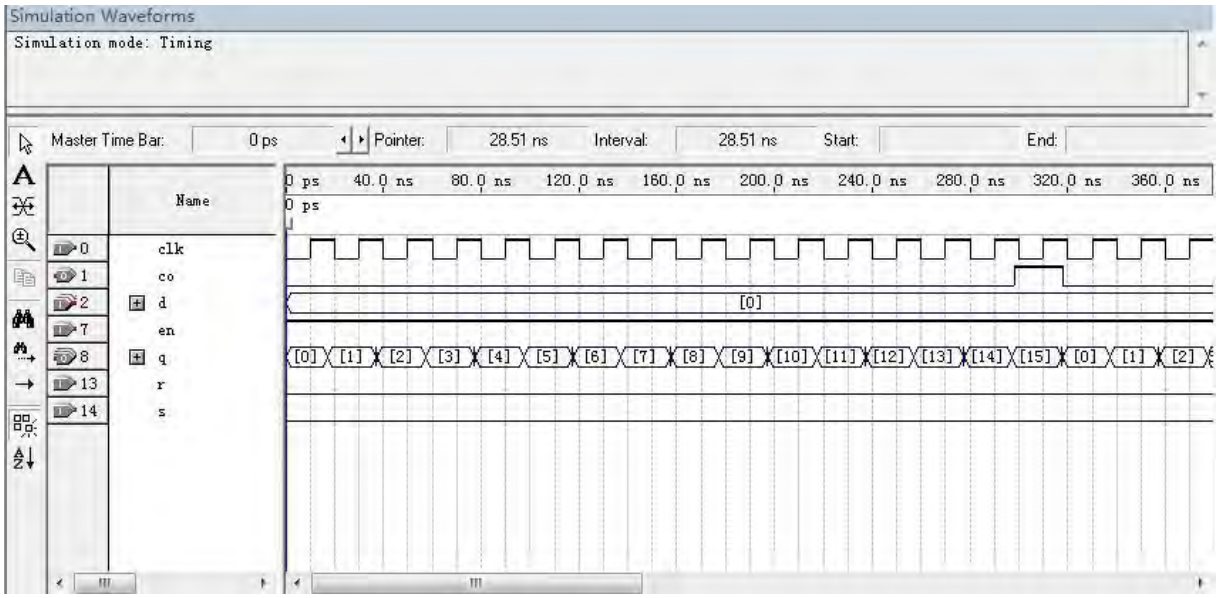
本例程的 verilog 程序为：

```
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：


b) 时序仿真结果：




## 8.3D 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 CLOCK 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。

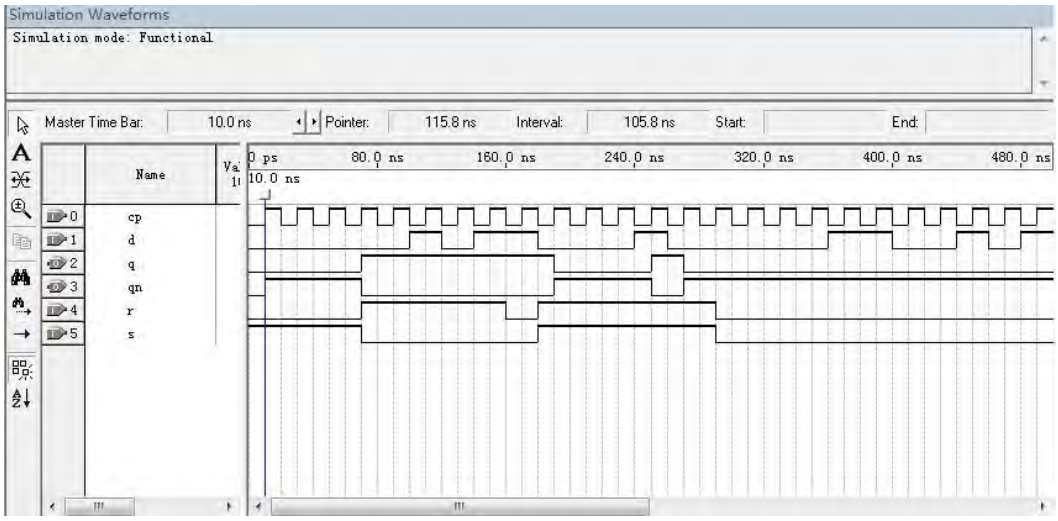
| 输入 | | | | 输出 | |
|---|---|---|---|---|---|
| CP | R | S | D | Q | ~Q |
| X | 0 | 1 | X | 0 | 1 |
| X | 1 | 0 | X | 1 | 0 |
| 0 | 1 | 1 | X | 保持 | 保持 |
| ↑ | 1 | 1 | 0 | 0 | 1 |
| ↑ | 1 | 1 | 1 | 1 | 0 |

本例程的 Verilog 程序为：

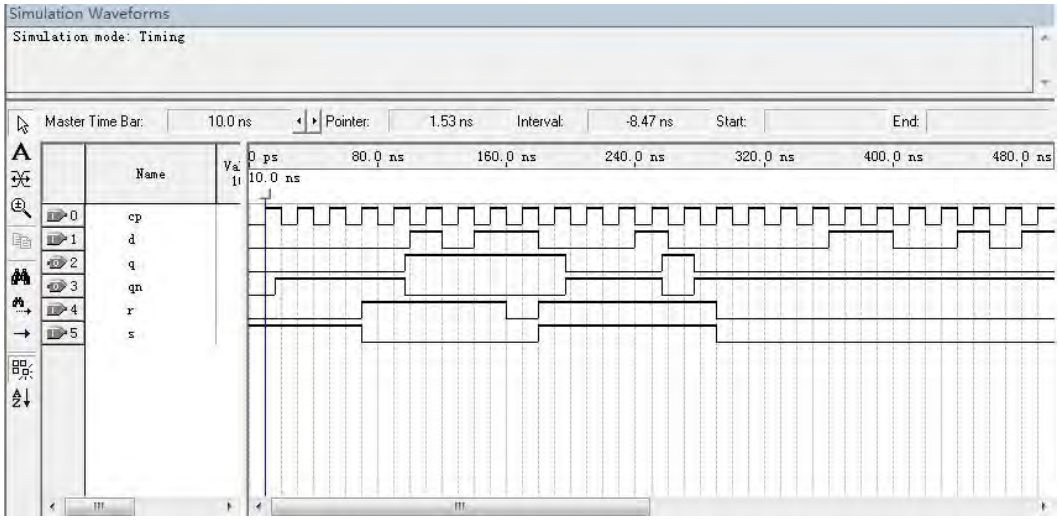
```
module D(q,qn,d,cp,r,s);
output q,qn;          //D 触发器的两个输出
input d,cp,r,s;       //D 触发器的四个输入

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
    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：




b) 时序仿真结果：




## 8.4 三态门

一、三态门简介:

三态门是置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。工作状态相对于隔离状态。三态门都有一个 EN 使能控制端，来控制门电路的通断。

二、三态门的设计:

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。

| 输入 | | | | 输出 | |
|---|---|---|---|---|---|
| CP | R | S | D | Q | ~Q |
| X | 0 | 1 | X | 0 | 1 |
| X | 1 | 0 | X | 1 | 0 |

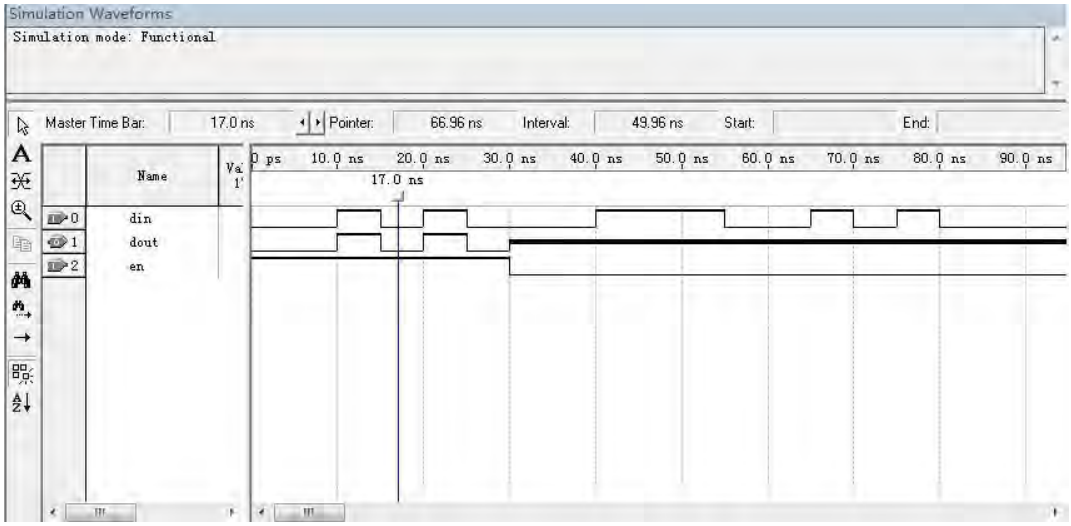
| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 1 | X | 保持 | 保持 |
| ↑ | 1 | 1 | 0 | 0 | 1 |
| ↑ | 1 | 1 | 1 | 1 | 0 |

本例程的 Verilog 程序为：

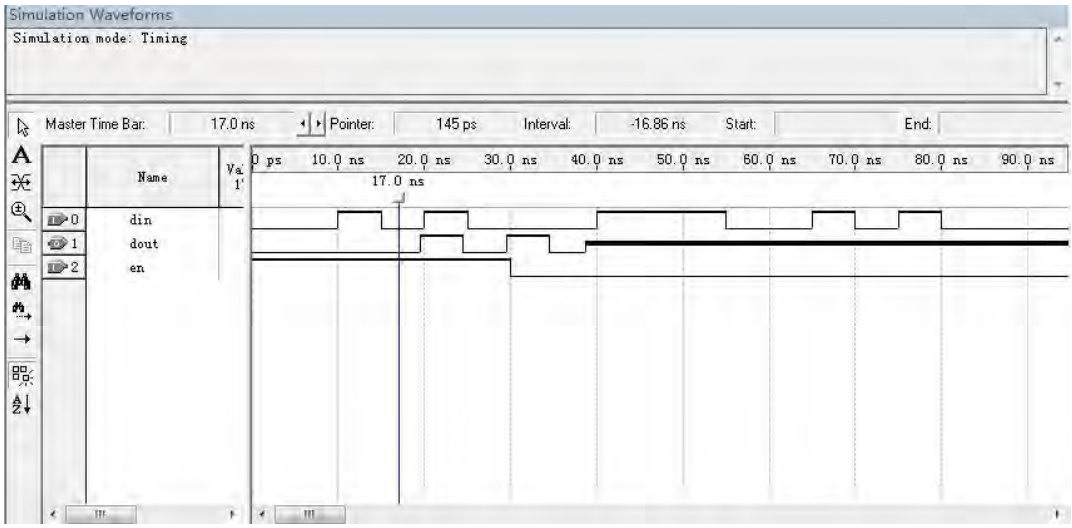
```
module tri_gata(dout,din,en);
input din,en;          //三态输入端，使能输入
output dout;           //三态输出端

assign dout=en?din:'bz; //判断 en 是否为 1，是的话，把 din 赋予 dout，否的话把
                        //dout 置高阻

endmodule
```

三、D 触发器的波形仿真：

a) 功能仿真结果：




b) 时序仿真结果：



# 8.5 8-3 编码器

一、8-3 编码器简介:

在数字系统里,此次需要将莫伊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。入，8-3 编码器有 8 个输入、3 个二进制输出。其真值表如下：

| 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]

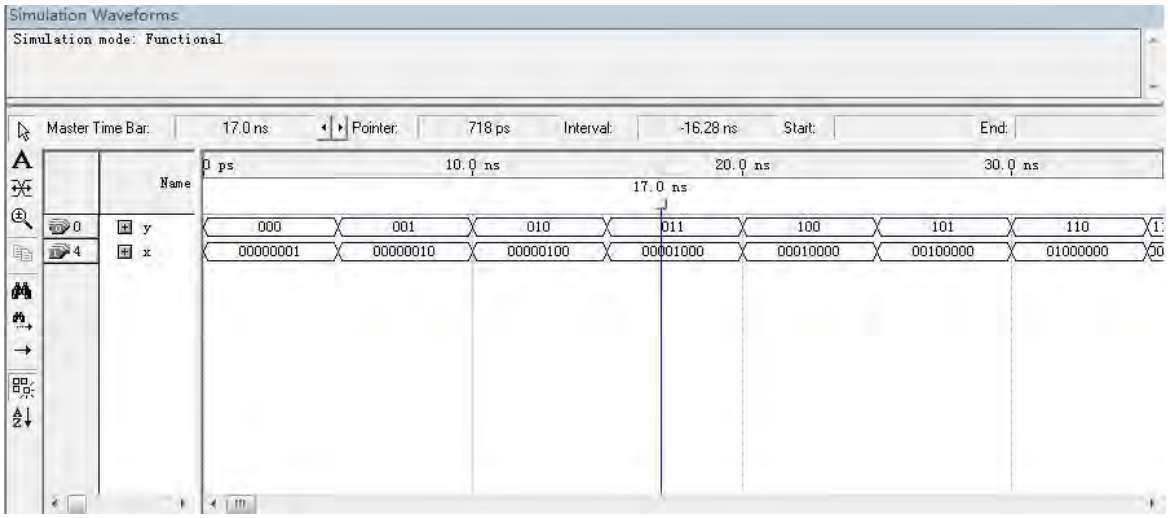
本例程的 Verilog 程序：

```
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
```

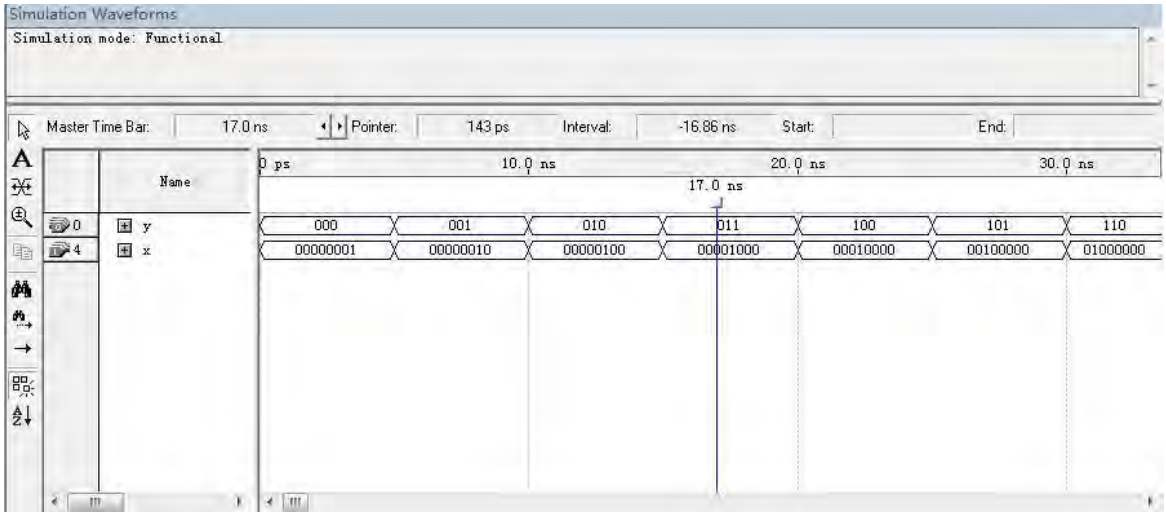
endmodule

三、8-3 编码器的波形仿真：

c) 功能仿真结果：




d) 时序仿真结果：




## 8.6 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,若同时又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 wie 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：

| 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：

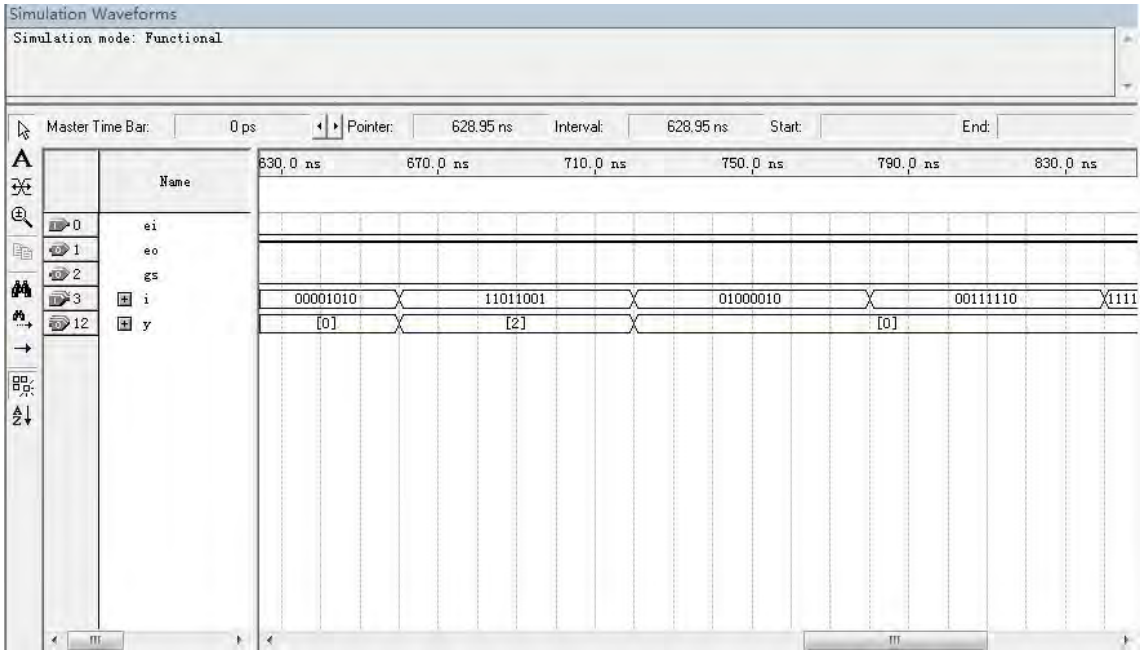
```
module yxbm_83(y,eo,gs,i,ei);
input [7:0] i;          //8 位输入 i
input ei;               //使能输入端 ei
output eo,gs;           //使能输出端 eo，优先标志端 gs
output[2:0] y;          //3 位输出 y

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;
```

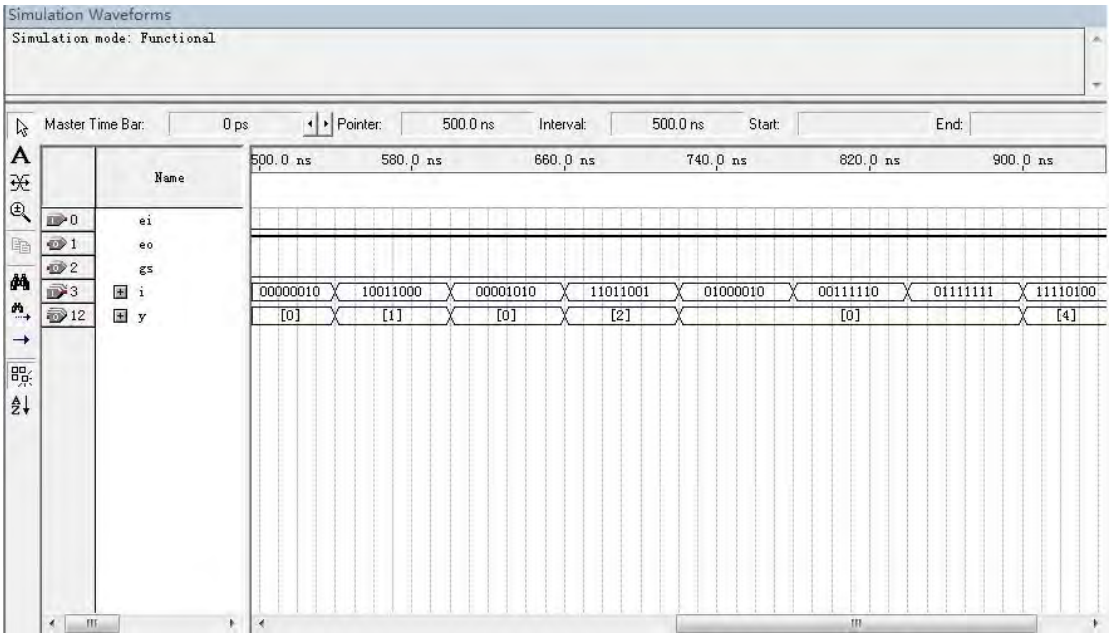
```
                    eo<=1'b1;
                end
            else if(i[0]==1'b0)//当 i 的第 1 位为 0 时候
                begin
                    y<=3'b111;
                    gs<=1'b0;
                    eo<=1'b1;
                end
            else if(i==8'b11111111)//当 i 为 8'b11111111 时候
                begin
                    y<=3'b111;
                    gs<=1'b1;
                    eo<=1'b0;
                end
        end
    end

    endmodule
```

三、8-3 优先编码器的波形仿真：

e) 功能仿真结果：




f) 时序仿真结果：




## 8.7 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,并装换成控制信号,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，则最多可译码装换成为 2 的 n 个数据。其真值表如下：

| 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。

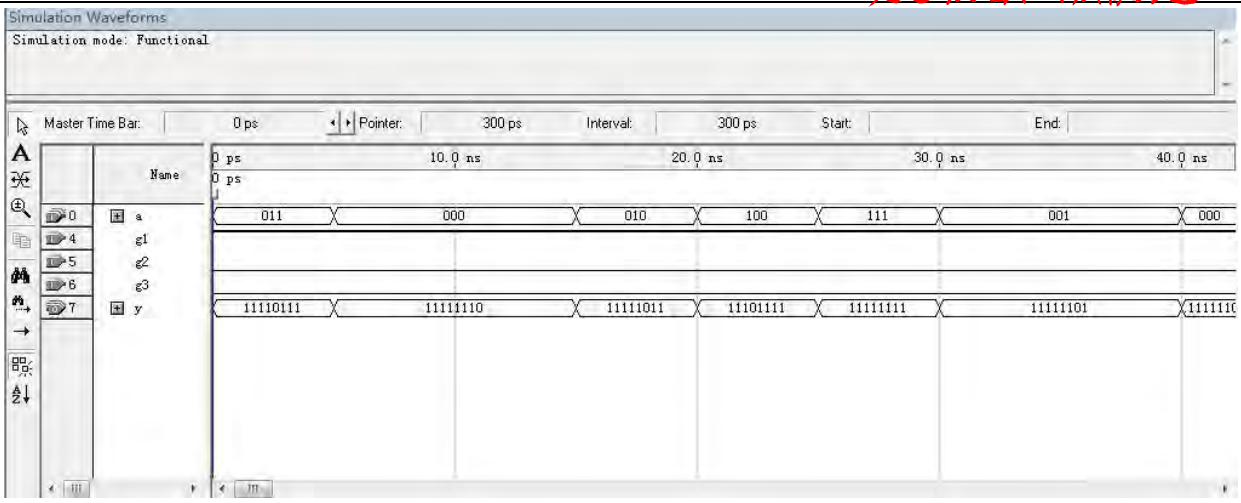
其中输入信号有：

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：

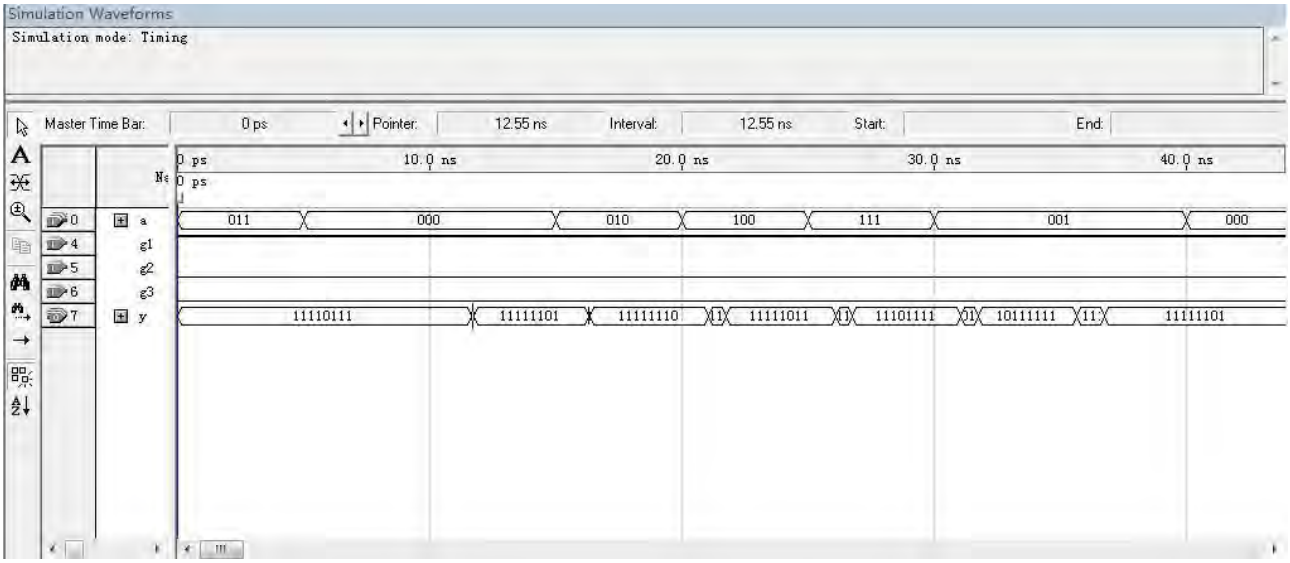
```
module ym_3_8(a,g1,g2,g3,y);
input[2:0] a;                    //3 位 2 进制编码输入端
input g1,g2,g3;                  //3 个使能输入端
output[7:0] y;                   //8 位编码输出端

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

endmodule
```

三、3-8 译码器的波形仿真：

g) 功能仿真结果：




h) 时序仿真结果：




## 8.8 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一次左移或

者右移，在数值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、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下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。

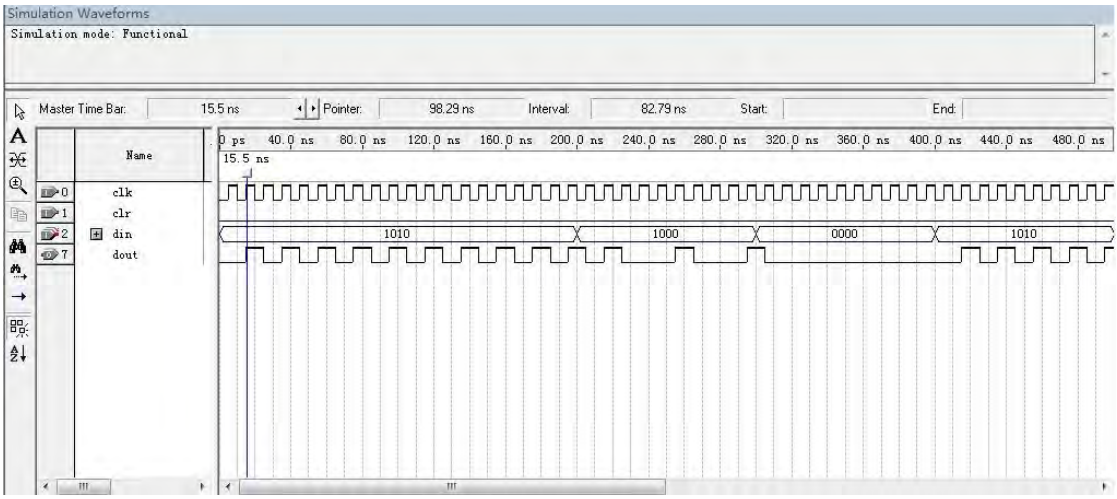
本例程的 Verilog 程序为：

```
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
```

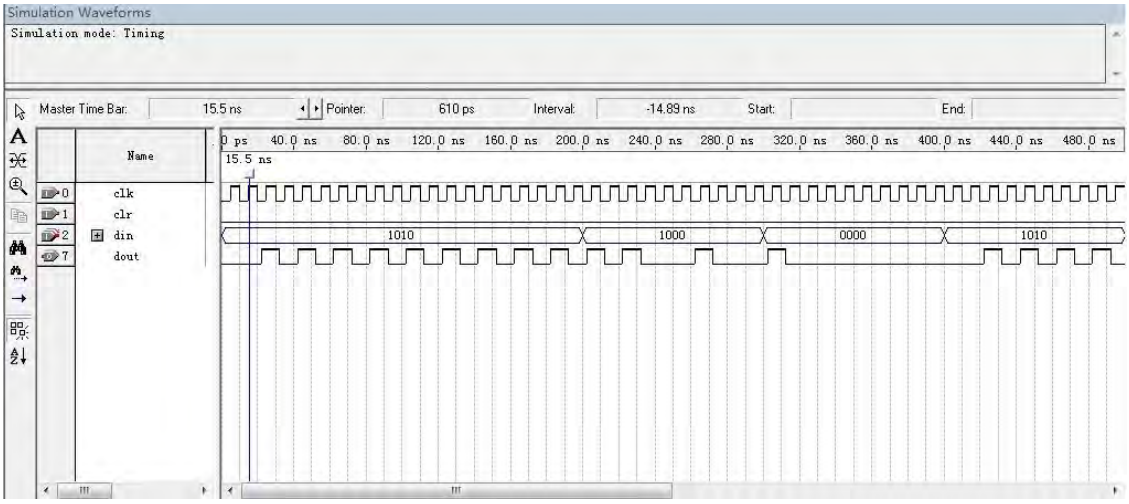
```
endmodule
```

三、移位寄存器的波形仿真：

i) 功能仿真结果：




j) 时序仿真结果：




# 8.9 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路逻辑电路称为数据选择器，它的作用相对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。四选一数据选择器书堆 4 个数据源进行选择，使用量为地址 A1A0 产生 4 个地址信号，由 A1A0 等于“00”、“01”、“10”、“11”来选择输出，器真值表如下：

| 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 |

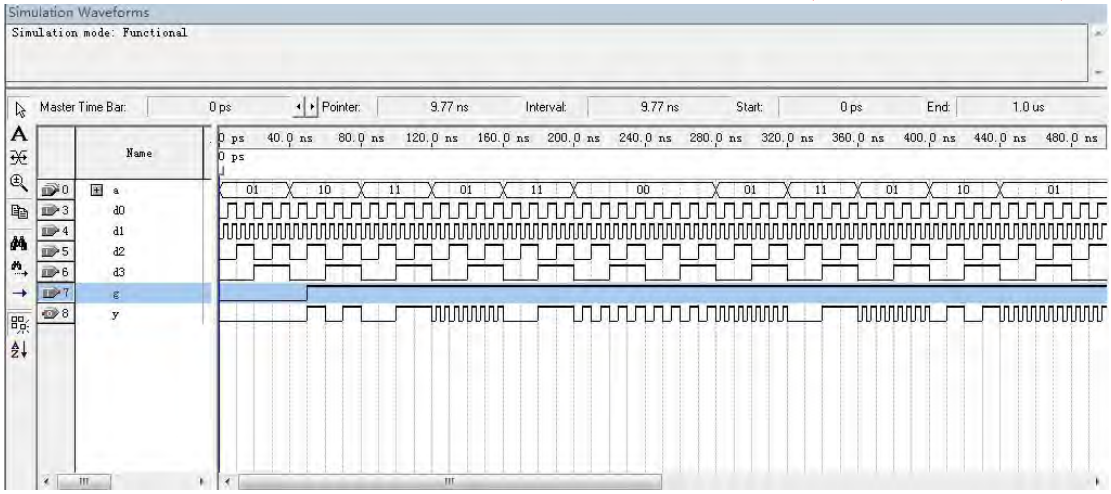
本例程的输入信号为：数据源 d0、d1、d2、d3；
两位地址码 a[1:0]、
使能端 g、
输出信号：选择输出端 y。

本例程的 Verilog 程序为：

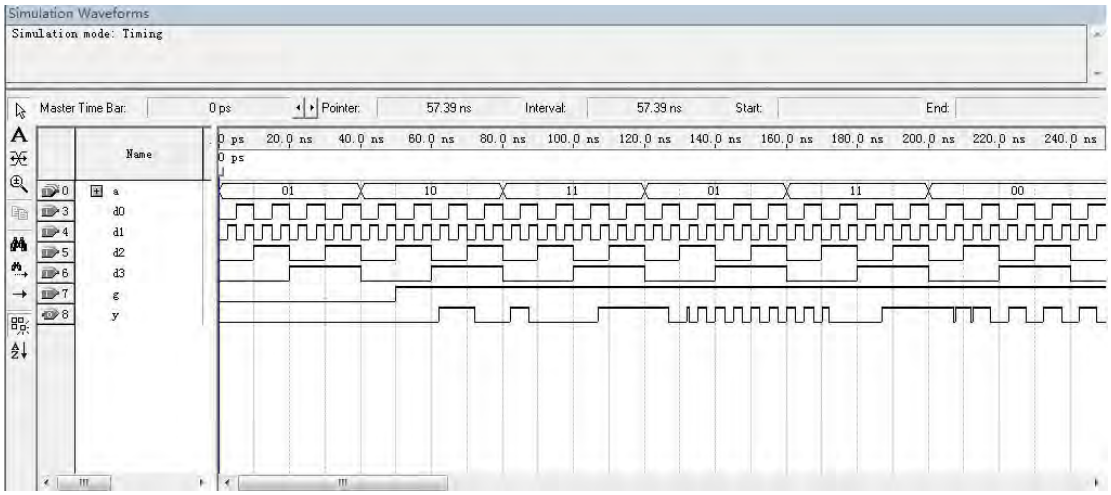
```
module mux4(d0,d1,d2,d3,y,a,g);
input d0,d1,d2,d3;          //输入的四个数据
input g;                    //输入使能端
input[1:0] a;               //输入的选择端
output y;                   //输出数据

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 有变化时候、//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、多路数据选择器的波形仿真：

k) 功能仿真结果：




l) 时序仿真结果：




## 8.10 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,更是计算机中不可后却的组成成分,包括半加器、全加器、多为全加器等。而全加器器即使指能进行加数、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：被加器 a、

加数 b、

低位进位 ci。

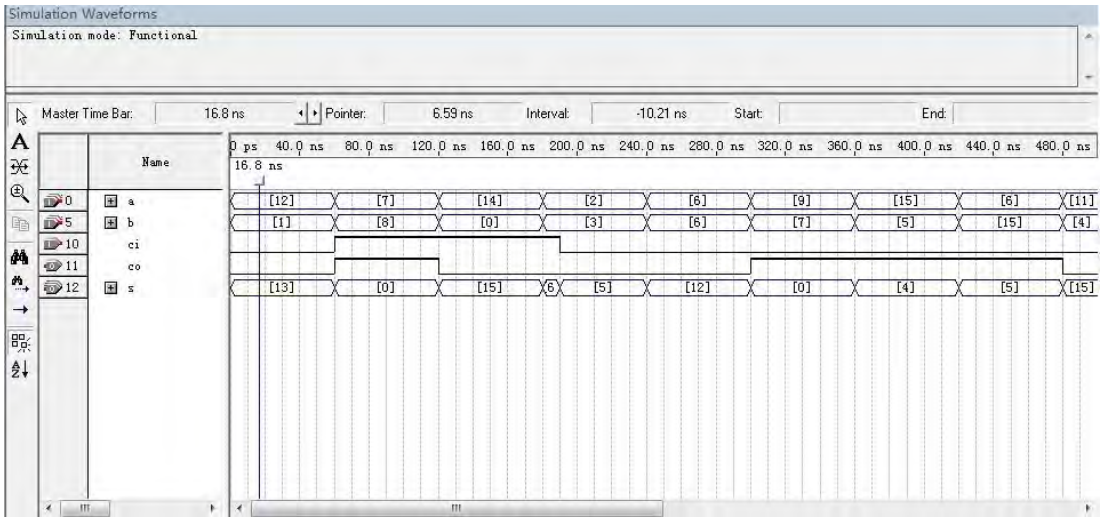
输出信号有：和数 s、
进位 co。

本例程的 Verilog 程序为：

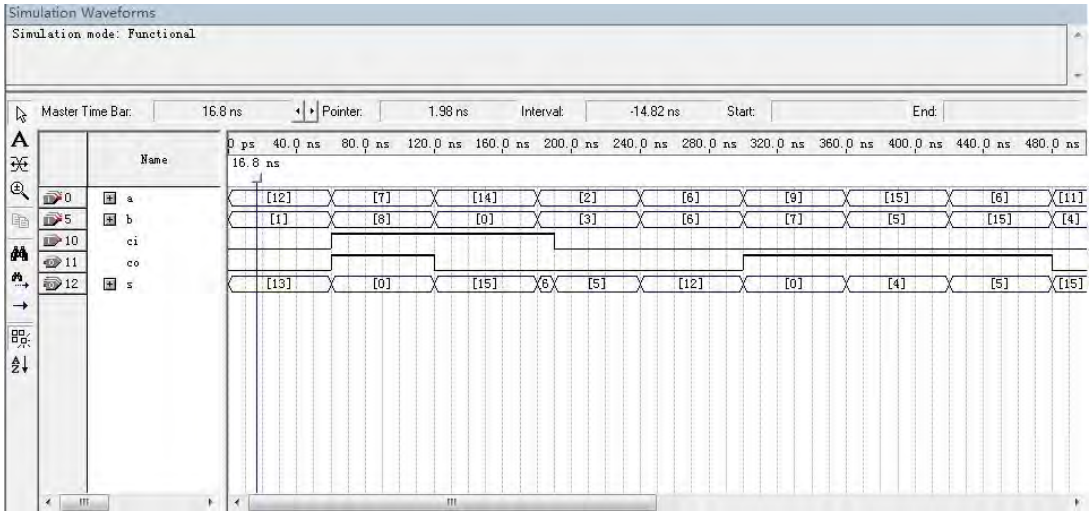
```
module add4(a,b,ci,s,co);
input [3:0] a,b;                    //输入四位数据 a，b
input ci;                           //输入进位 ci

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 a、b、ci 相加后的结果赋予 co、s，其中 co
                                    //放最高位，s 放低三位

endmodule
```

三、串行加法器的波形仿真：

m) 功能仿真结果：




n) 时序仿真结果：



# 8.11 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]。

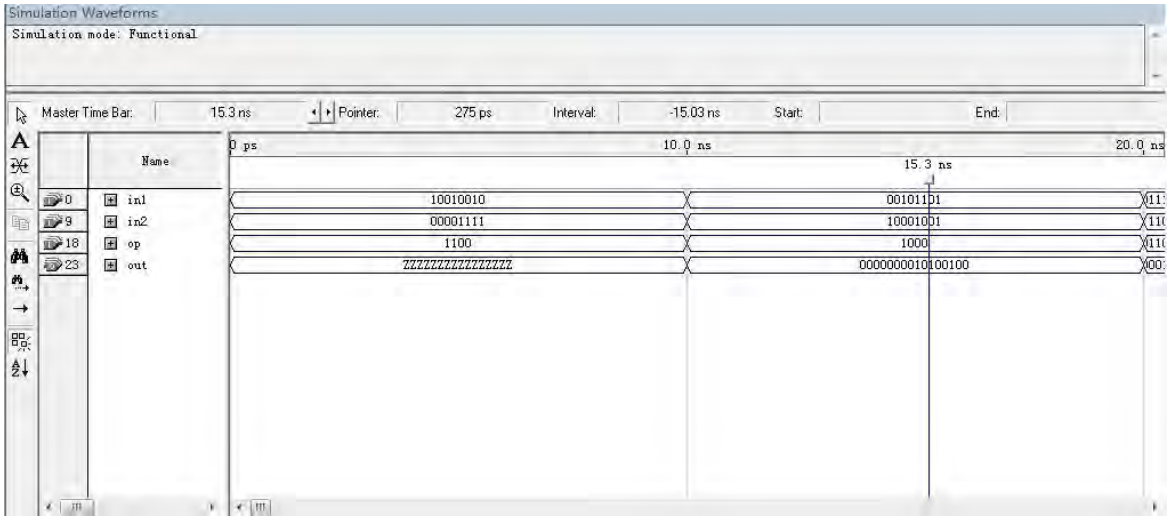
本例程的 Verilog 程序如下：

```
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,
          addtion    = 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;
```

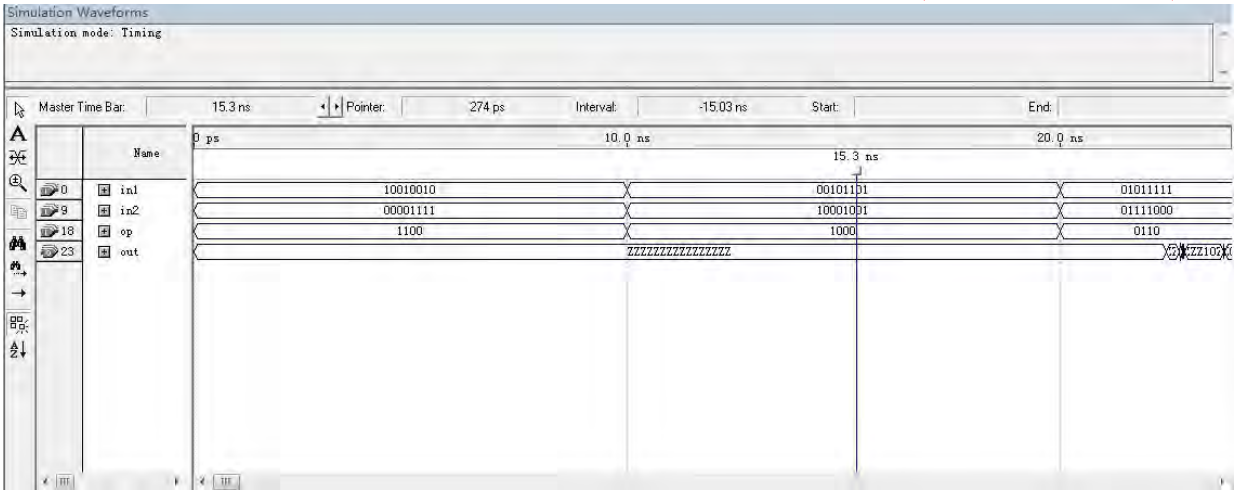
```
always @(in1 or in2 or op)                  //电平触发方式
  begin
    case (op)                               //case 语句选择运算操作
      transfer:     out=in1;                //赋值操作
      increase:     out=in1+1'b1;           //in1 自加一运算
    decrease:       out=in1-1'b1;           //in1 自减一运算
      addtion:      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 波形仿真：

o) 功能仿真结果：




p) 时序仿真结果：


Simulation Waveforms
Simulation mode: Timing
Master Time Bar: 15.3 ns
Pointer: 274 ps
Interval: -15.03 ns
Start:
End:
Name
0 ps
10.0 ns
20.0 ns
15.3 ns
0 in1 10010010 00101101 01011111
9 in2 00001111 10001001 01111000
18 op 1100 1000 0110
23 out ZZZZZZZZZZZZZZZZ ZZZ10Z

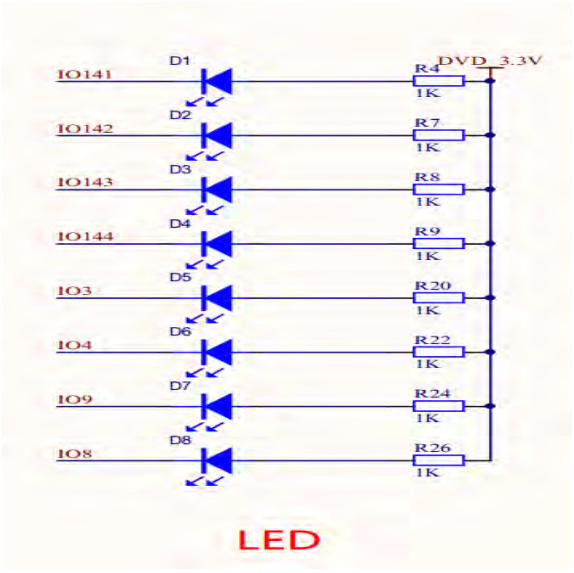
# 第九章基础实验

## 9.1 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，但不要超过 2K。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：




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 8 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。

**循环幅值：**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 幅值。

2. **实验程序**

**移位方式：**

```
always @ (posedge clk or negedge reset)
begin
```

```
 if(!reset)//复位
begin
                led_temp<=8'h80;
                count<=41'h0;
end
else
begin
              count<=count+1;
              if(count==41'hff_ff_ff)
                 begin
                  led_temp<=led_temp>>1;
                  led<=~led_temp;
                  count<=0;
                  if(led_temp==8'h01)
                     led_temp<=8'h80;
                 end
          end
    end
```

循环幅值：

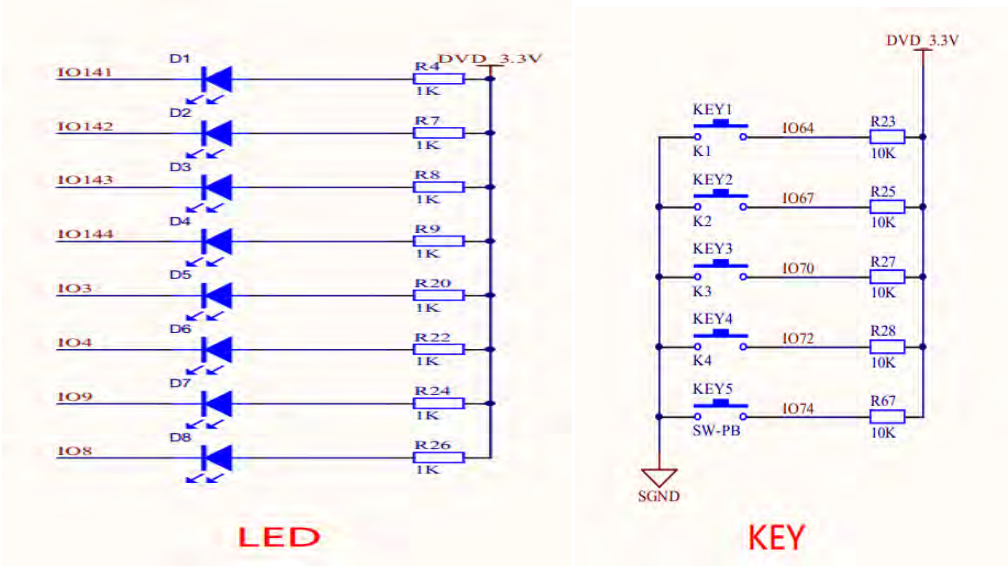
```
    always @(posedge clk)//显示时钟
      begin
          if(counter==24'h500000)
            begin
               clk_div<=~clk_div;
               counter<=24'h000000;
            end
          else
               counter<=counter+1'b1;
      end
    always @(posedge clk_div or negedge reset)
      begin
        if(!reset)
          begin
           led<=8'hff;
           led_state<=4'b0000;//时钟控制状态
          end
        else
         begin
          case (led_state)
            5'b00000: led<=8'b1111_1110;
            5'b00001: led<=8'b1111_1101;
            5'b00010: led<=8'b1111_1011;
            5'b00011: led<=8'b1111_0111;
```

```
            5'b00100: led<=8'b1110_1111;
            5'b00101: led<=8'b1101_1111;
            5'b00110: led<=8'b1011_1111;
            5'b00111: led<=8'b0111_1111;

            5'b01000: led<=8'b1011_1111;
            5'b01001: led<=8'b1101_1111;
            5'b01010: led<=8'b1110_1111;
            5'b01011: led<=8'b1111_0111;
            5'b01100: led<=8'b1111_1011;
            5'b01101: led<=8'b1111_1101;
            5'b01110: led<=8'b1111_1110;

            5'b01111: led<=8'b1110_0111;
            5'b10000: led<=8'b1101_1011;
            5'b10001: led<=8'b1011_1101;
            5'b10010: led<=8'b0111_1110;

            5'b10011: led<=8'b1011_1101;
            5'b10100: led<=8'b1101_1011;
            5'b10101: led<=8'b1110_0111;

            5'b10110: led<=8'b1010_1010;
            5'b10111: led<=8'b0101_0101;

            5'b11000: led<=8'b1000_0000;
            5'b11001: led<=8'b0100_0000;
            5'b11010: led<=8'b0010_0000;
            5'b11011: led<=8'b0001_0000;
            5'b11100: led<=8'b0000_1000;
            5'b11101: led<=8'b0000_0100;
            5'b11110: led<=8'b0000_0010;
            5'b11111: led<=8'b0000_0001;
            default:led<=8'b1111_1111;
        endcase
      led_state<=led_state+1'b1;
    end
end
```

# 9.2 PWM 控制灯的亮暗

## 实验硬件说明

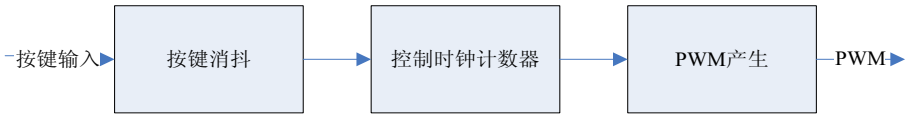
## 实验的原理图：




**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。由一个独立按键 key 来控制 FPGA 输出脉冲的占空比，用该脉冲去直接驱动发光二极管 LED，随着占空比不同，LED 的亮度将不同。

设计原理框图：



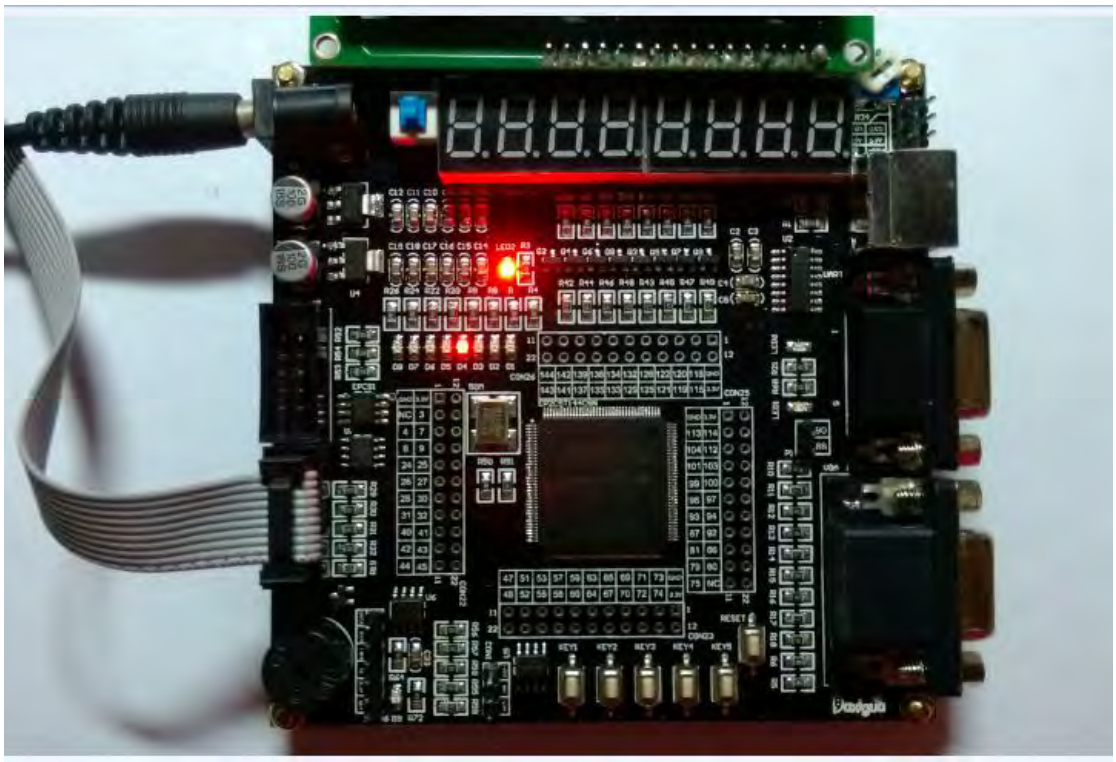

**2. 实验程序**

```
/*******************状态机按键消抖**************************/
always @(posedge clk)
 begin
    case (state)
     s0:
      begin
       key_out<=1'b1;
       if(key==1'b0)
          state<=s1;
       else
          state<=s0;
      end
     s1:
```

```
      begin
       if(key==1'b0)
         state<=s2;
       else
         state<=s0;
       end
     s2:
      begin
       if(key==1'b0)
         state<=s3;
       else
         state<=s0;
       end
     s3:
       begin
        if(key==1'b0)
         begin
         key_out<=1'b0;
         state<=s3;
         end
        else
         begin
         key_out<=1'b1;
         state<=s0;
         end
       end
     default:
         state<=s0;
   endcase
 end
/******************PWM 产生模块*************************/
always @(posedge clk)
 begin
   case (state)
    s0:
     begin
      key_out<=1'b1;
      if(key==1'b0)
         state<=s1;
      else
         state<=s0;
     end
    s1:
```

```
      begin
       if(key==1'b0)
         state<=s2;
       else
         state<=s0;
       end
     s2:
      begin
       if(key==1'b0)
         state<=s3;
       else
         state<=s0;
       end
     s3:
       begin
        if(key==1'b0)
         begin
         key_out<=1'b0;
         state<=s3;
         end
        else
         begin
         key_out<=1'b1;
         state<=s0;
         end
       end
     default:
         state<=s0;
    endcase
 end
always @(posedge clk)
 begin
   clk_counter<=clk_counter+1'b1;
    if (clk_counter[13:4]<pwm_counter)
        pwm_out=1;
    else
        pwm_out=0;

    if (clk_counter[15]==1'b1)
    begin
        if (flag==1'b1)
        begin
            flag<=1'b0;
```

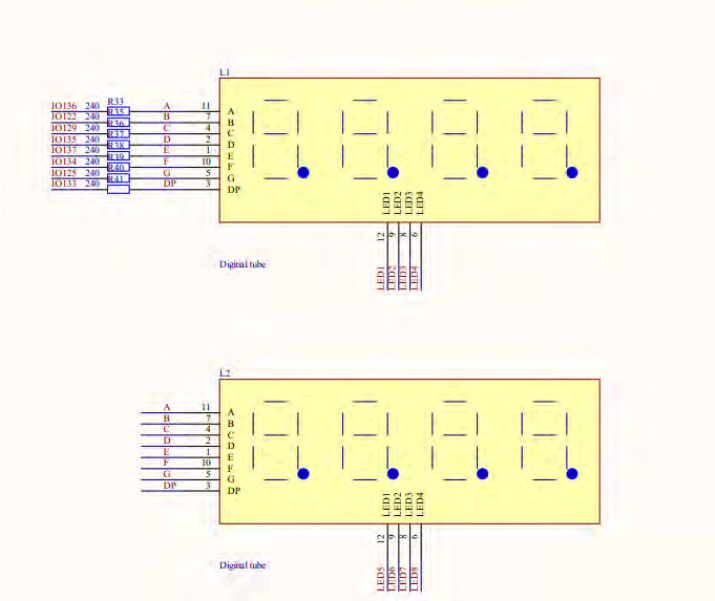
```
            if (key_out==1'b0)
                pwm_counter<=(pwm_counter+10'b0000000011);
            else
                pwm_counter<=pwm_counter;
        end
    end
    else
    flag<=1'b1;
 end
assign led=pwm_out;
always @(posedge clk)
 begin
   clk_counter<=clk_counter+1'b1;
    if (clk_counter[13:4]<pwm_counter)
        pwm_out=1;
    else
        pwm_out=0;

    if (clk_counter[15]==1'b1)
    begin
        if (flag==1'b1)
        begin
            flag<=1'b0;
            if (key_out==1'b0)
                pwm_counter<=(pwm_counter+10'b0000000011);
            else
                pwm_counter<=pwm_counter;
        end
    end
    else
    flag<=1'b1;
 end
assign led=pwm_out;
```



## 9.3 数码管的动态显示

## 共阳八段数码管

实验的原理图：




1. **实验设计原理**

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。

所谓动态扫描显示即轮流向各位数码管送出字形码和相应的位选，只要扫描显示速度够快，利用发光管的余辉和人眼视觉暂留作用，使人的感觉好像各位数码管同时都在显示。动态显示的亮度比静态显示要差一些，所以在选择限流电阻时应略小于静态显示电路中的。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，每个数码管显示时间不能太长也不能太短，时间太长会影响刷新率，导致总体显示呈现闪烁的现象，时间太短发光二极管的电流导通时间也就短，会影响总体的显示亮度。一般控制在 1ms 左右最佳。

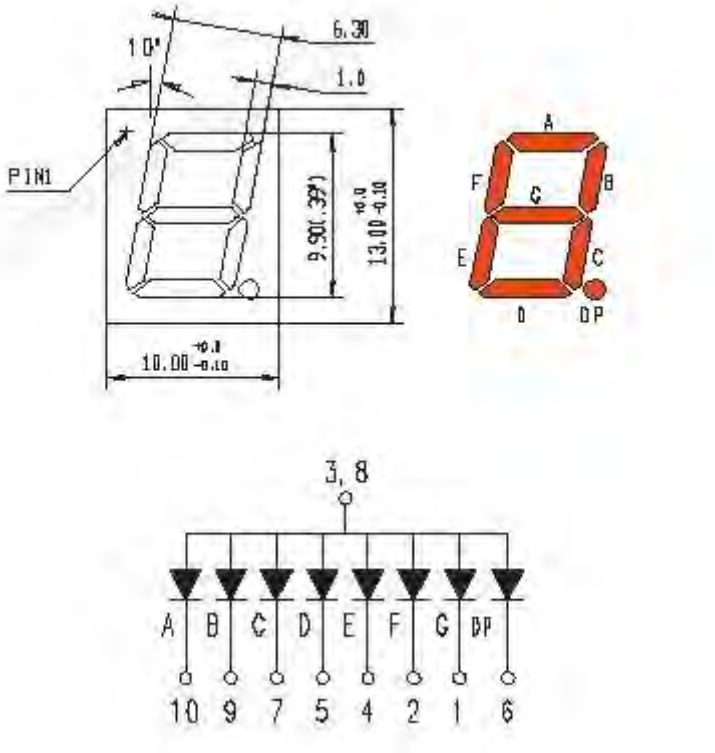
**2. 实验程序**

```
always @(posedge clk)//分频进程
begin
    if(count==50000)
       begin
       count=0;
       clk_1k=~clk_1k;
       end
    else
       count=count+1;
end
always @(posedge clk_1k)//数码管扫描进程
    begin
          case(wei_count)
               3'b000: begin
                          wei=8'b11111110; duan=8'b1100_0000;wei_count=wei_count+1'b1;end
               3'b001: begin
                          wei=8'b11111101; duan=8'b1111_1001;wei_count=wei_count+1'b1;end
               3'b010: begin
                          wei=8'b11111011; duan=8'b1010_0100;wei_count=wei_count+1'b1;end
               3'b011: begin
                          wei=8'b11110111; duan=8'b1011_0000;wei_count=wei_count+1'b1;end
               3'b100: begin
                          wei=8'b11101111; duan=8'b1001_1001;wei_count=wei_count+1'b1;end
               3'b101: begin
                           wei=8'b11011111; duan=8'b1001_0010;wei_count=wei_count+1'b1;end
               3'b110: begin
                           wei=8'b10111111; duan=8'b1000_0011;wei_count=wei_count+1'b1;end
               3'b111: begin
                           wei=8'b01111111; duan=8'b1111_1000;wei_count=wei_count+1'b1;end
            endcase
          end
```

## 9.4 秒表数码管显示

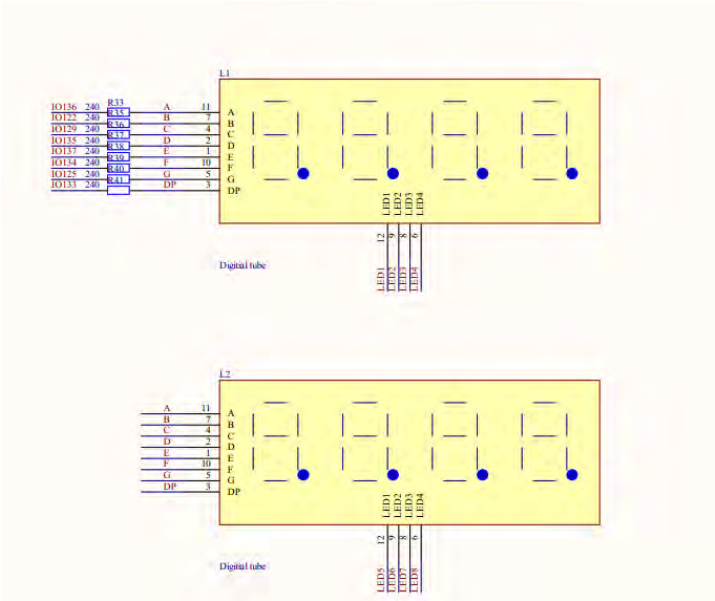
# 1、实验硬件说明

数码管就是由若干个 LED 灯组合而成的器件.如八段数码管就是由八个 LED 灯相互并联组成的,所以八个 LED 灯会有一个公共端点,如果是这个公共端点都是阳极,称为共阳数码管,如果公共端点是阴极,称为共阴数码管。




共阳八段数码管

实验的原理图：



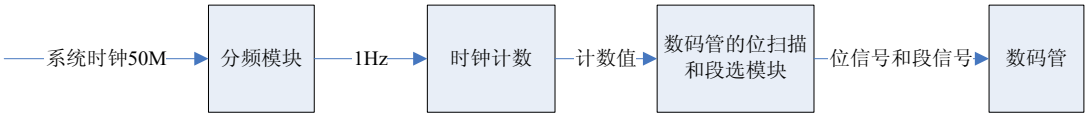

# 2、实验设计原理

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。

所谓动态扫描显示即轮流向各位数码管送出字形码和相应的位选，只要扫描显示速度够快，利用发光管的余辉和人眼视觉暂留作用，使人的感觉好像各位数码管同时都在显示。动态显示的亮度比静态显示要差一些，所以在选择限流电阻时应略小于静态显示电路中的。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，每个数码管显示时间不能太长也不能太短，时间太长会影响刷新率，导致总体显示呈现闪烁的现象，时间太短发光二极管的电流导通时间也就短，会影响总体的显示亮度。一般控制在 1ms 左右最佳。

秒表的设计原理：首先通过分频器产生一个频率为 1Hz 的时钟作为计数源，设置两个寄存器分别用来存储计数值，一个为寄存个位数，一个寄存十位数。同时产生一个数码管位选时钟，用来对数码管的位选进行扫描，这里仅需要两位数码管。

设计的原理框图：




## 3、实验程序

```
always @(posedge clk)//1HZ 时钟进程
  begin
    if(count==25000000)
      begin
          clk_1hz=~clk_1hz;
          count=0;
      end
    else
        count=count+1'b1;
  end
always @(posedge clk_1hz)//秒表功能进程
  begin
    if(ge==4'b1001)
      begin
          ge=4'b0000;
          if(shi==3'b101)
            shi=3'b000;
          else
            shi=shi+1'b1;
      end
    else
      ge=ge+1'b1;
  end
always @(posedge clk)//数码管扫描时钟产生进程
begin
```

```
    if(count2==50000)
        begin
        count2=0;
        clk_scan=~clk_scan;
        end
    else
        count2=count2+1;
end
always @(posedge clk_scan)
  begin
        select=select+1'b1;
  end
always @(ge or shi or select)
  begin
    if(select==1'b1)
    begin
    wei=2'b10;//秒表个位数显示
    case(ge)
          4'b0000:begin duan=8'b1100_0000;end
          4'b0001:begin duan=8'b1111_1001;end
          4'b0010:begin duan=8'b1010_0100;end
          4'b0011:begin duan=8'b1011_0000;end
          4'b0100:begin duan=8'b1001_1001;end
          4'b0101:begin duan=8'b1001_0010;end
          4'b0110:begin duan=8'b1000_0011;end
          4'b0111:begin duan=8'b1111_1000;end
          4'b1000:begin duan=8'b1000_0000;end
          4'b1001:begin duan=8'b1001_1000;end
          default duan=8'bx;
    endcase
    end
    else
    begin
    wei=2'b01;//秒表十位数显示
    case(shi)
          3'b000:duan=8'b1100_0000;
          3'b001:duan=8'b1111_1001;
          3'b010:duan=8'b1010_0100;
          3'b011:duan=8'b1011_0000;
          3'b100:duan=8'b1001_1001;
          3'b101:duan=8'b1001_0010;
          3'b110:duan=8'b1000_0011;
          default duan=8'bx;
```

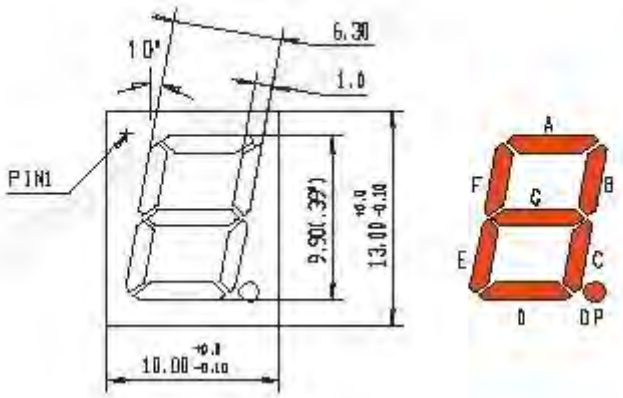
```
    endcase
    end
end
```

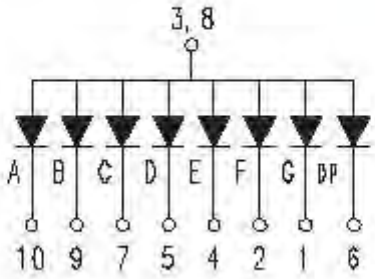
## 4、实验现象

# 9.5 时钟数码管显示

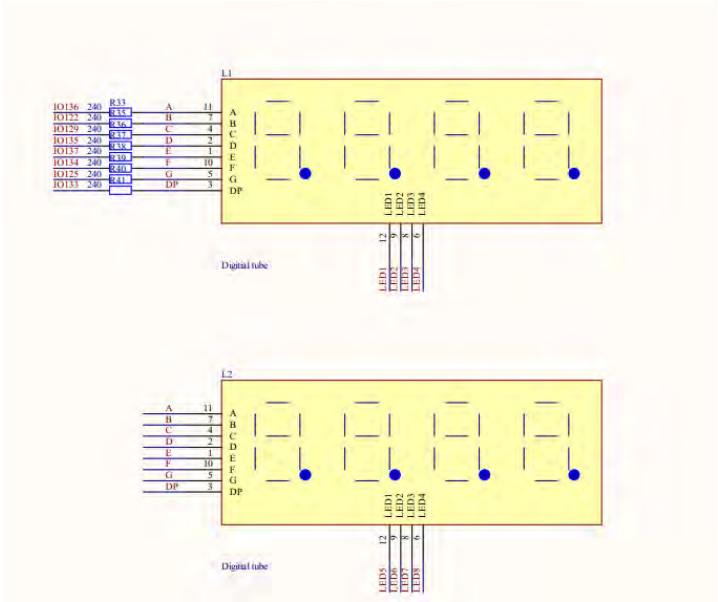
## 实验硬件说明

数码管就是由若干个 LED 灯组合而成的器件.如八段数码管就是由八个 LED 灯相互并联组成的,所以八个 LED 灯会有一个公共端点,如果是这个公共端点都是阳极,称为共阳数码管,如果公共端点是阴极,称为共阴数码管。




共阳八段数码管
实验的原理图：




## 1. 实验设计原理

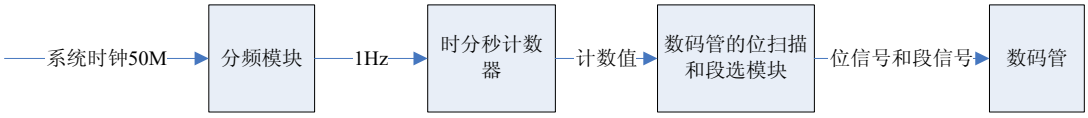
由于 LED 静态显示需要占用较多的 I/O 口，且功耗较大，因此在大多数场合通常不采用静态显示，而采用动态扫描的方法来控制 LED 数码管的显示。动态显示的特点是将 8 位数码管的段选线并联在一起，由位选线控制是哪一位数码管有效。点亮数码管采用动态扫描显示。

所谓动态扫描显示即轮流向各位数码管送出字形码和相应的位选，只要扫描显示速度够

快，利用发光管的余辉和人眼视觉暂留作用，使人的感觉好像各位数码管同时都在显示。动态显示的亮度比静态显示要差一些，所以在选择限流电阻时应略小于静态显示电路中的。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，每个数码管显示时间不能太长也不能太短，时间太长会影响刷新率，导致总体显示呈现闪烁的现象，时间太短发光二极管的电流导通时间也就短，会影响总体的显示亮度。一般控制在 1ms 左右最佳。

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，设置六个寄存器分别用来存储计数值，时分秒各用两个寄存器，一个为寄存个位数，另一个寄存十位数。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。产生一个数码管位选时钟，用来对数码管的位选进行扫描，这里需要六位数码管。

设计的原理框图：




## 2. 实验程序

```
always @(posedge clk)//1HZ 时钟进程
  begin
    if(count==25000000)
      begin
          clk_1hz<=~clk_1hz;
          count<=0;
      end
    else
        count<=count+1'b1;
  end
always @(posedge clk_1hz)//秒分时各位累加功能进程
  begin
    if(miao_ge==4'b1001)
    begin
                miao_ge<=4'b0000;
                if(miao_shi==3'b101)
                begin
                        miao_shi<=3'b000;
                        if(fen_ge==4'b1001)
                        begin
                                fen_ge<=4'b0000;
                                if(fen_shi==3'b101)
                                begin
                                    fen_shi<=3'b000;
                                    if(shi_ge==2'b11)
                                    begin
                                            shi_ge<=2'b00;
```

```
                                        if(shi_shi==2'b10)
                                            shi_shi<=2'b00;
                                        else
                                            shi_shi<=shi_shi+1'b1;
                                    end
                                    else
                                    shi_ge<=shi_ge+1'b1;
                                end
                                else
                                fen_shi<=fen_shi+1'b1;
                            end
                            else
                            fen_ge<=fen_ge+1'b1;
                        end
                        else
                        miao_shi<=miao_shi+1'b1;
      end
      else
         miao_ge<=miao_ge+1'b1;
   end
always @(posedge clk)//数码管扫描时钟产生进程
begin
    if(count2==10000)
       begin
       count2<=0;
       clk_scan<=~clk_scan;
       end
    else
       count2<=count2+1;
end
always @(posedge clk_scan)
 begin
       if(select==3'b110)
       select<=3'b000;
       else
       select<=select+1'b1;

 end
always @(miao_ge or miao_shi or fen_ge or fen_shi or shi_ge or shi_shi or select)//敏感信号列
表
 begin
    if(select==3'b001)
    begin
```

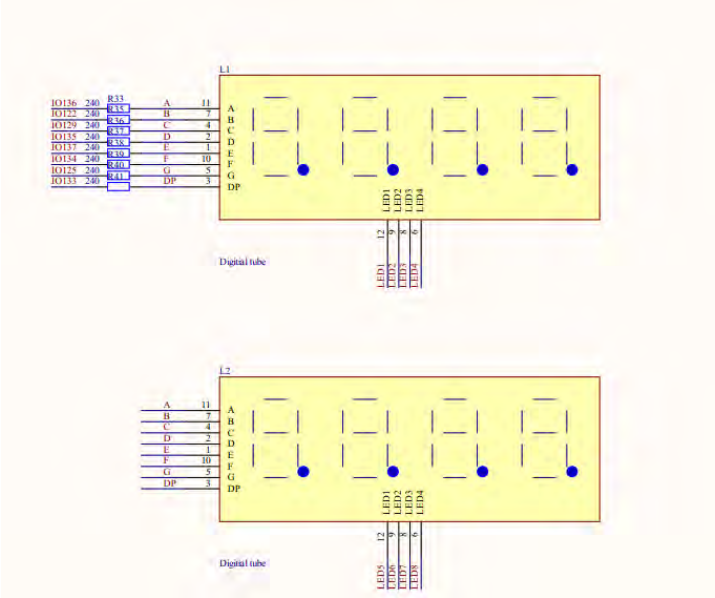
```
wei<=6'b111110;//秒个位数显示
case(miao_ge)
    4'b0000:begin duan<=8'b1100_0000;end
    4'b0001:begin duan<=8'b1111_1001;end
    4'b0010:begin duan<=8'b1010_0100;end
    4'b0011:begin duan<=8'b1011_0000;end
    4'b0100:begin duan<=8'b1001_1001;end
    4'b0101:begin duan<=8'b1001_0010;end
    4'b0110:begin duan<=8'b1000_0011;end
    4'b0111:begin duan<=8'b1111_1000;end
    4'b1000:begin duan<=8'b1000_0000;end
    4'b1001:begin duan<=8'b1001_1000;end
    default duan<=8'bx;
endcase
end
else if(select==3'b010)
begin
wei<=6'b111101;//秒十位数显示
case(miao_shi)
    3'b000:duan<=8'b1100_0000;
    3'b001:duan<=8'b1111_1001;
    3'b010:duan<=8'b1010_0100;
    3'b011:duan<=8'b1011_0000;
    3'b100:duan<=8'b1001_1001;
    3'b101:duan<=8'b1001_0010;
    3'b110:duan<=8'b1000_0011;
    default duan<=8'bx;
endcase
end
else if(select==3'b011)
begin
wei<=6'b111011;//分钟个位数显示
case(fen_ge)
    4'b0000:begin duan<=8'b1100_0000;end
    4'b0001:begin duan<=8'b1111_1001;end
    4'b0010:begin duan<=8'b1010_0100;end
    4'b0011:begin duan<=8'b1011_0000;end
    4'b0100:begin duan<=8'b1001_1001;end
    4'b0101:begin duan<=8'b1001_0010;end
    4'b0110:begin duan<=8'b1000_0011;end
    4'b0111:begin duan<=8'b1111_1000;end
    4'b1000:begin duan<=8'b1000_0000;end
    4'b1001:begin duan<=8'b1001_1000;end
```

```
            default duan<=8'bx;
        endcase
        end
        else if(select==3'b100)
        begin
        wei<=6'b110111;//分钟十位数显示
        case(fen_shi)
            3'b000:duan<=8'b1100_0000;
            3'b001:duan<=8'b1111_1001;
            3'b010:duan<=8'b1010_0100;
            3'b011:duan<=8'b1011_0000;
            3'b100:duan<=8'b1001_1001;
            3'b101:duan<=8'b1001_0010;
            3'b110:duan<=8'b1000_0011;
            default duan<=8'bx;
        endcase
        end
        else if(select==3'b101)
        begin
        wei<=6'b101111;//时钟个位数显示
        case(shi_ge)
            4'b0000:begin duan<=8'b1100_0000;end
            4'b0001:begin duan<=8'b1111_1001;end
            4'b0010:begin duan<=8'b1010_0100;end
            4'b0011:begin duan<=8'b1011_0000;end
            default duan<=8'bx;
        endcase
        end
        else
        begin
        wei<=6'b011111;//时钟十位数显示
        case(shi_shi)
            3'b000:duan<=8'b1100_0000;
            3'b001:duan<=8'b1111_1001;
            3'b010:duan<=8'b1010_0100;
            default duan<=8'bx;
        endcase
        end
end
```

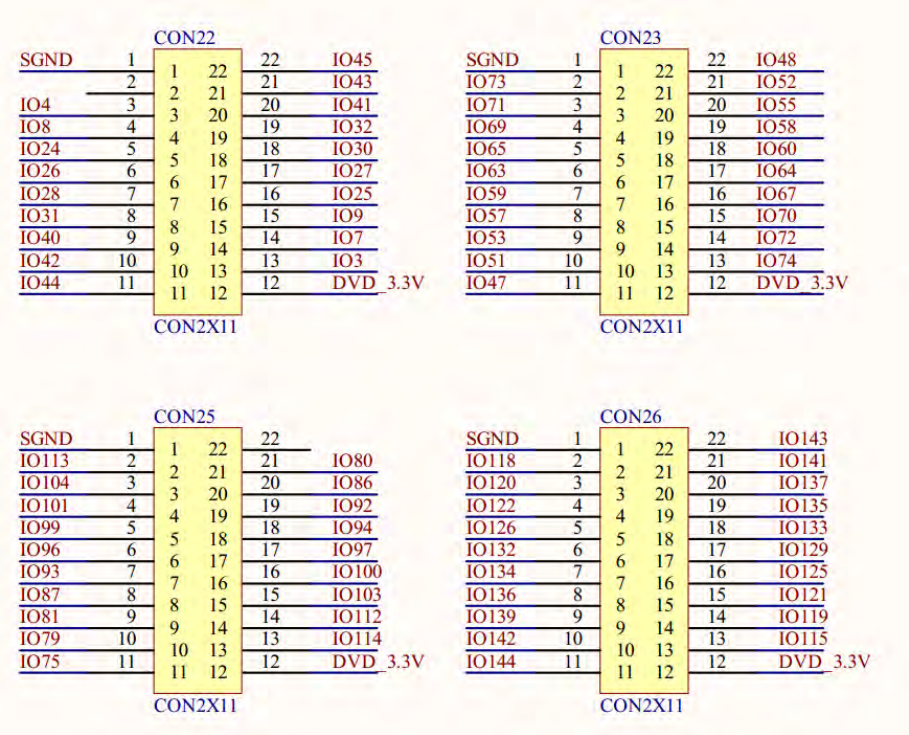
# 9.6 测频计的设计

## 实验硬件说明

实验的原理图：




**外扩引脚原理图：**选择两个 IO 口，一个作为时钟输出（测试源），另一个作为时钟输入



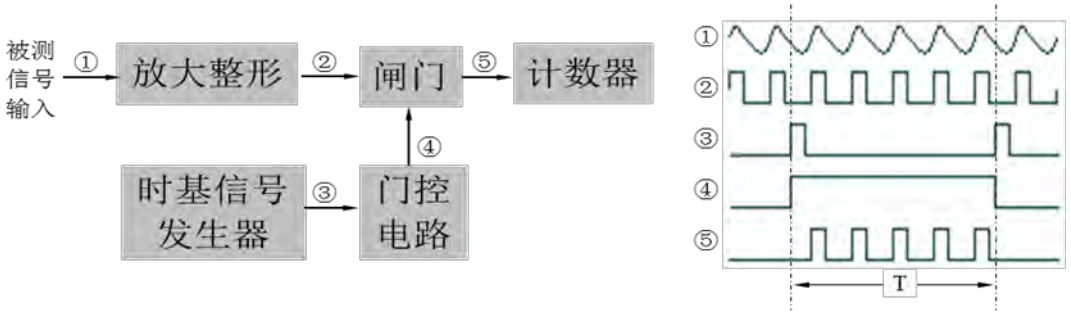

### 1. 实验设计原理

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。

**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，原理图如下图所示：

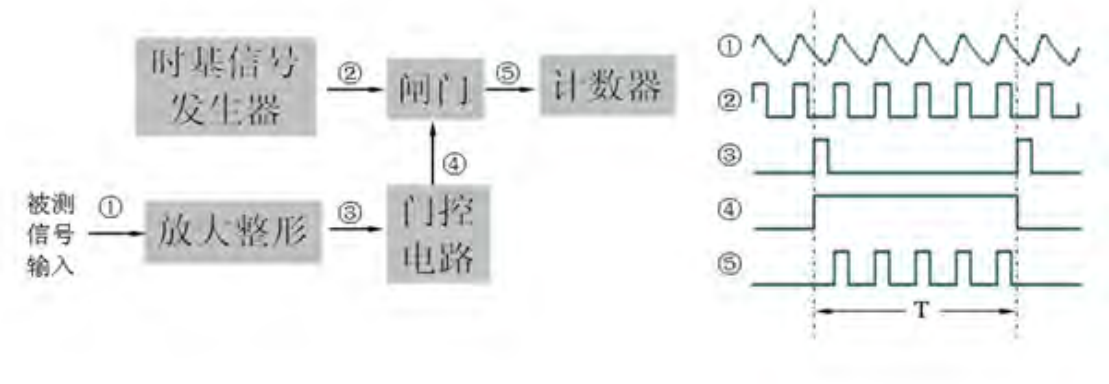



测频方法原理图

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=0.1S，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。

**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：

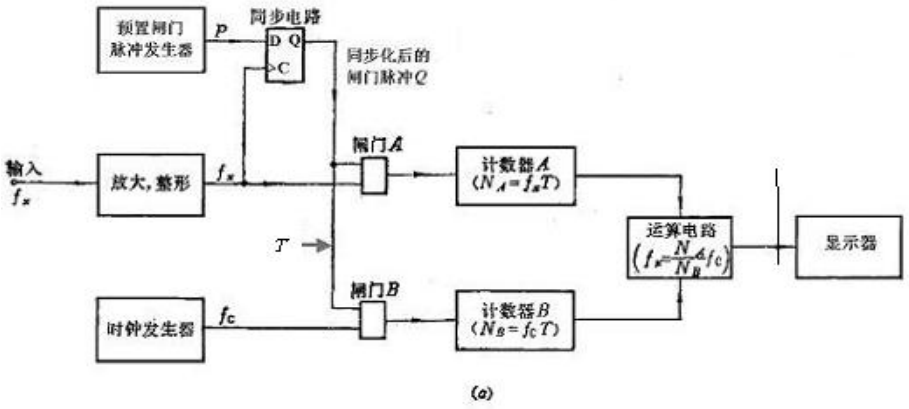



计数器测周的基本原理刚好与测频相反，即由被测信号控制主门开门，而用时标脉冲进行计数，所以实质上也是一种比较测量方法。

**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。

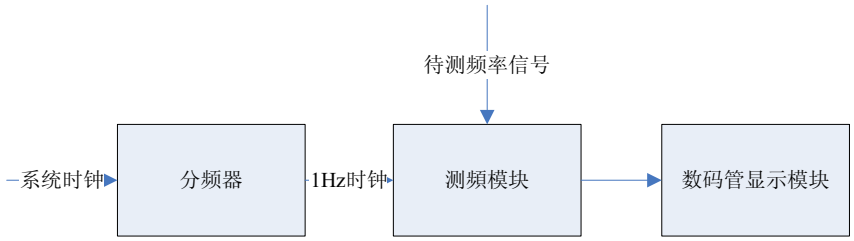
在等精度测量法中，相对误差与被测信号本身的频率特性无关，即对整个测量域而言，测量精度相等，因而称之为“等精度测量”。标准信号的计数值越大则测量相对误差越小，即提高门限时间 τ 和标准信号频率 $f_c$ 可以提高测量精度。在精度不变的情况下，提高标准信号频率可以缩短门限时间，提高测量速度。原理图如下：




等精度测量的原理图

这里选择测频方法进行设计。

设计的原理框图：




2. 实验程序

```
/**********************产生测试时钟**************************************/
always @(posedge sysclk)
  begin
    if(counter==15'b110_0001_1010_1000)
      begin
       test_clk<=~test_clk;//500HZ
       counter<=15'b0;
      end
    else
       counter<=counter+1'b1;
  end
/**********************产生 1HZ 的时钟**************************************/
always @(posedge sysclk)
  begin
    if(clk_counter==25'd24999999)
      begin
       clk_div<=~clk_div;
       clk_counter<=25'b0;
      end
    else
       clk_counter<=clk_counter+1'b1;
  end
/**********************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'b
0000|

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 seg_wei<=8'b1111_1110;data<=count1;end
     3'b001:begin seg_wei<=8'b1111_1101;data<=count2;end
     3'b010:begin seg_wei<=8'b1111_1011;data<=count3;end
     3'b011:begin seg_wei<=8'b1111_0111;data<=count4;end
     3'b100:begin seg_wei<=8'b1110_1111;data<=count5;end
     3'b101:begin seg_wei<=8'b1101_1111;data<=count6;end
     3'b110:begin seg_wei<=8'b1011_1111;data<=count7;end
     3'b111:begin seg_wei<=8'b0111_1111;data<=count8;end
      default:begin seg_wei<=8'bx;data<=4'bx;end
    endcase

  case(data[3:0])//数码管显示
   4'b0000:begin seg_duan<=8'b1100_0000;end//0
   4'b0001:begin seg_duan<=8'b1111_1001;end//1
   4'b0010:begin seg_duan<=8'b1010_0100;end//2
   4'b0011:begin seg_duan<=8'b1011_0000;end//3
   4'b0100:begin seg_duan<=8'b1001_1001;end//4
   4'b0101:begin seg_duan<=8'b1001_0010;end//5
```

```
    4'b0110:begin seg_duan<=8'b1000_0010;end//6
    4'b0111:begin seg_duan<=8'b1111_1000;end//7
    4'b1000:begin seg_duan<=8'b1000_0000;end//8
    4'b1001:begin seg_duan<=8'b1001_0000;end//9
    default:seg_duan<=8'bx;
  endcase
  end
```

## 9.7 蜂鸣器唱歌

# 实验硬件说明

**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。

**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郡朝上放置时，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 Rx1 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。

## 1. 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 |

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！

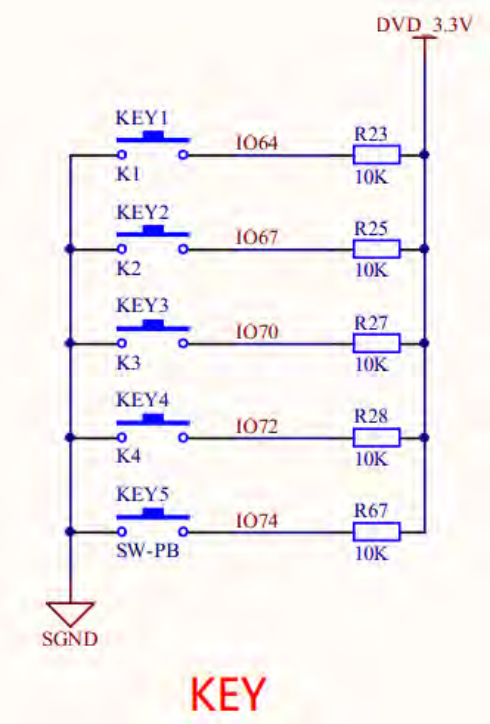
## 2. 实验程序

由于程序量较大，请参考实验工程

# 9.8 按键消抖

1. 实验硬件说明

按键原理图：



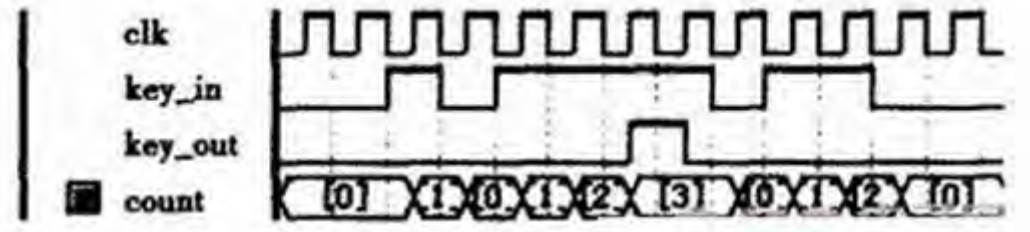

2. 实验设计原理

计数器型消抖电路：

计数器型消抖电路(一)

计数器型消抖电路(一)是设置一个模值为(N+1)的控制计数器，clk 在上升沿时，如果按键开关 key_in='1'，计数器加 1，key_in='0'时，计数器清零。当计数器值为 2 时，key_out 输出才为 1，其他值为 0 时。计数器值为 N 时处于保持状态。因此按键 key_in 持续时间大于 N 个 clk 时钟周期时，计数器输出一个单脉冲，否则没有脉冲输出。如果按键开关抖动产生的毛刺宽度小于 N 个时钟周期，因而毛刺作用不可能使计数器有输出，防抖动目的得以实现。clk 的时钟周期与 N 的值可以根据按键抖动时间由设计者自行设定。

下图为 N 为 3 的波形仿真图，当按键持续时间大于 3 个时钟周期，计数器输出一个单脉冲，其宽度为 1 个时钟周期，小于 3 个时钟周期的窄脉冲用作模拟抖动干扰，从图 1 可以看出，抖动不能干扰正常的单脉冲输出。

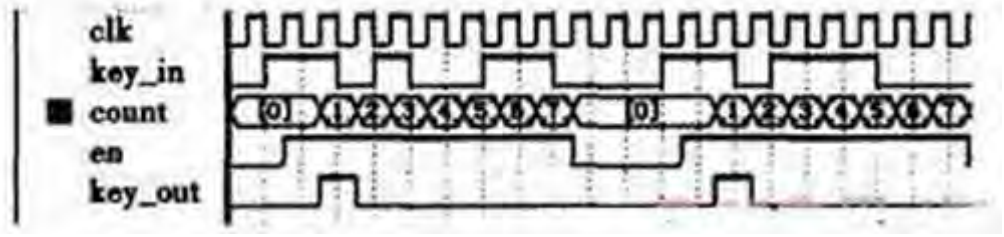



该方法的特点是能很好消除按键抖动产生的窄脉冲，还可以滤去干扰、噪音等其他尖峰波，但遇到脉宽大于 N 个 Tclk 时钟周期的干扰、噪音等时会有输出从而产生误操作，而对于按键操作要求按键时间必须大于 N 个 Tclk 时钟周期，否则按键操作也没有输出。

**计数器型消抖电路(二)**

计数器型消抖电路(二)是控制计数器工作一个循环周期(N+1 个状态)，且仅在计数器为 0 时输出为“1”。电路设计了连锁控制设施。在计数器处于状态 0 时，此时若有按键操作，则计数器进入状态 1，同时输出单脉冲(其宽度等于时钟周期)。计数器处于其他状态，都没有单脉冲输出。计数器处于状态 N 时，控制 en=’0’，导致计数器退出状态 N，进入状态 0。计数器能否保持状态 0，取决于人工按键操作，若按键 key_in=’1’，控制 en=’1’(计数器能正常工作)，key_in=’0’，计数器状态保持。显见计数器处于状态 0，人工不按键，则计数器保持状态 0。

下图是 N 为 7 的波形仿真图。在计数器状态为 0 时，key_in 有按键操作，计数器开始连续计数直到计数器状态为 0;计数器状态为 1-7 时,key_in 任何操作对计数器工作无影响，计数器在状态为 1 时，输出一个单脉冲，脉冲宽度为 1 个时钟周期。

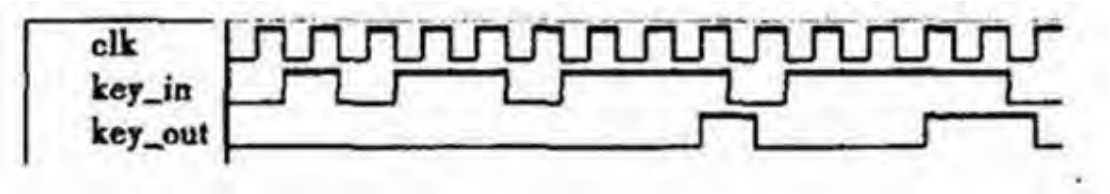



该设计方法的特点是能很好消除按键抖动产生的连续脉冲，对按键时间没有要求，缺点是在计数器状态为 0 时，遇到干扰、噪音等时会有输出，从而产生误操作。

**状态机型消抖电路**

状态机型消抖电路采用有限状态机的设计方法来描述与实现，状态机有 S0，S1，S2 三种状态,在 S0 状态下 key_out 输出为低电平,并以 clk 时钟信号的频率采样按键输入信号，如果 key_in=‘0’,则保持在 S0 状态,并继续采样按键输入信号的状态,如果 key_in=‘1’，则转入 S1 状态；在 S1 状态下 key_out 输出仍为低电平，继续采样按键输入信号的状态，如果 key_in=‘1’，则转入 S2 状态，如果 key_in=‘0’ 则转入 S0 状态；在 S2 状态下继续采样按键输入信号的状态，如果 key_in=‘1’，则保持在 S2 状态，key_out 输出正脉冲，如果 key_in=‘0’，则转入 S0 状态，key_out 输出低电平。

下图为状态机型消抖电路波形仿真图




**3. 实验程序**

**计数器型消抖电路(一)程序：**

```
always @(posedge clk)
  begin
    if(div_counter==15'd25000)
      begin
        clk_div<=~clk_div;
        div_counter<=15'd0;
      end
    else
```

```
        div_counter<=div_counter+15'd1;
  end

always @(posedge clk_div )
  begin
    if(key==1'b0)//判断按键是否按下
      begin
        if(clk_counter==3'b111)//计数到 7
          begin
            clk_counter<=3'b000;
            key_out<=1'b1;
          end
        else
              begin
                  clk_counter<=clk_counter+1'b1;
                if(clk_counter==3'b110)//判断是否计到 6
                  key_out<=1'b0;
                else
                  key_out<=1'b1;
              end
      end
    else
      clk_counter<=3'b000;
  end
always @(negedge key_out)//对消抖后的按键信号进行判断
  begin
   if(!key_out)//判断按键是否按下
      begin
        led<=~led;//led 灯状态反转
      end
   else
        led<=led;
  end
```

**状态机型消抖电路程序：**

```
parameter s0=2'b00,s1=2'b01,s2=2'b10,s3=2'b11;
reg [1:0] state;
always @(posedge clk)
 begin
     case (state)
       s0:
        begin
          key_out<=1'b1;
          if(key==1'b0)
```

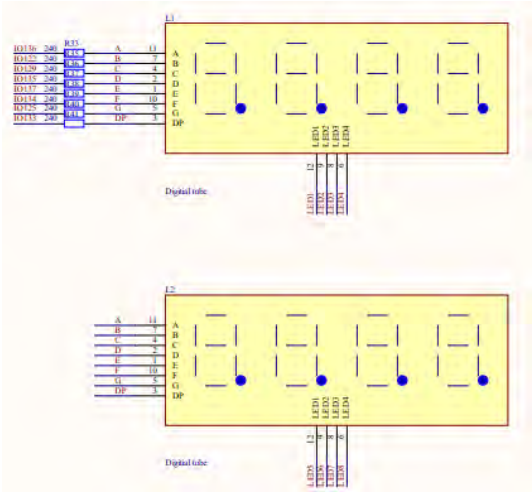
```
                state<=s1;
            else
                state<=s0;
          end
        s1:
          begin
            if(key==1'b0)
              state<=s2;
            else
              state<=s0;
            end
        s2:
          begin
            if(key==1'b0)
              state<=s3;
            else
              state<=s0;
            end
        s3:
            begin
              if(key==1'b0)
                begin
                key_out<=1'b0;
                state<=s3;
                end
              else
                begin
                key_out<=1'b1;
                state<=s0;
                end
            end
        default:
              state<=s0;
      endcase
  end

always @(negedge key_out)
  begin
    if(!key_out)//判断按键是否按下
        begin
          if(led_temp==8'b1111_1111)
            led_temp<=8'b0;
          else
```

```
            led_temp<=led_temp+1'b1;
        end
    end
assign led=~led_temp;
```

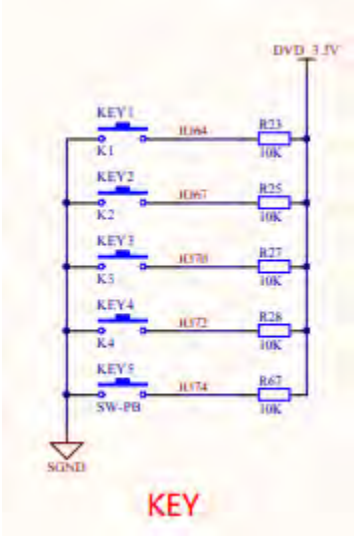
# 9.9 按键计数器

## 实验硬件说明

**数码管原理图：**



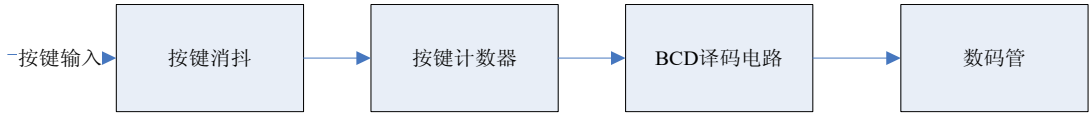
**按键原理图：**




**1. 实验设计原理**

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，在该工程中采用状态机消抖对按键进行消抖，消抖后对按键进行计数，最后将计数结构显示在数码管上。

设计原理流程图如下：




**2. 实验程序**

```
/*****************************按键消抖*************************************/
always @(posedge clk)
```

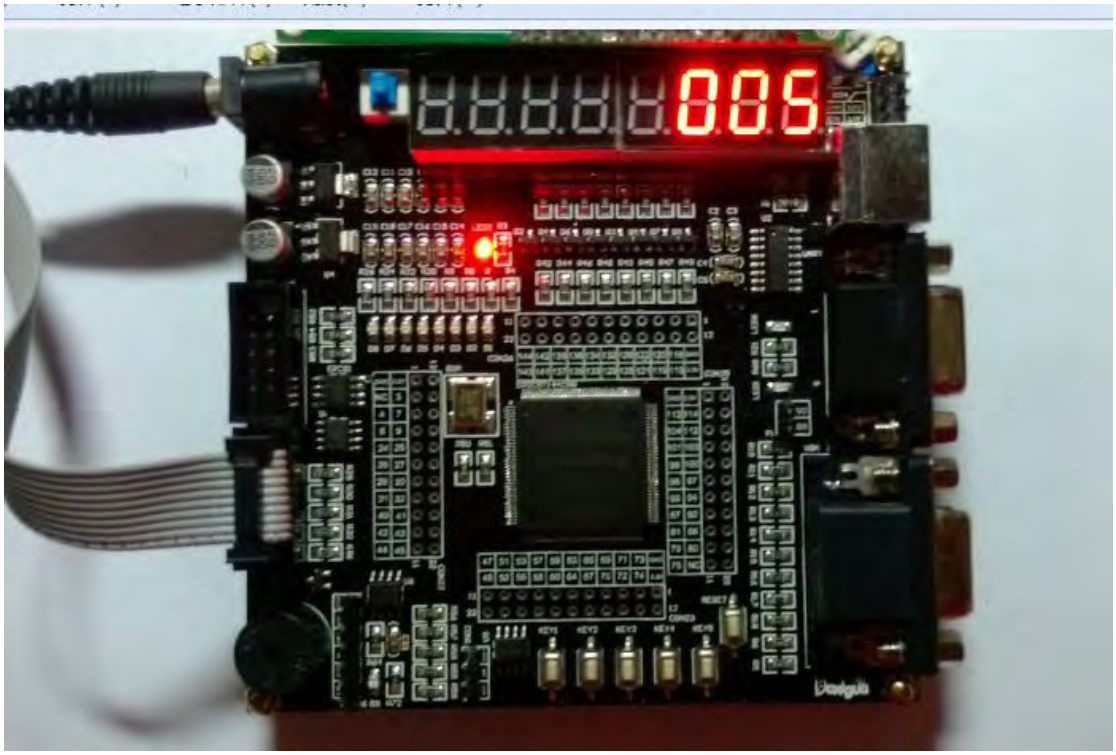
```
begin
   case (state)
    s0:
     begin
      key_out<=1'b1;
      if(key==1'b0)
         state<=s1;
      else
         state<=s0;
     end
    s1:
     begin
      if(key==1'b0)
        state<=s2;
      else
        state<=s0;
      end
    s2:
     begin
      if(key==1'b0)
        state<=s3;
      else
        state<=s0;
      end
    s3:
      begin
       if(key==1'b0)
        begin
        key_out<=1'b0;
        state<=s3;
        end
       else
        begin
        key_out<=1'b1;
        state<=s0;
        end
      end
    default:
        state<=s0;
   endcase
end
/*****************************按键计数*************************************/
always @(negedge key_out or negedge reset)
```

```
 begin
  if(!reset)
    begin
      key_counter_ge <=4'd0;
      key_counter_shi<=4'd0;
      key_counter_bai<=4'd0;
    end
  else
   begin
   if(!key_out)//判断按键是否按下
     begin
        key_counter_ge<=key_counter_ge+1'b1;//按键计数个位加一
      if(key_counter_ge==4'd9)
        begin
         key_counter_ge<=4'd0;//个位清零
         key_counter_shi<=key_counter_shi+1'b1;//按键计数十位加一
         if(key_counter_shi==4'd9)
          begin
           key_counter_shi<=4'd0;//十位清零
           key_counter_bai<=key_counter_bai+1'b1;//按键计数百位加一
           if(key_counter_bai==4'd9)
              key_counter_bai<=4'd0;//百位清零
          end
        end
     end
    end
 end
/*****************************数码管扫描时钟*********************************/
always @(posedge clk)//产生位选时钟
begin
   if(div_count==10'd1000)
     begin
     div_count<=10'd0;
     clk_scan<=~clk_scan;
     end
   else
     div_count<=div_count+10'd1;
end
always @(posedge clk_scan)//产生位选信号
 begin
     if(wei_select==2'b11)
      wei_select<=2'b00;
     else
```

```
      wei_select<=wei_select+1'b1;

 end
/*****************************数码管显示*********************************/
always @(key_counter_ge or key_counter_shi or key_counter_bai or wei_select)
 begin
   if(wei_select==2'b00)
     begin
      seg_wei<=3'b110;
      case(key_counter_ge)
       4'b0000:begin seg_duan<=8'b1100_0000;end//0
       4'b0001:begin seg_duan<=8'b1111_1001;end//1
       4'b0010:begin seg_duan<=8'b1010_0100;end//2
       4'b0011:begin seg_duan<=8'b1011_0000;end//3
       4'b0100:begin seg_duan<=8'b1001_1001;end//4
       4'b0101:begin seg_duan<=8'b1001_0010;end//5
       4'b0110:begin seg_duan<=8'b1000_0010;end//6
       4'b0111:begin seg_duan<=8'b1111_1000;end//7
       4'b1000:begin seg_duan<=8'b1000_0000;end//8
       4'b1001:begin seg_duan<=8'b1001_0000;end//9
        default seg_duan<=8'bx;
      endcase
     end
   else
     begin
     if(wei_select==2'b01)
         begin
          seg_wei<=3'b101;
          case(key_counter_shi)
           4'b0000:begin seg_duan<=8'b1100_0000;end//0
           4'b0001:begin seg_duan<=8'b1111_1001;end//1
           4'b0010:begin seg_duan<=8'b1010_0100;end//2
           4'b0011:begin seg_duan<=8'b1011_0000;end//3
           4'b0100:begin seg_duan<=8'b1001_1001;end//4
           4'b0101:begin seg_duan<=8'b1001_0010;end//5
           4'b0110:begin seg_duan<=8'b1000_0010;end//6
           4'b0111:begin seg_duan<=8'b1111_1000;end//7
           4'b1000:begin seg_duan<=8'b1000_0000;end//8
           4'b1001:begin seg_duan<=8'b1001_0000;end//9
            default seg_duan<=8'bx;
          endcase
         end
        else
```

```
        begin
          seg_wei<=3'b011;
         case(key_counter_bai)
          4'b0000:begin seg_duan<=8'b1100_0000;end//0
          4'b0001:begin seg_duan<=8'b1111_1001;end//1
          4'b0010:begin seg_duan<=8'b1010_0100;end//2
          4'b0011:begin seg_duan<=8'b1011_0000;end//3
          4'b0100:begin seg_duan<=8'b1001_1001;end//4
          4'b0101:begin seg_duan<=8'b1001_0010;end//5
          4'b0110:begin seg_duan<=8'b1000_0010;end//6
          4'b0111:begin seg_duan<=8'b1111_1000;end//7
          4'b1000:begin seg_duan<=8'b1000_0000;end//8
          4'b1001:begin seg_duan<=8'b1001_0000;end//9
          default seg_duan<=8'bx;
         endcase
       end
    end
end
```

3. 实验现象



# 9.10 串口通信

## 实验硬件说明

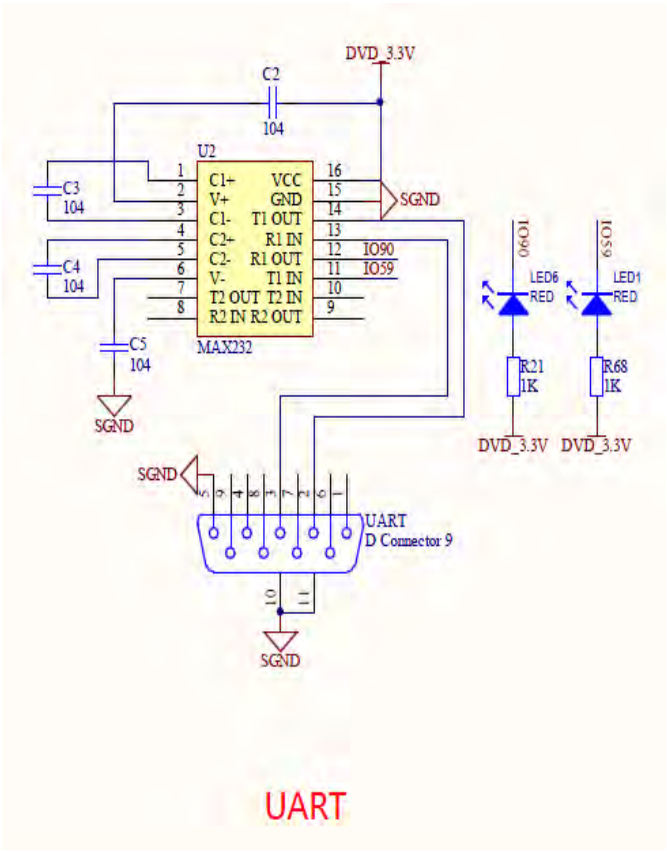
FPGA 和上位机电脑之间的通信采用 UART 串口通信方式，首先介绍下九针串口硬件的结构：

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.

接下来介绍串口的通信协议：
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。
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。
对于 RS-232 来说，是这样处理的：
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.

实验的原理图：




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。

**1. 实验设计原理（发送模块，接收模块，波特率发生模块的分别设计原理）**

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 650 分频可近似得到 9600bps*8 倍的时钟来控制发送和接收模块。

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。

**2. 实验程序**

波特率时钟产生：

```
parameter div_par=16'h145;//定义分频因子
    always@(posedge clk )
    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

    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'b1)
            clkbaud8x<=~clkbaud8x;
    end
```

发送模块:

```
    always@(posedge clkbaud8x or negedge rst)
    begin
        if(!rst) begin
            txd_reg<=1;
            trasstart<=0;
            txd_buf<=0;
            state_tras<=0;
            send_state<=0;
            key_entry2<=0;
         end
        else begin
            if(!key_entry2) begin
                if(key_entry1) begin
                    key_entry2<=1;
                    txd_buf<=8'd68; //"D"
                 end
             end
            else  begin
                case(state_tras)
                    4'b0000: begin  //发送起始位
                        if(!trasstart&&send_state<7)
                            trasstart<=1;
                        else if(send_state<7) begin
                            if(clkbaud_tras) begin
                                txd_reg<=0;
                                state_tras<=state_tras+1'b1;
                             end
                         end
                        else begin
                            key_entry2<=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
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<=8'd97;//"a"
            3'b001:
                txd_buf<=8'd120;//"x"
            3'b010:
                txd_buf<=8'd105;//"i"
            3'b011:
                txd_buf<=8'd103;//"g"
            3'b100:
                txd_buf<=8'd117;//"u"
            3'b101:
                txd_buf<=8'd97;//"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
```

接收模块：

```
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<=0;
        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
                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
             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
```

```
                if(clkbaud_rec) begin
                    state_rec<=0;
                    recstart<=0;
                end
            end
        end
    end
```

## 9.11 液晶 1602 显示

### 实验硬件说明

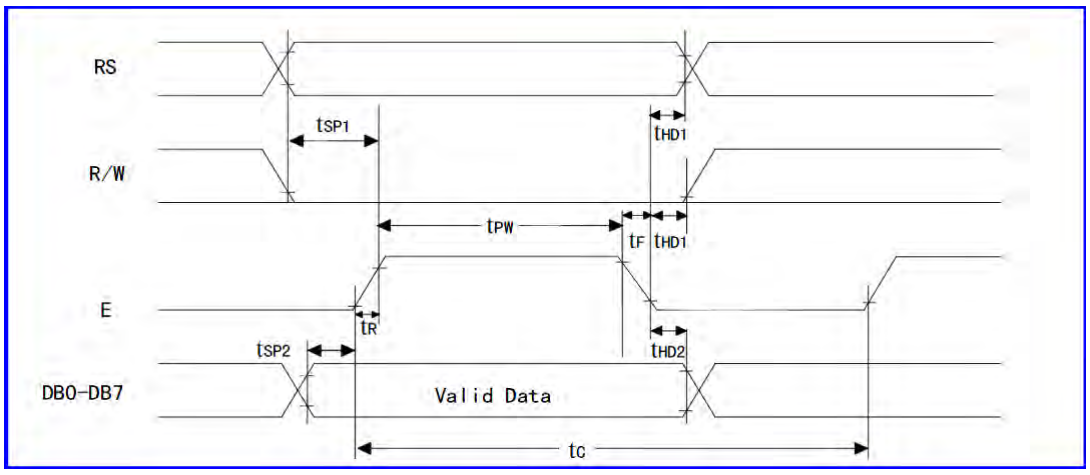
1602 常用与显示数字和字母，显示容量为 16*2 个字符，内部有 80 个字节的 RAM 缓冲区，当要向 1602 送数显示时，要先发送显示地址指针 80H+地址码。

内部地址码分布：

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F | 10 | …… | 27 |
| 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 4A | 4B | 4C | 4D | 4E | 4F | 50 | …… | 67 |

1602 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）。

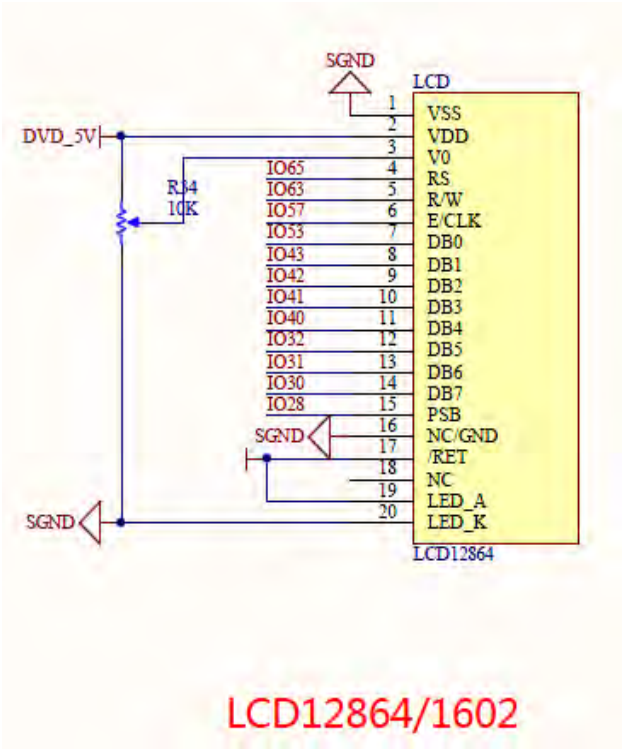
由于只要向 1602 写数据，因此给出写时序图和其时序图上的时序参数：



| 时序参数 | 符号 | 极限值 | | | 单位 | 测试条件 |
|---|---|---|---|---|---|---|
| | | 最小值 | 典型值 | 最大值 | | |
| E 信号周期 | $t_C$ | 400 | – | – | ns | 引脚 E |
| E 脉冲宽度 | $t_{PW}$ | 150 | – | – | ns | |
| E 上升沿/下降沿时间 | $t_R$, $t_F$ | – | – | 25 | ns | |
| 地址建立时间 | $t_{SP1}$ | 30 | – | – | ns | 引脚 E、RS、R/W |
| 地址保持时间 | $t_{HD1}$ | 10 | – | – | ns | |
| 数据建立时间(读操作) | $t_D$ | – | – | 100 | ns | 引脚 DB0~DB7 |
| 数据保持时间(读操作) | $t_{HD2}$ | 20 | – | – | ns | |
| 数据建立时间(写操作) | $t_{SP2}$ | 40 | – | – | ns | |
| 数据保持时间(写操作) | $t_{HD2}$ | 10 | – | – | ns | |

（其指令说明请详细查看 1602 数据手册）

实验的原理图：




该插口是 1602 和 12864 共用的插口（PSB-IO28 是背光源正极因此要输出高电平）

## 1. 实验设计原理

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。

## 2. 实验程序

使能信号的产生：

```
reg clk_e;
reg [15:0]count;
always @(posedge clk)
begin
   count=count+1'b1;
   if(count==16'hf000)
```

```
        begin
        clk_e=~clk_e;//作为使能端
        count=16'd0;
        end
    end

    reg D1,D2,D3,D4,D5;
    always @(posedge clk)//打入寄存器
    begin
    D1<=clk_e;
    D2<=D1;
    D3<=D2;
    D4<=D3;
    D5<=D4;
    end
    assign E=D5|temp;
```

数据显示：

```
    reg [1:0]jishu;
    reg [4:0]zhuangtai;//状态机状态
    reg temp;
    always @(posedge clk_e)
    begin
        case(zhuangtai)
            5'b00000:begin
                    temp<=1'b0;
                    RS<=1'b0;
                    Data<=8'h38;//显示模式设置
                    zhuangtai<=zhuangtai+1'b1;
                    end

            5'b00001:begin
                    RS<=1'b0;
                    Data<=8'h0c;//显示开及光标设置
                    zhuangtai<=zhuangtai+1'b1;
                    end

            5'b00010:begin
                    RS<=1'b0;
                    Data<=8'h06;//显示光标移动设置
                    zhuangtai<=zhuangtai+1'b1;
                    end
```

```
5'b00011:begin
        RS<=1'b0;
        Data<=8'h01;//显示清屏
        zhuangtai<=zhuangtai+1'b1;//前面几个状态是初始化的状态
        end

5'b00100:begin
        RS<=1'b0;
        Data<=address;//设置显示起始地址,第一行起始位置开始显示
        zhuangtai<=zhuangtai+1'b1;
        end

5'b00101:begin//显示"Daxigua!"
        RS<=1'b1;
        Data<="D";
        zhuangtai<=zhuangtai+1'b1;
        end

5'b00110:begin
        RS<=1'b1;
        Data<="a";
        zhuangtai<=zhuangtai+1'b1;
        end

5'b00111:begin
        RS<=1'b1;
        Data<="x";
        zhuangtai<=zhuangtai+1'b1;
        end

5'b01000:begin
        RS<=1'b1;
        Data<="i";
        zhuangtai<=zhuangtai+1'b1;
        end

5'b01001:begin
        RS<=1'b1;
        Data<="g";
        zhuangtai<=zhuangtai+1'b1;
        end

5'b01010:begin
```

```
            RS<=1'b1;
            Data<="u";
            zhuangtai<=zhuangtai+1'b1;
            end

    5'b01011:begin
            RS<=1'b1;
            Data<="a";
            zhuangtai<=zhuangtai+1'b1;
            end

    5'b01100:begin
            RS<=1'b1;
            Data<="!";
            zhuangtai<=zhuangtai+1'b1;
            end

    5'b01101:begin
            RS<=1'b0;
            Data<=address2;//切换到第二行显示
            zhuangtai<=zhuangtai+1'b1;
            end

    5'b01110:begin
            RS<=1'b1;
            Data<="Q";//后面显示大西瓜 QQ 号
            zhuangtai<=zhuangtai+1'b1;
            end

    5'b01111:begin
            RS<=1'b1;
            Data<="Q";
            zhuangtai<=zhuangtai+1'b1;
            end

    5'b10000:begin
            RS<=1'b1;
            Data<=":";
            zhuangtai<=zhuangtai+1'b1;
            end

    5'b10001:begin
            RS<=1'b1;
```

```
            Data<="2";
            zhuangtai<=zhuangtai+1'b1;
            end

        5'b10010:begin
            RS<=1'b1;
            Data<="0";
            zhuangtai<=zhuangtai+1'b1;
            end

        5'b10011:begin
            RS<=1'b1;
            Data<="3";
            zhuangtai<=zhuangtai+1'b1;
            end

        5'b10100:begin
            RS<=1'b1;
            Data<="7";
            zhuangtai<=zhuangtai+1'b1;
            end

        5'b10101:begin
            RS<=1'b1;
            Data<="6";
            zhuangtai<=zhuangtai+1'b1;
            end

        5'b10110:begin
            RS<=1'b1;
            Data<="9";
            zhuangtai<=zhuangtai+1'b1;
            end

        5'b10111:begin
            RS<=1'b1;
            Data<="7";
            zhuangtai<=zhuangtai+1'b1;
            end

        5'b11000:begin
            RS<=1'b1;
            Data<="8";
```

```
                zhuangtai<=zhuangtai+1'b1;
                end

            5'b11001:begin
                RS<=1'b1;
                Data<="8";
                zhuangtai<=zhuangtai+1'b1;
                end

            5'b11010:begin
                    if(jishu==2'b10)
                    begin
                    temp<=1'b1;//最后使能端一直拉高
                    zhuangtai<=4'b0111;
                    Data<=8'd0;
                    end

                    else
                    begin
                    zhuangtai<=4'b0000;
                    jishu=jishu+1'b1;
                    Data<=8'd0;
                    end
                end
            default zhuangtai<=4'b0000;
        endcase
end
```

3. 实验现象

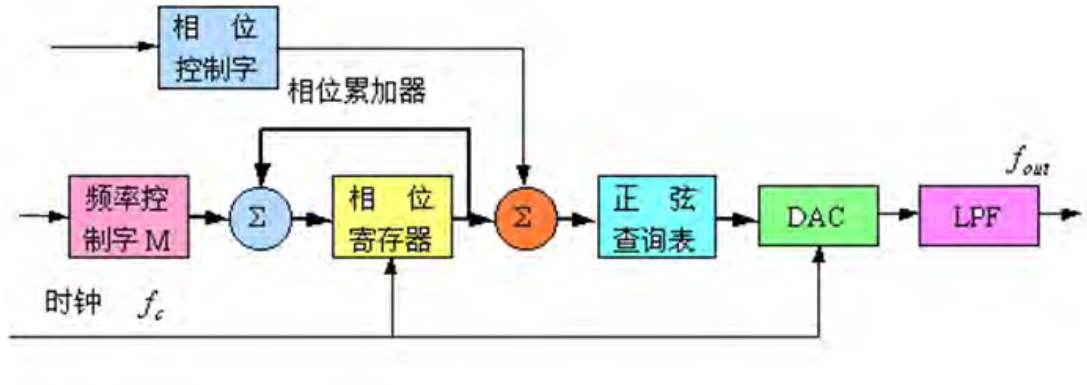
# 9.12DSS 与嵌入式逻辑分析仪的调用

## 9.12.1 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。

**DDS 原理框图**




主要构成:

内部:相位累加器,正弦查找表

外围:DAC,LPF(低通滤波器)

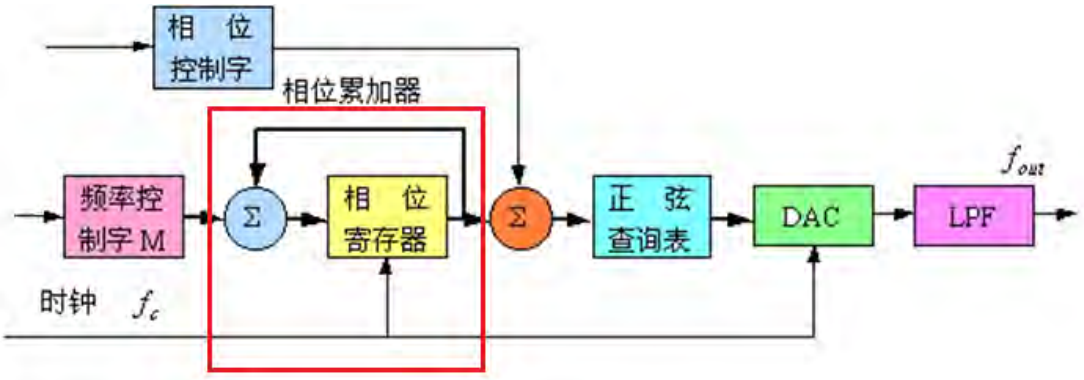
**工作过程**

1、将存于 ROM 中的数字波形,经 DAC,形成模拟量波形。

2、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。

3、DAC 输出的阶梯形波形,经低通*滤波*,成为*模拟波形*。

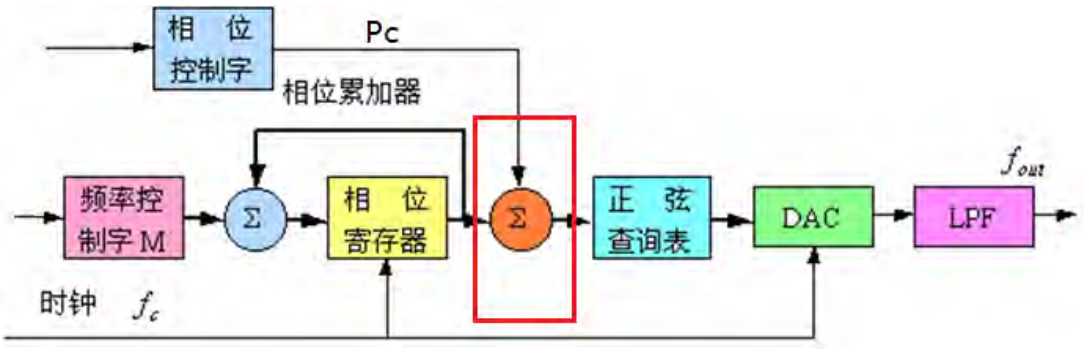
频率控制




DDS方程：$f_{out} = \frac{Mf_c}{2^N}$ Hz

$f_{out}$输出频率，$f_c$采样时钟，N相位累加器位宽，M频率控制字

相位控制

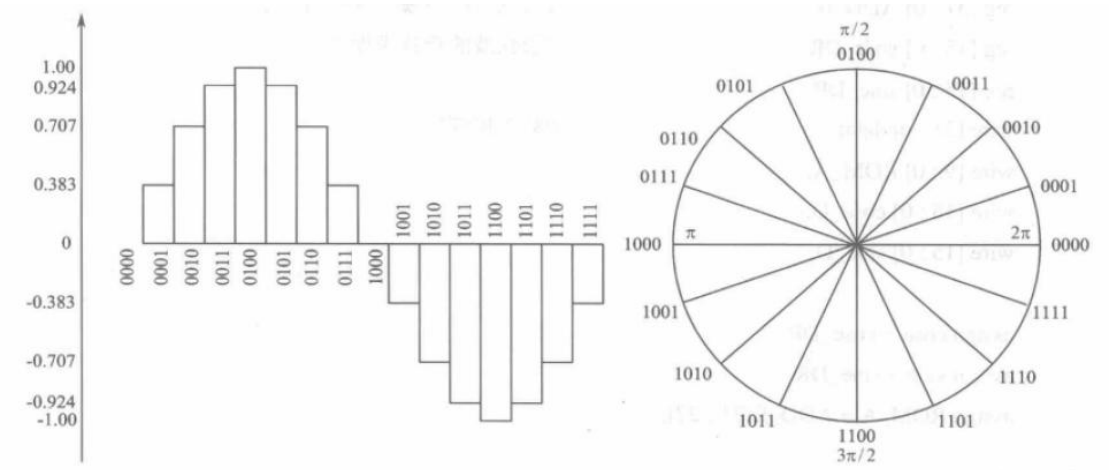



相位偏移：$P_{offset} = \frac{P_c}{2^N}$

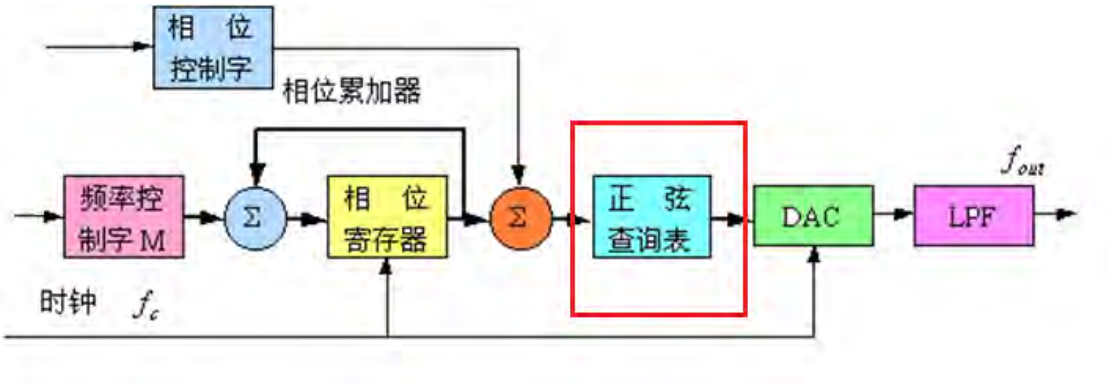
$P_{offset}$相位偏移量，$P_c$相位控制字

波形存储

正弦信号相位与幅度的对应关系

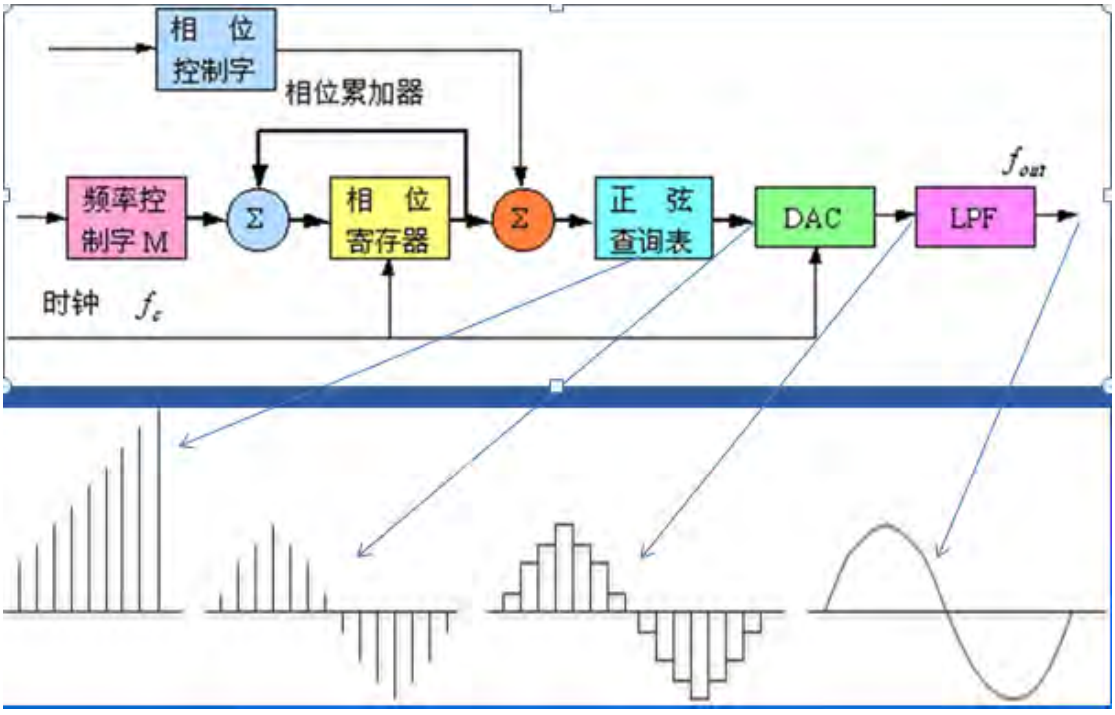



可以将正弦波波形看作一个矢量沿相位圆转动，相位圆对应正弦波一个周期的波形。波形中的每个采样点对应相位圆上的一个相位点。




相位累加器的值作为 ROM 的地址，读取 ROM 的相位幅度，实现相位到幅度的转换。

**AD 转换和滤波**



分析：

DDS 优点

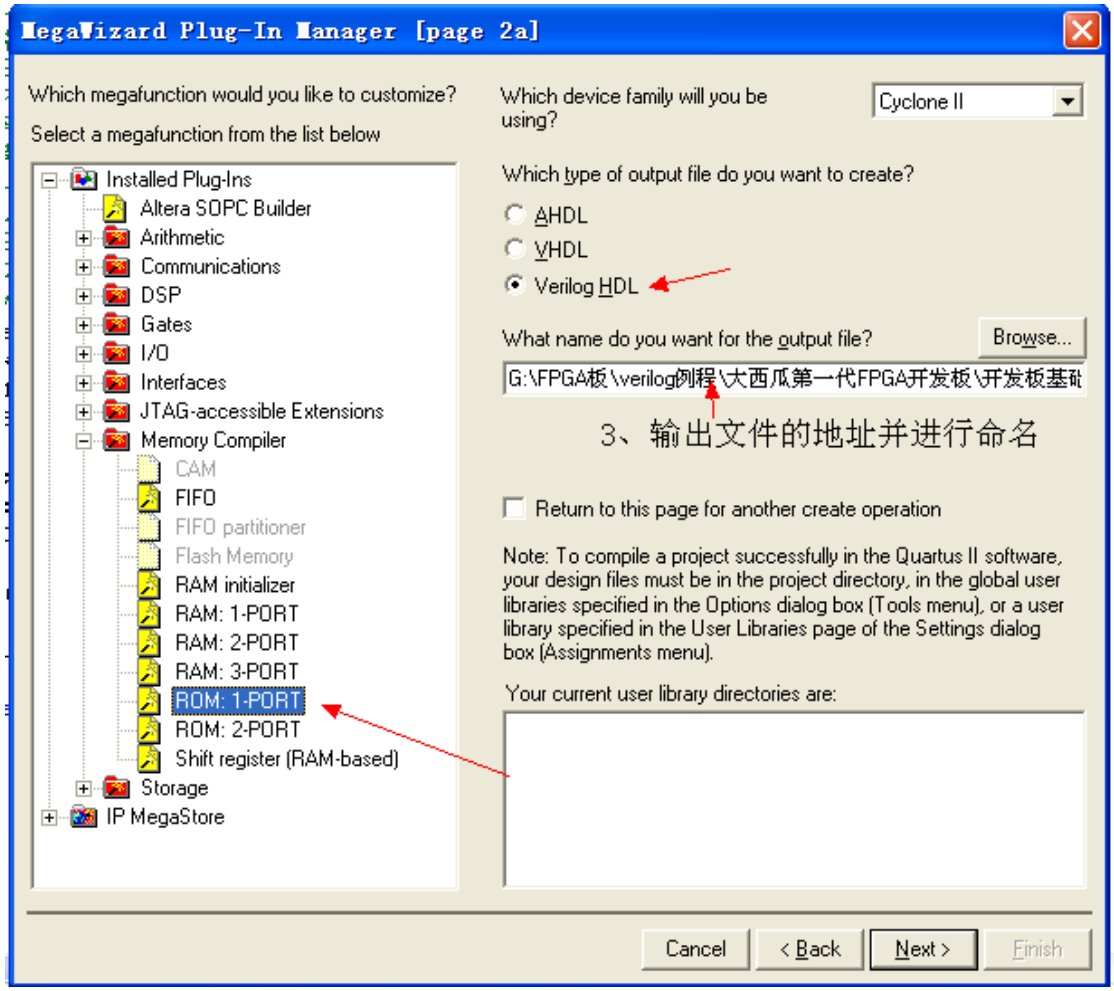
- 频率分辨率高，可达 2 的 N 次。
- 频率切换速度快，可达 us 量级。
- 频率切换时相位连续。
- 可以产生任意波形。

DDS 缺点

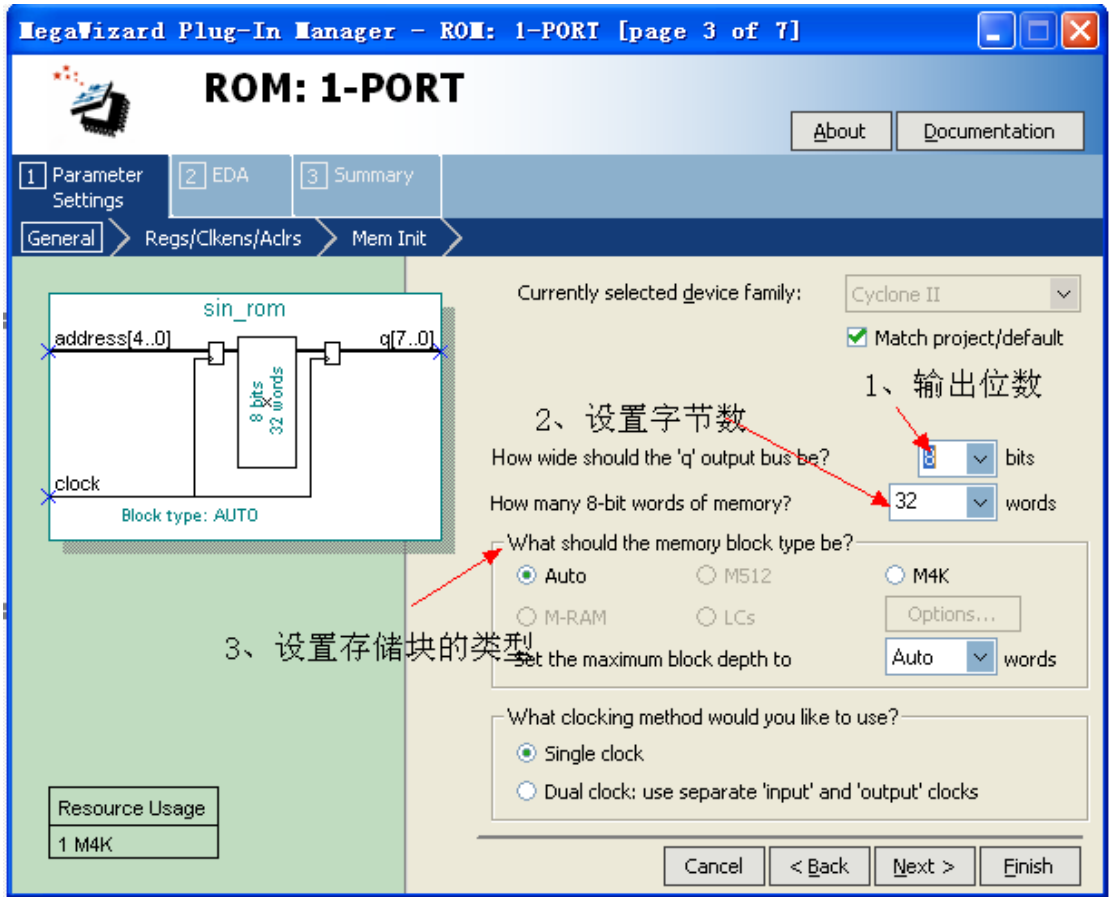
- 输出频带范围有限。
- 输出杂散大。

## 9.12.2 ROM 的调用

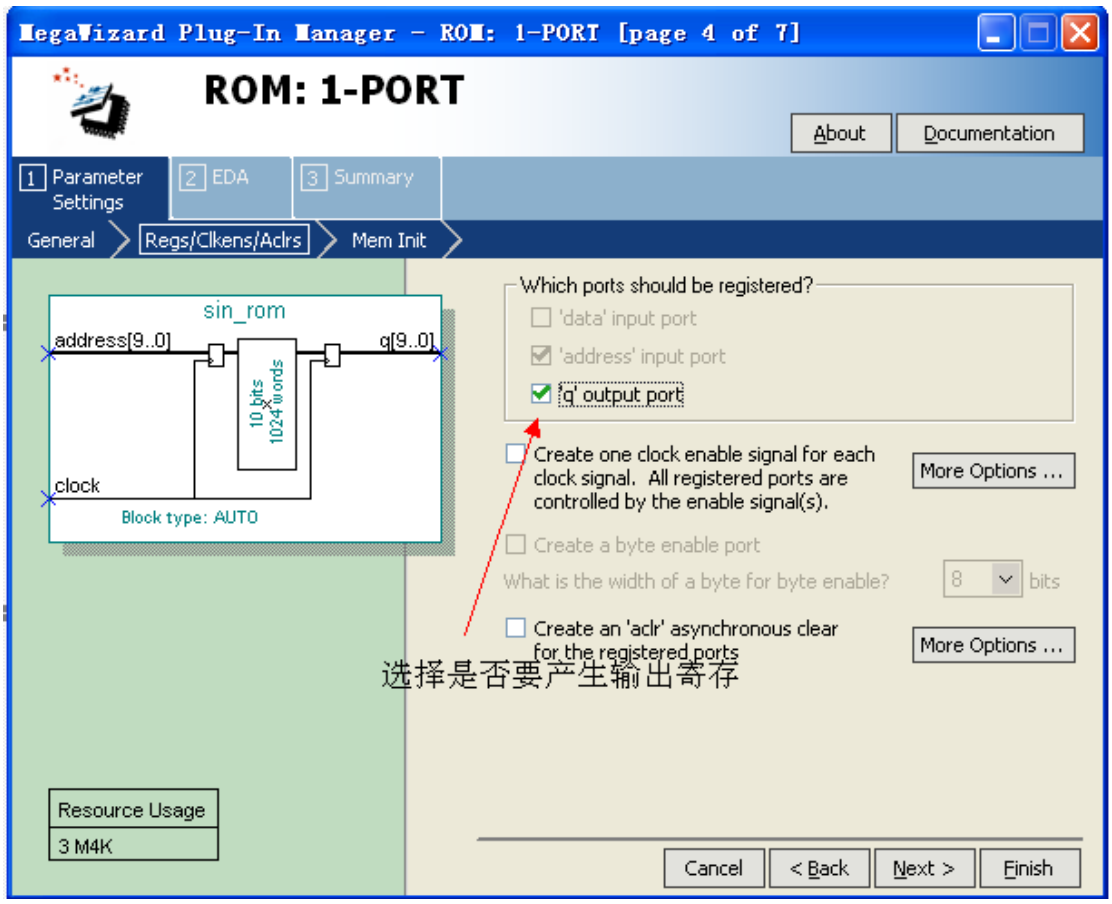
（1）选择【Tools】--【Megawizard plug-zn Manager】，然后创建一个新的宏功能模块，选择输出文件的语言类型和单端口 ROM




（2）ROM 位宽和存储深度的设置




（3）选择是否要产生输出寄存，这个选择不用



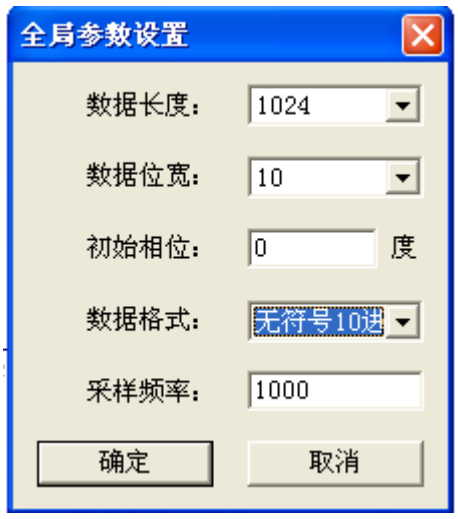

（4）MIF 文件的生成

ROM 的初始文件的格式有两种：一种为.HEX 文件；一种为.MIF 文件。

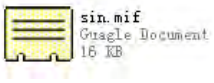
生成这两种文件的方法有很多种，这里因为要产生数字 10 位的正弦波，所以采用 Mif_Maker2010 软件进行生成。

步骤如下：

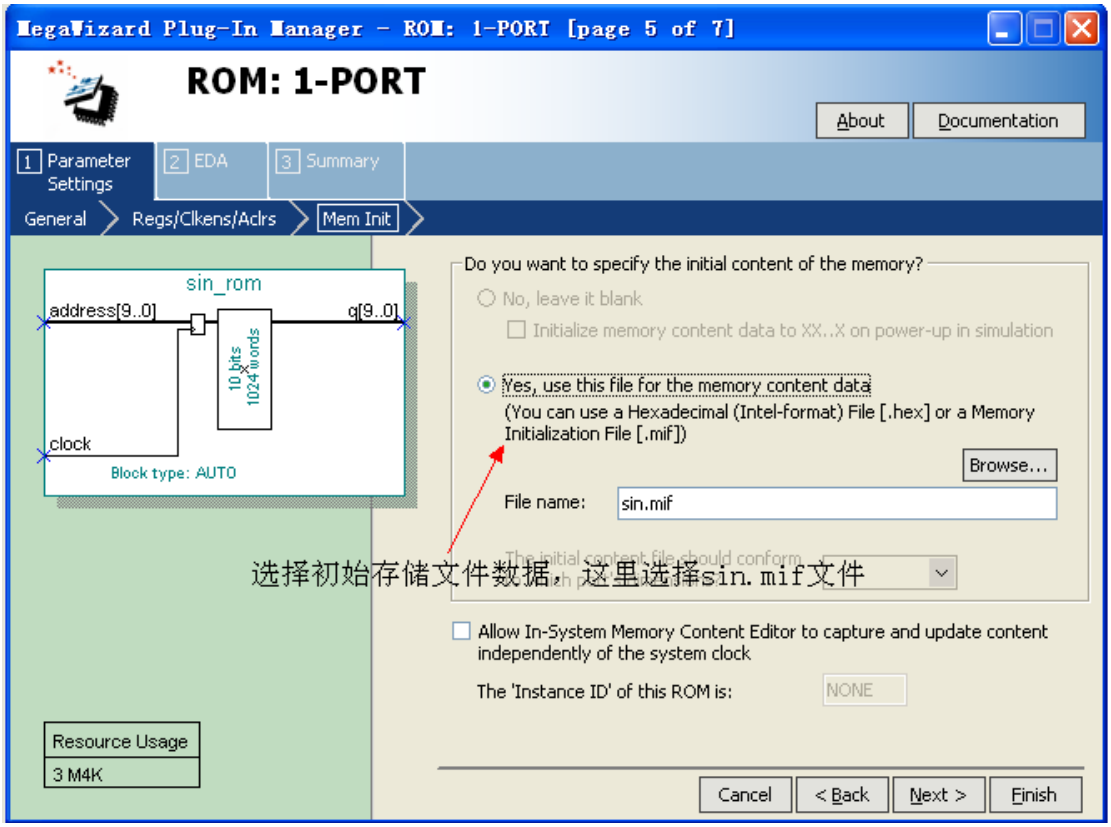
【设定波形】---【全局参数的设置】--【确定】--【设定波形】--【正弦波】，即可得到相应的波形；



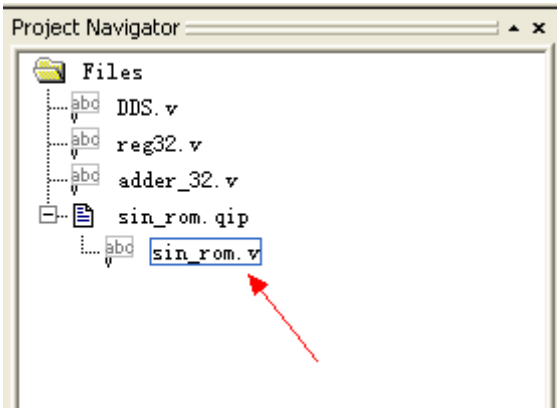
【文件】--【保存】，选择保存途径并进行命名即可得到生成的 MIF 文件




（5）MIF 文件的调用



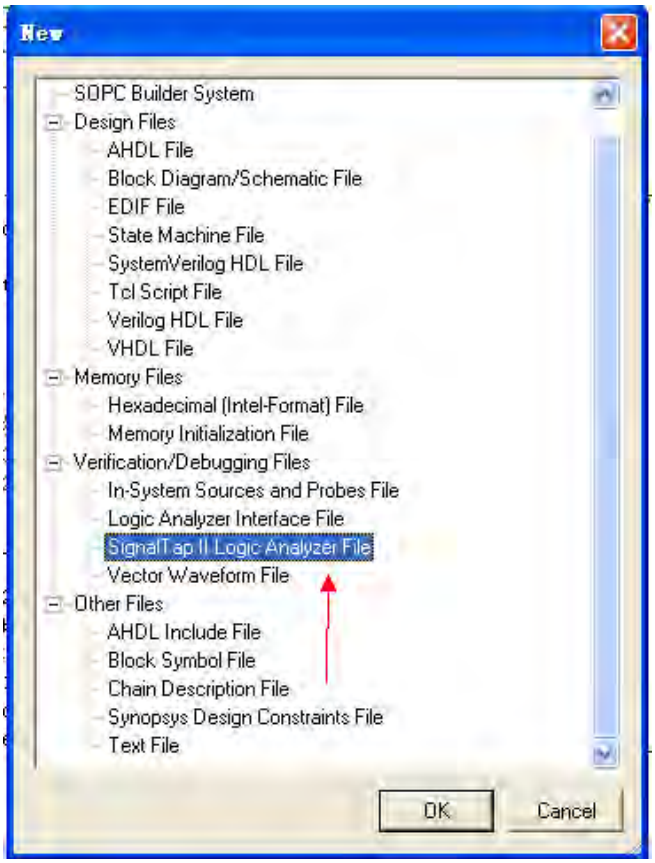

(6)然后一直点击 Next,最后点击 Finlish 便生成该文件：



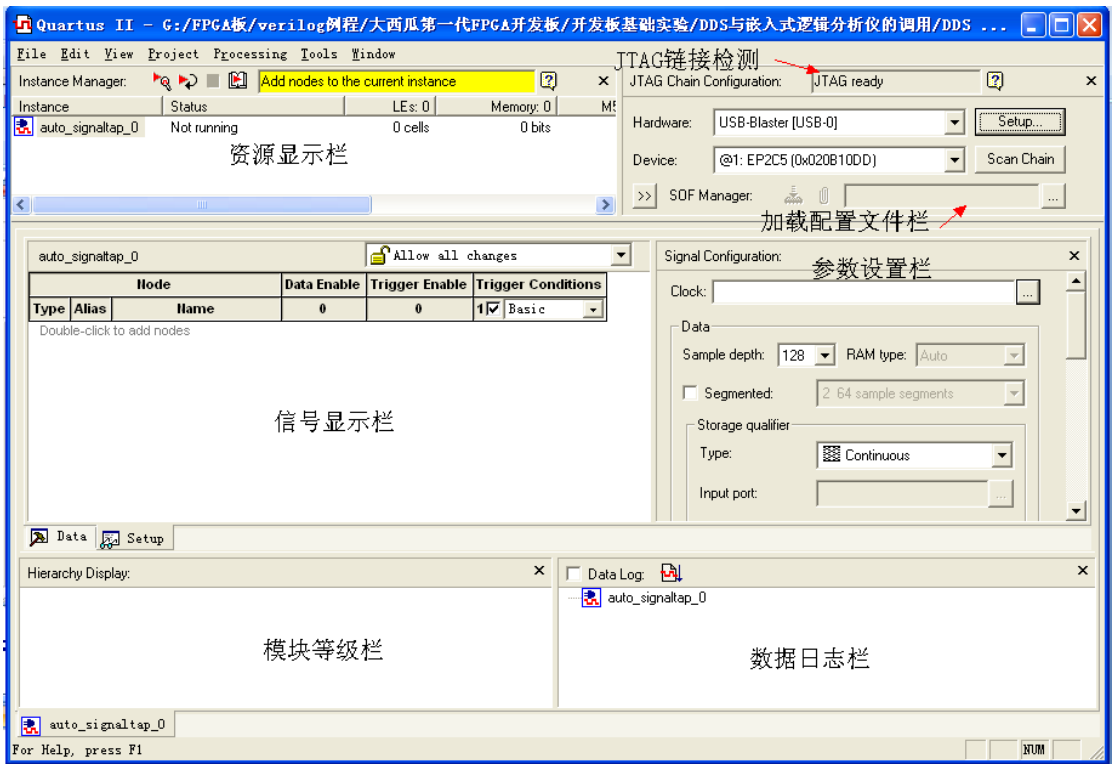

## 9.12.3 嵌入式逻辑分析仪的使用

- SignalTap II逻辑分析仪是 Quartus II 软件中集成的一个内部逻辑分析软件，主要是用来观察设计的内部信号波形，方便查找设计中的缺陷。
- 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。

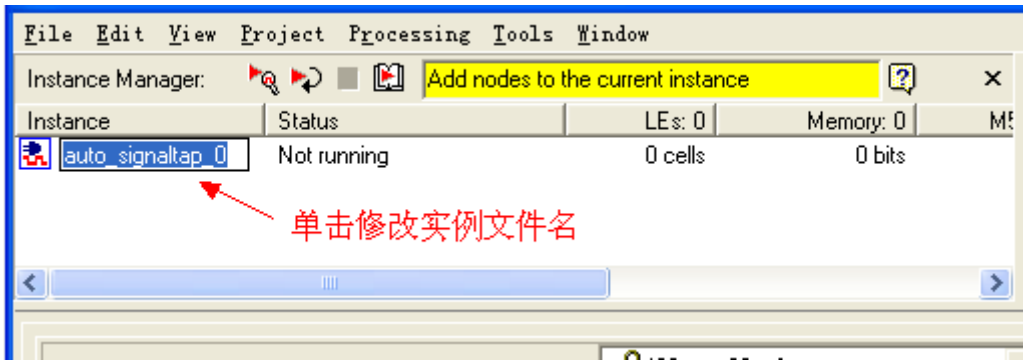
- 通过菜单【Tools】-【 SignalTap II Logic Analyzere】打开 SignalTapII 窗口，将自动新建一个 STP 文件



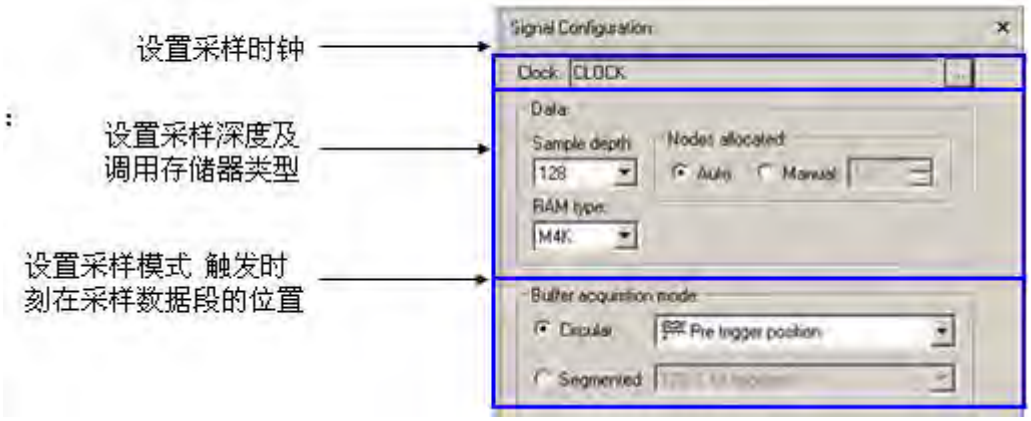

生成后出现如下窗口：



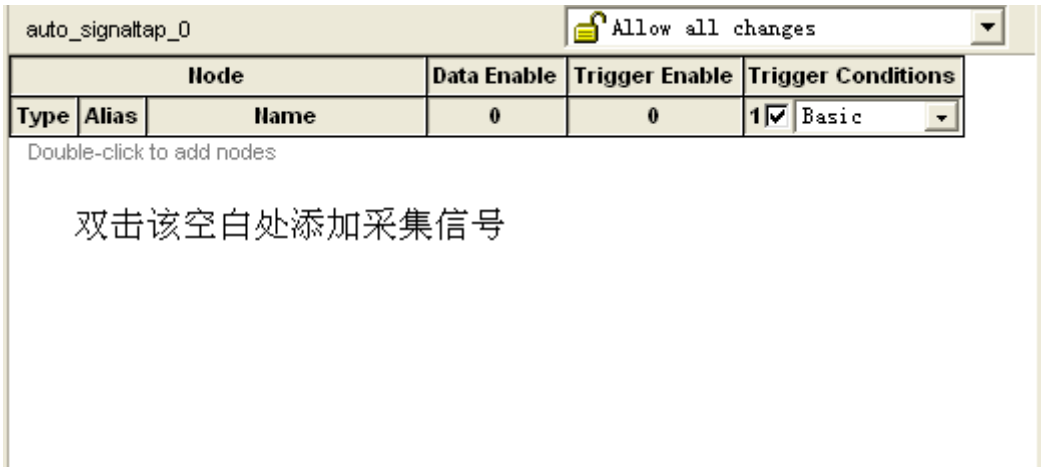
在上面的窗口中进行以下的设置：




1、参数设置栏的设置




2、信号显示栏的设置：

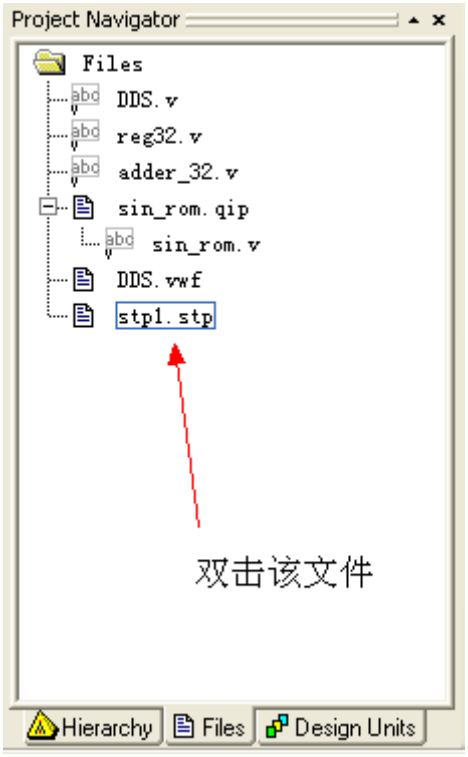



添加方式与功能仿真时添加测试引脚一致，SignalTap II 能够采集 FPGA 的引脚信号和内部寄存器的信号

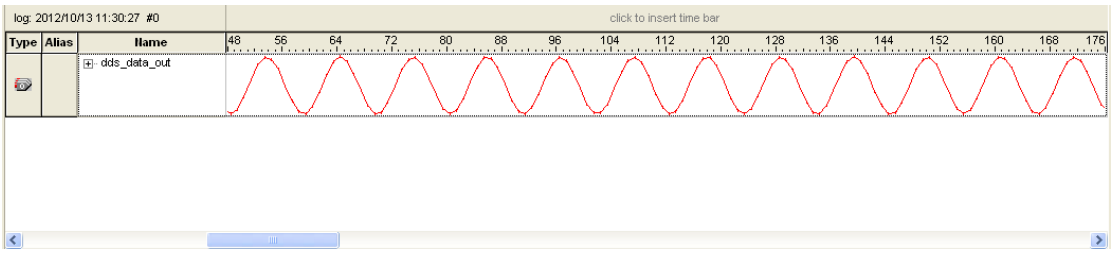
1、 保存 SignalTap II 文件，然后关闭 SignalTap II 窗口，关闭后会提示是否将该文件加入整个工程中，这里选择 YES，软件将重新综合整个工程项目。

观察 SignalTap II 采集回来的信号，该信号是实际真实信号

**下载**




在出现的窗口中，单击  ，在下载成功后单击  ，然后在信号显示栏中选中信号右击，在出现的下拉列表中选择【Bus Display Form】--【Unsigned Line Char】即可观察线性信号。如下：

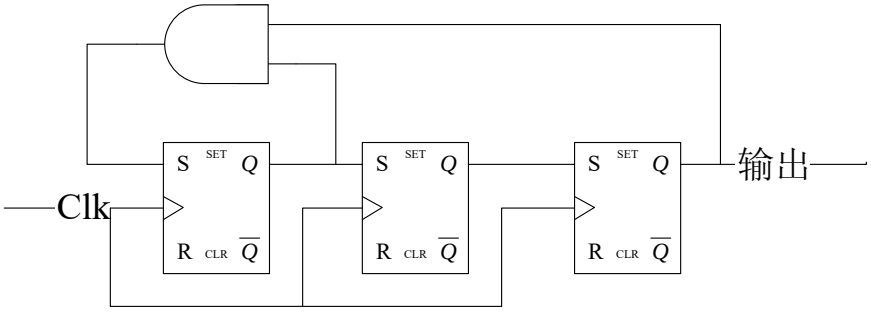


# 第十章基于 FPGA 的通信系统实验

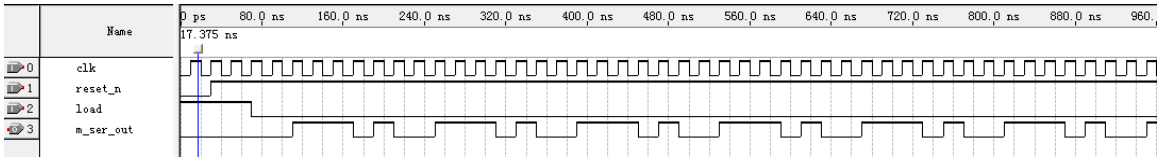
## 10.1、伪随机信号发生器

### 1、伪随机信号发生器原理

伪随机信号发生器又叫 PN 序列发生器或者是 m 序列发生器。m 序列是一种线性反馈寄存器序列，m 序列的产生可以利用 r 级寄存器产生长度为 2^r-1 的 m 序列，该实验中采用 3 级寄存器产生 7 序列发生器。其原理框图如下：（实验中反馈的信号采用异或进行反馈）




仿真波形：




### 2、实验代码

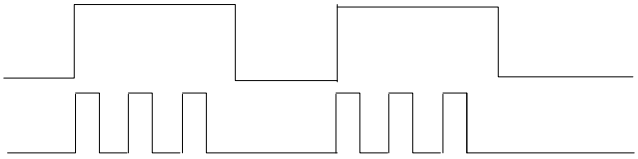
```
module m_ser(
                clk,
                reset_n,
                load,
                m_ser_out
                );
input       clk;
input       reset_n;
input       load;
output      m_ser_out;
wire        clk;
wire        reset_n;
wire        load;
reg         m_ser_out;
reg [2:0] m_code;
always @(posedge clk or negedge reset_n)
```

```
begin
  if(!reset_n)
    begin
      m_code<=3'b000;
      m_ser_out<=1'b0;
    end
  else
    if(load)
      begin
        m_code<=3'b001;          //置数初始化
        m_ser_out<=m_code[2];
      end
    else
      begin
        m_code[2:1]<=m_code[1:0];
        m_code[0]<=m_code[2] ^ m_code[0];//将 2 和 0 进行异或然后放到 0
        m_ser_out<=m_code[2];
      end
  end
endmodule
```

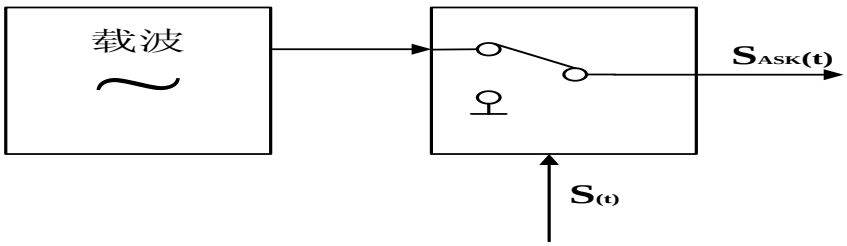
## 10.2、2ASK 调制

### 1、原理

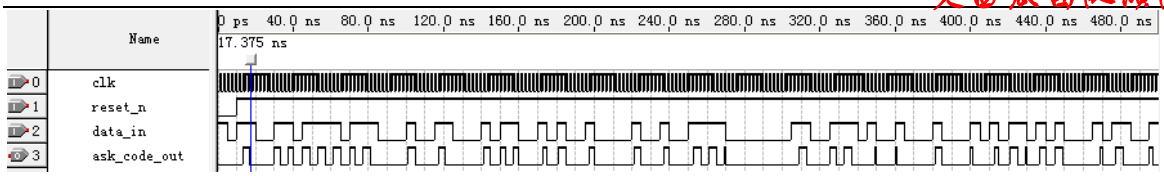
在通信系统中，有时经常需要进行二进制数字调制。2ASK 即是二进制幅值键控，2ASK 的调制原理就是：基带信号为“0”时，输出保持为“0”，基带信号为“1”时，输出一个特定频率的信号，如下图所示：



原理图如下：




### 2、实验仿真




通过仿真结果可以看出，当 data_in 输入为高时，调制输出一定频率的信号，输入为低时，调制输出低电平。

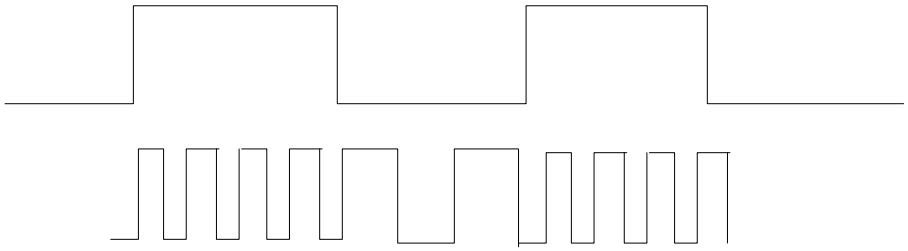
## 3、实验代码

```
module ask(
                clk,
                reset_n,
                data_in,
                ask_code_out
                );
input clk;
input reset_n;
input data_in;
output ask_code_out;
wire clk;
wire reset_n;
wire data_in;
reg [2:0]clk_cnt;
reg clk_div;
always @(posedge clk or negedge reset_n)//产生分频信号
  begin
    if(!reset_n)
      begin
        clk_cnt<=3'd0;
        clk_div<=1'b0;
      end
    else
      if(clk_cnt==3'd1)
        begin
          clk_div<=~clk_div;
          clk_cnt<=3'd0;
        end
      else
        clk_cnt<=clk_cnt + 1'b1;
  end
assign ask_code_out= (data_in) ? clk_div : 1'b0;
endmodule
```

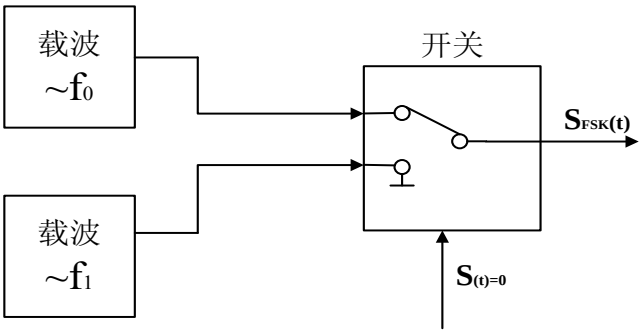
# 10.3、2FSK 调制

## 1、原理

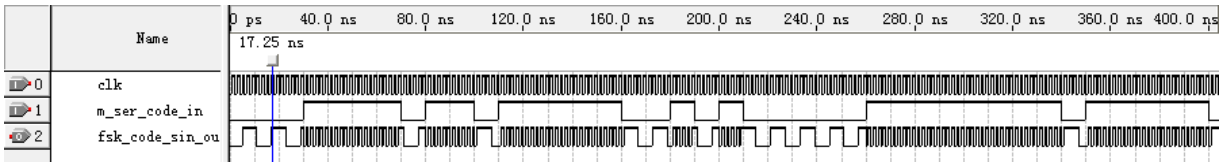
2FSK 的调制原理是当基带信号为“0”时，输出一个固定频率为 f1 的信号，当基带信号为“1”时，输出一个固定频率为 f2 的信号。如下图所示：



原理图如下：




## 2、实验仿真




## 3、实验代码

```
module fsk_code(
                clk,
                m_ser_code_in,
                fsk_code_sin_out
                );
input          clk;
input          m_ser_code_in;

output        fsk_code_sin_out;
wire          clk;
wire          m_ser_code_in;
reg       [2:0]cnt;
wire           f1;
```

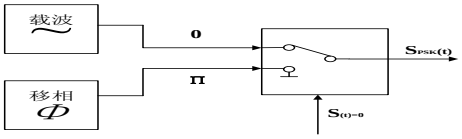
```
reg              f2;
always @(posedge clk )
  begin
    if(cnt==3'd2)
      begin
        cnt<=3'd0;
        f2<=~f2;
      end
    else
      cnt<=cnt+1'b1;
  end
assign f1=clk;
assign fsk_code_sin_out=(m_ser_code_in)? f1:f2;
endmodule
```

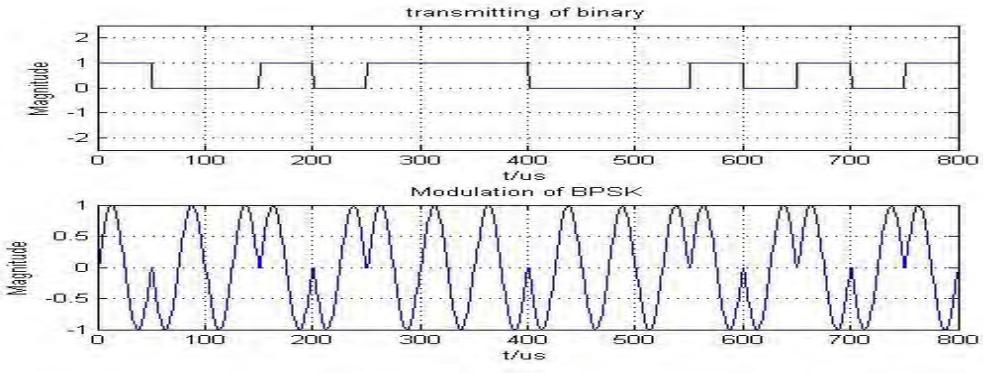
# 10.4、2PSK 调制

## 1、2PSK 调制原理

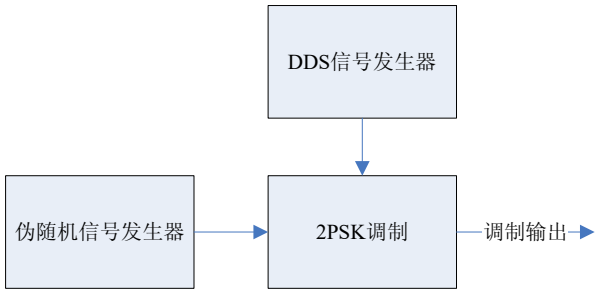
$$S_{PSK}(t)=\begin{cases} A\cos 2\pi fct, & a_n=1 \\ -A\cos 2\pi fct, & a_n=0 \end{cases} \qquad (n-1)T_b \le t \le nT_b$$

在 PSK 调制时，载波的相位随调制信号状态不同而改变。如果两个频率相同的载波同时开始振荡，这两个频率同时达到正最大值，同时达到零值，同时达到负最大值，此时它们就处于“同相”状态；如果一个达到正最大值时，另一个达到负最大值，则称为“反相”。一般把信号振荡一次（一周）作为 360 度。如果一个波比另一个波相差半个周期，我们说两个波的相位差 180 度，也就是反相。当传输数字信号时，“1”码控制发 0 度相位，“0”码控制发 180 度相位。调制原理如下：



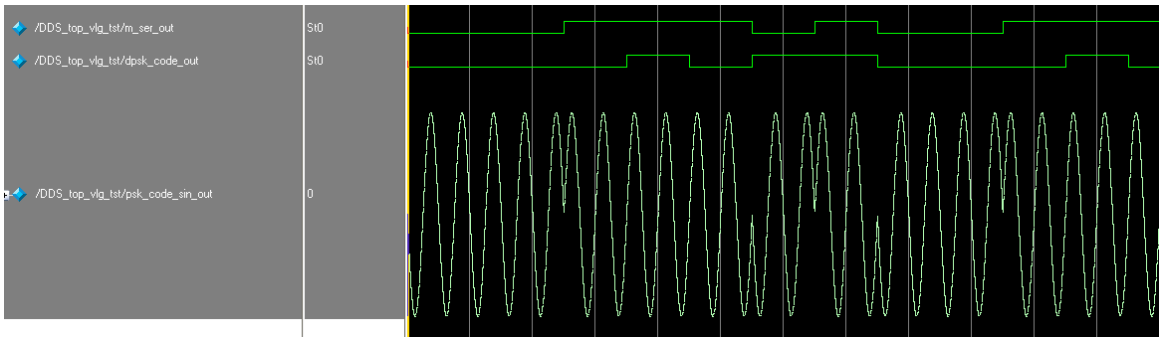
## 2、实验原理图说明




当伪随机信号发生器输出“1”时，PSK 调制输出 0 度相位的正弦波形；当伪随机信号发生器输出“0”时，PSK 调制输出 180 度相位的正弦波形；在这里，不详细说明“伪随机信号发生器”和“DDS 信号发生器的原理”，具体可以参考相应的章节。

## 3、仿真效果

采用 ModelSim 进行仿真：




## 4、实验代码

```
//“1”码控制发 0 度相位，“0”码控制发 180 度相位
module psk_code(
                clk,
                m_ser_code_in,
                dds_sin_data_in2,//10k 正弦波
                dds_sin_data_in3,//10k 正弦波,相位相差 180
                psk_code_sin_out//
                );
input          clk;
input          m_ser_code_in;
input   [9:0]dds_sin_data_in2;
input   [9:0]dds_sin_data_in3;
output [9:0]psk_code_sin_out;
wire           clk;
wire           m_ser_code_in;
wire    [9:0]dds_sin_data_in2;
wire    [9:0]dds_sin_data_in3;
```

```
assign psk_code_sin_out=(m_ser_code_in)? dds_sin_data_in2:dds_sin_data_in3;
endmodule
```

# 10.5、2DPSK 调制

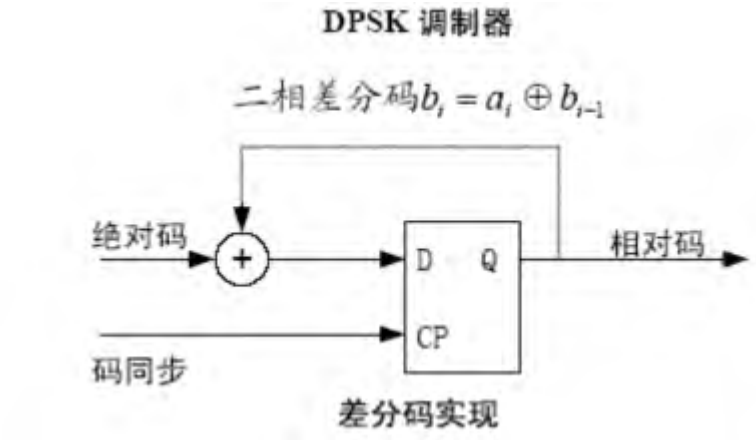
## 1、2DPSK 原理

在 2PSK 中，是利用载波相位的绝对数值来传送数字信息，因而称为绝对调相；2DPSK 即是二进制差分相移键控，不利用载波相位传送数字信息，而是利用前后码元的相对相位变化传送数字信息。

实现相对调相的常用方法有：先对数字基带信号进行差分编码，将绝对编码转换成差分编码，然后再进行绝对调相。

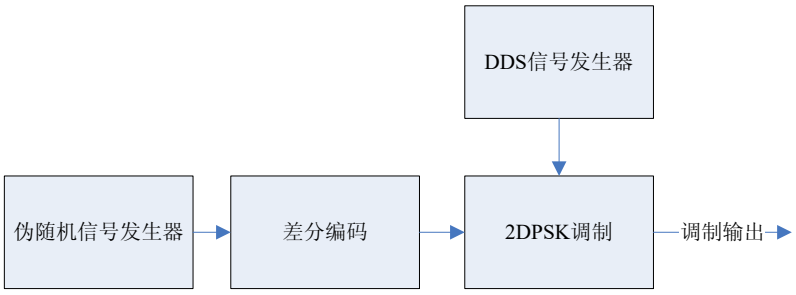
将数字基带信号由绝对编码转成差分编码的方法为：将前一个输出码元和当前的输入码元进行异或，这样就可以产生相对码。

其原理图为：



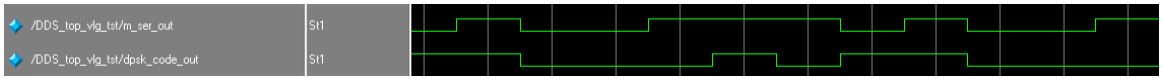

## 2、实验设计

将伪随机信号发生器产生的 m 序列进行差分编码，再将相对码进行 2DPSK 调制，当出现“0”码则输出 0 度相位正弦波，“1”码控制发 180 度相位正弦波。

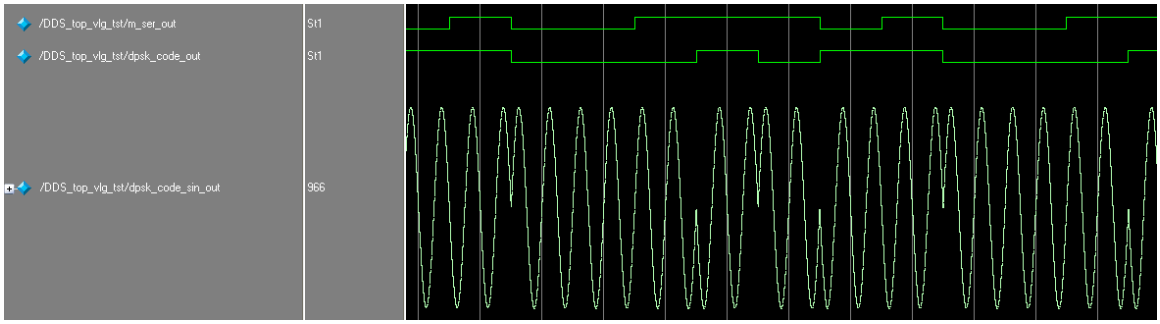



## 3、实验仿真

将绝对编码转为相对编码



2DPSK 调制：




## 4、实验代码

```
//“0”码控制发 0 度相位，“1”码控制发 180 度相位
module dpsk_code(
                    clk,
                    reset_n,            //
                    m_ser_code_in,     //PN 序列输入
                    dpsk_code_out,     //dpsk 调制输出
                    dds_sin_data_in2,//10k 正弦波
                    dds_sin_data_in3,//10k 正弦波,相位相差 180
                    dpsk_code_sin_out
                    );
input clk;
input reset_n;
input m_ser_code_in;
input [9:0]dds_sin_data_in2;
input [9:0]dds_sin_data_in3;

output dpsk_code_out;
output [9:0]dpsk_code_sin_out;

wire clk;
wire reset_n;
wire m_ser_code_in;
wire [9:0]dds_sin_data_in2;
wire [9:0]dds_sin_data_in3;
reg      dpsk_code_reg;
//差分编码
always @(posedge clk or negedge reset_n)
  begin
    if(!reset_n)
      begin
        dpsk_code_reg<=1'b0;
      end
```

```
    else
      dpsk_code_reg<=dpsk_code_reg ^ m_ser_code_in;//前一个码元与输入的码元进行异或
  end
assign dpsk_code_out=dpsk_code_reg;
assign dpsk_code_sin_out= (dpsk_code_reg) ? dds_sin_data_in3 : dds_sin_data_in2;
endmodule
```

# 第十一章宏功能模块调用实验

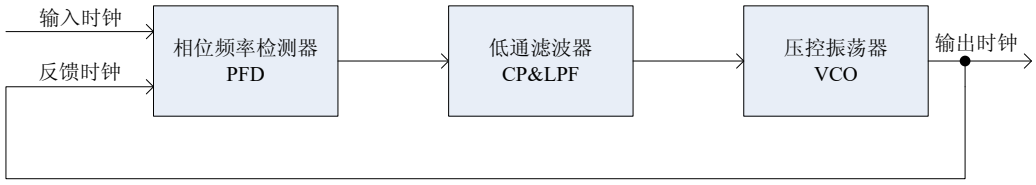
## 11.1 PLL 的使用

## 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结构

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（100M=2X50M）；

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—>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>”。

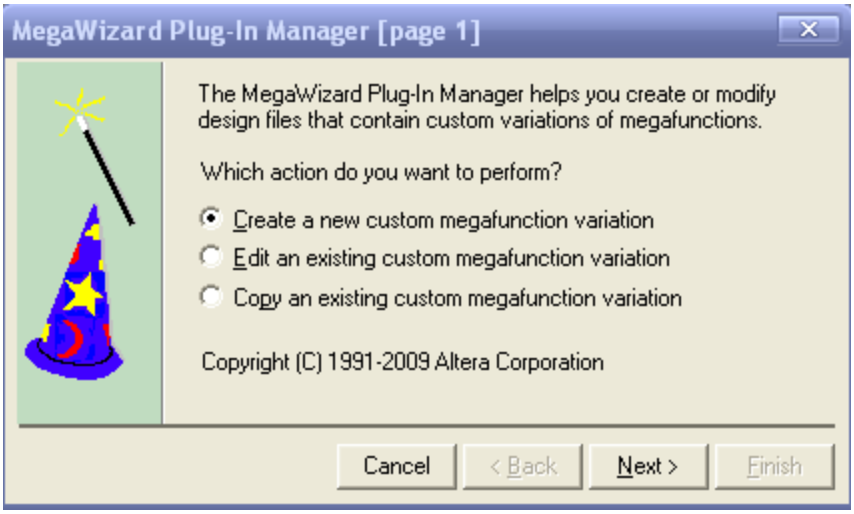



图 2

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“I/O”下拉框，

选择“ALTPLL”。

□在“Which type of output file do you wangt 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>”

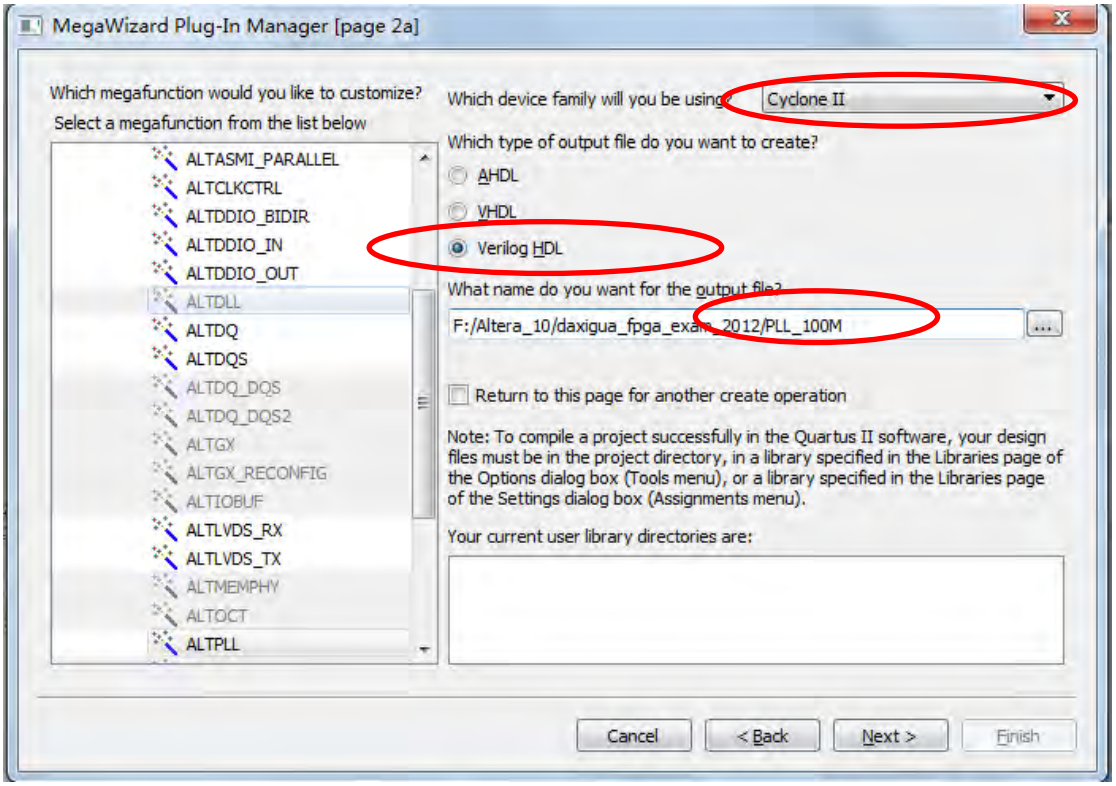



图 3 新建 PLL

**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>”。

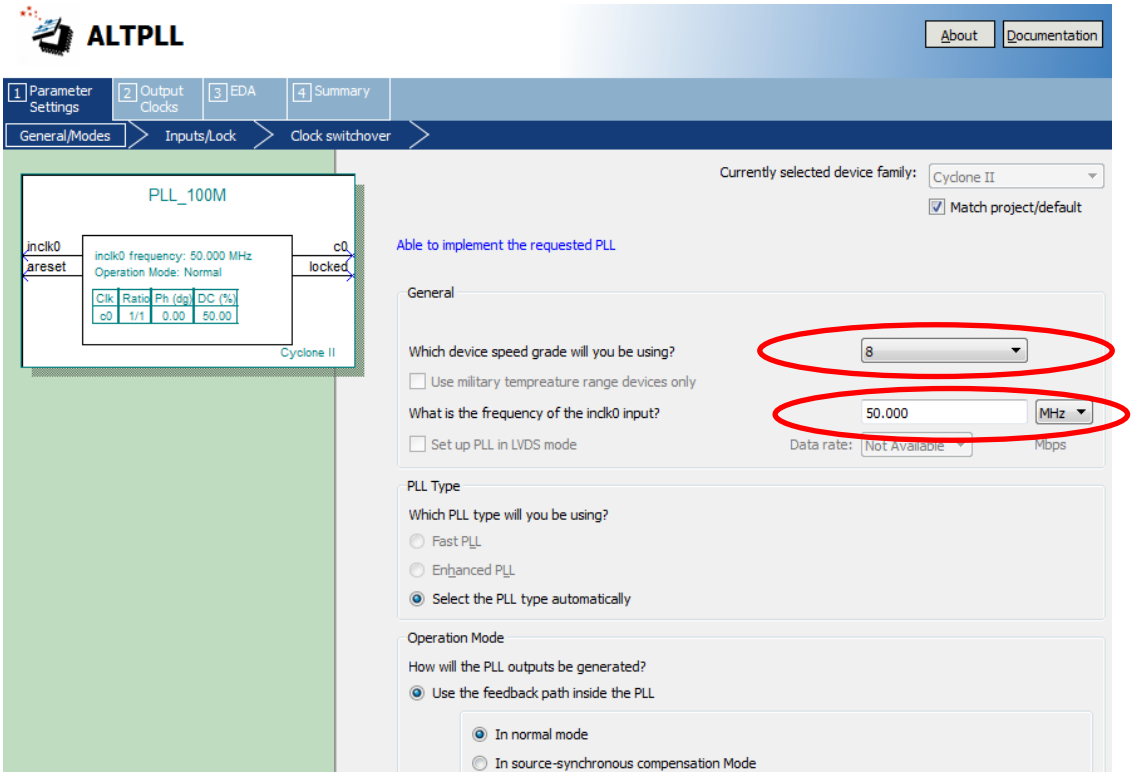



图 4 输入时钟配置

**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>”。

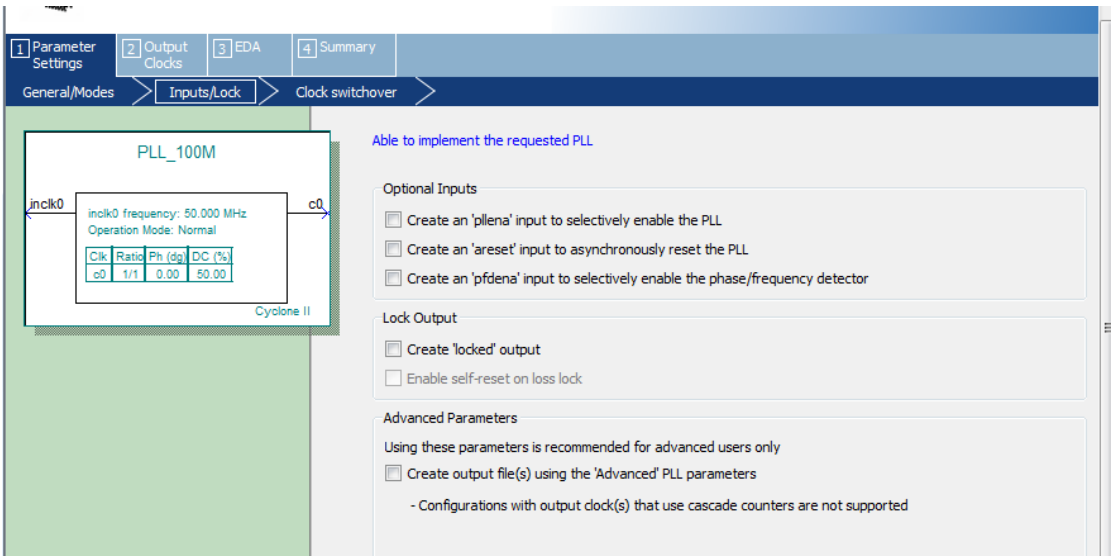



图 5 配置控制信号

**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。

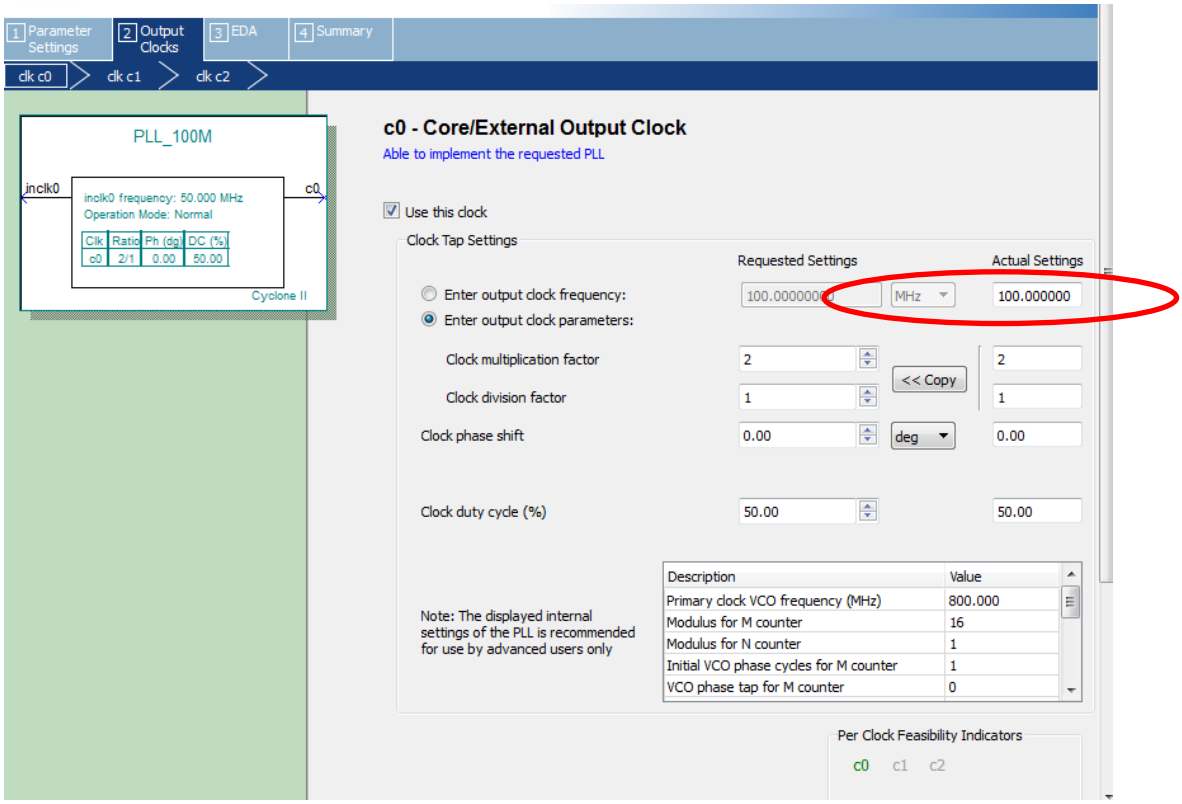
设计者也可以在“Enter output clock parameter?”后面设置相应的输出时钟和输入时钟的频率关系。“Clock Multiplication factor”后输入倍频系数，“Clock division factor”后输入分频系数，二者决定了输出时钟频率。

□在“Clock phase shift”中可以设置相位偏移。

□在“Clock ducy 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

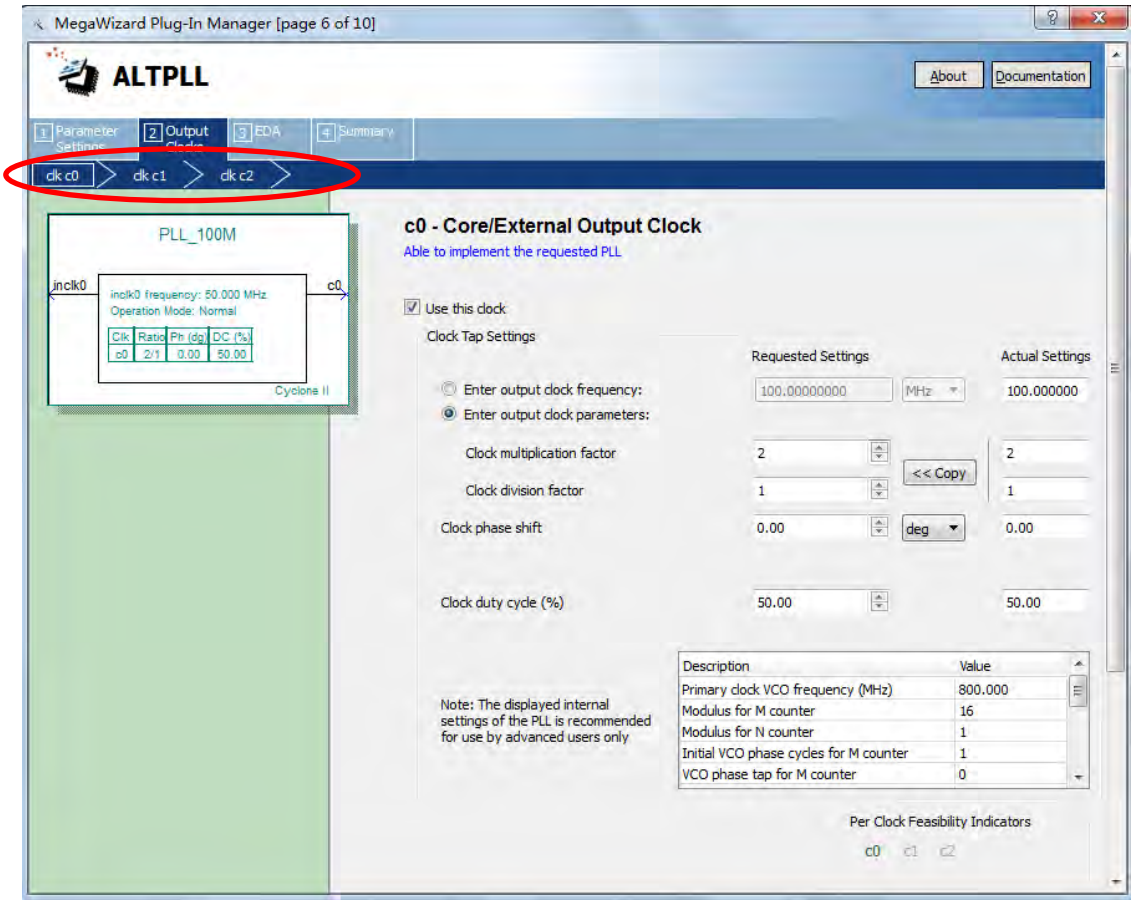



图 7 配置输出时钟 c1

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>”。


图 8

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。

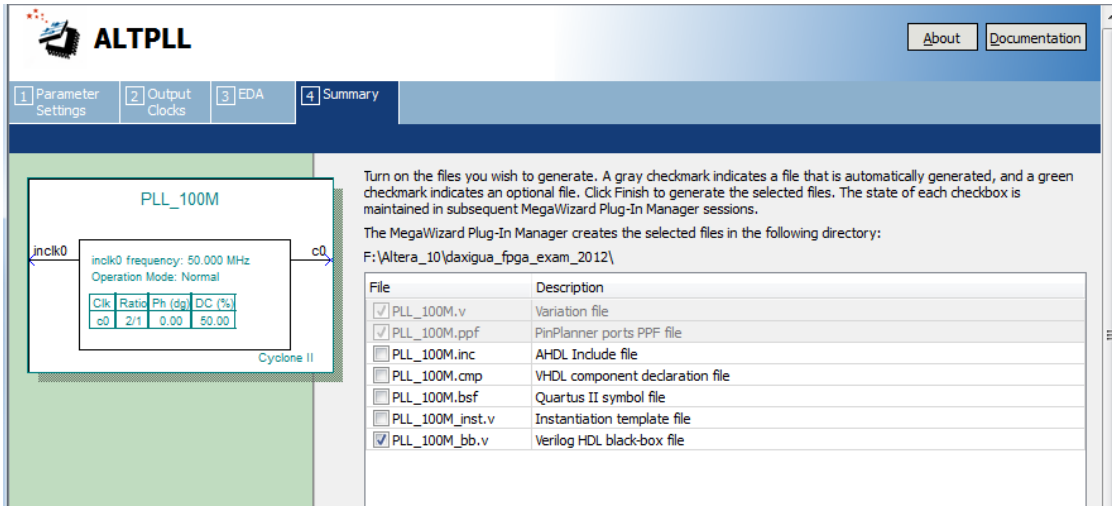



图10 输出文件

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。

## 11.2 ROM 的使用

具体请参考**第九章 9.12.2 节**

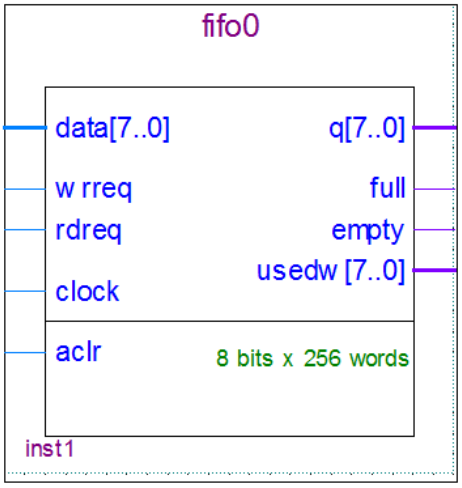
## 11.3 FIFO 的使用

### 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>”。

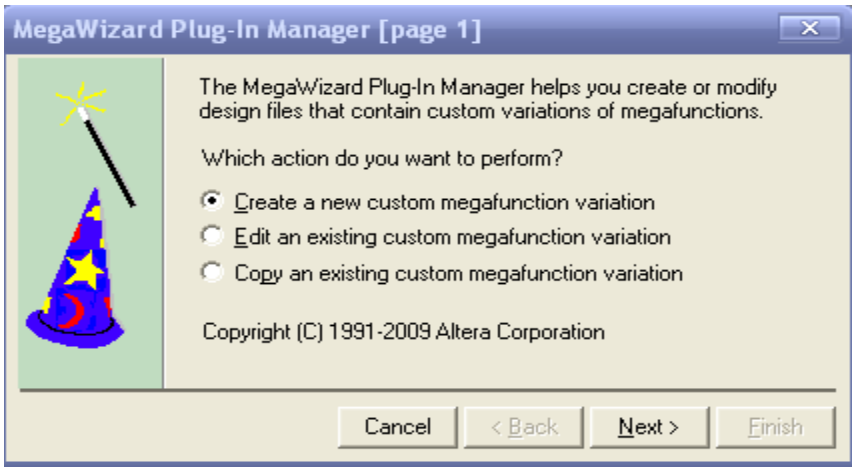



图 2

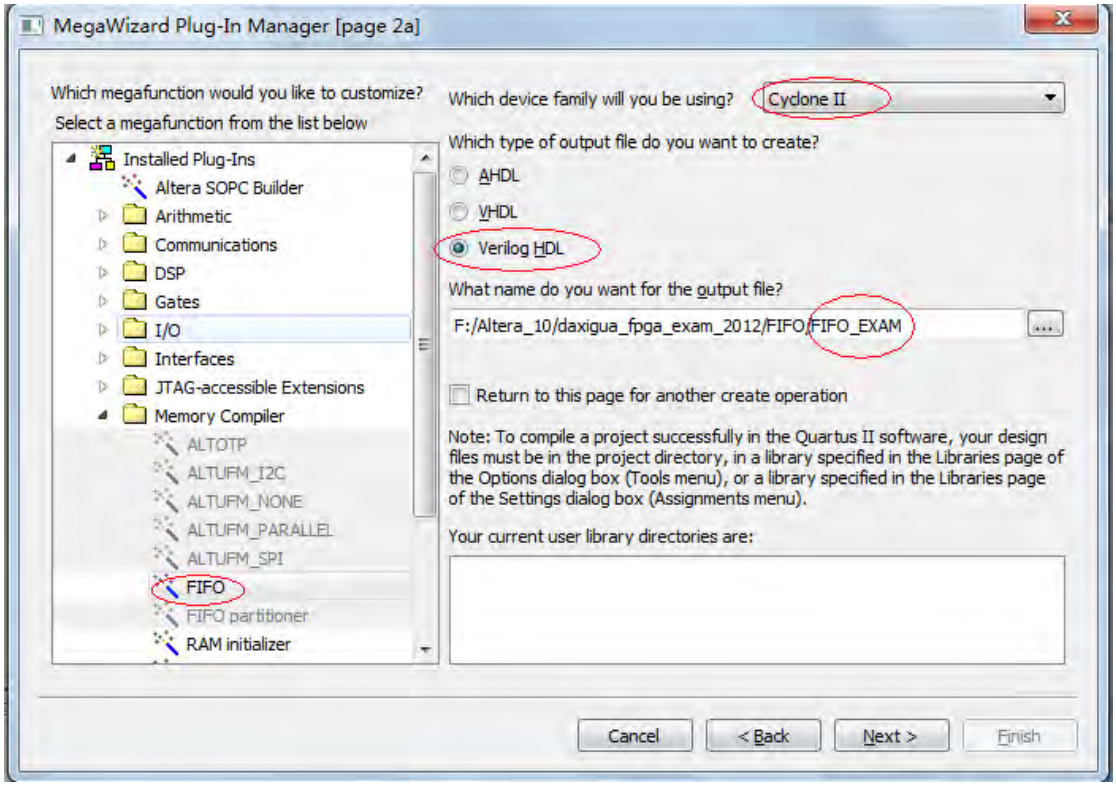
③ 如图 3 所示，进行以下配置：

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 wangt 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.

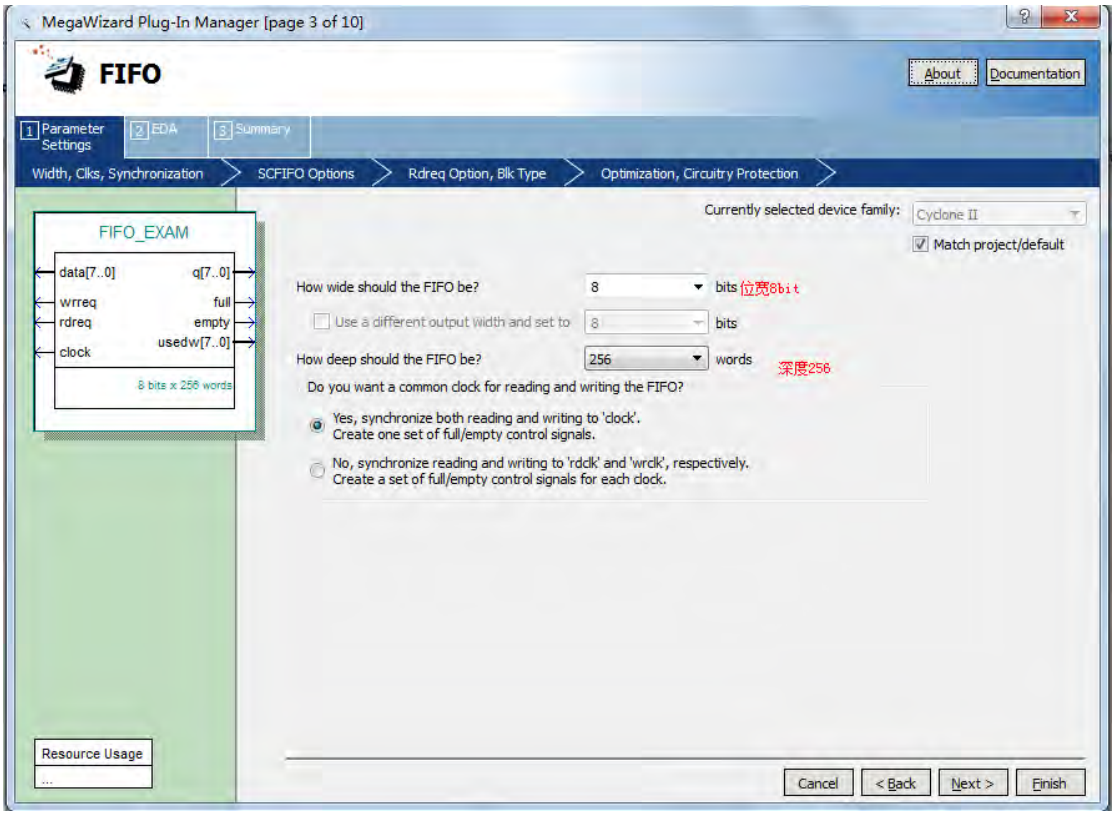



图 4

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。

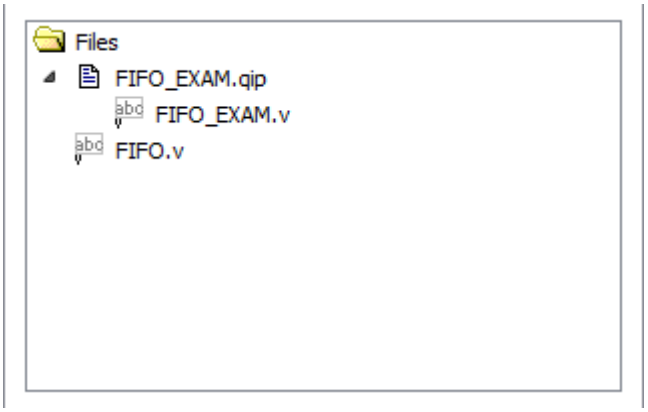
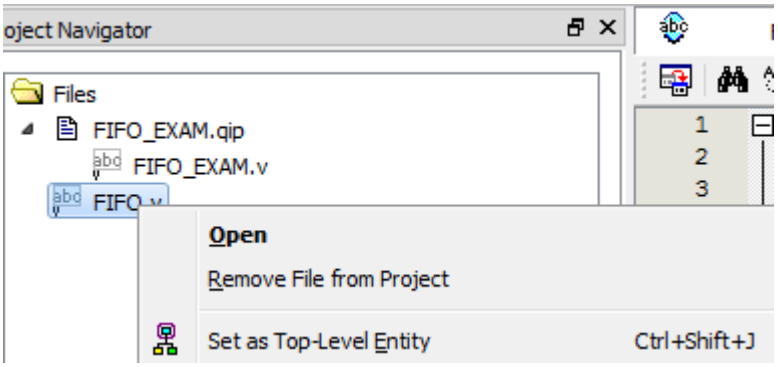



图 5

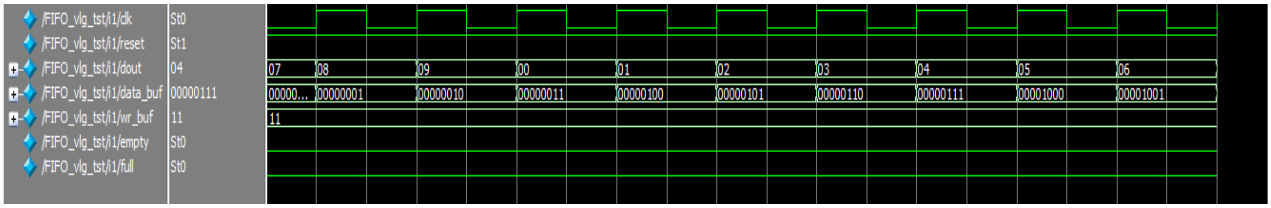
⑥ 右键点击 FIFO.V



选择 Set as Top-Level Entity

⑦ 再点击

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：




关于 modelsim 的使用，请详细参考教程 modelsim 使用篇。

# 11.4 RAM 的使用

**FPGA 片内存储器 ARM 的普通用法**

**一、Altera FPGA 内嵌 RAM**

在 Stratix IV 和 Stratix III 系列的 FPGA 中有 3 种内嵌的 RAM 块，分别是 640bit 的 MLAB、9kbit 的 M9K，以及 144kbit 的 M144K。由于实际应用中需要缓存各种类型的数据，因此这 3 种大小不同的 RAM 块就是为了满足不同的设计需求而提供给用户的。

Altera 新一代的 FPGA 内部的 RAM 都是纯同步 RAM，也就是说其读写操作都是由时钟沿触发。在 RAM 块的输入地址、数据和读写全能信号均有一级寄存器输入级，中间可以看作是一个纯异步读写的 RAM 内核，而输出的数据也有一级寄存器输出级，但该寄存器级是用户可选的。如果选上，输出数据会多延迟一拍，如图 9.4.1

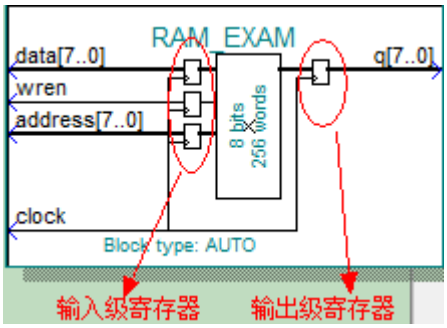



图 9.4.1RAM 块的输入和输出寄存器级

Altera 的同步 RAM 的读写时序如图 9.4.2，加上输出级寄存器后，读出数据延时了一个时钟同期，但是其时钟到输出延时非常小，因此适合于时钟频率较高的应用。

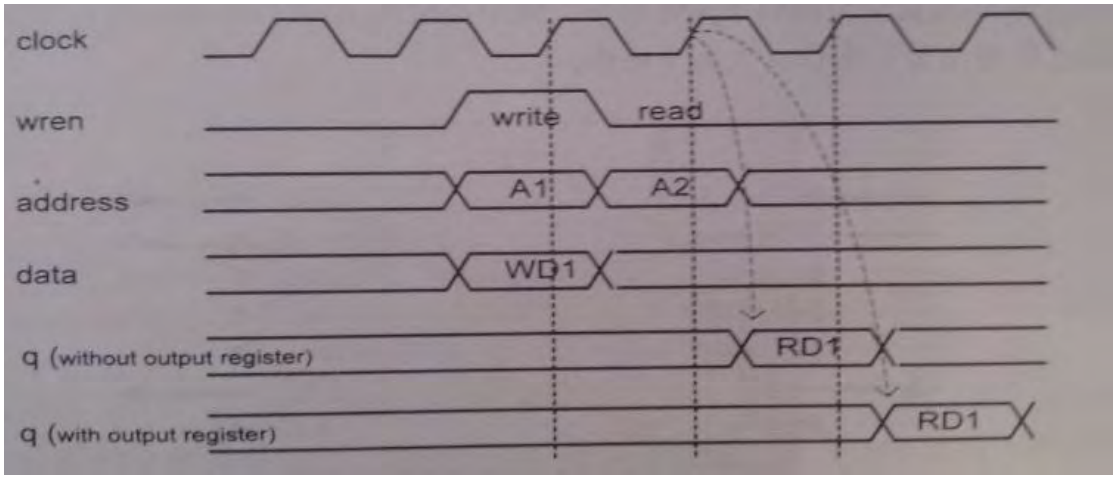
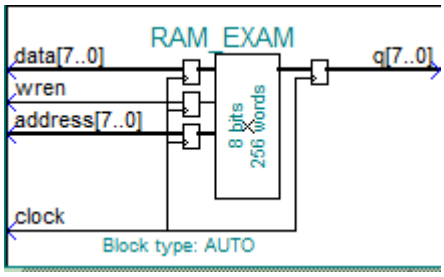



图 9.4.2 Altera 同步 RAM 读写时序

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
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>”。

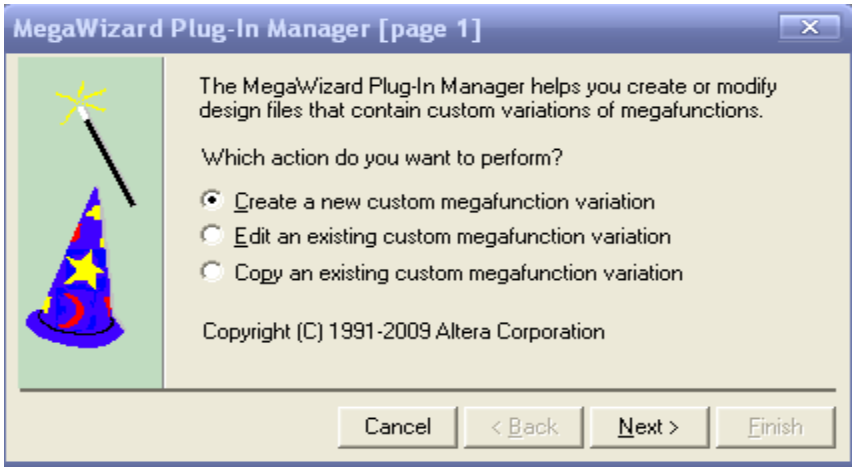



图 2

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 wangt 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>”

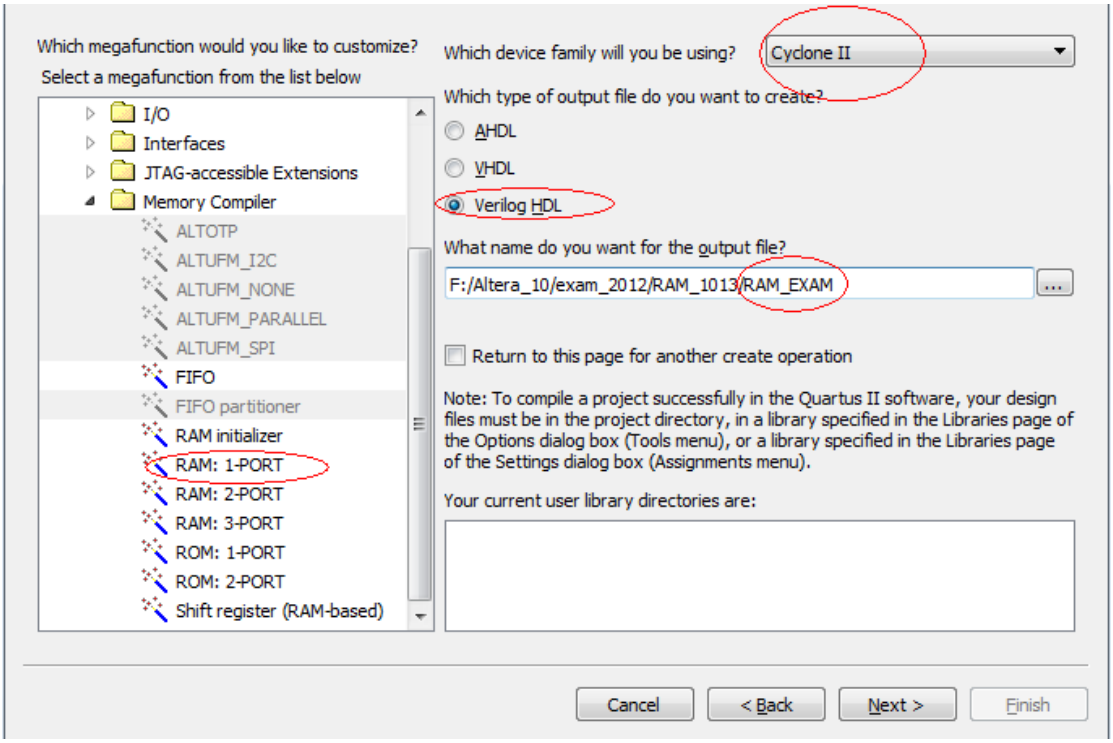



图 3

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.

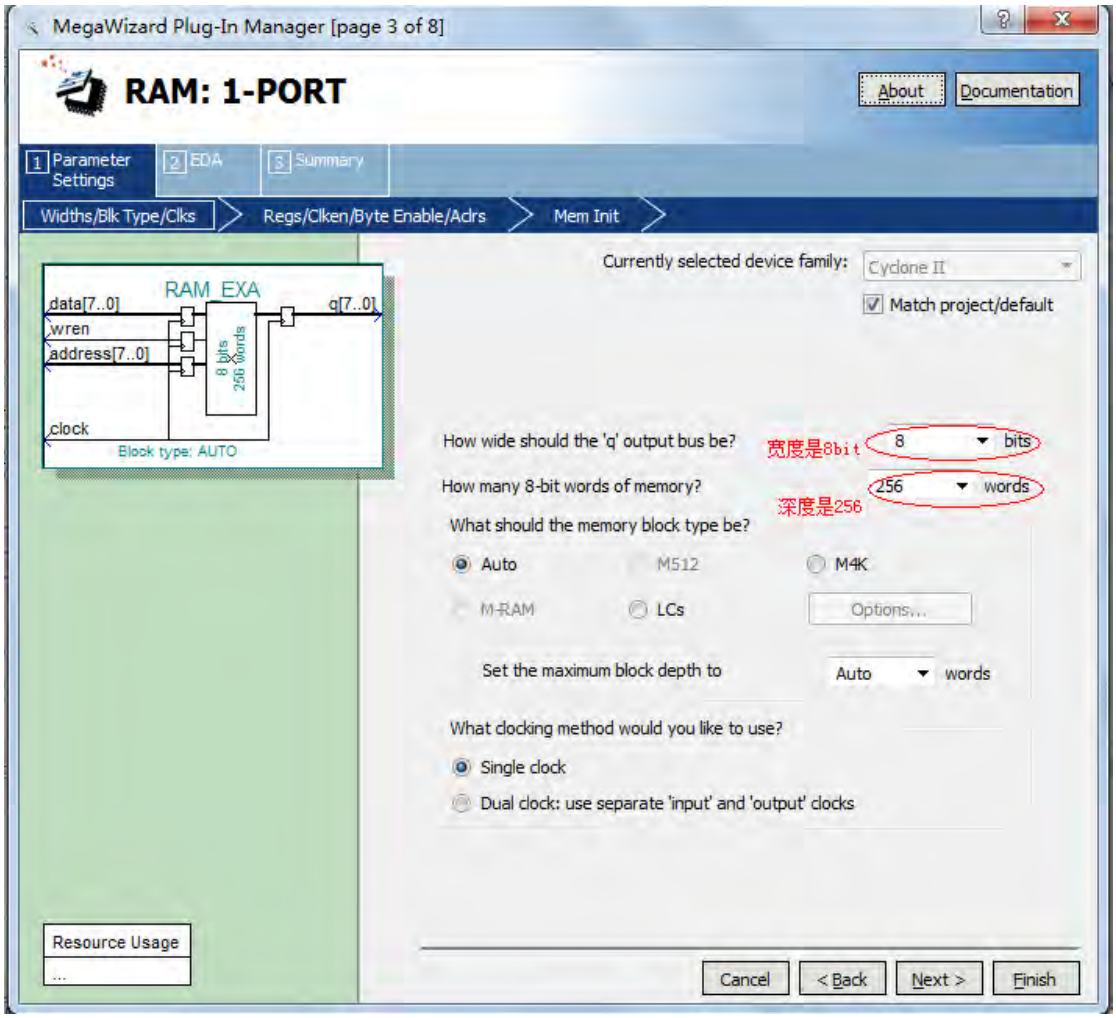



图 4

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。

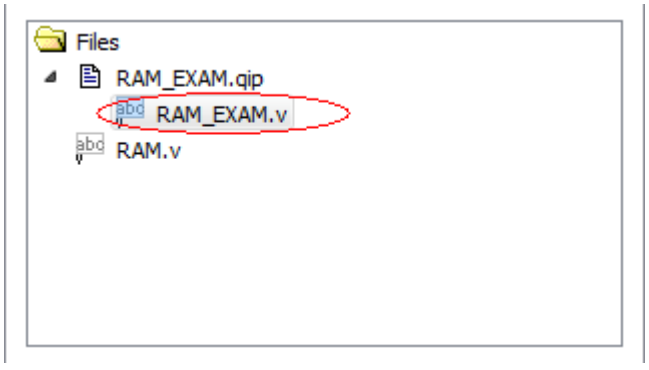
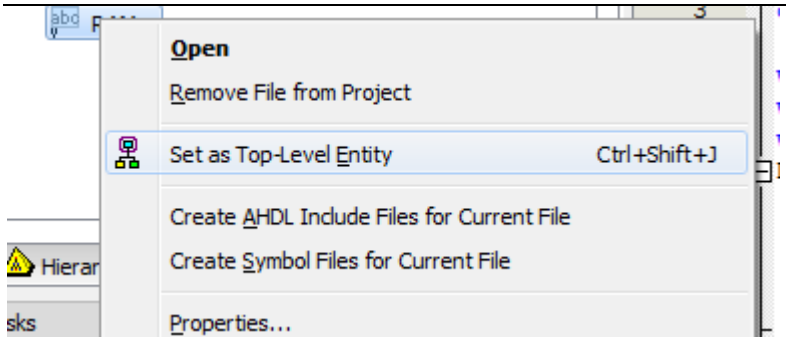



图 5

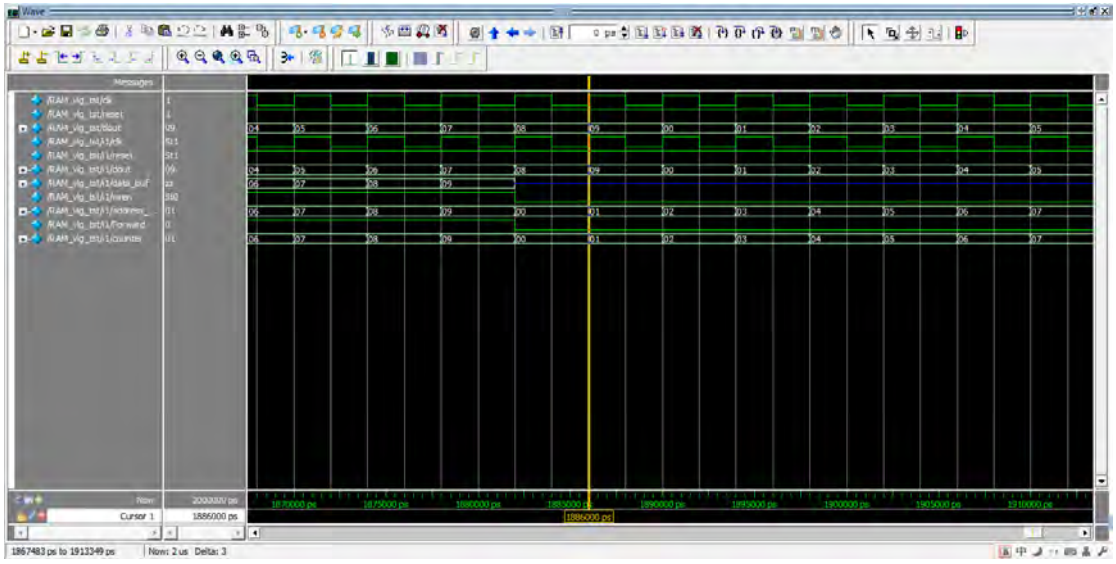
⑥ 右键点击 RAM.v




选择 Set as Top-Level Entity

⑦ 再点击

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：



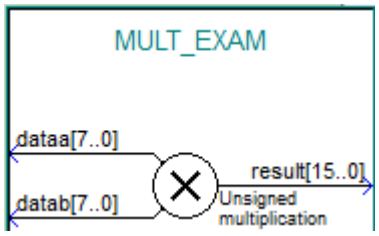
# 11.5 乘法器的使用

**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…”。


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>”。

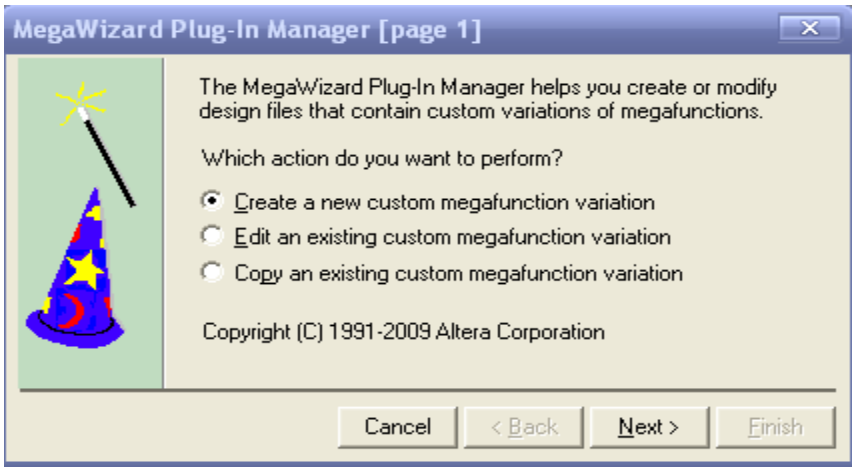



图 2

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“LPM_MULT”

在“Which type of output file do you wangt 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>”

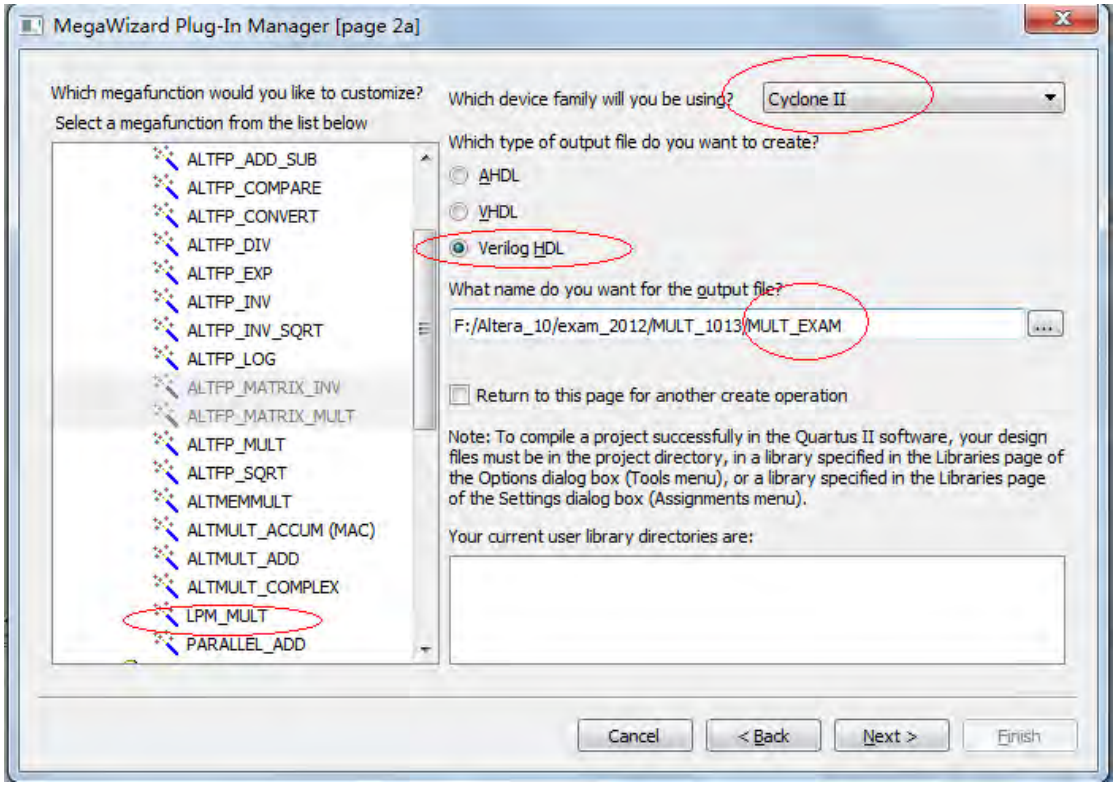



图 3

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.

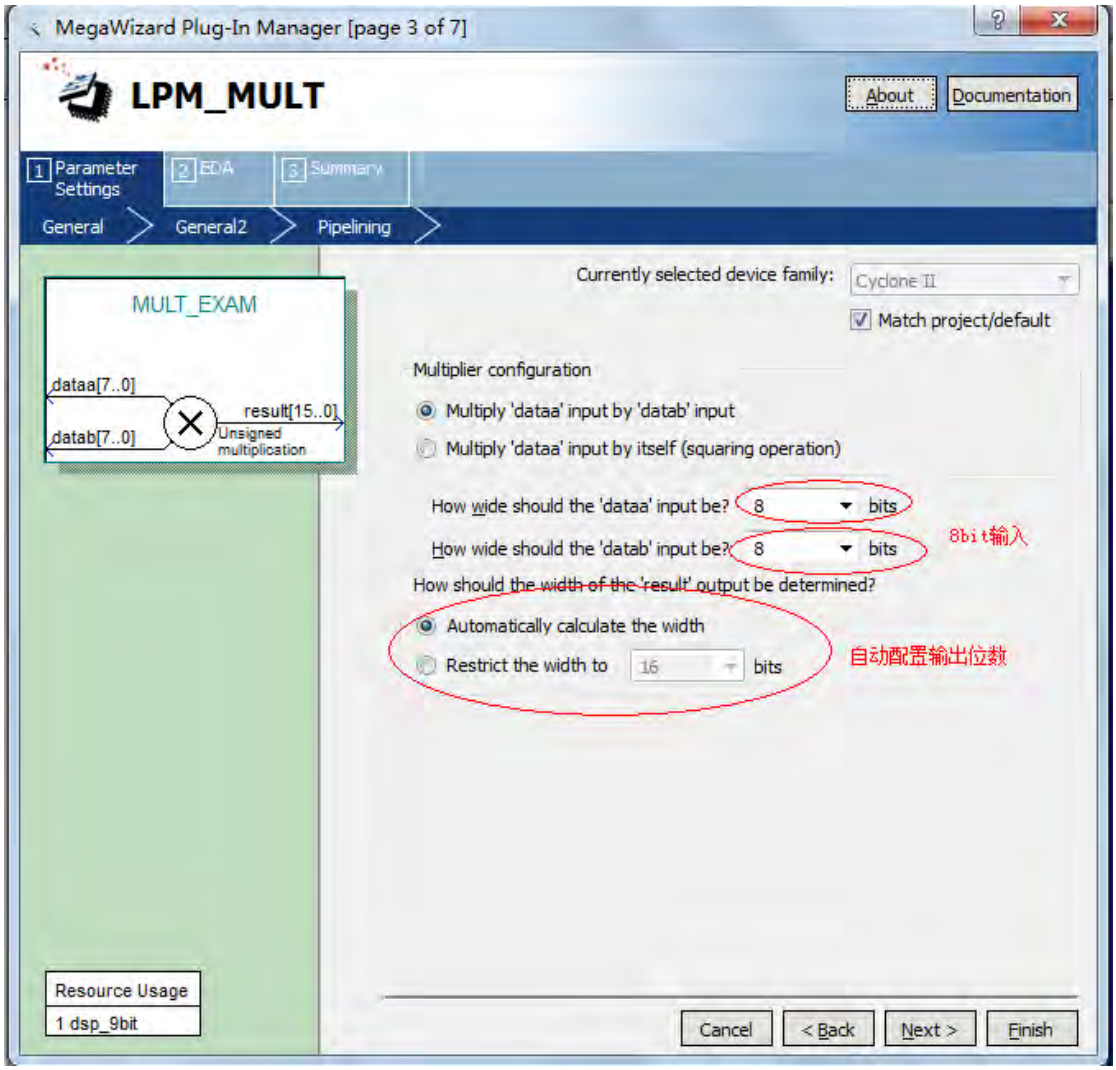



图 4

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。

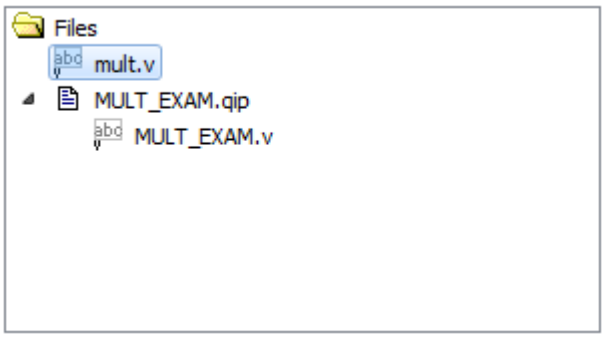
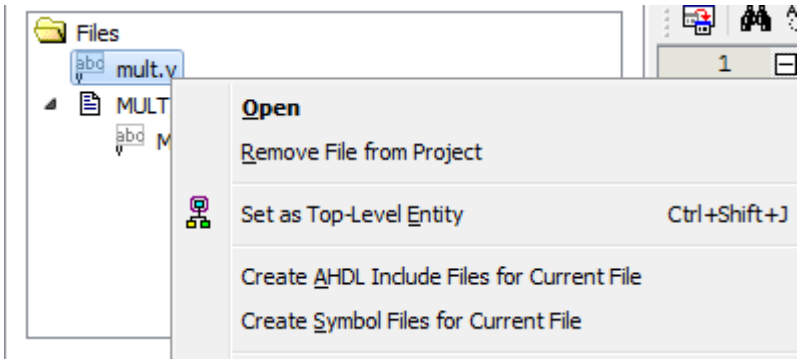



图 5

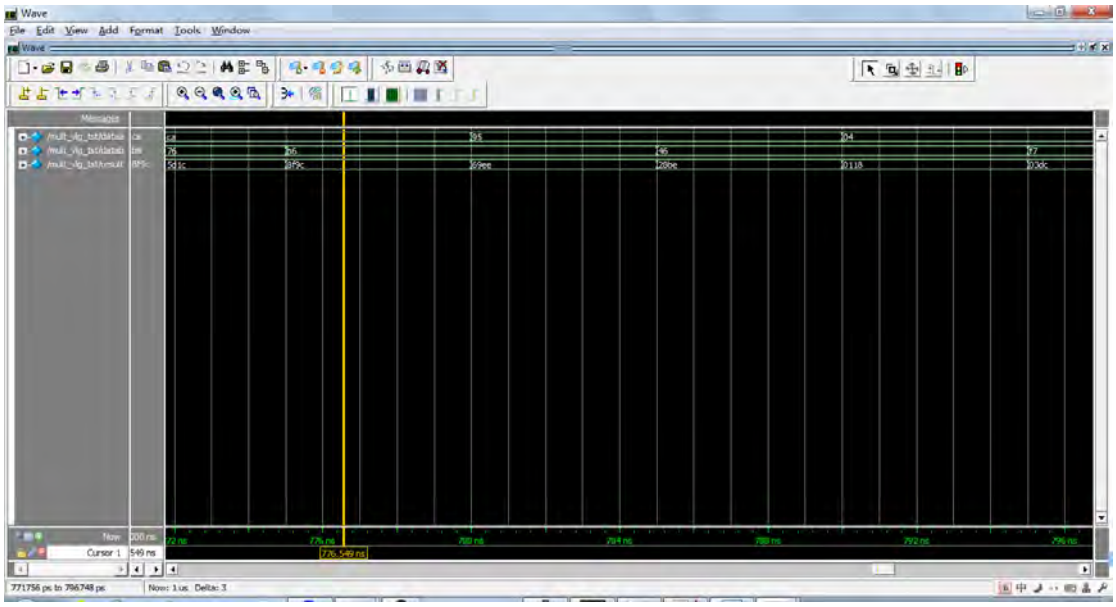
⑥ 右键点击 mult.V




选择 Set as Top-Level Entity

⑦ 再点击


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：



# 第十二章进阶实验

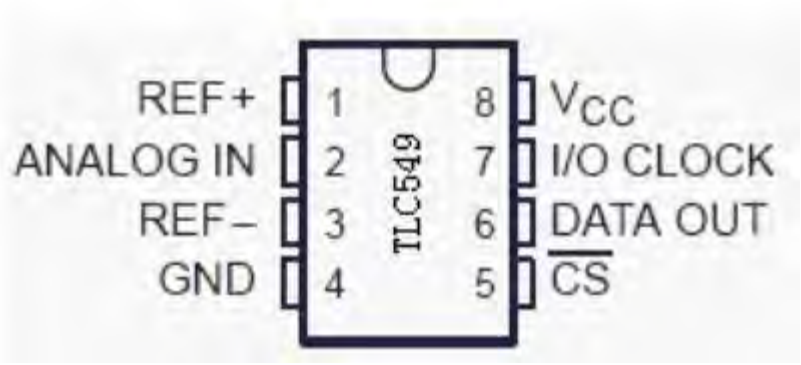
## 12.1 AD_TLC549 采集模拟信号

### 1. 了解 AD_TLC549 芯片

#### 1.1 芯片介绍

TLC549 是 TI 公司生产的一种低价位、高性能的 8 位 A/D 转换器，它以 8 位开关电容逐次逼近的方法实现 A/D 转换，其转换速度小于 17us，最大转换速率为 40000HZ，4MHZ 典型内部系统时钟，电源为 3V 至 6V。它能方便地采用三线串行接口方式与各种微处理器连接，构成各种廉价的测控应用系统。

### 1.2 TLC549 引脚图及各引脚功能

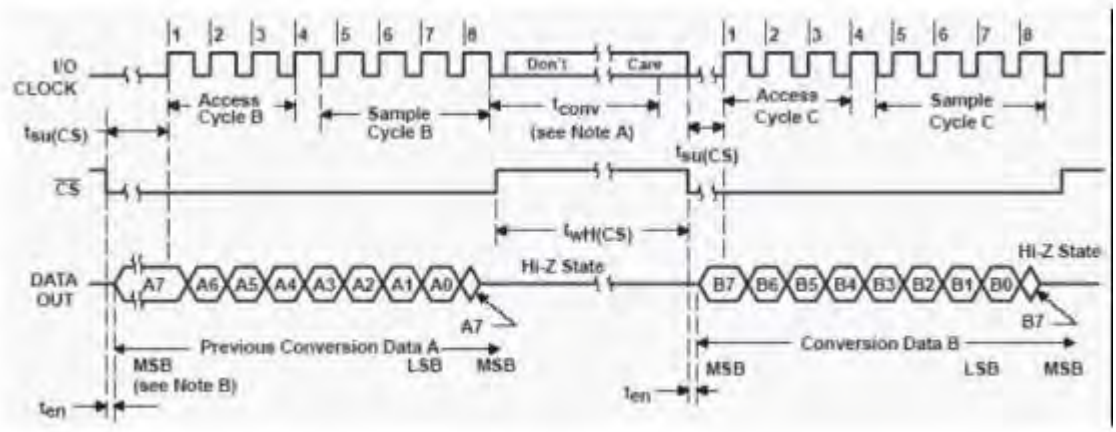



TLC549 引脚图

REF+：正基准电压输入 2.5V≤REF+≤Vcc+0.1。
REF－：负基准电压输入端，-0.1V≤REF-≤2.5V。且要求：（REF+）－（REF-）≥1V。
VCC：系统电源 3V≤Vcc≤6V。
GND：接地端。
/CS：芯片选择输入端，要求输入高电平 VIN≥2V，输入低电平 VIN≤0.8V。
DATA OUT：转换结果数据串行输出端，与 TTL 电平兼容，输出时高位在前，低位在后。
ANALOGIN：模拟信号输入端，0≤ANALOGIN≤Vcc，当 ANALOGIN≥REF+电压时，转换结果为全“1”(0FFH)，ANALOGIN≤REF-电压时，转换结果为全“0”(00H)。
I/O CLOCK：外接输入/输出时钟输入端，同于同步芯片的输入输出操作，无需与芯片内部系统时钟同步。

## 1.3TLC549 工作时序

当/CS 变为低电平后， TLC549 芯片被选中，同时前次转换结果的最高有效位 MSB （A7）自 DATA OUT 端输出，接着要求自 I/O CLOCK 端输入 8 个外部时钟信号，前 7 个 I/O CLOCK 信号的作用，是配合 TLC549 输出前次转换结果的 A6-A0 位，并为本次转换做准备：在第 4 个 I/O CLOCK 信号由高至低的跳变之后，片内采样/保持电路对输入模拟量采样开始，第 8 个 I/O CLOCK 信号的下降沿使片内采样/保持电路进入保持状态并启动 A/D 开始转换。转换时间为 36 个系统时钟周期，最大为 17us。直到 A/D 转换完成前的这段时间内，TLC549 的控制逻辑要求：或者/CS 保持高电平，或者 I/O CLOCK 时钟端保持 36 个系统时钟周期的低电平。由此可见，在自 TLC549 的 I/O CLOCK 端输入 8 个外部时钟信号期间需要完成以下工作：读入前次 A/D 转换结果；对本次转换的输入模拟信号采样并保持；启动本次 A/D 转换开始。




TLC549 工作时序

## 2.TLC549 在开发板上的应用

在我们的开发板上，已经有 TLC549 这款芯片。我们采用 TLC549 来完成一个模拟电压检测的实验。

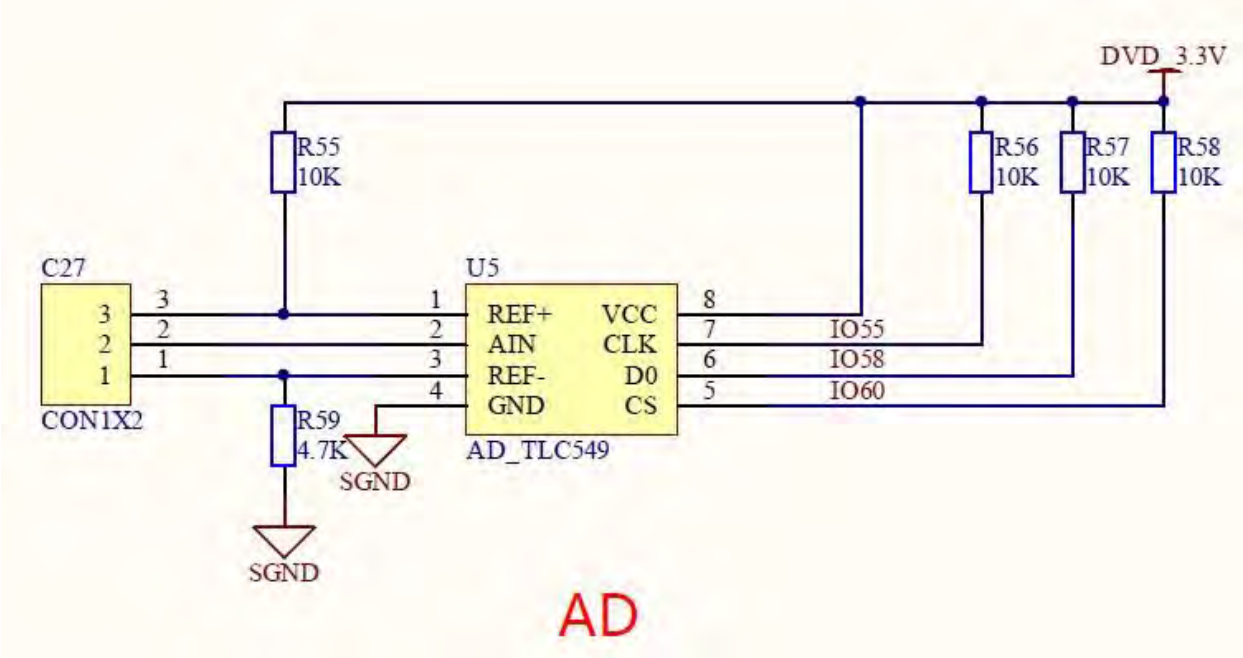
### 2.1 实验原理

本实验中，我们使用 TLC549 芯片检测模拟电压，为方便检测，我们在实验中，使用了数码管作为我们的显示模块，将检测到的电压显示到数码管上，与实际的电压进行比较。开发板上，芯片 VCC=3.3V，VREF+=3.3V，VREF-=0。

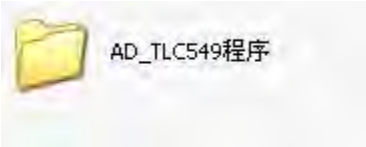
### 2.2 实验照片

在实验中，我们利用 TLC549 检测 3.3V 的电压，实验结果如下，在误差允许范围内，结果符合我们的要求。


3.TLC549 在开发板上的原理图



4.配带例程的文件夹名称和工程名称




在实验中，根据所配置的管脚，可使用杜邦线将 ANALOGIN 与模拟电压的输出口连接。

5. verilog 例程

```
module ADC_TLC549
(
clk,      //系统 50MHZ 时钟
ioclk,    //AD TLC549 的时钟
data,     //AD TLC549 的数据口
cs,       //AD TLC549 的片选择
segcs,    //数码管的为选择
segdata   //数码管的 7 段码
);

input clk;
input data;
output cs;
output ioclk;
output[7:0]segdata;
output[3:0]segcs;

reg cs,ioclk,clk1k,clk1ms;
reg[15:0] count;
reg[24:0]count1ms;
reg[3:0] cnt;
reg[2:0]number;
reg[1:0] state;
reg[3:0] segcs;
reg ledcs;
reg [7:0]segdata;
reg[7:0]dataout;
reg[16:0]tenvalue;
```

```
parameter sample=2'b00,
          display=2'b01;

/**********产生 100k 的采集时钟信号*********/
always@(posedge clk)
  begin
    if(count<=250)
      count<=count+1'b1;
    else
      begin
      count<=0;
      ioclk<=~ioclk;
      end
 end

/*******产生周期为 1ms 即 1kHz 的信号*********/
always@(posedge clk)
  begin
    if(count1ms>25'd25000)
      begin
      clk1ms<=~clk1ms;
      count1ms<=0;
      end
    else
      count1ms<=count1ms+1;
  end

/*********AD 采样程序**************/
always@(negedge ioclk)
   begin
     case(state)
      sample:
       begin
        cs<=0;
        dataout[7:0]<={dataout[6:0],data};
        if(cnt>4'd7)
           begin
           cnt<=0;
           state<=display;
           end
        else
           begin
           cnt<=cnt+1;
```

```
            state<=sample;
            end
         end
        display:
           begin
            cs<=1;//关 AD 片选
            tenvalue<=(tendata((dataout>>4)&8'b0000_1111)*16+
tendata(dataout&8'b0000_1111))*129;//
  //得到采集的数据
            state<=sample;
           end
      default: state<=display;
    endcase
  end

/***********2 进制转十进制函数*************/
function[7:0] tendata;//返回一个 4 位的数字
input[7:0]   datain;
begin
 case(datain)
    4'b00000000: tendata=4'd0;//0
    4'b00000001: tendata=4'd1;//1
    4'b00000010: tendata=4'd2;//2
    4'b00000011: tendata=4'd3;//3
    4'b00000100: tendata=4'd4;//4
    4'b00000101: tendata=4'd5;//5
    4'b00000110: tendata=4'd6;//6
    4'b00000111: tendata=4'd7;//7
    4'b00001000: tendata=4'd8;//8
    4'b00001001: tendata=4'd9;//9
    4'b00001010: tendata=4'd10;//
    4'b00001011: tendata=4'd11;//
    4'b00001100: tendata=4'd12;
    4'b00001101: tendata=4'd13;
    4'b00001110: tendata=4'd14;
    4'b00001111: tendata=4'd15;
   default:tendata=4'bzzzz_zzzz;
 endcase
end
endfunction

/*********十进制转 LED 段选函数*********/
function[7:0] leddata;//返回一个 8 位的数字
```

```
input[3:0]     datain;
begin
  case(datain)
    4'd0: leddata=8'b11000000;//0
    4'd1: leddata=8'b11111001;//1
    4'd2: leddata=8'b10100100;//2
    4'd3: leddata=8'b10110000;//3
    4'd4: leddata=8'b10011001;//4
    4'd5: leddata=8'b10010010;//5
    4'd6: leddata=8'b10000010;//6
    4'd7: leddata=8'b11111000;//7
    4'd8: leddata=8'b10000000;//8
    4'd9: leddata=8'b10010000;//9
    4'd10: leddata=8'b10111111;//-
    4'd11: leddata=8'b01111111;//.
    default:leddata=8'bzzzz_zzzz;
   endcase
end
endfunction

/********数码管扫描函数*************/
always@(posedge clk1ms)
 begin
    if(number==5) number<=0;
    else
    begin
        number<=number+1;
        case(number)
         4'd0:
         begin
         segdata<=leddata((tenvalue/10)%10);//个位
         segcs<=4'b1110;
         end
         4'd1:
         begin
         segdata<=leddata((tenvalue/100)%10);//十位
         segcs<=4'b1101;
         end
         4'd2:
         begin
         segdata<=leddata((tenvalue/1000)%10); //百位
         segcs<=4'b1011;
         end
```

```
            4'd3:
            begin
            segdata<=leddata(tenvalue/10000);//千位
            segcs<=4'b0111;
            end
            4'd4:
            begin
            segdata<=leddata(4'd11);//.    显示小数点
            segcs<=4'b0111;
            end
          endcase
    end
end

endmodule
```

**注：具体内容可看实际文件中的代码。**

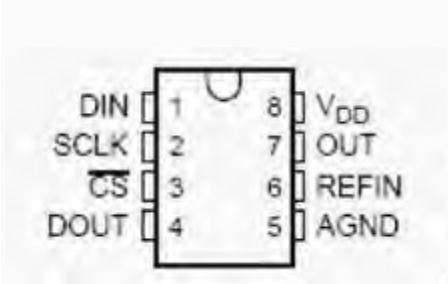
## 12.2 DA_TLC5615 驱动输出

### 1. 了解 DA_TLC5615 芯片

#### 1.1 芯片简介

TLC5615 为美国德州仪器公司 1999 年推出的产品，是具有串行接口的数模转换器，其输出为电压型，最大输出电压是基准电压值的两倍。带有上电复位功能，即把 DAC 寄存器复位至全零。性能比早期电流型输出的 DAC 要好。只需要通过 3 根串行总线就可以完成 10 位数据的串行输入，易于和工业标准的微处理器或微控制器(单片机) 接口，适用于电池供电的测试仪表、移动电话,也适用于数字失调与增益调整以及工业控制场合。

#### 1.2 TLC5615 器件的引脚图及各引脚功能



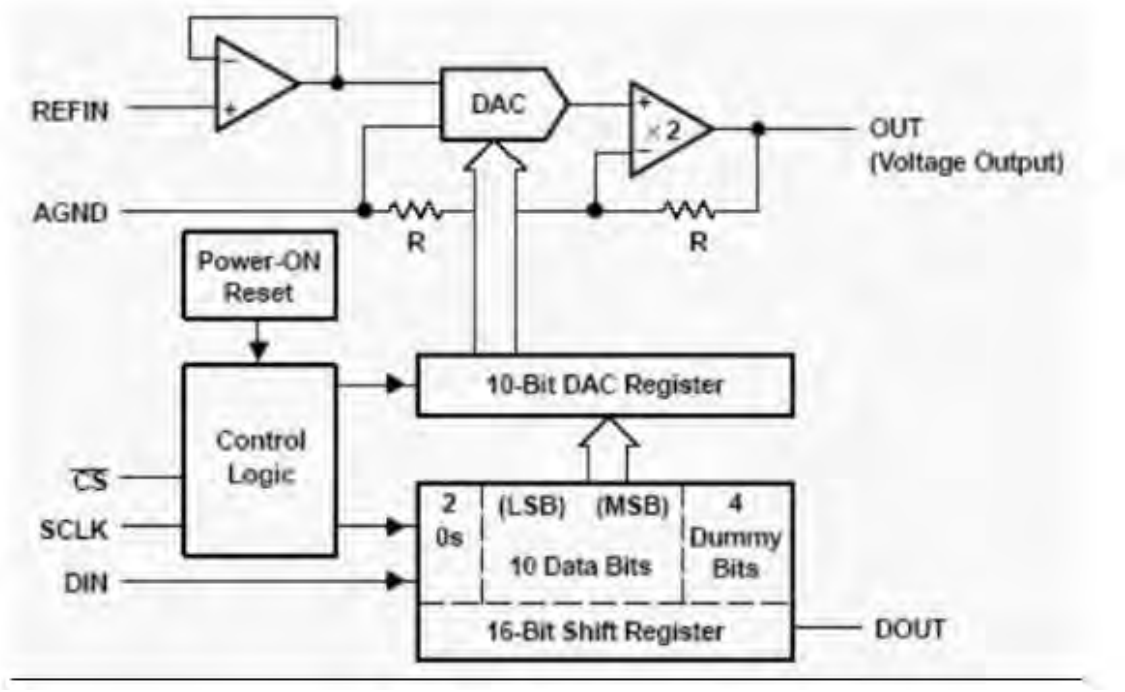
TLC5615 引脚图

DIN： 串行数据输入端； SCLK： 串行时钟输入端；
/CS： 芯片选用通端，低电平有效；
DOUT： 用于级联时的串行数据输出端；
AGND： 模拟地；
REFIN：基准电压输入端，2V～ (VDD - 2)，通常取 2.048V；
OUT： DAC 模拟电压输出端；
VDD： 正电源端，4.5～5.5V ，通常取 5V。

## 1.3 功能框图

TLC5615 的内部功能框图如下图所示,它主要由以下几部分组成:
1、10 位 DAC 电路;
2、一个 16 位移位寄存器，接受串行移入的二进制数,并且有一个级联的数据输出端 DOUT ；
3、并行输入输出的 10 位 DAC 寄存器，为 10 位 DAC 电路提供待转换的二进制数据;
4、电压跟随器为参考电压端 REFIN 提供很高的输入阻抗,大约 10MΩ ;
5、×2 电路提供最大值为 2 倍于 REFIN 的输出;
6、上电复位电路和控制电路。

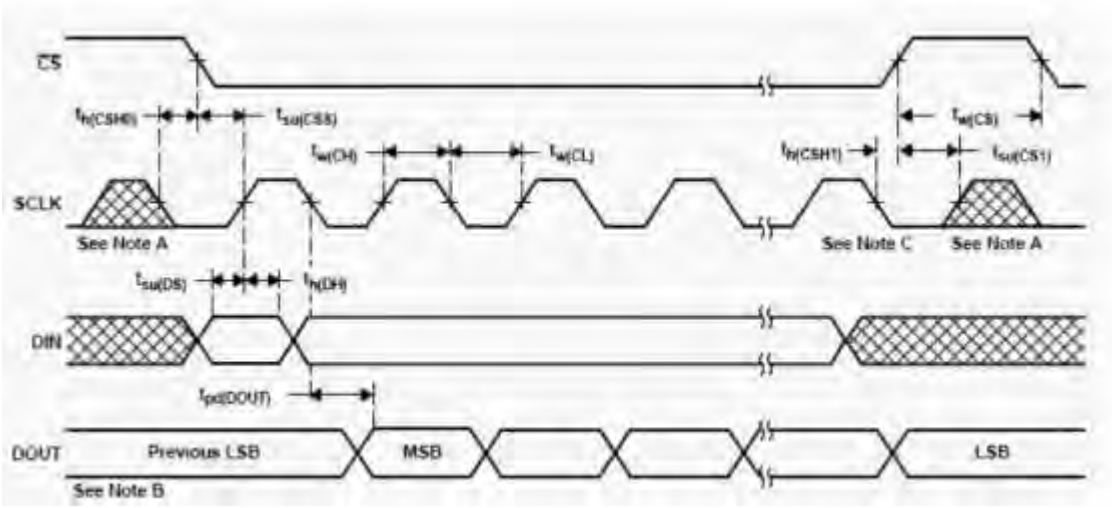
两种工作方式:（A）从上图可以看出，16 位移位寄存器分为高 4 位虚拟位、低两位填充位以及 10 位有效位。在单片 TLC5615 工作时,只需要向 16 位移位寄存器按先后输入 10 位有效位和低 2 位填充位， 2 位填充位数据任意,这是第一种方式,即 12 位数据序列。（B）第二种方式为级联方式，即 16 位数据列,可以将本片的 DOUT 接到下一片的 DIN，需要向 16 位移位寄存器按先后输入高 4 位虚拟位、10 位有效位和低 2 位填充位，由于增加了高 4 位虚拟位，所以需要 16 个时钟脉冲。




TLC5615 功能框图

### 1.4 TLC5615 的工作时序

TLC5615 工作时序如下图所示。可以看出,只有当片选 CS 为低电平时, 串行输入数据才能被移入 16 位移位寄存器。当 CS 为低电平时,在每一个 SCLK 时钟的上升沿将 DIN 的一位数据移入 16 位移寄存器。注意, 二进制最高有效位被导前移入。接着,CS 的上升沿将 16 位移位寄存器的 10 位有效数据锁存于 10 位 DAC 寄存器, 供 DAC 电路进行转换; 当片选 CS 为高电平时,串行输入数据不能被移入 16 位移位寄存器。注意, CS 的上升和下降都必须发生在 SCLK 为低电平期间。




TLC5615 工作时序

## 2. TLC5615 在开发板上的应用

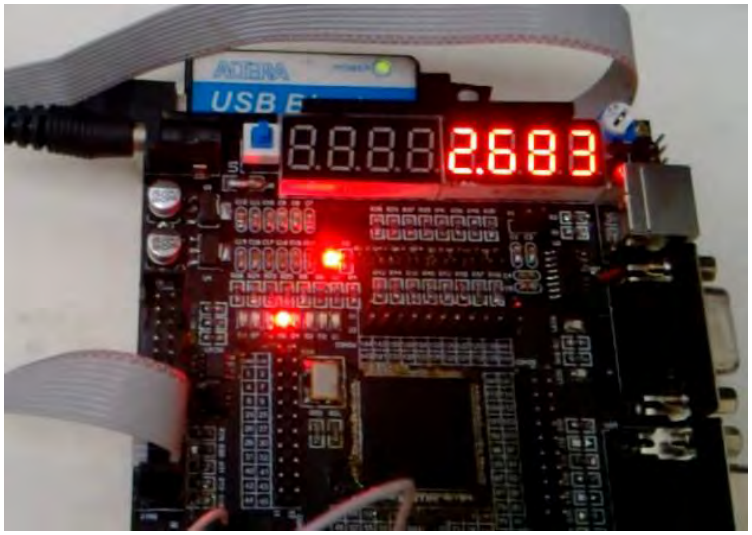
在我们的开发板上，已经有 TLC5615 这款芯片。我们采用 TLC5615 来完成一个模拟电压输出的实验。

### 2.1 工作原理

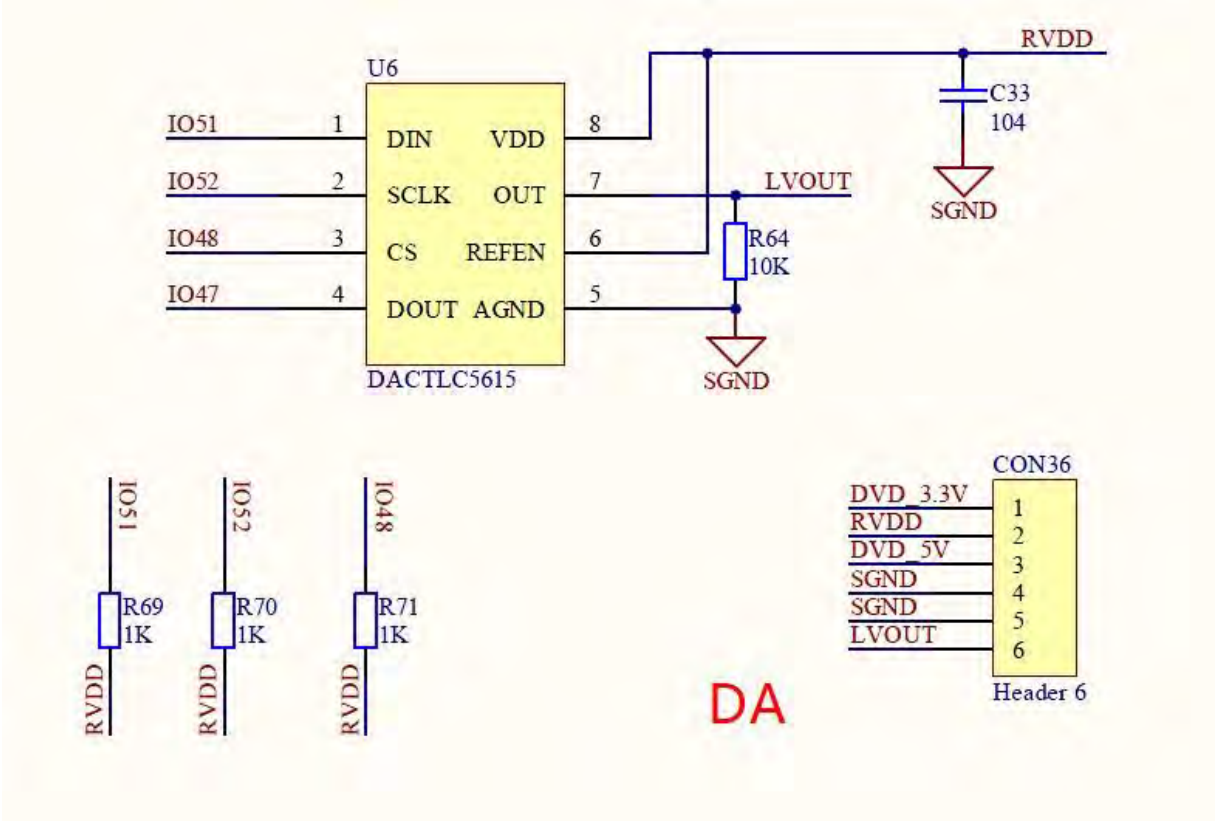
在我们的实验中，我们采用 TLC5615 的 12 位工作模式。其中 10 位是有效位，2 位是填充位，填充位补 0 即可。芯片在片选/CS 为低电平时工作。在每个 SCLK 上升沿将 DIN 的一位数据移入寄存器中。/CS 的上升和下降都必须发生在 SCLK 为低电平的时候。在实验中，我们可以根据需要输出的电压，设置一个 10 位二进制数，计算公式：Vout=VREF*(N/1024) N 为 10 位二进制码。开发板 DA 芯片 VDD=5V,VREF=3.3V

### 2.2 实验照片

我们实验中，我们计划输出 2.7V，在误差允许范围内，实验结果符合我们的要求。




3. TLC5615 在开发板上的原理图




4. 配带例程的文件夹名称和工程名称


为了方便大家能观察 TLC5615 输出的电压，我们附加一个例程，该例程添加了 AD_TLC549 芯片，只要用杜邦线把相应 IO 连接，通过数码管就可以显示输出的电压。

5. verilog 例程

```
module TLC5615
        (clk,                          //内部时钟
         sclk,                         //TLC5615 sclk 时钟脚
         din,                          //TLC5615 din 数据脚
         cs);                          //TLC5615 cs 片选
input clk;
output din;
output cs;
output sclk;

reg din;
reg cs;
reg sclk;
reg[3:0] count1,count2,count3;
reg[9:0] din_reg;                      //10 位数据寄存器

initial                                //初始化
begin
  cs=1;
  din=0;
  count1=0;
  count2=0;
  count3=0;
  din_reg=10'b11_1111_1111;                          //实验者可以根据
//需要修改 10 为二进制码
end

/*** sclk 的频率设置为 2.5MHz ***/
always@(posedge clk)
begin
  if(count3==4'd9)
  begin
     sclk<=~sclk;
     count3<=0;
  end
  else
    count3<=count3+1;
end
```

```
/*** TLC5615 cs 片选 ***/
always@(negedge sclk)
begin
  if(count1>=4'd12&&count1<4'd15)
  begin
     cs<=1;                                          //拉高片选
     count1<=count1+4'd1;
end
  else if(count1==4'd15)
  begin
    count1<=0;
  end
  else
  begin
    cs<=0;                                           //拉低片选
    count1<=count1+4'd1;
  end
end

/*** 10 位二进制码进行数模转换（采用 12 位传送方式即 10 位有效位+2
位填充位） ***/
always@(posedge sclk)
begin
  if(cs==0)
  begin
  case(count2)
  4'd0:din<=1'd0;                                    //无效位

  4'd1:begin din<=din_reg[9];end                     //10 位有效位
  4'd2:begin din<=din_reg[8];end
  4'd3:begin din<=din_reg[7];end
  4'd4:begin din<=din_reg[6];end
  4'd5:begin din<=din_reg[5];end
  4'd6:begin din<=din_reg[4];end
  4'd7:begin din<=din_reg[3];end
  4'd8:begin din<=din_reg[2];end
  4'd9:begin din<=din_reg[1];end
  4'd10:begin din<=din_reg[0];end

  4'd11:begin din<=1'd0;end                          //填充位 补0即可
  4'd12:begin din<=1'd0;end                          //填充位 补0即可
```

```
  4'd13:din<=1'd0;                                    //无效位
  4'd14:din<=1'd0;                                    //无效位
  4'd15:din<=1'd0;                                    //无效位
  default:begin count2<=0;din<=0;end
  endcase
  end
  if(count2==4'd15)
   count2<=0;
  else
  count2<=count2+4'd1;
  end
endmodule
```

**注：上面程序中，无效位只是设计者自己添加，操作者可以根据需要去掉，去掉时必须配合 SCLK 和/CS。具体内容可看实际文件中的代码。**

## 12.3 PS2 接口驱动

## PS/2 键盘解码实验

### 1. 了解 PS/2 接口

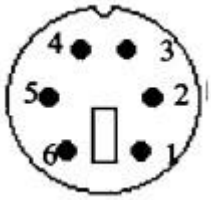
#### 1.1 什么是 PS/2

PS/2 是在较早电脑上常见的接口之一，用于鼠标、键盘等设备。一般情况下，PS/2 接口的鼠标为绿色，键盘为紫色。PS/2 原是“Personal System 2”的意思，“个人系统 2”，是 IBM 公司在上个世纪 80 年代推出的一种个人电脑。以前完全开放的 PC 标准让 IBM 觉得利益受了损失。所以 IBM 设计了 PS/2 这种电脑，目的是重新定义 PC 标准，不再采用开放标准的方式。在这种电脑上 IBM 使用了新型 MCA 总线，新的 OS/2 操作系统。PS/2 电脑上使用的键盘鼠标接口就是现在的 PS/2 接口。因为标准不开放，PS/2 电脑在市场中失败了。只有 PS/2 接口一直沿用到今天。

## 1.2 主用用途

PS/2 接口是输入装置接口，而不是传输接口。所以 PS2 口根本没有传输速率的概念，只有扫描速率。在 Windows 环境下，ps/2 鼠标的采样率默认为 60 次/秒，USB 鼠标的采样率为 120 次/秒。较高的采样率理论上可以提高鼠标的移动精度。 PS/2 接口设备不支持热插拔，强行带电插拔有可能烧毁主板。PS/2 可以与 USB 接口互转，即 PS/2 接口设备可以转成 USB，USB 接口设备也可以转成 PS/2。早期，在 PS/2 键盘中，包含了一个嵌入式的微控制器(如 InD1，8048 系列)，以用来执行各项的工作并减少整个系统工作中的负担。微控制器所要作的工作就是监测所有的按键，以及当按键被按下或放开时，就回报给主机。

## 1.3 PS/2 接口引脚定义



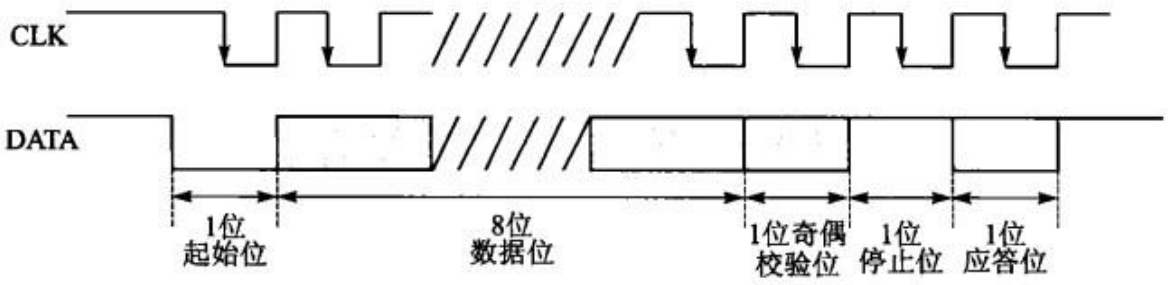

PS/2 标准接口

如上图所示，PS/2 标准使用了 6 个接口，各个接口的定义如下：
1— DATA ：数据信号
2— N.C：不连接
3— GND：数字地
4— VCC：+5V 电源
5— CLK：时钟信号
6— N.C：不连接

## 1.4 PS/2 的工作时序

数据帧格式如下图所示，起始位为低电平，停止位为高电平，应答位仅在主机对设备的通信中使用。如果数据位中 1 的个数为偶数，检验位就为 1；如果数据位中 1 的个数为奇数，检验位就为 0；总之，数据位中 1 的各个加上检验位中

1 的个数总为奇数，因此总进行奇校验。这一点与串口通讯有点类似。PC 通过 PS/2 接口与从设备通信时，总在时钟的下降沿读数据。




PS/2 的数据帧格式

## 2. PS/2 接口在开发板上的应用

在开发板上我们利用 PS/2 接口完成一个 PS/2 键盘解码的实验

### 2.1 实验说明

在实验中，我们使用到 FPGA 开发板，PS/2 键盘，还有串口调试助手（在基础实验-串口文件里可以找到）。在实验中，我们把 PS/2 键盘与 FPGA 连接，通过键盘任意按下一个按键（例程中只设定了按键字母 A-Z，操作者可以根据需要，自行添加），通过串口通讯，利用串口调试助手，显示到 PC 机上。

### 2.2 键盘编码返回值

键盘的返回值并不是和一般的 ASCII 码相对应。键盘的处理器如果发现有按键按下、释放或按住，将发送扫描码的信息包到计算机。扫描码有两种不同类型的通码和断码，当一个按键按下或按住，就发送通码；当一个键被释放，就发送断码。每个按键被分配了唯一的通码和断码，这样主机通过查找唯一的扫描码就可以测定是哪个键。每个键一整套的通断码就组成了扫描码集，有三套标准的扫描码集，分别是第一套、第二套和第三套。所有现代的键盘默认使用第二套扫描码。

虽然多数第二套通码都只有一个字节宽，但也有少数扩展按键的通码是 2 字节或 4 字节宽，这类的通码第 1 个字节总是为 8’be0.

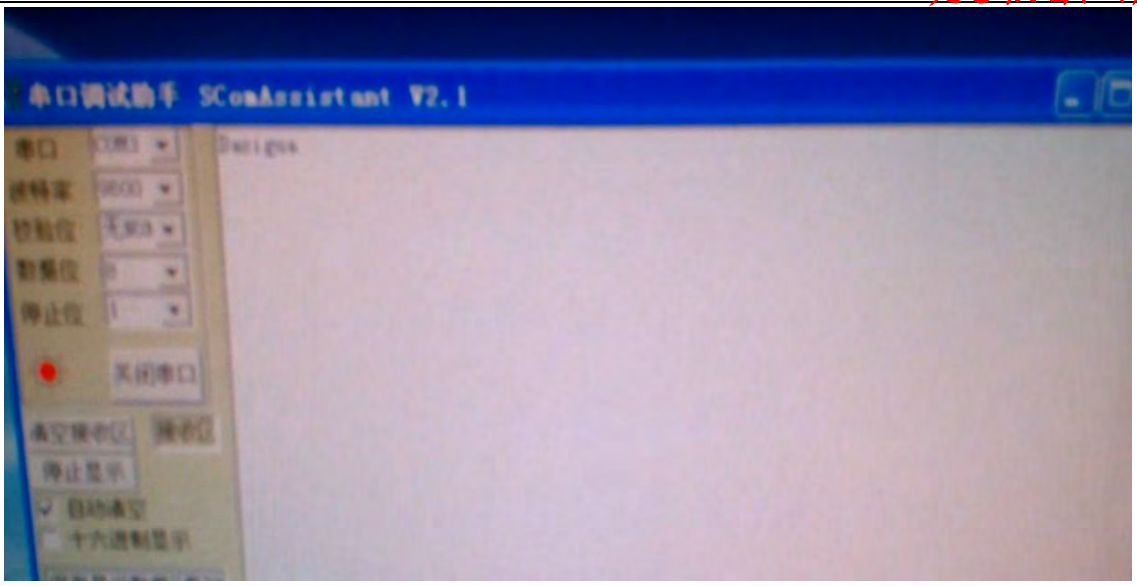
正如有键按下，通码就被发送计算机一样，只要键一释放，断码就会被发送。每个键都有它自己唯一的通码，也都有唯一的断码。在通码和断码之间存在着必然的联系。多数第二套断码有 2 字节长，它们的第 1 个字节是 8’hf0；第 2 个字节是这个键的通码。扩展按键的断码通常有 3 个字节，它们前 2 个字节是 8’he0、8’hf0，最后 1 个字节是这个按键的通码。

下面举例进行。通码和断码是以什么样的序列发送到计算机，从而使得字符 G 出现在串口调试助手字母显示框的。因为这个一个大写字母，需要一次按下 Shift 键，再按下 G 键，释放 G 键，再释放 Shift 键。与这些时间相关的扫描码如下：Shift 键的通码“8’h12”。G 键的通码“8’h34”,G 键的断码“8’hf0 8’h34”,Shift 键的断码“8’hf0 8’h12”.因此发送到计算机的数据应该是：8’h12 8’h34 8’hf0 8’h34 8’hf0 8’h12。

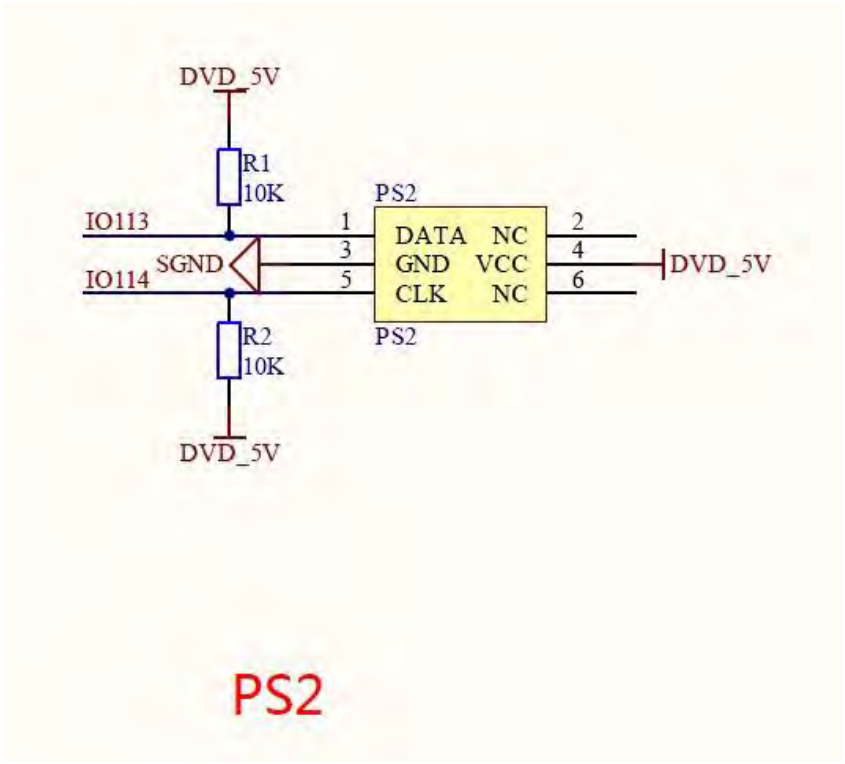
注：在我们配套的资料里，操作者可以查找到键盘每个按键的通码。


## 2.3 实验结果

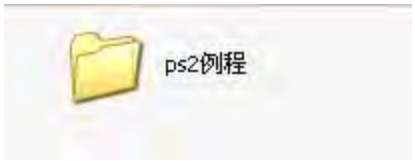
在实验中，我们键盘输入“Daxigua”，实验结果如下图所示：



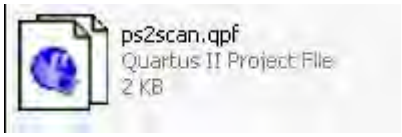

3. PS/2 接口在开发板上的原理图




4. 配带例程的文件夹名称和工程名称



1. 例程文件




2. 工程文件


3.文档资料（键盘通码）

## 5. verilog 例程

```
module ps2scan(clk,rst_n,ps2k_clk,ps2k_data,ps2_byte,ps2_state);

input clk;       //50M 时钟信号
input rst_n;     //复位信号
input ps2k_clk; //PS2 接口时钟信号
input ps2k_data;         //PS2 接口数据信号
output[7:0] ps2_byte;    // 1byte 键值，只做简单的按键扫描
output ps2_state;        //键盘当前状态，ps2_state=1 表示有键被按下

//-----------------------------------------
reg ps2k_clk_r0,ps2k_clk_r1,ps2k_clk_r2;     //ps2k_clk 状态寄存器

//wire pos_ps2k_clk;     // ps2k_clk 上升沿标志位
wire neg_ps2k_clk;  // ps2k_clk 下降沿标志位

always @ (posedge clk or negedge rst_n) begin
    if(!rst_n) begin
            ps2k_clk_r0 <= 1'b0;
            ps2k_clk_r1 <= 1'b0;
            ps2k_clk_r2 <= 1'b0;
        end
    else begin                                  //锁存状态，进行滤波
            ps2k_clk_r0 <= ps2k_clk;
            ps2k_clk_r1 <= ps2k_clk_r0;
            ps2k_clk_r2 <= ps2k_clk_r1;
        end
end
```

```
assign neg_ps2k_clk = ~ps2k_clk_r1 & ps2k_clk_r2;    //下降沿

//-------------------------------------------
reg[7:0] ps2_byte_r;          //PC 接收来自 PS2 的一个字节数据存储器
reg[7:0] temp_data;           //当前接收数据寄存器
reg[3:0] num;                 //计数寄存器

always @ (posedge clk or negedge rst_n) begin
    if(!rst_n) begin
            num <= 4'd0;
            temp_data <= 8'd0;
        end
    else if(neg_ps2k_clk) begin //检测到 ps2k_clk 的下降沿
            case (num)
                4'd0:   num <= num+1'b1;
                4'd1:   begin
                            num <= num+1'b1;
                            temp_data[0] <= ps2k_data;  //bit0
                        end
                4'd2:   begin
                            num <= num+1'b1;
                            temp_data[1] <= ps2k_data;  //bit1
                        end
                4'd3:   begin
                            num <= num+1'b1;
                            temp_data[2] <= ps2k_data;  //bit2
                        end
                4'd4:   begin
                            num <= num+1'b1;
                            temp_data[3] <= ps2k_data;  //bit3
                        end
                4'd5:   begin
                            num <= num+1'b1;
                            temp_data[4] <= ps2k_data;  //bit4
                        end
                4'd6:   begin
                            num <= num+1'b1;
                            temp_data[5] <= ps2k_data;  //bit5
                        end
                4'd7:   begin
                            num <= num+1'b1;
                            temp_data[6] <= ps2k_data;  //bit6
                        end
```

```
                4'd8:   begin
                            num <= num+1'b1;
                            temp_data[7] <= ps2k_data;  //bit7
                        end
                4'd9:   begin
                            num <= num+1'b1;    //奇偶校验位，不做处理
                        end
                4'd10: begin
                            num <= 4'd0;    // num 清零
                        end
                default: ;
                endcase
        end
end

reg key_f0;     //松键标志位，置 1 表示接收到数据 8'hf0，再接收到下一个数据后清零
reg ps2_state_r;    //键盘当前状态，ps2_state_r=1 表示有键被按下

always @ (posedge clk or negedge rst_n) begin   //接收数据的相应处理，这里只对
1byte 的键值进行处理
    if(!rst_n) begin
            key_f0 <= 1'b0;
            ps2_state_r <= 1'b0;
        end
    else if(num==4'd10) begin   //刚传送完一个字节数据
            if(temp_data == 8'hf0) key_f0 <= 1'b1;
            else begin
                    if(!key_f0) begin   //说明有键按下
                            ps2_state_r <= 1'b1;
                            ps2_byte_r <= temp_data;    //锁存当前键值
                        end
                    else begin
                            ps2_state_r <= 1'b0;
                            key_f0 <= 1'b0;
                        end
                end
        end
end

reg[7:0] ps2_asci;  //接收数据的相应 ASCII 码

always @ (ps2_byte_r) begin
    case (ps2_byte_r)       //键值转换为 ASCII 码，这里做的比较简单，只处理字母
```

```
        8'h15: ps2_asci <= 8'h51;   //Q
        8'h1d: ps2_asci <= 8'h57;   //W
        8'h24: ps2_asci <= 8'h45;   //E
        8'h2d: ps2_asci <= 8'h52;   //R
        8'h2c: ps2_asci <= 8'h54;   //T
        8'h35: ps2_asci <= 8'h59;   //Y
        8'h3c: ps2_asci <= 8'h55;   //U
        8'h43: ps2_asci <= 8'h49;   //I
        8'h44: ps2_asci <= 8'h4f;   //O
        8'h4d: ps2_asci <= 8'h50;   //P
        8'h1c: ps2_asci <= 8'h41;   //A
        8'h1b: ps2_asci <= 8'h53;   //S
        8'h23: ps2_asci <= 8'h44;   //D
        8'h2b: ps2_asci <= 8'h46;   //F
        8'h34: ps2_asci <= 8'h47;   //G
        8'h33: ps2_asci <= 8'h48;   //H
        8'h3b: ps2_asci <= 8'h4a;   //J
        8'h42: ps2_asci <= 8'h4b;   //K
        8'h4b: ps2_asci <= 8'h4c;   //L
        8'h1a: ps2_asci <= 8'h5a;   //Z
        8'h22: ps2_asci <= 8'h58;   //X
        8'h21: ps2_asci <= 8'h43;   //C
        8'h2a: ps2_asci <= 8'h56;   //V
        8'h32: ps2_asci <= 8'h42;   //B
        8'h31: ps2_asci <= 8'h4e;   //N
        8'h3a: ps2_asci <= 8'h4d;   //M
        8'h29: ps2_asci <= 8'h20;   //空格
        default: ;
        endcase
end

assign ps2_byte = ps2_asci;
assign ps2_state = ps2_state_r;

endmodule
```

## 12.4 IIC 协议

### 1. 了解 IIC

## 1.1 什么是 IIC

IIC 即 I2C，一种总线结构。IIC 即 Inter-Integrated Circuit，这种总线类型是由菲利浦半导体公司在八十年代初设计出来的，主要是用来连接整体电路(ICS) ，IIC 是一种多向控制总线，也就是说多个芯片可以连接到同一总线结构下，同时每个芯片都可以作为实施数据传输的控制源。这种方式简化了信号传输总线。例如：内存中的 SPD 信息,通过 IIC，与 BX 芯片组联系，IIC 存在于英特尔 PIIX4 结构体系中。

随着大规模集成电路技术的发展，把 CPU 和一个单独工作系统所必需的 ROM、RAM、I/O 端口、A/D、D/A 等外围电路集成在一个单片内而制成的单片机或微控制器愈来愈方便。目前，世界上许多公司生产单片机，品种很多。其中包括各种字长的 CPU，各种容量的 ROM、RAM 以及功能各异的 I/O 接口电路等等，但是，单片机的品种规格仍然有限，所以只能选用某种单片机来进行扩展。扩展的方法有两种：一种是并行总线，另一种是串行总线。由于串行总线的连线少，结构简单，往往不用专门的母板和插座而直接用导线连接各个设备。因此，采用串行线可大大简化系统的硬件设计。PHILIPS 公司早在十几年前就推出了 I2C 串行总线，利用该总线可实现多主机系统所需的裁决和高低速设备同步等功能。因此，这是一种高性能的串行总线。

飞利浦电子公司日前推出新型二选一 I2C 主选择器，可以使两个 I2C 主设备中的任何一个与共享资源连接，广泛适用于从 MP3 播放器到服务器等计算、通信和网络应用领域，从而使制造商和终端用户从中获益。PCA9541 可以使两个 I2C 主设备在互不连接的情况下与同一个从设备相连接，从而简化了设计的复杂性。此外，新产品以单器件替代了 I2C 多个主设备应用中的多个芯片，有效节省了系统成本。

## 1.2 IIC 的硬件结构

I2C 串行总线一般有两根信号线，一根是双向的数据线 SDA，另一根是时钟线 SCL。所有接到 I2C 总线设备上的串行数据 SDA 都接到总线的 SDA 上，各设备的时钟线 SCL 接到总线的 SCL 上。

为了避免总线信号的混乱，要求各设备连接到总线的输出端时必须是漏极开路（OD）输出或集电极开路（OC）输出。设备上的串行数据线 SDA 接口电路应该是双向的，输出电路用于向总线上发送数据，输入电路用于接收总线上的数据。而串行时钟线也应是双向的，作为控制总线数据传送的主机，一方面要通过 SCL 输出电路发送时钟信号，另一方面还要检测总线上的 SCL 电平，以决定什么时候发送下一个时钟脉冲电平；作为接受主机命令的从机，要按总线上的 SCL 信号发出或接收 SDA 上的信号，也可以向 SCL 线发出低电平信号以延长总线时钟信号周期。总线空闲时，因各设备都是开漏输出，上拉电阻 Rp 使 SDA 和 SCL 线都保持高电平。任一设备输出的低电平都将使相应的总线信号线变低，也就是说：各设备的 SDA 是“与”关系，SCL 也是“与”关系。

总线对设备接口电路的制造工艺和电平都没有特殊的要求（NMOS、

CMOS 都可以兼容)。在 I2C 总线上的数据传送率可高达每秒十万位，高速方式时在每秒四十万位以上。另外，总线上允许连接的设备数以其电容量不超过 400pF 为限。

总线的运行(数据传输)由主机控制。所谓主机是指启动数据的传送(发出启动信号)、发出时钟信号以及传送结束时发出停止信号的设备，通常主机都是微处理器。被主机寻访的设备称为从机。为了进行通讯，每个接到 I2C 总线的设备都有一个唯一的地址，以便于主机寻访。主机和从机的数据传送，可以由主机发送数据到从机，也可以由从机发到主机。凡是发送数据到总线的设备称为发送器，从总线上接收数据的设备被称为接受器。

I2C 总线上允许连接多个微处理器以及各种外围设备，如存储器、LED 及 LCD 驱动器、A/D 及 D/A 转换器等。为了保证数据可靠地传送，任一时刻总线只能由某一台主机控制，各微处理器应该在总线空闲时发送启动数据，为了妥善解决多台微处理器同时发送启动数据的传送（总线控制权）冲突，以及决定由哪一台微处理器控制总线的问题，I2C 总线允许连接不同传送速率的设备。多台设备之间时钟信号的同步过程称为同步化。

## 1.3 IIC 的数据传输

在 I2C 总线传输过程中，将两种特定的情况定义为开始和停止条件（见下图)：当 SCL 保持“高”时，SDA 由“高”变为“低”为开始条件；当 SCL 保持“高”且 SDA 由“低”变为“高”时为停止条件。开始和停止条件均由主控制器产生。使用硬件接口可以很容易地检测到开始和停止条件，没有这种接口的微机必须以每时钟周期至少两次对 SDA 取样，以检测这种变化。

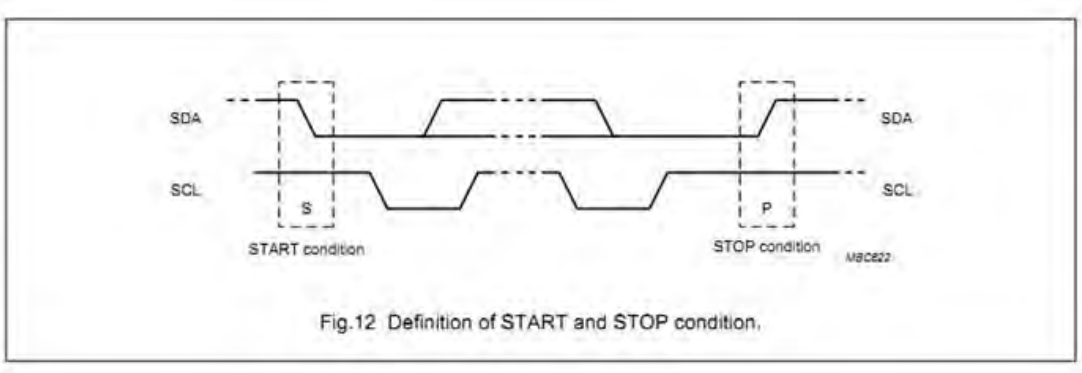



Fig.12 Definition of START and STOP condition.

SDA 线上的数据在时钟“高”期间必须是稳定的，只有当 SCL 线上的时钟信号为低时，数据线上的“高”或“低”状态才可以改变。输出到 SDA 线上的每个字节必须是 8 位,每次传输的字节不受限制,但每个字节必须要有一个应答 ACK。如果一接收器件在完成其他功能（如一内部中断）前不能接收另一数据的完整字节时，它可以保持时钟线 SCL 为低，以促使发送器进入等待状态；当接收器准备好接受数据的其它字节并释放时钟 SCL 后，数据传输继续进行。

数据传送具有应答是必须的。与应答对应的时钟脉冲由主控制器产生，发送器在应答期间必须下拉 SDA 线。当寻址的被控器件不能应答时，数据保持为高并使主控器产生停止条件而终止传输。在传输的过程中，在用到主控接收器的情况

下，主控接收器必须发出一数据结束信号给被控发送器，从而使被控发送器释放数据线，以允许主控器产生停止条件。

I2C 总线在开始条件后的首字节决定哪个被控器将被主控器选择，例外的是“通用访问”地址，它可以在所有期间寻址。当主控器输出一地址时，系统中的每一器件都将开始条件后的前 7 位地址和自己的地址进行比较。如果相同，该器件即认为自己被主控器寻址，而作为被控接收器或被控发送器则取决于 R/W 位。

## 1.4 IIC 的应用

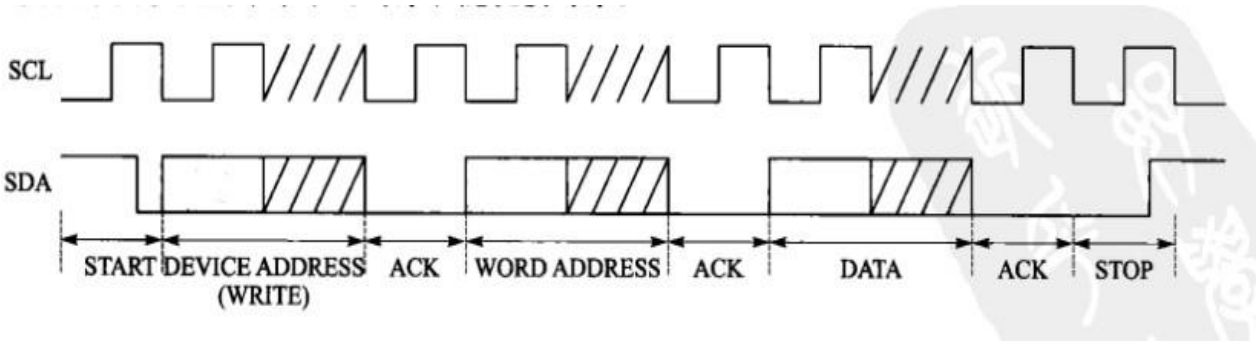
I2C 总线是各种总线中使用信号线最少，并具有自动寻址、多主机时钟同步和仲裁等功能的总线。因此，使用 I2C 总线设计计算机系统十分方便灵活，体积也小，因而在各类实际应用中得到广泛应用。

# 2. IIC 在开发板上的应用

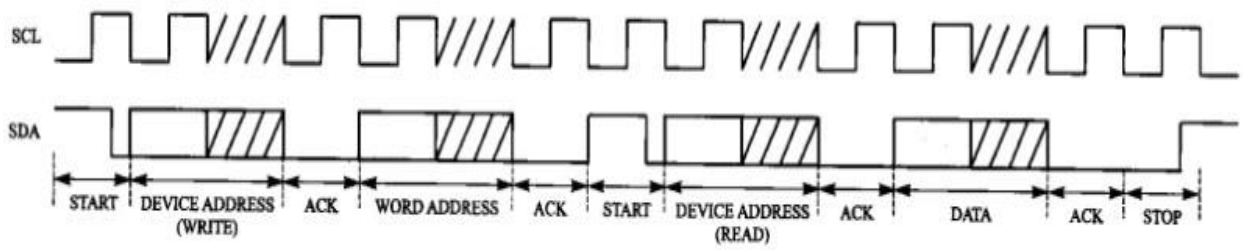
### 2.1 基于 AT24C02 的 IIC 通信协议

在使用开发板做实验之前，希望大家先对 IIC 协议有足够的了解。我们的开发板是基于 AT24C02 的 IIC 通信协议。如果对 IIC 的协议不是很熟悉的同学，可以先去看我们附带的文档资料或者上网了解 IIC。

在我们实验之前，我们还是先回顾下基于 AT24C02 的 IIC 通信协议。下图分别为单字节写时序和随机读时序。




单字节写时序




随机读时序

IIC 通信中只涉及两条信号线，即时钟线 SCL 和数据线 SDA。时钟线为高电平时均可所存数据，即时钟线上升沿和下降沿之间。当时钟线 SCL 为高电平时，如果把数据线 SDA 从高电平拉到低电平，则表示通信开始；如果把数据线 SDA 从低电平拉到高电平，则表示通信结束。器件地址(DEVICE ADDRESS)的定义如下图所示。最低位(LSB)R/W 表示读或者写状态，1 表示读，0 表示写。

| | MSB | | | | | | | LSB |
|---|---|---|---|---|---|---|---|---|
| AT24C01/2 | 1 | 0 | 1 | 0 | A2 | A1 | A0 | R/W |
| AT24C04 | 1 | 0 | 1 | 0 | A2 | A1 | P0 | R/W |
| AT24C08 | 1 | 0 | 1 | 0 | A2 | P1 | P0 | R/W |
| AT24C016 | 1 | 0 | 1 | 0 | P2 | P1 | P0 | R/W |

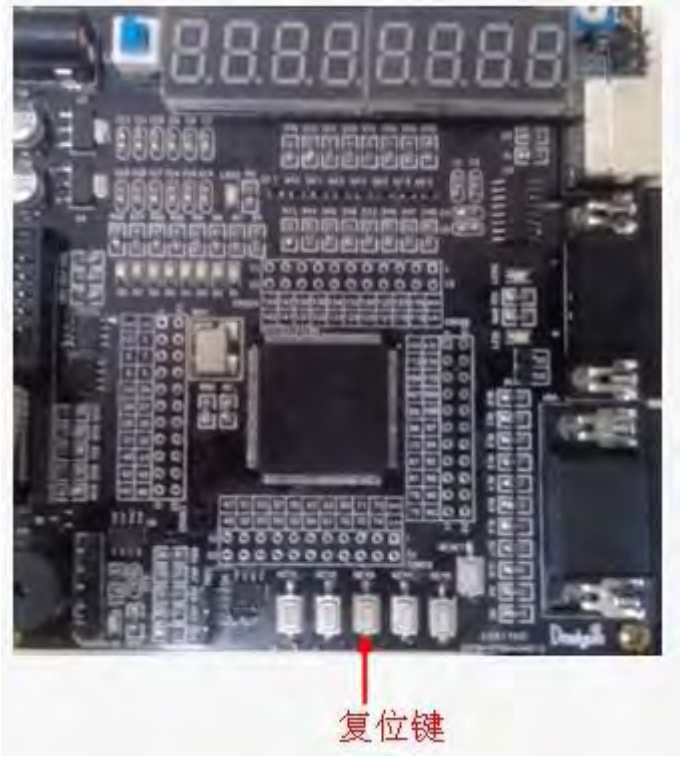
器件地址字节定义

## 2.2 实验操作

本实验中，主要学习使用基于 AT24C02 的 IIC 通信协议。实验中，我们使用到了两个按键 sw1 和 sw2，当 sw1 按下时进行的是写入操作，当 sw2 按下时进行的是读出操作。在实验中我们添加串口通讯的模块，利用串口调试助手，把我们 IIC 写入的数据读出后，显示于 PC 机上。

**步骤 1：烧入程序，代开串口调试助手。**

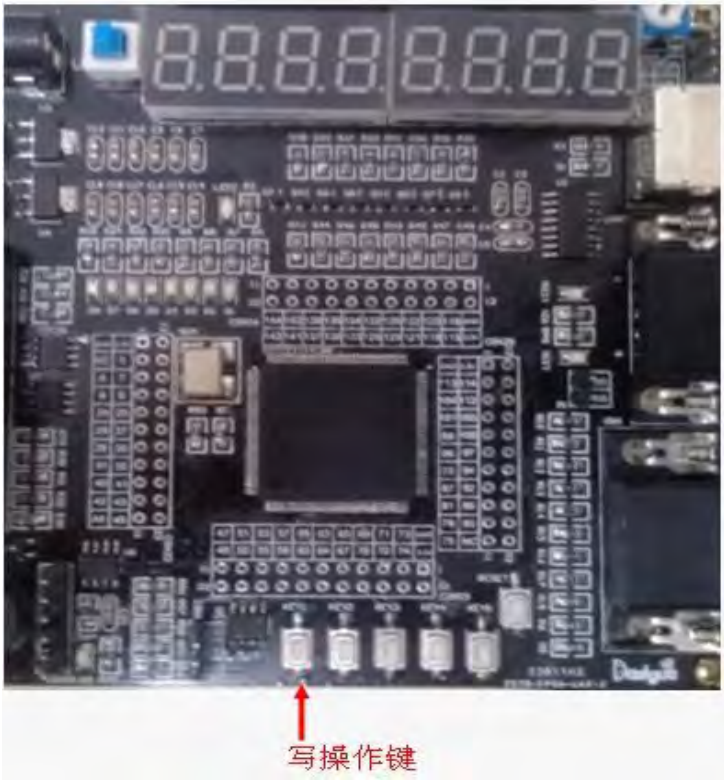
步骤 2：按下复位键，再下显示键（实质是让串口发送数据），观察 PC 机显示结果。

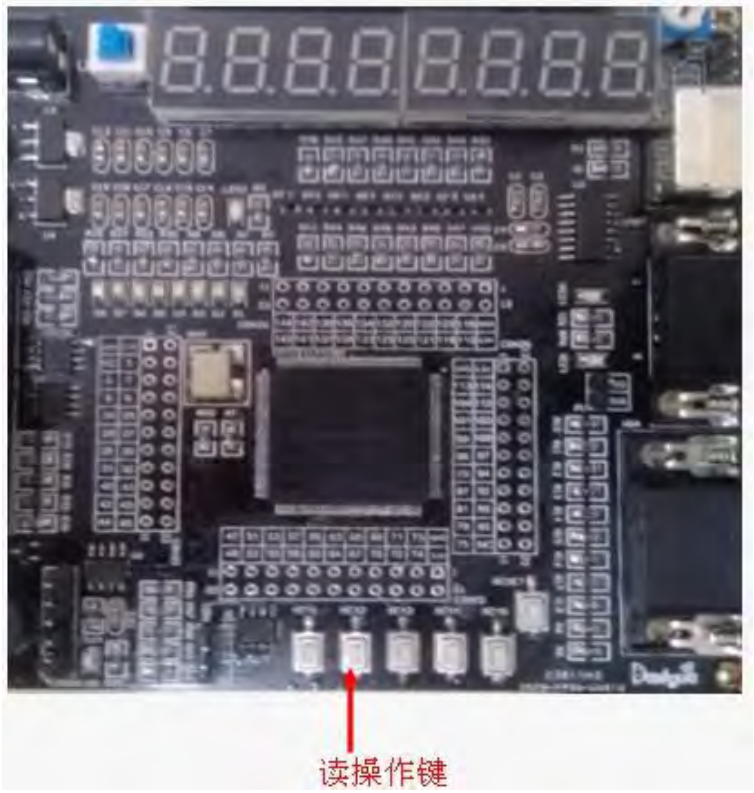




结果显示的数据都为 00，因为此时还没有进行写操作

**步骤 3：按下写操作键，再按下读操作键，观看 PC 显示结果。**

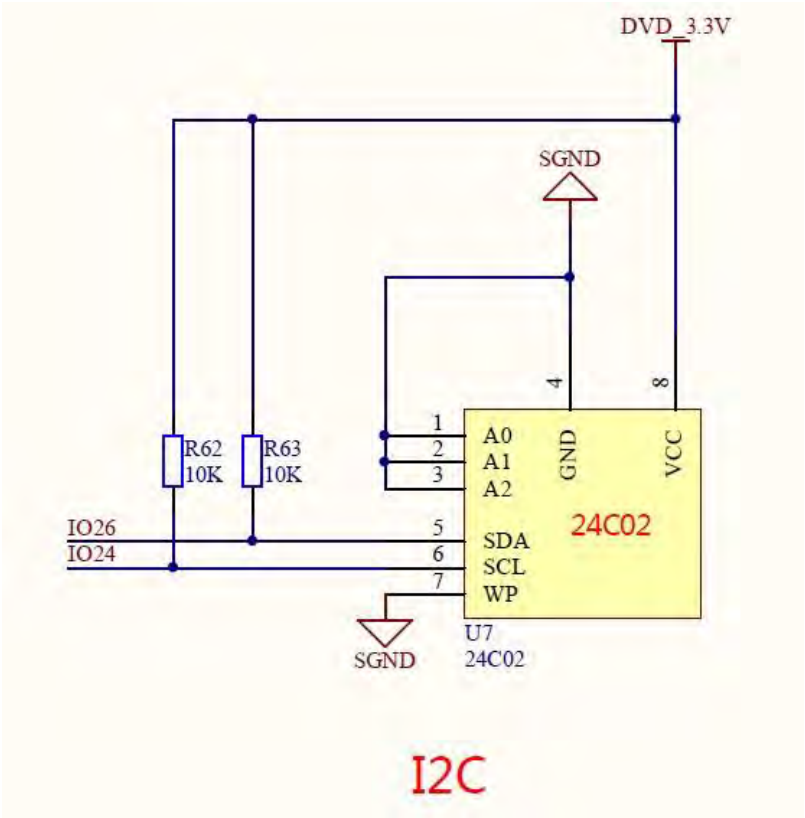






结果显示的是 0xD1 ，这与我们程序发送的数据一致。同学们可以在程序里修改发送数据再观察是否正确。

## 3. AT24C02 在开发板上的原理图




## 4. 配带例程的文件夹名称和工程名称



文档资料里有关于 IIC 的资料可供同学们参考。

## 5. verilog 例程

```
module iic_com(
              clk,rst_n,
              sw1,sw2,
              scl,sda,
              dis_data
         );

input clk;          // 50MHz
input rst_n;  //复位信号，低有效
input sw1,sw2;      //按键 1、2,(1 按下执行写入操作，2 按下执行读操作)
output scl;         // 24C02 的时钟端口
inout sda;          // 24C02 的数据端口
output[7:0] dis_data;   //数码管显示的数据


         //按键检测
reg sw1_r,sw2_r;   //键值锁存寄存器，每 20ms 检测一次键值
reg[19:0] cnt_20ms;     //20ms 计数寄存器

always @ (posedge clk or negedge rst_n)
     if(!rst_n) cnt_20ms <= 20'd0;
     else cnt_20ms <= cnt_20ms+1'b1;  //不断计数

always @ (posedge clk or negedge rst_n)
     if(!rst_n) begin
               sw1_r <= 1'b1;      //键值寄存器复位，没有键盘按下时键值都为 1
               sw2_r <= 1'b1;
          end
     else if(cnt_20ms == 20'hfffff) begin
               sw1_r <= sw1; //按键 1 值锁存
               sw2_r <= sw2; //按键 2 值锁存
          end

//---------------------------------------------
         //分频部分
reg[2:0] cnt; // cnt=0:scl 上升沿，cnt=1:scl 高电平中间，cnt=2:scl 下降沿，cnt=3:scl 低电平中
```

```
间
reg[8:0] cnt_delay;     //500 循环计数，产生 iic 所需要的时钟
reg scl_r;          //时钟脉冲寄存器

always @ (posedge clk or negedge rst_n)
    if(!rst_n) cnt_delay <= 9'd0;
    else if(cnt_delay == 9'd499) cnt_delay <= 9'd0;     //计数到 10us 为 scl 的周期，即 100KHz
    else cnt_delay <= cnt_delay+1'b1;     //时钟计数

always @ (posedge clk or negedge rst_n) begin
    if(!rst_n) cnt <= 3'd5;
    else begin
        case (cnt_delay)
            9'd124:  cnt <= 3'd1;  //cnt=1:scl 高电平中间,用于数据采样
            9'd249:  cnt <= 3'd2;  //cnt=2:scl 下降沿
            9'd374:  cnt <= 3'd3;  //cnt=3:scl 低电平中间,用于数据变化
            9'd499:  cnt <= 3'd0;  //cnt=0:scl 上升沿
            default: cnt <= 3'd5;
            endcase
        end
end


`define SCL_POS         (cnt==3'd0)        //cnt=0:scl 上升沿
`define SCL_HIG         (cnt==3'd1)        //cnt=1:scl 高电平中间,用于数据采样
`define SCL_NEG         (cnt==3'd2)        //cnt=2:scl 下降沿
`define SCL_LOW         (cnt==3'd3)        //cnt=3:scl 低电平中间,用于数据变化


always @ (posedge clk or negedge rst_n)
    if(!rst_n) scl_r <= 1'b0;
    else if(cnt==3'd0) scl_r <= 1'b1;     //scl 信号上升沿
    else if(cnt==3'd2) scl_r <= 1'b0;     //scl 信号下降沿

assign scl = scl_r;     //产生 iic 所需要的时钟
//----------------------------------------------
        //需要写入 24C02 的地址和数据

`define  DEVICE_READ       8'b1010_0001  //被寻址器件地址（读操作）
`define DEVICE_WRITE   8'b1010_0000  //被寻址器件地址（写操作）
`define  WRITE_DATA        8'b1101_0001  //写入 EEPROM 的数据
```

```
`define BYTE_ADDR       8'b0000_0011  //写入/读出 EEPROM 的地址寄存器
reg[7:0] db_r;          //在 IIC 上传送的数据寄存器
reg[7:0] read_data;     //读出 EEPROM 的数据寄存器

//----------------------------------------
        //读、写时序
parameter     IDLE      = 4'd0;
parameter     START1    = 4'd1;
parameter     ADD1      = 4'd2;
parameter     ACK1      = 4'd3;
parameter     ADD2      = 4'd4;
parameter     ACK2      = 4'd5;
parameter     START2    = 4'd6;
parameter     ADD3      = 4'd7;
parameter     ACK3 = 4'd8;
parameter     DATA      = 4'd9;
parameter     ACK4 = 4'd10;
parameter     STOP1     = 4'd11;
parameter     STOP2     = 4'd12;

reg[3:0] cstate;   //状态寄存器
reg sda_r;         //输出数据寄存器
reg sda_link; //输出数据 sda 信号 inout 方向控制位
reg[3:0] num; //


always @ (posedge clk or negedge rst_n) begin
    if(!rst_n) begin
            cstate <= IDLE;
            sda_r <= 1'b1;
            sda_link <= 1'b0;
            num <= 4'd0;
            read_data <= 8'b0000_0000;
        end
    else
        case (cstate)
            IDLE:    begin
                        sda_link <= 1'b1;           //数据线 sda 为 output
                        sda_r <= 1'b1;
                        if(!sw1_r || !sw2_r) begin  //SW1,SW2 键有一个被按下
                            db_r <= `DEVICE_WRITE; //送器件地址（写操作）
```

```
                    cstate <= START1;
                    end
                else cstate <= IDLE;    //没有任何键被按下
            end
        START1: begin
                if(`SCL_HIG) begin      //scl 为高电平期间
                    sda_link <= 1'b1;  //数据线 sda 为 output
                    sda_r <= 1'b0;          //拉低数据线 sda，产生起始位信号
                    cstate <= ADD1;
                    num <= 4'd0;        //num 计数清零
                    end
                else cstate <= START1; //等待 scl 高电平中间位置到来
            end
        ADD1:    begin
                if(`SCL_LOW) begin
                        if(num == 4'd8) begin
                                num <= 4'd0;            //num 计数清零
                                sda_r <= 1'b1;
                                sda_link <= 1'b0;       //sda 置为高阻态(input)
                                cstate <= ACK1;
                            end
                        else begin
                                cstate <= ADD1;
                                num <= num+1'b1;
                                case (num)
                                    4'd0: sda_r <= db_r[7];
                                    4'd1: sda_r <= db_r[6];
                                    4'd2: sda_r <= db_r[5];
                                    4'd3: sda_r <= db_r[4];
                                    4'd4: sda_r <= db_r[3];
                                    4'd5: sda_r <= db_r[2];
                                    4'd6: sda_r <= db_r[1];
                                    4'd7: sda_r <= db_r[0];
                                    default: ;
                                    endcase
                        //        sda_r <= db_r[4'd7-num];    //送器件地址，从高位开
始
                                end
                    end
        //        else if(`SCL_POS) db_r <= {db_r[6:0],1'b0};    //器件地址左移 1bit
                else cstate <= ADD1;
```

```
                end
        ACK1:   begin
                if(/*!sda*/`SCL_NEG) begin  //注：24C01/02/04/08/16 器件可以不考虑应答
位
                        cstate <= ADD2;     //从机响应信号
                        db_r <= `BYTE_ADDR;     // 1 地址
                    end
                else cstate <= ACK1;        //等待从机响应
                end
        ADD2:   begin
                if(`SCL_LOW) begin
                        if(num==4'd8) begin
                                num <= 4'd0;            //num 计数清零
                                sda_r <= 1'b1;
                                sda_link <= 1'b0;       //sda 置为高阻态(input)
                                cstate <= ACK2;
                            end
                        else begin
                                sda_link <= 1'b1;       //sda 作为 output
                                num <= num+1'b1;
                                case (num)
                                    4'd0: sda_r <= db_r[7];
                                    4'd1: sda_r <= db_r[6];
                                    4'd2: sda_r <= db_r[5];
                                    4'd3: sda_r <= db_r[4];
                                    4'd4: sda_r <= db_r[3];
                                    4'd5: sda_r <= db_r[2];
                                    4'd6: sda_r <= db_r[1];
                                    4'd7: sda_r <= db_r[0];
                                    default: ;
                                    endcase
                        //      sda_r <= db_r[4'd7-num];    //送 EEPROM 地址（高 bit
开始）
                                cstate <= ADD2;
                            end
                    end
        //      else if(`SCL_POS) db_r <= {db_r[6:0],1'b0};   //器件地址左移 1bit
                else cstate <= ADD2;
                end
        ACK2:   begin
                if(/*!sda*/`SCL_NEG) begin      //从机响应信号
```

```
                    if(!sw1_r) begin
                            cstate <= DATA;    //写操作
                            db_r <= `WRITE_DATA;    //写入的数据

                        end
                    else if(!sw2_r) begin
                            db_r <= `DEVICE_READ;  //送器件地址（读操作），特定地址
读需要执行该步骤以下操作
                            cstate <= START2;      //读操作
                        end
                    end
                else cstate <= ACK2;    //等待从机响应
            end
        START2: begin //读操作起始位
                if(`SCL_LOW) begin
                    sda_link <= 1'b1;  //sda 作为 output
                    sda_r <= 1'b1;          //拉高数据线 sda
                    cstate <= START2;
                    end
                else if(`SCL_HIG) begin//scl 为高电平中间
                    sda_r <= 1'b0;          //拉低数据线 sda，产生起始位信号
                    cstate <= ADD3;
                    end
                else cstate <= START2;
            end
        ADD3:    begin    //送读操作地址
                if(`SCL_LOW) begin
                        if(num==4'd8) begin
                                num <= 4'd0;            //num 计数清零
                                sda_r <= 1'b1;
                                sda_link <= 1'b0;       //sda 置为高阻态(input)
                                cstate <= ACK3;
                            end
                        else begin
                                num <= num+1'b1;
                                case (num)
                                    4'd0: sda_r <= db_r[7];
                                    4'd1: sda_r <= db_r[6];
                                    4'd2: sda_r <= db_r[5];
                                    4'd3: sda_r <= db_r[4];
                                    4'd4: sda_r <= db_r[3];
```

```
                                        4'd5: sda_r <= db_r[2];
                                        4'd6: sda_r <= db_r[1];
                                        4'd7: sda_r <= db_r[0];
                                        default: ;
                                        endcase

                                //   sda_r <= db_r[4'd7-num];    //送 EEPROM 地址（高 bit
开始）
                                        cstate <= ADD3;
                                end
                            end
                    //   else if(`SCL_POS) db_r <= {db_r[6:0],1'b0};    //器件地址左移 1bit
                        else cstate <= ADD3;
                    end
                ACK3:    begin
                        if(/*!sda*/`SCL_NEG) begin
                                cstate <= DATA;    //从机响应信号
                                sda_link <= 1'b0;
                            end
                        else cstate <= ACK3;        //等待从机响应
                    end
                DATA:    begin
                        if(!sw2_r) begin    //读操作
                                if(num<=4'd7) begin
                                    cstate <= DATA;
                                    if(`SCL_HIG) begin
                                        num <= num+1'b1;
                                        case (num)
                                            4'd0: read_data[7] <= sda;
                                            4'd1: read_data[6] <= sda;
                                            4'd2: read_data[5] <= sda;
                                            4'd3: read_data[4] <= sda;
                                            4'd4: read_data[3] <= sda;
                                            4'd5: read_data[2] <= sda;
                                            4'd6: read_data[1] <= sda;
                                            4'd7: read_data[0] <= sda;
                                            default: ;
                                            endcase

                        //                  read_data[4'd7-num] <= sda; //读数据（高 bit 开始）
                                            end
```

```
                    //                  else          if(`SCL_NEG)          read_data          <=
{read_data[6:0],read_data[7]};  //数据循环右移
                                end
                            else if((`SCL_LOW) && (num==4'd8)) begin
                                num <= 4'd0;            //num 计数清零
                                cstate <= ACK4;
                                end
                            else cstate <= DATA;
                        end
                    else if(!sw1_r) begin  //写操作
                            sda_link <= 1'b1;
                            if(num<=4'd7) begin
                                cstate <= DATA;
                                if(`SCL_LOW) begin
                                    sda_link <= 1'b1;       //数据线 sda 作为 output
                                    num <= num+1'b1;
                                    case (num)
                                        4'd0: sda_r <= db_r[7];
                                        4'd1: sda_r <= db_r[6];
                                        4'd2: sda_r <= db_r[5];
                                        4'd3: sda_r <= db_r[4];
                                        4'd4: sda_r <= db_r[3];
                                        4'd5: sda_r <= db_r[2];
                                        4'd6: sda_r <= db_r[1];
                                        4'd7: sda_r <= db_r[0];
                                        default: ;
                                        endcase

                                    //   sda_r <= db_r[4'd7-num];    //写入数据(高 bit 开始)
                                    end
            //                      else if(`SCL_POS) db_r <= {db_r[6:0],1'b0};    //写入数
据左移 1bit
                                end
                            else if((`SCL_LOW) && (num==4'd8)) begin
                                    num <= 4'd0;
                                    sda_r <= 1'b1;
                                    sda_link <= 1'b0;       //sda 置为高阻态
                                    cstate <= ACK4;
                                end
                            else cstate <= DATA;
                        end
```

```
                end
            ACK4: begin
                    if(/*!sda*/`SCL_NEG) begin
//                      sda_r <= 1'b1;
                        cstate <= STOP1;
                        end
                    else cstate <= ACK4;
                end
            STOP1:   begin
                    if(`SCL_LOW) begin
                            sda_link <= 1'b1;
                            sda_r <= 1'b0;
                            cstate <= STOP1;
                        end
                    else if(`SCL_HIG) begin
                            sda_r <= 1'b1;     //scl 为高时，sda 产生上升沿（结束信号）
                            cstate <= STOP2;
                        end
                    else cstate <= STOP1;
                end
            STOP2:   begin
                    if(`SCL_LOW) sda_r <= 1'b1;
                    else if(cnt_20ms==20'hffff0) cstate <= IDLE;
                    else cstate <= STOP2;
                end
            default: cstate <= IDLE;
            endcase
end

assign sda = sda_link ? sda_r:1'bz;
assign dis_data = read_data;

endmodule
```

**注：工程中除了 IIC 的程序外，还附带了串口通讯的程序，用于 PC 显示，这里就不列出来，同学们可以在工程里找到。**

# 12.5VGA 显示控制

## 1. VGA 简介

### 1.1 VGA 标准

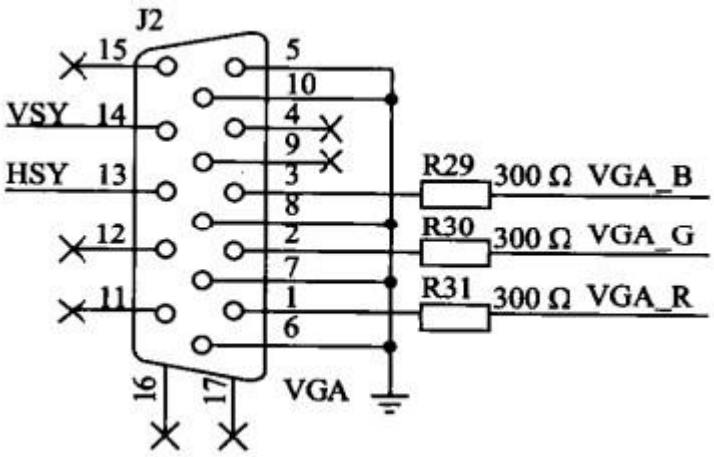
VGA（Video Graphics Array）即视频图形阵列，是 IBM 在 1987 年随 PS/2（PS/2 原是“Personal System 2”的意思，“个人系统 2”，是 IBM 公司在 1987 年推出的一种个人电脑）机推出的。PS/2 电脑上使用的键盘鼠标接口就是现在的 PS/2 接口。因为标准不开放，PS/2 电脑在市场中失败了。只有 PS/2 接口一直沿用到今天）一起推出的使用模拟信号的一种视频传输标准，在当时具有分辨率高、显示速率快、颜色丰富等优点，在彩色显示器领域得到了广泛的应用。这个标准对于现今的个人电脑市场已经十分过时。即使如此，VGA 仍然是最多制造商所共同支持的一个标准，个人电脑在加载自己的独特驱动程序之前，都必须支持 VGA 的标准。例如，微软 Windows 系列产品的开机画面仍然使用 VGA 显示模式，这也说明其在显示标准中的重要性和兼容性。

### 1.2 VGA 显示模式

VGA 最早指的是显示器 640X480 这种显示模式。VGA 技术的应用还主要基于 VGA 显示卡的计算机、笔记本等设备，而在一些既要求显示彩色高分辨率图像又没有必要使用计算机的设备上，VGA 技术的应用却很少见到。本文对嵌入式 VGA 显示的实现方法进行了研究。基于这种设计方法的嵌入式 VGA 显示系统，可以在不使用 VGA 显示卡和计算机的情况下，实现 VGA 图像的显示和控制。系统具有成本低、结构简单、应用灵活的优点，可广泛应用于超市、车站、飞机场等公共场所的广告宣传和提示信息显示，也可应用于工厂车间生产过程中的操作信息显示，还能以多媒体形式应用于日常生活。

## 1.3 VGA 接口

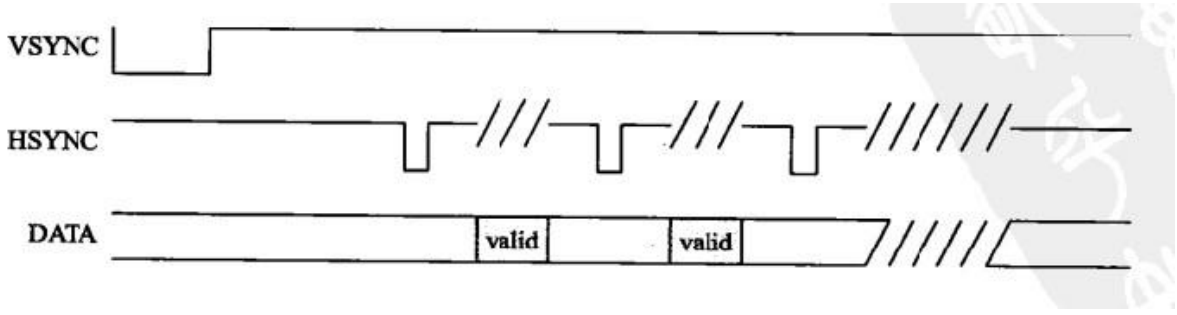
标准的 VGA 接口一共有 15 个接口(拔下任何一台 VGA 液晶或是 CRT 显示器看看就知道了)，真正用到的信号接口不多，就 5 个。HSYNC 是行同步信号，VSYNC 是场同步信号，同步信号就是为了让 VGA 显示器接收部分知道送来的数据是对应哪一行哪一列的哪一个像素点。VGA_R，VGA_G，VGA_B 是三原色信号，这三个信号接口的输入都是模拟信号(标准为 0—0.7V)，所以它们都有相应的地线需要连接。我们使用的大西瓜开发板的 VGA_R，VGA_G，VGA_B 三个信号线都有三个不同阻值的电阻，并且三原色信号接口输入的只可能是数字信号(0 或 1)，因此驱动液晶屏上显示的颜色最多能达到 512 种。一般来说，可以在 FPGA/CPLD 和 VGA 之间加一个 DAC 芯片，这样就可能实现 65536 种或者更多色彩的显示。




VGA 接口

## 1.4 VGA 接口时序

VGA 接口时序如下图所示，场同步信号 VSYNC 在每帧开始的时候产生一个固定宽度的低脉冲，行同步信号 HSYNC 在每行开始的时候产生一个固定宽度的低脉冲，数据再某些固定的行和列交汇处有效。




VGA 接口时序

# 2. VGA 接口在开发板上的应用

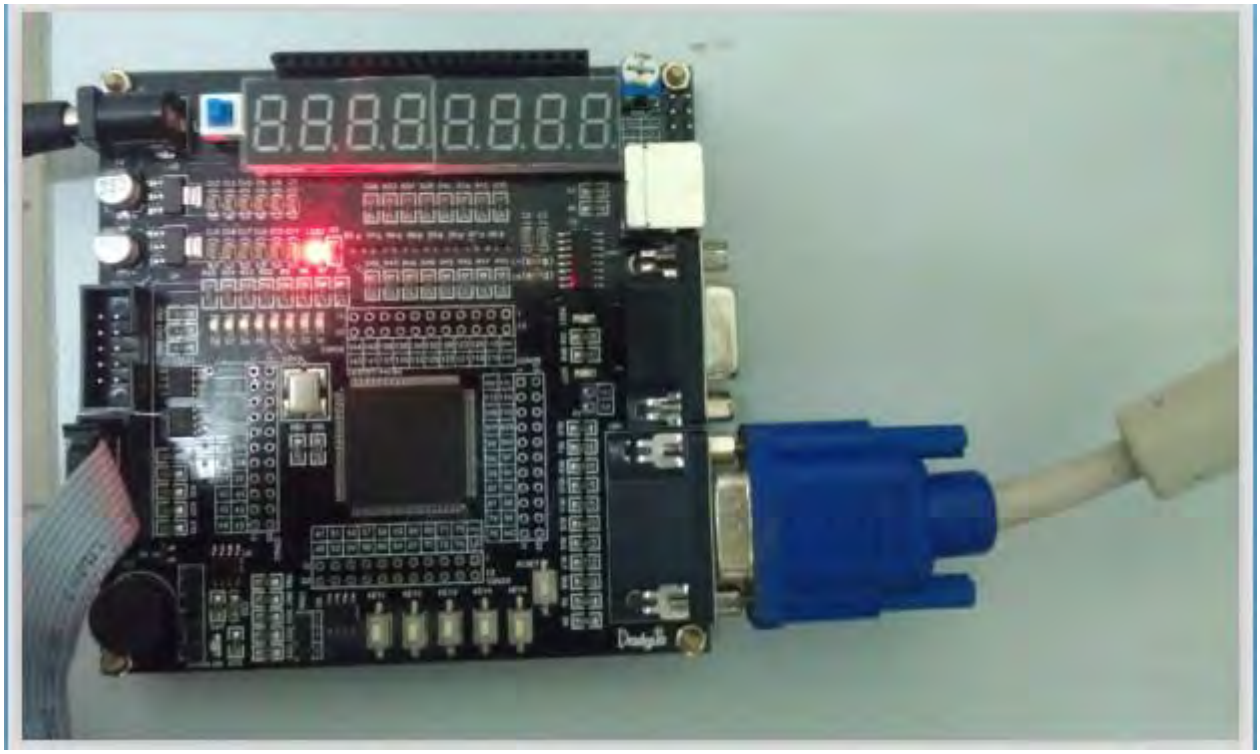
在开发板上我们使用 VGA 接口完成一个显示屏 8 种颜色循环变化的实验。

## 2.1 实验说明

在实验中，我们使用到开发板的 VGA 接口，使显示器循环地显示 8 种颜色。由于在实验中我们使用到了 9 个 IO 口，每 3 个 IO 口分别控制 VGA_R,VGA_G,VGA_B, 三原色信号为模拟信号，而开发板出来的信号为数字信号，在我们的开发板的 VGA 接口上,使用了电阻分压，所以在我们的开发板上，最多可以显示出 512 种颜色。读者可以自己查看我们开发板的原理图。

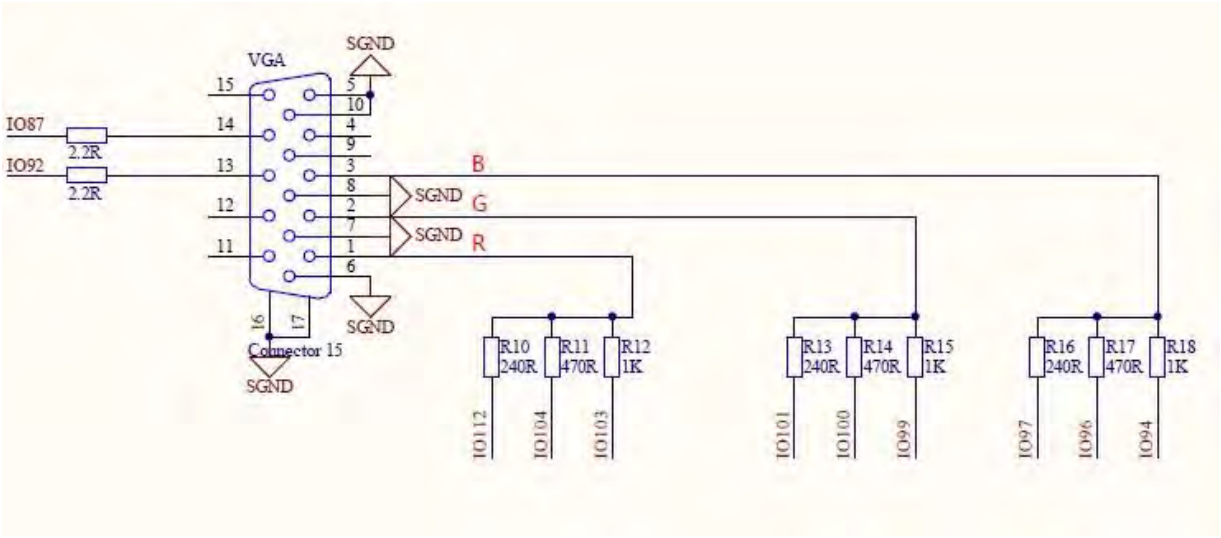
## 2.2 实验结果

（1） 接好 VGA 接口并把程序烧写进 FPGA 中



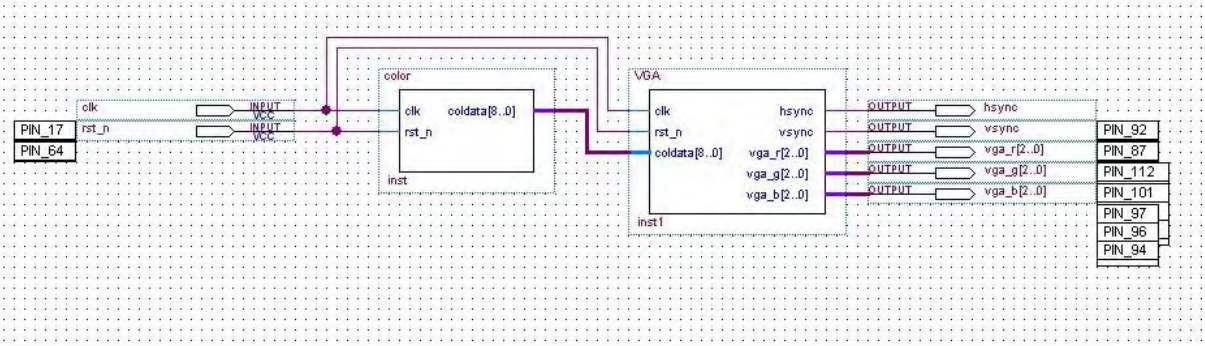
（2） 观察显示器的变化（在这我们就只提供 3 种颜色的图片）

## 3.VGA 接口在开发板上的原理图




开发板 VGA 接口原理图

## 4. verilog 例程




color.v：

```
module color(clk,rst_n,coldata);
input clk;        //系统时钟 50MHz
input rst_n;      //低电平复位
output[8:0] coldata;     //RGB 的数值

reg[24:0] count;
reg[2:0] colnum;
reg fclk;         //分频时钟
reg[8:0] coldata;

//------------------------------------------------
//进行时钟分频，fclk 的频率约为 1Hz
always@(posedge clk)
begin
  if(count==25'd24500000)
  begin
     fclk=~fclk;
     count<=0;
  end
  else
     count<=count+1;
end

//------------------------------------------------
//产生 8 组不同的 coldata 数值
always@(posedge fclk)
begin
  case(colnum)
  3'd0:begin coldata<=9'b000_000_011;colnum<=colnum+1;end
  3'd1:begin coldata<=9'b000_110_000;colnum<=colnum+1;end
```

```
 3'd2:begin coldata<=9'b110_000_000;colnum<=colnum+1;end
 3'd3:begin coldata<=9'b011_011_001;colnum<=colnum+1;end
 3'd4:begin coldata<=9'b100_100_001;colnum<=colnum+1;end
 3'd5:begin coldata<=9'b010_011_110;colnum<=colnum+1;end
 3'd6:begin coldata<=9'b101_010_100;colnum<=colnum+1;end
 3'd7:begin coldata<=9'b011_101_000;colnum<=0;end
 default:begin coldata<=9'b000_000_001;colnum<=0;end
 endcase
end

endmodule
```

**VGA.v：**

```
module VGA(clk,rst_n,hsync,vsync,vga_r,vga_g,vga_b,coldata);

input clk;          //系统时钟 50MHz
input rst_n;    //低电平复位
input[8:0]  coldata;   //需要显示的色彩，使用者可以自己在 color.v 中改变数值以获取需要的色彩，该例程实现的是 8 种色彩的循环变化
output hsync;  //行同步信号
output vsync;  //场同步信号
output[2:0] vga_r;     //红基色信号
output[2:0] vga_g;     //绿基色信号
output[2:0] vga_b;     //蓝基色信号

reg[2:0] vga_r,vga_g,vga_b;
reg[10:0] x_cnt;       //行坐标
reg[9:0] y_cnt;//列坐标

//----------------------------------------------
always @ (posedge clk or negedge rst_n)
      if(!rst_n) x_cnt <= 11'd0;
      else if(x_cnt == 11'd1039) x_cnt <= 11'd0;
      else x_cnt <= x_cnt+1'b1;

always @ (posedge clk or negedge rst_n)
      if(!rst_n) y_cnt <= 10'd0;
      else if(y_cnt == 10'd665) y_cnt <= 10'd0;
      else if(x_cnt == 11'd1039) y_cnt <= y_cnt+1'b1;

//----------------------------------------------
wire valid;       //有效显示区标志
```

```
assign valid = (x_cnt >= 11'd187) && (x_cnt < 11'd987)
&& (y_cnt >= 10'd31) && (y_cnt < 10'd631);
wire[9:0] xpos,ypos;      //有效显示区坐标
assign xpos = x_cnt-11'd187;
assign ypos = y_cnt-10'd31;
//----------------------------------------------
reg hsync_r,vsync_r;      //同步信号产生
always @ (posedge clk or negedge rst_n)
      if(!rst_n) hsync_r <= 1'b1;
      else if(x_cnt == 11'd0) hsync_r <= 1'b0; //产生 hsync 信号
      else if(x_cnt == 11'd120) hsync_r <= 1'b1;
always @ (posedge clk or negedge rst_n)
      if(!rst_n) vsync_r <= 1'b1;
      else if(y_cnt == 10'd0) vsync_r <= 1'b0; //产生 vsync 信号
      else if(y_cnt == 10'd6) vsync_r <= 1'b1;
assign hsync = hsync_r;
assign vsync = vsync_r;
//----------------------------------------------
wire dis;  //显示屏显示范围
assign dis = ( (xpos>=80) && (xpos<=720) )
                    &&  ( (ypos>=60) && (ypos<=540) );
//----------------------------------------------
//分别对 RGB 的 3 位数据进行判断
//R,G,B 控制液晶屏颜色显示
//最终显示屏显示的颜色是 RGB3 种颜色的叠加
always@(posedge clk)
begin
   case(coldata[8:6])   //R
   3'b000:begin vga_r[2]<=1'bz; vga_r[1]<=1'bz; vga_r[0]<=1'bz;end
   3'b001:begin vga_r[2]<=1'bz; vga_r[1]<=1'bz; vga_r[0]<= valid ? dis : 1'b0;end
   3'b010:begin vga_r[2]<=1'bz; vga_r[1]<=valid ? dis : 1'b0; vga_r[0]<= 1'bz;end
   3'b011:begin vga_r[2]<=1'bz; vga_r[1]<=valid ? dis : 1'b0; vga_r[0]<= valid ? dis : 1'b0;end
   3'b100:begin vga_r[2]<=valid ? dis : 1'b0; vga_r[1]<=valid ? dis : 1'b0; vga_r[0]<= 1'bz;end
   3'b101:begin vga_r[2]<=valid ? dis : 1'b0; vga_r[1]<=1'bz; vga_r[0]<= valid ? dis : 1'b0;end
   3'b110:begin vga_r[2]<=valid ? dis : 1'b0; vga_r[1]<=valid ? dis : 1'b0; vga_r[0]<= 1'bz;end
   3'b111:begin vga_r[2]<=valid ? dis : 1'b0; vga_r[1]<=valid ? dis : 1'b0;vga_r[0]<= valid ? dis :
1'b0;end
   default:begin vga_r[2]<=1'bz; vga_r[1]<=1'bz; vga_r[0]<=1'bz;;end
   endcase
end
always@(posedge clk)
begin
   case(coldata[5:3])   //G
```

```
    3'b000:begin vga_g[2]<=1'bz; vga_g[1]<=1'bz; vga_g[0]<=1'bz;end
    3'b001:begin vga_g[2]<=1'bz; vga_g[1]<=1'bz; vga_g[0]<= valid ? dis : 1'b0;end
    3'b010:begin vga_g[2]<=1'bz; vga_g[1]<=valid ? dis : 1'b0; vga_g[0]<= 1'bz;end
    3'b011:begin vga_g[2]<=1'bz; vga_g[1]<=valid ? dis : 1'b0; vga_g[0]<= valid ? dis : 1'b0;end
    3'b100:begin vga_g[2]<=valid ? dis : 1'b0; vga_g[1]<=valid ? dis : 1'b0; vga_g[0]<= 1'bz;end
    3'b101:begin vga_g[2]<=valid ? dis : 1'b0; vga_g[1]<=1'bz; vga_g[0]<= valid ? dis : 1'b0;end
    3'b110:begin vga_g[2]<=valid ? dis : 1'b0; vga_g[1]<=valid ? dis : 1'b0; vga_g[0]<= 1'bz;end
    3'b111:begin vga_g[2]<=valid ? dis : 1'b0; vga_g[1]<=valid ? dis : 1'b0; vga_g[0]<= valid ? dis :
1'b0;end
    default:begin vga_g[2]<=1'bz; vga_g[1]<=1'bz; vga_g[0]<=1'bz;;end
    endcase
end
always@(posedge clk)
begin
    case(coldata[2:0])    //B
    3'b000:begin vga_b[2]<=1'bz; vga_b[1]<=1'bz; vga_b[0]<=1'bz;end
    3'b001:begin vga_b[2]<=1'bz; vga_b[1]<=1'bz; vga_b[0]<= valid ? dis : 1'b0;end
    3'b010:begin vga_b[2]<=1'bz; vga_b[1]<=valid ? dis : 1'b0; vga_b[0]<= 1'bz;end
    3'b011:begin vga_b[2]<=1'bz; vga_b[1]<=valid ? dis : 1'b0; vga_b[0]<= valid ? dis : 1'b0;end
    3'b100:begin vga_b[2]<=valid ? dis : 1'b0; vga_b[1]<=valid ? dis    : 1'b0; vga_b[0]<= 1'bz;end
    3'b101:begin vga_b[2]<=valid ? dis : 1'b0; vga_b[1]<=1'bz; vga_b[0]<= valid ? dis : 1'b0;end
    3'b110:begin vga_b[2]<=valid ? dis : 1'b0; vga_b[1]<=valid ? dis : 1'b0; vga_b[0]<= 1'bz;end
    3'b111:begin vga_b[2]<=valid ? dis : 1'b0; vga_b[1]<=valid ? dis : 1'b0; vga_b[0]<= valid ? dis :
1'b0;end
    default:begin vga_b[2]<=1'bz; vga_b[1]<=1'bz; vga_b[0]<=1'bz;;end
    endcase
end

endmodule
```

# 12.6LCD12864 显示字符

## 1、LCD12864 驱动原理

LCD12864 的驱动原理与 LCD1602 的驱动原理相似，不同之处体现在 LCD12864 采用的是上下分屏或者是左右分屏进行驱动的。

12864是一种图形点阵液晶显示器,它主要由行驱动器/ 列驱动器及128×64全点阵液晶显示器组成.可完成图形显示,也可以显示8×4个(16×16点阵)汉字。

主要技术参数和性能:

1.电源:VDD:+5V；

2.显示内容:128(列)×64(行)点

3.全屏幕点阵

4.七种指令

5.与 CPU 接口采用 8 位数据总线并行输入输出和 8 条控制线.

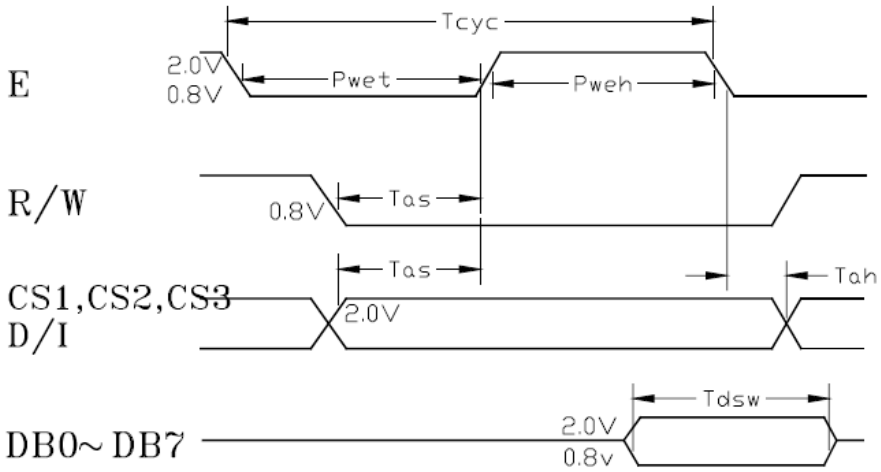
**LCD12964 的接口说明**

| 管脚号 | 管脚名称 | LEVER | 管脚功能描述 |
|---|---|---|---|
| 1 | VSS | 0V | 电源地 |
| 2 | VDD | 5.0V | 电源电压 |
| 3 | V0 | - | 液晶显示器驱动电压 |
| 4 | RS | H/L | D/I="H",表示 DB7~DB0 为显示数据<br>D/I="L",表示 DB7~DB0 为显示指令数据 |
| 5 | R/W | H/L | R/W="H",E="H",数据被读到 DB7~DB0<br>R/W="L",E="H→L", DB7~DB0 的数据被写到 IR 或 DR |
| 6 | E | H/L | 使能信号：R/W="L",E 信号下降沿锁存 DB7~DB0<br>R/W="H",E="H" DRAM 数据读到<br>DB7~DB0 |
| 7 | DB0 | H/L | 数据线 |
| 8 | DB1 | H/L | 数据线 |
| 9 | DB2 | H/L | 数据线 |
| 10 | DB3 | H/L | 数据线 |
| 11 | DB4 | H/L | 数据线 |
| 12 | DB5 | H/L | 数据线 |
| 13 | DB6 | H/L | 数据线 |
| 14 | DB7 | H/L | 数据线 |
| 15 | CS1 | H/L | H:选择芯片(右半屏)信号 |
| 16 | CS2 | H/L | H:选择芯片(左半屏)信号 |
| 17 | RET | H/L | 复位信号,低电平复位 |
| 18 | VOUT | -10V | LCD 驱动负电压 |
| 19 | LED+ | DC+5V | LED 背光板电源 |
| 20 | LED- | DC0V | LED 背光板电源 |

由该表可以知道，要对 LCD12864 进行显示操作则必须操作数据信号（DB）和控制信号（RS、R/W、E）。发送数据的时候，采用 RS 来区分数据和指令，采用 R/W 来控制数据的读

写。因为显示 LCD12864 只需用到写数据，所以当向 LCD12864 送入数据的时候将 E 输出一个下降沿。具体的操作方法要根据写数据时序进行设计。

**LCD12864 的写数据时序分析**




| 名　　称 | 符　　号 | 最小值 | 典型值 | 最大值 | 单位 |
|---|---|---|---|---|---|
| E 周期时间 | Tcyc | 1000 | --- | --- | ns |
| E 高电平宽度 | Pweh | 450 | --- | --- | ns |
| E 低电平宽度 | Pwel | 450 | --- | --- | ns |
| E 上升时间 | Tr | --- | --- | 25 | ns |
| E 下降时间 | Tf | --- | --- | 25 | ns |
| 地址建立时间 | Tas | 140 | --- | --- | ns |
| 地址保持时间 | Tah | 10 | --- | --- | ns |
| 数据建立时间 | Tdsw | 200 | --- | --- | ns |
| 数据延迟时间 | Tddr | --- | --- | 320 | ns |
| 写数据保持时间 | Tdhw | 10 | --- | --- | ns |
| 读数据保持时间 | Tdhw | 20 | --- | --- | ns |

对 LCD12864 写数据时，主要的控制信号有:E、R/W、RS（送数据时置为高电平，送指令时置为低电平）。根据时序的参数表，在该工程中将 E 的时钟频率设置为 1.5M（芯片手册的频率为 1M），在进行写数据时根据 E 的时钟边沿，在 E 的上升沿将 R/W 置低，然后送出数据，当 E 出现下降沿的时候数据便会读进 LCD12864 内部寄存器中。

**LCD12864 控制指令说明**

| 指令 | 指令码 | | | | | | | | | | 功能 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | RS | R/W | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | |
| 清除显示 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 将DDRAM填满"20H"，并且设定DDRAM的地址计数器(AC)到"00H" |
| 地址归位 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | X | 设定 DDRAM 的地址计数器(AC)到"00H"，并且将游标移到开头原点位置；这个指令不改变 DDRAM 的内容 |
| 显示状态开/关 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | D | C | B | D=1：整体显示 ON<br>C=1：游标 ON<br>B=1:游标位置反白允许 |
| 进入点设定 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | I/D | S | 指定在数据的读取与写入时，设定游标的移动方向及指定显示的移位 |
| 游标或显示移位控制 | 0 | 0 | 0 | 0 | 0 | 1 | S/C | R/L | X | X | 设定游标的移动与显示的移位控制位；这个指令不改变 DDRAM 的内容 |
| 功能设定 | 0 | 0 | 0 | 0 | 1 | DL | X | RE | X | X | DL=0/1：4/8 位数据<br>RE=1：扩充指令操作<br>RE=0：基本指令操作 |
| 设定 CGRAM 地址 | 0 | 0 | 0 | 1 | AC5 | AC4 | AC3 | AC2 | AC1 | AC0 | 设定 CGRAM 地址 |
| 设定 DDRAM 地址 | 0 | 0 | 1 | 0 | AC5 | AC4 | AC3 | AC2 | AC1 | AC0 | 设定 DDRAM 地址（显示位址）<br>第一行：80H－87H<br>第二行：90H－97H |
| 读取忙标志和地址 | 0 | 1 | BF | AC6 | AC5 | AC4 | AC3 | AC2 | AC1 | AC0 | 读取忙标志(BF)可以确认内部动作是否完成，同时可以读出地址计数器(AC)的值 |
| 写数据到 RAM | 1 | 0 | 数据 | | | | | | | | 将数据 D7——D0 写入到内部的 RAM (DDRAM/CGRAM/IRAM/GRAM) |
| 读出 RAM 的值 | 1 | 1 | 数据 | | | | | | | | 从内部 RAM 读取数据 D7——D0 (DDRAM/CGRAM/IRAM/GRAM) |

在进行写数据之前必须进行寄存器的设置，一般的显示设置涉及到的指令有:清除显示、显示开关状态、游标或显示移位控制、功能设定。

清除显示：01H

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 清除显示 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 将DDRAM填满"20H"，并且设定DDRAM的地址计数器(AC)到"00H" |

显示开关状态：0CH（整体显示、关游标、不允许反白）

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 显示状态开/关 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | D | C | B | D=1：整体显示 ON<br>C=1：游标 ON<br>B=1:游标位置反白允许 |

游标或显示移位控制：

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 游标或显示移位控制 | 0 | 0 | 0 | 0 | 0 | 1 | S/C | R/L | X | X | 设定游标的移动与显示的移位控制位；这个指令不改变 DDRAM 的内容 |

功能设定:31H(8 位格式,基本指令)

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 功能设定 | 0 | 0 | 0 | 0 | 1 | DL | X | RE | X | X | DL=0/1：4/8 位数据<br>RE=1：扩充指令操作<br>RE=0：基本指令操作 |

指令设置过程如下：

1. 设置 8 位格式
2. 整体显示,关光标,不闪烁
3. 设定输入方式,增量不移位
4. 清除显示

**LCD12864 字符显示地址：**

每屏可显示4 行8 列共32个16×16 点阵的汉字，每个显示RAM 可显示1个中文字符或2个16×8 点阵全高ASCII 码字符，即每屏最多可实现32个中文字符或64个ASCII 码字符的显示。根据写入内容的不同，可分别在液晶屏上显示CGROM（中文字库）、HCGROM（ASCII 码字库）及CGRAM（自定义字形）的内容。字符显示RAM 在液晶模块中的地址80H～9FH。字符显示的RAM 的地址与32个字符显示区域有着一一对应的关系，其对应关系如下表所示：

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 80H | 81H | 82H | 83H | 84H | 85H | 86H | 87H |
| 90H | 91H | 92H | 93H | 94H | 95H | 96H | 97H |
| 88H | 89H | 8AH | 8BH | 8CH | 8DH | 8EH | 8FH |
| 98H | 99H | 9AH | 9BH | 9CH | 9DH | 9EH | 9FH |

在该工程中需要将LCD12864的整个屏幕显示中文字符和英文字符，显示将整个屏幕分为4行进行显示控制。第一行到第四行的地址分别为80H、90H、88H、98H。

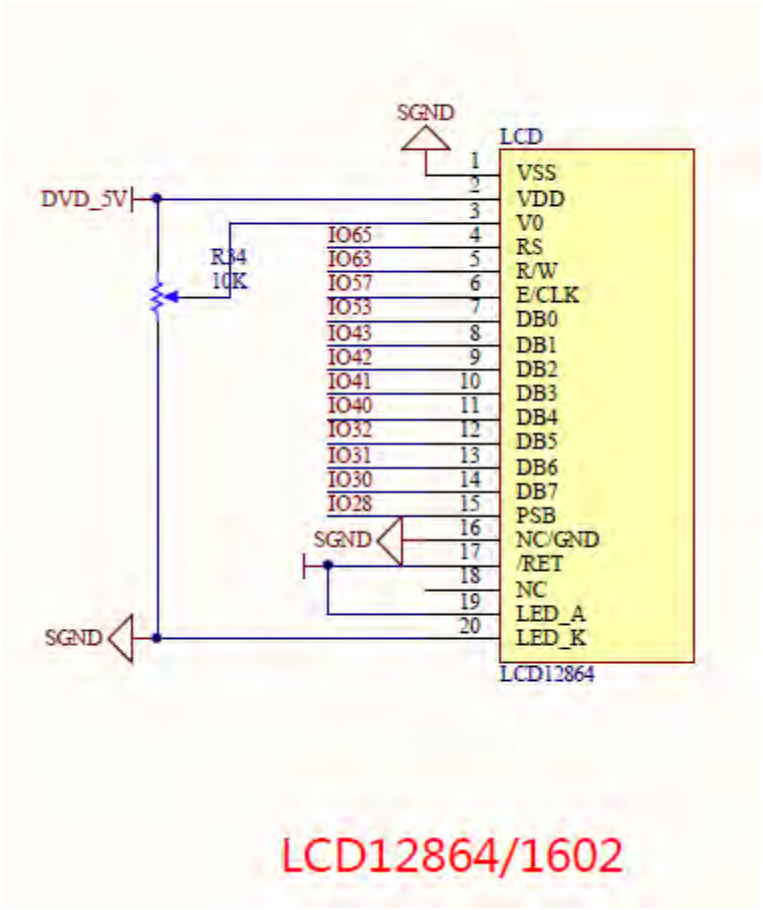
在显示数据发送完之后，将E置低，然后延迟一段时间后将E拉高，最后对屏幕进行刷新。

在显示ASCII 码字符时可以直接采用字符，而当显示中文时必须查找中文字型码表，中文字型采用的是16位的寻址，由于数据线只有8位，所以输出数据时采用分时复用，先输出高八位，后输出低八位。假如要显示“大”，通过中文字型码表可以得出其地址为：B4F3。

B4F0 答 搭 打 大 呆 歹 傣 戴 带 殆 代 贷 袋 待 逮

所以输出数据时，先输出B4，后输出F3。

## 2、实验硬件说明




LCD12864 的接口与 LCD1602 接口进行复用，所以使用时需要将 PSB 置高，作为背光电源，滑动变阻器用来设置背光灯的亮度。

## 3、程序设计

主要分为两个模块，分别为分频模块，用来产生驱动的信号信号和使能信号 E。另一个为显示驱动模块用来进行指令和数据的发送，控制过程采用状态机进行控制各个状态的转移。

**分频模块：**

```
always @(posedge clk)              //为 E 提高时钟信号，大概 1.5M
  begin
    counter=counter+1;
    if(counter==16'h000f)
    clkr=~clkr; // 1.562M
end
```

**显示驱动模块**

```
always @(posedge clkr)
begin
  current=next;
    case(current)
                                        //写指令
      7'd0:     begin    rs<=0; dat<=8'h31;    next<=next+1'b1;end    //设置 8 位格式
```

```
7'd1:    begin   rs<=0; dat<=8'h0C;    next<=next+1'b1;end //整体显示,关光标,不闪烁
7'd2:    begin   rs<=0; dat<=8'h06;    next<=next+1'b1; end    //设定输入方式,增量不移位
7'd3:    begin   rs<=0; dat<=8'h01;    next<=next+1'b1; end //清除显示
                                       //写数据
7'd4:    begin   rs<=1; dat<=8'hB4;    next<=next+1'b1; end //显示第一行
7'd5:    begin   rs<=1; dat<=8'hF3;    next<=next+1'b1; end//大
7'd6:    begin   rs<=1; dat<=8'hCE;    next<=next+1'b1; end
7'd7:    begin   rs<=1; dat<=8'hF7;    next<=next+1'b1; end //西
7'd8:    begin   rs<=1; dat<=8'hB9;    next<=next+1'b1; end
7'd9:    begin   rs<=1; dat<=8'hCF;    next<=next+1'b1; end//瓜
7'd10:    begin   rs<=1; dat<="-"; next<=next+1'b1; end
7'd11:    begin   rs<=1; dat<="F";next<=next+1'b1; end
7'd12:    begin   rs<=1; dat<="P"; next<=next+1'b1; end
7'd13:    begin   rs<=1; dat<="G";next<=next+1'b1; end
7'd14:    begin   rs<=1; dat<="A"; next<=next+1'b1; end
7'd15:    begin   rs<=1; dat<="!"; next<=next+1'b1; end
7'd16:    begin   rs<=1; dat<=" ";next<=next+1'b1; end
7'd17:    begin   rs<=1; dat<=" "; next<=next+1'b1; end
7'd18:    begin   rs<=0; dat<=8'h90; next<=next+1'b1; end //显示第二行
7'd19:    begin   rs<=1; dat<=8'hcc; next<=next+1'b1; end
7'd20:    begin   rs<=1; dat<=8'hd4; next<=next+1'b1; end //淘
7'd21:    begin   rs<=1; dat<=8'hb1; next<=next+1'b1; end
7'd22:    begin   rs<=1; dat<=8'ha6; next<=next+1'b1; end //宝
7'd23:    begin   rs<=1; dat<=8'hb5; next<=next+1'b1; end
7'd24:    begin   rs<=1; dat<=8'hea; next<=next+1'b1; end//店
7'd25:    begin   rs<=1; dat<=":"; next<=next+1'b1; end
7'd26:    begin   rs<=1; dat<="h"; next<=next+1'b1; end
7'd27:    begin   rs<=1; dat<="t"; next<=next+1'b1; end
7'd28:    begin   rs<=1; dat<="t"; next<=next+1'b1; end
7'd29:    begin   rs<=1; dat<="p"; next<=next+1'b1; end
7'd30:    begin   rs<=1; dat<=":"; next<=next+1'b1; end
7'd31:    begin   rs<=1; dat<="/"; next<=next+1'b1; end
7'd32:    begin   rs<=1; dat<="/"; next<=next+1'b1; end
7'd33:    begin   rs<=1; dat<="d"; next<=next+1'b1; end
7'd34:    begin   rs<=1; dat<="a"; next<=next+1'b1; end
7'd35:    begin   rs<=0; dat<=8'h88; next<=next+1'b1; end//显示第三行
7'd36:    begin   rs<=1; dat<="x"; next<=next+1'b1; end
7'd37:    begin   rs<=1; dat<="i"; next<=next+1'b1; end
7'd38:    begin   rs<=1; dat<="g"; next<=next+1'b1; end
7'd39:    begin   rs<=1; dat<="u"; next<=next+1'b1; end
7'd40:    begin   rs<=1; dat<="a"; next<=next+1'b1; end
7'd41:    begin   rs<=1; dat<="f"; next<=next+1'b1; end
7'd42:    begin   rs<=1; dat<="p"; next<=next+1'b1; end
```

```
        7'd43:    begin    rs<=1; dat<="g"; next<=next+1'b1;    end
        7'd44:    begin    rs<=1; dat<="a"; next<=next+1'b1;    end
        7'd45:    begin    rs<=1; dat<="."; next<=next+1'b1;    end
        7'd46:    begin    rs<=1; dat<="t"; next<=next+1'b1; end
        7'd47:    begin    rs<=1; dat<="a"; next<=next+1'b1; end

        7'd48:    begin    rs<=1; dat<="o"; next<=next+1'b1; end
        7'd49:    begin    rs<=1; dat<="b"; next<=next+1'b1; end
        7'd50:    begin    rs<=1; dat<="a"; next<=next+1'b1; end
        7'd51:    begin    rs<=1; dat<="o"; next<=next+1'b1; end
        7'd52:    begin    rs<=0; dat<=8'h98; next<=next+1'b1; end //显示第四行
        7'd53:    begin    rs<=1; dat<="."; next<=next+1'b1; end
        7'd54:    begin    rs<=1; dat<="c"; next<=next+1'b1; end
        7'd55:    begin    rs<=1; dat<="o"; next<=next+1'b1; end
        7'd56:    begin    rs<=1; dat<="m"; next<=next+1'b1; end
        7'd57:    begin    rs<=1; dat<=8'hd0; next<=next+1'b1; end
        7'd58:    begin    rs<=1; dat<=8'hbb; next<=next+1'b1; end
        7'd59:    begin    rs<=1; dat<=8'hd0; next<=next+1'b1; end
        7'd60:    begin    rs<=1; dat<=8'hbb; next<=next+1'b1; end
        7'd61:    begin rs<=0;    dat<=8'h00;                    //把液晶的 E 脚拉高
                    if(cnt!=2'b10) //延迟一小段时间，然后再进行刷新
                          begin
                                e<=0;next<=7'd0;cnt<=cnt+1;
                          end
                           else
                              begin next<=7'd61; e<=1; end
                    end
      default:    next=7'd0;
       endcase
   end
assign en=clkr|e;//将使能信号 E 与时钟信号进行同步
```

# 12.7LCD12864 显示图片

## 1、 LCD12864 图片显示原理

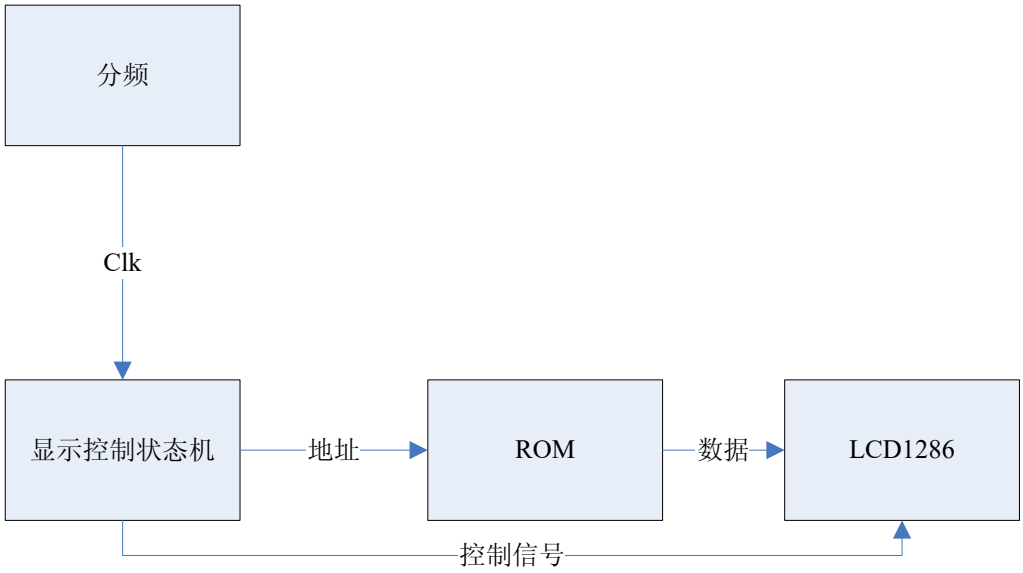
LCD12864 的驱动原理和时序可以参考上一节“LCD12864 显示字符”，LCD12864 显示图片的关键点就是要弄懂一个问题：LCD12864 中每一像素点要如何去点亮或者是熄灭。

LCD12864 的显示屏简图如下：


屏幕分为上下半屏，分别为 128*32 个像素点。而上下分别的控制是通过选择地址进行控制上下半屏的显示。

将屏幕分为：64 行*128 列，显示顺序为从左上到右下。

显示的驱动框架如下：




**分频模块：**

将 50M 的时钟信号进行简单的 16 分频，输出大概 1.5M 时钟信号，给显示控制模块。

```
/*****************时钟分频*********************/
always @(posedge clk or negedge rst)
```

```
begin
    if(!rst)
        div_cnt <= 15'd0;
    else
      if(div_cnt==15'h4000)
        begin
          div_cnt <= 15'd0;
          clk_div<=~clk_div;
        end
      else
    div_cnt <= div_cnt+ 1'b1;
end
```

**显示控制状态机：**

主要分为三个部分：

1、 LCD12864 初始化，设置显示模式。

2、 ROM 地址的生成

3、 LCD12864 行列地址的控制

只要解决了这三个问题，LCD12864 的显示控制就基本解决了。下面我们一一讲解：

在显示过程中，行数有 64 行，每一行有 128 列，这里我们将这 128 列分为 16 个部分，每一部分为 8 列，这样 128*64 就变成了 16*64，这样显示的最小元素就从一个像素点变成了 1*8 个像素点。这样就总共需要 16*64=1024 个数据单元，每个数据单元的位宽为 8。这样我们就可以很方便读出数据进行显示了。这里需要创建一个深度为 1024，位宽为 8 的 ROM，然后将我们要显示的图片数据保存在这个 ROM 中，然后通过控制 ROM 的地址，将显示屏上的每一个元素的对应数据读出送到数据线上。

在知道了数据从哪里来到数据到哪里去之后，接下来就是要解决一个重要问题：ROM 的寻址。这里将显示屏分为：行（X）和列（Y）。该工程中采用一个地址计数器进行地址的计数，由于显示屏分为上下半屏，各个屏显示的数据量为 512，所以这里就可以用该地址计数器的最高位来控制上下半屏的驱动，当最高位为 0 时，控制上半屏，为 1 时控制下半屏。在设置 LCD122864 的寄存器的时候将 DDRAM 的地址计数器(AC)设置为自动增加，这样在每写如一个元素数据之后，LCD12864 的行地址就会自动加一，显示的位置就会跳到下一个，就不需要不断往 LCD12864 里送地址了，只需在每写入一行后，切换列地址。

对应代码段如下：

```
wr_x_addr_1:
        begin
        next_state <=    wr_x_addr_2;
            lcd12864_en <= 1'b1;
    lcd12864_data <= {4'd8,cnt[9],3'd0};//上半屏:0x80;下半屏:0x88
        end
        wr_x_addr_2:
        begin
            next_state <= wr_data_1;
    lcd12864_data <= {4'd8,cnt[9],3'd0};
    end
```

同时，将地址计数器送入 ROM 的地址线，这样就可以对 ROM 进行寻址。等等，这里只是控制好了 X 地址，而 Y 地址的控制是通过每完成显示一行，即送出了 16 个数据块，Y 的地址便加一，然后更新到 LCD12864 上，由于 Y 地址是逢 16 进一的，所以就可以用地址计数器的第 8 位~第 4 位对 LCD 进行列控制。

对应代码段如下：

```
wr_y_addr_1:                          //设定图形显示区 Y 轴地址
        begin
        next_state <= wr_y_addr_2;
    lcd12864_data <= {3'b100,cnt[8:4]};//设置列地址（y）
            lcd12864_en <= 1'b1;
        end
        wr_y_addr_2:
        begin
            next_state <= wr_x_addr_1;
    lcd12864_data <= {3'b100,cnt[8:4]};
        end
```

在写完 64 行之后（其实就是一帧）整个显示屏就驱动完成了，最后再转到初始状态不断刷新屏幕。

对应代码段如下：

```
wr_data_1:
        begin
            next_state <= wr_data_2;
    lcd12864_rs <= 1'b1;
            lcd12864_en <= 1'b1;
    lcd12864_data <= data;
        end

        wr_data_2://写数据
        begin
            lcd12864_rs<=1'b1;
            lcd12864_data <= data;
            if(cnt[3:0] == 4'hf)//写完一行数据(16 个)
            begin
                if(cnt[9:4] == 7'h3f)//写完一屏数据(64 行)
                    next_state <= idle;//写完一屏的数据后重新跳转到初始状态
                else
                    next_state <= wr_y_addr_1;//每写完一行数据，重新写入行地址
            end
            else
                next_state <= wr_data_1;//每写完一行数据，重新写入行地址，地址会自动加一
        end
```

```
        default:        next_state<=idle；//跳转到初始状态
        endcase
end
```

## 2、程序设计

```
/**************状态机转换*********/
always @(posedge clk_div or negedge rst)
begin
        if(!rst)
                state <= idle;
        else
        state <= next_state;
end
/**********************状态机逻辑*******************************/
always @(state or cnt or data)
begin
        lcd12864_rs <= 1'b0;
        lcd12864_en <= 1'b0;
        cnt_rst <= 1'b0;
        lcd12864_data <= 8'h0;
        case(state)
                idle:
                begin
                        next_state <= setbase_1;
                end
/************************初使化 LCD*****************************/
/******************设置为基本指令操作***************/
                setbase_1:
                begin
                        next_state <= setmode_1;
                lcd12864_data <= 8'h30;
                        lcd12864_en <= 1'b1;
                end

/*************设定 DDRAM  的地址计数器(AC)自动增加********************/
                setmode_1:
                begin
                next_state <= setcurs_1;
                        lcd12864_data <= 8'h06;
                        lcd12864_en <= 1'b1;
                end

/**************开显示,关光标,不闪******************/
```

```
            setcurs_1:
            begin
                next_state <= setexte_1;
            lcd12864_data <= 8'h0c;
                lcd12864_en <= 1'b1;
            end
/**************扩充指令操作*******************/
            setexte_1:
            begin
                next_state<=setexte_2;
            lcd12864_data <= 8'h36;
                lcd12864_en <= 1'b1;
            end

            setexte_2:
            begin
                next_state <= wr_y_addr_1;
                lcd12864_data <= 8'h36;
                cnt_rst <= 1'b1;//图片数据帧复位标志
    end
/********************写图片数据到 LCD 显存************************/
            wr_y_addr_1:                        //设定图形显示区 Y 轴地址
            begin
            next_state <= wr_y_addr_2;
        lcd12864_data <= {3'b100,cnt[8:4]};//设置列地址（y）
                lcd12864_en <= 1'b1;
            end
            wr_y_addr_2:
            begin
                next_state <= wr_x_addr_1;
        lcd12864_data <= {3'b100,cnt[8:4]};
            end
/****************设定图形显示区 X 轴地址*****************/
          wr_x_addr_1:
            begin
            next_state <=    wr_x_addr_2;
                lcd12864_en <= 1'b1;
        lcd12864_data <= {4'd8,cnt[9],3'd0};//上半屏:0x80;下半屏:0x88
            end
            wr_x_addr_2:
            begin
                next_state <= wr_data_1;
        lcd12864_data <= {4'd8,cnt[9],3'd0};
```

```
            end
    /****************写数据到图形显示区****************/
            wr_data_1:
            begin
                next_state <= wr_data_2;
        lcd12864_rs <= 1'b1;
                lcd12864_en <= 1'b1;
        lcd12864_data <= data;
            end
            wr_data_2://写数据
            begin
                lcd12864_rs<=1'b1;
                lcd12864_data <= data;
                if(cnt[3:0] == 4'hf)//写完一行数据(16 个)
                begin
                    if(cnt[9:4] == 7'h3f)//写完一屏数据(64 行)
                        next_state <= idle;//写完一屏的数据后重新跳转到初始状态
                    else
                        next_state <= wr_y_addr_1;//每写完一行数据，重新写入行地址
                end
                else
                    next_state <= wr_data_1;//每写完一行数据，重新写入行地址，地址
会自动加一
            end
        default:        next_state<=idle；//跳转到初始状态
        endcase
    end
```

ROM 接口：

```
    /*****************调用 ROM(图片数据)*****************************/
    rom imagerom(.address (cnt),.clock (clk),.q (data));
```

## 3、图片取模与 MIF 文件的生成

1、打开 Image2Lcd 取模软件，



**2、** 打开要显示的图片，**图片的大小是 128*64 像素，可以选择实验提供的两张图片：**


Image2Lcd v2.9
打开
保存
设置
重新载入
上一幅
下一幅
帮助
关于
输出数据类型：
二进制(*.bin)
扫描模式：
水平扫描
输出灰度：
单色
最大宽度和高度
128
64
包含图像头数据
字节内象素数据反序
自右至左扫描
自底至顶扫描
高位在前(MSB First)
恢复缺省值
颜色反转
正常显示
亮度：
对比度：
输出图像调整
256色
4096色
16位彩色
18位彩色
24位彩色
32位彩色
注册
输入图像：lady.BMP (128,64)
输出图像：(128,64)

3、设置取模的参数：


Image2Lcd v2.9
打开
保存
设置
重新载入
上一幅
下一幅
帮助
关于
输出数据类型：
二进制(*.bin)
扫描模式：
水平扫描
输出灰度：
单色
最大宽度和高度
128
64
包含图像头数据
字节内象素数据反序
自右至左扫描
自底至顶扫描
高位在前(MSB First)
恢复缺省值
颜色反转
正常显示
亮度：
对比度：
输出图像调整
256色
4096色
16位彩色
18位彩色
24位彩色
32位彩色
注册
输入图像：lady.BMP (128,64)
输出图像：(128,64)

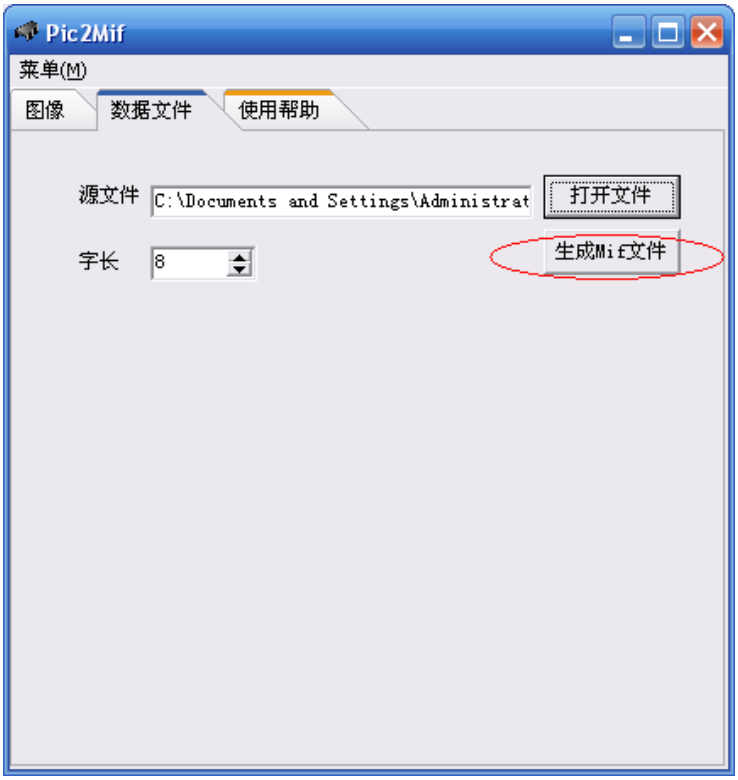
4、设置完成之后就点击保存： 


5、接下来就用 BmpToMif.exe 软件将.bin 文件转换为.Mif 文件



打开刚刚生成的.bin 文件，设置字长为 8

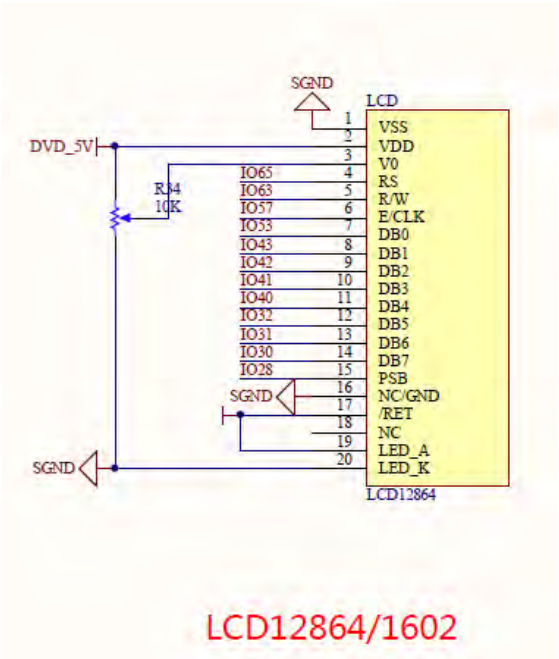

6、点击生成 MIF 文件





打开 MIF 文件，查看数据是否正确：

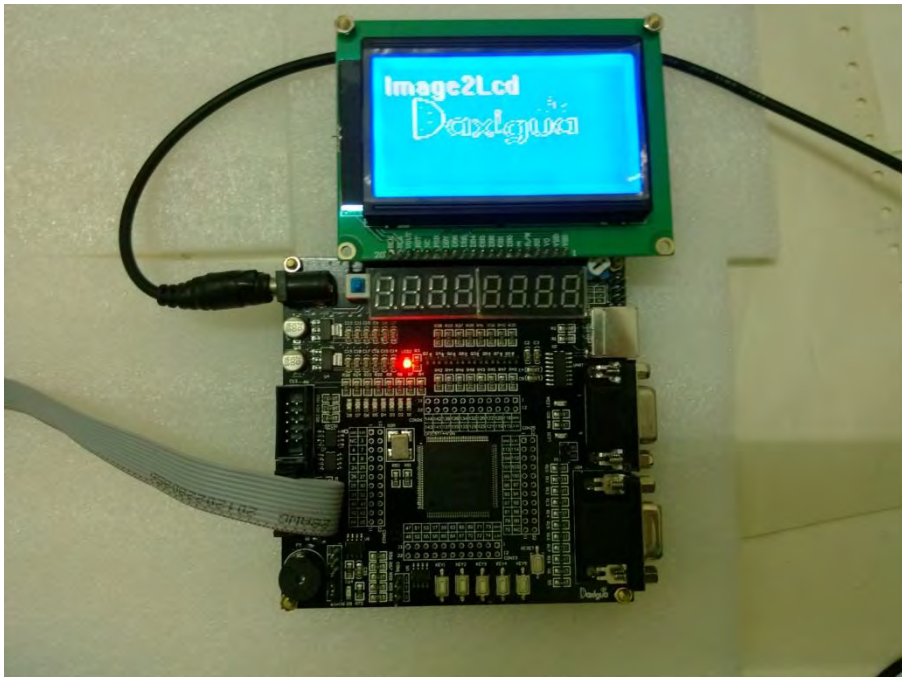
## 4、实验硬件说明




LCD12864 的接口与 LCD1602 接口进行复用，所以使用时需要将 PSB 置高，作为背

光电源，滑动变阻器用来设置背光灯的亮度。

## 5、实验效果



# 第十三章综合实验

## 13.1 基于 DDS 的任意波形发生器

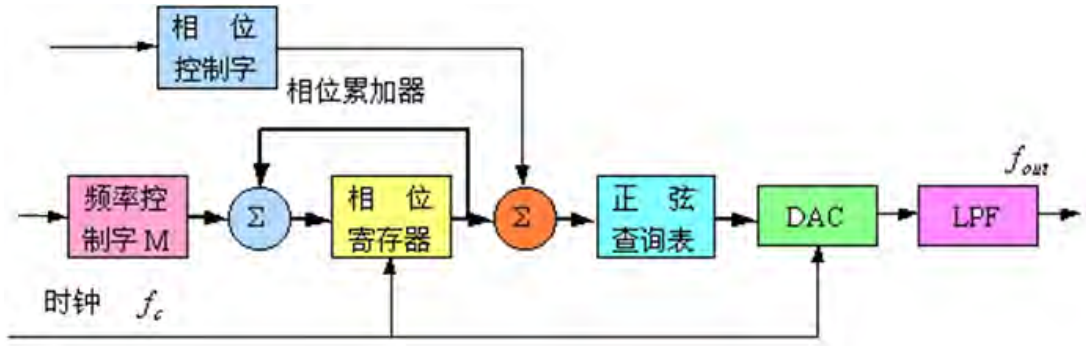
### 一、 DDS 的原理

DDS(Direct Digital Frequency Synthesizer)直接数字频率合成器,也可叫 DDFS。

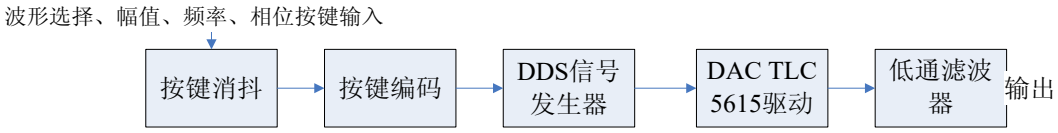
- DDS 是从相位的概念直接合成所需波形的一种频率合成技术。

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。

DDS 原理框图




### 二、 整体框架及其说明




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。整个工程主要分为五个模块：按键消抖、按键编码、DDS 信号发生器（在做这个实验前一定要先把 DDS 的原理弄懂!!!）DAC TLC5615 驱动、低通滤波器（硬件板上已经有了）。

按键消抖模块：由于需要控制波形、幅值、频率、相位三个参数，所以该模块需要三个按键，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，这里采用的消抖方法为状态机消抖，具体的过程可以看教程的按键消抖那个章节。

```
always @(posedge clk)
 begin
     case (state)
      s0:
       begin
        key_out<=1'b1;
       if(key==1'b0)
            state<=s1;
       else
            state<=s0;
```

```
            end
          s1:
            begin
              if(key==1'b0)
                 state<=s2;
              else
                 state<=s0;
              end
          s2:
            begin
              if(key==1'b0)
                 state<=s3;
              else
                 state<=s0;
              end
          s3:
             begin
               if(key==1'b0)
                 begin
                 key_out<=1'b0;
                 state<=s3;
                 end
               else
                 begin
                 key_out<=1'b1;
                 state<=s0;
                 end
             end
         default:
                 state<=s0;
        endcase
   end
```

按键编码：即是对已经经过按键消抖后的按键信号进行编码，通过按键的按下控制幅值控制变量、频率控制变量、相位控制变量的大小。该模块采用的方法是每按下按键一次相对应的控制变量的值便会增加，最后的相应的输出也会变化。

```
/******************波形设置*****************************************/
always @(negedge set_waveform_key or negedge reset)
begin
   if(!reset)
     set_waveform<=2'b00;
   else
      begin
       if(!set_waveform_key)//判断按键是否按下
```

```
            begin
                set_waveform<=set_waveform+1'b1;
            end
        end
    end
/***************频率设置*****************************************/
always @(negedge set_f_key or negedge reset)
    begin
     if(!reset)
        f_control<=16'b0100001100011011;//200Hz
     else
      begin
      if(!set_f_key)//判断按键是否按下
         begin
            if(f_control==16'hffff)
                f_control<=16'h0;
            else
                f_control<=f_control+16'd50;
         end
      end
    end
/***************幅值控制*****************************************/
always @(negedge set_a_key or negedge reset)
begin
   if(!reset)
    a_control<=4'd1;
   else
      begin
       if(!set_a_key)//判断按键是否按下
       begin
          if(a_control==4'd10)
           a_control<=4'd1;
          else
           a_control<=a_control+4'd1;
       end
      end
 end
/***************相位设置*****************************************/
always @(negedge set_p_key or negedge reset)
    begin
     if(!reset)
        p_control<=10'b00_0000_0000;//0 度
     else
```

```
        begin
        if(!set_p_key)//判断按键是否按下
          begin
            if(p_control==10'b11_1111_1111)
                p_control<=10'b0;
            else
                p_control<=p_control+10'd100;
          end
        end
      end
```

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 10 位，因为我们的 DAC 芯片是 10 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 DACTLC5615 的驱动模块中，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“8.11 DSS 与嵌入式逻辑分析仪的调用”。

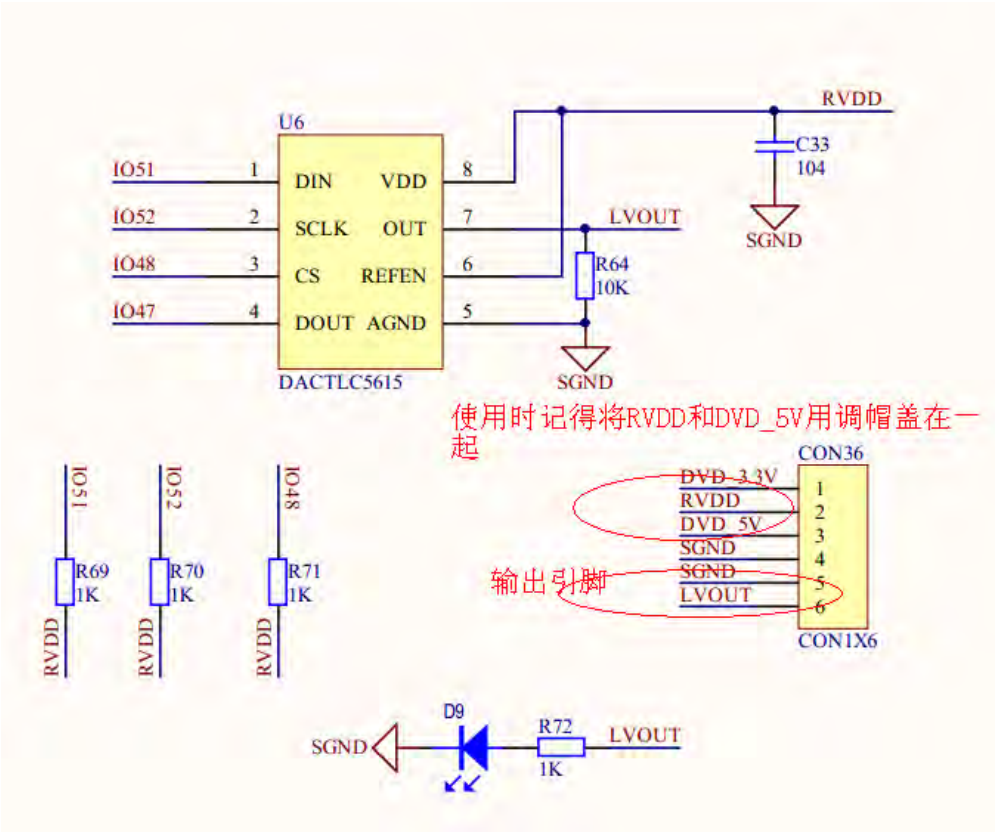
频率累加器和相位累加器，通过改变频率累加器的频率控制字就可以控制输出的波形频率，通过改变相位累加器的相位控制字就可以控制输出波形的相位。而幅值的改变是通过将通过 ROM 里面读出的数据就行等比例的缩减，通过改变缩减倍数就可以改变输出波形的幅值（0~3.3V），而改变的过程是通过外部的按键进行改变。

```
    /**********************元件例化************************************/
        adder_32  u1(.data1(f32_bus),.data2(reg32_out),.sum(reg32_in));//频率累加器
        reg32       u2(.clk(clk),.data_in(reg32_in),.data_out(reg32_out));//
        adder_10  u7(.data1(set_p),.data2(reg32_out[31:22]),.sum(reg10_in));//相位累加器
        reg_10      u8(.clk(clk),.data_in(reg10_in),.data_out(reg10_out_address));//
    /*****************波形选择*****************************************/
        sin_rom    u3(.address(reg10_out_address),.clock(clk),.q(sin_data));//正弦
        tri_rom    u4(.address(reg10_out_address),.clock(clk),.q(tri_data));
        squ_rom    u5(.address(reg10_out_address),.clock(clk),.q(squ_data));
        saw_rom    u6(.address(reg10_out_address),.clock(clk),.q(saw_data));
      always @(set_waveform,sin_data,tri_data,squ_data,saw_data)
        begin //对四种波形进行选择
          case (set_waveform)
            2'b00: dds_data_reg<=sin_data;
            2'b01: dds_data_reg<=tri_data;
            2'b10: dds_data_reg<=squ_data;
            2'b11: dds_data_reg<=saw_data;
          default:dds_data_reg<=sin_data;
          endcase;
        end
```

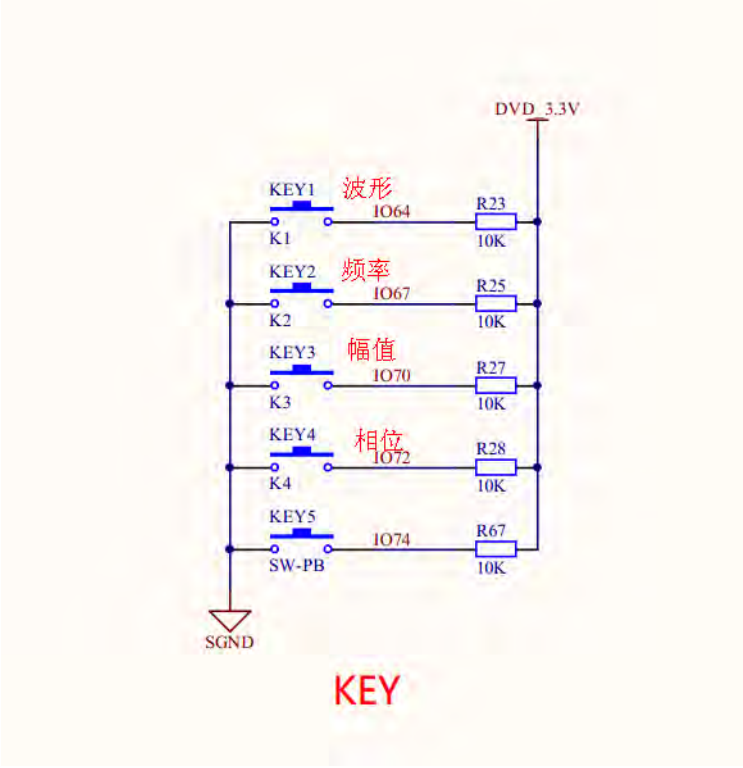
DAC TLC5615 驱动：具体看“10.2 DA_TLC5615 驱动输出”这一章节。

## 三、 实验原理图

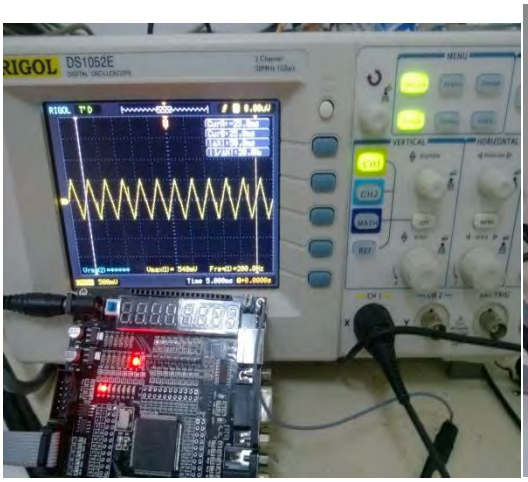
**DAC TLC 原理图机说明：**

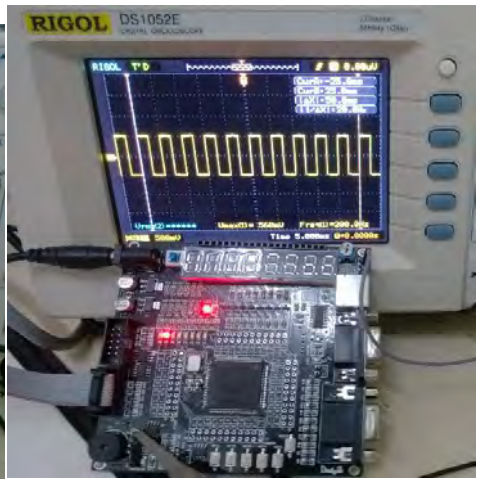


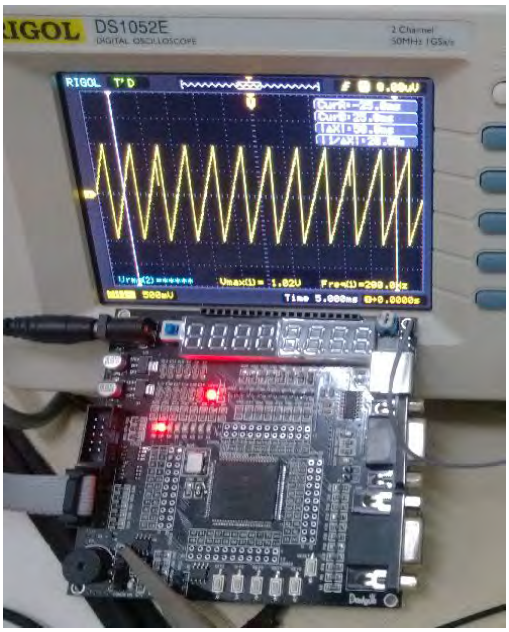

**按键原理图及说明：**

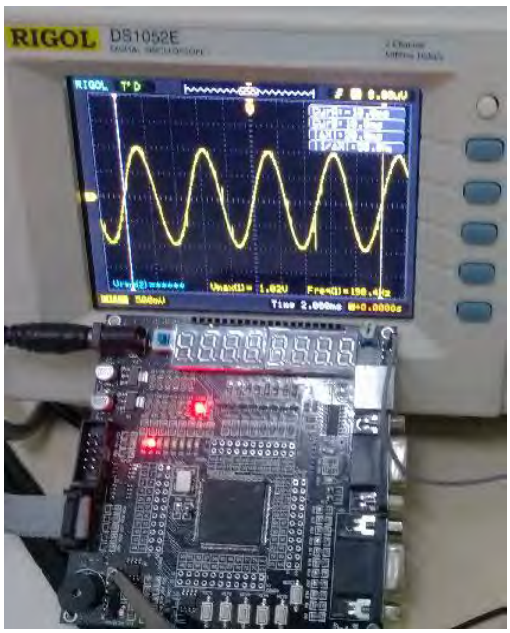


四、实验现象



# 13.2 基于 PS2 的 LCD1602 的显示

一、原理分析

该设计为基于 PS2 的 LCD 显示，实现的功能为在电脑键盘上按下一个按键，按下的按键的键值将显示在 LCD1602 上。其实现的过程为：FPGA 通过 PS2 通信接口与电脑键盘进行通信，读取电脑键盘上的键值，然后将键值寄存起来，再利用 FPGA 驱动 LCD1602，然后将键盘的键值显示在 LCD1602 上，显示的字符包括：A~Z、0~9。

1、 PS2 键盘接口原理

1.1、 PS2 的物理接口

| Male 公的 | Female 母的 | 6-pin Mini-DIN (PS/2): | 6 脚 Mini-DIN(PS/2) |
|---|---|---|---|
|  |  | 1 - Data | 1—数据 |
| | | 2 - Not Implemented | 2—未实现，保留 |
| | | 3 - Ground | 3—电源地 |
| | | 4 - +5v | 4—电源+5V |
| (Plug) 插头 | (Socket) 插座 | 5 - Clock | 5—时钟 |
| | | 6 - Not Implemented | 6—未实现，保留 |

1.2、 原理说明

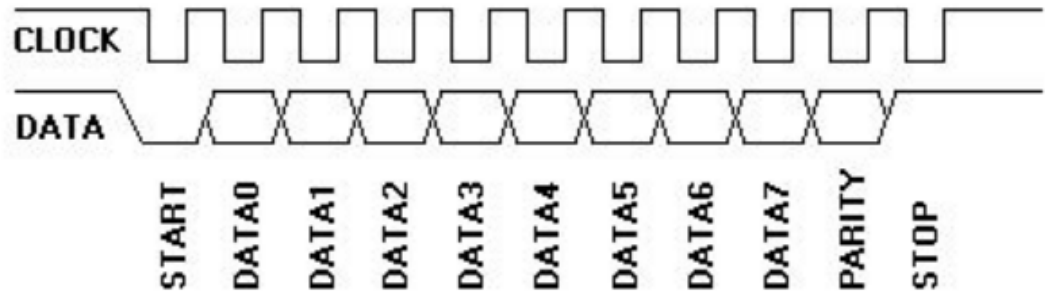
从键盘发送到主机的数据在时钟信号的下降沿当时钟从高变到低的时候被读取从主机发送到键盘的数据在上升沿当时钟从低变到高的时候被读取不管通讯的方向怎样键盘总是产生时钟信号，如果主机要发送数据它必须首先告诉设备开始产生时钟信号。

最大的时钟频率是 33kHz 而且大多数设备工作在 10 ~20kHz。

所有数据安排在字节中每个字节为一帧包含了 11 12 个位这些位的含义如下：

| | |
|---|---|
| 1 start bit. This is always 0. | 1 个起始位，总是为 0 |
| 8 data bits, least significant bit first. | 8 个数据位，低位在前 |
| 1 parity bit (odd parity). | 1 个校验位，奇校验 |
| 1 stop bit. This is always 1. | 1 个停止位，总是为 1 |

PS2 的通信时序：




PS2 的 ASCII 码表

| 键值 | 通码 | 断码 | 键值 | 通码 | 断码 | 键值 | 通码 | 断码 |
|---|---|---|---|---|---|---|---|---|
| A | 1C | F0,1C | 9 | 46 | F0,46 | [ | 54 | F0,54 |
| B | 32 | F0,32 | ` | 0E | F0,0E | INSERT | 67 | F0,67 |
| C | 21 | F0,21 | - | 4E | F0,4E | HOME | 6E | F0,6E |
| D | 23 | F0,23 | = | 55 | F0,55 | PG UP | 6F | F0,6F |
| E | 24 | F0,24 | \ | 5C | F0,5C | DELETE | 64 | F0,64 |
| F | 2B | F0,2B | BKSP | 66 | F0,66 | END | 65 | F0,65 |
| G | 34 | F0,34 | SPACE | 29 | F0,29 | PG DN | 6D | F0,6D |
| H | 33 | F0,33 | TAB | 0D | F0,0D | ↑ | 63 | F0,63 |
| I | 43 | F0,48 | CAPS | 14 | F0,14 | ← | 61 | F0,61 |
| J | 3B | F0,3B | L SHFT | 12 | F0,12 | ↓ | 60 | F0,60 |
| K | 42 | F0,42 | L CTRL | 11 | F0,11 | → | 6A | F0,6A |
| L | 4B | F0,4B | L WIN | 8B | F0,8B | NUM | 76 | F0,76 |
| M | 3A | F0,3A | L ALT | 19 | F0,19 | KP / | 4A | F0,4A |
| N | 31 | F0,31 | R SHFT | 59 | F0,59 | KP * | 7E | F0,7E |
| O | 44 | F0,44 | R CTRL | 58 | F0,58 | KP - | 4E | F0,4E |
| P | 4D | F0,4D | R WIN | 8C | F0,8C | KP + | 7C | F0,7C |
| Q | 15 | F0,15 | R ALT | 39 | F0,39 | KP EN | 79 | F0,79 |
| R | 2D | F0,2D | APPS | 8D | F0,8D | KP . | 71 | F0,71 |
| S | 1B | F0,1B | ENTER | 5A | F0,5A | KP 0 | 70 | F0,70 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| T | 2C | F0,2C | ESC | 08 | F0,08 | KP 1 | 69 | F0,69 |
| U | 3C | F0,3C | F1 | 07 | F0,07 | KP 2 | 72 | F0,72 |
| V | 2A | F0,2A | F2 | 0F | F0,0F | KP 3 | 7A | F0,7A |
| W | 1D | F0,1D | F3 | 17 | F0,17 | KP 4 | 6B | F0,6B |
| X | 22 | F0,22 | F4 | 1F | F0,1F | KP 5 | 73 | F0,73 |
| Y | 35 | F0,35 | F5 | 27 | F0,27 | KP 6 | 74 | F0,74 |
| Z | 1A | F0,1A | F6 | 2F | F0,2F | KP 7 | 6C | F0,6C |
| 0 | 45 | F0,45 | F7 | 37 | F0,37 | KP 8 | 75 | F0,75 |
| 1 | 16 | F0,16 | F8 | 3F | F0,3F | KP 9 | 7D | F0,7D |
| 2 | 1E | F0,1E | F9 | 47 | F0,47 | ] | 5B | F0,5B |
| 3 | 26 | F0,26 | F10 | 4F | F0,4F | ; | 4C | F0,4C |
| 4 | 25 | F0,25 | F11 | 56 | F0,56 | ' | 52 | F0,52 |
| 5 | 2E | F0,2E | F12 | 5E | F0,5E | . | 41 | F0,41 |
| 6 | 36 | F0,36 | PRNTSCRN | 57 | F0,57 | . | 49 | F0,49 |
| 7 | 3D | F0,3D | SCROLL | 5F | F0,5F | / | 4A | F0,4A |
| 8 | 3E | F0,3E | PAUSE | 62 | F0,62 | | | |

2、 LCD1602 原理

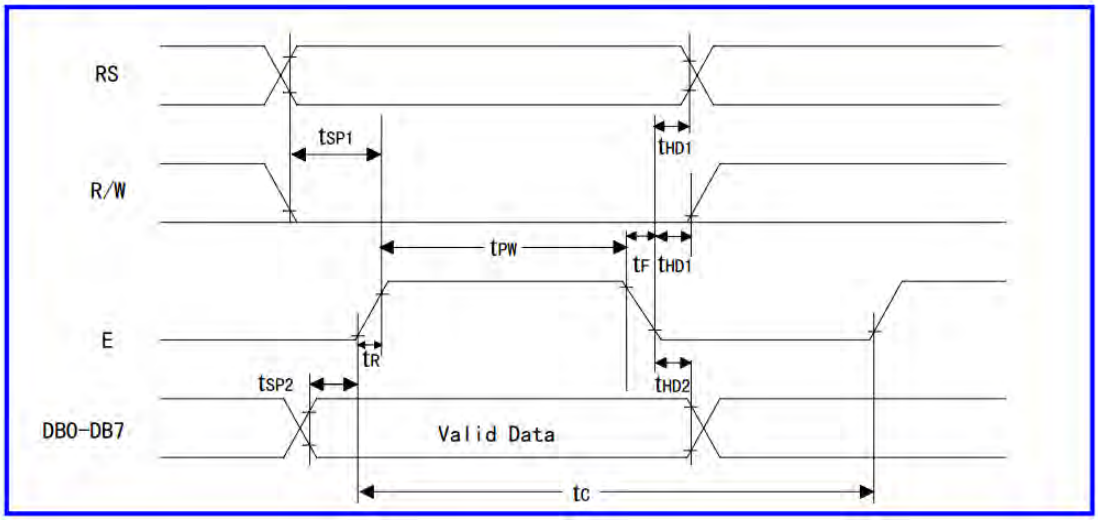
2.1、LCD1602 的物理接口

| 编号 | 符号 | 引脚说明 | 编号 | 符号 | 引脚说明 |
|---|---|---|---|---|---|
| 1 | VSS | 电源地 | 9 | D2 | Data I/O |
| 2 | VDD | 电源正极 | 10 | D3 | Data I/O |
| 3 | VL | 液晶显示偏压信号 | 11 | D4 | Data I/O |
| 4 | RS | 数据/命令选择端（H/L） | 12 | D5 | Data I/O |
| 5 | R/W | 读/写选择端（H/L） | 13 | D6 | Data I/O |
| 6 | E | 使能信号 | 14 | D7 | Data I/O |
| 7 | D0 | Data I/O | 15 | BLA | 背光源正极 |
| 8 | D1 | Data I/O | 16 | BLK | 背光源负极 |

2.2、LCD 的驱动原理

读状态：输入：RS=L, RW=H, E=H　　输出：D0～D7=状态字
写指令：输入：RS=L, RW=L, D0～D7=指令码, E=高脉冲　　输出：无
读数据：输入：RS=H, RW=H, E=H　　输出：D0～D7=数据
写数据：输入：RS=H, RW=L, D0～D7=数据, E=高脉冲　　输出：无

因为在驱动显示的过程中只需进行写操作，不用读取 LCD 的状态，所以必须读懂 LCD 的写操作时序：




| 时序参数 | 符号 | 极限值 | | | 单位 | 测试条件 |
|---|---|---|---|---|---|---|
| | | 最小值 | 典型值 | 最大值 | | |
| E 信号周期 | $t_C$ | 400 | – | – | ns | 引脚 E |
| E 脉冲宽度 | $t_{PW}$ | 150 | – | – | ns | |
| E 上升沿/下降沿时间 | $t_R$, $t_F$ | – | – | 25 | ns | |
| 地址建立时间 | $t_{SP1}$ | 30 | – | – | ns | 引脚 E、RS、R/W |
| 地址保持时间 | $t_{HD1}$ | 10 | – | – | ns | |
| 数据建立时间(读操作) | $t_D$ | – | – | 100 | ns | 引脚 DB0～DB7 |
| 数据保持时间(读操作) | $t_{HD2}$ | 20 | – | – | ns | |
| 数据建立时间(写操作) | $t_{SP2}$ | 40 | – | – | ns | |
| 数据保持时间(写操作) | $t_{HD2}$ | 10 | – | – | ns | |

根据 LCD1602 的驱动时序过程，首先必须通过写指令控制 LCD 的寄存器对 LCD 进行设置，在设置完成后再将要显示的数据写入 LCD 的寄存器中。所以整个驱动过程主要分为两个大的状态，由此可以利用状态机来对驱动时序进行控制。

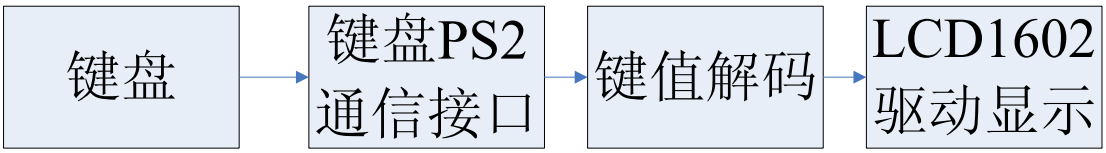
第一步：先对 LCD 进行显示模式设置，通过对寄存器写入指令：8'h38；
第二步：对 LCD 进行显示开及光标设置，通过对寄存器写入指令：8'h0C；
第三步：对 LCD 进行显示光标移动设置，通过对寄存器写入指令：8'h06；
第四步：对 LCD 进行显示清屏，通过对寄存器写入指令：8'h01；
第五步：对 LCD 进行设置第一行地址，通过对寄存器写入指令：8'h80；
在写如指令之后，就可以对 LCD 进行写数据，LCD1602 的数据位宽为 8 位，所以可以采用字符进行数据的写入。

二、方案论证

该设计主要分为两个部分：PS2 通信接口和 LCD1602 的驱动显示，由此先将两个模块独立进行验证，在独立验证通过后，再将两个模块连接起来。

PS2 通信接口由于时钟是有键盘产生，且数据在时钟的下降沿稳定，所以选择在时钟的下降沿读取数据，由于键盘对一个按下的按键有两次编码，分别为：通码和断码，而每一个数据帧为 11 位，分别为：起始位、数据位、奇偶校验位、停止位。所以在检测下降沿的同时，必须同时判断下降沿对应的数据，并准确地读取数据。由于键盘有发送断码的功能，所以可以利用该断码判断按键是否松开，以避免产生一次按下多次读取的情况。在读取数据之后再通过一个解码电路，主要是将 PS2 发送的通码转换成对应的 LCD1602 显示的字符，转换之后再将这字符发送到 LCD1602 的显示驱动模块进行显示。

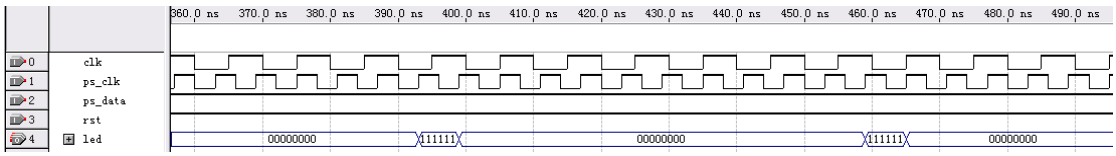
主要框架：



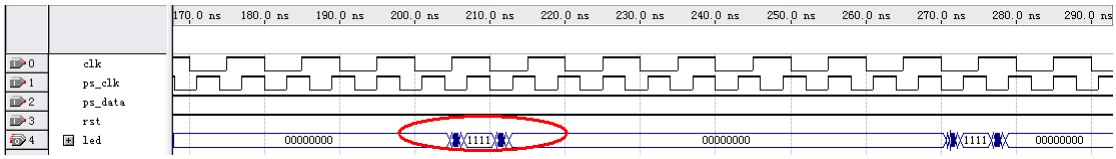

三、仿真验证

1、 PS2 通信接口仿真验证：

1.1、 功能仿真



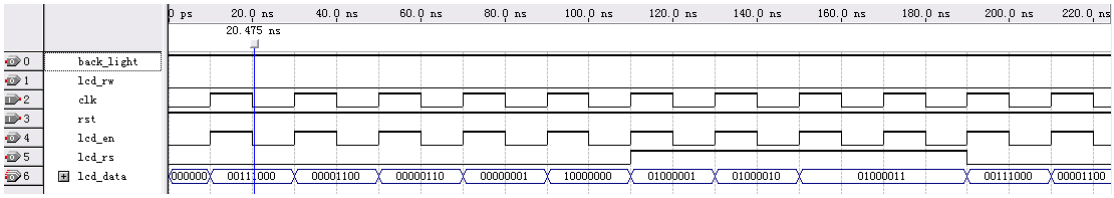

1.2、时序仿真



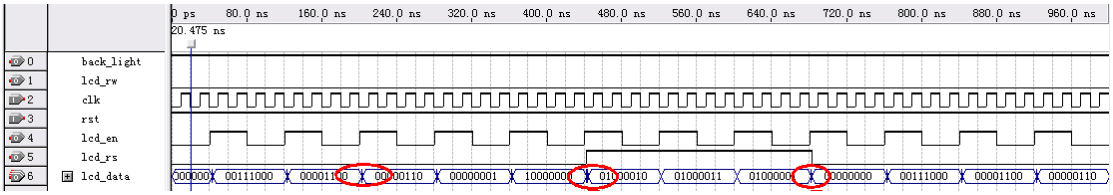

通过功能仿真和时序仿真可以看出，功能仿真是在不考虑硬件上的延迟的情况下达到的最理想的输出。而时序仿真考虑到硬件上的延迟所以，最后的输出会存在延迟，由此也可能产生竞争冒险。在该仿真中将读取的数据送往 LED 显示以便判断读取的数据是否准确，有利于硬件上的调试。

2、 LCD1602 仿真验证：

2.1、功能仿真




2.2、时序仿真

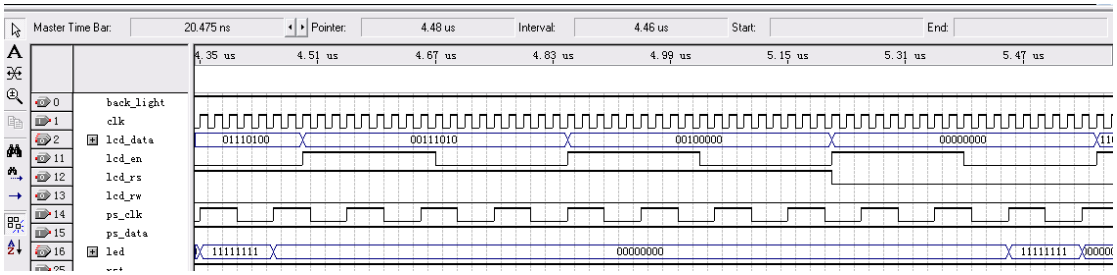



通过功能仿真和时序仿真可以看出，功能仿真是在不考虑硬件上的延迟的情况

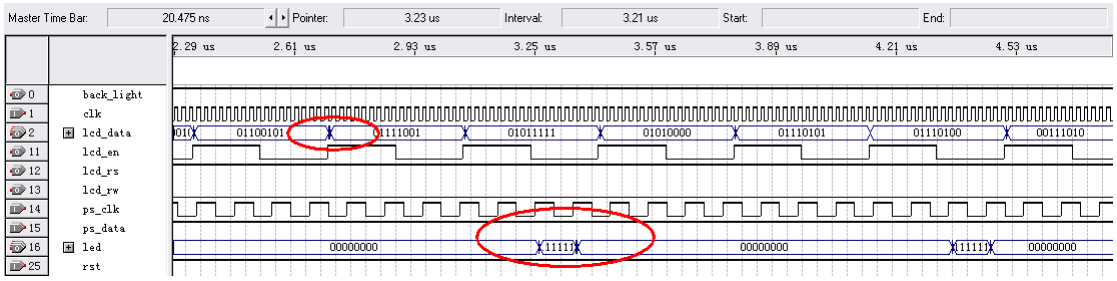
下达到的最理想的输出。而时序仿真考虑到硬件上的延迟所以，最后的输出会存在延迟，由此也可能产生竞争冒险。但是对于时序要求相对不是很严谨的 LCD1602 驱动时序，只要在每隔大于 5ms 的时候进行传送数据便可以了，对于时序仿真上的延迟锁带来的影响并不是太大。

3、 整体仿真测试：

3.1、功能仿真：




3.2、时序仿真：

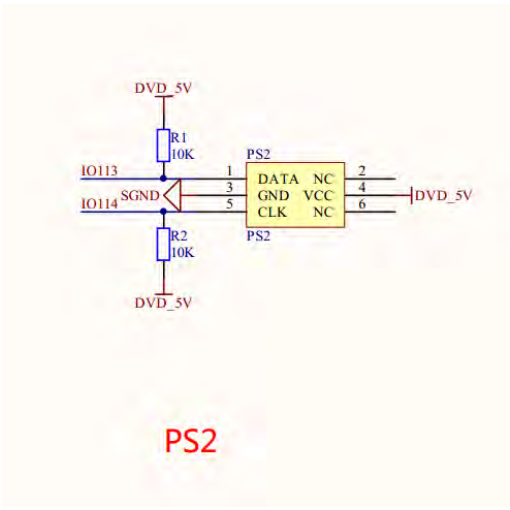



通过功能仿真和时序仿真对比可以看出，时序仿真考虑到了硬件的延迟，但对于该设计而言，硬件的延迟对于 PS2 和 LCD1602 的驱动时序没有太大的影响，在允许接受的范围内。

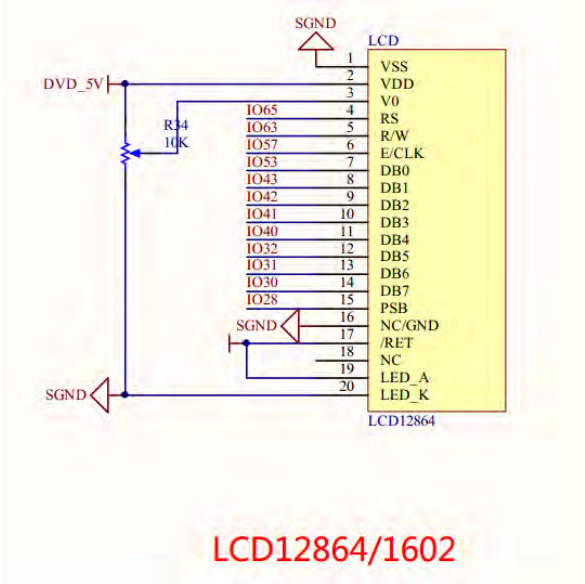
四、硬件测试

在通过仿真测试后，根据开发板的原理图进行分配引脚。

PS2 原理图：




LCD 原理图：




引脚分配：

| | Node Name | Direction | Location | I/O Bank | VREF Group | I/O Standard |
|---|---|---|---|---|---|---|
| 1 | back_light | Output | PIN_28 | 1 | B1_N1 | 3.3-V LVTTL (default) |
| 2 | clk | Input | PIN_17 | 1 | B1_N0 | 3.3-V LVTTL (default) |
| 3 | lcd_data[7] | Output | PIN_30 | 1 | B1_N1 | 3.3-V LVTTL (default) |
| 4 | lcd_data[6] | Output | PIN_31 | 1 | B1_N1 | 3.3-V LVTTL (default) |
| 5 | lcd_data[5] | Output | PIN_32 | 1 | B1_N1 | 3.3-V LVTTL (default) |
| 6 | lcd_data[4] | Output | PIN_40 | 4 | B4_N1 | 3.3-V LVTTL (default) |
| 7 | lcd_data[3] | Output | PIN_41 | 4 | B4_N1 | 3.3-V LVTTL (default) |
| 8 | lcd_data[2] | Output | PIN_42 | 4 | B4_N1 | 3.3-V LVTTL (default) |
| 9 | lcd_data[1] | Output | PIN_43 | 4 | B4_N1 | 3.3-V LVTTL (default) |
| 10 | lcd_data[0] | Output | PIN_53 | 4 | B4_N1 | 3.3-V LVTTL (default) |
| 11 | lcd_en | Output | PIN_57 | 4 | B4_N1 | 3.3-V LVTTL (default) |
| 12 | lcd_rs | Output | PIN_65 | 4 | B4_N0 | 3.3-V LVTTL (default) |
| 13 | lcd_rw | Output | PIN_63 | 4 | B4_N0 | 3.3-V LVTTL (default) |
| 14 | led[7] | Output | PIN_8 | 1 | B1_N0 | 3.3-V LVTTL (default) |
| 15 | led[6] | Output | PIN_9 | 1 | B1_N0 | 3.3-V LVTTL (default) |
| 16 | led[5] | Output | PIN_4 | 1 | B1_N0 | 3.3-V LVTTL (default) |
| 17 | led[4] | Output | PIN_3 | 1 | B1_N0 | 3.3-V LVTTL (default) |
| 18 | led[3] | Output | PIN_144 | 2 | B2_N1 | 3.3-V LVTTL (default) |
| 19 | led[2] | Output | PIN_143 | 2 | B2_N1 | 3.3-V LVTTL (default) |
| 20 | led[1] | Output | PIN_142 | 2 | B2_N1 | 3.3-V LVTTL (default) |
| 21 | led[0] | Output | PIN_141 | 2 | B2_N1 | 3.3-V LVTTL (default) |
| 22 | ps_clk | Input | PIN_114 | 2 | B2_N0 | 3.3-V LVTTL (default) |
| 23 | ps_data | Input | PIN_113 | 2 | B2_N0 | 3.3-V LVTTL (default) |
| 24 | rst | Input | PIN_64 | 4 | B4_N0 | 3.3-V LVTTL (default) |

整个设计共用的资源：

| | |
|---|---|
| Flow Status | Successful - Fri Dec 14 12:33:07 2012 |
| Quartus II Version | 9.0 Build 132 02/25/2009 SJ Full Version |
| Revision Name | ps2_lcd |
| Top-level Entity Name | ps2_lcd |
| Family | Cyclone II |
| Device | EP2C5T144C8 |
| Timing Models | Final |
| Met timing requirements | Yes |
| Total logic elements | 122 / 4,608 ( 3 % ) |
| Total combinational functions | 120 / 4,608 ( 3 % ) |
| Dedicated logic registers | 85 / 4,608 ( 2 % ) |
| Total registers | 85 |
| Total pins | 24 / 89 ( 27 % ) |
| Total virtual pins | 0 |
| Total memory bits | 1,536 / 119,808 ( 1 % ) |
| Embedded Multiplier 9-bit elements | 0 / 26 ( 0 % ) |
| Total PLLs | 0 / 2 ( 0 % ) |

程序：

```
/*********************************************************************************/
//LCD1602 的显示
/*********************************************************************************/
module lcd1602(clk,rst,lcd_rs,lcd_en,lcd_rw,lcd_data,back_light,ps_datain);
input clk,rst;//时钟、复位
input [7:0] ps_datain;//PS2 数据输入
output lcd_rs,lcd_en,lcd_rw,back_light;//指令/数据控制、片选、读写控制、背光控制
output [7:0] lcd_data;//数据线
reg lcd_rs;//指令/数据控制
reg [7:0] lcd_data;//数据线
reg [7:0] current_state;//当前状态
reg [1:0] state_counter;//状态计数
reg en_temp;//使能标志
reg [15:0] clk_counter;//时钟计数
reg clk_en;//时钟使能
//reg [7:0] ps_datain_temp;
assign back_light=1'b1;//一直设置为高电平
assign lcd_rw=1'b0;//一直为写状态
/*********************状态编码*******************************/
parameter set0=8'b0000_0000,
          set1=8'b0000_0001,
          set2=8'b0000_0011,
          set3=8'b0000_0100,
          set4=8'b0000_0101,
          data1=8'b0000_1000,
          data2=8'b0000_1001,
          data3=8'b0000_1010,
          data4=8'b0000_1011,
```

```
                data5=8'b0000_1100,
                data6=8'b0000_1101,
                data7=8'b0000_1110,
                data8=8'b0000_1111,
                data9=8'b0001_0000,
                stop=8'b1111_1111;
/**************状态转换时钟************************/
always @(posedge clk)
  begin
    if(clk_counter==16'h8000)
       begin
         clk_counter<=16'h0;
         clk_en<=~clk_en;
       end
    else
       clk_counter<=clk_counter+1'b1;
  end
/**************状态转换**************************///
always @(posedge clk_en or negedge rst)
  begin
     if(!rst)
        begin
          current_state<=set0;
        end
     else
        begin
           case (current_state)
/*********************************************************************************/
           set0:begin lcd_rs<=1'b0;lcd_data<=8'h38;current_state<=set1; end//显示模式设置
           set1:begin  lcd_rs<=1'b0;lcd_data<=8'h0c;current_state<=set2;  end//显示开及光标设
置
           set2:begin  lcd_rs<=1'b0;lcd_data<=8'h06;current_state<=set3;  end//显示光标移动设
置
           set3:begin lcd_rs<=1'b0;lcd_data<=8'h01;current_state<=set4; end//显示清屏
           set4:begin lcd_rs<=1'b0;lcd_data<=8'h80;current_state<=data1; end//设置第一行地址
/*********************************************************************************/
           data1:begin lcd_rs<=1'b1;lcd_data<="K";current_state<=data2; end//显示第 1 个字符
           data2:begin lcd_rs<=1'b1;lcd_data<="e";current_state<=data3; end//显示第 2 个字符
           data3:begin lcd_rs<=1'b1;lcd_data<="y";current_state<=data4; end//显示第 3 个字符
           data4:begin lcd_rs<=1'b1;lcd_data<="_";current_state<=data5; end//显示第 4 个字符
           data5:begin lcd_rs<=1'b1;lcd_data<="P";current_state<=data6; end//显示第 5 个字符
           data6:begin lcd_rs<=1'b1;lcd_data<="u";current_state<=data7; end//显示第 6 个字符
           data7:begin lcd_rs<=1'b1;lcd_data<="t";current_state<=data8; end//显示第 7 个字符
```

```
            data8:begin lcd_rs<=1'b1;lcd_data<=":";current_state<=data9; end//显示第 8 个字符
            data9:begin  lcd_rs<=1'b1;lcd_data<=ps_datain;current_state<=stop;  end//显示第 9 个
字                                                                                    符
/*****************************************************************************/
            stop:begin //控制指令与数据写入的次数
                        lcd_rs<=1'b0;
                        lcd_data<=8'b0000_0000;
                        if(state_counter!=2'b10)
                           begin
                              en_temp<=1'b0;
                              current_state<=set0;
                              state_counter<=state_counter+1'b1;
                           end
                        else
                            begin
                               current_state<=set0;
                               en_temp<=1'b0;//最后数据写入完成后将 lcd_en 线拉高
                            end
                  end
            default:    current_state<=set0;
          endcase
       end
 end
 assign lcd_en=clk_en|en_temp;//lcd_en 为‘1’有效

endmodule
/*****************************************************************************/
//PS2
/*****************************************************************************/
module PS2(clk,rst,ps_clk,ps_data,dataout,led);
input clk,rst,ps_clk,ps_data;
output [7:0] dataout,led;
reg [7:0] dataout;
wire [7:0] dataout_temp;
reg [3:0] num;//PS_CLK 时钟计数
reg [10:0] data_buffer;//接收数据缓存器
reg [7:0] data_reg1,data_reg2,data_reg3,result;
/*****************计数识别 PS_CLK 的时钟位置*******************/
always @(posedge ps_clk)
 begin
    if(num==4'd10)
          num<=4'd0;
    else
```

```
        num<=num+4'd1;
  end
/***************在 PS_CLK 的下降沿读取数据*********************/
always @(negedge ps_clk)
     begin
        data_buffer[num]<=ps_data;//下降沿读取数据
     end
/***************读取数据帧并判断是否按键是否放开*************/
always @(posedge ps_clk or negedge rst)
   begin
     if(!rst)
        result<=8'b0;
     else
       begin
       case (num)
          4'd9:data_reg1<=data_buffer[8:1];//读取数据
          4'd10:
                  begin
                      if((data_reg1==data_reg2) && (data_reg2==data_reg3))//一直按住某个按键
                            result<=data_reg1;
                      else
                        begin
                          if((data_reg1==data_reg3) && (data_reg2==8'b11110000)) //11110000 为断
码标志,触发某个按键
                              begin
                                result<=data_reg1;
                                data_reg2<=8'b0;
                                data_reg3<=8'b0;
                              end
                         else
                           begin
                            result<=8'b0;
                            data_reg3<=data_reg2; //将上一次的键码进行寄存
                            data_reg2<=data_reg1;//将上一次的键码进行寄存
                           end
                        end
                  end
          default:result<=8'b0;
       endcase
     end
   end
assign dataout_temp=result;
assign led=result;
```

```
/******************键值寻码*****************************/
always @(posedge clk)
  begin
     case (dataout_temp)
        8'h1c:dataout<="A";//A
        8'h32:dataout<="B";//B
        8'h21:dataout<="C";//C
        8'h23:dataout<="D";//D
        8'h24:dataout<="E";//E
        8'h2b:dataout<="F";//F
        8'h34:dataout<="G";//G
        8'h33:dataout<="H";//H
        8'h43:dataout<="I";//I
        8'h3b:dataout<="J";//J
        8'h42:dataout<="K";//K
        8'h4b:dataout<="L";//L
        8'h3a:dataout<="M";//M
        8'h31:dataout<="N";//N
        8'h44:dataout<="O";//O
        8'h4d:dataout<="P";//P
        8'h15:dataout<="Q";//Q
        8'h2d:dataout<="R";//R
        8'h1b:dataout<="S";//S
        8'h2c:dataout<="T";//T
        8'h3c:dataout<="U";//U
        8'h2a:dataout<="V";//V
        8'h1d:dataout<="W";//W
        8'h22:dataout<="X";//X
        8'h35:dataout<="Y";//Y
        8'h1a:dataout<="Z";//Z
        8'h45:dataout<="0";//0
        8'h16:dataout<="1";//1
        8'h1e:dataout<="2";//2
        8'h26:dataout<="3";//3
        8'h25:dataout<="4";//4
        8'h2e:dataout<="5";//5
        8'h36:dataout<="6";//6
        8'h3d:dataout<="7";//7
        8'h3e:dataout<="8";//8
        8'h46:dataout<="9";//9
       default:dataout<=" ";//
     endcase
  end
```

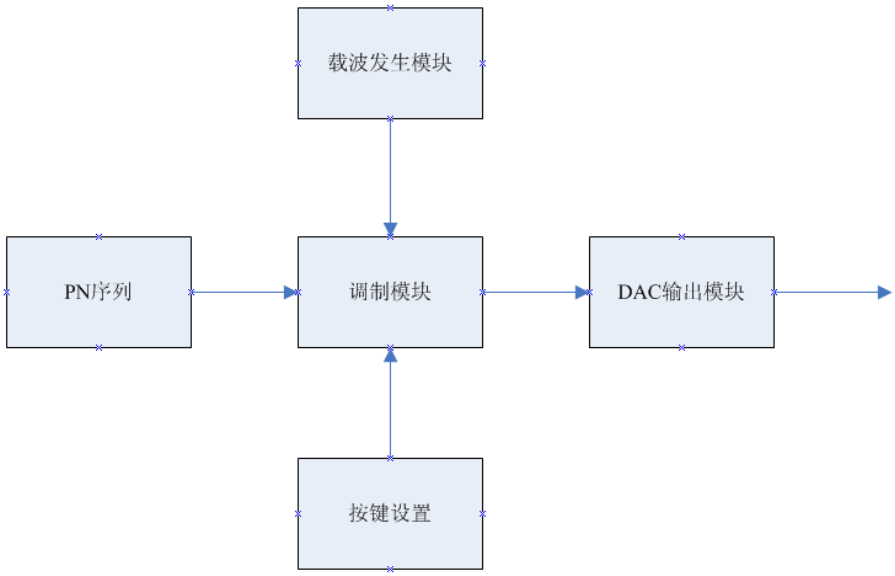
```
endmodule
/**************************************************************************************/
//基于 PS2 的 LCD 显示的顶层设计
/**************************************************************************************/
module ps2_lcd(clk,rst,ps_clk,ps_data,lcd_rs,lcd_en,lcd_rw,lcd_data,back_light,led);
input clk,rst,ps_clk,ps_data;
output lcd_rs,lcd_en,lcd_rw,back_light;
output [7:0] lcd_data,led;
wire [7:0] data_line;
PS2 u1(.clk(clk),.rst(rst),.ps_clk(ps_clk),.ps_data(ps_data),.dataout(data_line),.led(led));
lcd1602 u2(.clk(clk),.rst(rst),.lcd_rs(lcd_rs),.lcd_en(lcd_en),.lcd_rw(lcd_rw),
                .lcd_data(lcd_data),.back_light(back_light),.ps_datain(data_line));
Endmodule
```

## 13.3 基于 FPGA 的通信信号源的设计

# 1、通信信号源设计原理

通过设计一个 DDS 信号源，然后将该信号作为载波信号，再对基带信号进行 2ASK、2FSK、2PSK、2DPSK 调制，进而产生多种通信信号。

设计框图如下：




将 PN 序列进行 2ASK、2FSK、2PSK、2DPSK 调制，其中载波发生器提供三种不同的载波信号，按键设置用来选择当前 DAC 输出的最终调制信号，DAC 输出模块将调制好的数字化波形转换为模拟信号。

PN 序列采用的设计方法是 m 序列，采用 3 级寄存器生成 7 序列基带信号。

调制模块分别包含：2ASK、2FSK、2PSK、2DPSK 这四种调制方式，可以通过按键选择所要输出的调制信号。

按键设置,主要分为按键消抖和按键编码，用来切换输出信号。

载波发生器，采用 DDS 的方法进行生成三种载波信号，分别为：500Hz（起始相位为 0 度），1K（起始相位为 0 度），1K（起始相位为 180 度）。

DAC 输出模块，这是驱动 TLC5615 的硬件接口，用于将数字信号转换位模拟信号。

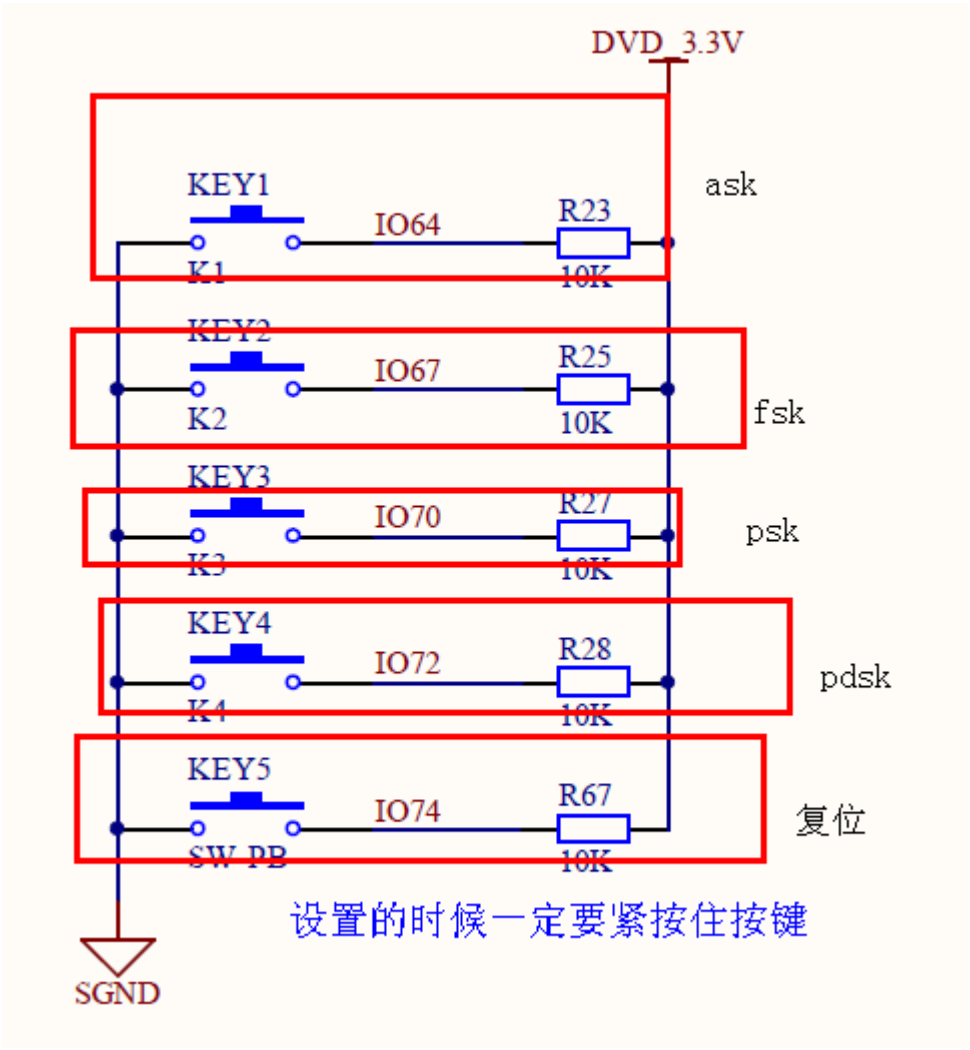
# 2、硬件原理图说明

**按键操作：**

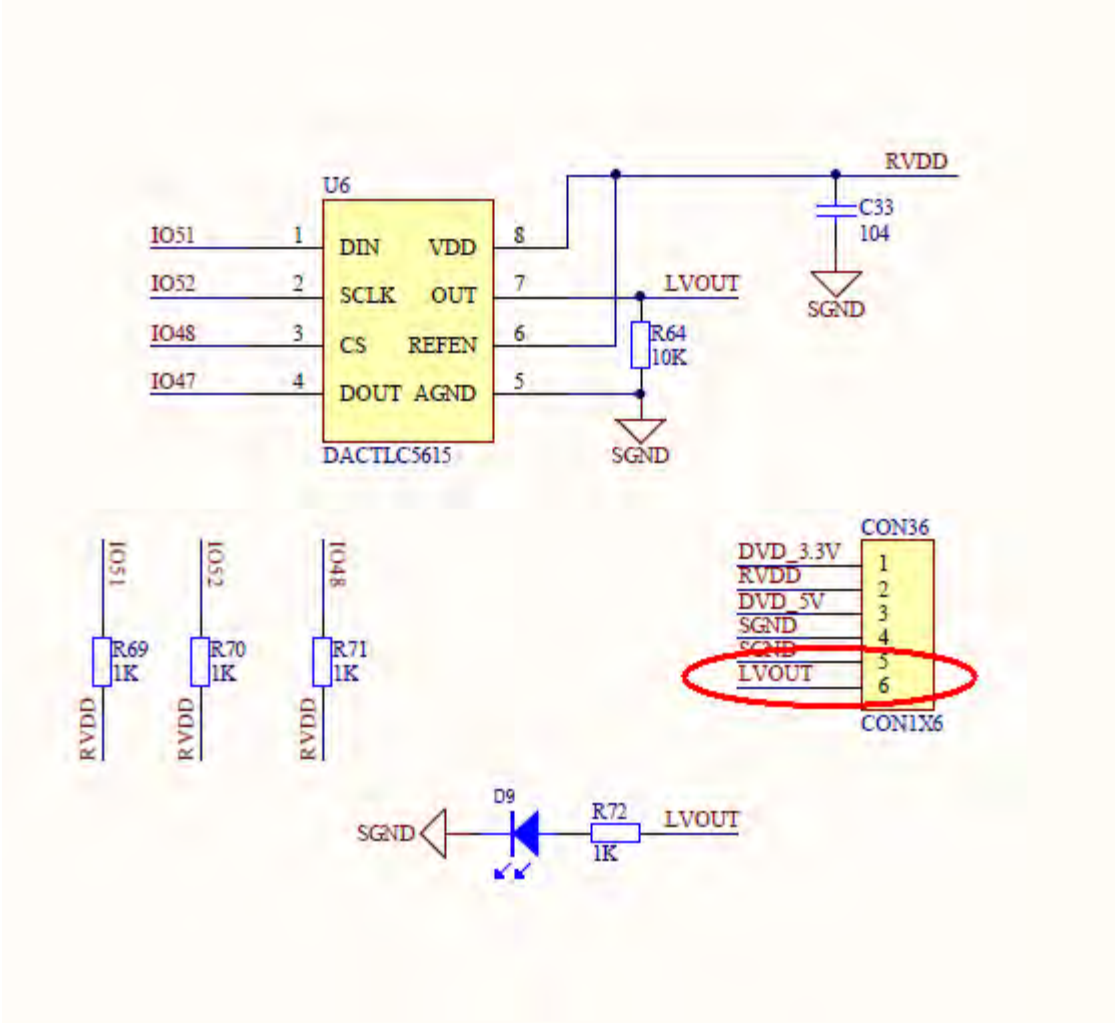
```
4   **-----------------------------文件信息--------------------------
5   ** 文件名称:  DDS_top.v
6   ** 创建日期:
7   ** 功能描述: 顶层设计
8   ** 硬件平台: 大西瓜第一代开发板 http://daxiguafpga.taobao.com
9   ** 版权声明: 本代码属个人知识产权,本代码仅供交流学习.
10  **-----------------------------修改文件的相关信息----------------
11  ** 修改人:
12  ** 修改日期:
13  ** 描述:PN序列频率为500HZ
14          载波1频率为500HZ
15          载波2频率为1000Hz
16          载波3频率为1000Hz相位相差180
17  ********************************************************************************/
18  module DDS_top(clk,                 //内部时钟
19                 reset,               //复位
20                 sclk,                //DAC时钟输入
21                 din,                 //DAC数据输入
22                 cs,                  //DAC片选
23                 m_ser_out,           //PN序列输出,25引脚
24                 set_ask,             //设置输出为ask
25                 set_fsk,             //设置输出为fsk
26                 set_psk,             //设置输出为psk
27                 set_dpsk,            //设置输出为pdsk
28                 dpsk_code_out        //pdsk调制输出,27引脚
29                 );
30  input          clk;
31  input          reset;
32  input          sclk;
```

DAC驱动信号

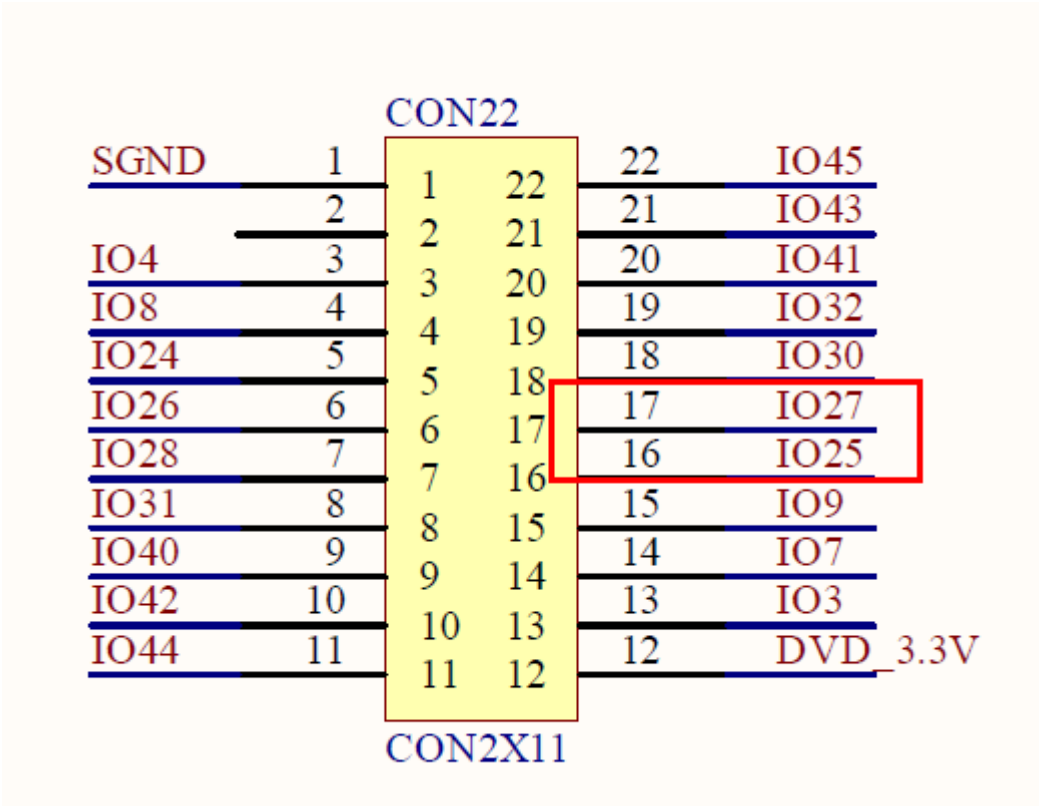
设置按键，设置时要一直紧按住按键




波形输出：




示波器要接 LVOUT 这个排针的引脚，旁边那个就是地；输出的是经过调制的正弦信号！




25 是输出 PN 序列，27 是输出 pdsk 调制。
将这两个信号接入示波器可以和前面的 DAC 输出进行对比。

## 3、实验代码

- **顶层接口模块：**

```
module DDS_top(clk,                    //内部时钟
               reset,                  //复位
               sclk,                   //DAC 时钟输入
               din,                    //DAC 数据输入
               cs,                     //DAC 片选
               m_ser_out,              //PN 序列输出,25 引脚
               set_ask,                //设置输出为 ask
               set_fsk,                //设置输出为 fsk
               set_psk,                //设置输出为 psk
               set_dpsk,               //设置输出为 pdsk
               dpsk_code_out           //pdsk 调制输出,27 引脚
              );
input          clk;
input          reset;
output         sclk;
output         din;
output         cs;
```

```
input            set_ask;
input             set_fsk;
input             set_psk;
input             set_dpsk;
output           m_ser_out;
output           dpsk_code_out;
wire             clk;
wire             reset;
wire             set_ask_line;
wire        set_fsk_line;
wire        set_psk_line;
wire        set_dpsk_line;
wire             clk_div;
wire [9:0]    data_line1;
wire [9:0]    data_line2;
wire [9:0]    data_line3;
wire [9:0]    ask_code_sin_out;
wire [9:0]    fsk_code_sin_out;
wire [9:0]    psk_code_sin_out;
wire [9:0]    dpsk_code_sin_out;
wire [9:0]    dac_data_in;

  DDS       u4(.clk(clk),
               .reset_n(reset),
               .dds_data_out1(data_line1),
               .dds_data_out2(data_line2),
               .dds_data_out3(data_line3));
  m_ser     u5(
              .clk(clk),                //sys clk
              .reset_n(reset),          //sys reset_n
              .clk_div(clk_div),
              .m_ser_out(m_ser_out)//PN ser_code out
              );
  ask_code u6(
                    .clk(clk),
                    .m_ser_code_in(m_ser_out),
                    .dds_sin_data_in(data_line2),//10k 正弦波
                    .ask_code_sin_out(ask_code_sin_out)
                    );
  fsk_code u7(
                    .clk(clk),
                    .m_ser_code_in(m_ser_out),
                    .dds_sin_data_in1(data_line1),//1k 正弦波
```

```
                .dds_sin_data_in2(data_line2),//10k 正弦波
                .fsk_code_sin_out(fsk_code_sin_out)
                );
  psk_code u8(
                .clk(clk),
                .m_ser_code_in(m_ser_out),
                .dds_sin_data_in2(data_line2),//10k 正弦波
                .dds_sin_data_in3(data_line3),//10k 正弦波
                .psk_code_sin_out(psk_code_sin_out)
                );
  dpsk_code u9(
                 .clk(clk_div),                  //sys 500Hz
                 .reset_n(reset),              //
                 .m_ser_code_in(m_ser_out),     //PN 序列输入
                 .dpsk_code_out(dpsk_code_out),     //dpsk 调制输出
                 .dds_sin_data_in2(data_line2),//10k 正弦波
                 .dds_sin_data_in3(data_line3),//10k 正弦波,相位相差 180
                 .dpsk_code_sin_out(dpsk_code_sin_out)//
                 );
  key          u10(.clk(clk),.key(set_ask),.key_out(set_ask_line));
  key          u11(.clk(clk),.key(set_fsk),.key_out(set_fsk_line));
  key          u12(.clk(clk),.key(set_psk),.key_out(set_psk_line));
  key          u13(.clk(clk),.key(set_dpsk),.key_out(set_dpsk_line));
  key_coding u14(
                  .clk(clk),
                  .reset_n(reset),
                  .set_ask(set_ask_line),
                  .set_fsk(set_fsk_line),
                  .set_psk(set_psk_line),
                  .set_dpsk(set_dpsk_line),
                  .ask_code_sin_out(ask_code_sin_out), //
                  .fsk_code_sin_out(fsk_code_sin_out), //
                  .psk_code_sin_out(psk_code_sin_out), //
                  .dpsk_code_sin_out(dpsk_code_sin_out),//
                  .code_data_out(dac_data_in));        //
  TLC5615      u15(
                  .clk(clk),
                  .sclk(sclk),
                  .din(din),
                  .cs(cs),
                  .din_in(dac_data_in));
endmodule
```

- 按键编码模块：

```
module key_coding(
                        clk,
                        reset_n,
                        set_ask,
                        set_fsk,
                        set_psk,
                        set_dpsk,
                        ask_code_sin_out, //
                        fsk_code_sin_out, //
                        psk_code_sin_out, //
                        dpsk_code_sin_out,//
                        code_data_out        //
);
input              clk;
input              reset_n;
input              set_ask;
input              set_fsk;
input              set_psk;
input              set_dpsk;
input    [9:0]   ask_code_sin_out;//
input    [9:0]   fsk_code_sin_out;//
input    [9:0]   psk_code_sin_out;//
input    [9:0]   dpsk_code_sin_out;//
output [9:0]   code_data_out;
wire             reset;
wire             set_ask;
wire             set_fsk;
wire             set_psk;
wire             set_dpsk;
wire [9:0]    ask_code_sin_out;//
wire [9:0]    fsk_code_sin_out;//
wire [9:0]    psk_code_sin_out;//
wire [9:0]    dpsk_code_sin_out;//
reg    [9:0]   code_data_out;

wire [3:0]     set_mode_code;
/******************调制模式选择*********************************************/
assign set_mode_code={set_ask,set_fsk,set_psk,set_dpsk};
always @(posedge clk or negedge reset_n)
  begin
    if(!reset_n)
```

```
    begin
      code_data_out<=10'd0;
    end
   else
    case (set_mode_code)
      4'b0111:code_data_out<=ask_code_sin_out;
      4'b1011:code_data_out<=fsk_code_sin_out;
      4'b1101:code_data_out<=psk_code_sin_out;
      4'b1110:code_data_out<=dpsk_code_sin_out;
      default:code_data_out<=ask_code_sin_out;
    endcase
 end
endmodule
```

- **按键消抖模块：**

```
module key(
                clk,
                key,
                key_out);
input clk;
input key; //时钟输入，按键输入
output key_out;//经消抖后的按键信号输出
wire clk;
wire key;
reg    key_out;
parameter s0=2'b00,s1=2'b01,s2=2'b10,s3=2'b11;
reg [1:0] state;
always @(posedge clk)
 begin
      case (state)
        s0:
          begin
            key_out<=1'b1;
            if(key==1'b0)
                state<=s1;
            else
                state<=s0;
          end
        s1:
          begin
            if(key==1'b0)
               state<=s2;
            else
               state<=s0;
```

```
            end
        s2:
          begin
            if(key==1'b0)
               state<=s3;
            else
               state<=s0;
            end
        s3:
            begin
              if(key==1'b0)
                begin
                key_out<=1'b0;
                state<=s3;
                end
              else
                begin
                key_out<=1'b1;
                state<=s0;
                end
            end
         default:
               state<=s0;
      endcase
  end
endmodule
```

- **DDS 模块：**

```
module DDS(
clk,
reset_n,
dds_data_out1,
dds_data_out2,
dds_data_out3
);
input              clk;//时钟输入
input              reset_n;
output [9:0]    dds_data_out1;
output [9:0]    dds_data_out2;
output [9:0]    dds_data_out3;
wire               clk;
wire               reset_n;
wire       [9:0]   dds_data_out1;
wire       [9:0]   dds_data_out2;
```

```
wire       [9:0]    dds_data_out3;
//*****************************************************//
//******************产生载波 1(500)*********************//
//*****************************************************//
/**************连接线***************/
wire     [31:0] f32_bus1;//AC 频率控制字输入
wire     [9:0]    p_bus1;
wire     [31:0] reg32_out1;//32 位寄存器输出
wire     [31:0] reg32_in1;//32 位寄存器输入
wire     [9:0]    reg10_in1;
wire     [9:0]    reg10_out_address1;
/************************************/
parameter [11:0] f32_bus_init1=12'd0;
parameter [9:0] p10_bus_init1=10'd0;//设置初始相位
assign f32_bus1[31:20]=f32_bus_init1;//初始化,高位置低
assign f32_bus1[19:0]=20'd42950;//低位可以设置 DDS 的输出频率
assign p_bus1=p10_bus_init1;
/**********************元件例化*************************************/
adder_32 u1(.data1(f32_bus1),.data2(reg32_out1),.sum(reg32_in1));
reg32       u2(.clk(clk),.reset_n(reset_n),.data_in(reg32_in1),.data_out(reg32_out1));
adder_10 u3(.data1(p_bus1),.data2(reg32_out1[31:22]),.sum(reg10_in1));
reg_10      u4(.clk(clk),.reset_n(reset_n),.data_in(reg10_in1),.data_out(reg10_out_address1));
sin_rom     u5(.address(reg10_out_address1),.clock(clk),.q(dds_data_out1));//正弦
//*****************************************************//
//******************产生载波 2(1K)*********************//
//*****************************************************//
/**************连接线***************/
wire     [31:0] f32_bus2;//AC 频率控制字输入
wire     [9:0]    p_bus2;
wire     [31:0] reg32_out2;//32 位寄存器输出
wire     [31:0] reg32_in2;//32 位寄存器输入
wire     [9:0]    reg10_in2;
wire     [9:0]    reg10_out_address2;
/************************************/
parameter [11:0] f32_bus_init2=12'd0;//
parameter [9:0] p10_bus_init2=10'd0;//设置初始相位
assign f32_bus2[31:20]=f32_bus_init2;//初始化,高位置低
assign f32_bus2[19:0]=20'd85899;//低位可以设置 DDS 的输出频率
assign p_bus2=p10_bus_init2;
/**********************元件例化*************************************/
adder_32 u6(.data1(f32_bus2),.data2(reg32_out2),.sum(reg32_in2));
reg32       u7(.clk(clk),.reset_n(reset_n),.data_in(reg32_in2),.data_out(reg32_out2));
adder_10 u8(.data1(p_bus2),.data2(reg32_out2[31:22]),.sum(reg10_in2));
```

```
reg_10     u9(.clk(clk),.reset_n(reset_n),.data_in(reg10_in2),.data_out(reg10_out_address2));
sin_rom    u10(.address(reg10_out_address2),.clock(clk),.q(dds_data_out2));//正弦
//*****************************************************//
//******************产生载波 3(1K,相位与 2 相反)**********************//
//*****************************************************//
/**************连接线***************/
wire     [31:0] f32_bus3;//AC 频率控制字输入
wire     [9:0]    p_bus3;
wire     [31:0] reg32_out3;//32 位寄存器输出
wire     [31:0] reg32_in3;//32 位寄存器输入
wire     [9:0]    reg10_in3;
wire     [9:0]    reg10_out_address3;
/***********************************/
parameter [11:0] f32_bus_init3=12'd0;//
parameter [9:0] p10_bus_init3=10'd512;//设置初始相位
assign f32_bus3[31:20]=f32_bus_init3;//初始化,高位置低
assign f32_bus3[19:0]=20'd85899;//低位可以设置 DDS 的输出频率
assign p_bus3=p10_bus_init3;
/**********************元件例化************************************/
adder_32 u11(.data1(f32_bus3),.data2(reg32_out3),.sum(reg32_in3));
reg32      u12(.clk(clk),.reset_n(reset_n),.data_in(reg32_in3),.data_out(reg32_out3));
adder_10 u13(.data1(p_bus3),.data2(reg32_out3[31:22]),.sum(reg10_in3));
reg_10     u14(.clk(clk),.reset_n(reset_n),.data_in(reg10_in3),.data_out(reg10_out_address3));
sin_rom    u15(.address(reg10_out_address3),.clock(clk),.q(dds_data_out3));//正弦
endmodule
```

- **m 序列生成模块：**

```
module m_ser(
                clk,          //sys clk
                reset_n,    //sys reset_n
                clk_div,
                m_ser_out //PN ser_code out
                );
input        clk;
input        reset_n;
output       clk_div;
output       m_ser_out;
wire         clk;
wire         reset_n;
reg          m_ser_out;
reg [2:0] m_code;
reg          clk_div;
reg [17:0]clk_cnt;
always @(posedge clk or negedge reset_n)
```

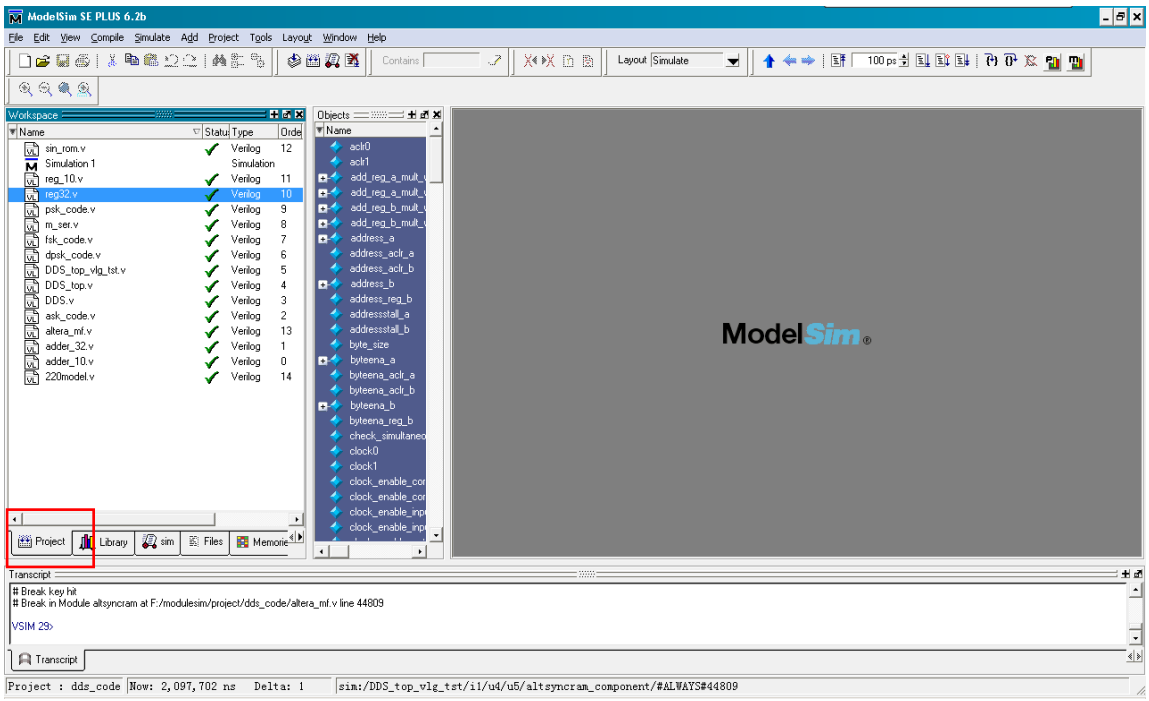
```
 begin
  if(!reset_n)
    begin
      clk_div<=1'b0;
      clk_cnt<=18'd0;
    end
  else
    if(clk_cnt==18'd49999)//500Hz PNcode
      begin
        clk_cnt<=18'd0;
        clk_div<=~clk_div;
      end
    else
      clk_cnt<=clk_cnt+1'b1;
 end

always @(posedge clk_div or negedge reset_n)
 begin
  if(!reset_n)
    begin
      m_code<=3'b001;          //置数初始化
      m_ser_out<=1'b0;
    end
  else
      begin
        m_code[2:1]<=m_code[1:0];
        m_code[0]<=m_code[2] ^ m_code[0];//将 2 和 0 进行异或然后放到 0
        m_ser_out<=m_code[2];//将 2 进行输出
      end
 end
endmodule
```

## 4、ModelSim 仿真过程

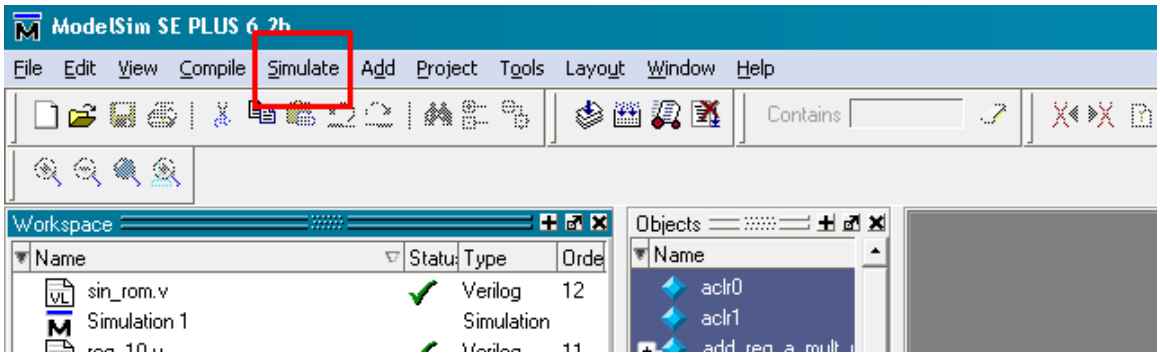
安装 ModelSim 软件之后，打开该文件夹


打开红色那个文件

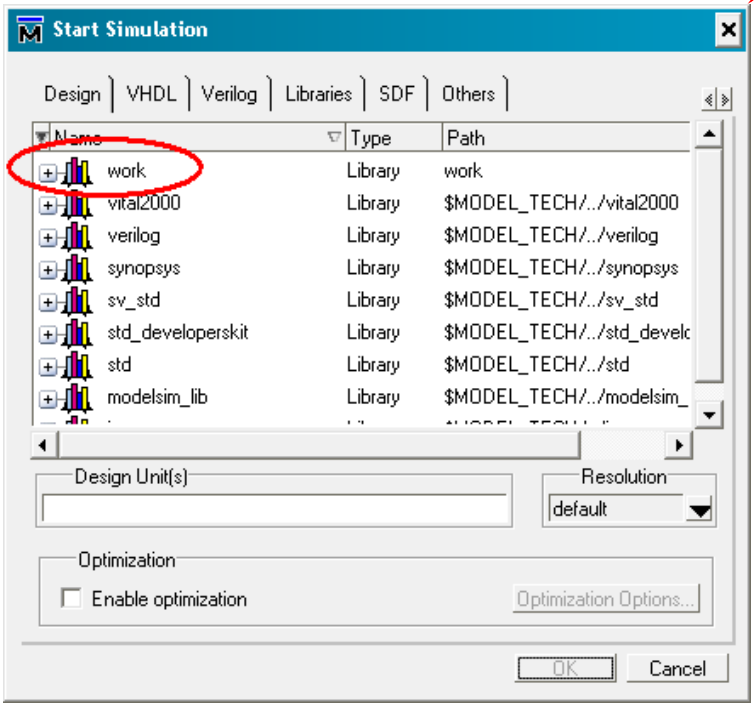

点击 project 出现了仿真的文件



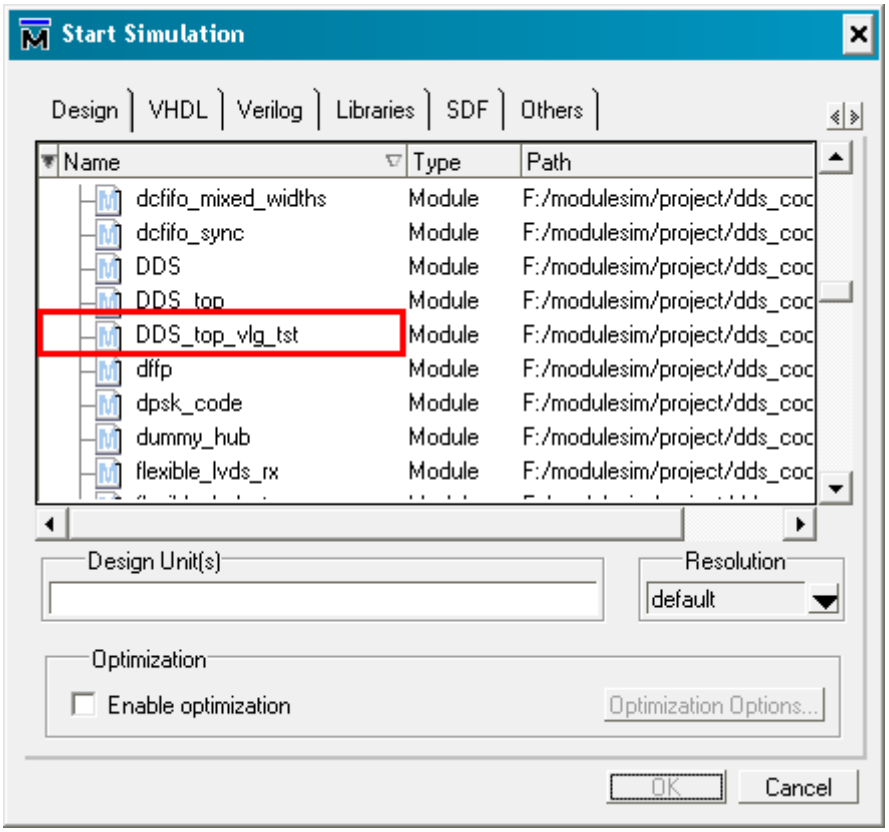

点击：start simulation

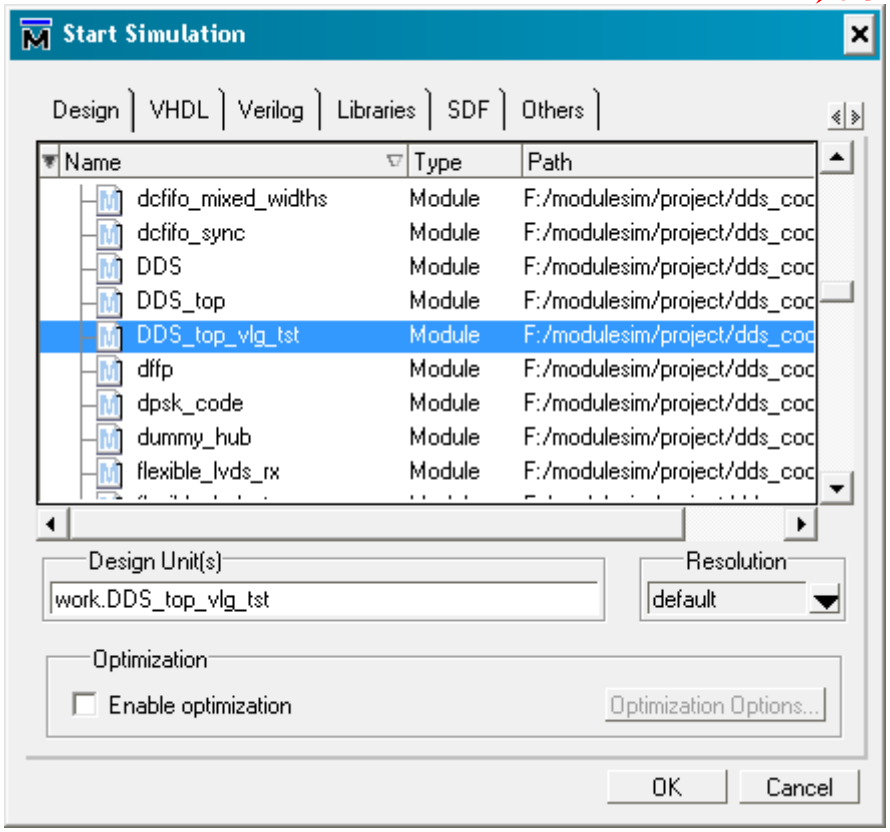



选择 work



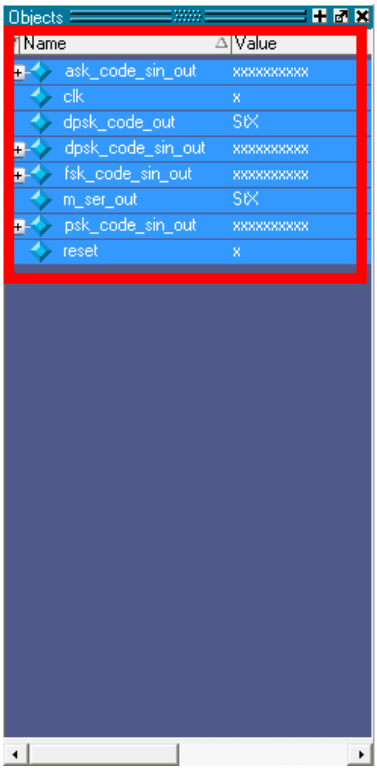

选择这个文件，然后点击 OK

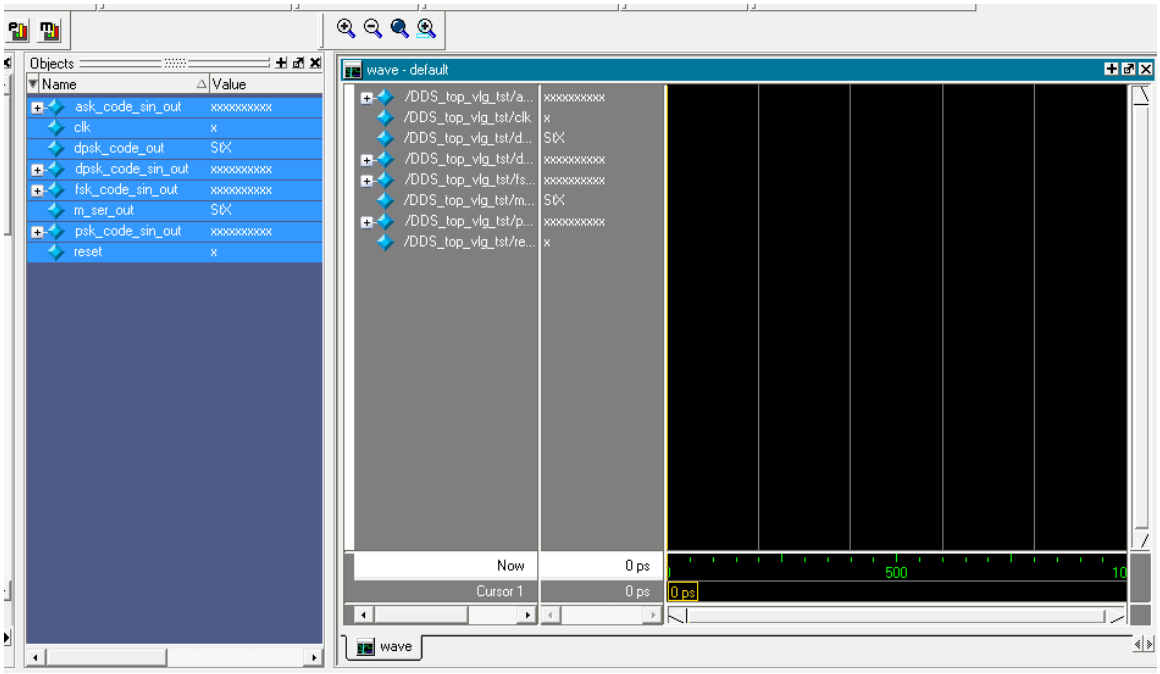




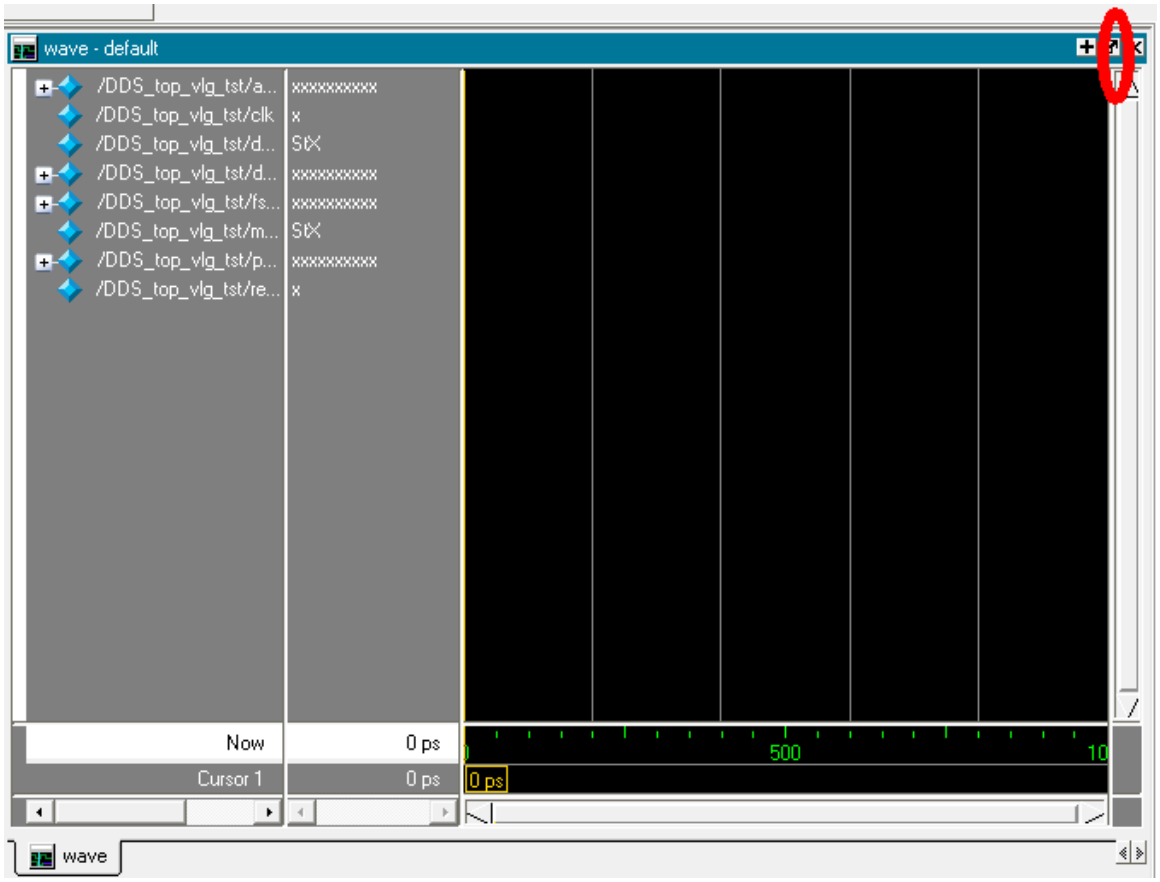


按住 ctrl，选择这几个信号，然后右击：ADD TO WAVE→SELECTED SIGNAL.

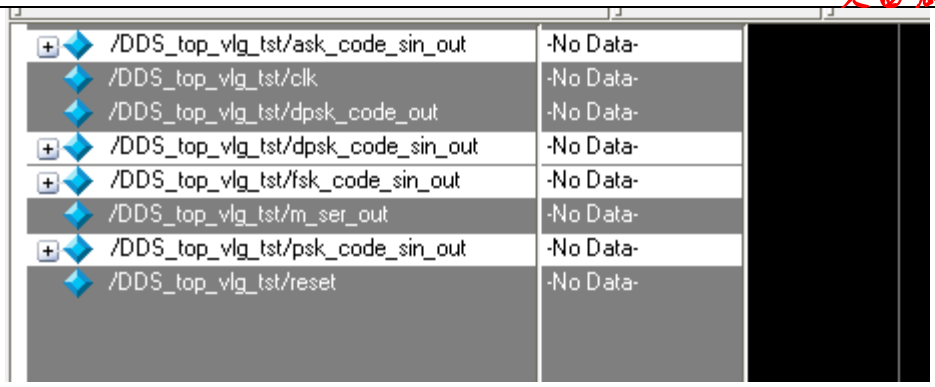





点击红圈，放大窗口

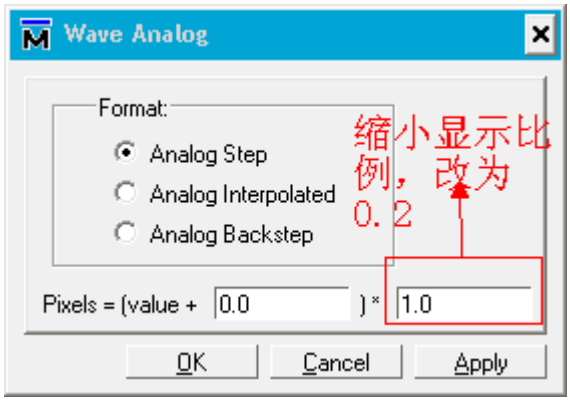



选择这几个信号

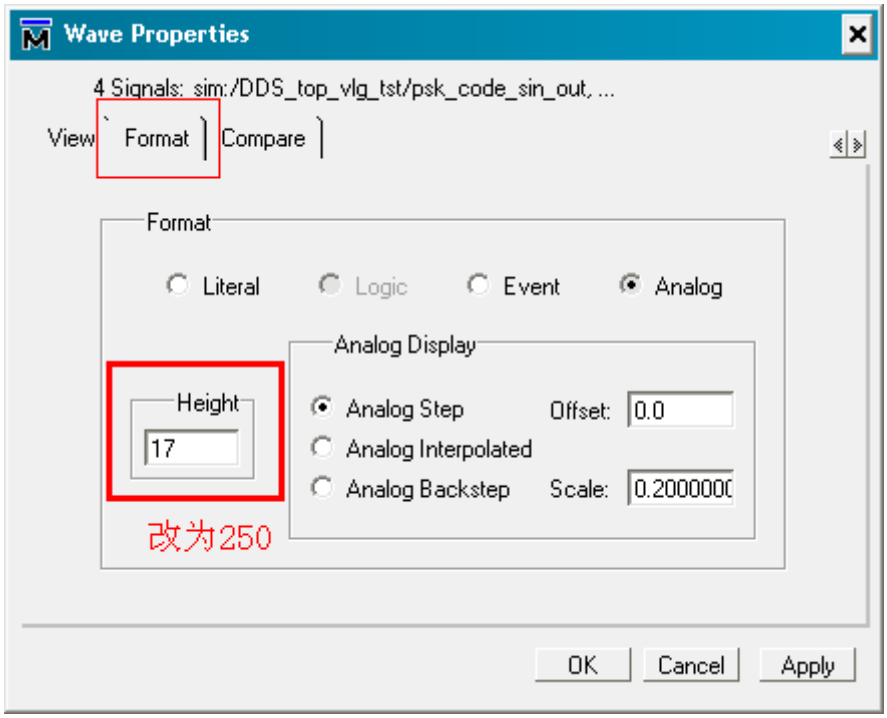


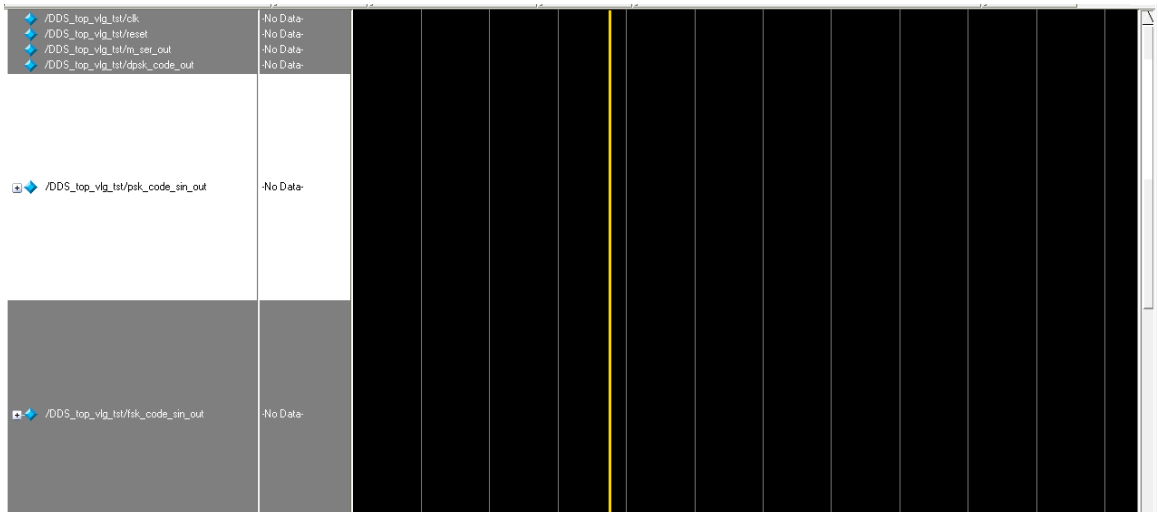

右击，radix→unsigned

然后再右击：format→analog:




选择那四个信号，再右击 properties，

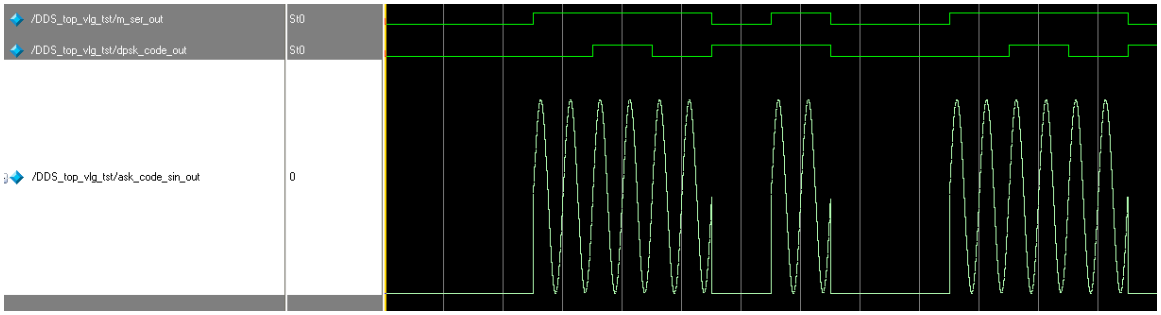




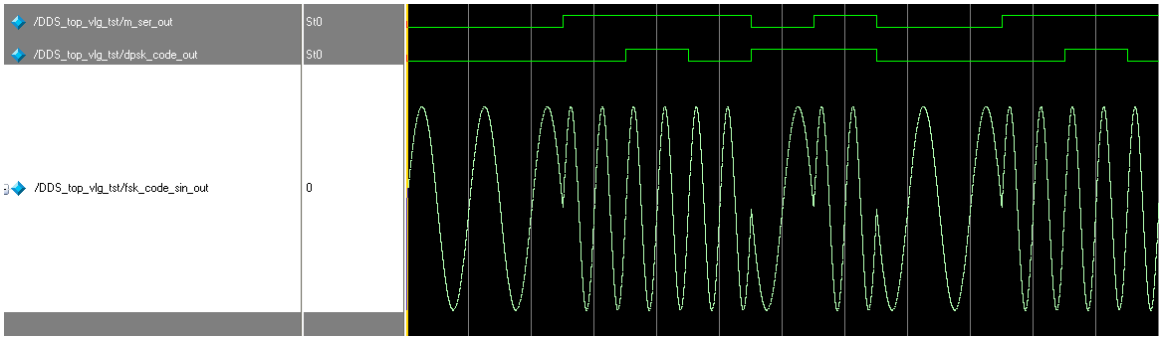


## 5、实验效果

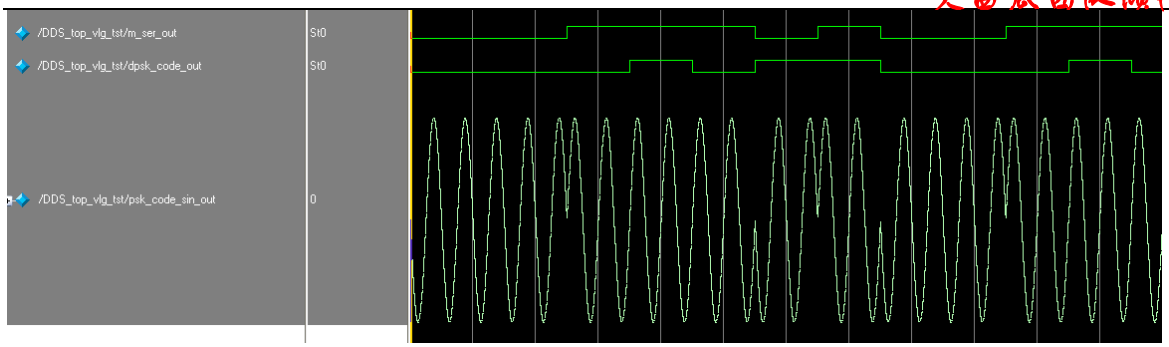
2ASK



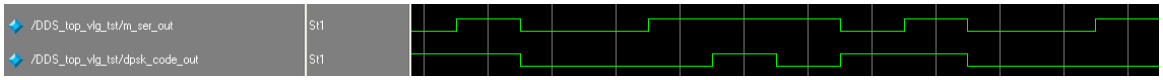

2FSK



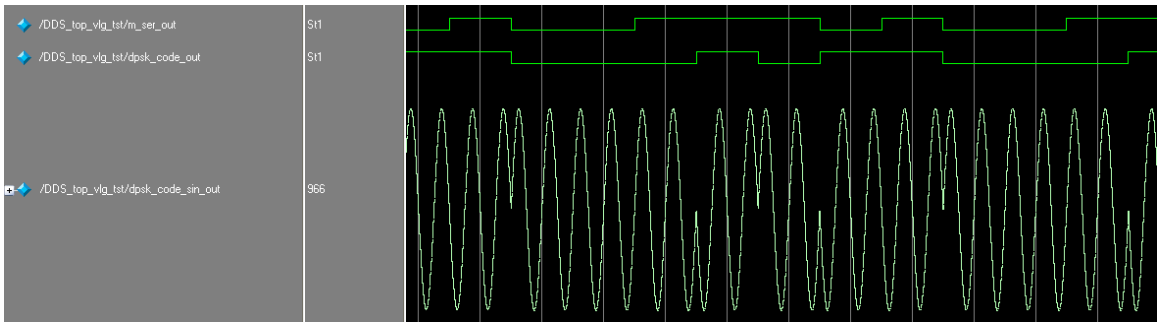

2PSK




**PN 序列和 DPSK 序列**



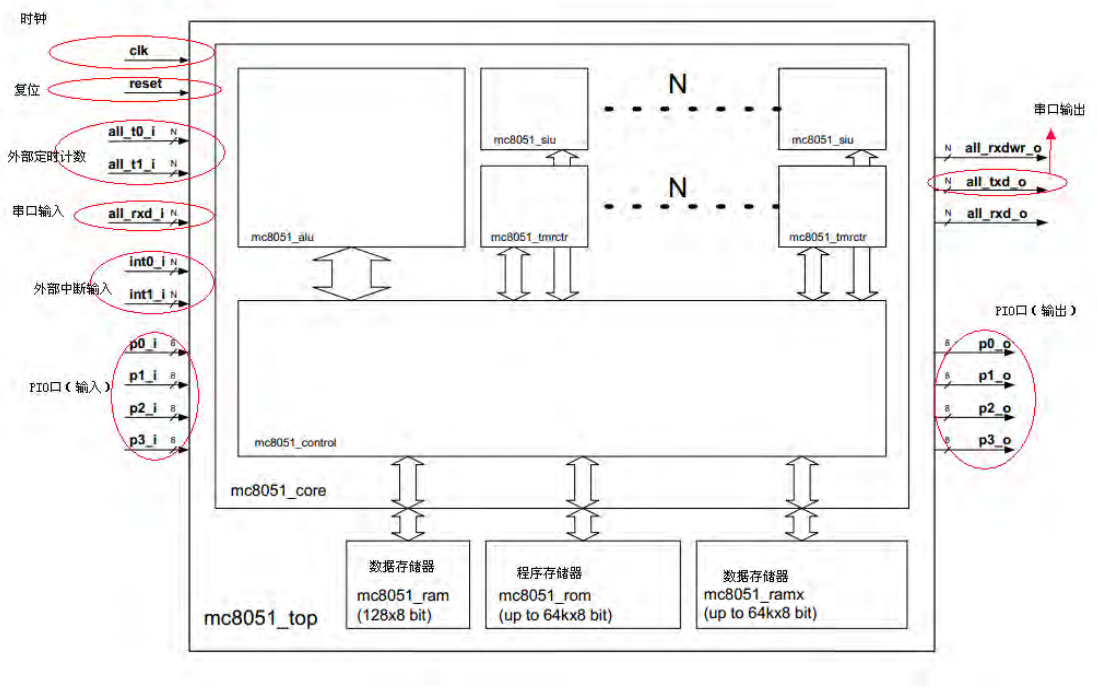

**2DPSK**



# 第十四章 8051 内核的使用

## 14.1 8051 内核介绍

12.1.1、8051 内核结构



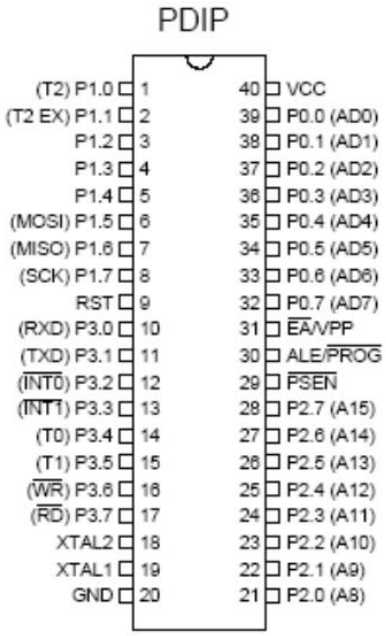

12.1.2、8051 内核说明管脚

| Signal Name | Description |
|---|---|
| clk | System clock. Only rising edge used. |
| reset | Asynchronous reset of all flip-flops. |
| all_t0_i | Timer/counter 0 inputs. |
| all_t1_i | Timer/counter 1 inputs. |
| all_rxd_i | Receive data input for serial interface units. |
| int0_i | Interrupt 0 inputs. |
| int1_i | Interrupt 1 inputs. |
| p0_i | Parallel port 0 input. |
| p1_i | Parallel port 1 input. |
| p2_i | Parallel port 2 input. |
| p3_i | Parallel port 3 input. |
| all_rxdwr_o | Data direction signal for bidirectional rxd input/output (high = output) data. |
| all_txd_o | Transmit data output for serial interface unit. |
| all_rxd_o | Data output for mode 0 operation of serial interface unit. |
| p0_o | Parallel port 0 output. |
| p1_o | Parallel port 1 output. |
| p2_o | Parallel port 2 output. |
| p3_o | Parallel port 3 output. |

该部分的8051内核的顶层信号引脚在8051内核的顶层便可查看到各个引脚模块的定

义。引脚的定义和普通的 8051 单片机相似，区别在于 P3 口的定义，普通的 8051 单片机的 P3 口有双重功能，在对普通的 8051 单片机编程时，特别是用 C 进行编程时所面向的是寄存器层面的，但在硬件层面上，在定义 P3 口的两种功能时是不一致的，原因是用 C 编程时，编译软件已经将 C 代码转变成机器代码，让机器能够辨别要用哪一个硬件模块，也就是 8051 的内核已经“解码”了机器码。普通的 8051 芯片的引脚框图和 P3 口的定义如下如下：

## 引脚结构



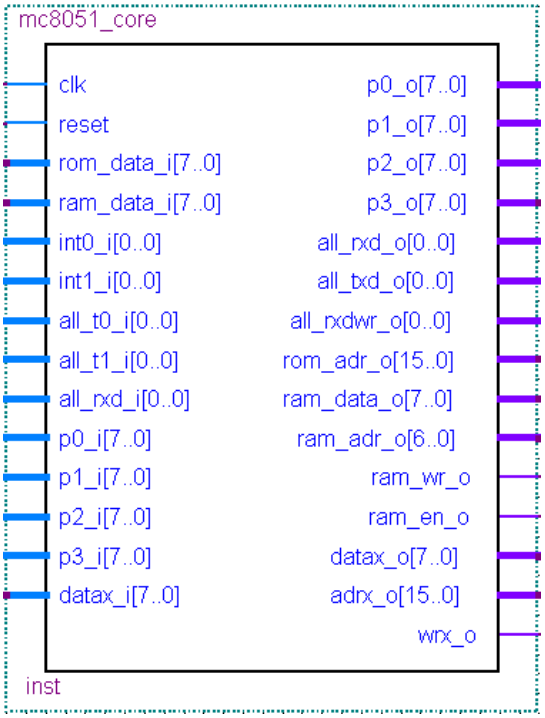

| 引脚号 | 第二功能 |
|---|---|
| P3.0 | RXD（串行输入） |
| P3.1 | TXD（串行输出） |
| P3.2 | $\overline{\text{INT0}}$(外部中断 0) |
| P3.3 | $\overline{\text{INT0}}$(外部中断 0) |
| P3.4 | T0（定时器 0 外部输入） |
| P3.5 | T1（定时器 1 外部输入） |
| P3.6 | $\overline{\text{WR}}$(外部数据存储器写选通) |
| P3.7 | $\overline{\text{RD}}$(外部数据存储器写选通) |

这里姑且不讨论哪些电源、地、时钟输入 XTAL1/XTAL2、地址锁存控制信号 ALE/PROG、外部程序存储器选通信号 PSEN、访问外部程序存储器控制信号 EA/VPP。可以看出剩下的都是单片机的外部通信操作接口（P0~P3 口）。如果想进一步了解 8051 单片机的更多内容可以参考 8051 单片机的芯片手册。（这里不在多讲）

### 12.1.3、8051IP 核

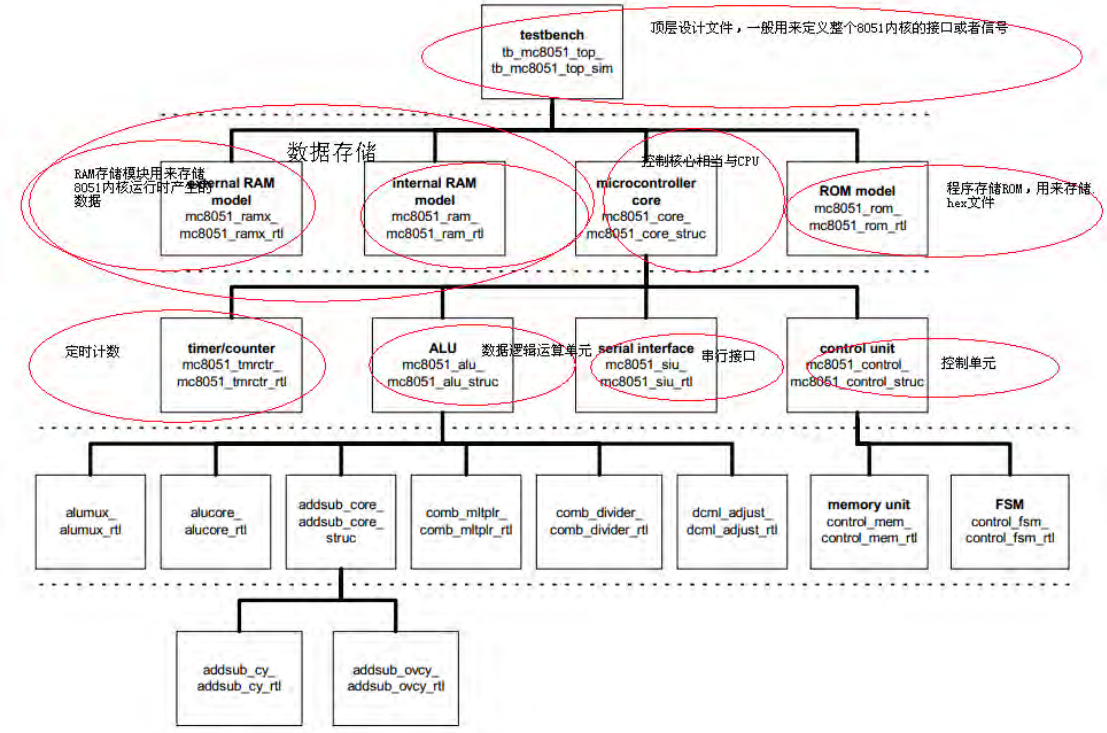
核心本身是由定时器/计数器, ALU, 串行接口, 和控制单元组成。

8015 内核的原理图：

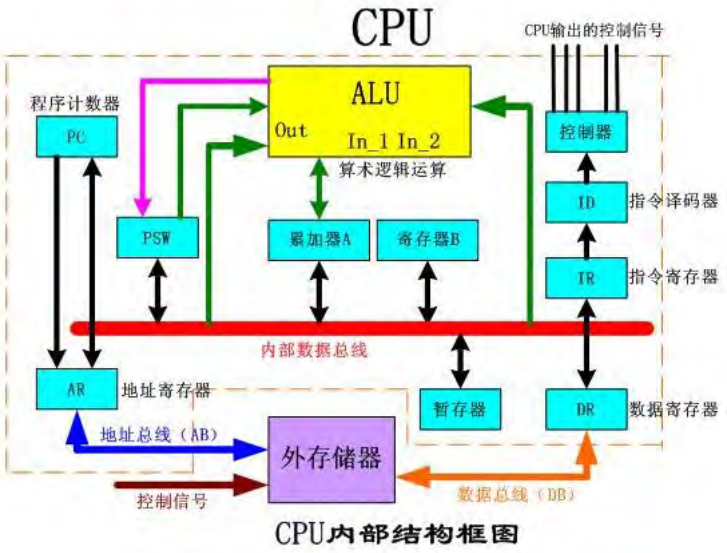



**Clk**：时钟输入
**Reset**：复位输入引脚
**rom_data_i**：ROM 数据输入
**ram_data_i** ：RAM 数据输入
**int0_i** 外部中断 0
**int1_i**：外部中断 1
**all_t0_i**：定时计数 0
**all_t1_i**：定时计数 1
**all_rxd_i**：串口接收
**p0_i**：IO-port0 input
**p1_i**：IO-port1 input
**p2_i**：IO-port2 input
**p3_i**：IO-port3 input
**p0_o**：IO-port0 output
**p1_o**：IO-port1 output
**p2_o**：IO-port2 output
**p3_o**：IO-port3 output
**all_rxd_o**：内部接收后直接输出，可以进入 all_rxd_i
**all_txd_o**：串口发送
**all_rxdwr_o**：接收方向信号
**rom_adr_o**：输出到 ROM 地址信号
**ram_data_o**：输出到 RAM 数据信号
**ram_adr_o**：输出到 RAM 地址信号
**ram_wr_o**：数据输出到 RAM 的使能信号
**ram_en_o**：RAM 的时钟使能信号

**datax_i**：从 RAM 的数据输入到单片机的信号
**datax_o**：从单片机输出数据到 RAM 的信号
**adrx_o**：RAM 的地址信号
**wrx_o**：RAM 的写使能信号
8051 内核的设计层次和设计文件：




层次设计框图中的 microcontroller core 相当与 CPU，一般的 CPU 的整体框图如下：



CPU内部结构框图

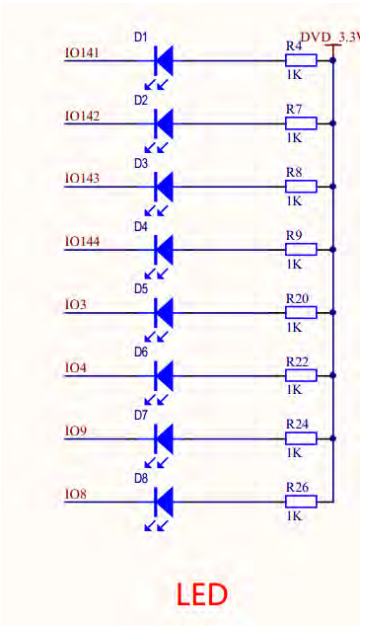
（该 8051 模块的设计文件都是用 VHDL 编写的所以要看懂器内部的构造还必须把 VHDL 学好）原理性的东西和层次结构已经说明了，现在就开始应用 8051 内核，把 FPGA 开发板打

造成一块 8051 单片机开发板，先通过一个简单的实验看认识 8051 内核的使用。

## 14.2、基于 8051 内核的实验—流水灯

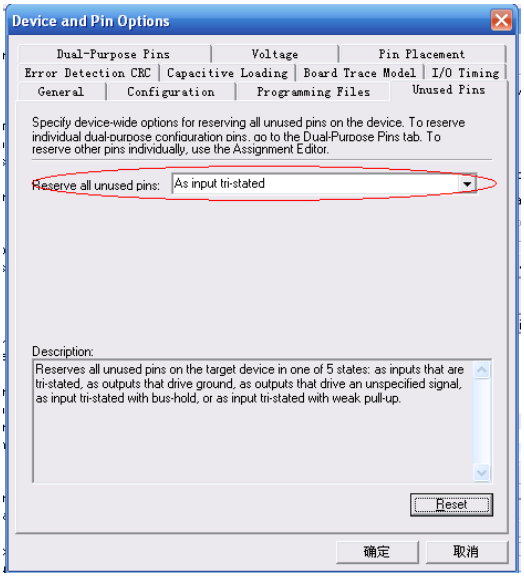
实验步骤：

（1） 打开 51 内核的工程，为内核的进行配置引脚，这个过程根据开发板的原理图就进行，引脚包括：clk、复位、LED 的驱动管脚。

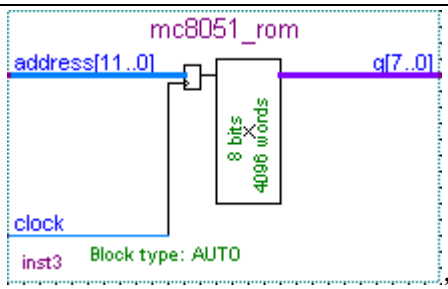



| | Node Name | Direction | Location |
|---|---|---|---|
| 1 | CLOCK_50 | Input | PIN_17 |
| 2 | LEDR[7] | Output | PIN_8 |
| 3 | LEDR[6] | Output | PIN_9 |
| 4 | LEDR[5] | Output | PIN_4 |
| 5 | LEDR[4] | Output | PIN_3 |
| 6 | LEDR[3] | Output | PIN_144 |
| 7 | LEDR[2] | Output | PIN_143 |
| 8 | LEDR[1] | Output | PIN_142 |
| 9 | LEDR[0] | Output | PIN_141 |
| 10 | SW[0] | Input | PIN_74 |
| 11 | <<new node>> | | |

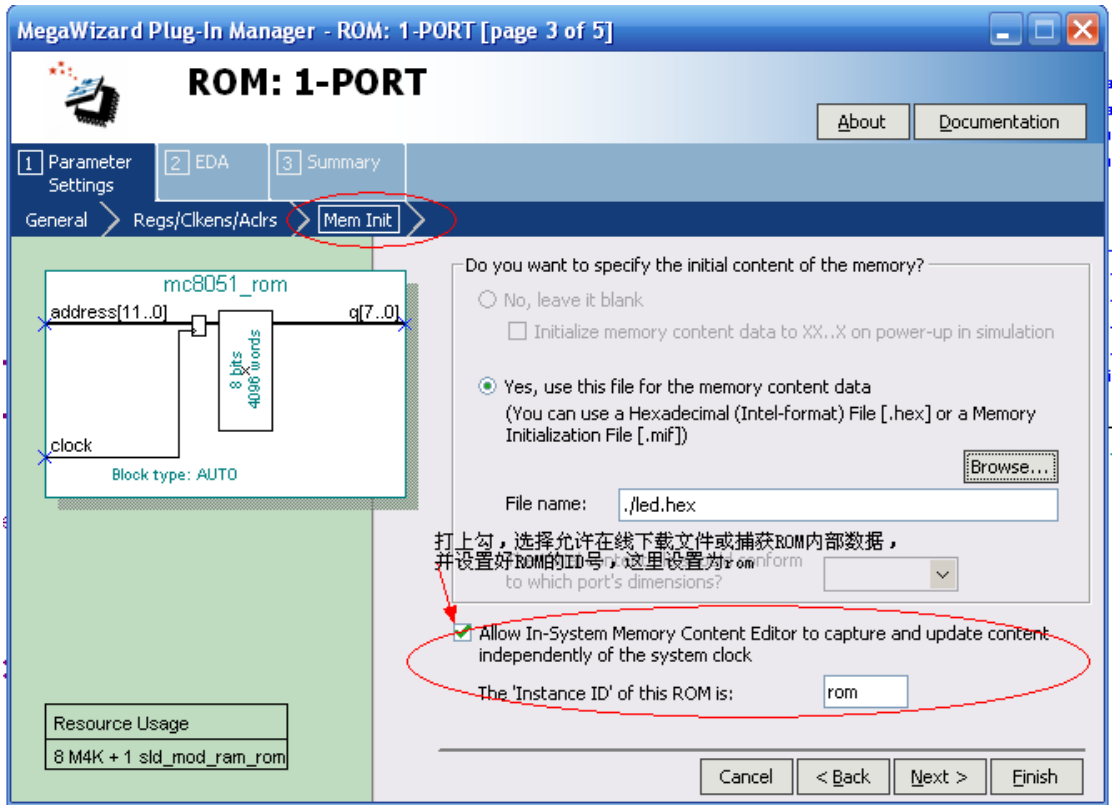
分配好了之后，记得把没有用的管脚设置为输入三阻态。




（2）设置 ROM 程序存储模块的编号，目的是在将网表文件下载到 FPGA 后能够通过 QuartusII 软件访问 ROM 模块，能将.hex 文件下载到 FPGA 中的 ROM（其实 FPGA 中是没有 ROM，ROM 都是用 RAM 做的，只是在操作的时候把它当成 ROM 而已，所以就认为是 ROM），这一步骤是方便以后在重新编译生成.hex 文件时快速下载，不用重新综合整个工程（综合整个工程要大概二十多分钟）。




双击该 ROM 模块：，在出现的窗口中点击 Mem Init，然后进行如下的设置：




设置完成后点击 Finish，完成后进行综合（综合时间比较长，这个时间就可以用来进行 C 编程）。

（3）进行 C 编程，这里采用 Keil 进行编程。（具体流程不详解）

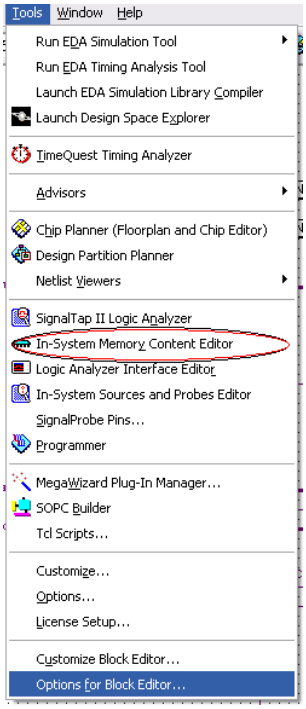
这里实现的功能是 LED 灯的左右移动，代码如下：

```
void display()
{
  unsigned char i,temp;
  temp=0x01;
        for(i=0;i<8;i++)
      {
        temp=0x01;
        for(i=0;i<8;i++)//从左到右逐个点亮
        {
            LED=~temp;
            temp<<=1;
```

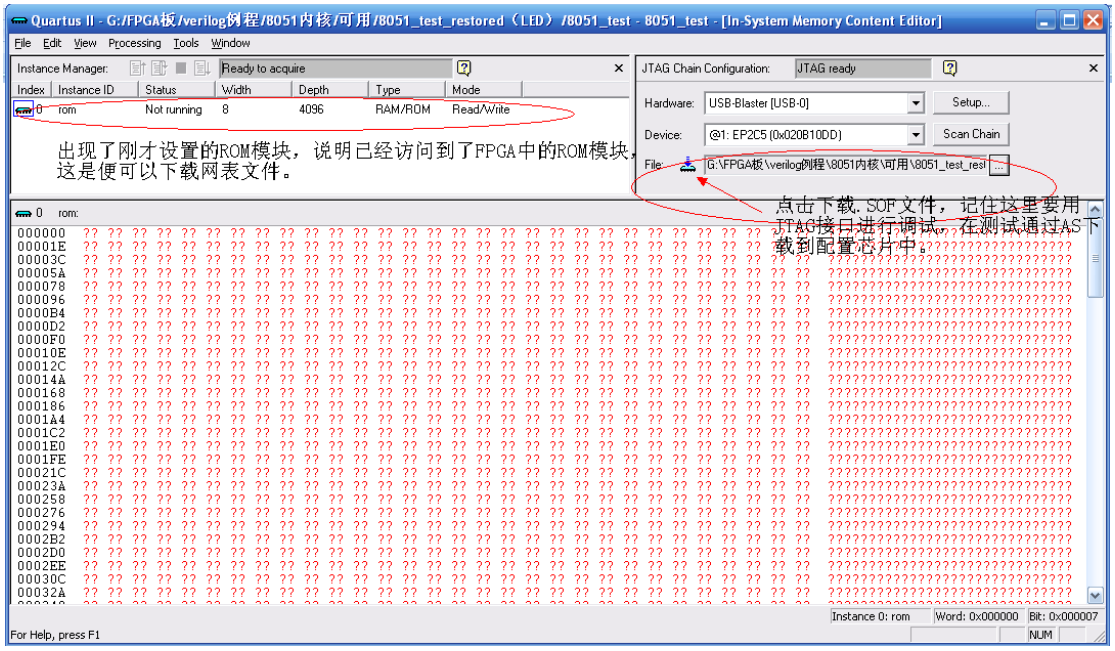
```
            delay_ms(500);
        }
        temp=0x80;
        for(i=0;i<8;i++)//从左到右逐个点亮
        {
            LED=~temp;
            temp>>=1;
            delay_ms(500);
        }
    }
}
```

（4）当你编写好代码生成.hex 后可能会想怎么下载到 FPGA 里面，不急！这里进行详细说明。还记得第二个步骤吗？做好了那个步骤，其实我们的工作已经完成了一半。

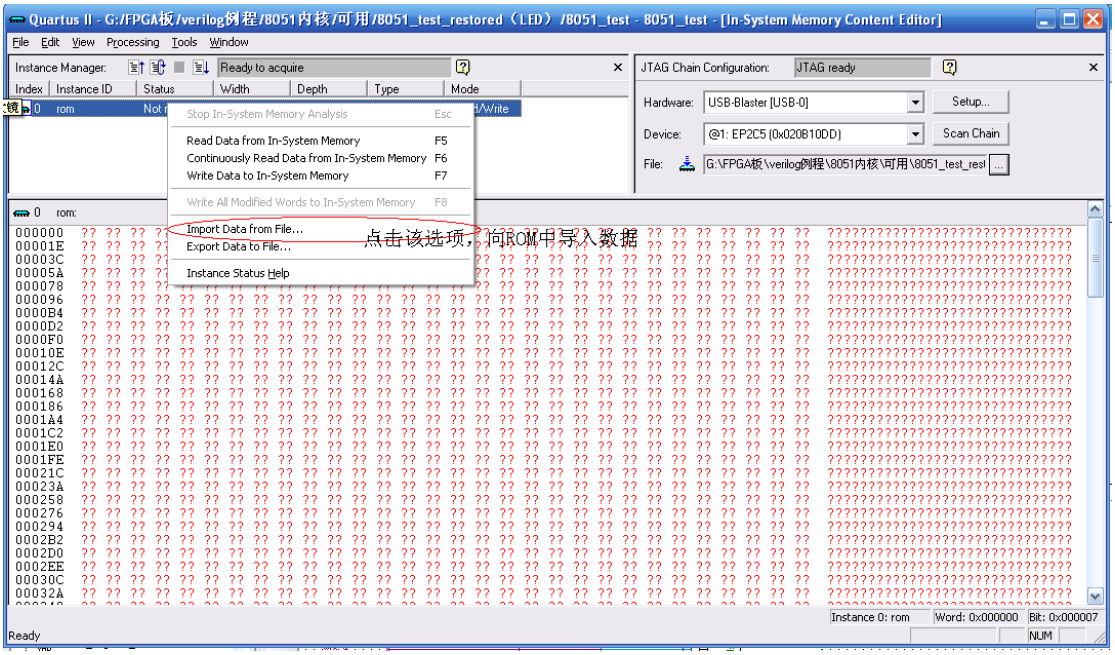
点击 Tools,在出现的下拉菜单中点击 In-System Memory Content Editor。如下图：



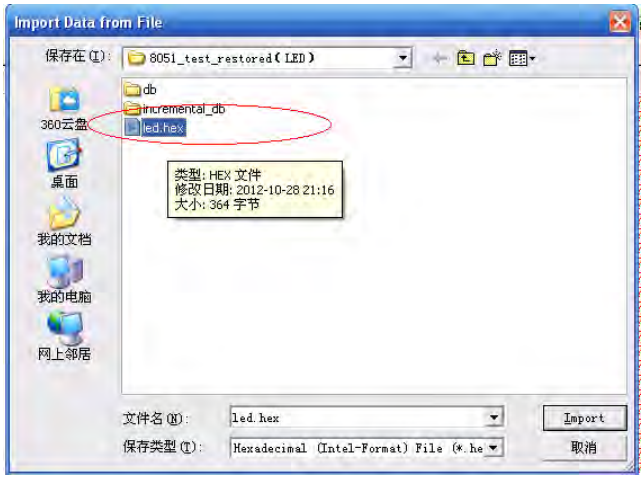

这是会出现以下窗口，



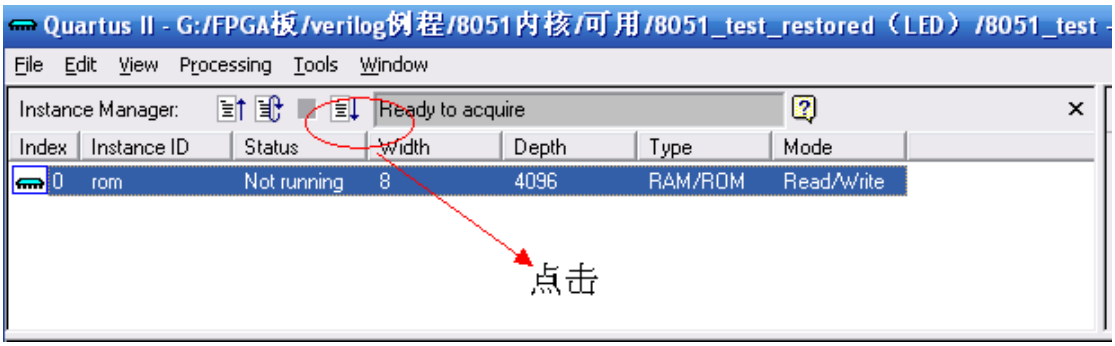

在下载好了之后，便可以将生成的.hex 文件下载到 ROM 中。具体操作如下：
（该工具功能很强大，不仅可以下载文件到 ROM 中，还可以从 ROM 中读取数据生成文件。同时也可以对 ROM 写入和读取数据）。




然后在文件选择窗口中选择生成的.hex 文件，然后点击导入。这里选择如下：




导入后，点击 




便将.hex 下载到 ROM 中。（是不是下载很快啊，比一般的单片机下载还快吧！）
资料持续更新中……

# 第十五章基于 FPGA 滤波器设计入门

## 15.1 直接型、线性相位型 FIR 滤波器的设计

### 一、 FIR 原理以及设计方法

有关 FIR 滤波器的原理，必须要弄懂线性时不变系统和 Z 变换。

**1.FIR 滤波器主要有直接型、级联型、线性相位型：**

1. 直接型结构

图 2-1 给出了 N 阶 LTI 型 FIR 滤波器的图解。可以看出 FIR 滤波器是有一个“抽头延迟线”加法器和乘法器的集合构成的。传给每个乘法器的操作数就是一个 FIR 系数，显然也可以称作“抽头权重”。因此该结构也称为“横向滤波器”。

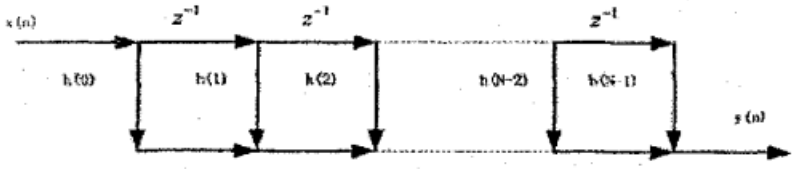



图 2-1 直接型结构的 FIR 滤波器

2. 级联型

如将(2-4)式分解为二阶实系数因子形式：

$$H(z)=\sum_{n=0}^{N-1}h(n)z^{-n}=\prod_{i=1}^{M}(\beta_{0i}+\beta_{1i}z^{-1}+\beta_{2i}z^{-2})$$

便可得二阶级联结构。这种结构每一节控制一对零点，因而在需要控制传输零点时可以采用。但相应的滤波系数增加，乘法运算次数增加，因此需要较多的存储器，运算时间也比直接型增加。

3. 线性相位 FIR 系统的结构

在许多应用领域，例如通信和图像处理中，在一定频率范围内维持相位的完整性是一个期望的系统属性。因此，设计能够建立线性相位－频率功能的滤波器是必须遵循的规范。系统相位线性度的标准尺度就是“组延迟”，其定义为：

$$\tau(w)=\frac{d\phi(w)}{dw} \qquad (2\text{-}5)$$

完全理想的线性相位滤波器对于一定频率范围的组延迟是一个常数。可以看到如果滤波器是对称或者反对称的，就可以实现线性相位。

线性相位(相移)表示一个系统的相频特性与频率成正比，由于不同频率传输速度都一样，所以信号通过它产生的时间延迟等于常数 k 所以不出现相位失真，对一个数字系统来说，即

$$\varphi(w) = -kw$$

假设一个离散时间系统的幅频特性等于 1，则当信号 $x(n)$ 通过该系统后，其输出 $y(n)$ 的频率特性

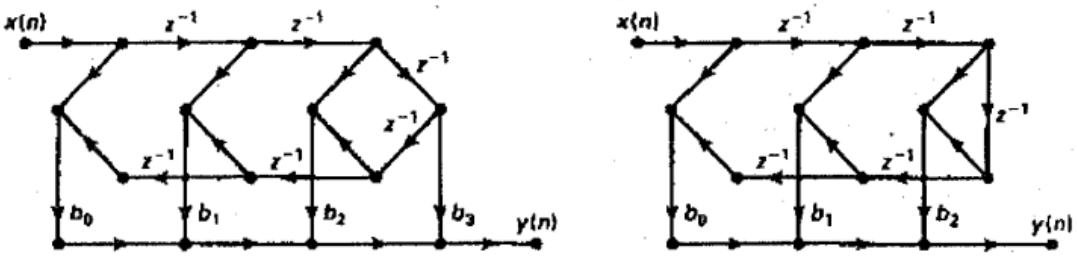
$$Y(e^{jw}) = H(e^{jw})X(e^{jw}) = e^{-jkw}\left|X(e^{jw})\right|e^{j\arg\left[X(e^{jw})\right]} = \left|X(e^{jw})\right|e^{j\arg[X(e^{jw})-kw]}$$

所以 $y(n) = x(n-k)$，这样输出 y(n) 等于输入在时间上的唯一，达到了无失真输出的目的。

可以证明，线性相位条件为：

$$\begin{cases} h(n) = h(N-1-n) \text{偶对称} \\ h(n) = -h(N-1-n) \text{奇对称} \end{cases}$$

即如果单位脉冲响应 h(n) 为实数，且具有偶对称或奇对称性，则 FIR 数字滤波器具有严格的线性相位特性。其对称中心在 $n = \frac{N-1}{2}$ 处。当 N 分别为奇数和偶数时，其网络结构可以分别用图 2-3(a)，(b) 的信号流图来实现。由该信号流图可以看出，线性相位结构比图 2-1 的直接实现形式少用 $\frac{N-1}{2}$ 个乘法器（或乘法运算）。




(a)　　(b)

图 2-3 线性相位 FIR 滤波器结构

(a)N=7；(b)N=6

**2.FIR 滤波器的设计方法**

FIR 滤波器设计方法是以直接逼近所需离散时间系统的频率响应为基础。设计方法包括窗函数法和最优化方法（等同纹波法），其中窗函数方法是设计 FIR 数字滤波器是最常用的方法之一。

1. 窗函数法

任何数字滤波器的频率响应 $H(e^{jw})$ 都是 w 的周期函数，它的傅立叶级数展开式为：

$$H_d(e^{jw}) = \sum_{-\infty}^{+\infty} h_d(n)e^{-jwn} \tag{2-6}$$

其中 $h_d(n) = \frac{1}{2\pi}\int_{-\pi}^{\pi} H(e^{jw})e^{jwn}dw$ (2-7)

傅立叶系数 h(n) 实际上就是数字滤波器的冲击响应，由于 h(n) 可能是无限长序列且为非因果响应，是物理不可实现的。为此要寻找一个因果的 h(n)，在相应的误差准则下最近逼近 $h_d(n)$。窗函数法设计的初衷是使设计的滤波器频率特性 $H(e^{jw})$ 在频域均方误差最小意义下进行逼近，即

$$\varepsilon^2 = \frac{1}{2}\int_{-\pi}^{\pi}\left|H_d(e^{jw}) - H(e^{jw})\right|^2 d\omega = \min$$

窗函数法就是用被称为窗函数的有限加权序列 $G_N(n)$ 来修正式(2-7)，则所需 h(n) 表示为： $h(n) = h_d(n)G_N(n)$ (2-8)

$G_N(n)$ 是有限长序列，当 $n > N-1$ 及 $n < 0$ 时，$G_N(n)$=0，这里我们仅以冲激响应对称即 $h(n) = h(N-1-n)(n = 0,1,2,\ldots N-1)$ 时低通滤波器为例进行说明。低通滤波器的频率响应函数 $H(e^{j\omega})$ 如下式所示：

$$H(e^{jw}) = e^{-jw(N-1)/2}, 0 \le |w| \le w_c \tag{2-9}$$

在 $\omega < |\omega| < \pi$ 时为 0，其中 $\omega$ 为对抽样频率归一化的角频率，$\omega_c$ 为归一化截止角频率。利用反傅立叶变换公式求出式(2-9)对应的冲激响应 h(n)为：

$$h_d(n) = \frac{\sin\left[\omega_c\left(n-\frac{N-1}{2}\right)\right]}{\pi\left[n-\left(\frac{N-1}{2}\right)\right]} \qquad (2\text{-}10)$$

几种窗函数及其窗函数选择原则：

设计 FIR 滤波器常用的窗函数有：矩形窗函数、三角窗（Bartlett）函数、汉宁（Hanning）窗函数、海明(Hamming)窗函数、布莱克曼（Blackman）窗函数和凯塞（Kaiser）窗函数，具体性能指标可参看表 2－1。

| 窗的类型 | 最大旁瓣幅度（相对值） | 过度带宽度 | 最大逼近误差 $20\log_{10}\delta(dB)$ | 等效 Kaiser 窗 $\beta$ |
|---|---|---|---|---|
| 矩形 | -13 | $4\pi/N$ | -21 | 0 |
| Bartlrtt | -25 | $8\pi/N$ | -25 | 1.33 |
| Hanning | -31 | $8\pi/N$ | -44 | 3.86 |
| Hamming | -41 | $8\pi/N$ | -53 | 4.86 |
| Blackman | -57 | $12\pi/N$ | -74 | 7.04 |

表 2-1 窗函数性能指标比较

窗函数的选择原则是：

（1） 具有较低的旁瓣幅度，尤其是第一旁瓣幅度。

（2） 旁瓣幅度下降速度要快，以利于增加阻带衰减。

（3） 主瓣宽度要窄，以获得较陡的过渡带。

通常上述几点很难同时满足，当选用主瓣宽度较窄时，虽然得到较陡的过渡带，但通带和阻带的波动明显增加；当选用最小的旁瓣幅度时，虽然能得到匀滑的幅度响应和较小的阻带波动，但是过渡带加宽。因此，实际选用的窗函数往往事它们的折中。在保证主瓣宽度达到一定要求的情况下，适当地牺牲主瓣的宽度来换取旁瓣波动的减少。

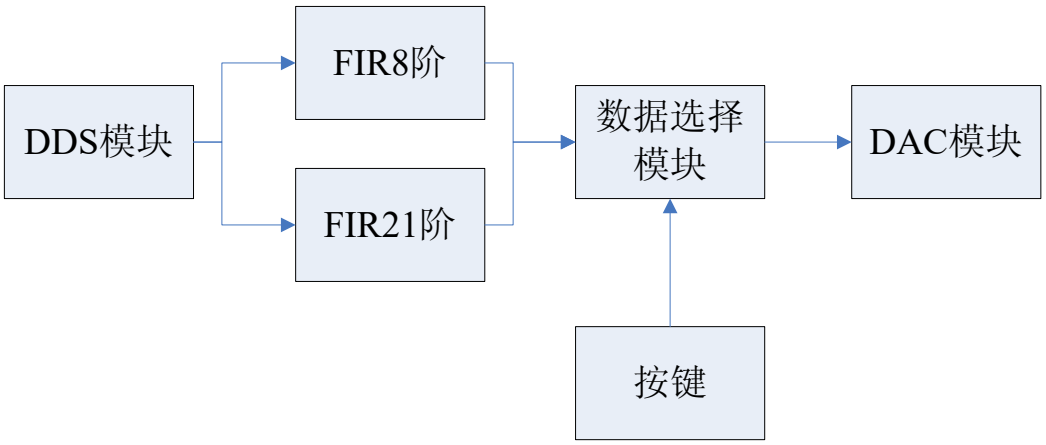
2. 等同纹波设计方法

窗函数存在某些缺陷。首先，在设计中不能将边缘频率 $\omega_p$ 和 $\omega_s$ 精确的给定；也就是意味着在设计完成之后无论得到何值都必须接受。其次，不能够同时标定纹波因子 $\delta_1$ 和 $\delta_2$；在窗函数设计法上只能设定 $\delta_1 = \delta_2$。最后，近似误差在频带区间上不是均匀分布的，在靠近频带边缘误差愈大，远离频带边缘误差愈小。

一种非常有效的解决这种问题的 FIR 滤波器就是等同纹波 FIR 滤波器。对于线性相位 FIR 滤波器来说，有可能导得一组条件，对这组条件能够证明，在最大近似误差最小化的意义下，这个设计时最优的。具有这种性质的滤波器就称为等同纹波滤波器，因为近似误差在通带和阻带上都是均匀分布的。

等同纹波法通常都是采用 Park-McClellan 迭代方法来实现的，与直接频率法相比，等同纹波设计法的优点在于通频带和抑制代偏差可以分别指定，且实现相同指标的滤波器时所用的滤波器阶数较小。

## 二、 设计框架




设计的滤波器的采样频率为 100K，截至频率为 20K。

通过一个 DDS 产生两个正弦波，一个为 1K 的正弦波幅值较大，另一个为 21K 的正弦波幅值较小，然后将幅值较小的正弦波叠加到幅值较大的正弦波上。这样就产生含有高次谐波的正弦波，最后就是将该正弦波（其实都已经失真了）送往两个 FIR 滤波器中进行处理。

FIR8 阶滤波器的采样频率为 100K，截至频率为 20K，通过 Matlab 软件导出需要的滤波器系数：0.009、0.048、0.164、0.279、0.279、0.164、0.048、0.009。因为该滤波器为线性相位滤波器，并且为偶对称滤波器。

FIR21 阶滤波器的采样频率为 100K，截至频率为 20K，通过 Matlab 软件导出需要的滤波器系数： -0.0000、-0.0021、-0.0063、-0.0116、-0.0124、0.0000、0.0318、0.0814、0.1375、0.1821 0.1992、0.1821、0.1375、0.0814、0.0318、0.0000、-0.0124、-0.0116、-0.0063、-0.0021、-0.0000。因为该滤波器为线性相位滤波器，并且为偶对称滤波器。

数据选择模块主要用来对两个滤波器输出的数据进行选择，其实就是一个二选一的数据选择器，选择后送往 DAC 模块驱动输出，最后就可以观察滤波器的效果了。

## 三、 Matlab 软件导出需要的滤波器系数过程

采用 Matlab 软件求取滤波器系数时，要先确定好截止频率、采样频率和滤波器系数，还有就是采用哪种窗函数，这里采用 Matlab 自带的两个函数，分别为：Fir1 和 Fir2，Fir1 为窗函数设计方法，Fir2 为任意频率响应的各种加窗 FIR 滤波器。在该设计中我们采用的是 Fir1 求取滤波器系数。过程如下：

打开 Matlab 软件

在指令窗口中键入：m=fir1(7,0.2)，即可得到如下的系数

```
Command Window
>> m=fir1(7,0.2)

m =

    0.0088    0.0479    0.1640    0.2793    0.2793    0.1640    0.0479    0.0088

>>
```

0.009、0.048、0.164、0.279、0.279、0.164、0.048、0.009

同样的，求取 21 阶的滤波器系数时，只要键入:n=fir1(20,0.2),便可得到系数。

## 四、 滤波器的设计部分

1. 8 阶

   1) 横截型链型结构

```
//*****************************************************//
//*****************横截型（链型结构）***************//
//*****************************************************//
always @(posedge clk_div or negedge reset_n)
 begin
   if(!reset_n)
    fir_data<=10'd0;
   else
    begin
     fir_data<=
           9*t1[0]/1000
                +48*t1[1]/1000
                +164*t1[2]/1000
                +279*t1[3]/1000
                +279*t1[4]/1000
                +164*t1[5]/1000
                +48*t1[6]/1000
                +9*t1[7]/1000;
     t1[1]<=t1[0];
     t1[2]<=t1[1];
```

```
        t1[3]<=t1[2];
        t1[4]<=t1[3];
        t1[5]<=t1[4];
        t1[6]<=t1[5];
        t1[7]<=t1[6];
        t1[0]<=data_in;
      end
end
```

2) 横截型树型结构

```
//*****************************************************//
//*****************横截型（树形结构）****************//
//*****************************************************//
reg [9:0]data_reg [5:0];
 always @(posedge clk_div or negedge reset_n)
   begin
    if(!reset_n)
     fir_data<=10'd0;
    else
     begin
       data_reg[0]<=(9*t1[0]+48*t1[1])/1000;
       data_reg[1]<=(164*t1[2]+279*t1[3])/1000;
       data_reg[2]<=(279*t1[4]+164*t1[5])/1000;
       data_reg[3]<=(48*t1[6]+9*t1[7])/1000;
       data_reg[4]<=data_reg[0]+data_reg[1];
       data_reg[5]<=data_reg[2]+data_reg[3];
       fir_data<=data_reg[4]+data_reg[5];

       t1[1]<=t1[0];
       t1[2]<=t1[1];
       t1[3]<=t1[2];
       t1[4]<=t1[3];
       t1[5]<=t1[4];
       t1[6]<=t1[5];
       t1[7]<=t1[6];
       t1[0]<=data_in;
     end
   end
```

3) 线性相位结构

```
//*****************************************************//
//**********************（线性相位结构）************//
//*****************************************************//
reg [31:0]data_reg [5:0];
reg [31:0]data_temp;
```

```
always @(posedge clk_div or negedge reset_n)
  begin
    if(!reset_n)
      fir_data<=10'd0;
    else
      begin
        data_reg[0]<=9*(t1[0]+t1[7]);
        data_reg[1]<=48*(t1[1]+t1[6]);
        data_reg[2]<=164*(t1[2]+t1[5]);
        data_reg[3]<=279*(t1[3]+t1[4]);
        data_reg[4]<=data_reg[0]+data_reg[1];
        data_reg[5]<=data_reg[2]+data_reg[3];

        data_temp<=(data_reg[4]+data_reg[5])/1000;
        fir_data<=data_temp[9:0];
        t1[1]<=t1[0];
        t1[2]<=t1[1];
        t1[3]<=t1[2];
        t1[4]<=t1[3];
        t1[5]<=t1[4];
        t1[6]<=t1[5];
        t1[7]<=t1[6];
        t1[0]<=data_in;
      end
  end
```

2. 21 阶

1) 横截型链型结构

```
//*****************************************************//
//******************21 阶横截型（链型结构）************//
//*****************************************************//
reg       [9:0] t[20:0];
always @(posedge clk_div or negedge reset_n)
begin
    if(!reset_n)
      fir_data_20<=10'd0;
    else
       fir_data_20<=
                          0*t[0]/1000
                        -2*t[1]/1000
                        -6*t[2]/1000
                        -12*t[3]/1000
                        -12*t[4]/1000
                        +0*t[5]/1000
```

```
                    +32*t[6]/1000
                    +81*t[7]/1000
                    +138*t[8]/1000
                    +182*t[9]/1000
                    +199*t[10]/1000
                    +182*t[11]/1000
                    +138*t[12]/1000
                    +81*t[13]/1000
                    +32*t[14]/1000
                    +0*t[15]/1000
                    -12*t[16]/1000
                    -12*t[17]/1000
                    -6*t[18]/1000
                    -2*t[19]/1000
                    +0*t[20]/1000;

                    t[1]<=t[0];
                    t[2]<=t[1];
                    t[3]<=t[2];
                    t[4]<=t[3];
                    t[5]<=t[4];
                    t[6]<=t[5];
                    t[7]<=t[6];
                    t[8]<=t[7];
                    t[9]<=t[8];
                    t[10]<=t[9];
                    t[11]<=t[10];
                    t[12]<=t[11];
                    t[13]<=t[12];
                    t[14]<=t[13];
                    t[15]<=t[14];
                    t[16]<=t[15];
                    t[17]<=t[16];
                t[18]<=t[17];
                    t[19]<=t[18];
                    t[20]<=t[19];

                    t[0]<=data_in;
end
```

2) 线性相位结构

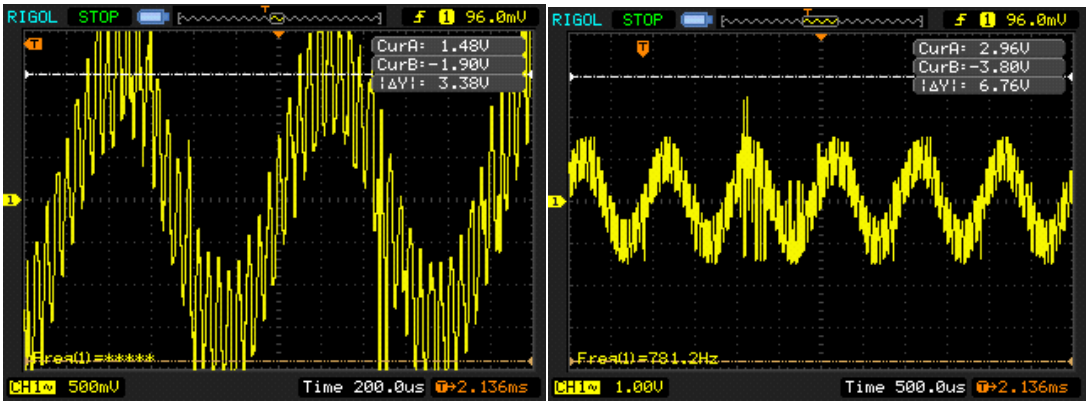
```
//*****************************************************//
//***********************（线性相位结构）************//
//*****************************************************//
```

```
reg             [9:0] t[20:0];
wire            [31:0]data_reg21 [16:0];
wire            [31:0]data_fir21;
assign          data_reg21[0]= 2*(t[1]+t[19]);//-
assign          data_reg21[1]= 6*(t[2]+t[18]);//-
assign          data_reg21[2]=12*(t[3]+t[17]);//-
assign          data_reg21[3]=12*(t[4]+t[16]);//-

assign          data_reg21[4]=32*(t[6]+t[14]);
assign          data_reg21[5]=81*(t[7]+t[13]);
assign          data_reg21[6]=138*(t[8]+t[12]);
assign          data_reg21[7]=182*(t[9]+t[11]);
assign          data_reg21[8]=199*t[10];

assign          data_reg21[9]= data_reg21[0]+data_reg21[1];
assign          data_reg21[10]=data_reg21[2]+data_reg21[3];
assign          data_reg21[11]=data_reg21[4]+data_reg21[5];
assign          data_reg21[12]=data_reg21[6]+data_reg21[7];

assign          data_reg21[13]=data_reg21[9]+data_reg21[10];
assign          data_reg21[14]=data_reg21[11]+data_reg21[12];
assign          data_reg21[15]=data_reg21[14]+data_reg21[8];

assign          data_reg21[16]=data_reg21[15]-data_reg21[13];
assign          data_fir21=data_reg21[16]/1000;
always @(posedge clk_div or negedge reset_n)
  begin
     if(!reset_n)
       fir_data_20<=10'd0;
     else
       begin
              fir_data_20<=data_fir21[9:0];
              t[1]<=t[0];
              t[2]<=t[1];
              t[3]<=t[2];
              t[4]<=t[3];
              t[5]<=t[4];
              t[6]<=t[5];
              t[7]<=t[6];
              t[8]<=t[7];
              t[9]<=t[8];
              t[10]<=t[9];
              t[11]<=t[10];
```

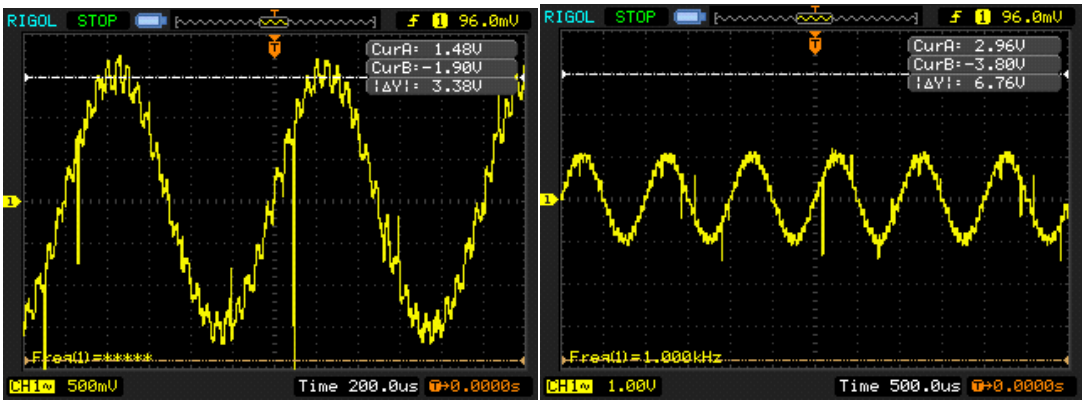
```
            t[12]<=t[11];
            t[13]<=t[12];
            t[14]<=t[13];
            t[15]<=t[14];
            t[16]<=t[15];
            t[17]<=t[16];
            t[18]<=t[17];
            t[19]<=t[18];
            t[20]<=t[19];

        t[0]<=data_in;
    end
end
```

## 五、 实验现象

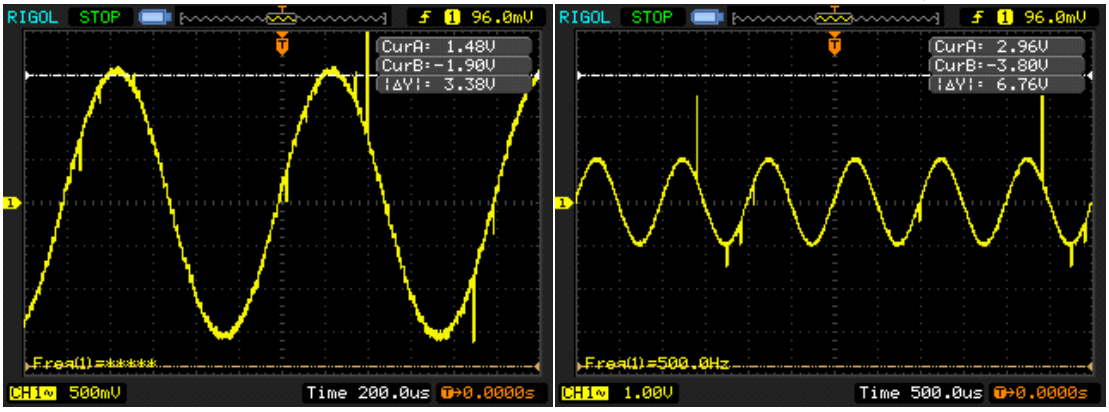
**1.含高次谐波：**




**2.经过 8 阶 FIR 滤波之后：**




**3.经过 21 阶滤波之后：**


RIGOL STOP
96.0mV
CurA: 1.48V
CurB:-1.90V
ΔV: 3.38V
Freq(1)=*****
CH1 500mV
Time 200.0us
0.0000s
RIGOL STOP
96.0mV
CurA: 2.96V
CurB:-3.80V
ΔV: 6.76V
Freq(1)=500.0Hz
CH1 1.00V
Time 500.0us
0.0000s

## 15.2 分布式 FIR 滤波器的设计

待编写

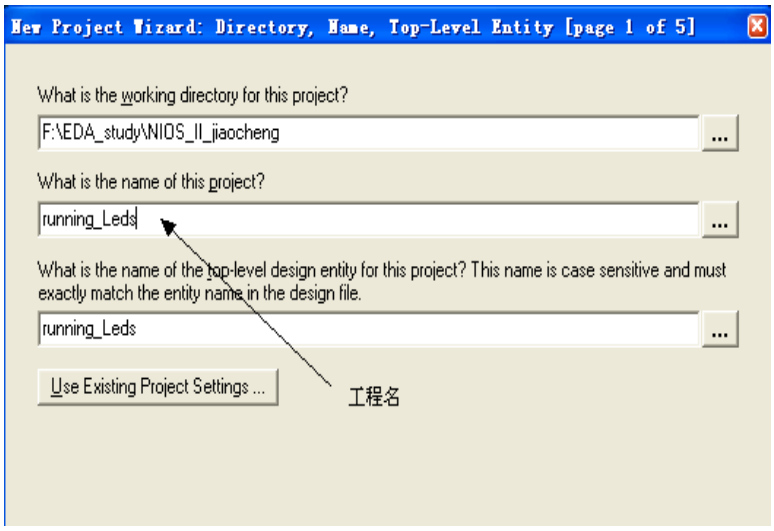
# 第十六章 NIOSII 手把手入门教学

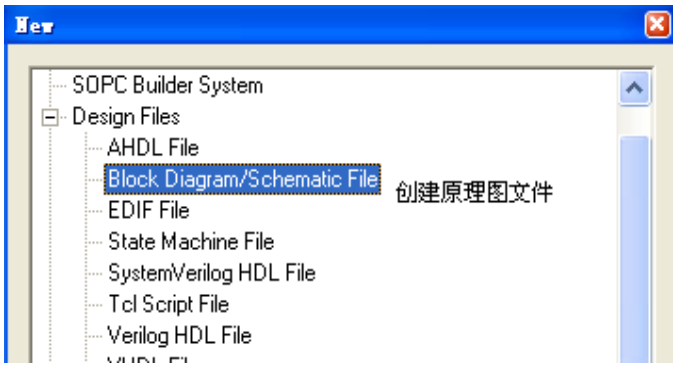
前言：NIOS II 是什么？也许你第一次听到这个词。通俗地讲，NIOS II 其实是一个软核，那什么是软核？软核是一个基于哈佛结构的可配置处理器。哈佛结构，联想到了吧，其实 51 单片机也是基于哈佛结构的处理器。至于为什么是哈佛结构，我觉得可能是哈佛结构的处理速度较快，因为他可以在一个周期读取数据和指令。（关于哈佛结构的详细信息请自己查阅！）因此 NIOS II 是把指令存储器和数据存储器分开的。在此我想详细介绍下可配置处理器的概念。我们都知道 51 单片机或者 ARM 里面的硬件资源都是固定的，是固化的，不能修改的。但是软核可配置就不一样啦，它里面的 CPU，外围器件都是可选择的。因此我们可以根据实际的情况来选择需要的资源，这样可以提高利用率。总之，大家把它当做单片机来学习就行了！


讲了一大堆，相信大家还是有点模糊，下面就以流水灯介绍 NIOS II 吧，让我们进入 NIOS II 的神奇世界！（在此相信大家都已经掌握 quartus 的基本使用和安装了与 quartus 同样版本的 NIOS II 软件！）

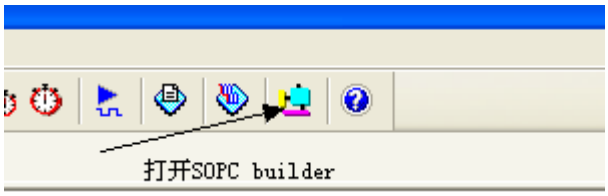
Quartus 环境操作：

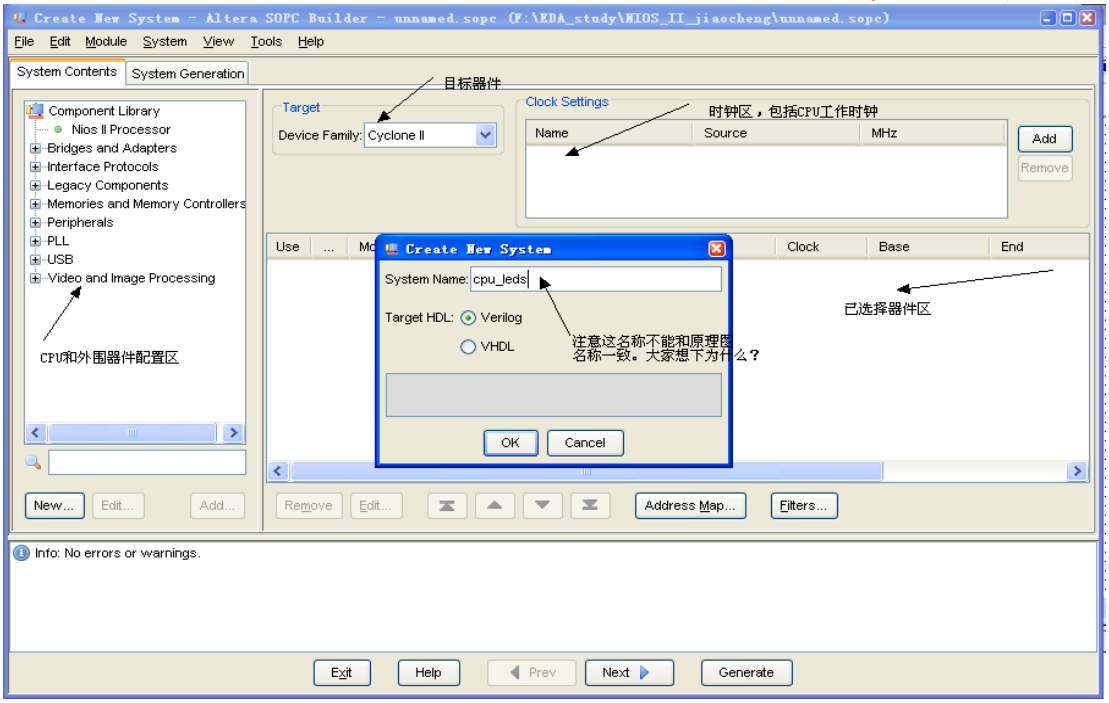
第一步：建立一个 Quartus II 工程和原理图文件



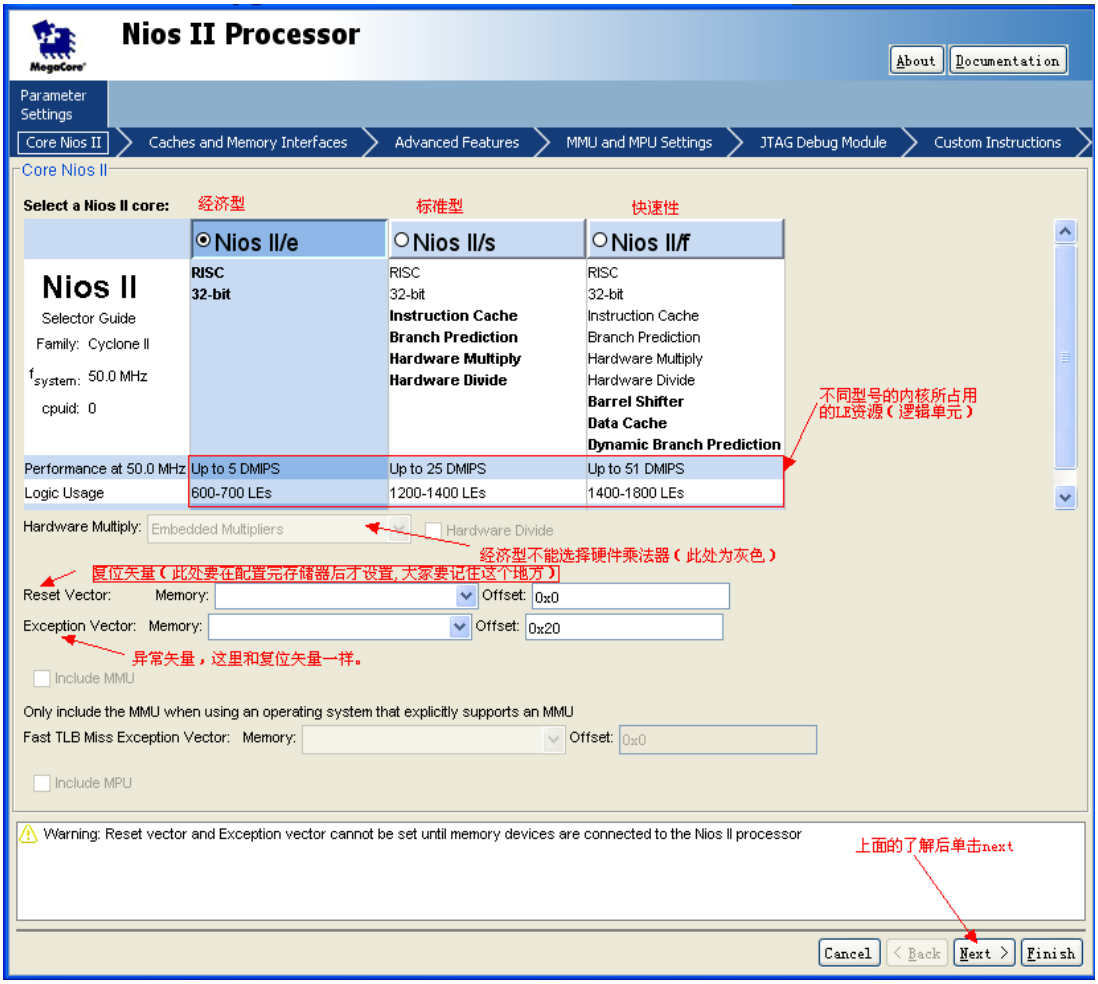

第二步：打开配置 NIOS II 窗口

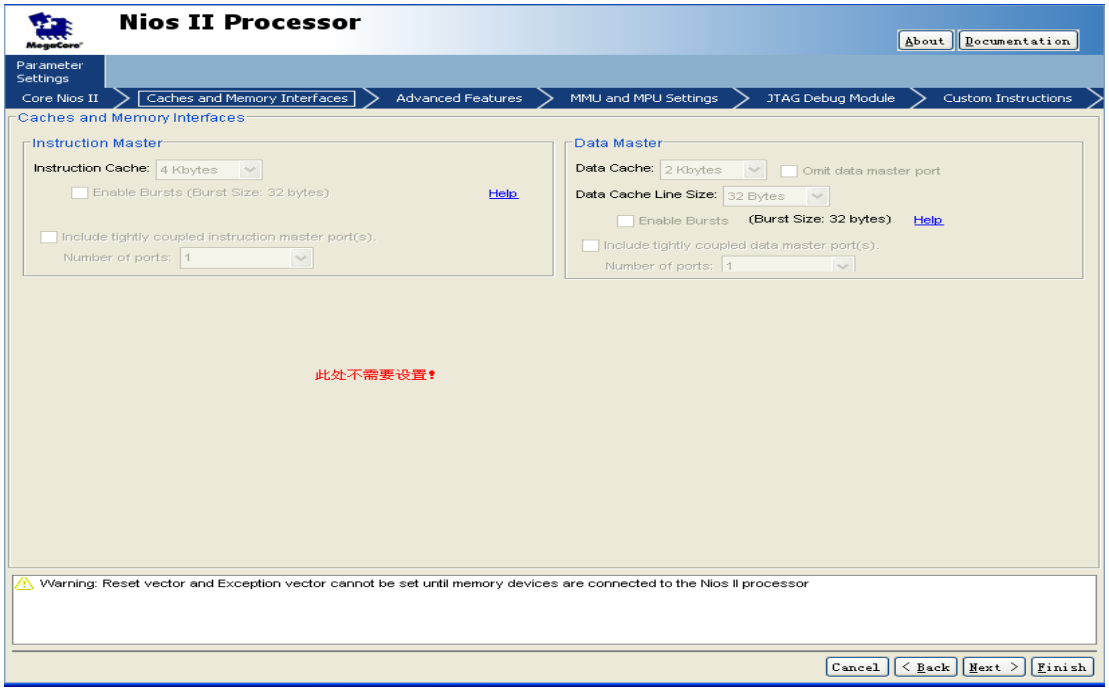


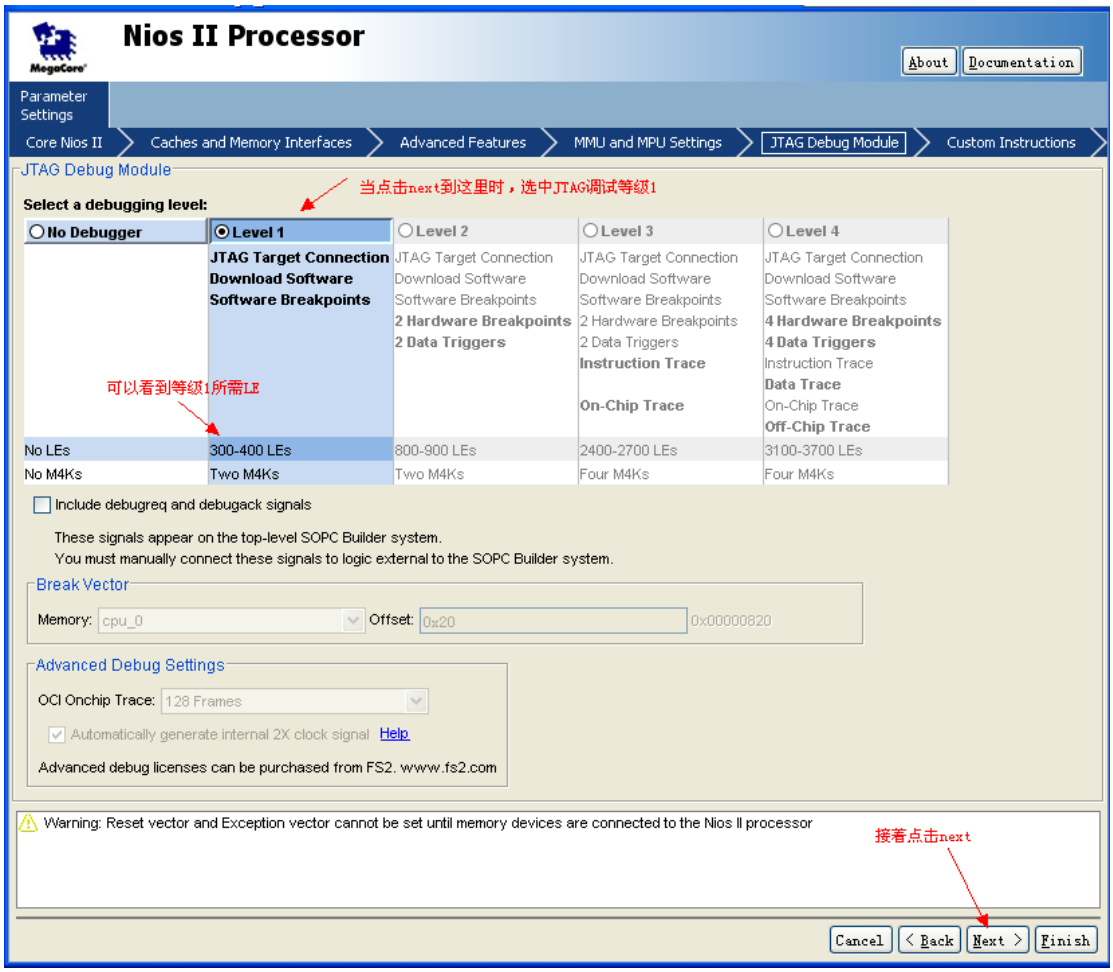


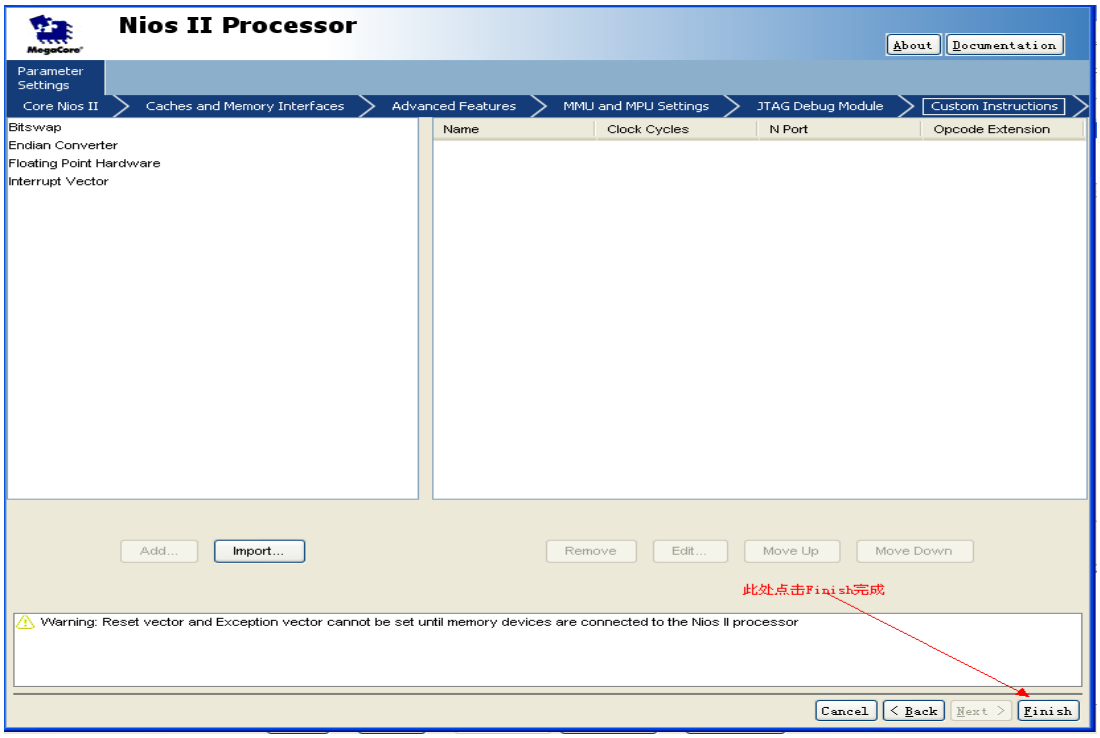

第三步：配置 NIOS II 内核和外围器件

（双击 system contents 窗口中 component library 下拉菜单的 Nios II processor）

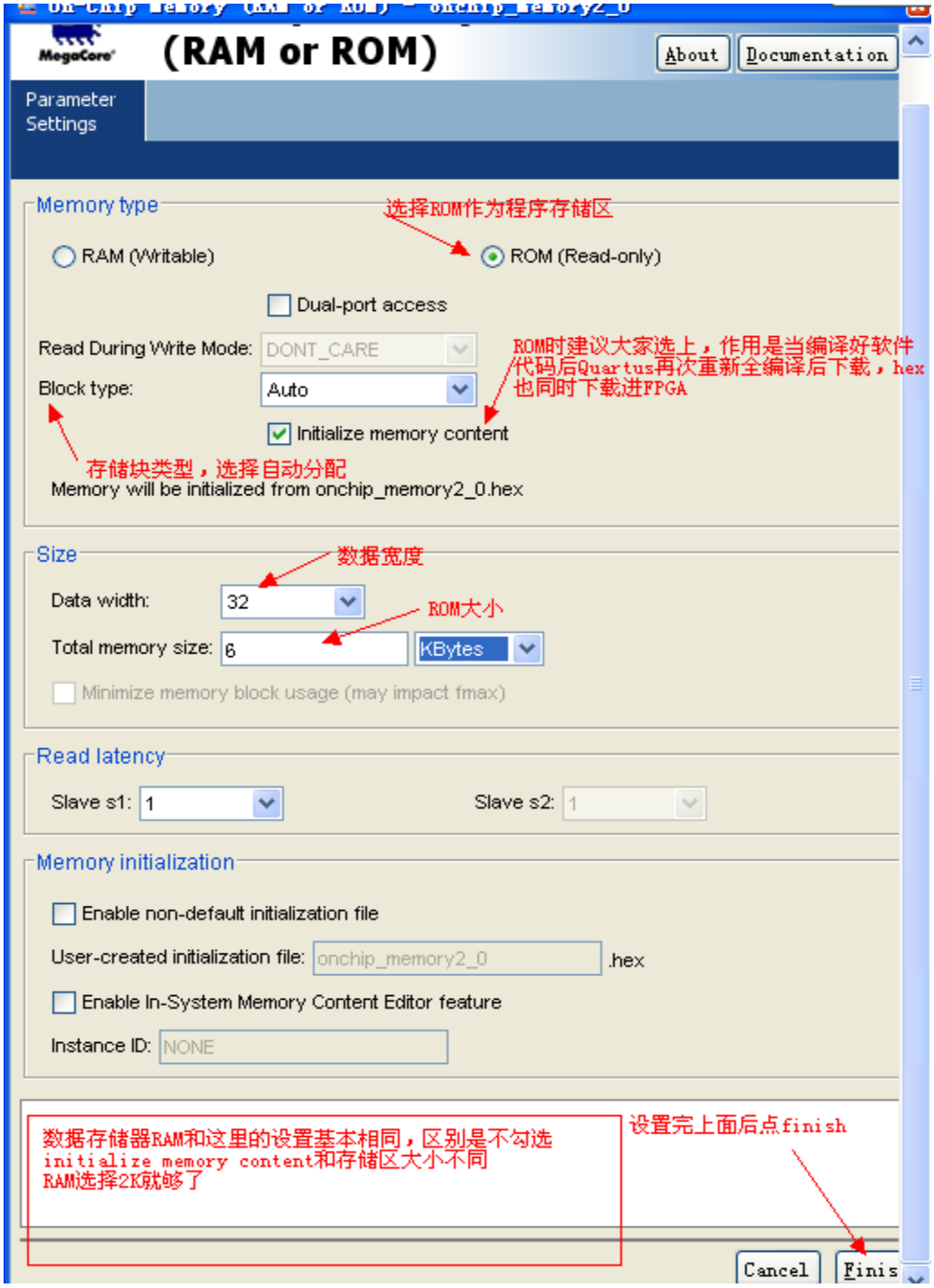



Nios II Processor
About
Documentation
Parameter Settings
Core Nios II
Caches and Memory Interfaces
Advanced Features
MMU and MPU Settings
JTAG Debug Module
Custom Instructions
Caches and Memory Interfaces
Instruction Master
Instruction Cache: 4 Kbytes
Enable Bursts (Burst Size: 32 bytes)
Help
Include tightly coupled instruction master port(s).
Number of ports: 1
Data Master
Data Cache: 2 Kbytes
Omit data master port
Data Cache Line Size: 32 Bytes
Enable Bursts (Burst Size: 32 bytes)
Help
Include tightly coupled data master port(s).
Number of ports: 1
此处不需要设置!
Warning: Reset vector and Exception vector cannot be set until memory devices are connected to the Nios II processor
Cancel
< Back
Next >
Finish

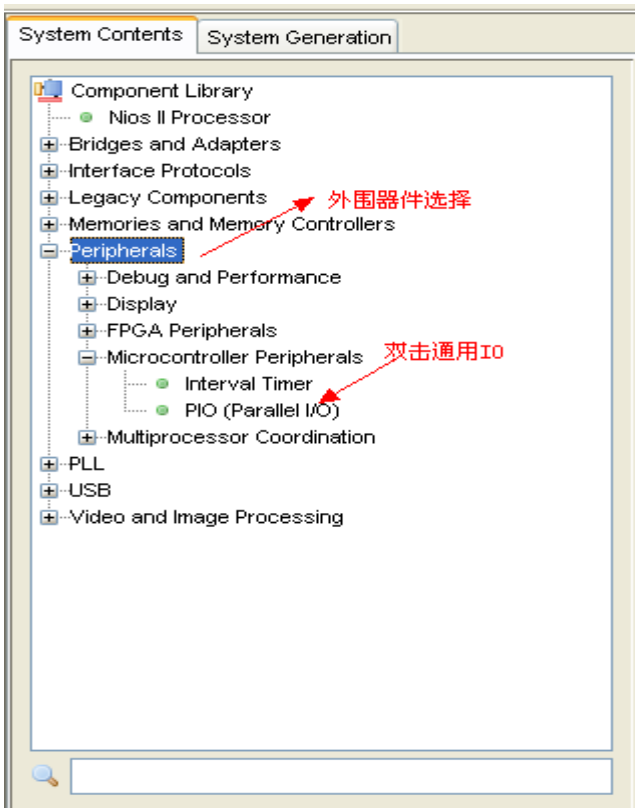

Nios II Processor
About
Documentation
Parameter Settings
Core Nios II
Caches and Memory Interfaces
Advanced Features
MMU and MPU Settings
JTAG Debug Module
Custom Instructions
JTAG Debug Module
当点击next到这里时，选中JTAG调试等级1
Select a debugging level:
No Debugger
Level 1
Level 2
Level 3
Level 4
JTAG Target Connection
Download Software
Software Breakpoints
2 Hardware Breakpoints
2 Data Triggers
Instruction Trace
On-Chip Trace
Data Trace
Off-Chip Trace
4 Hardware Breakpoints
4 Data Triggers
可以看到等级1所需LE
No LEs
300-400 LEs
800-900 LEs
2400-2700 LEs
3100-3700 LEs
No M4Ks
Two M4Ks
Two M4Ks
Four M4Ks
Four M4Ks
Include debugreq and debugack signals
These signals appear on the top-level SOPC Builder system.
You must manually connect these signals to logic external to the SOPC Builder system.
Break Vector
Memory: cpu_0
Offset: 0x20
0x00000820
Advanced Debug Settings
OCI Onchip Trace: 128 Frames
Automatically generate internal 2X clock signal
Help
Advanced debug licenses can be purchased from FS2. www.fs2.com
Warning: Reset vector and Exception vector cannot be set until memory devices are connected to the Nios II processor
接着点击next
Cancel
< Back
Next >
Finish



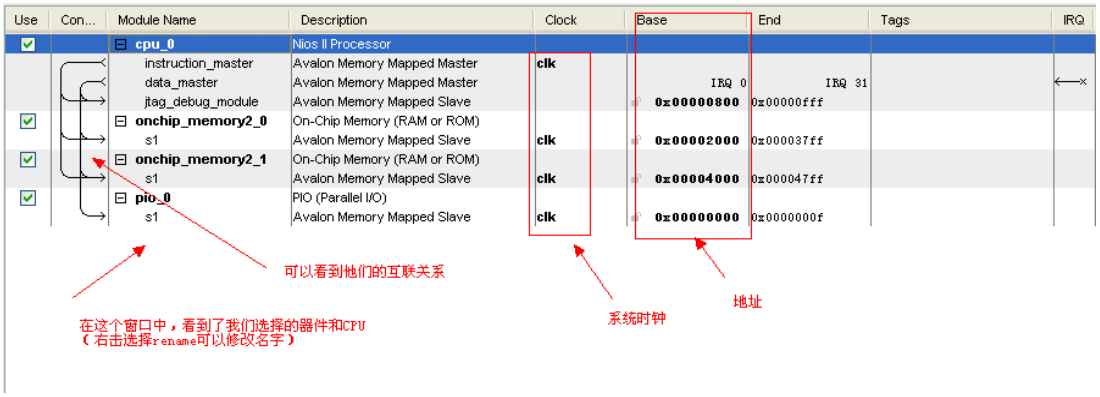

接下来选择程序存储区和数据存储器还有通用 8 位 I0 口。


(RAM or ROM)
About
Documentation
Parameter Settings
Memory type
选择ROM作为程序存储区
RAM (Writable)
ROM (Read-only)
Dual-port access
Read During Write Mode:
DONT_CARE
ROM时建议大家选上，作用是当编译好软件代码后Quartus再次重新全编译后下载，hex也同时下载进FPGA
Block type:
Auto
Initialize memory content
存储块类型，选择自动分配
Memory will be initialized from onchip_memory2_0.hex
Size
数据宽度
Data width:
32
ROM大小
Total memory size:
6
KBytes
Minimize memory block usage (may impact fmax)
Read latency
Slave s1:
1
Slave s2:
1
Memory initialization
Enable non-default initialization file
User-created initialization file:
onchip_memory2_0
.hex
Enable In-System Memory Content Editor feature
Instance ID:
NONE
数据存储器RAM和这里的设置基本相同，区别是不勾选initialize memory content和存储区大小不同
RAM选择2K就够了
设置完上面后点finish
Cancel
Finis

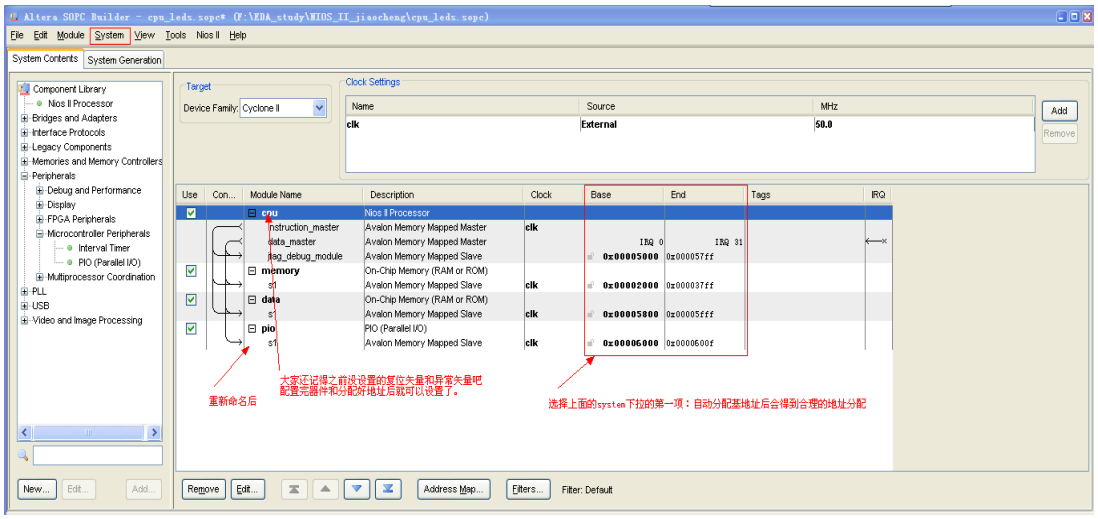
然后配置通用 IO 口：

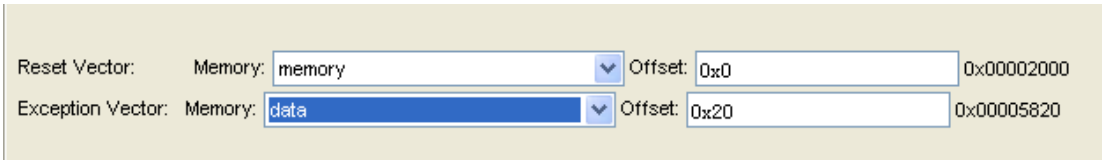

System Contents
System Generation
Component Library
Nios II Processor
Bridges and Adapters
Interface Protocols
Legacy Components
Memories and Memory Controllers
Peripherals
外围器件选择
Debug and Performance
Display
FPGA Peripherals
Microcontroller Peripherals
双击通用IO
Interval Timer
PIO (Parallel I/O)
Multiprocessor Coordination
PLL
USB
Video and Image Processing


（下面是选择器件区窗口，可以看到上面配置的器件!!）

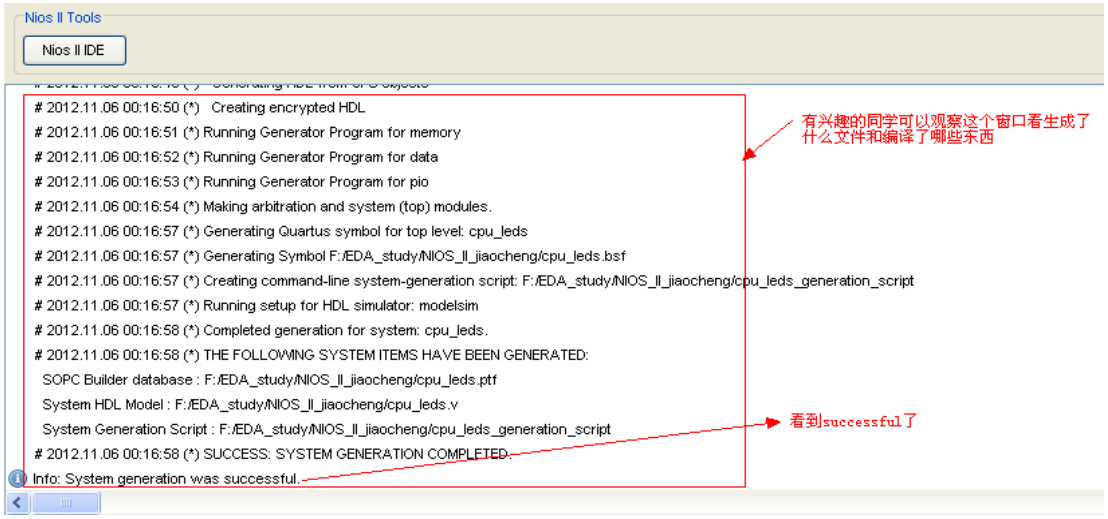



双击 CPU，把复位矢量和异常矢量分配好：

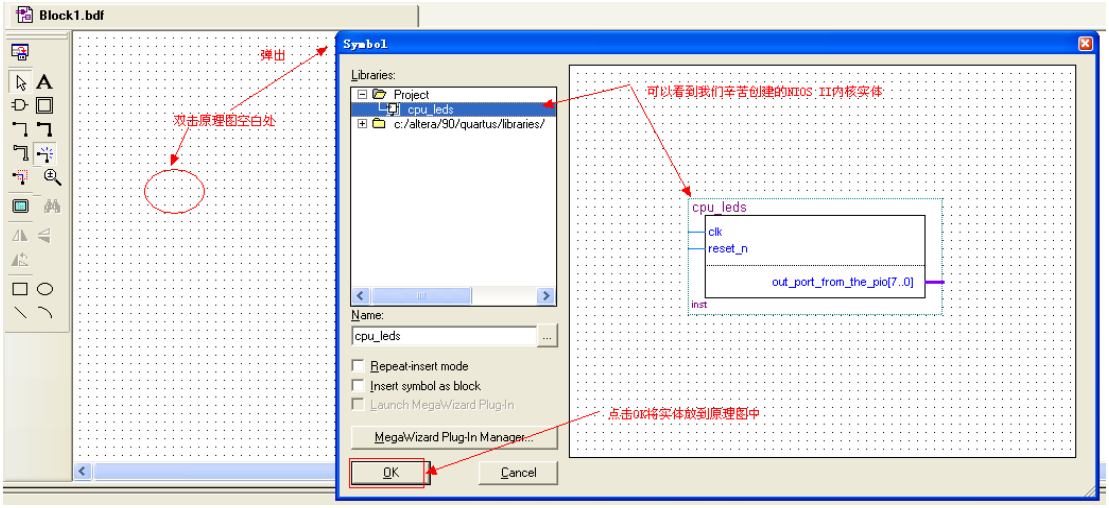



第四步：好了，到现在为止我们已经配置好一个简单的 NIOS II 内核，接下来就是漫长的等待吧！

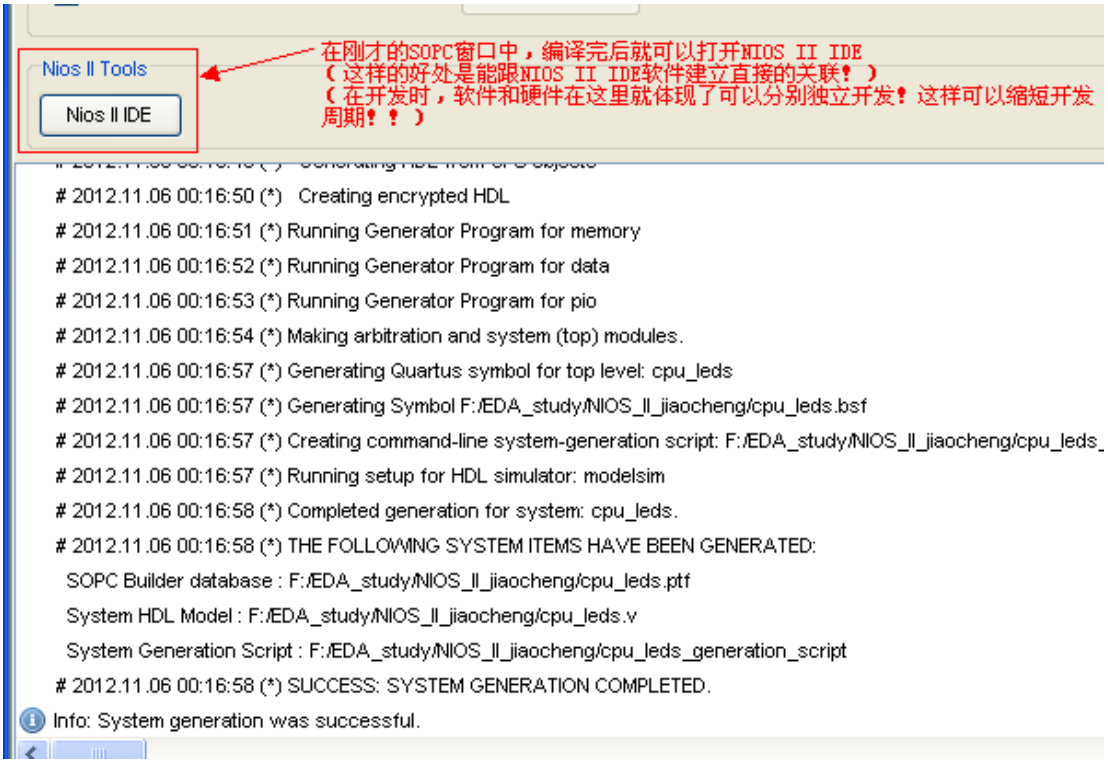




第五步：在原理图中添加内核 symbol

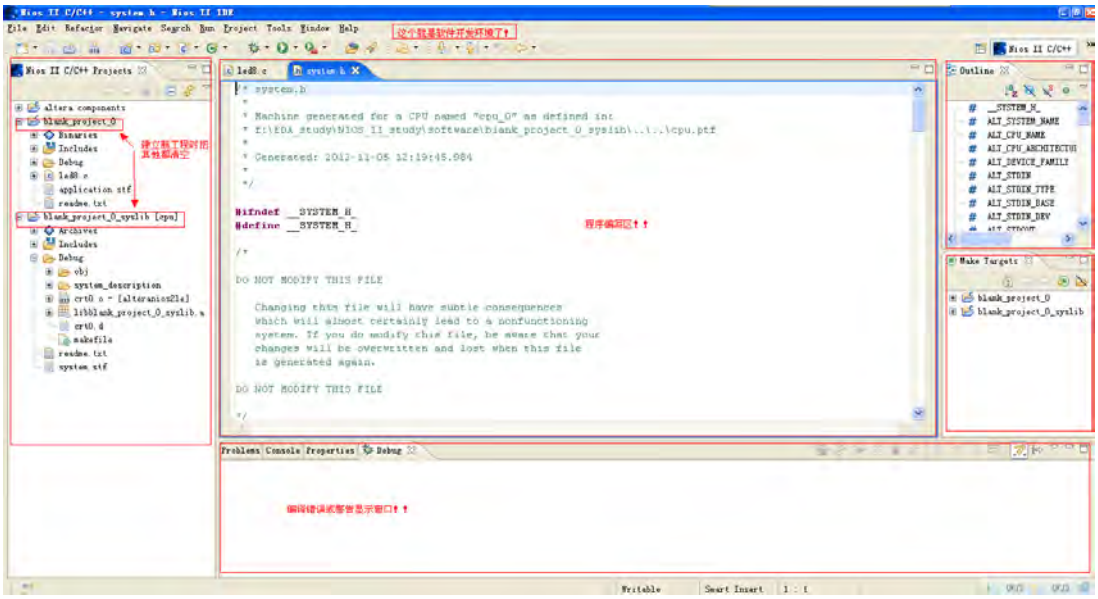



（其中内核复位信号是低电平复位!!）

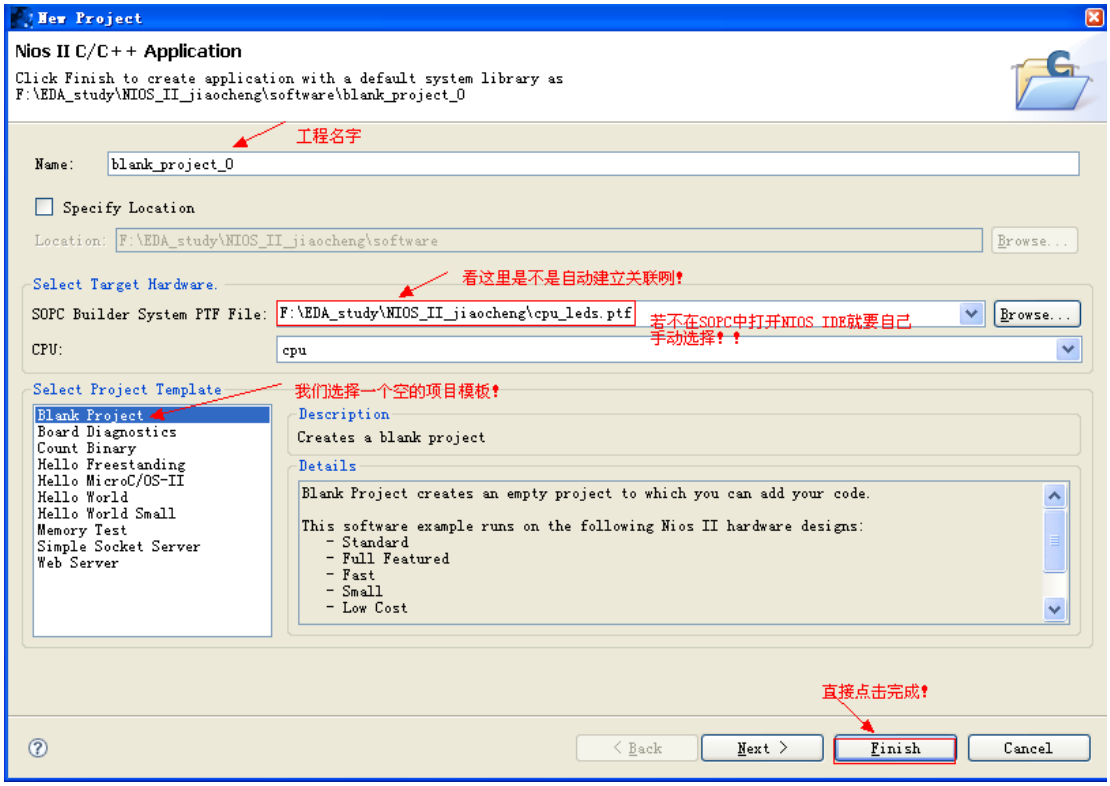
第六步：软件（C 或 C++）编程



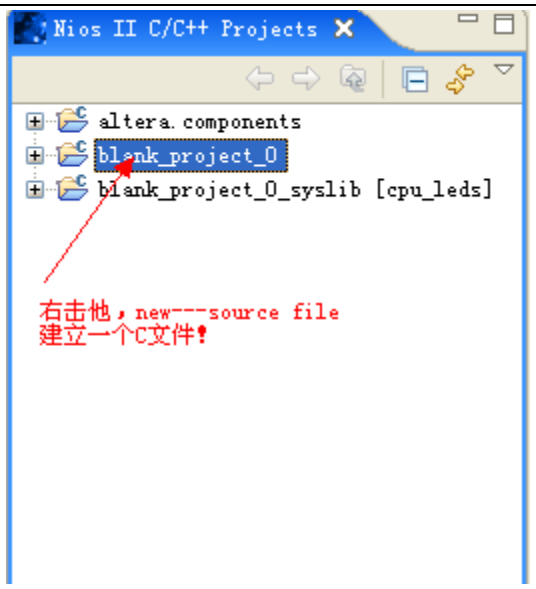

现在看一下 NIOS II 软件编写窗口吧，是不是跟单片机开发的有点类似！

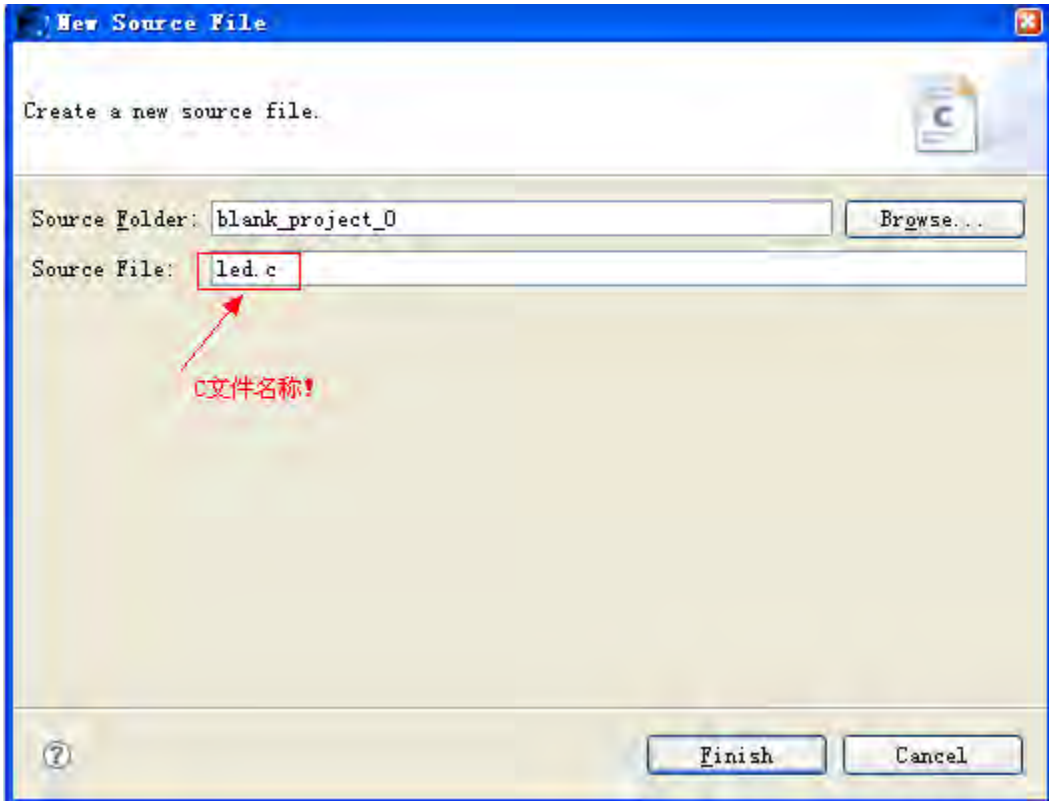


清空其他工程后，建立一个应用项目：file—new—Nios II C/C++ Application




然后建立一个 C 程序文件：



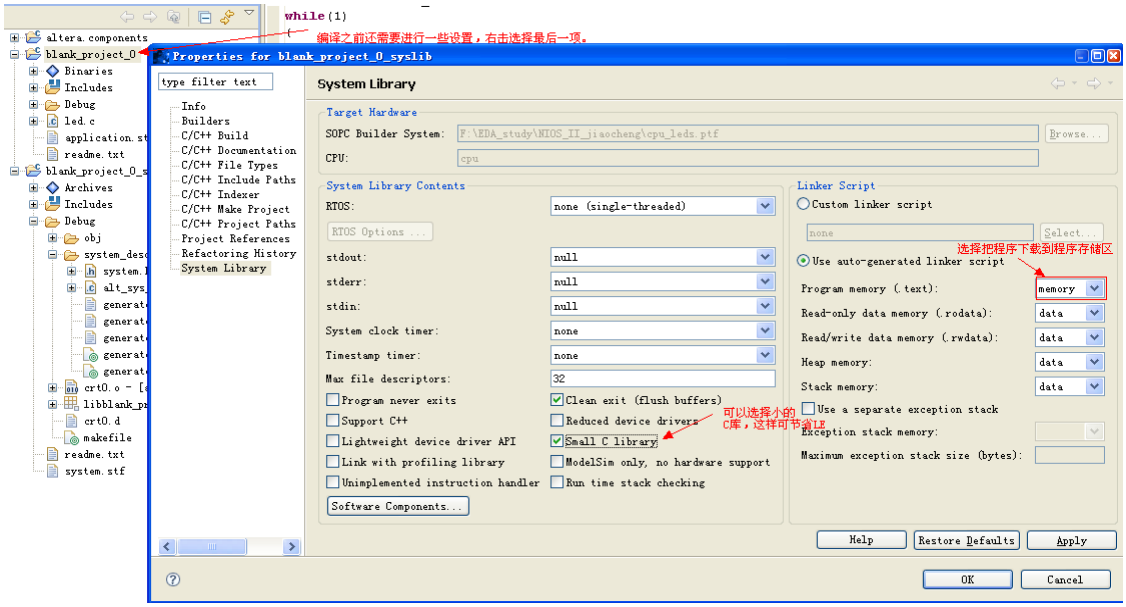

大家可以先把程序写下来，按下保存！等下会有相应的解释。。。

```
#include "system.h"  内核硬件描述信息
#include "altera_avalon_pio_regs.h"  PIO内核硬件描述心信息
void delay(void);

int alt_main(void) 主函数
{
unsigned char led_data;
unsigned char led_code;
while(1)
{
 for(led_data=0;led_data<8;led_data++)
 {
  led_code=0x01<<led_data;
  IOWR_ALTERA_AVALON_PIO_DATA(PIO_BASE,led_code);
  delay();
 }
}
return 0;
}

void delay(void)
{
 unsigned int i;
 i=1000000;
 while(i>0)
 {
  i--;
 }
```

大家可以看到，这程序和单片机的程序还是有点类似的，不同的是寄存器和头文件等。。

地址，要根据system.h头文件里的填写，等下会讲解

内核提供的API函数！

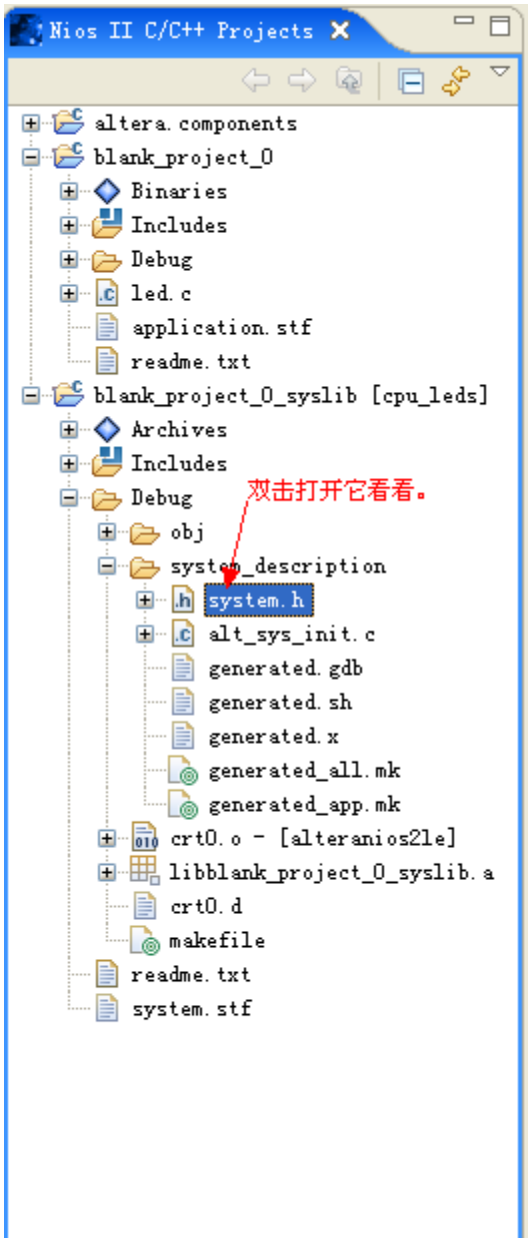



大家设置完后最好都保存下!!

好，现在可以编译工程了：project—build project，然后又是等待。。。。

趁这个时间来说下那两个头文件：

首先找到 system.h 头文件


Nios II C/C++ Projects
altera.components
blank_project_0
Binaries
Includes
Debug
led.c
application.stf
readme.txt
blank_project_0_syslib [cpu_leds]
Archives
Includes
Debug
双击打开它看看。
obj
system_description
system.h
alt_sys_init.c
generated.gdb
generated.sh
generated.x
generated_all.mk
generated_app.mk
crt0.o - [alteranios2le]
libblank_project_0_syslib.a
crt0.d
makefile
readme.txt
system.stf

```
#define PIO_NAME "/dev/pio"
#define PIO_TYPE "altera_avalon_pio"
#define PIO_BASE 0x00006000
#define PIO_SPAN 16
#define PIO_DO_TEST_BENCH_WIRING 0
#define PIO_DRIVEN_SIM_VALUE 0
#define PIO_HAS_TRI 0
#define PIO_HAS_OUT 1
#define PIO_HAS_IN 0
#define PIO_CAPTURE 0
#define PIO_DATA_WIDTH 8
#define PIO_RESET_VALUE 0
#define PIO_EDGE_TYPE "NONE"
#define PIO_IRQ_TYPE "NONE"
#define PIO_BIT_CLEARING_EDGE_REGISTER 0
#define PIO_BIT_MODIFYING_OUTPUT_REGISTER 0
#define PIO_FREQ 50000000
#define ALT_MODULE_CLASS_pio altera_avalon_pio

/*
```

大家找到IO口的硬件信息看看，这里的PIO_BASE就是刚才API形参的第一项！！

有兴趣的同学可以浏览下其他的宏定义，是不是有点熟悉的感觉，没错，那都是你制定内核的信息。

这个是从内核数据手册上截的图，大家看下，不懂的要谷歌翻一下哦！

## Software Files

The PIO core is accompanied by one software file, altera_avalon_pio_regs.h. This file defines the core's register map, providing symbolic constants to access the low-level hardware.

| Offset | Register Name | | R/W | (n-1) | ... | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| 0 | data | read access | R | Data value currently on PIO inputs | | | | |
| | | write access | W | New value to drive on PIO outputs | | | | |



这个是IO核的寄存器。可以看到他有四个寄存器。

这个是方向寄存器，我们在定制的时候已经设置了输出方式！

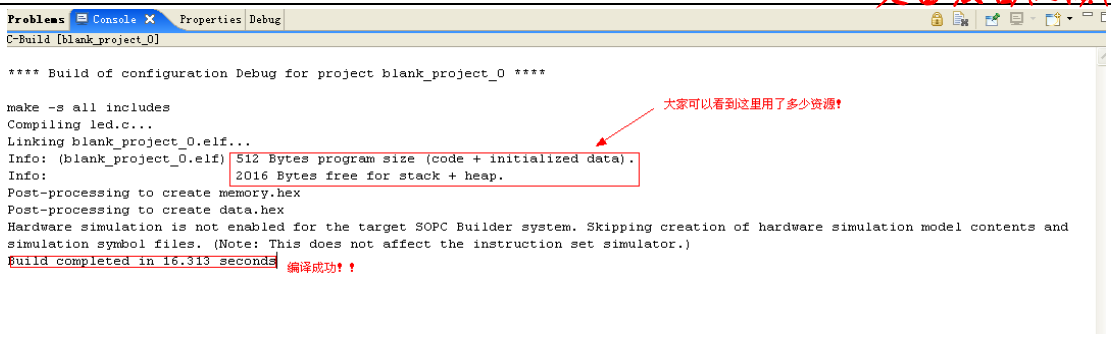


**Table 9–2.** Register Map for the PIO Core (Part 2 of 2)

| Offset | Register Name | R/W | (n-1) | ... | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 1 | direction *(1)* | R/W | Individual direction control for each I/O port. A value of 0 sets the direction to input; 1 sets the direction to output. | | | | |
| 2 | interruptmask *(1)* | R/W | IRQ enable/disable for each input port. Setting a bit to 1 enables interrupts for the corresponding port. | | | | |
| 3 | edgecapture *(1), (2)* | R/W | Edge detection for each input port. | | | | |

到这里相信大家都编译完了！那么下面就进入第七步吧！

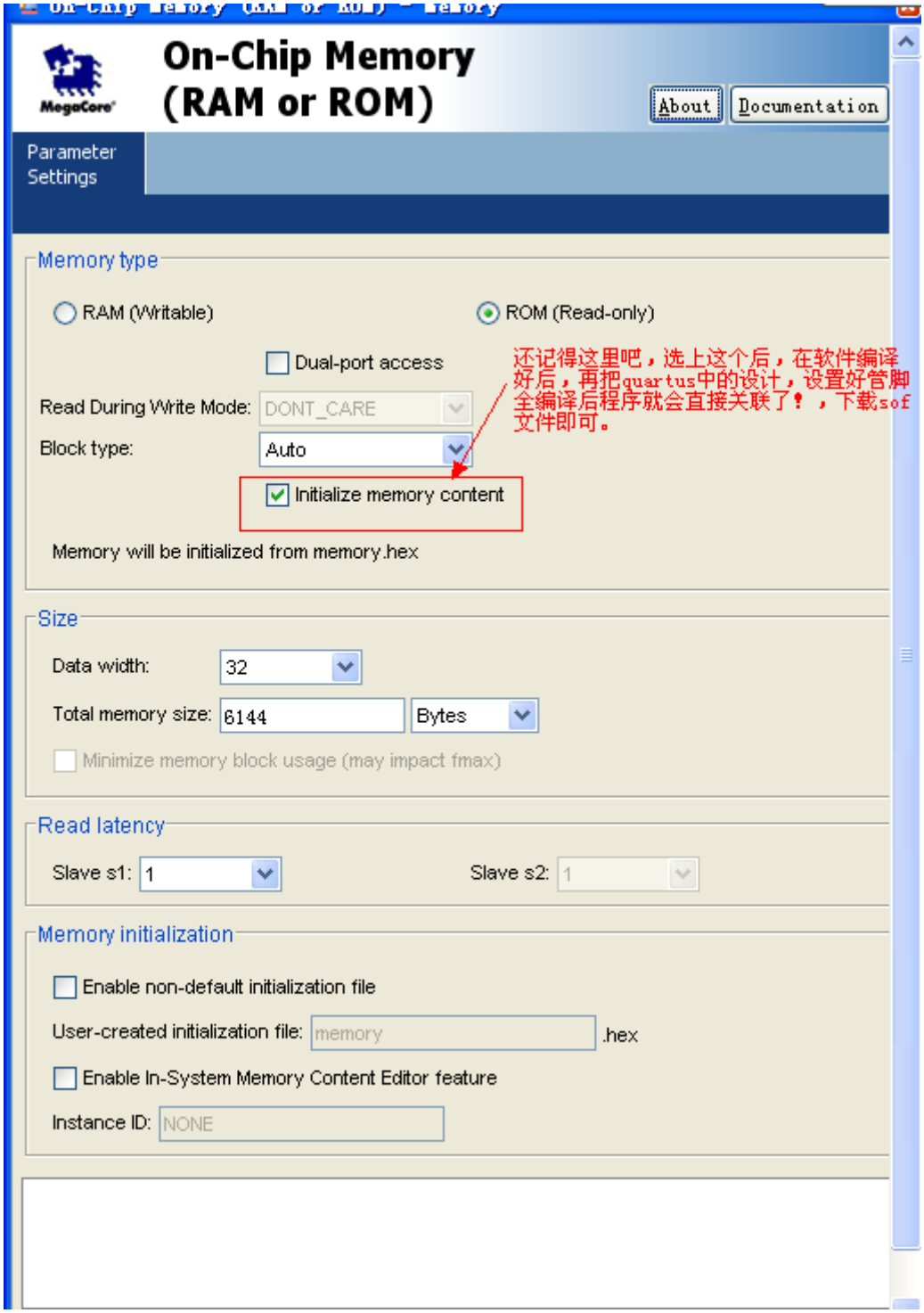
看下编译成功的样子：

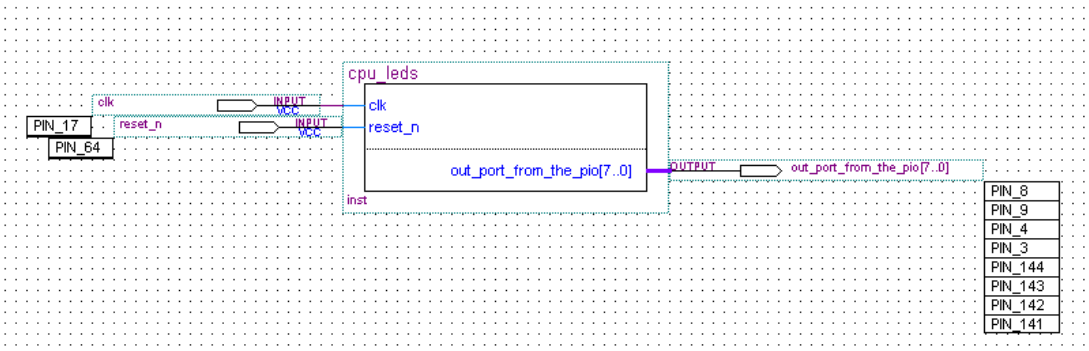



第七步：程序下载与调试

下面介绍两种方法把程序下载到 FPGA 芯片中：

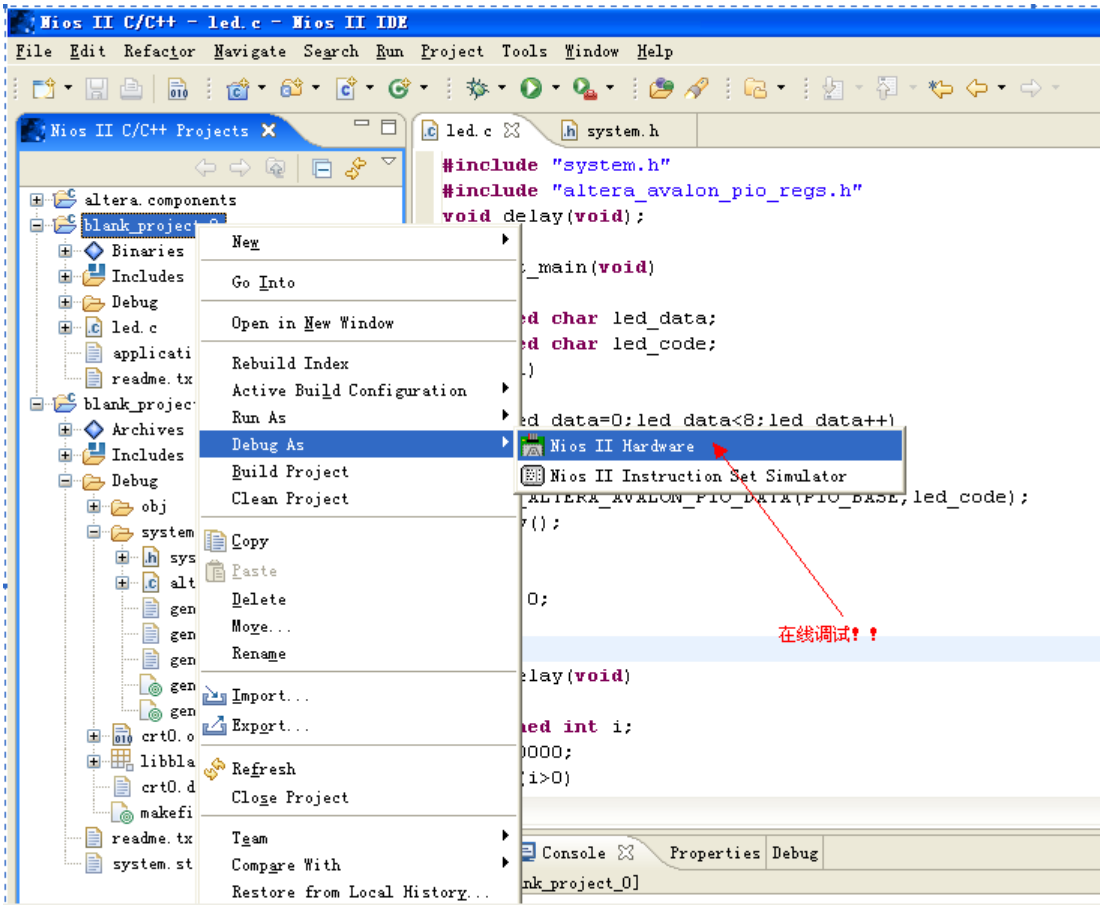
第一种方法：还记得这里吧！

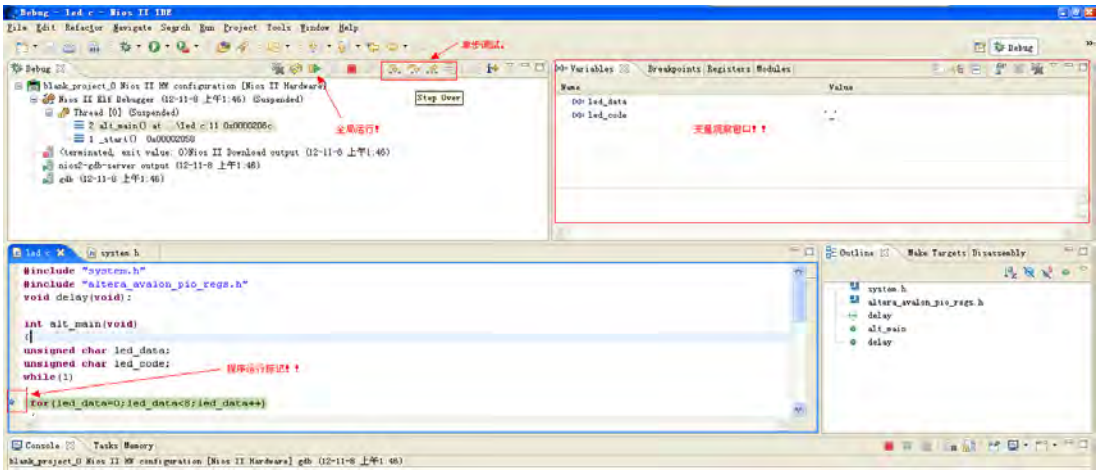

On-Chip Memory (RAM or ROM)
About
Documentation
Parameter Settings
Memory type
RAM (Writable)
ROM (Read-only)
Dual-port access
Read During Write Mode: DONT_CARE
Block type: Auto
Initialize memory content
Memory will be initialized from memory.hex
还记得这里吧，选上这个后，在软件编译好后，再把quartus中的设计，设置好管脚全编译后程序就会直接关联了！，下载sof文件即可。
Size
Data width: 32
Total memory size: 6144 Bytes
Minimize memory block usage (may impact fmax)
Read latency
Slave s1: 1
Slave s2: 1
Memory initialization
Enable non-default initialization file
User-created initialization file: memory .hex
Enable In-System Memory Content Editor feature
Instance ID: NONE




下载后就可以看到流水灯的效果了!!

第二种方法介绍在线调试（这个功能很强大把!）：
先把 sof 文件下载到 FPGA 中，然后选上






到此为止，相信大家都成功的看到自己定制的内核在成功运行了吧！

若大家对 NIOS II 处理器还有疑问的话，可以网上了解下！

http://baike.baidu.com/view/3079212.htm

也可以大西瓜 FPGA 交流群！

这次是入门教程，所以比较详细，接下来会有更多的例子讲解，敬请期待吧！