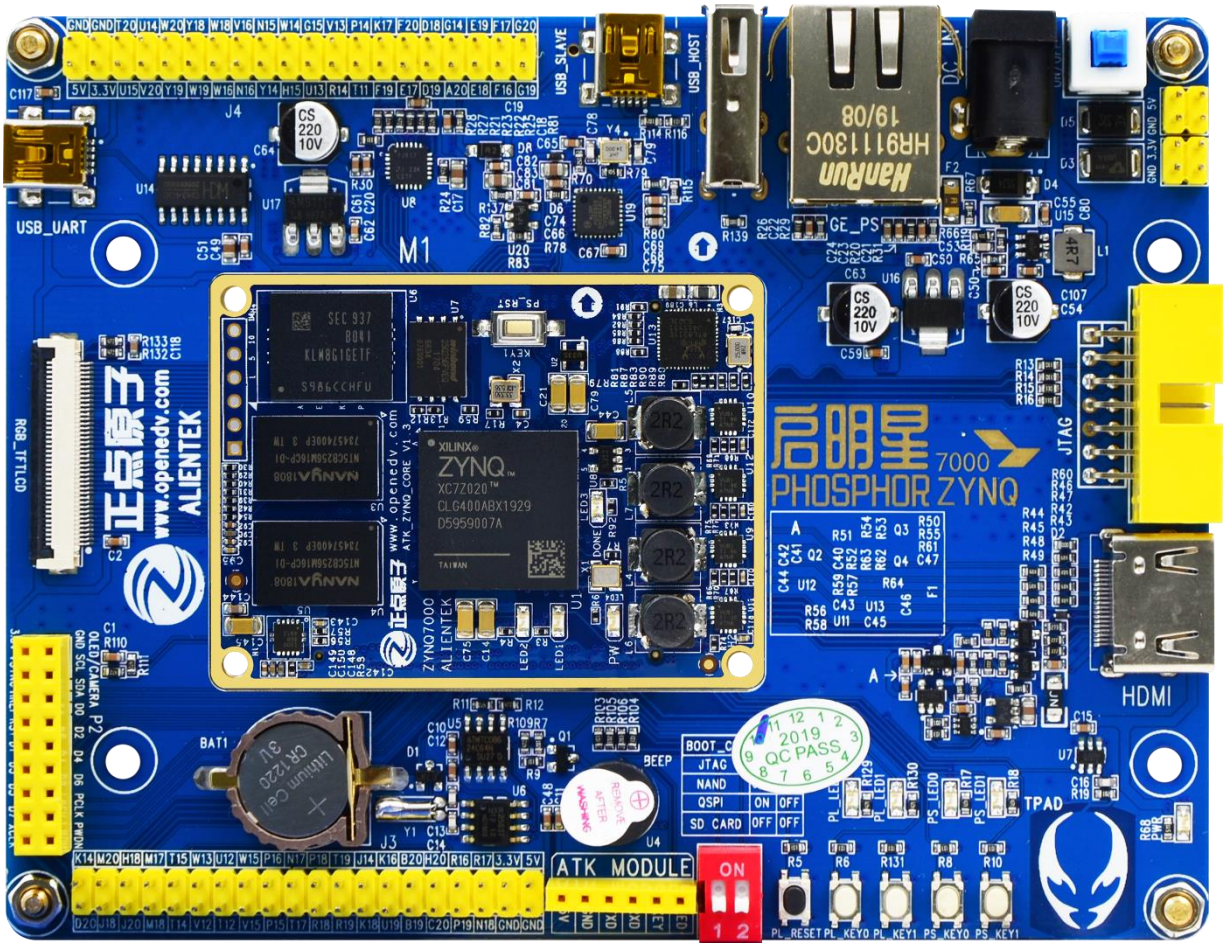
启明星 ZYNQ 之 FPGA 开发指南 V1.1.1

-正点原子 启明星 PHOSPHOR 开发板教程



正点原子 广州市星翼电子科技有限公司

淘宝店铺 1：http://eboard.taobao.com

淘宝店铺 2：http://openedv.taobao.com

技术支持论坛 (开源电子网) ：www.openedv.com

原子哥在线教学：www.yuanzige.com

官方网站：www.alientek.com

最新资料下载链接：http://www.openedv.com/posts/list/13912.htm

E-mail: 389063473@qq.com **QQ:** 389063473

咨询电话：020-38271790

传真号码：020-36773971

团队：正点原子团队

正点原子，做最全面、最优秀的嵌入式开发平台软硬件供应商。

友 情 提 示

如果您想及时免费获取“正点原子”最新资料，敬请关注正点原子微信公众平台，我们将及时给您发布最新消息和重要资料。

关注方法：

(1)微信“扫一扫”，扫描右侧二维码，添加关注

(2)微信→添加朋友→公众号→输入“正点原子”→关注

(3)微信→添加朋友→输入“alientek_stm32”→关注

目录

内容简介

启明星 ZYNQ 之 FPGA 开发指南将由浅入深的带领大家开启 ZYNQ 的学习之旅。本手册主要学习 ZYNQ 的 PL（Program Logic，可编程逻辑）部分，共分为硬件篇、软件篇、语法篇与实战篇共四个篇章。

硬件篇：硬件篇主要介绍本手册的硬件实验平台以及硬件资源详解。

软件篇：软件篇主要介绍 FPGA 常用开发软件的安装教程与使用方法。

语法篇：语法篇主要介绍 FPGA 的硬件描述语言 Verilog 的语法知识。

实战篇：实战篇主要通过 17 个实例带领大家一步步深入了解 FPGA。

本手册为启明星 ZYNQ 开发板的配套教程，在开发板配套的光盘里面，有开发板的原理图以及所有实例的完整代码，这些代码都有详细的注释，所有源码都经过我们严格测试，不会有任何编译错误。另外，源代码有我们生成好的 bit 文件（用于下载程序的文件），大家只需要通过下载器下载到开发板即可看到实验现象，亲自体验实验过程。

本手册不仅非常适合广大学生和电子爱好者学习 FPGA，其大量的实验以及详细的注释，也是公司产品开发的不二参考。

前　言

FPGA 自诞生以来，经历了从配角到主角的过程，由于 FPGA 飞速的发展，凭借其灵活性高、开发周期短、并行计算效率高等优势，使其应用到越来越多的领域中，如通信、消费电子、工业控制以及嵌入式等领域。

Zynq-7000 系列是 Xilinx 公司推出的全可编程片上系统（All Programmable SoC），包含 PS（Processing System，处理器系统）和 PL（Programmable Logic，可编程逻辑）两部分。Zynq SoC 整合了 ARM 双核 cortex-A9 处理器和 Xilinx 7 系列 FPGA 架构，使得它不仅拥有 ASIC 在能耗、性能和兼容性方面的优势，而且具有 FPGA 硬件可编程性的优点。

Zynq-7000 系列具有丰富的型号，如 XC7Z010、XC7Z020 和 XC7Z030 等，启明星 ZYNQ 底板搭配的核心板有两种型号，分别为 ZYNQ-7020（XC7Z020）和 ZYNQ-7010（XC7Z010）。XC7Z020 内部的 LC（Logic Cell，逻辑单元）达到 85K，XC7Z010 的内部 LC 为 28K。对于我们学习使用以及项目开发来说，已经足够了。

不管你是一个 FPGA 初学者，还是一个有经验的 FPGA 工程师，本手册都非常适合。尤其对于初学者，本手册将手把手的教你如何使用 FPGA 的开发软件 Vivado，包括新建工程、编译、下载调试等一系列步骤，让你轻松上手。

本手册的实验平台是启明星 ZYNQ 开发板，有这款开发板的朋友可以直接拿本手册配套的光盘上的例程在开发板上运行、验证。而没有这款开发板而又想要的朋友，可以上淘宝购买。当然你如果有了一款自己的开发板，而又不想再买，也是可以的，只要你的板子上有启明星 ZYNQ 开发板上的相同资源（需要实验用到的），代码一般都是可以通用的，你需要做的就只是把引脚 IO 的约束稍做修改，使之适合你的开发板即可。

最后，手册在编写过程中难免会有出错的地方，如果大家发现手册中有什么错误的地方，还请告诉本人一声，本人邮箱：3222632799@qq.com，也可以去 www.openedv.com 论坛给我留言。在此先向各位朋友表示真心的感谢。

第一篇　硬件篇

实践出真知，要想学好 FPGA，实验平台必不可少！本篇我们将详细介绍用来学习 FPGA 的硬件平台：启明星 ZYNQ 开发板。通过该篇的介绍，你将了解到我们的学习平台启明星 ZYNQ 开发板的功能及特点。

为了让读者更好的使用启明星 ZYNQ 开发板，本篇还介绍了开发板的一些使用注意事项，请读者在使用开发板的时候一定要注意。

第一章　ZYNQ 简介

ZYNQ 是赛灵思公司（Xilinx）推出的新一代全可编程片上系统（APSoC），它将处理器的软件可编程性与 FPGA 的硬件可编程性进行完美整合，以提供无与伦比的系统性能、灵活性与可扩展性。与传统 SoC 解决方案不同的是，高度灵活的可编程逻辑（FPGA）可以实现系统的优化和差异化，允许添加定制外设与加速器，从而适应各种广泛的应用。

本章包括以下几个部分：

1.1　ZYNQ 简介

1.2　FPGA 简介

1.3　ZYNQ PL 简介

1.4　ZYNQ PS 简介

1.1 ZYNQ 简介

Zynq-7000 系列是 Xilinx 于 2010 年 4 月推出的行业第一个**可扩展处理平台**，旨在为视频监视、汽车驾驶员辅助以及工厂自动化等**高端嵌入式应用**提供所需的处理能力与计算性能。这款基于 ARM 处理器的 SoC 可满足复杂嵌入式系统的高性能、低功耗和多核处理能力等要求。

ZYNQ 的本质特征，是它组合了一个双核 ARM Cortex-A9 处理器和一个传统的现场可编程门阵列（FPGA）逻辑部件。由于该新型器件的可编程逻辑部分基于赛灵思 28nm 工艺的 7 系列 FPGA，因此该系列产品的名称中添加了“7000”，以保持与 7 系列 FPGA 的一致性，同时也方便日后本系列新产品的命名。

ZYNQ 的全称是 Zynq-7000 All Programmable SoC，也就是说，ZYNQ 实际上是一个片上系统（System on Chip，SoC）。那么，什么是“SoC”？

一个能够实现一定功能的电路系统由多个模块构成，如处理器、接口、存储器、模数转换器等等。这些功能模块可以由分立的器件来实现，然后在印刷电路板（PCB）上组合起来，最终形成板上系统（System-on-a-Board）。板上系统的示意图如下所示：

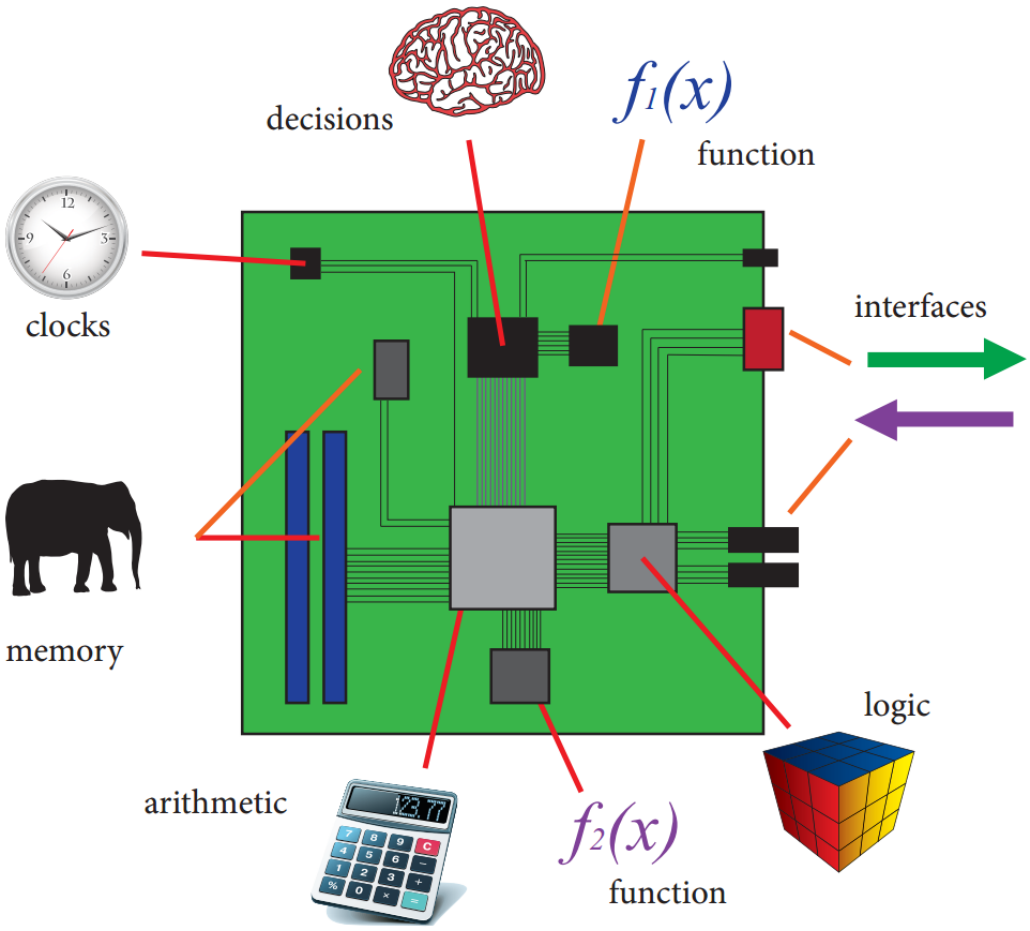


图 1.1.1 板上系统

在上图所示的板上系统中，绿色的矩形代表印刷电路板（PCB），上面各种颜色的小矩形代表了系统中各个功能模块，如存储器等。这些模块的功能都由一个个独立的硅芯片分别实现的，它们之间通过 PCB 上的金属走线连接，最终构成一个完整的系统。

而片上系统（System-on-Chip）指的是在单个硅芯片就可以实现整个系统的功能，其示意图如下所示：

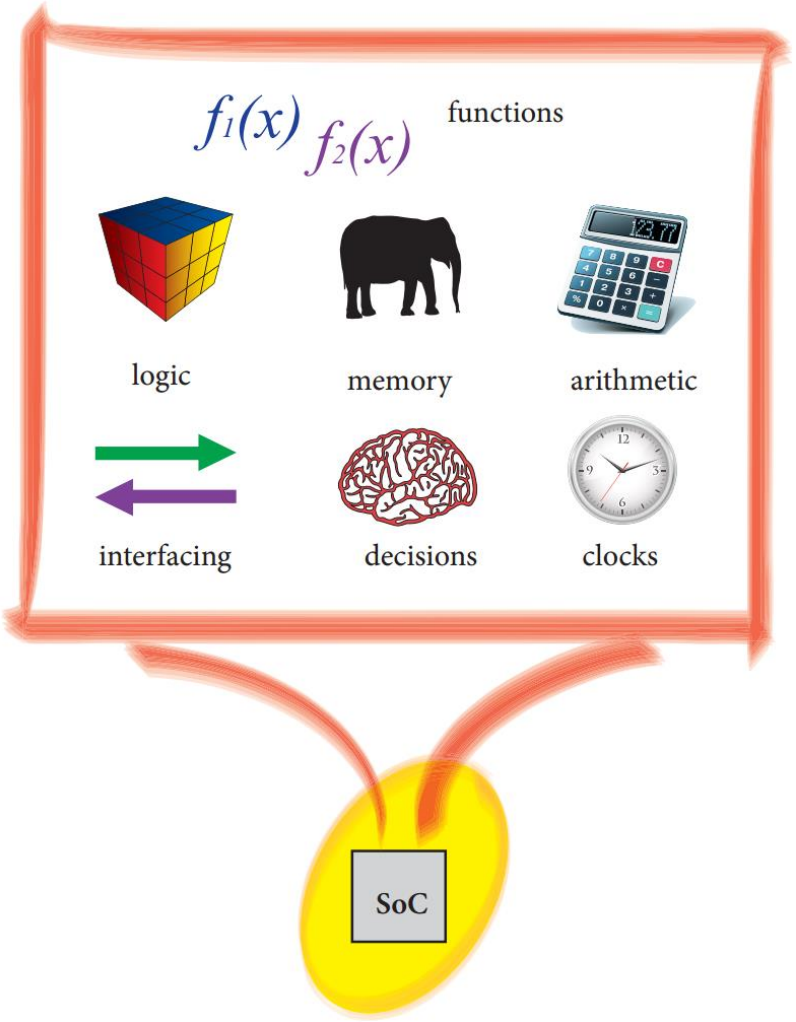


图 1.1.2 片上系统

如上图所示，片上系统 SoC 在一个芯片里就实现了存储、处理、逻辑和接口等各个功能模块，而不是像板上系统那样，需要用几个不同的物理芯片来实现。与板上系统相比，SoC 的解决方案成本更低，能在不同的系统单元之间实现更快更安全的数据传输，具有更高的整体系统速度、更低的功耗、更小的物理尺寸和更好的可靠性。

在过去，SoC 这个术语常用于指专用集成电路（Application Specific Integrated Circuit，ASIC）。基于 ASIC 的 SoC 的代表性例子包括在 PC、平板和智能手机上使用的处理器，如华为旗舰手机中的麒麟系列芯片。这些处理器典型地是由至少两个处理器核、存储器、图形处理器、接口和其他功能模块组合起来的。基于 ASIC 的 SoC 的主要缺点有两个：1、开发周期长且成本巨大；2、缺乏灵活性。开发 ASIC 时不可重用的工程投入是巨大的，使得这种 SoC 类型只适合于大批量而且寿命有限的产品中。

ASIC SoC 的局限性导致它们不适用于很多应用，特别是当快速投入市场能力、灵活性和升级能力已经成为重要的关键因素。对于小批量或中批量的产品，ASIC SoC 也不是好的解决方案。

可编程片上系统（SOPC，System-on-Progammable-Chip）为上述应用提供了一个更灵活的解决方案：一种在可编程、可重新配置的芯片上实现的 SoC。其中，可编程的芯片指的就是 FPGA。FPGA 天生的灵活性使其可以被随心所欲地重新配置，以实现不同系统的功能，包括嵌入式处理器。和使用 ASIC 来实现 SoC 相比，FPGA 能构成更为基础灵活的平台，方便系统的升级。

相比于 SOPC，ZYNQ 为实现灵活的 SoC 提供了一个更加理想的平台：Xilinx 将其打造成“**全可编程 片**

上系统（**APSoC**，All-Programmable SoC）”。它将处理器的软件可编程性与 FPGA 的硬件可编程性进行完美整合，以提供无与伦比的系统性能、灵活性与可扩展性。

ZYNQ 是由两个主要部分组成的：一个由双核 ARM Cortex-A9 为核心构成的处理系统（PS，Processing System），和一个等价于一片 FPGA 的可编程逻辑（PL，Programmable Logic）部分。ZYNQ 架构的简化模型如下图所示：

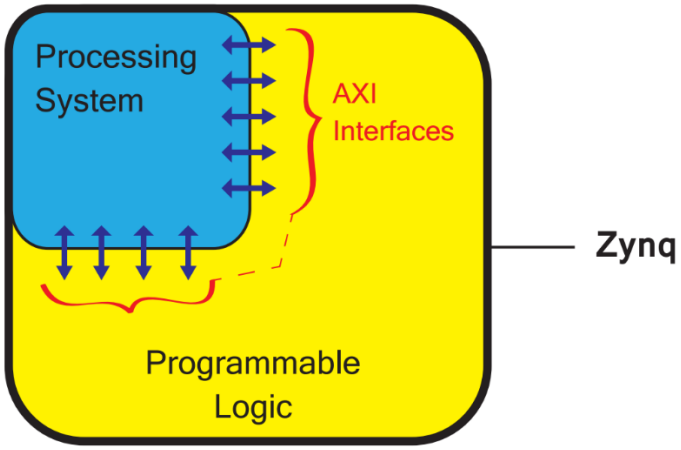


图 1.1.3 ZYNQ 架构简化模型

在上图中，PS 具有固定的架构，包含了处理器和系统的存储器；而 PL 完全是灵活的，给了设计者一块“空白画布”来创建定制的外设。

在 ZYNQ 上，ARM Cortex-A9 是一个应用级的处理器，能运行像 Linux 这样的操作系统，而可编程逻辑是基于 Xilinx 7 系列的 FPGA 架构。ZYNQ 架构实现了工业标准的 AXI 接口，在芯片的两个部分之间实现了高带宽、低延迟的连接。这意味着处理器和逻辑部分各自都可以发挥最佳的用途，而不会产生在两个分立器件之间的接口开销。与此同时，又能获得系统被简化为单一芯片所带来的好处，包括物理尺寸和整体成本的降低。

1.2 FPGA 简介

通过前面的介绍，我们知道 ZYNQ 中集成了 ARM 处理器与 FPGA。ZYNQ 作为一款全可编程 SoC，其中 FPGA 的硬件可编程性功不可没。那么 FPGA 是什么呢，它的灵活性又从何而来呢？

1） 数字集成电路的发展

在数字集成电路中，门电路是最基本的逻辑单元，用以实现最基本的逻辑运算（**与**、**或**、**非**）和复合逻辑运算（**与非**、**异或**等）。与上述逻辑运算相对应，常用的门电路有**与**门、**或**门、**非**门、**与非**门、**异或**门等，其电路符号如下图所示：

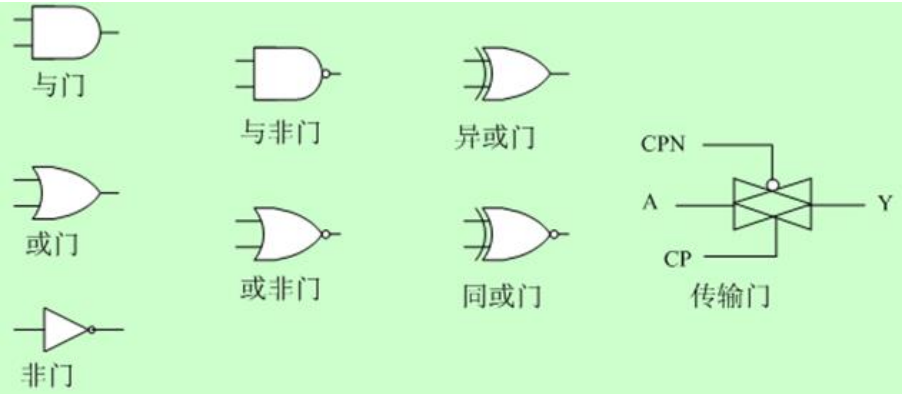


图 1.2.1 基本的门电路符号

在最初的数字逻辑电路中，每个门电路都是用若干个分立的半导体器件和电阻、电容连接而成的。不难想象，用这种单元电路组成大规模的数字电路是非常困难的，这就严重制约了数字电路的普遍应用。1961 年，美国德州仪器公司（TI）率先将数字电路的元器件制作在同一片硅片上，制成了集成电路（Intergrated Circuits，IC），并迅速取代了分立器件电路。

早期的数字逻辑设计需要设计师在一块电路板上或者如下图所示的面包板上用导线将多个芯片连接在一起。每个芯片包含一个或多个逻辑门，或者一些简单的逻辑结构（比如触发器或多路复用器等）。如下图中所示的芯片是在 1960 和 1970 年代，很多设计中都会使用的德州仪器 7400 系列的器件。

图 1.2.2 使用 74 系列器件搭建的电路

自 20 世纪 60 年代以来，随着集成电路工艺水平的不断进步，集成电路的集成度也不断提高。数字集成电路经历了从小规模集成电路（Small Scale Integrated circuit，**SSI**），到中规模集成电路（Medium Scale Integrated circuit，**MSI**），再到大规模集成电路（Large Scale Integrated circuit，**LSI**），然后是超大规模集成电路（Very Large Scale Integrated circuit，**VLSI**），以及甚大规模集成电路（Ultra Large Scale Integrated circuit，**ULSI**）的发展过程。今天我们已经可以把十分复杂的数字系统制作在一个很小的硅片上，构成“片上系统”。

2） FPGA 的由来

我们从逻辑功能的特点上将数字集成电路分类，可以分为通用型和专用型两类。前面介绍到的中、小规模集成电路（如 74 系列）都属于通用型数字集成电路。它们的逻辑功能都比较简单，而且是固定不变的。由于它们的这些功能在组成复杂数字系统时经常要用到，所以这些器件具有很强的通用性。

从理论上来讲，用这些通用型的中、小规模集成电路可以组成任何复杂的数字系统。随着集成电路的集成度越来越高，如果能把所设计的数字系统做成一片大规模集成电路，则不仅能减小电路的体积、重量和功耗，而且可以使电路的可靠性大为提高。像这种为某种专门用途而设计的集成电路称为专用集成电路，即所谓的 **ASIC**（Application Specific Integrated Circuit）。

ASIC 的使用在生产、生活中非常普遍，比如手机、平板电脑中的主控芯片都属于专用集成电路。

图 1.2.3 华为 Mate 30 手机中的麒麟 990 芯片

虽然 ASIC 有诸多优势，但是在用量不大的情况下，设计和制造这样的专用集成电路不仅成本很高，而且设计制造的周期也很长。可编程逻辑器件（Programmable Logic Device，**PLD**）的出现成功解决了这个矛盾。

可编程逻辑器件 PLD 是作为一种通用器件生产，但它的逻辑功能是由用户通过对器件进行编程来设定的。而且有些 PLD 的集成度很高，足以满足设计一般数字系统的需要。这样就可以由设计人员自行编程从而将一个数字系统“集成”在一片 PLD 上，做成“片上系统”（System on Chip，SoC），而不必去请芯片制造厂商设计和制作专用集成电路芯片了。

最后，我们再来总结一下这三种数字集成电路之间的差异。通用型数字集成电路和专用集成电路内部的电路连接都是固定的，所以它们的逻辑功能也是固定不变的。而可编程逻辑器件则不同，它们内部单元之间的连接是通过“写入”编程数据来确定的，写入不同的编程数据就可以得到不同的逻辑功能。

自 20 世纪 70 年代以来，PLD 的研制和应用得到了迅速的发展，相继开发出了多种类型和型号的产品。PLD 的发展历程如下图所示：

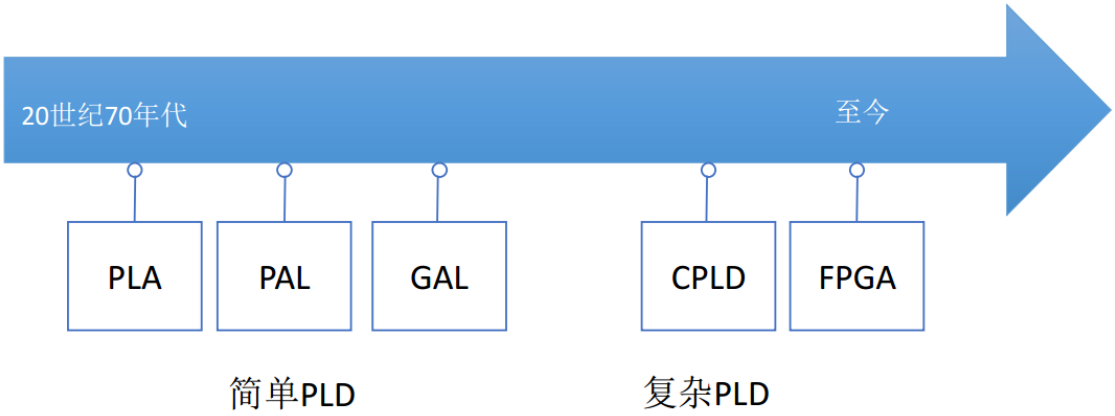


图 1.2.4 PLD 的发展历程

目前常见的 PLD 大体上可以分为 **SPLD**（simple PLD，简单 PLD）、**CPLD**（complex PLD，复杂 PLD）和 **FPGA**（field-programmable gate array，现场可编程门阵列）。SPLD 中又可分为 PLA、PAL 和 GAL 几种类型。FPGA 也是一种可编程逻辑器件，但由于在电路结构上与早期已经广为应用的 PLD 不同，所以采用 FPGA 这个名称，以示区别。

通过对数字电路的学习我们知道，任何一个逻辑函数式都可以变换成**与-或**表达式，因而任何一个逻辑函数都能用一级**与**逻辑电路和一级**或**逻辑电路来实现。PLD 最初的研制思想就来源于此。

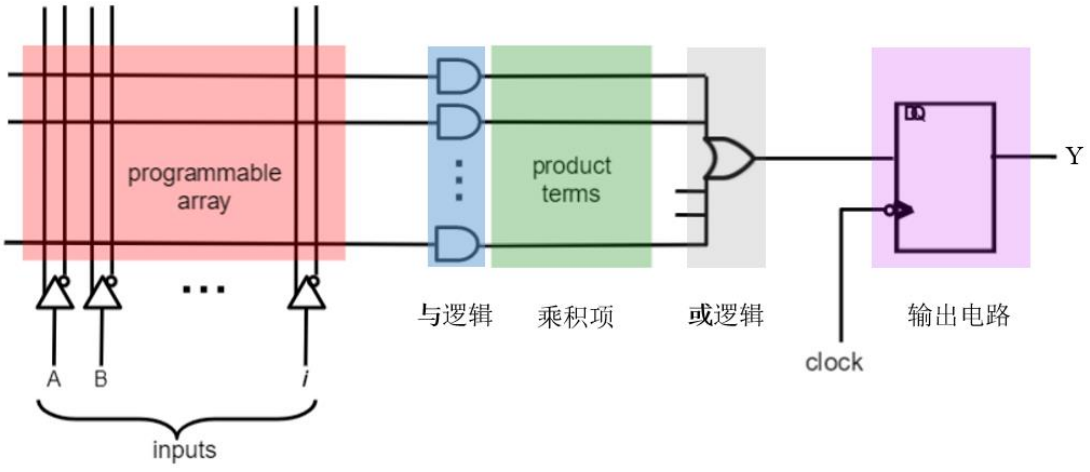


图 1.2.5 PAL 器件的基本电路结构

上图是 SPLD 中 PAL（可编程阵列逻辑）的电路结构图。通过对输入端（inputs）到**与**门之间的可编程阵列（programmable array）进行编程，利用 PAL 可以获得不同形式的组合逻辑函数。数字电路课程告诉我们，任何逻辑函数式都可以转化为若干**乘积项**（product tems）之和的形式，亦称“积之和”形式。通过对可编程阵列进行编程，**与**逻辑电路输出所需要的乘积项，再通过**或**逻辑电路将这些乘积项相加，就得到了最终的功能输出。然后该输出送给输出电路中的寄存器用于存储或者同步，当然也可以忽略寄存器直接输出。这就是 PAL 作为一种“可编程逻辑器件”能够实现不同逻辑功能的原理。

通过扩展 SPLD 的概念就可以得到 CPLD。CPLD 是复杂可编程逻辑器件，相当于将多个 PAL 用可编程互联阵列（Programmable Interconnect Array，PIA）连接起来，形成一个大的 PLD，如下图所示：

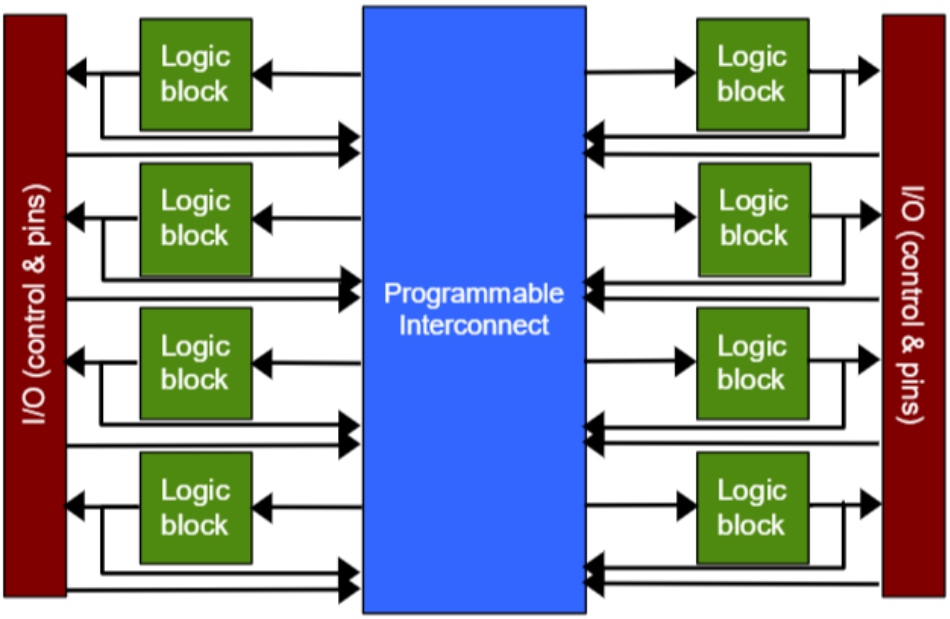


图 1.2.6 CPLD 结构示意图

上图中的 Logic block（逻辑块）通常被称为**逻辑阵列模块**，或者 **LAB**（Logic Array Block）。每个 LAB 相当于一个 PAL 电路，不同型号的 CPLD 器件可以包含十几个甚至上百个 LAB。通过 PIA 将这些 LAB 连接起来，就可以构成规模更大的逻辑电路了。另外，在 PAL 中，I/O 管脚是直接连接到逻辑的。而在 CPLD 中，I/O 管脚是通过 PIA 从器件的主要逻辑中分离出来的。I/O 管脚有它自己的控制逻辑，I/O 控制单元可以根据需要将相应的引脚设置成输入、输出或双向工作模式。

CPLD 相对于 SPLD 最大的优势就是拥有更大的逻辑资源和布线的可能性。CPLD 中 LAB 逻辑和 PIA 是完全可编程的，使得它具有在单芯片中非凡的设计灵活性。CPLD 的 I/O 特性和功能也远比 SPLD 中简单的 I/O 更有价值。

FPGA 是在 PAL、GAL 和 CPLD 等可编程逻辑器件的基础上进一步发展的产物，但是 FPGA 和其前辈 CPLD 有着非常大的差异。

FPGA 由许多“可配置逻辑模块”（Configurable Logic Block，**CLB**）、输入/输出单元（I/O Block，**IOB**）和分布式的可编程互联矩阵（Programmable Interconnection Matrix，**PIM**）组成。在 FPGA 中，CLB 被布置成阵列的形式，如图 1.2.7 所示。可编程的布线资源分布在 CLB 与 CLB 之间，像大城市的街道一样纵横联接。这些布线资源分为行互联和列互联，可以跨过整个器件，也可以是局部 CLB 之间的互联。

我们将图 1.2.6 与图 1.2.7 进行对比可以发现，FPGA 中的布线资源看上去似乎比 CPLD 中的互联阵列更简单，但它实际上提供了更大的功能性和连通性。FPGA 中的布线资源使得器件中所有的逻辑资源都可以

与芯片内其他资源进行通信，这种结构可以实现更大容量、低成本的逻辑器件。

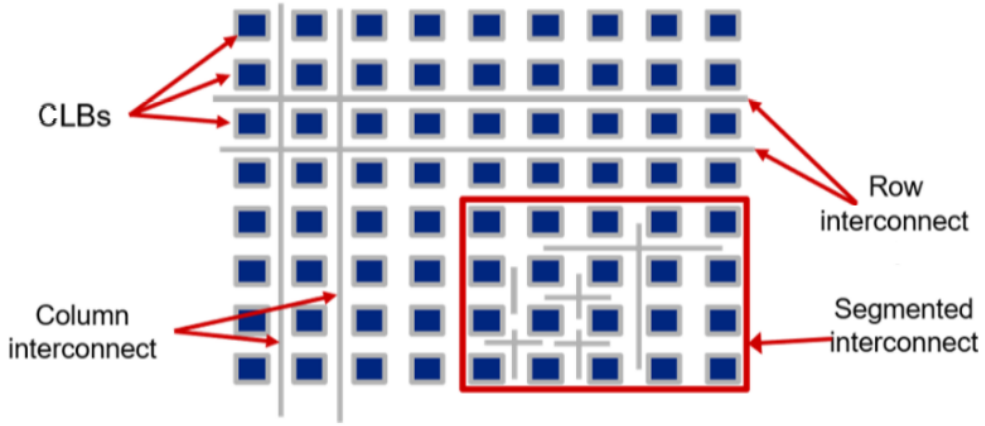


图 1.2.7 FPGA 结构示意图

在前面所讲的各种 SPLD 和 CPLD 电路中，都采用了**与或**逻辑阵列加上输出逻辑单元的结构形式。而 FPGA 则采用了完全不同的电路结构形式（查找表，LUT），有关 FPGA 的结构我们将在本章 ZYNQ PL 简介中作详细介绍。

3） FPGA 的用途

在介绍 FPGA 的用途之前，先给大家讲一个笑话：

话说一个资深工程师出国的时候带了一块 FPGA 开发板。
海关问道：“这是什么东西？”
工程师说：“FPGA 开发板”。
海关又问：“FPGA 是什么？”。
工程师回答说：“你想让它是什么，它就是什么（It can be whatever you want）”

能看懂这个笑话就说明你已经对 FPGA 有了一定的了解。作为一种可编程器件，FPGA 能实现任何数字器件的功能，上至高性能 CPU，下至简单的 74 电路，都可以用 FPGA 来实现。FPGA 就像一张白纸，任由你在上面涂鸦或者作画；FPGA 又如同一堆积木，随便你用来搭建城堡或者玩“过家家”。

FPGA 是什么这个问题可能不太好回答。但是如果说 FPGA 用来干什么，那么答案就显而易见了。你可能都还没意识到，在我们的生活中，FPGA 已经无处不在了。从你家里使用的高清电视，到附近的无线电接收塔；从银行门口的 ATM 机，到微软数据中心的服务器，都可以看到 FPGA 的身影。

图 1.2.8 FPGA 应用领域

如上图所示，FPGA 广范应用在汽车、军用装备、图像处理、有线和无线通信、医药，以及工业控制等诸多领域。

可编程逻辑天生就为并行地实现算法提供了理想的资源。比如在图像处理中，要同时对大量的像素点进行数学运算，而 FPGA 就很适合像这种像素点级别的图像处理所需的快速、并行的操作。

FPGA 并行的特性决定了它在某些特定行业应用上具有得天独厚的优势，例如在医疗领域。医学影像比普通图像纹理更多，分辨率更高，相关性更大。因此，为严格确保临床应用的可靠性，对图像的压缩、分割等预处理、图像分析及图像理解等要求更高。这些要求恰恰可以充分发挥 FPGA 的优势，通过 FPGA 加速图像压缩进程、删除冗余、提高压缩比、并确保图像诊断的可靠性。

在金融领域，由于采用流水线逻辑体系结构，数据流处理要求低延时，高可靠性。这在金融交易风险建模算法应用中是重要的关键点，而 FPGA 正具备了这种优势。类似的行业和领域还有很多，特别是在深度学习和神经网络，以及图像识别和自然语言处理等领域，FPGA 正显示出其独有的优势。

1.3 ZYNQ PL 简介

ZYNQ PL 部分等价于 Xilinx 7 系列 FPGA，因此我们将首先介绍 FPGA 的架构。

简化的 FPGA 基本结构由 6 部分组成，分别为可编程输入/输出单元、基本可编程逻辑单元、嵌入式块 RAM、丰富的布线资源、底层嵌入功能单元和内嵌专用硬核等，如下图所示：

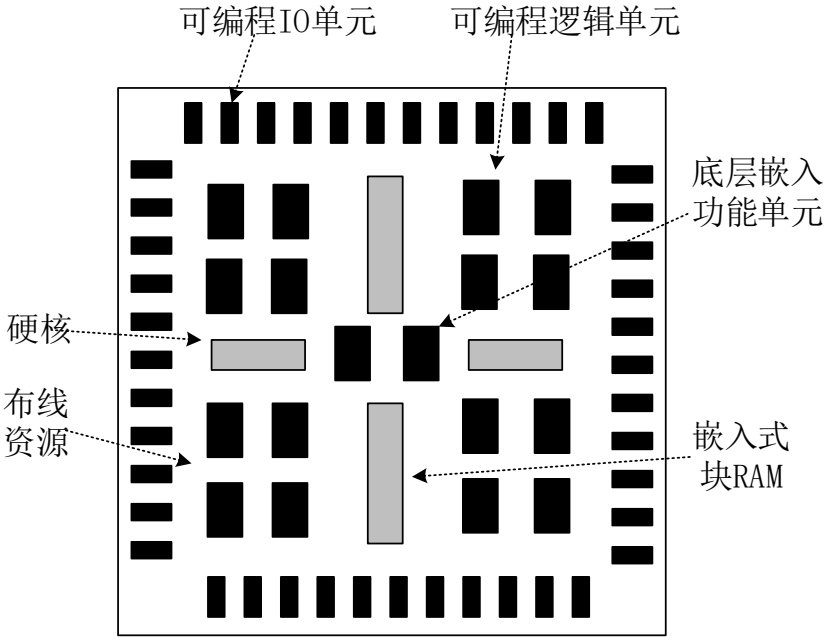


图 1.3.1 FPGA 基本结构

每个单元的基本概念介绍如下。

1) 可编程输入/输出单元

输入/输出（Input/Ouput）单元简称 I/O 单元，它们是芯片与外界电路的接口部分，完成不同电气特性下对输入/输出信号的驱动与匹配需求，为了使 FPGA 具有更灵活的应用，目前大多数 FPGA 的 I/O 单元被设计为可编程模式，即通过软件的灵活配置，可以适配不同的电气标准与 I/O 物理特性；可以调整匹配阻抗特性、上下拉电阻、以及调整驱动电流的大小等。

可编程 I/O 单元支持的电气标准因工艺而异，不同芯片商、不同器件的 FPGA 支持的 I/O 标准不同，一般来说，常见的电气标准有 LVTTL，LVCMOS，SSTL，HSTL，LVDS，LVPECL 和 PCI 等。值得一提的是，随着 ASIC 工艺的飞速发展，目前可编程 I/O 支持的最高频率越来越高，一些高端 FPGA 通过 DDR 寄存器技术，甚至可以支持高达 2Gbit/s 的数据数率。

ZYNQ 上的通用输入/输出功能（IOB）合起来被称作 SelectIO 资源，它们被组织成 50 个 IOB 一组。每个 IOB 有一个焊盘，是与外部世界连接来做单个信号的输入或输出的。每个 IOB 还包含一个 IOSERDES 资源，可以做并行和串行数据的可编程转换。

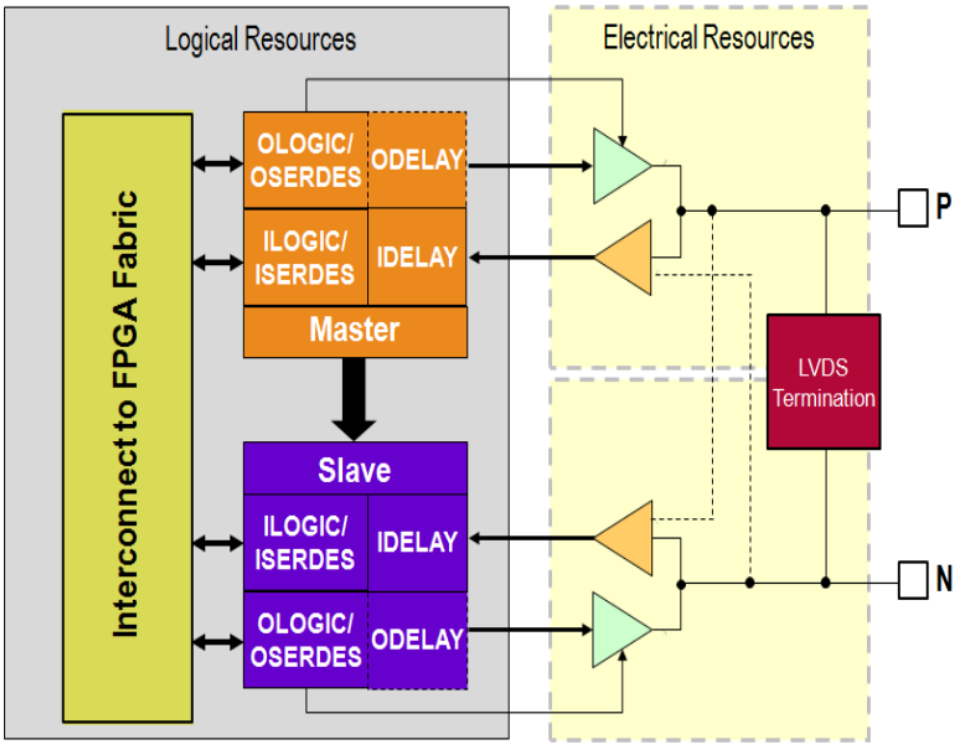


图 1.3.2 PL 中的 IOB

2) 基本可编程逻辑单元

基本可编程逻辑单元是可编程逻辑的主体，可以根据设计灵活地改变其内部连接与配置，完成不同的逻辑功能。FPGA 一般是基于 SRAM 工艺的，其基本可编程逻辑单元几乎都是由查找表（LUT，Look Up Table）和寄存器（Register）组成。Xilinx 7 系列 FPGA 内部查找表为 6 输入，查找表一般完成纯组合逻辑功能。FPGA 内部寄存器结构相当灵活，可以配置为带同步/异步复位或置位，时钟使能的触发器，也可以配置成锁存器，FPGA 依赖寄存器完成同步时序逻辑设计。

一般来说，比较经典的基本可编程逻辑单元的配置是一个寄存器加一个查找表，但是不同厂商的寄存器与查找表也有一定的差异，而且寄存器与查找表的组合模式也不同。当然这些可编程逻辑单元的配置结构随着器件的不断发展也在不断更新，最新的一些可编程逻辑器件常常根据需求设计新的 LUT 和寄存器的配置比率，并优化其内部的连接构造。

例如，Altera 可编程逻辑单元通常被称为 LE（Logic Element），由一个寄存器加一个 LUT 构成。Altera 大多数 FPGA 将 10 个 LE 有机地组合在一起，构成更大的功能单元——逻辑阵列模块（LAB，Logic Array Block）。LAB 中除了 LE 还包含 LE 之间的进位链，LAB 控制信号，局部互联线资源，LUT 级联链，寄存器级联链等连线与控制资源。

Xilinx 7 系列 FPGA 中的可编程逻辑单元叫 CLB（Configurable Logic Block，可配置逻辑块）每个 CLB 里包含两个逻辑片（Slice）。每个 Slice 由 4 个查找表、8 个触发器和其他一些逻辑所组成的。CLB 示意图如下所示：

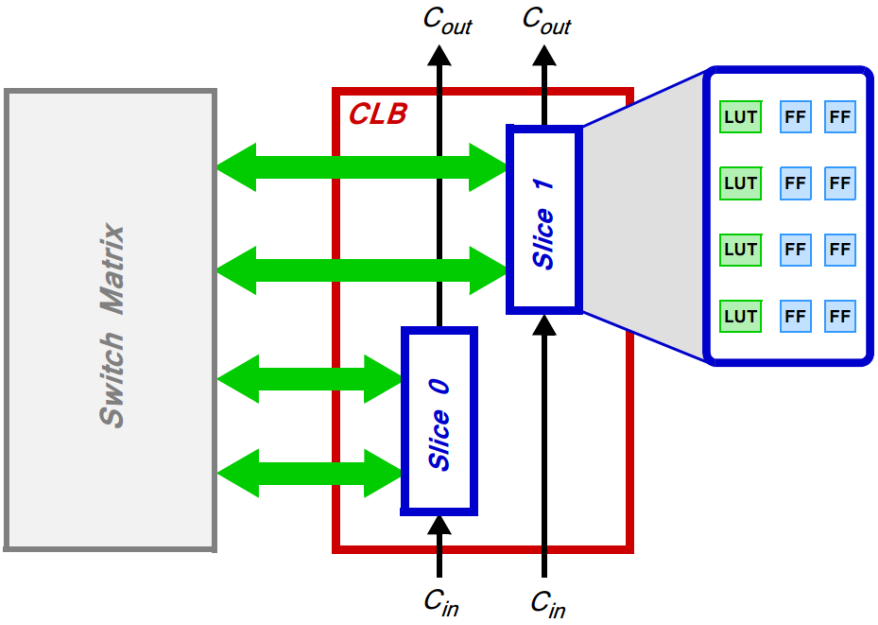


图 1.3.3 CLB 示意图

CLB 是逻辑单元的最小组成部分，在 PL 中排列为一个二维阵列，通过可编程互联连接到其他类似的资源。每个 CLB 里包含两个逻辑片，并且紧邻一个开关矩阵，如下图所示：

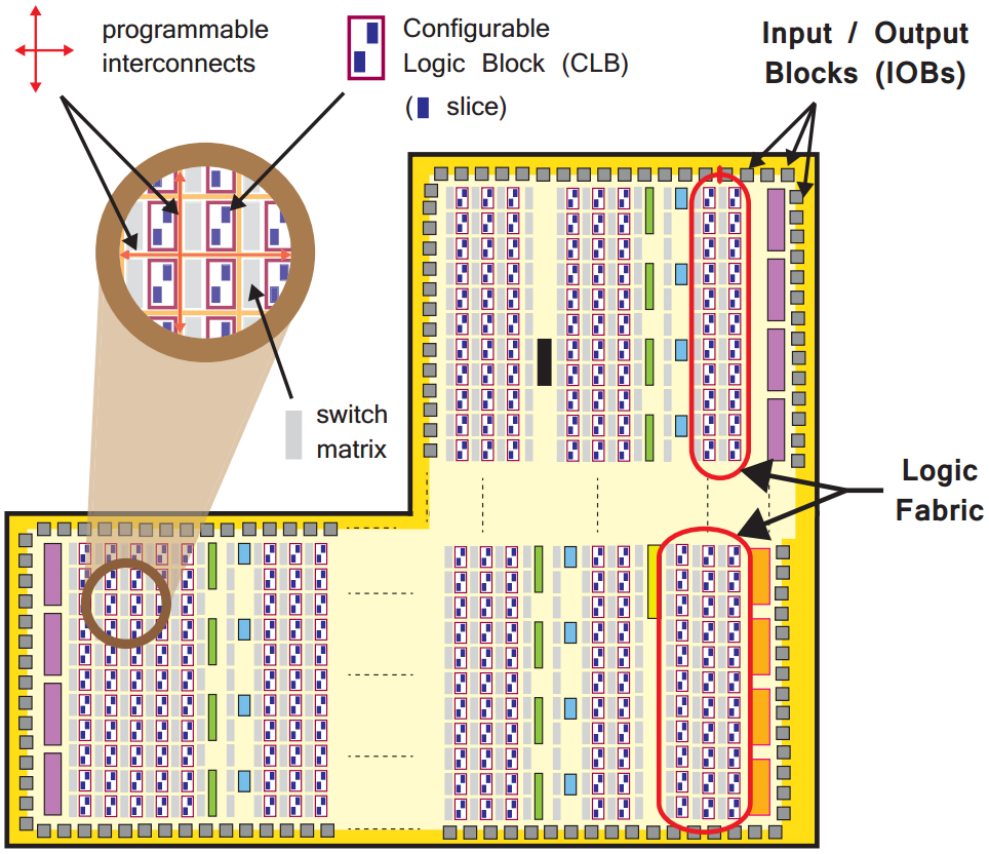


图 1.3.4 PL 中的 CLB

3) 嵌入式块 RAM

目前大多数 FPGA 都有内嵌的块 RAM（Block RAM），FPGA 内部嵌入可编程 RAM 模块，大大地拓

展了 FPGA 的应用范围和使用灵活性。不同器件商或不同器件族的内嵌块 RAM 的结构不同，Lattice 常用的块 RAM 大小是 9KBIT；Altera 的块 RAM 最灵活，一些高端器件内部同时含有 3 种块 RAM 结构，分别是 M512 RAM，M4K RAM，M9K RAM。

Zynq-7000 里的块 RAM 和 Xilinx 7 列 FPGA 里的那些块 RAM 是等同的，它们可以实现 RAM、ROM 和先入先出（First In First Out，FIFO）缓冲器。每个块 RAM 可以存储最多 36KB 的信息，并且可以被配置为一个 36KB 的 RAM 或两个独立的 18KB RAM。默认的字宽是 18 位，这样的配置下每个 RAM 含有 2048 个存储单元。RAM 还可以被“重塑”来包含更多更小的单元（比如 4096 个单元 x9 位，或 8192x4 位），或是另外做成更少更长的单元（如 1024 单元 x36 位 512x72 位）。把两个或多个块 RAM 组合起来可以形成更大的存储容量。PL 中的块 RAM 示意图如下所示：

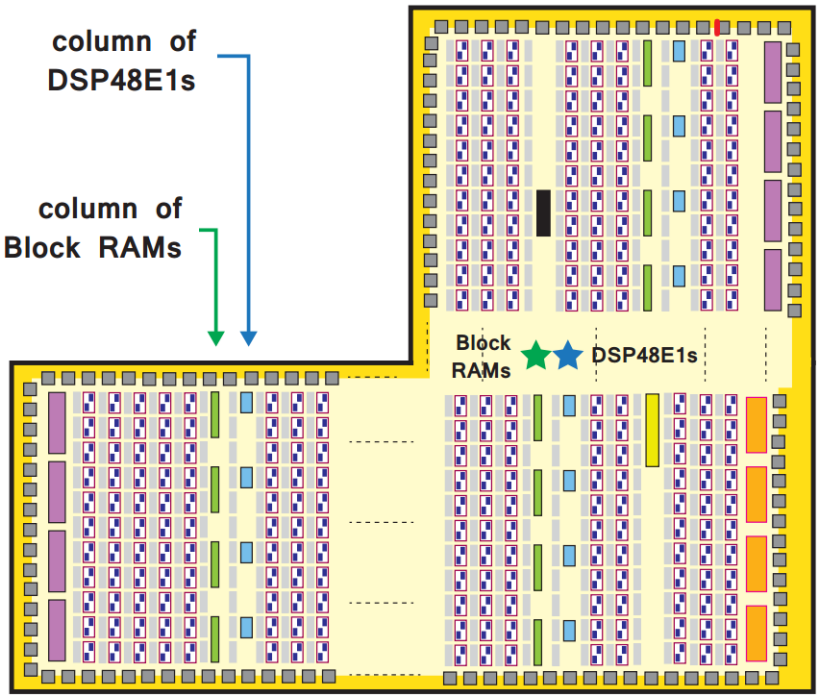


图 1.3.5 PL 中的 Block RAM

需要补充的一点是，除了块 RAM，还可以灵活地将 LUT 配置成 RAM，ROM，FIFO 等存储结构，这种技术被称为分布式 RAM。根据设计需求，块 RAM 的数量和配置方式也是器件选型的一个重要标准。

4) 丰富的布线资源

布线资源连通 FPGA 内部的所有单元，而连线的长度和工艺决定着信号在连线上的驱动能力和传输速度。FPGA 芯片内部有着丰富的布线资源，这些布线资源根据工艺、长度、宽度和分布位置的不同而划分为 4 类不同的类别：

第一类是全局布线资源，用于芯片内部全局时钟和全局复位/置位的布线；

第二类是长线资源，用以完成芯片 Bank 间的高速信号和第二全局时钟信号的布线；

第三类是短线资源，用于完成基本逻辑单元之间的逻辑互连和布线；

第四类是分布式的布线资源，用于专有时钟、复位等控制信号线。

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

5) 底层嵌入功能单元

底层嵌入功能单元的概念比较笼统，这里我们指的是那些通用程度较高的嵌入式功能模块，比如 PLL

（Phase Locked Loop）、DLL（Delay Locked Loop）、DSP、CPU 等。随着 FPGA 的发展，这些模块被越来越多地嵌入到 FPGA 的内部，以满足不同场合的需求。

目前大多数 FPGA 厂商都在 FPGA 内部集成了 DLL 或者 PLL 硬件电路，用以完成时钟的高精度、低抖动的倍频、分频、占空比调整、相移等功能。目前，高端 FPGA 产品集成的 DLL 和 PLL 资源越来越丰富，功能越来越复杂，精度越来越高。

另外，越来越多的高端 FPGA 产品将包含 DSP 或 CPU 等硬核，从而 FPGA 将由传统的硬件设计手段逐步过渡到系统级设计平台。例如 Altera 的 Stratix IV、Stratix V 等器件内部集成了 DSP 核；Xilinx 的 Virtes II 和 Virtex II pro 系列 FPGA 内部集成了 Power PC450 的处理器。FPGA 内部嵌入 DSP 或 CPU 等处理器，使 FPGA 在一定程度上具备了实现软硬件联合系统的能力，FPGA 正逐步成为 SOPC 的高效设计平台。

6) 内嵌专用硬核

这里的内嵌专用硬核与前面的底层嵌入单元是有区分的，这里讲的内嵌专用硬核主要指那些通用性相对较弱，不是所有 FPGA 器件都包含硬核。

在 ZYNQ 的 PL 端有一个数模混合模块——XADC，它就是一个硬核。XADC 包含两个模数转换器（ADC），一个模拟多路复用器，片上温度和片上电压传感器等。我们可以利用这个模块监测芯片温度和供电电压，也可以用来测量外部的模拟电压信号。

7) ZYNQ PL 架构

在介绍完 FPGA 的基本结构之后，我们给出 ZYNQ PL 架构的示意图，如下所示：

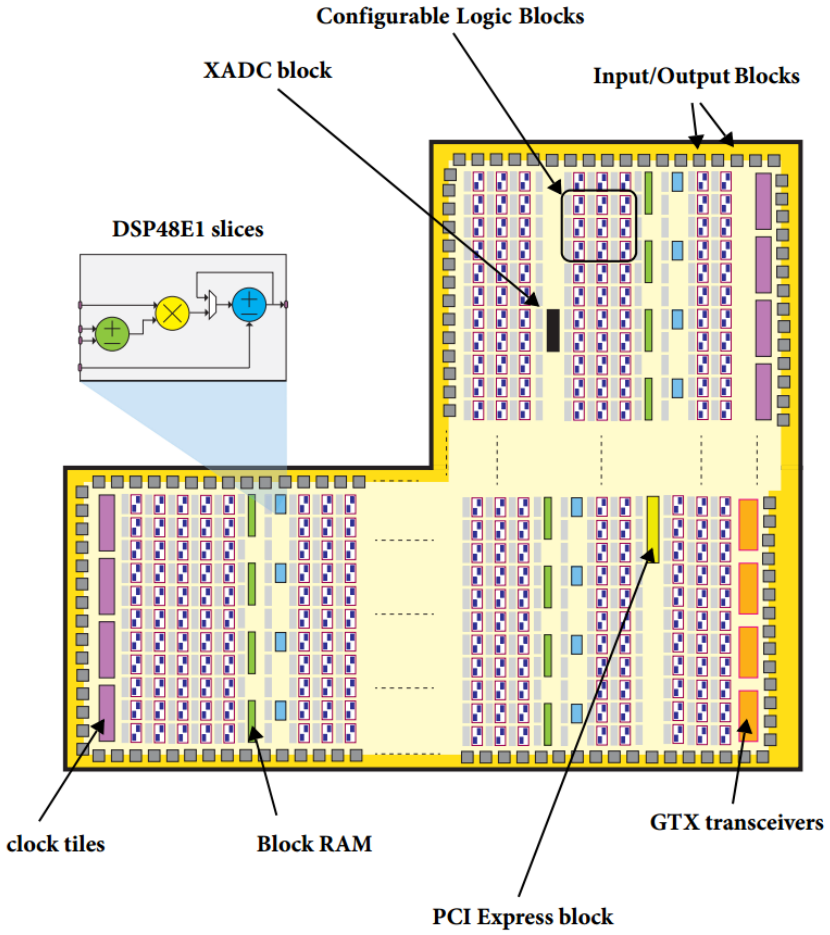


图 1.3.6 ZYNQ PL 架构示意图

1.4 ZYNQ PS 简介

虽然我们在前面花费了大量的篇幅来介绍 ZYNQ 的 PL 部分，但是 ZYNQ 实际上是一个以处理器为核心的系统，PL 只是它的一个外设。Zynq-7000 系列的亮点在于它包含了完整的 ARM 处理器系统，且处理器系统中集成了内存控制器和大量的外设，使 Cortex-A9 处理器可以完全独立于可编程逻辑单元。而且实际上在 ZYNQ 中，PL 和 PS 两部分的供电电路是独立的，这样 PS 或 PL 部分不被使用的话就可以被断电。

在前面我们介绍 SOPC 时提到过，FPGA 可以用来搭建嵌入式处理器，像 Xilinx 的 MicroBlaze 处理器或者 Altera 的 Nios II 处理器。像这种使用 FPGA 的可编程逻辑资源搭建的处理器我们称之为“软核”处理器，它的优势在于处理器的数量以及实现方式的灵活性。

而 ZYNQ 中集成的是一颗“硬核”处理器，它是硅芯片上专用且经过优化的硬件电路，硬核处理器的优势是它可以获得相对较高的性能。另外，ZYNQ 中的硬件处理器和软核处理器并不冲突，我们完全可以使用 PL 的逻辑资源搭建一个 Microblaze 软核处理器，来和 ARM 硬核处理器协同工作。

需要注意的是，Zynq 处理器系统里并非只有 ARM 处理器，还有一组相关的处理资源，形成了一个应用处理器单元（Application Processing Unit，APU），另外还有扩展外设接口、cache 存储器、存储器接口、互联接口和时钟发生电路等。

ZYNQ 处理器系统（PS）示意图如下所示，其中红色高亮区域为 APU。

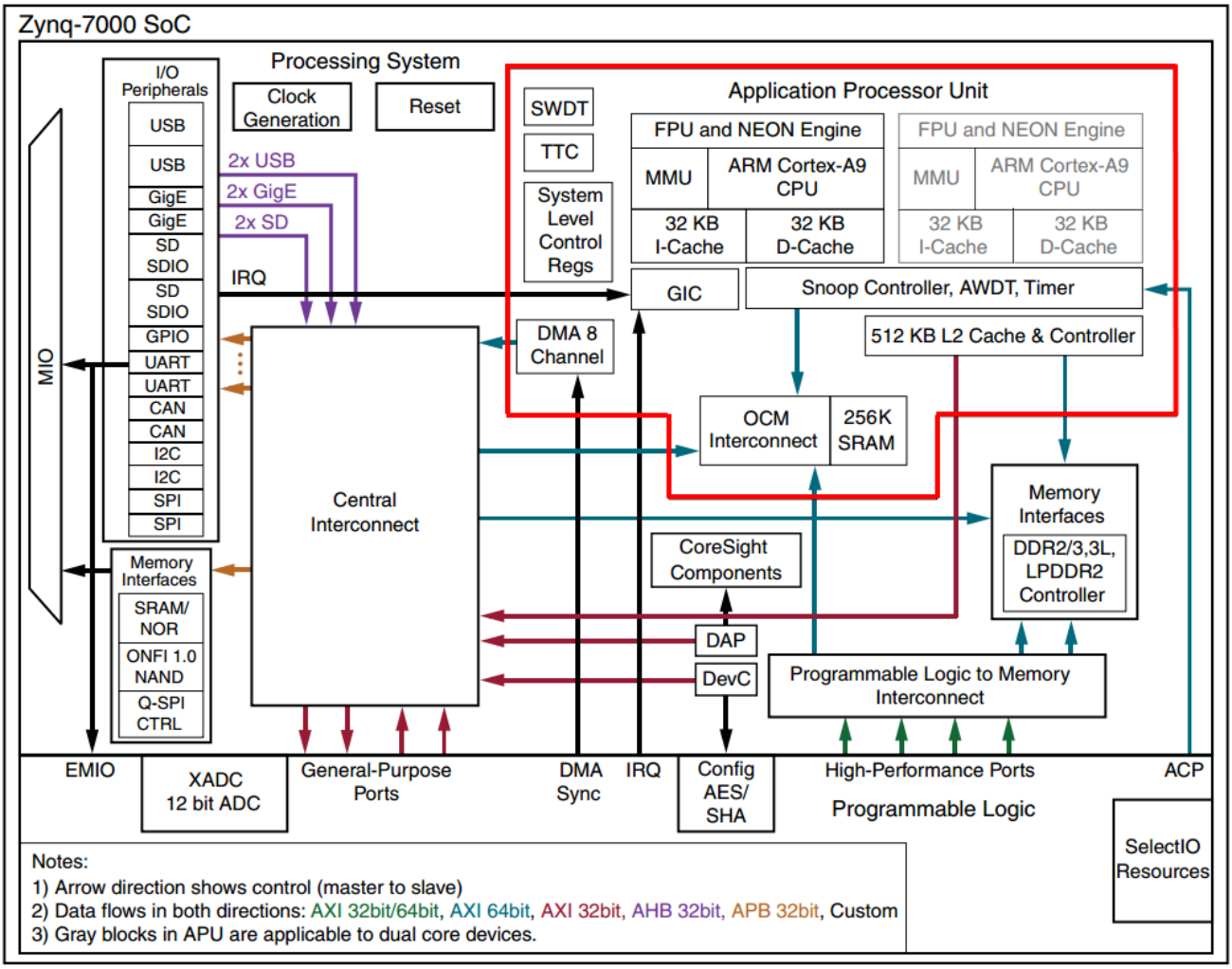


图 1.4.1 PS 系统示意图

1) APU

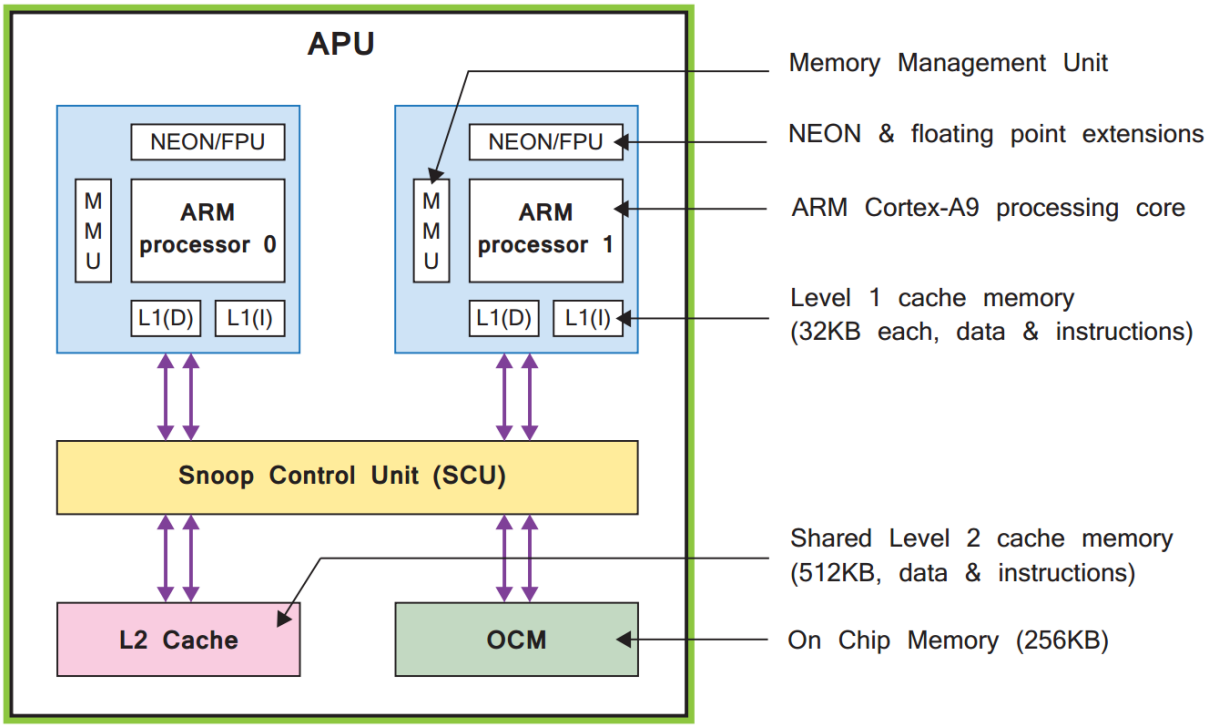


图 1.4.2 APU 简化示意图

如上图所示是 APU 的简化框图。APU 主要是由两个 ARM 处理器核组成的，每个都关联了一些可计算的单元：一个 NEONTM 媒体处理引擎（Media Processing Engine，MPE）和浮点单元（Floating Point Unit，FPU）；一个内存管理单元（Memory Management Unit，MMU）；和一个一级 cache 存储器（分为指令和数据两个部分）。APU 里还有一个二级 cache 存储器，再往下还有片上存储器（On Chip Memory，OCM），这些是两个 ARM 处理器共用的。最后，由一个一致性控制单元（Snoop Control Unit，SCU）在 ARM 核和二级 cache 及 OCM 存储器之间形成了桥连接。SCU 还部分负责与 PL 对接，图中没有标出这个接口。

2) 外部接口

如图 1.4.1 所示，Zynq PS 实现了众多接口，既有 PS 和 PL 之间的，也有 PS 和外部部件之间的。PS 和外部接口之间的通信主要是通过复用的输入/输出（Multiplexed Input/Output，MIO）实现的，它提供了可以灵活配置的 54 个引脚，这表明外部设备和引脚之间的映射是可以按需定义的。当需要扩展超过 54 个引脚的时候可以通过扩展 MIO（Extended MIO，EMIO）来实现，EMIO 并不是 PS 和外部连接之间的直接通路，而是通过共用了 PL 的 I/O 资源来实现的。

PS 中可用的 I/O 包括标准通信接口和通用输入/输出（General Purpose Input/Output，GPIO），GPIO 可以用做各种用途，包括简单的按钮、开关和 LED。如下图所示：

I/O 接口	说明
SPI (x2)	串行外设接口 (Serial Peripheral Interface) *基于 4 引脚接口的串行通信的事实标准，可以用于主机或从机模式。*
I2C (x2)	I^2C 总线 *与 I^2C 总线标准第二版兼容。支持主机和从机模式。*
CAN (x2)	控制器区域网络 (Controller Area Network) *兼容 ISO 118980-1，CAN 2.0A 和 CAN 2.0B 标准的接口控制器。*
UART (x2)	通用异步收发器 (Universal Asynchronous Receiver Transmitter) *用于串行通信的低速数据调制解调器接口。常用于和主机 PC 的终端连接。*
GPIO	通用输入/输出 (General Purpose Input/Output) *有 4 组 GPIO，每组 32 位。*
SD (x2)	*用于和 SD 卡存储器对接。*
USB (x2)	通用串行总线 (Universal Serial Bus) *兼容 USB 2.0，可以做主机、设备或灵活配置 (“on-the-go”也就是 OTG 模式，意思是它可以在主机和设备模式之间做切换)。*
GigE (x2)	以太网 *以太网 MAC 外设，支持 10Mbps、100Mbps 和 1Gbps 模式。*

图 1.4.3 PS 的外部接口

3) 存储器接口

Zynq-7000 APSoC 上的存储器接口单元包括一个动态存储器控制器和几个静态存储器接口模块。动态存储器控制器可以用于 DDR3、DDR3L、DDR2 或 LPDDR2。静态存储器控制器支持一个 NAND 闪存接口、一个 Quad-SPI 闪存接口、一个并行数据总线和并行 NOR 闪存接口。

4) 片上存储器

片上存储器包括 256kB 的 RAM（OCM）和 128kB 的 ROM（BootROM）。OCM 支持两个 64 位 AXI 从机接口端口，一个端口专用于通过 APU SCU 的 CPU/ACP 访问，而另一个是由 PS 和 PL 内其他所有的总线主机所共享的。BootROM 是 ZYNQ 芯片上的一块非易失性存储器，它包含了 ZYNQ 所支持的配置器件的驱动。BootROM 对于用户是不可见的，专门保留且只用于引导的过程。

5) AXI 接口

ZYNQ 将高性能 ARM Cotex-A 系列处理器与高性能 FPGA 在单芯片内紧密结合，为设计带来了如减小体积和功耗、降低设计风险，增加设计灵活性等诸多优点。在将不同工艺特征的处理器与 FPGA 融合在一个芯片上之后，片内处理器与 FPGA 之间的互联通路就成了 ZYNQ 芯片设计的重中之重。如果 Cotex-A9 与 FPGA 之间的数据交互成为瓶颈，那么处理器与 FPGA 结合的性能优势就不能发挥出来。

Xilinx 从 Spartan-6 和 Virtex-6 系列开始使用 AXI 协议来连接 IP 核。在 7 系列和 ZYNQ-7000 AP SoC 器件中，Xilinx 在 IP 核中继续使用 AXI 协议。AXI 的英文全称是 Advanced eXtensible Interface，即高级可扩展接口，它是 ARM 公司所提出的 AMBA（Advanced Microcontroller Bus Architecture）协议的一部分。

AXI 协议是一种高性能、高带宽、低延迟的片内总线，具有如下特点：

1、总线的地址/控制和数据通道是分离的；

2、支持不对齐的数据传输；

3、支持突发传输，突发传输过程中只需要首地址；

4、具有分离的读/写数据通道；

5、支持显著传输访问和乱序访问；

6、更加容易进行时序收敛。

在数字电路中只能传输二进制数 0 和 1，因此可能需要一组信号才能高效地传输信息，这一组信号就组成了接口。AXI4 协议支持以下三种类型的接口：

1、 AXI4：高性能存储映射接口。

2、 AXI4-Lite：简化版的 AXI4 接口，用于较少数据量的存储映射通信。

3、 AXI4-Stream：用于高速数据流传输，非存储映射接口。

在这里首先我们首先解释一下存储映射（Meamory Map）这一概念。如果一个协议是存储映射的，那么主机所发出的会话（无论读或写）就会标明一个地址。这个地址对应于系统存储空间中的一个地址，表明是针对该存储空间的读写操作。

AXI4 协议支持突发传输，主要用于处理器访问存储器等需要指定地址的高速数据传输场景。AXI-Lite 为外设提供单个数据传输，主要用于访问一些低速外设中的寄存器。而 AXI-Stream 接口则像 FIFO 一样，数据传输时不需要地址，在主从设备之间直接连续读写数据，主要用于如视频、高速 AD、PCIe、DMA 接口等需要高速数据传输的场合。

在 PS 和 PL 之间的主要连接是通过一组 9 个 AXI 接口，每个接口有多个通道组成。这些形成了 PS 内部的互联以及与 PL 的连接，如下图所示：

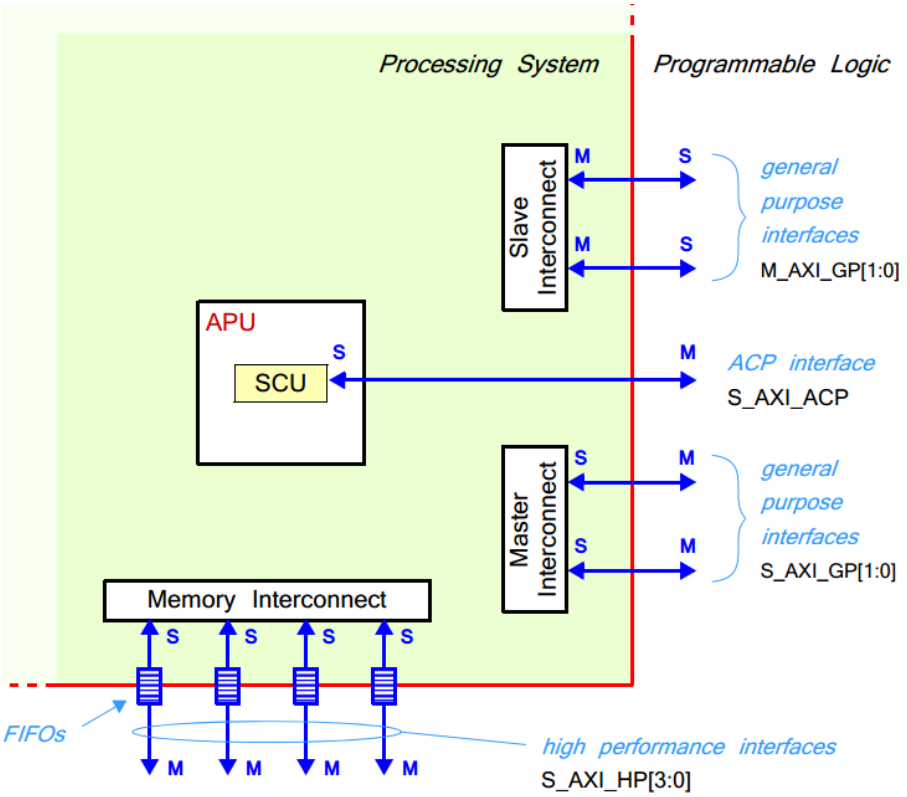


图 1.4.4 PS 与 PL 的 AXI 互联

我们将上图中的接口总结如下所示：

Interface Name	Interface Description	Master	Slave
M_AXI_GP0	General Purpose (AXI_GP)	PS	PL
M_AXI_GP1		PS	PL
S_AXI_GP0	General Purpose (AXI_GP)	PL	PS
S_AXI_GP1		PL	PS
S_AXI_ACP	Accelerator Coherency Port (ACP), cache coherent transaction	PL	PS
S_AXI_HP0	High Performance Ports (AXI_HP) with read/write FIFOs. (Note that AXI_HP interfaces are sometimes referred to as AXI Fifo Interfaces, or AFIs).	PL	PS
S_AXI_HP1		PL	PS
S_AXI_HP2		PL	PS
S_AXI_HP3		PL	PS

图 1.4.5 PS 与 PL 的 AXI 接口

上图给出了每个接口的简述，标出了主机和从机（按照惯例，主机是控制总线并发起会话的，而从机是做响应的）。需要注意的是，接口命名的第一个字母表示的是 PS 的角色，也就是说，第一个字母 “M” 表示 PS 是主机，而第一个字母“S”表示 PS 是从机。

上图中 PS 和 PL 之间的 9 个 AXI 接口可以分成三种类型：

1、**通用 AXI**（General Purpose AXI）：一条 32 位数据总线，适合 PL 和 PS 之间的中低速通信。接口是透传的不带缓冲。总共有四个通用接口：两个 PS 做主机，另两个 PL 做主机。

2、**加速器一致性端口**（Accelerator Coherency Port）：在 PL 和 APU 内的 SCU 之间的单个异步连接，总线宽度为 64 位。这个端口用来实现 APU cache 和 PL 的单元之间的一致性。PL 是做主机的。

3、**高性能端口**（High Performance Ports）：四个高性能 AXI 接口，带有 FIFO 缓冲来提供“批量”读写操作，并支持 PL 和 PS 中的存储器单元的高速率通信。数据宽度是 32 或 64 位，在所有四个接口中 PL 都是做主机的。

上面的每条总线都是由一组信号组成的，这些总线上的会话是根据 AXI4 总线协议进行通信的。有关 AXI4 协议更详细的内容，我们将在后续的章节进行介绍。

第二章　实验平台简介

本章内容主要向大家简要介绍我们的实验平台：启明星 ZYNQ 开发板。通过本章的学习，你将对我们后面使用的实验平台有个快速的了解，为后面的学习做铺垫。

本章包括以下几个部分：

2.1　启明星 ZYNQ 开发板资源初探

2.2　启明星 ZYNQ 开发板资源说明

2.1 启明星 ZYNQ 开发板资源初探

正点原子目前已经拥有多款 STM32、I.MXRT 以及 FPGA 开发板，这些开发板常年稳居淘宝销量冠军，累计出货超过 10W 套。这款 ZYNQ 开发板，是正点原子针对中高端应用场景所推出的开发板，采用核心板+底板的设计，方便用户对核心板进行二次开发。

2.1.1 启明星开发板底板资源

首先我们来看启明星 ZYNQ 开发板的底板资源图，如图 2.1.1 所示。

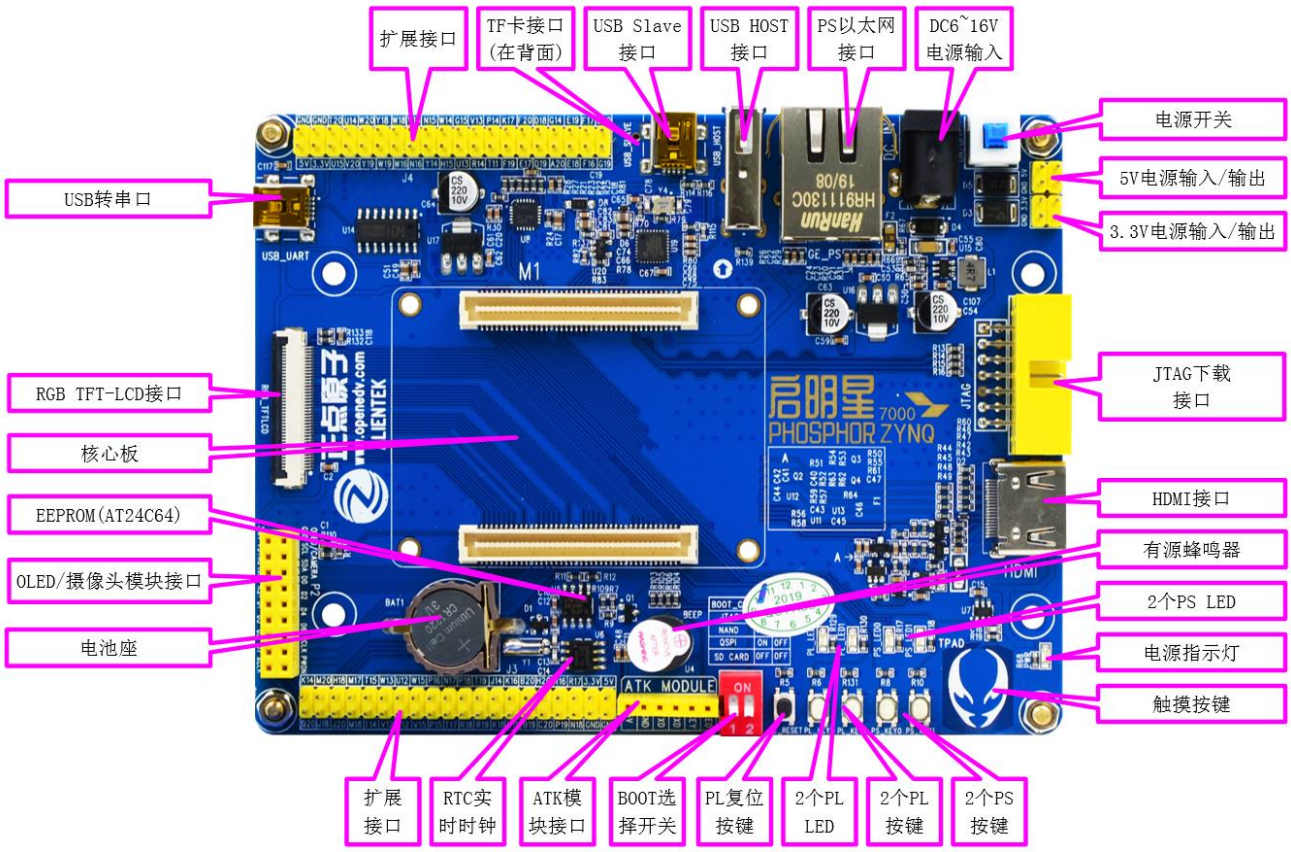


图 2.1.1 启明星 ZYNQ 开发板底板资源图

从图 2.1.1 可以看出，启明星 ZYNQ 开发板底板资源非常丰富，把 ZYNQ 芯片内部的资源发挥到了极致，同时扩充了丰富的接口和功能模块，整个开发板显得十分高端和大气。

开发板的外形尺寸为 100mm*130mm，板子的设计充分考虑了人性化设计，并结合正点原子多年的开发板设计经验，经过多次改进，最终确定了这样的设计。

正点原子启明星开发板底板资源如下：

- 1 个核心板接口，支持 ZYNQ7020/ZYNQ-7010 核心板
- EEPROM 芯片：AT24C64，容量：64Kbit（8K 字节）
- 1 个电源指示灯（蓝色）
- 2 个 PL LED（红色）
- 2 个 PS LED（红色）
- 1 个有源蜂鸣器

- 1 个 PL 复位按键
- 2 个 PL 功能按键
- 2 个 PS 功能按键
- 1 个电容触摸按键
- 1 个标准的 RGB888 TFT-LCD 接口
- 1 个 OLED/摄像头模块接口
- 2 个 20x2 扩展口，共 72 个扩展 IO 口
- 1 个 RTC 实时时钟，芯片型号为 PCF8563
- 1 个 RTC 电池座，并带电池
- 1 个 ATK MODULE 接口，支持正点原子蓝牙/GPS/UART 等模块
- 1 个 BOOT 模式选择开关
- 1 个电源指示灯
- 1 路 HDMI 输入/输出接口
- 1 路 USB HOST 接口
- 1 路 USB SLAVE 接口
- 14-Pin JTAG 接口，提供开发板下载和调试的功能
- 1 组 5V 电源供应/接入口
- 1 组 3.3V 电源供应/接入口
- 2 路 3.3V 电源口和 2 路 5V 电源口，方便给外扩模块供电
- 1 个电源开关，控制整个开发板的电源
- 1 个直流电源输入接口（输入电压范围：DC6~16V）
- 1 个 PS 端千兆以太网接口（RJ45）
- 1 个 Micro SD（TF）卡接口（位于开发板背面）
- 1 个 USB 串口

启明星 ZYNQ 开发板底板的特点包括：

1) 接口丰富。板子提供了丰富的标准外设接口，可以方便的进行各种外设的实验和开发。

2) 设计灵活。我们采用核心板+底板的形式，板上很多资源都可以灵活配置，以满足不同条件下的使用；板上很多资源都可以灵活配置，以满足不同条件下的使用。其中芯片引出两排 20x2 扩展口，共 72 个扩展 IO 口。

3) 资源充足。板载 HDMI 接口、RGB LCD 接口、1 个 PS 千兆网口、1 个 USB HOST 接口+1 个 USB SLAVE 接口以及各种接口芯片，满足各种应用需求。

4) 人性化设计，各个接口都有丝印标注，且用方框框出，使用起来一目了然；部分常

用外设大丝印标出，方便查找；接口位置设计合理，方便顺手。资源搭配合理，物尽其用。

2.1.2 启明星开发板核心板资源

接下来我们来看启明星 ZYNQ 核心板资源图，正点原子的 ZYNQ 核心板根据主控芯片和 DDR3 内存容量的不同分为 ZYNQ-7020 核心板和 ZYNQ-7010 核心板。启明星 ZYNQ-7020 核心板和启明星 ZYNQ-7010 核心板的资源图分别如图 2.1.2 和图 2.1.3 所示：

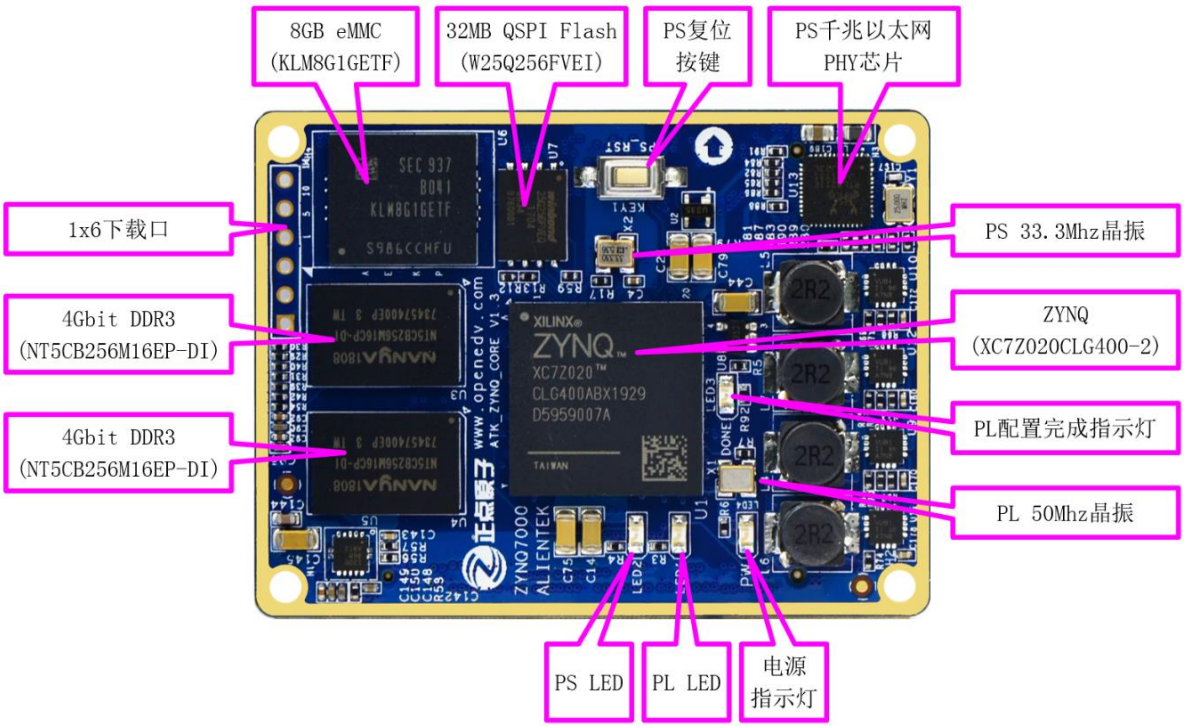


图 2.1.2 ZYNQ-7020 核心板资源图

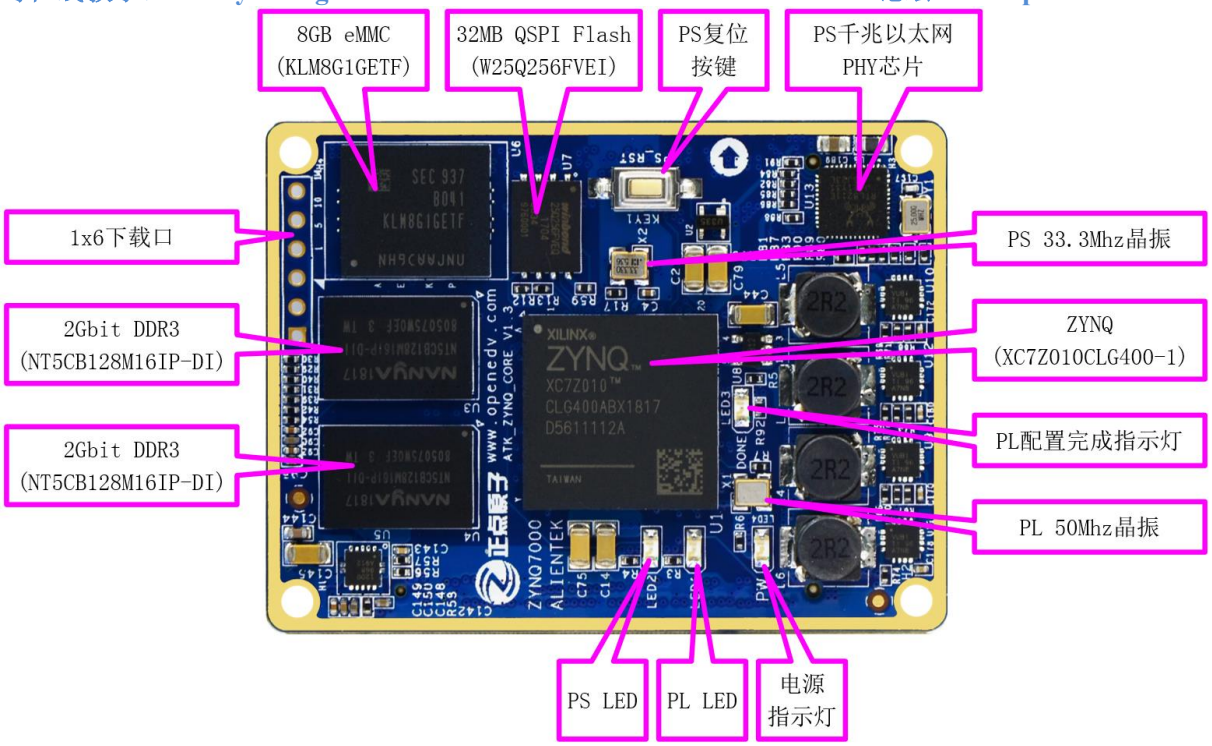


图 2.1.3 ZYNQ-7010 核心板资源图

由图 2.1.2 和图 2.1.3 可知，ZYNQ 核心板板载资源丰富，可以满足各种应用的需求。整个核心板的外形尺寸为 45mm*60mm，非常小巧，并且采用了贴片板对板连接器，使得其可以很方便的应用在各种项目上。

ZYNQ-7020 核心板和 ZYNQ-7010 核心板除 ZYNQ 主控芯片和 DDR3 存储容量不同外，其余外设完全相同。ZYNQ-7020 核心板和 ZYNQ-7010 核心板具体资源如下：

◆ ZYNQ 芯片采用 Xilinx 公司的 ZYNQ7000 系列芯片，ZYNQ-7020 核心板的 ZYNQ 芯片型号为 XC7Z020CLG400-2，PL 逻辑单元多达 85K，BRAM 存储资源为 4.9Mbit；ZYNQ-7010 核心板的 ZYNQ 芯片型号为 XC7Z010CLG400-1，PL 逻辑单元多达 28K，BRAM 存储资源为 2.1Mbit；两种 ZYNQ 芯片的处理器系统均为双核 Cortex-A9 结构。需要注意的是，ZYNQ-7020 核心板主控芯片的速度等级为“-2”，ZYNQ-7010 核心板主控芯片的速度等级为“-1”，因此 ZYNQ-7020 核心板主控芯片的速度等级更高，所支持的频率也更高

◆ 2 片 DDR3 SDRAM，ZYNQ-7020 核心板的 DDR3 型号为 NT5CB256M16EP-DI，每片 4Gbit，总容量为 8Gbit（1GB）；ZYNQ-7010 核心板的 DDR3 型号为 NT5CB128M16IP-DI，每片 2Gbit，总容量为 4Gbit（512MB）

◆ 1 个 1x6 下载接口，与底板的 14-Pin JTAG 接口功能一致

◆ 1 个 PL LED

◆ 1 个 PS LED

◆ 1 个 PL 晶振：50Mhz，给 PL 可编程逻辑提供时钟

◆ 1 个 PS 晶振：33.333Mhzz，给 PS CPU 逻辑提供时钟

◆ 1 个电源指示灯

- 1 个 PL 配置状态指示灯（DONE LED）
- 1 个 PS 端千兆以太网 PHY 芯片：RTL8211E-VL
- 1 个 PS 复位按键
- 1 个 QSPI FLASH，型号 W25Q256FVEI，容量：256Mbit（32MB）
- 1 个 eMMC，型号为 KLM8G1GETF，容量：8GB。

启明星 ZYNQ 开发板底板的特点包括：

1) 体积小巧。核心板仅 45mm*60mm 大小，方便使用到各种项目里面。

2) 集成方便。核心板使用两个 2*40P BTB 连接座，可以非常方便的集成到客户 PCB 上，更换简单，方便维修测试。

3) 资源丰富。核心板板载 1GB/512MB DDR3、32MB QSPI Flash、8GB eMMC 存储器，可以满足各种应用场合

4) 性能稳定。核心板采用 10 层板设计，单独地层、电源层，保证运行稳定、可靠。

5) 人性化设计。核心板各种放有详细丝印，使用起来一目了然。资源搭配合理，物尽其用。

6) 接口丰富。板子提供了丰富的标准外设接口，可以方便的进行各种外设的实验和开发。

2.2 启明星 ZYNQ 开发板资源说明

启明星 ZYNQ 开发板资源说明分为两个部分：硬件资源说明和软件资源说明。

2.2.1 硬件资源说明

启明星 ZYNQ 开发板分别为 ZYNQ 的 PL 端和 PS 端配备了丰富的硬件外设，如下图所示：

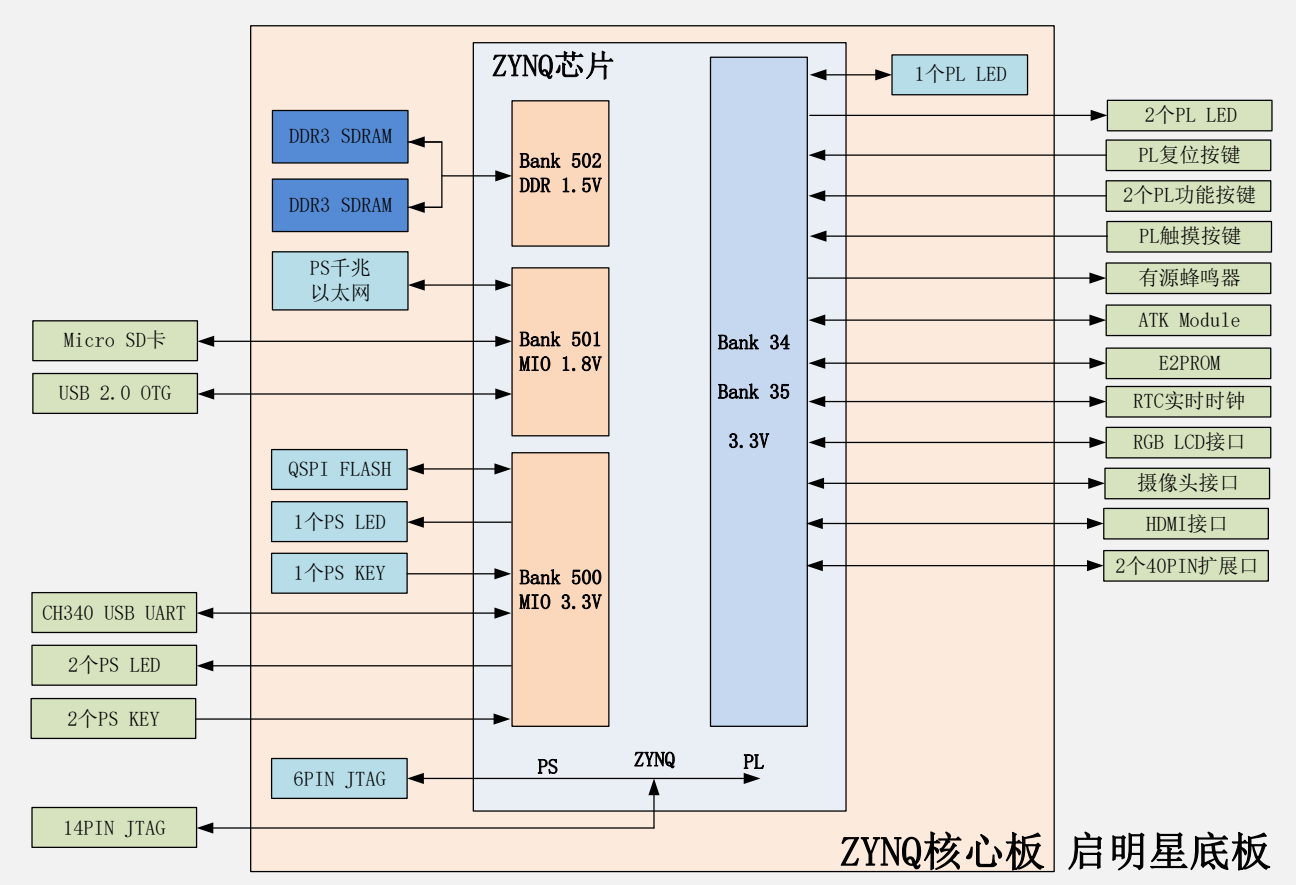


图 2.2.1 启明星 ZYNQ 开发板硬件资源框图

XC7Z020 和 XC7Z010 芯片的 PS 端 IO bank 包括 Bank 502、Bank 501 和 Bank 500，PL 端的 IO bank 包括 Bank 13、Bank 34 和 Bank 35，我们分别为 PS 端和 PL 端的 IO 连接了不同的外设，如图 2.2.1 所示。

下面我们分别介绍启明星 ZYNQ 核心板和底板上的硬件资源。

底板外设简介：

1. 有源蜂鸣器

启明星底板板载一个有源蜂鸣器（BEEP），可以实现简单的报警/闹铃。

2. 1 个 PL 复位按键

启明星底板板载一个 PL 端的复位按键（PL_RESET），可以作为 ZYNQ PL 端逻辑的复位信号，按键复位信号默认是高电平的，当复位按键按下之后为低电平。

3. 2 个 PL LED

启明星底板板载 2 个 PL LED 灯（PL_LED0~PL_LED1），连接到了 PL 端的 IO 口上。在调试代码的时候，使用 LED 来指示程序执行状态，是非常不错的一个辅助调试方法。

4. 2 个 PL 功能按键

启明星底板板载 2 个机械式按键（PL_KEY0~PL_KEY1）是直接连接在 PL 端的 IO 口上的，可以作为人机交互的输入信号。这 2 个按键信号默认都是高电平的，当按键被按下之后，按键信号变为低电平。

5. 2 个 PS LED

启明星底板板载 2 个 PS LED 灯（PS_LED0~PS_LED1），LED 连接到了 PS 端的 IO 口上。

6. 2 个 PS 功能按键

启明星底板板载 2 个机械式按键（PS_KEY0、PS_KEY1），按键都是直接连接在 PS 端的 IO 口上的，可以作为人机交互的输入信号。这 2 个按键信号默认都是高电平的，当按键被按下之后，按键信号变为低电平。

7. 1 个触摸按钮

启明星底板板载 1 个电容触摸输入按键（TPAD），触摸方案采用 AR101 芯片，该芯片利用电容充放电原理，实现触摸按键检测的功能。当手指触摸 TPAD 按键时，触摸芯片输出高电平，松开为低电平。

8. USB 转串口

启明星底板板载一个 PS 端的 USB 转串口，之所以设计成 USB 形式的串口，是出于现在电脑上串口正在消失，尤其是笔记本，几乎清一色的没有串口。所以板载了 USB 串口可以方便大家进行 USB 串口通信的试验。同时这个 USB 接口还可以给开发板提供电源，但是其最大电流只有 500mA，但是在运行 PS 端的 ARM 处理器时它却不能够提供足够大的电流，所以还是建议大家使用专门的电源适配器来为开发板供电。

9. RGB TFT-LCD 接口

启明星底板板载一个 RGB LCD 接口，可以连接各种分辨率的正点原子 RGB LCD 屏，采用的是 RGB888 格式，可显示 1677 万色，色彩显示丰富。并且支持触摸的功能。

10. OLED/摄像头模块接口

启明星底板板载一个 OLED/摄像头模块接口（P2）。如果是正点原子的摄像头模块，则刚好可以直接插上去。通过这个接口，可以分别完成 OLED 显示或者摄像头显示实验。

11. EEPROM (AT24C64)

启明星底板板载一个 I2C 接口的 EEPROM 芯片，容量为 64Kbit，也就是 8K 字节。用于存储一些掉电不能丢失的重要数据，比如系统设置的一些参数等。有了这个就可以方便的实现掉电数据保存。

12. RTC 实时时钟

启明星底板板载一个 RTC 实时时钟芯片（U6），芯片型号为 PCF8563。PCF8563 是 PHILIPS 公司推出的一款工业级多功能时钟/日历芯片，具有报警功能、定时器功能、时钟输出功能以及中断输出功能，能完成各种复杂的定时服务。

13. 电池接口

启明星底板板载一个 RTC 实时时钟的供电接口（BAT1），可以保证在开发板断电时，实时时钟仍然能够继续工作，这样的话，配置的日期与时间不会因开发板的断电而恢复到默认值。

14. ATK MODULE 接口

启明星底板板载一个 ALIENTEK 通用模块接口（U4），目前可以支持 ALIENTEK 开发的 GPS 模块、

蓝牙模块、MPU6050 模块和全彩 RGB 灯模块等，直接插上对应的模块，就可以进行相关模块的开发。后续我们将开发更多兼容该接口的其他模块，实现更强大的扩展性能。

15. BOOT 模式选择开关

启明星底板板载一个 ZYNQ 的 BOOT 模式选择开关（BOOT_CFG），用于设置 PS 端在上电后的启动源，包括 JTAG、NAND、QSPI FLASH 和 SD Card。

16. 电源指示灯

启明星底板板载 1 颗蓝色的 LED 灯（PWR），用于指示电源状态。在电源开启的时候电源指示灯会处于点亮的状态，否则为熄灭的状态。通过这个 LED，可以判断开发板的上电情况。

17. HDMI 接口

启明星底板板载 1 个 HDMI（High Definition Multimedia Interface，HDMI）接口，该接口可以连接到 HDMI 显示器上，从而显示出视频或者图片等。开发板没有板载 HDMI 的 PHY 芯片，HDMI 的 PHY 编解码功能由 ZYNQ PL 逻辑来模拟实现。

18. 14-Pin JTAG 接口

启明星底板板载 1 个 14 针标准 JTAG 调试口（JTAG），该 JTAG 口与核心板的 6-Pin JTAG 接口在硬件上是连在一起的，可以直接和 FPGA 下载器（调试器）连接，用于下载程序或者对程序进行在线调试。

19. 3.3V 电源输入/输出

启明星底板板载 1 组 3.3V 电源输入输出排针（2*3），用于给外部提供 3.3V 的电源，也可以从外部接 3.3V 的电源给板子供电。大家在做实验的时候可能经常会为没有 3.3V 电源而苦恼不已，有了启明星 ZYNQ 开发板，你就可以很方便的拥有一个简单的 3.3V 电源（最大电流不能超过 500mA）。

20. 5V 电源输入/输出

启明星底板板载 1 组 5V 电源输入输出排针（2*3），该排针用于给外部提供 5V 的电源，也可以从外部接 5V 的电源给板子供电。同样大家在实验的时候可能经常会为没有 5V 电源而苦恼不已，正点原子充分考虑到了大家的需求，有了这组 5V 排针，你就可以很方便的拥有一个简单的 5V 电源（USB 供电的时候，最大电流不能超过 500mA，外部供电的时候，最大可达 1000mA）。

21. 电源开关

启明星底板板载 1 个电源开关（K1）。该开关用于控制整个开发板的供电，如果通过开关切断电源，则整个开发板都将断电，电源指示灯（PWR）会随着此开关的状态而亮灭。

22. DC6~16V 电源输入

启明星底板板载 1 个外部电源输入口（DC_IN），采用标准的直流电源插座。开发板板载了 DC-DC 芯片（JW5060T），用于给开发板提供高效、稳定的 5V 电源。由于采用了 DC-DC 芯片，所以开发板的供电范围十分宽，大家可以很方便的找到合适的电源（只要输出范围在 DC6~16V 的基本都可以）来给开发板供电。在耗电比较大的情况下，比如用到 4.3 寸屏/7 寸屏/网口/高速 AD-DA 的时候，建议大家使用外部电源供电，可以提供足够的电流给开发板使用。

23. PS 端千兆以太网接口(RJ45)

启明星底板板载 1 个千兆的 RJ45 网口（PS_GE），通过转接板连接到了核心板上的以太网 PHY 芯片，支持 10Mbps/100Mbps/1000Mbps 的通信速率，板载的 RJ45 插座可以连接网线，实现网络通信功能。

24. USB2.0 接口

启明星开发板为 PS 端的 USB 模块配备了一个 USB2.0 PHY 芯片，并且根据主从模式的不同，外接了 1 个 HOST 模式的接口和一个 SLAVE 模式的接口。1 个 HOST 模式的接口可分别用来连接不同的 USB SLAVE 设备，可以用来连接鼠标、键盘等不同的设备，以满足各种不同的需求。一个 SLAVE 模式的接口可以用来连接 HOST 设备，以满足特定的应用需求。

25. Micro SD 接口

启明星底板板载 1 个标准 Micro SD 接口（TF_CARD），该接口位于开发板的背面，采用 Micro SD 接口，可以使用 SPI/SDIO 驱动方式。有了这个接口，就可以满足海量数据存储的需求。

26. 40PIN 扩展口

启明星底板板载两个个 40PIN 的扩展口，位于开发板的左下角，扩展口规格是 2*20 Pin，采用标准 2.54MM 排针间距，其中每个扩展口包括 36 个 IO 口，1 个+3.3V，1 个+5V，2 个 GND。它可以用来连接不同的功能模块，例如，正点原子开发的高速 AD/DA 模块和双目摄像头模块等。

核心板外设简介：

1. ZYNQ 主控芯片

ZYNQ-7020 核心板主控芯片为 XC7Z020CLG400-2，85K LC（逻辑单元），4.9Mbit BRAM；
ZYNQ-7010 核心板主控芯片为 XC7Z010CLG400-1，28K LC（逻辑单元），2.1Mbit BRAM。
Zynq 处理器系统里包含两个 Cortex-A9 处理器，除此之外，还有一组相关的处理资源，形成了一个应用处理器单元（Application Processing Unit，APU）。另外处理器系统里还有扩展外设接口、cache 存储器、存储器接口、互联接口和时钟发生电路等。

2. DDR3 SDRAM

ZYNQ-7020 核心板板载两片 4Gbit DDR3 内存，芯片型号为 NT5CB256M16EP-DI，总容量为 8Gbit（1GB）；
ZYNQ-7010 核心板板载两片 2Gbit DDR3 内存，芯片型号为 NT5CB128M16IP-DI，总容量为 4Gbit（512MB）；
板载的 DDR3 芯片可以轻松应对各种大内存、高带宽场景需求，比如摄像头图像数据存储等。此外，DDR3 内存也作为 PS 端处理器的运行内存。

3. 1 个 6-Pin 下载接口

核心板的 6-PIN 下载接口与底板的 14-Pin JTAG 接口是一体的，可以在单独使用核心板时做调试和下载使用。

4. 1 个 PL LED

它连接到了 PL 端的用户 IO 上，可由用户编写 Verilog 代码来控制其亮灭。

5. 1 个 PS LED

它连接到了 PS 端的 GPIO 上，可由用户使用 C 代码编程来控制其亮灭。

6. PL 端 50Mhz 晶振

这是开发板上用于为 ZYNQ PL 端提供时钟的晶振（XTAL）。该晶振输出的时钟是 PL 端最原始的时钟，其它外设需要的各种频率的时钟都在此基础上进行倍频或分频。

7. PS 端 33.333Mhz 晶振

这是开发板上用于为 ZYNQ PL 端提供时钟的晶振（XTAL）。该晶振输出的时钟是 PS 端最原始的时钟，PS 端外设所需要的各种频率的时钟都在此基础上进行倍频或分频。

8. 电源指示灯（PW）

这是核心板板载的一颗蓝色的 LED 灯（PWR），用于指示电源状态。在电源开启的时候电源指示灯会处于点亮的状态，否则为熄灭的状态。通过这个 LED，可以判断开发板的上电情况。需要说明的是，ZYNQ 核心板没有独立的供电接口，需要通过 BTB 转接板连接底板，由底板上的转接板进行供电。

9. PL 配置状态指示灯（DONE LED）

这是核心板板载的一颗 PL 配置状态指示灯，连接到了 PL 端的配置完成（DONE）信号，在 PL 端配置（下载程序）完成之后，该 LED 灯会被点亮。

10. PS 千兆以太网 PHY 芯片

这是核心板板载的一颗 PS 端千兆以太网 PHY（物理）芯片，型号为 RTL8211E-VL，实现了 10/100/1000M 以太网物理层功能。该 PHY 芯片的引脚连接到了底板上的 RJ45 接口上，能够满足高带宽通信的需求。

11. PS 端复位按键（PS_RST）

ZYNQ PS 端的复位按键，它连接到了 PS 端的复位逻辑，按下后，PS 端将重新从上电后的状态开始运行。

12. QSPI Flash(W25Q256FVEI)

这是开发板的 Flash 芯片，存储容量为 256Mbit（32M 字节），用于存储 ZYNQ 芯片的镜像数据，包括 PS 端的程序数据和 PL 端的配置镜像。

13. eMMC

eMMC 是非易失性 NAND 存储器，俗称电子硬盘，启明星核心板的 eMMC 芯片型号为 KLM8G1GETF，存储容量为 8GB，能够满足 PS 端的大容量非易失性存储需求。

2.2.2 软件资源说明

上面我们简单介绍了启明星 ZYNQ 开发板的硬件资源。接下来，我们将向大家简要介绍一下启明星 ZYNQ 开发板的软件资源。

启明星 ZYNQ 开发板 PL 端提供的标准例程多达 17 个，启明星 ZYNQ 开发板 PS 端提供的标准例程多达 34 个。我们提供的这些例程，全部都是原创自主开发，注释非常详细、代码风格统一、难易程度由浅入深，非常适合初学者入门。而其他家开发板的例程，要么注释比较少，要么工程文件管理不统一，对初学

者来说可能很难入门。

启明星 ZYNQ 开发板 PL 端的例程列表如下表所示：

表 2.2.1 启明星 ZYNQ 开发板 PL 端的例程

编号	实验名称	编号	实验名称
1	LED闪烁实验	10	RGB TFT-LCD彩条显示实验
2	按键控制LED灯实验	11	RGB LCD字符和图片显示实验
3	按键控制蜂鸣器实验	12	HDMI彩条显示实验
4	触摸按键控制LED灯实验	13	HDMI方块移动实验
5	呼吸灯实验	14	EEPROM读写实验
6	IP核之MMCM/PLL实验	15	RTC实时时钟LCD显示实验
7	IP核之RAM实验	16	频率计实验
8	IP核之FIFO实验	17	高速AD/DA实验
9	UART串口通信实验	18	

启明星 ZYNQ 开发板 PS 端的例程列表如下表所示：

表 2.2.2 启明星 ZYNQ 开发板 PS 端的例程

编号	实验名称	编号	实验名称
1	Hello World实验	18	PS通过VDMA驱动LCD显示实验
2	MIO控制LED闪烁实验	19	SD卡读BMP图片LCD显示实验
3	EMIO按键控制LED实验	20	SD卡读BMP图片HDMI显示实验
4	MIO按键中断控制LED实验	21	OV7725摄像头LCD显示实验
5	AXI GPIO按键控制LED实验	22	OV7725摄像头HDMI显示实验
6	自定义IP核-控制呼吸灯实验	23	OV5640摄像头LCD显示实验
7	固化程序（PS、PS+PL）实验	24	OV5640摄像头HDMI显示实验
8	UART串口中断实验	25	OV7725摄像头Sobel边沿检测实验
9	定时器中断实验	26	OV7725照相机实验
10	PS XADC接口实验	27	RGB LCD触摸画板实验
11	QSPI Flash读写测试实验	28	双目OV5640摄像头LCD显示实验
12	SD卡读写TXT文本实验	29	双目OV5640摄像头HDMI显示实验
13	双核AMP实验	30	基于lwip的echo server实验
14	基于BRAM的PL和PS的数据交互	31	基于lwip的TCP服务器性能测试实验

15	AXI4接口之DDR读写实验	32	基于lwip的tftp server实验
16	AXI DMA环路测试	33	基于TCP的远程更新QSPI实验
17	IP核封装与接口定义	34	基于UDP的远程更新QSPI实验

从上表可以看出，启明星 ZYNQ 开发板的例程是非常丰富的，并且扩展了很多有价值的例程。各个例程的难度是循序渐进的，首先从最基础的 LED 灯闪烁实验开始，然后一步步深入，从简单到复杂，有利于大家的学习和掌握，所以，启明星 ZYNQ 开发板是非常适合初学者的。当然，对于想深入学习 ZYNQ 开发的朋友，启明星 ZYNQ 开发板也是一个绝佳的选择。

第三章　硬件资源详解

本章，我们将向大家详细介绍启明星 ZYNQ 开发板各部分的硬件原理图，让大家对启明星 ZYNQ 开发板的各部分硬件原理有个深入理解，并向大家介绍开发板的使用注意事项，为后面的学习做好准备。

本章包括以下几个部分：

3.1　启明星 ZYNQ 的 IO 分配

3.2　核心板外设详解

3.3　底板外设详解

3.4　开发板使用注意事项

3.5　ZYNQ 的学习方法

3.1 启明星 ZYNQ 的 IO 分配

ZYNQ-7020 核心板的主控芯片为 XC7Z020CLG400-2，ZYNQ-7010 核心板的主控芯片为 XC7Z010CLG400-1。XC7Z020 芯片比 XC7Z010 芯片多出一个 BANK，即 BANK13。为了使 ZYNQ-7020 核心板和 ZYNQ-7010 核心板的引脚完全兼容，ZYNQ-7020 核心板的 BANK 13 我们没有连接到任何硬件外设，即没有引出至 BTB 转接板。

XC7Z020 芯片有 6 个用户 I/O BANK（比 XC7Z010 多一个 BANK）和最大 253 个用户 I/O，了解器件的 IO 分配方式，有助于我们在硬件设计时根据器件的一些约束，对设计进行合理的 IO 分配，减少硬件出错的可能性。ZYNQ 的 IO 口分成了 PL 和 PS 两部分，我们将分别介绍 PL 和 PS 两部分的 IO 分配。

3.1.1 PL 端的 IO 分配

ZYNQ 的 PL 侧和传统 FPGA 一样，可以灵活地分配到不同的 IO 口上。在 XC7Z020 中，PL 端的 IO 被分成了 3 组，每一组称为一个 IO Bank，分别是 BANK13、BANK34、BANK35。同一个 Bank 中的所有 IO 供电相同，而各个 Bank 的 IO 供电可以不同，启明星开发板都将它们连接到了 3.3V 的电源上。PL 端的 3 个 IO BANK 分别如图 3.1.1、图 3.1.2 和图 3.1.3 所示：

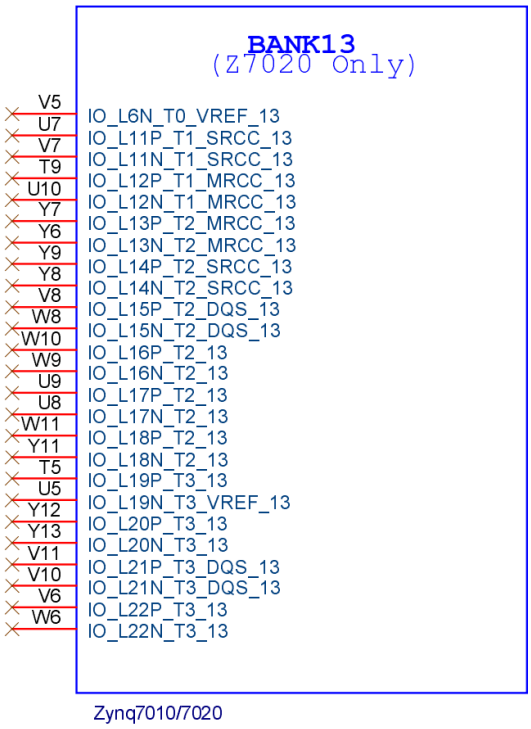


图 3.1.1 PL 端的 BANK13

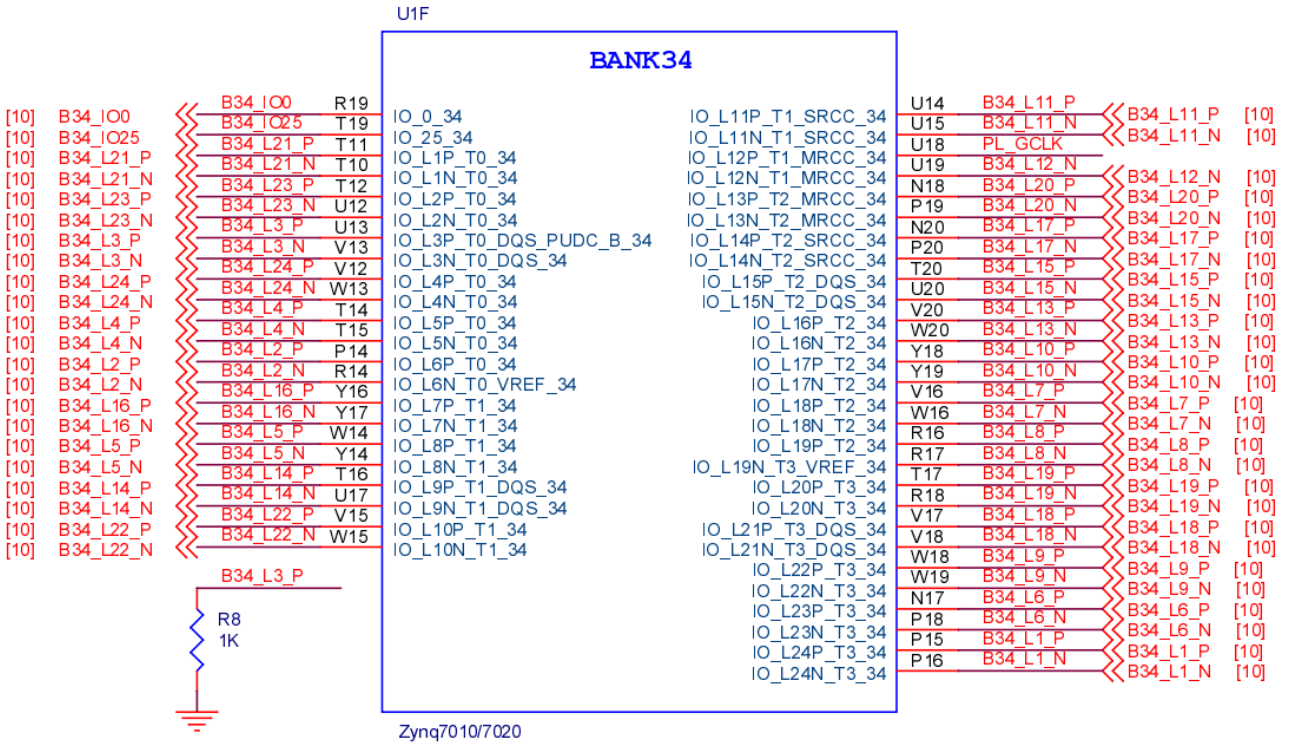


图 3.1.2 PL 端的 BANK34

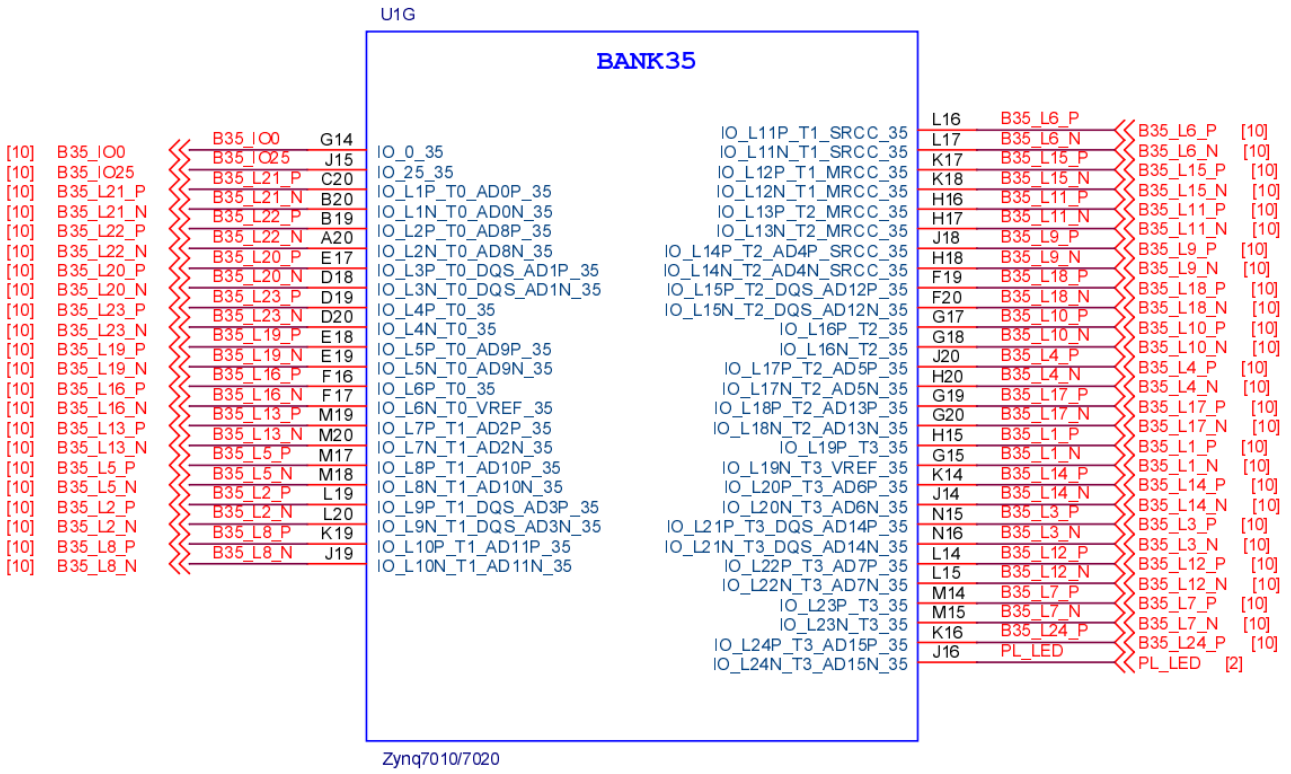


图 3.1.3 PL 端的 BANK35

需要说明的是，XC7Z020 芯片比 XC7Z010 芯片多出一个 BANK13，为了使这两颗芯片的引脚完全兼容，XC7Z020 芯片的 BANK13 没有连接到任何硬件外设。

为了让大家更快更好的使用我们的启明星 ZYNQ 开发板，这里特地将 ZYNQ PL 端的 IO 引脚分配做了

一个总表，以便大家查阅。启明星 ZYNQ PL 端 IO 引脚分配总表如下表所示：

表 3.1.1 启明星 ZYNQ PL 端 IO 引脚分配总表

信号名	方向	管脚	端口说明
系统时钟（50Mhz）			
sys_clk	input	U18	系统时钟，频率：50Mhz
PL 复位按键			
sys_rst_n	input	J15	PL 复位复位，低电平有效
2 个 PL 功能按键			
key[0]	input	L20	PL 按键 KEY0
key[1]	input	J20	PL 按键 KEY1
3 个 PL_LED 灯			
led[0]	output	J18	（底板）PL_LED0
led[1]	output	H18	（底板）PL_LED1
led	output	J16	（核心板）PL_LED
触摸按键			
touch_key	input	L19	触摸按键
蜂鸣器			
beep	output	G18	蜂鸣器
ATK MODULE			
uart_rx	input	P19	RXD 端口
uart_tx	output	N18	TXD 端口
gbc_key	input	J20	KEY 端口
gbc_led	output	H20	LED 端口
IIC 总线(EEPROM/RTC 实时时钟/)			
iic_scl	output	M17	IIC 时钟信号线
iic_sda	inout	M18	IIC 双向数据线
RGB TFT-LCD 接口			
lcd_hs	output	U17	RGB LCD 行同步
lcd_vs	output	P20	RGB LCD 场同步
lcd_de	output	N20	RGB LCD 数据使能
lcd_bl	output	Y16	RGB LCD 背光控制
lcd_clk	output	T16	RGB LCD 像素时钟
lcd_rgb[0]	output	Y18	RGB LCD 蓝色（最低位）
lcd_rgb[1]	output	Y19	RGB LCD 蓝色
lcd_rgb[2]	output	W20	RGB LCD 蓝色
lcd_rgb[3]	output	V20	RGB LCD 蓝色
lcd_rgb[4]	output	U14	RGB LCD 蓝色
lcd_rgb[5]	output	U15	RGB LCD 蓝色
lcd_rgb[6]	output	T20	RGB LCD 蓝色

lcd_rgb[7]	output	U20	RGB LCD 蓝色（最高位）
lcd_rgb[8]	output	W14	RGB LCD 绿色（最低位）
lcd_rgb[9]	output	Y14	RGB LCD 绿色
lcd_rgb[10]	output	N15	RGB LCD 绿色
lcd_rgb[11]	output	N16	RGB LCD 绿色
lcd_rgb[12]	output	V16	RGB LCD 绿色
lcd_rgb[13]	output	W16	RGB LCD 绿色
lcd_rgb[14]	output	W18	RGB LCD 绿色
lcd_rgb[15]	output	W19	RGB LCD 绿色（最高位）
lcd_rgb[16]	output	T10	RGB LCD 红色（最低位）
lcd_rgb[17]	output	T11	RGB LCD 红色
lcd_rgb[18]	output	P14	RGB LCD 红色
lcd_rgb[19]	output	R14	RGB LCD 红色
lcd_rgb[20]	output	V13	RGB LCD 红色
lcd_rgb[21]	output	U13	RGB LCD 红色
lcd_rgb[22]	output	G15	RGB LCD 红色
lcd_rgb[23]	output	H15	RGB LCD 红色（最高位）
lcd_scl	output	V17	触摸屏 IIC 接口的时钟
lcd_sda	inout	M19	触摸屏 IIC 接口的数据
ct_rst	output	Y17	触摸屏的复位
ct_int	input	V18	触摸屏的中断
HDMI 接口			
tmds_data_p[0]	output	K19	HDMI 的 DATA0 通道的 P 端
tmds_data_p[1]	output	M14	HDMI 的 DATA1 通道的 P 端
tmds_data_p[2]	output	L16	HDMI 的 DATA2 通道的 P 端
tmds_clk_p	output	L14	HDMI 的 CLK 通道的 P 端
tmds_oen	output	G17	HDMI 的输出使能信号
tmds_scl	output	V17	HDMI 的 SCL 信号
tmds_sda	output	M19	HDMI 的 SDA 信号
tmds_cec	output	H16	HDMI 的消费类电子控制信号
tmds_hpd	input	H17	HDMI 的热插拔信号
摄像头接口（OV5640/OV7725）			
cam_sgm_ctrl/cam_pwdn	output	R17	OV7725 时钟选择信号（0：使用引脚 XCLK 提供的时钟 1：使用摄像头自带的晶振提供时钟）/ OV5640 电源休眠控制信号
cam_rst_n	output	P15	cmos 复位信号，低电平有效
cam_vsync	input	R18	cmos 场同步信号
cam_href	input	T17	cmos 行同步信号

cam_pclk	input	T15	cmos 数据像素时钟
cam_data[0]	input	P16	cmos 数据
cam_data[1]	input	V15	cmos 数据
cam_data[2]	input	W15	cmos 数据
cam_data[3]	input	T12	cmos 数据
cam_data[4]	input	U12	cmos 数据
cam_data[5]	input	V12	cmos 数据
cam_data[6]	input	W13	cmos 数据
cam_data[7]	input	T14	cmos 数据
cam_scl	output	P18	cmos SCCB 时钟信号线
cam_sda	inout	N17	cmos SCCB 双向数据线

在上表中，表格中列出来了除扩展口外，开发板上所有的 PL IO 引脚，扩展口上的引脚 IO 可以参考开发板的原理图或者直接查看开发板上的丝印标注。

另外在资料盘（A 盘）→3_正点原子启明星 ZYNQ 开发板原理图文件夹下，有提供 Excel 格式的管脚分配表格，表格里共两个工作表，一个是“PL IO 引脚列表”，另一个是“PS IO 引脚列表”，方便大家查看。

3.1.2 PS 端的 IO 分配

在 ZYNQ PS 端同样也包含了 3 个 IO BANK，如下图所示：

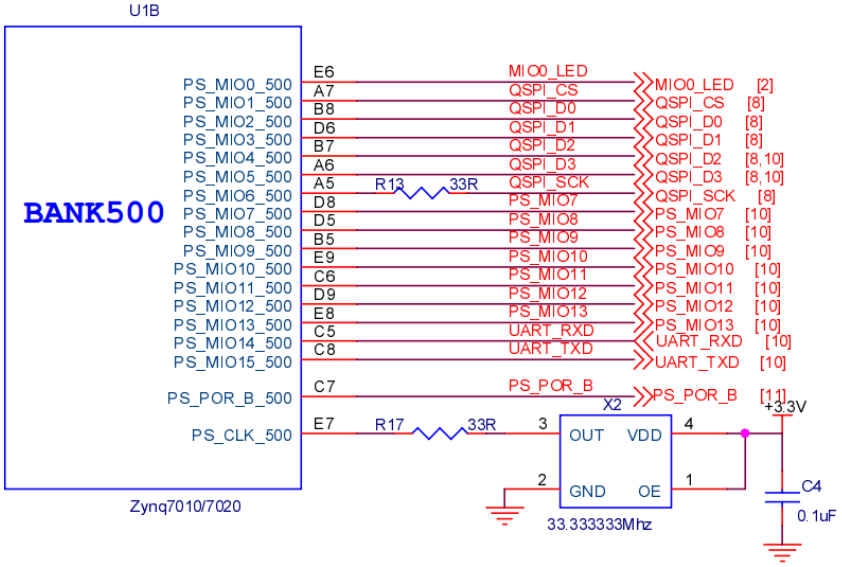


图 3.1.4 PS 端的 BANK500

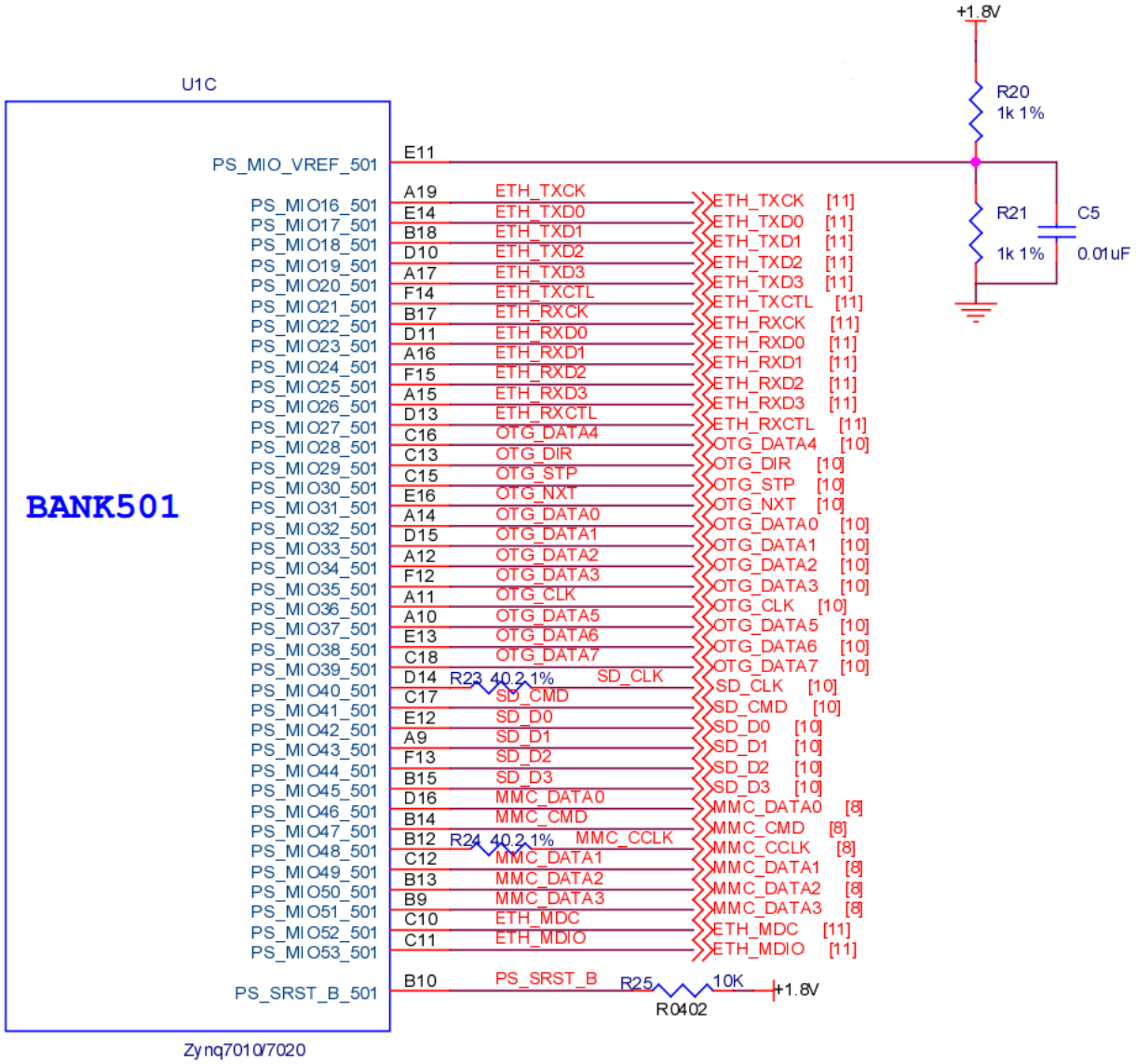


图 3.1.5 PS 端的 BANK501

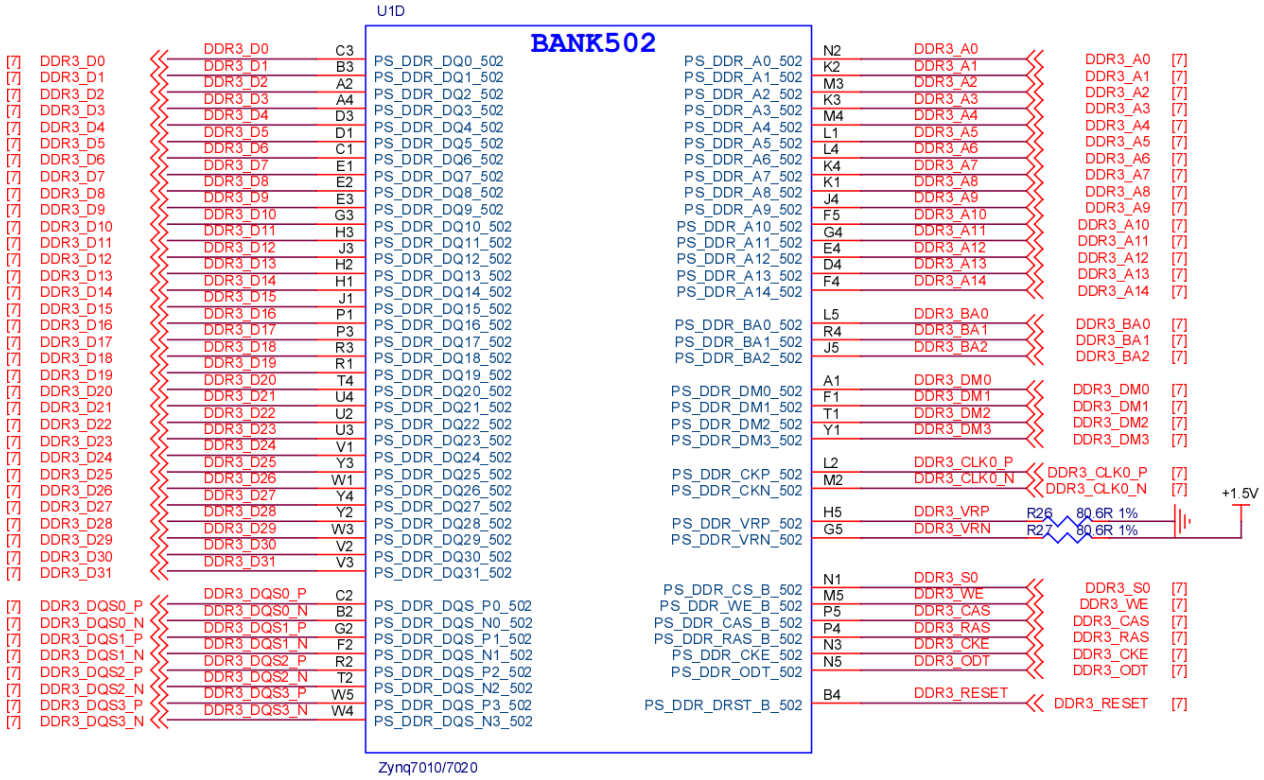


图 3.1.6 PS 端的 BANK502

ZYNQ 不同的 IO BANK 可以采用不同的供电电压，IO BANK 的引脚电平和供电电压的电平保持一致，我们根据 PS 端连接不同的外设来选择不同的供电电压。BANK500、BANK501 和 BANK502 分别采用 3.3V、1.8V 和 1.5V 供电。

与 PL 端 IO BANK 不同的是，PS 端的 IO 连接是是相对固定的，用户不能够像 PL 引脚那样，将 PS 端的 IO 随意分配到某个外设。PS 端外设的 IO 口与 MIO（多路复用 IO）之间，具有固定关系的映射，某个外设的 IO 口可能会映射到不同的 MIO 上，某个 MIO 也有可能会具有多个外设的 IO 口映射到其上面。两者之间的映射如下图所示：

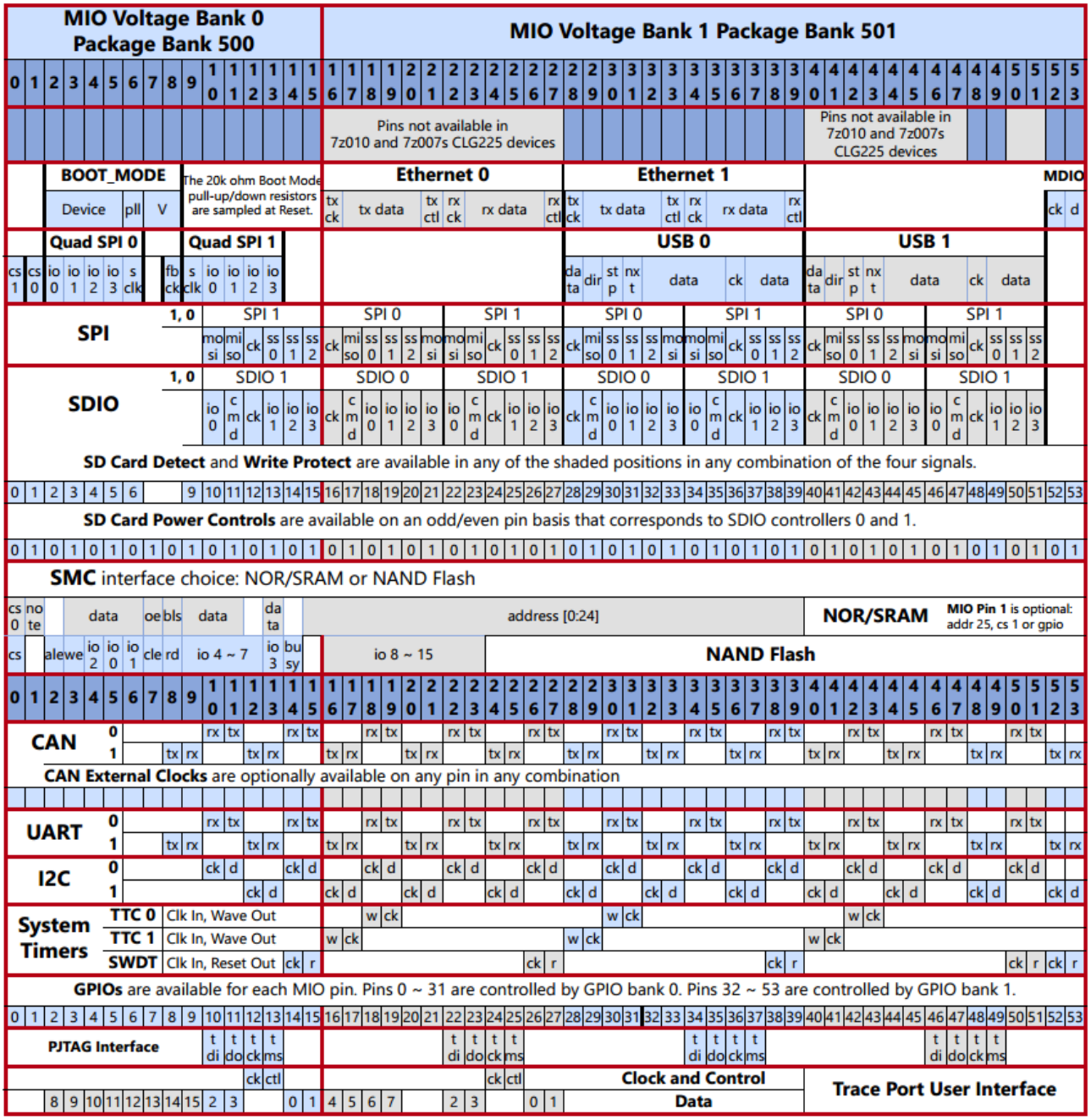


图 3.1.7 PS 端外设和 MIO 之间的映射

启明星 ZYNQ 开发板所使用到的所有外设与 MIO 之间所选择的映射关系如下表所示：

表 3.1.2 启明星 ZYNQ PS 端 IO 引脚分配总表

信号名	管脚	端口说明
2个PS功能按键		
ps_key[0]	MIO12	PS按键KEY0
ps_key[1]	MIO11	PS按键KEY1
3个PS_LED灯		
ps_led[0]（底板）	MIO7	PS_LED0
ps_led[1]（底板）	MIO8	PS_LED1

ps_led（核心板）	MIO0	PS_LED
QSPI FLASH		
QSPI_CS#	MIO1	QSPI FLASH的片选，低电平有效
QSPI_SCK	MIO6	QSPI FLASH的时钟
QSPI_D0	MIO2	QSPI FLASH的数据位0
QSPI_D1	MIO3	QSPI FLASH的数据位1
QSPI_D2	MIO4	QSPI FLASH的数据位2
QSPI_D3	MIO5	QSPI FLASH的数据位3
PS UART		
PS_UART_RXD	MIO14	PS UART的接收
PS_UART_TXD	MIO15	PS UART的发送
PS 以太网		
ETH_TXCK	MIO16	PS以太网RGMII接口的TX_CLK
ETH_TXD0	MIO17	PS以太网RGMII接口的TX_D0
ETH_TXD1	MIO18	PS以太网RGMII接口的TX_D1
ETH_TXD2	MIO19	PS以太网RGMII接口的TX_D2
ETH_TXD3	MIO20	PS以太网RGMII接口的TX_D3
ETH_TXCTL	MIO21	PS以太网RGMII接口的TX_CTL
ETH_RXCK	MIO22	PS以太网RGMII接口的RX_CLK
ETH_RXD0	MIO23	PS以太网RGMII接口的RX_D0
ETH_RXD1	MIO24	PS以太网RGMII接口的RX_D1
ETH_RXD2	MIO25	PS以太网RGMII接口的RX_D2
ETH_RXD3	MIO26	PS以太网RGMII接口的RX_D3
ETH_RXCTL	MIO27	PS以太网RGMII接口的RX_CTL
ETH_MDC	MIO52	PS以太网MDIO接口的时钟
ETH_MDIO	MIO53	PS以太网MDIO接口的数据
PS USB接口		
OTG_DIR	MIO29	USB总线方向控制
OTG_STP	MIO30	数据传输的结束信号
OTG_NXT	MIO31	当前数据接收完成指示信号
OTG_CLK	MIO36	PHY的时钟输出
OTG_DATA7	MIO39	双向数据总线位7
OTG_DATA6	MIO38	双向数据总线位6
OTG_DATA5	MIO37	双向数据总线位5
OTG_DATA4	MIO28	双向数据总线位4
OTG_DATA3	MIO35	双向数据总线位3
OTG_DATA2	MIO34	双向数据总线位2
OTG_DATA1	MIO33	双向数据总线位1
OTG_DATA0	MIO32	双向数据总线位0
SD卡		

SD_CLK	MIO40	SD卡的时钟信号
SD_CMD	MIO41	SD卡的命令信号
SD_D0	MIO42	SD卡的DATA0
SD_D1	MIO43	SD卡的DATA1
SD_D2	MIO44	SD卡的DATA2
SD_D3	MIO45	SD卡的DATA3
eMMC存储器		
eMMC_CCLK	MIO48	eMMC的时钟信号
eMMC_CMD	MIO47	eMMC的命令信号
eMMC_D0	MIO46	eMMC的DATA0
eMMC_D1	MIO49	eMMC的DATA1
eMMC_D2	MIO50	eMMC的DATA2
eMMC_D3	MIO51	eMMC的DATA3

3.2 开发板底板原理图详解

3.2.1 底板电源

底板电源拓扑结构如下图所示：

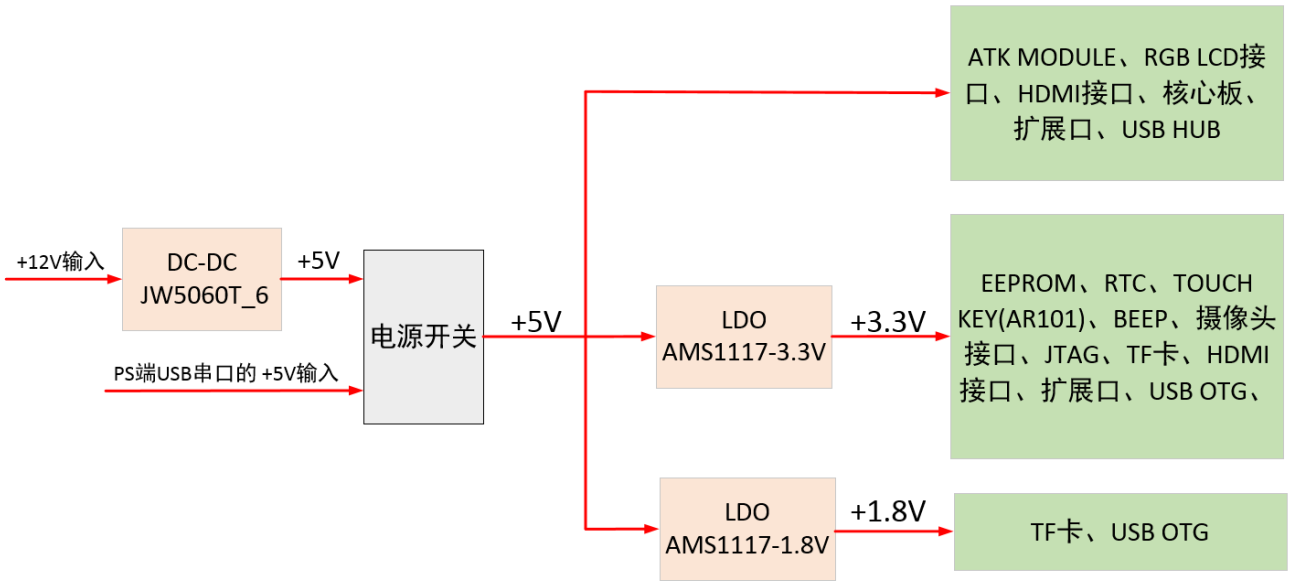


图 3.2.1 底板电源拓扑

整个板子的电源来自电源适配器的 12V 供电电压，经过 DC-DC 芯片（型号为 JW50690T_6）之后，转变为系统+5V 电压，再经过电源开关后，给系统供电，核心板的电源就来自系统+5V 电压。PS 端 USB 串口的+5V 也可以用来给系统供电，但是需要特别注意，由于 USB 接口的驱动能力有限，所以不能驱动供电电流需求大的器件如 LCD，这种情况下，请改用电源适配器供电。系统+5V 电压除了直接驱动某些外设之外，还用于电压转换，LDO（线性稳压器）芯片 AMS1117-3.3 和 AMS1117-1.8 用于将系统+5V 电压分别转变为+3.3V 和+1.8V，然后给各个外设供电。

使用+5V 电压供电的底板外设包括 ATK MODULE、RGB LCD 接口、HDMI 接口、核心板、扩展口。使用+3.3V 电压供电的底板外设包括 SP3232、EEPROM、RTC、TOUCH KEY(AR101)、BEEP、摄像头接口、

JTAG、Micro SD、HDMI 接口、扩展口、USB OTG。使用+1.8V 电压供电的底板外设包括 Micro SD、USB OTG。

电源适配器供电的电源接口原理图如下图所示：

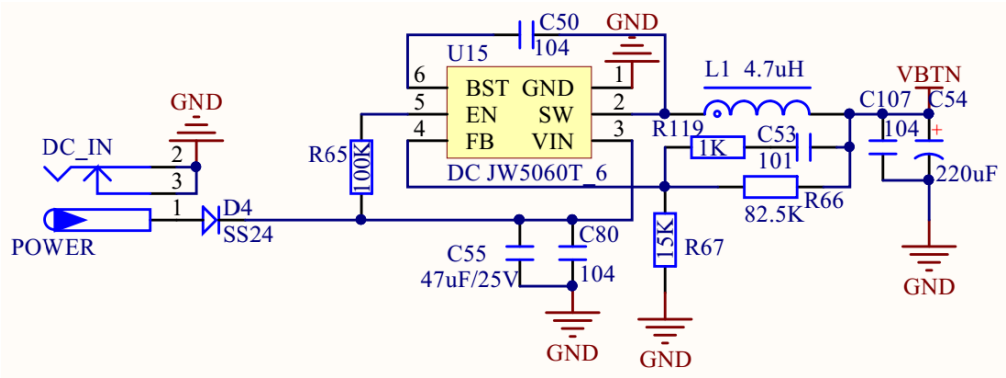


图 3.2.2 电源适配器供电电源接口

DC_IN 用于外部直流电源输入，输入电压（12V）经过 U15 DC-DC 芯片转换为 5V 电源输出，其中 D4 是防反接二极管，避免外部直流电源极性接反的时候，烧坏开发板。

ZYNQ 开发板底板的供电电源可以从电源适配器供电，也可以通过 USB 供电，无论通过哪种方式供电，都经过电源开关 K1 控制是否对开发板供电，其原理图如下图所示：

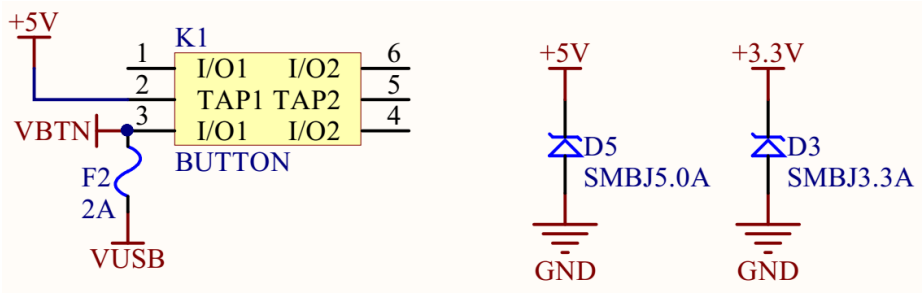


图 3.2.3 电源按键开关

VBTN 为电源适配器输入的 12V 电压经电源转换芯片转换后得到的 5V 电压，VUSB 为通过 USB 接口输入的电压。

除了+5V 电压之外， ZYNQ开发板底板还提供了 3.3V、1.8V 的电源电压，都是由 5V 电压转换而来，其电压转换电路原理图如下图所示：

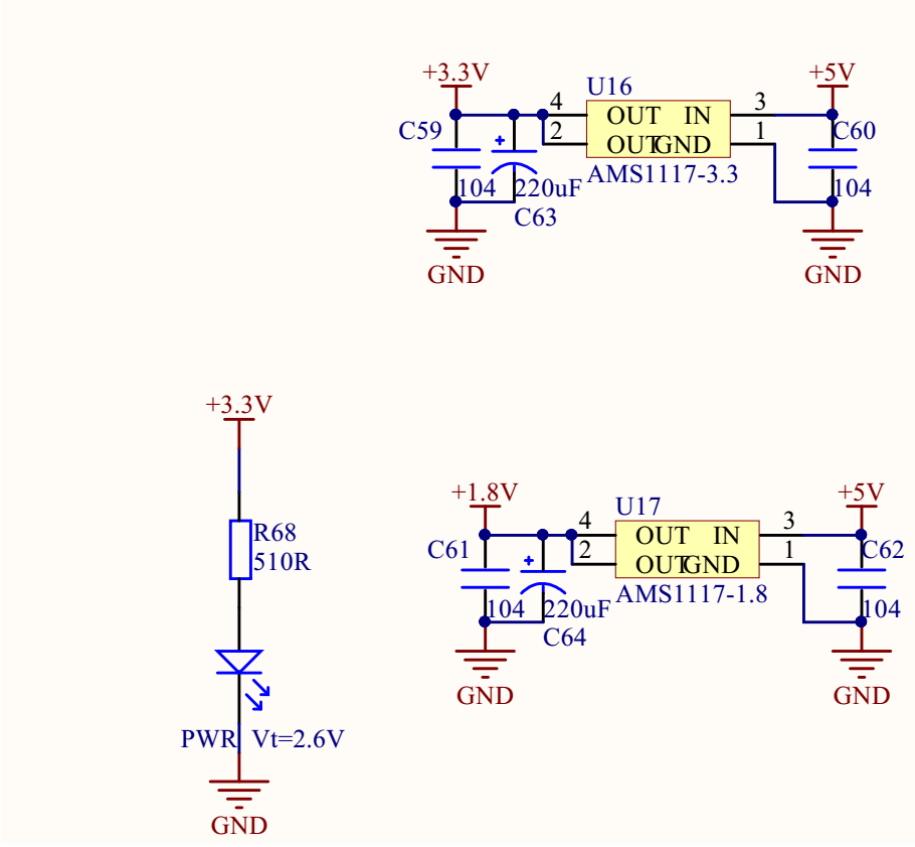


图 3.2.4 电压转换电路

当电路接通后，LDO（线性稳压器）芯片 AMS1117-3.3 和 AMS1117-1.8 将系统+5V 电压分别转变为+3.3V 和+1.8V。通过电压转换芯片得到 3.3V 电压，点亮电源指示灯 PWR，可以通过电源指示灯 PWR 判断开发板供电是否正常。

除了这些之外，启明星 ZYNQ开发板还板载了两组 5V 和 3.3V 输入输出接口，其原理图如下图所示：

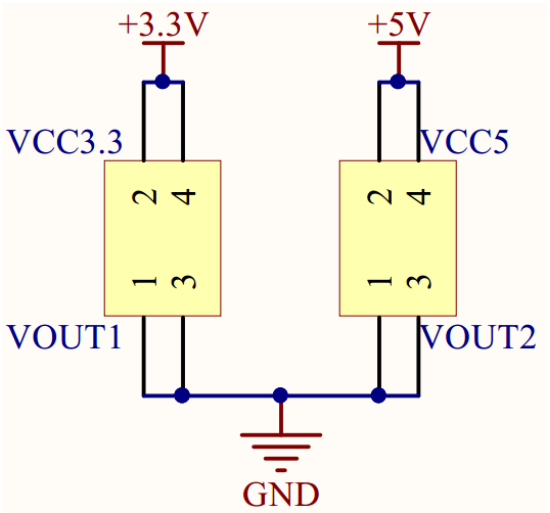


图 3.2.5 电源输入输出接口

图中，VOUT1 和 VOUT2 分别是 3.3V 和 5V 的电源输入输出接口，有了这 2 组接口，我们可以通过开发板给外部提供 3.3V 和 5V 电源了，虽然功率不大（最大 1000ma），但是一般情况都够用了，大家在调试自己的小电路板的时候，有这两组电源还是比较方便的。同时这两组端口，也可以用来由外部给开发板供

电。

3.2.2 ZYNQ 启动模式

ZYNQ 支持 4 种启动模式，分别是 JTAG、NAND、QSPI FLASH 和 SD Card。为了方便用户的操作，启明星底板板载了一个控制 ZYNQ 启动模式的拨码开关，原理图如图 3.2.6 所示。

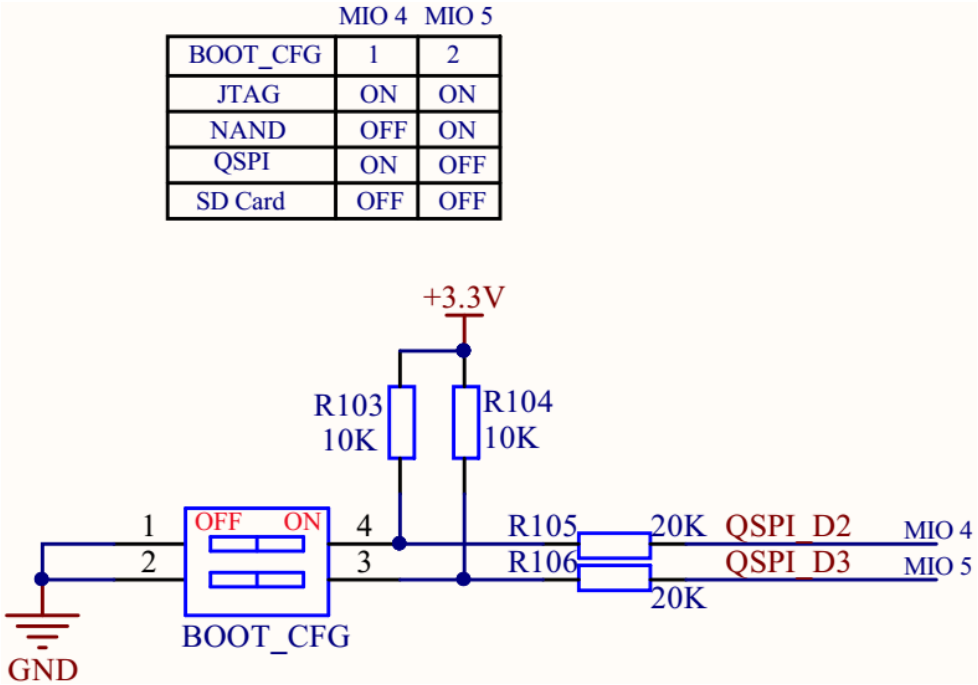


图 3.2.6 PS BOOT 模式选择开关

PS 的模式选择开关由 MIO4（和 QSPI_D2 引脚复用）和 MIO5（和 QSPI_D3 引脚复用）引脚控制，底板上的两个拨码开关用于设置 ZYNQ 的 BOOT 模式：

当拨开开关 1 设置为“ON”、拨开开关 2 设置为“ON”时，ZYNQ PS 端在上电后的启动源被设置为 JTAG；

当拨开开关 1 设置为“ON”、拨开开关 2 设置为“OFF”时，ZYNQ PS 端在上电后的启动源被设置为 NAND FLASH。由于 ZYNQ 开发板并没有板载 NAND FLASH 存储器，因此不能从 NAND FLASH 启动，此模式对于本开发板来说没有用。

当拨开开关 1 设置为“OFF”、拨开开关 2 设置为“ON”时，ZYNQ PS 端在上电后的启动源被设置为 QSPI FLASH；

当拨开开关 1 设置为“OFF”、拨开开关 2 设置为“OFF”时，ZYNQ PS 端在上电后的启动源被设置为 SD Card。

3.2.3 有源蜂鸣器

蜂鸣器是现代常用的一种电子发声器，主要用于产生声音信号。蜂鸣器（Buzzer）常用于计算机、打印机、报警器、电子玩具等电子产品中。常用的蜂鸣器有两种：有源蜂鸣器和无源蜂鸣器，这里的有“源”不是电源，而是震荡源，有源蜂鸣器内部带有震荡源，所以有源蜂鸣器只要通电就会叫。无源蜂鸣器内部不带震荡源，直接用直流电是驱动不起来的，需要 2K-5K 的方波去驱动。

为方便大家使用，启明星底板使用的是有源蜂鸣器，原理图如下图所示：

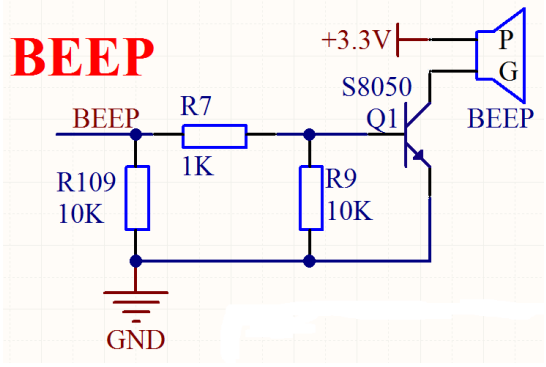


图 3.2.7 有源蜂鸣器

图中通过一个 NPN 型三极管 S8050 来驱动蜂鸣器，BEEP 连接到 PL 的 IO，这个 IO 可以控制三极管 Q1 的导通。当 BEEP 输出高电平的时候 Q1 导通，相当于蜂鸣器的正极连接到+3.3V，蜂鸣器形成一个通路，因此蜂鸣器会鸣叫。同理，当 BEEP 输出低电平的时候 Q1 不导通，那么蜂鸣器就没有形成一个通路，因此蜂鸣器也就不会鸣叫。

3.2.4 PL LED

启明星底板板载了两个 PL LED，在调试代码的时候，使用 LED 来指示程序执行状态，是一个很好的辅助调试方法。其电路原理图如下图所示：

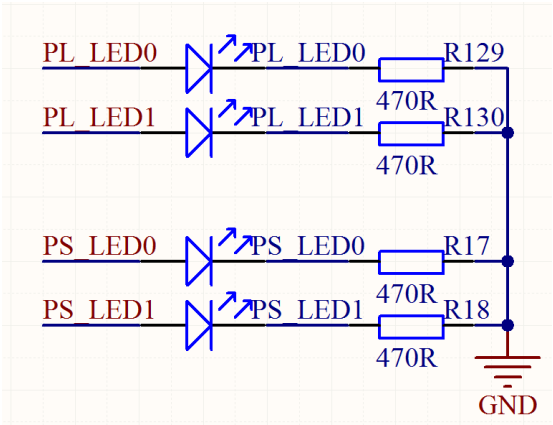


图 3.2.8 LED

上面的“PL_LED0”和“PL_LED1”是 PL 的 2 个 LED，当 PL 的 IO 口输出高电平“1”时，点亮 LED 灯，输出低电平“0”时，熄灭 LED 灯。

3.2.5 PS LED

启明星底板板载了两个 PS LED，在调试代码的时候，使用 LED 来指示程序执行状态，是一个很好的辅助调试方法。电路原理图如下图所示：

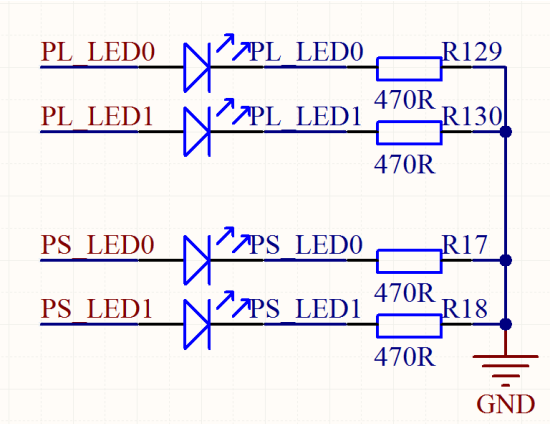


图 3.2.9 LED

其中下面的“PS_LED0”和“PS_LED1”是连接到 PS 的两个 LED，当 PS 的 IO 口输出高电平“1”时，点亮 LED 灯，输出低电平“0”时，熄灭 LED 灯。

3.2.6 PL 按键

启明星 ZYNQ 开发板底板板载了一个 PL 复位按键和两个 PL 功能按键，其电路原理图如下图所示：

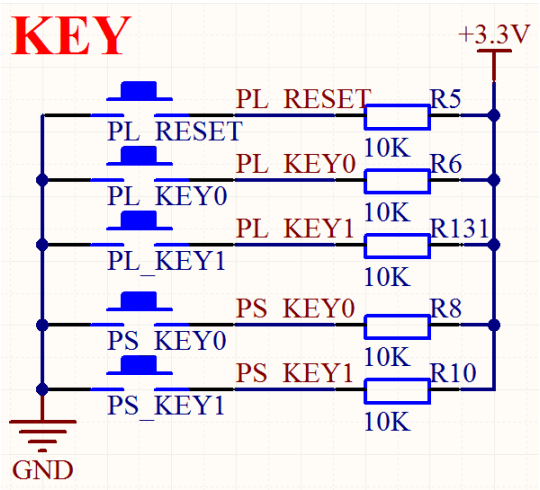


图 3.2.10 复位/按键接口

图中，最上面的“PL_RESET”按键为 PL 端的复位按键，按下时低电平复位，一般作为 PL 逻辑的系统复位信号。紧接着下面的“PL_KEY0”和“PL_KEY1”为 PL 端的功能按键输入。它们都外接上拉电阻，未按下时按键端口输出高电平，按下时输出低电平。按键作为最简单的输入设备，适合在需要给系统输入控制信号的场合使用。

3.2.7 PS 按键

启明星底板板载了两个 PS 功能按键，其电路原理图如下图所示：

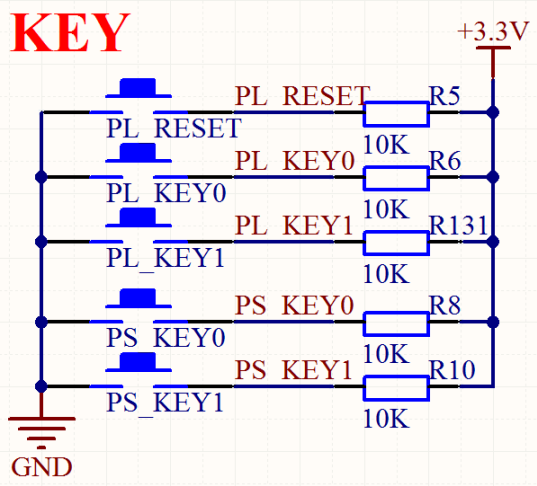


图 3.2.11 复位/按键接口

图中，最下面的两个按键“PS_KEY0”和“PS_KEY1”作为 PS 端的功能按键，它们都外接上拉电阻，未按下时按键端口输出高电平，按下时输出低电平。按键作为最简单的输入设备，适合在需要给系统输入控制信号的场合使用。

3.2.8 电容触摸按键

启明星底板板载一个触摸按键，触摸按键在稳定性、使用寿命、抗干扰能力等方面都优于传统的机械按键，被广泛应用于遥控器，便携电子设备，楼道电灯开关，各类家电控制面板等场景。下图是 ZYNQ 与触摸按键之间的连接框图：

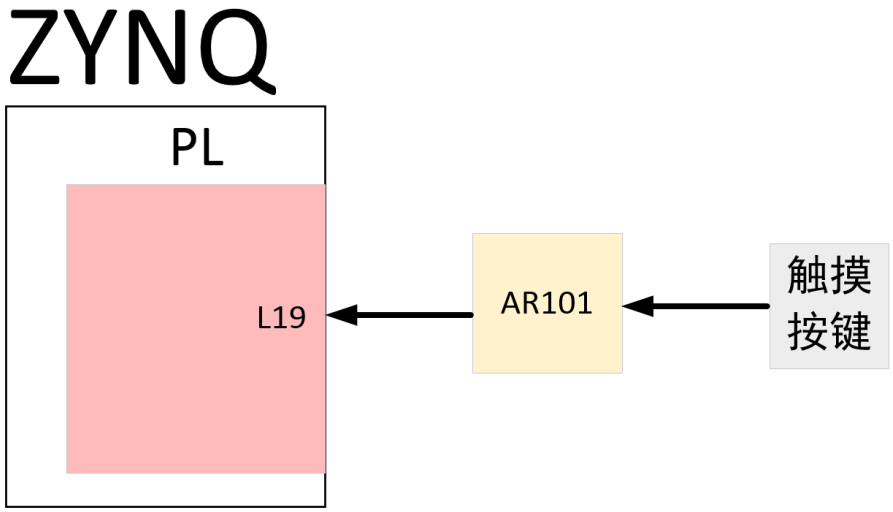


图 3.2.12 触摸按键连接框图

其电路原理图如下图所示：

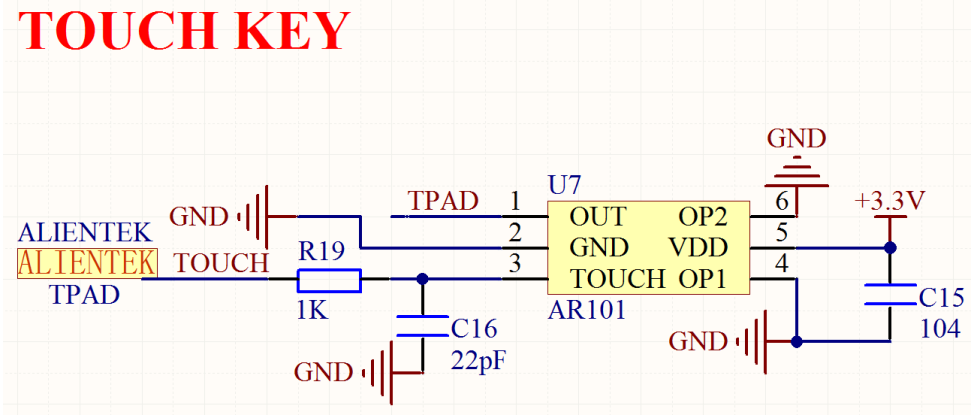


图 3.2.13 电容触摸按键

图中 1K 电阻是电容充电电阻，AR101 是触摸按键专用驱动芯片，用于产生稳定无毛刺的信号。 AR101 的 1 脚 TPAD 信号就是处理后的 IO 信号。当用户触摸了触摸按键时，TPAD 会变为高电平。用户不触摸按键时，TPAD 信号为低电平。

3.2.9 14-Pin JTAG 接口

启明星底板板载一个 14 针标准 JTAG 调试口（JTAG），该 JTAG 口与核心板的 6-Pin JTAG 接口是硬件连通的，该接口可以直接和 Xilinx 下载器（调试器）连接，用于下载程序或者对程序进行在线调试。

JTAG 接口一般由 TMS、TDI、TDO、TCK、3.3V 电源和 GND 组成。其原理图如下图所示：

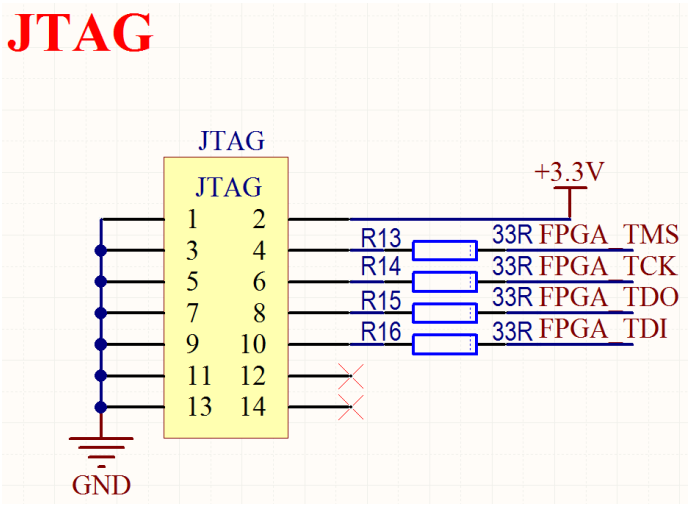


图 3.2.14 14 针标准 JTAG 调试口

3.2.10 USB 串口

启明星底板板载了一个 UART 转 USB接口的转换芯片，一般用于 PS 程序调日志显示功能，下图是其与 ZYNQ 之间的连接框图：

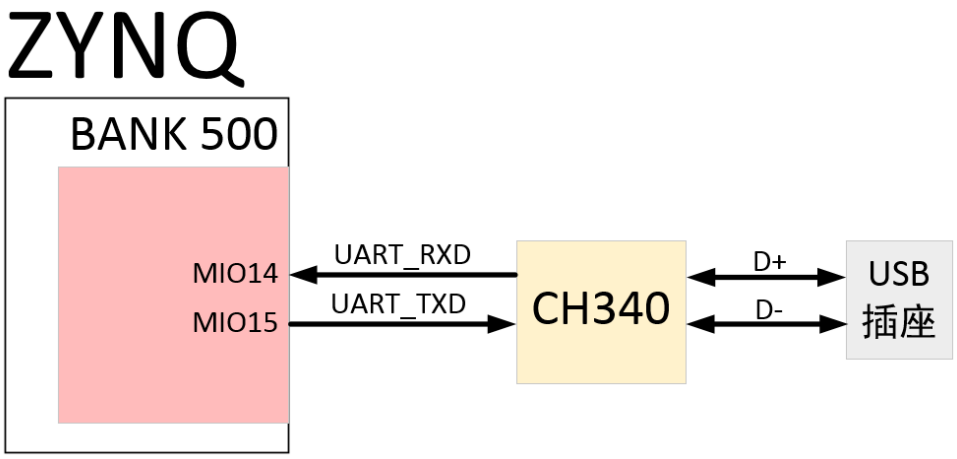


图 3.2.15 PS 端 UART 连接框图

其原理图如下图所示：

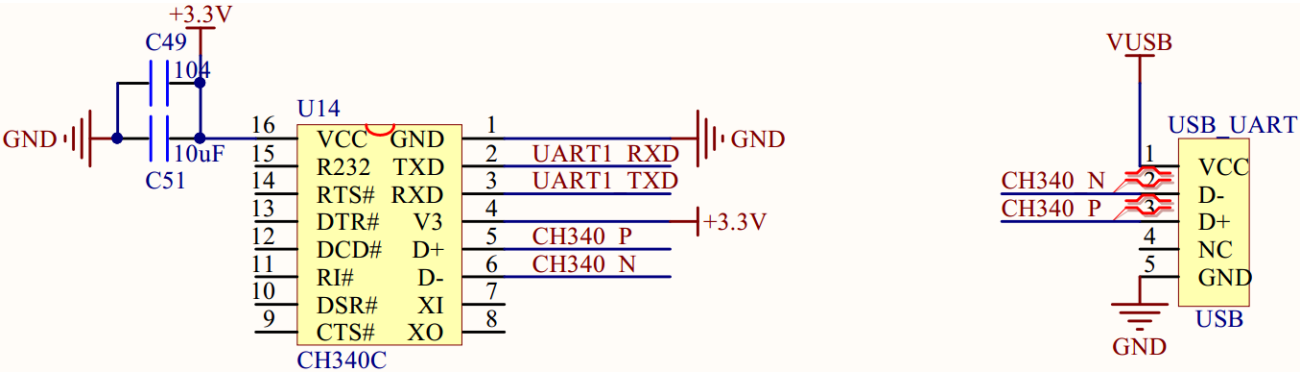


图 3.2.16 USB 串口

USB 转串口芯片，我们选择的是 CH340C，是国内芯片公司南京沁恒的产品，稳定性经测试还不错，所以我们还是要支持下国产。

USB_UART 是一个 MiniUSB 座，提供 CH340C 和电脑通信的接口，同时可以给开发板供电，VUSB 就是来自电脑 USB 的电源，USB_UART 是本开发板的可选供电口。

3.2.11 RGB LCD 模块接口

TFT-LCD 是一种液晶显示屏，它采用薄膜晶体管（TFT）技术提升图像质量，如提高图像亮度和对比度等。相比于传统的 CRT 显示器，TFT-LCD 有着轻薄、功耗低、无辐射、图像质量好等诸多优点，因此广泛应用于电视机、电脑显示器、手机等各种显示设备中。启明星底板板载的 RGB LCD 模块接口电路如下图所示：

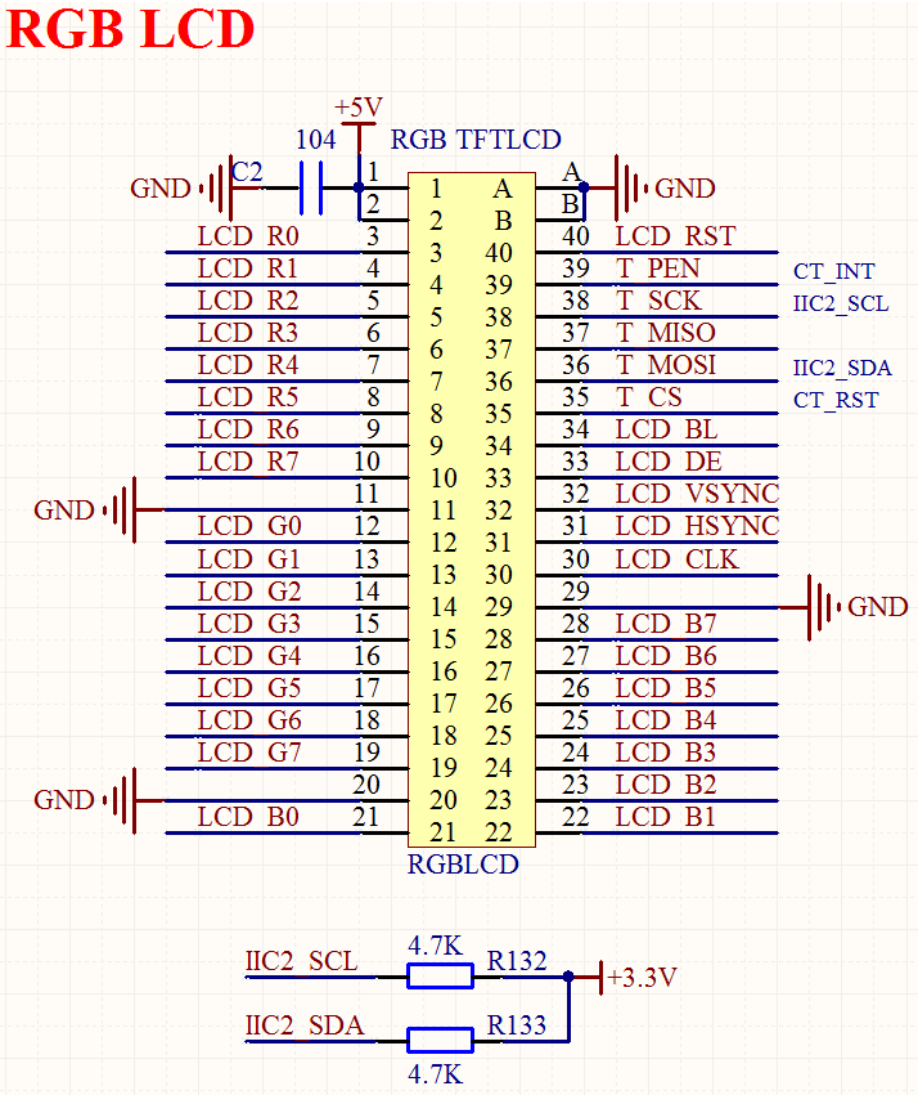


图 3.2.17 RGB LCD 模块接口

图中，RGBLCD 是 RGB LCD 液晶接口，采用 RGB888 数据格式，最大支持 256*256*256=1677W 色颜色显示，并支持触摸屏（支持电阻屏和电容屏）。该接口仅支持 RGB 接口的液晶（不支持 MCU 接口的液晶），目前正点原子的 RGB 接口 LCD 模块有：4.3 寸（ID:4342，480*272）和 7 寸（ID:7084，800*480 和 ID:7016，1024*600）等尺寸可选。

图中的 T_MISO/T_MOSI/T_PEN/T_CS/T_SCK 用于实现对液晶触摸屏的控制（支持电阻屏和电容屏），连接到 ZYNQ PL 端的 IO 的触摸屏的 4 个信号如图中的 CT_INT、CT_RST、IIC2_SCL、IIC2_SDA 所示，它们也可以使用 PS 端的 EMIO 来连接到 PS 系统。LCD_BL 用于控制 LCD 的背光，LCD_BL 为 1 表示打开背光。

3.2.12 OLED/摄像头模块接口

启明星 ZYNQ 开发板板载了一个 OLED/摄像头模块接口，电路原理图如下图所示：

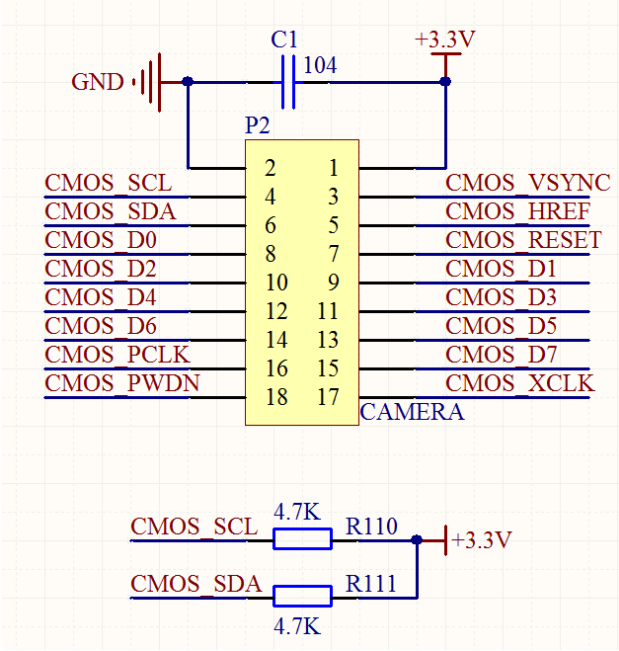


图 3.2.18 OLED/摄像头模块接口

图中的 OLED/摄像头接口可以用来连接正点原子 OLED 模块或者正点原子摄像头模块，正点原子摄像头模块支持 OV7725、OV5640 和其他类似接口的摄像头。

3.2.13 **EEPROM**

EEPROM (Electrically Erasable Progammable Read Only Memory，E2PROM)即电可擦除可编程只读存储器，是一种常用的非易失性存储器（掉电数据不丢失），也常在嵌入式领域中作为数据的存储设备，在物联网及可穿戴设备等需要存储少量数据的场景中也有广泛应用。

启明星底板板载的 EEPROM 是 Atmel 公司的 AT24C64，使用 I2C 接口进行通信，该芯片的容量为 64Kb，对于一般应用来说是足够了。驱动该 E2PROM 只需要用到两个 IO，其与 ZYNQ 之间的连接框图如下图所示：

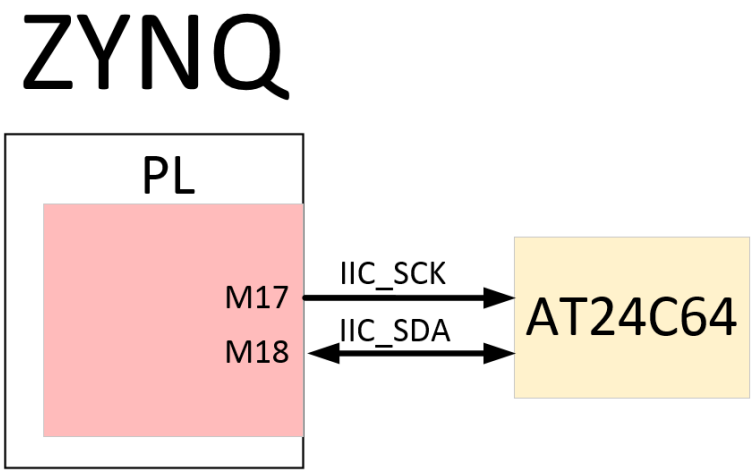


图 3.2.19 EEPROM 连接框图

其电路原理图如下图所示：

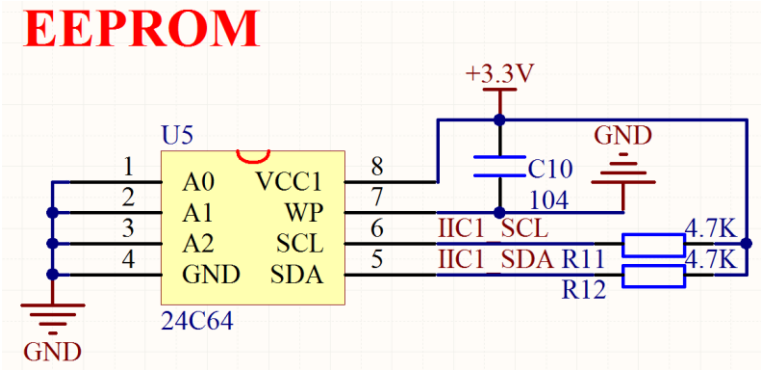


图 3.2.20 EEPROM

硬件原理图里我们把 A0~A2 均接地，对 AT24C64 来说也就是把地址位设置成了 0 了，写代码的时候要注意这点。另外 AT24C64 采用 IIC 协议进行数据的读写，而 IIC 的串行时钟线 SCL 和数据线 SDA 均是开漏的，所以需要接上拉电阻。

3.2.14 实时时钟

启明星底板板载了一个 I2C 接口的实时时钟 RTC，实时时钟芯片采用 NXP 公司的 PCF8563，可以提供年月日时分秒、星期以及日历功能。数字时钟一般用于给系统提供时间功能。PCF8563 与 ZYNQ 之间的连接框图如下图所示：

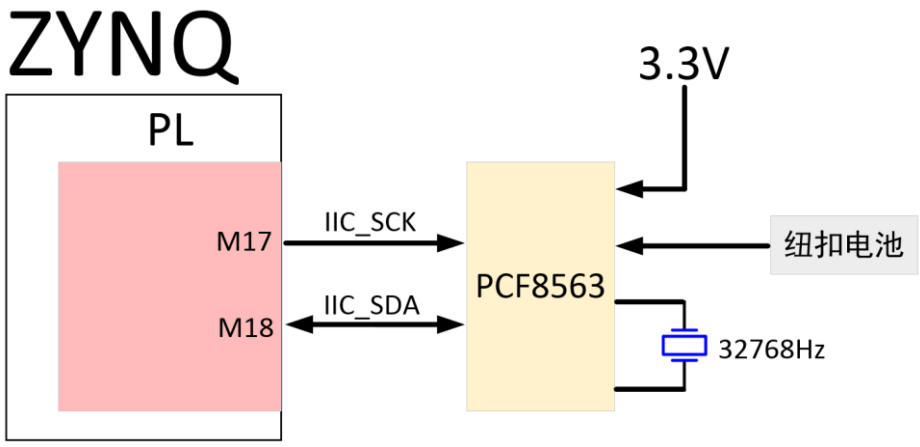


图 3.2.21 RTC 连接框图

其电路原理图如下图所示：

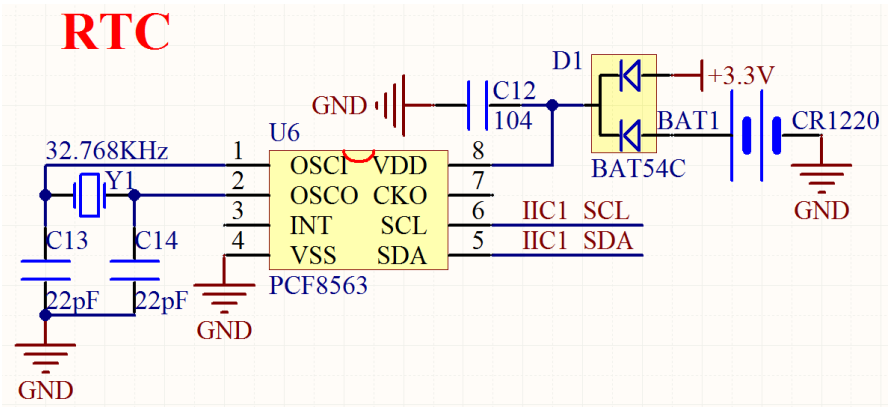


图 3.2.22 实时时钟 RTC

PCF8563 需要外接一个 32.768KHz 的无源时钟，采用直接供电（VCC3.3）和备用电源供电两种供电方式。PCF8563 采用 IIC 协议进行通信，与 EEPROM 共用 IIC 引脚。为了开发板掉电以后实时时钟还可以正常运行，还需要配一个电池给 PCF8563 供电，图中为 BAT1 为电池座，我们将纽扣电池（型号 CR1220，电压为 3V）放入 BAT1 插座以后，当系统掉电后，纽扣电池还可以继续给 PCF8563 供电。原理图中的 BAT54C 是一款正向电压为 320mV 的半导体二极管，防止 3.3V 和电池互相导通。

3.2.15 ATK 模块接口

启明星 ZYNQ 开发板板载了一个 ATK 模块接口，电路原理图如下图所示：

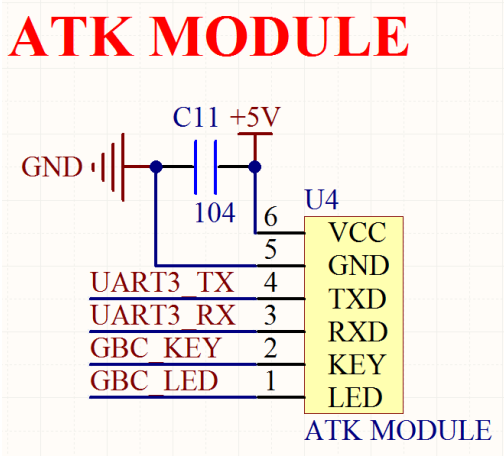


图 3.2.23 ATK 模块接口

如图所示，U4 是一个 1*6 的排母座，可以用来连接正点原子开发的“ATK-USB-UART 模块”或者其他 ATK 接口的外设模块。当连接“ATK-USB-UART 模块”时，则实现 UART 串口通信功能。其中，UART3_TX 和 UART3_RX 连接到了 PL 的 IO 口上。

3.2.16 HDMI 接口

启明星底板板载高清晰度多媒体接口 HDMI（High Definition Multimedia Interface，HDMI）接口，可以连接显示器用来实现图片和视频的显示。启明星开发板通过 PL 侧的差分 IO 直接连接到 HDMI 接口的差分信号和时钟，通过 FPGA 逻辑实现 HMDI 信号的差分转并行和编解码，实现数字视频输入和输出的传输解决方案，最高支持 1080P 的输入和输出功能。

其与 ZYNQ 之间的连接框图如下图所示：

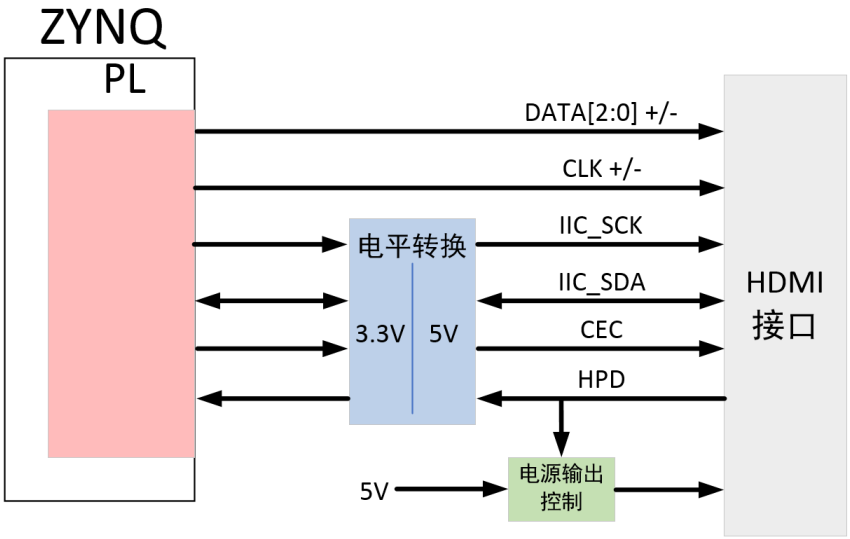


图 3.2.24 HDMI 连接框图

当开发板作为 HDMI 主设备时，HDMI 接口外接显示设备，HPD(hot plug detect)热插拔信号作为输入。开发板在作为 HDMI 主设备时，需要提供给 HDMI 显示设备一个+5V 的电源。另外 HMDI 主设备会通过 IIC 总线来读取 HDMI 显示设备的设备信息。

其电路原理图如下所示：

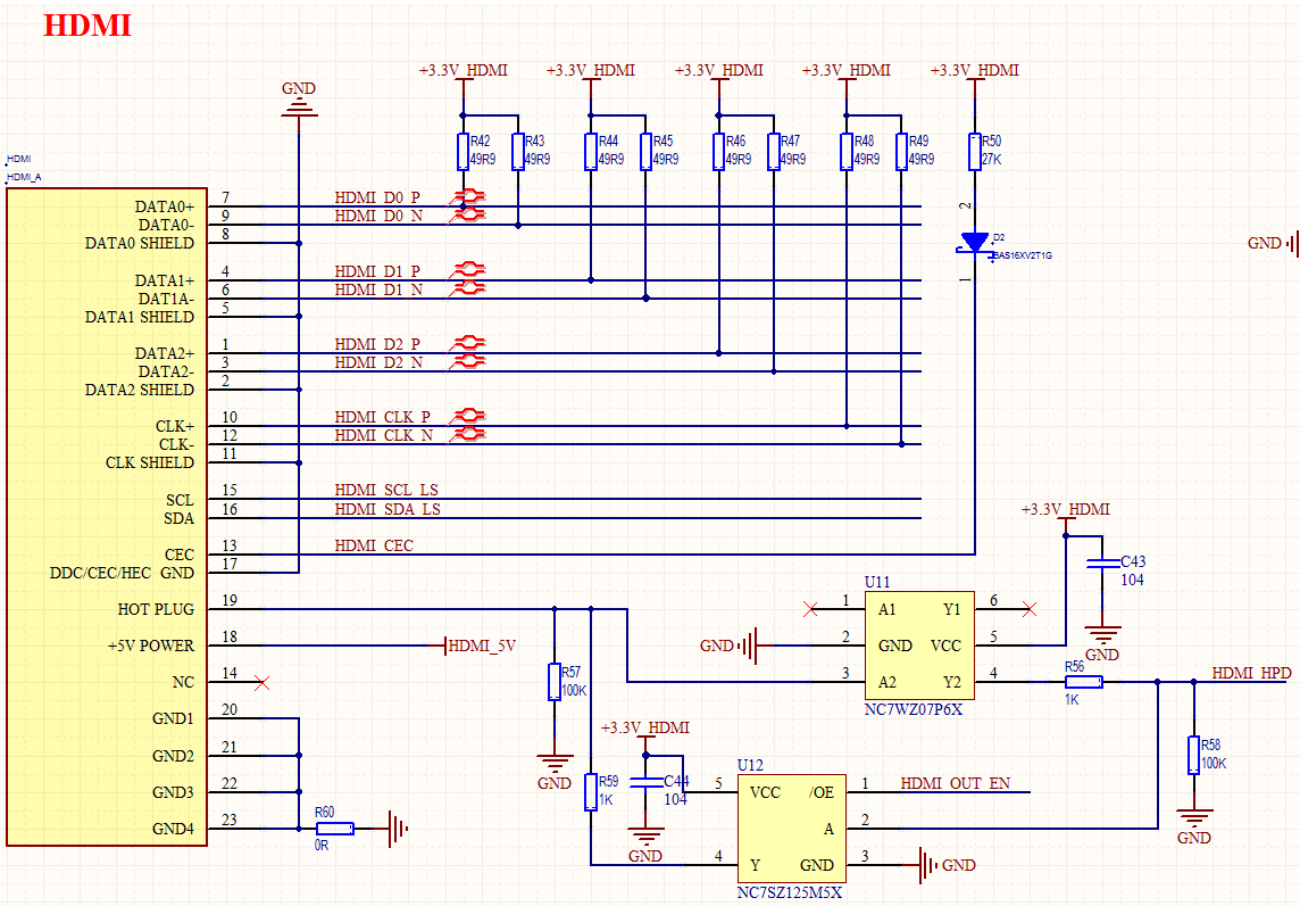


图 3.2.25 HDMI 接口

需要说明的是，HDMI 的 IIC 引脚是 5V 电平的，需要将其转换为 ZYNQ PL 端所需的 3.3V 电压。电平

转换电路如下图所示：

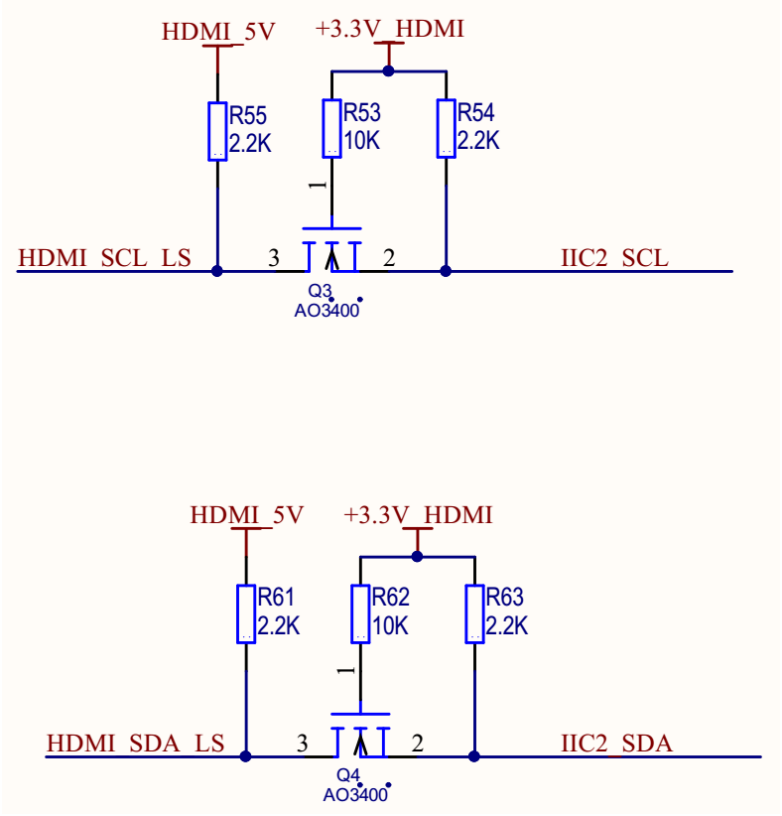


图 3.2.26 HDMI 的 IIC 电平转换电路

其中，IIC2_SCL 和 IIC2_SDA 连接到 ZYNQ PL 侧的 IO 口上，HDMI_SCL_LS 和 HDMI_SDA_LS 连接到 HDMI 插座上。

当总线空闲即 IIC2_SCL 和 IIC2_SDA 都被 ZYNQ 驱动为高电平时，MOS 管截止，HDMI_SCL_LS 和 HDMI_SDA_LS 也被上拉电阻 R55 和 R61 上拉到+5V 的高电平。

当 IIC2_SCL 或 IIC2_SDA 被 ZYNQ 驱动为低电平时，MOS 管导通，此时 HDMI_SCL_LS 或 HDMI_SDA_LS 也被下拉到了 GND。

若 HDMI_SDA_LS 需要向 ZYNQ 发送低电平时，HDMI_SDA_LS 为低，MOS 管内部二极管导通 IIC2_SDA 被下拉到了低电平。以此实现了双向电平转换的功能。

3.2.17 **USB 2.0 接口**

启明星底板板载了 1 个连接 PS 端的 USB2.0 收发器芯片，USB2.0 收发器采用的是一个 1.8V 的，高速且支持 ULPI 标准接口的 USB3320C 芯片，ZYNQ 的 USB 总线接口和 USB3320C 收发器相连接，实现高速的 USB2.0 Host 模式的数据通信。另外还外扩了一个 SLAVE 模式的接口。USB HOST 口采用侧立式的 USB 接口(USB Type A)，USB HOST 口采用迷你 USB 接口(Mini USB)。

USB3320C 与 ZYNQ 之间的连接框图如下图所示：

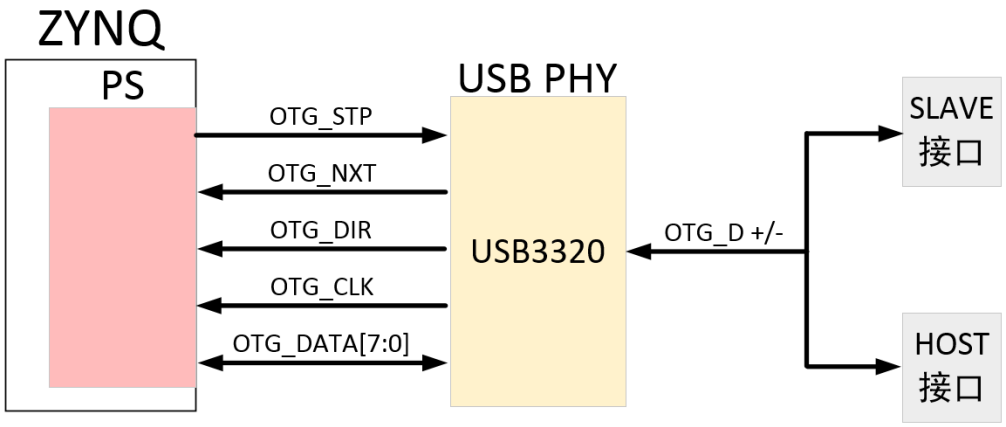


图 3.2.27 USB 连接框图

其电路原理图如下图所示：

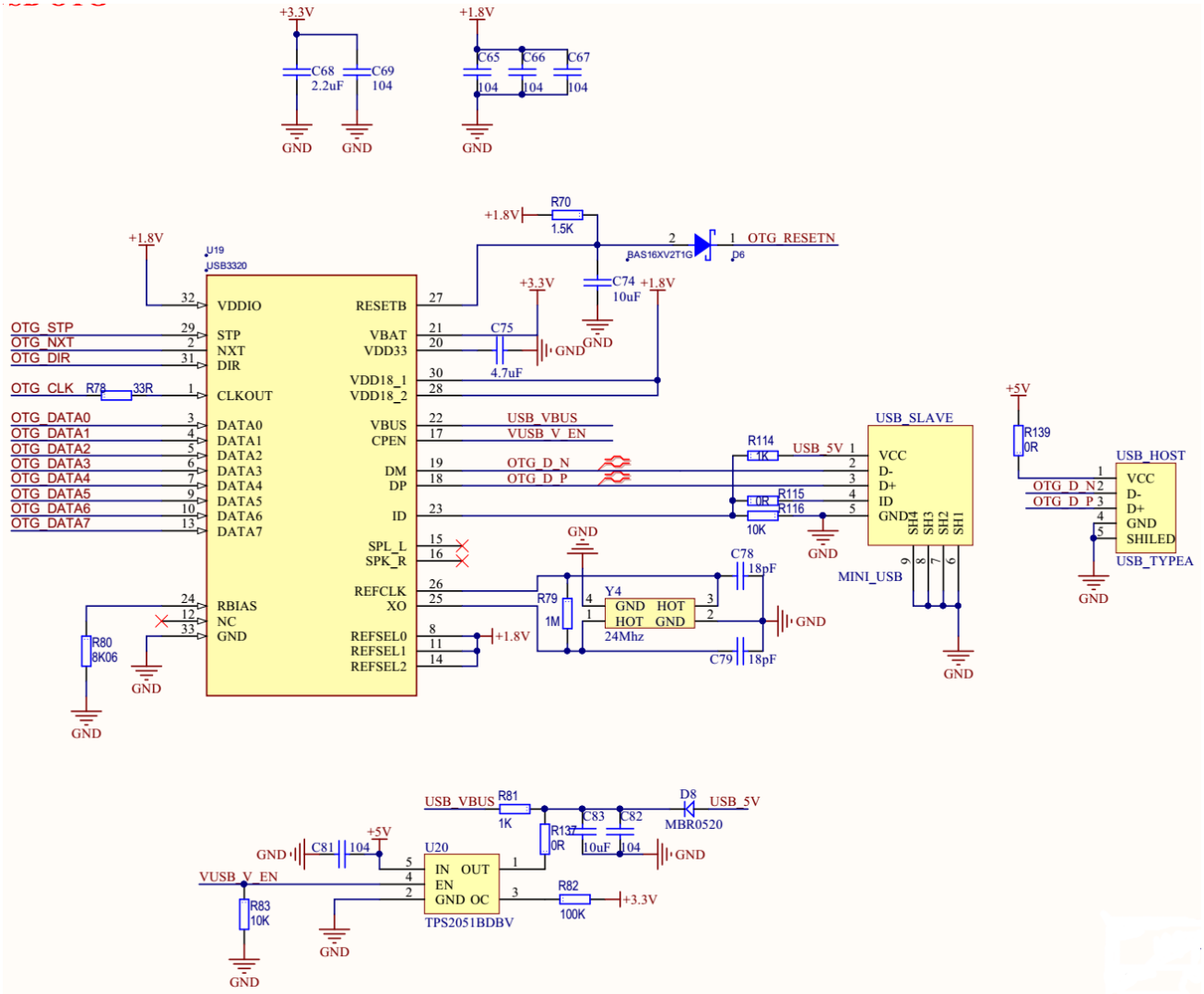


图 3.2.28 USB 接口原理图

USB2.0 PHY 芯片 USB3320 工作在 1.8V 电压下，与 ZYNQ PS BANK 501 的 MIO 之间通过 ULPI 标准高速接口来进行通信。USB 总线接口差分信号(OTG_D P/N)外接到了一个 USB HOST 接口和一个 USB

SLAVE 接口上。

HOST 接口可用来连接不同的 USB SLAVE 设备，可以用来连接鼠标、键盘等不同的设备。SLAVE 模式的接口，可以用来连接 HOST 设备，以满足特定的应用需求。

3.2.18 Micro SD 卡接口

启明星底板板载了一个 Micro SD（TF）卡接口，位于底板的背面。Micro SD 卡一般用于存储 ZYNQ 芯片的 BOOT 程序，Linux 操作系统内核，文件系统以及其它的用户数据文件。

下图是其与 ZYNQ 之间的连接框图：

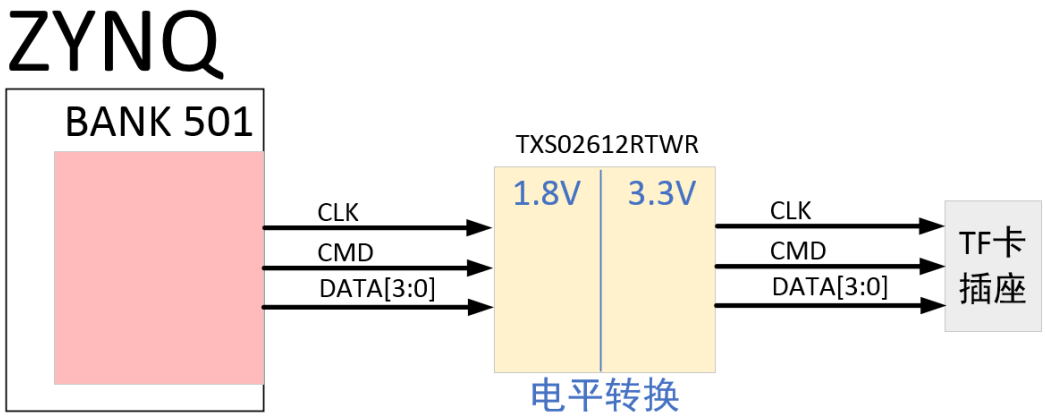


图 3.2.29 Micro SD 连接框图

其电路原理图如下图所示：

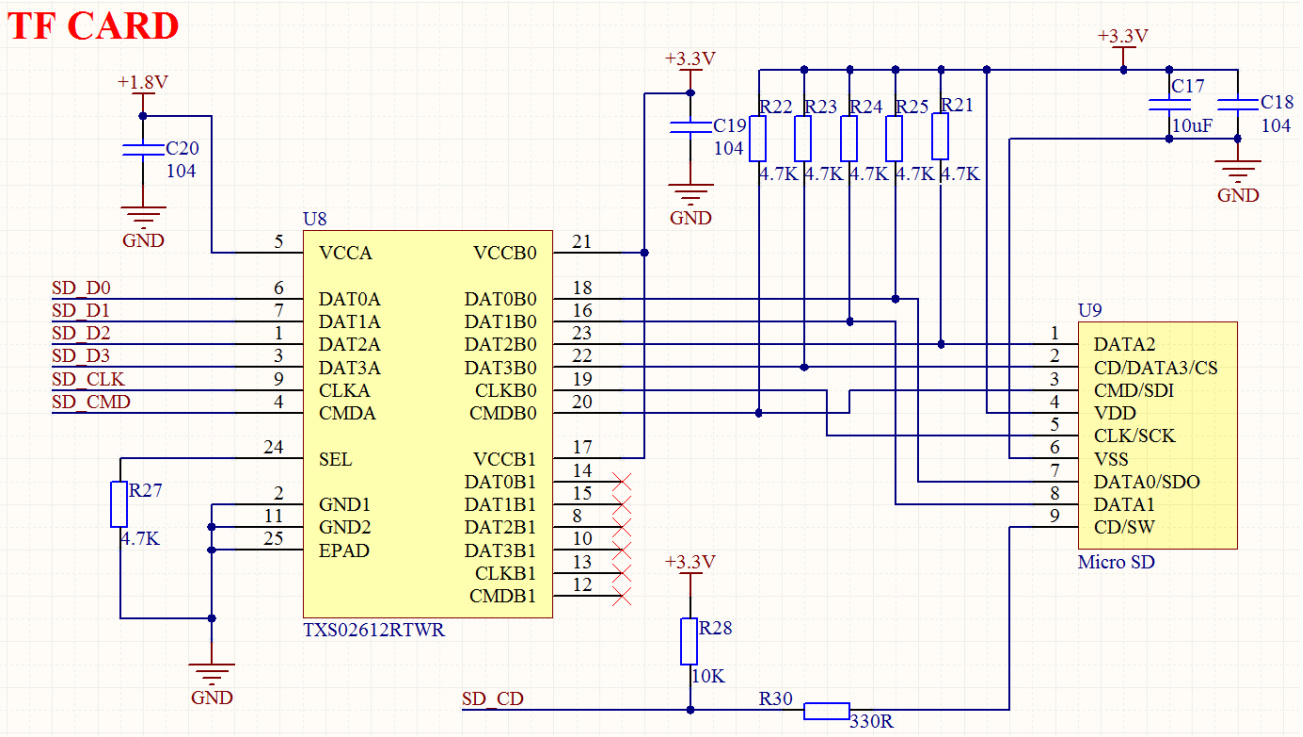


图 3.2.30 Micro SD 接口

上图中的 SD_CD 信号用于 Micro Card 的热插拔检测，当插上 Micro Card 时，CD 信号会被拉低。SD_CD 信号连接到了 IO BANK500 上的 MIO 10，该 BANK IO 为 3.3V，因此该引脚的上拉电平为 3.3V。由于在启明星开发板中，ZYNQ PS 端连接到 Micro SD 的 IO 是位于 BANK501 上面，BANK501 的供电电压是 1.8V，

而 Micro SD 的工作电压是 3.3V，所以要使用电平转换芯片 TXS02612RTWR 将 3.3V 的电平转换为 1.8V 的电平。

3.2.19 IO 扩展口

启明星底板板载两个 40PIN 的 IO 的扩展口，其原理图如下图所示：

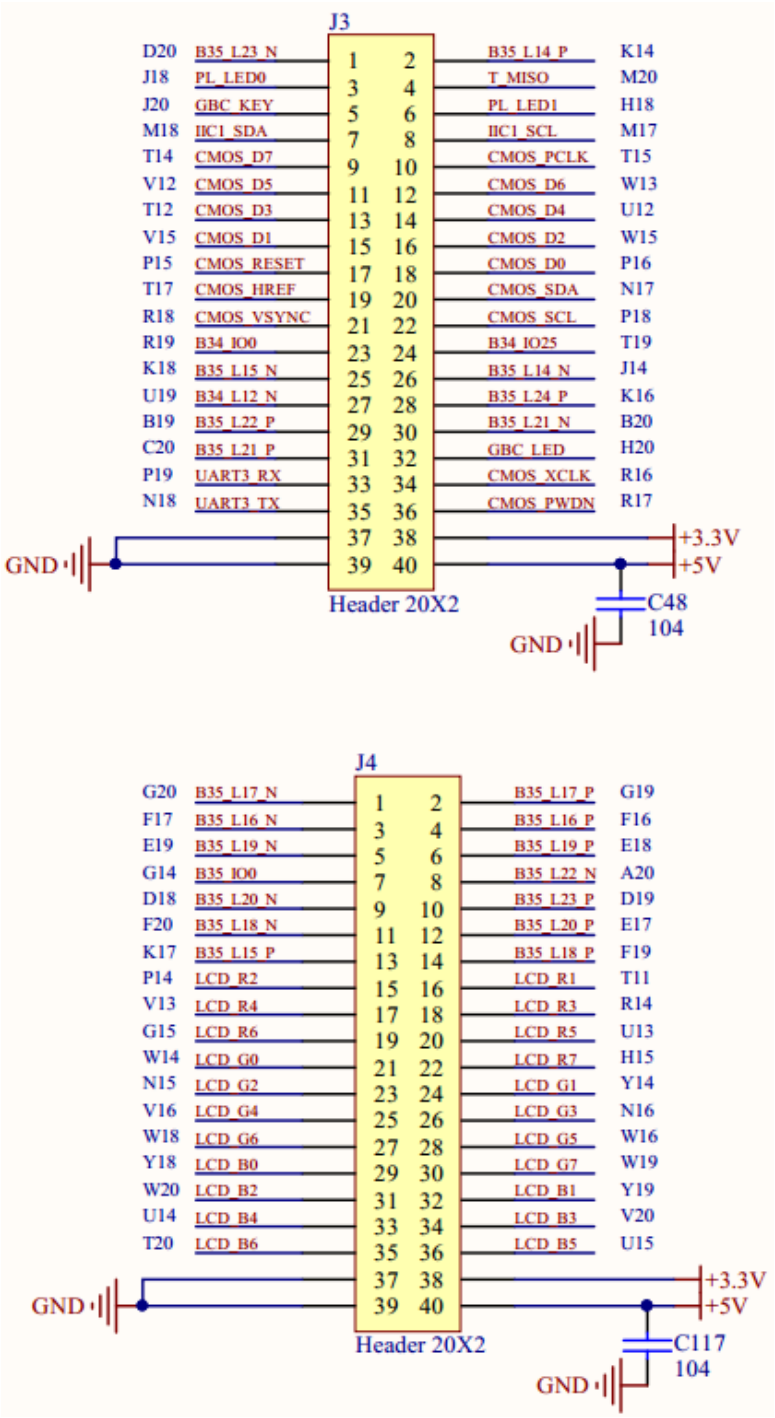


图 3.2.31 扩展口

每一个扩展口排针规格是 2*20 Pin，其中包括 36 个 IO 口、1 个+3.3V、1 个+5V、2 个 GND。它可以用来连接不同的功能模块，例如，正点原子开发的高速 AD/DA 模块、双目摄像头模块等。

3.3 开发板核心板原理图详解

3.3.1 核心板电源

ZYNQ-7020/ZYNQ-7010 核心板的电源来自底板转接板 IO 上的 5V 电源引脚，核心板电源拓扑结构如下图所示：

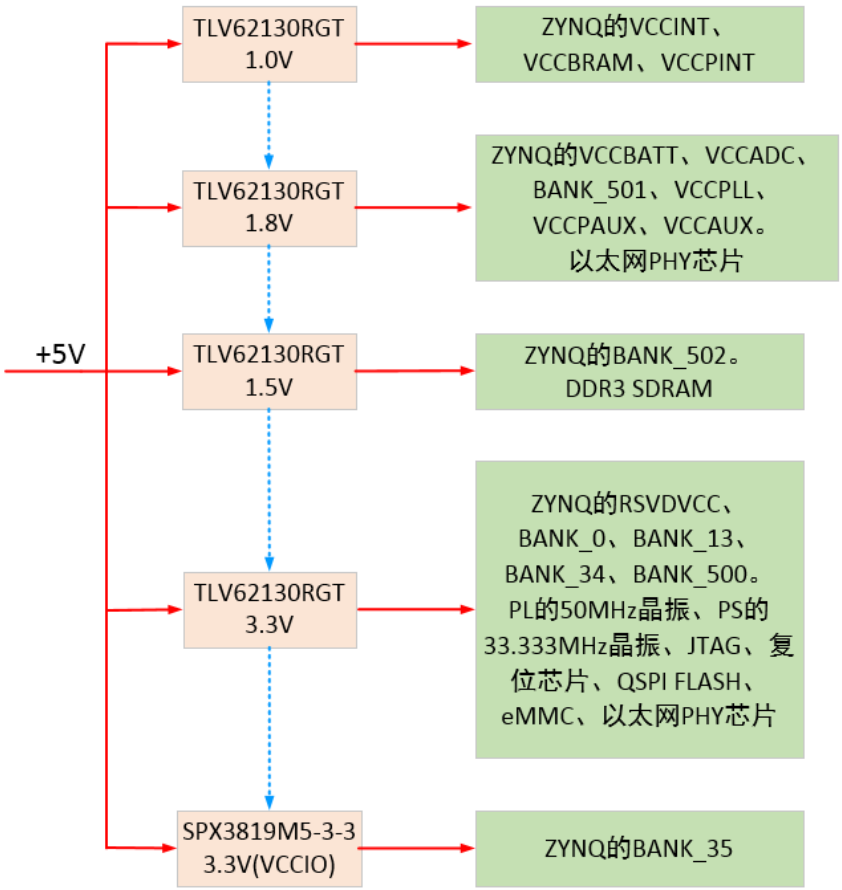


图 3.3.1 核心板电源拓扑

与底板相比，核心板需要的电源数量更多，包括 1.0V、1.8V、1.5V、3.3V 以及 VCCO 3.3V。DC-DC 芯片 TLV62130RGT 负责将+5V 电压转换为 1.0V、1.8V、1.5V、3.3V，LDO 芯片 SPX3819M5-3-3 负责将+5V 电压转换为 3.3V，专门给 ZYNQ 的 BANK35 提供电源，如果大家想 BANK35 工作在其他电压，可以自行把 SPX3819M5-3-3 替换为期望电压值的 LDO 型号。

另外，由于 ZYNQ 的供电有上电顺序的要求，所以这些电源的产生顺序必须符合 ZYNQ 的上电顺序要求，图中的蓝色虚线表示这些电源的产生顺序。

ZYNQ 芯片的 PS 和 PL 都需要多组电源。如下表所示：

表 3.3.1 ZYNQ PS 和 PL 的电源

PS	VCCPINT	PS 的 1.0V 逻辑电源。
	VCCPAUX	PS 的 1.8V 辅助电源。
	VCCO_MIO0	PS MIO bank 500 的 1.8V-3.3V I/O 电源，启明星使用 3.3V。
	VCCO_MIO1	PS MIO bank 501 的 1.8V-3.3V I/O 电源，启明星使用 1.8V。

	VCCO_DDR	PS 1.2V-1.8V DDR I/O 电源，启明星使用 1.5V。
	VCCPLL	PS 的 1.8V PLL 电源。
PL	VCCAUX	1.8V 电源引脚，用于辅助电路。
	VCCINT	内部逻辑的 1.0V 核心电源电压。
	VCCO_#	每个 IO bank 的电源，启明星 BANK13、BANK34、BANK35 全部使用 3.3V。
	VCCBRAM	PL Block RAM 的 1.0V 电源引脚。
	VCCBATT_0	解密器密钥存储器的备用电源。不使用时，应连接到相应的 VCC 或 GND。
	RSVDVCC[3:1]	保留引脚，必须连接到 VCCO_0。
独立电源	VCCADC	XADC 的供电引脚

PS 和 PL 的供电完全独立，两者之间没有任何上电顺序的要求，PS 电源（VCCPINT，VCCPAUX，VCCPLL，VCCO_DDR，VCCO_MIO0 和 VCCO_MIO1）可在任何 PL 电源之前或之后上电。但是 PS 和 PL 各自内部的各个电源之间却有上电顺序的要求。

官方推荐的 PS 上电顺序为：VCCPINT --> VCCPAUX 和 VCCPLL --> PS_VCCO(VCCO_MIO0、VCCO_MIO1、VCCO_DDR)。官方推荐的 PL 上电顺序为：VCCINT --> VCCBRAM --> VCCAUX --> VCCO，如果 VCCINT 和 VCCBRAM 具有相同的电源电压，则两者可以由相同的电源供电并同时供电。VCCADC 是 XADC 的独立供电引脚，与上述几种电源之间没有上电顺序的要求。

核心板的供电电路原理图如下图所示：

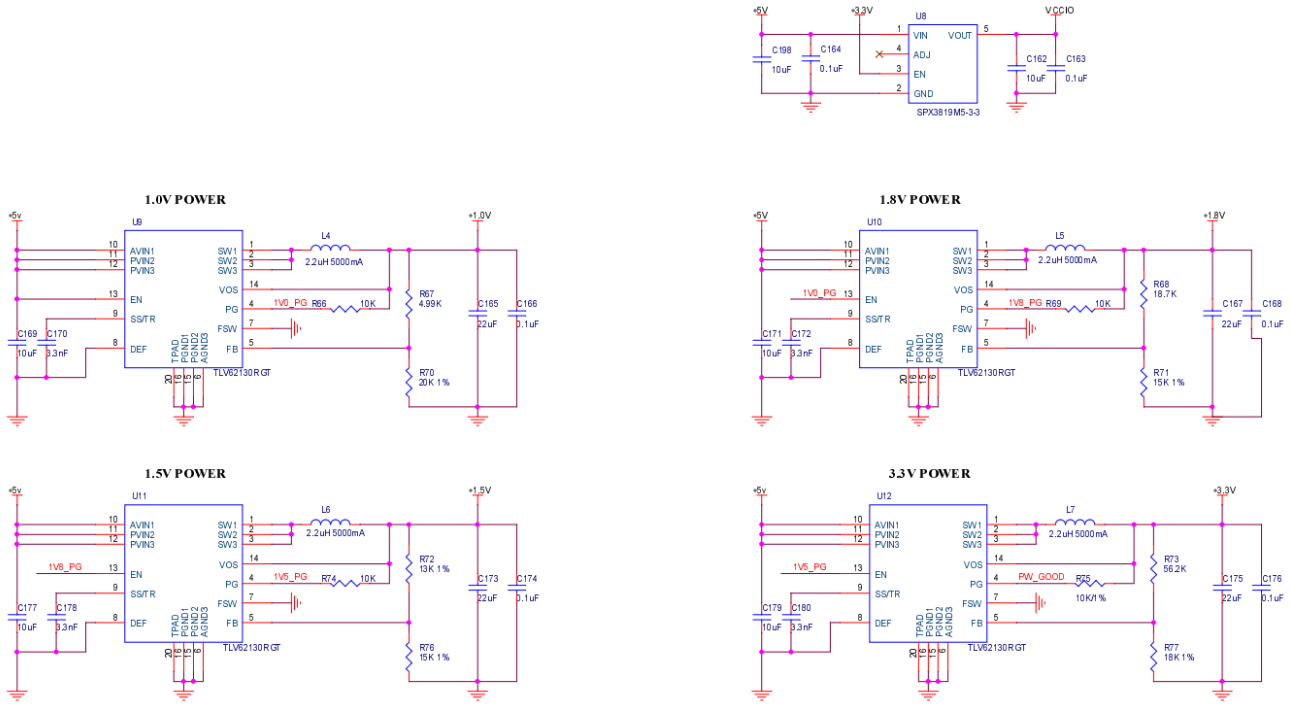


图 3.3.2 核心板供电电路原理图

其中，各个 DC-DC 芯片或 LDO 芯片的电源输入均是+5V，输出为所需的各路电源电压。各路电源电压之间的产生顺序是通过级联各个芯片的“EN”引脚来实现的，即，第一个产生的电源电压连接到第二个电源芯片的“EN”引脚上，这样，只有在第一个电源电压稳定输出之后，第二个电源电压才会开始产生。以此类推，直到产生最后一路电源电压。

另外，核心板上还具有一个电源指示灯，其原理图如下图所示：

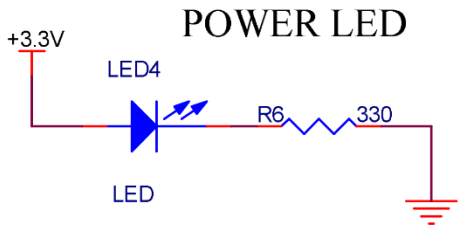


图 3.3.3 核心板电源指示灯

它连接到了 3.3V 电源上，可以通过核心板的电源指示灯来判断核心板供电是否正常。

3.3.2 ZYNQ 主控芯片

ZYNQ7020 核心板主控芯片采用 Xilinx 公司的 ZYNQ7000 系列的 XC7Z020，具体型号为 XC7Z020CLG400-2。ZYNQ 分为 PS（Processing System）和 PL（Programmable Logic）两部分。

PL 部分拥有 85K 个逻辑单元、4.9Mbits 的嵌入式存储资源、220 个 DSP 单元、4 个时钟管理单元（CMT）、16 个全局时钟网络、6 个用户 I/O BANK 和最大 253 个用户 I/O，是一款非常具有性价比的芯片。

ZYNQ7010 核心板主控芯片采用 Xilinx 公司的 ZYNQ7000 系列的 XC7Z010，具体型号为 XC7Z010CLG400-1。ZYNQ 分为 PS（Processing System）和 PL（Programmable Logic）两部分。

PL 部分拥有 28K 个逻辑单元、2.1Mbits 的嵌入式存储资源、80 个 DSP 单元、2 个时钟管理单元（CMT）、16 个全局时钟网络、5 个用户 I/O BANK 和最大 228 个用户 I/O，是一款非常具有性价比的芯片。

XC7Z020 和 XC7Z010 都集成了两个 Cortex-A9 处理器，AMBA®互连，内部存储器，外部存储器接口和外设。这些外设主要包括 USB 总线接口，以太网接口，SD/SDIO 接口，I2C 总线接口，CAN 总线接口，UART 接口，GPIO 等。

ZYNQ 芯片系统框图如下图所示：

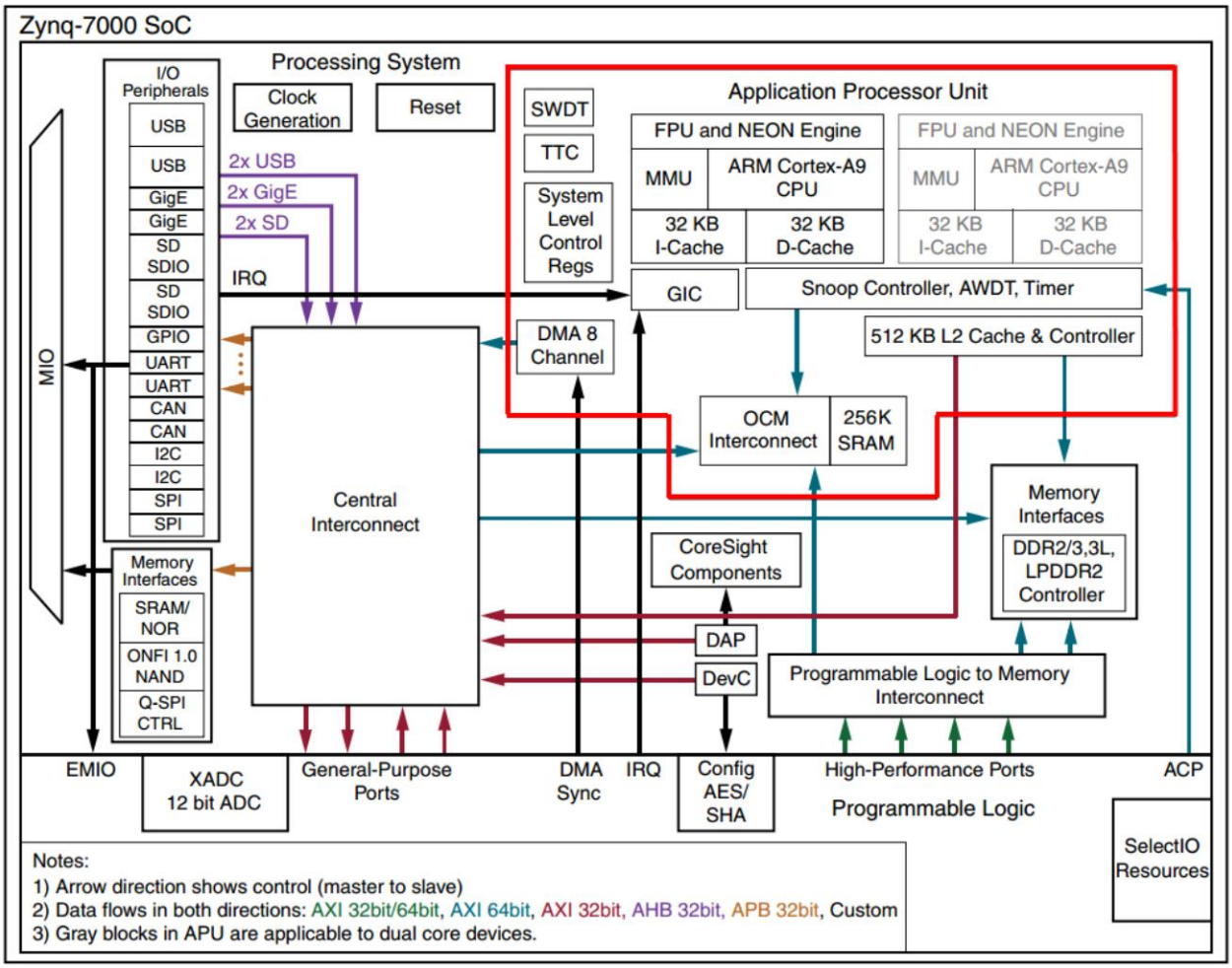


图 3.3.4 ZYNQ 芯片系统框图

PS 系统部分的主要参数如下：

- 基于 ARM 双核 CortexA9 的应用处理器，基于 ARM-v7 架构，高达 766Mhz（XC7Z020）/666Mhz（XC7Z010）核心频率
- 每个 CPU 32KB 1 级指令和数据缓存，2 个 CPU 共享 512KB 2 级缓存
- 集成片上 Boot ROM 和 256KB 片内 RAM
- 集成外部存储接口，支持 16/32 bit DDR2、DDR3 接口
- 支持两个千兆网控制器：支持 GMII，RGMII，SGMII 三种接口
- 支持两个 USB2.0 OTG 接口，每个最多支持 12 节点
- 支持两个 CAN2.0B 总线接口
- 支持两个 SD 卡、SDIO、MMC 兼容控制器
- 支持 2 个 SPI，2 个 UART，2 个 I2C 接口
- 4 组 32bit GPIO，54 个（32+22）作为 PS 系统 IO，64 个连接到 PL

- PS 外设内部和 PS 外设到 PL 外设的高带宽连接

ZYNQ-7020 核心板 PL 逻辑部分的主要参数如下：

- 逻辑单元 Logic Cells：85K；
- 查找表 LUT：53,200
- 寄存器(flip-flops)：106,400
- 乘法器 18x25MACCs：220;
- Block RAM：4.9Mb；
- 两个 AD 转换器，可以测量片上电压、温度感应和高达 17 个外部差分输入通道，1Mbps 采样率

ZYNQ-7010 核心板 PL 逻辑部分的主要参数如下：

- 逻辑单元 Logic Cells：28K；
- 查找表 LUT：17,600
- 寄存器(flip-flops)：35,200
- 乘法器 18x25MACCs：80;
- Block RAM：2.1Mb；
- 两个 AD 转换器，可以测量片上电压、温度感应和高达 17 个外部差分输入通道，1Mbps 采样率

XC7Z020-2CLG400I 芯片的速度等级为-2，工业级，封装为 BGA400，引脚间距为 0.8mm。XC7Z010-1CLG400C 芯片的速度等级为-1，商业级，封装为 BGA400。ZYNQ7000 系列的具体的芯片型号定义如下图所示：

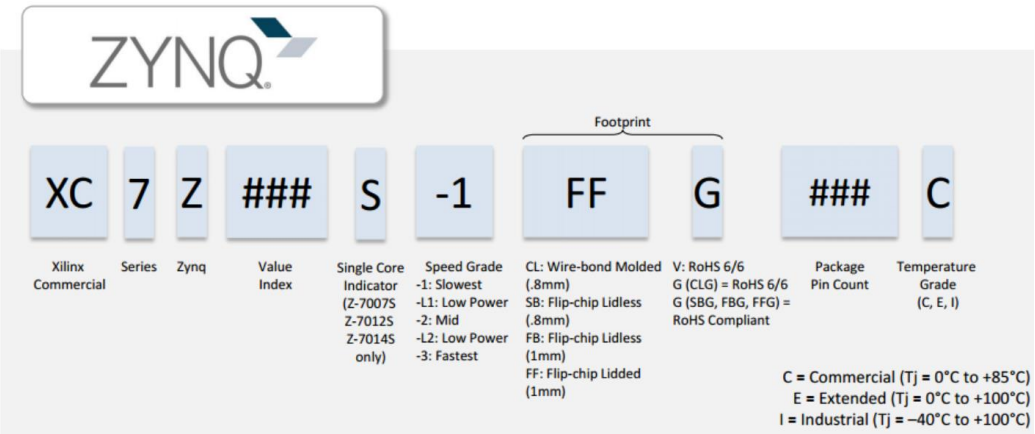


图 3.3.5 ZYNQ 芯片命名规则

另外当我们使用 BGA 封装的芯片以后，芯片引脚名称变为由字母+数字的形式，比如 A1，C2 等，因此我们在看原理图的时候，看到的字母+数字这种形式的，就是代表了 BGA 的引脚。

ZYNQ 芯片实物图如下图中间所示（以 ZYNQ-7020 核心板为例）：

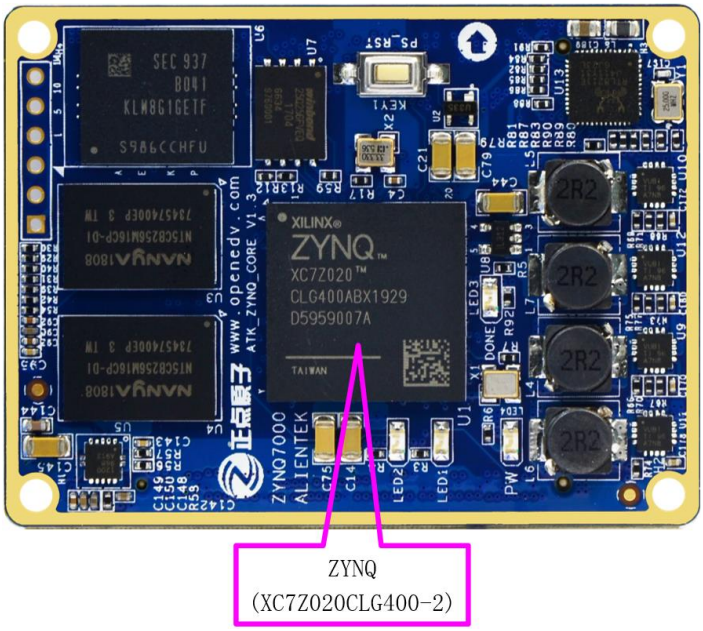


图 3.3.6 ZYNQ 芯片实物图

3.3.3 DDR3 SDRAM 存储器

ZYNQ-7020 核心板板载两片 4Gbit DDR3 内存，芯片型号为 NT5CB256M16EP-DI，总容量为 8Gbit（1GB）；

ZYNQ-7010 核心板板载两片 2Gbit DDR3 内存，芯片型号为 NT5CB128M16IP-DI，总容量为 4Gbit（512MB）；

DDR3 SDRAM 的最高运行速度可达 533MHz，由于 DDR3 为双倍数据采样，所以数据采样率可以达到 1066Mbps。它们连接到了 ZYNQ 的 IO BANK502 上，供电电压为 1.5V。下图是 DDR3 与 ZYNQ 之间的连接框图（以 ZYNQ-7020 核心板为例，ZYNQ-7010 核心板 DDR3 存储容量减半）：

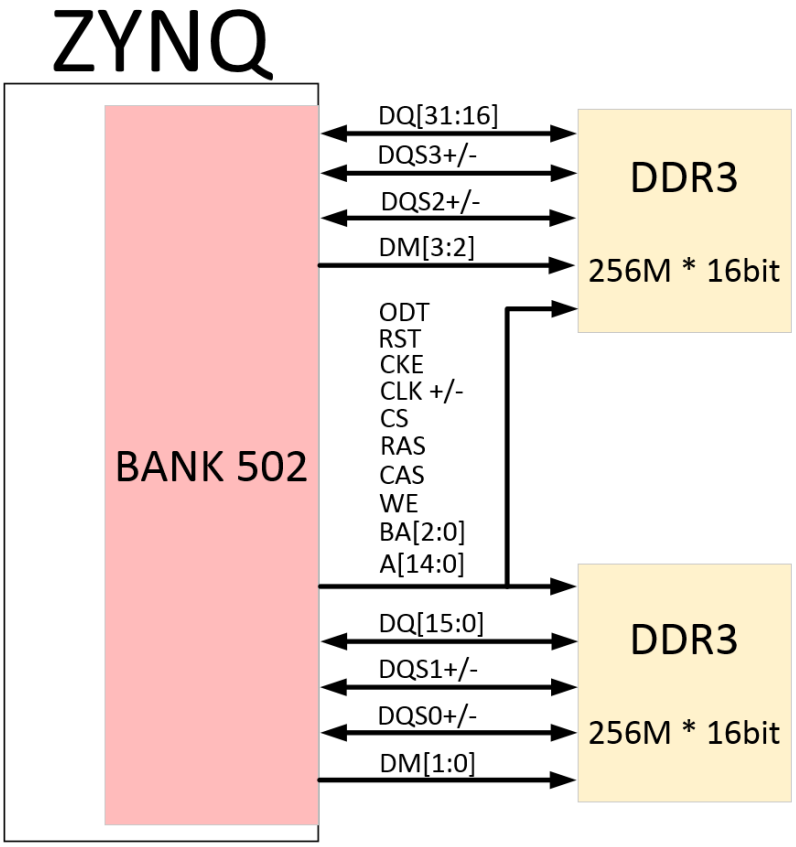


图 3.3.7 DDR3 SDRAM 连接框图

两片内存颗粒共享时钟信号、命令信号和地址总线，数据总线、数据选通信号和数据掩码信号各自分开。每片 DDR3 数据位宽为 16 位，两片 DDR3 的组成 32 位位宽的 DDR3 系统，其最大 IO 时钟频率为 533MHz，对应的等效数据传输频率为 1066MHz，两个 DDR3 颗粒提供的最大物理带宽为 1066MHz * 32bit*0.9 = 30.7Gbit/s，其中 0.9 是 DDR3 刷新造成的性能损失后的参数。这么高的存储带宽使得启明星能够轻松应对各种大内存和高带宽需求场景，比如普通图片存储、摄像头图像数据存储、录音数据存储等。

此外，它们还用来作为 PS 端处理器的运行内存。由于 DDR3 是 PS 部分的存储接口，因此 PL 逻辑需要通过 AXI 接口访问 DDR3。

其详细的原理图如图 3.3.8 和图 3.3.9 所示：

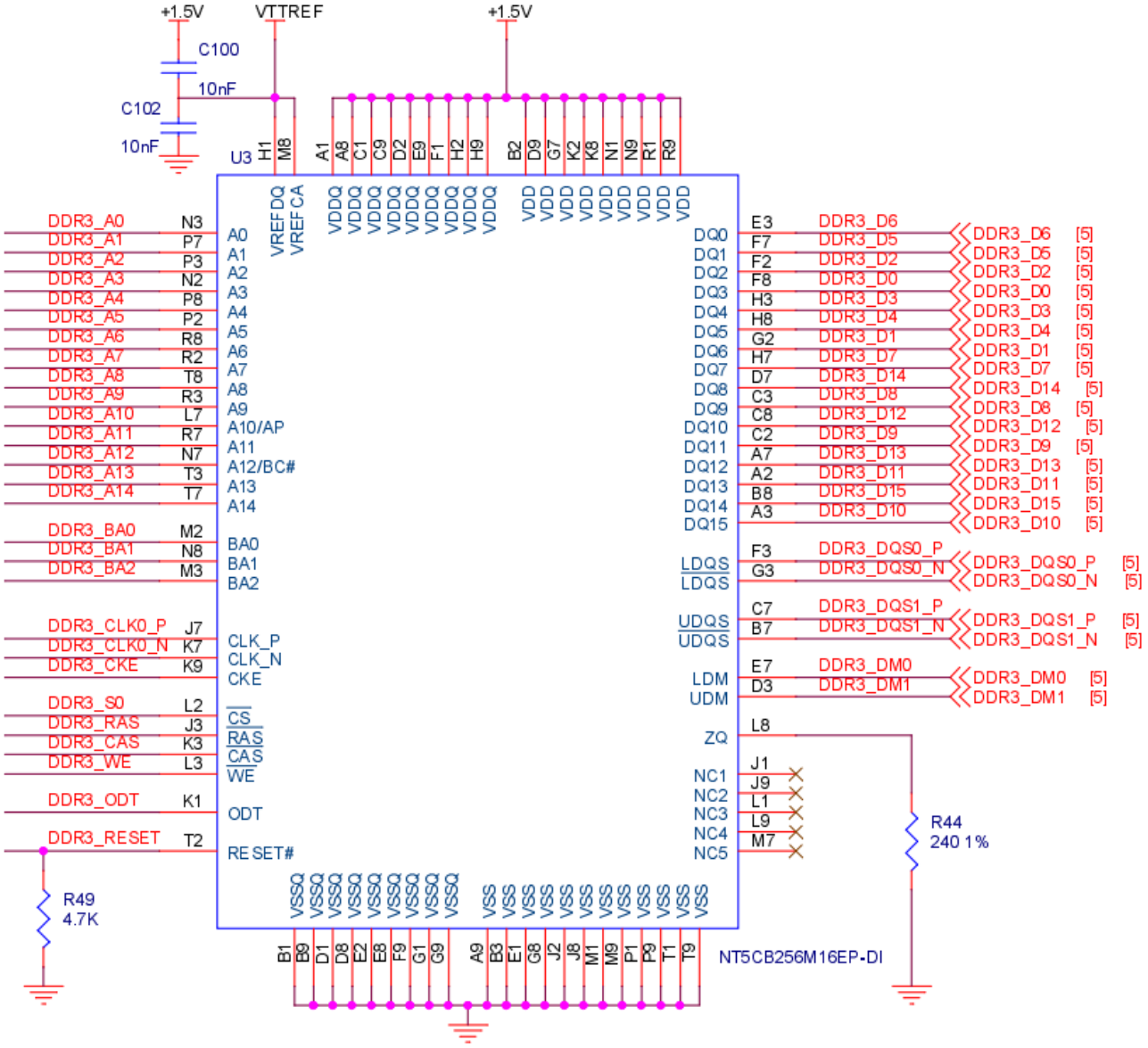


图 3.3.8 DDR3 SDRAM1

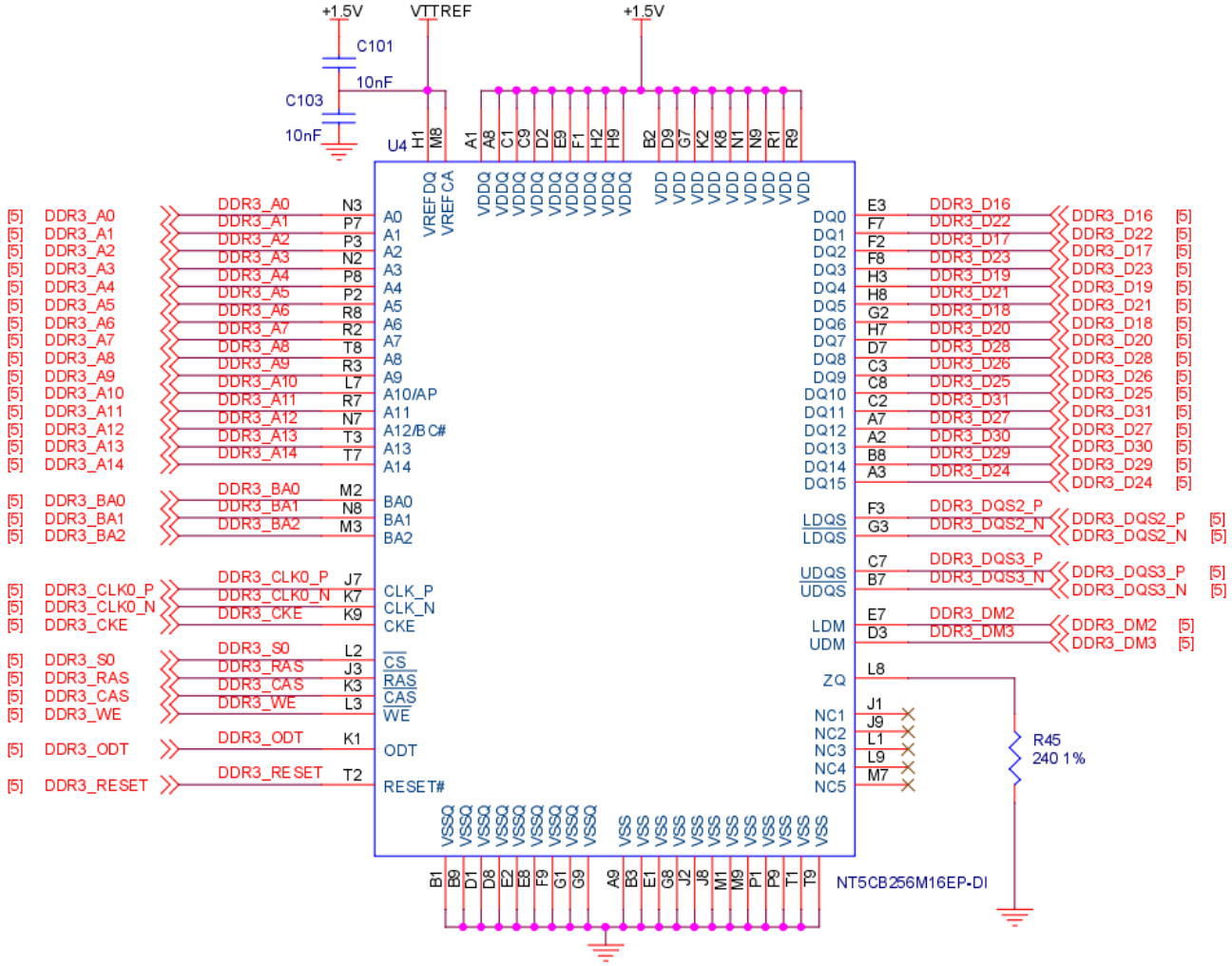


图 3.3.9 DDR3 SDRAM2

另外，DDR3 由于速度高且是双倍速率采样，所以硬件设计时需要严格考虑信号完整性，核心板在原理图设计、PCB 布线和 PCB 加工时候就充分考虑了匹配电阻/终端电阻,走线阻抗控制，走线等长控制，保证 DDR3 的高速稳定的工作。

3.3.4 6-Pin JTAG 接口

简化版的 JTAG 接口与底板的 14-Pin JTAG 接口是一体的，硬件电路是连通的，用于在单独调试核心板时使用，如下图所示：

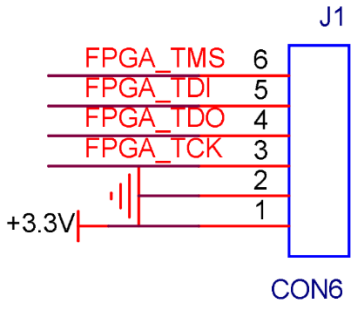


图 3.3.10 核心板 6-Pin JTAG 接口

JTAG 接口由 TMS、TDI、TDO、TCK、电源和 GND 组成。如果想单独使用核心板 6PIN JTAG 接口，可以焊接一个 6PIN 的 2.54MM 间距的排针，然后通过两个都是母头的杜邦线连接到下载器对应接口上面。

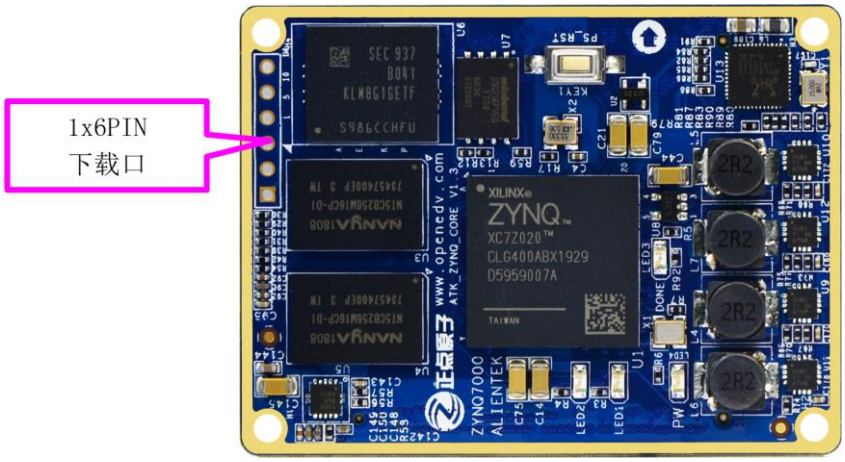


图 3.3.11 核心板 6-Pin JTAG 实物图

6Pin 下载口实物图如上图所示，从上到下引脚定义依次 TMS、TDI、TDO、TCK、GND 和 3.3V。

3.3.5 PS 复位按键

核心板板载一个 PS 的复位按键，它连接到了 PS 端的复位逻辑，按下后，PS 端将重新从上电后的状态开始运行。其原理图如下图所示：

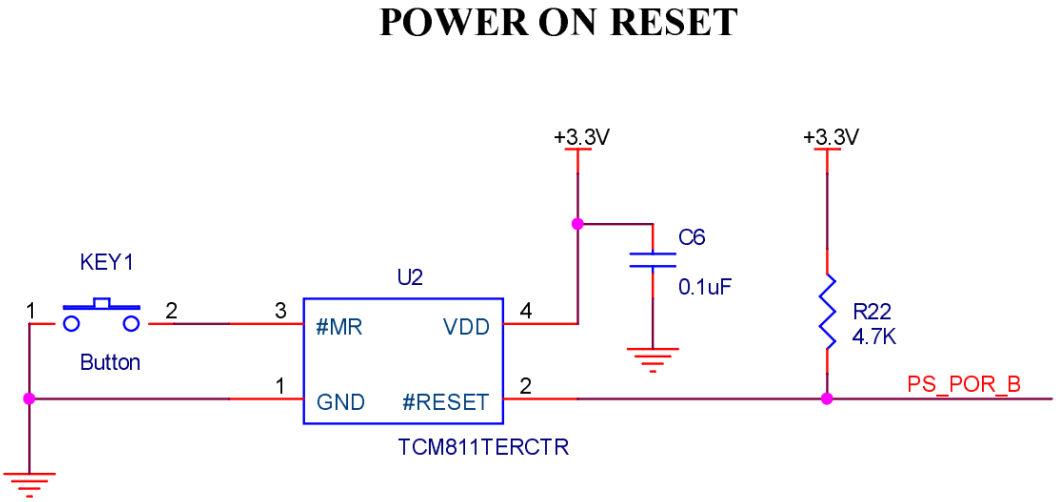


图 3.3.12 PS 端复位电路

PS 端复位电路采用了专用复位芯片 TCM811TERCTR，用于提供稳定可靠、无毛刺的复位信号。

3.3.6 PL LED

核心板板载一个 PL 控制的 LED，可以通过 Verilog 编程来控制其状态，逻辑输出为 1 时 LED 发光。这个 PL LED 灯的点亮状态可以作为程序运行状态的指示，方便对代码进行调试。其原理图如下图所示：

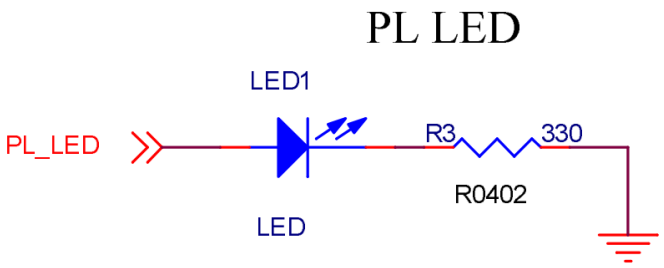


图 3.3.13 PL 的 LED

3.3.7 PS LED

核心板板载一个 PS 控制的 LED，可以通过 C 语言编程来控制其状态，逻辑输出为 1 时 LED 发光。这个 PS LED 灯的点亮状态可以作为程序运行状态的指示，方便对代码进行调试。其原理图如下图所示：

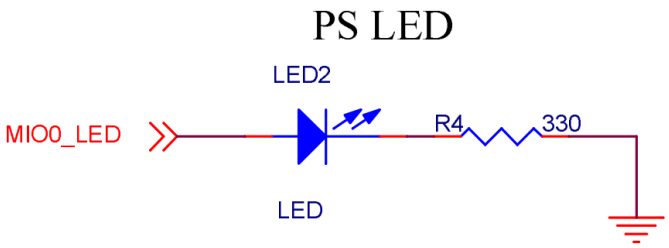


图 3.3.14 PS 的 LED

3.3.8 PL 时钟输入

核心板板载一个 50Mhz 的有源晶振，为 ZYNQ 的 PL 提供时钟，原理图如下图所示：

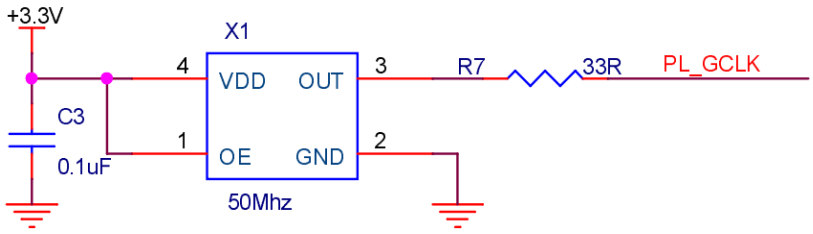


图 3.3.15 PL 的时钟晶振电路

晶振输出的 PL_GCLK（50Mhz）连接到 FPGA 的全局时钟(MRCC)，这个全局 GCLK 可以用来给 FPGA 用户逻辑提供时钟。当用户逻辑需要其它频率的时钟时可以通过 PL 内部中的MMCM/PLL 倍频或者分频来产生。

3.3.9 PS 时钟输入

核心板板载一个 33.333Mhz 的有源晶振，为 ZYNQ 的 PS 提供时钟，原理图如下图所示：

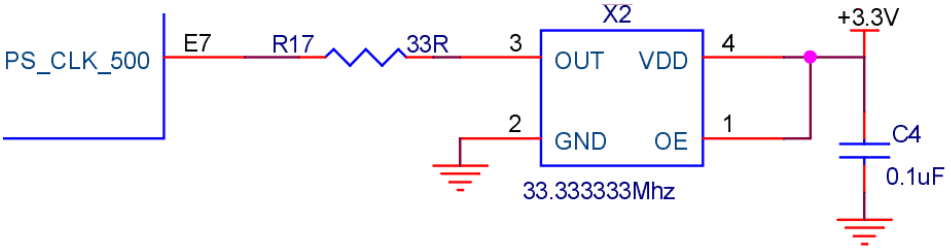


图 3.3.16 PS 的时钟晶振电路

PS 端的专用 PLL 会用此时钟来产生 PS 端所需的各种时钟频率。

3.3.10 PL 配置状态指示灯

核心板板载一颗 PL 配置状态指示灯，连接到了 PL 端的配置完成（DONE）信号，在 PL 端配置（下载程序）完成之后，该 LED 灯会被点亮。其原理图如下图所示：

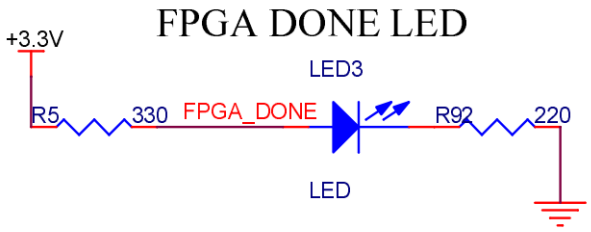


图 3.3.17 PL 端配置状态指示灯

3.3.11 PS 端千兆以太网

核心板板载一颗 PS 端千兆以太网 PHY（物理）芯片，型号为 RTL8211E-VL，实现了 10/100/1000M 以太网物理层功能。该 PHY 芯片的引脚连接到了底板上的 RJ45 接口上，能够满足高带宽通信的需求。

RTL8211E-VL 连接到了 PS 端的 BANK 501 上面，PHY 和 ZYNQ 芯片的连接框图如下图所示：

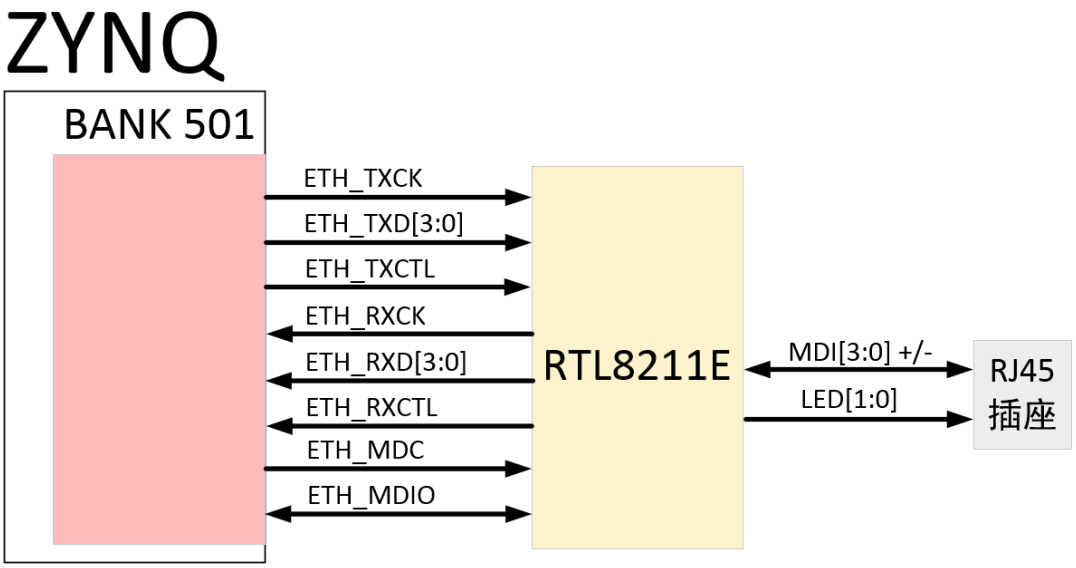


图 3.3.18 PS 端千兆以太网 PHY 芯片连接框图

其原理图如下图所示：

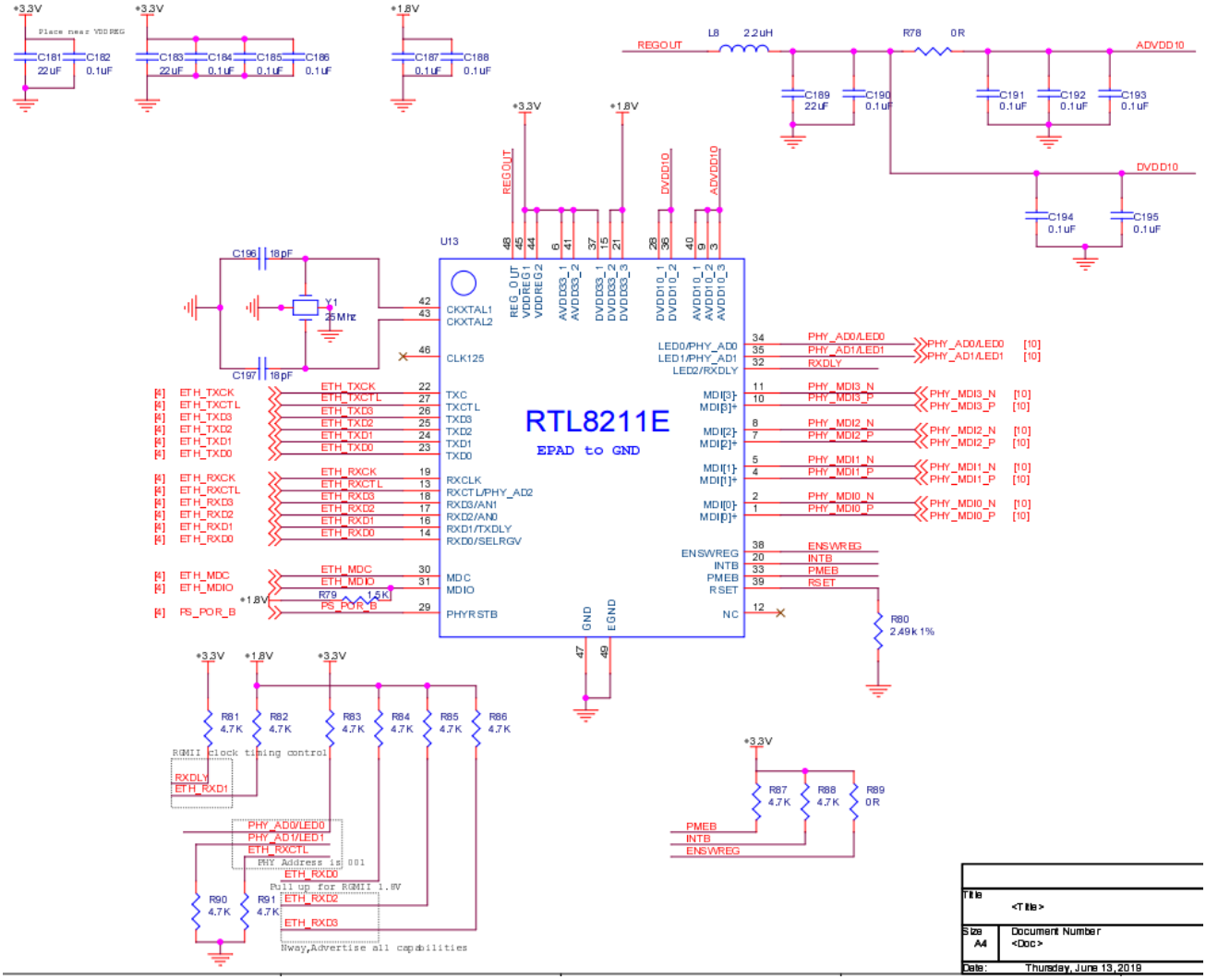


图 3.3.19 PS 端千兆以太网 PHY

RTL8211E 芯片支持 10/100/1000Mbps 网络传输速率，通过 RGMII 接口跟 ZYNQ PS 系统的 MAC 层进行数据通信，并支持通过 MDIO 总线进行 PHY 寄存器的管理。除此之外， RTL8211E 上电会检测一些特定 IO 引脚的电平状态，从而确定自己的工作模式，用于配置芯片的工作状态。RGMII 接口时序图如下图所示：

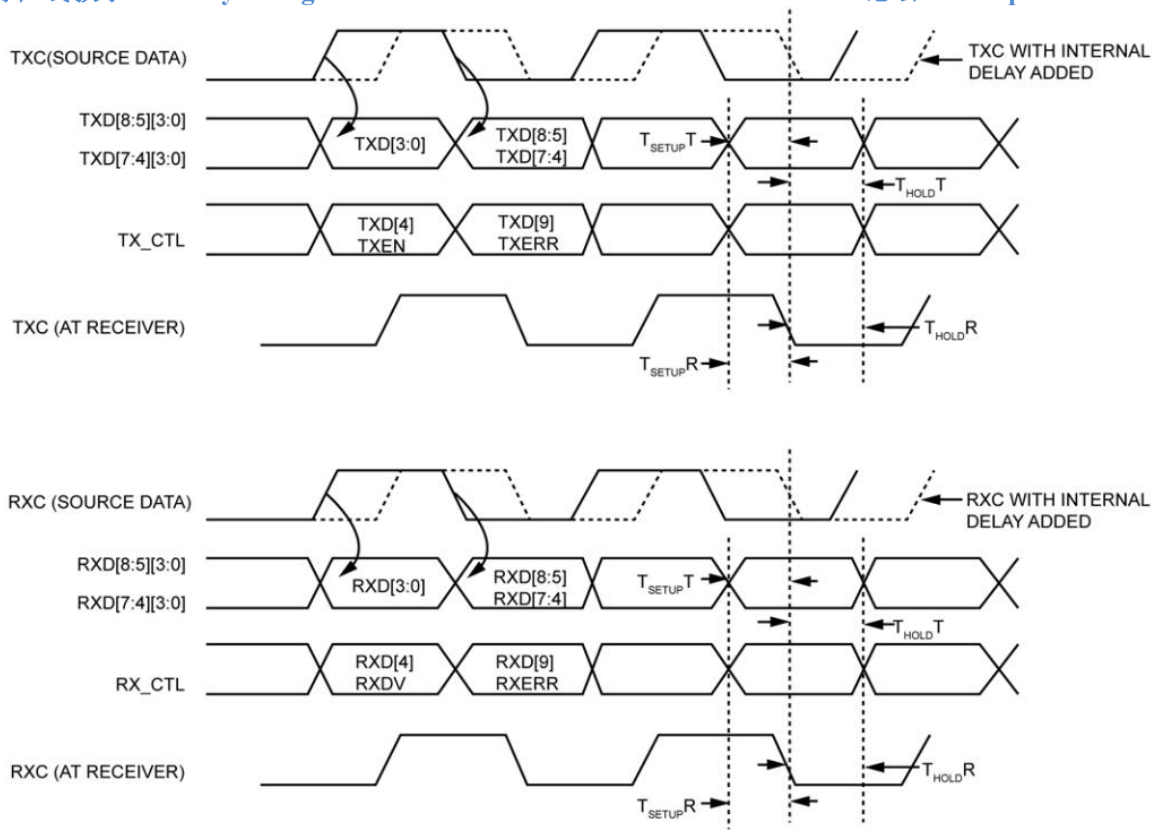


图 3.3.20 RGMII 接口时序图

RTL8211E 芯片支持 10/100/1000Mbps 网络传输速率，由通信双方所能达到的最高通信速率决定。当以太网 PHY 芯片通信速率为 1000Mbps 时，以太网接口时钟频率为 125Mhz，数据在时钟的上升沿和下降样被采样；当以太网 PHY 芯片通信速率为 100Mbps 时，以太网接口时钟频率为 25Mhz，数据在时钟的上升沿被采样；当以太网 PHY 芯片通信速率为 10Mbps 时，以太网接口时钟频率为 2.5Mhz，数据在时钟的上升沿被采样。

3.3.12 QSPI FLASH

核心板板载一颗 QSPI Flash 芯片，型号为 W25Q256FVEI，存储容量为 256Mbit（32M 字节），采用 SPI 协议和 FPGA 进行通信。QSPI Flash 可用于存储 ZYNQ 芯片的启动镜像数据，包括 PS 端的程序镜像和 PL 端的配置镜像，以保证 ZYNQ 在重新上电后仍能继续工作。

下图是其与 ZYNQ 之间的连接框图，QSPI 连接到了 BANK500 上面：

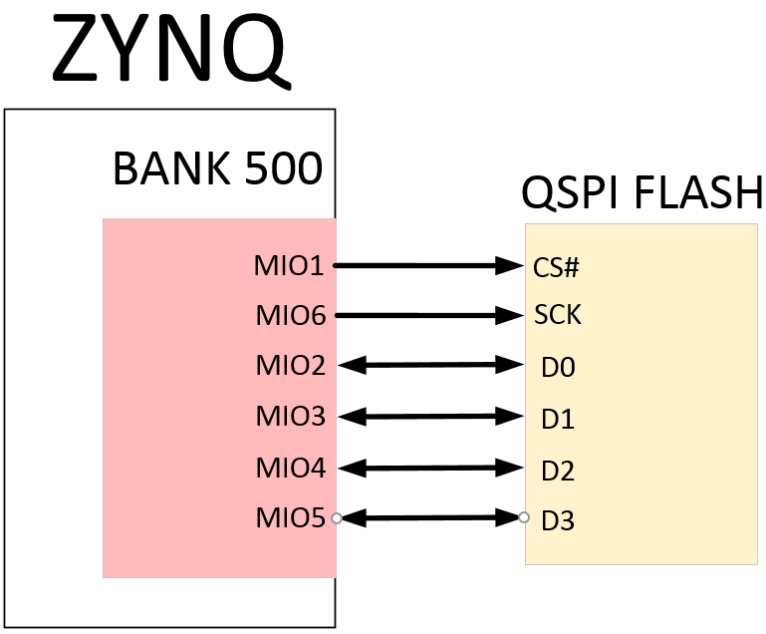


图 3.3.21 QSPI FLASH 芯片连接框图

下面是其详细的电路原理图，其中 CS 信号需要上拉到高电平：

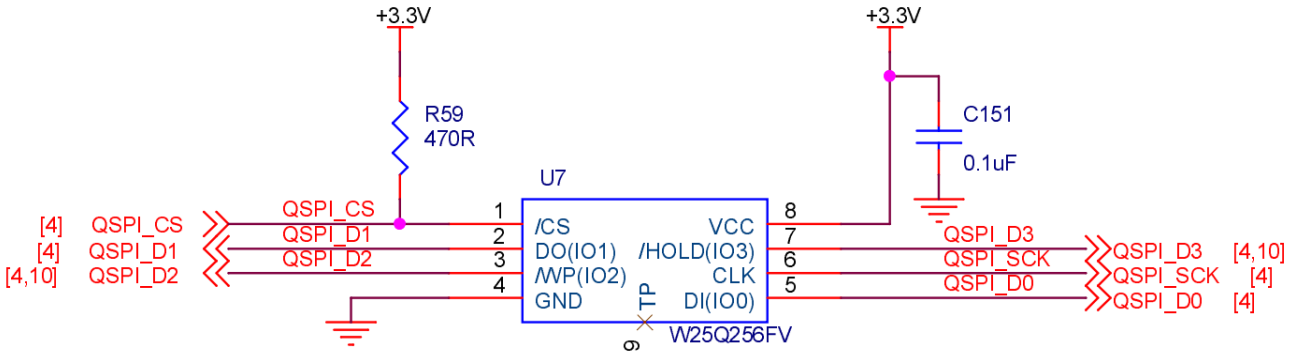


图 3.3.22 QSPI FLASH 芯片原理图

需要说明的是，QSPI_D2 和 QSPI_D3 也作为 ZYNQ 启动方式的引脚选择，这两个引脚连接到了开发板底板的拨码开关上。

3.3.13 eMMC

eMMC 是非易失性 NAND 存储器，俗称电子硬盘，核心板板载的 eMMC 芯片型号为 KLM8G1GETF，存储容量为 8GB，能够满足 PS 端大容量非易失性的存储需求，如存储 ARM 的应用程序、系统文件以及其它的用户数据文件。

ZYNQ 与 eMMC 之间的连接框图如下图所示：

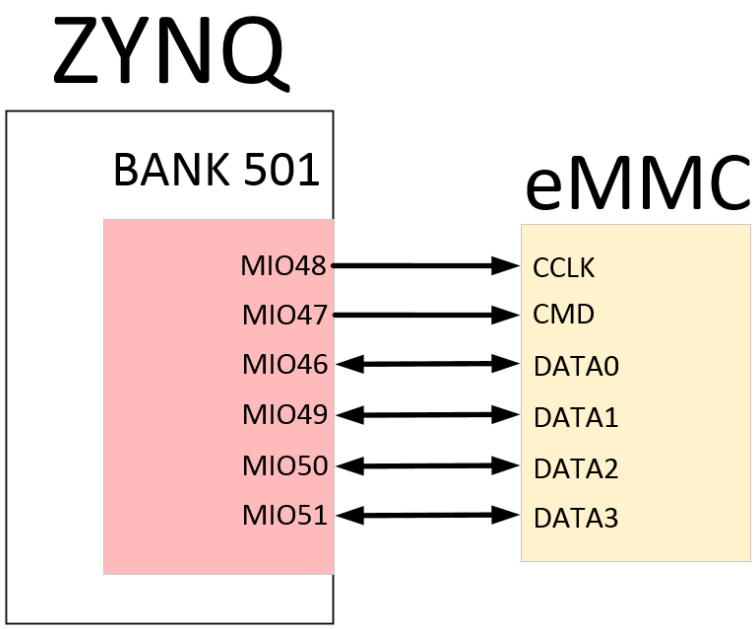


图 3.3.23 eMMC 芯片连接框图

eMMC FLASH 连接到 ZYNQ 芯片的 PS 部分 BANK501 的 MIO 口上，其原理图如下图所示：

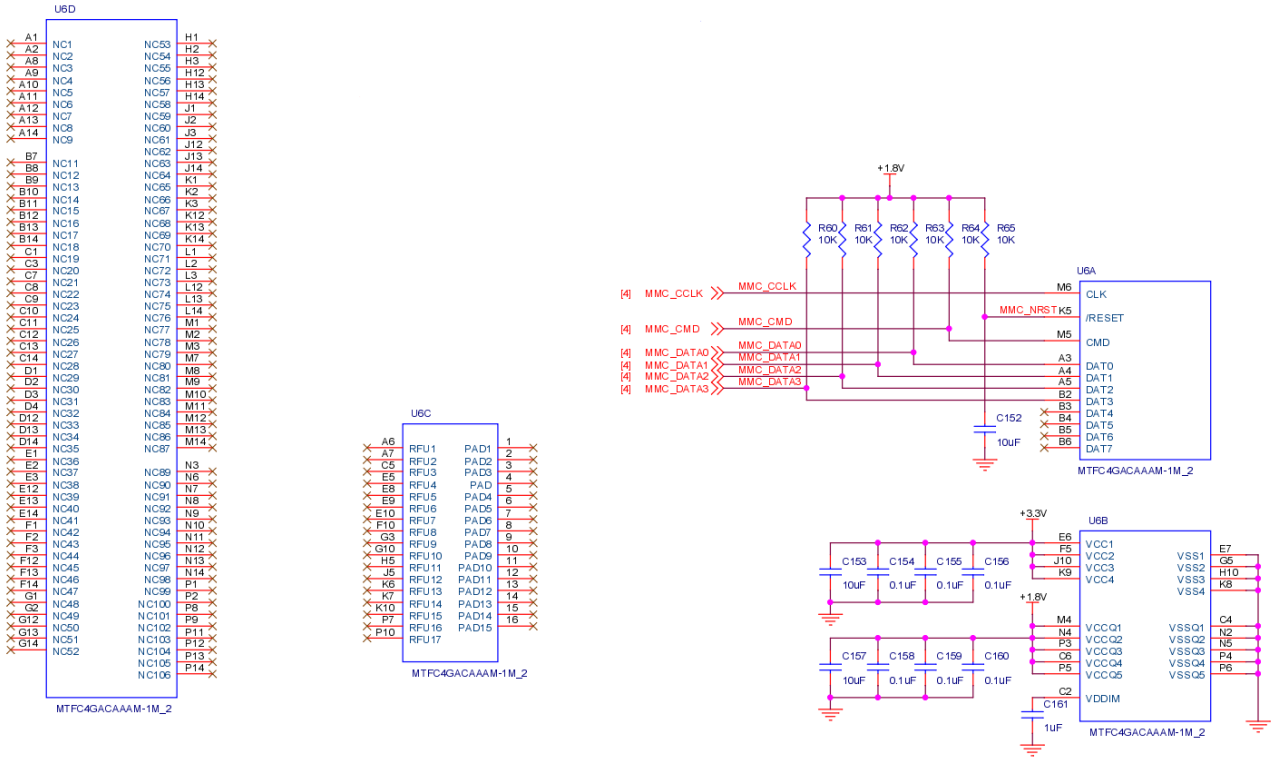


图 3.3.24 eMMC 芯片原理图

3.4 开发板使用注意事项

为了让大家更好的使用启明星 ZYNQ 开发板，我们在这里总结该开发板使用的时候尤其要注意的一些问题，希望大家在使用的时候多多注意，以减少不必要的问题。

1. USB 供电电流最大 500mA，且由于存在导线电阻，供到开发板的电压，一般都不会有 5V，如果使用了很多大负载外设，比如 4.3 寸屏、7 寸屏、摄像头模块等，那么可能引起 USB 供电不足，所以开发板如果连接大负载模块的朋友，或者同时用到多个模块的时候，建议大家使用一个电源适配器供电。

2. 当你想使用某个 IO 口用作其他用处的时候，请先看看开发板的原理图，该 IO 口是否有连接在开发板的某个外设上，如果有，该外设的这个信号是否会对你的使用造成干扰，先确定无干扰，再使用这个 IO。

3. 开发板上需要连接跳帽的地方比较多，大家在使用某个功能的时候，要先查查实现这个功能是否需要连接跳帽，以免浪费时间。

4. 当液晶显示白屏的时候，请先检查液晶模块是否插好（拔下来重新插试试），如果还不行，可以通过串口看看 LCD ID 是否正常，再做进一步的分析。

至此，本手册的实验平台启明星 FPGA 的硬件部分就介绍完了，了解了整个硬件对我们后面的学习会有很大帮助，有助于后面的管脚约束（分配），在编写程序的时候，可以事半功倍，希望大家细读！另外正点原子开发板的其他资料及教程更新，都可以在技术论坛 www.openedv.com 下载到，大家可以经常去这个论坛获取更新的信息。

3.5 ZYNQ 的学习方法

ZYNQ 作为目前热门的 SOC 类处理器，正在被越来越多的公司选择使用。学习 ZYNQ 的朋友也越来越多，初学者可能会认为 ZYNQ 很难学，以前只学过 51，或者甚至连 51 都没学过的，一看到 ZYNQ 就懵了。其实，万事开头难，只要掌握了方法，学好 ZYNQ，还是非常简单的，这里我们总结学习 ZYNQ 的几个要点：

1. 一款实用的开发板。

这个是实验的基础，有时候软件仿真通过了，在板上并不一定能跑起来，而且有个开发板在手，什么东西都可以直观的看到，效果不是仿真能比的。但开发板不宜多，多了的话连自己都不知道该学哪个了，觉得这个也还可以，那个也不错，那就这个学半天，那个学半天，结果学个四不像。倒不如从一而终，学完一个再学另外一个。

2. 掌握方法，勤学慎思。

ZYNQ 不是妖魔鬼怪，不要畏难，ZYNQ 的学习和普通单片机一样，基本方法就是：

3. 了解 ZYNQ 的基本结构。

学习 ZYNQ 之前需要先对 ZYNQ 的基本结构和其功能有个大概的了解，如 PS、AXI 总线、锁相环 PLL、FIFO 等。需要知道 PLL 是用来产生不同频率的时钟，如使用 WM8960（音频编解码芯片）时需要生成 12MHz 的时钟；FIFO 用于数据的缓存和异步时钟域数据的传递等。

4. 了解 Verilog HDL 基本语法

没有软件的硬件就如同行尸走肉一般。软件是硬件的灵魂，硬件是软件的舞台。好的软件设计才能发挥硬件的性能，而软件的精髓在于代码。学习 FPGA 也是这样，Verilog HDL 做为一种硬件描述语言，是对

数字电路的一种描述，而数字电路是并行工作的，因而在编写 Verilog HDL 时要有并行的思想，不同于软件设计语言，软件设计语言是由 CPU 统一进行处理，一条指令一条指令的串行运行，所以软件设计语言是基于串行的设计思想，因而在写 Verilog HDL 代码的时候要注意这种差别。另外对于 Verilog HDL 的基本语法是务必要掌握的，如一般常用的 module/endmodule、input/output/inout、wire/reg、begin/end、posedge/negedge、always/assign、if/else、case/default/endcase/parameter/localparam 等关键字要清楚它们的作用和区别。掌握了 Verilog HDL 的基本语法和 Verilog HDL 的并行设计思想后，会觉得 Verilog HDL 和 C 语言一样简单。

5. ZYNQ PS 的学习

学习 ZYNQ PS 首先要了解 C 语言的基本语法。然后了解 PS 端基本的系统框架和外设，例如 PS 端架构、片内互连、AXI 总线、DDR 控制器等等。最关键的是 PS 端的 C 程序。软件系统可以被认为是建立于基于硬件的系统上的一个栈，或者说是一系列层，从底至上依次是基础硬件系统（来自 Vivado 的自定义硬件）、板级支持包、Operating System、软件应用。

初学者可以多看看官方的文档和资料，对于 PS 端的软件，xilinx 提供了丰富的库函数，很多时候用户直接调用即可，读者要学会利用这些库函数。遇到问题时，读者可以借助 xilinx 的 SDK 开发环境中的各种调试功能来定位错误，以帮助解决问题。

6. 多思考，多动手。

所谓熟能生巧，先要熟，才能巧。如何熟悉？这就要靠大家自己动手，多多练习了，光看或说是没什么太多用的。只有在使用 ZYNQ 的过程中，才会一点点的熟悉，也只有动手实练，才能对 FPGA 有切实的感受。

熟悉了之后，就应该进一步思考，也就是所谓的巧了。我们提供了几十个例程，供大家学习，跟着例程走，无非就是熟悉 ZYNQ 的过程，只有进一步思考，才能更好的掌握 ZYNQ，也即所谓的举一反三。例程是死的，人是活的，所以，可以在例程的基础上，自由发挥，实现更多的其他功能，并总结规律，为以后的学习和使用打下坚实的基础，如此，方能信手拈来。

所以，学习一定要自己动手，光看视频，光看文档，是不行的。举个简单的例子，你看视频，教你如何煮饭，几分钟估计你就觉得学会了。实际上你可以自己测试下，是否真能煮好？机会总是留给有准备的人，只有平时多做准备，才可能抓住机会。

只要以上三点做好了，学习 ZYNQ 基本上就不会有什么太大问题了。如果遇到问题，可以在我们的技术论坛：开源电子网：www.openedv.com 提问，论坛 FPGA 板块有各种主题，很多疑问已经有网友提过了，所以可以在论坛先搜索一下，很多时候，就可以直接找到答案了。论坛是一个分享交流的好地方，是一个可以让大家互相学习，互相提高的平台，所以有时间，可以多上去看看。

第二篇　软件篇

上一篇，我们介绍了本手册的实验平台，本篇我们将详细介绍 Xilinx FPGA 的开发软件 Vivado Design Suite 的使用。通过该篇的学习，你将了解到一个完整的 FPGA 的开发流程和仿真过程。本篇将图文并茂的向大家介绍 Vivado 软件的使用，通过本篇的学习，希望大家能掌握 FPGA 的开发流程以及如何对编写的代码进行仿真，并能独立开始 FPGA 的编程和学习。

第四章　Vivado 软件的安装和使用

Vivado Design Suite 是 Xilinx 公司的综合性 FPGA 开发软件，可以完成从设计输入到硬件配置的完整 FPGA 设计流程。本章我们将学习如何安装 Vivado 软件以及 Vivado 软件的使用方法，为大家在接下来学习实战篇打下基础。

本章包括以下几个部分：

4.1　Vivado 软件的安装

4.2　Vivado 软件的使用

4.3　在线逻辑分析仪的使用

4.4　在 Vivado 中进行功能仿真

4.1 Vivado 软件的安装

Xilinx 公司每年都会对 Vivado 设计套件进行更新，各个版本之间除界面以及其它性能的优化之外，基本的使用功能都是一样的，我们光盘中提供的是相对稳定的 Vivado v18.3 版本，接下来我们安装 Vivado v18.3（以下简称 Vivado）版本的软件。

首先在启明星 ZYNQ 开发板工具盘(B 盘)→Vivado 文件夹下找到 Vivado 的安装包文件，文件列表如下图所示：

图 4.1.1 Vivado 安装包文件夹

将压缩包解压出来（**注意，解压目录的路径名称只能够包含字母、数字、下划线，否则安装程序有可能出问题**），为避免在安装过程中出错，在开始安装之前，请先关闭安全或杀毒软件。双击解压出来的文件夹下的“xsetup.exe”，开始安装 Vivado 软件，如下图所示：

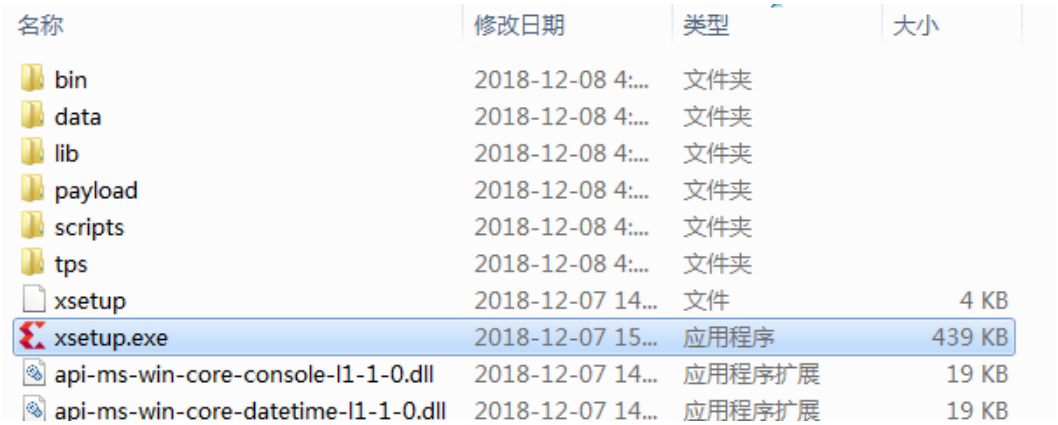


图 4.1.2 双击“xsetup.exe”

进入 Vivado 的安装引导页面，如下图所示：

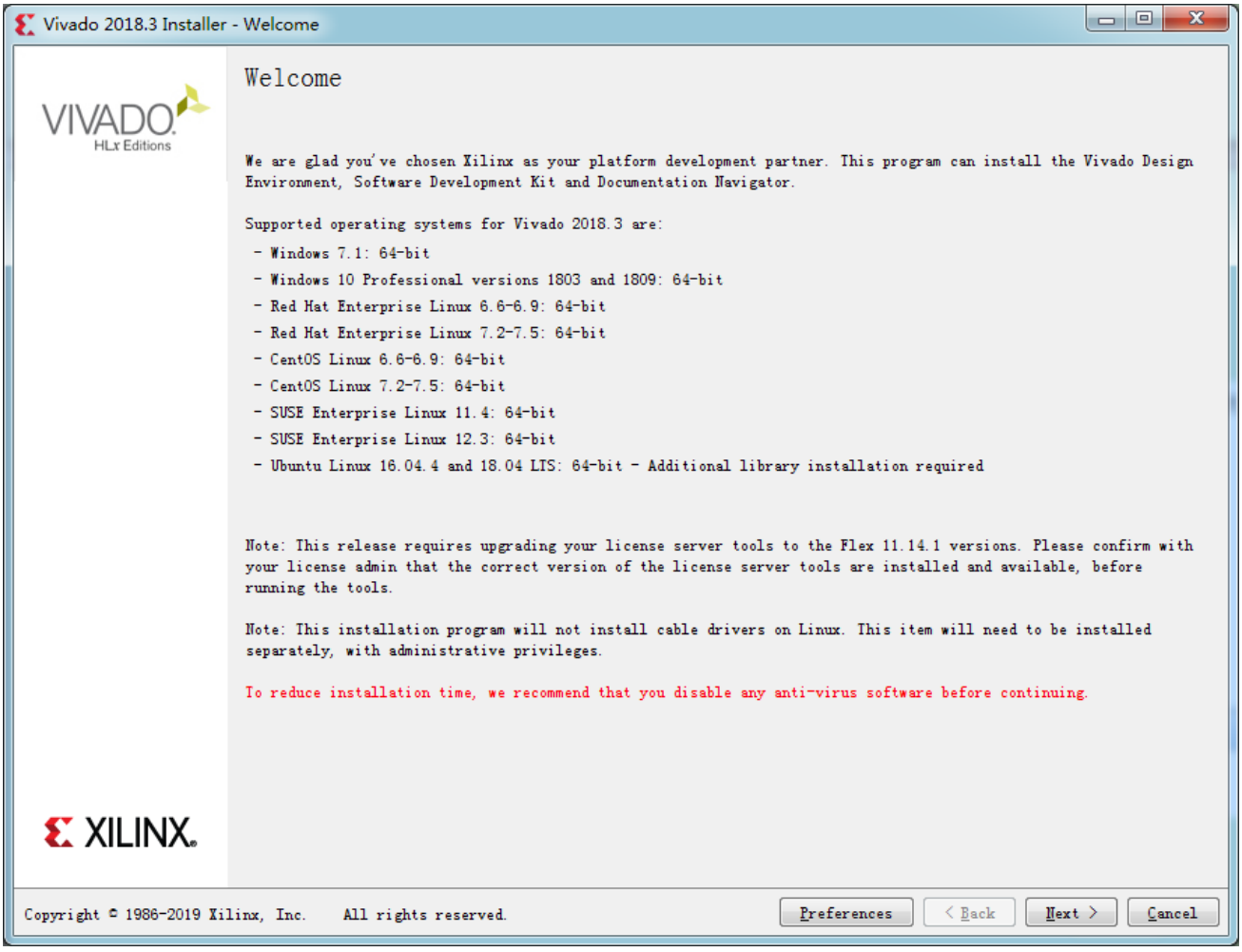


图 4.1.3 Vivado 软件的安装引导页面

此外，如果电脑连接到了互联网，有可能会弹出如下消息框：

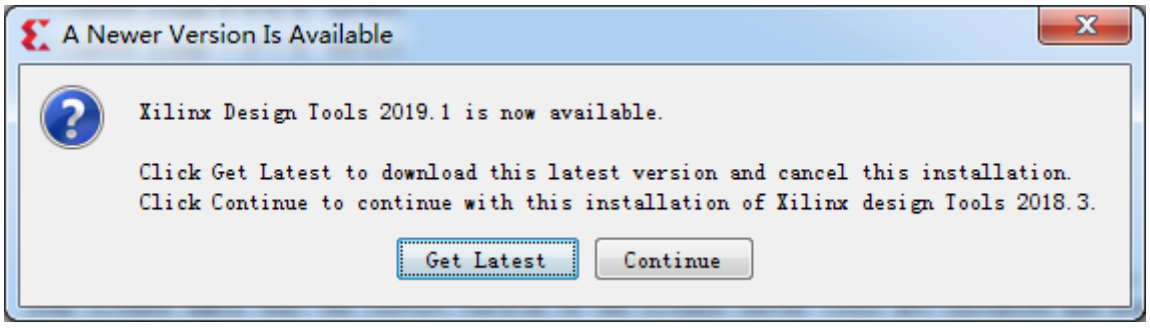


图 4.1.4 询问是否安装最新版本

意思是安装程序发现了比当前 v2018.3 更新的版本，询问用户是否安装最新版。如果要安装最新版，则还要重新下载最新版的安装包，会耗费很多时间。另外，为了避免出现不同版本之间软件兼容的问题，强烈建议大家安装和我们例程一致的版本，即 Vivado2018.3。所以这里我们点击“Continue”，即继续安装。

然后点击“Next”，如下图所示：

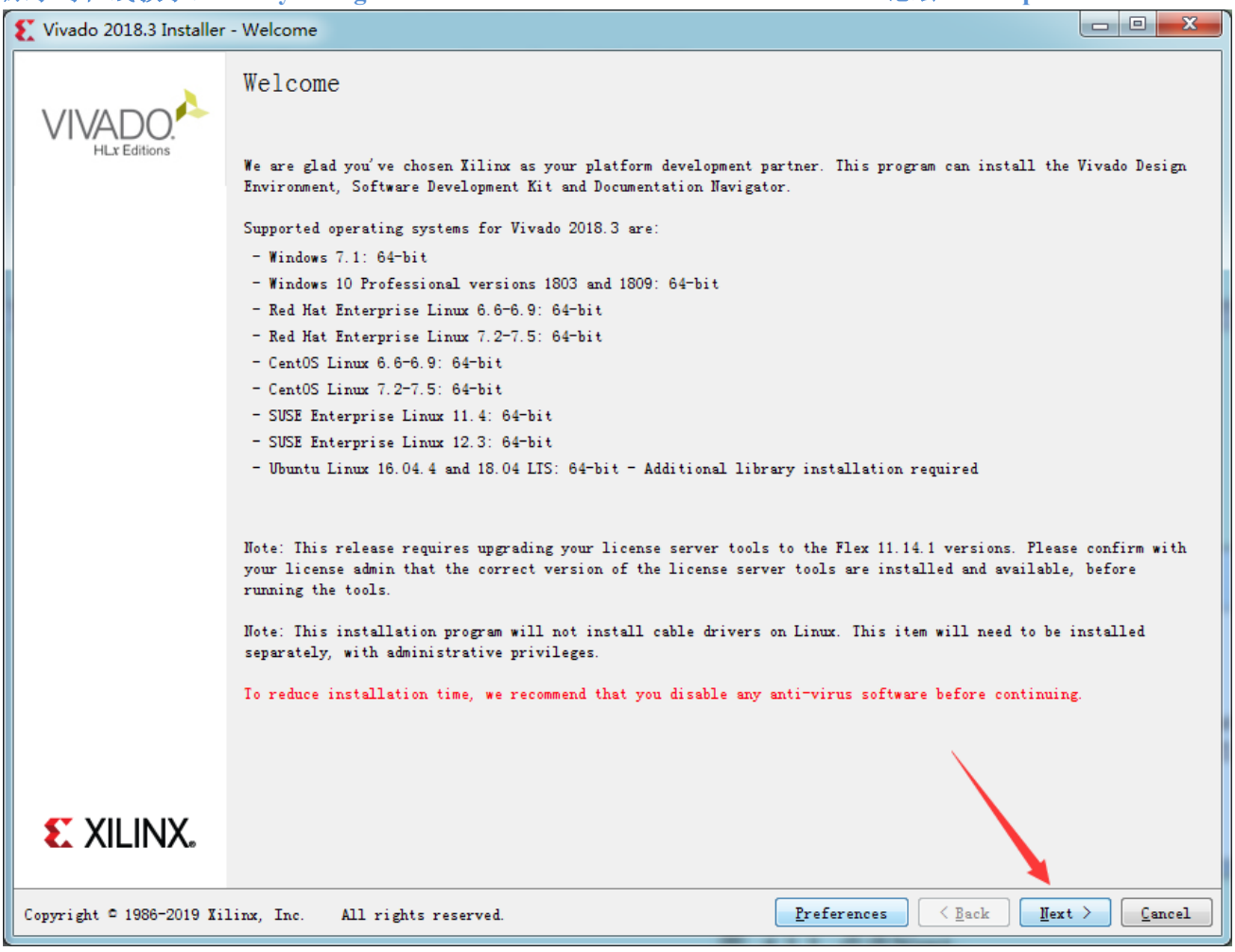


图 4.1.5 点击 Next

在接下来的页面中，勾选 3 个“I Agree”，然后点击“next”如下图所示：

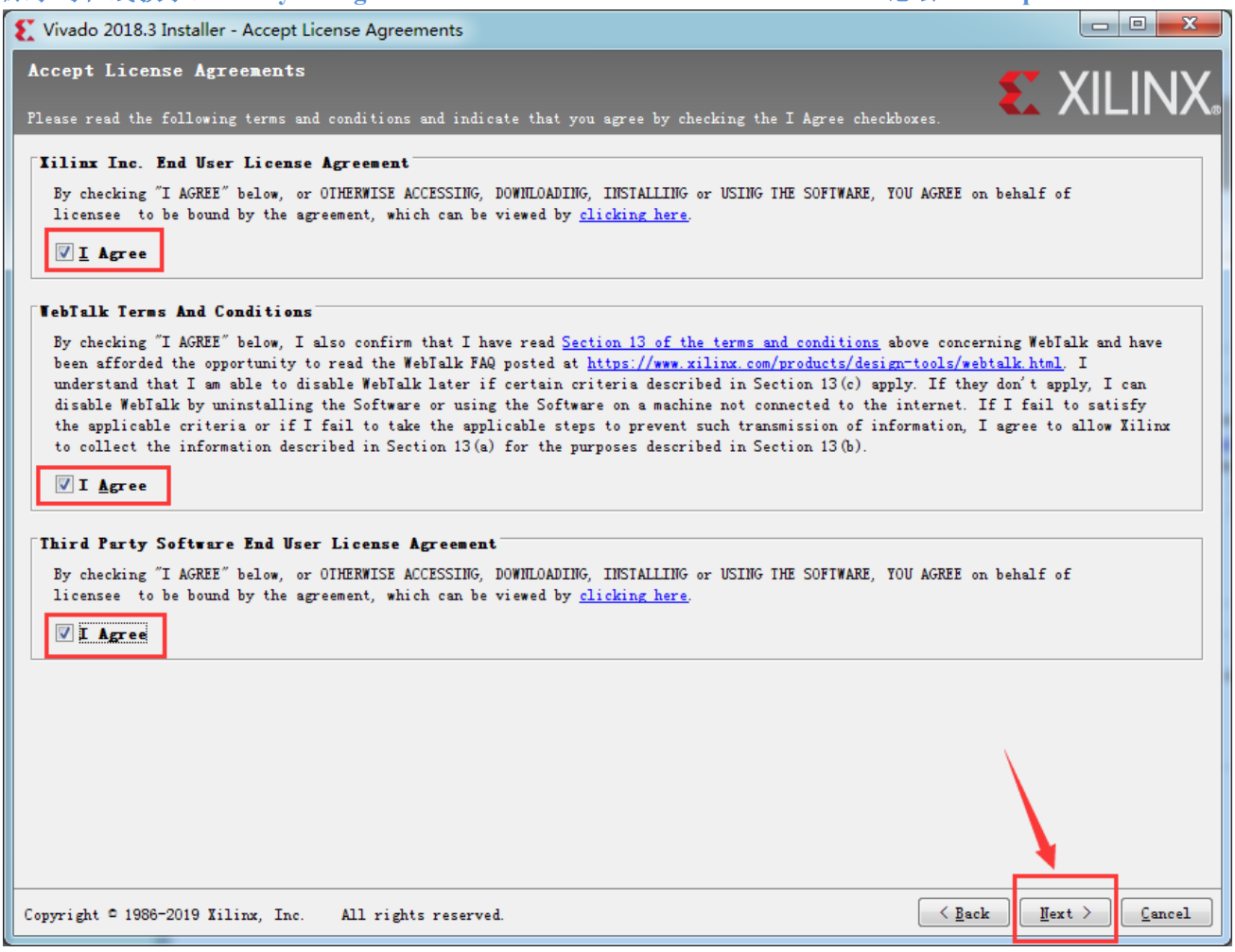


图 4.1.6 勾选“I Agree”

接下来是选择版次，这里我们选择全功能的版次，即“System Edition”，其包含最多的子组件。如下图所示：

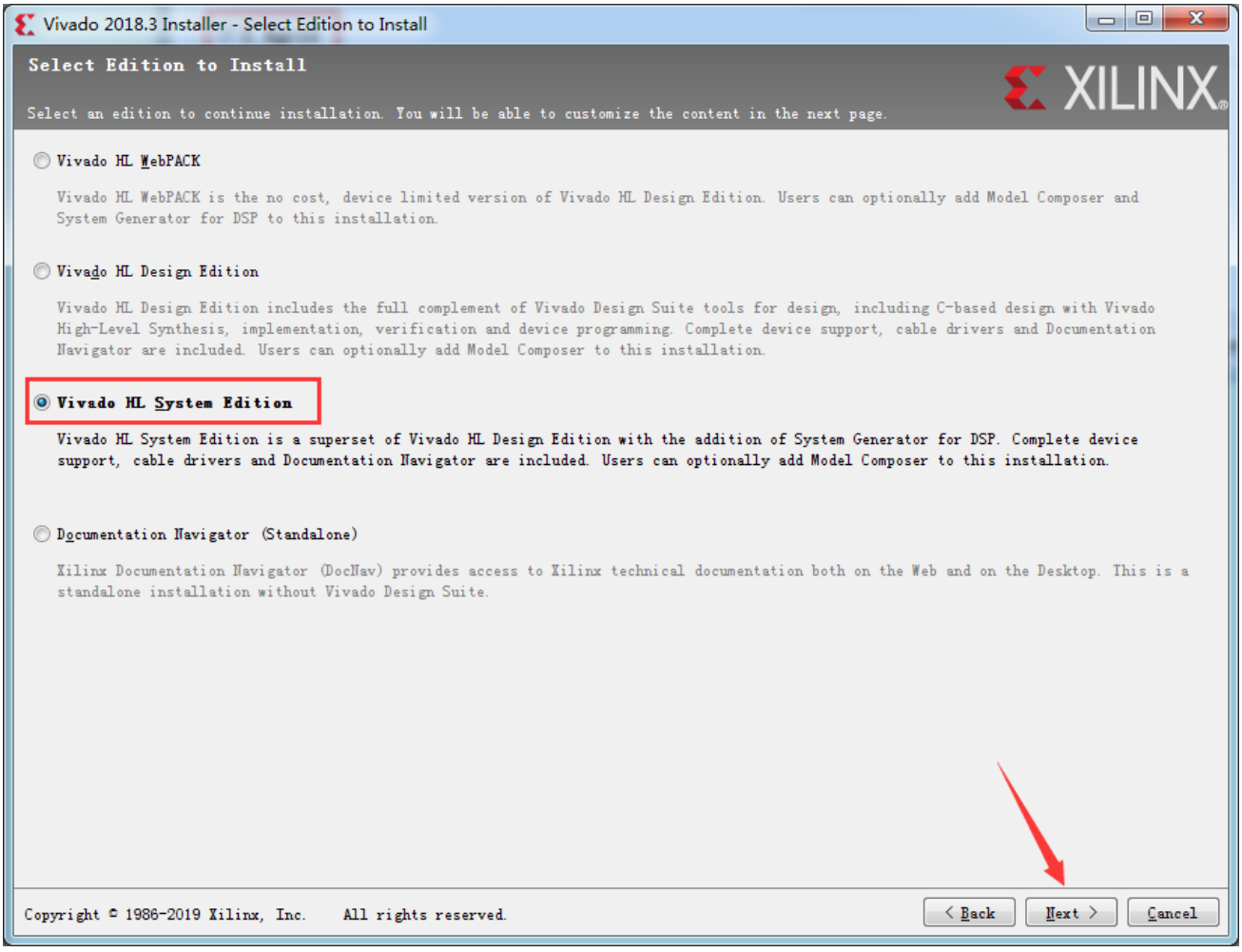


图 4.1.7 选择版次

接下来是选择工具组件和器件库。为了节省存储空间，我们将用不到的工具组件和器件库去掉，如下图所示：

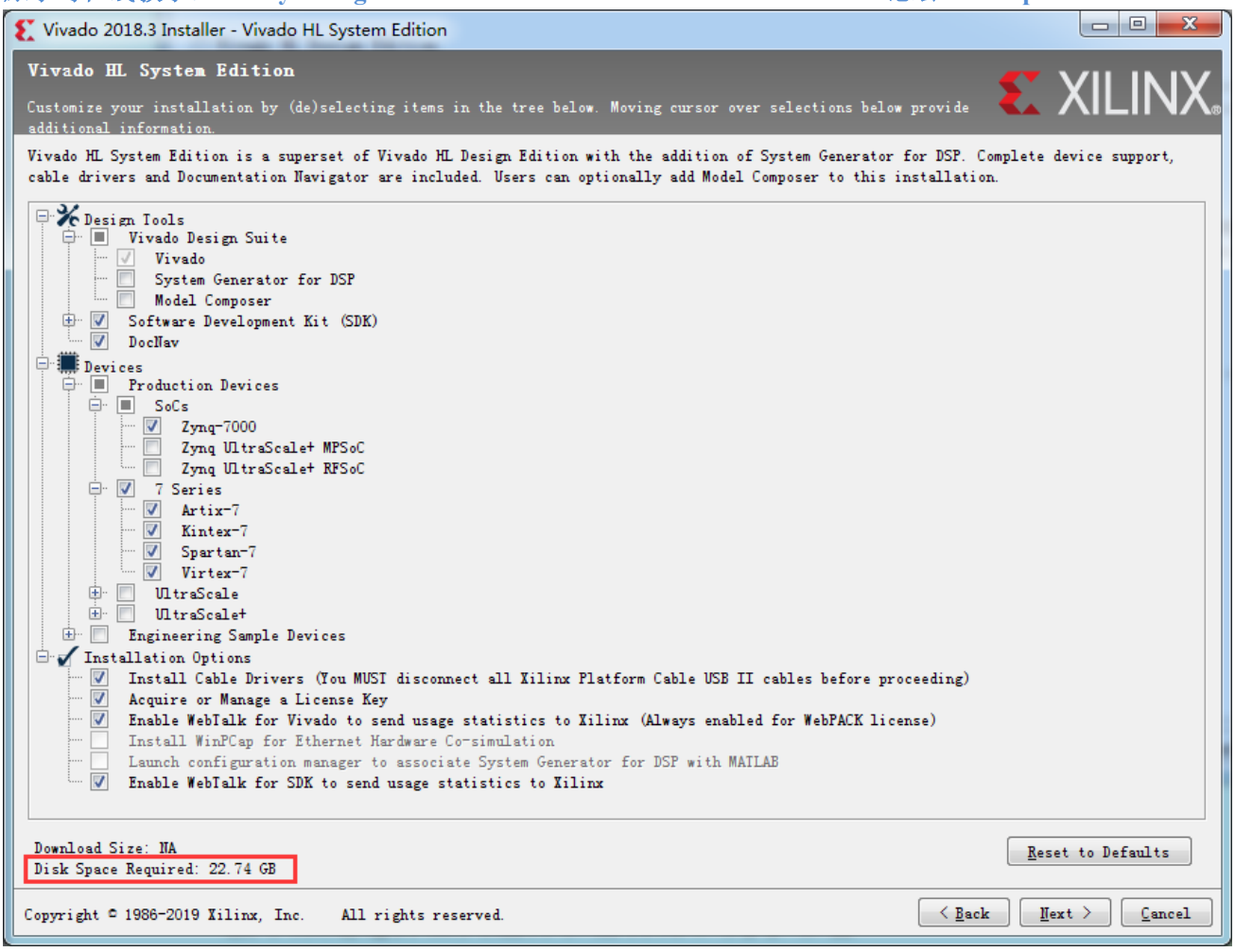


图 4.1.8 选择组件和器件库

最下面的“Disk Space Required”表示在当前选项下 Vivado 在安装完成后所占用的磁盘空间大小，为 22.74GB。由此可见，Vivado 对硬盘存储空间的占用相对来说还是挺大的。

点击 Next，进入安装目录设置页面，如下图所示：

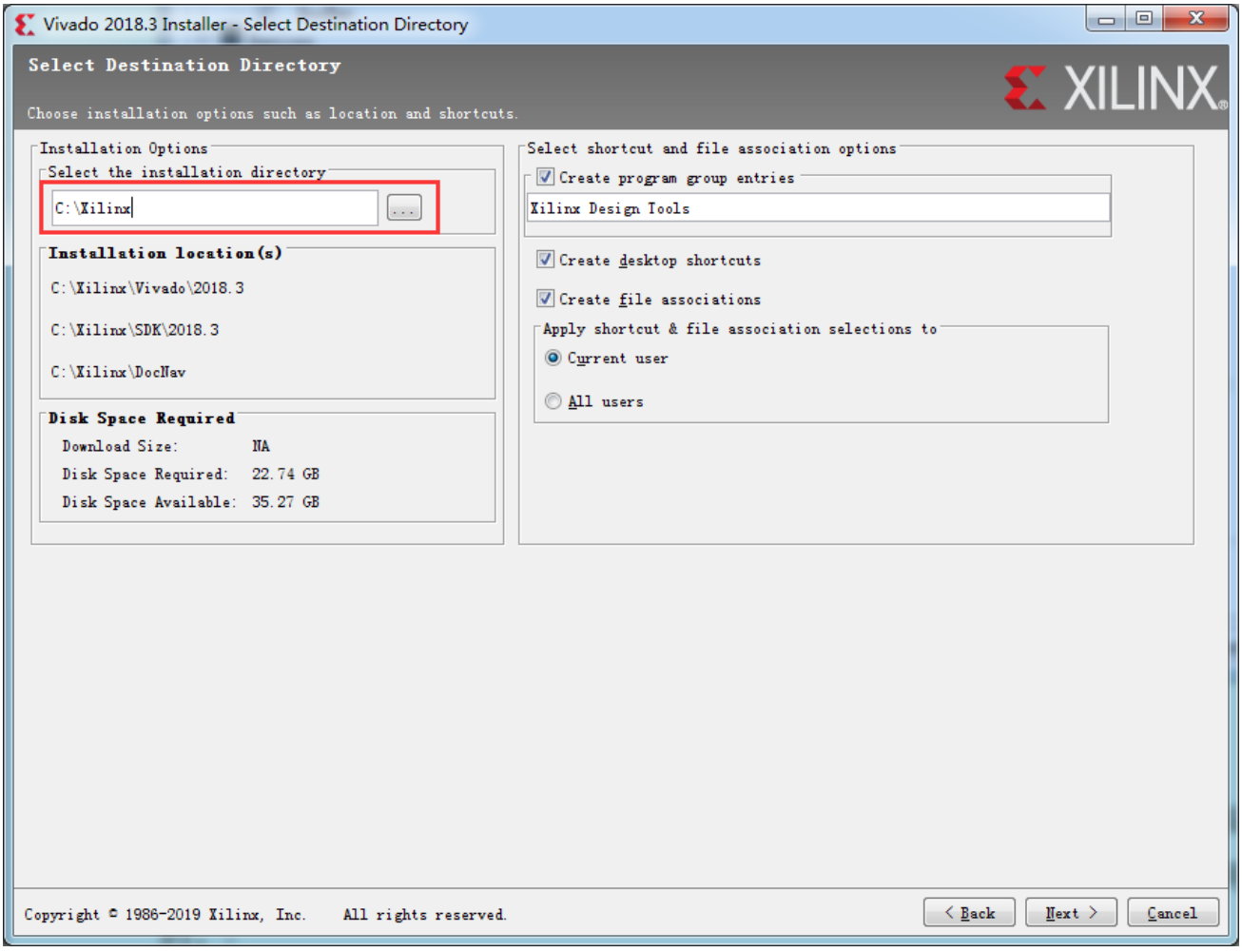


图 4.1.9 安装目录设置

图中红色方框内是对安装目录的设置，默认安装在 C 盘下的“Xilinx”文件夹下，如果需要，可以点击后面的三个点来修改安装目录（**注意，安装路径只能够包含字母、数字、下划线，否则安装程序有可能出问题**）。其他的设置保持默认即可。

点击 Next，进入 Summary 界面，该界面总结了前面所有安装的配置信息，供用户浏览确认。确认无误后，点击“Install”开始安装 Vivado 设计套件，如下图所示。（由于 Vivado 在安装期间会占用大量的电脑 CPU 资源和内存资源，所以笔者建议在开始安装之前，尽量关闭电脑中其他的不必要的应用软件）

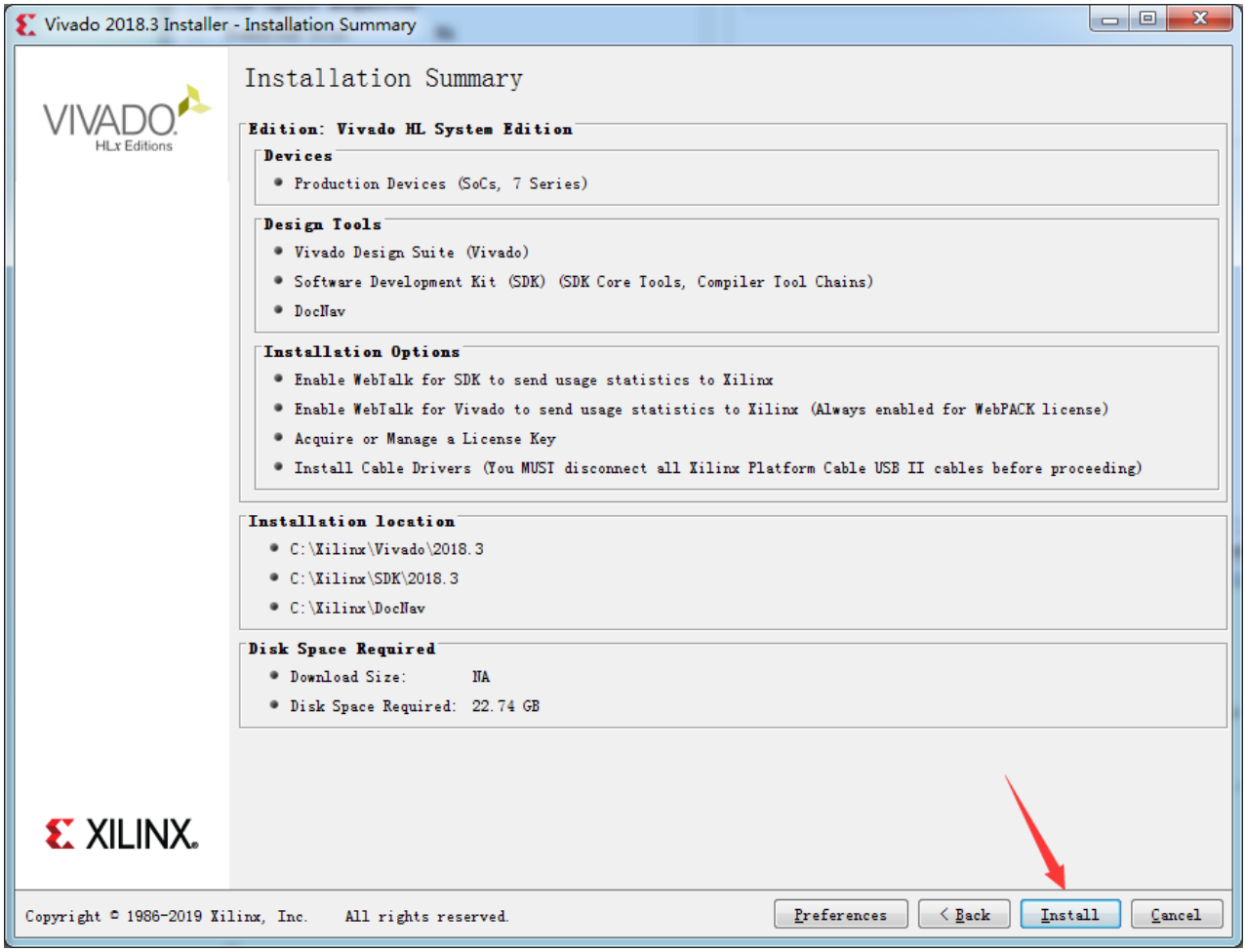


图 4.1.10 开始安装

之后会出现下面的正在安装界面：

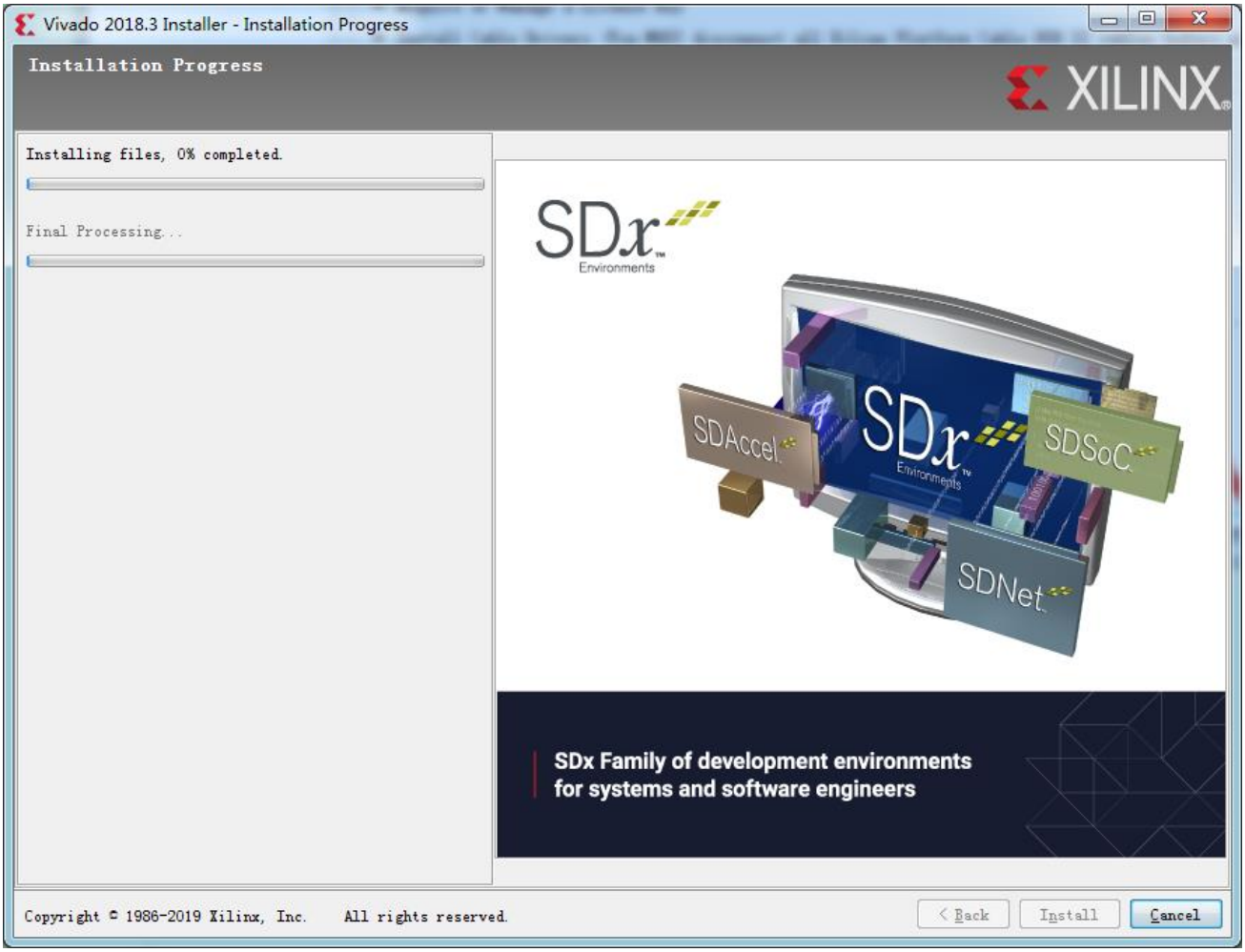


图 4.1.11 正在安装

安装过程可能会耗费一些时间，请读者耐心等待。

在安装期间可能会出面如下消息：

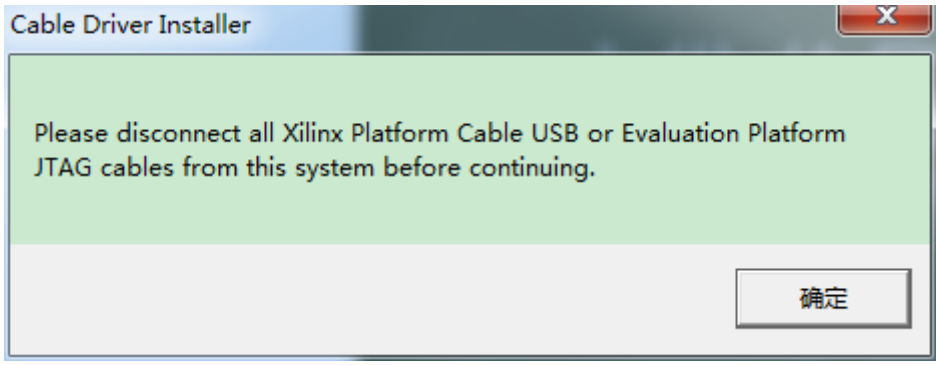


图 4.1.12 点击确定即可

弹出的这个界面是提示我们断开所有的 Xilinx 下载器与电脑的连接。值得注意是，在安装 Vivado 软件的过程中，会安装 Xilinx 下载器的驱动程序，这里必须断开 Xilinx 下载器和电脑的连接，否则下载器的驱动可能安装失败。断开连接后，点击确定即可。

最后出现了安装成功的消息窗口，如下图所示：

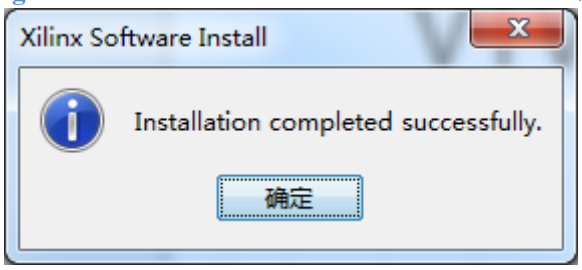


图 4.1.13 安装成功

我们直接点击确定即可。一同弹出的还有“Vivado License Manager”窗口，我们可以选择 30 天试用期，也可以通过购买 Xilinx 正版的 License 等途径来正常使用（请查看安装包目录下“安装说明.txt”）。如下图所示：

图 4.1.14 “Vivado License Manager”窗口

至此，Vivado 设计套件的安装就成功完成了，我们可以在电脑桌面上看到 Vivado2018.3 的图标，如下图所示：

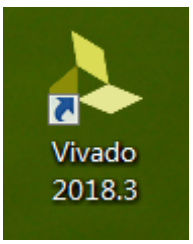


图 4.1.15 Vivado 2018.3 的桌面图标

4.2 Vivado 软件的使用

在开始使用 Vivado 软件之前，我们先来了解一下 Vivado 软件的使用流程，如下图所示：

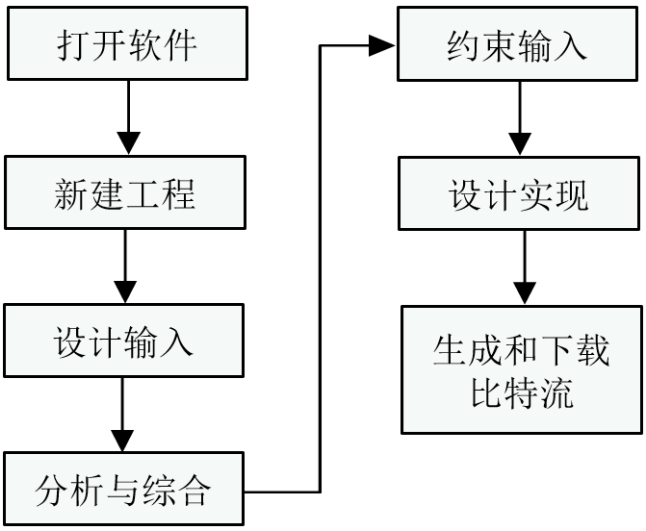


图 4.2.1 Vivado 软件使用流程

从上图可以看出，首先打开 Vivado 软件，新建一个工程，在新建工程的时候，我们可以通过新建工程向导的方式来创建工程；工程建立完成后，我们需要新建一个 Verilog 顶层文件，然后我们将设计的代码输入到新建的 Verilog 顶层文件中；HDL 源代码输入完毕之后，就是对设计文件进行分析与综合了。

在代码输入以及设计分析阶段，Vivado 软件会检查代码，如果代码出现语法错误，那么 Vivado 软件将会给出相关错误提示。在 FPGA 设计中，综合（Synthesis）就是将 RTL 设计转变为由 FPGA 器件中的查找表（LUT）、触发器（FF）等各种底层电路单元所组成的网表，在这个过程中综合器也会对设计进行优化，例如，删除多余的逻辑等等。

综合完成后，我们需要进行约束的输入。约束表达了设计者期望满足的时序要求，规范了设计的时序行为，并在综合、实现阶段来指导工具进行布局、布线，工具会按照你的约束尽量去努力实现以满足时序要求，并在时序报告中给出结果。常用的约束包括时序约束、引脚约束等等。

接下来就可以实现整个设计了，包括布局和布线等。如果实现成功，则 Vivado 会给出提示结果。此时，就可以生成用于下载到器件中的比特流文件了。最后，我们会通过下载器来将这个比特流文件下载到 FPGA 中，完成整个开发流程。

在这里，我们只是简单的介绍了一下上述的流程图，让大家对 Vivado 软件的开发流程有个大致的了解。接下来我们就以 LED 灯闪烁实验的工程为例，对每个流程进行详细的操作演示，一步步、手把手带领大家学习使用 Vivado 软件。

4.2.1 新建工程

我们直接双击桌面上的 Vivado 2018.3 软件图标，打开 Vivado 软件，Vivado 软件启动界面如下图所示，我们点击“Create Project 来创建一个新的工程”。

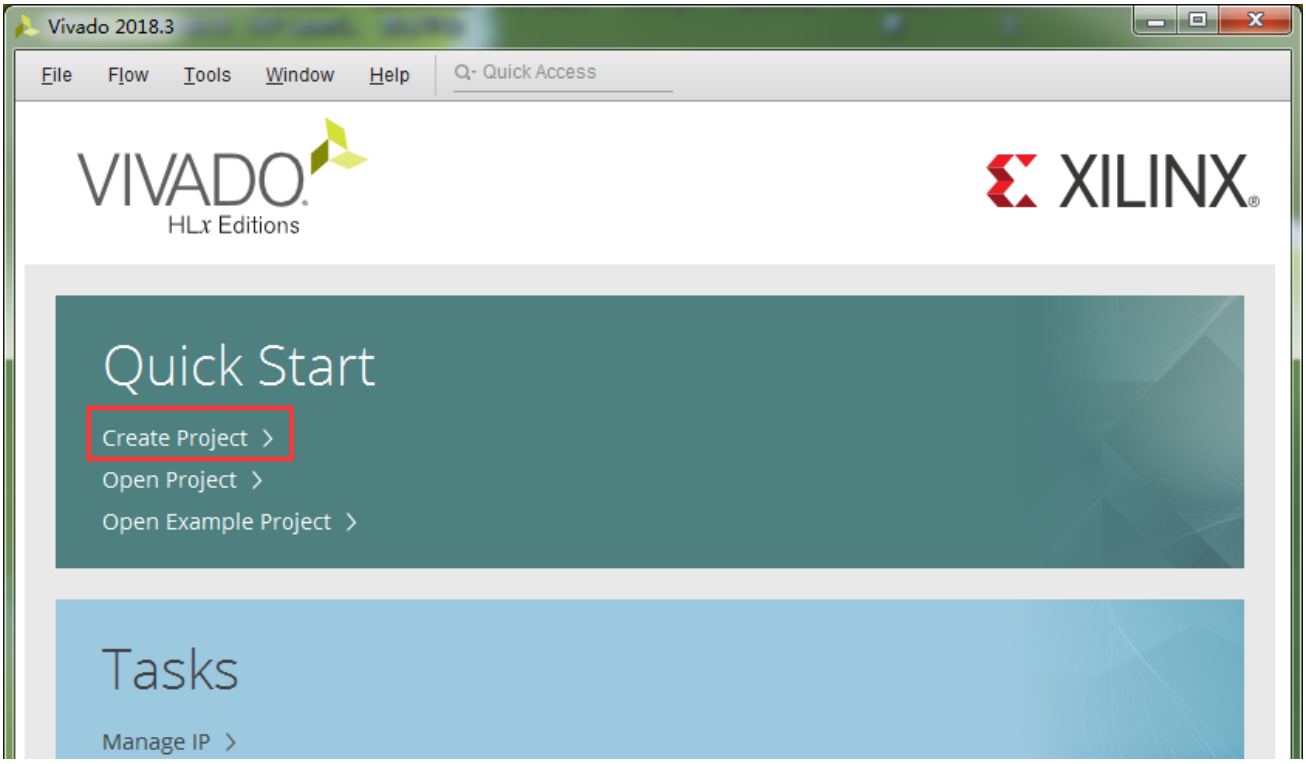


图 4.2.2 Vivado 软件启动界面

出现下图所示窗口，我们直接点击“Next”，如下图所示。

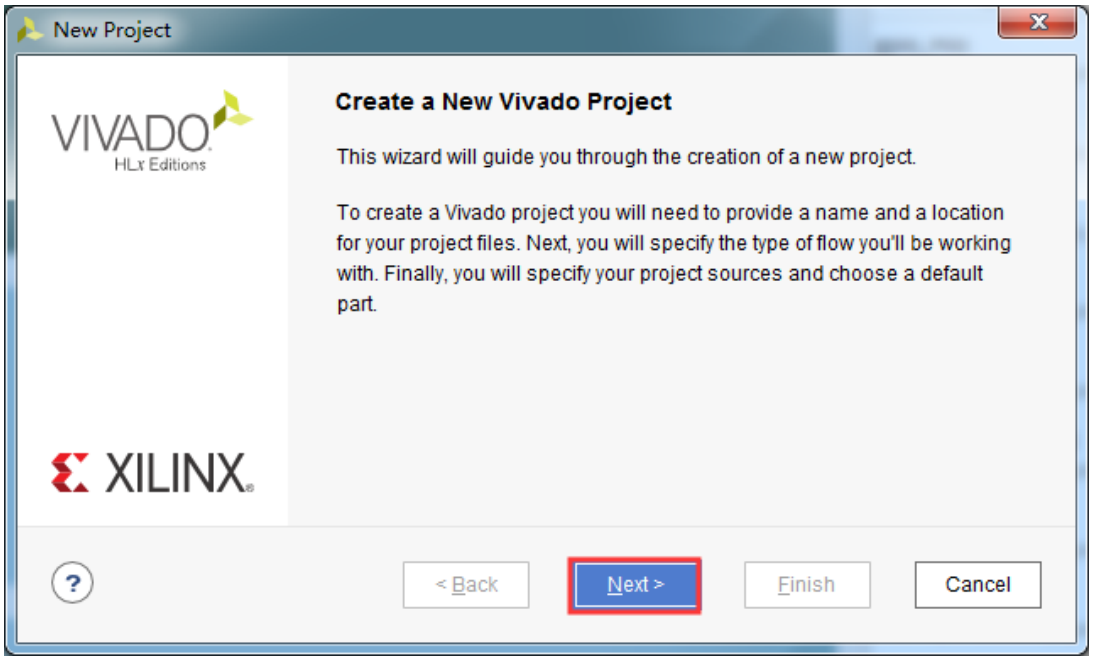


图 4.2.3 新建工程向导

接下来输入工程的名称和路径。名称要能反应出工程所实现的功能，本次工程实现了 LED 闪烁的功能，因此项目名称命名为“led_twinkle”。工程路径是指定本次工程存放在电脑磁盘中的位置，这个大家可以自行选择路径，需要说明的是，工程路径不能包含中文、空格或者其它一些特殊的符号，否则工程会创建失败。工程名和路径的设置如下图所示。

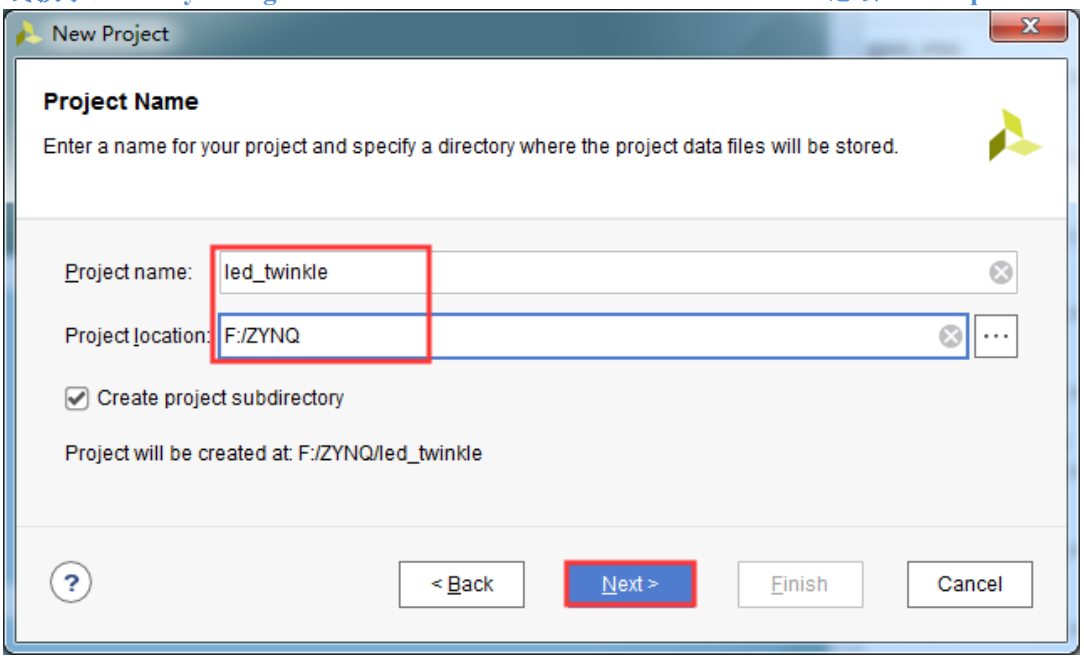


图 4.2.4 输入工程名称和路径

注意，由于默认勾选了“Create project subdirectory”选项，Vivado 会在所选工程目录下自动创建一个与工程名同名的文件夹，用于存放工程内的各种文件。并且 Vivado 会自动管理工程文件夹内的各种工程文件，并创建相应的子目录，这为我们的开发工作带来了很大的便捷。

我们继续点击“Next”按钮，接下来是工程类型的选择，我们选择“RTL Project”，如下图所示：

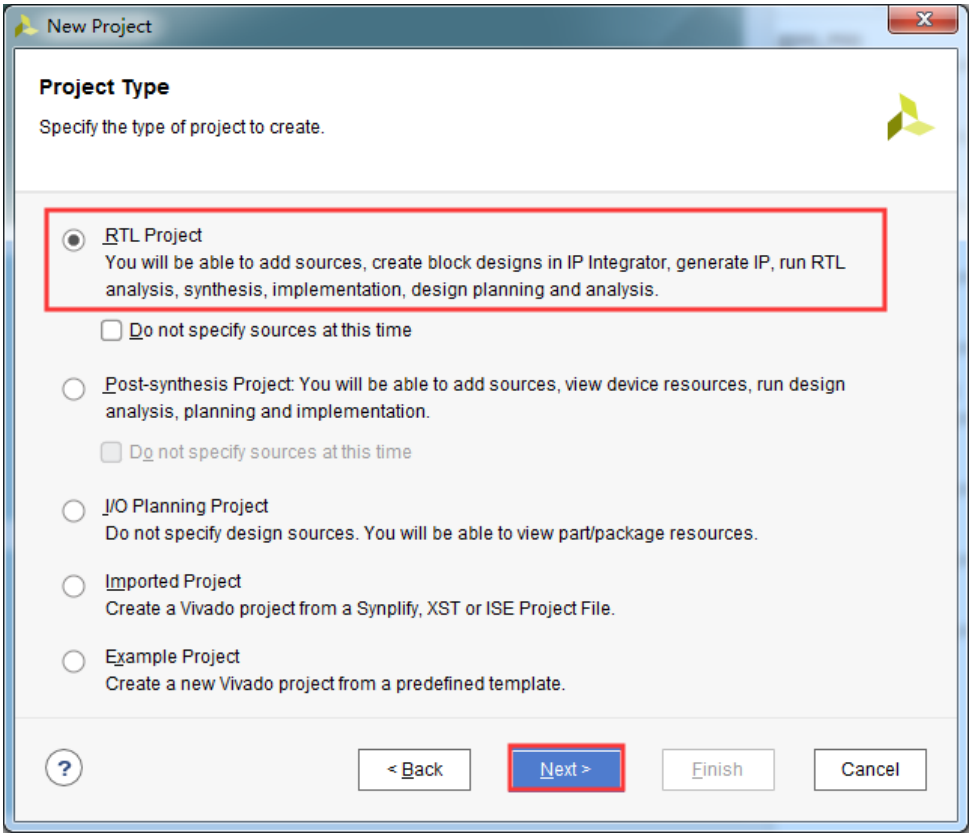


图 4.2.5 工程类型的选择

这里简单介绍下各个工程类型的含义。“RTL Project”是指按照正常设计流程所选择的类型，这也是常用的一种类型，“RTL Project”下的“Do not specify sources at this time”用于设置是否在创建工程向导的过程中添加设计文件，如果勾选后，则不创建或者添加设计文件；“Post-synthesis Project”在导入第三方工具所产生的综合后网表时才选择；“I/O Planning Project”一般用于在开始 RTL 设计之前，创建一个用于早期 IO 规划和器件开发的空工程；“Imported Project” 用于从 ISE、XST 或 Synopsys Synplify 导入现有的工程源文件；“Example Project”是指创建一个 Vivado 提供的工程模板。

选择了“RTL Project”后，我们点击“Next”，进入添加源文件页面。注意，如果勾选中图 4.2.5 中“RTL Project”下的“Do not specify sources at this time”，则不会出现添加源文件的界面。

在弹出添加源文件的界面后，可以在此处创建/添加源文件，当然也可以直接点击“Next”，创建完工程后再创建/添加源文件。这里直接点击“Next”，如下图所示：

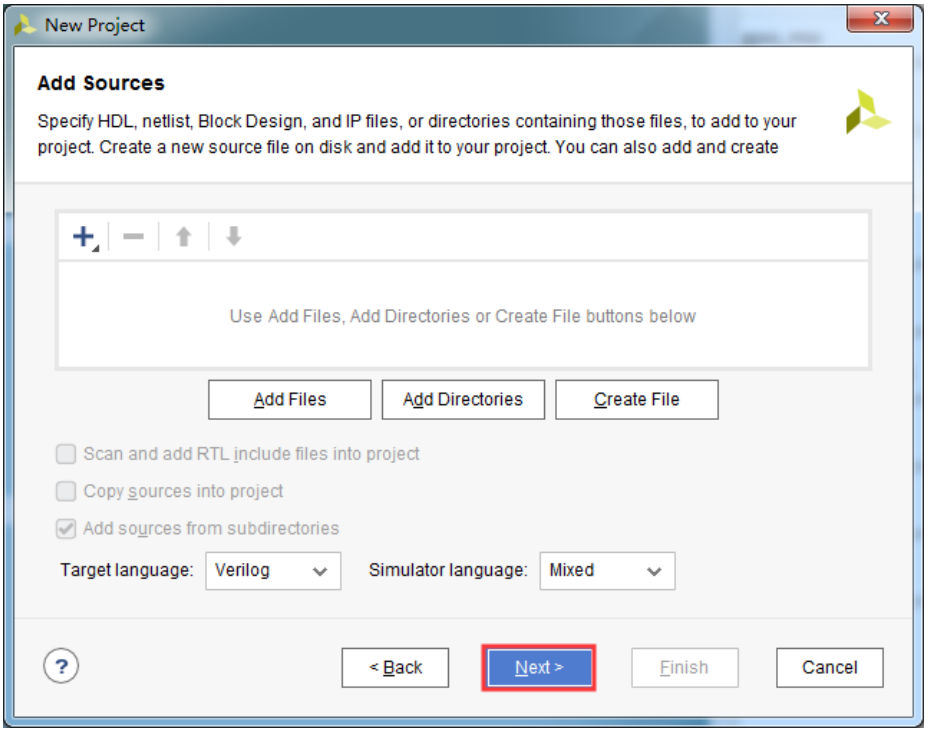


图 4.2.6 添加源文件

接下来是添加约束文件，我们也是直接点击“Next”，创建完工程后再创建/添加约束文件，如下图所示：

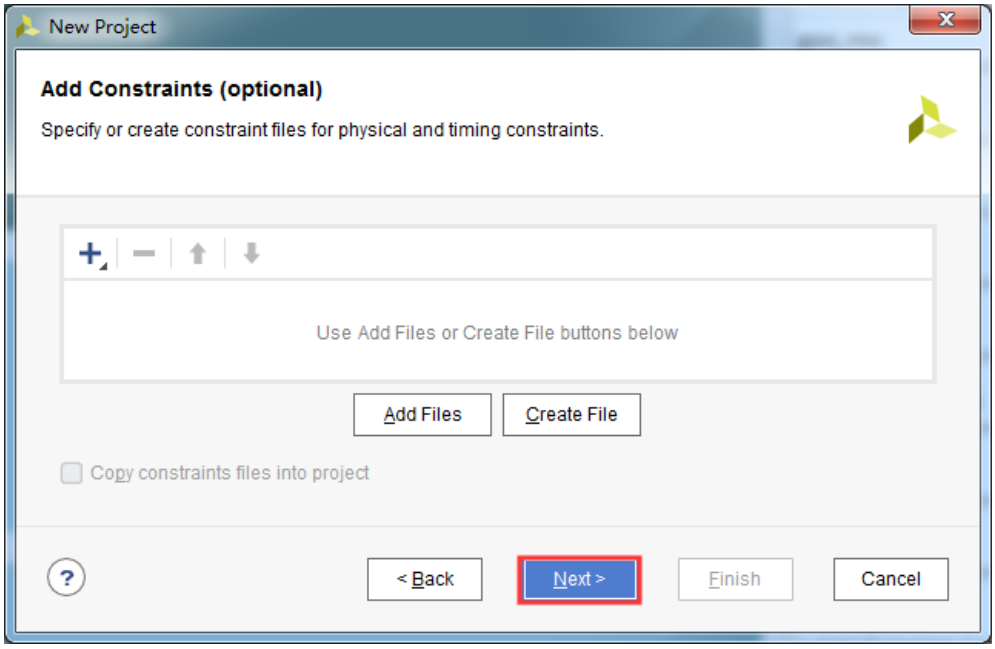


图 4.2.7 添加约束文件

接下来选择开发板的芯片型号，我们可以直接在搜素框中输入完整的芯片型号，大家根据自己所使用的 ZYNQ 核心板型号进行选择。如果使用的是 ZYNQ-7020 核心板，则输入“xc7z020clg400-2”，如下图所示：

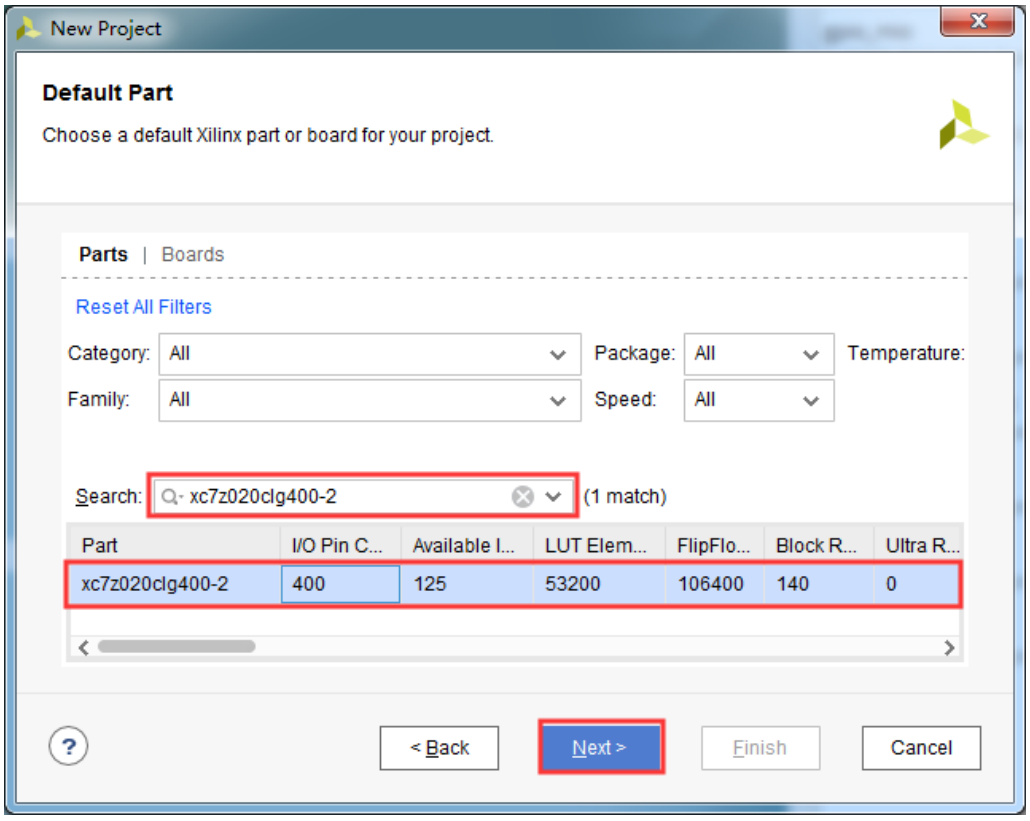


图 4.2.8 ZYNQ-7020 核心板芯片型号

如果使用的是 ZYNQ-7010 核心板，则输入“xc7z010clg400-1”，如下图所示：

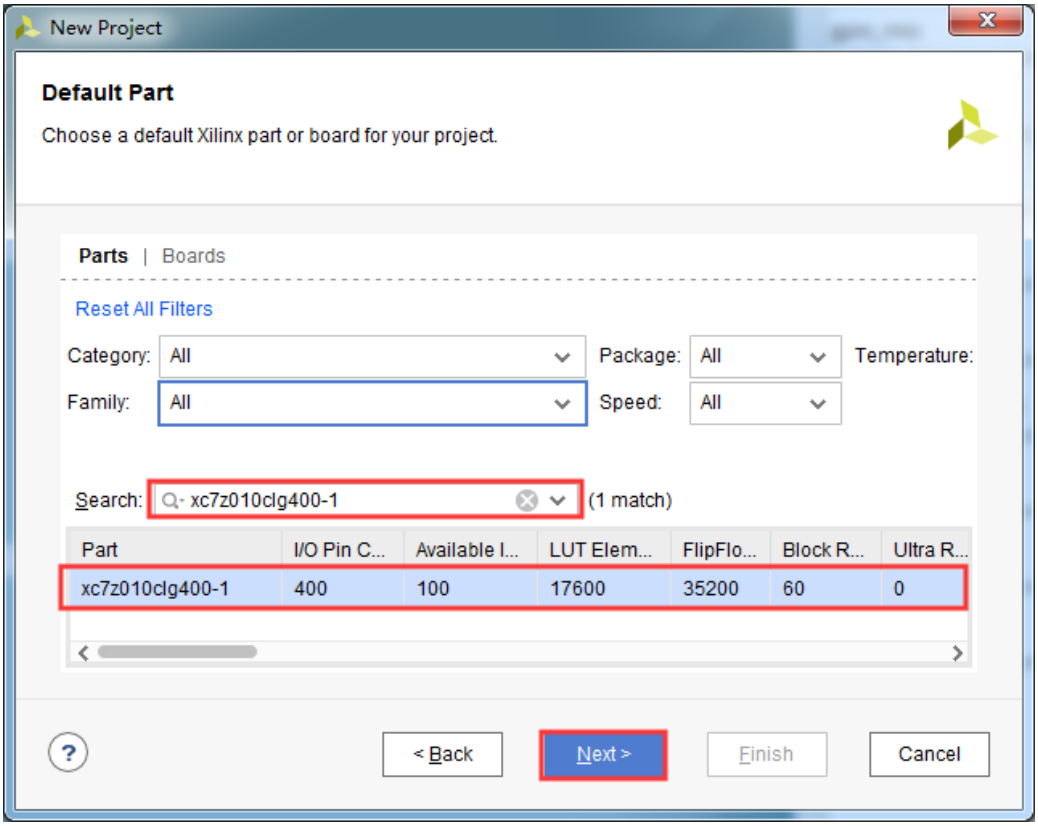


图 4.2.9 ZYNQ-7010 核心板芯片型号

在搜素框中输入完整的芯片型号后，在“Part”一栏会出现唯一匹配的型号，单击选中“Part”一栏的芯片型号，然后点击“Next”按钮。

需要说明的是，本次工程以 ZYNQ-7020 核心板为例，接下来的软件截图可能会出现 ZYNQ-7020 器件的芯片型号。大家使用 ZYNQ-7020 核心板和 ZYNQ-7010 核心板除了在创建工程向导选择的芯片型号不一样外，其余操作都是一样的，因此，我们接下来只贴 ZYNQ-7020 器件的软件截图。

最后进入工程概览页面，这个页面将之前几个步骤中的设置全部列了出来，供用户检查，选择不同的芯片型号，概览页面列举的芯片型号也不一样，我们直接点击“Finish”按钮完成工程的创建，如图 4.2.10 所示。

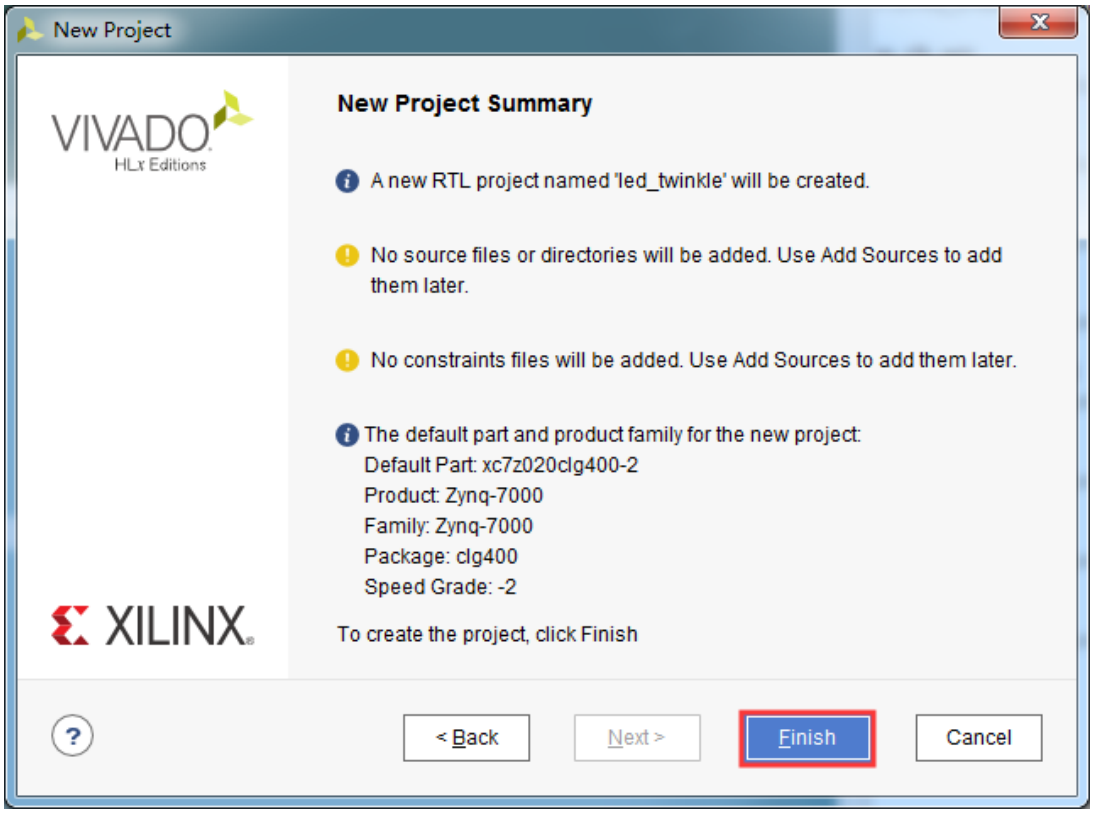


图 4.2.10 工程概览（Summary）页面

工程创建完成后，就进入了 Vivado 的工程主界面，如下图所示：

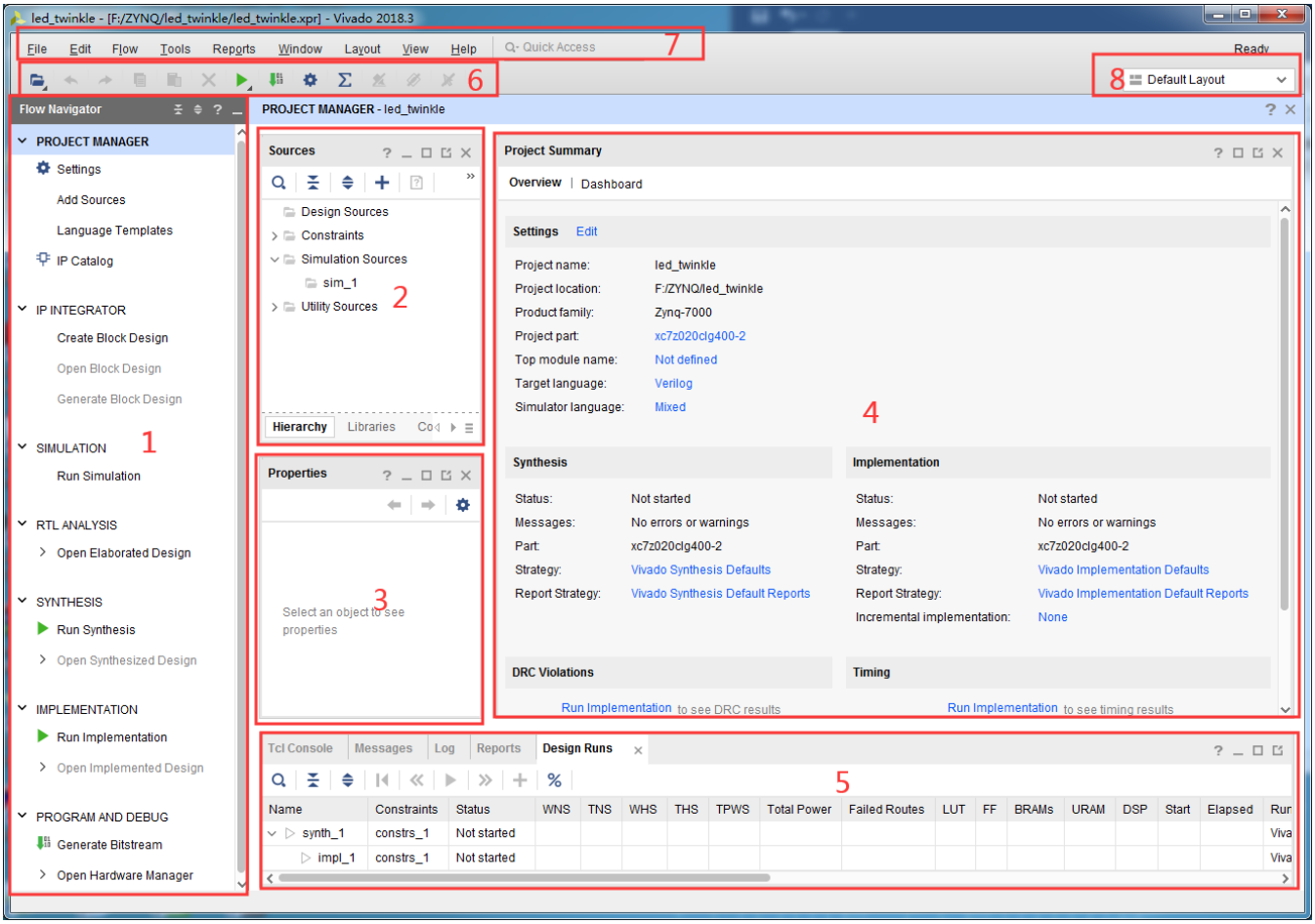


图 4.2.11 Vivado 工程主界面

下面介绍 Vivado 工程主界面中的几个主要子窗口：

（1）Flow Navigator。Flow Navigator 提供对命令和工具的访问，其包含从设计输入到生成比特流的整个过程。 在点击了相应的命令时，整个 Vivado 工程主界面的各个子窗口可能会作出相应的更改。

（2）数据窗口区域。默认情况下，Vivado IDE 的这个区域显示的是设计源文件和数据相关的信息。

• Sources 窗口：显示层次结构（Hierarchy）、IP 源文件（IP Sources）、库（Libraries）和编译顺序（Compile Order）的视图。

• Netlist 窗口：提供分析后的（elaborated）或综合后的（synthesized）逻辑设计的分层视图。

（3）Properties 窗口：显示有关所选逻辑对象或器件资源的特性信息。

（4）工作空间（Workspace）：工作区显示了具有图形界面的窗口和需要更多屏幕空间的窗口，包括：

• Project Summary。提供了当前工程的摘要信息，它在运行设计命令时动态地更新。

• 用于显示和编辑基于文本的文件和报告的 Text Editor。

• 原理图（Schematic）窗口。

• 器件（Device）窗口。

• 封装（Package）窗口。

（5）结果窗口区域：在 Vivado IDE 中所运行的命令的状态和结果，显示在结果窗口区域中，这是一组子窗口的集合。在运行命令、生成消息、创建日志文件和报告文件时，相关信息将显示在此区域。默认情况下，此区域包括以下窗口：

• Tcl Console：允许您输入 Tcl 命令，并查看以前的命令和输出的历史记录。

• Messages：显示当前设计的所有消息，按进程和严重性分类，包括“Error”、“Critical Warning”、“Warning”等等

• Log：显示由综合、实现和仿真 run 创建的日志文件。

• Reports：提供对整个设计流程中的活动 run 所生成的报告的快速访问。

• Designs Runs：管理当前工程的 runs。

（6）主工具栏：主工具栏提供了对 Vivado IDE 中最常用命令的单击访问。

（7）主菜单：主菜单栏提供对 Vivado IDE 命令的访问。

（8）窗口布局（Layout）选择器：Vivado IDE 提供预定义的窗口布局，以方便设计过程中的各种任务。布局选择器使您能够轻松地更改窗口布局。或者，可以使用菜单栏中的“Layout”菜单来更改窗口布局。

4.2.2 设计输入

下面我们就来创建工程顶层文件，我们点击“Sources”窗口中的“+”号，如下图所示：

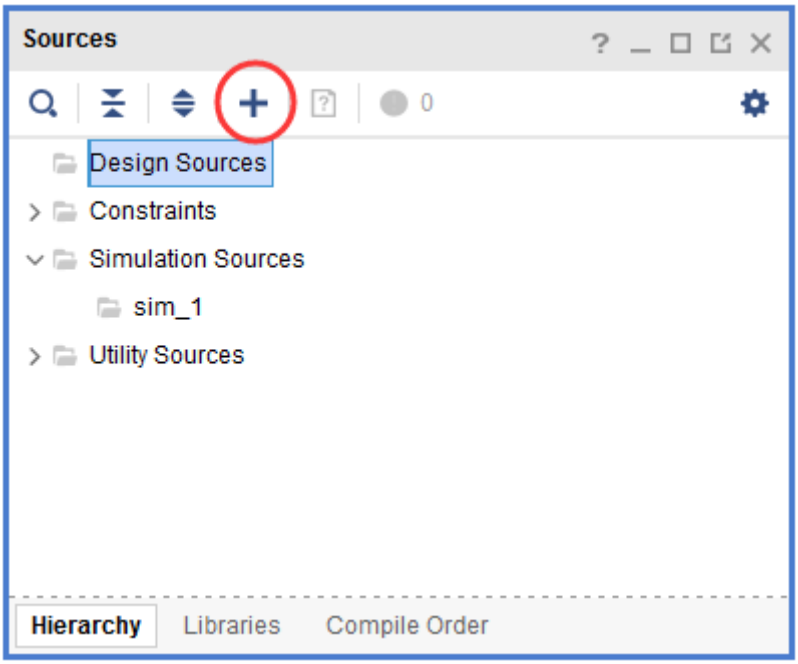


图 4.2.12 添加源文件按钮

弹出下图所示界面，我们选择添加设计源文件（注意，Vivado 不支持使用原理图的方式来输入设计），然后点击“Next”按钮，如下图所示。

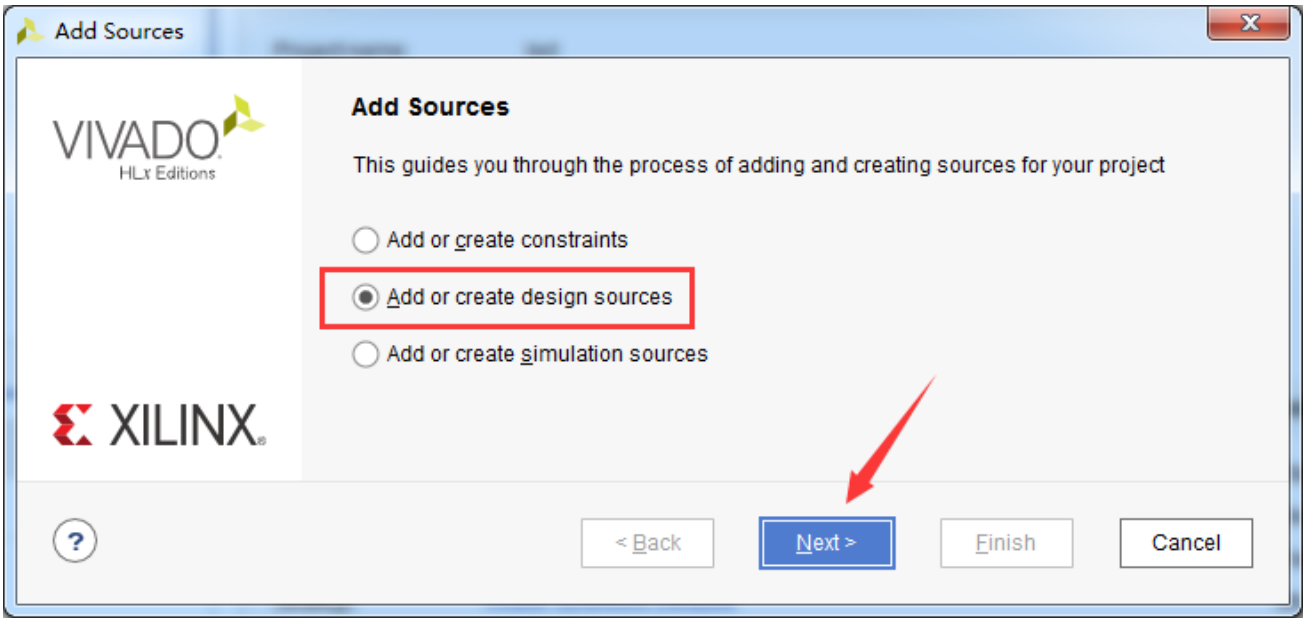


图 4.2.13 选择添加设计源文件

接下来在弹出的页面中添加或者创建一个文件。如果事先有编写好的代码，可以点击“Add Files”按钮来添加文件；如果没有，则点击“Create File”创建一个新的设计文件。由于我们事先没有编写好的设计文件，这里点击“Create File”来创建一个新的设计文件，如下图所示：

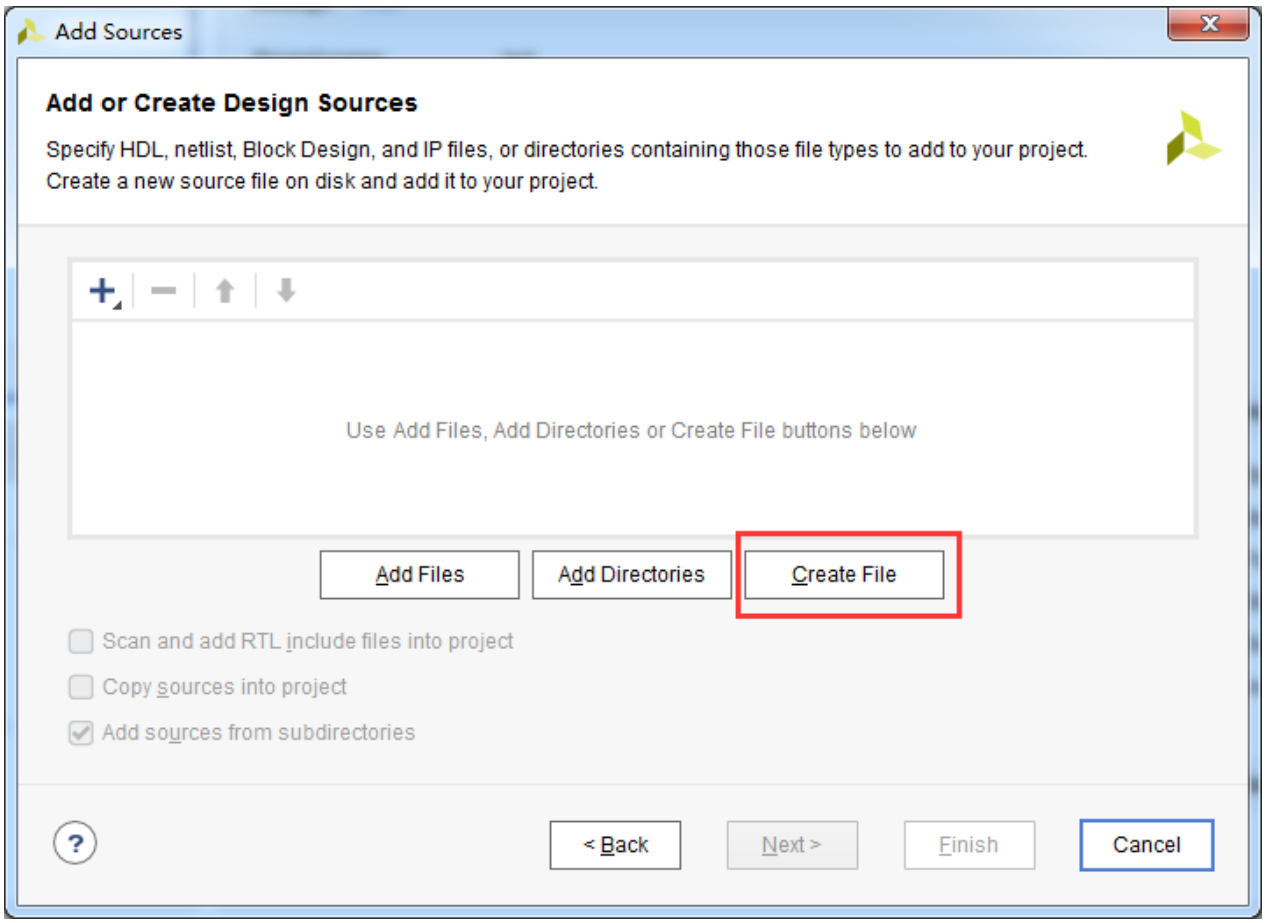


图 4.2.14 点击创建源文件

接下来会弹出一个对话框，对创建的设计文件进行命名。这里我们输入源文件的名称“led_twinkle”，然后点击“OK”按钮，如下图所示：

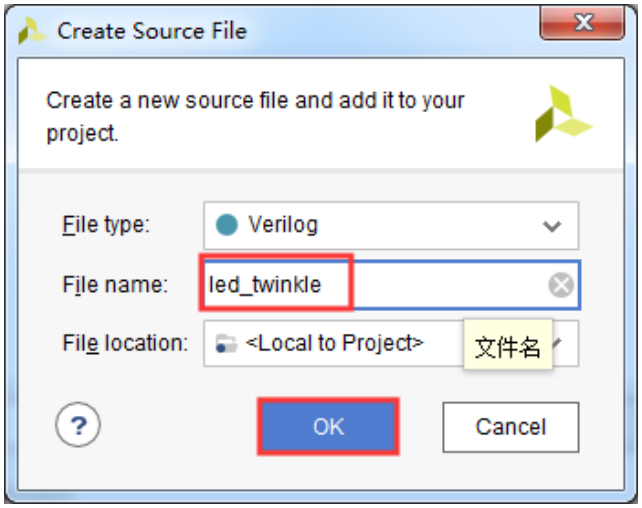


图 4.2.15 输入源文件名称

这时我们看到列表中已经出现了刚刚新创建的设计文件，点击“Finish”按钮，如下图所示：

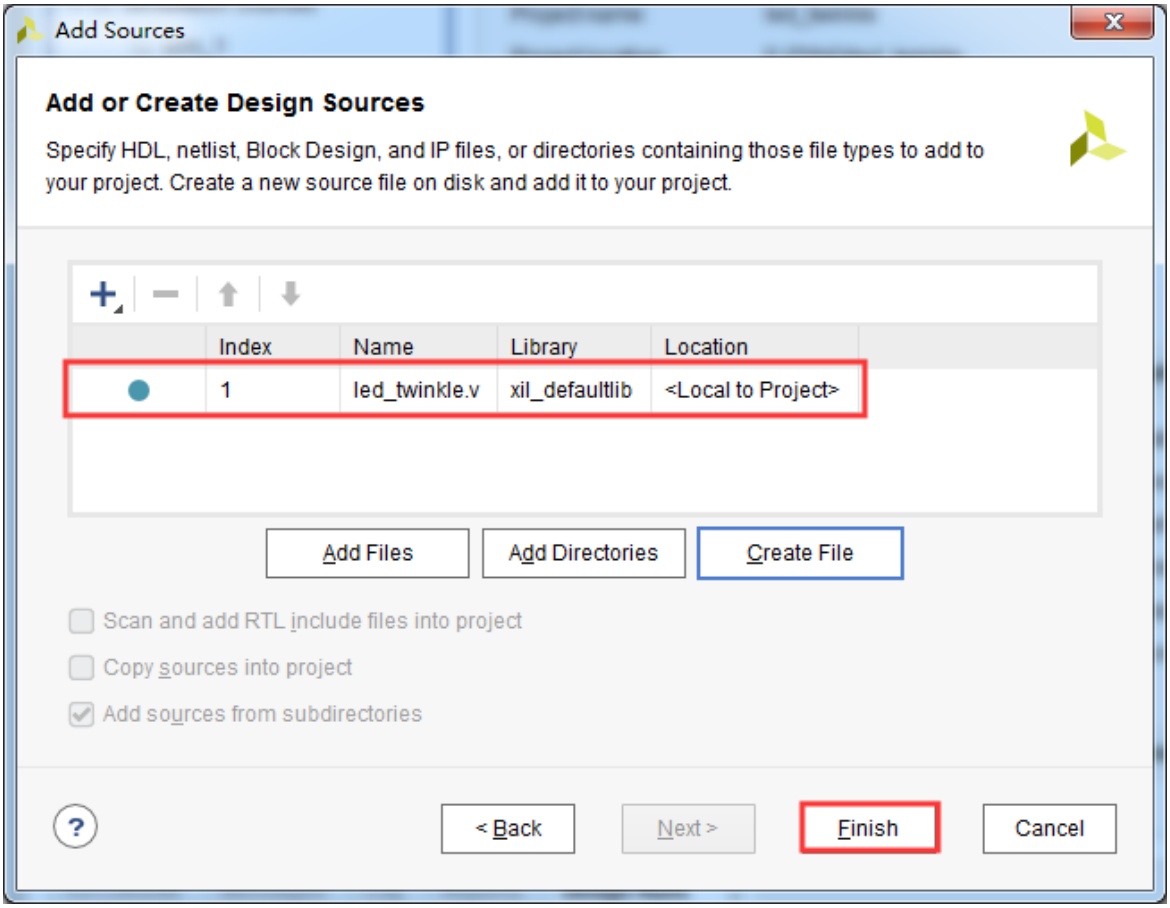


图 4.2.16 添加的源文件

接下来会弹出一个定义模块的页面，用于设置源文件的模块名称和端口列表，Vivado 会根据在此窗口中的设置，自动地在 HDL 源文件中写入相应的 verilog 语句。我们会手动输入代码，所以这里不作任何设置，直接点击“OK”按钮即可，如下图所示。

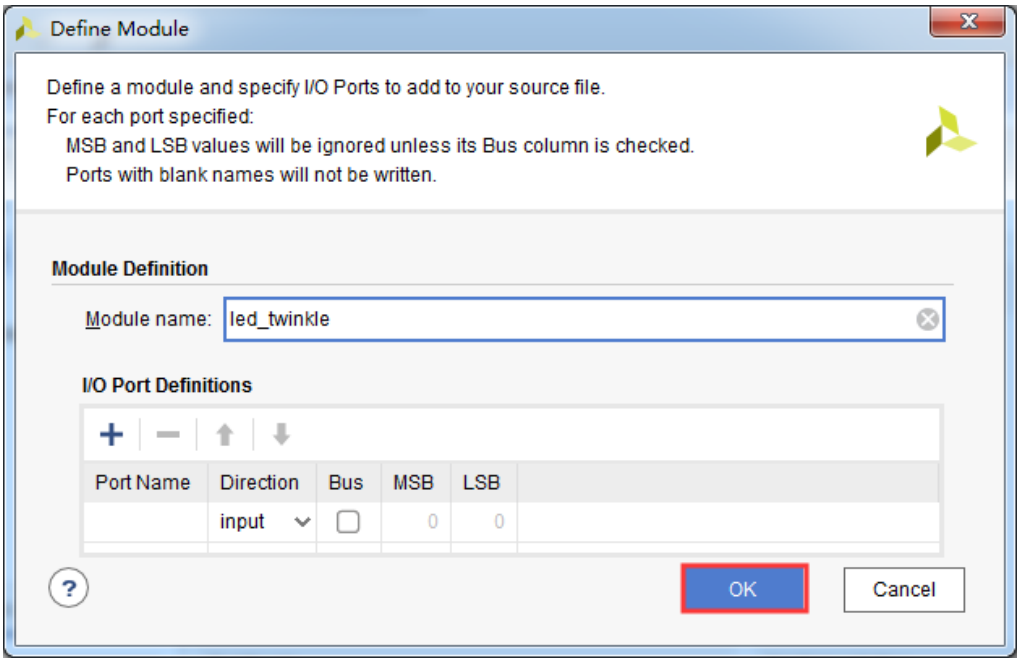


图 4.2.17 定义顶层模块名和模块端口

接下来会弹出一个模块定义确认的页面，直接点击“YES”即可，如下图所示：

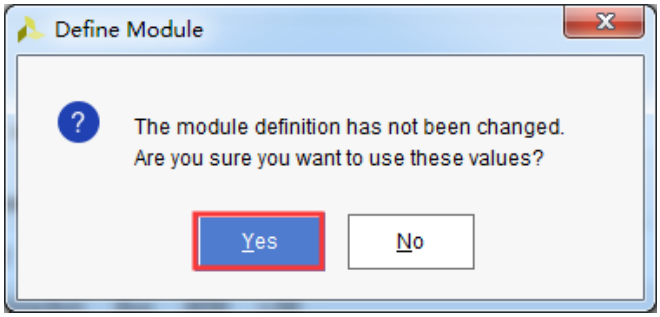


图 4.2.18 模块定义确认页面

这时工程主界面的“Sources”窗口中就出现了我们刚刚创建的源文件，如下图所示：

图 4.2.19 源文件创建完毕

我们双击打开“led_twinkle”文件，删除文件中默认的代码，然后替换成 LED 灯闪烁代码，代码如下：

```
1  module led_twinkle(
2      input           sys_clk  ,  //系统时钟
3      input           sys_rst_n,  //系统复位，低电平有效
4
5      output  [1:0]   led         //LED 灯
6  );
7
8  //reg define
9  reg  [25:0]  cnt ;
10
11 //*****************************************************
12 //**                    main code
13 //*****************************************************
14
15 //对计数器的值进行判断，以输出 LED 的状态
16 assign led = (cnt < 26'd2500_0000) ? 2'b01 : 2'b10 ;
17
18 //计数器在 0~5000_000 之间进行计数
19 always @ (posedge sys_clk or negedge sys_rst_n) begin
20     if(!sys_rst_n)
21         cnt <= 26'd0;
22     else if(cnt < 26'd5000_0000)
23         cnt <= cnt + 1'b1;
```

```
24      else
25          cnt <= 26'd0;
26  end
27
28  endmodule
```

这里需要注意的是，源代码前面的序号是为了方便大家查看代码的，在将源代码拷贝到软件编辑区的时候，需要去掉前面的序号，大家也可以直接从我们光盘中提供的源码代码中拷贝，源代码位于资料盘（A 盘）下的 4_SourceCode \ZYNQ_7020(ZYNQ_7010) \1_FPGA_Design \1_led_twinkle \led_twinkle.srcs\sources_1 \new \led_twinkle.v（如果是压缩包的话，需要先解压）。

本章我们只是带领大家熟悉 Vivado 软件的使用流程，不对代码做讲解，在后面的例程中，再来对代码做详细的介绍。

代码编写完成后，软件中显示的界面如下图所示。

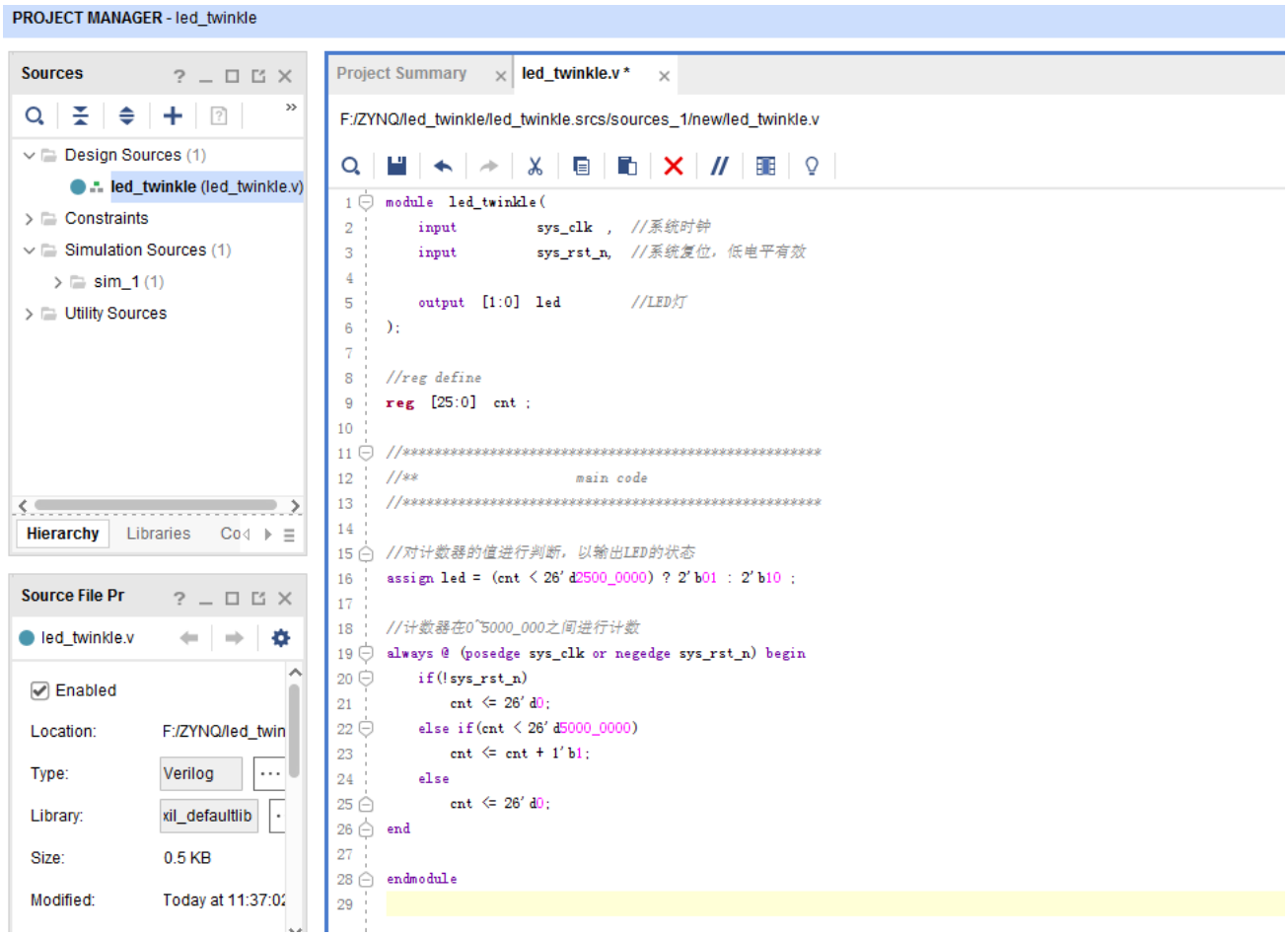


图 4.2.20 Verilog 文件编写完成界面

另外，如果读者认为 Vivado 的 Text Editor 默认的字体比较小，也可以依次点击“Settings”→“Text Editor”→“Fonts and Colors”，在窗口中的“Size”选项中来修改字体大小，默认是 12，我们修改为 20，如下图所示：

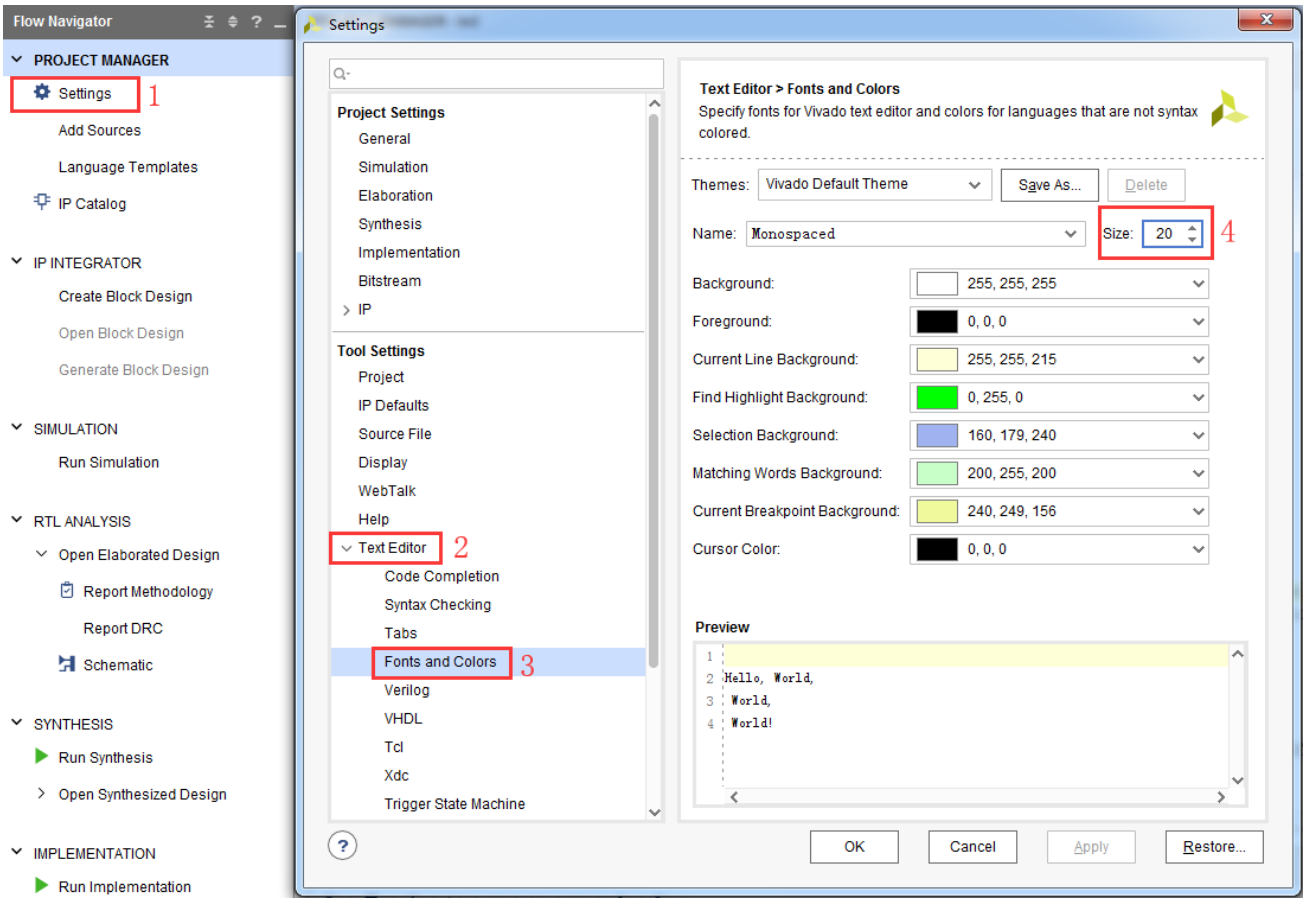


图 4.2.21 修改代码的字体大小

接下来我们单击工作空间中的保存按钮或者按下键盘的“Ctrl+S”，来保存编辑完成后的代码，如下图所示：

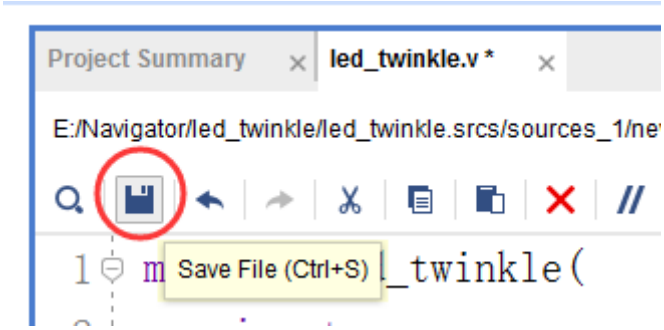


图 4.2.22 保存源文件按钮

每次保存后，Vivado 都会对源文件进行部分语法的检查，如果有语法的错误，Vivado 会给出提示。另外，在大多数情况下，Vivado IDE 会自动识别设计的顶层模块，当然，用户也可以手动指定顶层模块。从“Sources”窗口的右击菜单中选择“Set as Top”来手动定义顶级模块。

4.2.3 分析与综合

代码输入完毕之后，就可以对设计进行分析（Elaborated）了。点击“Flow Navigator”窗口中的“Open Elaborated Design”按钮，如下图所示：

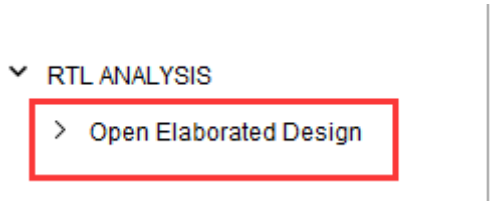


图 4.2.23 “Open Elaborated Design”按钮

此时，Vivado 会编译 RTL 源文件并进行全面的语法检查，并在 Messages 窗口中给出相应的“Error”和“Warning”。如果出现“Error”，则分析失败，用户必须修改设计文件，并重新编译源文件来取消“Error”。如果出现“Warning”，用户也可逐一查看，以确定哪些是设计缺陷，哪些是可以忽略的。打开分析后（Elaborated）的设计，Vivado 会生成顶层原理图视图，并在默认 view layout 中显示设计，如下图所示：

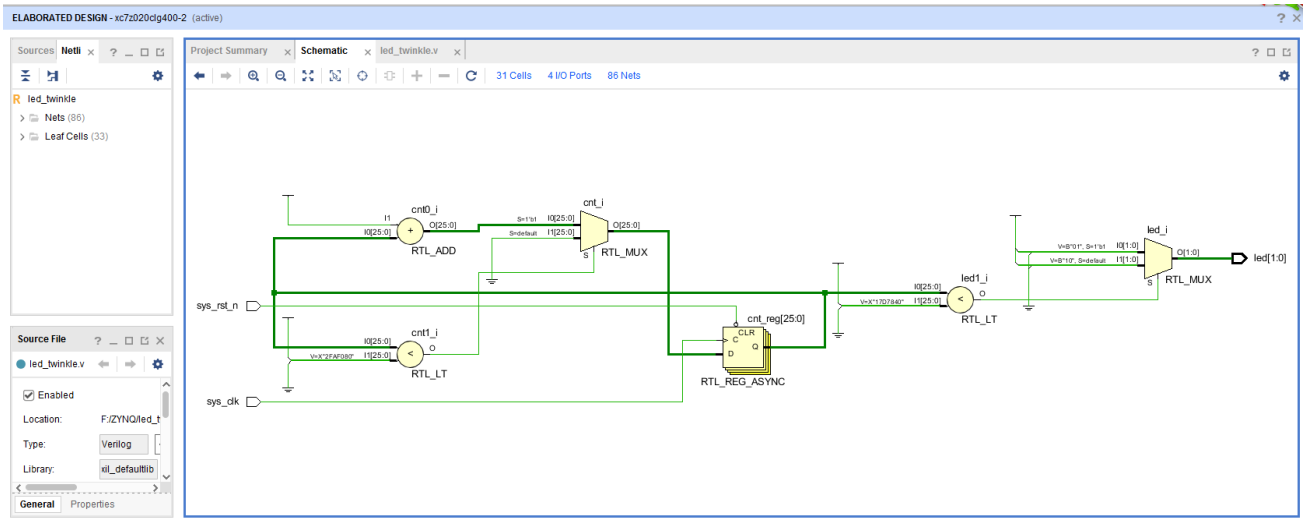


图 4.2.24 打开的 Elaborated Design

可以看到，此时窗口布局已经发生了变化，新增了 Schematic（原理图）、Netlist（网表）等窗口。此时，底部的 Messages 窗口会显示分析阶段产生的消息，如下图所示：

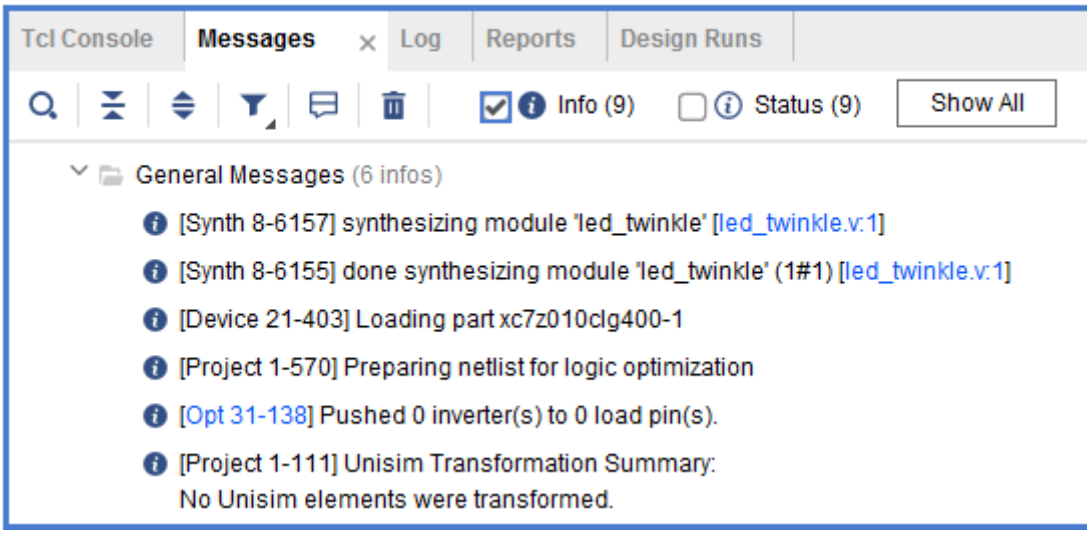


图 4.2.25 Messages 窗口中的消息

可以看出，我们 LED 闪烁实验的代码并没有产生警告或者错误。

此时，我们也可以进行 I/O 引脚分配，在右上角的窗口布局（Layout）选择器中选择“I/O Planing”，如

下图所示：

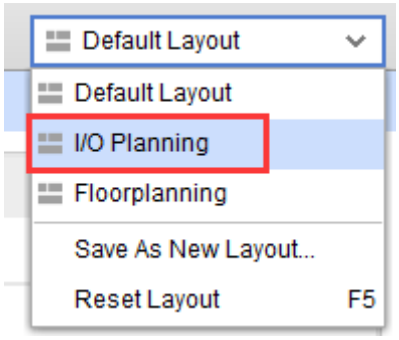


图 4.2.26 点击“I/O Planing”Layout

此时，窗口布局会打开 IO 相关的子窗口，如下图所示：

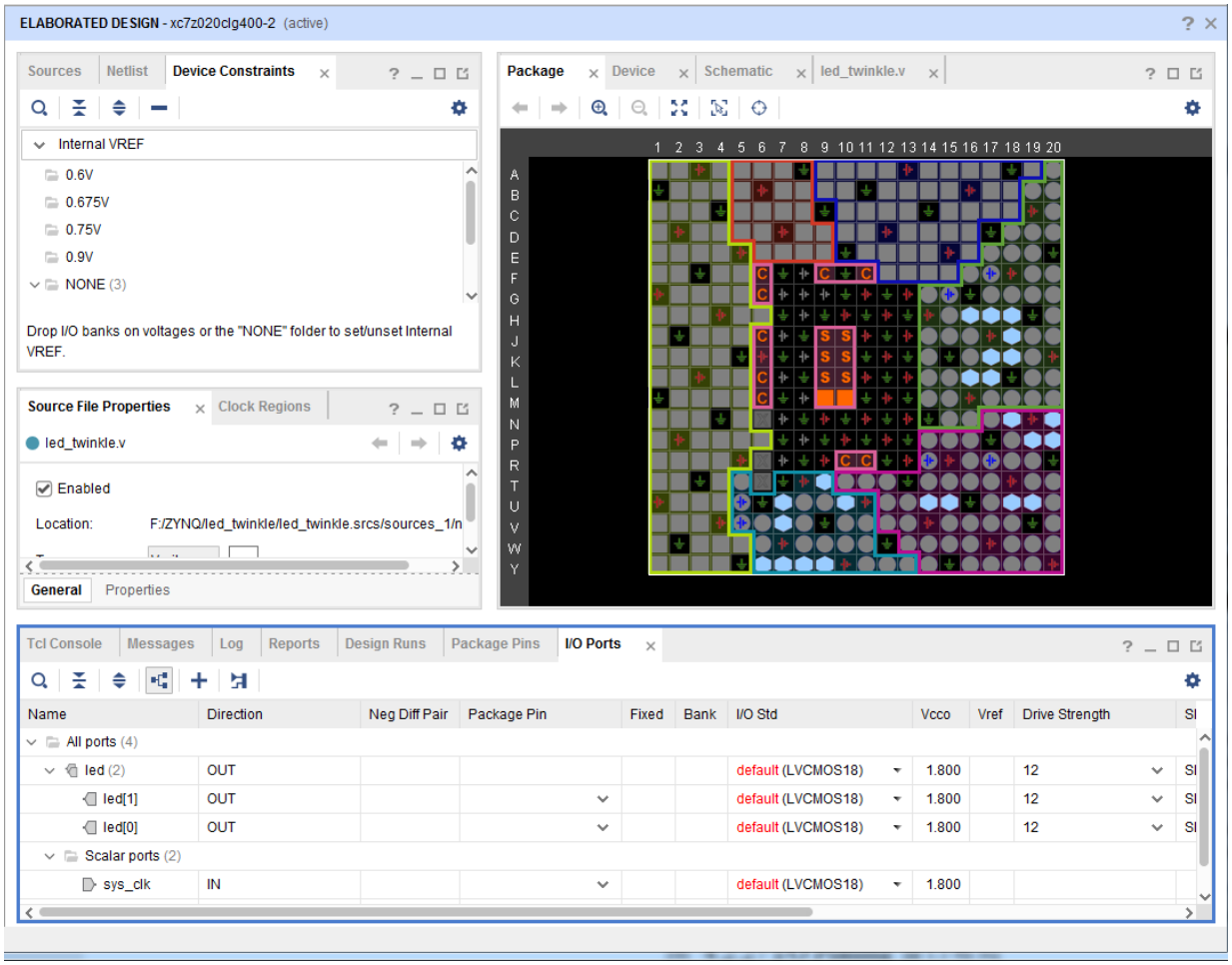


图 4.2.27 I/O Planing 窗口布局

在下方的“I/O Ports”窗口中，就可以进行 IO 的分配了。这里我们暂时不分配，先对设计进行综合，综合之后再统一输入时序约束和 IO 引脚的物理约束。

我们关闭分析后的界面。在工作区域的顶部标题栏处，用鼠标右击，在弹出的命令列表中选择“close”，如下图所示：

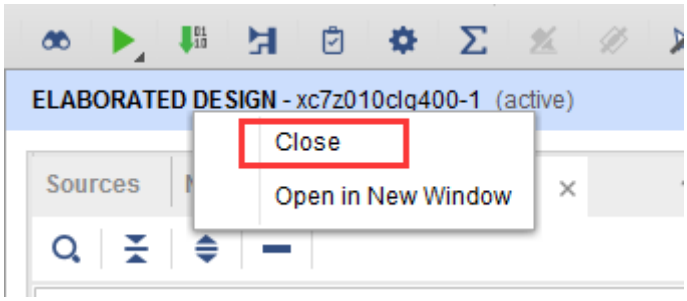


图 4.2.28 关闭分析后的设计

接下来点击“Flow Navigator”窗口中的“Run Synthesis”按钮，来对代码进行综合，如下图所示：

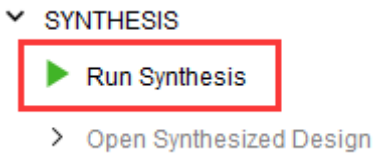


图 4.2.29 “Run Synthesis”按钮

在弹出的窗口中我们直接点击 OK，如下图所示：

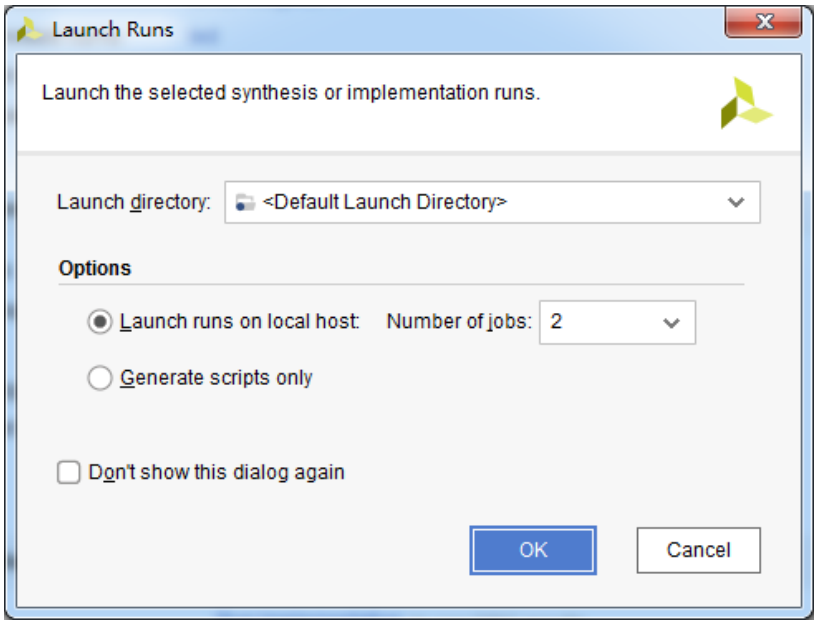


图 4.2.30 Launch Runs 窗口

这时可以看到在“Design Runs”窗口中显示正在综合，如下图所示：

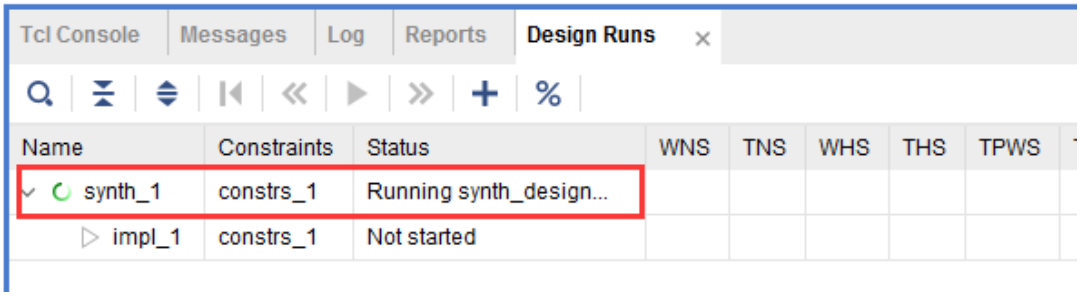


图 4.2.31 “Design Runs”窗口

综合完成后，弹出如下窗口：

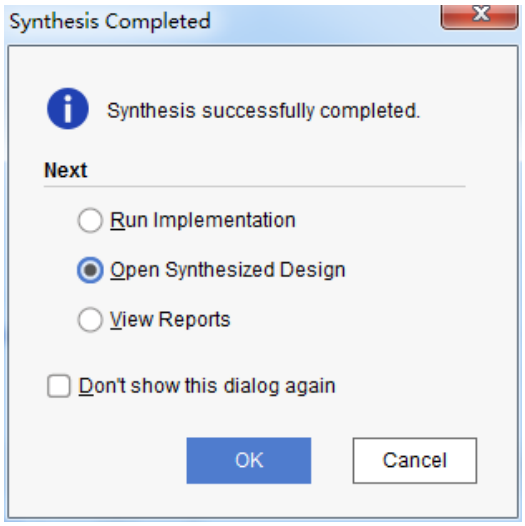


图 4.2.32 综合完成

我们关闭该窗口。接下来进行约束的输入。

4.2.4 约束输入

首先，我们需要先创建一个约束文件。点击“Sources”窗口中的“+”号，在弹出的窗口选择“Add or create constraints”，点击“NEXT”按钮，如下图所示：

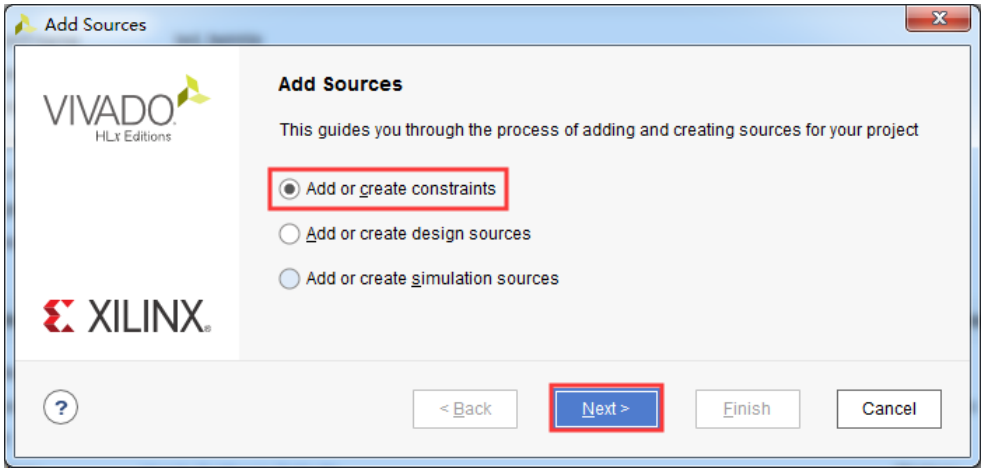


图 4.2.33 添加约束文件

在接下来的界面中点击“Create File”创建一个新的约束文件，如下图所示：

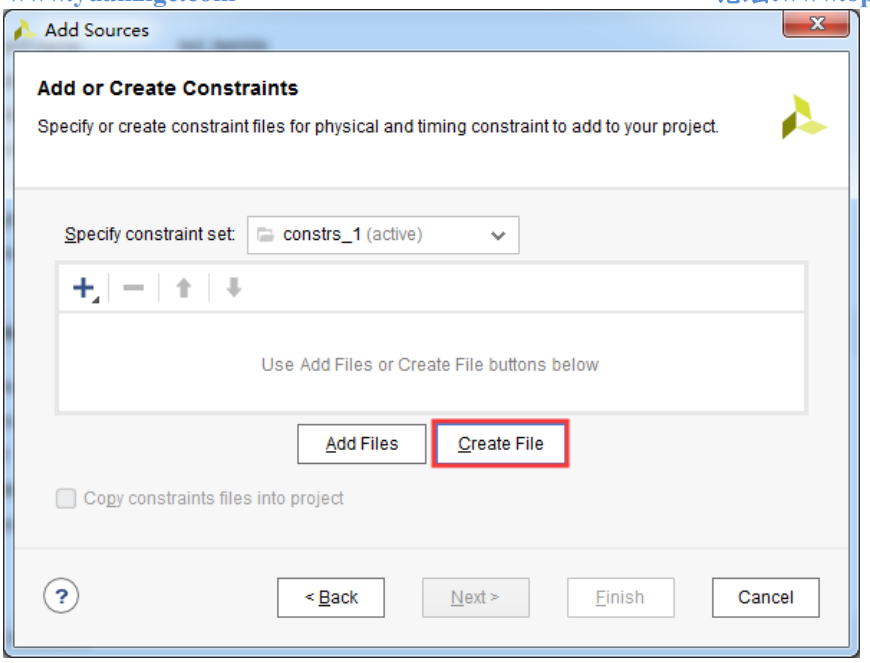


图 4.2.34 Create File

在弹出的对话框中输入约束文件的名称“led_twinkle”，然后点击“OK”按钮，如下图所示：

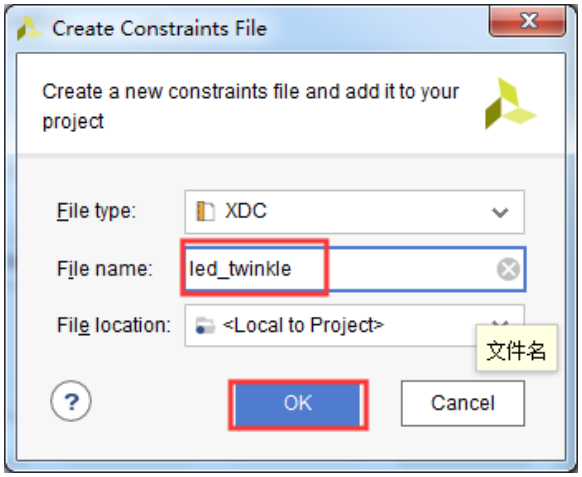


图 4.2.35 输入约束文件的名称

接下来点击“Finish”按钮，完成约束文件的创建，如下图所示：

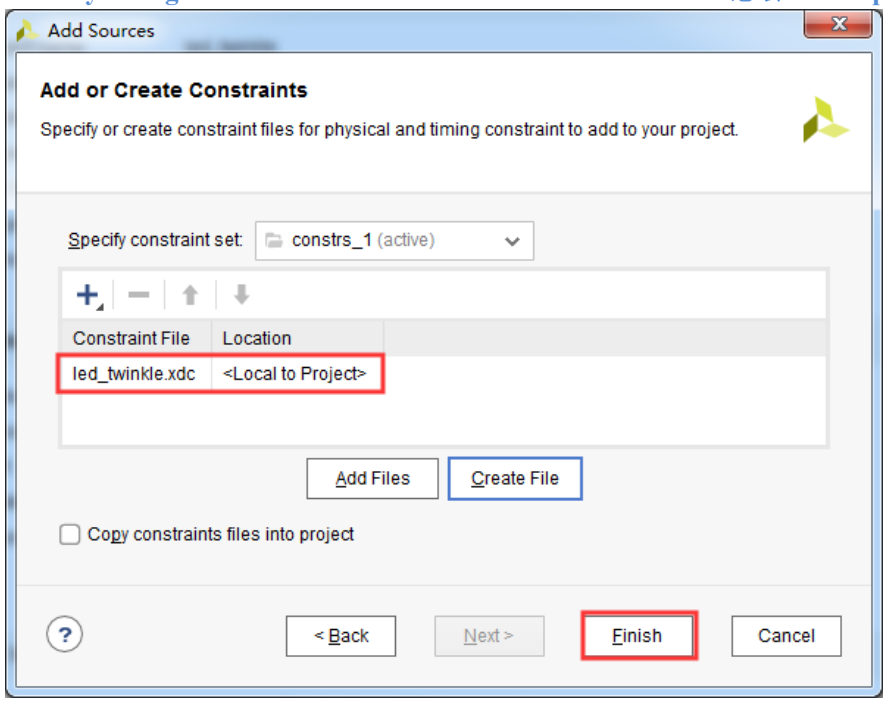


图 4.2.36 点击 Finish

这时我们就可以在“Sources”窗口中看到添加的这个约束文件了，如下图所示：

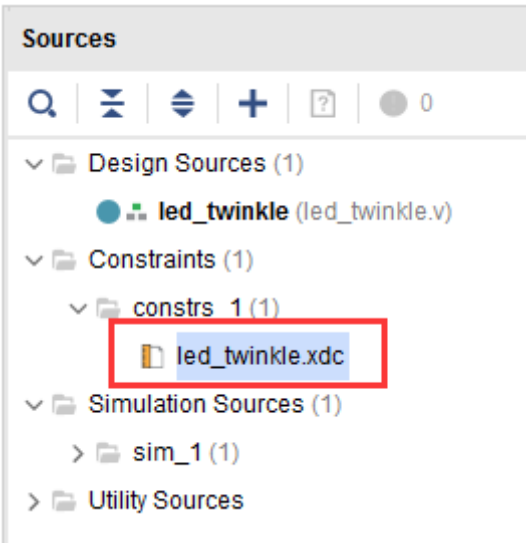


图 4.2.37 添加完成的约束文件

注意，Vivado 的约束文件是以“.xdc”为后缀的文本文件，其中存储的是一条条的 xdc 约束命令。

我们双击打开 led_twinkle.xdc 文件，开始对工程进行约束。约束一般分为两种，一种是 IO 管脚的约束，另外一种是对时序进行约束。我们先来看下 IO 管脚的约束，约束语句如下：

```
#IO 管脚约束
set_property -dict {PACKAGE_PIN U18 IOSTANDARD LVCMOS33} [get_ports sys_clk]
set_property -dict {PACKAGE_PIN J15 IOSTANDARD LVCMOS33} [get_ports sys_rst_n]
set_property -dict {PACKAGE_PIN J18 IOSTANDARD LVCMOS33} [get_ports {led[0]}]
set_property -dict {PACKAGE_PIN H18 IOSTANDARD LVCMOS33} [get_ports {led[1]}]
```

第一行以“#”号开头，表示这是一条注释语句，每一条注释也单独占用一行。第二行是有效的约束命

令，每一个约束命令单独占用一行，命令的结尾不需要像 verilog 代码一样，添加分号“;”结束符号。命令的第一个关键字代表该命令的名称，其后的所有字段都是该命令的参数列表。IO 管脚约束是对端口的引脚位置和电平标准进行约束，例如第二行命令是对系统时钟的管脚进行约束，“set_property”是命令的名称；“PACKAGE_PIN　U18”是引脚位置参数，代表引脚位置是 U18；“[get_ports sys_clk]”代表该约束所要附加的对象是 sys_clk 引脚；“IOSTANDARD　LVCMOS33”代表该引脚所使用的电平标准是 LVCMOS33。

IO 管脚约束比较好理解，就是我们的程序所驱动的 IO 和 ZYNQ 芯片的 IO 对应起来。而时序约束（Timing Constraints）用来描述设计人员对时序的要求，比如时钟频率，输入输出的延时等，以满足设计的时序要求。约束语句如下：

```
#时钟周期约束
create_clock -name clk -period 20 [get_ports sys_clk ]
```

“create_clock”是该命令的名称，它会创建一个时钟；其后的“-name clk”、“-period 20”、“[get_ports sys_clk]”都是该命令的各个参数，分别表示所创建时钟的名称是“clk”、时钟周期是 20ns、时钟源是 sys_clk 端口，一般只对输入的时钟做周期的约束。

对时钟的约束最简单的理解就是，设计者需要告诉 EDA 工具设计中所使用的时钟的频率是多少；然后工具才能按照所要求的时钟频率去优化布局布线，使设计能够在要求的时钟频率下正常工作。本次实验 sys_clk 的时钟频率为 50MHz，周期为 20ns，在做约束时可以等于这个值或者略低于这个值，不建议周期设置的太小，否则软件在布局布线时很难满足这个要求。

其实对于比较简单的设计，可以不对工程做时序约束，即使不进行时序约束，也不影响最终的功能。而当设计变得复杂起来，或者输入的时钟频率比较高的时候，如果不添加时序约束，那么就有可能在验证设计结果的时候出现一些意料之外的情况。由于本次实验较为简单，这里只对工程对 IO 管脚约束，不进行时序约束，led_twinkle.xdc 文件输入 IO 管脚约束语句后，点击“保持”的图标或者按下键盘的“Ctrl+S”进行保存。输入 IO 管脚约束语句后如下图所示：

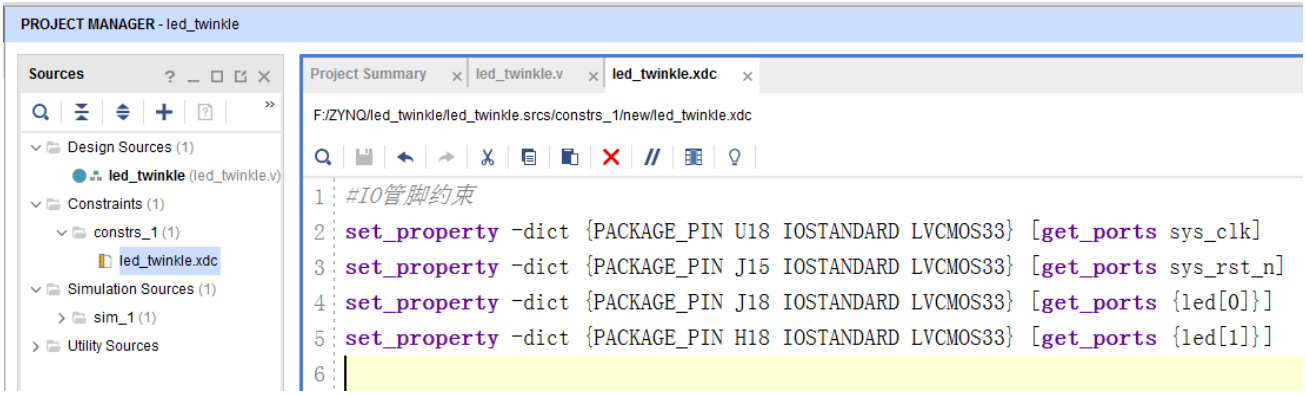


图 4.2.38 输入 IO 管脚约束语句

点击 Text Editor 中的保存按钮后，就完成了约束的输入。

另外，为了便于大家的查看，我们整理出了包含开发板上所有引脚分配的表格和 XDC 约束文件，位于资料盘（A 盘）\3_正点原子 ZYNQ 开发板原理图文件夹下。

4.2.5 设计实现

约束输入完毕之后，就可以开始实现设计了。我们点击“Flow Navigator”窗口中的“Run Implementation”按钮，如下图所示：

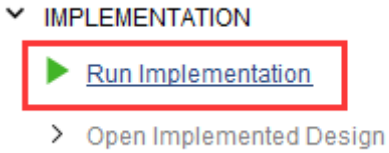


图 4.2.39 点击“Run Implementation”按钮

在弹出的界面中直接点击 OK 即可，如下图所示：

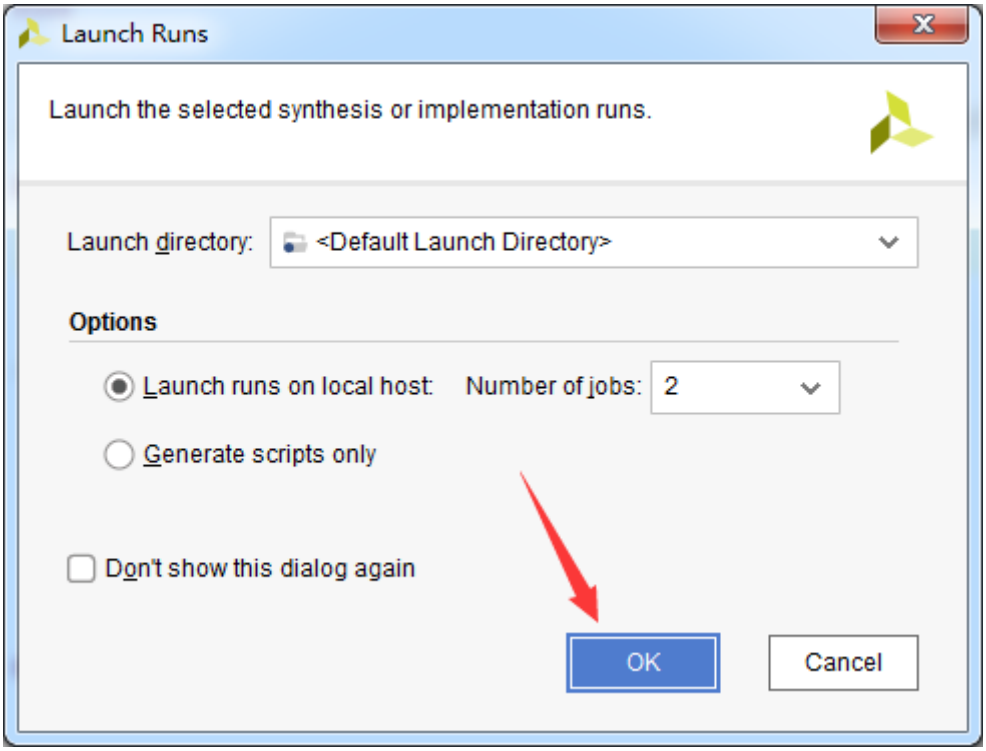


图 4.2.40 开始实现

这时可以看到“Design Runs”窗口中显示正在进行实现，如下图所示：

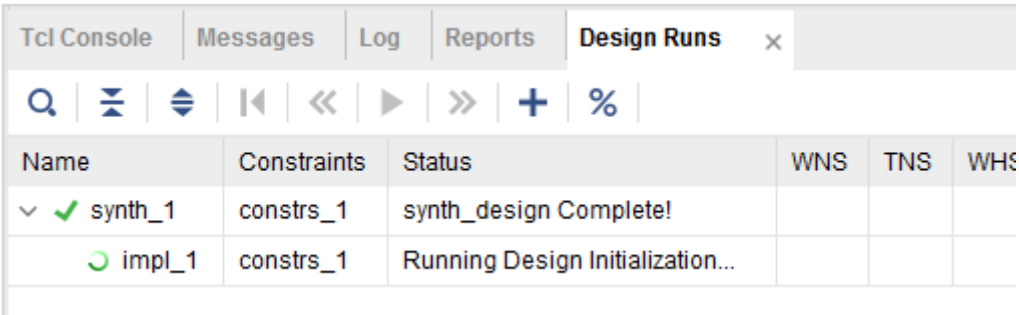


图 4.2.41 正在实现

实现完成后会弹出提示窗口，我们直接点击取消来关闭窗口，如下图所示：

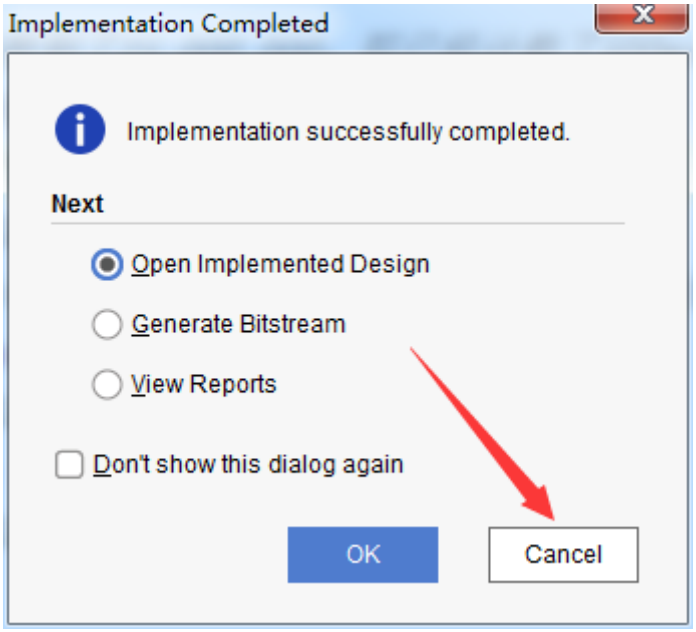


图 4.2.42 实现完成

这时我们再次查看“Design Runs”窗口中的实现结果，如下图所示：

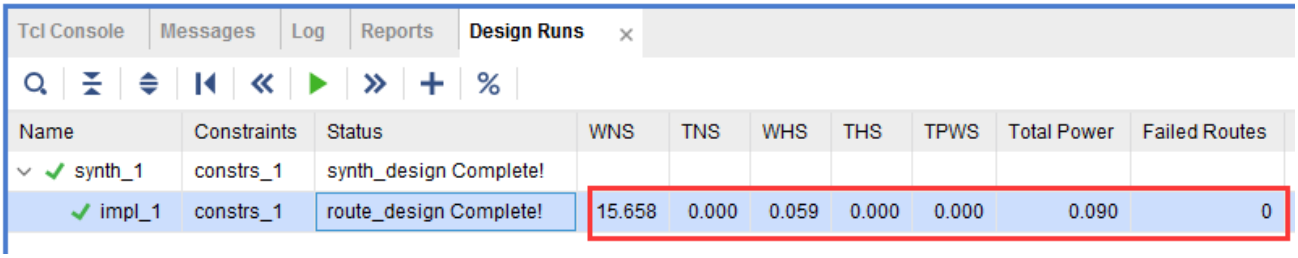


图 4.2.43 实现结果

4.2.6 下载比特流

在下载程序之前，首先要先生成用于下载到器件中的比特流文件，该文件的后缀为“.bit”。我们点击“Flow Navigator”窗口中的“Generate Bitstream”按钮，如下图所示：

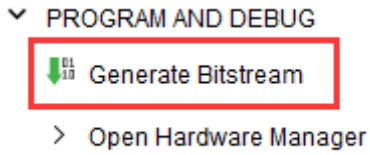


图 4.2.44 “Generate Bitstream”按钮

在弹出的窗口中直接点击 OK，如下图所示：

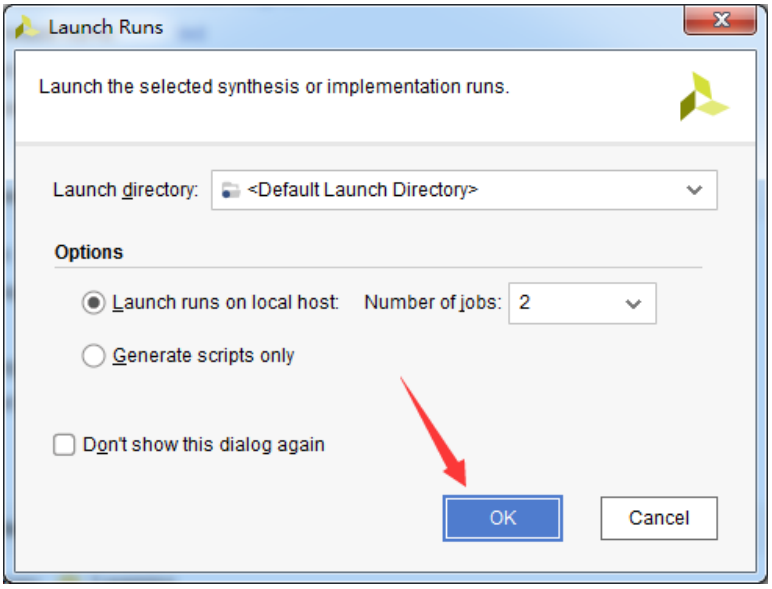


图 4.2.45 开始生成比特流

此时我们可以看到在“Design Runs”窗口中显示正在生成比特流，如下图所示：

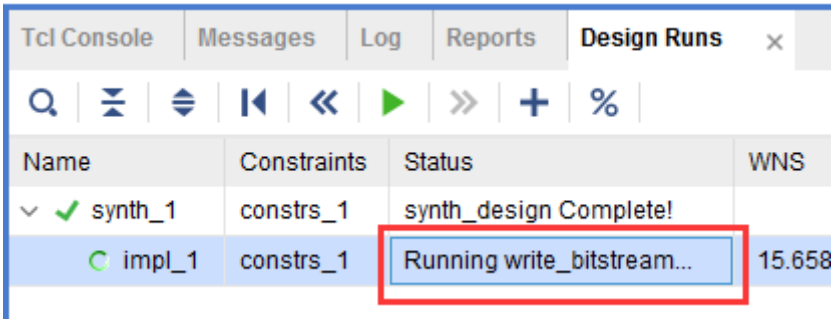


图 4.2.46 正在生成比特流

比特流生成完毕之后，Vivado 会弹出提示窗口，我们点击取消关闭该窗口：

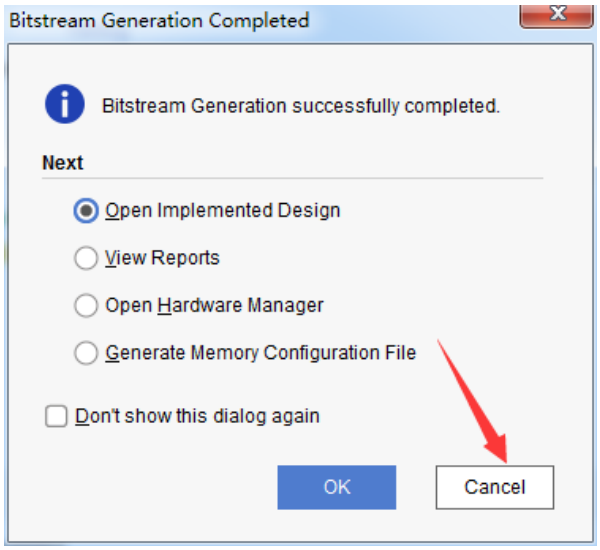


图 4.2.47 比特流生成完毕

接下来我们开始下载比特流，点击“Flow Navigator”窗口中的“Open Hardware Manager”按钮，如下

图所示：

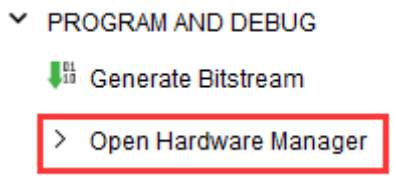


图 4.2.48 “Open Hardware Manager”按钮

接着 Vivado 就会打开 Hardware Manager，同时窗口布局也跟着发生了变化，如下图所示：

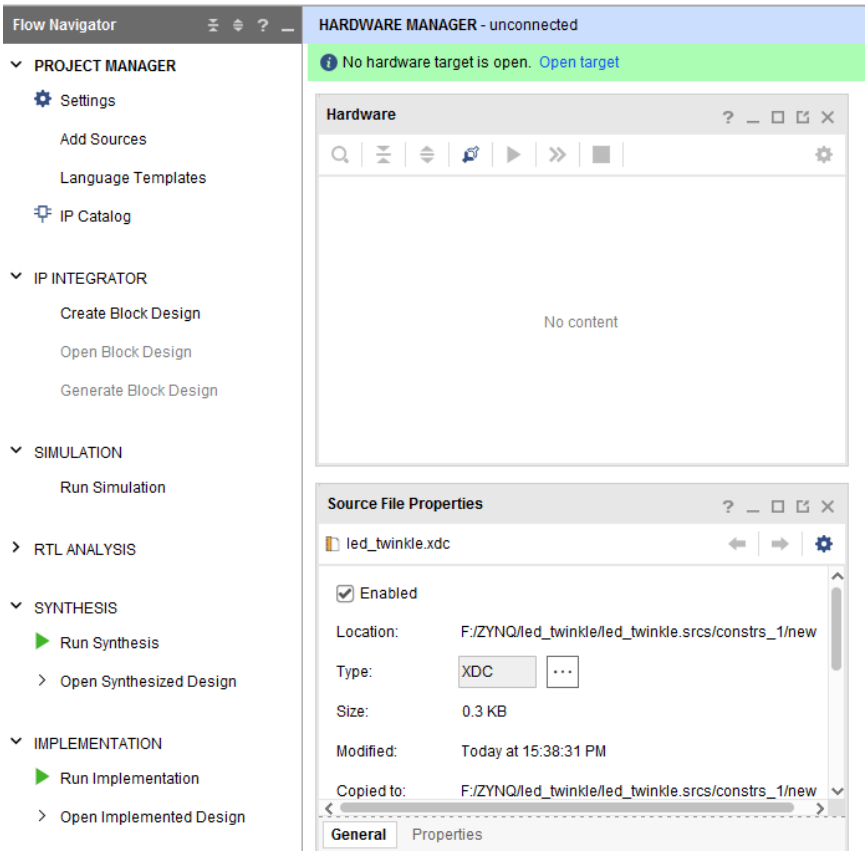


图 4.2.49 Hardware Manager 窗口布局

接下来我们就要用到开发板和 Xilinx 下载器了。首先将 Xilinx 下载器一端连接电脑，另一端与开发板上的 JTAG 接口相连接；然后连接开发板电源线，并打开电源开关。下图为启明星开发板的实物连接图：

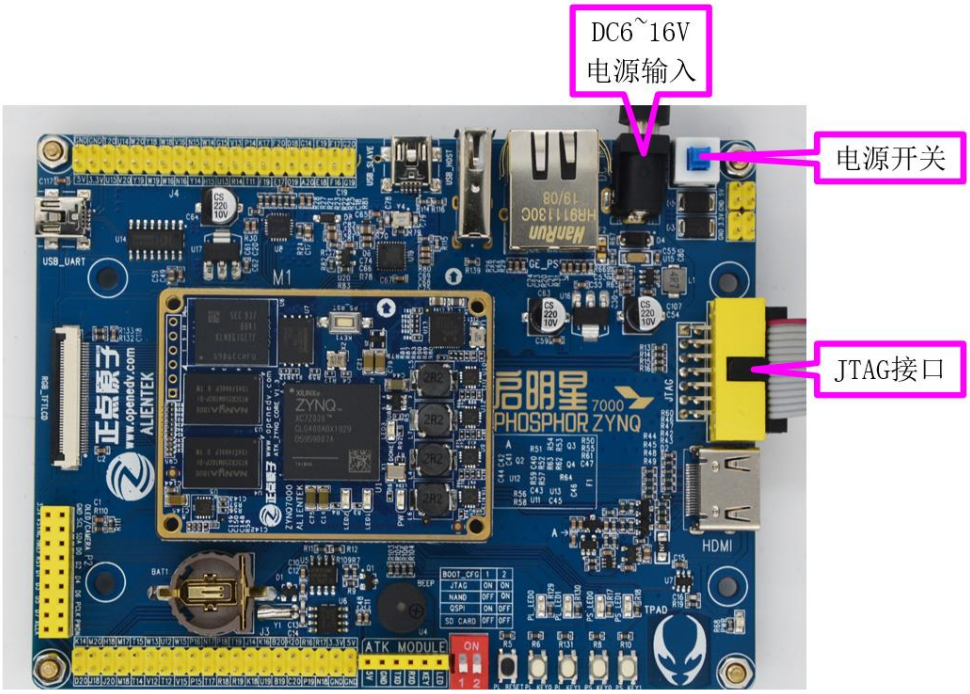


图 4.2.50 启明星开发板连接实物图

开发板完成并打开电源开关后，点击“Hardware”子窗口中的“Auto Connect”按钮，如下图所示：

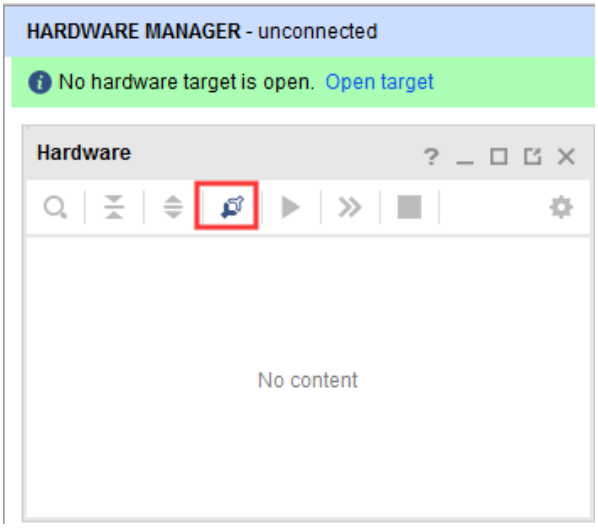


图 4.2.51 “Auto Connect”按钮

在“Hardware”子窗口中出现如下界面就表示 Vivado 就已经和下载器连接成功了，如下图所示：

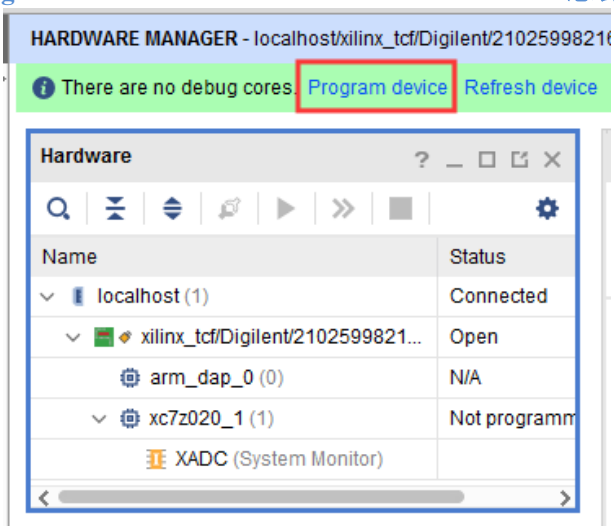


图 4.2.52 与下载器连接成功

我们点击上图中的“Program Device”，弹出的界面如下图所示：

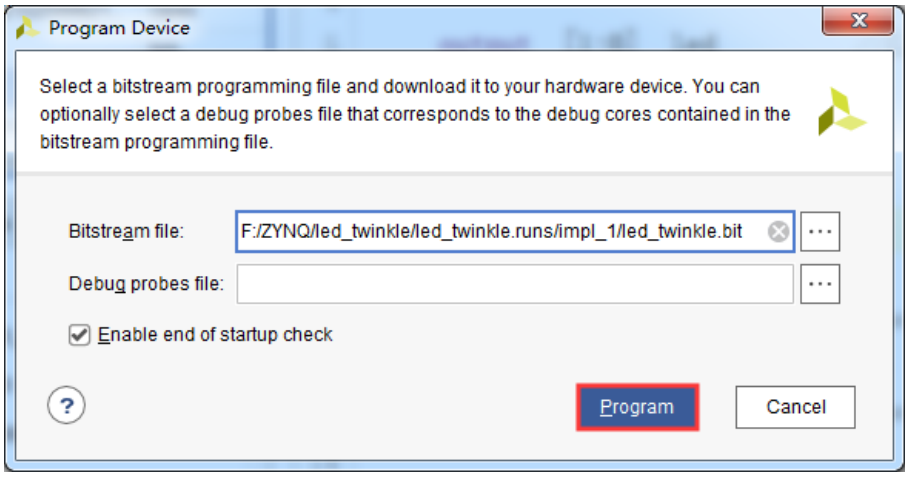


图 4.2.53 下载比特流界面

此时 Bitstream File 一栏会自动识别到工程的比特流文件，我们直接点击“Program”按钮下载程序，程序下载完成后，我们可以看到位于核心板上的 LED 灯在不断地闪烁了，如下图所示：

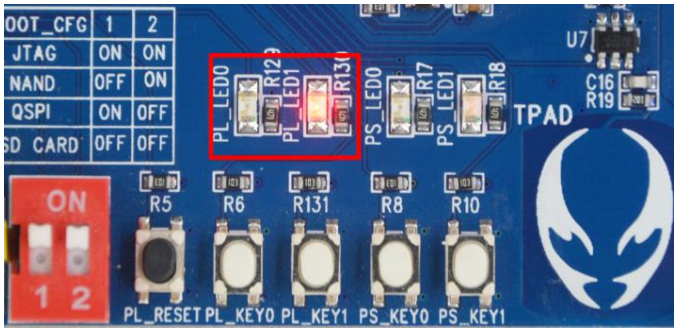


图 4.2.54 两个 PL LED 灯交替闪烁

需要说明的是，下载完比特流后，如果开发板断电，程序会丢失。如果想要程序断电不丢失的话，需要将程序固化至开发板中，这个需要在嵌入式 SDK 软件中完成，ZYNQ 芯片无法单独固化比特流文件。在《领航者 ZYNQ 之嵌入式开发指南.pdf》文档中“第七章 程序固化实验”，会有一个单独的章节向大家介绍程序固化的方法。

4.3 在线逻辑分析仪的使用

传统的 FPGA 板级调试是由外接的逻辑分析仪连接到 FPGA 的控制管脚，然后将内部信号引出至引脚 IO 上，进行板级调试。这种方法的缺点是首先我们需要一个逻辑分析仪，而逻辑分析仪一般价格都比较昂贵，且对于需要测试几十个引脚的时候，选择使用外接的逻辑分析仪就比较繁琐了。在线逻辑分析仪克服了以上所有的缺点，其借用了传统逻辑分析仪的理念以及大部分的功能，并利用 FPGA 中的逻辑资源，将这些功能植入到 FPGA 的设计当中。一般地，在线逻辑分析仪的应用原理框图如下图所示：

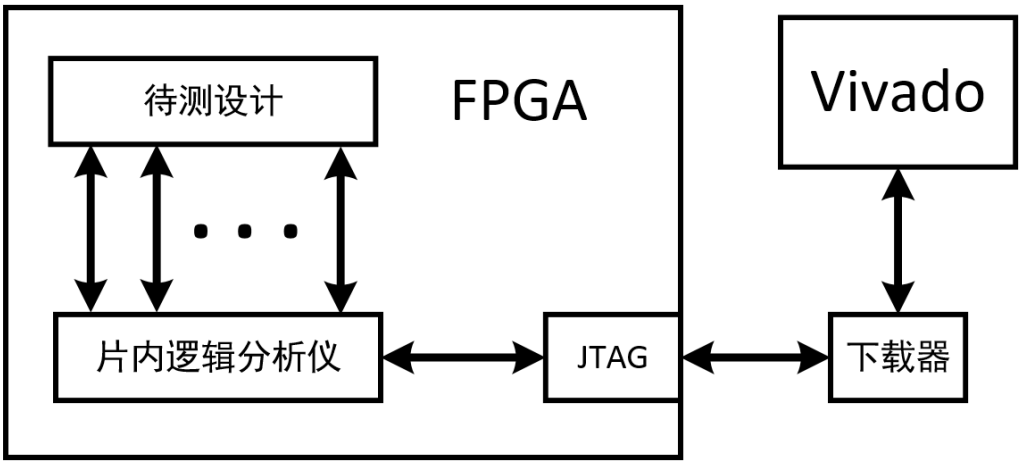


图 4.3.1 在线逻辑分析仪的应用原理框图

其中，待测设计（Design Under Test，DUT）就是用户逻辑，它和片内的在线逻辑分析仪都位于 FPGA 中。在线逻辑分析仪通过一个或多个探针（Probe）来采集希望观察的信号，然后通过片内的 JTAG 硬核组件，来将捕获到的数据传送给下载器，进而上传到 Vivado IDE 以供用户查看。Vivado IDE 也能够按照上述数据路径，反向地向 FPGA 中的在线逻辑分析仪传送一些控制信息。由此可见，在线逻辑分析仪不需要将待测信号引出至 I/O 上，也不需要电路板走线或者探测点，当然更不需要外部的逻辑分析仪的花费，在 Vivado 中就可以将在线逻辑分析仪添加到设计中。但是，在线逻辑分析仪会占用一定数量的内部逻辑资源，如块 RAM、查找表、触发器等等。

在 Vivado 中，在线逻辑分析仪的功能被称为“集成逻辑分析器（Integrated Logic Analyzer，ILA）”，它以 IP 核的形式来加入到用户设计中。Vivado 提供了三种具有不同集成层次的插入 ILA 方法，以满足不同 Vivado 用户群的不同需求：

第一种方法是直接在 HDL 代码中例化一个 ILA IP 核，也被称为“HDL 实例化调试探针流程”，这是集成层次最高的方法。ILA IP 核可以在 IP Catalog（IP 目录）中找到，并对其进行配置，以符合所需的调试需求。这是最直接的方法，但其灵活性也较差。在调试工作完毕之后，还需要在 HDL 源代码中删除 ILA IP 核，然后重新综合并实现，以生成最终的比特流。

第二种方法是在综合后的网表中，分别标记要进行调试观察的各个信号，然后通过一个简单的“Setup Debug”向导来设置各个探针和 ILA IP 核的工作参数，然后工具会根据用户设置的参数，自动地生成各个 ILA IP 核。这个方法也被称为“网表插入调试探针流程”。在此流程中，用户不需要修改 HDL 源代码，并且能够单独控制每个 ILA IP 核以及每个探针，这样就提供了很大的灵活性。用户设置的调试信息会以 Tcl XDC 调试命令的形式保存到 XDC 约束文件中，在实现阶段，Vivado 会读取这些 XDC 调试命令，并在布局布线时加入这些 ILA IP 核。在调试工作完毕之后，用户就可以在综合后的网表中删除 ILA IP 核，或者在 XDC 文件中删除调试命令，然后再对设计进行实现，以生成最终的比特流。

第三种方法是手动地在 XDC 约束文件中书写对应的 Tcl XDC 调试命令，在实现阶段工具会自动读取这

些命令，并在布局布线时加入这些 ILA IP 核。在调试工作完毕之后，用户还需要在 XDC 约束文件中删除这些命令，然后实现最终的设计。这种方法集成层次最低，一般不会使用这种方法。

本小节同样以 LED 灯闪烁实验的工程为例，介绍前两种 ILA 调试流程，即“HDL 实例化调试探针流程”和“网表插入调试探针流程”。

4.3.1 HDL 实例化调试探针流程

“HDL 实例化调试探针流程”需要在 HDL 源代码中实例化 ILA IP 核。我们点击“Flow Navigator”窗口中的“IP Catalog”按钮，如下图所示：

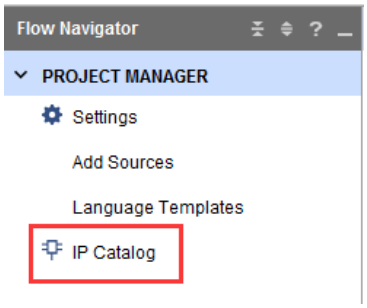


图 4.3.2 “IP Catalog”按钮

这时“IP Catalog”窗口就被打开了，如下图所示：

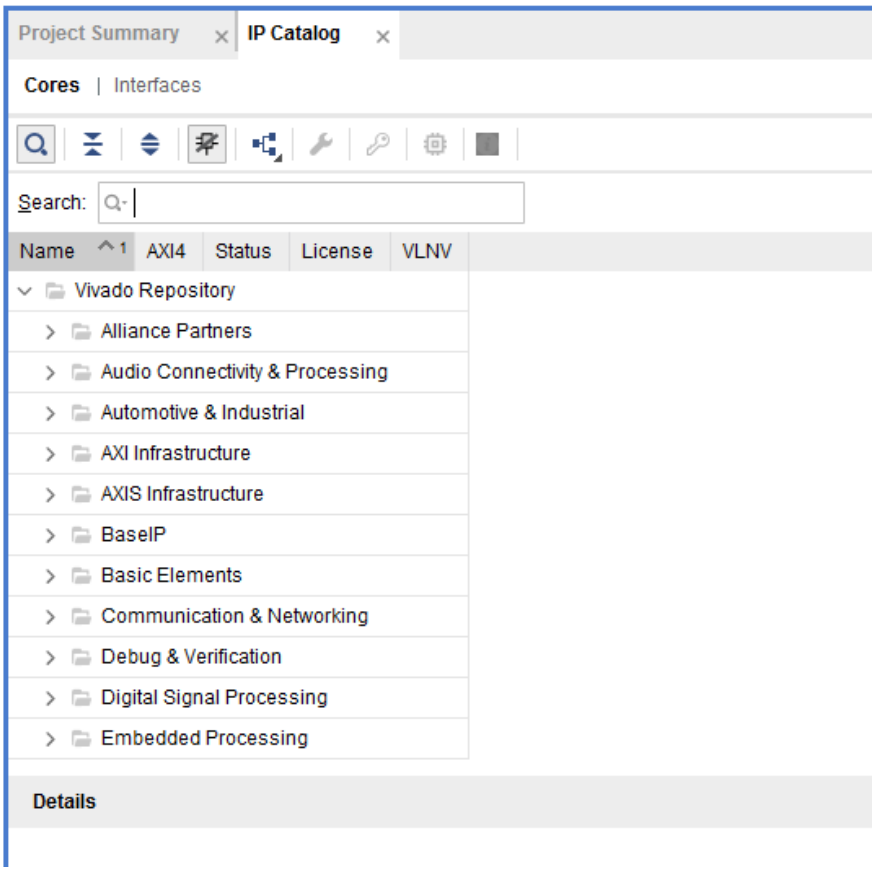


图 4.3.3 “IP Catalog”窗口

我们在搜索栏中输入“ILA”，这时 Vivado 会自动根据关键词搜索出相应的结果，我们双击“ILA（Integrated Logic Analyzer）”，如下图所示：

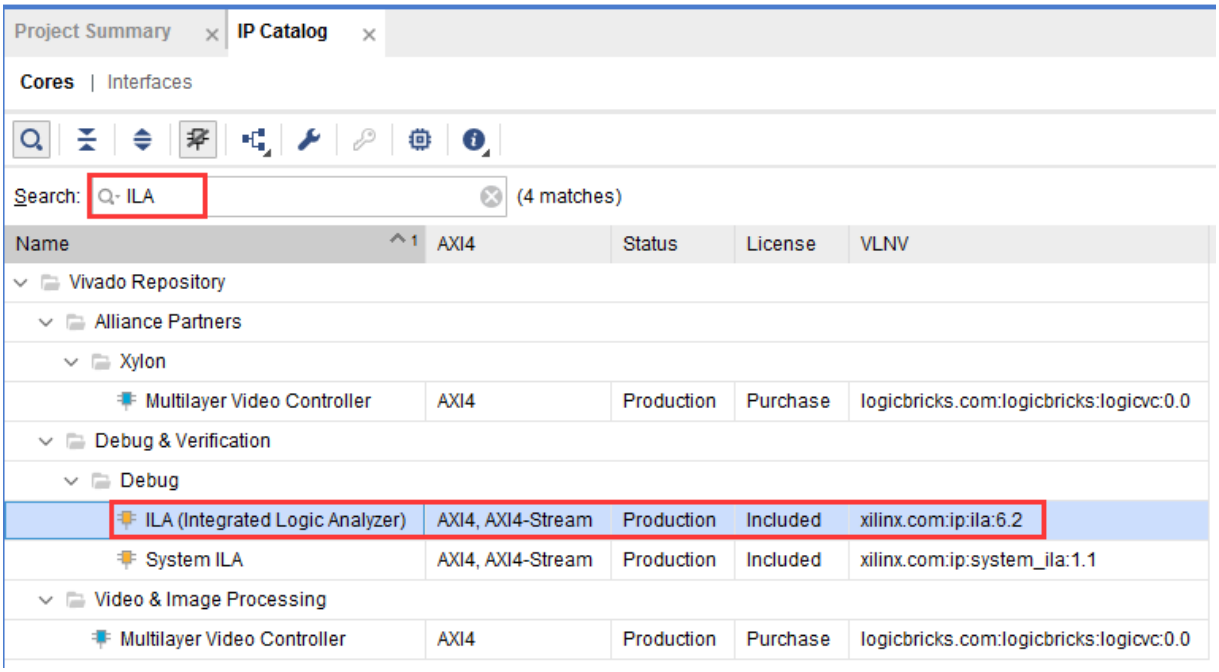


图 4.3.4 搜索 ILA

接下来会弹出“ILA IP”核的配置页面，如下图所示：

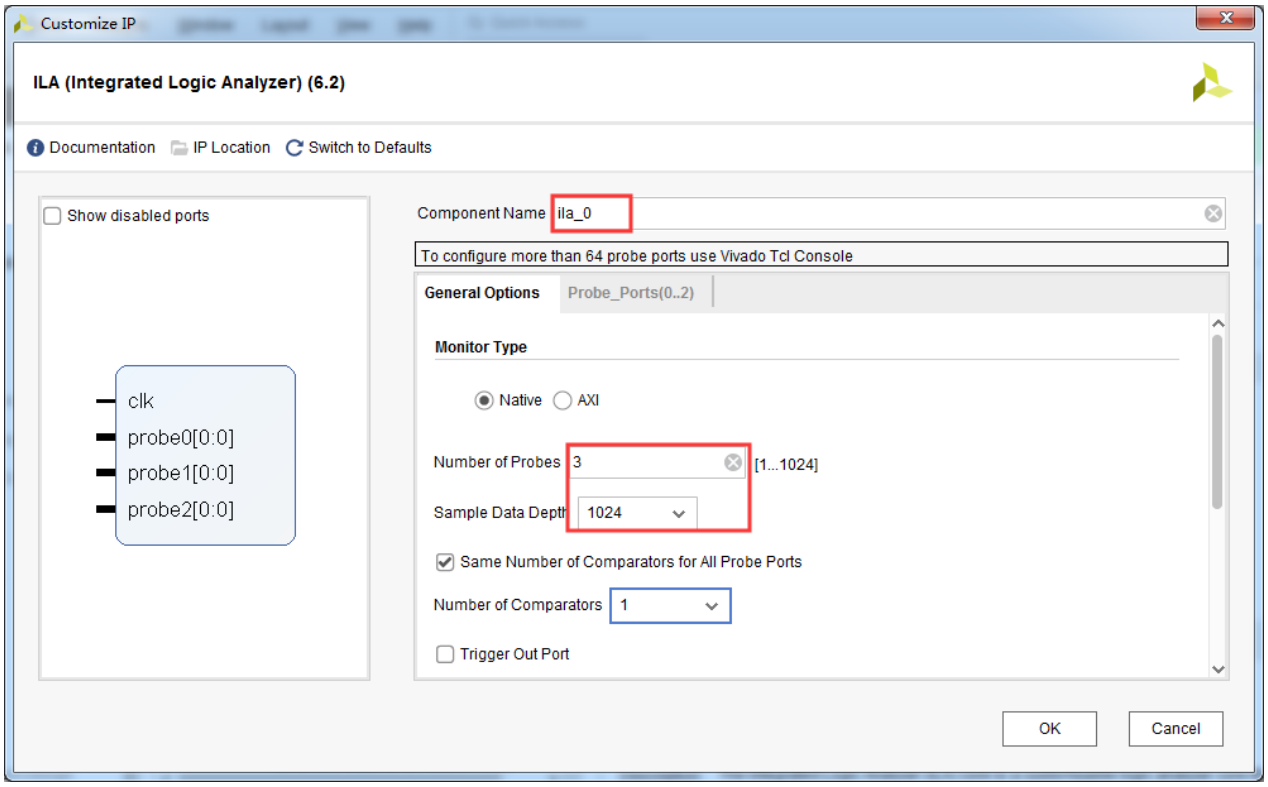


图 4.3.5 “Customize IP”窗口

“Component Name”一栏用于设置 ILA IP 核的名称。

IP 核的配置包含两个子页面，分别是“General Options”和“Probe_Ports(0..0)”，“General Options”页面用于设置 ILA IP 核的总体设置，“Probe_Ports(0..0)”页面用于对每个探针的参数进行设置。

在“General Options”页面中，“Number of Probes”一栏用于设置所需的探针数量，一般地，一个探

针用于连接一个待测信号，例如，我们想要观察“sys_rst_n”、“led”和“cnt”这三个信号，则需要把“Number of Probes”设置为 3；

“Sample Data Depth”用于设置采样深度，在每个采样时钟下，ILA 都会将捕获到的探针信号的值送入 RAM 中，由于 RAM 的存储空间是有限的，所以此选项就用于设置 RAM 最大存储多少个探针信号的值，我们保持其默认 1024，其数值越大，消耗的 RAM 资源也越多。其它选项保持默认即可。

在“Probe_Ports(0..0)”页面中，用于设置每个探针的参数，一般我们只需设置探针宽度“Probe Width [1..4096]”一栏即可，由于“sys_rst_n”、“led”和“cnt”这三个信号的位宽分别是 1 位、2 位和 26 位，所以我们需要将其分别设置为 1、2 和 26，然后点击“OK”按钮即可，如下图所示：

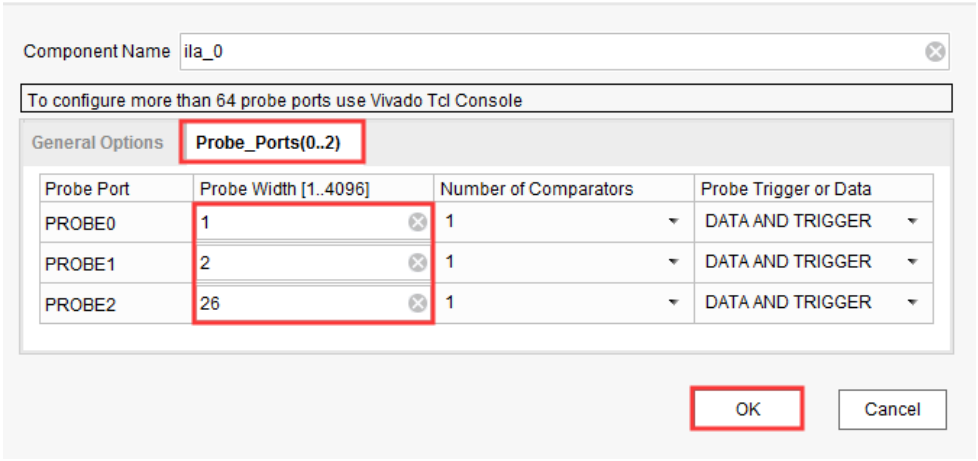


图 4.3.6 设置探针宽度

在弹出的对话框中直接点击 OK 即可，如下图所示：

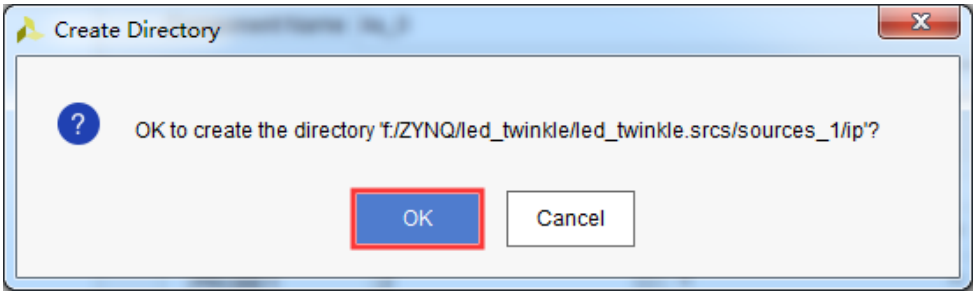


图 4.3.7 点击 OK

接下来就会弹出“Generate Output Products”对话框，我们保持默认设置，直接点击“Generate”即可，此时 Vivado 就开始对该 ILA IP 核进行 OOC 综合了。如下图所示：

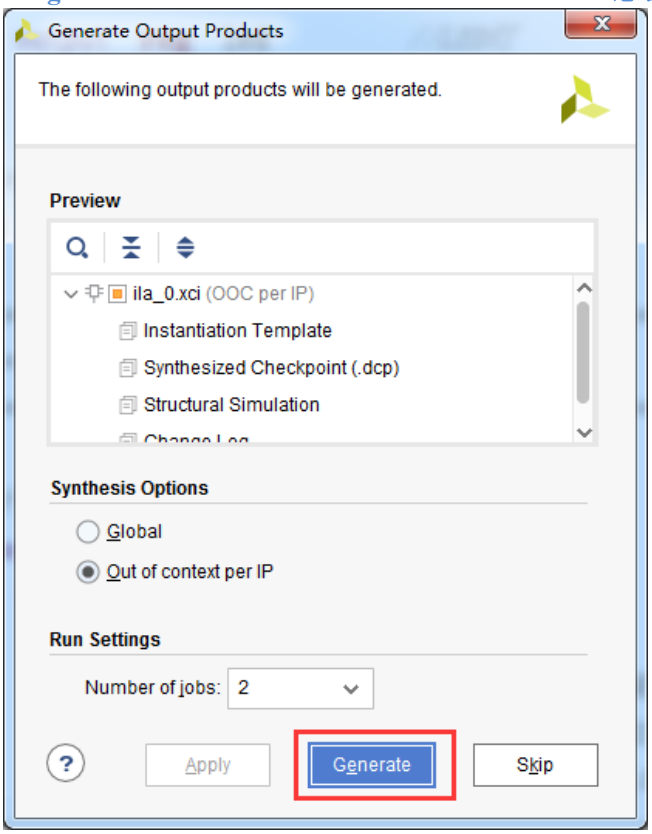


图 4.3.8 开始为 ILA IP 核生成输出文件

在这里，我们简要介绍一下 Vivado 的 OOC（Out-of-Context）综合的概念。对于顶层设计，Vivado 使用自顶向下的全局（Global）综合方式，将顶层之下的所有逻辑模块都进行综合，但是设置为 OOC 方式的模块除外，它们独立于顶层设计而单独综合。通常，在整个设计周期中，顶层设计会被多次修改并综合。但有些子模块在创建完毕之后不会因为顶层设计的修改而被修改，如 IP，它们被设置为 OOC 综合方式。OOC 模块只会在综合顶层之前被综合一次，这样在顶层的设计迭代过程中，OOC 模块就不必跟随顶层模块，而一次次的产生相同结果的多余综合了。所以，OOC 流程减少了设计的周期，并消除了设计迭代，使您可以保存和重用综合结果。

Out-of-Context(OOC) 综合是一种自底向上的设计流程，默认情况下，Vivado 设计套件使用 OOC 的设计流程来综合 OOC 模块。OOC 模块可以是来自 IP Catalog 的 IP、来自 Vivado IP Integrator 的 block design 或者顶层模块下手动设置为 OOC 方式的任何子模块。

来自 IP Catalog 的 IP 就默认使用 OOC 的综合方式，例如上图中的“Synthesis Options”选项就设置为了“Out of Context Per IP”。这些 IP 会在顶层的全局综合之前，单独地进行 OOC 综合并生成输出产品（Generate Output Products），包括综合后的网表等各种文件。在对顶层进行综合时，OOC 模块会被视为黑盒子，并且不会参与到顶层的综合中来。在综合之后的实现过程中，OOC 模块的黑盒子才会被打开，这时其网表才是可见的，并参与到全局设计的实现过程中来。

介绍完了 OOC 综合的概念，现在我们回到 ILA 的调试流程中。在“Design Runs”窗口中可以看到 Vivado 正在为 ILA IP 核运行 OOC 综合，如下图所示：

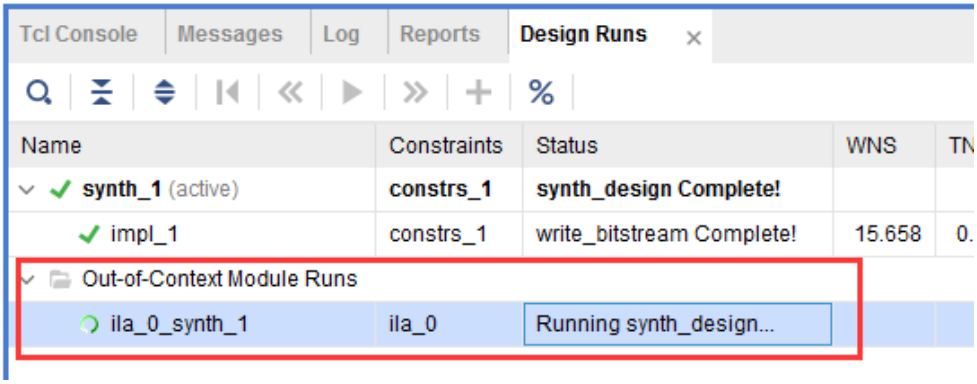


图 4.3.9 ILA IP 核正在进行 OOC 综合

OOC 综合完毕之后如下图所示：

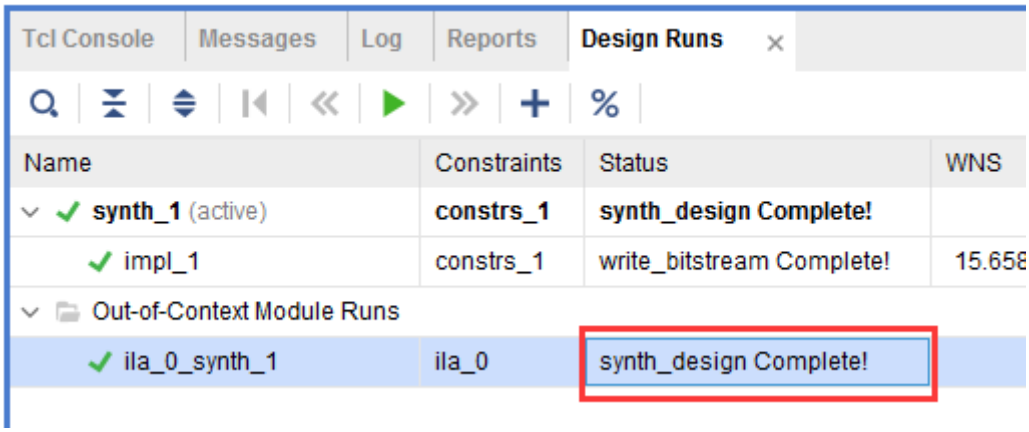


图 4.3.10 OOC 综合完毕

这时可以在“Source”窗口中看到已经出现了 ILA IP 核，如下图所示：

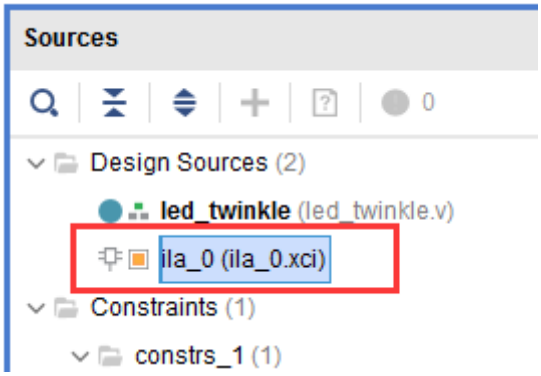


图 4.3.11 添加的 ILA IP 核

由于我们还没有把它例化到顶层的 HDL 代码中，所以在层次结构上它与顶层并排。下面我们将其例化到顶层的 HDL 代码中。在“Source”窗口中的“IP Sources”选项卡中双击 ILA IP 核的例化模板文本文件，找到例化模板的内容，如下图所示：

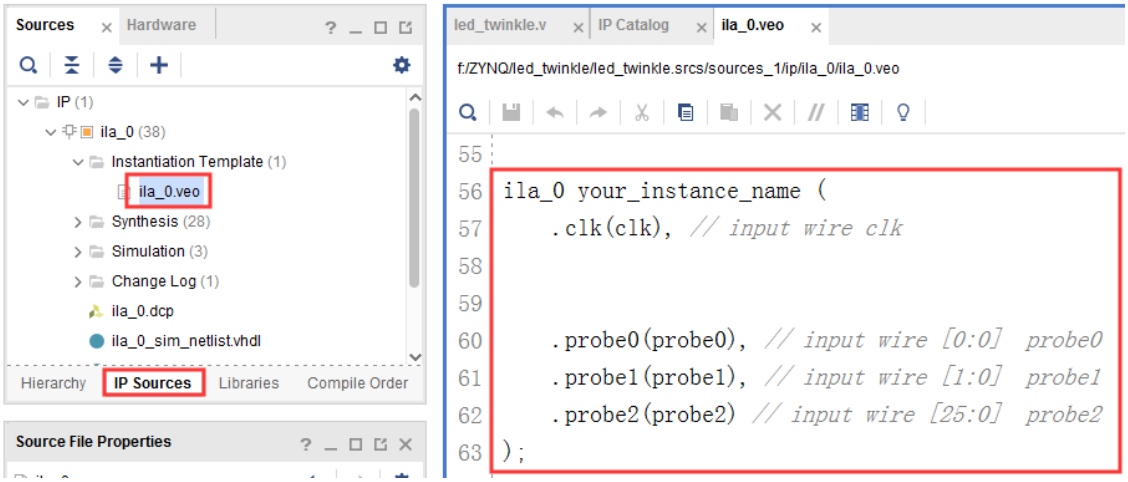


图 4.3.12 例化模板

将图中的红色方框中的模板代码复制并粘贴到 led_twinkle.v 顶层 HDL 代码中，并将 ILA 的时钟和探针信号连接到顶层设计中，例化 ILA IP 核的代码如下：

```
ila_0 u_ila_0 (
    .clk(sys_clk),          // input wire clk

    .probe0(sys_rst_n),     // input wire [0:0]  probe0
    .probe1(led),           // input wire [1:0]  probe1
    .probe2(cnt)            // input wire [25:0]  probe2
);
```

我们将 ILA 的时钟连接到了顶层时钟 sys_clk 信号上，probe0 探针连接到了 sys_rst_n，probe1 探针连接到了 led，probe2 探针连接到了 cnt。代码修改完成后如下图所示：

```
22      else if(cnt < 26'd5000_0000)
23          cnt <= cnt + 1'b1;
24      else
25          cnt <= 26'd0;
26  end
27
28  ila_0 u_ila_0 (
29      .clk(sys_clk), // input wire clk
30
31      .probe0(sys_rst_n), // input wire [0:0]  probe0
32      .probe1(led), // input wire [1:0]  probe1
33      .probe2(cnt) // input wire [25:0]  probe2
34  );
35
36  endmodule
```

图 4.3.13 例化 ILA IP 核

保存源文件之后就可以直接综合并实现设计，最后生成比特流。

至此，我们就成功地使用“HDL 实例化调试探针流程”将 ILA IP 核添加到了设计中。接下来就可以将

比特流下载到 FPGA 中，并对信号进行在线观察。

4.3.2 Hardware Manager 中观察调试信号

生成比特流之后，我们打开 Hardware Manager，连接到开发板，并下载比特流，如下图所示：

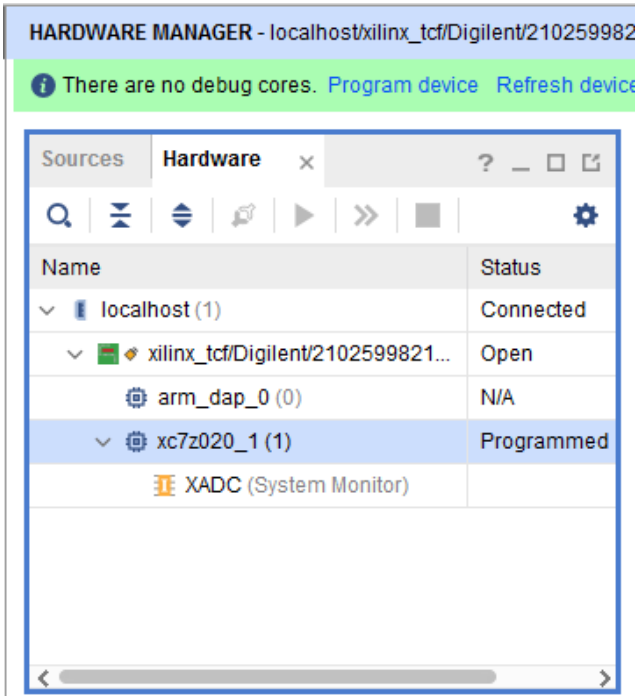


图 4.3.14 下载比特流

在弹出的窗口中，Vivado 会自动识别比特流文件和具有调试探针信息的.ltx 文件，如下图所示：

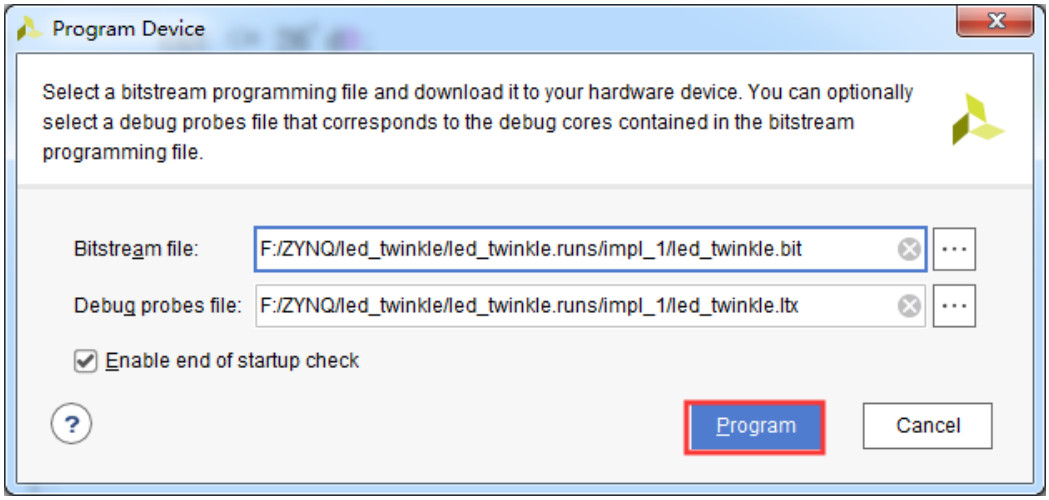


图 4.3.15 下载比特流和调试文件

.ltx 文件存储了调试探针的信息，用来传递给 Vivado IDE，它是从我们的设计中被提取出来的。通常，调试探测文件是在实现过程中自动创建的，并位于和比特流文件相同的目录下。若实现后的设计中包含了 ILA IP 核，则在下载比特流时，Vivado 会自动识别出.ltx 文件。这里我们不必太多地关注此文件，只要按照 Vivado 自动的设定来下载设计即可。

我们直接点击“Program”，此时 Vivado 会自动打开 ILA 的调试窗口，如下图所示：

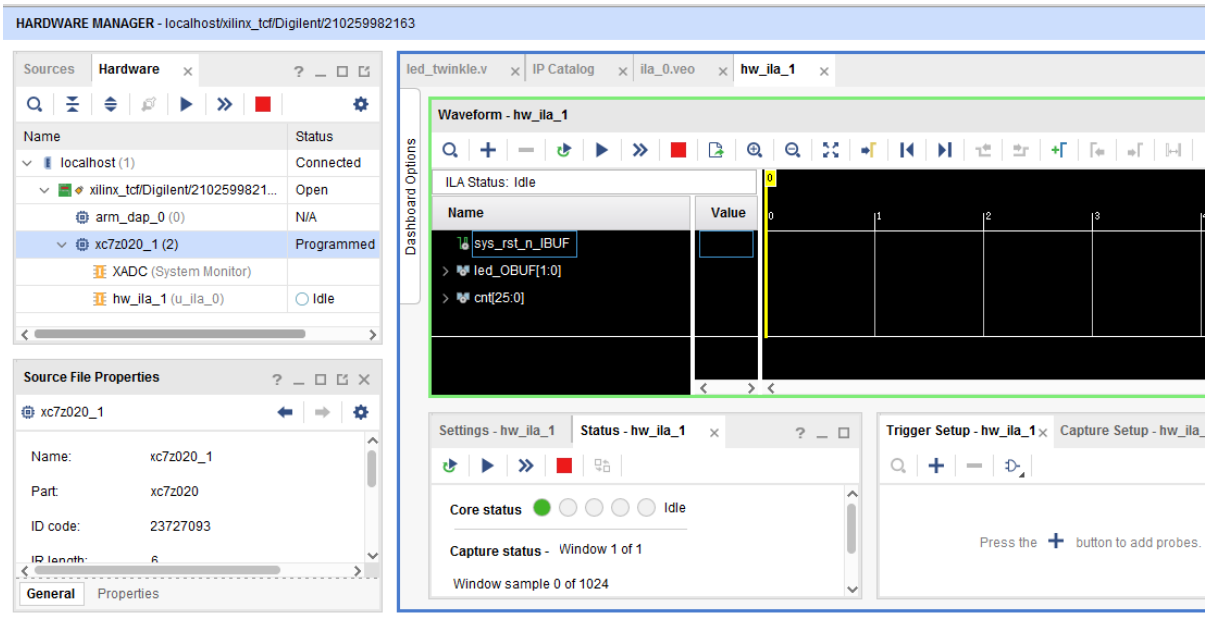


图 4.3.16 ILA 的调试窗口

如果 Waveform 窗口中显示的信号不全，可以点击“Waveform – hw_ila_1”窗口中的加号，来将所有的探针信号添加到波形窗口中（如果默认已经显示所有待观察的信号，不用重复添加），如下图所示：

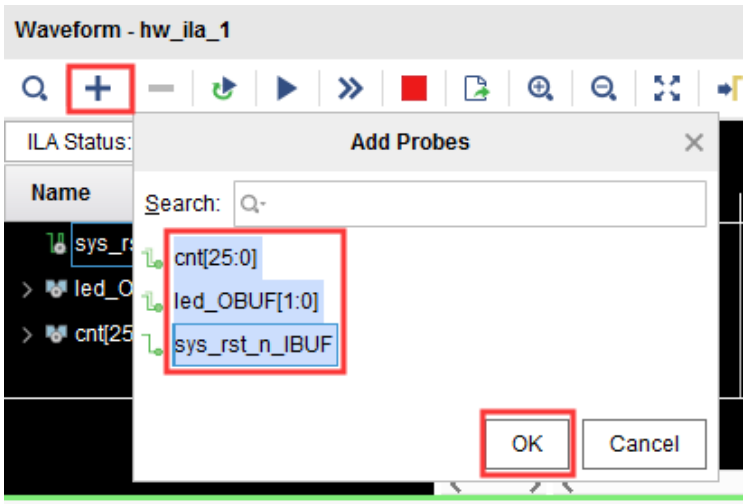


图 4.3.17 向波形窗口中添加探针信号

我们直接点击触发采集信号的按钮，可以观察到此时信号的波形，如图 4.3.18 和图 4.3.19 所示：

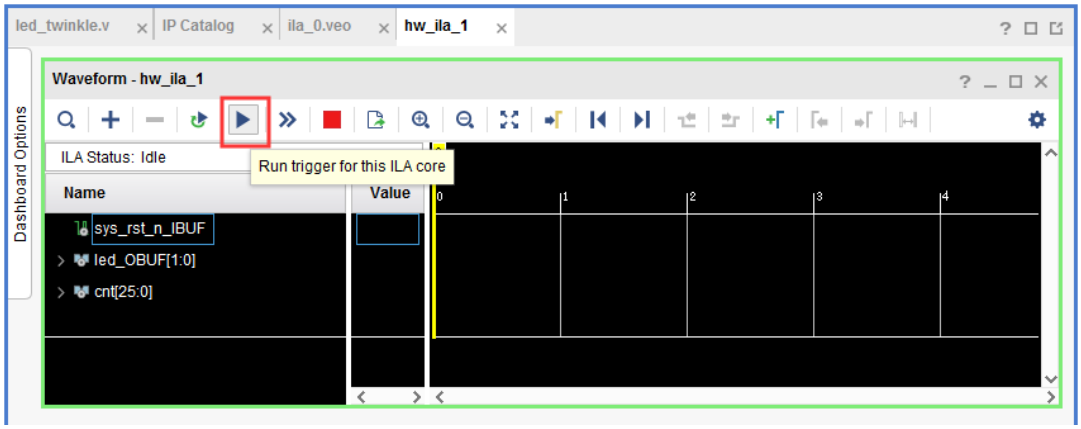


图 4.3.18 点击运行触发的图标

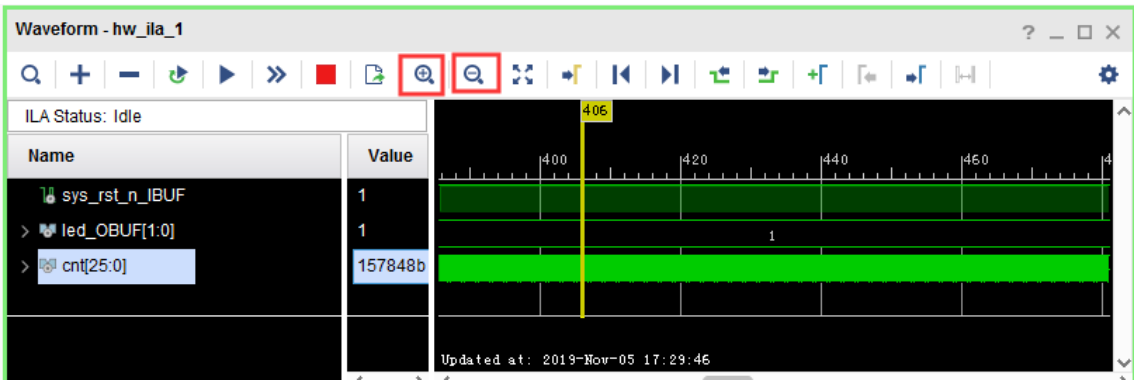


图 4.3.19 采集到的信号

采集到信号后，可以点击上图中的放大和缩小的图标，来对波形数据进行放大和缩小。

波形默认以十六进制显示的，可以右击 cnt 计数器，选择“Radix”→“Unsigned Decimal”，即可切换到无符号的十进制显示，如下图所示：

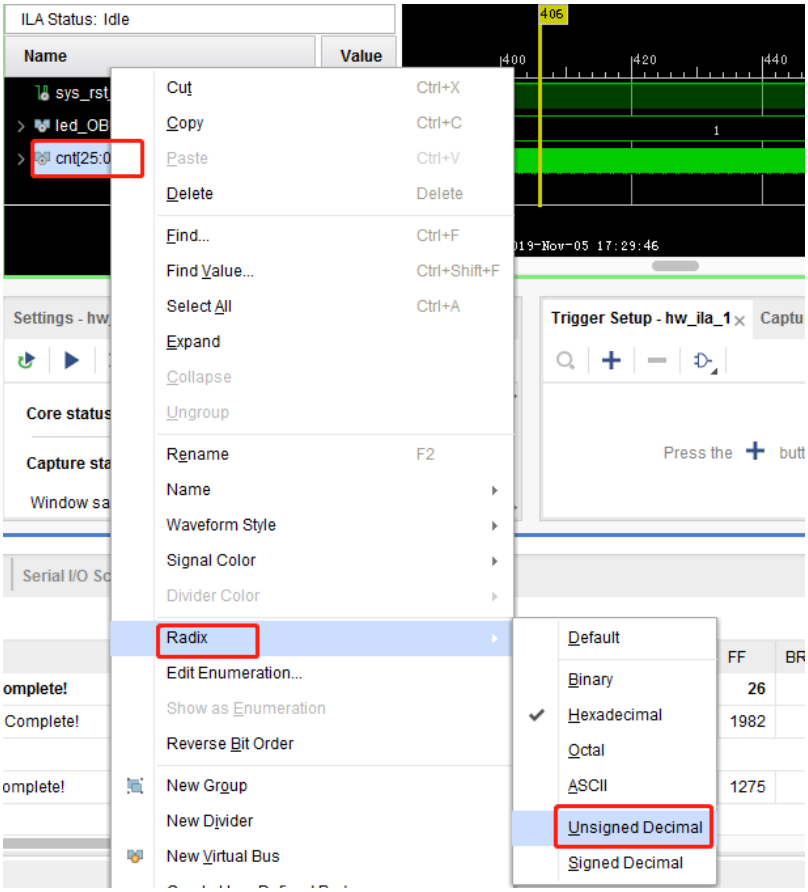


图 4.3.20 更改成无符号十进制显示

下面开始触发条件的设置，在这里简要介绍一个触发的概念。前面我们介绍过，ILA 会将所采集到的探针数据存放在 RAM 中，然后通过 JTAG 和下载器上传到 Vivado。那么触发就是决定 ILA 会在什么时候将 RAM 中的探针值数据上传到 Vivado，当 ILA 检测到触发条件得到满足时，就会把 RAM 中的探针值数据上载到 Vivado，然后 Vivado 将探针数据的波形显示出来。

我们在“Trigger Setup”窗口中添加触发条件，点击“+”号，将 cnt 信号添加进来，如下图所示：

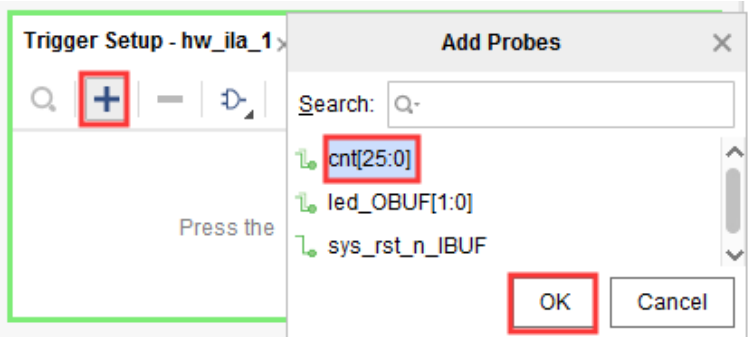


图 4.3.21 添加 counter 信号到“Trigger Setup 窗口中来

之前我们在 HDL 代码中设置为了 cnt 计数器每计数到 25'd25_000_000 和 25'd50_000_000 时就改变两个 led 的状态，为了方便 led 翻转状态的查看，我们可以将触发条件设置为“cnt 等于十进制数 25_000_000”，即，当 ILA 检测到 cnt 计数器的值等于 25'd25_000_000 时，就会将 RAM 中的数据上传到 Vivado，如下图所示：

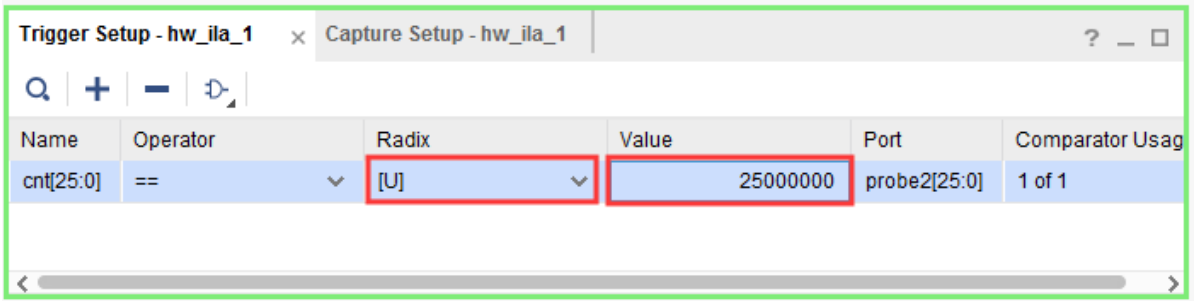


图 4.3.22 设置触发条件

然后我们就可以开始进行触发动作了。在波形窗口中有 4 个触发动作，如下图所示：

图 4.3.23 触发动作

从左至右依次是：（1）自动触发开关、（2）开始触发、（3）立即触发、（4）停止触发。下面对他们分别进行介绍：

（1）自动触发开关，它和“开始触发”按钮联合在一起使用。若打开了此选项，则在 ILA 开始运行触发（即点击了“开始触发”按钮）后，会不断地对触发条件进行检测，每次触发条件被满足时（即 cnt 计数到了 25_000_000），都会将 RAM 中存储的所有的探针值数据上传到 Vivado，Vivado 上显示的波形也会随之不断更新，直到用户点击了“停止触发”按钮。若没有点击此选项，则在 ILA 开始运行触发（即点击了“开始触发”按钮）后，在检测到触发条件得到满足并完成了上传数据之后，就会停止触发，等待用户下一步的指令。

（2）开始触发，它和“自动触发开关”按钮联合在一起使用。点击之后 ILA 就会开始进行触发操作。

（3）立即触发，立即将当前 ILA RAM 中的数据上传到 Vivado，而不管触发条件是否得到满足。

（4）停止触发，停止当前正在进行的触发活动。

我们这里直接点击“开始触发”按钮，而不打开“自动触发”开关。然后由于 cnt 计数器每隔 500ms 就会计数到 25_000_000，所以我们几乎马上就可以看到波形窗口中出现了波形，如下图所示：

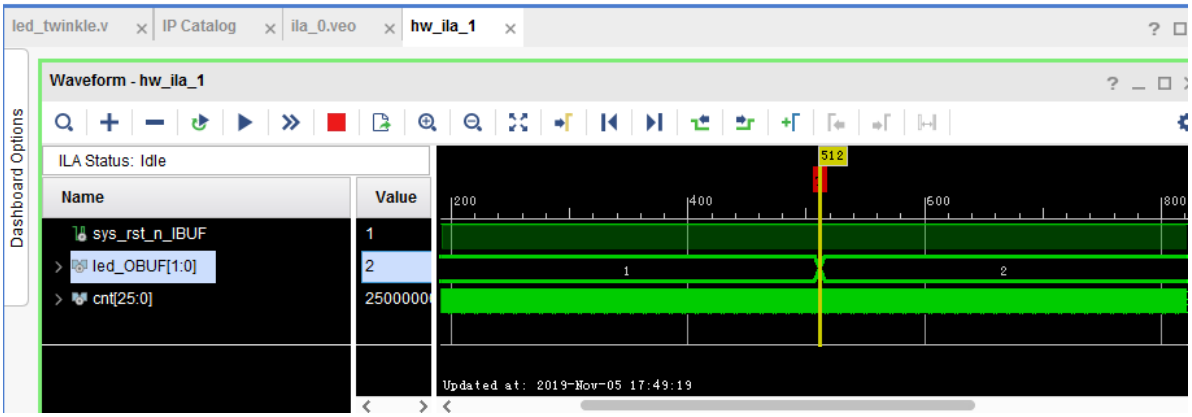


图 4.3.24 更新的波形

将波形放大之后就可以看到 counter 的具体值，如下图所示：

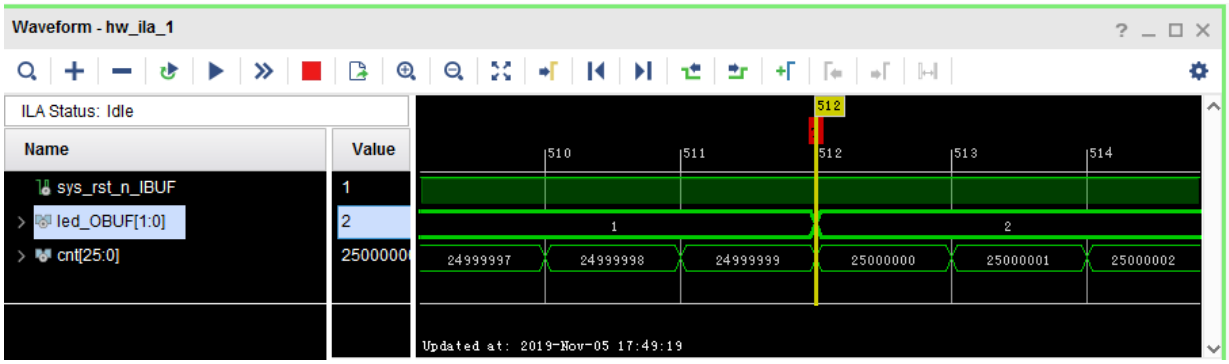


图 4.3.25 将波形放大

此时就可以看到，在当 cnt 计数到 25_000_000 时，led 的状态就会发生跳变。读者也可以尝试再次点击“开始触发”按钮或打开“自动触发”按钮，来观察波形。

如果调试工作完毕之后，可以在 led_twinkle.v 源代码中删除对 ILA IP 核的例化，或者通过添加“/*”和“*/”注释掉这段代码，如下图所示：

led_twinkle.v | IP Catalog | ila_0.veo | hw_ila_1

F:/ZYNQ/led_twinkle/led_twinkle.srcs/sources_1/new/led_twinkle.v

```
24      else
25          cnt <= 26'd0;
26  end
27  /*
28  ila_0 u_ila_0 (
29      .clk(sys_clk),          // input wire clk
30
31      .probe0(sys_rst_n), // input wire [0:0]  probe0
32      .probe1(led),       // input wire [1:0]  probe1
33      .probe2(cnt)        // input wire [25:0]  probe2
34  );
35  */
36  endmodule
```

图 4.3.26 注释掉对 ILA IP 核的例化

然后重新综合并实现，以生成最终的比特流。

4.3.3 网表插入调试探针流程

网表插入调试探针流程需要在综合后的网表中，将要进行调试观察的各个信号标记“mark_debug”属性，然后通过“Setup Debug”向导来设置 ILA IP 核的参数，最后工具会根据参数来自动创建 ILA IP 核。

可以在综合之后的网表中手动选择网络并点击“mark_debug”按钮；也可以在综合之前在 HDL 代码中为想要观察的 reg 或 wire 信号添加“Mark Debug”综合属性，例如：

(* mark_debug = “true” *)reg [25:0] cnt ;

其中“(* mark_debug = “true” *)”必须紧挨在变量声明的前面。这样，在综合完成之后并打开综合后的设计时，cnt 信号就自动被标记了“Mark Debug”属性。此外，被添加了(* mark_debug = “true” *)属性的 reg 或 wire 信号不会被工具优化掉。

这里我们选择第二个方法，即在 HDL 代码中添加(* mark_debug = “true” *)综合属性，如下图所示：

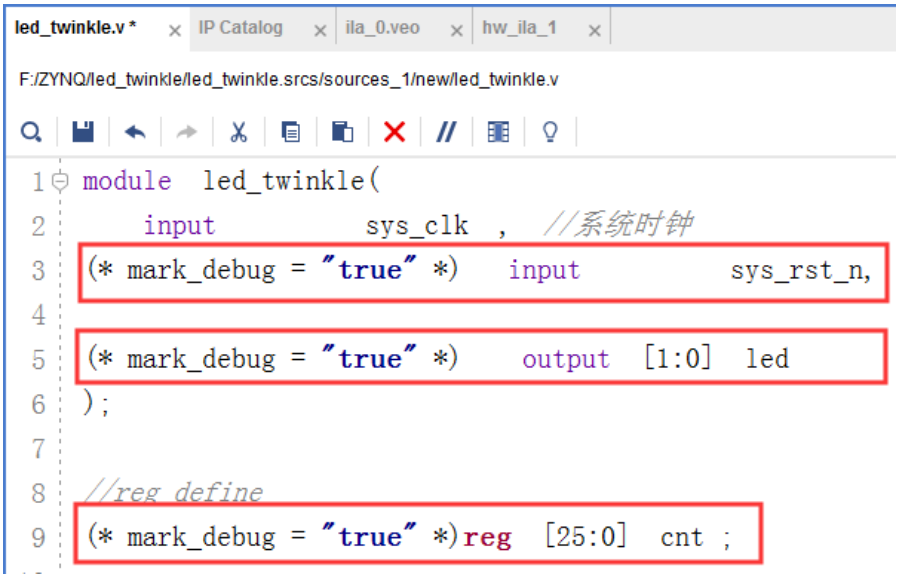


图 4.3.27 在 HDL 代码中添加“Mark Debug”综合属性

添加“Make Debug”属性之后，点击“Run Synthesis”进行综合，综合完之后我们点击“Flow Navigator”窗口中的“Open Synthesized Design”按钮，如下图所示：

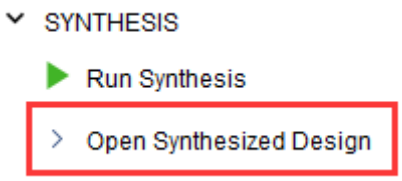


图 4.3.28 点击“Open Synthesized Design”按钮

在综合后设计的窗口布局选择器中，我们选择“Debug”窗口布局，如下图所示：

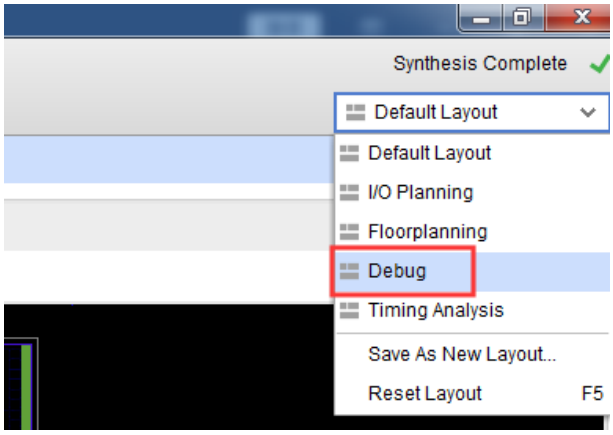


图 4.3.29 切换到“Debug”窗口布局

此时，Vivado 打开了“Netlist”子窗口、“Schematic”子窗口以及“Debug”子窗口。其中，“Netlist”子窗口和“Schematic”子窗口都可以用于标记要进行观察的信号，“Debug”子窗口用于显示并设置 ILA IP 核的各个参数。如下图所示：

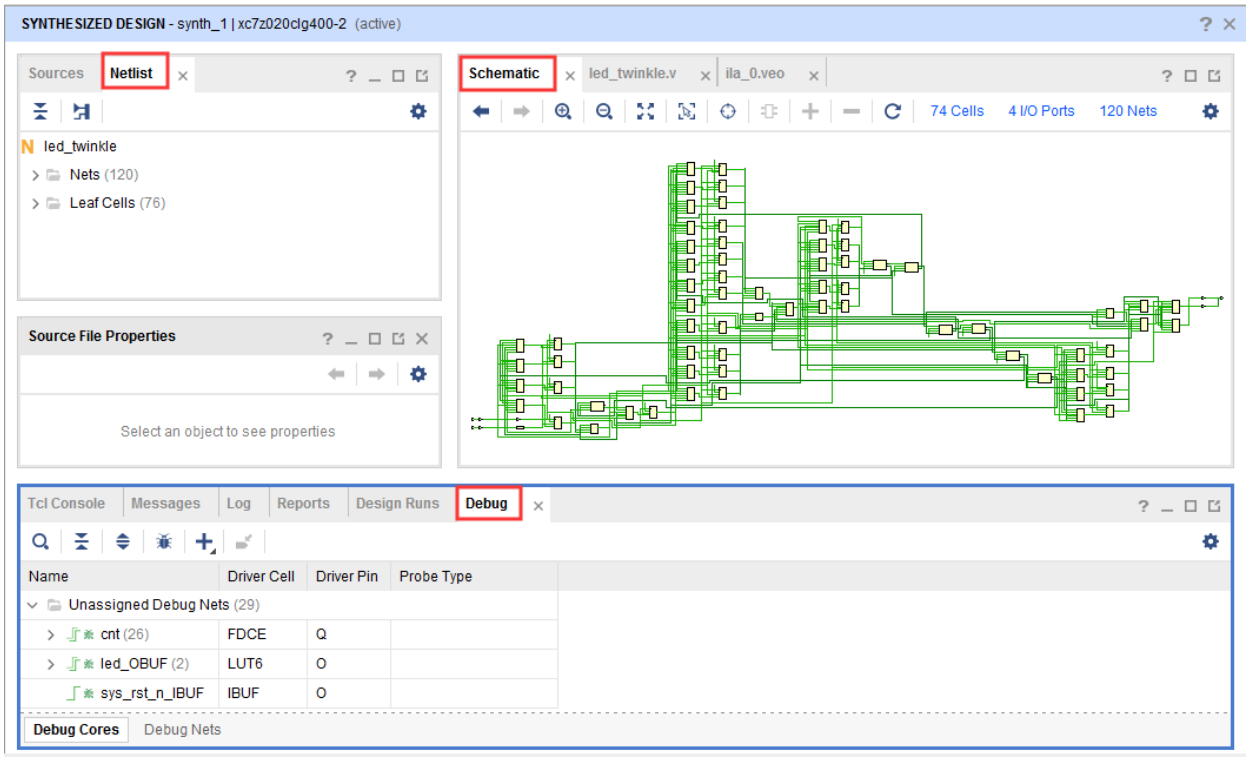


图 4.3.30 “Debug”窗口布局

在“Debug”子窗口中，又包含两个选项卡“Debug Cores”和“Debug Nets”。这两个选项卡都用于显示所有的已标记为“Mark_Debug”的信号。不同之处在于，“Debug Cores”选项卡是一个更加以 ILA IP 核为中心的视图，所有已标记为“Mark_Debug”的信号并且已经被分配到 ILA 探针的信号都会被显示在各个 ILA IP 核的视图树下，已标记为“Mark_Debug”的信号但是还没有被分配到 ILA 探针的信号被显示在“Unassigned Debug Nets”下，当然也可以在其中查看和设置 ILA IP 核的各种属性和参数。“Debug Nets”选项卡仅显示已标记为“Mark_Debug”的信号，但不显示 ILA IP 核，所有已标记为“Mark_Debug”的信号并且已经被分配到 ILA 探针的信号都会被显示在“Assigned Debug Nets”下，已标记为“Mark_Debug”的信号但是还没有被分配到 ILA 探针的信号被显示在“Unassigned Debug Nets”下。

（1）在综合后的网表中手动为信号添加 mark_debug 属性

如果未在 HDL 代码中书写(* mark_debug = “true” *)综合属性，则需要首先标记要进行观察的信号。在综合后的网表中，信号的名称可能会发生一定的变化，以 led 信号为例，在“Netlist”子窗口中的“Nets”目录下，找到“led_OBUF[0]”网络，右击该网络（此时右边的“Schematic”子窗口也会自动地高亮选择此网络，因为“Netlist”子窗口中的对象和“Schematic”子窗口中的对象，两者之间是交叉选择的），在弹出的菜单中心选择“Mark Debug”命令，如下图所示：

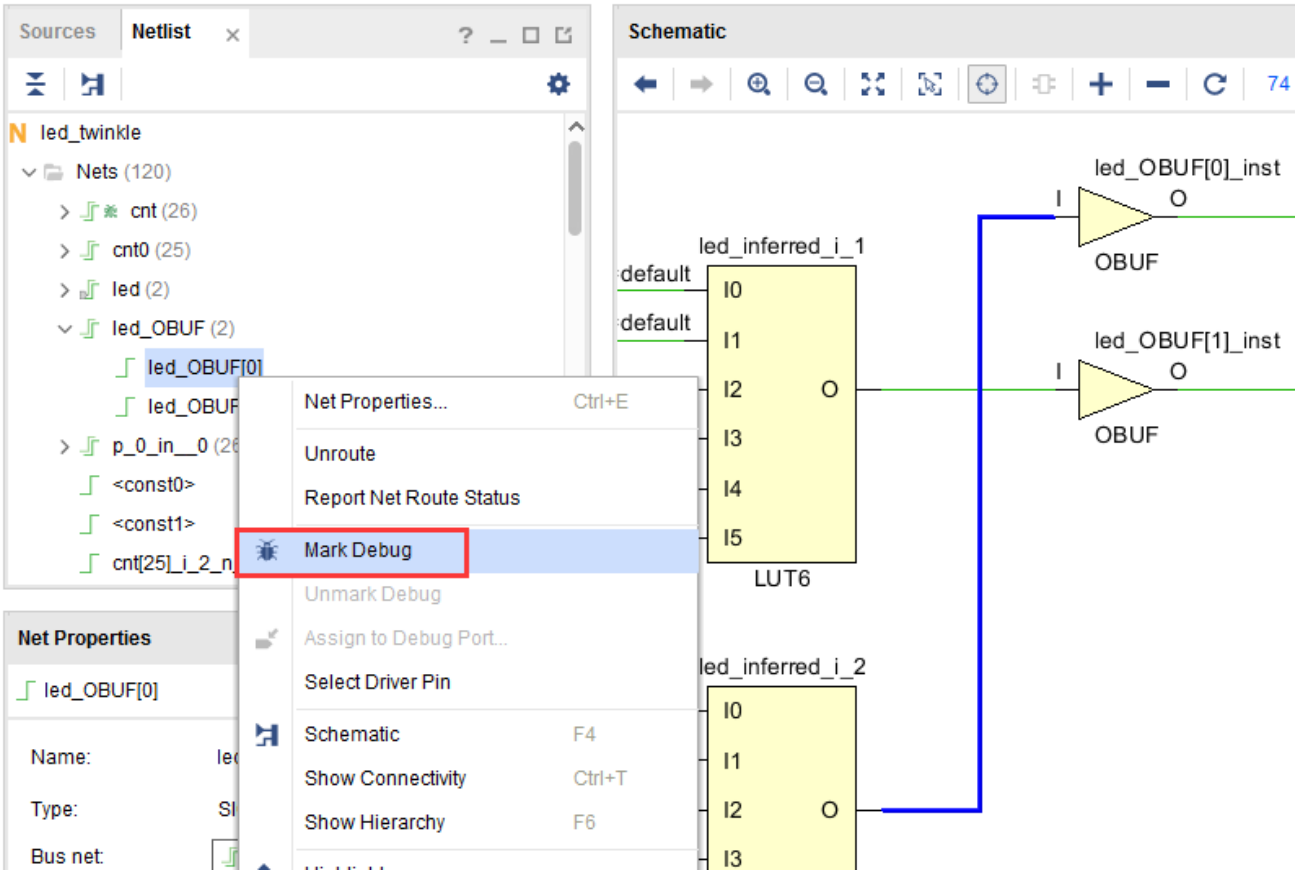


图 4.3.31 在“Netlist”子窗口中点击“Mark debug”命令

也可以在“Schematic”子窗口中选择网络，然后右键选择“Mark Debug”命令，如下图所示：

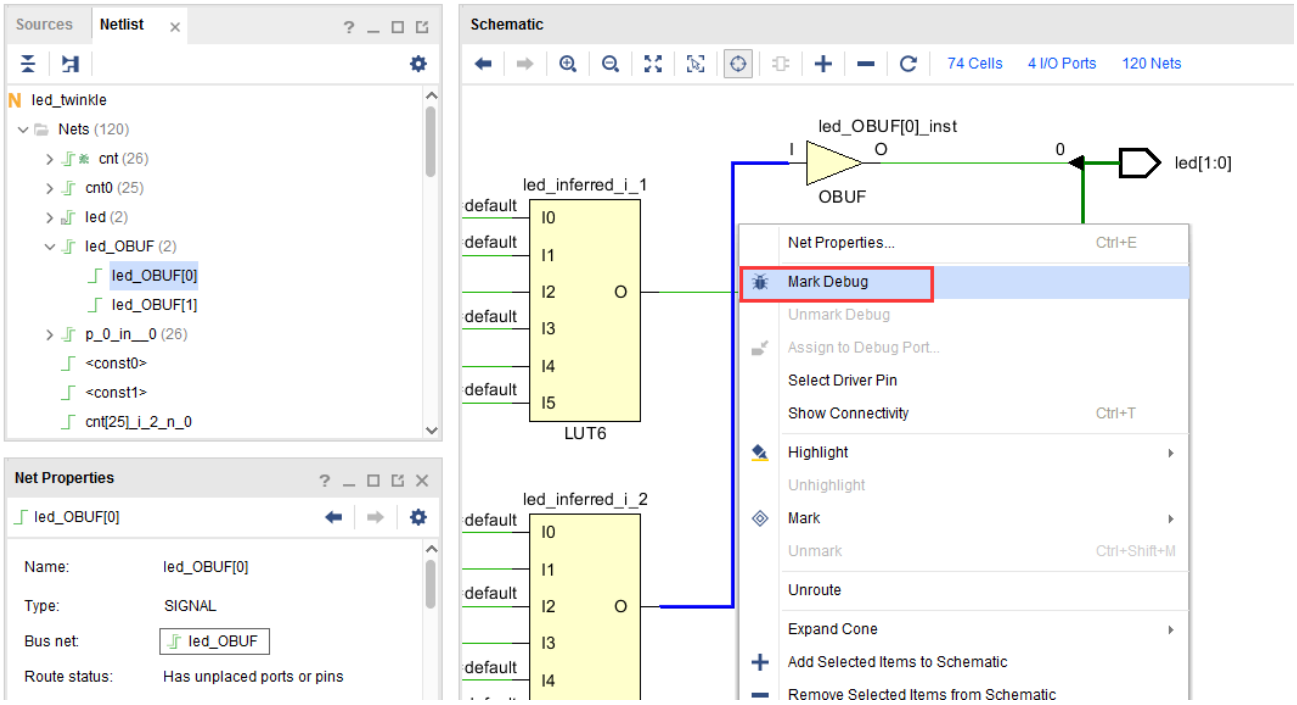


图 4.3.32 在“Schematic”子窗口中点击“Mark debug”命令

此时在“Debug”子窗口的“Debug Nets”选项卡的“Unassigned Debug Nets”目录下就会出现我们刚刚标记的“led_OBUF[0]”网络。，如下图所示：

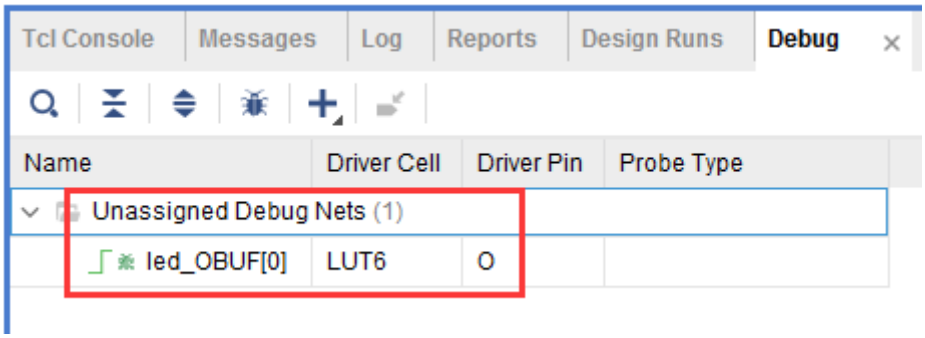


图 4.3.33 “Unassigned Debug Nets”目录下的“led_OBUF[0]”网络

（2）HDL 代码中已经具有 mark_debug 属性的信号

在 HDL 代码中已经添加了“Mark Debug”综合属性的信号会被自动出现在“Debug Nets”选项卡的“Unassigned Debug Nets”目录下，如下图所示：

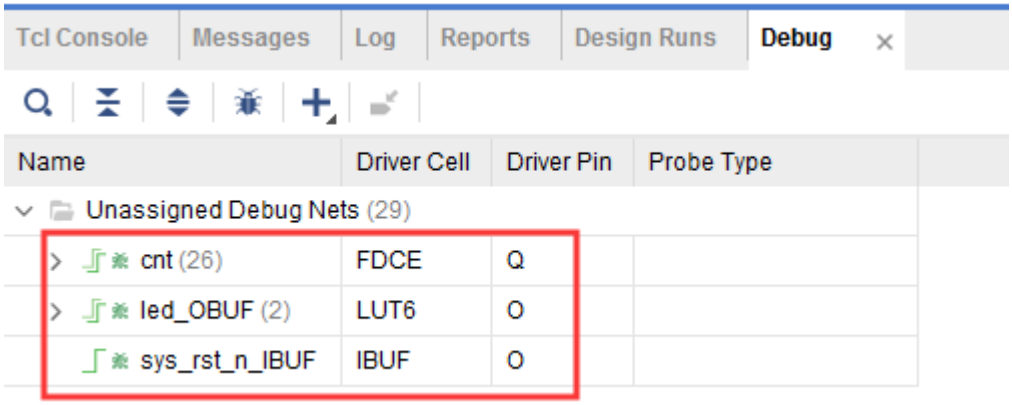


图 4.3.34 Unassigned Debug Nets 信号

为网络标记了“Mark Debug”之后，我们就可以进行 ILA IP 核的配置了。点击“Debug”子窗口中的“Setup Debug”按钮，如下图所示：

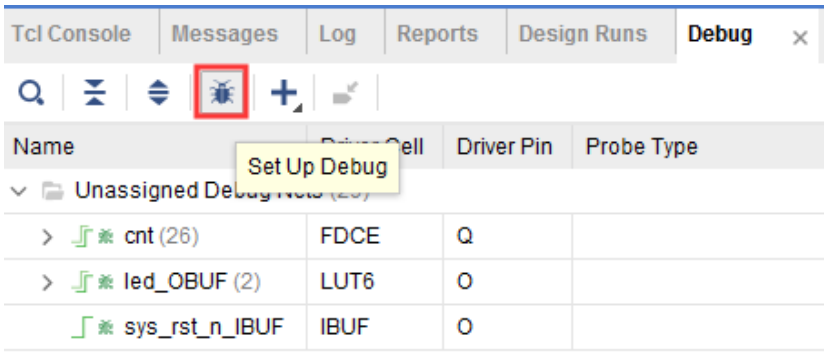


图 4.3.35 “Setup Debug”按钮

弹出“Setup Debug”向导，我们直接点击“Next”按钮，如下图所示：

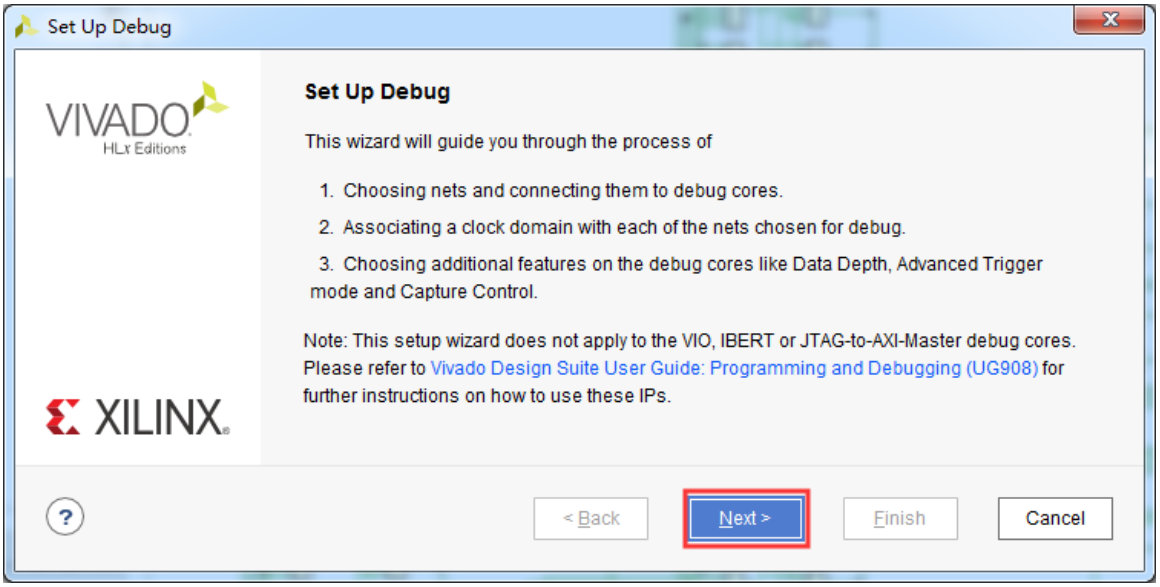


图 4.3.36 “Setup Debug”向导

接下来的页面是选择用于采样待测信号的时钟域，Vivado 会自动识别出各个待测信号所属的时钟域并将其自动设定为其采样时钟，本次添加的三个信号属于系统时钟（sys_clk）的时钟域，Vivado 也已经自动将“sys_clk_IBUF”时钟设置为了这两个信号的采样时钟，如下图所示：

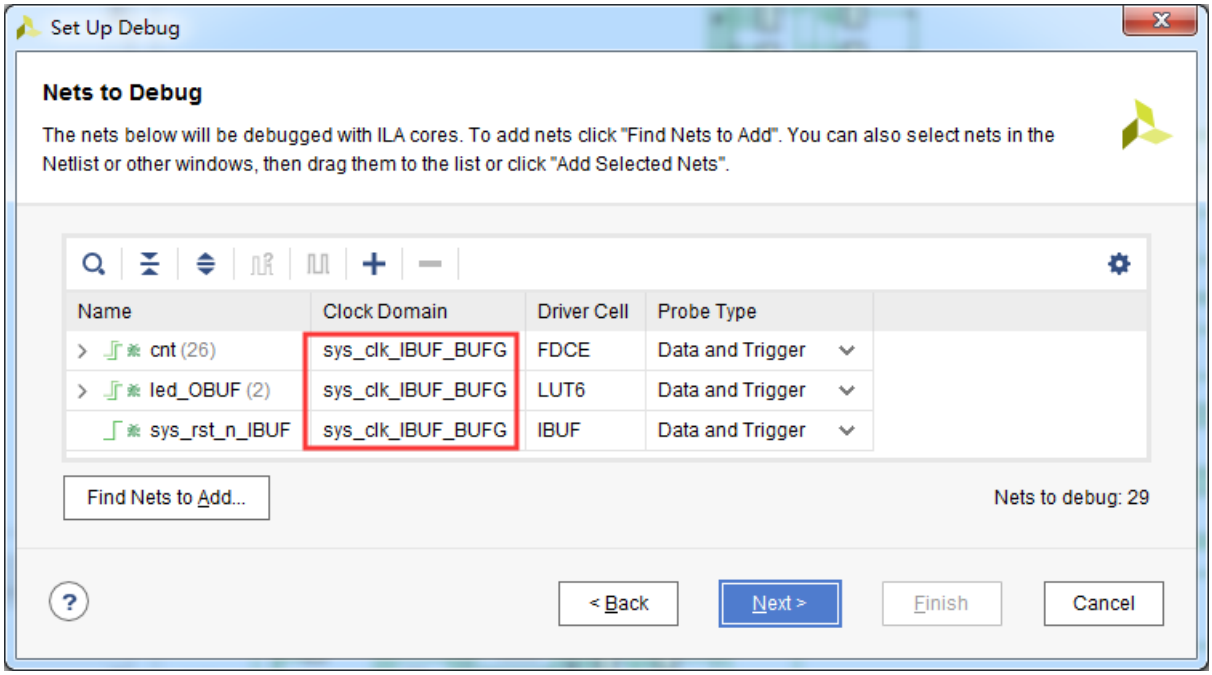


图 4.3.37 选择待测信号的时钟域

当然，用户也可以手动指定各个用于采样待测信号的时钟域，右击待测信号，选择“Select Clock Domain”，弹出“Select Clock Domain”窗口，如图 4.3.38 和图 4.3.39 所示：

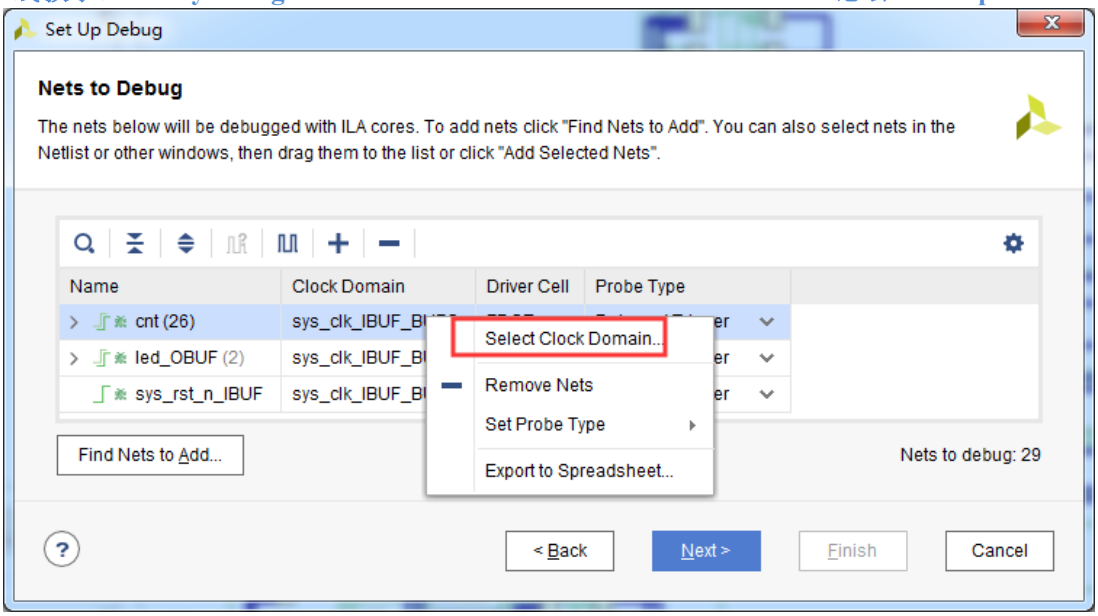


图 4.3.38 手动指定待测信号的时钟域 1

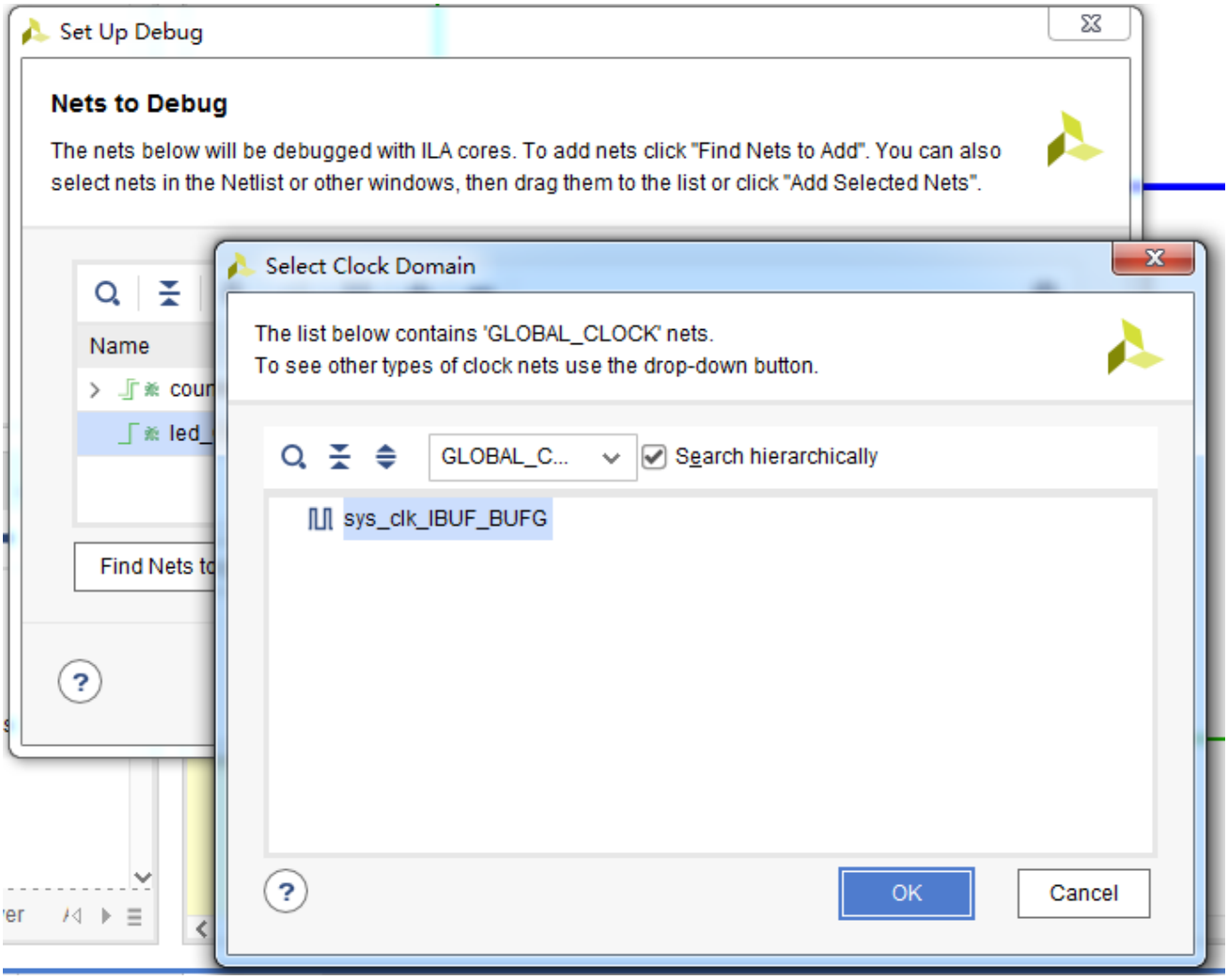


图 4.3.39 手动指定待测信号的时钟域 2

在“Select Clock Domain”窗口中就可以选择用于采样待测信号的时钟了。“Setup Debug”向导会为每

个采样时钟生成一个单独的 ILA IP 核，由于本例程中只有一个时钟，所以这里最后只会生成一个 ILA IP 核。

设置完采样时钟后，我们点击 next，接下来的页面用于设置 ILA IP 核的全局设置，如下图所示：

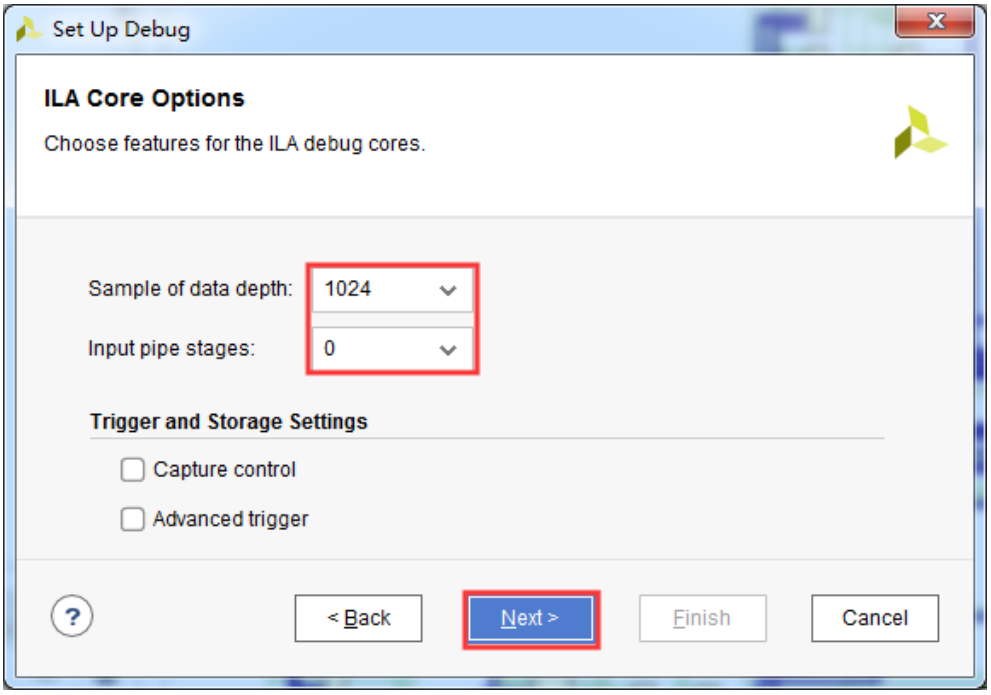


图 4.3.40 ILA IP 核的全局设置

其中“Sample of data depth”用于设置采样深度，“Input pipe stages”用于设置待测信号和其采样时钟之间的同步级数。如果在上一个设置时钟域页面中，存在与其采样时钟之间是异步的待测信号，则为了避免亚稳态，此数值最好不要低于 2。由于本例中的两个待测信号的其采样时钟是同步的，所以可以设置为 0。

我们点击 next，就进入了最后的概览页面，确认无误后直接点击“Finish”按钮即可，如下图所示：

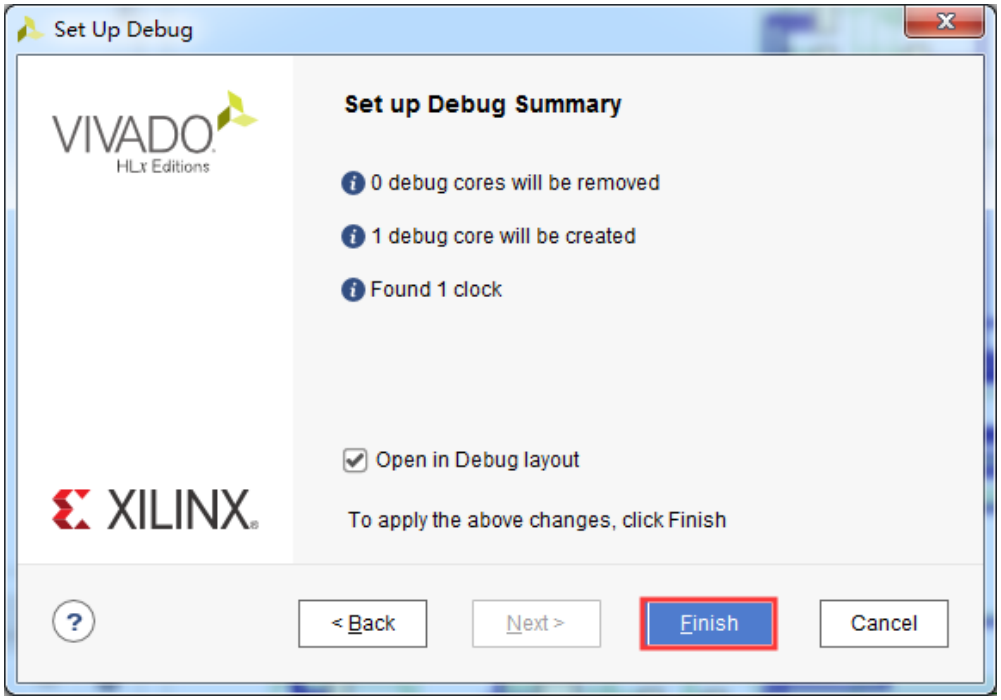


图 4.3.41 完成“Setup Debug”向导

在“Debug”子窗口中的“Debug Cores”选项卡中，可以看到 Vivado 已经添加了 ILA IP 核，并且“Unassigned

Debug Nets”目录下已经没有未被分配的信号了，如下图所示：

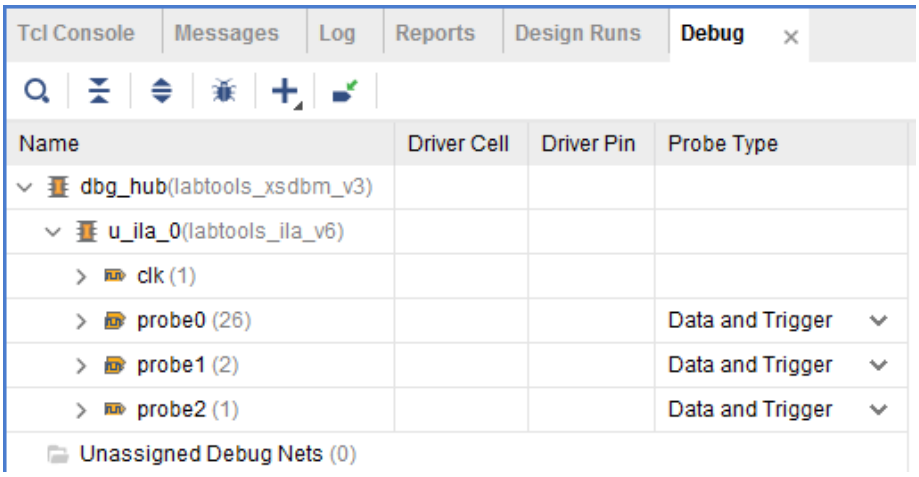


图 4.3.42 添加了 ILA IP 核的“Debug”子窗口

网表中被标记为 Mark Debug 的信号也变为了虚线，以表示其完成了 ILA IP 核的分配，如下图所示：

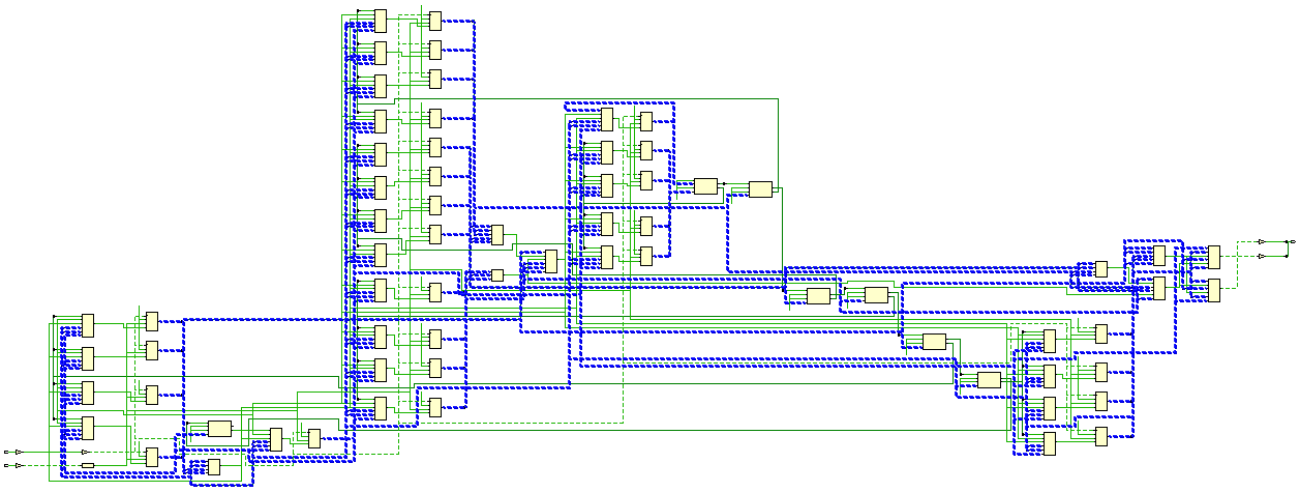


图 4.3.43 网表中完成 Setup Debug 的信号

前面我们提到过，在“网表插入调试探针流程”中，用户设置的调试信息最终会以 Tcl XDC 调试命令的形式保存到 XDC 约束文件中。在实现阶段，Vivado 会从 XDC 约束文件中读取这些 XDC 调试命令，并在布局布线时加入这些 ILA IP 核。此时，我们所做出的所有的更改和设置，都还只是停留在电脑内存中，我们需要将其保存在硬盘的 XDC 约束文件中，以供 Vivado 在实现阶段读取。点击工具栏中的保存按钮，如下图所示：

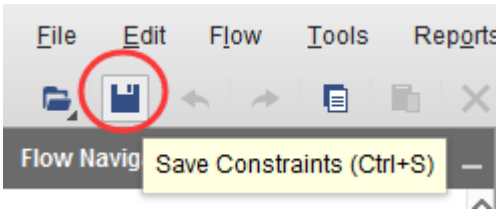


图 4.3.44 保存按钮

在出现的消息框中直接点击 OK，如下图所示：

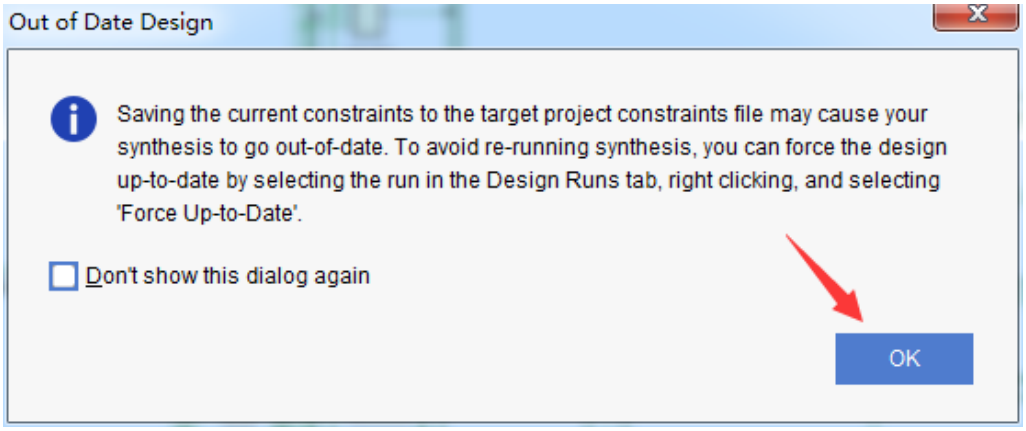


图 4.3.45 点击 OK

如果弹出“Save Constraints”对话框，询问用户将约束保存在哪个 XDC 约束文件中，本例的工程中只有一个 XDC 约束文件，选择“Select an existing file”，位于 led_twinkle.xdc 文件下，然后点击“OK”按钮即可（有时候也有可能没有弹出这个对话框）。

此时，我们打开 led_twinkle.xdc 文件，就会看到在用户约束的下面，Vivado 自动写入了用于 debug 的约束命令，如下图所示：

```
7  create_debug_core u_ila_0 ila
8  set_property ALL_PROBE_SAME_MU true [get_debug_cores u_ila_0]
9  set_property ALL_PROBE_SAME_MU_CNT 1 [get_debug_cores u_ila_0]
10 set_property C_ADV_TRIGGER false [get_debug_cores u_ila_0]
11 set_property C_DATA_DEPTH 1024 [get_debug_cores u_ila_0]
12 set_property C_EN_STRG_QUAL false [get_debug_cores u_ila_0]
13 set_property C_INPUT_PIPE_STAGES 0 [get_debug_cores u_ila_0]
14 set_property C_TRIGIN_EN false [get_debug_cores u_ila_0]
15 set_property C_TRIGOUT_EN false [get_debug_cores u_ila_0]
16 set_property port_width 1 [get_debug_ports u_ila_0/clk]
17 connect_debug_port u_ila_0/clk [get_nets [list sys_clk_IBUF_BUFG]]
18 set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports u_ila_0/probe0]
19 set_property port_width 26 [get_debug_ports u_ila_0/probe0]
```

图 4.3.46 Vivado 自动写入的用于 debug 的约束

在实现阶段，Vivado 会读取这些约束，并按照这些命令的参数来自动地加入 ILA IP 核。

至此，我们就成功地使用“网表插入调试探针流程”将 ILA IP 核添加到了设计中。接下来就可以实现设计并生成比特流，最后将比特流下载到 FPGA 中，以对信号进行在线观察。

对信号在线观察的方法请参考“4.3.2 Hardware Manager 中观察调试信号”一节，操作方法和“HDL 实例化调试探针”的观察波形信号一样，没有任何区别。

如果调试工作完毕之后，可以删除 led_twinkle.v 源代码信号中的“Make Debug”属性和删除 XDC 文件中的包含调试信息的 Tcl 文件，然后重新综合并实现，以生成最终的比特流。至此，在线逻辑分析仪的使用介绍完毕。

4.4 在 Vivado 中进行功能仿真

在进行功能仿真之前，我们先看一下典型的 FPGA 设计流程，流程图如下：

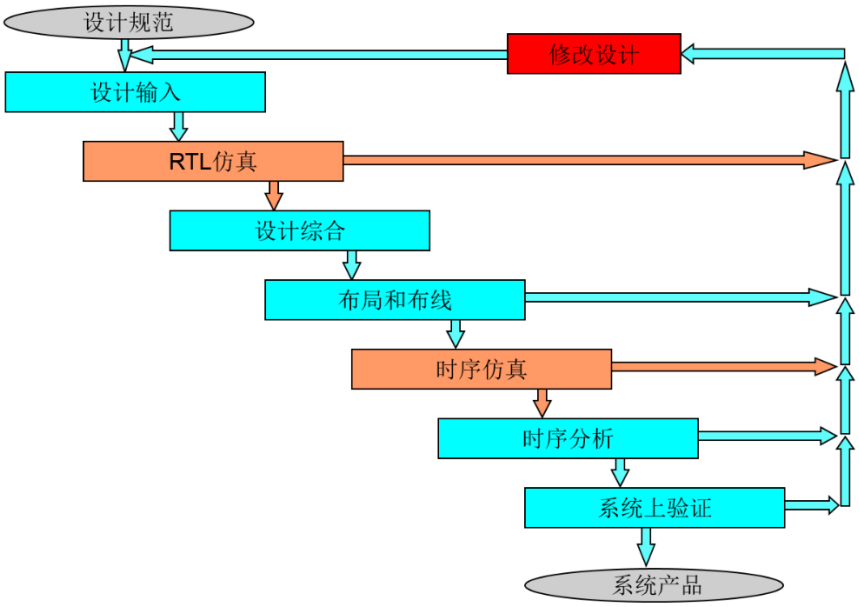


图 4.4.1 FPGA 的设计流程

从上图可以看到，在设计输入之后、设计综合之前进行 RTL 级仿真，称为综合前仿真，也称为前仿真或功能仿真。前仿真也就是纯粹的功能仿真，主旨在于验证电路的功能是否符合设计要求，其特点是不考虑电路门延迟与线延迟。在完成一个设计的代码编写工作之后，可以直接对代码进行仿真，检测源代码是否符合功能要求。这时，仿真的对象为 HDL 代码，可以比较直观的观察波形的变化，在设计的最初阶段发现问题，节省大量的精力。

在布局布线后进行的仿真称为布局布线后仿真，也称为后仿真或时序仿真。时序仿真真实地反映了逻辑的时延与功能，综合考虑电路的路径延迟与门延迟的影响，验证电路能否在一定时序条件下满足设计构想的过程，是否存在时序违规。

Vivado 设计套件内部集成了仿真器 Vivado Simulator，能够在设计流程的不同阶段运行设计的功能仿真和时序仿真，结果可以在 Vivado IDE 集成的波形查看器中显示。Vivado 还支持与诸如 ModelSim、Verilog Compiler Simulator (VCS)、Questa Advanced Simulator 等第三方仿真器的联合仿真。

功能仿真需要的文件：

1．设计 HDL 源代码，也被称为 UUT（Unit Under Test）：可以是 VHDL 语言或 Verilog 语言，既可以是设计的顶层模块，也可以是设计的下层子模块。

2．测试激励代码，也被称为 TestBench：根据 UUT 顶层输入/输出接口的设计要求，来产生顶层输入接口的测试激励并监视顶层输出接口。由于不需要进行综合，书写具有很大的灵活性。

TestBench 和 UUT 之间的关系如下图所示：

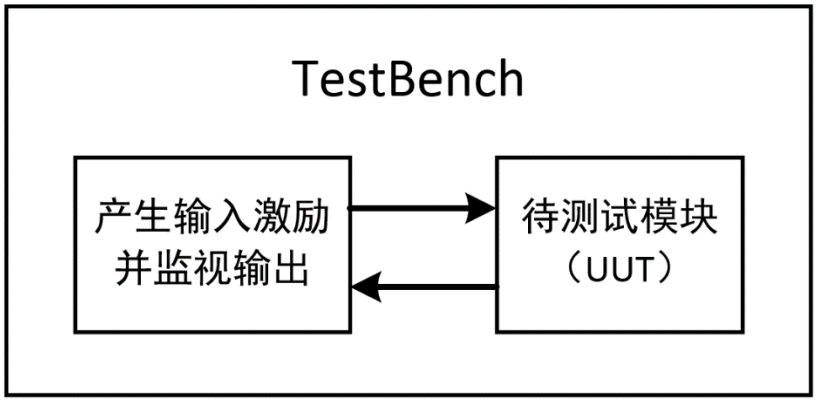


图 4.4.2 TestBench 和 UUT 之间的关系

3．仿真模型/库：根据设计内调用的器件供应商提供的模块而定，如：FIFO、ADD_SUB 等。在使用 Vivado Simulator 时，仿真器所需的仿真模型/库是预编译好并集成在 Vivado 中的，因此不需要进行额外的预编译操作，直接加载 HDL 设计和 TestBench 即可执行仿真。

接下来我们开始在 Vivado IDE 中进行仿真，首先需要创建一个 TestBench。我们点击“Sources”窗口中的“+”号（Add Sources 命令），在弹出的窗口中选择“Add or Create Simulation Sources”，如图 4.4.3 和图 4.4.4 所示：

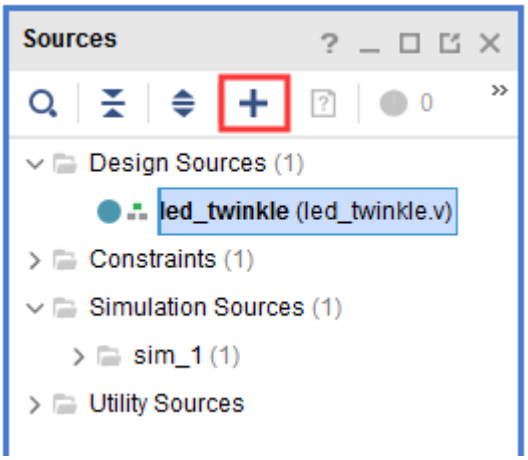


图 4.4.3 点击“Add Sources”

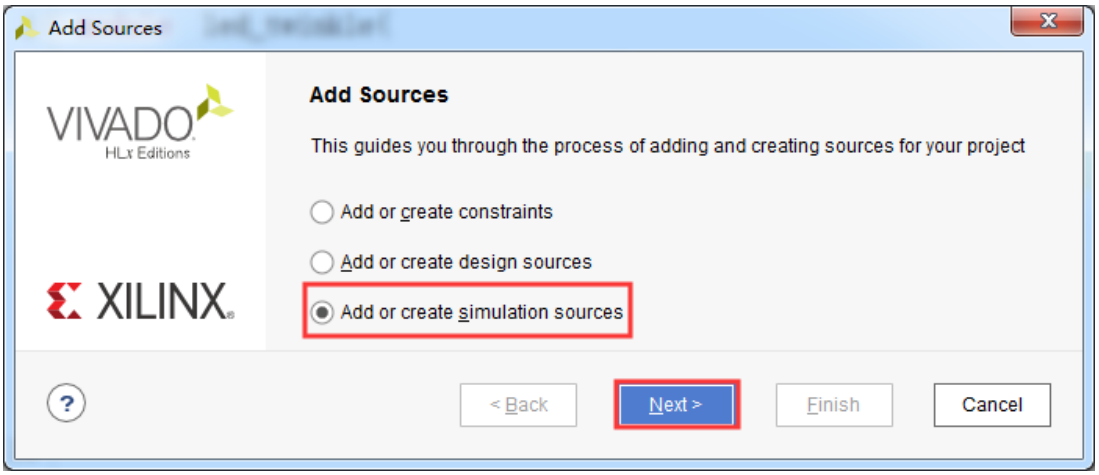


图 4.4.4 选择添加仿真源文件

点击“NEXT”按钮，在接下来的页面中点击“Create File”，如下图所示:

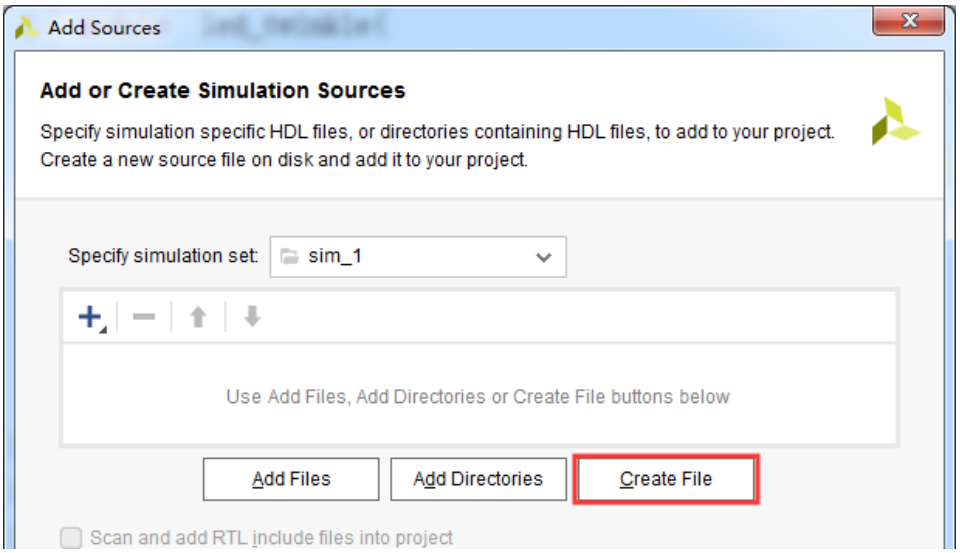


图 4.4.5 点击“Create File”

在弹出的对话框中输入 TestBench 的文件名“tb_led_twinkle”，并点击“OK”按钮，如下图所示：

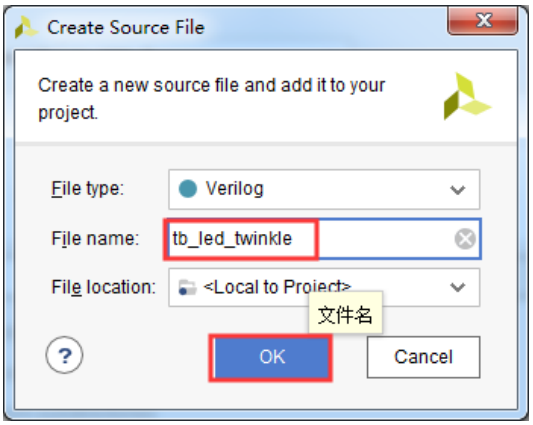


图 4.4.6 输入 TestBench 的文件名

TestBench 源文件名称的前缀“tb_”可以用来向用户示意：该源文件是一个 TestBench 源文件，仅用于仿真，并不能用于设计的综合和实现。建议大家按照这种规范来创建 TestBench，以免设计源文件和仿真源文件相混淆。

接下来直接点击“Finish”按钮，如下图所示：

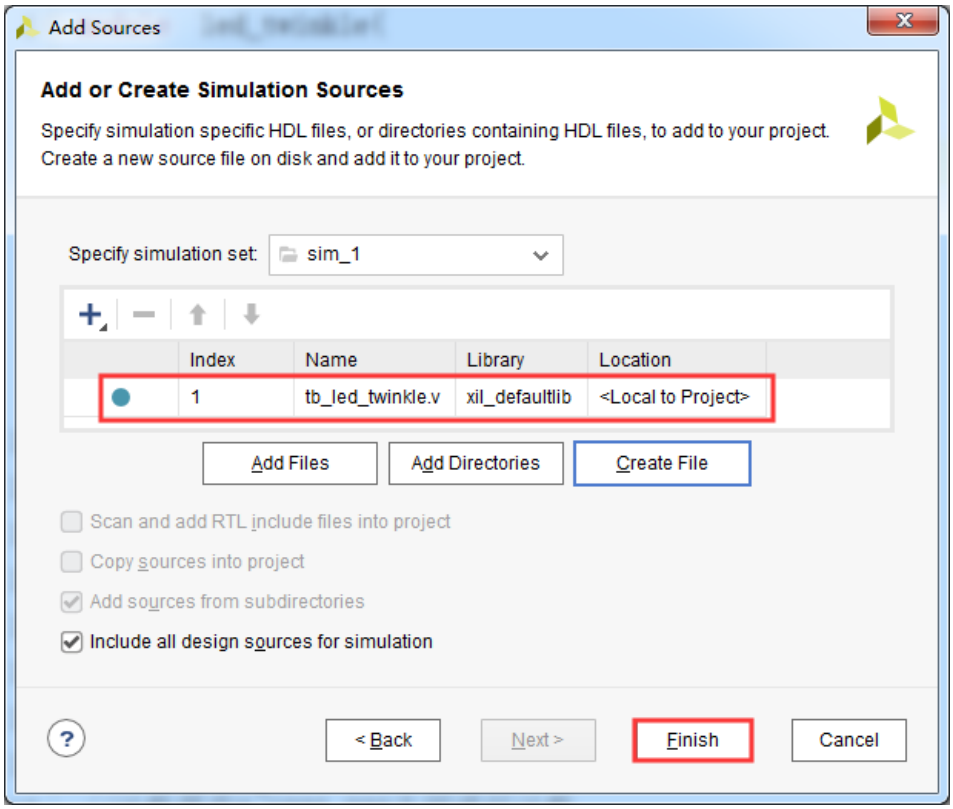


图 4.4.7 点击“Finish”按钮

在弹出的自动定义模块窗口中我们直接点击“OK”按钮即可，结束 TestBench 源文件端口的定义，如下图所示：

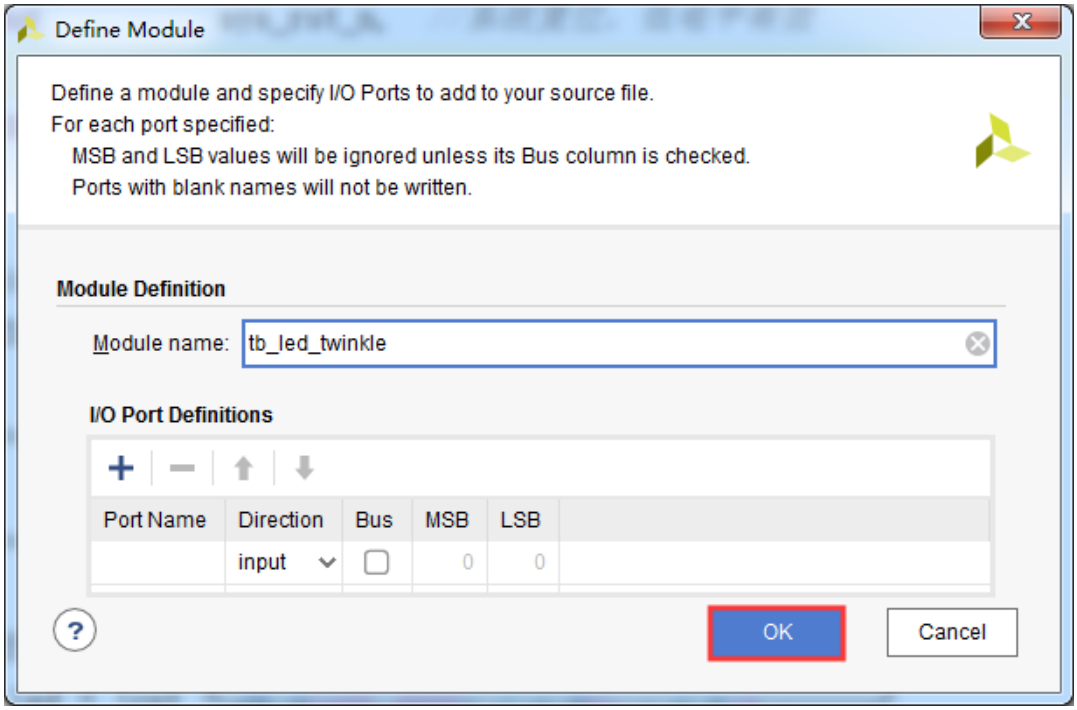


图 4.4.8 结束 TestBench 的创建

紧接着会弹出一个模块定义的确认按钮，点击“YES”按钮即可，如下图所示：

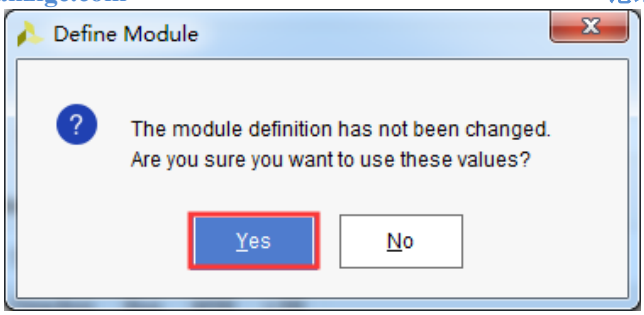


图 4.4.9 模块定义确认按钮

此时我们就可以在 Source 窗口→Simulation Sources→sim_1 下找到 tb_led_twinkle.v 文件，这个文件就是刚刚创建的 TestBench 文件，双击打开后发现其模块内部只定义了模块名，如下图所示：

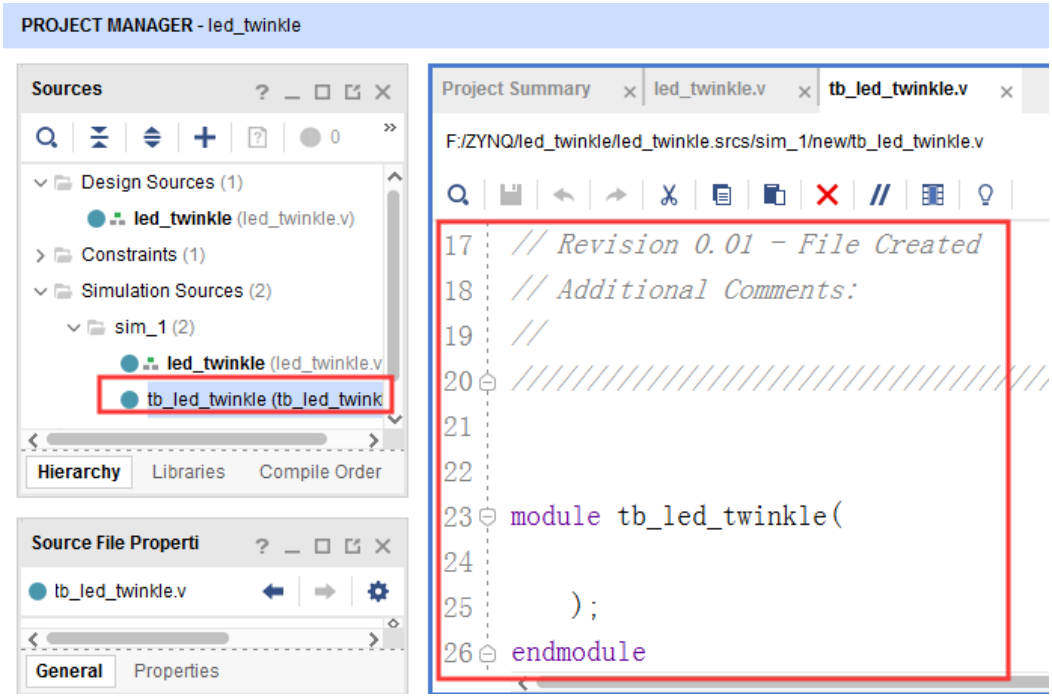


图 4.4.10 打开 TestBench 文件

我们删除 tb_led_twinkle.v 文件中默认的代码，然后替换成 LED 灯闪烁的 TestBench（激励）代码，代码如下：

```
1  `timescale 1ns / 1ps
2
3  module tb_led_twinkle();
4
5  //输入
6  reg          sys_clk;
7  reg          sys_rst_n;
8
9  //输出
10 wire  [1:0]   led;
11
12 //信号初始化
```

```
13 initial begin
14     sys_clk = 1'b0;
15     sys_rst_n = 1'b0;
16     #200
17     sys_rst_n = 1'b1;
18 end
19
20 //生成时钟
21 always #10 sys_clk = ~sys_clk;
22
23 //例化待测设计
24 led_twinkle  u_led_twinkle(
25     .sys_clk         (sys_clk),
26     .sys_rst_n       (sys_rst_n),
27     .led             (led)
28     );
29
30 endmodule
```

编写完成后，单击保存按钮来保存 TestBench。

为了让读者能够更好的理解，这里我们就简单介绍一下 TestBench 源代码。仿真代码首先要规定时间单位和精度，我们建议大家最好在 Testbench 里面统一规定时间单位，不要在工程代码里定义，因为不同的模块如果时间单位不同可能会为仿真带来一些问题。代码的第 1 行 timescale 是 Verilog 语法中的不可综合的语法，用于定义仿真文件中的单位，表示仿真的时间单位为 1ns，精度为 1ps，这是赛灵思官方推荐的仿真时间单位和精度。代码的第 3 行就是 TestBench 的模块名定义，其中第 5 行至第 10 行是我们的数据类型定义，第 12~18 行是信号的初始化，第 20-21 行用于生成时钟信号，到了第 23-28 行是对被测模块（led_twinkle）的例化。

在开始仿真之前，有一点需要注意，我们在 Vivado 软件中实现的功能是 LED 闪烁效果，它的间隔时间是 500ms，如果我们想要仿真这个功能，那么我们仿真软件运行时间最低就是 500ms。这 500ms 在我们看来是很短的，而对仿真软件来说是很漫长的，毕竟我们的仿真时间单位可是 1ns。为了便于我们仿真，这里我们需要稍微改动一下“led_twinkle.v”文件的代码，将计时器 cnt 的最大计时值设为 10，如下图所示：

```
15  //对计数器的值进行判断，以输出LED的状态
16  //assign led = (cnt < 26'd2500_0000) ? 2'b01 : 2'b10 ;
17    assign led = (cnt < 26'd5)          ? 2'b01 : 2'b10 ;  //仅用于仿真
18
19  //计数器在0~5000_000之间进行计数
20  always @ (posedge sys_clk or negedge sys_rst_n) begin
21      if(!sys_rst_n)
22          cnt <= 26'd0;
23  //    else if(cnt < 26'd5000_0000)
24        else if(cnt < 26'd10)  //仅用于仿真
25          cnt <= cnt + 1'b1;
26      else
27          cnt <= 26'd0;
28  end
```

图 4.4.11 修改源文件

这里，我们通过双斜杠“//”将源代码注释掉，并将源代码复制一份放到源代码的后面，并修改计数器的值。待仿真结束后进行分析和综合时，再将标记为“仅用于仿真的”的语句注释掉或者直接删除。

在“Flow Navigator”窗口中点击“Run Simulation”并选择“Run Behavioral Simulation”，如下图所示：

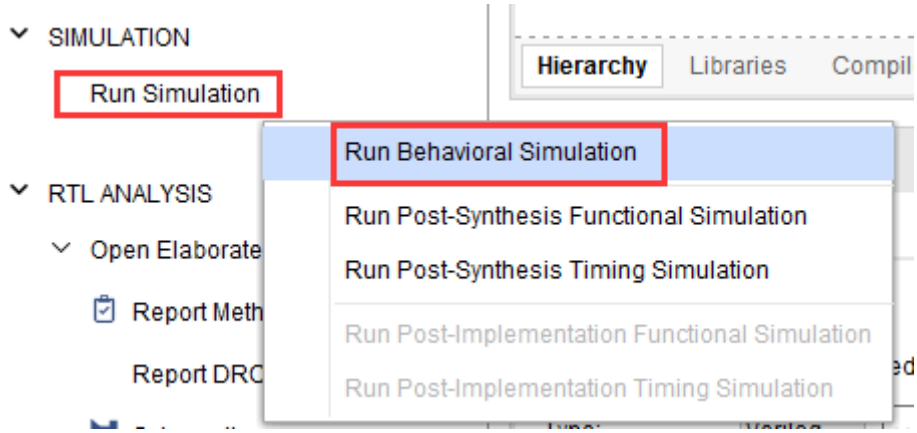


图 4.4.12 Run Simulation 命令

之后我们就进入了仿真界面，如下图所示：

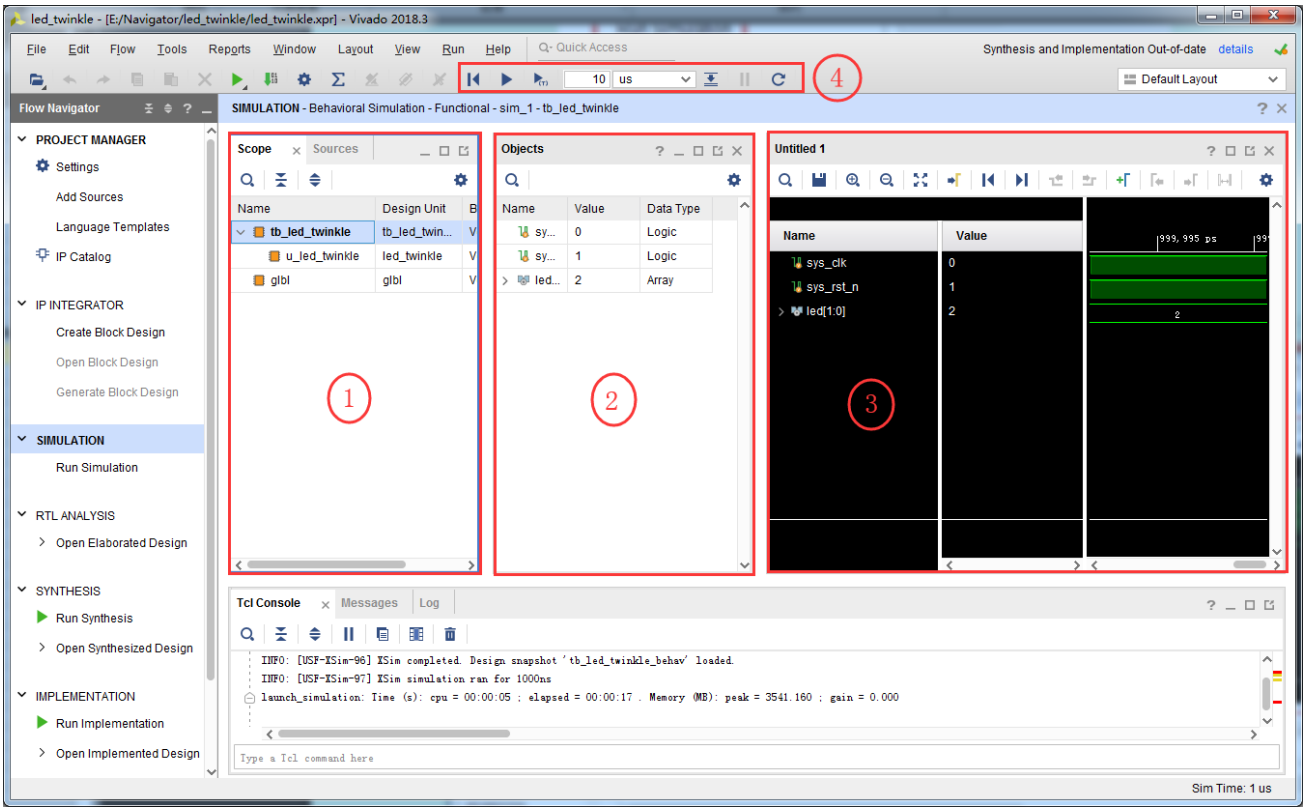


图 4.4.13 仿真界面

下面分别介绍仿真界面中的各个子窗口：

（1）Scope 窗口。Scope（范围）是 HDL 设计的层次划分。在 Scope 窗口中，您可以看到设计层次结构。当您选择了一个 Scope 层次结构中的作用域时，该作用域内的所有 HDL 对象，包括 reg、wire 等都会出现在“Objects”窗口中。您可以在“Objects”窗口中选择 HDL 对象，并将它们添加到波形窗口中。

（2）Object 窗口。“Objects”窗口会显示在“Scopes”窗口中选择的范围内的所有 HDL 仿真对象。例如，我们在 Scope 窗口中选择“u_led_twinkle”，在“Objects”窗口中就会自动显示出 led_twinkle 模块中所有的对象。如下图所示：

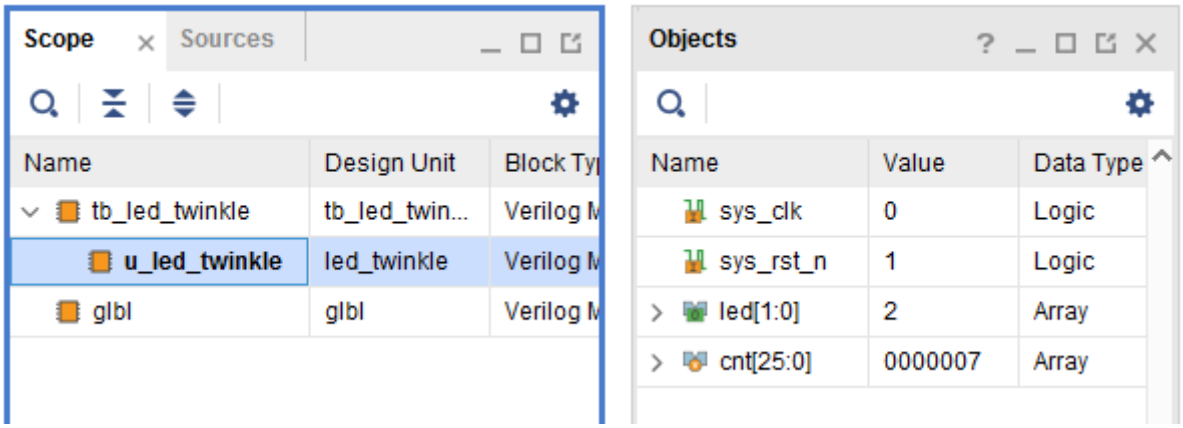


图 4.4.14 “Objects”窗口中的对象

可以看出，在 led_twinkle 顶层模块中除了顶层端口 sys_clk、sys_rst_n、led 之外，还包括在内部定义的计数器 cnt。

（3）波形窗口。用于显示所要观察信号的波形。若要向波形窗口添加单个 HDL 对象或多个 HDL 对象，

在“Objects”窗口中，右键单击一个或多个对象，然后从下拉菜单中选择“Add to Wave Window”选项。例如，我们把“u_led_twinkle”模块下的“cnt”计数器添加到波形窗口中，如下图所示：

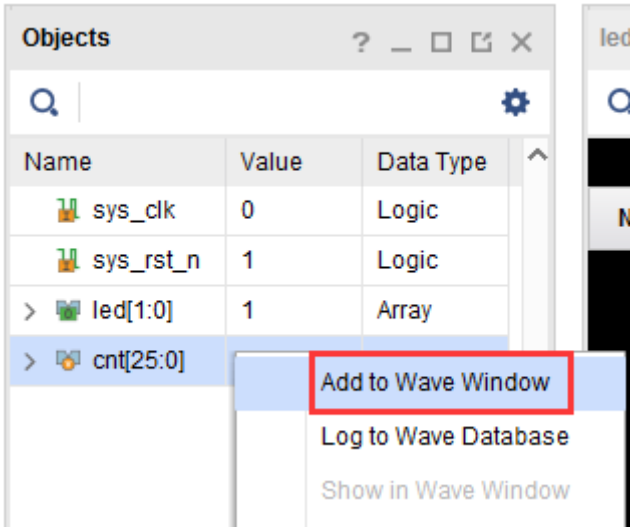


图 4.4.15 将“counter”计数器添加到波形窗口中

添加到波形窗口中的“cnt”计数器如下图所示：

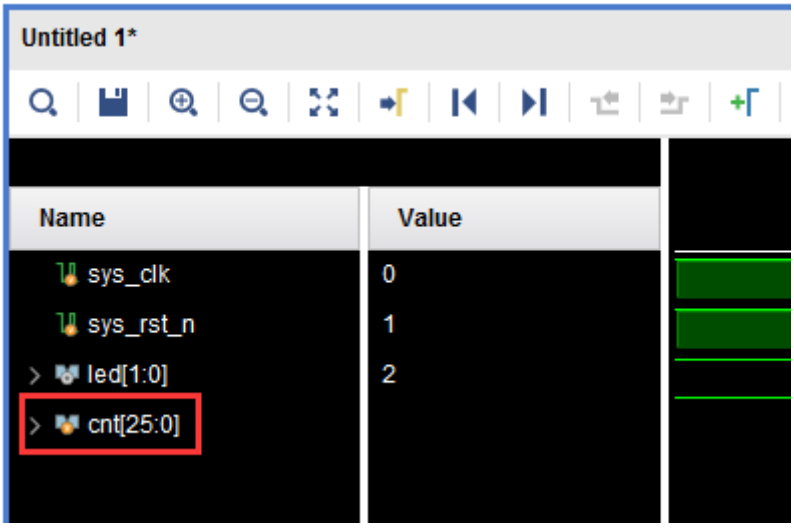


图 4.4.16 添加到波形窗口中的“cnt”计数器

一般地，每当我们进行一次仿真时，都会把当前波形的配置信息保存下来，包括波形窗口中具有哪些信号等等，以便在下一次打开仿真器进行仿真时，继续使用上一次仿真的配置信息。我们点击波形窗口中的保存按钮，如下图所示：

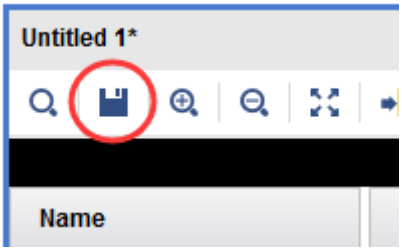


图 4.4.17 波形窗口中的保存按钮

之后会弹出“Save Waveform”对话框，提示用户将当前波形配置信息保存为“.wcfg”后缀的文件，且工具已经自动地输入了文件名“tb_led_twinkle_behav”，文件的保存目录也被工具自动设置为了当前的工程目录，所以我们保持其默认状态即可，直接点击“Save”。如下图所示：

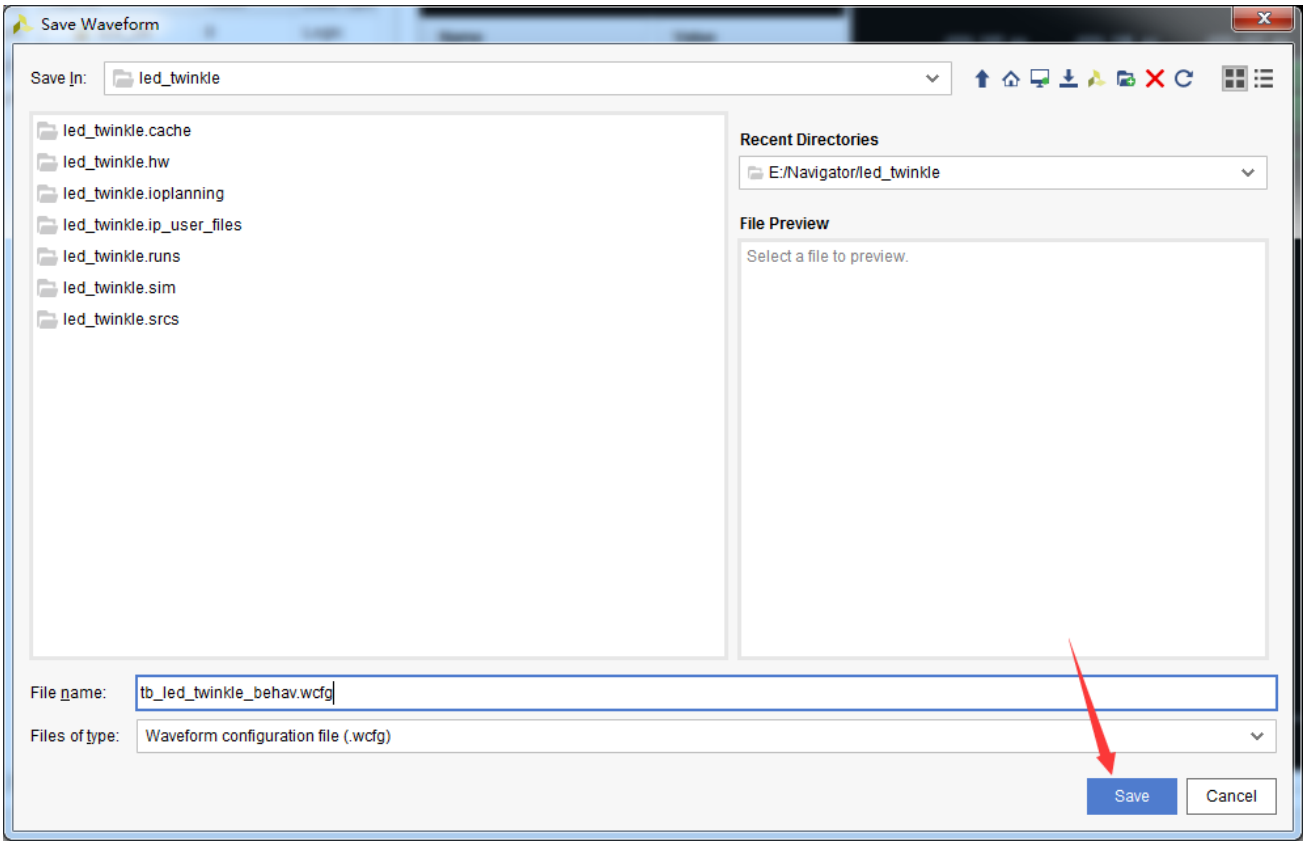


图 4.4.18 保存.wcfg 文件

注意，.wcfg 文件只是包含波形窗口的配置信息，并不包含波形的数据文件，波形的数据文件被存储在另外的文件中。

之后会弹出一个消息框，询问用户是否将刚刚创建的波形配置信息文件 tb_led_twinkle_behav.wcfg 添加到当前工程中，我们直接点击“Yes” 即可。如下图所示：

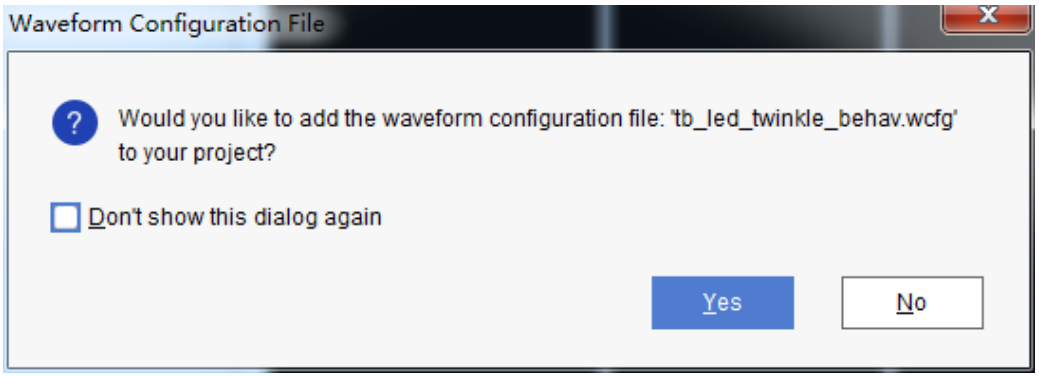


图 4.4.19 将 led_tb_behav.wcfg 文件添加到工程中

（4）仿真工具栏。仿真工具栏包含运行各个仿真动作的命令按钮，如下图所示：

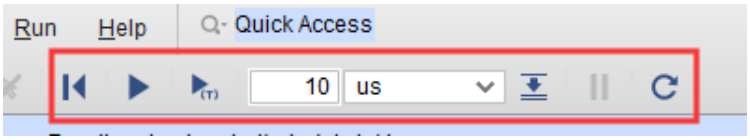


图 4.4.20 仿真工具栏

上图中的工具栏从左至右依次是：

• Restart：将仿真时间重置为零，此时波形窗口中原有的波形都会被清除。下次执行仿真时，会从 0 时刻重新开始。

• Run all：运行仿真，直到其完成所有事件或遇到 HDL 语句中的$stop 或$finish 命令为止。注意，如果没有在 TestBench 语句中加入$stop 或$finish 命令，当点击 Run all 命令时，仿真器会无休止地一直仿真下去，除非用户点击仿真工具栏中的“Break”按钮来手动地结束仿真。但是，如果此时需要仿真的设计比较复杂，则仿真器在运行仿真时会耗费电脑大量的 CPU 和内存资源，此时有可能会造成电脑卡顿甚至死机的情况。所以，如果设计比较复杂，且没有在 TestBench 语句中加入$stop 或$finish 命令，最好不要轻易点击 Run all 命令。

• Run For：运行特定的一段时间。紧随在后面的两个文本框用于设定仿真时长的数值大小和时间单位。

• Step：按步运行仿真，每一步仿真一个 HDL 语句。

• Break：暂停当前仿真。

• Relaunch：重新编译仿真源并重新启动仿真。在使用 Vivado 仿真器来调试 HDL 设计时，您可能会根据仿真结果来对您的 HDL 源代码进行修改。在修改完 HDL 源代码后，可以点击 Relaunch 按钮来重新加载 UUT 设计和 TestBench，以重新对修改后的 HDL 源代码进行仿真。此时就不需要再关闭并重新打开仿真器了。

介绍完各个窗口和命令按钮的使用后，下面我们开始进行仿真。

在刚打开仿真器时，仿真器会将 TestBench 中的信号加入到波形窗口中，并执行一段时长的仿真，仿真的时长由 Settings 设置窗口中的参数值指定，如下图所示：

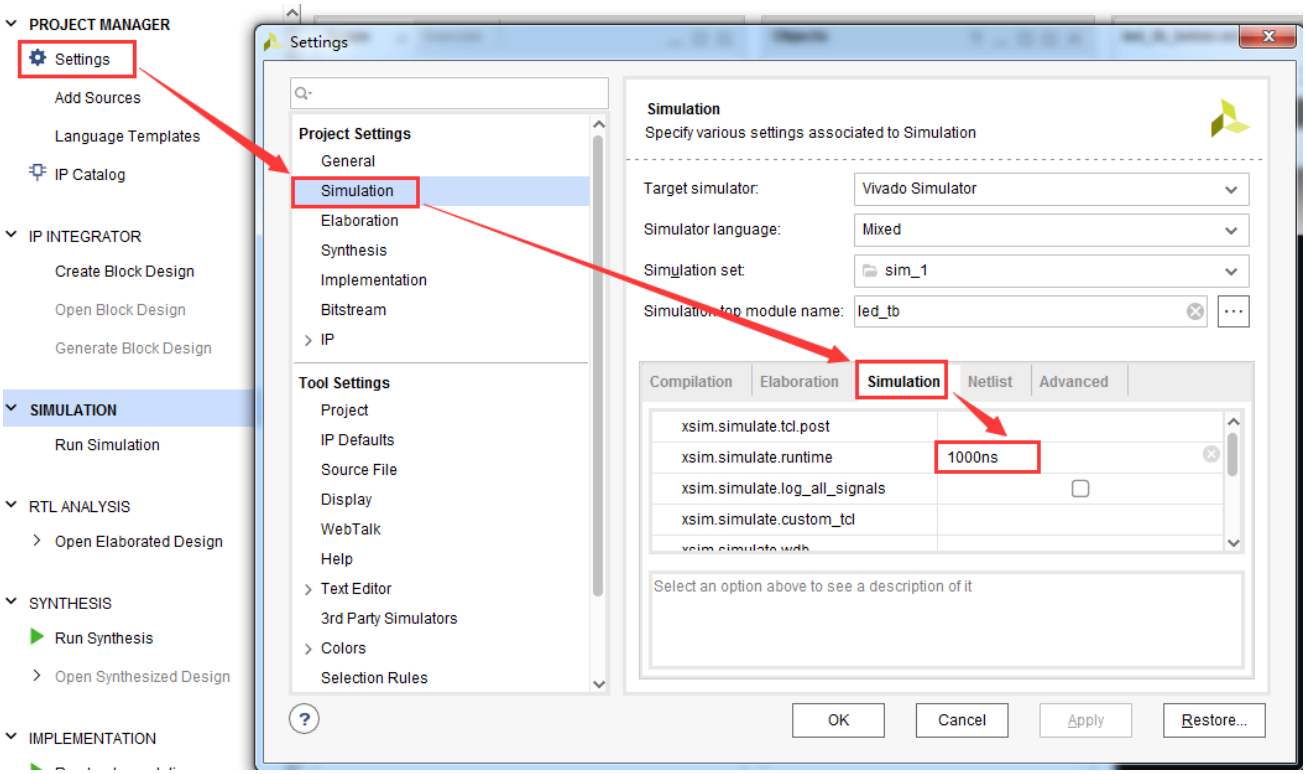


图 4.4.21 Settings 窗口中设置默认仿真时长的参数值

可以看到，仿真器启动后默认立即执行的仿真时长是 1000ns。由于我们是在默认立即执行仿真结束之后，才加入了“cnt”计数器信号，所以新加入的 cnt 信号并没有波形。此时我们需要将仿真时刻重置为 0，重新开始仿真。点击 Restart 按钮，波形窗口中的当前仿真时刻点（黄色标尺）就会回归到 0ns，且原先的所有波形都被清除，如下图所示：

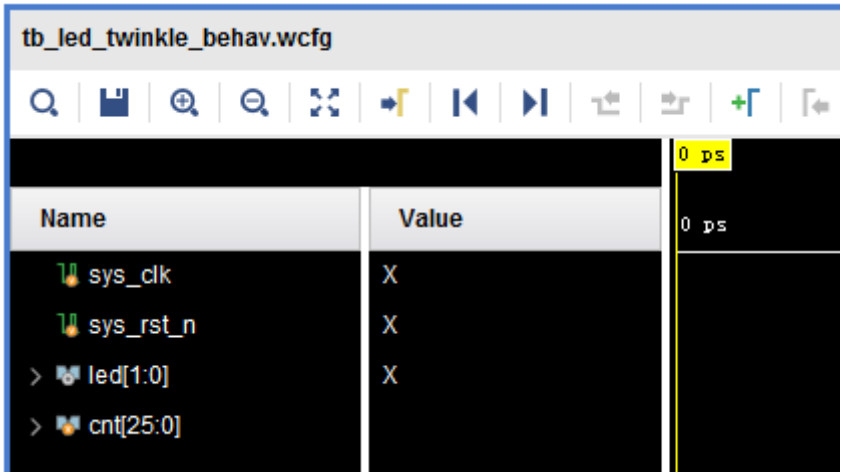


图 4.4.22 Restart 仿真

此时点击仿真工具栏中的 Run For 按钮，默认仿真时长是 10us，如下图所示：

图 4.4.23 Run For 按钮

此时就可以看到波形窗口中就出现了波形，我们点击波形窗口中的显示工具栏中的“Zoom Fit”按钮，波形就会自动缩放到整个窗口，如下图所示：

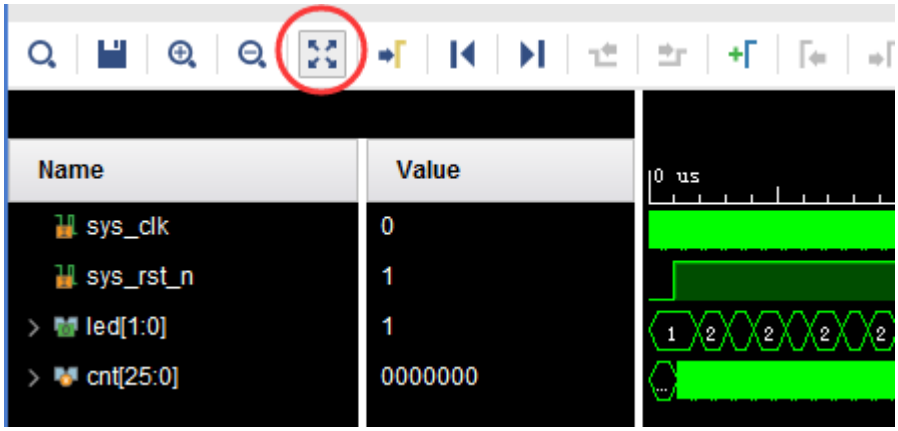


图 4.4.24 “Zoom Fit”按钮

按住“Ctrl”键并滚动鼠标滚轮，就可以放大/缩小波形，如下图所示：

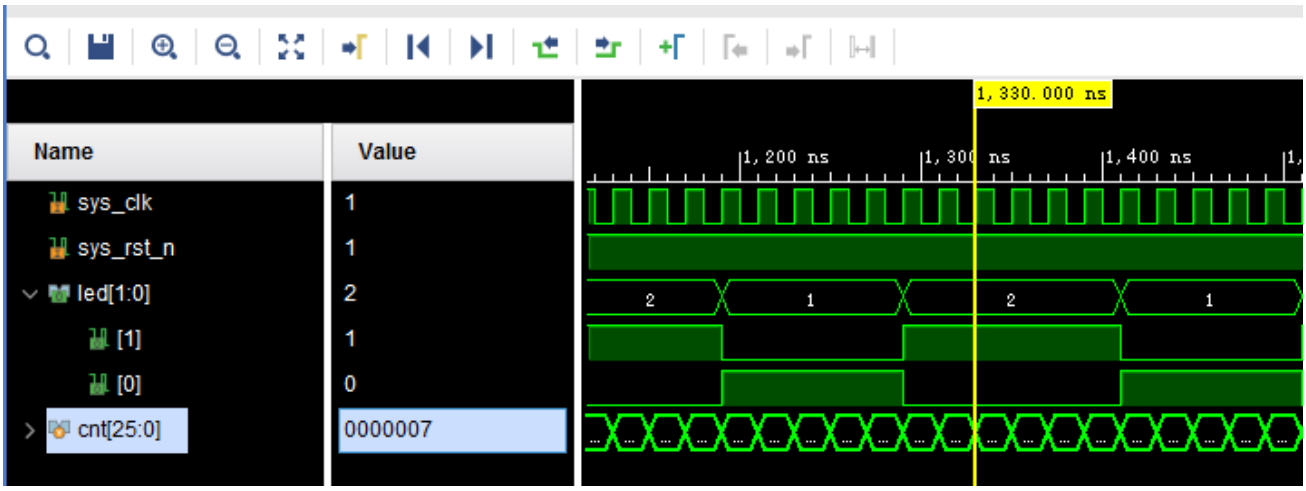


图 4.4.25 放大波形

cnt 信号默认显示为 16 进制，为了方便观察，我们将其设置为 10 进制。对 cnt 信号右键，在弹出的菜单中依次选择“Radix”——“Unsigned Decimal”，如下图所示：

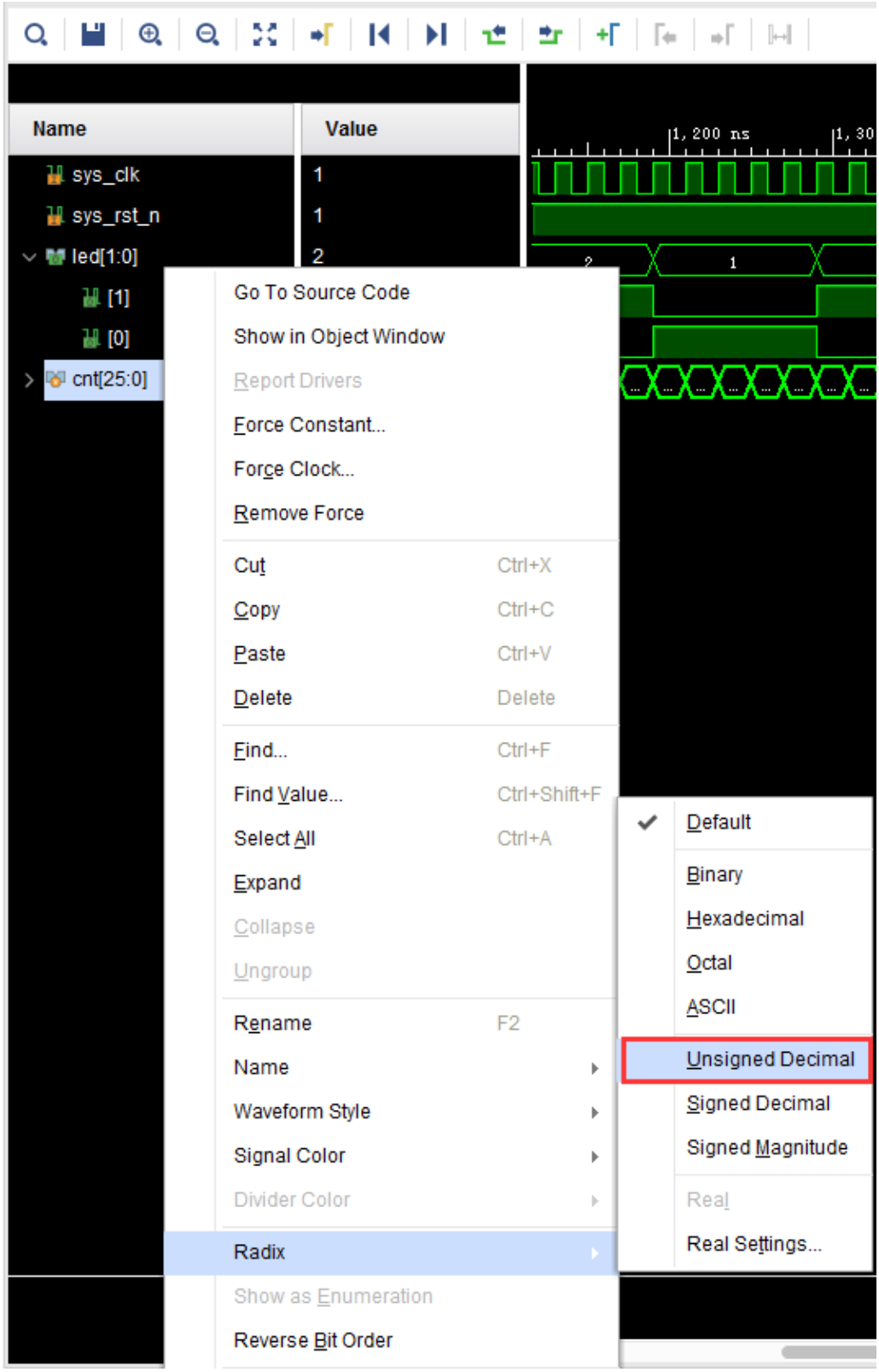


图 4.4.26 按无符号 10 进制显示

修改显示后的波形如下图所示：

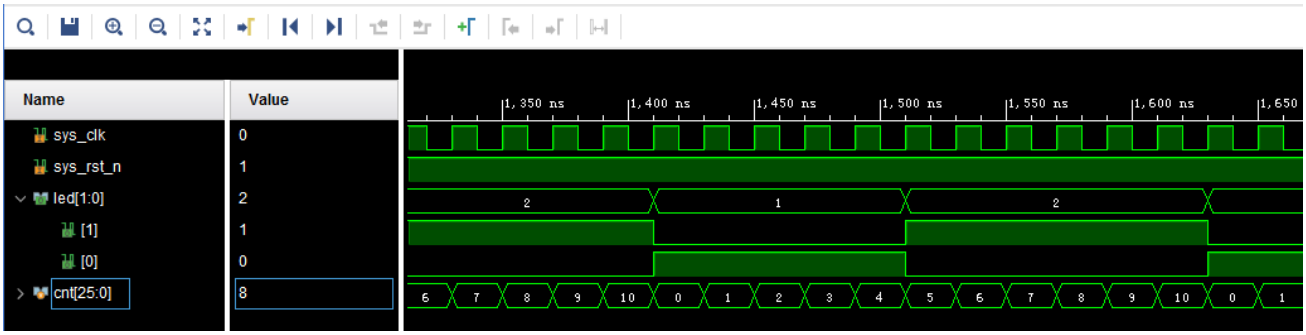


图 4.4.27 波形

可以看出，cnt 每计数到 4 和 10，两个 led 的电平状态就切换一次。证明我们的 HDL 设计达到了我们想要的功能。

在仿真结束后，可以在 Flow Navigator 窗体中找到 SIMULATION，鼠标右击 SIMULATION，选择 Close Simulation 来关闭仿真的界面，如下图所示：

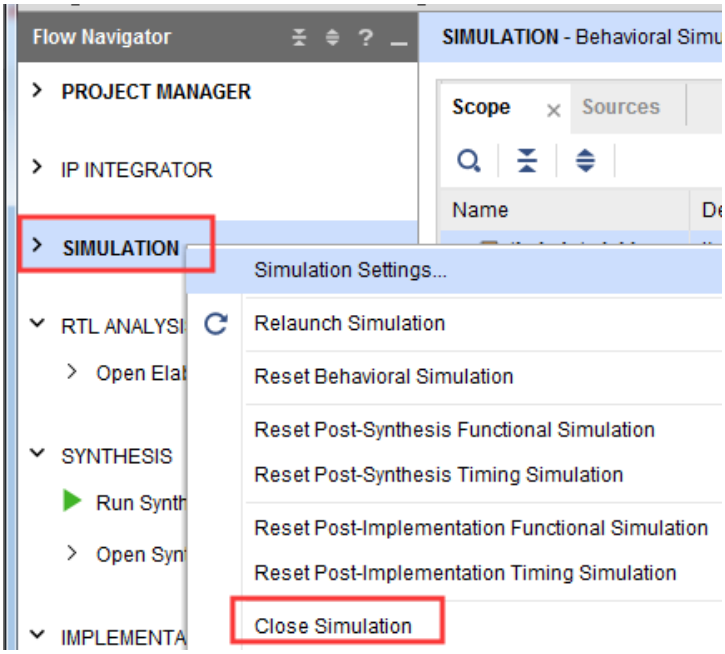


图 4.4.28 关闭仿真界面

紧接着会弹出一个关闭仿真的确认窗口，点击“OK”按钮即可。

第三篇　语法篇

上一篇，我们介绍了软件篇，本篇我们将详细介绍 FPGA 的开发语言：Verilog HDL。通过该篇的学习，你将了解到：1、Verilog 概述和基础知识；2、Verilog 程序框架和高级知识点；3、Verilog 编程规范。通过本篇的学习，希望大家能掌握 Verilog HDL 语言，并能使用 Verilog 语言进行 FPGA 编程开发和学习。

第五章　Verilog HDL 语法

Verilog HDL（Hardware Description Language）是在用途最广泛的 C 语言的基础上发展起来的一种硬件描述语言，具有灵活性高、易学易用等特点。Verilog HDL 可以在较短的时间内学习和掌握，目前已经在 FPGA 开发/IC 设计领域占据绝对的领导地位。

本章包括以下几个部分：

5.1　Verilog 概述

5.2　Verilog 基础知识

5.3　Verilog 程序框架

5.4　Verilog 高级知识点

5.5　Verilog 编程规范

5.1 Verilog 概述

本节主要描述了 Verilog HDL（以下简称 Verilog）简介、Verilog 和 VHDL 以及和 C 语言的区别。

5.1.1 Verilog 简介

Verilog 是一种硬件描述语言，以文本形式来描述数字系统硬件的结构和行为的语言，用它可以表示逻辑电路图、逻辑表达式，还可以表示数字逻辑系统所完成的逻辑功能。

数字电路设计者利用这种语言，可以从顶层到底层逐层描述自己的设计思想，用一系列分层次的模块来表示极其复杂的数字系统。然后利用电子设计自动化（EDA）工具，逐层进行仿真验证，再把其中需要变为实际电路的模块组合，经过自动综合工具转换到门级电路网表。接下来，再用专用集成电路 ASIC 或 FPGA 自动布局布线工具，把网表转换为要实现的具体电路结构。

Verilog 语言最初是于 1983 年由 Gateway Design Automation 公司为其模拟器产品开发的硬件建模语言。由于他们的模拟、仿真器产品的广泛使用，Verilog HDL 作为一种便于使用且实用的语言逐渐为众多设计者所接受。在一次努力增加语言普及性的活动中，Verilog HDL 语言于 1990 年被推向公众领域。Verilog 语言于 1995 年成为 IEEE 标准，称为 IEEE Std1364－1995，也就是通常所说的 Verilog-95。

设计人员在使用 Verilog-95 的过程中发现了一些可改进之处。为了解决用户在使用此版本 Verilog 过程中反映的问题，Verilog 进行了修正和扩展，这个扩展后的版本后来成为了电气电子工程师学会 Std1364-2001 标准，即通常所说的 Verilog-2001。Verilog-2001 是对 Verilog-95 的一个重大改进版本，它具备一些新的实用功能，例如敏感列表、多维数组、生成语句块、命名端口连接等。目前，Verilog-2001 是 Verilog 的最主流版本，被大多数商业电子设计自动化软件支持。

5.1.2 为什么需要 Verilog

在 FPGA 设计里面，我们有多种设计方式，如原理图设计方式、编写描述语言（代码）等方式。一开始很多工程师对原理图设计方式很钟爱，这种输入方式能够很直观的看到电路结构并快速理解，但是随着电路设计规模的不断增加，逻辑电路设计也越来越复杂，这种设计方式已经越来越不满足实际的项目需求了。这个时候 Verilog 语言就取而代之了，目前 Verilog 已经在 FPGA 开发/IC 设计领域占据绝对的领导地位。

5.1.3 Verilog 和 VHDL 区别

这两种语言都是用于数字电路系统设计的硬件描述语言，而且都已经是 IEEE 的标准。 VHDL 1987 年成为标准，而 Verilog 是 1995 年才成为标准的。这是因为 VHDL 是美国军方组织开发的，而 Verilog 是由一个公司的私有财产转化而来。为什么 Verilog 能成为 IEEE 标准呢？它一定有其独特的优越性才行，所以说 Verilog 有更强的生命力。

这两者有其共同的特点：

1. 能形式化地抽象表示电路的行为和结构；
2. 支持逻辑设计中层次与范围地描述；
3. 可借用高级语言地精巧结构来简化电路行为和结构；
4. 支持电路描述由高层到低层的综合转换；
5. 硬件描述和实现工艺无关。

但是两者也各有特点。Verilog 推出已经有 20 年了，拥有广泛的设计群体，成熟的资源，且 Verilog 容易掌握，只要有 C 语言的编程基础，通过比较短的时间，经过一些实际的操作，可以在 1 个月左右掌握这种语言。而 VHDL 设计相对要难一点，这个是因为 VHDL 不是很直观，一般认为至少要半年以上的专业培

训才能掌握。

近 10 年来，EDA 界一直在对数字逻辑设计中究竟用哪一种硬件描述语言争论不休，目前在美国，高层次数字系统设计领域中，应用 Verilog 和 VHDL 的比率是 80%和 20%；日本与中国台湾和美国差不多；而在欧洲 VHDL 发展的比较好；在中国很多集成电路设计公司都采用 Verilog。我们推荐大家学习 Verilog，本教程全部的例程都是使用 Verilog 开发的。

5.1.4 Verilog 和 C 的区别

Verilog 是硬件描述语言，在编译下载到 FPGA 之后，会生成电路，所以 Verilog 全部是并行处理与运行的；C 语言是软件语言，编译下载到单片机/CPU 之后，还是软件指令，而不会根据你的代码生成相应的硬件电路，而单片机/CPU 处理软件指令需要取址、译码、执行，是串行执行的。

Verilog 和 C 的区别也是 FPGA 和单片机/CPU 的区别，由于 FPGA 全部并行处理，所以处理速度非常快，这个是 FPGA 的最大优势，这一点是单片机/CPU 替代不了的。

5.2 Verilog 基础知识

本节主要讲解了 Verilog 的基础知识，包括 5 个小节，下面我们分别给大家介绍这 5 个小节的内容。

5.2.1 Verilog 的逻辑值

我们先看下逻辑电路中有四种值，即四种状态：
逻辑 0：表示低电平，也就是对应我们电路的 GND；
逻辑 1：表示高电平，也就是对应我们电路的 VCC；
逻辑 X：表示未知，有可能是高电平，也有可能是低电平；
逻辑 Z：表示高阻态，外部没有激励信号是一个悬空状态。
如下图所示：

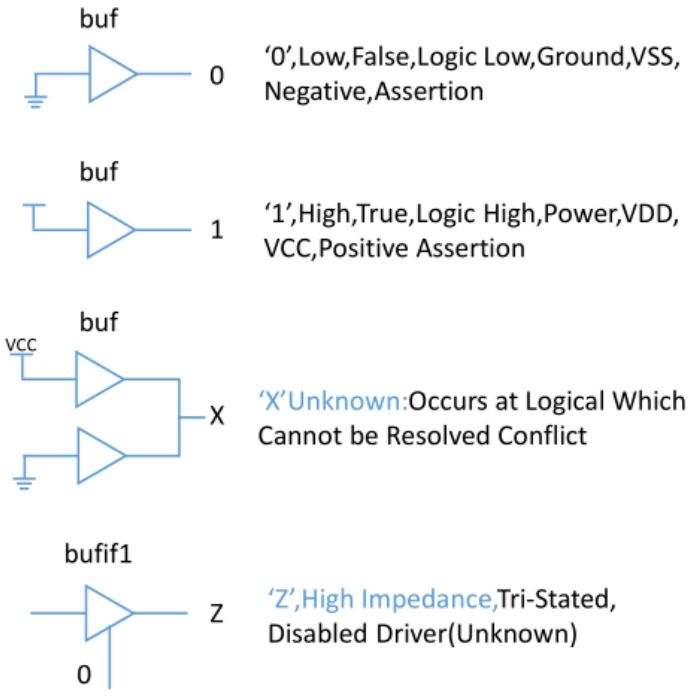


图 5.2.1 Verilog 逻辑值

5.2.2 Verilog 的标识符

定义

标识符(identifier）用于定义模块名、端口名和信号名等。Verilog 的标识符可以是任意一组字母、数字、$和_(下划线)符号的组合，但标识符的第一个字符必须是字母或者下划线。另外，标识符是区分大小写的。以下是标识符的几个例子：

Count

COUNT //与 Count 不同。

R56_68

FIVE$

虽然标识符写法很多，但是要简洁、清晰、易懂，推荐写法如下：

count

fifo_wr

不建议大小写混合使用，普通内部信号建议全部小写，参数定义建议大写，另外信号命名最好体现信号的含义。

规范建议

以下是一些书写规范的要求：

1、用有意义的有效的名字如 sum、cpu_addr 等。

2、用下划线区分词语组合，如 cpu_addr。

3、采用一些前缀或后缀，比如：时钟采用 clk 前缀：clk_50m，clk_cpu；低电平采用_n 后缀：enable_n；

4、统一缩写，如全局复位信号 rst。

5、同一信号在不同层次保持一致性，如同一时钟信号必须在各模块保持一致。

6、自定义的标识符不能与保留字（关键词）同名。

7、参数统一采用大写，如定义参数使用 SIZE。

5.2.3 Verilog 的数字进制格式

Verilog 数字进制格式包括二进制、八进制、十进制和十六进制，一般常用的为二进制、十进制和十六进制。

二进制表示如下：4’b0101 表示 4 位二进制数字 0101；

十进制表示如下：4’d2 表示 4 位十进制数字 2（二进制 0010）；

十六进制表示如下：4’ha 表示 4 位十六进制数字 a（二进制 1010），十六进制的计数方式为 0，1，2…9，a，b，c，d，e，f，最大计数为 f（f：十进制表示为 15）。

当代码中没有指定数字的位宽与进制时，默认为 32 位的十进制，比如 100，实际上表示的值为 32’d100。

5.2.4 Verilog 的数据类型

在 Verilog 语法中，主要有三大类数据类型，即寄存器类型、线网类型和参数类型。从名称中，我们可以看出，真正在数字电路中起作用的数据类型应该是寄存器类型和线网类型。

1) 寄存器类型

寄存器类型表示一个抽象的数据存储单元，它只能在 always 语句和 initial 语句中被赋值，并且它的值从一个赋值到另一个赋值过程中被保存下来。如果该过程语句描述的是时序逻辑，即 always 语句带有时钟

信号，则该寄存器变量对应为寄存器；如果该过程语句描述的是组合逻辑，即 always 语句不带有时钟信号，则该寄存器变量对应为硬件连线；寄存器类型的缺省值是 x（未知状态）。

寄存器数据类型有很多种，如 reg、integer、real 等，其中最常用的就是 reg 类型，它的使用方法如下：

```
//reg define
reg  [31:0]  delay_cnt;    //延时计数器
reg          key_flag ;    //按键标志
```

2) 线网类型

线网表示 Verilog 结构化元件间的物理连线。它的值由驱动元件的值决定，例如连续赋值或门的输出。如果没有驱动元件连接到线网，线网的缺省值为 z（高阻态）。线网类型同寄存器类型一样也是有很多种，如 tri 和 wire 等，其中最常用的就是 wire 类型，它的使用方法如下：

```
//wire define
wire         data_en;         //数据使能信号
wire  [7:0]  data    ;        //数据
```

3) 参数类型

我们再来看下参数类型，参数其实就是一个常量，常被用于定义状态机的状态、数据位宽和延迟大小等，由于它可以在编译时修改参数的值，因此它又常被用于一些参数可调的模块中，使用户在实例化模块时，可以根据需要配置参数。在定义参数时，我们可以一次定义多个参数，参数与参数之间需要用逗号隔开。这里我们需要注意的是参数的定义是局部的，只在当前模块中有效。它的使用方法如下：

```
//parameter define
parameter    DATA_WIDTH = 8;  //数据位宽为8位
```

5.2.5 Verilog 的运算符

大家看完了 Verilog 的数据类型，我们再来介绍下 Verilog 的运算符。Verilog 中的运算符按照功能可以分为下述类型：1、算术运算符、 2、关系运算符、3、逻辑运算符、 4、条件运算符、 5、位运算符、 6、移位运算符、 7、拼接运算符。下面我们分别对这些运算符进行介绍。

1) 算术运算符

算术运算符，简单来说，就是数学运算里面的加减乘除，数字逻辑处理有时候也需要进行数字运算，所以需要算术运算符。常用的算术运算符主要包括加减乘除和模除（模除运算也叫取余运算）如下表所示：

表 5.2.1 算术运算符

符号	使用方法	说明
+	a + b	a 加上 b
-	a - b	a 减去 b
*	a * b	a 乘以 b
/	a / b	a 除以 b
%	a % b	a 模除 b

大家要注意下，Verilog 实现乘除比较浪费组合逻辑资源，尤其是除法。一般 2 的指数次幂的乘除法使用移位运算来完成运算，详情可以看移位运算符章节。非 2 的指数次幂的乘除法一般是调用现成的 IP，QUARTUS/ISE 等工具软件会有提供，不过这些工具软件提供的 IP 也是由最底层的组合逻辑(与或非门等)

搭建而成的。

2) 关系运算符

关系运算符主要是用来做一些条件判断用的，在进行关系运算符时，如果声明的关系是假的，则返回值是 0，如果声明的关系是真的，则返回值是 1；所有的关系运算符有着相同的优先级别，关系运算符的优先级别低于算术运算符的优先级别如下表所示。

表 5.2.2 关系运算符

符号	使用方法	说明
>	a > b	a 大于 b
<	a < b	a 小于 b
>=	a >= b	a 大于等于 b
<=	a <= b	a 小于等于 b
==	a == b	a 等于 b
!=	a != b	a 不等于 b

3) 逻辑运算符

逻辑运算符是连接多个关系表达式用的，可实现更加复杂的判断，一般不单独使用，都需要配合具体语句来实现完整的意思，如下表所示。

表 5.2.3 逻辑运算符

符号	使用方法	说明
!	!a	a的非，如果a为0，那么a的非是1。
&&	a && b	a 与上 b，如果a和b都为1，a&&b结果才为1，表示真。
||	a || b	a 或上 b，如果a或者b有一个为1，a||b结果为1，表示真。

4) 条件运算符

条件操作符一般来构建从两个输入中选择一个作为输出的条件选择结构，功能等同于 always 中的 if-else 语句，如下表所示。

表 5.2.4 条件运算符

符号	使用方法	说明
? :	a ? b : c	如果 a 为真，就选择 b，否则选择 c

5) 位运算符

位运算符是一类最基本的运算符，可以认为它们直接对应数字逻辑中的与、或、非门等逻辑门。常用的位运算符如下表所示。

表 5.2.5 位运算符

符号	使用方法	说明
~	~a	将 a 的每个位进行取反
&	a & b	将 a 的每个位与 b 相同的位进行相与
|	a | b	将 a 的每个位与 b 相同的位进行相或
^	a ^ b	将 a 的每个位与 b 相同的位进行异或

位运算符的与、或、非与逻辑运算符逻辑与、逻辑或、逻辑非使用时候容易混淆，逻辑运算符一般用

在条件判断上，位运算符一般用在信号赋值上。

6) 移位运算符

移位运算符包括左移位运算符和右移位运算符，这两种移位运算符都用 0 来填补移出的空位。如下表所示。

表 5.2.6 移位运算符

符号	使用方法	说明
<<	a << b	将 a 左移 b 位
>>	a >> b	将 a 右移 b 位

假设 a 有 8bit 数据位宽，那么 a<<2，表示 a 左移 2bit，a 还是 8bit 数据位宽，a 的最高 2bit 数据被移位丢弃了，最低 2bit 数据固定补 0。如果 a 是 3（二进制：00000011），那么 3 左移 2bit，3<<2，就是 12（二进制：00001100）。一般使用左移位运算代替乘法，右移位运算代替除法，但是这种也只能表示 2 的指数次幂的乘除法。

7) 拼接运算符

Verilog 中有一个特殊的运算符是 C 语言中没有的，就是位拼接运算符。用这个运算符可以把两个或多个信号的某些位拼接起来进行运算操作。如下表所示。

表 5.2.7 位拼接运算符

符号	使用方法	说明
{}	{a, b}	将 a 和 b 拼接起来，作为一个新信号

8) 运算符的优先级

介绍完了这么多运算符，大家可能会想到究竟哪个运算符高，哪个运算符低。为了便于大家查看这些运算符的优先级，我们将它们制作成了表格，如下表所示。

表 5.2.8 运算符的优先级

运算符	优先级
!、 ~	最高
*、 /、 %	次高
+、 -	
<<、 >>	
<、 <=、 >、 >=	
==、 !=、 ===、 !==	
&	
^、 ^~	
|	
&&	
||	次低
?	最低

5.3 Verilog 程序框架

在介绍 Verilog 程序框架之前，我们先来看下 Verilog 一些基本语法，基础语法主要包括注释和关键字。

5.3.1 注释

Verilog HDL 中有两种注释的方式，一种是以“/*”符号开始，“*/”结束，在两个符号之间的语句都是注释语句，因此可扩展到多行。如：

/* statement1 ，
statement2，
.....
statementn */
以上 n 个语句都是注释语句。
另一种是以//开头的语句，它表示以//开始到本行结束都属于注释语句。如：
//statement1
我们建议的写法：使用//作为注释。

5.3.2 关键字

Verilog 和 C 语言类似，都因编写需要定义了一系列保留字，叫做关键字（或关键词）。这些保留字是识别语法的关键。我们给大家列出了 Verilog 中的关键字，如下表所示。

表 5.3.1 Verilog 的所有关键字

and	always	assign	begin	buf
bufif0	bufif1	case	casex	casez
cmos	deassign	default	defparam	disable
edge	else	end	endcase	endfunction
endprimitive	endmodule	endspecify	endtable	endtask
event	for	force	forever	fork
function	highz0	highz1	if	ifnone
initial	inout	input	integer	join
large	macromodule	medium	module	nand
negedge	nor	not	notif0	notif1
nmos	or	output	parameter	pmos
posedge	primitive	pulldown	pullup	pull0
pull1	rcmos	real	realtime	reg
release	repeat	rnmos	rpmos	rtran
rtranif0	rtranif1	scalared	small	specify
specparam	strength	strong0	strong1	supply0
supply1	table	task	tran	tranif0
tranif1	time	tri	triand	trior
trireg	tri0	tri1	vectored	wait
wand	weak0	weak1	while	wire

wor	xnor	xor		

虽然上表列了很多，但是实际经常使用的不是很多，实际经常使用的主要如下表所示。

表 5.3.2 Verilog 常用的关键字

关键字	含义
module	模块开始定义
input	输入端口定义
output	输出端口定义
inout	双向端口定义
parameter	信号的参数定义
wire	wire信号定义
reg	reg信号定义
always	产生reg信号语句的关键字
assign	产生wire信号语句的关键字
begin	语句的起始标志
end	语句的结束标志
posedge/negedge	时序电路的标志
case	Case语句起始标记
default	Case语句的默认分支标志
endcase	Case语句结束标记
if	if/else语句标记
else	if/else语句标记
for	for语句标记
endmodule	模块结束定义

注意只有小写的关键字才是保留字。例如，标识符 always(这是个关键词)与标识符 ALWAYS(非关键词)是不同的。

5.3.3 程序框架

我们以 LED 流水灯程序为例来给大家展示 Verilog 的程序框架，代码如下所示（注意：代码中前面的行号只是为了方便大家阅读代码与快速定位到行号的位置，在实际编写代码时不可以添加行号，否则编译代码时会报错）。

```
1  module led(
2      input               sys_clk   ,  //系统时钟
3      input               sys_rst_n,  //系统复位，低电平有效
4      output  reg  [3:0]  led         //4位LED灯
5      );
6
7  //parameter define
8  parameter  WIDTH     = 25         ;
9  parameter  COUNT_MAX = 25_000_000;  //板载50M时钟=20ns，0.5s/20ns=25000000，需要25bit
10                                     //位宽
```

```
11
12 //reg define
13 reg    [WIDTH-1:0]  counter     ;
14 reg    [1:0]        led_ctrl_cnt;
15
16 //wire define
17 wire                counter_en  ;
18
19 //**************************************************************************************
20 //**                                main code
21 //**************************************************************************************
22
23 //计数到最大值时产生高电平使能信号
24 assign  counter_en = (counter == (COUNT_MAX - 1'b1))  ?  1'b1  :  1'b0;
25
26 //用于产生0.5秒使能信号的计数器
27 always @(posedge sys_clk or negedge sys_rst_n) begin
28     if (sys_rst_n == 1'b0)
29         counter <= 1'b0;
30     else if (counter_en)
31         counter <= 1'b0;
32     else
33         counter <= counter + 1'b1;
34 end
35
36 //led流水控制计数器
37 always @(posedge sys_clk or negedge sys_rst_n) begin
38     if (sys_rst_n == 1'b0)
39         led_ctrl_cnt <= 2'b0;
40     else if (counter_en)
41         led_ctrl_cnt <= led_ctrl_cnt + 2'b1;
42 end
43
44 //通过控制IO口的高低电平实现发光二极管的亮灭
45 always @(posedge sys_clk or negedge sys_rst_n) begin
46     if (sys_rst_n == 1'b0)
47         led <= 4'b0;
48     else begin
49         case (led_ctrl_cnt)
50             2'd0 : led <= 4'b0001;
51             2'd1 : led <= 4'b0010;
```

```
52              2'd2 : led <= 4'b0100;
53              2'd3 : led <= 4'b1000;
54              default : ;
55          endcase
56      end
57  end
58
59  endmodule
```

首先//开头的都是注释，这个之前我们讲解过了。下面我们来看下具体的解释。

第 1 行为模块定义，模块定义以 module 开始，endmodule 结束，如 59 行所示。

其次 2 到 5 行为端口定义，需要定义 led 模块的输入信号和输出信号，此处输入信号为系统时钟和复位信号，输出为 led 控制信号。

7 到 9 行为参数 parameter 定义，语法如 7 到 9 行所示，定义 parameter 的好处是可以灵活改变参数数字就能控制一些计数器最大计数值或者信号位宽的最大位宽。

12 到 14 行为 reg 信号定义，reg 信号一般情况下代表寄存器，比如此处控制 0.5 秒使能信号的计数器 counter。

16 到 17 行为 wire 信号定义，wire 信号就是硬件连线，比如此处的 counter_en，代表计数到最大值时产生高电平使能，本质上是一个硬件连线，其实代表的是一些计数器/寄存器做逻辑判断的结果。

19 到 21 行为 moudle 开始的注释，不添加工具综合也不会报错，但是我们推荐添加，作为一个良好的编程规范。

23 到 24 行为 assign 语句的样式，条件成立选择 1，否则选择 0。

26 到 34 行是 always 语句的样式，27 行代表在时钟上升沿或者复位的下降沿进行信号触发。begin/end 代表语句的开始和结束。28 到 33 行为 if/else 语句，和 C 语言是比较类似的。29 行的 “<=”标记代表信号是非阻塞赋值，信号赋值有非阻塞赋值和阻塞赋值两个方式，这个我们后面会详细解释。

36 和 42 行也是一个 always 语句，和 26 到 34 行类似。

44 和 57 行也是一个 always 语句，不过这个 always 语句中嵌入了一个 case 语句，case 语句的语法如 49 到 55 行所示，需要一个 case 关键字开始，endcase 关键字结束，default 作为默认分支，和 C 语言也是类似的。当然 case 语句也可以用在不带时钟的 always 语句中，不过本例子的 always 都是带有时钟的。不带时钟的 always 和带时钟的 always 语句的差异这个我们后面也会详细解释。

59 行是 endmodule 标记，代表模块的结束。

在这里需要补充一点的是，一些初学者可能会有这样一个疑问，在 always 语句中编写 if 语句或 else 语句时，后面需要加 begin 和 end 吗？其实这个主要看 if 条件后面跟着几条赋值语句，如果只有一条赋值语句时，if 后面可以加 begin 和 end，也可以不加；如果超过一条赋值语句时，就必须加上 begin 和 end。

if 条件只有一条赋值语句时，下面两种写法都是可以的，这里更推荐第一种写法，因为第二种写法会占用更多的行号，代码如下所示：

```
if(en == 1'b1)
    a <= 1'b1;
```

或者

```
if(en == 1'b1) begin
    a <= 1'b1;
end
```

对于if条件超过一条赋值语句的情况，必须添加begin和end，代码如下所示：

```
if(en == 1'b1) begin
    b <= 1'b1;
    c <= 1'b1;
end
```

好了，程序框架就讲解完了，大家是不是觉得也很简单呢？这些都是基本的语法规范，希望大家能记住这些基础的知识点。如果有些地方大家还是觉得比较抽象，很难理解，没有关系，相信大家会在后面的学习中，会慢慢理解的。

5.4 Verilog 高级知识点

前几节主要介绍了 Verilog 一些基础的知识点和程序框架，本节给大家介绍一些高级的知识点。高级知识点包括阻塞赋值和非阻塞赋值、assign 和 always 语句差异、什么是锁存器、状态机、模块化设计等。

5.4.1 阻塞赋值（Blocking）

阻塞赋值，顾名思义，即在一个 always 块中，后面的语句会受到前语句的影响，具体来说，在同一个 always 中，一条阻塞赋值语句如果没有执行结束，那么该语句后面的语句就不能被执行，即被“阻塞”。也就是说 always 块内的语句是一种顺序关系，这里和 C 语言很类似。符号“=”用于阻塞的赋值（如:b = a;），阻塞赋值“=”在 begin 和 end 之间的语句是顺序执行，属于串行语句。

在这里定义两个缩写：

RHS：赋值等号右边的表达式或变量可以写作 RHS 表达式或 RHS 变量；

LHS：赋值等号左边的表达式或变量可以写作 LHS 表达式或 LHS 变量；

阻塞赋值的执行可以认为是只有一个步骤的操作，即计算 RHS 的值并更新 LHS，此时不允许任何其他语句的干扰，所谓的阻塞的概念就是值在同一个 always 块中，其后面的赋值语句从概念上来讲是在前面一条语句赋值完成后才执行的。

为了方便大家理解阻塞赋值的概念以及阻塞赋值和非阻塞赋值的区别，我们这里以在时序逻辑下使用阻塞赋值为例来实现这样一个功能：在复位的时候，a=1，b=2，c=3；而在没有复位的时候，a 的值清零，同时将 a 的值赋值给 b，b 的值赋值给 c，代码以及信号波形图如下图所示：

```
always @(posedge clk or negedge rst_n) begin
    if(!rst_n)begin
        a = 1;
        b = 2;
        c = 3;
    end
    else begin
        a = 0;
        b = a;
        c = b;
    end
end
```

图 5.4.1 阻塞赋值代码

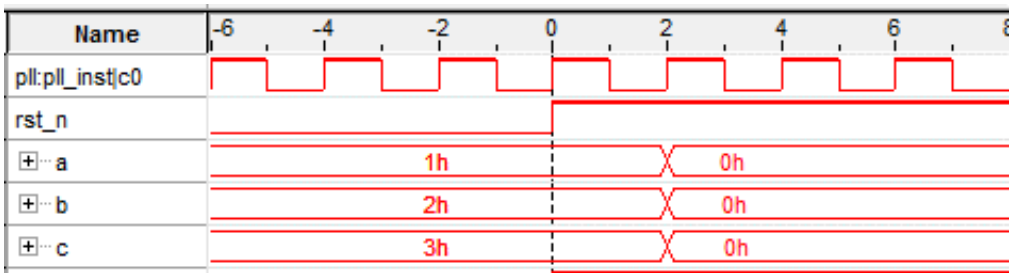


图 5.4.2 阻塞赋值的信号波形图

代码中使用的是阻塞赋值语句，从波形图中可以看到，在复位的时候（rst_n=0），a=1，b=2，c=3；而结束复位之后（波形图中的 0 时刻），当 clk 的上升沿到来时（波形图中的 2 时刻），a=0，b=0，c=0。这是因为阻塞赋值是在当前语句执行完成之后，才会执行后面的赋值语句，因此首先执行的是 a=0，赋值完成后将 a 的值赋值给 b，由于此时 a 的值已经为 0，所以 b=a=0，最后执行的是将 b 的值赋值给 c，而 b 的值已经赋值为 0，所以 c 的值同样等于 0。

5.4.2 非阻塞赋值（Non-Blocking）

符号“<=”用于非阻塞赋值（如:b <= a;），非阻塞赋值是由时钟节拍决定，在时钟上升到来时，执行赋值语句右边，然后将 begin-end 之间的所有赋值语句同时赋值到赋值语句的左边，注意：是 begin—end 之间的所有语句，一起执行，且一个时钟只执行一次，属于并行执行语句。这个是和 C 语言最大的一个差异点，大家要逐步理解并行执行的概念。

非阻塞赋值的操作过程可以看作两个步骤：

（1）赋值开始的时候，计算 RHS；

（2）赋值结束的时候，更新 LHS。

所谓的非阻塞的概念是指，在计算非阻塞赋值的 RHS 以及 LHS 期间，允许其它的非阻塞赋值语句同时计算 RHS 和更新 LHS。

我们下面使用非阻塞赋值同样来实现这样一个功能：在复位的时候，a=1，b=2，c=3；而在没有复位的时候，a 的值清零，同时将 a 的值赋值给 b，b 的值赋值给 c，代码以及信号波形图如下图所示：

```
always @(posedge clk or negedge rst_n) begin
    if(!rst_n)begin
        a <= 1;
        b <= 2;
        c <= 3;
    end
    else begin
        a <= 0;
        b <= a;
        c <= b;
    end
end
```

图 5.4.3 非阻塞赋值代码

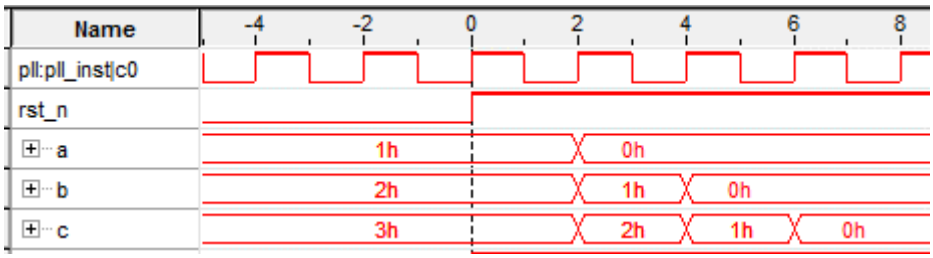


图 5.4.4 非阻塞赋值的信号波形图

代码中使用的是非阻塞赋值语句，从波形图中可以看到，在复位的时候（rst_n=0），a=1，b=2，c=3；而结束复位之后（波形图中的 0 时刻），当 clk 的上升沿到来时（波形图中的 2 时刻），a=0，b=1，c=2。这是因为非阻塞赋值在计算 RHS 和更新 LHS 期间，允许其它的非阻塞赋值语句同时计算 RHS 和更新 LHS。在波形图中的 2 时刻，RHS 的表达是 0、a、b，分别等于 0、1、2，这三条语句是同时更新 LHS，所以 a、b、c 的值分别等于 0、1、2。

在了解了阻塞赋值和非阻塞赋值的区别之后，有些朋友可能还是对什么时候使用阻塞赋值，什么时候使用非阻塞赋值有些疑惑，在这里给大家总结如下。

在描述组合逻辑电路的时候，使用阻塞赋值，比如 assign 赋值语句和不带时钟的 always 赋值语句，这种电路结构只与输入电平的变化有关系，代码如下：

示例1：assign赋值语句

```
assign  data  =  (data_en == 1'b1)  ?  8'd255  :  8'd0;
```

示例2：不带时钟的always语句

```
always @(*) begin
    if (en) begin
        a = a0;
        b = b0;
    end
    else begin
        a = a1;
        b = b1;
    end
end
```

在描述时序逻辑的时候，使用非阻塞赋值，综合成时序逻辑的电路结构，比如带时钟的 always 语句；这种电路结构往往与触发沿有关系，只有在触发沿时才可能发生赋值的变化，代码如下：

示例 3：

```
    always @(posedge sys_clk or negedge sys_rst_n) begin
    if (!sys_rst_n) begin
            a <= 1'b0;
            b <= 1'b0;
    end
    else begin
            a <= c;
            b <= d;
    end
```

```
    end
```

5.4.3 assign 和 always 区别

assign 语句和 always 语句是 Verilog 中的两个基本语句，这两个都是经常使用的语句。

assign 语句使用时不能带时钟。

always 语句可以带时钟，也可以不带时钟。在 always 不带时钟时，逻辑功能和 assign 完全一致，都是只产生组合逻辑。比较简单的组合逻辑推荐使用 assign 语句，比较复杂的组合逻辑推荐使用 always 语句。示例如下：

```
24  assign  counter_en = (counter == (COUNT_MAX - 1'b1))  ?  1'b1  :  1'b0;
45  always @(*) begin
49          case (led_ctrl_cnt)
50              2'd0    : led = 4'b0001;
51              2'd1    : led = 4'b0010;
52              2'd2    : led = 4'b0100;
53              2'd3    : led = 4'b1000;
54              default : led = 4'b0000;
55          endcase
57  end
```

5.4.4 带时钟和不带时钟的 always

always 语句可以带时钟，也可以不带时钟。在 always 不带时钟时，逻辑功能和 assign 完全一致，虽然产生的信号定义还是 reg 类型，但是该语句产生的还是组合逻辑。

```
44  reg   [3:0] led;
45  always @(*) begin
49          case (led_ctrl_cnt)
50              2'd0    : led = 4'b0001;
51              2'd1    : led = 4'b0010;
52              2'd2    : led = 4'b0100;
53              2'd3    : led = 4'b1000;
54              default : led = 4'b0000;
55          endcase
57  end
```

在 always 带时钟信号时，这个逻辑语句才能产生真正的寄存器，如下示例 counter 就是真正的寄存器。

```
26  //用于产生 0.5 秒使能信号的计数器
27  always @(posedge sys_clk or negedge sys_rst_n) begin
28      if (sys_rst_n == 1'b0)
29          counter <= 1'b0;
30      else if (counter_en)
31          counter <= 1'b0;
32      else
33          counter <= counter + 1'b1;
```

```
34  end
```

5.4.5 什么是 latch

latch 是指锁存器，是一种对脉冲电平敏感的存储单元电路。锁存器和寄存器都是基本存储单元，锁存器是电平触发的存储器，寄存器是边沿触发的存储器。两者的基本功能是一样的，都可以存储数据。锁存器是组合逻辑产生的，而寄存器是在时序电路中使用，由时钟触发产生的。

latch 的主要危害是会产生毛刺（glitch），这种毛刺对下一级电路是很危险的。并且其隐蔽性很强，不易查出。因此，在设计中，应尽量避免 latch 的使用。

代码里面出现 latch 的两个原因是在组合逻辑中，if 或者 case 语句不完整的描述，比如 if 缺少 else 分支，case 缺少 default 分支，导致代码在综合过程中出现了 latch。解决办法就是 if 必须带 else 分支，case 必须带 default 分支。

大家需要注意下，只有不带时钟的 always 语句 if 或者 case 语句不完整才会产生 latch，带时钟的语句 if 或者 case 语句不完整描述不会产生 latch。

下面为缺少 else 分支的带时钟的 always 语句和不带时钟的 always 语句，通过实际产生的电路图可以看到第二个是有一个 latch 的，第一个仍然是普通的带有时钟的寄存器。

```
always @(posedge clk)begin
        if(enable)
          q <= data;
         //else
          // q <= 0 ;
end
endmodule
```

```
always @(*)begin
        if(enable)
                    q <= data;
//                  else
//                    q <= 0 ;
end
endmodule
```

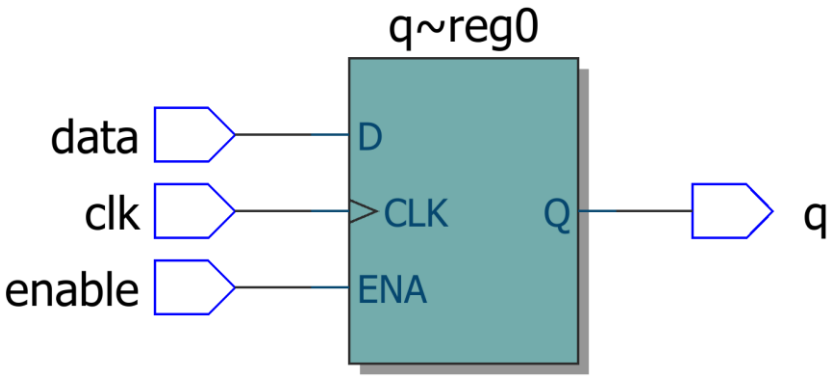


图 5.4.5 缺少 else 的带时钟的 always 语句电路图

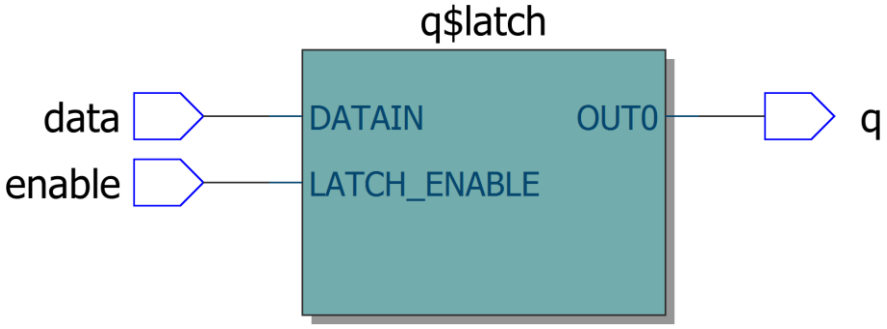


图 5.4.6 缺少 else 的不带时钟的 always 语句电路图

5.4.6 状态机

Verilog 是硬件描述语言，硬件电路是并行执行的，当需要按照流程或者步骤来完成某个功能时，代码中通常会使用很多个 if 嵌套语句来实现，这样就增加了代码的复杂度，以及降低了代码的可读性，这个时候就可以使用状态机来编写代码。状态机相当于一个控制器，它将一项功能的完成分解为若干步，每一步对应于二进制的一个状态，通过预先设计的顺序在各状态之间进行转换，状态转换的过程就是实现逻辑功能的过程。

状态机，全称是有限状态机（Finite State Machine，缩写为 FSM），是一种在有限个状态之间按一定规律转换的时序电路，可以认为是组合逻辑和时序逻辑的一种组合。状态机通过控制各个状态的跳转来控制流程，使得整个代码看上去更加清晰易懂，在控制复杂流程的时候，状态机优势明显，因此基本上都会用到状态机，如 SDRAM 控制器等。在本手册提供的例程中，会有多个用到状态机设计的例子，希望大家能够慢慢体会和理解，并且能够熟练掌握。

根据状态机的输出是否与输入条件相关，可将状态机分为两大类，即摩尔(Moore)型状态机和米勒(Mealy)型状态机。

- Mealy 状态机：组合逻辑的输出不仅取决于当前状态，还取决于输入状态。
- Moore 状态机：组合逻辑的输出只取决于当前状态。

1) Mealy 状态机

米勒状态机的模型如下图所示，模型中第一个方框是指产生下一状态的组合逻辑 F，F 是当前状态和输入信号的函数，状态是否改变、如何改变，取决于组合逻辑 F 的输出；第二框图是指状态寄存器，其由一组触发器组成，用来记忆状态机当前所处的状态，状态的改变只发生在时钟的跳边沿；第三个框图是指产生输出的组合逻辑 G，状态机的输出是由输出组合逻辑 G 提供的，G 也是当前状态和输入信号的函数。

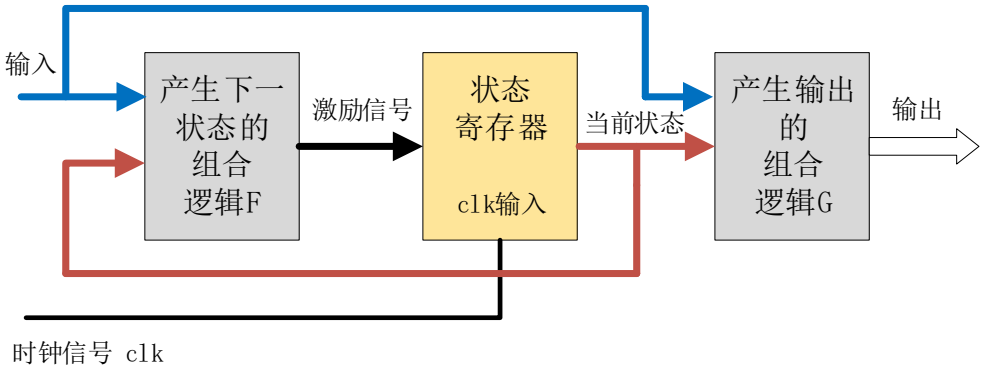


图 5.4.7 Mealy 状态机模型

2) Moore 状态机

摩尔状态机的模型如下图所示，对比米勒状态机的模型可以发现，其区别在于米勒状态机的输出由当前状态和输入条件决定的，而摩尔状态机的输出只取决于当前状态。

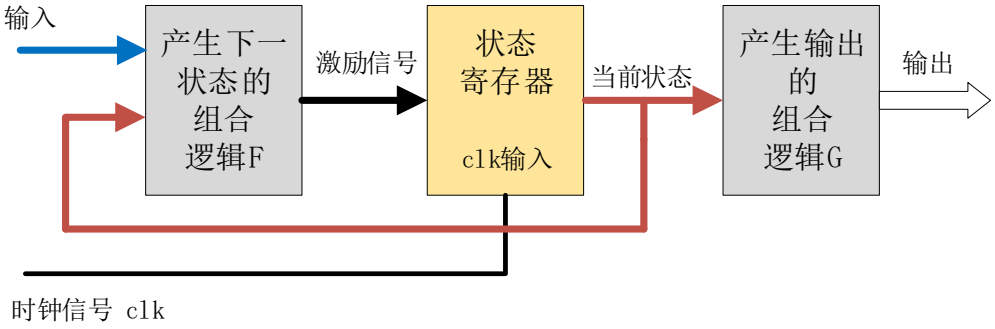


图 5.4.8 Moore 状态机模型

3) 三段式状态机

根据状态机的实际写法，状态机还可以分为一段式、二段式和三段式状态机。

一段式：整个状态机写到一个 always 模块里面，在该模块中既描述状态转移，又描述状态的输入和输出。不推荐采用这种状态机，因为从代码风格方面来讲，一般都会要求把组合逻辑和时序逻辑分开；从代码维护和升级来说，组合逻辑和时序逻辑混合在一起不利于代码维护和修改，也不利于约束。

二段式：用两个 always 模块来描述状态机，其中一个 always 模块采用同步时序描述状态转移；另一个模块采用组合逻辑判断状态转移条件，描述状态转移规律以及输出。不同于一段式状态机的是，它需要定义两个状态，现态和次态，然后通过现态和次态的转换来实现时序逻辑。

三段式：在两个 always 模块描述方法基础上，使用三个 always 模块，一个 always 模块采用同步时序描述状态转移，一个 always 采用组合逻辑判断状态转移条件，描述状态转移规律，另一个 always 模块描述状态输出(可以用组合电路输出，也可以时序电路输出)。

实际应用中三段式状态机使用最多，因为三段式状态机将组合逻辑和时序分开，有利于综合器分析优化以及程序的维护；并且三段式状态机将状态转移与状态输出分开，使代码看上去更加清晰易懂，提高了代码的可读性，推荐大家使用三段式状态机，本文也着重讲解三段式。

三段式状态机的基本格式是：

第一个 always 语句实现同步状态跳转；

第二个 always 语句采用组合逻辑判断状态转移条件；

第三个 always 语句描述状态输出(可以用组合电路输出，也可以时序电路输出)。

在开始编写状态机代码之前，一般先画出状态跳转图，这样在编写代码时思路会比较清晰，下面以一个 7 分频为例（对于分频等较简单的功能，可以不使用状态机，这里只是演示状态机编写的方法），状态跳转图如下图所示：

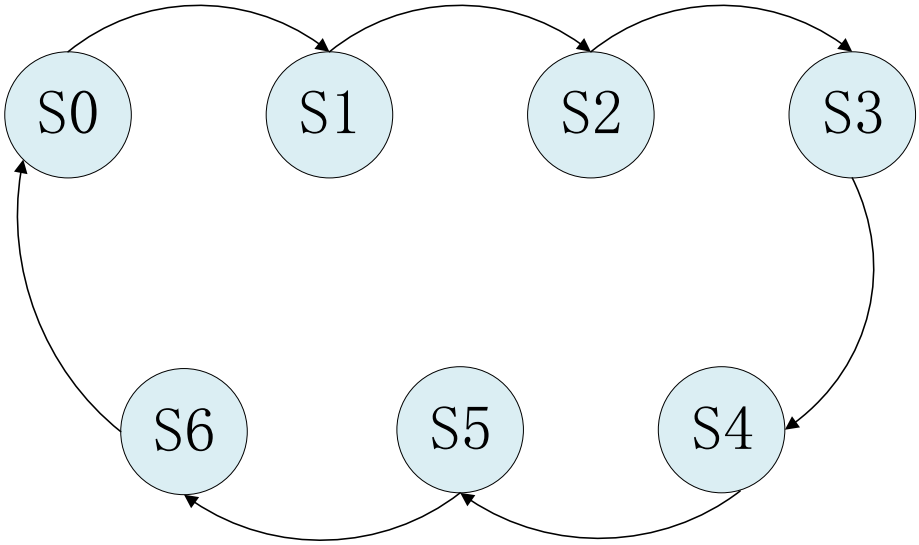


图 5.4.9 七分频状态跳转图

状态跳转图画完之后，接下来通过 parameter 来定义各个不同状态的参数，如下代码所示：

```
parameter S0 = 7'b0000001;      //独热码定义方式
parameter S1 = 7'b0000010;
parameter S2 = 7'b0000100;
parameter S3 = 7'b0001000;
parameter S4 = 7'b0010000;
parameter S5 = 7'b0100000;
    parameter S6 = 7'b1000000;
```

这里是使用独热码的方式来定义状态机，每个状态只有一位为 1，当然也可以直接定义成十进制的 0，1，2……7。

因为我们定义成独热码的方式，每一个状态的位宽为 7 位，接下来还需要定义两个 7 位的寄存器，一个用来表示当前状态，另一个用来表示下一个状态，如下所示：

```
reg  [6:0]   curr_st     ;     //当前状态
reg  [6:0]   next_st     ;     //下一个状态
```

接下来就可以使用三个 always 语句来开始编写状态机的代码，第一个 always 采用同步时序描述状态转移，第二个 always 采用组合逻辑判断状态转移条件，第三个 always 是描述状态输出，一个完整的三段式状态机的例子如下代码所示：

```
1  module divider7_fsm (
2      //系统时钟与复位
3      input       sys_clk      ,
4      input       sys_rst_n    ,
5
6      //输出时钟
7      output reg  clk_divide_7
8      );
9
```

```
10 //parameter define
11 parameter S0 = 7'b0000001;     //独热码定义方式
12 parameter S1 = 7'b0000010;
13 parameter S2 = 7'b0000100;
14 parameter S3 = 7'b0001000;
15 parameter S4 = 7'b0010000;
16 parameter S5 = 7'b0100000;
17 parameter S6 = 7'b1000000;
18
19 //reg define
20 reg  [6:0]   curr_st     ;     //当前状态
21 reg  [6:0]   next_st     ;     //下一个状态
22
23 //*****************************************************
24 //**                    main code
25 //*****************************************************
26
27 //状态机的第一段采用同步时序描述状态转移
28 always @(posedge sys_clk or negedge sys_rst_n) begin
29         if (!sys_rst_n)
30             curr_st <= S0;
31         else
32             curr_st <= next_st;
33 end
34
35 //状态机的第二段采用组合逻辑判断状态转移条件
36 always @(*) begin
37     case (curr_st)
38         S0: next_st = S1;
39         S1: next_st = S2;
40         S2: next_st = S3;
41         S3: next_st = S4;
42         S4: next_st = S5;
43         S5: next_st = S6;
44         S6: next_st = S0;
45         default: next_st = S0;
46     endcase
47 end
48
49 //状态机的第三段描述状态输出（这里采用时序电路输出）
50 always @(posedge sys_clk or negedge sys_rst_n) begin
```

```
51      if (!sys_rst_n)
52          clk_divide_7 <= 1'b0;
53      else if ((curr_st == S0) | (curr_st == S1) | (curr_st == S2)  | (curr_st == S3))
54          clk_divide_7  <= 1'b0;
55      else if ((curr_st == S4) | (curr_st == S5) | (curr_st == S6))
56          clk_divide_7  <= 1'b1;
57      else
58          ;
59  end
60
61  endmodule
```

在编写状态机代码时首先要定义状态变量（代码中的参数 S0~S6）与状态寄存器（curr_st、next_st），如代码中第 10 行至第 21 行所示；接下来使用三个 always 语句来实现三段状态机，第一个 always 语句实现同步状态跳转（如代码的第 27 至第 33 行所示），在复位的时候，当前状态处在 S0 状态，否则将下一个状态赋值给当前状态；第二个 always 采用组合逻辑判断状态转移条件（如代码的第 35 行至第 47 行代码所示），这里每一个状态只保持一个时钟周期，也就是直接跳转到下一个状态，在实际应用中，一般根据输入的条件来判断是否跳转到其它状态或者停留在当前转态，最后在 case 语句后面增加一个 default 语句，来防止状态机处在异常的状态；第三个 always 输出分频后的时钟（如代码的第 49 至第 59 行代码所示），状态机的第三段可以使用组合逻辑电路输出，也可以使用时序逻辑电路输出，一般推荐使用时序电路输出，因为状态机的设计和其它设计一样，最好使用同步时序方式设计，以提高设计的稳定性，消除毛刺。

从代码中可以看出，输出的分频时钟 clk_divide_7 只与当前状态（curr_st）有关，而与输入状态无关，所以属于摩尔型状态机。状态机的第一段对应摩尔状态机模型的状态寄存器，用来记忆状态机当前所处的状态；状态机的第二段对应摩尔状态机模型产生下一状态的组合逻辑 F；状态机的第三段对应摩尔状态机产生输出的组合逻辑 G，因为采用时序电路输出有很大的优势，所以这里第三段状态机是由时序电路输出的。

状态机采用时序逻辑输出的状态机模型如下图所示：

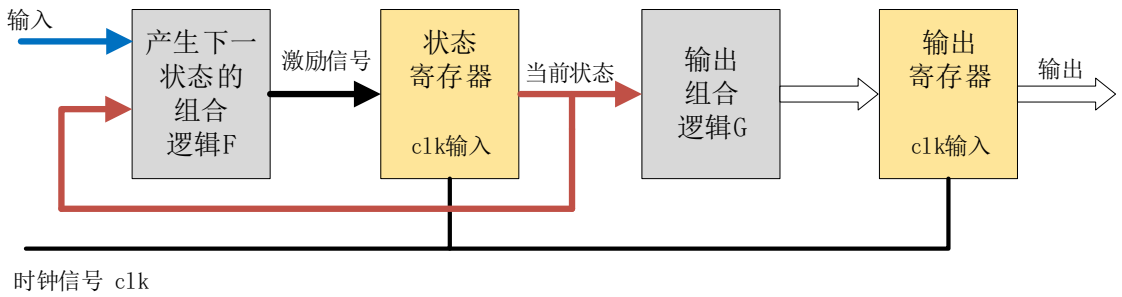


图 5.4.10 状态机时序电路输出模型

采用这种描述方法虽然代码结构复杂了一些，但是这样做的好处是可以有效地滤去组合逻辑输出的毛刺，同时也可以更好的进行时序计算与约束，另外对于总线形式的输出信号来说，容易使总线数据对齐，减小总线数据间的偏移，从而降低接收端数据采样出错的频率。

5.4.7 模块化设计

模块化设计是 FPGA 设计中一个很重要的技巧，它能够使一个大型设计的分工协作、仿真测试更加容

易，代码维护或升级更加便利，当更改某个子模块时，不会影响其它模块的实现结果。进行模块化、标准化设计的最终目的就是提高设计的通用性，减少不同项目中同一功能设计和验证引入的工作量。划分模块的基本原则是子模块功能相对独立、模块内部联系尽量紧密、模块间的连接尽量简单。

在进行模块化设计中，对于复杂的数字系统，我们一般采用自顶向下的设计方式。可以把系统划分成几个功能模块，每个功能模块再划分成下一层的子模块；每个模块的设计对应一个 module，一个 module 设计成一个 Verilog 程序文件。因此，对一个系统的顶层模块，我们采用结构化的设计，即顶层模块分别调用了各个功能模块。

下图是模块化设计的功能框图，一般整个设计的顶层模块只做例化（调用其它模块），不做逻辑。顶层下面会有模块 A、模块 B、模块 C 等，模块 A/B/C 又可以分多个子模块实现。

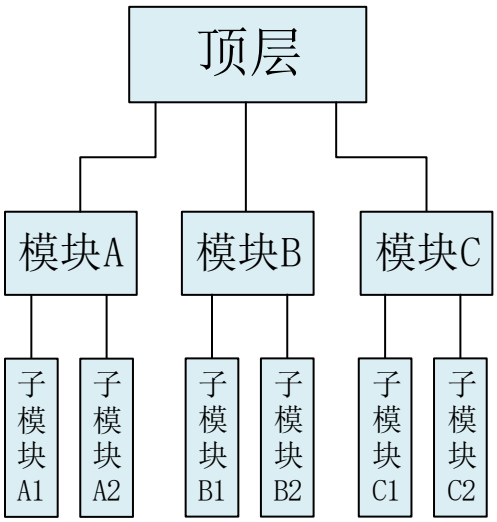


图 5.4.11 模块化设计框图

在这里我们补充一个概念，就是 Verilog 语法中的模块例化。FPGA 逻辑设计中通常是一个大的模块中包含了一个或多个功能子模块，Verilog 通过模块调用或称为模块实例化的方式来实现这些子模块与高层模块的连接，有利于简化每一个模块的代码，易于维护和修改。

下面以一个实例(静态数码管显示实验)来说明模块和模块之间的例化方法。

在静态数码管显示实验中，我们根据功能将 FPGA 顶层例化了以下两个模块：计时模块（time_count）和数码管静态显示模块（seg_led_static），如下图所示：

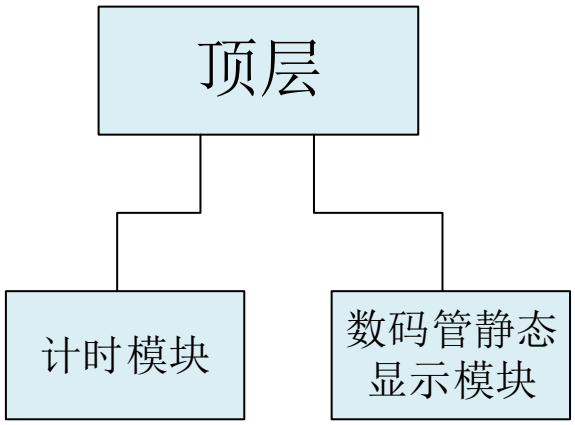


图 5.4.12 静态数码管显示模块框图

计时模块部分代码如下所示：

```
1  module time_count(
```

```
    input           clk     ,   // 时钟信号
    input           rst_n   ,   // 复位信号

    output  reg     flag        // 一个时钟周期的脉冲信号
);

//parameter define
parameter  MAX_NUM = 25000_000; // 计数器最大计数值
```

```
    ......
endmodule
```

数码管静态显示模块部分代码如下所示：

```
module seg_led_static (
    input               clk     ,   // 时钟信号
    input               rst_n   ,   // 复位信号（低有效）

    input               add_flag,   // 数码管变化的通知信号
    output  reg  [5:0]  sel     ,   // 数码管位选
    output  reg  [7:0]  seg_led     // 数码管段选
);
```

```
    ......

endmodule
```

顶层模块代码如下所示：

```
module seg_led_static_top (
    input               sys_clk  ,      // 系统时钟
    input               sys_rst_n,      // 系统复位信号（低有效）

    output    [5:0]     sel      ,      // 数码管位选
    output    [7:0]     seg_led          // 数码管段选

);

//parameter define
parameter  TIME_SHOW = 25'd25000_000;    // 数码管变化的时间间隔0.5s

//wire define
wire       add_flag;                     // 数码管变化的通知信号

//*************************************************
//**                    main code
//*************************************************
```

```
19
20 //例化计时模块
21 time_count #(
22     .MAX_NUM    (TIME_SHOW)
23 ) u_time_count(
24     .clk        (sys_clk  ),
25     .rst_n      (sys_rst_n),
26
27     .flag       (add_flag )
28 );
29
30 //例化数码管静态显示模块
31 seg_led_static u_seg_led_static (
32     .clk        (sys_clk  ),
33     .rst_n      (sys_rst_n),
34
35     .add_flag   (add_flag ),
36     .sel        (sel      ),
37     .seg_led    (seg_led  )
38 );
39
40 endmodule
```

我们上面贴出了顶层模块的完整代码，子模块只贴出了模块的端口和参数定义的代码。这是因为顶层模块对子模块做例化时，只需要知道子模块的端口信号名，而不用关心子模块内部具体是如何实现的。如果子模块内部使用 parameter 定义了一些参数，Verilog 也支持对参数的例化（也叫参数的传递），即顶层模块可以通过例化参数来修改子模块内定义的参数。

我们先来看一下顶层模块是如何例化子模块的，例化方法如下图所示：

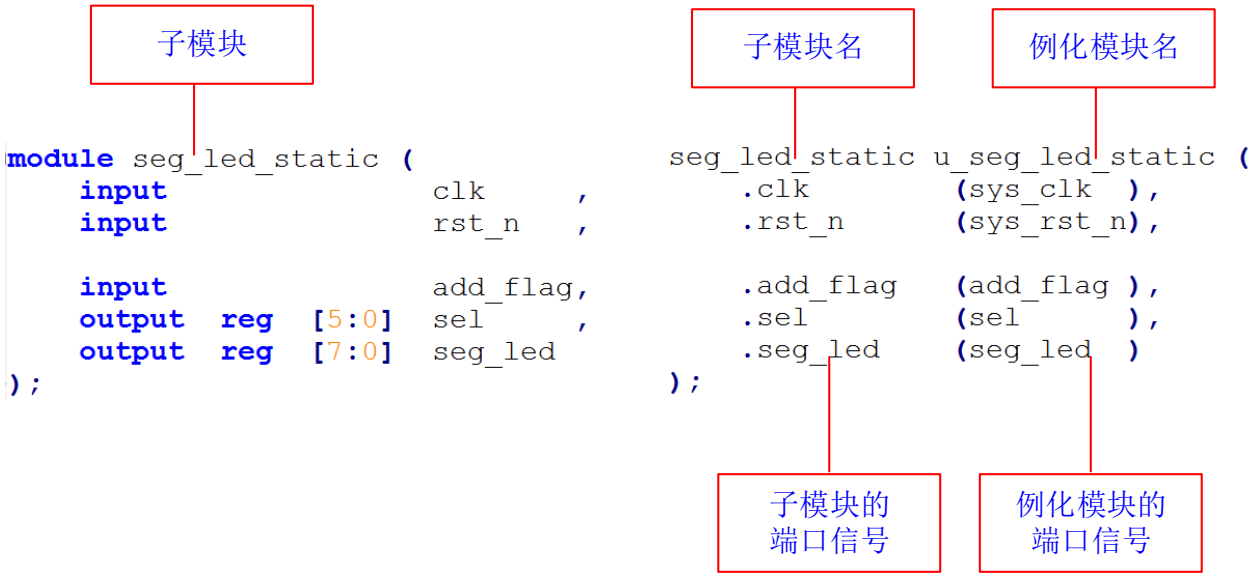


图 5.4.13 模块的例化

上图右侧是例化的数码管静态显示模块，子模块名是指被例化模块的模块名，而例化模块名相当于标识，当例化多个相同模块时，可以通过例化名来识别哪一个例化，我们一般命名为“u_”+“子模块名”。信号列表中“.”之后的信号是数码管静态显示模块定义的端口信号，括号内的信号则是顶层模块声明的信号，这样就将顶层模块的信号与子模块的信号一一对应起来，同时需要注意信号的位宽要保持一致。

接下来再来介绍一下参数的例化，参数的例化是在模块例化的基础上，增加了对参数的信号定义，如下图所示：

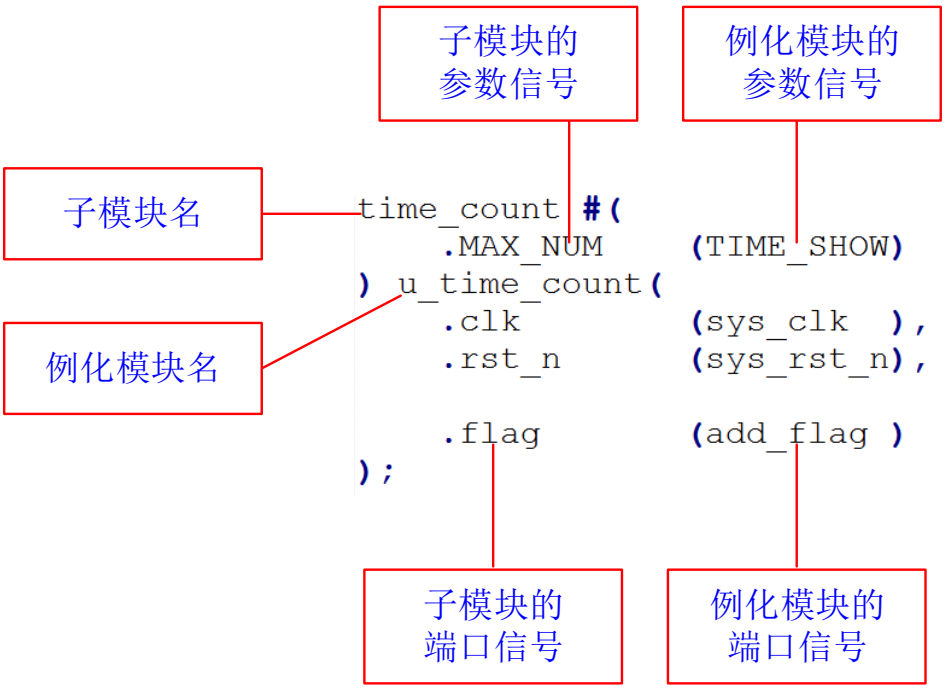


图 5.4.14 模块参数的例化

在对参数进行例化时，在模块名的后面加上“#”，表示后面跟着的是参数列表。计时模块定义的 MAX_NUM 和顶层模块的 TIME_SHOW 都是等于 25000_000，当在顶层模块定义 TIME_SHOW=12500_000 时，那么子模块的 MAX_NUM 的值实际上是也等于 12500_000。当然即使子模块包含参数，在做模块的例化时也可以不添加对参数的例化，这样的话，子模块的参数值等于该模块内部实际定义的值。

值得一提的是，Verilog 语法中的 localparam 代表的意思同样是参数定义，用法和 parameter 基本一致，区别在于 parameter 定义的参数可以做例化，而 localparam 定义的参数是指本地参数，上层模块不可以对 localparam 定义的参数做例化。

5.5 Verilog 编程规范

本节主要给大家介绍下编程规范，良好的编程规范是一个 FPGA 工程师必备的素质。

5.5.1 编程规范重要性

当前数字电路设计越来越复杂，一个项目需要的人越来越多，当几十号设计同事完成同一个项目时候，大家需要互相检视对方代码，如果没有一个统一的编程规范，那么是不可想象的。大家的风格都不一样，如果不统一的话，后续维护、重用等会有很大的困难，即使是自己写的代码，几个月后再看也会变的很陌生，也会看不懂（您可能不相信，不过笔者和同事交流发现大家都是这样的，时间长不看就忘记了），所以编程规范的重要性显而易见。

另外养成良好的编程规范，对于个人的工作习惯、思路等都有非常大的好处。可以让新人尽快融入项目中，让大家更容易看懂您的代码。

5.5.2 工程组织形式

工程的组织形式一般包括如下几个部分，分别是 doc、par、rtl 和 sim 四个部分。

```
XX 工程名
|--doc
|--par
|--rtl
|--sim
```

doc：一般存放工程相关的文档，包括该项目用到的 datasheet（数据手册）、设计方案等。不过为了便于大家查看，我们开发板文档是统一汇总存放在资料盘下的；

par：主要存放工程文件和使用到的一些 IP 文件；

rtl：主要存放工程的 rtl 代码，这是工程的核心，文件名与 module 名称应当一致，建议按照模块的层次分开存放；

sim：主要存放工程的仿真代码，复杂的工程里面，仿真也是不可或缺的部分，可以极大减少调试的工作量。

5.5.3 文件头声明

每一个 Verilog 文件的开头，都必须有一段声明的文字。包括文件的版权，作者，创建日期以及内容介绍等，如下表所示。

```
//****************************************Copyright (c)***********************************//
//原子哥在线教学平台：www.yuanzige.com
//技术支持：www.openedv.com
//淘宝店铺：http://openedv.taobao.com
//关注微信公众平台微信号："正点原子"，免费获取 ZYNQ & FPGA & STM32 & LINUX 资料。
//版权所有，盗版必究。
//Copyright(C) 正点原子 2018-2028
//All rights reserved
//----------------------------------------------------------------------------------------
// File name:           led_twinkle
// Last modified Date:  2019/4/14 10:55:56
// Last Version:        V1.0
// Descriptions:        LED 灯闪烁
//----------------------------------------------------------------------------------------
// Created by:          正点原子
// Created date:        2019/4/14 10:55:56
// Version:             V1.0
// Descriptions:        The original version
//
//----------------------------------------------------------------------------------------
```

```
//****************************************************************************************//
```

我们建议一个.V 只包括一个 module，这样模块会比较清晰易懂。

5.5.4 输入输出定义

端口的输入输出有 Verilog 95 和 2001 两种格式，推荐大家采用 Verilog 2001 语法格式。下面是 Verilog 2001 语法的一个例子，包括 module 名字、输入输出、信号名字、输出类型、注释。

```
1  module led(
2      input              sys_clk  ,  //系统时钟
3      input              sys_rst_n,  //系统复位，低电平有效
4      output  reg  [3:0] led         //4 位 LED 灯
5      );
```

我们建议如下几点：

① 一行只定义一个信号；

② 信号全部对齐；

③ 同一组的信号放在一起。

5.5.5 parameter 定义

我们建议如下几点：

① module 中的 parameter 声明，不建议随处乱放；

② 将 parameter 定义放在紧跟着 module 的输入输出定义之后；

③ parameter 等常量命名全部使用大写。

```
7  //parameter define
8  parameter  WIDTH     = 25         ;
9  parameter  COUNT_MAX = 25_000_000; //板载50M时钟=20ns，0.5s/20ns=25000000，需要25bit
10                                    //位宽
```

5.5.6 wire/reg 定义

一个 module 中的 wire/reg 变量声明需要集中放在一起，不建议随处乱放。
因此，我们建议如下：

① 将 reg 与 wire 的定义放在紧跟着 parameter 之后；

② 建议具有相同功能的信号集中放在一起；

③ 信号需要对齐，reg 和位宽需要空 2 格，位宽和信号名字至少空四格；

④ 位宽使用降序描述，[6:0]；

⑤ 时钟使用前缀 clk，复位使用后缀 rst；

⑥ 不能使用 Verilog 关键字作为信号名字；

⑦ 一行只定义一个信号。

```
12  //reg define
13  reg    [WIDTH-1:0]  counter     ;
14  reg    [1:0]        led_ctrl_cnt;
15
16  //wire define
17  wire                counter_en  ;
```

5.5.7 信号命名

大家对信号命名可能都有不同的喜好，我们建议如下：

① 信号命名需要体现其意义，比如 fifo_wr 代表 FIFO 读写使能；

② 可以使用“_”隔开信号，比如 sys_clk；

③ 内部信号不要使用大写，也不要使用大小写混合，建议全部使用小写；

④ 模块名字使用小写；

⑤ 低电平有效的信号，使用_n 作为信号后缀；

⑥ 异步信号，使用_a 作为信号后缀；

⑦ 纯延迟打拍信号使用_dly 作为后缀。

5.5.8 always 块描述方式

always 块的编程规范，我们建议如下：

① if 需要空四格；

② 一个 always 需要配一个 begin 和 end；

③ always 前面需要有注释；

④ beign 建议和 always 放在同一行；

⑤ 一个 always 和下一个 always 空一行即可，不要空多行；

⑥ 时钟复位触发描述使用 posedge sys_clk 和 negedge sys_rst_n

⑦ 一个 always 块只包含一个时钟和复位；

⑧ 时序逻辑使用非阻塞赋值。

```
26  //用于产生0.5秒使能信号的计数器
27  always @(posedge sys_clk or negedge sys_rst_n) begin
28      if (sys_rst_n == 1'b0)
```

```
29          counter <= 1'b0;
30      else if (counter_en)
31          counter <= 1'b0;
32      else
33          counter <= counter + 1'b1;
34  end
```

5.5.9 assign 块描述方式

assign 块的编程规范，我们建议如下：

① assign 的逻辑不能太复杂，否则易读性不好；

② assign 前面需要有注释；

③ 组合逻辑使用阻塞赋值。

```
23  //计数到最大值时产生高电平使能信号
24  assign  counter_en = (counter == (COUNT_MAX - 1'b1))  ?  1'b1  :  1'b0;
```

5.5.10 空格和 TAB

由于不同的解释器对于 TAB 翻译不一致，所以建议不使用 TAB，全部使用空格。

5.5.11 注释

添加注释可以增加代码的可读性，易于维护。我们建议规范如下：

① 注释描述需要清晰、简洁；

② 注释描述不要废话，冗余；

③ 注释描述需要使用“//”；

④ 注释描述需要对齐；

⑤ 核心代码和信号定义之间需要增加注释。

```
26  //用于产生 0.5 秒使能信号的计数器
27  always @(posedge sys_clk or negedge sys_rst_n) begin
28      if (sys_rst_n == 1'b0)
29          counter <= 1'b0;
30      else if (counter_en)                        // counter_en 为 1 时，counter 清 0
31          counter <= 1'b0;
32      else
33          counter <= counter + 1'b1;
34  end
```

5.5.12 模块例化

模块例化我们建议规范如下：

① moudle 模块例化使用 u_xx 表示。

```
20 //例化计时模块
21 time_count #(
22     .MAX_NUM    (TIME_SHOW)
23 ) u_time_count(
24     .clk        (sys_clk  ),
25     .rst_n      (sys_rst_n),
26
27     .flag       (add_flag )
28 );
29
30 //例化数码管静态显示模块
31 seg_led_static u_seg_led_static (
32     .clk        (sys_clk  ),
33     .rst_n      (sys_rst_n),
34
35     .add_flag   (add_flag ),
36     .sel        (sel      ),
37     .seg_led    (seg_led  )
38 );
```

5.5.13 其他注意事项

其他注意事项如下：

① 代码写的越简单越好，方便他人阅读和理解；

② 不使用 repeat 等循环语句；

③ RTL 级别代码里面不使用 initial 语句，仿真代码除外；

④ 避免产生 Latch 锁存器，比如组合逻辑里面的 if 不带 else 分支、case 缺少 default 语句；

⑤ 避免使用太复杂和少见的语法，可能造成语法综合器优化力度较低。

良好的编程规范是大家走向专业 FPGA 工程师的必备素质，希望大家都能养成良好的编程规范。

第四篇　实战篇

经过前三篇的学习，我们对 ZYNQ 开发的硬件平台、软件以及 Verilog 语法都有了比较深入的了解，接下来我们将通过实例，由浅入深的带大家一步步的学习 FPGA。

我们启明星 ZYNQ 开发板的外设非常丰富，本篇将从开发板上最简单的外设说起，然后一步步深入。每一个实例都配有详细的代码及解释，手把手教你如何开始 FPGA 的设计开发，通过本篇的学习，希望大家能掌握 FPGA 的设计开发与熟悉开发板上的外设，并且举一反三，应用到其它工程项目中。

本篇总共分为 18 章，每一章即一个实例，下面就让我们开始精彩的 FPGA 之旅。

第六章　LED 灯闪烁实验

LED 灯闪烁作为一个经典的入门实验，其地位堪比编程界的“Hello，World！”。对于很多电子工程师来说，LED 灯闪烁都是他们在硬件上观察到的第一个实验现象。本章我们同样通过 LED 灯闪烁实验，带你进入 ZYNQ 的精彩世界。

本章包括以下几个部分：

6.1 LED 灯简介

LED，又名发光二极管。LED 灯工作电流很小（有的仅零点几毫安即可发光），抗冲击和抗震性能好，可靠性高，寿命长。由于这些优点，LED 灯被广泛用在仪器仪表中作指示灯、液晶屏背光源等诸多领域。

不同材料的发光二极管可以发出红、橙、黄、绿、青、蓝、紫、白这八种颜色的光。图 6.1.1 是可以发出黄、红、蓝三种颜色的直插型二极管实物图，这种二极管长的一端是阳极，短的一端是阴极。图 6.1.2 是开发板上用的贴片二极管实物图。贴片二极管的正面一般都有颜色标记，有标记的那端就是阴极。

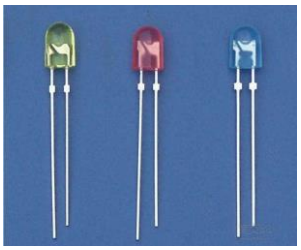


图 6.1.1 发光二极管实物图

图 6.1.2 贴片发光二极管实物图

发光二极管与普通二极管一样具有单向导电性。给它加上阳极正向电压后，通过 5mA 左右的电流就可以使二极管发光。通过二极管的电流越大，发出的光亮度越强。不过我们一般将电流限定在 3~20mA 之间，否则电流过大就会烧坏二极管。

6.2 实验任务

本节实验任务是使启明星底板上的 PL LED0 和 PL LED1 以固定的频率交替闪烁。

6.3 硬件设计

底板上 LED 的原理图如下图所示：

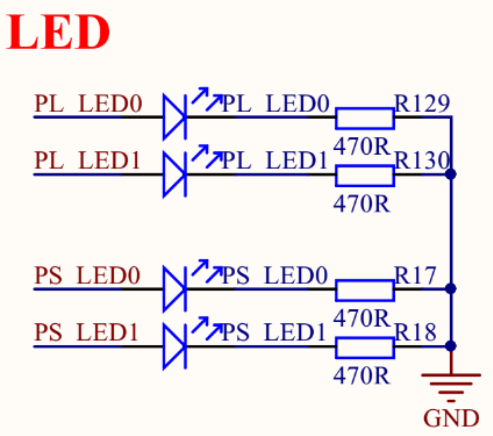


图 6.3.1 LED 灯硬件原理图

在图 6.3.1 中，PL_LED0 和 PL_LED1 连接到 ZYNQ 的 PL 端，PS_LED0 和 PS_LED1 连接到 ZYNQ 的 PS 端。在《启明星 ZYNQ 之 FPGA 开发指南》中，我们只使用 PL 端的外设。

PL_LED0 和 PL_LED1 的阴极通过 470 欧姆的电阻连到地（GND）上，阳极由 ZYNQ PL 的 IO 管脚控制，LED 与地之间的电阻起到限流作用。

本实验中，系统时钟、按键复位以及两个 LED 端口的管脚分配如下表所示，其中复位按键和两个 LED 位于底板上，时钟位于核心板上：

表 6.3.1 LED闪烁实验管脚分配

信号名	方向	管脚	端口说明	电平标准
sys_clk	input	U18	系统时钟，50MHz	LVCMOS33
sys_rst_n	input	J15	系统复位按键，低电平有效	LVCMOS33
led[0]	output	J18	PL_LED0（底板）	LVCMOS33
led[1]	output	H18	PL_LED1（底板）	LVCMOS33

对应的 XDC 约束语句如下所示：

```
set_property -dict {PACKAGE_PIN U18 IOSTANDARD LVCMOS33} [get_ports sys_clk]
set_property -dict {PACKAGE_PIN J15 IOSTANDARD LVCMOS33} [get_ports sys_rst_n]
set_property -dict {PACKAGE_PIN J18 IOSTANDARD LVCMOS33} [get_ports {led[0]}]
set_property -dict {PACKAGE_PIN H18 IOSTANDARD LVCMOS33} [get_ports {led[1]}]
```

6.4 程序设计

由于发光二极管的阳极与 ZYNQ 的管脚相连，只需要改变与 LED 灯相连的 ZYNQ 管脚的电平，LED 灯的亮灭状态就会发生变化。当 ZYNQ 管脚为高电平时，LED 灯点亮；为低电平时，LED 灯熄灭。

本次设计的模块端口及信号连接如下图所示：

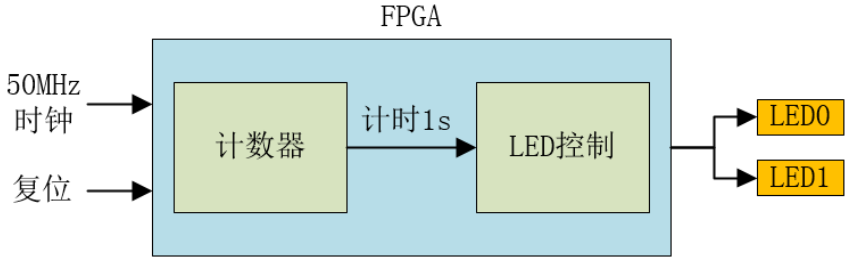


图 6.4.1 LED 灯模块原理图

其中，计数器对 50MHz 时钟进行计数，从而达到计时的目的。计数器在每次计时到 1 秒之后清零，然后重新开始计数，计数的值用于控制 LED 的显示状态。当计数器的值小于 0.5s 时，就把 LED0 点亮并把 LED1 熄灭；每当计数器的值大于 0.5s 时，就把 LED0 熄灭并把 LED1 点亮，以此实现两个 LED 的交替闪烁。

LED 闪烁模块的代码如下：

```
1  module  led_twinkle(
2      input          sys_clk  ,  //系统时钟
3      input          sys_rst_n,  //系统复位，低电平有效
4
5      output  [1:0]  led         //LED 灯
```

```
6   );
7
8   //reg define
9   reg  [25:0]  cnt ;
10
11  //*****************************************************
12  //**                    main code
13  //*****************************************************
14
15  //对计数器的值进行判断，以输出 LED 的状态
16  assign led = (cnt < 26'd2500_0000) ? 2'b01 : 2'b10 ;
17
18  //计数器在 0~5000_000 之间进行计数
19  always @ (posedge sys_clk or negedge sys_rst_n) begin
20      if(!sys_rst_n)
21          cnt <= 26'd0;
22      else if(cnt < 26'd5000_0000)
23          cnt <= cnt + 1'b1;
24      else
25          cnt <= 26'd0;
26  end
27
28  endmodule
```

本程序中输入时钟为 50MHz，所以一个时钟周期为 20ns（1/50MHz）。因此计数器 cnt 通过对 50MHz 系统时钟计数，计时到 1s，需要累加 1s/20ns=5000_0000 次。在代码第 23 行，每当计时到 1s 计数器清零一次。

同时，在代码的第 16 行，对根据计数器的计数值来赋值两个 LED 的状态。当计数值小于 26'd2500_000 即计时到 1s 中的前 500ms 时，LED0 点亮 LED1 熄灭；当计数值大于等于 26'd2500_000，即计时到 1s 中的后 500ms 时，LED0 熄灭 LED1 点亮。当计数到 1s 时，计数器又会回 0，重复此过程。以此实现两个 LED 的交替闪烁。

为了验证我们的程序，我们在 Vivado 内对代码进行仿真。为了更容易地看到仿真现象，我们将源代码中的计数器的最大计数值修改为 5，然后再仿真，如下图所示：

```
35
36  //对计数器的值进行判断，以输出LED的状态
37  //assign led = (cnt < 26'd2500_0000) ? 2'b01 : 2'b10 ;
38  assign led = (cnt < 26'd5)          ? 2'b01 : 2'b10 ;  //仅用于仿真
39
40  //计数器在0~5000_000之间进行计数
41  always @ (posedge sys_clk or negedge sys_rst_n) begin
42      if(!sys_rst_n)
43          cnt <= 26'd0;
44  //    else if(cnt < 26'd5000_0000)
45    else if(cnt < 26'd10)  //仅用于仿真
46          cnt <= cnt + 1'b1;
47      else
48          cnt <= 26'd0;
49  end
```

图 6.4.2 仅用于仿真的代码

Testbench 模块代码如下：

```
1   `timescale 1ns / 1ps
2
3   module tb_led_twinkle();
4
5   //输入
6   reg           sys_clk;
7   reg           sys_rst_n;
8
9   //输出
10  wire  [1:0]   led;
11
12  //信号初始化
13  initial begin
14      sys_clk = 1'b0;
15      sys_rst_n = 1'b0;
16      #200
17      sys_rst_n = 1'b1;
18  end
19
20  //生成时钟
21  always #10 sys_clk = ~sys_clk;
22
23  //例化待测设计
24  led_twinkle  u_led_twinkle(
25      .sys_clk         (sys_clk),
26      .sys_rst_n       (sys_rst_n),
```

```
27      .led            (led)
28      );
29
30  endmodule
```

仿真得到的波形图如下图所示：

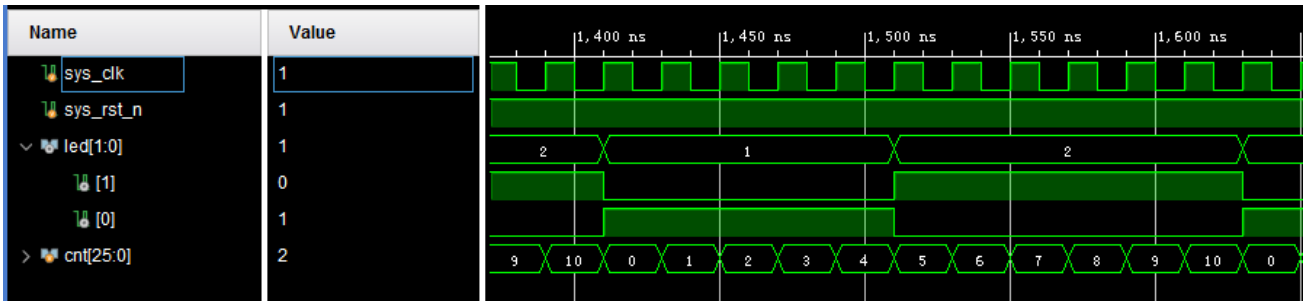


图 6.4.3 仿真波形图

从图 6.4.3 中可以看到，计数器 cnt 的值在 0 到 10 之间循环计数。当 cnt 的值小于 5 时，led0 为高电平；大于 5 时 led0 为高电平。两个 LED 的状态随着计数器的计数循环翻转，实现 LED 闪烁的效果。

6.5 下载验证

编译工程并生成比特流.bit 文件后，点击 Vivado 左侧“Flow Navigator”窗口最下面的“Open Hardware Manager”按钮如下图所示。

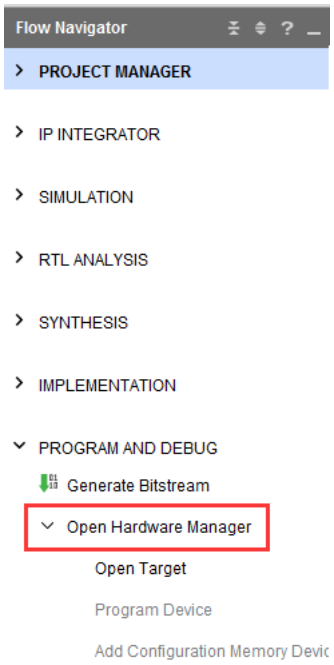


图 6.5.1 Hardware Manager 界面

此时将 Xilinx 下载器一端连接电脑，另一端与开发板上的 JTAG 下载口连接，开发板连接电源线，如下图所示：

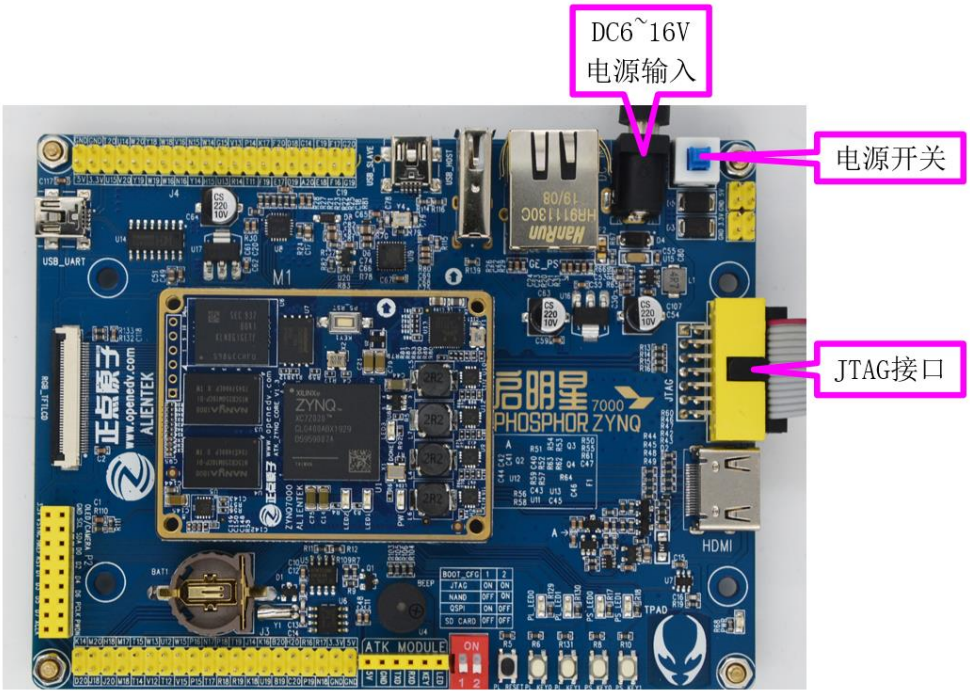


图 6.5.2 启明星开发板连接实物图

注意！一定要先把下载器的一端连接到了电脑、另一端连接了 JTAG 接口之后，再给开发板上电！否则，对开发板的 JTAG 接口进行带电热插拔，有一定概率会损坏 JTAG 接口！

开发板连接好电源线和下载器后，打开开发板电源开关，点击“Hardware”窗口中的“Auto Connect”图标，如下图所示：

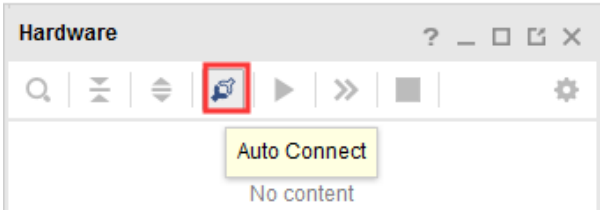


图 6.5.3 “Auto Connect”图标

在“Hardware”子窗口中出现如下界面就表示 Vivado 就已经和下载器连接成功了，如下图所示：

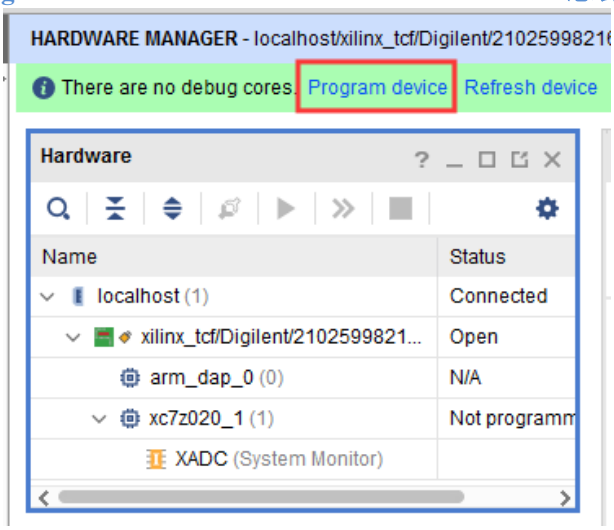


图 6.5.4 与下载器连接成功

我们点击上图中的“Program Device”，弹出的界面如下图所示：

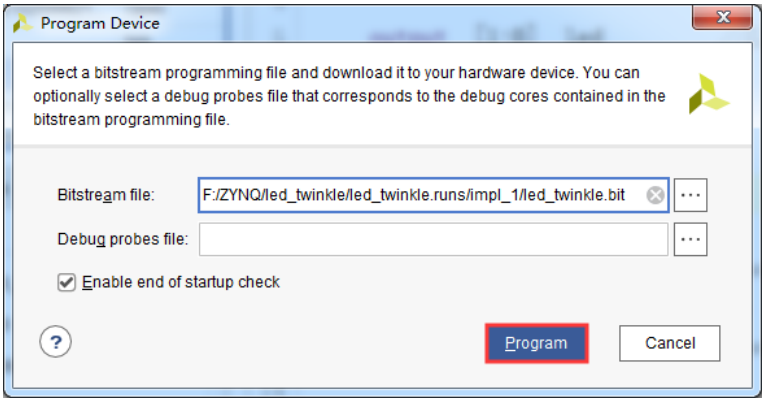


图 6.5.5 下载比特流界面

此时 Bitstream File 一栏会自动识别到工程的比特流文件，我们直接点击“Program”按钮下载程序，程序下载完成后，PL 配置完成灯会点亮（LED3），此时我们可以看到位于底板上的两个 LED 灯在不断地闪烁，如下图所示：

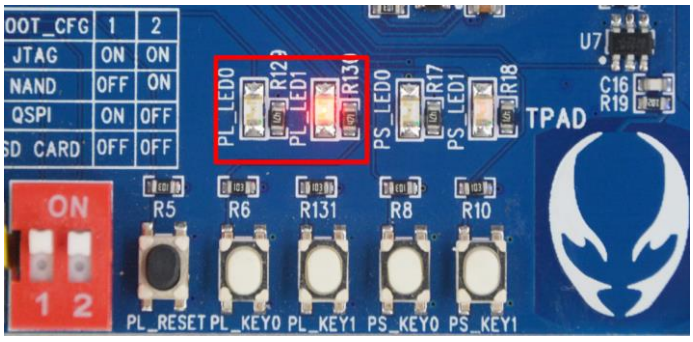


图 6.5.6 两个 PL LED 灯交替闪烁

第七章　按键控制 LED 闪烁实验

按键是常用的一种控制器件。生活中我们可以见到各种形式的按键，由于其结构简单，成本低廉等特点，在家电、数码产品、玩具等方面有广泛的应用。本章我们将介绍如何使用按键来控制 LED 闪烁的方式。

本章包括以下几个部分：

7.1 按键简介

按键开关是一种电子开关，属于电子元器件类。我们的开发板上有两种按键开关：第一种是本实验所使用的轻触式按键开关（如图 7.1.1），简称轻触开关。使用时以向开关的操作方向施加压力使内部电路闭合接通，当撤销压力时开关断开，其内部结构是靠金属弹片受力后发生形变来实现通断的；第二种是自锁按键（如图 7.1.2），自锁按键第一次按下后保持接通，即自锁，第二次按下后，开关断开，同时开关按钮弹出来，开发板上的电源键就是这种开关。

图 7.1.1 轻触式按键

图 7.1.2 自锁式按键

7.2 实验任务

本节实验任务是使用底板上的 PL_KEY0 和 PL_KEY1 按键来控制底板上的 PL_LED0 和 PL_LED1 两个 LED 的闪烁方式。没有按键按下时，两个 LED 保持常亮；如果按键 0 按下，则两个 LED 交替闪烁；如果按键 1 按下，则两个 LED 同时闪烁。

7.3 硬件设计

底板上按键的原理图如下图所示：

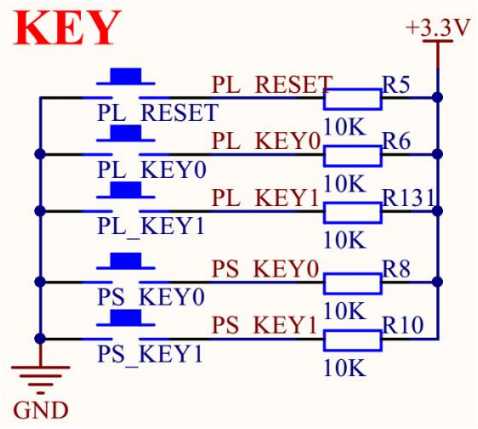


图 7.3.1 按键电路原理图

在图 7.3.1 中，PL_KEY0 和 PL_KEY1 连接到 ZYNQ 的 PL 端，PS_KEY0 和 PS_KEY1 连接到 ZYNQ 的 PS 端。在《启明星 ZYNQ 之 FPGA 开发指南》中，我们只使用 PL 端的外设。

PL 端的按键没有按下时，对应的 IO 端口为高电平；当按键按下时，对应的 IO 端口变为低电平。

本实验的管脚分配如下表所示：

表 7.3.1 按键控制LED实验管脚分配

信号名	方向	管脚	端口说明	电平标准
sys_clk	input	U18	系统时钟，50M	LVCMOS33
sys_rst_n	input	J15	PL 复位按键，低有效	LVCMOS33
key[0]	input	L20	PL 按键 0	LVCMOS33
key[1]	input	J20	PL 按键 1	LVCMOS33
led[0]	output	J18	PL LED 0	LVCMOS33
led[1]	output	H18	PL LED 0	LVCMOS33

对应的 XDC 约束语句如下所示：

```
set_property -dict {PACKAGE_PIN U18 IOSTANDARD LVCMOS33} [get_ports sys_clk]
set_property -dict {PACKAGE_PIN J15 IOSTANDARD LVCMOS33} [get_ports sys_rst_n]
set_property -dict {PACKAGE_PIN L20 IOSTANDARD LVCMOS33} [get_ports key[0]]
set_property -dict {PACKAGE_PIN J20 IOSTANDARD LVCMOS33} [get_ports key[1]]
set_property -dict {PACKAGE_PIN J18 IOSTANDARD LVCMOS33} [get_ports {led[0]}]
set_property -dict {PACKAGE_PIN H18 IOSTANDARD LVCMOS33} [get_ports {led[1]}]
```

7.4 程序设计

按键控制 LED 系统框图如下图所示：

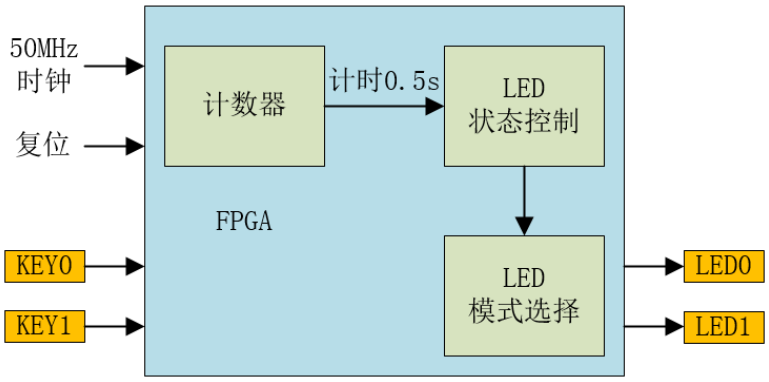


图 7.4.1 按键控制 LED 系统框图

在图 7.4.1 中，计数器对 50MHz 时钟进行计数，从而达到计时的目的。计数器在每次计时到 0.5 秒的时候，就改变 LED 的显示状态，然后清零并重新开始计数。

然后根据两个按键（KEY0 和 KEY1）的状态，在不同的 LED 状态下，分别设置 LED 的显示模式（是同时闪烁，或者交替闪烁）。

顶层模块代码如下：

```
module key_led(
    input               sys_clk ,
    input               sys_rst_n ,

    input       [1:0]   key ,
    output  reg [1:0]   led
);

//reg define
reg [24:0] cnt;
reg        led_ctrl;

//*****************************************************
//**                    main code
//*****************************************************

//计数器
always @ (posedge sys_clk or negedge sys_rst_n) begin
    if(!sys_rst_n)
        cnt <= 25'd0;
    else if(cnt < 25'd2500_0000)  //计数 500ms
        cnt <= cnt + 1'b1;
    else
        cnt <= 25'd0;
end

//每隔 500ms 就更改 LED 的闪烁状态
always @ (posedge sys_clk or negedge sys_rst_n) begin
    if(!sys_rst_n)
        led_ctrl <= 1'b0;
    else if(cnt == 25'd2500_0000)
        led_ctrl <= ~led_ctrl;
end

//根据按键的状态以及 LED 的闪烁状态来赋值 LED
always @ (posedge sys_clk or negedge sys_rst_n) begin
    if(!sys_rst_n)
        led <= 2'b11;
    else case(key)
```

```
40          2'b10 :  //如果按键 0 按下，则两个 LED 交替闪烁
41              if(led_ctrl == 1'b0)
42                  led <= 2'b01;
43              else
44                  led <= 2'b10;
45          2'b01 :  //如果按键 1 按下，则两个 LED 同时闪烁
46              if(led_ctrl == 1'b0)
47                  led <= 2'b11;
48              else
49                  led <= 2'b00;
50          2'b11 :  //如果两个按键都未按下，则两个 LED 都保持点亮
51                  led <= 2'b11;
52          default: ;
53      endcase
54  end
55
56  endmodule
```

代码的第 18 行的 always 块用于产生计数器，计时 500ms。代码的第 28 行的 always 块功能是每隔 500ms 就给出 led 的闪烁状态控制信号。代码第 36 行的 always 块使用了一个 case 语句，来根据当前按键的输入值和 led 闪烁状态控制信号，来进行两个 led 的赋值。如果按键 1 按下，则两个 LED 交替闪烁；如果按键 0 按下，则两个 LED 同时亮灭交替；如果两个按键都未按下，则两个 LED 都保持点亮。

7.5 下载验证

连接开发板的电源和下载器，并打开电源开关。在工程编译之后，将生成的 bit 文件下载到开发板中。下载完成之后，底板上两个 PL LED 处于点亮状态。然后按下 PL_KEY0，可以看到两个 PL LED 交替闪烁；按下 PL_KEY1，可以看到两个 PL 的 LED 同时闪烁。如下图所示：

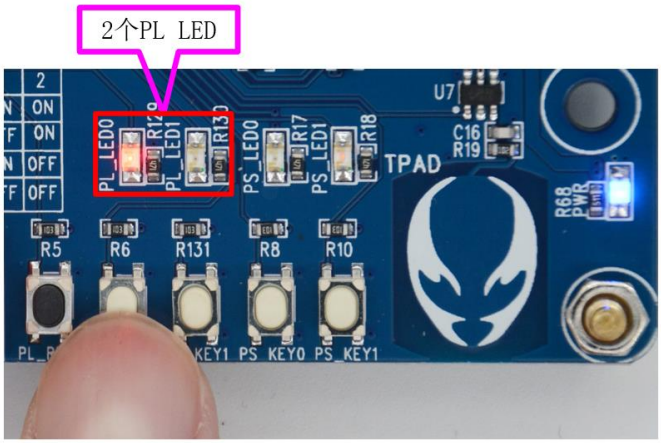


图 7.5.1 实验现象

第八章　按键控制蜂鸣器实验

蜂鸣器（Buzzer）是现代常用的一种电子发声器，主要用于产生声音信号。蜂鸣器在生活中已经得到广泛使用，其典型应用包括医疗，消防等领域的各种报警装置以及日常生活中的各种警报器等。本章我们主要学习如何使用按键来控制蜂鸣器发声。

本章包括以下几个部分：

8.1　蜂鸣器简介

8.2　实验任务

8.3　硬件设计

8.4　程序设计

8.5　下载验证

8.1 蜂鸣器简介

蜂鸣器按照驱动方式主要分为有源蜂鸣器和无源蜂鸣器，其主要区别为蜂鸣器内部是否含有震荡源。一般的有源蜂鸣器内部自带了震荡源，只要通电就会发声。而无源蜂鸣器由于不含内部震荡源，需要外接震荡信号才能发声。

图 8.1.1 有源蜂鸣器（左）和无源蜂鸣器（右）

如上图所示，从外观上看，两种蜂鸣器很相似，如将两种蜂鸣器的引脚都朝上放置，能看到绿色电路板的是无源蜂鸣器，没有电路板而用黑胶封闭的一种是有源蜂鸣器。

相较于有源蜂鸣器，无源蜂鸣器成本更低，且发声频率可控。而有源蜂鸣器控制相对简单，由于内部自带震荡源，只要加上合适的直流电压即可发声。本次实验使用的蜂鸣器为有源蜂鸣器。

8.2 实验任务

本节实验任务是使用启明星上的 PL KEY0 按键来控制蜂鸣器发声。初始状态为蜂鸣器鸣叫，按下按键后蜂鸣器停止鸣叫，再次按下开关，蜂鸣器重新鸣叫。

8.3 硬件设计

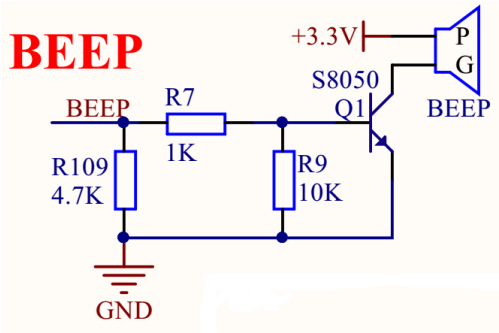


图 8.3.1 蜂鸣器控制电路原理图

上图为蜂鸣器控制电路的原理图。由于 ZYNQ 的 IO 其电流驱动能力有限，所以我们在蜂鸣器的驱动电路中加入三级管 Q1，以将 ZYNQ 的 IO 驱动电流放大，然后再驱动蜂鸣器。

本实验的管脚分配如下表所示,其中时钟源位于核心板上，复位按键、用户按键、蜂鸣器都位于底板上：

表 8.3.1 按键控制蜂鸣器实验管脚分配

信号名	方向	管脚	端口说明	电平标准
sys_clk	input	U18	系统时钟，50M	LVCMOS33
sys_rst_n	input	J15	PL复位按键，低有效	LVCMOS33

key	input	L20	PL KEY0按键	LVCMOS33
beep	output	G18	蜂鸣器	LVCMOS33

对应的 XDC 约束语句如下所示：

```
set_property -dict {PACKAGE_PIN G18 IOSTANDARD LVCMOS33} [get_ports {beep}]
set_property -dict {PACKAGE_PIN L20 IOSTANDARD LVCMOS33} [get_ports {key}]
set_property -dict {PACKAGE_PIN U18 IOSTANDARD LVCMOS33} [get_ports sys_clk]
set_property -dict {PACKAGE_PIN J15 IOSTANDARD LVCMOS33} [get_ports sys_rst_n]
```

8.4 程序设计

由实验任务可知，我们只需要在按键按下时改变蜂鸣器的鸣叫状态即可。但实际上在按键按下的过程中存在按键抖动的干扰，体现在数字电路中就是不断变化的高低电平。为避免在抖动过程中采集到错误的按键状态，我们需要对按键数据进行消除抖动处理。

在这里我们先介绍一下按键消抖的原理。通常我们所使用的开关为机械弹性开关，当我们按下或松开按键时，由于弹片的物理特性，不能立即闭合或断开，往往会在断开或闭合的短时间内产生机械抖动，。消除这种抖动的过程即称为按键消抖。

按键消抖可分为硬件消抖和软件消抖。硬件消抖主要使用 RS 触发器或电容等方法在硬件电路上实现消抖，一般在按键较少时使用。软件消抖的原理主要为按键按下或松开后，由处理器延时 5ms 至 20ms，然后再对按键状态进行采样并判断。如下图所示：

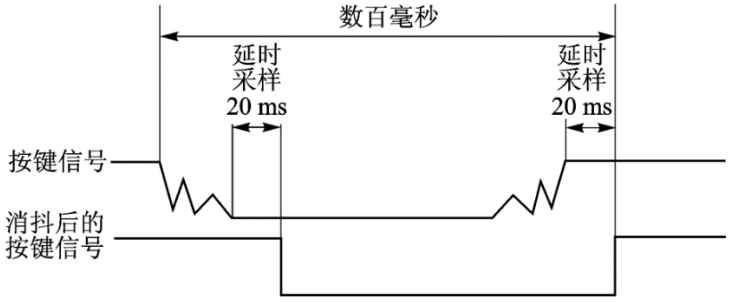


图 8.4.1 按键消抖原理图

由上面的分析可知，本次实验中的系统至少包含按键消抖模块和蜂鸣器控制两个模块。系统框图如下图所示：

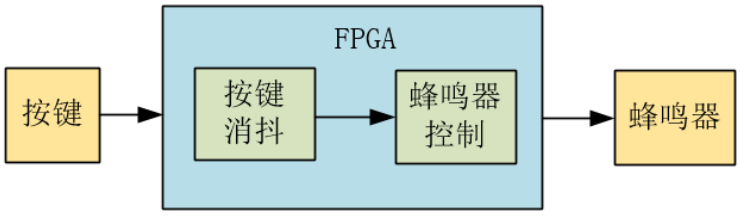


图 8.4.2 按键控制蜂鸣器系统框图

图 8.4.2 中，按键消抖模块用于消除按键的抖动，消抖之后的信号用于控制蜂鸣器的鸣叫状态。

程序中各模块端口及信号连接如下图所示：

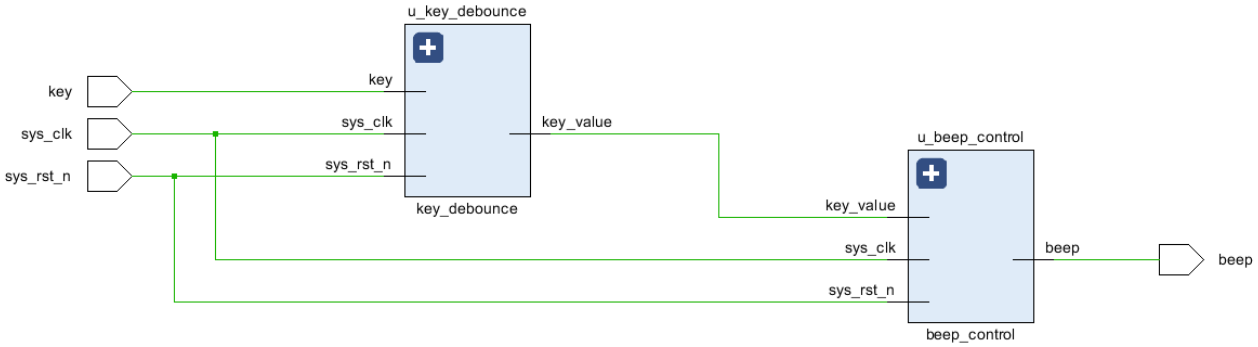


图 8.4.3 端口及信号连接图

由上图系统框图可知，代码部分包括三个模块。顶层模块（top_key_beep），作用为完成对另外两个模块的例化。按键消抖模块（key_debounce），主要用于对按键进行抖动滤除。按键控制蜂鸣器模块（beep_control），识别按键按下的那一刻，并对蜂鸣器的鸣叫状态进行翻转。

顶层模块代码如下：

```
1   module top_key_beep(
2       input    sys_clk ,
3       input    sys_rst_n ,
4
5       input    key ,
6       output   beep
7   );
8
9   //wire define
10  wire key_value ;
11  wire key_flag ;
12
13  //*****************************************************
14  //**                    main code
15  //*****************************************************
16
17  //例化按键消抖模块
18  key_debounce  u_key_debounce(
19      .sys_clk    (sys_clk),
20      .sys_rst_n  (sys_rst_n),
21
22      .key        (key),
23      .key_value  (key_value),
24      .key_flag   (key_flag)
25      );
26
```

```
27  //例化蜂鸣器控制模块
28  beep_control  u_beep_control(
29      .sys_clk    (sys_clk),
30      .sys_rst_n  (sys_rst_n),
31
32      .key_value  (key_value),
33      .key_flag   (key_flag),
34      .beep       (beep)
35      );
36
37  endmodule
```

在顶层模块中例化了按键消抖模块和按键控制蜂鸣器模块。

按键消抖模块代码如下：

```
1   module key_debounce(
2       input        sys_clk ,
3       input        sys_rst_n ,
4
5       input        key ,          //外部输入的按键值
6       output  reg  key_value ,    //消抖后的按键值
7       output  reg  key_flag       //消抖后的按键值的效标志
8   );
9
10  //reg define
11  reg [19:0] cnt ;
12  reg        key_reg ;
13
14  //*****************************************************
15  //**                    main code
16  //*****************************************************
17
18  //按键值消抖
19  always @ (posedge sys_clk or negedge sys_rst_n) begin
20      if(!sys_rst_n) begin
21          cnt <= 20'd0;
22          key_reg <= 1'b1;
23      end
24      else begin
25          key_reg <= key;             //将按键值延迟一拍
26          if(key_reg != key) begin    //检测到按键状态发生变化
27              cnt <= 20'd100_0000;    //则将计数器置为 20'd100_0000,
28                                      //即延时 100_0000 * 20ns(1s/50MHz) = 20ms
```

```
29          end
30          else begin                  //如果当前按键值和前一个按键值一样，即按键没有发生变化
31              if(cnt > 20'd0)         //则计数器递减到 0
32                  cnt <= cnt - 1'b1;
33              else
34                  cnt <= 20'd0;
35          end
36      end
37  end
38
39  //将消抖后的最终的按键值送出去
40  always @ (posedge sys_clk or negedge sys_rst_n) begin
41      if(!sys_rst_n) begin
42          key_value <= 1'b1;
43          key_flag  <= 1'b0;
44      end
45      //在计数器递减到 1 时送出按键值
46      else if(cnt == 20'd1) begin
47          key_value <= key;
48          key_flag  <= 1'b1;
49          end
50      else begin
51          key_value <= key_value;
52          key_flag  <= 1'b0;
53      end
54  end
55
56  endmodule
```

代码中的第 26 行，每检测到按键被按下或松开，就让计数器从 100_0000 开始递减，时长 20ms。在这 20ms 期间，每当有抖动产生，计数器就被重置回 100_0000，即重新开始计时 20ms。代码中的第 46 行，只有在计数器递减到 1 时，即此时计数器计时完了 20ms，才会寄存按键的值。这样，每当按键被按下或松开，20ms 内的抖动就被消除了。

蜂鸣器控制模块的代码如下：

```
1   module beep_control(
2       input         sys_clk,
3       input         sys_rst_n,
4
5       input         key_value,
6       input         key_flag,
7       output  reg   beep
8       );
```

```
9
10  //*****************************************************
11  //**                    main code
12  //*****************************************************
13
14  //每次按键按下时，就翻转蜂鸣器的状态
15  always @ (posedge sys_clk or negedge sys_rst_n) begin
16      if(!sys_rst_n)
17          beep <= 1'b1;
18      else if(key_flag && (key_value == 1'b0))
19          beep <= ~beep;
20  end
21
22  endmodule
```

beep 初始状态为高电平，蜂鸣器鸣叫。当 key_flag 拉高表明消抖之后的按键数据有效，此时若检测到按键值为 0（即按键被按下），就将 beep 状态取反，以改变蜂鸣器的鸣叫状态。

8.5 下载验证

连接开发板的电源和下载器，并打开电源开关。在工程编译之后，将生成的 bit 文件下载到开发板中。

下载完成后，蜂鸣器处于鸣叫状态。然后按下按键 PL_KEY0，蜂鸣器停止鸣叫。再次按下按键，蜂鸣器再次开始鸣叫。如下图所示：

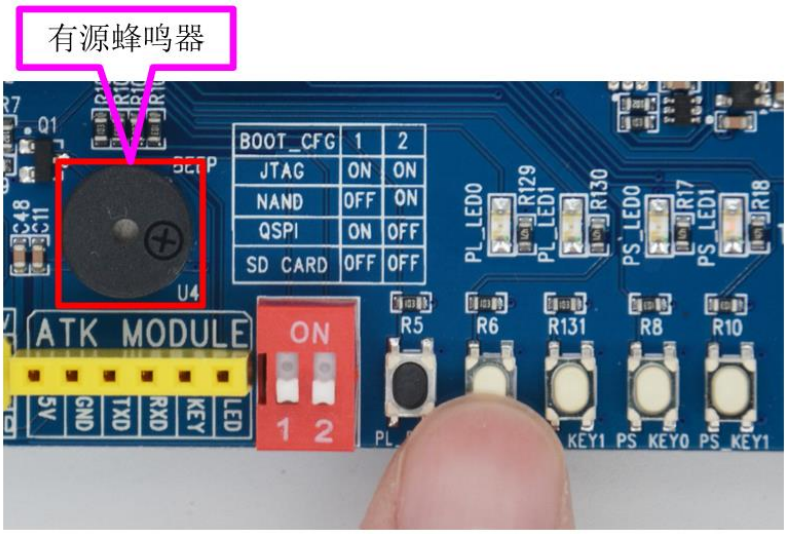


图 8.5.1 实验现象

第九章　触摸按键控制 LED 灯实验

随着电子技术的不断发展，按键的应用场景越来越广泛。触摸按键在稳定性、使用寿命、抗干扰能力等方面都优于传统的机械按键，被广泛应用于遥控器，便携电子设备，楼道电灯开关，各类家电控制面板等方面。本章将介绍触摸按键的控制方法并使用开发板上的触摸按键控制 led 的亮灭。

本章分为以下几个章节：

9.1 触摸按键简介

触摸按键主要可分为四大类：电阻式、电容式、红外感应式以及表面声波式。根据其属性的不同，每种触摸按键都有其合适的使用领域。

电阻式触摸按键由多块导电薄膜按照按键的位置印制而成，但由于耐用性较差且维护复杂，目前使用率较低；红外感应式触摸按键通过红外扫描来识别按键位置，一般在较恶劣的环境下使用；表面声波式触摸按键利用声波扫描来识别是否按下，使用寿命长，较适合公共场所的 POS 机，无人售货机等处使用。

电容式触摸按键：这种按键的诞生主要是为了克服电阻式按键耐用性差的不足所提出的。电容式触摸按键采用电容量为评判标准，其感应区域可穿透绝缘外壳（玻璃、塑料等）20mm 以上。其灵敏度和可靠性不会因环境条件的改变或长期使用而发生变化，具有防水、强抗干扰能力强、适应温度范围广以及使用寿命长等优点，是现代使用最广泛，发展最迅速的一种触摸按键。

接下来，我们具体的了解一下电容触摸按键的构造和工作原理。

电容触摸按键主要由按键 IC 部分和电容部分构成。按键 IC 部分主要由元器件供应商提供，用于将电容的变化转换为电信号。电容部分指的是由电容极板，地，隔离区等组成触摸按键的电容环境。

任何两个导电的物体之间都存在着感应电容，在周围环境不变的情况下，该感应电容值是固定不变的。如下图所示，手指接触到触摸按键时，按键和手指之间产生寄生电容，使按键的总容值增加。

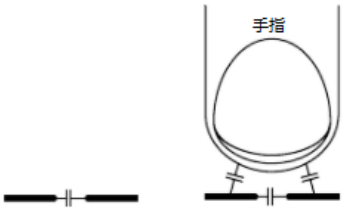


图 9.1.1 触摸按键寄生电容示意图

触摸按键按下前后，电容的变化如下图所示。电容式触摸按键 IC 在检测到按键的感应电容值改变，并超过一定的阈值后，将输出有效信号表示按键被按下。

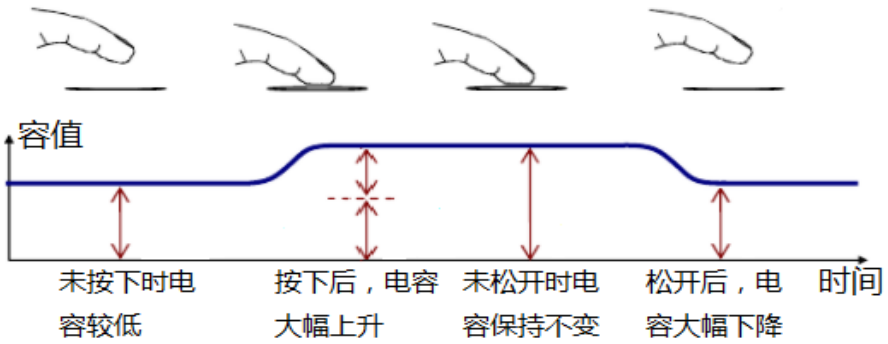


图 9.1.2 触摸过程电容变化示意图

9.2 实验任务

本节的实验任务是使用触摸按键控制 LED 灯的亮灭，开发板上电后 LED 为点亮状态，手指触摸后 LED 熄灭；当再次触摸时，LED 点亮。

9.3 硬件设计

启明星开发板上的触摸按键部分的原理图如图 9.3.1 所示。其中 TPAD 是芯片的输出引脚，连接到 ZYNQ

的 IO 端口。

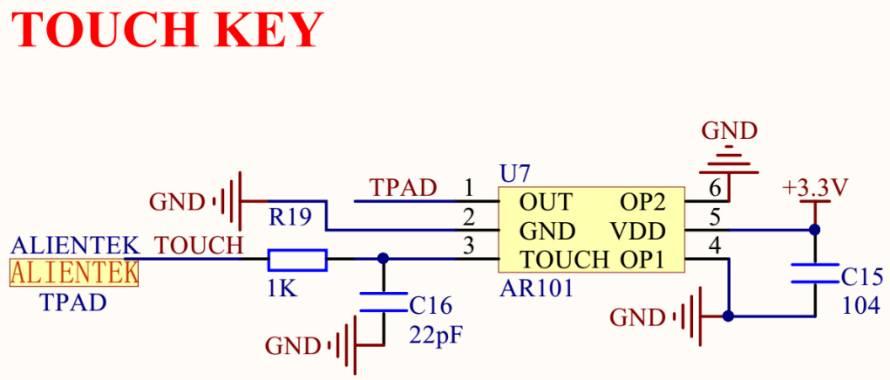


图 9.3.1 触摸按键电路原理图

开发板上所使用的触摸 IC 型号为 AR101（或者 JL223B，和 AR101 完全兼容），它可以通过 OP1 和 OP2 两个引脚选择不同的工作模式：OP1 拉低时，OUT 引脚输出信号高电平有效；OP1 拉高时，OUT 输出信号低电平有效。

当 OP2 拉低时，触摸 IC 工作在同步模式（类似于非自锁的轻触按键），即触摸时输出有效电平，松开后输出无有效电平；OP2 拉高时触摸 IC 工作在保持模式（类似于自锁按键），即检测到触摸操作后输出有效电平，松开后，输出电平保持不变。当再次检测到触摸操作时，输出电平变化并继续保持。

图 9.3.1 中触摸 IC 的引脚 OP1 和 OP2 均拉低，因此当手指按在触摸按键上时，TOUT 管脚输高电平，松开后输出低电平。

本实验中，系统时钟、复位按键、触摸按键和 LED 灯的管脚分配如下表所示，其中时钟晶振位于核心板上，复位按键、触摸按键和 LED 位于底板上。

表 9.3.1 触摸按键控制LED管脚分配图

信号名	方向	管脚	端口说明	电平标准
sys_clk	input	U18	系统时钟，50M	LVCMOS33
sys_rst_n	input	J15	系统复位，低有效	LVCMOS33
touch_key	input	L19	触摸按键信号	LVCMOS33
led	output	J18	LED	LVCMOS33

对应的 XDC 约束语句如下所示：

```
#IO 约束
set_property -dict {PACKAGE_PIN U18 IOSTANDARD LVCMOS33} [get_ports sys_clk]
set_property -dict {PACKAGE_PIN J15 IOSTANDARD LVCMOS33} [get_ports sys_rst_n]
set_property -dict {PACKAGE_PIN L19 IOSTANDARD LVCMOS33} [get_ports touch_key]
set_property -dict {PACKAGE_PIN J18 IOSTANDARD LVCMOS33} [get_ports led]
```

9.4 程序设计

本次设计的模块端口及信号连接如下图所示。通过捕获触摸按键端口的上升沿，得到一个时钟周期的脉冲信号，来控制 LED 灯的亮灭。

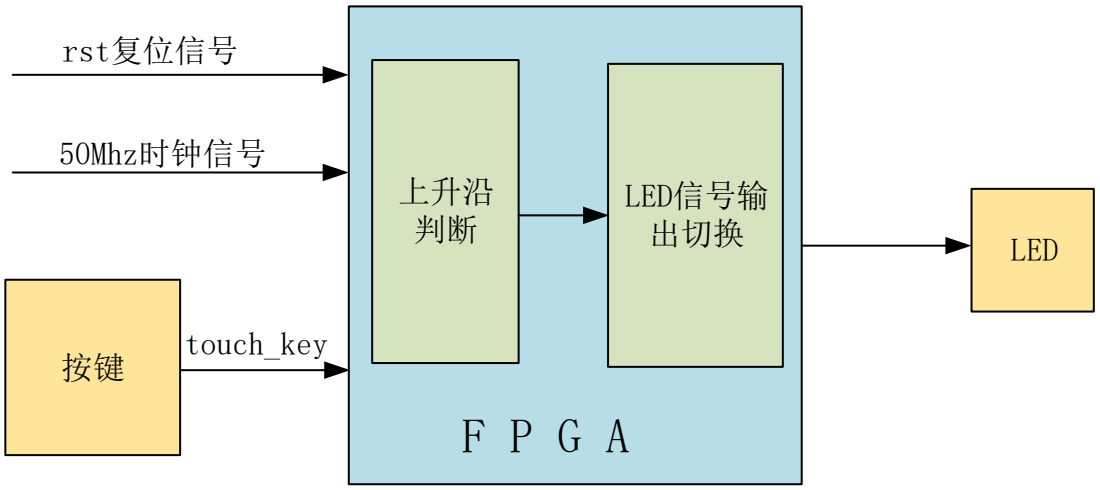


图 9.4.1 模块端口及信号连接图

触摸按键控制 led 代码如下：

```
1  module touch_led(
2      //input
3      input         sys_clk,      //时钟信号 50Mhz
4      input         sys_rst_n,    //复位信号
5      input         touch_key,    //触摸按键
6
7      //output
8      output  reg   led           //LED 灯
9  );
10
11 //reg define
12 reg    touch_key_d0;
13 reg    touch_key_d1;
14
15 //wire define
16 wire   touch_en;
17
18 //*****************************************************
19 //**                    main code
20 //*****************************************************
21
22 //捕获触摸按键端口的上升沿，得到一个时钟周期的脉冲信号
23 assign  touch_en = (~touch_key_d1) & touch_key_d0;
24
25 //对触摸按键端口的数据延迟两个时钟周期
26 always @ (posedge sys_clk or negedge sys_rst_n) begin
27     if(!sys_rst_n) begin
```

```
28          touch_key_d0 <= 1'b0;
29          touch_key_d1 <= 1'b0;
30      end
31      else begin
32          touch_key_d0 <= touch_key;
33          touch_key_d1 <= touch_key_d0;
34      end
35  end
36
37  //根据触摸按键上升沿的脉冲信号切换 led 状态
38  always @ (posedge sys_clk or negedge sys_rst_n) begin
39      if (!sys_rst_n)
40          led <= 1'b1;        //默认状态下,点亮 LED
41      else begin
42          if (touch_en)
43              led <= ~led;
44      end
45  end
46
47  endmodule
```

程序中第 22 至 35 行是一个经典的边沿检测电路，通过检测 touch_key 的上升沿来捕获按键按下的信号，一旦检测到按键按下，输出一个时钟周期的脉冲 touch_en。每当检测到 touch_en 的脉冲信号，led 取反一次。

为了验证我们的程序，我们在 Vivado 内对代码进行仿真。

Testbench 模块代码如下：

```
1   `timescale 1ns / 1ps
2
3   module tb_touch_led();
4
5   //reg define
6   reg     sys_clk;
7   reg     sys_rst_n;
8   reg     touch_key;
9
10  //wire define
11  wire          led ;
12
13  always #10 sys_clk = ~sys_clk;
14
15  initial begin
16      sys_clk = 1'b0;
17      sys_rst_n = 1'b0;
```

```
18      touch_key = 0;
19      #200
20      sys_rst_n = 1'b1;
21      //touch_key 信号变化
22      #40  touch_key = 1'b1 ;  //40ns 后触摸按键按下
23      #200 touch_key = 1'b0 ;  //200ns 触摸按键抬起
24      #40  touch_key = 1'b1 ;  //40ns 后触摸按键按下
25      #200 touch_key = 1'b0 ;  //200ns 触摸按键抬起
26      #40  touch_key = 1'b1 ;  //40ns 后触摸按键按下
27      #200 touch_key = 1'b0 ;  //200ns 触摸按键抬起
28      #40  touch_key = 1'b1 ;  //40ns 后触摸按键按下
29      #200 touch_key = 1'b0 ;  //200ns 触摸按键抬起
30  end
31
32  touch_led  u_touch_led(
33      .sys_clk   (sys_clk),
34      .sys_rst_n (sys_rst_n),
35      .touch_key (touch_key),
36      .led       (led)
37  );
38
39  endmodule
```

仿真得到的波形图如图 9.4.2 所示。从图中可以看出，当 touch_key 信号由电平变为高电平时，touch_key_d0 和 touch_key_d1 信号分别延迟 touch_key 一个时钟周期和两个时钟周期，将 touch_key_d0 信号的和取反后的 touch_key_d1 信号相与，就得到一个时钟周期的脉冲信号（touch_en）。当检测到 touch_en 信号为高电平时，对 led 信号进行取反，从而实现触摸按键控制 led 灯的功能。

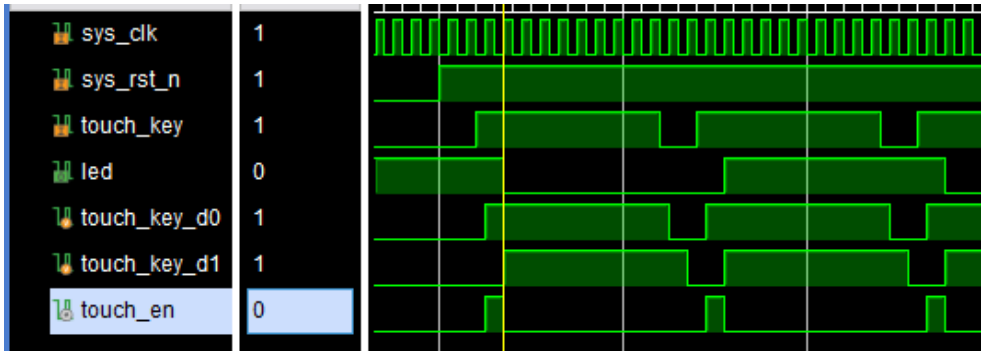


图 9.4.2 仿真波形

9.5 下载验证

编译工程并生成比特流.bit 文件。将下载器一端连接电脑，另一端与开发板上的 JTAG 下载口连接，连接电源线，并打开开发板的电源开关。

点击 Vivado 左侧“Flow Navigator”窗口最下面的“Open Hardware Manager”，此时 Vivado 软件识别到下载器，点击“Hardware”窗口中“Progam Device”下载程序，在弹出的界面中选择“Program”下载程

序。

程序下载完成后，可以看到底板上的 PL_LED0 处于点亮状态，触碰一次触摸按键，就可以看到 LED 灯熄灭，再次触摸点亮 LED 灯。如下图所示：

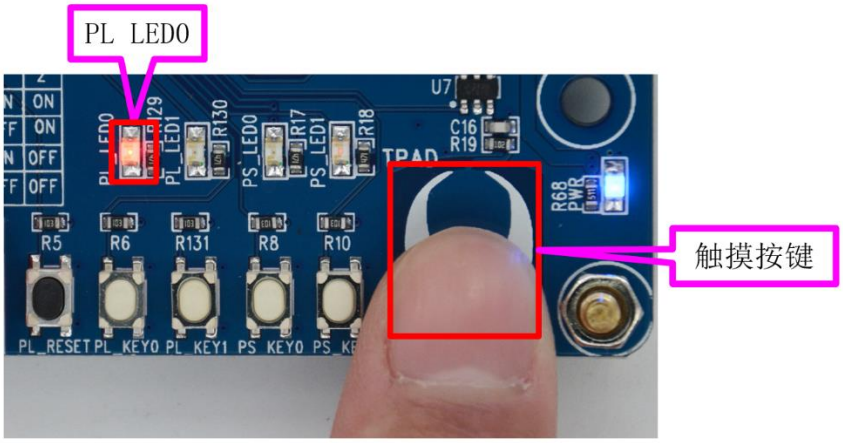


图 9.5.1 按键控制 LED 灯实验现象

第十章　呼吸灯实验

呼吸灯最早由苹果公司发明并应用于笔记本睡眠提示上，其一经展出，立刻吸引众多科技厂商争相效仿，并广泛用于各种电子产品中，尤其是智能手机。呼吸灯其实是在微处理器的控制下，由暗渐亮、然后再由亮渐暗，模仿人呼吸方式的 LED 灯。

本章分为以下几个章节：

10.1 呼吸灯简介

呼吸灯采用 PWM 的方式，在固定的频率下，通过调整占空比的方式来控制 LED 灯亮度的变化。PWM（Pulse Width Modulation），即脉冲宽度调制，它利用微处理器输出的 PWM 信号，实现对模拟电路控制的一种非常有效的技术，广泛应用于测量、通信、功率控制等领域。

在由计数器产生的固定周期的 PWM 信号下，如果其占空比为 0，则 LED 灯不亮；如果其占空比为 100%，则 LED 灯最亮。所以将占空比从 0 到 100%，再从 100%到 0 不断变化，就可以实现 LED 灯的“呼吸”效果。

PWM 占空比调节示意图如下图所示：

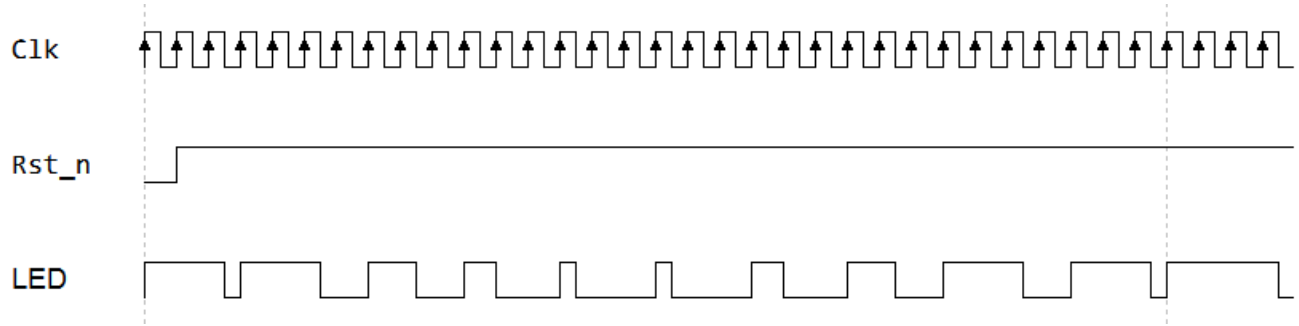


图 10.1.1 呼吸灯 PWM 占空比示意图

由上图可知，LED 高电平的时间由长渐渐变短，再由短渐渐变长，如果 LED 灯是高电平点亮，则 LED 灯会呈现出亮度由亮到暗，再由暗到亮的过程。

10.2 实验任务

本节实验任务是使用正点原子 ZYNQ 开发板（核心板）上的 PL LED，实现呼吸灯的效果，即由灭渐亮，然后再由亮渐灭。

10.3 硬件设计

发光二极管的原理图如下图所示，PL LED 发光二极管位于核心板上，其阴极通过 330 欧姆的电阻连到地（GND），阳极与 ZYNQ 的 IO 相连，LED 与地之间的电阻起到限流作用。当 PL_LED 输出高电平时，点亮 LED 灯，当 PL LED 输出低电平时，LED 灯熄灭。

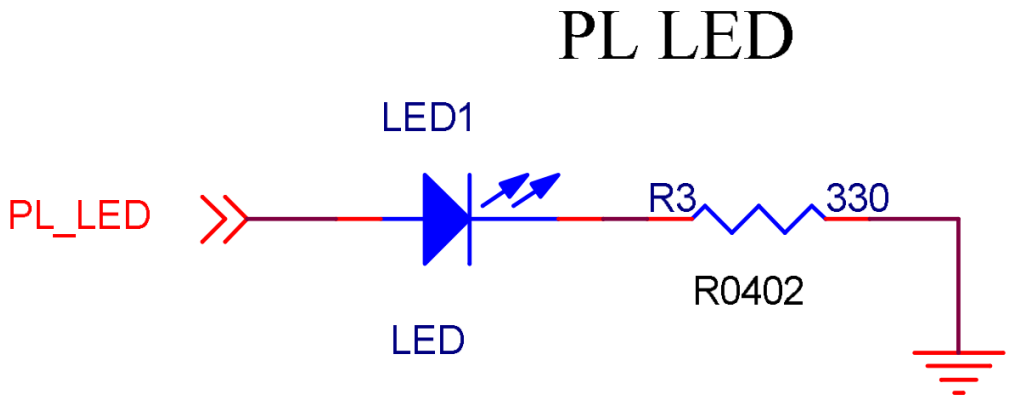


图 10.3.1 呼吸灯硬件原理图

本实验中，系统时钟、按键复位以及 LED 端口的管脚分配如下表所示：

表 10.3.1 呼吸灯实验管脚分配

信号名	方向	管脚	端口说明	电平标准
sys_clk	input	U18	系统时钟，50MHz	LVCMOS33
sys_rst_n	input	J15	系统复位按键，低电平有效	LVCMOS33
led	output	J16	PL LED（核心板）	LVCMOS33

对应的 XDC 约束语句如下所示：

```
#IO 管脚约束
set_property -dict {PACKAGE_PIN U18 IOSTANDARD LVCMOS33} [get_ports sys_clk]
set_property -dict {PACKAGE_PIN J15 IOSTANDARD LVCMOS33} [get_ports sys_rst_n]
set_property -dict {PACKAGE_PIN J16 IOSTANDARD LVCMOS33} [get_ports led]
```

10.4 程序设计

本次实验的模块端口及结构框图如下图所示。

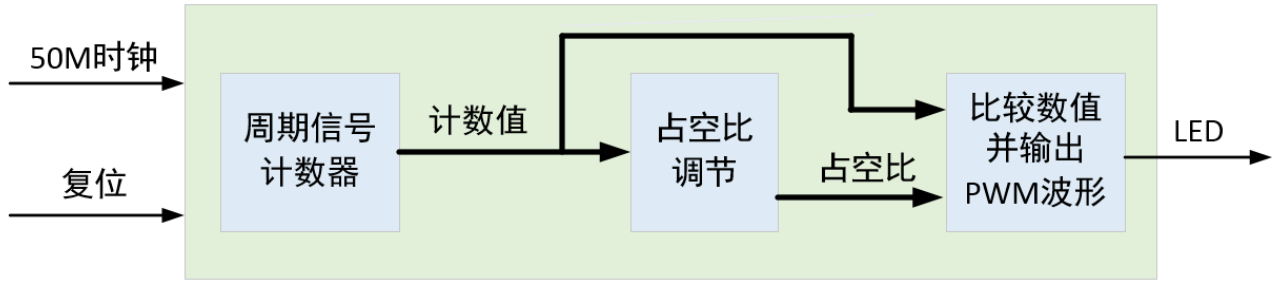


图 10.4.1 模块端口及信号连接图

周期信号计数器用于产生驱动 LED 的脉冲信号，本次实验的周期信号频率为 1Khz，其占空比由后级逻辑在每个周期之后进行递增或递减，最后再对当前计数值和占空比计数值进行比较，以输出占空比可调的脉冲信号。

呼吸灯代码如下：

```
1  module breath_led(
2      input   sys_clk   ,  //时钟信号 50Mhz
3      input   sys_rst_n ,  //复位信号
4
5      output  led          //LED
6  );
7
8  //reg define
9  reg  [15:0]  period_cnt ;   //周期计数器频率：1khz 周期:1ms  计数值:1ms/20ns=50000
10 reg  [15:0]  duty_cycle ;   //占空比数值
11 reg          inc_dec_flag ; //0 递增  1 递减
12
13 //*****************************************************
```

```
14 //**                     main code
15 //*************************************************
16
17 //根据占空比和计数值之间的大小关系来输出 LED
18 assign   led = (period_cnt >= duty_cycle) ?  1'b1  :  1'b0;
19
20 //周期计数器
21 always @(posedge sys_clk or negedge sys_rst_n) begin
22     if(!sys_rst_n)
23         period_cnt <= 16'd0;
24     else if(period_cnt == 16'd50000)
25         period_cnt <= 16'd0;
26     else
27         period_cnt <= period_cnt + 1'b1;
28 end
29
30 //在周期计数器的节拍下递增或递减占空比
31 always @(posedge sys_clk or negedge sys_rst_n) begin
32     if(!sys_rst_n) begin
33         duty_cycle   <= 16'd0;
34         inc_dec_flag <= 1'b0;
35     end
36     else begin
37         if(period_cnt == 16'd50000) begin    //计满 1ms
38             if(inc_dec_flag == 1'b0) begin   //占空比递增状态
39                 if(duty_cycle == 16'd50000)  //如果占空比已递增至最大
40                     inc_dec_flag <= 1'b1;    //则占空比开始递减
41                 else                         //否则占空比以 25 为单位递增
42                     duty_cycle <= duty_cycle + 16'd25;
43             end
44             else begin                       //占空比递减状态
45                 if(duty_cycle == 16'd0)      //如果占空比已递减至 0
46                     inc_dec_flag <= 1'b0;    //则占空比开始递增
47                 else                         //否则占空比以 25 为单位递减
48                     duty_cycle <= duty_cycle - 16'd25;
49             end
50         end
51     end
52 end
53
54 endmodule
```

第 21-28 行是 1KHz 周期信号的计数器，用于产生 1KHz 的 LED 驱动信号。第 31-52 行的 always 块为占空比设定模块，每次计数完了一个周期，就根据递增/递减标志来对占空比计数值（duty_cycle）进行递增/递减 25 个计数值，这个递增或者递减的数值大小可以用来控制呼吸灯的呼吸频率。

如果占空比计数值（duty_cycle）已经递增到了最大，则呼吸灯已经处于最亮的状态，接下来开始递减；反之，如果占空比计数至已经递减到了最小，即 0，则呼吸灯处于熄灭的状态，接下来开始递增；如此循环往复，最终实现了流水灯的效果。

在代码的第 18 行通过组合逻辑把当前的周期计数值和占空比计数值进行比较，来判断 LED 的输出电平。在一个周期内，如果当前的周期计数值小于等于占空比计数值，则 LED 输出高电平，即点亮；如果当前的周期计数值大于占空比计数值，则 LED 输出低电平，即熄灭。

10.5 下载验证

编译工程并生成比特流.bit 文件。将下载器一端连接电脑，另一端与开发板上的 JTAG 下载口连接，连接电源线，并打开开发板的电源开关。

点击 Vivado 左侧“Flow Navigator”窗口最下面的“Open Hardware Manager”，此时 Vivado 软件识别到下载器，点击“Hardware”窗口中“Progam Device”下载程序，在弹出的界面中选择“Program”下载程序。

程序下载完成后，可以看到核心板的 PL LED 灯由暗慢慢变亮，再由亮慢慢变暗，即呈现出“呼吸”的效果，如下图所示：

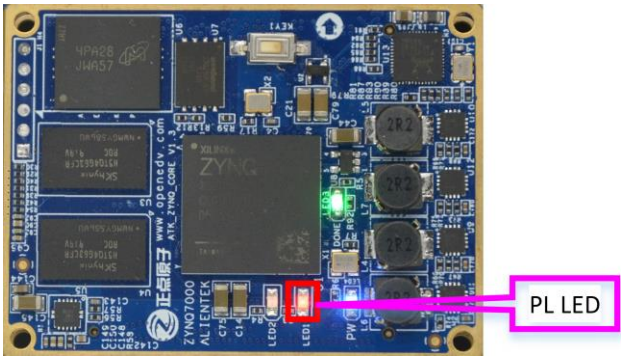


图 10.5.1 开发板实验现象

第十一章 IP 核之 MMCM/PLL 实验

PLL 的英文全称是 Phase Locked Loop，即锁相环，是一种反馈控制电路。PLL 对时钟网络进行系统级的时钟管理和偏移控制，具有时钟倍频、分频、相位偏移和可编程占空比的功能。Xilinx 7 系列器件中的时钟资源包含了时钟管理单元 CMT，每个 CMT 由一个 MMCM 和一个 PLL 组成。对于一个简单的设计来说，FPGA 整个系统使用一个时钟或者通过编写代码的方式对时钟进行分频是可以完成的，但是对于稍微复杂一点的系统来说，系统中往往需要使用多个时钟和时钟相位的偏移，且通过编写代码输出的时钟无法实现时钟的倍频，因此学习 Xilinx MMCM/PLL IP 核的使用方法是我们学习 FPGA 的一个重要内容。本章我们将通过一个简单的例程来向大家介绍一下 MMCM/PLL IP 核的使用方法。

本章包括以下几个部分：

11.1 MMCM/PLL IP 核简介

11.2 实验任务

11.3 硬件设计

11.4 程序设计

11.5 下载验证

11.1 MMCM/PLL IP 核简介

锁相环作为一种反馈控制电路，其特点是利用外部输入的参考信号控制环路内部震荡信号的频率和相位。因为锁相环可以实现输出信号频率对输入信号频率的自动跟踪，所以锁相环通常用于闭环跟踪电路。锁相环在工作的过程中，当输出信号的频率与输入信号的频率相等时，输出电压与输入电压保持固定的相位差值，即输出电压与输入电压的相位被锁住，这就是锁相环名称的由来。

Xilinx 7 系列器件中具有时钟管理单元 CMT 时钟资源，xc7z020 芯片内部有 4 个 CMT，xc7z010 芯片内部有 2 个 CMT，为设备提供强大的系统时钟管理以及高速 I/O 通信的能力。时钟管理单元 CMT 的总体框图如下图所示。

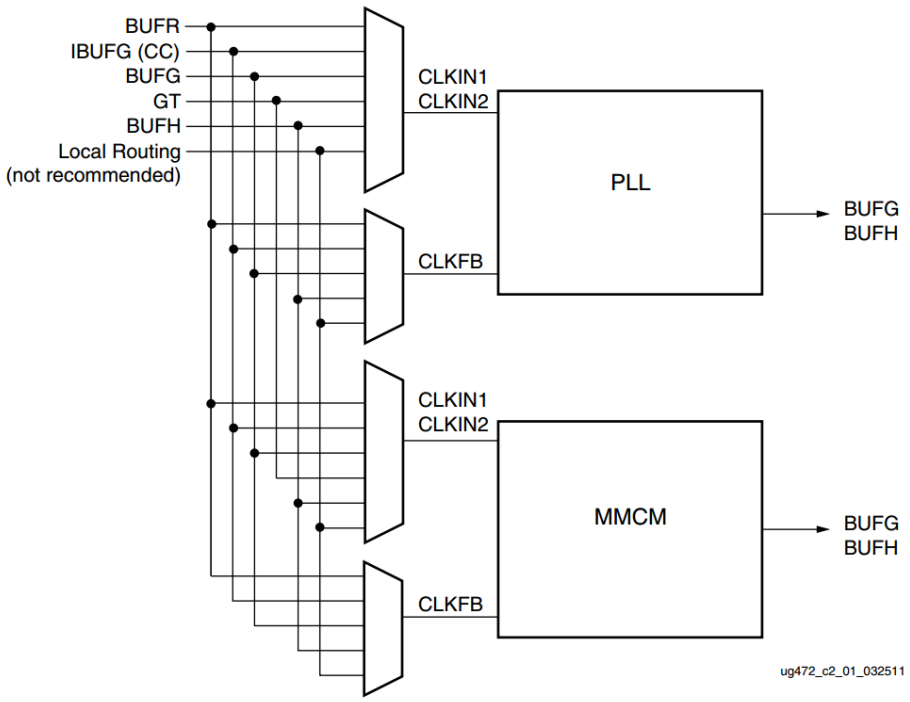


Figure 3-1: **Block Diagram of the 7 Series FPGAs CMT**

图 11.1.1 CMT 总体框图

MMCM/PLL 的参考时钟输入可以是 IBUFG(CC)即具有时钟能力的 IO 输入、区域时钟 BUFR、全局时钟 BUFG、GT 收发器输出时钟、行时钟 BUFH 以及本地布线（不推荐使用本地布线来驱动时钟资源）。在最多的情况下，MMCM/PLL 的参考时钟输入都是来自 IBUFG(CC)即具有时钟能力的 IO 输入，本实验也是如此。MMCM/PLL 的输出可以驱动全局时钟 BUFG 和行时钟 BUFH 等等。BUFG 能够驱动整个器件内部的 PL 侧通用逻辑的所有时序单元的时钟端口。BUFG/BUFH/CMT 在一个时钟区域内的连接框图如下图所示。

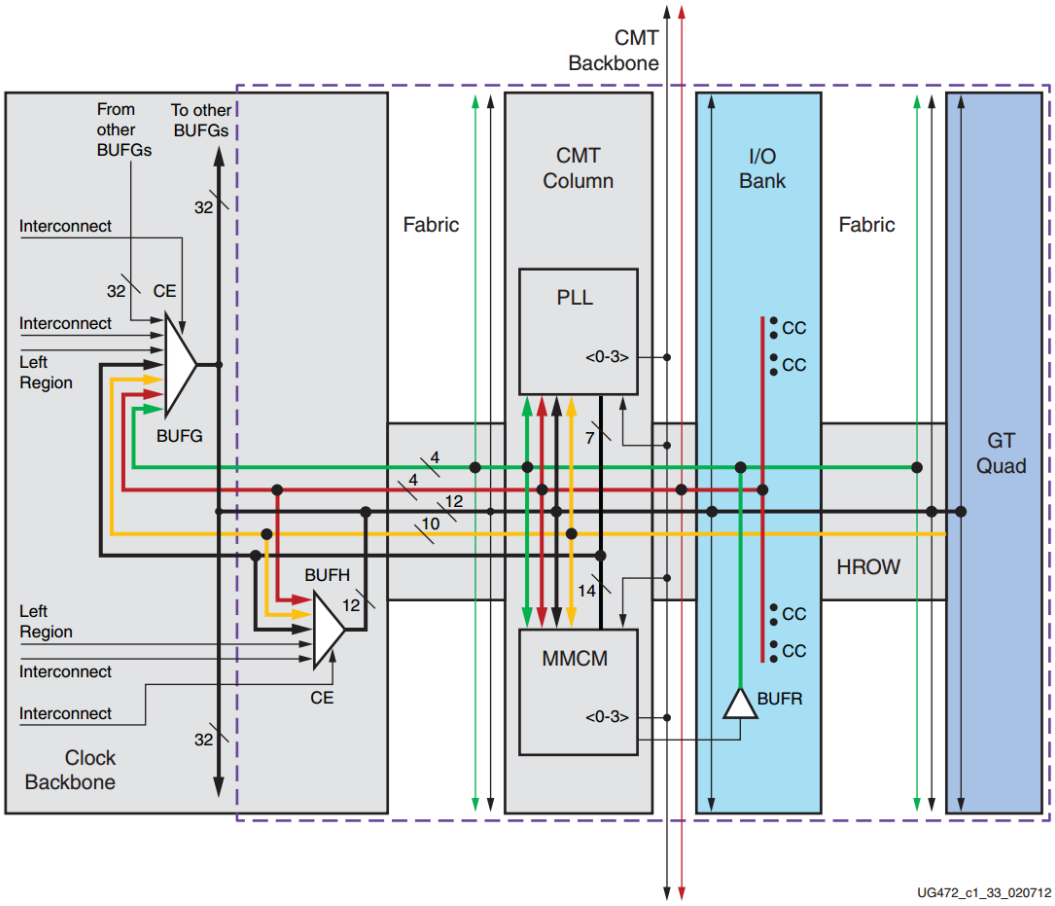


图 11.1.2 BUFG/BUFH/CMT 在一个时钟区域内的连接

在本实验中，读者可以简单地理解为：外部时钟连接到具有时钟能力的输入引脚 CCIO（Clock-Capable Input），进入 MMCM/PLL，产生不同频率和不同相位的时钟信号，然后驱动全局时钟资源 BUFG。但是要进行更深入的 FPGA 开发，就必须理解器件的时钟资源架构。有关 Xilinx 时钟资源和 CMT 的更详细信息，读者后期可以花一些时间和精力去学习一下 Xilinx 官方的手册文档“UG472，7 Series FPGAs Clocking Resources User Guide”里的介绍。

MMCM 和 PLL 的总体框图如下图所示。

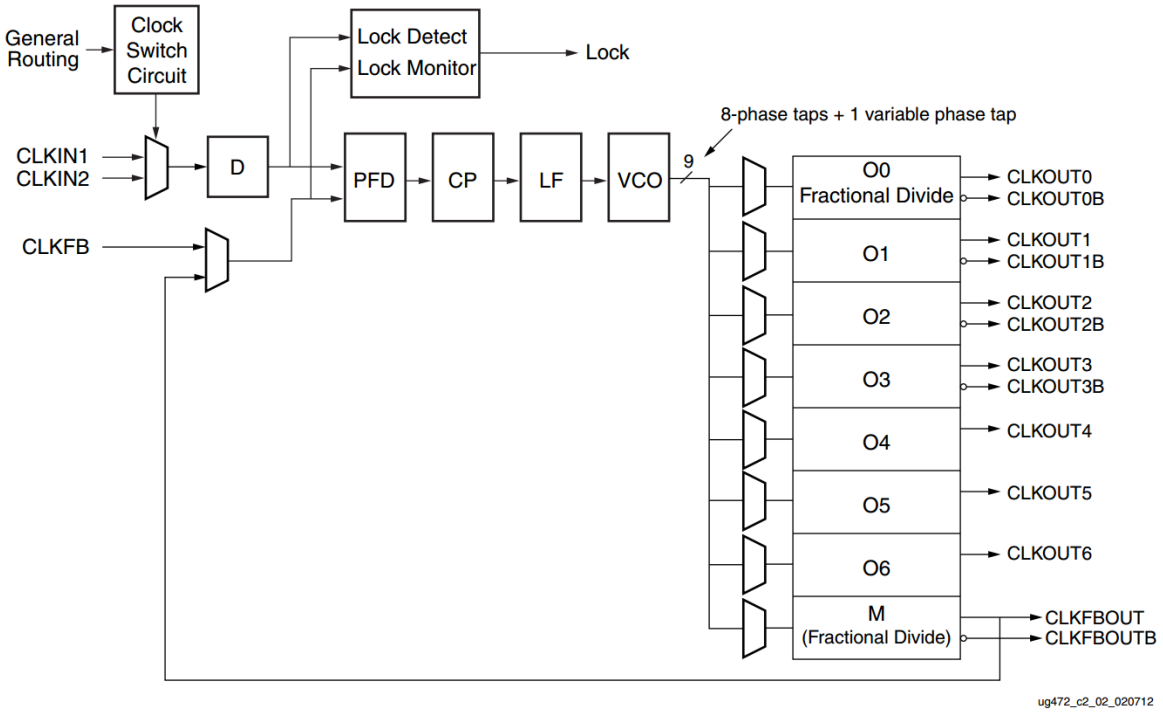


Figure 3-2: **Detailed MMCM Block Diagram**

图 11.1.3 MMCM 总体框图

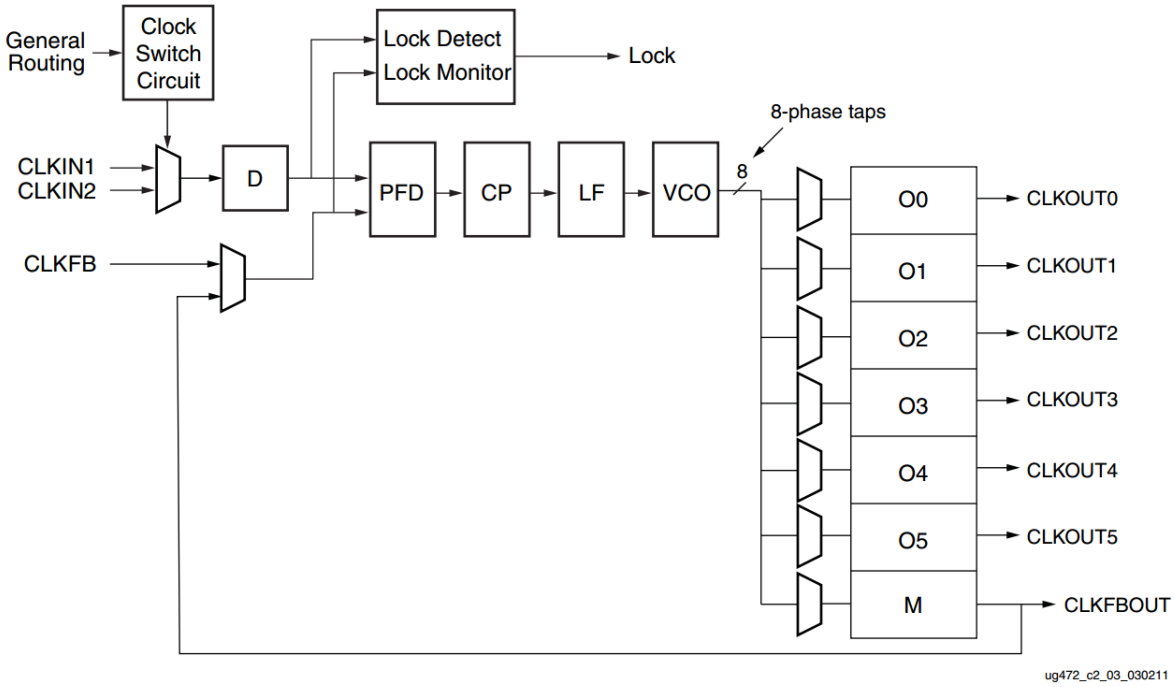


Figure 3-3: **Detailed PLL Block Diagram**

图 11.1.4 PLL 总体框图

其中 MMCM 的功能是 PLL 的超集，其具有比 PLL 更强大的相移功能。MMCM 主要用于驱动器件逻辑（CLB、DSP、RAM 等）的时钟。PLL 主要用于为内存接口生成所需的时钟信号，但也具有与器件逻辑的连接，因此如果需要额外的功能，它们可以用作额外的时钟资源。

PLL 由以下几部分组成：前置分频计数器（D 计数器）、相位-频率检测器（PFD，Phase-Frequency Detector）电路，电荷泵（Charge Pump）、环路滤波器（Loop Filter）、压控振荡器（VCO，Voltage Controlled Oscillator）、反馈乘法器计数器（M 计数器）和后置分频计数器（O1-O6 计数器）。

在工作时，PFD 检测其参考频率（F_{REF}）和反馈信号（Feedback）之间的相位差和频率差，控制电荷泵和环路滤波器将相位差转换为控制电压；VCO 根据不同的控制电压产生不同的震荡频率，从而影响 Feedback 信号的相位和频率。在 F_{REF} 和 Feedback 信号具有相同的相位和频率之后，就认为 PLL 处于锁相的状态。

在反馈路径中插入 M 计数器会使 VCO 的震荡频率是 F_{REF} 信号频率的 M 倍，F_{REF} 信号等于输入时钟（F_{IN}）除以预缩放计数器（D）。参考频率用以下方程描述：$F_{REF}= F_{IN}/D$，VCO 输出频率为 $F_{VCO}= F_{IN}*M/D$，PLL 的输出频率为 $F_{OUT}=（F_{IN}*M）/（N*O）$。

Xilinx 提供了用于实现时钟功能的 IP 核 Clocking Wizard，该 IP 核能够根据用户的时钟需求自动配置器件内部的 CMT 及时钟资源，以实现用户的时钟需求。在这里我们主要讲解的是如何使用该 IP 核，有关该 IP 核的更详细介绍，读者可以参阅 Xilinx 官方的手册文档“PG065，Clocking Wizard v6.0 LogiCORE IP Product Guide”。

11.2 实验任务

本节实验任务是使用 Zynq 开发板输出 4 个不同时钟频率或相位的时钟，并在 Vivado 中进行仿真以验证结果，最后生成比特流文件并将下载到开发板上，使用示波器来测量时钟的频率。

11.3 硬件设计

本章实验将 Clocking Wizard IP 核产生的 4 个时钟 100MHz、100MHz_180deg 、50MHz、25MHz，连接到开发板的 J3 扩展口 IO 上（靠近 ATK-MODULE 接口），分别是第 29、31、33、35 号脚。扩展口原理图如下图所示：

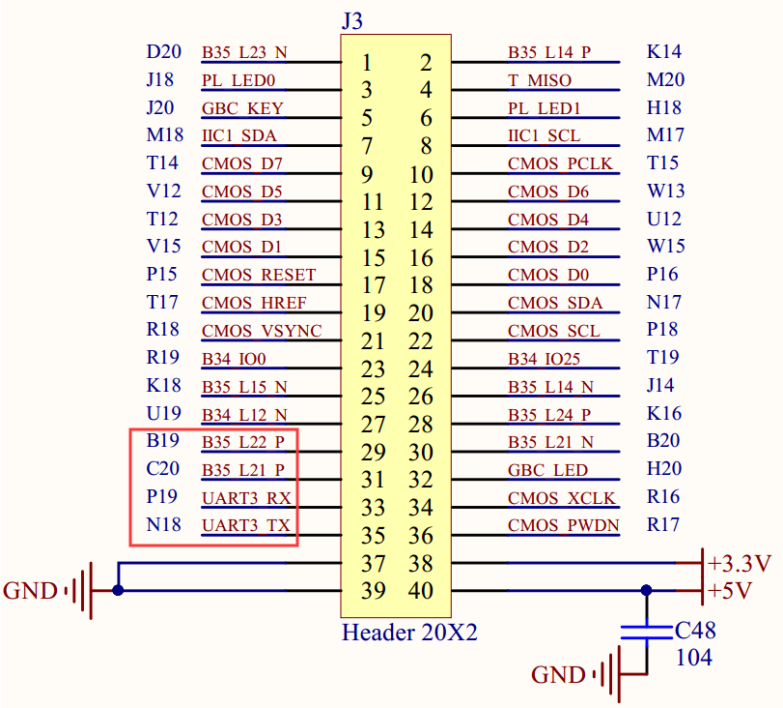


图 11.3.1 扩展口原理图

本实验中，各端口信号的管脚分配如下表所示：

表 11.3.1 IP 核之 PLL 实验管脚分配

信号名	方向	管脚	端口说明	电平标准
sys_clk	Input	U18	系统时钟，50Mhz	LVCMOS33
sys_rst_n	Input	J15	系统复位，低有效	LVCMOS33
clk_100m	output	B19	100Mhz 时钟	LVCMOS33
clk_100m_180deg	output	C20	100Mhz 时钟，相位偏移 180 度	LVCMOS33
clk_50m	output	P19	50Mhz 时钟	LVCMOS33
clk_25m	output	N18	25Mhz 时钟	LVCMOS33

对应的 XDC 约束语句如下所示：

```
set_property -dict {PACKAGE_PIN U18 IOSTANDARD LVCMOS33} [get_ports sys_clk]
set_property -dict {PACKAGE_PIN J15 IOSTANDARD LVCMOS33} [get_ports sys_rst_n]
set_property -dict {PACKAGE_PIN B19 IOSTANDARD LVCMOS33} [get_ports clk_100m]
set_property -dict {PACKAGE_PIN C20 IOSTANDARD LVCMOS33} [get_ports clk_100m_180deg]
set_property -dict {PACKAGE_PIN P19 IOSTANDARD LVCMOS33} [get_ports clk_50m]
set_property -dict {PACKAGE_PIN N18 IOSTANDARD LVCMOS33} [get_ports clk_25m]
```

11.4 程序设计

我们首先创建一个空的工程，工程名为“ip_clk_wiz”。接下来添加 PLL IP 核。在 Vivado 软件的左侧“Flow Navigator”栏中单击“IP Catalog”，“IP Catalog”按钮以及单击后弹出的“IP Catalog”窗口如下图所示。

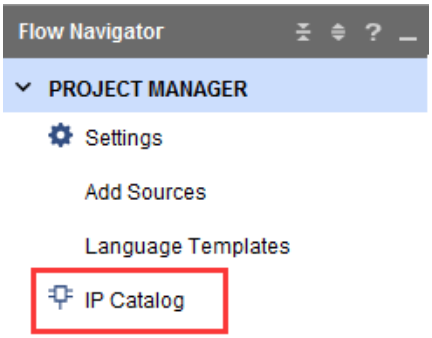


图 11.4.1 “IP Catalog”按钮

图 11.4.2 “IP Catalog”窗口

打开“IP Catalog”窗口后，在搜索栏中输入“clock”关键字，可以看到 Vivado 已经自动查找出了与关键字匹配的 IP 核名称，如下图所示。

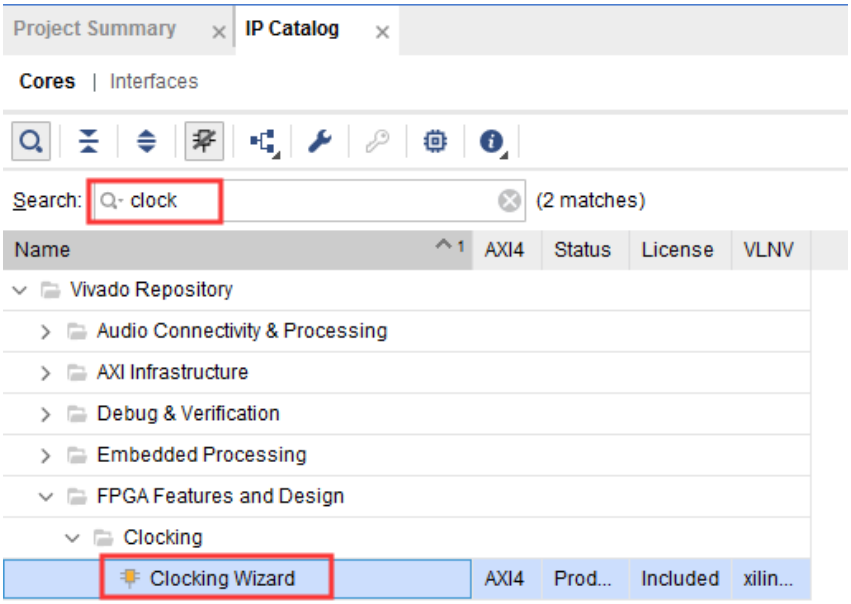


图 11.4.3 搜索栏中输入关键字

我们双击“FPGA Features and Design”→“Clocking”下的“Clocking Wizard”，弹出“Customize IP”窗口，如下图所示。

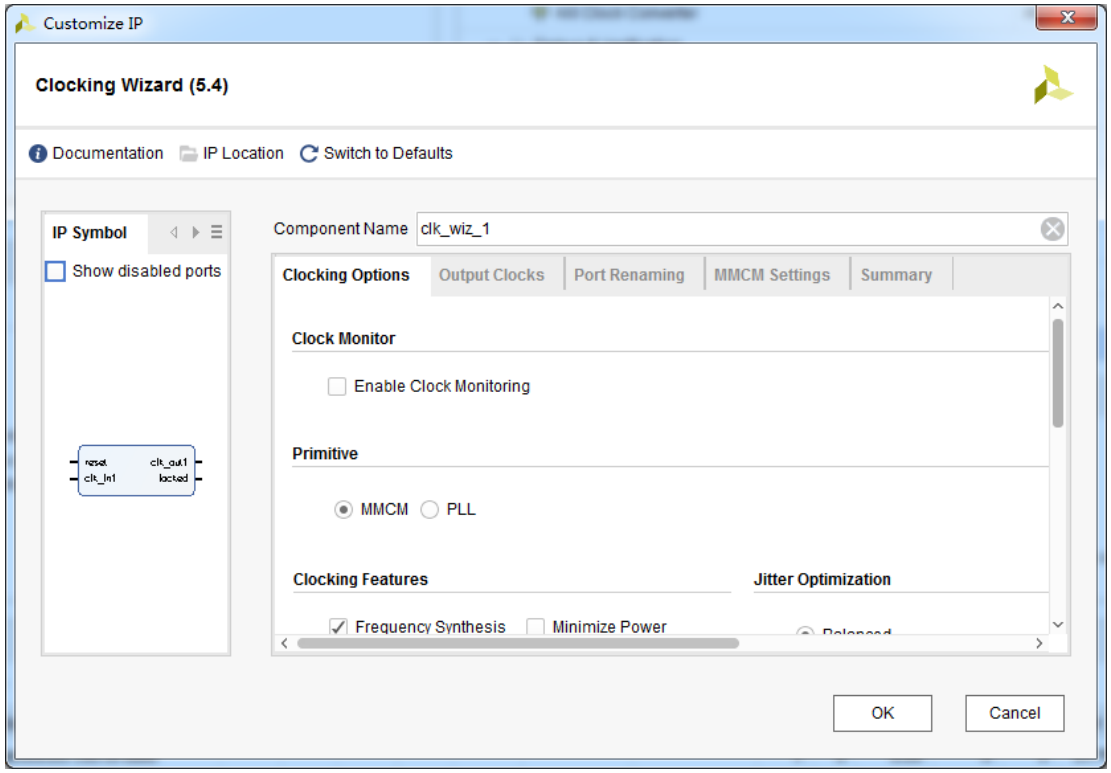


图 11.4.4 “Customize IP”窗口

接下来就是配置 IP 核的时钟参数。最上面的“Component Name”一栏设置该 IP 元件的名称，这里保持默认即可。在第一个“Clocking Options”选项卡中，“Primitive”选项用于选择是使用 MMCM 还是 PLL 来输出不同的时钟，对于我们的本次实验来说，MMCM 和 PLL 都可以完成，这里我们可以保持默认选择 MMCM。需要修改的是最下面的“Input Clock Information”一栏，把“Primary”时钟的输入频率修改为我们开发板的核心

板上的晶振频率 50MHz，其他的设置保持默认即可，如下图所示。

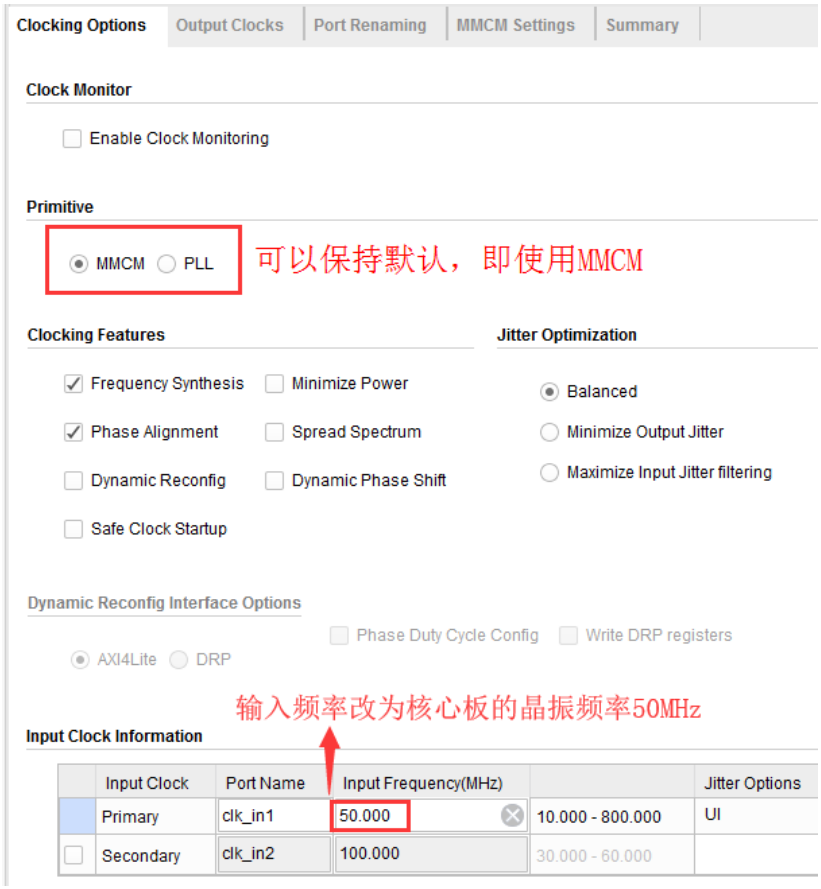


图 11.4.5 “Clocking Options”选项卡的设置

接下来切换至“Output Clocks”选项卡，在“Output Clock”选项卡中，勾选前 4 个时钟，并且将其“Output Freq(MHz)”分别设置为 100、100、50、25，注意，第 2 个 100MHz 时钟的相移“Phase(degrees)”一栏要设置为 180。其他设置保持默认即可，如下图所示。

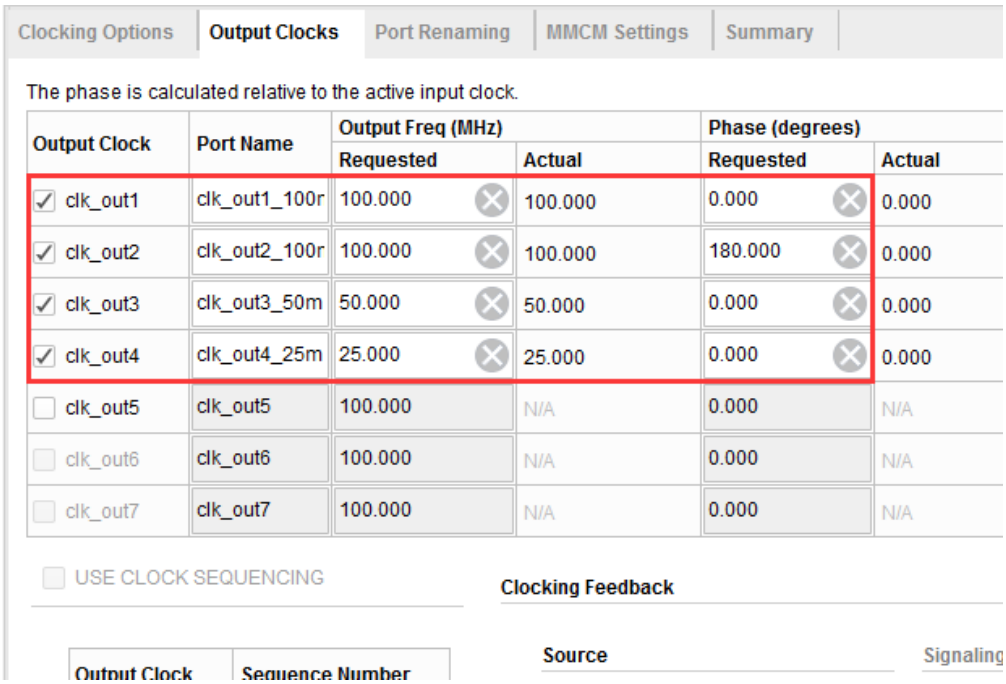


图 11.4.6 “Output Clocks”选项卡的设置

“Port Renaming”选项卡主要是对一些控制信号的重命名。这里我们只用到了锁定指示 locked 信号，其名称保持默认即可，如下图所示。

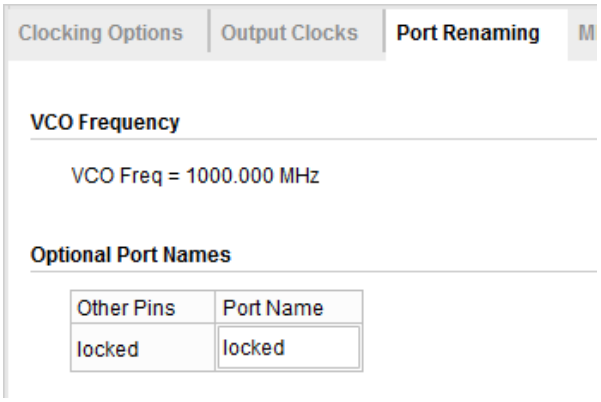


图 11.4.7 “Port Renaming”选项卡的设置

“MMCM Setting”选项卡展示了对整个 MMCM/PLL 的最终配置参数，这些参数都是根据之前用户输入的时钟需求由 Vivado 来自动配置，Vivado 已经对参数进行了最优的配置，在绝大多数情况下都不需要用户对它们进行更改，也不建议更改，所以这一步保持默认即可，如下图所示。

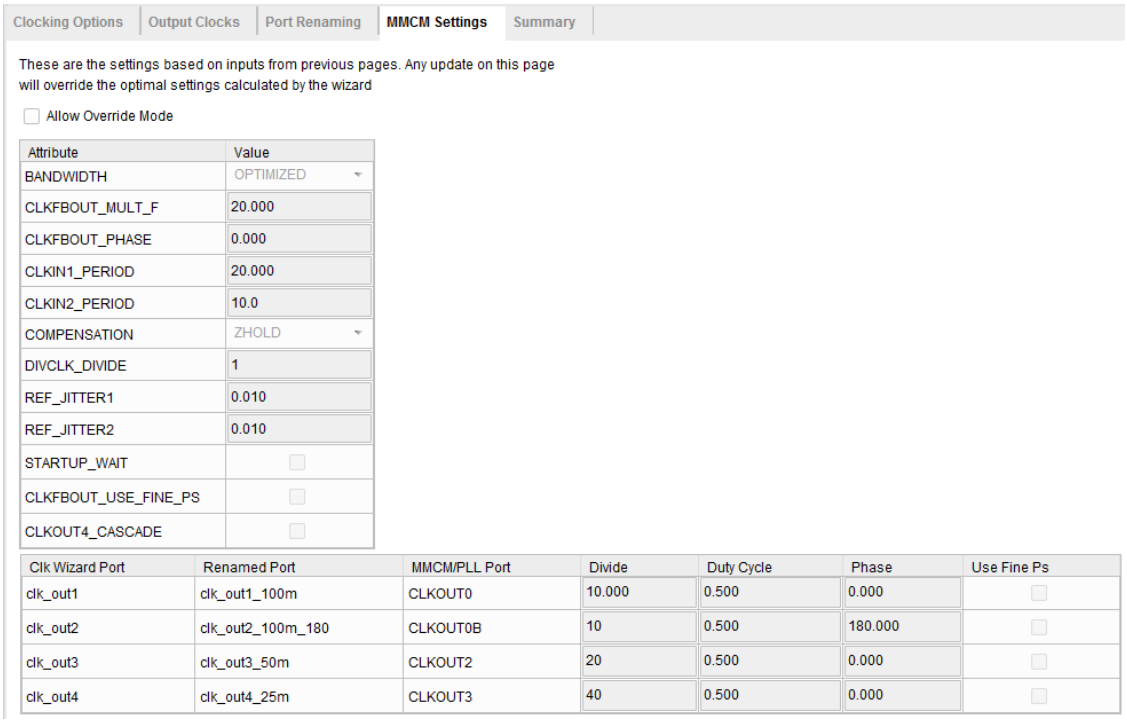


图 11.4.8 “MMCM Setting”选项卡的设置

最后的“Summary”选项卡是对前面所有配置的一个总结，在这里我们直接点击“OK”按钮即可，如下图所示。

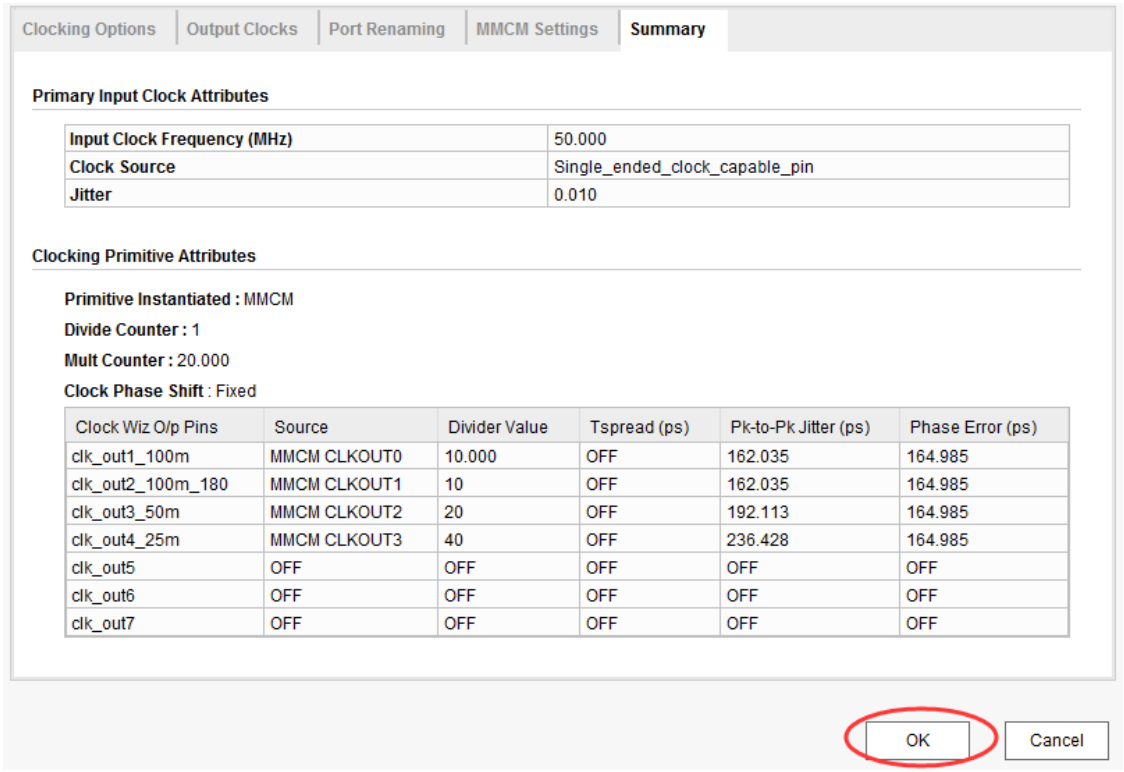


图 11.4.9 “Summary”选项卡

接着就弹出了“Genarate Output Products”窗口，我们直接点击“Generate”即可，如下图所示。

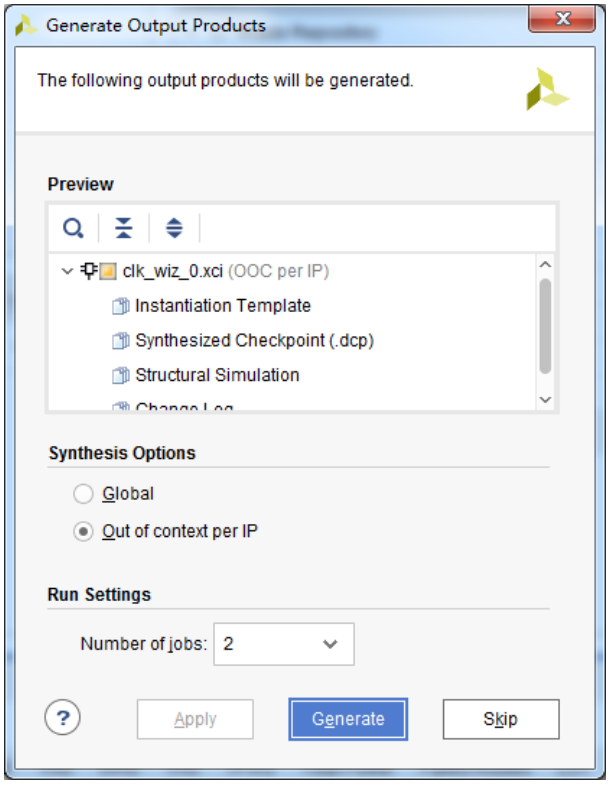


图 11.4.10 “Genarate Output Products”窗口

之后我们就可以在“Design Run”窗口的“Out-of-Context Module Runs”一栏中出现了该 IP 核对应的 run“clk_wiz_0_synth_1”，其综合过程独立于顶层设计的综合，所以在我们可以看到其正在综合，如下图

所示。

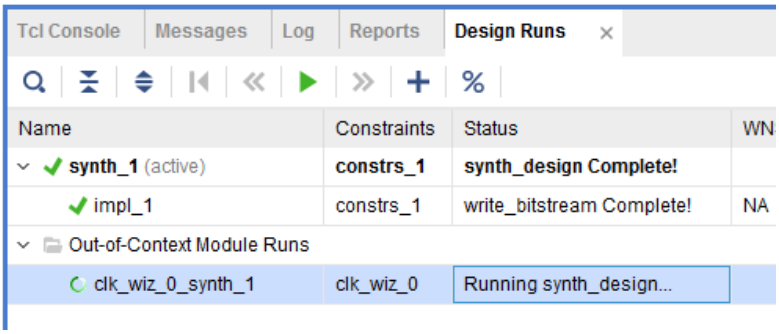


图 11.4.11 “clk_wiz_0_synth_1”run

在其Out-of-Context综合的过程中，我们就可以开始编写代码了。首先打开IP核的例化模板，在“Source”窗口中的“IP Sources”选项卡中，依次用鼠标单击展开“IP”-“clk_wiz_0”-“Instantitation Template”，我们可以看到“clk_wiz.veo”文件，它是由 IP 核自动生成的只读的 verilog 例化模板文件，双击就可以打开它，在例化时钟 IP 核模块的时钟，可以直接从这里拷贝，如下图所示。

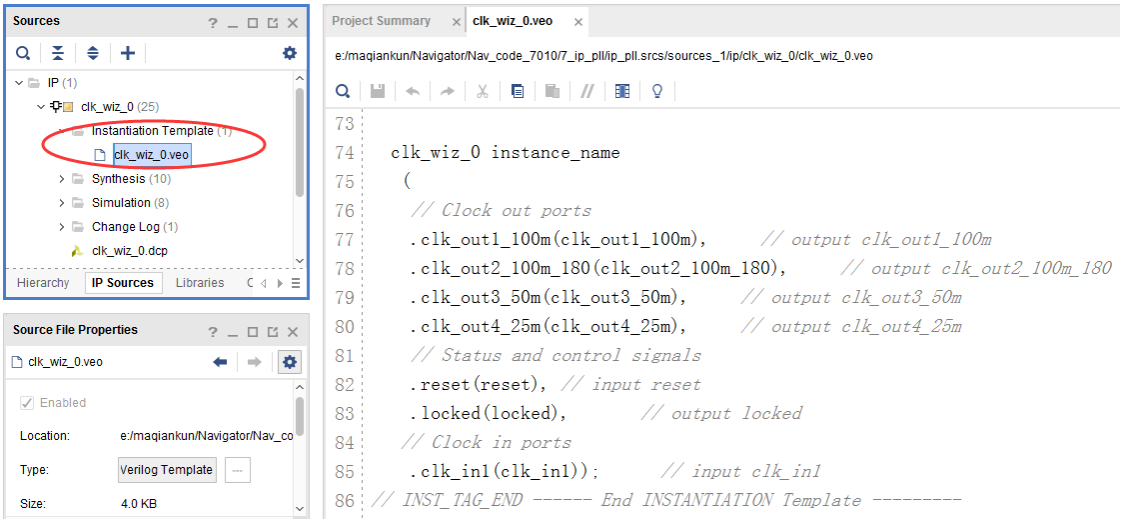


图 11.4.12 “clk_wiz.veo”文件

我们接下来创建一个 verilog 源文件，其名称为 ip_clk_wiz.v，代码如下：

```
module  ip_clk_wiz(
    input               sys_clk       ,  //系统时钟
    input               sys_rst_n     ,  //系统复位，低电平有效
    //输出时钟
    output              clk_100m      ,  //100Mhz 时钟频率
    output              clk_100m_180deg,  //100Mhz 时钟频率,相位偏移 180 度
    output              clk_50m       ,  //50Mhz 时钟频率
    output              clk_25m          //25Mhz 时钟频率
    );

//***************************************************
//**                    main code
//***************************************************
```

```
14
15 wire        locked;
16
17 //MMCM/PLL IP 核的例化
18 clk_wiz_0  clk_wiz_0
19 (
20  // Clock out ports
21  .clk_out1_100m     (clk_100m),          // output clk_out1_100m
22  .clk_out2_100m_180 (clk_100m_180deg),   // output clk_out2_100m_180
23  .clk_out3_50m      (clk_50m),           // output clk_out3_50m
24  .clk_out4_25m      (clk_25m),           // output clk_out4_25m
25  // Status and control signals
26  .reset             (~sys_rst_n),         // input reset
27  .locked            (locked),            // output locked
28  // Clock in ports
29  .clk_in1           (sys_clk)             // input clk_in1
30 );
31
32 endmodule
```

程序中例化了 clk_wiz_0，把 FPGA 的系统时钟 50Mhz 连接到 clk_wiz_0 的 clk_in1，系统复位信号连接到 clk_wiz_0 的 reset，由于时钟 IP 核默认是高电平复位，而输入的系统复位信号 sys_rst_n 是低电平复位，因此要对系统复位信号进行取反。clk_wiz_0 输出的 4 个时钟信号直接连接到顶层端口的四个时钟输出信号。

我们接下来先对代码进行仿真，TestBench 代码如下：

```
1  `timescale 1ns / 1ps
2
3  module tb_ip_clk_wiz();
4
5  reg     sys_clk;
6  reg     sys_rst_n;
7
8  wire    clk_100m;
9  wire    clk_100m_180deg;
10 wire    clk_50m;
11 wire    clk_25m;
12
13 always #10 sys_clk = ~sys_clk;
14
15 initial begin
16     sys_clk = 1'b0;
17     sys_rst_n = 1'b0;
18     #200
```

```
19      sys_rst_n = 1'b1;
20  end
21
22  ip_clk_wiz u_ip_clk_wiz(
23      .sys_clk          (sys_clk        ),
24      .sys_rst_n        (sys_rst_n      ),
25
26      .clk_100m         (clk_100m       ),
27      .clk_100m_180deg  (clk_100m_180deg),
28      .clk_50m          (clk_50m        ),
29      .clk_25m          (clk_25m        )
30      );
31
32  endmodule
```

对模块进行仿真的方法这里不再赘述，仿真后得到的波形如下图所示：

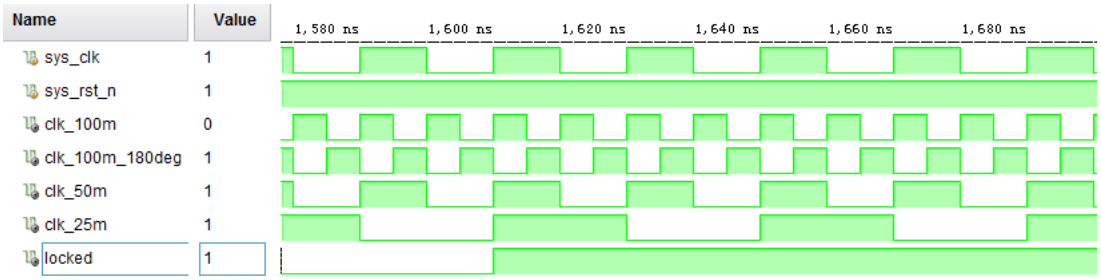


图 11.4.13 Vivado 仿真波形

由上图可知，locked 信号拉高之后，锁相环开始输出 4 个稳定的时钟。clk_100m 和 clk_100m_180deg 周期都为 10ns，即时钟频率都为 100Mhz，但两个时钟相位偏移 180 度，所以这两个时钟刚好反相；clk_50m 周期为 20ns，时钟频率为 50Mhz；clk_25m 周期为 40ns，时钟频率为 25Mhz。也就是说，我们创建的锁相环从仿真结果上来看是正确的。

11.5 下载验证

编译工程并生成比特流.bit 文件后，此时把将下载器一端连接电脑，另一端与开发板上的 JTAG 下载口连接，连接电源线，并打开开发板的电源开关。

点击 Vivado 左侧“Flow Navigator”窗口最下面的“Open Hardware Manager”，如果此时 Vivado 软件识别到下载器，则点击“Hardware”窗口中“Progam Device”下载程序，在弹出的界面中选择“Program”下载程序。

程序下载完成后，接下来我们使用示波器测量开发板 J3 扩展口的第 29、31、33、35 号脚。示波器测试依次为 B19（100MHz）、C20（100MHz_180）、P19（50MHz）和 N18（25MHz）。如下图所示：

图 11.5.1 示波器测量引脚

此时在示波器上就可以观察到时钟的波形图。下图为使用示波器测量扩展口第 35 号脚（N18）所显示的波形。

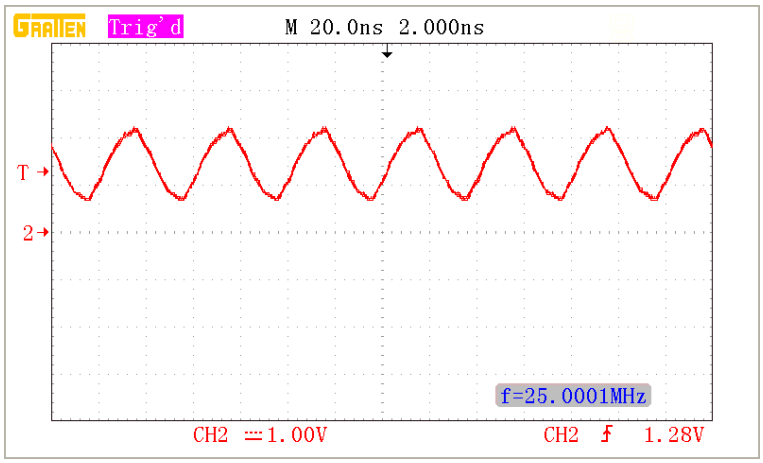


图 11.5.2 扩展口 N18 脚（25MHz）输出的波形

由上图可知，示波器测量出的时钟频率为 25Mhz，跟仿真结果是一样的，其它三个扩展口输出的时钟大家可以测试一下，这里不再贴出其它扩展口的波形图。

第十二章 IP 核之 RAM 实验

RAM 的英文全称是 Random Access Memory，即随机存取存储器，它可以随时把数据写入任一指定地址的存储单元，也可以随时从任一指定地址中读出数据，其读写速度是由时钟频率决定的。RAM 主要用来存放程序及程序执行过程中产生的中间数据、运算结果等。本章我们将对 Vivado 软件生成的 RAM IP 核进行读写测试，并向大家介绍 Xilinx RAM IP 核的使用方法。

本章包括以下几个部分：

12.1 RAM IP 核简介

12.2 实验任务

12.3 硬件设计

12.4 程序设计

12.5 下载验证

12.1 RAM IP 核简介

Xilinx 7 系列器件具有嵌入式存储器结构，满足了设计对片上存储器的需求。嵌入式存储器结构由一列列 BRAM（块 RAM）存储器模块组成，通过对这些 BRAM 存储器模块进行配置，可以实现各种存储器的功能，例如：RAM、移位寄存器、ROM 以及 FIFO 缓冲器。

Vivado 软件自带了 BMG IP 核（Block Memory Generator，块 RAM 生成器），可以配置成 RAM 或者 ROM。这两者的区别是 RAM 是一种随机存取存储器，不仅仅可以存储数据，同时支持对存储的数据进行修改；而 ROM 是一种只读存储器，也就是说，在正常工作时只能读出数据，而不能写入数据。需要注意的是，配置成 RAM 或者 ROM 使用的资源都是 FPGA 内部的 BRAM，只不过配置成 ROM 时只用到了嵌入式 BRAM 的读数据端口。本章我们主要介绍通过 BRAM IP 核配置成 RAM 的使用方法。

Xilinx 7 系列器件内部的 BRAM 全部是真双端口 RAM（True Dual-Port ram，TDP），这两个端口都可以独立地对 BRAM 进行读/写。但也可以被配置成伪双端口 RAM（Simple Dual-Port ram，SDP）（有两个端口，但是其中一个只能读，另一个只能写）或单端口 RAM（只有一个端口，读/写只能通过这一个端口来进行）。单端口 RAM 只有一组数据总线、地址总线、时钟信号以及其他控制信号，而双端口 RAM 具有两组数据总线、地址总线、时钟信号以及其他控制信号。有关 BRAM 的更详细的介绍，请读者参阅 Xilinx 官方的手册文档“UG473，7 Series FPGAs Memory Resources User Guide”。

单端口 RAM 类型和双端口 RAM 类型在操作上都是一样的，我们只要学会了单端口 RAM 的使用，那么学习双端口 RAM 的读写操作也是非常容易的。本章我们以配置成单端口 RAM 为例进行讲解。

BMG IP 核配置成单端口 RAM 的框图如下图所示。

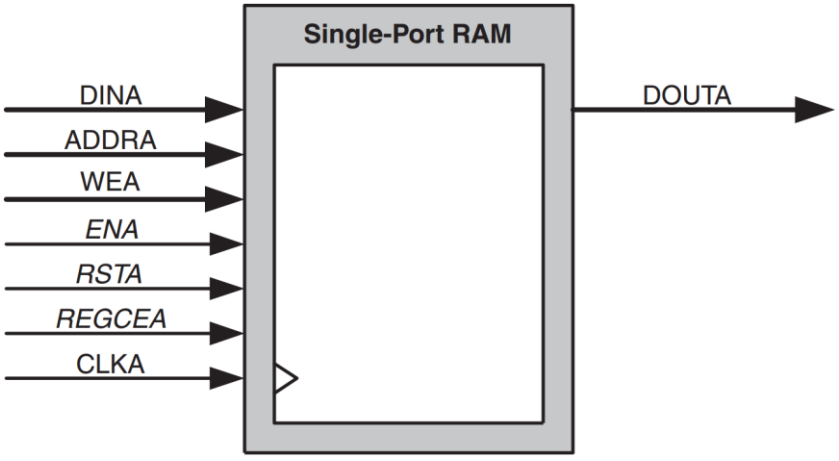


图 12.1.1 单端口 RAM 框图

各个端口的功能描述如下：

DINA：RAM 端口 A 写数据信号。

ADDRA：RAM 端口 A 读写地址信号，对于单端口 RAM 来说，读地址和写地址共用同该地址线。

WEA：RAM 端口 A 写使能信号，高电平表示向 RAM 中写入数据，低电平表示从 RAM 中读出数据。

ENA：端口 A 的使能信号，高电平表示使能端口 A，低电平表示端口 A 被禁止，禁止后端口 A 上的读写操作都会变成无效。另外 ENA 信号是可选的，当取消该使能信号后，RAM 会一直处于有效状态。

RSTA：RAM 端口 A 复位信号，可配置成高电平或者低电平复位，该复位信号是一个可选信号。

REGCEA：RAM 端口 A 输出寄存器使能信号，当 REGCEA 为高电平时，DOUTA 保持最后一次输出的数据，REGCEA 同样是一个可选信号。

CLKA：RAM 端口 A 的时钟信号。

DOUTA：RAM 端口 A 读出的数据。

12.2 实验任务

本节实验任务是使用 Xilinx BMG IP 核，配置成一个单端口的 RAM，然后对 RAM 进行读写操作，通过在 Vivado 自带的仿真器中观察波形是否正确，最后将设计下载到启明星 Zynq 开发板中，并使用 ILA 对其进行在线调试观察。

12.3 硬件设计

本章实验只用到了输入的时钟信号和按键复位信号，没有用到其它硬件外设，各端口信号的管脚分配如下表所示：

表 12.3.1 IP核之RAM实验管脚分配

信号名	方向	管脚	端口说明	电平标准
sys_clk	input	U18	系统时钟，50Mhz	LVCMOS33
sys_rst_n	input	J15	系统复位，低电平有效，位于底板上	LVCMOS33

对应的 XDC 约束语句如下所示：

```
set_property -dict {PACKAGE_PIN U18 IOSTANDARD LVCMOS33} [get_ports sys_clk]
set_property -dict {PACKAGE_PIN J15 IOSTANDARD LVCMOS33} [get_ports sys_rst_n]
```

12.4 程序设计

首先在 Vivado 软件中创建一个名为 ip_ram 的工程，工程创建完成后，在 Vivado 软件的左侧“Flow Navigator”栏中单击“IP Catalog”，如下图所示。

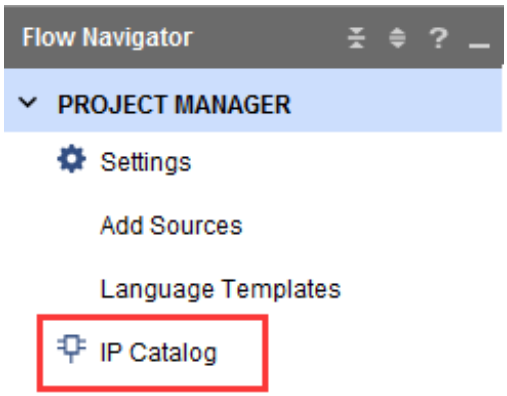


图 12.4.1 点击“IP Catalog”

在“IP Catalog”窗口的搜索框中输入“Block Memory”，出现唯一匹配的“Block Memory Generator”，如下图所示（图中出现的两个 IP 核为同一个 BMG IP 核）。

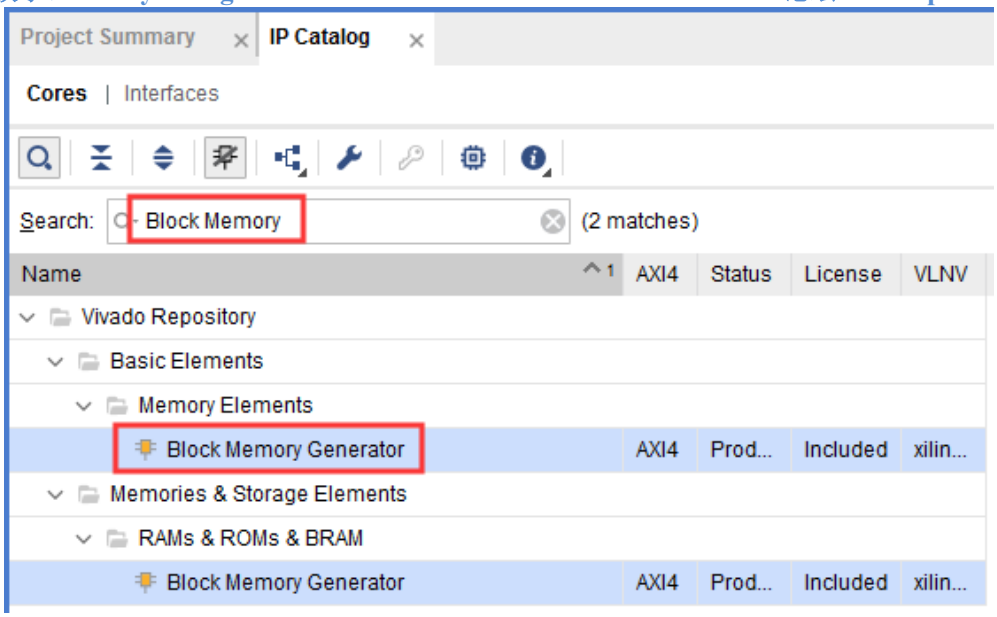


图 12.4.2 搜索框中输入“Block Memory”

双击“Block Memory Generator”后弹出 IP 核的配置界面，接下来对 BMG IP 核进行配置，“Basic”选项页配置界面如下图所示。

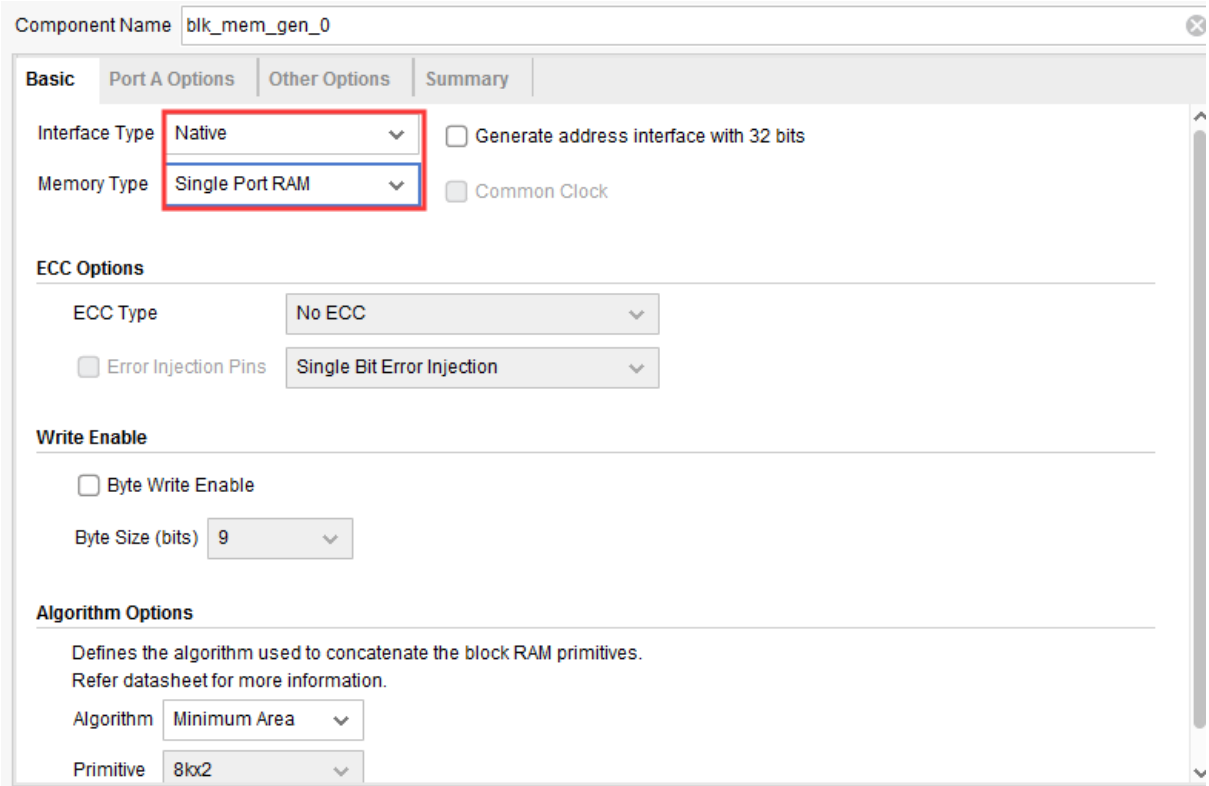


图 12.4.3 “Basic”选项页配置

Component Name：设置该 IP 核的名称，这里保持默认即可。

Interface Type：RAM 接口总线。这里保持默认，选择 Native 接口类型（标准 RAM 接口总线）；

Memory Type：存储器类型。可配置成 Single Port RAM（单端口 RAM）、Simple Dual Port RAM（伪双端口 RAM）、True Dual Port RAM（真双端口 RAM）、Single Port ROM（单端口 ROM）和 Dual Port ROM

（双端口 ROM），这里选择 Single Port RAM，即配置成单端口 RAM。

ECC Options：Error Correction Capability，纠错能力选项，单端口 RAM 不支持 ECC。

Write Enable：字节写使能选项，勾中后可以单独将数据的某个字节写入 RAM 中，这里不使能。

Algorithm Options：算法选项。可选择 Minimum Area（最小面积）、Low Power（低功耗）和 Fixed Primitives（固定的原语），这里选择默认的 Minimum Area。

接下来切换至“Port A”选项页，设置端口 A 的参数，该页面配置如下：

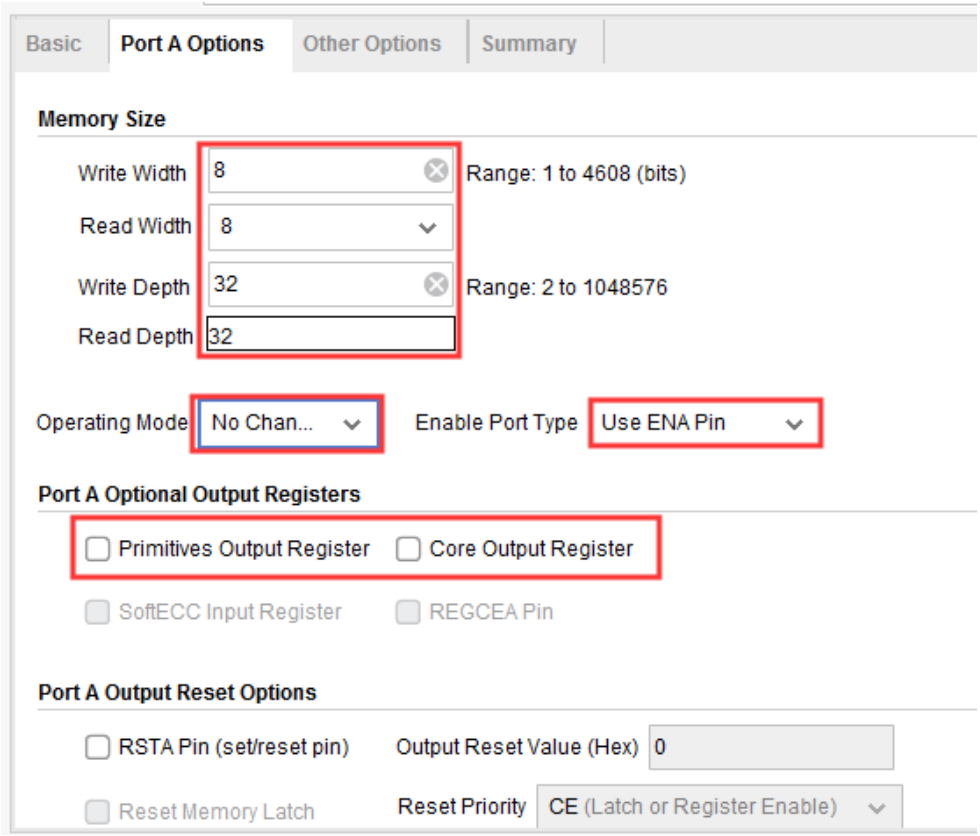


图 12.4.4 “Port A Options”选项页配置

Write Width：端口 A 写数据位宽，单位 Bit，这里设置成 8。

Read Width：端口 A 读数据位宽，一般和写数据位宽保持一致，设置成 8。

Write Depth：写深度，这里设置成 32，即 RAM 所能访问的地址范围为 0-31。

Read Depth：读深度，默认和写深度保持一致。

Operating Mode：RAM 读写操作模式。共分为三种模式，分别是 Write First（写优先模式）、Read First（读优先模式）和 No Change（不变模式）。写优先模式指数据先写入 RAM 中，然后在下一个时钟输出该数据；读优先模式指数据先写入 RAM 中，同时输出 RAM 中同地址的上一次数据；不变模式指读写分开操作，不能同时进行读写，这里选择 No Change 模式。

Enable Port Type：使能端口类型。Use ENA pin（添加使能端口 A 信号）；Always Enabled（取消使能信号，端口 A 一直处于使能状态），这里选择默认的 Use ENA pin。

Port A Optional Output Register：端口 A 输出寄存器选项。其中“Primitives Output Register”默认是选中状态，作用是打开 BRAM 内部位于输出数据总线之后的输出流水线寄存器，虽然在一般设计中为了改善时序性能会保持此选项的默认勾选状态，但是这会使得 BRAM 输出的数据延迟一拍，这不利于我们在 Vivado 的 ILA 调试窗口中直观清晰地观察信号；而且在本实验中我们仅仅是把 BRAM 的数据输出总线连接到了 ILA

的探针端口上来进行观察，除此之外数据输出总线没有别的负载，不会带来难以满足的时序路径，因此这里取消勾选。

Port A Output Reset Options：RAM 复位信号选项，这里不添加复位信号，保持默认即可。

另外，需要注意的是，下面的“Primitives Output Register”默认是选中状态的，此选项的作用是打开块 RAM 内部的位于输出数据总线之后的输出流水线寄存器，虽然在一般设计中为了改善时序性能会保持此选项的默认勾选状态，但是这会使得块 RAM 输出的数据延迟一拍，这不利于我们在 Vivado 的 ILA 调试窗口中直观清晰地观察信号；而且在本实验中我们仅仅是把块 RAM 的数据输出总线连接到了 ILA 的探针端口上来进行观察，除此之外数据输出总线没有别的负载，不会带来难以满足的时序路径。

接下来的“Other Options”选项页用于设置 RAM 的初始值等，本次实验不需要设置，直接保持默认即可。

最后一个是“Summary”选项页，该页面显示了存储器的类型，消耗的 BRAM 资源等，我们直接点击“OK”按钮完成 BMG IP 核的配置，如下图所示：

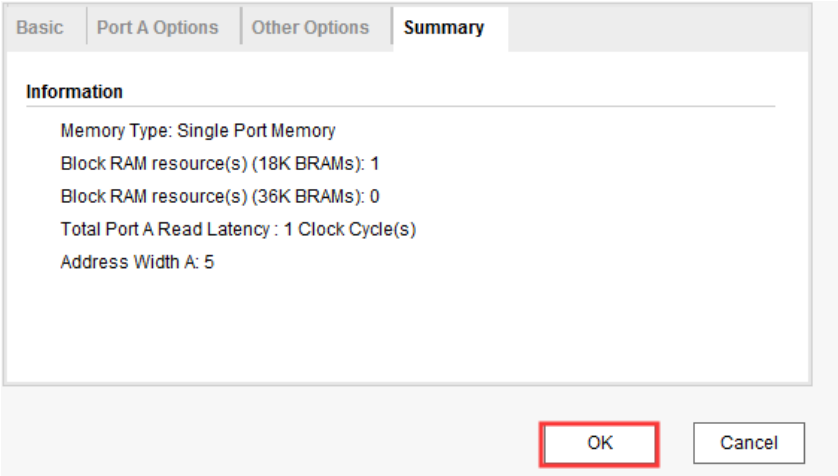


图 12.4.5 “Summary”选项页

接下来会弹出询问是否在工程目录下创建存放 IP 核的文件，我们点击“OK”按钮即可。

紧接着会弹出“Genarate Output Products”窗口，我们直接点击“Generate”，如下图所示。

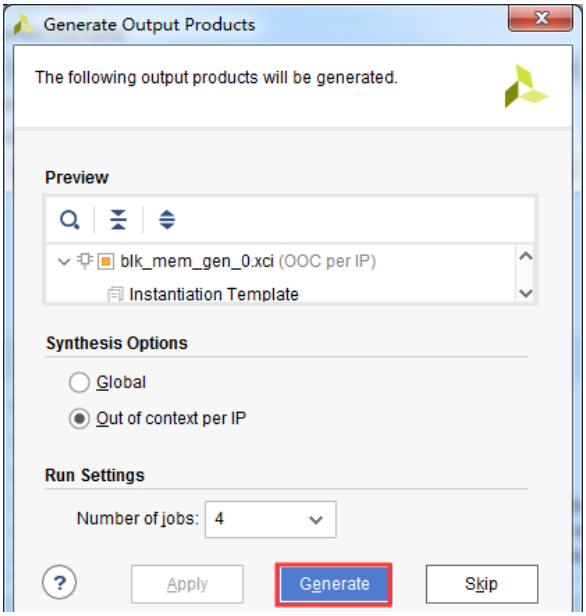


图 12.4.6 “Genarate Output Products”窗口

之后我们就可以在“Design Run”窗口的“Out-of-Context Module Runs”一栏中出现了该 IP 核对应的 run“blk_mem_gen_0_synth_1”，其综合过程独立于顶层设计的综合，所以在我们可以看到其正在综合，如下图所示。

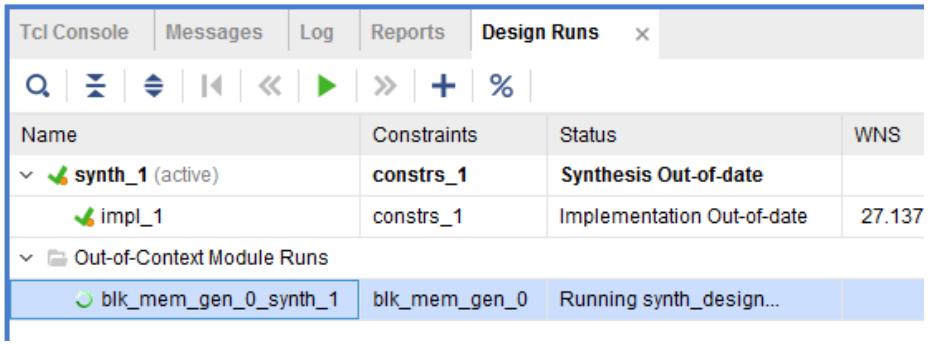


图 12.4.10 “blk_mem_gen_0_synth_1” run

在其 Out-of-Context 综合的过程中，我们就可以进行 RTL 编码了。首先打开 IP 核的例化模板，在“Source”窗口中的“IP Sources”选项卡中，依次用鼠标单击展开“IP”-“blk_mem_gen_0”-“Instantitation Template”，我们可以看到“blk_mem_gen_0.veo”文件，它是由 IP 核自动生成的只读的 verilog 例化模板文件，双击就可以打开它，如下图所示。

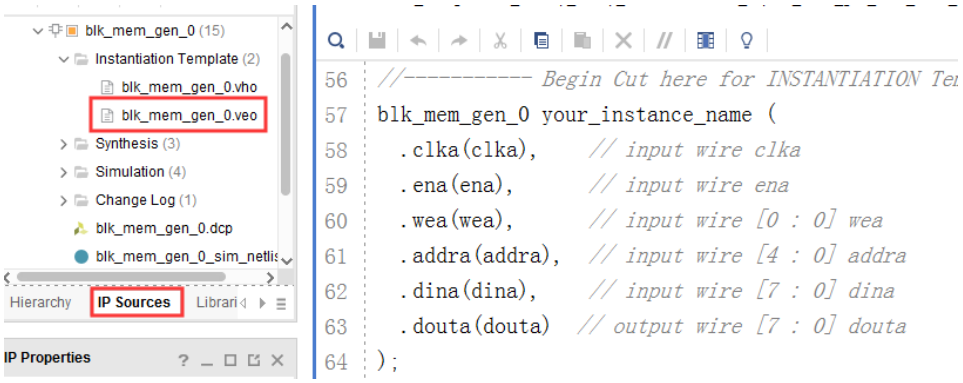


图 12.4.11 “blk_mem_gen_0.veo”文件

接下来我们创建一个新的设计文件，命名为 ram_rw.v，代码如下：

```
1  module ram_rw(
2      input              clk       ,  //时钟信号
3      input              rst_n     ,  //复位信号，低电平有效
4
5      output             ram_en    ,  //ram 使能信号
6      output             ram_wea   ,  //ram 读写选择
7      output  reg  [4:0] ram_addr  ,  //ram 读写地址
8      output  reg  [7:0] ram_wr_data,  //ram 写数据
9      input        [7:0] ram_rd_data   //ram 读数据
10     );
11
12 //reg define
13 reg    [5:0]  rw_cnt ;               //读写控制计数器
14
```

```
15 //****************************************************
16 //**                    main code
17 //****************************************************
18
19 //控制 RAM 使能信号
20 assign ram_en = rst_n;
21 //rw_cnt 计数范围在 0~31,写入数据;32~63 时,读出数据
22 assign ram_wea = (rw_cnt <= 6'd31 && ram_en == 1'b1) ? 1'b1 : 1'b0;
23
24 //读写控制计数器,计数器范围 0~63
25 always @(posedge clk or negedge rst_n) begin
26     if(rst_n == 1'b0)
27         rw_cnt <= 1'b0;
28     else if(rw_cnt == 6'd63)
29         rw_cnt <= 1'b0;
30     else
31         rw_cnt <= rw_cnt + 1'b1;
32 end
33
34 //产生 RAM 写数据
35 always @(posedge clk or negedge rst_n) begin
36     if(rst_n == 1'b0)
37         ram_wr_data <= 1'b0;
38     else if(rw_cnt <= 6'd31)  //在计数器的 0-31 范围内，RAM 写地址累加
39         ram_wr_data <= ram_wr_data + 1'b1;
40     else
41         ram_wr_data <= 1'b0 ;
42 end
43
44 //读写地址信号 范围：0~31
45 always @(posedge clk or negedge rst_n) begin
46     if(rst_n == 1'b0)
47         ram_addr <= 1'b0;
48     else if(ram_addr == 5'd31)
49         ram_addr <= 1'b0;
50     else
51         ram_addr <= ram_addr + 1'b1;
52 end
53
63 endmodule
```

模块中定义了一个读写控制计数器（rw_cnt），当计数范围在 0~31 之间时，向 ram 中写入数据；当计

数范围在 32~63 之间时，从 ram 中读出数据。

接下来我们设计一个 verilog 文件来实例化创建的 RAM IP 核以及 ram_rw 模块，文件名为 ip_ram.v，编写的 verilog 代码如下。

```
1  module ip_ram(
2      input          sys_clk ,  //系统时钟
3      input          sys_rst_n  //系统复位，低电平有效
4      );
5
6  //wire define
7  wire             ram_en      ;  //RAM 使能
8  wire             ram_wea     ;  //ram 读写使能信号,高电平写入,低电平读出
9  wire    [4:0]    ram_addr    ;  //ram 读写地址
10 wire    [7:0]    ram_wr_data ;  //ram 写数据
11 wire    [7:0]    ram_rd_data ;  //ram 读数据
12
13 //*****************************************************
14 //**                    main code
15 //*****************************************************
16
17 //ram 读写模块
18 ram_rw  u_ram_rw(
19     .clk            (sys_clk     ),
20     .rst_n          (sys_rst_n   ),
21  //RAM
22  .ram_en         (ram_en      ),
23     .ram_wea        (ram_wea     ),
24     .ram_addr       (ram_addr    ),
25     .ram_wr_data    (ram_wr_data ),
26     .ram_rd_data    (ram_rd_data )
27     );
28
29 //ram ip 核
30 blk_mem_gen_0  blk_mem_gen_0 (
31  .clka  (sys_clk      ),  // input wire clka
32  .ena   (ram_en       ),  // input wire ena
33  .wea   (ram_wea      ),  // input wire [0 : 0] wea
34  .addra (ram_addr     ),  // input wire [4 : 0] addra
35  .dina  (ram_wr_data  ),  // input wire [7 : 0] dina
36  .douta (ram_rd_data  )  // output wire [7 : 0] douta
37 );
38
```

```
39 endmodule
```

程序中例化了 ram_rw 模块和 ram IP 核 blk_mem_gen_0，其中 ram_rw 模块负责产生对 ram IP 核读/写所需的所有数据、地址以和读写使能信号，同时从 ram IP 读出的数据也连接至 ram_rw 模块。

接下来对 RAM IP 核进行仿真，来验证对 RAM 的读写操作是否正确。tb_ip_ram 仿真文件源代码如下：

```
1  `timescale 1ns / 1ps
2
3  module tb_ip_ram();
4
5  reg     sys_clk;
6  reg     sys_rst_n;
7
8  always #10 sys_clk = ~sys_clk;
9
10 initial begin
11     sys_clk = 1'b0;
12     sys_rst_n = 1'b0;
13     #200
14     sys_rst_n = 1'b1;
15 end
16
17 ip_ram u_ip_ram(
18     .sys_clk          (sys_clk        ),
19     .sys_rst_n        (sys_rst_n      )
20     );
21
22 endmodule
```

接下来就可以开始仿真了，仿真过程这里不再赘述，仿真波形图如下图所示。

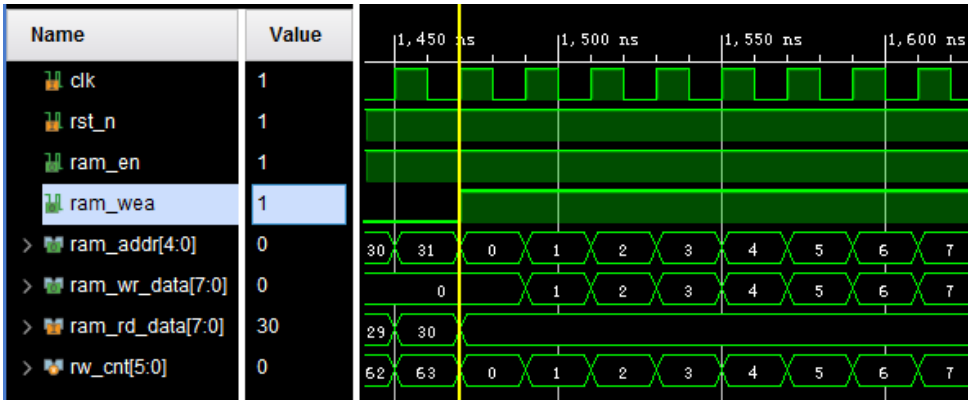


图 12.4.7 RAM 写操作波形图

图 12.4.7 为 RAM 的写操作仿真波形图，由上图可知，ram_wea 信号拉高，说明此时是对 ram 进行写操作。ram_wea 信号拉高之后，地址和数据都是从 0 开始累加，也就说当 ram 地址为 0 时，写入的数据也是 0；当 ram 地址为 1 时，写入的数据也是 1，我们总共向 ram 中写入 32 个数据。

RAM 读操作仿真波形图如下图所示：

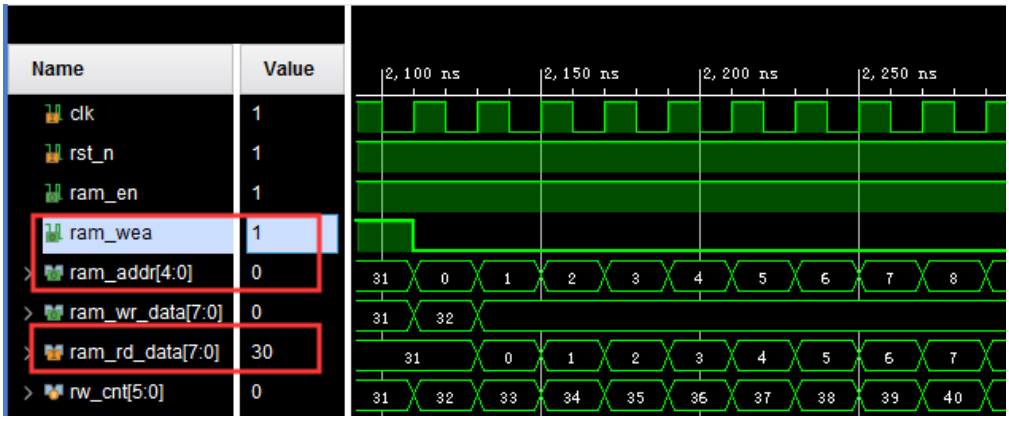


图 12.4.8 RAM 读操作波形图

由上图可知，ram_wea 信号拉低，说明此时是对 ram 进行读操作。ram_wea 信号拉低之后，ram_addr 从 0 开始增加，也就是说从 ram 的地址 0 开始读数据；ram 中读出的数据 ram_rd_data 在延时一个时钟周期之后，开始输出数据，输出的数据为 0，1，2……，和我们写入的值是相等的， 也就是说，我们创建的 RAM IP 核从仿真结果上来看是正确的。

接下来添加 ILA IP 核，将 ram_en、ram_wea、ram_addr、ram_wr_data 和 ram_rd_data 信号添加至观察列表中，添加 ILA IP 核的方法这里不再赘述。

最后为工程添加 IO 管脚约束，对应的 XDC 约束语句如下所示：

```
set_property -dict {PACKAGE_PIN U18 IOSTANDARD LVCMOS33} [get_ports sys_clk]
set_property -dict {PACKAGE_PIN J15 IOSTANDARD LVCMOS33} [get_ports sys_rst_n]
```

12.5 下载验证

编译工程并生成比特流.bit 文件。将下载器一端连接电脑，另一端与开发板上的 JTAG 下载口连接，连接电源线，并打开开发板的电源开关。

点击 Vivado 左侧“Flow Navigator”窗口最下面的“Open Hardware Manager”，此时 Vivado 软件识别到下载器，点击“Hardware”窗口中“Progam Device”下载程序，在弹出的界面中选择“Program”下载程序。

RAM 写操作在 ILA 中观察的波形如下图所示：

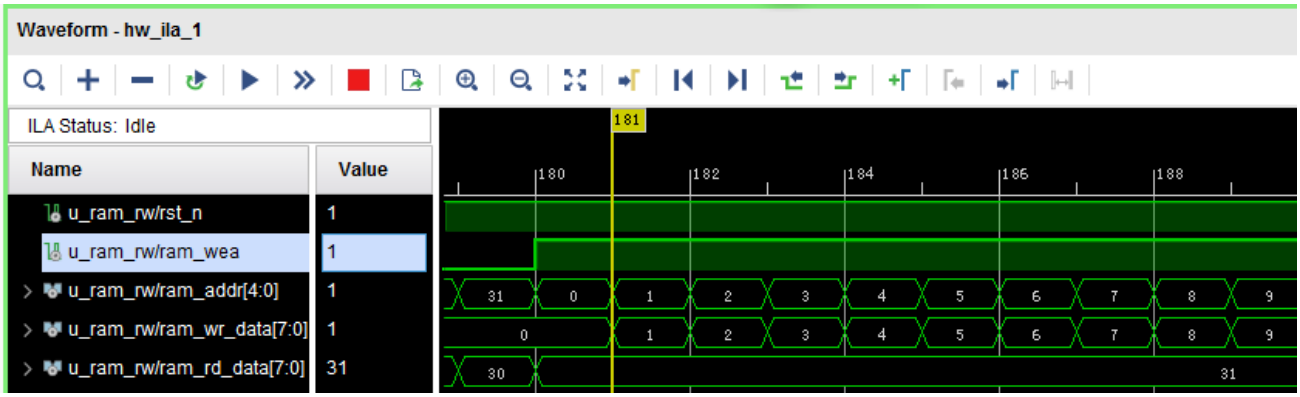


图 12.5.1 RAM 写操作 ILA 波形图

ram_wea 信号拉高之后，地址和数据都是从 0 开始累加，也就说当 ram 地址为 0 时，写入的数据也是 0；当 ram 地址为 1 时，写入的数据也是 1。我们可以发现，上图中的数据变化和在 Vivado 仿真的波形是一致

的。

RAM 读操作在 ILA 中观察的波形如下图所示：

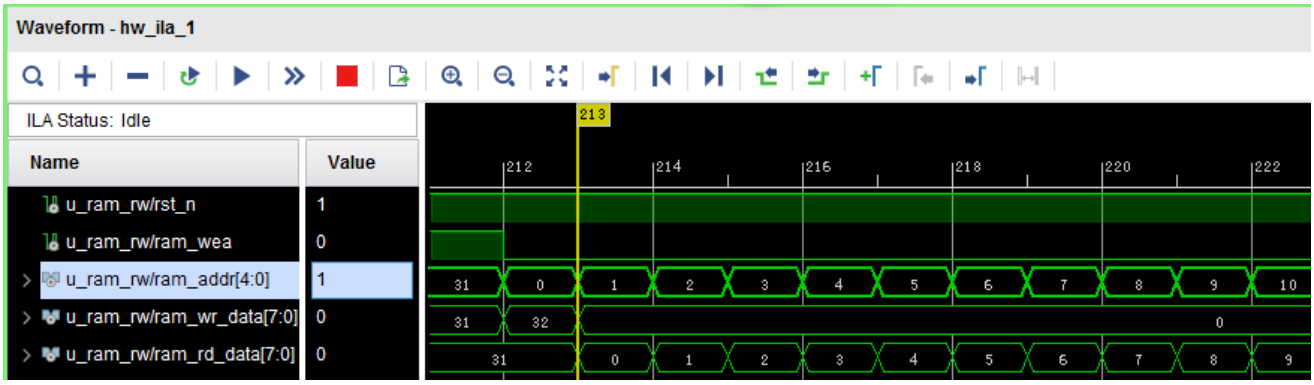


图 12.5.2 RAM 读操作 ILA 波形图

ram_wea（读使能）信号拉低之后，ram_addr 从 0 开始增加，也就是说从 ram 的地址 0 开始读数据；ram 中读出的数据 ram_rd_data 在延时一个时钟周期之后，开始输出数据，输出的数据为 0，1，2……，和我们写入的值是相等的。我们可以发现，上图中的数据变化同样和 Vivado 仿真的波形是一致的。本次实验的 IP 核之 RAM 读写实验验证成功。

第十三章 IP 核之 FIFO 实验

FIFO 的英文全称是 First In First Out，即先进先出。FPGA 使用的 FIFO 一般指的是对数据的存储具有先进先出特性的一个缓存器，常被用于数据的缓存,或者高速异步数据的交互也即所谓的跨时钟域信号传递。它与 FPGA 内部的 RAM 和 ROM 的区别是没有外部读写地址线，采取顺序写入数据，顺序读出数据的方式，使用起来简单方便，由此带来的缺点就是不能像 RAM 和 ROM 那样可以由地址线决定读取或写入某个指定的地址。本章我们将对 Vivado 软件生成的 FIFO IP 核进行读写测试，来向大家介绍 Xilinx FIFO IP 核的使用方法。

本章包括以下几个部分：

13.1 FIFO IP 核简介

13.2 实验任务

13.3 硬件设计

13.4 程序设计

13.5 下载验证

13.1 FIFO IP 核简介

根据 FIFO 工作的时钟域，可以将 FIFO 分为同步 FIFO 和异步 FIFO。同步 FIFO 是指读时钟和写时钟为同一个时钟，在时钟沿来临时同时发生读写操作。异步 FIFO 是指读写时钟不一致，读写时钟是互相独立的。Xilinx 的 FIFO IP 核可以被配置为同步 FIFO 或异步 FIFO，其信号框图如下图所示。从图中可以了解到，当被配置为同步 FIFO 时，只使用 wr_clk,所有的输入输出信号都同步于 wr_clk 信号。而当被配置为异步 FIFO 时，写端口和读端口分别有独立的时钟，所有与写相关的信号都是同步于写时钟 wr_clk，所有与读相关的信号都是同步于读时钟 rd_clk。

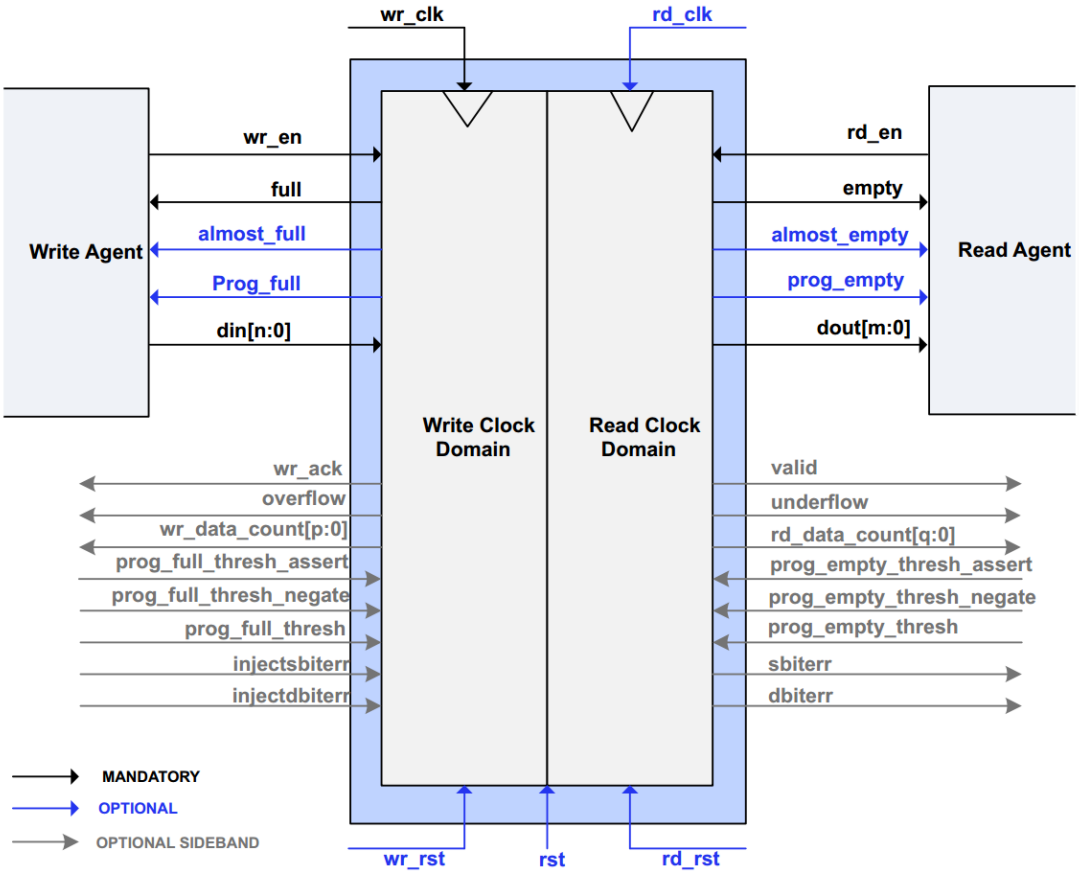


Figure 1-1: **Native Interface FIFOs Signal Diagram**

图 13.1.1 Xilinx 的 FIFO IP 核的信号框图

对于 FIFO 需要了解一些常见参数：

FIFO 的宽度：FIFO 一次读写操作的数据位 N。

FIFO 的深度：FIFO 可以存储多少个宽度为 N 位的数据。

将空标志：almost_empty。FIFO 即将被读空。

空标志：empty。FIFO 已空时由 FIFO 的状态电路送出的一个信号，以阻止 FIFO 的读操作继续从 FIFO 中读出数据而造成无效数据的读出。

将满标志：almost_full。FIFO 即将被写满。

满标志：full。FIFO 已满或将要写满时由 FIFO 的状态电路送出的一个信号，以阻止 FIFO 的写操作继续向 FIFO 中写数据而造成溢出。

读时钟：读 FIFO 时所遵循的时钟，在每个时钟的上升沿触发。

写时钟：写 FIFO 时所遵循的时钟，在每个时钟的上升沿触发。

这里请注意，“almost_empty”和“almost_full”这两个信号分别被看作“empty”和“full”的警告信号，他们相对于真正的空（empty）和满（full）都会提前一个时钟周期拉高。

对于 FIFO 的基本知识先了解这些就足够了，可能有人会好奇为什么会有同步 FIFO 和异步 FIFO，它们各自的用途是什么。之所以有同步 FIFO 和异步 FIFO 是因为各自的作用不同。同步 FIFO 常用于同步时钟的数据缓存，异步 FIFO 常用于跨时钟域的数据信号的传递，例如时钟域 A 下的数据 data1 传递给异步时钟域 B，当 data1 为连续变化信号时，如果直接传递给时钟域 B 则可能会导致收非所送的情况，即在采集过程中会出现包括亚稳态问题在内的一系列问题，使用异步 FIFO 能够将不同时钟域中的数据同步到所需的时钟域中。

13.2 实验任务

本节的实验任务是使用 Vivado 生成 FIFO IP 核，并实现以下功能：当 FIFO 为空时，向 FIFO 中写入数据，写入的数据量和 FIFO 深度一致，即 FIFO 被写满；然后从 FIFO 中读出数据，直到 FIFO 被读空为止，以此向大家详细介绍一下 FIFO IP 核的使用方法。

13.3 硬件设计

本章实验只用到了输入的时钟信号和按键复位信号，没有用到其它硬件外设。

本实验中，各端口信号的管脚分配如下表所示。

表 13.3.1 IP实验管脚分配

信号名	方向	管脚	端口说明	电平标准
sys_clk	input	U18	系统时钟，50Mhz	LVCMOS33
sys_rst_n	input	J15	系统复位，低有效	LVCMOS33

对应的 XDC 约束语句如下所示：

```
create_clock -period 20.000 -name clk [get_ports sys_clk]
set_property -dict {PACKAGE_PIN U18 IOSTANDARD LVCMOS33} [get_ports sys_clk]
set_property -dict {PACKAGE_PIN J15 IOSTANDARD LVCMOS33} [get_ports sys_rst_n]
```

13.4 程序设计

根据实验任务要求和模块化设计的思想，我们需要如下 4 个模块：fifo IP 核、写 fifo 模块、读 fifo 模块以及顶层例化模块实现前三个模块的信号交互。由于 FIFO 多用于跨时钟域信号的处理，所以本实验我们使用异步 FIFO 来向大家详细介绍双时钟 FIFO IP 核的创建和使用。为了方便大家理解，这里我们将读/写时钟都用系统时钟来驱动。系统的功能框图如下图所示：

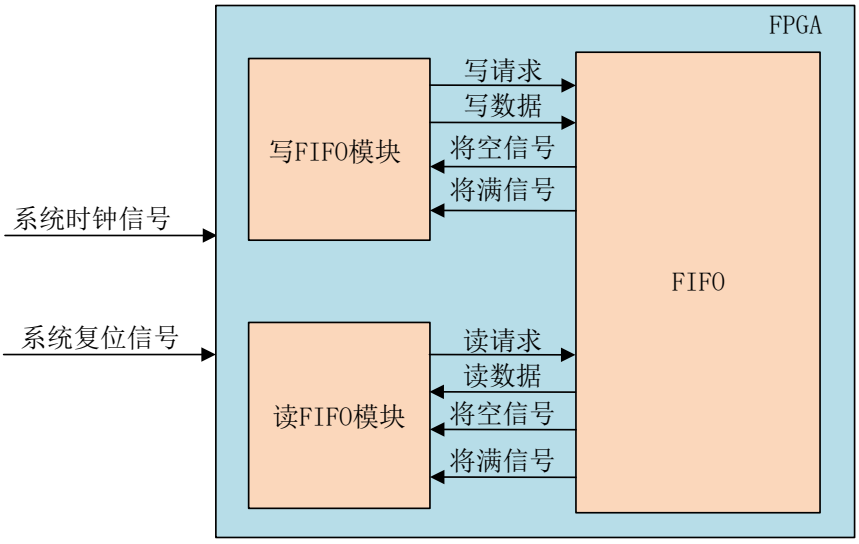


图 13.4.1 系统框图

首先创建一个名为 ip_fifo 的工程，接下来我们创建 fifo IP 核。在 Vivado 软件的左侧“Flow Navigator”栏中单击“IP Catalog”，“IP Catalog”按钮以及单击后弹出的“IP Catalog”窗口如下图所示。

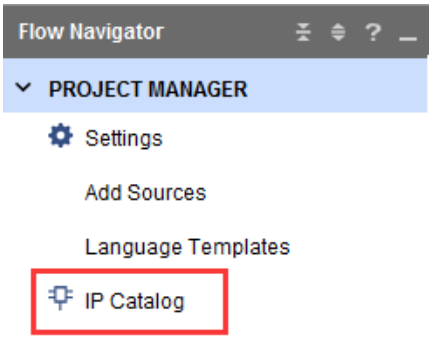


图 13.4.2 “IP Catalog”按钮

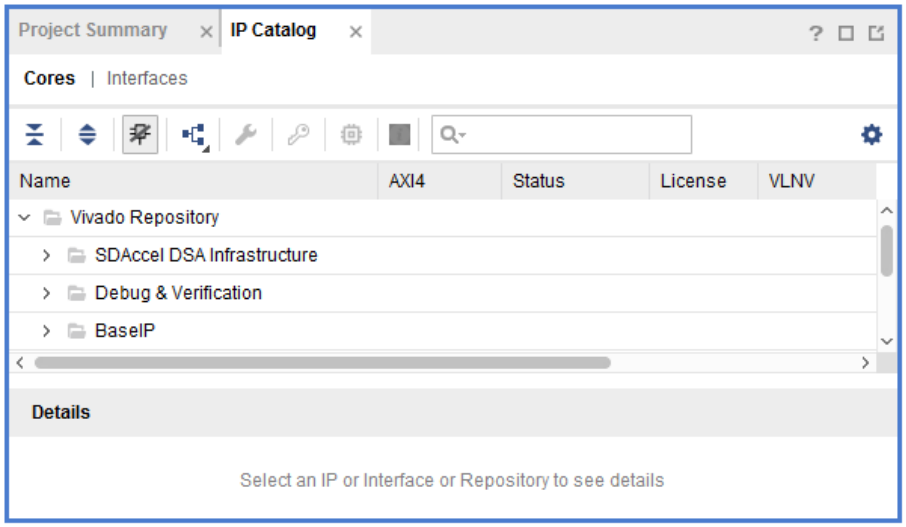


图 13.4.3 “IP Catalog”窗口

在“IP Catalog”窗口中，在搜索栏中输入“fifo”关键字，这时 Vivado 会自动查找出与关键字匹配的 IP 核

名称，我们双击“FIFO Generator”，如下图所示。

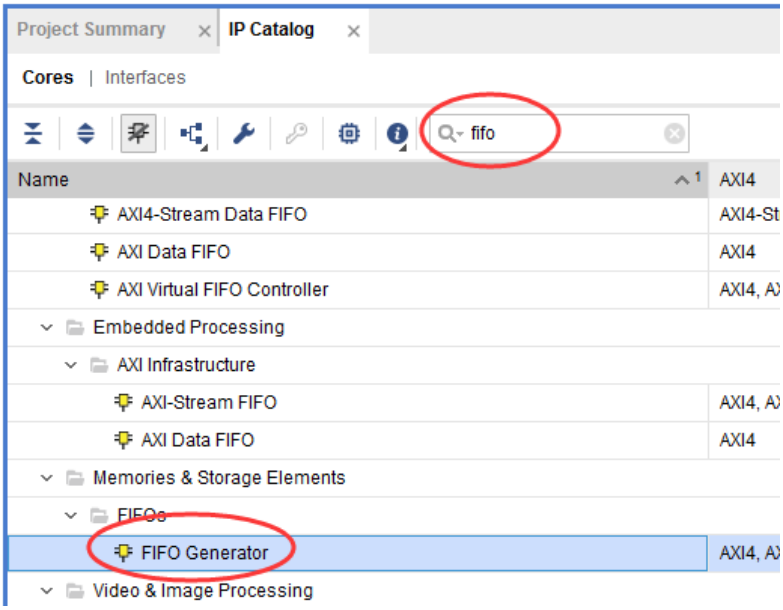


图 13.4.4 搜索栏中输入关键字

弹出“Customize IP”窗口，如下图所示。

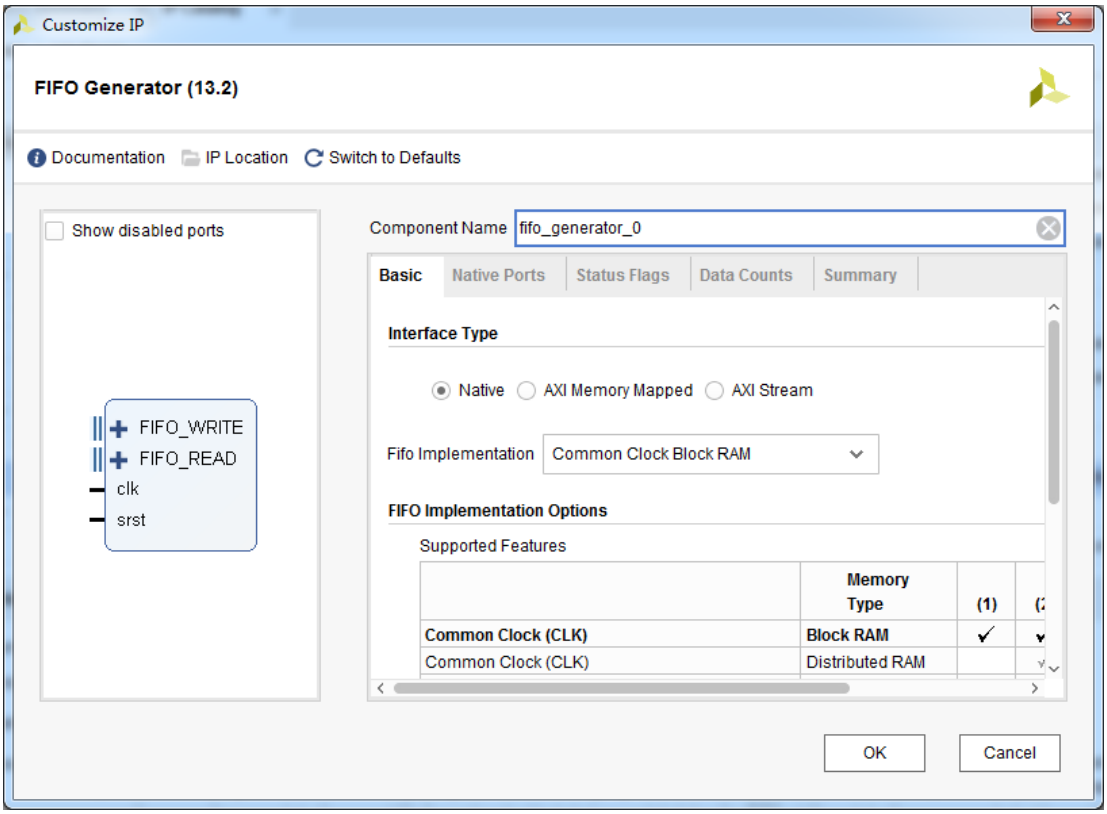


图 13.4.5 “Customize IP”窗口

接下来就是配置 IP 核的时钟参数的过程。

最上面的“Component Name”一栏设置该 IP 元件的名称，这里保持默认即可。在第一个“Basic”选项卡中，“Interface Type”选项用于选择 FIFO 接口的类型，这里我们选择默认的“Native”，即传统意义上的 FIFO 接口。“Fifo Implementation”选项用于选择我们想要实现的是同步 FIFO 还是异步 FIFO 以及使用哪种资源

实现 FIFO，这里我们选择“Independent Clocks Block RAM”,即使用块 RAM 来实现的异步 FIFO。如下图所示。

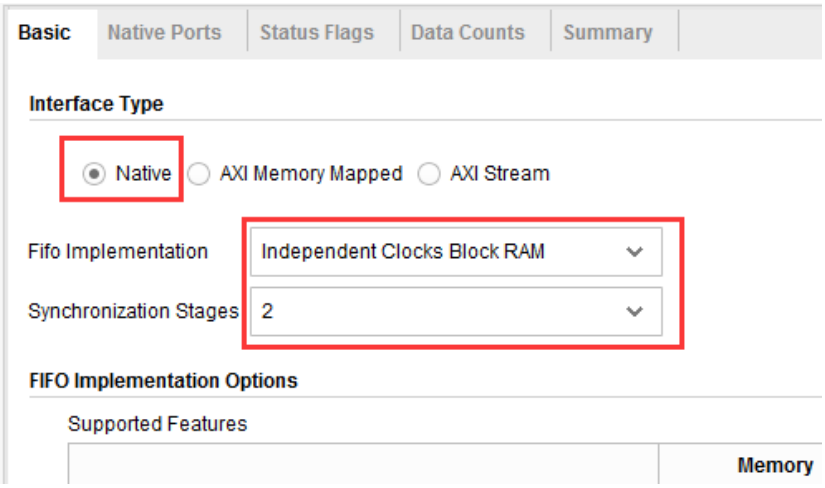


图 13.4.6 “Basic”选项卡

接下来是“Native Ports”选项卡，用于设置 FIFO 端口的参数。“Read Mode”选项用于设置读 FIFO 时的读模式，这里我们选择默认的“Standard FIFO”。“Data Port Parameters”一栏用于设置读写端口的数据总线的宽度以及 FIFO 的深度，写宽度“Write Width”我们设置为 8 位，写深度“Write Depth”我们设置为 256，注意此时 FIFO IP 核所能实现的实际深度却是 255；虽然读宽度“Read Width”能够设置成和写宽度不一样的位宽，且此时读深度“Read Depth”会根据上面三个参数被动地自动设置成相应的值；但是我们还是将读宽度“Read Width”设置成和写宽度“Write Width”一样的位宽，这也是在实际应用中最常用的情况。由于我们只是观察 FIFO 的读写，所以最下面的“Reset Pin”选项我们可以不使用，把它取消勾选。其他设置保持默认即可，如下图所示。

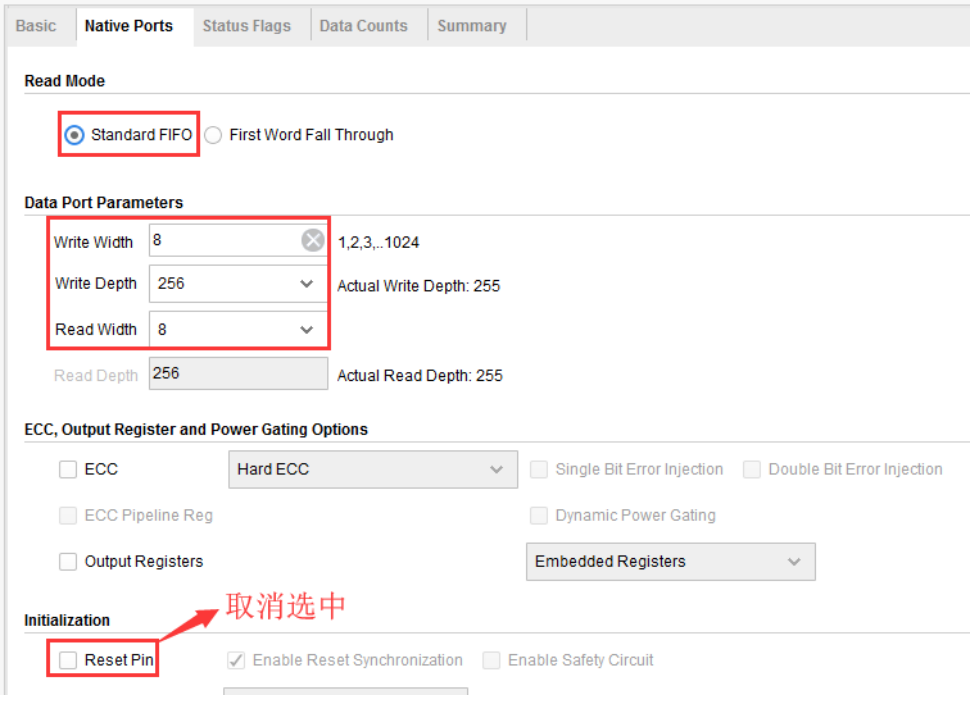


图 13.4.7 “Native Ports”选项卡

“Status Flags”选项卡，用于设置用户自定义接口或者用于设定专用的输入口。这里我们使用“即将写满”和“即将读空”这两个信号，所以我们把它们勾选上，其他保持默认即可，如下图所示。

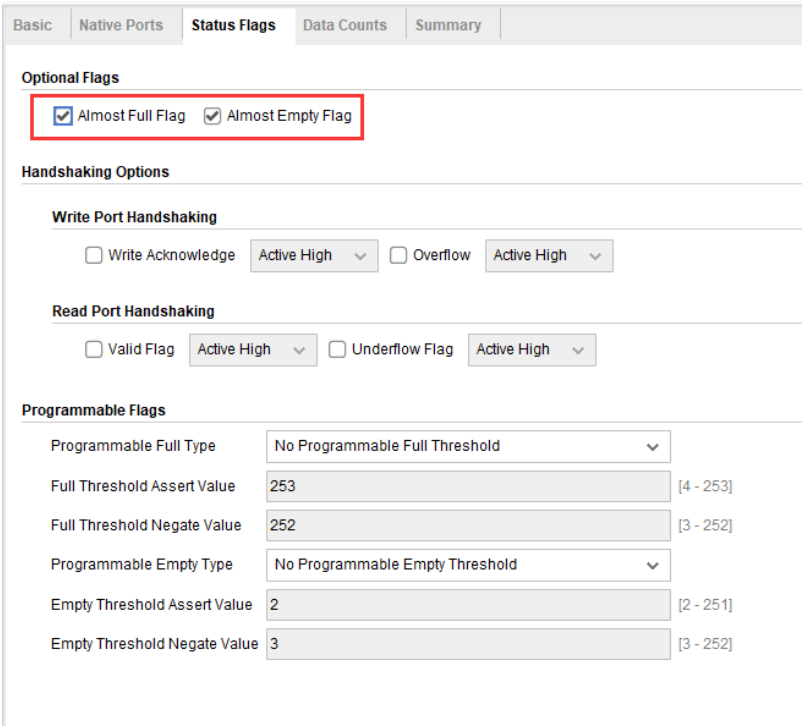


图 13.4.8 “Status Flags”选项卡

“Data Counts”选项卡用于设置 FIFO 内数据计数的输出信号，此信号表示当前在 FIFO 内存在多少个有效数据。为了更加方便地观察读/写过程，这里我们把读/写端口的数据计数都打开，且计数值总线的位宽设置为满位宽，即 8 位，如下图所示。

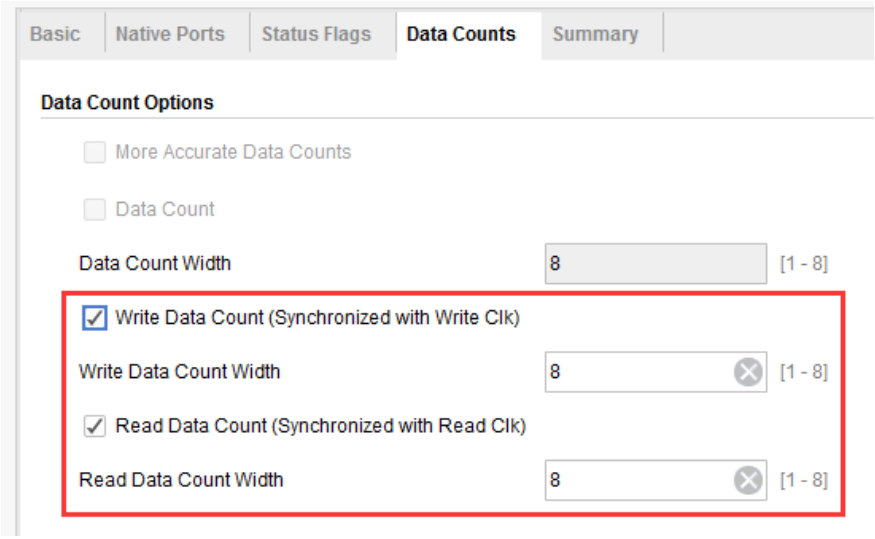


图 13.4.9 “Data Counts”选项卡

最后的“Summary”选项卡是对前面所有配置的一个总结，在这里我们直接点击“OK”按钮即可，如下图所示。

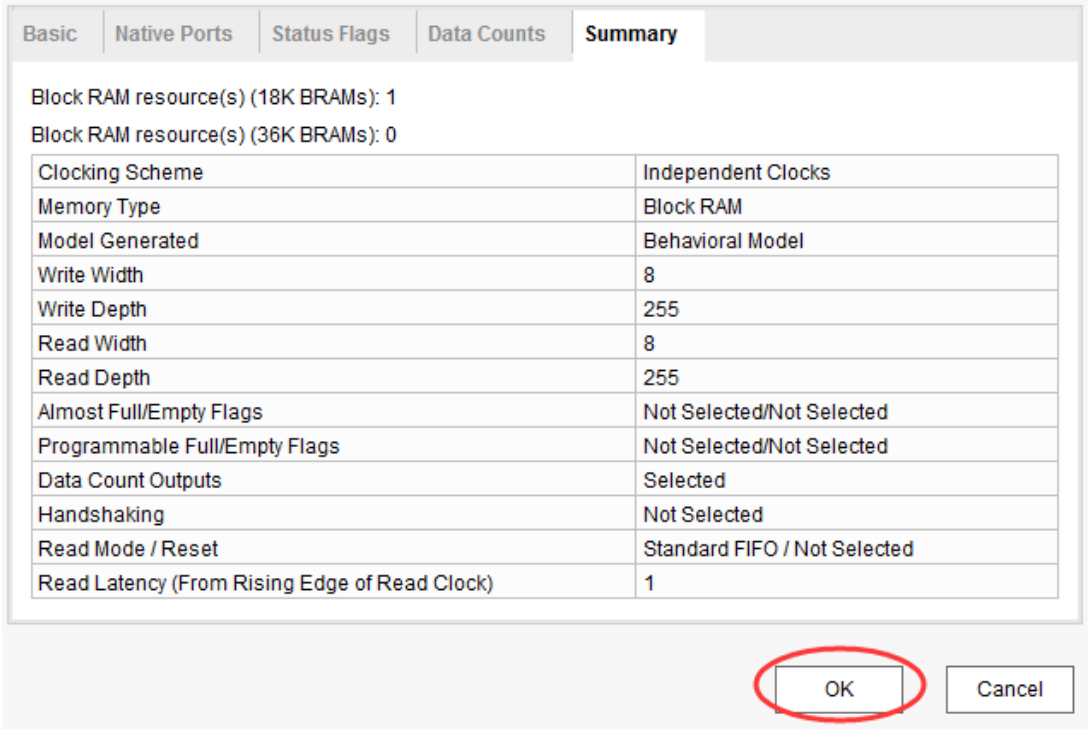


图 13.4.10 “Summary”选项卡

接着就弹出了“Genarate Output Products”窗口，我们直接点击“Generate”即可，如下图所示。

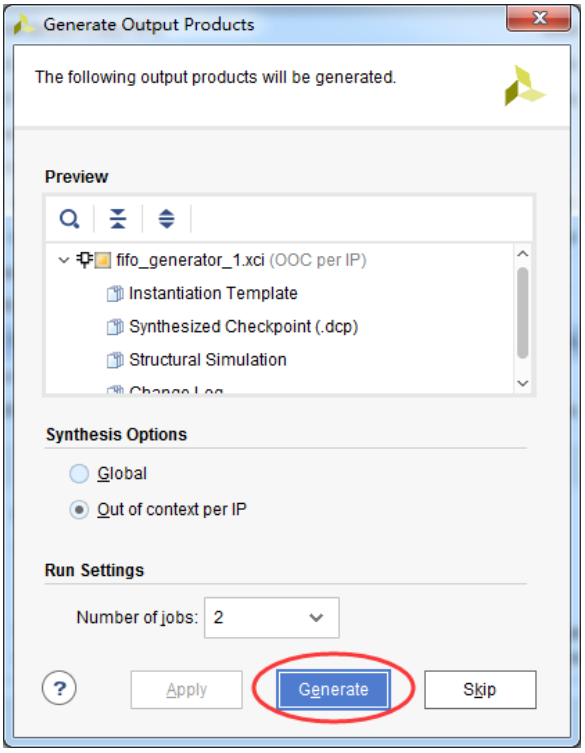


图 13.4.11 “Genarate Output Products”窗口

之后我们就可以在“Design Run”窗口的“Out-of-Context Module Runs”一栏中出现了该 IP 核对应的 run“fifo_generator_0_synth_1”，其综合过程独立于顶层设计的综合，所以在我们可以看到其正在综合，如下图所示。

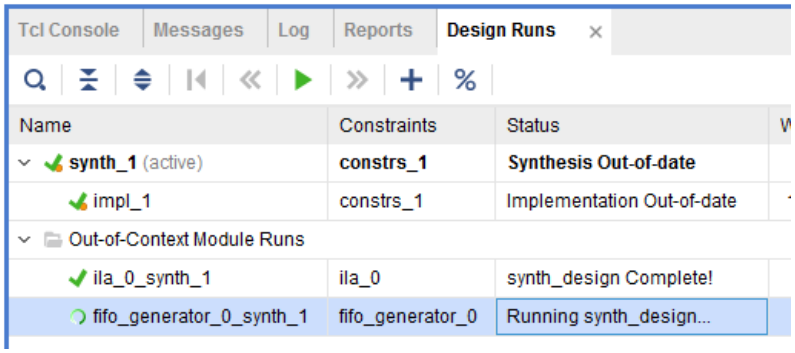


图 13.4.12 “fifo_generator _0_synth_1”run

在其 Out-of-Context 综合的过程中，我们就可以进行 RTL 编码了。首先打开 IP 核的例化模板，在“Source”窗口中的“IP Sources”选项卡中，依次用鼠标单击展开“IP”-“fifo_generator _0”-“Instantitation Template”，我们可以看到“fifo_generator_0.veo”文件，它是由 IP 核自动生成的只读的 verilog 例化模板文件，双击就可以打开它，如下图所示。

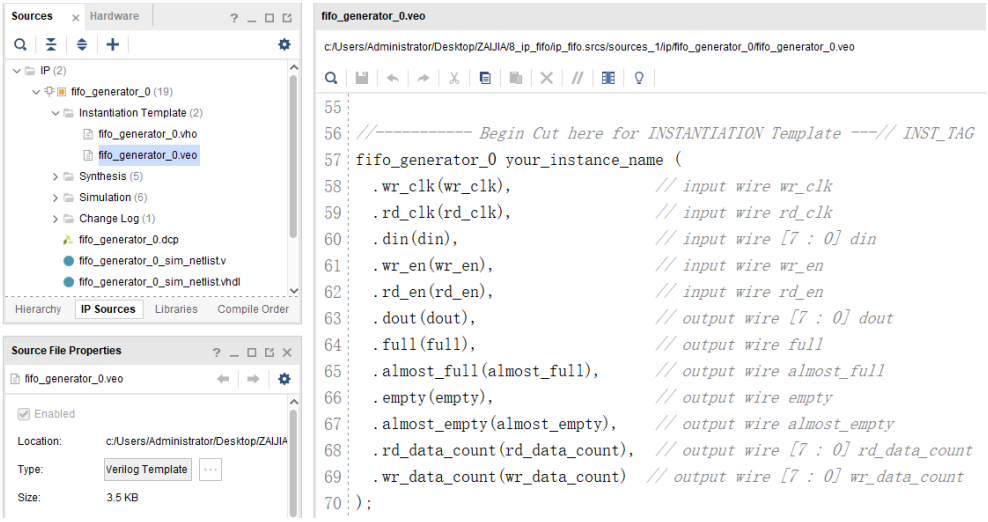


图 13.4.13 “fifo_generator_0.veo”文件

我们创建一个 verilog 源文件，其名称为 ip_fifo.v，作为顶层模块，其代码如下：

```
1  module ip_fifo(
2      input     sys_clk ,  // 时钟信号
3      input     sys_rst_n  // 复位信号
4  );
5
6  //wire define
7  wire         fifo_wr_en         ;  // FIFO 写使能信号
8  wire         fifo_rd_en         ;  // FIFO 读使能信号
9  wire  [7:0]  fifo_din           ;  // 写入到 FIFO 的数据
10 wire  [7:0]  fifo_dout          ;  // 从 FIFO 读出的数据
11 wire         almost_full        ;  // FIFO 将满信号
12 wire         almost_empty       ;  // FIFO 将空信号
13 wire         fifo_full          ;  // FIFO 满信号
```

```
14 wire         fifo_empty         ;  // FIFO 空信号
15 wire  [7:0]  fifo_wr_data_count ;  // FIFO 写时钟域的数据计数
16 wire  [7:0]  fifo_rd_data_count ;  // FIFO 读时钟域的数据计数
17
18 //*****************************************************
19 //**                    main code
20 //*****************************************************
21
22 //例化 FIFO IP 核
23 fifo_generator_0  fifo_generator_0 (
24     .wr_clk        ( sys_clk           ),  // input wire wr_clk
25     .rd_clk        ( sys_clk           ),  // input wire rd_clk
26
27     .wr_en         ( fifo_wr_en        ),  // input wire wr_en
28     .rd_en         ( fifo_rd_en        ),  // input wire rd_en
29
30     .din           ( fifo_din          ),  // input wire [7 : 0] din
31     .dout          ( fifo_dout         ),  // output wire [7 : 0] dout
32
33     .almost_full   (almost_full        ),  // output wire almost_full
34     .almost_empty  (almost_empty       ),  // output wire almost_empty
35     .full          ( fifo_full         ),  // output wire full
36     .empty         ( fifo_empty        ),  // output wire empty
37
38  .wr_data_count ( fifo_wr_data_count ),  // output wire [7 : 0] wr_data_count
39  .rd_data_count ( fifo_rd_data_count )   // output wire [7 : 0] rd_data_count
40 );
41
42 //例化写 FIFO 模块
43 fifo_wr  u_fifo_wr(
44     .clk            ( sys_clk    ),   // 写时钟
45     .rst_n          ( sys_rst_n  ),   // 复位信号
46
47     .fifo_wr_en     ( fifo_wr_en ) , // fifo 写请求
48     .fifo_wr_data   ( fifo_din    ) , // 写入 FIFO 的数据
49     .almost_empty   ( almost_empty ), // fifo 将空信号
50     .almost_full    ( almost_full  )  // fifo 将满信号
51 );
52
53 //例化读 FIFO 模块
54 fifo_rd  u_fifo_rd(
```

```
55      .clk          ( sys_clk    ),      // 读时钟
56      .rst_n        ( sys_rst_n  ),      // 复位信号
57
58      .fifo_rd_en   ( fifo_rd_en ),      // fifo 读请求
59      .fifo_dout    ( fifo_dout  ),      // 从 FIFO 输出的数据
60      .almost_empty ( almost_empty ),    // fifo 将空信号
61      .almost_full  ( almost_full  )     // fifo 将满信号
62  );
63
64  //例化 ILA IP 核
65  ila_0  ila_0 (
66   .clk    ( sys_clk           ), // input wire clk
67
68   .probe0 ( fifo_wr_en        ), // input wire [0:0]  probe0
69   .probe1 ( fifo_rd_en        ), // input wire [0:0]  probe1
70   .probe2 ( fifo_din          ), // input wire [7:0]  probe2
71   .probe3 ( fifo_dout         ), // input wire [7:0]  probe3
72   .probe4 ( fifo_empty        ), // input wire [0:0]  probe4
73   .probe5 ( almost_empty      ), // input wire [0:0]  probe5
74   .probe6 ( fifo_full         ), // input wire [0:0]  probe6
75   .probe7 ( almost_full       ), // input wire [0:0]  probe7
76   .probe8 ( fifo_wr_data_count ), // input wire [7:0]  probe8
77   .probe9( fifo_rd_data_count  )  // input wire [7:0]  probe9
78  );
79
80  endmodule
```

顶层模块主要是对 FIFO IP 核、写 FIFO 模块、读 FIFO 模块进行例化，除此之外本实验还生成并例化了一个 ILA IP 核，用于对顶层模块信号的进行在线捕获观察。

写 FIFO 模块 fifo_wr.v 源文件的代码如下：

```
1   module fifo_wr(
2       input                   clk    ,          // 时钟信号
3       input                   rst_n  ,          // 复位信号
4
5       input                   almost_empty,     // FIFO 将空信号
6       input                   almost_full ,     // FIFO 将满信号
7       output    reg           fifo_wr_en ,      // FIFO 写使能
8       output    reg  [7:0]    fifo_wr_data      // 写入 FIFO 的数据
9   );
10
11  //reg define
12  reg  [1:0]  state            ;  //动作状态
```

```
13 reg         almost_empty_d0  ;  //almost_empty 延迟一拍
14 reg         almost_empty_syn ;  //almost_empty 延迟两拍
15 reg  [3:0]  dly_cnt          ;  //延迟计数器
16 //*****************************************************
17 //**                    main code
18 //*****************************************************
19
20 //因为 almost_empty 信号是属于 FIFO 读时钟域的
21 //所以要将其同步到写时钟域中
22 always@( posedge clk ) begin
23  if( !rst_n ) begin
24      almost_empty_d0  <= 1'b0 ;
25      almost_empty_syn <= 1'b0 ;
26  end
27  else begin
28      almost_empty_d0  <= almost_empty ;
29      almost_empty_syn <= almost_empty_d0 ;
30  end
31 end
32
33 //向 FIFO 中写入数据
34 always @(posedge clk ) begin
35     if(!rst_n) begin
36         fifo_wr_en   <= 1'b0;
37         fifo_wr_data <= 8'd0;
38         state        <= 2'd0;
39      dly_cnt      <= 4'd0;
40     end
41     else begin
42         case(state)
43             2'd0: begin
44                 if(almost_empty_syn) begin  //如果检测到 FIFO 将被读空（下一拍就会空）
45                     state <= 2'd1;          //就进入延时状态
46                 end
47                 else
48                     state <= state;
49             end
50          2'd1: begin
51                 if(dly_cnt == 4'd10) begin  //延时 10 拍
52                                             //原因是 FIFO IP 核内部状态信号的更新存在延时
53                                             //延迟 10 拍以等待状态信号更新完毕
```

```
54                      dly_cnt    <= 4'd0;
55                  state      <= 2'd2;         //开始写操作
56                  fifo_wr_en <= 1'b1;         //打开写使能
57              end
58              else
59                  dly_cnt <= dly_cnt + 4'd1;
60              end
61          2'd2: begin
62                  if(almost_full) begin       //等待 FIFO 将被写满（下一拍就会满）
63                      fifo_wr_en   <= 1'b0;  //关闭写使能
64                      fifo_wr_data <= 8'd0;
65                      state        <= 2'd0;  //回到第一个状态
66                  end
67                  else begin                  //如果 FIFO 没有被写满
68                      fifo_wr_en   <= 1'b1;  //则持续打开写使能
69                      fifo_wr_data <= fifo_wr_data + 1'd1;  //且写数据值持续累加
70                  end
71              end
72          default : state <= 2'd0;
73          endcase
74      end
75  end
76
77  endmodule
```

fifo_wr 模块的核心部分是一个不断进行状态循环的小状态机，如果检测到 FIFO 为空，则先延时 10 拍，这里注意，由于 FIFO 的内部信号的更新比实际的数据读/写操作有所延时，所以延时 10 拍的目的是等待 FIFO 的空/满状态信号、数据计数信号等信号的更新完毕之后再进行 FIFO 写操作，如果写满，则回到状态 0，即等待 FIFO 被读空，以进行下一轮的写操作。

读 FIFO 模块 fifo_rd.v 源文件的代码如下：

```
1  module fifo_rd(
2      input               clk          ,   // 时钟信号
3      input               rst_n        ,   // 复位信号
4
5      input        [7:0]  fifo_dout    ,   // 从 FIFO 读出的数据
6      input               almost_full  ,   // FIFO 将满信号
7      input               almost_empty ,   // FIFO 将空信号
8      output  reg         fifo_rd_en       // FIFO 读使能
9  );
10
11 //reg define
12 reg  [1:0]  state          ;  //动作状态
```

```
13 reg         almost_full_d0  ;  //almost_full 延迟一拍
14 reg         almost_full_syn ;  //almost_full 延迟两拍
15 reg  [3:0]  dly_cnt         ;  //延迟计数器
16
17 //*****************************************************
18 //**                    main code
19 //*****************************************************
20
21 //因为 fifo_full 信号是属于 FIFO 写时钟域的
22 //所以要将其同步到读时钟域中
23 always@( posedge clk ) begin
24  if( !rst_n ) begin
25      almost_full_d0  <= 1'b0 ;
26      almost_full_syn <= 1'b0 ;
27  end
28  else begin
29      almost_full_d0  <= almost_full ;
30      almost_full_syn <= almost_full_d0 ;
31  end
32 end
33
34 //读出 FIFO 的数据
35 always @(posedge clk ) begin
36     if(!rst_n) begin
37         fifo_rd_en <= 1'b0;
38      state      <= 2'd0;
39      dly_cnt    <= 4'd0;
40     end
41     else begin
42         case(state)
43             2'd0: begin
44                 if(almost_full_syn)         //如果检测到 FIFO 被写满
45                     state <= 2'd1;          //就进入延时状态
46                 else
47                     state <= state;
48             end
49          2'd1: begin
50                 if(dly_cnt == 4'd10) begin //延时 10 拍
51                                            //原因是 FIFO IP 核内部状态信号的更新存在延时
52                                            //延迟 10 拍以等待状态信号更新完毕
53                     dly_cnt <= 4'd0;
```

```
54                  state   <= 2'd2;          //开始读操作
55              end
56              else
57                  dly_cnt <= dly_cnt + 4'd1;
58              end
59          2'd2: begin
60                  if(almost_empty) begin     //等待 FIFO 将被读空（下一拍就会空）
61                      fifo_rd_en <= 1'b0;    //关闭读使能
62                      state      <= 2'd0;    //回到第一个状态
63                  end
64                  else                       //如果 FIFO 没有被读空
65                      fifo_rd_en <= 1'b1;    //则持续打开读使能
66              end
67           default : state <= 2'd0;
68          endcase
69      end
70  end
71
72  endmodule
```

读模块的代码结构与写模块几乎一样，也是使用一个不断进行状态循环的小的状态机来控制操作过程，读者参考着代码应该很容易能够理解，这里就不再赘述。

我们对代码进行仿真，TestBench 中只要送出时钟的复位信号即可。TB 文件如下：

```
1  `timescale 1ns / 1ps
2  module tb_ip_fifo( );
3      // Inputs
4      reg sys_clk;
5      reg sys_rst_n;
6
7  // Instantiate the Unit Under Test (UUT)
8  ip_fifo  u_ip_fifo (
9          .sys_clk         (sys_clk),
10         .sys_rst_n       (sys_rst_n)
11     );
12
13 //Genarate the clk
14 parameter PERIOD = 20;
15 always begin
16     sys_clk = 1'b0;
17     #(PERIOD/2) sys_clk = 1'b1;
18     #(PERIOD/2);
19 end
```

```
20
21 initial begin
22     // Initialize Inputs
23     sys_rst_n = 0;
24     // Wait 100 ns for global reset to finish
25     #100  ;
26     sys_rst_n = 1;
27     // Add stimulus here
28 end
29
30 endmodule
```

写满后转为读的仿真波形图如下图所示：

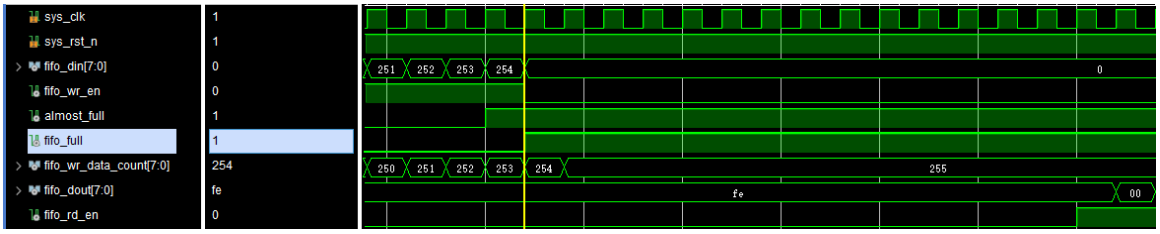


图 13.4.14 Vivado 仿真波形 1

由波形图可知，当写满 255 个数据后，fifo_full 满信号就会拉高。经过延时之后，fifo_rd_en 写使能信号拉高，经过一拍之后就开始将 fifo 中的数据送到 fifo_dout 端口上。

写满后转为读的仿真波形图如下图所示：

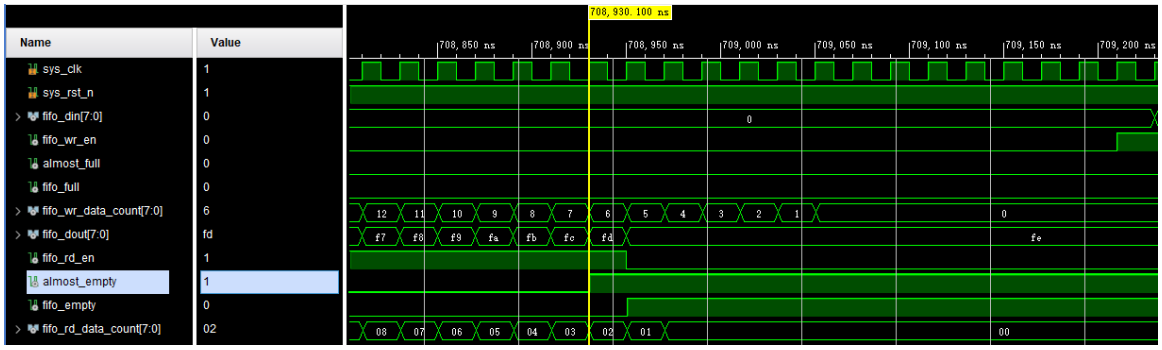


图 13.4.15 Vivado 仿真波形 2

由波形图可知，当读完 255 个数据后，fifo_empty 空信号就会拉高。经过延时之后，fifo_wr_en 写使能信号拉高，经过一拍之后就开始向 fifo 中继续写入数据。

13.5 下载验证

编译工程并生成比特流.bit 文件，将比特流.bit 文件下载到 Zynq 中。

下载完成后，接下来在 Vivado 中会自动出现“hw_ila_1”Dashboard 窗口。如下图所示：

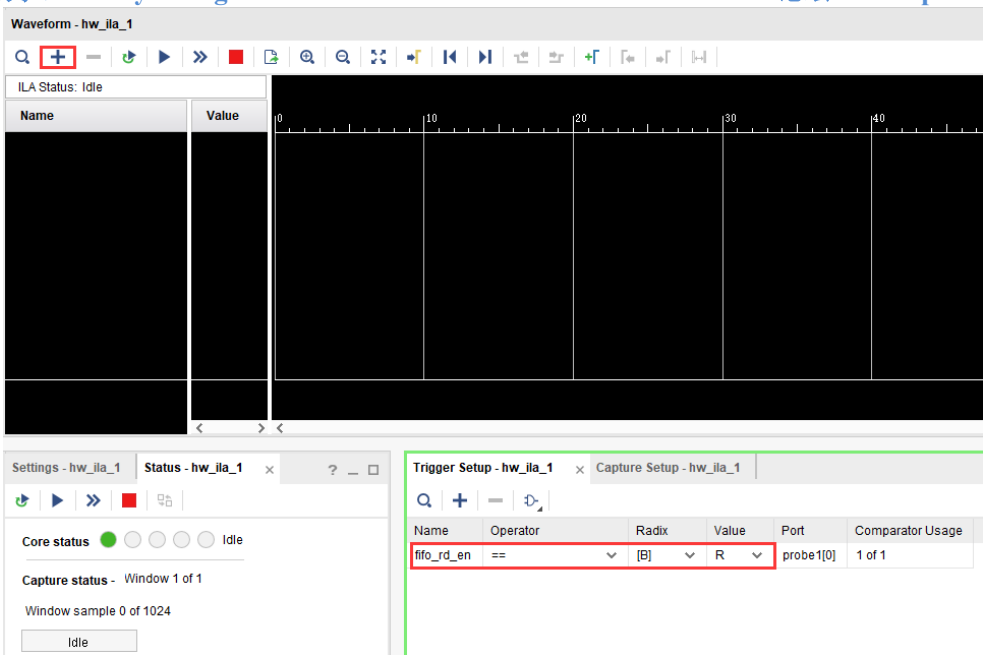


图 13.5.1 将探针信号添加到波形窗口中

将有关探针信号添加到波形窗口中，这里我们已经完成信号的添加，方法是点击“hw_ila_1”Dashboard 窗口左上角的“+”。同时我们在窗口右下角将“fifo_rd_en”信号添加到触发窗口中且设置为上升沿触发，单击左上角的触发按钮，如下图所示：

图 13.5.2 触发按钮

最后就看到了 ILA 捕获得到的数据，展开波形图如下图所示：

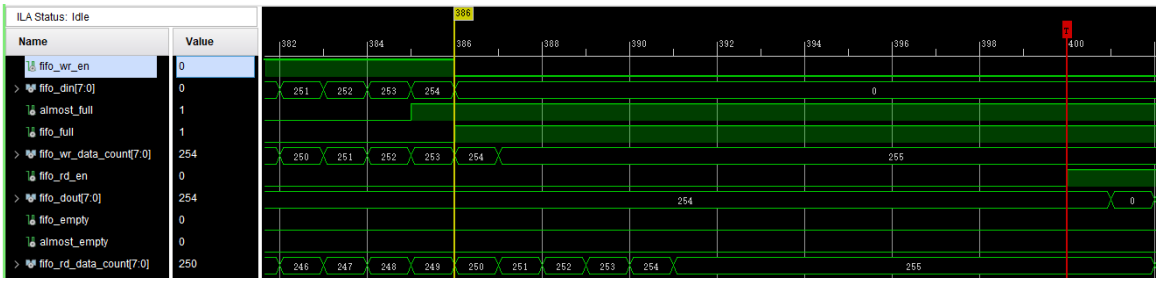


图 13.5.3 捕获得到的波形图

从捕获得到的波形图中可以看出，其逻辑行为与仿真波形图中的一致，证明我们的代码正确地实现了预期的功能。

第十四章 UART 串口通信实验

串口是“串行接口”的简称，即采用串行通信方式的接口。串行通信将数据字节分成一位一位的形式在一条数据线上逐个传送，其特点是通信线路简单，但传输速度较慢。因此串口广泛应用于嵌入式、工业控制等领域中对数据传输速度要求不高的场合。本章我们将使用启明星 Zynq 开发板，通过“ATK-USB-UART”模块来完成位机与 Zynq 的串口通信。

本章包括以下几个部分：

14.1 UART 串口简介

14.2 实验任务

14.3 硬件设计

14.4 程序设计

14.5 下载验证

14.1 UART 串口简介

串行通信分为两种方式：同步串行通信和异步串行通信。同步串行通信需要通信双方在同一时钟的控制下，同步传输数据；异步串行通信是指通信双方使用各自的时钟控制数据的发送和接收过程。

UART 是一种采用异步串行通信方式的通用异步收发传输器（universal asynchronous receiver-transmitter），它在发送数据时将并行数据转换成串行数据来传输，在接收数据时将接收到的串行数据转换成并行数据。

UART 串口通信需要两根信号线来实现，一根用于串口发送，另外一根负责串口接收。UART 在发送或接收过程中的一帧数据由 4 部分组成，**起始位**、**数据位**、**奇偶校验位**和**停止位**，如图 14.1.1 所示。其中，起始位标志着一帧数据的开始，停止位标志着一帧数据的结束，数据位是一帧数据中的有效数据。校验位分为奇校验和偶校验，用于检验数据在传输过程中是否出错。奇校验时，发送方应使数据位中 1 的个数与校验位中 1 的个数之和为奇数；接收方在接收数据时，对 1 的个数进行检查，若不为奇数，则说明数据在传输过程中出了差错。同样，偶校验则检查 1 的个数是否为偶数。

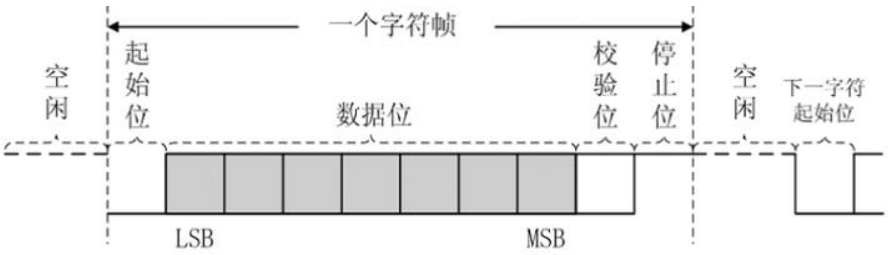


图 14.1.1 异步串行通信数据格式

UART 通信过程中的数据格式及传输速率是可设置的，为了正确的通信，收发双方应约定并遵循同样的设置。数据位可选择为 5、6、7、8 位，其中 8 位数据位是最常用的，在实际应用中一般都选择 8 位数据位；校验位可选择奇校验、偶校验或者无校验位；停止位可选择 1 位（默认），1.5 或 2 位。串口通信的速率用波特率表示，它表示每秒传输二进制数据的位数，单位是 bps（位/秒），常用的波特率有 9600、19200、38400、57600 以及 115200 等。

在设置好数据格式及传输速率之后，UART 负责完成数据的串并转换，而信号的传输则由外部驱动电路实现。电信号的传输过程有着不同的电平标准和接口规范，针对异步串行通信的接口标准有 RS232、RS422、RS485 等，它们定义了接口不同的电气特性，如 RS-232 是单端输入输出，而 RS-422/485 为差分输入输出等。

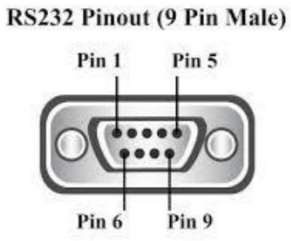
RS232 接口标准出现较早，可实现全双工工作方式，即数据发送和接收可以同时进行。在传输距离较短时（不超过 15m），RS232 是串行通信最常用的接口标准，本章主要介绍针对 RS-232 标准的 UART 串口通信。

RS-232 标准的串口最常见的接口类型为 DB9，样式如图 14.1.2 所示，工业控制领域中用到的工控机一般都配备多个串口，很多老式台式机也都配有串口。但是笔记本电脑以及较新一点的台式机都没有串口，它们一般通过 USB 转串口线（图 14.1.3）来实现与外部设备的串口通信。

图 14.1.2 DB9 接口

图 14.1.3 USB 串口线

DB9 接口定义以及各引脚功能说明如图 14.1.4 所示，我们一般只用到其中的 2（RXD）、3（TXD）、5（GND）引脚，其他引脚在普通串口模式下一般不使用，如果大家想了解，可以自行百度下。



引脚编号	引脚名称	功能说明
Pin 1	DCD	数据载波检测
Pin 2	RXD	接受数据
Pin 3	TXD	发送数据
Pin 4	DTR	数据终端准备
Pin 5	GND	地线
Pin 6	DSR	数据准备就绪
Pin 7	RTS	请求发送
Pin 8	CTS	清除发送
Pin 9	RI	振铃提示

图 14.1.4 DB9 接口定义

14.2 实验任务

本节实验任务是上位机通过串口调试助手发送数据给 Zynq，Zynq PL 端通过“ATK-USB-UART”模块接收数据，并将接收到的数据发送给上位机，完成串口数据环回。

14.3 硬件设计

由于启明星 Zynq 开发板上自带的 USB 转串口电路连接到了 Zynq 的 PS 端上，所以本实验要使用外接的 USB 转串口模块。本实验使用正点原子开发的“ATK-USB-UART 模块”，它使用了 CH340G 主芯片，如下图所示：

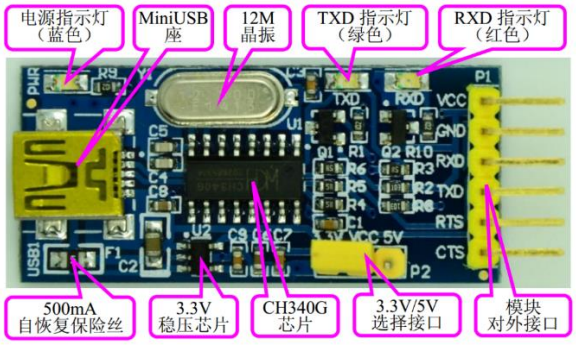


图 14.3.1 正点原子“ATK-USB-UART 模块”外观图

启明星 Zynq 开发板预留了“ATK-USB-UART 模块”的接口，其原理图如下图所示：

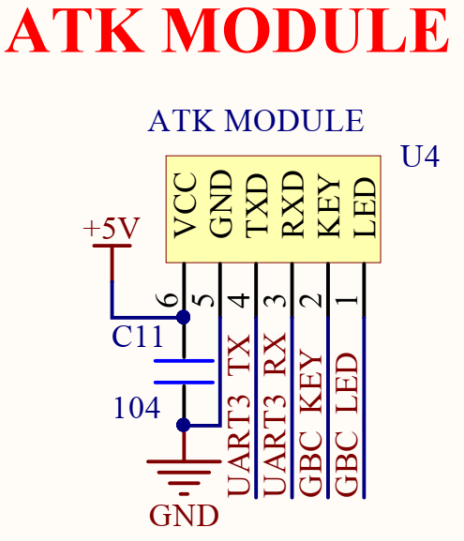


图 14.3.2 “ATK-USB-UART 模块”接口原理图

当然了，用户也可以选择使用其他的 USB 转串口模块。

本实验中，系统时钟、按键复位以及串口的接收、发送端口的管脚分配如下表所示：

表 14.3.1 串口通信实验管脚分配

信号名	方向	管脚	端口说明	电平标准
sys_clk	input	U18	系统时钟，50M	LVCMOS33
sys_rst_n	input	J15	系统复位，低有效	LVCMOS33
uart_rxd	input	P19	串口接收	LVCMOS33
uart_txd	output	N18	串口发送	LVCMOS33

对应的 XDC 约束语句如下所示：

```
create_clock -period 20.000 -name clk [get_ports sys_clk]
set_property -dict {PACKAGE_PIN U18 IOSTANDARD LVCMOS33} [get_ports sys_clk]
set_property -dict {PACKAGE_PIN J15 IOSTANDARD LVCMOS33} [get_ports sys_rst_n]
set_property -dict {PACKAGE_PIN P19 IOSTANDARD LVCMOS33} [get_ports uart_rxd]
set_property -dict {PACKAGE_PIN N18 IOSTANDARD LVCMOS33} [get_ports uart_txd]
```

14.4 程序设计

根据实验任务，我们不难想象本系统应该有一个串口接收模块，用来接收上位机发送的数据；还要有一个串口发送模块，用于将数据发回上位机；另外还应该有一个对数据进行环回控制的模块，它负责把从串口接收模块接收到的数据送给串口发送模块，以实现串口数据的环回。由此可以画出本次实验的系统框图，如下所示：

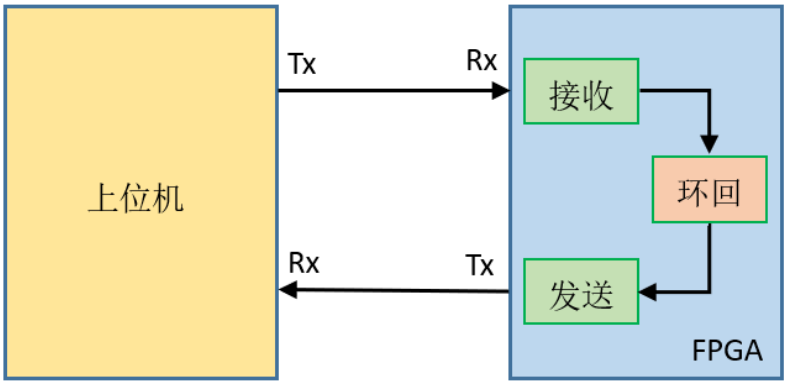


图 14.4.1 系统框图

由系统总体框图可知，ZYNQ PL 部分包括四个模块，顶层模块、接收模块、发送模块和数据环回模块。其中在顶层模块中完成对另外三个模块的例化，顶层模块原理图如下所示：

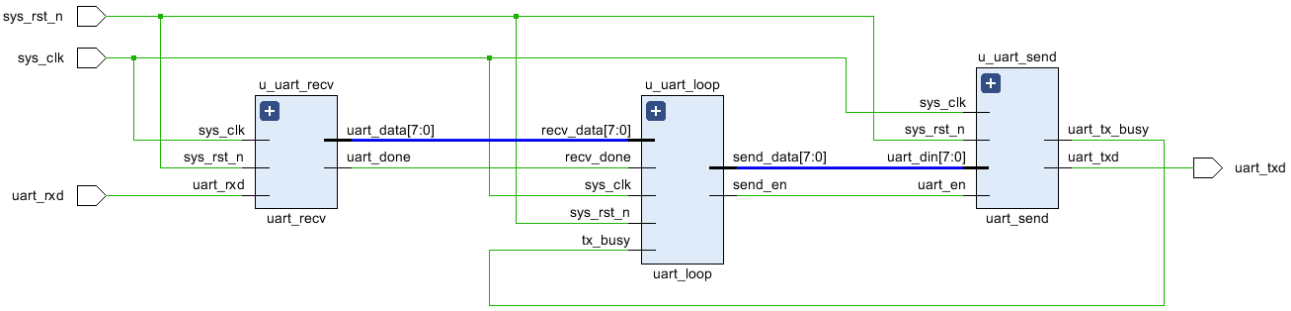


图 14.4.2 顶层模块原理图

在图 14.4.2 中，uart_recv 为串口接收模块，从串口接收端口 uart_rxd 来接收上位机发送的串行数据，并在一帧数据接收结束后给出通知信号 uart_done。

uart_send 为串口发送模块，以 uart_en 为发送使能信号。**uart_en 的上升沿将启动一次串口发送过程，将 uart_din 接口上的数据通过串口发送端口 uart_txd 发送出去**。

uart_loop 模块负责完成串口数据的环回功能。它在 uart_recv 模块接收完成后，将接收到的串口数据发送到 uart_send 模块，并通过 send_en 接口给出一个上升沿，以启动发送过程。

在编写代码之前，我们首先要确定串口通信的数据格式及波特率。在这里我们选择串口比较常用的一种模式，数据位为 8 位，停止位为 1 位，无校验位，波特率为 115200bps。则传输一帧数据的时序图如下图所示：

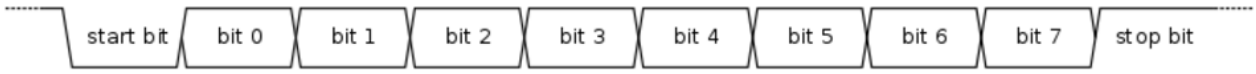


图 14.4.3 串口通信时序图

顶层模块的代码如下：

```
1  module uart_loopback_top(
2      input           sys_clk,          //外部 50M 时钟
3      input           sys_rst_n,        //外部复位信号，低有效
4
5      input           uart_rxd,         //UART 接收端口
6      output          uart_txd          //UART 发送端口
```

```
7       );
8
9   //parameter define
10  parameter  CLK_FREQ = 50000000;          //定义系统时钟频率
11  parameter  UART_BPS = 115200;            //定义串口波特率
12
13  //wire define
14  wire       uart_recv_done;               //UART 接收完成
15  wire [7:0] uart_recv_data;               //UART 接收数据
16  wire       uart_send_en;                 //UART 发送使能
17  wire [7:0] uart_send_data;               //UART 发送数据
18  wire       uart_tx_busy;                 //UART 发送忙状态标志
19
20  //*****************************************************
21  //**                    main code
22  //*****************************************************
23
24  //串口接收模块
25  uart_recv #(
26      .CLK_FREQ       (CLK_FREQ),          //设置系统时钟频率
27      .UART_BPS       (UART_BPS))          //设置串口接收波特率
28  u_uart_recv(
29      .sys_clk        (sys_clk),
30      .sys_rst_n      (sys_rst_n),
31
32      .uart_rxd       (uart_rxd),
33      .uart_done      (uart_recv_done),
34      .uart_data      (uart_recv_data)
35      );
36
37  //串口发送模块
38  uart_send #(
39      .CLK_FREQ       (CLK_FREQ),          //设置系统时钟频率
40      .UART_BPS       (UART_BPS))          //设置串口发送波特率
41  u_uart_send(
42      .sys_clk        (sys_clk),
43      .sys_rst_n      (sys_rst_n),
44
45      .uart_en        (uart_send_en),
46      .uart_din       (uart_send_data),
47      .uart_tx_busy   (uart_tx_busy),
```

```
48      .uart_txd       (uart_txd)
49      );
50
51  //串口环回模块
52  uart_loop u_uart_loop(
53      .sys_clk        (sys_clk),
54      .sys_rst_n      (sys_rst_n),
55
56      .recv_done      (uart_recv_done),    //接收一帧数据完成标志信号
57      .recv_data      (uart_recv_data),    //接收的数据
58
59      .tx_busy        (uart_tx_busy),      //发送忙状态标志
60      .send_en        (uart_send_en),      //发送使能信号
61      .send_data      (uart_send_data)     //待发送数据
62      );
63
64  endmodule
```

在顶层模块中完成了对其余各个子模块的例化。需要注意的是，顶层模块中第 10、11 行定义了两个变量：系统时钟频率 CLK_FREQ 与串口波特率 UART_BPS，使用时可以根据不同的系统时钟频率以及所需要的串口波特率设置这两个变量。我们可以尝试将串口波特率 UART_BPS 设置为其他值（如 9600），在模块例化时会将这个变量传递到串口接收与发送模块中，从而实现不同速率的串口通信。

串口接收模块的代码如下所示：

```
1   module uart_recv(
2       input              sys_clk,                 //系统时钟
3       input              sys_rst_n,               //系统复位，低电平有效
4
5       input              uart_rxd,                //UART 接收端口
6       output  reg        uart_done,               //接收一帧数据完成标志
7       output  reg [7:0]  uart_data                //接收的数据
8       );
9
10  //parameter define
11  parameter  CLK_FREQ = 50000000;                //系统时钟频率
12  parameter  UART_BPS = 9600;                    //串口波特率
13  localparam  BPS_CNT  = CLK_FREQ/UART_BPS;       //为得到指定波特率，
14                                                  //需要对系统时钟计数 BPS_CNT 次
15  //reg define
16  reg        uart_rxd_d0;
17  reg        uart_rxd_d1;
18  reg [15:0] clk_cnt;                            //系统时钟计数器
19  reg [ 3:0] rx_cnt;                             //接收数据计数器
```

```
20  reg         rx_flag;                            //接收过程标志信号
21  reg [ 7:0] rxdata;                              //接收数据寄存器
22
23  //wire define
24  wire        start_flag;
25
26  //*****************************************************
27  //**                    main code
28  //*****************************************************
29  //捕获接收端口下降沿(起始位)，得到一个时钟周期的脉冲信号
30  assign  start_flag = uart_rxd_d1 & (~uart_rxd_d0);
31
32  //对 UART 接收端口的数据延迟两个时钟周期
33  always @(posedge sys_clk or negedge sys_rst_n) begin
34      if (!sys_rst_n) begin
35          uart_rxd_d0 <= 1'b0;
36          uart_rxd_d1 <= 1'b0;
37      end
38      else begin
39          uart_rxd_d0  <= uart_rxd;
40          uart_rxd_d1  <= uart_rxd_d0;
41      end
42  end
43
44  //当脉冲信号 start_flag 到达时，进入接收过程
45  always @(posedge sys_clk or negedge sys_rst_n) begin
46      if (!sys_rst_n)
47          rx_flag <= 1'b0;
48      else begin
49          if(start_flag)                          //检测到起始位
50              rx_flag <= 1'b1;                    //进入接收过程，标志位 rx_flag 拉高
51                                                  //计数到停止位中间时，停止接收过程
52          else if((rx_cnt == 4'd9) && (clk_cnt == BPS_CNT/2))
53              rx_flag <= 1'b0;                    //接收过程结束，标志位 rx_flag 拉低
54          else
55              rx_flag <= rx_flag;
56      end
57  end
58
59  //进入接收过程后，启动系统时钟计数器
60  always @(posedge sys_clk or negedge sys_rst_n) begin
```

```
61      if (!sys_rst_n)
62          clk_cnt <= 16'd0;
63      else if ( rx_flag ) begin                   //处于接收过程
64          if (clk_cnt < BPS_CNT - 1)
65              clk_cnt <= clk_cnt + 1'b1;
66          else
67              clk_cnt <= 16'd0;                   //对系统时钟计数达一个波特率周期后清零
68      end
69      else
70          clk_cnt <= 16'd0;                       //接收过程结束，计数器清零
71  end
72
73  //进入接收过程后，启动接收数据计数器
74  always @(posedge sys_clk or negedge sys_rst_n) begin
75      if (!sys_rst_n)
76          rx_cnt  <= 4'd0;
77      else if ( rx_flag ) begin                   //处于接收过程
78          if (clk_cnt == BPS_CNT - 1)             //对系统时钟计数达一个波特率周期
79              rx_cnt <= rx_cnt + 1'b1;            //此时接收数据计数器加 1
80          else
81              rx_cnt <= rx_cnt;
82      end
83       else
84          rx_cnt  <= 4'd0;                        //接收过程结束，计数器清零
85  end
86
87  //根据接收数据计数器来寄存 uart 接收端口数据
88  always @(posedge sys_clk or negedge sys_rst_n) begin
89      if ( !sys_rst_n)
90          rxdata <= 8'd0;
91      else if(rx_flag)                            //系统处于接收过程
92          if (clk_cnt == BPS_CNT/2) begin         //判断系统时钟计数器计数到数据位中间
93              case ( rx_cnt )
94                  4'd1 : rxdata[0] <= uart_rxd_d1;   //寄存数据位最低位
95                  4'd2 : rxdata[1] <= uart_rxd_d1;
96                  4'd3 : rxdata[2] <= uart_rxd_d1;
97                  4'd4 : rxdata[3] <= uart_rxd_d1;
98                  4'd5 : rxdata[4] <= uart_rxd_d1;
99                  4'd6 : rxdata[5] <= uart_rxd_d1;
100                 4'd7 : rxdata[6] <= uart_rxd_d1;
101                 4'd8 : rxdata[7] <= uart_rxd_d1;   //寄存数据位最高位
```

```
102                 default:;
103                endcase
104            end
105            else
106                rxdata <= rxdata;
107        else
108            rxdata <= 8'd0;
109    end
110
111    //数据接收完毕后给出标志信号并寄存输出接收到的数据
112    always @(posedge sys_clk or negedge sys_rst_n) begin
113        if (!sys_rst_n) begin
114            uart_data <= 8'd0;
115            uart_done <= 1'b0;
116        end
117        else if(rx_cnt == 4'd9) begin                //接收数据计数器计数到停止位时
118            uart_data <= rxdata;                     //寄存输出接收到的数据
119            uart_done <= 1'b1;                       //并将接收完成标志位拉高
120        end
121        else begin
122            uart_data <= 8'd0;
123            uart_done <= 1'b0;
124        end
125    end
126
127    endmodule
```

串口接收模块程序中 29 至 42 行是一个经典的边沿检测电路，通过检测串口接收端 uart_rxd 的下降沿来捕获起始位。一旦检测到起始位，输出一个时钟周期的脉冲 start_flag，并进入串口接收过程。串口接收状态用 rx_flag 来标志，rx_flag 为高标志着串口接收过程正在进行，此时启动系统时钟计数器 clk_cnt 与接收数据计数器 rx_cnt。

由第 13 行的公式 BPS_CNT = CLK_FREQ/UART_BPS 可知，BPS_CNT 为当前波特率下，串口传输一位所需要的系统时钟周期数。因此 clk_cnt 从零计数到 BPS_CN-1 时，串口刚好完成一位数据的传输。由于接收数据计数器 rx_cnt 在每次 clk_cnt 计数到 BPS_CN-1 时加 1，因此由 rx_cnt 的值可以判断串口当前传输的是第几位数据。第 87 行至第 109 行就是根据 clk_cnt 的值将 uart 接收端口的数据寄存到接收数据寄存器对应的位，从而实现接收数据的串并转换。其中第 92 行选择 clk_cnt 计数至 BPS_CNT/2 时寄存接收端口数据，是因为计数到数据中间时的采样结果最稳定。

程序中需要额外注意的地方是串口接收过程结束条件的判定，由第 52 行可知，在计数到停止位中间时，标志位 rx_flag 就已经拉低。这样做是因为虽然此时一帧数据传输还没有完成（停止位只传送到一半），但是数据位已经寄存完毕。而在连续接收数据时，提前半个波特率周期结束接收过程可以为检测下一帧数据的起始位留出充足的时间。

我们使用上位机通过串口向开发板发送 16 进制数 55，在串口接收过程中 ILA 抓取的波形图如下所示：

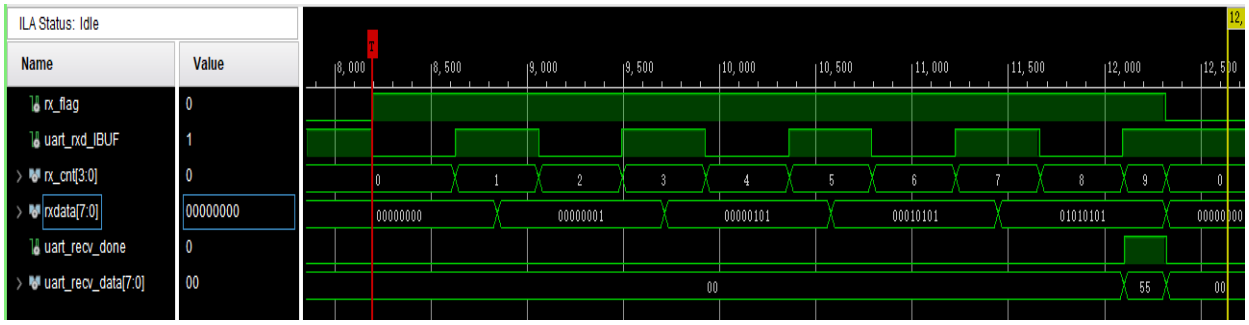


图 14.4.4 串口接收过程波形图

图 14.4.4 中红色的触发线标识出了串口接收端 uart_rxd 的起始位，在整个接收过程中 rx_flag 保持为高电平，同时 rx_cnt 对串口数据进行计数。当 rx_cnt 计数到 9 时，串口数据接收完成，uart_done 拉高，同时 uart_data 给出接收到的数据。从图中可以看到，接收模块能够正确接收串口数据并完成串并转换。

串口发送模块代码如下所示：

```
1   module uart_send(
2       input          sys_clk,                 //系统时钟
3       input          sys_rst_n,               //系统复位，低电平有效
4
5       input          uart_en,                 //发送使能信号
6       input  [7:0]   uart_din,                //待发送数据
7       output         uart_tx_busy,            //发送忙状态标志
8       output  reg    uart_txd                 //UART 发送端口
9       );
10
11  //parameter define
12  parameter  CLK_FREQ = 50000000;            //系统时钟频率
13  parameter  UART_BPS = 9600;                //串口波特率
14  localparam  BPS_CNT  = CLK_FREQ/UART_BPS;  //为得到指定波特率，对系统时钟计数 BPS_CNT 次
15
16  //reg define
17  reg         uart_en_d0;
18  reg         uart_en_d1;
19  reg [15:0] clk_cnt;                        //系统时钟计数器
20  reg [ 3:0] tx_cnt;                         //发送数据计数器
21  reg         tx_flag;                       //发送过程标志信号
22  reg [ 7:0] tx_data;                        //寄存发送数据
23
24  //wire define
25  wire        en_flag;
26
27  //*****************************************************
28  //**                    main code
```

```
29  //****************************************************
30  //在串口发送过程中给出忙状态标志
31  assign uart_tx_busy = tx_flag;
32
33  //捕获 uart_en 上升沿，得到一个时钟周期的脉冲信号
34  assign en_flag = (~uart_en_d1) & uart_en_d0;
35
36  //对发送使能信号 uart_en 延迟两个时钟周期
37  always @(posedge sys_clk or negedge sys_rst_n) begin
38      if (!sys_rst_n) begin
39          uart_en_d0 <= 1'b0;
40          uart_en_d1 <= 1'b0;
41      end
42      else begin
43          uart_en_d0 <= uart_en;
44          uart_en_d1 <= uart_en_d0;
45      end
46  end
47
48  //当脉冲信号 en_flag 到达时,寄存待发送的数据，并进入发送过程
49  always @(posedge sys_clk or negedge sys_rst_n) begin
50      if (!sys_rst_n) begin
51          tx_flag <= 1'b0;
52          tx_data <= 8'd0;
53      end
54      else if (en_flag) begin                 //检测到发送使能上升沿
55              tx_flag <= 1'b1;                //进入发送过程，标志位 tx_flag 拉高
56              tx_data <= uart_din;            //寄存待发送的数据
57          end
58                                              //计数到停止位结束时，停止发送过程
59          else if ((tx_cnt == 4'd9) && (clk_cnt == BPS_CNT -(BPS_CNT/16))) begin
60              tx_flag <= 1'b0;                //发送过程结束，标志位 tx_flag 拉低
61              tx_data <= 8'd0;
62          end
63          else begin
64              tx_flag <= tx_flag;
65              tx_data <= tx_data;
66          end
67  end
68
69  //进入发送过程后，启动系统时钟计数器
```

```
70  always @(posedge sys_clk or negedge sys_rst_n) begin
71      if (!sys_rst_n)
72          clk_cnt <= 16'd0;
73      else if (tx_flag) begin                 //处于发送过程
74          if (clk_cnt < BPS_CNT - 1)
75              clk_cnt <= clk_cnt + 1'b1;
76          else
77              clk_cnt <= 16'd0;               //对系统时钟计数达一个波特率周期后清零
78      end
79      else
80          clk_cnt <= 16'd0;                   //发送过程结束
81  end
82
83  //进入发送过程后，启动发送数据计数器
84  always @(posedge sys_clk or negedge sys_rst_n) begin
85      if (!sys_rst_n)
86          tx_cnt <= 4'd0;
87      else if (tx_flag) begin                 //处于发送过程
88          if (clk_cnt == BPS_CNT - 1)         //对系统时钟计数达一个波特率周期
89              tx_cnt <= tx_cnt + 1'b1;        //此时发送数据计数器加 1
90          else
91              tx_cnt <= tx_cnt;
92      end
93      else
94          tx_cnt  <= 4'd0;                    //发送过程结束
95  end
96
97  //根据发送数据计数器来给 uart 发送端口赋值
98  always @(posedge sys_clk or negedge sys_rst_n) begin
99      if (!sys_rst_n)
100         uart_txd <= 1'b1;
101     else if (tx_flag)
102         case(tx_cnt)
103             4'd0: uart_txd <= 1'b0;         //起始位
104             4'd1: uart_txd <= tx_data[0];   //数据位最低位
105             4'd2: uart_txd <= tx_data[1];
106             4'd3: uart_txd <= tx_data[2];
107             4'd4: uart_txd <= tx_data[3];
108             4'd5: uart_txd <= tx_data[4];
109             4'd6: uart_txd <= tx_data[5];
110             4'd7: uart_txd <= tx_data[6];
```

```
111             4'd8: uart_txd <= tx_data[7];    //数据位最高位
112             4'd9: uart_txd <= 1'b1;          //停止位
113             default: ;
114         endcase
115     else
116         uart_txd <= 1'b1;                    //空闲时发送端口为高电平
117 end
118
119 endmodule
```

串口发送模块与串口接收模块异曲同工，代码中也给出了详尽的注释，此处不再赘述。需要我们特别注意的是代码的第 59 行，与接收模块不同，发送模块需要计数到停止位结束之后才能停止发送过程。即保证停止位的时间维持一个完整的波特率周期，否则在连续发送数据时，会出现由于停止位提前结束而导致上位机接收到错误的数据。

另外我们在代码的第 31 行将用于标志串口发送过程的 tx_flag 信号赋值给 uart_tx_busy，并通过模块端口输出。这样其他模块就可以通过检测 uart_tx_busy 信号是否为低电平，从而判断串口发送模块是否处于空闲状态。若 uart_tx_busy 为高电平，那么 uart_send 正处于发送过程，外部模块需要等待当前发送过程结束之后，才能通过 uart_en 信号的上升沿来启动新的发送过程。

图 14.4.5 为串口发送过程中 ILA 抓取的波形图，我们使用开发板通过串口向上位机向发送 16 进制数 55。图中绿色的触发线标识出了串口发送使能信号 uart_en 的上升沿。在检测到 uart_en 的上升沿后，en_flag 会拉高一个时钟周期，此时将 uart_din 端口上的待发送数据寄存到 tx_data 中，并进入串口发送过程。

在整个发送过程中 tx_flag 保持为高电平，tx_cnt 对串口数据进行计数，同时 tx_data 的各个数据位依次通过串口发送端 uart_txd 发送出去。当 tx_cnt 计数到 9 时，串口数据发送完成，开始发送停止位。在一个波特率周期的停止位发送完成后，串口发送过程结束，uart_tx_busy 信号拉低，表明串口发送模块进入空闲状态。

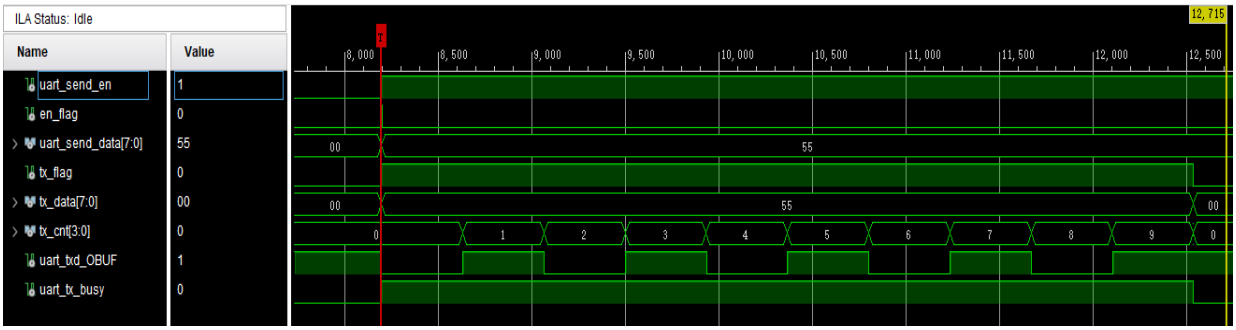


图 14.4.5 串口发送过程波形图

从图 14.4.5 中可以看出串口发送模块能够完成并串转换并正确发送串口数据。

在介绍完串口的接收和发送模块后，最后我们来看一下串口环回模块的代码：

```
1  module uart_loop(
2      input              sys_clk,                 //系统时钟
3      input              sys_rst_n,               //系统复位，低电平有效
4
5      input              recv_done,               //接收一帧数据完成标志
6      input      [7:0]   recv_data,               //接收的数据
```

```
7
8       input             tx_busy,                    //发送忙状态标志
9       output reg        send_en,                    //发送使能信号
10      output reg [7:0]  send_data                   //待发送数据
11      );
12
13 //reg define
14 reg recv_done_d0;
15 reg recv_done_d1;
16 reg tx_ready;
17
18 //wire define
19 wire recv_done_flag;
20
21 //*****************************************************
22 //**                    main code
23 //*****************************************************
24
25 //捕获 recv_done 上升沿，得到一个时钟周期的脉冲信号
26 assign recv_done_flag = (~recv_done_d1) & recv_done_d0;
27
28 //对发送使能信号 recv_done 延迟两个时钟周期
29 always @(posedge sys_clk or negedge sys_rst_n) begin
30     if (!sys_rst_n) begin
31         recv_done_d0 <= 1'b0;
32         recv_done_d1 <= 1'b0;
33     end
34     else begin
35         recv_done_d0 <= recv_done;
36         recv_done_d1 <= recv_done_d0;
37     end
38 end
39
40 //判断接收完成信号，并在串口发送模块空闲时给出发送使能信号
41 always @(posedge sys_clk or negedge sys_rst_n) begin
42     if (!sys_rst_n) begin
43         tx_ready  <= 1'b0;
44         send_en   <= 1'b0;
45         send_data <= 8'd0;
46     end
47     else begin
```

```
48          if(recv_done_flag)begin                 //检测串口接收到数据
49              tx_ready  <= 1'b1;                  //准备启动发送过程
50              send_en   <= 1'b0;
51              send_data <= recv_data;             //寄存串口接收的数据
52          end
53          else if(tx_ready && (~tx_busy)) begin   //检测串口发送模块空闲
54              tx_ready <= 1'b0;                   //准备过程结束
55              send_en  <= 1'b1;                   //拉高发送使能信号
56          end
57      end
58  end
59
60  endmodule
```

uart_loop 模块的代码比较简单，首先代码的 25 至 38 行实现了上升沿检测功能。当检测到 recv_done 信号的上升沿时，recv_done_flag 输出一个时钟周期的高电平，标志着串口接收模块接收到了一帧数据。在代码的 48 至 52 行，在判断到 recv_done_flag 为高电平时，寄存接收到的数据 recv_data 到 send_data 中；同时将 tx_ready 信号拉高，表示已经准备好了待发送的数据。另外还要将 send_en 信号拉低，为接下来产生一个上升沿作准备。

uart_loop 模块还有一个输入信号 tx_busy，它是由串口发送模块所输出，该信号为高电平表示串口发送模块正处于发送过程中。在代码的第 53 行，当 tx_ready 信号为高电平时，如果检测到 tx_busy 为低电平，则说明串口发送模块处于空闲状态。此时将 send_en 信号拉高，由此产生一个上升沿，以启动串口发送模块的发送过程，将寄存到 send_data 中的数据发送出去。于此同时，将 tx_ready 信号拉低，等待下一个串口接收数据的到来。

14.5 下载验证

将 ATK-USB-UART 模块连接到启明星 ZYNQ 开发板的“ATK MODULE”接口处，在这里直接将 ATK-USB-UART 模块的插针直接插到“ATK MODULE”接口的排母插口上即可，但是要注意插口的方向，千万不要插错方向，否则有可能烧毁器件！另外，由于“ATK MODULE”接口处的 VCC 是+5V，所以 ATK-USB-UART 模块的跳线帽要连接到“5V”。如下图所示：

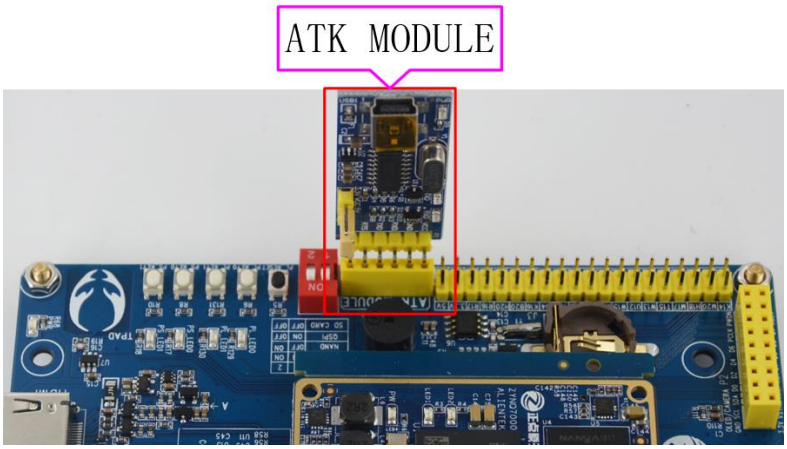


图 14.5.1 ATK MODULE 串口连接

如果用户使用了其他的 USB 转串口模块，则只需连接 3 根线即可：GND、UART_RX、UART_TX；“ATK MODULE”接口处的丝印所标注的是 Zynq 侧的内部信号名，用户要注意不要接错。

然后，将 ATK-USB-UART 模块的另一端连接到电脑的 USB 接口，然后开发板连接好电源线、下载电缆，并打开电源开关。

接下来分别连接 JTAG 接口和电源线，并打开电源开关。然后将“.bit”比特流配置文件下载到 Zynq 中。

注意上位机第一次使用 ATK-USB-UART 模块与开发板连接时，需要安装 USB 串口驱动。在开发板随附的资料中找到“6_软件资料/1_软件/CH340 驱动(USB 串口驱动)”的文件夹，双击打开文件夹中的“SETUP.EXE”进行安装，驱动安装界面如图 14.5.2 所示。界面中提示 INF 文件为 CH341SER.INF，我们不需要理会（CH341，CH340 驱动是共用的），直接点安装即可。

图 14.5.2 USB 串口驱动驱动安装界面

开发板电源打开后，将本次实验的 bit 文件下载到开发板中。

接下来打开串口助手。串口助手是上位机中用于辅助串口调试的小工具，可以选择安装使用开发板随附资料中“6_软件资料/1_软件/串口调试助手”文件夹中提供的 XCOM 串口助手。在上位机中打开串口助手 XCOM V2.0，如下图所示：

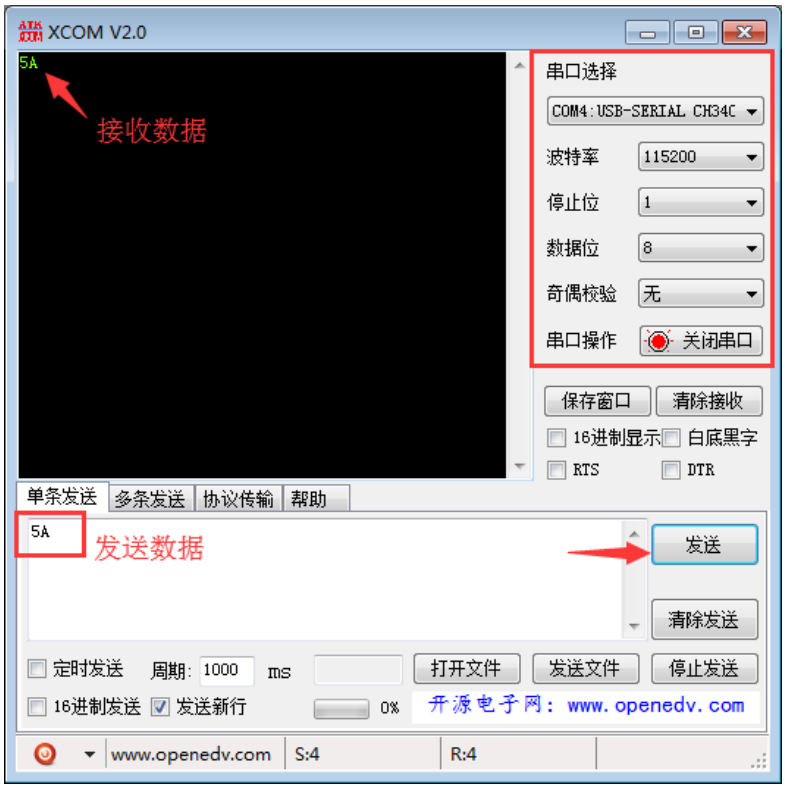


图 14.5.3 串口助手操作界面

在串口助手中选择与开发板相连接的 CH340 虚拟串口，具体的端口号（这里是 COM4）需要根据实际情况选择，可以在计算机设备器中查看，如下图所示：

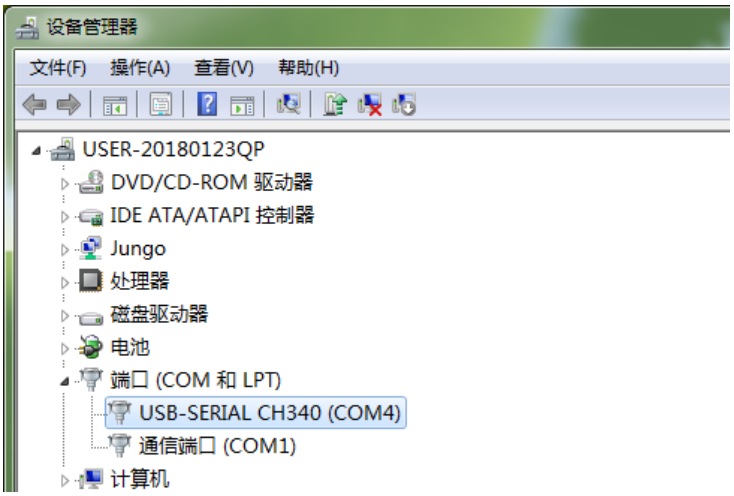


图 14.5.4 电脑的设备管理器窗口

在串口助手中设置波特率为 115200，数据位为 8，停止位为 1，无校验位，最后确认打开串口。

串口打开后，在发送文本框中输入数据“5A”并点击发送，可以看到串口助手中接收到数据“5A”，如图 14.5.3 所示。串口助手接收到的数据与发送的数据一致，说明程序所实现的串口数据环回功能验证成功。

第十五章 RGB TFT-LCD 彩条显示实验

TFT-LCD 是一种液晶显示屏，它采用薄膜晶体管（TFT）技术提升图像质量，如提高图像亮度和对比度等。相比于传统的 CRT 显示器，TFT-LCD 有着轻薄、功耗低、无辐射、图像质量好等诸多优点，因此广泛应用于电视机、电脑显示器、手机等各种显示设备中。

本章包括以下几个部分：

15.1 RGB TFT-LCD 简介

15.2 实验任务

15.3 硬件设计

15.4 程序设计

15.5 下载验证

15.1 RGB TFT-LCD 简介

TFT-LCD 的全称是 Thin Film Transistor-Liquid Crystal Display，即薄膜晶体管液晶显示屏，它显示的每个像素点都是由集成在液晶后面的薄膜晶体管独立驱动，因此 TFT-LCD 具有较高的响应速度以及较好的图像质量。正点原子推出的 RGB LCD 液晶屏较多，7 寸 RGB LCD 屏的实物图如下图所示：

图 15.1.1 ATK-7'RGB 接口 TFT 液晶屏模块

液晶显示器是现在最常用到的显示器，手机、电脑、各种人机交互设备等基本都用到了 LCD，最常见的就是手机和电脑显示器了。由于笔者不是 LCD 从业人员，对于 LCD 的具体原理不了解，百度百科对于 LCD 的原理解释如下：

LCD 的构造是在两片平行的玻璃基板当中放置液晶盒，下基板玻璃上设置 TFT（薄膜晶体管），上基板玻璃上设置彩色滤光片，通过 TFT 上的信号与电压改变来控制液晶分子的转动方向，从而达到控制每个像素点偏振光出射与否而达到显示目的。我们现在要在启明星开发板上使用 LCD，所以不需要去研究 LCD 的具体实现原理，我们只需要从使用的角度去关注 LCD 的几个重要点：

1、分辨率

提起 LCD 显示器，我们都会听到 720P、1080P、2K 或 4K 这样的字眼，这个就是 LCD 显示器分辨率。LCD 显示器都是由一个一个的像素点组成，像素点就类似一个灯(在 OLED 显示器中，像素点就是一个小灯)，这个小灯是 RGB 灯，也就是由 R(红色)、G(绿色)和 B(蓝色)这三种颜色组成的，而 RGB 就是光的三原色。1080P 的意思就是一个 LCD 屏幕上的像素数量是 1920*1080 个，也就是这个屏幕一列 1080 个像素点，一共 1920 列，如图 15.1.2 所示：

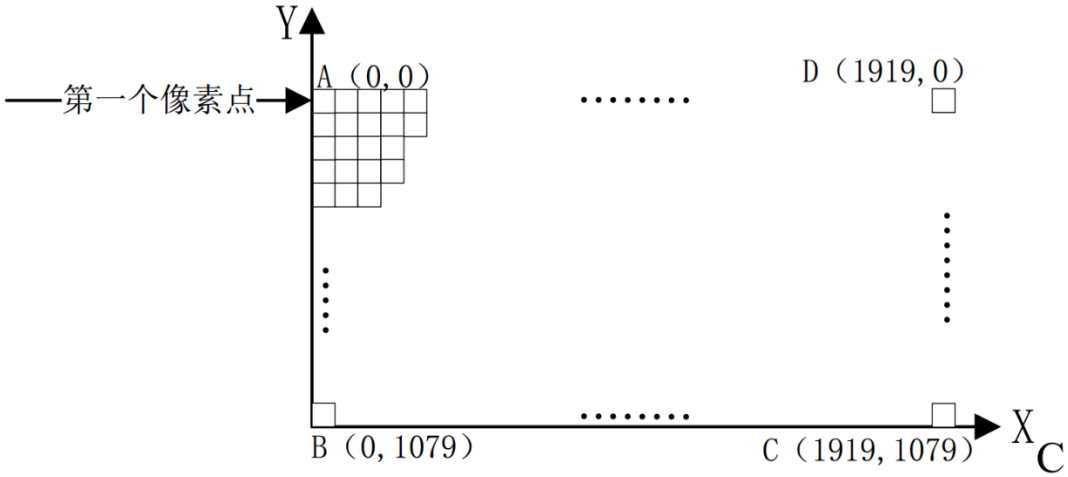


图 15.1.2 像素点排布

图 15.1.2 就是 1080P 显示器的像素示意图，X 轴就是 LCD 显示器的横轴，Y 轴就是显示器的竖轴。图

中的小方块就是像素点，一共有 1920*1080=2073600 个像素点。左上角的 A 点是第一个像素点，右下角的 C 点就是最后一个像素点。2K 就是 2560*1440 个像素点，4K 是 3840*2160 个像素点。很明显，在 LCD 尺寸不变的情况下，分辨率越高越清晰。同样的，分辨率不变的情况下，LCD 尺寸越小越清晰。比如我们常用的 24 寸显示器基本都是 1080P 的，而我们现在使用的 5 寸的手机基本也是 1080P 的，但是手机显示细腻程度就要比 24 寸的显示器要好很多！由此可见，LCD 显示器的分辨率是一个很重要的参数，但是并不是分辨率越高的 LCD 就越好。衡量一款 LCD 的好坏，分辨率只是其中的一个参数，还有色彩还原程度、色彩偏离、亮度、可视角度、屏幕刷新率等其他参数。

2、像素格式

上面讲了，一个像素点就相当于一个 RGB 小灯，通过控制 R、G、B 这三种颜色的亮度就可以显示出各种各样的色彩。那该如何控制 R、G、B 这三种颜色的显示亮度呢？一般一个 R、G、B 这三部分分别使用 8bit 的数据，那么一个像素点就是 8bit*3=24bit，也就是说一个像素点 3 个字节，这种像素格式称为 RGB888。当然常用的像素点格式还有 RGB565，只需要两个字节，但在色彩鲜艳度上较差一些。我们启明星开发板上的 RGB TFT-LCD 接口采用的 RGB888 的像素格式，共需要 24 位，每一位对应 RGB 的颜色分量如下图所示：

D23	D22	D21	D20	D19	D18	D17	D16	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
R7	R6	R5	R4	R3	R2	R1	R0	G7	G6	G5	G4	G3	G2	G1	G0	B7	B6	B5	B4	B3	B2	B1	B0

图 15.1.3 RGB888 数据格式

在图 15.1.3 中，一个像素点占用 3 个字节，其中 bit23~bit16 是 RED 通道，bit15~bit14 是 GREEN 通道，bit7~bit0 是 BLUE 通道。所以红色对应的值就是 24'hFF0000，蓝色对应的值就是 24'h00FF00，绿色对应的值为 24'h0000FF。通过调节 R、G、B 的比例可以产生其它的颜色，比如 24'hFFFF00 就是黄色，24'h000000 就是黑色，24'hFFFFFF 就是白色。大家可以打开电脑的“画图”工具，在里面使用调色板即可获取到想要的颜色对应的数值，如图 15.1.4 所示：

图 15.1.4 调色板颜色选取

3、LCD 屏幕接口

LCD 屏幕或者说显示器有很多种接口，比如在显示器上常见的 VGA、HDMI、DP 等等，启明星开发板支持 RGB 接口的 LCD 和 HDMI 接口的显示器。本章我们介绍的是 RGB LCD 接口，RGB LCD 接口的信号线如下表所示：

表 15.1.1 RGB 数据线

信号线	描述
R[7:0]	8根红色数据线
G[7:0]	8根绿色数据线
B[7:0]	8根蓝色数据线
DE	数据使能线
VSYNC	垂直同步信号线
HSYNC	水平同步信号线
PCLK	像素时钟信号线

表 15.1.1 就是 RGB LCD 的信号线，R[7:0]、G[7:0]和 B[7:0]是 24 位数据，DE、VSYNC、HSYNC 和 PCLK 是四个控制信号。

正点原子一共有五款 RGB LCD 屏幕，型号分别为 ATK-4342（4.3 寸，480*272）、ATK-4384（4.3 寸，800*480）、ATK-7084（7 寸，800*480）、ATK-7016（7 寸，1024*600）和 ATK-1018（10.1 寸，1280*800）。这里以 ATK-7016 这款屏幕为例讲解，ATK-7016 的屏幕接口原理图如所示：

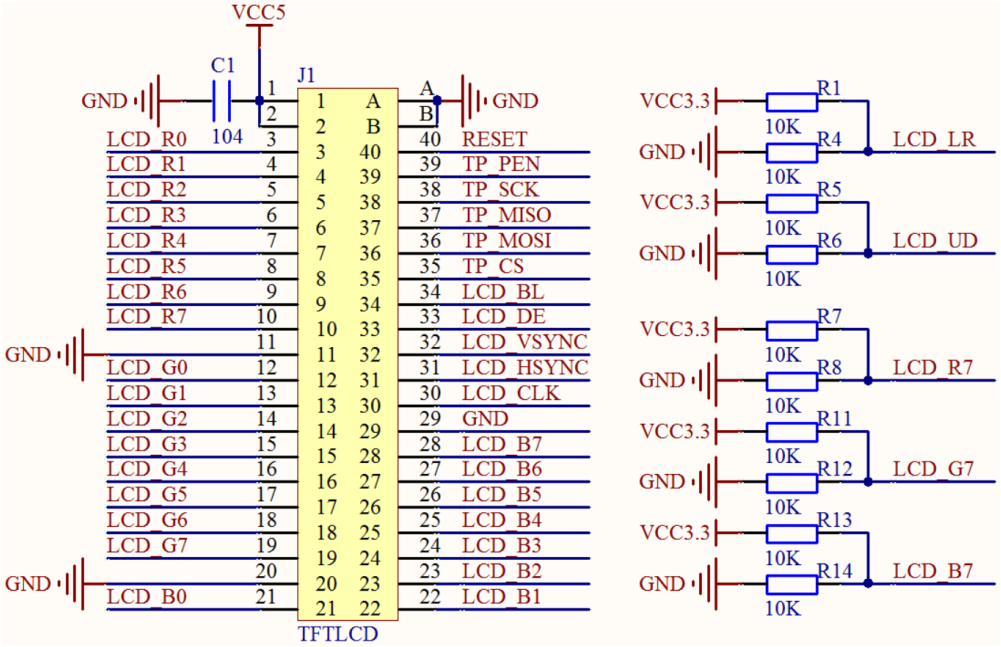


图 15.1.5 RGB LCD 液晶屏幕接口

图中 J1 就是对外接口，是一个 40PIN 的 FPC 座（0.5mm 间距），通过 FPC 线，可以连接到启明星 ZYNQ 开发板上面，从而实现和开发板的连接。该接口十分完善，采用 RGB888 格式，并支持触摸屏和背光控制。右侧的几个电阻，并不是都焊接的，而是可以用户自己选择。默认情况，R1 和 R6 焊接，设置 LCD_LR 和 LCD_UD，控制 LCD 的扫描方向，是从左到右，从上到下（横屏看）。而 LCD_R7/G7/B7 则用来设置 LCD 的 ID，由于 RGBLCD 没有读写寄存器，也就没有所谓的 ID，这里我们通过在模块上面，控制 R7/G7/B7 的

上/下拉，来自定义 LCD 模块的 ID，帮助 MCU 判断当前 LCD 面板的分辨率和相关参数，以提高程序兼容性。这几个位的设置关系如表 15.1.2 所示：

表 15.1.2 RGB LCD 模块和 ID 对应关系

M2 LCD_B7	M1 LCD_G7	M0 LCD_R7	LCD ID	说明
0	0	0	4342	ATK-4342 RGB LCD模块，分辨率：480*272
0	0	1	7084	ATK-7084 RGB LCD模块，分辨率：800*480
0	1	0	7016	ATK-7016 RGB LCD模块，分辨率：1024*600
1	0	0	4384	ATK-4384 RGB LCD模块，分辨率：800*480
1	0	1	1018	ATK-1018 RGB LCD模块，分辨率：1280*800
X	X	X	NC	暂时未用到

ATK-7016 模块，就设置 M2:M0 = 010 即可。这样，我们在程序里面，读取 LCD_R7/G7/B7，得到 M0:M2 的值，从而判断 RGB LCD 模块的型号，并执行不同的配置，即可实现不同 LCD 模块的兼容。

4、LCD 时间参数

如果将 LCD 显示一帧图像的过程想象成绘画，那么在显示的过程中就是用一根“笔”在不同的像素点画上不同的颜色。这根笔按照从左至右、从上到下的顺序扫描每个像素点，并且在像素画上对应的颜色，当画到最后一个像素点的时候一幅图像就绘制好了。假如一个 LCD 的分辨率为 1024*600，那么其扫描如图 15.1.6 所示：

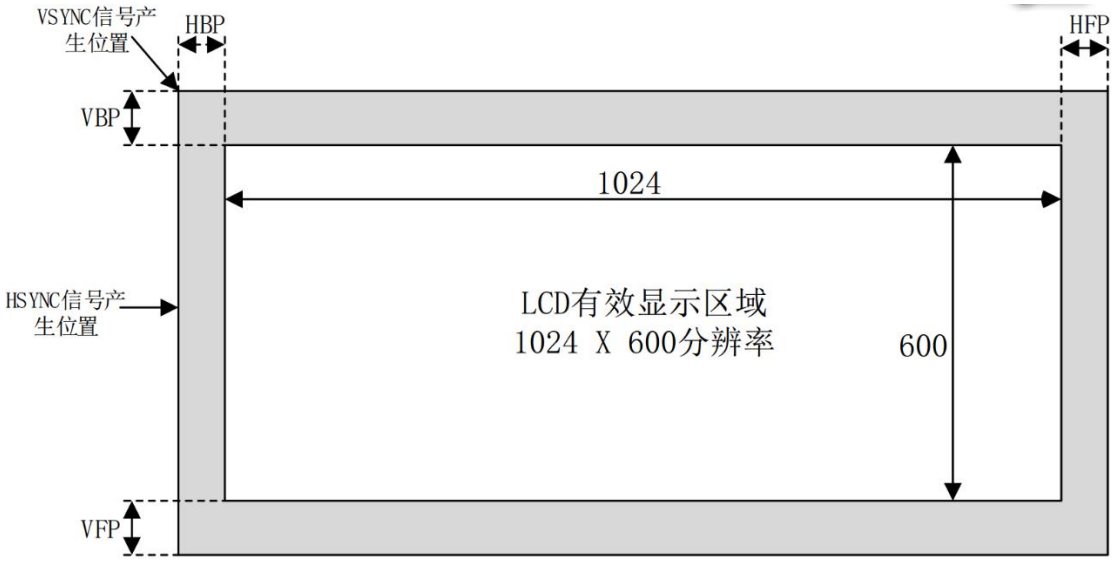


图 15.1.6 LCD 一帧图像扫描图

结合图 15.1.6 我们来看一下 LCD 是怎么扫描显示一帧图像的。一帧图像也是由一行一行组成的。HSYNC 是水平同步信号，也叫做行同步信号，当产生此信号的话就表示开始显示新的一行了，所以此信号都是在图 15.1.6 的最左边。VSYNC 信号是垂直同步信号，也叫做帧同步信号，当产生此信号的话就表示开始显示新的一帧图像了，所以此信号在图 15.1.6 的左上角。

在图 15.1.6 可以看到有一圈“黑边”，真正有效的显示区域是中间的白色部分。那这一圈“黑边”是什么东西呢？这就要从显示器的“祖先”CRT 显示器开始说起了，CRT 显示器就是以前很常见的那种大屁股显示器，在 2019 年应该很少见了，如果在农村应该还是可以见到的。CRT 显示器屁股后面是个电子枪，

这个电子枪就是我们上面说的“画笔”，电子枪打出的电子撞击到屏幕上的荧光物质使其发光。只要控制电子枪从左到右扫完一行(也就是扫描一行)，然后从上到下扫描完所有行，这样一帧图像就显示出来了。也就是说，显示一帧图像电子枪是按照‘Z’形在运动，当扫描速度很快的时候看起来就是一幅完成的画面了。

当显示完一行以后会发出 HSYNC 信号，此时电子枪就会关闭，然后迅速的移动到屏幕的左边，当 HSYNC 信号结束以后就可以显示新的一行数据了，电子枪就会重新打开。在 HSYNC 信号结束到电子枪重新打开之间会插入一段延时，这段延时就是图 15.1.6 中的 HBP。当显示完一行以后就会关闭电子枪等待 HSYNC 信号产生，关闭电子枪到 HSYNC 信号产生之间会插入一段延时，这段延时就是图 15.1.6 中的 HFP 信号。同理，当显示完一帧图像以后电子枪也会关闭，然后等到 VSYNC 信号产生，期间也会加入一段延时，这段延时就是图 15.1.6 中的 VFP。VSYNC 信号产生，电子枪移动到左上角，当 VSYNC 信号结束以后电子枪重新打开，中间也会加入一段延时，这段延时就是图 15.1.6 中的 VBP。

HBP、HFP、VBP 和 VFP 就是导致图 15.1.6 中黑边的原因，但是这是 CRT 显示器存在黑边的原因，现在是 LCD 显示器，不需要电子枪了，那么为何还会有黑边呢？这是因为 RGB LCD 屏幕内部是有一个 IC 的，发送一行或者一帧数据给 IC，IC 是需要反应时间的。通过这段反应时间可以让 IC 识别到一行数据扫描完了，要换行了，或者一帧图像扫描完了，要开始下一帧图像显示了。因此，在 LCD 屏幕中继续存在 HBP、HFP、VPB 和 VFP 这四个参数的主要目的是为了锁定有效的像素数据。

5、RGB LCD 屏幕时序

上面介绍了 LCD 的时间参数，我们接下来看一下行显示对应的时序图，如图 15.1.7 所示。

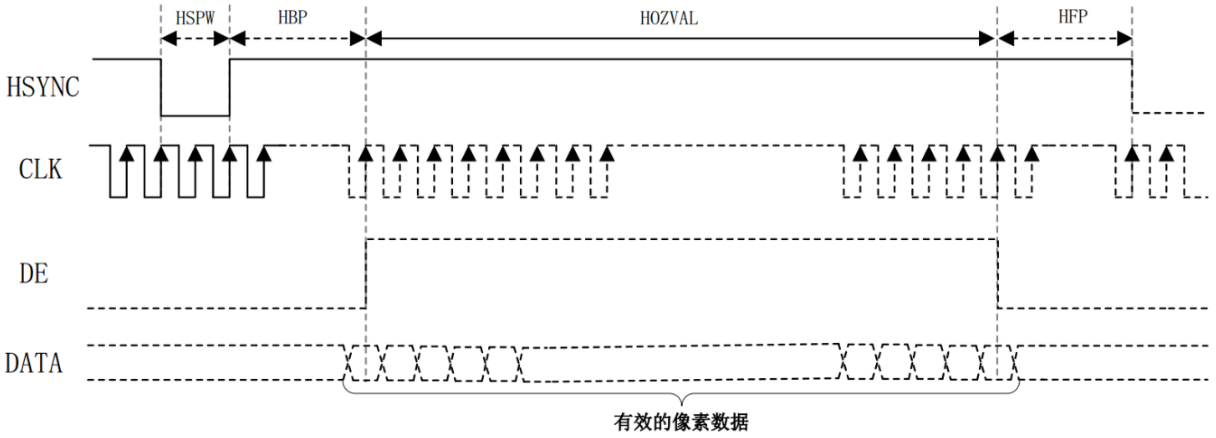


图 15.1.7 LCD 行显示时序

图 15.1.7 就是 RGB LCD 的行显示时序，我们来分析一下其中重要的几个参数：

HSYNC：行同步信号，当此信号有效的时候就表示开始显示新的一行数据，查阅所使用的 LCD 数据手册可以知道此信号是低电平有效还是高电平有效，图 15.1.7 为低电平有效。

HSPW：行同步信号宽度，也就是 HSYNC 信号持续时间。HSYNC 信号不是一个脉冲，而是需要持续一段时间才是有效的，单位为 CLK。

HBP：行显示后沿（或后肩），单位是 CLK。

HOZVAL：行有效显示区域，即显示一行数据所需的时间，假如屏幕分辨率为 1024*600，那么 HOZVAL 就是 1024，单位为 CLK。

HFP：行显示前沿（或前肩），单位是 CLK。

当 HSYNC 信号发出以后，需要等待 HSPW+HBP 个 CLK 时间才会接收到真正有效的像素数据。当显示完一行数据以后需要等待 HFP 个 CLK 时间才能发出下一个 HSYNC 信号，所以显示一行所需要的时间就是：HSPW + HBP + HOZVAL + HFP。

一帧图像就是由很多个行组成的，RGB LCD 的帧显示时序如图 15.1.8 所示：

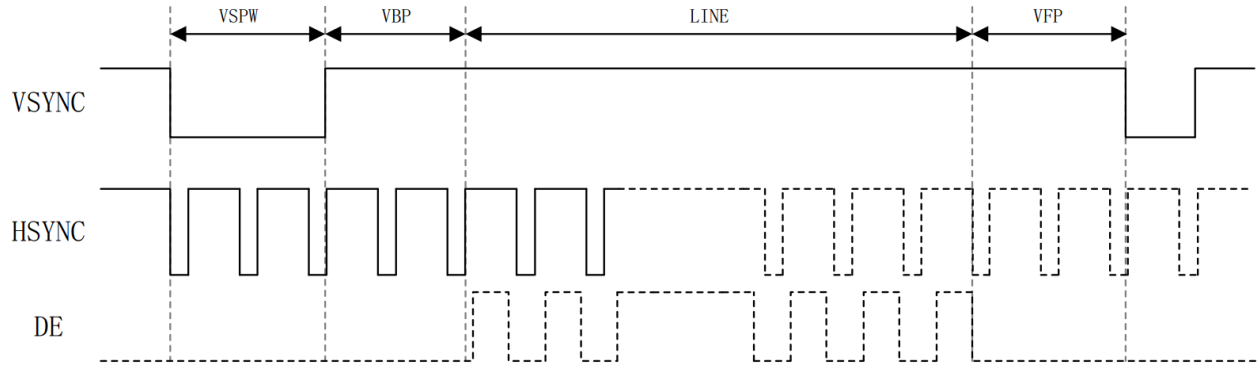


图 15.1.8 LCD 帧显示时序

图 15.1.8 就是 RGB LCD 的帧显示时序，我们来分析一下其中重要的几个参数：

VSYNC：帧（场）同步信号，当此信号有效的时候就表示开始显示新的一帧数据，查阅所使用的 LCD 数据手册可以知道此信号是低电平有效还是高电平有效，图 15.1.8 为低电平有效。

VSPW：帧同步信号宽度，也就是 VSYNC 信号持续时间，单位为 1 行的时间。

VBP：帧显示后沿（或后肩），单位为 1 行的时间。

LINE：帧有效显示区域，即显示一帧数据所需的时间，假如屏幕分辨率为 1024*600，那么 LINE 就是 600 行的时间。

VFP：帧显示前沿（或前肩），单位为 1 行的时间。

显示一帧所需要的时间就是：VSPW+VBP+LINE+VFP 个行时间，最终的计算公式：

T = (VSPW+VBP+LINE+VFP) * (HSPW + HBP + HOZVAL + HFP)

因此我们在配置一款 RGB LCD 屏的时候需要知道这几个参数：HSPW（行同步）、HBP（行显示后沿）、HOZVAL(行有效显示区域)、HFP（行显示前沿）、VSPW（场同步）、VBP（场显示后沿）、LINE(场有效显示区域)和 VFP（场显示后沿）。

RGB LCD 液晶屏一般有两种数据同步方式，一种是行场同步模式（HV Mode），另一种是数据使能同步模式（DE Mode）。当选择行场同步模式时，LCD 接口的时序与 VGA 接口的时序图非常相似，只是参数不同，如图 16.1.9 和图 16.1.8 中的行同步信号（HSYNC）和场同步信号（VSYNC）作为数据的同步信号，此时数据使能信号（DE）必须为低电平。

当选择 DE 同步模式时，LCD 的 DE 信号作为数据的有效信号，，如图 16.1.10 和图 16.1.8 中的 DE 信号所示。只有同时扫描到帧有效显示区域和行有效显示区域时，DE 信号才有效（高电平）。当选择 DE 同步模式时，此时行场同步信号 VS 和 HS 必须为高电平。

由于 RGB LCD 液晶屏一般都支持 DE 模式，不是所有的 RGB LCD 液晶屏都支持 HV 模式，因此本章我们采用 DE 同步的方式驱动 LCD 液晶屏。

6、像素时钟

像素时钟就是 RGB LCD 的时钟信号，以 ATK7016 这款屏幕为例，显示一帧图像所需要的时钟数就是：

N（CLK）= (VSPW+VBP+LINE+VFP) * (HSPW + HBP + HOZVAL + HFP)

= (3 + 20 + 600 + 12) * (20 + 140 + 1024 + 160) = 635 * 1344 = 853440

显示一帧图像需要 853440 个时钟数，那么显示 60 帧就是：853440 * 60 = 51206400≈51.2M，所以像素时钟就是 51.2MHz。

当然我们在为 RGB LCD 屏提供驱动时钟的时候，也可以不用严格按照 60 帧来进行计算。为了方便操作，我们可以给 ATK7016 模块输出一个 50MHz 的时钟，其刷新率是接近于 60Hz 的，同时也非常方便我们

来编写代码。

为了方便大家查找 LCD 屏的时序参数，这里整理了不同分辨率的时序参数，如表 15.1.3 所示：

表 15.1.3 不同分辨率的LCD时序参数

显示分辨率	时钟（Mhz）	行时序（像素数）					帧时序（行数）				
		行同步	显示后沿	显示区域	显示前沿	显示周期	场同步	显示后沿	显示区域	显示前沿	显示周期
480*272	9	41	2	480	2	525	10	2	272	2	286
800*480	33.3	128	88	800	40	1056	2	33	480	10	525
1024*600	50	20	140	1024	160	1344	3	20	600	12	635
1280*800	70	10	80	1280	70	1440	3	10	800	10	823

15.2 实验任务

本节的实验任务是使用正点原子 ZYNQ 开发板上的 RGB TFT-LCD 接口，驱动 RGB LCD 液晶屏（支持目前推出的所有 RGB LCD 屏），并显示出彩条。

15.3 硬件设计

启明星开发板上 RGB TFT-LCD 接口部分的原理图如下图所示。

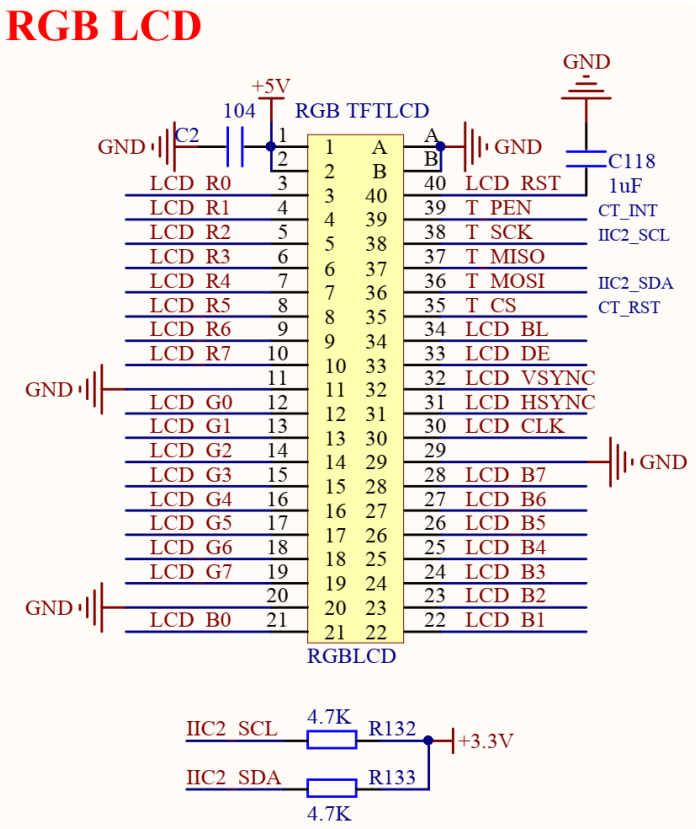


图 15.3.1 RGB TFTLCD 接口原理图

从上图中可以看到，FPGA 管脚输出的颜色数据位宽为 24bit，数据格式为 RGB888，即数据高 8 位表示红色，中间 8 位表示绿色，低 8 位表示蓝色。由于这 24 位数据不仅仅作为输出给 LCD 屏的颜色数据，

同时 LCD_R7、LCD_G7 和 LCD_B7 也用来获取 LCD 屏的 ID，因此这 24 位颜色数据对 ZYNQ 开发板来说，是一个双向的引脚。

另外，RGBLCD 模块支持触摸功能，图中以字母 T 开头的 5 个信号（T_PEN、T_SCK 等）与模块上的触摸芯片相连接。由于本次实验不涉及触摸功能的实现，因此这些信号并未用到。

需要说明的是，LCD 液晶屏有一个复位信号（LCD_RST），当 LCD_RST 为低电平时，可对 LCD 屏进行复位。由于受限于 ZYNQ PL 端 IO 引脚的数量，在硬件电路上，将 LCD 的复位信号和 PL 端的复位按键（PL_RESET）直连在一起，因此代码中不必对 LCF_RST 进行控制，而是通过开发板上的 PL 复位按键对 LCD 屏进行复位。

本实验中，各端口信号的管脚分配如下表所示：

表 15.3.1 RGB TFT-LCD彩条实验管脚分配

信号名	方向	管脚	端口说明	电平标准
sys_clk	input	U18	系统时钟，50M	LVCMOS33
sys_rst_n	input	J15	系统复位，低有效	LVCMOS33
lcd_rgb[0]	inout	Y18	RGB LCD蓝色（最低位）	LVCMOS33
lcd_rgb[1]	inout	Y19	RGB LCD蓝色	LVCMOS33
lcd_rgb[2]	inout	W20	RGB LCD蓝色	LVCMOS33
lcd_rgb[3]	inout	V20	RGB LCD蓝色	LVCMOS33
lcd_rgb[4]	inout	U14	RGB LCD蓝色	LVCMOS33
lcd_rgb[5]	inout	U15	RGB LCD蓝色	LVCMOS33
lcd_rgb[6]	inout	T20	RGB LCD蓝色	LVCMOS33
lcd_rgb[7]	inout	U20	RGB LCD蓝色（最高位）	LVCMOS33
lcd_rgb[8]	inout	W14	RGB LCD绿色（最低位）	LVCMOS33
lcd_rgb[9]	inout	Y14	RGB LCD绿色	LVCMOS33
lcd_rgb[10]	inout	N15	RGB LCD绿色	LVCMOS33
lcd_rgb[11]	inout	N16	RGB LCD绿色	LVCMOS33
lcd_rgb[12]	inout	V16	RGB LCD绿色	LVCMOS33
lcd_rgb[13]	inout	W16	RGB LCD绿色	LVCMOS33
lcd_rgb[14]	inout	W18	RGB LCD绿色	LVCMOS33
lcd_rgb[15]	inout	W19	RGB LCD绿色（最高位）	LVCMOS33
lcd_rgb[16]	inout	T10	RGB LCD红色（最低位）	LVCMOS33
lcd_rgb[17]	inout	T11	RGB LCD红色	LVCMOS33

lcd_rgb[18]	inout	P14	RGB LCD红色	LVCMOS33
lcd_rgb[19]	inout	R14	RGB LCD红色	LVCMOS33
lcd_rgb[20]	inout	V13	RGB LCD红色	LVCMOS33
lcd_rgb[21]	inout	U13	RGB LCD红色	LVCMOS33
lcd_rgb[22]	inout	G15	RGB LCD红色	LVCMOS33
lcd_rgb[23]	inout	H15	RGB LCD红色（最高位）	LVCMOS33
lcd_hs	output	U17	RGB LCD行同步	LVCMOS33
lcd_vs	output	P20	RGB LCD场同步	LVCMOS33
lcd_de	output	N20	RGB LCD数据使能	LVCMOS33
lcd_bl	output	Y16	RGB LCD背光控制	LVCMOS33
lcd_clk	output	T16	RGB LCD像素时钟	LVCMOS33

对应的 XDC 约束语句如下所示：

```
set_property -dict {PACKAGE_PIN U18 IOSTANDARD LVCMOS33} [get_ports sys_clk]
set_property -dict {PACKAGE_PIN J15 IOSTANDARD LVCMOS33} [get_ports sys_rst_n]

set_property -dict {PACKAGE_PIN Y18 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[0]}]
set_property -dict {PACKAGE_PIN Y19 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[1]}]
set_property -dict {PACKAGE_PIN W20 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[2]}]
set_property -dict {PACKAGE_PIN V20 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[3]}]
set_property -dict {PACKAGE_PIN U14 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[4]}]
set_property -dict {PACKAGE_PIN U15 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[5]}]
set_property -dict {PACKAGE_PIN T20 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[6]}]
set_property -dict {PACKAGE_PIN U20 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[7]}]
set_property -dict {PACKAGE_PIN W14 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[8]}]
set_property -dict {PACKAGE_PIN Y14 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[9]}]
set_property -dict {PACKAGE_PIN N15 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[10]}]
set_property -dict {PACKAGE_PIN N16 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[11]}]
set_property -dict {PACKAGE_PIN V16 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[12]}]
set_property -dict {PACKAGE_PIN W16 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[13]}]
set_property -dict {PACKAGE_PIN W18 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[14]}]
set_property -dict {PACKAGE_PIN W19 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[15]}]
set_property -dict {PACKAGE_PIN T10 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[16]}]
set_property -dict {PACKAGE_PIN T11 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[17]}]
set_property -dict {PACKAGE_PIN P14 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[18]}]
set_property -dict {PACKAGE_PIN R14 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[19]}]
set_property -dict {PACKAGE_PIN V13 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[20]}]
set_property -dict {PACKAGE_PIN U13 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[21]}]
```

```
set_property -dict {PACKAGE_PIN G15 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[22]}]
set_property -dict {PACKAGE_PIN H15 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[23]}]

set_property -dict {PACKAGE_PIN U17 IOSTANDARD LVCMOS33} [get_ports lcd_hs]
set_property -dict {PACKAGE_PIN P20 IOSTANDARD LVCMOS33} [get_ports lcd_vs]
set_property -dict {PACKAGE_PIN N20 IOSTANDARD LVCMOS33} [get_ports lcd_de]
set_property -dict {PACKAGE_PIN Y16 IOSTANDARD LVCMOS33} [get_ports lcd_bl]
set_property -dict {PACKAGE_PIN T16 IOSTANDARD LVCMOS33} [get_ports lcd_clk]
```

15.4 程序设计

RGB TFT-LCD 输入时序包含三个要素：像素时钟、同步信号、以及图像数据，由此我们可以大致规划出系统结构如下图所示。其中，读取 ID 模块用于获取 LCD 屏的 ID；由于不同分辨率的屏幕需要不同的驱动时钟，因此时钟分频模块根据 LCD ID 来输出不同频率的像素时钟；LCD 显示模块负责产生液晶屏上显示的数据，即彩条数据；LCD 驱动模块根据 LCD 屏的 ID，输出不同参数的时序，来驱动 LCD 屏，并将输入的彩条数据显示到 LCD 屏上。

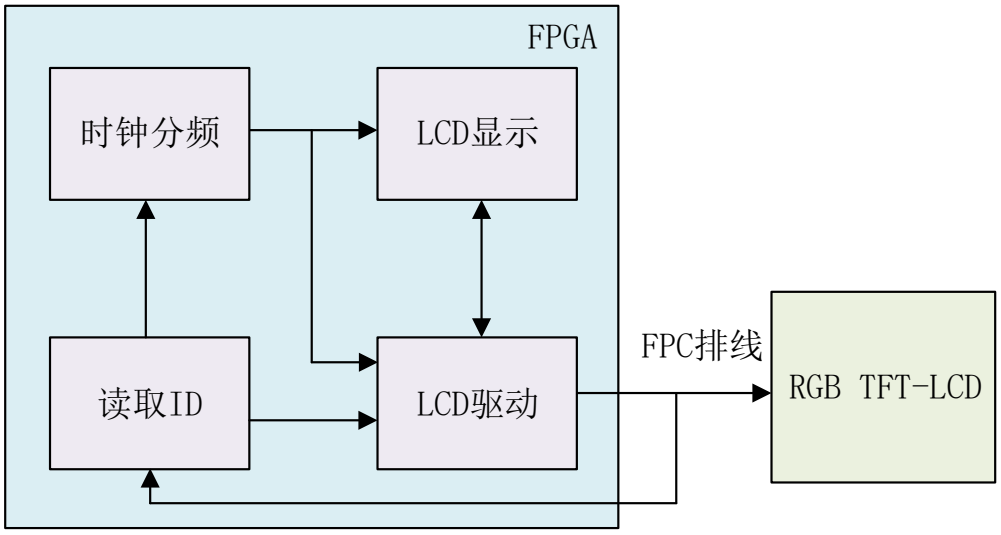


图 15.4.1 RGB TFT-LCD 彩条显示系统框图

由系统框图可知，FPGA 部分包括五个模块，顶层模块（lcd_rgb_colorbar）、读取 ID 模块（rd_id）、时钟分频模块（clk_div）、LCD 显示模块（lcd_display）以及 LCD 驱动模块（lcd_driver）。其中在顶层模块中完成对其余模块的例化。

各模块端口及信号连接如下图所示：

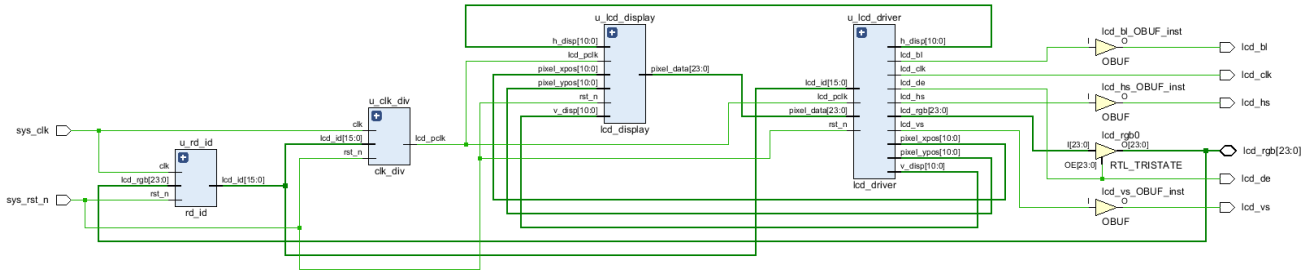


图 15.4.2 顶层模块原理图

读取 ID 模块（rd_id）在上电时将 RGB 双向数据总线设置为输入，来读取 LCD 屏的 ID；时钟分频模块（clk_div）根据读取的 ID 来配置 LCD 的像素时钟；LCD 驱动模块（lcd_driver）在像素时钟的驱动下输出数据使能信号用于数据同步，同时还输出像素点的纵横坐标，供 LCD 显示模块（lcd_display）调用，以绘制彩条图案。

顶层模块 lcd_rgb_colorbar 的代码如下：

```
module lcd_rgb_colorbar(
    input                sys_clk,     //系统时钟
    input                sys_rst_n,   //系统复位

    //RGB LCD 接口
    output               lcd_de,      //LCD 数据使能信号
    output               lcd_hs,      //LCD 行同步信号
    output               lcd_vs,      //LCD 场同步信号
    output               lcd_bl,      //LCD 背光控制信号
    output               lcd_clk,     //LCD 像素时钟
    inout        [23:0]  lcd_rgb      //LCD RGB888 颜色数据
    );

//wire define
wire  [15:0]  lcd_id    ;    //LCD 屏 ID
wire          lcd_pclk  ;    //LCD 像素时钟

wire  [10:0]  pixel_xpos;    //当前像素点横坐标
wire  [10:0]  pixel_ypos;    //当前像素点纵坐标
wire  [10:0]  h_disp    ;    //LCD 屏水平分辨率
wire  [10:0]  v_disp    ;    //LCD 屏垂直分辨率
wire  [23:0]  pixel_data;    //像素数据
wire  [23:0]  lcd_rgb_o ;    //输出的像素数据
wire  [23:0]  lcd_rgb_i ;    //输入的像素数据

//*****************************************************
//**                    main code
//*****************************************************

//像素数据方向切换
assign lcd_rgb = lcd_de ?  lcd_rgb_o :  {24{1'bz}};
assign lcd_rgb_i = lcd_rgb;

//读 LCD ID 模块
rd_id u_rd_id(
    .clk          (sys_clk  ),
```

```
37      .rst_n          (sys_rst_n),
38      .lcd_rgb        (lcd_rgb_i),
39      .lcd_id         (lcd_id    )
40      );
41
42  //时钟分频模块
43  clk_div u_clk_div(
44      .clk            (sys_clk  ),
45      .rst_n          (sys_rst_n),
46      .lcd_id         (lcd_id   ),
47      .lcd_pclk       (lcd_pclk )
48      );
49
50  //LCD 显示模块
51  lcd_display u_lcd_display(
52      .lcd_pclk       (lcd_pclk  ),
53      .rst_n          (sys_rst_n ),
54      .pixel_xpos     (pixel_xpos),
55      .pixel_ypos     (pixel_ypos),
56      .h_disp         (h_disp    ),
57      .v_disp         (v_disp    ),
58      .pixel_data     (pixel_data)
59      );
60
61  //LCD 驱动模块
62  lcd_driver u_lcd_driver(
63      .lcd_pclk       (lcd_pclk  ),
64      .rst_n          (sys_rst_n ),
65      .lcd_id         (lcd_id    ),
66      .pixel_data     (pixel_data),
67      .pixel_xpos     (pixel_xpos),
68      .pixel_ypos     (pixel_ypos),
69      .h_disp         (h_disp    ),
70      .v_disp         (v_disp    ),
71
72      .lcd_de         (lcd_de    ),
73      .lcd_hs         (lcd_hs    ),
74      .lcd_vs         (lcd_vs    ),
75      .lcd_bl         (lcd_bl    ),
76      .lcd_clk        (lcd_clk   ),
77      .lcd_rgb        (lcd_rgb_o )
```

```
78      );
79
80  endmodule
```

顶层模块主要完成对其他模块的例化。这里需要重点注意第 31 行代码，由于 lcd_rgb 是 24 位的双向引脚，所以这里对双向引脚的方向做一个切换。当 lcd_de 信号为高电平时，此时输出的像素数据有效，将 lcd_rgb 的引脚方向切换成输出，并将 LCD 驱动模块输出的 lcd_rgb_o（像素数据）连接至 lcd_rgb 引脚；当 lcd_de 信号为低电平时，此时输出的像素数据无效，将 lcd_rgb 的引脚方向切换成输入。代码中将高阻状态“Z”赋值给 lcd_rgb 的引脚，表示此时 lcd_rgb 的引脚电平由外围电路决定，此时可以读取 lcd_rgb 的引脚电平，从而获取到 LCD 屏的 ID。

读取 ID 模块的代码如下：

```
1   module rd_id(
2       input                   clk    ,    //时钟
3       input                   rst_n  ,    //复位，低电平有效
4       input           [23:0]  lcd_rgb,    //RGB LCD 像素数据,用于读取 ID
5       output   reg    [15:0]  lcd_id      //LCD 屏 ID
6       );
7
8   //reg define
9   reg             rd_flag;                //读 ID 标志
10
11  //*****************************************************
12  //**                    main code
13  //*****************************************************
14
15  //获取 LCD ID   M2:B7  M1:G7  M0:R7
16  always @(posedge clk or negedge rst_n) begin
17      if(!rst_n) begin
18          rd_flag <= 1'b0;
19          lcd_id <= 16'd0;
20      end
21      else begin
22          if(rd_flag == 1'b0) begin
23              rd_flag <= 1'b1;
24              case({lcd_rgb[7],lcd_rgb[15],lcd_rgb[23]})
25                  3'b000 : lcd_id <= 16'h4342;    //4.3' RGB LCD  RES:480x272
26                  3'b001 : lcd_id <= 16'h7084;    //7'   RGB LCD  RES:800x480
27                  3'b010 : lcd_id <= 16'h7016;    //7'   RGB LCD  RES:1024x600
28                  3'b100 : lcd_id <= 16'h4384;    //4.3' RGB LCD  RES:800x480
29                  3'b101 : lcd_id <= 16'h1018;    //10'  RGB LCD  RES:1280x800
30                  default : lcd_id <= 16'd0;
31              endcase
```

```
        end
    end
end

endmodule
```

读取 ID 模块根据输入的 lcd_rgb 值来寄存 LCD 屏的 ID。lcd_rgb[7]（B7）、lcd_rgb[15]（G7）和 lcd_rgb[23]（R7）分别对应 M2、M1 和 M0。尽管在顶层模块中，双向引脚 lcd_rgb 会根据 lcd_de 信号的高低电平来频繁的切换方向，但本模块实际上只在上电后获取了一次 ID，通过 rd_flag 作为标志。当 rd_flag 等于 0 时，获取一次 ID，并将 rd_flag 赋值为 1；而当 rd_flag 等于 1 时，不再获取 LCD 屏的 ID。

除此之外，为了方便将 LCD 的 ID 和分辨率对应起来，这里对 M2、M1 和 M0 的值做了一个译码，如 3'b000 译码成 16'h4342，表示当前连接的是 4.3 寸屏，分辨率为 480*272。其它 ID 的译码请参考注释。

分频模块的代码如下：

```
module clk_div(
    input               clk,          //50Mhz
    input               rst_n,
    input       [15:0]  lcd_id,
    output  reg         lcd_pclk
    );

reg         clk_25m;
reg         clk_12_5m;
reg         div_4_cnt;

//时钟 2 分频 输出 25MHz 时钟
always @(posedge clk or negedge rst_n) begin
    if(!rst_n)
        clk_25m <= 1'b0;
    else
        clk_25m <= ~clk_25m;
end

//时钟 4 分频 输出 12.5MHz 时钟
always @(posedge clk or negedge rst_n) begin
    if(!rst_n) begin
        div_4_cnt <= 1'b0;
        clk_12_5m <= 1'b0;
    end
    else begin
        div_4_cnt <= div_4_cnt + 1'b1;
        if(div_4_cnt == 1'b1)
            clk_12_5m <= ~clk_12_5m;
```

```
30      end
31  end
32
33  always @(*) begin
34      case(lcd_id)
35          16'h4342 : lcd_pclk = clk_12_5m;
36          16'h7084 : lcd_pclk = clk_25m;
37          16'h7016 : lcd_pclk = clk;
38          16'h4384 : lcd_pclk = clk_25m;
39          16'h1018 : lcd_pclk = clk;
40          default :  lcd_pclk = 1'b0;
41      endcase
42  end
43
44  endmodule
```

分频模块根据输入的 LCD ID 对 50Mhz 时钟进行分频。由于不同分辨率的 LCD 屏需要的像素时钟频率不一样，因此分频模块根据输入的 LCD ID，来输出不同频率的像素时钟 lcd_pclk。需要说明的是，我们在本章简介部分向大家列出了一张表，表格里记录了不同分辨率的屏幕所需的像素时钟频率，为了方便编写分频的代码，我们这里没有严格按照表格里所要求的时钟频率进行输出，而是输出接近于表格所要求的时钟频率。例如 10.1 寸屏，分辨率 1280*800，如果刷新率为 60Hz 的话，需要输出 70Mhz 的像素时钟，这个时钟频率是无法通过编写代码的方式来得到，而是必须使例化时钟模块 MMCM/PLL IP 核来得到。因此，对于分辨率为 1280*800 的 10.1 寸屏幕来说，我们输出的是 50Mhz 的像素时钟，当然大家使用的是 10.1 寸屏幕，也可以通过例化时钟模块的方式，来输出一个 70Mhz 的像素时钟。

分频模块通过两个 always 语句，分别进行 2 分频和 4 分频，得到一个 25Mhz 的时钟和一个 12.5Mhz 的时钟，如代码中第 12 行至第 31 行代码所示。下面我们介绍下如何对输入的时钟进行四分频，也就是 50Mhz 的时钟四分频后，得到一个 12.5Mhz 的时钟，其实只需要分频后时钟的周期时原时钟的四倍即可，四分频的波形图如下图所示：

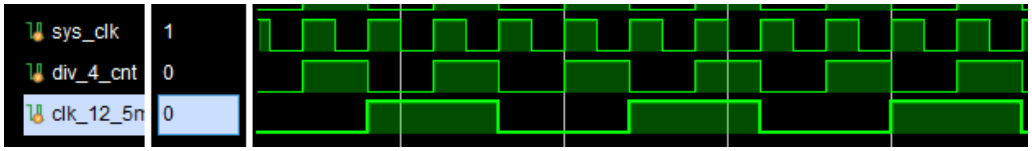


图 15.4.3 时钟四分频

上图中的 div_4_cnt 用于对系统时钟进行计数，四分频计数器只需要一位位宽，即 0 和 1 之间跳变。clk_12_5m 高电平和低电平分别占用了两个时钟周期，共占用四个时钟周期，sys_clk 的时钟频率为 50Mhz，周期为 20ns，因此 clk_12_5m 的时钟周期为 80ns，时钟频率为 12.5Mhz。

在代码的第 33 行至第 42 行，通过组合逻辑根据 LCD 屏的 ID 选择输出不同频率的像素时钟。

LCD 驱动模块的代码如下：

```
1   module lcd_driver(
2       input                lcd_pclk,     //时钟
3       input                rst_n,        //复位，低电平有效
4       input        [15:0]  lcd_id,       //LCD 屏 ID
```

```
5       input         [23:0]  pixel_data,   //像素数据
6       output        [10:0]  pixel_xpos,   //当前像素点横坐标
7       output        [10:0]  pixel_ypos,   //当前像素点纵坐标
8       output  reg   [10:0]  h_disp,       //LCD 屏水平分辨率
9       output  reg   [10:0]  v_disp,       //LCD 屏垂直分辨率
10      //RGB LCD 接口
11      output                lcd_de,       //LCD 数据使能信号
12      output                lcd_hs,       //LCD 行同步信号
13      output                lcd_vs,       //LCD 场同步信号
14      output                lcd_bl,       //LCD 背光控制信号
15      output                lcd_clk,      //LCD 像素时钟
16      output        [23:0]  lcd_rgb       //LCD RGB888 颜色数据
17      );
18
19  //parameter define
20  // 4.3' 480*272
21  parameter  H_SYNC_4342   =  11'd41;     //行同步
22  parameter  H_BACK_4342   =  11'd2;      //行显示后沿
23  parameter  H_DISP_4342   =  11'd480;    //行有效数据
24  parameter  H_FRONT_4342  =  11'd2;      //行显示前沿
25  parameter  H_TOTAL_4342  =  11'd525;    //行扫描周期
26
27  parameter  V_SYNC_4342   =  11'd10;     //场同步
28  parameter  V_BACK_4342   =  11'd2;      //场显示后沿
29  parameter  V_DISP_4342   =  11'd272;    //场有效数据
30  parameter  V_FRONT_4342  =  11'd2;      //场显示前沿
31  parameter  V_TOTAL_4342  =  11'd286;    //场扫描周期
```

代码较长，省略部分源代码……

```
85  //reg define
86  reg  [10:0] h_sync ;
87  reg  [10:0] h_back ;
88  reg  [10:0] h_total;
89  reg  [10:0] v_sync ;
90  reg  [10:0] v_back ;
91  reg  [10:0] v_total;
92  reg  [10:0] h_cnt  ;
93  reg  [10:0] v_cnt  ;
94
95  //wire define
96  wire        lcd_en;
97  wire        data_req;
```

```
98
99  //****************************************************
100 //**                    main code
101 //****************************************************
102
103 //RGB LCD 采用 DE 模式时，行场同步信号需要拉高
104 assign  lcd_hs = 1'b1;        //LCD 行同步信号
105 assign  lcd_vs = 1'b1;        //LCD 场同步信号
106
107 assign  lcd_bl = 1'b1;        //LCD 背光控制信号
108 assign  lcd_clk = lcd_pclk;   //LCD 像素时钟
109 assign  lcd_de = lcd_en;      //LCD 数据有效信号
110
111 //使能 RGB888 数据输出
112 assign  lcd_en = ((h_cnt >= h_sync + h_back) && (h_cnt < h_sync + h_back + h_disp)
113                  && (v_cnt >= v_sync + v_back) && (v_cnt < v_sync + v_back + v_disp))
114                  ? 1'b1 : 1'b0;
115
116 //请求像素点颜色数据输入
117 assign data_req = ((h_cnt>=h_sync + h_back - 1'b1) && (h_cnt < h_sync + h_back + h_disp -1'b1)
118                  && (v_cnt >= v_sync + v_back) && (v_cnt < v_sync + v_back + v_disp))
119                  ? 1'b1 : 1'b0;
120
121 //像素点坐标
122 assign pixel_xpos = data_req ? (h_cnt - (h_sync + h_back - 1'b1)) : 11'd0;
123 assign pixel_ypos = data_req ? (v_cnt - (v_sync + v_back - 1'b1)) : 11'd0;
124
125 //RGB888 数据输出
126 assign lcd_rgb = lcd_en ? pixel_data : 24'd0;
127
128 //行场时序参数
129 always @(*) begin
130     case(lcd_id)
131         16'h4342 : begin
132             h_sync  = H_SYNC_4342;
133             h_back  = H_BACK_4342;
134             h_disp  = H_DISP_4342;
135             h_total = H_TOTAL_4342;
136             v_sync  = V_SYNC_4342;
137             v_back  = V_BACK_4342;
138             v_disp  = V_DISP_4342;
```

```
139                 v_total = V_TOTAL_4342;
140             end
```

代码较长，省略部分源代码……

```
191         endcase
192     end
193
194     //行计数器对像素时钟计数
195     always@ (posedge lcd_pclk or negedge rst_n) begin
196         if(!rst_n)
197             h_cnt <= 11'd0;
198         else begin
199             if(h_cnt == h_total - 1'b1)
200                 h_cnt <= 11'd0;
201             else
202                 h_cnt <= h_cnt + 1'b1;
203         end
204     end
205
206     //场计数器对行计数
207     always@ (posedge lcd_pclk or negedge rst_n) begin
208         if(!rst_n)
209             v_cnt <= 11'd0;
210         else begin
211             if(h_cnt == h_total - 1'b1) begin
212                 if(v_cnt == v_total - 1'b1)
213                     v_cnt <= 11'd0;
214                 else
215                     v_cnt <= v_cnt + 1'b1;
216             end
217         end
218     end
219
220     endmodule
```

由本章简介部分可知，在 DE 模式下，液晶显示屏的同步信号 DE 对应的是帧和行同时有效的区域段。程序第 20 行至第 83 行代码，根据不同分辨率的屏幕做了不同参数的定义，参数的值参考了本章的表 15.1.3。程序第 128 至 192 行则根据 LCD 屏的 ID 选择不同的时序参数。

程序第 103 至 109 行是 LCD 驱动模块输出的液晶屏控制信号。其中 lcd_bl 为液晶屏背光控制端口，可以利用该端口输出一个频率在 200Hz~1kHz 范围之内的 PWM（脉冲宽度调制）信号，通过调整 PWM 信号的占空比来调节液晶屏的显示亮度。这里我们对 lcd_bl 作简单处理，将其直接赋值为 1，此时液晶屏亮度最高。分频模块输出的 lcd_pclk 直接赋值给 LCD 屏的 lcd_clk（像素时钟）引脚，为 RGB LCD 屏提供驱动时钟。另外由于我们采用 DE 同步模式驱动 RGB LCD 屏，输出给 LCD 的数据使能信号 lcd_de 在图像数据有

效时拉高，因此可以将模块内部的 lcd_en 信号直接赋值给 lcd_de。另外在 DE 模式下，需要将输出给 LCD 的行场同步信号 lcd_hs、lcd_vs 拉高。

程序第 194 至 204 行通过行计数器 h_cnt 对像素时钟计数，计满一个行扫描周期后清零并重新开始计数。程序第 206 至 218 行通过场计数器 v_cnt 对行进行计数，即扫描完一行后 v_cnt 加 1，计满一个场扫描周期后清零并重新开始计数。

将行场计数器的值与 LCD 时序中的参数作比较，我们就可以判断 DE 信号何时有效，以及何时输出 RGB888 格式的图像数据（第 111~114 行和第 126 行）。程序第 116 至 123 行输出当前像素点的横纵坐标值，由于坐标输出后下一个时钟周期才能接收到像素点的颜色数据，因此数据请求信号 data_req 比数据输出使能信号 lcd_en 提前一个时钟周期。

LCD 显示模块的代码如下：

```
1   module lcd_display(
2       input                lcd_pclk,     //时钟
3       input                rst_n,        //复位，低电平有效
4       input        [10:0]  pixel_xpos,   //当前像素点横坐标
5       input        [10:0]  pixel_ypos,   //当前像素点纵坐标
6       input        [10:0]  h_disp,       //LCD 屏水平分辨率
7       input        [10:0]  v_disp,       //LCD 屏垂直分辨率
8       output  reg  [23:0]  pixel_data    //像素数据
9       );
10
11  //parameter define
12  parameter WHITE = 24'hFFFFFF;  //白色
13  parameter BLACK = 24'h000000;  //黑色
14  parameter RED   = 24'hFF0000;  //红色
15  parameter GREEN = 24'h00FF00;  //绿色
16  parameter BLUE  = 24'h0000FF;  //蓝色
17
18  //根据当前像素点坐标指定当前像素点颜色数据，在屏幕上显示彩条
19  always @(posedge lcd_pclk or negedge rst_n) begin
20      if(!rst_n)
21          pixel_data <= BLACK;
22      else begin
23          if((pixel_xpos >= 11'd0) && (pixel_xpos < h_disp/5*1))
24              pixel_data <= WHITE;
25          else if((pixel_xpos >= h_disp/5*1) && (pixel_xpos < h_disp/5*2))
26              pixel_data <= BLACK;
27          else if((pixel_xpos >= h_disp/5*2) && (pixel_xpos < h_disp/5*3))
28              pixel_data <= RED;
29          else if((pixel_xpos >= h_disp/5*3) && (pixel_xpos < h_disp/5*4))
30              pixel_data <= GREEN;
31          else
```

```
32                pixel_data <= BLUE;
33        end
34    end
35
36    endmodule
```

LCD 显示模块将屏幕显示区域按照横坐标划分为五列等宽的区域，通过判断像素点的横坐标所在的区域，给像素点赋以不同的颜色值，从而实现彩条显示。

下图为 RGB TFT-LCD 彩条程序显示一行图像时仿真抓取的波形图，图中包含了一个完整的行扫描周期，其中的有效图像区域被划分为五个不同的区域，不同区域的像素点颜色各不相同。

图 15.4.4 仿真波形图

15.5 下载验证

首先将 FPC 排线一端与 RGB LCD 模块上的 J1 接口连接，另一端与启明星开发板上的 RGB TFTLCD 接口连接。连接时，先掀开 FPC 连接器上的黑色翻盖，将 FPC 排线蓝色面朝上插入连接器，最后将黑色翻盖压下以固定 FPC 排线，如图 15.5.1 和图 15.5.2 所示。

图 15.5.1 正点原子 RGBLCD 模块 FPC 连接器

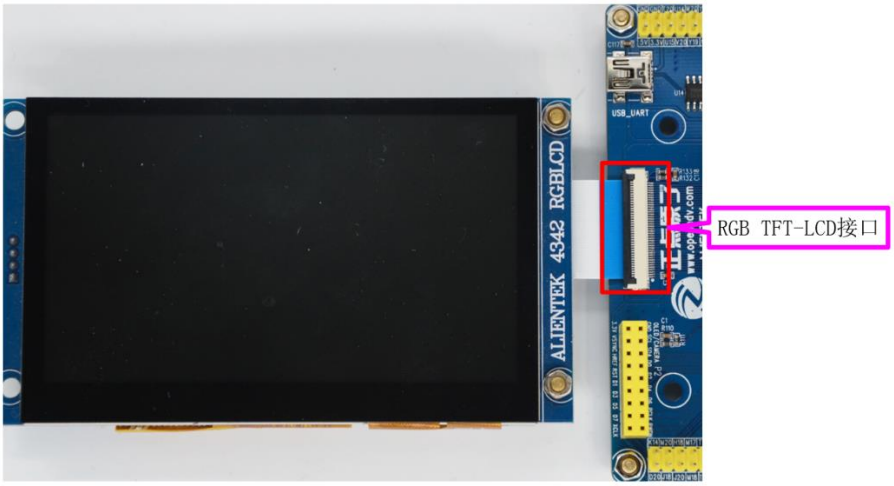


图 15.5.2 启明星开发板连接 RGB LCD 液晶屏

最后将下载器一端连电脑，另一端与开发板上的 JTAG 端口连接，连接电源线并打开电源开关。

接下来我们下载程序，验证 RGB TFT-LCD 彩条显示功能。下载完成后观察 RGB LCD 模块显示的图案如下图所示，说明 RGB TFT-LCD 彩条显示程序下载验证成功。

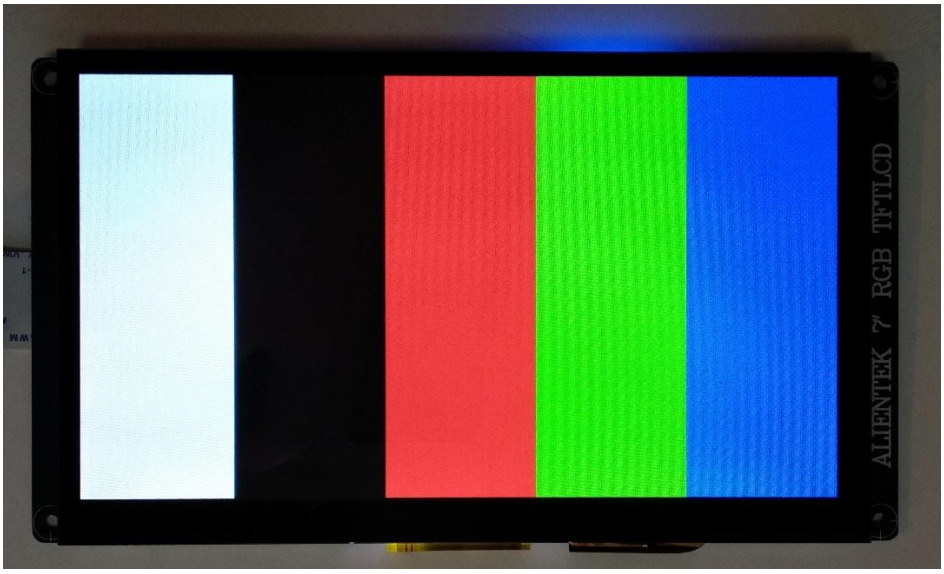


图 15.5.3 RGB TFT-LCD 彩条显示

第十六章 RGB TFT-LCD 字符和图片显示实验

我们在 RGB TFT-LCD 彩条显示实验中成功的在正点原子的 RGB LCD 液晶屏模块上显示出了彩条。本章我们在 RGB TFT-LCD 彩条显示实验的基础上稍作修改，在 LCD 上完成“正点原子 logo 图片”和汉字“正点原子”的显示。

本章包括以下几个部分：

16.1 RGB TFT-LCD 简介

16.2 实验任务

16.3 硬件设计

16.4 程序设计

16.5 下载验证

16.1 RGB TFT-LCD 简介

我们在“RGB TFT-LCD 彩条显示实验”中对正点原子的 RGB LCD 液晶屏模块作了详细的介绍，包括数据输入时序、同步方式、以及分辨率等。如果大家对这部分内容不是很熟悉的话，请参考“RGB TFT-LCD 彩条显示实验”中的简介部分。

16.2 实验任务

本节的实验任务是通过启明星开发板上的 RGB TFT-LCD 接口，在正点原子的 RGB LCD 液晶屏的左上角位置显示图片以及 4 个汉字“正点原子”。其中每个汉字的大小为 32*32，图片的大小为 100*100。

16.3 硬件设计

RGB TFT-LCD 接口部分的硬件设计原理及本实验中各端口信号的管脚分配与“RGB TFT-LCD 彩条显示实验”完全相同，请参考“RGB TFT-LCD 彩条显示实验”中的硬件设计部分。

16.4 程序设计

图 16.4.1 是根据本章实验任务画出的系统框图。可以看出，其中本次实验的系统框图与“RGB TFT-LCD 彩条显示实验”基本一致，我们只需要修改 LCD 显示模块就可以实现在 LCD 液晶屏上显示字符和图片的功能。另外，由于图片的像素数据较多，因此我们在 LCD 显示模块中例化了一个 ROM，用来存储图片数据。

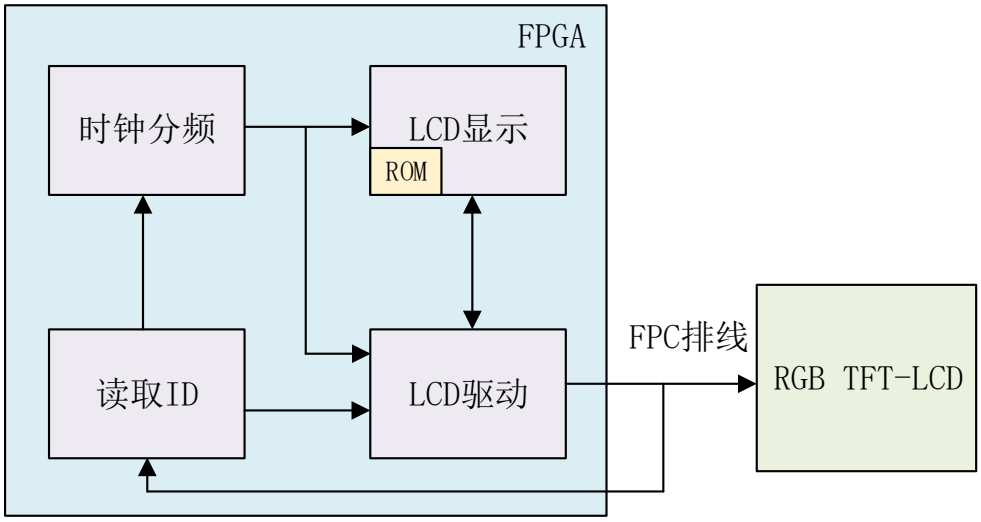


图 16.4.1 RGB TFT-LCD 字符和图片显示实验系统框图

字符（包括汉字、字母和符号等）的本质都是点阵，在 LCD 屏幕上体现为字符显示区域内像素点的集合。字符的大小决定了字符显示区域内像素点的数目，而字符的样式（字体、颜色等）则决定了各像素点的颜色值。因此，在显示字符之前，我们需要先指定字符的大小、样式，然后获取该字符的点阵，这个过程我们称之为“提取字模”，或简称“取模”。

我们一般使用 0 和 1 的组合来描述字符的点阵排列：点阵中每个像素点用一位（1 bit）数据来表示，其中用于表征字符的像素点用数字 1 来表示，其他的像素点作为背景用数字 0 来表示，如图 16.4.2 所示。采用这种方式描述的字符是不含有颜色特征的，只能区分点阵中的字符和背景。

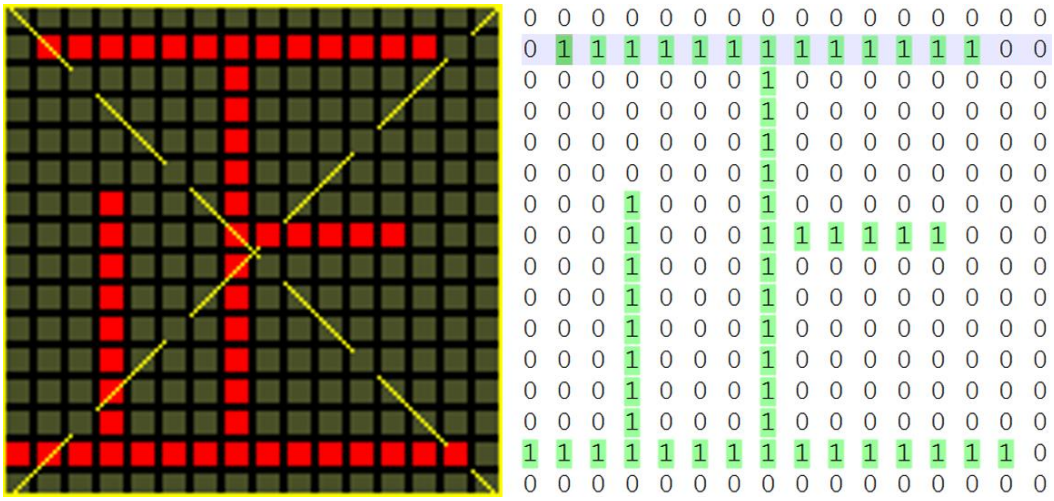


图 16.4.2 汉字“正”及其点阵描述

字模的提取可通过字符取模软件来实现，在这里我们使用取模软件“PCtoLCD2002”来获取汉字“正点原子”的字模。首先在开发板所随附的资料盘（A 盘）中“6_软件资料/1_软件/PCtoLCD2002 完美版”目录下找到“PCtoLCD2002”并双击打开，如下图所示：

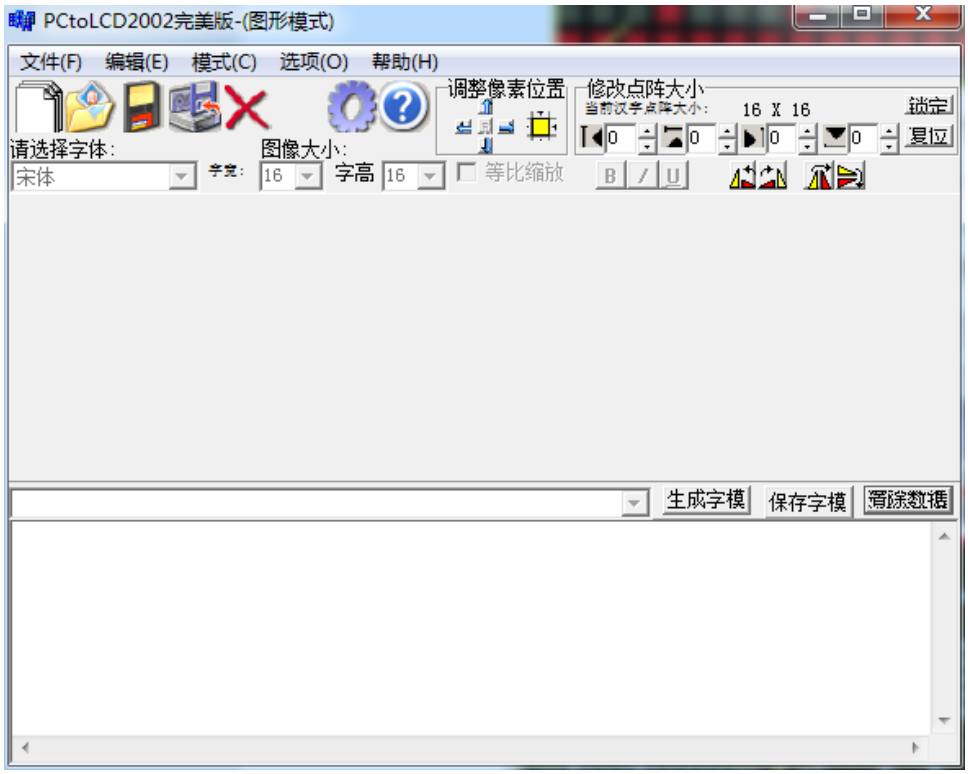


图 16.4.3 取模软件 PCtoLCD2002

打开之后会发现软件中的字体、字宽和字高都是无法设置的，这个时候点击菜单栏的“模式”，选择“字符模式”，如下图所示。

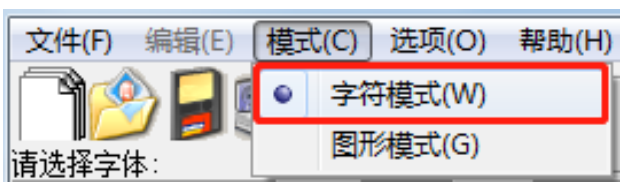


图 16.4.4 切换到字符模式

切换到字符模式后，就可以设置字体、字宽和字高了。字宽和字高的值越高，显示在 LCD 屏上的字符就越大，但是代码也需要做相应的修改。这里将字体选择默认的“宋体”，字宽和字高设置成“32”，然后在下方文本框中输入汉字“正点原子”，如下图所示：

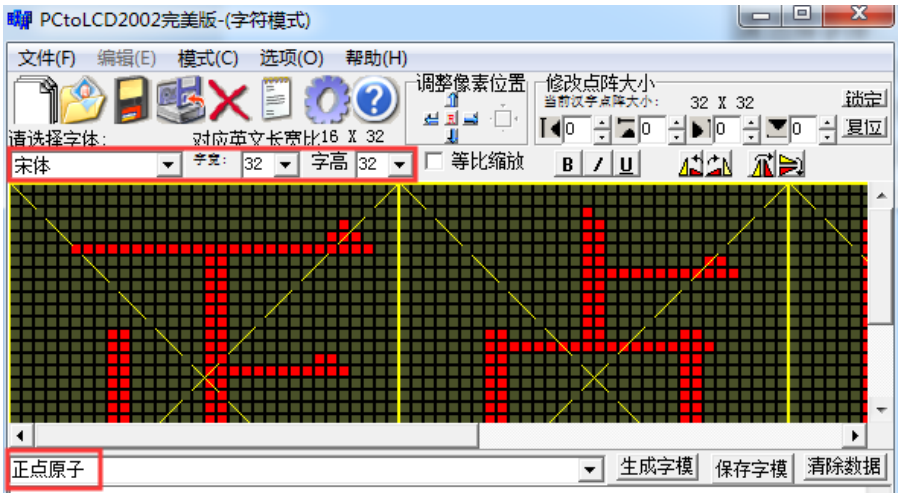


图 16.4.5 字符设置

由于 PCtoLCD2002 会给每个字符生成一个独立的字模，如果此时点击文本框右侧的“生成字模”按钮，我们将会得到四个 32*32 的字模。然而为了方便在 LCD 上显示，我们将四个汉字看作一个整体，从而获得一个字宽为 128，字高为 32 的“大字模”。为了达到这个目的，我们首先将图 16.4.5 中四个汉字的点阵保存为.BMP 格式的图片。在菜单栏中点击“文件”并选择“另存为”，在保存界面中指定文件存储路径，并选择保存类型为“BMP 图像文件”，然后输入文件名“正点原子_bmp”，最后点击“保存”。本次我们在工程路径下新建一个“doc”文件夹，将生成的 BMP 图片保存在 doc 文件夹下。如图 16.4.6 和图 16.4.7 所示。

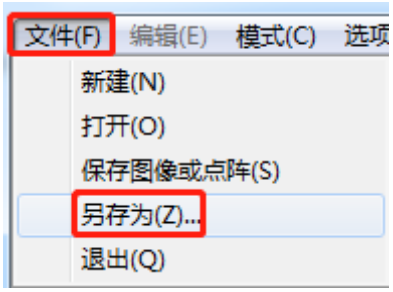


图 16.4.6 点击“文件”并另存为图像

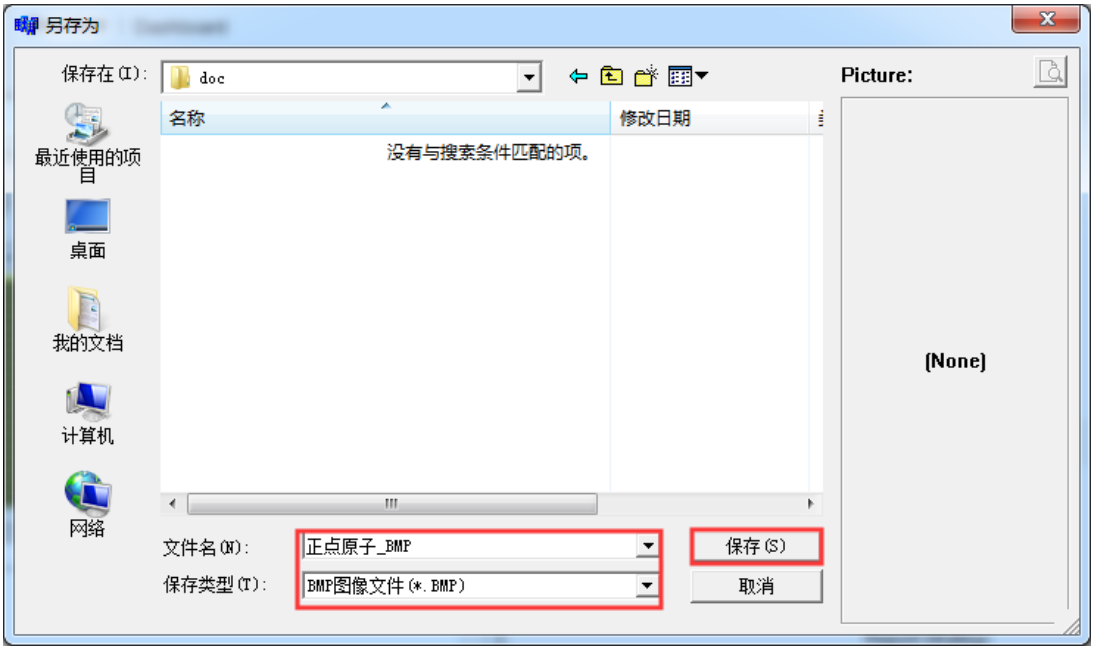


图 16.4.7 BMP 格式图片保存界面

我们在“画图”中打开刚刚保存的 BMP 格式的图片如下所示：

图 16.4.8 保存的 BMP 格式图片

接下来我们将取模软件 PCtoLCD2002 切换至图形模式，在菜单栏中点击“模式”，选择“图形模式”。

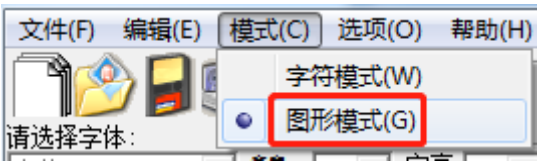


图 16.4.9 切换至图形模式

然后在菜单栏中点击“文件”并选择“打开”，指定图 16.4.7 中存放 BMP 格式图片的路径并打开图片“正点原子_bmp”，图片打开后如下所示。

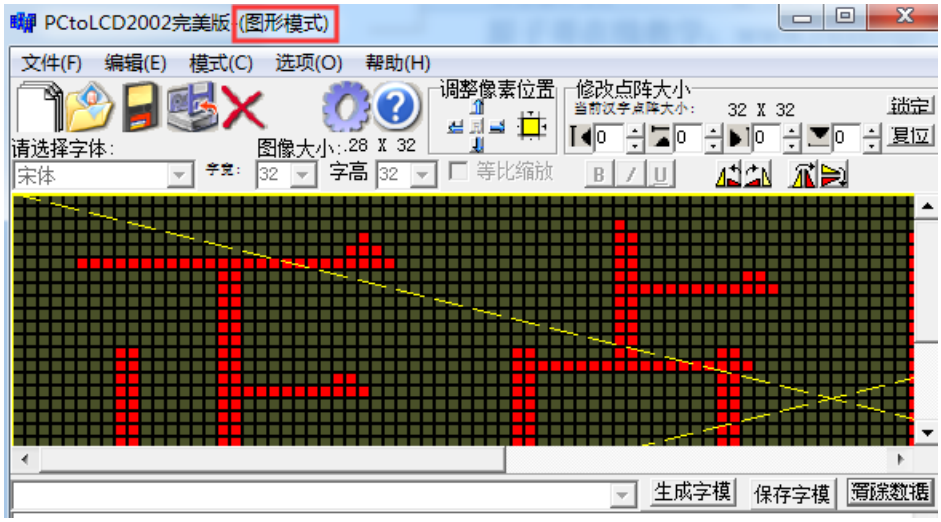


图 16.4.10 PCtoLCD2002 图形模式

请大家注意比较图 16.4.10 与图 16.4.5 的差异，在上图中，四个汉字“正点原子”被看作一个整体，而不再是四个独立的字符。实际上，这四个汉字也确实是作为一个整体以 BMP 图片的形式导入到取模软件中的。

在生成字模之前，我们需要先设置字模的格式。在菜单栏中点击“选项”，并在弹出的配置界面中按照下图进行配置，配置完成后点击确定。

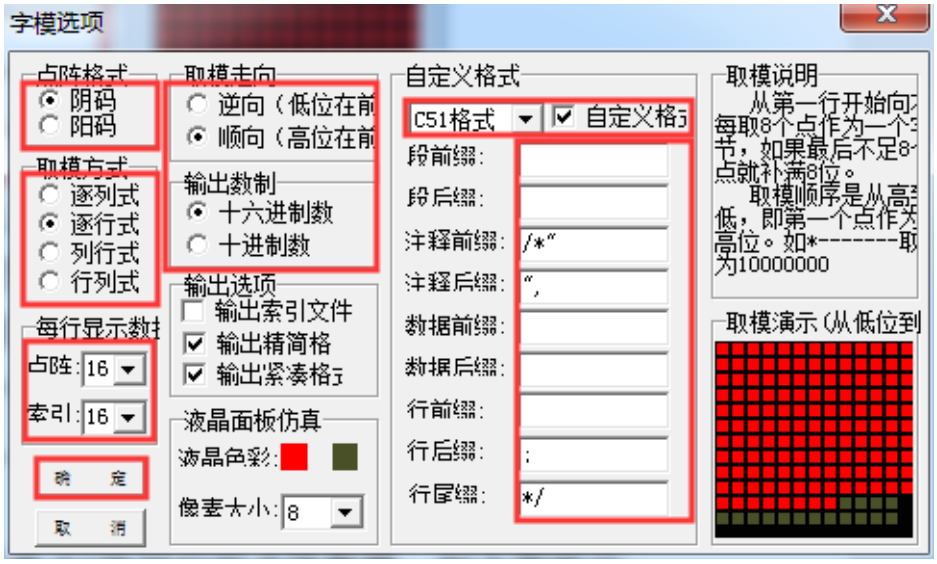


图 16.4.11 字模格式配置界面

在配置界面中，当鼠标悬浮在各配置选项上时，软件会自动提示当前配置的含义。需要注意的是图 16.4.11 左下角“每行显示数据”是以字节（Byte）为单位的，而一个字节的数据为 8 个 bit，即可以表示一行点阵中的 8 个像素点。由于图 16.4.10 中的点阵每行为 128 个像素点，所以需要 16 个 Byte 的数据来表示一行，因此将“每行显示数据—点阵”处设置为 16。

配置字模选项完成后，点击“生成字模”，即可得到汉字 “正点原子”所对应的点阵数据，如下图所示：

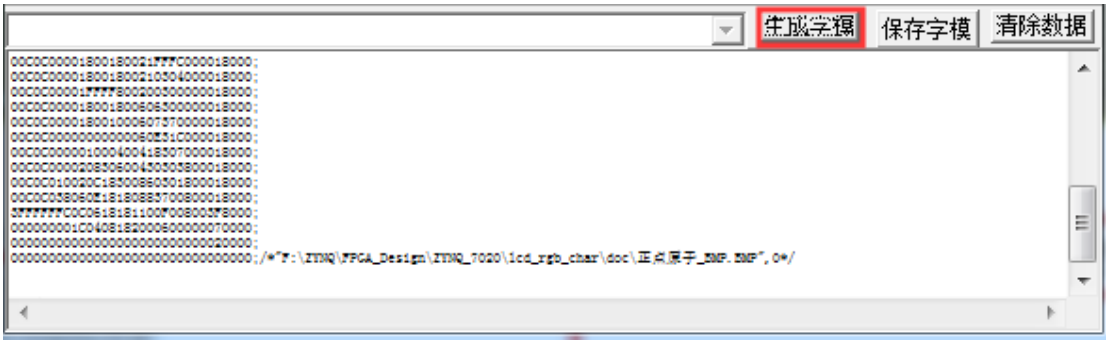


图 16.4.12 生成字模

最后点击保存字模，命名成“正点原子_字模”，可将生成的点阵数据保存在 txt 格式的文本文档中，如图 16.4.13 所示。数据以十六进制显示，每行有 16 个 Byte，对应每行四个汉字共 128 个像素点；共有 32 行，对应每个汉字的高度为 32。

正点原子_字模.TXT

```
F:\ZYNQ\FPGA_Design\ZYNQ_7020\lcd_rgb_char\doc\正点原子_BMP.BMP(0)

00000000000000000000000000000000;
00000000000000000000000000000000;
00000000000100000000002000000000;
00000100001800002000070000000C0;
00000038000180000 3FFFFF803FFFFE0;
07FFFFFC000180000300 6000000001E0;
0000C000000180600300600000000300;
0000C0000001FFF00300C00000000600;
0000C000000180000310804000001800;
0000C000000180000 31FFFE000003000;
0000C000000180000 31800400001C000;
0000C000000180000 31800400001C000;
00C0C000018181800318004000018000;
00C0C00001FFFFC0031FFFC000018010;
00C0C060018001800318004000018038;
00C0FFF001800180031800403FFFFFFC;
00C0C000018001800318004000018000;
00C0C000018001800218004000018000;
00C0C000018001800 21FFFC000018000;
00C0C000018001800210304000018000;
00C0C00001FFFF800200300000018000;
00C0C000018001800606300000018000;
00C0C000018001000607370000018000;
00C0C00000000000060E31C000018000;
00C0C000001000400418307000018000;
00C0C000020830600430303800018000;
00C0C010020C18300860301800018000;
00C0C038060E18180883700800018000;
3FFFFFFC0C0618181100F008003F8000;
000000001C040818200060000070000;
00000000000000000000000000020000;
00000000000000000000000000000000;/*"F:\ZYNQ\FPGA_Design\ZYNQ_7020\l
```

图 16.4.13 “正点原子”字模

提取字模完成后，我们需要在 LCD 显示模块中将获取的点阵数据映射到液晶屏中心 32*128 个的像素点的字符显示区域，从而实现字符的显示。

到这里提取字模的过程就已经完成了，接下来我们介绍图片像素数据的获取方法。

LCD 显示模块中的 ROM 是通过例化 IP 核来实现的只读存储器，它使用 FPGA 的片上存储资源，即 BRAM。由于 FPGA 的片上存储资源有限，所以 ROM 中存储的图片大小也受到限制，本次实验采用的图片分辨率为 100*100。启明星 ZYNQ 开发板上的 RGB TFT-LCD 接口采用 RGB888 数据格式，即每个像素点的颜色用 24bit 的数据来表示，因此大小为 100*100*24bit = 240000bit=234.375Kbit≈0.23Mbit。XC7Z020 芯片的 BRAM 存储容量为 4.9Mbit，XC7Z010 芯片的 BRAM 存储容量为 2.1Mbit，都能够满足本次实验中的

图片存储需求。

ROM 作为只读存储器，在调用 IP 核时需要指定初始化文件，在这里就是写入存储器中的图片数据，各种格式的图片（bmp、jpg 等）在 Xilinx 开发软件中都是以 COE 文件或者 HEX 文件的形式导入到 ROM 中的。COE 文件格式较为简单，因此本次实验选取 COE 的文件格式。COE 是一种 Xilinx 工具能识别的文件格式，在文件的开头定义了存储数据的进制和初始化的数据，存储的数据最后一行以“;”结束，其余的以“,”结束。例如一个存储数据为 16 进制，深度为 5 的 COE 文件内容如下图所示：

```
1 memory_initialization_radix=16;
2 memory_initialization_vector=
3 ff0000,
4 00ff00,
5 0000ff,
6 ffff00,
7 00ffff;
```

图 16.4.14 COE 文件格式

图中第一行定义存储的数据为 16 进制，第二行初始化的数值向量，分别对应不同存储单元的数据，存储地址从 0 开始，依次累加，最后一个存储地址的数据以“;”结束。

当需要存储的数据量较小时，如果我们知道数据的内容，那么就可以仿照图 16.4.14 的格式手动编写 COE 文件。但是由于图片的数据量较大，并且我们无法直接看出各个像素点对应的颜色数据，因此需要借助工具来实现图片到 COE 文件的转换。在这里我们使用正点原子提供的工具“PicToLCD”来实现这一转换过程，该工具位于开发板所随附的资料中“6_软件资料/1_软件/PicToLCD”目录下。

我们在 Windows 自带的“画图”工具中将正点原子的 LOGO 图片大小调整 100*100，并利用工具 PicToLCD 转换得到 COE 文件“ZDYZ.coe”。

双击运行“PicToLCD.exe”，点击“加载图片”并在弹出的界面中选择需要转换的图片(注意：待转换图片分辨率的大小必须是 100*100，否则代码中访问 ROM 的最大地址需要修改)。图片加载成功后工具会在图片属性中指示出图片的文件名和大小；接下来选择图片转换的数据格式为 RGB888；文件类型选择“COE”；最后点击“一键转换” 按钮，在弹出的界面中选择 COE 文件的存放路径并输入文件名。PicToLCD 转换过程中的软件界面如图 16.4.15 所示：

图 16.4.15 PicToLCD 转换界面

最终转换得到的 COE 文件部分截图如下所示：

ZDYZ.coe

```
; Picture Resolution : 100 x 100 x 24Bit

memory_initialization_radix=16;

memory_initialization_vector=

FFFFFF,
FFFFFF,
FFFFFF,
FFFFFF,
FFFFFF,
```

图 16.4.16 转换得到的 COE 文件

在 Vivado 软件中，RAM 和 ROM 都是由 BMG IP 核（Block Memory Generator）配置生成的，ROM 的配置过程和 RAM 类似，在 BGM IP 核配置过程的“Basic”选项卡中，要选择“Single Port ROM”选项，如下图所示：

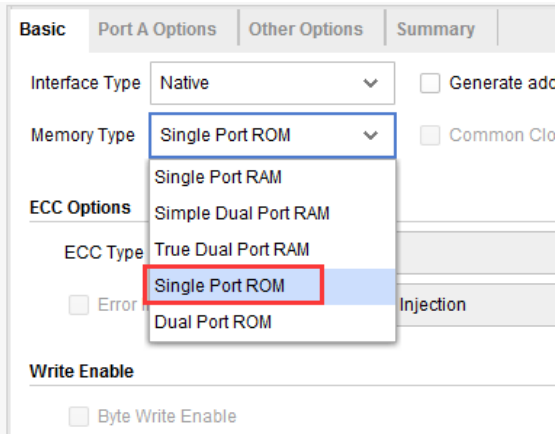


图 16.4.17 “Single Port ROM”选项

在“Port A Options”选项卡中，设置 ROM 读端口的位宽和深度，因为我们的像素数据是“RGB888”格式，所以端口位宽要设置成 24 位；使用“Notepad++”编辑器打开.coe 初始化文件，可以看到存储的数据共有 10000 个数据，所以端口深度设置成 10000。与 RAM IP 核一样，我们同样不使用流水线寄存器。“Port A Options”选项卡的设置如下图所示：

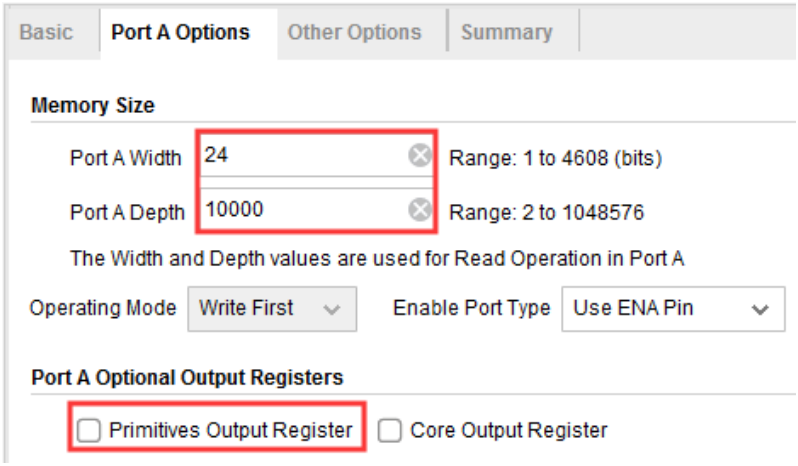


图 16.4.18 “Port A Options”选项卡

接下来是最重要的一步，在“Other Options”选项卡中，加载我们刚刚使用“PicToLCD”软件生成的“.coe”文件，如下图所示：

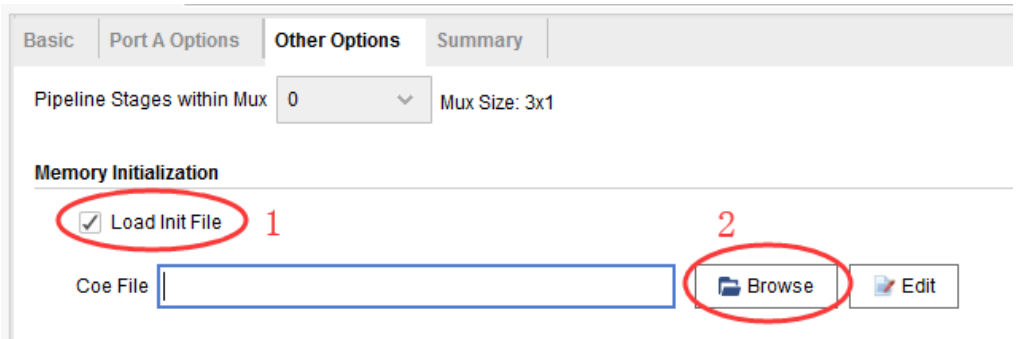


图 16.4.19 加载“.coe”文件

最后点击 OK，完成对 ROM 的配置。

程序中各模块端口及信号连接如下图所示：

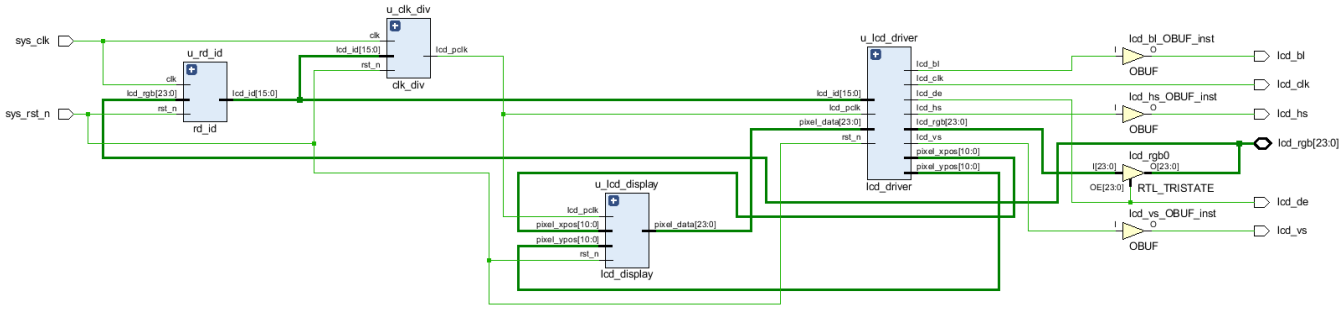


图 16.4.20 顶层模块原理图

图 16.4.20 中的顶层模块中的 ID 读取模块（rd_id）、时钟分频模块（clk_div）以及 LCD 驱动模块（lcd_driver）均与“RGB TFT-LCD 彩条显示实验”完全相同，本次实验只对 LCD 显示模块（lcd_display）作了修改。因此，这里我们重点讲解 lcd_display 模块，其他部分大家可以参考“RGB TFT-LCD 彩条显示实验”。

LCD 显示模块的代码如下：

```
1  module lcd_display(
2      input             lcd_pclk,     //时钟
3      input             rst_n,        //复位，低电平有效
4
5      input      [10:0] pixel_xpos,   //像素点横坐标
6      input      [10:0] pixel_ypos,   //像素点纵坐标
7      output reg [23:0] pixel_data    //像素点数据,
8  );
9
10 //parameter define
11 localparam PIC_X_START = 11'd1;      //图片起始点横坐标
12 localparam PIC_Y_START = 11'd1;      //图片起始点纵坐标
13 localparam PIC_WIDTH   = 11'd100;    //图片宽度
14 localparam PIC_HEIGHT  = 11'd100;    //图片高度
15
16 localparam CHAR_X_START= 11'd1;      //字符起始点横坐标
17 localparam CHAR_Y_START= 11'd110;    //字符起始点纵坐标
18 localparam CHAR_WIDTH  = 11'd128;    //字符宽度,4 个字符:32*4
19 localparam CHAR_HEIGHT = 11'd32;     //字符高度
20
21 localparam BACK_COLOR  = 24'hE0FFFF; //背景色，浅蓝色
22 localparam CHAR_COLOR  = 24'hff0000; //字符颜色，红色
23
24 //reg define
25 reg   [127:0] char[31:0];  //字符数组
26 reg   [13:0]  rom_addr  ;  //ROM 地址
27
28 //wire define
29 wire  [10:0]  x_cnt;       //横坐标计数器
30 wire  [10:0]  y_cnt;       //纵坐标计数器
31 wire          rom_rd_en ;  //ROM 读使能信号
32 wire  [23:0]  rom_rd_data ;//ROM 数据
33
34 //*****************************************************
35 //**                    main code
36 //*****************************************************
37
38 assign  x_cnt = pixel_xpos - CHAR_X_START; //像素点相对于字符区域起始点水平坐标
39 assign  y_cnt = pixel_ypos - CHAR_Y_START; //像素点相对于字符区域起始点垂直坐标
40 assign  rom_rd_en = 1'b1;                  //读使能拉高，即一直读 ROM 数据
41
```

```
42  //给字符数组赋值，显示汉字“正点原子”，每个汉字大小为 32*32
43  always @(posedge lcd_pclk) begin
44      char[0 ]  <= 128'h00000000000000000000000000000000;
45      char[1 ]  <= 128'h00000000000000000000000000000000;
46      char[2 ]  <= 128'h00000000000100000000002000000000;
47      char[3 ]  <= 128'h00000010000180000200007000000C0;
48      char[4 ]  <= 128'h0000003800018000003FFFFF803FFFFE0;
49      char[5 ]  <= 128'h07FFFFFC000180000300600000001E0;
50      char[6 ]  <= 128'h0000C0000001806003006000000000300;
51      char[7 ]  <= 128'h0000C0000001FFF00300C00000000600;
52      char[8 ]  <= 128'h0000C0000001800003108040000001800;
53      char[9 ]  <= 128'h0000C0000001800003 1FFFE000003000;
54      char[10]  <= 128'h0000C0000001800003 1800400001C000;
55      char[11]  <= 128'h0000C0000001800003 1800400001C000;
56      char[12]  <= 128'h00C0C0000181818003 18004000018000;
57      char[13]  <= 128'h00C0C00001FFFFC0031FFFC000018010;
58      char[14]  <= 128'h00C0C0600180018003 18004000018038;
59      char[15]  <= 128'h00C0FFF0018001800318004 03FFFFFC;
60      char[16]  <= 128'h00C0C0000180018003 18004000018000;
61      char[17]  <= 128'h00C0C0000180018002 18004000018000;
62      char[18]  <= 128'h00C0C0000180018002 1FFFC000018000;
63      char[19]  <= 128'h00C0C0000180018002 10304000018000;
64      char[20]  <= 128'h00C0C00001FFFF800200300000018000;
65      char[21]  <= 128'h00C0C0000180018006 06300000018000;
66      char[22]  <= 128'h00C0C0000180010006 07370000018000;
67      char[23]  <= 128'h00C0C0000000000060E31C000018000;
68      char[24]  <= 128'h00C0C0000010004004 18307000018000;
69      char[25]  <= 128'h00C0C0000208306004 30303800018000;
70      char[26]  <= 128'h00C0C010020C183008 60301800018000;
71      char[27]  <= 128'h00C0C038060E181808 83700800018000;
72      char[28]  <= 128'h3FFFFFFC0C0618181 100F008003F8000;
73      char[29]  <= 128'h000000001C040818 2000600000070000;
74      char[30]  <= 128'h00000000000000000000000000020000;
75      char[31]  <= 128'h00000000000000000000000000000000;
76  end
77
78  //为 LCD 不同显示区域绘制图片、字符和背景色
79  always @(posedge lcd_pclk or negedge rst_n) begin
80      if (!rst_n)
81          pixel_data <= BACK_COLOR;
82      else if( (pixel_xpos >= PIC_X_START) && (pixel_xpos < PIC_X_START + PIC_WIDTH)
```

```
83              && (pixel_ypos >= PIC_Y_START) && (pixel_ypos < PIC_Y_START + PIC_HEIGHT) )
84          pixel_data <= rom_rd_data ;  //显示图片
85      else if((pixel_xpos >= CHAR_X_START) && (pixel_xpos < CHAR_X_START + CHAR_WIDTH)
86           && (pixel_ypos >= CHAR_Y_START) && (pixel_ypos < CHAR_Y_START + CHAR_HEIGHT)) begin
87          if(char[y_cnt][CHAR_WIDTH -1'b1 - x_cnt])
88              pixel_data <= CHAR_COLOR;    //显示字符
89          else
90              pixel_data <= BACK_COLOR;    //显示字符区域的背景色
91      end
92      else
93          pixel_data <= BACK_COLOR;        //屏幕背景色
94  end
95
96  //根据当前扫描点的横纵坐标为 ROM 地址赋值
97  always @(posedge lcd_pclk or negedge rst_n) begin
98      if(!rst_n)
99          rom_addr <= 14'd0;
100     //当横纵坐标位于图片显示区域时,累加 ROM 地址
101     else if((pixel_ypos >= PIC_Y_START) && (pixel_ypos < PIC_Y_START + PIC_HEIGHT)
102         && (pixel_xpos >= PIC_X_START) && (pixel_xpos < PIC_X_START + PIC_WIDTH))
103         rom_addr <= rom_addr + 1'b1;
104     //当横纵坐标位于图片区域最后一个像素点时,ROM 地址清零
105     else if((pixel_ypos >= PIC_Y_START + PIC_HEIGHT))
106         rom_addr <= 14'd0;
107 end
108
109 //ROM: 存储图片
110 blk_mem_gen_0  blk_mem_gen_0 (
111   .clka  (lcd_pclk),     // input wire clka
112   .ena   (rom_rd_en),    // input wire ena
113   .addra (rom_addr),     // input wire [13 : 0] addra
114   .douta (rom_rd_data)   // output wire [23 : 0] douta
115 );
116
117 endmodule
```

程序中第 10 行至 22 行定义了一系列的参数，方便大家修改图片的位置，字符位置和字符颜色等。

程序中第 25 行定义了一个大小为 32*128bit 的二维数组 char，用于存储取模得到的点阵数据。二维数组 char 共 32 行，每一行有 128 位数据，在程序的第 42 至 76 行完成了对该二维数组的赋值。赋值后数组中每一行数据从高位到低位分别对应点阵中该行从左向右的每一个像素点。

程序中第 78 行至 94 行完成了字符和图片的显示，根据当前像素点扫描的坐标，为 pixel_data 赋值字符颜色、背景色或者图片数据（从 ROM 中读出的数据）。屏幕上字符显示区域内的像素点与字符数组 char

中的点阵数据一一映射。当点阵数据为 1 时，将像素点颜色赋值为红色，用来显示字符；当点阵数据为 0 时，将像素点颜色赋值为浅蓝色，用来作为字符显示区域的背景。屏幕上除字符和图片显示区域之外的其他区域内的像素点均赋值为浅蓝色。

程序中第 96 行至 107 行根据当前的扫描坐标为 ROM 地址赋值。需要说明的是，我们将 ROM 的读使能信号固定为高电平，即一直读 ROM，而 ROM 中的数据是由 ROM 地址来进行控制。

图 16.4.21 为LCD显示模块的仿真波形图，其中rom_rd_en固定为高电平，当像素点横坐标（pixel_xpos）和像素点纵坐标（pixel_ypos）位于图片显示区域时，读 ROM 地址依次累加，从而依次获取 ROM 中的数据。

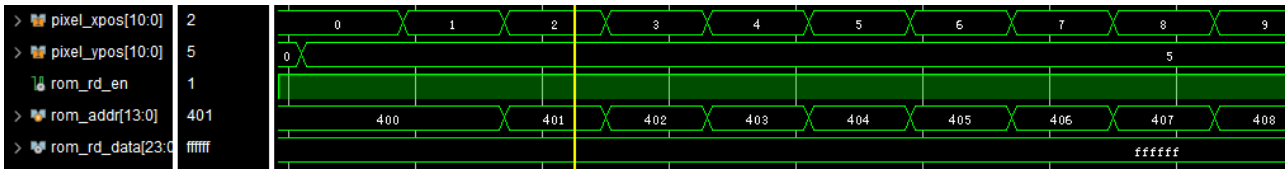


图 16.4.21 LCD 显示模块仿真图

16.5 下载验证

首先将 FPC 排线一端与 RGB LCD 模块上的 J1 接口连接，另一端与启明星开发板上的 RGB TFTLCD 接口连接。然后将下载器一端连电脑，另一端与开发板上的 JTAG 端口连接，最后连接电源线并打开电源开关。

接下来我们下载程序，验证 RGB LCD 字符和图片显示的功能。下载完成后观察 RGB LCD 液晶屏上显示的图案如下图所示，说明 RGB TFT-LCD 字符和图片显示程序下载验证成功。

图 16.5.1 RGB TFT-LCD 字符和图片显示

第十七章 HDMI 彩条显示实验

HDMI 接口在消费类电子行业，如电脑、液晶电视、投影仪等产品中得到了广范的应用。一些专业的视频设备如摄像机、视频切换器等也都集成了 HDMI 接口。本章我们将学习如何驱动 ZYNQ 开发板上的 HDMI 接口。

本章包括以下几个部分：

17.1 简介

17.2 实验任务

17.3 硬件设计

17.4 程序设计

17.5 下载验证

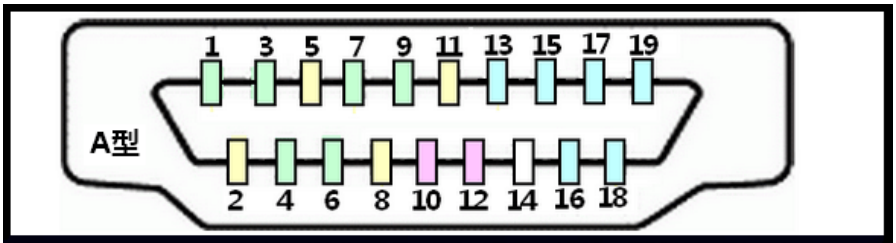
17.1 简介

HDMI 是新一代的多媒体接口标准，英文全称是 High-Definition Multimedia Interface，即高清多媒体接口。它能够同时传输视频和音频，简化了设备的接口和连线；同时提供了更高的数据传输带宽，可以传输无压缩的数字音频及高分辨率视频信号。HDMI 1.0 版本于 2002 年发布，最高数据传输速度为 5Gbps；而 2017 年发布的 HDMI 2.1 标准的理论带宽可达 48Gbps。

HDMI 向下兼容 DVI，但是 DVI（数字视频接口）只能用来传输视频，而不能同时传输音频，这是两者最主要的差别。此外，DVI 接口的尺寸明显大于 HDMI 接口，如下图所示：

图 17.1.1 DVI 接口（左）和 HDMI 接口（右）实物图

图 17.1.1 右侧是生活中最常见的 A 型 HDMI 接口，其引脚定义如下图所示：



HDMI 信号引脚定义

引脚	信号	引脚	信号	引脚	信号
1	数据2+	2	数据2 屏蔽	3	数据2-
4	数据1+	5	数据1 屏蔽	6	数据1-
7	数据0+	8	数据0 屏蔽	9	数据0-
10	时钟+	11	时钟屏蔽	12	时钟-
13	CEC	14	保留	15	SCL
16	SDA	17	DDC/CEC地	18	+5V电源
19	热插拔检测				

数据通道 屏蔽 即插即用 时钟

图 17.1.2 HDMI 接口引脚定义

DVI 和 HDMI 接口协议在物理层使用 TMDS 标准传输音视频数据。TMDS（Transition Minimized Differential Signaling，最小化传输差分信号）是美国 Silicon Image 公司开发的一项高速数据传输技术，在 DVI 和 HDMI 视频接口中使用差分信号传输高速串行数据。TMDS 差分传输技术使用两个引脚（如图 17.1.2 中的“数据 2+”和“数据 2-”）来传输一路信号，利用这两个引脚间的电压差的正负极性和大小来决定传

输数据的数值（0 或 1）。

Xilnx 在 Spartan-3A 系列之后的器件中，加入了对 TMDS 接口标准的支持，用于在 FPGA 内部实现 DVI 和 HDMI 接口。

由于本次实验只是使用 HDMI 接口来显示图像，不需要传输音频，因此我们只需要实现 DVI 接口的驱动逻辑即可。不过在此之前我们还需要简单地了解一下 TMDS 视频传输协议。

图 17.1.3 是 TMDS 发送端和接收端的连接示意图。DVI 或 HDMI 视频传输所使用的 TMDS 连接通过四个串行通道实现。对于 DVI 来说，其中三个通道分别用于传输视频中每个像素点的红、绿、蓝三个颜色分量（RGB 4:4:4 格式）。HDMI 默认也是使用三个 RGB 通道，但是它同样可以选择传输像素点的亮度和色度信息（YCrCb 4:4:4 或 YCrCb 4:2:2 格式）。第四个通道是时钟通道，用于传输像素时钟。独立的 TMDS 时钟通道为接收端提供接收的参考频率，保证数据在接收端能够正确恢复。

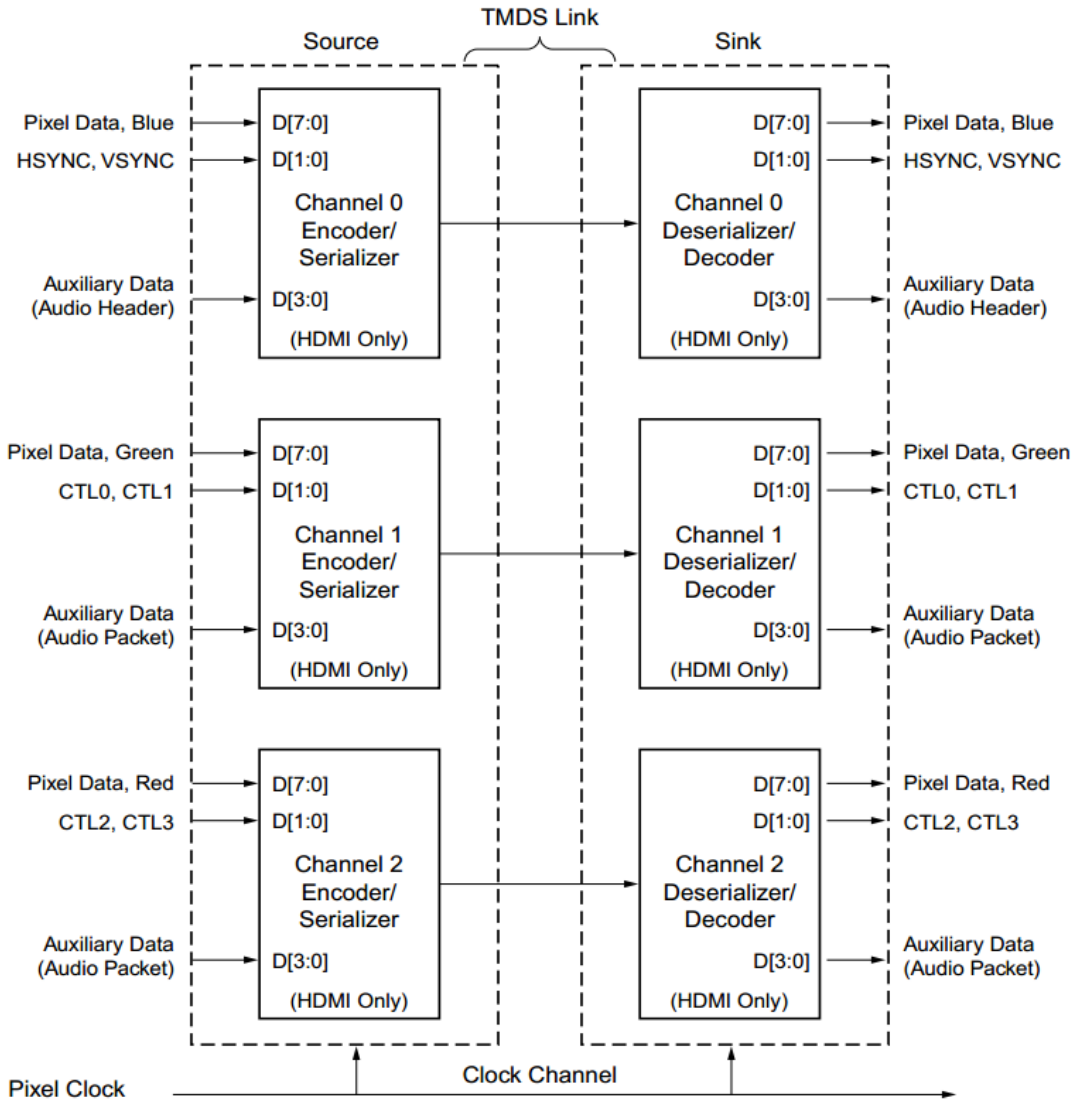


图 17.1.3 TMDS 连接示意图

如果每个像素点的颜色深度为 24 位，即 RGB 每个颜色分量各占 8 位，那么每个通道上的颜色数据将通过一个 8B/10B 的编码器（**Encoder**）来转换成一个 10 位的像素字符。然后这个 10 位的字符通过并串转换器（**Serializer**）转换成串行数据，最后由 TMDS 数据通道发送出去。这个 10:1 的并转串过程所生成的串行数据速率是实际像素时钟速率的 10 倍。

在传输视频图像的过程中，数据通道上传输的是编码后的有效像素字符。而在每一帧图像的行与行之间，以及视频中不同帧之间的时间间隔（消隐期）内，数据通道上传输的则是控制字符。每个通道上有两位控制信号的输入接口，共对应四种不同的控制字符。这些控制字符提供了视频的行同步（HZYNC）以及帧同步（VSYNC）信息，也可以用来指定所传输数据的边界（用于同步）。

对于 DVI 传输，整个视频的消隐期都用来传输控制字符。而 HDMI 传输的消隐期除了控制字符之外，还可以用于传输音频或者其他附加数据，比如字幕信息等。这就是 DVI 和 HDMI 协议之间最主要的差别。从图 17.1.3 中也可以看出这一差别，即“Auxiliary Data”接口标有“HDMI Olny”，即它是 HDMI 所独有的接口。

从前面的介绍中我们可以看出，TMDS 连接从逻辑功能上可以划分成两个阶段：编码和并串转换。在编码阶段，编码器将视频源中的像素数据、HDMI 的音频/附加数据，以及行同步和场同步信号分别编码成 10 位的字符流。然后在并串转换阶段将上述的字符流转换成串行数据流，并将其从三个差分输出通道发送出去。

DVI 编码器在视频有效数据段输出像素数据，在消隐期输出控制数据，如图 17.1.4 所示。其中 VDE（Video Data Enable）为高电平时表示视频数据有效，为低电平代表当前处于视频消隐期。

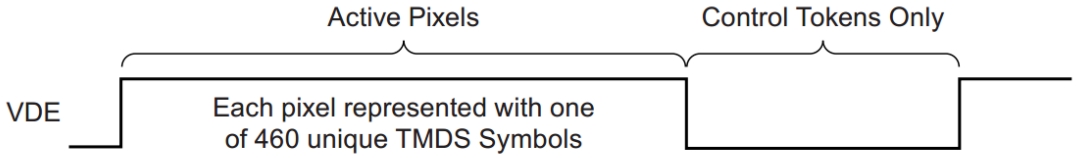


图 17.1.4 DVI 编码输出示意图

图 17.1.5 给出了三个通道的 DVI 编码器示意图。对于像素数据的 RGB 三个颜色通道，编码器的逻辑是完全相同的。VDE 用于各个通道选择输出视频像素数据还是控制数据。HSYNC 和 VSYNC 信号在蓝色通道进行编码得到 10 位字符，然后在视频消隐期传输。绿色和红色通道的控制信号 C0 和 C1 同样需要进行编码，并在消隐期输出。但是 DVI 规范中这两个通道的控制信号是预留的（未用到），因此将其置为 2’b00。

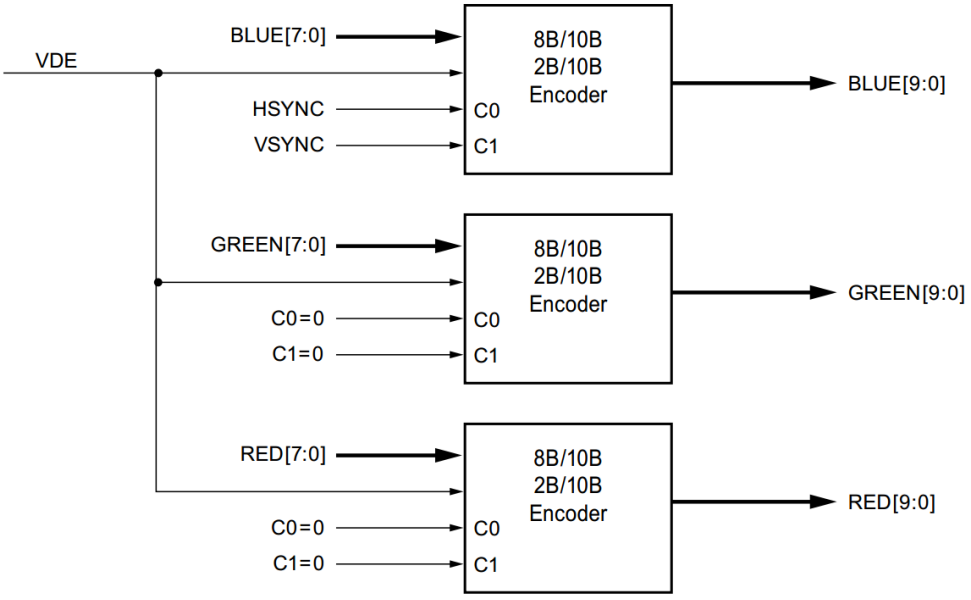


图 17.1.5 DVI 编码器示意图

每个通道输入的视频像素数据都要使用 DVI 规范中的 TMDS 编码算法进行编码。每个 8-bit 的数据都将被转换成 460 个特定 10-bit 字符中的一个。这个编码机制大致上实现了传输过程中的直流平衡，即一段时

间内传输的高电平（数字“1”）的个数大致等于低电平（数字“0”）的个数。同时，每个编码后的 10-bit 字符中状态跳转（“由 1 到 0”或者“由 0 到 1”）的次数将被限制在五次以内。

除了视频数据之外，每个通道 2-bit 控制信号的状态也要进行编码，编码后分别对应四个不同的 10-bit 控制字符，分别是 10'b1101010100，10'b0010101011，10'b0101010100，和 10'b1010101011。可以看出，每个控制字符都有七次以上的状态跳转。视频字符和控制字符状态跳转次数的不同将会被用于发送和接收设备的同步。

HDMI 协议与 DVI 协议在很多方面都是相同的，包括物理连接（TMDS）、有效视频编码算法以及控制字符的定义等。但是，相比于 DVI，HDMI 在视频的消隐期会传输更多的数据，包括音频数据和附加数据。4-bit 音频和附加数据将通过 TERC4 编码机制转换成 10-bit TERC4 字符，然后在绿色和红色通道上传输。

HDMI 在输入附加数据的同时，还需要输入 ADE（Aux/Audio Data Enable）信号，其作用和 VDE 是类似的：当 ADE 为高电平时，表明输入端的附加数据或者音频数据有效。如果大家想了解更多有关 HDMI 的细节，可以参考开发板资料(A 盘)/软件资料中的 HDMI 接口规范——《High-Definition Multimedia Interface Specification Version 1.3a》。为了简单起见，我们在这里把 HDMI 接口当作 DVI 接口进行驱动。

在编码之后，3 个通道的 10-bit 字符将进行并串转换，这一过程是使用 7 系列 FPGA 中专用的硬件资源来实现的。ZYNQ PL 部分与 7 系列的 FPGA 是等价的，它提供了专用的并串转换器——OSERDESE2。单一的 OSERDESE2 模块可以实现 8:1 的并串转换，通过位宽扩展可以实现 10:1 和 14:1 的转换率。

17.2 实验任务

本章的实验任务是驱动启明星 ZYNQ 开发板上的 HDMI 接口，在显示器上显示彩条图案。

17.3 硬件设计

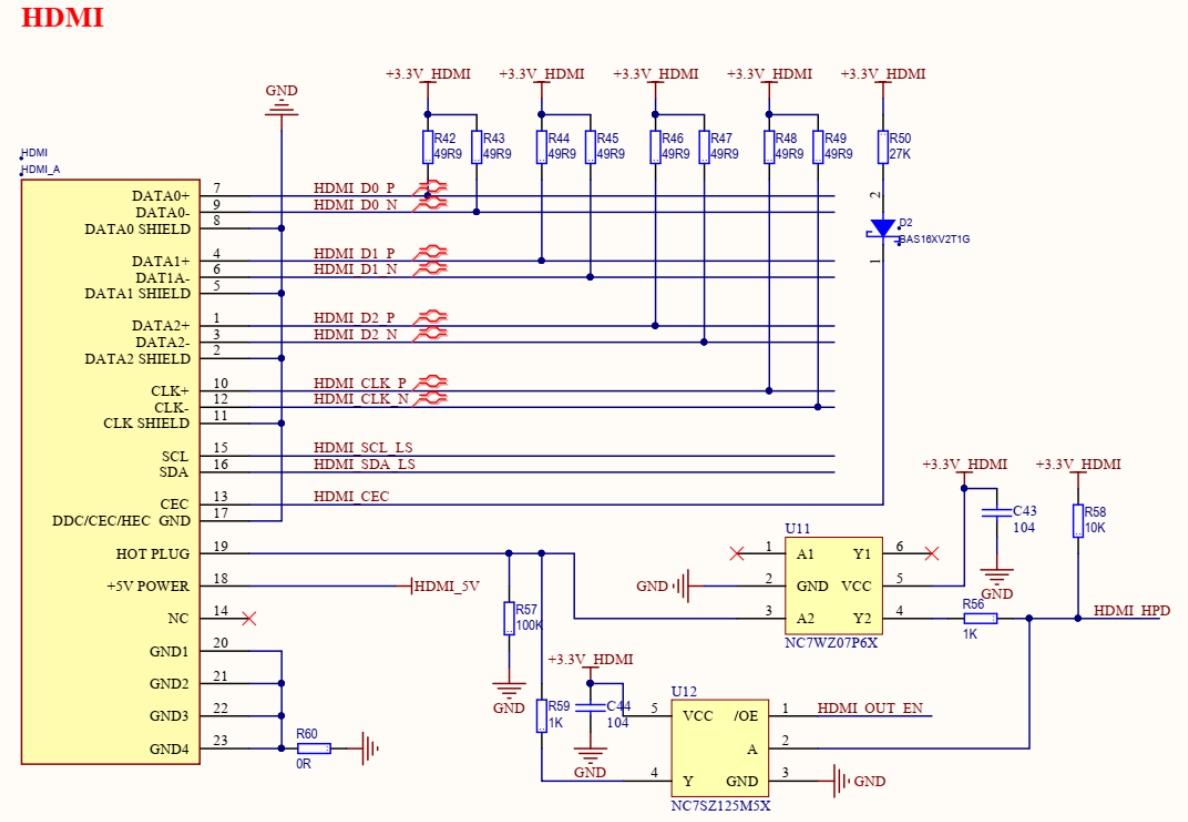


图 17.3.1 HDMI 接口原理图 1

图 17.1.1 是启明星 ZYNQ 底板 HDMI 接口原理图的一部分，其中 HDMI 的三个数据通道 HDMI_D[2:0]至和一个时钟通道 HDMI_CLK 直接与 ZYNQ PL 端的 TMDS 差分引脚相连。

HDMI_CEC 指的是用户电气控制（Consumer Electronics Control），它用于 HDMI 连接线上的设备之间进行信息交换。当一个设备的状态发生变化时，CEC 可以使用远程控制或自动改变设置来命令连接的关联设备的状态发生相应的变化。例如，如果用户放置一张碟片到蓝光播放器并开始播放，那么高清电视机将会自动打开电源，设置正确的视频输入格式和打开环绕声设备等等，这种关联通信提供了一个更好的客户体验。

HDMI_HPD 指的是热拔插检测（Hot Plug Detect），当视频设备与接收设备通过 HDMI 连接时，接收设备将 HPD 置为高电平，通知发送设备。当发送设备检测到 HPD 为低电平时，表明断开连接。

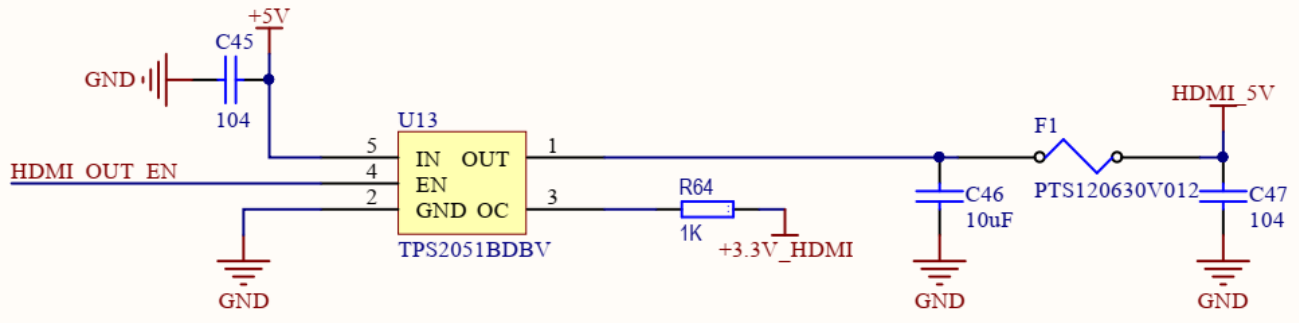


图 17.3.2 HDMI 接口原理图 2

在图 17.3.2 中，HDMI_OUT_EN 信号用于设置 HDMI 接口的输入输出模式，当其为高电平时作为输出端，此时由 ZYNQ 底板输出 HDMI 接口的 5V 电源。同时，图 17.3.1 中 HDMI_HPD 将作为输入信号使用；反之，当 HDMI_OUT_EN 为低电平时，开发板上的 HDMI 接口作为输入端，接口上的 5V 电源将由外部连接设备提供。同时，HDMI_HPD 将输出高电平，用于指示 HDMI 连接状态。

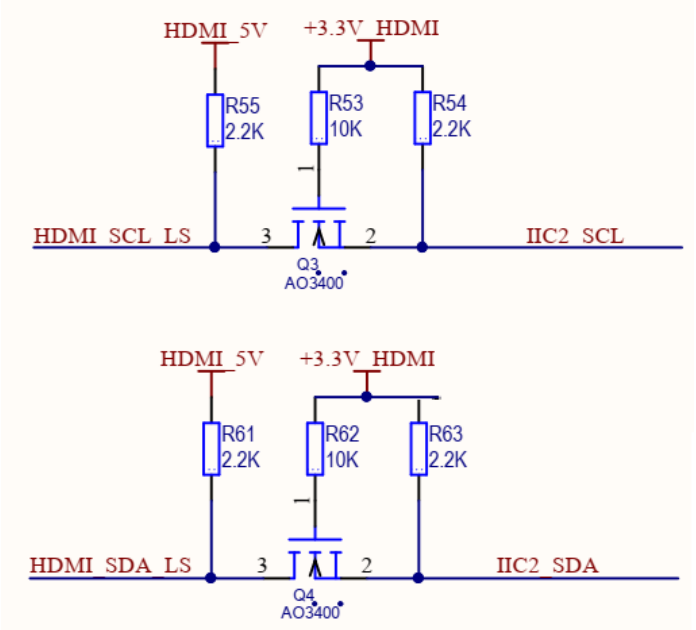


图 17.3.3 HDMI 接口原理图 3

在图 17.3.3 中，HDMI_SCL_LS 和 HDMI_SDA_LS 是 HDMI 接口的显示数据通道（DDC，Display Data Channel），用于 HDMI 发送端和接收端之间交换一些配置信息，通过 I2C 协议通信。发送端通过 DDC 通道，读取接收端保存在 EEPROM 中的 EDID 数据，获取接收端的信息，确认接收端终端显示的设置和功能，

决定跟接收端之间以什么格式传输音/视频数据。

本次实验只使用了 HDMI 接口的 TMDS 数据、TMDS 时钟以及 HDMI 输出使能等信号，各端口的管脚分配如下表所示：

表 17.3.1 HDMI彩条显示实验管脚分配

信号名	方向	管脚	端口说明	电平标准
tmds_data_p[2]	output	L16	TMDS 数据通道2（正极）	TMDS_33
tmds_data_p[1]	output	M14	TMDS 数据通道1（正极）	TMDS_33
tmds_data_p[0]	output	K19	TMDS 数据通道0（正极）	TMDS_33
tmds_clk_p	output	L14	TMDS 时钟通道（正极）	TMDS_33
tmds_oen	output	G17	TMDS 输出使能	LVCMOS33
sys_clk	input	U18	系统时钟，50M	LVCMOS33
sys_rst_n	input	J15	系统复位，低有效	LVCMOS33

需要注意的是，TMDS 数据和时钟信号需要在约束文件中指定电平标准为 TMDS_33。另外，对于差分信号我们只需要指定正极的引脚位置，工具会自动对负极进行管脚分配。

相关的管脚约束如下所示：

```
set_property -dict {PACKAGE_PIN L16  IOSTANDARD TMDS_33 } [get_ports {tmds_data_p[2]}]
set_property -dict {PACKAGE_PIN M14  IOSTANDARD TMDS_33 } [get_ports {tmds_data_p[1]}]
set_property -dict {PACKAGE_PIN K19  IOSTANDARD TMDS_33 } [get_ports {tmds_data_p[0]}]
set_property -dict {PACKAGE_PIN L14  IOSTANDARD TMDS_33 } [get_ports tmds_clk_p]

set_property -dict {PACKAGE_PIN G17  IOSTANDARD LVCMOS33} [get_ports tmds_oen]
set_property -dict {PACKAGE_PIN U18  IOSTANDARD LVCMOS33} [get_ports sys_clk]
set_property -dict {PACKAGE_PIN J15  IOSTANDARD LVCMOS33} [get_ports sys_rst_n]
```

17.4 程序设计

由于本次实验只需要通过 HDMI 接口显示图像，因此将其当成 DVI 接口进行驱动。另外我们只需要实现图像的发送功能。由此得出本次实验的系统框图如下所示：

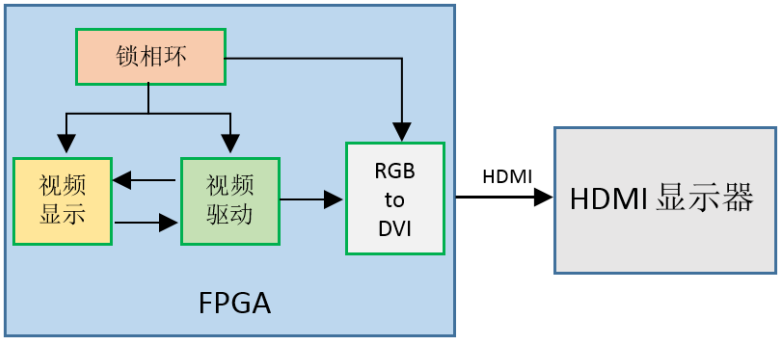


图 17.4.1 系统框图

本次实验在 LCD 彩条显示实验的基础上添加一个 RGB2DVI 模块，将 RGB888 格式的视频图像转换成 TMDS 数据输出。本章的重点是介绍 RGB2DVI 模块，其余模块的介绍请参考 LCD 彩条显示实验。

RGB2DVI 顶层模块的设计框图如下所示：

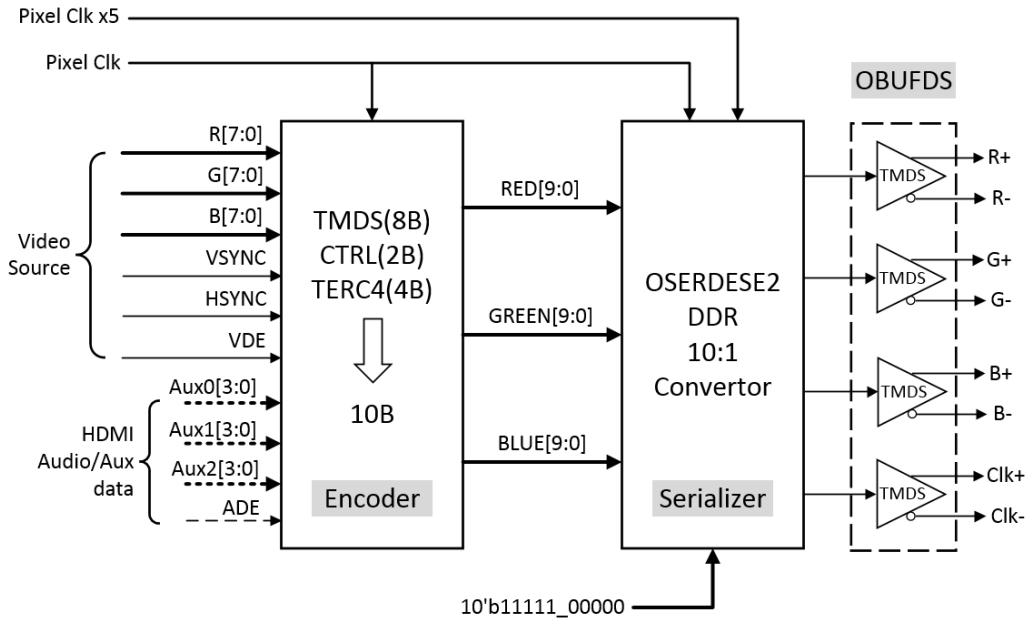


图 17.4.2 RGB2DVI 模块框图

图 17.4.2 中，Encoder 模块负责对数据进行编码，Serializer 模块对编码后的数据进行并串转换，最后通过 OBUFDS 转化成 TMDS 差分信号传输。

整个系统需要两个输入时钟，一个是视频的像素时钟 Pixel Clk，另外一个时钟 Pixel Clk x5 的频率是像素时钟的五倍。由前面的简介部分我们知道，并串转换过程的实现的是 10:1 的转换率，理论上转换器需要一个 10 倍像素时钟频率的串行时钟。这里我们只用了一个 5 倍的时钟频率，这是因为 OSERDESE2 模块可以实现 DDR 的功能，即它在五倍时钟频率的基础上又实现了双倍数据速率。

TMDS 连接的时钟通道我们采用与数据通道相同的并转串逻辑来实现。通过对 10 位二进制序列 10’b11111_00000 在 10 倍像素时钟频率下进行并串转换，就可以得到像素时钟频率下的 TMDS 参考时钟。

另外需要注意的是，图中左下脚 HDMI 的音频/附加数据输入在本次实验中并未用到，因此以虚线表示。

首先来看一下 Encoder 模块的代码：

```
1   `timescale 1 ps / 1ps
2
3   module dvi_encoder (
4     input            clkin,    // pixel clock input
5     input            rstin,    // async. reset input (active high)
6     input      [7:0] din,      // data inputs: expect registered
7     input            c0,       // c0 input
8     input            c1,       // c1 input
9     input            de,       // de input
10    output reg [9:0] dout      // data outputs
11  );
12
```

```
13    ////////////////////////////////////////////////////////////////
14    // Counting number of 1s and 0s for each incoming pixel
15    // component. Pipe line the result.
16    // Register Data Input so it matches the pipe lined adder
17    // output
18    ////////////////////////////////////////////////////////////////
19    reg [3:0] n1d; //number of 1s in din
20    reg [7:0] din_q;
21
22  //计算像素数据中“1”的个数
23    always @ (posedge clkin) begin
24      n1d <=#1 din[0]+din[1]+din[2]+din[3]+din[4]+din[5]+din[6]+din[7];
25
26      din_q <=#1 din;
27    end
28
29    ////////////////////////////////////////////////////////////
30    // Stage 1: 8 bit -> 9 bit
31    // Refer to DVI 1.0 Specification, page 29, Figure 3-5
32    ////////////////////////////////////////////////////////////
33    wire decision1;
34
35    assign decision1 = (n1d > 4'h4) | ((n1d == 4'h4) & (din_q[0] == 1'b0));
36
37    wire [8:0] q_m;
38    assign q_m[0] = din_q[0];
39    assign q_m[1] = (decision1) ? (q_m[0] ^~ din_q[1]) : (q_m[0] ^ din_q[1]);
40    assign q_m[2] = (decision1) ? (q_m[1] ^~ din_q[2]) : (q_m[1] ^ din_q[2]);
41    assign q_m[3] = (decision1) ? (q_m[2] ^~ din_q[3]) : (q_m[2] ^ din_q[3]);
42    assign q_m[4] = (decision1) ? (q_m[3] ^~ din_q[4]) : (q_m[3] ^ din_q[4]);
43    assign q_m[5] = (decision1) ? (q_m[4] ^~ din_q[5]) : (q_m[4] ^ din_q[5]);
44    assign q_m[6] = (decision1) ? (q_m[5] ^~ din_q[6]) : (q_m[5] ^ din_q[6]);
45    assign q_m[7] = (decision1) ? (q_m[6] ^~ din_q[7]) : (q_m[6] ^ din_q[7]);
46    assign q_m[8] = (decision1) ? 1'b0 : 1'b1;
47
48    ////////////////////////////////////////////////////////////
49    // Stage 2: 9 bit -> 10 bit
50    // Refer to DVI 1.0 Specification, page 29, Figure 3-5
51    ////////////////////////////////////////////////////////////
52    reg [3:0] n1q_m, n0q_m; // number of 1s and 0s for q_m
53    always @ (posedge clkin) begin
```

```
54      n1q_m  <=#1 q_m[0]+q_m[1]+q_m[2]+q_m[3]+q_m[4]+q_m[5]+q_m[6]+q_m[7];
55      n0q_m  <=#1 4'h8 - (q_m[0]+q_m[1]+q_m[2]+q_m[3]+q_m[4]+q_m[5]+q_m[6]+q_m[7]);
56    end
57
58    parameter CTRLTOKEN0 = 10'b1101010100;
59    parameter CTRLTOKEN1 = 10'b0010101011;
60    parameter CTRLTOKEN2 = 10'b0101010100;
61    parameter CTRLTOKEN3 = 10'b1010101011;
62
63    reg [4:0] cnt; //disparity counter, MSB is the sign bit
64    wire decision2, decision3;
65
66    assign decision2 = (cnt == 5'h0) | (n1q_m == n0q_m);
67    ////////////////////////////////////////////////////////////////////////
68    // [(cnt > 0) and (N1q_m > N0q_m)] or [(cnt < 0) and (N0q_m > N1q_m)]
69    ////////////////////////////////////////////////////////////////////////
70    assign decision3 = (~cnt[4] & (n1q_m > n0q_m)) | (cnt[4] & (n0q_m > n1q_m));
71
72    ////////////////////////////////////
73    // pipe line alignment
74    ////////////////////////////////////
75    reg       de_q, de_reg;
76    reg       c0_q, c1_q;
77    reg       c0_reg, c1_reg;
78    reg [8:0] q_m_reg;
79
80    always @ (posedge clkin) begin
81      de_q    <=#1 de;
82      de_reg  <=#1 de_q;
83
84      c0_q    <=#1 c0;
85      c0_reg  <=#1 c0_q;
86      c1_q    <=#1 c1;
87      c1_reg  <=#1 c1_q;
88
89      q_m_reg <=#1 q_m;
90    end
91
92    ////////////////////////////////
93    // 10-bit out
94    // disparity counter
```

```
95    ////////////////////////////////
96    always @ (posedge clkin or posedge rstin) begin
97      if(rstin) begin
98        dout <= 10'h0;
99        cnt <= 5'h0;
100     end else begin
101       if (de_reg) begin
102         if(decision2) begin
103           dout[9]   <=#1 ~q_m_reg[8];
104           dout[8]   <=#1 q_m_reg[8];
105           dout[7:0] <=#1 (q_m_reg[8]) ? q_m_reg[7:0] : ~q_m_reg[7:0];
106
107           cnt <=#1 (~q_m_reg[8]) ? (cnt + n0q_m - n1q_m) : (cnt + n1q_m - n0q_m);
108         end else begin
109           if(decision3) begin
110             dout[9]   <=#1 1'b1;
111             dout[8]   <=#1 q_m_reg[8];
112             dout[7:0] <=#1 ~q_m_reg[7:0];
113
114             cnt <=#1 cnt + {q_m_reg[8], 1'b0} + (n0q_m - n1q_m);
115           end else begin
116             dout[9]   <=#1 1'b0;
117             dout[8]   <=#1 q_m_reg[8];
118             dout[7:0] <=#1 q_m_reg[7:0];
119
120             cnt <=#1 cnt - {~q_m_reg[8], 1'b0} + (n1q_m - n0q_m);
121           end
122         end
123       end else begin
124         case ({c1_reg, c0_reg})
125           2'b00:   dout <=#1 CTRLTOKEN0;
126           2'b01:   dout <=#1 CTRLTOKEN1;
127           2'b10:   dout <=#1 CTRLTOKEN2;
128           default: dout <=#1 CTRLTOKEN3;
129         endcase
130
131         cnt <=#1 5'h0;
132       end
133     end
134   end
135
```

```
136 endmodule
```

dvi_encoder 模块按照 DVI 接口规范中 TMDS 编码算法对输入的 8 位像素数据以及 2 位行场同步信号进行编码。该模块是 Xilinx 应用笔记 XAPP460 中所提供的编码模块，其具体实现的编码算法如下图所示：

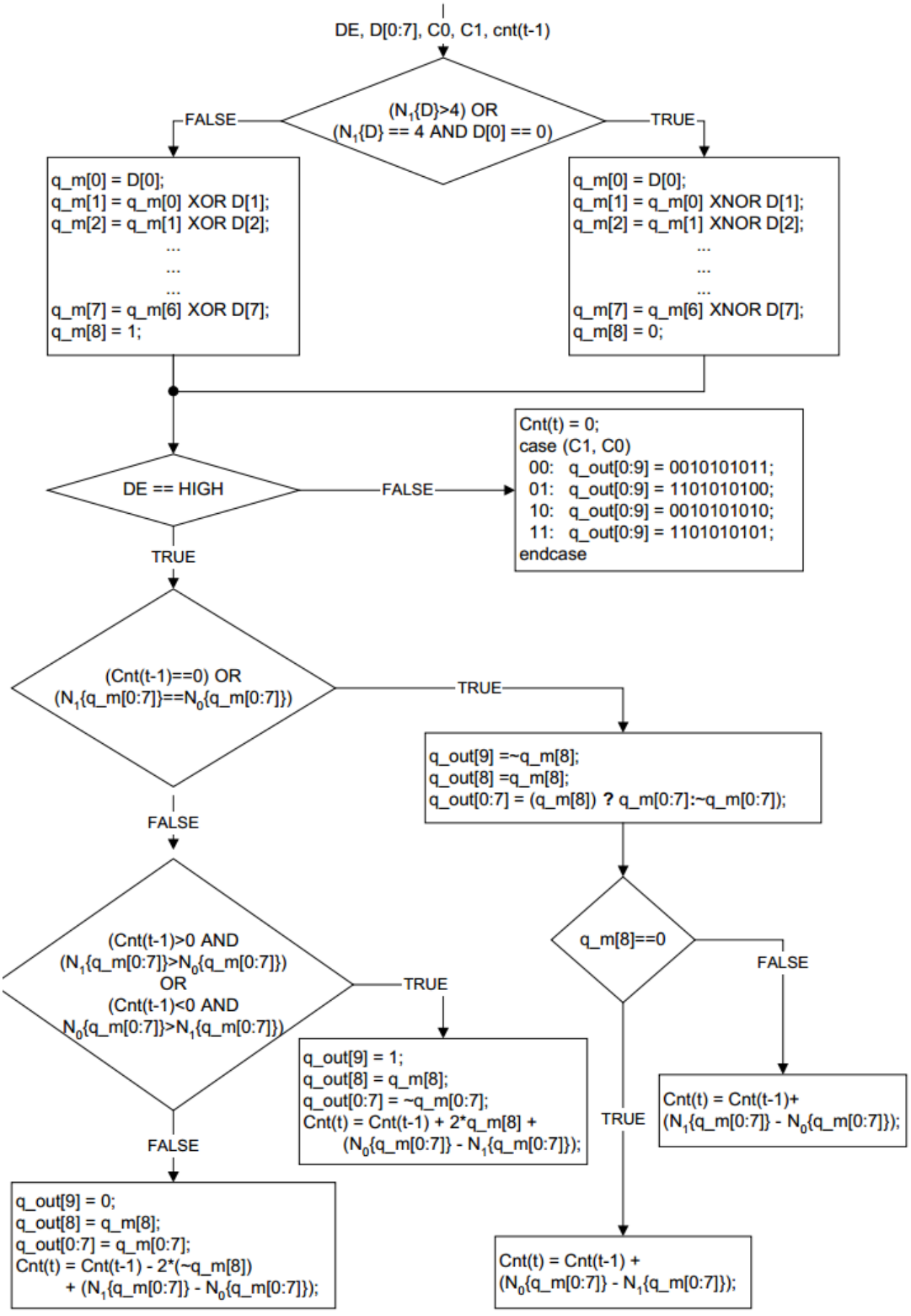


图 17.4.3 TMDS 编码算法

TMDS 通过逻辑算法将 8 位字符数据通过最小转换编码为 10 位字符数据，前 8 位数据由原始信号经运

算后获得，第 9 位表示运算的方式，1 表示异或 0 表示异或非。经过 DC 平衡后（第 10 位），采用差分信号传输数据。第 10 位实际是一个反转标志位，1 表示进行了反转而 0 表示没有反转，从而达到 DC 平衡。

接收端在收到信号后，再进行相反的运算。TMDS 和 LVDS、TTL 相比有较好的电磁兼容性能。这种算法可以减小传输信号过程的上冲和下冲，而 DC 平衡使信号对传输线的电磁干扰减少，可以用低成本的专用电缆实现长距离、高质量的数字信号传输。

图 17.4.3 所描述的算法是 DVI 接口规范所定义的，我们不作深入研究，大家有兴趣的话也可以对照 dvi_encoder 模块中的代码来分析整个算法流程是如何使用 Verilog 来实现的。算法中各个参数的含义如下图所示：

D C1 C0 DE	D 为视频信号，C 为控制信号，DE 为使能信号
Cnt	Cnt 为寄存器参数
N1{X}	输入视频信号“1”的个数
N0{X}	输入视频信号“0”的个数
q_out	编码输出

图 17.4.4 TMDS 编码算法的参数

TMDS 编码之后的数据由 Serializer 模块进行并串转换，代码如下所示：

```
1   `timescale 1ns / 1ps
2
3   module serializer_10_to_1(
4       input           reset,              // 复位,高有效
5       input           paralell_clk,       // 输入并行数据时钟
6       input           serial_clk_5x,      // 输入串行数据时钟
7       input   [9:0]   paralell_data,      // 输入并行数据
8
9       output          serial_data_out     // 输出串行数据
10      );
11
12  //wire define
13  wire        cascade1;     //用于两个 OSERDESE2 级联的信号
14  wire        cascade2;
15
16  //*********************************************************
17  //**                    main code
18  //*********************************************************
19
20  //例化 OSERDESE2 原语，实现并串转换,Master 模式
21  OSERDESE2 #(
22      .DATA_RATE_OQ   ("DDR"),        // 设置双倍数据速率
23      .DATA_RATE_TQ   ("SDR"),        // DDR, BUF, SDR
24      .DATA_WIDTH     (10),           // 输入的并行数据宽度为 10bit
```

```
25      .SERDES_MODE     ("MASTER"),     // 设置为 Master，用于 10bit 宽度扩展
26      .TBYTE_CTL       ("FALSE"),      // Enable tristate byte operation (FALSE, TRUE)
27      .TBYTE_SRC       ("FALSE"),      // Tristate byte source (FALSE, TRUE)
28      .TRISTATE_WIDTH (1)              // 3-state converter width (1,4)
29  )
30  OSERDESE2_Master (
31      .CLK         (serial_clk_5x),    // 串行数据时钟,5 倍时钟频率
32      .CLKDIV      (paralell_clk),     // 并行数据时钟
33      .RST         (reset),            // 1-bit input: Reset
34      .OCE         (1'b1),             // 1-bit input: Output data clock enable
35
36      .OQ          (serial_data_out),  // 串行输出数据
37
38      .D1          (paralell_data[0]), // D1 - D8: 并行数据输入
39      .D2          (paralell_data[1]),
40      .D3          (paralell_data[2]),
41      .D4          (paralell_data[3]),
42      .D5          (paralell_data[4]),
43      .D6          (paralell_data[5]),
44      .D7          (paralell_data[6]),
45      .D8          (paralell_data[7]),
46
47      .SHIFTIN1    (cascade1),         // SHIFTIN1 用于位宽扩展
48      .SHIFTIN2    (cascade2),         // SHIFTIN2
49      .SHIFTOUT1   (),                 // SHIFTOUT1: 用于位宽扩展
50      .SHIFTOUT2   (),                 // SHIFTOUT2
51
52      .OFB         (),                 // 以下是未使用信号
53      .T1          (1'b0),
54      .T2          (1'b0),
55      .T3          (1'b0),
56      .T4          (1'b0),
57      .TBYTEIN     (1'b0),
58      .TCE         (1'b0),
59      .TBYTEOUT    (),
60      .TFB         (),
61      .TQ          ()
62  );
63
64  //例化 OSERDESE2 原语，实现并串转换,Slave 模式
65  OSERDESE2 #(
```

```
66      .DATA_RATE_OQ   ("DDR"),          // 设置双倍数据速率
67      .DATA_RATE_TQ   ("SDR"),          // DDR, BUF, SDR
68      .DATA_WIDTH     (10),             // 输入的并行数据宽度为 10bit
69      .SERDES_MODE    ("SLAVE"),        // 设置为 Slave，用于 10bit 宽度扩展
70      .TBYTE_CTL      ("FALSE"),        // Enable tristate byte operation (FALSE, TRUE)
71      .TBYTE_SRC      ("FALSE"),        // Tristate byte source (FALSE, TRUE)
72      .TRISTATE_WIDTH (1)               // 3-state converter width (1,4)
73  )
74  OSERDESE2_Slave (
75      .CLK         (serial_clk_5x),     // 串行数据时钟,5 倍时钟频率
76      .CLKDIV      (paralell_clk),      // 并行数据时钟
77      .RST         (reset),             // 1-bit input: Reset
78      .OCE         (1'b1),              // 1-bit input: Output data clock enable
79
80      .OQ          (),                  // 串行输出数据
81
82      .D1          (1'b0),              // D1 - D8: 并行数据输入
83      .D2          (1'b0),
84      .D3          (paralell_data[8]),
85      .D4          (paralell_data[9]),
86      .D5          (1'b0),
87      .D6          (1'b0),
88      .D7          (1'b0),
89      .D8          (1'b0),
90
91      .SHIFTIN1    (),                  // SHIFTIN1 用于位宽扩展
92      .SHIFTIN2    (),                  // SHIFTIN2
93      .SHIFTOUT1   (cascade1),          // SHIFTOUT1: 用于位宽扩展
94      .SHIFTOUT2   (cascade2),          // SHIFTOUT2
95
96      .OFB         (),                  // 以下是未使用信号
97      .T1          (1'b0),
98      .T2          (1'b0),
99      .T3          (1'b0),
100     .T4          (1'b0),
101     .TBYTEIN     (1'b0),
102     .TCE         (1'b0),
103     .TBYTEOUT    (),
104     .TFB         (),
105     .TQ          ()
106 );
```

```
107
108 endmodule
```

serdes_10_to_1 模块通过调用 OSERDESE2 原语来实现 10:1 的并串转换。原语是 Xilinx 器件底层硬件中的功能模块，它使用专用的资源来实现一系列的功能。相比于 IP 核，原语的调用方法更简单，但是一般只用于实现一些简单的功能。

需要注意的是，一个 OSERDESE2 只能实现最多 8:1 的转换率，在这里我们通过位宽扩展实现了 10:1 的并串转换，如下图所示：

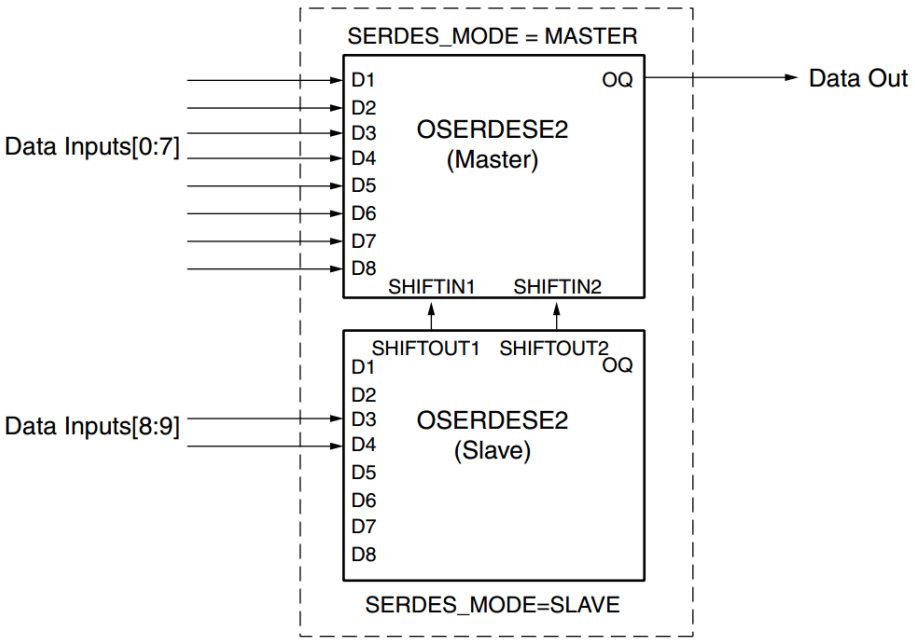


图 17.4.5 OSERDESE2 位宽扩展

如图 17.4.5 所示，OSERDESE2 位宽扩展通过两个 OSERDESE2 模块来实现，其中一个作为 Master，另一个作为 Slave，通过这种方式最多可实现 14:1 的并串转换。需要注意的是，在位宽扩展时，Slave 模块的数据输入端只能使用 D3 至 D8。

最后，我们在 RGB2DVI 顶层模块中调用上述两个模块，其代码如下所示：

```
1  module dvi_transmitter_top(
2      input        pclk,          // pixel clock
3      input        pclk_x5,       // pixel clock x5
4      input        reset_n,       // reset
5
6      input [23:0] video_din,     // RGB888 video in
7      input        video_hsync,   // hsync data
8      input        video_vsync,   // vsync data
9      input        video_de,      // data enable
10
11     output       tmds_clk_p,    // TMDS 时钟通道
12     output       tmds_clk_n,
13     output [2:0] tmds_data_p,   // TMDS 数据通道
14     output [2:0] tmds_data_n,
```

```
15      output          tmds_oen        // TMDS 输出使能
16      );
17
18  //wire define
19  wire        reset;
20
21  //并行数据
22  wire [9:0]  red_10bit;
23  wire [9:0]  green_10bit;
24  wire [9:0]  blue_10bit;
25  wire [9:0]  clk_10bit;
26
27  //串行数据
28  wire [2:0]  tmds_data_serial;
29  wire        tmds_clk_serial;
30
31  //*****************************************************
32  //**                    main code
33  //*****************************************************
34  assign tmds_oen = 1'b1;
35  assign clk_10bit = 10'b1111100000;
36
37  //异步复位，同步释放
38  asyn_rst_syn reset_syn(
39      .reset_n    (reset_n),
40      .clk        (pclk),
41
42      .syn_reset  (reset)     //高有效
43      );
44
45  //对三个颜色通道进行编码
46  dvi_encoder encoder_b (
47      .clkin      (pclk),
48      .rstin      (reset),
49
50      .din        (video_din[7:0]),
51      .c0         (video_hsync),
52      .c1         (video_vsync),
53      .de         (video_de),
54      .dout       (blue_10bit)
55      ) ;
```

```
56
57  dvi_encoder encoder_g (
58      .clkin      (pclk),
59      .rstin      (reset),
60
61      .din        (video_din[15:8]),
62      .c0         (1'b0),
63      .c1         (1'b0),
64      .de         (video_de),
65      .dout       (green_10bit)
66      ) ;
67
68  dvi_encoder encoder_r (
69      .clkin      (pclk),
70      .rstin      (reset),
71
72      .din        (video_din[23:16]),
73      .c0         (1'b0),
74      .c1         (1'b0),
75      .de         (video_de),
76      .dout       (red_10bit)
77      ) ;
78
79  //对编码后的数据进行并串转换
80  serializer_10_to_1 serializer_b(
81      .reset              (reset),                // 复位,高有效
82      .paralell_clk       (pclk),                 // 输入并行数据时钟
83      .serial_clk_5x      (pclk_x5),              // 输入串行数据时钟
84      .paralell_data      (blue_10bit),           // 输入并行数据
85
86      .serial_data_out    (tmds_data_serial[0])   // 输出串行数据
87      );
88
89  serializer_10_to_1 serializer_g(
90      .reset              (reset),
91      .paralell_clk       (pclk),
92      .serial_clk_5x      (pclk_x5),
93      .paralell_data      (green_10bit),
94
95      .serial_data_out    (tmds_data_serial[1])
96      );
```

```
97
98  serializer_10_to_1 serializer_r(
99      .reset              (reset),
100     .paralell_clk       (pclk),
101     .serial_clk_5x      (pclk_x5),
102     .paralell_data      (red_10bit),
103
104     .serial_data_out    (tmds_data_serial[2])
105     );
106
107 serializer_10_to_1 serializer_clk(
108     .reset              (reset),
109     .paralell_clk       (pclk),
110     .serial_clk_5x      (pclk_x5),
111     .paralell_data      (clk_10bit),
112
113     .serial_data_out    (tmds_clk_serial)
114     );
115
116 //转换差分信号
117 OBUFDS #(
118     .IOSTANDARD         ("TMDS_33")     // I/O 电平标准为 TMDS
119 ) TMDS0 (
120     .I                  (tmds_data_serial[0]),
121     .O                  (tmds_data_p[0]),
122     .OB                 (tmds_data_n[0])
123     );
124
125 OBUFDS #(
126     .IOSTANDARD         ("TMDS_33")     // I/O 电平标准为 TMDS
127 ) TMDS1 (
128     .I                  (tmds_data_serial[1]),
129     .O                  (tmds_data_p[1]),
130     .OB                 (tmds_data_n[1])
131     );
132
133 OBUFDS #(
134     .IOSTANDARD         ("TMDS_33")     // I/O 电平标准为 TMDS
135 ) TMDS2 (
136     .I                  (tmds_data_serial[2]),
137     .O                  (tmds_data_p[2]),
```

```
138     .OB                 (tmds_data_n[2])
139     );
140
141 OBUFDS #(
142     .IOSTANDARD         ("TMDS_33")     // I/O 电平标准为 TMDS
143 ) TMDS3 (
144     .I                  (tmds_clk_serial),
145     .O                  (tmds_clk_p),
146     .OB                 (tmds_clk_n)
147     );
148
149 endmodule
```

在 dvi_transmitter_top 模块中，不仅例化了编码模块和并转串模块，同时还例化了四个 OBUFDS 原语，用于将三路数据和一路时钟信号转换成差分信号输出，如程序第 116 至 147 行所示。

OBUFDS 是差分输出缓冲器，用于将来自 FPGA 内部逻辑的信号转换成差分信号输出，支持 TMDS 电平标准。OBUFDS 原语示意图如下所示：

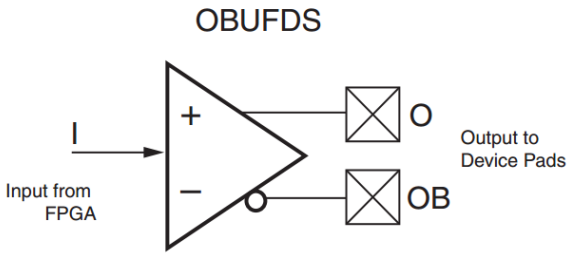


图 17.4.6 OBUFDS 原语示意图

在程序的第 37 至 43 行，例化了 asyn_rst_syn 模块。OSERDESE2 模块要求复位信号高电平有效，并且需要将异步复位信号同步到串行时钟域。因此，我们在 asyn_rst_syn 模块中将低电平有效的异步复位信号转换成高有效，同时对其进行异步复位，同步释放处理。

asyn_rst_syn 模块的代码如下所示：

```
1  module asyn_rst_syn(
2      input clk,          //目的时钟域
3      input reset_n,      //异步复位，低有效
4
5      output syn_reset    //高有效
6      );
7
8  //reg define
9  reg reset_1;
10 reg reset_2;
11
12 //**************************************************
13 //**                    main code
```

```
14 //*****************************************************
15 assign syn_reset  = reset_2;
16
17 //对异步复位信号进行同步释放，并转换成高有效
18 always @ (posedge clk or negedge reset_n) begin
19     if(!reset_n) begin
20         reset_1 <= 1'b1;
21         reset_2 <= 1'b1;
22     end
23     else begin
24         reset_1 <= 1'b0;
25         reset_2 <= reset_1;
26     end
27 end
28
29 endmodule
```

可以看出，该模块的代码非常简单，相当于在需要同步的时钟域下对输入的异步复位信号连接寄存了两次，这是一种非常常用的对异步信号进行同步的方法。需要注意的是，在程序第 18 行的 always 块中，还实现了将低电平有效的复位信号转换成高电平有效的功能。

到这里 RGB2DVI 模块的程序设计就介绍完了，整个模块的原理图如下所示：

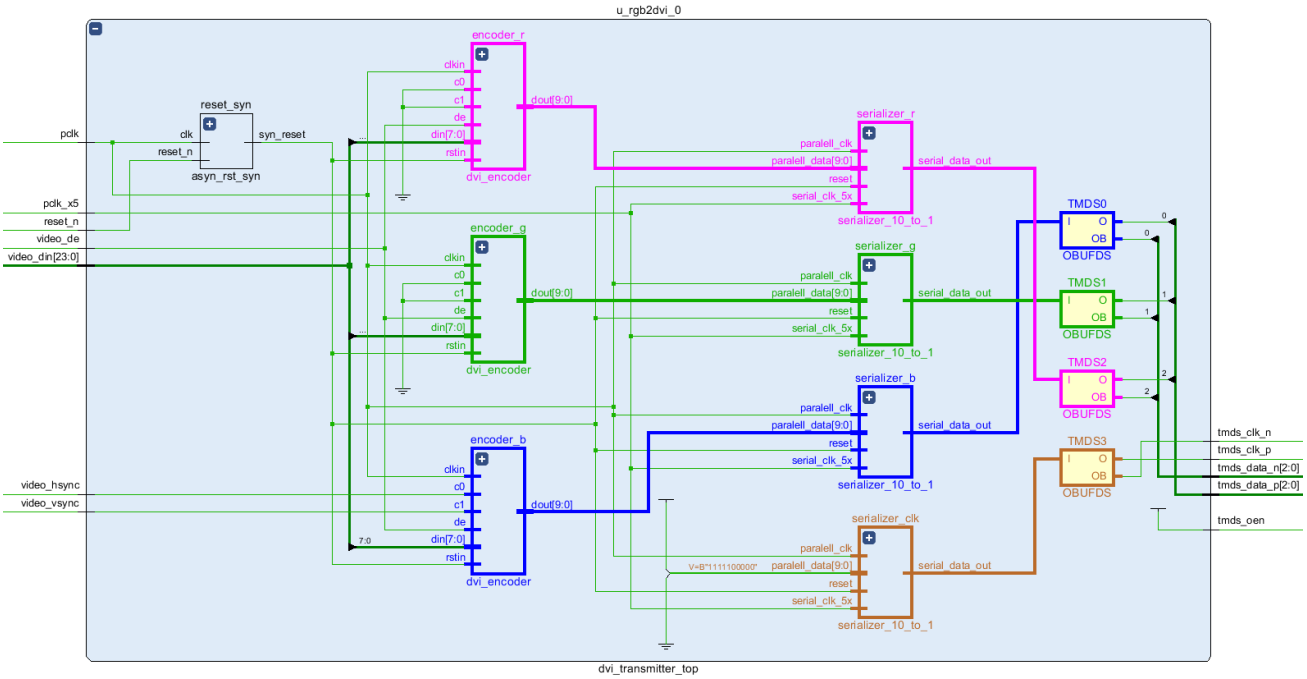


图 17.4.7 RGB2DVI 模块原理图

在图 17.4.2 中，我们用红/绿/蓝三种颜色分别标识出了输入的视频数据 video_din 中三个不同的颜色通道。从图中也可以看出，每个颜色通道的处理过程都是一样的，都是先经过 dvi_encoder 进行编码，然后经过 serializer_10_to_1 模块进行并串转换，最后通过 OBUFDS 原语转换成 TMDS 差分信号。

接下来，我们在整个系统的顶层模块中调用 RGB2DVI 模块，通过 HDMI 接口输出彩条图案。

系统的顶层模块为 hdmi_colorbar_top，其代码如下所示：

```
module  hdmi_colorbar_top(
    input         sys_clk,
    input         sys_rst_n,

    output        tmds_clk_p,    // TMDS 时钟通道
    output        tmds_clk_n,
    output [2:0]  tmds_data_p,   // TMDS 数据通道
    output [2:0]  tmds_data_n,
    output        tmds_oen       // TMDS 输出使能
);

//wire define
wire          pixel_clk;
wire          pixel_clk_5x;
wire          clk_locked;

wire  [10:0]  pixel_xpos_w;
wire  [10:0]  pixel_ypos_w;
wire  [23:0]  pixel_data_w;

wire          video_hs;
wire          video_vs;
wire          video_de;
wire  [23:0]  video_rgb;

//*****************************************************
//**                    main code
//*****************************************************

//例化 MMCM/PLL IP 核
clk_wiz_0  clk_wiz_0(
    .clk_in1        (sys_clk),
    .clk_out1       (pixel_clk),        //像素时钟
    .clk_out2       (pixel_clk_5x),     //5 倍像素时钟

    .reset          (~sys_rst_n),
    .locked         (clk_locked)
    );

//例化视频显示驱动模块
```

```
41 video_driver u_video_driver(
42     .pixel_clk      (pixel_clk),
43     .sys_rst_n      (sys_rst_n),
44
45     .video_hs       (video_hs),
46     .video_vs       (video_vs),
47     .video_de       (video_de),
48     .video_rgb      (video_rgb),
49
50     .pixel_xpos     (pixel_xpos_w),
51     .pixel_ypos     (pixel_ypos_w),
52     .pixel_data     (pixel_data_w)
53     );
54
55 //例化视频显示模块
56 video_display  u_video_display(
57     .pixel_clk      (pixel_clk),
58     .sys_rst_n      (sys_rst_n),
59
60     .pixel_xpos     (pixel_xpos_w),
61     .pixel_ypos     (pixel_ypos_w),
62     .pixel_data     (pixel_data_w)
63     );
64
65 //例化 HDMI 驱动模块
66 dvi_transmitter_top u_rgb2dvi_0(
67     .pclk           (pixel_clk),
68     .pclk_x5        (pixel_clk_5x),
69     .reset_n        (sys_rst_n & clk_locked),
70
71     .video_din      (video_rgb),
72     .video_hsync    (video_hs),
73     .video_vsync    (video_vs),
74     .video_de       (video_de),
75
76     .tmds_clk_p     (tmds_clk_p),
77     .tmds_clk_n     (tmds_clk_n),
78     .tmds_data_p    (tmds_data_p),
79     .tmds_data_n    (tmds_data_n),
80     .tmds_oen       (tmds_oen)
81     );
```

```
82
83 endmodule
```

在代码的 30 至 38 行，我们通过调用时钟 IP 核来产生两个时钟，其中 pixel_clk 为像素时钟，而 pixel_clk_5x 为并串转换模块所需要的串行数据时钟，其频率为 pixel_clk 的 5 倍。

在顶层模块中，video_display 模块（第 56 行）负责产生 RGB888 格式的彩条图案，然后在 video_driver 模块（第 41 行）的驱动下按照工业标准的 VGA 显示时序输出视频信号、行场同步信号以及视频有效信号。这些信号作为 RGB2DVI 模块（第 66 行）的输入，最终转换成 DVI/HDMI 接口标准的 TMDS 串行数据输出到 HDMI 接口。

上述代码中的 video_display 模块和 video_driver 模块与 LCD 彩条显示实现中的 lcd_driver 和 lcd_display 模块几乎完全相同，如果大家对这两个模块不熟悉的话，请参考《LCD 彩条显示实验》。

顶层模块 hdmi_colorbar_top 的原理图如下所示：

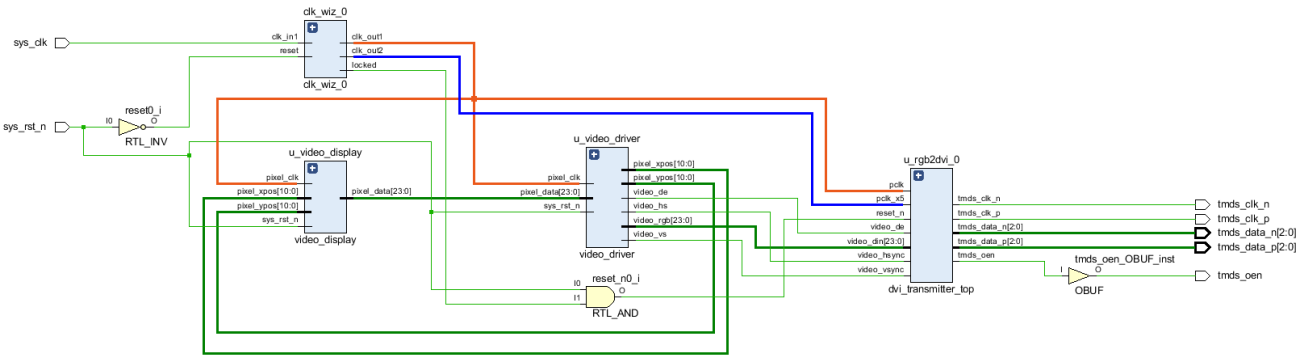


图 17.4.8 顶层模块原理图

在图 17.4.8 中，橙色的线条为像素时钟 pixel_clk，该时钟信号连接到了其他所有模块。而蓝色的线条为串行数据时钟 pixel_clk_5x，该时钟信号仅连接到了 RGB2DVI 模块。

17.5 下载验证

首先我们将下载器与启明星底板上的 JTAG 接口连接，下载器另外一端与电脑连接。然后使用 HDMI 连接线将 HDMI 显示器连接到启明星底板上的 HDMI 接口。最后连接开发板的电源，并打开电源开关，如下图所示：

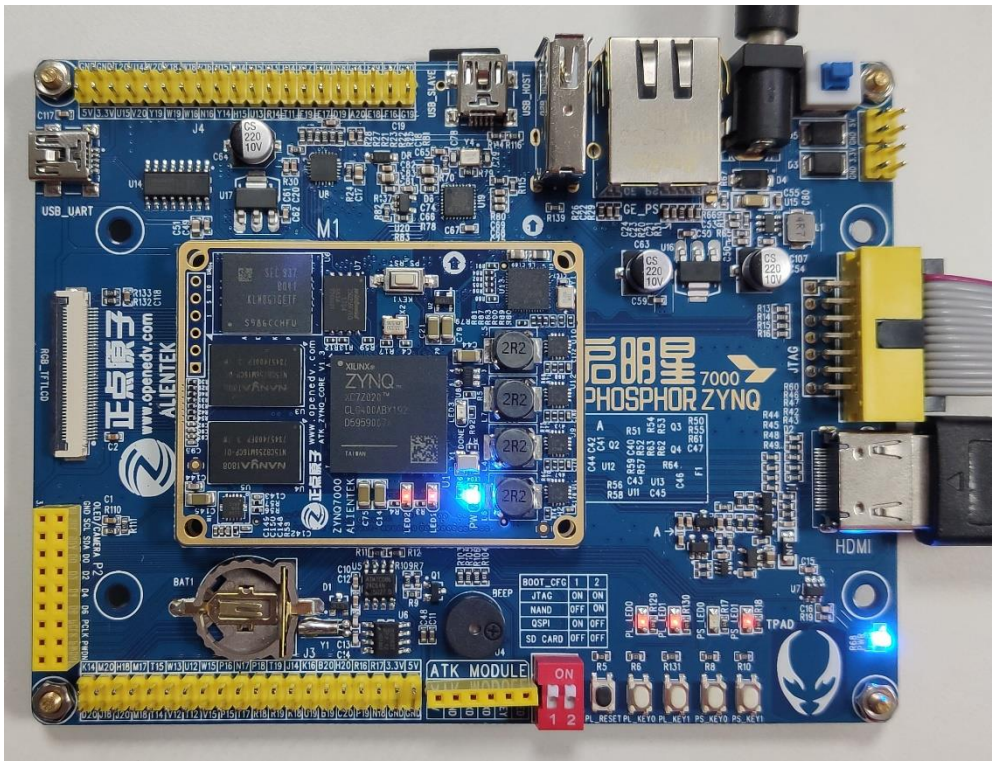


图 17.5.1 启明星开发板连接示意图

然后我们将本次实验生成的 BIT 文件下载下开发板中，下载完成之后 HDMI 显示器上显示彩条图案，说明本次实验在启明星 ZYNQ 开发板上面下载验证成功。

实验结果如下图所示：

图 17.5.2 实验结果

第十八章 HDMI 方块移动实验

在 HDMI 彩条显示实验中，我们成功地在显示器上显示出了静态的彩条图案。本章我们通过在屏幕上显示一个移动的小方块，来给大家演示如何使用 HDMI 接口显示动态图案。

本章包括以下几个部分：

18.1 HDMI 简介

18.2 实验任务

18.3 硬件设计

18.4 程序设计

18.5 下载验证

18.1 HDMI 简介

我们在“HDMI 彩条显示实验”中对 HDMI 视频传输标准作了详细的介绍，包括 HDMI 接口定义、行场同步时序、以及显示分辨率等。如果大家对这部分内容不是很熟悉的话，请参考“HDMI 彩条显示实验”中的 HDMI 简介部分。

18.2 实验任务

本章的实验任务是使用启明星开发板上的 HDMI 接口在显示器上显示一个不停移动的方块，要求方块移动到边界处时能够改变移动方向。显示分辨率为 1280*720，刷新速率为 60hz。

18.3 硬件设计

HDMI 接口部分的硬件设计原理及本实验中各端口信号的管脚分配与“HDMI 彩条显示实验”完全相同，请参考“HDMI 彩条显示实验”中的硬件设计部分。

18.4 程序设计

图 18.4.1 是根据本章实验任务画出的系统框图。其中，时钟生成 IP 核（clk_wiz_0）生成 1280*720 分辨率所需的像素时钟 74.25MHz、以及 TMDS 编码驱动 IP 核所需的像素时钟的 5 倍频率即 74.25 * 5 = 371.25MHz。RGB 驱动模块（rgb_driver）负责产生 RGB 时序，并输出来自 RGB 显示模块（rgb_display）的 RGB 数据。RGB 数据被送入 TMDS 编码驱动 IP 核（rgb2dvi_0）并最后输出到 HDMI 接口。

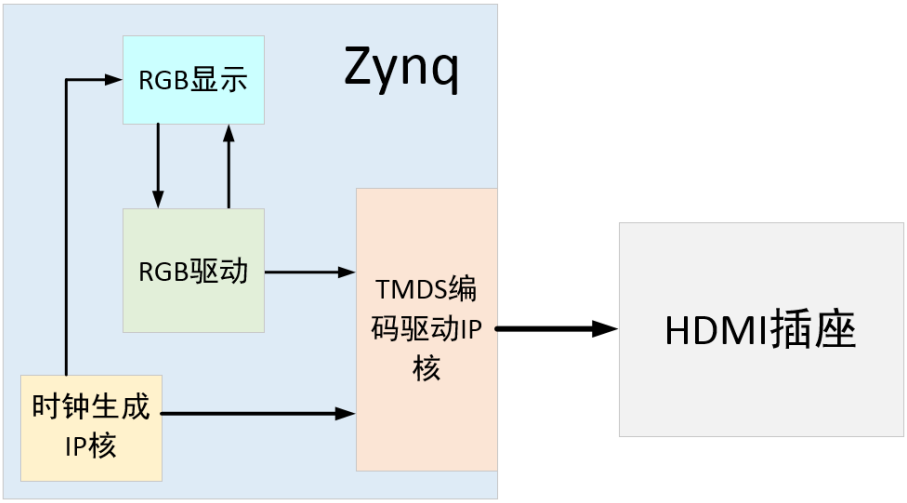


图 18.4.1 HDMI 方块移动实验系统框图

在“HDMI 彩条显示实验”中，我们利用 RGB 驱动模块输出的像素点的横坐标，在 RGB 显示模块中完成了彩条图案的绘制。而在本次实验中，为了完成方块的显示，需要同时使用像素点的横坐标和纵坐标来绘制方块所在的矩形区域，另外还需要知道矩形区域左上角的顶点坐标。由于 RGB 显示的图像在行场同步信号的同步下不停的刷新，因此只要连续改变方块左上角顶点的坐标，并在新的坐标点处重新绘制方块，即可实现方块移动的效果。

各模块端口及信号连接如图 18.4.2 所示：

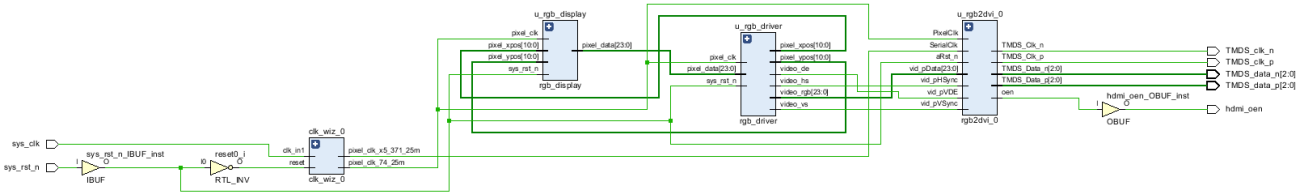


图 18.4.2 顶层模块原理图

图 18.4.2 中的顶层模块（hdmi_block_move_top）、时钟分频 IP 核（clk_wiz_0）、TMDS 编码驱动 IP 核（rgb2dvi_0）以及 RGB 驱动模块（rgb_display）均与“HDMI 彩条显示实验”完全相同，只对 RGB 显示模块（RGB_display）作了修改。因此，这里我们重点讲解 RGB 显示模块，其他部分大家可以参考“HDMI 彩条显示实验”。

RGB 显示模块的代码如下：

```
1   module  rgb_display(
2       input                pixel_clk,              //像素时钟
3       input                sys_rst_n,              //复位信号
4
5       input       [10:0] pixel_xpos,               //像素点横坐标
6       input       [10:0] pixel_ypos,               //像素点纵坐标
7       output  reg [23:0] pixel_data                //像素点数据
8       );
9
10  //parameter define
11  parameter  H_DISP  = 11'd1280;                   //分辨率--行
12  parameter  V_DISP  = 11'd720;                    //分辨率--列
13
14  localparam SIDE_W  = 11'd40;                     //屏幕边框宽度
15  localparam BLOCK_W = 11'd40;                     //方块宽度
16  localparam BLUE    = 24'b00000000_00000000_11111111;  //屏幕边框颜色 蓝色
17  localparam WHITE   = 24'b11111111_11111111_11111111;  //背景颜色 白色
18  localparam BLACK   = 24'b00000000_00000000_00000000;  //方块颜色 黑色
19
20  //reg define
21  reg [10:0] block_x = SIDE_W ;                    //方块左上角横坐标
22  reg [10:0] block_y = SIDE_W ;                    //方块左上角纵坐标
23  reg [21:0] div_cnt;                              //时钟分频计数器
24  reg        h_direct;                             //方块水平移动方向，1：右移，0：左移
25  reg        v_direct;                             //方块竖直移动方向，1：向下，0：向上
26
27  //wire define
28  wire move_en;                                    //方块移动使能信号，频率为 100hz
29
30  //*****************************************************
```

```
31  //**                     main code
32  //*************************************************
33  assign move_en = (div_cnt == 22'd734214) ? 1'b1 : 1'b0;
34
35  //通过对像素时钟计数，实现时钟分频
36  always @(posedge pixel_clk ) begin
37      if (!sys_rst_n)
38          div_cnt <= 22'd0;
39      else begin
40          if(div_cnt < 22'd734214)
41              div_cnt <= div_cnt + 1'b1;
42          else
43              div_cnt <= 22'd0;                    //计数达 10ms 后清零
44      end
45  end
46
47  //当方块移动到边界时，改变移动方向
48  always @(posedge pixel_clk ) begin
49      if (!sys_rst_n) begin
50          h_direct <= 1'b1;                        //方块初始水平向右移动
51          v_direct <= 1'b1;                        //方块初始竖直向下移动
52      end
53      else begin
54          if(block_x == SIDE_W - 1'b1)             //到达左边界时，水平向右
55              h_direct <= 1'b1;
56          else                                     //到达右边界时，水平向左
57          if(block_x == H_DISP - SIDE_W - BLOCK_W)
58              h_direct <= 1'b0;
59          else
60              h_direct <= h_direct;
61
62          if(block_y == SIDE_W - 1'b1)             //到达上边界时，竖直向下
63              v_direct <= 1'b1;
64          else                                     //到达下边界时，竖直向上
65          if(block_y == V_DISP - SIDE_W - BLOCK_W)
66              v_direct <= 1'b0;
67          else
68              v_direct <= v_direct;
69      end
70  end
71
```

```
72  //根据方块移动方向，改变其纵横坐标
73  always @(posedge pixel_clk ) begin
74      if (!sys_rst_n) begin
75          block_x <= SIDE_W;                      //方块初始位置横坐标
76          block_y <= SIDE_W;                      //方块初始位置纵坐标
77      end
78      else if(move_en) begin
79          if(h_direct)
80              block_x <= block_x + 1'b1;          //方块向右移动
81          else
82              block_x <= block_x - 1'b1;          //方块向左移动
83
84          if(v_direct)
85              block_y <= block_y + 1'b1;          //方块向下移动
86          else
87              block_y <= block_y - 1'b1;          //方块向上移动
88      end
89      else begin
90          block_x <= block_x;
91          block_y <= block_y;
92      end
93  end
94
95  //给不同的区域绘制不同的颜色
96  always @(posedge pixel_clk ) begin
97      if (!sys_rst_n)
98          pixel_data <= BLACK;
99      else begin
100         if(  (pixel_xpos < SIDE_W) || (pixel_xpos >= H_DISP - SIDE_W)
101           || (pixel_ypos < SIDE_W) || (pixel_ypos >= V_DISP - SIDE_W))
102             pixel_data <= BLUE;                 //绘制屏幕边框为蓝色
103         else
104         if(  (pixel_xpos >= block_x) && (pixel_xpos < block_x + BLOCK_W)
105           && (pixel_ypos >= block_y) && (pixel_ypos < block_y + BLOCK_W))
106             pixel_data <= BLACK;                //绘制方块为黑色
107         else
108             pixel_data <= WHITE;                //绘制背景为白色
109     end
110 end
111
112 endmodule
```

代码中 14 至 18 行声明了一系列的参数，方便大家修改边框尺寸、方块大小、以及各部分的颜色等。其中边框尺寸和方块宽度均以像素点为单位。当方块的宽度确定时，如果我们知道方块左上方顶点的坐标，就能轻而易举的画出整个方块区域。因此，我们将方块的移动简化为其左上角顶点的移动。

由于像素时钟相对于方块移动速度而言过快，我们通过计数器对时钟计数，得到一个频率为 100hz 的脉冲信号 move_en，用它作为使能信号来控制方块的移动（33~45 行）。方块的移动方向分为水平方向 h_direct 和竖直方向 v_direct，由代码的 47 至 70 行可知，当方块移动到上下边框时，竖直移动方向改变；当方块移动到左右边框时，水平移动方向改变。代码 72 至 93 行根据方块的移动方向，在使能信号 move_en 到来时改变其左上角顶点的纵横坐标值。当 move_en 的频率为 100hz 时，方块每秒钟在水平和竖直方向上分别移动 100 个像素点的距离，也可以通过调整 move_en 的频率，来加快或减慢方块移动的速度。

代码第 95 至 110 行根据 RGB 驱动模块输出的纵横坐标判断当前像素点所在的区域，对不同区域中的像素点赋以不同的颜色值，从而实现边框、方块以及背景颜色的绘制。

18.5 下载验证

编译工程并生成比特流.bit 文件。然后将 HDMI 电缆的一端与启明星开发板上 HDMI 插座相连接，另一端与显示器的 HDMI 接口相连接，如下图所示。

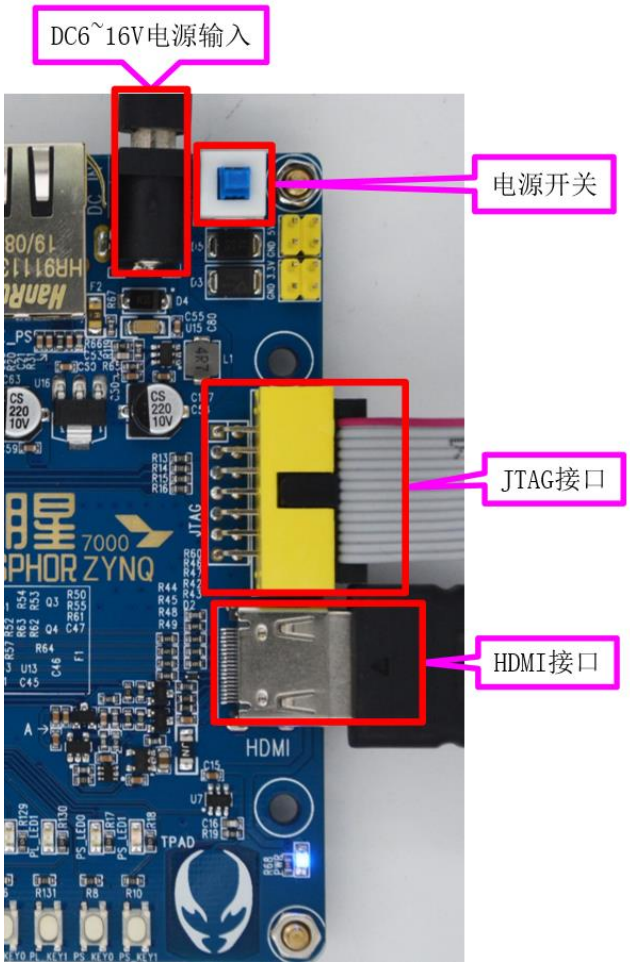


图 18.5.1 启明星 HDMI 连接图

最后将下载器一端连电脑，另一端与开发板上的 JTAG 端口连接，连接电源线并打开电源开关。

接下来我们下载比特流.bit 文件，验证 HDMI 显示方块移动的功能。下载完成后观察显示器显示的图案

如下图所示，图中的黑色方块能够不停的移动，且碰撞到蓝色边框时能改变移动方向，说明 HDMI 方块移动程序下载验证成功。

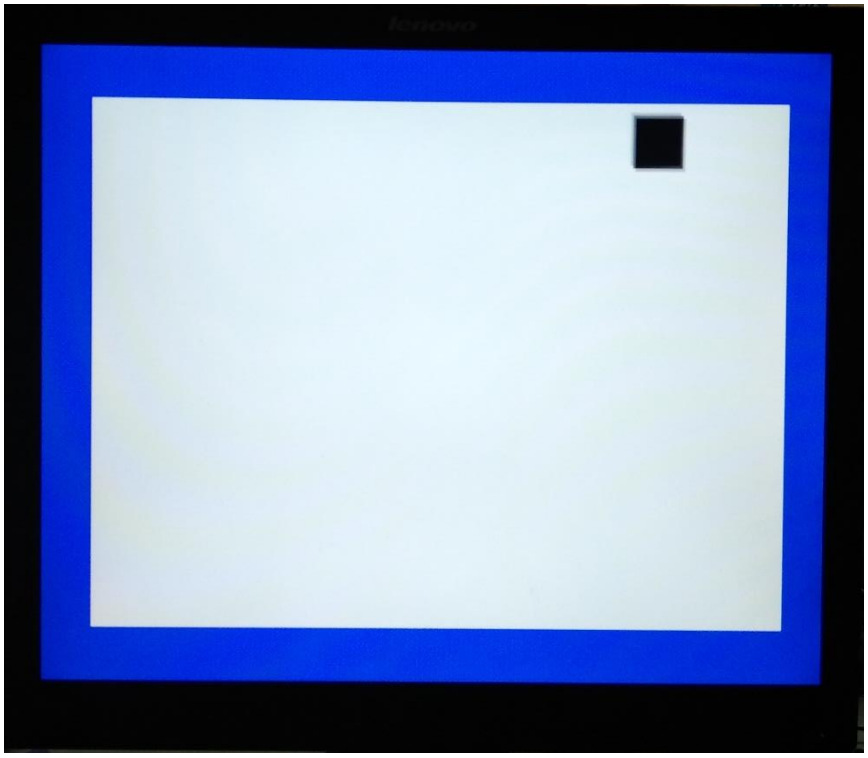


图 18.5.2 HDMI 方块移动效果图

第十九章 EEPROM 读写测试实验

EEPROM 是一种用于计算机系统的非易失性存储器，也常在嵌入式领域中作为数据的存储设备，在物联网及可穿戴设备等需要存储少量数据的场景中也有广泛应用。本章我们学习 EEPROM 的读写操作并进行 EEPROM 读写实验。

本章包括以下几个部分：

19.1 EEPROM 简介

19.2 实验任务

19.3 硬件设计

19.4 程序设计

19.5 下载验证

19.1 EEPROM 简介

EEPROM (Electrically Erasable Progammable Read Only Memory，E2PROM)即电可擦除可编程只读存储器，是一种常用的非易失性存储器（掉电数据不丢失），EEPROM 有多种类型的产品，我们启明星 ZYNQ 开发板上使用的是 ATMEL 公司生产的 AT24C 系列的 AT24C64 这一型号。AT24C64 具有高可靠性，可对所存数据保存 100 年，并可多次擦写，擦写次数达一百万次。

一般而言，对于存储类型的芯片，我们比较关注其存储容量。我们这次实验所用的 AT24C64 存储容量为 64Kbit，内部分成 256 页，每页 32 字节，共有 8192 个字节，且其读写操作都是以字节为基本单位。可以把 AT24C64 看作一本书，那么这本书有 256 页，每页有 32 行，每行有 8 个字，总共有 256*32*8=65536 个字，对应着 AT24C64 的 64*1024=65536 个 bit。

知道了 AT24C64 的存储容量，就知道了读写的空间大小。那么我们该如何对 AT24C64 进行读写操作呢？

由于 AT24C64 采用两线串行接口的双向数据传输协议——I2C 协议实现读写操作，所以我们有必要了解一下 I2C 协议。

I2C 即 Inter-Integrated Circuit(集成电路总线），是由 Philips 半导体公司（现在的 NXP 半导体公司）在八十年代初设计出来的一种简单、双向、二线制总线标准。多用于主机和从机在数据量不大且传输距离短的场合下的主从通信。主机启动总线，并产生时钟用于传送数据，此时任何接收数据的器件均被认为是从机。

I2C 总线由数据线 SDA 和时钟线 SCL 构成通信线路，既可用于发送数据，也可接收数据。在主控与被控 IC 之间可进行双向数据传送，数据的传输速率在标准模式下可达 100kbit/s，在快速模式下可达 400kbit/s，在高速模式下可达 3.4Mbit/s，各种被控器件均并联在总线上，通过器件地址（SLAVE ADDR，具体可查器件手册）识别。我们启明星 I2C 总线物理拓扑结构如下图所示。

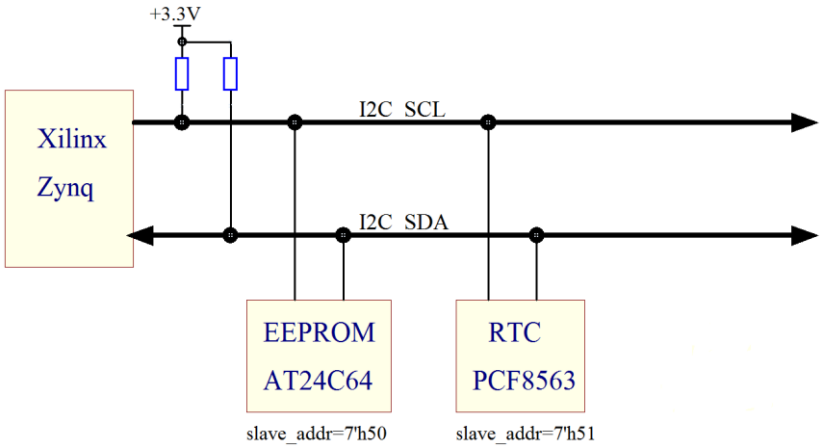


图 19.1.1 启明星 I2C 总线物理拓扑结构图

图中的 I2C_SCL 是串行时钟线，I2C_SDA 是串行数据线，由于 I2C 器件一般采用开漏结构与总线相连，所以 I2C_SCL 和 I2C_SDA 均需接上拉电阻，也正因此，当总线空闲时，这两条线路都处于高电平状态，当连到总线上的任一器件输出低电平，都将使总线拉低，即各器件的 SDA 及 SCL 都是“线与”关系。

I2C 总线支持多主和主从两种工作方式，通常工作在主从工作方式，我们的开发板就采用主从工作方式。在主从工作方式中，系统中只有一个主机，其它器件都是具有 I2C 总线的外围从机。在主从工作方式中，主机启动数据的发送（发出启动信号）并产生时钟信号，数据发送完成后，发出停止信号。

I2C 总线结构虽然简单，使用两线传输，然而要实现器件间的通信，需要通过控制 SCL 和 SDA 的时序，使其满足 I2C 的总线传输协议，方可实现器件间的数据传输。那么 I2C 协议的时序是怎样的呢？

在 I2C 器件开始通信（传输数据）之前，串行时钟线 SCL 和串行数据线 SDA 线由于上拉的原因处于高

电平状态，此时 I2C 总线处于空闲状态。如果主机（此处指 FPGA）想开始传输数据，只需在 SCL 为高电平时将 SDA 线拉低，产生一个起始信号，从机检测到起始信号后，准备接收数据，当数据传输完成，主机只需产生一个停止信号，告诉从机数据传输结束，停止信号的产生是在 SCL 为高电平时，SDA 从低电平跳变到高电平，从机检测到停止信号后，停止接收数据。I2C 整体时序如下图。起始信号之前为空闲状态，起始信号之后到停止信号之前的这一段为数据传输状态，主机可以向从机写数据，也可以读取从机输出的数据，数据的传输由双向数据线（SDA）完成。停止信号产生后，总线再次处于空闲状态。

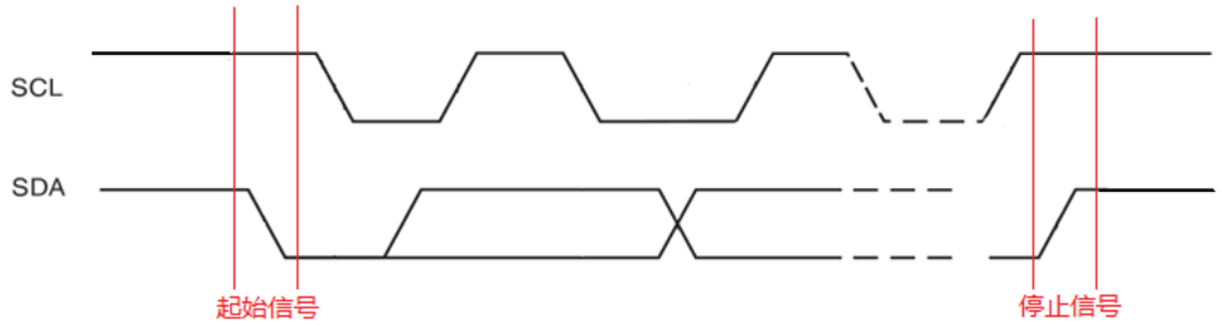


图 19.1.2 I2C 整体时序图

了解到了整体时序之后，我们可能有疑问，数据是以什么样的格式传输的呢？满足怎样的时序要求呢？是在任何时候改变都可以吗？怎么知道从机有没有接收到数据呢？带着这些疑问，我们继续学习 I2C。

由于只有一根数据线进行数据的传输，如果不规定好传输规则肯定会导致信息错乱，如同在单条道路上驾驶，没有交通规则，再好的道路也会发生拥堵甚至更糟。采用两线结构的 I2C 虽然只有一根数据线，但由于还有一条时钟线，可以让数据线在时钟线的带领下有顺序的传送，就好像单条道路上的车辆在交警或信号指示灯的指示下有规则的通行。那么 I2C 遵循怎样的规则呢？

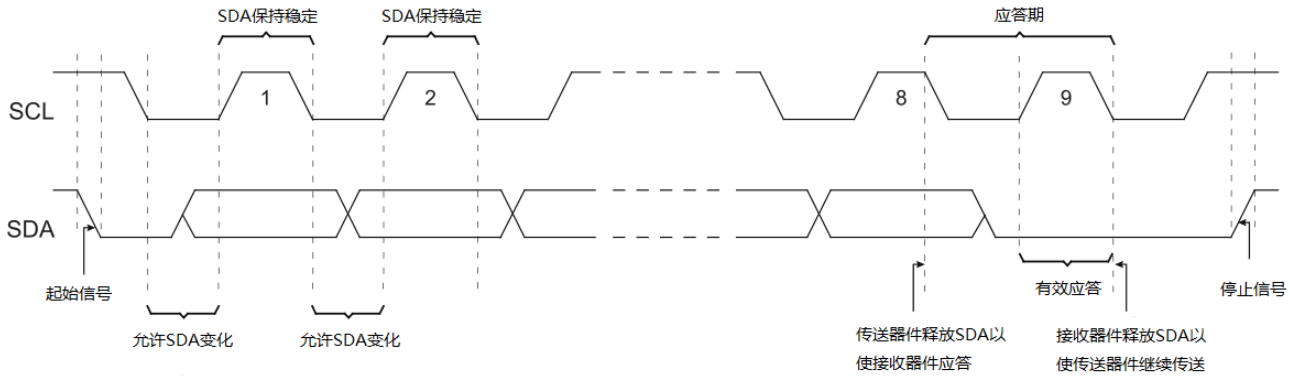


图 19.1.3 I2C 具体时序图

如果要想回答这些问题，我们得读懂图 19.1.3。由图 19.1.3 可知，我们在起始信号之后，主机开始发送传输的数据；在串行时钟线 SCL 为低电平状态时，SDA 允许改变传输的数据位（1 为高电平，0 为低电平），在 SCL 为高电平状态时，SDA 要求保持稳定，相当于一个时钟周期传输 1bit 数据，经过 8 个时钟周期后，传输了 8bit 数据，即一个字节。第 8 个时钟周期末，主机释放 SDA 以使从机应答，在第 9 个时钟周期，从机将 SDA 拉低以应答；如果第 9 个时钟周期，SCL 为高电平时，SDA 未被检测到为低电平，视为非应答，表明此次数据传输失败。第 9 个时钟周期末，从机释放 SDA 以使主机继续传输数据，如果主机发送停止信号，此次传输结束。我们要注意的是数据以 8bit 即一个字节为单位串行发出，其最先发送的是字节的最高位。

I2C 的时序部分已经基本介绍完了，但还有一个小问题，就是当多个 I2C 器件挂接在总线上时，怎样才

能与我们想要传输数据的器件进行通信。这就涉及到了器件地址（也称从机地址，SLAVE ADDRESS）。

每个 I2C 器件都有一个器件地址，有些 I2C 器件的器件地址是固定的，而有些 I2C 器件的器件地址由一个固定部分和一个可编程的部分构成，这是因为很可能在一个系统中有几个同样的器件，器件地址的可编程部分能最大数量的使这些器件连接到 I2C 总线上，例如 EEPROM 器件，为了增加系统的 EEPROM 容量，可能需要多个 EEPROM。器件可编程地址位的数量由它可使用的管脚决定，比如 EEPROM 器件一般会留下 3 个管脚用于可编程地址位。但有些 I2C 器件在出厂时器件地址就设置好了，用户不可以更改（如实时时钟 PCF8563 的器件地址为固定的 7'h51）。所以当主机想给某个器件发送数据时，只需向总线上发送接收器件的器件地址即可。

对于 AT24C64 而言，其器件地址为 1010 加 3 位的可编程地址，3 位可编程地址由器件上的 3 个管脚 A2、A1、A0（见图 19.3.2）的硬件连接决定。当硬件电路上分别将这三个管脚连接到 GND 或 VCC 时，就可以设置不同的可编程地址。对于我们的开发板，这 3 个管脚连接到地。

进行数据传输时，主机首先向总线上发出开始信号，对应开始位 S，然后按照从高到低的位序发送器件地址，一般为 7bit，第 8bit 位为读写控制位 R/W，该位为 0 时表示主机对从机进行写操作，当该位为 1 时表示主机对从机进行读操作，然后接收从机响应。对于 AT24C64 来说，其传输器件地址格式如下图所示。

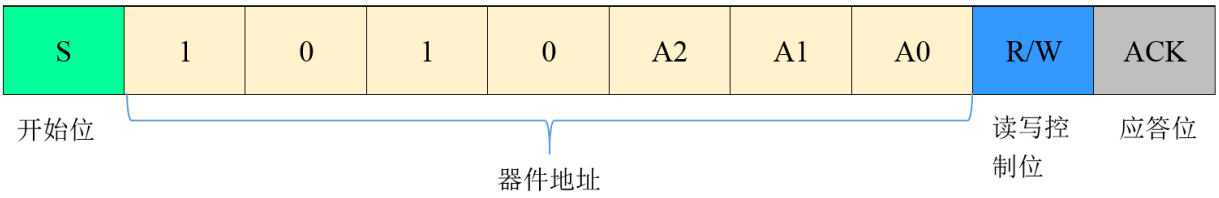


图 19.1.4 器件地址格式示意图

发送完第一个字节（7 位器件地址和一位读写控制位）并收到从机正确的应答后就开始发送字地址（Word Address）。一般而言，每个兼容 I2C 协议的器件，内部总会有可供读写的寄存器或存储器，对于我们本次实验用到的 EEPROM 存储器，内部就是一系列顺序编址的存储单元。所以，当我们对一个器件中的存储单元（包括寄存器）进行读写时，首先要指定存储单元的地址即字地址，然后再向该地址写入内容。该地址为一个或两个字节长度，具体长度由器件内部的存储单元的数量决定，当存储单元数量不超过一个字节所能表示的最大数量（2^8=256）时，用一个字节表示，超过一个字节所能表示的最大数量时，就需要用两个字节来表示，例如同是 EEPROM 存储器，AT24C02 的存储单元容量为 2Kbit=256Byte（一般 bit 缩写为 b，Byte 缩写为 B），用一个字节地址即可寻址所有的存储单元，而 AT24C64 的存储单元容量为 64Kb=8KB，需要 13 位（2^13=8KB）的地址位，而 I2C 又是以字节为单位进行传输的，所以需要用两个字节地址来寻址整个存储单元。图 19.1.5 和图 19.1.6 分别为单字节字地址和双字节字地址器件的地址分布图，其中单字节字地址的器件是以存储容量为 2Kb 的 EEPROM 存储器 AT24C02 为例，双字节字地址的器件是以存储容量为 64Kb 的 EEPROM 存储器 AT24C64 为例，WA7 即字地址 Word Address 的第 7 位，以此类推，用 WA 是为了区别前面器件地址中的 A。

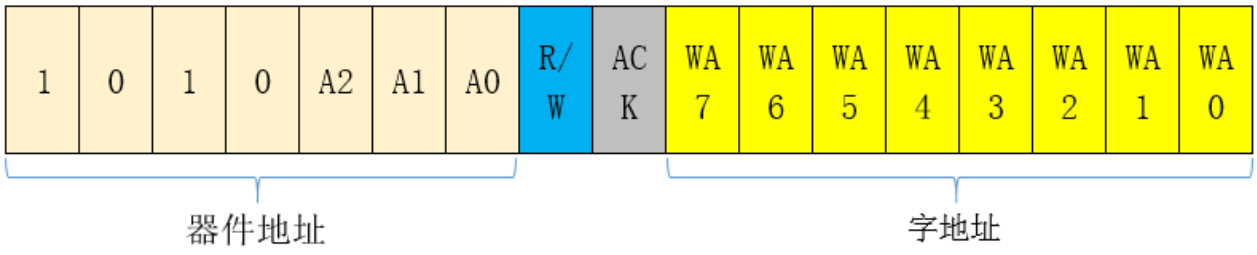


图 19.1.5 单字节字地址分布

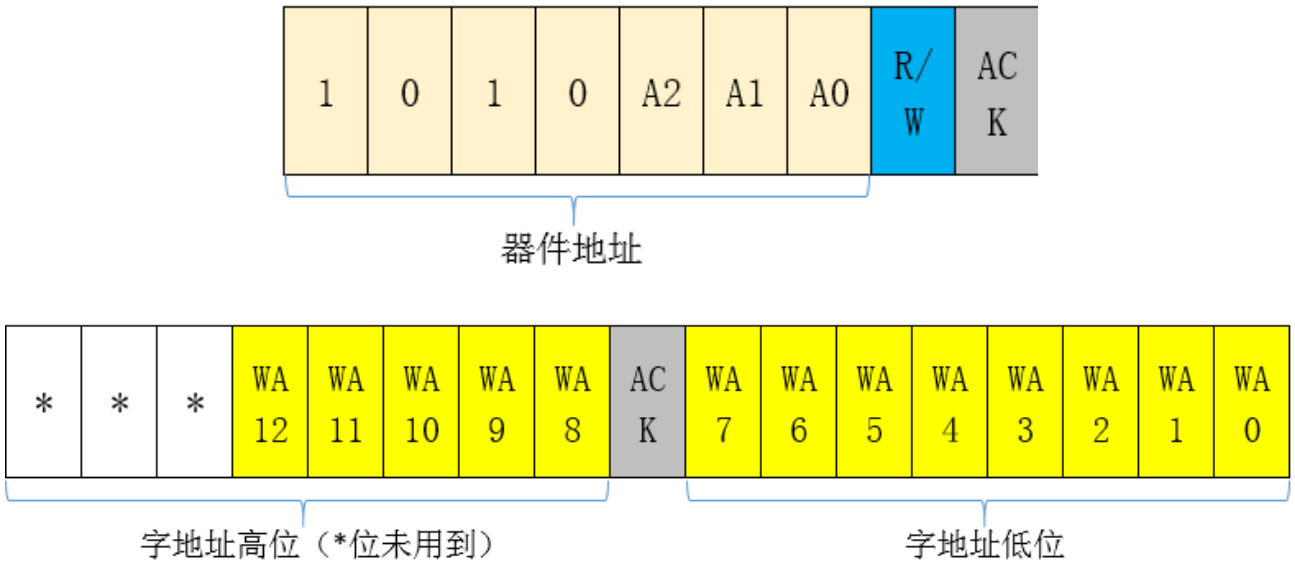


图 19.1.6 双字节字地址分布

主机发送完字地址，从机正确应答后就把内部的存储单元地址指针指向该单元。如果读写控制位 R/W 位为“0”即写命令，从机就处于接收数据的状态，此时，主机就开始写数据了。写数据分为单次写（对于 EEPROM 而言，称为字节写）和连续写（对于 EEPROM 而言，称为页写），那么这两者有什么区别呢？对比图 19.1.7 和图 19.1.8 可知，两者的区别在于发送完一字节数据后，是发送结束信号还是继续发送下一字节数据，如果发送的是结束信号，就称为单次写，如果继续发送下一字节数据，就称为连续写。图 19.1.7 是 AT24C64 的单次写（字节写）时序，对于字地址为单字节的 I2C 器件而言，在发送完字地址（对应图 19.1.7 的字地址高位），且从机应答后即可串行发送 8bit 数据。图 19.1.8 是 AT24C64 连续写（页写）时序。要注意的是，对于 AT24C64 的页写，是不能发送超过一页的单元容量的数据的，而 AT24C64 的一页的单元容量为 32Byte，当写完一页的最后一个单元时，地址指针指向该页的开头，如果再写入数据，就会覆盖该页的起始数据。

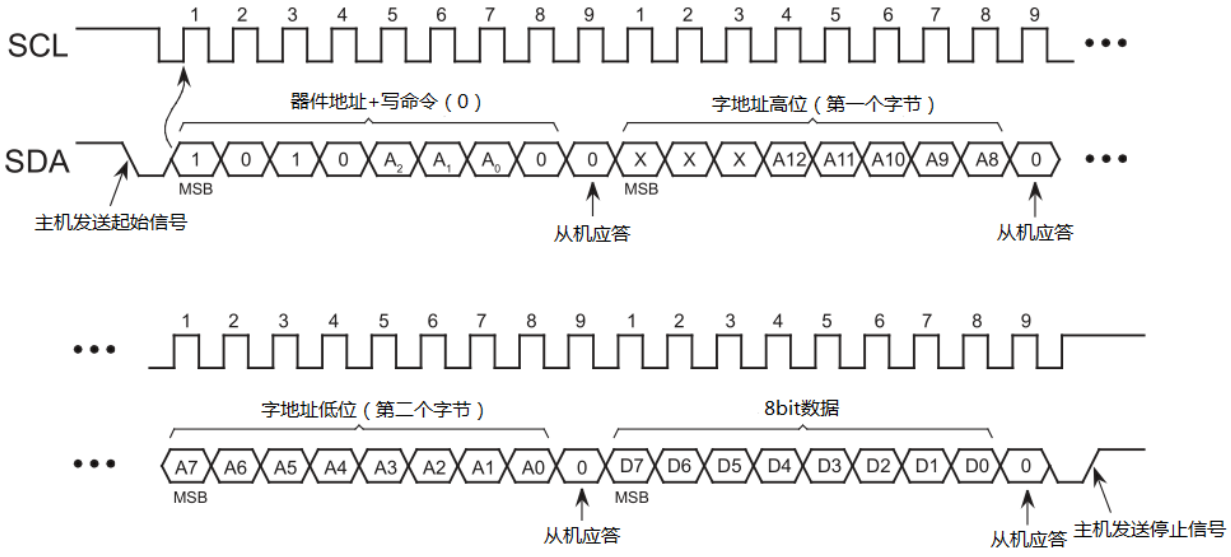


图 19.1.7 单次写（字节写）时序

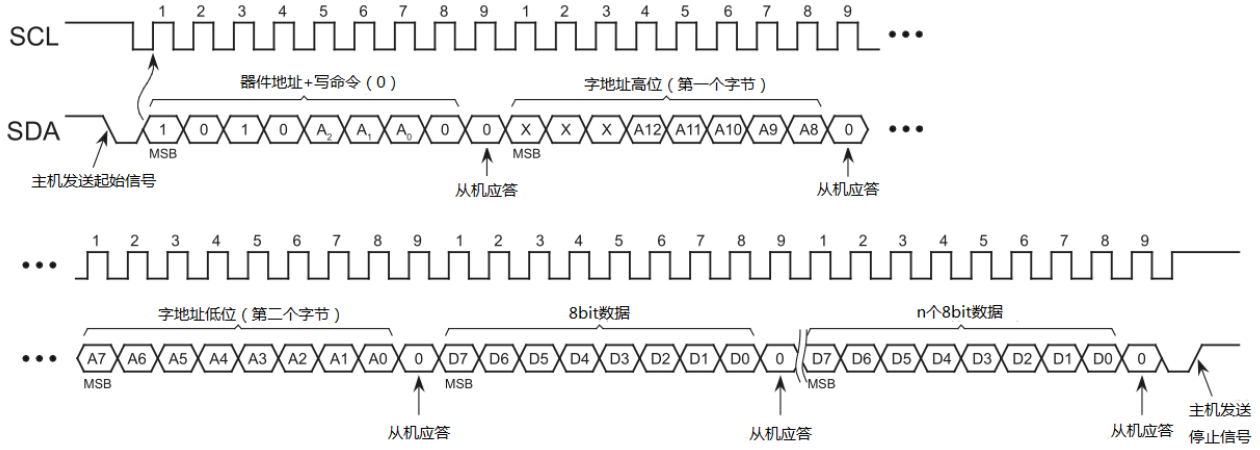


图 19.1.8 连续写（页写）时序

如果读写控制位 R/W 位为“1”即读命令，主机就处于接收数据的状态，从机从该地址单元输出数据。读数据有三种方式：当前地址读、随机读和连续读。当前地址读是指在一次读或写操作后发起读操作。由于 I2C 器件在读写操作后，其内部的地址指针自动加一，因此当前地址读可以读取下一个字地址的数据。也就是说上次读或写操作的单元地址为 02 时，当前地址读的内容就是地址 03 处的单元数据，时序图如图 19.1.9 所示。

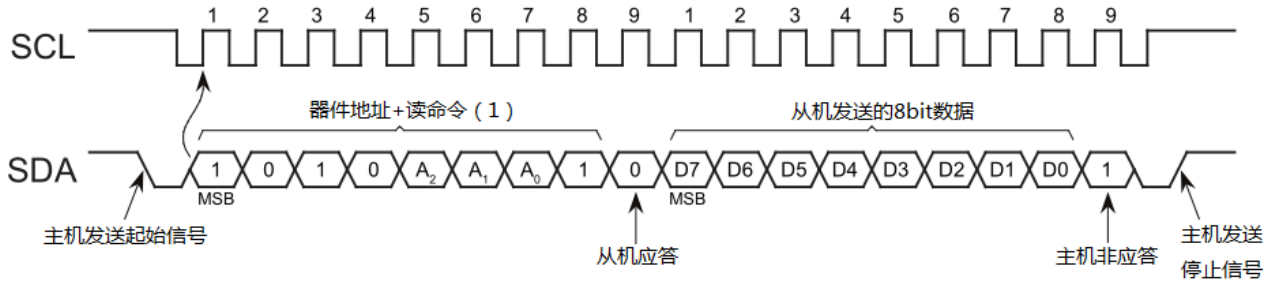


图 19.1.9 当前地址读时序

由于当前地址读极不方便读取任意的地址单元的数据，所以就有了随机读，随机读的时序有点奇怪，见图 19.1.10，发送完器件地址和字地址后，竟然又发送起始信号和器件地址，而且第一次发送器件地址时后面的读写控制位为“0”，也就是写命令，第二次发送器件地址时后面的读写控制位为“1”，也就是读。为什么会有这样奇怪的操作呢？这是因为我们需要使从机内的存储单元地址指针指向我们想要读取的存储单元地址处，所以首先发送了一次 Dummy Write 也就是虚写操作，只所以称为虚写，是因为我们并不是真的要写数据，而是通过这种虚写操作使地址指针指向虚写操作中字地址的位置，等从机应答后，就可以以当前地址读的方式读数据了，如图 19.1.10 所示，随机地址读是没有发送数据的单次写操作和当前地址读操作的结合体。

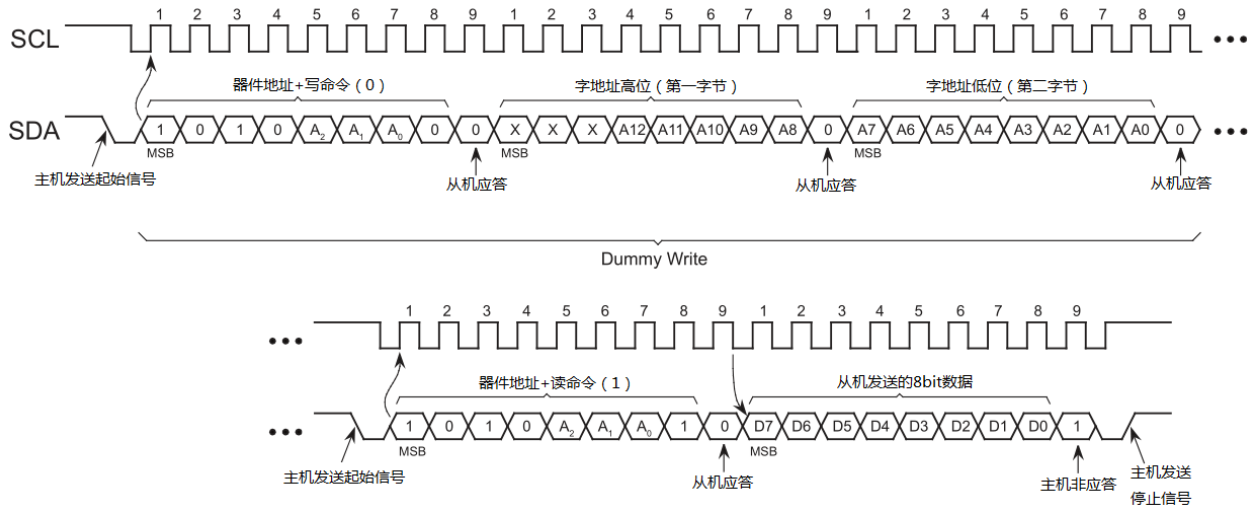


图 19.1.10 随机地址读时序

至于连续读，对应的是当前地址读和随机读都是一次读取一个字节而言的，它是将当前地址读或随机读的主机非应答改成应答，表示继续读取数据，图 19.1.11 是在当前地址读下的连续读。

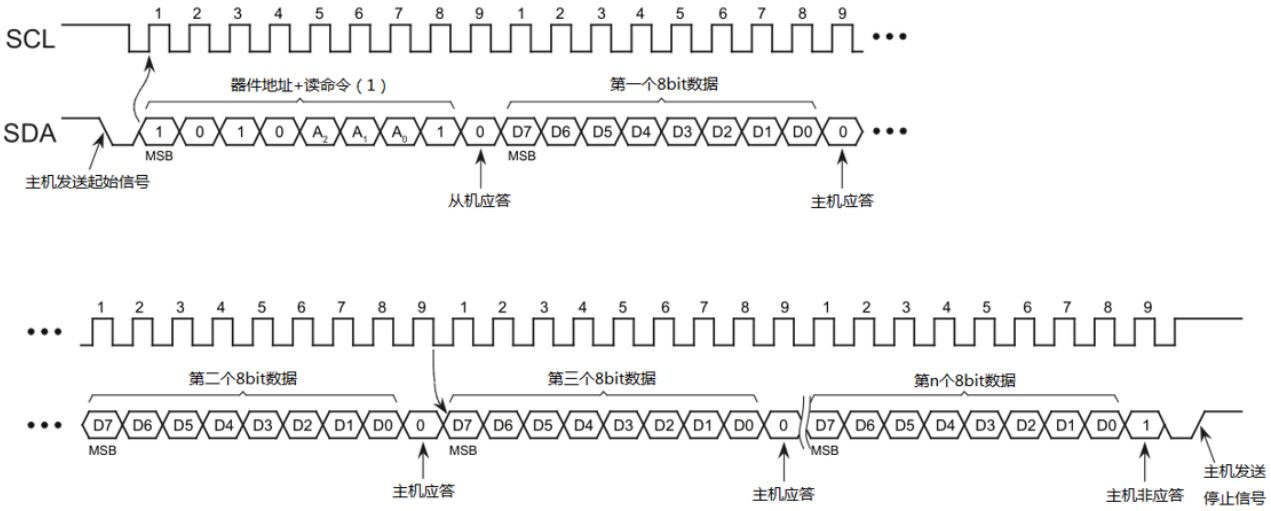


图 19.1.11 顺序读时序

至此，I2C 协议就基本讲完了，本章我们主要采用单次写和随机读的方式进行 EEPROM 读写测试。

19.2 实验任务

本节的实验任务是先向 EEPROM（AT24C64）的存储器地址 0 至 255 分别写入数据 0~255；写完之后再读取存储器地址 0~ 255 中的数据，若读取的值全部正确则 LED 灯常亮，否则 LED 灯闪烁。

19.3 硬件设计

AT24C64 芯片的常用封装形式有直插（DIP8）式和贴片（SO-8）式两种，无论是直插式还是贴片式，其引脚功能与序号都一样，我们开发板上采用的是贴片式，实物图和引脚图分别如图 19.3.1 和图 19.3.2 所示。

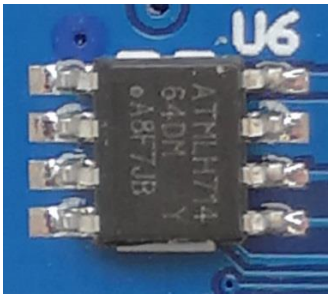


图 19.3.1 开发板上的 AT24C64 实物图

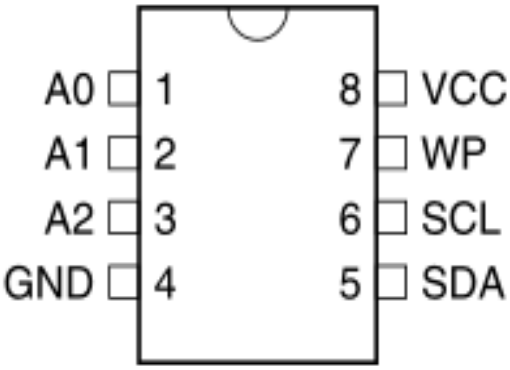


图 19.3.2 AT24C64 引脚图

AT24C64 的引脚功能如下：

A2,A1,A0：可编程地址输入端。

GND：电源地引脚

SDA：SDA（Serial Data，串行数据）是双向串行数据输入/输出端。

SCL：SCL（Serial clock，串行时钟）串行时钟输入端。

WP（写保护）：AT24C64 有一个写保护引脚用于提供数据保护，当写保护引脚连接至 GND 时，芯片可以正常写，当写保护引脚连接至 VCC 时，使能写保护功能，此时禁止向芯片写入数据，只能进行读操作。

VCC：电源输入引脚

我们的启明星 Zynq 开发板上 EEPROM 的原理图如图 19.3.3 所示：

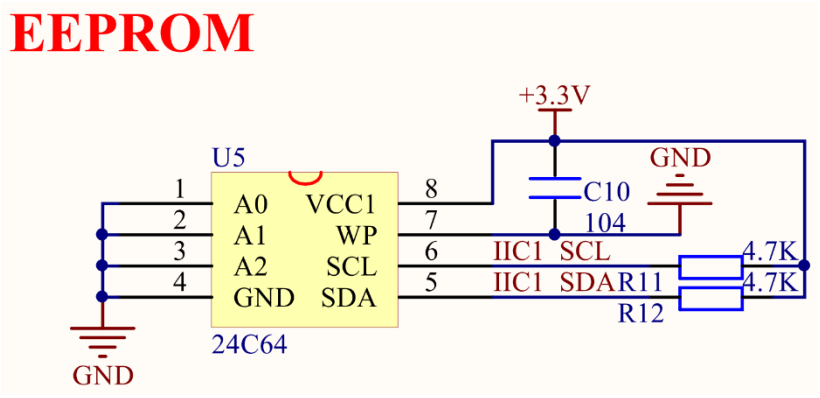


图 19.3.3 EEPROM 原理图

由上图可知，我们开发板上的 EEPROM 可编程地址 A2、A1、A0 连接到地，所以 AT24C64 的器件地址为 1010000，如下图所示：

1	0	1	0	0	0	0

图 19.3.4 AT24C64 的器件地址

本实验中，系统时钟、按键复位以及 EEPROM 的 SCL 和 SDA 的管脚分配如下表所示：

表 19.3.1 EEPROM读写测试实验管脚分配

信号名	方向	管脚	端口说明	电平标准
sys_clk	input	U18	系统时钟，50M	LVCMOS33
sys_rst_n	input	J15	系统复位，低有效,位于底板上	LVCMOS33
iic_scl	output	M17	eeprom的时钟线	LVCMOS33
iic_sda	inout	M18	eeprom的数据线	LVCMOS33
led	output	J18	led灯	LVCMOS33

对应的 XDC 约束语句如下所示：

```
set_property -dict {PACKAGE_PIN U18 IOSTANDARD LVCMOS33} [get_ports sys_clk]
set_property -dict {PACKAGE_PIN J15 IOSTANDARD LVCMOS33} [get_ports sys_rst_n]
set_property -dict {PACKAGE_PIN M17 IOSTANDARD LVCMOS33} [get_ports iic_scl]
set_property -dict {PACKAGE_PIN M18 IOSTANDARD LVCMOS33} [get_ports iic_sda]
set_property -dict {PACKAGE_PIN J18 IOSTANDARD LVCMOS33} [get_ports led]
```

19.4 程序设计

根据实验任务，我们可以大致规划出系统的控制流程：首先 FPGA 向 EEPROM 写数据，写完之后从 EEPROM 读出所写入的数据，并判断读出的数据与写入的数据是否相同，如果相同则 LED 灯常亮，否则 LED 灯闪烁。由此画出系统的功能框图如下图所示：

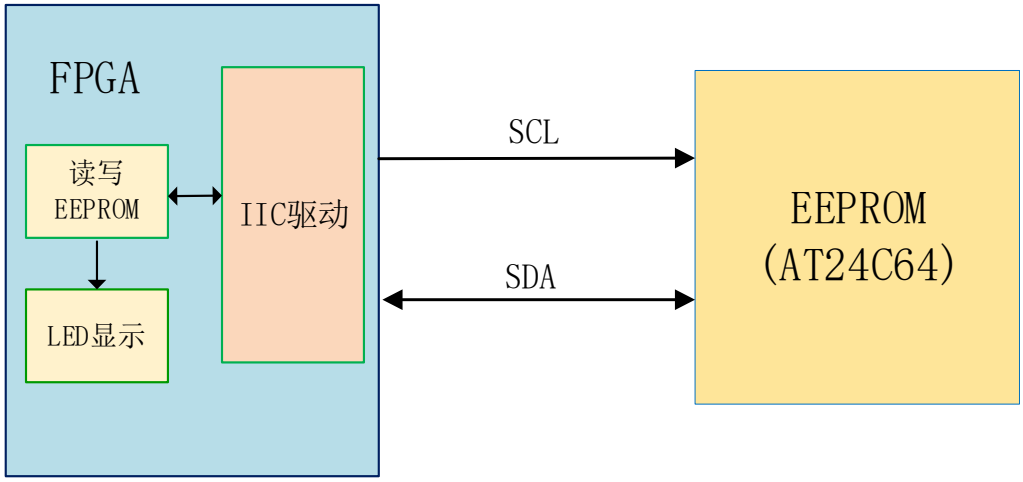


图 19.4.1 EEPROM 读写实验系统框图

由系统总体框图可知，FPGA 部分包括四个模块，顶层模块（e2prom_top）、读写模块（e2prom_rw）、I2C 驱动模块（i2c_dri）和 LED 灯显示模块（led_alarm）。其中在顶层模块中完成对其余模块的例化。

各模块端口及信号连接如下图所示：

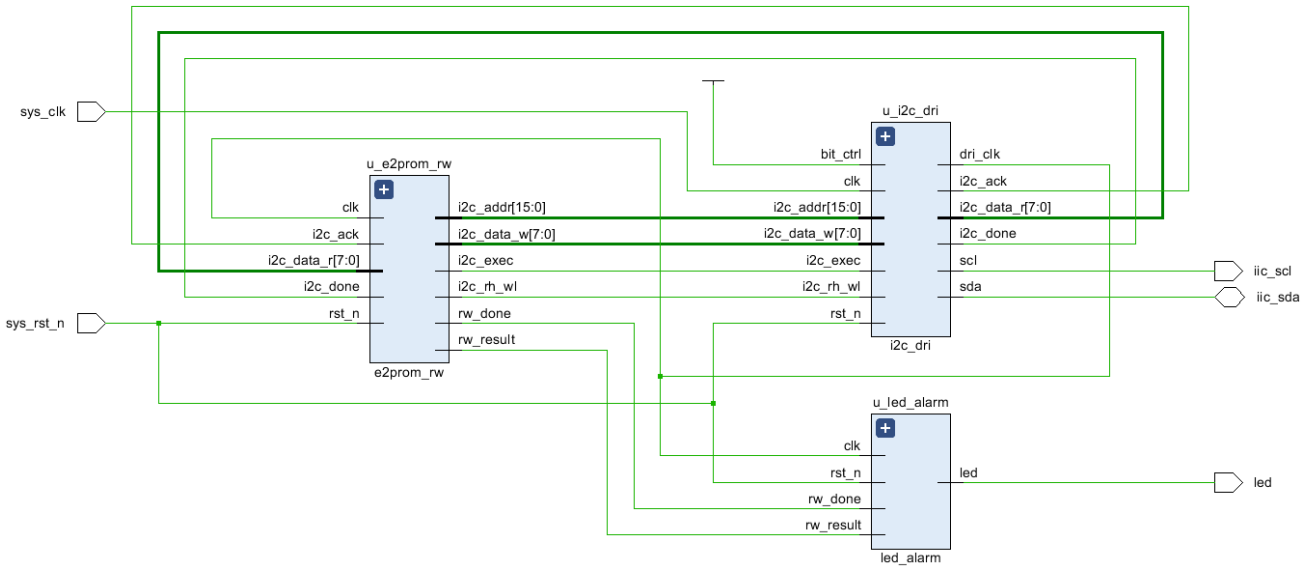


图 19.4.2 顶层模块原理图

i2c_dri 为 I2C 驱动模块，用来驱动 I2C 的读写操作。当 FPGA 通过 EEPROM 读写模块 e2prom_rw 向 EEPROM 读写数据时，拉高 i2c 触发控制信号 i2c_exec 以使能 I2C 驱动模块，并使用读写控制信号 i2c_rh_wl 控制读写操作，当 i2c_rh_wl 为低电平时，I2C 驱动模块 i2c_dri 执行写操作，当 i2c_rh_wl 为高电平时，I2C 驱动模块 i2c_dri 执行读操作。此外，e2prom_rw 模块通过 i2c_addr 接口向 i2c_dri 模块输入器件字地址，通过 i2c_data_w 接口向 i2c_dri 模块输入写的数据，并通过 i2c_data_r 接口读取 i2c_dri 模块读到的数据。rw_done 信号是读写测试完成的标志，rw_result 是读写测试的结果。

顶层模块的代码如下：

```
1  module e2prom_top(
2      input               sys_clk    ,       //系统时钟
3      input               sys_rst_n  ,       //系统复位
4      //eeprom interface
5      output              iic_scl    ,       //eeprom 的时钟线 scl
6      inout               iic_sda    ,       //eeprom 的数据线 sda
7      //user interface
8      output              led                //led 显示
9  );
10
11 //parameter define
12 parameter    SLAVE_ADDR = 7'b1010000    ; //器件地址(SLAVE_ADDR)
13 parameter    BIT_CTRL   = 1'b1          ; //字地址位控制参数(16b/8b)
14 parameter    CLK_FREQ   = 26'd50_000_000; //i2c_dri 模块的驱动时钟频率(CLK_FREQ)
15 parameter    I2C_FREQ   = 18'd250_000   ; //I2C 的 SCL 时钟频率
16 parameter    L_TIME     = 17'd125_000   ; //led 闪烁时间参数
17
18 //wire define
19 wire           dri_clk    ; //I2C 操作时钟
```

```
wire           i2c_exec  ; //I2C 触发控制
wire   [15:0]  i2c_addr  ; //I2C 操作地址
wire   [ 7:0]  i2c_data_w; //I2C 写入的数据
wire           i2c_done  ; //I2C 操作结束标志
wire           i2c_ack   ; //I2C 应答标志
wire           i2c_rh_wl ; //I2C 读写控制
wire   [ 7:0]  i2c_data_r; //I2C 读出的数据
wire           rw_done   ; //E2PROM 读写测试完成
wire           rw_result ; //E2PROM 读写测试结果 0:失败 1:成功

//*****************************************************
//**                    main code
//*****************************************************

//e2prom 读写测试模块
e2prom_rw u_e2prom_rw(
    .clk         (dri_clk   ),  //时钟信号
    .rst_n       (sys_rst_n ),  //复位信号
    //i2c interface
    .i2c_exec    (i2c_exec  ),  //I2C 触发执行信号
    .i2c_rh_wl   (i2c_rh_wl ),  //I2C 读写控制信号
    .i2c_addr    (i2c_addr  ),  //I2C 器件内地址
    .i2c_data_w  (i2c_data_w),  //I2C 要写的数据
    .i2c_data_r  (i2c_data_r),  //I2C 读出的数据
    .i2c_done    (i2c_done  ),  //I2C 一次操作完成
    .i2c_ack     (i2c_ack   ),  //I2C 应答标志 0:应答 1:未应答
    //user interface
    .rw_done     (rw_done   ),  //E2PROM 读写测试完成
    .rw_result   (rw_result )   //E2PROM 读写测试结果 0:失败 1:成功
);

//i2c 驱动模块
i2c_dri #(
    .SLAVE_ADDR  (SLAVE_ADDR),  //EEPROM 从机地址
    .CLK_FREQ    (CLK_FREQ  ),  //模块输入的时钟频率
    .I2C_FREQ    (I2C_FREQ  )   //IIC_SCL 的时钟频率
) u_i2c_dri(
    .clk         (sys_clk   ),
    .rst_n       (sys_rst_n ),
    //i2c interface
    .i2c_exec    (i2c_exec  ),  //I2C 触发执行信号
```

```
61      .bit_ctrl     (BIT_CTRL  ),  //器件地址位控制(16b/8b)
62      .i2c_rh_wl    (i2c_rh_wl ),  //I2C 读写控制信号
63      .i2c_addr     (i2c_addr  ),  //I2C 器件内地址
64      .i2c_data_w   (i2c_data_w),  //I2C 要写的数据
65      .i2c_data_r   (i2c_data_r),  //I2C 读出的数据
66      .i2c_done     (i2c_done  ),  //I2C 一次操作完成
67      .i2c_ack      (i2c_ack   ),  //I2C 应答标志
68      .scl          (iic_scl   ),  //I2C 的 SCL 时钟信号
69      .sda          (iic_sda   ),  //I2C 的 SDA 信号
70      //user interface
71      .dri_clk      (dri_clk   )   //I2C 操作时钟
72  );
73
74  //led 指示模块
75  led_alarm #(.L_TIME(L_TIME  )    //控制 led 闪烁时间
76  ) u_led_alarm(
77      .clk          (dri_clk   ),
78      .rst_n        (sys_rst_n ),
79
80      .led          (led       ),
81      .rw_done      (rw_done   ),
82      .rw_result    (rw_result )
83  );
84
85  endmodule
```

顶层模块中主要完成对其余模块的例化，需要注意的是程序第 11 行到第 16 行定义了五个参数，在模块例化时会将这些变量传递到相应的模块。

SLAVE_ADDR 定义了 EEPROM 的器件地址；字地址位控制参数(16b/8b)BIT_CTRL 是用来控制不同字地址的 I2C 器件读写时序中字地址的位数，当 I2C 器件的字地址为 16 位时，参数 BIT_CTRL 设置为“1”，当 I2C 器件的字地址为 8 位时，参数 BIT_CTRL 设置为“0”；i2c_dri 模块的驱动时钟频率 CLK_FREQ 是指在例化 I2C 驱动模块 i2c_dri 时，驱动 i2c_dri 模块的时钟频率；I2C 的 SCL 时钟频率参数 I2C_FREQ 是用来控制 I2C 协议中的 SCL 的频率，一般不超过 400KHz；led 闪烁时间参数 L_TIME 用来控制 led 的闪烁间隔时间，参数值与驱动该模块的 clk 时钟频率有关。例如，控制 led 闪烁的间隔时间为 0.25s，clk 的频率为 1MHz 时，0.25s/1us=250000，由于代码中当计数器计数到 L_TIME 的值时，LED 的状态改变一次，LED 高电平加上低电平的时间才是一次闪烁的时间，所以 L_TIME 的值应定义成 125000。

由前面的 I2C 读写操作时序图我们可以发现，I2C 驱动模块非常适合采用状态机来编写。无论是字节写还是随机读，都要先从空闲状态开始，先发送起始信号，然后发送器件地址和读写命令（这里为了方便我们使用“控制命令”来表示器件地址和读写命令）。发送完控制命令并接收应答信号后发送字地址，然后就可以进行读写数据的传输了。读写数据传输结束后接收应答信号，最后发送停止信号，此时 I2C 读写操作结束，再次进入空闲状态。

状态机的状态跳转图如图 19.4.3 所示，总共有 8 个状态，一开始状态机处于空闲状态 st_idle，当 I2C

触发执行信号触发（i2c_exec=1）时，状态机进入发送控制命令状态 st_sladdr；发送完控制命令后就发送字地址，这里出于简单考虑，不对从机 EEPROM 的应答信号进行判断。由于字地址存在单字节和双字节的区别，我们通过 bit_ctrl 信号判断是单字节还是双字节字地址。对于双字节的字地址我们先发送高 8 位即第一个字节，发送完高 8 位后进入发送 8 位字地址状态 st _addr8，也就是发送双字节地地址的低 8 位；对于单字节的字地址我们直接进入发送 8 位字地址状态 st_addr8。发送完字地址后，根据读写判断标志来判断是读操作还是写操作。如果是写（wr_flag=0）就进入写数据状态 st_data_wr，开始向 EEPROM 发送数据；如果是读（wr_flag=1）就进入发送器件地址读状态 st_addr_rd 发送器件地址，此状态结束后就进入读数据状态 st_data_rd 接收 EEPROM 输出的数据。读或写数据结束后就进入结束 I2C 操作状态 st_done 并发送结束信号，此时，I2C 总线再次进入空闲状态 st_idle。

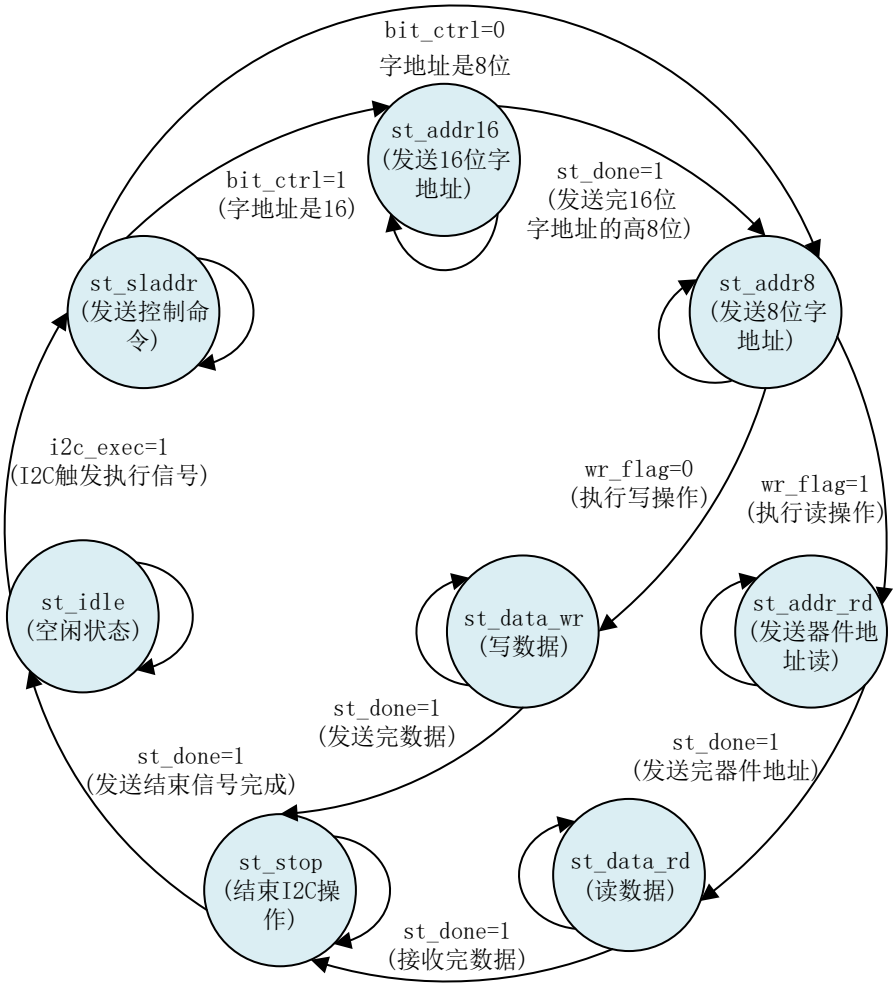


图 19.4.3 I2C 驱动模块状态跳转图

在程序中我们采用三段式状态机。由于代码较长，我们在这里将其中第二段的源代码粘贴如下：

```
108 //组合逻辑判断状态转移条件
109 always @(*) begin
110     next_state = st_idle;
111     case(cur_state)
112         st_idle: begin                          //空闲状态
113             if(i2c_exec) begin
```

```
114                 next_state = st_sladdr;
115             end
116             else
117                 next_state = st_idle;
118         end
119         st_sladdr: begin
120             if(st_done) begin
121                 if(bit_ctrl)                    //判断是 16 位还是 8 位字地址
122                     next_state = st_addr16;
123                 else
124                     next_state = st_addr8 ;
125             end
126             else
127                 next_state = st_sladdr;
128         end
129         st_addr16: begin                        //写 16 位字地址
130             if(st_done) begin
131                 next_state = st_addr8;
132             end
133             else begin
134                 next_state = st_addr16;
135             end
136         end
137         st_addr8: begin                         //8 位字地址
138             if(st_done) begin
139                 if(wr_flag==1'b0)               //读写判断
140                     next_state = st_data_wr;
141                 else
142                     next_state = st_addr_rd;
143             end
144             else begin
145                 next_state = st_addr8;
146             end
147         end
148         st_data_wr: begin                       //写数据(8 bit)
149             if(st_done)
150                 next_state = st_stop;
151             else
152                 next_state = st_data_wr;
153         end
154         st_addr_rd: begin                       //写地址以进行读数据
```

```
155             if(st_done) begin
156                 next_state = st_data_rd;
157             end
158             else begin
159                 next_state = st_addr_rd;
160             end
161         end
162         st_data_rd: begin                       //读取数据(8 bit)
163             if(st_done)
164                 next_state = st_stop;
165             else
166                 next_state = st_data_rd;
167         end
168         st_stop: begin                          //结束 I2C 操作
169             if(st_done)
170                 next_state = st_idle;
171             else
172                 next_state = st_stop ;
173         end
174         default: next_state= st_idle;
175     endcase
176 end
```

我们可以对照着图 19.4.3 来分析程序中各状态之间是如何跳转的。

EEPROM 读写模块主要实现对 I2C 读写过程的控制，包括给出字地址及需要写入该地址中的数据、启动 I2C 读写操作、判断读写数据是否一致等。

EEPROM 读写模块的代码如下：

```
1  module e2prom_rw(
2      input                clk        , //时钟信号
3      input                rst_n      , //复位信号
4
5      //i2c interface
6      output   reg         i2c_rh_wl  , //I2C 读写控制信号
7      output   reg         i2c_exec   , //I2C 触发执行信号
8      output   reg  [15:0] i2c_addr   , //I2C 器件内地址
9      output   reg  [ 7:0] i2c_data_w , //I2C 要写的数据
10     input         [ 7:0] i2c_data_r , //I2C 读出的数据
11     input                i2c_done   , //I2C 一次操作完成
12     input                i2c_ack    , //I2C 应答标志
13
14     //user interface
15     output   reg         rw_done    , //E2PROM 读写测试完成
```

```
16      output   reg          rw_result    //E2PROM 读写测试结果 0:失败 1:成功
17  );
18
19  //parameter define
20  //EEPROM 写数据需要添加间隔时间,读数据则不需要
21  parameter      WR_WAIT_TIME = 14'd5000; //写入间隔时间
22  parameter      MAX_BYTE     = 16'd256 ; //读写测试的字节个数
23
24  //reg define
25  reg   [1:0]    flow_cnt  ; //状态流控制
26  reg   [13:0]   wait_cnt  ; //延时计数器
27
28  //*****************************************************
29  //**                    main code
30  //*****************************************************
31
32  //EEPROM 读写测试,先写后读，并比较读出的值与写入的值是否一致
33  always @(posedge clk or negedge rst_n) begin
34      if(!rst_n) begin
35          flow_cnt <= 2'b0;
36          i2c_rh_wl <= 1'b0;
37          i2c_exec <= 1'b0;
38          i2c_addr <= 16'b0;
39          i2c_data_w <= 8'b0;
40          wait_cnt <= 14'b0;
41          rw_done <= 1'b0;
42          rw_result <= 1'b0;
43      end
44      else begin
45          i2c_exec <= 1'b0;
46          rw_done <= 1'b0;
47          case(flow_cnt)
48              2'd0 : begin
49                  wait_cnt <= wait_cnt + 1'b1;                //延时计数
50                  if(wait_cnt == WR_WAIT_TIME - 1'b1) begin   //EEPROM 写操作延时完成
51                      wait_cnt <= 1'b0;
52                      if(i2c_addr == MAX_BYTE) begin          //256 个字节写入完成
53                          i2c_addr <= 1'b0;
54                          i2c_rh_wl <= 1'b1;
55                          flow_cnt <= 2'd2;
56                      end
```

```
57                      else begin
58                          flow_cnt <= flow_cnt + 1'b1;
59                          i2c_exec <= 1'b1;
60                      end
61                  end
62              end
63              2'd1 : begin
64                  if(i2c_done == 1'b1) begin                  //EEPROM 单次写入完成
65                      flow_cnt <= 2'd0;
66                      i2c_addr <= i2c_addr + 1'b1;            //地址 0~255 分别写入.
67                      i2c_data_w <= i2c_data_w + 1'b1;        //.数据 0~255
68                  end
69              end
70              2'd2 : begin
71                  flow_cnt <= flow_cnt + 1'b1;
72                  i2c_exec <= 1'b1;
73              end
74              2'd3 : begin
75                  if(i2c_done == 1'b1) begin                  //EEPROM 单次读出完成
76                      //读出的值错误或者 I2C 未应答,读写测试失败
77                      if((i2c_addr[7:0] != i2c_data_r) || (i2c_ack == 1'b1)) begin
78                          rw_done <= 1'b1;
79                          rw_result <= 1'b0;
80                      end
81                      else if(i2c_addr == MAX_BYTE - 1'b1) begin //读写测试成功
82                          rw_done <= 1'b1;
83                          rw_result <= 1'b1;
84                      end
85                      else begin
86                          flow_cnt <= 2'd2;
87                          i2c_addr <= i2c_addr + 1'b1;
88                      end
89                  end
90              end
91              default : ;
92          endcase
93      end
94  end
95
96  endmodule
```

程序中第 21 行和第 22 行定义了两个参数，WR_WAIT_TIME（写入间隔时间）和 MAX_BYTE（读写

测试的字节个数）。AT24C64 官方手册对写入数据后，数据写入完成的时间最大不超过 10ms，所以为了保证数据能够正确写入，单次写入数据操作完成后，最好延时 10ms 的时间。本次实验为了节省数据写入的时间，WR_WAIT_TIME 的值设置为 5000，即 5ms（输入时钟的周期为 1us，1us*5000=5ms），实测延时 5ms 也可以正确写入。这里不建议大家将写入的间隔设置的过于短，否则会导致数据写入失败。另外，EEPROM 只有对写操作有时间间隔要求，对读操作没有间隔要求，因此读写测试模块仅对写操作增加时间间隔。

程序中第 32 至 94 行代码先对 I2C 驱动模块发起写操作，即拉高 i2c_exec，拉低 i2c_rh_wl（低电平表示写），然后分别向 EEPROM 的地址 0 至地址 255 写入数据 0 至 255，并且在每次写操作之间增加 5ms 的延时。数据全部写完后，发起读操作，即拉高 i2c_exec，拉高 i2c_rh_wl（高电平表示读），然后分别从 EEPROM 的地址 0 至地址 255 读出数据，并判断读出的值与写入的值是否一致，如果数据一致并且每次操作 IIC 都有应答信号产生（i2c_ack），EEPROM 的读写测试才正确，否则读写测试失败。

读写测试完成后，输出 rw_done 信号和 rw_result 信号，rw_done 为 EEPROM 读写测试完成信号，rw_result 为读写测试的结果，0 表示读写失败，1 表示读写正确。

EEPROM 写操作的 ILA 波形图如下图所示：

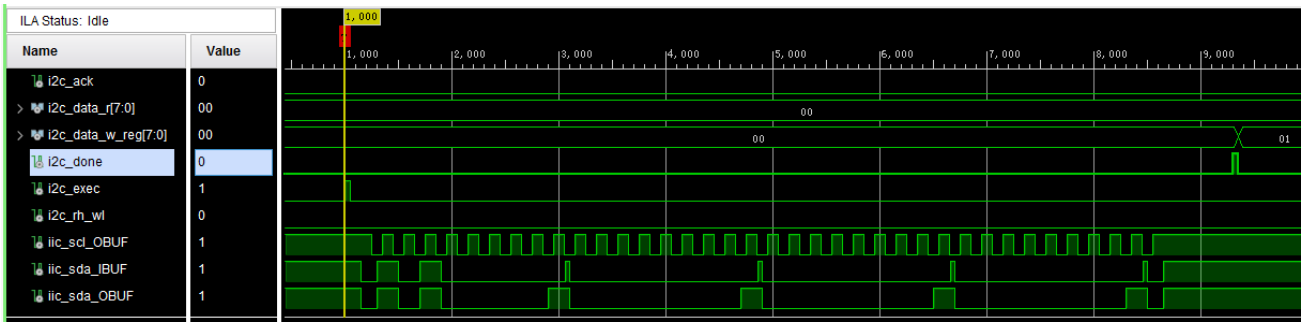


图 19.4.4 EEPROM 写操作 ILA 波形图

从该波形图中我们看到读写控制信号 i2c_rh_wl 为低电平，表示处于写操作状态。当 I2C 触发执行信号 i2c_exec 为高电平时开始执行 I2C 写操作。在 IIC 操作结束后，拉高 i2c_done 信号。另外，在 IIC 写操作期间，i2c_ack（IIC 应答标志）一直处于低电平，说明 EEPROM 响应了主机并应答。

EEPROM 读操作的 ILA 波形图如下图所示：

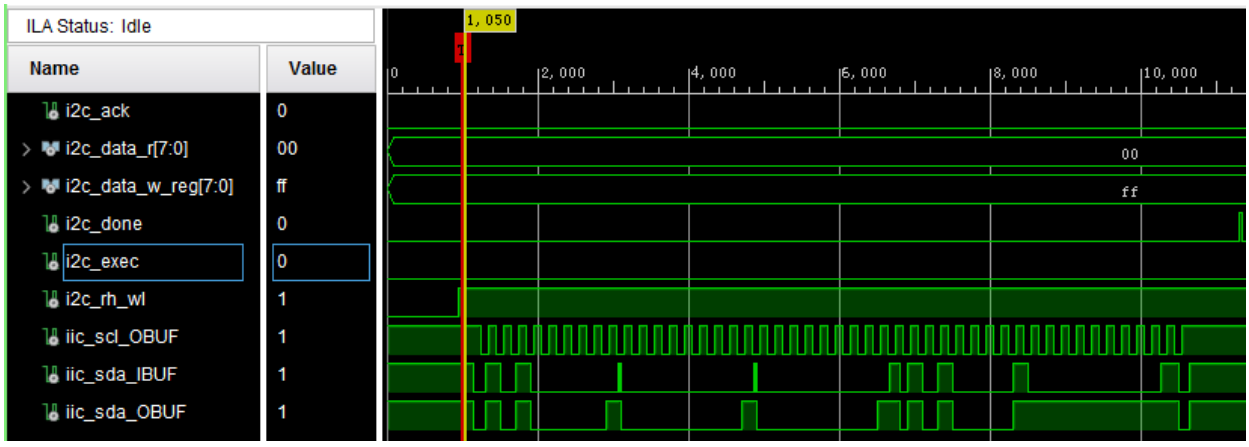


图 19.4.5 EEPROM 读操作 ILA 波形图

从该波形图中我们看到读写控制信号 i2c_rh_wl 为高电平，表示处于读操作状态。当 I2C 触发执行信号 i2c_exec 为高电平时开始执行 I2C 读操作。在 IIC 操作结束后，拉高 i2c_done 信号。另外，在 IIC 读操作期间，i2c_ack（IIC 应答标志）一直处于低电平，说明 EEPROM 响应了主机并应答。

LED 显示模块的代码如下：

```
1  module led_alarm
2      #(parameter L_TIME = 25'd25_000_000
3      )
4      (
5      input         clk       ,  //时钟信号
6      input         rst_n     ,  //复位信号
7
8      input         rw_done   ,  //错误标志
9      input         rw_result ,  //E2PROM 读写测试完成
10     output  reg   led          //E2PROM 读写测试结果 0:失败 1:成功
11 );
12
13 //reg define
14 reg           rw_done_flag;    //读写测试完成标志
15 reg  [24:0]  led_cnt     ;    //led 计数
16
17 //*****************************************************
18 //**                    main code
19 //*****************************************************
20
21 //读写测试完成标志
22 always @(posedge clk or negedge rst_n) begin
23     if(!rst_n)
24         rw_done_flag <= 1'b0;
25     else if(rw_done)
26         rw_done_flag <= 1'b1;
27 end
28
29 //错误标志为 1 时 PL_LED0 闪烁，否则 PL_LED0 常亮
30 always @(posedge clk or negedge rst_n) begin
31     if(!rst_n) begin
32         led_cnt <= 25'd0;
33         led <= 1'b0;
34     end
35     else begin
36         if(rw_done_flag) begin
37             if(rw_result)                          //读写测试正确
38                 led <= 1'b1;                       //led 灯常亮
39             else begin                             //读写测试错误
40                 led_cnt <= led_cnt + 25'd1;
```

```
41                  if(led_cnt == L_TIME - 1'b1) begin
42                      led_cnt <= 25'd0;
43                      led <= ~led;                    //led 灯闪烁
44                  end
45              end
46          end
47          else
48              led <= 1'b0;                            //读写测试完成之前,led 灯熄灭
49      end
50  end
51
52  endmodule
```

led 显示模块利用 LED 灯的显示状态来标识读写过程是否出错。程序中第 21 行至 27 行代码寄存输入的 rw_done 信号，采集到 rw_done 信号之后，拉高 rw_done_flag 信号。

程序中第 21 至 50 行代码根据 rw_done_flag 和 rw_result 信号控制 LED 灯的状态。在 EEPROM 读写测试完成之前，LED 灯处于熄灭状态；如果 EEPROM 读写测试成功，LED 灯处于常亮状态；如果 EEPROM 读写测试失败，LED 灯会不停的闪烁。

19.5 下载验证

将下载器一端连接电脑，另一端与开发板上的 JTAG 下载口连接，连接电源线，并打开开发板的电源开关。

点击 Vivado 左侧“Flow Navigator”窗口最下面的“Open Hardware Manager”，此时 Vivado 软件识别到下载器，点击“Hardware”窗口中“Progam Device”下载程序，在弹出的界面中选择“Program”下载程序。

程序下载完成后，PL_LED0 在短暂延时之后，开始处于常亮的状态，如下图所示：

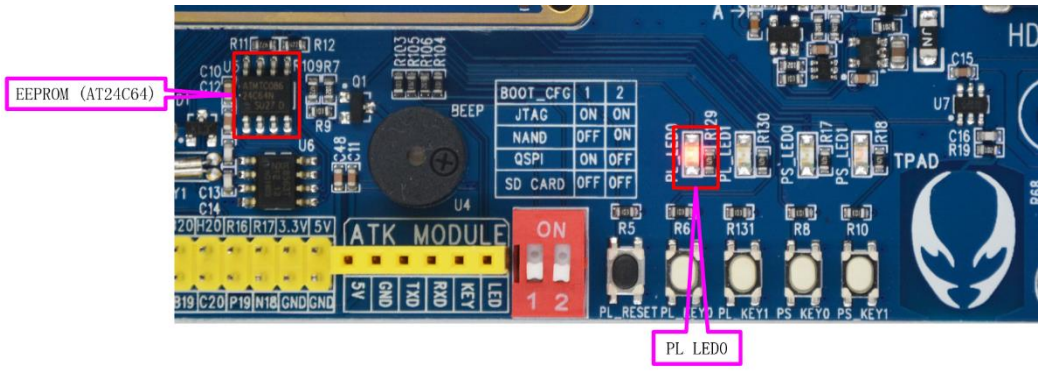


图 19.5.1 实验现象

第二十章 RTC 实时时钟 LCD 显示实验

PCF8563 是一款多功能时钟/日历芯片。因其功耗低、控制简单、封装小而广泛应用于电表、水表、传真机、便携式仪器等产品中。本章我们将使用启明星 Zynq 开发板上的 PCF8563 器件实现实时时钟的显示。

本章包括以下几个部分：

20.1 PCF8563 简介

20.2 实验任务

20.3 硬件设计

20.4 程序设计

20.5 下载验证

20.1 PCF8563 简介

PCF8563 是 PHILIPS 公司推出的一款工业级多功能时钟/日历芯片，具有报警功能、定时器功能、时钟输出功能以及中断输出功能，能完成各种复杂的定时服务。其内部功能模块的框图如下图所示：

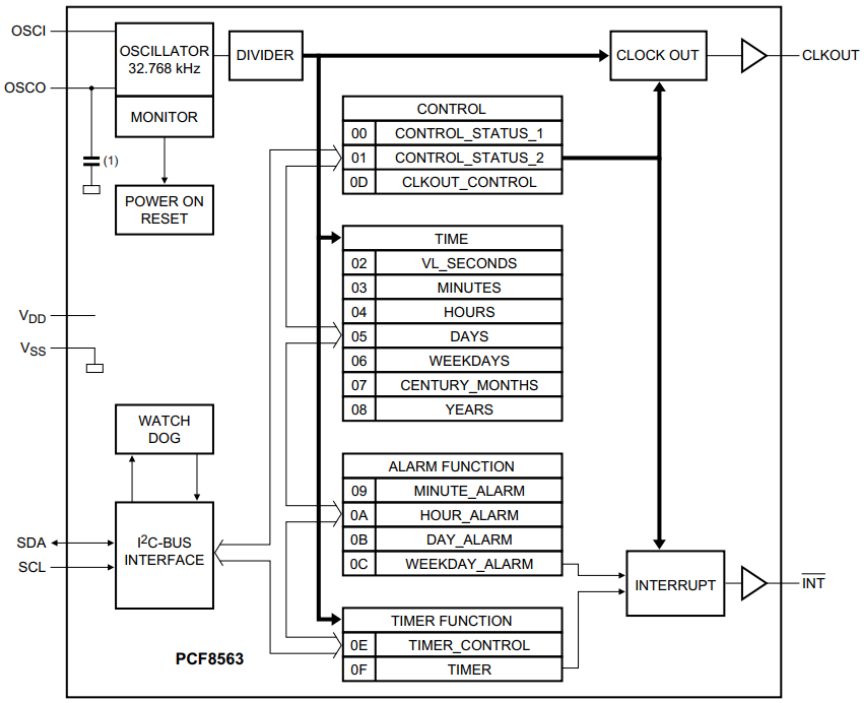


图 20.1.1 PCF8563 功能框图

PCF8563 有 16 个可寻址的 8 位寄存器，但不是所有位都有用到。前两个寄存器（内存地址 00H、01H）用作控制寄存器和状态寄存器（CONTROL_STATUS）；内存地址 02H~08H 用作 TIME 计时器(秒~年计时器)；地址 09H~0CH 用于报警（ALARM）寄存器(定义报警条件)；地址 0DH 控制 CLKOUT 管脚的输出频率；地址 0EH 和 0FH 分别用于定时器控制寄存器和定时器寄存器。

秒、分钟、小时、日、月、年、分钟报警、小时报警、日报警寄存器中的数据编码格式为 BCD，只有星期和星期报警寄存器中的数据不以 BCD 格式编码。BCD 码（Binary-Coded Decimal）是一种二进制的数字编码形式，用四个二进制位来表示一位十进制数（0~9），能够使二进制和十进制之间的转换得以快捷的进行。

PCF8563 通过 I2C 接口与 Zynq 进行通信。使用该器件时，Zynq 先通过 I2C 接口向该器件相应的寄存器写入初始的时间数据（秒~年），然后通过 I2C 接口读取相应的寄存器的时间数据。有关 I2C 总线协议详细的介绍请大家参考“EEPROM 读写实验”。

下面我们对本次实验用到的寄存器做简要的描述和说明，其他寄存器的描述和说明，请大家参考 PCF8563 的数据手册。

秒寄存器的的地址为 02h，说明如下表所示：

表 20.1.1 秒寄存器描述（地址02h）

Bit	符号	描述
7	VL	VL=0保证准确的时钟/日历数据 VL=1不保证准确的时钟/日历数据

6~0	秒	用BCD格式表示的秒数值

当电源电压低于 PCF8563 器件的最低供电电压时，VL 为“1”，表明内部完整的时钟周期信号不能被保证，可能导致时钟/日历数据不准确。

BCD 编码的秒数值如下表所示：

表 20.1.2 秒数值的 BCD 编码

Seconds value (decimal)	Upper-digit (ten's place)			Digit (unit place)			
	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
00	0	0	0	0	0	0	0
01	0	0	0	0	0	0	1
02	0	0	0	0	0	1	0
:	:	:	:	:	:	:	:
09	0	0	0	1	0	0	1
10	0	0	1	0	0	0	0
:	:	:	:	:	:	:	:
58	1	0	1	1	0	0	0
59	1	0	1	1	0	0	1

秒寄存器的地址为 03h，说明如下表所示：

表 20.1.3 分钟寄存器描述（地址03h）

Bit	符号	描述
7	-	无效
6~0	分钟	用BCD格式表示的分钟值

小时寄存器的地址为 04h，说明如下表所示：

表 20.1.4 小时寄存器描述（地址04h）

Bit	符号	描述
7~6	-	无效
5~0	小时	用BCD格式表示的小时值

天寄存器的地址为 05h，说明如下表所示：

表 20.1.5 天寄存器描述（地址05h）

Bit	符号	描述
7~6	-	无效
5~0	天	用BCD格式表示的天数值

当年计数器的值是闰年时，PCF8563 自动给二月增加一个值，使其成为 29 天。

月/世纪寄存器的地址为 07h，说明如下表所示：

表 20.1.6 月/世纪寄存器（地址07h）

Bit	符号	描述
7	C	当C为0时表明是当前世纪，为1时表明是下一世纪
6~5	-	未用
4~0	月	用BCD格式表示的月数值

表 20.1.7 月份表

月份	Bit4	Bit3	Bit2	Bit1	Bit0
一月	0	0	0	0	1
二月	0	0	0	1	0
三月	0	0	0	1	1
四月	0	0	1	0	0
五月	0	0	1	0	1
六月	0	0	1	1	0
七月	0	0	1	1	1
八月	0	1	0	0	0
九月	0	1	0	0	1
十月	1	0	0	0	0
十一月	1	0	0	0	1
十二月	1	0	0	1	0

年寄存器的地址为 08h，说明如下表所示：

表 20.1.8 寄存器（地址08h）

Bit	符号	描述
7~0	年	用BCD格式表示的当前年数值，值为00~99

20.2 实验任务

本节的实验任务是通过启明星 Zynq 开发板上的 PCF8563 实时时钟芯片，在 RGB LCD 液晶屏上来显示时间。

20.3 硬件设计

启明星开发板上 PCF8563 接口部分的原理图如下图所示。

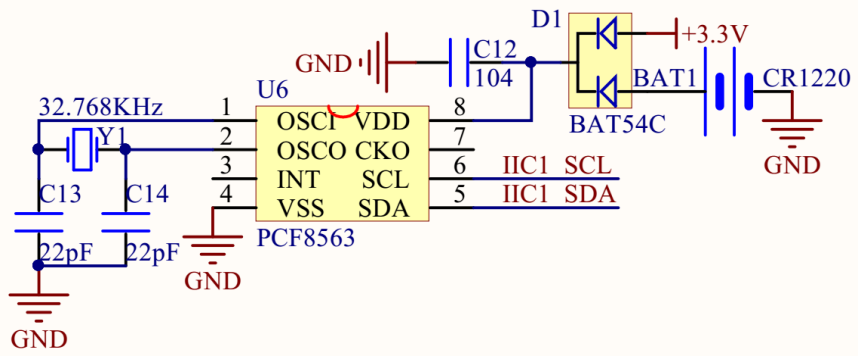


图 20.3.1 PCF8563 接口原理图

PCF8563 作为 I2C 接口的从器件与 EEPROM 等模块统一挂接在启明星开发板上的 IIC 总线上。OSCI、OSCO 与外部 32.768KHz 的晶振相连，为芯片提供驱动时钟；SCL 和 SDA 分别是 I2C 总线的串行时钟接口和串行数据接口。

由于本实验中的管脚较多，这里仅给出 XDC 约束语句，XDC 约束语句如下

```
set_property -dict {PACKAGE_PIN U18 IOSTANDARD LVCMOS33} [get_ports sys_clk]
set_property -dict {PACKAGE_PIN J15 IOSTANDARD LVCMOS33} [get_ports sys_rst_n]

set_property -dict {PACKAGE_PIN M17 IOSTANDARD LVCMOS33} [get_ports iic_scl]
set_property -dict {PACKAGE_PIN M18 IOSTANDARD LVCMOS33} [get_ports iic_sda]

set_property -dict {PACKAGE_PIN Y18 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[0]}]
set_property -dict {PACKAGE_PIN Y19 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[1]}]
set_property -dict {PACKAGE_PIN W20 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[2]}]
set_property -dict {PACKAGE_PIN V20 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[3]}]
set_property -dict {PACKAGE_PIN U14 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[4]}]
set_property -dict {PACKAGE_PIN U15 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[5]}]
set_property -dict {PACKAGE_PIN T20 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[6]}]
set_property -dict {PACKAGE_PIN U20 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[7]}]
set_property -dict {PACKAGE_PIN W14 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[8]}]
set_property -dict {PACKAGE_PIN Y14 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[9]}]
set_property -dict {PACKAGE_PIN N15 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[10]}]
set_property -dict {PACKAGE_PIN N16 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[11]}]
set_property -dict {PACKAGE_PIN V16 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[12]}]
set_property -dict {PACKAGE_PIN W16 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[13]}]
set_property -dict {PACKAGE_PIN W18 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[14]}]
set_property -dict {PACKAGE_PIN W19 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[15]}]
set_property -dict {PACKAGE_PIN T10 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[16]}]
set_property -dict {PACKAGE_PIN T11 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[17]}]
set_property -dict {PACKAGE_PIN P14 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[18]}]
```

```
set_property -dict {PACKAGE_PIN R14 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[19]}]
set_property -dict {PACKAGE_PIN V13 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[20]}]
set_property -dict {PACKAGE_PIN U13 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[21]}]
set_property -dict {PACKAGE_PIN G15 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[22]}]
set_property -dict {PACKAGE_PIN H15 IOSTANDARD LVCMOS33} [get_ports {lcd_rgb[23]}]

set_property -dict {PACKAGE_PIN U17 IOSTANDARD LVCMOS33} [get_ports lcd_hs]
set_property -dict {PACKAGE_PIN P20 IOSTANDARD LVCMOS33} [get_ports lcd_vs]
set_property -dict {PACKAGE_PIN N20 IOSTANDARD LVCMOS33} [get_ports lcd_de]
set_property -dict {PACKAGE_PIN Y16 IOSTANDARD LVCMOS33} [get_ports lcd_bl]
set_property -dict {PACKAGE_PIN T16 IOSTANDARD LVCMOS33} [get_ports lcd_clk]
set_property -dict {PACKAGE_PIN U17 IOSTANDARD LVCMOS33} [get_ports lcd_hs]
set_property -dict {PACKAGE_PIN P20 IOSTANDARD LVCMOS33} [get_ports lcd_vs]
set_property -dict {PACKAGE_PIN N20 IOSTANDARD LVCMOS33} [get_ports lcd_de]
set_property -dict {PACKAGE_PIN Y16 IOSTANDARD LVCMOS33} [get_ports lcd_bl]
set_property -dict {PACKAGE_PIN T16 IOSTANDARD LVCMOS33} [get_ports lcd_clk]
```

20.4 程序设计

根据实验任务，我们可以大致规划出系统的控制流程：ZYNQ 首先通过 I2C 总线向 PCF8563 写入初始时间值，然后不断地读取时间数据，并将读到的时间数据显示到 LCD 上。由此画出系统的功能框图如下所示：

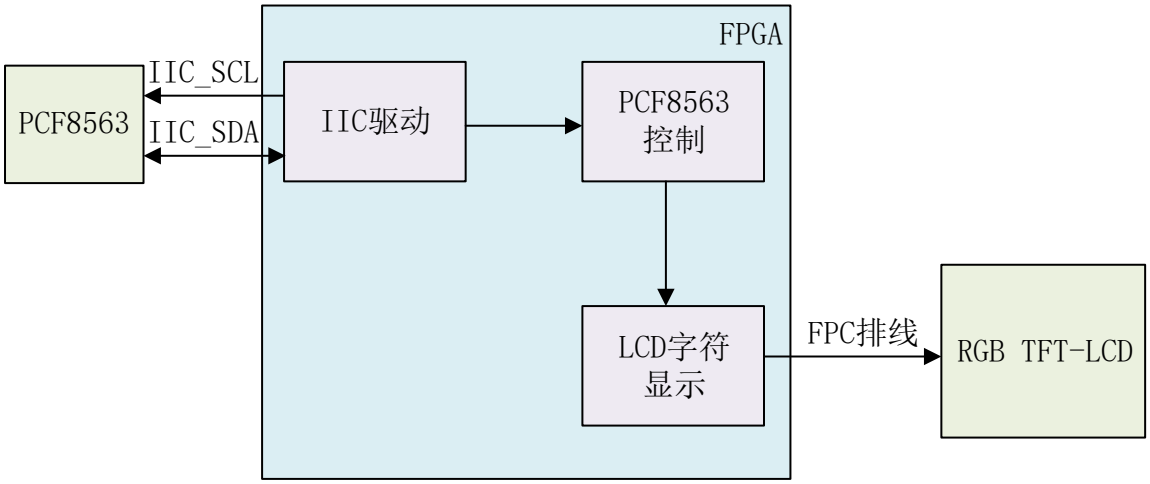


图 20.4.1 PCF8563T 实时时钟 LCD 显示系统框图

由系统框图可知，顶层模块（rtc_lcd）例化了以下三个模块，分别是 IIC 驱动模块（iic_dri）、PCF8563 控制模块（pcf8563_ctrl）和 LCD 字符显示模块（lcd_disp_char）。其中 LCD 字符显示模块例化了读取 ID 模块（rd_id）、时钟分频模块（clk_div）、LCD 显示模块（lcd_display）以及 LCD 驱动模块（lcd_driver）。

各模块端口及信号连接如图 20.4.2 所示：

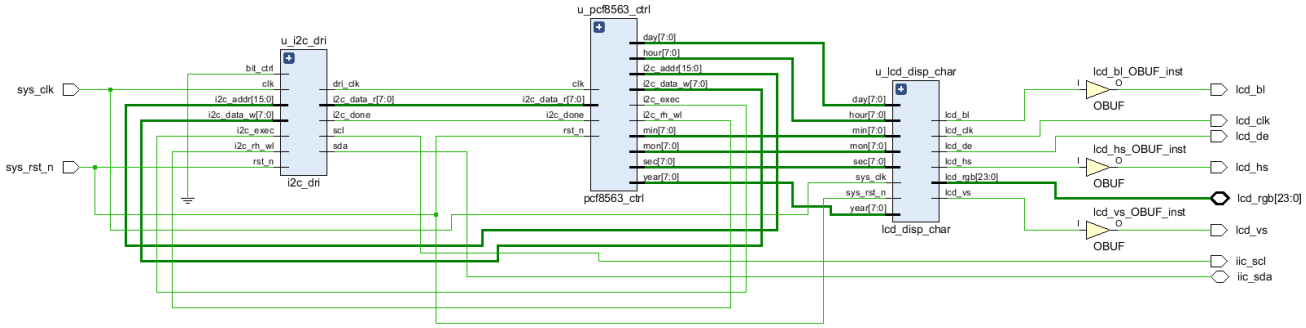


图 20.4.2 顶层模块原理图

PCF8563 实时时钟控制模块（pcf8563_ctrl）通过与 IIC 驱动模块（iic_dri）进行通信来实现对 PCF8563 实时时钟数据的读取；PCF8563 实时时钟控制模块（pcf8563_ctrl）再将从 IIC 读取的时间数据送给 LCD 字符显示模块（lcd_disp_char），以进行显示。

顶层模块的代码如下：

```
1   module rtc_lcd(
2       input                   sys_clk,      //系统时钟
3       input                   sys_rst_n,    //系统复位
4
5       //RGB LCD 接口
6       output                  lcd_de,       //LCD 数据使能信号
7       output                  lcd_hs,       //LCD 行同步信号
8       output                  lcd_vs,       //LCD 场同步信号
9       output                  lcd_bl,       //LCD 背光控制信号
10      output                  lcd_clk,      //LCD 像素时钟
11      inout         [23:0]    lcd_rgb,      //LCD RGB888 颜色数据
12
13      //RTC 实时时钟
14      output                  iic_scl,      //RTC 的时钟线 scl
15      inout                   iic_sda       //RTC 的数据线 sda
16      );
17
18  //parameter define
19  parameter    SLAVE_ADDR = 7'b101_0001    ; //器件地址(SLAVE_ADDR)
20  parameter    BIT_CTRL   = 1'b0           ; //字地址位控制参数(16b/8b)
21  parameter    CLK_FREQ   = 26'd50_000_000; //i2c_dri 模块的驱动时钟频率(CLK_FREQ)
22  parameter    I2C_FREQ   = 18'd250_000    ; //I2C 的 SCL 时钟频率
23  parameter    TIME_INIT  = 48'h19_01_01_09_30_00;//初始时间
24
25  //wire define
26  wire          dri_clk   ;   //I2C 操作时钟
27  wire          i2c_exec  ;   //I2C 触发控制
28  wire  [15:0]  i2c_addr  ;   //I2C 操作地址
```

```
29  wire  [ 7:0]  i2c_data_w;   //I2C 写入的数据
30  wire          i2c_done  ;   //I2C 操作结束标志
31  wire          i2c_ack   ;   //I2C 应答标志 0:应答 1:未应答
32  wire          i2c_rh_wl ;   //I2C 读写控制
33  wire  [ 7:0]  i2c_data_r;   //I2C 读出的数据
34
35  wire    [7:0]  sec      ;   //秒
36  wire    [7:0]  min      ;   //分
37  wire    [7:0]  hour     ;   //时
38  wire    [7:0]  day      ;   //日
39  wire    [7:0]  mon      ;   //月
40  wire    [7:0]  year     ;   //年
41
42  //*****************************************************
43  //**                    main code
44  //*****************************************************
45
46  //i2c 驱动模块
47  i2c_dri #(
48      .SLAVE_ADDR  (SLAVE_ADDR),  //EEPROM 从机地址
49      .CLK_FREQ    (CLK_FREQ  ),  //模块输入的时钟频率
50      .I2C_FREQ    (I2C_FREQ  )   //IIC_SCL 的时钟频率
51  ) u_i2c_dri(
52      .clk         (sys_clk   ),
53      .rst_n       (sys_rst_n ),
54      //i2c interface
55      .i2c_exec    (i2c_exec  ),
56      .bit_ctrl    (BIT_CTRL  ),
57      .i2c_rh_wl   (i2c_rh_wl ),
58      .i2c_addr    (i2c_addr  ),
59      .i2c_data_w  (i2c_data_w),
60      .i2c_data_r  (i2c_data_r),
61      .i2c_done    (i2c_done  ),
62      .i2c_ack     (i2c_ack   ),
63      .scl         (iic_scl   ),
64      .sda         (iic_sda   ),
65      //user interface
66      .dri_clk     (dri_clk   )
67  );
68
69  //PCF8563 控制模块
```

```
70  pcf8563_ctrl #(
71      .TIME_INIT (TIME_INIT)
72     )u_pcf8563_ctrl(
73      .clk          (dri_clk   ),
74      .rst_n        (sys_rst_n ),
75      //IIC
76      .i2c_rh_wl    (i2c_rh_wl ),
77      .i2c_exec     (i2c_exec  ),
78      .i2c_addr     (i2c_addr  ),
79      .i2c_data_w   (i2c_data_w),
80      .i2c_data_r   (i2c_data_r),
81      .i2c_done     (i2c_done  ),
82      //时间和日期
83      .sec          (sec       ),
84      .min          (min       ),
85      .hour         (hour      ),
86      .day          (day       ),
87      .mon          (mon       ),
88      .year         (year      )
89      );
90
91  //LCD 字符显示模块
92  lcd_disp_char u_lcd_disp_char(
93      .sys_clk      (sys_clk   ),
94      .sys_rst_n    (sys_rst_n ),
95      //时间和日期
96      .sec          (sec       ),
97      .min          (min       ),
98      .hour         (hour      ),
99      .day          (day       ),
100     .mon          (mon       ),
101     .year         (year      ),
102     //RGB LCD 接口
103     .lcd_de       (lcd_de    ),
104     .lcd_hs       (lcd_hs    ),
105     .lcd_vs       (lcd_vs    ),
106     .lcd_bl       (lcd_bl    ),
107     .lcd_clk      (lcd_clk   ),
108     .lcd_rgb      (lcd_rgb   )
109     );
110
```

```
111 endmodule
```

代码中第 18 至 23 行定义了一些参数，其中 TIME_INIT 表示 RTC 实时时钟的初始日期和时间，可以通过修改此参数值使 PCF8563 从不同的时间开始计时，例如从 2019 年 1 月 1 号 09：30：00 开始计时，需要将该参数值设置为 48'h190101093000。

顶层模块中主要完成对其余模块的例化。其中 I2C 驱动模块（iic_dri）的代码与“EEPROM 读写实验”章节中的 IIC 驱动模块完全相同，只是在例化时对字地址位控制（BIT_CTRL）和 IIC 器件地址（SLAVE_ADDR）两个参数作了修改，有关 IIC 驱动模块的详细介绍请大家参考“EEPROM 读写实验”。

PCF8563 实时时钟控制模块的代码如下所示：

```
1  module pcf8563_ctrl #(
2      // 初始时间设置，从高到低为年到秒，各占 8bit
3      parameter  TIME_INIT = 48'h19_10_26_09_30_00)(
4      input                clk       , //时钟信号
5      input                rst_n     , //复位信号
6
7      //i2c interface
8      output   reg         i2c_rh_wl , //I2C 读写控制信号
9      output   reg         i2c_exec  , //I2C 触发执行信号
10     output   reg  [15:0] i2c_addr  , //I2C 器件内地址
11     output   reg  [7:0]  i2c_data_w, //I2C 要写的数据
12     input         [7:0]  i2c_data_r, //I2C 读出的数据
13     input                i2c_done  , //I2C 一次操作完成
14
15     //PCF8563T 的秒、分、时、日、月、年数据
16     output   reg  [7:0]  sec,        //秒
17     output   reg  [7:0]  min,        //分
18     output   reg  [7:0]  hour,       //时
19     output   reg  [7:0]  day,        //日
20     output   reg  [7:0]  mon,        //月
21     output   reg  [7:0]  year        //年
22 );
23
24 //reg define
25 reg   [3:0]     flow_cnt  ;          // 状态流控制
26 reg   [12:0]    wait_cnt  ;          // 计数等待
27
28 //*****************************************************
29 //**                    main code
30 //*****************************************************
31
32 //先向 PCF8563 中写入初始化日期和时间，再从中读出日期和时间
33 always @(posedge clk or negedge rst_n) begin
```

```
34      if(!rst_n) begin
35          sec         <= 8'h0;
36          min         <= 8'h0;
37          hour        <= 8'h0;
38          day         <= 8'h0;
39          mon         <= 8'h0;
40          year        <= 8'h0;
41          i2c_exec    <= 1'b0;
42          i2c_rh_wl   <= 1'b0;
43          i2c_addr    <= 8'd0;
44          i2c_data_w  <= 8'd0;
45          flow_cnt    <= 4'd0;
46          wait_cnt    <= 13'd0;
47      end
48      else begin
49          i2c_exec <= 1'b0;
50          case(flow_cnt)
51              //上电初始化
52              4'd0: begin
53                  if(wait_cnt == 13'd8000) begin
54                      wait_cnt<= 12'd0;
55                      flow_cnt<= flow_cnt + 1'b1;
56                  end
57                  else
58                      wait_cnt<= wait_cnt + 1'b1;
59              end
60              //写读秒
61              4'd1: begin
62                  i2c_exec  <= 1'b1;
63                  i2c_addr  <= 8'h02;
64                  flow_cnt  <= flow_cnt + 1'b1;
65                  i2c_data_w<= TIME_INIT[7:0];
66              end
67              4'd2: begin
68                  if(i2c_done == 1'b1) begin
69                      sec     <= i2c_data_r[6:0];
70                      flow_cnt<= flow_cnt + 1'b1;
71                  end
72              end
73              //写读分
74              4'd3: begin
```

```
75                  i2c_exec  <= 1'b1;
76                  i2c_addr  <= 8'h03;
77                  flow_cnt  <= flow_cnt + 1'b1;
78                  i2c_data_w<= TIME_INIT[15:8];
79              end
80              4'd4: begin
81                  if(i2c_done == 1'b1) begin
82                      min     <= i2c_data_r[6:0];
83                      flow_cnt<= flow_cnt + 1'b1;
84                  end
85              end
86              //写读时
87              4'd5: begin
88                  i2c_exec  <= 1'b1;
89                  i2c_addr  <= 8'h04;
90                  flow_cnt  <= flow_cnt + 1'b1;
91                  i2c_data_w<= TIME_INIT[23:16];
92              end
93              4'd6: begin
94                  if(i2c_done == 1'b1) begin
95                      hour    <= i2c_data_r[5:0];
96                      flow_cnt<= flow_cnt + 1'b1;
97                  end
98              end
99              //写读天
100             4'd7: begin
101                 i2c_exec  <= 1'b1;
102                 i2c_addr  <= 8'h05;
103                 flow_cnt  <= flow_cnt + 1'b1;
104                 i2c_data_w<= TIME_INIT[31:24];
105             end
106             4'd8: begin
107                 if(i2c_done == 1'b1) begin
108                     day     <= i2c_data_r[5:0];
109                     flow_cnt<= flow_cnt + 1'b1;
110                 end
111             end
112             //写读月
113             4'd9: begin
114                 i2c_exec  <= 1'b1;
115                 i2c_addr  <= 8'h07;
```

```
116                 flow_cnt  <= flow_cnt + 1'b1;
117                 i2c_data_w<= TIME_INIT[39:32];
118             end
119             4'd10: begin
120                 if(i2c_done == 1'b1) begin
121                     mon     <= i2c_data_r[4:0];
122                     flow_cnt<= flow_cnt + 1'b1;
123                 end
124             end
125             //写读年
126             4'd11: begin
127                 i2c_exec  <= 1'b1;
128                 i2c_addr  <= 8'h08;
129                 flow_cnt  <= flow_cnt + 1'b1;
130                 i2c_data_w<= TIME_INIT[47:40];
131             end
132             4'd12: begin
133                 if(i2c_done == 1'b1) begin
134                     year     <= i2c_data_r;
135                     i2c_rh_wl<= 1'b1;
136                     flow_cnt <= 4'd1;
137                 end
138             end
139             default: flow_cnt <= 4'd0;
140         endcase
141     end
142 end
143
144 endmodule
```

程序中定义了一个状态流控制计数器（flow_cnt），先将初始日期和时间（TIME_INIT）写入 PCF8563 中，然后会循环从 PCF8563 中读出秒、分、时、日、月和年。在写操作是 i2c_rh_wl（I2C 读写控制信号）为低电平，读操作时拉高 i2c_rh_wl 信号。

LCD 字符显示模块（lcd_disp_char）的代码由“RGB TFT-LCD 字符和图片显示”实验的代码修改而来，除 lcd_disp_char 顶层模块外，唯一不同的地方在 LCD 显示模块。

LCD 显示模块的代码如下所示：

```
1   module lcd_display(
2       input               lcd_pclk ,
3       input               rst_n ,
4
5       //日历数据
6       input        [7:0]  sec,        //秒
```

```
7       input          [7:0]  min,         //分
8       input          [7:0]  hour,        //时
9       input          [7:0]  day,         //日
10      input          [7:0]  mon,         //月
11      input          [7:0]  year,        //年
12
13      //LCD 数据接口
14      input         [10:0]  pixel_xpos, //像素点横坐标
15      input         [10:0]  pixel_ypos, //像素点纵坐标
16      output  reg   [23:0]  pixel_data  //像素点数据
17  );
18
19  //parameter define
20  localparam CHAR_POS_X_1  = 11'd1;  //第 1 行字符区域起始点横坐标
21  localparam CHAR_POS_Y_1  = 11'd1;  //第 1 行字符区域起始点纵坐标
22  localparam CHAR_POS_X_2  = 11'd17; //第 2 行字符区域起始点横坐标
23  localparam CHAR_POS_Y_2  = 11'd17; //第 2 行字符区域起始点纵坐标
24  localparam CHAR_WIDTH_1  = 11'd80; //第 1 行字符区域的宽度，第 1 行共 10 个字符(加空格)
25  localparam CHAR_WIDTH_2  = 11'd64; //第 2 行字符区域的宽度，第 2 行共 8 个字符(加空格)
26  localparam CHAR_HEIGHT   = 11'd16; //单个字符的高度
27  localparam WHITE  = 24'hffffff;    //背景色,白色
28  localparam BLACK  = 24'h000000;    //字符颜色,黑色
29
30  //reg define
31  reg  [127:0]  char  [9:0] ;        //字符数组
32
33  //**************************************************
34  //**                    main code
35  //**************************************************
36
37  //字符数组初始值,用于存储字模数据(由取模软件生成,单个数字字体大小:16*16)
38  always @(posedge lcd_pclk ) begin
39      char[0] <= 128'h00000018244242424242424224180000 ;  // "0"
40      char[1] <= 128'h00000010701010101010101007C0000 ;  // "1"
41      char[2] <= 128'h0000003C4242420404081020427E0000 ;  // "2"
42      char[3] <= 128'h0000003C4242041804020242443800000 ;  // "3"
43      char[4] <= 128'h000000040C14242444447E04041E0000 ;  // "4"
44      char[5] <= 128'h0000007E4040405864020242443800000 ;  // "5"
45      char[6] <= 128'h0000001C2440405864424242241800000 ;  // "6"
46      char[7] <= 128'h0000007E4444080810101010101000000 ;  // "7"
47      char[8] <= 128'h0000003C4242422418244242423C0000 ;  // "8"
```

```
48      char[9] <= 128'h00000018244242422261A020224380000 ;  // "9"
49  end
50
51  //不同的区域绘制不同的像素数据
52  always @(posedge lcd_pclk or negedge rst_n ) begin
53      if (!rst_n)  begin
54          pixel_data <= BLACK;
55      end
56
57      //在第一行显示年的千位 固定值"2"
58      else if(     (pixel_xpos >= CHAR_POS_X_1)
59                && (pixel_xpos <  CHAR_POS_X_1 + CHAR_WIDTH_1/10*1)
60                && (pixel_ypos >= CHAR_POS_Y_1)
61                && (pixel_ypos <  CHAR_POS_Y_1 + CHAR_HEIGHT)  ) begin
62          if(char [2] [ (CHAR_HEIGHT+CHAR_POS_Y_1 - pixel_ypos)*8
63                        - ((pixel_xpos-CHAR_POS_X_1)%8) -1 ] )
64              pixel_data <= BLACK;          //显示字符为黑色
65          else
66              pixel_data <= WHITE;         //显示字符区域背景为白色
67      end
68
69      //在第一行显示年的百位 固定值"0"
70      else if(     (pixel_xpos >= CHAR_POS_X_1 + CHAR_WIDTH_1/10*1)
71                && (pixel_xpos <  CHAR_POS_X_1 + CHAR_WIDTH_1/10*2)
72                && (pixel_ypos >= CHAR_POS_Y_1)
73                && (pixel_ypos <  CHAR_POS_Y_1 + CHAR_HEIGHT)  ) begin
74          if(char [0] [ (CHAR_HEIGHT+CHAR_POS_Y_1 - pixel_ypos)*8
75                        - ((pixel_xpos-CHAR_POS_X_1)%8) -1 ]  )
76              pixel_data <= BLACK;
77          else
78              pixel_data <= WHITE;
79      end
80
81      //在第一行显示年的十位
82      else if(     (pixel_xpos >= CHAR_POS_X_1 + CHAR_WIDTH_1/10*2)
83                && (pixel_xpos <  CHAR_POS_X_1 + CHAR_WIDTH_1/10*3)
84                && (pixel_ypos >= CHAR_POS_Y_1)
85                && (pixel_ypos <  CHAR_POS_Y_1 + CHAR_HEIGHT)  ) begin
86          if(char [year[7:4]] [ (CHAR_HEIGHT+CHAR_POS_Y_1 - pixel_ypos)*8
87                                - ((pixel_xpos-CHAR_POS_X_1)%8) -1 ]  )
88              pixel_data <= BLACK;
```

```
89          else
90              pixel_data <= WHITE;
91      end
92
93      //在第一行显示年的个位
94      else if(     (pixel_xpos >= CHAR_POS_X_1 + CHAR_WIDTH_1/10*3)
95                && (pixel_xpos <  CHAR_POS_X_1 + CHAR_WIDTH_1/10*4)
96                && (pixel_ypos >= CHAR_POS_Y_1)
97                && (pixel_ypos <  CHAR_POS_Y_1 + CHAR_HEIGHT)  ) begin
98          if(char [year[3:0]] [ (CHAR_HEIGHT+CHAR_POS_Y_1 - pixel_ypos)*8
99                                - ((pixel_xpos-CHAR_POS_X_1)%8) -1 ]  )
100             pixel_data <= BLACK;
101         else
102             pixel_data <= WHITE;
103     end
104
105     //在第一行显示空格
106     else if(     (pixel_xpos >= CHAR_POS_X_1 + CHAR_WIDTH_1/10*4)
107               && (pixel_xpos <  CHAR_POS_X_1 + CHAR_WIDTH_1/10*5)
108               && (pixel_ypos >= CHAR_POS_Y_1)
109               && (pixel_ypos <  CHAR_POS_Y_1 + CHAR_HEIGHT)  ) begin
110         pixel_data <= WHITE;
111     end
112
113     //在第一行显示月的十位
114     else if(     (pixel_xpos >= CHAR_POS_X_1 + CHAR_WIDTH_1/10*5)
115               && (pixel_xpos <  CHAR_POS_X_1 + CHAR_WIDTH_1/10*6)
116               && (pixel_ypos >= CHAR_POS_Y_1)
117               && (pixel_ypos <  CHAR_POS_Y_1 + CHAR_HEIGHT)) begin
118         if(char [mon[7:4]] [ (CHAR_HEIGHT+CHAR_POS_Y_1 - pixel_ypos)*8
119                               - ((pixel_xpos-CHAR_POS_X_1)%8) -1 ] )
120             pixel_data <= BLACK;
121         else
122             pixel_data <= WHITE;
123     end
124
125     //在第一行显示月的个位
126     else if(     (pixel_xpos >= CHAR_POS_X_1 + CHAR_WIDTH_1/10*6)
127               && (pixel_xpos <  CHAR_POS_X_1 + CHAR_WIDTH_1/10*7)
128               && (pixel_ypos >= CHAR_POS_Y_1)
129               && (pixel_ypos <  CHAR_POS_Y_1 + CHAR_HEIGHT)  ) begin
```

```
130         if(char [mon[3:0]] [ (CHAR_HEIGHT+CHAR_POS_Y_1 - pixel_ypos)*8
131                              - ((pixel_xpos-CHAR_POS_X_1)%8) -1 ] )
132             pixel_data <= BLACK;
133         else
134             pixel_data <= WHITE;
135     end
136
137     //在第一行显示空格
138     else if(     (pixel_xpos >= CHAR_POS_X_1 + CHAR_WIDTH_1/10*7)
139               && (pixel_xpos <  CHAR_POS_X_1 + CHAR_WIDTH_1/10*8)
140               && (pixel_ypos >= CHAR_POS_Y_1)
141               && (pixel_ypos <  CHAR_POS_Y_1 + CHAR_HEIGHT)  ) begin
142         pixel_data <= WHITE;
143     end
144
145     //在第一行显示日的十位
146     else if(     (pixel_xpos >= CHAR_POS_X_1 + CHAR_WIDTH_1/10*8)
147               && (pixel_xpos <  CHAR_POS_X_1 + CHAR_WIDTH_1/10*9)
148               && (pixel_ypos >= CHAR_POS_Y_1)
149               && (pixel_ypos <  CHAR_POS_Y_1 + CHAR_HEIGHT)  ) begin
150         if(char [day[7:4]] [ (CHAR_HEIGHT+CHAR_POS_Y_1 - pixel_ypos)*8
151                              - ((pixel_xpos-CHAR_POS_X_1)%8) -1 ] )
152             pixel_data <= BLACK;
153         else
154             pixel_data <= WHITE;
155     end
156
157     //在第一行显示日的个位
158     else if(     (pixel_xpos >= CHAR_POS_X_1 + CHAR_WIDTH_1/10*9)
159               && (pixel_xpos <  CHAR_POS_X_1 + CHAR_WIDTH_1)
160               && (pixel_ypos >= CHAR_POS_Y_1)
161               && (pixel_ypos <  CHAR_POS_Y_1 + CHAR_HEIGHT)  ) begin
162         if(char [day[3:0]] [ (CHAR_HEIGHT+CHAR_POS_Y_1 - pixel_ypos)*8
163                              - ((pixel_xpos-CHAR_POS_X_1)%8) -1 ] )
164             pixel_data <= BLACK;
165         else
166             pixel_data <= WHITE;
167     end
168
169     //在第二行显示时的十位
170     else if(     (pixel_xpos >= CHAR_POS_X_2)
```

```
171             && (pixel_xpos <  CHAR_POS_X_2 + CHAR_WIDTH_2/8*1)
172             && (pixel_ypos >= CHAR_POS_Y_2)
173             && (pixel_ypos <  CHAR_POS_Y_2 + CHAR_HEIGHT)  ) begin
174         if(char [hour[7:4]] [ (CHAR_HEIGHT+CHAR_POS_Y_2 - pixel_ypos)*8
175                               - ((pixel_xpos-CHAR_POS_X_2)%8) -1 ] )
176             pixel_data <= BLACK;
177         else
178             pixel_data <= WHITE;
179     end
180
181     //在第二行显示时的个位
182     else if(     (pixel_xpos >= CHAR_POS_X_2 + CHAR_WIDTH_2/8*1)
183             && (pixel_xpos <  CHAR_POS_X_2 + CHAR_WIDTH_2/8*2)
184             && (pixel_ypos >= CHAR_POS_Y_2)
185             && (pixel_ypos <  CHAR_POS_Y_2 + CHAR_HEIGHT)  ) begin
186         if(char [hour[3:0]] [ (CHAR_HEIGHT+CHAR_POS_Y_2 - pixel_ypos)*8
187                               - ((pixel_xpos-CHAR_POS_X_2)%8) -1 ] )
188             pixel_data <= BLACK;
189         else
190             pixel_data <= WHITE;
191     end
192
193     //在第二行显示空格
194     else if(     (pixel_xpos >= CHAR_POS_X_2 + CHAR_WIDTH_2/8*2)
195             && (pixel_xpos <  CHAR_POS_X_2 + CHAR_WIDTH_2/8*3)
196             && (pixel_ypos >= CHAR_POS_Y_2)
197             && (pixel_ypos <  CHAR_POS_Y_2 + CHAR_HEIGHT)  ) begin
198         pixel_data <= WHITE;
199     end
200
201     //在第二行显示分的十位
202     else if(     (pixel_xpos >= CHAR_POS_X_2 + CHAR_WIDTH_2/8*3)
203             && (pixel_xpos <  CHAR_POS_X_2 + CHAR_WIDTH_2/8*4)
204             && (pixel_ypos >= CHAR_POS_Y_2)
205             && (pixel_ypos <  CHAR_POS_Y_2 + CHAR_HEIGHT)  ) begin
206         if(char [min[7:4]] [ (CHAR_HEIGHT+CHAR_POS_Y_2 - pixel_ypos)*8
207                              - ((pixel_xpos-CHAR_POS_X_2)%8) -1 ] )
208             pixel_data <= BLACK;
209         else
210             pixel_data <= WHITE;
211     end
```

```
212
213     //在第二行显示分的个位
214     else if(     (pixel_xpos >= CHAR_POS_X_2 + CHAR_WIDTH_2/8*4)
215               && (pixel_xpos <  CHAR_POS_X_2 + CHAR_WIDTH_2/8*5)
216               && (pixel_ypos >= CHAR_POS_Y_2)
217               && (pixel_ypos <  CHAR_POS_Y_2 + CHAR_HEIGHT)  ) begin
218         if(char [min[3:0]] [ (CHAR_HEIGHT+CHAR_POS_Y_2 - pixel_ypos)*8
219                              - ((pixel_xpos-CHAR_POS_X_2)%8) -1 ] )
220             pixel_data <= BLACK;
221         else
222             pixel_data <= WHITE;
223     end
224
225     //在第二行显示空格
226     else if(     (pixel_xpos >= CHAR_POS_X_2 + CHAR_WIDTH_2/8*5)
227               && (pixel_xpos <  CHAR_POS_X_2 + CHAR_WIDTH_2/8*6)
228               && (pixel_ypos >= CHAR_POS_Y_2)
229               && (pixel_ypos <  CHAR_POS_Y_2 + CHAR_HEIGHT)  ) begin
230         pixel_data <= WHITE;
231     end
232
233     //在第二行显示秒的十位
234     else if(     (pixel_xpos >= CHAR_POS_X_2 + CHAR_WIDTH_2/8*6)
235               && (pixel_xpos <  CHAR_POS_X_2 + CHAR_WIDTH_2/8*7)
236               && (pixel_ypos >= CHAR_POS_Y_2)
237               && (pixel_ypos <  CHAR_POS_Y_2 + CHAR_HEIGHT)  ) begin
238         if(char [sec[7:4]] [ (CHAR_HEIGHT+CHAR_POS_Y_2 - pixel_ypos)*8
239                              - ((pixel_xpos-CHAR_POS_X_2)%8) -1 ] )
240             pixel_data <= BLACK;
241         else
242             pixel_data <= WHITE;
243     end
244
245     //在第二行显示秒的个位
246     else if(     (pixel_xpos >= CHAR_POS_X_2 + CHAR_WIDTH_2/8*7)
247               && (pixel_xpos <  CHAR_POS_X_2 + CHAR_WIDTH_2)
248               && (pixel_ypos >= CHAR_POS_Y_2)
249               && (pixel_ypos <  CHAR_POS_Y_2 + CHAR_HEIGHT)  ) begin
250         if(char [sec[3:0]] [ (CHAR_HEIGHT+CHAR_POS_Y_2 - pixel_ypos)*8
251                              - ((pixel_xpos-CHAR_POS_X_2)%8) -1 ] )
252             pixel_data <= BLACK;
```

```
253         else
254             pixel_data <= WHITE;
255     end
256
257     else begin
258         pixel_data <= WHITE;     //屏幕背景为白色
259     end
260 end
261
262 endmodule
```

我们的显示内容首先分成两行，第一行显示年月日，第二行显示时分秒。程序中第 19 至 28 行代码定义了一些参数，前 4 个参数定义每一行字符显示的参考点，结合具体参数值我们知道：第一行是（1，1），第二行是（17，17），接着后面两个参数分别定义了每一行各自显示的宽度（长度），分别是 80 和 64，最后两个参数定义了字符的颜色和背景色，字符颜色为黑色，背景色为白色。

代码第 38 到 49 行定义了 0 到 9 每个阿拉伯数字所对应的数组，具体的每个数组的字模数据都是一个长度为 128 的数组，实际上我们把二维数组的所有数据都放在了第一行上，使用时把它看成一个二维数组，大小为 16*8bit，16 行，每一行有 8 位数据。

代码第 57 到 67 行是一个具体的字符显示的逻辑。首先判断当前像素坐标的位置，如代码第 58 到 61 行，如果处在字符显示的区域则开始根据字符数组值来显示像素。显示时，数组参数 pixel_xpos，pixel_ypos 分别从小到大取不同的值时，代入数组，此时我们实际上就是在从左到右，从上到下扫描一个字符像素平面，pixel_xpos 变化表示行扫描，pixel_ypos 则表示列扫描。

对于第 62 行的代码，“ (CHAR_HEIGHT+CHAR_POS_Y_1 - pixel_ypos)*8”，我们不难理解“*8”的由来，因为在查找数组元素的时候，pixel_ypos 的每次变化表示换到下一行扫描，一行跨过 8 个数据，所有乘以 8。这里就可总结一下：字符数组一行的 128 个数据从高位到低位，每 8 位代表一行，分别对应点阵中该行从左向右的每一个像素点。

代码第 62 行到 63 行是对数组的每个元素分别赋值，具体是数组元素为 1 的点赋值为黑色，否则为白色。其它字符的显示逻辑和上面类似，这里不再赘述。

程序中第 37 至 49 行代码初始化字符数组的值，即数字“0”~“9”的字模数据，由取模软件生成，先将软件设置成字符模式，取模软件的设置如下：

图 20.4.3 字符软件设置

这里将点阵设置为 16，即一个数字的字符用一行来表示。

生成字模的界面如下：

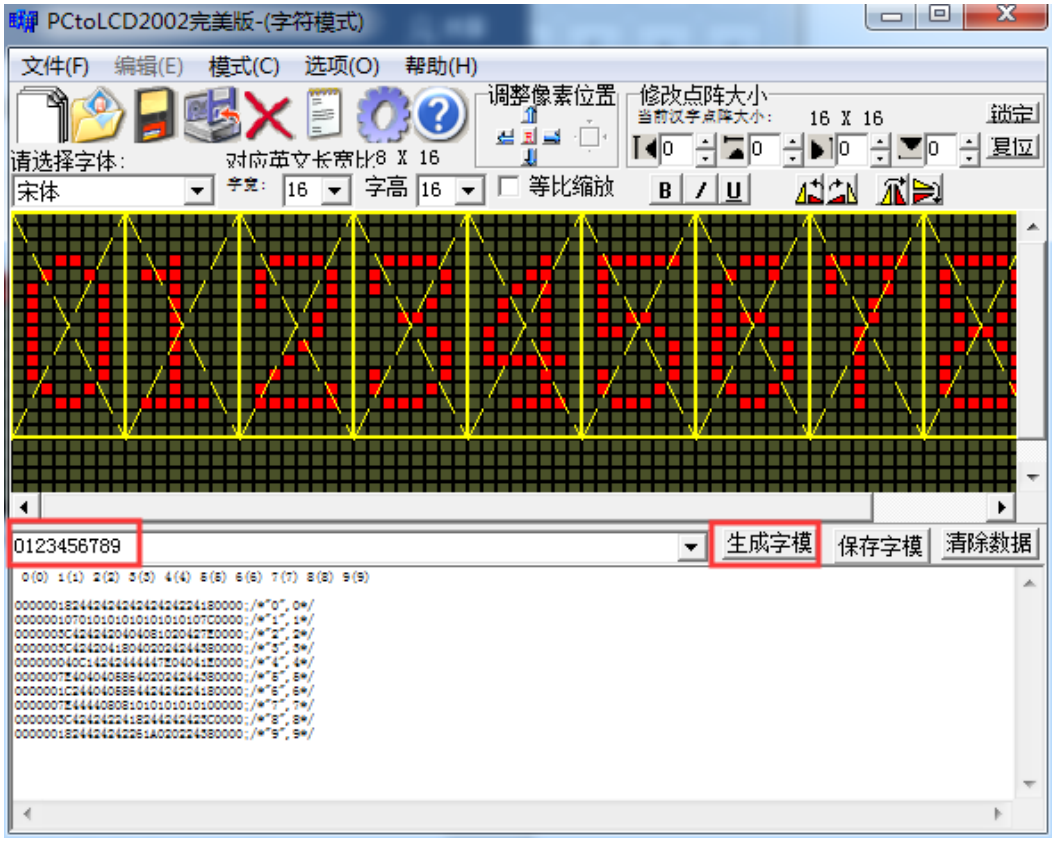


图 20.4.4 生成字模的软件设置

程序中第 51 行至 260 行代码根据输入的日期、时间和字符区域的坐标显示在 LCD 上，字符颜色为黑色，背景色为白色。

20.5 下载验证

首先将 FPC 排线一端与 RGB LCD 模块上的 J1 接口连接，另一端与启明星开发板上的 RGB TFTLCD 接口连接。然后将下载器一端连电脑，另一端与开发板上的 JTAG 端口连接，最后连接电源线并打开电源开关。

接下来我们下载程序，验证 RGB LCD 字符和图片显示的功能。下载完成后观察 RGB LCD 液晶屏上显示出日期和时间，并且时间在不断的计时，如下图所示，说明 RGB TFT-LCD 字符和图片显示程序下载验证成功。

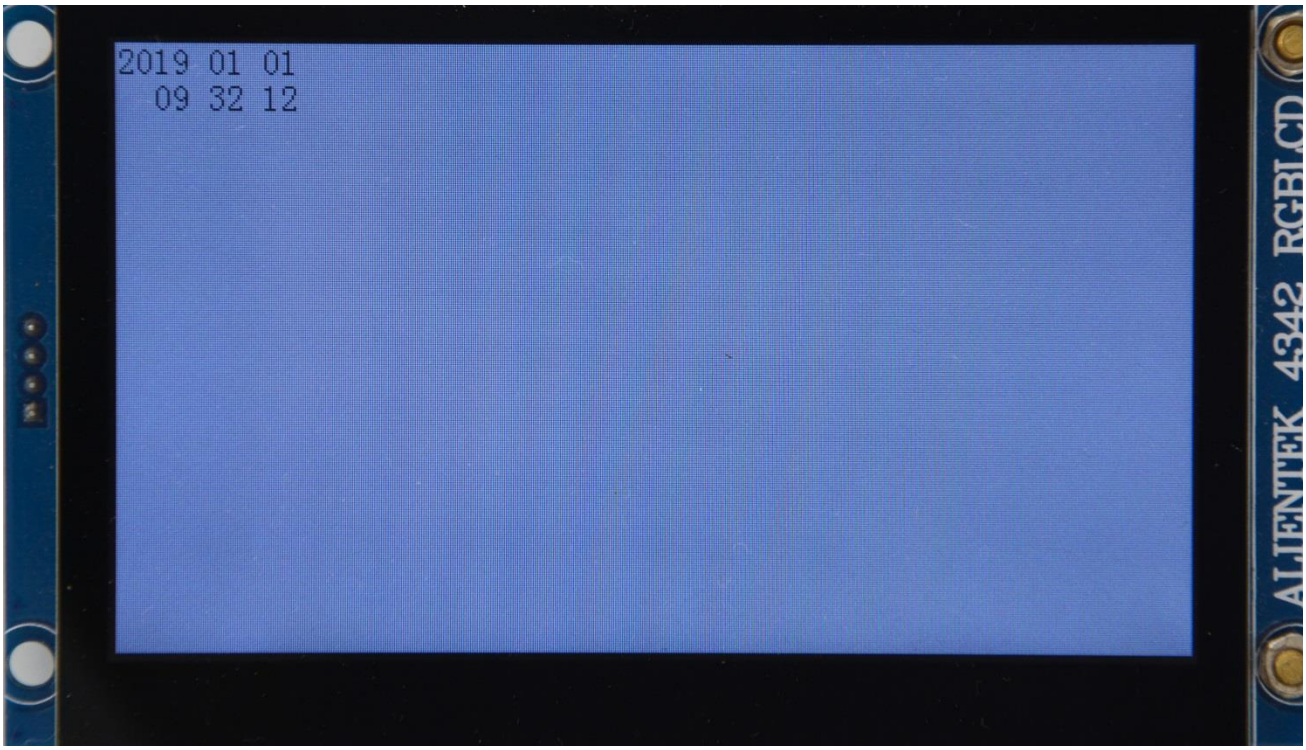


图 20.5.1 实验结果

第二十一章 频率计实验

数字频率计是一种基本的测量仪器，被广泛应用于航天、电子、测控等领域。基于传统测频原理的频率计的测量精度将随被测信号频率的下降而降低，在使用中有较大的局限性，而等精度频率计不但具有较高的测量精度，而且在整个频率区域能保持恒定的测试精度。本章我们通过 Zynq 启明星开发板搭建等精度频率计，学习等精度频率计的设计思想和实现方案。

本章分为以下几个章节：

21.1 等精度频率计简介

21.2 实验任务

21.3 硬件设计

21.4 程序设计

21.5 下载验证

21.1 等精度频率计简介

频率测量在电子设计和测量领域中经常用到，因此对频率测量方法的研究在实际工程应用中具有重要意义。常用的频率测量方法有两种：周期测量法和频率测量法。周期测量法是先测量出被测信号的周期 T，然后根据频率f = 1/T求出被测信号的频率。频率测量法是在时间 t 内对被测信号的脉冲数 N 进行计数，然后求出单位时间内的脉冲数，即为被测信号的频率。但是上述两种方法都会产生±1 个被测脉冲的误差，在实际应用中有一定的局限性。根据测量原理，很容易发现周期测量法适合于低频信号测量，频率测量法适合于高频信号测量，但二者都不能兼顾高低频率同样精度的测量要求。

等精度测量的一个最大特点是测量的实际门控时间不是一个固定值，而是一个与被测信号有关的值，刚好是被测信号的整数倍。在计数允许时间内，同时对基准时钟和被测信号进行计数，再通过数学公式推导得到被测信号的频率。由于门控信号是被测信号的整数倍，就消除了对被测信号产生的±1 周期误差，但是会产生对基准时钟±1 周期的误差。等精度测量原理如图 21.1.1 所示。

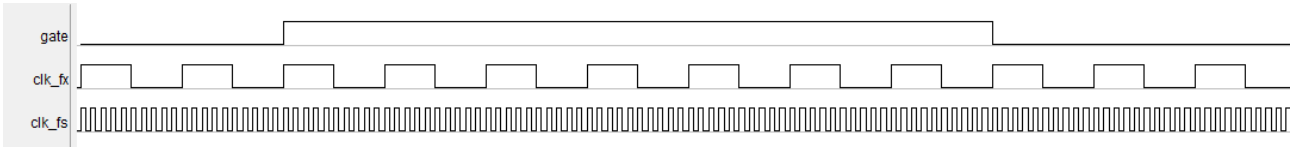


图 21.1.1 等精度测量原理

从以上叙述的等精度的测量原理可以很容易得出如下结论：首先，被测信号频率 clk_fx 的相对误差与被测信号的频率无关；其次，增大测量时间段“软件闸门”或提高“标频”clk_fs，可以减小相对误差，提高测量精度；最后，由于一般提供基准时钟 clk_fs 的石英晶振稳定性很高，所以基准时钟的相对误差很小，可忽略。假设基准时钟的频率为 100MHz，只要实际闸门时间大于或等于 1s，就可使测量的最大相对误差小于或等于 10^(-8)，即精度达到 1/100MHz。等精度测量的核心思想在于如何保证在实际测量门闸内被测信号为整数个周期，这就需要在设计中让实际测量门闸信号与被测信号建立一定的关系。基于这种思想，设计中以被测信号的上升沿作为开启门闸和关闭门闸的驱动信号，只有在被测信号的上升沿才将图 21.1.1 中预置的“软件闸门”的状态锁存，因此在“实际闸门”Tx 内被测信号的个数就能保证整数个周期，这样就避免普通测量方法中被测信号的±1 的误差，但会产生高频的基准时钟信号的±1 周期误差，由于基准时钟频率远高于被测信号，因此它产生的±1 周期误差对测量精度的影响十分有限，特别是在中低频测量的时候，相较于传统的频率测量和周期测量方法，可以大大提高测量精度。

等精度测频的原理图如图 21.1.2 所示。图中，预置软件闸门信号 GATE 是由 Zynq 的定时模块产生，GATE 的时间宽度对测频精度的影响较小，故可以在较大的范围内选择，GATE 信号经被测时钟 clk_fx 同步化（图中的 D 触发器）到被测时钟域下。另外，为了方便处理，这里选择预置闸门信号的长度由参数 GATE_TIME 设置。图中的 fs_cnt 和 fx_cnt 是 2 个可控的 32 位高速计数器，fs_cnt_en 和 fx_cnt_en 分别是其计数使能端，由同步化后的 GATE 信号控制，基准时钟信号 clk_fs 从时钟输入端 clk_fs 输入，待测信号 clk_fx 从时钟输入端 clk_fx 输入。测量时，生成的 GATE 信号，在被测时钟同步化后用来控制启动和关闭 2 个计数器，2 个计数器分别对被测信号和基准时钟计数。若在一次实际闸门时间 GATE_TIME 中，计数器对被测信号的计数值为 fx_cnt，对基准时钟的计数值为 fs_cnt，而基准时钟的频率为 CLK_FS，则被测信号的频率为 clk_fx，则由公式

$$\frac{fs\_cnt}{\mathrm{CLK\_FS}} = GATE\_TIME = \frac{fx\_cnt}{clk\_fx} \tag{1-1}$$

推出:

$$\mathrm{clk\_fx} = \mathrm{fx\_cnt} \times \frac{\mathrm{CLK\_FS}}{\mathrm{fs\_cnt}} \tag{1-2}$$

图 21.1.2 中的所有功能都在 ZYNQ PL 端实现。

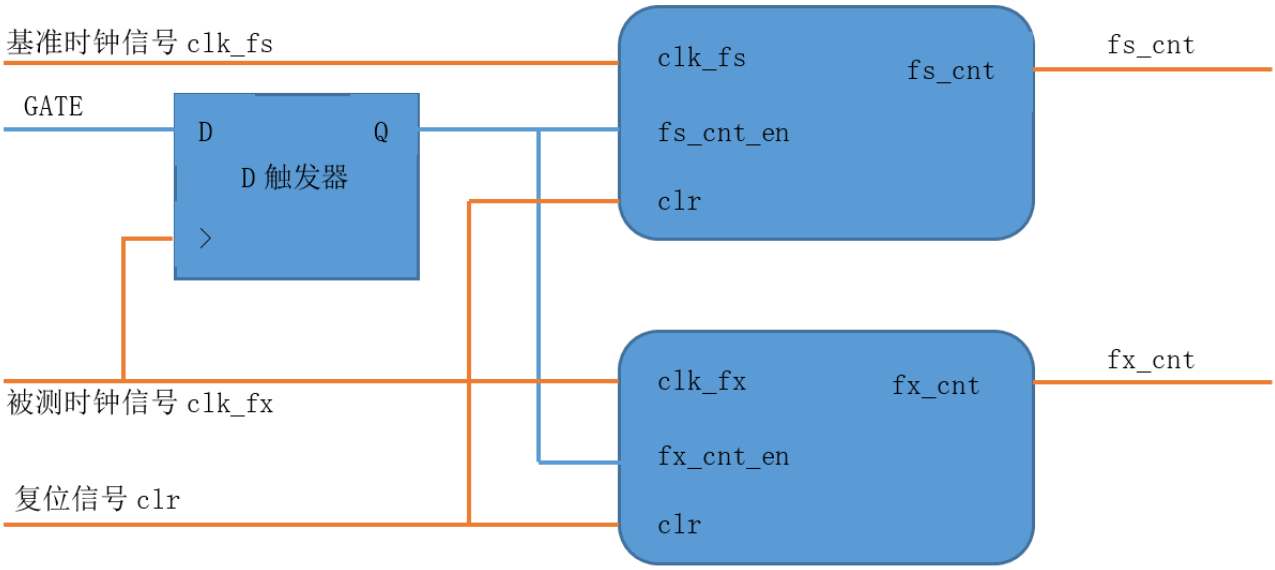


图 21.1.2 Zynq 实现的功能的原理图

21.2 实验任务

板载 50MHz 的时钟通过内部逻辑进行分频，来产生 500KHz 频率的信号，作为被测时钟，然后用 Verilog HDL 编写的等精度测量模块测量被测时钟，并通过 LCD 显示。

21.3 硬件设计

本次实验只需将启明星开发板 J3（靠近 ATK MODULE 侧）扩展口的两个 IO 使用跳帽或者杜邦线连接即可。本次实验将 Zynq 的 D20 引脚做为分频产生的时钟的输出端，J18 引脚（全局差分时钟的 P 端，因为在 Xilinx 7 系列器件中，只有 CC 差分全局时钟对的 P 端才能作为单端的全局时钟输入）作为被测时钟的输入端，通过一根导线（杜邦线）或者跳帽进行连接。

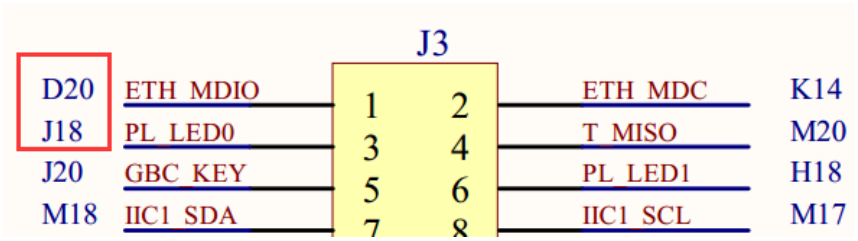


图 21.3.1 硬件原理图

由于端口引脚较多，这里仅给出部分管脚列表，如下表所示：

表 21.3.1 等精度频率计实验管脚分配

信号名	方向	管脚	端口说明	电平标准
sys_clk	input	U18	系统时钟，50M	LVCMOS33
sys_rst_n	input	J15	系统复位，低有效	LVCMOS33
clk_fx	input	J18	被测时钟	LVCMOS33

clk_out	output	D20	输出时钟	LVCMOS33

21.4 程序设计

根据实验任务，我们可以大致规划出系统的控制流程：首先我们设计一个测试时钟模块用于生成被测的时钟，然后用等精度频率计模块测量被测时钟的频率，并将测得的时钟频率值送入 LCD 显示模块进行显示。由此画出系统的功能框图如下所示：

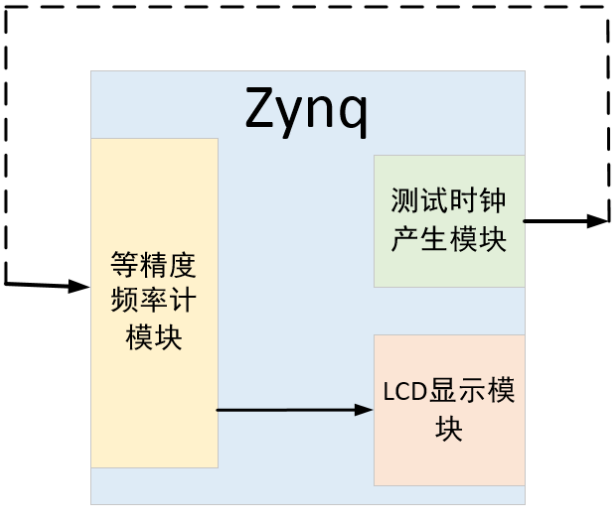


图 21.4.1 等精度频率计实验系统框图

由系统框图可知，FPGA 部分包括四个模块：顶层模块（top_cymometer）、等精度频率计模块（cymometer）、时钟产生模块（clk_test）、以及 LCD 显示模块（lcd_rgb_char）。各模块功能如下：

顶层模块（top_cymometer）：顶层模块完成了对其它三个模块的例化，实现各模块之间的数据交互。时钟产生模块产生被测时钟输出，并从外部接入至等精度频率计模块，以进行频率测量，将测量的结果传输给 LCD 显示模块进行显示。顶层模块的原理图如下图所示：

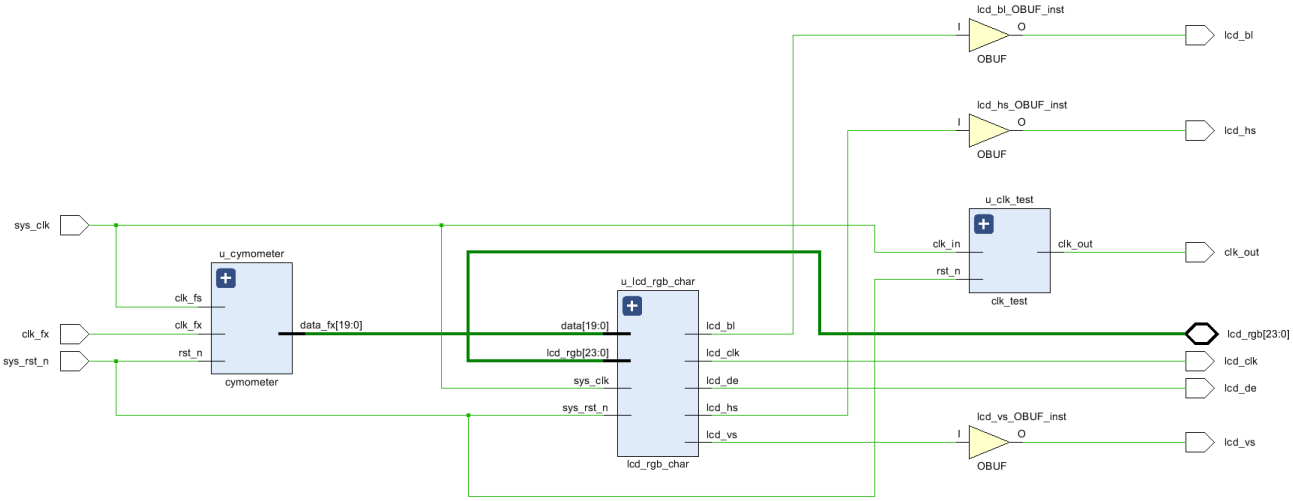


图 21.4.2 顶层模块原理图

等精度频率计模块（cymometer）：等精度频率计模块测量输入的被测时钟的频率。并将测得的频率结果输出。

时钟产生模块（clk_test）：时钟产生模块产生被测的时钟。

LCD 显示模块（lcd_rgb_char）：将等精度频率计测得的时钟频率值在 LCD 上显示出来。

顶层模块的代码如下：

```
module top_cymometer(
    //system clock
    input             sys_clk  ,    // 时钟信号
    input             sys_rst_n,    // 复位信号

    //cymometer interface
    input             clk_fx   ,    // 被测时钟
    output            clk_out  ,    // 输出时钟

    //RGB LCD 接口
    output            lcd_hs   ,    //LCD 行同步信号
    output            lcd_vs   ,    //LCD 场同步信号
    output            lcd_de   ,    //LCD 数据输入使能
    inout      [23:0] lcd_rgb  ,    //LCD RGB565 颜色数据
    output            lcd_bl   ,    //LCD 背光控制信号
    output            lcd_clk       //LCD 采样时钟

);

//parameter define
parameter    CLK_FS = 26'd50000000;  // 基准时钟频率值

//wire define
wire    [19:0]       data_fx;        // 被测信号测量值

//*****************************************************
//**                    main code
//*****************************************************

//例化等精度频率计模块
cymometer #(.CLK_FS(CLK_FS)          // 基准时钟频率值
) u_cymometer(
    //system clock
    .clk_fs      (sys_clk  ),        // 基准时钟信号
    .rst_n       (sys_rst_n),        // 复位信号
    //cymometer interface
    .clk_fx      (clk_fx   ),        // 被测时钟信号
    .data_fx     (data_fx  )         // 被测时钟频率输出
);
```

```
40
41  //例化测试时钟模块，产生测试时钟
42  clk_test #(.DIV_N(7'd100)              // 分频系数
43  ) u_clk_test(
44      //源时钟
45      .clk_in       (sys_clk  ),         // 输入时钟
46      .rst_n        (sys_rst_n),         // 复位信号
47      //分频后的时钟
48      .clk_out      (clk_out  )          // 测试时钟
49  );
50
51  //例化 LCD 显示模块
52  lcd_rgb_char  u_lcd_rgb_char
53  (
54     .sys_clk     (sys_clk),
55     .sys_rst_n   (sys_rst_n),
56     .data        (data_fx),
57     //RGB LCD 接口
58     .lcd_hs      (lcd_hs),         //LCD 行同步信号
59     .lcd_vs      (lcd_vs),         //LCD 场同步信号
60     .lcd_de      (lcd_de),         //LCD 数据输入使能
61     .lcd_rgb     (lcd_rgb),        //LCD RGB565 颜色数据
62     .lcd_bl      (lcd_bl),         //LCD 背光控制信号
63     .lcd_clk     (lcd_clk)         //LCD 采样时钟
64  );
65
66  endmodule
```

顶层代码主要完成对各模块的例化并实现模块信号间的交互。第 21 行的基准时钟频率值参数为基准时钟频率值，当用不同的基准时钟时修改此参数即可。

时钟产生模块的代码如下：

```
1   module clk_test #(parameter DIV_N = 7'd100)     //分频系数
2       (
3        //源时钟
4        input         clk_in     ,                 // 输入时钟
5        input         rst_n      ,                 // 复位信号
6        //分频后的时钟
7        output  reg   clk_out                      // 输出时钟
8   );
9
10  //reg define
11  reg [9:0] cnt;                                  // 时钟分频计数
```

```
12
13  //****************************************************
14  //**                    main code
15  //****************************************************
16
17  //时钟分频，生成 500KHz 的测试时钟
18  always @(posedge clk_in or negedge rst_n) begin
19      if(rst_n == 1'b0) begin
20          cnt     <= 0;
21          clk_out <= 0;
22      end
23      else begin
24          if(cnt == DIV_N/2 - 1'b1) begin
25              cnt     <= 10'd0;
26              clk_out <= ~clk_out;
27          end
28          else
29              cnt <= cnt + 1'b1;
30      end
31  end
32
33  endmodule
```

时钟产生模块通过分频产生被测时钟，这里是用偶数分频方法产生，修改代码第一行的 DIV_N 分频参数，可得到不同频率的时钟信号，时钟频率为clk_in/DIV_N 。由于该模块在顶层例化时 clk_in 为系统时钟 50MHz，分频参数为 100，产生的时钟频率为 50000000/100 = 500000Hz。

等精度频率计模块的代码如下：

```
1   module cymometer
2      #(parameter    CLK_FS = 26'd50_000_000) // 基准时钟频率值
3       (   //system clock
4           input                clk_fs ,     // 基准时钟信号
5           input                rst_n  ,     // 复位信号
6
7           //cymometer interface
8           input                clk_fx ,     // 被测时钟信号
9           output   reg [19:0]  data_fx      // 被测时钟频率输出
10  );
11
12  //parameter define
13  localparam   MAX       =  6'd32;          // 定义 fs_cnt、fx_cnt 的最大位宽
14  localparam   GATE_TIME = 16'd5_000;       // 门控时间设置
15
```

```
16  //reg define
17  reg                 gate        ;           // 门控信号
18  reg                 gate_fs     ;           // 同步到基准时钟的门控信号
19  reg                 gate_fs_r   ;           // 用于同步 gate 信号的寄存器
20  reg                 gate_fs_d0  ;           // 用于采集基准时钟下 gate 下降沿
21  reg                 gate_fs_d1  ;           //
22  reg                 gate_fx_d0  ;           // 用于采集被测时钟下 gate 下降沿
23  reg                 gate_fx_d1  ;           //
24  reg    [   63:0]    data_fx_t    ;          //
25  reg    [   15:0]    gate_cnt    ;           // 门控计数
26  reg    [MAX-1:0]    fs_cnt      ;           // 门控时间内基准时钟的计数值
27  reg    [MAX-1:0]    fs_cnt_temp ;           // fs_cnt 临时值
28  reg    [MAX-1:0]    fx_cnt      ;           // 门控时间内被测时钟的计数值
29  reg    [MAX-1:0]    fx_cnt_temp ;           // fx_cnt 临时值
30
31  //wire define
32  wire                neg_gate_fs;            // 基准时钟下门控信号下降沿
33  wire                neg_gate_fx;            // 被测时钟下门控信号下降沿
34
35  //*****************************************************
36  //**                    main code
37  //*****************************************************
38
39  //边沿检测，捕获信号下降沿
40  assign neg_gate_fs = gate_fs_d1 & (~gate_fs_d0);
41  assign neg_gate_fx = gate_fx_d1 & (~gate_fx_d0);
42
43  //门控信号计数器，使用被测时钟计数
44  always @(posedge clk_fx or negedge rst_n) begin
45      if(!rst_n)
46          gate_cnt <= 16'd0;
47      else if(gate_cnt == GATE_TIME + 5'd20)
48          gate_cnt <= 16'd0;
49      else
50          gate_cnt <= gate_cnt + 1'b1;
51  end
52
53  //门控信号，拉高时间为 GATE_TIME 个实测时钟周期
54  always @(posedge clk_fx or negedge rst_n) begin
55      if(!rst_n)
56          gate <= 1'b0;
```

```
57      else if(gate_cnt < 4'd10)
58          gate <= 1'b0;
59      else if(gate_cnt < GATE_TIME + 4'd10)
60          gate <= 1'b1;
61      else if(gate_cnt <= GATE_TIME + 5'd20)
62          gate <= 1'b0;
63      else
64          gate <= 1'b0;
65  end
66
67  //将门控信号同步到基准时钟下
68  always @(posedge clk_fs or negedge rst_n) begin
69      if(!rst_n) begin
70          gate_fs_r <= 1'b0;
71          gate_fs   <= 1'b0;
72      end
73      else begin
74          gate_fs_r <= gate;
75          gate_fs   <= gate_fs_r;
76      end
77  end
78
79  //打拍采门控信号的下降沿（被测时钟下）
80  always @(posedge clk_fx or negedge rst_n) begin
81      if(!rst_n) begin
82          gate_fx_d0 <= 1'b0;
83          gate_fx_d1 <= 1'b0;
84      end
85      else begin
86          gate_fx_d0 <= gate;
87          gate_fx_d1 <= gate_fx_d0;
88      end
89  end
90
91  //打拍采门控信号的下降沿（基准时钟下）
92  always @(posedge clk_fs or negedge rst_n) begin
93      if(!rst_n) begin
94          gate_fs_d0 <= 1'b0;
95          gate_fs_d1 <= 1'b0;
96      end
97      else begin
```

```
98          gate_fs_d0 <= gate_fs;
99          gate_fs_d1 <= gate_fs_d0;
100     end
101 end
102
103  //门控时间内对被测时钟计数
104 always @(posedge clk_fx or negedge rst_n) begin
105     if(!rst_n) begin
106         fx_cnt_temp <= 32'd0;
107         fx_cnt <= 32'd0;
108     end
109     else if(gate)
110         fx_cnt_temp <= fx_cnt_temp + 1'b1;
111     else if(neg_gate_fx) begin
112         fx_cnt_temp <= 32'd0;
113         fx_cnt   <= fx_cnt_temp;
114     end
115 end
116
117 //门控时间内对基准时钟计数
118 always @(posedge clk_fs or negedge rst_n) begin
119     if(!rst_n) begin
120         fs_cnt_temp <= 32'd0;
121         fs_cnt <= 32'd0;
122     end
123     else if(gate_fs)
124         fs_cnt_temp <= fs_cnt_temp + 1'b1;
125     else if(neg_gate_fs) begin
126         fs_cnt_temp <= 32'd0;
127         fs_cnt <= fs_cnt_temp;
128     end
129 end
130
131 //计算被测信号频率
132 always @(posedge clk_fs or negedge rst_n) begin
133     if(!rst_n) begin
134         data_fx_t <= 64'd0;
135     end
136     else if(gate_fs == 1'b0)
137         data_fx_t <= CLK_FS * fx_cnt ;
138 end
```

```
139
140 always @(posedge clk_fs or negedge rst_n) begin
141     if(!rst_n) begin
142         data_fx <= 20'd0;
143     end
144     else if(gate_fs == 1'b0)
145         data_fx <= data_fx_t / fs_cnt ;
146 end
147
148 endmodule
```

在前面的等精度频率计简介中，我们知道在等精度测量中需要一个闸门信号（门控信号），并且该闸门信号需要同步化到被测时钟域下。这里我们为了方便处理，用被测时钟控制闸门信号的产生，这样就避免了同步化处理，当然了，完全可以用基准时钟控制闸门信号的产生，不过这时产生的闸门信号我们需要同步化到被测时钟域下，这样做的目的是为了不让被测时钟计数产生±1周期的误差。门控时间由参数 GATE_TIME 设置，此处设为 5000，需要说明的是该值越大测得的被测时钟频率值越精确，但测量时间也会相应的变慢一些。另外因为闸门信号是由被测时钟产生的，当测量频率较高的信号或者说信号频率大于 10KHz（此值跟门控时间有关）时是不会有什么问题的，但当测量低频信号像 Hz 级这种，如果门控时间设置的大的话，测量时间就会非常长，此时可修改门控时间的值，为被测时钟频率的 5~10 倍即可，对于几十 KHz 及以上的时钟信号，门控时间的大小对测量速度的影响较小，频率越高影响越小，但对测量精度影响较大，因而在测量频率较高的信号时，建议增大门控时间。

代码中为了防止复位对测量造成的干扰，门控信号在复位后延迟了 10 个被测信号的周期（第 54~57 行）。另外计算被测信号频率是在基准时钟下的门控信号为低电平时进行。

建立了门控信号之后，我们需要通过门控信号分别使能基准时钟和被测时钟的计数。因为门控信号对基准时钟而言是异步信号，所以这里我们对门控信号进行了两次打拍处理得到基准频率下的门控信号 gate_fs（代码第 67 行的 always 语句块）。在门控信号的下降沿将计数值寄存并清零计数寄存器。

在取得数值后，我们需要计算被测信号的频率值，由于在计算周期内数值已不再发生变化（计数值已寄存），而且保留了足够的计算时间，所以可以不用 FIFO 进行异步处理。计算完之后，把所得的结果赋给寄存器变量 data_fx，其数值的单位为 Hz。

LCD 显示部分的代码和“RGB TFT-LCD 字符和图片显示”实验的代码基本是一模一样的，唯一不同的地方在于 lcd_display 模块。

lcd_display 模块的代码如下所示：

```
1   module lcd_display(
2       input               lcd_clk,            //lcd 驱动时钟
3       input               sys_rst_n,          //复位信号
4
5       input       [19:0] data ,
6
7       input       [10:0] pixel_xpos,          //像素点横坐标
8       input       [10:0] pixel_ypos,          //像素点纵坐标
9       output reg  [23:0] pixel_data           //像素点数据,
10  );
```

```
11
12  //parameter define
13  localparam CHAR_POS_X  = 11'd1;                //字符区域起始点横坐标
14  localparam CHAR_POS_Y  = 11'd1;                //字符区域起始点纵坐标
15  localparam CHAR_WIDTH  = 11'd64;               //字符区域宽度
16  localparam CHAR_HEIGHT = 11'd16;               //字符区域高度
17
18  localparam WHITE  = 24'b11111111_11111111_11111111;   //背景色，白色
19  localparam BLACK  = 24'b00000000_00000000_00000000;   //字符颜色，黑色
20
21  //reg define
22  reg  [127:0]  char  [11:0] ;                   //字符数组
23
24  //wire define
25  wire   [3:0]              data0    ;          // 十万位数
26  wire   [3:0]              data1    ;          // 万位数
27  wire   [3:0]              data2    ;          // 千位数
28  wire   [3:0]              data3    ;          // 百位数
29  wire   [3:0]              data4    ;          // 十位数
30  wire   [3:0]              data5    ;          // 个位数
31
32  //*****************************************************
33  //**                    main code
34  //*****************************************************
35
36  assign  data5 = data / 17'd100000;             // 十万位数
37  assign  data4 = data / 14'd10000 % 4'd10;      // 万位数
38  assign  data3 = data / 10'd1000 % 4'd10 ;      // 千位数
39  assign  data2 = data /  7'd100 % 4'd10  ;      // 百位数
40  assign  data1 = data /  4'd10 % 4'd10   ;      // 十位数
41  assign  data0 = data %  4'd10;                 // 个位数
42
43   //给字符数组赋值，用于存储字模数据
44  always @(posedge lcd_clk) begin
45      char[0 ]  <= 128'h00000018244242424242424224180000 ; // "0"
46      char[1 ]  <= 128'h0000001070101010101010107C0000 ; // "1"
47      char[2 ]  <= 128'h0000003C4242420404081020427E0000 ; // "2"
48      char[3 ]  <= 128'h0000003C4242041804020242244380000 ; // "3"
49      char[4 ]  <= 128'h000000040C14242444447E04041E0000 ; // "4"
50      char[5 ]  <= 128'h0000007E4040405864020242244380000 ; // "5"
51      char[6 ]  <= 128'h0000001C2440405864424242224180000 ; // "6"
```

```
52      char[7 ]  <= 128'h0000007E444408081010101010100000 ; // "7"
53      char[8 ]  <= 128'h0000003C424242241824424242423C0000 ; // "8"
54      char[9 ]  <= 128'h000000182442424242261A020224380000 ; // "9"
55      char[10]  <= 128'h000000E7424242427E42424242E70000 ; // "H"
56      char[11]  <= 128'h00000000000000E44081010227E0000 ; // "z"
57  end
58
59  //给不同的区域赋值不同的像素数据
60  always @(posedge lcd_clk ) begin
61      if (!sys_rst_n)  begin
62          pixel_data <= BLACK;
63      end
64      else if(     (pixel_xpos >= CHAR_POS_X)
65           && (pixel_xpos < CHAR_POS_X + CHAR_WIDTH/8*1)
66           && (pixel_ypos >= CHAR_POS_Y)
67           && (pixel_ypos < CHAR_POS_Y + CHAR_HEIGHT)  ) begin
68         if(char [data5] [ (CHAR_HEIGHT+CHAR_POS_Y - pixel_ypos)*8
69                         - ((pixel_xpos-CHAR_POS_X)%8) -1 ] )
70             pixel_data <= BLACK;        //显示字符为黑色
71         else
72             pixel_data <= WHITE;        //显示字符区域背景为白色
73     end
74     else if(     (pixel_xpos >= CHAR_POS_X + CHAR_WIDTH/8*1)
75           && (pixel_xpos < CHAR_POS_X + CHAR_WIDTH/8*2)
76           && (pixel_ypos >= CHAR_POS_Y)
77           && (pixel_ypos < CHAR_POS_Y + CHAR_HEIGHT)  ) begin
78         if(char [data4] [ (CHAR_HEIGHT+CHAR_POS_Y - pixel_ypos)*8
79                         - ((pixel_xpos-CHAR_POS_X)%8) -1 ]  )
80             pixel_data <= BLACK;
81         else
82             pixel_data <= WHITE;
83     end
84     else if(     (pixel_xpos >= CHAR_POS_X + CHAR_WIDTH/8*2)
85           && (pixel_xpos < CHAR_POS_X + CHAR_WIDTH/8*3)
86           && (pixel_ypos >= CHAR_POS_Y)
87           && (pixel_ypos < CHAR_POS_Y + CHAR_HEIGHT)  ) begin
88         if(char [data3] [ (CHAR_HEIGHT+CHAR_POS_Y - pixel_ypos)*8
89                         - ((pixel_xpos-CHAR_POS_X)%8) -1 ]  )
90             pixel_data <= BLACK;
91         else
92             pixel_data <= WHITE;
```

```
93      end
94      else if(     (pixel_xpos >= CHAR_POS_X + CHAR_WIDTH/8*3)
95            && (pixel_xpos < CHAR_POS_X + CHAR_WIDTH/8*4)
96                && (pixel_ypos >= CHAR_POS_Y)
97            && (pixel_ypos < CHAR_POS_Y + CHAR_HEIGHT)  ) begin
98          if(char [data2] [ (CHAR_HEIGHT+CHAR_POS_Y - pixel_ypos)*8
99                          - ((pixel_xpos-CHAR_POS_X)%8) -1 ]  )
100             pixel_data <= BLACK;
101         else
102             pixel_data <= WHITE;
103     end
104     else if(     (pixel_xpos >= CHAR_POS_X + CHAR_WIDTH/8*4)
105           && (pixel_xpos < CHAR_POS_X + CHAR_WIDTH/8*5)
106           && (pixel_ypos >= CHAR_POS_Y)
107           && (pixel_ypos < CHAR_POS_Y + CHAR_HEIGHT)  ) begin
108         if(char [data1] [ (CHAR_HEIGHT+CHAR_POS_Y - pixel_ypos)*8
109                         - ((pixel_xpos-CHAR_POS_X)%8) -1 ]  )
110             pixel_data <= BLACK;
111         else
112             pixel_data <= WHITE;
113     end
114     else if(     (pixel_xpos >= CHAR_POS_X + CHAR_WIDTH/8*5)
115           && (pixel_xpos < CHAR_POS_X + CHAR_WIDTH/8*6)
116               && (pixel_ypos >= CHAR_POS_Y)
117           && (pixel_ypos < CHAR_POS_Y + CHAR_HEIGHT)  ) begin
118         if(char [data0] [ (CHAR_HEIGHT+CHAR_POS_Y - pixel_ypos)*8
119                         - ((pixel_xpos-CHAR_POS_X)%8) -1 ] )
120             pixel_data <= BLACK;
121         else
122             pixel_data <= WHITE;
123     end
124     else if(     (pixel_xpos >= CHAR_POS_X + CHAR_WIDTH/8*6)
125           && (pixel_xpos < CHAR_POS_X + CHAR_WIDTH/8*7)
126               && (pixel_ypos >= CHAR_POS_Y)
127           && (pixel_ypos < CHAR_POS_Y + CHAR_HEIGHT)  ) begin
128         if(char [10]    [ (CHAR_HEIGHT+CHAR_POS_Y - pixel_ypos)*8
129                         - ((pixel_xpos-CHAR_POS_X)%8) -1 ] )
130             pixel_data <= BLACK;
131         else
132             pixel_data <= WHITE;
133     end
```

```
134     else if(     (pixel_xpos >= CHAR_POS_X + CHAR_WIDTH/8*7)
135           && (pixel_xpos < CHAR_POS_X + CHAR_WIDTH)
136               && (pixel_ypos >= CHAR_POS_Y)
137           && (pixel_ypos < CHAR_POS_Y + CHAR_HEIGHT)  ) begin
138         if(char [11]    [ (CHAR_HEIGHT+CHAR_POS_Y - pixel_ypos)*8
139                         - ((pixel_xpos-CHAR_POS_X)%8) -1 ] )
140             pixel_data <= BLACK;
141         else
142             pixel_data <= WHITE;
143     end
144     else begin
145         pixel_data <= WHITE;            //绘制屏幕背景为白色
146     end
147 end
148
149 endmodule
```

lcd_display 模块实现的功能还是根据当前像素点的坐标，送出该像素点应该显示的数据，这里把像素的颜色数据设置为黑色或白色，字符部分为黑色，其他背景色为白色。代码中的第 36-41 行是计算频率值的十进制数的各个位上的值，它们就是要在 LCD 上显示的 6 个数字。44 行的 always 块用于存储字模数据。最后的第 60 行的 always 块就是主要的逻辑功能，它根据当前像素点坐标所在的字符坐标的范围，来计算当前像素点是否应该显示有效字符，即黑色。

至此，我们的设计部分已基本完成，现在我们来做一下误差分析。

$$\delta = \frac{clk\_fxe - clk\_fx}{clk\_fxe} \times 100\% \tag{1-3}$$

其中 clk_fxe 为被测频率信号的准确值。

在测量中，由于 clk_fx 计数的起停时间都是由该信号的上升沿触发的，在闸门时间 GATE_TIME 内对 clk_fx 的计数 fx_cnt 无误差(GATE_TIME = fx_cnt/clk_fxe)；对 clk_fs 的计数 fs_cnt 最多相差一个时钟的误差，即|Δfs_cnt|≤1,其测量频率如式(1-4):

$$clk\_fxe = \left[\frac{fx\_cnt}{fs\_cnt + \Delta fs\_cnt}\right] \times CLK\_FS \tag{1-4}$$

将式(1-2)和(1-4)代入式(1-3)，并整理如式

$$\delta = \frac{|\Delta fs\_cnt|}{fs\_cnt} \le \frac{1}{fs\_cnt} = \frac{1}{GATE\_TIME \times CLK\_FS} \tag{1-5}$$

由上式可以看出，测量频率的相对误差与被测信号频率的大小无关，仅与闸门时间和基准时钟频率有关，即实现了整个测试频段的等精度测量。闸门时间越长，基准时钟频率越高，测频的相对误差就越小。基准时钟频率可由稳定度好、精度高的高频率晶体振荡器产生，在保证测量精度不变的前提下，提高基准时钟频率，可使闸门时间缩短，即提高测试速度。

21.5 下载验证

编译工程并生成比特流.bit 文件。将 Zynq 的 D20 引脚和 J18 引脚，即扩展口最左下角的两个引脚，用一根杜邦线或者跳帽连接起来，并连接好 LCD 排线的接口。将下载器一端连电脑，另一端与开发板上的 JTAG

端口连接，连接电源线并打开电源开关。

下载完成后 LCD 上面显示“500000Hz”，如下图所示，与时钟产生模块产生的时钟频率一致，等精度频率计实验下载验证成功。

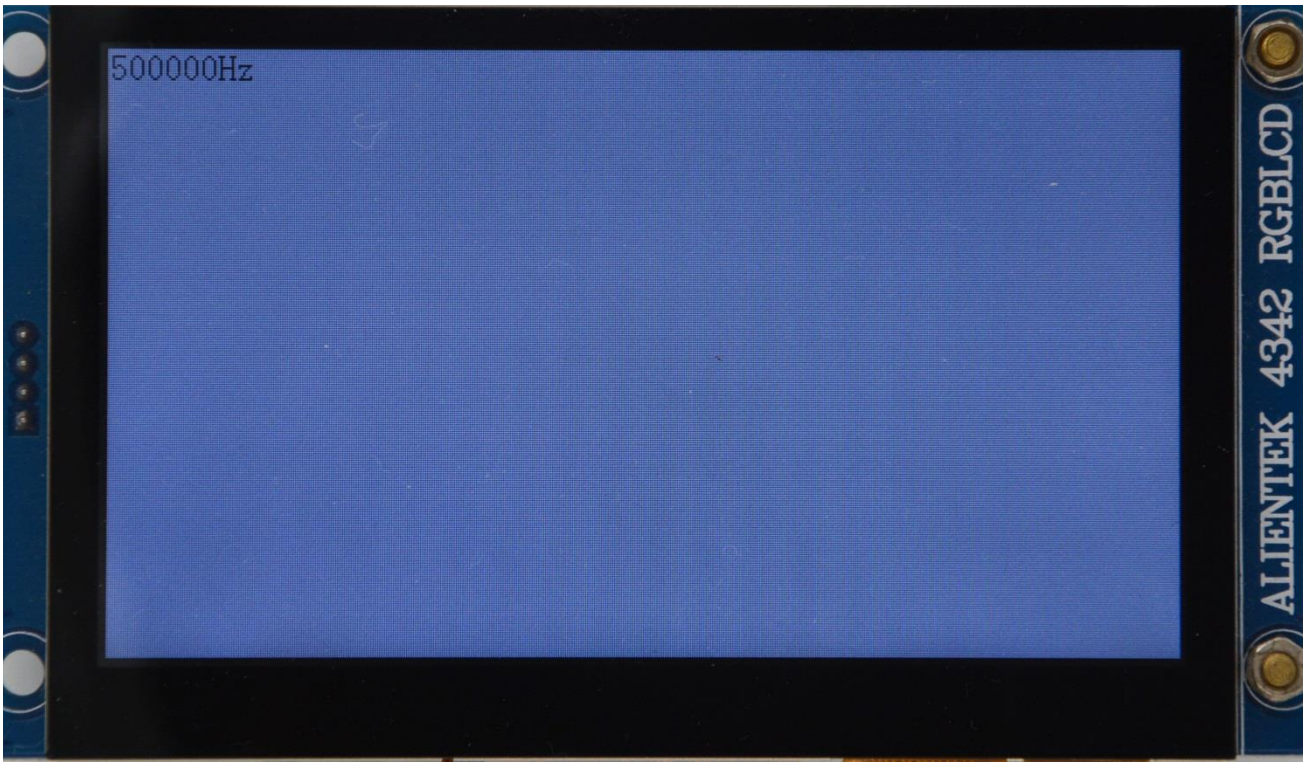


图 21.5.2 实验结果

第二十二章 高速 AD/DA 实验

ADC/DAC（Analog to Digital Converter/ Digital to Analog Converter，即模数转换器/数模转换器）是大多数系统中必不可少的组成部件，用于将连续的模拟信号转换成离散的数字信号，或者将离散的数字信号转换成连续的模拟信号，它们是连接模电电路和数字电路必不可少的桥梁。在很多场合下，ADC/DAC 的转换速度甚至直接决定了整个系统的运行速度。本章我们将使用高速 DA 芯片实现数模转换，产生正弦波模拟电压信号，并通过高速 AD 芯片将模拟信号转换成数字信号。

本章包括以下几个部分：

22.1 高速 AD-DA 简介

22.2 实验任务

22.3 硬件设计

22.4 程序设计

22.5 下载验证

22.1 高速 AD/DA 简介

本章我们使用的 AD-DA 模块是正点原子推出的一款高速模数-数模转换模块（ATK_HS_AD_DA），高速 AD 转换芯片和高速 DA 转换芯片都是由 ADI 公司生产的，分别是 AD9280 和 AD9708。

ATK_HS_AD_DA 模块的硬件结构图如下图所示。

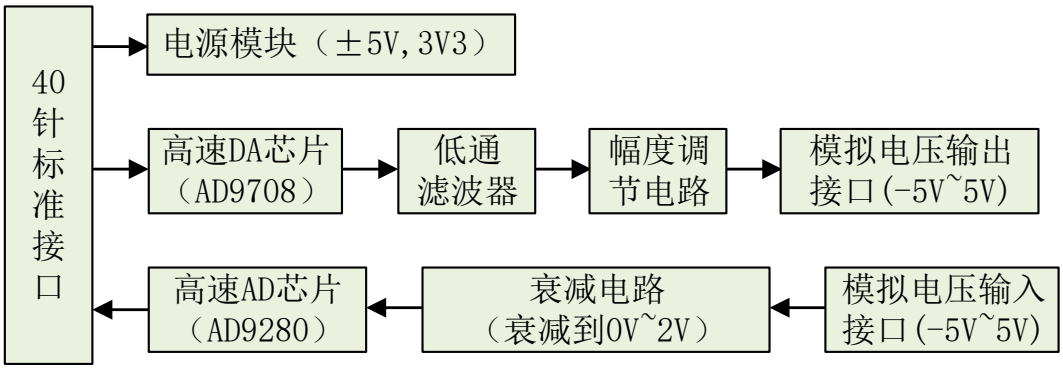


图 22.1.1 ATK_HS_AD_DA 模块硬件结构图

由上可知，AD9708 芯片输出的是一对差分电流信号，为了防止受到噪声干扰，电路中接入了低通滤波器，然后通过高性能和高带宽的运放电路，实现差分变单端以及幅度调节等功能，使整个电路性能得到了最大限度的提升，最终输出的模拟电压范围是-5V~+5V。

AD9280 芯片的输入模拟电压转换范围是 0V~2V，所以电压输入端需要先经过电压衰减电路，使输入的-5V~+5V 之间的电压衰减到 0V~2V 之间，然后经过 AD9280 芯片将模拟电压信号转换成数字信号。

下面我们分别介绍下这两个芯片。

AD9708 芯片

AD9708 是 ADI 公司（Analog Devices，Inc.，亚德诺半导体技术有限公司）生产的 TxDAC 系列数模转换器，具有高性能、低功耗的特点。AD9708 的数模转换位数为 8 位，最大转换速度为 125MSPS（每秒采样百万次 Million Samples per Second）。

AD9708 的内部功能框图如下图所示：

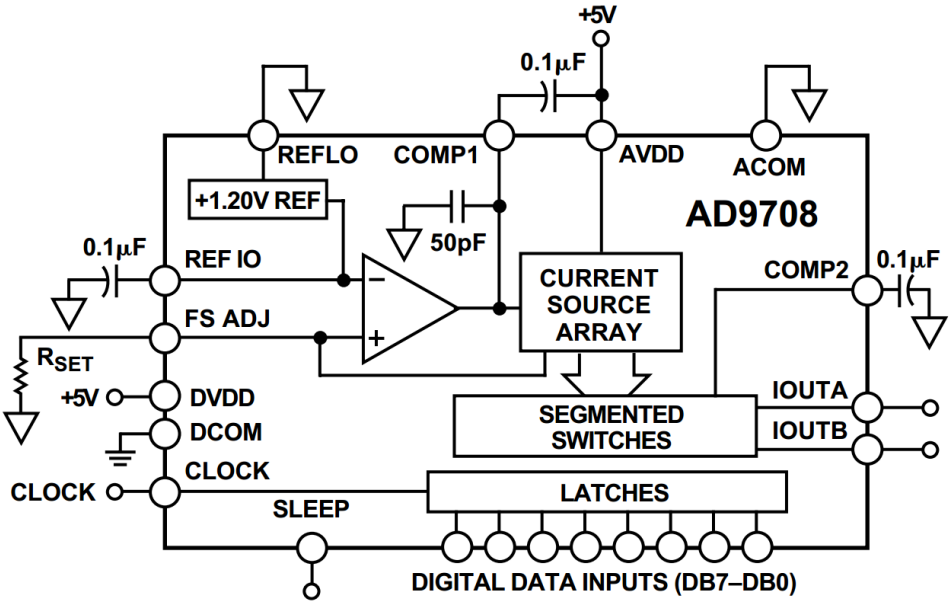


图 22.1.2 AD9708 内部功能框图

AD9708 在时钟（CLOCK）的驱动下工作，内部集成了+1.2V 参考电压(+1.20V REF)、运算放大器、电

流源（CURRENT SOURCE ARRAY）和锁存器（LATCHES）。两个电流输出端 IOUTA 和 IOUTB 为一对差分电流，当输入数据为 0（DB7~DB0=8'h00）时，IOUTA 的输出电流为 0，而 IOUTB 的输出电流达到最大，最大值的大小跟参考电压有关；当输入数据全为高点平（DB7~DB0=8'hff）时，IOUTA 的输出电流达到最大，最大值的大小跟参考电压有关，而 IOUTB 的输出电流为 0。

AD9708 必须在时钟的驱动下才能把数据写入片内的锁存器中，其触发方式为上升沿触发，AD9708 的时序图如下图所示：

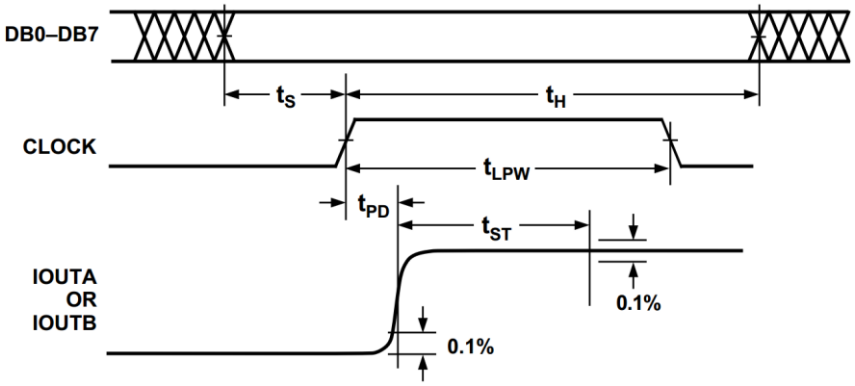


图 22.1.3 AD9708 时序图

上图中的 DBO-DB7 和 CLOCK 是 AD9708 的 8 位输入数据和为输入时钟，IOUTA 和 IOUTB 为 AD9708 输出的电流信号。由上图可知，数据在时钟的上升沿锁存，因此我们可以在时钟的下降沿发送数据。需要注意的是，CLOCK 的时钟频率越快，AD9708 的数模转换速度越快，AD9708 的时钟频率最快为 125Mhz。

IOUTA 和 IOUTB 为 AD9708 输出的一对差分电流信号，通过外部电路低通滤波器与运放电路输出模拟电压信号，电压范围是-5V 至+5V 之间。当输入数据等于 0 时，AD9708 输出的电压值为 5V；当输入数据等于 255（8'hff）时，AD9708 输出的电压值为-5V。

AD9708 是一款数字信号转模拟信号的器件，内部没有集成 DDS（Direct Digital Synthesizer，直接数字式频率合成器）的功能，但是可以通过控制 AD9708 的输入数据，使其模拟 DDS 的功能。例如，我们使用 AD9708 输出一个正弦波模拟电压信号，那么我们只需要将 AD9708 的输入数据按照正弦波的波形变化即可，下图为 AD9708 的输入数据和输出电压值按照正弦波变化的波形图。

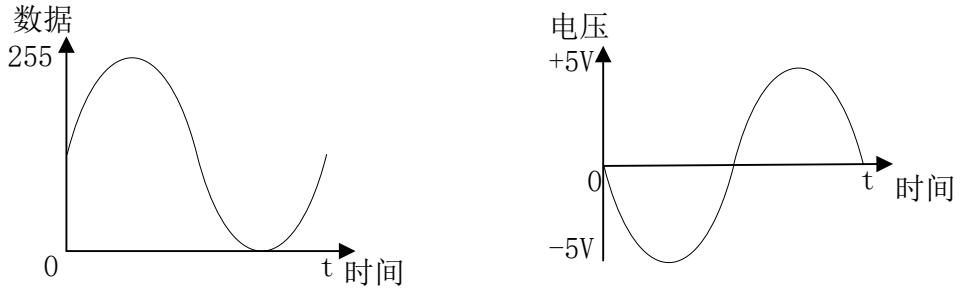


图 22.1.4 AD9708 正弦波数据（左）、电压值（右）

由上图可知，数据在 0 至 255 之间按照正弦波的波形变化，最终得到的电压也会按照正弦波波形变化，当输入数据重复按照正弦波的波形数据变化时，那么 AD9708 就可以持续不断的输出正弦波的模拟电压波形。需要注意的是，最终得到的 AD9708 的输出电压变化范围由其外部电路决定的，当输入数据为 0 时，AD9708 输出+5V 的电压；当输入数据为 255 时，AD9708 输出-5V 的电压。

由此可以看出，只要输入的数据控制的得当，AD9708 可以输出任意波形的模拟电压信号，包括正弦波、方波、锯齿波、三角波等波形。

在了解完高速 DA 转换芯片后，接下来我们介绍下高速 AD 转换芯片 AD9280。

AD9280 芯片

AD9280 是 ADI 公司生产的一款单芯片、8 位、32MSPS（Million Samples Per Second，每秒采样百万次）模数转换器，具有高性能、低功耗的特点。

AD9280 的内部功能框图如下图所示：

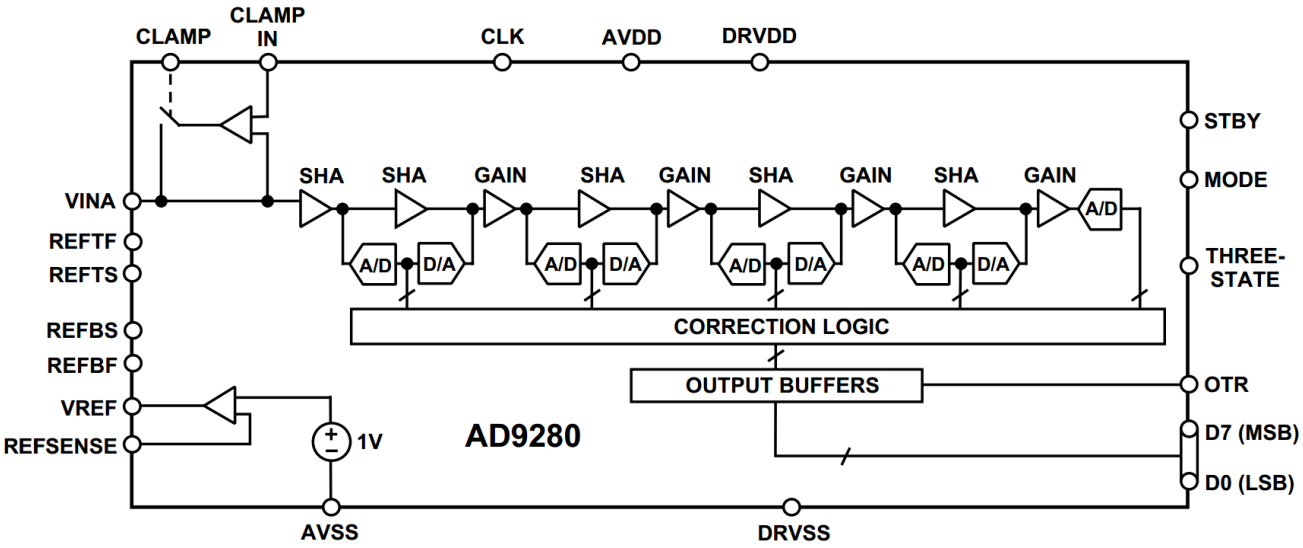


图 22.1.5 AD9280 内部功能框图

AD9280 在时钟（CLK）的驱动下工作，用于控制所有内部转换的周期；AD9280 内置片内采样保持放大器（SHA），同时采用多级差分流水线架构，保证了 32MSPS 的数据转换速率下全温度范围内无失码；AD9280 内部集成了可编程的基准源，根据系统需要也可以选择外部高精度基准满足系统的要求。

AD9280 输出的数据以二进制格式表示，当输入的模拟电压超出量程时，会拉高 OTR（out-of-range）信号；当输入的模拟电压在量程范围内时，OTR 信号为低电平，因此可以通过 OTR 信号来判断输入的模拟电压是否在测量范围内。

AD9280 的时序图如下图所示：

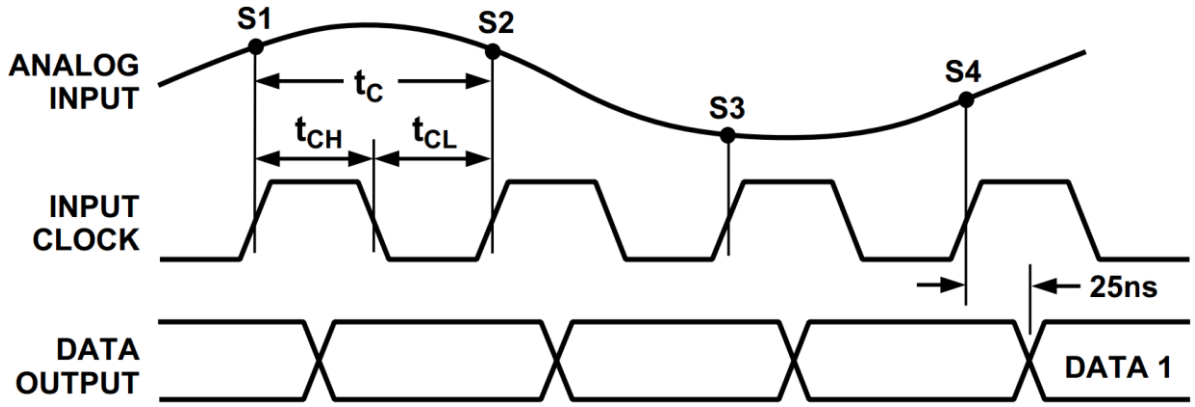


图 22.1.6 AD9280 时序图

模拟信号转换成数字信号并不是当前周期就能转换完成，从采集模拟信号开始到输出数据需要经过 3 个时钟周期。比如上图中在时钟 CLK 的上升沿沿采集的模拟电压信号 S1，经过 3 个时钟周期后（实际上再加上 25ns 的时间延时），输出转换后的数据 DATA1。需要注意的是，AD9280 芯片的最大转换速度是 32MSPS，即输入的时钟最大频率为 32MHz。

AD9280 支持输入的模拟电压范围是 0V 至 2V，0V 对应输出的数字信号为 0，2V 对应输出的数字信号

为 255。而 AD9708 经外部电路后，输出的电压范围是-5V~+5V，因此在 AD9280 的模拟输入端增加电压衰减电路，使-5V~+5V 之间的电压转换成 0V 至 2V 之间。那么实际上对我们用户使用来说，当 AD9280 的模拟输入接口连接-5V 电压时，AD 输出的数据为 0；当 AD9280 的模拟输入接口连接+5V 电压时，AD 输出的数据为 255。

当 AD9280 模拟输入端接-5V 至+5V 之间变化的正弦波电压信号时，其转换后的数据也是成正弦波波形变化，转换波形如下图所示：

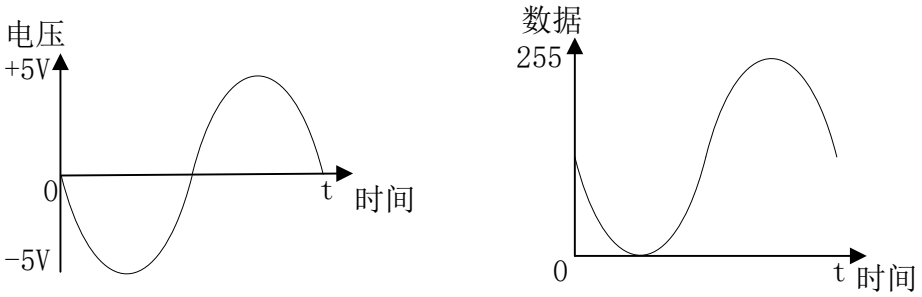


图 22.1.7 AD9280 正弦波模拟电压值（左）、数据（右）

由上图可知，输入的模拟电压范围在-5V 至 5V 之间，按照正弦波波形变化，最终得到的数据也是按照正弦波波形变化。

22.2 实验任务

本节实验任务是使用启明星 ZYNQ 开发板及高速 AD-DA 扩展模块（ATK_HS_AD_DA 模块）实现数模及模数的转换。首先 ZYNQ PL 产生正弦波变化的数字信号，经过 DA 芯片后转换成模拟信号，将 DA 的模拟电压输出端连接至 AD 的模拟电压输入端，AD 芯片将模拟信号转换成数字信号，然后通过 ILA 观察数字信号的波形是否按照正弦波波形变化。

22.3 硬件设计

ATK_HS_AD_DA 模块由 DA 转换芯片（AD9708）和 AD 转换芯片（AD9280）组成。AD9708 的原理图如下图所示。

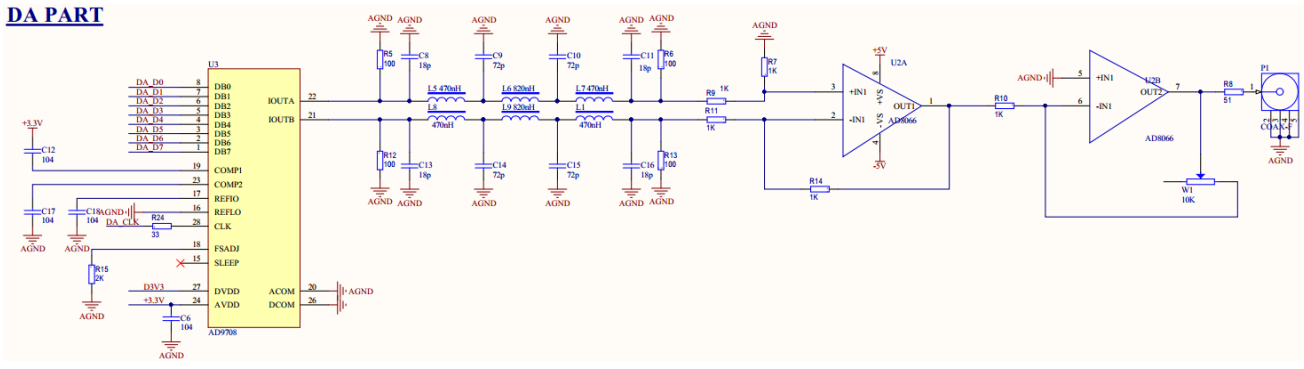


图 22.3.1 AD9708 原理图

由上图可知，AD9708 输出的一对差分电流信号先经过滤波器，再经过运放电路得到一个单端的模拟电压信号。图中右侧的 W1 为滑动变阻器，可以调节输出的电压范围，推荐通过调节滑动变阻器，使输出的电压范围在-5V 至+5V 之间，从而达到 AD 转换芯片的最大转换范围。

AD9280 的原理图如下图所示。

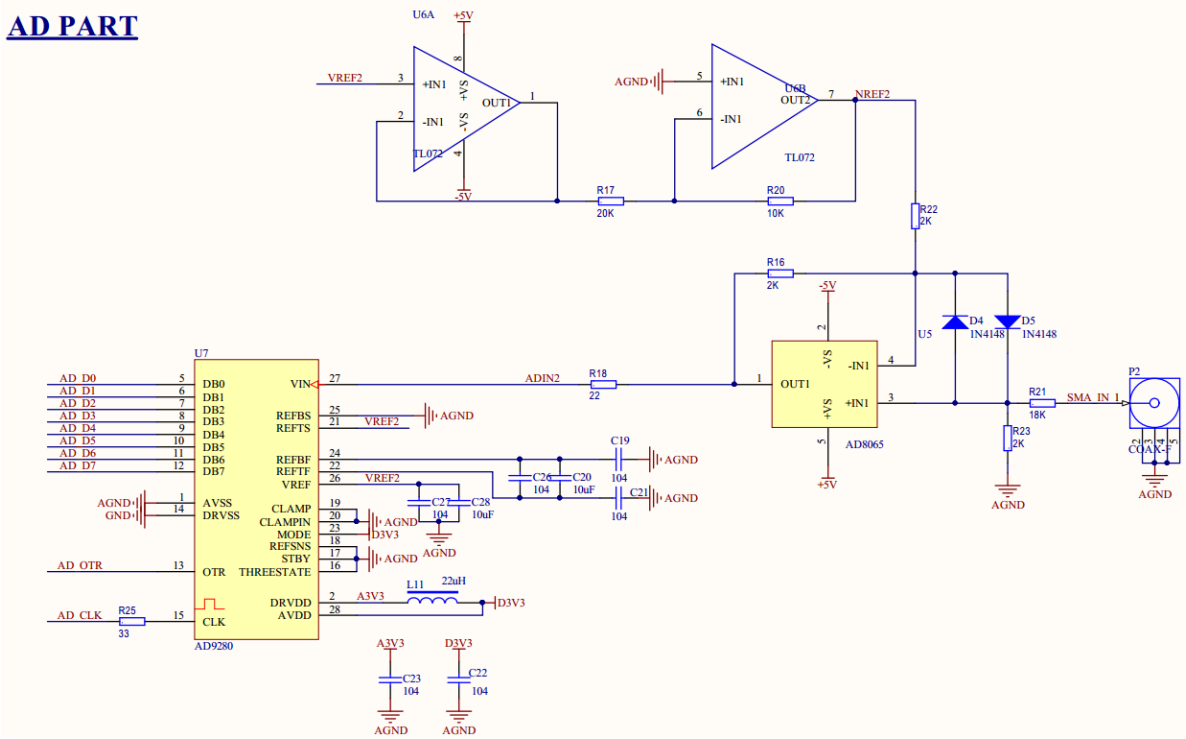


图 22.3.2 AD9280 原理图

上图中输入的模拟信号 SMA_IN（VI）经过衰减电路后得到 AD_IN2（VO）信号，两个模拟电压信号之间的关系是 VO=VI/5+1，即当 VI=5V 时，VO=2V；VI=-5V 时，VO=0V。

ATK_HS_AD_DA 模块的实物图如下图所示。

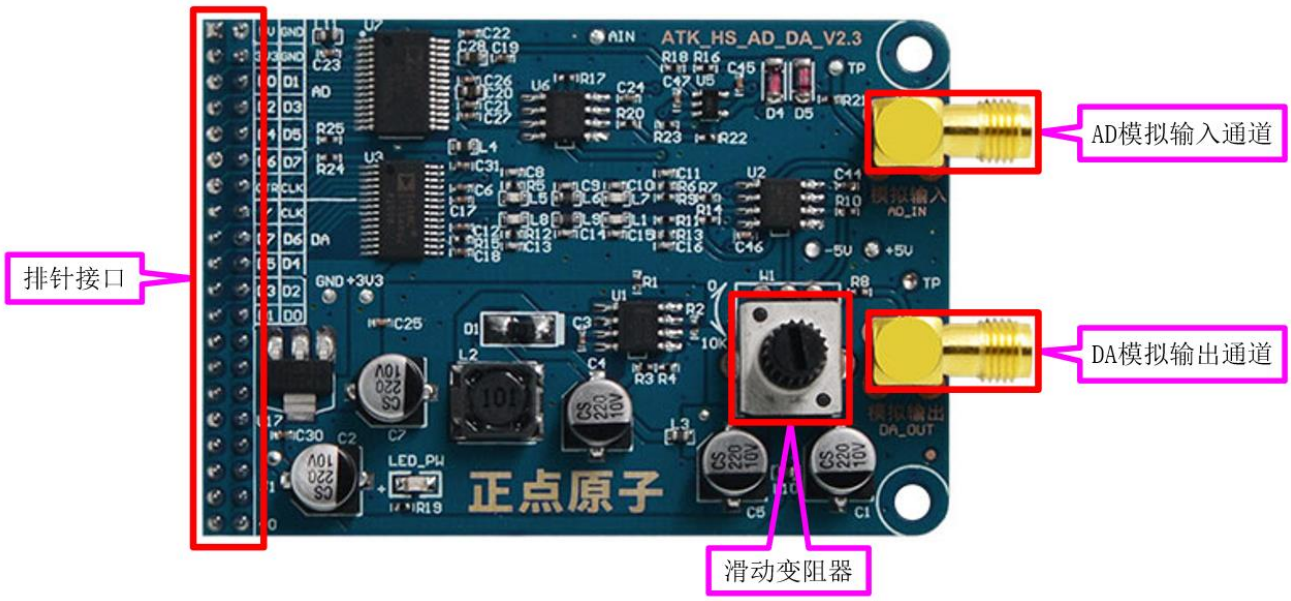


图 22.3.3 ATK-HS-AD-DA 模块实物图

本实验中，各端口信号的管脚分配如下表所示。

表 22.3.1 高速AD-DA转换实验管脚分配

信号名	方向	管脚	端口说明	电平标准

sys_clk	input	U18	系统时钟，50Mhz	LVCMOS33
sys_rst_n	input	J15	系统复位，低有效	LVCMOS33
da_clk	output	K18	DA（AD9708）驱动时钟	LVCMOS33
da_data[0]	output	P15	输出给DA的数据	LVCMOS33
da_data[1]	output	P16	输出给DA的数据	LVCMOS33
da_data[2]	output	T17	输出给DA的数据	LVCMOS33
da_data[3]	output	N17	输出给DA的数据	LVCMOS33
da_data[4]	output	R18	输出给DA的数据	LVCMOS33
da_data[5]	output	P18	输出给DA的数据	LVCMOS33
da_data[6]	output	R19	输出给DA的数据	LVCMOS33
da_data[7]	output	T19	输出给DA的数据	LVCMOS33
ad_data[0]	input	R17	AD输入数据	LVCMOS33
ad_data[1]	input	N18	AD输入数据	LVCMOS33
ad_data[2]	input	R16	AD输入数据	LVCMOS33
ad_data[3]	input	P19	AD输入数据	LVCMOS33
ad_data[4]	input	H20	AD输入数据	LVCMOS33
ad_data[5]	input	C20	AD输入数据	LVCMOS33
ad_data[6]	input	B20	AD输入数据	LVCMOS33
ad_data[7]	input	B19	AD输入数据	LVCMOS33
ad_otr	input	K16	模拟电压超出量程标志	LVCMOS33
ad_clk	output	U19	AD（AD9280）驱动时钟	LVCMOS33

对应的 XDC 约束语句如下所示：

```
create_clock -period 20.000 -name sys_clk [get_ports sys_clk]

set_property -dict {IOSTANDARD LVCMOS33 PACKAGE_PIN U18} [get_ports sys_clk]
set_property -dict {IOSTANDARD LVCMOS33 PACKAGE_PIN J15} [get_ports sys_rst_n]
set_property -dict {IOSTANDARD LVCMOS33 PACKAGE_PIN T19} [get_ports {da_data[7]}]
set_property -dict {IOSTANDARD LVCMOS33 PACKAGE_PIN R19} [get_ports {da_data[6]}]
set_property -dict {IOSTANDARD LVCMOS33 PACKAGE_PIN P18} [get_ports {da_data[5]}]
set_property -dict {IOSTANDARD LVCMOS33 PACKAGE_PIN R18} [get_ports {da_data[4]}]
set_property -dict {IOSTANDARD LVCMOS33 PACKAGE_PIN N17} [get_ports {da_data[3]}]
set_property -dict {IOSTANDARD LVCMOS33 PACKAGE_PIN T17} [get_ports {da_data[2]}]
```

```
set_property -dict {IOSTANDARD LVCMOS33 PACKAGE_PIN P16} [get_ports {da_data[1]}]
set_property -dict {IOSTANDARD LVCMOS33 PACKAGE_PIN P15} [get_ports {da_data[0]}]
set_property -dict {IOSTANDARD LVCMOS33 PACKAGE_PIN K18} [get_ports da_clk]
set_property -dict {IOSTANDARD LVCMOS33 PACKAGE_PIN B19} [get_ports {ad_data[7]}]
set_property -dict {IOSTANDARD LVCMOS33 PACKAGE_PIN B20} [get_ports {ad_data[6]}]
set_property -dict {IOSTANDARD LVCMOS33 PACKAGE_PIN C20} [get_ports {ad_data[5]}]
set_property -dict {IOSTANDARD LVCMOS33 PACKAGE_PIN H20} [get_ports {ad_data[4]}]
set_property -dict {IOSTANDARD LVCMOS33 PACKAGE_PIN P19} [get_ports {ad_data[3]}]
set_property -dict {IOSTANDARD LVCMOS33 PACKAGE_PIN R16} [get_ports {ad_data[2]}]
set_property -dict {IOSTANDARD LVCMOS33 PACKAGE_PIN N18} [get_ports {ad_data[1]}]
set_property -dict {IOSTANDARD LVCMOS33 PACKAGE_PIN R17} [get_ports {ad_data[0]}]
set_property -dict {IOSTANDARD LVCMOS33 PACKAGE_PIN U19} [get_ports ad_clk]
set_property -dict {IOSTANDARD LVCMOS33 PACKAGE_PIN K16} [get_ports ad_otr]
```

22.4 程序设计

根据本章的实验任务，ZYNQ 需要连续输出正弦波波形的数据，才能使 AD9708 连续输出正弦波波形的模拟电压，如果通过编写代码使用三角函数公式运算的方式输出正弦波数据，那么程序设计会变得非常复杂。在工程应用中，一般将正弦波波形数据存储在 RAM 或者 ROM 中，由于本次实验并不需要写数据到 RAM 中，因此我们将正弦波波形数据存储在只读的 ROM 中，直接读取 ROM 中的数据发送给 DA 转换芯片即可。

图 22.4.1 是根据本章实验任务画出的系统框图。ROM 里面事先存储好了正弦波波形的数据，DA 数据发送模块从 ROM 中读取数据，将数据和时钟送到 AD9708 的输入数据端口和输入时钟端口；AD 数据接收模块给 AD9280 输出驱动时钟信号和使能信号，并采集 AD9280 输出模数转换完成的数据。

高速 AD/DA 实验的系统框图如图 22.4.1 所示：

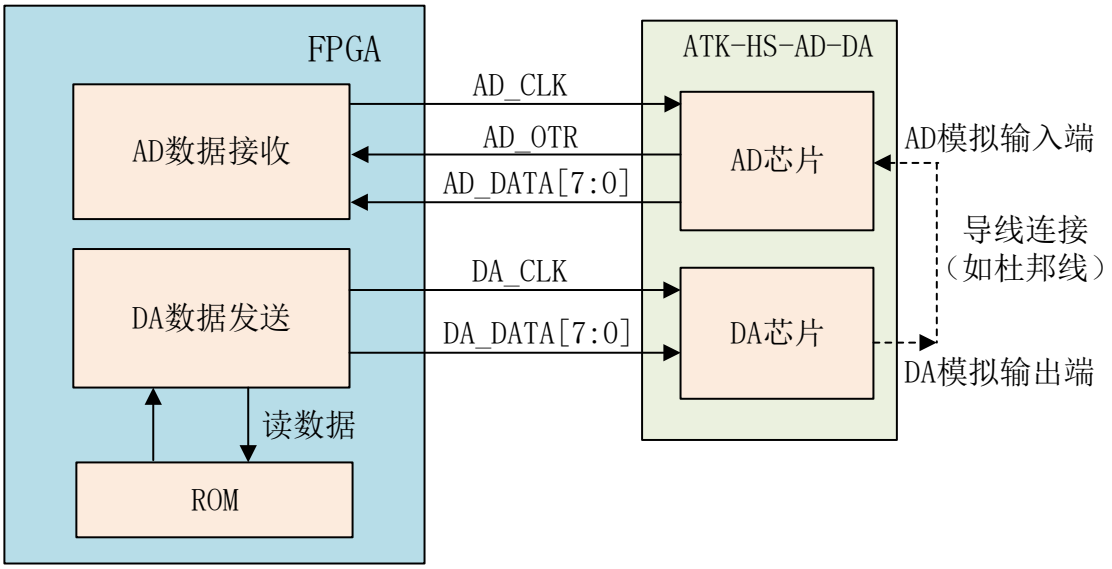


图 22.4.1 高速 AD/DA 系统框图

顶层模块的原理图如下图所示：

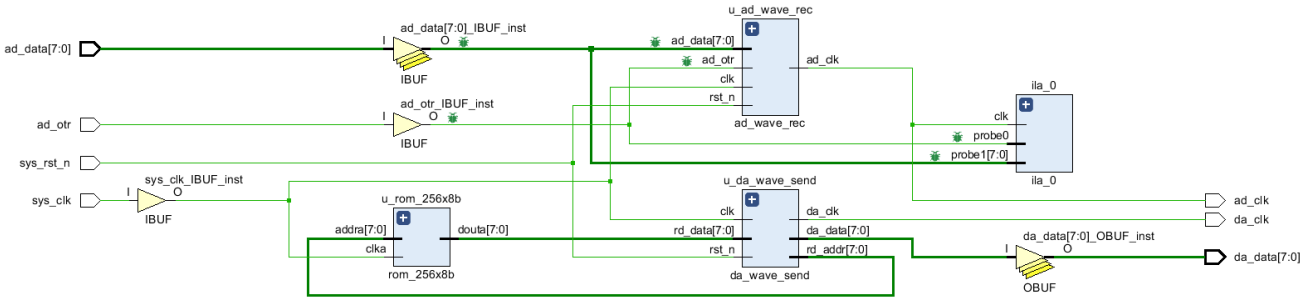


图 22.4.2 顶层模块原理图

FPGA 顶层模块（hs_ad_da）例化了以下三个模块：DA 数据发送模块（da_wave_send）、ROM 波形存储模块（rom_256x8b）和 AD 数据接收模块（ad_wave_rec）。

DA 数据发送模块（da_wave_send）：DA 数据发送模块输出读 ROM 地址，将输入的 ROM 数据发送至 DA 转换芯片的数据端口。

ROM 波形存储模块（rom_256x8b）：ROM 波形存储模块由 Vivado 软件自带的 Block Memory Generator IP 核实现，其存储的波形数据可以使用波形转存储文件的上位机来生成.coe 文件。

AD 数据接收模块（ad_wave_rec）：AD 数据接收模块输出 AD 转换芯片的驱动时钟和使能信号，随后接收 AD 转换完成的数据。

顶层模块的代码如下：

```
1  module hs_ad_da(
2      input                 sys_clk     ,  //系统时钟
3      input                 sys_rst_n   ,  //系统复位，低电平有效
4      //DA 芯片接口
5      output                da_clk      ,  //DA(AD9708)驱动时钟,最大支持 125Mhz 时钟
6      output    [7:0]       da_data     ,  //输出给 DA 的数据
7      //AD 芯片接口
8      input     [7:0]       ad_data     ,  //AD 输入数据
9      //模拟输入电压超出量程标志(本次试验未用到)
10     input                 ad_otr      ,  //0:在量程范围 1:超出量程
11     output                ad_clk         //AD(AD9280)驱动时钟,最大支持 32Mhz 时钟
12 );
13
14 //wire define
15 wire      [7:0]    rd_addr;              //ROM 读地址
16 wire      [7:0]    rd_data;              //ROM 读出的数据
17 //*****************************************************
18 //**                    main code
19 //*****************************************************
20
21 //DA 数据发送
22 da_wave_send u_da_wave_send(
23     .clk          (sys_clk),
24     .rst_n        (sys_rst_n),
```

```
25      .rd_data      (rd_data),
26      .rd_addr      (rd_addr),
27      .da_clk       (da_clk),
28      .da_data      (da_data)
29      );
30
31  //ROM 存储波形
32  rom_256x8b  u_rom_256x8b (
33    .clka  (sys_clk),     // input wire clka
34    .addra (rd_addr),     // input wire [7 : 0] addra
35    .douta (rd_data)      // output wire [7 : 0] douta
36  );
37
38  //AD 数据接收
39  ad_wave_rec u_ad_wave_rec(
40      .clk          (sys_clk),
41      .rst_n        (sys_rst_n),
42      .ad_data      (ad_data),
43      .ad_otr       (ad_otr),
44      .ad_clk       (ad_clk)
45      );
46
47  //ILA 采集 AD 数据
48  ila_0  ila_0 (
49      .clk          (ad_clk ), // input wire clk
50      .probe0       (ad_otr ), // input wire [0:0]  probe0
51      .probe1       (ad_data)  // input wire [7:0]  probe0
52  );
53
54  endmodule
```

DA 数据发送模块输出的读 ROM 地址（rd_addr）连接至 ROM 模块的地址输入端，ROM 模块输出的数据（rd_data）连接至 DA 数据发送模块的数据输入端，从而完成了从 ROM 中读取数据的功能。

在代码的第 32 至 36 行例化了 ROM 模块，由 Block Memory Generator IP 核配置生成。

代码的第 48 行例化了一个 ILA 的 IP 核，用于捕获 ad_otr 和 ad_data 的数据。需要注意的是，ILA 的采样时钟必须使用 ad_clk，否则数据可能采集错误。ILA IP 核的配置如下图所示：

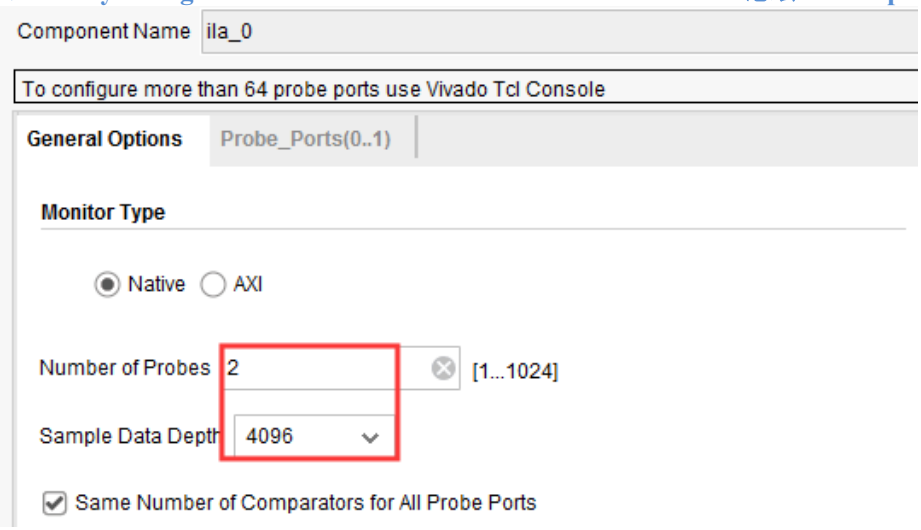


图 22.4.3 ILA IP 核的 Genaral Options 配置

我们把探针数量设置为 2，并且把采样深度设置为 4096。探针宽度的设置如下图所示：

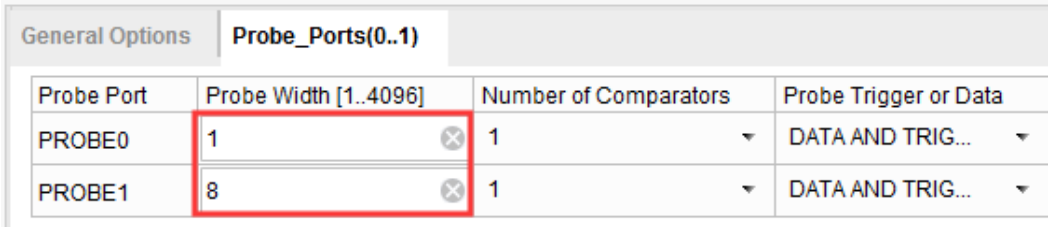


图 22.4.4 ILA IP 核的 Probe_Ports 配置

我们将两个探针的位宽设置成 1 和 8，分别对应 ad_otr 和 ad_data 的位宽，设置完成后点击“OK”按钮即可。

我们在前面说过，ROM 中存储的波形数据可以使用上位机波形转 COE 软件生成，在这里我们介绍一个简单易用的波形转 COE 工具的使用方法，该工具位于开发板所随附的资料“6_软件资料/1_软件/WaveToMem”目录下，双击“WaveToMem_V1.2.exe”运行软件。

接下来我们对软件进行设置，如图 22.4.5 所示，这里对软件界面做个简单的介绍。

位宽：波形数据的位宽。由于 ATK_HS_AD_DA 模块的 DA 芯片数据位宽为 8 位，因此这里保持默认，即设置成 8 位。

深度：一个波形周期包含了多少个数据量。这里保持默认，即设置成 256。需要说明的是，在用 Block Memory Generator IP 核生成 ROM 时，配置 ROM 的宽度和深度和上位机设置的位宽和深度保持一致。

波形频率设置：对波形倍频，倍数值越大，最终生成的波形频率越快（频率太高，可能导致波形失真），这里保持默认，即设置成 1 位。

波形类型：软件支持将正弦波、方波、锯齿波和三角波的波形转换成存储波形格式的文件。

生成文件：软件支持将波形转换成 COE（Vivado 软件支持的存储格式）和 MIF（Quartus 软件支持的存储格式）格式文件，这里保持默认，即选中 COE 文件格式。

然后点击“一键生成”按钮，在弹出的界面中选择 COE 文件的存放路径并输入文件名，这里将 COE 文件保存在工程的 doc 文件夹下。WaveToMem 转换过程中的软件界面如下图所示：

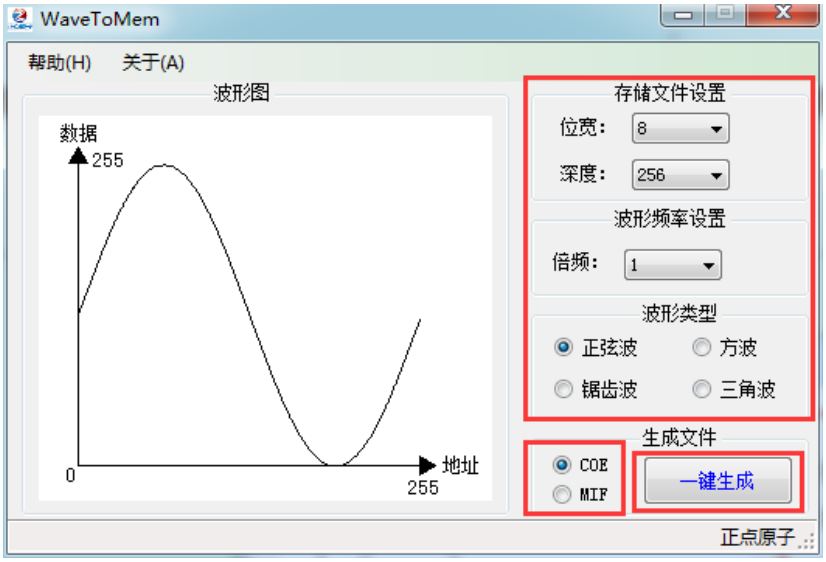


图 22.4.5 WaveToMem 软件界面

使用 Notepad++代码编辑器打开生成的 COE 文件后如下图所示：

```
memory_initialization_radix=16;

memory_initialization_vector=

7F,
82,
85,
88,
8B,
8E,
91,
```

图 22.4.6 COE 文件打开界面

工程中创建了一个单端口 ROM，并命名为“rom_256x8b”，在调用 Block Memory Generator IP 核时，“Basic”选项也配置如下图所示：

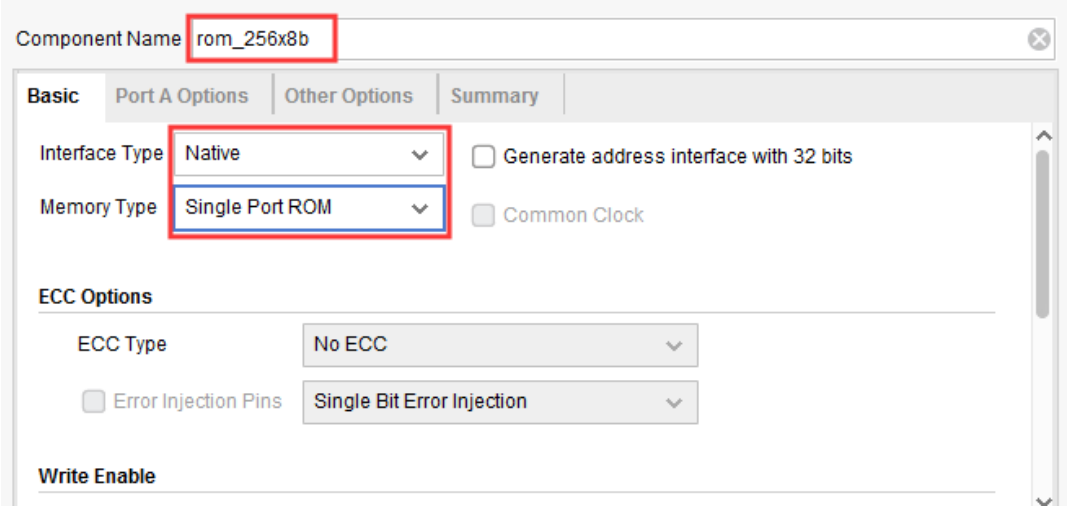


图 22.4.7 Block Memory Generator IP 核的 Basic 配置页面

我们将其接口类型设置为“Native”、Memory Type 设置为“Single Port ROM”，即单端口 ROM。“Port A Options”选项页的配置页面如下图所示：

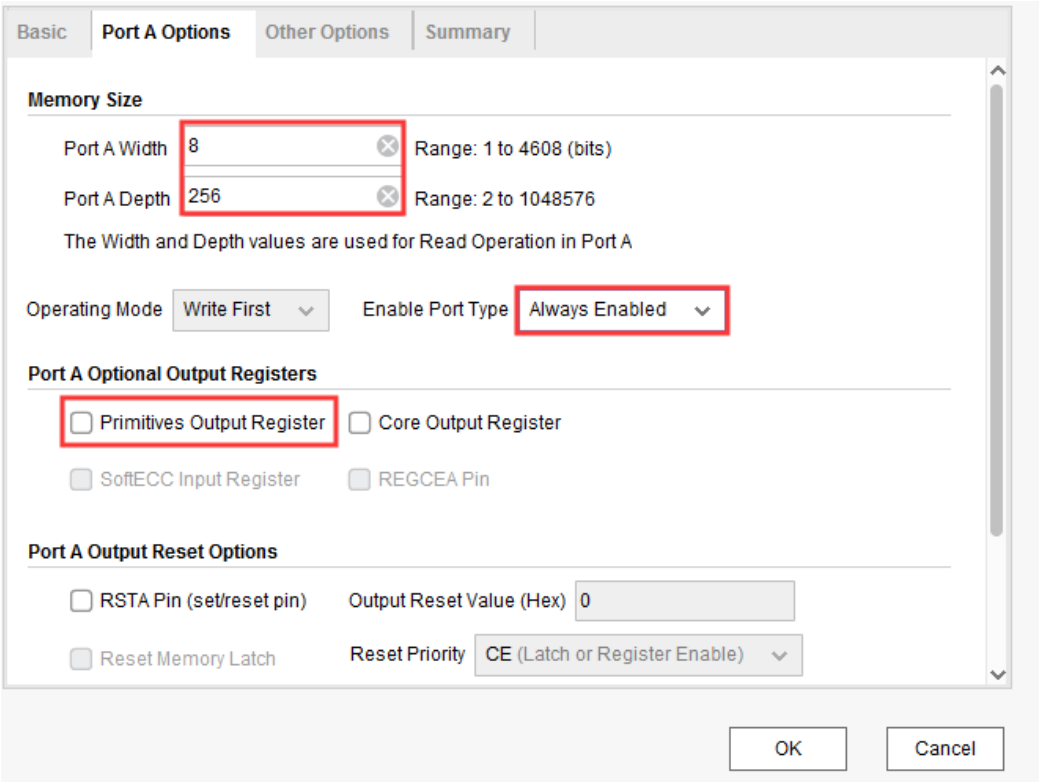


图 22.4.8 Block Memory Generator IP 核的 PortA Options 配置页面

我们将 PortA 的位宽设置为 8，深度设置为 256，以存储上位机生成的 256 个数据。此外，将使能引脚的类型设置为“Always Enabled”，即 ROM 一直处于使能的状态。

接下来配置“Other Options”选项页，加载刚才生成的.coe 文件，如下图所示：

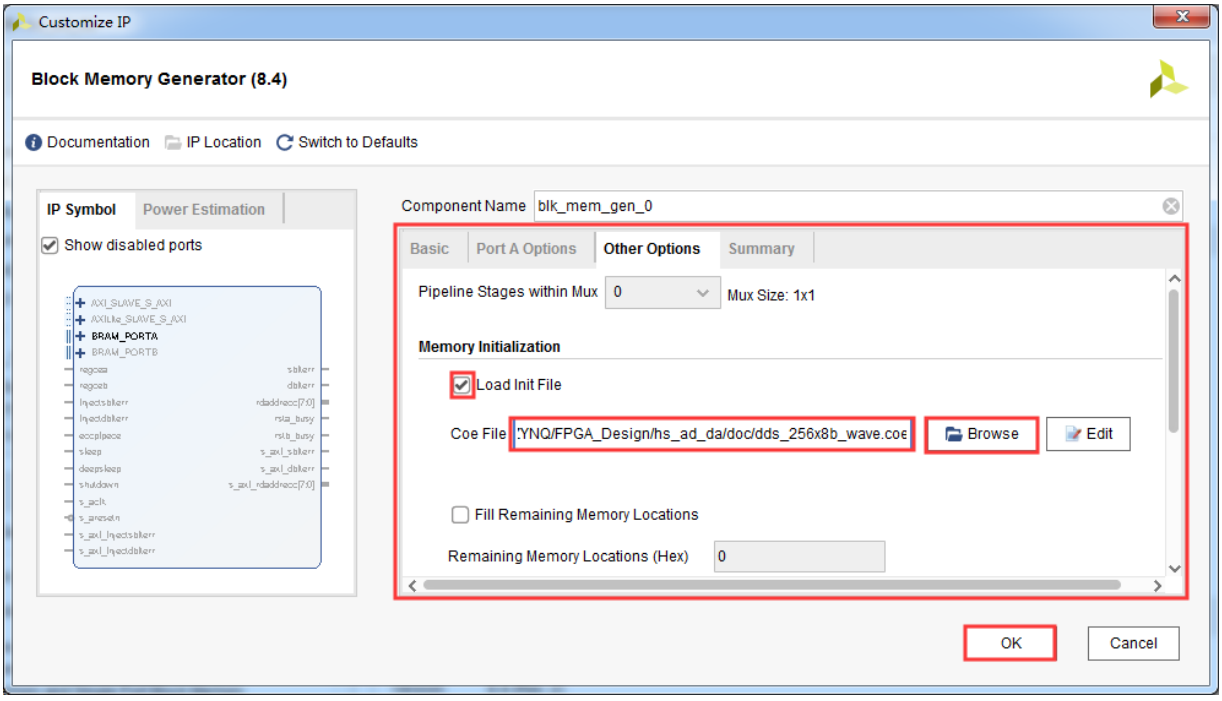


图 22.4.9 Block Memory Generator IP 核的 Other Options 配置页面

最后点击“OK”按钮完成 IP 核的配置。

DA 数据发送模块的代码如下：

```
module da_wave_send(
    input                clk    ,  //时钟
    input                rst_n  ,  //复位信号，低电平有效

    input        [7:0]   rd_data,  //ROM 读出的数据
    output  reg  [7:0]   rd_addr,  //读 ROM 地址
    //DA 芯片接口
    output               da_clk ,  //DA(AD9708)驱动时钟,最大支持 125Mhz 时钟
    output       [7:0]   da_data   //输出给 DA 的数据
    );

//parameter
//频率调节控制
parameter  FREQ_ADJ = 8'd5;  //频率调节,FREQ_ADJ 的越大,最终输出的频率越低,范围 0~255

//reg define
reg    [7:0]    freq_cnt  ;  //频率调节计数器

//*****************************************************
//**                    main code
//*****************************************************

//数据 rd_data 是在 clk 的上升沿更新的，所以 DA 芯片在 clk 的下降沿锁存数据是稳定的时刻
//而 DA 实际上在 da_clk 的上升沿锁存数据,所以时钟取反,这样 clk 的下降沿相当于 da_clk 的上升沿
assign  da_clk = ~clk;
assign  da_data = rd_data;   //将读到的 ROM 数据赋值给 DA 数据端口

//频率调节计数器
always @(posedge clk or negedge rst_n) begin
    if(rst_n == 1'b0)
        freq_cnt <= 8'd0;
    else if(freq_cnt == FREQ_ADJ)
        freq_cnt <= 8'd0;
    else
        freq_cnt <= freq_cnt + 8'd1;
end

//读 ROM 地址
always @(posedge clk or negedge rst_n) begin
```

```
40      if(rst_n == 1'b0)
41          rd_addr <= 8'd0;
42      else begin
43          if(freq_cnt == FREQ_ADJ) begin
44              rd_addr <= rd_addr + 8'd1;
45          end
46      end
47  end
48
49  endmodule
```

在代码的第 14 行定义了一个参数 FREQ_ADJ（频率调节），可以通过控制频率调节参数的大小来控制最终输出正弦波的频率大小，频率调节参数的值越小，正弦波频率越大。频率调节参数调节正弦波频率的方法是通过控制读 ROM 的速度实现的，频率调节参数越小，freq_cnt 计数到频率调节参数值的时间越短，读 ROM 数据的速度越快，那么正弦波输出频率也就越高；反过来，频率调节参数越大，freq_cnt 计数到频率调节参数值的时间越长，读 ROM 数据的速度越慢，那么正弦波输出频率也就越低。由于 freq_cnt 计数器的位宽为 8 位，计数范围是 0~255，所以频率调节参数 FREQ_ADJ 支持的调节范围是 0~255，可通过修改 freq_cnt 计数器的位宽来修改 FREQ_ADJ 支持的调节范围。

WaveToMem 软件设置 ROM 深度为 256，倍频系数为 1，而输入时钟为 50Mhz，那么一个完整的正弦波周期长度为 256*20ns = 5120ns，当 FREQ_ADJ 的值为 0 时，即正弦波的最快输出频率为 1s/5120ns(1s = 1000000000ns) ≈ 195.3Khz。当我们把 FREQ_ADJ 的值设置为 5 时，一个完整的正弦波周期长度为 5120ns*(5+1) = 30720ns，频率约为 32.55KHz。也可以在 WaveToMem 软件设置中增加倍频系数或者增加 AD 的驱动时钟来提高正弦波输出频率。

AD 数据接收模块的代码如下：

```
1   module ad_wave_rec(
2       input                   clk         ,   //时钟
3       input                   rst_n       ,   //复位信号，低电平有效
4
5       input           [7:0]   ad_data     ,   //AD 输入数据
6       //模拟输入电压超出量程标志(本次试验未用到)
7       input                   ad_otr      ,   //0:在量程范围 1:超出量程
8       output   reg            ad_clk          //AD(TLC5510)驱动时钟,最大支持 20Mhz 时钟
9       );
10
11  //*****************************************************
12  //**                    main code
13  //*****************************************************
14
15  //时钟分频(2 分频,时钟频率为 25Mhz),产生 AD 时钟
16  always @(posedge clk or negedge rst_n) begin
17      if(rst_n == 1'b0)
18          ad_clk <= 1'b0;
```

```
19      else
20          ad_clk <= ~ad_clk;
21  end
22
23  endmodule
```

由于 AD 转换芯片支持的最大时钟频率为 32Mhz，而 ZYNQ PL 的系统时钟频率为 50Mhz，所以需要先对时钟进行分频，将分频后的时钟作为 AD 转换芯片的驱动时钟（分频计数见代码的第 16 行至第 21 行）。

需要说明的是，AD 数据接收模块没有对输入的 ad_otr（输入的模拟电压超出量程指示）和 ad_data（AD 输入的数据）做任何处理，这两个信号是在 ILA 中观察信号的变化的。

22.5 下载验证

将高速 AD-DA 模块插入 ZYNQ 启明星开发板的扩展口（靠近 ATK-Module 的位置），连接时注意扩展口电源引脚方向和开发板电源引脚方向一致，然后将下载器一端连接电脑，另一端与开发板上对应端口连接，最后连接电源线并打开电源开关。

启明星开发板硬件连接实物图如下图所示：

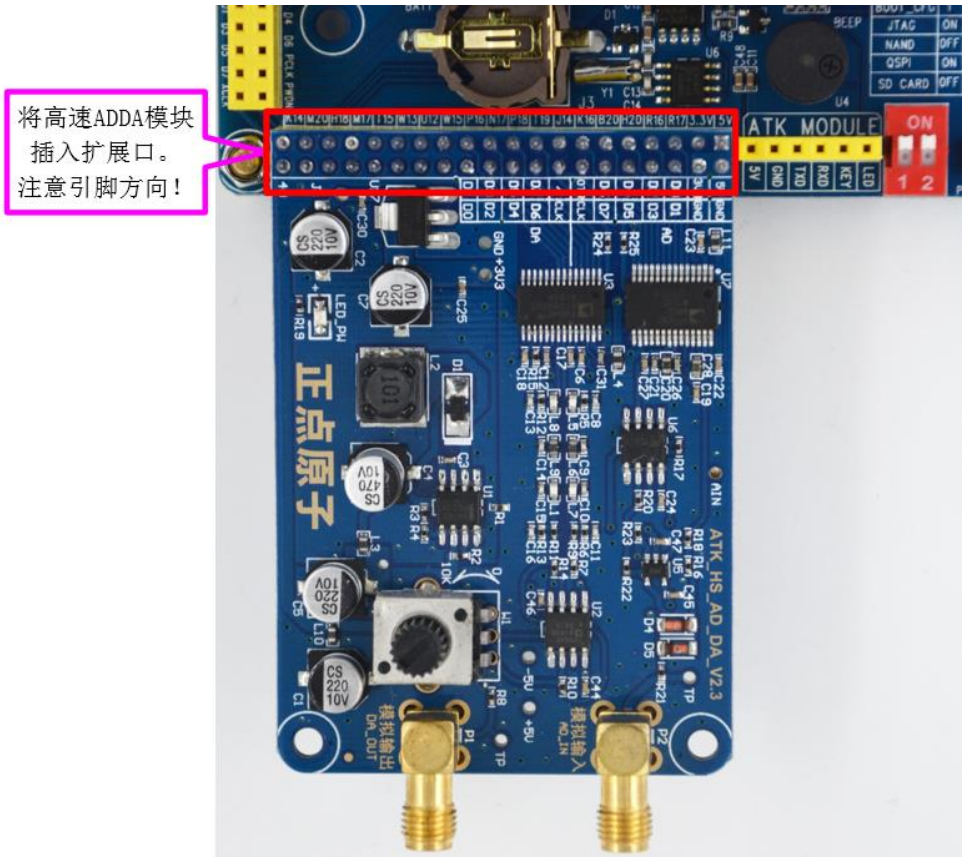


图 22.5.1 启明星开发板硬件连接实物图

将工程生成的比特流文件下载到 ZYNQ 中后，然后使用示波器测量 DA 输出通道的波形。首先将示波器带夹子的一端连接到开发板的 GND 位置（可使用杜邦线连接至开发板上的任一的 GND 管脚），然后将另一端探针插入高速 AD-DA 模块的 DA 通道中间的金属圆圈内（注意将红色的保护套拿掉），如图 22.5.2 所示。或者也可以直接测试高速 AD-DA 模块的 TP 引脚，如图 22.5.3 所示。

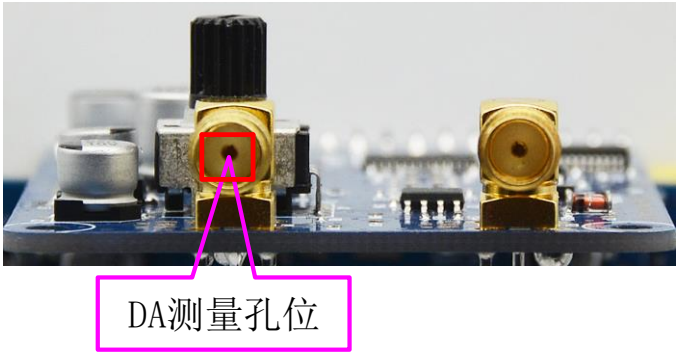


图 22.5.2 DA 模拟电压测量孔位

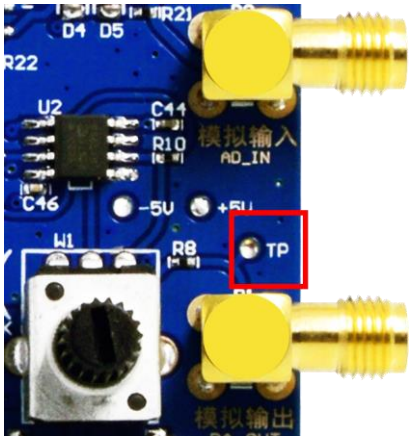


图 22.5.3 DA 模拟电压测量点（TP）

此时观察示波器可以看到正弦波的波形，如果观察不到波形，可查看示波器设置是否正确，可以尝试按下示波器的“AUTO”，再次观察示波器波形。示波器的显示界面如下图所示：

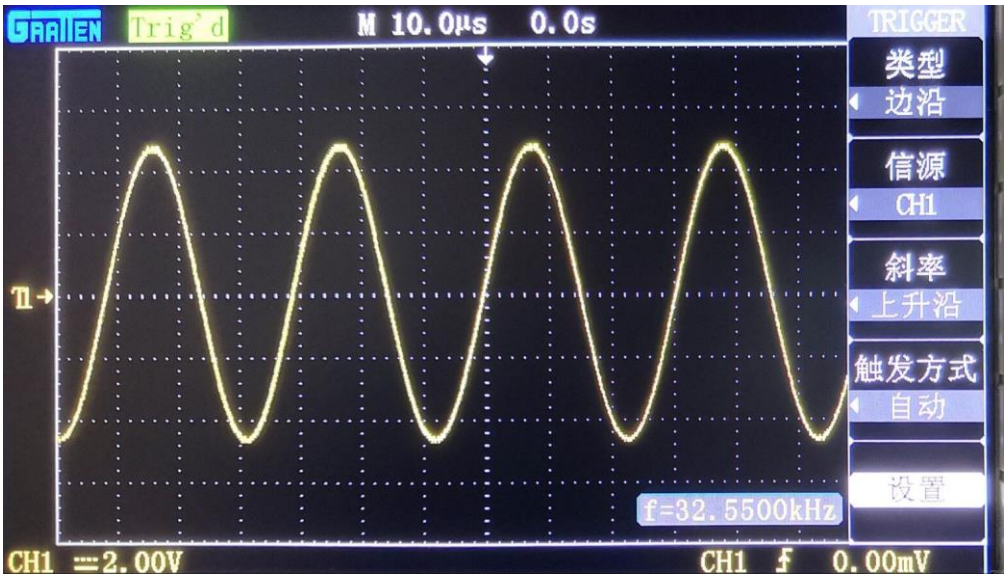


图 22.5.4 示波器显示界面

观察到正弦波波形后，说明 DA 已经正确输出模拟电压波形了，接下来我们来验证 AD 的功能，首先使用两头都是公头的杜邦线，将 DA 输出通道和 AD 输入通道连接起来，杜邦线连接图下图所示。

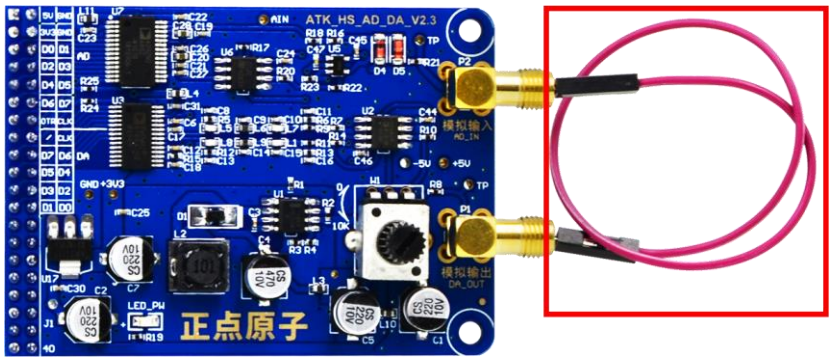


图 22.5.5 AD-DA 通道杜邦线连接图

连接后在 ILA 中观察 ad_data 数据的变化，观察到的波形如下图所示。

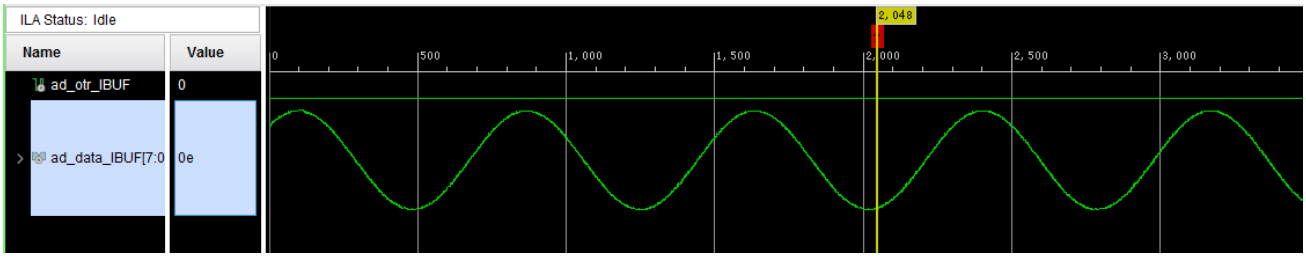


图 22.5.5 AD 数据接收模块采集到的 ILA 波形图

由上图可知，输入的 ad_data 数据为正弦波变化的波形，说明 AD-DA 实验验证成功。

另外，在这里介绍一下如何将数据设置成波形图显示。首先选中 ILA 波形图中的 ad_data，右键选择 Waveform Style，然后选择 Analog 即可。如果要切换成数据显示的话，同样选中 ad_data，右键选择 Waveform Style，然后选择 Digital 就可以了，如下图所示：

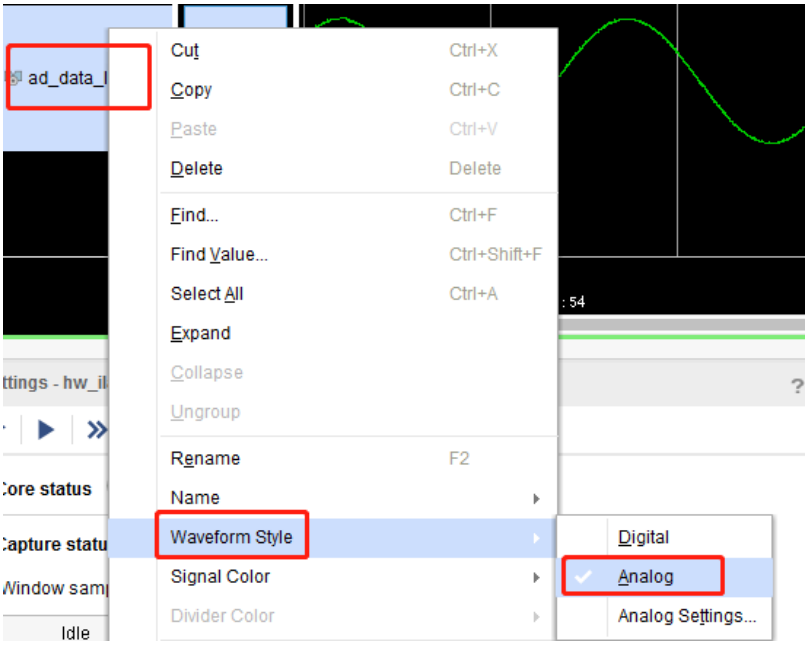


图 22.5.6 ILA 波形显示设置界面