手把手教你学 FPGA设计

——基于大道至简的至简设计法

潘文明　　易文兵　编著
[网名 大道至简]

北京航空航天大学出版社
BEIHANG UNIVERSITY PRESS

手把手教你学 FPGA 设计

——基于大道至简的至简设计法

潘文明　　易文兵　编著

[网名　大道至简]

北京航空航天大学出版社

内 容 简 介

本书介绍利用硬件描述语言(Verilog HDL)设计数字系统的方法,提出计数器、状态机、FIFO的三大架构,采用理论与实践相结合的方式,详细介绍各个架构的实现步骤——“八步法”,并配以相应的项目练习。在书的后半部分,介绍模块划分的基本方法,总结并提出了模块间的交互架构,在项目实践中给出了温度监控系统、边缘检测系统等综合性项目的模式。

本书可作为高等院校电子工程类、自动控制类、计算机类等专业的教材,亦可供FPGA相关行业的爱好者和从业人员自学与参考。

图书在版编目(CIP)数据

手把手教你学FPGA设计:基于大道至简的至简设计法 / 潘文明,易文兵编著. -- 北京 : 北京航空航天大学出版社, 2017.4

ISBN 978-7-5124-2401-2

Ⅰ. ①手… Ⅱ. ①潘… ②易… Ⅲ. ①可编程序逻辑器件 Ⅳ. ①TP332.1

中国版本图书馆CIP数据核字(2017)第086191号

版权所有,侵权必究。

手把手教你学FPGA设计

——基于大道至简的至简设计法

潘文明　　易文兵　编著

[网名　大道至简]

责任编辑　王　实

*

北京航空航天大学出版社出版发行

北京市海淀区学院路37号(邮编100191)　http://www.buaapress.com.cn

发行部电话:(010)82317024　传真:(010)82328026

读者信箱: emsbook@buaacm.com.cn　邮购电话:(010)82316936

北京市同江印刷有限公司印装　各地书店经销

*

开本:710×1 000　1/16　印张:19　字数:405千字

2017年5月第1版　2017年5月第1次印刷　印数:3 000册

ISBN 978-7-5124-2401-2　定价:45.00元

若本书有倒页、脱页、缺页等印装质量问题,请与本社发行部联系调换。联系电话:(010)82317024

序

随着微电子技术的发展，可编程门阵列（FPGA）技术已经成为信息产业最热门的技术之一，其应用范围遍及电子、通信、自动化、医疗、军事和航空航天等多个热门领域。在数字集成电路设计领域，前端验证工作通常依托 FPGA，并由那些精通硬件构架的 FPGA 系统工程师来完成。因此，FPGA 技术的设计和开发已成为当前发展迅速的朝阳行业之一。

作者曾在华为海思、展讯通信等集成电路设计企业参与了 FPGA 大型项目的开发设计，近年来又在明德扬科技教育公司从事 FPGA 的教学培训，从而有机会深入研究和采用多种教学方法进行试验，以便把那些构成 FPGA 设计基础的知识模块和工具平台，变成 FPGA 学习者能够实际运用的方法，而更重要的是能够帮助学习者消化核心功能及其设计流程。于是，本书便应运而生，旨在为 FPGA 学习者提供一种 FPGA 的至简设计方法。

与一些将 FPGA 设计的理论知识和软件工具作为叙述重点的教程不同，本书的重点不在于介绍 FPGA 设计“是什么”，而侧重于引导学习者“怎么做”。经过这样设计后的内容可以引起学习者浓厚的兴趣，并使学习者在短时间内快速掌握 FPGA 设计方法，正如诺贝尔文学奖获得者、法国作家 Anatole France 的精辟格言：“教学的全部艺术就是唤醒年轻心灵的天然好奇心。”

黄君凯

2016 年 10 月于广州暨南园

前　言

随着我国工业智能化的发展，工业企业的个性化需求剧增。相应的，对 FPGA/IC 行业人才的需求，特别是高端人才的需求也随之剧增。然而，目前国内的 FPGA/IC 教育相对落后：高等院校缺乏项目经验丰富的教师；现有教材陈旧，无法跟上最新技术的发展；学习资料缺乏，大都重视软件操作示范，不重视设计理念传授等。这些客观因素使学生对 FPGA/IC 望而生畏，即使有大量学生立志学好 FPGA/IC 设计，但多数不得要领，最终放弃学习。由于失去了群众基础，导致国内缺乏高素质的 FPGA 人才，从而对我国工业智能化的发展产生了一定的影响。为了扩大 FPGA/IC 设计的群众基础，降低 FPGA/IC 设计的学习门槛，明德扬科技教育公司（简称明德扬）结合培训班、企业内训，以及多年国内顶尖企业的工作经验，研发出一套 FPGA/IC 设计流程——“至简设计法”。“至简设计法”是明德扬丰富项目经验的结晶，它将高级的设计技巧转化成几条规则，掌握了这些规则就相当于拥有多年的经验。“至简设计法”流程中的每一步都简单易操作，容易上手，便于掌握。掌握“至简设计法”，可以实现 0 仿真、0 调试、一次成功的设计，极大地提高了工作效率。明德扬致力于打造“最简单、最实用、最高效”的 FPGA/IC 设计法，不遗余力地为普及 FPGA/IC 设计，提升客户竞争力，提高我国该行业整体的设计能力做贡献。

很多同学在刚开始学习 FPGA/IC 设计时都会走弯路，下面是两位学员的学习经历与心得。

学员一：

“① 买一本书，花大量的时间学习语法；

② 到网上搜一些资料，类似“大全”、“玩转”之类的速成之法；

③ 买一个开发板，看代码、下载例程看现象。”

学员一的情况具有普遍性，作为一个刚接触 FPGA 的初学者，这似乎是必经之路，很多学员在来明德扬学习之前就以这样的方式学了大约半年的时间。当然，这样的学习方法不是说学不好，而是很难学好，或者需花费很长的时间。以看书学习为例，可以说书是根源，学习任何知识都离不开书，书作为一个知识体系，要求全面、系统，但作为初学者就没有必要学得那么全面、系统了。很多学员在学 Verilog 时，就是拿一本书，仔仔细细地看，花费了不少时间，但效果却不理想。我们都知道，Verilog 的很多语法是不能综合的，而这些语法也没有实际的电路与之对应，当然在

具体的逻辑设计时也是不能用的，但书上并不会很明确地指出。再举一个例子，也是很多人都犯过的一个错误：以普通信号作为 always 的敏感信号列表。当我们这样写的时候，根本不会意识到这是错误的，因为书上就是这么教的。有经验的工程师都知道，对于 Verilog 语法，书上是一个全集，很多用法作为知识点是对的，但在实际应用中就是错误的。因此，太依赖书本的学习，反而会学得全而不好，还浪费时间。

学员二：

“一开学我立马就买了当时很多人推荐的某一品牌的一款开发板，但却不知道怎么学，上来就死抠代码，感觉好费劲啊！一方面我当时的基础确实很差，本科时候就没写过任何代码；另一方面看别人写的代码本来就费解，而卖家的视频也是蜻蜓点水一带而过，视频上提到的问题不是我的问题，我想解决的问题视频中又没说，反正我的困惑与卖家的视频就不在一个频道上，可能我的问题人家压根儿就觉得不是问题，因为我当时就是不折不扣的零基础。就这样，我混混沌沌地看了好多天代码，最终还是云里雾里。当时有人跟我讲过要去仿真看波形，而不是看代码，但是装什么软件，怎么装，软件怎么用，我一点都不知道。确实蛮痛苦的。”

以上两位同学遇到的问题，相信大多数初学者也遇到过，也很受困扰。明德扬在培训学员的过程中发现了一个普遍存在的问题：不同的人，代码风格往往也是不一样的。而一套优秀的代码规范，对团队、企业有非常强的指导意义，可以给他人提供一个可读性强、便于维护纠错的环境，这可以大大提高团队、企业的工作效率。

而市面上大多数 FPGA 教材往往并不重视代码规范，大篇幅的软件使用、语法知识和大量理论知识令初学者望而生畏，其实很多语法在实际工程中并没有机会用到。另外，很多 FPGA 书籍都会讲一个个独立的项目如何实现，以及针对不同项目的相应设计思路，但却缺少一套通用的设计方法，这样同学们最多只能学一个项目会一个项目，而不能举一反三。也正是因为如此，才有这本书创作的初衷。

为了帮助更多的 FPGA 学习者更好地理解 FPGA 设计的精髓，提高学习效率，明德扬提出了“至简”的设计思想，并创作了这本书。

大道至简与至简设计法

“大道至简”出自老子的《道德经》：“万物之始，大道至简，衍化至繁。”大道至简是指大道理（基本原理、方法和规律）是极其简单的，简单到一两句话就能说明白，把复杂冗繁的表象层层剥离之后就是事物最本质的东西。所谓“真传一句话，假传万卷书”。一门技术一门学问，弄得很深奥是因为没有看穿实质，就像 FPGA 的学习，搞得很复杂是因为没有抓住代码的关键。

1. 大道至简，抓住根本

大道理是极其简单的，简单到一两句话就能说明白。事情难就难在简单，简单不是敷衍了事，也不是单纯幼稚，而是最高级的智慧和成熟睿智的表现。

至简设计法是明德扬通过多年的教学经验找到的学习 FPGA 的关键和根本方法。FPGA 学习既不能花大量时间在软件工具/语法上，也不能花精力在独立的知识点上，而应花费 80%的时间在 20%的核心功能上。

什么是核心功能？至简设计中一个完整的设计流程就是核心功能。注意，这里说的是“设计”流程，而非项目流程。市面上的 FPGA 设计教程大部分讲的都是项目流程，即介绍功能→新建项目→加入设计文件→编译工程→配置引脚→上板。这些流程都是介绍软件是怎么用的，而最重要的——设计文件是如何设计的，都没有详细介绍。

至简设计法的设计流程，讲的是实现一个功能的过程，是将功能转化成代码的过程。不仅简单的 UART/SPI/VAG/IIC/摄像头采集，甚至复杂的边缘检测/SDRAM 接口/软件无线电等，都可以采用这个设计流程将功能转化成代码。

2. 大道至简，衍化至繁

大道至简是一种哲学，无论是做人还是做事它都会给人以指导。在当今这个大千世界中，我们要学会把复杂变成简单，用智慧创造“简单”，在变迁中不断地升华。

至简设计法是先将复杂的模块划分成简单的模块，再用模块设计流程设计出来。FPGA 学习的过程，就是不断地将复杂的模块，划分成简单模块，再用至简设计法设计的过程。不断地重复这个过程，像搭积木一样一层层地堆砌，最终就能通过简单的规则做出复杂的设计。

3. 大道至简，凝聚智慧

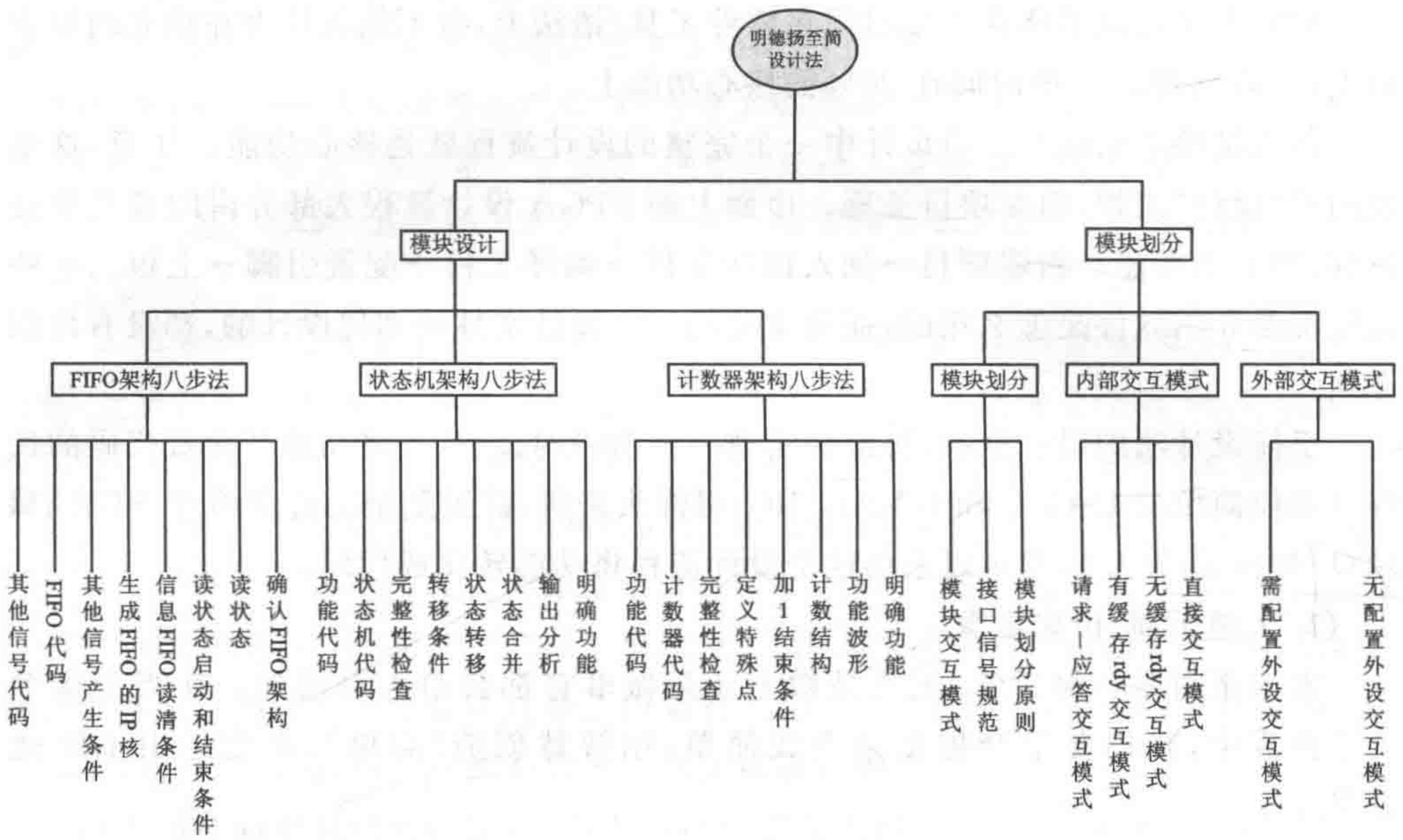
大道至简，经过整合创新，跳出原来的框框，去粗取精，抓住关键和根本，挥动奥卡姆剃刀，剔除无效的、可有可无的、非本质的东西，提炼出精华。

至简设计法，看似简单的几个规则，然而总结归纳出这些规则，却需要相当大的智慧：

- 必须要有众多领域的项目经验，才能获知各领域的设计需求和要点；
- 必须要有丰富的工作经验，才能拥有高超的设计技巧；
- 必须要有丰富的教学经验，明白学习的难点和重点，明白学生思考的方式，才能将高超的设计技巧转成可学习和操作的流程；
- 必须要有很大的智慧，不断地归纳和总结，才能将这些流程和步骤变得简单易用。

至简设计法，是结合以上要点，经过明德扬多年探索的智慧结晶。在此，我希望与更多 FPGA 学习者分享，让更多的人应用此设计方法，在 FPGA 设计学习和做实际项目中得心应手，为推进国家产业智能化发展贡献绵薄之力。

至简设计法内容体系



本书共 8 章，分为三篇，其中每篇对应的章和具体内容介绍如下：

第一篇包括第 1～4 章，主要介绍 FPGA 设计核心——模块。本书提出把模块分为三大类型，即计数器架构、状态机架构及 FIFO 架构，并配有项目设计练习。

第二篇包括第 5、6 章，主要介绍模块的由来——模块划分。本书总结了两大类常见的模块划分应用场合，即 FPGA 内部模块之间的交互和 FPGA 与外设的交互，并配有实践练习。

第三篇包括第 7、8 章，主要介绍基于 OV7670 的图像采集，以及基于 SDRAM 的 VGA 显示控制器。应用前两篇提到的三大架构及模块划分来完成这个工程，进一步深化对前面内容的理解。

本书特点

1. 注重设计思路

本书重点讲项目实现的整体思路，而并非讲某一具体项目的实现。

2. 设计理念

本书提出了“至简”的设计理念，一次只考虑一个问题。

3. 优秀的代码风格及规范

本书提供的代码风格统一，代码的规范是资深从业人员从多年工作经验中总结、提练，并结合明德扬就业培训班培训成果所得到的。优秀的代码风格对团队、企业都有非常大的指导意义。

4. 注重实践

本书选择以案例的形式进行编写，所有方法均用例子阐述，并配有丰富的项目实践练习。

5. 不罗列知识点

坚持项目中遇到什么问题就解决（讲解）什么问题，避免罗列知识点，让 FPGA 学习者集中精力学习 FPGA 设计的核心。

如何学习本书

至简设计法的目的，是对程序编写工作中易出现的混乱状态进行规范化，以达到简便、有序、快捷的效果。本书通过例题引出概念，并详举大量实际案例进行分析、对比，帮助读者理解至简设计法的设计思想，清晰直观地学习和掌握该方法。此外，作者对于“二八定律”在 FPGA 设计方面的应用有独到见解：采取“权值阶梯模式”，即强调核心功能的学习、巩固和使用，以及一般功能在实际中边做边学的模式，来达到在最短的时间能独立完成项目的效果。另外，希望读者在完成项目时，先根据功能要求自己试着构思、编写代码，再与给出的思路、代码相比较，这样读者才能深刻理解至简设计法，并把这套设计思想真正运用到实践中去。

从明德扬就业培训班的培训成果来看，只要耐心学完整套设计思想，就能独立完成项目设计。设计者拿到一个新项目，知道从哪里下手，且编写的代码可综合，逻辑缜密，在设计过程中对逻辑变化考虑周全，就可在短时间内调试成功并上板实现。

本书虽然没有语法篇，但通篇用到的语法都是比较简单、常用的，有一定 Verilog 基础的读者读起来必定毫无障碍，而初学者也不必担心看不懂，明德扬会推出配套视频，介绍用到的语法知识点。另外，软件使用篇也包括在配套视频中，主要包括 Quartus II，Modelsim，Gvim 的使用。

本书可作为高等院校电子工程类、自动控制类、计算机类等专业的教材，亦可供 FPGA 相关行业的爱好者和从业人员自学与参考。

感谢参与本书部分内容编写及在编写过程中提供宝贵建议的黄君凯、崔苗、李国庆和陈泽熹。还要感谢北京航空航天大学出版社的编辑老师们，在他们的大力支持和帮助下本书才得以与广大读者见面。

由于作者水平有限，难免存在错误和遗漏之处，恳请各位读者批评指正。同时，也欢迎大家就 FPGA 开发相关技术与作者交流，联系邮箱：book@mdy-edu.com。

配套视频、开发板购买地址以及例程源码和相关资料下载地址：http://www.mdy-edu.com/。

潘文明　易文兵

2016 年 12 月于深圳明德扬科技教育有限公司

目　录

第一篇　模块设计

第 1 章　模　块 …… 2
1.1　模块结构 …… 2
1.2　模块的三大架构概述 …… 4
第 2 章　计数器架构 …… 6
2.1　计数器架构八步法 …… 6
2.1.1　计数器规则 …… 6
2.1.2　第一步:明确功能 …… 9
2.1.3　第二步:功能波形 …… 10
2.1.4　第三步:计数结构 …… 10
2.1.5　第四步:加 1 和结束条件 …… 11
2.1.6　第五步:定义特殊点 …… 12
2.1.7　第六步:完整性检查 …… 12
2.1.8　第七步:计数器代码 …… 13
2.1.9　第八步:功能代码 …… 14
2.1.10　总　结 …… 16
2.2　计数器项目实践 …… 16
2.2.1　PWM 流水灯项目 …… 16
2.2.2　UART 接口项目 …… 22
2.2.3　VGA 接口项目 …… 34
2.2.4　SCCB(IIC)接口项目 …… 42
第 3 章　状态机架构 …… 54
3.1　状态机架构八步法 …… 54
3.1.1　状态机规则 …… 54
3.1.2　第一步:明确功能 …… 58
3.1.3　第二步:输出分析 …… 58
3.1.4　第三步:状态合并 …… 59
3.1.5　第四步:状态转移 …… 60

3.1.6 第五步:转移条件 …… 60
3.1.7 第六步:完整性检查 …… 63
3.1.8 第七步:状态机代码 …… 63
3.1.9 第八步:功能代码 …… 67
3.1.10 总 结 …… 68
3.2 状态机项目实践 …… 68
3.2.1 简易报文类型识别器项目 …… 68
3.2.2 SPI 接口项目 …… 78
3.2.3 SDRAM 接口项目 …… 88
第 4 章 FIFO 架构 …… 112
4.1 FIFO 架构八步法 …… 112
4.1.1 概 述 …… 112
4.1.2 第一步:确认 FIFO 架构 …… 123
4.1.3 第二步:读状态 …… 124
4.1.4 第三步:读状态启动和结束条件 …… 124
4.1.5 第四步:信息 FIFO 读清条件 …… 125
4.1.6 第五步:生成 FIFO 的 IP 核 …… 125
4.1.7 第六步:其他信号的产生条件 …… 125
4.1.8 第七步:FIFO 代码 …… 126
4.1.9 第八步:其他信号代码 …… 127
4.1.10 总 结 …… 129
4.2 FIFO 项目实践 …… 129
4.2.1 报文汇聚模块项目 …… 129
4.2.2 以太网切包模块项目 …… 134
4.2.3 UDP 打包模块项目 …… 142

第二篇 模块划分

第 5 章 模块划分核心知识 …… 158
5.1 模块划分概述 …… 158
5.1.1 模块划分原则 …… 158
5.1.2 模块划分端口规范 …… 159
5.2 模块划分过程 …… 159
5.2.1 模块划分思路 …… 159
5.2.2 模块的端口及模块之间的数据流向 …… 162
第 6 章 模块划分常用架构及项目实践 …… 164
6.1 直接交互架构 …… 164

6.2 无缓存 rdy 交互架构 …… 165
6.3 有缓存 rdy 交互架构 …… 165
6.4 请求-应答交互架构 …… 165
6.5 外设交互架构 …… 167
6.6 模块划分项目实践 …… 167
6.6.1 AT93C46 的存储与读取项目 …… 167
6.6.2 温度采集与显示项目 …… 170
6.6.3 OV7670 图像采集项目 …… 172

第三篇 项目实践

第 7 章 基于 FPGA 的温度监控系统 …… 176
7.1 背景介绍 …… 176
7.2 项目要求 …… 176
7.3 基本原理 …… 179
7.3.1 PC 端(串口调试助手) …… 179
7.3.2 温度传感器 DS18B20 …… 179
7.3.3 数码管显示 …… 181
7.3.4 蜂鸣器 …… 183
7.4 设计方案 …… 183
7.5 FPGA 设计实现 …… 187
7.5.1 硬 件 …… 187
7.5.2 Verilog 代码设计 …… 188
7.6 功能仿真 …… 222
7.7 板级调试 …… 227
7.8 总 结 …… 227
第 8 章 基于 FPGA 的图像边缘检测系统 …… 228
8.1 背景介绍 …… 228
8.2 设计要求 …… 228
8.3 基本原理 …… 229
8.3.1 PLL …… 229
8.3.2 OV7670 配置与采集 …… 233
8.3.3 灰度转换 …… 234
8.3.4 高斯滤波 …… 235
8.3.5 Sobel 边缘检测 …… 237
8.3.6 乒乓操作 …… 238
8.4 设计方案 …… 239

8.5 FPGA 设计实现 …… 241
8.5.1 硬 件 …… 241
8.5.2 Verilog 代码设计 …… 242
8.6 功能仿真 …… 279
8.7 板级调试 …… 286
8.8 总 结 …… 287
参考文献 …… 288

第一篇　模块设计

第 1 章　模　块
第 2 章　计数器架构
第 3 章　状态机架构
第 4 章　FIFO 架构

第 1 章

模　块

1.1　模块结构

采用模块化的设计使系统更有条理且便于仿真和测试，那么整个项目的设计思想就是模块套模块，自顶向下依次展开。

本书主要以 Verilog 硬件描述语言为主，模块是 Verilog 的基本描述单位，用于描述每个设计的功能和结构，以及其他模块通信的外部接口。模块有五个主要部分：端口定义、参数定义（可选）、I/O 说明、内部信号说明、功能定义。模块总是以关键词 module 开始，以关键词 endmodule 结尾。它的一般语法结构如下：

```
1   module module_name(
2           clk   ,    //端口1，时钟
3           rst_n,    // 端口2，复位
4           dout      // 其他信号，如dout
5     );
6
7     parameter     DATA_W = 8;
8
9     input       clk   ;    // 输入信号定义
10    input       rst_n;    // 输入信号定义
11
12    output[DATA_W-1:0]  dout; //输出信号定义
13
14    reg    [DATA_W-1:0] dou;//信号类型(reg、wire)定义
15
16    reg              signal1;//信号类型(reg、wire)定义
17
18  //.........以下为描述功能部分 ........//
```

端口定义

参数定义（可选）

I/O说明

内部信号说明

```
19
20      //组合逻辑写法
21      always@(*)begin
22      end
23
24      //时序逻辑写法
25      always@(posedge clk or negedge rst_n)begin
26          if(rst_n==1'b0)begin
27          end
28          else begin
29          end
30      end
31
32  endmodule
```

功能定义(功能描述)

下面详细分析这段代码：

1. 模块的端口定义(第1～5行)

模块的端口声明了模块的输入/输出口。其格式如下：

```
module 模块名(端口 1, 端口 2, 端口 3,……);
```

其中，模块是以 module 开始，以 endmodule 结束。模块名是模块唯一的标识符，建议模块名尽量用能够描述其功能的名字来命名。模块的端口表示模块的输入口和输出口，也就是它与其他模块联系端口的标识。

2. 参数定义(第7行)

参数定义是将常量用符号代替，以增加代码的可读性和可修改性，是一个可选择的语句，不用时可以省略。参数定义的一般格式如下：

```
parameter  DATA_W = x;
```

3. I/O 说明(第9～12行)

输入口：

```
input [信号位宽 - 1 : 0] 端口名 1;
      input [信号位宽 - 1 : 0] 端口名 2;
       ……;
```

输出口：

```
output [信号位宽 - 1:0] 端口名 1;
        output [信号位宽 - 1:0] 端口名 2;
        ……;
```

输入/输出口：

```
inout [信号位宽 - 1:0] 端口名 1;
        inout [信号位宽 - 1:0] 端口名 2;
        ……;
```

4. 内部信号说明(第 14～16 行)

在模块内，用到的和与端口有关的 wire 和 reg 类型变量的声明，如：

```
reg [width - 1:0] R 变量 1, R 变量 2 ……;
wire [width - 1:0] W 变量 1,W 变量 2……;
……;
```

5. 功能定义(描述功能)(第 21～31 行)

模块中最重要的部分是逻辑功能定义部分。以下三种方法可在模块中产生逻辑：

① 用“assign”声明语句，如描述一个两输入“与”门：assign a = b & c;

② 模块例化；

③ 用“always”块。

看到这里，相信很多读者都会一头雾水，因为整个模块语法结构很复杂，有各种型号定义、类型，以及功能描述。为了避免学员花过多时间在这些固定格式的学习上，明德扬制作了一套 GVIM(Graphical Vi Improved)模板，只需根据端口名和信号位宽进行修改，就可以轻松地写出 1～4 部分。如在 GVIM 输入 Module(首字母大写)，便可以得到上述语法结构中的模板。另外，输入一些其他指令可以得到不同的模板，如 Shixu(时序逻辑电路)、Zuhe(组合逻辑电路)、Jsq(计数器)和 Ztj(状态机)等，在后续章节使用时再做说明。

模块设计的难点在于功能定义部分，因此本书将重点介绍该部分的设计。设计模块的功能定义时只要按照明德扬模块架构一步步进行，不管多么复杂的功能，都能设计出条理清晰、高效正确的代码。

1.2 模块的三大架构概述

明德扬根据模块实现的功能，将其分成三种架构——计数器架构、状态机架构和 FIFO 架构。几乎所有模块，都可以应用以上三种架构进行设计。不同架构，设计步骤不同。本书将在第 2 章介绍计数器架构八步法及其项目应用，在第 3 章介绍状态

机架构八步法及其项目应用，在第4章介绍FIFO架构八步法及其项目应用。

1. 计数器架构

定义：以计数器作为模块的主体结构。

设计特点：先将计数器设计好，再以计数器为框架，其余信号与之对齐。

应用场合：串口、VGA等接口。

2. 状态机架构

定义：以状态机作为模块的主体结构，根据需求以计数器为其状态转移条件。

设计特点：先将状态机设计好，再以状态机为框架，其余信号根据不同状态产生。

应用场合：SPI接口、SDRAM等。

3. FIFO架构

定义：以FIFO读、写时钟分离的性质，使用FIFO作为模块间数据传输的缓存。

设计特点：先选择好FIFO类型并确定其宽度和深度，再设计读侧信号和写侧信号。

应用场合：数据跨时钟域传输等。

第 2 章

计数器架构

计数器设计是 FPGA 设计的核心，计数器架构是 FPGA 设计最常用的架构之一，比较简单。它也是明德扬至简设计法的核心部分，相信广大初学者认真学习这部分内容必能掌握 FPGA 设计的精髓，本章将通过大量项目练习帮助读者巩固明德扬至简设计法。

2.1 计数器架构八步法

本节主要介绍明德扬计数器架构八步法。在使用八步法前，明德扬根据多年项目和培训经验，制定了一套严谨的计数器规则。

2.1.1 计数器规则

计数器规则 1：计数器逐一考虑三要素——初值、加 1 条件和结束值。

任何计数器都有三个要素：初值、加 1 条件和结束值。

初值：计数器的默认值或者开始计数的值。

加 1 条件：计数器执行加 1 的条件。

结束值：计数器计数周期的最后一个值。

设计计数器，要逐一考虑这三个要素，一般是先考虑初值，再考虑加 1 条件，最后再考虑结束值。

计数器规则 2：计数初值必须为 0。

计数器的默认值和开始值一定要为 0。这是明德扬规范的统一要求。我们知道一般编程语言计数都是从 0 开始的，0 表示第 1 个，1 表示第 2 个。明德扬也参考这种做法，计数器都从 0 开始计数。

所有计数器都统一从 0 开始计数，有助于我们阅读理解、方便使用，从而不用从头看具体代码，就能清楚这个数值的含义。

计数器规则 3：使用某一计数值，必须同时满足加 1 条件。

计数器从 0 开始计数，计数器的默认值，也就是复位值也是 0。当计数器值为 0 时，如何判断这是计数的第 1 个值，还是没开始计数的默认值呢？

可以通过加 1 条件来判断。当加 1 条件无效时，计数器值为 0 表示未开始计数

的默认值；当加1条件有效时，计数器值为0表示计数的第1个值。同样的道理，当cnt==x-1时，不表示数到x；只有当cnt==x-1，并且加1条件有效时，才表示数到x。

例如，当加1条件为add_cnt，且add_cnt && cnt==4时，表示计数到5个；而当add_cnt==0 && cnt==4时，不表示计数到5个。

计数器规则4：结束条件必须同时满足加1条件，且结束值必须是x-1的形式。

计数器的结束条件必须同时满足加1条件。例如假设要计数5个，那么结束值是4，但是结束条件不是cnt==4，而是add_cnt && cnt==4。因为cnt==4不表示计数到5个，只有add_cnt && cnt==4时，才表示计数到5个。

为了更好地阅读代码，明德扬规定结束值必须是x-1的形式，即add_cnt && cnt==4要写成add_cnt && cnt==5-1。这里的"5"表示希望计算的个数，"-1"则是固定格式。有了这个约定后，计数的边界就很明确了。明德扬多年的项目实践发现，无论多么有经验的工程师，对于边界值都要多花一些心思来考虑。因此，明德扬索性就制定了这个规则，以后就不用再想边界条件了。

计数器规则5：当取某个数时，assign形式必须为：(加1条件) && (cnt==计数值-1)。

当要从计数器取某个数时，例如要取计数器的第5个点，就很容易写成cnt==5-1，这是不正确的。正确的写法是：(加1条件) && (cnt==计数值-1)，如add_cnt&&cnt==5-1。原因可以参考计数器规则3的说明。

计数器规则6：结束后必须回到0。

每轮计数周期结束后，计数器变回0，这是为了使计数器能够循环重复计数。

计数器规则7：若需要限定范围，则推荐使用">="和"<"两种符号。

设计时，考虑边界值通常要花费一些心思，而且容易出错。为此，明德扬规则约定：若需要限定范围，则推荐使用">="和"<"两种符号。例如要取前8个数，那么就取cnt>=0 && cnt<8。注意，一定是"大于或等于"和"小于"符号，而不能使用"大于"和"小于或等于"符号。

该规则参考编程里的for循环语句。假如要循环8次，for循环的条件通常会写成"i=0;i<8;i++"，前面的0表示开始值，后面的8表示循环次数。当然，也可以写成"i=0;i<=7;i++"，但这些数字的意义就实在令人费解了，虽然知道7是从8-1得来的，但多一个"-1"的思考，就纯属画蛇添足了。

计数器规则8：设计步骤是，先写计数器的always段，条件用名字代替；然后用assign写出加1条件；最后用assign写出结束条件。

明德扬规定的计数器代码包括三段。

第一段，写出计数器的always段。

模板如下：

```
1  always  @(posedge clk or negedge rst_n)begin
2      if(rst_n == 1'b0)begin
3          cnt <= 0;
4      end
5      else if(加 1 条件) begin
6          if(结束条件)
7              cnt <= 0;
8          else
9              cnt <= cnt + 1;
10     end
11 end
```

大家有没有发现上述模板的特点？这个模板只需要填两项内容：加 1 条件和结束条件。如果为加 1 条件和结束条件定义一个信号名，例如 add_cnt 和 end_cnt，则代码变成：

```
1  always  @(posedge clk or negedge rst_n)begin
2      if(rst_n == 1'b0)begin
3          cnt <= 0;
4      end
5      else if(add_cnt) begin
6          if(end_cnt)
7              cnt <= 0;
8          else
9              cnt <= cnt + 1;
10     end
11 end
```

至此，就完成了 always 段的设计，是不是很简单？只要想好命名，就完成了这段设计代码。

第二段，用 assign 写出加 1 条件。

在此阶段，只需要想好一个点，就是计数器的加 1 条件。假设计数器的加 1 条件为 a=2，则代码如下：

```
1  assign  add_cnt = a == 2;
```

第三段，用 assign 写出结束条件。

在此阶段，只需要想好一个点，就是计数器的结束值。参考计数器规则 5 的要求，结束条件的形式一定是：(加 1 条件) && (cnt==计数值-1)。假设计数器要计数 10 个，则代码如下：

```
1  assign  end_cnt = add_cnt && cnt == 10 - 1;
```

至此，就完成了计数器代码的设计。总结一下这段代码特点：每次只考虑一件事，按这要求去做，就非常容易完成代码设计。

计数器规则 9：加 1 条件必须与计数器严格对齐，其他信号一律向计数器对齐。

我们设计出计数器，但一般计数器不是最终的目的，最终的目的是输出各种信号。设计计数器是为方便产生这些输出信号（包括中间信号），并能从计数器获取变化条件。例如信号 dout 在计数到 6 时拉高，则其变 1 的条件是：add_cnt && cnt==6-1。

假设有两个信号：dout0 在计数到 6 时拉高；dout1 在计数到 7 时拉高。一种做法是 dout0 变 1 的条件是 add_cnt && cnt==6-1，dout1 变 1 的条件是 dout0==1。这个 dout1 就是间接与计数器对齐。这是非常不好的方法。明德扬建议一律向计数器对齐，dout1 变 1 的条件应该为 add_cnt && cnt==7-1。

计数器规则 10：命名必须符合规范，比如：add_cnt 表示加 1 条件；end_cnt 表示结束条件。

如无特别说明，计数器的命名都要符合规范，加 1 条件的前缀为“add_”，结束条件的前缀为“end_”。

计数器规则 11：减 1 计数器暂时不用。

减 1 计数器在项目中应用也比较广泛。但在学习阶段，明德扬建议暂时不用，进入公司或者做项目时，根据公司要求而定。

以上就是明德扬的计数器规则。计数看似简单，但要用好却并非易事。明德扬强调，要真正掌握计数器的使用，需多实践，故在本书有大量的实践项目。读者可通过项目来掌握、搞透计数器的设计。

下面通过一个例子，来介绍计数器架构八步法。

当收到 en=1 时，执行以下操作：

a. 间隔 1 个时钟周期后，dout 产生宽度为 1 的高电平脉冲；

b. 间隔 1 个时钟周期后，dout 产生宽度为 2 的高电平脉冲；

c. 间隔 1 个时钟周期后，dout 产生宽度为 3 的高电平脉冲；

d. 间隔 1 个时钟周期后，dout 产生宽度为 4 的高电平脉冲。

2.1.2　第一步：明确功能

明确功能是将要实现的功能，用具体、清晰和无疑义的话描述出来，要重点描述每个信号的变化情况。要明确的功能包括：模块的信号列表（见表 2-1）；每个信号具体的变化情况。

表 2-1　信号列表

信号名	I/O	位　宽	说　明
clk	I	1	模块工作时钟
en	I	1	使能信号
dout	O	1	模块输出信号

2.1.3 第二步:功能波形

功能波形是将第一步描述的功能,用波形表示出来(见图 2-1)。波形的每个变化点都要清晰和精确,要具体到有多少个时钟。不能有模糊的情况,如果这一步模糊了,那么后面无论如何也设计不出来。

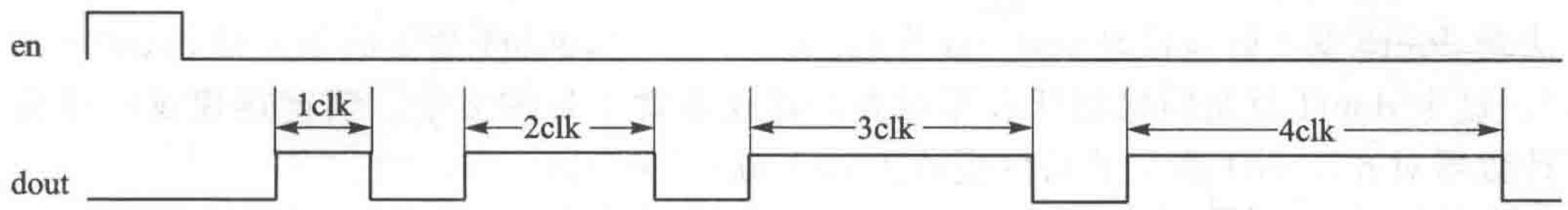


图 2-1 功能波形图

2.1.4 第三步:计数结构

计数结构是通过一个或多个计数器,搭建成整个设计的框架,从而作为其他信号的对齐条件。计数器优秀的标准是:用计数器能指示任何一个时钟;能方便地被其他信号归纳使用。

计数结构一般有两种:一种计数结构如图 2-2 所示,其从 0 一直计数到 13;另一种计数结构,这里包括两个计数器,即一个计数器 cnt 用于计时 dout 在每个阶段的低电平和高电平时钟周期,另一个计数器(cnt_c)用于计时并判断当收到 en=1 后 dout 是在 a、b、c、d 的哪个阶段。明德扬将后一种计数结构的设计方法称为计数器框架法。

明德扬推荐使用图 2-3 所示的计数结构。

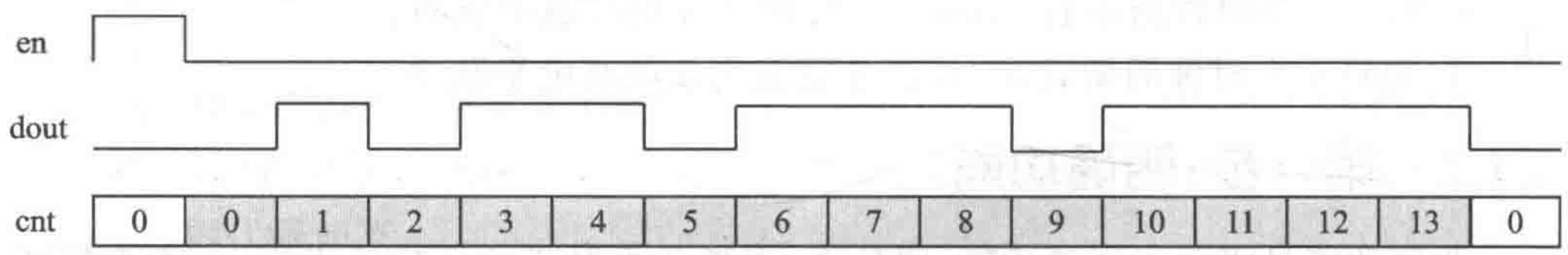


图 2-2 计数结构 1

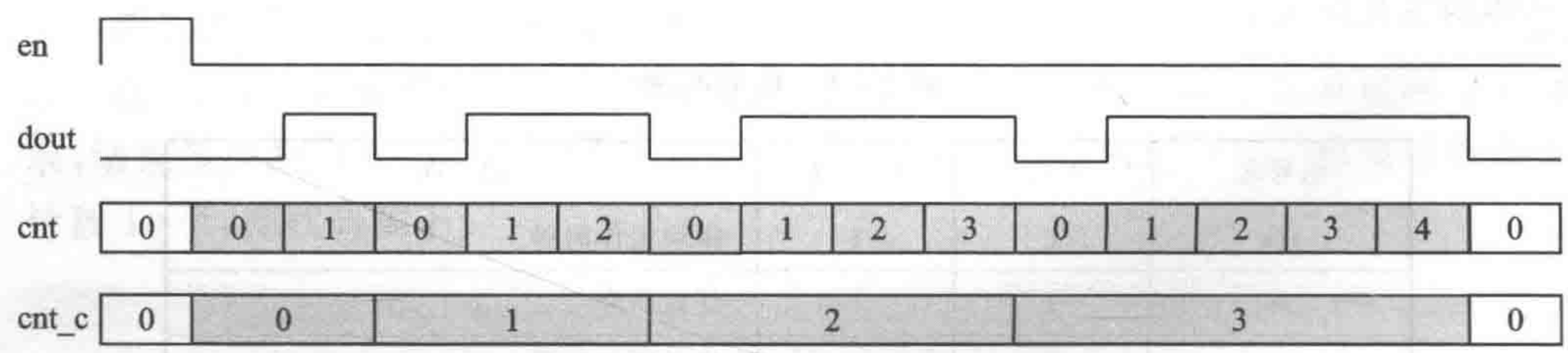


图 2-3 计数结构 2

2.1.5　第四步:加 1 和结束条件

加 1 和结束条件是考虑计数器的加 1 条件和结束条件,如果条件不足时,则要添加信号来指示。

① 逐个计数器考虑其加 1 条件和结束条件;

② 加 1 条件和结束条件必须精确到某时钟上升沿;

③ 条件必须用信号表示,而不是文字。

所谓计数器加 1 条件就是在满足计数条件的情况下计数器值加 1,一般其值每过一个时钟周期加 1,而有些计数器则可能会经过好几个时钟周期甚至更多。结束值则是满足加 1 条件下,计数的最后一个值。

1. cnt

如图 2-3 所示,cnt 的加 1 条件是图中的灰色区域,但没有任何信号可以区分出来。因此,增加一个指示信号 flag,即当 flag=1 时,cnt 加 1,如图 2-4 所示。

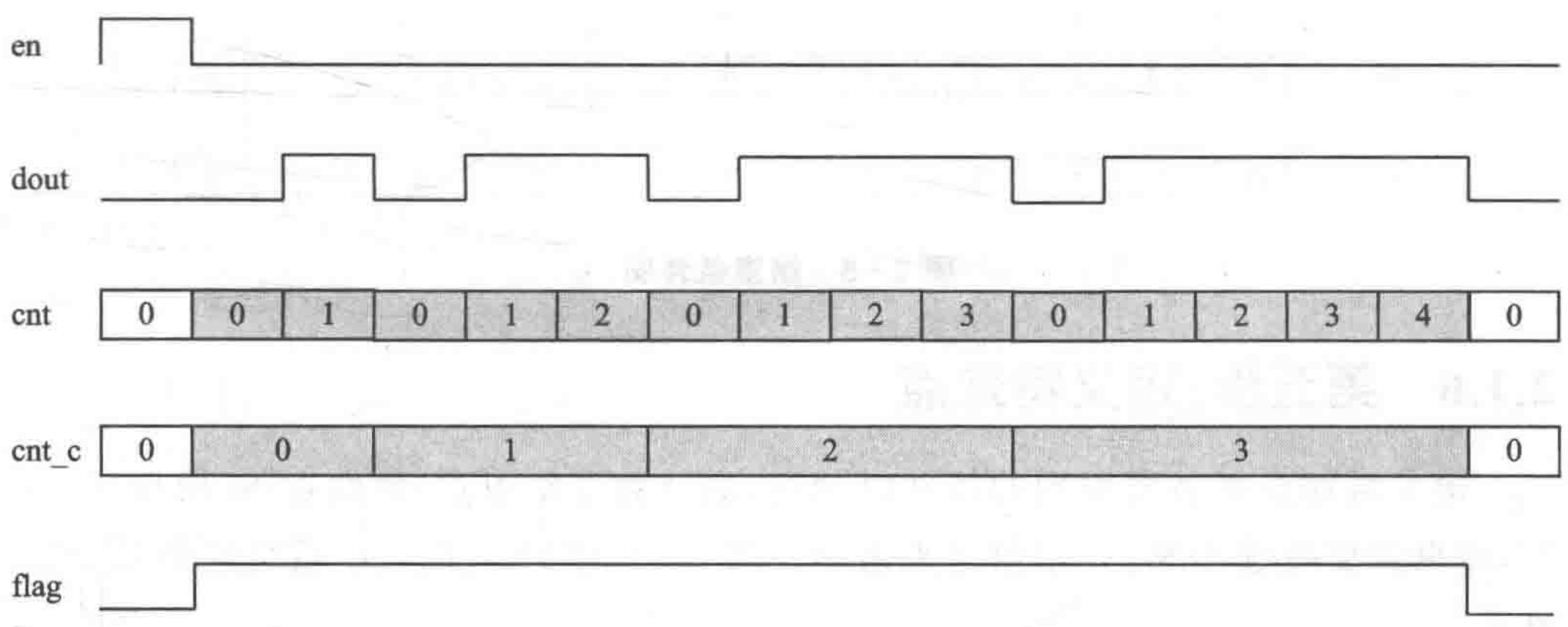


图 2-4　加 1 条件图

cnt 的结束值有四种情况:1、2、3、4。如图 2-5 所示,在 a 阶段计数到 1,在 b 阶段计数到 2,在 c 阶段计数到 3,在 d 阶段计数到 4。因为 cnt 的结束值在 a、b、c、d 各阶段不同,并且低电平和高电平持续的周期也不同,但它们的波形相似,所以引进两个变量 x 和 y,用 x 表示在 a、b、c、d 各阶段 dout 的低电平持续的时钟周期,y 表示高电平持续的时钟周期,用 $x+y-1$ 表示 cnt 的结束值,则 cnt 的结束条件为计数至 cnt==$x+y-1$。

这种用统计变量作为指示变化点的方法,明德扬称为计数器变量法。

综上所述,cnt 的加 1 条件是 flag==1;结束条件是 cnt==$x+y-1$。

2. cnt_c

cnt_c 是在 cnt 计数结束时加 1。收到 en 后，dout 有四个不同的输出阶段，故 cnt_c 的结束条件为计数至 cnt_c==4-1。

综上所述，cnt_c 的加 1 条件是 cnt==$x+y-1$，结束条件是 cnt==4-1，见图 2-5。

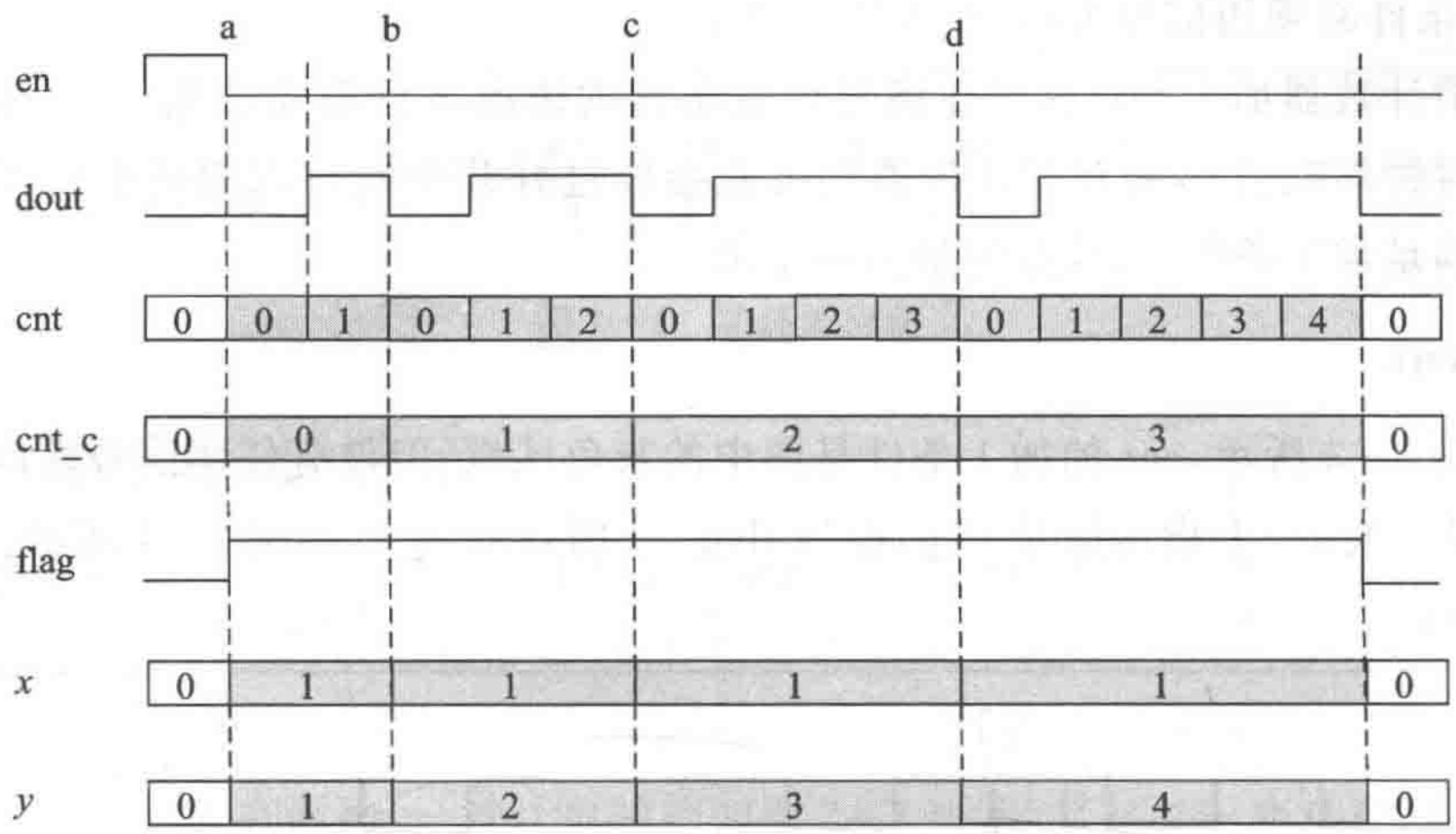


图 2-5 结束条件图

2.1.6 第五步：定义特殊点

定义特殊点是将要用到的特殊时刻点，如计数器结束点/触发点/开始点/中点等，根据需要挑选出来，并用信号来表示。图 2-5 中的 a、b、c、d 同时也是几个特殊点。

a：整个功能的开始点 en。

b：cnt 的结束条件 cnt==$x+y-1$，定为 end_cnt。

c：cnt_c 的结束条件 cnt_c==4-1，定为 end_cnt_c。

d：dout 变高的条件 cnt==$x-1$，定为 dout_l2h。

2.1.7 第六步：完整性检查

完整性检查是保证每个信号，包括计数器/添加的信号/输出信号的变化条件都已经清晰明确，变化条件必须用信号表示出来。当所有信号的变化条件都明确时，表示所有代码都可以写出来了。

1. cnt

cnt 的初值:0;

cnt 的加 1 条件:flag==1;

cnt 的结束值:计数至 cnt==$x+y-1$。

2. cnt_c

cnt_c 的初值:0;

cnt_c 的加 1 条件:end_cnt;

cnt_c 的结束值:cnt_c==4-1。

3. dout

dout 由 0 变 1:dout_l2h;

dout 由 1 变 0:end_cnt;

其他情况保持不变。

4. flag

flag 由 0 变 1:en==1;

flag 由 1 变 0:end_cnt_c;

其他情况不变。

5. x,y

当 cnt_c==0 时,$x=1$,$y=1$;

当 cnt_c==1 时,$x=1$,$y=2$;

当 cnt_c==2 时,$x=1$,$y=3$;

当 cnt_c==3 时,$x=1$,$y=4$。

2.1.8 第七步:计数器代码

计数器代码是按照明德扬计数器模板,写出计数器加 1 条件和结束条件代码(在 GVIM 下,输入 Jsg,回车,即可直接使用此模板)。

```
1  always  @(posedge clk or negedge rst_n)begin
2      if(rst_n == 1'b0)begin
3          cnt <= 0;
4      end
5      else if(add_cnt) begin
6          if(end_cnt)
```

```
            cnt <= 0;
        else
            cnt <= cnt + 1;
    end
end
assign  add_cnt = flag == 1'b1;
assign  end_cnt = add_cnt &&cnt == x + y - 1;

always  @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        cnt_c<= 0;
    end
    else if(add_cnt_c) begin
        if(end_cnt_c)
            cnt_c<= 0;
        else
            cnt_c<= cnt_c + 1;
    end
end

assign  add_cnt_c = end_cnt;
assign end_cnt_c = add_cnt_c&& cnt_c == 4 - 1;
```

assign 的形式，用一个信号表示满足此条件的时刻，如果后续用到这个条件，则均用该信号表示。这种信号复用的方法，明德扬称为计数器提取法。

2.1.9 第八步：功能代码

功能代码是按照完整性检查，写出计数器之外的代码。

```
//按照第六步第 3 点，写出 dout 代码
always  @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        dout<= 0;
    end
    else if(dout_12h)begin
            dout<= 1;
    end
    else if(end_cnt)begin
            dout<= 0;
```

```
11      end
12  end
13  assign dout_12h = add_cnt && cnt == x - 1;
14  //按照第六步第 4 点,写出 flag 代码
15  always  @(posedge clk or negedge rst_n)begin
16          if(rst_n == 1'b0)begin
17                  flag<= 0;
18          end
19          else if(en)begin
20                  flag<= 1;
21          end
22          else if(end_cnt_c)begin
23                  flag<= 0;
24          end
25  end
26
27  //按照第六步第 5 点,写出 x,y 代码
28  always  @( * )begin
29          if(cnt_c == 0)begin
30                  x = 1;
31                  y = 1;
32          end
33          else if(cnt_c == 1)begin
34                  x = 1;
35                  y = 2;
36          end
37          else if(cnt_c == 2)begin
38                  x = 1;
39                  y = 3;
40          end
41          else if(cnt_c == 3)begin
42                  x = 1;
43                  y = 4;
44          end
45          else begin
46                  x = 0;
47                  y = 0;
48          end
49  end
```

2.1.10 总 结

本节介绍了计数器架构设计八步法，在设计过程中用到了计数器框架法、计数器变量法、计数器提取法。在计数器架构中，计数器的设计是最重要的，几乎所有的设计都要使用计数器，如统计接收了多少数据、发送了多少数据、判断脉冲宽度等计数。在FPGA 设计中所有有关时间的内容都要通过计数器来实现，计时的本质是对时钟周期的计数，即以时钟周期为基准，通过计数多少个时钟周期来确定时间；或者是状态指示，在 FPGA 设计中常用的标志(flag)运用，如 led_flag 信号，led_flag 为 0 时将 led 熄灭，led_flag 为 1 时将 led 点亮，led_flag 其实也是一种计数器，其值只有两个，即 0 和 1。

本节内容看似简单，但要真正用好计数器却并非易事，请大家务必重视。如下节的项目实践，掌握计数器的设计，就能设计出 PWM、UART、VGA 等常用的接口应用了。

2.2 计数器项目实践

2.2.1 PWM 流水灯项目

脉冲宽度调制 PWM(Pulse Width Modelation)是利用微处理器的数字输出来对模拟电路进行控制的一种非常有效的技术，广泛应用于测量、通信、功率控制与变换的许多领域。

在本章的应用中可以认为 PWM 就是一种方波，如图 2-6 所示。

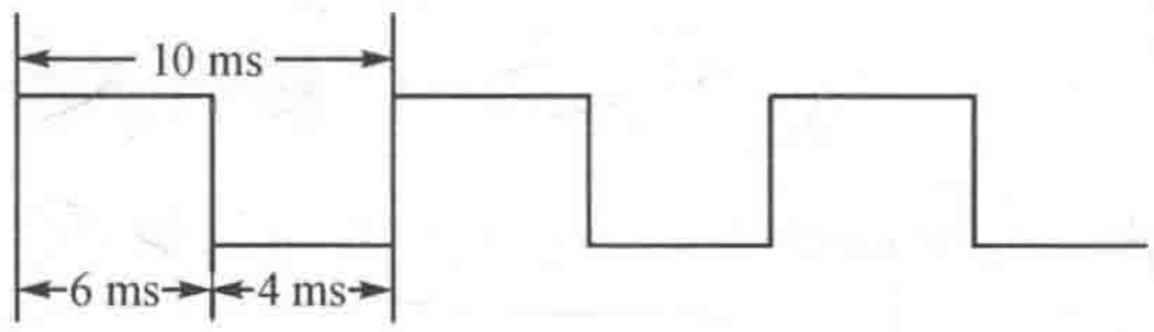


图 2-6 PWM 波形图

一个周期为 10 ms，即高电平为 6 ms、低电平为 4 ms 的 PWM，其占空比(高电平时间占整个周期的比例)为 60%。

明德扬的 FPGA 开发板共有 8 个 LED 灯。产生 8 个引脚的 PWM 图，如图 2-7 所示。每个引脚对应的占空比分别为 80%、70%、60%、50%、40%、30%、20%和 10%。系统工作时钟为 100 MHz。

FPGA 通过 8 个引脚分别控制 8 个 LED 灯，引脚值为 0，对应的 LED 灯亮；引脚值为 1，对应的 LED 灯灭(低亮高灭)。如果引脚不停地变化，则 LED 灯会闪烁；如果这种高低变化非常快，由于人的视觉暂留现象，LED 就会出现不同的亮度。

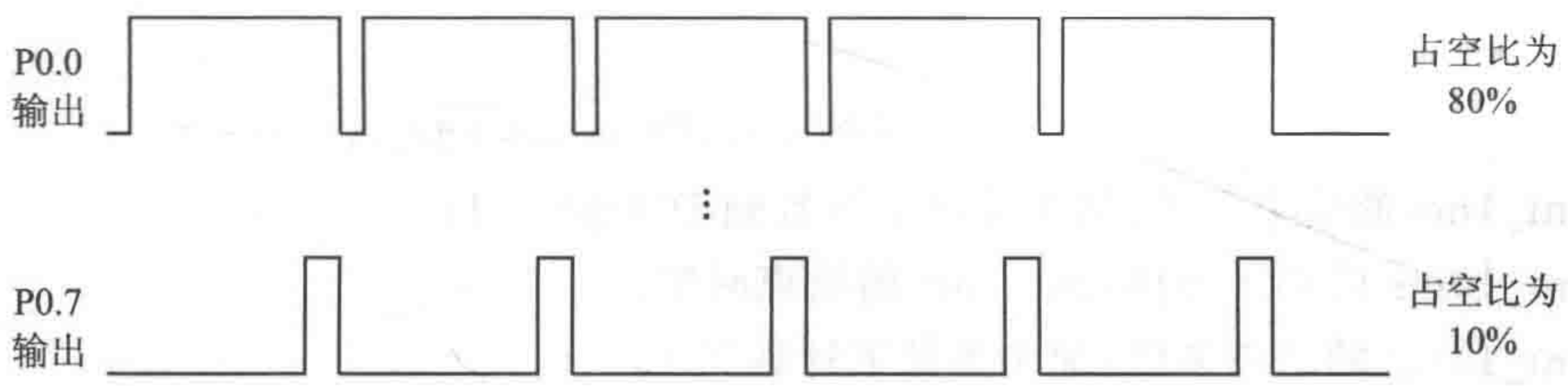


图 2－7　8 个 LED 灯对应的 PWM 波形

基于这个原则，也可以通过产生 PWM 波形，来控制 LED 灯的亮度。

1. 明确功能

确定模块信号列表，如表 2－2 所列。

表 2－2　LED 信号列表

信号名	I/O	位　宽	说　明
clk	I	1	系统工作时钟 100 MHz
rst_n	I	1	系统复位信号，低电平有效
led	O	8	LED 输出信号

产生 8 个脉冲，每个脉冲周期为 10 ms 对应的占空比分别为 80%、70%、60%、50%、40%、30%、20%和 10%。

2. 功能波形

LED 所有信号的变化都是相似的，这里以 led[0]为例，见图 2－8。

图 2－8　led[0]信号变化图

3. 计数结构

因为每个脉冲的高低电平持续时间都是以 1 ms 为单位的，所以引入两个计数器，计数器 cnt_1ms 计数 1 ms，计数器 cnt_10ms 计数每个脉冲高低电平分别持续的时间。具体计数情况如图 2－9 所示。

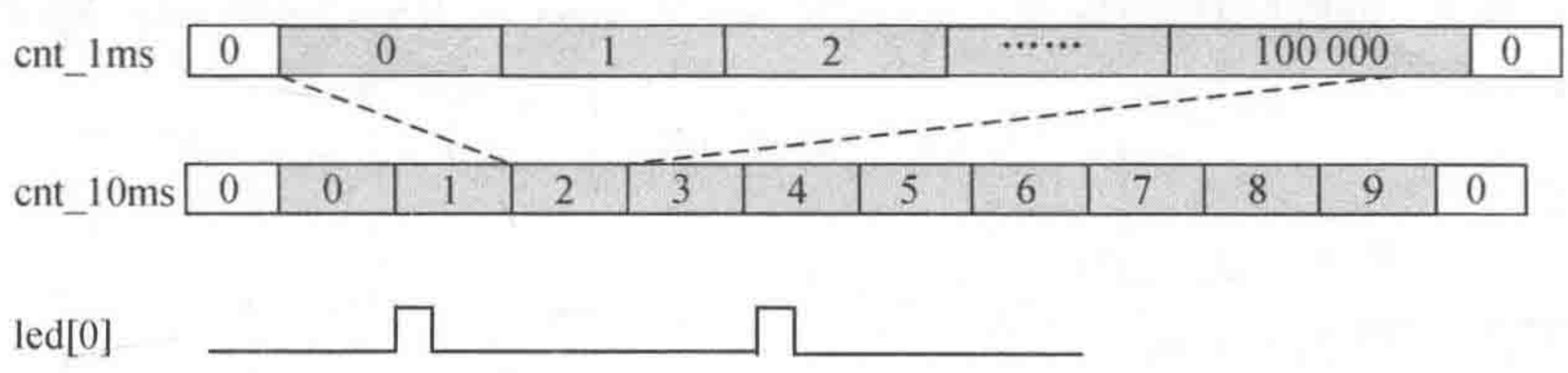


图 2－9　LED 计数结构图

4. 加 1 和结束条件

cnt_1ms 的加 1 条件：计数器一直在计数，即 assign add_cnt_1ms=1；

cnt_1ms 的结束条件：加 1 条件下计数到 100 000－1；

cnt_10ms 的加 1 条件：cnt_1ms 的结束时刻；

cnt_10ms 的结束条件：加 1 条件下计数到 10－1。

5. 定义特殊点

有几个特殊点需要记住。

cnt_1ms 的结束条件：cnt_1ms==100 000－1，定为 end_cnt_1ms。

cnt_10ms 的结束条件：cnt_10ms==10－1，定为 end_cnt_10ms。

LED 所有位的信号变化都是相似的，所以以 led[0]为例定义特殊点。

led[0]变高的条件：cnt_10ms==1－1，定为 led0_off。

led[0]变低的条件：cnt_10ms==10－1，定为 led_on。

6. 完整性检查

(1) 计数器 cnt_1ms

cnt_1ms 的初值：0；

cnt_1ms 的加 1 条件：assign add_cnt_1ms=1；

cnt_1ms 的结束条件：assign end_cnt_1ms=add_cnt_1ms && cnt_1ms == 100 000 000－1。

(2) 计数器 cnt_10ms

cnt_10ms 初值：0；

cnt_10ms 的加 1 条件：assign add_cnt_10ms=end_cnt_1ms；

cnt_10ms 的结束条件：assign end_cnt_10ms=add_cnt_10ms && cnt_10ms==10－1。

(3) led[0]

led[0]由 0 变 1：led0_off；

led[0]由 1 变 0：led_on。

(4) led[1]

led[1]由 0 变 1：led1_off；

led[1]由 1 变 0：led_on。

(5) led[2]

led[2]由 0 变 1：led2_off；

led[2]由 1 变 0：led_on。

(6) led[3]

led[3]由 0 变 1：led3_off；

led[3]由 1 变 0:led_on。

(7) led[4]

led[4]由 0 变 1:led4_off;

led[4]由 1 变 0:led_on。

(8) led[5]

led[5]由 0 变 1:led5_off;

led[5]由 1 变 0:led_on。

(9) led[6]

led[6]由 0 变 1:led6_off;

led[6]由 1 变 0:led_on。

(10) led[7]

led[7]由 0 变 1:led7_off;

led[7]由 1 变 0:led_on。

7. 计数器代码

```
parameter  TIME_1MS   = 100_000;
always  @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        cnt_1ms <= 0;
    end
    else if(add_cnt_1ms)begin
        if(end_cnt_1ms)
            cnt_1ms <= 0;
        else
            cnt_1ms <= cnt_1ms + 1;
    end
end

assign add_cnt_1ms = 1'b1;
assign end_cnt_1ms = add_cnt_1ms && cnt_1ms == TIME_1MS - 1;

always  @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        cnt_10ms <= 0;
    end
    else if(add_cnt_10ms)begin
        if(end_cnt_10ms)
            cnt_10ms <= 0;
        else
```

```
            cnt_10ms <= cnt_10ms + 1;
        end
end

assign add_cnt_10ms = end_cnt_1ms;
assign end_cnt_10ms = add_cnt_10ms && cnt_10ms == 10 - 1;
```

8. 完整代码

```
//按照第六步第 3 点,写出 led[0]的代码
always  @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        led[0] <= 0;
    end
    else if(led_on)begin
        led[0] <= 0;
    end
    else if(led0_off)begin
        led[0] <= 1;
    end
end
assign led0_off = add_cnt_10ms && cnt_10ms == 1 - 1;
assign led_on   = add_cnt_10ms && cnt_10ms == 10 - 1;

//按照第六步第 4 点,写出 led[1]的代码
always  @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        led[1] <= 0;
    end
    else if(led_on)begin
        led[1] <= 0;
    end
    else if(led1_off)begin
        led[1] <= 1;
    end
end
assign led1_off = add_cnt_10ms && cnt_10ms == 2 - 1;

//按照第六步第 5 点,写出 led[2]的代码
always  @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        led[2] <= 0;
```

```
34        end
35        else if(led_on)begin
36            led[2] <= 0;
37        end
38        else if(led2_off)begin
39            led[2] <= 1;
40        end
41    end
42    assign led2_off = add_cnt_10ms && cnt_10ms == 3 - 1;
43
44    //按照第六步第 6 点,写出 led[3]的代码
45    always  @(posedge clk or negedge rst_n)begin
46        if(rst_n == 1'b0)begin
47            led[3] <= 0;
48        end
49        else if(led_on)begin
50            led[3] <= 0;
51        end
52        else if(led3_off)begin
53            led[3] <= 1;
54        end
55    end
56    assign led3_off = add_cnt_10ms && cnt_10ms == 4 - 1;
57
58    //按照第六步第 7 点,写出 led[4]的代码
59    always  @(posedge clk or negedge rst_n)begin
60        if(rst_n == 1'b0)begin
61            led[4] <= 0;
62        end
63        else if(led_on)begin
64            led[4] <= 0;
65        end
66        else if(led4_off)begin
67            led[4] <= 1;
68        end
69    end
70    assign led4_off = add_cnt_10ms && cnt_10ms == 5 - 1;
71
72    //按照第六步第 8 点,写出 led[5]的代码
73    always  @(posedge clk or negedge rst_n)begin
74        if(rst_n == 1'b0)begin
75            led[5] <= 0;
```

```
76      end
77      else if(led_on)begin
78          led[5] <= 0;
79      end
80      else if(led5_off)begin
81          led[5] <= 1;
82      end
83  end
84  assign led5_off = add_cnt_10ms && cnt_10ms == 6 - 1;
85
86  //按照第六步第 9 点,写出 led[6]的代码
87  always  @(posedge clk or negedge rst_n)begin
88      if(rst_n == 1'b0)begin
89          led[6] <= 0;
90      end
91      else if(led_on)begin
92          led[6] <= 0;
93      end
94      else if(led6_off)begin
95          led[6] <= 1;
96      end
97  end
98  assign led6_off = add_cnt_10ms && cnt_10ms == 7 - 1;
99
100 //按照第六步第 10 点,写出 led[7]的代码
101 always  @(posedge clk or negedge rst_n)begin
102     if(rst_n == 1'b0)begin
103         led[7] <= 0;
104     end
105     else if(led_on)begin
106         led[7] <= 0;
107     end
108     else if(led7_off)begin
109         led[7] <= 1;
110     end
111 end
112 assign led7_off = add_cnt_10ms && cnt_10ms == 8 - 1;
```

2.2.2 UART 接口项目

UART(Universal Asynchronous Receiver/Transmitter)是指异步串行接收/发送。UART 是一个并行输入转换为串行输出的接口,通常集成在主板上。两个设备之间的通信可以通过一收一发两根线实现,见图 2-10。

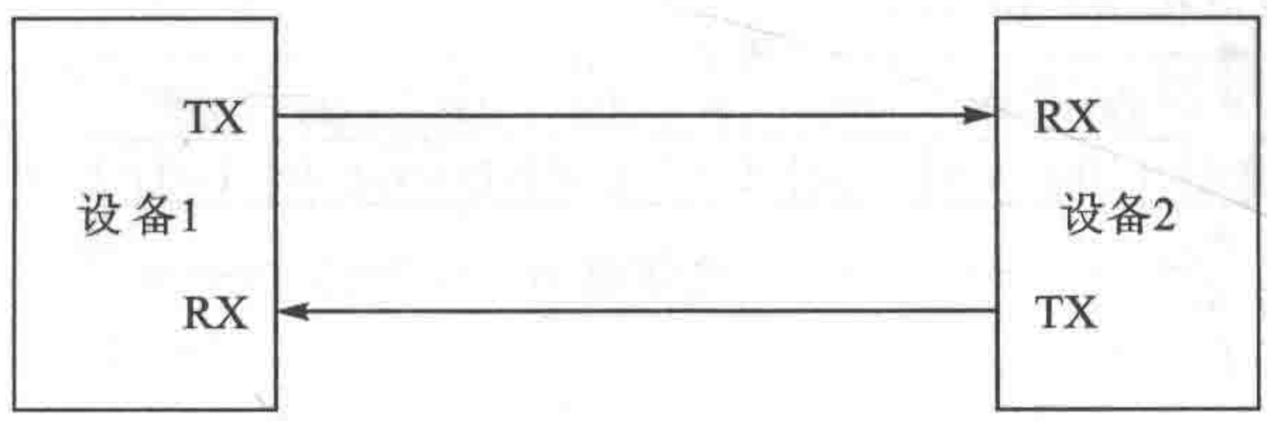


图 2－10　UART 传输结构图

对于接口模块设计来说，了解它的时序图是必要的，UART 传输时序图如图 2－11 所示。

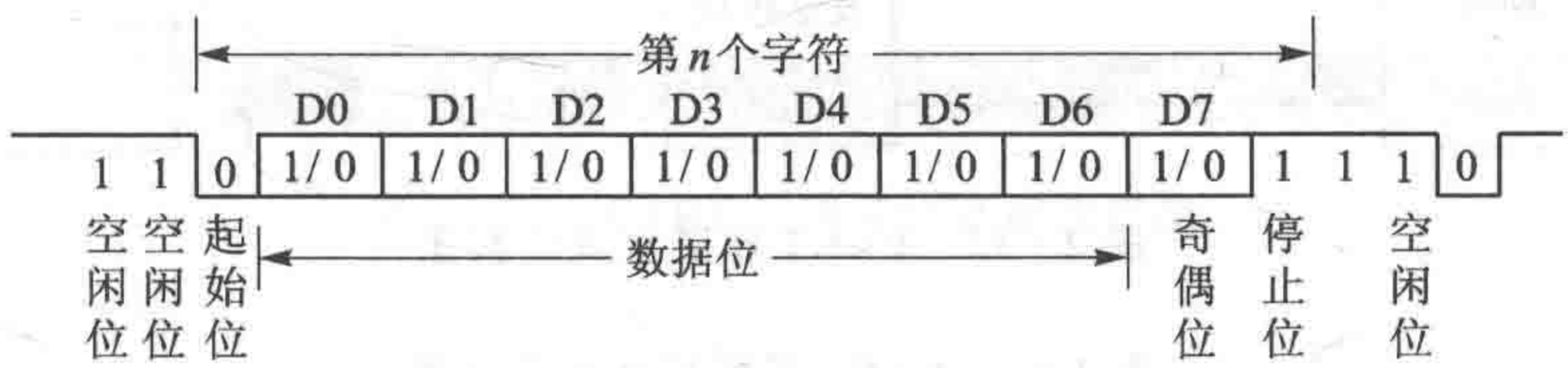


图 2－11　UART 传输时序图

空闲位：UART 接口无数据传输；

起始位：字符帧的起始位置，占 1 bit；

数据位：字符数据的内容，大小由数据位指定；

奇偶位：用于传输奇偶校验位，占 1 bit；

停止位：字符帧的终止位，附加于末尾，本项目使用 1 bit。

此外，要完成 UART 模块设计，还要理解波特率。所谓波特率（Baud rate），在本项目中可以通俗地理解为一个设备在 1 s 内发送/接收了多少比特的数据，项目要求波特率为 9 600 bit/s，得到所需要的时钟周期数（系统时钟为 50 MHz）计算过程如下：

$$\text{时钟周期数}=\frac{1}{9\ 600\ \text{bit/s}}\times 50\ \text{MHz}=5\ 208$$

1. UART 接口发送

模块的工作时钟为 50 MHz，在空闲时（tx_rdy＝＝1），每当收到 tx_vld＝＝1 时，按照以下格式将 tx_data 发送给 PC。波特率为 9 600 bit/s，起始位 1 位，数据位 8 位，没有奇偶位，停止位 1 位，如图 2－12 所示。

（1）明确功能

当 tx_vld 有效时，接口模块通过 uart_tx 把 tx_data 的数据传给 PC 端，如图 2－13 所示，表 2－3 为 UART 发送模块信号列表。

图 2-12　UAR 时序图

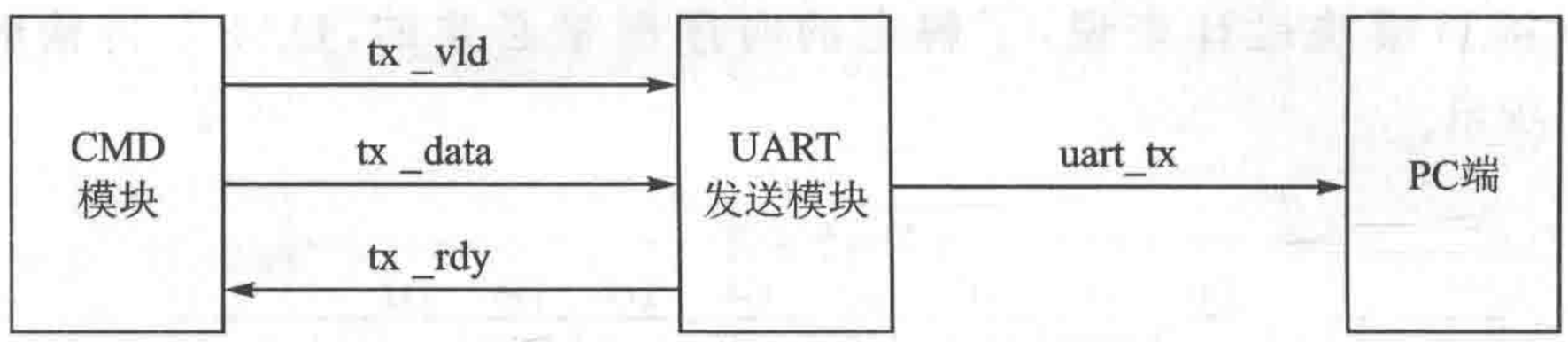


图 2-13　UART 发送模块结构图

表 2-3　UART 发送模块信号列表

信号名	I/O	位　宽	说　明
clk	I	1	系统工作时钟 50 MHz
rst_n	I	1	系统复位信号，低电平有效
tx_vld	I	1	数据有效指示信号，高电平有效
tx_data	I	8	数据输入信号
uart_tx	O	1	UART 接口数据输出信号
tx_rdy	O	1	模块忙闲指示信号，表示发送模块是否在传输过程中，在 UART 接口数据发送的过程中，为低电平，且此时不能响应发送数据有效指示信号 tx_vld；在空闲位时，为高电平，此时可响应 tx_vld

在进一步设计 UART 发送接口前，先对 rdy 信号进行详细的讲解。此信号在很多场合都会用到，后续章节的项目中也会反复用到，希望读者透彻理解以下内容：

如图 2-14 所示，如果用时序逻辑产生 rdy，那么 rdy 就在收到第一个 en(a)后的 clk 上升沿拉低，此时上游模块会认为本模块仍然空闲，可能接着发出第二个使能 en

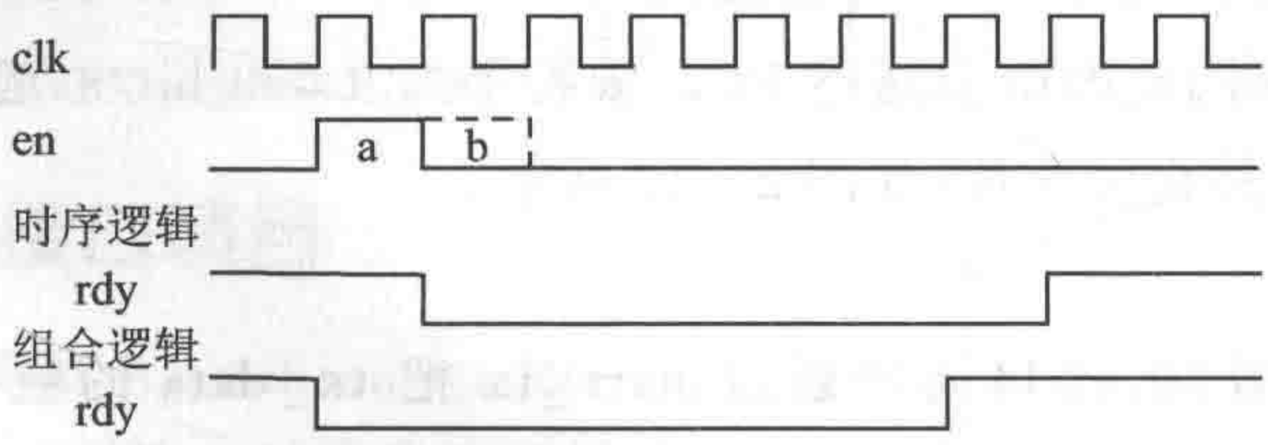


图 2-14　rdy 信号详解

(b)，但由于本模块正在处理执行 en(a)的操作，不能响应 en(b)，因此 en(b)的操作就被丢弃了，从而使数据传输出现错误。但是，用组合逻辑就可以避免这个问题，在收到 en(a)时，rdy 立即拉低，上游模块在下一个时钟上升沿知道本模块正忙，故不会发送 en(b)。因此 rdy 必须用组合逻辑产生。

(2) 功能波形

UART 发送模块功能波形如图 2－15 所示。

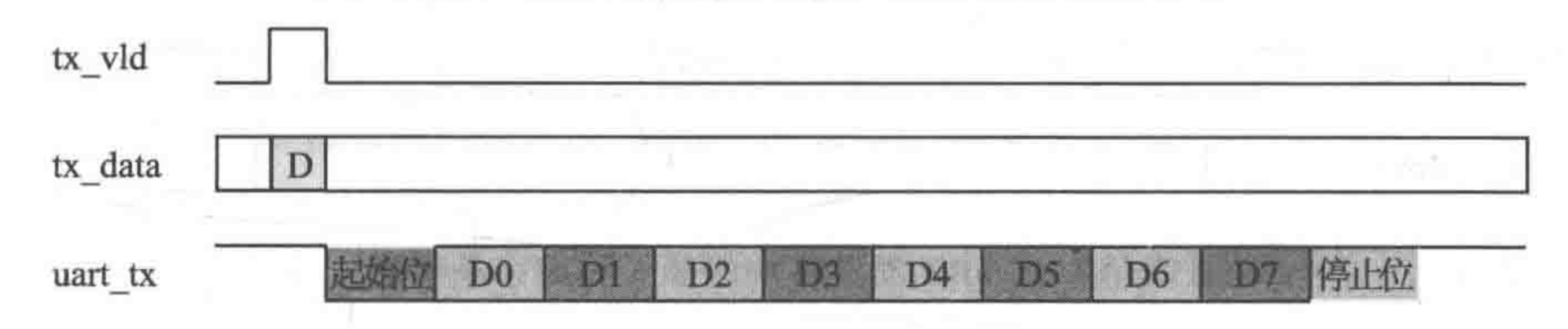


图 2－15　UART 发送模块功能波形图

(3) 计数结构

当收到 tx_vld 为 1 时，uart_tx 会产生起始位到停止位的输出。输出的数据来自 tx_vld 为 1 期间 tx_data 的值 D。

引入一个计数器 cnt1 来计数 uart_tx 每一位的宽度。引入另一个计数器 cnt2 计数 uart_tx 输出的是起始位到停止位之间的哪一位。UART 接收模块计数结构如图 2－16 所示。

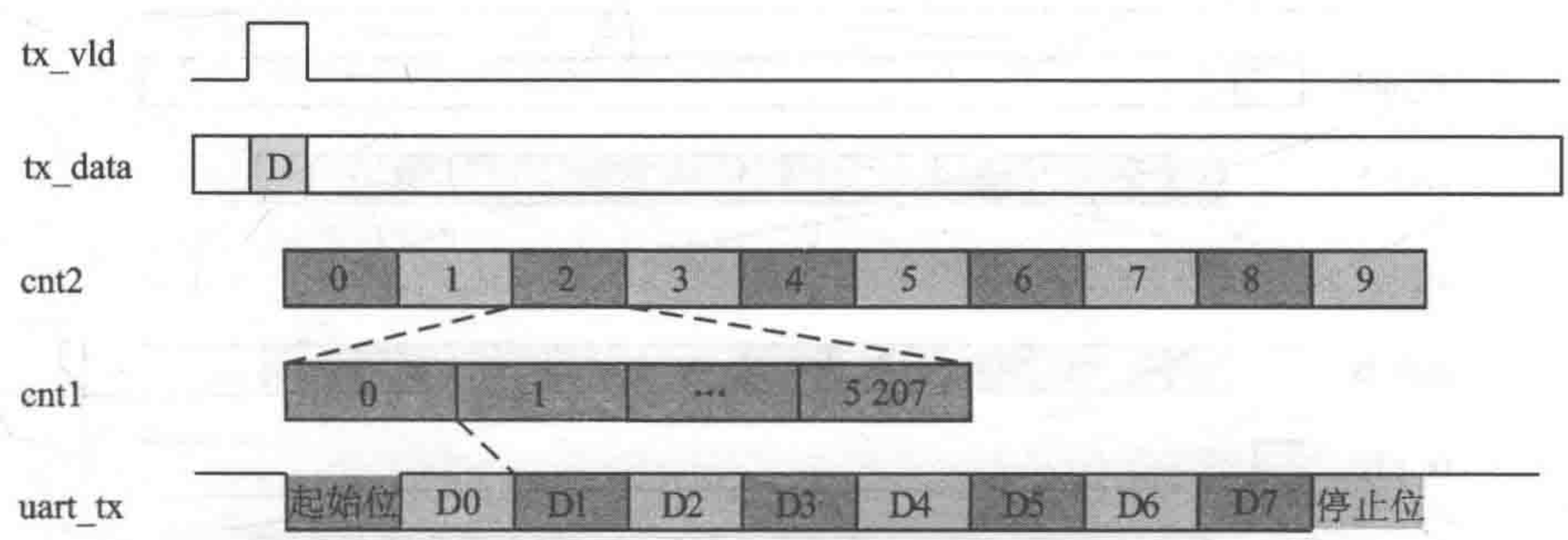


图 2－16　UART 发送模块计数结构图

(4)加 1 和结束条件

UART 发送模块的加 1 和结束条件图如图 2－17 所示。

cnt1 的加 1 条件：从现有的信号无法知道 cnt1 的加 1 条件，所以添加一个信号 flag 用以指示 cnt1 的加 1 区间，当 flag==1 时 cnt1 加 1；

cnt2 的加 1 条件：cnt1 的结束时刻；

cnt1 的结束条件：cnt1==5 208－1；

cnt2 的结束条件：cnt2 计数至 10－1。

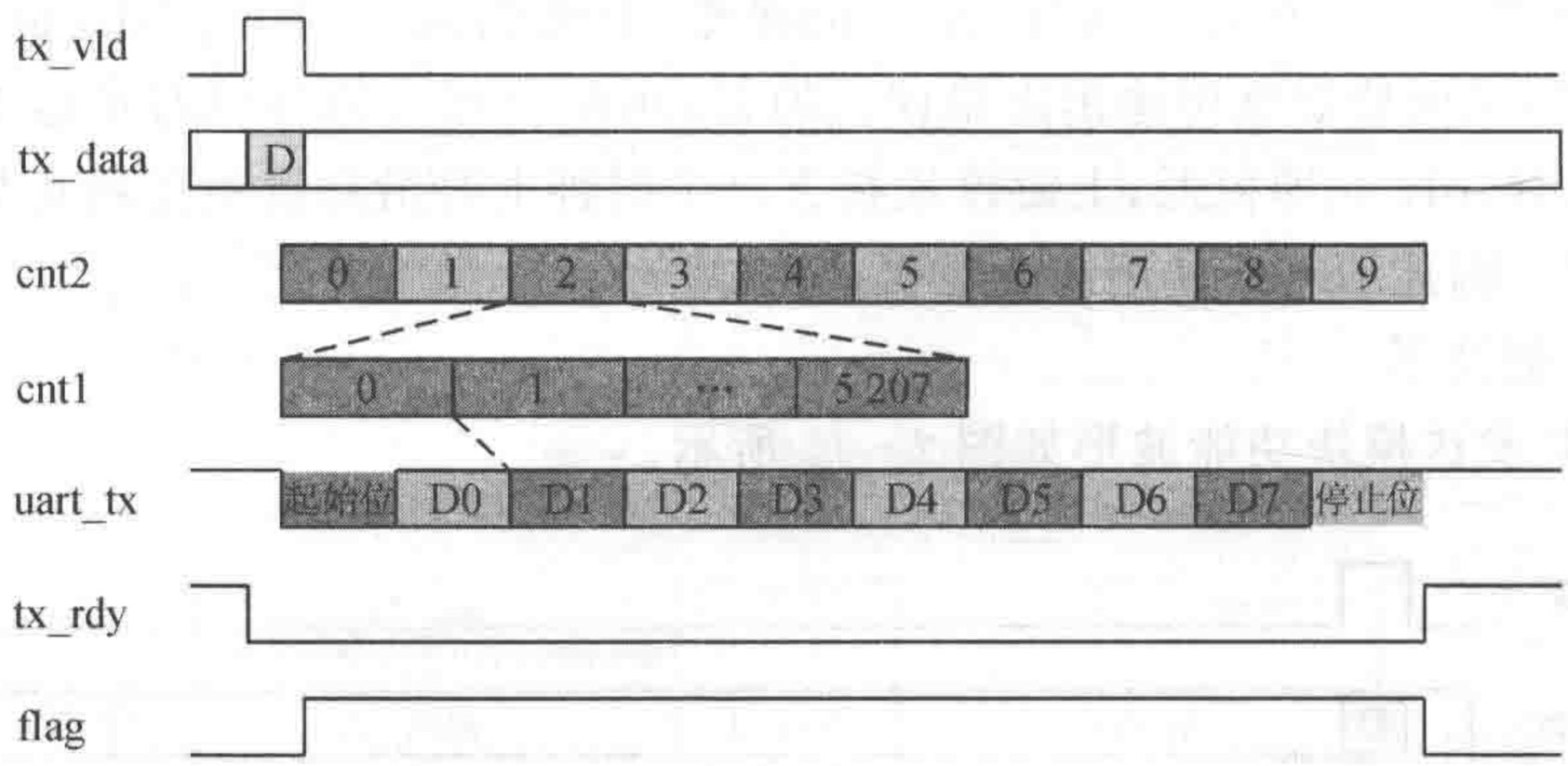


图 2-17　UART 发送模块的加 1 和结束条件图

(5) 定义特殊点

图 2-18 中有几个特殊点需要记住。

整个功能的开始点:tx_vld;

cnt1 的结束条件:cnt1==5208-1,定为 end_cnt1;

cnt2 的结束条件:cnt2==10-1,定为 end_cnt2;

tx 产生输出的点 a:cnt==0 && flag==1;

flag 变高的点: tx_vld==1。

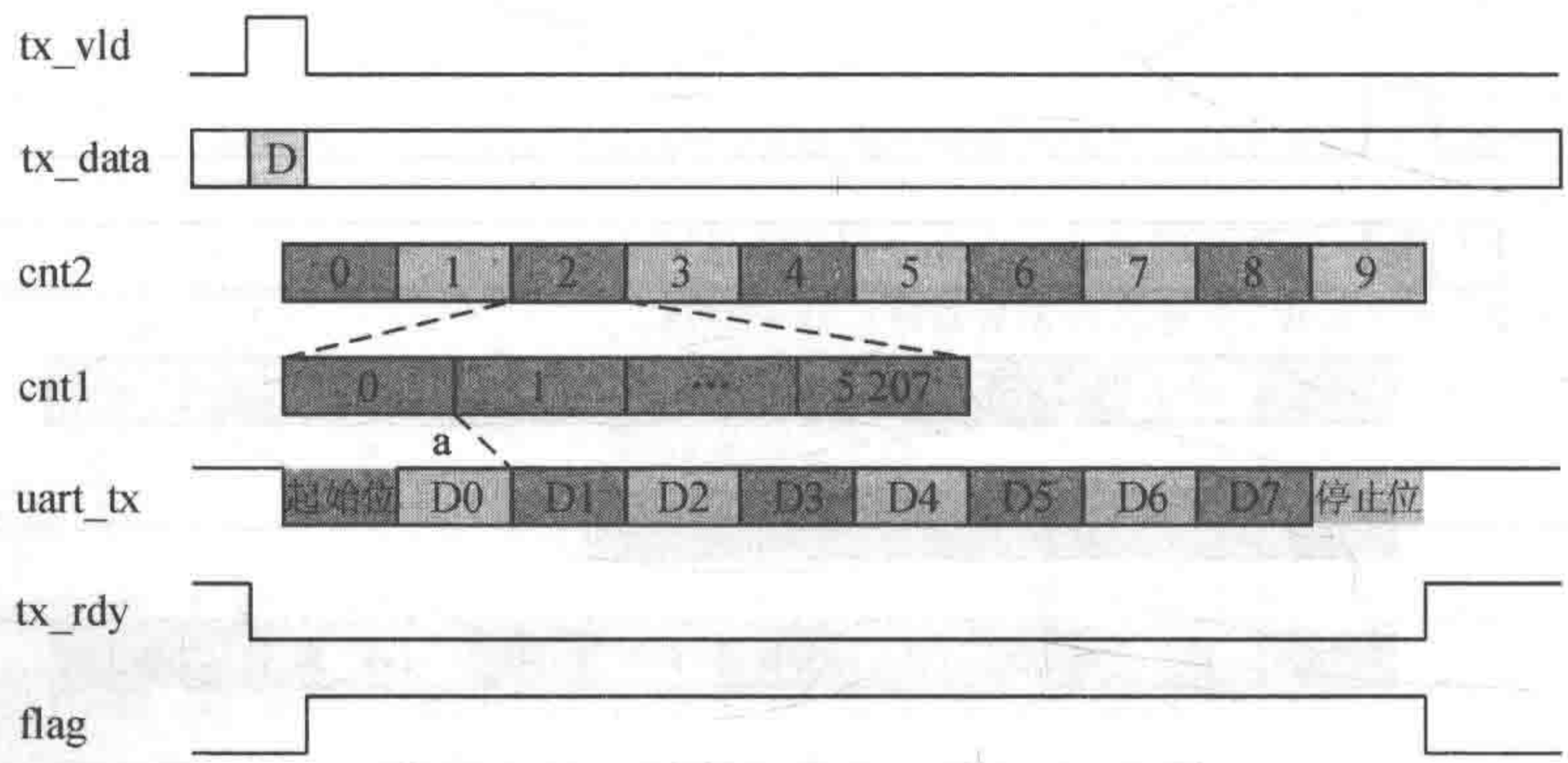


图 2-18　UART 发送模块定义特殊点

(6) 完整性检查

1) 计数器 cnt1

cnt1 的初值:0;

cnt1 的加 1 条件:flag==1;

cnt1 的结束值:计数至 cnt1==5 208-1。

2) 计数器 cnt2

cnt2 的初值:0;

cnt2 的加 1 条件：end_cnt1；

cnt2 的结束值：计数至 cnt2==10－1 。

3）flag

flag 的初值：0；

flag 由 0 变为 1：tx_vld==1 且 rdy==1；

flag 由 1 变为 0：end_cnt2。

4）tx_data_tmp

当 tx_vld==1 且 rdy==1 时，tx_data_tmp 变为 1'b1（停止位）＋ tx_data（数据位）＋ 1'b0（起始位）。

5）uart_tx

在 cnt1 的加 1 条件下，当 cnt1==0 时，uart_tx 依次从低位到高位输出 tx_data_tmp。

6）tx_rdy

tx_rdy 的初值：1；

tx_rdy 由 1 变为 0：flag==1 或者 tx_vld==1；

其他时刻 tx_rdy 为 1。

（7）计数器代码

```
1  always @(posedge clk or negedge rst_n)begin
2      if(rst_n == 1'b0)begin
3          cnt1 <= 0;
4      end
5      else if(add_cnt1)begin
6          if(end_cnt1) begin
7              cnt1 <= 0;
8          end
9          else begin
10             cnt1 <= cnt1 + 1;
11         end
12     end
13 end
14
15 assign add_cnt1 = flag == 1;
16 assign end_cnt1 = add_cnt1&&cnt1 == 5208 - 1;
17
18 always @(posedge clk or negedge rst_n)begin
19     if(rst_n == 1'b0)begin
20         cnt2 <= 0;
21     end
22     else if(add_cnt2)begin
23         if(end_cnt2) begin
24             cnt2 <= 0;
25         end
26         else begin
27             cnt2 <= cnt2 + 1;
28         end
```

```
    end
end

assign add_cnt2 = end_cnt1;
assign end_cnt2 = add_cnt2 && cnt2 == 10 - 1;
```

(8) 功能代码

```
//按照第六步第 3 点,写出 flag 的代码
always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        flag <= 0;
    end
    else if(tx_start)begin
        flag <= 1;
    end
    else if(end_cnt2)begin
        flag <= 0;
    end
end
assign  tx_start = tx_vld && ! flag;

//按照第六步第 4 点,写出 tx_data_tmp 的代码
always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        tx_data_tmp <= 0;
    end
    else if(tx_start)begin
        tx_data_tmp <= {1'b1,tx_data,1'b0};
    end
end

//按照第六步第 5 点,写出 uart_tx 的代码
always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        uart_tx <= 0;
    end
    else if(add_cnt1 && cnt1 == 0)begin
        uart_tx <= tx_data_tmp[cnt2];
    end
end

//按照第六步第 6 点,写出 tx_rdy 的代码
always  @( * )begin
    if(flag || tx_vld)
        tx_rdy = 0;
    else
        tx_rdy = 1;
end
```

2. UART 接口接收

模块的工作时钟为 50 MHz，在空闲时，如果接收到 uart_rx 的下降沿，开始接收数据，将接收到的数据串转换到 rx_data。接收完后，将 rx_vld 拉高 1 个高电平脉冲。波特率为 9 600 bit/s，起始位 1 位，数据位 8 位，没有奇偶位，停止位 1 位，见图 2－12。

（1）明确功能

发送模块有 rdy，是由于上游模块工作频率为 50 MHz，产生 vld 非常快，而发送模块接收到 1 个 vld 后，需要很多时间来处理，所以需要 rdy 信号来告知上游模块等待。而现在接收模块没有此类问题，所以不需要 rdy 信号。

rx_vld 不仅可以表示数据有效，同时也表示接收数据的个数。例如下游模块检测到 rx_vld＝1，说明有串口接收到 1 个 8 位的数据，检测到多少个 rx_vld＝1，就表示接收到了多少个 8 位数据。

因此得到 UART 接收模块结构图如图 2－19 所示。UART 接收模块信号列表如表 2－4 所列。

图 2－19　UART 接收模块结构图

表 2－4　UART 接收模块信号列表

信号名	I/O	位　宽	说　明
clk	I	1	系统工作时钟 50 MHz
rst_n	I	1	系统复位信号，低电平有效
uart_rx	I	1	UART 接口数据输入信号
rx_vld	O	1	数据有效指示信号，高电平有效
rx_data	O	8	数据输出信号

（2）功能波形

UART 接收模块功能波形如图 2－20 所示。

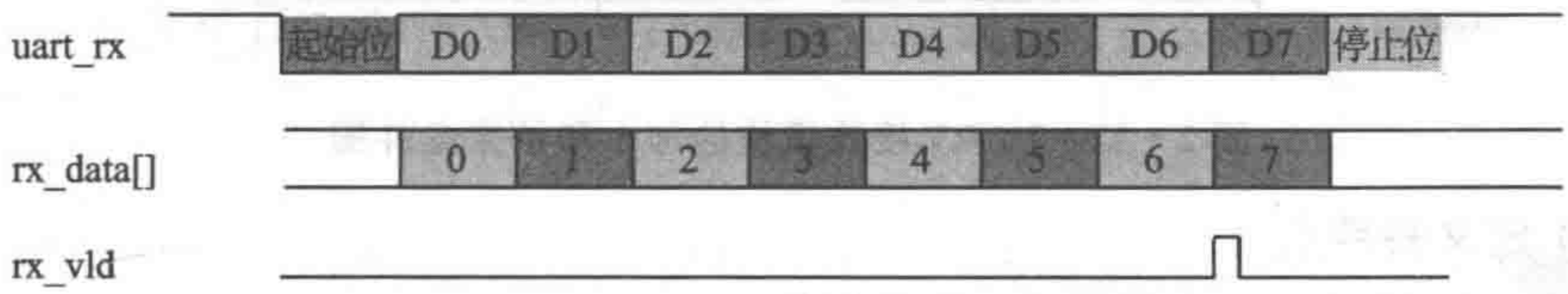


图 2－20　UART 接收模块功能波形图

(3) 计数结构

复位后，当 uart_rx 为下降沿时，就开始接收数据给 rx_data[]。其中波特率为 9 600 bit/s，D7 给 rx_data[7]，依此类推，D0 给 rx_data[0]。接收完毕后，又等待 uart_rx 的下降沿。

引入一个计数器 cnt1 来计数 uart_rx 每一位的宽度。引入另一个计数器 cnt2 计数 uart_rx 输入的是起始位到停止位之间的哪一位。UART 接收模块计数结构如图 2-21 所示。

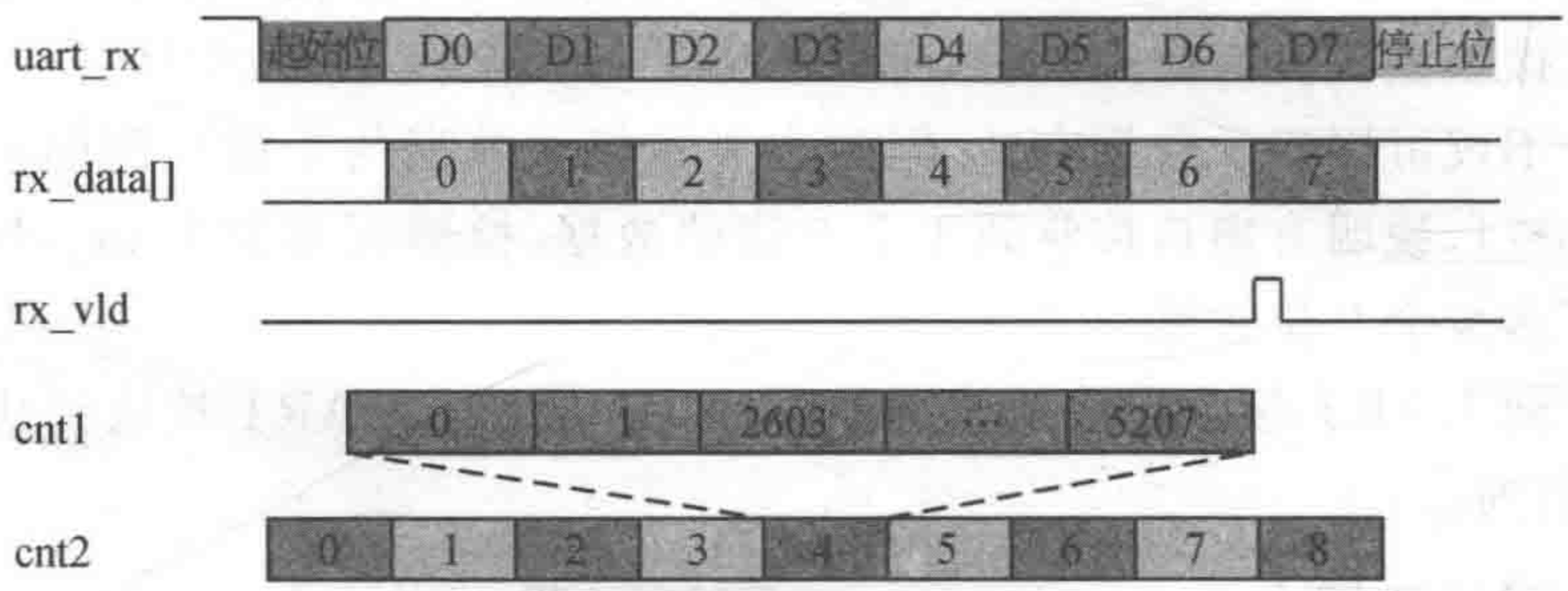


图 2-21 UART 接收模块计数结构图

(4) 加 1 和结束条件

cnt1 的加 1 条件：在现有的信号中无法知道 cnt1 的加 1 条件，所以增加一个信号 flag 用于指示 cnt1 的加 1 条件，当 flag==1 时，cnt1 加 1；

cnt2 的加 1 条件：cnt1 的结束时刻；

cnt1 的结束条件：cnt1==5 208−1；

cnt2 的结束条件：cnt2 计数至 8−1。

UART 接收模块的加 1 和结束条件如图 2-22 所示。

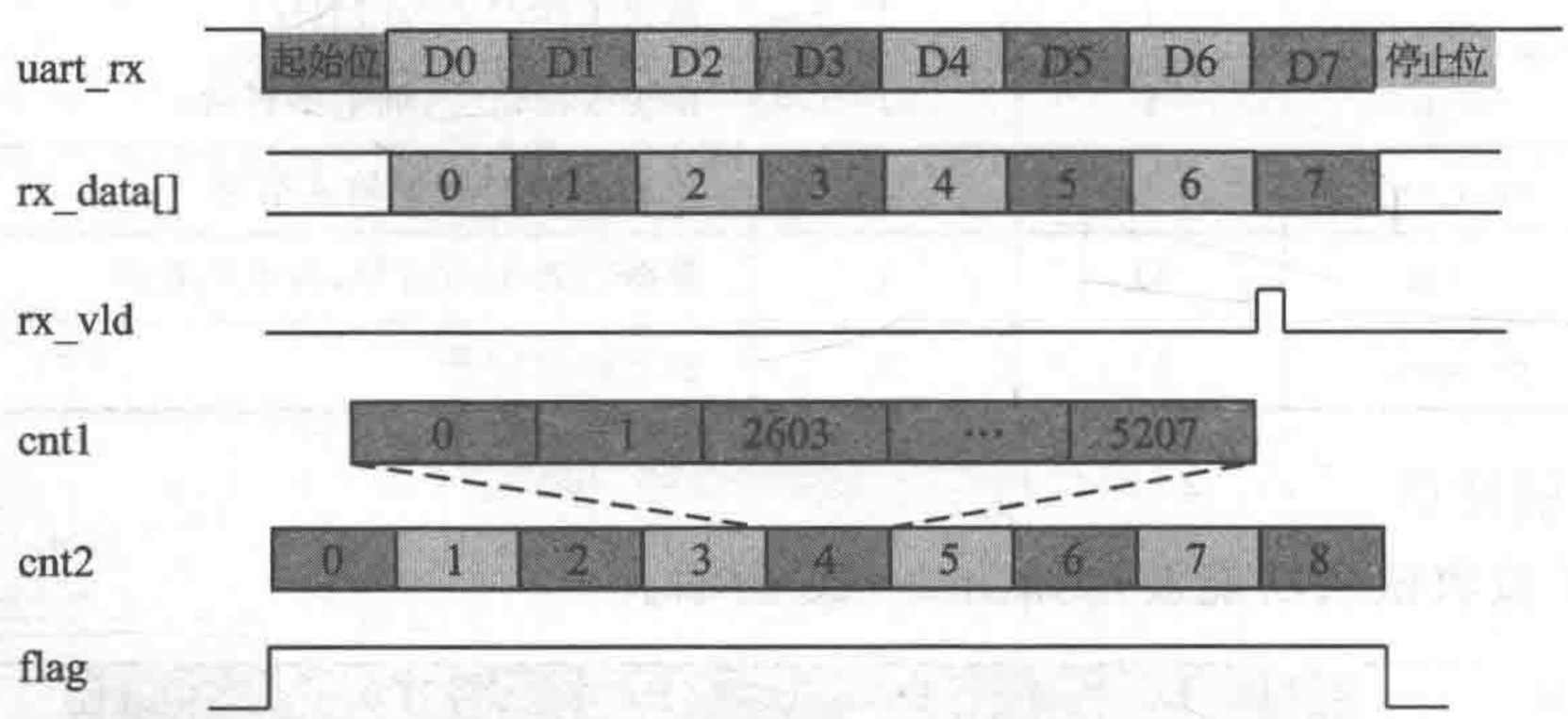


图 2-22 UART 接收模块的加 1 和结束条件图

(5) 定义特殊点

图 2-23 所示为几个特殊点，需要记住。

整个功能的开始点：uart_tx 的下降沿；

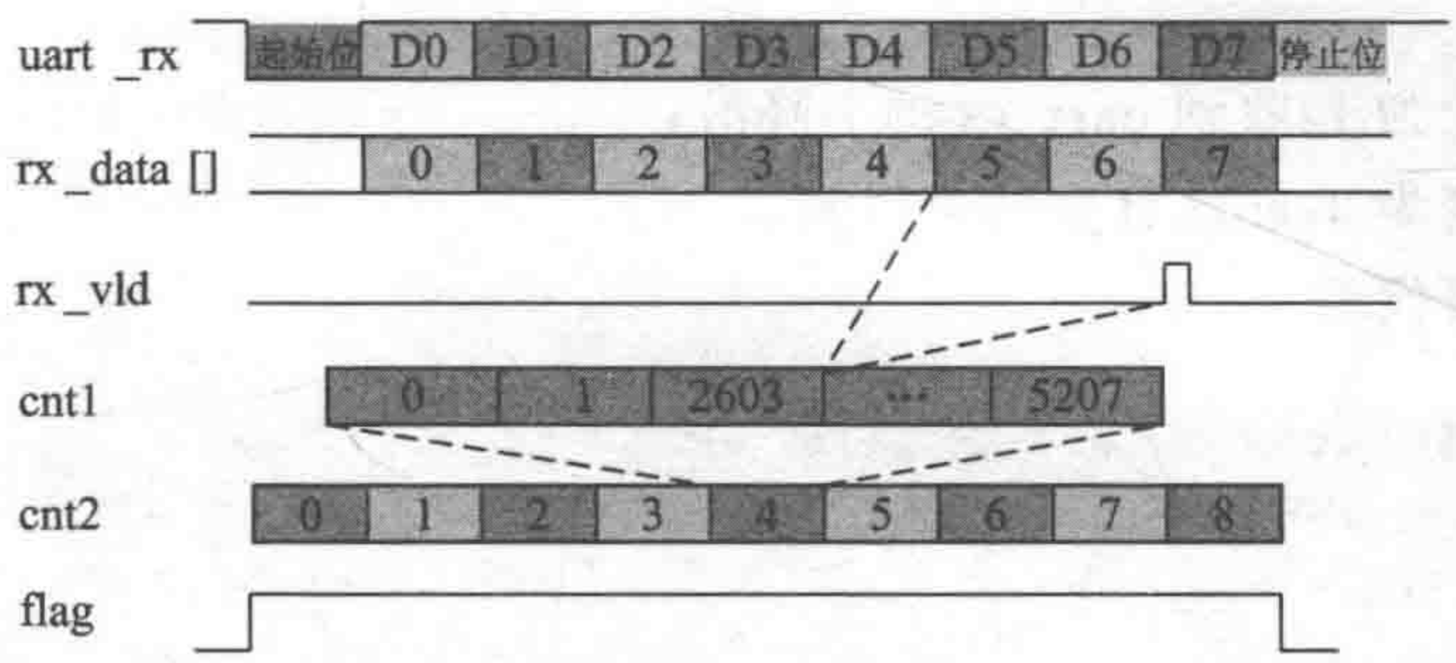


图 2-23　UART 接收模块定义特殊点图

cnt1 的结束条件：cnt1==5 208−1，定为 end_cnt1；

cnt2 的结束条件：cnt2==9−1，定为 end_cnt2；

flag 变高的点：uart_tx 的下降沿；

rx_data 的变化点：cnt2>=1&&cnt2<9&&add_cnt1&&cnt1==2 604−1；

rx_vld 的变高点：cnt2==8&&add_cnt1&&cnt1==2 604−1。

(6) 完整性检查

1) 计数器 cnt1

cnt1 的初值：0；

cnt1 的加 1 条件：flag==1；

cnt1 的结束值：计数至 cnt1==5 208−1。

2) 计数器 cnt2

cnt2 的初值：0；

cnt2 的加 1 条件：cnt1 的结束时刻；

cnt2 的结束值：计数至 cnt2==9−1。

3) uart_rx

因 uart_rx 是由 PC 端发送过来的信号，其对于 FPGA 来说是异步信号。

对 uart_rx 进行异步时序处理：把 uart_rx 打两拍，产生 uart_rx_ff0，uart_rx_ff1；

检测 uart_rx_ff1 的下降沿：把同步后的 uart_rx_ff1 打一拍，产生 uart_rx_ff2；

uart_rx_ff1 的下降沿：uart_rx_ff1==0&& uart_rx_ff2==1。

4) rx_data

当收到 uart_rx 的下降沿时，就开始接收数据给 rx_data。其中，D7 给 rx_data[7]，依此类推，D0 给 rx_data[0]。给 rx_data 赋值时是在 uart_rx 每一位的中间时刻。

5) rx_vld

rx_vld 由 0 变 1：在赋值完成后拉高，cnt2==8 && add_cnt1 && cnt1==2 604−1；

其他时刻均为 0。

6）flag

flag 由 0 变为 1：收到 uart_rx 的下降沿；

flag 由 1 变为 0：cnt2 计数结束时刻。

（7）计数器代码

```
1  always @(posedge clk or negedge rst_n)begin
2      if(rst_n == 1'b0)begin
3          cnt1 <= 0;
4      end
5      else if(add_cnt1)begin
6          if(end_cnt1) begin
7              cnt1 <= 0;
8          end
9          else begin
10             cnt1 <= cnt1 + 1;
11         end
12     end
13 end
14
15 assign add_cnt1 = flag == 1;
16 assign end_cnt1 = add_cnt1&&cnt1 == 5208 - 1;
17
18 always @(posedge clk or negedge rst_n)begin
19     if(rst_n == 1'b0)begin
20         cnt2 <= 0;
21     end
22     else if(add_cnt2)begin
23         if(end_cnt2)begin
24             cnt2 <= 0;
25         end
26         else begin
27             cnt2 <= cnt2 + 1;
28         end
29     end
30 end
31
32 assign add_cnt2 = end_cnt1;
33 assign end_cnt2 = add_cnt2&&cnt2 == 9 - 1;
```

（8）完整代码

```
1  //按照第六步第 3 点，写出 uart_rx 的代码
2  always  @(posedge clk or negedge rst_n)begin
3      if(rst_n == 1'b0)begin
```

```
4            uart_rx_ff0 <= 1;
5            uart_rx_ff1 <= 1;
6            uart_rx_ff2 <= 1;
7        end
8        else begin
9            uart_rx_ff0 <= uart_rx;
10           uart_rx_ff1 <= uart_rx_ff0;
11           uart_rx_ff2 <= uart_rx_ff1;
12       end
13 end
14
15 //按照第六步第 4 点,写出 rx_data 的代码
16 always  @(posedge clk or negedge rst_n)begin
17       if(rst_n == 1'b0)
18           rx_data <= 0;
19       else if(cnt2> = 1&&cnt2<9&&add_cnt1&&cnt1 == 2604 - 1)begin
20           rx_data[cnt2 - 1] <= uart_rx_ff2;
21       end
22 end
23
24 //按照第六步第 5 点,写出 rx_vld 的代码
25 always  @(posedge clk or negedge rst_n)begin
26       if(rst_n == 1'b0)
27           rx_vld <= 0;
28       else if(cnt2 == 8&&add_cnt1&&cnt1 == 2604 - 1)begin
29           rx_vld <= 1;
30       end
31       else begin
32           rx_vld <= 0;
33       end
34 end
35
36 //按照第六步第 6 点,写出 flag 的代码
37 always  @(posedge clk or negedge rst_n)begin
38       if(rst_n == 1'b0)begin
39           flag <= 0;
40       end
41       else if(uart_rx_ff2 == 1&&uart_rx_ff1 == 0)begin
42           flag <= 1;
43       end
44       else if(end_cnt2)begin
45           flag <= 0;
46       end
47 end
```

2.2.3 VGA 接口项目

VGA(Video Graphics Array)是 IBM 公司在 1987 年随 PS/2 机一起推出的一种视频传输标准,当时具有分辨率高、显示速率快、颜色丰富等优点,在彩色显示器领域得到了广泛应用。

VGA 接口就是显卡上输出模拟信号的接口,也叫 D-Sub 接口。VGA 接口是一种 D 型口,上面共有 15 针/孔,分成 3 排,每排 5 个。VGA 接口是目前中低端电脑配置的主流接口。

VGA 显示中,FPGA 需要产生 5 个信号:R、G、B 三基色信号,行同步信号 HS,场同步信号 VS。VGA 接口信号列表如表 2-5 所列。

表 2-5 VGA 接口信号列表

信号线	定 义
HS	行同步信号(3.3 V 电平)
VS	场/帧同步信号(3.3 V 电平)
R	红基色(0～0.714 V 模拟信号)
G	绿基色(0～0.714 V 模拟信号)
B	蓝基色(0～0.714 V 模拟信号)

VGA 接口图如图 2-24 所示。

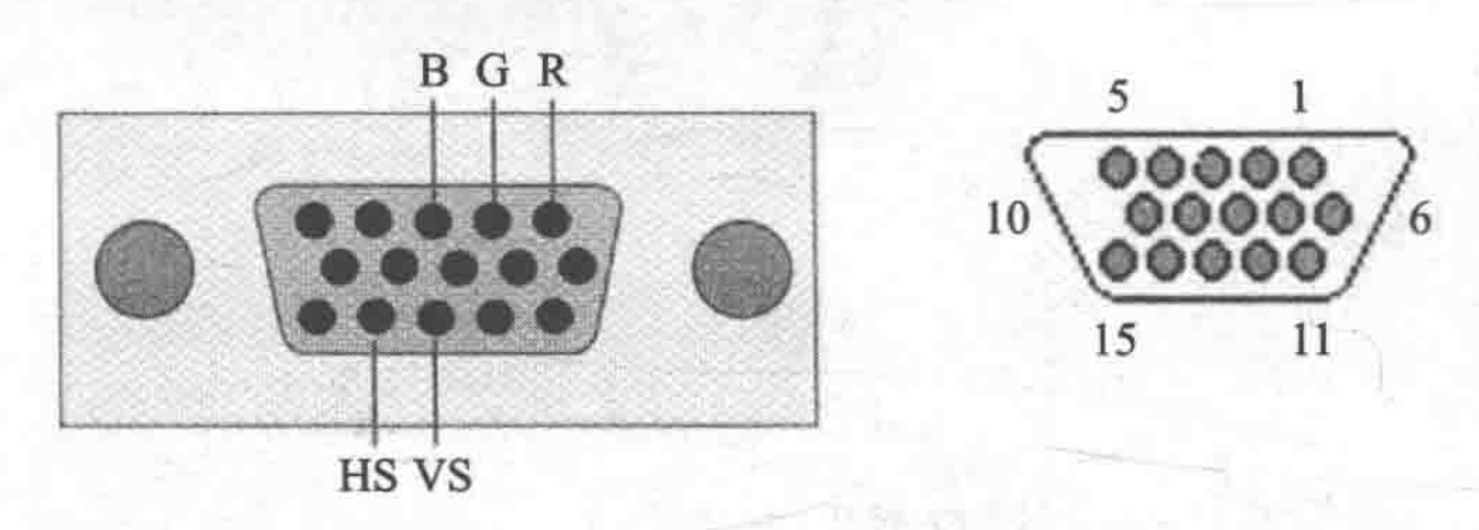


图 2-24 VGA 接口实物图

以上接口的 5 个孔对应着 FPGA 中产生的 5 个重要信号,其中 R、G、B 是数据信号;HS、VS 是控制信号。

1. VGA 色彩原理

像素是产生各种颜色的基本单元。根据物理学中的混色原理,三色发光的亮度比例适当,可呈现白色。适当的调整发光比例可以出现不同的颜色。三基色颜色编码表如表 2-6 所列。

表 2-6　三基色颜色编码表

颜　色	黑	蓝	红	紫	绿	青	黄	白
R	0	0	1	1	0	0	1	1
G	0	0	0	0	1	1	1	1
B	0	1	0	1	0	1	0	1

以上 RGB 一共有 8 种组合，即可以产生 8 种颜色。但显示器显示的色彩却非常丰富，远远多于 8 种颜色，这是如何做到的呢？

对于显示器来说，RGB 三个信号其实都是模拟信号，其电平的高低可以表示颜色的深浅。利用这个原理，即可产生丰富的色彩。为了控制电压的高低，必须用到 D/A 芯片。例如图 2-25 中，FPGA 产生 RGB 三种信号，这时 RGB 都是多位的数字信号。D/A 芯片根据数字信号的值，产生不同电压的模拟信号 rgb。

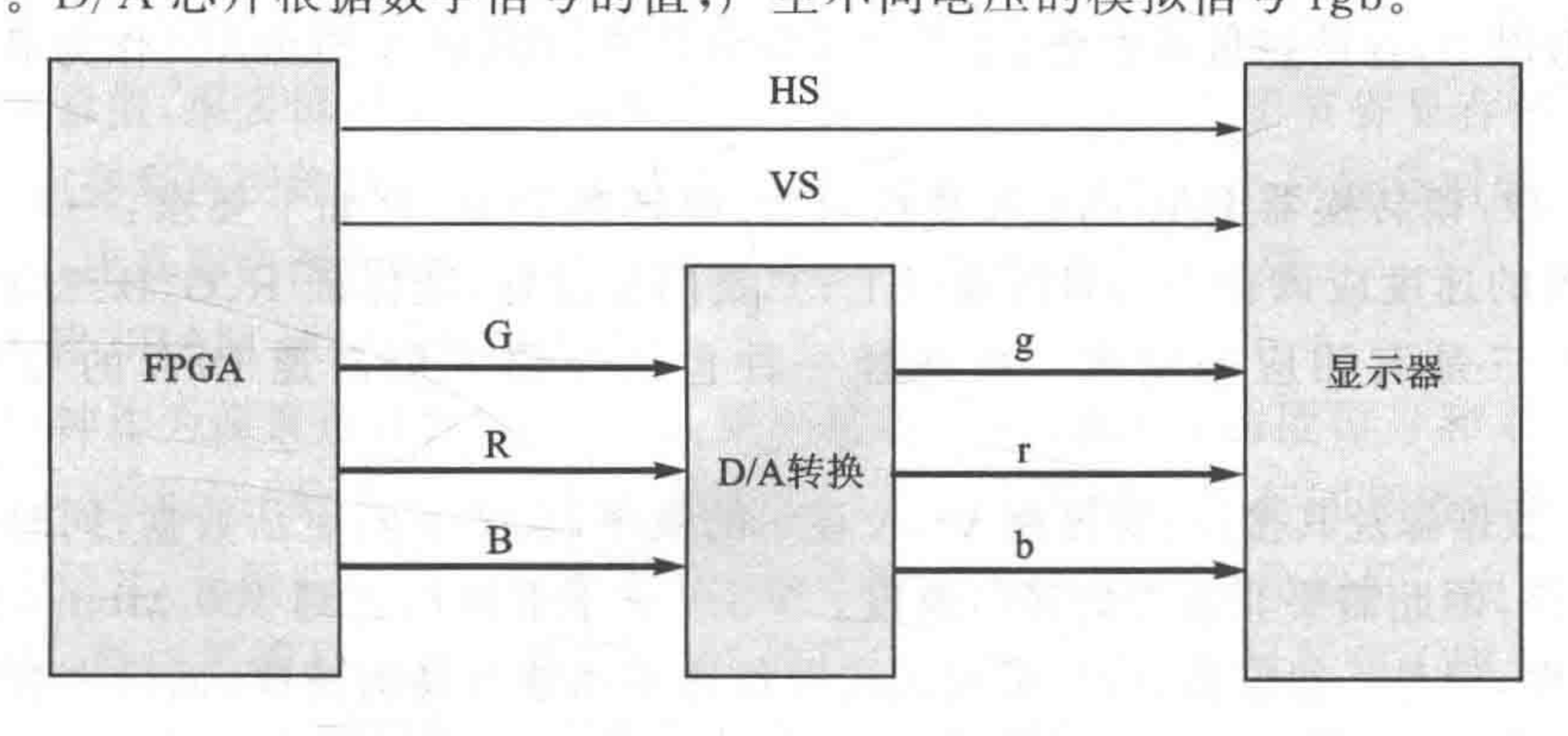


图 2-25　FPGA 与显示器的结构图

明德扬的第二代开发板中，R 由 3 位数字信号组成（VGA_R0，VGA_R1，VGA_R2）；G 由 3 位数字信号组成（VGA_G0，VGA_G1，VGA_G2）；B 由 2 位数字信号组成（VGA_B0，VGA_B1）。VGA 通过电压进行分压，得到不同的电平。实际电路如图 2-26 所示。

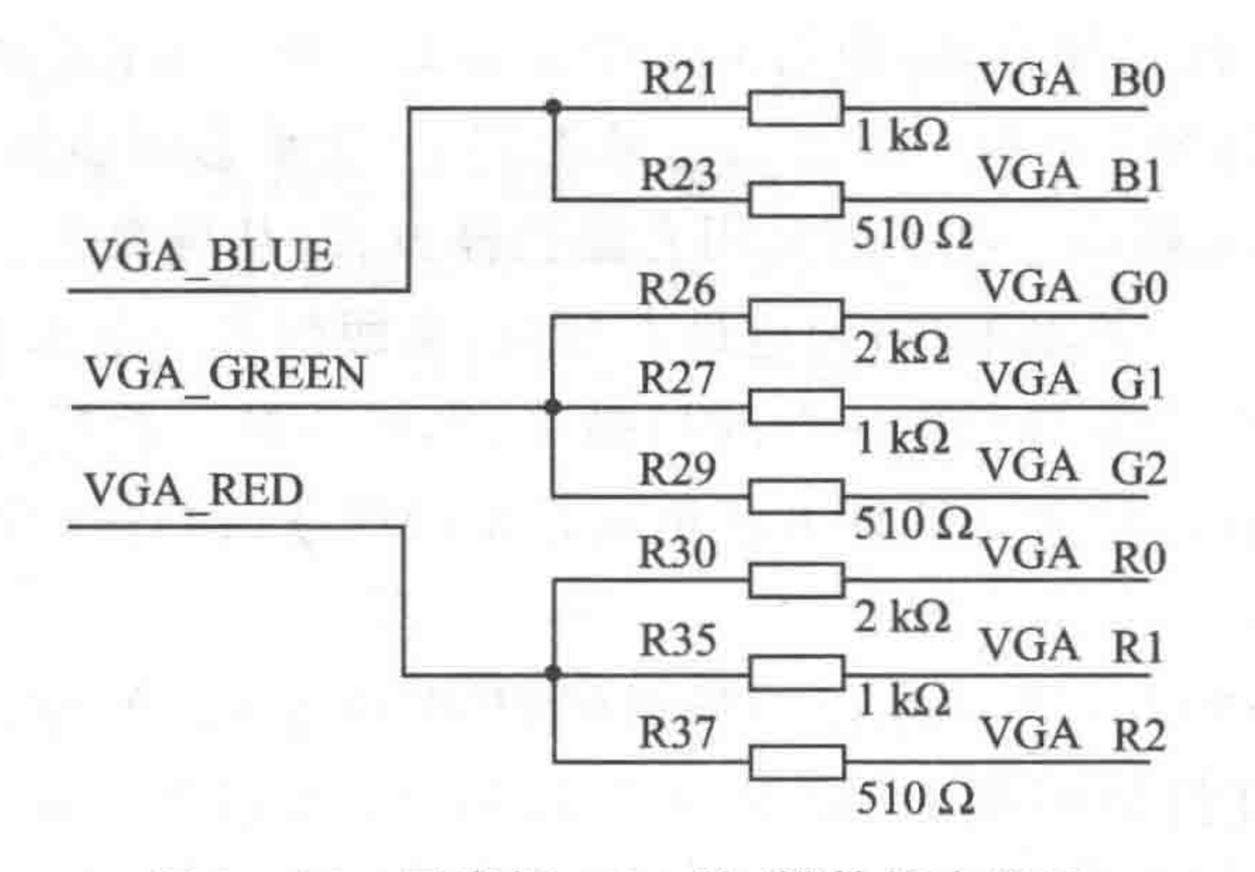


图 2-26　开发板 RGB 数/模转换电路图

2. VGA 显示

(1) 显示原理

通用 VGA 显示卡系统主要由控制电路、显示缓存区和视频 BIOS 程序三个部分组成。控制电路主要完成时序发生、显示缓冲区数据操作、主时钟选择和 D/A 转换等功能;显示缓冲区提供显示数据缓存空间;视频 BIOS 程序作为控制程序固化在显示卡的 ROM 中。

实现 VGA 显示,除了实现时序控制外,还必须有其他功能单元的支持才能实现完整的图像显示。

① 控制器:VGA 显示有多种模式,需要通过控制器实现模式间的切换,还需要对显示的内容进行接收、处理和显示。所以,控制器的性能越高,数据更新和显示的效果越好。显示数据缓存区:VGA 显示要求显存速度快、容量大。读速度要达到 65 MHz以上,存储容量至少要 2 MB。可采用高速 SRAM 或 SDRAM 作为显示数据缓存。

② 数/模转换器 DAC:VGA 显示对数/模转换 DAC 有如下要求:一是高速转换,转换的速度应该在 80 MHz 或以上;二是同步性好,能保证 R、G、B 三路信号的同步性;三是有相应的精度。可选择一种包括 3 路 8 位高速 D/A 的专用视频芯片。

③ 数据源及其接口:要提高 VGA 显示的效率,就要不断更新数据,同时还要保证实时性,因此需要非常高的接口速度。VGA 显示卡虽可达到 100 Mbit/s 的数据更新速度,但是一般设备、特别是嵌入式设备达不到这么高的速度,而且大多数情况下也不需要这么高的数据更新率。目前常用接口为 EPP 接口、USB 接口、TCP/IP、RS232C/485 等。其中,TCP/IP、EPP 接口和 USB 接口是基于计算机的,速度较快;TCP/IP、RS232C/485 是基于网络通信的接口,RS485 速度虽慢,但应用广泛且容易实现远程控制。

(2) 扫描方式

显示器采用光栅扫描方式,即轰击荧光屏的电子束在 CRT 屏幕上从左到右(受水平同步信号 HSYNC 控制见表 2-7)、从上到下(受垂直同步信号 VSYNC 控制见表 2-8)做有规律的移动。电子束采用光栅扫描方式,从屏幕左上角一点开始,向右逐点进行扫描,形成一条水平线;到达最右端后,又回到下一条水平线的左端,重复上面的过程;当电子束完成右下角一点的扫描后,形成一帧。此后,电子束又回到左上方起点,开始下一帧的扫描。这种方法也就是常说的逐行扫描显示。

(3) 扫描频率

完成一行扫描的时间称为水平扫描时间,其倒数称为行频率;完成一帧(整屏)扫描的时间称为垂直扫描时间,其倒数称为场频率,即刷新一屏的频率,常见的有 60 Hz、75 Hz 等。标准的 VGA 显示的场频率为 60 Hz。

表2-7　水平时序

分辨率	刷新速率/Hz	像素频率	同步脉冲	后沿	有效时间/列	前沿	帧长
640×480	60	25	96	45	646	13	800
640×480	72	31	40	125	646	21	832
800×600	56	36	72	125	806	21	1 024
800×600	60	40	128	85	806	37	1 056
800×600	72	50	120	611	806	53	1 040

注：有效时间包括6列过扫描边界列，有些时序表将这几列加在后沿和前沿中。

表2-8　垂直时序

分辨率	刷新速率/Hz	行宽	同步脉冲	后沿	有效时间/行	前沿	帧长
640×480	60	31	2	30	484	9	525
640×480	72	26	3	26	484	7	520
800×600	56	28	1	20	604	−1	625
800×600	60	26	4	21	604	−1	628
800×600	72	20	6	21	604	35	666

注：有效时间包括4行过扫描边界行，有些时序表中将这几行加在后沿和前沿中。

(4) 数据宽度和格式

如果 VGA 显示真彩色 BMP 图像，则需要 R、G、B 三个分量各 8 位，即 24 位表示一个像素值，很多情况下还采用 32 位表示一个像素值。为了节省显存的存储空间，可采用高彩色图像，即每个像素值由 16 位表示，R、G、B 三个分量分别使用 5 位、6 位、5 位，比真彩色图像数据量减少一半，同时又能满足显示效果。

(5) 读 SRAM 地址的产生方法

主时钟作为像素点计数脉冲信号，同时提供显存 SRAM 的读信号和 D/A 转换时钟，它所驱动的计数器的输出端作为读 SRAM 的低位地址。行同步信号作为行数计数脉冲信号，它所驱动的计数器的输出端作为读 SRAM 的高位地址。由于采用两片 SRAM，所以最高位地址作为 SRAM 的片选使用。由于信号经过 CPLD 内部逻辑器件时存在一定的时间延迟，在 CPLD 产生地址和读信号读取数据时，读信号、地址信号和数据信号不能满足 SRAM 读数据的时序要求。可以利用硬件电路对读信号进行一定的时序调整，使各信号之间能够满足读 SRAM 和为 DAC 输入数据的时序要求。

3. VGA 支持的规格

我们以第一个分辨率 640×480 来分析，其刷新速率是 60 Hz，每幅图像为 525 行，每行为 800 个值。也就是说，完成一幅图像约是 1 s/60＝16.6 ms，完成一行约为 16.6 ms/525＝31.75 μs，完成一个像素传送约为 31.75 μs/800＝40 ns。因此为了方便设计，接口设为 25 MHz 最方便，每个时钟送一个数据。

4.VGA 接口项目

屏幕分辨率为 640×480，刷新速率为 60 Hz，要求在屏幕中间显示一个 200×200 的绿色方块，屏幕其他地方均为黑色。

(1) 明确功能

图 2-27 所示为 VGA 接口时序图。

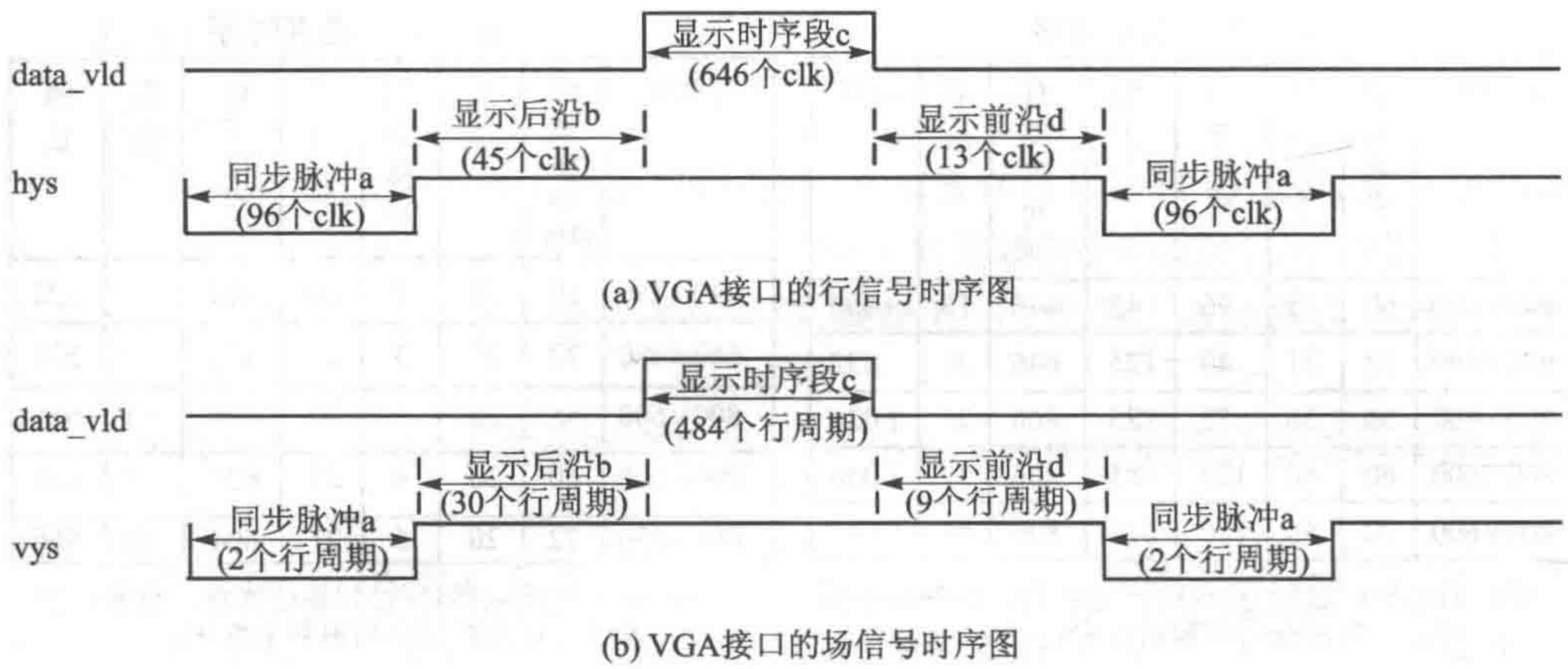


图 2-27　VGA 接口时序图

从时序图中可以知道，在第 32～515 个行周期中的第 141～786 个时钟周期间，VGA 输入数据有效。另外，根据屏幕分辨率和刷新速率，可以得出使用25 MHz 系统时钟最合适。表 2-9 为 VGA 接口信号列表。

表 2-9　VGA 接口信号列表

信号名	I/O	位　宽	说　明
clk	I	1	系统工作时钟 25 MHz
rst_n	I	1	系统复位信号，低电平有效
hys	O	1	行同步信号
vys	O	1	场同步信号
rgb_data	O	8	输出显示的 RGB 数据

(2) 功能波形

本项目的功能波形在上一步已经给出，只需要在 VGA 输入数据有效期间，根据题目要求把 rgb_data 变为绿色或黑色即可。

(3) 计数结构

因为从复位后开始，hys 与 vys 不断循环产生波形，其顺序为同步脉冲→显示后沿→显示时序段→显示前沿，所以使用计数器 cnt_hs 与 cnt_vs，分别用于计数 clk 个数和行周期个数，如图 2-28 所示。

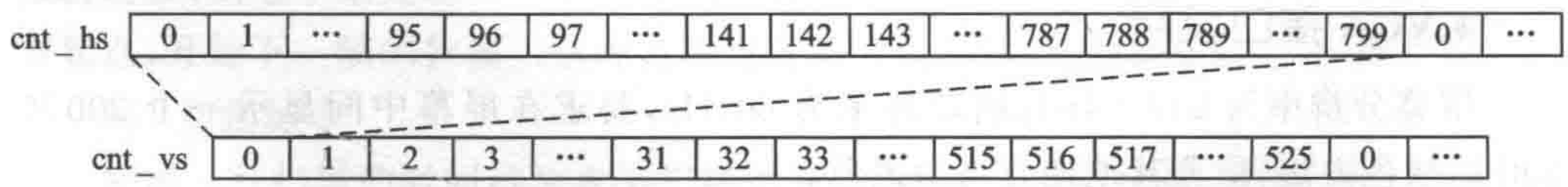


图 2-28　VGA 接口计数结构图

(4) 加 1 和结束条件

cnt_hs 的加 1 条件：由于 cnt_hs 在模块开始工作后不断计数，因此其加 1 条件为 1；

cnt_vs 的加 1 条件：在 cnt_hs 计数结束时加 1，因此其加 1 条件为 end_cnt_hs；

cnt_hs 的结束条件：cnt_hs==800－1；

cnt_vs 的结束条件：cnt_vs==525－1。

(5) 定义特殊点

按要求，这里应是黑色区域和绿色区域的组合，因此要定义好两个区域的边界。图 2－29 中有几个特殊点需要记住。

cnt_hs 的结束条件：cnt_hs==800－1，定为 end_cnt_hs；

cnt_vs 的结束条件：cnt_vs==525－1，定为 end_cnt_vs。

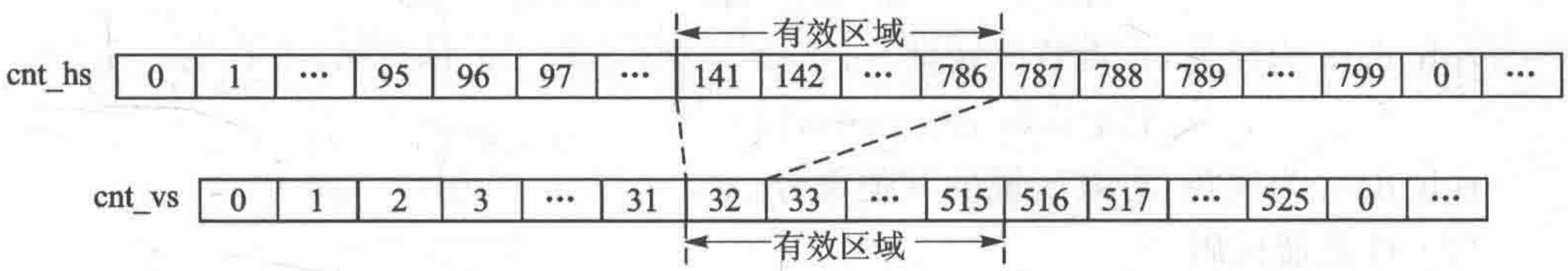


图 2－29　特殊点示意图

首先从时序图可知，有效区域为第 32～515 个行周期中的第 141～786 个时钟周期。由于要在中间显示绿色，因此先计算屏幕中点位置：

$$X_{\text{CENT}}=\frac{646}{2}+96+45=464$$

$$Y_{\text{CENT}}=\frac{484}{2}+2+30=274$$

由于绿色方框像素为 200×200，即有

$$X_{\text{CENT}}-100\leqslant\text{绿色区域行长度}<X_{\text{CENT}}+100$$

$$Y_{\text{CENT}}-100\leqslant\text{绿色区域场长度}<Y_{\text{CENT}}+100$$

除此之外，有效区域的其他部分均为黑色。

(6) 完整性检查

1) 计数器 cnt_hs

cnt_hs 的初值：0；

cnt_hs 的加 1 条件：1；

cnt_hs 的结束值：计数至 cnt_hs==800－1。

2) 计数器 cnt_vs

cnt_vs 的初值：0；

cnt_vs 的加 1 条件：end_cnt_hs；

cnt_vs 的结束值：计数至 cnt_vs==525－1。

3）行信号 hys

hys 的初值：0；

hys 由 0 变 1：cnt_hs==10'd95 && add_cnt_hs；

hys 由 1 变 0：end_cnt_hs。

4）场信号 vys

vys 的初值：0；

vys 由 0 变 1：cnt_vs==1'd1 && add_cnt_vs；

vys 由 1 变 0：end_cnt_vs。

5）RGB 数据信号 rgb_data

rgb_data 的初值：0；

rgb_data 有效区域：add_cnt_hs && cnt_hs>=141 && cnt_hs<787 && cnt_vs>=32 && cnt_vs<516；

rgb_data 为绿色：在有效区域且 cnt_hs>=364 && cnt_hs<564 && cnt_vs>=174 && cnt_vs<374；

rgb_data 为黑色：有效区域的其他部分。

（7）计数器代码

```
always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        cnt_hs <= 0;
    end
    else if(add_cnt_hs)begin
        if(end_cnt_hs) begin
            cnt_hs <= 0;
        end
        else begin
            cnt_hs <= cnt_hs + 1;
        end
    end
end

assign add_cnt_hs = 1;
assign end_cnt_hs = add_cnt_hs && cnt_hs == 800 - 1;

always @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        cnt_vs <= 0;
    end
    else if(add_cnt_vs)begin
        if(end_cnt_vs) begin
```

```
            cnt_vs <= 0;
        end
        else begin
            cnt_vs <= cnt_vs + 1;
        end
    end
end

assign add_cnt_vs = end_cnt_hs;
assign end_cnt_vs = add_cnt_vs && cnt_vs == 525 - 1;
```

(8) 功能代码

```
//按照第六步第 3 点,写出 hys 的代码
assign hs_rise = add_cnt_hs && cnt_hs == 10'd95;
always  @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        hys <= 1;
    end
    else if(hs_rise)begin
        hys <= 1;
    end
    else if(end_cnt_hs)begin
        hys <= 0;
    end
end

//按照第六步第 4 点,写出 vys 的代码
assign vs_rise = add_cnt_vs && cnt_vs == 1'd1;
always  @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        vys <= 1;
    end
    else if(vs_rise)begin
        vys <= 1;
    end
    else if(end_cnt_vs)begin
        vys <= 0;
    end
end

//有效区域与显示绿色区域定义
parameter    X0        = 141;
parameter    X1        = 787;
```

```
32 parameter    Y0       = 32 ;
33 parameter    Y1       = 516;
34 parameter    X_CENT   = 464;
35 parameter    Y_CENT   = 274;
36 parameter    GREEN    = 8'b000_111_00;
37 parameter    BLACK    = 8'b000_000_00;
38
39 assign valid_area = add_cnt_hs && cnt_hs>= X0 && cnt_hs<X1 && cnt_vs>= Y0
40                     && cnt_vs<Y1;
41 assign green_area = valid_area && (cnt_hs>= X_CENT - 100
42                     && cnt_hs<X_CENT + 100 && cnt_vs>= Y_CENT - 100
43                     && cnt_vs<Y_CENT + 100);
44
45 //按照第六步第 5 点,写出 rgb_data 的代码
46 always  @(posedge clk or negedge rst_n)begin
47     if(rst_n == 1'b0)begin
48         rgb_data <= 8'h00;
49     end
50     else if(valid_area)begin
51         if(green_area)begin
52             rgb_data <= GREEN;
53         end
54         else begin
55             rgb_data <= BLACK;
56         end
57     end
58     else begin
59         rgb_data <= 8'h00;
60     end
61 end
```

2.2.4 SCCB(IIC)接口项目

1. SCCB 背景知识

SCCB(Serial Camera Control Bus)即串行摄像机控制总线协议,是与 IIC 相似的一个协议。SCCB 协议由串行时钟线 sio_c 及串行数据线 sio_d 即可完成数据的传输。主、从两个设备之间的通信如图 2-30 所示。

2. 接口时序

图 2-31 所示为 SCCB 接口时序图。

① 起始信号:当 sio_c 为 1 时,sio_d 拉低,表示 SCCB 接口数据传输开始。

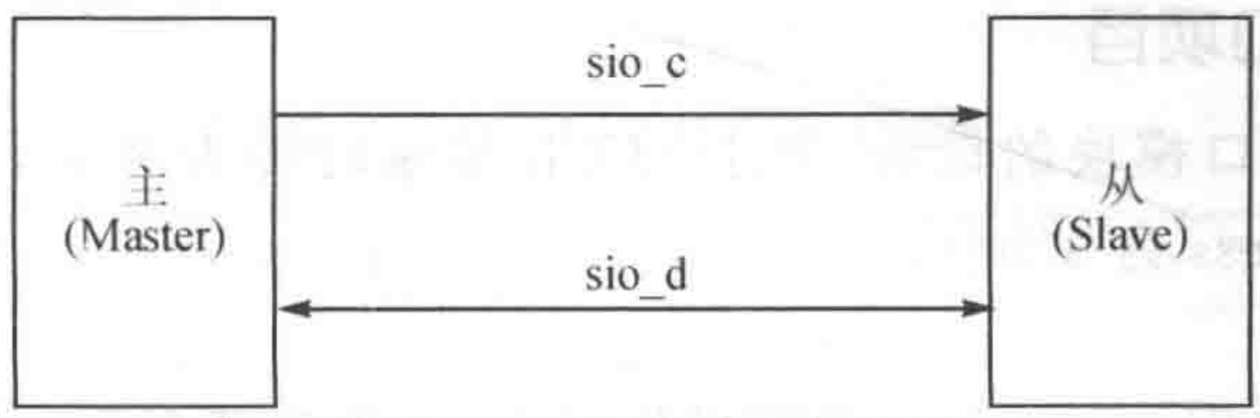


图 2－30　SCCB/IIC 接口结构图

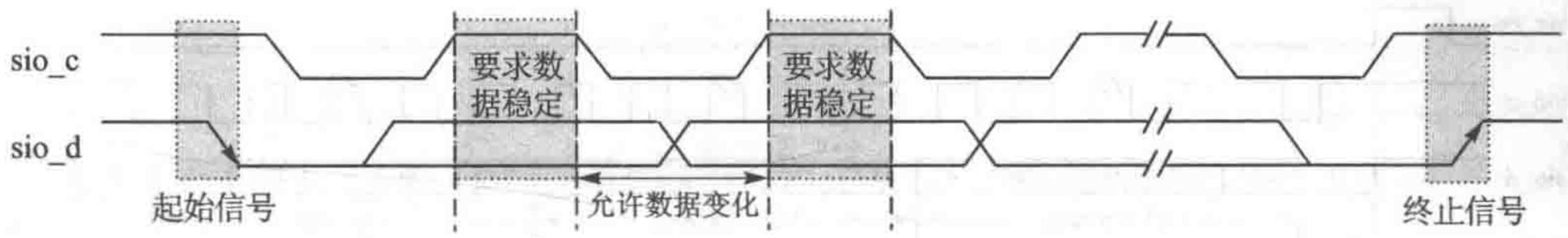


图 2－31　SCCB 接口时序图

② 终止信号：当 sio_c 稳定为 1 时，sio_d 拉高，表示 SCCB 接口数据传输终止。

③ 有效数据传输：在起始信号与终止信号之间，sio_d 只在 sio_c 为 1 时数据有效，因此 sio_d 的数据变化必须在 sio_c 高电平到来前完成（sio_c 低电平期间），而在 sio_c 为 1 时数据要保持稳定。

④ 写操作时序：SCCB 协议对于写操作有特定的数据格式要求，sio_d 只需要在起始信号与终止信号之间按照该格式产生信号，就可以完成写操作。该数据格式如图 2－32 所示。

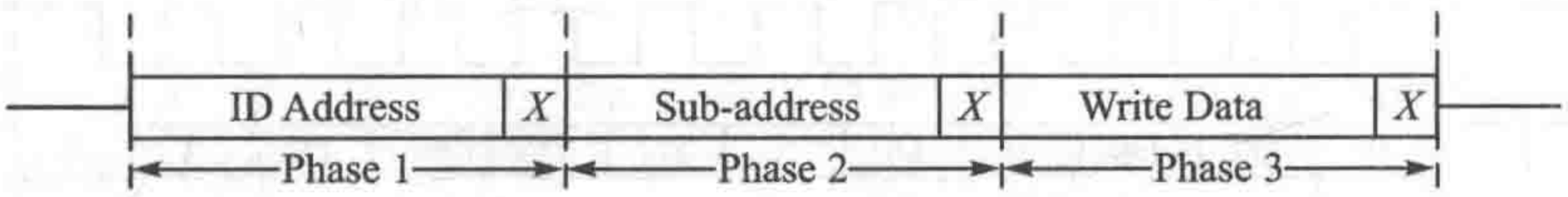


图 2－32　写操作数据格式

⑤ 读操作时序：按照 SCCB 协议，读操作时序分为两个阶段，需要通过 SCCB 接口连续产生两段起始—终止的 SCCB 接口时序。与写操作相同，读操作也有特定的数据格式要求。该格式如图 2－33 所示。

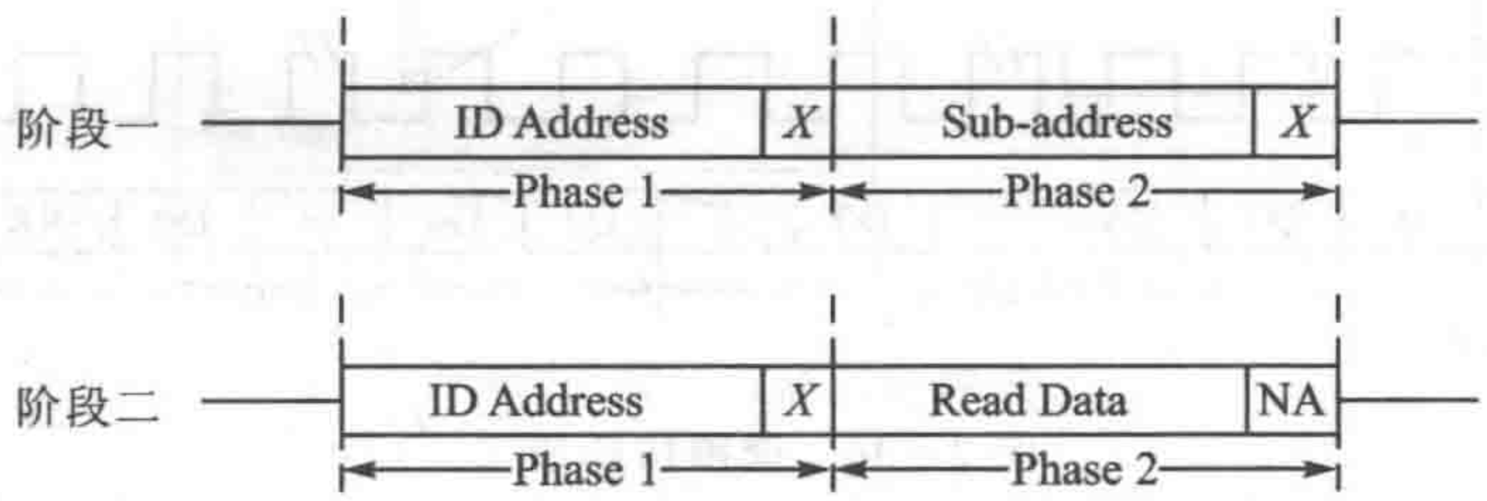


图 2－33　读操作数据格式

3. SCCB 接口项目

完成 SCCB 接口模块的设计，要求 SCCB 传输时钟为 8 μs，写 ID Address 为 8'h42，读 ID Address 为 8'h43。

(1) 明确功能

① 写操作。根据 SCCB 接口的写操作时序，当收到写命令 wr_en 时，产生以下波形，完成写操作，如图 2-34 所示。

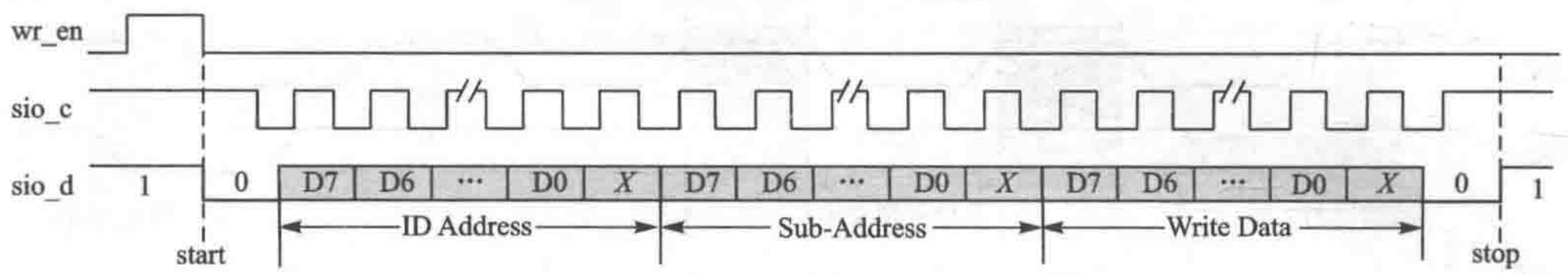


图 2-34 SCCB 接口功能波形图

ID Address 是设备地址(从片地址)，Sub-Address 是寄存器的地址，Write Data 是需要写入的数据。X 代表不定态，在本项目中令 X 等于 1。

② 读操作。根据 SCCB 接口的读操作时序，当收到读命令 rd_en 时，产生以下波形，完成读操作。

读操作阶段一如图 2-35 所示。

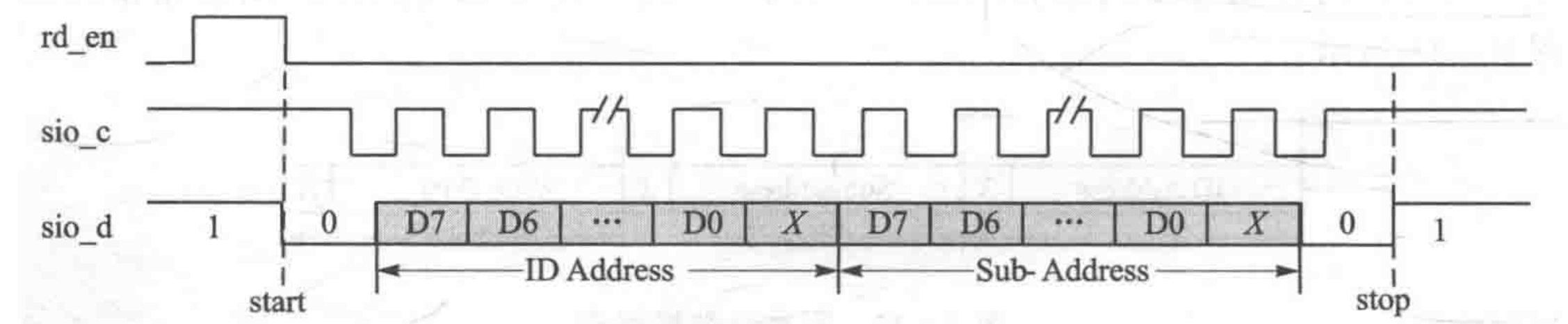


图 2-35 读操作阶段一

读操作阶段二如图 2-36 所示。

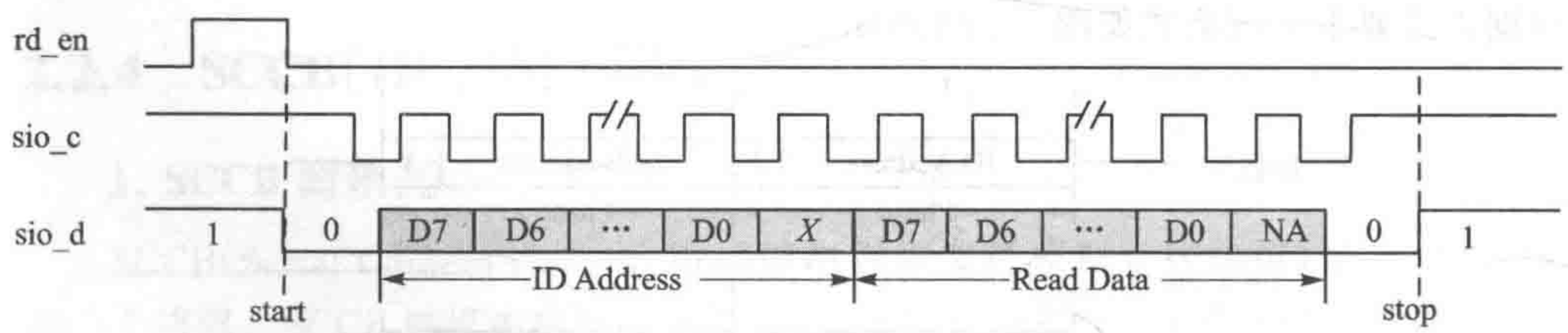


图 2-36 读操作阶段二

ID Address 和 Sub-address 分别为设备地址(从片地址)和寄存器地址，Read Data 为从设备中读出来的数据。X 与 NA 的值都为 1。

从所需实现的时序接口信号及读/写命令、地址、数据等信号，可以得到表 2-10

所列的 SCCB 接口信号列表。

表 2-10　SCCB 接口信号列表

信号名	I/O	位　宽	功　能
clk	I	1	系统时钟为 50 MHz
rst_n	I	1	系统复位信号
addr	I	8	读/写地址
wr_en	I	1	写使能信号
wdata	I	8	写入的数据
rd_en	I	1	读使能信号
rdata	O	8	读出的数据
rdata_vld	O	1	读出数据有效
sclk	O	1	SCCB 传输时钟，其时钟周期为 8 μs
sio_out	O	1	SCCB 数据输出
sio_out_en	O	1	SCCB 数据线传输使能，为 1 时开始写数据
sio_din	I	1	SCCB 数据输入，在读的第二阶段采样 sio_d 上 Read Data 的值，并赋予 rdata
rdy	O	1	准备就绪，等待接收读或写信号

从 SCCB 的时序要求可以知道，SCCB 接口只有 1 根串行数据线——sio_d。但细心的读者会发现，在信号列表中，并没有 sio_d 信号，反而有三个相对特殊的信号：输出信号 sio_out、输入信号 sio_din 和输出使能信号 sio_out_en。这样处理的原因，在这里作简要的说明。首先了解一下三态门电路，如图 2-37 所示。

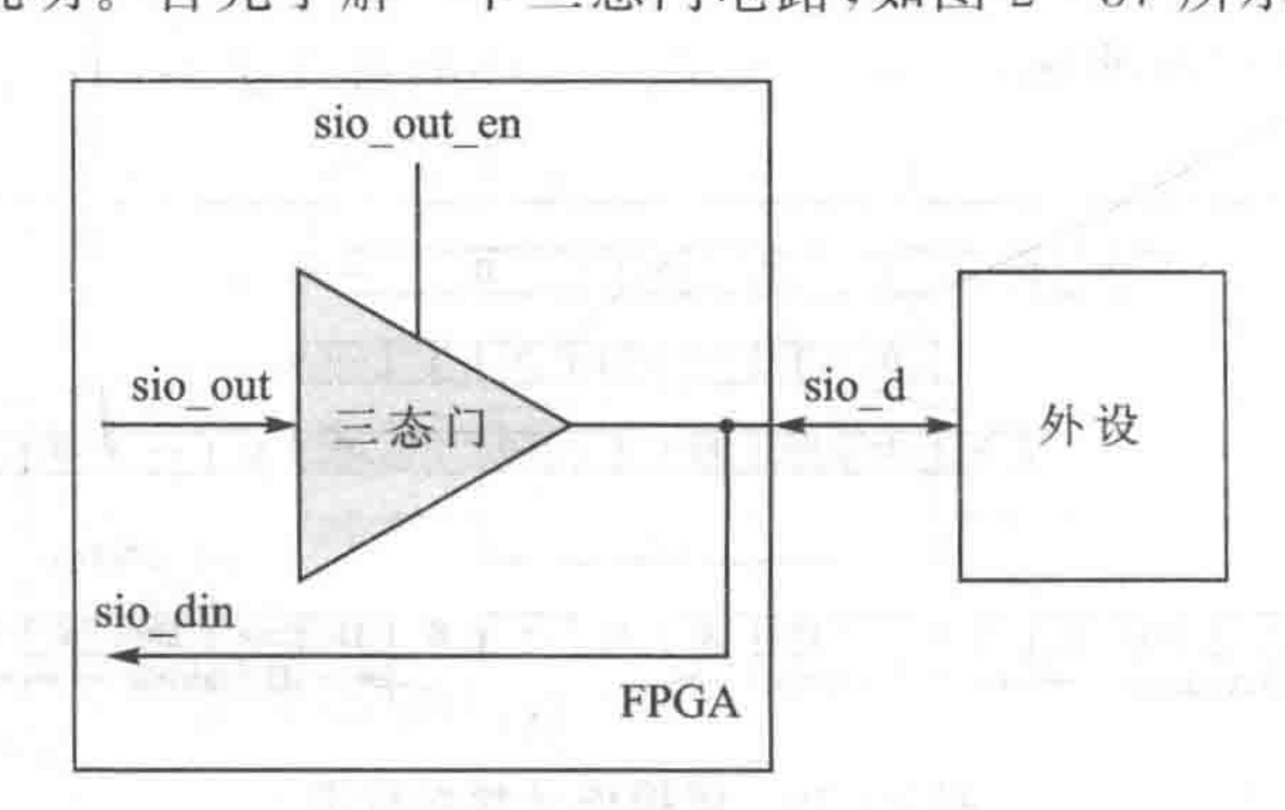


图 2-37　三态门电路图

在 FPGA 内部，并没有不定态、高阻态，因此增加 sio_out_en、sio_out 及 sio_din 三个信号，只需要在顶层文件使用该接口时，写出以下三态门的代码，综合时即可综合成三态门电路。其代码如下：

```
inout    sio_d;
```

```
assign  sio_d   = sio_out_en? sio_out:1'bz;
assign  sio_din = sio_d;
```

其中，sio_out_en 为三态门的使能端。当其有效时，将 sio_out 的值送给 sio_d；当其无效时，给 sio_d 送高阻态，此时 sio_din 的值从 sio_d 取得。三态门电路是常用的电路，在本书后面的项目中还会出现，希望读者能够熟练掌握。

(2) 功能波形

本项目的功能波形在上一步已经给出，只需要读/写命令，产生相应的读/写操作时序即可。

(3) 计数结构

写操作如图 2-38 所示。

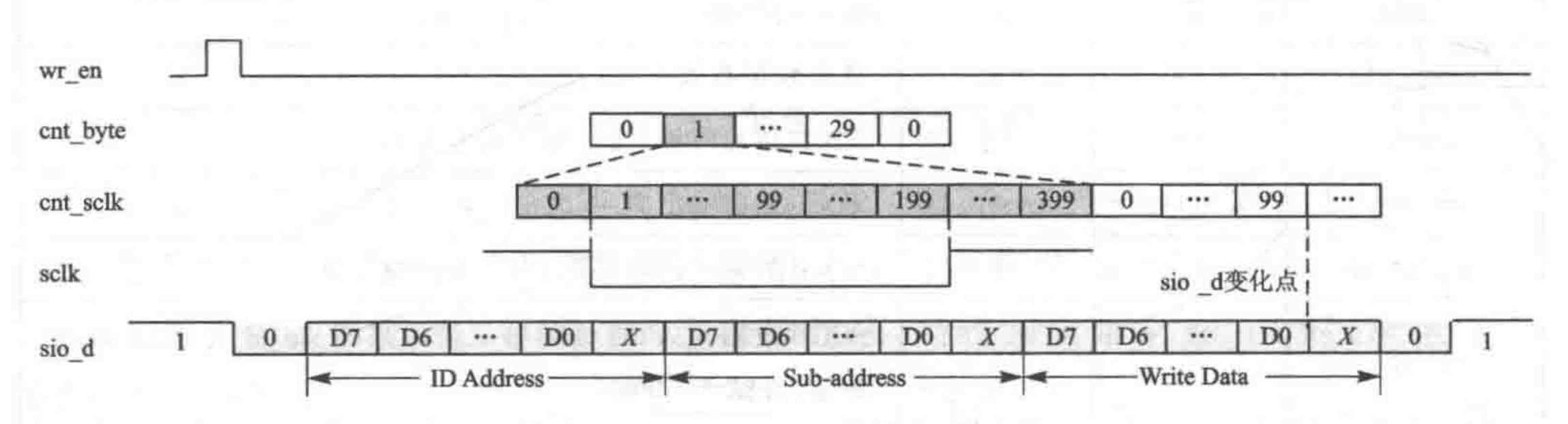


图 2-38 写操作计数结构图

当收到写使能时，sclk 周期发生变化，sio_out 输出 30 bit 数据。

这里用到两个计数器：第一个计数器(cnt_sclk)用来计数 sclk 的低电平和高电平持续的时钟周期。第二个计数器(cnt_byte)用来计数 sio_out 输出到第几个数。由于模块的工作时钟是 50 MHz，sclk 的时钟周期为 8 μs，故需 400 个时钟周期。

读操作如图 2-39 所示。

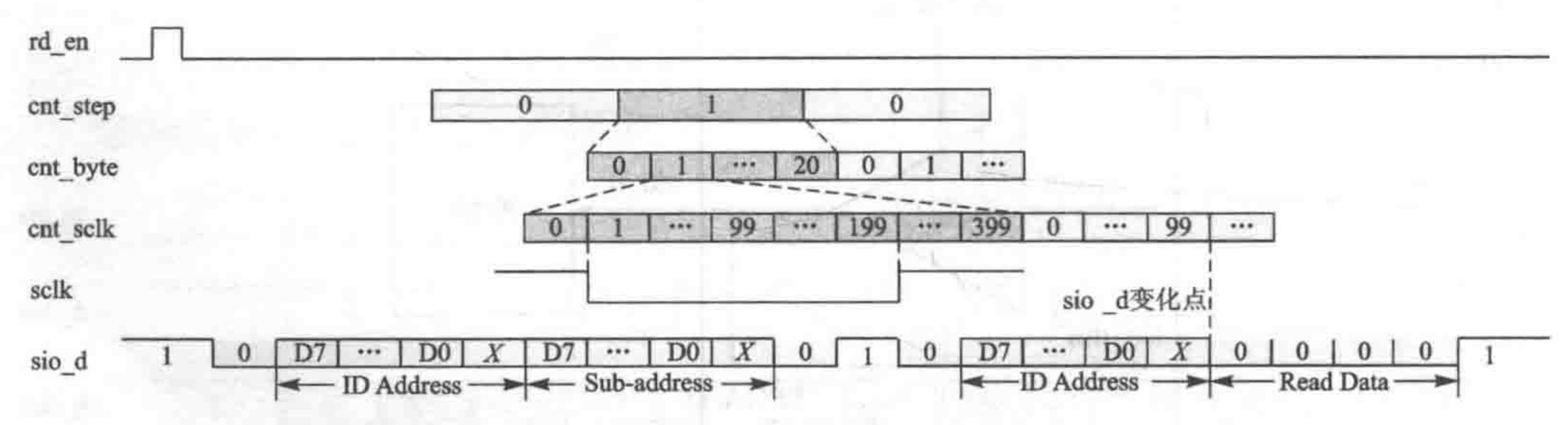


图 2-39 读操作计数结构图

当收到读使能时，sclk 周期发生变化，sio_out 分两个阶段输出数据。

这里用到三个计数器：计数器 cnt_sclk 和 cnt_byte 的用途及计数情况，与写操作类似。只是 cnt_byte 的结束条件不同。计数器 cnt_step 是用来计数到读操作的第几阶段。

(4) 加 1 和结束条件

写操作如图 2－40 所示。

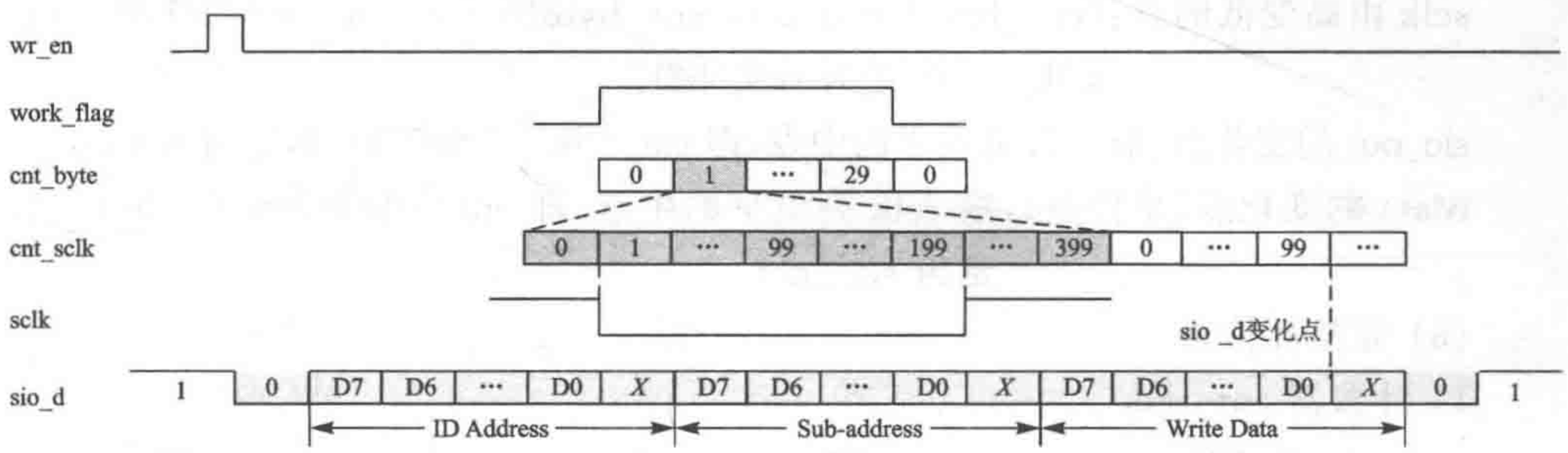


图 2－40　写操作加 1 和结束条件图

读操作如图 2－41 所示。

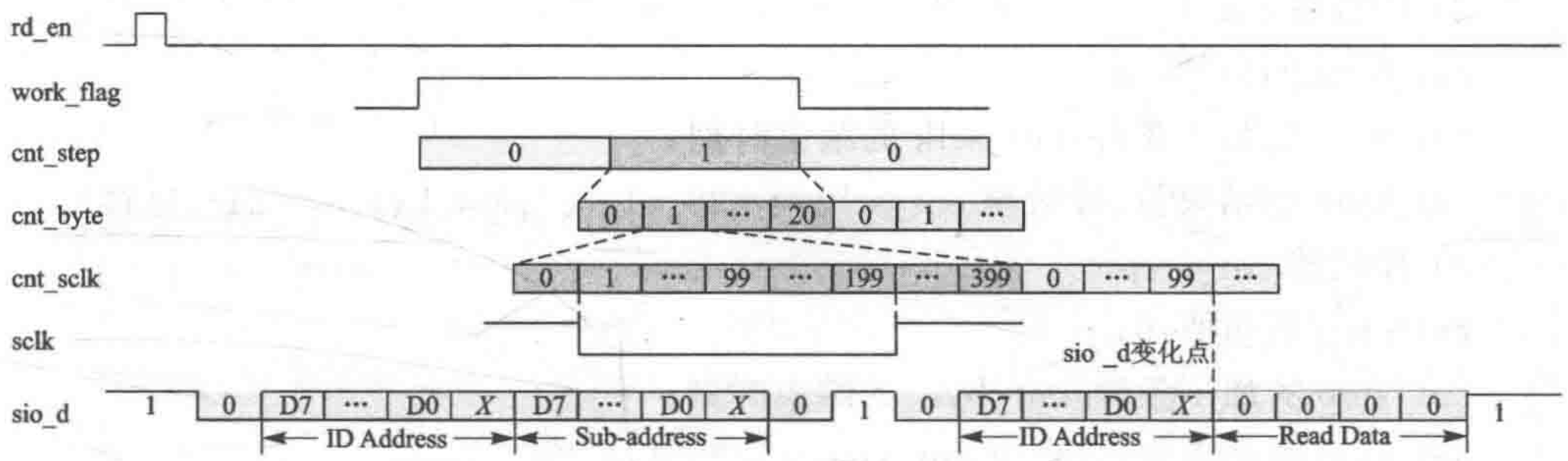


图 2－41　读操作加 1 和结束条件图

1) cnt_sclk

cnt_sclk 加 1 条件：从现有条件中，无法确定 cnt_sclk 的计数区间，故需要增加一个指示信号 work_flag。当 work_flag＝＝1 时，cnt_sclk 计数。

cnt_sclk 结束条件：cnt_sclk＝＝400－1。

2) cnt_byte

cnt_byte 加 1 条件：cnt_sclk 的结束时刻；

cnt_byte 结束条件：cnt_byte＝＝21－1。

3) cnt_step

cnt_step 加 1 条件：cnt_byte 的结束时刻；

cnt_step 结束条件：cnt_byte＝＝2－1。

(5) 定义特殊点

在读/写操作的加 1 和结束条件图中，有几个特殊点需要记住。

cnt_sclk 结束条件：cnt_sclk＝＝400－1，定为 end_cnt_sclk。

cnt_byte 结束条件：cnt_byte＝＝byte_num－1（写时 byte_num＝＝30；读时 byte_num＝＝21），定为 end_cnt_byte。

cnt_step 结束条件：cnt_step==2-1，定为 end_cnt_step。

sclk 由低变高的点：cnt_sclk==400/2-1，定为 sclk_l2h。

sclk 由高变低的点：(cnt_byte!= 0 && cnt_byte!=byte_num-1) && cnt_sclk==0，定为 sclk_h2l。

sio_out 的变化点：取 sclk 低电平的中点，即 cnt_sclk==400/4-1，定为 sio_send。

rdata 的变化点(采样点)：取 sclk 高电平的中点，即 cnt_sclk==3×400/4-1，定为 sio _get。

(6) 完整性检查

1) 计数器 cnt_sclk

cnt_sclk 的初值：0；

cnt_sclk 的加 1 条件：work_flag==1；

cnt_sclk 的结束值：计数至 cnt_sclk==400-1。

2) 计数器 cnt_byte

cnt_byte 的初值：0；

cnt_byte 的加 1 条件：cnt_sclk 的结束时刻；

cnt_byte 的结束值：计数至 cnt_sclk==30-1(写)，cnt_byte==21-1(读)。

3) 计数器 cnt_step

cnt_step 的初值：0；

cnt_step 的加 1 条件：cnt_byte 的结束时刻；

cnt_step 的结束值：计数至 cnt_step==2-1。

4) work_flag

work_flag 由 0 变 1：收到 wr_en 或 rd_en==1；

work_flag 由 1 变 0：cnt_step 计数结束时刻；

其他情况不变。

5) rd_flag

rd_flag 为 1：收到 rd_en==1；

rd_flag 为 0：收到 wr_en==1；

其他情况不变。

6) sclk

sclk 的初值：1；

sclk 由 1 变 0：在 0<cnt_byte<29 期间，cnt_sclk==0；

sclk 由 0 变 1：cnt_sclk==400/2-1；

其他情况保持不变。

7) sio_out

sio_out 的初值：1；

sio_out 的改变条件：cnt_sclk==400/4-1。

8）sio_out_en

sio_out_en 为 1：cnt_step==1 且 10<=cnt_byte<18 期间；

sio_out_en 为 0：其他情况。

9）rdata

rdata 的初值：1；

rdata 的改变条件：cnt_sclk==3×400/4−1 且 cnt_step==1，10<=cnt_byte<18 期间；

其他情况不变。

10）rdata_vld

rdata_vld 为 1：end_cnt_step；

rdara_vld 为 0：其他情况。

11）rdy

rdy 为 0：work_flag==1 或 rd_en==1 或 wr_en==1；

rdy 为 1：其他情况。

（7）计数器代码

```
always@(posedge clk or negedge rst_n)begin
        if(rst_n == 1'b0)begin
            cnt_sclk <= 0;
        end
        else if(add_cnt_sclk) begin
            if(end_cnt_sclk)
                cnt_sclk <= 0;
            else
                cnt_sclk <= cnt_sclk + 1;
        end
        else begin
            cnt_sclk <= 0;
        end
end
assign  add_cnt_sclk = work_flag;
assign  end_cnt_sclk = cnt_sclk == SCLK_TIME - 1 && work_flag;

always  @(posedge clk or negedge rst_n)begin
        if(rst_n == 1'b0)begin
            cnt_byte  <= 0;
        end
        else if(add_cnt_byte) begin
            if(end_cnt_byte)
                cnt_byte <= 0;
```

```
            else
                cnt_byte <= cnt_byte + 1;
        end
end
assign add_cnt_byte = end_cnt_sclk;
assign end_cnt_byte = add_cnt_byte&& cnt_byte == byte_num - 1;

always  @(posedge clk or negedge rst_n)begin
        if(rst_n == 1'b0)begin
            cnt_step <= 0;
        end
        else if(add_cnt_step)begin
            if(end_cnt_step)
                cnt_step <= 0;
            else
                cnt_step <= cnt_step + 1;
        end
end

assign  add_cnt_step = end_cnt_byte;
assign  end_cnt_step = add_cnt_step&& cnt_step == step_num - 1;
```

(8) 功能代码

```
parameter  IDWADD = 8'h42;
parameter  IDRADD = 8'h43;
parameter  SCLK_TIME = 400;
parameter  SCLK_HALF_TIME = SCLK_TIME/2;
parameter  SCLK_W_TIME = SCLK_TIME/4;
parameter  SCLK_R_TIME = (SCLK_TIME/4) * 3;

//按照第六步第 4 点,写出 work_flag 的代码
assign en = work_flag == 1'b0 && (wr_en || rd_en);
always  @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        work_flag <= 1'b0;
    end
    else if(en) begin
        work_flag <= 1'b1;
    end
    else if(end_cnt_step)begin
        work_flag <= 1'b0;
    end
```

```
20 end
21
22 //按照第六步第 5 点,写出 rd_flag 的代码
23 always  @(posedge clk or negedge rst_n)begin
24      if(rst_n == 1'b0)begin
25          rd_flag <= 1'b0;
26      end
27      else if(rd_en) begin
28          rd_flag <= 1'b1;
29      end
30      else if(wr_en)begin
31          rd_flag <= 1'b0;
32      end
33 end
34
35 assign  wr_state = work_flag && rd_flag == 1'b0;
36 assign  rd_state = work_flag && rd_flag;
37 assign  rd_0_state = rd_state && cnt_step == 0;
38 assign  rd_1_state = rd_state && cnt_step == 1;
39 assign  rd_get_state = rd_1_state && cnt_byte > = 10 && cnt_byte<18;
40
41 always  @(posedge clk or negedge rst_n)begin
42      if(rst_n == 1'b0)begin
43          subadd     <= 0;
44          wdata_ff0 <= 0;
45      end
46      else if(en) begin
47          subadd     <= addr;
48          wdata_ff0 <= wdata;
49      end
50 end
51
52 always  @( * )begin
53      if(wr_state)begin
54          wdata_tmp = {1'b0,IDWADD,1'b1,subadd,1'b1,wdata_ff0,1'b1, 1'b0,1'b1};
55          byte_num   = 30;
56          step_num   = 1;
57      end
58      else if(rd_0_state)begin
59          wdata_tmp = {1'b0,IDWADD,1'b1,subadd,1'b1,1'b0,1'b1,9'b0};
60          byte_num   = 21;
61          step_num   = 2;
```

```
62      end
63      else begin
64          wdata_tmp = {1'b0,IDRADD,1'b1,8'b0,1'b1,1'b0,1'b1,9'b0};
65          byte_num  = 21;
66          step_num  = 2;
67      end
68  end
69
70  //按照第六步第 6 点,写出 sclk 的代码
71  assign  start_area = add_cnt_sclk && cnt_byte == 0;
72  assign  stop_area = add_cnt_sclk && cnt_byte == byte_num - 1;
73
74  assign  sclk_h2l = add_cnt_sclk && cnt_sclk == 0 && ((! start_area) && (! stop_area));
75  assign  sclk_l2h = add_cnt_sclk && cnt_sclk == SCLK_HALF_TIME - 1;
76  always  @(posedge clk or negedge rst_n)begin
77      if(rst_n == 1'b0)begin
78          sclk <= 1'b1;
79      end
80      else if(sclk_h2l) begin
81          sclk <= 1'b0;
82      end
83      else if(sclk_l2h)begin
84          sclk <= 1'b1;
85      end
86  end
87
88  //按照第六步第 7 点,写出 sio_out 的代码
89  assign  sio_send = add_cnt_sclk && cnt_sclk == SCLK_W_TIME - 1
90                     && rd_get_state == 1'b0;
91  assign  sio_get = add_cnt_sclk && cnt_sclk == SCLK_R_TIME - 1 && rd_get_state;
92  always  @(posedge clk or negedge rst_n)begin
93      if(rst_n == 1'b0)begin
94          sio_out <= 1'b1;
95      end
96      else if(sio_send)begin
97          sio_out <= wdata_tmp[29 - cnt_byte];
98      end
99  end
100
101 //按照第六步第 8 点,写出 sio_out_en 的代码
102 always  @(posedge clk or negedge rst_n)begin
103     if(rst_n == 1'b0)begin
```

```
104             sio_out_en <= 1'b0;
105         end
106         else if(work_flag && rd_get_state == 1'b0) begin
107             sio_out_en <= 1'b1;
108         end
109         else begin
110             sio_out_en <= 1'b0;
111         end
112 end
113 //按照第六步第 9 点,写出 rdata 的代码
114 always  @(posedge clk or negedge rst_n)begin
115         if(rst_n == 1'b0)begin
116             rdata <= 1'b0;
117         end
118         else if(rd_get_state && sio _get) begin
119             rdata <= {rdata[6:0],sio_in};
120         end
121 end
122
123 //按照第六步第 10 点,写出 rdata_vld 的代码
124 always  @(posedge clk or negedge rst_n)begin
125         if(rst_n == 1'b0)begin
126             rdata_vld <= 1'b0;
127         end
128         else if(end_cnt_step && rd_1_state) begin
129             rdata_vld <= 1'b1;
130         end
131         else begin
132             rdata_vld <= 1'b0;
133         end
134 end
135
136 //按照第六步第 11 点,写出 rdy 的代码
137 always  @( * )begin
138         if(work_flag || rd_en || wr_en)
139             rdy = 1'b0;
140         else
141             rdy = 1'b1;
142 end
```

第 3 章

状态机架构

在数字电路设计中，状态机是一个非常重要的组成部分，也是贯穿于整个设计的最基本的设计思想和设计方法。在现代数字系统设计中，状态机的设计对系统的高速性、高可靠性、高稳定性都具有决定性的作用。熟练掌握状态机的设计，在数字电路的设计中必能达到事半功倍的效果。

在前一章中介绍了计数器。其实，计数器本质上也可以认为是一个状态机，只不过它是用数字来区分不同状态而已。

那么什么时候会用计数器，什么时候会用状态机呢？

如果是顺序处理或者是简单的流程控制，例如其步骤是 0→1→2→3→0，则用计数器实现是最方便的。但是在复杂的流程控制场合，其跳转顺序是乱序时，例如其步骤是 0→1→5→2→4，就应该利用状态机来设计。规范的状态机代码，可以极大地提高设计效率，减少状态出错的可能，缩短调试时间，同时也能设计出稳健的系统。

明德扬从项目实践和培训经验出发，总结出一套科学的、适用于状态机设计的方法，称为状态机架构八步法。

状态机架构八步法具有如下优点：

- 通用的设计方法，无论是简单还是复杂的状态机，均能按照此法进行设计；
- 步骤清晰易懂，每步只考虑一个问题，易于掌握和使用；
- 状态机代码严谨规范，不容易出错；
- 设计出的状态机结构简单并且稳健。

3.1 状态机架构八步法

本节主要介绍明德扬状态机架构八步法，在使用八步法前，先介绍明德扬总结的状态机规则。

3.1.1 状态机规则

状态机规则 1：使用四段式写法。

四段式不是指四个 always 代码，而是四段代码。另外需要注意的是，四段式状态机并非固定不变。如果没有输出信号就只有三段代码（两个 always）；如果有多个

输出信号，那么就会有多个 always。使用四段式的写法，可参照明德扬 GVIM 特色指令 Ztj 产生的状态机模板。

第一段，同步时序的 always 模块，格式化描述次态迁移到现态寄存器。

```
1  always@(posedge clk or negedge rst_n)begin
2      if(!rst_n)begin
3          state_c <= IDLE;
4      end
5      else begin
6          state_c <= state_n;
7      end
8  end
```

第二段，组合逻辑的 always 模块，描述状态转移条件判断。注意，转移条件用信号名来表示，信号名要按明德扬规则来命名。

```
1  always@( * )begin
2      case(state_c)
3          IDLE:begin
4              if(idle2s1_start)begin
5                  state_n = S1;
6              end
7              else begin
8                  state_n = state_c;
9              end
10         end
11         S1:begin
12             if(s12s2_start)begin
13                 state_n = S2;
14             end
15             else begin
16                 state_n = state_c;
17             end
18         end
19         S2:begin
20             if(s22idl)begin
21                 state_n = IDLE;
22             end
23             else begin
24                 state_n = state_c;
25             end
26         end
```

```
27            default:begin
28                state_n = IDLE;
29            end
30        end  case
31 end
32
```

第三段，用 assign 定义转移条件。注意条件一定要加上现态。

```
1  assign idle2s1_start   =   state_c == IDLE   && ;
2  assign s12s2_start     =   state_c == S1     && ;
3  assign s22idl_start    =   state_c == S2     && ;
```

第四段，设计输出信号。明德扬规范要求一个 always 设计一个信号，因此有多少个输出信号，就有多少个 always。

```
1  always  @(posedge clk or negedge rst_n)begin
2      if(!rst_n)begin
3          out1 <= 1'b0
4      end
5      else if(state_c == S1)begin
6          out1 <= 1'b1;
7      end
8      else begin
9          out1 <= 1'b0;
10     end
11 end
```

状态机规则 2：四段式状态机第一段写法不变。

设计状态机时，所有四段式状态机的第一段除了名字外，可以直接用，不需要改动。

状态机规则 3：第二段的状态转移条件用信号名来表示。

设计状态机时，其四段式状态机的第二段，明德扬要求转移条件用信号名来表示，无须直接写出具体的转移条件。

如果只用信号名代替，则须改动信号的名字，并且根据状态机的命名修改对应的跳转条件。

状态机规则 4：用 assign 的形式将状态转移条件写成 xx2xx_start 形式。

状态机规则 3 要求转移条件用信号名来表示，这样就要求读者想很多信号名，这通常是一大困扰。

为此，明德扬制定此规则，将状态转移条件的信号写成 xx2xx_start 形式。

例如，有三个状态 IDLE、READ、WRITE，若从 IDLE 状态跳转到 READ 状态，其跳转条件可以命名为：idle2read_start；若从 IDLE 状态跳转到 WRITE 状态，其跳

转条件可以命名为:idle2write_start。

这种命名方式,既能解决读者命名的困扰,又能直接从信号名看出信号的作用。明德扬提出并主张,所有信号都用有意义的字母来命名,达到"见名知义"。后续章节还会提及明德扬关于信号命名的规范,希望读者注意。

状态机规则5:状态转移条件,用assign产生变化条件时,必须加上当前状态。

第二段使用信号名来表示转移条件,在后面则要求用assign定义这个信号。注意,定义这个转移条件信号时,必须加上当前状态,避免因两种不同状态由同一个变化条件发生转移时导致错误。

例如下面的格式:

```
1  assign idle2s1_start   =   state_c == IDLE   && XX;
2  assign s12s2_start     =   state_c == S1     && XX;
3  assign s22idl_start    =   state_c == S2     && XX;
```

状态机规则6:状态不变,使用state_n = state_c。

明德扬项目和培训实践统计发现,编写状态机代码时,有很大一部分错误是由于复制粘贴时出错,即复制其他状态的代码时,忘记修改状态导致的。

同时,也有一部分同学写第二段状态机时,容易把状态保持不变写成state_n = state_n。这个写法是错误的,因为组合逻辑只有锁存器才能有保持电路,而锁存器是数字电路中通常不希望出现的。

为此,明德扬规定,四段式状态机的第二段,状态不变时使用state_n = state_c。如以下代码所示,读者可自行对比。这样写不仅可以减少出错的可能,还可以节省调试的时间。

```
1  IDLE:begin                                  IDLE:begin
2    if(idle2s1_start)begin                      if(idle2s1_start)begin
3      state_n = S1;                               state_n = S1;
4    end                                         end
5    else begin                                  else begin
6      state_n = state_c;                         state_n = IDLE;
7    end                                         end
8  end                                         end
```

下面通过一个例子,详细介绍状态机架构八步法。

例 设计一个模块,该模块有1个输入信号en,两个输出信号dout0和dout1。该模块信号列表见表3-1。

表 3-1 模块信号列表

信号名	I/O	位 宽	说 明
clk	I	1	模块工作时钟
rst_n	I	1	系统复位
en	I	1	使能信号
dout0	O	2	输出信号
dout1	O	2	输出信号

dout0 和 dout1 信号的变化情况如下：

dout0 复位后输出 0，1 个 en 后输出 1，4 个 en 后输出 2，2 个 en 后输出 3，3 个 en 后输出 0，之后重复上述步骤。

dout1 复位后输出 0，1 个 en 后输出 1，2 个 en 后输出 2，4 个 en 后输出 3，3 个 en 后输出 0，之后重复上述步骤。

3.1.2 第一步：明确功能

明确功能是将要实现的功能，用具体、清晰和无疑义的话描述出来。明确的功能要包括：模块的信号列表、每个信号具体的变化情况。

对于本例，其功能要求在题目叙述中已经非常明确，在此不再赘述。

根据功能要求，可以确定 en 为输入，dout0、dout1 为输出。

3.1.3 第二步：输出分析

输出分析是逐个分析每个输出信号所对应的最佳状态机。

首先要明确，状态划分的目的是什么？不是为了划分状态而去划分状态，划分状态的目的是方便用其作为条件，从而设计出输出信号，一切为了输出信号而服务。

所以划分状态时，首先要分析所有的输出信号情况，逐个考虑每个输出信号，分析产生这个信号所需要的状态，然后再合并和汇总，最终确定最好的状态划分。

1. dout0

复位后输出 0，1 个 en 后输出 1，4 个 en 后输出 2，2 个 en 后输出 3，3 个 en 后输出 0，之后重复上述步骤。

根据题意，划分成 4 个状态，产生 dout0 时最容易。在输出 0 状态时，dout0 输出 0；在输出 1 状态时，dout0 输出 1；在输出 2 状态时，dout0 输出 2；在输出 3 状态时，dout0 输出 3。dout0 状态转移图如图 3-1 所示。

2. dout1

复位后输出 0，1 个 en 后输出 1，2 个 en 后输出 2，4 个 en 后输出 3，3 个 en 后输出 0，之后重复上述步骤。

根据题意，划分成4个状态，产生dout1时最容易。在输出0状态时，dout1输出0；在输出1状态时，dout1输出1；在输出2状态时，dout1输出2；在输出3状态时，dout1输出3。dout1状态转移图如图3-2所示。

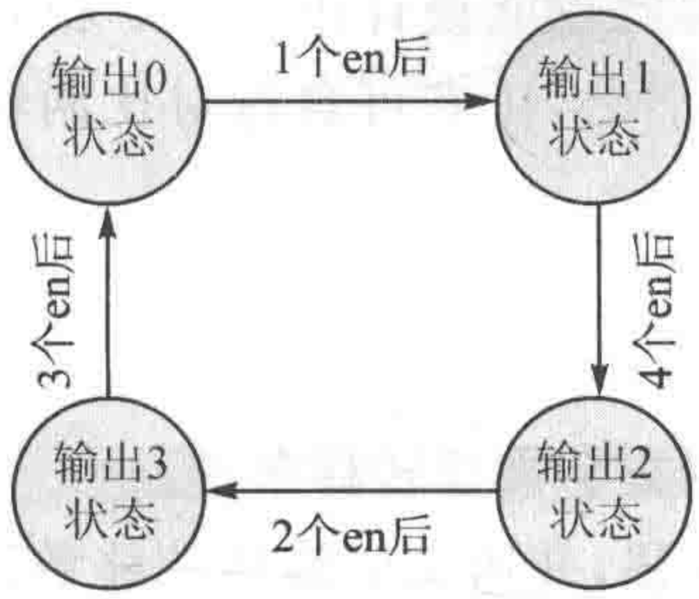


图3-1 dout0状态转移图

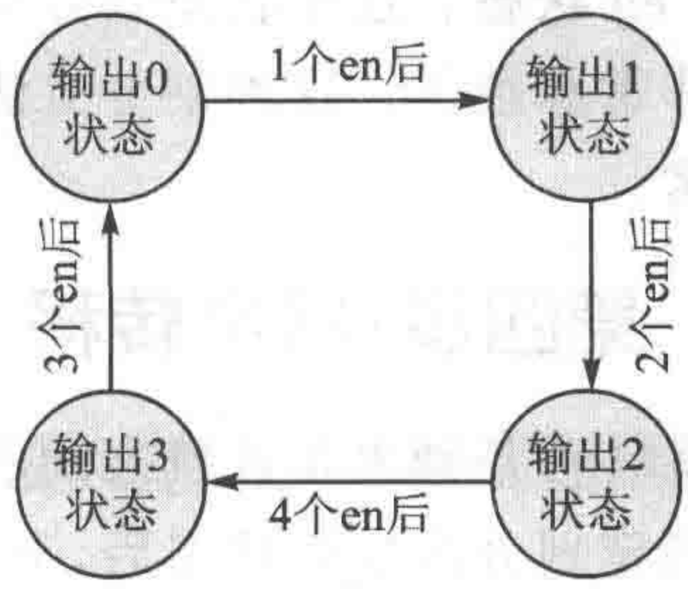


图3-2 dout1状态转移图

3.1.4 第三步：状态合并

状态合并是根据输出分析，对每个输出信号所对应的最佳状态机进行合并，从而得到能方便产生所有信号的状态机。

分析第二步的输出分析，虽然产生dout0和dout1，建议都分成4个状态：输出0状态、输出1状态、输出2状态和输出3状态。但是这两个信号不是同时处于同一状态。所以有必要对它们的两个状态进行合并，使得产生dout0和dout1都相对方便。合并的结果如图3-3所示。

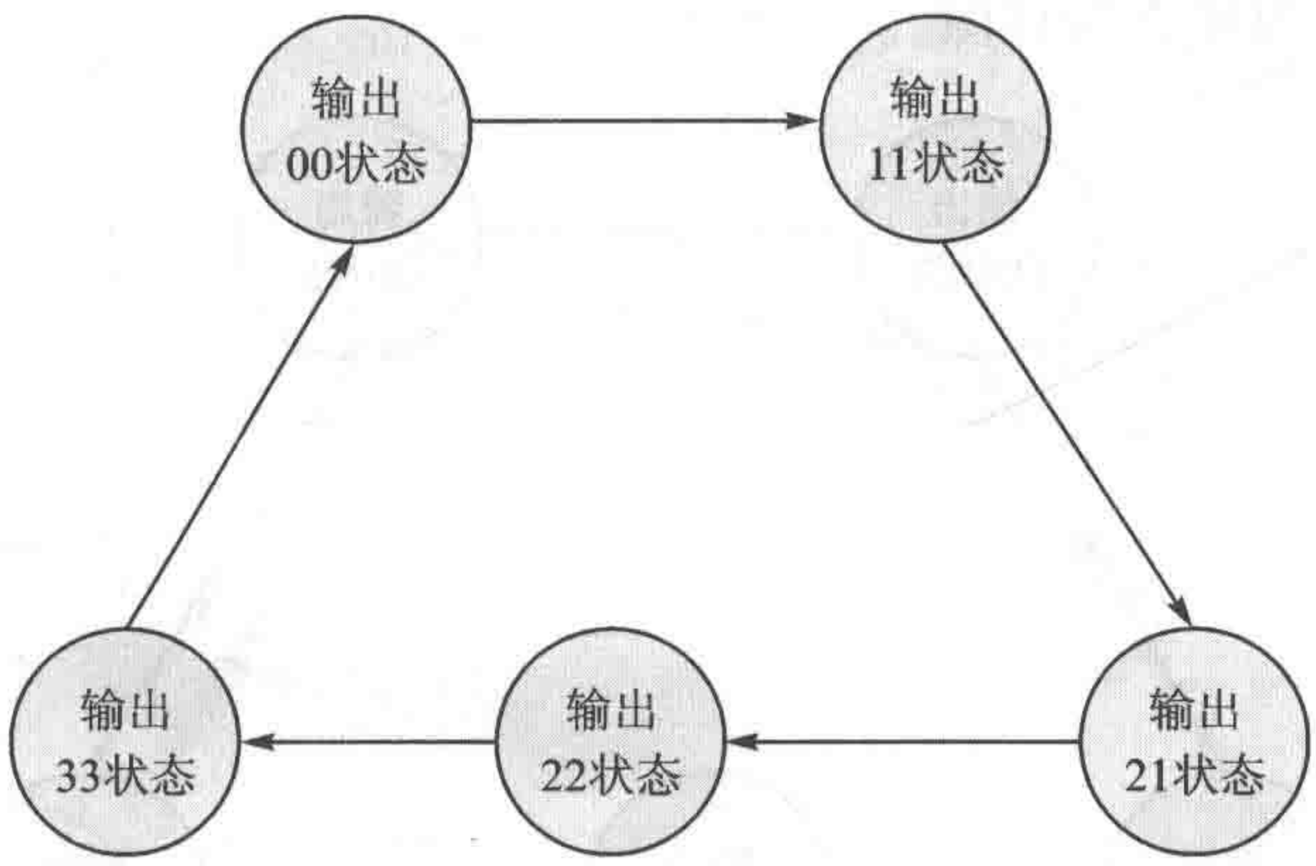


图3-3 状态合并图

输出00状态时，dout0=0，dout1=0；输出11状态时，dout0=1，dout1=1；输出21状态时，dout0=1，dout1=2；输出22状态时，dout0=2，dout1=2；输出33状态时，dout0=3，dout1=3。这样划分的状态，对dout0和dout1的产生都非常方便，设计起来也非常简单。

总结一下，状态划分不是划得越细越好，也不是越粗越好，而是从输出信号的需求出发，这是明德扬的核心思想。

本题可以有另一种划分方法：idle 状态、收到 1 个 en 状态、2 个 en 状态、……收到 10 个 en 状态。这个状态划分非常详细，并且据此也能设计出 dout0 和 dout1，但设计就非常麻烦了。这种划分方法是明德扬不提倡的，读者可自行对比两种划分方法的优劣。

3.1.5 第四步：状态转移

状态转移是画出状态机之间的跳转方向，并用文字描述转移条件。

在确定划分多少个状态后，下一步就是状态转移，并用文字描述转移条件。

dout0 复位后输出 0，1 个 en 后输出 1，4 个 en 后输出 2，2 个 en 后输出 3，3 个 en 后输出 0，之后重复上述步骤。

dout1 复位后输出 0，1 个 en 后输出 1，2 个 en 后输出 2，4 个 en 后输出 3，3 个 en 后输出 0，之后重复上述步骤。

本题我们划分了 5 个状态，其状态之间的跳转及条件分别如下：

输出 00 状态 → 输出 11 状态：收到 1 个 en。

输出 11 状态 → 输出 21 状态：收到 2 个 en。

输出 21 状态 → 输出 22 状态：收到 2 个 en。

输出 22 状态 → 输出 33 状态：收到 2 个 en。

输出 11 状态 → 输出 00 状态：收到 3 个 en。

图 3-4 所示为状态转移图。

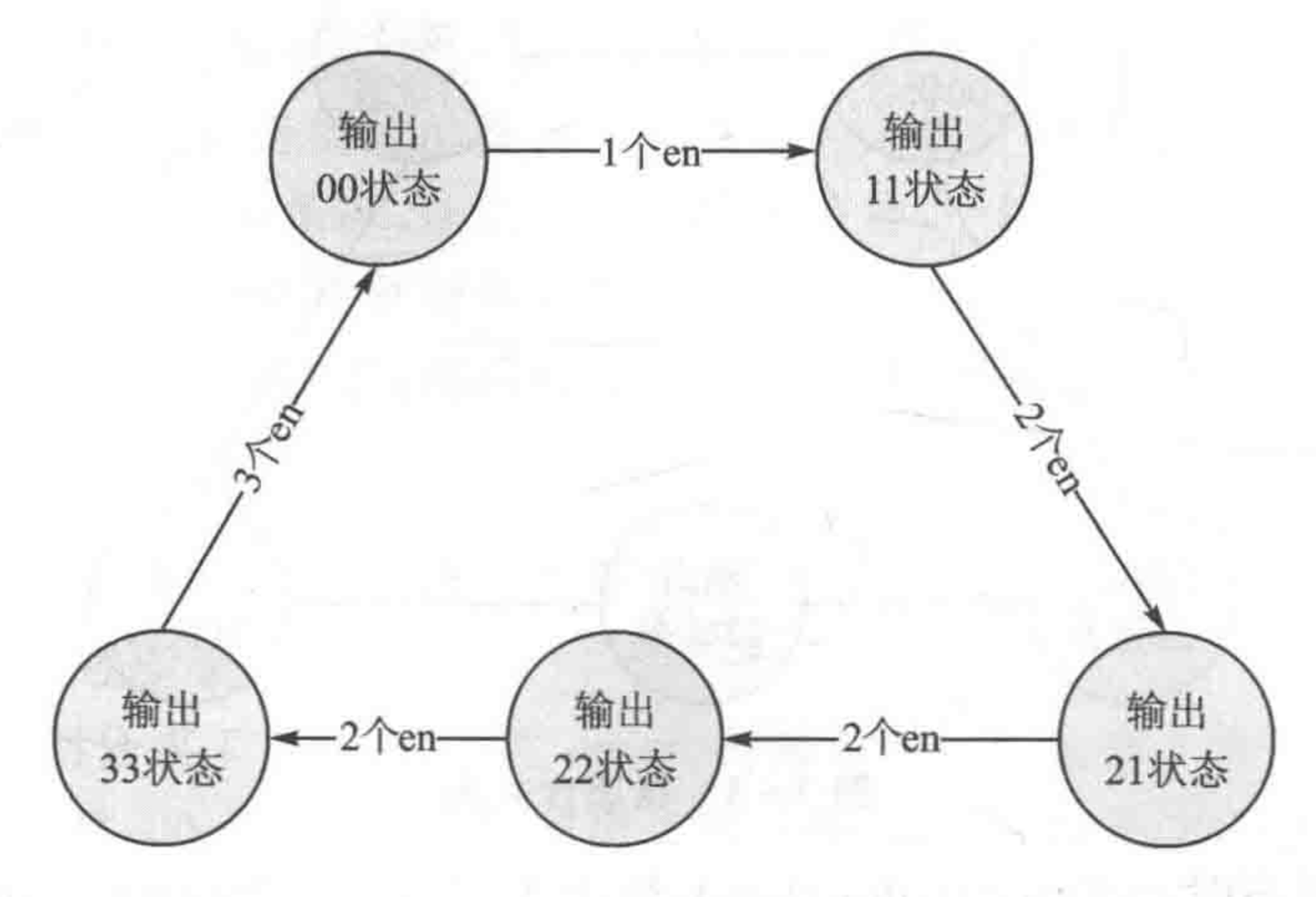


图 3-4 状态转移图

3.1.6 第五步：转移条件

转移条件是用信号来描述的条件。

将转移条件用信号来表示的过程，其实就是状态机设计的过程。转移条件必须具体到某一时钟，不能有任何偏差。

首先用英文数字来表示状态，即为状态命名，状态机可以更新如图 3－5 所示。

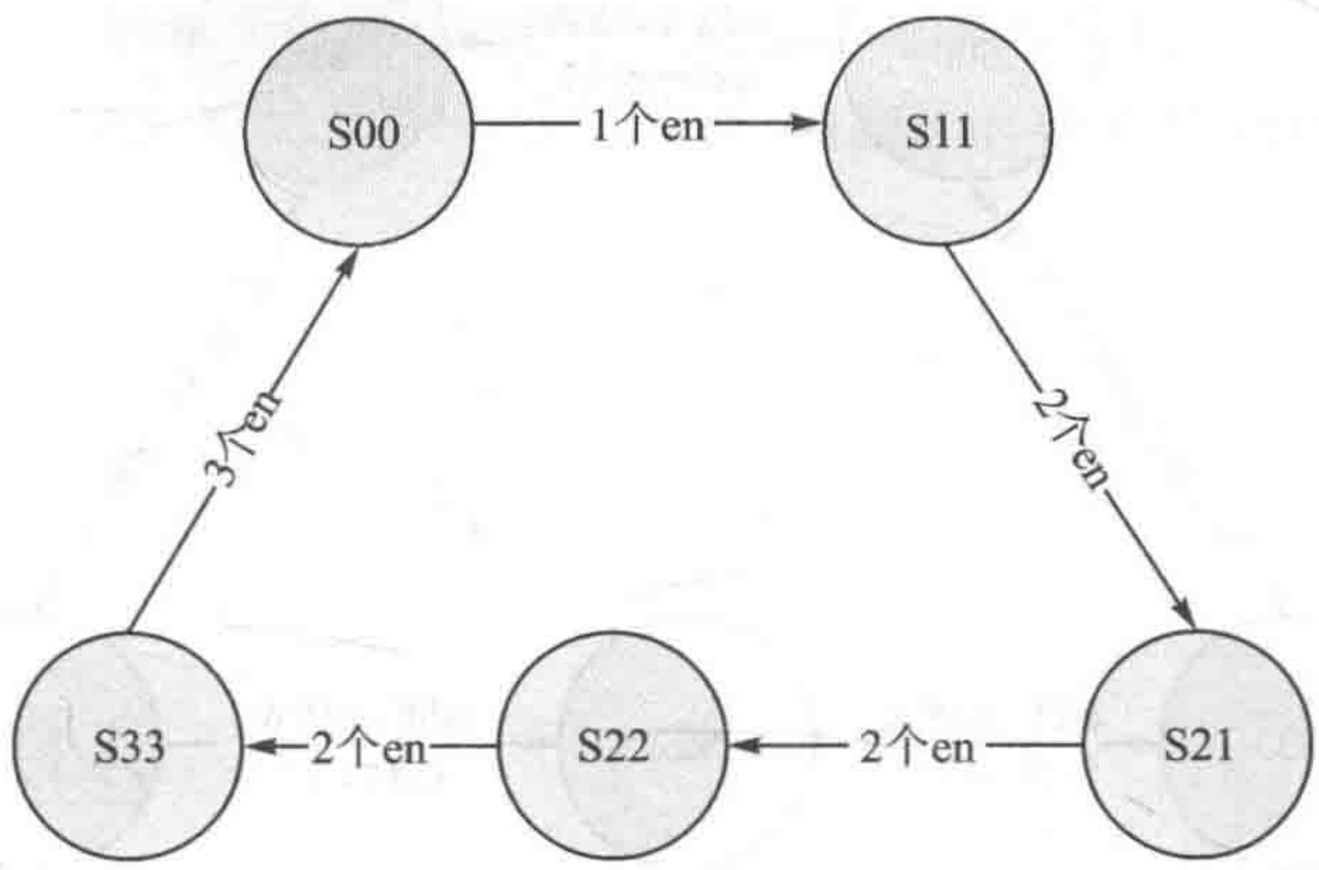


图 3－5　明确状态命名

S00 表示输出 00 状态；

S11 表示输出 11 状态；

S21 表示输出 21 状态；

S22 表示输出 22 状态；

S33 表示输出 33 状态。

在本例中，转移条件都是对 en 进行计数，为此我们需要用到计数器。设计数器为 cnt，其加 1 条件是 add_cnt，则状态机的转移条件可用信号表示为图 3－6 的形式。

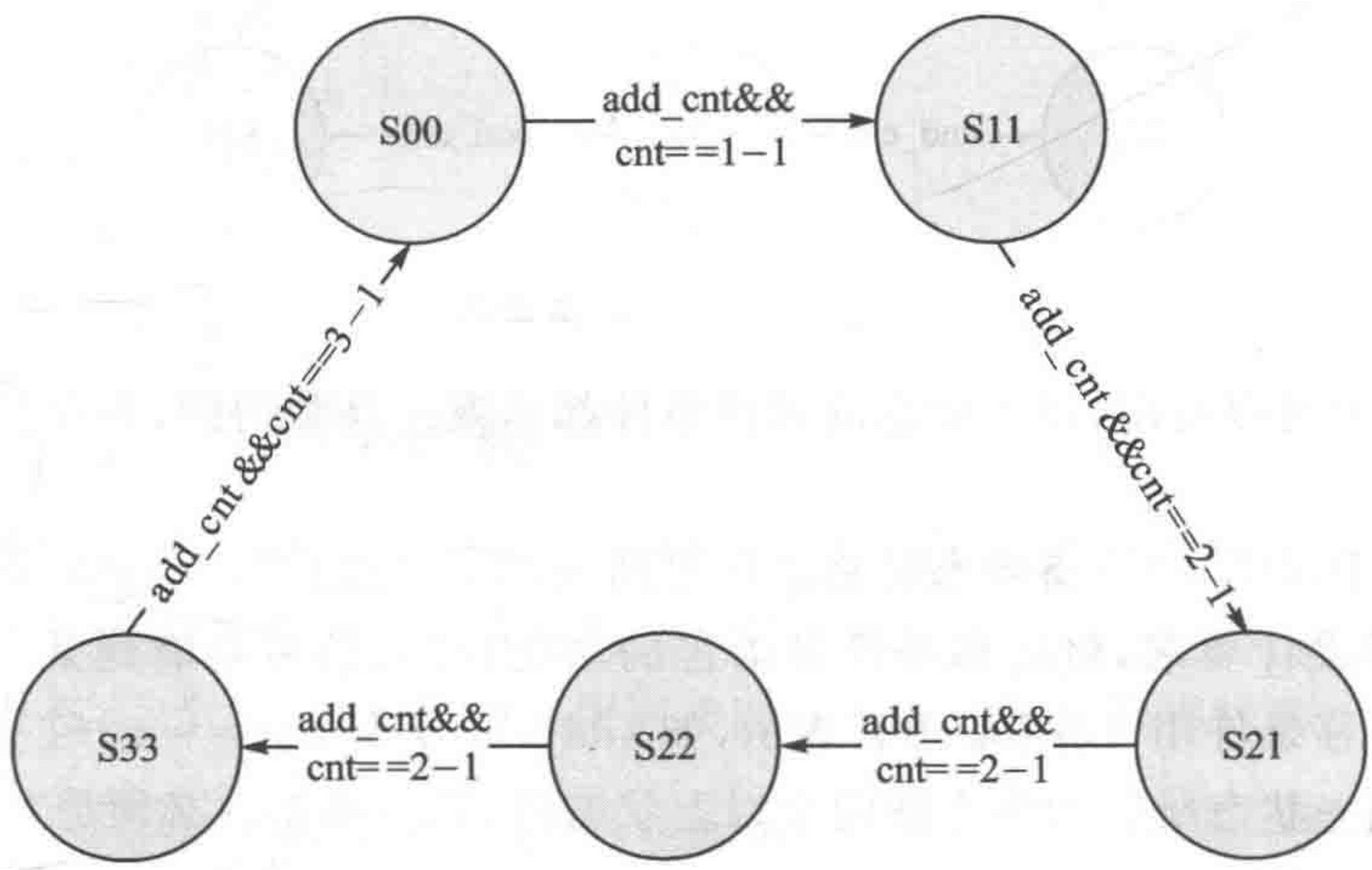


图 3－6　明确具体转移条件

根据明德扬计数器变量法，将结束值看成变量 x，此状态机可以完善如图 3-7 所示。

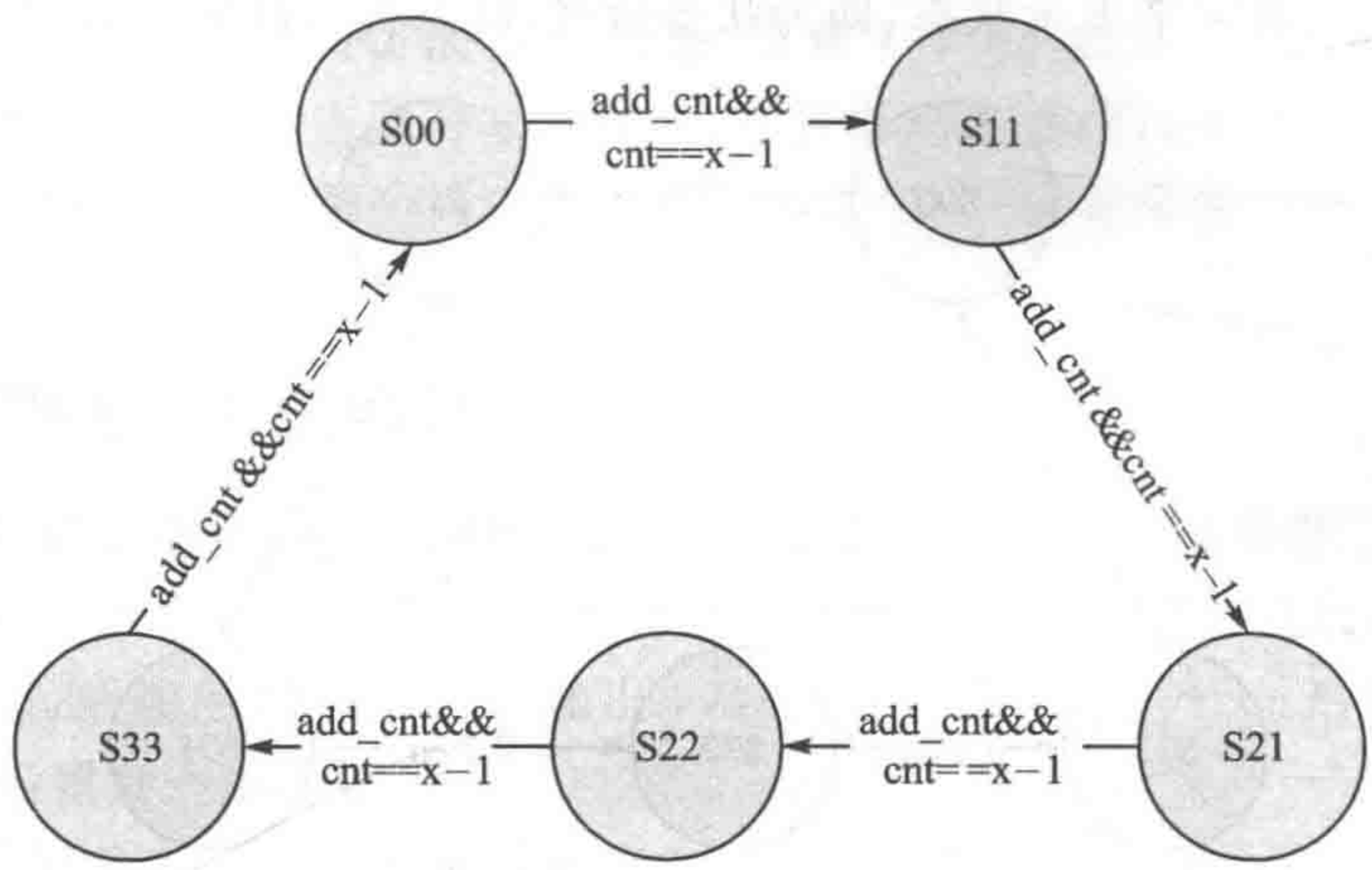


图 3-7 完善转移条件

图中的转移条件，其实都是计数器的结束条件，假设用 end_cnt 表示，其中 end_cnt = add_cnt && cnt==x-1，则可完善如图 3-8 所示。

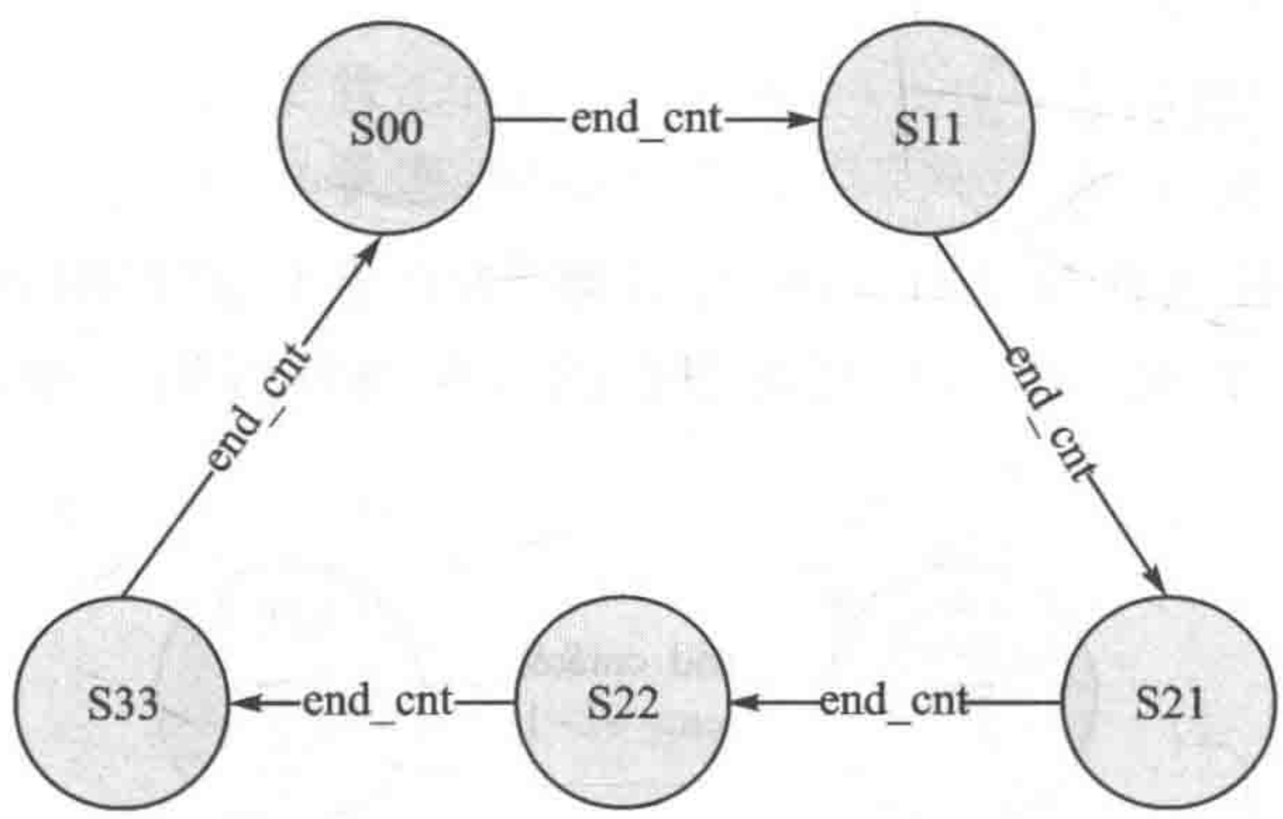


图 3-8 最终转移条件

从状态图可以看出，每个状态的转移条件都是唯一且确切的，是在某一时钟时刻触发跳转。

通过以上说明，可以将状态转移条件总结为以下三点：

① 分析设计要求，将转移条件做语言的功能描述，信号精确到某一时钟周期；

② 将转移条件用 Verilog 语言表示，如 add_cnt && cnt==x-1 这种形式；

③ 当同一状态跳转到两个或两个以上状态时，其转移条件必须是互斥的。

3.1.7　第六步:完整性检查

完整性检查是检查除状态机外其他所有信号的完整性。

当列出所有状态转移条件后,状态机代码就一定能设计出来。

除状态机外,其他信号也要做完整性检查,确定所有信号都能设计出来。

1. cnt

cnt 的初值:0;

cnt 的加 1 条件:en==1;

cnt 的结束值:$x-1$。

2. dout0

dout0 为 0:state_c==S00;

dout0 为 1:state_c==S11 || state_c==S21;

dout0 为 2:state_c==S22;

dout0 为 3:state_c==S33。

3. dout1

dout1 为 0:state_c==S00;

dout1 为 1:state_c==S11;

dout1 为 2:state_c==S21 || state_c==S22;

dout1 为 3:state_c==S33。

4. *x*

state_c==S00,$x=1$;

state_c==S11,$x=2$;

state_c==S21,$x=2$;

state_c==S22,$x=2$;

state_c==S33,$x=3$。

3.1.8　第七步:状态机代码

状态机代码是使用四段式状态机写出的状态机代码,即把完整性检查中总结的状态和状态转移条件转换成 Verilog 代码

第一段现态:一个 always 模块采用同步时序的方式描述状态转移。

```
1  always  @(posedge clk or negedge rst_n)begin
2      if(rst_n == 1'b0)begin
3          state_c <= S00;
4      end
```

```
5        else begin
6            state_c <= state_n;
7        end
8    end
```

这里，明德扬规定使用 state_c 表示当前状态，state_n 表示次态，state_c<= state_n 表示状态改变。这样命名是保证在设计时使所有状态机的状态转移都可以写成这样，从而减少编写时的错误，这就是使用明德扬状态机架构八步法的好处。

这一段代码，是不需要有任何更改的，照写就可以了。

第二段次态：一个 always 采用组合逻辑的方式判断状态转移条件，代码如下：

```
1   always  @( * )begin
2       case(state_c)
3           S00:begin
4               if(s002s11_start)begin
5                   state_n = S11;
6               end
7               else begin
8                   state_n = state_c;
9               end
10          end
11          S11:begin
12              if(s112s21_start)begin
13                  state_n = S21;
14              end
15              else begin
16                  state_n = state_c;
17              end
18          end
19          S21:begin
20              if(s212s22_start)begin
21                  state_n = S22;
22              end
23              else begin
24                  state_n = state_c;
25              end
26          end
27          S22:begin
28              if(s222s33_start)begin
29                  state_n = S33;
30              end
31              else begin
```

```
32                  state_n = state_c;
33              end
34          end
35          S33:begin
36              if(s332s00_start)begin
37                  state_n = S00;
38              end
39              else begin
40                  state_n = state_c;
41              end
42          end
43
44          default:begin
45              state_n = S00;
46          end
47      endcase
48 end
```

第三段:用 assign 定义转移条件。注意,条件一定要加上现态。

```
1  assigns002s11_start = state_c == S00 && end_cnt;
2  assigns112s21_start = state_c == S11 && end_cnt;
3  assigns212s22_start = state_c == S21 && end_cnt;
4  assigns222s33_start = state_c == S22 && end_cnt;
5  assigns332s00_start = state_c == S33 && end_cnt;
```

明德扬的这份代码有几个特点:

① always 里面的转移条件全部用信号名来表示,而具体的转移条件在上面第三段中定义。

这样写 always 时只考虑跳转的方向和命名即可,而具体的转移条件只关心具体的信号,从而做到更加专心、代码也更加整洁。还有一个好处就是,当要修改转移条件时,只需要修改第三段,而无须调整整个状态机的结构,从而避免了频繁修改,减少错误。

② 当转移不跳转时,全部写成"state_n=state_c"。

如果状态保持不变,如保持在 S00 状态,则通常的写法是"state_n=S00"。经常看到同事和学员的这部分代码需要修改,经过分析发现,原来很多人喜欢复制这部分代码,但却忘记修改这个状态。为避免这个问题,明德扬规定全部写成"state_n=state_c"的形式,从而避免了这个错误。

此写法还有一个好处:当保持不变时,有人会写成"state_n=state_n",由于这段是组合逻辑,因此会生成我们不希望的锁存器。所以明德扬规定写成"state_n=state_c"。

③ 具体的转移条件代码形式是：当前所在状态 && 跳转条件。

如上面第三段中第一行的：state_c==S00 && end_cnt。有同学可能会有疑问，没有 state_c==S00，效果是一样的。但我们认为，对本题确实是这样，但如果是更复杂的状态机，不加上这个条件则可能会出错。为了适应所有的状态机，明德扬制定了这个严谨的规则。

第四段输出：使用同步时序电路来设计每一个状态的输出，代码如下：

```
1  always  @(posedge clk or negedge rst_n)begin
2      if(rst_n == 1'b0)begin
3          dout0 <= 0;
4      end
5      else if(state_c == S00)begin
6          dout0 <= 0;
7      end
8      else if(state_c == S11 || state_c == S21)begin
9          dout0 <= 1;
10     end
11     else if(state_c == S22)begin
12         dout0 <= 2;
13     end
14     else if(state_c == S33)begin
15         dout0 <= 3;
16     end
17 end
18
19 always  @(posedge clk or negedge rst_n)begin
20     if(rst_n == 1'b0)begin
21         dout1 <= 0;
22     end
23     else if(state_c == S00)begin
24         dout1 <= 0;
25     end
26     else if(state_c == S11)begin
27         dout1 <= 1;
28     end
29     else if(state_c == S21|| state_c == S22)begin
30         dout1 <= 2;
31     end
32     else if(state_c == S33)begin
33         dout1 <= 3;
34     end
35 end
```

至此，完成了状态机设计的步骤。根据明德扬状态机规范和设计步骤设计状态机框架，代码简捷高效，易读性好，不易出错，设计时只须修改状态变化条件，代码通用性强。

3.1.9 第八步：功能代码

功能代码是补充其他信号的代码，即把完整性检查中的信号变化点转化成Verilog代码。

在完整性检查中，还有 cnt 和 x 没有设计，其代码如下：

```
1  //按照第六步第1点，写出cnt的代码
2  always  @(posedge clk or negedge rst_n)begin
3      if(rst_n == 1'b0)begin
4          cnt <= 0;
5      end
6      else if(add_cnt)begin
7      if(end_cnt)
8          cnt <= 0;
9      else
10         cnt <= cnt + 1;
11     end
12 end
13
14 assign   add_cnt = en;
15 assign   end_cnt = add_cnt && cnt == x - 1;
16
17 //按照第六步第4点，写出x的代码
18 always@( * )begin
19         if(state_c == S00)
20             x = 1;
21         else if(state_c == S11)
22             x = 2;
23         else if(state_c == S21)
24             x = 2;
25         else if(state_c == S22)
26             x = 2;
27         else
28             x = 3;
29 end
```

3.1.10 总 结

明德扬建议使用四段式状态机的设计，理由如下：

① 一段式状态机可得到锁存后的输出信号，输出比较稳定，但其在描述当前状态时需考虑下一个状态的输出，整个代码不清晰、不易于理解维护，也不利于时序约束、功能更改和调试等，容易导致逻辑功能错误。两段式状态机虽然能较好地描述状态机，但其输出为组合逻辑输出，会有毛刺问题，在精密控制时会有影响，若在输出部分加入寄存器，则可有效消除毛刺，但输出会延迟一个时钟周期。而四段式状态机能够消除组合逻辑输出的毛刺隐患，同时能够做到同步寄存器输出。

② 四段式状态机次态到现态的转变、状态间的转移、状态的输出都是独立的，非常利于阅读和维护。

③ 合理的状态划分和状态规范是简捷高效状态机设计的前提。根据上述描述的状态划分标准以及转移条件原则，明德扬四段式状态机的一般设计步骤，即所谓“四段”分别为现态（当前状态）、次态（下一状态）、转移条件和输出，而不是只有四个 always 模块，当状态机复杂时，可能会有更多的 always 模块。

通过上述例子可以看出：采用明德扬状态机架构八步法进行设计，不仅结构清晰，代码简捷易懂，不易出错，而且通用性强，易移植。熟练掌握明德扬状态机架构八步法，必能在以后的设计过程中达到事半功倍的效果。

3.2 状态机项目实践

通过状态机架构八步法的学习，相信大家对状态机的设计跃跃欲试。本节将以简易报文类型识别器、SPI 接口以及 SDRAM 接口为例，帮助读者巩固状态机的设计方法，提高大家的设计水平，希望读者通过本章内容，能熟练掌握状态机架构八步法。

3.2.1 简易报文类型识别器项目

内容：实现一个报文检测器（模块 B），报文支持检测数据报文和控制报文。信号传输架构如图 3-9 所示。

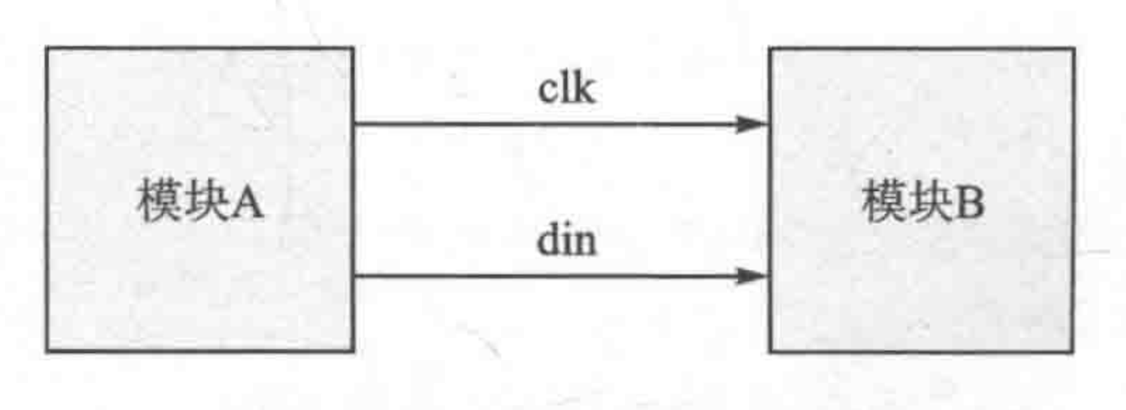


图 3-9 信号传输架构图

报文格式定义如表 3-2 和表 3-3 所列。

表3-2 数据报文格式

	2字节	1字节	2字节	1～65 535字节	4字节	
无效数据	报文头 (head)	报文类型 (pkt_type)	长度 (length)	数据 (playload)	检验码 (FCS)	无效数据

表3-3 控制报文格式

	2字节	1字节	64字节	4字节	
无效数据	报文头 (head)	报文类型 (pkt_type)	数据 (playload)	检验码 (FCS)	无效数据

复位后,检测输入的信号 din。如果连续收到1个16'h55d5,则表示检测到报文头。

报文头之后是1字节的报文类型 pkt_type。如果 pkt_type! =0,则表示数据报文;如果 pkt_type==0,则表示是控制报文。

如果是控制报文,则报文类型后是固定的64字节的 playload 和4字节的 FCS,报文结束。

如果是数据报文,则报文类型后是2 B的 length,之后是 length 字节的 playload,例如 length=127,表示后面的数据有127字节。数据最后是4字节的 FCS,报文结束。

报文结束后,继续检测报文头。在检测出报文头前,都是无效数据。信号见表3-4。

表3-4 报文信号列表

信号名	I/O	位 宽	说 明
clk	I	1	系统工作时钟
rst_n	I	1	系统复位信号,低电平有效
din	I	8	输入数据
dout	O	8	输出数据
dout_sop	O	1	输出数据报文头指示信号。表示该时刻的数据是报文的开始。注意,此处 sop 是指 pkt_type 的首字节
dout_eop	O	1	输出数据报文尾指示信号。表示该时刻的数据是报文的结束。注意,此处 eop 是指 FCS 的最后一字节
dout_vld	O	1	输出数据有效表示信号(从 pkt_type 到 FCS)

1. 明确功能

内容:实现一个报文检测器(模块B),报文支持检测数据报文和控制报文。其波

形见图 3－10 和图 3－11。

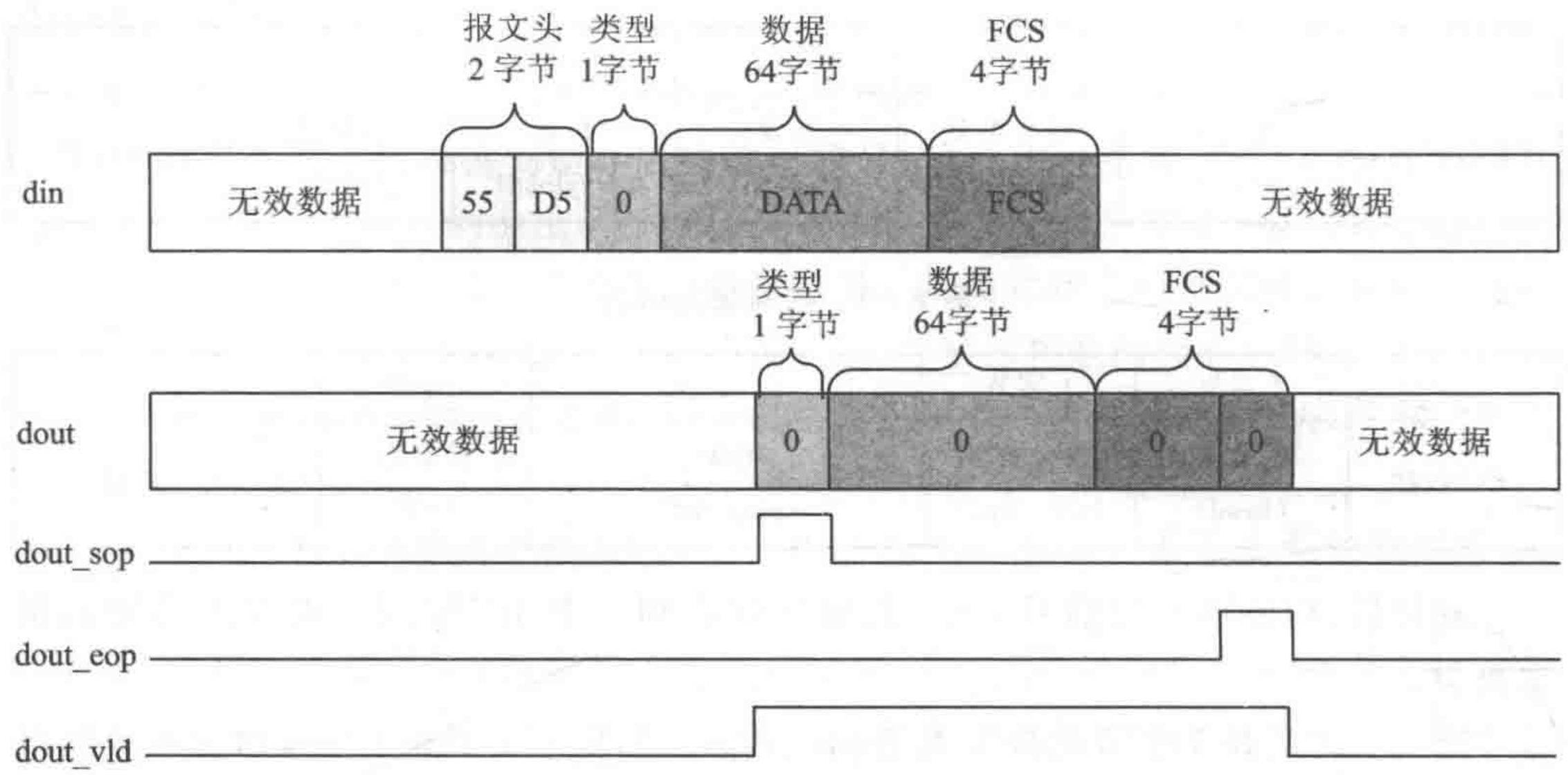


图 3－10　控制报文

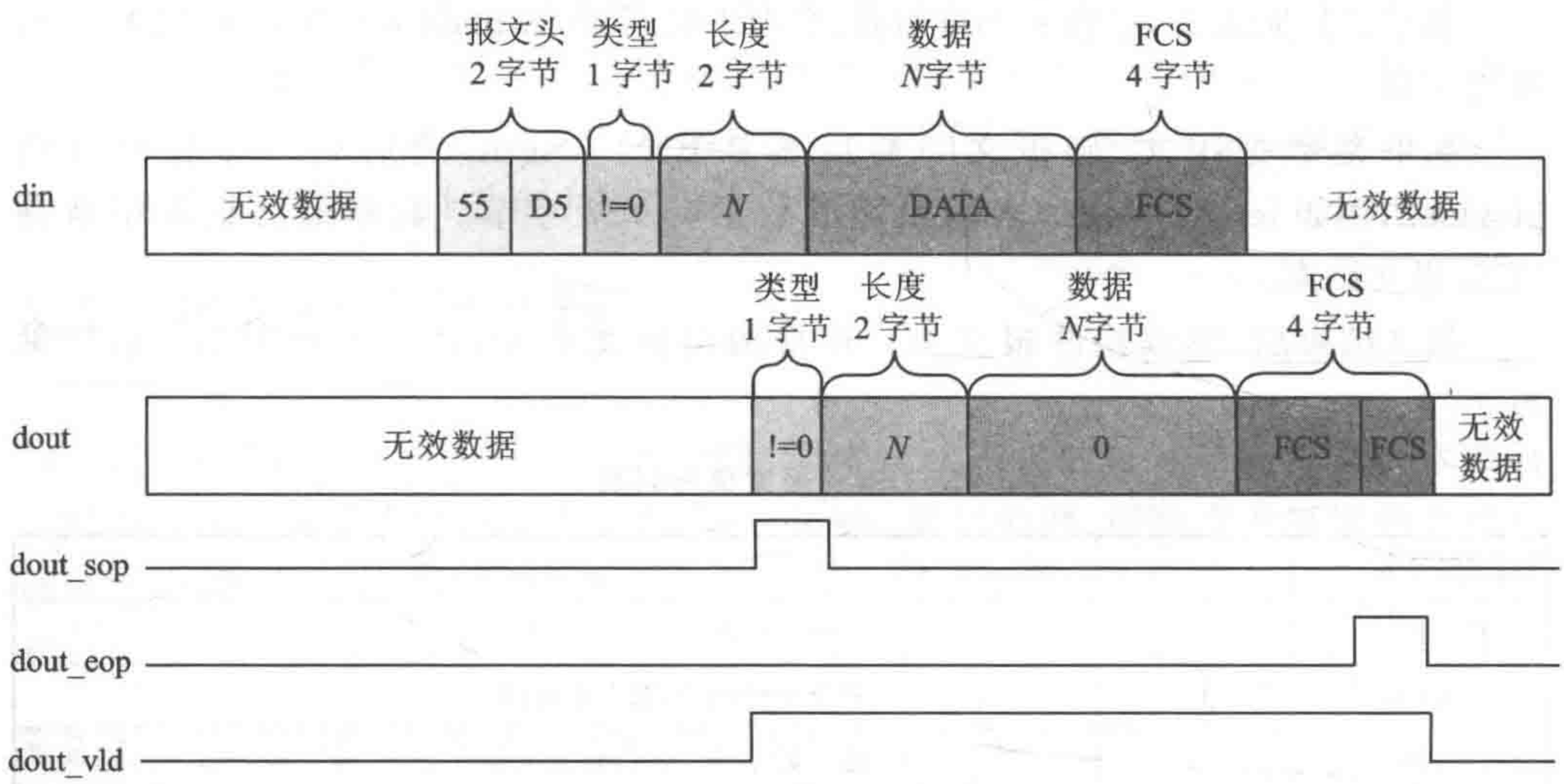


图 3－11　数据报文

2. 输出分析

(1) dout

dout 是输出数据。对于输入的 din 数据，dout 要保证输出的类型、长度、数据和 FCS 要与 din 完全一致，其他部分的内容不关心。既然是不关心，那么 dout 其他部分也与 din 一致，也是符合要求的。所以为了简单设计，dout 就设计成与 din 一样，打一拍输出。

仅从 dout 这个输出信号分析，其完全不关心处于什么状态。

(2) dout_sop

dout_sop 是报文头指示信号，在 dout 为类型状态时，输出高电平，其他时候输出低电平。

仅从 dout_sop 分析，其理想的状态机是两种状态：类型状态和其他状态。

(3) dout_eop

dout_eop 是报文尾指示信号，在 dout 为 FCS 的最后一字节时，输出高电平，其他时候输出低电平。

仅从 dout_eop 分析，其理想的状态机是两种状态：FCS 最后一字节状态和其他状态。

(4) dout_vld

dout_vld 是输出数据有效指示信号，在 dout 为类型状态直到 FCS 的最后一字节状态时，均输出高电平，其他时候输出低电平。

仅从 dout_vld 分析，其理想的状态机是三种状态：其他状态、类型状态、FCS 最后一字节状态。

(5) 其他因素

为了正确判断 FCS 的最后一字节状态，必须清楚报文的长度。但每个报文长度是不定的。

数据报文比控制报文多了一个长度字段，有必要增加一个状态：长度状态。数据报文要进入长度状态，而控制报文无须进行长度状态。

控制报文中的 DATA 长度固定是 64 字节，而数据报文长度是根据长度字段而确定的。此时，我们通过计数的方式来采集数据，因此有必要增加一个状态：数据状态。在此状态，根据需要的长度来采集数据。

3. 状态合并

综合考虑第二步的输出分析，按照明德扬状态划分原则，建议划分成 5 个状态，分别是：检测报文头状态(HEAD)、类型状态(TYPE)、长度状态(LEN)、数据状态(DATA)和检验状态(FCS)。状态划分如图 3-12 所示。

4. 状态转移

状态转移如图 3-13 所示。

HEAD→TYPE：检测输入的信号 din，收到 1 个 16'h55d5；

TYPE→LEN： 检测到 din ！=0，是数据报文；

LEN→DATA： 接收 2 字节；

TYPE→DATA：检测到 din=0，是控制报文；

DATA→FCS： 接收到预期长度；

FCS→DATA： 接收 4 字节。

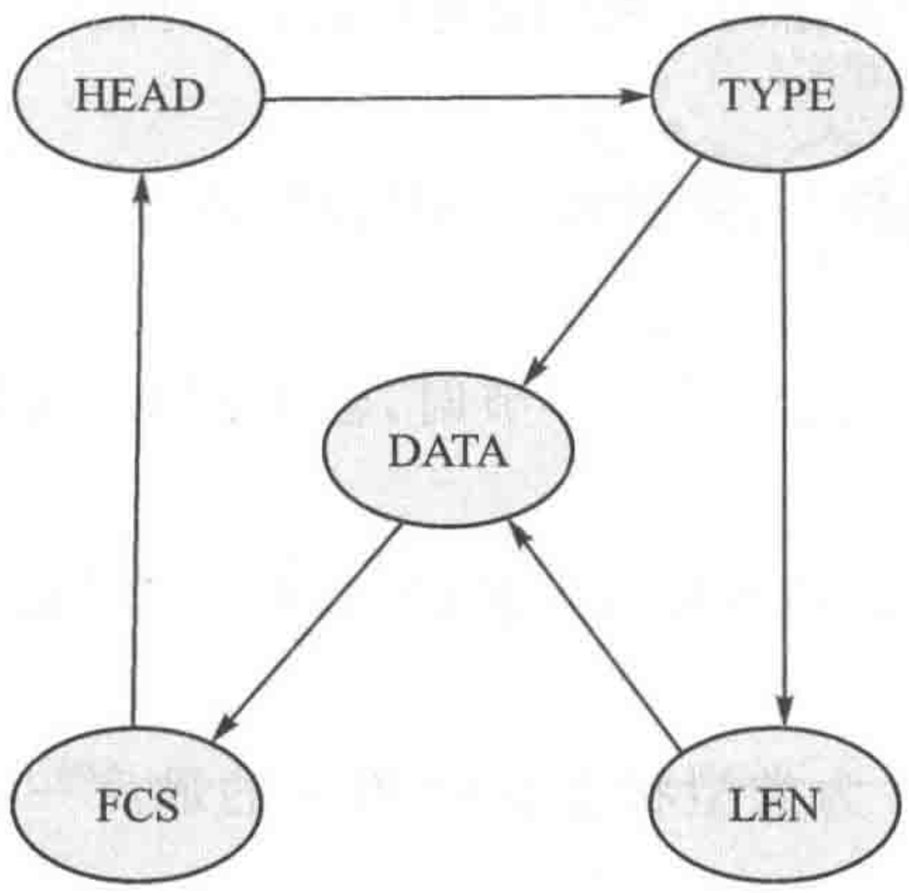


图 3-12 状态划分图

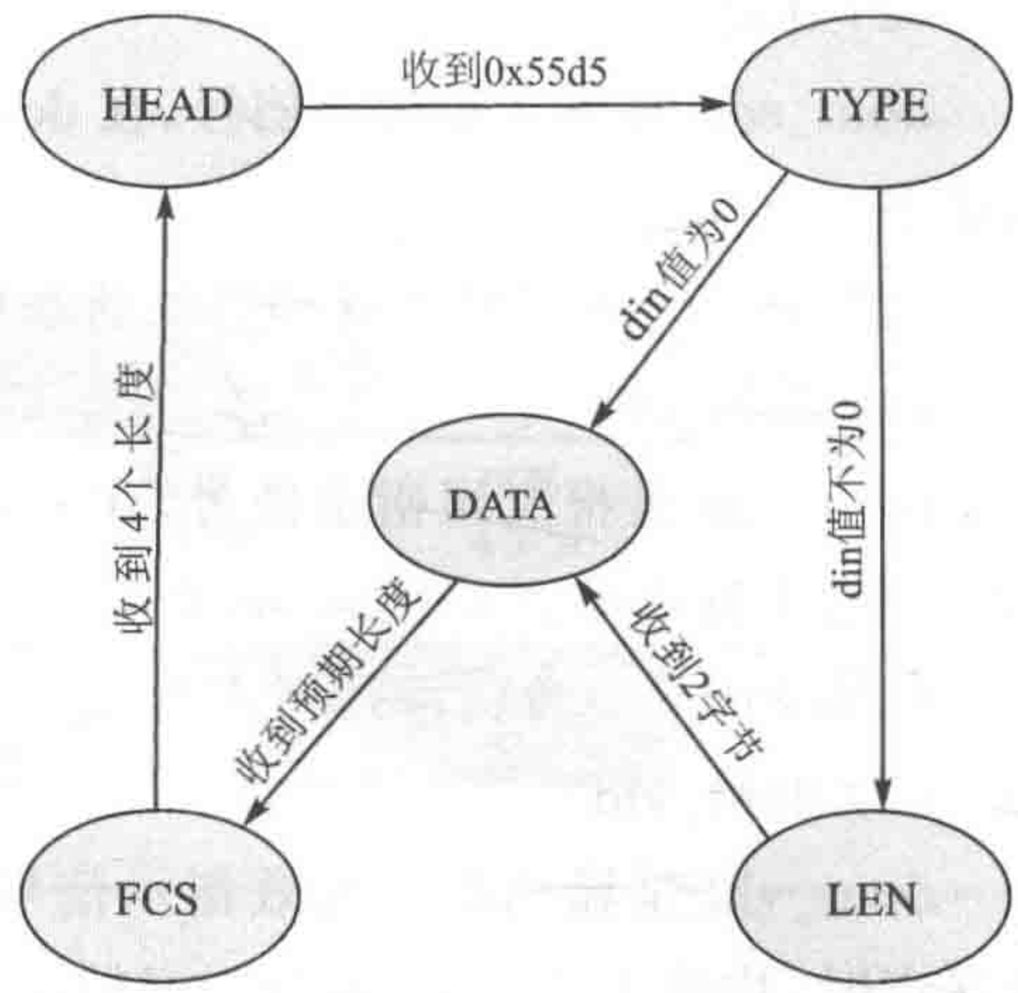


图 3-13 状态转移图

5. 转移条件

HEAD→TYPE:检测输入的信号 din,收到 1 个 0x55d5。

由于数据是逐字节传送过来的,所以要对 din 寄存一拍得到 din_ff0,如果检测到 din_ff0==8'h55 并且 din==8'hd5,就说明检测成功。所以转移条件用信号表示为 din_ff0==8'h55&&din==8'hd5。

TYPE→LEN:检测到 din !=0,是数据报文。

在此状态,只需要检测 din 的值即可,所以转移条件用信号表示为 din!=0。

LEN→DATA:接收 2 字节。

在 LEN 状态,需要计数 2 字节,才跳到 DATA 状态,因此需要计数器 cnt,从而转移条件用信号表示为 add_cnt && cnt==2-1。

TYPE→DATA:检测到 din=0,是控制报文。

在此状态,只需要检测 din 的值即可,所以转移条件用信号表示为 din!=0。

DATA→FCS:接收到预期长度。

在 LEN 状态,需要计数预期长度的个数,才跳到 FCS 状态,因此需要一个计数器,此时可以共用前面的计数器 cnt,根据明德扬计数器变量法,假设预期长度为 x,从而转移条件用信号表示为 add_cnt && cnt==x-1。

如果是控制报文,x 的值为 64,则在 TYPE 跳到 DATA 状态时可以赋值给 x 为 64。如果是数据报文,x 的值在 LEN 状态时获取,则第 1 字节获取高 8 位,第 2 字节获取低 8 位。

FCS→HEAD:接收 4 字节。

在 FCS 状态,需要计数 4 个数据,才跳到 HEAD 状态,因此需要一个计数器,此时可以共用前面的计数器 cnt,从而转移条件用信号表示为 add_cnt &&

cnt==4-1。

综上，明确状态转移条件如图 3-14 所示。

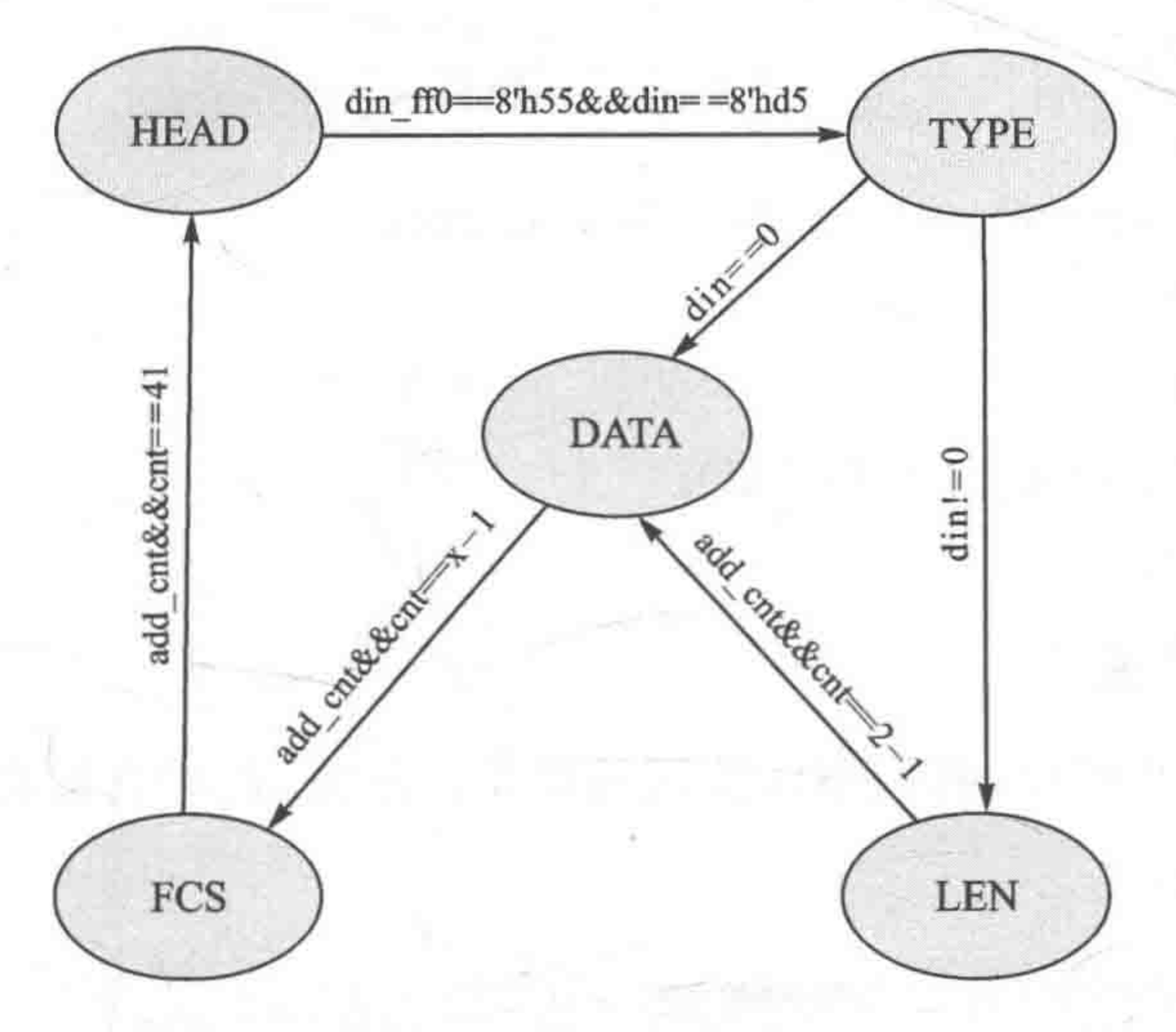


图 3-14　明确状态转移条件

6. 完整性检查

当列出所有状态转移条件后，状态机代码就一定能设计出来。

除状态机外，其他信号也要做完整性检查，确定所有信号都能设计出来。

(1) cnt

cnt 的初值：0；

cnt 的加 1 条件：state_c==LEN || state_c==DATA || state_c==FCS；

cnt 的结束值：按照明德扬变量法，可以用 y 表示计数的个数，所以结束值为 cnt==y-1，设为 end_cnt。

(2) length

length 是指 DATA 时的长度。

如果是控制报文，length 的值为 64，则在 TYPE 跳到 DATA 状态时可以赋值给 length 为 64。如果是数据报文，length 的值在 LEN 状态时获取，则第 1 字节获取高 8 位，第 2 字节获取低 8 位。

(3) x

state_c==LEN:　x = 2;

state_c==DATA: x = length;

state_c==TYPE: x = 1;

state_c==FCS:　x = 4;

(4) dout

din 延迟一拍。

(5) dout_sop

dout_sop 为 1:state_c==TYPE;

其他情况为 0。

(6) dout_eop

dout_eop 为 1:state_c==FCS && end_cnt;

其他情况为 0。

(7) dout_vld

dout_vld 为 1:state_c! =HEAD;

其他情况为 0。

7. 状态机代码

第一段,用同步时序,将次态的值赋给现态。注意,此时直接套用模块,不要做任何更改。

```
1  always@(posedge clk or negedge rst_n)begin
2      if(!rst_n)begin
3          state_c <= HEAD;
4      end
5      else begin
6          state_c <= state_n;
7      end
8  end
```

第二段,用组合逻辑描述状态转移条件。注意,转移条件用信号来表示,信号名要按明德扬规则来命名。

```
1  always@( * )begin
2      case(state_c)
3          HEAD:begin
4              if(hea2typ_start)begin
5                  state_n = TYPE;
6              end
7              else begin
8                  state_n = state_c;
9              end
10         end
11         TYPE:begin
12             if(typ2len_start)begin
13                 state_n = LEN;
14             end
15             else if(typ2dat_start)begin
```

```
16                  state_n = DATA;
17              end
18              else begin
19                  state_n = state_c;
20              end
21          end
22          LEN:begin
23              if(len2dat_start)begin
24                  state_n = DATA;
25              end
26              else begin
27                  state_n = state_c;
28              end
29          end
30          DATA:begin
31              if(dat2fcs_start)begin
32                  state_n = FCS;
33              end
34              else begin
35                  state_n = state_c;
36              end
37          end
38          FCS:begin
39              if(fcs2hea_start)begin
40                  state_n = HEAD;
41              end
42              else begin
43                  state_n = state_c;
44              end
45          end
46          default:begin
47              state_n = HEAD;
48          end
49      endcase
50  end
```

第三段，用 assign 定义转移条件。注意，条件一定要加上现态。

```
1   assign hea2typ_start = state_c == HEAD && din_ff0 == 8'h55 && din == 8'hd5;
2   assign typ2len_start = state_c == TYPE && din!  = 0;
3   assign len2dat_start = state_c == LEN && end_cnt;
4   assign typ2dat_start = state_c == TYPE && din == 0;
5   assign dat2fcs_start = state_c == DATA && end_cnt;
6   assign fcs2hea_start = state_c == FCS  && end_cnt;
```

第四段，是输出信号设计，见功能代码部分。

8. 功能代码

```
1  //按照第六步第 1 点，写出 cnt 的代码
2  always  @(posedge clk or negedge rst_n)begin
3      if(rst_n == 1'b0)begin
4          cnt <= 0;
5      end
6      else if(add_cnt)begin
7          if(end_cnt)
8              cnt <= 0;
9          else
10             cnt <= cnt + 1;
11     end
12 end
13
14 assign add_cnt = state_c! = HEAD;
15 assign end_cnt = add_cnt && cnt == x - 1;
16
17 always  @(posedge clk or negedge rst_n)begin
18     if(rst_n == 1'b0)begin
19         din_ff0 <= 0;
20     end
21     else begin
22         din_ff0 <= din;
23     end
24 end
25
26 //按照第六步第 2 点，写出 length 的代码
27 always  @(posedge clk or negedge rst_n)begin
28     if(rst_n == 1'b0)begin
29         length <= 0;
30     end
31     else if(typ2dat_start)begin
32         length <= 64;
33     end
34     else if(state_c == LEN)begin
35         length <= {length[7:0],din};
36     end
37 end
38
```

```
39 //按照第六步第3点,写出x的代码
40 always  @( * )begin
41     if(state_c == LEN)
42         x = 2;
43     else if(state_c == DATA)
44         x = length;
45     else if(state_c == TYPE)
46         x = 1;
47     else
48         x = 4;
49 end
50
51 //按照第六步第4点,写出dout的代码
52 always  @(posedge clk or negedge rst_n)begin
53     if(rst_n == 1'b0)begin
54         dout <= 0;
55     end
56     else begin
57         dout <= din;
58     end
59 end
60
61 //按照第六步第5点,写出dout_sop的代码
62 always  @(posedge clk or negedge rst_n)begin
63     if(rst_n == 1'b0)begin
64         dout_sop <= 0;
65     end
66     else if(state_c == TYPE)begin
67         dout_sop <= 1;
68     end
69     else begin
70         dout_sop <= 0;
71     end
72 end
73
74 //按照第六步第6点,写出dout_eop的代码
75 always  @(posedge clk or negedge rst_n)begin
76     if(rst_n == 1'b0)begin
77         dout_eop <= 0;
78     end
79     else if(state_c == FCS && end_cnt)begin
80         dout_eop <= 1;
```

```
81      end
82      else begin
83          dout_eop <= 0;
84      end
85 end
86
87 //按照第六步第 7 点,写出 dout_vld 的代码
88 always  @(posedge clk or negedge rst_n)begin
89      if(rst_n == 1'b0)begin
90          dout_vld <= 0;
91      end
92      else if(state_c! = HEAD)begin
93          dout_vld <= 1;
94      end
95      else begin
96          dout_vld <= 0;
97      end
98 end
```

3.2.2 SPI 接口项目

SPI(Serial Peripheral Interface)总线系统是一种同步串行外设接口,可以使 MCU 与各种外围设备以串行方式进行通信以交换信息。本项目通过 SPI 接口实现 FPGA 与 EEPROM 芯片——AT93C46 的通信。示意图如图 3-15 所示。

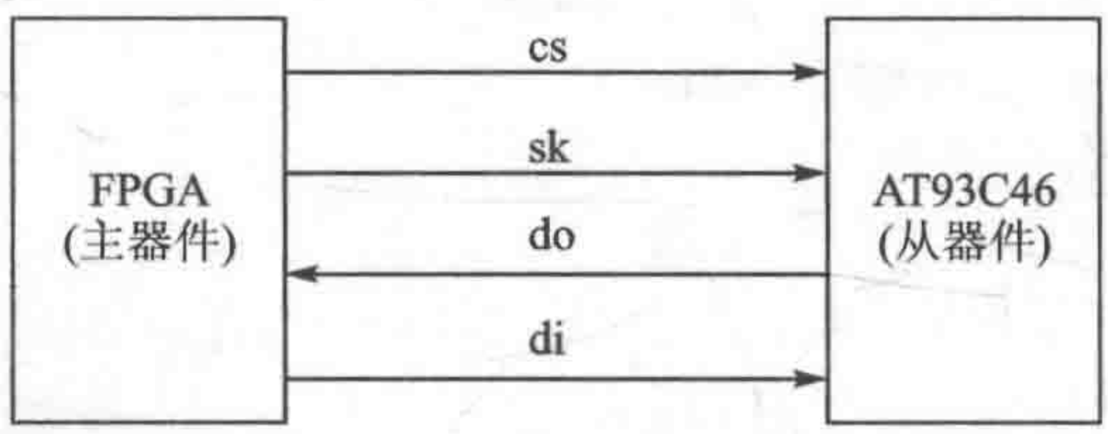


图 3-15 SPI 接口传输结构图

先查阅 AT93C46 的 datasheet,了解各引脚的作用。

cs:从器件使能信号,由主器件控制;

sk:时钟信号,由主器件产生;

do:主器件数据输入,从器件数据输出;

di:主器件数据输出,从器件数据输入。

再明确实现相互通信需要哪些操作。

1. 三种基本的操作及项目要求

(1) EWEN 时序(见图 3－16)

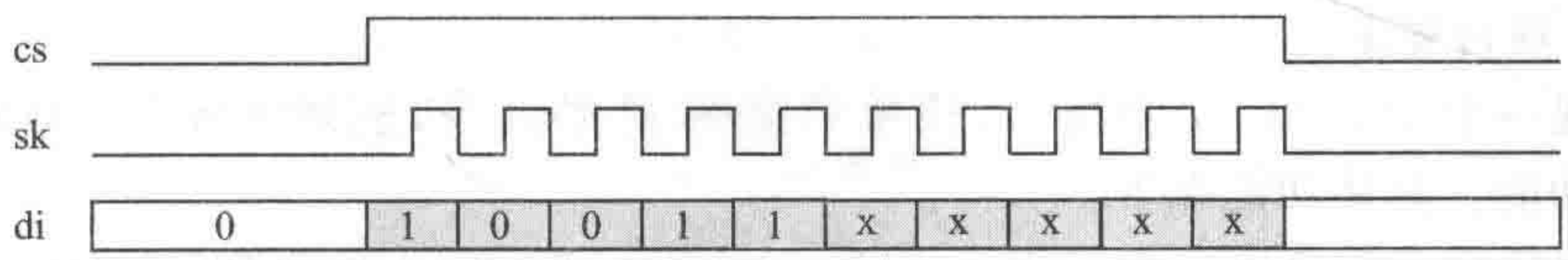


图 3－16　EWEN 时序图

EWEN 可以理解为激活设备,写、擦除(这里没有用到,有兴趣的读者可以自行查阅 datasheet)操作执行之前必须先执行完 EWEN 操作。其操作过程如下:

① cs 在开始操作时拉高,等到读操作完成,再拉低;

② sk 在 cs 拉高时,产生 10 个脉冲,脉冲周期大于 400 ns;

③ di 在 sk 的周期内输出数据,前面输出固定的指令“10011”,指示是 EWEN 操作,后面则继续输出 5 bit 的“0”。

(2) WRITE 时序(见图 3－17)

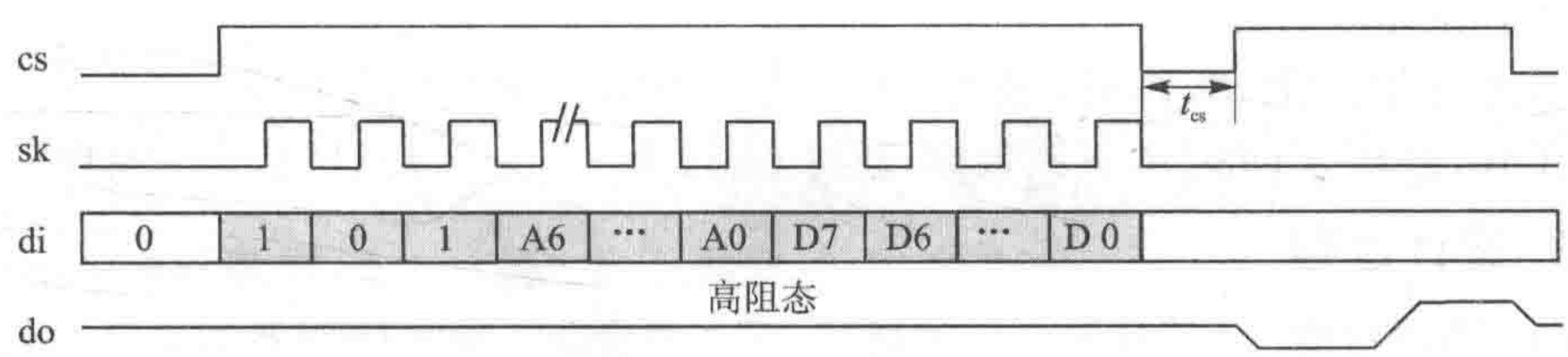


图 3－17　WRITE 时序图

写操作过程如下:

① cs 先拉高一段时间,然后拉低 t_{cs}时间,再拉高,直至检测到 D0 为 1,再拉低;

② sk 在 cs 第一段拉高时,产生 18 个脉冲,脉冲周期大于 400 ns;

③ di 在 sk 的周期内输出数据,前面输出固定的指令“101”,指示是写操作,后面则继续输出地址“AN～A0”和数据“DN～D0”。

(3) READ 时序(见图 3－18)

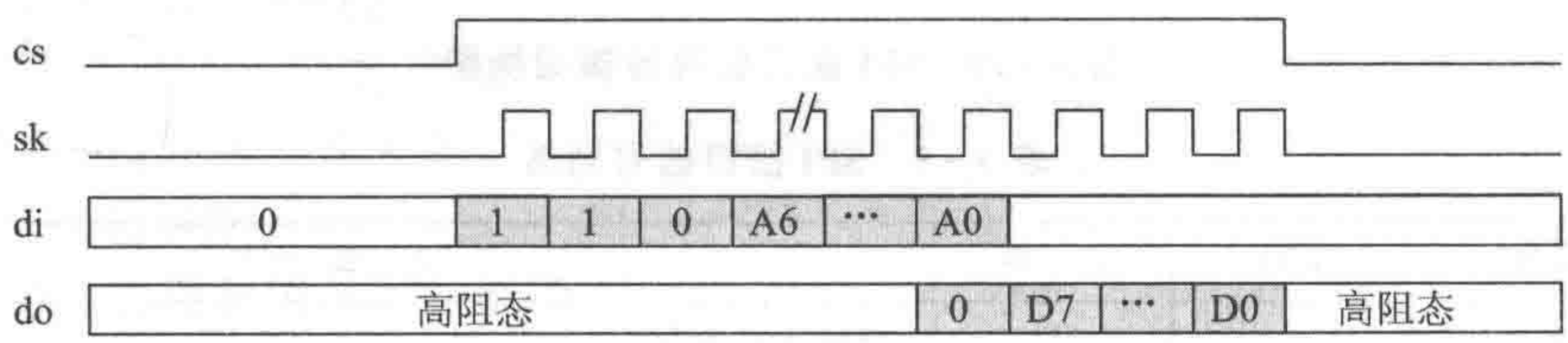


图 3－18　ERAD 时序图

读操作过程如下:

① cs 在开始操作时拉高,等读操作完成后,拉低;

② sk 在 cs 第一段拉高时,产生 18 个脉冲,脉冲周期大于 400 ns;

③ di 在 sk 的周期内输出数据，前面输出固定的指令“110”，指示是读操作，后面则继续输出地址“AN～A0”和固定的“0”；

④ 模块从 sk 的第 11 个周期开始从 do 中读取数据，每个周期读取 1 bit，共 8 bit。

(4) 项目要求

实现一个 AT93C46 的接口，该接口能够根据命令，实现 EWEN、WRITE 和 READ 功能。具体功能如下：

明德扬在变化范围内取了 1 μs 作为 t_{cs} 的值；AT93C46 时钟取 1 MHz。

上游模块在 rdy=1 时，给出 start 命令，开始进行 EWEN、WRITE 或 READ 操作。在 rdy=0 期间，start 命令无效。

当 start 有效时，如果 mode=0 则表示进行 EWEN 操作；mode=1 则表示进行 WRITE 操作；mode=2 则表示进行 READ 操作。

当 start 有效时，addr 和 wdata 都有效。

当进行 EWEN 操作时，模块将按 AT93C46 EWEN 的要求，将 addr 写入 AT93C46。

当进行 WRITE 操作时，模块将按 AT93C46 WRITE 的要求，将 addr 和 wdata 写入 AT93C46。

当进行 READ 操作时，模块将按 AT93C46 READ 的要求，将 addr 写入 AT93C46，并从 AT93C46 读到数据，通过 rdata 和 rdata_vld 返回给上游模块。

2. 设计过程

(1) 明确功能

根据题目可以画出 FPGA 与 AT93C46 之间的通信框图，如图 3-19 所示。其信号列表如表 3-5 所列。

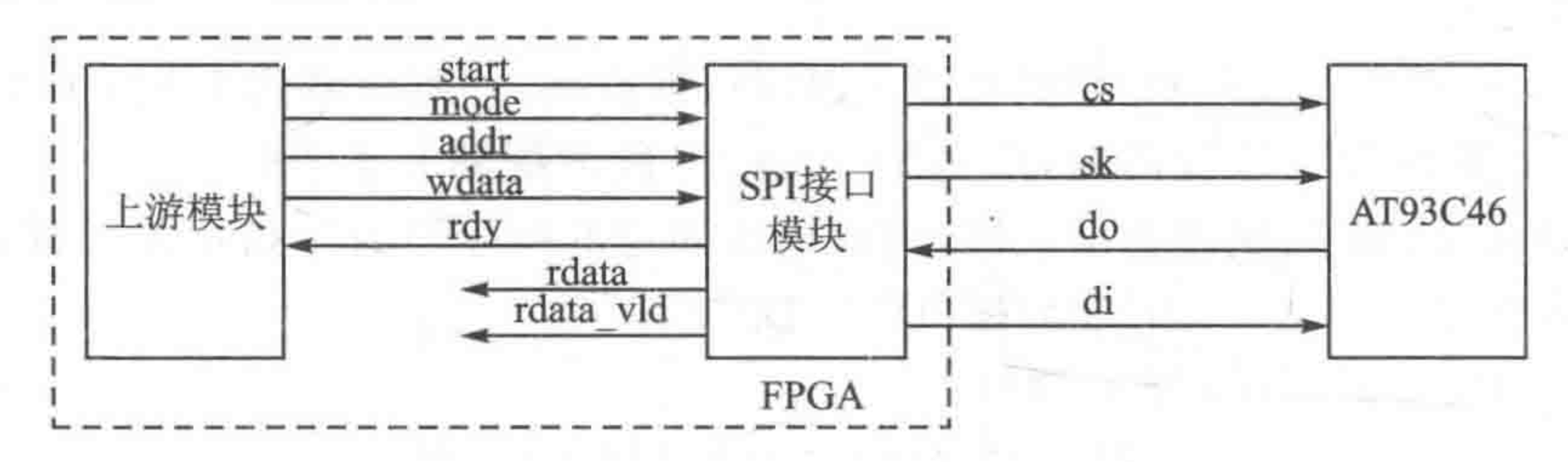


图 3-19 SPI 接口信号传输架构图

表 3-5 SPI 接口信号列表

信号名	I/O	位 宽	说 明
clk	I	1	时钟，100 MHz
rst_n	I	1	复位信号
addr	I	7	地址信号，在 start 有效时，此值有效
wdata	I	8	写数据信号，在 start 有效时，此值有效
start	I	1	开始命令，仅在 rdy=1 时才有效

续表 3-5

信号名	I/O	位 宽	说 明
mode	I	2	操作模式,start 有效时,此值有效 0:EWEN 操作; 1:WRITE 操作; 2:READ 操作; 其他:无变化
rdy	O	1	准备好信号。当 rdy 为 1 时,start 才有效;当 rdy 为 0 时,start 无效
rdata	O	8	从 AT93C46 读取的数据
rdata_vld	O	1	从 AT93C46 读到的数据有效指示信号
do	I	1	AT93C46 数据输出
di	O	1	AT93C46 数据输入
cs	O	1	AT93C46 片选
sk	O	1	AT93C46 时钟,1 MHz

(2) 输出分析

cs:在 WRITE 操作时,写入一个地址和数据后拉低,间隔 t_{cs} 拉高,等待写操作完成;READ 操作和 EWEN 操作都是在操作期间为高,操作结束拉低。

sk:引入计数器,通过计数产生 1 MHz 的芯片时钟。

rdy:从操作开始到结束一直为低电平,其他时刻为高电平。

di:根据操作的不同输出相应的值。

rdata:仅在 READ 操作时 do 的值从高位到低位,逐比特给 rdata 赋值。

rdata_vld:在 rdata 赋值结束后,拉高一个时钟周期,表示此时 rdata 有效。

(3) 状态划分

从 EWEN、READ 和 WRITE 的时序图可以发现,在不同操作中有很多阶段是相似的,总结起来有以下 4 个状态:

IDLE:初始状态,模块在等待 start 信号有效。

WR_RD:读/写状态。

TCS:片选信号拉低。

DO:等待写入操作完成。

(4) 状态转移

状态转移图如图 3-20 所示。

(5) 转移条件

确定了状态转移图后,需要明确状态转移条件:

wr_rd_start:在 IDLE 状态下收到 start 有效。

tcs_start:在 WR_RD 状态结束。

idle_start1:处于 EWEN 或 READ 模式,在 TCS 状态结束(1 μs)。

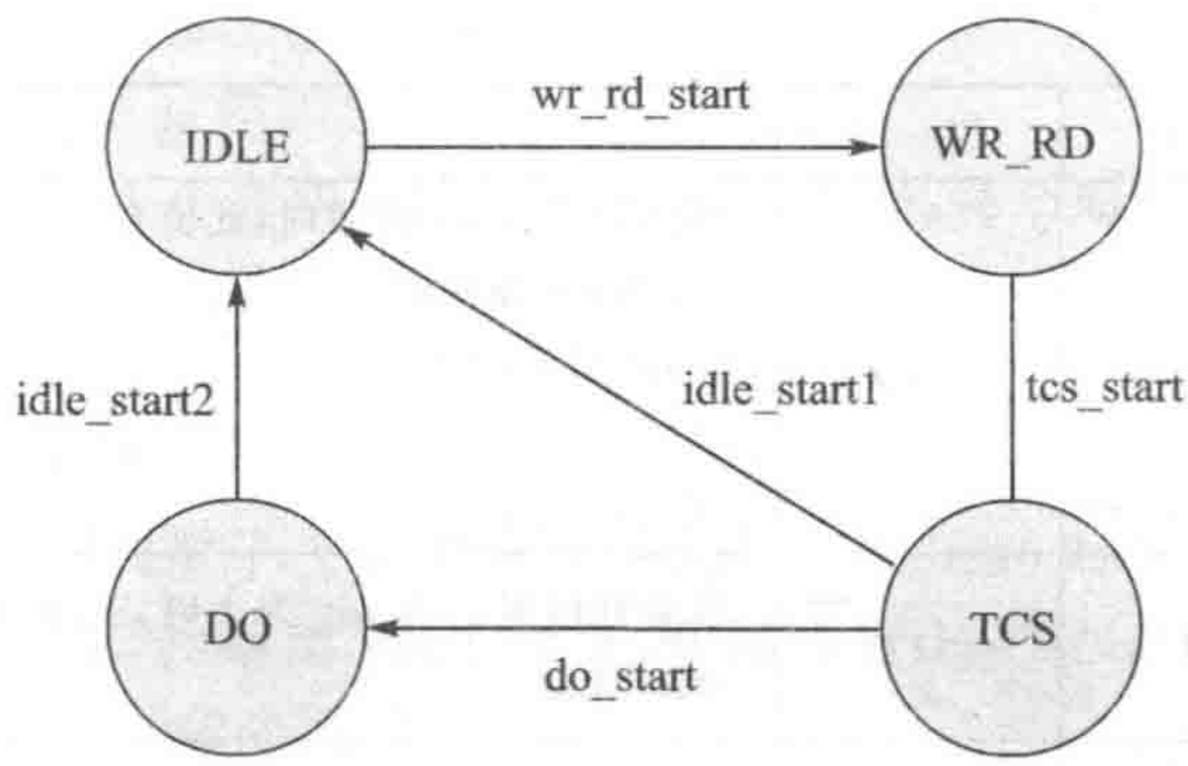


图 3-20 SPI 状态转移图

do_start：处于 WRITE 模式，在 TCS 状态结束(1 μs)。

idle_start2：处于 WRITE 模式，在 TCS 状态下，收到 do==1。

(6) 完整性检查

① 计数器 cnt

cnt 的初值：0；

cnt 的加 1 条件：当前状态为 WR_RD 或者 TCS；

cnt 的结束值：在 cnt 的加 1 条件下，计数至 99。

② 计数器 cnt1

cnt1 的初值：0；

cnt1 的加 1 条件：cnt 的结束时刻；

cnt1 的结束值：在 cnt 的结束时刻，计数到 $x-1$，包含了 rdy 需要延时的 1 μs，x 的值随着操作模式的不同而变化。

③ do

异步信号需要打两拍进行异步处理。

④ sk

sk 的初值：0；

sk 由 0 变 1：当前状态为 WR_RD 且 cnt 计数到 49；

sk 由 1 变 0：当前状态为 WR_RD 且 cnt 计数结束时刻；

其他情况保持不变。

⑤ di

di 输出条件：当前状态为 WR_RD 且 cnt==0；

其他情况保持不变。

⑥ cs

cs 初值为 0；

cs 由 0 变 1：idl2wrd_start==1 或者 tcs2do_start==1；

cs 由 1 变 0：wrd2tcs_start==1 或者 do2idl_start==1；

变化点都在状态跳转的时刻。

⑦ rdy

rdy 变化规律如下：

rdy 的值为 0：收到；

rdy 的值为 1：idl2wrd_start==1 或者 tcs2do_start==1；

变化点都在状态跳转的时刻。

⑧ rdata

rdata 变化条件：在 mode_reg==READ 时，若当前状态为 WR_RD，则在 cnt 计数结束且 cnt1 计数为 10～17，do 给 rdata 从高位到低位赋值。

⑨ rdata_vld

rdata_vld 为 1 条件：在 mode_reg==READ 时，tcs2idl_start==1；

其他情况均为 0。

⑩ dout

dout 变化条件：在 start==1 时，若 mode==0，则 dout 为{3'b100，addr，8'b0}；若 mode==1，则 dout 为{3'b101，addr，wdata}；若 mode==2，则 dout 为{3'b110，addr，8'b0}。

其他情况保持不变。

(7) 状态机代码

第一段，用同步时序，将次态的值赋给现态。注意，此时直接套用模块，不要做任何更改。

```
1  always  @(posedge clk or negedge rst_n)begin
2      if(rst_n == 1'b0)begin
3          state_c <= IDLE;
4      end
5      else begin
6          state_c <= state_n;
7      end
8  end
```

第二段，用组合逻辑描述状态转移条件。注意，转移条件用信号来表示，信号名要按明德扬规则来命名。

```
1  always  @( * )begin
2      case(state_c)
3          IDLE:begin
4              if(idl2wrd_start)begin
5                  state_n = WR_RD;
6              end
7              else begin
8                  state_n = state_c;
9              end
10         end
```

```
11          WR_RD:begin
12              if(wrd2tcs_start)begin
13                  state_n = TCS;
14              end
15              else begin
16                  state_n = state_c;
17              end
18          end
19          TCS:begin
20              if(tcs2do_start)begin
21                  state_n = DO;
22              end
23              else if(tcs2idl_start)begin
24                  state_n = IDLE;
25              end
26              else begin
27                  state_n = state_c;
28              end
29          end
30          DO:begin
31              if(do2idl_start)begin
32                  state_n = IDLE;
33              end
34              else begin
35                  state_n = state_c;
36              end
37          end
38          default:begin
39              state_n = IDLE;
40          end
41      endcase
42  end
```

第三段，用 assign 定义转移条件。注意，条件一定要加上现态。

```
1  assign idl2wrd_start = state_c == IDLE && start == 1;
2  assign wrd2tcs_start = state_c == WR_RD&& end_cnt1;
3  assign tcs2idl_start = state_c == TCS && end_cnt
4                         && (mode_reg == EWEN || mode_reg == READ);
5  assign tcs2do_start = state_c == TCS && mode_reg == WRITE && end_cnt;
6  assign do2idl_start = state_c == DO && mode_reg == WRITE && do_ff1 == 1;
```

第四段，输出信号设计，见功能代码部分。

(8) 功能代码

```
1  //根据第六步第 1 点,写出 cnt 的代码
2  always  @(posedge clk or negedge rst_n)begin
3      if(rst_n == 1'b0)begin
4          cnt <= 0;
5      end
6      else if(add_cnt) begin
7          if(end_cnt)
8              cnt <= 0;
9          else
10             cnt <= cnt + 1;
11     end
12 end
13 assign  add_cnt = state_c == WR_RD || state_c == TCS;
14 assign  end_cnt = add_cnt && cnt == 100 - 1 ;
15
16 //根据第六步第 2 点,写出 cnt1 的代码
17 always  @(posedge clk or negedge rst_n)begin
18     if(rst_n == 1'b0)begin
19         cnt1<= 0;
20     end
21     else if(add_cnt1) begin
22         if(end_cnt1)
23             cnt1 <= 0;
24         else
25             cnt1 <= cnt1 + 1;
26     end
27 end
28 assign  add_cnt1 = end_cnt;
29 assign  end_cnt1 = add_cnt1 && cnt1 == x - 1 ;
30
31 always  @( * )begin
32     if(mode_reg == EWEN)begin
33         x = 10;
34     end
35     else if(mode_reg == WRITE)begin
36         x = 18;
37     end
38     else if(mode_reg == READ)begin
39         x = 18;
40     end
```

```
41      else begin
42          x = 0;
43      end
44 end
45 //根据第六步第 3 点,写出 do 的代码
46 always  @(posedge clk or negedge rst_n)begin
47      if(rst_n == 1'b0)begin
48          do_ff0<= 0;
49          do_ff1<= 0;
50      end
51      else begin
52          do_ff0<= d0;
53          do_ff1<= do_ff0;
54      end
55 end
56
57 //根据第六步第 4 点,写出 sk 的代码
58 always  @(posedge clk or negedge rst_n)begin
59      if(rst_n == 1'b0)begin
60          sk <= 0;
61      end
62      else if(sk_high)begin
63          sk <= 1;
64      end
65      else if(sk_low)begin
66          sk <= 0;
67      end
68 end
69 assign sk_high = state_c == WR_RD && add_cnt && cnt == 50 - 1;
70 assign sk_low = state_c == WR_RD && end_cnt;
71
72 //根据第六步第 5 点,写出 di 的代码
73 always  @(posedge clk or negedge rst_n)begin
74      if(rst_n == 1'b0)begin
75          di <= 0;
76      end
77      else if(di_en)begin
78          di <= dout[17 - cnt1];
79      end
80 end
81 assign di_en = cnt == 0 && state_c == WR_RD;
82
83 //根据第六步第 6 点,写出 cs 的代码
```

```
84 always  @(posedge clk or negedge rst_n)begin
85      if(rst_n == 1'b0)begin
86          cs <= 0;
87      end
88      else if(cs_high)begin
89          cs <= 1;
90      end
91      else if(cs_low)begin
92          cs <= 0;
93      end
94 end
95 assign cs_high = idl2wrd_start || tcs2do_start;
96 assign cs_low = wrd2tcs_start || do2idl_start;
97
98 //根据第六步第 7 点,写出 rdy 的代码
99 always  @( * )begin
100     if(rdy_low)
101         rdy = 0;
102     else
103         rdy = 1;
104 end
105 assign rdy_low = start || state_c != IDLE;
106
107 //根据第六步第 8 点,写出 rdata 的代码
108 always  @(posedge clk or negedge rst_n)begin
109     if(rst_n == 1'b0)begin
110         rdata <= 0;
111     end
112     else if(rdata_en)begin
113         rdata <=  {rdata[6:0],do};
114     end
115 end
116 assign rdata_en = mode_reg == READ && state_c == WR_RD && end_cnt
117                 && cnt1 >= 10 && cnt1 < 18;
118
119 //根据第六步第 9 点,写出 rdata_vld 的代码
120 always  @(posedge clk or negedge rst_n)begin
121     if(rst_n == 1'b0)begin
122         rdata_vld <= 0;
123     end
124     else if(rdata_vld_en)begin
125         rdata_vld <= 1;
```

```
126     end
127     else begin
128         rdata_vld <= 0;
129     end
130 end
131 assign rdata_vld_en = mode_reg == READ &&tcs2idl_start;
132
133 //根据第六步第 10 点,写出 dout 的代码
134 always  @(posedge clk or negedge rst_n)begin
135     if(rst_n == 1'b0)begin
136         dout <= 0;
137     end
138     else if(start && mode == 0)begin
139         dout <= {3'b100,2'b11,13'b0};
140     end
141     else if(start && mode == 1)begin
142         dout <= {3'b101,addr,wdata};
143     end
144     else if(start && mode == 2)begin
145         dout <= {3'b110,addr,8'b0};
146     end
147 end
148
149 always  @(posedge clk or negedge rst_n)begin
150     if(rst_n == 1'b0)begin
151         mode_reg <= 0;
152     end
153     else if(start && mode == 0)begin
154         mode_reg <= EWEN;
155     end
156     else if(start && mode == 1)begin
157         mode_reg <= WRITE;
158     end
159     else if(start && mode == 2)begin
160         mode_reg <= READ;
161     end
162 end
```

3.2.3 SDRAM 接口项目

SDRAM 全称为 Synchronous Dynamic Random Access Memory,同步动态随机存储器。“同步”是指存储器工作需要同步时钟,内部命令的发送与数据的传输都以它

为基准;“动态”是指存储阵列需要不断刷新来保证数据不丢失,在掉电之后,其存储内容会消失;“随机”是指数据不是线性依次存储,而是自由指定地址进行数据读/写。

本项目使用的 SDRAM 型号是 IS42S16400F。下面总结的是与项目相关的 IS42S16400F 的知识点,可能会有遗漏或者不清晰的地方,希望读者能参考英文原版 datasheet 完成本项目,这样可以提高自己阅读英语文献的能力。

1. 通信结构图

FPGA 与 SDRAM 的通信结构图见图 3-21。

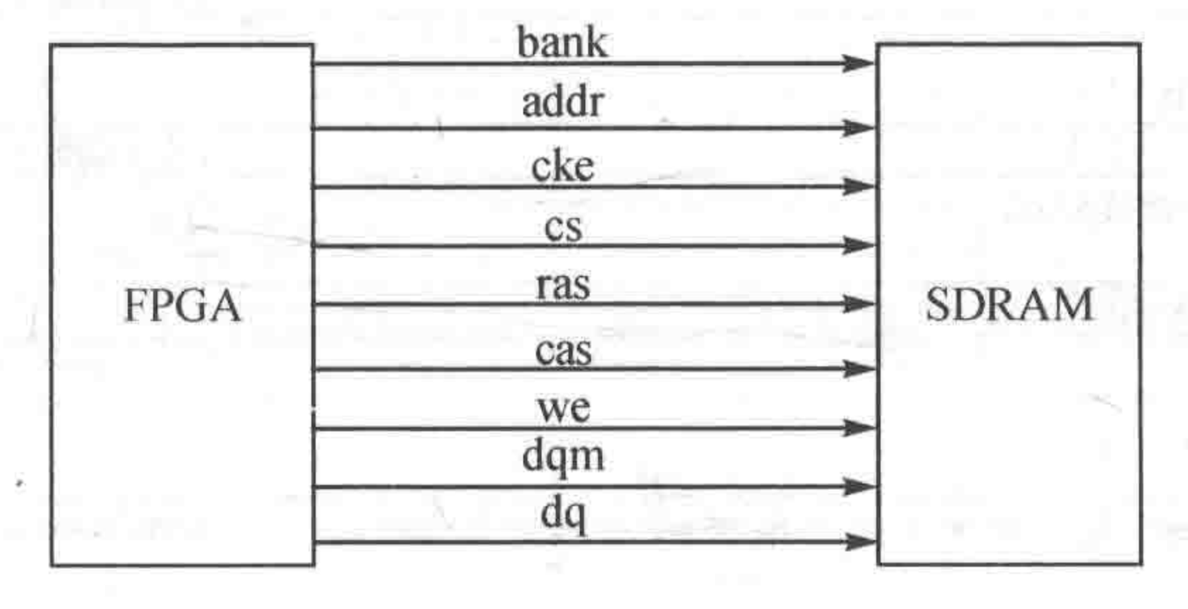


图 3-21 通信结构图

表 3-6 所列为 SDRAM 引脚说明,详细的说明请参考数据手册。

表 3-6 SDRAM 引脚说明

引 脚	说 明
A0～A11	输入地址(A0～A11 为行地址,A0～A7 为列地址)
BA0～BA1	行地址选择
DQ0～DQ15	数据输入输出(I/O)
CKE	时钟使能
L/U DQM	控制 I/O 缓冲器的下/上限字节

2. 常用操作时序

初始化时序:在对 SDRAM 进行读/写操作之前,首先要对其进行初始化,如图 3-22 所示。

① 上电、时钟稳定、DQM 拉高、CKE 拉高。100 μs 等待期间只能且至少有一个 COMMAND INHIBIT(命令禁止)命令或 NOP(空操作)命令。

② 预充电(PERCHARGE)操作。

③ 两个自动刷新命令(AUTO REFRESH)。

④ 配置模式寄存器(Load MODE REGISTER),此时配置值 CODE 从地址线输入。

读时序:SDRAM 读操作(全页读),如图 3-23 所示。

① 激活状态(ACTIVE);

② 读数据状态(READ);

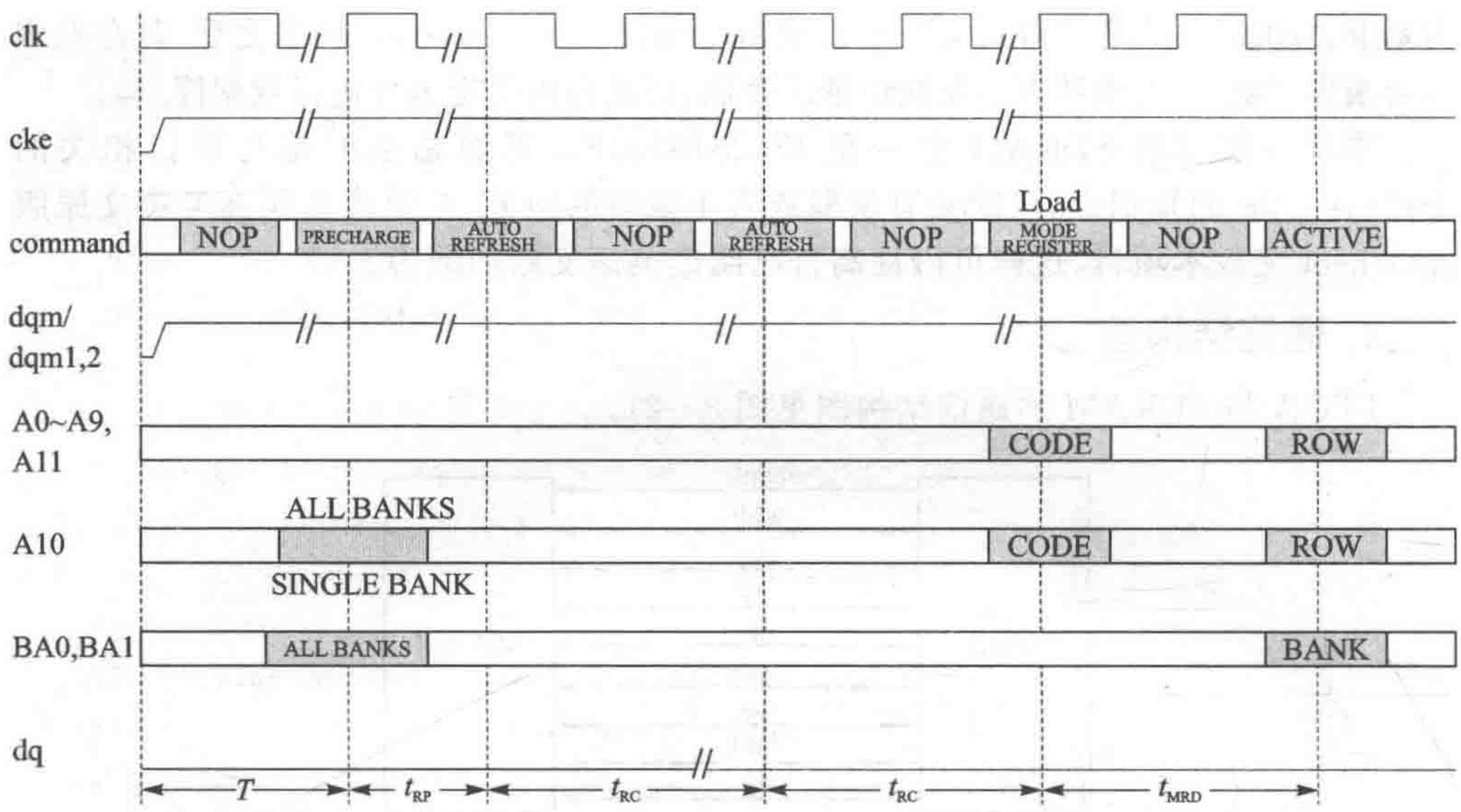


图 3－22　初始化时序

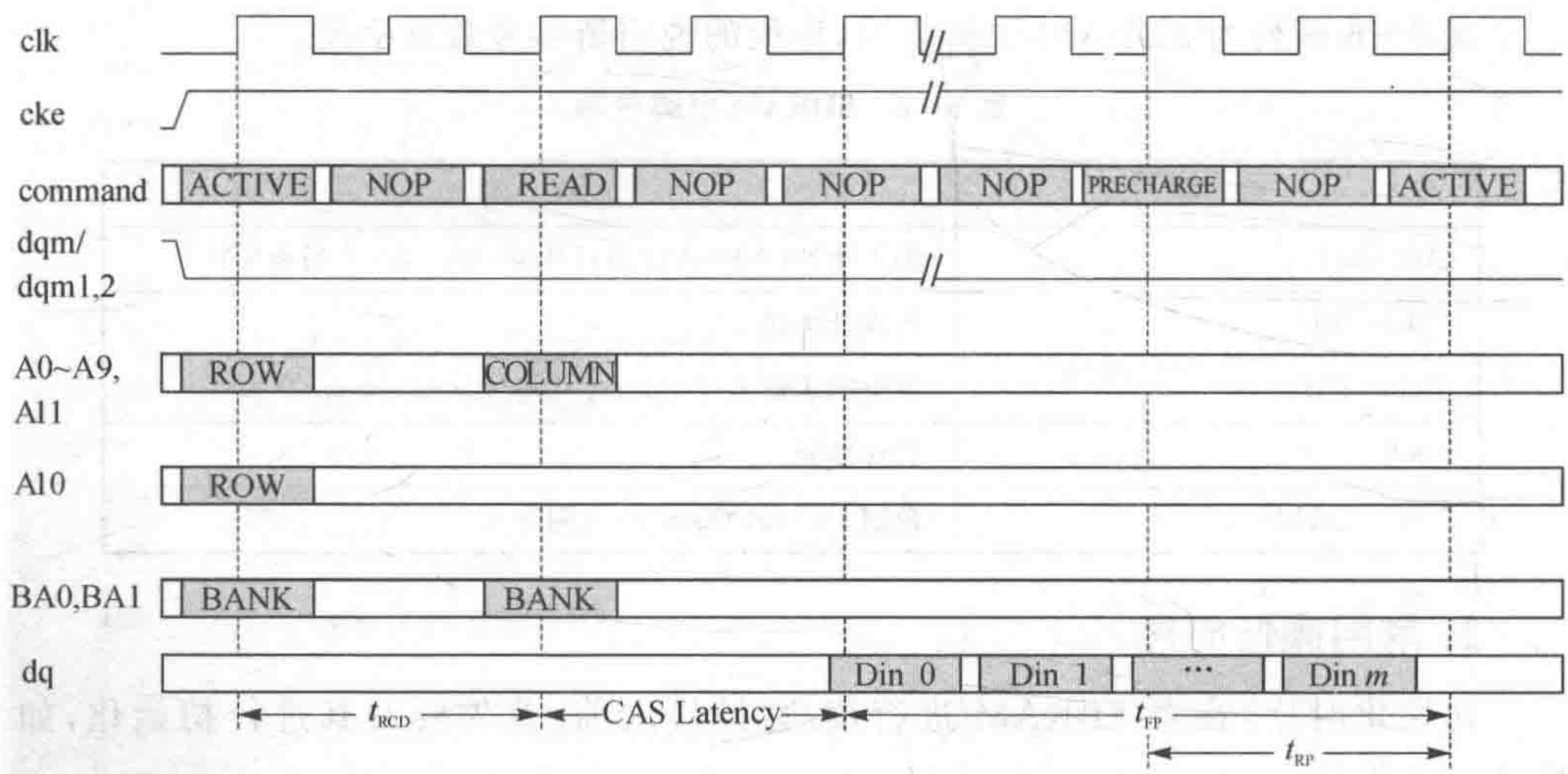


图 3－23　读时序

③ 预充电状态(PRECHARGE)。

上述阶段完成之后，读操作便已完成，进入 IDEL 状态，等待新的操作命令。

写时序：SDRAM 写操作(全页写)，如图 3－24 所示。

① 激活状态(ACTIVE)；

② 写数据状态(WRITE)；

③ 预充电状态(PRECHARGE)。

上述阶段完成之后，写操作便已完成，进入 IDEL 状态，等待新的操作命令。

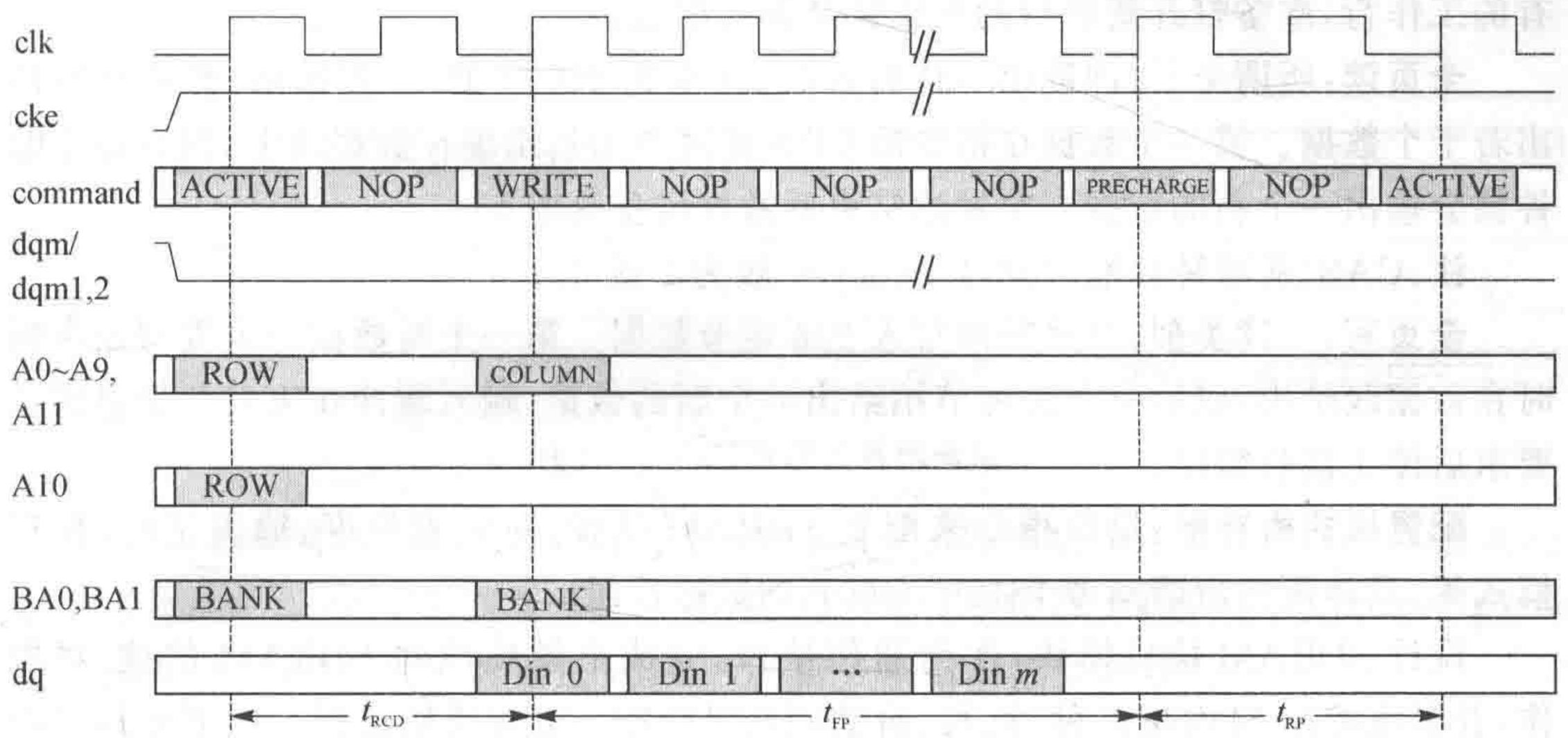


图 3－24　写时序

3. 命令解释

自动刷新：SDRAM 自动刷新操作可参考初始化过程。自动刷新是一个内部的自动操作。SDRAM 内部有一个行地址生成器（也称刷新计数器）用来自动地依次生成行地址。由于刷新是针对一行中的所有存储体进行，所以无需列寻址。而每次刷新所占用的时间为 t_{RC}，之后就可进入正常的工作状态，也就是说在 t_{RC} 时间内，所有工作指令只能等待而无法执行。64 ms 之后则再次对同一行进行刷新，如此周而复始地进行循环刷新。显然，刷新操作会对 SDRAM 的性能造成影响，这是 DRAM 相对于 SRAM（静态内存，无需刷新仍能保留数据）取得成本优势所付出的代价。

由于 SDRAM 是电容存储的器件，所以初始化完成后，FPGA 要定时给 SDRAM 刷新命令，如果不给命令，则保存到 SDRAM 上的数据就有可能丢失。

那么要隔多长时间重复一次刷新呢？目前公认的标准是，存储体中电容的数据有效保存期上限是 64 ms（毫秒，1/1 000 秒），也就是说每一行刷新的循环周期是 64 ms。这样刷新速度就是：行数量/64 ms。我们在看内存规格时，经常会看到 4 096 Refresh Cycles/64 ms 或 8 192 Refresh Cycles/64 ms 的标识，这里的 4 096 与 8 192 就代表这个芯片中每个 L-BANK 的行数。刷新命令一次对一行有效，发送间隔也是随总行数而变化，4 096 行时为 15.625 μs（微秒，1/1 000 毫秒），8 192 行时就为7.812 5 μs。

本项目的芯片共有 4 096 行，所以每隔 64 ms/4 096＝15 625 μs 就要给一个刷新命令。

预充电：每次刷新所占用的时间为 t_{RP}。读/写完之后，如果要对同一个 BANK 的另一行进行寻址，就要将原来有效的行关闭，重新发送行/列地址。BANK 关闭现

有的工作行，准备打开新的行的操作就是预充电过程。

全页读：所谓全页，即读出一行的数据（这里是 256 字节）。先激活，然后连续读出若干个数据。第一个数据在指定的 CAS 延时节拍后出现在数据线上，以后每个时钟便会输出一个新的数据。突发读结束后数据线呈高阻态。

注：CAS（从时钟存取时间）Latency 一般为 2 或 3。

全页写：与读类似，全页写即写入 256 字节数据。第一个写数据与突发写命令同时在数据线给出，以后每个时钟节拍给出一个新的数据，输入缓冲在突发数据量满足要求后停止接收数据。

配置模式寄存器：给出指令来配置 SDRAM 的读/写数据长度、输出延时、操作模式等，具体配置值请参考 IS42S16400F 的数据手册。

设计 SDRAM 接口模块，作为通信接口，完成上游模块对 SDRAM 的读/写操作，并要求控制 SDRAM，使读、写、自动刷新操作按一定优先级进行。以下分别说明上游模块的写、读操作时序与操作优先级。

4. 写操作时序

写操作时序，如图 3－25 所示。

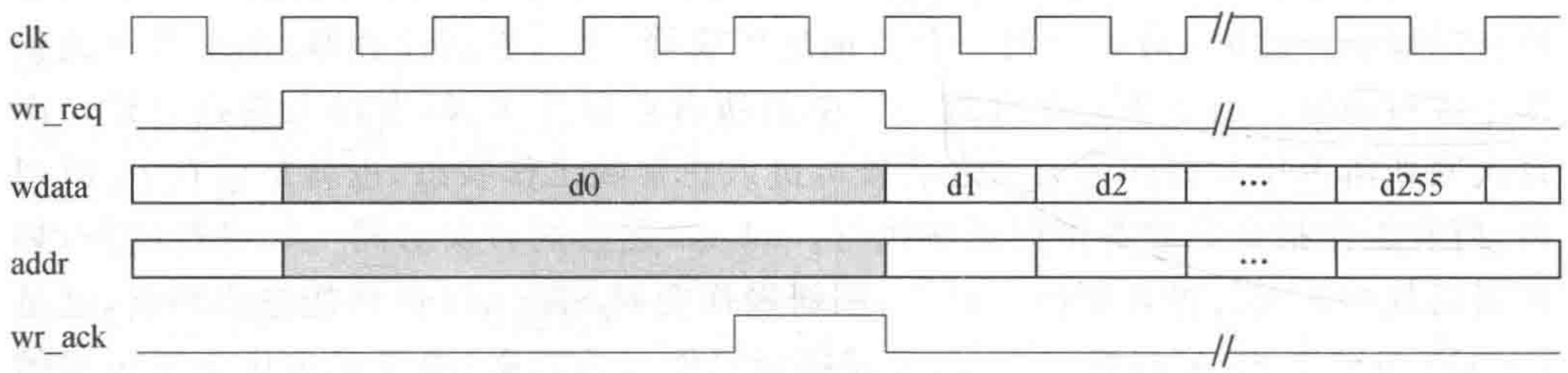


图 3－25　写请求应答信号时序

写操作的过程如下：

① 上游模块给出写请求，即将 wr_req 置为 1，同时给出第一个要写的数据 d0，以及要写的地址 waddr；如果本模块可以处理写请求时，则将 wr_ack 产生一个高电平脉冲，表示应答。

② 上游模块收到 wr_ack＝1 后，应该及时将 wr_req 撤销，同时后面每个时钟依次给出剩余要写的 255 个数据。

③ 本模块产生 wr_ack 高电平脉冲后，依次将 d0～d255 等 256 个数据写到 SDRAM。

5. 读操作时序

读操作时序，如图 3－26 所示。

读操作的过程如下：

① 上游模块给出写请求，即将 rd_req 置为 1，同时要写的地址 raddr。

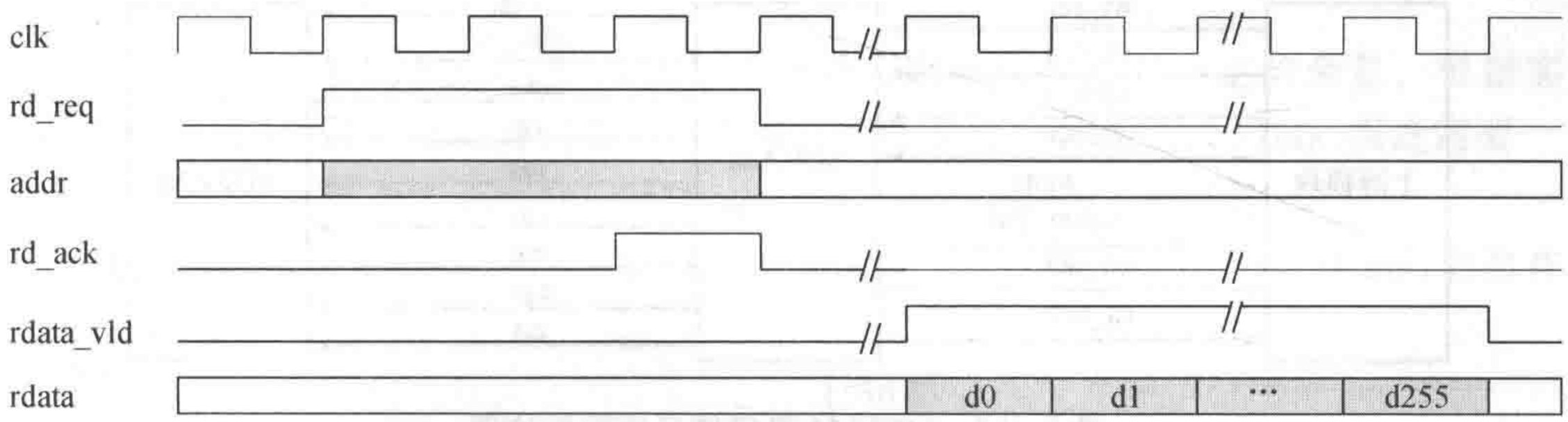


图 3-26　读请求应答信号时序

② 如果本模块可以处理读请求时，则将 rd_ack 产生一个高电平脉冲，表示应答。本模块产生 rd_ack 高电平脉冲后，从 SDRAM 中读取 1 行数据，依次通过 rdata 接口传送给上游模块，同时通过 rdata_vld 信号来指示数据有效。

6. 优先级说明

(1) 刷新优先的原则

数据手册要求 64 ms 内给 4 096 个刷新命令，这是必须的硬件要求。对于本项目，我们的做法是在 64 ms 内平均给出 4 096 个刷新命令。

为了保证刷新的可靠性，必须采用刷新优先原则，即在空闲状态下，如果同时有刷新请求、读请求和写请求，则必须要优先满足刷新请求，读请求和写请求暂不响应，让上游模块等待。如果不采用刷新优先原则，当一直有读/写请求时，虽不响应刷新操作，但会造成数据丢失。

如果模块处于读/写状态，当有刷新请求时，则应该等待读/写完一页，处于空闲状态后，再响应刷新请求。细心的读者会想到一个问题，如果等待读/写完再刷新，那么完成一行刷新的刷新时间肯定会超过 64 ms，如何解决这个问题呢？请读者认真思考，解答将在设计部分的明确功能中给出。

(2) 读、写交替原则

读、写交替原则是指读和写的优先级应该轮流变换。否则，如果读优先级高于写，则有可能某一时刻连续有读操作，而写可能会阻塞很长时间，从而造成要临时保存很多数据。同理，也不能是写优先级高于读优先级，否则可能出现读数据非常慢。读和写的优先级应该交替，即：

- 当同时出现读、写请求时，如果上次一执行了读操作，则此时执行写操作；如果上一次执行了写操作，则此时执行读操作。
- 如果没有同时出现读、写请求，则有什么请求就执行什么操作。

7. 设计过程

(1) 明确功能

根据题目要求画出设备与 SDRAM 的通信框图，如图 3-27 所示。

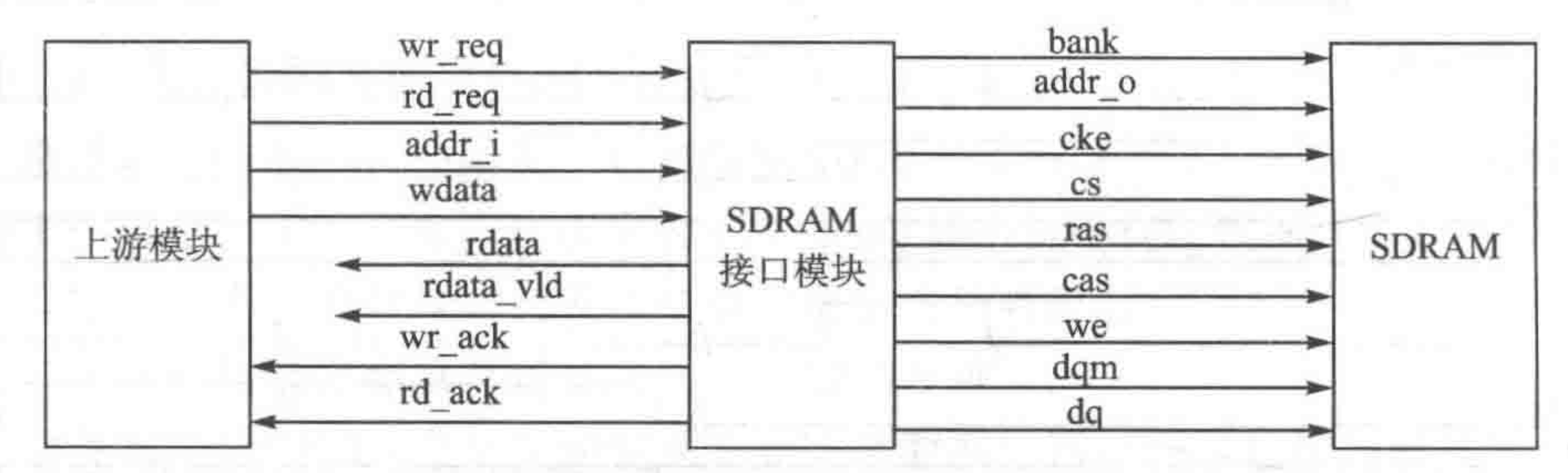


图 3－27　SDRAM 接口信号传输架构图

SDRAM 接口信号列表见表 3－7。

表 3－7　SDRAM 接口信号列表

信号名	I/O	位　宽	说　明
clk	I	1	系统工作时钟 100 MHz
rst_n	I	1	系统复位信号，低电平有效
wr_req	I	1	写请求信号
rd_req	I	1	读请求信号
addr_i	I	22	读、写地址。 [21:20]：bank 地址 [19：8]：行地址 [7:0]：列地址
wdata	I	16	写数据，在写请求期间发送第一个数据，收到应答信号后立刻发送剩余的数据，一个时钟一个
wr_ack	O	1	写应答信号。如果响应了写请求，则立刻回答一个时钟周期的写应答。此处要求组合逻辑输出
rd_ack	O	1	读应答信号。如果响应了读请求，则立刻回答一个时钟周期的写应答。此处要求组合逻辑输出
rdata	O	16	读到的数据
rdata_vld	O	1	读到的数据有效指示信号
cke	O	1	时钟使能信号
cs	O	1	与 SDRAM 引脚相连的 CS 信号
ras	O	1	与 SDRAM 引脚相连的 RAS 信号
cas	O	1	与 SDRAM 引脚相连的 CAS 信号
we	O	1	与 SDRAM 引脚相连的 WE 信号
dqm	O	2	与 SDRAM 引脚相连的 DQM 信号
addr_o	O	12	与 SDRAM 引脚相连的 A 信号
bank	O	2	与 SDRAM 引脚相连的 BA 信号
dq	IO	16	与 SDRAM 引脚相连的 DQ 信号

下面计算自动刷新条件：

针对优先级中提出的问题，解决办法就是给刷新操作留有一定的余量。根据实际情况，每隔 15.625 μs 就要完成一次刷新。由于系统时钟为 100 MHz，因此需要

$$15.625\ \mu s \times 100\ MHz = 1\ 562.5 \approx 1\ 562\ 个时钟周期$$

如果用 1 562 来计算间隔，实际上一轮刷新下来，总的时间并不是 64 ms，会存在余量 ΔT 为

$$\Delta T = 64\ ms - 4\ 096 \times \frac{1\ 562}{100\ MHz} = 0.020\ 5\ ms$$

即其实是有 0.020 5 ms 的余量。而在 0.020 5 ms，100 MHz 情况下，FPGA 可以打 2 050 拍。而一个页写、页读最多 262 拍（可从全页读、写时序中知道），远小于 2 050。所以，可以保证完成一行刷新的刷新时间在 64 ms 以内。这个设计技巧非常实用、巧妙，希望读者认真理解。

另外，本设计中 CODE 值为 12'b0000_0010_0111（查 datasheet 可知），表示全页连续读/写；CAS Latency 为 2。

（2）输出分析

由于本设计输出较多，仅在此处分析较为复杂的信号 command，addr_o 和 dq。

发送命令 command 是由输出信号 cs，ras，cas，we 组成的 4 bit 信号，其变化是根据当前指令来确定的，在预充电、自动刷新、加载模式、读、写、激活状态下，都有不一样的值，见表 3-8，详细介绍读者可以自行查阅对应 SDRAM 的 datasheet。

表 3-8 操作命令表

符 号	command{cs，ras，cas，we}
C_NOP	4'b0111
C_PRECHARGE	4'b0010
C_AUTO	4'b0001
C_LOAD	4'b0000
C_ACTIVE	4'b0011
C_READ	4'b0101
C_WRITE	4'b0100

输出地址 addr_o 信号与 command 类似，也随着指令的变化而变化。

下面再根据设计要求，从时序图中确定 addr_o 的值：

在预充电期间，当 addr_o[10]为 1 时，表示对所有 BANK 进行预充电；addr_o[10]为 0 时，通过片选信号 BA1 和 BA0 确定预充电的 BANK。

在初始化加载模式期间为 CODE，可以选定 SDRAM 的工作模式。

在写激活状态及读激活状态，用于选择读/写操作的起始行地址。

在写状态及读状态，用于选择读/写操作的起始列地址。

dq 是 I/O 类型的信号，数据通过 dq 写入 SDRAM，也通过 dq 从 SDRAM 中读出来，因此要使用一个三态门电路，如图 3－28 所示。

当 wdata_en 有效时，三态门导通，wdata 的值通过 dq 输入 SDRAM；当 wdata_en 无效时，三态门对外显示为高阻态，dq 与 rdata 连通，数据可以通过 dq 输出。

（3）状态划分

根据对 command，addr_o 的输出分析可以知道，一定要把不同操作下的不同指令区分开，才能更加清晰简单地设计这些信号。因此，首先要将接口模块划分为以下几个状态，如图 3－29 所示：初始化、自动刷新、读和写。

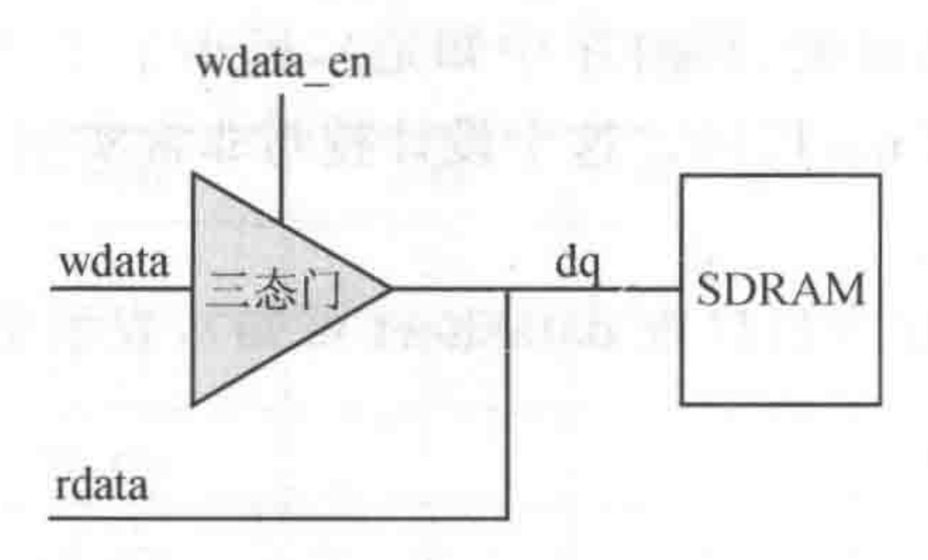


图 3－28　三态门电路

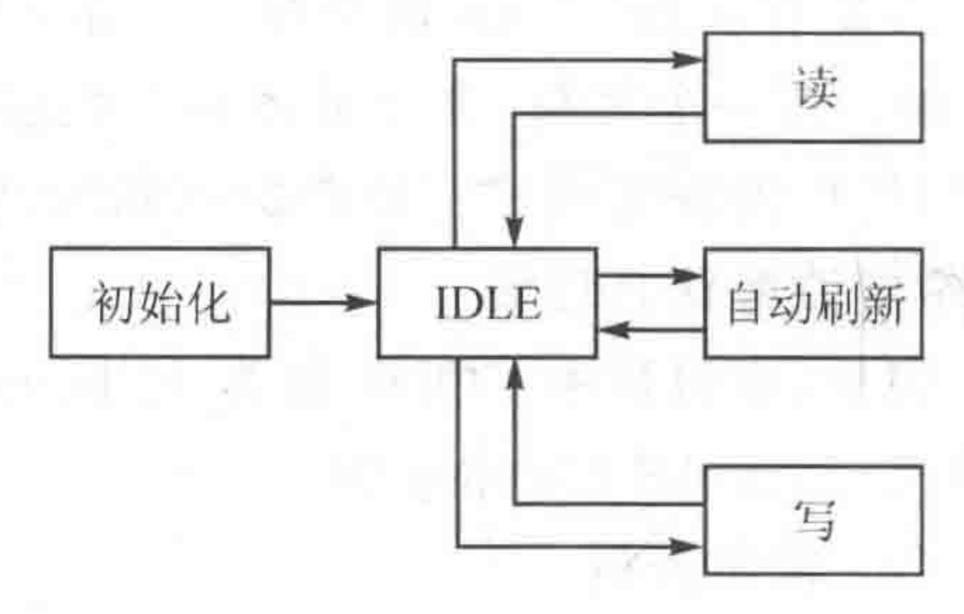


图 3－29　SDRAM 状态划分

根据这个划分思路，可以设计出状态转换图，如图 3－30 所示。

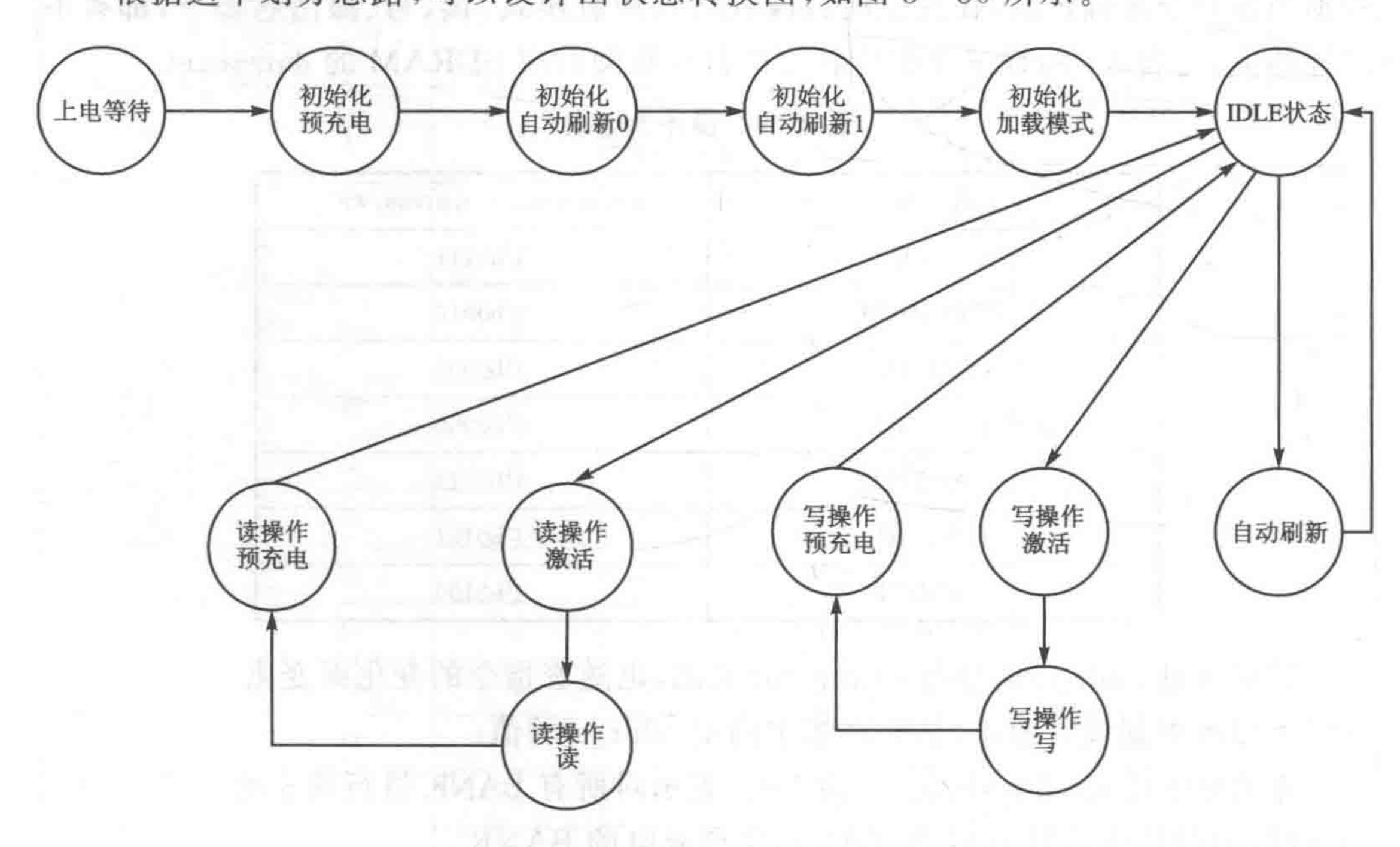


图 3－30　状态转换图

分析设计说明给出的各个功能的时序图，不难发现，各功能中有很多阶段是相同的，下面一一列举：

① 预充电状态在初始化时序、读时序和写时序中都出现。

② 激活状态在读时序和写时序中出现。

③ 刷新状态在初始化时序和自动刷新时序中出现。

由于SDRAM在某一时刻只能执行一个功能，因此不同功能中相同的阶段是不会同时出现的。我们可以把相同的阶段合并起来，最终状态划分如下：

上电等待(INIT_NOP)、预充电(PRECHARGE)、自动刷新(REFRESH)、加载模式(MODE)、IDLE(IDLE)、激活(ACTIVE)、写操作(WRITE)、读操作(READ)。

这样做的好处是，可以复用某些寄存器，虽然各状态之间的跳转会变得更复杂，但对于设计来说，会更节省资源，所以明德扬提倡这种设计形式。

(4) 状态转移

确认状态转移如图3-31所示。

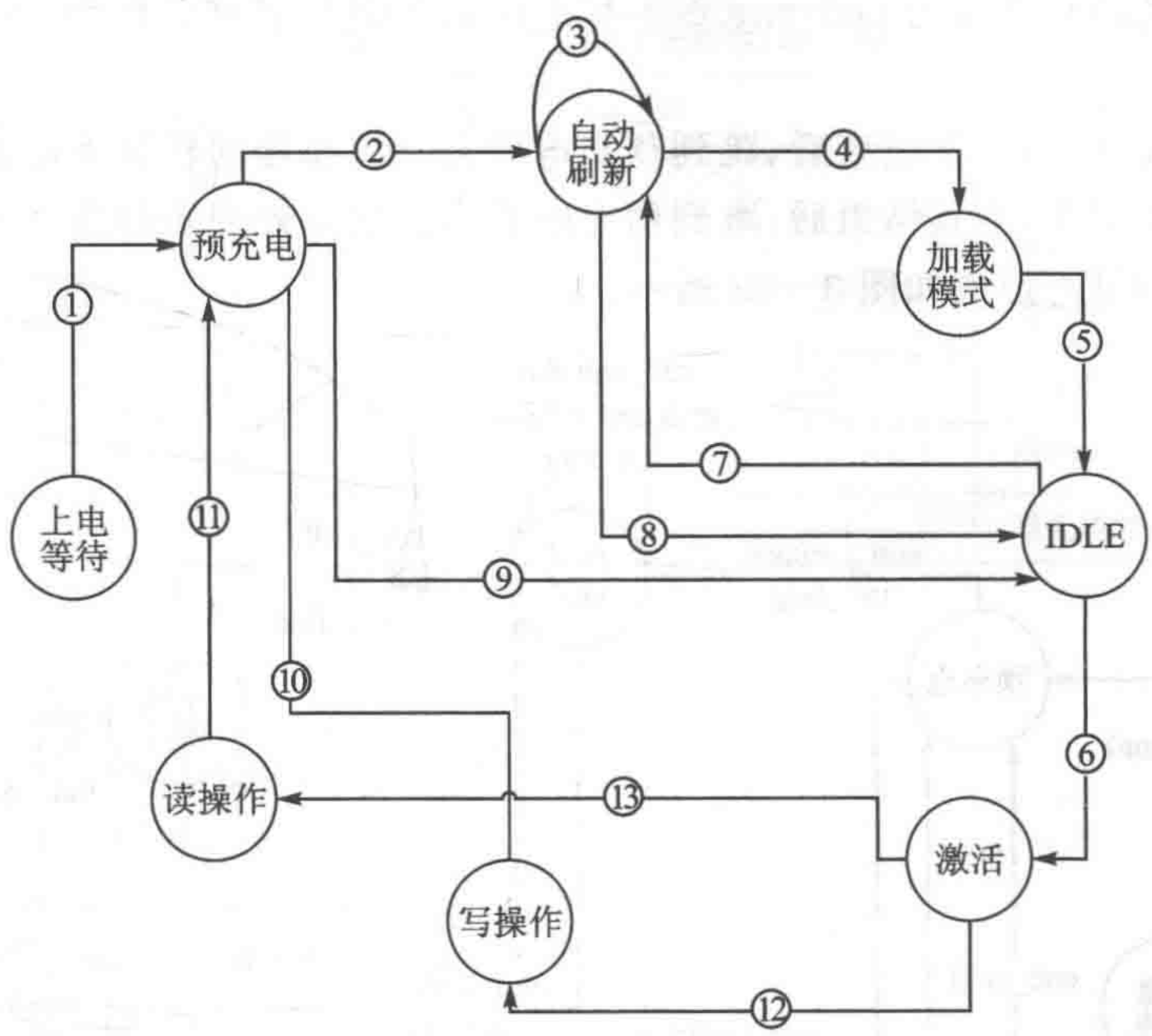


图3-31 状态转移图

(5) 转移条件

图3-31中，在上电等待状态，只要计数结束就跳转到预充电状态，不存在其他情况。故①处的转移条件为end_cnt0。

在预充电状态，如果是在初始化操作期间，则跳到自动刷新；如果是读操作或写操作，则跳到IDLE状态。我们可以通过增加init_flag信号来区分，当其为1时是在初始化中。故②处的转移条件为end_cnt0&&init_flag；⑨处的转移条件为end_cnt0&&!init_flag。

在自动刷新状态，如果是初始化操作，就要刷新两次，然后跳到加载模式；如果不

是初始化操作，则其结束后跳到 IDLE 状态。由于在初始化操作中要刷新两次，所以可引入刷新阶段标志信号 resh_cnt，在 init_flag 有效时，resh_cnt 为 0 表示在自动刷新 0 阶段，为 1 表示在自动刷新 1 阶段。故③处的转移条件为 end_cnt0&&resh_cnt==0&&init_flag；④处的转移条件为 end_cnt0&&resh_cnt==1&&init_flag。在非初始化操作时，刷新完成跳到 IDLE 状态，故⑧处的转移条件为 end_cnt0&&!init_flag。

在加载模式，只要计数结束就跳到 IDLE 状态，故⑤处的转移条件为 end_cnt0。

在 IDLE 状态，有两种情况。当自动刷新时间到时，就跳到自动刷新；否则，当读或写请求有效时，要跳到激活状态。故⑦处的转移条件为 resh_flag；⑥处的转移条件为!resh_flag&&(rd_req||wr_req)。

在激活状态，有两种情况：读或者写。可用 rd_flag 来区分读和写，其为 1 表示读。故⑫处的转移条件为 end_cnt0&&!rd_flag；⑬处的转移条件为 end_cnt0&&rd_flag。

在写操作状态，当其结束后，跳到预充电状态。故⑩处的转移条件为 end_cnt0。

在读操作状态，当其结束后，跳到预充电状态。故⑪处的转移条件为 end_cnt0。

状态图的最终形式如图 3-32 所示。

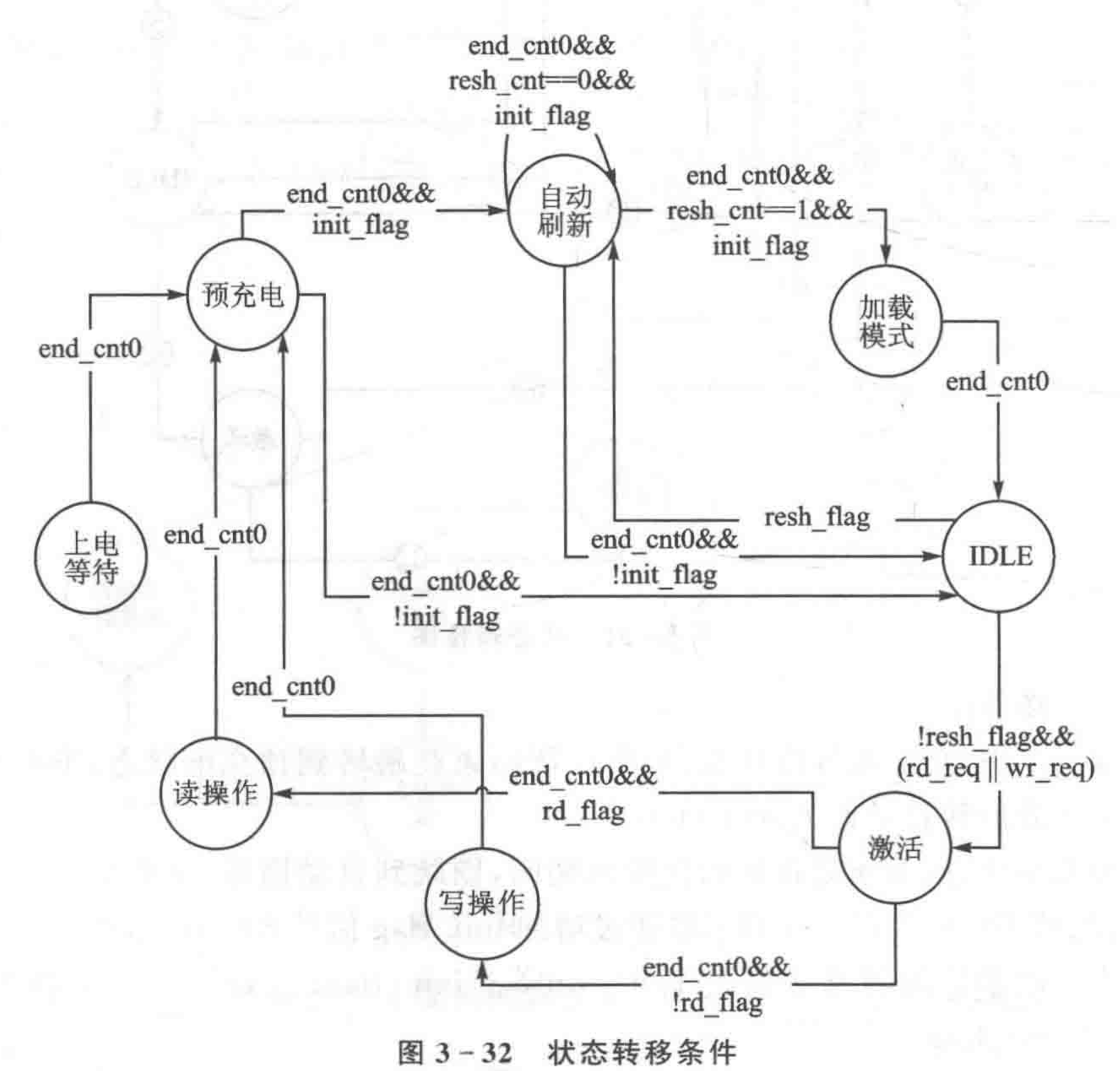


图 3-32　状态转移条件

(6) 完整性检查

从上述状态跳转情况知，还需要增加两个计数器：第一个计数器(cnt0)用来计各个状态(IDLE状态除外)的持续时间，第二个计数器(cnt1)用来计初始化完成后的循环刷新时间间隔。而对于计数器的设计，相信读者经过前面章节的学习已经非常熟悉了，在此直接给出设计结果。

① cnt0

初始值为0；

加1条件：当前状态不为IDLE；

结束值：计数到 $x-1$。

x 的值见表3-9。

表3-9 计数器 x 的值

状 态	参数名	时钟周期数(即 x 的值)	说 明
INIT_NOP	TIME_100US	10 000	上电等待时间
REFRESH	TRC	6	刷新持续时间
MODE	TMRD	2	加载持续时间
ACTIVE	TRCD	2	激活持续时间
WRITE	TWRITE	256	写操作持续时间
READ	TREAD	256	读操作持续时间
PRECHARGE	TRP	2	预充电持续时间

② cnt1

cnt1 初始值为0；

cnt1 加1条件：非初始状态；

cnt1 结束值：计数到 1 562－1。

③ 自动刷新请求信号 resh_flag

resh_flag 的初值为0；

resh_flag 由0变1：cnt1 计数到 1 562－1；

resh_flag 由1变0：自动刷新开始后；

其他情况保持不变。

④ 初始化标志信号 init_flag

init_flag 初始化期间为1；

init_flag 非初始化期间为0。

⑤ 初始化过程中自动刷新计数 resh_cnt

resh_cnt 的初值为0；

resh_cnt 自动刷新1状态期间为1；

resh_cnt 加载状态期间为0；

其他状态保持不变。

⑥ rd_flag 读标志信号

rd_flag=1 允许进入读状态;rd_flag=0 允许进入写状态。当同时出现读、写请求时,如果上一次执行了读操作,则此时执行写操作;如果上一次执行了写操作,则此时执行读操作。如果没有同时出现读、写请求,则有什么请求就执行什么操作。

⑦ 发送命令 command

command 的初值为 C_NOP;

在初始化预充电、写操作预充电及读预充电期间:C_PRECHARGE;

在初始化自动刷新时间 0、初始化自动刷新时间 1 以及自动刷新期间:C_AUTO;

在初始化加载模式期间:C_LOAD;

写激活状态及读激活状态:C_ACTIVE;

读状态:C_READ;

写状态:C_WRITE;

其他情况:C_NOP。

⑧ 地址 addr_o

在预充电期间值为 12'b1 ≪ 10;

在初始化加载模式期间:CODE;

在写激活状态及读激活状态:addr_o 为 addr_i [19:8];

在写状态及读状态:addr_o 为 addr_i [7:0];

其他情况下,值为 0。

⑨ bank

bank 的值为 addr_i[21:20],用于选择第几个 bank。

(7) 状态机代码

第一段,用同步时序,将次态的值赋给现态。

因状态机及其他代码,会用到大量参数,故在设计前,先进行相关参数的定义。

```
1    //****************位宽参数定义****************
2    parameter    DATA_W =        16  ;
3    parameter    ADDR_W =        22  ;
4
5    //****************状态转移参数定义****************
6    parameter    INIT_NOP        =    4'b0001 ;
7    parameter    PRECHARGE       =    4'b0010 ;
8    parameter    REFRESH         =    4'b0011 ;
9    parameter    MODE            =    4'b0100 ;
10   parameter    IDLE            =    4'b0101 ;
11   parameter    ACTIVE          =    4'b0110 ;
12   parameter    WRITE           =    4'b0111 ;
```

```
13      parameter    READ              =     4'b1000 ;
14
15      //**************命令参数定义**************
16      parameter    NOP_CMD           = 4'b0111;
17      parameter    PRECHARGE_CMD     = 4'b0010;
18      parameter    REFRESH_CMD       = 4'b0001;
19      parameter    MODE_CMD          = 4'b0000;
20      parameter    ACTIVE_CMD        = 4'b0011;
21      parameter    WRITE_CMD         = 4'b0100;
22      parameter    READ_CMD          = 4'b0101;
23      parameter    CMD_W             = 4     ;
24     /**************时间参数定义**************/
25      parameter    T_100US           =     10000;
26      parameter    TRP               =     3    ;
27      parameter    TRC               =     7    ;
28      parameter    TMRD              =     2    ;
29      parameter    TRCD              =     3    ;
30      parameter    T_256             =     256  ;
31      parameter    MODE_VALUE        = 12'b000000110111;
32      parameter    ALL_BANK          = 12'b001000000000;
33
34 //第一段
35 always@(posedge clk or negedge rst_n)begin
36      if(rst_n == 1'b0)begin
37          state_c <= INIT_NOP;
38      end
39      else begin
40          state_c <= state_n;
31      end
42 end
```

第二段，用组合逻辑描述状态转移条件。注意，转移条件用信号来表示，信号名要按明德扬规则来命名。

```
1  //第二段
2  always  @( * )begin
3      case(state_c)
4          INIT_NOP:begin
5              if(ini2pre_start)begin
6                  state_n = PRECHARGE;
7              end
8              else begin
```

```
9                   state_n = state_c;
10              end
11          end
12          PRECHARGE:begin
13              if(pre2ref_start)begin
14                  state_n = REFRESH;
15              end
16              else if(pre2idl_start)begin
17                  state_n = IDLE;
18              end
19              else begin
20                  state_n = state_c;
21              end
22          end
23          REFRESH:begin
24              if(ref2ref_start)begin
25                  state_n = REFRESH;
26              end
27              else if(ref2mod_start)begin
28                  state_n = MODE;
29              end
30              else if(ref2idl_start)begin
31                  state_n = IDLE;
32              end
33              else begin
34                  state_n = state_c;
35              end
36          end
37          MODE:begin
38              if(mod2idl_start)begin
39                  state_n = IDLE;
40              end
41              else begin
42                  state_n = state_c;
43              end
44          end
45          IDLE:begin
46              if(idl2ref_start)begin
47                  state_n = REFRESH;
48              end
49              else if(idl2act_start)begin
50                  state_n = ACTIVE;
```

```
51                end
52                else begin
53                    state_n = state_c;
54                end
55            end
56            ACTIVE:begin
57                if(act2wrt_start)begin
58                    state_n = WRITE;
59                end
60                else if(act2red_start)begin
61                    state_n = READ;
62                end
63                else begin
64                    state_n = state_c;
65                end
66            end
67            WRITE:begin
68                if(wrt2pre_start)begin
69                    state_n = PRECHARGE;
70                end
71                else begin
72                    state_n = state_c;
73                end
74            end
75            READ:begin
76                if(red2pre_start)begin
77                    state_n = PRECHARGE;
78                end
79                else begin
80                    state_n = state_c;
81                end
82            end
83            default:begin
84                state_n = INIT_NOP;
85            end
86        endcase
87 end
```

第三段，用 assign 定义转移条件。注意，条件一定要加上现态。

```
1 //状态转移条件定义
2 assign ini2pre_start = state_c == INIT_NOP && end_cnt;
3 assign wrt2pre_start = state_c == WRITE && end_cnt;
```

```
assign red2pre_start = state_c == READ && end_cnt;
assign pre2ref_start = state_c == PRECHARGE && end_cnt && init_flag;
assign pre2idl_start = state_c == PRECHARGE && end_cnt && init_flag == 0;
assign ref2ref_start  = state_c == REFRESH && end_cnt && resh_cnt == 0 && init_flag;
assign ref2mod_start  = state_c == REFRESH && end_cnt && resh_cnt == 1 && init_flag;
assign ref2idl_start = state_c == REFRESH && end_cnt && init_flag == 0;
assign mod2idl_start = state_c == MODE && end_cnt;
assign idl2act_start = state_c == IDLE &&resh_flag == 0 && (rd_req || wr_req);
assign idl2ref_start = state_c == IDLE && (resh_flag || end_cnt1);
assign act2wrt_start = state_c == ACTIVE && end_cnt && rd_flag == 0;
assign act2red_start = state_c == ACTIVE && end_cnt && rd_flag == 1;
```

第四段，是输出信号设计，见功能代码部分。

(8) 功能代码

```
/***************时间参数定义***************/
    parameter    T_100US          =    10000;
    parameter    TRP              =    3    ;
    parameter    TRC              =    7    ;
    parameter    TMRD             =    2    ;
    parameter    TRCD             =    3    ;
    parameter    T_256            =    256  ;
    parameter    MODE_VALUE       = 12'b000000110111;
    parameter    ALL_BANK         = 12'b001000000000;

//根据第六步第 1 点，写出 cnt 的代码
always  @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        cnt <= 0;
    end
    else if(add_cnt)begin
        if(end_cnt)begin
            cnt <= 0;
        end
        else begin
            cnt <= cnt + 1'b1;
        end
    end
end

assign add_cnt = state_c != IDLE;
assign end_cnt = add_cnt && cnt == x - 1;
```

```
28
29  always  @( * )begin
30      if(state_c == INIT_NOP)begin
31          x = T_100US;
32      end
33      else if(state_c == PRECHARGE)begin
34          x = TRP;
35      end
36      else if(state_c == REFRESH)begin
37          x = TRC;
38      end
39      else if(state_c == MODE)begin
40          x = TMRD;
41      end
42      else if(state_c == ACTIVE)begin
43          x = TRCD;
44      end
45      else if(state_c == WRITE || state_c == READ)begin
46          x = T_256;
47      end
48      else begin
49          x = 0;
50      end
51  end
52
53  //根据第六步第 2 点,写出 cnt1 的代码
54  always  @(posedge clk or negedge rst_n)begin
55      if(rst_n == 1'b0)begin
56          cnt1 <= 0;
57      end
58      else if(add_cnt1)begin
69          if(end_cnt1)begin
60              cnt1 <= 0;
61          end
62          else begin
63              cnt1 <= cnt1 + 1'b1;
64          end
65      end
66  end
67
68  assign add_cnt1 = init_flag == 1'b0;
69  assign end_cnt1 = add_cnt1 && cnt1 == 1562 - 1;
```

```
70
71  //根据第六步第 3 点,写出 resh_flag 的代码
72  /*****************状态跳转指示信号*****************/
73  always  @(posedge clk or negedge rst_n)begin
74      if(rst_n == 1'b0)begin
75          resh_flag <= 1'b0;
76      end
77      else if(end_cnt1) begin
78          resh_flag <= 1'b1;
79      end
80      else if(idl2ref_start)begin
81          resh_flag <= 1'b0;
82      end
83  end
84
85  //根据第六步第 4 点,写出 init_flag 的代码
86  always  @(posedge clk or negedge rst_n)begin
87      if(rst_n == 1'b0)begin
88          init_flag <= 1;
89      end
90      else if(mod2idl_start)begin
91          init_flag <= 0;
92      end
93  end
94
95  //根据第六步第 5 点,写出 resh_cnt 的代码
96  always  @(posedge clk or negedge rst_n)begin
97      if(rst_n == 1'b0)begin
98          resh_cnt <= 0;
99      end
100     else if(add_resh_cnt)begin
101         if(end_resh_cnt)begin
102             resh_cnt <= 0;
103         end
104         else begin
105             resh_cnt <= resh_cnt + 1'b1;
106         end
107     end
108 end
109
110 assign add_resh_cnt = init_flag && state_c == REFRESH && end_cnt;
111 assign end_resh_cnt = add_resh_cnt && resh_cnt == 2-1;
```

```
112
113 //根据第六步第6点,写出rd_flag的代码
114 /*在读/写状态时的precharge状态下,将rd_flag延迟一拍给rd_hty,指示当前响应的是
115 读,还是写,rd_hty保存了rd_flag的值,给下次同时来读写信号判断用*/
116 /*在active状态下,当收到读请求时,若要响应读有效,则应满足以下两个条件:1.上次
117 响应了读,且此时写无效;2.上次响应了写;
118 在active状态下,当收到写请求时,若要响应写有效,则应满足以下两个条件:1.上次响
119 响应了写,且此时读无效;2.上次响应了读*/
120
121 always  @(posedge clk or negedge rst_n)begin
122     if(rst_n == 1'b0)begin
123         rd_hty <= 1'b0;
124     end
125     else if(pre2idl_start) begin
126         rd_hty <= rd_flag;
127     end
128 end
129
130 assignwrite_sel = idl2act_start && ((rd_hty == 1'b0 && rd_req == 1'b0) ||rd_hty) && wr_req;
131 assign read_sel = idl2act_start && ((rd_hty && wr_req == 1'b0) ||rd_hty == 1'b0) && rd_req;
132
133 always  @(posedge clk or negedge rst_n)begin
134     if(rst_n == 1'b0)begin
135         rd_flag <= 1'b0;
136     end
137     else if(write_sel) begin
138         rd_flag <= 1'b0;
139     end
140     else if(read_sel)begin
141         rd_flag <= 1'b1;
142     end
143 end
144
145
146 /*****************地址信号*****************/
147 //地址信号暂存寄存器addr_i_ff0
148 always  @(posedge clk or negedge rst_n)begin
149     if(rst_n == 1'b0)begin
150         addr_i_ff0 <= 0;
151     end
152     else begin
153         addr_i_ff0 <= addr_i;
```

```
154         end
155     end
156
157     //根据第六步第 9 点,写出 bank 的代码
158     always  @(posedge clk or negedge rst_n)begin
159         if(rst_n == 1'b0)begin
160             bank <= 2'b00;
161         end
162         else if(idl2act_start)begin
163             bank <= addr_i[21:20];
164         end
165         else if(act2wrt_start || act2red_start)begin
166             bank <= addr_i_ff0[21:20];
167         end
168         else begin
169             bank <= 2'b00;
170         end
171     end
172
173     //根据第六步第 8 点,写出 addr_o 的代码
174     always  @(posedge clk or negedge rst_n)begin
175         if(rst_n == 1'b0)begin
176             addr_o <= 0;
177         end
178         else if(ref2mod_start)begin
179             addr_o <= MODE_VALUE;
180         end
181         else if(ini2pre_start || wrt2pre_start || red2pre_start)begin
182             addr_o <= ALL_BANK  ;
183         end
184         else if(idl2act_start)begin
185             addr_o <= addr_i[19:8];
186         end
187         else if(act2wrt_start || act2red_start)begin
188             addr_o <= {addr_i_ff0[7:0]};
189         end
190         else begin
191             addr_o <= 0;
192         end
193     end
194
195     /**************读、写相关信号**************/
```

```
196 //读应答信号 rd_ack
197 always  @( * )begin
198     rd_ack = read_sel;
199 end
200
201 //读出的数据信号 rdata/数据有效信号 rd_vld
202 assign read_flag = state_c == READ;
203
204 always  @(posedge clk or negedge rst_n)begin
205     if(rst_n == 1'b0)begin
206         read_flag_ff0 <= 1'b0       ;
207         read_flag_ff1 <= 1'b0       ;
208         read_flag_ff2 <= 1'b0       ;
209     end
210     else begin
211         read_flag_ff0 <= read_flag    ;
212         read_flag_ff1 <= read_flag_ff0;
213         read_flag_ff2 <= read_flag_ff1;
214     end
215 end
216
217
218 always  @(posedge clk or negedge rst_n)begin
219     if(rst_n == 1'b0)begin
220         rdata <= 0;
221     end
222     else if(read_flag_ff2)begin
223         rdata <= dq;
224     end
225 end
226
227 always  @(posedge clk or negedge rst_n)begin
228     if(rst_n == 1'b0)begin
229         rdata_vld <= 0;
230     end
231     else if(read_flag_ff2)begin
232         rdata_vld <= 1;
233     end
234     else if(pre2idl_start && rd_flag)begin
235         rdata_vld <= 0;
236     end
237 end
238
```

```
239 //写应答信号 wr_ack
240 always  @( * )begin
241     wr_ack = write_sel;
242 end
243
244 / * 根据题目 CAS Latency 为 3,加上进入 active 状态,写数据 wdata 共需要延时 4 个时
245  钟周期 * /
246 always  @(posedge clk or negedge rst_n)begin
247     if(rst_n == 1'b0)begin
248         wdata_ff0 <= 0;
249         wdata_ff1 <= 0;
250         wdata_ff2 <= 0;
251         wdata_ff3 <= 0;
252     end
253     else begin
254         wdata_ff0 <= wdata;
255         wdata_ff1 <= wdata_ff0;
256         wdata_ff2 <= wdata_ff1;
257         wdata_ff3 <= wdata_ff2;
258     end
259 end
260
261 //三态门使能
262 assign dq_en = state_c == WRITE
263
264 //三态门数据输出
265 assign dq     = dq_en? wdata_ff3:16'hzzzz
266
267
268 //有效使能信号 cke,上电开始工作后一直为 1
269 always  @(posedge clk or negedge rst_n)begin
270     if(rst_n == 1'b0)begin
271         cke <= 1;
272     end
273     else begin
274         cke <= 1;
275     end
276 end
277
278 //根据第六步第 7 点,写出 cmd 的代码
279 //发送命令 cmd
280 always  @(posedge clk or negedge rst_n)begin
281     if(rst_n == 1'b0)begin
282         cmd <= NOP_CMD;
```

```
283         end
284         else if(ini2pre_start || wrt2pre_start || red2pre_start)begin
285             cmd <= PRECHARGE_CMD;
286         end
287         else if(pre2ref_start || ref2ref_start || idl2ref_start)begin
288             cmd <= REFRESH_CMD;
289         end
290         else if(ref2mod_start)begin
291             cmd <= MODE_CMD;
292         end
293         else if(act2wrt_start)begin
294             cmd <= WRITE_CMD;
295         end
296         else if(act2red_start)begin
297             cmd <= READ_CMD;
298         end
299         else if(idl2act_start)begin
300             cmd <= ACTIVE_CMD;
301         end
302         else begin
303             cmd <= NOP_CMD;
304         end
305     end
306
307     //cs,ras,cas,we 命令指示信号输出
308     always  @( * )begin
309         {cs,ras,cas,we} = cmd;
310     end
311
312     //dqm 信号,复位时为高,其他为低
313     always  @(posedge clk or negedge rst_n)begin
314         if(rst_n == 1'b0)begin
315             dqm <= 2'b00;
316         end
317         else if(dqm_en)begin
318             dqm <= 2'b11;
319         end
320         else begin
321             dqm <= 2'b00;
322         end
323     end
324     assign dqm_en = state_c == init_flag ;
```

第4章

FIFO 架构

FIFO(First Input First Output),即先入先出队列。在计算机中,先入先出队列是一种传统的按序执行方法,先进入的指令先完成并引退,跟着才执行第二条指令(指令就是计算机在响应用户操作的程序代码,对用户是透明的)。

在数字电路设计中所说的 FIFO 实际是指 FIFO 存储器,主要用于数据缓存和异步处理,当然 FIFO 存储器缓存数据也遵循先入先出的原则。由于微电子技术的飞速发展,新一代 FIFO 芯片容量越来越大,体积越来越小,价格越来越便宜。作为一种新型大规模集成电路,FIFO 芯片以其灵活、方便、高效的特性,在高速数据采集、高速数据处理、高速数据传输及多机处理系统中得到越来越广泛的应用。

FIFO 本质上是一个 RAM,它与普通存储器的区别是没有外部读/写地址线,这样使用起来非常简单,但缺点是只能顺序写入数据,顺序读出数据,其数据地址由内部读/写指针自动加 1 完成,不能像普通存储器那样可以由地址线决定读取或写入某个指定的地址。

在实际项目中,大部分数据都是先进先出的,因此非常适合选用 FIFO 来实现。熟练掌握 FIFO 使用规则和技巧则是非常重要的"基本功"。

4.1 FIFO 架构八步法

4.1.1 概 述

1. FIFO 规则

FIFO 规则 1:使用 Show-ahead 读模式。

根据 FIFO 的读模式,一般有两种使用模式:Show-ahead 和 Normal 模式。这两种模式的区别将在生成 FIFO IP 核的步骤中说明。

FIFO 规则 2:读、写隔离规则。

FIFO 读、写隔离规则是明德扬经过长期的实践总结出来的经验规则。

读、写隔离规则是指:读控制和写控制是独立的,它们之间除了用 FIFO 交流信息外,不能有任何信息传递。因此,既不能根据 FIFO 的读状态或者读出的数据来决

定写行为，也不能根据 FIFO 的写状态和写入的数据来决定读行为，如图 4－1 所示。

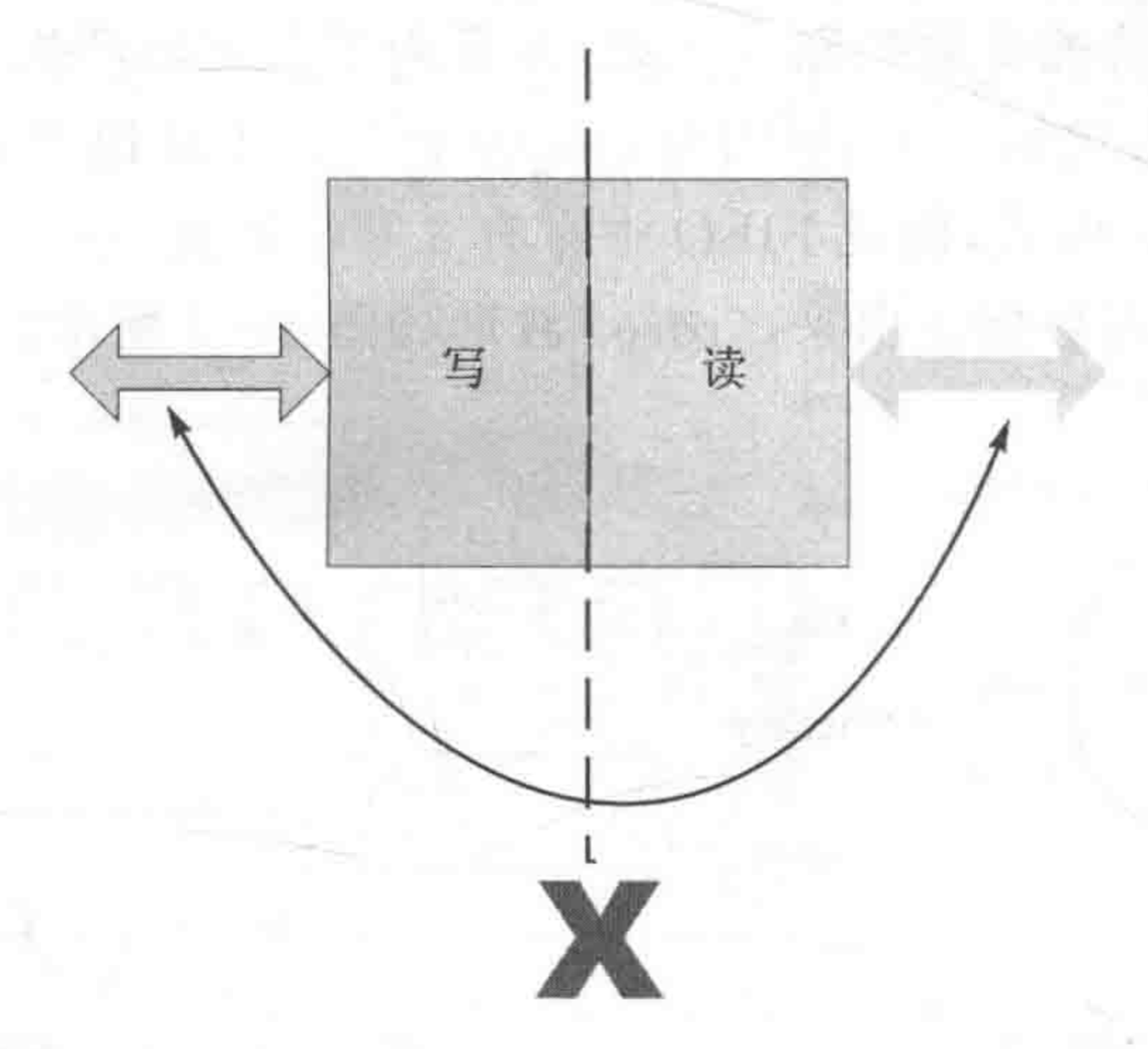


图 4－1　读、写隔离规则图

下面通过一个例子来帮助读者理解读、写隔离规则。

设计一个模块，该模块将输入的报文保存到 FIFO 中，当保存完一个完整的报文后，再将其读出，并传送给下游模块。

什么时候保存完一个完整的报文呢？就是收到 din_eop 的时候。那么是否可以使用 din_eop 作为读 FIFO 的开始条件呢？答案是不允许。

假设以 din_eop 作为读开始的条件，如图 4－2 所示，先后输入一个 6 字节和 3 字节报文。当收到第 1 个报文的 din_eop 时，开始读 FIFO 数据并输出。由于输入是 6 个数据，输出也是 6 个数据，那么就需 6 个时钟来处理。而在第 2 个报文的 din_eop 有效时，按道理应该读取第 2 个报文并输出，但此时由于对第 1 个报文还没有处理完，因此根本不能处理第 2 个报文，此时第 2 个报文将被“丢弃”，数据出错。

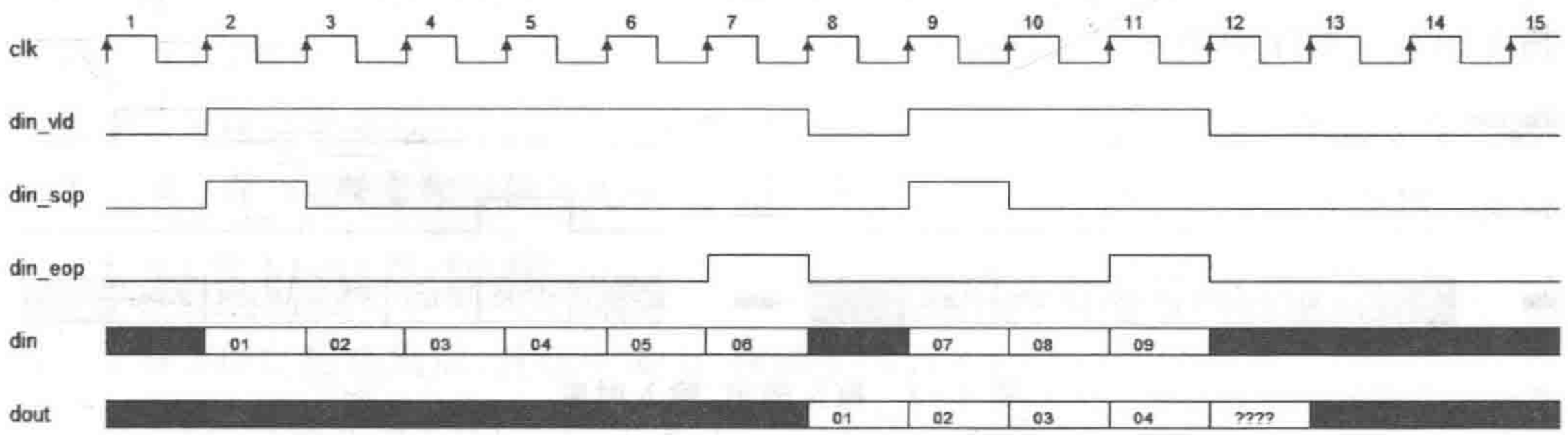


图 4－2　连续收到两个报文

所以使用 din_eop 来作为读开始的条件是不允许的，其根本原因是，数据是要缓存后输出的，同样的，din_eop 这个信号也是要缓存的。

基于这个原因，我们提出用双 FIFO 架构的概念，如图 4－3 所示。

在 4.1.2 小节中有双 FIFO 架构的具体应用介绍。

FIFO 规则 3：读使能必须判断空状态，并且用组合逻辑产生。

rdreq 必须由组合逻辑产生，原因与 empty 有关。下面说明 rdreq 与 empty 的关系。使用 Show-ahead 模式，假设 FIFO 中存有 3 字节数据，现对 FIFO 进行读操作，要把 FIFO 中存的所有数据读出来。rdreq 波形如图 4－4 所示。

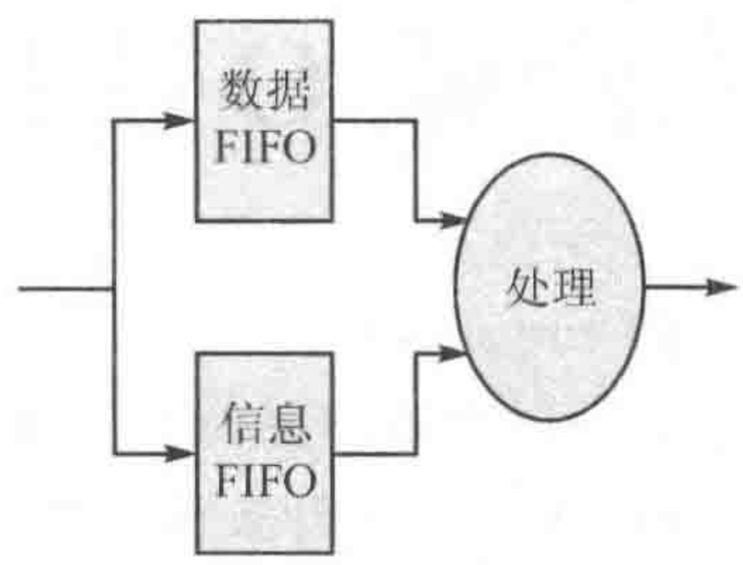


图 4－3　双 FIFO 架构

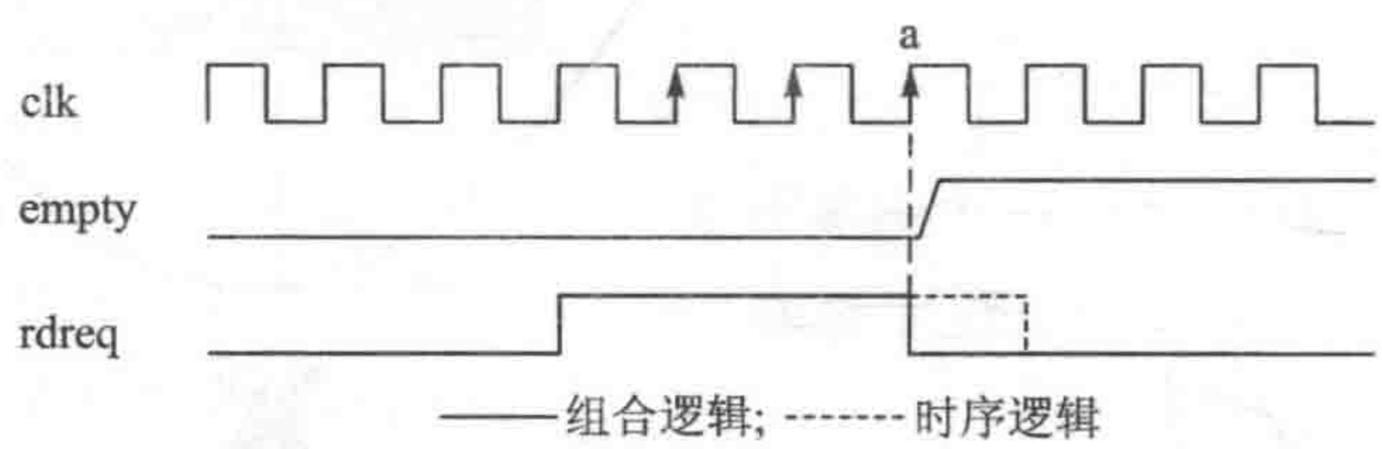


图 4－4　rdreq 波形图

从图中可以看出，由于电路存在一定延时，在 a 时刻 FIFO 已经为空，empty 变为高电平需要经历一段时间。

如果用时序逻辑产生 rdreq，在 a 时刻 empty 为 0，则表示 FIFO 中还有数据（实际上已经为空），因此 rdreq 还要保持一个时钟周期。在 FIFO 为空的情况下要再读取一个数据，读操作会出错。如果用组合逻辑产生 rdreq，当 empty 为 1 时，rdreq 马上拉低，如图 4－4 中的实线部分波形，就不会出现读取空 FIFO 的错误。

FIFO 规则 4：处理报文时，把指示信号与数据一起存入 FIFO。

通过一个例子来说明此规则。

现要求设计一个模块，先将输入报文存储到内部 FIFO 中，等待适当时机，再将报文数据原封不动地传送给下游模块。注意，要同时输出报文数据、报文头和报文尾指示信号。时序如图 4－5 所示。

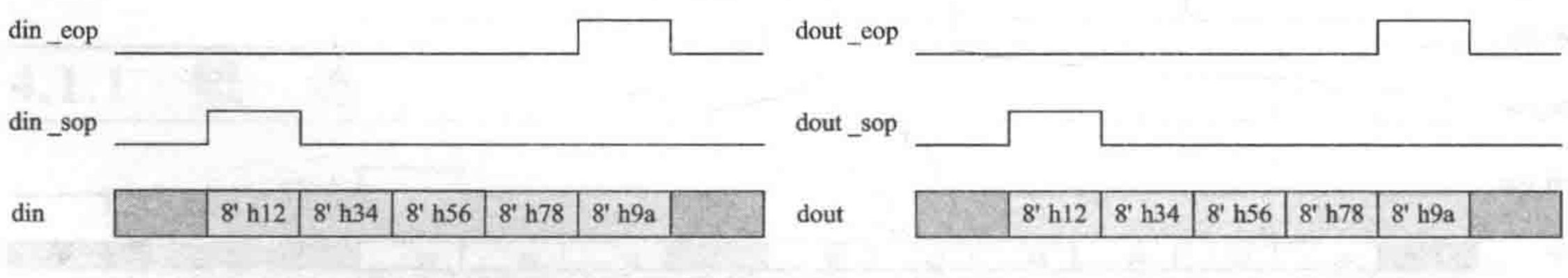


图 4－5　报文输出、输入时序

例如输入的报文 din 是 8'h12、8'h34、8'h56、8'h78 和 8'h9a，其中 8'h12 是报文头数据，8'h9a 是报文尾数据。输出的报文 dout 同样是 8'h12、8'h34、8'h56、8'h78 和 8'h9a，同样也要指出 8'h12 是报文头数据，8'h9a 是报文尾数据。

根据一般思路，可以生成一个 8 位宽度的 FIFO，然后将 din 数据保存到 FIFO 中。读出 FIFO 的数据赋给 dout。但是怎么产生 dout_sop 和 dout_eop 呢？

其实，FIFO不仅能保存“数据”，也能保存“指示信号”，因此可以将数据和对应的“指示信号值”一起写入FIFO。如生成一个10位宽度的FIFO，保存到FIFO的数据是{din，din_sop，din_eop}。其示意图如图4-6所示。

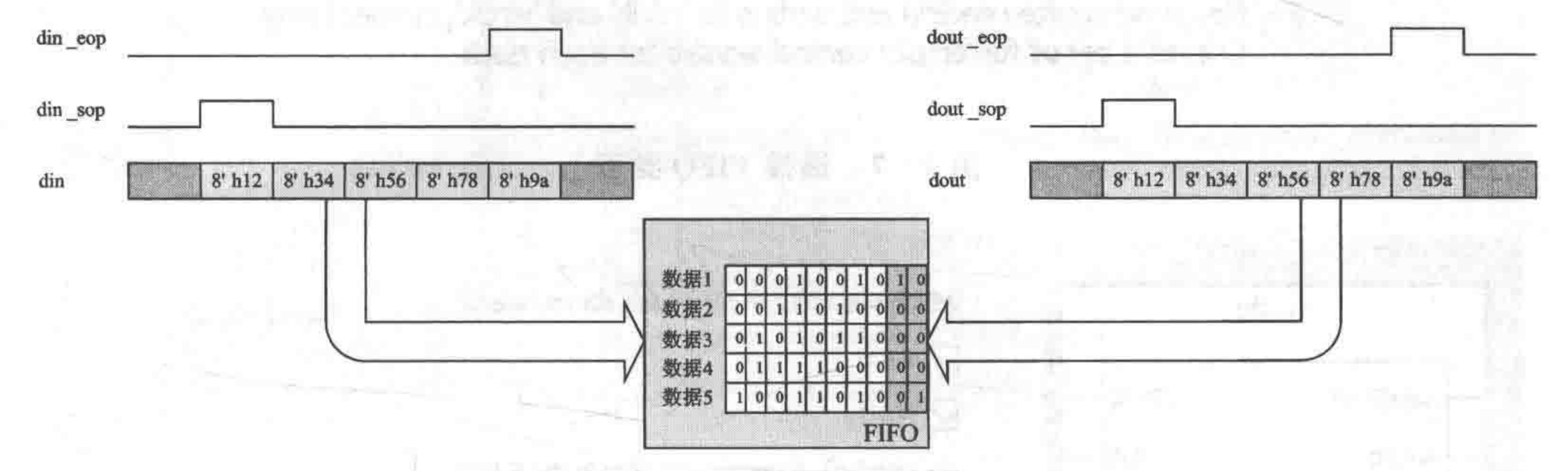


图4-6 FIFO作为报文传输缓存

在FIFO中，保存的第1个数据是10'b0001001010，也就是第1个数8'h12及此时din_sop(值为1)和din_eop(值为0)。第2个数据是10'b0011010000，也就是第2个数8'h34及此时din_sop(值为0)和din_eop(值为0)。依此类推，最后一个数据是10'b1001101001，也就是8'h9a及此时din_sop(值为0)和din_eop(值为1)。

通过这种方法保存后，就很容易产生dout_sop和dout_eop信号了，从FIFO中读到的数据，如果最低位为1，则表示报文最后一个数据，此时dout_eop为1；如果次低位为1，则表示报文第一个数据，此时dout_sop为1。

通过此种方式，还可以判断报文的开始和结束，从而用于其他判断。

FIFO规则5：读、写时钟不同时，必须用异步FIFO。

FIFO按时钟可分为异步FIFO和同步FIFO。

同步FIFO是指读时钟和写时钟都相同的FIFO。同步FIFO内部没有异步处理，因此结构简单，资源占用较少。

异步FIFO是指读时钟和写时钟可以不同的FIFO。异步FIFO内部有专门的异步处理电路，处理读和写信号的交互，因为异步FIFO结构复杂，占用资源较大。

2. 生成FIFO的IP核

了解FIFO的规则后，具体生成IP核的步骤如下：

第一步，选择同步FIFO(上)或异步FIFO(下)，见图4-7。

第二步，根据需求选择所需要的信号，同步FIFO见图4-8和表4-1，异步FIFO见图4-9与表4-2。

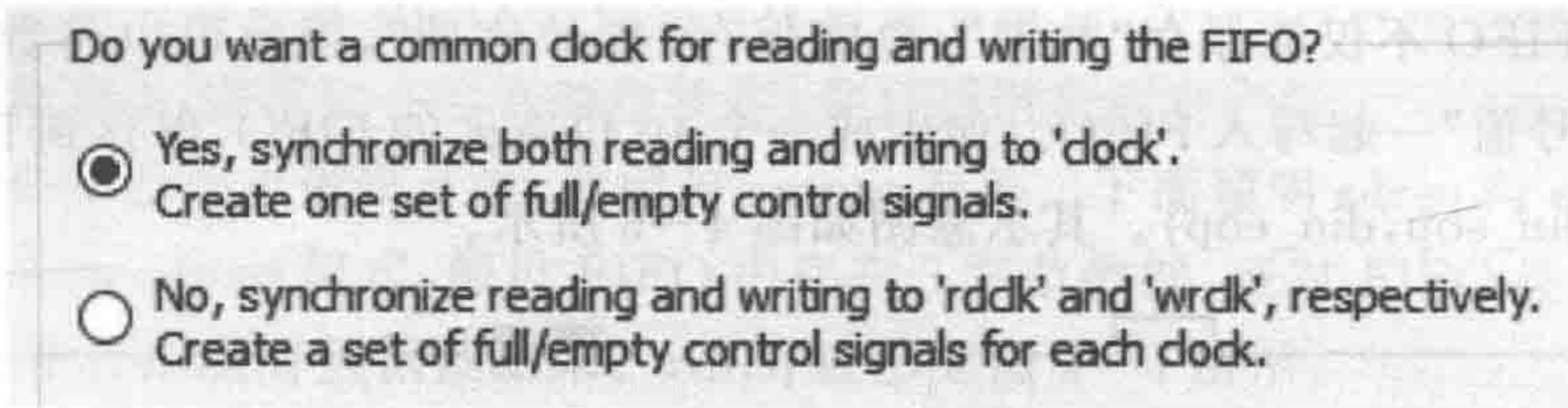


图 4-7　选择 FIFO 类型

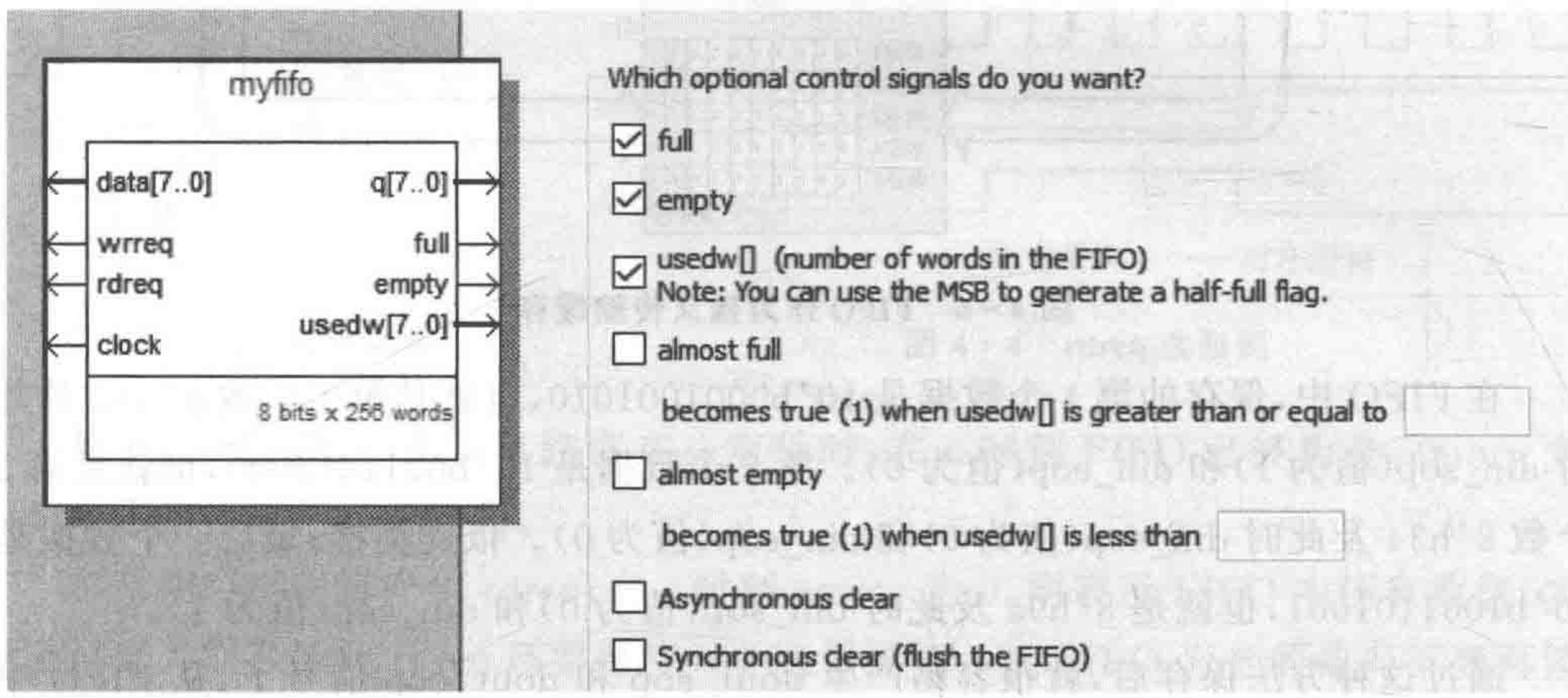


图 4-8　同步 FIFO 信号选择

表 4-1　同步 FIFO 信号列表

信　号	信号说明
clk	FIFO 的工作时钟
data	必选。写到 FIFO 的数据，如果 wrreq=0，data 不起作用
wrreq	必选。写使能信号，当其为 1 时表示将 data 的值写到 FIFO
rdreq	必选。读使能信号，表示从 FIFO 中读一个数据
q	必选。从 FIFO 里读到的数据
empty	可选，通常都会选。FIFO 状态指示信号，当其为 1 时表示 FIFO 处于空状态，此时绝对不能再读数据
full	可选，通常都会选。FIFO 状态指示信号，当其为 1 时表示 FIFO 处于满状态，此时绝对不能再写数据
usedw	可选。FIFO 状态指示信号，指示现在 FIFO 保存有多少个数据
almost_full	可选。FIFO 状态指示信号，当 FIFO 的数量大于指定的个数时，此信号为 1
almost_empty	可选。FIFO 状态指示信号，当 FIFO 的数量小于指定的个数时，此信号为 1
aclr	可选，异步复位信号。当其为 1 时，复位 FIFO
sclr	可选，同步复位信号。当其为 1 时，复位 FIFO

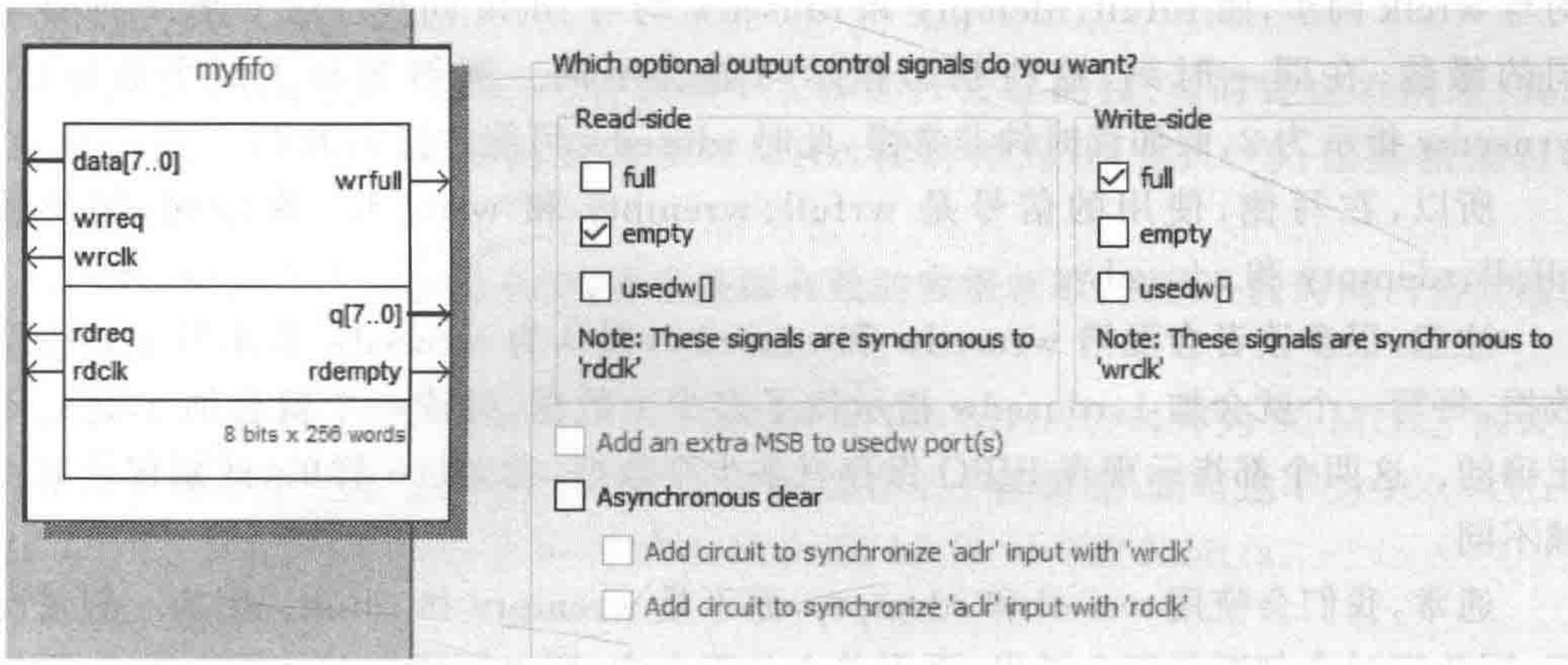


图 4-9　异步 FIFO 信号选择

表 4-2　异步 FIFO 信号列表

信　号	信号说明
rdclk	FIFO 读侧的工作时钟，即读时钟
wrclk	FIFO 写侧的工作时钟，即写时钟
data	写到 FIFO 的数据，如果 wrreq=0，data 不起作用。写时钟域信号
wrreq	必选。写使能信号，当其为 1 时表示将 data 的值写到 FIFO。写时钟域信号
rdreq	必选。读使能信号，表示从 FIFO 中读一个数据。读时钟域信号
q	必选。从 FIFO 里读到的数据。读时钟域信号
wrempty	可选，通常都不会选。FIFO 状态指示信号，当其为 1 时表示 FIFO 处于空状态，此时绝对不能再读数据。写时钟域信号
wrfull	可选，通常都会选。FIFO 状态指示信号，当其为 1 时表示 FIFO 处于满状态，此时绝对不可再写数据。写时钟域信号
wrusedw	可选。FIFO 状态指示信号，指示现在 FIFO 保存有多少个数据。写时钟域信号
rdfull	可选，通常都不会选。FIFO 状态指示信号，当其为 1 时表示 FIFO 处于满状态，此时绝对不可再写数据。读时钟域信号
rdempty	可选，通常都会选。FIFO 状态指示信号，当其为 1 时表示 FIFO 处于空状态，此时绝对不能再读数据。读时钟域信号
rdusedw	可选。FIFO 状态指示信号，指示现在 FIFO 保存有多少个数据。读时钟域信号
aclr	可选，异步复位信号。当其为 1 时复位 FIFO

从异步 FIFO 可以看到，读时钟域和写时钟域都有空、满、保存个数等状态指示信号。这两侧信号有什么不同呢？这些信号都指示了 FIFO 的空、满、保存个数等状态，定义都是相同的，但这些信号同步的时钟不相同。wrfull、wrempty 和 wrusedw

均与 wrclk 同步，而 rdfull、rdempty 和 rdusedw 均与 rdclk 同步。由于读、写时钟不同的缘故，在同一时刻，这些状态指示可能会不同。例如当写了两个数据后，wrusedw 指示为 2，假如读时钟非常慢，此时 rdusedw 可能才指示为 1。

所以，在写侧，使用的信号是 wrfull、wrempty 和 wrusedw；在读侧，则使用 rdfull、rdempty 和 rdusedw。

注意，很多读者会混淆 wrusedw 和 rdusedw，误认为 wrusedw 指示写了多少个数据，每写一个就会加 1；rdusedw 指示读了多少个数据，每读一个就会加 1，这是不正确的。这两个都指示现在 FIFO 保存有多少个数据，意义是一样的，区别仅是时钟域不同。

通常，我们会使用 wrfull 和 rdempty，而不是 wrempty 和 rdfull。因为一般情况下，写数据时会判断是否会溢出，而不关心是否会空，所以写数据时使用 wrfull，而不用 wrempty。同理，读数据时，会判断是否为空，而不关心是否已经满了，所以读数据时使用 rdempty，而不用 rdfull。

第三步，确定 FIFO 的宽度和深度，见图 4－10。

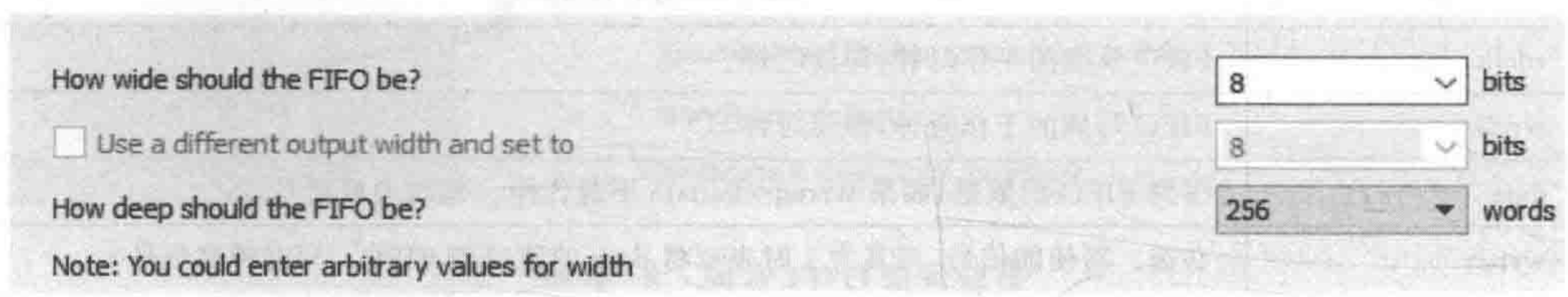


图 4－10　确定 FIFO 宽度和深度

1）FIFO 宽度

FIFO 的宽度可以根据需要定义，并且不受 2 的 n 次方限制。

确定 FIFO 宽度，也就是要考虑保存什么数据。FIFO 宽度不等于数据的宽度，很多时候为了处理方便，在保存数据的同时，会保存指示信号。例如以太网报文数据位宽是 32 位，可以把数据 data 和报文头 sop、报文尾 eop 一起保存到 FIFO 中，这样读数据时，就可以立刻知道读出的数据是不是报文头和报文尾了。

由于前面已经提及，数据 FIFO 不仅能保存数据，还能保存 sop eop 等指示信号。为了保存这些信号，就要适当增加 FIFO 宽度，要能一拍保存所有信息。

信息 FIFO 的宽度，根据要保存的信息来组成，多个信息要合并起来，一拍写到 FIFO 中。例如要用到 16 位的长度信息和 16 位的检验和信息，那么这两个合并起来就是 32 位，一起写到信息 FIFO 中，而不是分两次写。

2）FIFO 深度

根据需求，确定 FIFO 的深度。为了确定合理的 FIFO 深度，需要根据应用场景，分析其最恶劣的突发情况，确保在最恶劣的情况下都能不丢失数据。

在讲解 FIFO 深度之前必须要先讲清楚速率的相关概念。

时钟频率和速率：

系统都是在一个稳定的时钟频率下工作的，FIFO 的读、写时钟也不例外，其频率都是稳定不变的。如写时钟为 100 MHz，读时钟为 80 MHz 等，这些就是时钟频率。

但时钟快并不表示速率快，速率是指有效的数据速率，表示一段时间内有效数据的快慢。

如有 1 个 FIFO，其位宽是 16 位，写时钟为 100 MHz，读时钟为 80 MHz。假设每 4 个写时钟才有 1 个写使能，每 2 个读时钟就有 1 个读使能，则写速率为 100 MHz×16 bit/4＝400 Mbit/s，读速率为 80 MHz×16 bit/2 ＝ 640 Mbit/s。

平均速率：

平均速率是指一段时间内平均的速率，例子中写使能有时是 10 个时钟才有 1 个写使能，有时是 2 个时钟有 1 个写使能，平均是每 8 个写时钟才有 1 个写使能，则其写平均速率是 100 MHz×16 bit/4＝200 Mbit/s。

突发速率：

突发速率是指其可以达到的最快速率。例如当每个时钟都有使能时，上述 FIFO 的写突发速率就是 100 MHz×16 bit/1＝1 600 Mbit/s，而读突发速率是 80 MHz×16 bit/1＝1 280 Mbit/s。

3）FIFO 深度的影响

我们可以把 FIFO 想像成一个湖，这个湖有一个入水口和一个出水口。我们的目标是希望进来的水能一滴不丢失地出去。下面分析湖的深度对这个目标的影响：

① 如果入水的速度比出水的速度慢，那么湖无论深浅，进来的水应该全部都能出去，所以湖的深浅没有任何影响。

② 如果入水的速度和出水的速度相等，那么湖无论深浅，进来的水也全部都能出去，所以湖的深浅也没有任何影响。

③ 如果入水的速度比出水的速度快，那么无论湖有多深，湖水都会有溢出的时候，只不过湖越深，溢出越久才出现而已。

对比可知，如果 FIFO 的写平均速率比读平均速率小或相等，那么 FIFO 的深度可以设置很小。但如果写平均速率比读平均速率大，那么无论 FIFO 有多深，都有溢出的时候。

在稳健的系统中，一般都是不允许丢失数据的，如果出现写平均速率比读平均速率大而丢失数据的情况，增加 FIFO 深度，是不能从根本上解决问题的。解决的方法是更改架构，确保读平均速率比写平均速率大，这才是稳定的系统（注意，这里讲的是平均速率）。

既然系统要保证读平均速率比写平均速率大，那么是否意味着 FIFO 的深度没有什么作用，可以随便设置？

还是以上述湖水的例子来回答此问题。湖挖得越深，能抵御的暴雨就越大，例如原来可以抵御百年一遇暴雨，挖深一些，就可以抵御千年一遇暴雨了。

FIFO 深度的作用也是类似，虽然系统要确保读平均速率比写平均速率大，但难免会出现有一段时间写突发速率大于读突发速率的情况。如一段时间内连续写使能有效，或者连续的读使能无效。这时就需要用 FIFO 暂存一段数据，FIFO 深度越大，能保存的数据就越多。

4）深度计算

为了确定合理的 FIFO 深度，需要根据应用场景，分析其最恶劣的突发情况，确保在最恶劣的情况下都不丢失数据。下面通过 3 个例子来说明深度计算方法。

例 1 有 1 个 FIFO，其位宽是 16 位，写时钟为 100 MHz，读时钟为 80 MHz。写平均速率是 400 Mbit/s，读平均速率是 640 Mbit/s。经过分析使用的场合，发现 100 μs 内最多会有 3 200 bit 数据进来，而读使能时刻都会保持 640 Mbit/s 的速率。那么 FIFO 的深度设置为多少合理呢？

分析 100 μs 内最多有 3 200 bit 数据，这些数据可以集中突发过来，即在 200 个写时钟内每个时钟都有数据。那么最恶劣的场景：第 1 个 100 μs 周期内的最后 200 个写时钟和第 2 个 100 μs 周期内前 200 个写时钟连续来数据，即连续 400 个写数据有效。换句话说，FIFO 的深度要能保证连续 400 个写数据有效时，都不会溢出，合理的深度是多少呢？

设 FIFO 深度为 x，可以得出以下公式：

$$400\times\left(\frac{1}{100\ \text{MHz}}\right)=(400-x)\times\left(\frac{1}{80\ \text{MHz}}\right)$$

得到 $x=80$，由于 FIFO 深度只能是 2 的 N 次方，所以 FIFO 深度为 128 个最好。

上述公式左边计算的是写 400 个数据所需的时间，公式右边是读走 $400-x$ 个数据所需要的时间。

例 2 有 1 个 FIFO，其位宽是 16 位，写时钟为 100 MHz，读时钟为 80 MHz。写平均速率是 400 Mbit/s，读平均速率是 640 Mbit/s。经过分析使用的场合，发现 100 μs 内最多会有 3 200 bit 数据进来，而读使能平均速率是 640 Mbit/s，但有时候会连续不读数据。请问，FIFO 的深度设置多少合理？

分析 100 μs 内最多有 3 200 bit 数据，这些数据可以集中突发过来，即在 200 个写时钟内每个时钟都有数据。那么最恶劣的场景：第 1 个 100 μs 周期内的最后 200 个写时钟和第 2 个 100 μs 周期内前 200 个写时钟连续来数据，即连续 400 个写数据有效，同时刚好这段时间不读取 FIFO 的数据，此场景下 FIFO 合理的深度是多少呢？

此最恶劣的场景下只写不读，因此 FIFO 要保存全部写进来的数据，且深度要大于 400 个，取 2 的 N 次方，就是 512 个。

现实中经常会有这种应用场景：读平均速率比写平均速率大，但下游模块要求

读出的报文数据必须连续不间断，即从 sop 到 eop 数据是一直有效的。这个时候就需要先将进来的数据保存到 FIFO，当积累到一定量时，才连续读出数据，形成不间断的报文。这个积累的量就是发送水线。下面的例子说明了这个水线值如何设置。

例 3 某个以太网模块，数据位宽为 8 bit，写工作时钟是 80 MHz，写速率是 640 Mbit/s，读时钟是 100 MHz，读速率是 800 Mbit/s。假设输入的报文均是 8 192 B，且一旦开始输入一个报文，则连续不断输入。要求输出的报文都是连续不间断的。请问发送水线为多少是最合理的(存够多少数据才开始发送)?

分析 当然，发送水线设置成 8 192 B 肯定是没问题的，但这样 FIFO 的深度就要很大，从而造成资源浪费且报文的延迟也比较长。设发送水线为 x 字节，则有以下等式：

$$8\ 192\times\left(\frac{1}{100\ \text{MHz}}\right)=(8\ 192-x)\times\left(\frac{1}{80\ \text{MHz}}\right)$$

得到 x 约为 1 639 B。由于 FIFO 的深度只能是 2 的 N 次方，所以 FIFO 的深度设置为 2 048 个。可以看出，这个 FIFO 深度比直接按报文长度设置小很多。

上述公式左边计算的是读 8 192 个数据所需的时间，公式右边是写入(8 192－x)个数据所需要的时间。

第四步，设置使用 Show-ahead 模式，见图 4－11。

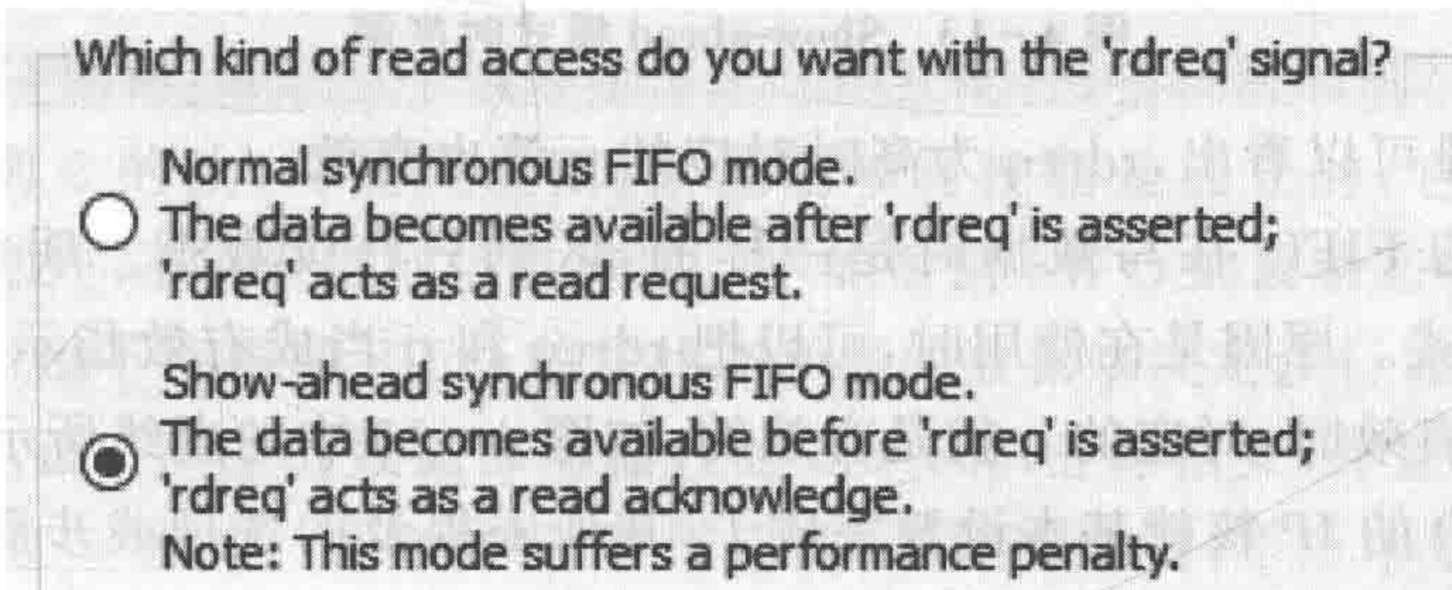


图 4－11 选择 FIFO 工作模式

下面介绍 Show-ahead 模式与 Normal 模式的区别，以及为什么明德扬会推荐使用 Show-ahead 模式。

1) Normal 模式

先有读使能，之后 FIFO 才输出这个数据。

例如，先后写入 D0、D1 和 D2 三个数据。此时，FIFO 信号 q 输出无效的数据。当收到 rereq＝1 后的第一个 clk 的上升沿，FIFO 把数据输出到数据总线，数据在下一个时钟周期才有效。接收到第二个、第三个 rdreq 后，D1、D2 输出到总线上，如图 4－12 所示。

这也意味着，当 rdreq 有效时，q 还无效，下一拍才有效(虚线所示时刻)，即先有

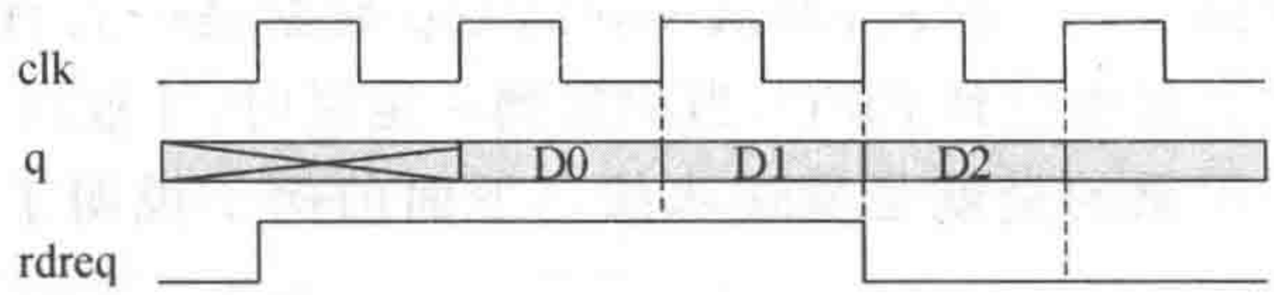


图 4-12　Normal 模式时序图

读使能，才有读数据。

2）Show-ahead 模式

FIFO 先输出数据，遇到读使能后，FIFO 更新输出数据。

Show-ahead 模式 FIFO，当 FIFO 中存在有效数据时，FIFO 会首先将第一个数据输出在总线上，当收到 rdrep=1 时，下一拍将输出第二个数据，如果没有第二个数据，则输出无效数据。

假设 FIFO 先后写了三个数据 D0、D1 和 D2，此时 q 输出第一个数据 D0，如果没有 rdreq，则一直输出 D0。当收到 rdreq==1 时，下一个时钟上升沿 D1 输出到总线上，接收到第二个 rdreq 后，D2 输出到总线上，如图 4-13 所示。

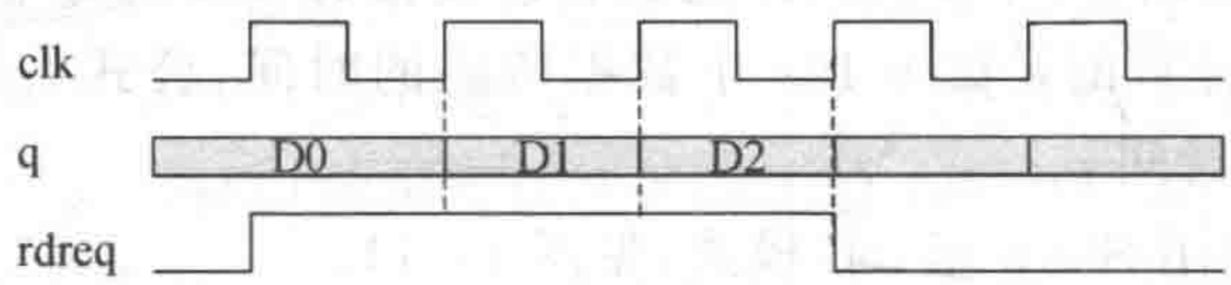


图 4-13　Show-ahead 模式时序图

从图中明显可以看出，rdreq 为高时对应的 q 值也有效。

两种模式的 FIFO 在写数据时是一样的，区别只在读数据。明德扬推荐使用 Show-ahead 模式。原因是在使用时，可以把 rdreq 和 q 当成有效指示信号和数据来使用，当 rdreq 有效时，对应的 q 就是有效的，如图 4-13 中的虚线所示。

到此，FIFO 的 IP 核就基本设置完成了，其他未提及的选项或步骤，只需要根据软件提示或默认生成即可。

下面举例讲解 FIFO 架构八步法。

一个模块输入的报文格式如下：

数据 1	数据 2	数据 3	…	数据 $n-2$	数据 $n-1$

经过模块处理后，输出的报文格式如下：

长度 n	数据 1	数据 2	…	数据 $n-2$	数据 $n-1$

其中，长度 n 是通过对输入报文统计得到的。也就是说，模块要先将输入的报文保存到 FIFO 中，同时对输入的报文数据进行统计。等到收到完整报文时（即收到 din_eop）才开始输出。输出时，先输出统计长度 n，然后再输出原来的报文数据。本例程的信号列表见表 4-3。

表 4-3　FIFO 信号列表

信号名	I/O	位　宽	说　明
rst_n	I	1	复位信号
clk	I	1	写时钟
din	I	8	输入的报文数据
din_sop	I	1	输入的报文头指示信号
din_eop	I	1	输入的报文尾指示信号
din_vld	I	1	输入的报文数据有效指示信号
dout	O	8	输出的报文数据
dout_sop	O	1	输出的报文头指示信号
dout_eop	O	1	输出的报文尾指示信号
dout_vld	O	1	输出的报文数据有效指示信号

注意：输入报文最大为 250 字节；每个输入报文之间的间隔至少为 1 个时钟周期（即 din_eop 和下一个 din_sop 不能连续出现）。输出报文无限制。

4.1.2　第一步：确认 FIFO 架构

确认 FIFO 架构：通过一个或多个 FIFO 相互配合，实现所需的功能。

FIFO 规则 2 的读、写隔离规则：读控制和写控制是独立的，它们之间除了用 FIFO 交流信息外，不能有任何信息传递。不能根据 FIFO 的读状态或者读出的数据来决定写行为，也不能根据 FIFO 的写状态和写的数据来决定读行为。

结合例题，读 FIFO 的数据都与输入的 din_eop 密切相关，并且读出的数据，还与输入报文长度 n 有关。这个时候，要谨记读、写隔离规则，千万不要让读和写有任何交互，din_eop 不可以，长度 n 也不行。

那么应该如何处理呢？其实认真想一想为什么会出错就会明白，既然进来的数据要通过 FIFO 来缓一缓，等待处理。同样的道理，din_eop 也需要缓一缓来等待处理。根据这个思路，再增加一个 FIFO，用来保存这个所需要的信息。为了区分，这里称其为信息 FIFO，而保存数据的 FIFO 称为数据 FIFO。

信息 FIFO 的作用有以下两点：

① 保存由写侧产生的，并由读侧使用的数据；

② 信息 FIFO 的空标志和保存个数统计，用来指示数据 FIFO 里报文的个数。

本例的实现架构为双 FIFO 架构。

数据 FIFO(DFIFO)用于保存输入报文的数据，写数据是{din，din_eop}，写使能是 din_vld。

信息 FIFO(MFIFO)用于保存报文个数的信息,写数据为长度 len,写使能是 din_eop。

通过 MFIFO 的 empty 信号,可以知道 DFIFO 是否有一个完整的报文,通过 MFIFO 的 usedw 信号,可以知道 DFIFO 中有多少个完整的报文。由于 FIFO 使用的是 Show-ahead 模式,此时 MFIFO 已经输出当前长度信息 n,因此可以将当前 n 的信息赋给输出,但不读 MFIFO。只有当输出完一个完整报文后,才读走 MFIFO,每读一次 MFIFO,表示处理完一个报文。

4.1.3 第二步:读状态

读状态是指在此状态下可以读 FIFO 数据,不在此状态时就不读 FIFO 数据。

有些 FIFO 的读状态可以一直为 1,只要 FIFO 有数据,就读取 FIFO 的数据,即没有启动条件,如非空即读情景。

而有些模块的读状态是分阶段的。这些 FIFO 则有启动条件,当条件不满足时,不会读 FIFO 的数据;当条件满足时,开始读 FIFO 的数据,并一直读到结束条件为止。那么应该有一个状态信号(如 flag)来指示当前状态。当 flag=0 就是条件未满足前的状态;flag=1 就是条件满足后的状态。

有了这个状态指示信号,则读数据很容易产生:flag=1 并且数据 FIFO 非空时产生读使能,需要注意一定要用组合逻辑。

本例的数据 FIFO——DFIFO 是有启动条件的,因为 DFIFO 在一个完整报文存入后才能开始读,本题的读状态指示信号是 rd1_flag。具体的启动条件分析请看第三步。

4.1.4 第三步:读状态启动和结束条件

如果 FIFO 的读状态一直为 1,则不需要考虑启动条件和结束条件;如果读状态是分阶段的,则需要认真考虑启动条件和结束条件。

启动条件,即什么时候开始读 FIFO 数据,也就是读状态什么时候由 0 变 1。启动条件一般有以下几种:

① FIFO 存够指定个数的数据;

② FIFO 保存有一个完整报文;

③ 其他特殊条件。

本例的启动条件是,信息 FIFO——MFIFO 非空,empty2==0(即 DFIFO 保存有一个完整的报文)。

结束条件,即什么时候结束读 FIFO 数据,也就是读状态什么时候由 1 变成 0。

以报文为单位的输出,一般是发送到报文最后一个数据就结束。但读侧如何知道当前数据是报文的最后一个数据呢?一个方法是计算读出来的长度,当读出来的长度等于写进来的长度,就表示结束点。另一种更简单的方法是,将 din_eop 和输入数据一起,保存到数据 FIFO。当读到的 FIFO 数据中的 eop 值为 1 时,表示报文到

了结束点。

本例中，写到数据 FIFO——DFIFO 的数据 data1 为 data1＝ {din，din_eop}；写使能为 wrreq1 ＝ din_vld。那么读出来的数据中 eop 指示为 eop_flag ＝ rdreq1 &&q[0]。当检测到 eop_flag＝1 时，表示读到报文最后一个数据，此时读状态结束。

4.1.5　第四步：信息 FIFO 读清条件

信息 FIFO 的作用有以下两点：

① 保存由写侧产生的、并由读侧使用的数据；

② 信息 FIFO 的空标志和保存个数统计，用来指示数据 FIFO 中报文的个数。

在发送报文的过程中，可以随时使用信息 FIFO 的数据，但什么时候将信息 FIFO 的数据读走呢？

一般有两种做法：一是使用后读走；二是处理完整个报文后再读走。

明德扬建议，在处理完整个报文时，才读走信息 FIFO 的数据。这是因为：

不同应用场景下，使用信息 FIFO 的数据是不相同的，有些场景可能开始就使用，有些场景可能到中间时使用，甚至有些场景是多次使用。所以，如果在使用后就读走，读者势必设计时要考虑这个因素；而如果统一在处理完整个报文后再读走，则无须再考虑这个因素，因为这适用于所有的场景。

而且信息 FIFO 是用于指示当前报文信息的，它指示的信息应该与当前报文对齐。如果是中间改变，那么指示就显得很混乱。而在结束时读走（可理解为清走一个数据），就可以完全与当前报文对齐，在调试时就非常方便。

本例的信息 FIFO——MFIFO 的读清条件就是 eop_flag，此时读完整个报文，可清除 MFIFO 中的对应数据。回顾第三步的 eop_flag，发现 rdreq2 与 eop_flag 变化情况一致，因此本例代码省去 eop_flag。

4.1.6　第五步：生成 FIFO 的 IP 核

根据题目要求，读、写时钟是同一个，因此选择同步 FIFO。

DFIFO 宽度为 9 bit，即 8 bit 数据 din 加 1 bit 报文尾指示信号 din_eop。报文最大长度 250 字节，因此至少选择 256 字节深度。

MFIFO 宽度为 8 bit（报文最大长度 250 字节，其二进制表示为 1111_1010）。深度至少为 256 字节（考虑最坏情况输入进来一个 250 字节报文后，又连续输入很多个 1 字节的报文，在 250 字节报文输出完成前，所有 1 字节报文的长度都必须存在 MFIFO 中）。

4.1.7　第六步：其他信号的产生条件

在本例中，要统计输入报文长度，因此需要引入一个计数器 cnt，用于统计报文长度。此外，还有 4 个输出信号 dout、dout_vld、dout_sop 和 dout_eop 需要考虑。

(1) cnt

cnt 的初值为 0；

cnt 的加 1 条件：din_vld==1，输入一个 din_vld 表示收到 1 字节的报文；

cnt 的结束条件：din_eop==1。

(2) dout

dout 输出长度 q2：DFIFO 未开始读且 MFIFO 非空，rdreq1 == 0&&empty2==0；

dout 输出数据 q1[8:1]：DFIFO 读使能有效，rdreq1==1。

(3) dout_vld

dout_vld 初值为 0；

dout_vld 从 0 变 1：(rdreq1==0 && empty2==0)||rdreq1==1；

其他时刻 dout_vld 均为 0。

(4) dout_sop

dout_sop 初值为 0；

dout_sop 从 0 变 1：输出长度的时刻，rdreq1==0 && empty2==0；

其他时刻 dout_sop 均为 0。

(5) dout_eop

dout_eop 初值为 0；

dout_eop 从 0 变 1：数据读出结束时刻，rdreq1==1 && q1[0] ==1；

其他时刻 dout_eop 均为 0。

4.1.8 第七步：FIFO 代码

```
fifo1 u_fifo1(
        .clock (clk   ),
        .data  (data1 ),
        .rdreq (rdreq1),
        .wrreq (wrreq1),
        .empty (empty1),
        .q     (q1    )
);

fifo2 u_fifo2(
        .clock (clk   ),
        .data  (data2 ),
        .rdreq (rdreq2),
        .wrreq (wrreq2),
        .empty (empty2),
```

```
16      .q      (q2     )
17 );
18
19 assign wrreq1 = din_vld;
20 assign data1  = {din[7:0],din_eop};
21
22 assign wrreq2 = end_cnt;
23 assign data2  = cnt + 1;
24
25 always  @(posedge clk or negedge rst_n)begin
26      if(rst_n == 1'b0)begin
27              rd1_flag <= 1'b0;
28                  end
29      else if(start)begin
30              rd1_flag <= 1'b1;
31      end
32      else if(rdreq2)begin
33              rd1_flag <= 1'b0;
34      end
35 end
36
37 assign rdreq1 = rd1_flag;
38 assign start  = rdreq1 == 0 && empty2 == 0;
39 assign rdreq2 = rdreq1 == 1 && q1[0] == 1;
```

4.1.9 第八步:其他信号代码

```
1  //根据第六步第 1 点,写出 cnt 的代码
2  always  @(posedge clk or negedge rst_n)begin
3      if(rst_n == 1'b0)begin
4          cnt <= 0;
5      end
6      else if(add_cnt)begin
7          if(end_cnt)
8              cnt <= 0;
9          else
10             cnt <= cnt + 1;
11     end
12 end
13
14 assign add_cnt = din_vld;
15 assign end_cnt = add_cnt && din_eop;
```

```
16
17 //根据第六步第 2 点,写出 dout 的代码
18 always  @(posedge clk or negedge rst_n)begin
19      if(rst_n == 1'b0)begin
20          dout <= 8'b0;
21      end
22      else if(start)begin
23          dout <= q2;
24      end
25      else if(rdreq1)begin
26          dout <= q1[8:1];
27      end
28 end
29
30 //根据第六步第 3 点,写出 dout_vld 的代码
31 always  @(posedge clk or negedge rst_n)begin
32      if(rst_n == 1'b0)begin
33          dout_vld <= 1'b0;
34      end
35      else if(start || rdreq1)begin
36          dout_vld <= 1'b1;
37      end
38      else begin
39          dout_vld <= 1'b0;
40      end
41 end
42
43 //根据第六步第 4 点,写出 dout_sop 的代码
44 always  @(posedge clk or negedge rst_n)begin
45      if(rst_n == 1'b0)begin
46          dout_sop <= 1'b0;
47      end
48      else if(start)begin
49          dout_sop <= 1'b1;
50      end
51      else begin
52          dout_sop <= 1'b0;
53      end
54 end
55
56 //根据第六步第 5 点,写出 dout_eop 的代码
57 always  @(posedge clk or negedge rst_n)begin
58      if(rst_n == 1'b0)begin
59          dout_eop <= 1'b0;
```

```
60      end
61      else if(rdreq2)begin
62          dout_eop <= 1'b1;
63      end
64      else begin
65          dout_eop <= 1'b0;
66      end
67  end
```

4.1.10 总　结

本节介绍了 FIFO 架构设计八步法，在 FIFO 的 IP 核使用时，需要明确写条件、读条件以及 FIFO 中应当存储的数据是什么。FIFO 在工程中应用非常广泛，希望读者能够通过八步法掌握好 FIFO 的使用。

4.2 FIFO 项目实践

本节给出了 FPGA 设计中 FIFO 的典型应用，希望读者通过本节的学习掌握 FIFO 架构八步法。

4.2.1 报文汇聚模块项目

在生活中，经常会遇到图 4－14 所示的情况，比如说有三台电脑，它们的数据通过一个交换机收集起来或一根线传出去，把电脑 A、B、C 的数据送给第一个二级交换机，把电脑 D、E、F 的数据送给第二个二级交换机，然后把两个交换机的数据给一级交换机。

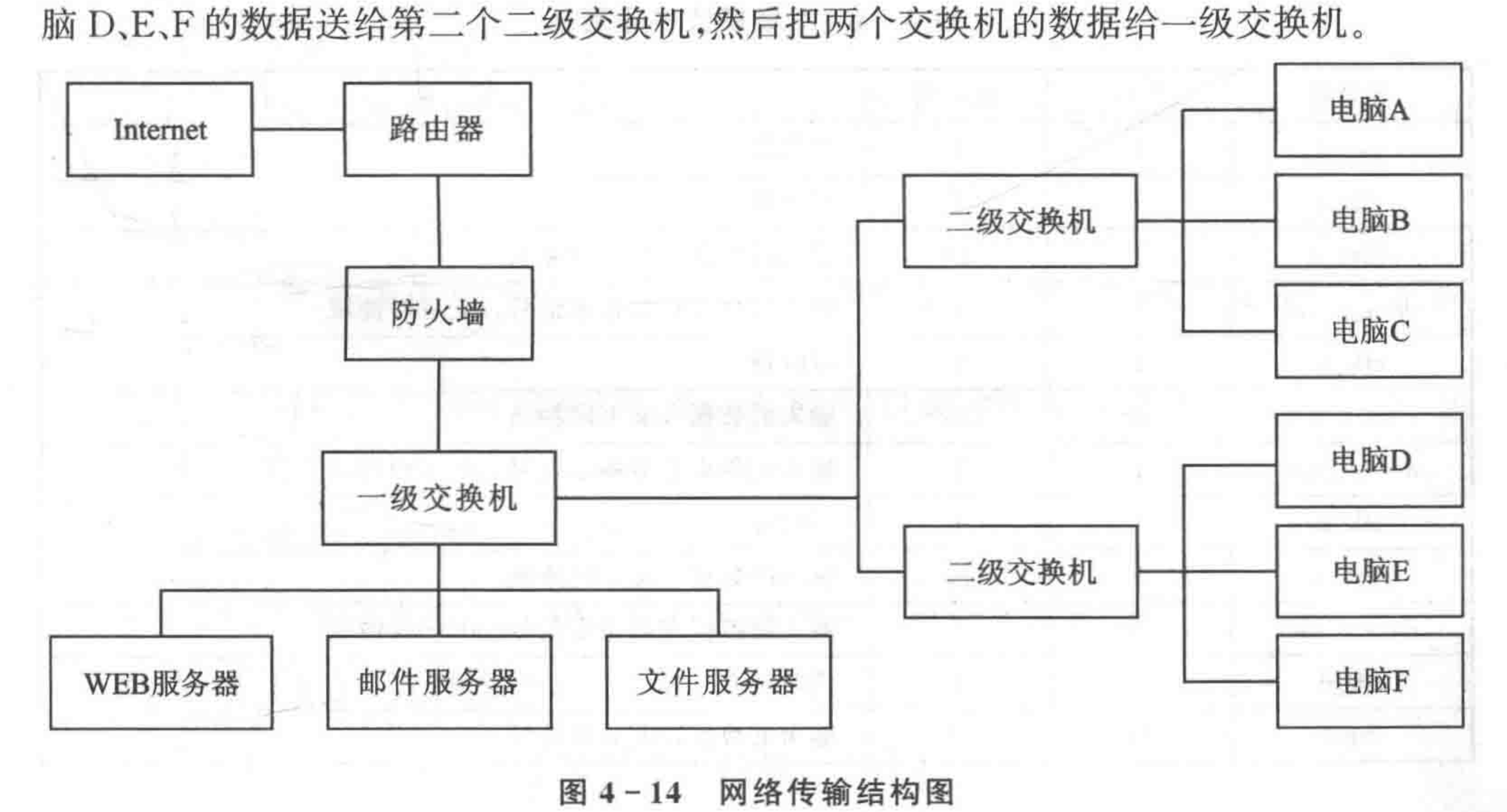


图 4－14　网络传输结构图

这种数据的汇聚，在 FPGA 里也有相似的情况，如图 4-15 所示，类似 3 个网口，分别使用 3 个 FIFO 缓存数据，然后通过逻辑判断需要读出哪个 FIFO 的数据，从而保证数据传输的可靠性。

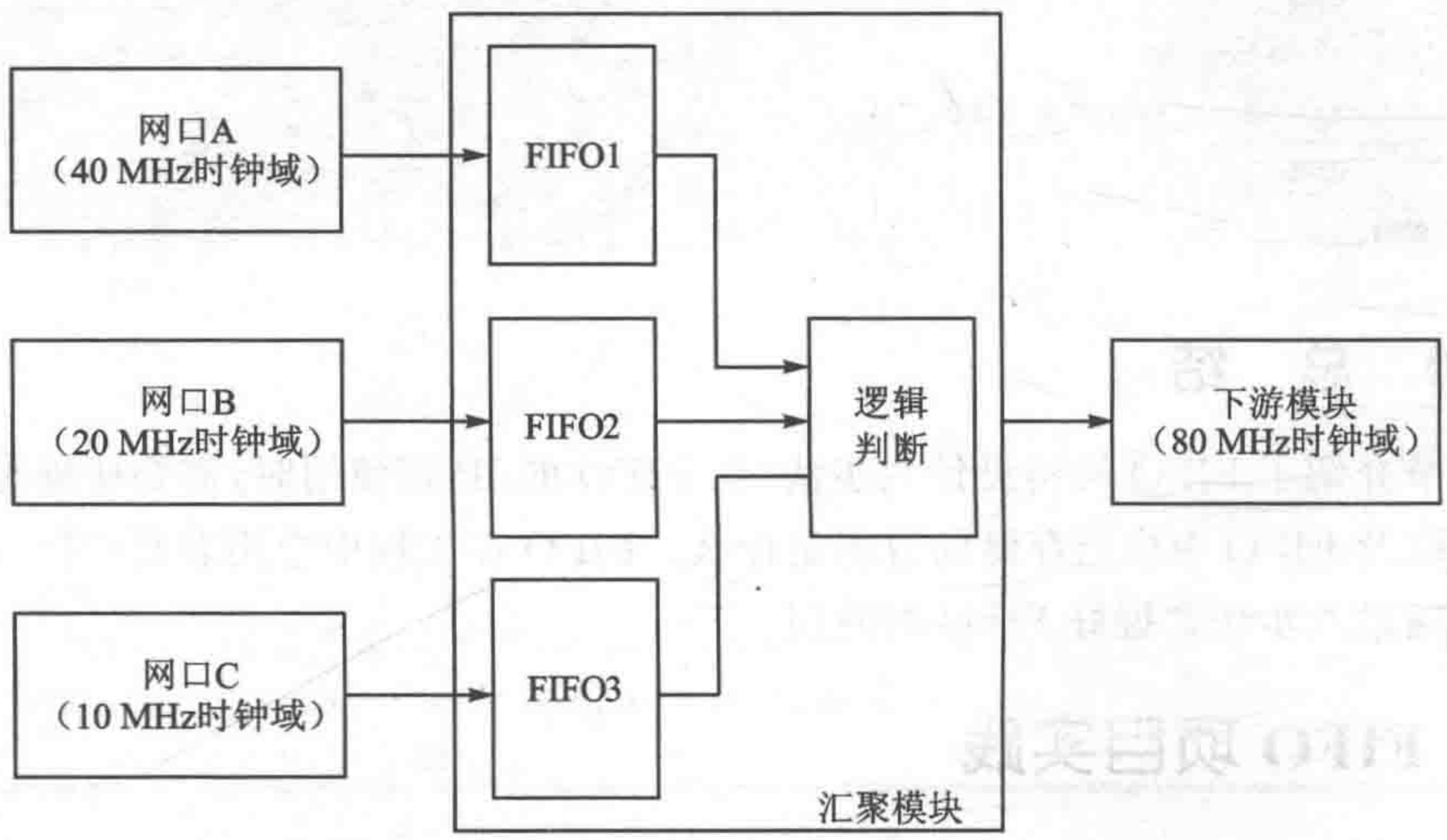


图 4-15　汇聚模块应用图

有三个网口，分别是网口 A、网口 B 和网口 C，网口的数据要通过下游模块传输，下游模块就相当于交换机一样，三个网口就相当于 3 台电脑。传输汇聚模块将三个网口的数据汇集起来，统一送往下游模块。其中，网口 A 的工作时钟是 40 MHz，网口 B 的工作时钟是 20 MHz，网口 C 的工作时钟是 10 MHz，下游模块的工作时钟是 80 MHz。汇聚模块的信号列表见表 4-4。

表 4-4　汇聚模块信号列表

信号名	I/O	位　宽	说　明
rst_n	I	1	复位信号
clk_a	I	1	写时钟
data_a	I	16	输入的数据，clk_a 时钟域
data_a_vld	I	1	输入的数据有效指示信号，clk_a 时钟域
clk_b	I	1	写时钟
data_b	I	16	输入的数据，clk_b 时钟域
data_b_vld	I	1	输入的数据有效指示信号，clk_b 时钟域
clk_c	I	1	写时钟
data_c	I	16	输入的数据，clk_c 时钟域
data_c_vld	I	1	输入的数据有效指示信号，clk_c 时钟域
clk_d	I	1	读时钟
dout	O	16	输出的数据，clk_d 时钟域

续表 4－4

信号名	I/O	位　宽	说　明
dout_vld	O	1	输出数据有效指示信号，clk_d 时钟域
channel	O	2	输出通道号指示信号，表示当前数据来自于哪一个网口，当 dout_vld 有效时，此信号有效。clk_d 时钟域。0：网口 A；1：网口 B；2：网口 C；其他：无意义

结合表 4－4 完成汇聚模块的设计。该模块具有如下功能：

① 模块内部包含三个 FIFO，每个 FIFO 数据位宽均为 16 bit，深度为 64 个，有 empty 指示信号。三个 FIFO 命名为 FIFO1～3，分别存储模块 A、B、C 的数据；每组输入数据进来后，分别保存到三个 FIFO 中。

② 汇聚模块查询三个 FIFO 的状态，如果看到 FIFO 中有数据，就从 FIFO 中读取数据，送给下游模块。汇聚模块首先检查 FIFO1，如果 FIFO1 有数据，则读取送给下游模块；如果 FIFO1 没有数据，而 FIFO2 有数据，则读取 FIFO2 数据；如果 FIFO1 和 FIFO2 均无数据，而 FIFO3 有数据，则读取 FIFO3 数据。

③ 输出数据的同时，通过 chan_d 指示数据是来自于哪个网口的，以便下游模块做处理。

1. 确认 FIFO 架构

根据模块功能说明，可知本模块有 3 个数据 FIFO，分别是 FIFO1、FIFO2 和 FIFO3。由于不需要对网口来的数据进行处理，只是用 FIFO 解决读、写速率匹配的问题。因此本项目不需要信息 FIFO。

2. 读状态

首先考虑优先级，FIFO1＞FIFO2＞FIFO3，要求优先级高的 FIFO 先读空，再开始读优先级低的 FIFO。

再考虑读、写速率问题。因为 3 个 FIFO 的数据写入总速率（40 MHz＋20 MHz＋10 MHz）小于数据读出速率（80 MHz），所以在满足优先级的情况下，FIFO 的数据会一直被读出。

因此，本模块 3 个 FIFO 的读状态一直为 1，只要 FIFO 有数据，满足优先级条件，就可以读取 FIFO 数据。

3. 读状态启动和结束条件

3 个 FIFO 的读状态一直为 1，因此不需要考虑启动条件和结束条件。

4. 信息 FIFO 读清条件

本题无信息 FIFO，因此不需要考虑信息 FIFO 读清条件。

5. 生成 FIFO 的 IP 核

模块功能说明中已经给出，FIFO 的宽度为 16 bit，深度为 64 个。由于读、写时钟不同，因此使用异步 FIFO。

6. 其他信号的产生条件

考虑输出信号 dout、dout_vld 以及输出通道号指示信号 channel。

(1) dout

dout 的值为对应 FIFO 的 q 值。

当 rdreq1==1 时，dout 的值为 q1；

当 rdreq2==1 时，dout 的值为 q2；

当 rdreq3==1 时，dout 的值为 q3。

(2) dout_vld

dout_vld 初值为 0；

dout_vld 从 0 变 1：rdreq1==1 || rdreq2==1 || rdreq3==1；

其他时刻 dout_vld 均为 0。

(3) channel

当 rdreq1==1 时，channel 的值为 0；

当 rdreq2==1 时，channel 的值为 1；

当 rdreq3==1 时，channel 的值为 2。

7. FIFO 代码

```
1  fifo_a u_fifo_a(
2      .data       (data_a    ),
3      .rdclk      (clk_rd    ),
4      .rdreq      (rdreq_a   ),
5      .wrclk      (clk_a     ),
6      .wrreq      (data_a_vld),
7      .q          (q_a       ),
8      .rdempty    (rdempty_a )
9  );
10
11 fifo_b u_fifo_b(
12     .data       (data_b    ),
13     .rdclk      (clk_rd    ),
14     .rdreq      (rdreq_b   ),
15     .wrclk      (clk_b     ),
16     .wrreq      (data_b_vld),
17     .q          (q_b       ),
```

```
    .rdempty     (rdempty_b )
);

fifo_c u_fifo_c(
    .data        (data_c     ),
    .rdclk       (clk_rd     ),
    .rdreq       (rdreq_c    ),
    .wrclk       (clk_c      ),
    .wrreq       (data_c_vld),
    .q           (q_c        ),
    .rdempty     (rdempty_c )
);

always  @( * )begin
    if(!rdempty_a)
        rdreq_a = 1;
    else
        rdreq_a = 0;
end

always  @( * )begin
    if(rdempty_a && ! rdempty_b)
        rdreq_b = 1;
    else
        rdreq_b = 0;
end

always  @( * )begin
    if(rdempty_a && rdempty_b && ! rdempty_c)
        rdreq_c = 1;
    else
        rdreq_c = 0;
end
```

8. 其他信号代码

```
//根据第六步第 1 点,写出 dout 的代码
always  @(posedge clk_rd or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        dout <= 0;
    end
    else if(rdreq_a)begin
```

```
7            dout <= q_a;
8        end
9        else if(rdreq_b)begin
10           dout <= q_b;
11       end
12       else if(rdreq_c)begin
13           dout <= q_c;
14       end
15       else begin
16           dout <= 0;
17       end
18 end
19
20 //根据第六步第 2 点,写出 dout_vld 的代码
21 always  @(posedge clk_rd or negedge rst_n)begin
22       if(rst_n == 1'b0)begin
23           dout_vld <= 0;
24       end
25       else begin
26           dout_vld <= rdreq_a || rdreq_b || rdreq_c;
27       end
28 end
29
30 //根据第六步第 3 点,写出 channel 的代码
31 always  @(posedge clk_rd or negedge rst_n)begin
32       if(rst_n == 1'b0)begin
33           channel <= 0;
34       end
35       else if(rdreq_a)begin
36           channel <= 0;
37       end
38       else if(rdreq_b)begin
39           channel <= 1;
40       end
41       else if(rdreq_c)begin
42           channel <= 2;
43       end
44 end
```

4.2.2 以太网切包模块项目

以太网有很多种,最常用到的是 Ethernet II。Ethernet II 即 DIX 2.0:Xerox 与

DEC、Intel 等公司在 1982 年制定的以太网标准帧格式，见图 4－16。

目标MAC地址	源MAC地址	类型	数据	FCS
6字节	6字节	2字节	46~1 500字节	4字节

图 4－16　以太网标准帧格式

Ethernet II 类型以太网帧的最小长度为 64 字节(6＋6＋2＋46＋4)，最大长度为 1 518 字节(6＋6＋2＋1 500＋4)。其中，前 12 字节分别标识发送数据帧的源节点 MAC 地址和接收数据帧的目标节点 MAC 地址。(注：ISL 封装后可达 1 548 字节，802.1Q 封装后可达 1 522 字节)为什么会有最大长度限制呢？

① 以太网是所有的终端共享传输介质，如果一台终端发送一个很长的帧，传输时将占用很长的时间，其他的终端将等待，直到这个帧传输完毕。帧越长，等待的时间就越长。

② 如果一个帧太长，在传输的过程中，若发生一个传输错误，那么整个帧要重传，这样又要占用大量的时间，导致效率不高——早期的以太网传输的错误率非常高。

③ 帧越长，需要的缓存越大，1979 年在设计网卡时，缓存的价格非常昂贵，所以当时选用 1 500 字节作为最大的长度。

由帧格式可以看出，数据的长度只能在 46～1 500 字节之间。但实际用户发送的 1 个报文长度会小于 46 字节或大于 1 500 字节，这时就要把这个大报文切成多个以太网报文，使每个报文都控制在 46～1 500 字节。

本项目要求输入报文范围在 1～9 600 字节，报文与报文的间隔至少 46 字节，设计以太网切包模块。

以太网切包模块的功能是对输入的报文进行切割，使每个输出的报文都控制在 46～1 500 字节范围内。

如果输入报文小于 46 字节，则通过在末尾补 0 的方式，输出长度为 46 字节的报文。

如果输入报文在 46～1 500 字节，则输出报文等于输入的报文。

如果输入的报文大于 1 500 字节，则输出 1 500 字节报文，剩余的报文再进行切割。如果剩余的报文小于 46 字节，则通过在末尾补 0 的方式，输出长度为 46 字节的报文。

例如输入 3 001 字节的报文，则输出 2 个 1 500 字节的报文和 1 个长度为 46 字节(补 45 字节的 0)的报文。输入 3 100 字节报文，则输出 2 个 1 500 字节报文和 1 个长度为 100 字节的报文。切包模块信号列表见表 4－5。

表 4-5 切包模块信号列表

信号名	I/O	位 宽	说 明
clk	I	1	写时钟
rst_n	I	1	复位信号
din	I	8	输入的数据
din_vld	I	1	输入数据有效指示信号
din_sop	I	1	输入数据报文头指示信号
din_eop	I	1	输入数据报文尾指示信号
dout	O	8	输出的报文数据
dout_vld	O	1	输出的报文数据有效指示信号
dout_sop	O	1	输出的报文数据报文头指示信号
dout_eop	O	1	输出的报文数据报文尾指示信号

1. 确认 FIFO 架构

按照切包模块的要求，应该对输入报文的长度进行统计，然后判断是切包、补 0 还是直接输出。因此本项目的 FIFO 架构与 4.1.1 小节最后所举的例子类似，也使用了数据 FIFO(data_fifo)与信息 FIFO(len_fifo)。

为了统计输入报文的字节数，引入计数器 cnt_wr，计数器加 1 条件 add_cnt_wr 是 din_vld(收到 1 字节输入数据)，计数器的结束条件 end_cnt_wr 是计数到 1 500－1 或收到 din_eop(报文结束)。

那么，当 din_vld==1 且 cnt_wr==0 时，此时的 din 为将要输出子报文的首字节，定义该时刻为 fir_cnt_wr；在 end_cnt_wr 时刻，din 为输出子报文的尾字节。

以下面例子讲述各个信号之间的关系，各信号的关系如图 4-17 所示。

din 的输入数据有两段，长度分别为 1 200 字节和 1 530 字节，根据项目要求，第一段不需要处理，第二段则要切成 1 500＋30 两部分，第二部分还要补 0 至 46 字节，每一个数据包的起始字节 fir_cnt_wr 拉高，结束字节 end_cnt_wr 拉高。

data_fifo 用于保存输入报文的数据，写数据是{din，din_sop，din_eop，fir_cnt_wr，end_cnt_wr}，写使能是 din_vld。

len_fifo 用于保存报文长度的信息，写数据为长度 cnt_wr＋1，写使能是 end_cnt_wr。

2. 读状态

虽然在图 4-17 中，dout 与 din 是对齐的，但实际上它们在时序上并不是对齐的，这里仅是为了方便示图。data_fifo 是在报文输入完毕或处理完毕后才可以读出，

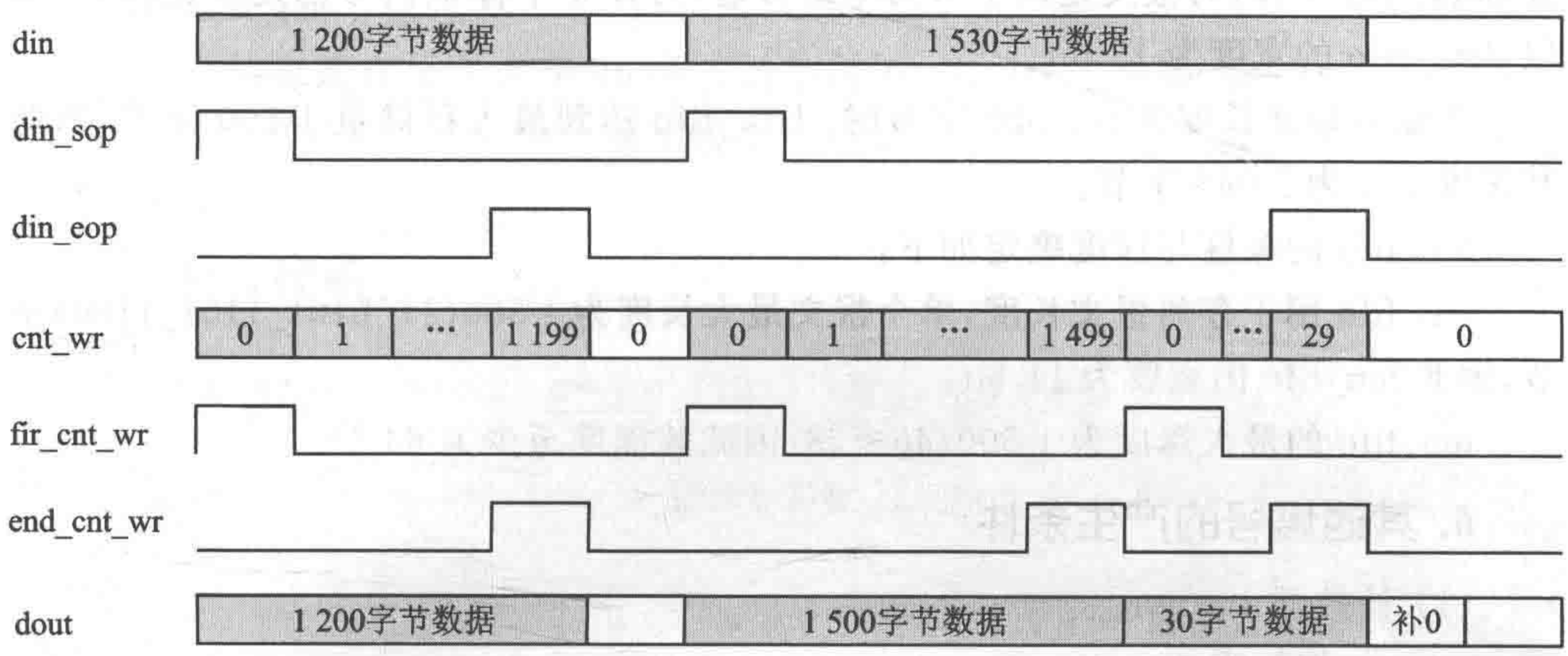


图 4－17　计数结构图

因此它的读状态是有启动条件和结束条件的。这在下面的第三步中考虑。

3. 读状态启动和结束条件

在考虑读状态启动和结束条件之前，要先考虑读数据的一种特殊情况，即 din 数据长度小于 46 字节时，要在输出数据末尾补 0，最终输出长度为 46 字节的报文（通过 dout_eop 来指示报文结束）。但在补 0 的过程中，FIFO1 中存储的数据是不能读出的，因为此时再读 FIFO1，得到的会是下一个报文的数据，所以需要一个指示信号 flag_zero（有效则表示现在正在进行补 0）和一个补 0 计数器 cnt_zero。计数器 cnt_zero 的加 1 条件是 flag_zero 有效，结束条件是计数到 46－len（数据长度）－1。

data_fifo 读状态启动条件：在 FIFO1 和 FIFO2 都有数据时（表示有一个完整的报文存储在 FIFO1 中），flag_zero＝＝0（表示不需要补 0 或者已经补完 0）；

data_fifo 读状态结束条件：data_fifo_rdreq && data_fifo_q[0]。

4. 信息 FIFO 读清条件

本项目的信息 FIFO 为 len_fifo，需要分情况考虑 len_fifo 的读清条件：

当输入报文长度小于 46 字节时，收到 end_cnt_zero（表示补 0 结束），则 FIFO2 读清；

当其他长度报文输入，且从 data_fifo 读出来的数据最低为 1 时，FIFO2 读清。

5. 生成 FIFO 的 IP 核

切包是在一个模块内完成的，因为读、写侧共用一个工作时钟，所以选择同步 FIFO。

data_fifo 的宽度与深度确定如下：

为了更易判断存入 FIFO 的字节是不是报文头（din_sop）或报文尾（din_eop），在报文长度大于 1 500 字节需要切包时，更要标识切出来子报文的首字节与尾字节。因此，我们把 din，din_sop，din_eop，以及子报文的首字节与子报文的尾字节一起存

进 data_fifo 中，因为报文是以字节为单位传输，另外 4 个标识信号都是单比特的，所以 data_fifo 的宽度为 12 bit。

当输入报文长度>=1 500 字节时，data_fifo 达到最大存储量 1 500 字节，因此其深度至少为 2 048 字节。

len_fifo 的宽度与深度确定如下：

len_fifo 用于存放报文长度，单个报文最大长度为 1 500(11'b101_1101_1100)字节，因此 len_fifo 的宽度为 11 bit。

len_fifo 的最大深度为 1 500/46≈33，因此其深度至少为 64 个。

6. 其他信号的产生条件

(1) 计数器 cnt_wr

cnt_wr 初值为 0；

cnt_wr 加 1 条件：din_vld==1；

cnt_wr 结束值：cnt_wr==1 500−1 或 din_eop==1。

(2) 计数器 cnt_zero

cnt_zero 初值为 0；

cnt_zero 加 1 条件：flag_zero==1；

cnt_zero 结束值：cnt_zero==46−len_fifo_q−1。

(3) flag

flag 从 0 变 1：flag==0 且 flag_zero==0 且 len_fifo_empty==0；

flag 从 1 变 0：data_fifo_rdreq==1 且写入此字节时 end_cnt_wr==1，表示该字节是此段输出数据的最后一字节。

(4) flag_zero

flag_zero 从 0 变 1：当前字节的数据长度 len_fifo_q < 46 字节且写入此字节时 end_cnt_wr==1，表示该字节是此段输出数据的最后一字节，而且这段数据是小于 46 字节的。

flag_zero 从 1 变 0：end_cnt_zero==1，表示补 0 完成。

(5) dout

由于我们对 din 做了如下操作：data_fifo_data = {din，din_sop，din_eop，fir_cnt_wr，end_cnt_wr}。因此在 data_fifo_rdreq 有效时，输出 dout 是 data_fifo_q[11:4]。

(6) dout_sop

在读到 data_fifo_q[1]==1 时，dout_sop 等于 1。其他时刻都为 0。

(7) dout_vld

在 data_fifo_rdreq 或者 flag_zero 有效时，表示正在输出报文数据或正在补 0，此时的输出都是有效的，因此 dout_vld 为 1。其他时刻都为 0。

(8) dout_eop

在补 0 结束或在报文大于 46 字节的情况下，读到该字节输入时刻的 end_cnt_wr 为 1，表示该字节是最后一个输出报文的最后一字节，此时 dout_eop 为 1。其他时刻都为 0。

7. FIFO 代码

```
1  /*********www.mdy-edu.com 明德扬科教 注释开始*********
2                          数据 FIFO
3   *********www.mdy-edu.com 明德扬科教 注释结束*********/
4  fifo_12b data_fifo(
5          .clock    (clk              ),
6          .data     (data_fifo_data ),
7          .rdreq    (data_fifo_rdreq),
8          .wrreq    (data_fifo_wrreq),
9          .empty    (data_fifo_empty),
10         .q        (data_fifo_q     )
11 );
12
13 assign data_fifo_wrreq = din_vld;
14 assign data_fifo_data  = {din,din_sop,din_eop,fir_cnt_wr,end_cnt_wr};
15 assign data_fifo_rdreq = flag && data_fifo_empty == 1'b0;
16
17 //根据第六步第 1 点，写出 cnt_wr 的代码
18 always   @(posedge clk or negedge rst_n)begin
19      if(rst_n == 1'b0)begin
20          cnt_wr <= 0;
21      end
22      else if(add_cnt_wr) begin
23          if(end_cnt_wr)
24              cnt_wr <= 0;
25          else
26              cnt_wr <= cnt_wr + 1;
27      end
28 end
29
30 assign  add_cnt_wr = din_vld;
31 assign  end_cnt_wr = add_cnt_wr && ((cnt_wr == 1500 - 1) || din_eop);
32 assign  fir_cnt_wr = add_cnt_wr && cnt_wr == 0;
33 assign  data_fifo_q_eoc = data_fifo_rdreq && data_fifo_q[0];
34
35 /*********www.mdy-edu.com 明德扬科教 注释开始*********
```

```
                                长度 FIFO
********www.mdy-edu.com 明德扬科教 注释结束*********/
fifo_12b len_fifo(
        .clock    (clk            ),
        .data     (len_fifo_data  ),
        .rdreq    (len_fifo_rdreq ),
        .wrreq    (len_fifo_wrreq ),
        .empty    (len_fifo_empty ),
        .q        (len_fifo_q     )
        );

assign len_fifo_wrreq = end_cnt_wr;
assign len_fifo_data  = cnt_wr + 1;
assign len_fifo_rdreq = (data_fifo_q_eoc && start_cnt_zero == 1'b0) || end_cnt_zero;

/*********www.mdy-edu.com 明德扬科教 注释开始*********
                              中间信号
********www.mdy-edu.com 明德扬科教 注释结束*********/
//根据第六步第 3 点,写出 flag 的代码
always  @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        flag <= 1'b0;
    end
    else if(flag == 1'b0 && flag_zero == 1'b0 && len_fifo_empty == 1'b0) begin
        flag <= 1'b1;
    end
    else if(data_fifo_q_eoc)begin
        flag <= 1'b0;
    end
end

//根据第六步第 4 点,写出 flag_zero 的代码
always  @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        flag_zero <= 1'b0;
    end
    else if(start_cnt_zero) begin
        flag_zero <= 1'b1;
    end
    else if(end_cnt_zero)begin
        flag_zero <= 1'b0;
    end
```

```
end
assign start_cnt_zero = data_fifo_q_eoc && len_fifo_q<46;

//根据第六步第 1 点,写出 cnt_zero 的代码
always   @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        cnt_zero <= 0;
    end
    else if(add_cnt_zero) begin
        if(end_cnt_zero)
            cnt_zero <= 0;
        else
            cnt_zero <= cnt_zero + 1;
    end
end
assign add_cnt_zero = flag_zero;
assign end_cnt_zero = add_cnt_zero && cnt_zero == (46 - len_fifo_q - 1);
```

8. 其他信号代码

```
//根据第六步第 5 点,写出 dout 的代码
always   @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        dout <= 0;
    end
    else if(data_fifo_rdreq) begin
        dout <= data_fifo_q[11:4];
    end
    else begin
        dout <= 0;
    end
end

//根据第六步第 6 点,写出 dout_sop 的代码
always   @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        dout_sop <= 0;
    end
    else if(data_fifo_q_soc) begin
        dout_sop <= 1'b1;
    end
    else begin
```

```
23            dout_sop <= 1'b0;
24        end
25 end
26 assign  data_fifo_q_soc = data_fifo_rdreq && data_fifo_q[1];
27
28 //根据第六步第 7 点,写出 dout_vld 的代码
29 always   @(posedge clk or negedge rst_n)begin
30        if(rst_n == 1'b0)begin
31            dout_vld <= 1'b1;
32        end
33        else if(data_fifo_rdreq || flag_zero) begin
34            dout_vld <= 1'b1;
35        end
36        else begin
37            dout_vld <= 1'b0;
38        end
39 end
40
41 //根据第六步第 8 点,写出 dout_eop 的代码
42 always   @(posedge clk or negedge rst_n)begin
43        if(rst_n == 1'b0)begin
44            dout_eop <= 1'b0;
45        end
46        else if(end_cnt_zero || (data_fifo_q_eoc && start_cnt_zero == 1'b0)) begin
47            dout_eop <= 1'b1;
48        end
49        else begin
50            dout_eop <= 1'b0;
51        end
52 end
```

4.2.3 UDP 打包模块项目

1. 背景介绍

UDP 协议全称是用户数据报协议，在网络中它与 TCP 一样用于处理数据报，是一种无连接的协议。在 OSI 模型中，在第四层——传输层，处于 IP 的上一层。UDP 有不提供数据报分组、组装和不能对数据报进行排序等缺点，也就是说，当报文发送之后，是无法得知其是否安全完整到达的。UDP 用来支持那些需要在计算机之间传输数据的网络应用。包括网络视频会议系统在内的众多的客户/服务器模式的网络应用都要使用 UDP。

UDP 分为首部字段和数据字段，其中首部字段只占用 8 字节，分别是各占用 2 字节的源端口、目的端口、长度和校验和，UDP 帧格式见图 4－18。

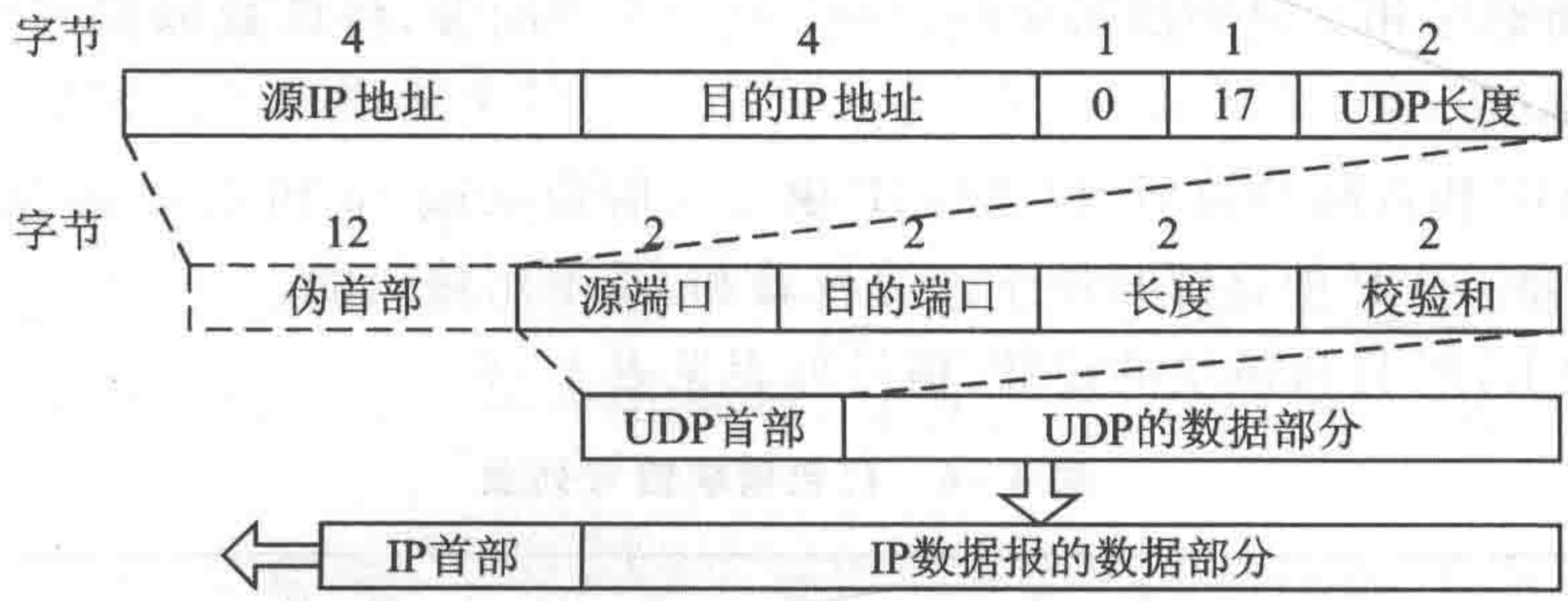


图 4－18　UDP 帧格式

长度：整个 UDP 报文的大小，最小为 8 字节（仅为首部）。

校验和：在进行校验和计算时，会添加伪首部一起进行运算。伪首部（占用 12 字节）为：4 字节的源 IP 地址、4 字节的目的 IP 地址、1 字节的 0、1 字节的数字 17 以及占用 2 字节 UDP 长度。这个伪首部不是报文的真正首部，只是引入为了计算校验和。相对于 IP 的只计算首部，UDP 校验和会把首部和数据一起进行校验。接收端进行的校验和与 UDP 报文中的校验和相“与”，如果无差错应该全为 1。如果有误，则将报文丢弃或者发给应用层，并附上差错警告。

以太网 UDP 打包模块功能是对进来的报文数据添加 UDP 首部，然后发送给下游模块。

输入的报文如下：

数据 1	数据 2	数据 3	…	数据 $n-2$	数据 $n-1$

输出的报文如下：

UDP 首部	数据 1	数据 2	…	数据 $n-2$	数据 $n-1$

(1) UDP 首部包括源端口、目的端口、长度和校验和。

(2) 首部的源端口和目的端口由输入信号 sour_port 和 dest_port 给出。

(3) 首部的长度包括 UDP 首部和 UDP 数据的长度，单位为字节。UDP 首部长度固定为 8 字节，UDP 数据长度来自对输入报文的统计。

(4) 首部校验和计算的是 UDP 伪首部、UDP 首部和 UDP 数据的校验和。其计算方法如下：

① 把伪首部添加到 UDP 上。

② 计算初始时是需要将校验和字段添 0 的。

③ 把所有位划分为 16 位（2 字节）的字。

④ 把所有 16 位的字相加，如果遇到进位，则将高于 16 字节的进位部分的值加

到最低位上。举例，16'hBB5E＋ 16'hFCED＝17'h1 B84B，将进位 1 加到最低位，得到结果是 16'hB84C。

⑤ 将所有字相加得到的结果应该是一个 16 位的数，将该数取反则可以得到校验和 checksum。

(5) UDP 伪首部的源 IP 和目的 IP 由输入信号 sour_ip 和 dest_ip 给出。

(6) 注意，UDP 伪首部只用于计算校验和，是不用输出的。

请完成 UDP 打包模块的设计，信号列表见表 4－6。

表 4－6　打包模块信号列表

信号名	I/O	位　宽	说　明
rst_n	I	1	复位信号
clk	I	1	写时钟
dest_port	I	16	目的端口
sour_port	I	16	源端口
dest_ip	I	32	目的 IP 地址
sour_ip	I	32	源 IP 地址
din	I	8	输入的数据
din_vld	I	1	输入数据有效指示信号
din_sop	I	1	输入数据报文头指示信号
din_eop	I	1	输入数据报文尾指示信号
dout	O	8	输出的报文数据
dout_vld	O	1	输出的报文数据有效指示信号
dout_sop	O	1	输出的报文数据报文头指示信号
dout_eop	O	1	输出的报文数据报文尾指示信号

2. 设计过程

(1) 确认 FIFO 架构

按照 UDP 打包模块的要求，需要对输入报文的长度进行统计，并且计算报文的校验和，最后组合成“UDP 首部＋数据”的格式输出。因此本项目的 FIFO 架构与 4.1 节的例题及 4.2.2 小节以太网切包模块类似，使用了数据 FIFO(data_fifo)与信息 FIFO(mes_fifo)。

同样的，与前面项目的做法类似，为了统计输入报文的字节数，我们引入计数器 cnt_wr，计数器加 1 条件 add_cnt_wr 是 din_vld(收到 16 bit 输入数据)，计数器的结束条件 end_cnt_wr 是收到 din_eop(报文输入结束)。此外，还要注意向 data_fifo 输入的数据是 16 bit 的，而长度的单位是字节(8 bit)。因此，报文长度 data_len 是 cnt_wr＋1 的 2 倍。

明确了如何统计报文长度后，就需要确定校验和如何计算。按照题目，校验和的计算包含伪首部、首部以及数据部分，因此在报文输入 data_fifo 的过程中，就可以对数据部分进行校验和计算，得出来的数据部分校验和（data_check_sum）存入 mes_fifo 中，在报文输出前再把报文的校验和 udp_check_sum 计算出来（包含伪首部、首部以及 data_check_sum）。计算过程如下：

$$\boxed{\text{UDP校验和}} = \boxed{\text{伪首部}} + \boxed{\text{UDP首部}} + \boxed{\text{data_check_sum}}$$

data_fifo 用于保存输入报文的数据，写数据是{din ，din_eop }，写使能是 din_vld。

mes_fifo 用于保存报文长度和输入数据部分校验和的信息，写数据是{data_check_sum，data_len}，写使能是 end_cnt_wr，由于计算校验和，写使能需要用时序逻辑产生，以保证时序对齐。其原因在校验和计算中会详细说明。

（2）读状态

data_fifo 是在报文输入完毕或处理完毕后才可以读出，因此它的读状态是有启动条件和结束条件的。这在下面的第三步中考虑。

（3）读状态启动和结束条件

在明确读状态启动和结束条件之前，先来定义几个信号。首先定义报文输出指示信号 flag_rd，因为输出报文格式是“UDP 首部＋数据”，因此定义输出报文数据指示信号 flag_data。又引入了计数器 cnt_head，用于计数 UDP 首部依次输出的源端口 sour_port、目的端口 dest_port、报文长度 udp_len，以及校验和 udp_check_sum。波形图如图 4－19 所示。

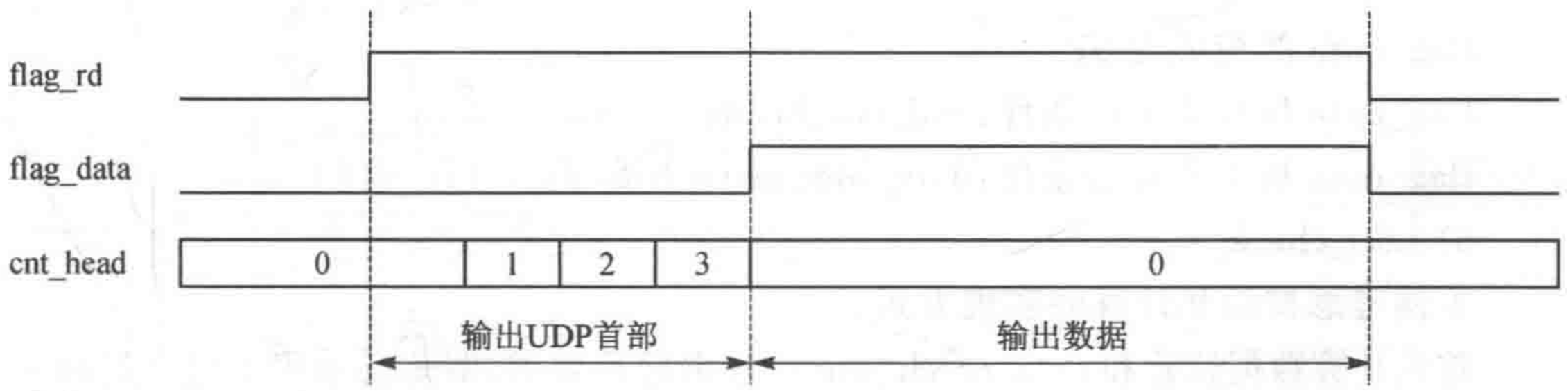


图 4－19　读状态启动和结束条件

data_fifo 的读状态启动条件：flag_data＝＝1，表示此时应该读出报文数据；

data_fifo 的读状态结束条件：data_fifo_rdreq && data_fifo_q[0]，读到 eop 就结束。

（4）信息 FIFO 读清条件

本项目的信息 FIFO 为 mes_fifo，mes_fifo 的读清是在报文输出完成时刻，即与 data_fifo 的读状态结束是同一时刻。在 data_fifo_rdreq && data_fifo_q[0]时，读清 mes_fifo。

(5) 生成 FIFO 的 IP 核

UDP 打包在一个模块内完成，因为读、写侧共用一个工作时钟，所以选择同步 FIFO。本题的上游模块与下游模块的读、写速率未知，为了简化设计，不考虑 FIFO 溢出的情况，所以此步骤不考虑 FIFO 的深度。

data_fifo 的宽度为 17 bit(16 bit 的 din+1 bit 的 din_eop)。

mes_fifo 的宽度 32 bit(16 bit 的 data_check_sum+16 bit 的 data_len)。

(6) 其他信号的产生条件

下面总结前面提到的几个简单信号的产生条件。

1) cnt_wr

cnt_wr 初值为 0；

cnt_wr 的加 1 条件：din_vld；

cnt_wr 的结束条件：din_eop。

2) cnt_head

cnt_head 初值为 0；

cnt_head 的加 1 条件：flag_rd==1 && flag_data==0；

cnt_head 的结束条件：cnt_head==4-1。

3) flag_rd

flag_rd 的初值为 0；

flag_rd 从 0 变 1 的条件：flag_rd==0 && mes_fifo_empty==0；

flag_rd 从 1 变 0 的条件：data_fifo_rdreq && data_fifo_q[0]。

4) flag_data

flag_data 的初值为 0；

flag_data 从 0 变 1 的条件：end_cnt_head；

flag_data 从 1 变 0 的条件：data_fifo_rdreq && data_fifo_q[0]。

5) udp_check_sum

下面考虑校验和计算的实现方式。

首先计算数据校验和 data_check_sum。根据题目要求，明德扬采用了组合逻辑+时序逻辑的方式计算校验和。实现方式如图 4-20 所示，假设 din 连续输入 3 个有效数据。

计算公式为

data_check_sum_tmp=data_check_sum+din

data_check_sum=data_check_sum_tmp[16]+data_check_sum_tmp[15:0]

其中：data_check_sum_tmp 用组合逻辑实现；data_check_sum 用时序逻辑实现。这样做就可以计算出校验和了，但是从图 4-20 可以看到，data_check_sum 出结果需要多 1 个时钟周期，因此在前面考虑 mes_fifo 的写使能时，需要用时序逻辑多打一拍，保证写入 mes_fifo 的 data_check_sum 是正确的结果。

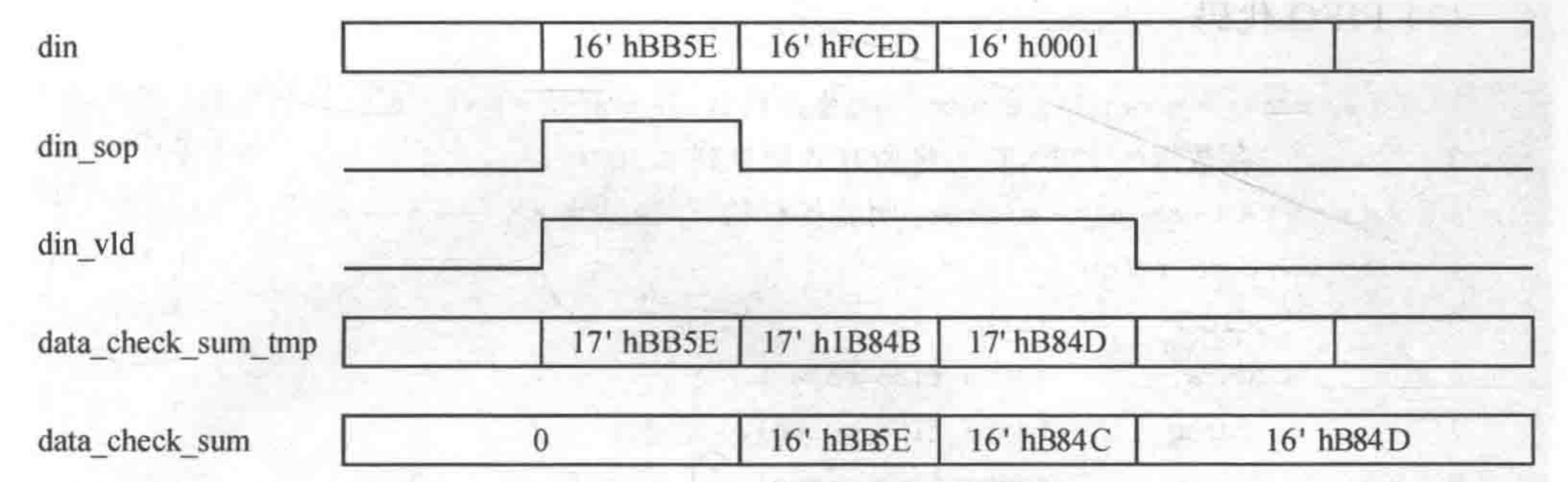


图 4-20　数据校验和计算

接着计算整个报文的校验和 udp_check_sum。

下面定义几个信号：

UDP 首部信号 head = {sour_port,dest_port,udp_len,udp_check_sum}

(UDP 伪首部＋UDP 首部)信号 pse_head = {sour_ip,dest_ip,16'd17,udp_len,head}

从题目对校验和的定义可以知道，要计算 udp_check_sum 的值，需要把 pse_head 分解成多段，每段 16 bit，使用上述计算数据校验和 data_check_sum 的方法对每段进行计算。明德扬在此处使用了流水线处理，如图 4-21 所示。

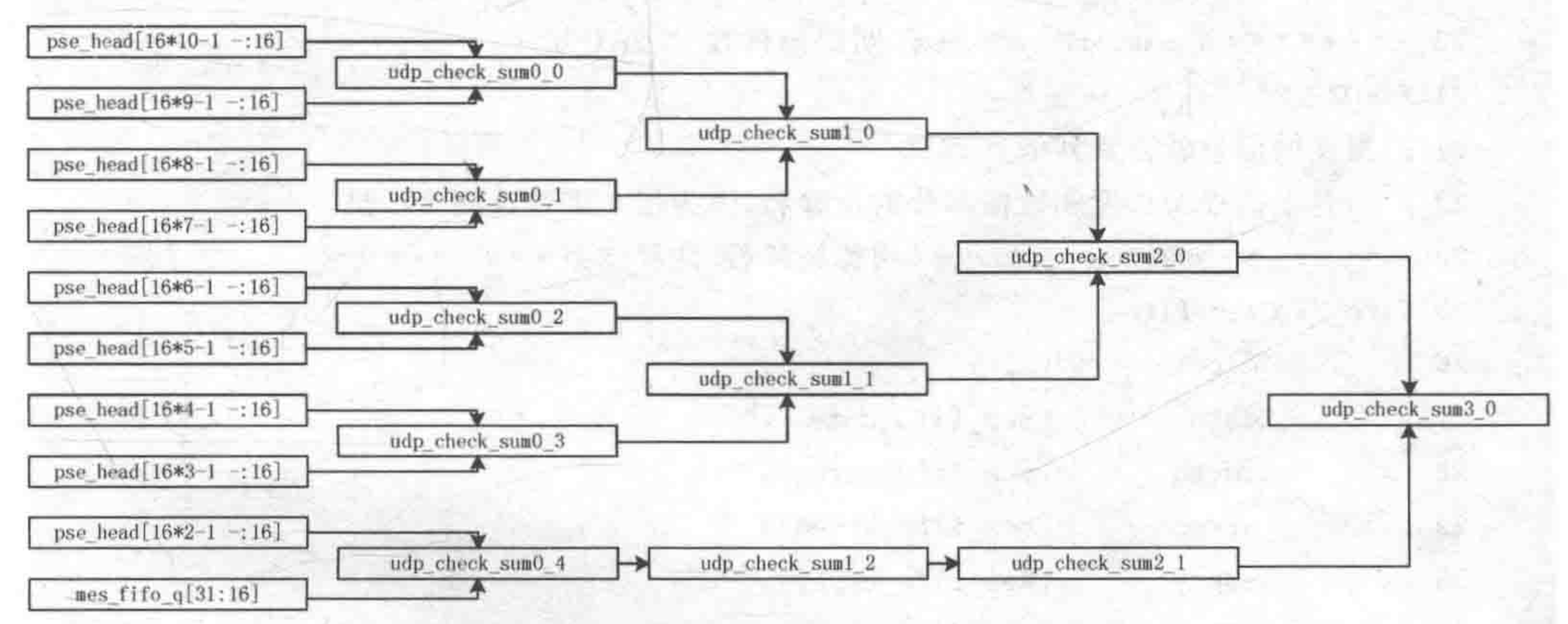


图 4-21　流水线处理

然后对 udp_check_sum3_0 取反，即可得到 udp_check_sum。完成一次 16 bit 的加法运算需要 1 个时钟周期，因此流水线需要 4 个时钟周期计算出结果，由于是时序逻辑，所以在第 5 个时钟周期，才可以读到最终的 udp_check_sum。

最后，考虑输出信号 dout，dout_sop、dout_vld 与 dout_eop。

dout 的输出顺序是源端口、目的端口、报文长度、校验和及数据。因此在第 4 个时钟周期就需要使用校验和的值。从上面的分析可以知道，校验和需要 5 个时钟周期才能读出，因此对 4 个输出信号都需要进行打一拍的处理，这样才能输出正确的值。

(7) FIFO 代码

```
/*********www.mdy-edu.com 明德扬科教 注释开始*********
        存数据的 FIFO,首先将所有数据存到 FIFO 中
 *********www.mdy-edu.com 明德扬科教 注释结束*********/
fifo_17b data_fifo(
        .clock      (clk            ),
        .data       (data_fifo_data ),
        .rdreq      (data_fifo_rdreq),
        .wrreq      (data_fifo_wrreq),
        .empty      (data_fifo_empty),
        .q          (data_fifo_q    )
);

assign data_fifo_wrreq = din_vld;
//加上 EOP,是为了读侧能够知道一个报文的结束
assign data_fifo_data   = {din,din_eop};
assign data_fifo_rdreq = flag_data && data_fifo_empty == 1'b0;
assign data_fifo_q_eop = data_fifo_rdreq && data_fifo_q[0];


/*********www.mdy-edu.com 明德扬科教 注释开始*********
FIFO2
//报文的信息就存储到信息 FIFO
//信息包括报文长度和数据部分的校验和,因为报文需要这两个信息
 *********www.mdy-edu.com 明德扬科教 注释结束*********/
fifo_32b mes_fifo(
        .clock      (clk            ),
        .data       (mes_fifo_data ),
        .rdreq      (mes_fifo_rdreq),
        .wrreq      (mes_fifo_wrreq),
        .empty      (mes_fifo_empty),
        .q          (mes_fifo_q     )
);

assign mes_fifo_data = {data_check_sum,data_len};//高 16 位是 UDP 数据的校验和

assign mes_fifo_rdreq = data_fifo_q_eop;  //当整个报文发送完的同时,读走信息
                                          //FIFO 一个数据

//信息 FIFO 的写使能,注意此时用了时序逻辑
//数据校验和要晚一拍,为了对齐所以用时序逻辑
```

```
always   @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        mes_fifo_wrreq <= 1'b0;
    end
    else begin
        mes_fifo_wrreq <= end_cnt_wr;
    end
end
```

(8) 其他信号代码

```
//根据第六步第 1 点,写出 cnt_wr 的代码
//统计一个报文的长度,因为 UDP 报文头有一个长度信息
always   @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        cnt_wr <= 0;
    end
    else if(add_cnt_wr) begin
        if(end_cnt_wr)
            cnt_wr <= 0;
        else
            cnt_wr <= cnt_wr + 1;
    end
end
assign  add_cnt_wr = din_vld;
assign  end_cnt_wr = add_cnt_wr && din_eop;

//为了与校验和对齐,所以打一拍
always   @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        data_len <= 0;
    end
    else if(end_cnt_wr) begin
        data_len <= {cnt_wr + 1,1'b0};
  //输入的数据一拍是 2 字节的,所以长度是要乘以 2;
    end
end

//统计数据部分的校验和,UDP 报文头有这部分内容
always   @( * )begin
    if(din_vld && din_sop)
        data_check_sum_tmp = din;
    else
        data_check_sum_tmp = data_check_sum + din;
end
```

```
35
36  always   @(posedge clk or negedge rst_n)begin
37      if(rst_n == 1'b0)begin
38          data_check_sum <= 0;
39      end
40      else if(din_vld) begin
41          data_check_sum <= data_check_sum_tmp[16] +
42                            data_check_sum_tmp[15:0];
43      end
44  end
45
46  //根据第六步第 3 点,写出 flag_rd 的代码
47  //flag_rd = 1 开始发送数据
48  always   @(posedge clk or negedge rst_n)begin
49      if(rst_n == 1'b0)begin
50          flag_rd <= 1'b0;
51      end
52      else if(flag_rd == 1'b0 && mes_fifo_empty == 1'b0) begin
53          flag_rd <= 1'b1;
54      end
55      else if(data_fifo_q_eop)begin
56          flag_rd <= 1'b0;
57      end
58  end
59
60  //根据第六步第 4 点,写出 flag_data 的代码
61  //flag_data = 1 表示发送的是数据
62  always   @(posedge clk or negedge rst_n)begin
63      if(rst_n == 1'b0)begin
64          flag_data <= 1'b0;
65      end
66      else if(end_cnt_head) begin
67          flag_data <= 1'b1;
68      end
69      else if(data_fifo_q_eop)begin
70          flag_data <= 1'b0;
71      end
72  end
73
74  //根据第六步第 2 点,写出 cnt_head 的代码
75  //对发送的 UDP 首部进行计数
76  always   @(posedge clk or negedge rst_n)begin
```

```
77        if(rst_n == 1'b0)begin
78            cnt_head <= 0;
79        end
80        else if(add_cnt_head) begin
81            if(end_cnt_head)
82                cnt_head <= 0;
83            else
84                cnt_head <= cnt_head + 1;
85        end
86    end
87
88    assign  add_cnt_head = flag_rd && flag_data == 1'b0;
89    assign  end_cnt_head = add_cnt_head && cnt_head == HEAD_LEN_2B - 1;
90
91    //组合成 UDP 首部和 UDP 伪首部,方便后面 dout 的产生
92    assign  udp_len          = mes_fifo_q[15:0] + 8;
93    //数据长度 + 8 字节 UDP 首部
94    assign  udp_check_sum = ~udp_check_sum3_0;
95    assign  head             = {sour_port,dest_port,udp_len,udp_check_sum};
96    //8 字节
97    assign  pse_head         = {sour_ip,dest_ip,16'd17,udp_len,head};
98    //12 + 8 字节
99
100   //根据第六步第 5 点,写出 udp_check_sum 的代码
101   //产生校验和:伪首部和数据总的校验和
102   //寄存器之间最大只有 1 个 16 位的加法器
103   /*********www.mdy-edu.com 明德扬科教 注释开始*********
104                                流水线处理
105    *********www.mdy-edu.com 明德扬科教 注释结束*********/
106   always   @(*)begin
107       udp_check_sum0_0_tmp = mes_fifo_q[31:16]       + pse_head[16*2-1 -:16];
108       udp_check_sum0_1_tmp = pse_head[16*3-1 -:16] + pse_head[16*4-1 -:16];
109       udp_check_sum0_2_tmp = pse_head[16*5-1 -:16] + pse_head[16*6-1 -:16];
110       udp_check_sum0_3_tmp = pse_head[16*7-1 -:16] + pse_head[16*8-1 -:16];
111       udp_check_sum0_4_tmp = pse_head[16*9-1 -:16] + pse_head[16*10-1 -:16];
112   end
113
114   always   @(posedge clk or negedge rst_n)begin
115       if(rst_n == 1'b0)begin
116           udp_check_sum0_0 <= 0;
117           udp_check_sum0_1 <= 0;
118           udp_check_sum0_2 <= 0;
```

```
119             udp_check_sum0_3 <= 0;
120             udp_check_sum0_4 <= 0;
121         end
122         else if(add_cnt_head && cnt_head == 0) begin
123             udp_check_sum0_0 <= udp_check_sum0_0_tmp[16]   +
124                                  udp_check_sum0_0_tmp[15:0];
125             udp_check_sum0_1 <= udp_check_sum0_1_tmp[16]   +
126                                  udp_check_sum0_1_tmp[15:0];
127             udp_check_sum0_2 <= udp_check_sum0_2_tmp[16]   +
128                                  udp_check_sum0_2_tmp[15:0];
129             udp_check_sum0_3 <= udp_check_sum0_3_tmp[16]   +
130                                  udp_check_sum0_3_tmp[15:0];
131             udp_check_sum0_4 <= udp_check_sum0_4_tmp[16]   +
132                                  udp_check_sum0_4_tmp[15:0];
133         end
134 end
135
136 always   @( * )begin
137         udp_check_sum1_0_tmp = udp_check_sum0_0 + udp_check_sum0_1;
138         udp_check_sum1_1_tmp = udp_check_sum0_2 + udp_check_sum0_3;
139         udp_check_sum1_2_tmp = udp_check_sum0_4                   ;
140 end
141
142 always   @(posedge clk or negedge rst_n)begin
143         if(rst_n == 1'b0)begin
144             udp_check_sum1_0 <= 0;
145             udp_check_sum1_1 <= 0;
146             udp_check_sum1_2 <= 0;
147         end
148         else if(add_cnt_head && cnt_head == 1) begin
149             udp_check_sum1_0 <= udp_check_sum1_0_tmp[16]   +
150                                  udp_check_sum1_0_tmp[15:0];
151             udp_check_sum1_1 <= udp_check_sum1_1_tmp[16]   +
152                                  udp_check_sum1_1_tmp[15:0];
153             udp_check_sum1_2 <= udp_check_sum1_2_tmp[16]   +
154                                  udp_check_sum1_2_tmp[15:0];
155         end
156 end
157
158 always   @( * )begin
159         udp_check_sum2_0_tmp = udp_check_sum1_0 + udp_check_sum1_1;
160         udp_check_sum2_1_tmp = udp_check_sum1_2                   ;
```

```
161 end
162
163 always   @(posedge clk or negedge rst_n)begin
164     if(rst_n == 1'b0)begin
165         udp_check_sum2_0 <= 0;
166         udp_check_sum2_1 <= 0;
167     end
168     else if(add_cnt_head && cnt_head == 2) begin
169         udp_check_sum2_0 <= udp_check_sum2_0_tmp[16]   +
170                             udp_check_sum2_0_tmp[15:0];
171         udp_check_sum2_1 <= udp_check_sum2_1_tmp[16]   +
172                             udp_check_sum2_1_tmp[15:0];
173     end
174 end
175
176
177 always   @( * )begin
178     udp_check_sum3_0_tmp = udp_check_sum2_0 + udp_check_sum2_1;
179 end
180
181 always   @(posedge clk or negedge rst_n)begin
182     if(rst_n == 1'b0)begin
183         udp_check_sum3_0 <= 17'hffff;
184     //初值要为 17'hffff,取反后 udp_check_sum 为 0;
185     end
186     else if(add_cnt_head && cnt_head == 3) begin
187         udp_check_sum3_0 <= udp_check_sum3_0_tmp[16]   +
188                             udp_check_sum3_0_tmp[15:0];
189     end
190 end
191
192 /*********www.mdy-edu.com 明德扬科教 注释开始*********
193                 输出 dout/dout_vld/dout_sop/dout_eop
194 *********www.mdy-edu.com 明德扬科教 注释结束*********/
195 //dout 来自于首部和 FIFO 的数据
196 //head = {sour_port,dest_port,udp_len,udp_check_sum};
197 always   @(posedge clk or negedge rst_n)begin
198     if(rst_n == 1'b0)begin
199         dout_temp <= 0;
200     end
201     else if(flag_rd) begin
202         if(flag_data)
```

```
203                 dout_temp <= data_fifo_q[16:1];
204             else
205                 dout_temp <= head[(HEAD_LEN_2B - cnt_head) * 16 - 1 - :16];
206         end
207         else begin
208             dout_temp <= 0;
209         end
210     end
211
212     //dout
213     always   @(posedge clk or negedge rst_n)begin
214         if(rst_n == 1'b0)begin
215             dout <= 0;
216         end
217         else if(end_cnt_head_ff0)begin
218             dout <= udp_check_sum;
219         end
220         else begin
221             dout <= dout_temp;
222         end
223     end
224
225     //end_cnt_head_ff0
226     always   @(posedge clk or negedge rst_n)begin
227         if(rst_n == 1'b0)begin
228             end_cnt_head_ff0 <= 0;
229         end
230         else begin
231             end_cnt_head_ff0 <= end_cnt_head;
232         end
233     end
234
235     //首部第 1 个数据自然是 SOP
236     always   @(posedge clk or negedge rst_n)begin
237         if(rst_n == 1'b0)begin
238             dout_sop_temp <= 0;
239         end
240         else if(add_cnt_head && cnt_head == 0) begin
241             dout_sop_temp <= 1'b1;
242         end
243         else begin
244             dout_sop_temp <= 1'b0;
```

```
245         end
246     end
247
248     always   @(posedge clk or negedge rst_n)begin
249         if(rst_n == 1'b0)begin
250             dout_sop <= 0;
251         end
252         else begin
253             dout_sop <= dout_sop_temp;
254         end
255     end
256
257     //输出数据有效指示
258     always   @(posedge clk or negedge rst_n)begin
259         if(rst_n == 1'b0)begin
260             dout_vld_temp <= 1'b1;
261         end
262         else begin
263             dout_vld_temp <= flag_rd;
264         end
265     end
266
267     always   @(posedge clk or negedge rst_n)begin
268         if(rst_n == 1'b0)begin
269             dout_vld <= 1'b1;
270         end
271         else begin
272             dout_vld <= dout_vld_temp;
273         end
274     end
275
276     //输出报文的最后一个数据
277     always   @(posedge clk or negedge rst_n)begin
278         if(rst_n == 1'b0)begin
279             dout_eop_temp <= 1'b0;
280         end
281         else begin
282             dout_eop_temp <= data_fifo_q_eop;
283         end
284     end
285
286     always   @(posedge clk or negedge rst_n)begin
```

```
287         if(rst_n == 1'b0)begin
288             dout_eop <= 1'b0;
289         end
290         else begin
291             dout_eop <= dout_eop_temp;
292         end
293 end
```

第二篇　模块划分

第 5 章　模块划分核心知识
第 6 章　模块划分常用架构及项目实践

第5章

模块划分核心知识

在前面的章节中，我们学习了单个模块的设计方法。但是当我们拿到一个项目后，最先要考虑的是模块划分，不同的人对同一个项目可能会有不同的模块划分，但是模块划分得好坏直接影响到设计、调试的难易程度，而初学者的设计往往一个工程一个模块，在调试仿真时需要同时考虑很多信号，耗时且费力。很多书籍在介绍项目时会直接给出划分结果而省略过程，读者有没有想过模块是如何划分的，是依据什么原则划分的。为了让读者初窥模块划分的门径，明德扬总结了模块划分的两个原则，以及模块的常用信号，结合实例进行模块划分思路的讲解。读者基于这些原则多实践，积累经验，一定可以掌握好模块划分。

5.1 模块划分概述

模块划分，顾名思义是指模块的划分。但是，由于模块划分时，必须同时考虑模块之间的信号交互及数据流向，所以明德扬在这里提出的模块划分，是广义的“模块划分”。后续所提及的“模块划分”，不仅指模块的划分，还包括模块划分好后，模块的端口及模块之间数据流向的确定。

5.1.1 模块划分原则

明德扬的模块划分原则有两条。

原则一：一个功能一个模块。

按功能划分模块是模块划分最基本的指导思想，当拿到一个项目后，最直接的想法就是先确定项目需要多少小功能为之服务，然后一个一个功能来实现，最后通过顶层模块的例化，完成项目要求。

原则二：相同架构一个模块。

这里的相同架构不仅指模块整体为计数器架构、状态机架构或 FIFO 架构，还要求在架构内部相同或相似。比如：计数器架构中计数器的个数或计数规律相同或高度相似；状态机架构中状态的划分或跳转条件相同或高度相似。

5.1.2 模块划分端口规范

在确定模块划分后，需要明确模块的端口及模块与模块之间的数据交互。明德扬在实际项目经验总结得到的一般模块端口信号规范，如表5-1所列。读者可在完成项目模块划分后，确定端口及数据流向时参考使用。

表5-1 端口信号规范

信 号	说 明
clk	模块工作时钟
rst_n	系统复位信号，低电平有效
en	en有两种用法： 1.使能命令信号，类似于vld，当其为1时表示一个命令有效； 2.使能状态指示，当其为1时表示处于使能状态
vld	数据有效标志指示信号，表示当前的data数据有效。注意，vld不仅指示了数据有效，而且还指示了有效的次数。时钟收到多少个vld=1，就表示有多少个数据有效
data	数据总线。输入一般名称为din，输出一般名称为dout。类似的信号还有addr、len、等
err	整个报文错误指示，在eop=1且vld=1有效时才有效
sop	报文起始指示信号，用于表示有效报文数据的第一个比特，当vld=1时此信号有效
eop	报文结束指示信号，用于表示有效报文数据的最后一个比特，当vld=1时此信号有效
rdy	模块准备好信号，用于模块之间控制数据发送速度。例如模块A发数据给模块B，则rdy信号由模块B产生，连到模块A(即对于B是输出信号，对于A是输入信号)； B要保证rdy产生正确，即此信号为1时，B一定能接收数据； A要保证仅在rdy=1时才发送数据

5.2 模块划分过程

下面通过一个实际项目，结合模块划分原则、模块划分端口规范来说明模块划分的过程，希望读者能够掌握模块划分的具体方法。

项目要求

要求实现将任意数据写到EEPROM AT93C46任意地址上，或者从AT93C46任意地址上读取数据。

5.2.1 模块划分思路

从项目要求中可知，项目的实现有两个关键点：一个是数据如何写到指定地址，又如何从指定地址读出；另一个是数据从哪里来？地址从哪里来？前者是实现AT93C46的读、写功能，即产生AT93C46读、写等操作的时序；后者是实现对AT93C46的控制功能，给出数据和地址。

根据以上分析，此项目有两个功能：读、写功能和控制功能，根据明德扬模块划分原则一：一个功能一个模块，整体模块划分结果如图 5-1 所示。

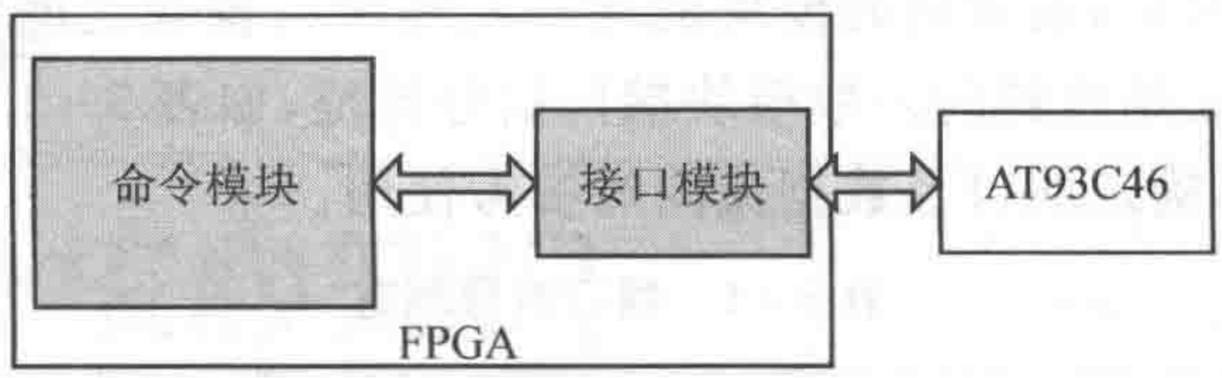


图 5-1　整体模块划分结果

接口模块用于完成对 AT93C46 的读、写操作，即完成读、写功能；命令模块负责给接口模块数据和地址，即完成控制功能。

很多初学者在设计过程中缺乏模块划分意识，喜欢一个模块完成所有功能。下面总结以上模块划分的好处，读者可以与一个模块完成所有功能的实现方式进行对比，体会模块划分的重要性。

① 模块各司其职。

接口模块专心负责接口时序的实现（接口时序在 datasheet 中可查到），根据命令模块传输过来的命令，产生相应操作的时序。至于为什么有这些命令，接口模块不关心。命令模块根据功能要求产生命令，只要将不同操作的命令送给接口模块即可，至于接口模块是如何实现 AT93C46 的操作时序则完全不关心。

② 方便调试。

由于不同功能有不同的需求，例如调试时，可能需要读和写特定的地址，或者重复写和读某一段地址，我们只需要更改调试模块的对应内容即可。但具体的时序由接口模块处理，并不需要改变，因此修改这些操作就变得非常简单了。

在根据项目功能要求划分模块之后，再分别考虑命令模块和接口模块是否需要继续细分。

1. 命令模块

命令模块的设计相对简单，只根据读、写需求产生读使能、写使能，以及对应的地址和数据等。因此不需要细分，只用一个模块即可。

2. 接口模块

首先回顾 3.2.2 小节的 SPI 接口项目中的 EEPROM 器件 AT93C46 的工作方式，分别是 EWEN 操作、写操作和读操作。一个操作就相当于一个功能，因此按原则一对接口模块进行细分，可以得到 EWEN 操作模块、WRITE 操作模块和 READ 操作模块。划分框图如图 5-2 所示。

继续分析 EWEN、WRITE、READ 这三个模块。图 5-3～图 5-5 是三个操作的时序图：

分析上面的三个时序图，可得：

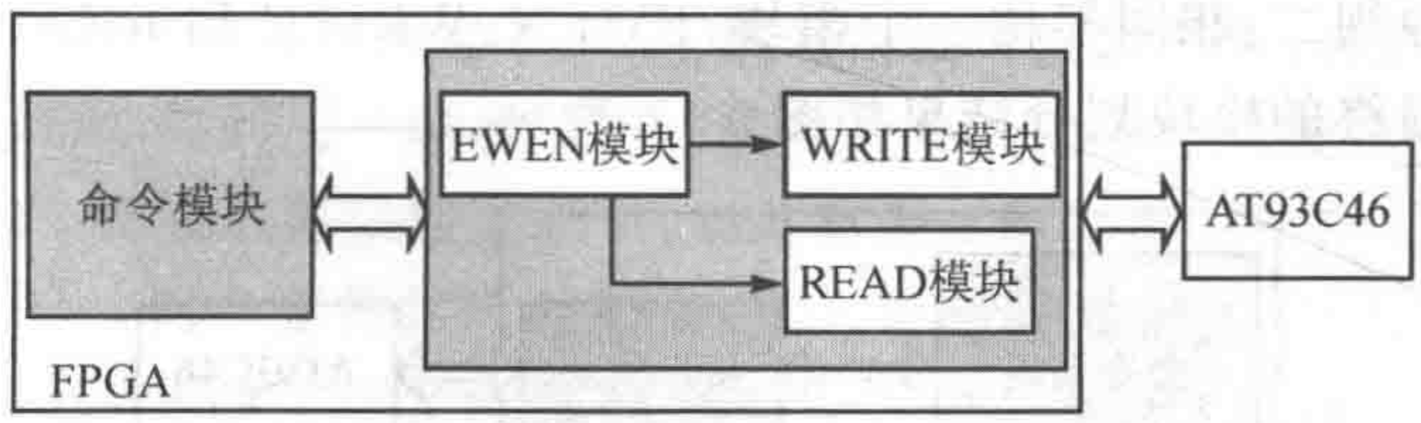


图 5－2　接口模块的模块划分框图

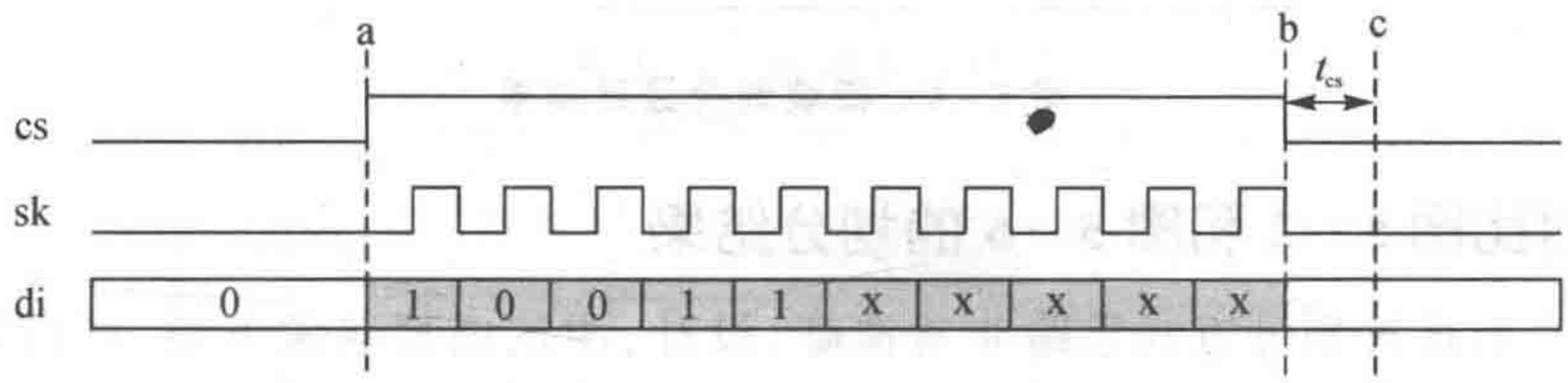


图 5－3　EWEN 时序

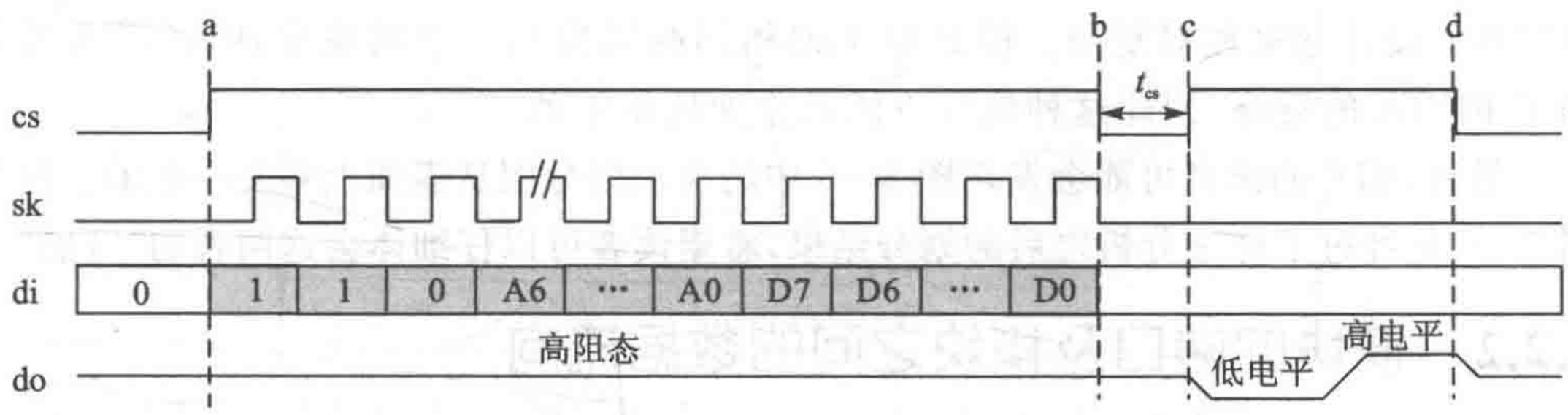


图 5－4　WRITE 时序

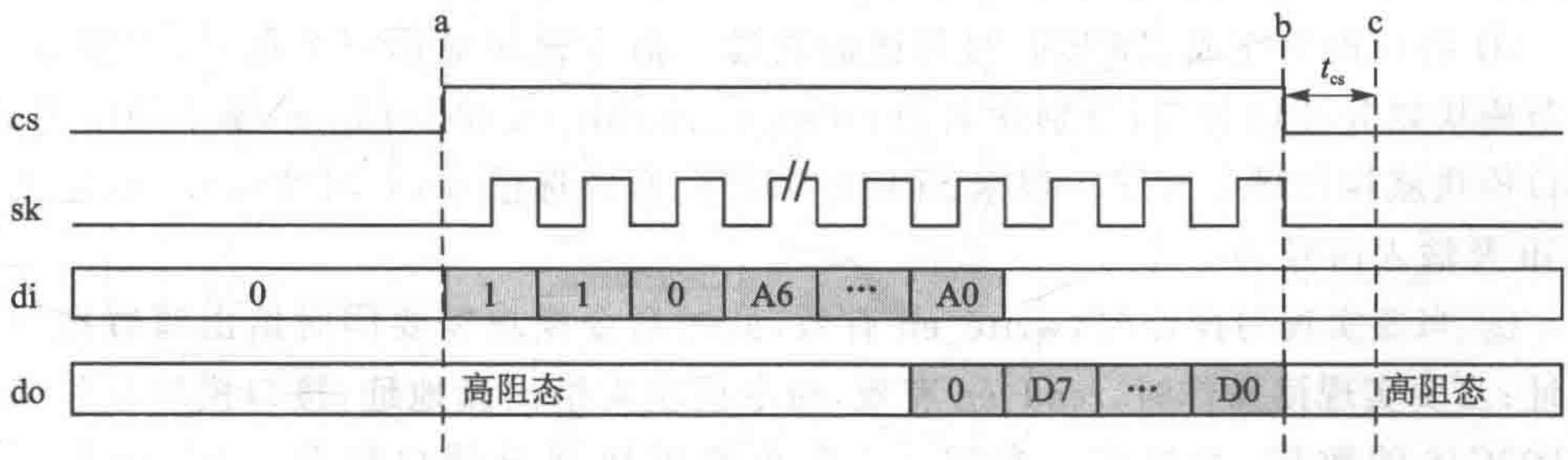


图 5－5　READ 时序

① EWEN 和 READ 的 a 时刻之前与 c 时刻之后，以及 WRITE 的 a 时刻之前与 d 时刻之后，接口输出是一致的；

② EWEN、WRITE 与 READ 的 a 时刻与 b 时刻之间，cs 均为高电平，sk 均按一定周期变化，接口输出规律是一致的；

③ EWEN、WRITE 与 READ 的 b 时刻与 c 时刻之间，接口输出一致；

④ WRITE 的 c 时刻与 d 时刻之间，有一个特殊的区域。

可以发现，三个模块的变化规律是高度相似的，即都可用同一模块架构来实现。

根据模块划分原则二：相同架构一个模块，EWEN、WRITE 和 READ 可用一个模块实现。此项目最终的模块划分结果如图 5－6 所示。

图 5－6　模块划分最终结果

3. 对比图 5－2 和图 5－6 的划分结果

图 5－2：这种划分方式思路非常清晰，设计、调试也都非常方便，可以逐个进行操作模块的设计、调试。但是，站在企业的角度看，这种设计方式占用了更多的资源。

图 5－6：要结合具体时序图来分析才能得出，这样实际上已经深入地考虑了模块实现，设计起来相对复杂。但是由于把相同的架构用一个模块实现，这样大大节约了 FPGA 的资源，因此这种设计方法是企业所推崇的。

另外，细心的读者可能会发现图 5－6 中的模块划分图从表面上看是一致的。但是图 5－6 是经过了详细分析之后的划分结果，希望读者可以仔细体会这两者的“区别”。

5.2.2　模块的端口及模块之间的数据流向

在完成模块划分之后，需要明确各个模块的端口及每个模块之间的数据流向。下面继续以 5.1 节模块划分原则的 AT93C46 项目来说明：

① 接口模块完成 EWEN、写和读的功能。命令模块输出三个信号，参照 5.1.2 小节模块划分端口规范，分别命名为 ewen_en、write_en 和 read_en，哪个使能有效，接口模块就执行哪个时序。根据 SPI 接口时序，有片选信号 cs、时钟信号 sk、输出信号 di 及输入信号 do。

② 当要实现写操作时，write_en 有效，此时命令模块需要同时给出写数据和写地址；当要实现读操作时，read_en 有效，命令模块需给出读地址，接口模块接收来自 AT93C46 的数据，并输出。参照 5.1.2 小节模块划分端口规范，写数据命名为 wdata，读、写地址为 addr，接口模块输出数据为 rdata 及数据有效为 rdata_vld（当 rdata_vld 有效时，表示数据有效，并且有多少次 rdata_vld 有效，就说明有多少个 rdata 数据有效）。

③ 由于接口模块做不到同时处理多个命令，或者在处理一个命令中间不能再有其他命令，因此命令模块要保证命令不同时有效，要保证处理过程中不产生其他命令。接口时序模块处理一个命令，需要较长的时间，而命令模块产生命令则非常迅速，基本上可以做到一个时钟发送一个命令。这需要一个机制，需要控制命令模块发送命令的速率，使得两个模块的速率一致。参照 5.1.2 小节模块划分端口规范，我们

可以用rdy机制来控制两者的速率。rdy信号由接口模块产生，当其为1时表示其可以接收命令并处理，当其为0时表示不能接收命令。命令模块发送命令时，检查rdy信号，如果其为1则表示可以发送命令，如果其为0则要等到其为1后再发送。接口模块要保证rdy=1时一定能接收并处理命令，否则错误责任在接口模块；命令模块要保证只能在rdy=1时才能发送命令，否则错误责任在命令模块。

综上所述，本例中两个模块之间的信号见表5-2。

表5-2 模块交互信号

信号名	说 明
ewen_en	EWEN使能命令
write_en	写使能命令
read_en	读使能命令
addr	读/写的地址
wdata	写数据
rdata	读到的数据
rdata_vld	读到的数据有效指示信号
rdy	接口模块准备好指示信号
cs、sk、di、do	SPI接口信号

结合表5-2中的信号，可以得出图5-7所示的端口数据流向。

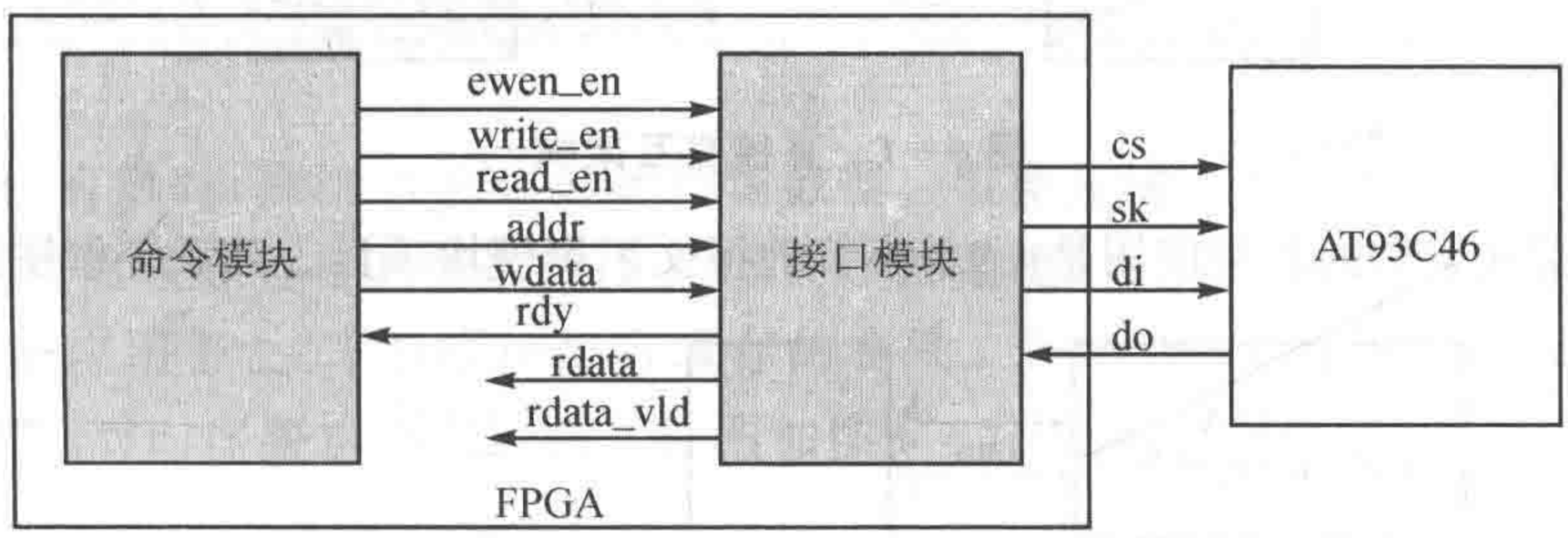


图5-7 确认端口数据流向

根据图5-7的模块划分，模块与模块之间的数据流向，即可根据第一篇所叙述的单个模块的设计方法完成功能设计。

第6章

模块划分常用架构及项目实践

上一章通过 AT93C46 项目，讲解了如何根据划分好的模块架构，确认模块的端口及数据流向。下面给出明德扬在 FPGA/ASIC 设计领域多年的项目经历，总结出来的五个常用的模块交互架构，读者在进行项目设计时，可结合明德扬模块划分原则，参照本章模块划分的常用架构进行模块划分。

6.1 直接交互架构

直接交互不需要模块 B 反馈，模块 A 直接发送数据。一般应用于下游模块处理速率高于上游模块发送速率的场景。直接交互架构如图 6-1 所示。

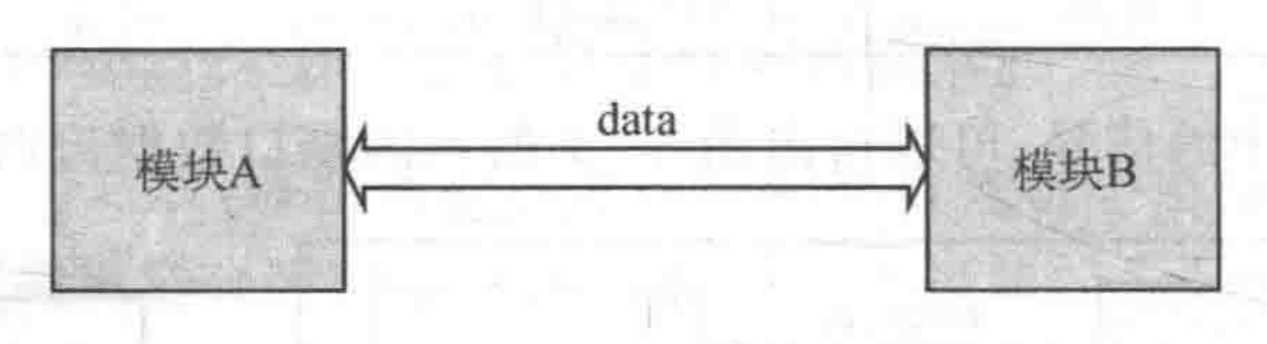


图 6-1　直接交互架构

直接交互架构的典型应用是 4.2.1 小节的报文汇聚模块项目。其系统框图见图 6-2。

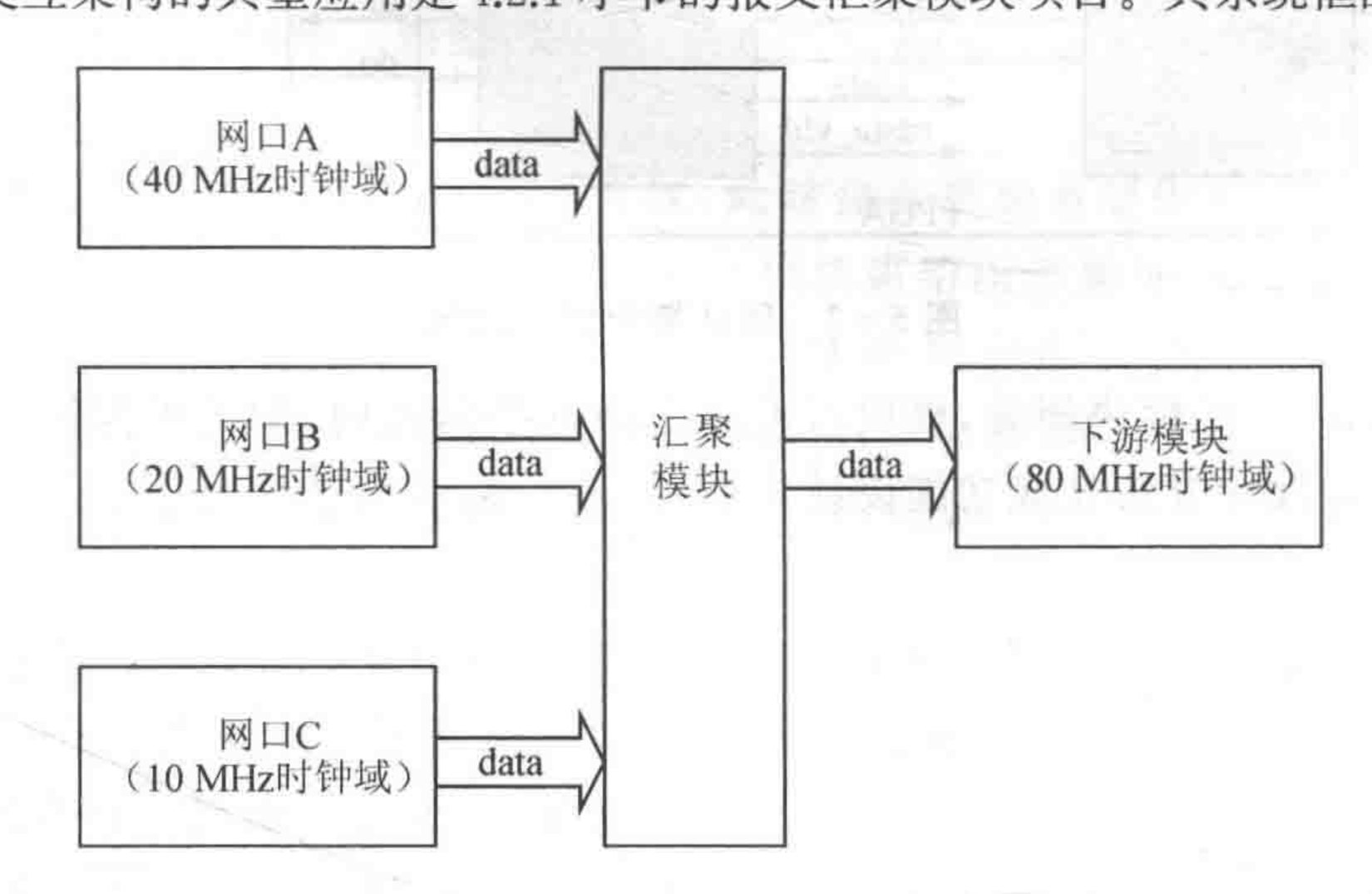


图 6-2　报文汇聚模块的模块划分框图

在此项目中，网口 A、B、C 作为上游模块其工作时钟分别是 40 MHz、20 MHz、10 MHz，下游模块的工作时钟是 80 MHz。由于下游模块的处理速率高于上游模块的总发送速率，所以即使三个网口同时全速发送数据，下游模块仍然可以完成对所有数据的处理。

因此，下游模块不需要通过反馈信号控制上游模块数据的发送，即采用直接交互架构即可。

6.2　无缓存 rdy 交互架构

当 rdy 为高电平时，表示模块 B 可以接受模块 A 的数据（或命令），开始传输数据，模块 A 每发送一个数据（或命令），模块 B 就接受并处理，处理完后模块 A 再发送下一个数据（或命令）。一般应用于上游模块速率高，下游模块速率低，并且上游模块有缓存的场景。框图如图 6－3 所示。

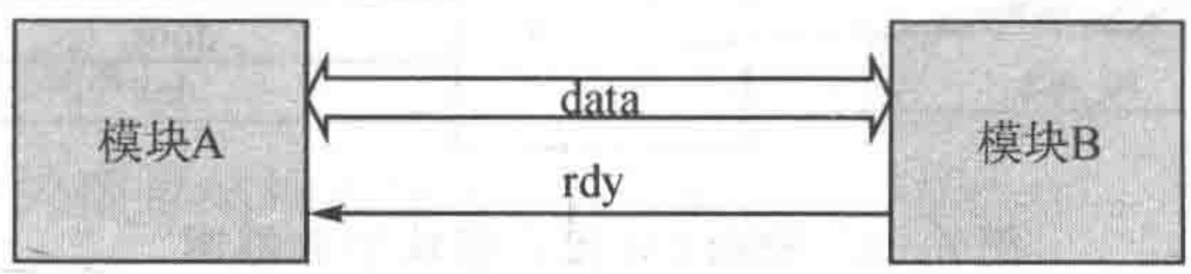


图 6－3　无缓存 rdy 交互架构

其实际应用参见 6.6.1 小节 AT93C46 的存储与读取项目。

6.3　有缓存 rdy 交互架构

当 rdy 有效时，模块 A 可以开始传输数据。数据先输入模块 B 的 FIFO，然后模块 A 和模块 B 按照各自时钟进行读、写 FIFO 的数据，可以解决读、写速率不匹配的问题。一般应用于上、下游模块都有缓存的场景。采用有缓存 rdy 交互架构的优点是控制简单，对时序要求不太严格。框图如图 6－4 所示。

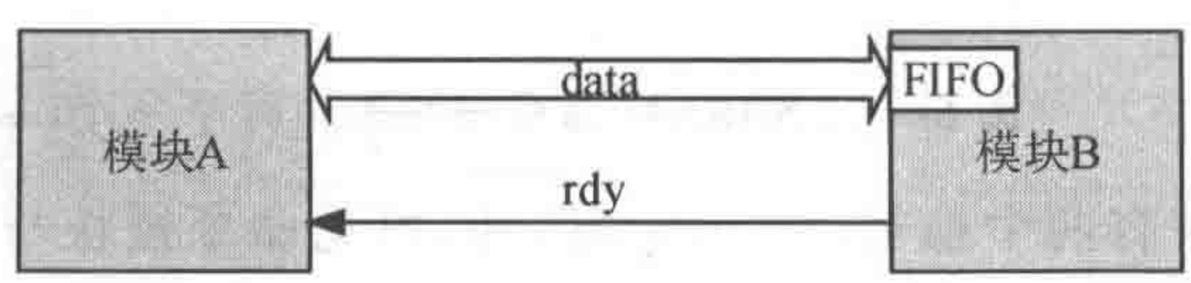


图 6－4　有缓存 rdy 交互架构

其实际应用参见 6.6.1 小节 AT93C46 的存储与读取项目。

6.4　请求-应答交互架构

当模块 A 请求发送，模块 B 可以开始接收来自模块 A 的数据时，返回给模块 A 应答信号，当模块 A 收到应答信号时就可以传输数据了。一般应用于上游模块等待

响应后才能连续发送一段数据的场景。框图如图 6－5 所示。

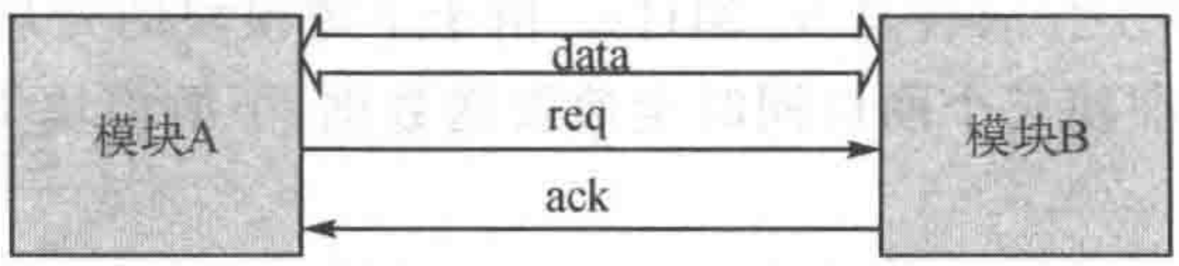


图 6－5　请求-应答交互架构

此架构的一个典型应用是 3.2.3 小节 SDRAM 接口项目。其模块划分框图如图 6－6 所示。

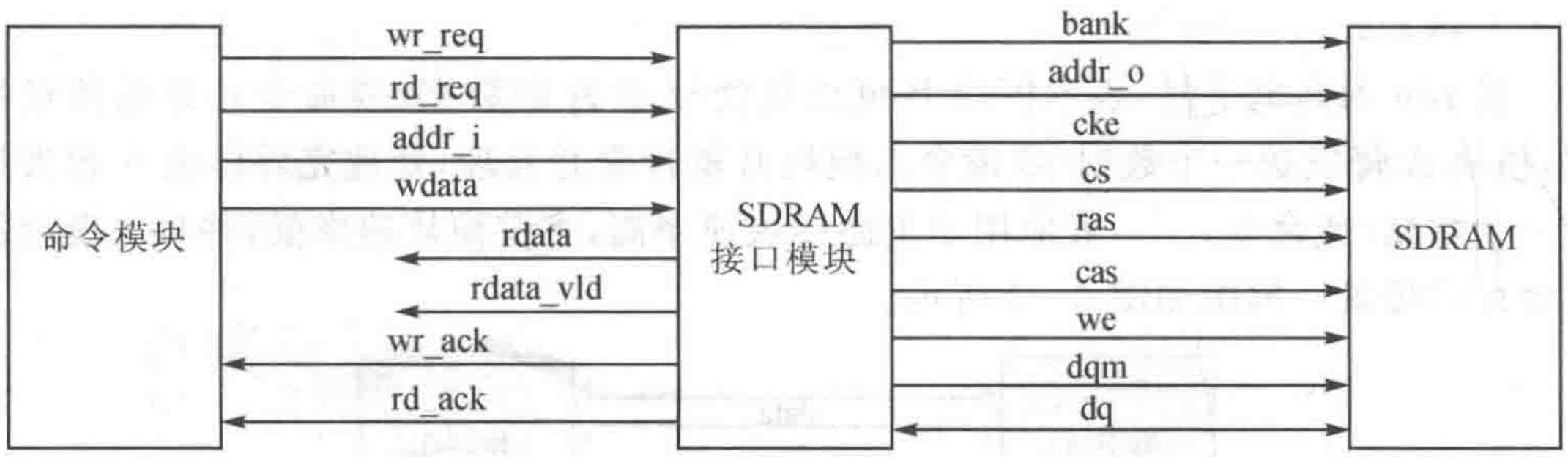


图 6－6　SDRAM 接口模块划分框图

在 3.2.3 小节中，只设计了 SDRAM 的接口模块，可以按照 SDRAM 的时序完成读/写操作，而读/写指令、地址以及需要写入的数据等，只作为来自命令模块的输入信号，并不考虑它们是如何产生的。下面回顾一下命令模块的写时序，如图 6－7 所示。

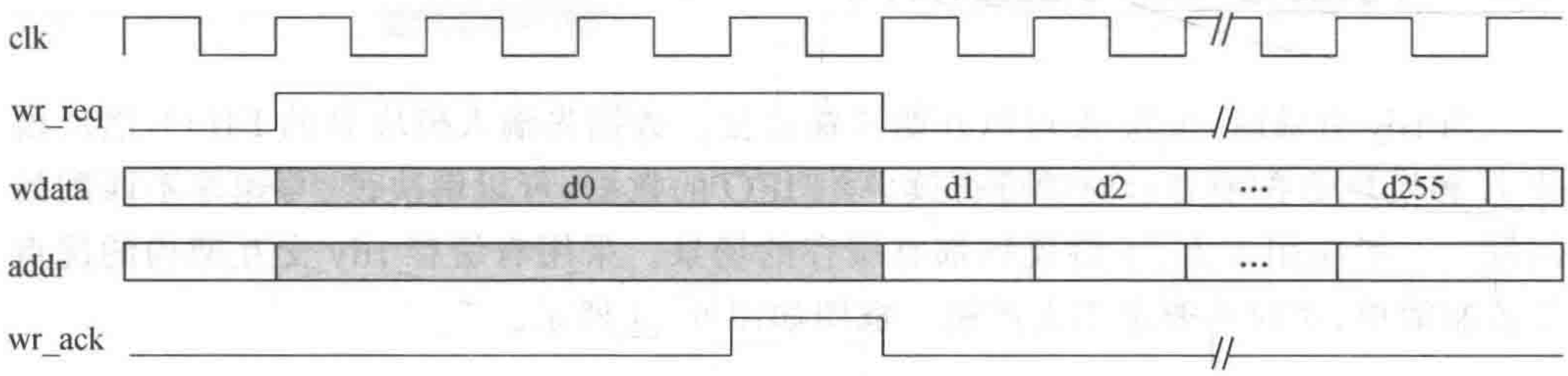


图 6－7　SDRAM 命令模块的写时序

读者会发现其中有两个信号——req 与 ack，可见命令模块与 SDRAM 接口模块的交互方式是请求应答交互架构。为什么此处要使用请求-应答交互架构？下面给出分析过程。

假设命令模块给接口模块一个写命令，而此时 SDRAM 正在执行自动刷新操作，则必须要等到自动刷新完成，SDRAM 才可以响应写命令，但是写命令一般只持续一个时钟周期，等到自动刷新完成，写命令已经失效了。

为了解决此问题，我们采用了请求-应答交互架构。命令模块给接口模块一个写请求，接口模块在空闲时会给命令模块回复一个写应答信号，命令模块在收到应答信号后，撤销写请求。

6.5　外设交互架构

1. 不需配置交互架构

不需配置交互架构的图形结构如图 6－8 所示。

图 6－8　不需配置交互架构

5.2 节中提到的 AT93C46 项目就是用此交互架构，不需要配置寄存器即可正常工作的外设，这里不再赘述。

2. 需配置交互架构

需配置交互架构的图形结构如图 6－9 所示。

图 6－9　需配置交互架构

对于需要配置寄存器才能正常工作的外设，或者可以通过配置寄存器实现多种功能的外设，采用此架构。整个架构包括接口模块和配置模块，其中配置模块内部还有一个配置表，要实现多种功能，只需要更改配置表即可。

配置表的具体应用及详细说明请见 6.6.3 小节。

6.6　模块划分项目实践

6.6.1　AT93C46 的存储与读取项目

题　目

AT93C46 是一款 EEPROM 芯片，现要求能够连续存储或读取 AT93C46，即写使能与读使能可能连续给出，AT93C46 能够响应这些使能，实现读、写功能。另外，响应写使能的优先级高于响应读使能。请读者考虑实现该功能需要哪些模块？进行模块划分并给出端口定义，不需要写出代码。

1. 模块划分

根据外设交互架构，由于 AT93C46 的使用不需要配置寄存器，所以使用不需配置交互架构，如图 6－10 所示。

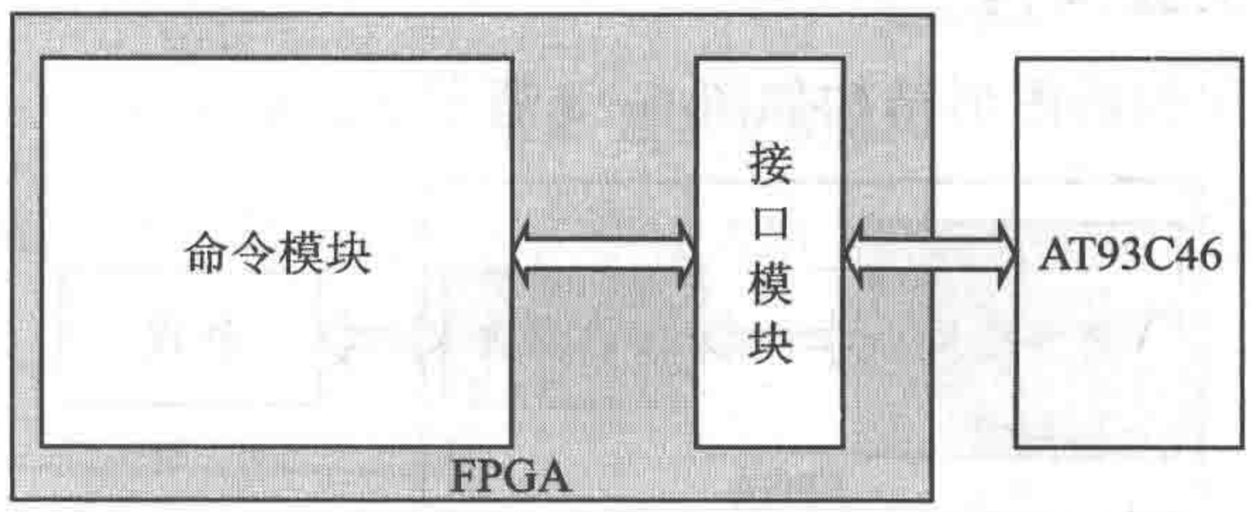


图 6－10　初步模块划分

下面分别考虑命令模块和接口模块是否需要细分。

(1) 命令模块

分析一下命令模块需要实现的功能：

首先，根据读、写需求，产生读、写使能。根据模块划分原则一，一个功能一个模块，将此功能单独成一个模块——读写命令模块。进一步分析，项目要求，读、写使能可以同时，或者连续产生，所以此模块的数据线、地址线不能共用，为简化设计，可以进一步将读写命令模块划分为存储模块（完成产生写使能功能）和读取模块（完成产生读使能功能）。

其次，读、写需求，可能会连续出现或同时出现，但是接口模块一次只能响应一个操作。由于每一个读使能、写使能的有效时间都只需维续一个时钟周期，但是 AT93C46 完成一个读操作或写操作，却需要成百上千个时钟周期。若 AT93C46 正在执行读、写操作时，又接收到一个新的读、写使能，就没有办法响应新的读、写使能。为了解决此问题，需要将后面接收到的读、写使能缓存起来，等当前的读、写操作完成后，再响应新的读、写使能。当同时接收到读、写使能，或者缓存了多个读、写操作时，则需要判断所执行操作的先后顺序，即实现对读、写使能的调度。

结合以上，并参考有缓存 rdy 交互架构，可以将命令模块细分为三个模块，即存储模块、读取模块、调度模块。存储模块负责产生写使能；读取模块负责产生读使能；调度模块负责缓存读、写使能，并将读、写使能按照项目需求发送给接口模块，以完成对芯片的读、写。

(2) 接口模块

接口模块的模块划分参考 5.2 节，不需要进一步细分。

在这里另外再说明一下，命令模块与接口模块之间使用的是无缓存 rdy 交互架构——命令模块需要判断接口模块的 rdy 信号，如是有效，则发送读、写等请求过来；否则，不能产生读、写请求。

综上，此项目模块划分框图如图 6－11 所示。

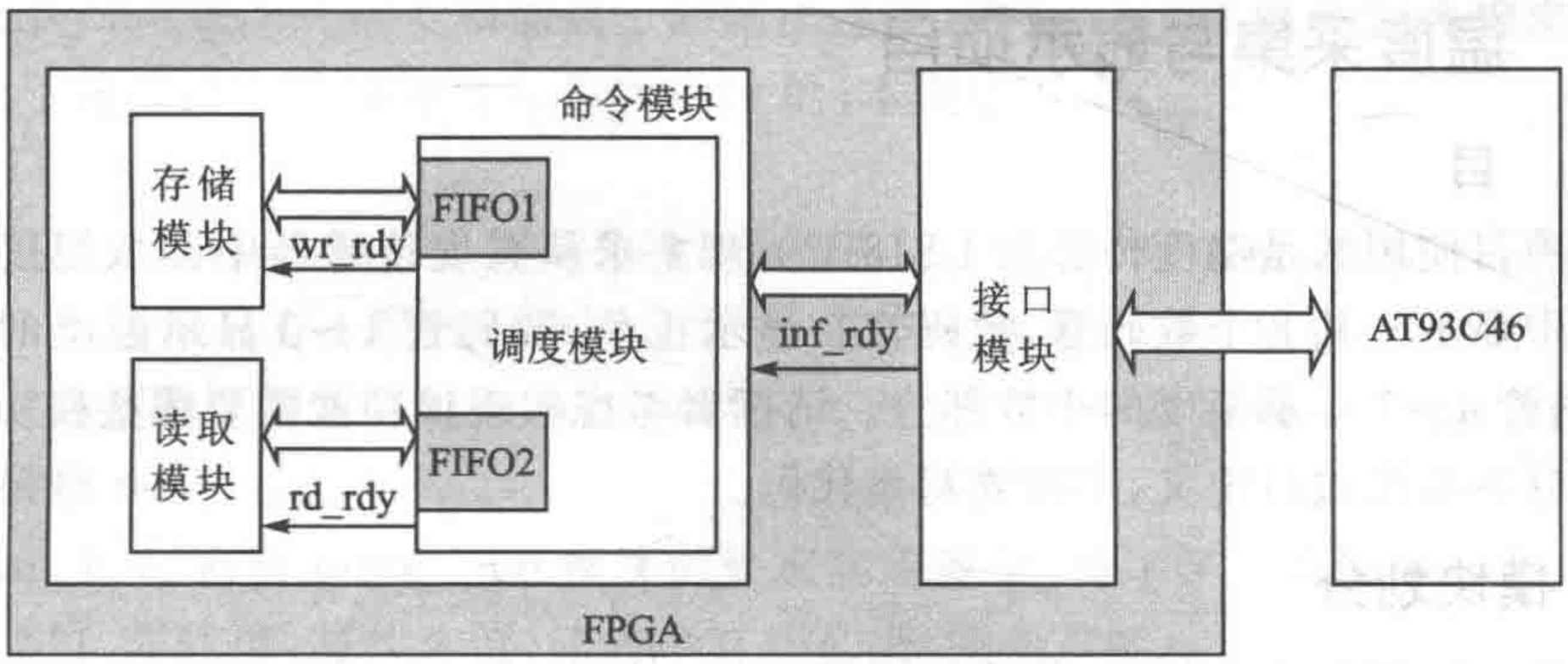


图 6－11　模块划分框图

存储模块将写使能、写数据及写地址，输出给调度模块。只要 wr_rdy 有效，就可以连续给写使能。下游模块如何实现写使能的缓存，写时序如何产生，存储模块不关心。

读取模块将读使能、读地址，输出给调度模块。只要 rd_rdy 有效，就可以连续给读使能。下游模块如何实现读使能的缓存，读时序如何产生，读取模块不关心。

调度模块对收到的读、写使能进行缓存和调度（先响应哪个读/写操作，就将对应操作使能发送给接口模块）。调度模块会检查 FIFO1 及 FIFO2，若存满则对应将 wr_rdy 或rd_rdy 拉低；也会检查接口模块发送来的 inf_rdy 的状态，如果 FIFO 非空且 inf_rdy 有效，则读取 FIFO 的使能送给接口模块处理（FIFO0 的优先级更高，若 FIFO0 与 FIFO1 都有数据，且 inf_rdy 有效，则先读取 FIFO0，产生写使能给接口模块）。

接口模块，负责与 AT93C46 的通信，根据上游模块发送来的使能信号，产生相应的接口时序，完成相应的操作。

2. 模块的端口及模块之间数据流向

模块划分好后，需要进一步确认模块的端口及模块之间的数据流向。可参考 5.1.2 小节模块划分端口规范。模块划分最终结果如图 6－12 所示。

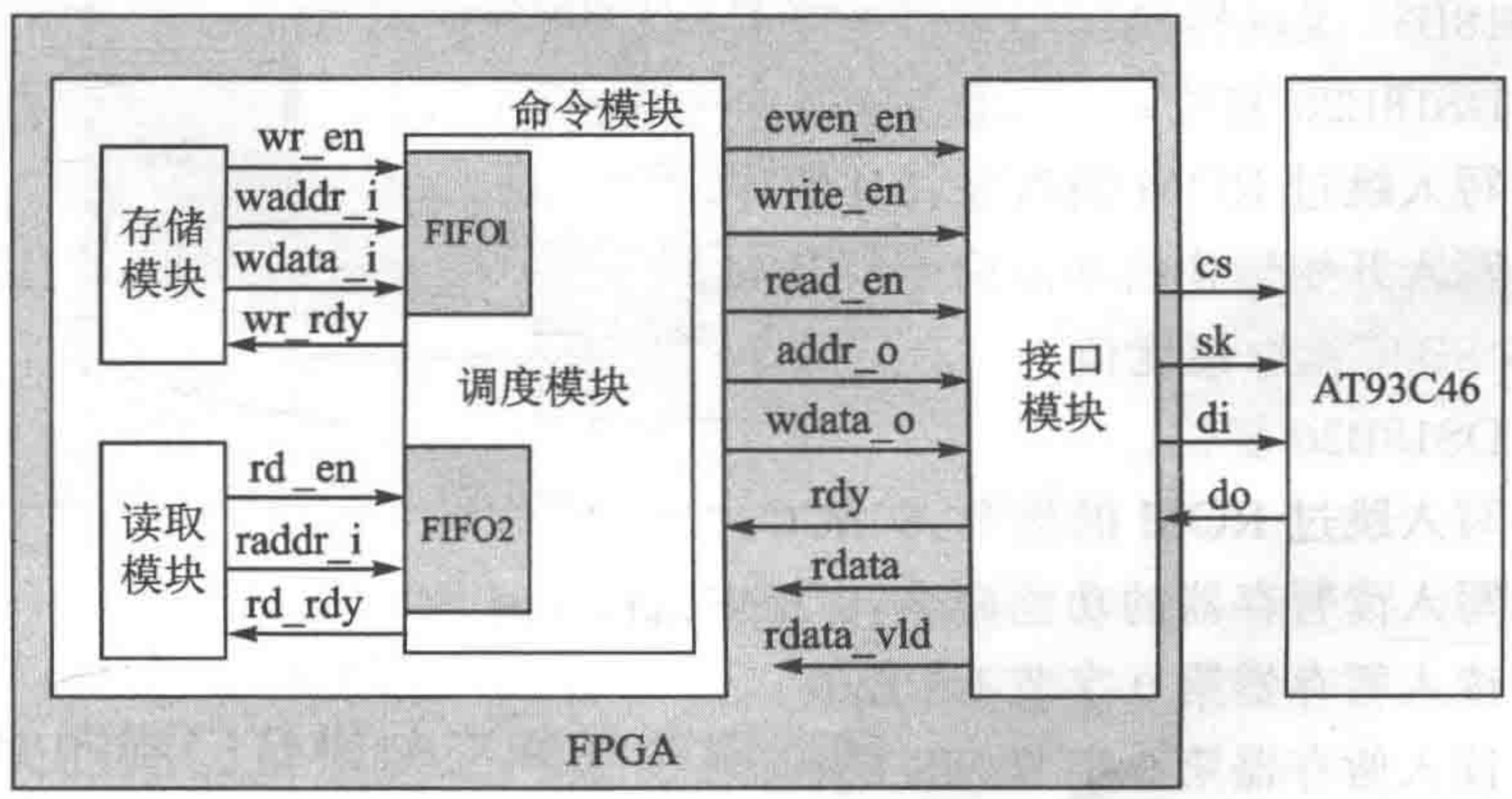


图 6－12　模块划分最终框图

6.6.2 温度采集与显示项目

题 目

本项目使用的是温度传感器 DS18B20，现要求从温度传感器中读取温度值并在数码管中显示。共 8 个数码管，数码管 0 显示正负，数码管 1～3 显示温度的整数部分，数码管 4～7 显示温度的小数部分。请读者考虑实现该功能需要哪些模块？进行模块划分并给出端口定义，不需要写出代码。

1. 模块划分

本项目使用了两个外设，均为不需配置寄存器即可正常工作的外设，参考不需配置交互架构，本项目的模块划分如图 6－13 所示。

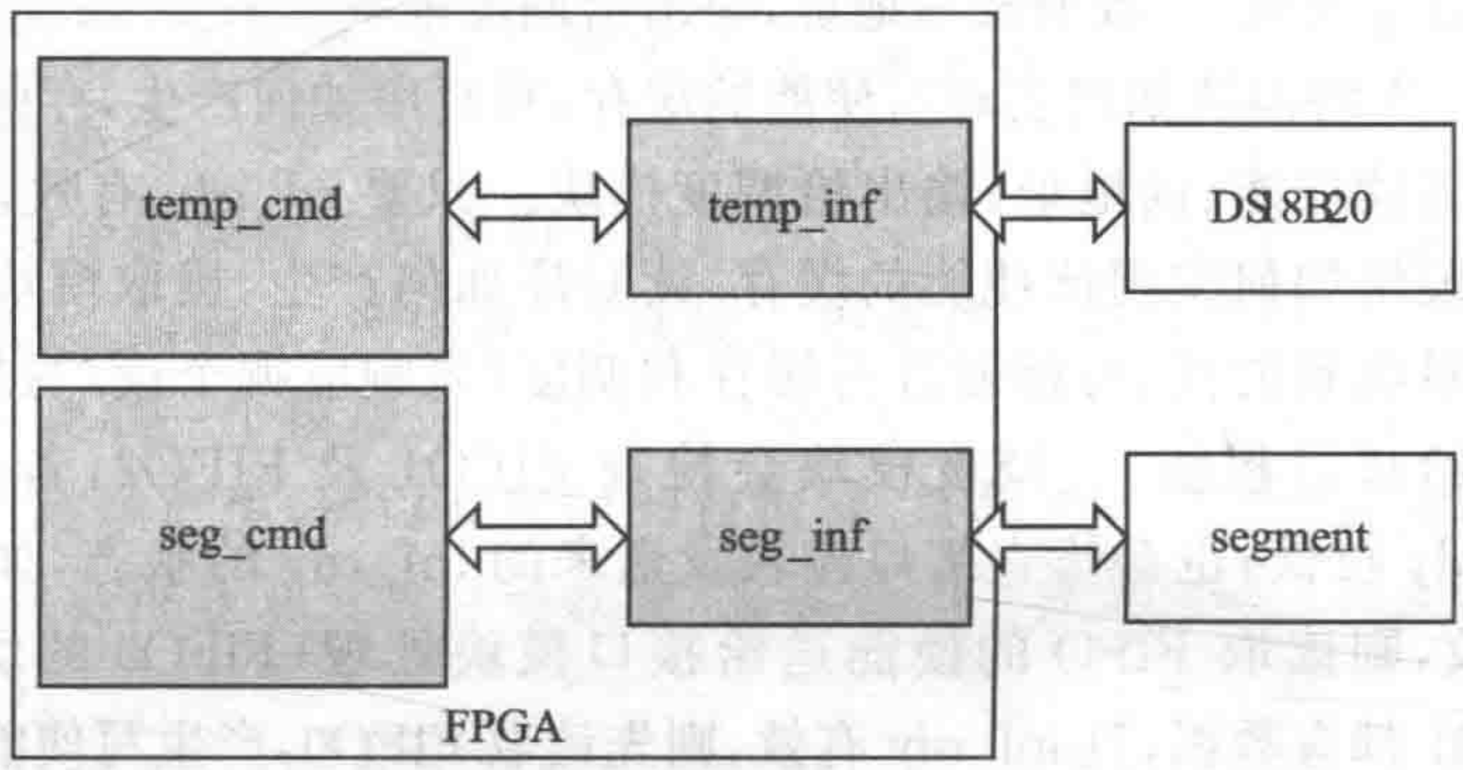


图 6－13 初步模块划分

下面进一步分析以上模块，看是否需要再进行细分。

(1) temp_cmd

查阅 DS18B20 的 datasheet，可以发现，要读取温度值，必须经过复位，发送一系列命令，最后才能读取到温度值。下面给出 DS18B20 的温度转换与读取温度值过程。

① DS18B20 温度转换：

- DS18B20 复位；
- 写入跳过 ROM 的指令：8'hCC；
- 写入开始转换的功能命令：8'h44。

② DS18B20 读取温度值：

- DS18B20 复位；
- 写入跳过 ROM 的指令：8'hCC；
- 写入读暂存器的功能命令：8'hBE；
- 读入暂存器第 0 字节 LS Byte；
- 读入暂存器第 0 字节 MS Byte；
- DS18B20 复位，表示读取暂存器结束。

temp_cmd 模块只需要根据以上的操作过程，给 temp_inf 模块发送相应的指令即可，至于这些命令是如何写入 DS18B20 的，本模块不关心。

(2) temp_inf

上游模块 temp_cmd 给出的命令、数据都是以字节为单位的，而 temp_inf 与 DS18B20 的连接只有 1 根线。每写 1 字节，都要经过 8 次写逻辑，每读 1 字节，也要经过 8 次读逻辑。因此可以把 temp_inf 模块细分为字节接口模块 temp_byte 与比特接口模块 temp_bit。

temp_byte 根据 temp_cmd 模块的要求实现复位、写 1 字节或读 1 字节，至于什么时候复位、写或读、写什么值、读的值有什么用，完全不关心。

temp_bit 只根据 temp_byte 的命令，去复位、读 1 比特或写 1 比特，即处理 DS18B20 的时序。

(3) seg_cmd

seg_cmd 模块将要显示的数据送给 seg_inf 即可完成温度显示，也就是说此模块不需要产生其他控制命令。又由于温度值是由 temp_byte 输出，并由 temp_cmd 转换成能够在 8 个数码管上显示的数据格式，temp_cmd 模块直接把该数据送到 seg_inf 即可，所以不需要 seg_cmd 模块。

(4) seg_inf

seg_inf 模块要完成的是数码管动态扫描与译码功能，8 个数码管将按一定的扫描频率依次点亮，并显示对应的数值。

综上，我们删除 seg_cmd 模块，而 temp_cmd 模块既完成读取数据的转换，又给 temp_inf 模块发送命令与写数据，因此更名为 control 模块。模块划分结果如图 6－14 所示。

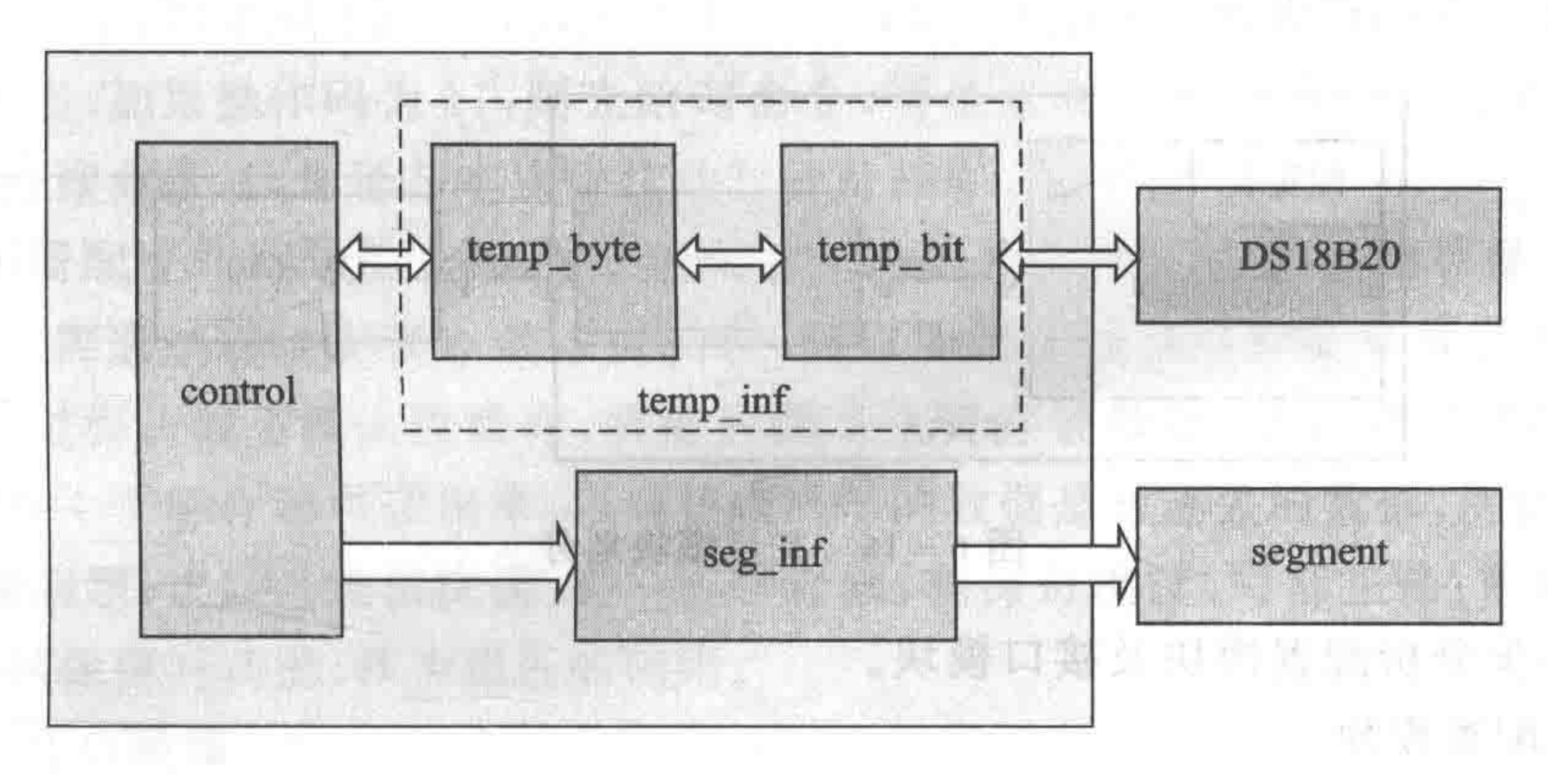


图 6－14　模块划分框图

2. 模块的端口及模块之间的数据流向

模块划分好后，需要进一步确认模块的端口及模块之间的数据流向。可参考

5.1.2小节模块划分端口规范。模块划分最终结果如图 6－15 所示。

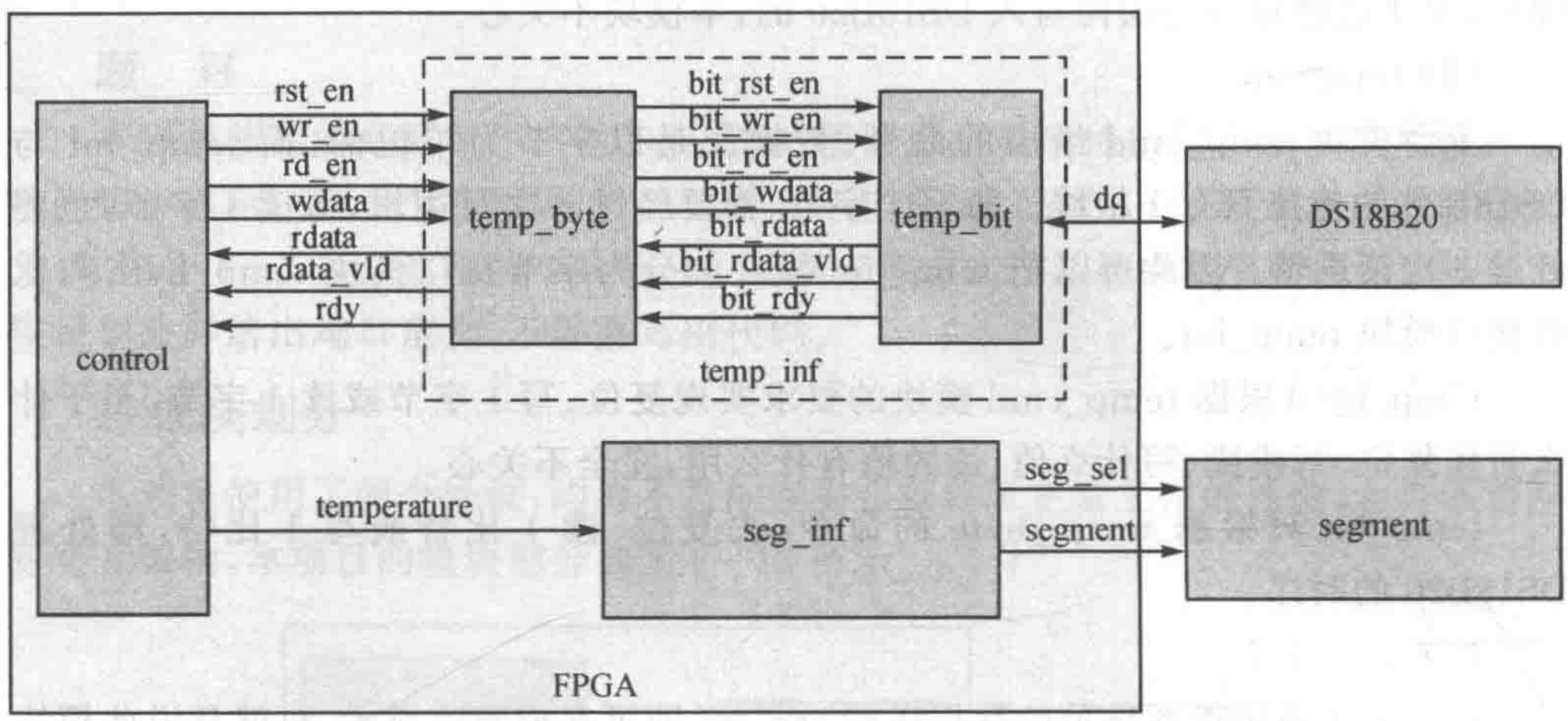


图 6－15　模块划分最终框图

6.6.3　OV7670 图像采集项目

题　目

实现摄像头 OV7670 的配置及完成图像的采集。请读者考虑实现该功能需要哪些模块？进行模块划分并给出端口定义，不需要写出代码。

1. 模块划分

由于 OV7670 正常工作前需要配置多个寄存器，故根据需配置外设交互架构，模块划分如图 6－16 所示。

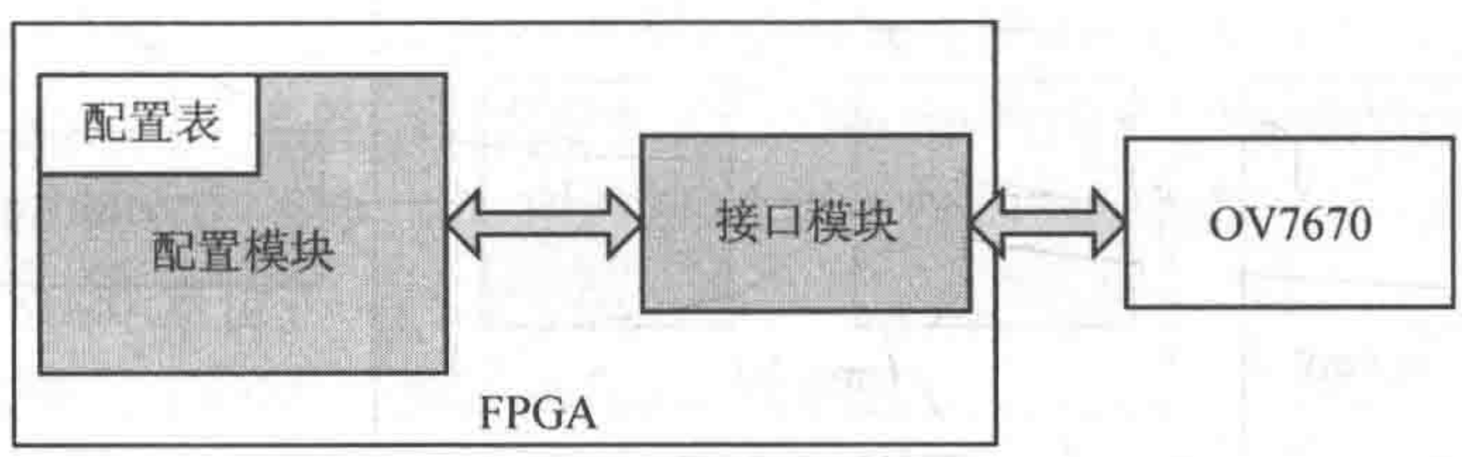


图 6－16　初步模块划分

进一步分析配置模块及接口模块。

(1) 配置模块

从 OV7670 的 datasheet 可知，完成某些必要寄存器的配置，OV7670 才能正常工作，而配置不同的寄存器，又能够实现不同的功能。配置模块根据配置要求产生读/写命令、数据与地址给接口模块。配置表里包含操作码、所有寄存器地址及寄存器的配置值。

下面详细介绍配置表，以下是 OV7670 的部分配置表：

```
1  parameter        REG_NUM =        164;
2
3  always@( * ) begin
4      case(reg_cnt)
5          0 : add_wdata = {2'b11,16'h1204};
6          1 : add_wdata = {2'b11,16'h40d0};
7          2 : add_wdata = {2'b11,16'h3a04};
8          3 : add_wdata = {2'b11,16'h3dc8};
9          4 : add_wdata = {2'b11,16'h1e31};
10         5 : add_wdata = {2'b11,16'h6b00};
11         ......
12         163 : add_wdata = {2'b11,16'h0903};
13         164 : add_wdata = {2'b11,16'h3b42};
14         default : add_wdata = 0;
15     endcase
16 end
```

① REG_NUM 表示此外设寄存器总数。

② 配置表存放的格式为“操作码”＋“地址”＋“配置值”，比如：add_wdata = {2'b11,16'h1204}，其操作码是 2'b11，地址是 8'h12，配置值是 8'h04。

③ 操作码为 2 bit：00 表示空操作；01 表示只读；10 表示只写；11 表示先写后读。

④ 配置模块读取配置表，根据操作码来发送命令给接口模块。如果操作码为 00，则跳过此命令；如果操作码为 01，则产生一个读命令给接口模块，读地址是配置表给出的地址；如果操作码为 10，则产生一个写命令给接口模块，写地址和写数据由配置表给出；如果操作码为 11，则先给写命令，写地址和写数据由配置表给出，执行完后，再给读命令，读地址与本次操作的写地址相同。

采用需配置外设交互架构，在配置具有多功能外设的寄存器时，能够很方便地进行调试，仅需通过修改配置表就可以实现，所以明德扬强烈推荐此类外设使用该架构。调试过程一般是确认读操作，看能否读 1 个寄存器数据，从而确保读操作没问题；然后写 1 个寄存器再读出来，以确认读出来的数据是写进去的数据，从而确保读、写操作没问题；之后连续依次读、写一组寄存器，确保每次读、写都正确；最后把读功能取消，只保留写功能，作为最后的代码。

(2) 接口模块

接口模块需要完成两个功能：一是完成 OV7670 的配置，即根据配置模块的读、写命令及数据产生 SCCB 操作时序（查阅 OV7670 的 datasheet，OV7670 的配置接口是 SCCB）；二是完成 OV7670 的图像采集。所以，根据模块划分原则一，一个功能一个模块，接口模块可以划分为 SCCB 接口模块和采集模块。

综上，模块划分结果如图 6－17 所示。

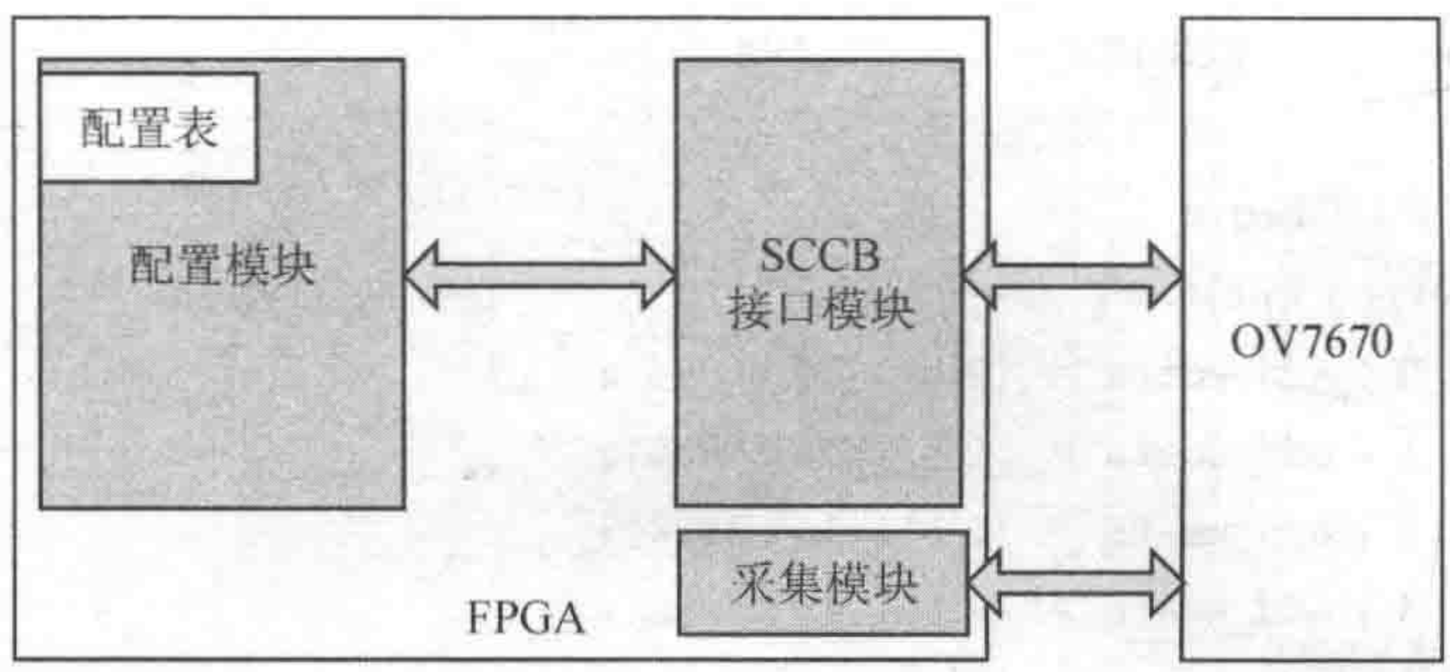


图 6－17　模块划分框图

2. 模块的端口及模块之间的数据流向

模块划分好后，需要进一步确认模块的端口及模块之间的数据流向。可参考 5.1.2 小节模块划分端口规范。模块划分最终结果如图 6－18 所示。

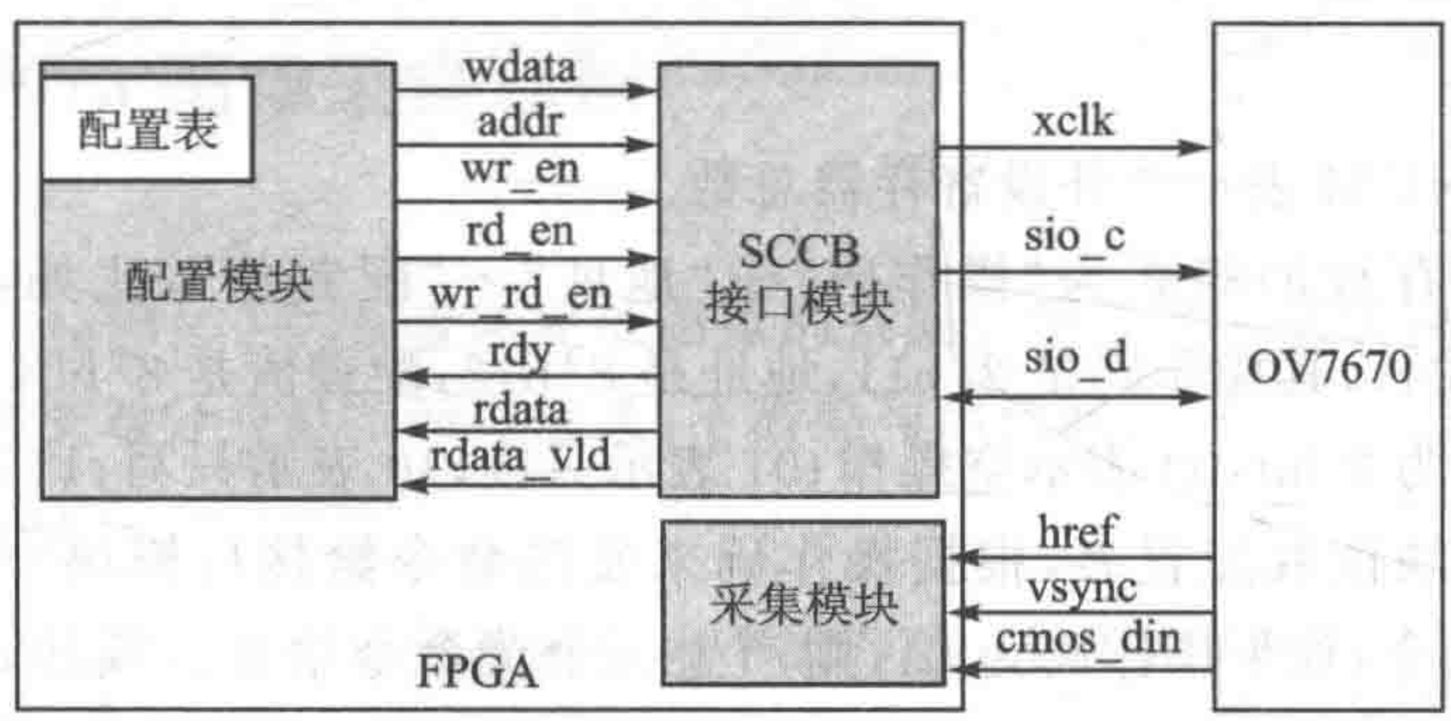


图 6－18　模块划分最终框图

第三篇　项目实践

第 7 章　基于 FPGA 的温度监控系统
第 8 章　基于 FPGA 的图像边缘检测系统

第 7 章

基于 FPGA 的温度监控系统

7.1 背景介绍

温度是生活中最基本的环境参数。温度的监测与控制，对于生物的生存与生长以及工业生产都具有重要的意义。温度传感器的应用涉及机械制造、工业过程控制、汽车电子产品、消费电子产品和专用设备等各个方面。

由于单片机设计的控制器生产成本较低，所以过去人们通常使用这种方法来设计数字温度监控系统。这种设计方法的缺点是它并不能完成很复杂的功能，只能进行一些简单的运算和控制，并且设计的控制器性能也很一般。因此，在这种情况下，研究人员要想进一步提升技术，就不得不冒着牺牲机器性能的风险来扩展外部硬件电路，显然用这种传统的方法设计数字温度监控系统，既具有局限性又具有一定的难度，所以要寻找一种新的设计方法来替代这种传统的研究方法。

本项目旨在分析和设计数字化温度测控系统。本系统采用 FPGA 实现一个数字温度监控系统，采用 Verilog HDL 作为开发语言，实现对温度的检测与报警。

7.2 项目要求

本项目实现图 7-1 中的温度监控系统。该系统根据 PC 发送过来的指令（通过 UART 接口），完成温度传感器的温度采集、转换、显示及其他功能。采集的温度经过转换在数码管上显示，如果温度超过设置范围，则报警（蜂鸣器响）。

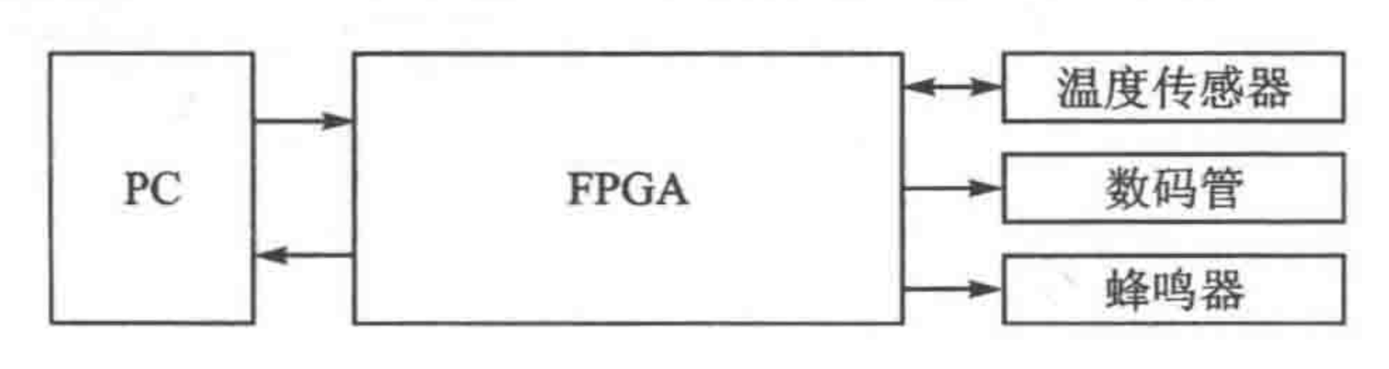


图 7-1 项目框图

1. 指令格式

PC 端（串口调试助手）通过发送指令给 FPGA，温度监控系统根据不同的指令完

成不同的功能。具体指令定义表见表7-1。

PC按以下格式发送指令:1字节的类型(type)+1字节的数据(data)。

表7-1 指令定义表

指令(十六进制)	功能定义
80XX	复位温度传感器
81XX	将XX值写到温度传感器
82XX	从温度传感器的温度寄存器(LSB)读到1字节数据,并保存在FPGA温度低位寄存器DS_TEMP_LSB_REG
83XX	从温度传感器的温度寄存器(MSB)读到1字节数据,并保存在FPGA温度高位寄存器DS_TEMP_MSB_REG
84XX	从传感器读到1字节数据,并保存在FPGA读通用寄存器(读除了温度寄存器外的其他寄存器)DS_TEMP_RDATA_REG
01XX	设置数码管开关。XX最低位有效: 0:关闭数码管显示 1:开启数码管显示 其他位无效
02XX	设置温度报警范围的下限,XX一共有8位,其中最高位是符号位,0表示正,1表示负。低7位是数值
03XX	设置温度报警范围的上限,XX一共有8位,其中最高位是符号位,0表示正,1表示负。低7位是数值
04XX	利用DS_TEMP_LSB_REG和DS_TEMP_MSB_REG计算出十进制数值,结果保存到TEMP_US寄存器中
05XX	设置温度报警功能。XX最低位有效: 0:关闭温度报警功能 1:开启温度报警功能 其他:无效
06XX	读寄存器TEMP_US[31:24],返回给PC
07XX	读寄存器TEMP_US[23:16],返回给PC
08XX	读寄存器TEMP_US[15: 8],返回给PC
09XX	读寄存器TEMP_US[7: 0],返回给PC
0aXX	读寄存器DS_TEMP_LSB_REG,返回给PC
0bXX	读寄存器DS_TEMP_MSB_REG,返回给PC
0cXX	读寄存器DS_TEMP_RDATA_REG,返回PC
其他	无效,可用做延时

注:1 每次发送必须是两个字节一起发送,否则会出错;

2 表中的XX,如果是发送数据,则XX是发送的内容;否则,XX表示任何值。

2. PC 端(串口调试助手)设置

串口调试助手按以下参数设置：

波特率为 9 600 kbit/s；

无校验位；

数据位为 8 位；

停止位为 1 位；

使用十六进制发送和显示数据。

3. 读取温度传感器的值(PC 发送指令顺序)

首先回顾一下，6.6.2 小节温度采集与显示项目，DS18B20 的温度转换与读取温度值的过程。

(1)DS18B20 温度转换

① DS18B20 复位；

② 写入跳过 ROM 的指令：8'hCC；

③ 写入开始转换的功能命令：8'h44。

(2) DS18B20 读取温度值

① DS18B20 复位；

② 写入跳过 ROM 的指令：8'hCC；

③ 写入读暂存器的功能命令：8'hBE；

④ 读入暂存器第 0 字节 LS Byte；

⑤ 读入暂存器第 0 字节 MS Byte；

⑥ DS18B20 复位，表示读取暂存器结束。

根据指令功能定义，要完成以上操作，PC 端(串口调试助手)需要依次发送命令：16'h8000、16'h81cc、16'h8144、16'h8000、16'h81CC、16'h81BE、16'h8200、16'h8300。此时温度值保存到 FPGA 的 DS_TEMP_LSB_REG 和 DS_TEMP_MSB_REG 寄存器。

4. 温度转换(PC 发送指令)

控制 FPGA，将温度传感器定义的格式，转换成 BCD 码。

发送命令：16'h0400。此时 FPGA 将根据 DS_TEMP_LSB_REG 和 DS_TEMP_MSB_REG 寄存器转 BCD 码，并保存到 TEMP_US 中。

5. PC 读取转换后的温度值(PC 发送指令，通过串口调试助手显示)

读取 TEMP_US 的数据。

依次发送命令：16'h0600、16'h0700、16'h0800、16'h0900。

6. 数码管显示温度(PC 发送指令)

数码管显示可控制。

关闭数码管显示，发送命令：16'h0100；打开显示，发送命令：16'h0101。

读取的温度值在数码管上显示。一共使用 8 个数码管，从左至右分别是数码管 0～7：数码管 0 是正负指示，用"－"表示负数，用"0"表示正数；数码管 1～3 是温度值的整数部分；数码管 4～7 是温度值的小数部分。（显示的是 TEMP_US 的值）

7. 蜂鸣器报警(PC 发送指令)

控制蜂鸣器开关。

关闭蜂鸣器报警，发送命令：16'h0500；打开蜂鸣器报警，发送命令：16'h0501。

FPGA 内部有两个寄存器：temp_cmp_h 和 temp_cmp_l，分别是温度上限值和下限值。如果转换后的温度值 TEMP_US 大于 temp_cmp_h 或者小于 temp_cmp_l，则蜂鸣器报警。

温度值的上限和下限也可以修改。复位后上下限都为 0。

发送命令：16'h02XX，下限值设置为 XX；发送命令：16'h03XX，上限值设置为 XX。

7.3　基本原理

7.3.1　PC 端(串口调试助手)

串口调试助手与 FPGA 通信接口为 UART。UART 接口在前文已有详细描述，其实现原理参见 2.2.2 小节 UART 接口项目，这里不再赘述。

7.3.2　温度传感器 DS18B20

本项目使用了温度传感器的温度转换功能与读取温度功能，其操作步骤在 PC 发送指令顺序处已有说明。要注意的是，所有 PC 发送的指令都是以字节(8 bit)为单位的，而 DS18B20 是单总线的，即一次只能写入或读出 1 bit 的数据。数据在 DS18B20 总线上的传输，是按照一定时序要求的，此处给出 DS18B20 的复位、写入和读取的时序图。

1. 复位时序

DS18B20 的复位时序图如图 7－2 所示。

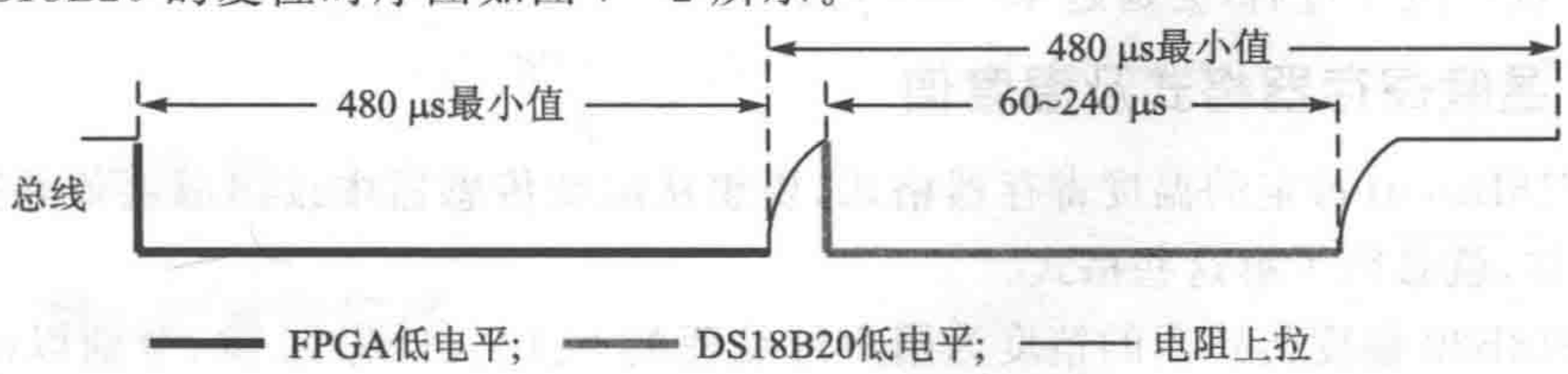


图 7－2　DS18B20 复位时序

① FPGA 拉低总线 480～950 μs，然后释放总线（拉高电平）；

② DS18B20 拉低信号 60～240 μs 表示应答；

③ DS18B20 拉低电平 60～240 μs 之后，会释放总线。

④ 整个复位操作最少持续 960 μs。

2. 写时序

DS18B20 的写时序图如图 7－3 所示。

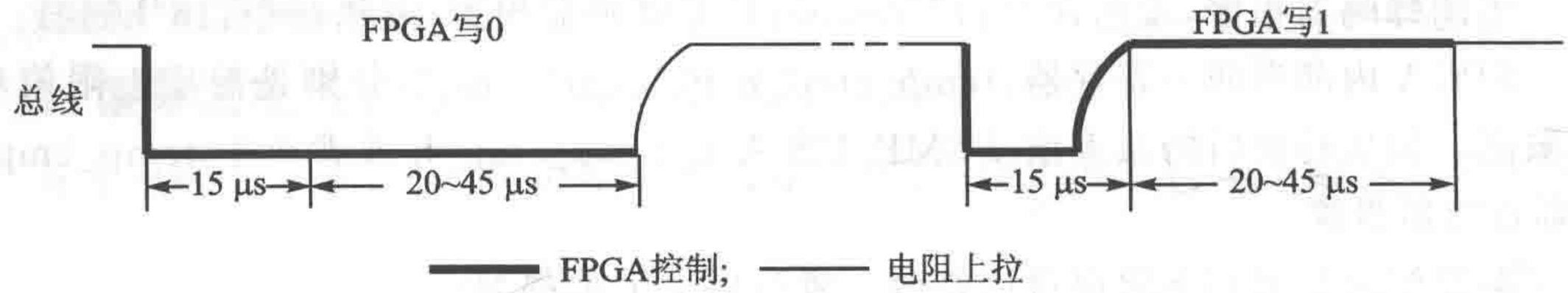


图 7－3　DS18B20 写时序

① FPGA 拉低电平 10～15 μs；

② FPGA 持续拉低/高电平 20～45 μs 的时间；

③ 释放总线。

3. 读时序

DS18B20 的读时序图如图 7－4 所示。

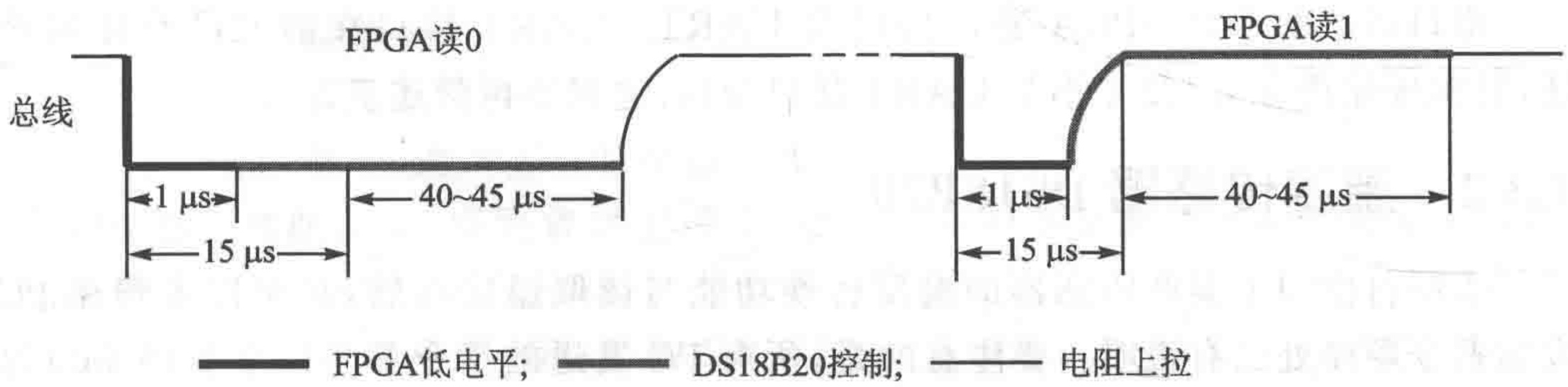


图 7－4　DS18B20 读时序

① 在读取的时候 FPGA 拉低电平大约 1 μs；

② FPGA 释放总线，此时 DS18B20 会拉低/高电平，FPGA 读取总线电平（必须在 15 μs 内读取）；

③ 读取电平过后，会延迟 40～45 μs。

4. 温度寄存器格式及温度值

DS18B20 有特定的温度寄存器格式，要想从温度传感器中读出温度值并进行显示等操作，就必须了解这些格式。

DS18B20 温度传感器的精度为用户可编程的 9、10、11 或 12 位，分别以0.5 ℃、0.25 ℃、0.125 ℃和 0.062 5 ℃增量递增。在上电状态下，默认的精度为 12 位，温度

增量为0.062 5 ℃。本项目采用默认的精度及增量，温度寄存器的格式如下：

	位7	位6	位5	位4	位3	位2	位1	位0
LS Byte	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
	位15	位14	位13	位12	位11	位10	位9	位8
MS Byte	S	S	S	S	S	2^6	2^5	2^4

根据精度与增量，可以得到以下温度值计算公式：

当温度为正值时：温度值＝{ds_temp_msb_reg，ds_temp_lsb_reg}×0.062 5 ℃

当温度为负值时：温度值＝(～{ds_temp_msb_reg，ds_temp_lsb_reg}＋1)×0.062 5 ℃

表7－2所列是温度与数据转换的关系。

表7－2 温度与数据转换关系

温度/℃	数据输出(二进制)	数据输出(十六进制)
＋125	0000 0111 1101 0000	07D0h
＋85	0000 0101 0101 0000	0550h
＋25.062 5	0000 0001 1001 0001	0191h
＋10.125	0000 0000 1010 0010	00A2h
＋0.5	0000 0000 0000 1000	0008h
0	0000 0000 0000 0000	0000h
－0.5	1111 1111 1111 1000	FFF8h
－10.125	1111 1111 0101 1110	FF5Eh
－25.062 5	1111 1110 0110 1111	FE6Eh
－55	1111 1100 1001 0000	FC90h

上电复位时温度寄存器默认值为＋85 ℃。

下面举两个温度计算的例子。

计算16'h07D0对应的温度值：

$$16'h07D0 = 16'd2000$$

$$16'd2000 \times 0.062\ 5 = 125\ ℃$$

计算16'hFFF8对应的温度值：

$$\sim 16'hFFF8 = 16'h0007$$

$$(16'h0007 + 1) \times 0.062\ 5 = -0.5\ ℃$$

7.3.3 数码管显示

1. 数码管显示原理

数码管显示原理图如图7－5所示。

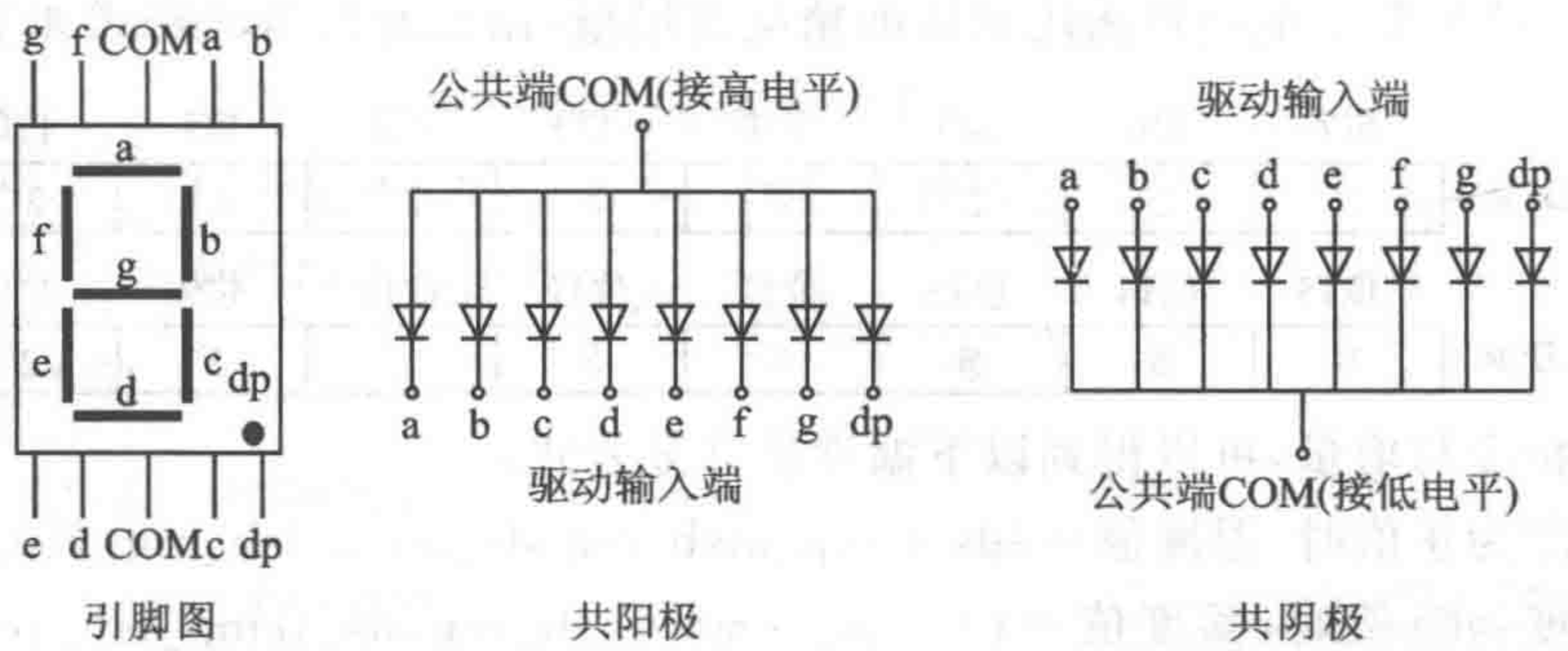


图 7-5 数码管显示原理图

数码管是由 8 个发光二极管组成的，每一段代表一个发光二极管。明德扬开发板使用的是共阳极数码管，只要给编号 a～dp 的任一引脚输入低电平，该段的二极管就被点亮。共阳极数码管显示数字与各个引脚电平的对应关系见表 7-3。

表 7-3 数码管译码表

显示数字＼对应显示段	a	b	c	d	e	f	g	dp
0	0	0	0	0	0	0	1	1
1	1	0	0	1	1	1	1	1
2	0	0	1	0	0	1	0	1
3	0	0	0	0	1	1	0	1
4	1	0	0	1	1	0	0	1
5	0	1	0	0	1	0	0	1
6	0	1	0	0	0	0	0	1
7	0	0	0	1	1	1	1	1
8	0	0	0	0	0	0	0	1
9	0	0	0	1	1	0	0	1

2. 多个数码管同时显示——动态扫描原理

图 7-6 所示为 4 位数码管 com1～com4，共用引脚 a～dp，为了简化硬件电路，将所有数码管的段选线相应地并联在一起，由 FPGA 的 8 个 I/O 口控制，形成段选线的多路复用。

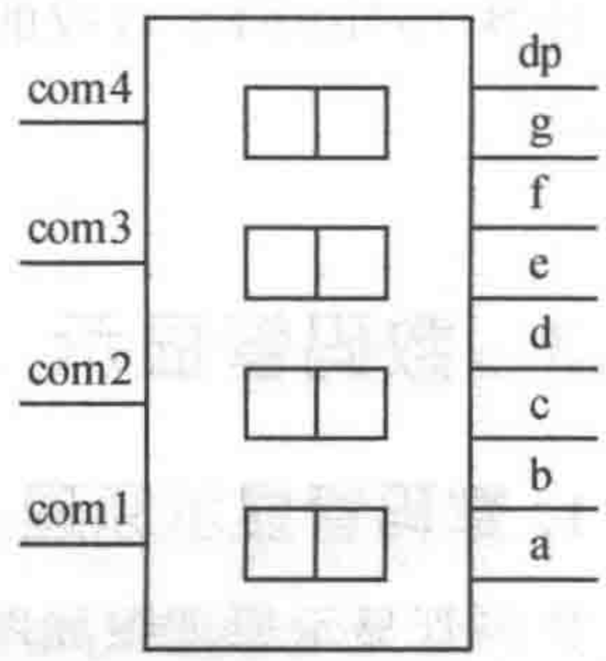


图 7-6 数码管电路图

各位数码管的公共端 COM（位选线）分别由 FPGA 独立的 I/O 口线控制，在某一位的位选线处于选通状态时，其他各位的位选线处于关闭状态。这样，4 位数码管中只有选通的那一位显示出字符，而

其他位则是熄灭的。同样，在下一时刻，只让下一位的位选线处于选通状态，而其他的位选线处于关闭状态，如此循环下去，顺序地点亮每位数码管。而段选线则随着数码管的顺序点亮而变化，使得每一位数码管能够显示出本位相应的字符。这样的数码管驱动方式就称为“动态扫描”。

由于人眼有视觉暂留现象，只要每位显示间隔足够短，就可造成多位同时亮的假象，达到显示的目的。一般时间间隔可设置为 20 μs。

7.3.4 蜂鸣器

明德扬开发板采用的是如图 7-7 所示的有源蜂鸣器。有源蜂鸣器内部带振荡源，所以只要一通电就会叫。因此它的驱动非常简单，只需要 FPGA 与蜂鸣器相连的引脚为高电平，蜂鸣器就会响。

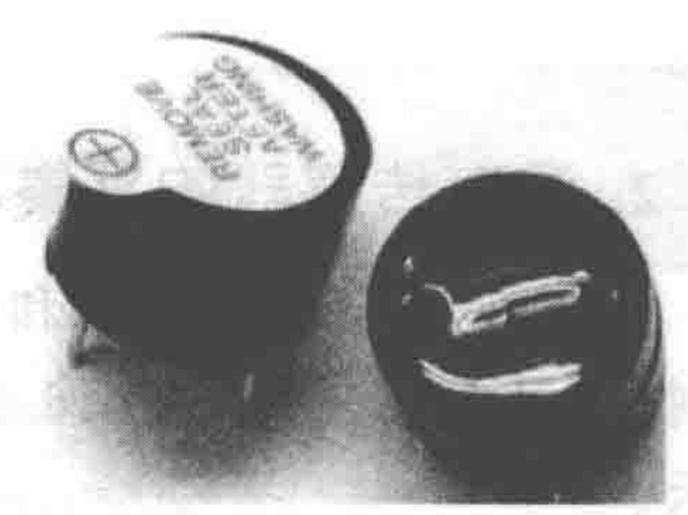


图 7-7 蜂鸣器实物图

7.4 设计方案

在本项目涉及 FPGA 与 PC 的相互通信、温度传感器的控制与采集、数码管的显示及蜂鸣器的驱动，下面依次考虑其设计方案。

1. FPGA 与 PC 的通信

FPGA 与 PC 是通过 UART 接口来相互通信的。在 2.2.2 小节的计数器实践项目中，介绍了 UART 接口的接收和发送，关于 UART 接口的接收模块和发送模块的设计，在前面已经讲得非常详细，所以在此处不再赘述，直接给出模块划分框图，如图 7-8 所示。

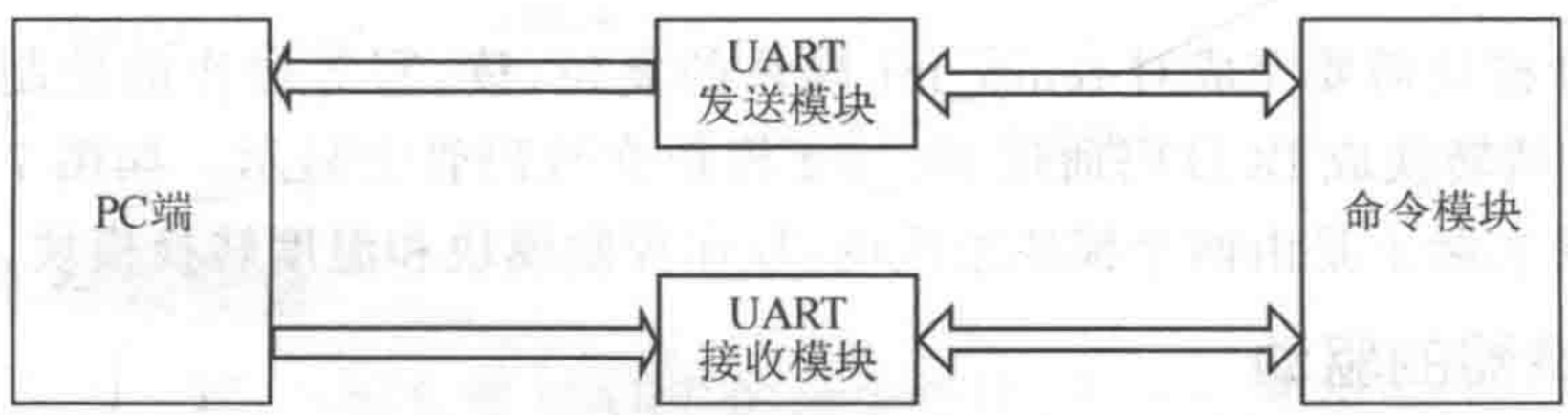


图 7-8 FPGA 与 PC 通信框图

在本项目中，命令模块可以进一步细分。根据项目要求，UART 接收模块接收来自 PC 端发送的命令，送给下游模块进行命令解析及识别；而从温度传感器中读取出来的数据则要转换成 BCD 码，再通过 UART 发送模块返回给 PC。因此，该命令模块完成两个功能，根据模块划分原则一，一个功能一个模块，将命令模块划分为指令识别模块和温度转换模块。划分结果如图 7-9 所示。

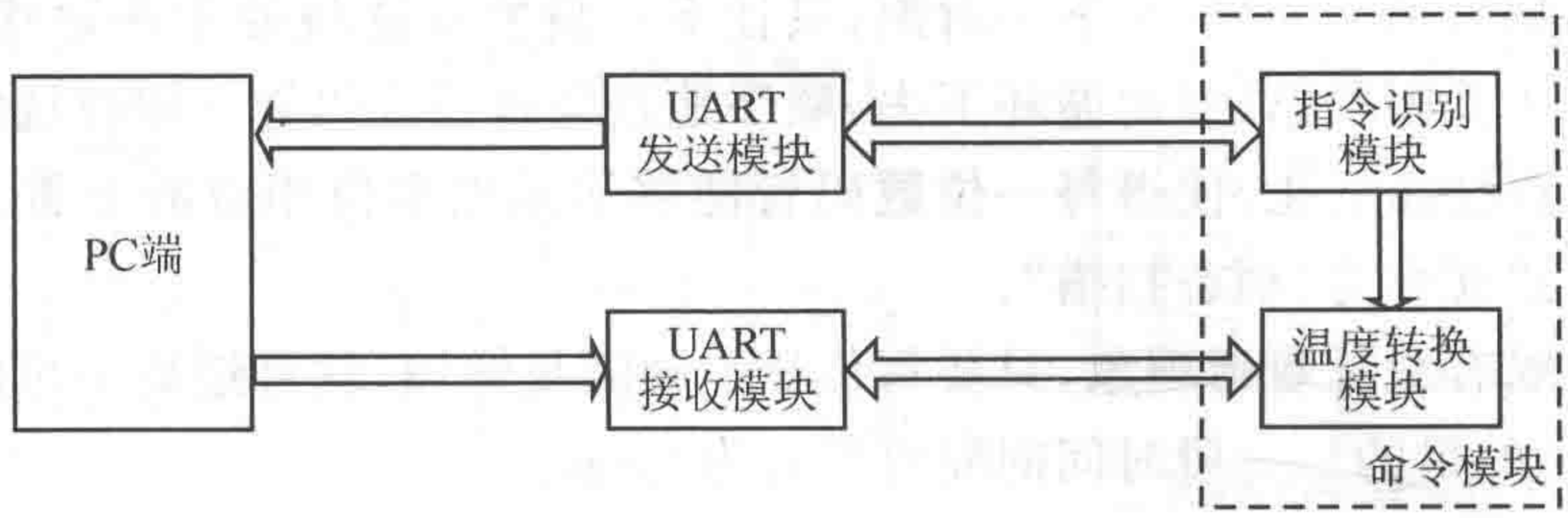


图 7-9　命令模块模块划分

2. 温度传感器的采集与显示

温度传感器 DS18B20 和数码管作为外设，使用不需配置外设交互架构完成模块划分，其划分过程见 6.6.2 小节温度采集与显示项目。划分结果如图 7-10 所示。

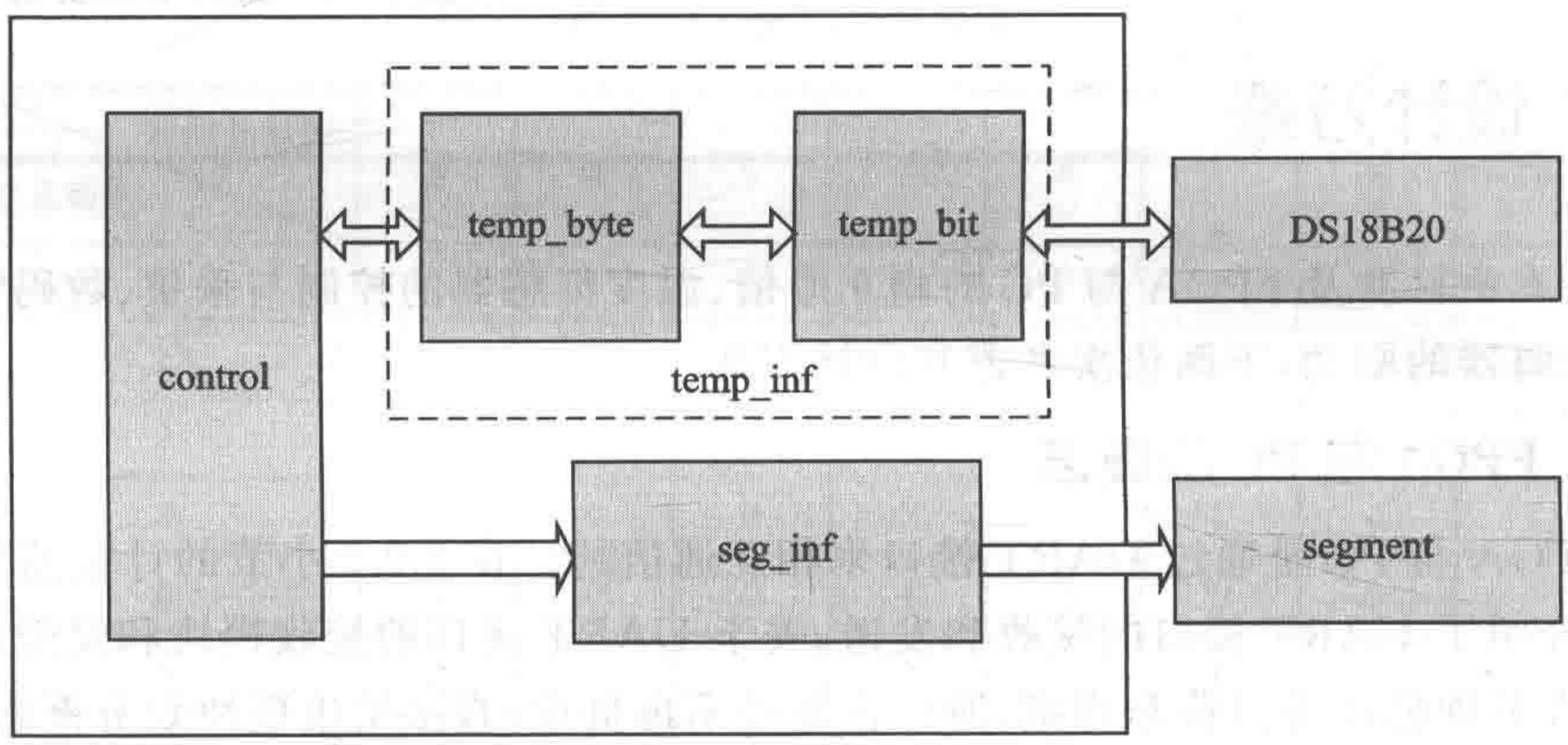


图 7-10　温度传感器模块划分

control 模块需要完成对 temp_inf 模块的复位、读、写等操作的控制，还需要把读取的温度值转换成 BCD 码通过 seg_inf 模块在数码管上显示。如图 7-11 所示，control 模块实际上是由两个模块组成的：操作控制模块和温度转换模块。

3. 蜂鸣器的驱动

从 7.3.4 小节的蜂鸣器原理可知，蜂鸣器的使用非常简单，不需要专门的接口模块对其进行驱动，在此可以直接通过命令模块完成对蜂鸣器的驱动。蜂鸣器模块划分结果见图 7-12。

本项目要求蜂鸣器能够在温度超出 PC 设置的温度上下限时报警。因此，蜂鸣器的命令模块应该有两部分：边界设置模块和温度比较模块。划分结果如图 7-13 所示。

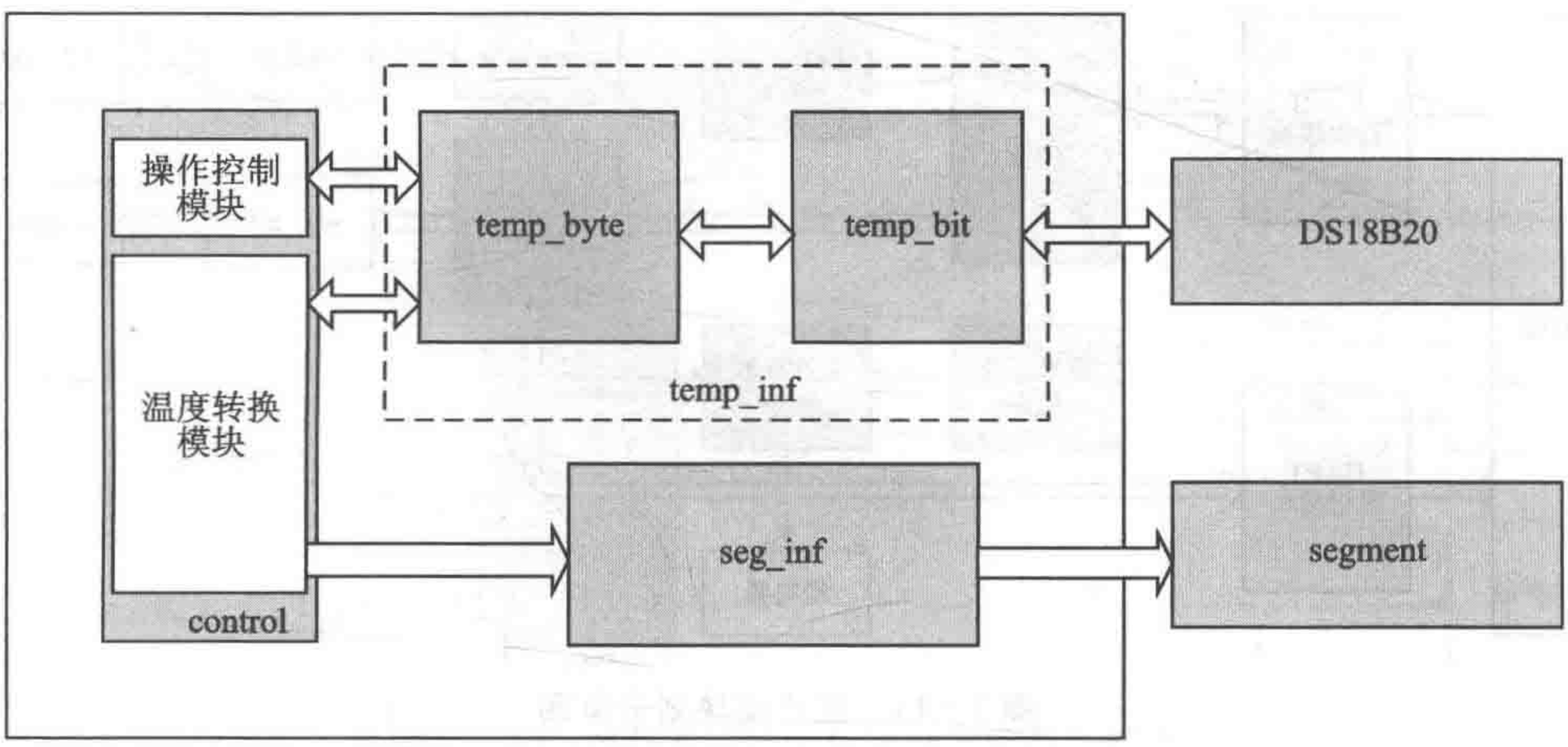


图 7－11　结合本项目温度传感器的模块划分

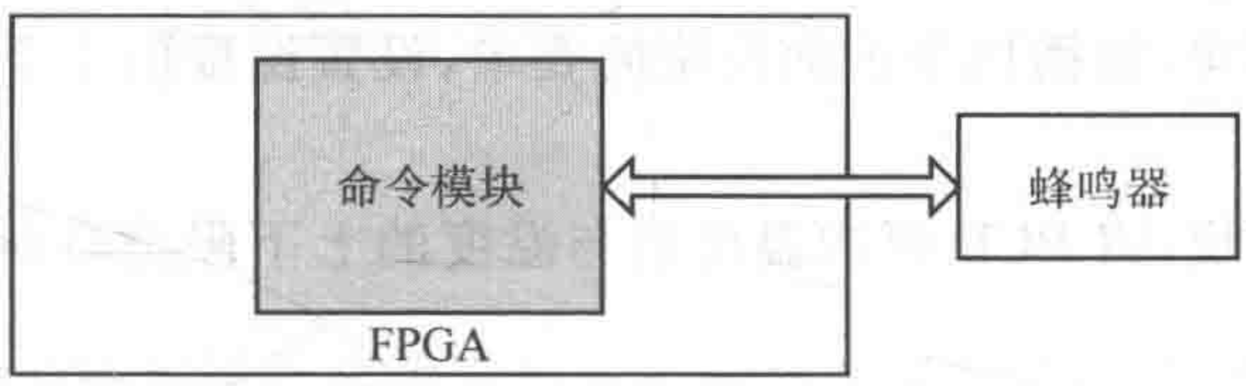


图 7－12　蜂鸣器模块划分

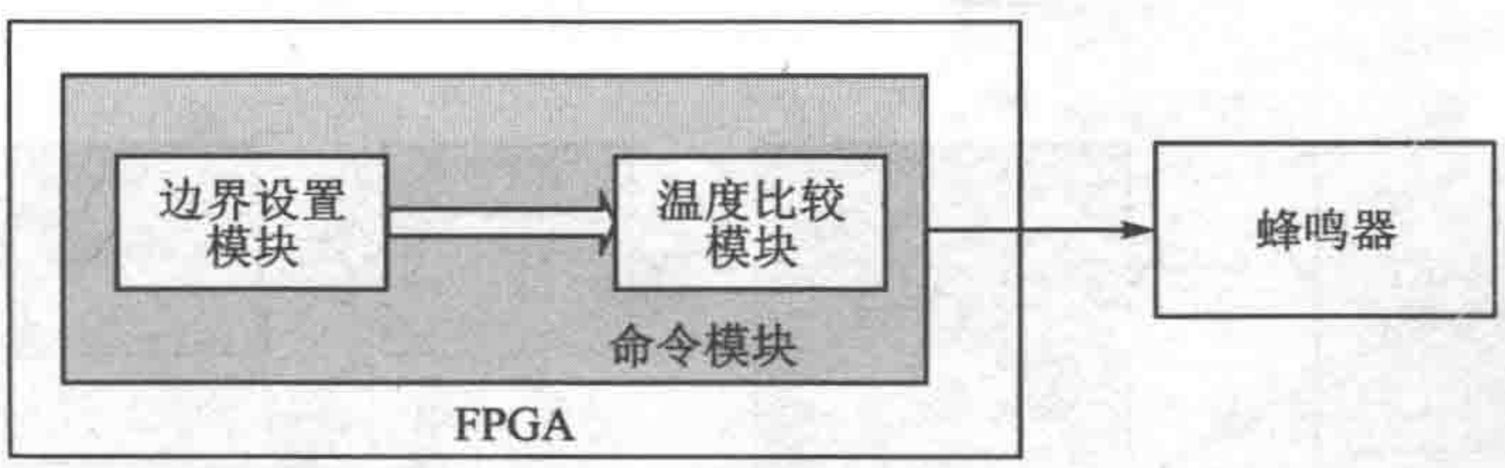


图 7－13　结合本项目蜂鸣器的模块划分

4. 整体模块划分

综合上面的分析，不难发现 UART 的命令模块、温度传感器与数码管的控制模块，以及蜂鸣器的命令模块都是互相关联的，可以得出系统整体框图如图 7－14 所示。

下面简要分析图 7－14 中的几个命令和控制模块。

- 指令识别模块：先识别 PC 发送指令中的类型，然后根据类型与数据产生对应的操作命令发送给下游模块。
- 操作控制模块：根据指令识别模块的要求，对 DS18B20 的接口模块发出复位、写/读等操作使能及输出相应的写数据。

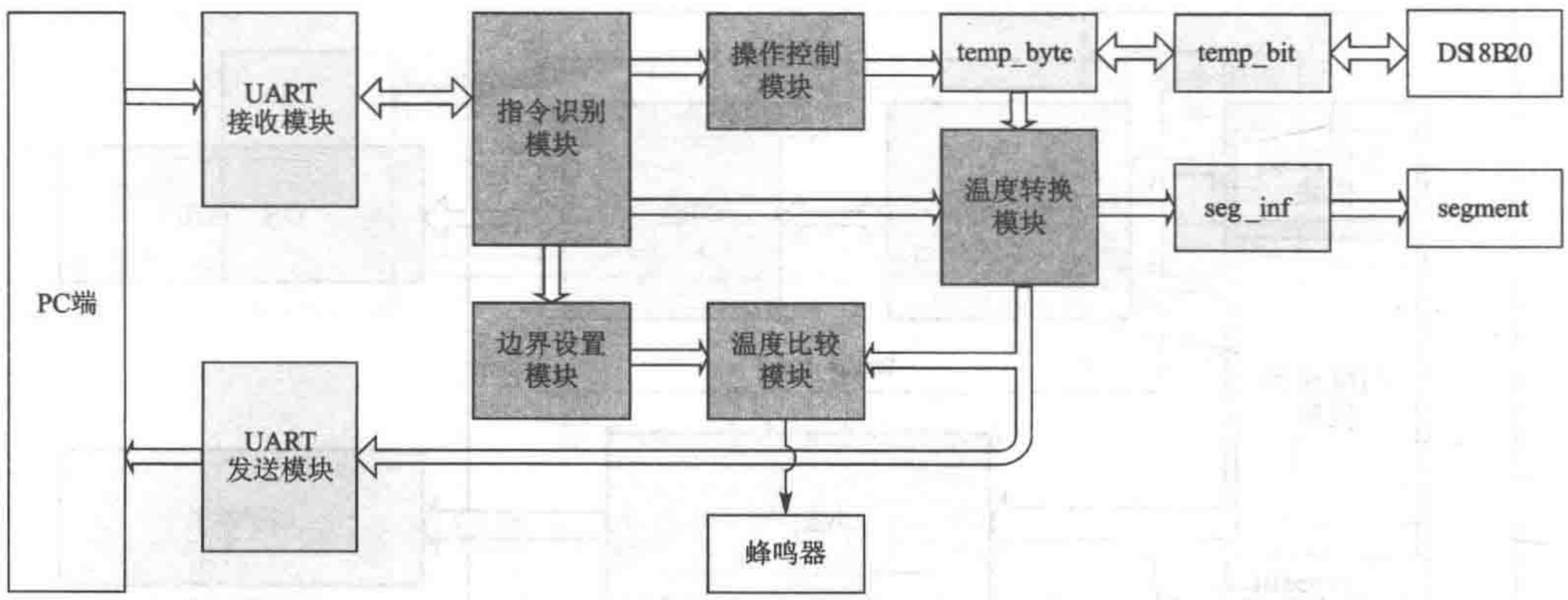


图 7－14　整体模块划分框图

- 温度转换模块：根据指令识别模块的要求，把从 DS18B20 读出的数据转换成 BCD 码的温度值，发送给温度比较模块并送到数码管显示。
- 边界设置模块：根据指令识别模块的要求，设置温度的上下限，并发送到温度比较模块。
- 温度比较模块：将 BCD 码的温度值与温度的上下限进行对比，判断是否要驱动蜂鸣器。

可以发现，上述 5 个模块只完成简单的逻辑判断，互相配合控制着整个系统。因此，在设计项目时，把这些模块合并成为一个模块，命名为控制模块。最终的系统框图如图 7－15 所示。

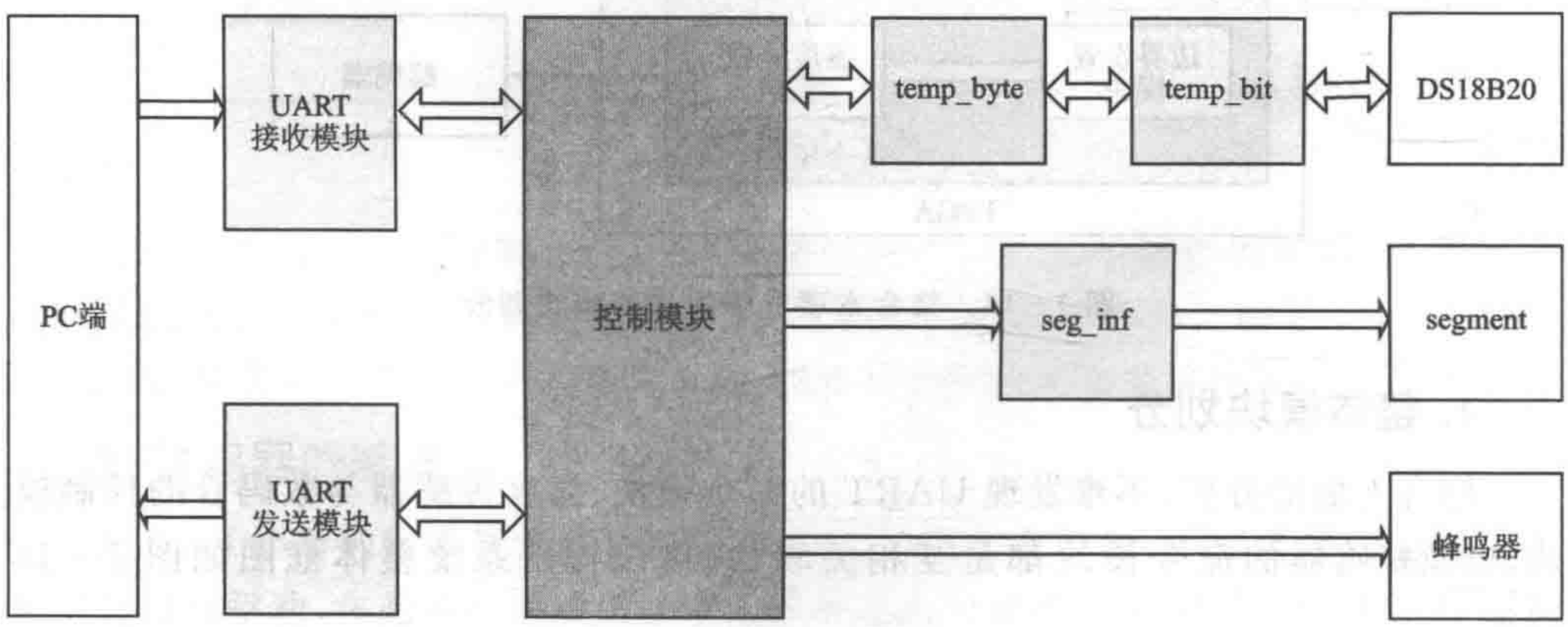


图 7－15　温度监控系统框图

5. 模块的端口及模块之间的数据流向

模块划分好后，需要进一步确认模块的端口及模块之间的数据流向。可参考 5.1.2小节模块划分端口规范。模块划分的最终结果如图 7－16 所示。

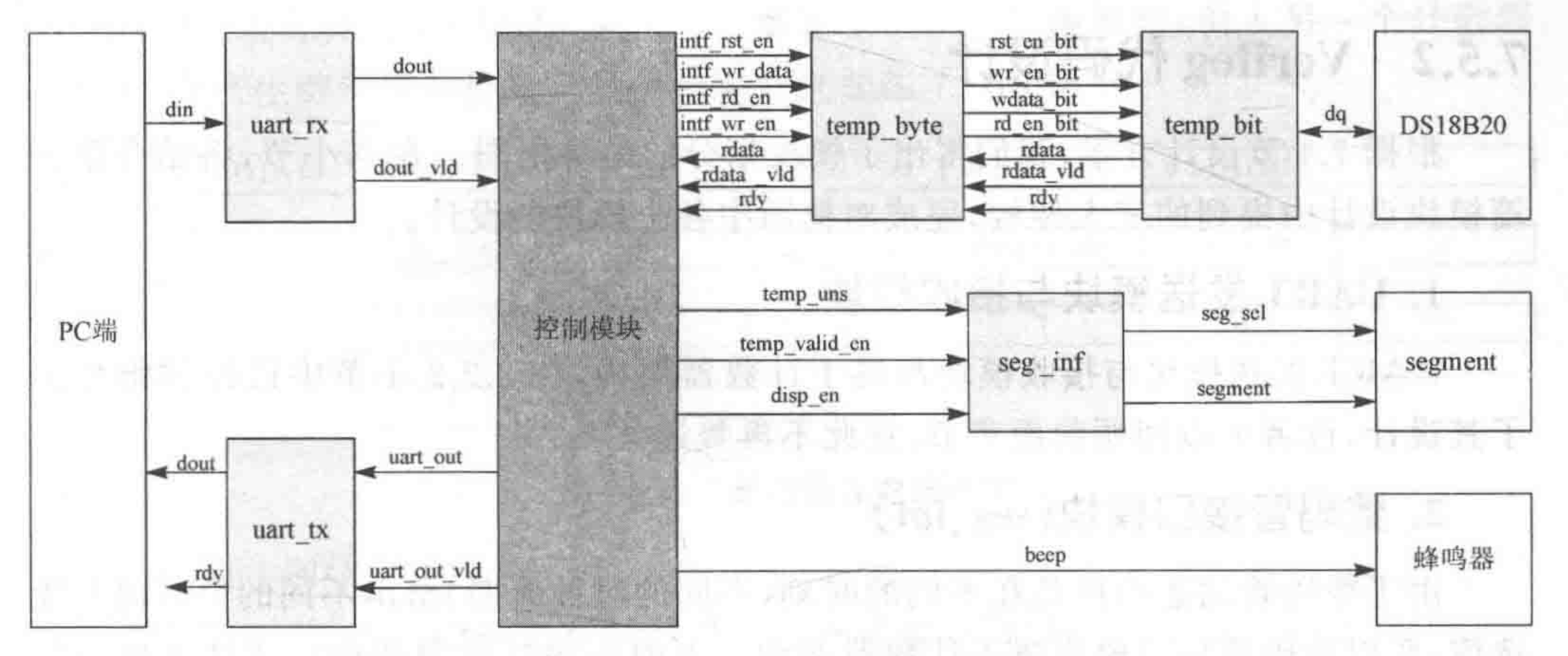


图 7－16　温度监控系统模块划分的最终框图

7.5　FPGA 设计实现

7.5.1　硬　件

图 7－17 所示为明德扬至简荣耀版开发板以及温度传感器、数码管模块。

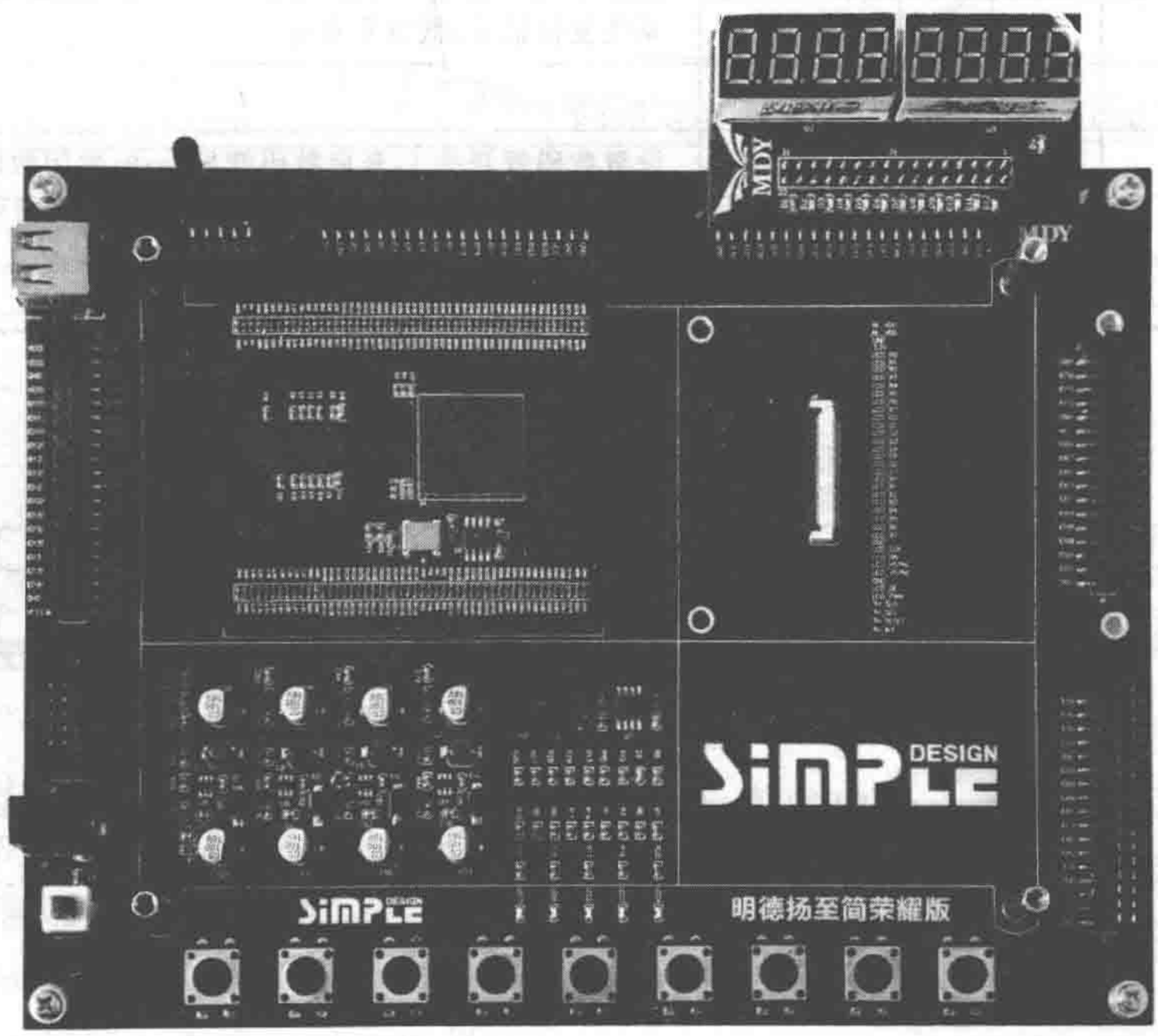


图 7－17　明德扬至简荣耀版开发板以及温度传感器、数码管模块

7.5.2 Verilog 代码设计

根据 7.4 节设计方案，我们得出了模块划分的最终框图。在本小节，将结合第一篇模块设计中提到的三大架构，完成对框图中各个模块的设计。

1. UART 发送模块与接收模块

UART 发送模块与接收模块都属于计数器架构，在 2.2.2 小节中已经详细介绍了其设计，读者可以回顾前面章节，在此不再赘述。

2. 数码管接口模块(seg_inf)

由于数码管动态扫描是在不同的时刻(不同的时钟周期)给出不同的位选值与段选值，所以数码管接口模块属于计数器架构。下面使用计数器架构八步法完成模块设计。

(1) 明确功能

把上游模块传输来的 32 位 BCD 码在 8 个数码管上显示出来。信号列表见表 7-4。

表 7-4 数据管信号列表

信号名	I/O	位 宽	说 明
clk	I	1	系统时钟 50 MHz
rst_n	I	1	系统复位信号，低电平有效
din	I	32	32 位输入数据
disp_en	I	1	设置数码管开关 1：开启数码管显示；0：关闭数码管显示
seg_sel	O	8	数码管位选信号，低电平有效
segment	O	8	数码管段选信号

(2) 功能波形

图 7-18 所示为数码管功能波形图。

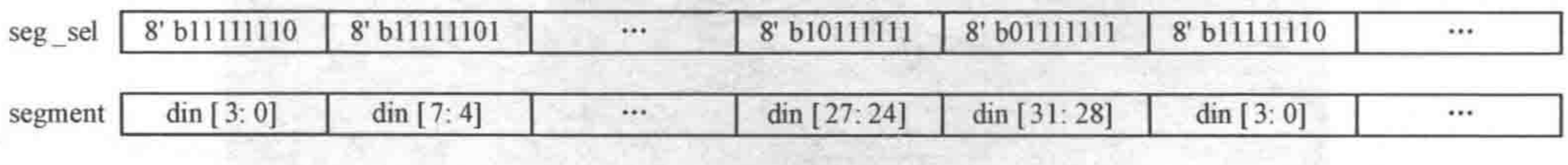


图 7-18 数据管功能波形图

要注意的是，segment 中的值 din[3:0]，…，din[31:28]，需要经过译码电路把 BCD 码转换成对应的数码管编码值，BCD 码及对应的编码值请参考 7.3.3 小节数码管显示基本原理。

(3) 计数结构

根据 7.3.3 小节中的数码管动态扫描原理，引入一个计数器 cnt_20us 计数

20 μs，因系统时钟为 50 MHz，故 20 μs 需要 1 000 个时钟周期；引入另一个计数器 sel_cnt 来产生数码管的位选。具体计数情况如图 7－19 所示。

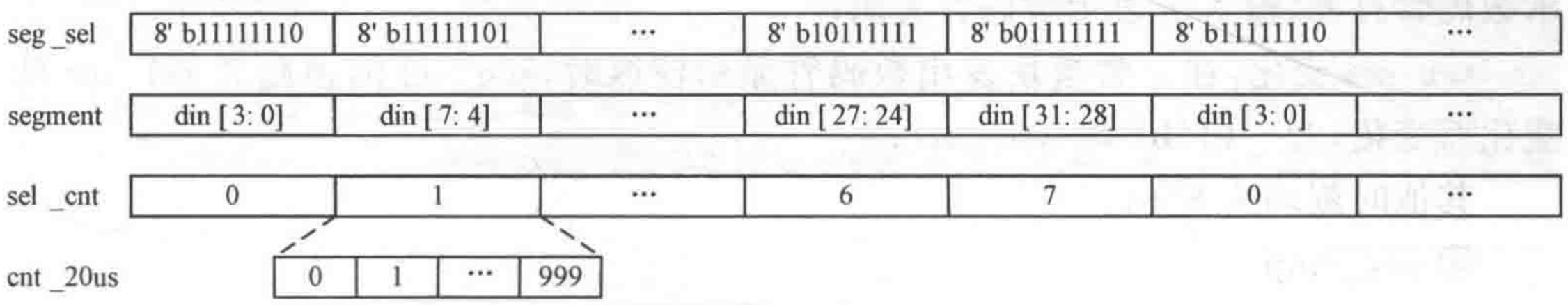


图 7－19 数码管计数结构图

(4) 加 1 和结束条件

cnt_20us 的加 1 条件：由于计数器一直在计数，所以加 1 条件为 1；

cnt_20us 的结束条件：加 1 条件下计数到 1 000－1；

sel_cnt 的加 1 条件：cnt_20us 的结束时刻；

sel_cnt 的结束条件：加 1 条件下计数到 8－1。

(5) 定义特殊点

图 7－20 中定义了几个特殊点，需要记住：

a 点，cnt_20us 的结束点：cnt＝＝1 000－1，定为 end_cnt_20us。

b 点，sel_cnt 的结束点：sel_cnt＝＝8－1，定为 end_sel_cnt。

另外，seg_sel，segment 与 sel_cnt 是在同一时刻变化的。

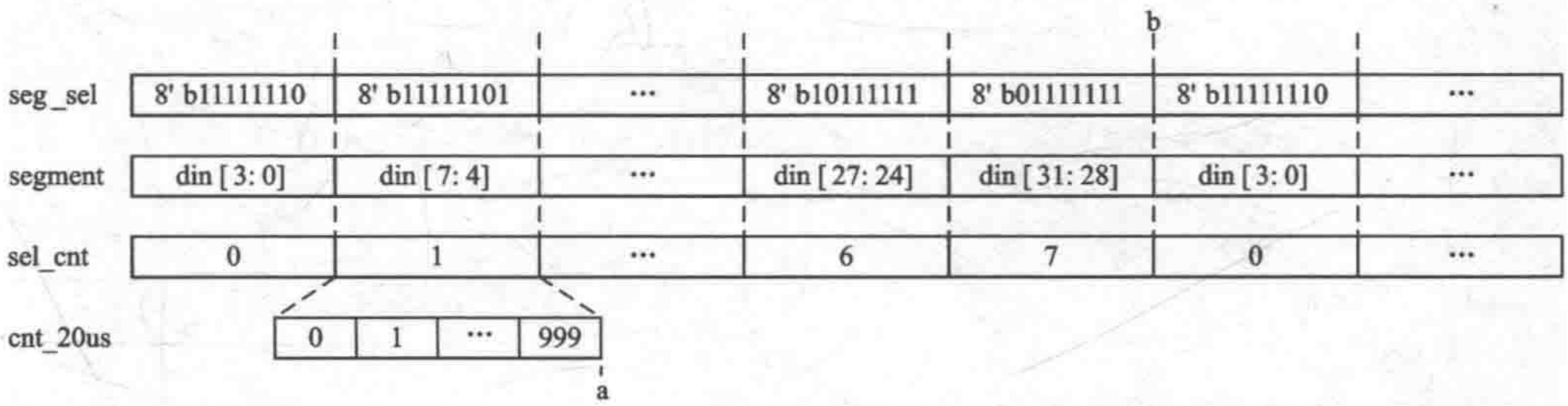


图 7－20 数码管定义特殊点图

(6) 完整性检查

① 计数器 cnt_20us

cnt_20us 的初始值：0；

cnt_20us 的加 1 条件：1；

cnt_20us 的结束条件：cnt_20us＝＝1 000－1。

② 计数器 sel_cnt

sel_cnt 的初始值：0；

sel_cnt 的加 1 条件：end_cnt_20us；

sel_cnt 的结束条件：sel_cnt＝＝8－1。

③ seg_sel

seg_sel 初值：8'b1，明德扬开发板中，FPGA 给数码管的 COM 段电路输入 0 表示数码管打开，输入 1 表示数码管关闭；

seg_sel 变化：在上游模块发出数码管显示使能时，seg_sel 的值随着 sel_cnt 的变化而变化，为～(1'b1 ≪ sel_cnt)；

其他时刻均为 8'b1。

④ seg_tmp

seg_tmp 表示当前数码管应该显示 din 的第几位，随着 sel_cnt 的变化而变化。

其值为 din[(sel_cnt + 1) * 4－1 －: 4]，当 sel_cnt 为 0 时，din[3:0]；当 sel_cnt 为 1 时，din[7:4]；依次类推。

⑤ segment

把 seg_tmp 译码后的送给数码管的值。

(7) 计数器代码

```
parameter TIME_20US   = 1_000;
always@(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        cnt_20us<= 26'b0;
    end
    else if (add_cnt_20us)begin
        if(end_cnt_20us) begin
            cnt_20us<= 0;
        end
        else begin
            cnt_20us<= cnt_20us + 1;
        end
    end
end
assign add_cnt_20us = 1;
assign end_cnt_20us = add_cnt_20us && cnt_20us == TIME_20US - 1;
```

```
parameter  SEG_NUM          =        8;
always  @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        sel_cnt <= 0;
    end
    else if(add_sel_cnt) begin
        if( end_sel_cnt)begin
            sel_cnt <= 0;
        end
        else begin
```

```
            sel_cnt <= sel_cnt + 1;
        end
    end
end
assign add_sel_cnt = end_cnt_20us;
assign end_sel_cnt = add_sel_cnt && sel_cnt == SEG_NUM - 1;
```

(8)功能代码

```
parameter  SEG_WID          =       8;

parameter  NUM_0            =       8'b1100_0000;
parameter  NUM_1            =       8'b1111_1001;
parameter  NUM_2            =       8'b1010_0100;
parameter  NUM_3            =       8'b1011_0000;
parameter  NUM_4            =       8'b1001_1001;
parameter  NUM_5            =       8'b1001_0010;
parameter  NUM_6            =       8'b1000_0010;
parameter  NUM_7            =       8'b1111_1000;
parameter  NUM_8            =       8'b1000_0000;
parameter  NUM_9            =       8'b1001_0000;
parameter  NUM_F            =       8'b1011_1111;
parameter  NUM_ERR          =       8'b1000_0110;

always   @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        seg_sel <= {SEG_NUM{1'b1}};
    end
    else if(disp_en)
        seg_sel <= ~(1'b1 << sel_cnt);
    else
        seg_sel <= {SEG_NUM{1'b1}};
end

always   @( * )begin
    seg_tmp = din_ff0[(sel_cnt + 1) * 4 - 1 - : 4];
end

always   @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        din_ff0 <= 0;
    end
```

```
34      else begin
35          din_ff0 <= din;
36      end
37  end
38
39  /**********数码管段选信号译码**********/
40  always@(posedge clk or negedge rst_n)begin
41      if(rst_n == 1'b0)begin
42          segment<= NUM_0;
43      end
44      else if(seg_tmp == 0)begin
45          segment<= NUM_0;
46      end
47      else if(seg_tmp == 1)begin
48          segment<= NUM_1;
49      end
50      else if(seg_tmp == 2)begin
51          segment<= NUM_2;
52      end
53      else if(seg_tmp == 3)begin
54          segment<= NUM_3;
55      end
56      else if(seg_tmp == 4)begin
57          segment<= NUM_4;
58      end
59      else if(seg_tmp == 5)begin
60          segment<= NUM_5;
61      end
62      else if(seg_tmp == 6)begin
63          segment<= NUM_6;
64      end
65      else if(seg_tmp == 7)begin
66          segment<= NUM_7;
67      end
68      else if(seg_tmp == 8)begin
69          segment<= NUM_8;
70      end
71      else if(seg_tmp == 9)begin
72          segment<= NUM_9;
73      end
74      else if(seg_tmp == 4'hf)begin
75          segment<= NUM_F;
76      end
77      else begin
```

```
78            segment<= NUM_ERR;
79        end
80 end
```

3. DS18B20 接口模块(temp_bit 与 temp_byte)

temp_bit 模块与 temp_byte 模块需要实现复位、读操作和写操作,与 3.2.2 小节的 SPI 接口项目类似,可以使用状态机来区分互斥(不会同时发生)的三个操作,这种做法比较节省 FPGA 资源。因此,temp_bit 模块与 temp_byte 模块的设计都使用状态机架构。下面用状态机架构八步法分别完成两个模块的设计。

(1) temp_bit 模块设计

1) 明确功能

接收上游模块的指令与数据,实现对温度传感器 DS18B20 的复位、读和写。信号列表见表 7-5。

表 7-5 lemp_bit 模块信号列表

信号名	I/O	位 宽	说 明
clk	I	1	系统工作时钟
rst_n	I	1	系统复位信号,低电平有效
rst_en	I	1	温度传感器复位使能
wr_en	I	1	温度传感器写使能
wdata	I	1	温度传感器写数据
rd_en	I	1	温度传感器读使能
dq_in	I	1	温度传感器输入本模块的数据
rdata	O	1	从温度传感器读到的数据
rdata_vld	O	1	温度传感器读数据有效指示信号
dq_out	O	1	三态门输入信号
dq_out_en	O	1	三态门使能端
rdy	O	1	模块忙闲指示信号

2) 输出分析

① dq_out 的输出参考 7.3.2 小节温度传感器 DS18B20 的复位、读和写时序;

② dq_out_en:在 dq_out 有效时为 1;

③ rdata:读到的数据;

④ rdata_vld:读数据有效指示信号;

⑤ rdy:闲为 1,忙为 0。

3) 状态划分

根据接口时序,需要完成复位、读、写等操作,故可划分为四个状态,IDEL(初始

态)、RST_S(复位)、WR_S(写)、RD_S(读)。

4) 状态转移

确定划分后状态的转移条件及方向,如图 7-21 所示。

IDEL→RST_S:收到复位命令;

IDEL→RD_S:收到读命令;

IDEL→WR_S:收到写命令;

RST_S→IDEL:复位结束;

RD_S→IDEL:读过程结束;

WR_S→IDEL:写过程结束;

其他情况状态不变。

5) 转移条件

根据转移条件,画出状态转移图如图 7-22 所示。

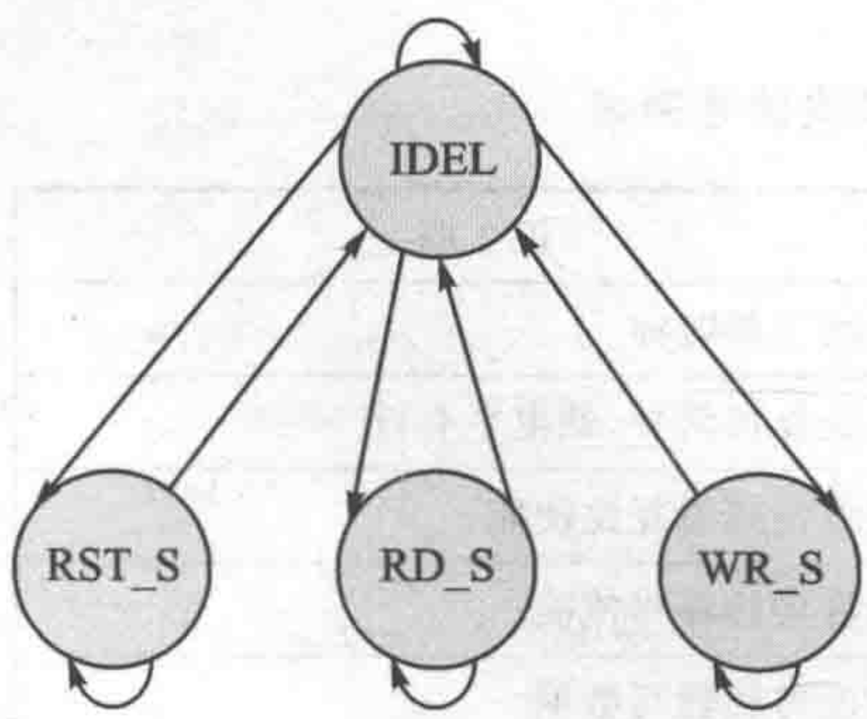


图 7-21 温度传感器 DS18B20 状态划分

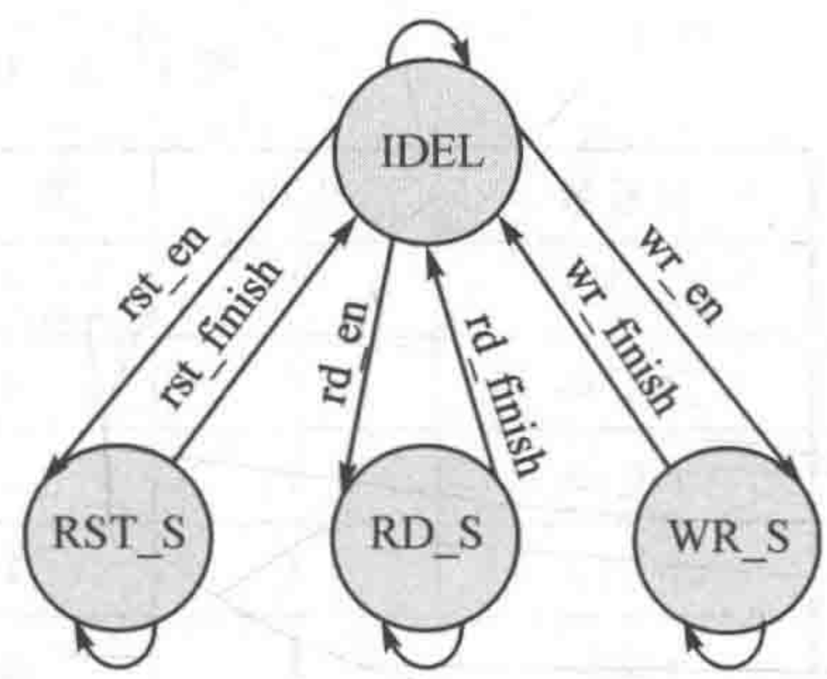


图 7-22 温度传感器 DS18B20 状态转移图

IDEL→RST_S:收到复位命令 rst_en;

IDEL→RD_S:收到读命令 rd_en;

IDEL→WR_S:收到写命令 wd_en;

RST_S→IDEL:复位结束 rst_finish;

RD_S→IDEL:读过程结束 rd_finish;

WR_S→IDEL:写过程结束 wr_finish;

其他情况状态不变。

6)完整性检查

① cnt

cnt 初始值:0;

cnt 加 1 条件:不在 IDEL 状态, add_cnt = state_c! =IDLE。

cnt 结束值:复位、读、写结束,end_cnt = add_cnt&&rst_finish || wr_finish ||

rd_finish。其中：

rst_finish 为 state_c==RST_S && cnt == 50 000−1 (1 000 μs)；

wr_finish 为 state_c==WR_S && cnt ==3 100−1(62 μs)；

rd_finish 为 state_c==RD_S && cnt == 3 100−1(62 μs)。

从 DS18B20 的数据手册可以查到这些时间值。

② dq_out

dq_out 是三态门的输入信号，当 dq_out_en 有效时，三态门把 dq_out 的值输出到温度传感器；当 dq_out_en 无效时，三态门输出为高阻态。而三态门的具体代码请见顶层模块相关代码。

③ dq_out_en

dq_out_en 是三态门的使能信号，在写温度传感器时为 1，读时为 0。

④ rdata

rdata 变化点：state_c==RD_S && cnt==TIME_RD_GET−1。

⑤ rdata_vld

rdata_vld 由 0 变为 1：state_c==RD_S && cnt==TIME_RD_GET−1；其他情况为 0。

⑥ rdy

rdy 为 1：在 rst_en 有效或 wr_en 有效或 rd_en 有效或当前不在 IDLE 状态；其他情况为 0。

7）状态机代码

```
1  parameter      IDLE =   0;
2  parameter      RST_S =   1;
3  parameter      WR_S =   2;
4  parameter      RD_S =   3;
5
6  always   @(posedge clk or negedge rst_n)begin
7      if(rst_n == 1'b0)begin
8          state_c <= IDLE;
9      end
10     else begin
11         state_c <= state_n;
12     end
13 end
14 always   @( * )begin
15     case(state_c)
16         IDLE : begin
17             if(idl2rst_start)
18                 state_n = RST_S;
```

```
19                  else if(idl2wr_start)
20                      state_n = WR_S;
21                  else if(idl2rd_start)
22                      state_n = RD_S;
23                  else
24                      state_n = state_c;
25              end
26              RST_S:begin
27                  if(rst2idl_start)
28                      state_n = IDLE;
29                  else
30                      state_n = state_c;
31              end
32              WR_S:begin
33                  if(wr2idl_start)
34                      state_n = IDLE;
35                  else
36                      state_n = state_c;
37              end
38              RD_S:begin
39                  if(rd2idl_start)
40                      state_n = IDLE;
41                  else
42                      state_n = state_c;
43              end
44              default: state_n = IDLE;
45          endcase
46  end
47  assign idl2rst_start = state_c == IDLE && rst_en;
48  assign idl2wr_start  = state_c == IDLE && wr_en;
49  assign idl2rd_start  = state_c == IDLE && rd_en;
50  assign rst2idl_start = state_c == RST_S && cnt == TIME_RST - 1;
51  assign wr2idl_start  = state_c == WR_S && cnt  == TIME_WR  - 1;
52  assign rd2idl_start  = state_c == RD_S && cnt  == TIME_RD  - 1;
```

8）功能代码

```
1   parameter     TIME_RST      =  50_000;   //1 000 μs
2   parameter     TIME_RST_LOW  =  37_500;   //750 μs
3   parameter     TIME_WR       =  3_100;    //62 μs
4   parameter     TIME_WR_INSTR =  750;      //15 μs
5   parameter     TIME_WR_DATA  =  3_000;    //60 μs
6   parameter     TIME_RD       =  3_100;    //62 μs
```

```
7   parameter       TIME_RD_INSTR   =   60;       //1 μs
8   parameter       TIME_RD_GET     =   700;      //14 μs
9   parameter       TIME_RD_DAT     =   3_000;    //60 μs
10
11  always   @(posedge clk or negedge rst_n)begin
12           if(rst_n == 1'b0)begin
13               cnt <= 0;
14           end
15           else if(add_cnt)begin
16               if(end_cnt)
17                   cnt <= 0;
18               else
19                   cnt <= cnt + 1;
20           end
21           else
22               cnt <= 0;
23  end
24  assign add_cnt = state_c! = IDLE;
25  assign end_cnt = add_cnt && rst_finish || wr_finish || rd_finish;
26  assign rst_finish = state_c == RST_S && cnt == TIME_RST - 1;
27  assign wr_finish  = state_c == WR_S  && cnt == TIME_WR - 1;
28  assign rd_finish  = state_c == RD_S  && cnt == TIME_RD - 1;
29
30  always   @(posedge clk or negedge rst_n)begin
31       if(rst_n == 1'b0)begin
32           wdata_ff0 <= 1'b0;
33       end
34       else if(state_c == IDLE && rst_en == 1'b0 && wr_en)begin
35           wdata_ff0 <= wdata;
36       end
37  end
38
39  always   @(posedge clk or negedge rst_n)begin
40       if(rst_n == 1'b0)begin
41           dq_out <= 1'b1;
42       end
43       else if(state_c == RST_S) begin
44           if(cnt<TIME_RST_LOW)
45               dq_out <= 1'b0;
46           else
47               dq_out <= 1'b1;
48       end
49       else if(state_c == WR_S)begin
```

```
50            if(cnt<TIME_WR_INSTR)
51                dq_out <= 1'b0;
52            else
53                dq_out <= wdata_ff0;
54        end
55        else if(state_c == RD_S)begin
56            if(cnt<TIME_RD_INSTR)
57                dq_out <= 1'b0;
58            else
59                dq_out <= 1'b1;
60        end
61        else begin
62            dq_out <= 1'b1;
63        end
64    end
65
66    always   @(posedge clk or negedge rst_n)begin
67        if(rst_n == 1'b0)begin
68            dq_out_en <= 1'b0;
69        end
70        else if(state_c != IDLE) begin
71            if(state_c == RD_S && cnt<TIME_RD_INSTR)
72                dq_out_en <= 1'b1;
73            else if(state_c == RST_S && cnt<TIME_RST_LOW)
74                dq_out_en <= 1'b1;
75            else if(state_c == WR_S && cnt<TIME_WR_DATA)
76                dq_out_en <= 1'b1;
77            else
78                dq_out_en <= 1'b0;
79        end
80        else begin
81            dq_out_en <= 1'b0;
82        end
83    end
84
85    always   @(posedge clk or negedge rst_n)begin
86        if(rst_n == 1'b0)begin
87            rdata <= 1'b0;
88        end
89        else if(state_c == RD_S && cnt == TIME_RD_GET - 1)begin
90            rdata <= dq_in;
91        end
92    end
```

```
93
94  always   @(posedge clk or negedge rst_n)begin
95       if(rst_n == 1'b0)begin
96           rdata_vld <= 1'b0;
97       end
98       else if(state_c == RD_S && cnt == TIME_RD_GET - 1)begin
99           rdata_vld <= 1'b1;
100      end
101      else begin
102          rdata_vld <= 1'b0;
103      end
104 end
105
106 always   @( * )begin
107      if(rst_en || wr_en || rd_en||state_c! = IDLE)
108          rdy = 1'b0;
109      else
110          rdy = 1'b1;
111 end
```

(2) temp_byte 模块

1) 明确功能

temp_byte 模块只是根据要求去复位、写 1 字节或读 1 字节，至于什么时候复位、写或读、写什么值、读的值有什么用，完全不关心。信号列表见表 7-6。

表 7-6 temp_byte 模块信号列表

信号名	I/O	位 宽	说 明
clk	I	1	系统工作时钟
rst_n	I	1	系统复位信号，低电平有效
rst_en	I	1	控制模块复位使能信号
wr_en	I	1	控制模块写使能信号
wdata	I	8	控制模块给出的写数据
rd_en	I	1	控制模块读使能信号
rdata	O	8	送给控制模块的读数据
rdata_vld	O	1	读数据有效信号
rdy	O	1	temp_byte 忙闲指示信号
rdata_bit	I	1	从 temp_bit 模块中读到的单比特读数据
rdata_vld_bit	I	1	单比特读数据有效信号
rdy_bit	I	1	temp_bit 模块忙闲指示信号

续表 7－6

信号名	I/O	位　宽	说　明
rst_en_bit	O	1	temp_bit 模块复位使能信号
wr_en_bit	O	1	temp_bit 模块写使能信号
wdata_bit	O	1	给 temp_bit 模块的单比特写数据
rd_en_bit	O	1	temp_bit 模块读使能信号

2）输出分析

temp_byte 模块的输出信号主要有三类：复位、写和读。每一类输出都是相互独立的，在进行某一操作期间，只能输出对应类别的信号。

复位：rst_en_bit。

写：wr_en_bit，wdata_bit。

读：rdata，rdata_vld，rd_en_bit。

而输出信号 rdy 应该是在进行复位、写和读等操作时输出忙，其他时候输出闲。

3）状态划分

可以用三个状态表示相互独立的三个操作，RST_S 是复位操作，RD_S 是读操作，WR_S 是写操作，另外还有一个初始状态 IDLE。

4）状态转移

确定划分后状态的转移条件及方向，如图 7－23 所示。

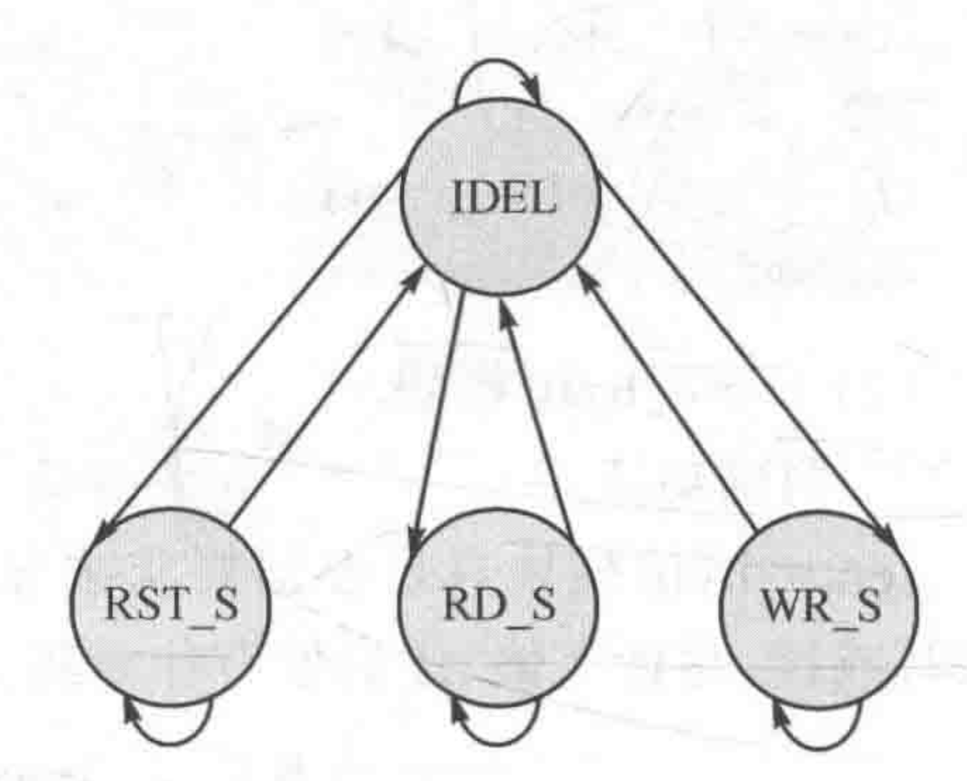


图 7－23　temp-byte 模块状态划分

IDLE→RST_S：收到来自控制模块的复位使能 rst_en。

IDLE→RD_S：收到来自控制模块的读使能 rd_en。

IDLE→WR_S：收到来自控制模块的写使能 wr_en。

RST_S→IDLE：在进入复位状态 RST_S 后，检测到 temp_bit 模块忙闲指示信号 rdy_bit 有效，表示复位操作已完成。

RD_S→IDLE：在进入复位状态 RST_S 后，检测到 8 次 temp_bit 模块读数据有效信号 rdata_vld_bit 有效，表示对 temp_bit 模块进行了 8 次读操作，共读出了 1 字节数据。

WR_S→IDLE：在进入复位状态 RST_S 后，检测到 8 次 temp_bit 模块忙闲指示信号 rdy_bit 有效，表示对 temp_bit 模块进行了 8 此写操作，共写入了 1 字节数据。

5）转移条件

由于不同操作需要的 rdy_bit 信号的有效次数不同，因此引入一个计数器 cnt，用于计算 rdy_bit 信号或 rdata_vld_bit 信号的有效次数。RST_S 状态有效次数为

1,RD_S 状态与 WR_S 状态有效次数为 8。

状态转移图见图 7-24。

IDLE→RST_S:收到 rst_en==1;

IDLE→RD_S:收到 rd_en==1;

IDLE→WR_S:收到 wr_en==1;

RST_S→IDLE:在 end_cnt 时刻;

RD_S→IDLE:在 end_cnt 时刻;

WR_S→IDLE:在 end_cnt 时刻。

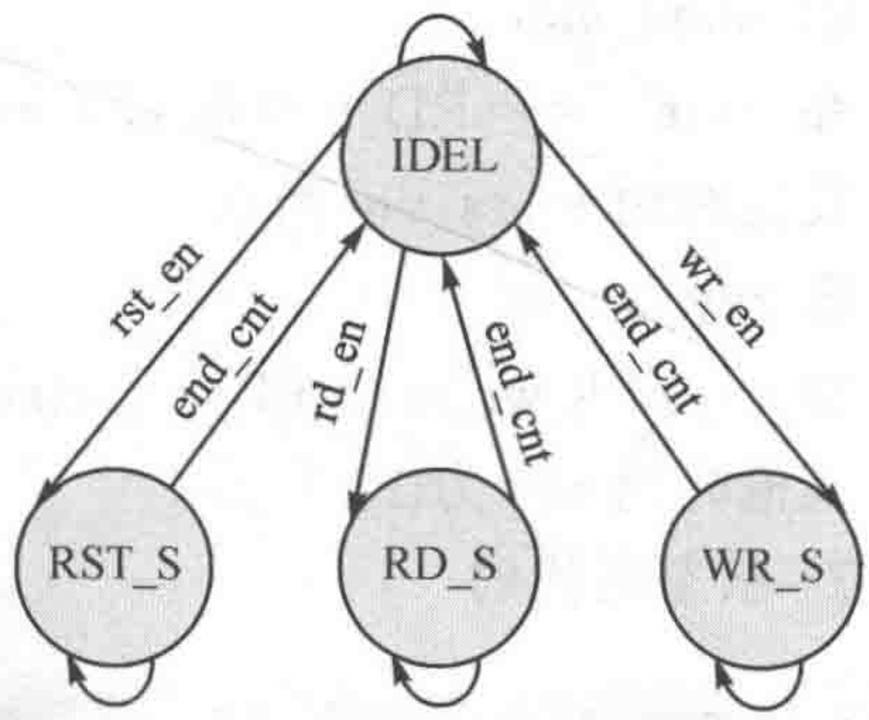


图 7-24 temp-btte 模块状态转移图

6) 完整性检查

① 计数器 cnt

cnt 的初值:0;

cnt 的加 1 条件:复位操作,stata_c==RST_S && rdy_bit;写操作,stata_c==WR_S && rdy_bit;读操作,stata_c==RD_S && rdata_vld_bit;

cnt 的结束值:复位操作计数到 1-1,写操作与读操作计数到 8-1。

② rst_en_bit

rst_en_bit 初值:0;

rst_en_bit 变 1:在 stata_c==RST_S 时,rdy_bit 为 1(temp_bit 模块可以接收复位指令);

其他时刻 rst_en_bit 为 0。

③ wr_en_bit

wr_en_bit 初值:0;

wr_en_bit 变 1:在 stata_c==WR_S 时,rdy_bit 为 1(temp_bit 模块可以接收写指令);

其他时刻 wr_en_bit 为 0。

④ wdata_bit

先把 wdata 的值缓存到 wdata_ff0 中,在满足 stata_c==WR_S && rdy_bit 时,wdata_ff0 从低位到高位(小头模式)依次赋值给 wdata_bit。代码是 wdata_bit <= wdata_ff0[cnt]。

⑤ rd_en_bit

rd_en_bit 初值:0;

rd_en_bit 变 1:在 stata_c==RD_S 时,rdy_bit 为 1(temp_bit 模块可以接收读指令);

其他时刻 rd_en_bit 为 0。

⑥ rdata

与 wdata_bit 类似,在满足 stata_c==RD_S && rdata_vld_bit 时,rdata_bit 依次赋值给 rdata 的低位到高位。代码是 rdata[cnt] <= rdata_bit。

⑦ rdata_vld

在 state_c==RD_S && end_cnt 时为 1，表示 rdata 已经接收完 1 字节的数据。其他时刻 rdata_vld 为 0。

⑧ rdy

在 rst_en || wr_en || rd_en || state_c! =IDLE 时，本模块正在忙，因此 rdy 为 0。其他时刻 rdy 为 1。

7）状态机代码

```
1  parameter     IDLE    =   0;
2  parameter     RST_S   =   1;
3  parameter     WR_S    =   2;
4  parameter     RD_S    =   3;
5
6  always   @(posedge clk or negedge rst_n)begin
7      if(rst_n == 1'b0)begin
8          state_c <= IDLE;
9      end
10     else begin
11         state_c <= state_n;
12     end
13 end
14
15 always   @( * )begin
16     case(state_c)
17         IDLE : begin
18             if(idl2rst_start)
19                 state_n = RST_S;
20             else if(idl2wr_start)
21                 state_n = WR_S;
22             else if(idl2rd_start)
23                 state_n = RD_S;
24             else
25                 state_n = state_c;
26         end
27         RST_S:begin
28             if(rst2idl_start)
29                 state_n = IDLE;
30             else
31                 state_n = state_c;
32         end
33         WR_S:begin
```

```
34                if(wr2idl_start)
35                    state_n = IDLE;
36                else
37                    state_n = state_c;
38            end
39            RD_S:begin
40                if(rd2idl_start)
41                    state_n = IDLE;
42                else
43                    state_n = state_c;
44            end
45            default: state_n = IDLE;
46        endcase
47 end
48
49 assign idl2rst_start = state_c == IDLE && rst_en;
50 assign idl2wr_start  = state_c == IDLE && wr_en;
51 assign idl2rd_start  = state_c == IDLE && rd_en;
52 assign rst2idl_start = state_c == RST_S && end_cnt;
53 assign wr2idl_start  = state_c == WR_S && end_cnt;
54 assign rd2idl_start  = state_c == RD_S && end_cnt;
```

8)功能代码

```
1   temp_bit u_temp_bit(
2                   .clk        (clk            ),
3                   .rst_n      (rst_n          ),
4                   .rst_en     (rst_en_bit     ),
5                   .wr_en      (wr_en_bit      ),
6                   .wdata      (wdata_bit      ),
7                   .rd_en      (rd_en_bit      ),
8                   .rdata      (rdata_bit      ),
9                   .rdata_vld (rdata_vld_bit),
10                  .dq_out     (dq_out         ),
11                  .dq_out_en (dq_out_en      ),
12                  .dq_in      (dq_in          ),
13                  .rdy        (rdy_bit        )
14              );
15
16  parameter      RW_NUM = 8;
17
18  always   @(posedge clk or negedge rst_n)begin
19      if(rst_n == 1'b0)begin
```

```
20          cnt <= 0;
21      end
22      else if(add_cnt)begin
23          if(end_cnt)
24              cnt <= 0;
25          else
26              cnt <= cnt + 1;
27      end
28  end
29  assign  add_cnt = (state_c == RD_S && rdata_vld_bit) ||
30                    ((state_c == RST_S || state_c == WR_S) && rdy_bit);
31  assign  end_cnt = add_cnt && cnt == x - 1;
32
33  always  @( * )begin
34      if(state_c == WR_S || state_c == RD_S)
35          x = RW_NUM;
36      else
37          x = 1;
38  end
39
40  always  @(posedge clk or negedge rst_n)begin
41      if(rst_n == 1'b0)begin
42          rst_en_bit <= 1'b0;
43      end
44      else if(state_c == RST_S && rdy_bit) begin
45          rst_en_bit <= 1'b1;
46      end
47      else begin
48          rst_en_bit <= 1'b0;
49      end
50  end
51
51  always  @(posedge clk or negedge rst_n)begin
53      if(rst_n == 1'b0)begin
54          wr_en_bit <= 1'b0;
55      end
56      else if(state_c == WR_S && rdy_bit) begin
57          wr_en_bit <= 1'b1;
58      end
59      else begin
60          wr_en_bit <= 1'b0;
61      end
```

```
62  end
63
64  always   @(posedge clk or negedge rst_n)begin
65      if(rst_n == 1'b0)begin
66          wdata_ff0 <= 0;
67      end
68      else if(state_c == IDLE && rst_en == 1'b0 && wr_en)begin
69          wdata_ff0 <= wdata;
70      end
71  end
72
73  always   @(posedge clk or negedge rst_n)begin
74      if(rst_n == 1'b0)begin
75          wdata_bit <= 1'b0;
76      end
77      else if(state_c == WR_S && rdy_bit) begin
78          wdata_bit <= wdata_ff0[cnt];
79      end
80  end
81
82  always   @(posedge clk or negedge rst_n)begin
83      if(rst_n == 1'b0)begin
84          rd_en_bit <= 1'b0;
85      end
86      else if(state_c == RD_S && rdy_bit) begin
87          rd_en_bit <= 1'b1;
88      end
89      else begin
90          rd_en_bit <= 1'b0;
91      end
92  end
93
94  always   @(posedge clk or negedge rst_n)begin
95      if(rst_n == 1'b0)begin
96          rdata <= 0;
97      end
98      else if(state_c == RD_S && rdata_vld_bit) begin
99          rdata[cnt] <= rdata_bit;
100     end
101 end
102
103 always   @(posedge clk or negedge rst_n)begin
```

```
104         if(rst_n == 1'b0)begin
105             rdata_vld <= 1'b0;
106         end
107         else if(state_c == RD_S && end_cnt)begin
108             rdata_vld <= 1;
109         end
110         else
111             rdata_vld <= 0;
112 end
113
114 always   @( * )begin
115         if(rst_en || wr_en || rd_en || state_c! = IDLE)
116             rdy = 1'b0;
117         else
118             rdy = 1'b1;
119 end
```

4. 控制模块

从 7.4 节设计方案中，我们知道控制模块需要实现哪些功能，这些功能是根据各种各样的实际需求而设计的，因此模块设计篇章中提到的模块三大架构并不适用于这些特殊功能的设计。下面介绍每个功能的设计思路，并给出设计结果。

首先考虑传输速率问题，从串口发送一个 16 bit 的指令到控制模块，大约需要(16+4)×5 208 个时钟周期，而控制模块对温度传感器执行复位操作需要约 50 000 个时钟周期，写操作和读操作都需要约 3 100×8 个时钟周期。因此执行指令的速度远远快于接收指令的速度，在这个前提下，设计控制模块时不需要考虑 DS18B20 接口模块的 rdy 信号。

(1) 指令解析功能

先判断数据是 type 还是 data。opcode_cnt==0 表示 type；opcode_cnt==1 表示 data。代码如下：

```
1  always   @(posedge clk or negedge rst_n)begin
2          if(rst_n == 1'b0)begin
3              opcode_cnt <= 0;
4          end
5          else if(add_opcode_cnt)begin
6              if(end_opcode_cnt)
7                  opcode_cnt <= 0;
8              else
9                  opcode_cnt <= opcode_cnt + 1;
10         end
```

```
end
assign add_opcode_cnt = uart_in_vld;
assign end_opcode_cnt = add_opcode_cnt && opcode_cnt == 2 - 1;
```

把输入的指令类型 type 给 opcode_instr。代码如下：

```
always  @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        opcode_instr <= 0;
    end
    else if(add_opcode_cnt && opcode_cnt == 0) begin
        opcode_instr <= uart_in;
    end
end
```

把输入的指令的数据 data 给 opcode_data。代码如下：

```
always  @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        opcode_data <= 0;
    end
    else if(add_opcode_cnt && opcode_cnt == 1) begin
        opcode_data <= uart_in;
    end
end
```

opcode_cnt 计数结束，表示接收到一个完整的指令，因此令 opcode_vld 有效。代码如下：

```
always  @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        opcode_vld <= 1'b0;
    end
    else if(end_opcode_cnt) begin
        opcode_vld <= 1'b1;
    end
    else begin
        opcode_vld <= 1'b0;
    end
end
```

(2) DS18B20 操作使能

根据项目要求，当收到指令的格式为 8'h80 时，FPGA 对 DS18B20 进行复位操作。代码如下：

```
1  parameterOP_DS_RST   =   8'h80;
2
3  //根据 opcode_instr,产生复位信号
4  always   @(posedge clk or negedge rst_n)begin
5       if(rst_n == 1'b0)begin
6            intf_rst_en <= 1'b0;
7       end
8       else if(opcode_vld && opcode_instr == OP_DS_RST) begin
9            intf_rst_en <= 1'b1;
10      end
11      else begin
12           intf_rst_en <= 1'b0;
13      end
14 end
```

当收到指令的格式为 8'h81 时，FPGA 对 DS18B20 进行写操作，写入的数据为 8bit，是指令的数据部分。代码如下：

```
1  parameter  OP_DS_WR   =   8'h81;
2
3  //根据 opcode_instr,产生写使能及写数据
4  always   @(posedge clk or negedge rst_n)begin
5       if(rst_n == 1'b0)begin
6            intf_wr_en <= 0;
7       end
8       else if(opcode_vld && opcode_instr == OP_DS_WR) begin
9            intf_wr_en <= 1'b1;
10      end
11      else begin
12           intf_wr_en <= 1'b0;
13      end
14 end
15
16 always   @(posedge clk or negedge rst_n)begin
17      if(rst_n == 1'b0)begin
18           intf_wdata <= 0;
19      end
20      else if(opcode_vld && opcode_instr == OP_DS_WR) begin
21           intf_wdata <= opcode_data;
22      end
23 end
```

当分别收到指令的格式为 8'h82,8'h83 或 8'h84 时,FPGA 读 DS18B20 的 LSB 寄存器、MSB 寄存器或其他寄存器。代码如下:

```
1  parameter  OP_DS_LSB_RD    =   8'h82;
2  parameter  OP_DS_MSB_RD    =   8'h83;
3  parameter  OP_DS_ RD       =   8'h84;
4
5  //根据 opcode_instr,产生读使能
6  always   @(posedge clk or negedge rst_n)begin
7      if(rst_n == 1'b0)begin
8          intf_rd_en <= 0;
9      end
10     else if(opcode_vld &&
11                 (opcode_instr == OP_DS_LSB_RD
12                 || opcode_instr == OP_DS_MSB_RD || opcode_instr == OP_DS_RD)) begin
13         intf_rd_en <= 1'b1;
14     end
15     else begin
16         intf_rd_en <= 1'b0;
17     end
18 end
19
20 //用信号 rdata_addr 区分不同的读指令 opcode_instr
21 always   @(posedge clk or negedge rst_n)begin
22     if(rst_n == 1'b0)begin
23         rdata_addr <= 0;
24     end
25     else if(opcode_vld)begin
26         if(opcode_instr == OP_DS_LSB_RD)
27             rdata_addr <= 0;
28         else if(opcode_instr == OP_DS_MSB_RD)
29             rdata_addr <= 1;
30         else if(opcode_instr == OP_DS_RD)
31             rdata_addr <= 2;
32     end
33 end
34
35 //根据 rdata_addr,把读出的数据存到相应寄存器
36 always   @(posedge clk or negedge rst_n)begin
37     if(rst_n == 1'b0)begin
38         ds_temp_lsb_reg <= 0;
39     end
```

```
40      else if(intf_rdata_vld && rdata_addr == 0) begin
41          ds_temp_lsb_reg <= intf_rdata;
42      end
43 end
44
45 always   @(posedge clk or negedge rst_n)begin
46      if(rst_n == 1'b0)begin
47          ds_temp_msb_reg <= 0;
48      end
49      else if(intf_rdata_vld && rdata_addr == 1) begin
50          ds_temp_msb_reg <= intf_rdata;
51      end
52 end
53
54 always   @(posedge clk or negedge rst_n)begin
55      if(rst_n == 1'b0)begin
56          ds_temp_rdata_reg <= 0;
57      end
58      else if(intf_rdata_vld && rdata_addr == 2) begin
59          ds_temp_rdata_reg <= intf_rdata;
60      end
61 end
```

(3) 数码管开关功能

当收到指令的格式为 8'h01 时，指令的数据部分的最低位要送给数码管，根据该位判断数码管是否打开显示功能。代码如下：

```
1  parameter  OP_TEMP_VALID   =   8'h01;
2
3  //根据 opcode_ data 的最低位，判断数码管的开关
4  always   @(posedge clk or negedge rst_n)begin
5      if(rst_n == 1'b0)begin
6          temp_valid_en <= 1'b0;
7      end
8      else if(opcode_vld && opcode_instr == OP_TEMP_VALID)begin
9          temp_valid_en <= opcode_data[0];
10     end
11 end
```

(4) 设置温度上下限

当收到指令的格式为 8'h02 时，设置温度下限为此时指令的数据部分；当收到指令的格式为 8'h03 时，设置温度上限为此时指令的数据部分。代码如下：

```
1  parameter  OP_TEMP_CMP_L   =   8'h02;
2  parameter  OP_TEMP_CMP_H   =   8'h03;
3
4  //根据 opcode_data,分别设置温度比较的上下限
5  always   @(posedge clk or negedge rst_n)begin
6       if(rst_n == 1'b0)begin
7           temp_cmp_l <= 0;
8       end
9       else if(opcode_vld && opcode_instr == OP_TEMP_CMP_L)begin
10          temp_cmp_l <= opcode_data;
11      end
12 end
13
14 always   @(posedge clk or negedge rst_n)begin
15      if(rst_n == 1'b0)begin
16          temp_cmp_h <= 0;
17      end
18      else if(opcode_vld && opcode_instr == OP_TEMP_CMP_H)begin
19          temp_cmp_h <= opcode_data;
20      end
21 end
```

(5) 温度转换功能

假设读到的温度数据为 ds_temp_reg,则 ds_temp_reg={ds_temp_msb_reg,ds_temp_lsb_reg}),这是温度传感器自己定义的格式,要转换成可在数码管显示的 temp_uns。由 7.3.2 小节温度寄存器格式及温度值可知,温度显示可分成两部分:正负指示和数值。

正负:由表 7-2 可知,正负是由温度值的高 5 位来指示,如果全 1 则为负数,给数码管的值为 4'hF;高 5 位如果为 0 则为正数,给数码管的值为 4'h0。

数值:由表 7-2 和温度寄存器格式,可以观察出温度值和温度之间的规律:温度值的位 0~3 表示小数部分,位 4~10 表示整数部分。

$$正数的温度值=de_temp_reg\times 0.062\ 5$$

$$负数的温度值=((\sim de_temp_reg)+1)\times 0.062\ 5$$

上面的公式中用到了小数 0.062 5,那么 FPGA 如何表示小数呢?其实,FPGA 信号组合起来后表示的意义完全是自己定义的。例如,我们可以用 1'b1 表示 0.1,2'b10 表示 0.2,只要整个工程都按这个定义去运算(包括加法减除)即可。在此例中,上述公式两边可以乘以 10 000,则有

$$正数的温度值=de_temp_reg\times 625$$

$$负数的温度值=((\sim de_temp_reg)+1)\times 625$$

我们知道 ds_temp 代表的是实际温度乘以 10 000。

得到 ds_temp 之后，就可以计算 temp_uns 的各个部分的值了。例如：

$$temp_uns[3:0]=ds_temp\%10$$

$$temp_uns[7:4]=\left(\frac{ds_temp}{10}\right)\%10$$

上面的过程中，由于 ds_temp 位宽较大，所以需要用到比较大的乘法器和除法器。

将 ds_temp 分成整数部分和小数部分，则可将乘法和除法变小。依次得到正负符号位、整数位、小数位。

正数温度值小数部分：temp_hex_dot＝ds_temp_reg[3:0]×625；

负数温度值小数部分：temp_hex_dot＝ds_temp_reg_rev[3:0]×625；

正数温度值整数部分：temp_hex_int＝ds_temp_reg[10:4]；

负数温度值整数部分：temp_hex_int＝ds_temp_reg_rev[10:4]。

其中，

$$ds_temp_reg_rev=\sim ds_temp_reg+1$$

符号位为 ds_temp_msb_reg[7]的值。转换为 Verilog 代码如下：

```
1  assign  ds_temp_reg = {ds_temp_msb_reg,ds_temp_lsb_reg};
2  assign  ds_temp_reg_rev = (~ds_temp_reg) + 1;
3
4  always   @(posedge clk or negedge rst_n)begin
5       if(rst_n == 1'b0)begin
6           temp_hex_dot <= 0;
7       end
8       else if(opcode_vld && opcode_instr == OP_TEMP_CHANGE_EN) begin
9           temp_hex_dot<=
10              (temp_msb_reg[7]? ds_temp_reg_rev[3:0]:ds_temp_reg[3:0]) * 625;
11      end
12 end
13
14 always   @(posedge clk or negedge rst_n)begin
15      if(rst_n == 1'b0)begin
16          temp_hex_int <= 0;
17      end
18      else if(opcode_vld && opcode_instr == OP_TEMP_CHANGE_EN) begin
19          temp_hex_int <=
20              (ds_temp_msb_reg[7]? ds_temp_reg_rev[10:4]:ds_temp_reg[10:4]);
21      end
22 end
23
```

```
24 always   @(posedge clk or negedge rst_n)begin
25      if(rst_n == 1'b0)begin
26          temp_uns_sign <= 1'b0;
27      end
28      else if(opcode_vld && opcode_instr == OP_TEMP_CHANGE_EN) begin
29          temp_uns_sign <= ds_temp_msb_reg[7];
30      end
31 end
```

把温度的小数部分装换为BCD码，共有四位小数。

$$temp_uns_0dot1 <= \frac{temp_hex_dot}{1\ 000}$$

$$temp_uns_0dot01 <= \frac{temp_hex_dot}{100} \% 10$$

例如小数部分7 654，小数的最高位是7 654/1000=7，次高位为7 654/100=76，76%10=6，以此类推。Verilog代码如下：

```
1  always   @(posedge clk or negedge rst_n)begin
2      if(rst_n == 1'b0)begin
3          temp_uns_0dot1 <= 0;
4      end
5      else begin
6          temp_uns_0dot1 <= temp_hex_dot/1000;
7      end
8  end
9  always   @(posedge clk or negedge rst_n)begin
10     if(rst_n == 1'b0)begin
11         temp_uns_0dot01 <= 0;
12     end
13     else begin
14         temp_uns_0dot01 <= (temp_hex_dot/100) % 10;
15     end
16 end
17 always   @(posedge clk or negedge rst_n)begin
18     if(rst_n == 1'b0)begin
19         temp_uns_0dot001 <= 0;
20     end
21     else begin
22         temp_uns_0dot001 <= (temp_hex_dot/10) % 10;
```

```
    end
end
always   @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        temp_uns_0dot0001 <= 0;
    end
    else begin
        temp_uns_0dot0001 <= (temp_hex_dot % 10);
    end
end
```

将温度的整数部分转换为 BCD 码，共有三位整数，计算方法与小数位类似。代码如下：

```
always   @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        temp_uns_1 <= 0;
    end
    else begin
        temp_uns_1 <= temp_hex_int % 10;
    end
end

always   @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        temp_uns_10<= 0;
    end
    else begin
        temp_uns_10<= (temp_hex_int/10) % 10;
    end
end
always   @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        temp_uns_100 <= 0;
    end
    else begin
        temp_uns_100 <= (temp_hex_int/100) % 10;
    end
end
```

最终转换得到的 32 位的温度值。代码如下：

```
assign  temp_uns = {{4{temp_uns_sign}},
                    temp_uns_100,
                    temp_uns_10,
                    temp_uns_1,
                    temp_uns_0dot1,
                    temp_uns_0dot01,
                    temp_uns_0dot001,
                    temp_uns_0dot0001};
```

(6) 温度报警开关功能

当收到指令的格式为 8'h05 时，指令的数据部分的最低位用于判断是否打开温度报警功能。代码如下：

```
parameter  OP_TEMP_CMP_L   =   8'h05;

//根据 opcode_data 最低位，判断温度报警的开关
always   @(posedge clk or negedge rst_n)begin
      if(rst_n == 1'b0)begin
          beep_en <= 1'b0;
      end
      else if(opcode_vld && opcode_instr == OP_BEEP_EN) begin
         beep_en <= opcode_data[0];
      end
end
```

(7) 温度比较功能

在明德扬开发板上 beep==1 表示蜂鸣器不响；beep==0 表示蜂鸣器响。

当温度的数据大于 temp_cmp_l(下限)并且小于 temp_cmp_h(上限)时，蜂鸣器不响。此处还要注意正负数的比较，这样产生 beep 的情况将有很多种。例如仅是四个信号的正负符号就有 16 种情况，然后数值比较又有很多种情况，如果都写在一个 always，那么 beep 的这个 always 必定非常庞大。可以将比较分成两部分：上限侧比较，即 temp_cmp_h 侧；下限侧比较，即 temp_cmp_l 侧比较。这样就可以将一个 always 拆分成多个 always 来实现。

上限侧：一个 always 产生是否大于 temp_cmp_h 的指示信号 temp_cmp_h_err；下限侧：一个 always 产生是否小于 temp_cmp_l 的指示信号 temp_cmp_l_err；而温度最终是否超过设置值，用另一个 always，判断 temp_cmp_h_err 和 temp_cmp_l_err 都为 0 时，就将 beep 置 1。前两个 always 都是组合逻辑，从而保证时序延迟不变。温度比较功能的逻辑判断流程图如图 7-25 所示。

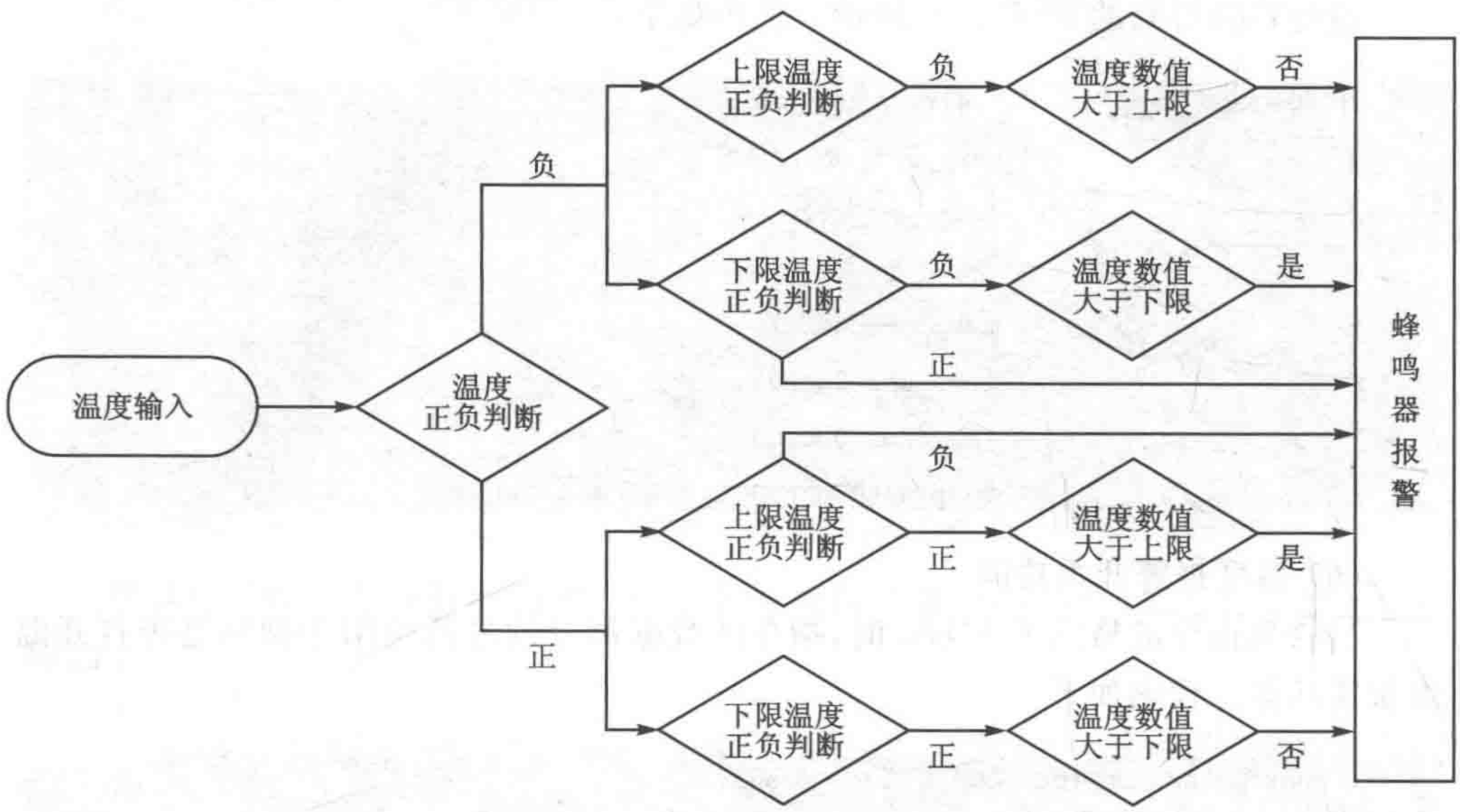


图 7-25 温度报警判断思维导图

判断是否超过上限，代码如下：

```
always   @( * )begin
    if(temp_uns_sign)begin
        if(temp_cmp_h[7] == 1'b0)
            temp_cmp_h_err = 1'b0;
        else if(temp_hex_int[6:0] > temp_cmp_h[6:0])
            temp_cmp_h_err = 1'b0;
        else
            temp_cmp_h_err = 1'b1;
    end
    else begin
        if(temp_cmp_h[7] == 1'b1)
            temp_cmp_h_err = 1'b1;
        else if(temp_hex_int[6:0] > temp_cmp_h[6:0])
            temp_cmp_h_err = 1'b1;
        else
            temp_cmp_h_err = 1'b0;
    end
end
```

判断是否超过下限，代码如下：

```
always   @( * )begin
    if(temp_uns_sign)begin
```

```
        if(temp_cmp_l[7] == 1'b0)
            temp_cmp_l_err = 1'b1;
        else if(temp_hex_int[6:0] > temp_cmp_l[6:0])
            temp_cmp_l_err = 1'b1;
        else
            temp_cmp_l_err = 1'b0;
    end
    else begin
        if(temp_cmp_l[7] == 1'b1)
            temp_cmp_l_err = 1'b0;
        else if(temp_hex_int[6:0] > temp_cmp_l[6:0])
            temp_cmp_l_err = 1'b0;
        else
            temp_cmp_l_err = 1'b1;
    end
end
```

通过 temp_cmp_h_err 和 temp_cmp_l_err 产生 beep，代码如下：

```
always   @(posedge clk or negedge rst_n)begin
        if(rst_n == 1'b0)begin
            beep <= 1'b1 ;
        end
        else if(beep_en) begin
            if(temp_cmp_l_err || temp_cmp_h_err)
                beep <= 1'b0;
            else
                beep <= 1'b1;
        end
        else begin
            beep <= 1'b1;
        end
    end
```

(8) 温度通过 UART 接口返回 PC

当 opcode_vld 有效时，根据 opcode_instr(type)的值来把不同寄存器的值送到 UART 接口输出，并产生输出有效信号。代码如下：

```
always   @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        uart_out      <= 0;
        uart_out_vld <= 1'b0;
    end
```

```
6       else if(opcode_vld) begin
7           case(opcode_instr)
8               OP_RD_CMP_L: begin
9                   uart_out <= temp_lsb_reg;
10                  uart_out_vld <= 1'b1;
11              end
12              OP_RD_CMP_H: begin
13                  uart_out <= temp_msb_reg;
14                  uart_out_vld <= 1'b1;
15              end
16              OP_RD_RDATA: begin
17                  uart_out <= ds_temp_rdata_reg;
18                      uart_out_vld <= 1'b1;
19              end
20              OP_RD_UNS_0: begin
21                  uart_out<= {{4{temp_uns_sign}},temp_uns[27:24]};
22                  uart_out_vld <= 1'b1;
23              end
24              OP_RD_UNS_1: begin
25                  uart_out <= temp_uns[23:16];
26                  uart_out_vld <= 1'b1;
27              end
28              OP_RD_UNS_2: begin
29                  uart_out <= temp_uns[15: 8];
30                  uart_out_vld <= 1'b1;
31              end
32              OP_RD_UNS_3: begin
33                  uart_out <= temp_uns[ 7: 0];
34                  uart_out_vld <= 1'b1;
35              end
36              default : begin
37                  uart_out <= 0 ;
38                  uart_out_vld <= 1'b0;
39              end
40          endcase
41      end
42      else begin
43          uart_out_vld <= 1'b0;
44      end
45  end
```

5. 顶层模块

例化 UART 的接收模块与发送模块、控制模块、数码管显示模块、温度传感器的字节模块与比特模块，形成顶层模块。注意，FPGA 与温度传感器相连的线要经过一个三态门电路进行处理，以达到单总线读/写的目的。代码如下：

```
1   module com_prj(
2                   rst_n,
3                   clk,
4                   rx_uart,
5                   tx_uart,
6                   dq      ,
7                   seg_sel,
8                   segment,
9                   beep
10  );
11  parameter       BPS       =     5208;
12
13  input               rst_n ;
14  input               clk   ;
15  input               rx_uart;
16  output              tx_uart;
17  inout               dq ;
18  output [ 7 : 0]      seg_sel;
19  output [ 7 : 0]      segment;
20  output               beep ;
21
22  wire   [7 : 0]       uart_in    ;
23  wire                 uart_in_vld;
24  wire   [7 : 0]       uart_out     ;
25  wire                 uart_out_vld;
26
27  wire   [7 : 0]       intf_rdata      ;
28  wire                 intf_rdata_vld ;
29  wire                 intf_rst_en     ;
30  wire                 intf_wr_en      ;
31  wire   [7:0]         intf_wdata      ;
32  wire                 intf_rd_en      ;
33  wire   [31:0]        temp_uns        ;
34  wire                 temp_valid_en   ;
35
36  wire                 intf_dq_out     ;
```

```
37  wire                    intf_dq_out_en ;
38  wire                    intf_dq_in     ;
39  wire                    intf_rdy       ;
40
41  //三态门
42  assign  dq    = (intf_dq_out_en)? intf_dq_out:1'bz;
43  assign  intf_dq_in = dq;
44
45  //串口接收模块
46  uart_rx#(.BPS(BPS)) u_uart_rx(
47        .clk            (clk          ),
48        .rst_n          (rst_n        ),
49        .din            (rx_uart      ),
50        .dout           (uart_in      ),
51        .dout_vld       (uart_in_vld)
52  );
53  //串口发送模块
54  uart_tx#(.BPS(BPS)) u_uart_tx(
55        .clk            (clk          ),
56        .rst_n          (rst_n        ),
57        .din            (uart_out     ),
58        .din_vld        (uart_out_vld ),
59        .rdy            (             ),
60        .dout           (tx_uart      )
61  );
62
63  //控制模块
64  control u_control(
65        .clk            (clk            ),
66        .rst_n          (rst_n          ),
67        .uart_in        (uart_in        ),
68        .uart_in_vld    (uart_in_vld    ),
69        .uart_out       (uart_out       ),
70        .uart_out_vld   (uart_out_vld   ),
71        .intf_rst_en    (intf_rst_en    ),
72        .intf_wr_en     (intf_wr_en     ),
73        .intf_wdata     (intf_wdata     ),
74        .intf_rd_en     (intf_rd_en     ),
75        .intf_rdata     (intf_rdata     ),
76        .intf_rdata_vld (intf_rdata_vld ),
77        .intf_rdy       (intf_rdy       ),
78        .beep           (beep           ),
```

```
79        .temp_uns          (temp_uns          ),
80        .temp_valid_en     (temp_valid_en     )
81    );
82
83    //温度字节接口模块
84    ds_intf_byte     u_ds_inft_byte(
85        .clk               (clk               ),
86        .rst_n             (rst_n             ),
87        .rst_en            (intf_rst_en       ),
88        .wr_en             (intf_wr_en        ),
89        .wdata             (intf_wdata        ),
90        .rd_en             (intf_rd_en        ),
91        .rdata             (uart_out          ),
92        .rdata_vld         (uart_out_vld      ),
93        .rdy               (intf_rdy          ),
94        .rdata_bit         (rdata_bit         ),
95        .rdata_vld_bit     (rdata_vld_bit     ),
96        .rdy_bit           (rdy_bit           ),
97        .rst_en_bit        (rst_en_bit        ),
98        .wr_en_bit         (wr_en_bit         ),
99        .wdata_bit         (wdata_bit         ),
100       .rd_en_bit         (rd_en_bit         )
101   );
102
103   //温度比特接口模块
104   ds_intf_bit u_ds_intf_bit(
105       .clk               (clk               ),
106       .rst_n             (rst_n             ),
107       .rst_en            (rst_en_bit        ),
108       .wr_en             (wr_en_bit         ),
109       .wdata             (wdata_bit         ),
110       .rd_en             (rd_en_bit         ),
111       .rdata             (rdata_bit         ),
112       .rdata_vld         (rdata_vld_bit     ),
113       .rdy               (rdy_bit           ),
114       .dq_out            (intf_dq_out       ),
115       .dq_out_en         (intf_dq_out_en    ),
116       .dq_in             (intf_dq_in        ),
117   );
118
119   //数码管显示模块
120   seg_disp u_seg_disp(
```

```
121     .rst_n          (rst_n          ),
122     .clk            (clk            ),
123     .disp_en        (temp_valid_en  ),
124     .din            (temp_uns       ),
125     .seg_sel        (seg_sel        ),
126     .segment        (segment        )
127 );
128
129 endmodule
```

7.6 功能仿真

1. UART 发送模块与接收模块

串口发送的数据是 8'h11，从低位到高位发送，加上起始位 1'b0 和停止位 1'b1，则输出 uart_tx 依次是 0100010001。从图 7－26 可以看出，各信号均符合要求。

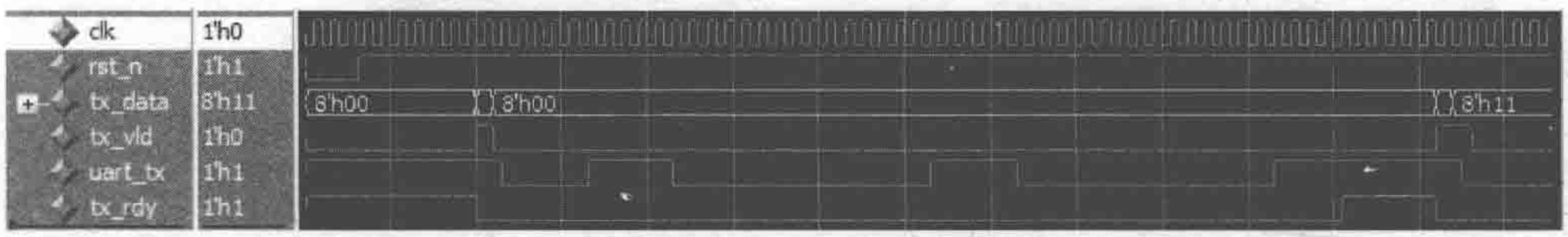


图 7－26　UART 发送模块仿真图

串口接收的数据是 011011010，其中接收到的第 1 比特是起始位，从低位到高位接收，因此最终输出结果应该是 8'b01011011。从图 7－27 可以看出，输出有效指示信号 rx_vld 在 rx_data 为 8'b01011011 时有效，即波形符合要求。

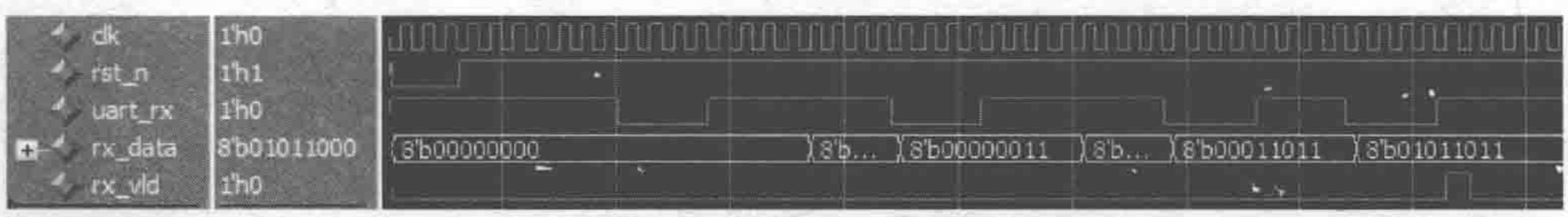


图 7－27　UART 接收模块仿真图

2. 数码管接口模块(seg_inf)

为了减少仿真时间，缩短了扫描时间计数器 cnt 的计数结束值，并不影响检查代码是否存在逻辑问题，这是一个很实用的技巧。

从图 7－28 可知，seg_tmp 会根据 sel_cnt 采下对应 din 位数的值，在下一时刻，数码管位选信号以及段选信号则分别根据 sel_cnt 与 seg_tmp 变化，与代码的期望结果符合。

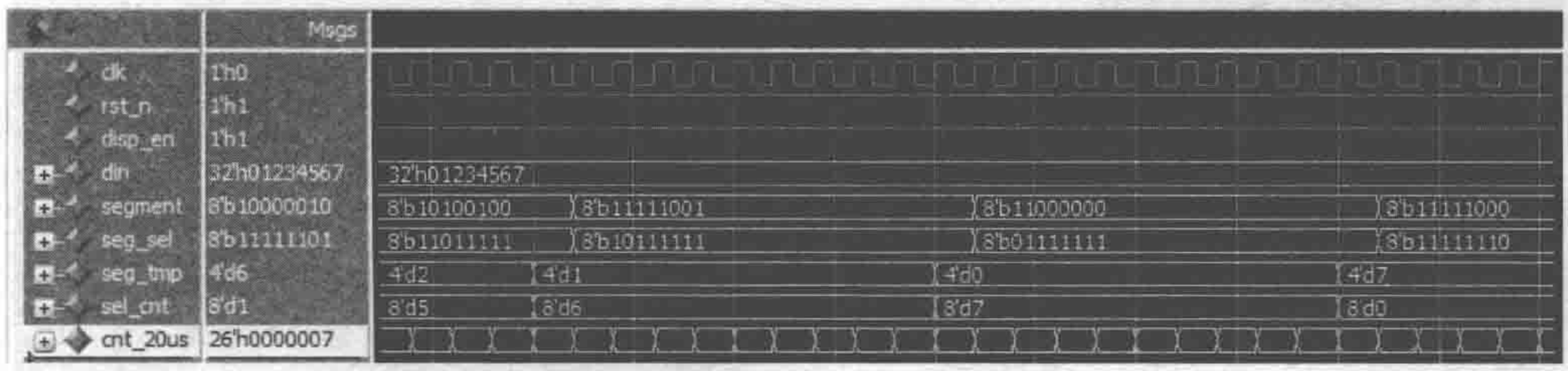


图 7－28　数码管接口模块仿真图

3. DS18B20 接口模块(temp_bit 与 temp_byte)

分别对 temp_bit 模块与 temp_byte 模块的复位、读、写时序进行仿真。

temp-bit 模块的仿真图如图 7－29～图 7－31 所示。

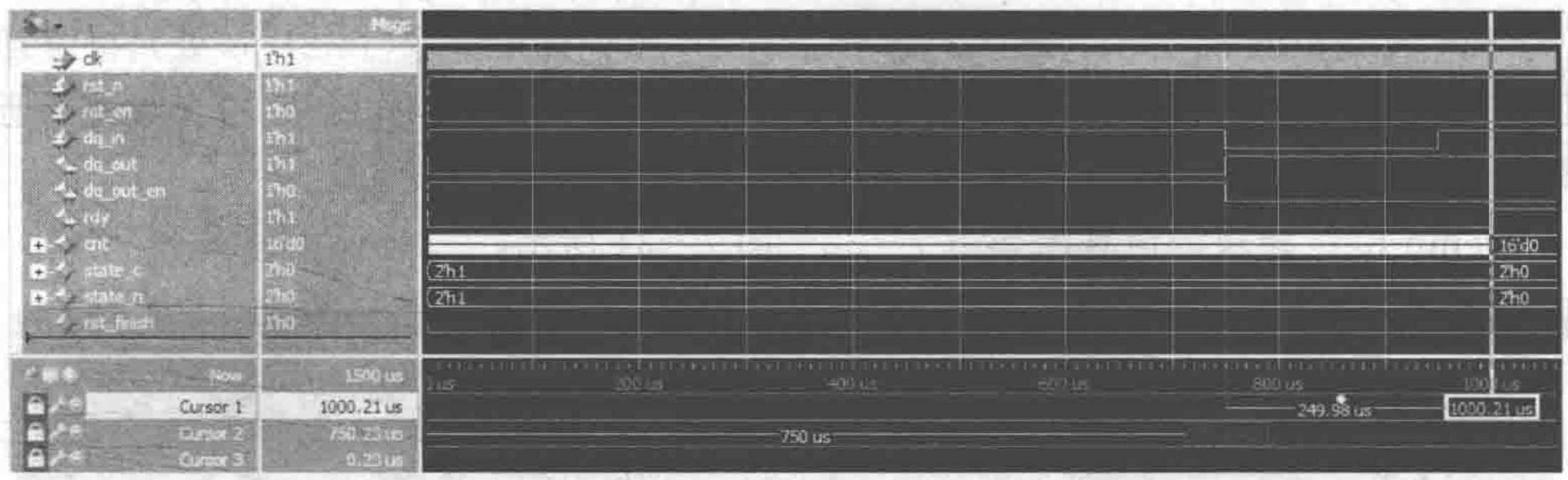


图 7－29　temp_bit 模块复位时序仿真图

从 1 μs 处 dq_out 拉低，dq_out_en 有效，持续 750 μs，然后释放，接着 dq_in 拉低 200 μs，随后释放总线，整个复位操作持续约 1 000 μs，符合数据手册要求。

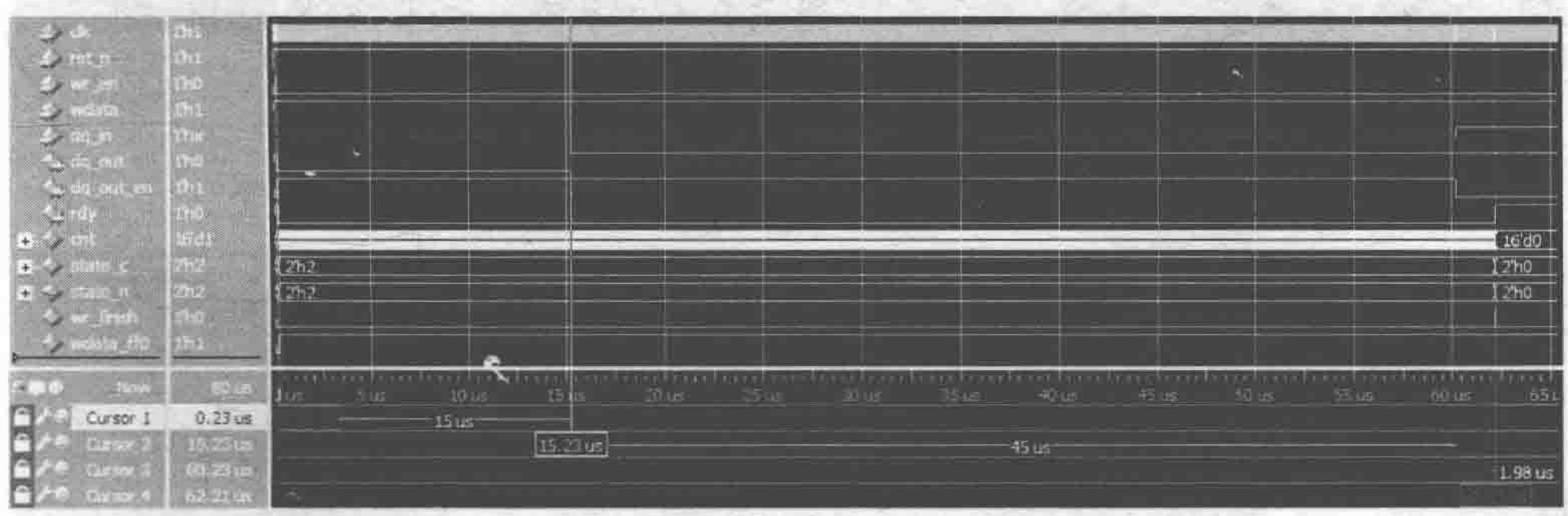


图 7－30　temp_bit 模块写 1 时序仿真图

从 15.23 μs 处开始 FPGA 拉低 dq_out，持续 15 μs，接着拉高 dq_out(写 1)，持续 45 μs，最后释放总线，dq_in 拉高约 2 μs，写 1 操作结束。整个写 1 操作持续 62 μs，符合数据手册要求。

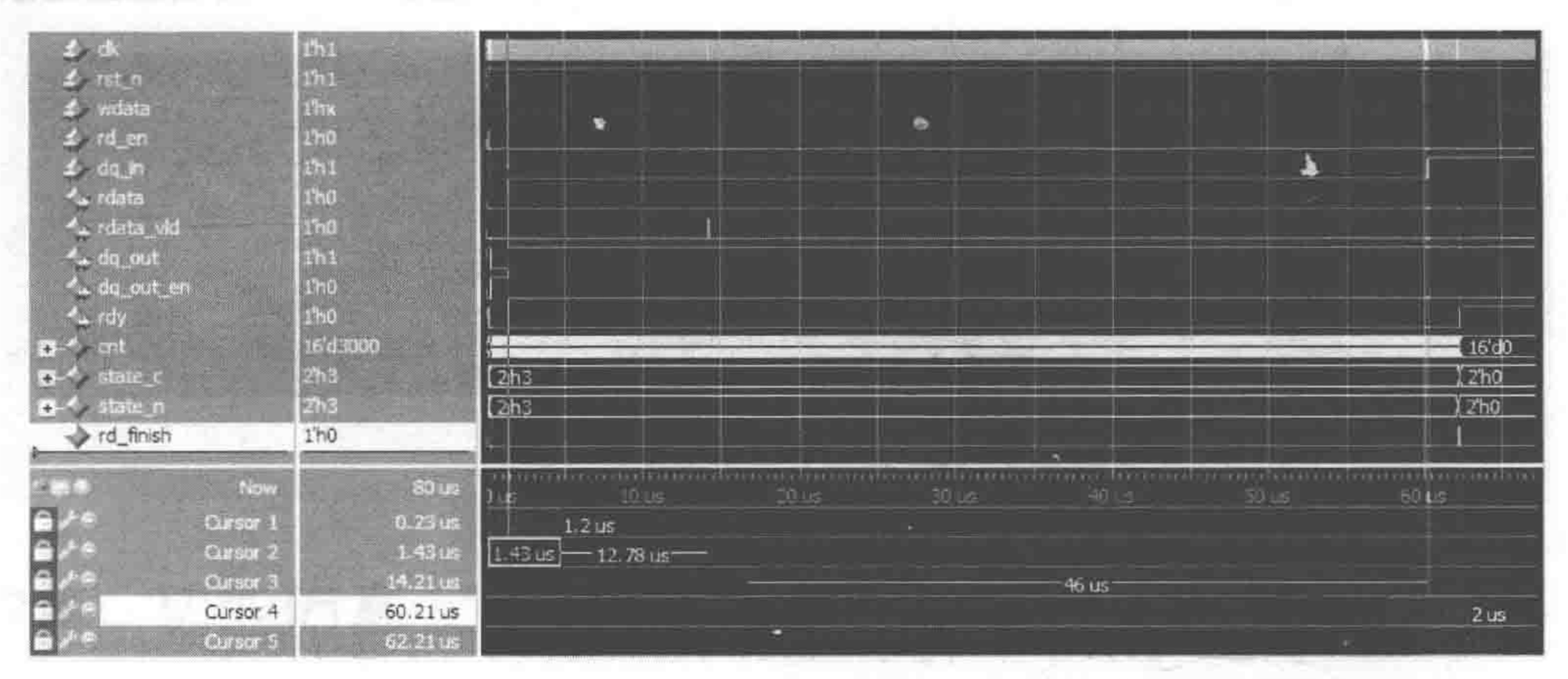


图 7－31　temp_bit 模块读 0 时序仿真图

FPGA 先拉低 dq_out，持续 1.2 μs，然后在 15 μs 内完成读取（14.21 μs 处），接着延迟 46 μs（到 60.21 μs 处），最后释放总线，dq_in 拉高 2 μs，读 0 操作结束，符合数据手册要求。

temp-byte 模块的仿真图如图 7－32～图 7－34 所示。

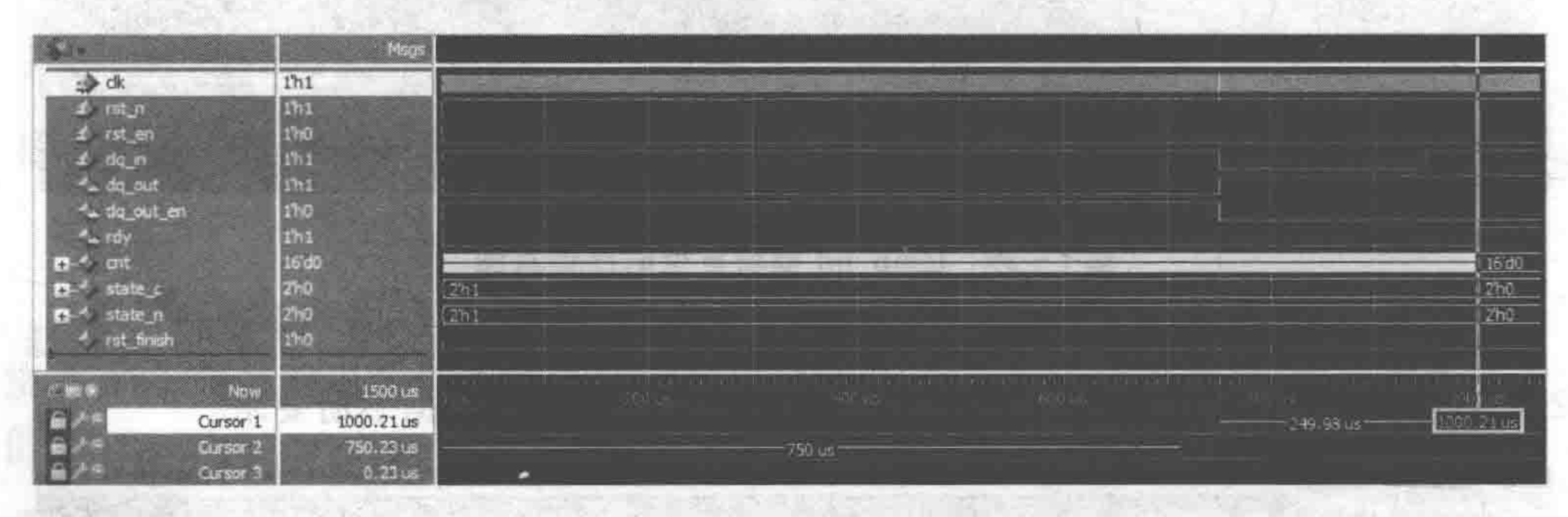


图 7－32　temp_byte 模块复位时序仿真图

temp_byte 模块的复位时序与 temp_bit 模块的复位时序是相同的。

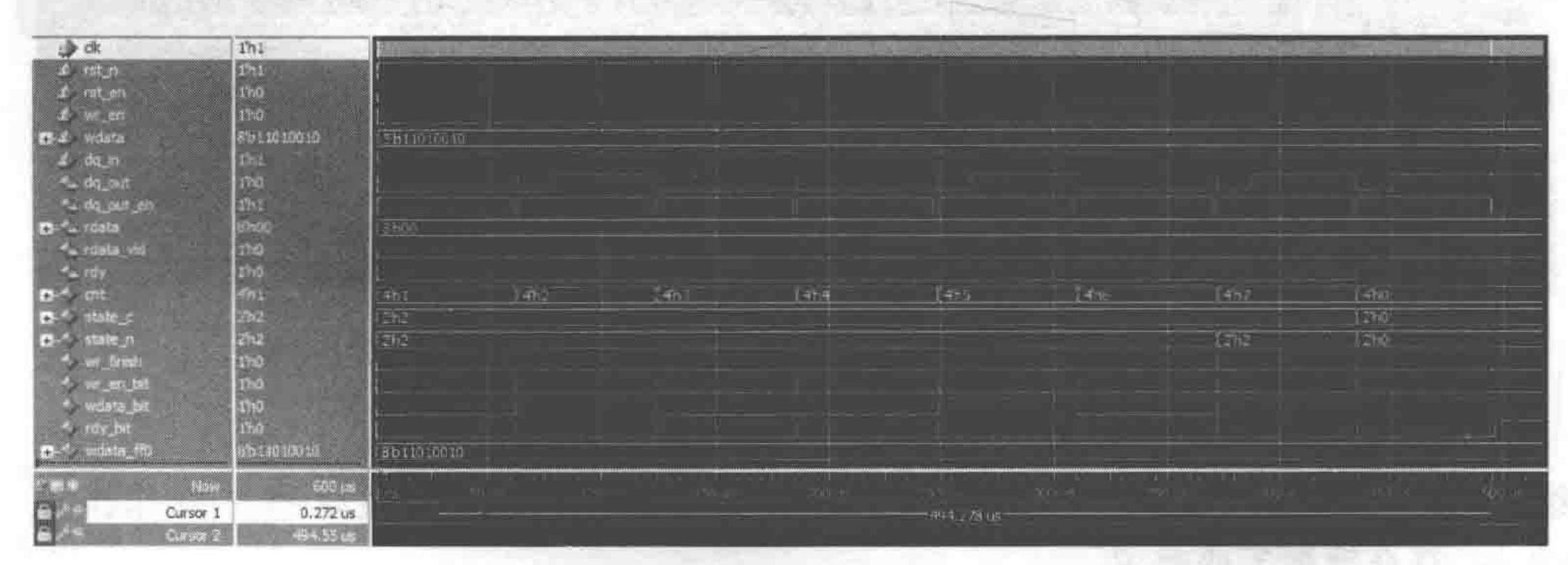


图 7－33　temp_byte 模块写时序仿真图

通过测试文件让 wdata 为 8'b11010010，从低位到高位写入 DS18B20，可以看到，dq_out 的输出依次为 0，1，0，0，1，0，1，1，而且输出期间 dq_out_en 有效，则 temp_byte 写时序正确。

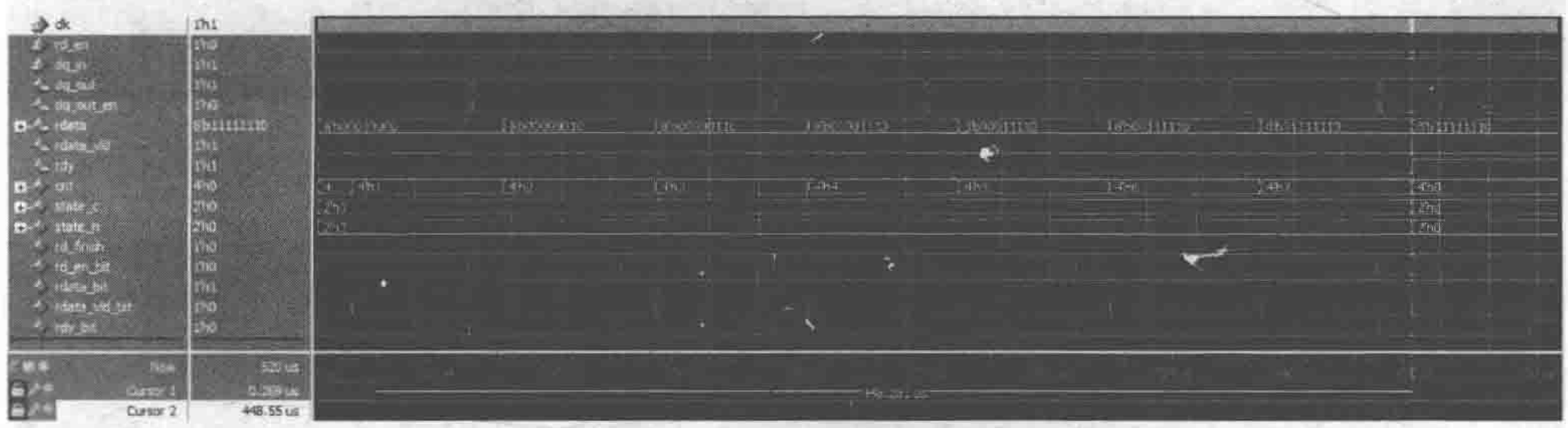


图 7－34　temp_byte 模块读时序仿真图

测试文件产生 dq_in 的值依次为 0，1，1，1，1，1，1，1，从低位到高位读，则 rdata 应为 8'b11111110，读完后 rdy 拉高且 rdata_vld 有效持续一个 clk，与图 7－34 波形一致，则 temp_byte 读时序正确。

4. 控制模块

从图 7－35 可以看出，依次给出指令 16'h8000，16'h81aa，16'h8200，16'h8300，16'h8400，控制模块依次给出复位使能、写使能和写数据 8'haa，读传感器 LSB 寄存器并保存到 ds_temp_lsb_reg 中，读传感器 MSB 寄存器并保存到 ds_temp_msb_reg 中，读传感器其他寄存器并保存到通用寄存器 ds_temp_rdata_reg 中。

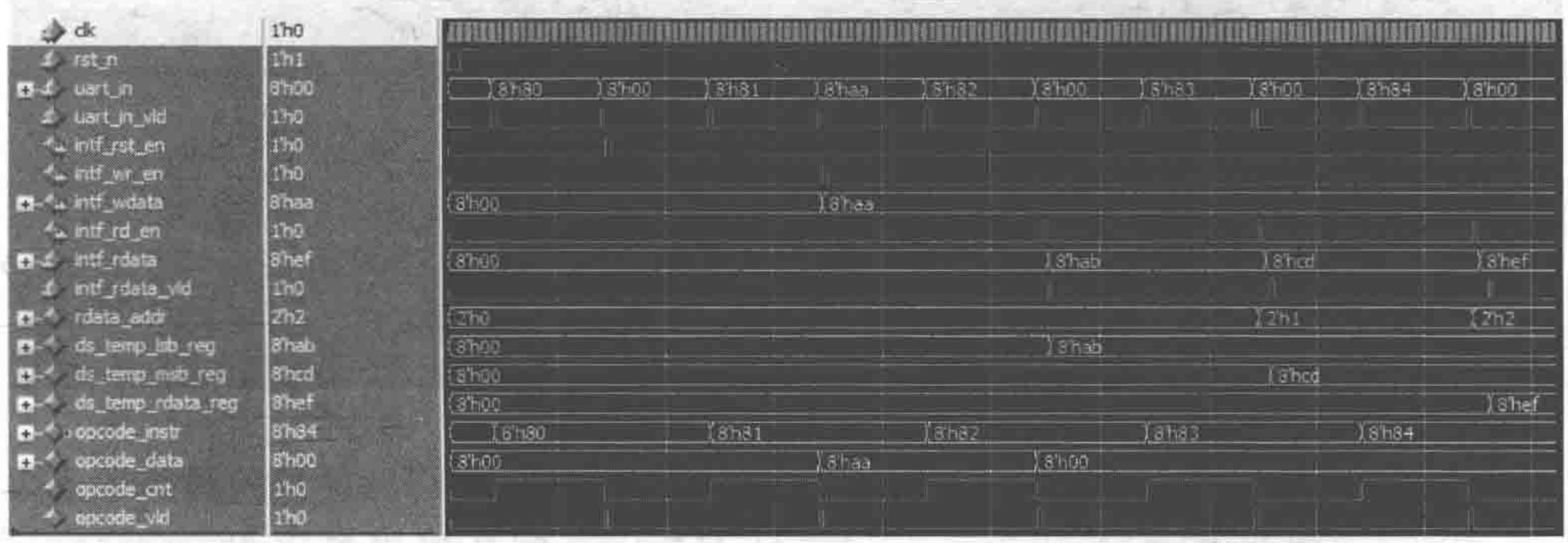


图 7－35　控制模块操作使能指令仿真图

从图 7－36 可知，依次给出指令 16'h0101，16'h0290，16'h0340，16'h0501，控制模块依次做以下操作：打开数码管显示，temp_vaild_en＝＝1；设置温度下限为 8'h90（－16 ℃）；设置温度上限为 8'h40（＋64 ℃）；打开蜂鸣器，beep_en＝＝1。

先读取温度传感器中对应寄存器的数值到 lsb_reg 与 msb_reg，分别是 8'hab 与 8'hcd，然后结合 7.3.2 小节的转换公式计算，并得出转换结果，用 32 位 BCD 码表示为 32'hf0373125，与仿真结果相同，即所设计的代码是正确的。转换结果如图 7－37 所示。

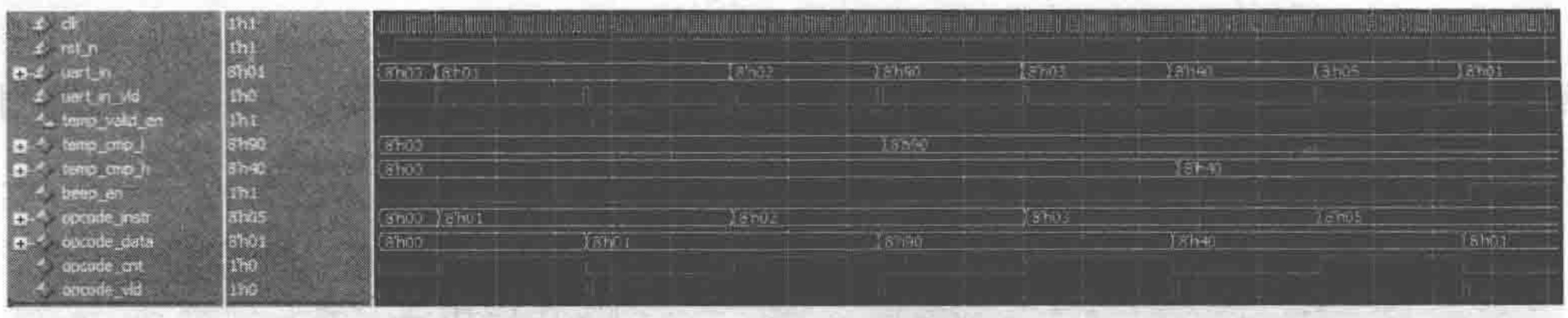


图 7－36　控制模块开关指令与温度设置指令仿真图

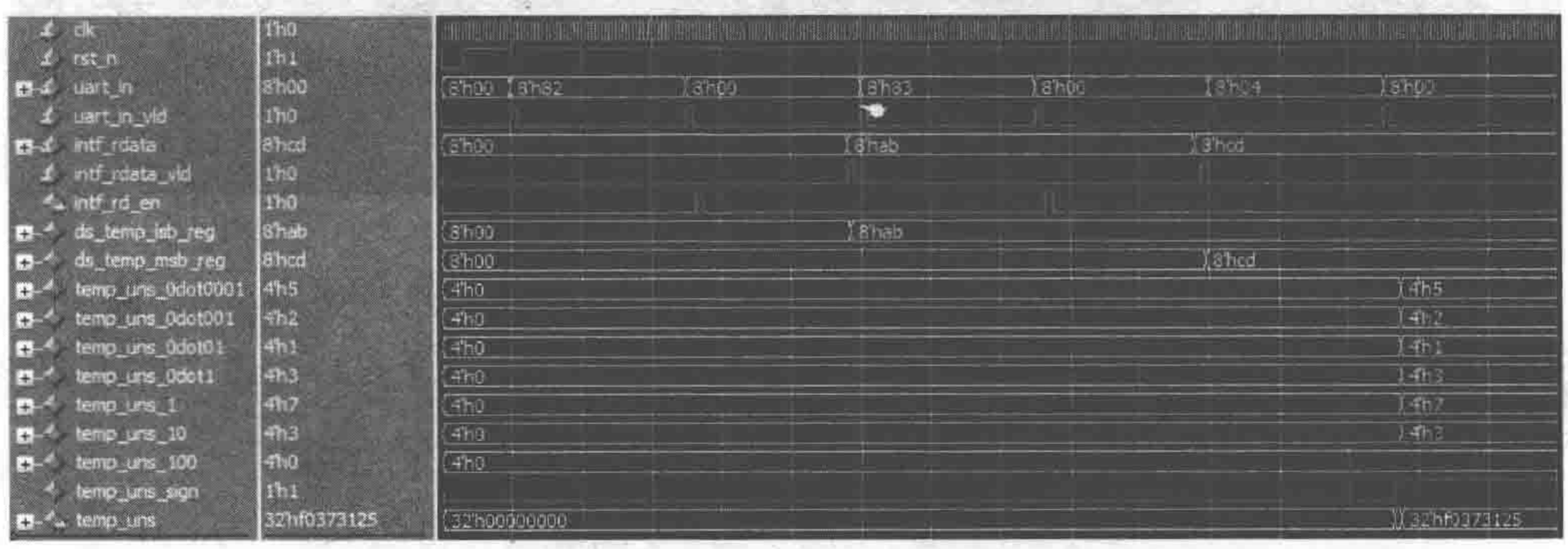


图 7－37　控制模块温度转换仿真图

如图 7－38 所示算出温度值为－37.312 5 ℃后，设置温度范围为－16～＋32 ℃，打开蜂鸣器报警使能，测量温度超过设定范围，看到图 7－38 中 beep 拉低，蜂鸣器响。

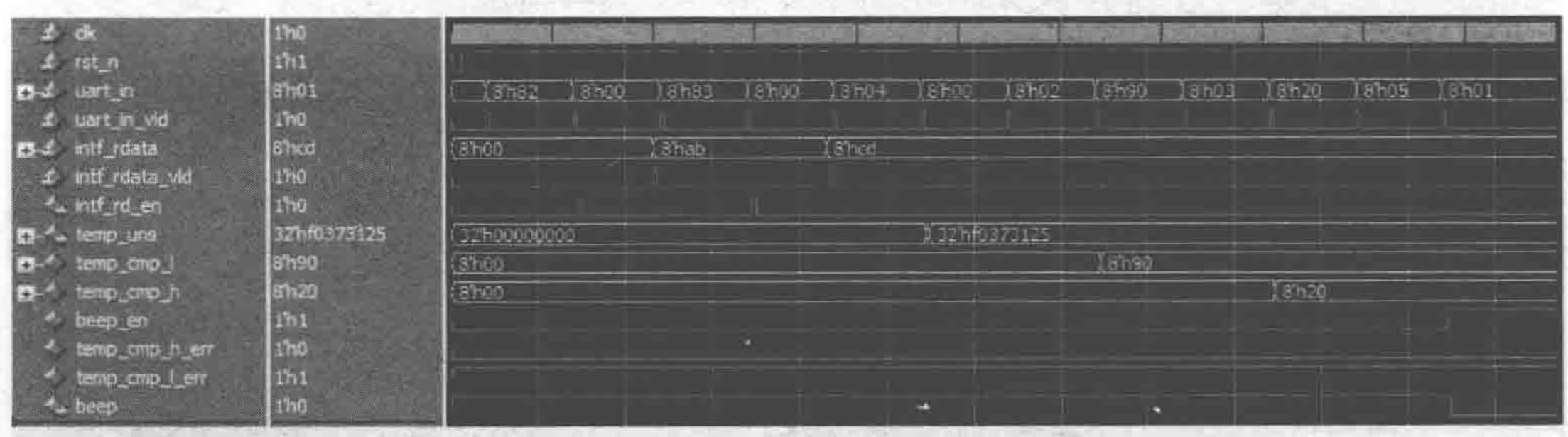


图 7－38　控制模块温度比较仿真图

如图 7－39 所示，已算出温度值为－37.312 5 ℃，依次给出指令 16’h0600，16’h0700，16’h0800，16’h0900，16’h0a00，16’h0b00，16’h0c00，从高位到低位读取 temp_uns 的值以及 lsb_reg，msb_reg 与 rdata_reg 的值。从仿真图可以看出，读取结果正确。

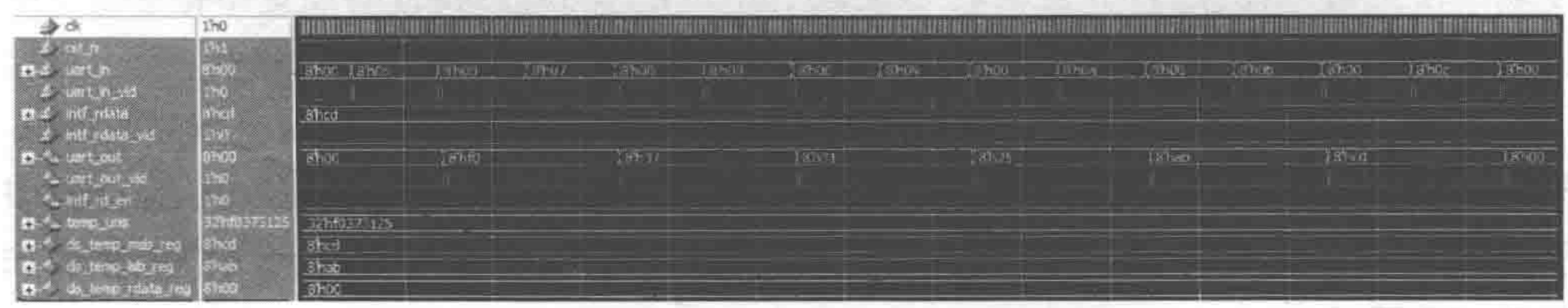


图 7－39　控制模块温度返回 PC 仿真图

7.7 板级调试

从图 7－40 的数码管显示值可以看出，当前检测到的温度值为＋28.762 4 ℃。

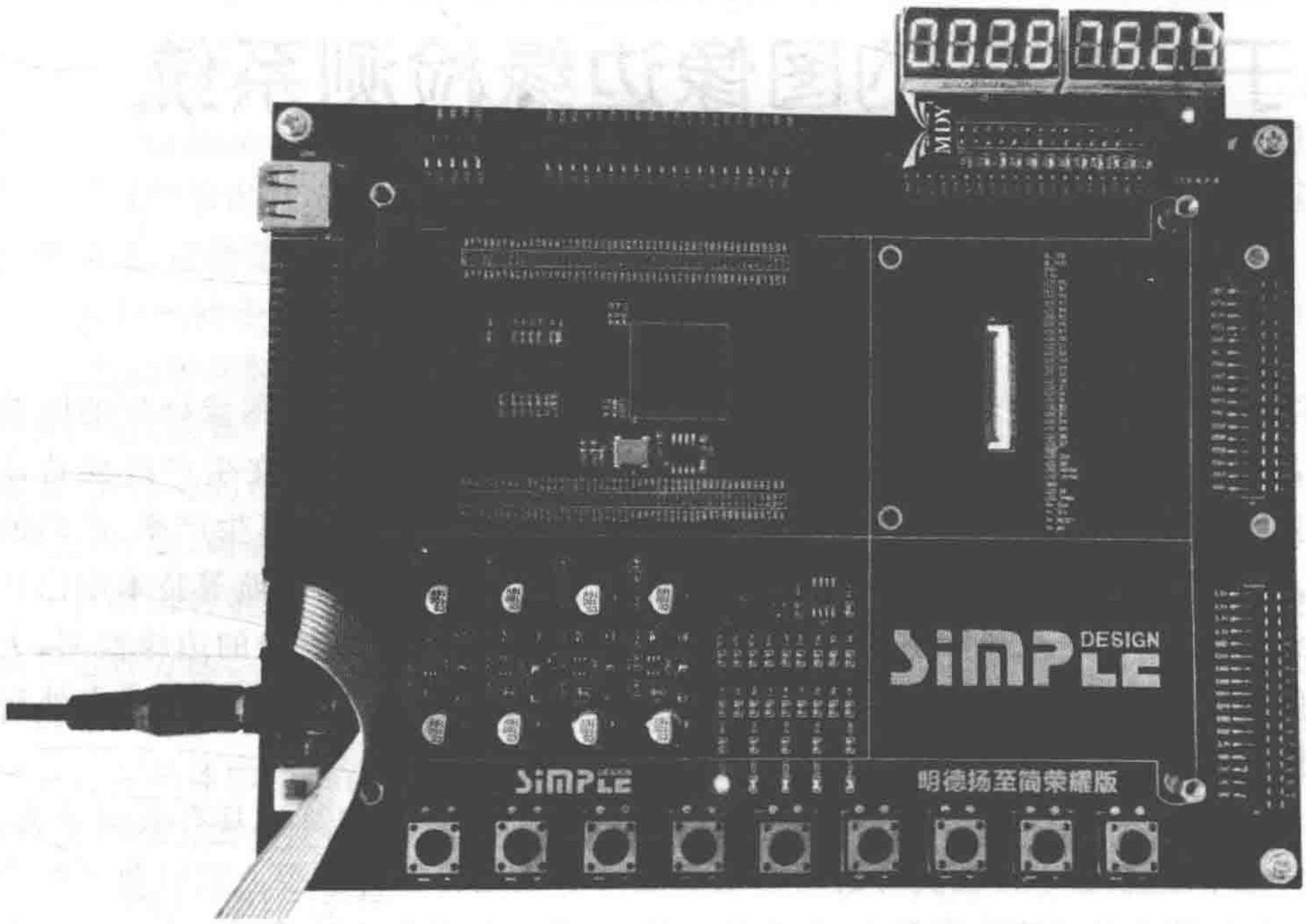


图 7－40　程序上板效果图

7.8 总　结

本设计可以分为 3 大部分，分别是 FPGA 与 PC 机通信、温度传感器采集与显示以及蜂鸣器的驱动。FPGA 与 PC 机的通信采用的串口时序，温度传感器 DS18B20 的接口时序并不复杂，而本设计的难点在于温度传感器的控制模块，要根据功能需求实现一个个功能。

读者不妨尝试根据项目要求，结合基本原理，去完成一次设计，再与书中给出的例程作对比，相信这样一定能大大提升设计能力。

第8章

基于 FPGA 的图像边缘检测系统

8.1 背景介绍

图像的边缘包含了图像大量的关键信息，图像边缘检测作为图像处理的重要内容，在航天、工业、医学、军事等领域都有重要应用。边缘检测在生产线的自动化过程中起了推动性的作用，利用对产品图像的边缘检测可以提高生产率、产品的质量，降低生产成本。边缘检测在车牌识别、车流量监控和自动导航等技术中已是重要环节，这些都是现代交通信息控制领域的重要应用。通过有效的边缘检测，大大简化了后续图像处理过程对图像的分析。所以，边缘检测技术已成为国内外研究的热点。

上面所有的应用环境都是在实时条件下的，而且边缘检测算法具有数据量大，重复度高等特点，所以开发实时的图像处理硬件系统显得非常重要。实时数字图像处理系统的特点是处理数据量大、处理速度快，这是传统软件方法所不能实现的。现场可编程门阵列（FPGA）器件的出现为数字图像处理提出了新的思路和途径。FPGA 是当今最流行的可编程器件，其丰富的逻辑和存储资源使其能够实现并行和流水线处理，利用 FPGA 设计实现并行流水的图像边缘检测，能大大提高算法实现效率，节约图像处理时间。

本项目使用常用的边缘检测算法——Sobel 算法作为本系统的边缘检测算法。

8.2 设计要求

本项目实现图 8-1 中的图像边缘检测系统。首先，通过摄像头采集实时视频图像送入 FPGA，图像在进行灰度转换之后，数据经过高斯低通滤波，然后对结果进行

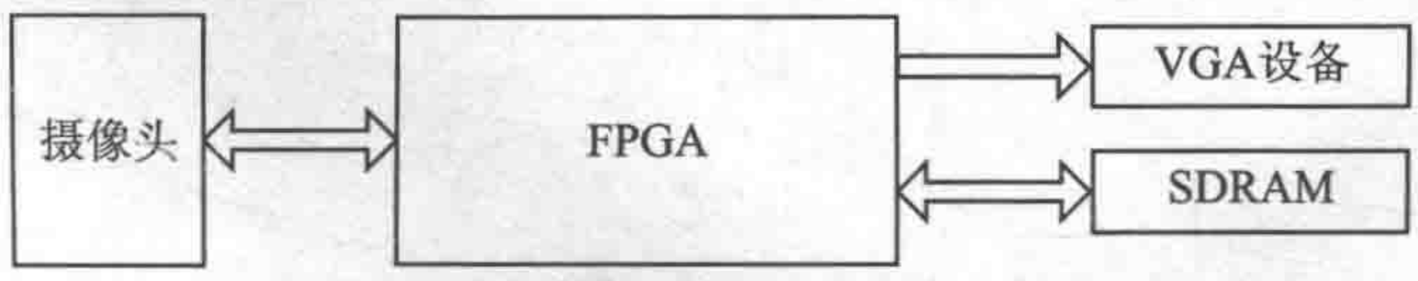


图 8-1 图像边缘检测系统

二值化处理，再用 Sobel 算子进行边缘检测，将图像数据缓存至 SDRAM 中，再从 SDRAM 中读取数据并在 VGA 设备上实时显示。

8.3 基本原理

8.3.1 PLL

PLL(Phase Locked Loop)即锁相回路或锁相环，是一种反馈控制回路，通过锁相环可以使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”的含义。

PLL 在同步时序设计中具有重要的意义。通常复杂的 FPGA 系统需要多个时钟，这些时钟在频率或相位上会有所不同，此时即可利用 PLL，根据需要通过一个时钟源分频(倍频)出其他时钟。这样一个系统中虽然有多个时钟，但归属于同一个时钟源，同样为同步时序设计。同时，经过 PLL 分频(倍频)出的时钟信号，依然是经过特殊优化的，可有效保证设计的稳定性。

在对时钟进行分频时，还有一种常用的方法是通过计数器实现。但是，明德扬不推荐这种方法。通过计数器分频出的时钟在噪声、毛刺、偏斜等方面是不可控的，同时通过计数器的方法只能实现分频(如 100 MHz 时钟分频为 50 MHz)，而无法实现倍频。在具体的项目设计中，只要所使用的 FPGA 芯片具有 PLL，就必须严格按照明德扬规范，禁止使用计数器分频出的时钟作为系统的工作时钟。

1. PLL 使用场合

在 FPGA 设计中，PLL 的使用场合比较简单，当系统需要多个时钟时，即可通过调用 PLL 来产生其他时钟。PLL 的作用可简要表述为：将输入的时钟进行分频或倍频，并且其相位偏移、占空比等都可以通过设置来控制。

在 QuartusⅡ中可直接生成 PLL 的 IP 核，但需要注意的是，PLL 是需要硬件支持的，即所选择的 FPGA 芯片需要有 PLL 模块，否则无法使用 PLL。

2. PLL 使用技巧

QuartusⅡ生成 PLL 的步骤如下：

① 打开 quartus 软件，选择菜单栏 Tools →MegaWizard Plug－in Manager，打开 IP 核生成方式管理窗口，如图 8－2 所示。

② 选择生成新的 IP 文件，单击 next 按钮，打开 IP 核类型选择窗口，按照图 8－3所示进行选择并完成 PLL 的基本设置即可。

③ 完成以上操作后单击 next 按钮，打开 IP 核参数设置窗口，此处主要设置输入时钟频率，如明德扬开发板时钟为 50 MHz，此处设置为 50 MHz，其他选择默认即可，如图 8－4 所示。

图 8－2　IP 核生成方式管理

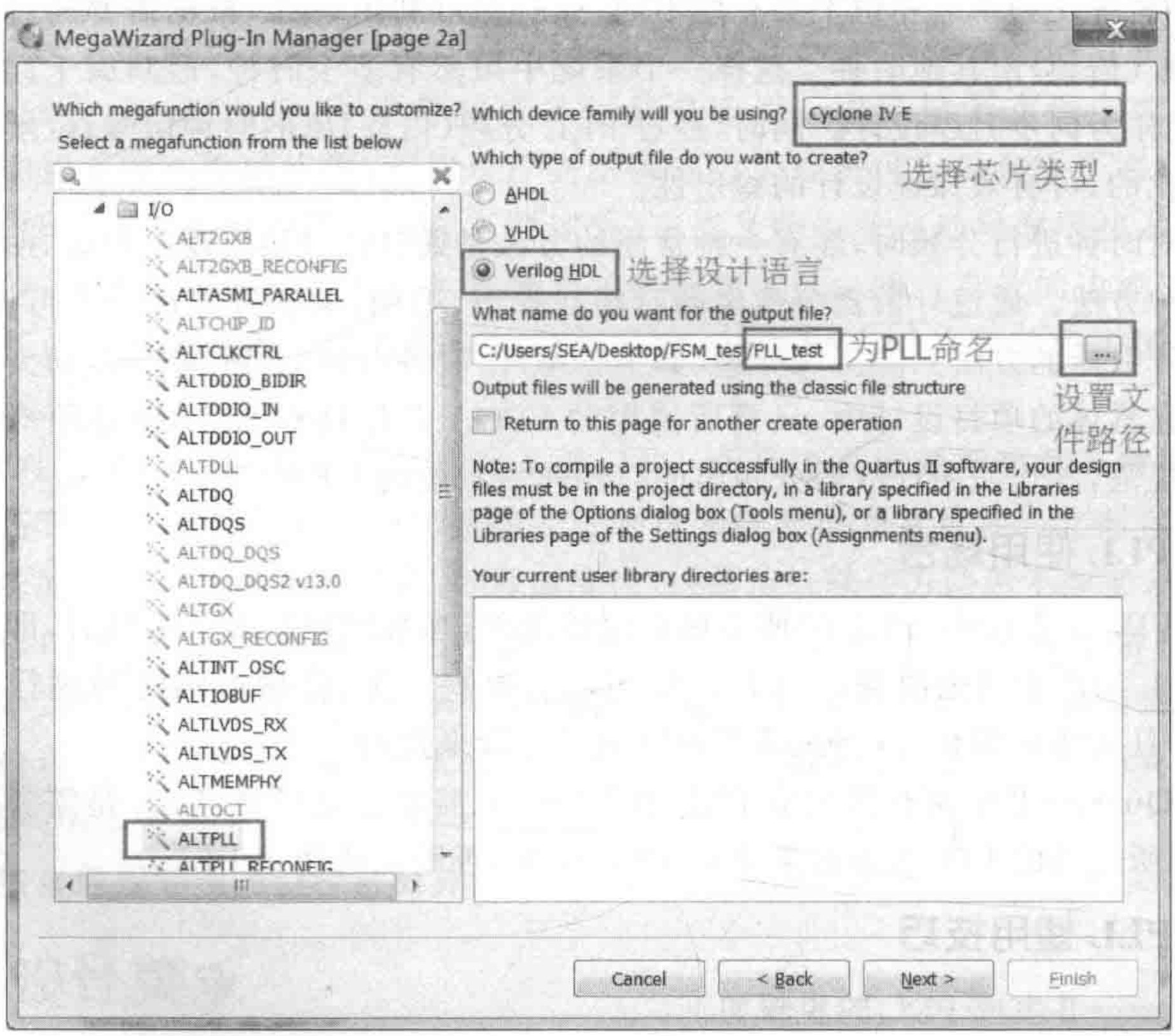


图 8－3　IP 核类型选择及其基本设置

④ 设置输入时钟频率后，打开如图 8－5 所示的窗口，进一步设置参数，此处根据需要选择是否添加复位信号“areset”、锁定指示信号“locked”。locked 信号用于指示 PLL 输出是否已稳定，高电平有效，即 locked 为 1 时表示 PLL 输出时钟已稳定可用。

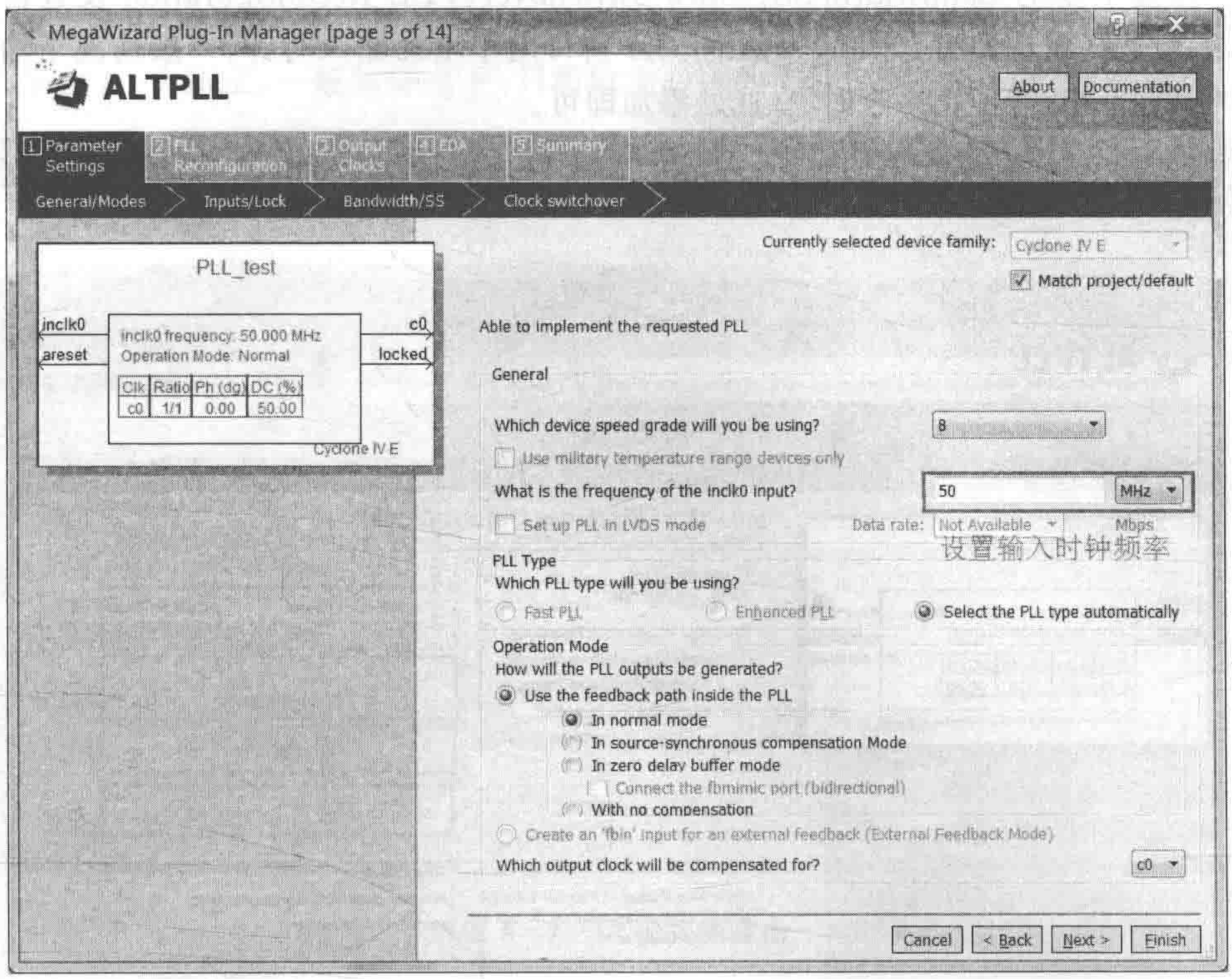


图 8－4　PLL 输入时钟频率配置

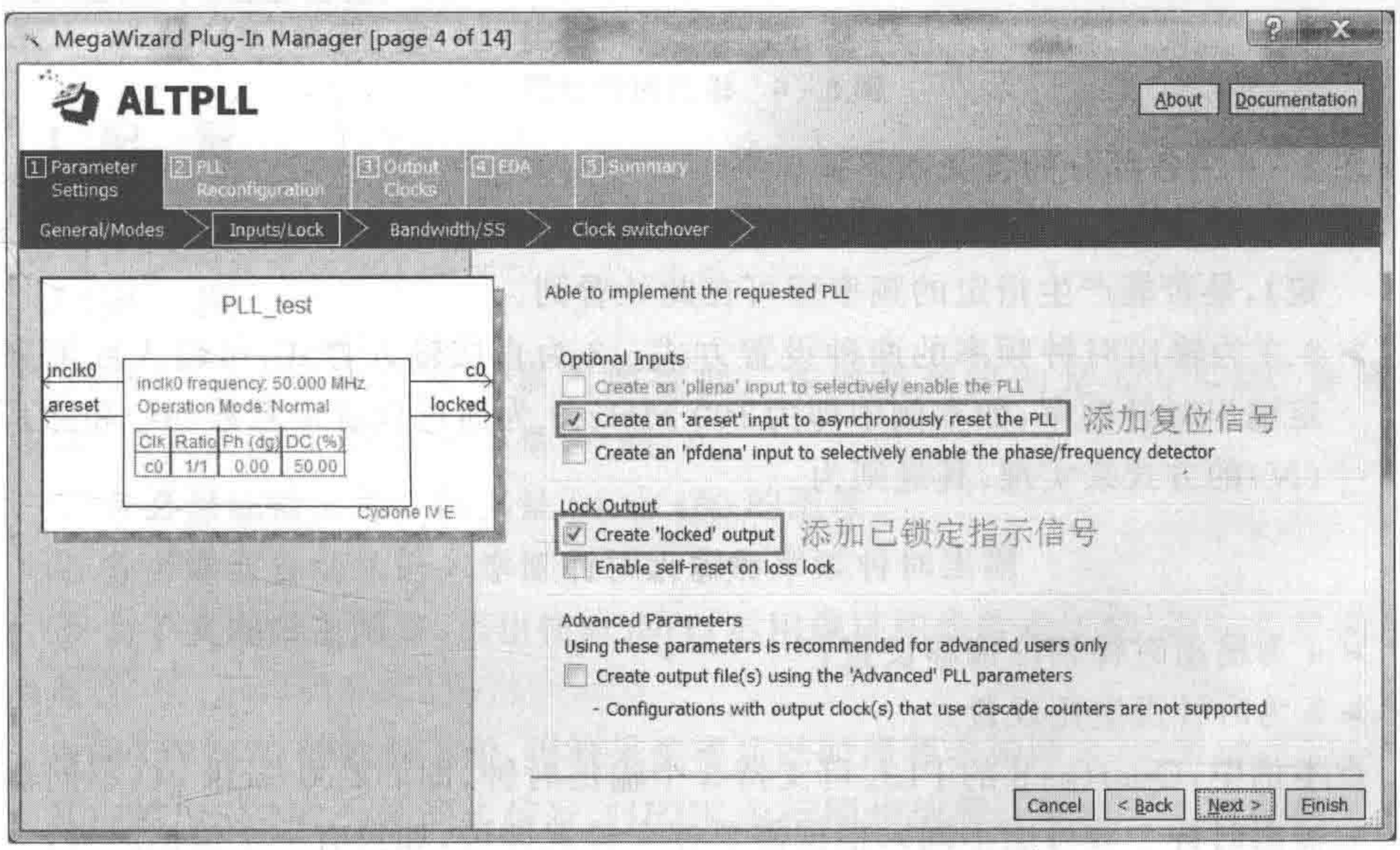


图 8－5　辅助信号设置

在接下来的 Bandwidth/SS、Clock switchover、PLL Reconfiguration 设置窗口中，一般选择默认即可。Clock switchover 窗口用于增加输入时钟，一般情况下输入时钟只有一个，当不止一个时，在此处添加即可。

⑤ 输出时钟设置。完成以上操作后，接下来开始设置输出时钟，如图 8－6 所示。

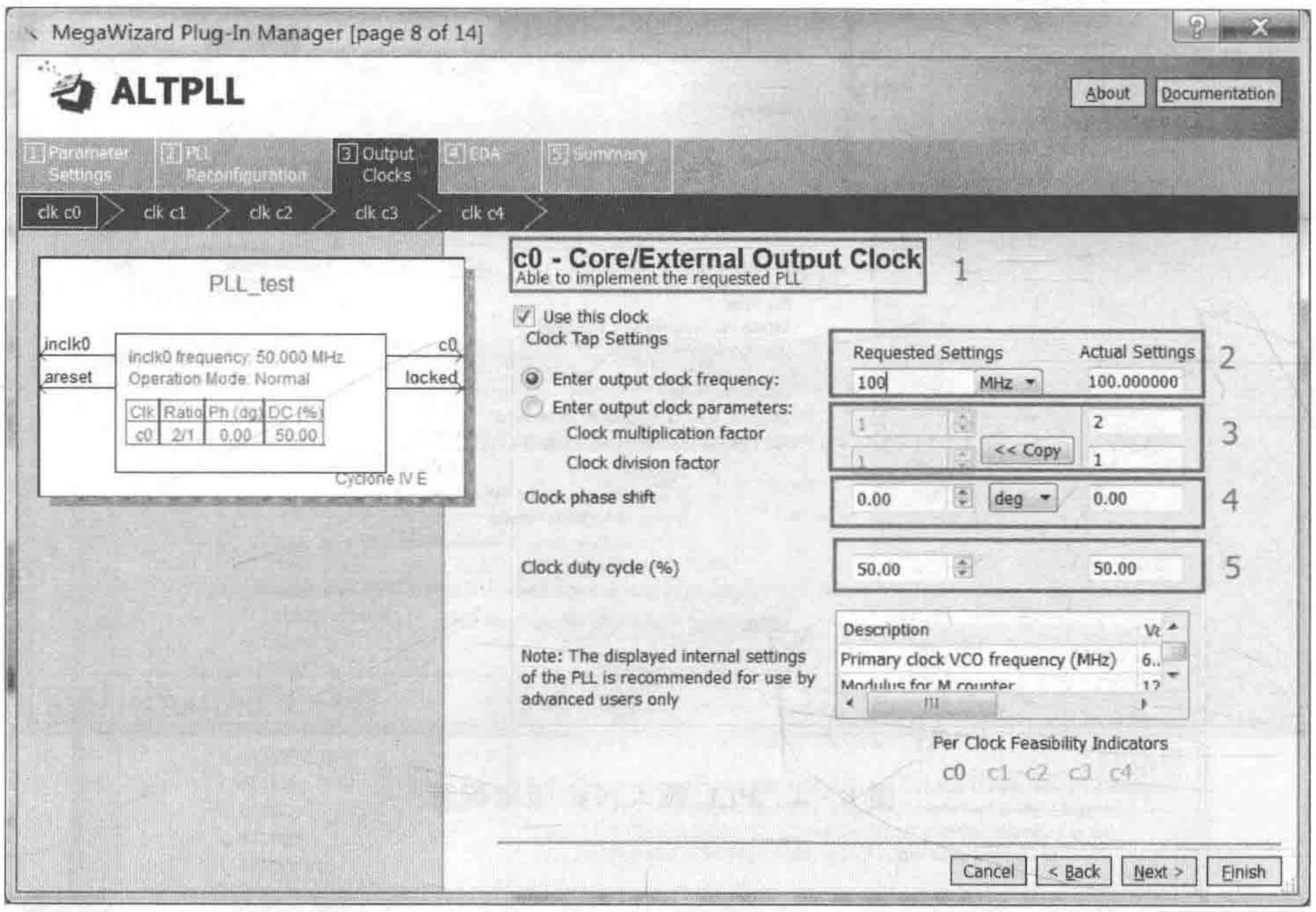


图 8－6　输出时钟设置

图 8－6 中各部分的含义如下：

- 1 为可行性指示，PLL 能产生的时钟频率有一定的范围(具体由芯片型号确定)，是否能产生指定的频率即可在此处看到。
- 2、3 为输出时钟频率的两种设置方式。2 为直接输入方式，可输入数值来指定输出时钟频率，如本例中即为 100 MHz；3 为通过设置乘系数(M)和除系数(N)的方式来实现，其规则为

$$输出时钟频率=输入时钟频率\times\frac{M}{N}$$

- 4 为输出时钟相位偏移设置；
- 5 为时钟占空比设置。

在本例中，Quartus Ⅱ 的 PLL 可支持 5 个输出时钟，即 clk c0～clk c4(不同器件支持的输出时钟个数可能不同)，根据需要逐个设置即可，如只有一个输出时钟，无需设置 clk c0，其余 clk 可直接跳过。

⑥ 输出 IP 核文件。以上操作全部完毕之后，即可根据需要输出相应的 IP 核文件，如图 8－7 所示，一般的情况下，只需输出必要的文件（V 文件、ppf 文件）即可。

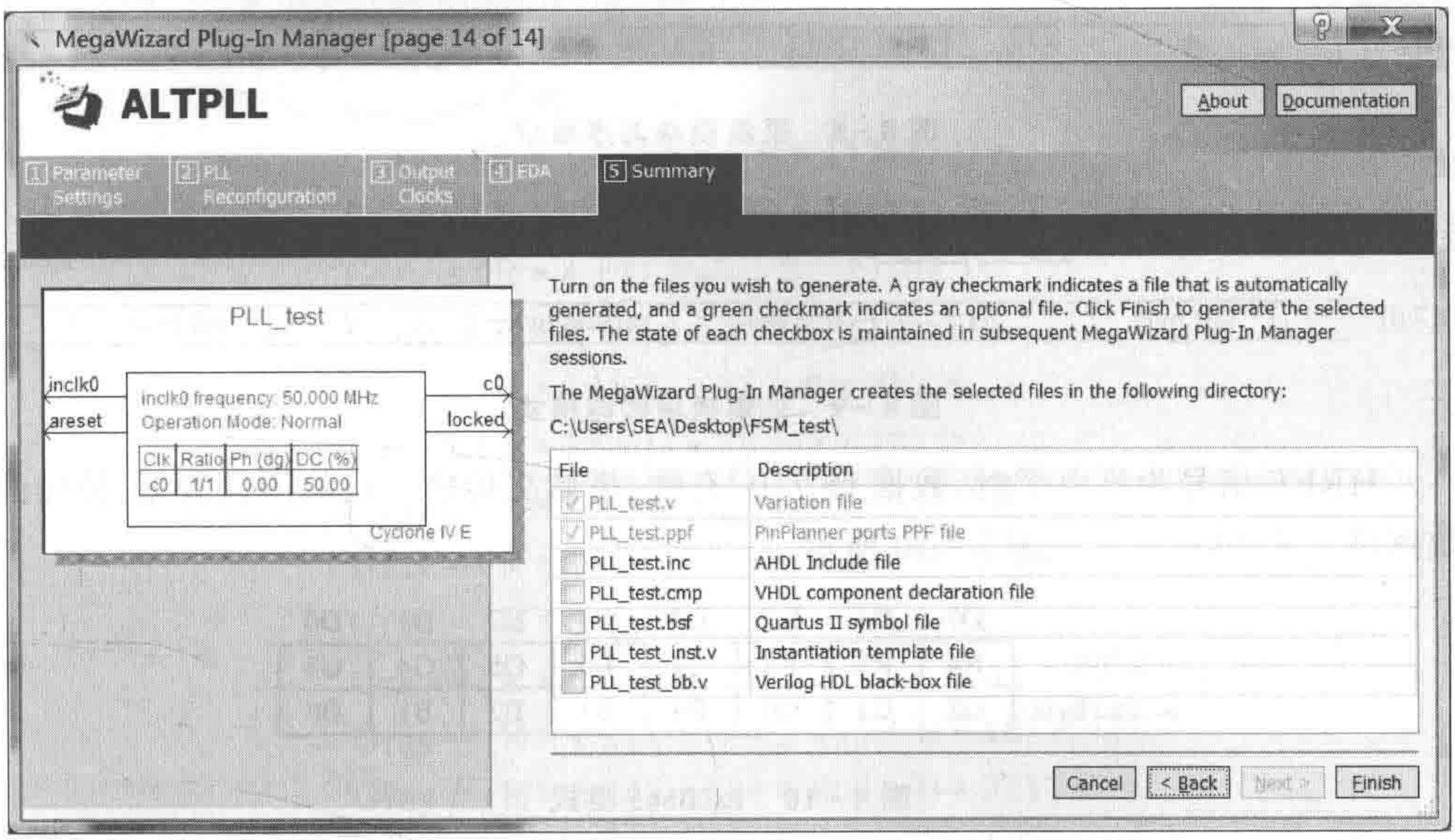


图 8－7　IP 核文件输出

⑦ 调用文件，使用 IP 核。上一步中输出的 V 文件，可以像普通 V 文件一样，通过例化的方式被其他模块调用，从而使用生成的 IP 核。

8.3.2　OV7670 配置与采集

1. 配　置

SCCB 接口见计数器八步法项目。

2. 采　集

OV7670 采集模块负责采集图像数据。本项目采集的是 640×480 分辨率、RGB565 格式的图像，图像速率是 30 帧/s。

① 务必保证输出每一帧都是 640×480 的图像。

② 务必能够自动从第一个像素开始采集。

③ 为了实现以上两点，当出错时，可以不用保证图像数据正确，但一定要满足以上两点。

查阅 OV7670 的数据手册，得到采集模块的时序要求如图 8－8 所示。

其中的 VSYNC 为帧同步信号，HREF 为行同步信号。本项目配置摄像头产生 640×480 的图像数据，因此采集一帧图像的数据（两个 VSYNC 高电平之间）会有 480 段 HREF 的高电平。采集模块数据格式如图 8－9 所示。

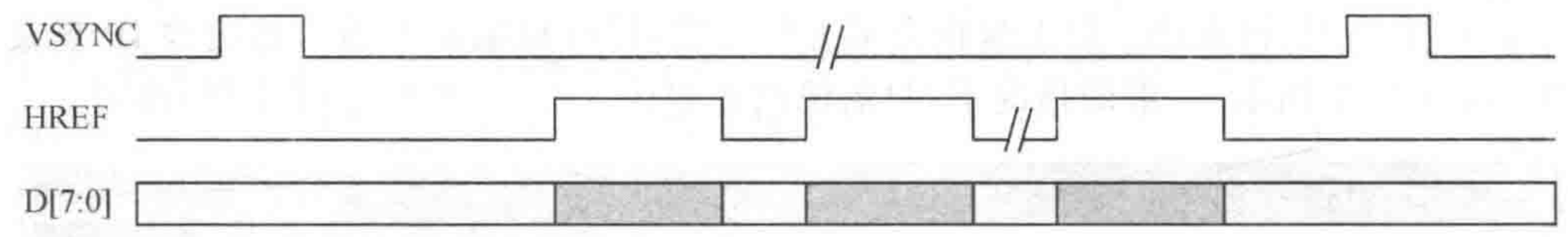


图 8-8 采集模块时序要求

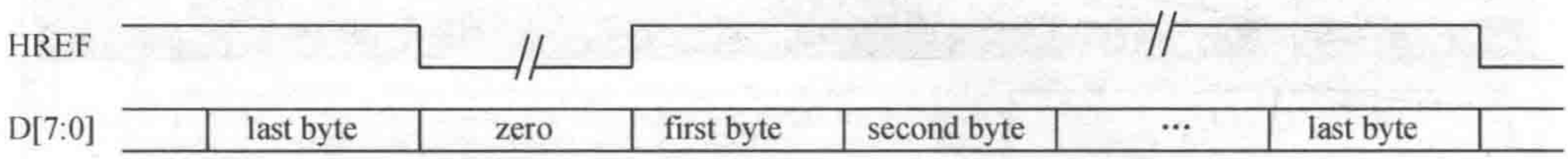


图 8-9 采集模块数据格式

HREF 信号为高电平时，数据 D[7:0]有效，摄像头传输一行的图像数据，传输的数据格式是 RGB565，如图 8-10 所示。

	D7	D6	D5	D4	D3	D2	D1	D0
first byte	R4	R3	R2	R1	R0	G5	G4	G3
second byte	G2	G1	G0	B4	B3	B2	B1	B0

图 8-10 RGB565 格式

图 8-10 中需要 2 字节(两个时钟周期)才可以完成 1 个像素点数据的采集。

8.3.3 灰度转换

1. 基 础

对于彩色转灰度，有一个很著名的色彩心理学公式为

$$Gray = R * 0.299 + G * 0.587 + B * 0.114$$

2. 整数算法

当实际应用时，希望避免低速的浮点运算，所以需要整数算法。

注意，系数都是 3 位精度，我们可以将它们缩放 1 000 倍来实现整数运算算法：

$$Gray = (R * 299 + G * 587 + B * 114 + 500)/1\ 000$$

RGB 一般是 8 位精度，现在缩放 1 000 倍，所以上面的运算是 32 位整型的运算。注意，后面那个除法是整数除法，所以需要加上 500 来实现四舍五入。

3. 整数移位算法

上面的整数算法已经很快了，但是有一点仍制约速度，就是最后的那个除法。移位比除法快多了，所以可以将系数缩放成 2 的整数幂。比如使用 16 位精度，2 的 16 次幂是 65 536，这样计算系数为

$$0.299 * 65\ 536 = 19\ 595.264 \approx 19\ 595$$

$$0.587 * 65\ 536 + (0.264) = 38\ 469.632 + 0.264 = 38\ 469.896 \approx 38\ 469$$

$$0.114 * 65\ 536 + (0.896) = 7\ 471.104 + 0.896 = 7\ 472$$

此处所使用的舍入方式不是四舍五入。四舍五入会有较大的误差，应该将以前的计算结果的误差一起计算进去，舍入方式是去尾法。写成表达式为

$$Gray = (R * 19\ 595 + G * 38\ 469 + B * 7\ 472) \gg 16$$

以下是2～16位精度的系数：

$$Gray = (R * 1 + G * 2 + B * 1) \gg 2$$
$$Gray = (R * 2 + G * 5 + B * 1) \gg 3$$
$$Gray = (R * 4 + G * 10 + B * 2) \gg 4$$
$$Gray = (R * 9 + G * 19 + B * 4) \gg 5$$
$$Gray = (R * 19 + G * 37 + B * 8) \gg 6$$
$$Gray = (R * 38 + G * 75 + B * 15) \gg 7$$
$$Gray = (R * 76 + G * 150 + B * 30) \gg 8$$
$$Gray = (R * 153 + G * 300 + B * 59) \gg 9$$
$$Gray = (R * 306 + G * 601 + B * 117) \gg 10$$
$$Gray = (R * 612 + G * 1\ 202 + B * 234) \gg 11$$
$$Gray = (R * 1\ 224 + G * 2\ 405 + B * 467) \gg 12$$
$$Gray = (R * 2\ 449 + G * 4\ 809 + B * 934) \gg 13$$
$$Gray = (R * 4\ 898 + G * 9\ 618 + B * 1\ 868) \gg 14$$
$$Gray = (R * 9\ 797 + G * 19\ 235 + B * 3\ 736) \gg 15$$
$$Gray = (R * 19\ 595 + G * 38\ 469 + B * 7\ 472) \gg 16$$

仔细观察上面的表格，某些精度实际上是一样的：3与4、7与8、10与11、13与14。其中2位精度的系数非常有意思，完全可以移位优化：

$$Gray = (R + G \ll 1 + B) \gg 2$$

由于误差很大，所以做图像处理绝不用该公式（最常用的是16位精度）。但对于游戏编程，场景经常变化，用户一般不可能观察到颜色的细微差别，所以最常用的是2位精度。

而本项目采用的计算公式是使用8位精度：

$$Gray = (R * 76 + G * 150 + B * 30) \gg 8$$

8.3.4 高斯滤波

消除图像中的噪声成分称为图像的平滑化或滤波操作。信号或图像的能量大部分集中在幅度谱的低频段和中频段是很常见的，而在较高频段，感兴趣的信息经常被噪声淹没。因此，一个能降低高频成分幅度的滤波器就能够减弱噪声的影响。

滤波的目的有两个：一是抽出对象的特征作为图像识别的特征模式；另一个是为满足图像处理的要求，消除图像数字化时所混入的噪声。

对滤波处理的要求有两条：一是不能损坏图像的轮廓及边缘等重要信息；二是使图像清晰视觉效果好。

图像的滤波方法很多，主要可以分为频率域法和空间域法两大类。频率域法的处理是在图像的某种变换域内，对图像的变换系数值进行运算，然后通过逆变换获得增强图像。这是一种间接的图像滤波方法。空间滤波方法是一类直接的滤波方法，它在处理图像时直接对图像灰度作运算。

频率域滤波是将图像从空间域或时间域转换到频率域，再利用变换系数反映某些图像特征的性质进行图像滤波的方法。傅里叶变换是一种常用的变换。在傅里叶变换域，频谱的直流分量正比于图像的平均亮度，噪声对应于频率较高的区域，图像实体位于频率较低的区域。图像在变换时所具有的这些内在特性可用于图像滤波。可以构造一个低通滤波器，使低频分量顺利通过而有效地阻止高频分量，即可滤除图像的噪声，再经过反变换来取得平滑的图像。

低通的数学表达式如下式：

$$G(u,v)=F(u,v)*H(u,v)$$

式中：$F(u,v)$是含有噪声的原图像的傅里叶变换；

$H(u,v)$为传递函数，也称转移函数（即低通滤波器的系统函数）；

$G(u,v)$为经低通滤波后输出图像的傅里叶变换。

低通滤波滤去高频成分，而低频信息基本无损失地通过。滤波后，经傅里叶变换反变换可得平滑图像，选择适当的传递函数 $H(u,v)$，与频率域低通滤波的效果关系很大。常用的传递函数有梯形函数、指数函数、巴特沃思函数等。这些传递函数，都能在图像内有噪声干扰成分时起到改善的作用。

常用的平面空间域滤波法有两类：一类是拟合图像的方法，包括 n 阶多项式拟合、离散正交多项式拟合、二次曲面拟合等多种方法；另一类是平滑图像的方法，包括领域平均法、中值滤波法、梯度倒数加权法、选择式掩膜法等。

高斯滤波器是平滑线性滤波器的一种，线性滤波器很适合于去除高斯噪声。而非线性滤波则很适合用于去除脉冲噪声，中值滤波就是非线性滤波的一种。平滑滤波器就是用滤波掩膜确定的邻域内像素的平均灰度值去替代图像的每个像素点的值，这很容易用硬件实现。而高斯滤波器是带有权重的平均值，即加权平均，中心的权重比邻近像素的权重更大，这样就可以克服边界效应。高斯滤波采用的 3×3 掩膜的具体公式如下：

$$\begin{aligned}g(x,y)=\{&f(x-1,y-1)+f(x-1,y+1)+\\&f(x+1,y-1)+f(x+1,y+1)+\\&[f(x-1,y)+f(x,y-1)+f(x+1,y)+\\&f(x,y+1)]\times 2+f(x,y)\times 4\}/16\end{aligned}$$

式中：$f(x,y)$为原图像中(x,y)像素点的灰度值；

$g(x,y)$为经过高斯滤波后的值。

上述的公式可以结构化为 3×3 的掩膜如图 8-11 所示。

1	2	1
2	4	2
1	2	1

×1/16

图 8-11　3×3 掩膜

从结构化掩膜中可以看到，处于掩膜中心的位置比其他

任何像素的权值都大，因此在均值计算中给定的这一像素显得更为重要。而距离掩膜中心较远的像素就显得不太重要，这样做是为了减小平滑处理中的模糊。虽然可以采取其他权重达到相同的目的，但是由于1、2、4、16都是2的整数次幂，很便于硬件的实现，其中16＝1＋2＋1＋2＋4＋2＋1＋2＋1。

8.3.5 Sobel边缘检测

索贝尔算子(Sobel operator)主要用作边缘检测，在技术上，它是一离散性差分算子，用来运算图像亮度函数的灰度之近似值。在图像的任何一点使用此算子，都会产生对应的灰度矢量或是其法矢量。Sobel卷积因子 $\boldsymbol{G}_x$、$\boldsymbol{G}_y$ 如图8－12所示。

−1	0	+1
−2	0	+2
−1	0	+1

$\boldsymbol{G}_x$

+1	+2	+1
0	0	0
−1	−2	−1

$\boldsymbol{G}_y$

图8－12　Sobel卷积因子

该算子包含两组3×3的矩阵，分别为横向和纵向，将之与图像作平面卷积，即可分别得出横向和纵向的亮度差分近似值。如果以 $\boldsymbol{A}$ 代表原始图像，$\boldsymbol{G}_x$ 和 $\boldsymbol{G}_y$ 分别代表经横向和纵向边缘检测的图像灰度值，其卷积因子计算公式如下：

$$\boldsymbol{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \boldsymbol{A} \qquad \boldsymbol{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \boldsymbol{A}$$

具体计算为

$$\begin{aligned}\boldsymbol{G}_x = {} & (-1)\times f(x-1,\ y-1)+0\times f(x,y-1)+1\times f(x+1,y-1)+ \\ & (-2)\times f(x-1,y)+0\times f(x,y)+2\times f(x+1,y)+(-1)\times f(x-1,y+1)+ \\ & 0\times f(x,y+1)+1\times f(x+1,y+1)=[f(x+1,y-1)+2\times f(x+1,y)+ \\ & f(x+1,y+1)]-[f(x-1,y-1)+2\times f(x-1,y)+f(x-1,y+1)]\end{aligned}$$

$$\begin{aligned}\boldsymbol{G}_y = {} & 1\times f(x-1,\ y-1)+2\times f(x,y-1)+1\times f(x+1,y-1)+0\times f(x-1,y) \\ & 0\times f(x,y)+0\times f(x+1,y)+(-1)\times f(x-1,y+1)+(-2)\times f(x,y+1)+ \\ & (-1)\times f(x+1,\ y+1)=[f(x-1,y-1)+2f(x,y-1)+f(x+1,y-1)]- \\ & [f(x-1,\ y+1)+2\times f(x,y+1)+f(x+1,y+1)]\end{aligned}$$

式中：$f(a,b)$表示图像(a,b)点的灰度值。图像的每一个像素点的横向和纵向灰度值均通过以下公式来计算：

$$G=\sqrt{G_x^2+G_y^2}$$

通常，为了提高效率使用不开平方的近似值：

$$|G|=|G_x|+|G_y|$$

如果梯度 G 大于某一阈值，则认为该点(x,y)为边缘点。

用以下公式还可以计算梯度的方向：

$$\theta=\arctan\left(\frac{G_y}{G_x}\right)$$

但本项目中没有使用。

使用 Sobel 算子根据像素点上下、左右邻点灰度的加权差，在边缘处达到极值这一现象来检测边缘。对噪声具有平滑作用，提供较为精确的边缘方向信息，但是边缘定位精度不够高。当对精度要求不是很高时，是一种较为常用的边缘检测方法。

8.3.6 乒乓操作

乒乓操作是一种完成数据无缝缓冲与处理的技巧，典型的乒乓操作如图 8－13 所示。

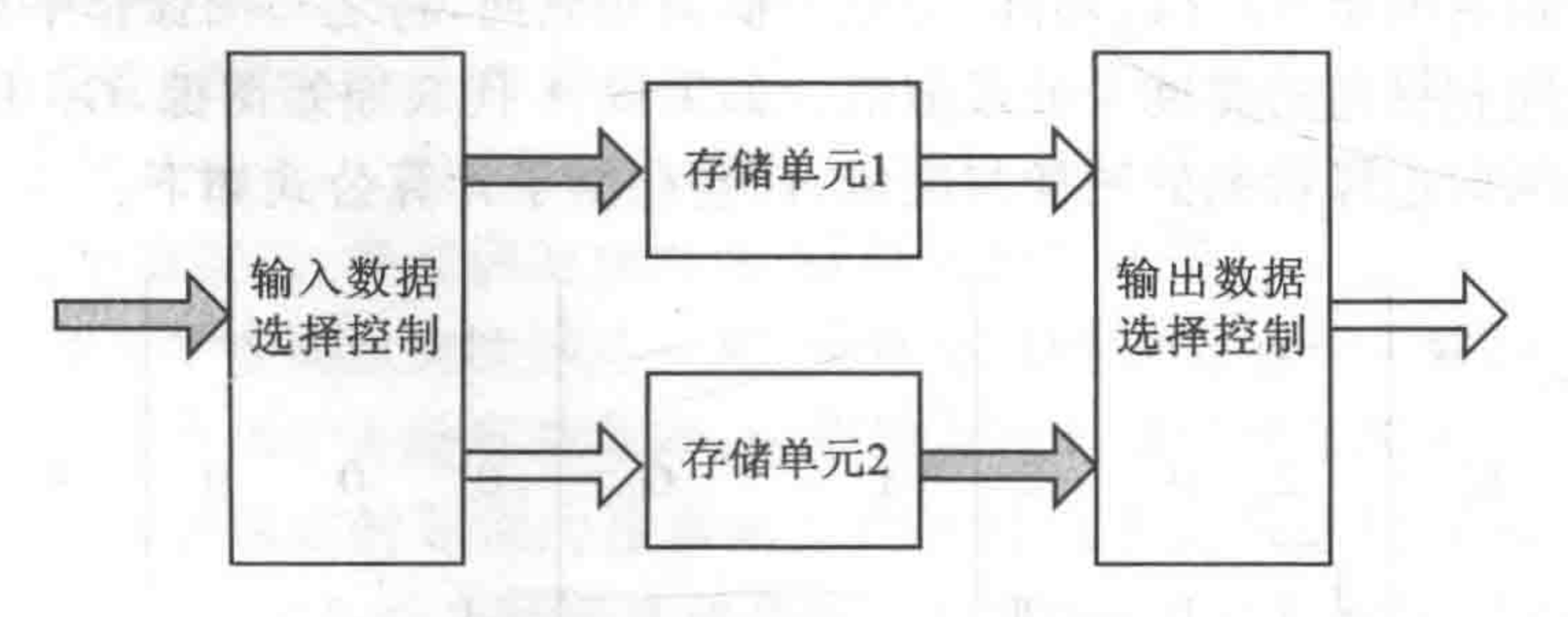


图 8－13　乒乓操作示意图

存储单元是具有存储功能的模块，本项目使用 SDRAM 作为存储单元。

乒乓操作可以分为三个阶段，其操作过程如下：

在阶段一，外部输入数据流通过“输入数据选择控制”送到“存储单元 1”。

在阶段二，外部输入数据流通过“输入数据选择控制”送到“存储单元 2”，同时“输出数据选择控制”选择“存储单元 1”，并输出数据。

在阶段三，外部输入数据流通过“输入数据选择控制”送到“存储单元 1”，同时“输出数据选择控制”选择“存储单元 2”，并输出数据。

按此顺序不断循环。

8.4 设计方案

1. 外设部分

根据设计要求完成图像的采集、存储与显示，要用到的外设有 OV7670、SDRAM 和 VGA，这三个外设可以结合 6.5 节外设交互架构进行设计。

(1) 摄像头 OV7670

在 6.6.3 小节的项目中，已经详细介绍过 OV7670 的配置与图像采集，模块划分结果如图 8－14 所示。

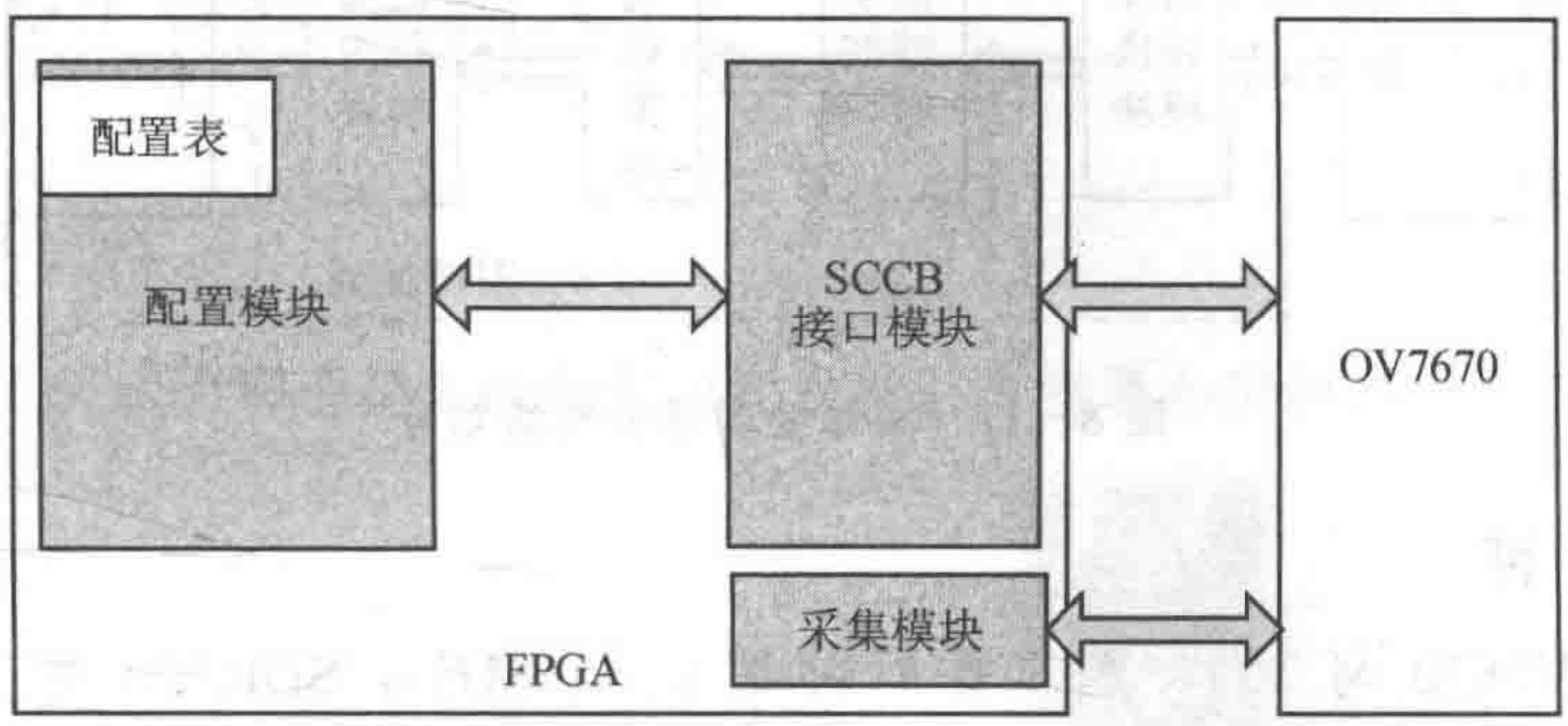


图 8－14　摄像头部分模块划分

(2) SDRAM

本项目需要存储的数据量其实并不大，用片内 RAM 就足够了。但这里用 SDRAM 存储，是为了通过本项目将 3.2.3 小节的 SDRAM 接口项目应用到实际项目中，以便读者掌握 SDRAM 的使用。

SDRAM 是不需要配置寄存器就可以正常使用的外设，所以使用不需配置交互架构。模块划分结果如图 8－15 所示。

(3) VGA

VGA 也是使用不需配置交互架构。模块划分结果如图 8－16 所示。

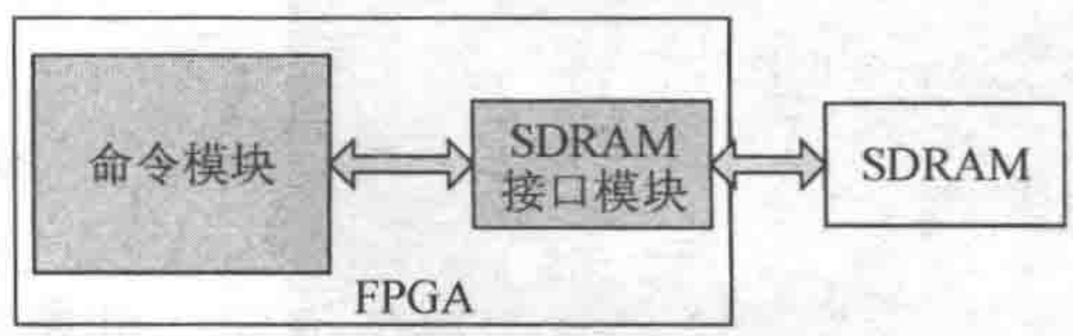


图 8－15　SDRAM 模块划分

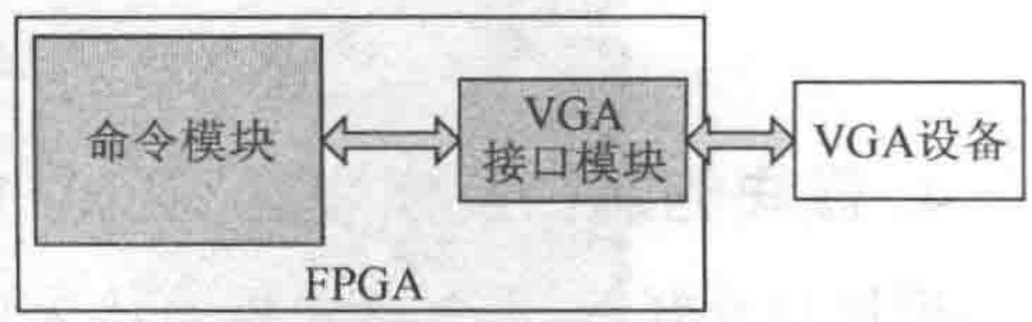


图 8－16　VGA 部分模块划分

2. 图像处理部分

在采集到摄像头的数据后，要先进行灰度转换，将彩色的图像转换成 256 灰阶的图像；因为在摄像头数据传输或采集过程中可能会混入噪声，所以对转换后的 256 灰阶图像进行高斯滤波；然后进行二值化处理，把 256 灰阶图像变成黑白图；接着用 Sobel 算子进行边缘检测，结果存入 SDRAM。

根据模块划分原则一，一个功能一个模块，可得如图 8－17 所示的模块划分结果。

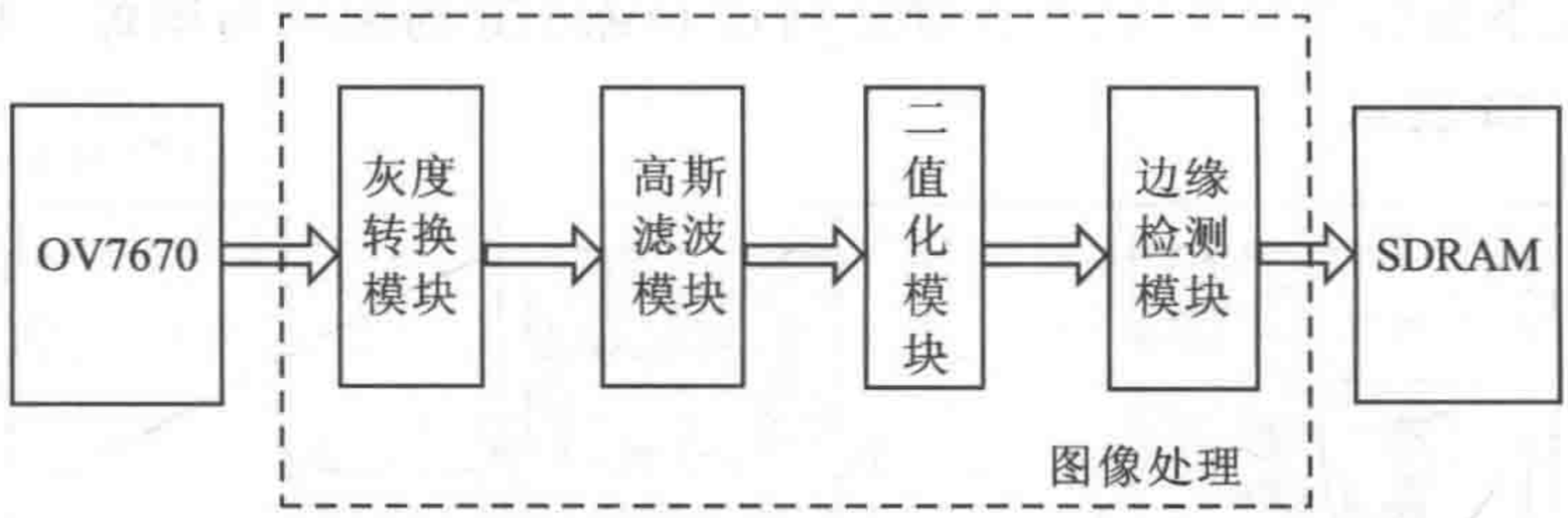


图 8－17　图像处理部分模块划分

3. 时　钟

由于 OV7670 与 VGA 的工作时钟都是 25 MHz，SDRAM 的工作时钟是 100 MHz，为方便设计，整个工程的主体时钟就设定为 25 MHz。而明德扬开发板上 FPGA 的系统时钟是 50 MHz，因此使用 PLL 的 IP 核进行分频与倍频。

PLL 的具体使用方法请参考 8.3.1 小节 PLL 的基本原理。

综上，整体模块划分结果如图 8－18 所示。

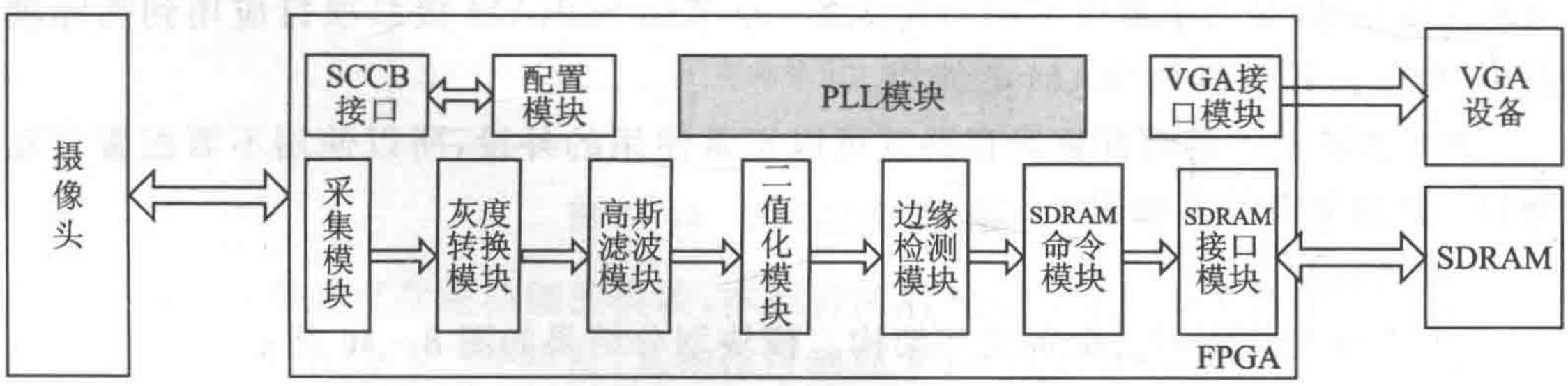


图 8－18　整体模块划分框图

4. 模块的端口及模块之间的数据流向

模块划分好后，需要进一步确认模块的端口及模块之间的数据流向。可参考 5.1.2 小节的模块划分端口规范。模块划分最终结果如图 8－19 所示，由于内部模块较多，此处没有画出内部模块的端口信号，但读者在设计前一定要明确各端口信号。

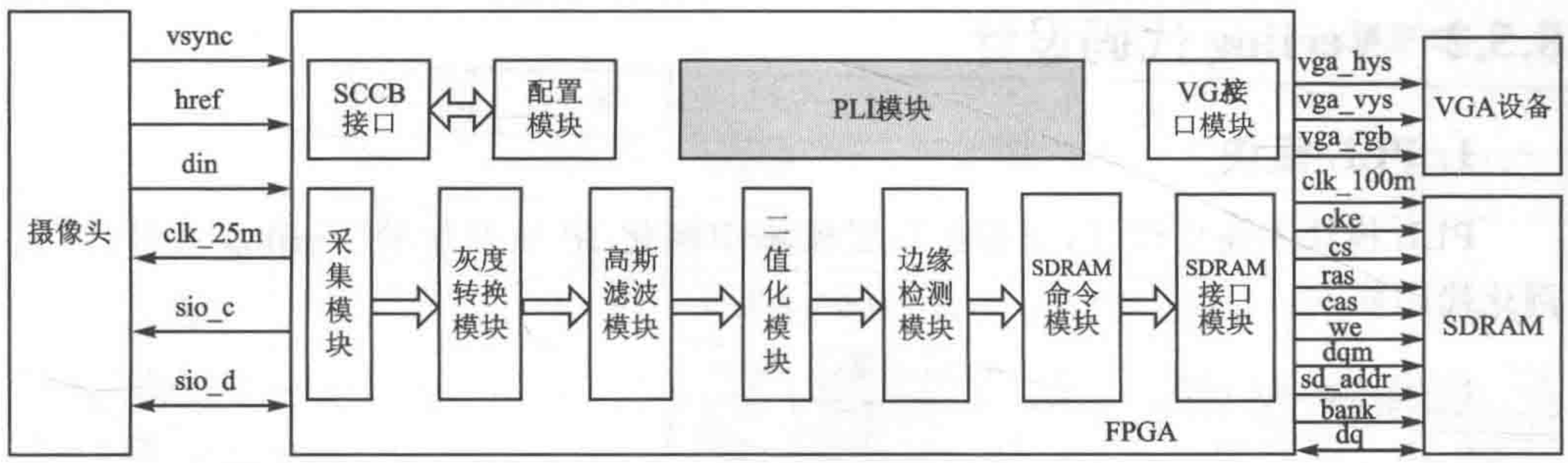


图 8－19　模块数据流向确认

8.5　FPGA 设计实现

8.5.1　硬　件

图 8－20 所示为明德扬至简荣耀版开发板以及摄像头模块、VGA 模块。

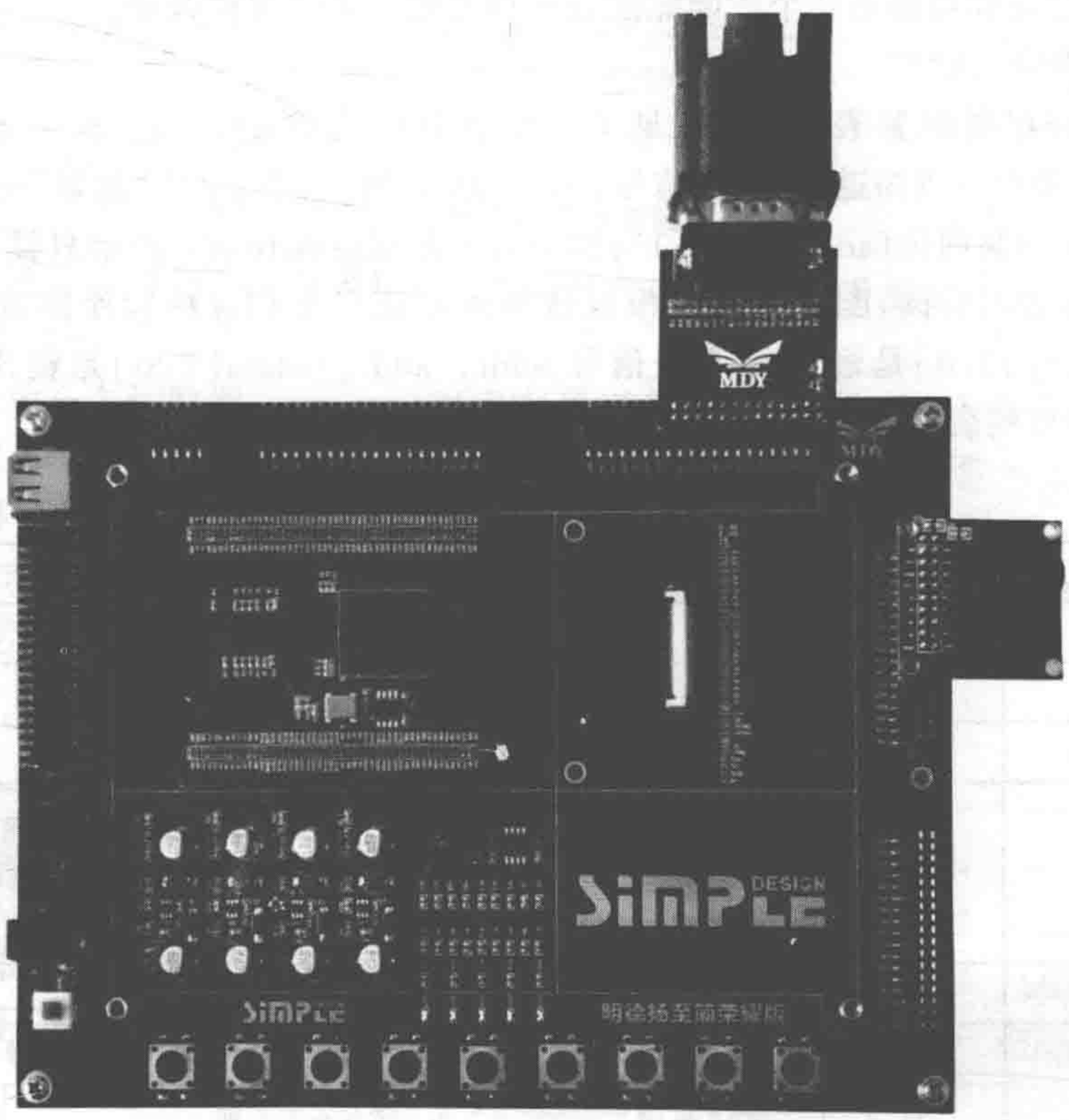


图 8－20　明德扬至简荣耀版开发板以及摄像头模块、VGA 模块

8.5.2 Verilog 代码设计

1. PLL 模块

PLL 模块不需要设计，只需在顶层模块中例化 IP 核产生的 Verilog 文件即可。例化代码如下：

```
1  pll_ipcore u0(
2      .inclk0  (clk      ),
3      .c0      (clk_25m ),
4      .c1      (clk_100m)
5  );
```

2. SCCB 接口与配置模块

SCCB 接口模块的设计请参考 2.2.4 小节进行学习。在此主要讲配置模块的设计。

摄像 OV7670 共有 164 个寄存器需要配置，我们自然会想到使用计数器去计算当前配置的寄存器是第几个。因此配置模块可用计数器架构设计。

(1) 明确功能

结合寄存器配置表的介绍(见 6.6.3 小节)，完成 OV7670 寄存器的配置。从 6.6.3小节中可以知道，配置表信号 add_wdata 为："操作码"+"地址"+"配置值"。add_wdata 的高两位(add_wdata[17:16])，00 表示空操作；01 表示只读；10 表示只写；11 表示先写后读，配置模块要根据这两位数值产生相应的读使能或者写使能。addr_wdata[15:8]是输出的地址信号 addr。addr_wdata[7:0]是输出的配置值 wdata。信号列表见表 8-1。

表 8-1 SCCB 接口模块信号列表

信号名	I/O	位 宽	说 明
clk	I	1	模块工作时钟 25 MHz
rst_n	I	1	复位信号
en_cmos	I	1	配置使能信号，来自上游模块，高电平有效
rdy	I	1	rdy 为忙闲指示信号。rdy 为 1 时表示摄像头处于空闲状态；rdy 为 0 时表示摄像头的某寄存器正在进行读操作或写操作，此时不能响应读使能或写使能
rdata	I	8	从摄像头某一寄存器读出的数据
rdata_vld	I	1	从摄像头某一寄存器读出的数据有效指示信号
wdata	O	8	写入摄像头某一寄存器的数据
addr	O	8	进行操作的摄像头寄存器地址

续表 8-1

信号名	I/O	位 宽	说 明
wr_en	O	1	写使能
rd_en	O	1	读使能
en_capture	O	1	摄像头采集使能信号，输出给采集模块。在配置过程中为低电平，配置完成后为高电平

(2) 功能波形

功能波形如图 8-21 所示。

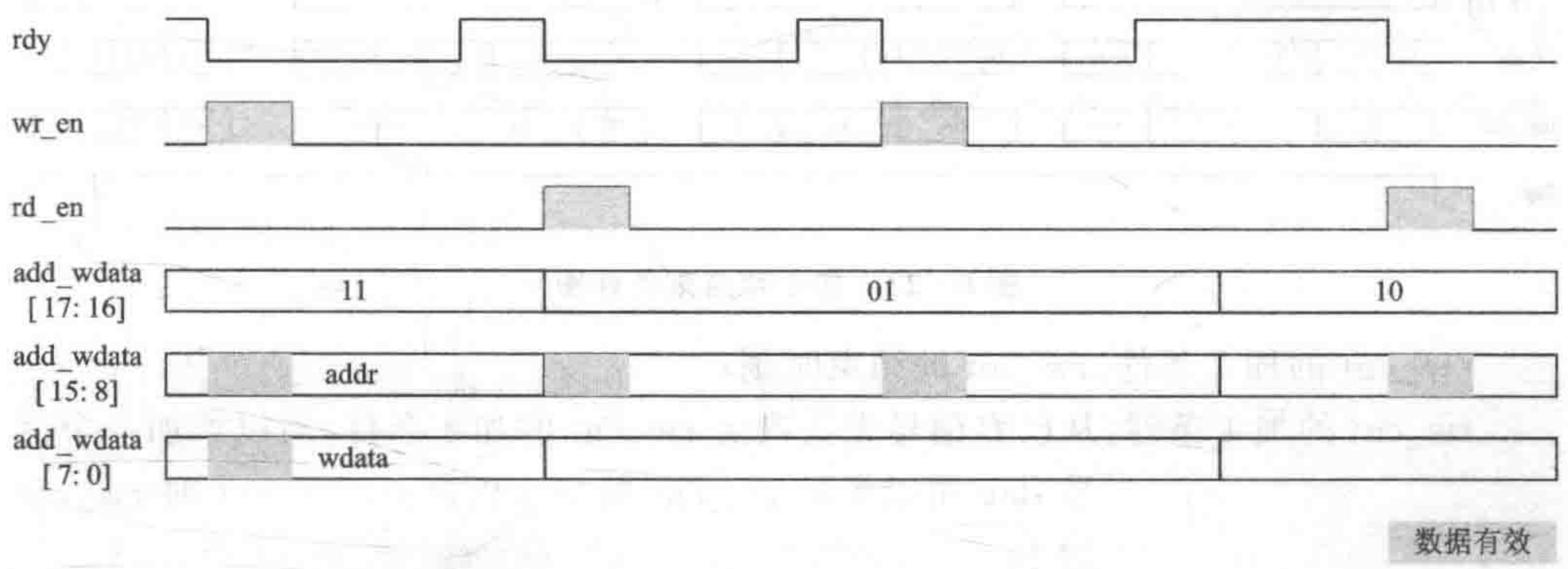


图 8-21 SCCB 接口模块功能波形图

注：本章所有图都省略了时钟 clk，所有信号均在 clk 的上升沿变化。

(3) 计数结构

首先，使用一个计数器(reg_cnt)对寄存器进行计数，依次配置 164 个寄存器。然后，再使用一个计数器 rw_cnt 判断当 rdy 有效时，需要产生读使能还是写使能。在配置过程中，当 rw_cnt 为 0 时，根据操作码第 17 位判断是否给出 wr_en 信号；当 rw_cnt 为 1 时，根据操作码第 18 位判断是否给出 rd_en 信号。其计数情况如图 8-22 所示。

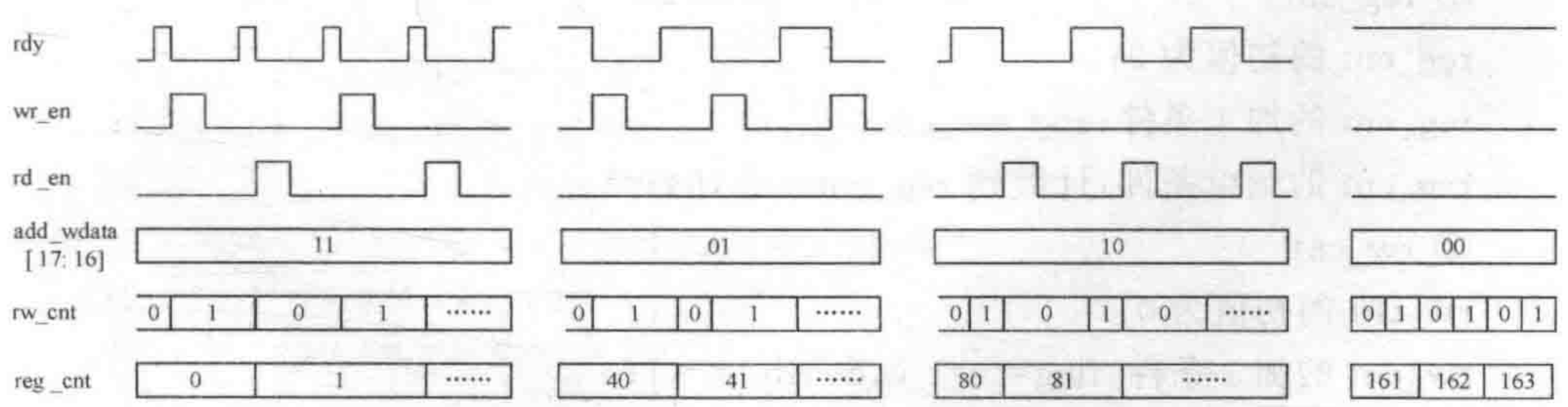


图 8-22 SCCB 接口模块计数结构图

图中，reg_cnt 的值与寄存器操作码 add_wdata[17:16]的对应关系是假设的，只

用于举例说明，实际的对应关系需要根据需求来设置。按此假设关系，配置结果是对第 0～39 个寄存器先写后读，对第 40～79 个寄存器只写，对第 80～160 个寄存器只读，对第 161～163 个寄存器不操作。

(4) 加 1 和结束条件

加 1 和结束条件如图 8－23 所示。

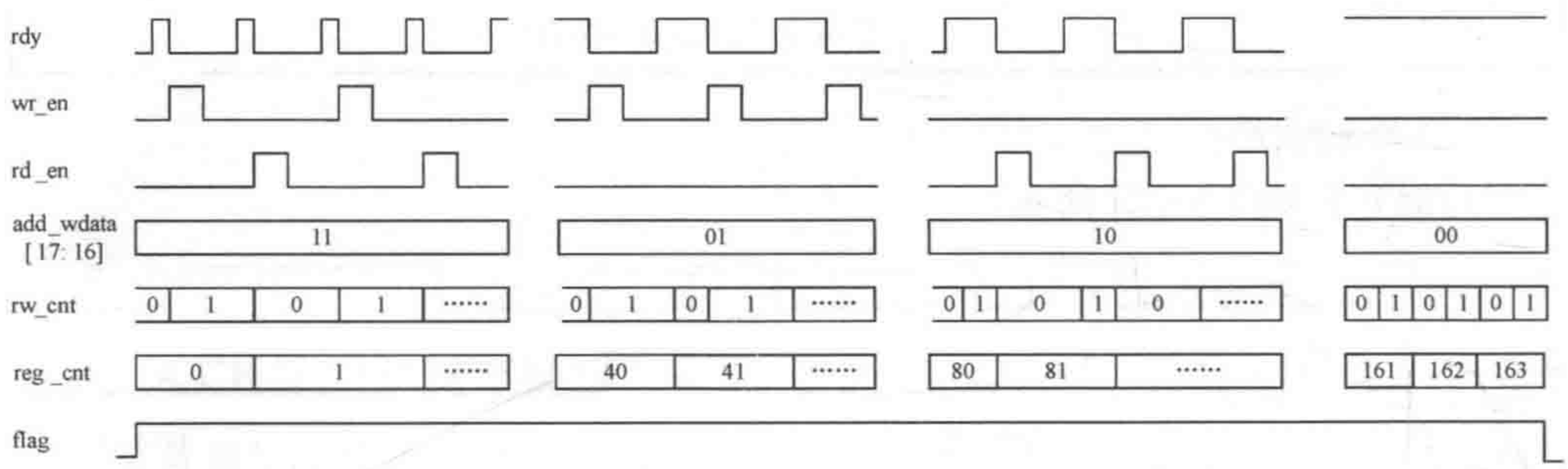


图 8－23　加 1 和结束条件图

reg_cnt 的加 1 条件：rw_cnt 的结束时刻；

rw_cnt 的加 1 条件：从已有信号无法确定 rw_cnt 的加 1 条件，所以添加一个信号 flag 用以指示 rw_cnt 的加 1 区间，flag＝＝1 时 rw_cnt 加 1；

reg_cnt 的结束条件：由于要配置 164 个寄存器，因此计数到 164－1；

rw_cnt 的结束条件：用于区分读、写，只需要两个计数值，因此计数到 2－1。

(5) 定义特殊点

reg_cnt 的结束条件：reg_cnt＝＝164－1，定为 end_reg_cnt；

wr_cnt 的结束条件：wr_cnt＝＝2－1，定为 end_wr_cnt；

判断是否产生写使能：rw_cnt＝＝0 && add_wdata[16]；

判断是否产生读使能：rw_cnt＝＝1 && add_wdata[17]。

(6) 完整性检查

① reg_cnt

reg_cnt 的初值为 0；

reg_cnt 的加 1 条件：end_rw_cnt；

reg_cnt 的结束条件：计数到 reg_cnt＝＝164－1。

② rw_cnt

rw_cnt 的初值为 0；

rw_cnt 的加 1 条件：flag＝＝1 && rdy＝＝1；

rw_cnt 的结束条件：计数到 rw_cnt＝＝2－1。

③ flag

flag 的初值为 0；

flag 变为 1 的条件：配置使能信号 en_cmos ==1；

flag 变为 0 的条件：配置结束时刻 end_reg_cnt。

④ en_capture

en_capture 的初值为 0；

en_capture 变为 1 的条件：配置结束时刻 end_reg_cnt；

en_capture 变为 0 的条件：en_cmos==1。

⑤ wr_en 与 rd_en

wr_en 与 rd_en 的初值为 0。

wr_en 与 rd_en 变为 1 的条件：在 flag==1 && rdy==1 时，若 rw_cnt==0 && add_wdata[16]==1，则 wr_en==1；若 rw_cnt==1 && add_wdata[17]==1，则 rd_en==1。

其他时刻 wr_en 与 rd_en 均为 0。

(7) 计数器代码

```
parameter  REG_NUM = 164;
parameter  RW_NUM  = 2;

always   @(posedge clk or negedge rst_n)begin
     if(rst_n == 1'b0)begin
          reg_cnt <= 0;
     end
     else if(add_reg_cnt)begin
          if(end_reg_cnt)
               reg_cnt <= 0;
          else
               reg_cnt <= reg_cnt + 1;
     end
end

assign add_reg_cnt = end_rw_cnt;
assign end_reg_cnt = add_reg_cnt && reg_cnt == REG_NUM - 1;

always   @(posedge clk or negedge rst_n)begin
     if(rst_n == 1'b0)begin
          rw_cnt <= 0;
     end
     else if(add_rw_cnt) begin
          if(end_rw_cnt)
               rw_cnt <= 0;
```

```
        else
            rw_cnt <= rw_cnt + 1;
    end
end

assign  add_rw_cnt = flag && rdy;
assign  end_rw_cnt = add_rw_cnt && rw_cnt == RW_NUM - 1;
```

(8)功能代码

```
`include "ov7670_para.v"

always   @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        flag <= 1'b0;
    end
    else if(en_cmos)begin
        flag <= 1'b1;
    end
    else if(end_reg_cnt)begin
        flag <= 1'b0;
    end
end

//cmos_en
always   @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        en_capture <= 1'b0;
    end
    else if(end_reg_cnt)begin
        en_capture <= 1'b1;
    end
    else if(en_cmos)begin
        en_capture <= 1'b0;
    end
end

//add_wdata
always   @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        wdata <= 8'b0;
    end
    else begin
```

```
34            wdata <= add_wdata[7:0];
35        end
36 end
37
38 always   @(posedge clk or negedge rst_n)begin
39        if(rst_n == 1'b0)begin
40            addr <= 8'b0;
41        end
42        else begin
43            addr <= add_wdata[15:8];
44        end
45 end
46
47 //wr_en
48 always   @(posedge clk or negedge rst_n)begin
49        if(rst_n == 1'b0)begin
50            wr_en <= 1'b0;
51        end
52        else if(add_rw_cnt && rw_cnt == 0 && add_wdata[16])begin
53            wr_en <= 1'b1;
54        end
55        else begin
56            wr_en <= 1'b0;
57        end
58 end
59
60 always   @(posedge clk or negedge rst_n)begin
61        if(rst_n == 1'b0)begin
62            rd_en <= 1'b0;
63        end
64        else if(add_rw_cnt && rw_cnt == 1 && add_wdata[17])begin
65            rd_en <= 1'b1;
66        end
67        else begin
68            rd_en <= 1'b0;
69        end
70 end
```

3. 采集模块

在 8.3.2 小节中已经详细介绍了 OV7670 采集的时序以及采集数据的格式。特别要注意，采集模块需要 2 个时钟周期才可以采集到 1 个像素点的数据。采集模块

需要做的是把从摄像头采集到的一帧 640×480 图像，用 dout_sop、dout_eop 和 dout_vld 等信号指示好，并输出到灰度转换等图像处理模块。

由于要计算像素点的个数来完成 1 帧 640×480 图像的采集，因此采用计数器架构来设计该模块。

(1) 明确功能

结合 8.3.2 小节摄像头采集原理，确定信号列表，见表 8-2。

表 8.2　采集模块信号列表

信号名	I/O	位　宽	说　明
clk	I	1	采集模块工作时钟 25 MHz
rst_n	I	1	模块复位信号
en_capture	I	1	摄像头采集使能信号，输出给采集模块。在配置过程中为低电平，配置完成后为高电平
vsync	I	1	帧同步信号
href	I	1	行同步信号
din	I	8	摄像头的图像数据
dout	O	16	采集到的图像数据
dout_vld	O	1	图像数据有效指示信号
dout_sop	O	1	一帧图像数据起始指示信号
dout_eop	O	1	一帧图像数据结束指示信号

(2) 功能波形

功能波形图如图 8-24 所示。

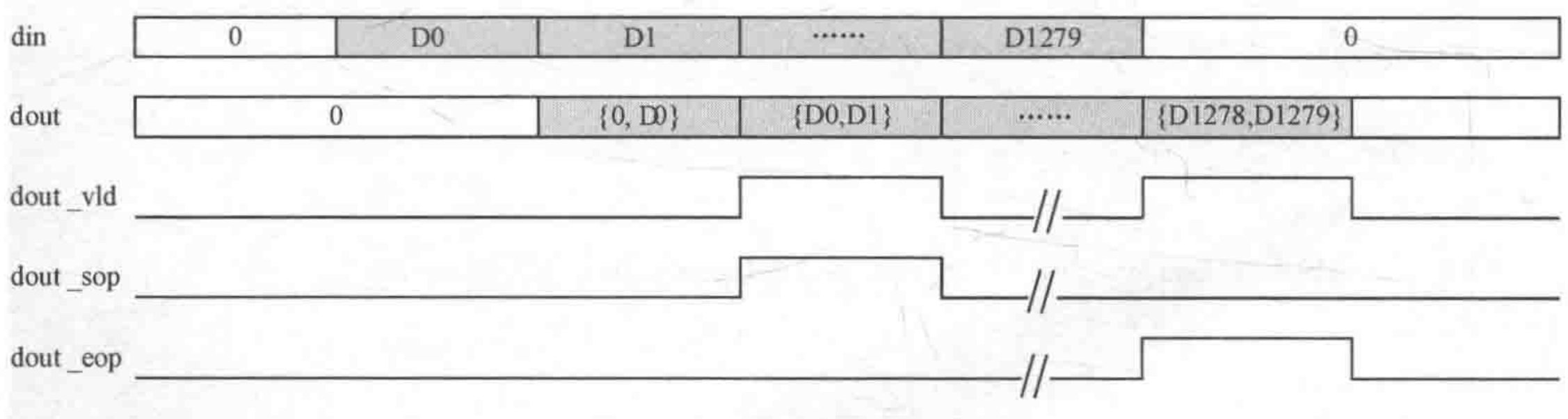


图 8-24　采集模块功能波形图

注意，图中只给出了一帧图像的第一行第 1 个像素点(dout_sop 高电平)和最后一行的第 640 个像素点(dout_eop 高电平)，中间行的所有像素点都省略了。

(3) 计数结构

由于要计数一行的 640 个像素点和一帧的 480 行，与 VGA 接口模块设计很相似，所以分别使用计数器 hys_cnt 和计数器 vys_cnt 来进行计数。计数情况如图 8-25 所示。

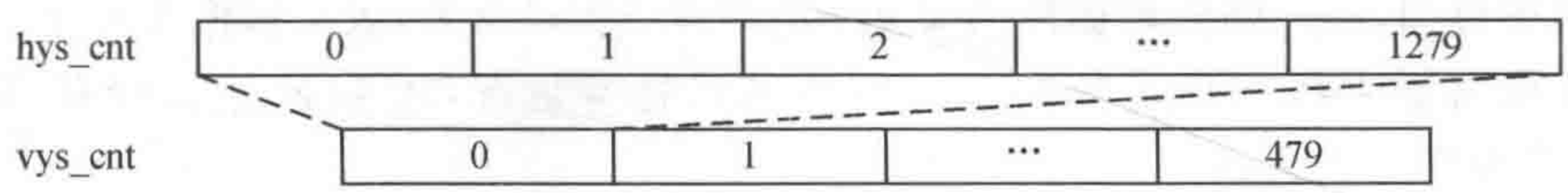


图 8－25　采集模块计数结构图

(4) 加 1 和结束条件

hys_cnt 的加 1 条件：由于 hys_cnt 在摄像头配置完后，检测到 vsync 的上升沿，就开始计数，因此引入一个指示信号 flag_capture，用于判断当前时刻是否满足 hys_cnt 的加 1 条件。当 flag_capture==1，hys_cnt 加 1。

vys_cnt 的加 1 条件：在 hys_cnt 计数结束时加 1，因此其加 1 条件为 end_hys_cnt。

hys_cnt 的结束条件：因需要两个时钟周期才能采完一个像素点，因此 hys_cnt==640×2－1 时计数结束。

vys_cnt 的结束条件：一帧共有 480 行，所以 vys_cnt==480－1 时计数结束。

(5) 定义特殊点

定义输入数据有效指示信号 din_vld，在 flag_capture==1（摄像头配置完成）&& href==1(摄像头输出数据 din 有效)时有效。

在数据有效期间，每两个时钟周期会采集一个像素点，因此定义一个指示信号 flag_dout_vld，在 hys_cnt[0] ==1 时有效，表示当前输出一个有效像素点。

vsync 上升沿指示信号 vsync_l2h，在 vsync_ff0 == 0（前一个时钟周期为 0）&& vsync == 1(当前时钟周期为 1)时为 1，表示检测到上升沿，开始一帧图像的传输。

(6) 完整性检查

① hys_cnt

hys_cnt 初值为 0；

hys_cnt 的加 1 条件：flag_capture==1 且 href==1；

hys_cnt 的结束条件：hys_cnt==640×2－1。

② vys_cnt

vys_cnt 初值为 0；

vys_cnt 的加 1 条件：end_hys_cnt；

vys_cnt 的结束条件：hys_cnt==480－1。

③ flag_capture

flag_capture 初值为 0；

flag_capture 从 0 变 1：vsync_l2h==1（检测到 vsync 上升沿）&& en_capture==1；

flag_capture 从 1 变 0：end_vys_cnt。

这里采用检测 vsync 上升沿的方法来判断是否开始采集数据。从 8.3.2 小节的

时序图中可知，每一帧的数据传输前 vsync 都有一段高电平，表示摄像头发来一帧数据。如果由于干扰出现了如图 8－26 所示的特殊情况，A 帧最后一行在 B 帧的 vsync 有效后才采集到。

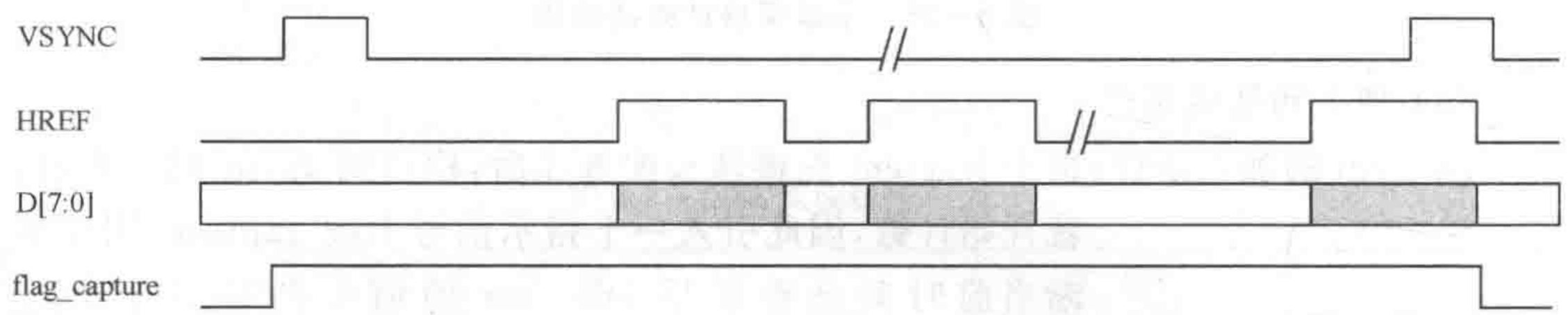


图 8－26　受干扰 HREF 出现时序偏差

而检测 vsync 的上升沿，会把 A 帧的图像完整地采集下来，而把 B 帧丢弃了，这种处理方式可以保证自动从第一个像素点开始采集，输出 dout 的每一帧都是从第一个像素点到最后一个像素点的，避免前一帧由于各种线路、器件干扰产生的采集错误（漏采一行、采集延迟等）累积到下一帧。

④ dout

在 din_vld==1，即 flag_capture==1 && href==1 时缓存 din 到低 8 位。

⑤ dout_vld

在 din 有效期间，每两个时钟周期有效一次，即当 hys_cnt[0]==1 时拉高。

⑥ dout_sop

在 vys_cnt==0（第一行）&& hys_cnt[10:1]==0（第一个像素点）时拉高。

⑦ dout_eop

在 vys_cnt==480－1（最后一行）&& hys_cnt[10:1]==640－1（最后一个像素点）时拉高。

（7）计数器代码

```
parameter      COL      = 640;
parameter      ROW      = 480;

always @ (posedge clk or negedge rst_n)begin
    if(!rst_n)begin
        hys_cnt <= 0;
    end
    else if(add_hys_cnt)begin
        if(end_hys_cnt)begin
            hys_cnt <= 0;
        end
        else begin
            hys_cnt <= hys_cnt + 1;
```

```
        end
    end
end

assign add_hys_cnt = flag_capture && din_vld;
assign end_hys_cnt = add_hys_cnt && hys_cnt == COL * 2 - 1;

always @ (posedge clk or negedge rst_n)begin
    if(!rst_n)begin
        vys_cnt <= 0;
    end
    else if(add_vys_cnt)begin
        if(end_vys_cnt)begin
            vys_cnt <= 0;
        end
        else begin
            vys_cnt <= vys_cnt + 1;
        end
    end
end

assign add_vys_cnt = end_hys_cnt;
assign end_vys_cnt = add_vys_cnt && vys_cnt == ROW - 1;
```

(8) 功能代码

```
always @ (posedge clk or negedge rst_n)begin
    if(!rst_n)begin
        flag_capture <= 0;
    end
    else if(flag_capture == 0 && vsync_l2h && en_capture)begin
        flag_capture <= 1;
    end
    else if(end_vys_cnt)begin
        flag_capture <= 0;
    end
end

assign vsync_l2h = vsync_ff0 == 0 && vsync == 1;

always @ (posedge clk or negedge rst_n)begin
    if(!rst_n)begin
        vsync_ff0 <= 0;
```

```
18      end
19      else begin
20          vsync_ff0 <= vsync;
21      end
22  end
23
24  always @ (posedge clk or negedge rst_n)begin
25      if(!rst_n)begin
26          dout <= 0;
27      end
28      else if(din_vld)begin
29          dout <= {dout[7:0],din};
30      end
31  end
32
33  assign din_vld = flag_capture && href;
34
35  always @ (posedge clk or negedge rst_n)begin
36      if(!rst_n)begin
37          dout_vld <= 0;
38      end
39      else if(flag_dout_vld)begin
40          dout_vld <= 1;
41      end
42      else begin
43          dout_vld <= 0;
44      end
45  end
46
47  assign flag_dout_vld = add_hys_cnt && hys_cnt[0] == 1;
48
49  always @ (posedge clk or negedge rst_n)begin
50      if(!rst_n)begin
51          dout_sop <= 0;
52      end
53      else if(flag_dout_vld && hys_cnt[10:1] == 0 && vys_cnt == 0)begin
54          dout_sop <= 1;
55      end
56      else begin
57          dout_sop <= 0;
58      end
59  end
```

```
60
61 always @ (posedge clk or negedge rst_n)begin
62     if(!rst_n)begin
63         dout_eop <= 0;
64     end
65     else if(flag_dout_vld && hys_cnt[10:1] == COL-1
66         && vys_cnt == ROW-1)begin
67         dout_eop <= 1;
68     end
69     else begin
70         dout_eop <= 0;
71     end
72 end
```

4. 灰度转换模块

根据8.3.3小节灰度转换的原理，本模块代码如下：

RGB565转为255个灰阶。

```
1  assign red    = {din[15:11],din[13:11]};
2  assign green  = {din[10:5],din[6:5]};
3  assign blue   = {din[4:0],din[2:0]};
4
5  always @ (posedge clk or negedge rst_n)begin
6      if(!rst_n)begin
7          dout <= 0;
8      end
9      else if(din_vld)begin
10         dout <= (red*76 + green*150 + blue*30) >> 8;
11     end
12 end
```

产生dout_vld，其跟随din_vld变化。

```
1  always @ (posedge clk or negedge rst_n)begin
2      if(!rst_n)begin
3          dout_vld <= 0;
4      end
5      else begin
6          dout_vld <= din_vld;
7      end
8  end
```

产生dout_sop，其跟随din_sop变化。

```
1  always @ (posedge clk or negedge rst_n)begin
2      if(!rst_n)begin
3          dout_sop <= 0;
4      end
5      else begin
6          dout_sop <= din_sop;
7      end
8  end
```

产生 dout_eop，其跟随 din_eop 变化。

```
1  always @ (posedge clk or negedge rst_n)begin
2      if(!rst_n)begin
3          dout_eop <= 0;
4      end
5      else begin
6          dout_eop <= din_eop;
7      end
8  end
```

5. 高斯滤波模块

从 8.3.4 小节高斯滤波原理可知，高斯滤波在计算上就是对 3×3 矩阵的各个元素求加权平均，在 FPGA 中表示矩阵可以用移位寄存器来实现。图 8-27 所示的移位寄存器是用 IP 核——RAM-Based Shift Register 生成的，由于图像的分辨率为 640×480，因此该移位寄存器有三个 taps，每个 taps 可存 640 个数据，每个数据的位宽为 8 bit，这些参数都可以在 Quartus II 中设置。

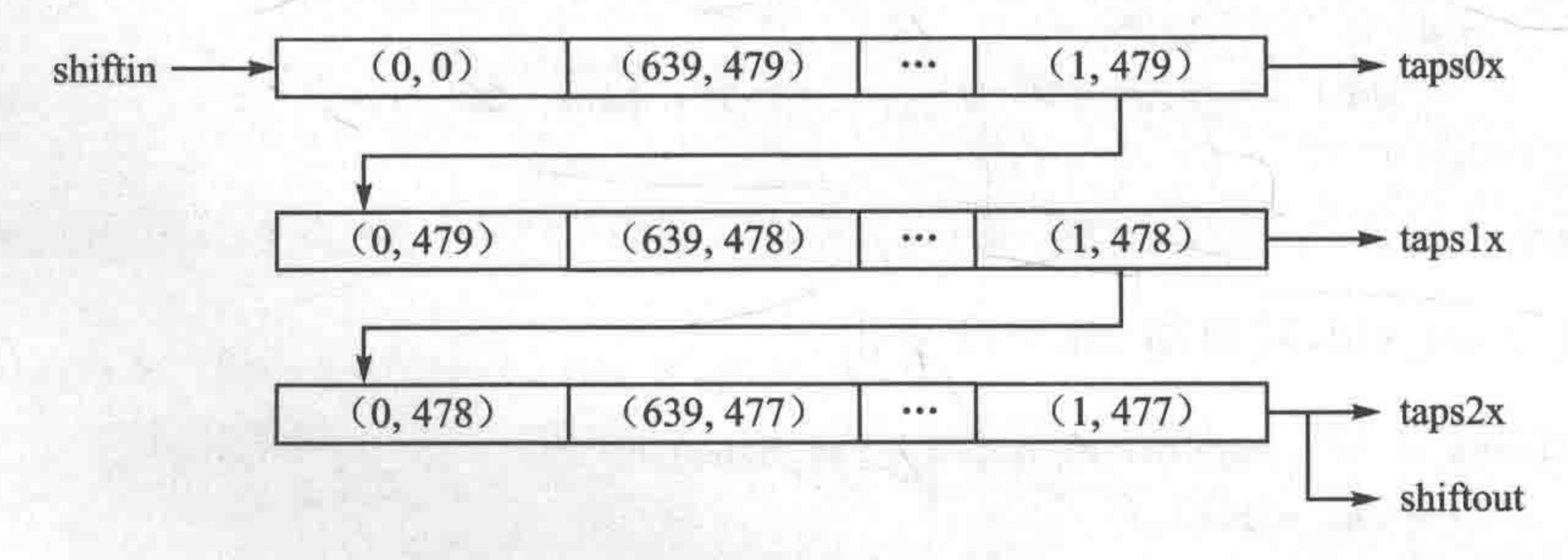


图 8-27 移位寄存器

数据以(0,0)到(639,479)的顺序送入移位寄存器，在每一行存满后会继续移位到下一行，taps0x～taps2x 输出每一行的最后一个数据，例如 taps0x，若连续来 3 个移位触发信号，那么 taps0x 的值依次为(1,479)，(2,479)，(3,479)。

我们对 taps0x～taps2x 打拍暂存，就可以得到 3×3 矩阵，如图 8-28 所示。

现在输入了下一帧图像的第一个像素点(0,0)，此时上面矩阵各元素的具体像素

点如图8-29所示。

taps0x	taps0x_ff0	taps0x_ff1
taps1x	taps1x_ff0	taps1x_ff1
taps2x	taps2x_ff0	taps2x_ff1

图8-28　3×3矩阵

(0, 479)	(639, 478)	(638, 478)
(0, 478)	(639, 477)	(638, 477)
(0, 477)	(639, 476)	(638, 476)

图8-29　矩阵对应像素点

那么高斯滤波的Verilog表达式为

gs_0＝taps0x_ff1＋2×taps1x_ff1＋taps2x_ff1

gs_1＝2×taps0x_ff0＋4×taps1x_ff0＋2×taps2x_ff0

gs_2＝taps0x＋2×taps1x＋taps2x

明白了移位寄存器的使用，很容易就可以写出高斯滤波的代码，要注意的是，为了方便处理，允许处理结果存在一定误差，在我们的实现方式中，每一帧640×480的图像的前三行与后三行都会有误差。

高斯滤波总得来说有以下三步：

第一步，建立3×3存储矩阵。

第二步，按公式计算每一列的代数和。

第三步，把每一列的代数和相加除以16，并输出。

相关代码如下：

```
shift_ipcore u1(
        .clken      (din_vld    ),
        .clock      (clk        ),
        .shiftin    (din        ),
        .taps0x     (taps0      ),
        .taps1x     (taps1      ),
        .taps2x     (taps2      )
);

always @ (posedge clk or negedge rst_n)begin
        if(!rst_n)begin
            din_vld_ff0 <= 0;
            din_vld_ff1 <= 0;
            din_vld_ff2 <= 0;
            din_sop_ff0 <= 0;
            din_sop_ff1 <= 0;
            din_sop_ff2 <= 0;
            din_eop_ff0 <= 0;
```

```
19          din_eop_ff1 <= 0;
20          din_eop_ff2 <= 0;
21      end
22      else begin
23          din_vld_ff0 <= din_vld;
24          din_vld_ff1 <= din_vld_ff0;
25          din_vld_ff2 <= din_vld_ff1;
26          din_sop_ff0 <= din_sop;
27          din_sop_ff1 <= din_sop_ff0;
28          din_sop_ff2 <= din_sop_ff1;
29          din_eop_ff0 <= din_eop;
30          din_eop_ff1 <= din_eop_ff0;
31          din_eop_ff2 <= din_eop_ff1;
32      end
33  end
34  /**********www.mdy-edu.com 明德扬科教 注释开始**********
35                          第一步,建立 3×3 存储矩阵
36   **********www.mdy-edu.com 明德扬科教 注释结束**********/
37  always @ (posedge clk or negedge rst_n)begin
38      if(!rst_n)begin
39          taps0_ff0 <= 0;
40          taps0_ff1 <= 0;
41          taps1_ff0 <= 0;
42          taps1_ff1 <= 0;
43          taps2_ff0 <= 0;
44          taps2_ff1 <= 0;
45      end
46      else if(din_vld_ff0)begin
47          taps0_ff0 <= taps0;
48          taps0_ff1 <= taps0_ff0;
49          taps1_ff0 <= taps1;
50          taps1_ff1 <= taps1_ff0;
51          taps2_ff0 <= taps2;
52          taps2_ff1 <= taps2_ff0;
53      end
54  end
55  /**********www.mdy-edu.com 明德扬科教 注释开始**********
56                          第二步,按公式计算每一列的代数和
57   **********www.mdy-edu.com 明德扬科教 注释结束**********/
58  always @ (posedge clk or negedge rst_n)begin
59      if(!rst_n)begin
60          gs_0 <= 0;
```

```
61        end
62        else if(din_vld_ff1)begin
63            gs_0 <= taps0_ff1 + 2 * taps1_ff1 + taps2_ff1;
64        end
65    end
66
67    always @ (posedge clk or negedge rst_n)begin
68        if(!rst_n)begin
69            gs_1 <= 0;
70        end
71        else if(din_vld_ff1)begin
72            gs_1 <= 2 * taps0_ff0 + 4 * taps1_ff0 + 2 * taps2_ff0;
73        end
74    end
75
76    always @ (posedge clk or negedge rst_n)begin
77        if(!rst_n)begin
78            gs_2 <= 0;
79        end
80        else if(din_vld_ff1)begin
81            gs_2 <= taps0 + 2 * taps1 + taps2;
82        end
83    end
84    /**********www.mdy-edu.com 明德扬科教 注释开始**********
85                    第三步,把每一列的代数和相加除以16,并输出
86     **********www.mdy-edu.com 明德扬科教 注释结束**********/
87    always @ (posedge clk or negedge rst_n)begin
88        if(!rst_n)begin
89            dout <= 0;
90        end
91        else if(din_vld_ff2)begin
92            dout <= (gs_0 + gs_1 + gs_2) >> 4;    //除以16
93        end
94    end
95
96    always @ (posedge clk or negedge rst_n)begin
97        if(!rst_n)begin
98            dout_vld <= 0;
99        end
100       else if(din_vld_ff2)begin
101           dout_vld <= 1;
102       end
```

```
103         else begin
104             dout_vld <= 0;
105         end
106 end
107
108 always @ (posedge clk or negedge rst_n)begin
109         if(!rst_n)begin
110             dout_sop <= 0;
111         end
112         else if(din_sop_ff2)begin
113             dout_sop <= 1;
114         end
115         else begin
116             dout_sop <= 0;
117         end
118 end
119
120 always @ (posedge clk or negedge rst_n)begin
121         if(!rst_n)begin
122             dout_eop <= 0;
123         end
124         else if(din_eop_ff2)begin
125             dout_eop <= 1;
126         end
127         else begin
128             dout_eop <= 0;
129         end
130 end
```

6. 二值化模块

二值化模块的设计比较简单，只需判断高斯滤波模块输出的数据(din)是否大于某一阈值，若是则输出 1(白色)；若不是则输出 0(黑色)。本项目使用的阈值是 127。代码如下：

```
1  always @ (posedge clk or negedge rst_n)begin
2          if(!rst_n)begin
3              dout <= 0;
4          end
5          else if(din >= 127)begin
6              dout <= 1;
7          end
8          else begin
```

```
9            dout <= 0;
10       end
11 end
12
13 always @ (posedge clk or negedge rst_n)begin
14       if(!rst_n)begin
15            dout_vld <= 0;
16       end
17       else begin
18            dout_vld <= din_vld;
19       end
20 end
21
22 always @ (posedge clk or negedge rst_n)begin
23       if(!rst_n)begin
24            dout_sop <= 0;
25       end
26       else begin
27            dout_sop <= din_sop;
28       end
29 end
30
31 always @ (posedge clk or negedge rst_n)begin
32       if(!rst_n)begin
33            dout_eop <= 0;
34       end
35       else begin
36            dout_eop <= din_eop;
37       end
38 end
```

7. 边缘检测模块

从 8.3.5 小节 Sobel 边缘检测的原理可知，Sobel 算法的原理与高斯滤波原理相似，都是用 3×3 矩阵对图像的像素点进行处理。送入边缘检测模块的数据是经过二值化处理的一个个像素点，我们的处理方式与高斯滤波相同，也是用移位寄存器的 IP 核，把这些数据以矩阵的方式表示出来。具体过程不再详细叙述。

边缘检测总得来说有以下四步：

第一步，建立 3×3 存储矩阵。

第二步，按公式计算一列或一行的代数和。

第三步，按公式求出矩阵的行或列差值，以绝对值的形式表示出来。

第四步，把行的绝对值与列的绝对值相加，判断结果是否为边缘点，并输出。按照这些步骤，一步步实现即可。代码如下：

```
1   shift2_ipcore u1(
2       .clken      (din_vld    ),
3       .clock      (clk        ),
4       .shiftin    (din        ),
5       .taps0x     (taps0      ),
6       .taps1x     (taps1      ),
7       .taps2x     (taps2      )
8   );
9
10  always @ (posedge clk or negedge rst_n)begin
11      if(!rst_n)begin
12          din_vld_ff0 <= 0;
13          din_vld_ff1 <= 0;
14          din_vld_ff2 <= 0;
15          din_vld_ff3 <= 0;
16          din_sop_ff0 <= 0;
17          din_sop_ff1 <= 0;
18          din_sop_ff2 <= 0;
19          din_sop_ff3 <= 0;
20          din_eop_ff0 <= 0;
21          din_eop_ff1 <= 0;
22          din_eop_ff2 <= 0;
23          din_eop_ff3 <= 0;
24      end
25      else begin
26          din_vld_ff0 <= din_vld;
27          din_vld_ff1 <= din_vld_ff0;
28          din_vld_ff2 <= din_vld_ff1;
29          din_vld_ff3 <= din_vld_ff2;
30          din_sop_ff0 <= din_sop;
31          din_sop_ff1 <= din_sop_ff0;
32          din_sop_ff2 <= din_sop_ff1;
33          din_sop_ff3 <= din_sop_ff2;
34          din_eop_ff0 <= din_eop;
35          din_eop_ff1 <= din_eop_ff0;
36          din_eop_ff2 <= din_eop_ff1;
37          din_eop_ff3 <= din_eop_ff2;
38      end
39  end
```

```
40  /**********www.mdy-edu.com 明德扬科教 注释开始**********
41                  第一步,建立3×3存储矩阵
42  **********www.mdy-edu.com 明德扬科教 注释结束**********/
43  always @ (posedge clk or negedge rst_n)begin
44      if(!rst_n)begin
45          taps0_ff0 <= 0;
46          taps0_ff1 <= 0;
47          taps1_ff0 <= 0;
48          taps1_ff1 <= 0;
49          taps2_ff0 <= 0;
50          taps2_ff1 <= 0;
51      end
52      else if(din_vld_ff0)begin
53          taps0_ff0 <= taps0;
54          taps0_ff1 <= taps0_ff0;
55          taps1_ff0 <= taps1;
56          taps1_ff1 <= taps1_ff0;
57          taps2_ff0 <= taps2;
58          taps2_ff1 <= taps2_ff0;
59      end
60  end
61  /**********www.mdy-edu.com 明德扬科教 注释开始**********
62                  第二步,按公式计算一列或一行的代数和
63  **********www.mdy-edu.com 明德扬科教 注释结束**********/
64  always @ (posedge clk or negedge rst_n)begin
65      if(!rst_n)begin
66          gx_0 <= 0;
67      end
68      else if(din_vld_ff1)begin
69          gx_0 <= taps0_ff1 + 2*taps1_ff1 + taps2_ff1;
70      end
71  end
72
73  always @ (posedge clk or negedge rst_n)begin
74      if(!rst_n)begin
75          gx_2 <= 0;
76      end
77      else if(din_vld_ff1)begin
78          gx_2 <= taps0 + 2*taps1 + taps2;
79      end
80  end
81
```

```
82  always @ (posedge clk or negedge rst_n)begin
83      if(!rst_n)begin
84          gy_0 <= 0;
85      end
86      else if(din_vld_ff1)begin
87          gy_0 <= taps0_ff1 + 2 * taps0_ff0 + taps0;
88      end
89  end
90
91  always @ (posedge clk or negedge rst_n)begin
92      if(!rst_n)begin
93          gy_2 <= 0;
94      end
95      else if(din_vld_ff1)begin
96          gy_2 <= taps2_ff1 + 2 * taps2_ff0 + taps2;
97      end
98  end
99  /**********www.mdy-edu.com 明德扬科教 注释开始**********
100         第三步,按公式求出矩阵的行或列差值,以绝对值的形式表示出来
101 **********www.mdy-edu.100 com 明德扬科教 注释结束**********/
102 always @ (posedge clk or negedge rst_n)begin
103     if(!rst_n)begin
104         gx <= 0;
105     end
106     else if(din_vld_ff2)begin
107         gx <= (gx_0>gx_2) ? (gx_0-gx_2) : (gx_2-gx_0);
108     end
109 end
110
111 always @ (posedge clk or negedge rst_n)begin
112     if(!rst_n)begin
113         gy <= 0;
114     end
115     else if(din_vld_ff2)begin
116         gy <= (gy_0>gy_2) ? (gy_0-gy_2) : (gy_2-gy_0);
117     end
118 end
119 /**********www.mdy-edu.com 明德扬科教 注释开始**********
120 第四步,把行的绝对值和列的绝对值相加,判断结果是否为边缘点,并输出
121 **********www.mdy-edu.com 明德扬科教 注释结束**********/
122 always @ (posedge clk or negedge rst_n)begin
123     if(!rst_n)begin
```

```
124             g <= 0;
125         end
126         else if(din_vld_ff3)begin
127             g <= gx + gy;
128         end
129 end
130
131 always @ ( * )begin
132         dout = (g> = 1) ? 1 : 0;
133 end
134
135 always @ (posedge clk or negedge rst_n)begin
136         if(! rst_n)begin
137             dout_vld <= 0;
138         end
139         else if(din_vld_ff3)begin
140             dout_vld <= 1;
141         end
142         else begin
143             dout_vld <= 0;
144         end
145 end
146
147 always @ (posedge clk or negedge rst_n)begin
148         if(! rst_n)begin
149             dout_sop <= 0;
150         end
151         else if(din_sop_ff3)begin
152             dout_sop <= 1;
153         end
154         else begin
155             dout_sop <= 0;
156         end
157 end
158
159 always @ (posedge clk or negedge rst_n)begin
160         if(! rst_n)begin
161             dout_eop <= 0;
162         end
163         else if(din_eop_ff3)begin
164             dout_eop <= 1;
165         end
```

```
166        else begin
167            dout_eop <= 0;
168        end
169 end
```

8. SDRAM 命令模块与接口模块

SDRAM 接口模块的设计请参考 3.2.3 小节。此处主要讲解命令模块的设计。结合 8.3.6 小节乒乓操作的原理，SDRAM 命令模块的基本框架如图 8－30 所示。

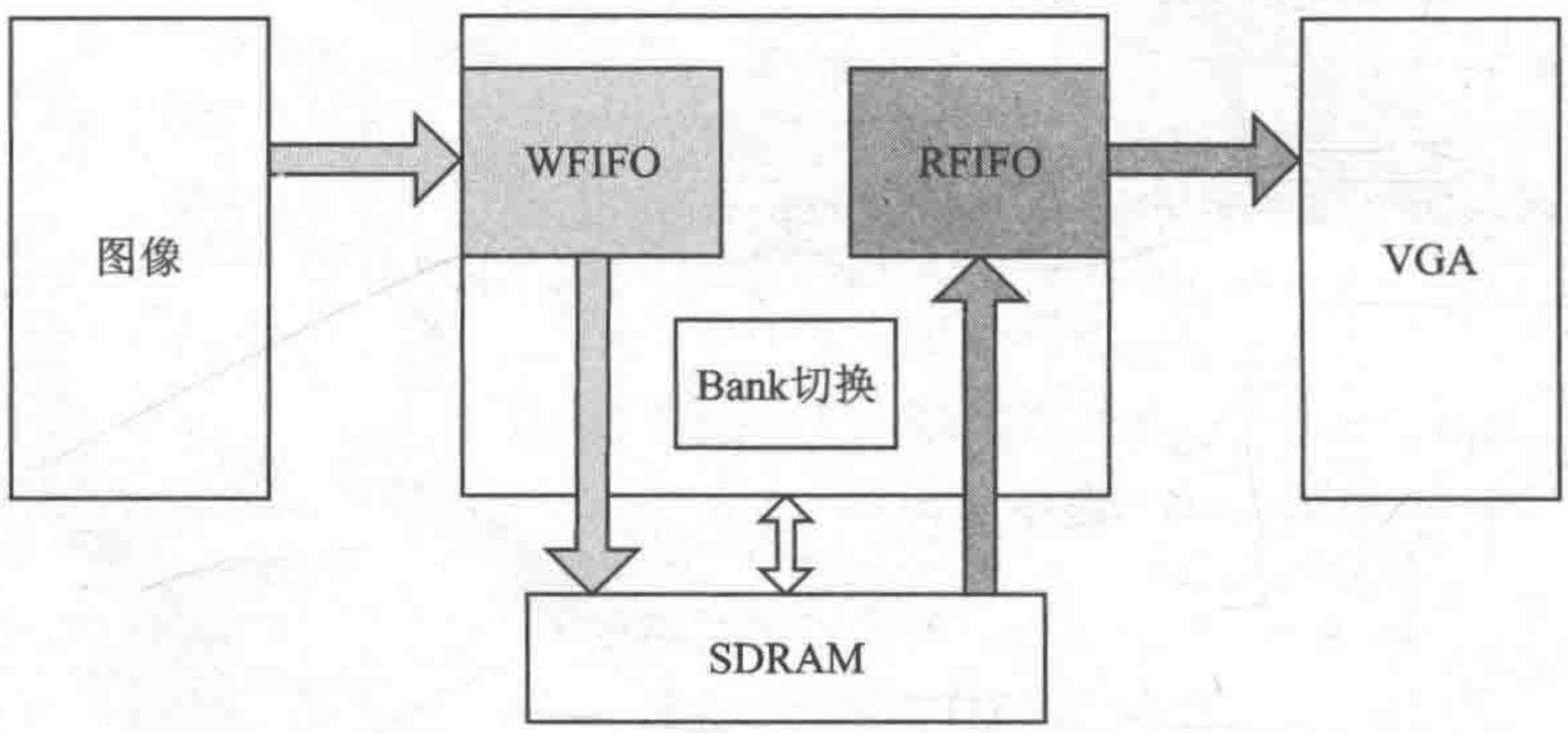


图 8－30　SDRAM 命令模块的模块划分

考虑到图像处理模块，VGA 接口模块是 25 MHz 时钟域，而 SDRAM 接口模块是 100 MHz 时钟域，因此引入了 WFIFO 与 RFIFO 解决数据跨时钟域传输的问题。如图 8－30 所示，模块可以分成三部分：写入图像部分 WFIFO、读取图像部分 RFIFO 与 Bank 切换部分。

写入图像部分是图像数据（以一帧 640×480 的格式采集和处理）的每个像素点按顺序写入到 WFIFO。当 WFIFO 中的数据大于或等于 256 个时，WFIFO 请求写 SDRAM；当 SDRAM 响应了写请求时，从 WFIFO 中读取数据写入 SDRAM。

读取图像部分是当 RFIFO 中的数据小于 300 个时，命令模块给 SDRAM 发送读请求，把从 SDRAM 中读出的数据写到 RFIFO 中，VGA 接口模块在行场信号到 Vaild 区域时，给 RFIFO 发送读请求，读取图像数据。

Bank 切换部分是用 SDRAM 内部的 Bank0 与 Bank3 实现乒乓操作。根据8.3.6 小节乒乓操作的原理，乒乓操作可以分为三个阶段：

阶段一，WFIFO 数据写入 Bank0。

阶段二，WFIFO 数据写入 Bank3，同时 RFIFO 从 Bank0 读取数据。

阶段三，WFIFO 数据写入 Bank0，同时 RFIFO 从 Bank3 读取数据。

按此顺序不断循环。

在这种处理方式下，阶段一是非常特殊的，因为 RFIFO 并没有读取数据，这样的特殊情况不方便设计。从 2.2.3 小节 VGA 接口可知，采用的 VGA 刷新速率是

60 Hz，即有 60 帧/s 图像，因此只在最开始的第一帧图像出现错误，对人眼来说是看不出差别的。为简化设计，跳过阶段一与阶段二，乒乓操作简化如图 8－31 所示。

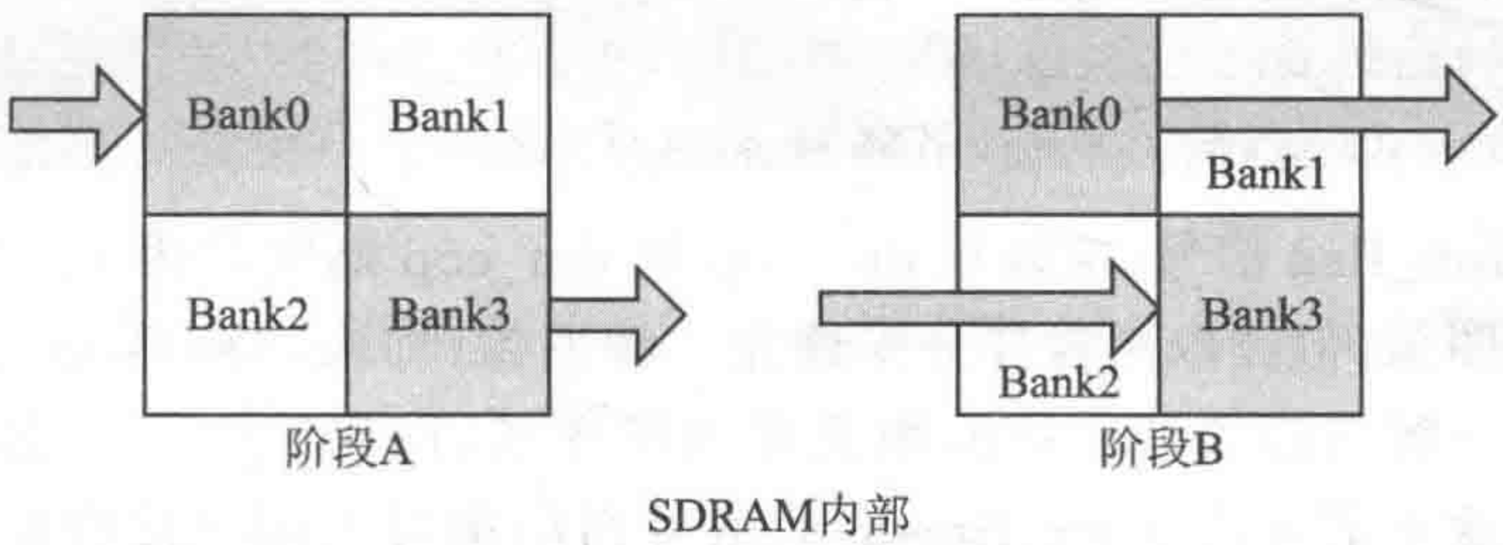


图 8－31　Bank 切换过程

阶段 A，WFIFO 数据写入 Bank0，RFIFO 从 Bank3 读出数据，等写完且读完一帧图像后，切换 Bank。

阶段 B，WFIFO 数据写入 Bank3，RFIFO 从 Bank0 读出数据，等写完且读完一帧图像后，再切换 Bank。

依此反复循环。

下面分 3 大部分来考虑模块的设计，分别是：写入图像部分、读取图像部分和 Bank 切换部分。

(1) 写入图像部分

WFIFO 写时钟：25 MHz，与采集模块、图像处理模块相同的时钟，即 clk_in。

WFIFO 读时钟：100 MHz，与 SDRAM 工作时钟相同，即 clk。

WFIFO 写数据：经过图像处理后的数据是 1 bit 的，而 SDRAM 的读/写位宽是 16 bit 的，因此 FIFO 位宽也设定为 16 bit，对 1 bit 数据进行位拼接即可。

WFIFO 写使能：考虑到并不是每一个图像输入的有效数据都写入 WFIFO 中。因此引入一个写状态指示信号 wr_fifo_flag，当 wr_fifo_flag==1 && din_vld==1 时，写使能有效。

基于以上分析，可写出下面的代码：

```
1  fifo_16b wfifo(
2      .aclr    (~rst_n        ),
3      .data    (wfifo_data    ),
4      .rdclk   (wfifo_rdclk   ),
5      .rdreq   (wfifo_rdreq   ),
6      .wrclk   (wfifo_wrclk   ),
7      .wrreq   (wfifo_wrreq   ),
8      .q       (wfifo_q       ),
9      .rdempty(wfifo_rdempty),
10     .rdusedw(wfifo_rdusedw)
11 );
```

```
12
13 assign  wfifo_data   = {16{din}};
14 assign  wfifo_rdclk = clk     ;
15 assign  wfifo_wrclk = clk_in ;
16 assign  wfifo_wrreq = din_vld && wr_wfifo_flag;
```

考虑 wr_fifo_flag 信号，应该从 din_sop 到 din_eop 期间都为 1。当图像写入部分接收完一帧图像，但读取图像部分未读完一帧图像（即未切换 Bank 之前），图像写入部分收到下一帧图像的 din_sop，则丢弃该帧图像，以保证已写入 Bank 的那帧图像不被覆盖。这里引入信号 wr_finish，此信号在图像写入部分接收完一帧图像（即收到 din_eop）拉高，直到读取图像部分读完一帧图像并且 Bank 切换完成后拉低。在 wr_finish 为高期间收到 din_sop（即收到新的一帧图像），wr_fifo_flag 不能拉高，即不接收此帧图像。代码如下：

```
1  assign  wr_wfifo_flag = (wr_finish == 1'b0 && din_ sop) || get_data_flag;
2
3  //wr_finish,接收完一帧图像拉高,Bank 切换完成后拉低
4  // change_bank_instr 上升沿就是表示 Bank 切换完成
5  always   @(posedge clk_in or negedge rst_n)begin
6       if(rst_n == 1'b0)begin
7           wr_finish <= 1'b0;
8       end
9       else if(get_data_flag&&din_eop) begin
10          wr_finish <= 1'b1;
11      end
12      else if(change_bank_instr_ff2 == 0 && change_bank_instr_ff1)begin
13          wr_finish <= 1'b0;
14      end
15 end
16
17 //开始写一帧的指示
18 always   @(posedge clk_in or negedge rst_n)begin
19      if(rst_n == 1'b0)begin
20          get_data_flag <= 1'b0;
21      end
22      else if(wr_finish == 1'b0 && din_sop) begin
23          get_data_flag <= 1'b1;
24      end
25      else if(din_eop) begin
26          get_data_flag <= 1'b0;
27      end
28 end
```

WFIFO读使能：首先使用的是全页读/写SDRAM方式，即一次连续向SDRAM中写入256个16 bit的图像数据。当图像写入部分WFIFO中的数据大于或等于256个(wfifo_rdusedw>=256)时，给SDRAM发送写请求wr_req，当收到来自SDRAM的写响应wr_ack时，从WFIFO中连续读出256个数据写入SDRAM。那么需要使用一个计数器cnt_rd_wfifo，从0计数到255，连续给WFIFO发送256个周期的读使能。

引入一个读状态指示信号rd_wfifo_flag来指示满足以上条件的时间段。代码如下：

```
1  //当WFIFO的数据大于或等于256时，就开始指示
2  assign rd_wfifo_start = rd_wfifo_flag == 1'b0 && wfifo_rdusedw> = 256;
3
4  always   @(posedge clk or negedge rst_n)begin
5      if(rst_n == 1'b0)begin
6          rd_wfifo_flag <= 1'b0;
7      end
8      else if(rd_wfifo_start) begin
9          rd_wfifo_flag <= 1'b1;
10     end
11     else if(end_cnt_rd_wfifo)begin
12         rd_wfifo_flag <= 1'b0;
13     end
14 end
15
16 //写请求wr_req
17 always   @(posedge clk or negedge rst_n)begin
18     if(rst_n == 1'b0)begin
19         wr_req <= 1'b0;
20     end
21     else if(rd_wfifo_start) begin
22         wr_req <= 1'b1;
23     end
24     else if(wr_ack)begin
25         wr_req <= 1'b0;
26     end
27 end
```

设计计数器cnt_rd_wfifo，考虑其加1条件和结束条件。

当WFIFO读状态指示信号rd_wfifo_flag有效并收到来自SDRAM的写应答wr_ack时，开始加1，计数到cnt_rd_wfifo==256-1结束，代码如下。SDRAM使用的是请求-应答交互架构，详细说明请参考本书的6.4节。

```
//从 WFIFO 中读数据送到 SDRAM 的计数,一个 BURST 长度
always   @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        cnt_rd_wfifo <= 0;
    end
    else if(add_cnt_rd_wfifo) begin
        if(end_cnt_rd_wfifo)
            cnt_rd_wfifo <= 0;
        else
            cnt_rd_wfifo <= cnt_rd_wfifo + 1;
    end
end
assign add_cnt_rd_wfifo = rd_wfifo_flag    && ((cnt_rd_wfifo == 0&&wr_ack)
                          ||(cnt_rd_wfifo! = 0));
assign end_cnt_rd_wfifo = add_cnt_rd_wfifo && cnt_rd_wfifo == 256 - 1;
```

有了上述分析后,WFIFO 读请求的设计就非常明确了,在读状态有效时,收到写应答且 WFIFO 不空,则读使能有效。代码如下:

```
always  @( * )begin
    if(add_cnt_rd_wfifo && wfifo_rdempty == 1'b0)
        wfifo_rdreq = 1'b1;
    else
        wfifo_rdreq = 1'b0;
end
```

WFIFO 图像数据写入 SDRAM 的地址:全页写一次写入 256 个像素点,因此完成一帧图像的写入共需要

$$\frac{640\times480}{256}=1\ 200\ 次$$

引入计数器 cnt_waddr,在 cnt_rd_wfifo 计数结束时加 1,计数到 cnt_waddr==1 200-1 结束。而计数器 cnt_waddr 的计数值可以作为 SDRAM 的行地址。代码如下:

```
always   @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        cnt_waddr <= 0;
    end
    else if(add_cnt_waddr) begin
        if(end_cnt_waddr)
            cnt_waddr <= 0;
        else
            cnt_waddr <= cnt_waddr + 1;
    end
```

```
end

assign add_cnt_waddr =   end_cnt_rd_wfifo;
assign end_cnt_waddr =   add_cnt_waddr && cnt_waddr == 1200 - 1;

//SDRAM 的写地址,1'b0,是为了与接口模块对齐:2 bit Bank, 12 bit ROW, 8 bit COLUMN;
assign waddr = {waddr_bank,{1'b0,cnt_waddr},8'b0};
assign wdata = wfifo_q                          ;
```

(2) 读取图像部分

RFIFO 写时钟:100 MHz,与 SDRAM 工作时钟相同,即 clk。

RFIFO 读时钟:25 MHz,与 VGA 接口模块相同的时钟,即 clk_in。

RFIFO 写数据:SDRAM 中读出的数据 rdata。

RFIFO 写使能:SDRAM 给出的读数据有效信号 rdata_vld。

RFIFO 读使能:VGA 接口模块在行场信号到 Vaild 区域时,会给 RFIFO 发送读请求 dout_rd,此时 RFIFO 读使能有效。

基于以上分析,可写出下面的代码:

```
fifo_16b rfifo(
      .aclr    (~rst_n        ),
      .data    (rfifo_data    ),
      .rdclk   (rfifo_rdclk   ),
      .rdreq   (rfifo_rdreq   ),
      .wrclk   (rfifo_wrclk   ),
      .wrreq   (rfifo_wrreq   ),
      .q       (rfifo_q       ),
      .rdempty(rfifo_rdempty),
      .rdusedw(rfifo_rdusedw)
);

assign  rfifo_data   = rdata     ;
assign  rfifo_rdclk  = clk_in    ;
assign  rfifo_rdreq  = dout_rd   ;
assign  rfifo_wrclk  = clk       ;
assign  rfifo_wrreq  = rdata_vld ;
assign  dout         = rfifo_q   ;//RFIFO 输出,送给 VGA 接口模块
```

那么命令模块什么时候向 SDRAM 发出读请求呢?当 RFIFO 中的数据少于 300 个时,向 SDRAM 发出读请求,这样可以保证 RFIFO 中至少有两个 BURST 数据,不会出现读空情况。代码如下:

```
//当 RFIFO 中的数据少于 300 时,开始读 SDRAM 的数据
```

```
assign wr_rfifo_start = wr_rfifo_flag == 1'b0 && rfifo_rdusedw<300;

always   @(posedge clk or negedge rst_n)begin
     if(rst_n == 1'b0)begin
         wr_rfifo_flag <= 1'b0;
     end
     else if(wr_rfifo_start) begin
         wr_rfifo_flag <= 1'b1;
     end
     else if(end_cnt_wr_rfifo)begin
         wr_rfifo_flag <= 1'b0;
     end
end

//读请求 rd_req
always   @(posedge clk or negedge rst_n)begin
     if(rst_n == 1'b0)begin
         rd_req <= 1'b0;
     end
     else if(wr_rfifo_start) begin
         rd_req <= 1'b1;
     end
     else if(rd_ack)begin
         rd_req <= 1'b0;
     end
end
```

由于全页读，故需要引入计数器 cnt_wr_rfifo，收到 rdata_vld 时加 1，计数到 cnt_wr_rfifo==256－1 结束。代码如下：

```
//对读到的数据进行计数，统计一个 BURST 长度
always   @(posedge clk or negedge rst_n)begin
     if(rst_n == 1'b0)begin
         cnt_wr_rfifo <= 0;
     end
     else if(add_cnt_wr_rfifo)begin
         if(end_cnt_wr_rfifo)
             cnt_wr_rfifo <= 0;
         else
             cnt_wr_rfifo <= cnt_wr_rfifo + 1;
     end
end
```

```
assign  add_cnt_wr_rfifo = rdata_vld ;
assign  end_cnt_wr_rfifo = add_cnt_wr_rfifo && cnt_wr_rfifo == 256 - 1;
```

与 WFIFO 相同，引入计数器 add_cnt_raddr 用于计算 SDRAM 的读行地址。代码如下：

```
//读 SDRAM 的 BURST 个数,1200 个刚好完 1 帧
always   @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        cnt_raddr <= 0;
    end
    else if(add_cnt_raddr) begin
        if(end_cnt_raddr)
            cnt_raddr <= 0;
        else
            cnt_raddr <= cnt_raddr + 1;
    end
end
assign add_cnt_raddr = end_cnt_wr_rfifo;
assign end_cnt_raddr = add_cnt_raddr && cnt_raddr == 1200 - 1;

//1'b0,是为了与接口模块对齐:2 bit Bank, 12bit ROW, 8 bit COLUMN;
assign raddr = {raddr_bank,{1'b0,cnt_raddr},8'b0};
```

(3) Bank 切换部分

SDRAM 的写结束时，对 waddr_bank 进行切换，Bank0（地址 2'b00）与 Bank3（地址 2'b11）的相互切换只需要取反即可。代码如下：

```
always   @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        waddr_bank <= 2'b0;
    end
    else if(end_cnt_waddr) begin
        waddr_bank <= ~waddr_bank ;
    end
end
```

考虑 raddr_bank 的切换，首先要了解图像写入速率与 VGA 显示速率的关系，我们知道两者的速率都是 25 MHz，但是 VGA 的时序中有同步脉冲、显示前沿、显示后沿，占据了一部分时间。因此对于 SDRAM 命令模块来说，图像写入一定比图像读出快，即一帧图像写入了 SDRAM 的 Bank0，而在 Bank3 中图像的一帧数据还没读完。因此 raddr_bank 的切换代码如下：

```
assign  change_bank = end_cnt_raddr && raddr_bank == waddr_bank;

always  @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        raddr_bank <= 2'b11;
    end
    else if(change_bank) begin
        raddr_bank <= ~raddr_bank;
    end
end
```

我们在前面 wr_finish 信号的设计中用到了 Bank 切换作为判断条件，但是，细心的读者可能已经发现代码中使用的是 change_bank_instr，而没有直接使用 change_bank。这是因为 Bank 切换指示信号 change_bank 是在 100 MHz 时钟域(clk)产生的，而 wr_finish 信号是在 25 MHz 时钟域(clk_in)产生的。因此需要对 Bank 切换信号进行一定处理，保证 wr_finish 可以采集到 Bank 切换指示信号 change_bank。

对 change_bank 信号进行延时，并引入 change_bank_instr 信号，当 change_bank 有效时就拉高。那么应该拉高多长时间呢？周期 10 ns 产生的数据要被 40 ns 的时钟采集到，即至少要拉高 40 ns，为了保险起见，拉高 80 ns，有两个 25 MHz 时钟周期那么长的时间，就可以保证在 25 MHz 时钟的上升沿采集到 change_bank_instr 为 1。这里需要引入一个计数器 cnt_change_bank，从 change_bank_instr 有效开始加 1，计数到 cnt_change_bank==8-1 结束。代码如下：

```
always  @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        change_bank_instr <= 0;
    end
    else if(change_bank) begin
        change_bank_instr <= 1'b1;
    end
    else if(end_cnt_change_bank)begin
        change_bank_instr <= 1'b0;
    end
end

always  @(posedge clk or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        cnt_change_bank <= 0;
    end
    else if(add_cnt_change_bank) begin
        if(end_cnt_change_bank)
```

```
            cnt_change_bank <= 0;
        else
            cnt_change_bank <= cnt_change_bank + 1;
    end
end

assign add_cnt_change_bank = change_bank_instr;
assign end_cnt_change_bank = add_cnt_change_bank && cnt_change_bank == 8 - 1;
```

100 MHz 时钟域到 25 MHz 时钟域，还要进行打两拍的异步处理，然后再打一拍用于上升沿检测。代码如下：

```
//异步处理
always  @(posedge clk_in or negedge rst_n)begin
    if(rst_n == 1'b0)begin
        change_bank_instr_ff0 <= 1'b0;
        change_bank_instr_ff1 <= 1'b0;
        change_bank_instr_ff2 <= 1'b0;
    end
    else begin
        change_bank_instr_ff0 <= change_bank_instr     ;
        change_bank_instr_ff1 <= change_bank_instr_ff0;
        change_bank_instr_ff2 <= change_bank_instr_ff1;
    end
end
```

完成写入图像部分、读取图像部分和 Bank 切换部分三部分的设计后，SDRAM 命令模块的设计就完成了。

9. VGA 接口模块

VGA 接口模块的设计在 2.2.3 小节有详细说明，此处不再赘述。

10. 顶层模块

例化 SCCB 接口模块、摄像头采集模块、摄像头配置模块、灰度转换模块、高斯滤波模块、二值化模块、边缘检测模块、SDRAM 命令模块、SDRAM 接口模块、VGA 接口模块以及 PLL 模块，形成顶层模块。注意 SCCB 接口的 sio_d 信号是 inout 类型，要在顶层经过一个三态门电路进行处理，以达到单总线读/写的目的。为了简化代码，此处略去了 reg 型和 wire 型的定义，代码如下：

```
module sobel_edge_top(
    clk          ,
    rst_n        ,

    vsync        ,
```

```
6        href        ,
7        din         ,
8        clk_25m     ,
9        sclk        ,
10       sio_d       ,
11
12       clk_100m    ,
13       cke         ,
14       cs          ,
15       ras         ,
16       cas         ,
17       we          ,
18       dpm         ,
19       sd_addr     ,
20       bank        ,
21       dq          ,
22
23       vga_hys     ,
24       vga_vys     ,
25       vga_rgb
26   );
27
28       input       clk     ;
29       input       rst_n   ;
30       input       vsync   ;
31       input       href    ;
32       input  [7:0]  din   ;
33
34       output      clk_25m ;
35       output      clk_100m ;
36       output      cke     ;
37       output      cs      ;
38       output      ras     ;
39       output      cas     ;
40       output      we      ;
41       output      dpm     ;
42       output [11:0]  sd_addr ;
43       output [1:0]   bank   ;
44       output      vga_hys ;
45       output      vga_vys ;
46       output [7:0]   vga_rgb;
47
```

```
48      output          sclk      ;
49      inout           sio_d     ;
50      inout           dq        ;
51
52      //省略了 reg 和 wire 定义
53
54      assign sio_d = sda_out_en ? sda_out : 1'dz;
55      assign sda_in = sio_d;
56
57      pll_ipcore u0(
58          .inclk0 (clk      ),
59          .c0     (clk_25m ),
60          .c1     (clk_100m)
61      );
62
63      ov7670_config u_config(
64          .clk          (clk_25m     ),
65          .rst_n        (rst_n       ),
66          .config_en    (config_en   ), //为 1 时,表示摄像头开始配置
67          .rd_en        (rd_en       ),
68          .wr_en        (wr_en       ),
69          .rdy          (rdy         ),
70          .rdata        (rdata       ),
71          .rdata_vld    (rdata_vld   ),
72          .wdata        (wdata       ),
73          .addr         (addr        ),
74          .cmos_en      (en_capture ) //为 1 时,表示摄像头配置完成
75      );
76
77      sccb u_sccb(
78          .clk          (clk_25m     ),
79          .rst_n        (rst_n       ),
80          .addr         (addr        ),
81          .wr_en        (wr_en       ),
82          .wdata        (wdata       ),
83          .rd_en        (rd_en       ),
84          .rdata        (rdata       ),
85          .rdata_vld    (rdata_vld   ),
86          .rdy          (rdy         ),
87          .sclk         (sio_c       ),
88          .sda_in       (sda_in      ),
89          .sda_out      (sda_out     ),
```

```
90          .sda_out_en   (sda_out_en )
91      );
92
93      cmos_capture u_cmos_capture(
94          .clk          (clk_25m          ),
95          .rst_n        (rst_n            ),
96          .en_capture   (en_capture       ),
97          .vsync        (vsync            ),
98          .href         (href             ),
99          .din          (din              ),
100         .dout         (cmos_dout        ),
101         .dout_vld     (cmos_dout_vld    ),
102         .dout_sop     (cmos_dout_sop    ),
103         .dout_eop     (cmos_dout_eop    )
104     );
105
106     rgb565_gray u_rgb565_gray(
107         .clk          (clk_25m          ),
108         .rst_n        (rst_n            ),
109         .din          (cmos_dout        ),
110         .din_vld      (cmos_dout_vld    ),
111         .din_sop      (cmos_dout_sop    ),
112         .din_eop      (cmos_dout_eop    ),
113         .dout         (gray_dout        ),
114         .dout_vld     (gray_dout_vld    ),
115         .dout_sop     (gray_dout_sop    ),
116         .dout_eop     (gray_dout_eop    )
117     );
118
119     gs_filter u_gs_filter(
120         .clk          (clk_25m          ),
121         .rst_n        (rst_n            ),
122         .din          (gray_dout        ),
123         .din_vld      (gray_dout_vld    ),
124         .din_sop      (gray_dout_sop    ),
125         .din_eop      (gray_dout_eop    ),
126         .dout         (gs_dout          ),
127         .dout_vld     (gs_dout_vld      ),
128         .dout_sop     (gs_dout_sop      ),
129         .dout_eop     (gs_dout_eop      )
130     );
131
```

```
132     gray_bit u_gray_bit(
133         .clk            (clk_25m            ),
134         .rst_n          (rst_n              ),
135         .value          (value_gray         ),
136         .din            (gs_dout            ),
137         .din_vld        (gs_dout_vld        ),
138         .din_sop        (gs_dout_sop        ),
139         .din_eop        (gs_dout_eop        ),
140         .dout           (bit_dout           ),
141         .dout_vld       (bit_dout_vld       ),
142         .dout_sop       (bit_dout_sop       ),
143         .dout_eop       (bit_dout_eop       )
144     );
145
146     sobel u_sobel(
147         .clk            (clk_25m            ),
148         .rst_n          (rst_n              ),
149         .din            (bit_dout           ),
150         .din_vld        (bit_dout_vld       ),
151         .din_sop        (bit_dout_sop       ),
152         .din_eop        (bit_dout_eop       ),
153         .dout           (sobel_dout         ),
154         .dout_vld       (sobel_dout_vld     ),
155         .dout_sop       (sobel_dout_sop     ),
156         .dout_eop       (sobel_dout_eop     )
157     );
158
159     sdram_cmd u_sdram_cmd(
160         .rst_n          (rst_n              ),
161         .clk_in         (clk_25m            ),
162         .din            (sobel_dout         ),
163         .din_vld        (sobel_dout_vld     ),
164         .din_sop        (sobel_dout_sop     ),
165         .din_eop        (sobel_dout_eop     ),
166         .clk            (clk_100m           ),
167         .wr_req         (wr_req             ),
168         .waddr          (waddr              ),
169         .wdata          (wdata              ),
170         .wr_ack         (wr_ack             ),
171         .rd_req         (rd_req             ),
172         .raddr          (raddr              ),
173         .rd_ack         (rd_ack             ),
```

```
174         .rdata          (rdata          ),
175         .rdata_vld      (rdata_vld      ),
176         .dout           (vga_din        ),
177         .dout_rd        (dout_rd        )
178     );
179
180     sdram_inf u_sdram_inf(
181         .clk            (clk_100m       ),
182         .rst_n          (rst_n          ),
183         .wr_req         (wr_req         ),
184         .waddr          (waddr          ),
185         .wdata          (wdata          ),
186         .wr_ack         (wr_ack         ),
187         .raddr          (raddr          ),
188         .rd_req         (rd_req         ),
189         .rw_ack         (rd_ack         ),
190         .cke            (cke            ),
191         .cs             (cs             ),
192         .ras            (ras            ),
193         .cas            (cas            ),
194         .we             (we             ),
195         .dqm            (dqm            ),
196         .addr           (sd_addr        ),
197         .bank           (bank           ),
198         .dq             (dq             ),
199         .rdata          (rdata          ),
200         .rd_vld         (rdata_vld      )
201     );
202
203     vga_driver u_vga_driver(
204         .clk            (clk_25m        ),
205         .rst_n          (rst_n          ),
206         .din            (vga_din        ),
207         .hys            (vga_hys        ),
208         .vys            (vga_vys        ),
209         .rgb_data       (rgb_data       ),
210         .dout_rd        (dout_rd        )
211     );
212
213 endmodule
```

8.6　功能仿真

1. SCCB 接口与配置模块

对前四个寄存器进行操作，操作码分别为 2'b00，2'b01，2'b10，2'b11，对其他的寄存器不进行操作，即操作码为 2'b00，仿真结果如图 8－32 和图 8－33 所示。

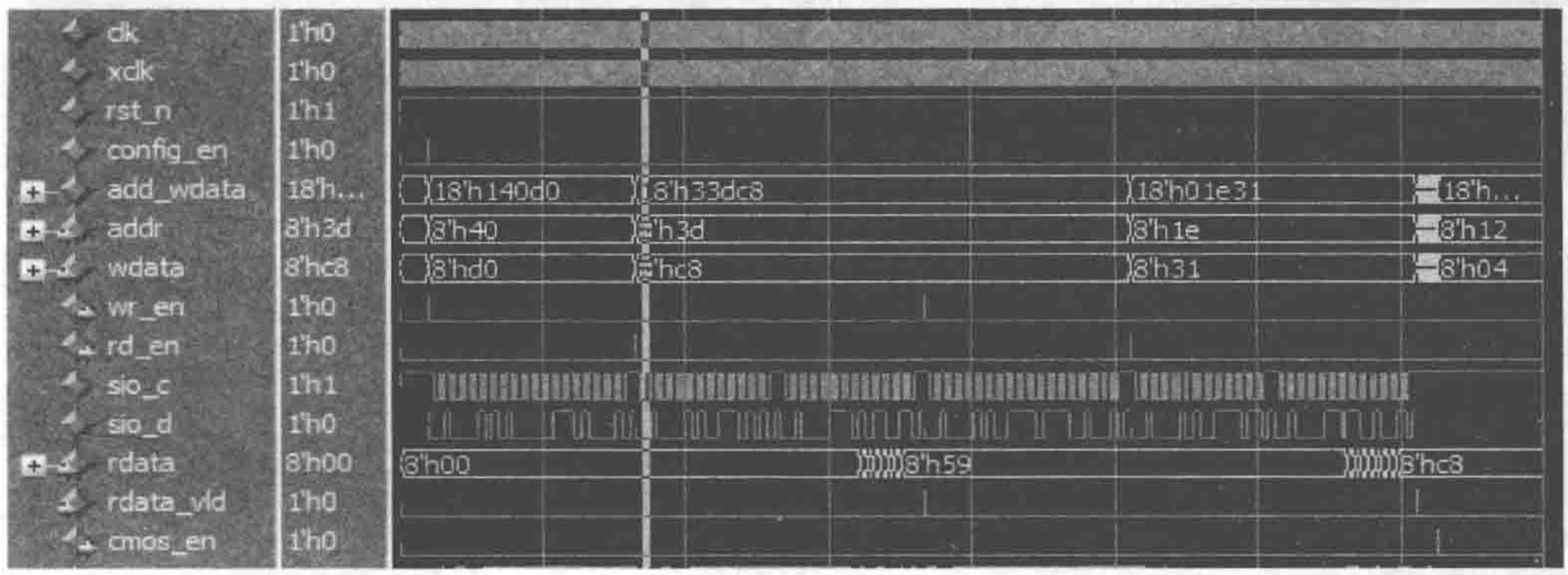


图 8－32　SCCB 配置模块仿真图 1

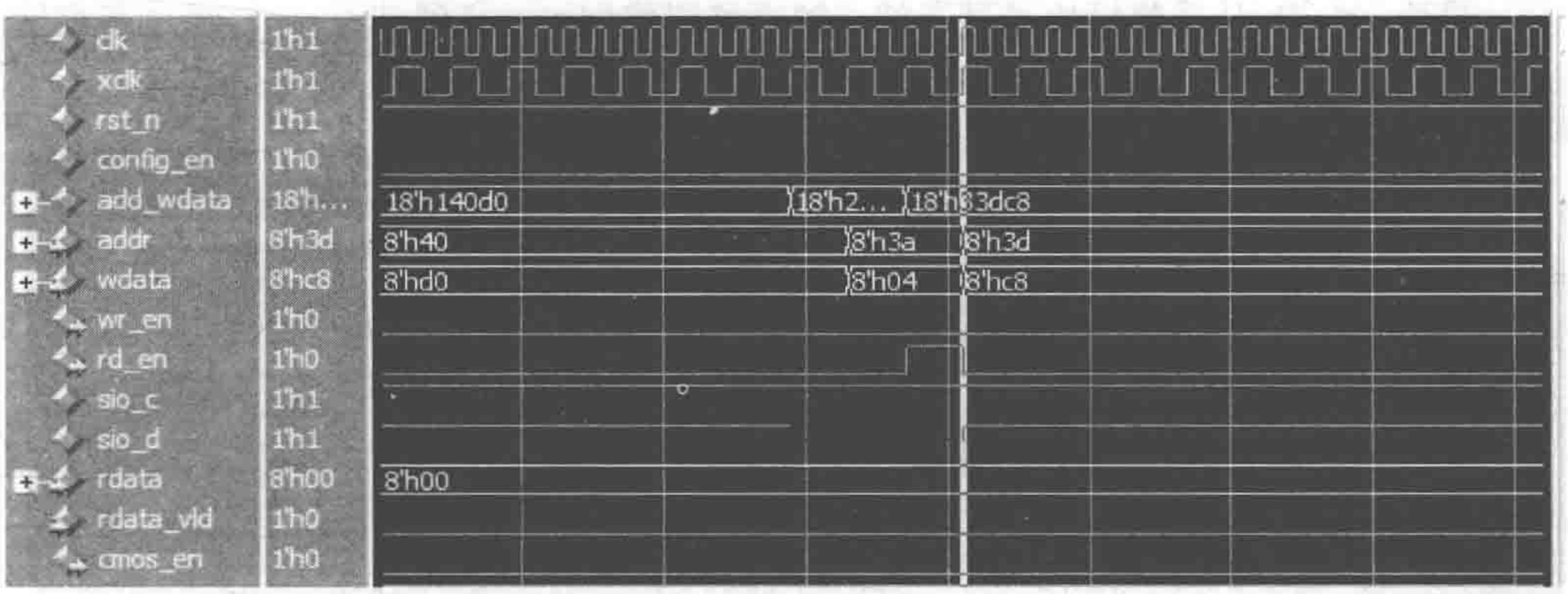


图 8－33　SCCB 配置模块仿真图 2

从图 8－32 中可以看出，由于对第一个寄存器不进行操作，所以 sio_d 开始并没有数据传输；然后对地址为 8'h40 的寄存器进行写操作，将 8'hd0 写入；接着对地址为 8'h3a 的寄存器进行读操作（见图 8－33），读出 rdata 的值为 8'h59，rdata_vld 为 1 时 rdata 稳定；最后对地址为 8'h3d 的寄存器进行写、读操作，当收到 wr_en 为 1 时，将数据 8'hc8 写入到该地址中，写完之后收到 rd_en 为 1，把该寄存器中的数据读出，值为 8'hc8。与写入的数据相同，故符合我们的设计要求。

配置模块可以通过 SCCB 接口完成配置操作，即配置模块与 SCCB 接口模块设

计正确。

2. 采集模块

如图 8-34 所示，测试文件给出连续的 8 bit 输入 din，采集模块输出 dout 为 16 bit，在第一个 dout 有效时，dout_sop 拉高，在 vys_cnt 计数结束（接收完一帧），dout_eop 拉高。图中 dout_sop 与 dout_eop 之间出现了帧同步信号 vsync 脉冲，由于此时上一帧图像没接收完，不响应此 vsync 信号，该帧图像被丢弃，即 dout_eop 到下一帧之间的 din 不接收，从而保证输出的是完整的一帧图像。

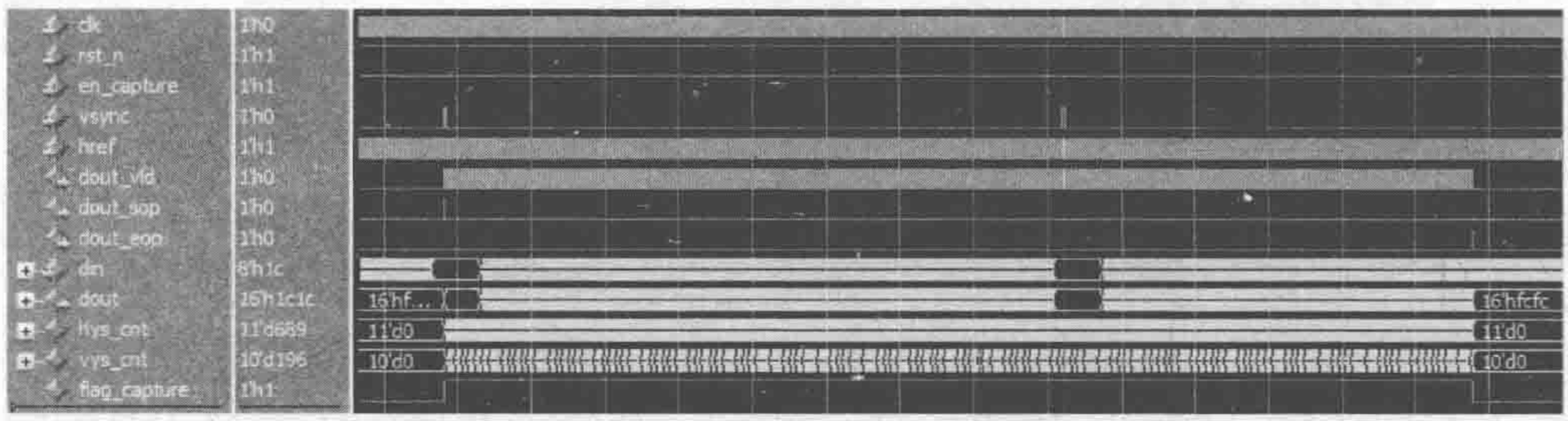


图 8-34　采集模块仿真图

3. 灰度转换模块

图 8-35 中，从采集模块送过来的一帧完整图像（din_sop 到 din_sop），经过灰度转换公式转换成相应的灰阶 dout，并以一帧图像的格式输出（dout_sop 到 dout_eop）。

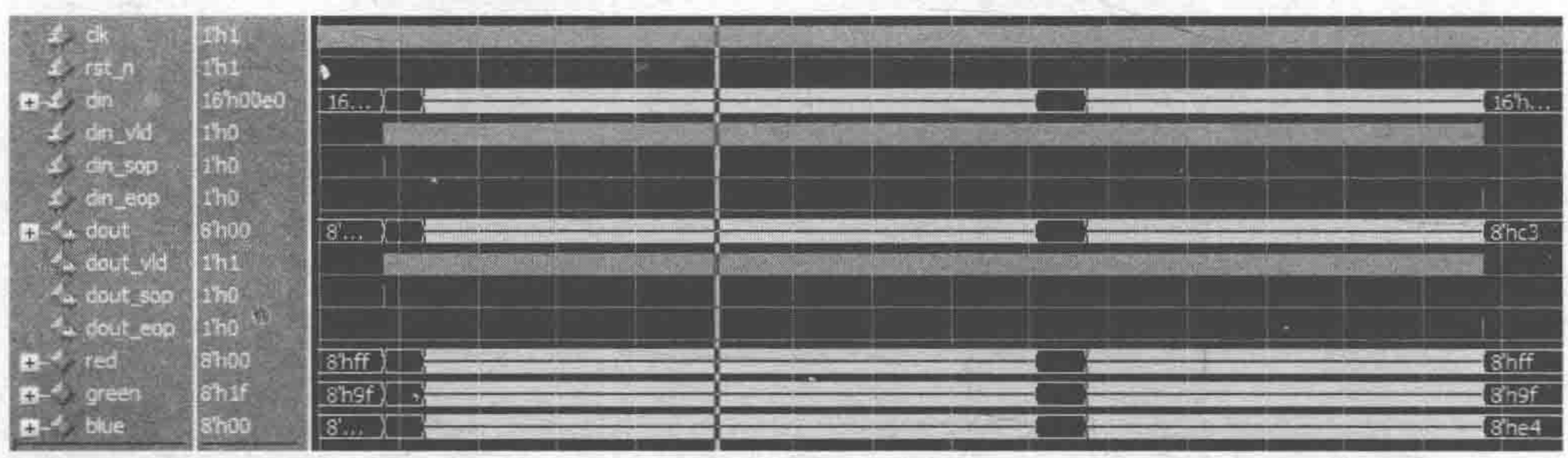


图 8-35　灰度转换模块仿真图

4.高斯滤波模块

图 8-36 中 din 是灰度转换模块数据输出，进入高斯滤波模块的移位寄存器，通过 8.3.4 小节中的高斯滤波公式，可以计算出滤波结果，输出为 dout。

5. 二值化模块

图 8-37 中 din 是高斯滤波的输出，经过二值化后单比特输出 dout。二值化中 value 的取值可以通过计算得到，也可以在上板过程中通过不断测试得到。

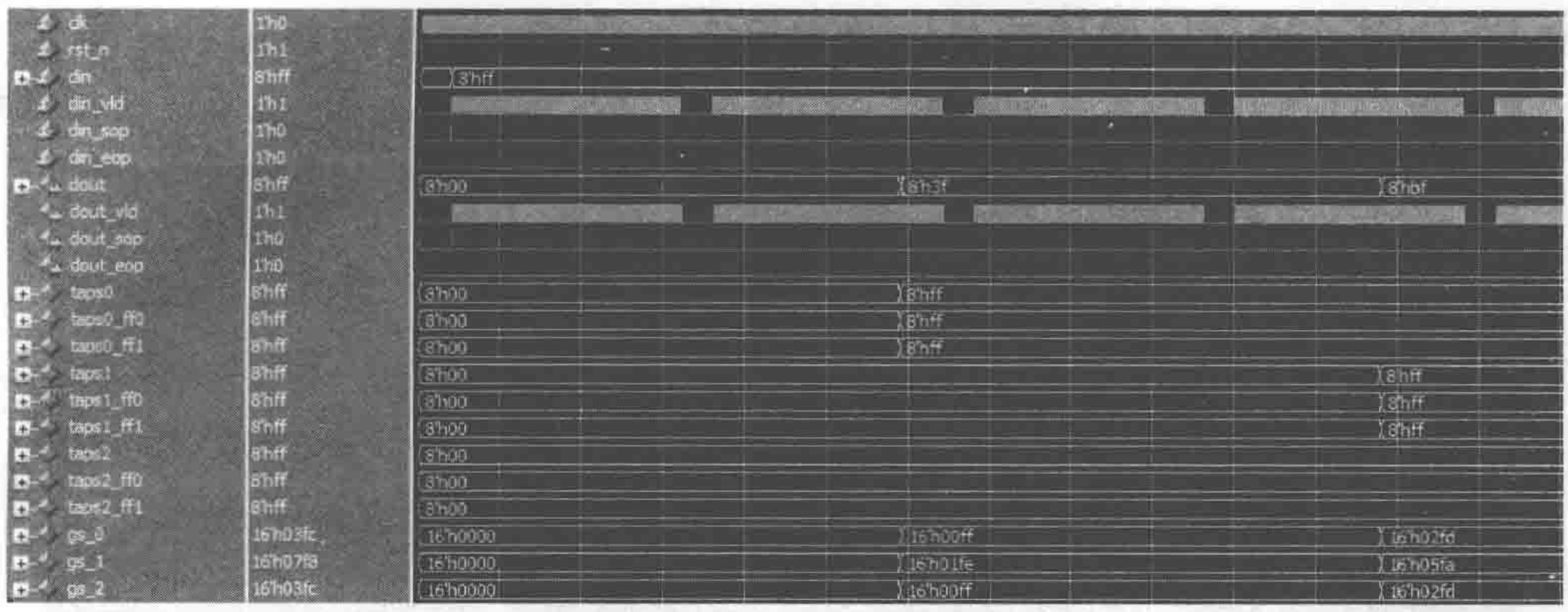


图 8－36　高斯滤波模块仿真图

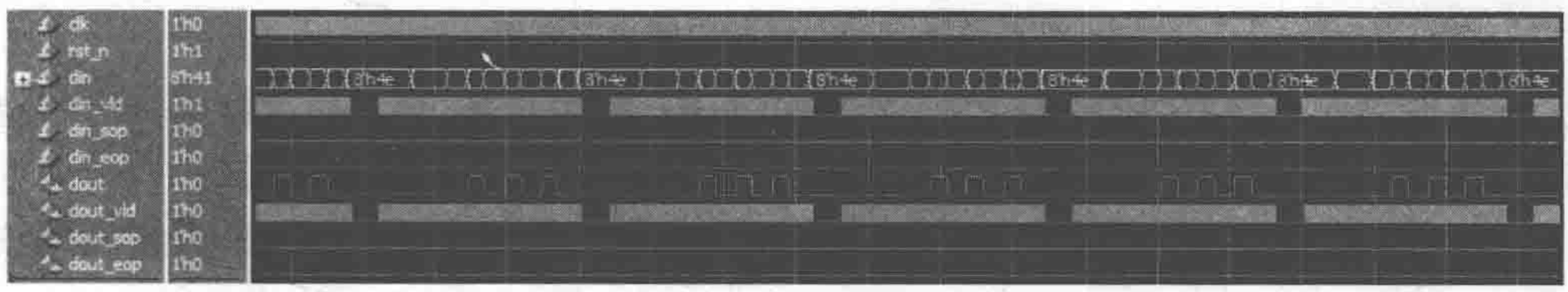


图 8－37　二值化模块仿真图

6. 边缘检测模块

图 8－38 中 din 是二值化的输出，经过 Sobel 算法之后判断出该点是否为边缘点，输出为 dout。

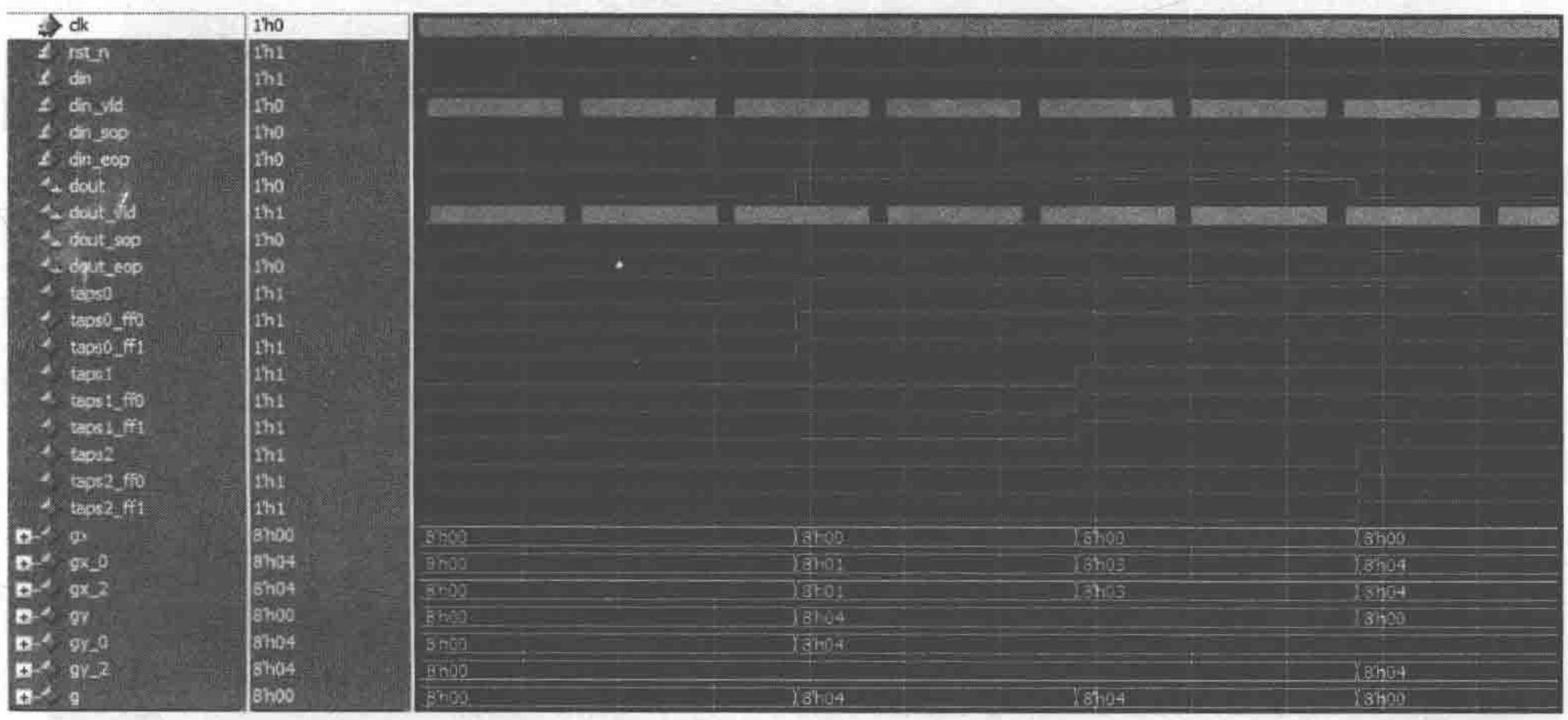


图 8－38　边缘检测模块仿真图

7. SDRAM 接口模块

SDRAM 接口模块仿真图如图 8－39 所示。

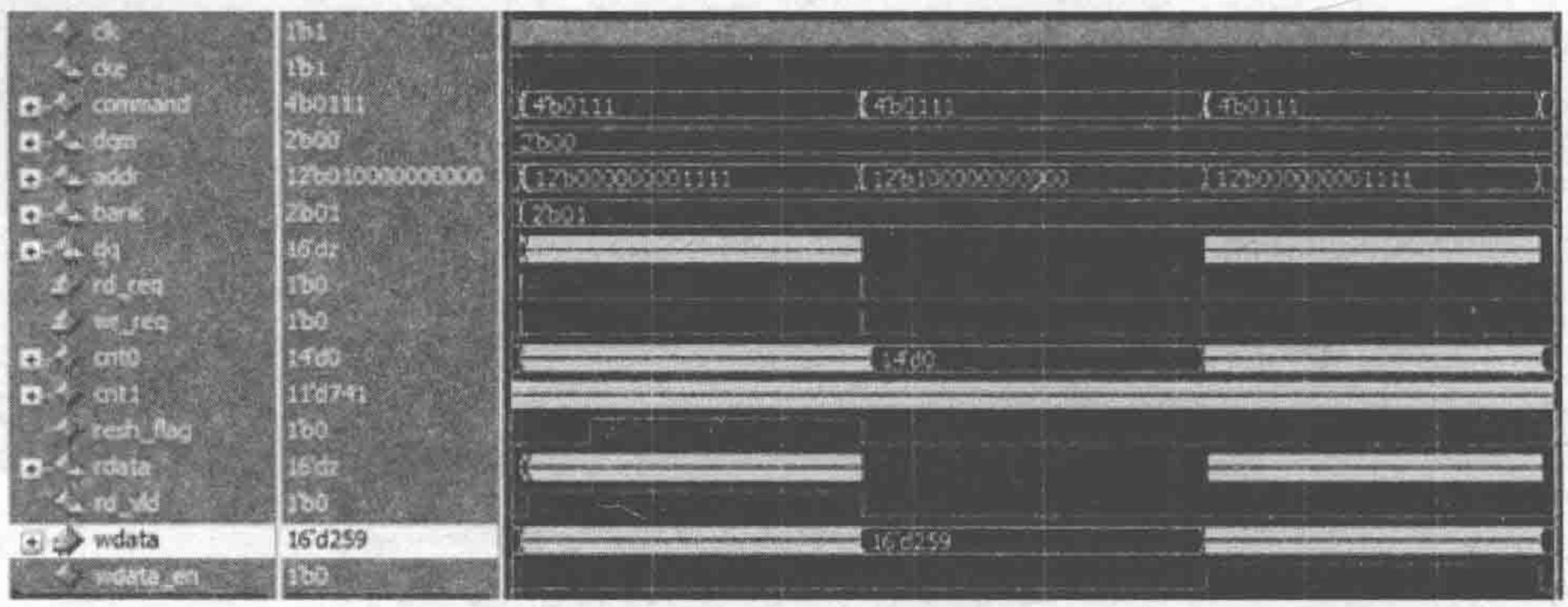


图 8－39　SDRAM 接口模块仿真图

测试文件产生 4 次同一时刻的读请求 rd_req 和写请求 wr_req，按时间顺序，将这几个时刻称为 a，b，c，d。

a 时刻是 SDRAM 初始化后收到的第一个读、写请求，按功能要求，应该响应读请求。

b 时刻是在 SDRAM 进行读请求期间又收到读、写请求，此时应该不响应。

c 时刻是在读操作结束时，收到读、写请求，且此时刷新标志有效，按功能要求，刷新标志的优先级最高，因此应该执行刷新操作。

d 时刻是 SDRAM 在 IDLE 态时收到读、写请求，由于上一次执行的是读操作，所以这次应该执行写操作。

下面观察几个关键时刻的仿真结果：

经过 2 个时钟周期(20ns)的 ACTIVE 状态后，进入读状态(4'b0110)，满足设计要求，如图 8－40 所示。

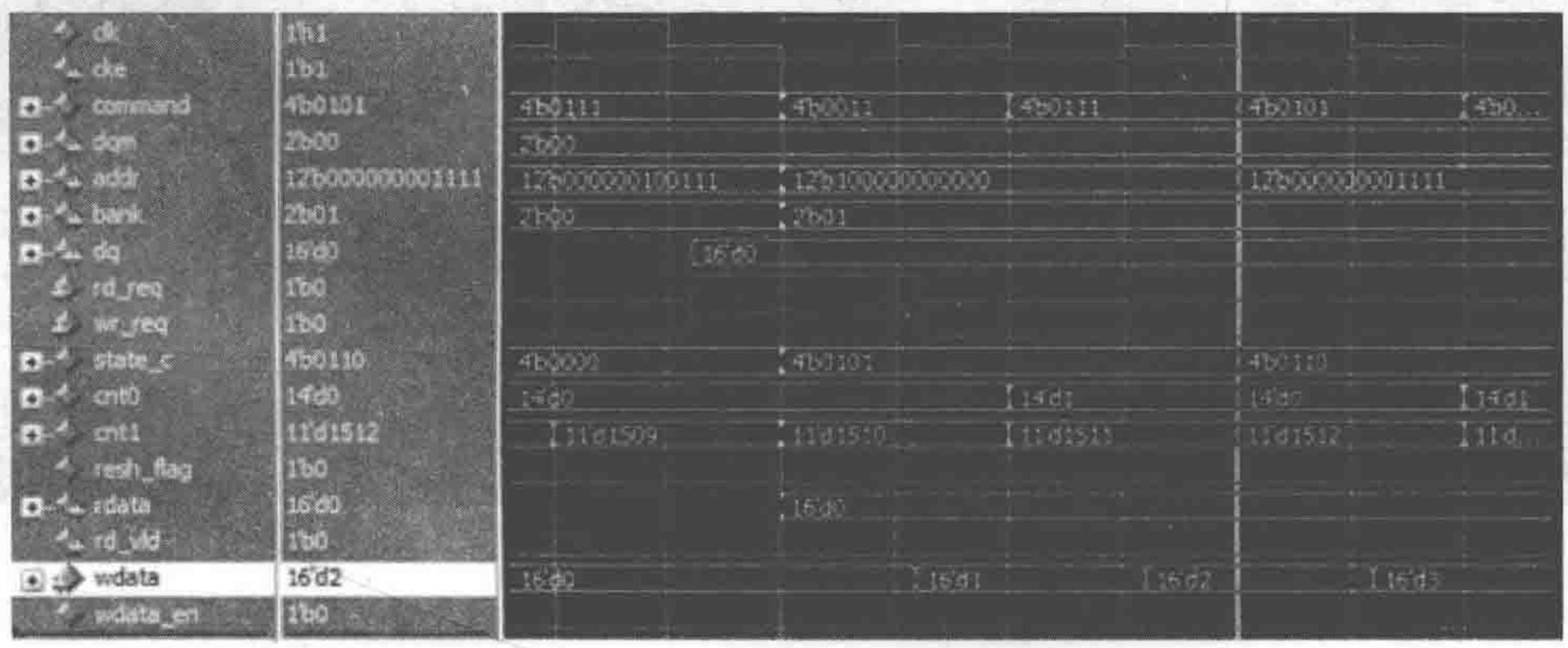


图 8－40　第一次收到读、写请求

当 cnt1 计数到 1 561 时，resh_flag 拉高，表示刷新标志有效，符合要求，如图 8－41所示。

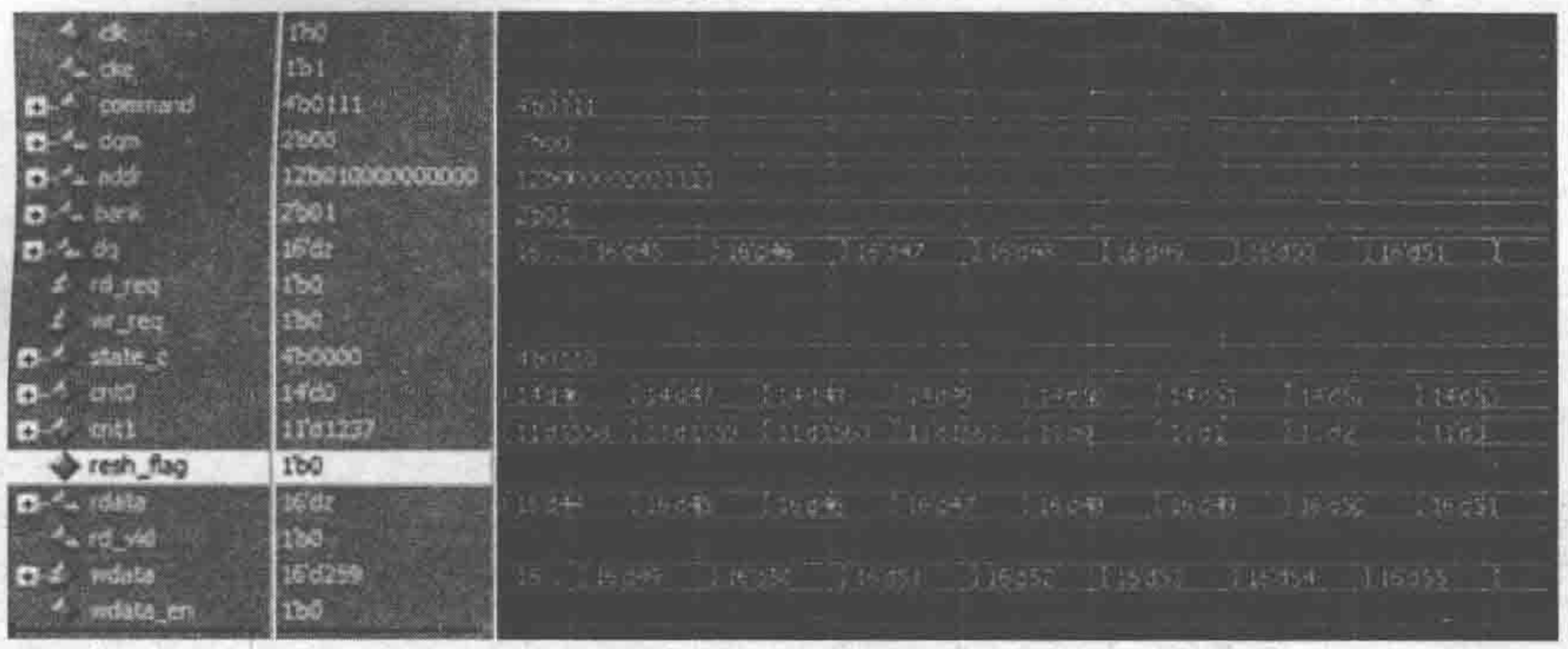


图 8－41　刷新请求产生时刻

在刷新标志有效时，收到读、写请求，SDRAM 执行自动刷新操作，不响应读、写请求，整个刷新过程持续 6 个时钟周期(60 ns)，如图 8－42 所示。

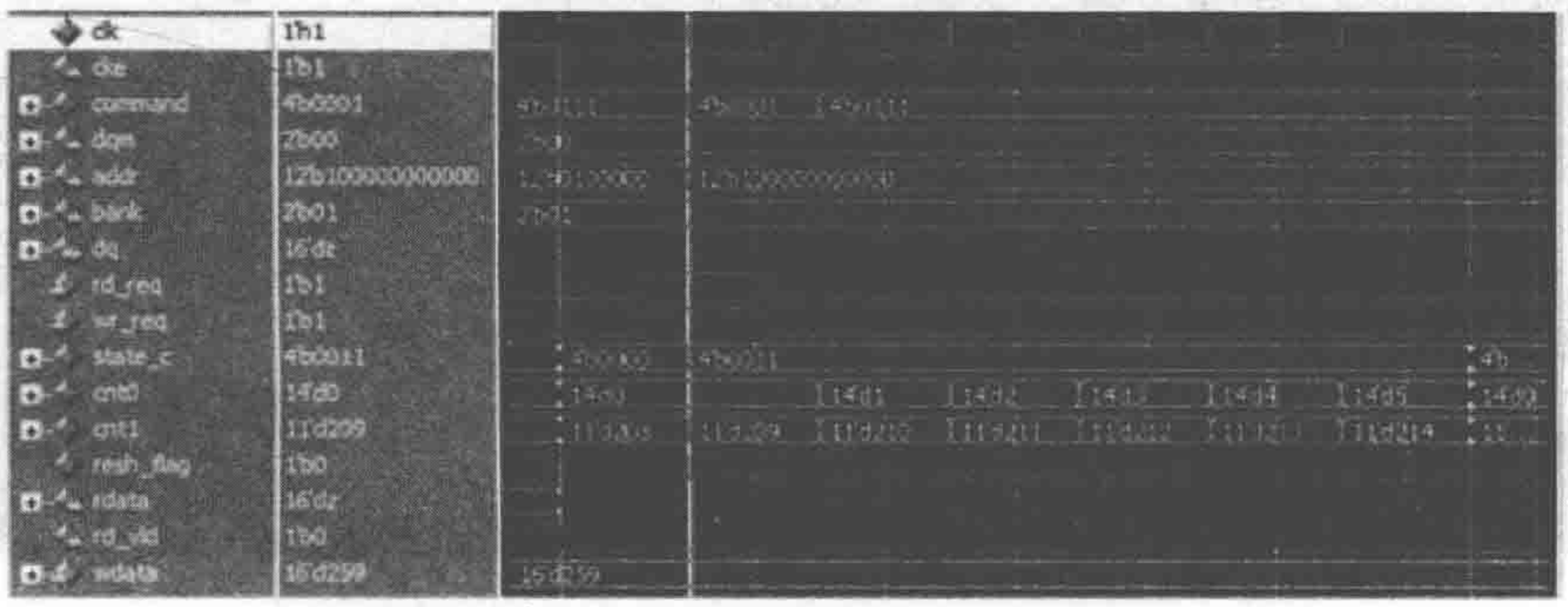


图 8－42　SDRAM 自动刷新过程

可以看出，在经过 2 个时钟周期(20 ns)的 ACTIVE 状态后，进入写状态(4'b0111)，符合设计要求中轮流读、写的处理，如图 8－43 所示。

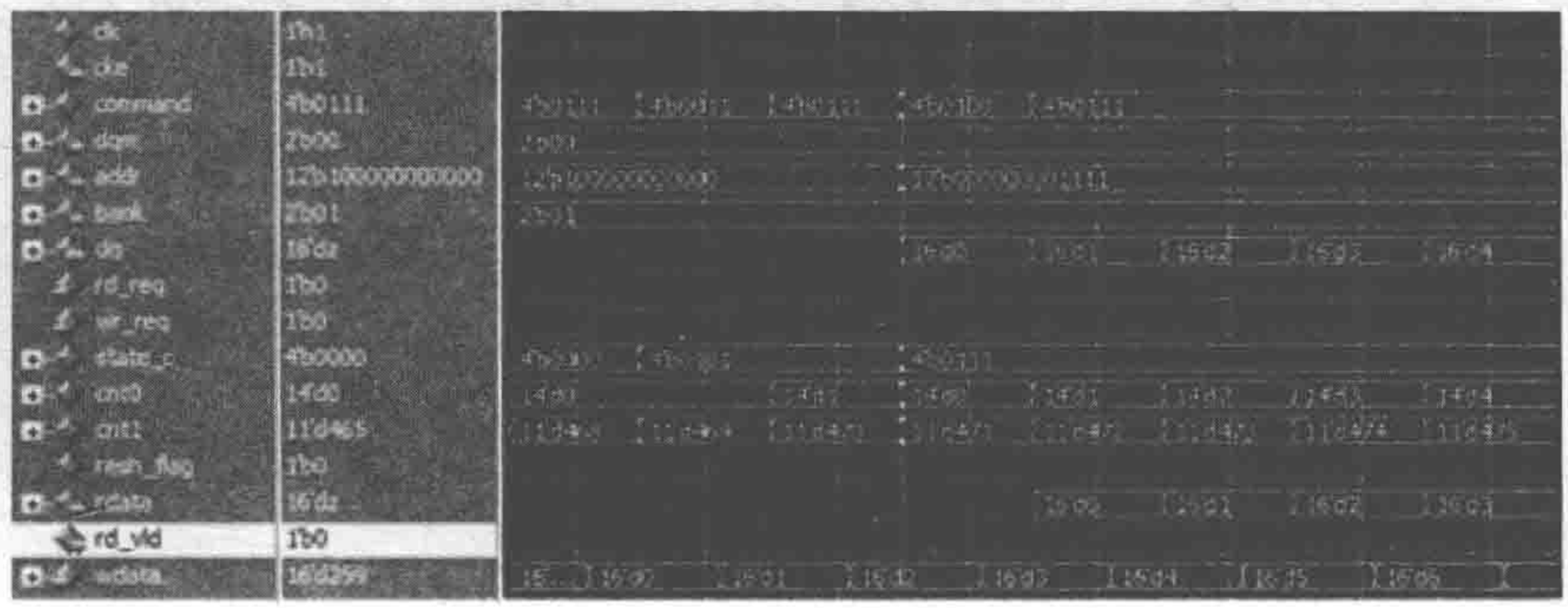


图 8－43　最后一次收到读、写请求

8. SDRAM 命令模块

与设计该模块时一样，我们也分 3 部分进行功能仿真。首先是写入图像部分。

在 wr_finish 信号有效时，表示已经接收了一副完整的图像，正在把 WFIFO 的数据写入 SDRAM，因此不响应下一帧的 din_sop 信号，如图 8－44 所示，丢弃了一帧。只有在 wr_finish 信号为低电平时，写请求 wfifo_wrreq 才有效。

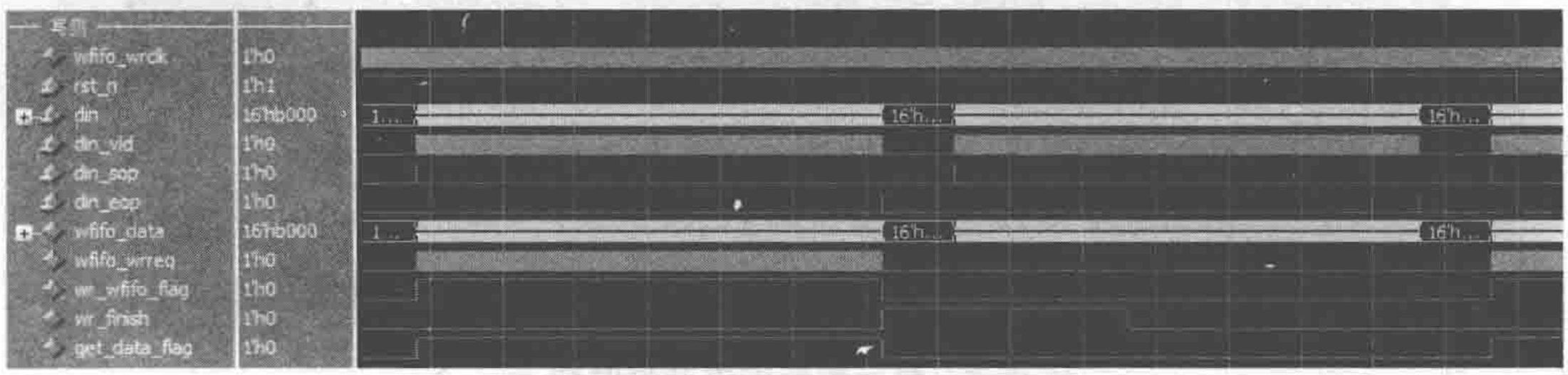


图 8－44　WFIFO 写侧仿真

在 rdusedw 为 256 且收到 wr_ack 时，WFIFO 的读使能开始有效，如图 8－45 所示，往 SDRAM 中练习写 256 个数据。

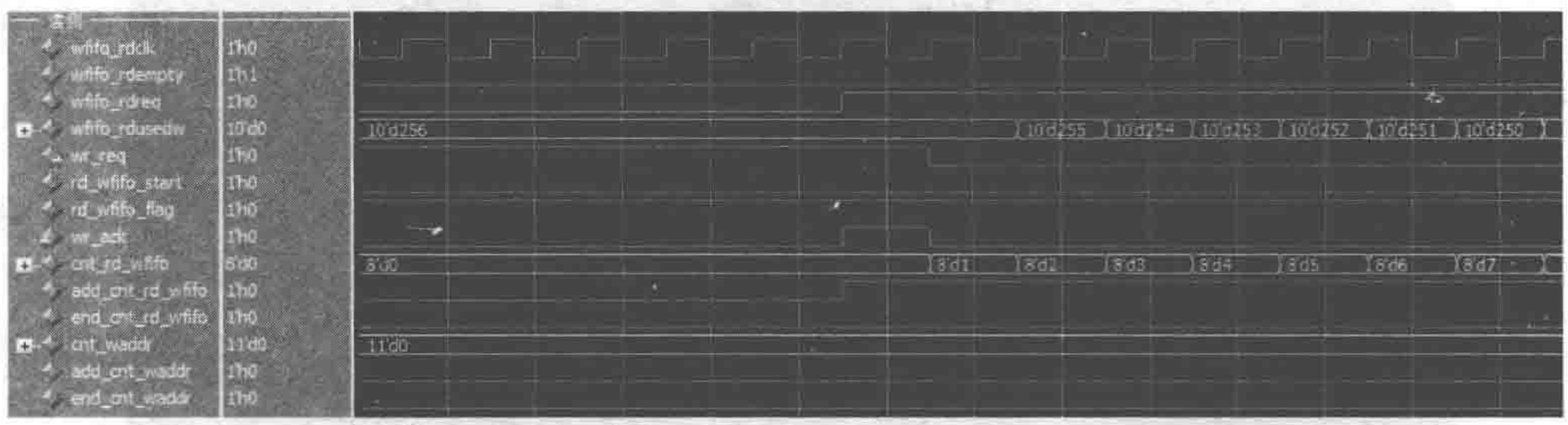


图 8－45　WFIFO 读侧起始

在读完 256×1200 ＝ 640×480(一帧)图像后，读使能拉低，如图 8－46 所示。

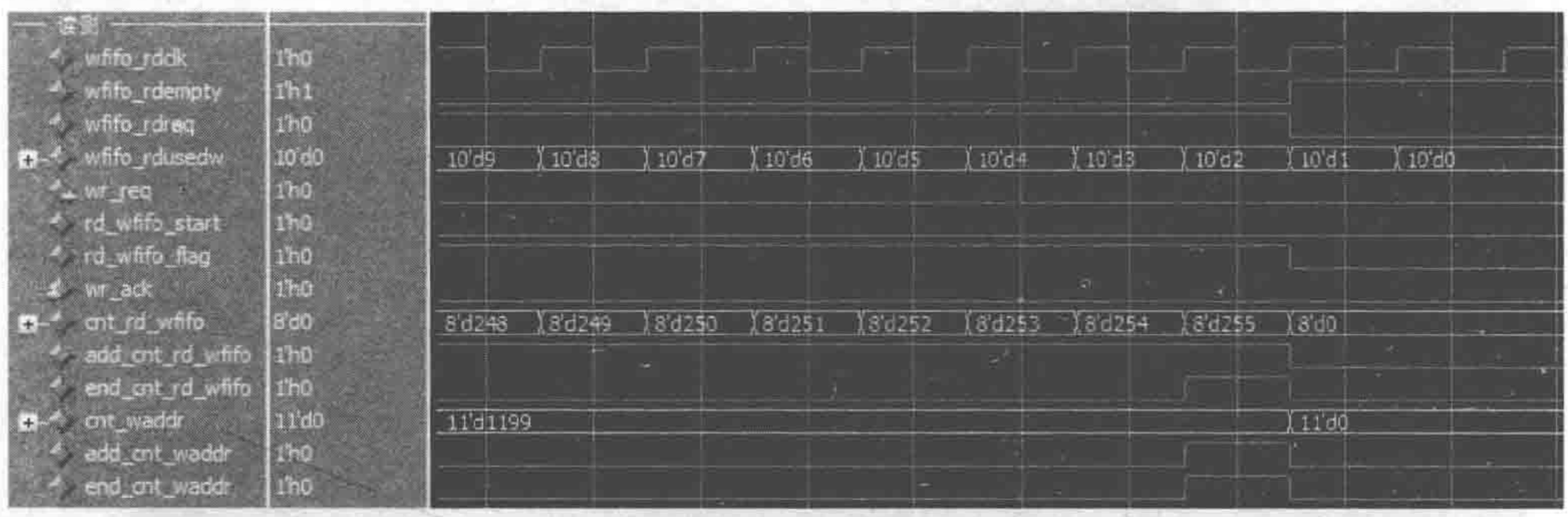


图 8－46　WFIFO 读侧结束

接下来是读取图像部分。

从图 8－47 可以看出，当 RFIFO 中的数据少于 300 个时，向 SDRAM 发出读请求，收到应答信号后，开始从 SDRAM 中读出数据。

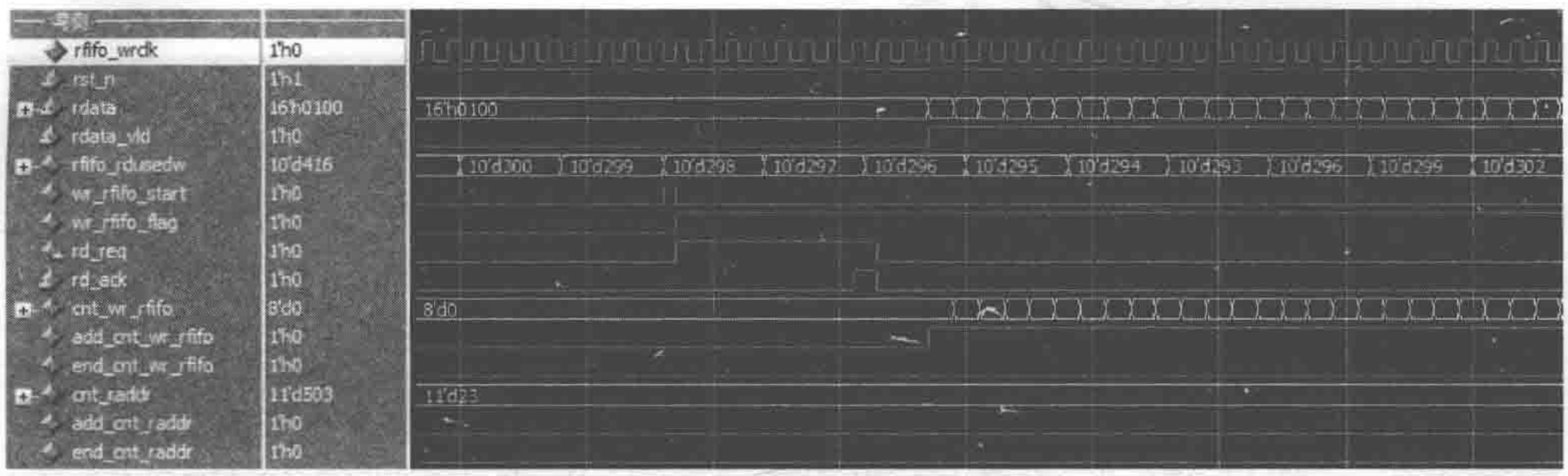


图 8－47　RFIFO 写侧起始

当读完 1 200 次连续的 256 个数据后，表明一帧完整图像已写入 RFIFO，此时结束计数，如图 8－48 所示，拉低 RFIFO 的写使能。

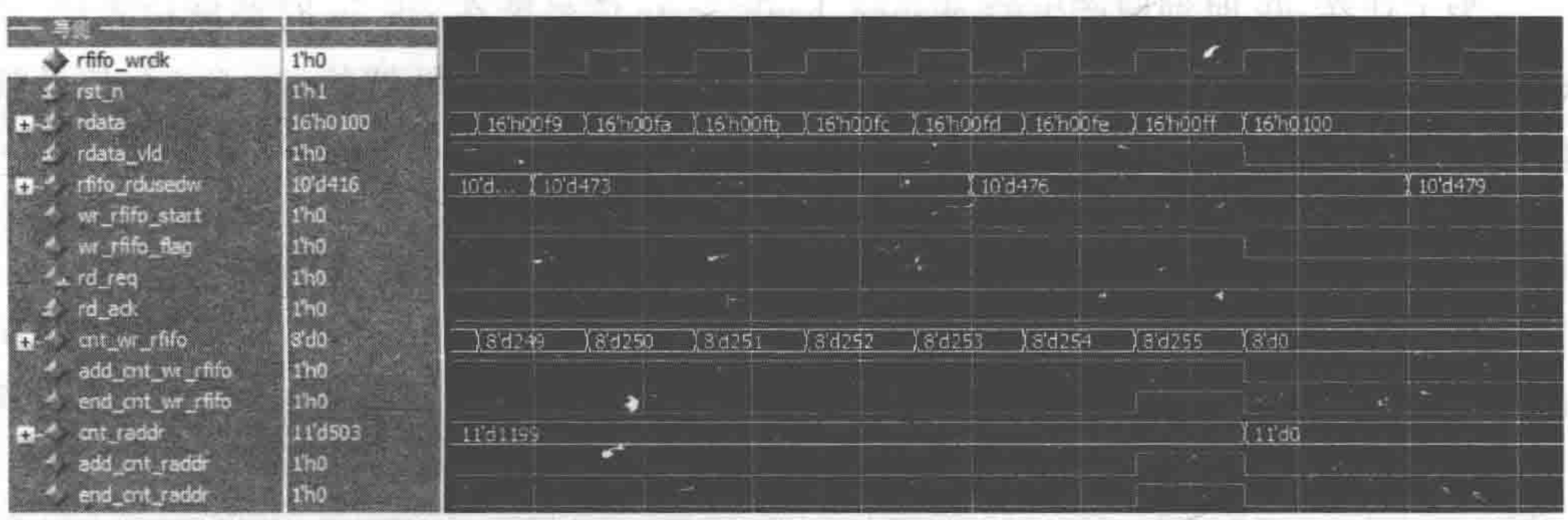


图 8－48　RFIFO 写侧结束

最后是 Bank 切换部分。

从图 8－49 可以看出，在 end_cnt_waddr 有效时刻，waddr_bank 切换；在 end_cnt_raddr 有效时刻，raddr_bank 切换。

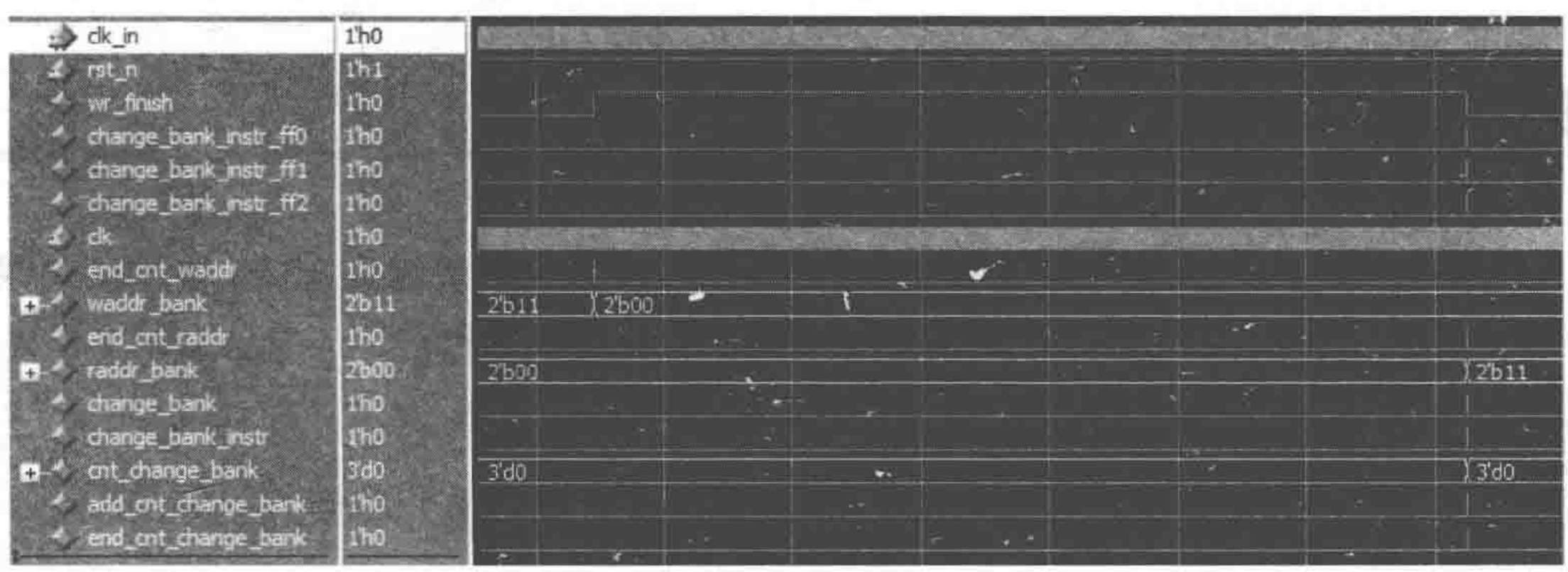


图 8－49　Bank 切换时刻

再看 bank 切换指示信号 change_bank_instr 跨时钟域检测，如图 8－50 所示。

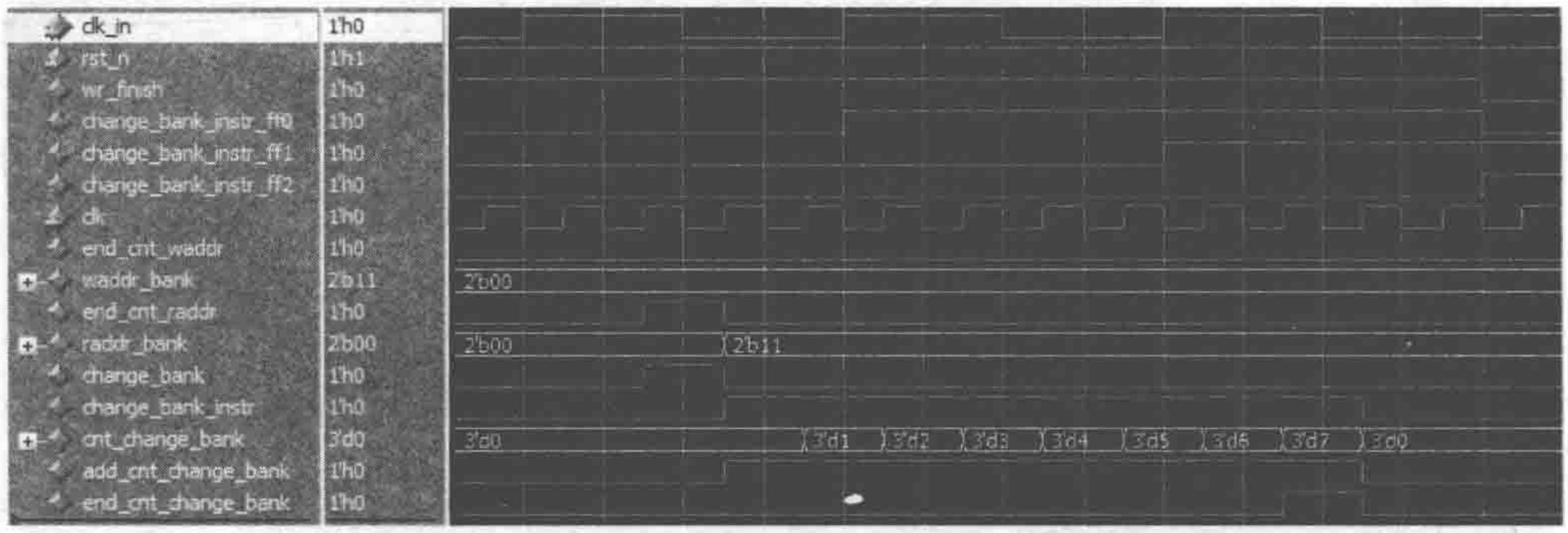


图 8－50　change_bank_instr 跨时钟域检测

为了让在 clk 时钟域产生的 change_bank_instr 信号能在 clk_in 时钟域检测到，对 change_bank_instr 进行延时，再检测其上升沿，这样 clk_in 时钟域的 wr_finish 信号就能采集到 change_bank_instr 信号的变化点。

9. VGA 接口模块

从图 8－51 可以看出，在 VGA 行场信号的有效区域，dout_rd 拉高，在 SDRAM 命令模块中读取相应像素点的值，并转换成 8 位位宽 rgb_data 输出。

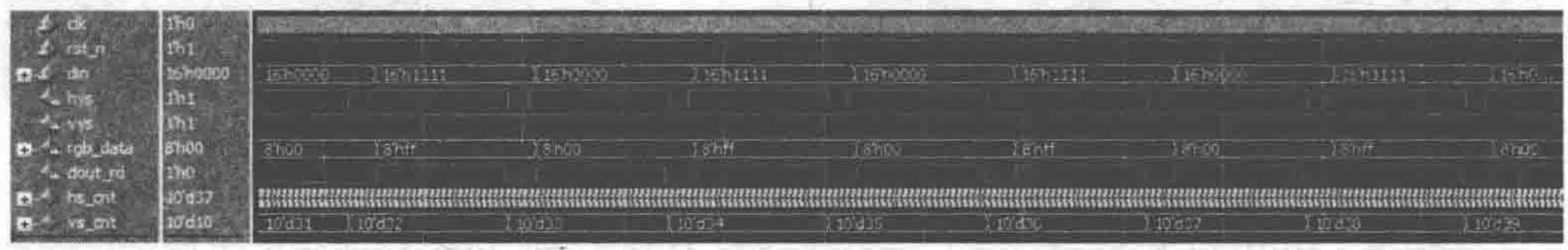


图 8－51　VGA 接口模块仿真图

8.7　板级调试

图 8－52 是摄像头 OV7670 采集并经过 FPGA 处理，最终用 VGA 输出的效果图。可以看到，图像的边缘比较清晰，并可以看出大致的轮廓。

图 8-52　边缘检测效果图

8.8 总　结

本设计使用高斯平滑滤波与 Sobel 边缘检测算子来进行图像处理，并用 SDRAM 进行图像的缓存，其中还用到乒乓操作。由于涉及的原理较多，读者要先理解 8.3 节中的基本原理，再进行设计或者查看参考代码。

参考文献

[1] Samir Palnitkar. Verilog HDL A Guide to Digital Design and Synthes[M].2ed. [S.l.]:SunSoft Press A Prentice Hall Title,2003.

[2] Stuart Sutherland. Verilog 2001:A Guide to the New Features of Verilog Hardware Description Language[M]. [S.l.]:Kluwer Academic Publishers,1998.

[3] 王诚,蔡海宁,吴继华. Altera FPGA/CPLD 设计:基础篇[M].2 版.北京:人民邮电出版社,2011.

[4] 王诚,蔡海宁,吴继华. Altera FPGA/CPLD 设计:高级篇[M]. 2 版.北京:人民邮电出版社,2011.

[5] 夏宇闻.Verilog 数字系统设计教程[M].3 版.北京:北京航空航天大学出版社,2013.

策划编辑：董立娟
封面设计：runsign

作者简介

潘文明，广东韶关人，暨南大学通信与信息系统专业硕士；现任明德扬科技教育有限公司总经理、广州敏道信息科技有限公司理论研究所名誉所长、东莞市瑞航信息科技有限公司高级技术顾问；先后在中国电子科技集团第七研究所和华为海思半导体公司参与了跳频-OFDM 系统、超高速网络芯片、工业以太网相机等大型项目研发。

在深厚的理论基础和丰富的实际项目开发经验基础上，发表和出版了一系列相关专业论文和书籍，如《DES 加密芯片的研制及其实现》、*Low-Power Design of Ethernet Data Transmission*、《点拨 FPGA 教程》、《至简设计法教程》等，深受广大 FPGA 学习者喜爱。

内容简介

本书介绍利用硬件描述语言(Verilog HDL)设计数字系统的方法，提出计数器、状态机、FIFO 的三大架构，采用理论与实践相结合的方式，详细介绍各个架构的实现步骤——“八步法”，并配以相应的项目练习。在书的后半部分，介绍模块划分的基本方法，总结并提出了模块间的交互架构，在项目实践中给出了温度监控系统、边缘检测系统等综合性项目的模式。

上架建议：自动化技术/嵌入式系

ISBN 978-7-5124-2401

9 787512 424012 >

定价：45.00元