国外电子与通信教材系列

异步电路设计原理
——系统透视

Principles of Asynchronous Circuit Design
A Systems Perspective

［丹］ Jens Sparsø

［英］ Steve Furber 编著

赵不贿　徐雷钧　孙智权　等译

電子工業出版社

Publishing House of Electronics Industry

北京 · BEIJING

内容简介

欧盟的“低功耗设计领航行动”被誉为欧洲的低功耗电子系统设计开端，旨在研究和论证新的降低功耗的设计方法，包括19个研发项目和1个配套项目，该配套项目用于确保这些方法、实验和结果的整理与出版。

本书是该配套项目出版的“新型低功耗设计结构、方法和设计实践”系列丛书的第三册。全书由三部分组成，第一部分是异步电路设计教程，主要介绍异步电路设计的基本理论，使初学者理解异步电路的特性，学会如何思考异步系统，跳出传统的时钟设计的思维框架，能够用高级语言和工具创造性地开发出新的设计方案。第二部分介绍了异步电路描述语言——高级综合系统Balsa。第三部分给出了一些复杂异步系统设计实例，包括飞利浦公司设计的商用智能卡芯片和曼彻斯特大学设计的异步Viterbi解码器以及Amulet系列异步处理器。

本书是专为那些具有传统的(时钟的)数字设计背景而又希望对异步设计有所了解的设计人员而写的入门教材，适合作为电子、通信、计算机、测量和控制等专业高年级本科生、研究生的教材和自学参考书，也可供教师、科研人员及工程技术人员参考。

版权贸易合同登记号　图字：01-2007-4099

图书在版编目（CIP）数据

异步电路设计原理：系统透视/（丹）斯派思（Sparsø, J.），（英）弗伯（Furber, S.）编著. 赵不贿等译.
北京：电子工业出版社，2009.1
（国外电子与通信教材系列）
书名原文：Principles of Asynchronous Circuit Design: A Systems Perspective
ISBN 978-7-121-07671-8

Ⅰ.异…　Ⅱ.①斯…　②弗…　③赵…　Ⅲ.电路设计-教材　Ⅳ.TM02

中国版本图书馆CIP数据核字（2008）第169199号

策划编辑：谭海平
责任编辑：秦淑灵
印　　刷：北京市顺义兴华印刷厂
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　邮编：100036
开　　本：787 × 980　1/16　　印张：19.25　　字数：430千字
印　　次：2009年1月第1次印刷
定　　价：49.00元

凡所购买电子工业出版社的图书有缺损问题，请向购买书店调换；若书店售缺，请与本社发行部联系。联系及邮购电话：（010）88254888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：（010）88258888。

序

2001年7月间，电子工业出版社的领导同志邀请各高校十几位通信领域方面的老师，商量引进国外教材问题。与会同志对出版社提出的计划十分赞同，大家认为，这对我国通信事业、特别是对高等院校通信学科的教学工作会很有好处。

教材建设是高校教学建设的主要内容之一。编写、出版一本好的教材，意味着开设了一门好的课程，甚至可能预示着一个崭新学科的诞生。20世纪40年代MIT林肯实验室出版的一套28本雷达丛书，对近代电子学科、特别是对雷达技术的推动作用，就是一个很好的例子。

我国领导部门对教材建设一直非常重视。20世纪80年代，在原教委教材编审委员会的领导下，汇集了高等院校几百位富有教学经验的专家，编写、出版了一大批教材；很多院校还根据学校的特点和需要，陆续编写了大量的讲义和参考书。这些教材对高校的教学工作发挥了极好的作用。近年来，随着教学改革不断深入和科学技术的飞速进步，有的教材内容已比较陈旧、落后，难以适应教学的要求，特别是在电子学和通信技术发展神速、可以讲是日新月异的今天，如何适应这种情况，更是一个必须认真考虑的问题。解决这个问题，除了依靠高校的老师和专家撰写新的符合要求的教科书外，引进和出版一些国外优秀电子与通信教材，尤其是有选择地引进一批英文原版教材，是会有好处的。

一年多来，电子工业出版社为此做了很多工作。他们成立了一个“国外电子与通信教材系列”项目组，选派了富有经验的业务骨干负责有关工作，收集了230余种通信教材和参考书的详细资料，调来了100余种原版教材样书，依靠由20余位专家组成的出版委员会，从中精选了40多种，内容丰富，覆盖了电路理论与应用、信号与系统、数字信号处理、微电子、通信系统、电磁场与微波等方面，既可作为通信专业本科生和研究生的教学用书，也可作为有关专业人员的参考材料。此外，这批教材，有的翻译为中文，还有部分教材直接影印出版，以供教师用英语直接授课。希望这些教材的引进和出版对高校通信教学和教材改革能起一定作用。

在这里，我还要感谢参加工作的各位教授、专家、老师与参加翻译、编辑和出版的同志们。各位专家认真负责、严谨细致、不辞辛劳、不怕琐碎和精益求精的态度，充分体现了中国教育工作者和出版工作者的良好美德。

随着我国经济建设的发展和科学技术的不断进步，对高校教学工作会不断提出新的要求和希望。我想，无论如何，要做好引进国外教材的工作，一定要联系我国的实际。教材和学术专著不同，既要注意科学性、学术性，也要重视可读性，要深入浅出，便于读者自学；引进的教材要适应高校教学改革的需要，针对目前一些教材内容较为陈旧的问题，有目的地引进一些先进的和正在发展中的交叉学科的参考书；要与国内出版的教材相配套，安排好出版英文原版教材和翻译教材的比例。我们努力使这套教材能尽量满足上述要求，希望它们能放在学生们的课桌上，发挥一定的作用。

最后，预祝“国外电子与通信教材系列”项目取得成功，为我国电子与通信教学和通信产业的发展培土施肥。也恳切希望读者能对这些书籍的不足之处、特别是翻译中存在的问题，提出意见和建议，以便再版时更正。

吴佑寿

中国工程院院士、清华大学教授

“国外电子与通信教材系列”出版委员会主任

出 版 说 明

进入21世纪以来，我国信息产业在生产和科研方面都大大加快了发展速度，并已成为国民经济发展的支柱产业之一。但是，与世界上其他信息产业发达的国家相比，我国在技术开发、教育培训等方面都还存在着较大的差距。特别是在加入WTO后的今天，我国信息产业面临着国外竞争对手的严峻挑战。

作为我国信息产业的专业科技出版社，我们始终关注着全球电子信息技术的发展方向，始终把引进国外优秀电子与通信信息技术教材和专业书籍放在我们工作的重要位置上。在2000年至2001年间，我社先后从世界著名出版公司引进出版了40余种教材，形成了一套“国外计算机科学教材系列”，在全国高校以及科研部门中受到了欢迎和好评，得到了计算机领域的广大教师与科研工作者的充分肯定。

引进和出版一些国外优秀电子与通信教材，尤其是有选择地引进一批英文原版教材，将有助于我国信息产业培养具有国际竞争能力的技术人才，也将有助于我国国内在电子与通信教学工作中掌握和跟踪国际发展水平。根据国内信息产业的现状、教育部《关于“十五”期间普通高等教育教材建设与改革的意见》的指示精神以及高等院校老师们反映的各种意见，我们决定引进“国外电子与通信教材系列”，并随后开展了大量准备工作。此次引进的国外电子与通信教材均来自国际著名出版商，其中影印教材约占一半。教材内容涉及的学科方向包括电路理论与应用、信号与系统、数字信号处理、微电子、通信系统、电磁场与微波等，其中既有本科专业课程教材，也有研究生课程教材，以适应不同院系、不同专业、不同层次的师生对教材的需求，广大师生可自由选择和自由组合使用。我们还将与国外出版商一起，陆续推出一些教材的教学支持资料，为授课教师提供帮助。

此外，“国外电子与通信教材系列”的引进和出版工作得到了教育部高等教育司的大力支持和帮助，其中的部分引进教材已通过“教育部高等学校电子信息科学与工程类专业教学指导委员会”的审核，并得到教育部高等教育司的批准，纳入了“教育部高等教育司推荐——国外优秀信息科学与技术系列教学用书”。

为做好该系列教材的翻译工作，我们聘请了清华大学、北京大学、北京邮电大学、南京邮电大学、东南大学、西安交通大学、天津大学、西安电子科技大学、电子科技大学、中山大学、哈尔滨工业大学、西南交通大学等著名高校的教授和骨干教师参与教材的翻译和审校工作。许多教授在国内电子与通信专业领域享有较高的声望，具有丰富的教学经验，他们的渊博学识从根本上保证了教材的翻译质量和专业学术方面的严格与准确。我们在此对他们的辛勤工作与贡献表示衷心的感谢。此外，对于编辑的选择，我们达到了专业对口；对于从英文原书中发现的错误，我们通过与作者联络、从网上下载勘误表等方式，逐一进行了修订；同时，我们对审校、排版、印制质量进行了严格把关。

今后，我们将进一步加强同各高校教师的密切关系，努力引进更多的国外优秀教材和教学参考书，为我国电子与通信教材达到世界先进水平而努力。由于我们对国内外电子与通信教育的发展仍存在一些认识上的不足，在选题、翻译、出版等方面的工作中还有许多需要改进的地方，恳请广大师生和读者提出批评及建议。

电子工业出版社

教材出版委员会

主　任	吴佑寿	中国工程院院士、清华大学教授
副主任	林金桐	北京邮电大学校长、教授、博士生导师
	杨千里	总参通信部副部长，中国电子学会会士、副理事长 中国通信学会常务理事、博士生导师
委　员	林孝康	清华大学教授、博士生导师、电子工程系副主任、通信与微波研究所所长 教育部电子信息科学与工程类专业教学指导分委员会委员 清华大学深圳研究生院副院长
	徐安士	北京大学教授、博士生导师、电子学系主任
	樊昌信	西安电子科技大学教授、博士生导师 中国通信学会理事、IEEE 会士
	程时昕	东南大学教授、博士生导师
	郁道银	天津大学副校长、教授、博士生导师 教育部电子信息科学与工程类专业教学指导分委员会委员
	阮秋琦	北京交通大学教授、博士生导师 计算机与信息技术学院院长、信息科学研究所所长 国务院学位委员会学科评议组成员
	张晓林	北京航空航天大学教授、博士生导师、电子信息工程学院院长 教育部电子信息科学与电气信息类基础课程教学指导分委员会副主任委员 中国电子学会常务理事
	郑宝玉	南京邮电大学副校长、教授、博士生导师 教育部电子信息科学与工程类专业教学指导分委员会副主任委员
	朱世华	西安交通大学副校长、教授、博士生导师 教育部电子信息科学与工程类专业教学指导分委员会副主任委员
	彭启琮	电子科技大学教授、博士生导师
	毛军发	上海交通大学教授、博士生导师、电子信息与电气工程学院副院长 教育部电子信息与电气学科教学指导委员会委员
	赵尔沅	北京邮电大学教授、《中国邮电高校学报（英文版）》编委会主任
	钟允若	原邮电科学研究院副院长、总工程师
	刘　彩	中国通信学会副理事长兼秘书长，教授级高工 信息产业部通信科技委副主任
	杜振民	电子工业出版社原副社长
	王志功	东南大学教授、博士生导师、射频与光电集成电路研究所所长 教育部高等学校电子电气基础课程教学指导分委员会主任委员
	张中兆	哈尔滨工业大学教授、博士生导师、电子与信息技术研究院院长
	范平志	西南交通大学教授、博士生导师、信息科学与技术学院院长

译 者 序

众所周知，时序逻辑电路分为同步电路和异步电路两种，二者之间的主要区别在于电路的控制机制。同步电路受统一时钟信号的控制，采用的是一种时钟控制机制；而异步电路则用握手替代统一时钟，采用的是一种数据驱动的控制机制。由于控制机制不同，电路的特点也不一样。

同步电路因设计相对简单，且具有丰富的设计工具支持而逐渐占据电路设计的主流。异步电路则由于设计复杂、缺乏工程化的设计验证手段和成熟的设计工具支持，发展一直比较缓慢。

随着半导体工艺进入到超深亚微米阶段，高速同步电路的设计遇到了前所未有的挑战，时钟分布、功耗、时序收敛、工艺偏差、设计复杂性等问题日益突出。而异步电路没有统一的时钟，也就不存在时钟偏差的问题，并以其功耗低、高速、辐射低、模块化程度高、移植性好和平均效率的性能指标等优良特性，逐渐受到人们的重视。

1997年由欧盟信息技术计划“Esprit”所发起的“低功耗设计领航行动”被誉为欧洲的低功耗电子系统设计开端，旨在研究和论证新的降低功耗的设计方法，包括19个研发项目和1个用于确保这些方法、实验和结果的整理与出版的配套项目。在该项目的资助下，出版了“新型低功耗设计结构、方法和设计实践”系列丛书，*Principles of Asynchronous Circuit Design:A Systems Perspective* 是该丛书的第三册。

本书的编著者参与了许多大型项目的设计与研究工作，有丰富的设计经验。书中不仅详细介绍了异步电路设计的基本理论，还介绍了国际最新研究成果，内容由浅入深，图文并茂，是为那些暂时还不太了解而又希望能够尽快熟悉异步技术的设计者编写的，是初学者入门的好教材。

我们深知，翻译这样一本著作面临巨大的挑战并将承受很大的压力，但抑制不住我们将它介绍给国内读者的愿望，因为国内异步电路的研究刚刚起步，介绍异步电路设计的中文资料相当匮乏。

本书第一部分由赵不贿和谢晔翻译，第二部分由孙智权和白雪翻译，第三部分由徐雷钧和孙智权翻译，全书由赵不贿修改定稿。硕士研究生周辉、杨旻参加了部分工作。东南大学博士研究生张亮仔细阅读了初稿，提出了许多宝贵的意见和建议，在此致以诚挚的谢意。

由于译者水平有限，书中定有不少错误之处，恳请读者批评指正，以便重印时改正。邮件请发至 zbh2007@21cn.com。

译者于江苏大学

2008.11

前　言

本书的编写旨在为学习异步设计的读者提供一本入门性教材。有关异步设计已经出版了几本专业性的著作，但出发点显然不是为那些希望能够尽快熟悉异步技术的初学者编写的，我们希望本书能成为初学者的入门教材。

本书需要读者具有一定的数字设计的基础，熟悉诸如逻辑门、触发器和布尔逻辑等概念。在学习本书后续一些章节时，需要读者熟悉一些高级数字设计的知识，例如微处理器结构和片上系统（ Systems-On-Chip, SOC ）。但即使读者对于这些高级数字设计知识不是很熟悉，也不会妨碍其对本书大部分章节的学习。

本书的预期读者是

- 业界设计人员：那些具有传统的（时钟的）数字设计背景而又希望对异步设计有一些了解的设计者。例如，通过掌握一些异步设计的知识能够更好地对他们今后的设计任务进行评估，用以判断采用异步技术是否能获得更好的效果。
- 学生：正在学习有关异步设计课程的电子或计算机工程专业的学生。

本书由三部分构成。第一部分是异步电路设计教程，主要为初学者介绍最重要的问题，即如何思考异步系统。初学者需要消除的第一个大障碍是习惯性思维——异步设计方法不同于一般用时钟进行设计的方法。对现有的时钟系统，如果只是将时钟去除，而用异步握手电路简单地替代它，其结果注定是失败的。异步设计的另一个障碍是电路的设计方法——目前存在很多完全不同的设计方法。异步设计教程的主要目标是避开上述不同点，而主要介绍各种异步设计中一致且共同的内容。通过对这一部分的学习，读者应该能够理解异步系统的特性，从而能够跳出传统的时钟设计的思维框架，并且能够充分发挥无时钟系统的优势，创造性地开发出新的设计方案。

掌握了异步设计思想后，第二个障碍就是设计的效率问题。VLSI 设计通常有强大的软件工具支撑，从而能够得到高效的设计。异步设计在辅助设计工具方面相对落后，但最近这些年来，情况也正在改善。

第二部分主要介绍异步电路的一种高级综合系统——Balsa。该系统是由Doug Edwards（负责曼彻斯特大学该系统的开发）和 Andrew Bardsley（完成了系统中的大部分软件）编写的。Balsa 并不能解决所有异步设计中的问题，但它能够对一些非常复杂的系统进行综合（例如在第 15 章中所描述的一个用在DRACO芯片上的32通道DMA控制器），而且是掌握大型异步系统开发方法的良好途径。

在了解了如何分析异步设计并且有了一套设计工具之后，下一个问题就是：我们能够用已有的方法和工具来设计什么样的系统？对于这个问题，在第三部分中我们给出了一些复杂异步系统实例。这些例子都给出了具体的描述，使读者能够对设计过程有一个清晰的了解。这些例子包括飞利浦公司设计的商用智能卡芯片和曼彻斯特大学设计的Viterbi解码器。第三部分最后一章对高级异步微处理器进行了分析和讨论，这些讨论主要针对 Amulet 系列处理器（该系列处理器也是由曼彻斯特大学开发的）。

虽然本书主要是在很多作者所发表文章的基础上编写的，但文中的所有内容都基于以下一个原则——解决异步设计初学者可能会遇到的基本问题。为了保证本书的可读性以及篇幅适中，书中不得不有选择地忽略了一些内容。我们的目的主要是引导读者熟悉异步设计，如果读者想深入地学习，或者希望全面地展开我们所介绍的一些问题，那么本书中所列出的大量参考文献将是充足的信息来源。

Jens Sparsø 和 Steve Furber

致　谢

本书的编写得到了很多人的大力支持。除了本书部分章节的作者之外，还有一些论文的作者阅读了本书的部分初稿并且给予了很有价值的意见和建议，从而使本书的质量得到了明显的提高。

编著者亦感谢 Alan Williams, Russell Hobson 和 Steve Temple，他们认真阅读了书稿并且提出了许多建设性的意见。

本书的第一部分已经作为课程的教材。基于 2001 年春在 DTU 开设的“49425 Design of Asynchronous Circuits”课程的学生反馈情况，这部分内容得到了进一步的改进。

本书出现的任何错误或遗漏，责任都在于编者。

本书写作的初衷是为欧洲低功耗电子系统设计开端（ESD-LPD）的普及做出一些贡献，本书是该系列丛书中的一册。读者将会清楚地看到，本书的内容不是仅仅把 ESD-LPD 中的项目结果拿来公布。编者对异步电路设计工作组（ACiD-WG）的支持表示感谢，它提供了一个可以相互交流思想的论坛。ACiD-WG 从 1992 年由 European Commission 提供资助，完成了几个框架构建项目：FP3 基础研究（EP7225）、FP4 组件和子系统技术（EP219491）以及 FP5 微电子学（IST-1999-29119）。

序　言

本书是“新型低功耗设计结构、方法和设计实践”系列丛书的第三册。该丛书的编写源于自1997年开始的一个大型欧洲项目，这个项目的目标是通过进一步的研发以及更快、更广地在工业中采用高级设计方法，降低整个电子系统的功耗。

由于当前便携式信息和通信终端的广泛应用，从而要求一块小型电池能够使用很长时间，所以低功耗设计变得越来越重要了。除此之外，对于一些高性能的电子系统，由于时钟频率的增加，其芯片上每平方毫米的功耗也相应增加，从而可能造成诸如冷却和可靠性问题，或者出现其他限制系统性能的因素。

由欧盟信息技术项目“Esprit”所发起的“低功耗设计领航行动”最终包括了19个研发项目和1个配套项目，总预算达到1400万欧元。这次行动也被称为欧洲低功耗电子系统设计开端（ESD-LPD），预计在2002年完成。这次行动旨在研究和论证新的降低功耗的设计方法，配套项目用于确保这些方法、实验和结果的整理与出版。

本计划涉及低功耗的各种不同级别的设计：系统级和算法级、指令集处理器级、定制处理器级、寄存器传输级、门级、电路级和版图级，主要涵盖了数据单元、控制单元和异步结构等方面。其中有10个项目主要研究数字电路，7个项目研究模拟和混合信号电路，还有2个项目研究的是软件方面的相关内容。这些研究主要应用在通信、医疗器械和电子商务等设备上。

以下列出了对20个项目及其目标的描述，按预算资金的递减顺序排列。

CRAFT　基于无线应用的CMOS射频电路设计

- 高级CMOS RF电路设计，包括LNA、下变频混频器和移相器、振荡器和频率综合器、集成的Δ-Σ变换滤波器、功率放大器
- 基于硅样品可微调和验证的有源及无源器件新模型的开发
- 对复杂结构的分析和规范以满足低功耗单芯片实现

PAPRICA　功耗和部件数减少的新型通信架构

- 通过对核心模块的物理设计和描述可以容易地估计DQIF
- 基于标准CMOS数字工艺的低功耗RF设计技术
- RF设计工具和框架，PAPRICA设计包
- 一个具体应用实现的示例

MELOPAS　低功耗 ASIC 设计方法

- 开发一种用在设计前期评估复杂 ASIC 功耗的方法
- 开发一种硬件 / 软件协同仿真工具
- 快速实现电子设备功耗的显著降低

TARDIS　技术的协调和普及

- 组织设计试验之间的交流并开发它们之间可能的合作
- 指导从设计试验中获得方法和经验
- 组织并促进更广的普及，使用获得的设计技术及经验

LUCS　低功耗超声芯片组

- 低功耗 ADC、存储器和电路设计的设计方法
- 一种手持式医疗超声扫描仪的样品展示

ALPINS　通信系统的模拟低功耗设计

- 低电压音频平滑滤波器和 DECT 系统模拟前端电路中的模数 / 数模转换器
- 用于工作电压为 2.5 V 的 GSM 模拟接口电路的高线性度跨导电容（gm-C）滤波器
- 能运行于工业界合作者设计环境的形式化验证工具，这些工具支持从系统级到晶体管级的完整设计过程

SALOMON　针对低功耗的系统级模数折中分析

- 针对通信用混合信号 ASIC 的一种通用的自顶向下的设计流程
- 模拟和数字模块的高级模型以及对这些模块的功率估算
- 用样品来演示通用设计流程，该样品使用具有特定软件工具的设计流程来设计

DESCALE　针对低能量智能卡应用的设计实验

- 高度创新的握手技术应用
- 与同步运行方法相比较，目标是功率降低 3~5 倍，峰值电流降低 10 倍

SUPREGE　应用于短距离无线数据传输的微功率超再生收发器

- 与各种参数（功耗、面积、带宽、灵敏度等）密切相关的微功率接收器 / 发射器设计的折中及优化
- 调制 / 解调和数据传输系统的接口
- 基于超再生原理的集成微功率接收器 / 发送器的实现

PREST　针对系统技术的功耗削减

- 调查当前低功耗设计技术和商业化的功耗分析软件工具
- 通过功耗对比来调查设计结构和算法
- 调查异步设计技术和算法类型

- 构建一个低功耗设计流程
- 制造并描述一个 Viterbi 示例用于评估最优的功耗降低技术

DABLP　多处理器 DAB 应用的低功耗开发

- 一种低功耗 DAB 通道译码器架构
- 对 ATOMIUM 方法和支持工具的改进与扩展

COSAFE　针对安全攸关应用的低功耗软硬件协同设计

- 针对软硬件严格的安全性，开发其能量效率分配策略
- 设计并实现一个低功耗、高安全度ASIP，它用来实现一个便携式输液泵系统的控制单元

AMIED　异步低功耗方法和一个加密 / 解密系统的实现

- 能极大降低功耗的 IDEA 加密 / 解密方法的实现
- 重点在算法和结构优化上的高级低功耗设计流程
- 基于商业化工具的异步设计方法的产品示范

LPGD　一种低功耗设计方法 / 流程并应用于 DCS1800-GSM/DECT 调制 / 解调器的实现

- 完成一个自顶向下的低功耗 DSP 应用设计方法 / 流程的开发
- 一个使用 DCS1800-GSM/DECT 的集成 GFSK/GMSK 调制 / 解调器实现方法的示范

SOFLOPO　嵌入式应用低功耗软件开发

- 映射一个特殊算法代码到合适指令子集上的开发技术和指导原则
- 把这些技术集成在针对具有功耗意识 ARM-RISC 和 DSP 代码优化的软件中

I-MODE　多模式手提电话低功率 RF 基带接口

- 通过使用基于低功率技术的模拟基带低通滤波器和数据转换器来提高 COMS 工艺下 DECT/DCS1800 收发器的集成度

COOL-LOGOS　通过使用局部条件忽略和全局门尺寸调整技术降低功耗的实验评估

- 把开发出来的低功耗设计技术应用于一个制造好的 24 位 DSP 上
- 通过将新型的DSP预计的功耗减小值（在仿真中）与实测值相比较来评估新技术用于实验芯片的优势，评估商业化的影响

LOVO　低输出电压 DC/DC 转换器的低功耗应用

- 针对高级低功耗系统供电技术的开发
- 针对低输出电压功率变换器的同步整流新方法

PCBIT　便携式 PC 的低功耗 ISDN 接口

- 设计一个 PC 卡板来实现 PCBIT 接口

- 在单 ASIC 中集成第 1 层和第 2 层通信协议
- 在 ASIC 设计中加入功耗管理技术
  - 系统级：关闭电路中的空闲模块
  - 门级：预计算，门控时钟 FSM

COLOPODS　设计一个耳蜗助听低功率 DSP 系统

- 选择一个有前景的低功耗技术方向能够通过集成模拟模块来降低功耗
- 设计一个语音处理 IC，与 3.3 V 的设计相比较，可以使功率减少 90%

低功耗设计项目已经获得了以下成果：

- 已设计的原型芯片可以证明功耗降低 10% ~ 30%
- 已开发出新的低功耗设计库
- 已证明新的低功耗 RF 结构可用
- 已开发出新的更小和更轻的移动设备

与运行众多相互独立的Esprit项目不同，在低功耗设计项目中，指导工作已经将各个项目紧密结合在一起。这主要归功于本项目的协调者——DIMES的存在及其作用，DIMES-Delft微电子及亚微米技术研究所，位于荷兰的Delft（http://www.dimes.tudelft.nl），DIMES的任务是协调、促进和组织：

- 项目之间的信息交换
- 对方法和经验做系统化的归档
- 面向公众出版并进行普及

协调者所取得的最为重要的成绩是：

- 制定了新的人员联络方式，使合作者之间的协同工作得以更好、更快地发展。
- 组织了低功耗设计研讨会、专门的会议议程以及一个低功耗设计网站http://www.esdlpd.dimes.tudelft.nl，在这个站点上可以找到来自项目的所有公开报告及相关的各种信息。
- 设计方法学、设计方法与/或设计经验得到说明，并很好地归档与利用。基于项目工作及项目合作，计划出版一系列低功耗设计的书，由项目成员所著。这一系列关于低功耗设计的图书将向大众传达欧洲低功耗电子系统设计开端运行中获得的新的设计方法和设计经验。

最后，本项目组的主要贡献除了已提到的技术上的成就外，还包括加快了把关于低功耗设计方法的新知识引入主流开发过程的步伐。

感谢来自于不同公司和机构的所有项目合作者，低功耗的开端因为他们而获得了成功。

Rene van Leuken, Reinder Nouta, Alexander de Graaf, Delft

目　录

第一部分　异步电路设计教程

第二部分 Balsa ——异步硬件综合系统

第三部分　大规模异步电路设计

第一部分　异步电路设计教程

Jens Sparsϕ
Technical University of Denmark
jsp@imm.dtu.dk

摘要：异步电路与同步电路在电路特性上有很大不同，在阅读下面的章节后，读者将会更加清楚这些差异。我们可以利用异步电路的这些特性来设计电路，以获得更好的性能参数，如电路的功耗、性能以及电磁干扰（EMI）等。

异步设计是一种还没有被广泛接受与应用的设计方法。现有的一些教材综述性地介绍了异步电路的一些基本理论，但是由于这方面还不够成熟，所以还没有像大学传统教学课程那样传授异步电路设计的完整课程给电子工程和计算机工程专业的学生。

在理解异步设计的基本概念和能够设计出具有一定功能的复杂电路之间还存在一定的差距。本书的第一部分所提供的异步电路设计教程的目的就在于解决这个问题。

具体来说，第一部分的主要目的如下：

1. 给那些具有一定的同步数字电路设计背景的读者介绍一些异步电路设计的基本理论，使得这些读者能够更好地阅读和理解本书；
2. 使读者理解异步电路的一些特性，从而能够设计出一些具有优良性能参数的特殊电路。

本部分的素材建立在多个异步芯片设计经验的基础上，此外，近十年来一些欧洲会议和丹麦技术大学的一些课程所提供的文献，使得内容更加完备。1999年5月，我在Delft技术大学进行了为期一周的强化课程，当时相关的资料已经具备，所以我开始准备这部分内容的写作。大部分的素材是近来才采用的，并且2001年春季在丹麦技术大学的课程中得到了修正。作者又补充了一些期刊文献和一些设计项目，使得教材可以作为异步设计一学期的课程内容。

关键词：异步电路，教程

第1章 引 言

1.1 为什么要考虑异步电路

现今大部分电路的设计和制造采用的都是同步方式。实质上，这些电路是基于两个基本假设来简化它们的设计：

1. 电路中的所有信号都是二进制的；
2. 所有的元件都共享一个公共的离散时序，该时序由分布于整个电路中的时钟信号来定义。

异步电路和同步电路在本质上是不同的，虽然异步电路也采用二进制信号，但是其中没有公共离散的时序。异步电路中用握手（handshaking）替代公共时钟来实现它的各个部件之间的同步（synchronization）、通信（communication）以及运算的顺序。使用“同步电路的术语”来表达：异步电路类似于系统的精细度时钟门控和由实际电路的延迟决定的非同相局部时钟，从而使寄存器只在需要的位置和需要的时候触发。

与同步电路相比，这种内在的特性使得异步电路能够在以下几个方面体现出优势。对这些方面有兴趣的读者可以在文献[140]中找到更为详细的介绍。

- 低功耗[136, 138, 42, 45, 99, 102]
  ……精细度时钟门控和零备用功耗。
- 高速[156, 157, 88]
  ……运算速度由实际局部延时决定，而不是由全局最差（worst-case）延时决定。
- 低电磁噪声辐射[136, 109]
  ……局部时钟倾向于随机启动。
- 对于电源电压、温度以及制作过程中参数的变化具有鲁棒性[87, 98, 100]
  ……时序是基于匹配延迟的（并且对电路和导线延迟不敏感）。
- 更好的可重组性（composability）和模块化（modularity）[92, 80, 142, 128, 124]
  ……采用简单的握手接口和局部时钟。
- 没有时钟分配和时钟偏差（skew）问题
  ……因为没有全局时钟信号，所以不需要在整个电路中以最小相位偏差来分配时钟。

但异步电路同样也有一些缺点。电路中实现握手功能的异步控制逻辑单元常常会给芯片面积、电路速度以及功耗方面带来额外的开销。因此在设计时必须权衡采用异步技术能否获得更好的性能，例如，在采用异步技术后能否使系统在上述三个方面中的某个或全部方面带来实质性的性能改进。另一些障碍是缺乏CAD工具、设计策略以及相关的测试工具和测试向量生成工具。

对于异步设计的研究可以追溯到20世纪50年代[93, 92]。直到20世纪90年代，学术界和工业界才在一些重要的实际应用中采用异步技术，并展现出更好的性能，从而商业上开始逐步采用异步技术。在文献[106]和本书的第三部分中列举了一些最新的例子。

1.2 目的与背景

现在已有许多介绍异步设计的好文章和书籍[54, 33, 34, 35, 140, 69, 124]，并且还有一些相关的专著和教科书[106, 14, 25, 18, 95]。读者可能会问为什么还要写这样一本介绍异步设计的书呢？主要有以下几个原因。

- 从设计一些异步芯片[123, 103]和过去十多年为学生和设计师讲授异步设计过程中得到的经验告诉我们，要设计出有效的异步电路，仅仅掌握基本的设计原则和理论是不够的。上面提到的那些关于异步设计方法和理论的文章、书籍，与那些描述实际设计和当前研究的论文之间存在较大的差距。“要在游戏中获胜，仅仅依靠了解游戏规则是远远不够的”。要弥补这个差距，必须要有一定的经验以及对异步电路的本质有较好的理解。我与其他的一些研究者得到的经验是：“直接异步”往往会得到一个面积更大、速度更慢和功耗更大的电路。最关键的是通过采用异步技术去解决问题中应用的算法与结构。这意味着异步技术并不总是解决问题的最好方式，往往要根据具体情况来考虑。
- 另一方面，异步设计是一个相对年轻的学科。不同的研究者可能会提出不同的电路结构和设计方法，初看似乎不同——因为使用的术语不同，但是深入了解后会发现基本的准则和最终电路是相似的。
- 最后，以上所提到的大部分文章和书籍都是介绍性的，内容广泛，种类繁多。对于一个刚开始学习设计异步电路的新手来说，欣赏在这个领域中前人所积累的各种不同理论与方法的同时，要理清学习的思路并不容易。

与以上所介绍的多数文献相比，本教程的目的是：(1) 有选择地介绍异步设计方法和理论；(2) 强调各种不同方法之间的基本原则和相似性；(3) 进一步引导读者设计一些实用的电路。

1.3 时钟与握手

图1.1(a)所示为一个同步电路。为了阅读方便，图中只画出了一条流水线，但实际上是可以用它来表示任何同步电路的。当使用硬件描述语言和综合工具设计ASIC时，设计者的注意力往往集中在数据处理方面并且假设电路中已经存在一个全局时钟。例如，设计者可以把通过全局时钟锁存到寄存器R3的数据视为在前一个时钟锁存到寄存器R2的数据的函数CL3，其变量赋值表达式为R3 := CL3(R2)。图1.1(a)是具有全局时钟的顶层图。

当处于物理层的设计时，与上述方法有所不同。现今的ASIC采用的是时钟缓冲器结构，从而导致电路中出现了大量的时钟（也可能是门控时钟）信号，如图1.1(b)所示。众所周知，这需要通过CAD工具和工程学方面的考虑来设计时钟门控电路，并且尽可能地减小和控制不同时钟信号之间的偏移。电路的设计必须确保满足两个时序约束条件——时钟跳变沿的建立时间和保持时间——由于这两个时序约束主要是由导线延迟决定的，所以满足这两个条件并不是一件简单的事情。在当前商业CAD工具中所采用的先插入缓冲器再综合的过程也并非一定收敛，即使是收敛的，由于它依赖于延迟模型，所以会出现精度方面的问题。

异步设计是一种很好的替代方式，异步电路中相邻寄存器之间用一些握手形式来代替同步电路中的时钟信号，例如，图1.1(c)所示的是基于握手协议的简单请求－应答电路。在接下来的章节中，我们会介绍其他的握手协议和数据编码方式，但在此之前让我们首先对异步电路进行一个总体的认识，如图1.1(d)所示。

- 把图1.1(c)中连接寄存器的数据和握手信号视为“握手通道”或“链路”。
- 把寄存器中存储的数据视为具有数值的托肯（token），这个值可能会随着托肯流经组合电路而发生变化。
- 存在于寄存器之间的组合电路对于握手是透明的，组合电路从它的输入端口得到一个托肯，进行相应的运算，然后通过它的输出端口输出托肯（这与Petri网中的变迁有点类似，在6.2.1节中会进行介绍）。

从以上方式看，异步电路可以视为一个简单的静态数据流结构[36]。直观上讲，要使电路正确地运行，必须保证包含数据的数据托肯在电路中流动而不会消失，保证某个托肯不会覆盖另外的托肯，也不会凭空出现一个新托肯。遵守如下的简单规则就可以实现上述条件：

> 如果某个寄存器的后续寄存器已经将这个寄存器在前一时刻所存储的数据托肯输入并存储，则允许这个寄存器从它的前向寄存器中输入并存入一个新的数据托肯（前向寄存器和后续寄存器的状态分别由引入的请求和应答信号指示）。

按照这个规则，数据沿着电路中的路径从一个寄存器传送到下一个寄存器。在这个过程中，后续的寄存器常常拥有相同数值的多个备份，但原有数值备份后将被新的数值以严格的顺

序替换，链路中的握手周期总是包含一个数据托肯的转换，理解这个“托肯流游戏”对于设计有效的异步电路是至关重要的，我们将会在后面对这些问题做进一步的讨论，并进一步将托肯流应用于除流水线外的其他结构中。以上内容目的是为读者对异步设计与同步设计本质的不同之处提供直观的理解。

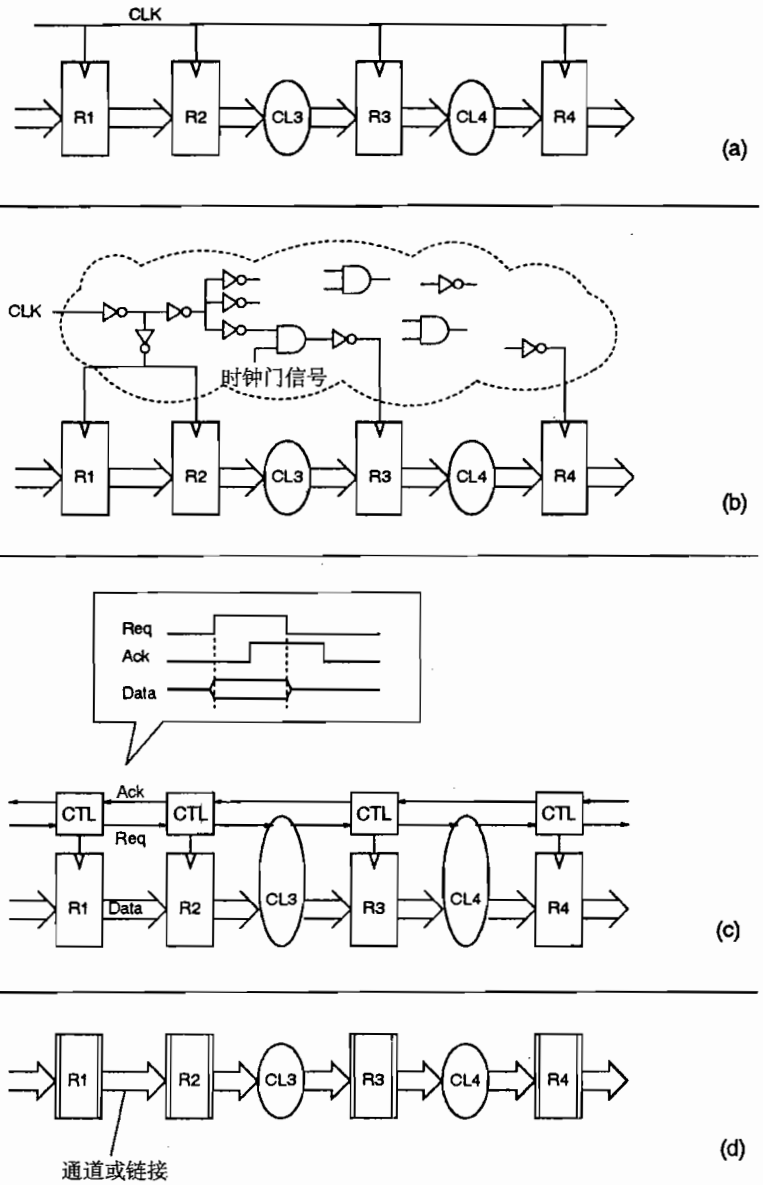


图 1.1 (a) 同步电路；(b) 带有时钟驱动和门控时钟的同步电路；(c) 等效的异步电路；(d) 异步电路抽象数据流概貌（图示为流水线，但可用来表示任何电路拓扑）

一个很重要的信息是:“握手通道和数据托肯观点”,这是一个非常有用的抽象概念,这个抽象概念的重要性就如同寄存器传输级(RTL)的概念在同步设计中的作用。数据流——下文采用这一说法,将电路的结构和功能从电路各部分的具体实现中分离出来了。

另一个重要信息是:在寄存器之间,握手电路控制托肯的流动,而组合电路块必须对于握手是完全透明的。确保这种透明性并不简单,不能再用传统的组合电路,我们将用“功能块”这一术语来表示输入和输出端口都是握手通道或链路的组合电路。

最后,书中还会涉及更多实际工程中的术语。图1.1(b)的同步电路是由一个周期时钟信号产生的同相时钟脉冲来控制的;而图1.1(c)中的异步电路是由局部时钟脉冲控制的,这个局部时钟脉冲可以在任意时刻产生;局部握手结构可以保证在任意需要的时刻和位置产生局部时钟脉冲。通过随机地产生时钟脉冲,可使电路的电磁辐射更低,并提供更为平稳的电流,而不会像在同步电路中那样出现很大峰值的 di/dt。

1.4 第一部分概要

第2章给出了许多基本的概念和电路,这些概念和电路对于理解后续内容很重要。读者可以对其大致地通读一遍,以后需要时再仔细研读某些部分的内容。

第3章和第4章描述数据流级的异步设计:第3章给出流水线(pipelines)和环(rings)的工作原理,并且介绍一组握手元件以及如何设计(大型)计算电路结构;第4章定性与定量地对上述电路结构进行性能分析和优化。

第5章重点解决第3章所介绍的握手元件的电路级实现问题;第6章着重于如何设计无冒险时序(控制)电路,主要包括对一些设计方法的概要介绍,并对一种特殊的方法进行比较深入的讨论,这种方法是:通过信号转换图(Signal Transition Graph, STG)规范来设计速度无关(speed-independent)控制电路。这些技术将通过具体的控制电路来阐述,这些控制电路用来实现在第3章中所介绍的几种握手元件。

第2章至第6章旨在以适当的深度介绍异步电路的一些基本技术和设计方法。第一部分最后两章的内容相对较少,第7章介绍一些高级主题,这些主题与4相捆绑数据协议(4-phase bundled-data protocol)的电路实现相关;第8章介绍异步设计的硬件描述语言和综合工具。第8章并不是对这方面内容的完整描述,主要集中于类CSP语言和面向语法的(syntax-directed)编译,当然也讲解如何通过VHDL标准语言来设计异步电路。

第2章 基础知识

本章将对许多主题和概念进行阐释，这些主题和概念是理解后续章节内容和分析不同异步设计风格异同点的基础。为了给读者以直观的印象，本章的写作风格与其他部分有些不同。

2.1 握手协议

第1章中介绍过一个特殊的握手协议——归零（return-to-zero）握手协议，如图1.1(c)所示。在异步设计中，它还有一个包含更多信息的名字——4相捆绑数据协议（4-phase bundled-data protocol）。

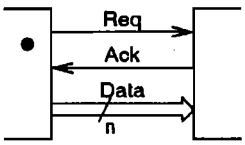
2.1.1 捆绑数据协议

“捆绑数据”是指，数据信号采用布尔型数值，请求（request）线和应答(acknowledge)线相互分开并且与数据信号线捆绑在一起，如图2.1(a)所示。在图2.1(b)所示的4相协议中，请求和应答线也是采用布尔型数值，“4相”表示的是通信活动的数量：

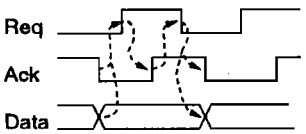
1. 发送端发出数据并且将请求信号置为高电平；
2. 接收端接收数据并同时将应答信号置为高电平；
3. 发送端响应接收端并将请求信号置为低电平（此时数据不再保证有效）；
4. 接收端在请求信号转为低电平后也将应答信号置为低电平。

完成以上4个步骤后，发送端可以继续启动下一个通信周期。

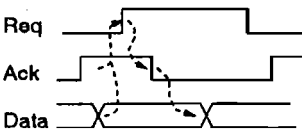
大部分的数字设计者应该都比较熟悉4相捆绑数据协议，但这个协议也存在缺陷：由于它有多余的“归零动作”，会增加电路在速度和功耗上的开销。图2.1(c)所示的2相捆绑数据协议能够避免这种缺陷。在这个协议中，请求线和应答线上的信息用信号跳变来表示，此时无论是从0→1还是1→0的跳变都表示相同的信息，它们表示的都是一个“信号事件”。从理论上讲，使用2相捆绑数据协议电路要比采用4相捆绑数据协议的电路速度快，但通常前者的事件响应电路实现起来较为复杂，所以到底使用哪种协议往往要根据实际情况来决定。



(a) 数据捆绑（推）通道



(b) 4 相协议



(c) 2 相协议

图 2.1 (a) 捆绑数据通道；(b) 4 相协议；(c) 2 相协议

接下来我们对一些术语进行说明。“捆绑数据”一词会贯穿于本书的始终。有些书中使用“单轨”（single-rail）这个名词来代替“捆绑数据”，“捆绑数据”突出的是数据信号与握手信号之间的时序关系，而“单轨”表示的是用一条线来表示一位信息。当然，一般认为“单轨”是相对应于本书后续章节中讨论的“双轨”（dual-rail）而言。除了 4 相握手（或 4 相信号）的名称外，在有些书中称为归零（Return-To-Zero, RTZ）信号或电平信号；同样地，2 相握手也常称为非归零（Non-Return-to-Zero, NRZ）信号或跳变信号。因此，归零单轨协议与 4 相捆绑数据协议的意思是相同的。

以上所描述的协议都默认发送端是主动方（active party），主动方开启通道的数据传输，此时这种通道称为“推通道”（push channel）。与之相反，当接收端请求接收新数据时，这种通道称为“拉通道”（pull channel）。后一种情况中，请求信号与应答信号的方向与前一种情况是相反的，并且指示数据有效性的应答信号是从发送端传输至接收端的。在抽象的电路图中，我们将会在通道的主动方处用一个圆点来做标记，如图 2.1(a)所示。

为了使描述更完整，有必要对以下一些变化形式进行介绍：(1) 不带数据的通道可以用做同步化（synchronization）；(2) 通道中的数据在两个方向都可以传输，req 与 ack 用于表明传输数据的有效性。第二种变化形式可用于与 ROM 的接口：地址将与 req 捆绑在一起，同时数据与 ack 捆绑在一起。我们将会在 7.1.1 节中详述这两种通道。在以后的章节中，我们只限于对推通道的讨论。

所有捆绑数据协议都依赖于延迟匹配（delay matching），因此发送端和接收端中信号变化的时序是一致的。在推通道中，发送端的数据在请求信号变为高电平之前到来（有效），公式表示形式为 Valid(Data) $\prec$ Req。在接收端这个时序也没有变化，这种电路的物理实现要求一些特殊细节。这些细节的实现方案包括：

- 控制布局布线过程，尽可能把所有的信号作为一组放置于同一通道中，这在平铺式（tile-based）数据通道结构中是不难做到的。

- 在发送端留有一定的时间裕量。
- 在布局后插入或调整缓冲器（现有的综合和布局CAD工具就是这样做的）。

另外一种选择是选用那些对导线延迟鲁棒性更强、更经典的协议。在接下来的章节中，我们会介绍一些对导线延迟完全不敏感的协议。

2.1.2 4相双轨协议

如图2.2所示，4相双轨协议通过使用两条线表示一个信息位的方式将请求信号与数据信号放在一起编码，形成用于通信的信号。实质上，它是用两条导线来表示一位信息d的4相协议；一条导线d.t用来表示逻辑1信号（或真值），而另一条导线d.f用来表示逻辑0信号（或假值）。当我们观察一个1位通道时，将会看到一个4相握手的完整的过程，其中“请求”信号可能为d.t或d.f。4相双轨协议具有很强的鲁棒性，通信双方可以很稳定地进行通信而不受导线延迟的影响——该协议是延迟不敏感（delay-insensitive）的。

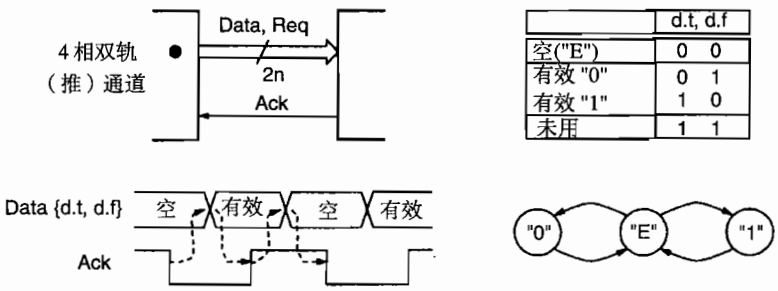


图2.2 使用4相双轨协议的延迟不敏感通道

现在把双线{x.f, x.t}看作一个码字。{x.f, x.t} = {1, 0}和{x.f, x.t} = {0, 1}表示“有效数据”（它们分别表示的是逻辑0和逻辑1）；{x.f, x.t} = {0, 0}表示“无数据”（也称“空白”、“空值”或“NULL”）。码字{x.f, x.t} = {1, 1}在协议中不使用。协议不允许从当前有效码字直接跳变到另一个有效码字，而必须经过一个中间步骤，如图2.2所示。

4相协议的握手过程概括如下：(1) 发送方发出一个有效码字，(2) 接收方收到码字并把应答信号置为高电平，(3) 发送方响应应答信号同时发出空值码字，(4) 接收方把应答信号置为低电平。完成以上4个步骤后，发送方可以开启新的通信周期。也就是说，4相协议可以视为通道内有效码字被空值码字所隔开的数据流。

可以将这个协议扩展到多位并行通道中去。N位数据通道可以由N对导线简单并置而成，每对导线按照上述方式进行编码。当所有位都处于有效状态时，接收方把应答信号置为高电平；而当所有位都处于空值状态时，接收方把应答信号置为低电平。这些结论都很直观，其实，这种编码具有一定的数学背景——双轨码属于延迟不敏感码类中的一种简单编码方式[147]，它具有以下一些优点：

- 任何由双轨码字并置得到的码字也是一个双轨码字。
- 对于给定的 N（N 表示通信的位数），所有可能的码字的集合可无交集地分成 3 组：
  - 空值码字，所有的 N 对线都为{0, 0}；
  - 中间码字，一些线处于空值状态而其他线处于有效状态；
  - 2^N 个不同的有效码字。

N 位通道的握手示意图如图 2.3 所示。接收端首先得到空值码字，接着是一系列中间码字（随着时间的推移，越来越多的“位/线对”变为有效状态）并且最终成为完整的有效码字。当接收和应答完毕后，接收端又接收到一系列中间码字（随着时间的推移，越来越多的位变为空值），最终又变为空值码字，于是接收端把应答信号置为低电平。

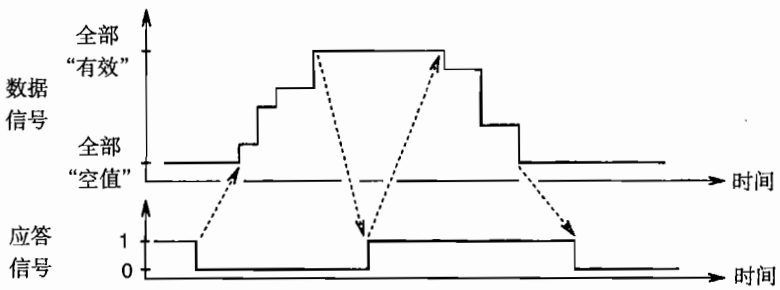


图 2.3　4 相双轨通道握手示意图

2.1.3　2 相双轨协议

2 相双轨协议也是用两条线{d.t, d.f}来表示一位数据的，但如 2.1.1 节所述，此时信息用跳变（事件）的形式表示。N 位通道的 N 对线之中，每组有且只有一条线发生跳变，则表示接收到一个新码字。这种协议不存在空值，一条有效消息被应答后，紧接着另一条有效消息被应答。图 2.4 表示采用 2 相双轨协议二位通道的通信时序图。

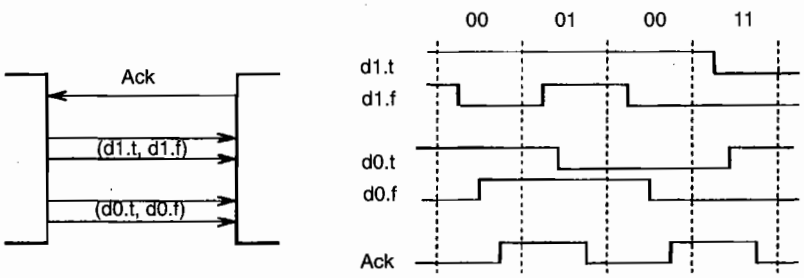


图 2.4　2 相双轨通道握手示意图

2.1.4 其他协议

以上介绍了4种最常用的通道协议：4相捆绑数据推通道、2相捆绑数据推通道、4相双轨推通道和2相双轨推通道。除了这4种常用的协议外，还有其他的一些协议。在双轨协议中使用两条线表示一位数据也可以看作是对一位数据进行独热（one-hot）编码，在一些控制逻辑和更高基数数据编码中，常常将编码扩展到1-of-n独热编码形式。如果关注通信而不是计算时，也可以使用m-of-n编码形式。协议的解空间可以由一些选项的叉乘来表示：

$$\{2\text{相}，4\text{相}\}\times\{\text{捆绑数据}，\text{双轨}，1\text{-of-}n，\cdots\}\times\{\text{推}，\text{拉}\}$$

协议的选择将会直接影响电路的特性（面积、速度、功耗、鲁棒性等）。在我们对电路实现的相关问题进行讨论前有必要介绍一下“指示”（indication）或“确认”（acknowledgement）的概念，同时还将介绍一个新元件——Muller C单元。

2.2 Muller C单元和指示原理

在同步电路中，时钟的作用是指示信号稳定和有效的时间点。在时钟节拍之间，电路中的某些信号可能会像组合电路一样出现冒险，或者出现多次跳变的情况，但这些现象并不会影响电路的功能。不过在异步（控制）电路中情况却不同。由于异步电路中没有时钟，因此在很多情况下要求信号在所有时刻都必须有效，从而每一次信号跳变都会对功能产生影响，所以在电路中必须要避免冒险和竞争。

直观地讲，电路可以视为由各种门电路组成（通常还包括一些反馈），所以当某个门的输出发生变化时，与这个门相连的某些门的输出可能会发生改变。图2.5描述的是图1.1(c)中CTL电路的一种实现形式。在这里对它的功能不展开解释，而是使读者对于正在讨论的这种类型的电路有一个总体的印象。显然，如果这个电路用于图1.1(c)所示的流水线结构中，一旦在Ro, Ai, Lt信号中出现了冒险，将会造成致命的错误。

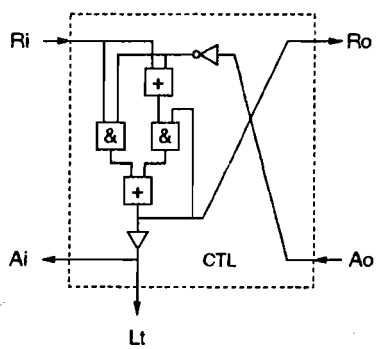
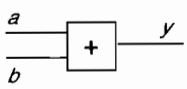


图2.5 异步控制电路实例。Lt是控制存取的局部时钟

指示或确认的概念对于设计CTL电路是很重要的。图2.6所示为简单2输入或门。当门的输出从1变到0时，可以得出此时或门的两个输入都为0。然而，当门输出从0变化到1时，可以确定至少有一个输入为1，但不能确定是哪个输入端。因此我们可以说或门只能指示或确认所有输入都为0的情况。同样，与门只能指示所有输入都为1的情况。

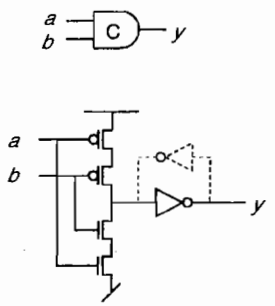


a	b	y
0	0	0
0	1	1
1	0	1
1	1	1

图2.6 或门及其真值表

如果某个信号的跳变不能指示或确认其他信号的跳变情况，则可能出现冒险现象，在电路设计中必须避免，详细内容将在2.5.1节以及第6章中介绍。

图2.7所示的“Muller C单元”在这一方面是一个较好的元件。Muller C单元是一种状态保持元件，类似于异步置位复位锁存器（set-reset latch）。当它的所有输入都为0时输出为0，而当所有的输入为1时输出也为1，而对于其他的输入组合，输出状态保持不变。因此，当输出从0变化到1时我们可以判断此时输入都为1；同样，当输出从1变化到0时可以判断此时的输入都为0。



说明：

1: if $a = b$ then $y := a$

2: $a = b \mapsto y := a$

3: $y = ab + y(a + b)$

4:

a	b	y
0	0	0
0	1	不变
1	0	不变
1	1	1

图2.7 Muller单元：符号、可能的实现方式及输入输出说明

综上所述，我们可以知道几乎所有采用握手形式的异步电路都会包含0和1之间的周期性跳变。而Muller C单元是一个在异步电路中使用得比较频繁的基本元件。

2.3 Muller流水线

图2.8所示是一个由C单元和反相器组成的电路。这个电路就是所谓的Muller流水线或Muller分配器。该电路的一些演变和扩展组成了绝大多数异步电路控制部分的主体。有的电路

看起来Muller流水线的特征并不明显，但若去除电路中一些琐碎的细节，则可看出Muller流水线在异步电路中所起到的至关重要的作用。这个电路具有对称性，一旦掌握了它的行为，就为理解大多数异步电路打下良好基础。

图2.8所示的Muller流水线是传递握手协议的元件。当所有的C单元都已初始化为0后从左侧开始启动握手。为了更好地理解它的原理，让我们考虑第i个C单元——C[i]的运行情况：如果后向C单元C[$i+1$]输出为“0”，则C[i]传播（输入和存储）它的前向C单元C[$i-1$]输出的“1”；同理，若C[$i+1$]输出为“1”，则C[i]传播（输入和存储）C[$i-1$]输出的“0”。我们可以把异步电路中信号传播想像成波的传播，如图2.8中的最后一图所示。从这一观点出发，C单元段在流水线中的作用是：以精确的控制方式来传递波峰和波谷以保持波的完整性。

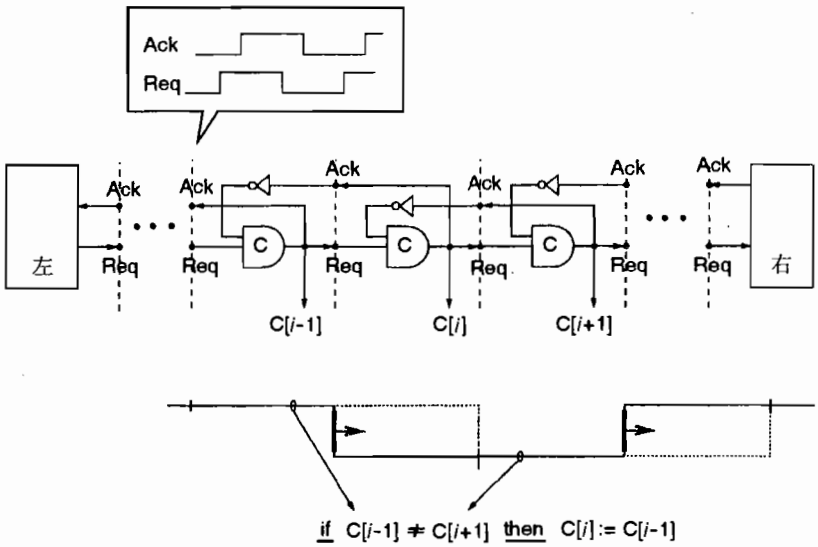


图2.8 Muller流水线（Muller分配器）

在C单元流水线级间的接口部分都可以看到正确的握手过程，但是此时的时序可能会与左边的时序不同；一旦某个波形进入了Muller流水线，则它的传播速度就由电路的实际延迟所决定。

第一个握手信号从左边时序环境进入，最终会到达右边时序环境。如果右边环境没有应答这个握手信号，则流水线最终处于满状态。处于满状态时，流水线将会终止与左边环境进行握手——Muller流水线的行为与FIFO传输相类似。

除了这个良好特性外，流水线还具有对称性。首先，无论采用的是2相还是4相握手，都可以用相同的电路来实现，差别在于你如何解释信号与使用的电路。其次，电路也可以从右向左运行。只要将信号极性的定义反过来，将请求和应答信号的角色换过来，电路就能够从右向

左运行。这一点与半导体中的电子和空穴的概念类似，电流的产生可能是由电子沿某个方向移动而形成的，也可能是由空穴沿相反方向移动而形成。

最后可以得出一个令人感兴趣的特性：电路的正确运行跟门和导线的延迟无关——Muller流水线是延迟不敏感的。

2.4 电路实现方式

如前所述，握手协议的选择将直接影响到电路的性能（面积、速度、功耗和鲁棒性等）。下面是大部分实际电路采用的协议：

4相捆绑数据——这个协议与同步电路最相似。由于广泛使用时序假设，通常可以使电路获得最好的性能。

2相捆绑数据——由Ivan Sutherland在其1988年的图灵奖获奖演讲中以“Micropipelines”名称首次提出。

4相双轨——源于David Muller早期（20世纪50年代）工作成果的经典方法。

以上所有这些协议的共同点是：这些协议都是采用某种Muller流水线来控制存储元件的。下面我们将解释流水线的基本原理，它是用锁存器作为存储元件而实现的。我们将在以后的章节中介绍电路的优化以及更为复杂的电路结构。

2.4.1 4相捆绑数据

4相捆绑数据流水线特别简单。Muller流水线用于产生局部时钟脉冲。产生在某级的时钟脉冲与相邻级的时钟脉冲以互锁的形式交迭。图2.9(a)表示的是一个FIFO，也就是不包含数据处理部分的流水线结构；图2.9(b)显示了如何把组合电路（也称为功能块）加在锁存器之间。此时，为了确保电路的行为正确，必须在请求信号路径中加入延迟匹配环节。

以上所介绍的电路，我们可将其视为传统的“同步”数据通道，它由锁存器和组合电路构成，电路由分布的选通时钟来驱动。也可以认为它们是一个异步数据流结构，由锁存器和功能块两种握手元件构成，如图中的虚线框所示。

图2.9所示的流水线实现电路虽然很简单，但也有一些缺点：当C单元的状态分别是(0, 1, 0, 1, …)时，只有每隔一个锁存器才锁存数据，这虽比采用主从触发器的同步电路好些，但可以设计出更好的流水线和FIFO电路。另外一个不利的因素是速度。一个流水线或FIFO的吞吐量是由电路完成一个握手周期的时间来决定的，对于以上实现的电路，涉及到相邻单元间的通信。在第7章将介绍速度更快、利用率更高的实现方式。

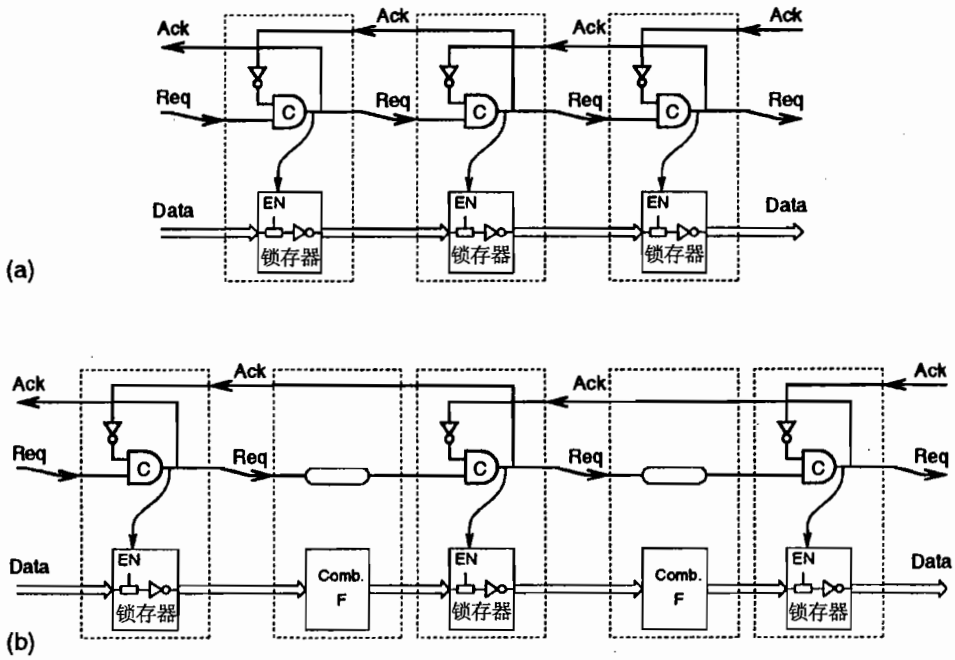


图 2.9 简单 4 相捆绑数据流水线

2.4.2 2 相捆绑数据（微流水线）

2相捆绑数据流水线同样是把Muller流水线作为控制电路的主体，但此时控制信号被解释为事件或跳变，如图 2.10 所示。正是由于这个原因，在电路中必须要引入特殊的捕获 - 通过（capture-pass）锁存器：事件交替出现在C输入端和P输入端，引起锁存器在捕获模式与通过模式之间的交替切换。这就需要设计如图2.11所示的特殊的锁存器，后面将对这种锁存器做详细介绍。图 2.11 中的开关符号是用一个多路选择器（multiplexer）来表示的，图中的事件控制锁存器（也可以认为是边沿触发器），是由两个交替运行的电平敏感锁存器、一个多路选择器和一个缓冲器组成。

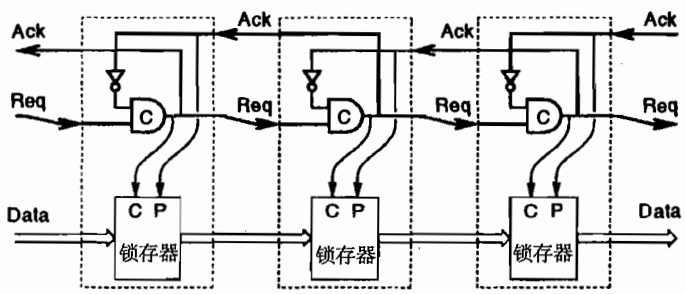


图 2.10 简单的 2 相捆绑数据流水线

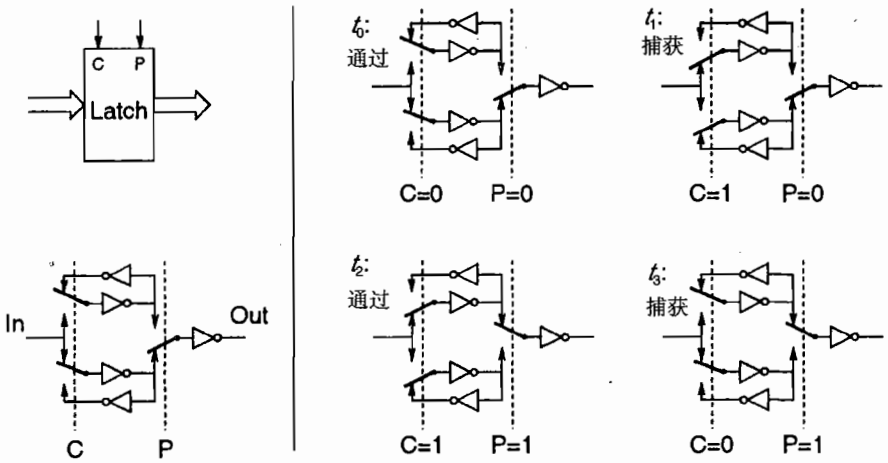


图 2.11 “捕获－通过”事件控制锁存器的实现和工作原理。在 t_0 时刻，锁存器是透明的（处于通过模式），C和P都处于低电平，C输入端的事件使锁存器进入捕获模式

图2.10所示为一个不含有数据处理部分的流水线电路，可以把匹配延迟元件和组合电路插入到锁存器之间，插入的方式与图 2.9 所示的 4 相捆绑数据的方式类似。

2 相捆绑数据方法在 20 世纪 80 年代末由 Ivan Sutherland 首先提出，并在其 1988 年的图灵奖获奖演讲中做了很好的介绍[128]。“微流水线”常常作为 2 相捆绑数据协议的同义词使用，但它还涉及到一些基于事件信号的特殊元件的使用。除了图2.11中所示的锁存器外，这些特殊元件还包括：与门、或门、选择器、开关、调用器（Call）和仲裁器。图 2.10、图2.11 和文献[128]中的图 15、图 12类似，但图 2.10、图 2.11 着重强调控制结构是Muller流水线。在文献[128]中还提出了一些值得注意的更小、更慢的锁存器的设计。

从概念上来讲，2 相捆绑数据方法相对于 4 相捆绑数据方法来说更简洁、更有效，因为在4相捆绑数据中的握手信号的归零部分会影响电路的性能和功耗。然而，从锁存器设计的角度来看，响应信号跳变的元件要比响应信号电平的元件更为复杂。除了存储元件外，响应信号跳变的条件控制逻辑电路实现起来也比较复杂。这已被作者[123]、曼彻斯特大学[42, 45]以及许多学者的经验所证实。

现在可以得出这样一个结论：对一个无条件数据流，以及要求高速的系统设计时，2 相捆绑数据协议是一个更好的选择。但是就像前面所提到的那样，高速意味着必须在其他一些方面做出牺牲：更大的面积和更高的功耗。在这一点上，异步电路设计与同步电路设计是一样的。

2.4.3 4 相双轨

4 相双轨流水线同样也是基于Muller流水线的，但是它更为复杂：对数据和请求信号进行组合编码。图 2.12 表示的是一个 1 位 3 级无数据处理的流水线电路。它可以理解为由两个并行

的Muller流水线组成，并且在每一级上都有一个相应的应答信号来完成同步化操作。当每个流水线级中的一对C单元存储空值码字{d.t, d.f} = {0, 0}时，此级输出的应答信号为逻辑0；当存储两个有效码字（{0, 1}和{1, 0}）中的任意一个时，此级输出的应答信号为逻辑1。从2.2节可知，码字{1, 1}是非法的且不会出现，因此由或门产生的应答信号能够正确地指示流水线级的状态是“有效”还是“空值”。

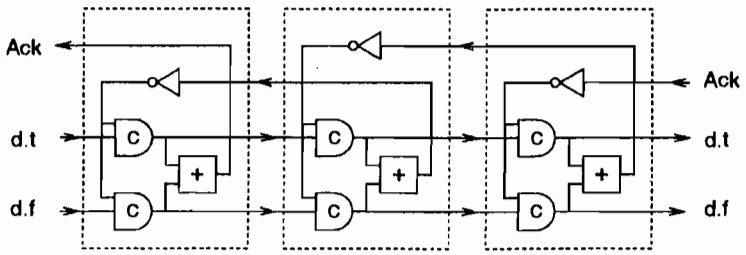


图2.12 简单的3级1位4相双轨流水线

一个N位的流水线可以用一系列1位流水线并行组成。由于不能保证所有位的数据在同一时刻到达接收端，所以需要功能块来完成同步化工作。在文献[124, 125]中我们描述了采用DIMS组合电路（后面将会介绍）来设计这种类型的电路。

如果需要对位并行（bit-parallel）进行同步化，则可以通过一个C单元把各个位应答信号组合成一个全局应答信号。图2.13给出一个位宽N的锁存器。虚线框中的或门和C单元构成一个“完成探测器”（completion detector），完成探测器用来指示存储在锁存器中的N位双轨码字是有效值还是空值。图中还给出了如何用2输入C单元来实现完成探测器。

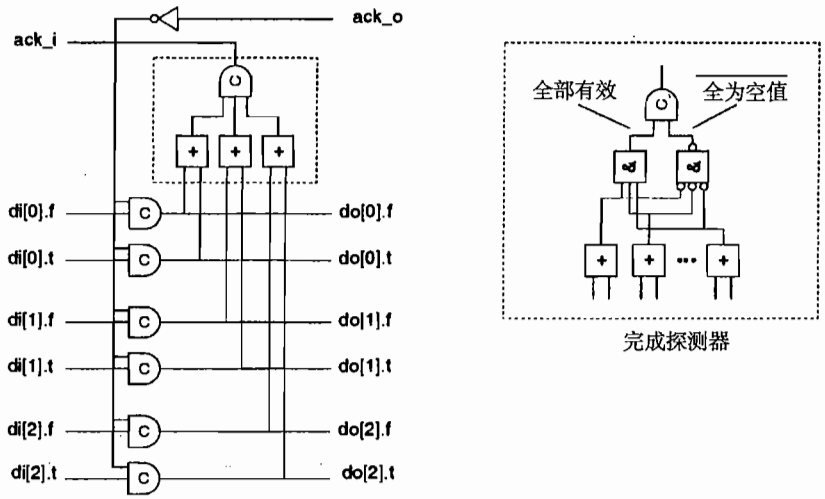
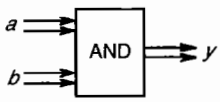


图2.13 带有“完成探测器”的N位锁存器

现在让我们看看在4相双轨电路中的组合电路是如何实现的。在第1章中已经提到锁存器之间的组合电路必须对握手信号是透明的。因此，一个组合电路的所有输出只有在它的所有输入都变为有效值之后才能变为有效值。否则接收锁存器可能会在发送锁存器发出的信号都变为有效值之前，就已经把应答信号置为高电平。同理，一个组合电路的所有输出只有在它的所有输入都变为空值之后才能变为空值。否则接收锁存器可能会在发送锁存器发出的信号都变为空值之前，就已经把应答信号置为低电平。因此，使用4相双轨方法中的组合电路中需要引入状态保持元件，并且电路在由“空值到有效”（empty-to-valid）或由“有效到空值”（valid-to-empty）的跳变过程中呈现出滞后特性。

图2.14是一个双轨与门的简单实现图，只用了C单元和或门。这个电路可以理解为直接把两线输出的最小项和的表达式映射到硬件中。当所有输入都有效时，4个C单元中的一个被置成高电平。这又引起与其相连的输出线变为高电平（相当于门电路产生了期望的有效输出）。如果所有的输入都变为空值，则所有的C单元也被置为低电平，从而双轨与门的输出也为空值。可以看到C单元的两个作用是：提供必要的“与”运算以及为“空值到有效”或“有效到空值”的跳变过程提供迟滞，这个迟滞对握手的可见性至关重要。还必须意识到，或门的输入信号不允许同时有多个信号变为高电平。



a	b	y.f	y.t
E	E	0	0
		无变化	
F	F	1	0
F	T	1	0
T	F	1	0
T	T	0	1

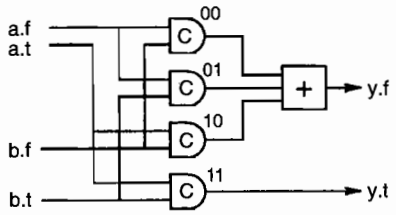


图2.14 4相双轨与门：符号、真值表及实现

其他的双轨门，如或门和异或门（EXOR）等，都可以用这种方式来实现，而双轨反相器的实现只需要交换真值线（x.t）和假值线（x.f）。这些基本双轨门实现时所用晶体管的数量相当大，在第5章中我们将会探讨更有效的电路实现方式，这里的目的在于讲述基本原理。

通过采用普通组合电路的综合技术，我们可以用基本的双轨门来构造任意布尔表达式的双轨组合电路。这些双轨组合电路，仍然保留基本双轨门对握手的可见性。

以上所阐述的基本观点都可追溯到David Muller在20世纪50年代末和60年代初期的工作[93, 92]。而文献[93]发展了速度无关（speed-independent）电路设计的基本理论，文献[92]是一

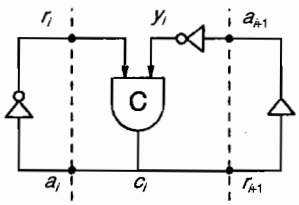
个更为贴近实际应用的介绍，其中包括了一个设计实例——采用前面介绍的锁存器和门电路来实现位串行乘法器。

2.5 理论

根据不同延迟假设，异步电路可以分为：自同步（self-timed）电路，速度无关电路（speed-independent）也称为延迟不敏感电路（delay-insensitive）。在这一节中我们将介绍这种分类的几个重要理论概念，目的是让读者了解其基本思想并对问题与解决办法有直观的印象，对理论感兴趣的读者可参考相关文献。最近的发展可参考文献[95, 54, 69, 35, 18]。

2.5.1 速度无关电路基础

首先回顾David Muller电路模型以及速度无关电路所需具备的条件[93]。电路及其（虚拟的）环境建模为封闭的门网络，此处封闭的意思是指所有的输入都与输出相连，反之亦然。门的模型可以为具有任意非零（non-zero）延迟的布尔运算，且导线是理想无延迟的。这种情况下，电路被描述为并行的布尔函数的集合，其中每个门的输出对应一个函数。电路的状态就是所有门输出的集合。图2.15说明了这样一个例子：用带有反相器和缓冲器的一个Muller流水线级来模拟左右两边环境之间的握手行为。



$$
\begin{aligned}
r_i' &= not(c_i) \\
c_i' &= r_i y_i + (r_i + y_i) c_i \\
y_i' &= not(a_{i+1}) \\
a_{i+1}' &= c_i
\end{aligned}
$$

图2.15 带有“虚拟”环境的Muller流水线中的Muller单元的行为

当门的输出与它的输入符合逻辑关系时，可以认为这个门是稳定的（stable），它的“下一个输出”和“当前输出”是相同的，即 $z_i' = z_i$。当门的输入发生变化时其输出也相应的发生变化，则称这个门处于受激状态，它的“下一个输出”和“当前输出”是不同的，即 $z_i' \neq z_i$。在经过一段不确定的延迟后，处于受激状态的门会自发地改变它的输出并且最终达到稳定状态，我们称该门被激发了。同样，受激发的门稳定在新的输出状态，其他的门依次受激发，等等。

为了更好地说明这一点，在此引用图2.15所示的例子，假设此时电路处于$(r_i, y_i, c_i, a_{i+1}) = (0, 1, 0, 0)$状态。此时 r_i 受到左边的环境激发，将把请求信号置为高电平。r_i 被激发后，电路的状态变为$(r_i, y_i, c_i, a_{i+1}) = (1, 1, 0, 0)$，此时 c_i 处于受激状态。为了对电路更好地进行综合和分

析，可以构建状态图来描述门激发的所有过程，详细的内容将在第6章中介绍，这里只对基本的概念做一些解释。

对于一个给定的状态，电路中可能会有多个门同时处于受激状态。如果这些门中的一个，如门 z_i，恰好是其他受激门的输入，则此时 z_i 的激发对于这些门会出现什么影响？这些门仍然保持受激状态；或者它们将会获得另外的输入信号而不等待门 z_i 输出的变化。速度无关电路不会出现后面这种情况。一个受激门没有激发就已经稳定，意味着潜在的冒险。由于延迟是未知的，所以门可能改变它的输出，也可能不改变它的输出。还可能是当“相反命令”到来要求门的输出保持不变时，这个门还处于实现上述动作的中间过程。

由于电路模型为每个门引入一个布尔状态变量（对于延迟不敏感电路还必须为每个导线段引入布尔状态变量），所以即使简单电路的状态空间也会非常大。在第6章中我们将会介绍能直接用于综合的信号转移图，它的表达形式更抽象。

现在已有了用来描述门级电路行为的模型，接下来将讨论异步电路的分类。

2.5.2 异步电路的分类

在门级电路层次上，根据电路所做的延迟假设，异步电路可以分为自同步（self- timed）、速度无关或者延迟不敏感电路。图2.16所描述的电路是为下面的讨论服务的。电路由A, B和C三个门组成，门A的输出连到门B和门C的输入。

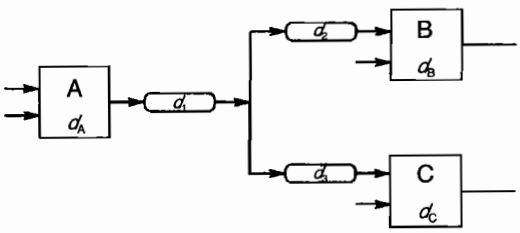


图2.16 带有门延时、传输线延时的电路模块

如前所述，对速度无关（Speed-Independent，SI）电路的延迟假设是：门电路具有有界但大小未知的正延迟，而导线延迟是零（理想状态）。参考图2.16，对于一个SI电路，意味着门延迟 d_A, d_B 和 d_C 为任意值，而导线延迟 $d_1 = d_2 = d_3 = 0$。考虑到在当今的半导体制造工艺中，实现导线的零延迟是很不现实的，因此对于这一电路，我们可以假设 d_1, d_2 和 d_3 都有一定的延迟并且 $d_2 = d_3$，从而可以将导线上的延迟加到门的延迟中去。从理论角度而言，这个电路仍是一个SI电路。

除了门是有界未知延迟外，导线也是有界未知延迟的，此时这个电路通常被称为延迟不敏感电路（Delay-Insensitive，DI）。参考图2.16，它作为延迟不敏感电路，意味着此时 d_A, d_B, d_C,

d_1, d_2, d_3都为任意值。显然，这种电路的鲁棒性非常强。判断电路为延迟不敏感电路的一种方法是：建立电路的Muller模型，其中（分叉后的）导线用缓冲器来建模，如果等效电路模型是速度无关的，则说明电路是延迟不敏感的。

然而在实际中，延迟不敏感电路是相当少的。只有那些由C单元和反相器组成的电路才是延迟不敏感的[82]，图2.5、图2.8和图2.15中所示的Muller流水线是延迟不敏感电路的重要形式。如果某个电路除了一些具有相同延迟的导线分支（$d_2 = d_3$）外是延迟不敏感电路，则这个电路称为准延迟不敏感电路（Quasi-Delay-Insensitive, QDI）。信号跳变能够同时到达分支终端的导线分支称为等时线（isochronic），等时线将在下节中详细讨论。典型的等时分支一般出现在基本构造块的门级实现中，此时设计者能够控制导线延迟。在抽象度更高的层次中，构成电路的模块几乎都是延迟不敏感的。经过以上讨论，我们应该对DI, QDI和SI的区别比较清楚了。

属于延迟不敏感的电路很少，所以我们几乎不考虑这种电路的计算，文献中涉及的大多数所谓的延迟不敏感电路实际上只是准延迟不敏感电路。

最后以一句话来介绍自同步（self-timed）电路：根据门和线无界延迟模型，我们推出了速度无关和延迟不敏感电路的特性。而那些依赖于更为精细及工程化的时序假设才能正确运行的电路则简称为自同步电路。

2.5.3 等时分支

从以上分析可知，速度无关电路与延迟不敏感电路的区别主要在于导线的分支，更具体地讲，就是到分支导线的各个终端的延迟是否相同。如果延迟是相同的，则导线分支称为等时分支。

电路中对等时分支的需求与2.2节中介绍的指示概念有关。考虑图2.16中的门A的输出发生了变化，这个变化最终会传递到门B与门C的输入端，经过一定的时间后门B与门C响应这个变化并输出新的结果。此时我们就可以说门A输出的变化可以通过门B与门C的输出来指示。如果只有门B有新的输出，此时我们不能确定门C是否得到了输入的变化。这种情况下，如果加上$d_2 = d_3$（也就是分支是等时的）这个假设，则门C也得到了输入的变化，从而得出输入信号的变化可以由门B的输出来指示。

2.5.4 不同电路之间的关系

在2相和4相捆绑数据方法中的控制电路通常都是速度无关电路（在某些情况下甚至是延迟不敏感电路），但是具有延迟匹配的数据通道是自同步的。采用4相双轨方法实现的电路通常为准延迟不敏感电路。图2.12和图2.14所示的电路中，鉴于连接到或门输入端的分支是延迟不敏感的，电路中连接到各个C单元输入端的分支必须是等时的。

在构建完整的系统时，DI, QDI, SI和自同步这几种不同类型的电路并不是相互排斥的，对其进行有效的提炼可以应用在不同的设计层次上。在大部分实际的设计中，都采用各种电路混合的设计方法。例如，在Amulet处理器[44, 43, 48]设计中，SI设计方法被用于局部异步控制器的设计，捆绑数据用于局部数据处理，DI用于高层部分。另一个例子是文献[103]给出的助听器滤波器音色库设计。它在RAM模块和算法电路内采用DI 4相双轨协议来提供鲁棒性很强的指示完成信号，在顶层设计使用带有SI控制的4相捆绑数据，这与Amulet设计稍有不同。为了使异步数字系统设计最优化，需要特别考虑对于握手协议和电路实现类型的选择。

还必须强调的一点是，速度无关和延迟不敏感是两种数学性质，可以通过一个给定的电路来加以验证。如果电路中的某个抽象元件——比如一个C单元或一个复杂的“与或非”门，用一些简单的门以及分支线来实现，则此时这个电路可能不再是速度无关或延迟不敏感电路了。例如，图2.8和图2.15所示电路中的简单Muller流水线，如果用图2.5所示的用与门和或门组成的门级电路来代替C单元，则此时这个电路将不再是延迟不敏感电路。此外，即使简单的门也是抽象的；CMOS电路中的基本元件是N型和P型晶体管，并且最简单的门也含有分支。

在第6章中我们将详细介绍SI控制电路的设计（因为相关的理论和综合工具已经很完善了）。由于SI电路中完全忽略了导线上的延迟，因此在电路的物理实现过程中还需要特别注意。常常有人会认为在10~20个门组成的较小的电路中，零导线延迟的假设一般能够满足，但实际上往往并非如此：普通的布局布线CAD工具可能会把控制器的门分布在芯片的各个位置。即使所有门布局在一起，但由于各个门的输入有不同的逻辑阈值，使得当电路中出现了缓慢的上升或者下降信号时，会引起电路的误操作。对于静态CMOS以及运行在低电源电压（即$V_{DD} \sim Vt_N + |Vt_P|$）的电路来说，可能不会出现什么问题，但是对那些采用高电压V_{DD}（即3.3 V或5 V）的动态电路来说，逻辑阈值可能会有很大差别，这个经常容易被忽略的问题在文献[134]中有详细的说明。

2.6 测试

进行异步电路的商业开发时，会面临电路的测试问题。测试本身就是一个重要的主题，但它并不属本书的讨论范围，我们只是说明一些有关测试的基本问题和面临的挑战。虽然以下的内容比较简单，但也涉及测试的一些知识。这些内容与后面的章节没有什么联系，读者如果对这部分内容没兴趣的话，可以直接跳过。

前面所讨论的Muller电路、指示的原理以及等时分支等内容与电路的“固定故障”（stuck at fault）有非常密切的联系。在固定故障模型中，故障在门级被建模为每个输入与输出的固定于1故障或固定于0故障。指示原理说明，门的所有输入信号的跳变一定能够通过门的一个输出信号跳变来指示。此外，异步电路频繁使用握手，使得信号在0与1之间进行周期性的跳变。在这种情况下，固定故障的出现可能导致电路停止运行；如果电路中的某个元件停止握手，则

会影响到其相邻的元件，最终整个电路都会停止运行。因此，开发一套用来对所有的故障进行测试的测试方法，可以理解为开发一套可以切换所有节点的测试方法，这比较容易实现。

由于等时分支电路中的分支信号的跳变不能由与其相连的门来指示，从而等时分支可能会出现不可测的固定故障。

对异步电路的测试还可能有其他的问题。我们在后面的章节中可以看到，在异步电路中倾向于用锁存器来实现寄存器而不用触发器。又由于电路中不存在全局时钟，所以电路不能够直接地把寄存器连接到扫描路径中。分布式自同步控制的另外一个问题是，电路不能够单步地依次遍历电路的所有状态，这使得电路很难进入I_{DDQ}测试时所必须处于的静止状态，I_{DDQ}测试是当今CMOS工艺中用于对电路典型的短路和开路的测试技术。

电路中的状态保持元件（如Muller C单元）的大量使用以及自同步的特性，使得很难对那些实现状态保持的反馈电路进行测试。对延迟故障的测试又是另一个挑战。

以上所讨论的问题会让我们觉得异步电路测试中还存在很多未解决的问题。其实并不完全如此，实际情况是，不能直接把测试同步电路的方法用于测试异步电路。这种情形与下一章将要讨论的异步电路的设计是很相似的。这里需要将新的测试技术和已有的测试技术结合起来。文献[120]是一篇关于异步电路测试的入门文章。最后，我们将会在第13章和第15章中提到一些测试方法。

2.7 小结

本章介绍了许多基本概念。我们现在将回到设计异步电路的主题上。读者在阅读以后的章节时，可能还会再来阅读本章的有关内容。

第3章　静态数据流结构

本章将在更高层次上研究异步设计，这相当于同步设计中的RTL级（寄存器传输级）。在该层次上，可以把异步电路抽象为静态数据流结构，目标主要集中在电路的行为特性，并且对握手信号进行抽象，忽略细节，以便能够考虑直接实现的问题。

3.1　引言

前面所介绍的各种握手协议和相应的电路实现方式存在很大的差异。然而，当从更高的层次上审视电路时，会发现差异明显减小，如第1章中介绍过的数据流握手通道层。这有助于选择合适的握手协议和电路实现方式，高层的设计往往取决于整个电路的结构和运算，而低层的实现独立于高层的设计。

本章中的电路采用常用的4相协议。从数据流的角度来看，我们将对有效值和空值交替出现的数据流进行处理——在2相协议中，我们看到的则是由有效值组成的数据流，除了这一点与4相协议不同外，其他的都是相同的。我们还将讨论用简单锁存器作为存储元件的问题。锁存器是按照第1章中规定的法则来进行控制的：

> 如果某个锁存器的托肯被后续锁存器输入并存储，则允许该锁存器从其前向锁存器中输入和存储一个新的托肯（有效或空值）。

在握手过程中，锁存器是唯一的初始化和主动元件，其他元件对于握手都是“透明的”。为了进一步减少锁存器和组合电路之间存在的差异并强调电路图中的托肯流，在以后的章节中我们将采用一个带有两条竖线的方框来表示锁存器（见图3.1）。

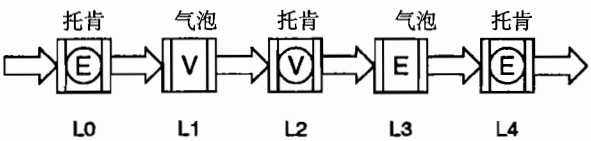


图3.1　5级流水线可能的状态

3.2　流水线和环路

图 3.1 是由 5 个锁存器组成的流水线的示意图。图中的“空心箭头”代表的是含有请求信号、应答信号和数据信号的通道或连接。L1 中的有效值被复制到 L2 中，同时 L3 的空值被复制到 L4 中。这意味着 L1 和 L3 存储的是 L2 和 L4 存储值的旧副本（old duplicates），把这种旧副本称为“气泡”（bubble），而最新 / 最右边的有效值和空值称为“托肯”（token）。为了区别它们，托肯用加圈的值来表示。这样，锁存器存储的可能是气泡，也可能是有效值托肯或空托肯。气泡可视为一种催化剂：气泡允许托肯向前移一步，同时该气泡向后移一步。

任何电路中必须要有气泡，否则电路将会陷入死锁状态。稍后我们将会详细讲解正确初始化电路的问题。此外，后面还会介绍气泡的数量对电路性能的影响。

在不少于3个锁存器构成的流水线电路中，可以通过将最后一级的输出与第一级的输入相连，形成一个数据托肯可以自动循环的环路。假设这个环路在 t_0 时刻的初始状态如图 3.2 (a)所示，此时环路中含有一个有效托肯、一个空托肯和一个气泡，在 t_1, t_2, t_3 时刻的循环过程如图 3.2(b)所示。环是实现迭代计算的主要结构。图 3.2 中环的周期是 6 “步”（t_6 时刻和 t_0 时刻是同一状态）。有效托肯和空托肯都沿圈绕行，绕行一周需要 3 “步”。由于环路中只有一个气泡，因此电路的周期为 6 “步”。初始状态为一个有效托肯、一个空托肯和两个气泡的 4 级环，其周期为4 “步”，即使增加一个锁存器也不会改变电路的功能（而在同步电路中往往会引起相应的改变），此时仍然是一个单数据托肯循环环路。

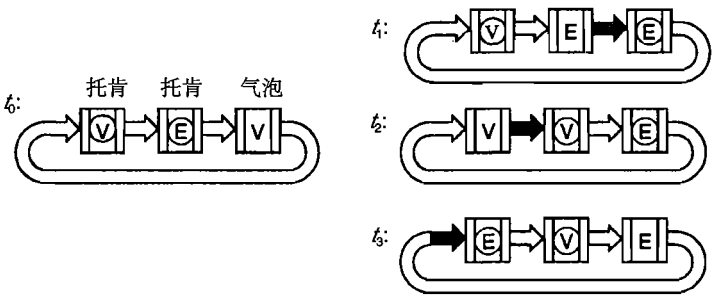


图 3.2　(a) 环可能的状态；(b) 数据传输顺序

3.3　构造块

图 3.3 给出的是组成异步电路（具有确定行为的静态数据流结构，如不含仲裁器）元件的最小集合，这些元件分成4类。在下一节中，我们将会看到由这些元件构成的数据流结构的例子，互斥和仲裁元件将在 5.8 节中介绍。

锁存器：用来存储变量和实现支持托肯流的握手电路。除了这些一般的锁存器外，电路中常常还用到一些退化的锁存器：只有一个输出通道的锁存器是产生托肯的源（这些托肯具有相同的值）；只有一个输入通道的锁存器是消耗托肯的阱。图 2.9 所示是一个 4 相捆绑数据的锁存器，图 2.11 是一个 2 相捆绑数据锁存器，图 2.12 和图 2.13 表示的是一个 4 相双轨锁存器。

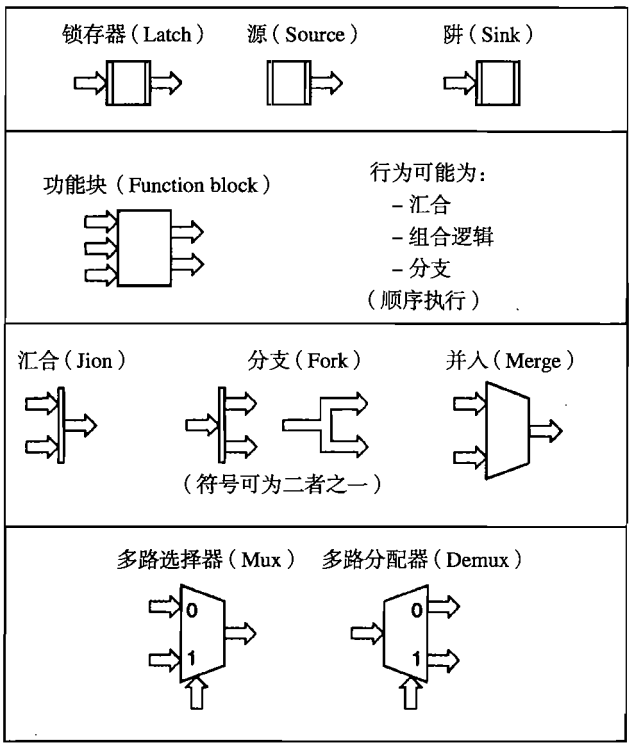


图 3.3 常用异步元件集

功能块：它是组合电路的异步等效电路。从握手角度来看功能块是透明/被动的。功能块将：(1) 等待输入托肯（隐式汇合）；(2) 实现要求的组合功能；(3) 输出托肯。空托肯和有效托肯都是以这种方式处理的。某些实现可能需要输入同步化，此时就可需要采用一个隐式汇合元件。在第 5 章中我们将详细介绍功能块的实现问题。

无条件流控制：分支和汇合元件通常用于并行线程运算。用工程术语来讲，当某个元件的输出送到多个元件的输入时要使用分支；而当来自于多个独立的通道的数据（典型情况是，这些数据作为电路的独立的输入）需要同步化时常常要使用汇合。在后面的章节中，电路图中常常省略汇合和分支部分：一个通道的扇出隐含一个分支，而多个通道的扇入隐含一个汇合。

并入（merge）元件具有两个或多个输入通道以及一个输出通道。输入通道的握手被认为是互斥的，通过并入元件将输入托肯和握手传到输出端。

条件流控制：多路选择器（MUX）从多个输入端进行选择输出，多路分配器（DEMUX）则把输入送到多个输出中的某一个。控制输入是一个类似于数据输入与输出的通道。多路选择器对控制通道和相关的输入通道进行同步，并且把指定输入通道的数据输出，而把其他的输入通道忽略。同样，多路分配器对控制通道和数据输入通道进行同步，并且把输入数据输出到已选定的输出通道。

前面提到，锁存器用于实现电路的握手，因此托肯可以在电路中流动。但所有其他元件对于握手都必须是“透明的”，这对于这些元件的实现是非常重要的。

3.4　一个简单的例子

图 3.4 所示为一个由锁存器、分支和汇合构成的电路，我们将用它来说明异步电路托肯流的行为特性。这个电路结构可以看作流水线的一部分，也可以看作是利用分支和汇合元件把一个环路连接到更大的结构中。

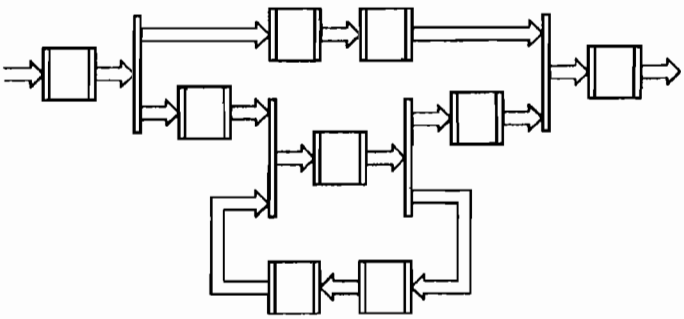
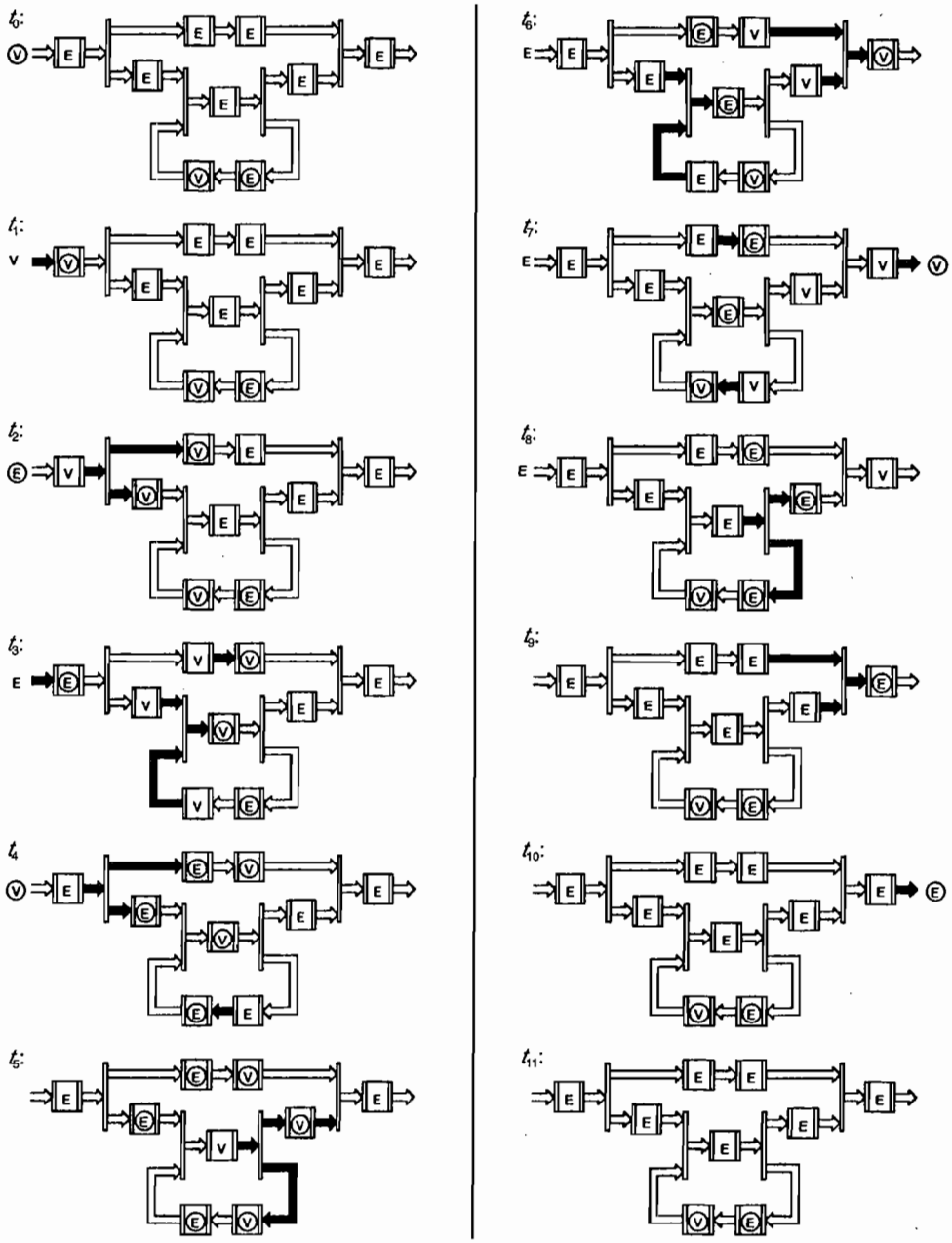


图 3.4　由锁存器、分支和汇合组成的异步电路

假定 t_0 时刻电路的初始状态如图 3.5 所示，除了电路底部环路中的两个锁存器分别初始化为有效值和空值外，其余所有的锁存器都初始化为空值。图中用圆圈包围的值代表托肯，而没有包围的则表示气泡。假设左右两边的外部环境（图中没有表示）也参与握手，此时电路已经准备运行。电路的运行步骤 $t_0 - t_{11}$ 如图中所示，左边外部环境完成一个握手周期并输入一个有效值和一个空值。同样，右边外部环境参与一个握手周期并且消耗一个有效值和一个空值。

由于托肯的流动是由局部握手来控制的，所以电路还会表现出其他的行为。例如，在 t_5 时刻电路准备从左边环境接收新的有效值。还可以看到，如果在初始状态时环路中没有托肯，则运行几步后电路会进入死锁状态。我们建议读者试着去继续研究这个托肯气泡数据流“游戏”，研究在不同的初始状态下该电路的行为。



图例： Ⓥ 有效托肯 Ⓔ 空托肯
v 气泡 ε 空值

图 3.5 图 3.4 所示电路可能的操作顺序

3.5　环路的简单应用

这一节给出几个以单个环路为基础的简单电路。

3.5.1　顺序电路

图 3.6 所示是有限状态机的一种简单实现方式，其结构与同步有限状态机很相似；它由一个功能块和一个用来保持当前状态的环路组成。状态机接受一个与“当前状态托肯”汇合的“输入托肯”，接着功能块计算输出与下一状态，最终分支把结果分成一个“输出托肯”和一个“下一状态托肯”。

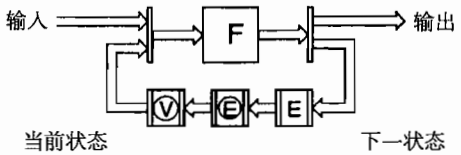


图 3.6　使用环路实现的异步有限状态机

3.5.2　迭代计算

环路同样可以用来构造实现迭代计算的电路，图 3.7 是这种电路的例子。电路的运行过程是：(1) 接收操作数；(2) 连续执行同样的操作，直到迭代运算终止；(3) 输出结果。图中的电路只是实现迭代计算的一种方式，并未标出必需的控制部分。改变锁存器和功能块的位置，或者把功能块进一步分解，并把这些更简单的功能块放置在锁存器之间，这样就可以得到迭代计算电路的其他实现方式。在文献[156]中，Ted Williams 提出了一个用自同步 5 级环路实现的除法电路。这一设计后来被应用于商业微处理器的浮点协处理器中[157]。

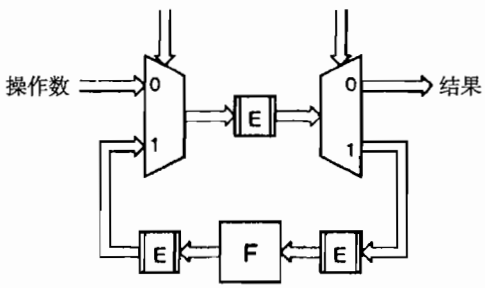


图 3.7　使用环路实现的迭代计算

3.6 FOR, IF 和 WHILE 结构

电路的功能通常是用程序语言（C, C++, VHDL, Verilog 等）来描述。这一节中我们将给出典型条件结构和循环结构的实现模板。熟悉控制数据流图（control-data-flow graph）的读者，从高级综合的角度来看，或许会发现异步电路与控制数据流图非常相似[36, 127]。

If <cond> then <body1> else <body2>　图 3.8(a)表示的是一个 if 语句的异步电路实现模板。If电路中输入输出通道的数据类型是包含<cond>表达式中所有变量以及<body1>和<body2>中操作变量的记录（record）。cond块输出通道输出布尔型数据，该布尔数据用来控制DEMUX和MUX元件。为了简洁，图中省略了与通道相关的分支电路。

由于<body1>和<body2>的执行是互斥的，因此电路底部的受控多路选择器可用简单的并入电路来代替，如图 3.8(b)所示。图 3.8 所示的电路中没有反馈环节和锁存器——因此可视为一个功能块。为了改进电路的性能，可以通过插入锁存器来构成流水线电路。

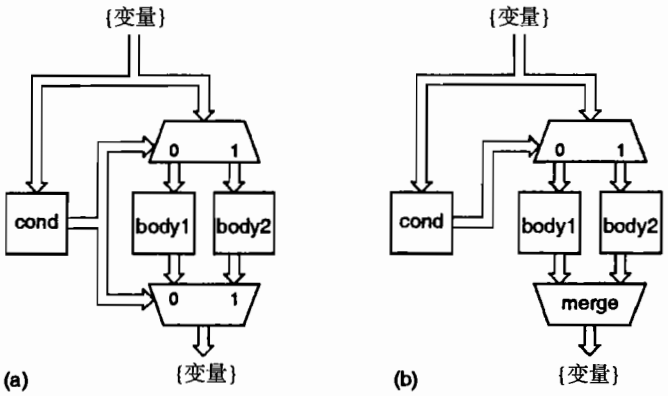


图 3.8　if 语句实现模板

for <cond> do <body>　图 3.9 所示是一个 for 语句异步电路实现模板。for 电路中输入通道的数据类型是包含<body>以及循环计数<count>中所有变量的记录，<count>中的变量是非负整数。输出通道的数据类型是包含<body>中所有变量的记录。

<count>模块输出通道的数据类型是布尔型，<count>模块输入通道上的一次握手会引起输出通道的<count>次握手：<count> –1 次握手提供布尔值 1，最后一次握手提供布尔值 0。初始化 MUX 的两个输入锁存器时，必须把他们一个置为“0”托肯，另一个置为空托肯，使得 for 电路能够将变量读到环路中来。

一旦执行完 for 语句，count 块的最后一次握手会把环路中的变量输出到输出通道，并且置两个锁存器分别为“0”托肯和空托肯，这就为 for 电路的下一个动作做好了准备。图中省略了 count 块输入通道的分支和输出通道的分支，同时图中还省略了一些锁存器。读者需要注意以

下几点：(1) 所有的环路必须至少包含 3 个锁存器；(2) 在 4 相握手的情况下，当有一个锁存器初始化成一个数据托肯时，必定要有另一个锁存器初始化成空托肯。

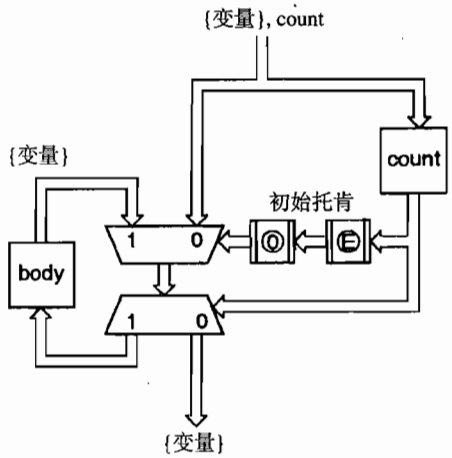


图 3.9　for 语句实现

while <cond> do <body>　实现 while 语句的异步电路模板如图 3.10 所示。电路的输入和输出是<cond>表达式中的变量，这些变量由<body>控制。与前面的 for 电路一样，while 电路也必须把 MUX 的控制输入中的两个锁存器分别初始化为“0”托肯和空托肯。同样，图示电路中也省略了构成两个环路的一些锁存器。当 while 电路终止时，从环路中得到输出数据，并使 MUX 的控制输入锁存器进行恰当的初始化，为电路的下一次工作做好准备。

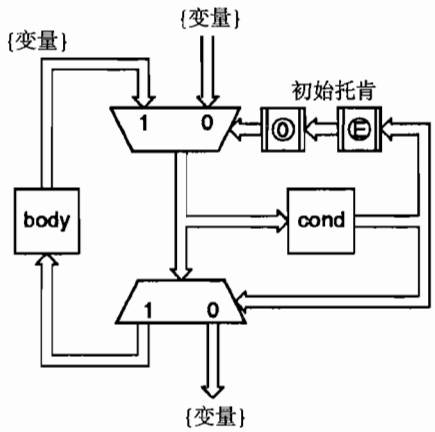


图 3.10　while 语句的实现

3.7 更复杂的例子：最大公约数电路

利用上面介绍的模块，我们来设计一个简单的最大公约数（Greast Common Divisor, GCD）电路，这个电路用来计算两个整数的最大公约数。GCD常常被用来作为一个入门性的实例，图3.11是GCD算法的程序。

```
input (a, b);
while a ≠ b do
    if a > b    then a ← a − b;
                else b ← b − a;
output (a);
```

图3.11 采用编程语言实现GCD

GCD除了作为一个设计实例外，还常用它来比较各种设计方法的异同点。在第8章中我们还将采用这个例子来解释Tangram语言和与面向语法（syntax-directed）有关的编译过程。

GCD的实现电路见图3.12。电路中有一个while模块，while模块的body是一个if模块。图3.12所示的电路中标出了所有必要的锁存器（标有初始状态）。电路的实现没有使用共享资源，只是用前面介绍的那些模块直接映射得到。

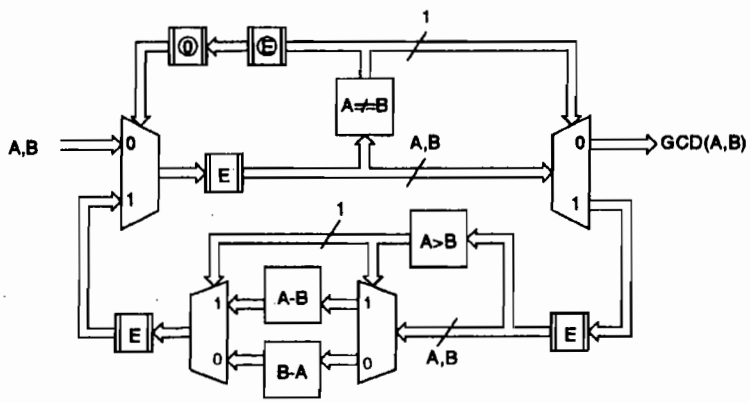


图3.12 使用异步电路实现

3.8 补充例子

3.8.1 低功耗滤波器组

在文献[103]中给出了数字助听器中低功耗IFIR滤波器的设计。在本章前面介绍过这个电路的设计方法。文中给出了关于低功耗电路设计、存储结构和数据通道单元电路级实现的一些见解。

3.8.2 异步微处理器

在文献[23]中，我们给出了MIPS系列微处理器（称为异步精简指令计算机，ARISC）的设计。在设计诸如微处理器这类大规模数字电路时，设计者必须了解很多具体的细节，图3.13只给出了微处理器的一些基本结构，这些结构可以视为简单的数据流结构。图中黑色长方形代表锁存器，而宽箭头表示通道，文本框表示功能块（组合电路）。

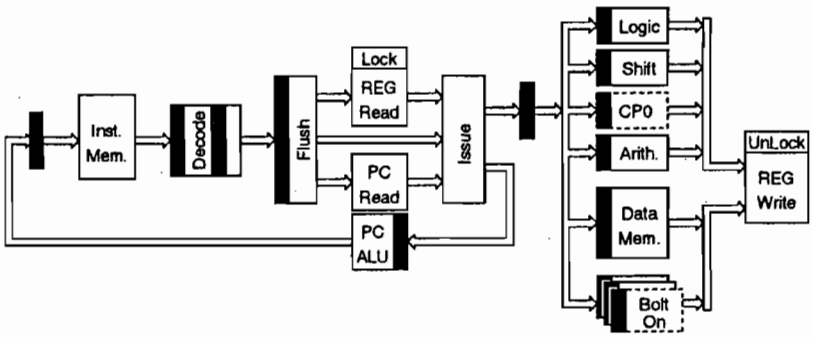


图3.13 ARISC微处理器结构

处理器可以视为一个按照一定的程序顺序安排指令的简单流水线结构，它包括一个具有固定托肯数的取指-译码-执行（fetch-decode-issue）的环路，确保了固定的指令预取深度。执行段把译码后的指令分到可执行流水线中，并且开始再取一条指令。电路可以通过一个锁定机制来避免寄存器推进（forwarding）：当发出一条指令后，指令执行的目标寄存器一直处于锁定状态直到回写（write-back）发生。如果后面的指令具有写后读（read-after-write）数据冒险，这条指令被推迟到寄存器开启后执行，设计中的托肯流包含所有有关指令的操作与控制信号，即类似于被存储在同步处理器中的流水线段，读者若想了解更多的信息，可参考文献[23]，其他异步微处理器建立在类似原理的基础之上。

3.8.3 细粒度流水线矢量乘法器

前面章节所提到的GCD和ARISC都采用并行通信通道。文献[124, 125]中的串并矢量乘法器设计是采用1位通道和细粒度流水线的静态数据流结构的实例。这里所有必要的字级（word-level）同步化都是由功能块来实现的。静态数据流结构中存在的大量相互作用的环路和流水线使得电路的设计相当复杂。当完成了下一章中有关电路性能分析的学习后，有兴趣的读者可能会想进一步分析这个设计，它包括了几个有趣的优化方法。

3.9 小结

本章在更高层次——静态数据流结构层次上研究了异步电路设计，这相当于同步设计中的RTL级设计。下一章将在这一抽象层次上进行性能分析。

第4章 性 能

本章讲述了异步电路的性能分析与优化，其内容在前一章介绍的“静态数据流结构”基础上加以扩展。

4.1 引言

在同步电路中，电路的性能分析和优化主要是找出两个寄存器之间的最长信号延迟路径，它决定了电路的时钟周期。全局时钟把电路分解成多个可以独立进行分析的组合电路模块，即使对于大型电路，也很容易分析，这就是所谓的静态时序分析。

而对于异步电路，性能分析和优化是基于全局的，因此也更为复杂。电路中采用的握手，使得电路某单元的时序与它相邻单元的时序有关，而相邻单元的时序又与它们其他相邻单元的时序有关。因此，电路的性能不仅取决于其结构，还与电路的初始状态以及外部环境相关。异步电路的性能还可能呈现出瞬变性（transient）和振荡性（oscillation）。

首先，定性的了解一下异步电路中托肯流的动态特性，这对于设计具有良好性能的异步电路非常重要。然后，引入几个定量的性能参数来描述单个流水线级以及由多个相同流水线级组成的流水线或环的特性。设计者利用这些参数可以做出初步的设计决策。最后，将讨论如何分析复杂和非规则结构的电路。

以下内容主要取自于文献[124]并对其进行了修改，它建立在Ted Williams原创性工作[153, 154,155]的基础上。读者要阅读这些相关文献，必须掌握托肯的准确定义。在本书中，“托肯”被定义为一个有效值或一个空值，然而在某些被引用的专门针对4相握手的文献中，托肯表示的是一个“有效值-空值”数据对。本书中托肯的定义主要是强调异步电路中托肯与Petri网中托肯的相似性。此外，也给出了4相握手与2相握手之间的统一性，即2相握手除了没有空托肯这一点以外，实质与4相握手是一样的。

在后续部分中，我们假设电路采用的是4相握手，所给出的例子都使用捆绑数据电路。对于采用2相握手的电路，只需做简单的修改，这里作为练习留给读者来完成。

4.2 电路性能的定性分析

4.2.1 例1：移位寄存器的FIFO

下面通过一个FIFO的例子来说明基本概念：这个FIFO由一系列锁存器组成，其中N个有效托肯被N个空托肯分开，外部环境从FIFO读取一个托肯，同时也往FIFO写入一个托肯，如图4.1(a)所示。因此，FIFO内的托肯数是不变的。之所以举这个例子，是因为很多设计者都是使用这种FIFO，同时它模拟了移位寄存器和环的行为，它们结构中的托肯数量保持不变。

吞吐量是一个与其相关的性能指标。吞吐量是指移位寄存器输入或输出托肯的速率。这个指标与锁存器链中的内容右移一位所需的时间有关。

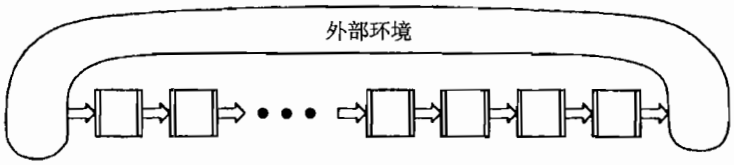
假设FIFO电路中有效托肯数为N，图4.1(b)所示是具有$2N$个锁存器的FIFO的行为，而图4.1(c)则是具有$3N$个锁存器的FIFO的行为。在这两个例子中，FIFO的有效托肯数都是$N=3$，而这两种结构的唯一区别是所含的气泡数不同，见图4.1(b)和图4.1(c)。

在图4.1(b)中，t_1时刻外部环境读取有效托肯D1，图中用一个实心箭头符号表示。同时引入了一个气泡，这个气泡能够使得t_2-t_5时刻的数据按一定的顺序移动一位。在t_6时刻，从外界输入一个有效托肯D4，此时电路中的所有托肯都已右移了一位。因此，使所有托肯右移一位所需要的时间与托肯数成正比，图中情况需要$2N=6$个时间步（time step）。

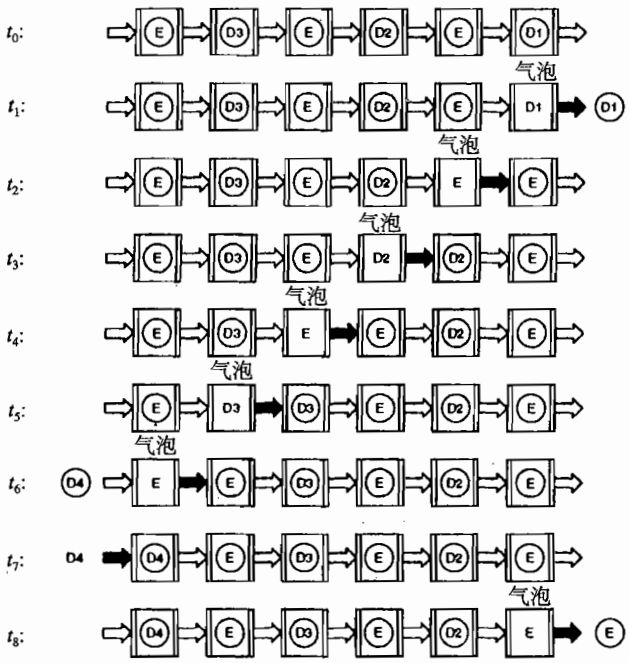
在电路中，增加锁存器数量会使气泡的数量相应地增加，而气泡数量的增加又会增加同一时刻移动的数据的数量，从而改善了电路的性能。在图4.1(c)中，移位寄存器有$3N$级，即每个有效-空托肯对有一个气泡。这就使得在同一时刻有N位数据移位，而使所有单元往右移一位所需要的时间是两个时间步。

如果锁存器的数量增加到$4N$，则每个托肯对应一个气泡，而所有单元右移一位只需要一个时间步。这种情况下，流水线只有一半被托肯填满。相对于存储托肯的锁存器来说，存储气泡的锁存器相当于一个从锁存器。此时，即使进一步增加气泡的数量也不会改进电路的性能了。最后，我们发现在图4.1(b)中只需要增加一个带有气泡的锁存器就能够使电路的性能加倍。异步电路设计者在用更多的锁存器换取电路性能方面有较大的自由度。

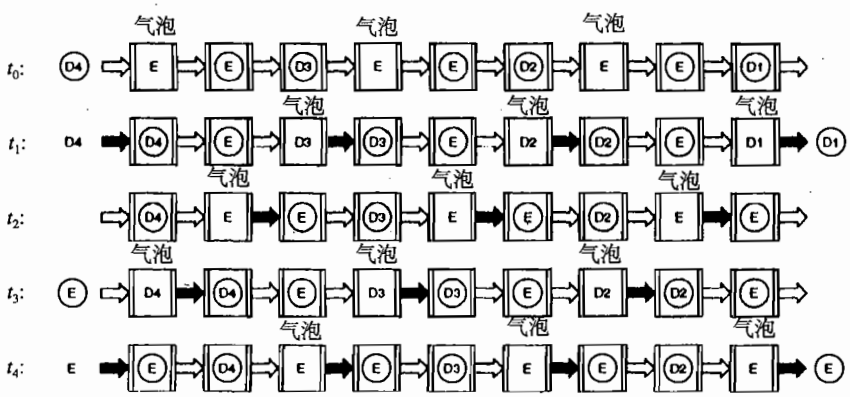
由于设计中气泡的数量取决于每个托肯对应的锁存器的数量，以上的分析表明，在给定的电路进行性能优化中最关键的是对电路结构改进，而电路级优化，诸如晶体管尺寸等是次要的。



(a) FIFO 及其外部环境



(b) 2N 级中的 N 位数据托肯和 N 位空托肯



(c) 3N 级中的 N 位数据托肯和 N 位空托肯

图 4.1 FIFO 及外部环境。外部环境在读取托肯和写入托肯之间进行切换

4.2.2 例2：并行置数移位寄存器

为了说明电路中托肯与气泡的分布随着电路的动态和外部环境改变而改变，这里另举一例：并行置数移位寄存器。图4.2是一个4位移位寄存器的原始设计。电路具有与数据生成环境相连的位并行输入通道din[3:0]，以及与数据消耗环境相连的位数据通道do和位控制通道ctl。数据消耗环境要求电路如下操作：当ctl = 0时，执行并行置数并且把位并行通道中的最低有效位赋给do通道；而当ctl = 1时，右移并把并行通道中的下一位赋给do通道。这样，从数据消耗环境输入一个控制托肯（有效或空值）给电路，而电路输出一个数据托肯（有效或空）来响应。在并行置数过程中，移位寄存器中以前的内容导入“终端”阱锁存器（“dead end” sink-latches）。右移时，把常数0移入最高有效位——相当于逻辑右移。数据消耗环境并非一定要读取所有的输入数据位，它也可继续读取最高有效位外移入的0。

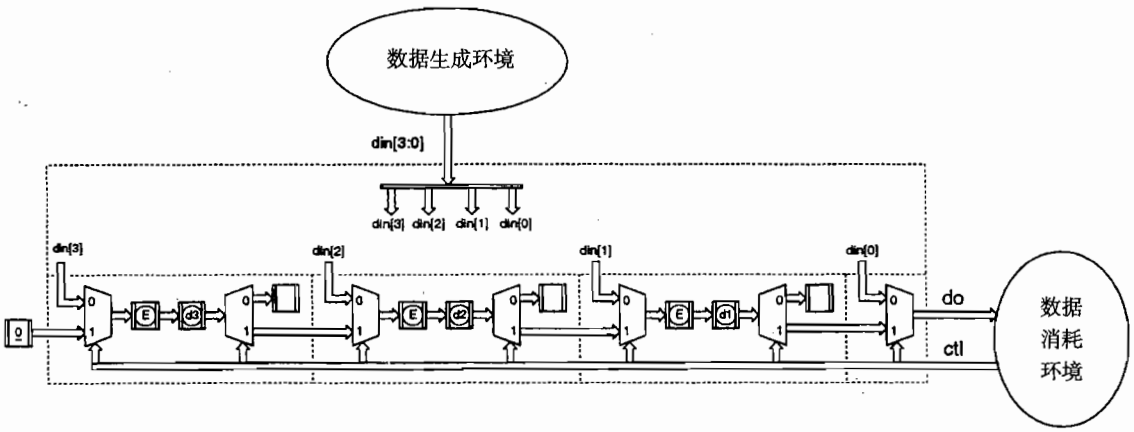


图4.2 并行置数移位寄存器的原始设计

图4.2所示的原始设计其性能有两方面的缺陷：首先，它和图4.1(b)所示的移位寄存器具有同样的问题——电路中气泡的数量太少，电路位串行输出的最高速率随移位寄存器的长度增加而线性下降；其次，在设计中要把控制信号加到所有的多路选择器和多路分配器上，这意味着请求和数据信号有很大的扇出，需要加多个缓冲器，同时要对所有的单个应答信号进行同步，需要多个输入端的C单元或者C单元树来实现。第一个问题可以通过在每级数据通道中加第三个锁存器来避免，如图4.1(c)所示，但如果把额外的锁存器加入控制通道，则这两个问题都可以得到解决，如图4.3(a)所示。

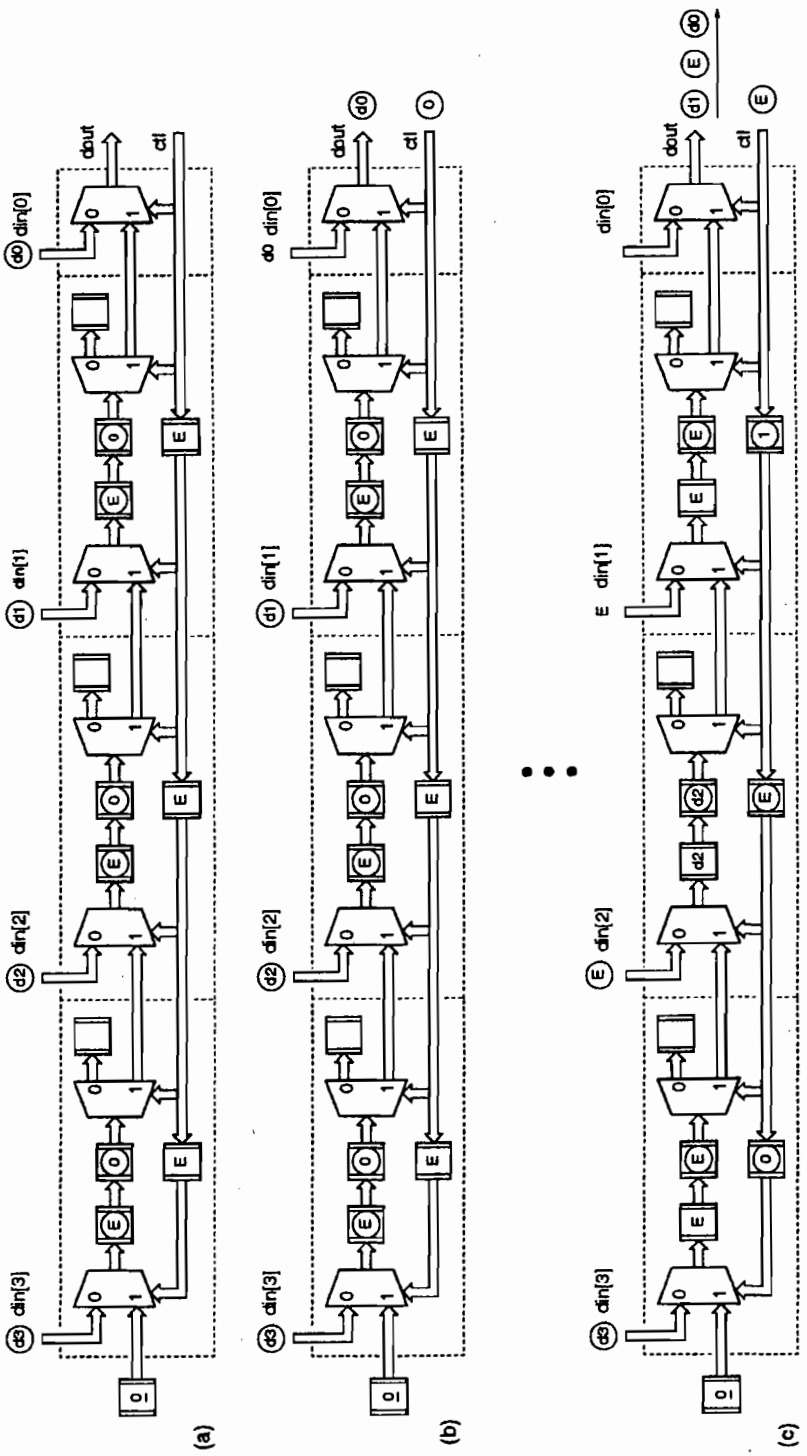


图 4.3 并行置数移位寄存器的改进型设计

设计改进后的动态行为说明如下：初始时，数据锁存器都存储托肯，所有控制锁存器存储气泡，如图4.3(a)所示。并行置数循环的第一步如图4.3(b)所示，而图4.3(c)表示的是数据消耗环境读取一组数据位后电路的状态。最高位准备执行“并行置数”，而气泡位于数据锁存器链中。如果此时数据消耗环境暂停，则控制通道中的托肯将慢慢地消失，而数据通道的托肯会再次补充。由此，我们注意到任何时刻电路中的托肯总数是恒定不变的。

4.3 性能的定量分析

4.3.1 延迟、吞吐量和波长

在确定了设计的总体结构之后，要确定设计中环和流水线的最优锁存器个数或最优流水线级数。为了能够建立“首位决策”（fisrt order design decision）的基本概念，本节将介绍一些定量的性能参数，我们的讨论只限于对4相握手和捆绑数据的电路实现，并且只讨论具有单个有效托肯的环路。4.3.4节中会对4.3.1节中的性能参数进行小结，并对其他协议与实现形式进行探讨。

流水线电路的性能通常由两个性能参数来描述：延迟和吞吐量（或其倒数：周期或循环时间）。对于异步流水线电路，另一个参数动态波长（dynamic wavelength），也是衡量电路性能的一个重要参数。参考图4.4和文献[153, 154, 155]，这些参数定义如下。

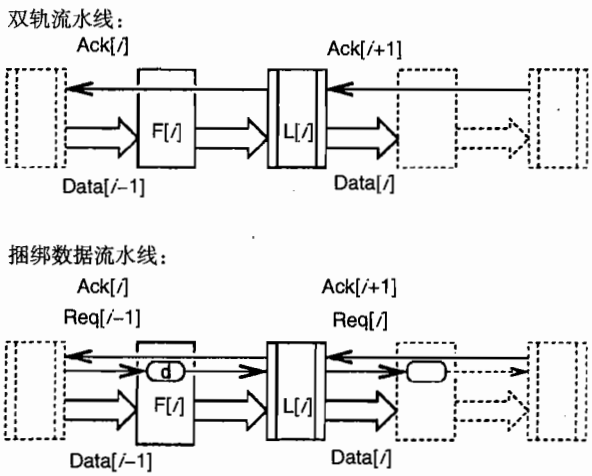


图4.4 普通流水线中性能参数的定义

延迟：从数据输入到相应的数据输出的延迟。数据向前流动，而应答信号沿相反的方向流动。由此可以定义以下两个参数。

- 前向延迟L_f：假定应答信号先于数据到达，流水线从输入新数据（Data[$i-1$]或Req[$i-1$]）开始，直到相应的输出（Data[i]或Req[i]）生成之间的延迟。$L_{f.V}$和$L_{f.E}$分别表示传送有效托肯和空托肯的延迟。通常认为这些延迟是不变的常数，即它们是与传送的数据值无关（由于空托肯在前向传送过程中没有“计算”部分，因此可以用$L_{f.V}$的最小值来近似表示$L_{f.E}$。在4相捆绑数据方法中可以通过采用一个不对称延迟元件来实现）。
- 反向延迟L_r：假定请求信号先于应答信号到达，指从接收到后级的应答信号（Ack[$i+1$]）开始直到给前级的应答信号（Ack[i]）产生之间的延迟。$L_r\uparrow$和$L_r\downarrow$分别表示传送Ack↑和Ack↓的延迟。

周期P：指从输入一个有效托肯（其后接着一个空托肯）到输入下一个有效托肯之间的时间，即一个完整的握手周期。对一个4相协议，这个过程包括：(1) 前向传送一个有效数据值，(2) 反向传送一个应答信号，(3) 前向传送一个空数据值，(4) 反向传送一个应答信号。

因此周期的下界是

$$P = L_{f.V} + L_r\uparrow + L_{f.E} + L_r\downarrow \tag{4.1}$$

在本书中，我们所讨论的大部分电路都具有对称性，即$L_{f.V} = L_{f.E}$且$L_r\uparrow = L_r\downarrow$，因此周期可以简化为

$$P = 2L_f + 2L_r \tag{4.2}$$

对于$L_{f.V} > L_{f.E}$的电路，将在4.4.1节和7.3节中讨论。考虑到锁存器的实际实现，电路的周期要大于式(4.1)给出的周期下限值。在4.4.1节中，我们还将分析另一流水线，它的周期如下：

$$P = 2L_r + 2L_{f.V} \tag{4.3}$$

吞吐量T：$T = 1/P$，是指单位时间内通过一个流水线级的有效托肯的数量。

动态波长W_d：流水线的动态波长是指在周期P内一个前向传播托肯所通过的流水线的级数。

$$W_d = \frac{P}{L_f} \tag{4.4}$$

另一种解释是：W_d是两个连续的有效（或空）托肯之间的距离（用流水线级来度量），这时假定这些托肯在流水线中的流动不受阻碍。把有效托肯视为波峰，而把相应的空托肯视为波谷。如果$L_{f.V} \neq L_{f.E}$，则在上面的方程中应采用平均延迟：$L_f = (L_{f.V} + L_{f.E})/2$。

静态扩展（static spread）S： S是充满的流水线中两个连续的有效（或空）托肯间的距离，充满是指流水线中没有气泡。有时也用占有期来表示，占有期是S的倒数。

4.3.2 环路的循环时间

以上所定义的参数都是局部性能参数，用于描述单个流水线级的特征。当多个流水线连接成环时，则会引入以下参数来对环的性能进行描述。

循环时间： 环的循环时间T_{cycle}，指的是一个（有效或空）托肯流经环路中所有流水线级所需要的时间。为了使电路性能最好（即循环时间最小），必须使每个有效托肯对应的流水线级数与动态波长相匹配，也就是$T_{cycle}=P$。如果流水线级数过少，则循环时间会受气泡过少的影响，而如果流水线级数过多，则循环时间会受流水线级的前向延迟的制约。在文献[153, 154, 155]中，这两种工作模式分别称为气泡限制和数据限制。

含有一个有效托肯、一个空托肯和$N-2$个气泡的N级环路的循环时间可以通过以下两个方程求得（见图4.5）。

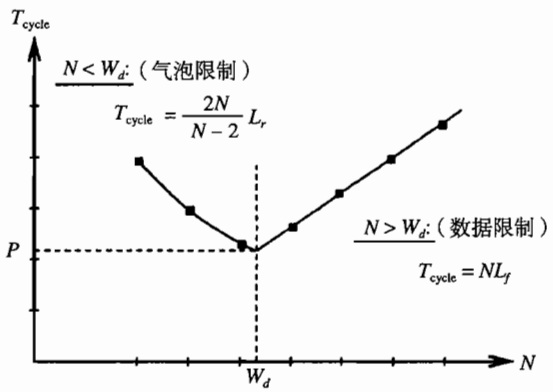


图4.5 流水线级数与环状电路周期的关系

- 当$N \geqslant W_d$时，循环时间受限于通过N级流水线的前向延迟：

$$T_{cycle}（数据限制）= N \times L_f \tag{4.5}$$

如果$L_{f.V} \neq L_{f.E}$，则采用$L_f = \max\{L_{f.V}; L_{f.E}\}$。

- 当$N \leqslant W_d$时，循环时间受限于反向延迟。对于具有一个有效托肯、一个空托肯和$N-2$个气泡的N级流水线，它的一个周期内包括了$2N$个数据传送（N个有效托肯和N个空托肯），循环时间为

$$T_{\text{cycle}}(\text{气泡限制}) = \frac{2N}{N-2}L_r \tag{4.6}$$

如果 $L_r\uparrow \neq L_r\downarrow$ 时，则采用 $L_r = (L_r\uparrow + L_r\downarrow)/2$。

为了使描述更为完整，这里必须提到，在一些电路[153, 154, 155]中还存在第三种工作模式——控制限制。然而，本书并未涉及控制限制的电路结构。

最近发表的一些论文中[31, 90, 91, 37]涉及了有关性能分析和优化的内容，而且有些论文还使用了“松弛匹配”（slack matching）这个术语，“松弛匹配”是指前向流动托肯和后向流动气泡时序之间的平衡过程。

4.3.3 例 3：3 级环路的性能分析

可以通过一个简单的3级环路来说明上述内容，该环路的每一级都是相同的4相捆绑数据流水线级，如图 4.6(a)所示。数据通道由锁存器和组合电路 CL 组成，控制部分由用来控制锁存器的C单元和反相器以及与组合电路延迟匹配的延迟元件组成。如果该电路中没有组合电路和延迟元件部分，则可视为一个简单的 FIFO 电路。为了便于说明，控制部分元件的延迟设定如下：C 单元，$t_c = 2$ ns；反相器，$t_i = 1$ ns；延迟元件，$t_d = 3$ ns。

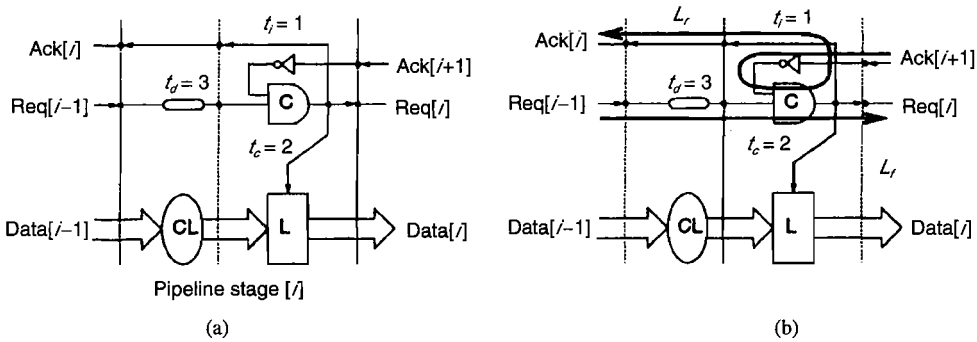


图 4.6 一个简单 4 相数据捆绑流水线及其前向和反相信号延迟路径的说明

图 4.6(b)所示为对应于前向和反向延迟的信号路径，表 4.1 列出了这些参数的表达式及它们的值。利用这些数值，可以计算出两种电路结构的周期和动态波长。FIFO 的 $W_d = 5.0$ 级，而流水线的 $W_d = 3.2$ 级。但一个环路的级数必须是整数，如果 W_d 不是整数，那么必须分析分别由 W_d 向下取整和向上取整得到的级数所组成的环路，以最小循环时间原则确定环路的级数。表 4.1 给出了 3 ~ 6 级环分析结果（包括循环时间）。

表 4.1 简单环路结构的不同性能比较

	FIFO		流水线	
参数	表达式	值	表达式	值
L_r	$t_c + t_i$	3 ns	$t_c + t_i$	3 ns
L_f	t_c	2 ns	$t_c + t_d$	5 ns
$P = 2L_f + 2L_r$	$4t_c + 2t_i$	10 ns	$4t_c + 2t_i + 2t_d$	16 ns
W_d		5级		3.2级
T_{cycle}(3级)	$6L_r$	18 ns	$6L_r$	18 ns
T_{cycle}(4级)	$4L_r$	12 ns	$4L_f$	20 ns
T_{cycle}(5级)	$3.3L_r = 5L_f$	10 ns	$5L_f$	25 ns
T_{cycle}(6级)	$6L_f$	12 ns	$6L_f$	30 ns

4.3.4 后注

上面的分析做了一些假设：(1) 环路和流水线都由相同的流水线级组成；(2) 电路的功能块对延迟具有对称性（即 $L_{f.V} = L_{f.E}$）；(3) 功能块的延迟是固定的，忽略了与数据有关的延迟以及平均性能等重要问题；(4) 环路中只有一个有效托肯；(5) 只考虑4相握手和捆绑数据电路。

对于4相双轨电路（请求信号嵌入在数据编码中），性能参数的表达式可以采用前面所定义的表达式，不需要做修改。对于采用2相协议的设计，则有必要对这些表达式做一些简单的修改：由于没有空托肯，因此前向延迟 L_f 只有一个值，而对于反向延迟 L_r 也只有一个值。对于具有多个托肯的环，获得它的循环时间的表达式也不复杂。

对于与数据有关的延迟以及非典型流水线级，比较难处理。虽然前面章节中介绍的性能参数存在一些不足，但它们可以是“首位决策”的重要基础。

4.4 相关图分析法

当流水线级含有不同的功能块或非对称延迟的功能块时，确定电路的关键路径更为复杂。因此有必要构造一个能够表示信号转换之间相关性的图形，对这个相关图进行分析并识别出关键路径周期[19, 153, 154, 155]。这可以用系统化的甚至是机械化的方式来完成，但是由于要处理大量的细节，故这项任务也相当复杂。

相关图中的节点代表信号的跳变（上升或下降），而图中的边代表信号跳变之间的依赖关系。在形式上，相关图可以视为一个标识图[28]。现在让我们来看一些例子。

4.4.1 例4：流水线的相关图

首先让我们考虑一个由相同级组成的（很长）流水线，这些流水线级采用了非对称延迟的功能块（$L_{f.E} < L_{f.V}$）。图4.7(a)所示的一段流水线有3级。每一流水线级都有如下延迟参数：

$$L_{f.V} = t_{d(0\to1)} + t_c = 5\text{ ns} + 2\text{ ns} = 7\text{ ns}$$
$$L_{f.E} = t_{d(1\to0)} + t_c = 1\text{ ns} + 2\text{ ns} = 3\text{ ns}$$
$$L_r\uparrow = L_r\downarrow = t_i + t_c = 3\text{ ns}$$

电路图与相关图之间有着非常紧密的联系。由于信号在上升(↑)与下降(↓)之间(也可以说在有效数据值与空值之间)交替，图中每个电路元件对应相关图的两个节点；同样，电路中每条线对应相关图的两个边。图4.7(b)所示的是一级流水线的两个图形片断，图4.7(c)是与图4.7(a)所示三级流水线对应的相关图。

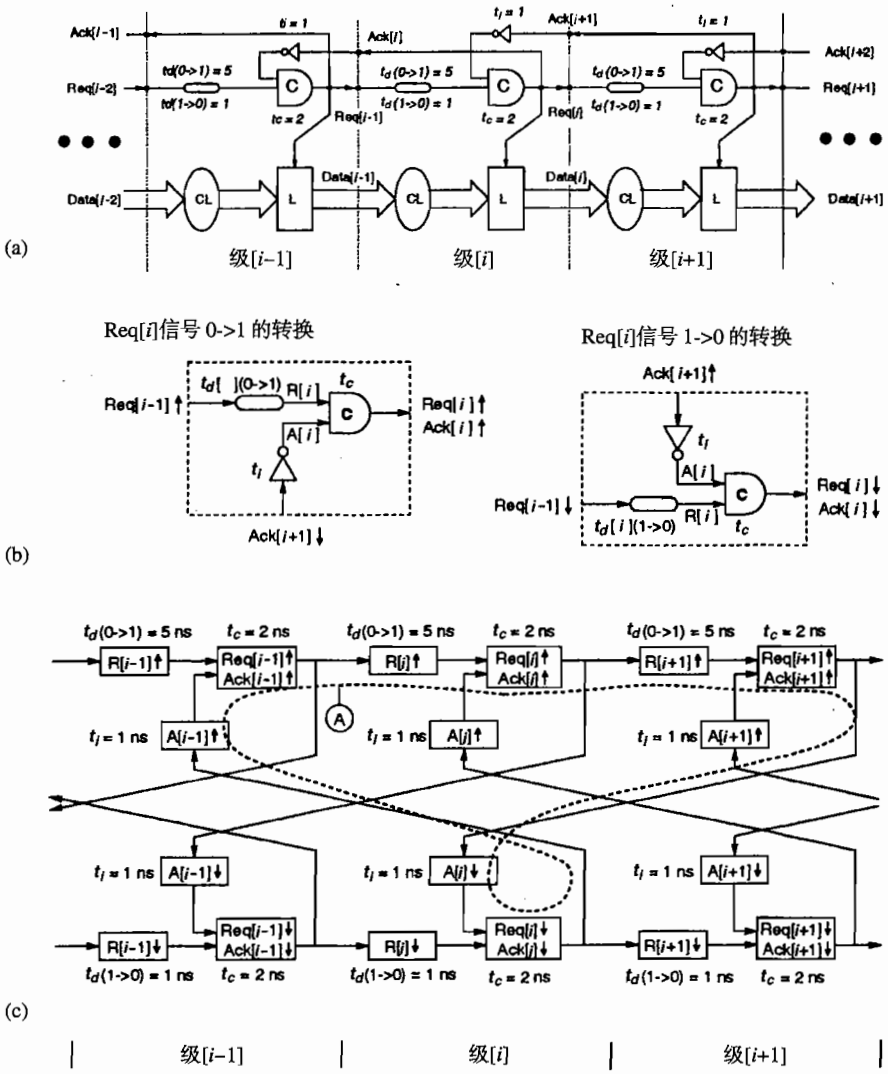


图4.7 三级流水线的数据相关图：(a) 电路图；(b) 一级流水线的两个图形片段；(c) 生成的数据相关图

节点旁的标记表示与信号跳变有关的电路延迟。为了使相关图更直观，采用了一种特殊的布局：代表前向流动的数据（有效和空值）的节点，排列成两个平行的行，而代表反向流动的应答信号的节点，则用两行之间的对角线表示。

流水线的循环时间或周期是指一个信号发生跳变到再次发生同样的跳变所经历的时间。因此流水线的循环时间可以通过找出图中最长简单回路来确定，即不包含任何子回路且总电路延迟最大的回路。图4.7(c)中虚线连起来的回路就是流水线的最长简单回路。这个回路起始于点A，其相应的周期是：

$$\begin{aligned} P &= \underbrace{t_{d(0\to1)}+t_c}_{L_{f.V}} + \underbrace{t_i+t_c}_{L_r\downarrow} + \underbrace{t_{d\,(1\to0)}+t_c}_{L_{f.V}} + \underbrace{t_i+t_c}_{L_r\uparrow} \\ &= 2L_r+2L_{f.V} = 20\ \text{ns} \end{aligned}$$

这就是由4.3.1节中公式(4.3)计算出的周期。另一种周期的计算方法是

$$\underbrace{A_{[i]}\uparrow;\mathrm{Req}_{[i]}\uparrow;}_{L_{f.V}}\underbrace{A_{[i-1]}\downarrow;\mathrm{Req}_{[i-1]}\downarrow;}_{L_r\downarrow}\underbrace{R_{[i]}\downarrow;\mathrm{Req}_{[i]}\downarrow;}_{L_{f.E}}\underbrace{A_{[i-1]}\uparrow;\mathrm{Req}_{[i-1]}\uparrow;}_{L_r\uparrow}$$

相应的周期为

$$P = 2L_r + L_{f.V} + L_{f.E} = 16\ \text{ns}$$

这就是由4.3.1节中公式(4.1)计算出的最小可能周期。这是由最长回路确定的周期，也就是20 ns。因此，这个例子说明对于某些简单锁存器电路，采用非对称延迟（$L_{f.E} < L_{f.V}$）的功能块不一定能降低循环时间。

4.4.2 例5：一个3级环路的相关图

我们再举一个用相关图进行分析的例子，它是由不同流水线级组成的3级4相捆绑数据环路，如图4.8(a)所示：第1级中有一个组合电路和一个与之匹配的对称延迟元件，第2级中没有组合逻辑部分，第3级中也有一个组合电路和一个与之匹配的对称延迟元件。

这个相关图类似于4.4.1节中的3级流水线的相关图，唯一区别是第3级的输出端口连接到第1级的输入端口，构成一个闭合的图。图中有多个“最长简单回路”可供挑选。

1. 有效托肯前向流动的回路：
   (R1↑; Req1↑; R2↑; Req2↑; R3↑; Req3↑)
   它的循环时间是 $T_{\text{cycle}} = 14$ ns。
2. 空托肯前向流动的回路：
   (R1↓; Req1↓; R2↓; Req2↓; R3↓; Req3↓)
   它的循环时间是 $T_{\text{cycle}} = 9$ ns。

3. 气泡反向流动的回路：

(A1↑; Req1↑; A3↓; Req3↓; A2↑; Req2↑; A1↓; Req1↓; A3↑; Req3↑; A2↓; Req2↓)

它的循环时间是 $T_{cycle} = 6L_r = 18$ ns。

这个3级环路包括一个有效托肯、一个空托肯和一个气泡。这个气泡参与了6次数据转换，因此对于有效托肯的每次前向环行，气泡要做两次反向环行。

4. 然而，还有一个循环时间更长的回路，如图4.8(b)所示。这个回路对应气泡的反向流动，其流动顺序为：

由(R3↑)替代(A1↓; Req1↓; A3↑)

它的循环时间是 $T_{cycle} = 6L_r = 20$ ns。

4级环路的相关图分析与上面的分析相似。唯一的区别是环路中有两个气泡。在相关图中，存在两个互不影响的“气泡回路”。

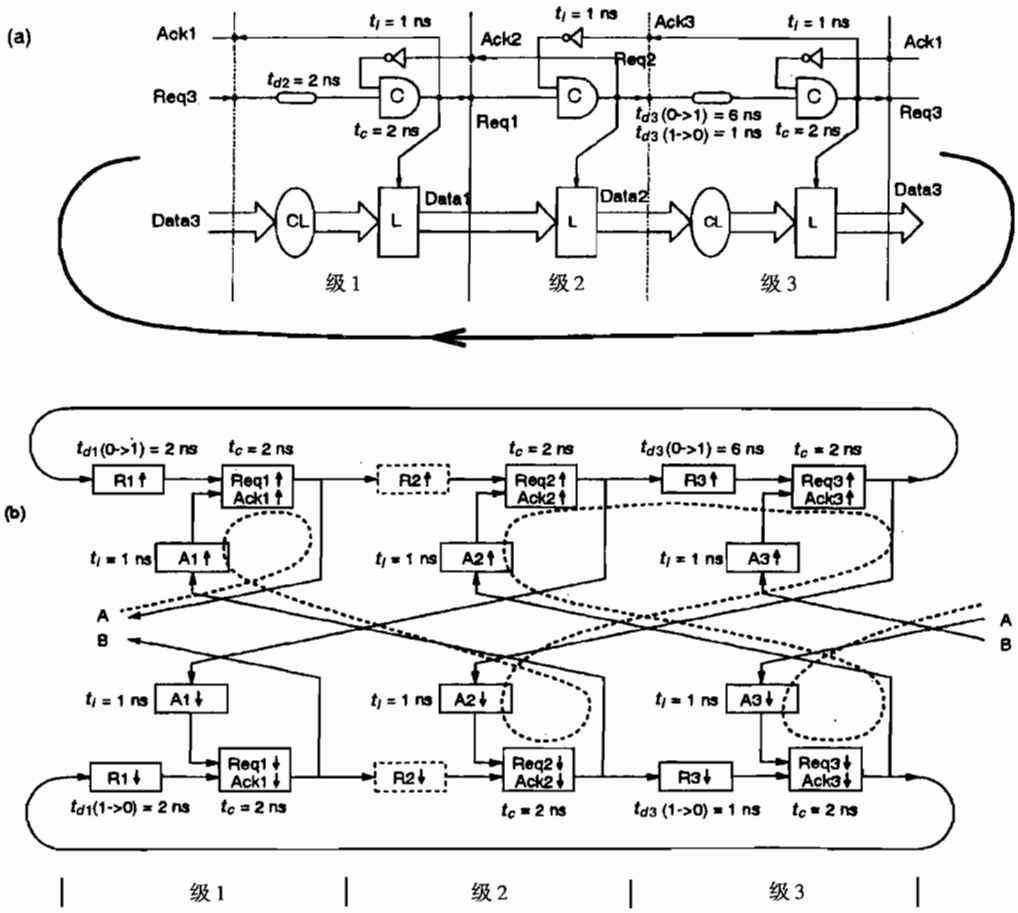


图 4.8 具有 3 级环路的相关图：(a) 环路的电路图；(b) 数据相关图

以上提出的相关图分析法，是假设一个闭合电路产生一个闭合的相关图。如果要对诸如一段流水线这样的元件进行分析，还必须包括其虚拟的外部环境的模型，通常外部环境是以独立的托肯产生环境和托肯消耗环境的形式出现的，即虚拟电路完成对握手的简单响应。图2.15说明了这种单级流水线控制电路。

以上介绍的相关图类似于信号转换图（STG），我们将在第6章中对STG进行详细的介绍。

4.5 小结

这一章在多个层面上对异步电路的性能进行了分析：首先，基于电路托肯流的动态特性，对电路性能进行定性的理解；其次，介绍了由相同流水线级构成的流水线和环的定量性能参数；最后，介绍了相关图，并应用相关图来分析由不同流水级构成的流水线和环。

到目前为止，我们已经在“静态数据流结构”层面上介绍了异步电路的设计和性能分析。在接下来的两章中，我们将介绍低层电路设计原理与技术。

第5章　握手电路实现

本章讨论握手元件的实现。首先讨论在3.3节中介绍过的几种基本元件：(1) 锁存器；(2) 无条件数据流控制元件——汇合、分支和并入；(3) 功能块；(4) 条件流控制元件——多路选择器和多路分配器。除这些基本元件外，还将讨论互斥元件（mutual exclusion element）和仲裁器（arbiter）的实现，并涉及亚稳定性（metastability）问题。本章的主要内容为5.3节至5.6节，重点是功能块的实现，主要内容包括一些基本概念和电路的实现方式。

5.1　锁存器

如前所述，锁存器的主要功能是：(1) 存储有效托肯和空托肯；(2) 通过与相邻锁存器的握手来实现托肯流。握手锁存器的一些实现方式见第2章，图2.9描述了用普通锁存器和控制电路来实现4相捆绑数据握手锁存器，并给出了用多个握手锁存器组成流水线的例子。图2.11给出了2相捆绑数据锁存器的实现，图2.12和图2.13是4相双轨锁存器的实现。

握手锁存器的特性可以通过FIFO的吞吐量、动态波长和静态扩展来描述，该FIFO是由相同的锁存器组成的。上文提到的两个4相锁存器设计的共同点是：FIFO中的锁存器由有效托肯和空托肯交替填满，如图4.1(b)所示，因此，这些FIFO的静态扩展 $S = 2$。

2相握手的实现不需要引入空托肯，因此有可能设计出静态扩展 $S = 1$ 的锁存器。图2.11中的2相捆绑数据握手锁存器的实现使用了电平触发锁存器，但实际上采用电平触发锁存器并不好。

理论上，我们希望能够把每个电平触发的锁存器都装入有效托肯，在第7章中我们将描述如何设计具有小的静态扩展的4相捆绑数据握手锁存器。

5.2　分支、汇合与并入

图5.1描述了用于4相捆绑数据和4相双轨的分支、汇合和并入元件。为简单起见，图中分支元件只有两个输出通道，汇合和并入元件都只有两个输入通道。而且，所有的通道都是1位通道。当然，我们可以将这些元件分别扩展到3个或多个输入和输出，以及 n 位通道。基于下文的描述，可以容易的实现扩展，这作为练习留给读者自己完成。

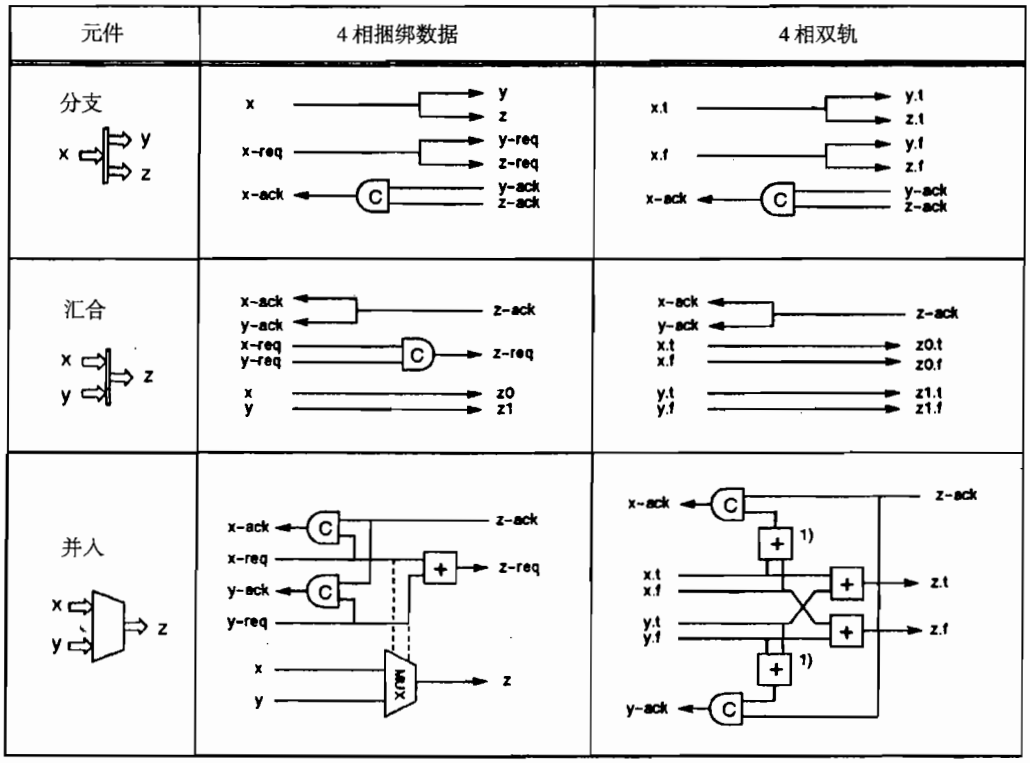


图5.1　分支、汇合与并入组件的4相捆绑数据及4相双轨实现

4相分支和4相汇合。每个分支含有一个C单元，该C单元将输出通道上的多个应答信号合成为输入通道上的单个应答信号。同理，4相捆绑数据汇合也包含一个C单元，该C单元是将输入通道上的多个请求信号合成为输出通道上的单个请求信号。4相双轨汇合中不包含任何主动元件（C单元），因为请求信号被编码到数据中。

图5.1中的特定分支起到复制输入数据的作用，而汇合起连接输入数据的作用。在静态数据流结构中，分支和汇合用得最多的正是这两种作用，但他们还有更多的功能：如分支可以分割输入数据，使它与图5.1中的汇合更具对称性。但无论怎样，两者之间的唯一区别仅仅是数据从输入传送到输出端的方式不同。从控制的角度来看，两者虽有不同，但本质是一样的：汇合使多个输入通道同步而分支使多个输出通道同步。

4相并入。并入的实现则需要更详细的说明。输入通道的握手是互斥的，而并入则是把主动的输入握手简单地传送到输出通道。

首先讨论4相捆绑数据并入的实现问题。该并入含有一个异步控制电路和一个由输入请求信号控制的多路选择器，以下将对控制电路进行介绍。

输入通道的请求信号之间是互斥的，可以将这些请求信号“或”在一起，从而在输出通道上产生一个请求信号。

而对每个输入通道，如果输入通道上有有效数据，C单元就会产生一个应答信号来响应输出通道上的应答信号。例如，当x_{req}和z_{ack}都为高电平时，C单元把x_{ack}信号置成高电平；而当x_{req}和z_{ack}都为低电平时，C单元被复位。当x_{req}变为低电平时，z_{ack}变为低电平来响应x_{req}的变化，这使得C单元复位来响应z_{ack}的变化。如图5.2所示，如果电路中有非对称C单元则可进一步优化。这同样适用于驱动y_{ack}信号的C单元。在第6章的6.4.1节和6.4.5节中将会详细介绍如何实现C单元和相关的状态保持器件。

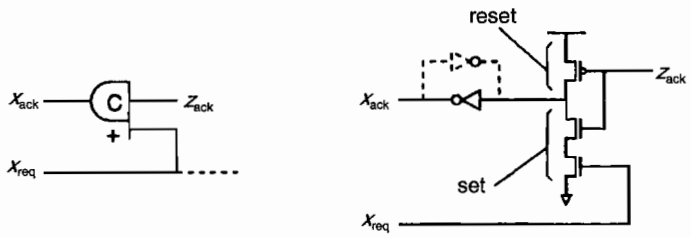


图5.2 图5.1所示的4相捆绑数据并入元件中上部非对称C单元的一种实现

4相双轨并入的实现与4相捆绑数据并入的实现是极其相似的。由于请求信号被编码到数据信号中，电路中两个输出信号$z.t$和$z.f$都用一个或门来实现。如果输入通道上有有效数据，输入通道产生应答信号来响应输出通道上的应答信号。由于例子中假设数据通道的宽度是1位，后者是通过一个或门（标记为1）来实现的，但对于N位宽度的通道则需要探测器（如图2.13所示）。

2相分支、汇合与并入。最后简要介绍一下基于2相捆绑数据的分支、汇合和并入元件：如果信号初始电平为低电平，2相捆绑数据分支和汇合元件的实现与4相捆绑数据分支和汇合元件的实现是相同的。

另一方面，2相捆绑数据并入元件的实现比较复杂且与4相捆绑数据的实现很不相同，但可以用它来说明为什么某些2相捆绑数据元件的实现很复杂。当某个请求或应答信号出现时，可以看到跳变沿交替上升下降，但不清楚输入通道握手的时序，因此输入通道与相应输出通道上的请求信号跳变的极性之间没有联系。同样，输出通道与相应输入通道上的应答信号跳变的极性之间也没有联系。因此，电路产生的各个请求和应答信号需要存储元件，这使电路的实现变得复杂，而且存储元件的控制逻辑也很复杂。

5.3 功能块基础

本节主要介绍功能块设计的基本原理，在后续章节中还将举例说明不同握手协议功能块的实现，使用的例子是一个N位行波进位加法器。

5.3.1 引言

功能块是与组合逻辑电路等效的异步电路：它从一组输入信号中计算得到一个或多个输出信号。采用“功能块”这个术语主要是为了强调纯粹是用功能的行为来处理电路。

然而，功能块除了按指定的功能计算输入信号以外，它与相邻锁存器间的握手也是透明的。对握手的透明，使功能块与组合电路区别开来。下面将看到，对隐含表示完成的功能块（比如双轨信号的电路），“透明”还有更深刻的含义。

最常见的情况是功能块的操作数的通道与操作结果的通道分开，如图5.3所示。使用独立的输入与输出通道，意味着在输入端有汇合，而在输出端有分支。它们可以单独实现（如前面介绍的），也可以集成到功能块电路中。在后面的章节中，我们假定，功能块的所有操作数都有各自单独的通道，而所有结果都存放到另外的单独通道中。

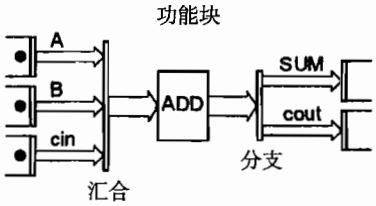


图5.3 操作数与操作结果通道分开的功能块，输入通道采用汇合结构，输出通道采用分支结构

为了给不同方法实现的案例提供必要的背景知识，我们首先讨论握手透明度的问题，然后回顾行波进位加法器的基本原理。文献[97]是关于功能块设计的一篇好文章。

5.3.2 握手的透明度

握手透明度的基本概念可通过一个4相双轨例子来说明，捆绑数据协议的功能块可理解成是它的一个特例。图5.4(a)是两个直接连在一起的握手锁存器，图5.4(b)在这两个握手锁存器中间插入了功能块，此时该功能块对握手必须是透明的。这意味着如果观察锁存器端口上的信号，则可以看到同样的握手信号跳变时序，只是由于功能块的延迟，可能使某些信号有所滞后。

功能块输出端的请求信号不能在输入端接收到请求信号之前产生。相反，功能块输出端的请求信号应当指示所有的输入信号都是有效的，并且所有有关的内部信号和输出信号都已经计算完成（这里再次提到了有关指示的原理）。在4相协议中，在握手归零部分有许多对称的需求。

根据握手的透明度，功能块可以分为强指示类（strongly indicating）和弱指示类（weakly indicating）。图5.4(a)中两个锁存器之间的通道上的信号变化情况是可被观察的，这在图2.3中已讨论过。我们以同样的方式来分析图5.4(b)中的握手情况。

- 对于强指示功能块，如图5.5所示，那么(1) 在其所有输入都变为有效后，才开始计算和产生有效输出；(2) 在其所有输入都变为空后，才产生空值的输出。
- 对于弱指示功能块，如图5.6所示，那么(1) 会尽可能快地开始计算和得出有效输出，并不是等所有的输入信号变为有效后才开始计算和输出；(2) 会尽可能快地生成空值输出，并不是等所有的输入信号变为空值后才输出。

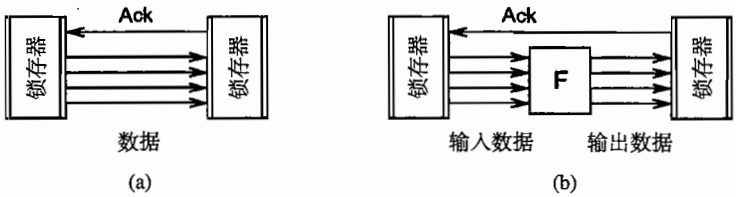
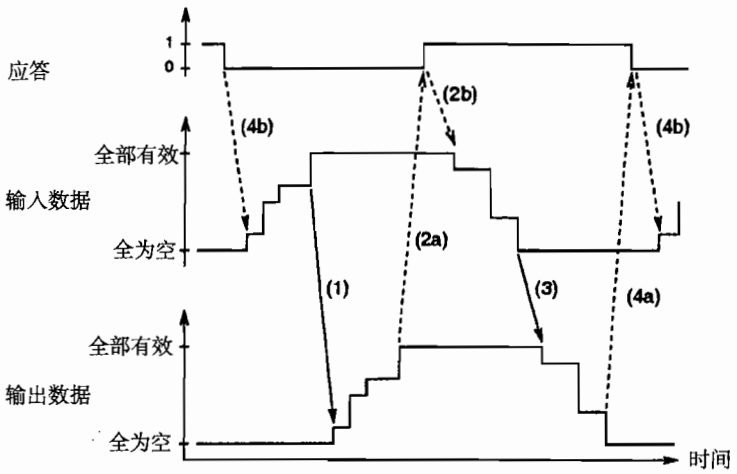


图 5.4 (a) 两个直接连接在一起的握手锁存器; (b) 两个握手锁存器中间插入功能块的情形，两种情况下的握手被视为是相同的，即功能块对握手是透明的

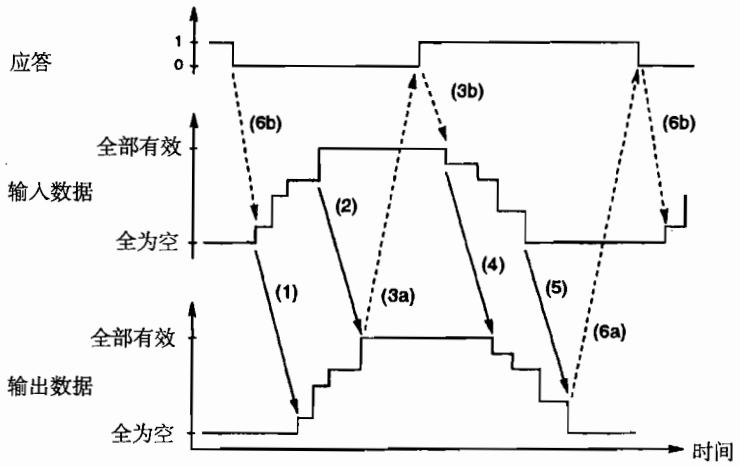


(1) 所有输入被定义 ⇒ 部分输出被定义
(2) 所有输出被定义 ⇒ 部分输入未被定义
(3) 所有输入未被定义 ⇒ 部分输出未被定义
(4) 所有输出未被定义 ⇒ 部分输入被定义

图 5.5 强指示功能块的信号轨迹与事件顺序

要使弱指示功能块能够正确地工作，它还必须满足以下条件：只有当所有输入都变为有效后，它才能生成所有的有效输出，并且只有在所有输入都变为空值后，它才能生成所有的空值输出。这一行为与Seitz在文献[121]中所提出的弱条件是等价的。在文献[121]中，Seitz进一步

证明：如果单个功能块满足弱条件，则任何由功能块组成的“有效组合电路结构”也满足弱条件，即功能块能够组合成更大的功能块。“有效组合电路结构”是指结构中没有元件的输入或输出端口是未连接的，且没有反馈路径。强指示功能块具有同样的性质——由强指示功能块构成的“有效组合电路结构”也是一个强指示功能块。



(1) 部分输入被定义 ⇒ 部分输出被定义
(2) 所有输入被定义 ⇒ 所有输出被定义
(3) 所有输出被定义 ⇒ 部分输入未被定义
(4) 部分输入未被定义 ⇒ 部分输出未被定义
(5) 所有输入未被定义 ⇒ 所有输出未被定义
(6) 所有输出未被定义 ⇒ 部分输入被定义

图5.6　弱指示功能块的信号轨迹和事件顺序

可以看到无论是弱指示功能块还是强指示功能块，它们在有效-空和空-有效跳变过程中都表现出了一种滞后行为：(1) 在某些/全部输入都变为空值前，某些/全部的输出必须保持有效；(2) 在某些/所有输入都变为有效值前，某些/所有输出必须保持为空值。正是这种滞后特性保证了握手的透明度，且需要一个或多个状态保持电路（通常以C单元的形式）来实现这种滞后特性。

最后，简要介绍一下4相捆绑数据协议。由于Req↑等价于“所有数据信号都有效”，而Req↓等价于“所有数据信号都是空值”，所以4相捆绑数据功能块可以归为强指示类型。

在后面我们将会知道强指示功能块具有最坏延时。为了能够获得实际情况的延时，必须使用弱指示功能块。在考虑不同握手协议的功能块的实现方式之前，先介绍二进制行波进位加法器的基本知识，这个例子将会贯穿于后面的章节。

5.3.3 行波进位加法器

图 5.7 所示为行波进位加法器的实现原理。1 位全加器的实现表达式是

$$s = a \oplus b \oplus c \tag{5.1}$$

$$d = ab + ac + bc \tag{5.2}$$

在很多实现方式中，用一些中间信号来表示输入 a 和 b：

$$p = a \oplus b \quad (\text{“传输”进位}) \tag{5.3}$$

$$g = ab \quad (\text{“发生”进位}) \tag{5.4}$$

$$k = \bar{a}\,\bar{b} \quad (\text{“清除”进位}) \tag{5.5}$$

输出信号的表达式是

$$s = p \oplus c \tag{5.6}$$

$$d = g + pc \quad \text{或} \tag{5.7a}$$

$$\bar{d} = k + p\bar{c} \tag{5.7b}$$

对于行波进位加法器，最坏情况下的关键路径是指贯穿于整个加法器的进位链。如果一个 1 位全加器的延时是 t_{add}，则一个 N 位全加器最坏情况下的延时是 $N \cdot t_{add}$。但是这种情况很少出现，通常情况下，运算过程中的最长进位传输时间比它要短得多。考虑到随机的和不相关的操作数，平均延时是 $\lg(N) \cdot t_{add}$，并且如果数值小的操作数出现的次数越多，则平均延时越小。如果采用的是普通的布尔信号（如在捆绑数据协议中使用的信号），则无法得知运算完成的时刻，所以此时的性能只能是最坏情况。

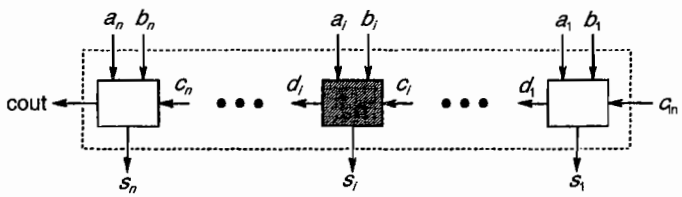


图 5.7 行波进位加法器，本级的输出 d_i 连到下一级的输入 c_{i+1}

而采用双轨进位信号($d.t$, $d.f$)，则有可能设计出指示某一部分运算已完成的电路，从而使电路获得实际情况的延时。关键是双轨进位信号 d 能传送以下三种信息之一：

$(d.t, d.f) = (0, 0)$ = Empty（空）	进位还未计算出来，可能依赖进位 c
$(d.t, d.f) = (1, 0)$ = True（真）	进位为 1
$(d.t, d.f) = (0, 1)$ = False（假）	进位为 0

因此，对于一个 1 位加法器，如果它的输入为（$a = b = 0$ 或 $a = b = 1$），则这个加法器可以在输入进位有效之前就输出一个有效进位。这种思想最早是由 Gilchrist 于 1995 年提出的[52]，在文献[62]的 75~78 页和文献[121]中也介绍了这种思想。

5.4　捆绑数据功能块

5.4.1　采用匹配延时

图 5.7 中的加法器的捆绑数据实现如图 5.8 所示。它是由传统的组合电路加法器和一个匹配延时元件构成的。延时元件提供一个恒定的延时来匹配组合加法器的最坏延时，它包括电路中最坏情况的关键路径（某个贯穿整个加法器的进位链）和最坏的运算条件。为了使电路可靠运行，需要一些安全裕度。

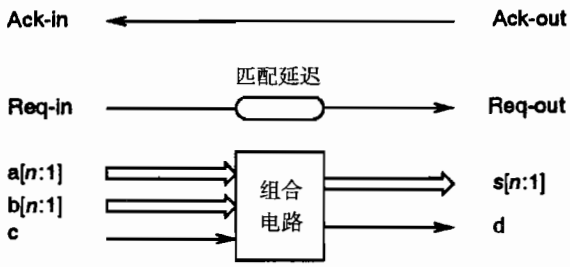


图 5.8　图 5.7 中 N 位握手加法器的 4 相捆绑数据实现

除了组合电路本身的设计以外，对于电路设计而言，延时元件是一个挑战，主要有以下几点原因：首先，延时元件要追踪制造过程的偏差以及温度和电源电压的变化所带来的延时变化。另一方面，连线延时非常重要但却难以控制。显然，对于延时匹配还需要制定一些相关的设计策略。在全定制设计环境中，设计者可能会采用具有相同版图但晶体管不工作的伪电路。在标准化单元的自动布局布线环境中，设计者不得不采用大的安全裕量，或者做布局后时序分析以及延时调整。后面这个步骤可能会很复杂，类似于同步设计中布局布线后对建立和保持时间的检查以及延时调整。

在 4 相捆绑数据设计中，从电路性能方面来看，为了使握手中的归零部分执行的尽可能地快，更倾向于采用非对称的延时元件。另外一个需要考虑的问题就是延时元件的功耗。在文献[23]中提到的一个 ARISC 处理器设计中，延时元件的功耗占到总功耗的 10%。

5.4.2 延时选择

在文献[105]中，Nowick 提出了一种称为“推测完成”（speculative completion）的方法，图 5.9 说明了其基本原理。电路中除了实现期望功能的电路模块以外，还加入了辅助电路，用来从多个匹配延时中选择合适的延时。对延时的估计要保守，也就是要保证电路可靠。电路中的输入信号与 / 或某些实现期望功能需要的内部信号是估计的基础。

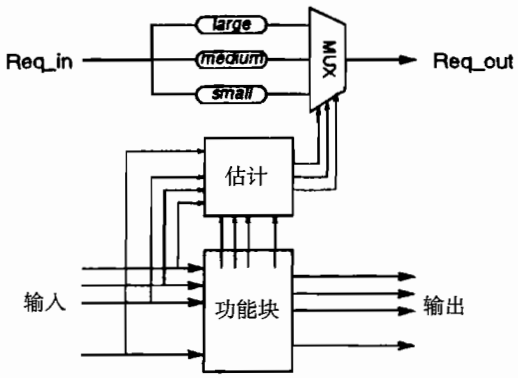


图 5.9 推测完成的基本思想

对于一个 N 位行波进位加法器，可用构成 1 位全加器（见图 5.7）的传输信号（见式(5.3)）来进行估计。现在以 16 位加法器为例：如果 $p_8 = 0$，则电路中最长的进位行波不会超过 8 级；如果 $p_{12} \wedge p_8 \wedge p_4 = 0$，则最长进位行波不会超过 4 级。按照这种简单的估计，可以选择一个足够大的匹配延时。此外，如果采用 4 相协议，从性能角度选择非对称延时元件更合适。

设计者需要在安全裕度和电路的面积与功耗之间做权衡，大的安全裕度将增加电路的面积和功耗。如果读者想了解更多的有关实现与所获性能改善的详细资料，请参考文献[105, 107]。

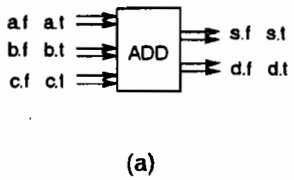
5.5 双轨功能块

5.5.1 延时无关最小项综合（DIMS）

在第 2 章中我们已对用于双轨信号的与门进行了讨论（见图 2.14）。采用相同的电路结构，可以实现其他的基本门，如 OR 门、EXOR 门等。一个反相器只要把两条线交换，而不涉及其它电路。

任意逻辑功能可通过把门组合在一起来实现，其设计方法与同步电路中的组合电路的设计完全一样。由于握手被隐含的考虑了，在组成门电路和实现逻辑功能时可以不考虑握手。它可以采用现有的逻辑综合技术与工具来进行设计，唯一的不同是基本门的实现方式不一样。

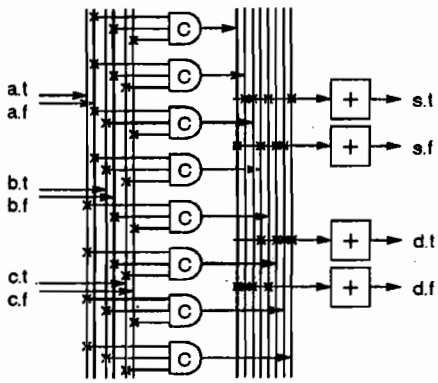
图2.14中的双轨与门显然是相当低效的：4个C单元和1个OR门总共大约需要30个晶体管，这是普通与门（只需要6个晶体管）的5倍。在大功能块的实现中可以降低这种开销。为了进一步说明，图 5.10(b)、(c)是 1 位全加器的实现，稍后将讨论图 5.10(d)所示的电路。



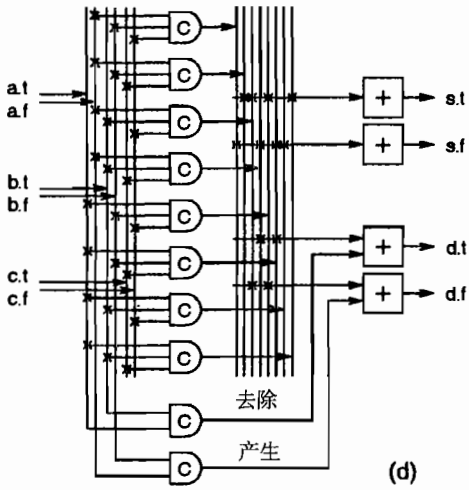
(a)

a	b	c	s.t	s.f	d.t	d.f
E	E	E	0	0	0	0
			不变化			
去除 F	F	F	0	1	0	1
F	F	T	1	0	0	1
F	T	F	1	0	0	1
F	T	T	0	1	1	0
T	F	F	1	0	0	1
T	F	T	0	1	1	0
产生 T	T	F	0	1	1	0
T	T	T	1	0	1	0

(b)



(c)



(d)

图 5.10　4 相双轨全加器：(a) 符号；(b) 真值表；(c) DIMS 实现；(d) 全加器弱指示优化

图 5.10(c)中类似于 PLA 结构的电路，可用来说明实现任意逻辑功能的一般原则。由于这种电路是延时不敏感的，而且 C 单元产生输入信号的所有最小项，在文献[124]中称这种方法为延时不敏感最小项综合（Delay-Insensitive Minterm Synthesis , DIMS）。真值表中的输出行分为 3 类情况：(1) 输入全为空码，电路的相应输出也是空；(2) 输入为一中间码，输出不变；(3) 输入为有效码，电路相应输出为合适的有效值。

上述基本思想可追溯到David Muller在20世纪50年代末、60年代初所做的工作[92, 93]。而文献[93]发展了速度无关电路设计的基本理论，文献[92]介绍了更为实际的设计实例：用上例中介绍的锁存器和门实现了位串行乘法器。

参考 5.3.2 节，这里所谈的 DIMS 电路可以归为强指示电路，因此它呈现的是最坏延时。N位行波进位加法器的空→有效和有效→空的转变将会按严格的顺序从最低位依次传递到最高位。

如果改变一下全加器的设计，如图5.10(d)所示，那么可能在输入的c信号为有效之前就产生有效的d，此时，由这种全加器构成的N位行波进位加法器会呈现出实际的延时——电路是弱指示功能块。

图5.10(c)和图5.10(d)中用全加器构成的行波进位加法器电路是对称的，因此传播一个空值的延时与传播一个有效值的延时是相同的。这可能并非所期望的。在稍后的 5.5.4 节中我们将会介绍一种更为精巧的设计，此时空值的传输时间是恒定的（两个全加器单元的延时）。

5.5.2 零协议逻辑

前面所介绍的C单元和OR门可视为具有滞后作用的n-of-n和1-of-n阈门，如图 5.11 所示。Theseus Logic公司[39]提出了通过采用具有滞后作用的任意m-of-n阈门来降低实现的复杂性。对于具有滞后作用的m-of-n阈门，当它的任意m个输入变为高电平时，它的输出将置为高电平；而当所有的输入都为低电平时，它的输出置为低电平。这种电路实现的思想是Theseus Logic公司的零协议逻辑（Null Convention Logic, NCL）的核心部分。但是在更高层的设计中，NCL 和第 3 章介绍的数据流观点基本上没有区别，而且NCL与文献[92, 122, 124, 97]中提出的那些电路设计方法非常相似。图 5.11 表明，OR门和C单元可视为阈门的特殊情况。门符号中的数字是这个门的阈值。图 5.12 给出的是一个由 NCL 阈门实现的双轨全加器，这个电路是弱指示的。

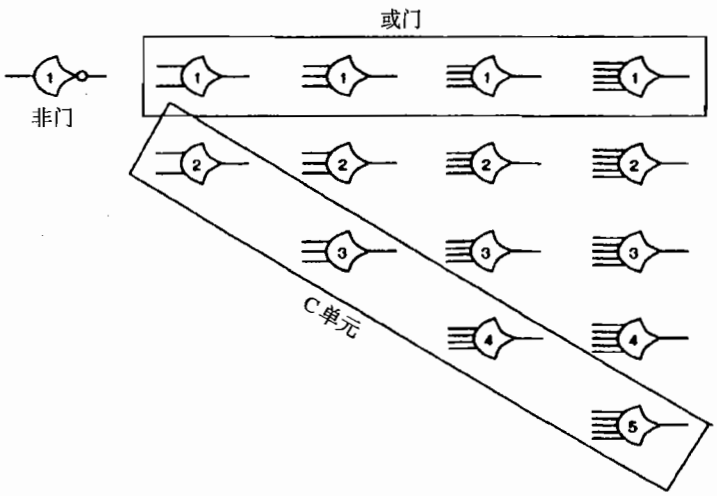


图 5.11 NCL门：带有滞后的m-of-n阈门

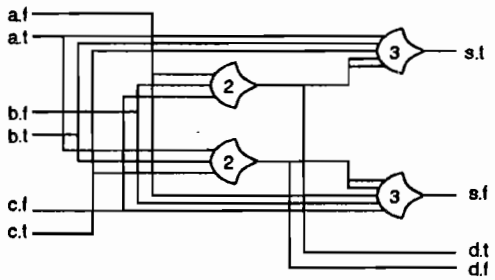


图 5.12 使用NCL门的全加器

5.5.3 晶体管级 CMOS 实现

最后将要介绍的两个加法器的设计是基于双轨信号的CMOS晶体管级实现。双轨信号是由预充电差动逻辑电路产生的，而预充电差动逻辑电路常用于存储结构和像DCVSL那样的逻辑系列，如图5.13所示[151, 55]。

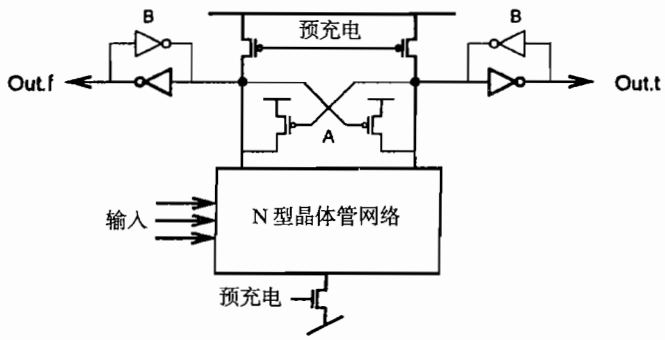


图 5.13 预充电差动CMOS组合电路，通过增加交叉耦合的P型晶体管“A”和（弱）反馈反相器“B”，使电路成为（伪）静态

在捆绑数据设计中，功能块输入通道的请求信号可以作为预充电信号。在双轨设计中，可以用检查所有输入是否为空的晶体管网络来替代预充电P管。类似地，只有当所有的请求输入都有效时，用来下拉的N管信号路径才导通。

DIMS和NCL门的晶体管实现很简单。图5.14是强指示全加器进位电路的晶体管级实现。在下拉电路中，每列晶体管对应于一个最小项。在实现DCVSL门时，两个下拉网络可以共享晶体管，但为了更好地说明晶体管实现和图5.10(c)中的门级实现之间的关系，这里没有共享晶体管。

不希望P管有过多的堆叠。它们可用单个晶体管替代，该晶体管可由其他电路产生的“全空”信号来控制。下一节将给出弱指示全加器设计，包括通过优化来减少P管的堆叠。

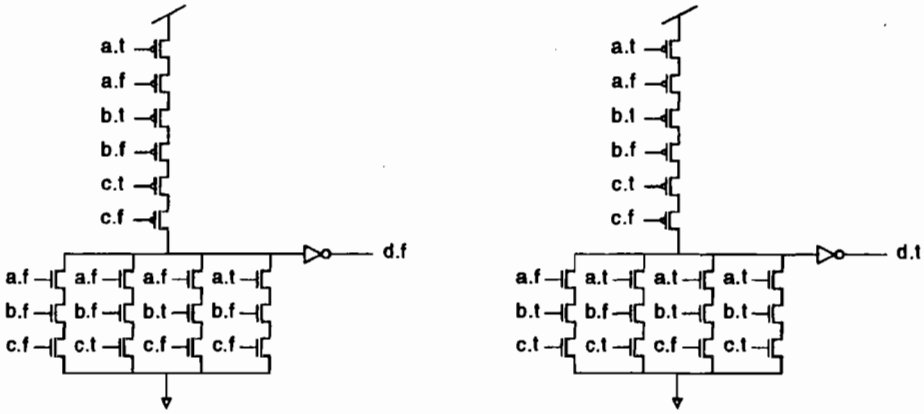


图 5.14 图 5.10(c)中的强指示全加器进位信号的晶体管级实现

5.5.4 Martin 加法器

在文献[85]中，Martin 提到一般双轨功能块的设计。他通过一个双轨行波进位加法器来说明他的思想。这个加法器用的晶体管数量很少，当对有效数据进行加法运算时呈现的是实际延时，并且它传输空值的时间是一个常数——这个加法器代表了弱指示功能块设计的极致。

在图 5.14 所示的强指示晶体管级进位电路中，d 在 a, b 和 c 变为空值之前都保持有效。如果设计一个类似的加法电路，电路的输出 s 在 a, b 和 c 变为空值之前同样保持有效。图 5.6 中的弱条件只要求在所有的输入都变为无效之前至少有一个输出仍然保持有效。从而允许在进位和累加电路中分出 a, b 和 c 为空的指示。

在文献[85]中，Martin 采用了很直观的有向图来说明输出信号是如何指示输入信号和内部信号为有效值或空值的。图中节点代表了电路中的信号，有向边表示的是信号之间的指示依赖关系。实线边表示的是有保证的依赖关系，而虚线边表示的是可能的依赖关系。图 5.15(a)所示的是由 3 个全加器组成的行波进位加法器，图 5.15(b)和图 5.15(c)分别表示有效和空值输入信号是如何在电路中传递的。

有效值的传递和指示与前面讨论的其他加法器类似，但空值的传递和指示有所不同而且呈现出恒定延时。当输出 d_3, s_3, s_2 和 s_1 都有效时，则指示此时所有的输入信号和所有的内部进位信号都有效。同样，当输出 d_3, s_3, s_2 和 s_1 都为空值时，则指示所有的输入信号和所有的内部进位信号都为空值——行波进位加法器满足弱指示条件。

图 5.16 所示是该全加器的晶体管实现，它用了 34 个晶体管，所需晶体管的数量与用传统的组合电路实现的晶体管数量相当。

上述原理适用于一般的功能块设计。图 5.15 所示的“有效/空指示（或应答），相关图”对于理解和设计低延时及最弱指示电路非常有用。

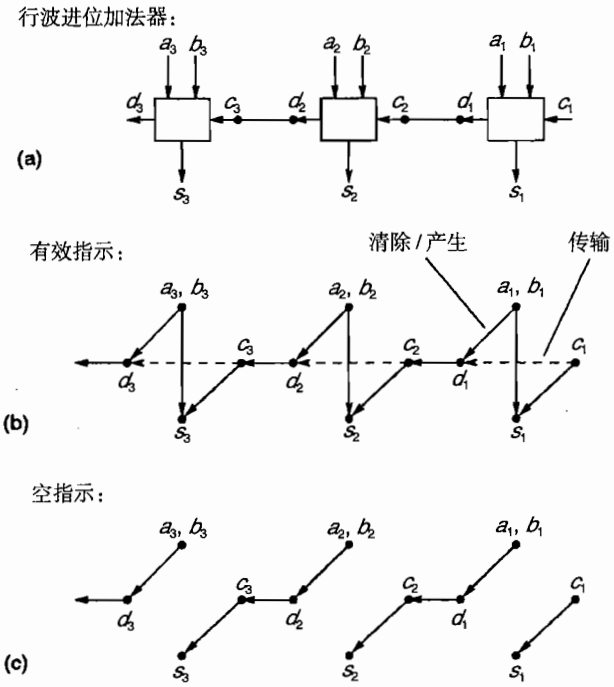


图 5.15 (a) 3 阶行波进位加法器；(b) 空数据；(c) 数值的传递[85]

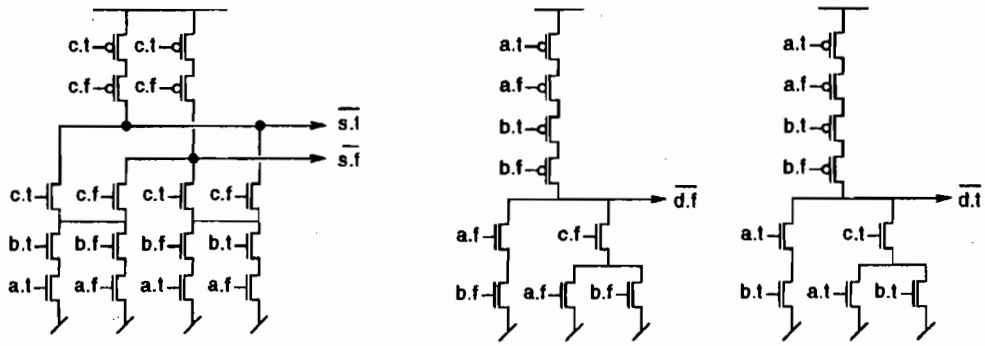


图 5.16 Martin 加法器的 CMOS 晶体管实现[85]

5.6 混合功能块

我们最后给出的这个加法器，具有4相捆绑数据输入和输出通道以及一个双轨进位链。这种设计与上一节给出的Martin双轨加法器类似：当传递有效数据时，电路是实际的延时；而当传递空值时，电路是恒定延时，且电路实现所需要的晶体管数量适中。这个混合加法器的基本结构如图 5.17 所示。每个全加器都是由进位电路和累加电路组成的。图 5.18(a)和图 5.18(b)给

出了进位电路和累加电路的预充电CMOS实现。其主要思路是：当$Req_{in}=0$时，电路预充电；当$Req_{in}=1$时，电路计算；检测所有的进位信号是否都为有效，并用检测信息指示完成，即$Req_{out}\uparrow$。如果全加器中完成检测电路的延时小于累加器电路的延时，则还需要一个匹配延时元件，如图5.17所示。

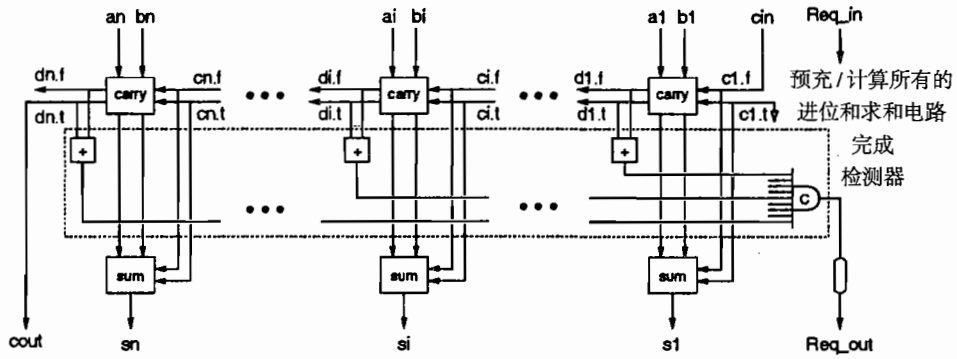


图5.17 带4相捆绑数据输入、输出通道和内部双轨进位链的混合加法器的框图

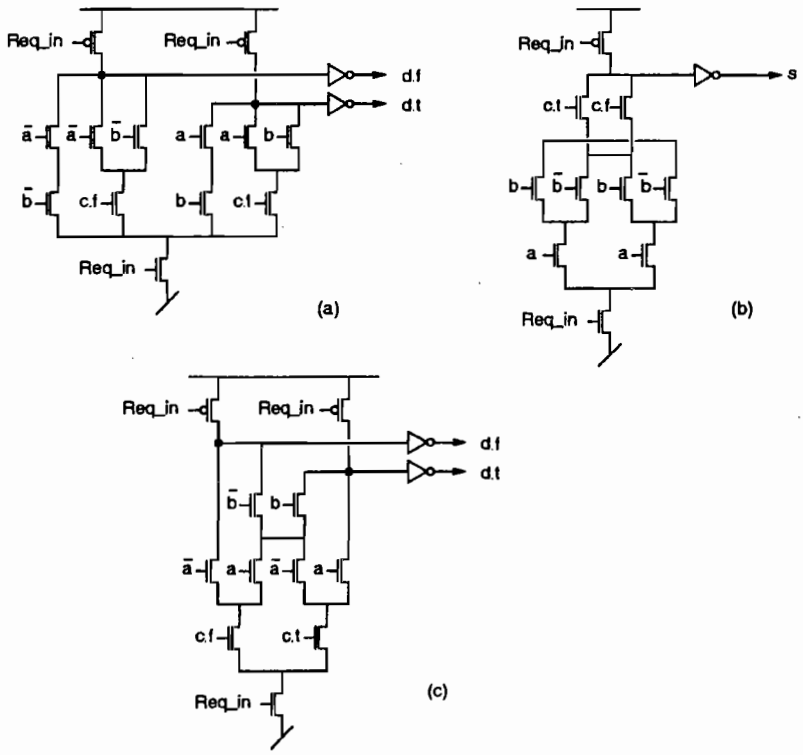


图5.18 图5.17中的混合全加器的晶体管级实现：(a) 弱指示进位电路；(b) 相加电路；(c) 强指示进位电路

图5.17中所示的完成检测器的大小和延时随加法器的位数的增加而增加，在位数较多的加法器中完成检测器的延时可能要大大超过累加电路的延时。可以用弱、强指示混合的功能块[101]来降低完成检测器的开销，但会增加电路的延时(Req_{in}↑到Req_{out}↑)。根据图5.7中确定的命名方法，使加法器1, 4, 7, …为弱指示电路而其他的加法器为强指示电路。此时，只需要对第3, 6, 9,…级送出的进位信号进行完成检测。因为$i = 3, 6, 9, \cdots$, d_i能够指示d_{i-1}和d_{i-2}的完成情况。当然还有其他强、弱指示混合加法器的方案，例如，在文献[101]中提出了一种特殊方案，它利用了（音频数据的）小数值的典型操作数，并且用单个进位信号来检测完成。

功能块设计小结

前面几节介绍了实现功能块的基本原理，并通过一系列的行波进位加法器来进行说明。其要点是“握手的透明度”，以及通过弱指示元件来实现“实际情况延时”。

最后，强调一下要注意的地方：前面所提到的行波进位加法器，在某种程度上过分地强调了平均性能的优点，容易使人们过于追求精巧的电路设计，但这并非跟系统级设计特别有关：

- 在很多系统中，行波进位加法器的最坏情况延时可能是不可接受的。
- 如果系统中存在多个用于同步和高速交换数据的并发主动元件，任意时刻系统的性能往往是由最慢元件来决定的；因此系统的平均性能可能并不会比单个元件的平均延时性能好。
- 在很多情况下，加法运算只是更加复杂的复合算术运算中的一部分。例如，在文献[103]中提到的异步滤波组的最终设计并没有采用上面介绍的那些思想。而是完全采用强指示全加器，因为它能够实现一个有效的二维预充电的复合加-乘累加器单元。

5.7　多路选择器和多路分配器

前面介绍了功能块设计的原理，下面准备讨论图3.3所示的多路选择器和多路分配器的实现问题。首先概括一下它们的功能：多路选择器将控制通道同步，并把选定输入通道的数据和握手信号传送到输出通道，而其他的输入通道不起作用（或许有请求在等待）。类似地，多路分配器将控制通道与数据输入通道同步，并且把输入数据传送到选定的输出通道，而其他的输出通道处于被动与空闲状态。

如果只考虑那些“活动”的通道，则可以把多路选择器和多路分配器理解为功能块并且可以按照功能块设计方法来设计——必须像功能块那样对握手是透明的。控制通道和选定的输入

数据通道首先连接在一起，然后产生输出。由于只有在控制通道和选定的输入数据通道都处于活动状态时才有数据传送，因此它们属于强指示功能块。

现在考虑用4相协议来实现多路选择器和多路分配器。最简单直观的设计是采用双轨控制通道。图5.19给出了多路选择器和多路分配器的实现，输入和输出数据通道采用的是4相捆绑数据协议，而控制通道采用的是4相双轨协议。在这两个电路中，ctl.t和ctl.f可理解为两个互斥的请求信号，用来选择两个输入中的一个输出，ctl.t和ctl.f与两种输入请求汇合在一起（在标有“汇合”的C单元中）。多路选择器其余部分的实现类似于图5.1中所示的4相捆绑数据并入的实现。多路分配器其余部分不用特别说明，两个输出端口上的握手是互斥的，而应答信号 y_{ack} 与 z_{ack} 通过或门输出，形成 $x_{ack} = ctl_{ack}$。

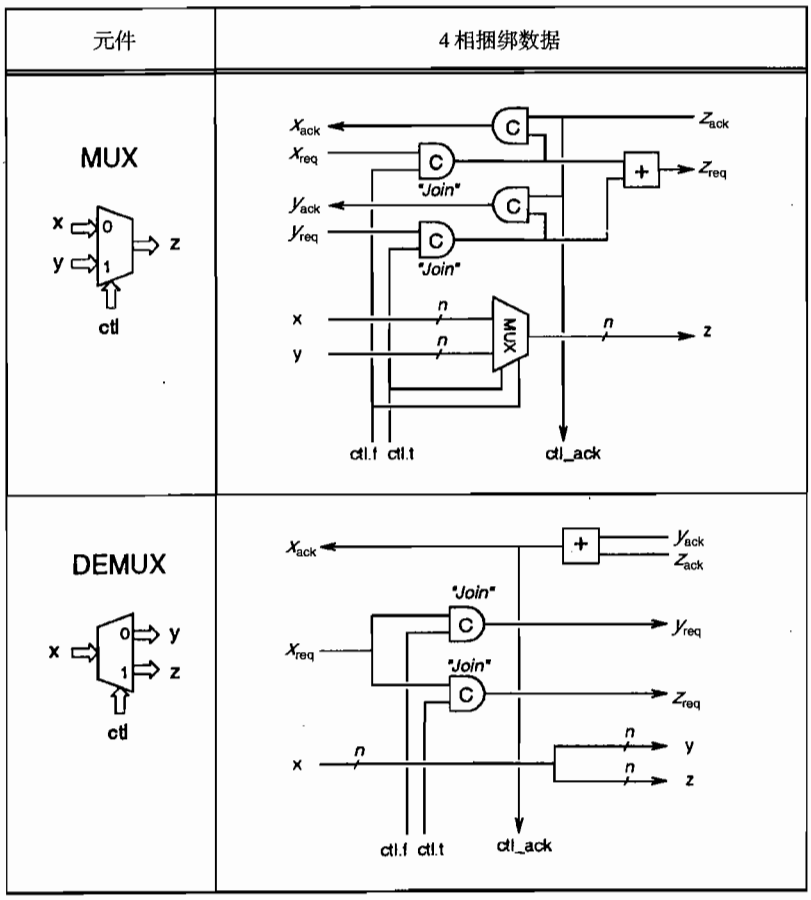


图5.19 MUX和DEMUX的实现，输入、输出通道 x, y, z 使用4相捆绑数据协议，控制通道ctl使用4相双轨协议（为了简化设计）

多路选择器和多路分配器的所有4相双轨实现都非常类似，通过将控制通道上的4相双轨协议改为4相捆绑数据协议，可以获得4相捆绑数据的实现。在第6章的结尾部分，将用一个完全4相捆绑数据多路选择器来说明速度无关控制电路的设计。

5.8　互斥、仲裁和亚稳定性

5.8.1　互斥

某些握手元件（包括并入）都要求几个输入通道之间的通信是互斥的。前面讨论的简单静态数据流电路结构就是这种情况，但是可能会遇到多个独立的部分/进程共享一个资源的情况。

用于处理资源共享问题的基本电路称为互斥器（MUTEX），如图5.20所示（稍后介绍它的具体实现）。输入信号R_1和R_2是两个独立源的请求信号，而互斥器的任务是要把这两个输入传递到相应的输出G_1和G_2，而且要保证任意时刻最多只有一个输出是活动的。如果只有一个输入请求，则互斥器无需操作。如果输入请求是在另一个输入请求之前到来，则后到的请求一直处于被阻塞状态直到第一个请求被释放(de-assert)。但是当两个输入请求同时到达时就会出现问题，此时互斥器要做出仲裁判决，这里将出现亚稳定性（metastability）现象。

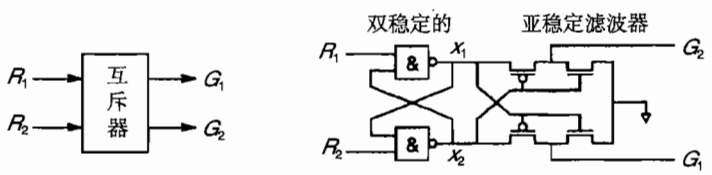


图5.20　互斥器、符号及可能的实现方式

当在同步电路中加入异步输入信号时，由于异步输入信号不满足所要求的建立与保持时间，也存在这个问题。用时钟控制的触发器对异步输入信号进行同步时，问题的关键是数据信号是否会在时钟有效边沿的前后发生变化。对于互斥器，问题的关键是确定哪个信号首先发生跳变，如果数据信号的跳变和时钟有效边沿跳变同时发生，则要求互斥器随机决定。

互斥器和同步触发器的根本问题都是如何处理双稳态电路，这个双稳态电路在同一时刻接收到两个要求触发器进入不同稳定状态的请求信号。这将引起电路进入亚稳定状态。在亚稳定状态中，电路要经过一段时间才会随机地在两个状态之间做出选择。同步化问题在很多有关数字设计和VLSI的书籍中都会介绍，而那些书中对亚稳定性的分析同样适合于互斥器。相关的参考文献有：文献[95]中的9.4节，文献[53]中的5.4节和6.5节，文献[151]中的5.5.7节，文献[115]中的6.2.2节和9.4.5节，文献[150]中的8.9节。

对于同步设计者，问题是亚稳定状态的持续时间可能会超出系统留给亚稳定状态的恢复时间，这只是在规定的时间内得不到所需的判决。而对于异步设计者来说，最终会得到所需的判决，但用于等待答案的时间并没有一个确定的上限。在文献[22]中引入了术语——“时间安全”（time safe）和“数值安全”（value safe），来对这两种情况进行表示和分类。

图5.20给出了互斥器的一种电路实现，它由交叉耦合与非门和亚稳态滤波器组成。交叉耦合与非门可使先到的输入阻塞后到的输入。如果这两个输入同时到来，则电路会进入亚稳定状态，即信号x_1和x_2的电压处于电源电压与地之间。亚稳态滤波器会阻止不确定信号传输到输出端，G_1和G_2会一直保持低电平直到信号x_1和x_2之间的差值大于晶体管的阈值电压。

图5.20中是亚稳态滤波器的CMOS晶体管级实现[83]，文献[121]采用NMOS实现的，也可以采用门电路来实现：亚稳态滤波器可以用两个缓存器来实现，通过调整上拉和下拉晶体管通路的逻辑强度（strengths），使这两个缓存器的逻辑阈值电压变得特别大（或特别小）[151]。例如，可以把4输入与非门的所有输入连接在一起去实现逻辑阈值电压特别大的缓冲器。文献[6, 139]在互斥器的实现中应用了这一思想。

5.8.2 仲裁

互斥器可用来构造握手仲裁器（Arbiter），握手仲裁器主要用于控制对多个独立部分所共享资源的访问。图5.21给出了一种实现方案。

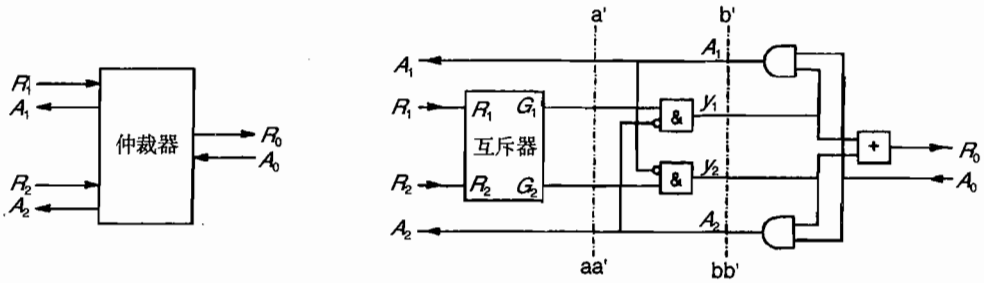


图5.21 握手仲裁器符号及可能的实现方式

互斥器能确保信号G_1和G_2在a' – aa'界面是互斥的，在互斥器之后的是两个与门，它们保证了(y_1, A_1)和(y_2, A_2)通道在b' – bb'界面上的握手也是互斥的：当A_1为低电平时，y_2只能为高电平；而当信号A_2为低电平时，y_1只能为高电平。因此，某个通道中进行的握手会阻塞另外通道中握手的发生。由于握手在通道(y_1, A_1)和通道(y_2, A_2)上是互斥的，则仲裁器的其余部分就是简单的并入，如图5.1所示。如果要把数据传送到共享资源内，则需要一个同并入一样的多路选择器，多路选择器可以由信号y_1与/或y_2控制。

5.8.3　亚稳定性的概率

最后，对亚稳定性进行定量分析：假设$P(\text{met}_t)$表示在一个周期时间t内或更长时间内（在一秒的观测时间间隔内）互斥器处于亚稳定状态的概率，如果这种情况被认为是故障情况，则可以计算平均故障间隔时间：

$$\text{MTBE} = \frac{1}{P(\text{met}_t)} \tag{5.8}$$

$P(\text{met}_t)$的概率计算如下：

$$P(\text{met}_t) = P(\text{met}_t|\text{met}_{t=0}) \cdot P(\text{met}_{t=0}) \tag{5.9}$$

式中，

- $P(\text{met}_t|\text{met}_{t=0})$是指互斥器在$t=0$时刻为亚稳定状态的条件下，在$t$时刻仍处于亚稳定状态的概率。
- $P(\text{met}_{t=0})$是指在给定观测时间间隔内，互斥器将进入亚稳定状态的概率。

概率$P(\text{met}_{t=0})$可计算如下：如果输入R_1和R_2同时变化（即小时间间隔Δ内），则互斥器将进入亚稳定状态。假设这两个输入相互独立，且它们的平均跳转频率分别是f_{R_1}和f_{R_2}，则

$$P(\text{met}_{t=0}) = \frac{1}{\Delta \cdot f_{R1} \cdot f_{R2}} \tag{5.10}$$

上式可理解为：在1秒的观测时间间隔内，输入信号R_1和R_2同时发生变化的概率为$1/(f_{R_1} \cdot f_{R_2})$，互斥器在持续时间$\Delta$内可能进入亚稳定状态。

$P(\text{met}_t)$的概率计算如下：

$$P(\text{met}_t|\text{met}_{t=0}) = \text{e}^{-t/\tau} \tag{5.11}$$

式中，τ表示互斥器从亚稳定状态自发退出的能力。该式可用两种不同的方式解释，其正确性已通过实验加以验证。一种解释是交叉耦合的两个与非门不能记忆它们已经处于亚稳定状态的时间，因此唯一可能的分布是“无记忆的”指数分布。另外一种解释是在亚稳态点上交叉耦合的与非门的小信号模型有一个主极点。

联立式(5.8)至式(5.11)得

$$\text{MTBF} = \frac{\text{e}^{t/\tau}}{\Delta \cdot f_{R1} \cdot f_{R2}} \tag{5.12}$$

实验与仿真已经证明：在t的值不太小时，该式非常准确，并且可以通过实验或仿真来确定Δ和τ两个参数的值。用0.25 μm CMOS工艺加工实现设计良好的电路时，Δ和τ这两个参数的典型值是$\Delta = 30$ ps，$\tau = 25$ ps。

5.9 小结

本章介绍了各种握手元件的实现：锁存器、分支、汇合、并入、功能块、多路选择器、多路分配器、互斥器和仲裁器，主要是实现功能块的原理与方法。

第6章　速度无关控制电路

本章介绍了异步时序电路的设计，详细阐述了一种成熟的设计规范和综合方法：从信号转换图规范综合速度无关控制电路。

6.1　引言

目前已经提出许多有关异步控制电路设计的形式化方法和理论（如时序电路或状态机）。多数方法都包含以下几点：(a) 不同的规范形式；(b) 门和连线延迟模型的不同假设；(c) 电路和外部环境相互关系的不同假设。要包含上述所有内容，则会远远超出本书的范围，因此我们只介绍各种设计方法的一些基本假设和特性，并给出一些相关的参考资料，最后详细介绍用信号转换图设计速度无关电路的方法，它由成熟且公开的设计工具 Petrify 支持。

Myers的著作[95]是进一步学习这些知识的一本好书，它深入地介绍了异步时序电路设计的各种形式、方法和理论，并提供了大量的参考文献。

6.1.1　异步时序电路

图6.1所示是一个普通的同步时序电路和两个异步控制电路。这两个异步控制电路，一个是反馈路径中带有缓冲器（延时元件）的Huffman型基本模式电路，另一个是导线作为反馈路径的Muller型输入输出模式电路。

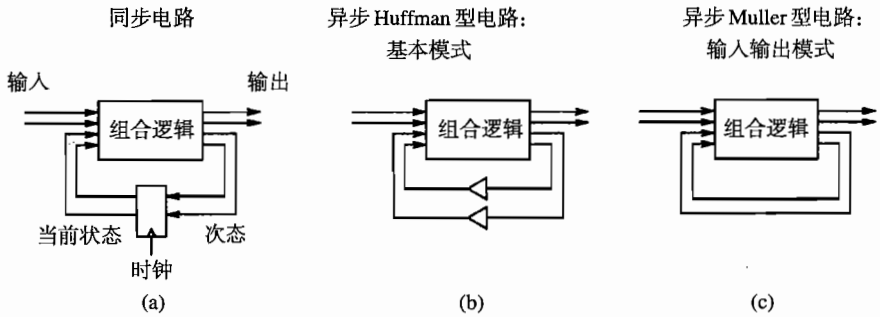


图 6.1　(a) 同步时序电路；(b) 反馈路径中有缓冲器的 Huffman 型异步时序电路；(c) 反馈路径是导线的 Muller 型异步时序电路

同步电路是由一组寄存器和组合逻辑电路组成的，寄存器用来保存当前状态，组合逻辑电路用来计算输出和次态。当时钟到来时，电路的次态进入寄存器中并成为电路的现态。保证同

步电路能够稳定运行的唯一要求是：在时钟上升沿到来前后的一段时间内，从组合逻辑电路中得到的次态输出信号必须是稳定的。这段时间由寄存器的建立时间和保持时间来决定。在两个时钟沿之间，允许组合逻辑电路产生冒险信号，只是在时钟边沿到来时需要确保信号准备就绪且稳定。

在异步电路中，由于电路中不存在时钟，所以所有的信号必须一直处于有效状态。这就意味着至少那些对外部环境可见的输出信号必须是无冒险的。为了使输出无冒险，通常情况下，必须保证内部信号也是无冒险的，这就使得对异步电路进行综合非常困难。所以，不同的设计者基于不同的假设，提出了许多不同的设计方法。

6.1.2 冒险

对于电路设计者而言，冒险可视为有害的短时脉冲干扰信号。图6.2中给出了可观测到的4种可能的冒险。而处于稳定状态下的电路是不会自发产生冒险的——冒险一般来源于电路的动态运行。它同样与电路中输入信号的动态特性以及门和导线的延时有关。因此，在对电路的冒险进行讨论前，必须明确电路采用的是何种延时模型，以及电路和外部环境的交互基于何种假设模型。冒险的理论深度往往会超出人们的想象。

图 6.2 信号上可观测到的可能冒险

通常假设门电路是有延时的。关于导线延时，我们在2.5.3节中已经讨论过了，并且特别指出了在一个带有分支的连线中，不同的分支具有不同的延时。除了门和导线延时外，还必须确定电路采用哪种延迟模型。

6.1.3 延迟模型

对于纯延时（pure delay），或许我们首先想到的是任何信号波形在时间上向后简单移动。在硬件描述语言VHDL中，这种延迟称为传输延迟（transport delay）。但是，这种传输延迟模型并不是真实的模型，因为这种模型假设门和导线都具有无限的带宽。更为真实的延迟模型是惯性延迟（inertial delay）模型。除了信号波形传输的滞后外，惯性延迟还可以抑制短脉冲。在VHDL中使用惯性延迟还必须确定两个参数："延迟时间"（delay time）和"拒绝时间"（reject time），且短于拒绝时间的脉冲被滤掉了。在VHDL中默认的延迟模型是惯性延迟模型。

根据不同的延迟时间参数值，这两种基本延迟模型可以衍生出许多其他延时模型。最简单的模型是“固定延迟”（fixed delay），这种情况下延迟时间是一个常数。另外一种是“最小-最大延迟”（min-max delay），这种延迟的延迟时间是一个不确定的值，但是它的取值范围是明确的，即 $t_{min} \leqslant t_{delay} \leqslant t_{max}$。一种更保守的模型是无界延迟（unbounded delay）模型，此时的延迟时间大于0，但未知且无界，即 $0 < t_{delay} < \infty$。这种延迟模型常用于速度无关电路的门电路中。

直觉告诉我们，惯性延迟模型和最小-最大延迟模型都具有滤去某些潜在冒险的特性。

6.1.4　基本模式和输入输出模式

除了门和导线的延迟外，还有必要考虑所设计的电路和外部环境交互作用的形式。同样，“强假设”（strong assumption）可能会简化电路的设计。当前提出的所有设计方法都是基于下列某个假设。

基本模式

假设电路处于稳定状态，即所有的输入信号、内部信号和输出信号都处于稳定状态。在这种稳定状态下，环境可以改变一个输入信号。在此之后，直到整个电路再次处于稳定状态时为止，外部环境不能再改变输入信号。由于外部环境无法获知电路中诸如状态变量这类的内部信号，所以必须计算出电路中的最长延迟时间，而外部环境输入信号的稳定时间必须大于这个最长的延迟时间。因此，电路中的门和导线的延迟必须受到以上的限制，从而对环境的限制可简化为一个绝对的时间要求。

最早采用基本模式进行异步电路设计的是David Huffman（20世纪50年代）[59, 60]。

输入输出模式

同样也假设电路处于稳定状态，允许环境改变输入信号；当电路已经产生了相应的输出信号后（如果输出没有改变也可以），允许环境再次改变输入信号。由于没有对内部信号作出假设，所以在对前一个输入变化的响应尚未稳定之前，下一个输入信号可能已经到来。

对环境的约束是用输入、输出信号传输之间的因果关系来描述的，因此，电路通常采用基于迹线（trace）的方法来说明，此时设计者详细列出在电路接口处可观测到的所有输入和输出信号变化的可能时序。后面所介绍的信号转换图，就是一种基于迹线的规范技术。

最早采用输入输出模式进行异步电路设计的是David Muller（20世纪50年代）[93, 92]，这在2.5.1节中已做过介绍，这种电路属于速度无关电路。

6.1.5　基本模式电路的综合

在Huffman的经典论文中，异步电路的外部环境一次只允许改变一个输入信号。为了响应这个输入信号的改变，组合逻辑电路会产生新的输出，这些输出的一部分信号作为反馈信号，如图6.1(b)所示。在早期的工作中，进一步要求在任一时刻只有一个反馈信号发生变化，并且

反馈缓冲器的延迟必须足够大，从而使组合逻辑电路在得到反馈信号的改变之前还能够处于稳定状态。这个反馈信号的改变又会使组合逻辑电路产生新的输出。最终，经过一系列单个信号的变化后，电路又会到达另一个稳定状态，此时再次允许环境改变一个输入信号。可以用另一种方式来表达这种行为，电路原先处于稳定状态（稳定状态的含义是指电路所处的状态保持不变直到一个输入信号发生变化），电路为了响应单输入信号的改变会依次经历一系列短暂的不稳定状态，直到最后处于一个新的稳定状态。这一系列状态之间依次只有一个变量发生变化。

鼓励感兴趣的读者参考文献[75, 133]或[95]，并详细说明和综合一个C单元。以下给出一些相关的设计流程和步骤。

- 设计可以从与同步时序电路规范相似的状态图规范开始，当然，采用何种规范是可以选择的。图6.3是C单元的一个Mealy型状态图规范。

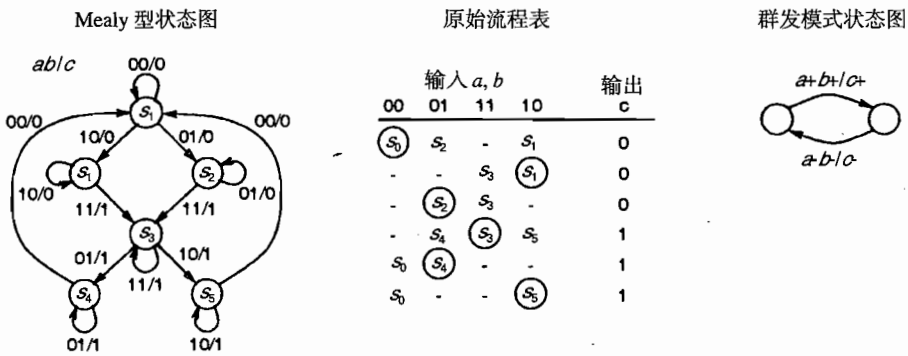


图6.3 Muller C单元的可选规范：Mealy型状态图、原始流程表及群发模式状态图

典型的设计过程包括以下步骤：

- 用原始流程表（primitive flow table）来表示目标时序电路（每个稳定状态一行的状态表）。图6.3中描述了C单元的原始流程表；
- 通过合并原始流程表中相容的状态可以得到一个最小行化简流程表；
- 对状态编码；
- 求出输出变量和状态变量的布尔方程。

我们将基本模式进一步推广，允许多个输入和多个输出改变的约束形式，这种方法称为“群发模式”（burst mode）[32, 27]。当群发模式电路处于稳定状态时，电路将等待一组输入信号发生变化（以任意顺序）。当这样的输入群体发生变化以后，电路计算出输出群体和内部变量的新值。电路只有在完全完成了对前一组输入群发响应后，外部环境才能输入一个新的群发——基本模式仍然成立，只是此时是两个群发间的变化。为了更好地与前面的基本模式进行比较，图6.3同时也给出了C单元的群发模式的描述。群发模式电路用状态图来描述，它与同步电路

设计中所采用的状态图是非常相似的。在学术界中已经开发了几种比较成熟的群发模式控制器综合工具[40, 160]，这些工具可以免费得到。

6.2　信号转换图

本章其余内容主要介绍速度无关控制电路的描述与综合。这些电路采用输入输出模式且用信号转换图（Signal Transition Graphs, STG）描述。STG 是 Petri 网的一个子类，可视为时序图的一种形式。在接下来的几节中将阐述综合过程，主要包括以下几个部分：(1) 在 STG 中描述目标电路的行为特性以及与它相关的外部环境；(2) 产生相应的状态图并且加入需要的状态变量；(3) 得出状态变量和输出的布尔方程。

6.2.1　Petri 网和 STG

简单说来，Petri 网是由有向弧和两类结点组成的图[3, 113, 94]，这两类结点称为变迁（transition）和库所（place）。通过对库所、变迁和弧赋予不同含义，Petri 网可用来对许多不同的（并发）系统进行建模与分析。某些库所用托肯来标识，且 Petri 网模型可以通过变迁的发生来“执行”。如果某个变迁的所有输入库所都含有托肯，则这个变迁有发生权，且有发生权的变迁最终会发生。当某个变迁发生后，则这个变迁的每个输入库所都会减少一个托肯，而它的每个输出库所都会增加一个托肯，后面将通过一个简单的例子进行说明。Petri 网可以非常方便地描述系统的选择与并发。

另外，需要强调的是 Petri 网有许多重要的种类和扩展，Petri 网并不只是一个已定义好的独特的模型，它是一族模型的总称。为了便于分析，通常会给 Petri 网加上某些约束。我们将要讨论的 STG 就属于带有这种约束的 Petri 网的一个子类：STG 是一个“1 有界”Petri 网，它只允许简单的输入选择。“1 有界”和“简单的输入选择”的严格定义将会在这一节的最后给出。

在 STG 中，变迁表示信号跳变，而库所和有向弧用来表示信号跳变之间的因果关系。图 6.4 表示一个 C 单元和它的虚拟环境。该虚拟环境具有一种“良好行为”，就是保持 C 单元的输入信号不变直到它的输出信号改变。C 单元的行为特性可以用图中的时序图来表示。图 6.4 中也给出了 C 单元的 Petri 网描述。在该 Petri 网中，变迁 $a+$ 和变迁 $b+$ 的输入库所中各有一个托肯，此时对应的状态是 $(a, b, c) = (0, 0, 0)$。变迁 $a+$ 和 $b+$ 可以发生，顺序不限。当它们都发生后，变迁 $c+$ 有发生权。STG 通常画成简单形式，省略了大部分的库所。可以认为每条连接两个变迁的弧中包含一个库所。图 6.4 中给出了 C 单元的 STG 描述。

Petri 网的一个给定标识，对应于该 Petri 网所表示系统的一种可能的状态，通过 Petri 网的执行获得所有可能的标识，就可得到系统的状态图。状态图通常要比相应的 Petri 网复杂。

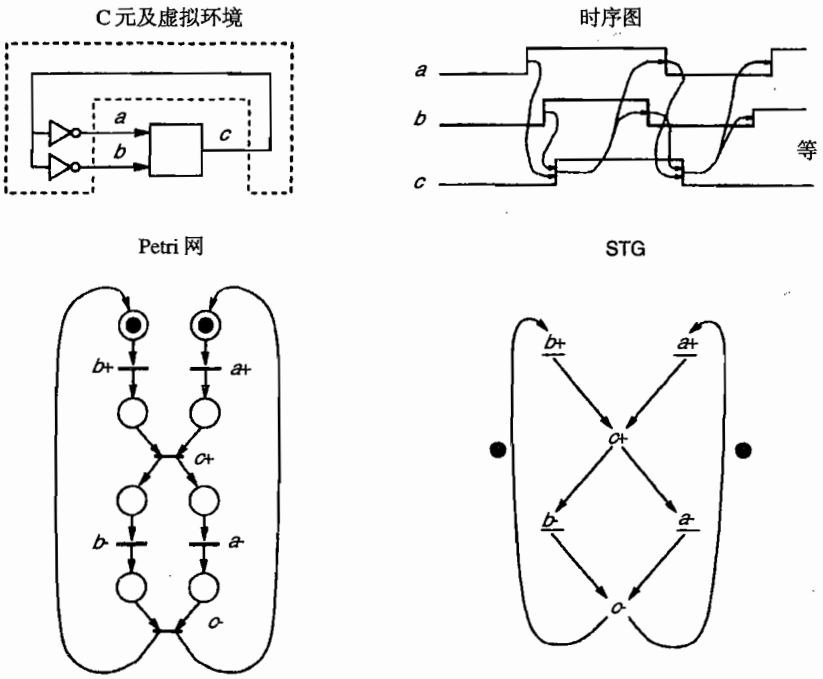


图 6.4 C 单元及其“良好行为”的虚拟环境，它的时序图形式的规范、Petri 网和时序的 STG 形式

STG 要满足某些性质才能描述具有某些特性的电路，由于使用了 Petrify 此类工具中的综合算法，还必须添加一些性质和约束。STG 可视为具有下列特性的 Petri 网：

1. **输入自由选择**：多路的选择一定只能由互斥输入来控制。
2. **1 有界**：每个库所中的托肯数不超过 1。
3. **活性**：STG 必须保证不会进入死锁状态。

描述有特定含义的速度无关电路的 STG 应具有以下特性。

4. **一致性状态赋值**：信号的跳变必须严格在 STG 中的 + 和 – 之间进行选择与执行。
5. **持久性**：如果某个信号转换能够发生，则它一定会发生，即它不会因为其他信号的转换而失去转换的可能性。电路的 STG 规范必须确保内部信号（状态变量）和输出信号的持久性，而输入信号的持久性由外部环境保证。

要能够对一个电路进行综合，则还必须具有以下特性。

6. **完全状态编码（CSC）**：在STG中不允许两个或两个以上的不同标识具有相同的信号值（即对应于相同的状态）。否则，还必须引入辅助（extra）状态变量，使不同的标识对应不同的状态。综合工具 Petrify 能够自动完成这一步。

6.2.2　常用的 STG 片段

对于初学者来说，可能需要经过一些练习才能熟练地使用 STG 描述和设计电路。在这一节中我们将介绍一些常用于构造完整描述的 STG 模板。

STG 的基本结构如图 6.5 所示，有分支、汇合、选择及并入。其中选择只在输入自由选择的情况下使用：选择库所后的变迁代表的是互斥输入信号变迁。这个要求是很自然的，我们只描述和设计确定性的电路。图 6.6 描述了使用分支、汇合、选择和并入构成的 Petri 网的图例。该例中，系统可能会顺序执行变迁 T_6 和 T_7；或者先执行变迁 T_1，接着并发执行变迁 T_2, T_3 和 T_4（也可能是以任何顺序执行），最后执行变迁 T_5。

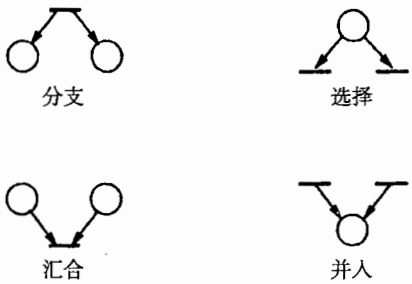


图 6.5　Petri 网中的分支、汇合、选择和并入结构

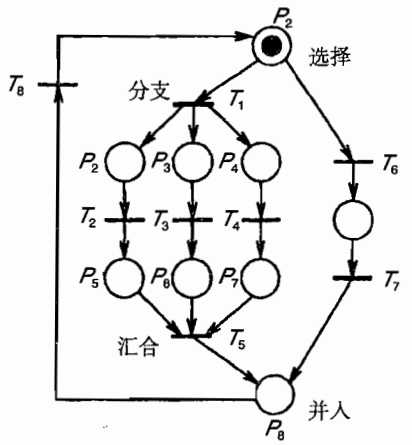


图 6.6　描述分支、汇合、自由选择、并入的 Petri 网图例

在本章的最后，我们将设计一个 4 相捆绑数据多路选择器（见图 3.3）。为了设计它，还需要增加一些结构：受控选择（controlled choice）和一个捆绑数据通道输入端的 Petri 网片段。

图 6.7 给出的 Petri 网片段中，库所 P_1、变迁 T_3 和 T_4 是一个受控选择：库所 P_1 中的托肯只能参与变迁 T_3，T_4 中的一个。选择是由库所 P_2 或 P_3 中的托肯的出现来控制的。至关重要

的一点是，这两个库所中不能同时拥有托肯，本例中是通过互斥的输入信号变迁T_1，T_2来保证的。

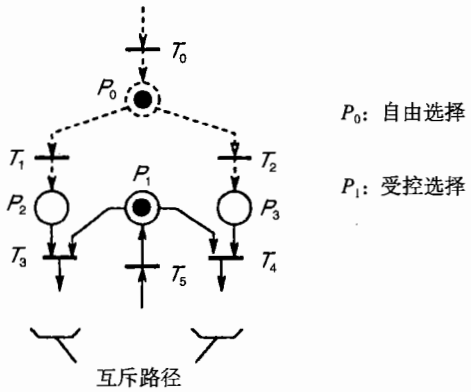


图 6.7　包含控制选择的 Petri 网

图6.8给出了一个元件的Petri网的片段，该元件有1位输入通道，使用4相捆绑数据协议，可用多路选择器和多路分配器（图3.3中已介绍过）来控制通道。图中的两个变迁dummy1和dummy2并不代表通道中3个信号的变迁，它们只是为了更好地描述电路而引入的虚拟变迁。这些虚拟变迁代表的是基本STG的一种扩展。

还必须注意到4条连接弧：

库所P_5和变迁Ctl+之间的弧
库所P_5和变迁Ctl–之间的弧
库所P_6和变迁dummy2之间的弧
库所P_7和变迁dummy1之间的弧

这些连接弧的两端都有箭头，这是每个方向都有一条弧的简单表示形式。图中还有一些库所，它既是某个变迁的输入库所又是该变迁的输出库所。库所P_5和变迁Ctl+就是这种情况。

这个Petri网片段的整体结构可以这样来理解：顶端是用来描述Req和Ack信号握手过程的一连串的变迁和库所。底部是一个由库所P_6和P_7以及变迁Ctl+和Ctl–构成的一个环路，这个环路用于捕获控制信号在高、低电平之间变化的情况。当Req为高电平时，库所P_5中没有托肯，则表示在当前周期中Ctl是稳定的。当Req为低电平时，库所P_5中有一个托肯，则此时允许Ctl做任意多的跳变。当变迁Req+发生后，库所P_4（受控选择库所）中会出现一个托肯。现在Ctl信号处于稳定状态，这时变迁dummy1和dummy2之间，有一个有发生权并最终会发生。这时就可以开始进行需要的输入到输出的运算，图中并没有显示出这一部分，最后在控制端的握手也完成了(Ack+, Req–, Ack–)。

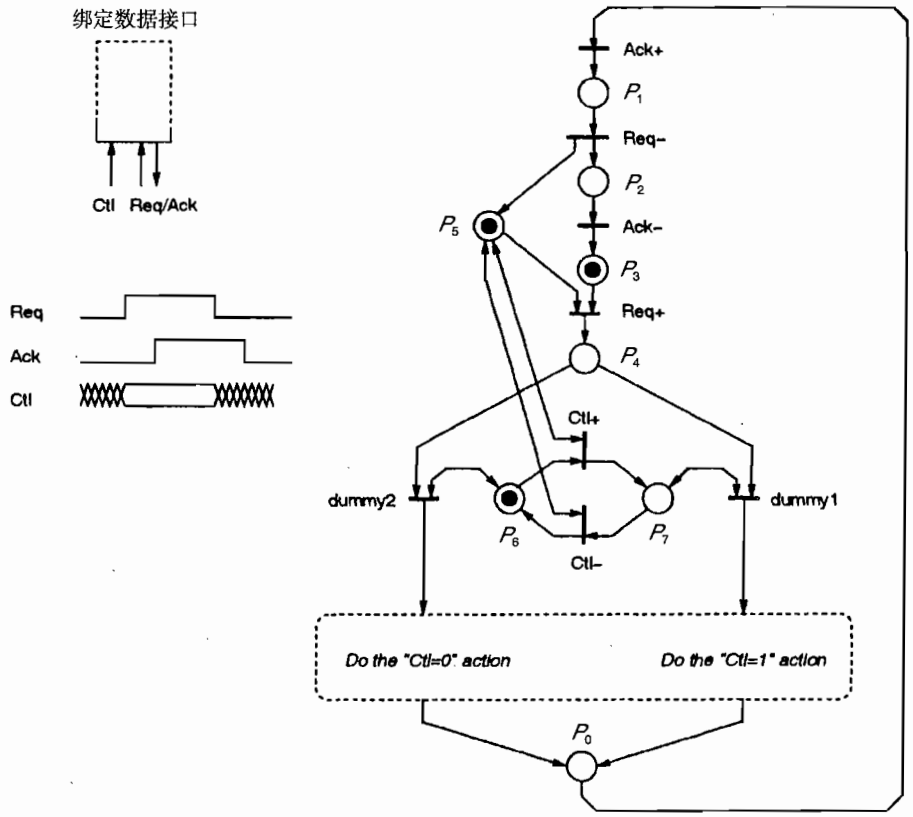


图 6.8　有 1 位输入通道的 4 相捆绑数据协议元件的 Petri 网片段

6.3　基本综合过程

综合过程从 STG 开始，STG 应满足 6.2.1 节中所列条件。根据 STG，在给定初始标识下，识别出 STG 所有的可达标识，并得到相应的状态图。综合过程的最后一步是得出状态变量和输出变量的布尔方程。

我们将通过一些例子更好地说明综合过程。由于要得到电路的状态，必须要得出电路中信号的所有值，因此综合过程中的计算复杂度就可能很高，即使是小规模的电路其计算量也可能会很大。在实际情况下，设计者常常会用一些已有的 CAD 工具，如后面将要介绍的 Petrify。

6.3.1　例 1：C 单元

图 6.4 给出的 STG 的状态图如图 6.9 所示。在给定状态下处于受激状态的变量上都标有一个星号。图 6.9 中还有输出信号 c 的卡诺图。c 的布尔方程必须包括 $c = 1$ 的状态和 $c = 0$*

（从0跳变到1）的受激状态的情况。在卡诺图中为了更好的区分受激变量和稳态变量，本书后面部分用R（rise的简写，表示上升沿）来替代0*，用F（fall的简写，表示下降沿）来替代1*。

令人鼓舞的是，我们可以成功地实现常见的电路，但用C单元来说明设计过程的方方面面实在是太过于简单了。

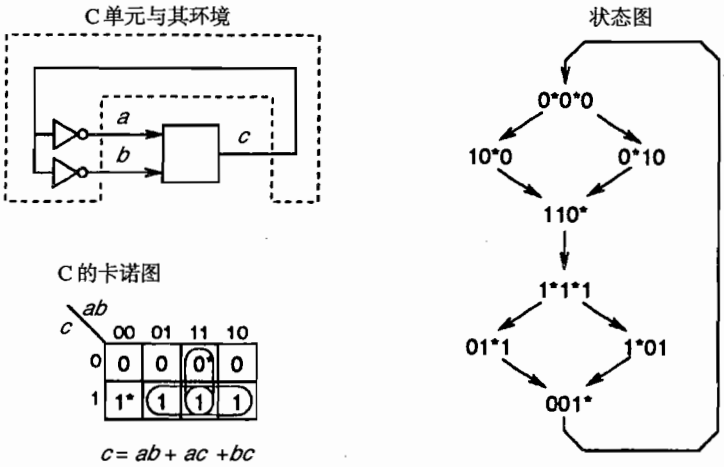


图6.9 图6.4所示C单元STG的状态图和布尔方程

6.3.2 例2：具有选择的电路

下面这个例子能够更好地说明综合过程，并且在后面的章节中我们还将再次用该例来解释其他更有效的实现方法。这个例子还是比较简单的——电路中只有2个输入和2个输出——但它却涉及到几乎所有与综合有关的问题。本例选自犹他州立大学Chris Myers在1996年时所讲述的课程“异步VLSI系统设计”（EE 587），但该例最早出自论文[12, 13]。

如图6.10所示的电路中，有两个输入a和b以及两个输出c和d，时序图显示出这个电路有两种可选的工作方式。电路相应的STG描述以及状态图如图6.11所示。STG中只包含自由选择库所P_0和并入库所P_1。所有直接连接两个变迁的弧都包含一个库所。状态图中所有的状态都用十进制数来表示，这样能够更方便地填入到卡诺图中。

这个STG满足6.2.1节中的所有6条性质，因此能继续处理并得出输出信号c和d的布尔方程［注意，在状态0时两个输入信号都处于受激状态，$(a, b) = (0^*, 0^*)$，且在状态4和状态8时两个输入信号中有一个仍为0，但不处于受激状态。这只是表述的问题。在实际情况下，在状态0时两个输入变量中只有一个处于受激状态，但是具体是哪个并不清楚。除此之外，还要求STG对于内部信号和输出信号都具有持久性。而输入信号的持久性是由外部环境来保证的］。

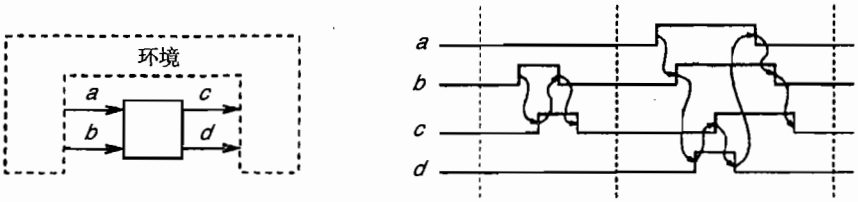


图 6.10　文献[12, 13]中的实例电路

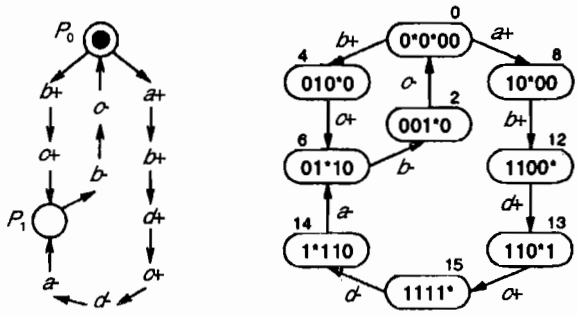


图 6.11　STG 规范及相应的状态图

图 6.12 中给出了输出 c 的卡诺图、布尔方程和两种可供选择的门级实现：采用基本复合门（an atomic complex gate）的实现，采用简单的与门和或门的实现。有些状态在电路中不会出现，所以在卡诺图中不可达状态是不需要考虑的（任意态）。至于输出信号 d 的实现问题，这里作为一个练习留给读者自己完成（$d = ab\bar{c}$）。

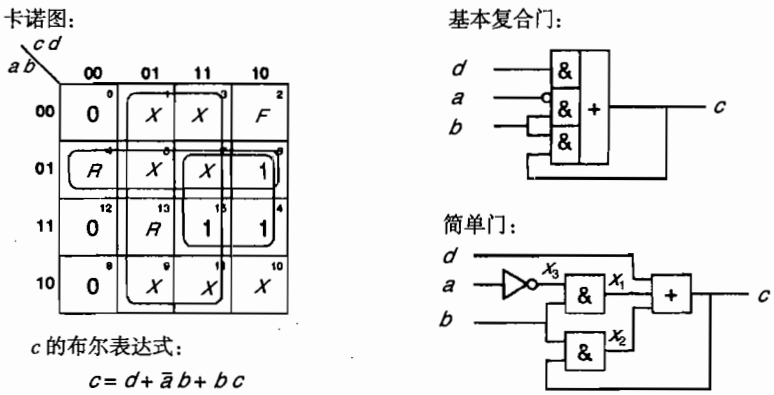


图 6.12　输出 c 的卡诺图、布尔表达式和两种可选择的门级实现

6.3.3　例 2：简单门实现中的冒险问题

图 6.10 所示的 STG 满足所有的 6 条实现条件（包括持久性），并且每个输出信号用一个复合门来实现是无冒险的。对于输出信号 c，需要对输入信号 a 取反，并需要一个复合的“与或”

门来实现。一般来说，这种简单复合门的实现并不容易，它常常需要把电路的实现分解成由更简单门组成的形式，然而这会引入辅助变量，而这些辅助变量可能并不能满足持久性要求，即所有的受激信号变迁最终必须要能够发生。因此，保持速度无关的逻辑分解是很有意义的课题[20, 76]。

图6.12给出了用简单门来实现输出信号c的电路，但该电路并不是一个速度无关电路，可能会出现静态冒险和动态冒险，这是说明冒险机制的一个很好的例子。问题在于原先的STG和状态图中并不包含信号x_1，x_2和x_3。进一步分析发现，包含这些信号的STG将不再满足持久性的要求。下面解释可能的故障序列，它是由信号c上的静态1冒险和动态10冒险引起的，我们通过图6.13来讨论。

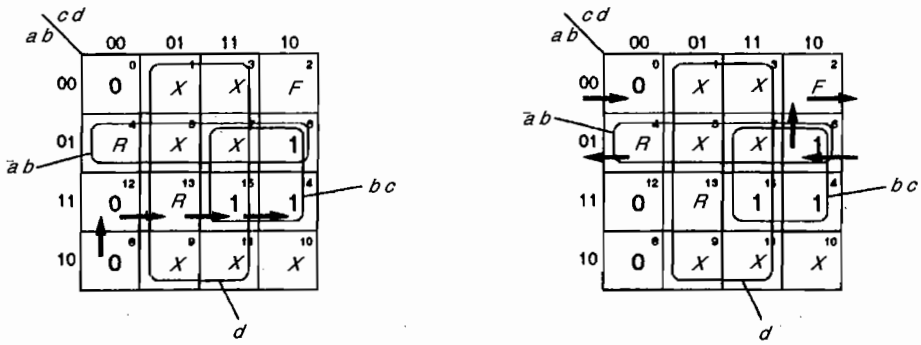


图6.13 输出信号c的卡诺图，表明可能导致冒险的状态顺序

静态1冒险：如果电路依次经过12, 13, 15和14这几个状态，则可能发生静态1冒险。从状态12到状态13的跳变对应于d变成高电平，而从状态15到状态14的跳变对应于d又再次变为低电平。在状态13时c处于受激状态（R）且在状态13, 15, 14和6时c都保持高电平。状态13和状态15都包含在方框d内，而状态14包含在方框bc内，当d变为低电平时，bc会“接管”并保持$c=1$。如果以对应于方框bc的x_2作为输出信号的与门变化较慢，则此时会出现静态1冒险。

动态10冒险：如果电路依次经过4, 6, 2和0这几个状态，则可能发生动态10冒险。此时对应的是位于上部的与门（输出信号x_1）和或门，起传播$b+$到$c+$或$b-$到$c-$的作用。然而当发生$c+$跳变后，位于下部的与门（x_2）会处于受激状态（R），但是这个门的激发并不需要通过用其他信号的变化来指示——因为或门已经有一个输入为高电平。如果下面的这个与门（x_2）激发，则它稍后会因为响应$c-$而再次处于受激状态（但此时是F）。在$c-$转变发生后，下面的与门（x_2）会在输出c上加一个0-1-0脉冲。

以上我们并没有讨论输入信号为a和输出信号为x_3的反相器。由于输入a并不是其他任何门的输入，所以这个分解属于SI。

总之，这两种冒险都与电路经过的一系列状态有关，这些状态都包含在几个方框中，而这些方框表示信号维持在同一电平（稳定状态）上。“接管”的方框表示某个信号可能不能被其他的任何信号指示。它和我们在2.2节和2.4.3节中所涉及的问题本质上是一样的——只有当或门的一个输入信号为高电平时它才能够被指示。

6.4　采用状态保持元件实现

6.4.1　引言

运算过程中，电路中的每个变量都会经历一系列的状态变化。如果变量当前处于（稳定的）0状态，则接下来会是一个或多个受激状态（R），而再接着会是一系列（稳定的）1状态，然后再经过一个或多个受激状态（F），依此类推。在这样的实现过程中，已经包含了变量z为高电平或受激到高电平（$z=1$和$z=R=0^*$）时的所有状态。

电路的实现还可以采用状态保持元件，如置位-复位锁存器。对于置位和复位信号的布尔方程，只需分别考虑$z=R=0^*$和$z=F=1^*$的情况。这样可以得到更简单的表达式和潜在的简单分解。图6.14所示的是采用标准的置位、复位锁存器以及标准C单元的实现模板。在采用标准C单元实现模板中，复位信号还必须要取反。在6.4.5节中我们还将讨论一些其他的更巧妙的实现方式。但对于下面所要进行的讨论，图6.14所示的基本拓扑电路已经足够了。

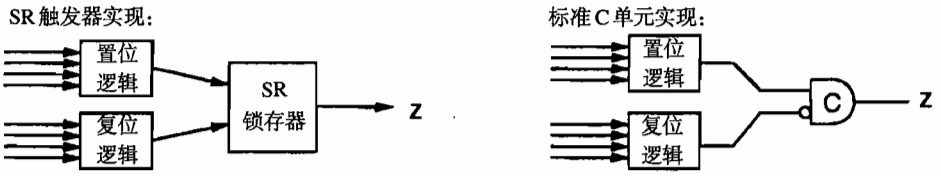


图6.14　使用（简单的）状态保持单元的实现模板

此时我们还必须提到，用于确定信号z的状态保持元件何时复位与何时置位的布尔方程，可改为以下形式：

$$z = \text{“Set”} + z \cdot \overline{\text{“Reset”}} \tag{6.1}$$

对6.3.2节中图6.12中的信号c，可以得到它的置位和复位函数表达式为$c_{set}=d+\bar{a}b$和$c_{reset}=\bar{b}$（这与6.4.3节的图6.15中的结果是一致的）。除此之外，很显然的一点是对于所有的可达状态，信号z的置位和复位函数不能在同一时刻都处于活动状态：

$$\text{“Set”} \wedge \text{“Reset”} \equiv 0$$

接下来的章节中还将继续介绍这种用状态保持元件来实现的方法，并且我们还将通过前一节中的例 2 的重新实现来说明这一技术。

6.4.2 激发区和静态区

前面所说的为每个变量采用一个状态保持元件的思想可描述如下。

激发区 ER：对于变量 z，ER 是指它处于受激状态的状态最大连接集：

- ER(z+)表示 $z = R = 0^*$ 时所处的状态区域；
- ER(z−)表示 $z = F = 1^*$ 时所处的状态区域。

静态区 QR：对于变量 z，QR 是指它处于非受激状态的状态最大连接集：

- QR(z+)表示 $z = 1$ 时所处的状态区域；
- QR(z−)表示 $z = 0$ 时所处的状态区域。

对一个给定的电路，其状态空间可以被分成每种类型的无交集的一个或多个区域。

变量 z 的置位函数：

- 必须包含 ER(z+)区域中的所有状态；
- 可能包含 QR(z+)区域中的某些状态；
- 可能包含电路中不可达的状态。

变量 z 的复位函数：

- 必须包含 ER(z−)区域中的所有状态；
- 可能包含 QR(z−)区域中的某些状态；
- 可能包含电路中不可达的状态。

在接下来的 6.4.4 节中，我们将会增加单调包含约束（monotonic cover constraint）或唯一进入约束（unique entry constraint）来避免冒险的出现：

- 在变量的置位、复位函数中，只能允许通过变量受激时所处的状态进入方框（乘积项）。

有了最后这个约束后，我们已经有了速度无关电路设计的完整方法，电路中非输入信号用一个状态保持元件实现。接下来我们将继续讨论例 2。

6.4.3　例 2：采用状态保持元件

仍以 6.3.2 节和 6.3.3 节中的例 2 为例，用图 6.15 说明上述方法。如前所述，为了确保所设计的电路是速度无关的，布尔方程（对于置位和复位函数）可能需要采用基本复合门来实现。

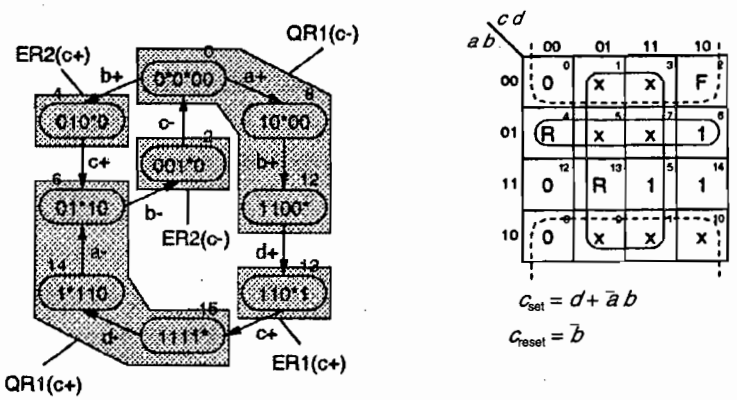


图 6.15　图 6.10 所示电路实例中信号 c 的状态图中的激发区域和静止区域及相关的置位、复位方程

6.4.4　单调包含约束

图 6.16 中描述了基于标准 C 单元的信号 c 的实现，以及置位和复位函数的简单门实现。图中有一个卡诺图，卡诺图中对置位和复位函数进行了化简。置位函数只涉及两个方框 d 和 $\bar{a}b$，它们是或门的输入。如前面所讨论的那样，这种实现方式在信号 c_{set} 上可能会出现动态 10 冒险。图 6.16 中的卡诺图给出了一个可能会导致故障的状态序列（8, 12, 13, 15, 14, 6, 0）。信号 d 在状态 12 时为低电平，而在状态 13 和 15 时为高电平，在状态 14 时又再次处于低电平。这个状态序列相当于 d 上的一个脉冲。通过或门使信号 c_{set} 产生一个脉冲，这又会使得 c 变为高电平。在后面的状态 2 时，c 又会再次变为低电平。这是电路所期望的动作，问题是 c_{set} 表达式中的内部信号 x_1 在状态 6 时会成为受激（$x_1 = R$）状态。如果与门输出比较慢，c 重新复位后会在 c_{set} 信号上出现一个不希望的脉冲。

如果方框 $\bar{a}b$（包括状态 4, 5, 7 和 6）简化成只包括状态 4 和 5，对应于 $c_{set} = d + \bar{a}b\bar{c}$，就可避免以上问题。这一修改的效果是或门的输入不会有多个高电平出现，如果确实出现了这种情况，也没有指示原则的问题（在第 2 章中讨论过指示和双轨电路）。即只允许经过那些属于激发区的状态才能进入方框。这个条件可以视为

- **单调包含约束：**在任何给定时刻，在最小项之和的实现中，只允许一个乘积项为高电平。显然，这个条件只需要在状态或电路的可达状态内满足即可。
- **唯一进入约束：**只可通过激发区内的状态进入方框。

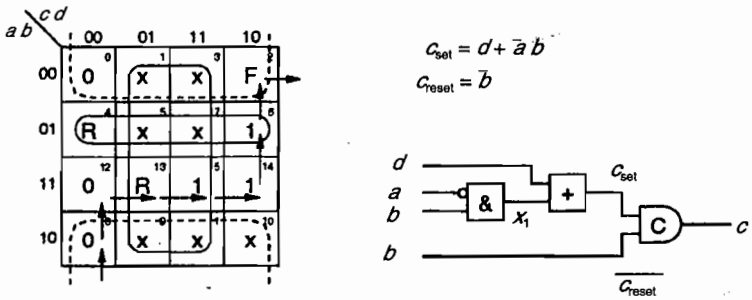


图 6.16 用标准 C 单元和简单门实现的 c，从卡诺图中得出置位和复位函数

6.4.5 采用状态保持元件的电路拓扑

除上述用置位复位触发器和标准C单元模板外，还有其他用状态保持元件实现变量的方法。

对 CMOS 晶体管级设计者来说，常用的方法是采用普通的 C 单元。这样，状态保持机制和置位复位功能可由 N 型和 P 型晶体管的混合结构实现。图 6.17 给出了 $z_{set} = ab$ 和 $z_{reset} = \bar{b}\bar{c}$ 电路的门级符号，以及动态和静态 CMOS 实现。

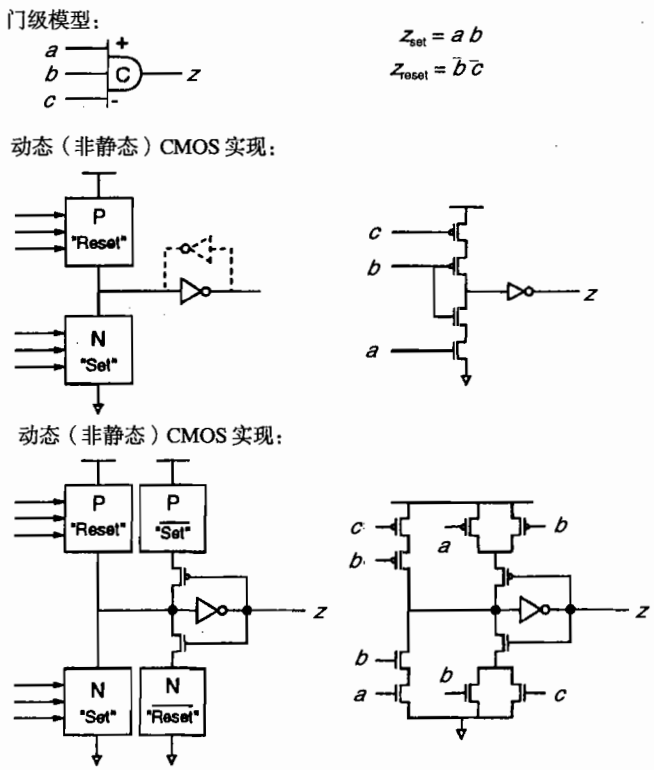


图 6.17 C 单元：门级模型及一些 CMOS 晶体管实现电路

另一种实现方式是采用标准元件库，它对设计者来说更具吸引力。图6.18描述了采用标准元件库的与或非复合门的实现。这个电路有一个非常好的特性：它能够得到期望的信号z和$\bar{z}$，并且在信号跳变过程中不会出现$(z, \bar{z}) = (1, 1)$的情况。这个例子中同样也是$z_{set} = ab$和$z_{reset} = \bar{b}\bar{c}$。

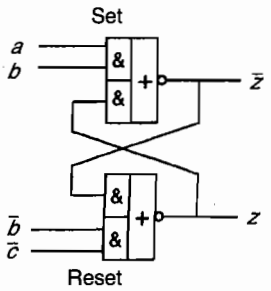


图6.18　用两个与或非复合门的SR实现

6.5　初始化

在实际电路设计过程中，电路的初始化是非常重要的一个方面，但在前面我们对此还未涉及。在综合过程中会给电路设定一个初始状态，这个初始状态对应STG的初始标识。如果电路起始于同一初始状态，则综合得到的电路就是正确的速度无关的电路。由于综合后电路中通常采用状态保持元件或带有反馈回路的电路，因此有必要使电路进入预先设置的初始状态。

所以，设计者还必须以手工方式对电路进行后综合处理以及给电路加一个额外的信号，这个信号有效时，把所有的状态保持结构设置为期望的状态。通常来说，对这个初始信号而言电路不会是速度无关的，必须假定这个初始信号有足够长的时间保持有效，使电路做出期望的行为。

对于采用诸如置位-复位锁存器和标准C单元等状态保持元件实现的电路，如果这些状态保持元件除了通常的输入外还有特殊的清零/预置信号，则可以不对电路进行初始化。通常情况下，设计者必须把一个初始化信号明确地加入相关的布尔方程中。如果针对给定的元件库进行综合，则这个修正的方程可能需要进一步地进行逻辑分解，就如我们前面所了解到的，这样可能会影响电路的速度无关性。

事实上，综合过程中不包含初始化显然是一个缺陷，但通常来说有些设计者可能会设计控制电路元件库。在更为抽象的“静态数据流结构”层次上（见第3章）设计电路时，可以把这些设计好的元件作为构造块。

根据以上所述，对所有的控制电路初始化是一个简单而又鲁棒性很强的方法，但也可以采用隐式的方法来对基于握手元件的异步电路进行初始化，这种方法利用电路的内在功能来把初

始信号值“传递”给电路。在Tangram中（见8.3节和第三部分的第13章），这种方法称为“自初始化法”[135]。

6.6 综合过程概述

前面章节的内容覆盖了利用STG技术指标来综合SI控制电路的基本理论。文中的表述有意选择一种非正式的方式，主要侧重于用一些例子来说明综合过程以及综合理论背后的一些比较直观的知识。

这些理论主要来自于以下一些大学和研究小组的工作：Illinois大学[93]、MIT[26, 24]、Stanford[13]、IMEC[145, 159]、St. Petersburg电子工程研究所[146]以及由多国研究者共同合作开发的Petrify工具[29]，我们将在下节介绍这一工具。作者也参加过一些讨论，通过这些讨论清楚地了解到，在很多情况下理论和概念是由不同的小组独立提出来的，本书没有详细地去描述这个理论的发展历史。建议有兴趣深入了解这些内容的读者参考相关的文章，特别推荐Myers的著作[95]。

总之，前面章节所列出的综合过程主要由以下几个步骤构成。

1. 采用STG来描述电路的行为及其（虚拟）外部环境。
2. 检验所得到的STG是否满足6.2.1节中所列出的性质1~性质5：1有界，一致性状态赋值，活性，只有输入自由选择和受控选择，持久性。STG只有满足以上这些条件才是一个有效的SI电路描述。
3. 检查STG描述是否满足6.2.1节中的性质6：完全状态编码（CSC）。如果这个描述不满足CSC，还必须在STG中加入一个或多个状态变量或者改变描述（在4相控制电路中，可以改变向下跳变的信号）。一些工具（如Petrify）可以自动地插入状态变量，然而对信号的重组（意味着需要对描述进行修改）则需要设计者自己完成。
4. 选择一个实现模板并且得出变量本身所对应的布尔方程，对于采用状态保持元件则需要得出置位复位函数。另外还必须做出选择：这些表达式是用基本复合门（一般为与或非门）来实现还是采用更简单的门来实现。这些选择可以在综合工具中通过设定选项来完成。
5. 对所采用的实现模板推导出布尔表达式。
6. 手工修改实现，如通过使用确定的复位信号或初始化信号可以强制电路进入一个期望的初始状态。
7. 把设计输入到CAD工具进行仿真且对电路（或者是由电路组成的系统）进行布局布线。

6.7 Petrify：从STG综合SI电路的工具

Petrify是一个可以控制Petri网并可以由STG描述综合SI控制电路的公开工具，可通过网址http://www.lsi.upc.es/~jordic/petrify/petrify.html获得。

Petrify是具有很多选项和开关项的典型的UNIX程序。对于一个电路设计者来说，可能更喜欢输入一个描述就能得到一个电路的按钮操作式的工具。Petrify可以这样用，但它还有其他的功能。如果进一步了解，则会发现Petrify是一个交互式的工具，可以对Petri网、STG和状态图进行描述、检查以及操作。在下一节中，我们将通过一些实例说明如何用Petrify来设计速度无关的控制电路。

通过简单的文本的形式把STG描述输入到Petrify中。采用Petrify的绘图程序draw_astg（基于由AT&T开发的图形可视化包“dot”）可以得到STG的图形表示和状态图。图形表示的形式固然很“好”，但拓扑结构可能会与电路设计者的初衷不同。即使是检查STG文本形式的描述是否与STG等价，这样简单的工作也是很困难的。

为了克服以上缺点，丹麦技术大学开发了一种称为VSTGL（Visual STG Lab）的图形化STG输入和仿真工具。为了帮助设计者得到一个正确的描述，VSTGL提供了一个交互式仿真器，它允许设计者自由地加入托肯和触发变迁。利用这个仿真器还可以对STG进行一些简单的检查。

可以从网站http://vstgl.sourceforge.net/中得到VSTGL。VSTGL来自于两个大学4年级的学生所做的一个小型学生项目。虽然信号变迁的命名可能让人觉得有点别扭，但VSTGL还是一个稳定且可靠的工具。

Petrify可以通过在STG中插入状态变量来解决CSC冲突的问题，可以通过6.4节中介绍过的方法来实现。

- **-cg选项**：将产生由复合门实现的电路（电路中的每个非输入信号由一个单输入复合门实现）。
- **-gc选项**：将产生由普通的C单元实现的电路。对每个非输入信号，Petrify输出的是置位和复位函数的布尔表达式。
- **-gcm选项**：将产生由普通C单元实现的电路，此时置位和复位函数满足单调包含约束。因此，该电路同样可以用标准C单元实现，而此时置位和复位功能采用简单的与门和或门实现。
- **-tm选项**：Petrify把电路工艺映射到用户指定的门元件库上。很显然，工艺映射不能与-cg和-gc选项组合使用。

Petrify工具中带有使用手册和一些例子。下一节中将讨论几个例子，这些例子在本书前面章节中已讨论过。

6.8　用Petrify设计的实例

下面将通过几个实例的描述和综合来说明Petrify的使用。(a) 例2具有选择的电路；(b) 用3.3节图3.3中的锁存器实现的用于4相捆绑数据的控制电路；(c) 用3.3节图3.3中的多路选择器实现的用于4相捆绑数据的控制电路。上述所有实例都假设它们是推通道。

6.8.1 例2回顾

作为第一个例子，我们将对不同版本手工设计的例2进行综合。图6.19是输入到VSTGL中例2的STG。相应地，Petrify的文本输入（ex2.g文件）和由Petrify图形化显示的STG如图6.20所示。在图6.20中，为使文本输入更为简便，当一个信号跳变多次发生时，给其加一个序号。

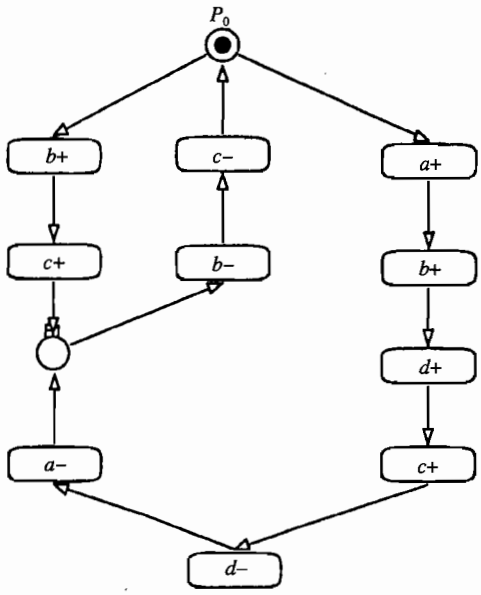


图6.19 例2中输入到VSTGL的STG

```
.model ex2
.inputs a b
.outputs c d
.graph
P0 a+ b+
c+ P1
b+ c+
P1 b-
c- P0
b- c-
a- P1
d- a-
c+/1 d-
a+ b+/1
b+/1 d+
d+ c+/1
.marking { P0 }
.end
```

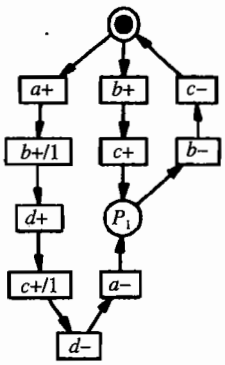


图6.20 例2的STG的文字描述及由Petrify产生的STG图

使用复合门：

```
> petrify ex2.g -cg -eqn ex2-cg.eqn

The STG has CSC.
# File generated by petrify 4.0 (compiled 22-Dec-98 at 6:58 PM)
# from <ex2.g> on 6-Mar-01 at 8:30 AM
     ...
# The original TS had (before/after minimization) 9/9 states
# Original STG:    2 places,  10 transitions,  13 arcs  ...
# Current STG:     4 places,   9 transitions,  18 arcs  ...
# It is a Petri net with 1 self-loop place
     ...

> more ex2-cg.eqn

# EQN file for model ex2
# Generated by petrify 4.0 (compiled 22-Dec-98 at 6:58 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 7.00

INORDER = a b c d;
OUTORDER = [c] [d];
[c] = b (c + a') + d;
[d] = a b c';
```

使用通用C单元：

```
> petrify ex2.g -gc -eqn ex2-gc.eqn
   ...
> more ex2-gc.eqn

# EQN file for model ex2
# Generated by petrify 4.0 (compiled 22-Dec-98 at 6:58 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 12.00
INORDER = a b c d;
OUTORDER = [c] [d];
[0] = a' b + d;
[1] = a b c';
```

```
[d] = d c' + [1];        # mappable onto gC
[c] = c b + [0];         # mappable onto gC
```

对于普通C单元的方程应当根据式(6.1)来“解释”。

使用标准C单元置位和复位函数满足单调包含约束：

```
> petrify ex2.g -gcm -eqn ex2-gcm.eqn
  ...
> more ex2-gcm.eqn

# EQN file for model ex2
# Generated by petrify 4.0 (compiled 22-Dec-98 at 6:58 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 10.00

INORDER = a b c d;
OUTORDER = [c] [d];
[0] = a' b c' + d;
[d] = a b c';
[c] = c b + [0];        # mappable onto gC
```

同样，对于普通C单元的方程应当根据式(6.1)来“解释”。

6.8.2 4相捆绑数据锁存器的控制电路

图6.21所示的是一个左右两端各带有虚拟外部环境的异步握手锁存器。它可用一个普通的N位透明锁存器和控制电路来实现，而控制电路需要设计。锁存器控制信号Lt可能需要一个驱动器。为了使锁存控制器具有鲁棒性且独立于驱动器的延时，我们可以反馈缓存信号（Lt），从而控制器能够知道信号何时到达锁存器。图6.21给出了STG描述的片段——左右端外部环境的握手和锁存器控制器的行为特性。首先Lt为低电平且锁存器是透明的，当新的输入数据到达时将会流经锁存器，如果右端外部环境已经准备开始握手（Aout = 0），为了响应Rin+，控制器可能会立即生成Rout+信号。此外，数据被Lt+锁存，且发送一个应答信号Ain+到左端外部环境。当锁存器跳回到透明模式后，对信号Rin-的响应跟上面的描述对称。把这些STG片段组合起来就得到了图6.22的STG。

Petrify运行后的结果如下：

```
> petrify lctl.g -cg -eqn lctl-cg.eqn

The STG has CSC.
# File generated by petrify 4.0 (compiled 22-Dec-98 at 6:58 PM)
```

```
# from <lctl.g> on 6-Mar-01 at 11:18 AM
...
# The original TS had (before/after minimization) 16/16 states
# Original STG:    0 places,  10 transitions,  14 arcs (  0 pt + ...
# Current STG:     0 places,  10 transitions,  12 arcs (  0 pt + ...
# It is a Marked Graph
.model lctl
.inputs Aout Rin
.outputs  Lt Rout Ain
.graph
Rout+ Aout+ Lt+
Lt+ Ain+
Aout+ Rout-
Rin+ Rout+
Ain+ Rin-
Rin- Rout-
Ain- Rin+
Rout- Lt- Aout-
Aout- Rout+
Lt- Ain-
.marking { <Aout-,Rout+> <Ain-,Rin+> }
.end
```

```
> more lctl-cg.eqn

# EQN file for model lctl
# Generated by petrify 4.0 (compiled 22-Dec-98 at 6:58 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 7.00
INORDER = Aout Rin Lt Rout Ain;
OUTORDER = [Lt] [Rout] [Ain];
[Lt] = Rout;
[Rout] = Rin (Rout + Aout') + Aout' Rout;
[Ain] = Lt;
```

[Rout]的表达式可以写成

```
[Rout] = Rin Aout' + Rout (Rin + Aout')
```

该式可视为输入是Rin和Aout的C单元。

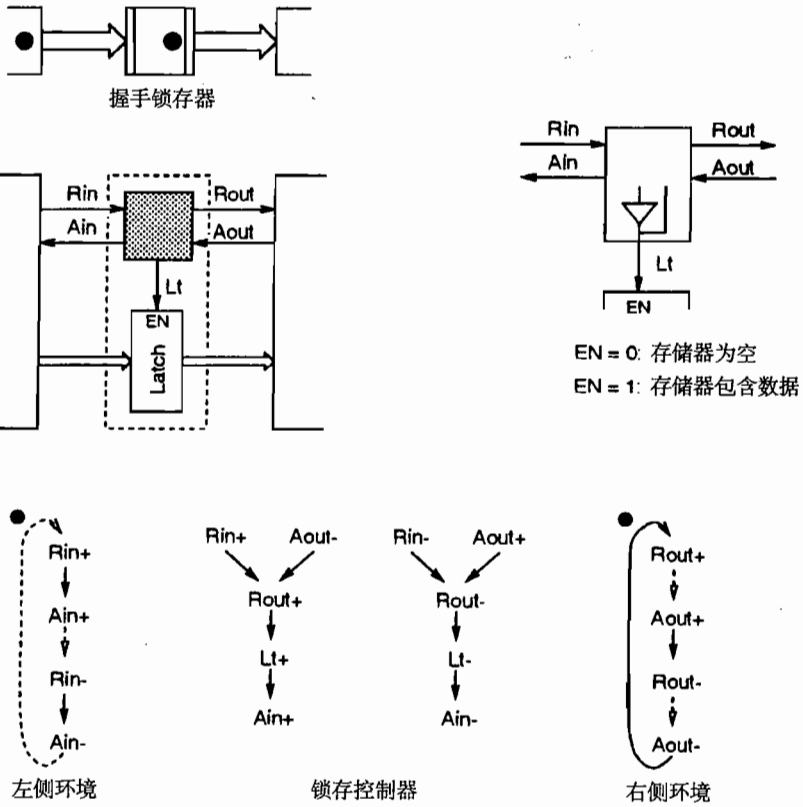


图 6.21　4 相捆绑数据锁存器和有关行为“捕获方法”的 STG 片段

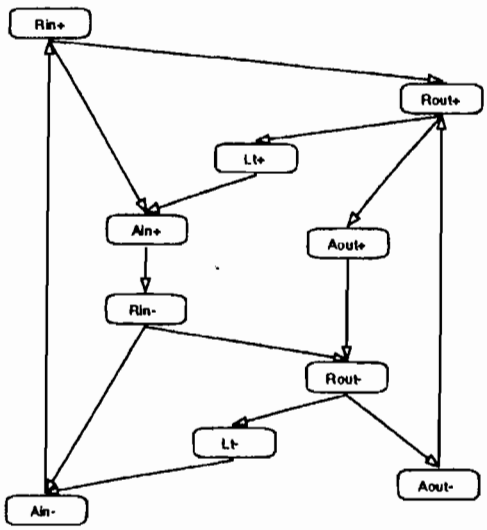


图 6.22　锁存控制器的 STG（作为 VSTGL 的输入）

6.8.3　4 相数据捆绑多路选择器的控制电路

通过前面两个例子的介绍，对这些熟悉的电路，我们已经解决了它们的实现问题。现在让我们考虑一个更为复杂的例子，它不能（不易）由手工实现。图 6.23 给出的握手多路选择器来自于图 3.3。图中也描述了如何用“规则”的组合电路构成的多路选择器和控制电路来实现一个握手多路选择器。下面设计一个 4 相捆绑数据的速度无关的 MUX 控制电路。

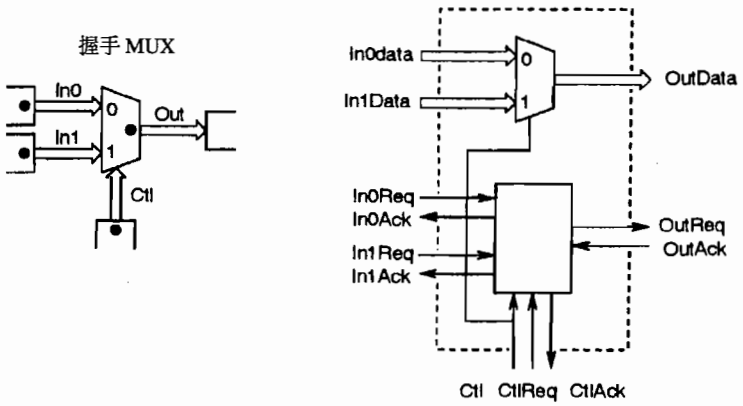


图 6.23　握手多路选择器和 4 相捆绑数据实现的结构

这个 MUX 有 3 个输入通道，并且我们假定它们与 3 个独立的虚拟环境相连。图中的黑点提示我们这些通道是推通道。当描述 MUX 控制电路和它的（虚拟）环境的行为时，明确这一点是非常重要的。在绘制STG时出现的一个典型错误是指定的环境比真实的环境具有更为有限的行为。对于3个输入通道中的每一个，其STG都要引入一个循环（Req+; Ack+; Req-; Ack-; …），并且初始化时每个循环中都包含一个托肯。

如前面所提到的，有时采用双轨（或一般的 1-of-N）数据编码方式能够更容易地处理控制通道问题，此处是指处理独热（编码）控制信号。为了得到一个采用完全 4 相捆绑数据通道的MUX的STG，第一步先得到一个控制通道采用双轨信号（Ctl.t, Ctl.f和CtlAck）的MUX的 STG，如图 6.24 所示。将这个 STG 与图 6.8 所示的 4 相捆绑数据通道的 STG 片段组合在一起，可得到图 6.25 所示的 STG。图 6.24 中的“中间形式的”STG 强调了这样一个事实：MUX 可视为一个受控的汇合——二者互斥且结构相同各占一半，它们是汇合的STG的主要部分。

Petrify 运行后的结果如下，这里选择 -o 选项，把作为结果的 STG（可能还带有附加的状态信号）写入文件，而不是用标准输出。

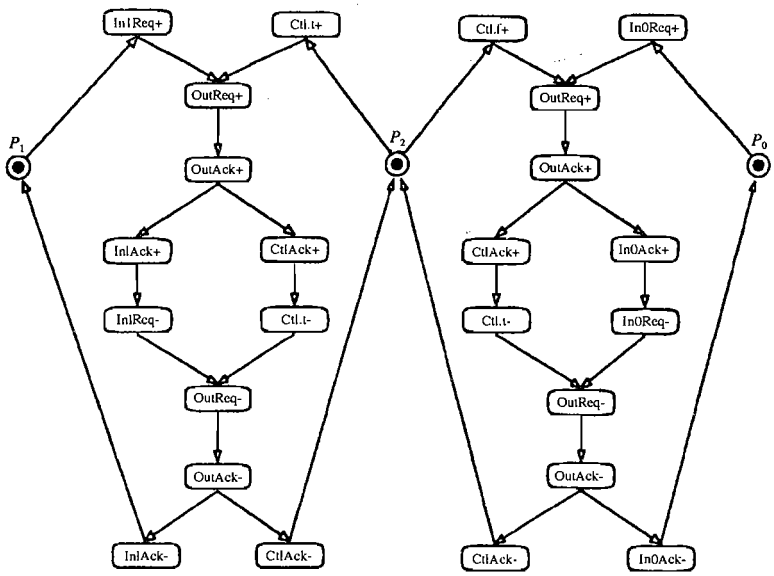


图 6.24　使用4相双轨控制通道的4相捆绑数据MUX控制电路的STG规范，由捆绑数据（控制）通道片段和一个完整的 4 相双轨 MUX 产生的 STG 结合所得（图 6.25）

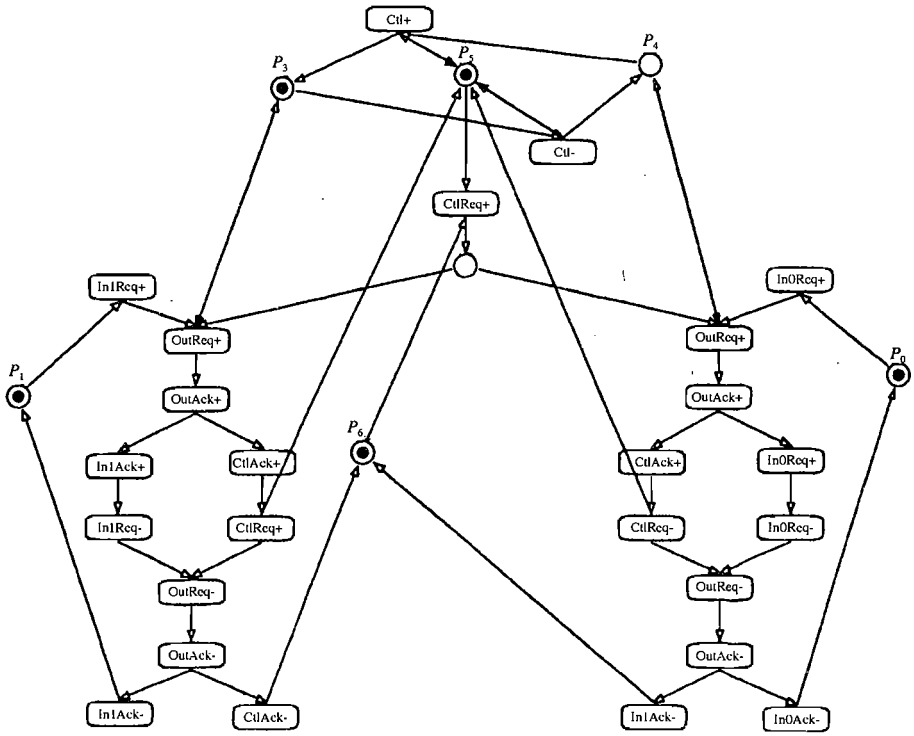


图 6.25　4 相捆绑数据 MUX 控制电路的最终 STG 规范：所有通道包括控制通道都是 4 相捆绑数据

```
> petrify MUX4p.g -o MUX4p-csc.g -gcm -eqn MUX4p-gcm.eqn

  State coding conflicts for signal In1Ack
  State coding conflicts for signal In0Ack
  State coding conflicts for signal OutReq
The STG has no CSC.
Adding state signal: csc0
The STG has CSC.

> more MUX4p-gcm.eqn

# EQN file for model MUX4p
# Generated by petrify 4.0 (compiled 22-Dec-98 at 6:58 PM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 29.00
INORDER = In0Req OutAck In1Req Ctl CtlReq In1Ack In0Ack OutReq
          CtlAck csc0;
OUTORDER = [In1Ack] [In0Ack] [OutReq] [CtlAck] [csc0];
[In1Ack] = OutAck csc0';
[In0Ack] = OutAck csc0;
[2] = CtlReq (In1Req csc0' + In0Req Ctl');
[3] = CtlReq' (In1Req' csc0' + In0Req' csc0);
[OutReq] = OutReq [3]' + [2];       # mappable onto gC
[5] = OutAck' csc0;
[CtlAck] = CtlAck [5]' + OutAck;    # mappable onto gC
[7] = OutAck' CtlReq';
[8] = CtlReq Ctl;
[csc0] = csc0 [8]' + [7];           # mappable onto gC
```

从结果可以看到，由于有多个标识对应于相同的状态向量，所以这个STG并不满足CSC（完全状态编码），因此Petrify会自动地加入一个内部状态信号csc0。当CtlReq-到来之后，布尔信号Ctl不再有效，但此时MUX控制电路还没有结束它的工作。如果电路不能够从它的输入信号中得知电路下一步的动作是什么，则需要一个中间状态变量来保持这个信息。信号csc0是低电平有效的：如果Ctl = 0，当CtlReq+到来时，则csc0为低电平，而当CtlReq和OutAck都为低电平时，csc0又再次回到高电平状态。当处理复位电路时，在所有的通道处于空闲状态时，应当紧记信号csc0处于高电平，详见6.5节。

至于如何加入状态变量的具体细节可以从STG中得知，这个STG中包含的csc0是在综合电路的逻辑表达式之前由petrify加入的。

有时候可以通过重组电路中的信号跳变方式来避免增加状态变量。当然，哪种方法更好并不总是那么明显。原则上，较多的并发性会改进电路的性能，但同样也可能会使电路的状态空间更大，常常导致电路规模更大、速度更慢。在考虑电路的性能时，还必须注意与外部环境交互的问题。还有很大的空间来探索其他的解决方法。

在图6.26中，我们通过信号跳变排序In0 Ack/In1 Ack及Ctl Ack (In0 Ack+ ≺ Ctl Ack+, In1 Ack+ ≺ Ctl Ack+, …)的方式来除去多路选择STG中的某些并发情况，此时这个STG满足CSC且电路具有更小的面积：

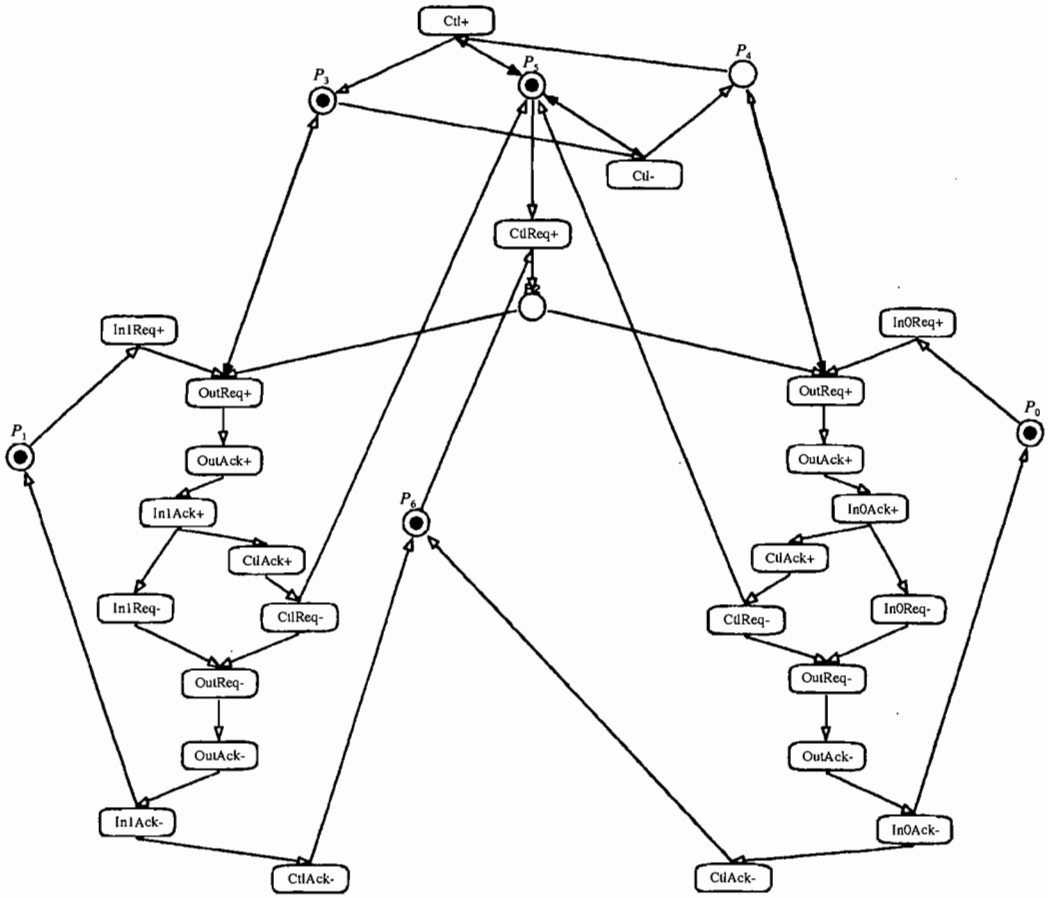


图6.26 4相捆绑数据多路选择器控制电路改进的STG规范

```
> more MUX4p-gcm.eqn

# EQN file for model MUX4pB
# Generated by petrify 4.0 (compiled 22-Dec-98 at 6:58 PM)
```

```
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 27.00

INORDER = In0Req OutAck In1Req Ctl CtlReq In1Ack In0Ack OutReq CtlAck;
OUTORDER = [In1Ack] [In0Ack] [OutReq] [CtlAck];
[0] = Ctl CtlReq OutAck;
[1] = Ctl' CtlReq OutAck;
[2] = CtlReq (Ctl' In0Req + Ctl In1Req);
[3] = CtlReq' (In0Ack' In1Req' + In0Req' In0Ack);
[OutReq] = OutReq [3]' + [2];       # mappable onto gC
[CtlAck] = In1Ack + In0Ack;
[In1Ack] = In1Ack OutAck + [0];     # mappable onto gC
[In0Ack] = In0Ack OutAck + [1];     # mappable onto gC
```

6.9 小结

本章给出了异步时序（控制）电路的设计基础，核心是速度无关电路和使用STG描述电路。所介绍的内容从实际应用出发，使读者能够设计出自己的速度无关控制电路。为了达到这一目的，考虑的面并不是太广，正如引言中提到的，忽略了一些其他重要的设计方法，包括群发模式和基本模式电路。

第7章　高级4相捆绑数据协议和电路

前面的章节介绍了异步电路设计的基础知识。本章我们将详细讨论4相捆绑数据协议和电路，其主要内容包括：(1) 通道的各种类型；(2) 具有不同数据有效性方案的协议；(3) 一些更复杂的锁存控制电路。介绍这些锁存控制器的意义在于：它们对电路面积、功耗以及速度方面的优化会有很大帮助；并可以作为用STG技术对控制电路进行描述和综合的好例子。

7.1　通道和协议

7.1.1　通道类型

到目前为止，我们只讨论了推（push）通道，在这种通道中，发送端作为主动方，发起数据的通信，而接收端则是被动方。与之相反，接收端也可以作为主动方，发起数据的通信，而发送端成为被动方，这样的通道称为拉（pull）通道。没有数据传送，只进行握手的通道称为“纯握手”（nonput）通道，这种通道常用作同步化。另外，除了发送应答信号，接收端还传送数据到发送端，这种通道称为双向（biput）通道。在4相捆绑数据协议中，接收端把数据与应答信号捆绑在一起；在4相双轨协议中，被动方通过返回一个码字而非一个简单的应答信号来响应请求动作。图7.1中给出了捆绑数据协议中的4种通道类型（纯握手、推、拉和双向）。当然，每种通道类型都可以采用任意的握手协议（2相或4相）以及任意的数据编码方式（捆绑数据、双轨、*m*-of-*n*等）。

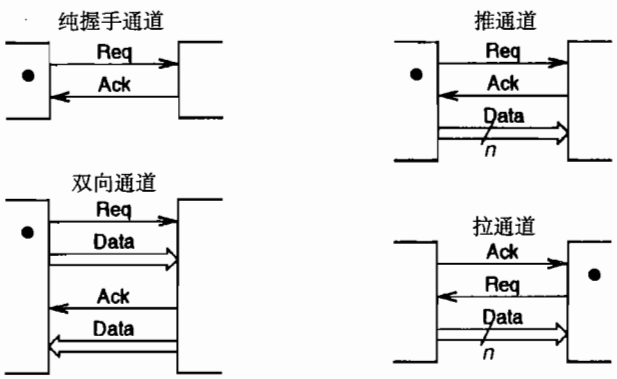


图7.1　4种基本的通道类型：推通道、拉通道、纯握手通道、双向通道

7.1.2　数据有效模式

对于捆绑数据协议，还应定义数据有效的时间段，图 7.2 给出了几种不同的情况。

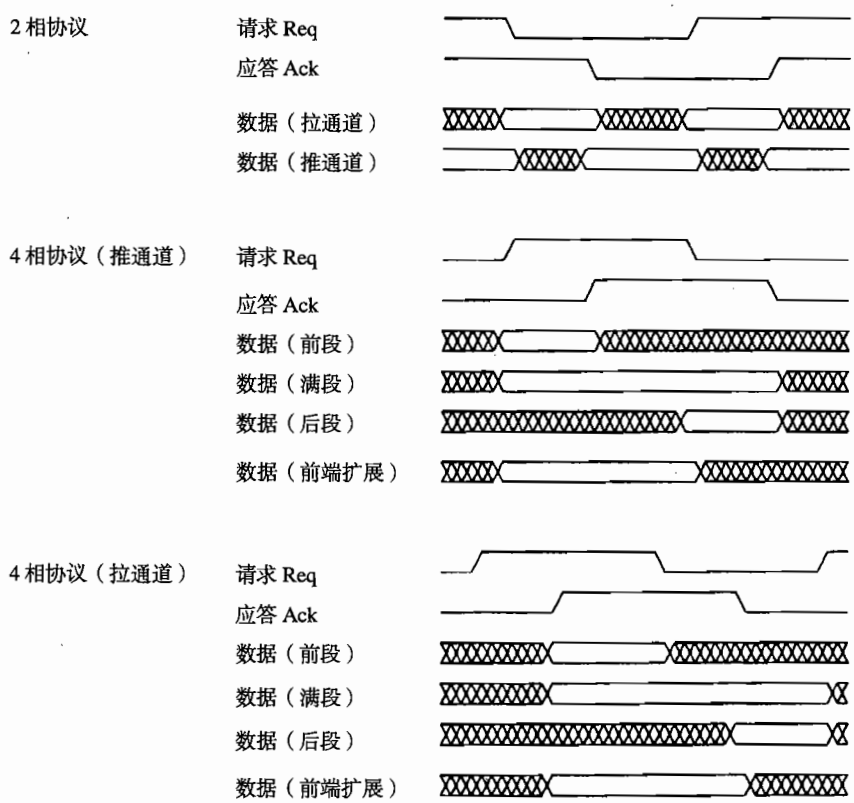


图 7.2　2 相、4 相捆绑数据的数据有效模式

对于推通道，请求信号表示的信息是“这是为你准备好的新数据”，而应答信号表示的信息则是“谢谢，我已经接收到数据，你可以释放数据线了”。类似地，对于拉通道，请求信号表示的是“请发送新数据”，而应答信号表示的信息则是“这就是你要的数据”。请求和应答线上的信号跳变可以按照上面这种方式来理解。在一个 4 相握手过程中，请求和应答线上都会包含 2 个跳变，无论请求和应答信号是上升跳变还是下降跳变，按照上述方式理解，可以得到以下几种数据有效模式：early（前段）、broad（满段）、late（后段）和 extended early（前段扩展）。

而 2 相握手中没有多余的信号跳变，因此 2 相握手中的每种通道类型（推或拉）只有一种数据有效模式，如图 7.2 所示。

对于所有的数据有效模式，在时间段开始前数据就已经有效了，而直到时间段结束后数据还要保持一段时间。除此之外，所有的数据有效模式还体现了接收数据方的需求。实际上应答

信号表示“谢谢，我已经接收到数据，你可以释放数据线了”时，并不意味着这个动作实际发生了，发送方可能会延长数据有效的时间，有时候这段延长的时间对接收方有很大的意义。

典型的例子如图7.2所示的前段扩展数据有效模式。在推通道中数据有效区间开始于Req↑到来之前的某个时刻而终止于Req↓到来后的某个时刻。

7.1.3 讨论

上面所述的有关通道类型和握手协议的分类大部分都来自于Peeters的博士论文[112]。对于通道类型、握手协议以及数据有效模式的选择会直接影响握手单元电路的面积、速度和功耗等多个方面。就如在设计中可能会采用由不同的捆绑数据及双轨协议构成的混合方式一样，也可以使用由不同通道类型和数据有效模式构成的混合方式。

例如，采用满段或前段扩展数据有效模式的4相捆绑数据推通道，可以非常方便的输入到由预充电CMOS电路构成的功能块。由于满段或前段扩展数据有效模式能够保证输入数据在评估阶段是稳定的，请求信号可以直接控制预充电和评估晶体管。

在4相捆绑数据设计中，另一种较好的选择是：在输入通道上采用满段数据有效模式的功能块，在输出通道上采用后段数据有效模式的功能块。对于这种情况，电路可以采用一个对称的延时元件来匹配，这个延时元件的延时只是组合电路延时的一半，也就是用Req↑和Req↓的总延时匹配组合电路的延时，且Req↓表示输出数据有效。在文献[112]的第46页中，这种实现方式因为握手的归零部分不再是多余的，而被称为“真正的单相”实现。这一方法还暗含连接到功能块的元件的实现。

关于在什么时候、什么情况下如何选择的问题的探讨，已经超出本书的讨论范围，有兴趣的读者可以参考文献[112]第77页。

7.2 静态类型检查

像编程时要考虑数据类型一样，考虑通道类型和数据有效模式的选择对设计电路是非常有帮助的，并且可以通过以下问题来对所设计电路的某些静态类型进行检查：“握手元件的输入端口上允许何种类型的输入数据？”“握手元件的输出端口上会得出何种类型的输出数据？”后面这个问题的回答可能会依赖输入端口的数据类型。文献[104]中给出了与之相类似的2相非重叠时钟同步电路的类型检查实例，并且被用于Genesil硅编译器中[67]。

图7.3是4相捆绑数据推通道的4种数据有效性类型的层次图：“满段”是最强的类型并可以替代任何较弱类型作为电路的输入。“前段扩展”可用于只要求“前段”类型的情况。对于对握手（功能块、汇合、分支、并入、多路选择器、多路分配器）透明的电路，其输出类型的强度不能超过输入中最弱的类型。一般来讲，输入和输出的类型应该是一样的，但也

有例外的情况，锁存器是能够使输出类型比输入类型强的唯一电路。接下来让我们看看以下例子：

- “汇合”连接两个“前段扩展”类型的输入产生“前段”类型的输出。
- 从图 6.21 中的 STG 图可知，前面章节（见图 2.9）介绍的简单 4 相捆绑数据锁存控制器是，采取“前段”类型的输入，产生“前段扩展”类型的输出。
- 在 6.8.3 节中的 4 相捆绑数据多路选择器设计中，其控制输入为“前段扩展”类型（图 6.25 中的 STG 指定了从 CtlReq+ 到 CtlReq- 的稳定输入）。

希望读者能够继续这部分的训练，最好画出相关的时序图，从时序图中可以推导出输出的类型。这里所讲的类型检查对于调试那些存在非法行为的电路非常有用。

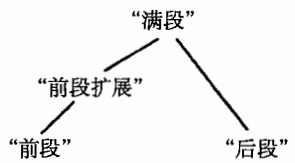


图 7.3　4 相捆绑数据协议的 4 种数据有效模式的层次排序

7.3　更高级的锁存器控制电路

在前面的章节中，我们只讨论了由 C 单元和反相器组成的锁存控制器来控制 4 相捆绑数据握手锁存器，如图 2.9 所示。在文献[41]中，这种锁存控制器被称为“简单锁存控制器”，而在文献[77]中它被称为“非去耦锁存控制器”。

如果流水线或 FIFO 是由这种简单锁存控制器构成的，那么每两个握手锁存器中，一个锁存器存储有效托肯，而另一个锁存器存储空托肯，如图 7.4(a)所示——这个流水线的静态扩展 $S=2$。

这个托肯图可能会有点令人费解。图中的空托肯对应于握手的归零部分，而在实际中锁存器并不会“存储空托肯”——它们是透明的，这表示浪费了一定的硬件资源。

理想状态下，设计者都希望如图 7.5 那样，每个电平触发锁存器都能存储一个有效托肯，并且只要在接口的数据流中“加入”空托肯，从而使其作为握手的一部分。在文献[41]中，Furber 和 Day 讨论了两种改进型 4 相捆绑数据锁存器控制电路的设计：半去耦锁存控制器和完全去耦锁存控制器。除了这两种特殊电路外，文献[41]中还介绍了基于第 6 章介绍的 STG 设计的控制电路。这 3 种锁存控制器有以下特性：

- 对于简单或非去耦锁存控制器，只有当输出通道上的前一个握手已经完成后才允许锁存新的输入数据，即在$A_{out}\downarrow$到来之后。此外，输入和输出通道上的握手相互影响：$R_{out}\uparrow \preceq A_{in}\uparrow$和$R_{out}\downarrow \preceq A_{in}\downarrow$。
- 半去耦锁存控制器在某种程度上放宽了要求：新输入可以在$R_{out}\downarrow$到来后被锁存，且控制器可能独立于输出通道的握手来产生$A_{in}\uparrow$——输入和输出通道之间的相互影响已经减弱：$A_{out}\uparrow \preceq A_{in}\uparrow$。
- 完全去耦锁存器控制器则进一步放宽要求：新输入可以在$A_{out}\uparrow$到来后被锁存（即只要下级锁存器表明已经锁存了当前数据），且输入通道可以在不与输出通道有任何交互作用的情况下完成握手。

简单锁存控制器的另一个潜在缺点是：它无法利用具有非对称延时的功能块（4.4.1节中介绍过）。而文献[41]中给出的完全去耦锁存控制器则没有这个问题。由于输入和输出通道的去耦，相关图中决定电路周期P的关键环路只访问与两个相邻流水线级相关的节点，使得周期变为最短（参考4.4.1节）。表7.1对简单、半去耦和完全去耦锁存控制器的特性进行了总结。

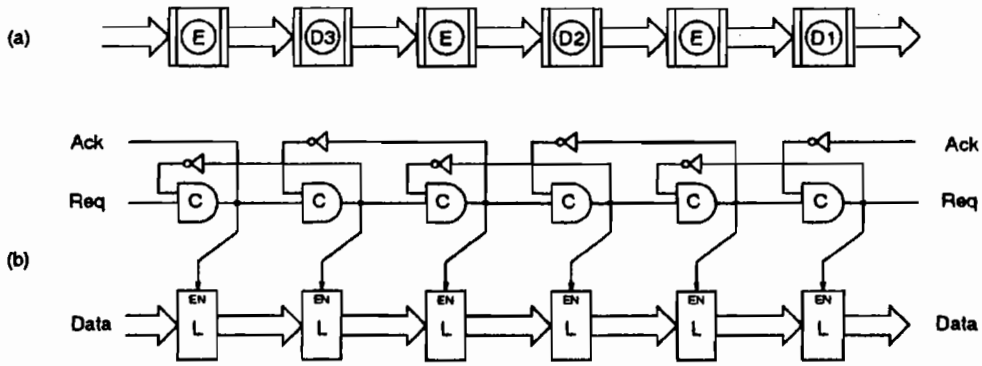


图7.4 (a) 基于握手锁存器的FIFO; (b) 用简单锁存控制器和电平触发锁存器实现。FIFO中每个锁存器都有一个有效托肯，当EN = 0时，锁存器为“透明的”；当EN = 1时，锁存器为“非透明的”（拥有数据）

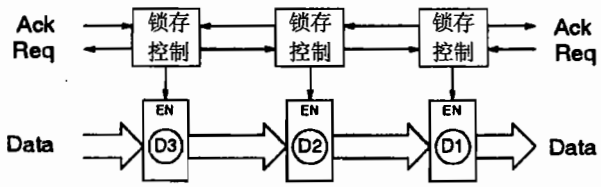


图7.5 FIFO：当FIFO为满时，每个电平触发锁存器都包含一个有效数据。文献[41]中的半去耦锁存控制器和完全去耦锁存控制器允许这种行为

表 7.1　文献[41]中所给控制器特性总结

锁存控制器	静态扩展 S	周期 P
简单	2	$2L_r + 2L_{f.v}$
半去耦	1	$2L_r + 2L_{f.v}$
完全去耦	1	$2L_r + L_{f.v} + L_{f.E}$

以上提到的所有锁存控制器都是“常透明的”，由于输入在稳定前的多次跳变会通过几级连续的流水线，这将会引起更多的功耗。如果使用“常不透明”锁存控制器，其中的每个锁存器都可以看作是一个障碍物。如果含有气泡的握手锁存器的输入端有托肯，则锁存控制器会把锁存器切换到透明模式；当输入数据已经安全传输到锁存器时，锁存控制器关闭锁存器使其回到不透明模式，此时锁存器将会保存这个数据。在文献[23]中提到的异步 MIPS 处理器的设计过程中，我们用常不透明锁存控制器代替常透明锁存控制器，实验结果表明电路的功耗降低了近 50%。

文献[23]中给出的常不透明锁存控制器的 STG 描述和电路实现（如图 7.6 所示）。从 STG 中可以看到输入和输出通道之间有很强的交互作用，但是决定电路周期的相关图关键环路只访问与两个相邻流水线级有关的节点并且周期变为最短。为了确保输入信号在Rout+到来之前通过锁存器，有必要在 Lt+ 到 Rout+ 的路径上加入延时。此外，使锁存器为透明的 Lt = 0 脉冲的持续时间是由锁存控制器本身的门延时来决定的，而且为了确保锁存器安全锁存数据，这个脉冲的持续时间必须足够长。锁存控制器输入通道采用满段数据有效模式，而其输出通道也为满段数据有效模式。

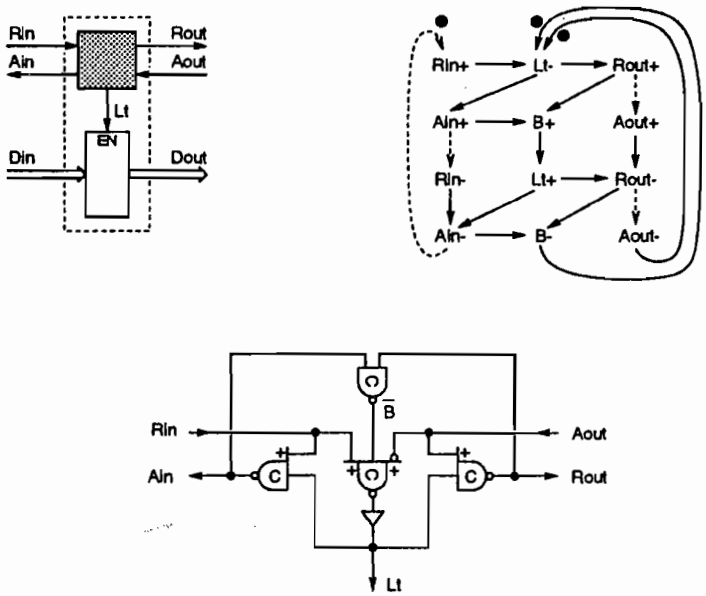


图 7.6　文献[23]中给出的常不透明锁存控制器的 STG 描述和电路实现

7.4 小结

本章简单介绍了通道类型、数据有效模式以及锁存控制器的选择，主要目的是介绍一些基础知识、选择方式以及优化电路的可能性。有兴趣了解更多细节的读者可阅读参考文献。

最后要注意的是，在第3章中提到的异步电路静态数据流图（即由锁存控制器之间的握手来控制有效托肯和空托肯的向前复制）和第4章的性能分析中，都假定所有的握手锁存器采用的是简单常透明锁存控制器。而要使用半去耦或完全去耦锁存控制器，还必须对数据流图做一些修改并重新进行性能分析。因此最好用一对简单锁存控制器代替半去耦或完全去耦锁存控制器。另外，如果使用了半去耦锁存控制器和完全去耦锁存控制器，则每个环路只需要包含两个握手锁存器。

第8章　高级语言和工具

本章介绍用于异步电路高层建模与综合的语言和CAD工具。主要介绍基本概念和有代表性的设计方法。感兴趣的读者可以参考本书的其他章节（第二部分和第13章）以及本书给出的参考文献。本章最后一节将介绍VHDL在异步电路设计中的应用。

8.1　引言

异步电路高层建模与综合中使用的语言几乎都是属于CSP族的，而不是采用工业中的标准硬件描述语言（VHDL或Verilog）。异步电路具有高并发性，并且用握手通道来完成模块间的通信，因此，用于异步电路设计的硬件描述语言应该能够支持这两个特性。由Hoare[57, 58]提出的CSP语言能够满足这些要求。CSP是“通信顺序进程”（Communicating Sequential Processes）的缩写，它最核心的特性是：

- 并发进程；
- 在一个进程内，既有顺序语句又有并行语句；
- 用点对点通道传送同步信息[由发送、接收机制——可能性-探针支持（possibly-probe）]。

CSP是并行系统编程语言中的一种，并行系统编程语言包括OCCAM[68]、LOTOS[108, 16]和CCS[89]，以及专门用于设计异步电路的语言Tangram[142, 135]、CHP[81]和Balsa[9, 10]。Tangram和Balsa的详细描述分别见本书第三部分的第13章和第二部分。

本章首先详细介绍支持通信和并发的CSP语言结构，其中包括用来说明这类语言风格的几个例子，接下来我们将简要地介绍两种基于CSP类（CSP-like）程序的不同设计方法：

- 在荷兰飞利浦公司的实验室，van Berkel，Peeters和Kessels等开发了一种名为Tangram的专用语言以及相应的硅编译器[142, 141, 135, 112]。使用面向语法（syntax-directed）的编译过程，综合工具将Tangram程序映射成由握手元件组成的电路结构。采用这些工具，飞利浦实验室已经设计出不少重要的异步芯片[137, 138, 144, 73, 74]。其中最新的设计成果是一个智能卡电路，我们将在第13章中介绍。
- 在加州理工学院，Martin开发了一种名为通信硬件进程（Communications Hardware Processes——CHP）的语言以及相应的设计流程工具，这套工具支持部分手动和部分自动的设计流程，其目标是能够对QDI（准延迟无关）4相双轨电路[80, 83]的晶体管级实现进行高级优化。

CHP 的语法和 CSP 类似，它们都使用各种专用符号，而 Tangram 的语法更像传统的编程语言，它使用关键字，但实质上两者都与 CSP 非常相似。

在本章的最后一节，我们将介绍一个 VHDL 程序包，它提供 CSP 类的信息传送，并讨论了基于 VHDL 的设计流程，该设计流程支持手工逐步求精的设计过程。

8.2 CSP 中的并发和信息传送

CSP 中的“顺序进程”是指每个进程由顺序执行语句构成的程序来描述，语句之间像其他许多程序语言那样用分号分隔。可以把分号视为将语句组合成程序的操作符。在这一点上，CSP 中的进程与 VHDL 中的进程非常相似。然而，CSP 还允许在进程中出现并行语句。用符号“||”表示并行语句。VHDL 没有这个性质，而 Verilog 中的 fork-join 结构允许在进程内出现并行语句。

CSP 中的“通信”表示采用点对点通道传送同步信息，如图 8.1 所示，其中的两个进程 P1 和 P2 用通道 C 连接在一起。通过发送语句 C!x，进程 P1 把变量 x 的值传送（用符号“!”表示传送）到通道 C 上，并且再通过一个接收语句 C?y，进程 P2 接收（用符号“?”表示接收）通道 C 上的值并把这个值赋给变量 y。通道 C 是无记忆的，且把 P1 中的变量 x 的值传送到 P2 的变量 y 是一个原子行为。这使得进程 P1 和 P2 同步化。这两个进程中无论哪个先到来都必须等待另外一方，并且发送和接收语句在同一时刻完成。用术语“约会”（rendezvous）来表示这种同步。

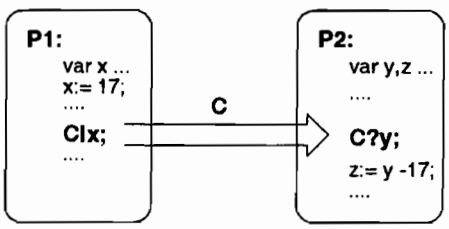


图 8.1 两个进程 P1 和 P2 由通道 C 连接。进程 P1 发送其变量 x 的值到通道 C，进程 P2 接收该值并把它赋给它的变量 y

当进程执行发送（接收）语句时，它开始通信并挂起，直到通道另外一端的进程执行它的接收（发送）语句。这不一定是可取的，Martin 已经扩展了一个具有探针（probe）结构的 CSP[79]，这个结构允许处于通道被动端的进程探测是否有某个通信挂在通道上，而不用提交任何信息。探针是一个将通道名作为它的自变量并返回一个布尔值的函数。用探针探测通道 C 的语法表示是 $\overline{C}$。

顺便提一下，一些用于并发系统编程的语言都假设通道具有（可能是无限的）缓冲能力。这意味着通道被视为一个 FIFO，且当通道进行通信时通信进程没有同步化，这种形式的通信称为异步信息传送。

再回到同步信息传送。显然，无记忆通道是用带有同步通信进程协议的一组导线来实现的，它可以是前面章节中的任何一种协议。因此，同步信息传送是非常有用的语言结构，它支持异步电路的高层建模，而不考虑数据编码和握手协议的具体细节。

遗憾的是，VHDL 和 Verilog 语言都不支持这种原语。虽然可以通过编写低级代码来实现握手，但是把这种低级代码混入那些用于描述电路高级行为的代码中去却是很不可取的。

在下一节中，我们将通过几个用 Tangram 语言编写的小程序来说明这种语言的风格，在后面也将用它们来说明面向语法编译。程序的源代码、握手电路图以及部分文本都是由飞利浦公司的 Ad Peeters 友情提供的。

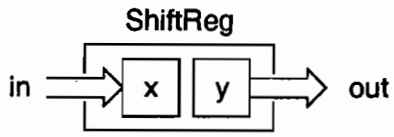
曼彻斯特大学最近开发了一种与 Tangram 语言相似的语言和综合工具，这些成果是公开的[10]，我们将在本书的第二部分进行介绍，与此相关的工作见文献[17]和文献[21]。

8.3　Tangram：程序实例

本节给出了几个简单的 Tangram 程序实例：2 位移位寄存器、2 位行波 FIFO 和最大公约数函数。

8.3.1　2 位移位寄存器

图 8.2 给出了一个名为 ShiftReg 的 2 位移位寄存器的代码。它是一个输入通道为 In、输出通道为 Out 的进程，输入、输出通道上变量的类型为[0..255]，有两个局部变量 x 和 y，其初始值都为 0。进程执行由 3 条顺序语句组成的一个无限循环：out!y; y:=x; in?x。

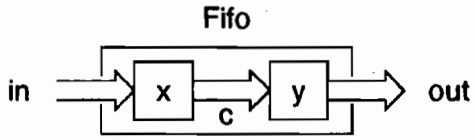



```
  T = type [0..255]
& ShiftReg : main proc(in? chan T & out! chan T).
  begin
  & var x,y: var T := 0
    |
    forever do
      out!y ; y:=x ; in?x
    od
  end
```

图 8.2　一个 2 位移位寄存器的 Tangram 程序

8.3.2 2位（行波）FIFO

图8.3给出了一个名为FIFO的2位先进先出缓冲器的Tangram程序。它可理解为由通道c连接的两个并行运行的1位缓冲器。初看之下，它很像前面那个2位移位寄存器，但进一步分析发现这个FIFO具有更好的灵活性和更高的并发性。




```
  T = type [0..255]
& Fifo : main proc(in? chan T & out! chan T).
  begin
  & x,y: var T
  & c : chan T
  |
       forever do in?x ; c!x   od
    || forever do c?y  ; out!y od
  end
```

图8.3 一个2位（行波）FIFO的Tangram程序

8.3.3 使用while和if语句的GCD

图8.4所示为一个计算最大公约数模块的代码，这个例子来源于3.7节。“do x< >y then... od”是一个while语句，除了语法不同外，图8.4中所示的代码和图3.11中的代码是完全一样的。

这个模块有一个用于接收两个操作数的输入通道和一个用于输出结果的输出通道。

```
  int = type [0..255]
& gcd_if : main proc (in?chan <<int,int>> & out!chan int).
  begin x,y:var int ff
  | forever do
      in?<<x,y>>
    ; do x<>y then
        if x<y then y:=y-x
               else x:=x-y
        fi
      od
    ; out!x
    od
  end
```

图8.4 使用while和if语句的GCD的Tangram程序

8.3.4　使用哨命令的 GCD

图 8.5 是 GCD 的另一种实现方式。此时模块把两个操作数分到输入通道，且程序体是基于一个哨（guarded）命令的循环。这个哨循环可视为广义的 while 语句。语句一直循环，直到所有的哨都变为假。当至少有一个哨为真时，真的哨对应的指令被选定（确定性的或非确定性的）并执行。

```
  int = type [0..255]
& gcd_gc : main proc (in1,in2?chan int & out!chan int).
  begin x,y:var int ff
  | forever do
      in1?x || in2?y
    ; do x<y then y:=y-x
      or y<x then x:=x-y
      od
    ; out!x
    od
  end
```

图 8.5　使用哨循环 GCD 的 Tangram 程序

8.4　Tangram：面向语法编译

现在来讨论综合过程。设计流程采用的是基于握手电路的一种中间形式。前端设计称为 VLSI 编程，且使用面向语法的编译，Tangram 程序最后映射成一个握手单元的电路结构。从下面的例子中可以看出，Tangram 程序与握手电路之间是一一对应的。编译过程对于设计者来说是完全透明的，设计者完全是在 Tangram 程序级中进行设计。

设计流的后端包括编译器所指向的握手电路库和可以进行异步电路综合后窥孔（peephole）（即用一些更有效的握手单元结构来替代普通的结构）优化的工具。由于有许多握手电路库，允许使用不同的握手协议（4 相双轨、4 相捆绑数据等）和不同的实现技术（CMOS 标准元、FPGA 等）。握手单元可由以下几种方式来描述和设计：(i) 人工设计；(ii) 使用在第 6 章介绍过的 STG 和 Petrify；(iii) 使用 Martin 的基于转换方法中的初级方法，这将在下一节给出。

对于编译过程中具体内容的介绍已超出了本书的范围。我们将通过前面章节的一些 Tangram 程序所对应的握手电路来讲解“面向语法编译”的风格。

8.4.1　2 位移位寄存器

作为面向语法编译的第一个例子，图 8.6 给出了与图 8.2 所示 Tangram 程序对应的握手电路。

图中，握手单元用圆表示，而连接单元的通道则用弧表示。单元符号上的小点表示端口。空心点代表被动端口，而实心点代表主动端口。箭头表示数据传送的方向。纯握手通道中没有

数据的传送，因此它没有方向也没有箭头。从图8.6中可看出握手电路同时使用了推通道和拉通道。

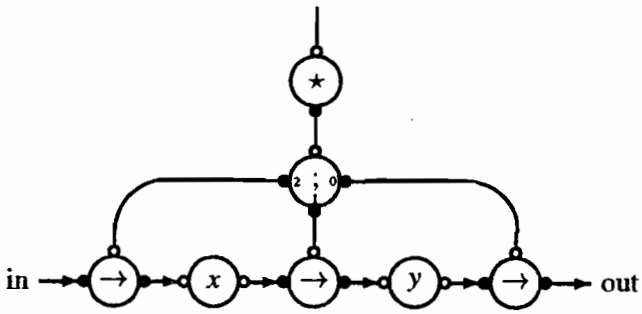


图 8.6　2位移位寄存器编译后的握手电路

这个程序的结构是一个 forever-do 语句，其中包含 3 条顺序执行语句（它们由分号分开）。每一条语句都是一种类型的赋值语句：变量 y 的值赋给输出通道 out，变量 x 的值赋给变量 y，输入通道 in 中接收到的值赋给变量 x。握手电路的结构也是完全相同的：

- 在顶部是一个实现 forever-do 语句的中继器（repeater）。中继器等待其被动输入端口上请求信号的到来，接着在它的主动输出通道运行一个无限握手循环，输入通道上的握手会一直进行。
- 接下来是一个 3 路顺序发生器（sequencer），它实现程序中“分号”的功能。当一个请求信号到达顺序发生器的被动输入通道时，按顺序在其每一个主动输出通道执行完全的握手（按照符号中的数字指示的顺序执行），最后在被动输入通道上完成握手。按照这种方式顺序发生器依次激活对应于 forever-do 中的各个语句的握手电路结构。
- 握手单元的最底下一行包括两个变量 x 和 y 以及 3 个由“→”表示的传送器（transferer），注意变量符号上只有被动的读、写端口。用传送器来实现构成 forever-do 程序体的 3 个赋值语句（out!y; y:=x; in?x）。每个传送器都先等待其被动纯握手通道（nonput channel）上的请求信号的到来，接着开始其拉输入通道上的握手，拉输入通道的握手被传递到推输出通道。用这种方式，传送器从其输入通道拉出数据并将它推到其输出通道上。最后，它在其被动纯握手通道上完成握手。

8.4.2　2 位 FIFO

图8.7给出了与图8.3所示Tangram程序对应的握手电路。在图8.7的握手电路中标有“psv”的单元被称为钝化器（passivator）。它与FIFO的内部通道 c 有关，并且实现主动发送器（c!x）和主动接收器（c?y）之间的同步和通信。

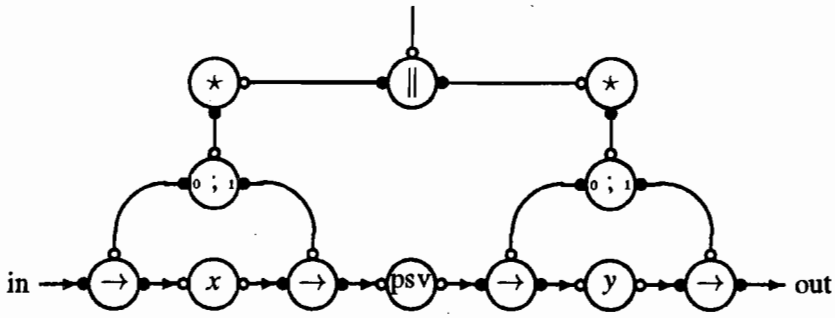


图 8.7　FIFO 程序编译后的握手电路

FIFO中优化后的握手电路如图8.8所示。用于实现数据路径中同步的钝化器被用于实现控制中同步的汇合（join）单元所替代。这个FIFO设计的握手电路，它的数据路径和图 8.2 所示的移位寄存器的数据路径是相同的，唯一不同的是在电路的控制部分。

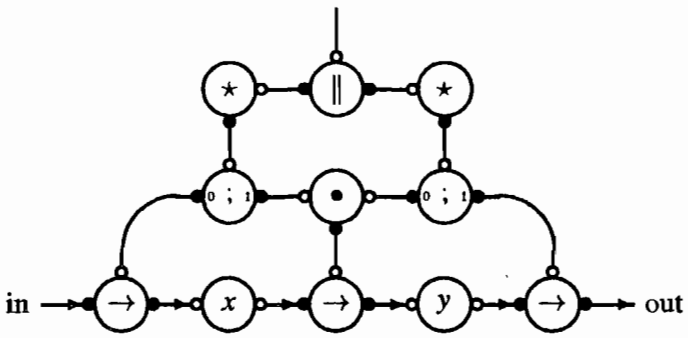


图 8.8　FIFO 程序优化后的握手电路

8.4.3　使用哨循环的 GCD

面向语法编译的更复杂的例子如图 8.9 所示，它是由图 8.5 中的 Tangram 程序编译后得到的握手电路。与前面的握手电路相比，GCD 程序的握手电路引入了两种新型的元件，下面将给出详细的描述。

首先，电路包含一个 bar 和一个 do 元件，二者都是与数据有关的控制单元。其次，握手电路中的元件不是与语言结构直接对应的，而是进行了一些共享：多路选择器（由 mux 表示），多路分配器（由 dmx 表示）和分支单元（由●表示）。

注意，Tangram 中的分支（fork）与图 3.3 中的分支是相同的，但 Tangram 中的多路选择器和多路分配器却与图3.3中的多路选择器和多路分配器不同。Tangram多路选择器和图3.3中的“并入”相同，而Tangram多路分配器是“反相的并入”，它的输出端口是被动的，要求两个输出端的握手是互斥的。

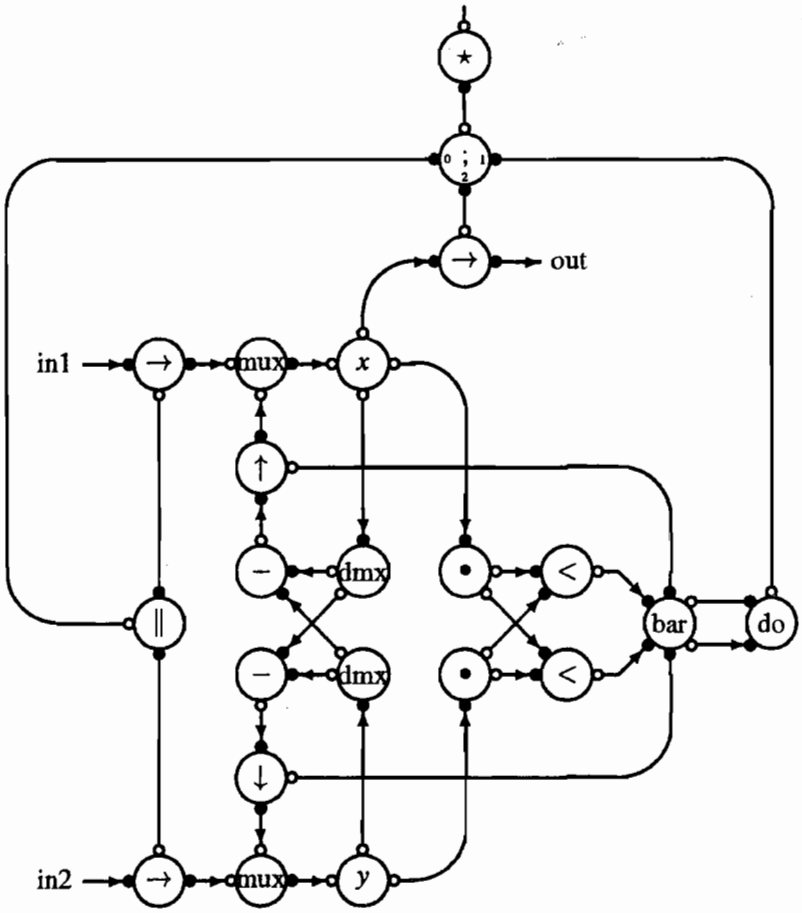


图 8.9 使用哨循环的 GCD 程序编译后的握手电路

bar 和 do 单元： do 和 bar 单元用于实现具有两个哨的哨命令结构，其中 do 单元实现重复部分（do od 部分，包括这两个哨的分支判定），bar 单元实现选择部分（命令的 then or then 部分）。

do 单元当其被动端口被激活后，它首先通过其主动数据端口的握手收集所有条件分支的值。当收集到的值为真时，它激活其主动纯握手端口（去激活哨命令），完成后开始一个新的判定循环。当收集到的值为假时，do 单元通过它的被动端口完成握手来结束其运行。

bar 单元可以通过它的被动数据端口或者被动控制端口被激活（例如，do 单元按顺序进行这两种激活）。当通过数据端口激活时，它通过与主动数据端口的握手收集两个哨值，并在其被动数据端口分别发送这些值，这样就完成了握手。当通过控制口激活时，bar 单元激活一个主动控制端口，该控制端口与最近的数据循环中返回真值的数据端口有关（为了简化，这种选

择通常以确定的方式实现，尽管从编程角度来说并不要求这样）。可以看出bar单元可以组合成树或列表来实现一个任意长度的哨命令序列。而且，不是每一个数据循环都需要控制循环。

mux, demux 和 fork 单元：图 8.4 中的 GCD 程序中有两次给 x 变量赋值，即输入 in1?x 和赋值 x:=x-y。在图 8.9 所示的握手电路中，Tangram 变量 x 的这两次赋值被多路选择器汇合，从而到达握手变量 x 的写端口。

变量 x 作为一个表达式在程序中出现了 5 次，在输出表达式 out!x 中出现一次，在哨表达式 x<y 和 y<x 中出现两次，在赋值表达式 x-y 和 y-x 中出现两次。可以通过握手变量 x 的 5 个读端口来实现对这 5 个值的观察。这一实现见文献[135]（第 34 页图 2.7）中的握手电路。在图 8.9 中，给出了另一种编译结果，此时握手变量 x 具有 3 个读端口：

- 一个用于输出行为中事件的读端口。
- 一个用于哨表达式的读端口。表达式的判定是互斥的，因此可以用一个同步 fork 单元来组合。
- 一个用于赋值表达式的读端口。这个赋值过程也是互斥的，因此可以用一个多路分配器来组合。

在第 13 章中，将进一步详细讨论 GCD 的例子。

8.5　Martin 的转换过程

Martin 和他的小组（在 Caltech）对异步设计做出了基础性的贡献，并影响了许多其他研究者。这些方法在 Caltech 被用来设计了几个重要芯片，最新也是最引人注意的是一个异步 MIPS R3000 处理器[88]。以下将要介绍的设计流程给我们这样一个启示：这个设计过程如此精巧而成熟，对那些在 Caltech 工作组工作的设计者来说这可能是他们唯一的选择。

主要的手动设计过程包括以下几个步骤（语义保持性变换）。

1. **进程分解：**把每个进程提炼成一组相互作用的简单进程。这个步骤不断重复，直到所有的进程都简单到可以在下一步骤处理。
2. **握手扩张：**通信通道用具体的导线来代替，通信行为（例如发送或接收）用协议所要求的信号转换来代替。例如，一个接收语句表示为

$$C?y$$

被一系列简单语句代替，例如，

$$[\overset{\circ}{C}_{req}];\ y{:=}\text{data};\ C_{ack}\uparrow;\ [\neg C_{req}];\ C_{ack}\downarrow$$

这组语句可以这样理解："等待请求信号变为高"，"读取数据"，"驱动应答信号为高"，"等待请求信号变为低"，以及"驱动应答信号为低"。

在这一级中，可能有必要增加状态变量和（或）调整信号变换，以获得类似于第6章所述的CSC的条件。

3. **生成规则展开：**用一组生成规则（或哨命令）来代替握手展开，例如，

$$a \wedge b \mapsto c\uparrow \text{ 和 } \neg b \wedge \neg c \mapsto c\downarrow$$

每个生成规则中包括一个条件和一个动作，当条件为真时执行相应的动作。在这里顺便提一下，以上两个生成规则同样可以表达6.4.1节中信号c的置位和复位功能。生成规则确定了进程的内部信号和输出信号的行为特性。生成规则本身是简单的并发进程，而且哨必须确保按程序的顺序进行信号转换（即保持原来的CHP程序的语义）。这可能要求加强哨。除此之外，为了得到更简单的电路实现，条件可能被修改并使之更为对称。

4. **操作简化：**把生成规则分组，而且把每个组映射成一个基本的硬件单元，如一个通用的C单元。上述两个生成规则将被映射成通用的C单元，如图6.17所示。

8.6 使用VHDL进行异步设计

8.6.1 引言

在这一节中，我们将介绍两个VHDL程序包，这些程序包给设计者提供了在进程之间传送同步信息的机制——类似于CSP语言族中的结构（发送、接收和检测）。

这一素材是在M.Sc.项目中开发出来的，并用于32位浮点ALU的设计[110]，该ALU采用的是IEEE浮点数表示法，后来还被用于异步电路设计课程中。还有一些与VHDL包和方法相关的研究[95, 118, 149, 78]。

下面介绍的通道程序包，它只支持使用32位4相数据捆绑推协议的通道。由于VHDL允许过程和函数的重载，因此可以用任何数据类型定义通道。所要做的只是一些剪切和粘贴之类的操作。如果要想支持4相数据捆绑推协议外的其他协议，还需对程序包做进一步的扩展。

8.6.2 VHDL和CSP类语言

前面章节中已经介绍了几种类似于CSP的用于异步设计的硬件描述语言，这些语言的优势主要是它们支持并发和同步信息传送，语言结构的有限性与明确定义也会使面向语法的编译相对简单。

设计者并非不能用工业标准语言VHDL（或Verilog）去设计异步电路。其实这些语言中的一些基本概念（并发进程和信号事件等）都“非常适合”异步电路的建模与设计。图8.10说明了如何用简单的VHDL语言来描述图8.2所示的Tangram程序。除了说明可行性以外，图8.10

还突出了采用VHDL对异步电路建模的局限性：大部分VHDL代码表达的是低层的握手细节，对电路功能的描述很杂乱。

```
library IEEE;
use IEEE.std_logic_1164.all;

  type T is std_logic_vector(7 downto 0)

  entity ShiftReg is
    port ( in_req   : in  std_logic;
           in_ack   : out std_logic;
           in_data  : in  T;
           out_req  : out std_logic;
           out_ack  : in  std_logic;
           out-data : out T );
  end ShiftReg;

  architecture behav of ShiftReg is
  begin
    process
      variable x, y: T;
    begin
      loop
        out_req <= '1' ;              --  out!y
        out_data <= y ;
        wait until out_ack = '1';
        out_req <= '0';
        wait until out_ack = '0';
        y := x;                       --  y := x
        wait until in_req = '1';      --  in?x
        x := in_data;
        in.ack <= '1';
        wait until ch_req = '0';
        ch_ack <= '0';
      end loop;
    end process;
  end behav;
```

图 8.10　图 8.2 所示 2 位移位寄存器 FIFO 状态的 VHDL 描述

VHDL显然缺乏CSP类语言中的那种在通道上传送同步信息的内在机制。CSP族语言具有的另一个特性是进程中的语句层的并发性。当然，采用诸如VHDL的工业标准硬件描述语言也有如下一些优点：

- 这些语言能够很好地得到现有的CAD工具体系的支持，提供仿真器、基本设计模块、混合模式仿真，以及一些用于综合、布局和反向标注时序信息的工具；
- 对某种目标工艺，从顶层描述到底层实现的整个过程，可采用同一仿真器和测试平台，如单元标准布局；

- 在某些实体用行为描述来建模，而另一些实体用目标工艺的元件来实现的情况下，可以进行混合模式仿真；
- 许多实际系统既包括同步子系统又包括异步子系统，像这样的混合系统用VHDL建模不会有任何问题。

8.6.3 通道通信和设计流程

讨论以下设计流程的动机来自于上面提到的那些优点，目的是使VHDL具有CSP类语言的通道通信机制，即过程`send(<channel>,<variable>)`和`receive(<channel>,<variable>)`以及函数`probe(<channel>)`。另一个目的是对于通道的一端连接用电路实现的实体，而另一端连接用具有上述通信机制的采用行为描述的实体的情况下，仍然可以进行混合模式仿真，如图8.11(b)所示。这样就能够支持人工的自顶向下逐步求精的设计过程，从高层描述到低层实现的整个过程中使用同一测试平台，如图8.11(a)至图8.11(c)所示。

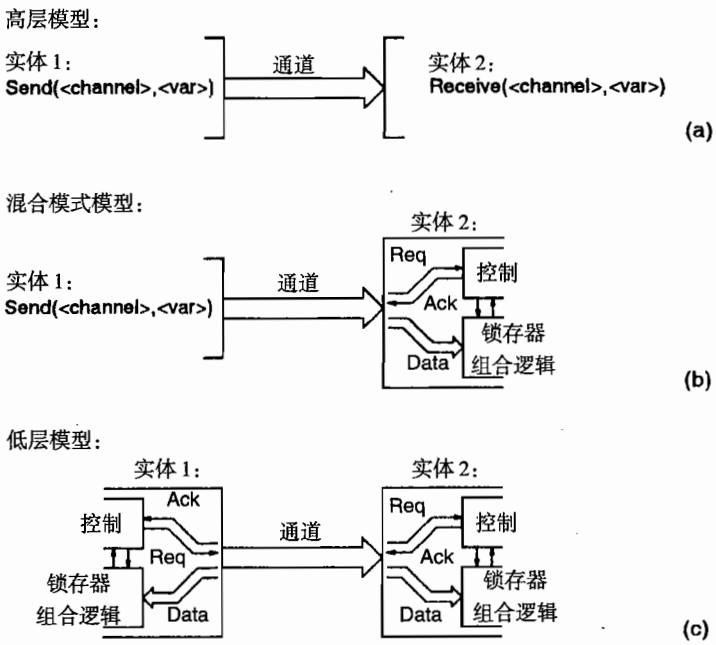


图 8.11 通道通信支持高层、混合模式和门级/标准单元仿真的VHDL程序包

在VHDL语言中，所有进程间通信都是通过信号发生的，因此通道不得不声明为信号，最好是一个通道对应一个信号。对于推通道，由于发送端驱动通道的请求和数据，而接收端驱动通道的应答信号，这样，一个信号有两个驱动器。如果信号是确定的，这在VHDL语言中是允许的。因此，可以把通道类型定义为记录类型，它具有请求、应答和数据域，然后为通道类型

定义了一个判决函数，该函数将决定通道的结果值。这种类型的通道，带有独立的请求域和应答域，被称为实际通道（real channel），将在 8.6.5 节中介绍。在仿真时，每个通道将有 3 条线分别显示请求信号、应答信号和通信数据的波形。

通道也可以被定义为仅有两个域：一个描述握手的状态［称为“握手相”（handshake phase）或简称为“相”］，另一个包含数据。相域的类型是枚举类型，它的值是通道所假设状态的握手相，其值可以被发送端和接收端所驱动。这种类型的通道被称为“抽象通道”（abstract channel）。在仿真时，每个通道有两条轨迹，很容易看出通道假设的相和传递的数值。

以上程序和定义被分成两个 VHDL 程序包：一个命名为 `abstpack.vhd`，用于仿真高层模型；另一个命名为 `realpack.vhd`，用于所有层的设计。完整的列表见本章末的附录 8.A。由这些包支持的设计流程如下所示：

- 电路及其外围环境（或者测试平台）首先用抽象通道来建模和仿真。只需在顶层设计单元中加入声明：“`usepackage work.abstpack.all`”。
- 接着把电路分解成更简单的实体。这些实体仍然采用通道进行通信，使用抽象通道程序包进行仿真，这一步可能需要重复进行。
- 在顶层设计单元，设计者改用“`usepackage work.realpack.all`”，就可改为使用实际通道包。除这个简单的变化外，VHDL 源代码是一样的。
- 可以把实体划分为控制电路部分（按第 6 章的说明设计）和数据电路部分（包括普通的锁存器和组合电路）。也可以进行图 8.11(b)所示的混合模式仿真。控制电路的仿真模型既可以是目标工艺上电路的实际实现，也可以是包含一组并行信号赋值的实体——例如由 Petrify 生成的布尔方程。
- 最后，当所有的实体都被划分成控制和数据，且所有的子实体都用目标工艺元件实现时，则设计完成。可以使用标准工艺映射工具来完成电路的实现，电路可以利用反向标注时序信息来进行仿真。

注意，在从高层描述到使用目标工艺元件的低层实现的整个设计过程中，都使用同一仿真测试平台。

8.6.4　抽象通道包

用称为 fp 的数据类型（用 32 位标准逻辑向量来表示 IEEE 浮点数）来定义图 8.12 中的抽象通道。实际通道类型为 `channel_fp`。需要为设计中使用的每种数据类型定义一个通道。数据类型可以为任意类型，包括记录类型，但是建议尽可能地使用由 `std_logic` 构建的数据类型，因为目标元件库（如标准元件库）使用的就是这种数据类型，而且电路最终还是要用目标元件库实现的。

```
type handshake_phase is
(
  u,          -- uninitialized
  idle,       -- no communication
  swait,      -- sender waiting
  rwait,      -- receiver waiting
  rcv,        -- receiving data
  rec1,       -- recovery phase 1
  rec2,       -- recovery phase 2
  req,        -- request signal
  ack,        -- acknowledge signal
  error       -- protocol error
);

subtype fp is std_logic_vector(31 downto 0);

type uchannel_fp is
  record
    phase : handshake_phase;
    data  : fp;
  end record;

type uchannel_fp_vector is array(natural range <>) of uchannel_fp;

function resolved(s : uchannel_fp_vector) return uchannel_fp;

subtype channel_fp is resolved uchannel_fp;
```

图 8.12 一个抽象通道的定义

Handshake_phase类型中值的意义详述如下。

u：未初始化的通道。这是驱动器的默认值。只要发送器或接收器用此值驱动通道，这个通道就保持未初始化。

idle：没有通信。发送端和接收端都用idle值驱动通道。

swait：发送端正在等待执行一个通信。发送端用req值驱动通道，而接收端用idle值驱动。

rwait：接收端正等待执行一个通信。发送端用idle值驱动通道，而接收端用rwait值驱动。这个值既可以作为通道的驱动值，也可以作为通道的结果值，就像idle和u值。

rcv：数据传送。发送端用req值驱动通道而接收端用rwait值驱动。经过一段预先确定的时间（位于包顶端的tpd，下一节将介绍）后，接收端把它的驱动值变为ack，通道把它的相变为rec1。在仿真中，只能在rcv相和swait相期间看到传送的值。在其他时间，数据域为预先定义的默认数值。

rec1：恢复相。此时通道瞬时变为rec2相，该相在仿真中不可见。

rec2：恢复相。此时通道瞬时变到idle段，该相在仿真中不可见。

req：当要执行一个通信时，发送端用此值驱动通道。此值不能用于通道。

ack：当要执行一个通信时，接收端用此值驱动通道。此值不能用于通道。

error：协议出错。当判决函数检测到一个错误时，通道显示error值。如果多个驱动器有rwait, req或ack值，会产生一个错误。如果有两个以上的驱动器连接到同一个通道；或者发送命令意外地取代了接收命令（反之亦然），也会产生错误。

图8.13给出了抽象通道协议的图解表示。图中大写的值是通道产生的结果，它们下面的小写的值分别是发送端和接收端的驱动值。发送端和接收端都允许发起一个通信。这样可以在仿真中看到是发送端还是接收端正在等待通信。过程send和receive遵循该协议。

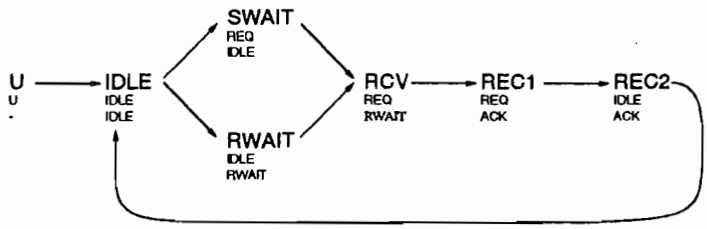


图8.13 抽象通道协议。大写字母的值是通道产生的确定结果值，它们下面的小写字母的值分别是发送端和接收端的驱动值

把具有不同数据类型的通道定义成不同的通道类型，因此对这些通道类型的过程send、receive和probe都必须进行定义。由于VHDL允许程序名的重载，因此能够完成这些定义。通道定义之间的不同之处在于数据类型、通道类型名以及通道中数据域的默认值。所以很容易通过拷贝现有通道的定义来产生一个新的通道类型，而不必重新定义类型handshake_phase。所有这些定义都包含在一个VHDL包中，然后可以在任何需要的地方引用这个包。只有一种通道类型的这种包的例子见附录A.1。程序initialize_in和initialize_out分别用来初始化通道的输入端和输出端。如果发送端或接收端不能初始化通道，那么在这个通道上不能发生任何通信。

一个简单子电路的例子如图8.14所示，它是FIFO级fp_latch。注意到实体中的通道是inout模式，FIFO级在初始化后等待复位信号resetn。按照这种方式，FIFO级等待其他使用这个复位信号进行初始化的子电路。

FIFO级使用了一个通用的参数delay。出于实验目的，加入delay是为了显示通道的不同相。3个FIFO级被连接到图8.15所示的流水线上并已经储存了数据。中间一级的延迟是其他两级延迟时间的两倍。这将会导致在慢的FIFO级之前出现通道阻塞而在慢的FIFO级之后出现通道饥饿。

这个实验结果如图8.16所示。使用的仿真器是Synopsys VSS。图中可见ch_in在swait相非常明显地表现为一个阻塞的通道，而ch_out在rwait相非常明显地表现为一个饥饿通道。

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.abstract_channels.all;

entity fp_latch is
  generic(delay : time);
  port ( d      : inout channel_fp;   -- input data channel
  port ( q      : inout channel_fp;   -- output data channel
         resetn : in std_logic      );
end fp_latch;

architecture behav of fp_latch is
begin

  process
    variable data : fp;
  begin
    initialize_in(d);
    initialize_out(q);
    wait until resetn = '1';
    loop
      receive(d, data);
      wait for delay;
      send(q, data);
    end loop;
  end process;

end behav;
```

图 8.14 一个 FIFO 级的描述

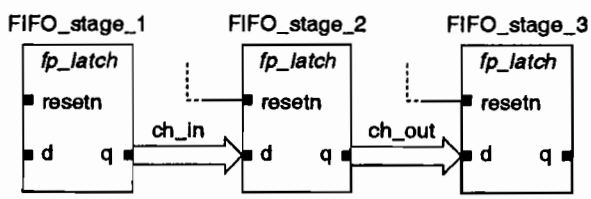


图 8.15 用图 8.14 定义的锁存器构建一个 FIFO

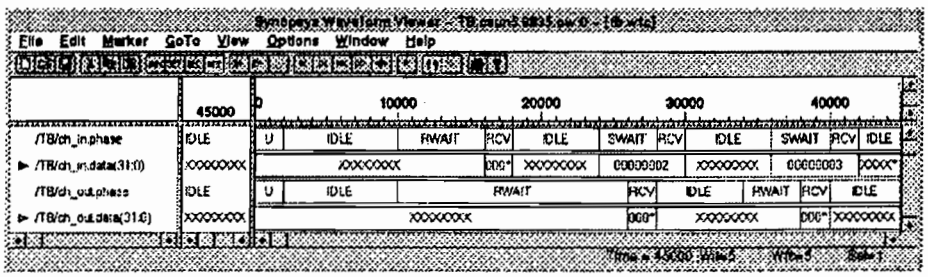


图 8.16 FIFO 用抽象通道包的仿真

8.6.5　实际通道包

在设计过程中，有时需要把通信实体分成控制和数据实体。这是由实际通道类型支持的，在这里，请求和应答信号是单独的std_logic信号——目标元件模型所用的类型。实际通道中的数据类型和抽象通道中的数据类型相同，但是对握手的建模却是不同的。实际通道的定义见图 8.17。

```
subtype fp is std_logic_vector(31 downto 0);

type uchannel_fp is
  record
    req  : std_logic;
    ack  : std_logic;
    data : fp;
  end record;

type uchannel_fp_vector is array(natural range <>) of
       uchannel_fp;

function resolved(s : uchannel_fp_vector) return uchannel_fp;

subtype channel_fp is resolved uchannel_fp;
```

图 8.17　实际通道的定义

所有有关实际通道的定义都包含在一个程序包中（与抽象通道程序包类似），而且通道类型、过程和函数都使用同样的名称。因此由抽象通道改为实际通道进行仿真是很简单的，所要做的只是在设计实体的顶层的use语句中改变程序包的名称。另外一种方法是抽象通道程序包和实际通道程序包使用同样的名称，这种情况下，仿真过程中使用的是最近分析过的那个包。

只有一种通道类型的实际通道程序包的例子见附录A.2。这个程序包定义了一个32位标准逻辑 4 相数据捆绑推通道。包中的常数tpd是指从请求或应答信号的跳变到响应跳变的延迟时间。“Synopsys 编译器指令”嵌入在这个包的多个地方。这是因为 Synopsys 在 floor planner 中生成 EDIF 网表文件时需要清楚通道类型和它们的判决函数，而不是程序包中的过程。

图 8.18 所示是前一节 FIFO 改用实际通道包的仿真结果，注意图中 4 相握手的时序。

可以看到通道上的数值总是从发送端送到通道上的。例外情况是在数据的有效期以外，判决函数产生默认数据，但是这将破坏锁存器的建立和保持时间。过程 send 提供满段（broad）数据有效模式，这意味着无论通道上的有效数据模式是前段（early）、后段（late）还是满段，

它都可以和通道接收端进行通信。过程 receive 要求前段的数据有效模式，这意味着它可以与那些支持前段和满段数据有效模式的发送端进行通信。

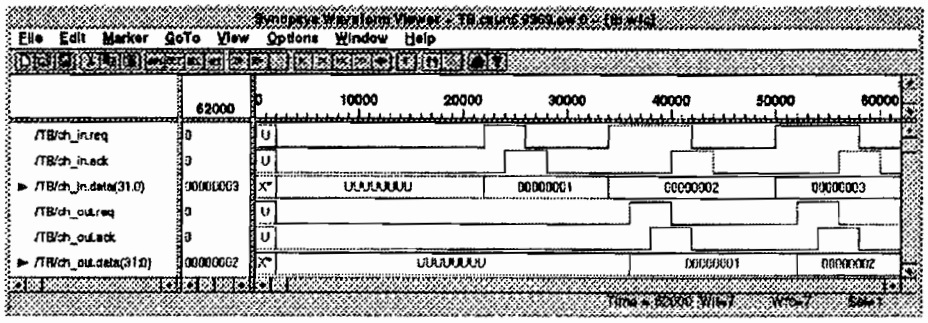


图 8.18 FIFO 用实际通道包的仿真

实际通道（与抽象通道）的判决函数可以检测协议错误。如果在一个通道上有不止一个发送端或接收端，或在通道的终端错误地使用了发送命令或者接收命令，这都会产生错误。在这些情况下，通道的请求或应答信号上呈现出 x 值。

8.6.6 控制与数据的划分

本节介绍如何将一个实体划分成控制和数据实体。当采用的是实际通道程序包时，这一划分是可行的，但正如下面解释的那样，这种划分还须遵循一定的规则。

为了更好地说明划分过程是如何实现的，将前节图 8.14 所示的 FIFO 级分解为锁存器控制电路 `latch_ctrl` 和锁存器 `std_logic_latch`。相应的 VHDL 代码如图 8.19 所示，图 8.20 所示的是这一分解的图形说明，其中包含下面将要说明的未定信号 `ud` 和 `uq`。

在 VHDL 中，要驱动一个复合判决信号，驱动器要驱动该信号所有的域。因此，控制电路不能只驱动通道中的应答域。为克服这个问题，与未定通道类型相对应的信号必须在分解的实体中进行声明，这就是图 8.17 中 `uchannel_fp` 类型的信号 `ud` 和 `uq` 的功能。因为信号未判决，控制电路只驱动这个信号的应答域，剩下的域保持未初始化。这个未判决信号驱动通道，由于它驱动通道中的所有域，因此这是允许的。通道的判决函数将忽略通道被驱动的未初始化的值。使用 `send` 和 `receive` 过程的元件同样也驱动通道中的那些域，并且不受未初始化的值控制。例如，通道的输出可以驱动带有 `U` 值的通道的应答域。通道中作为输入端使用的域被通道连接到读取这些域的电路中。

信号 `ud` 和 `uq` 不直接驱动 `d` 和 `q`，但是通过 `connect` 函数驱动。这个函数只返回它的参数。这看起来没有必要，但当一些子电路被描述成用标准单元实现时，这还是必要的。在仿真

中，专门的“门级仿真引擎”被用来仿真标准单元[129]。在初始化过程中，将把一些信号的值置为X而不是U。我们不能用通道的判决函数来忽略这些X，因为门级仿真引擎设定了通道中的一些值。通过引入行为描述函数 connect，通常的仿真器用判决函数来对通道进行处理和评估。必须强调的一点是，“门级仿真引擎”的缺陷是必须要增加 connect 函数。

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.real_channels.all;

entity fp_latch is
  port ( d      : inout channel_fp;     -- input data channel
         q      : inout channel_fp;     -- output data channel
         resetn : in std_logic      );
end fp_latch;

architecture struct of fp_latch is

  component latch_ctrl
    port ( rin, aout, resetn : in  std_logic;
           ain, rout, lt : out std_logic      );
  end component;

  component std_logic_latch
    generic (width : positive);
    port ( lt : in  std_logic;
           d  : in  std_logic_vector(width-1 downto 0);
           q  : out std_logic_vector(width-1 downto 0)  );
  end component;

  signal lt : std_logic;
  signal ud, uq : uchannel_fp;

begin

  latch_ctrl1 : latch_ctrl
    port map (d.req,q.ack,resetn,ud.ack,uq.req,lt);
  std_logic_latch1 : std_logic_latch
    generic map (width => 32)
    port map (lt,d.data,uq.data);

  d <= connect(ud);
  q <= connect(uq);

end struct;
```

图 8.19　FIFO 级分成一个普通数据锁存器和一个锁存控制电路

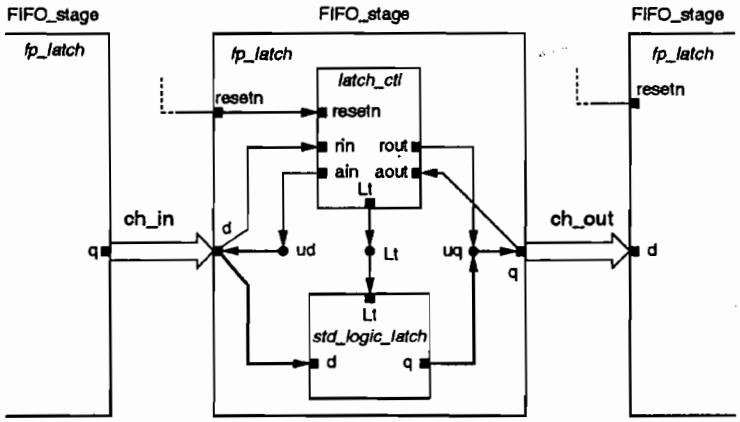


图 8.20 分成控制和数据

8.7 小结

这一章介绍了异步电路高层次建模和综合的语言及CAD工具，重点是几种有代表性的基于类CSP语言的设计方法。之所以选择这些语言，是因为它们支持进程之间基于通道的通信（同步信息传送）和在进程及语句层的并发，这两个特性对异步电路建模很重要。文中通过图例介绍了面向语法编译的综合方法。本书后续章节中将详细讨论这些问题。

本章最后通过图例介绍了基于通道的通信如何用VHDL实现，给出了两个程序包，包中包含了所有必要的过程和函数：send、receive和probe。这些包支持人工自顶向下逐步求精的设计流程，在这个从高层规范到低层电路实现的整个设计过程中，可以用相同的测试平台来仿真设计。

这一章介绍了异步设计的语言和CAD工具，可以作为异步电路设计教程的结束。把整个教程总结如下：第2章介绍了基本概念和基本原理，给出了参考文献。第3章和第4章给出了异步电路的类似于RTL的抽象观点（托肯在静态数据流结构中流动），这对于理解异步电路的运行与实现非常有用。这部分内容可能是本教程对现有文献增补最多的地方。第5章和第6章介绍了数据路径操作符和控制电路的设计。第6章重点放在速度无关电路上，但这不是唯一的方法。近年来，在综合多输入变化的基本模式的电路方面也取得了很大进步。第7章讨论了更高级的4相数据捆绑协议与电路。最后的第8章介绍了用来对异步电路进行高层次建模与综合的语言和工具。

教程有意不涉及该领域的各个方面，目的是为进入“异步设计世界”铺路。现在你已站在路的起点，希望你具有坚实的基础，深入发掘文献资料，同样重要的是带着对异步电路性能的充分理解，开始设计你自己的电路。最后要说的是，异步电路不是要代替同步电路。它们在某些方面具有优点，而在另外一些方面又有许多不足，它们应该被视为一个补充，这样它们就拓宽了数字设计者探索的解空间。即使至今，许多电路仍不能被单纯地划成同步电路或异步电路，它们常常包含同步和异步这两种元素。

接下来的章节将介绍一些最近的工业级异步芯片。文献[106]还给出了一些另外的设计。

附录A VHDL通道包

A.1 抽象通道包

```
    -- Abstract channel package: (4-phase bundled-data push channel, 32-bit
data)

    library IEEE;
    use IEEE.std_logic_1164.all;

    package abstract_channels is

       constant tpd : time := 2 ns;

    -- Type definition for abstract handshake protocol
      type handshake_phase is
      (
        u,          -- uninitialized
        idle,       -- no communication
        swait,      -- sender waiting
        rwait,      -- receiver waiting
        rcv,        -- receiving data
        rec1,       -- recovery phase 1
        rec2,       -- recovery phase 2
        req,        -- request signal
        ack,        -- acknowledge signal
        error       -- protocol error
      );

    -- Floating point channel definitions

      subtype fp is std_logic_vector(31 downto 0);

      type uchannel_fp is
```

```
    record
      phase : handshake_phase;
      data  : fp;
    end record;

  type uchannel_fp_vector is array(natural range <>) of
        uchannel_fp;

  function resolved(s : uchannel_fp_vector) return uchannel_fp;

  subtype channel_fp is resolved uchannel_fp;

  procedure initialize_in(signal ch : out channel_fp);

  procedure initialize_out(signal ch : out channel_fp);

  procedure send(signal ch : inout channel_fp; d : in fp);

  procedure receive(signal ch : inout channel_fp; d : out fp);

  function probe(signal ch : in channel fp) return boolean;

end abstract_channels;
package body abstract_channels is

-- Resolution table for abstract handshake protocol

  type table_type is array(handshake_phase, handshake_phase) of
        handshake_phase;
  constant resolution_table : table_type := (
-------------------------------------------------------------------------
-- 2. parameter:                                                |      |
-- u     idle  swait rwait rcv   recl  rec2  req   ack   error    |1.par:|
-------------------------------------------------------------------------
  (u,    u,    u,    u,    u,    u,    u,    u,    u,    u    ),--|u     |
  (u,    idle, swait,rwait,rcv,  recl, rec2, swait,rec2, error),--|idle  |
  (u,    swait,error,rcv,  error,error,recl, error,recl, error),--|swait |
  (u,    rwait,rcv,  error,error,error,error,rcv,  error,error),--|rwait |
  (u,    rcv,  error,error,error,error,error,error,error,error),--|rcv   |
```

```
    (u,    recl, error,error,error,error,error,error,error,error), --|recl |
    (u,    rec2, recl, error,error,error,error,recl, error,error), --|rec2 |
    (u,    error,error,error,error,error,error,error,error,error), --|req  |
    (u,    error,error,error,error,error,error,error,error,error), --|ack  |
    (u,    error,error,error,error,error,error,error,error,error));--|error|

-- Fp channel

    constant default_data_fp : fp := "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";

    function resolved(s : uchannel_fp_vector) return uchannel_fp is
      variable result : uchannel_fp := (idle, default_data_fp);
    begin
      for i in s'range loop
        result.phase := resolution_table(result.phase, s(i).phase);
        if (s(i).phase = req) or (s(i).phase = swait) or
           (s(i).phase = rcv) then
             result.data :- s(i).data;
        end if;
      end loop;
      if not((result.phase = swait) or (result.phase = rcv)) then
        result.data := default_data_fp;
      end if;
      return result;
    end resolved;

    procedure initialize_in(signal ch : out channel_fp) is
    begin
      ch.phase <= idle after tpd;
    end initialize_in;

    procedure initialize_out(signal ch : out channel fp) is
    begin
      ch.phase <= idle after tpd;
    end initialize_out;

    procedure send(signal ch : inout channel_fp; d : in fp) is
    begin
      if not((ch.phase = idle) or (ch.phase = rwait)) then
        wait until (cb.phase = idle) or (ch.phase = rwait);
```

```
    end if;
    ch <= (req, d);
    wait until ch.phase = recl;
    ch.phase <= idle;
  end send;

  procedure receive(signal ch : inout channel_fp; d : out fp) is
  begin
    if not((ch.phase = idle) or (ch.phase = swait)) then
      wait until (ch.phase = idle) or (ch.phase = swait);
    end if;
    ch.phase <= rwait;
    wait until ch.phase =rcv;
    wait for tpd;
    d := ch. data;
    ch.phase <= ack;
    wait until ch.phase = rec2;
    ch. phase <= idle;
  end receive;

  function probe(signal ch : in channel_fp) return boolean is
  begin
    return (ch.phase= swait);
  end probe;

end abstract_channels;
```

A.2 实际通道包

```
-- Low- level channel package (4-phase bundled-data push channel, 32-bit data)

library IEEE;
use IEEE. std_logic_1164, all;

package real_channels is

  -- synopsys synthesis_off
  constant tpd : time := 2 ns;
  -- synopsys synthesis_on
```

```
-- Floating point channel definitions

  subtype fp is std_logic_vector(31 downto 0);

  type uchannel_fp is
    record
      req  : std_logic;
      ack  : std_logic;
      data : fp;
    end record;

  type uchannel_fp_vector is array(natural range <>) of
        uchannel_fp;

  function resolved(s : uchannel_fp_vector) return uchannel_fp;
  subtype channel_fp is resolved uchannel_fp;

  -- synopsys synthesis_off
  procedure initialize_in(signal ch : out channel_fp);

  procedure initialize_out(signal ch : out channel_fp);

  procedure send(signal ch : inout channel_fp; d : in fp);

  procedure receive(signal ch : inout channel_fp; d : out fp);

  function probe(signal ch : in uchannel_fp) return boolean;
  -- synopsys synthesis_on

  function connect(signal ch : in uchannel_fp) return channel_fp;

end real_channels;

package body real_channels is

-- Resolution table for 4-phase handshake protocol
  -- synopsys synthesis_off
  type stdlogic_table is array(std_logic, std_logic) of std_logic;
```

```
  constant resolution_table : stdlogic_table := (
  --  ---------------------------------------------------------------------
  --  | 2. parameter:                                                |       |
  --  |  U    X    0    1    Z    W    n    H    -                  |1. par:|
  --  ---------------------------------------------------------------------
      ( 'U', 'X', '0', '1', 'X', 'X', 'X', 'X', 'X', ),  -- |   U   |
      ( 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', ),  -- |   X   |
      ( '0', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', ),  -- |   0   |
      ( '1', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', ),  -- |   1   |
      ( 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', ),  -- |   Z   |
      ( 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', ),  -- |   W   |
      ( 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', ),  -- |   L   |
      ( 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', ),  -- |   H   |
      ( 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', )); -- |   -   |
  -- synopsys synthesis_on

-- Fp channel

  -- synopsys synthesis_off
  constant default_data_fp : fp := "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
  -- synopsys synthesis_on

  function resolved(s : uchannel_fp_vector) return uchannel_fp is
  -- pragma resolution_method three_state
  -- synopsys synthesis_off
    variable result : uchannel_fp := ('U','U',default_data_fp);
  -- synopsys synthesis_on
  begin
  -- synopsys synthesis_off
    for i in s'range loop
      result.req := resolution_table(result.req, s(i).req);
      result.ack := resolution_table(result.ack, s(i).ack);
      if (s(i).req = '1') or (s(i).req = '0') then
        result.data := s(i).data;
      end if;
    end loop;
    if not((result.req = '1') or (result.req = '0')) then
      result.data := default_data_fp;
    end if;
    return result;
```

```
-- synopsys synthesis_on
end resolved;

-- synopsys synthesis_off
procedure initialize_in(signal ch : out channel_fp) is
begin
  cb.ack <= '0' after tpd;
end initialize_in;

procedure initialize_out (signal ch : out channel_fp) is
begin
  ch. req <= '0' after tpd;
end initialize_out;

procedure send(signal ch : inout channel_fp; d : in fp) is
begin
  if ch.ack /= '0' then
     wait until ch.ack = '0';
  end if;
  ch.req <= '1' after tpd;
  ch.data <= d after tpd;
  wait until ch.ack = '1';
  ch. req <= '0' after tpd;
end send;

procedure receive(signal ch : inout channel fp; d : out fp) is
begin
  if ch. req /= '1' then
     wait until ch.req = '1';
  end if;
  wait for tpd;
  d := ch.data;
  ch.ack <= '1';
  wait until ch.req = '0';
  ch.ack <= '0' after tpd;
end receive;

function probe(signal ch : in uchannel_fp) return boolean is
begin
```

```
    return (ch.req = '1');
  end probe;
  -- synopsys synthesis_on

  function connect(signal ch : in uchannel_fp) return channel_fp is
  begin
    return ch;
  end connect;

end real_channels;
```

第二部分　Balsa——异步硬件综合系统

Doug Edwards, Andrew Bardsley
Department of Computer Science
The University of Manchester
{doug, bardsley}@cs.man.ac.uk

摘要： Balsa是一种用来描述和综合异步电路的系统（基于面向语法编译成通信握手电路）。在以下几章中，将介绍Balsa的基本设计流程，并且用一些简单的电路例子以非规范的形式来阐述Balsa语言，并以一个完整的设计作为这一部分的总结：一个完全用Balsa描述的4通道DMA控制器。

关键词： 异步电路，高级综合

第9章　Balsa入门

9.1　概述

Balsa既是综合异步硬件系统的架构，也是描述此类系统的语言。采用的是面向语法的编译方法，将语言描述编译成通信握手元件，与Philips的Tangram系统（文献[141, 135]和第13章）非常接近。这种方法的优势在于编译透明：语言结构与由它生成的中间握手电路之间是一一对应的。这对一个有经验的使用者来说，可以从原始的描述语言中容易看出电路的微观体系结构。语句的增量变化只会引起电路的可预测性的变化。使优化和设计调整变得简单是非常重要的，因为同步VHDL综合过程中，描述语句的微小变化都可能会导致最终电路发生根本性的变化。

理解Balsa为设计者提供了什么以及设计者还应该做些什么是很重要的。通过快速编译缩短了“编辑描述－综合－仿真－修改描述”的循环过程，这使得探究开发系统的设计空间变得容易，并且可以快速地评估系统的原型。然而，创造力是不可替代的，差的设计与好的设计一样容易设计出来。在异步电路的设计中需要一些经验，甚至优秀的传统同步电路的设计者在完全胜任系统开发之前也是如此。但要注意的是，虽然Balsa能保证电路的结构正确，但并不能保证系统正确。例如，在任意一个异步系统中，可以描述一个本身存在死锁的电路。此外，当电路通过传统的CAD工具布局布线后，需要布局后（post-layout）仿真以检验它是否满足基本的时序要求。另一方面，通过选择不同的实现库（implementation libraries），设计者可以在简化延迟不敏感电路的实现过程和较小面积（可能要增加布局后验证的难度）之间做出折衷的选择。

虽然Balsa是在研究环境中开发出来的，但它并不是那种不适合大规模设计仅玩玩而已的系统；Balsa已被用来综合用于Amulet3i的异步微处理器宏单元[48]的32通道DMA控制器[11]。这种控制器有复杂的指标，是用0.35 μm的3层金属工艺来实现的，所占面积为2 mm^2。在本书撰写期间，Balsa正被用来综合一个完整的Amulet核，该核是EU（欧盟）基金支持的G3智能卡项目的一部分[46]。

上文曾提到，Balsa与Tangram非常相似。但是Balsa是一个尚未完善的工具包，它缺少一些Tangram包中的有用工具，如功耗性能分析器。但是，Balsa可以免费获得，而Tangram

一般只能在 Philips 内部使用。就语言表述能力而言，Balsa 用递归扩展定义工具（recursive expansion definition facilities）增加了功能强大的参数表示，而 Tangram 在与非延迟不敏感外部接口的交互中具有更大的灵活性。Balsa 不增加这些特性是为了确保单通道延迟不敏感模型不受影响。

读者应当注意，以下内容并未涉及 Balsa 语言或语法的全部，更详细的介绍参见 Balsa 用户指南[7]，Balsa 系统设计软件也可从该网站免费得到。这个系统软件还在不断升级中：本书介绍参考的是 Balsa 3.1.0 版本。

9.2　基本概念

由 Balsa 描述的电路被编译成一个通信网络，该网络是由一组（大约 40 个）握手元件的集合组成的。这些元件由通道连接在一起，原子（atomic）通信或握手就在通道中发生。通道可能有与之相关的数据路径（在此情况下，握手包含了数据传递），或者只是纯粹的控制（此时，握手作为同步（synchronisation）或约会点（rendezvous point）使用）。

每个通道正确地把一个握手单元的被动端连接到另一个握手单元的主动端。主动端是通信的起始端口，被动端（当准备好时）通过应答信号对主动端发出的请求作出响应。

数据通道可能是推通道也可能是拉通道。在推通道中，数据流的方向是从主动端口到被动端口。这与微流水线（micropipelines）的通信方式相似，当请求信号到来时数据有效，当应答信号到来时数据释放。在拉通道中，数据流的方向从被动端口流入到主动端口。当主动端口请求发送时，数据的有效性通过被动端口的应答来确认。图 9.1 给出了一个由握手单元组成的电路实例。主动端口在握手单元旁边用一个实心的小圆圈表示，而被动端口用空心小圆圈表示。

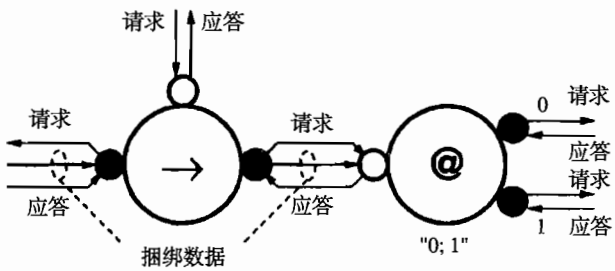


图 9.1　两个相连的握手单元

这里，提取元件（Fetch）或传送元件（用“->”表示）和装入元件（Case）（用 @ 表示）由内部携带数据的通道连接。电路被发送到传送元件的请求信号激活，传送元件再在它的拉输

入端口主动端（在图的左边）发出一个请求信号给环境，环境提供需求的数据并通过应答信号确认数据的有效性。然后传送元件在它的推输出端口的主动端将握手请求和数据传输到装入元件，并由装入元件的被动端接收。根据数据值，装入元件在其右上或右下端口发布一个握手信号到它的环境中。最后，当装入元件接收到环境的应答信号后，另一个应答信号通过原信道返回，结束这次握手通信，电路准备下次操作的到来。

在该例中，数据流的方向和请求信号的方向相同，而请求信号与应答信号方向相反。在上图中，清楚地展示了各个请求、应答信号和数据的物理线路，数据传送线与信号传送线分开（和控制信号“捆绑”在一起）。对其他数据/信号编码方式而言，这未必一定正确。

图9.1所示的捆绑数据方式并非为唯一的实现方式。有很多方法可以用来实现延时不敏感信号连接的通道。在这样的通道中，单个线路间的时序关系不会影响电路的功能。握手电路可以用这些对 naïve（质朴的）实现、工艺偏差和互联延迟特性具有鲁棒性的方法来实现。将来 Balsa 的版本会提供一些可供选择的后端。2.1 节和 7.1 节对握手协议有更为详细的讨论。

一般来说，握手电路图不会像图 9.1 中表示的那么详细，通道通常用一条有数据流向的弧线来表示，用弧线上的箭头来表示数据流向。类似地，只含控制信号的通道，仅由请求/应答信号线构成，用一条没有箭头的弧线表示。握手电路一般来说不算复杂：例如，传送元件可能只用导线来实现。一个十进制计数器（见 10.6 节）握手电路的例子如图 9.2 所示。相应的门级实现见图 9.3。

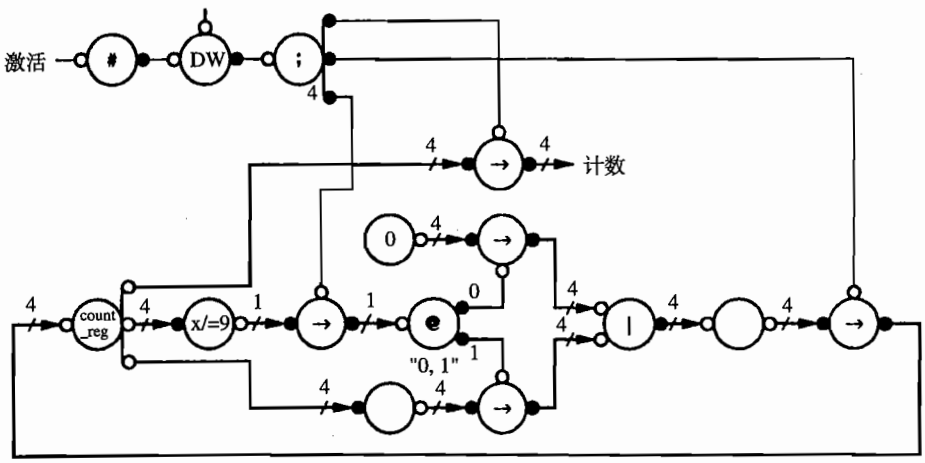


图 9.2　十进制计数器的握手电路

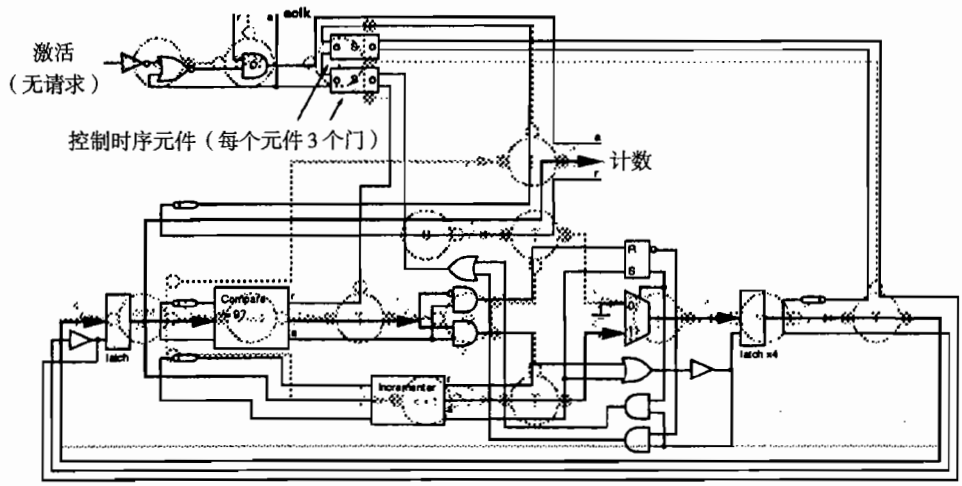


图 9.3　十进制计数器的门级电路

9.3　工具集和设计流程

Balsa 的设计流程如图 9.4 所示。用 LARD 进行行为仿真[38]，LARD 是 Amulet 工作组为模拟异步系统而开发的一种语言。也可以使用目标 CAD 系统进行更为精确的仿真和验证设计。大多数 Balsa 工具都支持 Breeze 中间文件，Breeze 文件是编译 Balsa 描述所生成的。利用 Breeze 文件，后端工具可以为 Balsa 描述提供具体实现。Breeze 文件还包含 Balsa 源文件提供的过程和类型定义，这样可以使 Breeze 用作 Balsa 的包描述格式。

Balsa 系统包括以下工具集。

- balsa-c：Balsa 语言的编译器。该编译器将 Balsa 代码生成 Breeze。
- balsa-netlist：将 Breeze 代码生成网表（当前有 EDIF，Compass 和 Verilog 三种格式的网表），执行工艺映射和握手扩展。
- breeze2ps：生成握手电路图的 PostScript 文件的工具。
- breeze2lard：从 Breeze 文件到 LARD 行为模型的转换器。
- breeze-cost：用于估计电路面积的工具。
- balsa-md：用于为 make(1)生成 Makefiles 的工具。
- balsa-mgr：具有项目管理的 balsa-md 图形化前端工具。

Balsa 和目标 CAD 系统间的接口用以下指令来执行。

- balsa-pv：使用 powerview 工具把顶层 powerview 原理图（含有 Balsa 产生的电路）转换为 EDIF 文件。

- balsa-xi：由 EDIF 描述生成 Xilinx 的下载文件。
- balsa-ihdl：到 Cadence Verilog-XL 环境的接口。

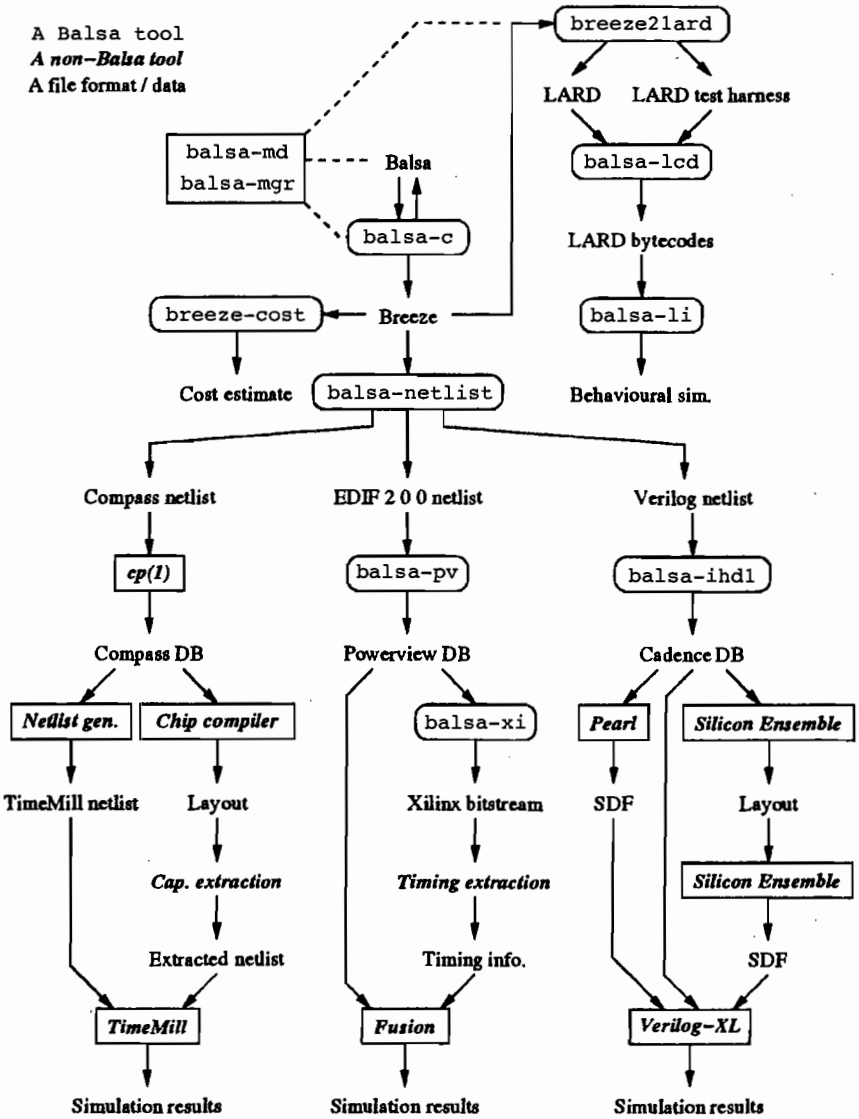


图 9.4 设计流程

9.4 开始设计

在这一节中，通过用 Balsa 描述的简单缓冲电路来介绍 Balsa 描述的基本元素。

9.4.1　单级缓冲器

这个缓冲电路相当于 HDL 中的“hello，world”简单小程序。它的 Balsa 描述是：

```
import [balsa.types.basic]
-- a single line comment
-- bufferla: A single place buffer
procedure buffer1 (input i: byte; output o: byte) is
  variable x: byte
begin
  loop
    i -> x -- Input communication
    ;      -- sequence the two communications
    o <- x -- Output communication
  end
end
```

代码解释

这个 Balsa 描述构建了 8 位宽度的单级缓冲器。电路请求环境发送一个字节，环境准备好后将数据传送到寄存器。电路通过输出通道给外部环境发信号来指示数据有效，当外部环境选择该缓冲器时从中读取数据。这一小程序介绍了如下内容。

注释： Balsa 支持多行注释和单行注释。

模块化编译： Balsa 支持模块化编译。本例中 `import` 语句定义了一些标准数据类型，如字节、半字节（nibble）等。在 `import` 语句中给出的寻找路径类似于 Java 的分离点（dot-separated）的目录路径（尽管不能实现多文件包）。`import` 语句可用来包含其他预编译的 Balsa 程序，因此它的机制类似于一个库。任何 `import` 语句必须位于文件中的其他声明之前进行声明。

过程： 过程声明中引入了一个对象，该对象与传统编程语言的过程定义相类似。一个 balsa 过程就是一个进程。过程的参数定义了与电路模块外部环境的接口。在本例子中，模块有 8 位输入和 8 位输出。过程定义的主体部分定义了电路的算法行为；它也隐含了电路的结构实现。例中的变量 x（字节类型）的声明意味着在综合后的电路内会有一个 8 位宽度的存储单元。

从代码中很容易看出电路的行为：8 位数值从环境传输到存储变量 x，然后从变量输出到环境。这些事件按顺序不断循环（`loop...end`）。

通道通信： 通信操作符“->”和“<-”表示通道分配以及通道上的通信或握手。因为先后顺序已经描述得很清楚，变量 x 只有当它准备好的时候才接收一个新值，也只有当请求到来时才输出到环境中。注意到通道总是在操作符的左边，相应的变量或表达式在操作符的右边。

顺序：用于分离两个任务的“;”操作符，不仅仅是语法中的语句分隔符，它还明确地指示了顺序。输入完成后，x的内容传送到输出端口。由于操作符“;”连接了两个顺序的语句或模块，所以在模块中的最后语句放置“;”是错误的。

循环：loop...end结构表示包含在程序体中的代码的无限循环。程序可以没有loop...end，这将会终止程序。当需要时，可以顺序调用程序。

编译电路

```
balsa-c bufferla
```

编译器生成输出文件bufferla.breeze。这是一个中间格式文件，可以重新导入到其他Balsa源文件中（提供了一种简单的库机制）。Breeze设计成便于语法分析的文本格式，所以它不容易被理解。要得到编译的电路（以握手元件形式出现）的简单图形表示可通过以下命令实现（为bufferla.ps）：

```
breeze2ps bufferla
```

综合后电路

最后生成的握手电路如图9.5所示。但实际上，这幅图并不是直接取自“breeze2ps”命令的输出，为了便于阅读，我们进行了一些修改。虽然没有必要精确地理解编译后电路的具体操作，但是了解一些结构方面的知识有助于更好地理解如何使用Balsa描述出最有效的综合电路。电路中标注了各种握手单元的名称，下面简要介绍一下电路的操作。

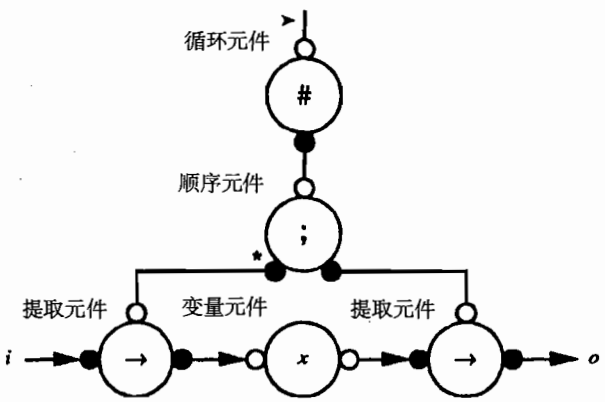


图9.5 一个单级缓冲器的握手电路

顶部的端口用“>”表示，它是一个“激活端口”，用来产生进入电路行为的握手。可以把它看成一个复位信号，当它脱离有效状态时，电路进行初始化。所有的Balsa电路都含有激活端口。

激活端口启动循环元件（“#”）运行，并使它与顺序元件发生握手。循环元件对应于loop...end结构，顺序元件对应于“;”操作符。顺序元件首先向左边的提取元件发送一个握手信号，

使数据传送到变量元件的存储单元。然后顺序元件与右边的提取元件握手，并从变量元件中读取数据。完成这些操作后，顺序元件完成它和循环元件的握手，循环元件将重复这一过程。

9.4.2　二级缓冲器

设计 1

构建单级缓冲器后，我们下一步的目标是构建一个由单级缓冲器组成的流水线。我们先考虑两级缓冲器，有很多方法来描述它，其中之一是定义一个有两个存储单元的电路。

```
-- buffer2a: Sequential 2-place buffer with assignment
--           between variables
import [balsa.types.basic]

procedure buffer2 (input i: byte; output o: byte) is
  variable x1, x2: byte
  begin
  loop
    i -> x1;   -- Input communication
    x2 := x1; -- Implied communication
    o <- x2    -- Output communication
  end
end
```

在这个例子中，我们引入了两个存储单元 x1 和 x2。变量 x1 的内容通过任务操作符 ":=" 传送给变量 x2。当然，传送还是通过握手通道实现的。任务操作符只是为了方便而隐藏通道的一种做法。

设计 2

这个隐藏的通道可以在 buffer2b.balsa 中清晰地看到：

```
-- buffer2b: Sequential version with an explicit
--           internal channel
import [balsa.types.basic]

procedure buffer2 (input i:byte; output o:byte) is
  variable x1, x2: byte
  channel chan: byte
begin
  loop
    i -> x1;                    -- Input communication
    chan <- x1 || chan -> x2; -- Transfer x1 to x2
```

```
    o <- x2                    -- Output communication
  end
end
```

前面一个例子中隐藏在“:=”任务操作符后的通道在这里已经显示出来了。生成的握手电路（经过一些简单优化后）与buffer2a是相同的。“||”操作符的含义将在下一个例子中解释。

理解由buffer2a与buffer2b代码生成电路过程的意义是很重要的。记住“;”不只是一个语句分隔符：它是一个表示顺序的操作符。因此，输入i首先传送到x1，完成这个操作后，x1传送到x2，最后x2的内容通过端口“o”写入环境。只有在这一系列的操作完成后，新数据才可以再次从环境中读入x1。

9.4.3 并行成分和模块复用

我们没有必要完全按照上述约束进行操作：即当x2的数据输出至外部环境的同时，电路不能读入一个新值给x1，这并不是必须的。buffer2c中的程序对此进行了优化。

```
-- buffer2c: a 2-place buffer using parallel composition
import [buffer1a]

procedure buffer2 (input i: byte; output o: byte) is
  channel c: byte
begin
  buffer1 (i, c) ||
  buffer1 (c, o)
end
```

代码解释

在以上程序中，二级缓冲器由两个单级缓冲器组成。第一个缓冲器的输出与第二个缓冲器的输入相连。但是，除了在共有通道上的通信外，这两个缓冲器的工作是相互独立的。

以上看似简单的程序说明了Balsa语言的一些新特点，列举如下。

模块化编译：先前介绍的输入机制中包含了buffer1a电路。buffer1a电路必须先进行编译。Makefile的生成命令“balsa-md”（见9.5.1节）可以自动生成一个含有编译依赖关系的Makefile。

通过命名来连接：由于在缓冲器例化的参数列表中有共用的通道名c，因此，第一个缓冲器的输出端连接到第二个缓冲器的输入端。

并行成分：操作符“||”表示它连接的两个单元是并行操作的。但这并不意味着这两个单元是完全独立的：在这个例子中，一个缓冲器的输出端往另一个缓冲器的输入端写入数据，产生了一个同步的点。需要注意的是，所指的并行只是时间上的并行。这两个缓冲器物理上是串联的。

9.4.4　设置多重结构

如果我们想扩展缓冲器级数，像前面那样具体地列出每个缓冲器的方法显得很烦琐。需要一种设置缓冲器长度的参数化方法（尽管真实的硬件实现中，缓冲器的数量必须事先知道而且不能变化）。在 buffer_n 中的方法是用 for 结构和编译常量一起实现的。

```
-- buffer_n: an n-place parameterised buffer
import [buffer1a]
constant n = 8

procedure buffer_n (input i:byte; output o:byte)
is
  array 1 .. n-1 of channel c: byte
begin
  buffer1 (i, c[1]) ||        -- First buffer
  buffer1 (c[n-1], o) ||      -- Last buffer
  for || i in 1 .. n-2 then -- Buffer i
    buffer1 (c[i], c[i+1])
  end
end
```

代码解释

常量：（任何类型的）一个表达式的值都可以使用一个名称来代替，表达式的值在编译时产生，当用到这个名称时，它的类型与在常量声明下的原始表达式的类型相同。数字可以用十进制（无前缀，1 至 9 中任一开头），16 进制（前缀是 0x），八进制（前缀是 0）和二进制（前缀是 0b）来表示。

数组通道：过程的端口和局部通道可能是一个数组。每个通道都能通过数字或枚举来索引表示，但是从握手的角度看，每个通道又都是不同的，有索引的通道除了共享通道名称之外与其他各通道没有任何关系。数组不是一种通道类型。

for 循环：for 循环允许反复例化一个子电路。电路的结构可能是并行的（如前面的例子）或者是顺序的。在后一种情况下，操作符“；”用来代替循环中的“||”。循环的重复范围在编译时必须是明确的。

在第 11 章中，我们将讨论另一种更加灵活的参数化程序的方法。

9.5　Balsa 辅助工具

9.5.1　Makefile 生成

在 UNIX 操作系统中，“make(1)”应用程序通常使用 Makefile 来说明和控制复杂程序的编译过程，确定程序之间的依赖关系一般很复杂而且容易出错。Balsa 系统内有一个称为

balsa-md的工具，它可以自动生成给定程序的Makefile。生成的Makefile不仅知道怎样编译多个端口的Balsa模块，而且知道怎样为balsa的仿真环境LARD生成和运行测试工具。Balsa-mgr给balsa-md提供了方便而直观的GUI界面，能够尽可能地简化工程管理，特别适合处理多种测试工具的共同使用。然而对GUI的介绍很冗长，所以我们对balsa-mgr不做深入讨论，只在下面的例子中进行介绍，后台的balsa-md为其提供一个通路(gateway)，balsa-mgr的界面如图9.6所示。

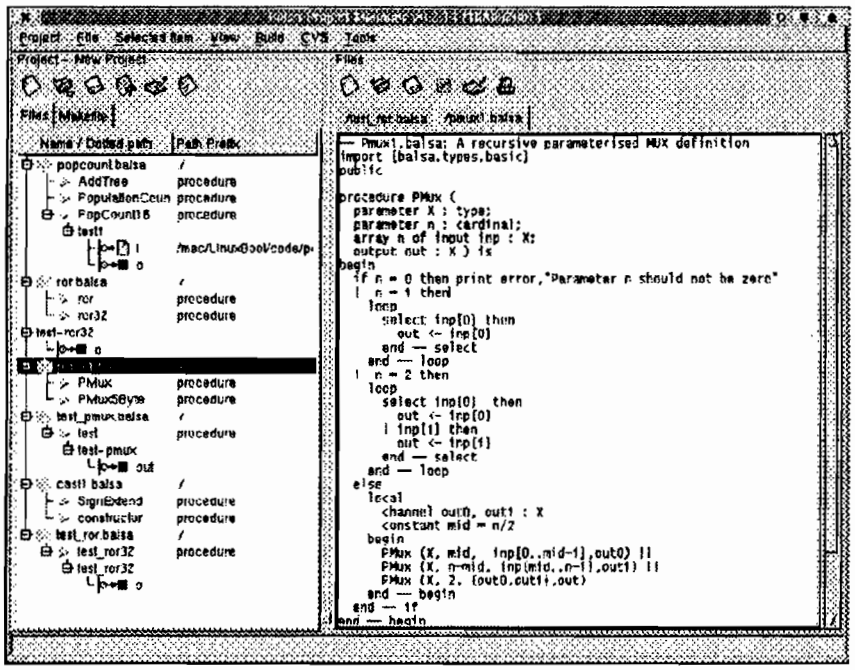


图9.6 balsa-mgr IDE

9.5.2 面积消耗估计

电路的面积可以通过执行Makefile的cost命令来进行估计。例如，二级缓冲器生成的输出报告摘录如下：

```
Part: buffer2
(0 (component "$BrzFetch" (8) (10 2 9)))
(0 (component "$BrzFetch" (8) (8 6 7)))
(0 (component "$BrzFetch" (8) (5 4 3)))
(20.75 (component "$BrzLoop" () (1 11)))
(99.0 (component "$BrzSequence" (3) (11 (10 8 5))))
```

```
(198.0 (component "$BrzVariable" (8 1 "x1[0..7]") (9 (6))))
(198.0 (component "$BrzVariable" (8 1 "x2[0..7]") (7 (4))))

Total cost: 515.75
```

报告的格式及数据的准确意义可能多少有些让人费解。实际上，每行对应着一个握手元件，行中的第1个数字是它占用的面积。在元件名后面的参数对应该元件所含各通道的宽度和内部通道名。报告中的面积与用特定的制造工艺制造的电路的面积是成比例的，并且经常用它来比较不同的电路描述。

9.5.3　查看握手电路图

运行命令 make ps 可以生成握手电路图的 PostScript 视图。图 9.7 是 buffer.2c 例中握手电路图的平面视图。

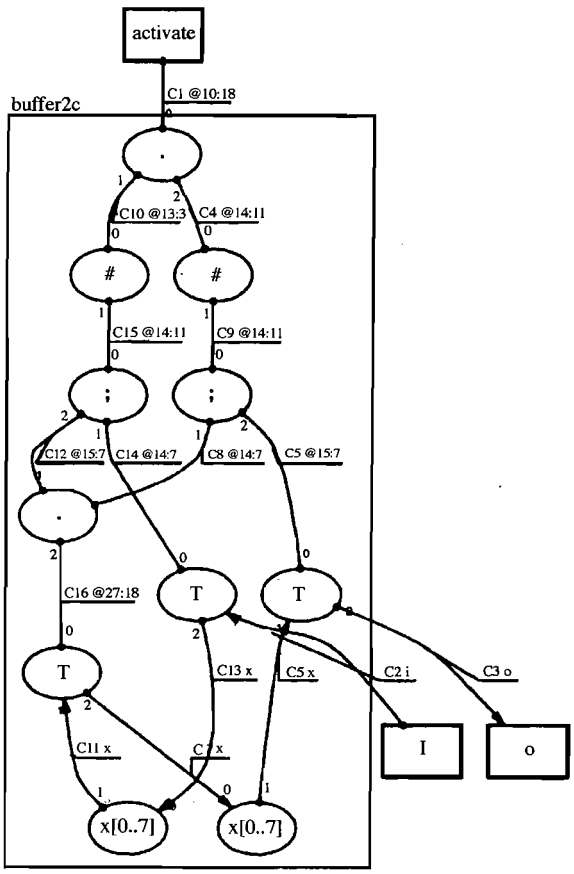


图 9.7　buffer.2c 的平面图

从该图中可以看出电路由两个单级缓冲器组成。除了在握手元件符号的标注中稍有差别外，电路与8.4节中的图8.6所示的电路是相同的，并且采用的优化也相同。

9.5.4 仿真

一旦设计转换为物理版图，各种能用的仿真就可以从略。有三种方法可以用于评估和仿真Balsa中的设计。

1. 默认的LARD测试工具
   `make sim`命令可以生成一个LARD测试工具并且运行它。测试工具使待测模块的各个输入端从文件中读取数据，输出数据被发送到标准输出端口的输出通道上。这一方法根本不需要有关LARD的知识。
2. Balsa测试工具
   如果需要更复杂的测试序列，Balsa是一种灵活的语言，其本身可以描述大多数的测试序列。因此能够为Balsa测试工具生成一个默认的LARD测试工具。这同样不需要多少LARD的知识。
3. 定制的LARD工具
   对于一些应用，有必要在LARD中写一个定制的测试工具。生成Makefile的测试工具可以当作模板来使用。

默认的测试工具通过在所有外部通道的重复握手来测试目标Balsa块；即使输入通道可能与数据文件连在一起，其每一次握手也可以接收到数值0。

buffer.2c仿真

在Makefile中调用适当的仿真命令可以运行仿真，并生成以下结果：

```
0: chan 'i': writing 0
6: chan 'i': writing 0
15: chan 'o': reading 0
19: chan 'i': writing 0
28: chan 'o': reading 0
32: chan 'i': writing 0
41: chan 'o': reading 0
45: chan 'i': writing 0
54: chan 'o': reading 0
58: chan 'i': writing 0
67: chan 'o': reading 0
```

```
71: chan 'i': writing 0
80: chan 'o': reading 0
```

仿真会一直运行直到终止它（按 Ctrl-C 键）。在每个通道的活动行（activity line）的左边显示的数据是仿真的次数。LARD 使用了单位延时模型，所以处理这些数值时应该注意。

数据文件仿真

以上特有的仿真激励内容不是很详尽。一种更好的方法是外部定义输入通道 *i* 上的数据，在下面的例子中，一个文件包含以下的测试数据集（可以采用多种数字的表达方式）。

```
1
0x10
022
0b011101
5
```

Makefile 可以产生一个命令来从这个激励文件运行仿真。现在如果运行仿真，会生成以下结果：

```
3: chan 'i': writing 1
15: chan 'o': reading 1
16: chan 'i': writing 16
28: chan 'o': reading 16
29: chan 'i': writing 18
41: chan 'o': reading 18
42: chan 'i': writing 29
54: chan 'o': reading 29
55: chan 'i': writing 5
67: chan 'o': reading 5
Program terminated
```

通道浏览器

在先前的例子中，以标准输出形式出现的仿真结果是文本格式的。LARD 有一个图形界面，它可以与内外部通道结合起来显示握手和数据值。假设对`balsa-md`已经指定了测试工具命令集，就可以调用通道浏览器，屏幕上会显示 2 个窗口：LARD 解释程序控制窗口和通道浏览器窗口。

开始仿真后，通道浏览窗口中会显示设计中不同通道的轨迹。在每一个通道中，会显示请求、应答信号和数据值，如图 9.8 所示。

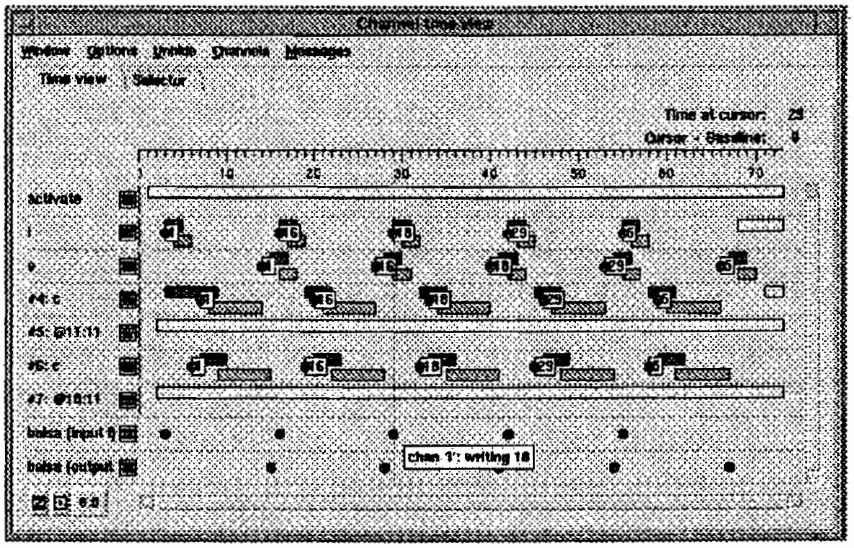

Channel time view
Window Options Unhide Channels Messages
Time view Selector
Time at cursor: 25
Cursor - Baseline: 0
1 10 20 30 40 50 60 70
activate
i
o
#4: c
#5: @11:11
#6: c
#7: @10:11
Index (input 0
Index (output
chan 1': writing 18

图 9.8 通道浏览器窗口

第10章　Balsa语言

本章概括地介绍了Balsa语言，并用一些设计实例来说明语言的各个方面。

10.1　数据类型

Balsa是基于位矢量数据类型的强类型语言。表达式的结果一定要确保在基本的位矢量表示范围以内。类型名可以指定也可以省略（即匿名的数据类型）。匿名的数据类型依赖于其大小及属性进行确定，而指定类型名的数据类型则由声明部分来说明。

匿名类型有两种：以关键字“bit”声明的数值类型和其他类型的数组，数值类型可以是有符号的也可以无符号。符号会影响表达式的运算符和计算。只有数值类型和其他类型的数组可以不对它们命名。Balsa有三种独立命名空间：第一种是程序和函数名，第二种是变量和通道名，第三种是类型声明。

数值类型

数值类型支持n位无符号数的范围是$[0, 2^n - 1]$，n位有符号数的范围是$[-2^{n-1}, 2^{n-1} - 1]$。经命名的数值类型也具有相同的取值范围，名称则是同一范围的别名。一个数值类型声明举例如下：

```
type word is 16 bits
```

该声明定义了一个新类型`word`，它是无符号的类型（无符号没有关键字），范围为$[0, 2^{16} - 1]$。同样地，一个符号类型可以声明为：

```
type sword is 16 signed bits
```

它定义了一个新类型`sword`，范围为$[-2^{15}, 2^{15} - 1]$。

唯一预定义的类型是位（bit）。而标准版的Balsa提供了一个声明库，包含了以下数据类型的声明：字节（byte）、半字节（nibble）、布尔量（boolean）和基数（cardinal）以及常量真（true）和假（false）。

枚举类型

枚举类型由被命名的数值组成。这些被命名的数值从0开始，从左到右逐个加1。可以通过指定确切数值来重新设置计数值，多个不同的名字可以赋予相同的数值，例如：

```
type Colour is enumeration
  Black, Brown, Red, Orange, Yellow, Green, Blue,
  Violet, Purple=Violet, Grey, Gray=Grey, White
end
```

Colour中Violet的值为7，和Purple的值相等。Grey和Gray的值都为8。元素的总数是10。也可以使用over关键字来规定固定位数的枚举类型：

```
type SillyExample is enumeration
  e1=1, e2 over 4 bits
end
```

这里用两位足够指定枚举的3种可能的值（0没有给出名称，e1的值是1，e2的值是2）。over关键字保证了枚举类型的表达实际上是4位。枚举类型在使用前必须通过类型声明进行命名。

常量

常量的值可以用表达式来定义，在编译时，这些表达式将被替换为实际的数值。常量在声明时，需要为其指定数据类型，如果不指定，默认为数值类型。例如：

```
constant minx = 5
constant maxx = minx + 10
constant hue = Red: Colour
constant colour = Colour'Green  -- explicit enumeration element
```

记录类型

记录是由命名的元素以位方式组成的，这些元素可能为不同的（预声明）类型，并且第一个元素在最低位。如：

```
type Resistor is record
  FirstBand, SecondBand, Multiplier: Colour;
  Tolerance: ToleranceColour
end
```

Resistor有4个元素：FirstBand，SecondBand，Multiplier（这3个是Colour类型）和ToleranceColour（Tolerance类型），Colour和Tolerance这两种类型都必须预先声明。FirstBand是第一个元素，它代表类型Resistor的最低位。可通过点符号来选择记录结构中的元素。因此，如果R15是Resistor类型的一个变量，R15.SecondBand代表了它的SecondBand值。和枚举类型一样，记录类型可以用over关键字。

数组类型

数组由数字索引的数值组成，这些数值具有相同的类型。以下是一个关于数组类型声明的例子：

```
type RegBank_t : array 0..7 of byte
```

这里定义了一个新类型 RegBank_t，它是由 8 个元素组成的数组类型，索引范围为[0, 7]，其中每个元素都是byte类型。范围指定符的顺序并不重要，如array 0..7和array 7..0是等价的。通常可用单个表达式expr来指定范围的大小：这等价于一个0..expr-1的范围。Balsa 允许使用匿名数组类型，所以变量可以声明为一个未定义的类型的数组：

```
variable RegBank : array 0..7 of byte
```

数组中指定范围内的元素可以通过调用数组片段（Array slices）来访问，例如a[5..7]表示取出a5，a6，a7三个元素。数组片段范围指示符与数组整体的范围指示符一样，顺序并不重要。Balsa 中的复合结构都是以从最低位到最高位、从左到右的方式来排列。数组片段返回值的索引范围是从索引号 0 开始标定的。

数组可以通过 tupling 机制构建，也可以由基本类型相同的其他数组组成：

```
variable a, b, c, d, e , f: byte
variable z2 : array 2 of byte
variable z4 : array 4 of byte
variable z6 : array 6 of byte
z4:= {a,b,c,d}              -- array construction
z6:= z4 @ {e,f}             -- array concatenation
z2:= (z4 @ {e, f}) [3..4] -- element extraction by array slicing
```

在最后一个例子中，z2的第一个元素是d，第二个元素是e。使用数组片段可以很方便的从其他类型中任意抽取位段（bit fields）。

数组通道

通道可以是数组形式，数组通道可能是由几个不同的通道组成，这些通道可以通过数字或枚举索引来指定。除了每个通道握手不同以外，通道数组与变量的数组类型非常类似。除了有相同通道名以外，每个通道与数组中的其他通道没有关系。数组通道的语法和数组类型的变量不同，这使得从数组通道来解释数组更加容易。例如：

```
array 4 of channel XYZ : array 4 of byte
```

声明了4个通道，XYZ[0]到XYZ[3]，每一个通道都是32位宽的array 0..3 of byte类型。在 9.4.4 节中有一个关于使用数组通道的例子。

10.2 数据类型相关问题

如上文所述，Balsa是强类型语言：赋值符号左、右两边的类型须相同。隐式类型转换的唯一形式是使数值和常量扩展至位数更长的数字类型。特别注意的是要保证算术运算的结果和声明过的结果类型一致。考虑赋值语句：`x := x + 1`，这不是一个正确的Balsa语句，因为结果可能比变量x多一位。如果要忽略这个加法的进位，必须通过明确的类型转换来截断。

类型转换

如果变量x声明为32位，以上赋值的正确格式为：

```
x := (x + 1 as 32 bits)
```

关键字as表示类型转换操作符。圆括号是这个语句的必要部分。如果需要两个32位数加法的进位，可以使用一个记录数据类型来存储合成的结果：

```
type AddResult is record
  Result: 32 bits;
  Carry: bit;
end
variable r: AddResult

r := (a + b as AddResult)
```

表达式`r.Carry`存放需要的进位位，`r.Result`是32位加法的结果。

当取位段时，也需要类型转换。这里是一个简单微处理器的指令译码器的例子。16位指令字中的低5位包含了5位有符号立即数。现在需要取出立即数段并将其带符号扩展至16位：

```
type Word is 16 signed bits
type Imm5 is 5 signed bits

variable Instr: 16 bits -- bottom 5 bits contain an immediate
variable Imm16: Word

Imm16 := (((Instr as array 16 of bit) [0..4] as Imm5) as Word)
```

首先，将指令字`Instr`转型成一个位数组，从这个位数组中，可以取出任意的子范围：

```
(Instr as array 16 of bit)
```

接下来，取出末端（最低的）5位：

```
(Instr as array 16 of bit) [0..4]
```

现在，把取出的这 5 位转型成 5 位有符号数：

```
((Instr as array 16 of bit) [0..4] as Imm5)
```

最后，把这 5 位有符号数扩展为 16 位的立即数：

```
(((Instr as array 16 of bit) [0..4] as Imm5) as Word)
```

这个过程需要两次类型转换，若将取出的这 5 位直接转换至 Word 类型的变量 Imm16，即使 Word 是一个有符号类型，也只会用 0 填补最高位。但是，从一个有符号数的类型转换为另一个（更宽的）有符号数的类型，会将较短的数值带符号的扩展至较长的目标类型的宽度。

在很多硬件设计中，常常需要从一个数域内取出某些位。一般在取位段之前，原始的数据首先要转为一个数组类型。分化（Smash）操作符“#”是把一个对象转换成位数组的快捷操作。因此，上面符号扩展的例子可以写得更简洁：

```
((#Instr [0..4] as Imm5) as Word)
```

自动赋值

语句格式：

```
x := f(x)
```

在 Balsa 中允许使用上式，但实现后会产生一个辅助变量，用来把值返回给程序设计者可见的变量。这个辅助变量包含在握手中，它不能同时进行读和写。由于自动赋值产生变量的数目可能会增加一倍，实际应用中最好不要使用自动赋值，应明确引入附加变量，然后修改程序来隐藏时序更新，从而避免时间损失。关于这一方法的例子见 10.6 节中的 count10b。

10.3　控制流和指令

表 10.1 列出了 Balsa 语言的指令集。

表 10.1　Balsa 语言指令

命令	备注
sync	只用于控制（无数据）握手
<-	从表达式到输出端口的握手数据传送
->	从输入端口到变量的握手数据传送
:=	变量到变量的赋值
;	顺序操作符
||	并行成分操作符
continue	no-op，空操作
halt	引起死锁（在仿真中有用）

（续表）

命令	备注
loop...end	不断重复
while...else...end	有条件重复
for...end	结构化的迭代（非时间的）
if...then...else...end	条件执行，可能有多个哨命令
case...end	基于常量表达式的条件执行
select	非仲裁式选择操作符
arbitrate	仲裁式选择操作符
print	编译期间打印诊断结果

无数据握手

sync<Channel>——在指定的通道上等待握手。在握手完成之前，电路不能动作。

通道通信

数据可以在变量与通道、通道与通道或通道与指令代码块之间进行传输，如下所示。

<Channel><-<Variable>——从一个变量到指定通道的传输数据。这个通道可能是程序中的内部通道或在程序声明中列出的输出端口。

<Channel>-><Variable>——从已连接的通道到变量的传输数据。这个通道可能是程序中的内部通道或是在程序声明中列出的输入端口。

<Channel1>-><Channel2>——在通道之间传输数据。

<Channel>->then<Command>——允许访问指令块中的数据。但是，直到指令块自身停止后，通道上的握手才能完成，数据才能释放。

变量赋值

<Variable> :=<Expresstion>——把表达式的结果赋给变量。表达式的结果类型必须和变量的类型相一致。

顺序成分

<Command1>; <Command2>——这两条指令是顺序执行的。第一个终止后，第二个才开始执行。

并行成分

<Command1>||<Command2>——由两条指令组成，它们独立地并行执行。只有在这两条指令完成后，后继电路才能发生动作。如果这两条指令具有相关性，可能会导致两条指令的执行依赖于其他指令的完成。“||”操作符比“;”操作符使用起来更加严格。如果不能满足需要，指令还可以像下面这样按块分组：

```
[<Command1>; <Command2>] || <Command3>
```

注意，使用方括号而不是圆括号对指令进行分组。另一种方法是使用关键字 begin...end，如果块中的局部变量没有声明，则必须用这种方法。

继续和暂停指令

继续（continue）是空操作指令。暂停（halt）指令使进程的线程进入死锁。

循环结构

循环（loop）指令可以引起代码块的无限循环。有限循环可以用 while 结构来构造。下面是一个简单的例子：

```
while <Condition> then <Command> end
```

不过，也可以允许有多个哨，所以这个结构更一般的格式是：

```
while
  <Condition1> then <Command1>
| <Condition1> then <Command2>
| <Condition3> then <Command3>
else
  <Command4>
end
```

while 结构可以带一个 else 从句，从而可以简便一些。以上代码可以写成下面的形式，与上面的代码只是在最终的握手电路实现上会有少许区别：

```
while
  <Condition1> then <Command1>
| <Condition2> then <Command2>
| <Condition3> then <Command3>
end;
<Command4>
```

如果多个哨同时满足，则无法确定要执行哪条指令。

结构循环

Balsa 具有结构化的循环。在许多程序设计语言中，循环写成 for 循环还是 while 循环，只是代码书写的便利与风格不同，但在 Balsa 语言中并非如此。for 循环类似于 VHDL 中的 for...generate 命令，它可以用来将重复结构展开。9.4.4 节中有一个介绍 for 循环用法的例子。10.6 节中的 count10e 例子是一个 for 命令使用不当的例子。使用“for ;”还是“for ||”的形式决定了结构重复是顺序地运行还是并发地运行。

条件执行

Balsa语言中有两种结构可以用来实现条件执行。Balsa的case语句和传统的程序设计语言中的case语句相似。单独的哨可能对应于哨表达式的多个值。

```
case x+y of
  1 .. 4, 11  then o <- x
| 5 .. 10 then o <- y
  else o <- z
end
```

另一种是if...then...else结构，它在执行时判断条件表达式的值并据此执行相应的语句。它的语法与while循环相似。注意if语句嵌套中的内在顺序，如下所示：

```
if <Condition1> then
  <Command1>
else
  if <Condition2>  then
    <Command2>
  end
end
```

条件Condition2的检测是在条件Condition1检测之后进行的。如果知道这两个条件是互斥的，则可以表示成：

```
if <Condition1> then <Command1>
|  <Condition2> then <Command2>
end
```

“|”分离符使得Condition1和Condition2可以同时检验。如果有多个哨（条件）同时满足，则结果无法确定。

10.4 二进制/单目运算符

表10.2是Balsa的单目运算符和双目运算符，按照优先级递减顺序排列。

表10.2 Balsa双目/单目运算符

符号	运算	有效类型	备注
-	记录索引	记录	
#	分化	任意	从任意类型中取一个值并将其转化为位数组
[]	数组索引	数组	索引可以是非常量（能产生许多硬件）
not, log,	单目运算符	数值	log只能用于常量并返回一个向上取整的函数值：如log15返回4

（续表）

符号	运算	有效类型	备注
-(unary)			"-" 返回的结果比自变量宽 1 位
^	幂运算	数值	
*, /, %	乘，除，求余	数值	只用于常量
+, -	加，减	数值	
@	串接	数组	
<, >, <=, =>	不等式	数值，牧举	
=, /=	等于，不等于	所有类型	通过符号扩展来对有符号数值类型进行比较
and	位与	数值	对于 if 和 while 的条件，Balsa 使用类型为 1 位，所以位操作与逻辑运算符相同
or, xor	位或，位异或	数值	

10.5　程序结构

文件结构

一个典型的设计通常由多个文件组成，这些文件包括子程序、类型和常量声明等，它们与顶层程序一起构成了完整的设计。通常情况下，顶层程序在文件的结尾部分。这个文件用于输入其他相关的设计文件。这种输入风格可以简单而有效的实现器件复用和把输入程序映射到器件，器件可以是预编译的握手电路或是现有库（可能是手工设计的）中的器件。声明有语法上的先后顺序（从左到右，自上而下），每一个声明都有它的定义范围，作用范围一般从声明处开始到当前（或输入）文件的结尾。因此，Balsa 与 C 语言或 Modula 语言一样，遵从简单的 "先定义后使用" 原则，虽然对器件的原形设计并不方便。

声明

声明引入新的类型名、常数名或程序名，并将其加入程序的整个命名空间中，其作用范围从声明处开始直到程序块（或文件——声明位于顶层）结束。有 3 个独立的命名空间：类型名空间、程序名空间和所有其他声明的命名空间。在顶层，只有常量名隶属于上面的最后一种命名空间。但是，变量和通道都可能包含在程序的局部声明中。如果在子程序（或内部程序块）中的声明和先前在子程序外部（或子程序文本）中的声明名字相同的话，则局部声明在程序模块的内部起作用，而外部声明将被忽略。

程序

程序构成了 Balsa 描述的主体。每个程序都有名称、端口集和相应的行为描述。"sync" 关键字引入了无数据通道。无数据通道和数据通道都可以作为 "数组通道" 的元素。数组通道允许其他功能不同的通道通过数字或枚举的索引来引用。程序中可以有一系列的局部声明，声明中也可以包含对其他程序、类型或常数的声明。

共享程序

一般来说，对程序的每次调用都会生成独立的硬件来实现对这个程序的例化。实际上，通过使用多路技术，程序可以被共享。程序调用时使用公共硬件，这样可以避免同一电路被复制，并允许共享发生。共享程序的用法将在10.6节共享硬件部分做进一步的讨论。

函数

在很多程序设计语言中，可以认为函数是带有返回值，但对外部没有影响的程序。但是，在Balsa语言中，函数和程序具有根本的区别。程序中定义的参数，确定了与程序定义的电路模块接口的握手通道；而函数参数只是返回值的表达式名，11.2.4节仲裁树设计中有使用函数定义的例子。

10.6 电路实例

本节将介绍用Balsa描述各种计数器的设计。在形式上，它们类似于传统的同步计数器描述，因此这些设计对异步系统的初学者来说并不复杂。而van Berkel提出的更复杂的脉动（systolic）计数器[14]，则更适合用异步方法。

本设计中，用于更新计数器状态的时钟被无数据的sync型通道aclk所取代。计数器在sync通道上发出握手请求，环境发出应答信号来响应，从而完成握手，并使计数器的状态更新。

16进制计数器

```
-- count16a.balsa: modulo 16 counter
import [balsa.types.basic]
procedure count16 (sync aclk; output count: nibble) is
  variable count_reg : nibble
begin
  loop
    sync aclk ;
    count <- count_reg ;
    count_reg := (count_reg + 1 as nibble)
  end
end
```

该计数器通过两条通道与外部环境相连接：无数据的aclk通道和输出当前计数器值的输出通道count。用于实现变量的内部寄存器count_reg和输出通道count是预定义类型nibble（半字节）。count_reg值增加后，结果必须转换到nibble类型。为了便于理解，本例忽略了初始化及复位操作。若使用LARD对本电路进行仿真，当访问未初始化变量时，会给出一个无碍的警告。

消除自动赋值

上例中所使用的自动赋值语句虽然简明直观，但是隐藏了这样一个事实：在大多数的后端编译过程中，会产生一个辅助变量，以实现无竞争的更新。通过在代码中明确指定辅助变量的优点在于它的更新可以与其他操作并行进行，如下例所示：

```
-- count16b.balsa: write-back overlaps output assignment
import [balsa.types.basic]

procedure count16 (sync aclk; output count: nibble) is
  variable count_reg, tmp: nibble
begin
  loop
    sync aclk;
    tmp := (count_reg + 1 as nibble) ||
    count <- count_reg;
    count_reg := tmp
  end
end
```

本例中，从计数寄存器到输出通道的传送与辅助映像寄存器的更新是并行的。涉及并行操作时，可能会增加一定的面积开销，而且在这种情况下对速度的优化空间很小，但该例说明了在源代码层做出折中的原则。

十进制计数器

以上基本计数器的描述可以进行简单的修改，使之成为一个十进制计数器。只需要添加简单的测试语句来检测内部寄存器何时达到它的最大值，然后将它复位到 0。

```
-- count10a.balsa: an asynchronous decade counter
import [balsa.types.basic]

type C_size is nibble
constant max_count = 9

procedure count10(sync aclk; output count: C_size) is
  variable count_reg: C_size
  variable tmp: C_size
begin
  loop
    sync aclk;
    if count_reg /= max_count then
```

```
      tmp := (count_reg + 1 as C_size)
    else
      tmp := 0
    end || count <- count_reg ;
    count_reg := tmp
  end -- loop
end -- begin
```

可预置数十进制加/减计数器

该例描述了一个可预置数的十进制加/减计数器，它具有本章前面所讨论过的Balsa语言的许多特点。这个计数器需要两个控制位，一个用于加减控制，另一个用来确定计数器在下次操作时是进行置数还是进行加/减计数。实现这种计数器的方法很多，在count10b例子中，控制位和需预置的数据捆绑在一个信号通道中，记为in_sigs。

```
-- count10b.balsa: an aysnchronous up/down decade counter
import [balsa.types.basic]

type C_size is nibble
constant max_count = 9

type dir is enumeration down, up end
type mode is enumeration load, count end
type In_bundle is record
  data : C_size ;
  mode : mode;
  dir : dir
end

procedure updown10 (
  input in_sigs: In_bundle;
  output count: C_size
) is
  variable count_reg: C_size
  variable tmp: In_bundle
begin
  loop
    in_sigs -> tmp; -- read control+data bundle
    if tmp.mode = count then
      case tmp.dir of
        down then  -- counting down
```

```
        if count_reg /= 0 then
          tmp.data := (count_reg - 1 as C_size)
        else
          tmp.data := max_count
        end -- if
      | up then -- counting up
        if count_reg /= max_count then
              tmp.data := (count_reg + 1 as C_size)
        else
          tmp.data := 0
        end -- if
      end -- case tmp.dir
    end; -- if
    count <- tmp.data || count_reg := tmp.data
  end -- loop
end
```

上例说明了if...then...else和case控制结构的使用方法，同样说明了记录类型和枚举类型的用法。在枚举类型中使用符号使代码更具可读性。由Balsa系统生成的测试工具也可以读取符号枚举值。例如，下面是一个测试文件，它可以将计数器初始化为8并进行加法计数，测试计数器循环计数到零；然后进行减法计数，让用户检查计数器是否正确地返回9。

```
{8, load, up}       load counter with 8
{0, count, up}      count to 9
{0, count, up}      count & wrap to 0
{0, count, up}      count to 1
{0, count, down}    count down to 0
{0, count, down}    count down to 9
{1, load, down}     load counter with 1
{0, count, down}    count down to 0
{0, count, down}    count down & wrap to 9
```

共享硬件

在Balsa语言描述中，每条语句都会在最终电路中例化为硬件。因此，检查代码是否存在重复结构显得很有必要，重复的结构可被移到代码中的公共部分或用共享程序来代替。在上面的“count10b”中，例化了两个加法器：一个用于加法，另一个用于减法。由于这两个单元并不同时使用，因此可以通过共享程序让一个加法器完成这两项功能（根据计数方向决定+1或-1），从而节约面积。以下代码说明了如何利用共享程序重新编写count10b。共享程序

add_sub通过现有的计数值与变量inc相加来计算下一个计数值，inc可以是+1也可以是-1。注意，为了符合要求，inc必须声明为2位有符号数。

这种方法的面积优势可以通过breeze-cost产生的报告来观察：count10b需要2141个单元，而采用共享程序方法只需要1760个单元。当然，若计数器长度增加，这个优势就会变得更加明显。

```
-- count10c.balsa: introducing shared procedures
import [balsa.types.basic]

type C_size is nibble
constant max_count = 9

type dir is enumeration down, up end
type mode is enumeration load, count end
type inc is 2 signed bits

type In_bundle is record
  data: C_size ;
  mode: mode;
  dir: dir
end

procedure updown10 (
  input in_sigs: In_bundle;
  output count: C_size
) is
  variable count_reg: C_size
  variable tmp: In_bundle
  variable inc: inc

  shared add_sub is
  begin
    tmp.data := (count_reg + inc as C_size)
  end

begin
  loop
    in_sigs -> tmp; -- read control+data bundle
    if tmp.mode = count then
```

```
      case tmp.dir of
      down then -- counting down
        if count_reg /= 0 then
          inc := -1;
          add_sub()
        else
          tmp.data := max_count
        end -- if
      | up then -- counting up
        if count_reg /= max_count then
          inc := +1;
          add_sub()
        else
          tmp.data := 0
        end -- if
      end -- case tmp.dir
    end; -- if
    count <- tmp.data || count_reg := tmp.data
  end -- loop
end
```

为了确保正确地实现功能，使用共享程序时要加上一些限制：

- 对于共享程序的使用不可以有任何冲突；
- 共享程序不能使用局部通道；
- select 语句（见 10.7 节）中，共享程序使用的通道元件必须在 select 块内部声明为局部变量。

“while” 循环结构

十进制计数器的另一种描述方法是采用 while 循环结构：

```
-- count10d.balsa: mod-10 counter alternative implementation
import [balsa.types.basic]

type C_size is nibble
constant max_count = 10

procedure count10(sync aclk; output count: C_size) is
  variable count_reg: C_size
begin
```

```
  loop
    while count_reg < max_count then
      sync aclk;
      count <- count_reg;
      count_reg := (count_reg + 1 as C_size)
    end; -- while
    count_reg := 0
  end -- loop
end
```

"for" 循环结构

for循环对Balsa语言的初学者来说，可能容易出错。在很多程序设计语言中，while循环和for循环可以交替使用。但Balsa语言中不是这样：for循环实现了结构上的重复，换句话说，每次进入循环都会产生独立的硬件结构。以下描述中，表面上看类似于先前count10d例中的while循环，似乎是正确的。对它进行编译，也没有问题，使用LARD仿真也好像给出了正确的行为。但是，面积检测的结果是11 577个，比先前增加了许多。理解为什么会出现这个问题是很重要的，for循环结构在编译时重复的创建了10个加法计数器，循环的每次例化都是顺序执行的。握手电路图的PostScript图很难看懂；将max_count设置为3，则可以生成一张容易看懂的图。

```
-- count10e.balsa: beware the "for" construct
import [balsa.types.basic]

type C_size is nibble
constant max_count = 10

procedure count10(sync aclk; output count: C_size) is
  variable count_reg: C_size
begin
  loop
    for ; i in 1 .. max_count then
      sync aclk;
      count <- count_reg;
      count_reg := (count_reg + 1 as C_size)
    end; -- for ; i
    count_reg := 0
  end -- loop
end -- begin
```

如果以并行 for 结构（for || ...）替代顺序 for 结构，编译器会给出一条有关并行线程读、写冲突的错误信息。在这种情况下，所有计数器电路会同时更新计数寄存器，这可能会形成冲突。如果读者能较好地理解最终可能生成的握手电路，那么将有助于理解 Balsa 语言的方法学。

10.7　通道选择

下面描述的异步电路将两个输入通道并为一个输出通道，可以认为它是一个自选择的多路选择器。select 语句等待通道的数据到达，选择相应的输入通道 a 或 b。当握手在通道 a 或 b 上出现时，数据会在输入通道上保持有效，并且直到 select...end 块结束后才完成握手。

这个电路是封闭式（enclosure）握手的一个例子，而且不需要生成用于存放输入通道数据的内部锁存器。不过，可能存在的缺点是：因为握手存在延时，输入通道不会被立即释放，从而不能继续独立地进行其他操作。在这个例子中，数据被传送到输出通道上，并且一旦输出通道上的数据被移走后，输入握手会立即完成。在 11.2.2 节中，将会在人口计数器（population counter）的代码中见到更大范围封闭式握手的例子。

```
-- mux1.balsa: unbuffered Merge
import [balsa.types.basic]

procedure mux (input a, b:byte; output c:byte) is
begin
  loop
    select a then c <- a  -- channel behaves like a variable
    |      b then c <- b  -- ditto
    end -- select
  end -- loop
end
```

因为与 select 相关的握手有封闭特性，所以在执行由 select 封闭的代码块时，输入 a 和 b 应该是互斥的。在多数情况下，满足这一要求并不困难。然而，如果 a 和 b 是完全独立的，select 应该由可以作出仲裁选择的 arbitrate 语句替代。在速度方面，仲裁器较慢，并且在某些技术中很难实现。此外，带有仲裁器的电路具有不确定性，这会导致测试和设计的正确性验证成为问题。所以，设计者应当谨慎使用仲裁器。

```
-- mux2.balsa: unbuffered arbitrated Merge.
import [balsa.types.basic]
```

```
procedure mux (input a, b:byte; output c:byte) is
begin
  loop
    arbitrate a then c <- a  -- channel behaves like a variable
    |         b then c <- b  -- ditto
    end -- arbitrate
  end -- loop
end
```

第11章　建立库元件

11.1　参数化描述

参数化程序允许设计者开发一个常用的元器件库，然后可以通过改变参数来实现不同元件的例化。举一个简单的例子，可以把数据宽度不确定的缓冲器描述为一个库元件。相应的，可以用库中定义的缓冲器的例化来组成流水线，不需要任何流水线深度选择方面的知识。

11.1.1　可变宽度缓冲器定义

下面的pbuffer1例子是带宽度参数的单级缓冲器。

```
-- pbuffer1.balsa - parameterised buffer example
import [balsa.types.basic]

procedure Buffer (
  parameter X : type ;
  input i : X ; output o : X
) is
  variable x: X
begin
  loop
    i -> x ;
    o <- x
  end
end

-- now define a byte wide buffer
procedure Buffer8 is Buffer (byte)

-- now use the definition
procedure test1 (input a : byte ; output b : byte) is
begin
  Buffer8 (a, b)
end
```

```
-- alternatively
procedure test2 (input a : byte ; output b : byte) is
begin
  Buffer (byte, a, b)
end
```

对9.4.1节定义的单级缓冲器通过增加参数声明来进行修改，这个参数声明为X，类型是type型。换句话说，X的类型需要在后面进行严格定义。一旦声明了参数类型，就可以在后面的声明和语句中使用：例如，输入通道i可以定义为类型X。参数化程序定义本身不生成任何硬件。

已经定义了的程序可以在其他程序定义中使用。Buffer8定义了一个长度为单字节的缓冲器，如程序test1所示，需要的时候将它例化即可。另一种方法如test2所示，可以直接使用参数化程序来具体实现。

11.1.2 宽度与深度可变的流水线

下例说明了在一个程序中如何定义多个参数。由参数化缓冲器组成的流水线，其深度也是参数。

```
-- pbuffer2.balsa - parameterised pipeline example
import [balsa.types.basic]
import [pbuffer1]

-- BufferN: an n-place parameterised, variable width buffer
procedure BufferN (
  parameter n: cardinal ;
  parameter X: type ;
  input i: X ;
  output o: X
) is
begin
  if n = 1 then -- single place pipeline
    Buffer(X, i, o)
  | n >= 2 then -- parallel evaluation
    local array 1 .. n-1 of channel c: X
    begin
      Buffer(x, i, c[1])    ||  -- first buffer
      Buffer(x, c[n-1], o) ||  -- last buffer
      for || i in 1 .. n-2 then
        Buffer(X, c[i], c[i+1])
```

```
      end -- for || i
    end
  else print error, "zero length pipeline specified"
  end -- if

end

-- Now define a 4 deep, byte-wide pipeline.
procedure Buffer4 is BufferN(4, byte)
```

Buffer是前面例子中的宽度为参数的单级缓冲器，可以通过库语句import[pbuffer1]来调用。在这个程序的代码中，BufferN的定义方式与9.4.4节中的例子很相似，只是流水线的级数*n*不是常数而是cardinal类型的参数。这个定义中包括了某些错误诊断程序。在定义时，如果试图构建长度为0的流水线，就会显示出错误信息。

11.2　递归定义

Balsa在定义时允许使用递归形式（只要在编译时，最终的结构是静态确定的）。这种方法可以很好地描述许多结构，并且自然地扩展了强大的参数化机制。本章其余部分将通过几个有用的例子来阐述递归参数。

11.2.1　n 路选择器

一个*n*路选择器可以用多个2路选择器通过树形结构的方式来构造。递归定义要求其自身就是规范的技术：一个*n*路选择器可以分解为两个*n*/2路选择器，这两个*n*/2路选择器通过内部通道连接到一个2路选择器上，如图11.1所示。

```
--- Pmux1.balsa: A recursive parameterised MUX definition
import [balsa.types.basic]

procedure PMux (
  parameter X : type;
  parameter n : cardinal;
  array n of input inp : X;  -- note use of arrayed port
  output out : X
) is
begin
  if n = 0 then print error, "Parameter n should not be zero"
  | n = 1 then
    loop
```

```
      select inp[0] then
        out <- inp[0]
      end -- select
    end -- loop
  | n = 2 then
    loop
      select inp[0] then
        out <- inp[0]
      | inp[1] then
        out <- inp[1]
      end -- select
    end -- loop
  else
    local -- local block with local definitions
      channel out0, out1: X
      constant mid = n/2
    begin
      PMux (X, mid,   inp[0..mid-1], out0) ||
      PMux (X, n-mid, inp[mid..n-1], out1) ||
      PMux (X, 2, {out0, out1}, out)
    end
  end -- if
end

-- Here is a 5-way multiplexer
procedure PMux5Byte is PMux(byte, 5)
```

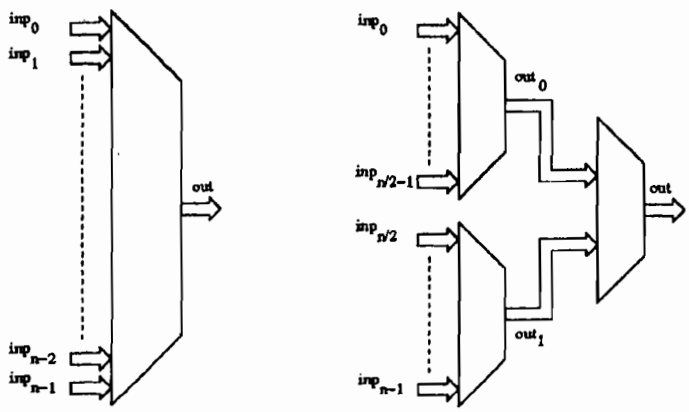


图 11.1 n 路选择器的分解

代码注释

这个多路选择器对输入类型和通道个数n进行参数化，代码很直观。一个输入超过2路的多路选择器分解为两个输入个数只有原来一半的多路选择器，并通过内部通道连接到2选1选择器。注意数组通道 out0 和 out1 是如何组合成一个数组对的。当输入个数是2或1时停止递归分解过程（定义一个0输入的多路选择器会产生错误）。1输入的多路选择器对输入不进行选择。

Balsa 测试工具

以下代码说明了如何利用简单的Balsa程序生成多路选择器的测试激励。测试程序其实很简单。

```
-- test_pmux.balsa - A test-harness for Pmux1
import [balsa.types.basic]
import [pmux1]

procedure test (output out : byte) is
  type ttype is sizeof byte + 1 bits
  array 5 of channel inp: byte
  variable i: ttype
begin
  begin
    i := 1;
    while i <= 0x80 then
      inp[0] <- (i as byte);
      inp[1] <- (i+1 as byte);
      inp[2] <- (i+2 as byte);
      inp[3] <- (i+3 as byte);
      inp[4] <- (i+4 as byte);
      i := (i + i as ttype)
    end
  end || PMux5Byte(inp, out)
end
```

11.2.2　人口（population）计数器

下例是计算一个字中置位的个数的例子。Amulet 处理器中需要用它来确定在执行 LDM/STM(Load/Store Multiple)指令时，用于还原 / 保存的寄存器个数。

采用的方法是把问题分成两部分。首先，把相邻的位相加，形成一个由2位宽度的通道组成的数组，表示每相邻对被置位的个数。然后，把由2位数字组成的数组加入到以递归定义的加法器树（AddTree 程序）中。位计数器的结构如图 11.2 所示。

```
-- popcount: count the number of bits set in a word
import [balsa.types.basic]

procedure AddTree (
  parameter inputCount : cardinal;
  parameter inputSize : cardinal;
  parameter outputSize : cardinal;
  array inputCount of input i : inputSize bits;
  output o : outputSize bits
) is
begin
  if inputCount = 1 then
    select i[0] then o <- (i[0] as outputSize) end
  | inputCount = 2 then
    select i[0], i[1] then
      o <- (i[0] + i[1] as outputSize bits)
    end -- select
  else
    local
      constant lowHalfInputCount = inputCount / 2
      constant highHalfInputCount = inputCount - lowHalfInputCount

      channel lowO, highO: outputSize - 1 bits
    begin
      AddTree (lowHalfInputCount, inputSize, outputSize - 1,
        i[0..lowHalfInputCount-1], lowO) ||
      AddTree (highHalfInputCount, inputSize, outputSize - 1,
        i[lowHalfInputCount..inputCount-1], highO) ||
      AddTree (2, outputSize - 1, outputSize, {lowO, highO}, o)
    end
  end -- if
end

procedure PopulationCount (
  parameter n: cardinal;
  input i: n bits;
  output o: log (n+1) bits
) is
begin
  if n % 2 = 1 then
```

```
    print error, "number of bits must be even"
    end; -- if
    loop
      select i then
        if n = 1 then
          o <- i
        | n = 2 then
          o <- (#i[0] + #i[1]) add bits 0 and 1
        else
          local
            constant pairCount = n - (n / 2)
            array pairCount of channel addedPairs: 2 bits
          begin
            for || c in 0..pairCount-1 then
              addedPairs[c] <- (#i[c*2] + #i[(c*2)+1])
            end ||
            AddTree (pairCount, 2, log (n+1), addedPairs, o)
          end
        end -- if
      end -- select
    end -- loop
end

procedure PopCount16 is PopulationCount (16)
```

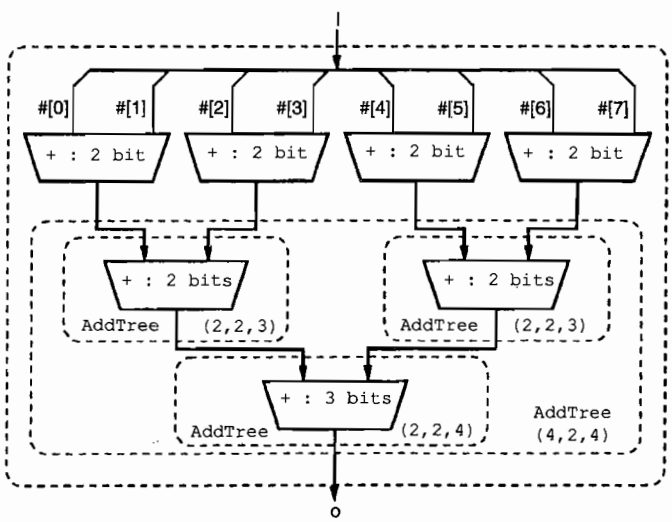


图 11.2　位人口计数器的结构

代码注释

参数： AddTree和PopulationCount都是参数化程序。PopulationCount可以对任何长度字中置位的个数进行计算。参数化程序AddTree中定义了一个具有任意位宽的递归加法器。

半封闭选择： select语句的封闭握手语义允许输入i在select块中出现多次，而不需要内部锁存器。

避免死锁： 注意相邻位求和的队列使用了并行的for循环。

```
for || c in 0..pairCount-1 then
  addedPairs[c] <- (#i[c*2] + #i[(c*2)+1])
end
```

也许会这样想，如果使用串行的“for;”循环语句，会使得电路的速度变慢。其实不是速度变慢，而是会使系统出现死锁，这也说明了为什么设计异步电路一定要真正理解设计方法学。在这种情况下，在与addPairs数组连接的加法器完成加法并且释放输入数据之前，在输入端应该已经有一对准备好的数据。但是，如果串行计算相邻位的和，后面的一对数要等到前面的一对数的握手完成后才会进行计算——这是不可能的，因为AddTree要等待所有数都有效，结果导致了死锁。

11.2.3 Balsa移位寄存器

通用移位寄存器是所有微处理器中必不可少的组成部分，Amulet处理器也不例外。以下介绍这种移位寄存器的主要部分，虽然它只能实现循环右移功能，但很容易扩展其他的移位功能。

移位寄存器的主要工作由局部程序rorBody完成，它可以采用递归的形式生成一个子移位寄存器，移位寄存器位数在1, 2, 4, 8…中任意选择。该移位寄存器的结构如图11.3所示。

```
import [balsa.types.basic]

-- ror: rotate right shifter
procedure ror (
  parameter X : type;
  input d : sizeof X bits;
  input i : X;
  output o : X
) is
begin
  loop
```

```
    select d then
      local
        constant typeWidth = sizeof X

        procedure rorBody (
          parameter distance: cardinal;
          input i : X;
          output o : X
        ) is
        local
          procedure rorStage (
            output o : X
          ) is
          begin
            select i then
              if #d[log distance] then
                o <- (#i[typeWidth-1..distance] @
                      #i[distance-1..0] as X) {shift}
              else
                o <- i {don't shift}
              end -- if
            end -- select
          end
          channel c : X
        begin
          if distance > 1 then
            rorStage (c) ||
            rorBody (distance/2, c, o)
          else
            rorStage (o)
          end -- if
        end
      begin
        rorBody (typeWidth/2, i, o)
      end
    end -- select
  end -- loop
end

procedure ror32 is ror (32 bits)
```

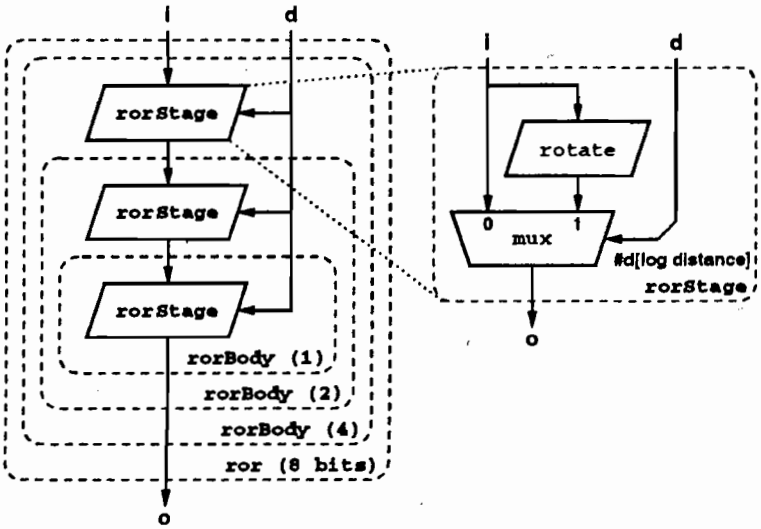


图 11.3 循环移位寄存器的结构

移位寄存器测试

下例中的代码是用 Balsa 编写的，用来测试这一移位寄存器的测试程序。

```
import [balsa.types.basic]
import [ror]

--test ror32
procedure test_ror32(output o : 32 bits)
is
  variable i : 5 bits
  channel shiftchan : 32 bits
  channel distchan : 5 bits
begin
  begin
    i := 1;
    while i < 31 then
      shiftchan <- 7 || distchan <- i;
      i := (i+1 as 5 bits)
    end -- while
  end || ror32(distchan, shiftchan, o)
end
```

11.2.4　仲裁树

最后的例子是一个参数化的仲裁树。这个电路是第12章中DMA控制器的一部分。一个8输入仲裁器的结构如图 12.3 所示。ArbFunnel 是一个参数化的树，它由两种元素组成：ArbHead和ArbTree。每一对进来的sync请求经过ArbHead元素的仲裁合成为一位判定结果。然后由ArbTree元素仲裁这些一位的通道。ArbTree取出一些输入（i端口上）的判定位，产生一排2输入的仲裁器，每个仲裁器还有一个判定位，这将使问题的复杂度降低至以前输入的一半。递归调用ArbTree可以将输入通道的个数降到1（最后的判定结果返回到输出端口 o）。

```
-- ArbHead: 2 way arbcall with channel no. output
import [balsa.types.basic]
procedure ArbHead (
  sync i0, i1;
  output o: bit
) is
begin
  loop
    arbitrate i0 then o <- 0
    |         i1 then o <- 1
    end -- arbitrate
  end -- loop
end -- begin

-- ArbTree: a tree arbcall which outputs a channel number
-- prepended onto the input channel°øs data. (invokes itself
-- recursively to make the tree)

procedure ArbTree (
  parameter inputCount: cardinal;
  parameter depth: cardinal; -- bits to carry from inputs
  array inputCount of input i: depth bits;
  output o: (log inputCount) + depth bits
) is
  type BitArray is array 1 of bit
  type BitArray2 is array 2 of bit
  function AddTopBit (hd : bit; tl : depth bits) =
    (#tl @ {hd} as depth + 1 bits)
  function AddTopBit2 (hd : bit; tl : depth + 1 bits) =
```

```
    (#tl @ {hd} as depth + 2 bits)
  function AddTop2Bits (hd0 : bit; hd1 : bit; tl : depth bits) =
    (#tl @ {hd0, hd1} as depth + 2 bits)
begin
  case inputCount of
    0, 1 then print error, "Can't build an ArbTree with fewer than 2 inputs"
  | 2 then loop
      arbitrate i[0] -> i0 then o <- AddTopBit (0, i0)
      |         i[1] -> i1 then o <- AddTopBit (1, i1)
      end -- arbitrate
    end -- loop
  | 3 then local channel lo : 1 + depth bits
    begin
      ArbTree (2, depth, i[0 .. 1], lo) ||
      loop
        arbitrate lo then o <- AddTopBit2 (0, lo)
        | i[2] -> i2 then o <- AddTop2Bits (1, 0, i2)
        end -- arbitrate
      end -- loop
    end
  else local
      constant halfCount = inputCount / 2
      constant halfBits = depth + log halfCount
      channel l, r : halfBits bits
    begin
      ArbTree (halfCount, depth, i[0 .. halfCount-1], l) ||
      ArbTree (inputCount - halfCount, depth,
        i[halfCount .. inputCount-1], r) ||
      ArbTree (2, halfBits, {l, r}, o)
    end -- begin
  end -- case inputCount
end

-- ArbFunnel: build a tree arbcall (balanced apart from the last
-- channel which is faster than the rest) which produces a chann
-- number from an array of sync inputs
procedure ArbFunnel (
  parameter inputCount : cardinal;
  array inputCount of sync i;
  output o : log inputCount bits
```

```
) is
  constant halfCount = inputCount / 2
  constant oddInputCount = inputCount % 2
begin
  if inputCount < 2 then
    print error, "can't build an ArbFunnel with fewer than 2 inputs"
  | inputCount = 2 then
    ArbHead (i[0], i[1], o)
  | inputCount > 2 then
    local
      array halfCount + 1 of channel li : bit
    begin
      for || j in 0 .. halfCount - 1 then
        ArbHead (i[j*2], i[j*2+1], li[j])
      end ||
      if oddInputCount then
        ArbTree (halfCount + 1, 1, li[0 .. halfCount], o) ||
        loop
          select i[inputCount - 1] then li[halfCount] <- 0
          end -- select
        end -- loop
      else
        ArbTree (halfCount, 1, li[0 .. halfCount-1], o)
      end -- if
    end
  end -- if
end
```

第 12 章　一个简单的 DMA 控制器

本章将介绍一个简单的 4 通道 DMA 控制器，这是一个完全用 Balsa 语言编写的规模适中的实际设计，并且可以根据 Balsa 支持的各种后端工艺进行编译。读者会注意到这个控制器和第 15 章中介绍的 Amulet3i 的 DMA 控制器是不一样的。有关该控制器更详细的说明与设计动机可参见文献[8]。该控制器的完整代码可以从文献[7]所给的网址下载。

简化过的控制器提供：

- 4 个具有完整寻址范围的通道，它们带有独立的源寄存器、目的寄存器和计数寄存器。
- 8 个客户端 DMA 请求输入，以及相应的应答。
- 外设到外设、存储器到存储器和外设到存储器的传输。每个通道都有源和目的客户端请求。“真实的”外设到外设的传输，可以通过等待双方的请求来完成。

图 12.1 从程序设计者的角度给出了控制器中寄存器内存映射。寄存器组可以分为两类：通道寄存器和全局寄存器。

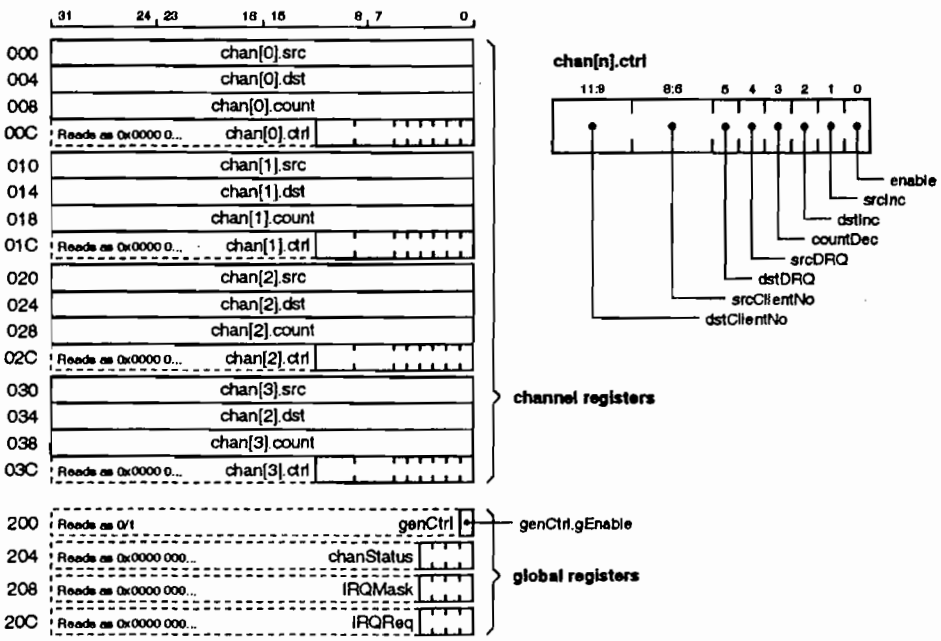


图 12.1　DMA 控制器程序设计者的模型

12.1　全局寄存器

全局寄存器存储控制状态，包括当前活动通道的控制状态和由终止传输产生的中断状态。本例中有以下 4 个全局寄存器。

genCtrl:常规控制。在这个DMA控制器中，常规控制寄存器只有1位：全局使能信号——gEnable。这个全局使能信号是唯一在系统上电时复位的控制位。其他所有的控制状态位在gEnable被置位前必须初始化。这样用全局使能信号可以使 Balsa 描述的初始化部分变得简单。

chanStatus:通道运行结束状态。chanStatus寄存器有4位，DMA的每个通道都对应一位。当寄存器中的某位被 DMA 控制器置位时，表示与这一位对应的通道已经结束了传送。

IRQMask, IRQReq:中断屏蔽和中断状态。IRQMask 寄存器为每个通道分配一位（像chanStatus那样），当某一位被置1时，则说明相应通道在完成传送后（当对应的chanStatus 位被置位时）会发出一个中断。IRQReq包含了每一条通道的当前中断状态。

通道的状态位、IRQ屏蔽位及IRQ状态位都保存在全局寄存器中，以便减少DMA寄存器的数量。当接收到中断请求后，必须由 CPU 读取 DMA 寄存器的值来判断服务的通道。

12.2　通道寄存器

与Amulet3i的DMA控制器一样，每个通道都有4个寄存器。两个地址寄存器（channel[*n*].src和channel[*n*].dst）包含了传送用的32位源地址和目的地址。当通道传送时，计数寄存器（channel[*n*].count），进行32位计数，计数寄存器的值减到0时，传送终止。控制寄存器（channel[*n*].ctrl）完成另外3个寄存器和与通道连接的客户端内容的更新。控制寄存器写入时，本通道上的中断及运行结束标志被清除。控制寄存器有以下 8 位。

enable:传输使能。如果使能位被置位，当一个新的DMA请求到来时，这条通道可以用于传输。通道使能在系统上电时不会被清0。genCtrl.gEnable位可以用来防止传输通道使能位在启动时被清除。

srcInc, dstInc, countDec:加/减控制。这些位用于源、目的寄存器及计数寄存器传输后的更新使能。如果（分别地）置位srcInc和dstInc，那么在传输后，源和目的寄存器的值加4（因为在这种控制器中只支持字传输）。注意这些地址的低 2 位受到保护。如果置位countDec，那么在每次传送后计数寄存器减 1。复位srcInc或dstInc将导致传送过程中相应的地址保持不变。这在指定外设（而不是存储器）地址时很有用。复位countDec将引起“不同步（free-running）”传输。

srcDRQ, dstDRQ:初始化 DMA 请求。初始化 DMA 请求。当接收到源客户端和目的客户端[请求等待（requests-pending）寄存器]的一对 DMA 请求时，则可以在通道上开启传输。srcDRQ, dstDRQ位为这对请求信号指定初始状态。若这两位都被置位时，则表明源和目的请求应该都已经到达。若其中一位或者两位被复位，表明与之对应的名为{src,dst}ClientNo号的客户端应该发出一次传送请求（当两位都被复位时，需要两个客户端的请求）。

srcClientNo, dstClientNo:客户端到通道的映射。这些位指定了通道接收源和目的的DMA请求的客户端编号。这些位只有当srcDRQ或dstDRQ复位时（或两个同时复位）才可以使用。

12.3 DMA 控制器结构

简化的 DMA 控制器的结构如图 12.2 所示，它由以下 5 个单元组成。

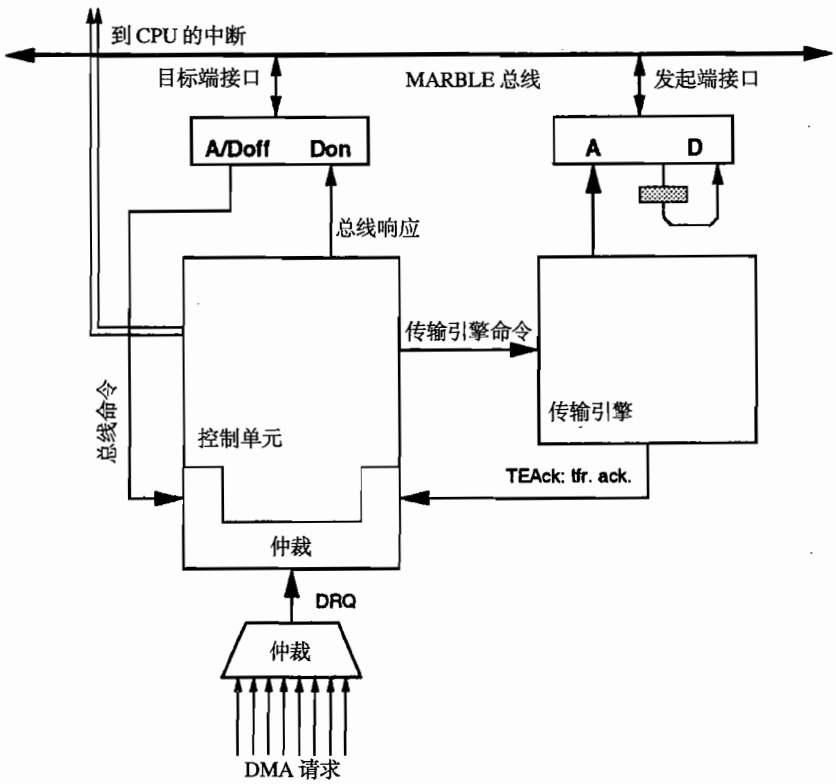


图 12.2 DMA 控制器结构

MARBLE 目标接口

假设控制器连接在MARBLE异步总线上，而MARBLE总线是用来连接Amulet3i片上系统（见第15章）中各个子系统的。MARBLE总线与其他形式的片上系统总线的接口相比较要简单。

MARBLE目标接口提供了外设与MARBLE总线的连接。通过这个连接，可以对控制器进行编程。需要对来自于DMA请求和来自于传输引擎中的“传送应答”的信号进行仲裁，然后确定哪个信号通过这个接口对寄存器进行访问。仲裁器和控制单元与传输引擎的分离，使DMA控制器能够避免潜在的因总线访问导致死锁的情况。

与Amulet3i的DMA控制器类似，这里用到的MARBLE接口也具有一个8位的地址（8位字地址，10位字节地址）。这使得通道寄存器可以具有相同的地址映射，并且可以将通道数扩展至32个而不改变全局寄存器地址。

MARBLE 发起端接口

DMA 控制器利用发起端接口来进行传送。只有这个接口上的地址和控制位才能连接到Balsa综合后的控制器。通过锁存器（在图 12.2 中用浅色的方框表示）来处理发起端流入和流出的数据。因为只支持字的传输，所以需要这个锁存器在读、写总线时保持数据值。要支持不同的传输长度并不难，但为了简化代码，本例并没有包含该功能。

控制单元

DMA的每个通道都有两位：“请求等待（request-pending）”位，为该通道的源和目的记录请求信号的到来。当收到传输请求时，控制单元轮流检测每条通道的请求等待寄存器，查出要执行传输的通道。执行传输时，该通道的寄存器内容会传送到传输引擎，并且更新寄存器内容以反映地址增加和计数减小。在没有发送传输引擎指令，或者传输指令已经发送到传输引擎时，DMA请求会直接得到应答。对DMA请求信号的应答并不能保证相关的传输能及时完成，为此外设必须检测总线的访问。应答只用于确认DMA传输请求的到达。在产生应答信号之后必须把请求标志清除，这样才可以接收其他的请求，并通过请求仲裁树标出其他可能进行传输的通道。

传输引擎

当执行 DMA 传输而没有 DMA 请求映射或者自动清除掉本次 DMA 请求时，控制器的传输引擎会从控制单元中取得命令。之所以采用传输引擎，唯一原因是为了防止潜在的总线死锁，即当DMA控制器试图通过MARBLE总线实现传输时，正好有一个通过MARBLE总线对寄存器组的访问也在进行。在这种情况下，总线的控制权属于试图去访问DMA控制器的发起端（通常是CPU）。而在DMA控制器试图为自己获得总线控制权时，发起端不能继续进行总线操作，于是形成了总线死锁。有了传输引擎，可以在传输操作中把DMA请求与CPU访问分开，当传输引擎在等待总线空闲时，控制单元可以自由地满足发起端寄存器的请求。

执行一次传输后，传输引擎会发信号到控制单元，让控制单元提供新的传输指令；这是通过在传输应答通道（在图 12.2 中标记为 `TEAck`）上的握手来实现的。通过控制单元的指令仲

裁器，来告知控制单元：传输引擎空闲，并从请求等待寄存器中选出下一次合适的传输者。应答信号不但提供了实现存储器到存储器传输所需要的自循环操作，而且当传输引擎忙时，允许循环响应其他类型的传输请求。

在控制单元中有一个标志寄存器，TEBusy，用于记录传输引擎的状态。当正在进行传输时，指令不能发送到传输引擎。当一条传输指令发送到传输引擎时，标志寄存器置1；当控制单元接收到传输应答时，标志寄存器清0。如果TEBusy置1，不会重新检查请求等待寄存器（并发送一条传输指令）。

仲裁树

DMA控制器接受来自于8条sync通道的DMA请求，这8条通道连接于仲裁单元的输入端，如图12.2所示。仲裁单元是由2路仲裁元所组成的仲裁树，将这8输入合并为一个DMA请求号，并把它传送到控制单元。控制单元一旦记录下DMA请求，就会立即响应请求。只有外设之间成功的数据传输，才能作为DMA操作实际完成的标志。当传输开始时（也就是从控制单元传送到传输引擎），传输的通道寄存器和请求等待寄存器都要在控制单元接受新的仲裁访问之前完成更新。因此，只要发生与任何传输相关的总线操作，则在此通道的传输请求会立刻到达，并且可以正确地检测到。

12.4 Balsa描述

DMA控制器的Balsa描述由3部分组成：仲裁树、控制单元和传输引擎。两个MARBLE接口位于Balsa块的外部，并且通过目标（mta和mtd）端口（与指令和响应端口对应）和发起端地址/控制（mia）端口实现控制。DMA控制器的顶层描述如下。

```
procedure DMAArb is ArbFunnel (NoOfClients)

procedure dma (
  input mta : MARBLE8bACommand;
  output mtd : MARBLEResponse;
  output mia : MARBLECommandNoData;
  output irq : bit;
  array NoOfClients of sync drq
) is
  channel DRQClientNo : ClientNo
  channel TECommand : array 2 of Word \textbf{--srcAddr, dstAddr}
  sync TEAck
begin
  DMAArb (drq, DRQClientNo) ||
```

```
    DMAControl (mta, mtd, DRQClientNo, TECommand, TEAck, IRQ) ||
    DMATransferEngine (TECommand, TEAck, mia)
end
```

通过向 irq 端口写入 0 或 1 来标记中断。外部锁存器必须锁住这个中断值以产生无屏蔽中断（bare interrupt）信号。

12.4.1　仲裁树

来自外设的 DMA 请求到达 sync 通道 drq 上，这些通道连接至请求仲裁器 DMAArb。在程序 ArbFunnel 的顶层以参数化形式对 DMAArb 进行程序声明，程序 ArbFunnel 在 11.2.4 节曾做过介绍。8 输入 ArbFunnel 的结构如图 12.3 所示。

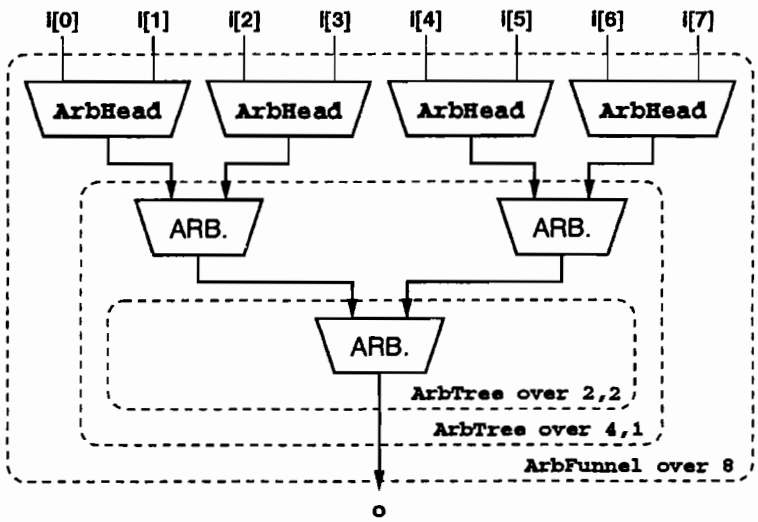


图 12.3　8 输入仲裁器—— ArbFunnel

12.4.2　传输引擎

和仲裁器一样，传输引擎也很简单。它只是控制单元和 MARBLE 发起端接口之间的缓冲器级。它的功能以顺序的 Balsa 语句进行描述，锁存器用来存放输出地址。

```
procedure DMATransferEngine (
  input command : array 2 of Word;
  sync ack;
  output busCommand : MARBLECommandNoData
) is
  variable commandV : array 2 of Word
```

```
begin
  loop
    command -> commandV;
    busCommand <- {commandV[0], read, word};
    busCommand <- {commandV[1], write, word};
    sync ack
  end
end
```

12.4.3 控制单元

这个控制器的主要部分是控制单元。它包含了所有通道的寄存器锁存位和寄存器访问的多路选择器/多路分配器。通过降低通道数并且使用单一通道类型，可以使这种结构更加实用。程序设计者可访问的状态一共有445位。控制单元的端口、局部变量和局部通道的Balsa描述如下。

```
procedure DMAControl (
  input busCommand : MARBLE8bACommand;
  output busResponse : MARBLEResponse;
  input DRQ : ClientNo;
  output TECommand : array 2 of Word;
  sync TEAck;
  output IRQ : bit
) is
  -- combined channel registers
  variable channelRegisters :
    array NoOfChannels of ChannelRegister
  variable channelR, channelW : ChannelRegister
    array over ChannelRegType of bit
  variable channelNo : ChannelNo
  variable clientNo : ClientNo

  variable TEBusy : bit

  variable gEnable : bit
  variable chanStatus : array NoOfChannels of bit
  variable IRQMask, IRQReq : array NoOfChannels of bit

  variable requestPending :
    array NoOfChannels of RequestPair
```

```
channel commandSourceC : DMACommandSource
channel busCommandC : MARBLE8bACommand
channel DRQC : ClientNo
variable commandSource : DMACommandSource
. . .
```

通道寄存器（ChannelRegister）将一个通道的源、目的、计数和控制寄存器结合在一起。通过读写这个 108 位长的寄存器（32 + 32 + 32 + 12）实现变量 ChannelRegisters 的访问。channelR 和 channelW 这两个寄存器作为通道寄存器的读写缓冲器。当 CPU 访问独立的 32 位字时，需要进行的局部写操作涉及这两个寄存器而不是全部的通道寄存器。变量 channelNo 和 clientNo 在操作中用于保存通道和客户端编号。DMA 请求信号的到达和请求标志可以修改 clientNo，对通道寄存器的访问和选择传输可以修改 channelNo。

3 个通道声明用于 DMAControl 和 RequestHandler 子程序之间的通信，其中由控制单元做出仲裁的请求信号来自于仲裁树、MARBLE 目标接口及控制单元的传输引擎应答。RequestHandler 的描述很乏味，所以不在此讨论。

控制单元的程序体如下，其中删减了一些意义不大的部分。

```
begin
  Init ();
  -- RequestHandler is an ArbFunnel
  --   with accompanying data
  RequestHandler (busCommand, DRQ, TEAck, commandSourceC,
    busCommandC, DRQC) ||
  loop
    -- find source of service requests
    commandSourceC -> commandSource;
    case commandSource of
      DRQ then DRQC -> clientNo; MarkUpClientRequest ()
    | bus then
      select busCommandC then
        if (busCommandC.a as RegAddrType).globalNchannel
        then . . . -- global R/W from the CPU
        else -- channel regs
          channelNo :=
            (busCommandC.a as ChannelRegAddr).channelNo;
          ReadChannelRegisters ();
          case busCommandC.rNw of
            . . . -- most of CPU reg. access code omitted
            -- CPU ctrl register write
```

```
            | ctrl then channelW.ctrl :=
              (busCommandC.d as ControlRegister) ||
              requestsPending[channelNo] := {0, 0} ||
              ClearChanStatus ()
            end;
            WriteChannelRegisters ()
          end
        end
      end
      else -- TEAck
        TEBusy := 0;
        if gEnable then AssessInterrupts () end
      end;
      if gEnable and not TEBusy then
        TryToIssueTransfer ()
      end
    end
  end
```

这个控制单元程序体中运用了几次程序调用，例如AssessInterrupts()。这些程序调用是针对共享程序的，这些共享程序的定义与DMAControl描述中的局部变量相一致。在Balsa中，被声明为“共享的”局部程序只在包含该程序的握手电路的某处例化（一般的程序调用，每次调用时都会放置该程序体的复制体）。对共享程序的调用相当于调用元件，只作一次例化，这样它们使用时比普通的程序调用更节约空间，因为普通的程序调用在每个调用位置上都会放置一个与调用程序体相同的复制体。

DMA请求处理——MarkUpClientRequest

上文曾提到，引入的DMA请求信号在请求等待寄存器中被标记。程序MarkUpClient-Request通过clientNo（引入请求的客户端ID）并行检查所有通道的srcClientNo和dstClientNo 控制位来完成这一操作。MarkUpClientRequest的描述如下。

```
shared MarkUpClientRequest is
 begin
   for || i in 0..NoOfChannels-1 then
     if channelRegisters[i].ctrl.srcClientNo = clientNo
       then requestsPending[i].src := 1
     end ||
     if channelRegisters[i].ctrl.dstClientNo = clientNo
       then requestsPending[i].dst := 1
```

```
    end
  end
end
```

该程序中的“for ||”循环实现了 NoOfChannels 个 if 语句的并行结构例化。

寄存器访问——ReadChannelRegister，WriteChannelRegisters

用于访问通道寄存器的共享程序很短。它们通过唯一的变量索引来访问通道寄存器。这两个程序为：

```
shared ReadChannelRegisters is begin
  channelR := channelRegisters[channelNo]
end

shared WriteChannelRegisters is begin
  channelRegisters[channelNo] := channelW
end
```

由于没有提供单字的写使能，修改单字时，整个通道寄存器必须先被读出，然后修改，最后再写回。在 CPU 访问通道寄存器的描述中，使用这样的更新方法：ReadChannelRegisters 后面跟着 channelW := channelR。

通道状态和中断——ClearChanStatus, AssessInterrupts

中断输出位是通过 AssessInterrupts 声明的。当 IRQReq 寄存器非零时，产生中断信号，并且每当有清除中断的行为发生时，中断将被重新计算。在通道控制寄存器变为清除中断和通道状态（运行已结束）指示时，调用 ClearChanStatus。它们的描述为：

```
shared AssessInterrupts is begin
  IRQ <- (IRQReq as NoOfChannels bits) /= 0
end

shared ClearChanStatus is begin
  chanStatus[channelNo] := 0 ||
  IRQReq[channelNo] := 0;
  AssessInterrupts ()
end
```

准备传输选择——TryToIssueTransfer, IssueTransfer

只要 DMA 控制器被它的指令接口激活，它就试图执行一次传输。轮流检查每条通道的请求等待寄存器和 ctrl[n].enable 位来确定这条通道是否准备好传输。对于通道的检查是通

过递增通道号的顺序完成的，使用一条局部通道可以并行的从两个位置同时进行通道号递增检查。TryToIssueTransfer 的 Balsa 描述如下：

```
shared TryToIssueTransfer is local
  variable foundChannel : bit
  variable newChannelNo : ChannelNo
begin
  foundChannel := 0 || channelNo := 0;

  while not foundChannel then
    -- source and destination requests arrived?
    if requestsPending[channelNo] = {1, 1}
      and channelRegisters[channelNo].ctrl.enable then
      ReadChannelRegisters ();
      requestsPending[channelNo] :=
        channelR.ctrl.srcDRQ, channelR.ctrl.dstDRQ ||
      foundChannel := 1 ||
      IssueTransfer () ||
      UpdateRegistersAfterTransfer ()
    else
      local
        channel incChanNo: array ChannelNoLen + 1 of bit
      begin
        incChanNo <- (channelNo + 1 as
          array ChannelNoLen + 1 of bit) ||
        select incChanNo then
          foundChannel := incChanNo[ChannelNoLen] ||
          newChannelNo := (incChanNo[0..ChannelNoLen-1]
            as ChannelNo)
        end;
        channelNo := newChannelNo
      end
    end
  end
end
```

注意，如果执行了一次传输，这条通道的 requestPending 位将被该通道的 ctrl.{srcDRQ,dstDRQ} 控制位重新初始化。程序IssueTransfer把这次传输的内容传送给了传输引擎，它的定义为：

```
shared IssueTransfer is begin
  TEBusy := 1 ||
  TECommand <- {channelR.src, channelR.dst}
end
```

执行传输前通过检查TEBusy形成互锁，并且在传输开始时对TEBusy置位，传输完成后对TEBusy复位，这样就确保了在传输引擎被占用时不会出现向传输引擎的传输（死锁控制单元）的情况。传回控制单元的TEAck也提供了重新触发DMA控制器的激励，来实现未完成的请求。这次重新触发与通道的顺序选择相结合，使未完成的请求（当传输引擎忙时接收到的请求）能够正确地执行。需要注意的是，顺序通道选择使先前到达的请求形成一个静态优先队列。

寄存器递增、递减——UpdateRegistersAfterTransfer

在一次传输发生后，这次传输的通道的寄存器必须更新。用程序UpdateRegisters-AfterTransfer来执行这项任务：

```
shared UpdateRegistersAfterTransfer is begin
  channelW.ctrl := channelR.ctrl ||
  if channelR.ctrl.srcInc then
    channelW.src := (channelR.src + 1 as Word)
  end ||
  if channelR.ctrl.dstInc then
    channelW.dst := (channelR.dst + 1 as Word)
  end ||
  if channelR.ctrl.countDec then
    channelW.count := (channelR.count - 1 as Word)
  end;
  if channelW.count = 0 then
    chanStatus[channelNo] := 1 ||
    if IRQMask[channelNo] then
      IRQReq[channelNo] := 1
    end ||
    channelW.ctrl.enable := 0
  end;
  WriteChannelRegisters ()
end
```

该程序使用两个增量和一个减量分别修改这个通道的源地址、目的地址和计数器。如果通道的传输计数器减为0，则chanStatus位、中断状态和通道使能位都要进行更新，来标志一次传输的结束。

到这里为止，已经完成了对Balsa描述的DMA控制器的细节介绍，如要进一步详细了解请参见文献[8]，如本章开头提到的，完整代码清单可从文献[7]所给的网址中获得。

第三部分　大规模异步电路设计

摘要：在本书的最后部分，我们给出一些大规模异步VLSI设计的实例来说明异步技术的能力。前面的两个设计——由飞利浦公司开发的非接触智能卡和由曼彻斯特大学开发的Viterbi解码器——均为“低功耗设计起步”中的欧盟基金项目，也是本系列丛书的发起者。第15章介绍有关Amulet微处理器系列，它也是由曼彻斯特大学开发的，其他的几个EU-基金项目虽然不包含在低功率设计起步之中，但也以低功率为重要目标。本书这一部分介绍的芯片曾经是当时开发的最大和最复杂的异步电路设计，这些芯片所涉及的全部技术细节的描述远远超出了本书的范围，但异步设计示例中所包含的技术是完全能够支持大规模设计的，它们展示了熟练而有经验的团队能够用异步技术做什么。这里所描写的是一个从感性到理性的过程，一个高级的异步系统设计者应当去经历这种设计开发过程。

关键词：异步电路，大规模设计

第13章 DESCALE：应用于低功耗智能卡设计实验①

Joep Kessels，Ad Peeters
Philips Research, NL-5656AA Eindhoven, The Netherlands
Joep.Kessels@philips.com; Ad.Peeters@philips.com

Torsten Kramer
Kramer-Consulting, D-21079 Hamburg, Germany
Kramer@kramer-consulting.de

Volker Timm
Philips Semiconductors, D-22529 Hamburg, Germany
Volker.Timm@philips.com

摘要：设计了一种基于异步芯片的非接触式智能卡。异步电路具有平均功耗低及电流峰值小两个特性，非常适合于非接触式装置。它可以自动地调整电路速度使其可以在较大的电源电压范围内正常运行。这一特性使得所设计的电路可以适应电源电压变化很大的情况，在给定的功率下仍可获得最高性能。该异步电路已完成了构建、测试和评估。

关键词：低功率异步电路，智能卡，非接触式装置，DES加密系统

13.1 引言

自从智能卡在20世纪80年代被发明之后，它的应用领域就持续不断地扩大，如银行业、通信业（电话及SIM卡）、接入控制（付费电视）、保健、公交卡、电子签名以及身份识别等。目前，大多数卡都是接触式的，因此需要将其插入读卡器。在某些应用场合，快速处理是十分重要的，而非接触式智能卡则只需接近（几厘米的距离）读卡器便可以完成事务的处理。这种智能卡中的芯片必须具有相当高的电能使用效率，因为它是由电磁辐射进行供电的。

① 基金资助情况：_Part of the work described in this paper was funded by the European Commission under Esprit TCS/ESDLPD. contract 25519 (Design Experiment DESCALE)。

异步CMOS电路只在工作状态转换时有能量损耗，因此在低功耗方面具有相当大的潜力。但遗憾的是，异步电路在门级和寄存器传输级设计非常困难。为此，定义了一种被称为Tangram的高级设计语言[141]，并由“硅编译器”将Tangram程序编译成异步电路。

Tangram编译器产生一些特定类型的异步电路，这种电路被称为“握手电路”[135, 112]。握手电路是通过从一个基本元件库中调用元件来构建的，这个元件库大约40个元件，这些元件用来产生通信过程中的握手信号。

利用Tangram已经成功地设计出了一些芯片[136, 144]，如果我们把这些芯片与现有的同步电路芯片相比较，异步芯片在面积上要大20%左右，而功耗只是同步芯片的25%。

为了发掘用异步电路设计非接触智能卡的优点，我们设计出了异步智能卡电路芯片。本章指出了利用异步电路的哪些性质有了相应的结果。本章其他部分的内容安排如下：在13.2节中，介绍了使用Tangram设计异步电路的方法；13.3节对异步电路与同步电路在功耗方面的差别做了概述；13.4节首先介绍了非接触式智能卡的背景，然后比较了非接触式装置与电池供电装置在耗电特性方面的区别，最后指出了异步电路非常适合设计非接触式装置的原因；13.5节介绍了数字电路；13.6节给出了硅片的版图；13.8节介绍了功率调节器，它也是用异步电路实现的；本章最后，总结了异步设计的所有优势。

13.2　VLSI中异步电路的程序设计

本文所述的智能卡IC设计流程基于Tangram程序语言及相应的编译器和工具集。这种设计方法中十分重要的方面是硅编译器的透明度[142]。硅编译器的透明度高，则允许设计者在程序级（Tangram）时即可推断出诸如面积、功率、性能及可测试性等指标。

本节首先介绍Tangram工具集，然后简要描述基本的握手技术，最后使用第8章所介绍的GCD算法说明大规模集成电路（VLSI）的程序设计技术。

13.2.1　Tangram工具集

图13.1描述了Tangram工具集。Tangram是一种类似于C语言和Pascal语言的传统编程语言，扩展了包括类似CSP语言中表示并发及通信方式的结构[58]。另外，Tangram还包括了表示特殊硬件问题的语言结构，如块的共享和时钟边沿的等待。

编译器将Tangram程序编译成“握手电路”——即网表，构成网表的元件来自于由40个不同握手元件组成的元件库。每一个握手元件实现一种语言结构，例如顺序、循环、通信及共享。

握手电路仿真器及相应的性能分析器会给设计者反馈有关的电路信息，如功能、面积、时序、功率及可测试性等。

将设计映射成传统的（同步）标准元件库一般需要经历两个步骤：首先，元件扩张器（expander）使用元件库产生一个抽象的网表，该网表由组合逻辑、寄存器和类似Muller C元

的异步元件构成。在这一步骤中同时确定数据编码及握手协议，通常是4相单轨实现。然后，使用商业化的综合工具及映射工具产生标准元件网表。在这一映射过程中并不需要专门的（异步）元件，因为所有的异步元件都分解成标准单元库中的元件。

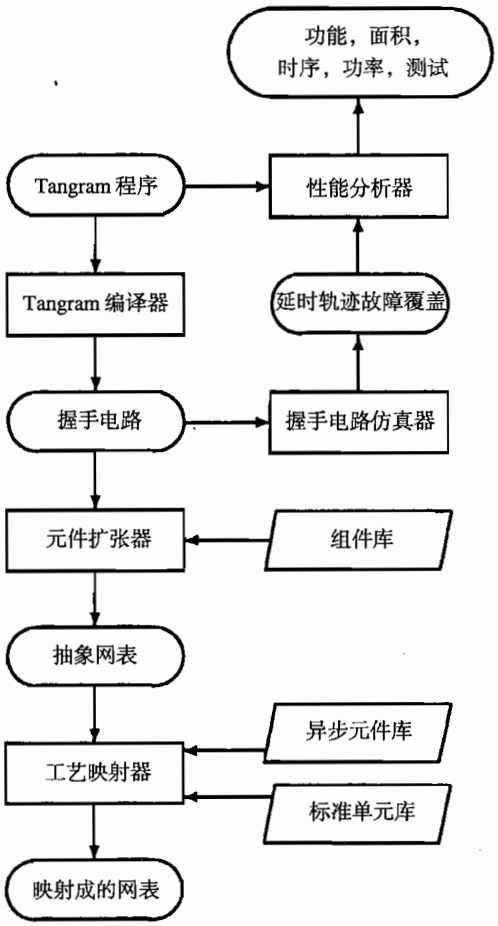


图 13.1 Tangram 工具集：方框表示工具，椭圆表示（设计）描述

与此相似的基于语言并使用握手电路作为中间媒介的方法在文献[17, 9]中都有介绍。开放底层细节的异步电路设计方法同样也被证明是成功的[80, 21, 83, 30, 64]。有关异步电路设计方法的一般综述可参见文献[69]及本书的第一部分。

13.2.2 握手技术

大规模异步集成电路的设计，要求使用时序规则来替代传统VLSI设计中使用的时钟规则。我们已选择握手信令[121]作为异步时序规则，因为它支持组件到系统的即插即用，且在VLSI中

易于实现。替代握手技术的一种技术是合成异步状态机，它使用基本模式或突发模式假设来进行通信[27, 132]。

图13.2描述了一个在主动方及被动方之间进行点对点通信的握手通道。在抽象图中，实心圆代表通道中的主动方，空心圆代表被动方。而从下面的实现框图中可以看出，主动方与被动方都是通过两根线相连接：一条是请求线（Req），另一条是应答线（Ack）。握手操作需要主、被动双方配合才可以实现。由主动方完成初始化工作，首先通过Req线向被动方发送信号，然后便等待通过Ack线返回的信号。被动方则等待请求信号到来后发送应答信号。握手通道不仅可以用来实现同步，还可以用于通信。由此，数据可以在请求、应答或者两者之间进行编码。

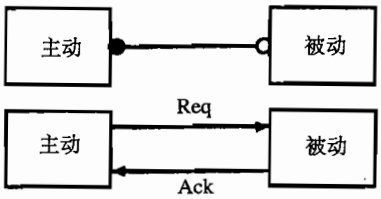


图 13.2　握手通道：抽象图（上图）与实现（下图）

大多数的异步 VLSI 电路中采用的协议是Req及Ack在通道初始状态时都为低电平的4相握手协议。主动方在开始通信时将Req置高，当被动方接收到Req的高电平后将Ack信号置为高电平。其后跟着一个归零周期，先将Req回归到低电平，再将Ack回归到低电平，从而回到初始状态。

握手元件通过握手通道来达到信号传递的目的。我们可以利用语言结构来构造握手元件，图 13.3 所示为顺序元件（sequencer）和并行（parallel）元件的例子。

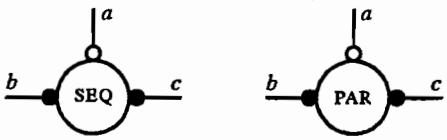


图 13.3　握手元件：顺序（左图）和并行（右图）

顺序元件（SEQ）被来自 a 通道的信号激活之后，首先通过 b，然后通过 c 进行握手操作。它常用来控制连接到 b 和 c 上命令的顺序执行。具体地说，就是当SEQ由 a 接到请求信号之后，通过 b 发出一个请求信号，并等待相应的应答信号，应答信号到达后，再经 c 发送请求信号，并等待由 c 传来的应答信号，最后向通道 a 发送一个应答信号来结束操作。

并行元件（PAR）被来自 a 通道的请求信号激活后，同时向 b 通道和 c 通道发送请求信号，待它们的应答信号都到达后，通过向 a 通道发送应答信号来结束操作。

除此之外，也可以构造数据（变量）存储元件和数据运算（如加法、位操作等）元件。Tangram程序可通过面向语法的方式（见第 8 章）编译成握手电路。例如，Tangram 中的 while 循环可以表达为：

```
do 判断语句 then 语句 od
```

这个语句所表达的握手电路的功能如图13.4所示。“do单元”被激活后，通过主动数据端口（左侧）的一次握手获得判断语句的值，如果值为“假”，则通过它的被动端口完成握手，否则通过其主动控制端口（右侧）的一次握手激活执行语句，执行完成之后，开始新一轮的循环，重新对判断语句的值进行判断。

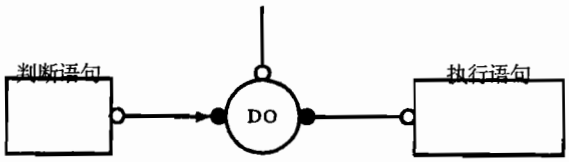


图 13.4 while循环的握手电路

有关握手电路、从Tangram到中间结构的编译、4相握手协议及握手单元的门级实现可参见文献[141, 135, 112]。

13.2.3 GCD算法

在使用Tangram进行设计时，功能的正确性通常是最初设计需要强调的，但如果只考虑这一点，可能会导致设计面积过大、速度过慢及功耗过高。可以通过变通的方法来实现合理的数据路径、控制结构、某些特定目标的优化标准或功耗函数等。Tangram工具集可以用来对设计进行评估和分析，由此也可以判定哪种设计方式更适合。硅编译器的透明度（所编即所得）有助于预知这些设计方法的效果。

GCD算法被用来作为例子说明基于透明编译器的VLSI程序设计（也可参见8.3.3节中的讨论），从图13.5中的算法可知，功能是正确的，但在实现VLSI时却耗费了大量的资源，远远没有达到优化的效果。相应的握手电路如图13.6所示。

```
  int = type [0..255]
& gcd_gc : main proc (in?chan <<int,int>> & out!chan int).
  begin x,y:var int ff
  | forever do
      in?<<x,y>>
    ; do x<>y then
        if x<y then y:=y-x
               else x:=x-y
        fi
      od
    ; out!x
    od
  end
```

图 13.5 GCD算法的Tangram程序

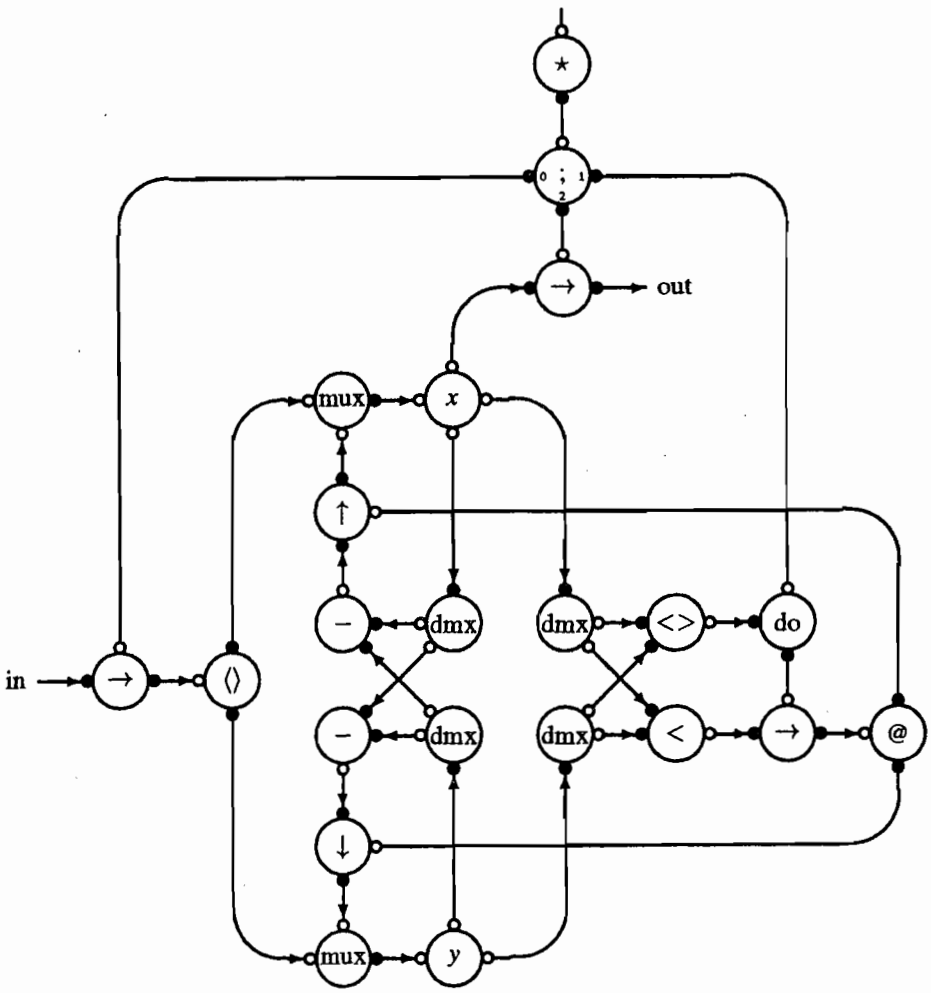


图 13.6　对最初的 GCD 程序编译后的握手电路

Tangram 工具集给出了该 GCD 设计占用资源情况的报告，如下所示。

celltype	#occ	area	gate eq. [eqv]	(%)
Delay Matchers	19	2052	38.0	12.5
Memory	16	3744	69.3	22.8
C-elements	12	1242	23.0	7.5
Logic	81	9414	174.3	57.2
Total:	128	16452	304.7	100.0

该设计的一个重要方面是在整个数据通路中，采用了 4 个运算符。为了达到优化的目的，我们可以通过改变 Tangram 语句结构来减少运算符的个数，将两次减法运算改为一次。为此要

改变算法的时序行为，并通过 x 与 y 两数的互换或相减来多次使用 do-loop 循环。这需要把 x 和 y 作为变量存入到一个单独的触发器中。新的 Tangram 算法如图 13.7 所示，获得的门级报告表明，新的 GCD 程序将等效门的数量从 305 个减少到了 274 个。

celltype	#occ	area	gate eq. [eqv]	(%)
Delay Matchers	14	1512	28.0	10.2
Memory	16	3744	69.3	25.3
C-elements	10	1008	18.7	6.8
Logic	86	8532	158.0	57.7
Total:	126	14796	274.0	100.0

```
int = type [0..255]
& gcd_gc : main proc (in?chan <<int,int>> & out!chan int).
  begin
    xy : var <<int,int>> ff
  & x = alias xy.0
  & y = alias xy.1
  | forever do
      in?xy
    ; do x<>y then
        if x<y then xy:= <<x,y-x>>
               else xy:= <<y,x>>
        fi
      od
    ; out!x
    od
  end
```

图 13.7 具有优化控制的 GCD 算法的 Tangram 程序

另一种简化方法是将 x<y 和 y-x 这两个运算合用一个运算符来实现，并将对变量 x、y 赋值的两个语句变为一个赋值语句，以便进一步简化控制。不过，这样会使条件表达式总是需要最长的计算时间，即使只将 x 和 y 的位置对调，也会这样。另外，如图 13.8 所示，在计算过程中，可以通过一个附加的步骤，将 loop 语句的结束条件由 x<>y 变为 y<>0。这种程序的握手电路如图 13.9 所示，相对于元件分开来说，数据通道的操作已被抽象的方式所替代。

由此可见，经过优化后的握手电路比最初的设计具有更简单的结构。逻辑块的数量（在单轨电路的实现中，延时匹配门链的数目）已经从 4 个减少到 2 个。这种改善在下面所给出的面积统计中也是显而易见的。

celltype	#occ	area	gate eq. [eqv]	(%)
Delay Matchers	10	1080	20.0	9.4
Memory	16	3744	69.3	32.7
C-elements	6	576	10.7	5.0

Logic	42	6048	112.0	52.8
Total:	74	11448	212.0	100.0

从优化面积的角度出发，新的算法在控制（C单元的使用数目由12个减少到6个）、逻辑（组合逻辑由174个减少到112个等效门单元）及总延迟单元（由19个减少到10个）的数目等方面所耗资源都有所改善。

```
  int = type [0..255]
& gcd_gc : main proc (in?chan <<int,int>> & out!chan int).
  begin
    xy : var <<int,int>> ff
  & x = alias xy.0
  & y = alias xy.1
  & comp: func(). (y-x) cast <<int,bool>>
  | forever do
      in?xy
    ; do y<>0 then
        xy:= if -comp.1 then <<x,comp.0>> else <<y,x>> fi
      od
    ; out!x
    od
  end
```

图 13.8　有优化数据通道的GCD算法的Tangram程序

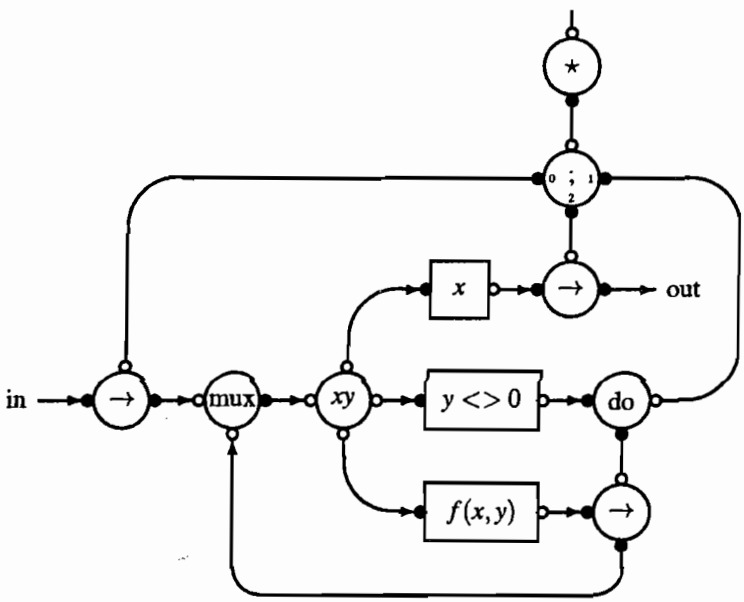


图 13.9　优化GCD程序经编译后的握手电路

13.3 异步电路的机遇

如果把Tangram编译器所编译出来的异步电路与相应的同步电路比较，会发现它们有三点区别，并由此可以发现异步电路的4个引人注目的优点。

1. 同步电路中的子电路是由时钟驱动的，而在异步电路中则由需求驱动。也就是说，异步电路的子电路只有在需要用到它的场合和时间才会处于活动状态。由此，异步电路的功耗要远小于同步电路。
2. 同步电路通过全局（主要的）时钟进行同步，而异步电路通过分布式的握手实现同步。
   a) 同步电路在时钟边沿会产生大的电流尖峰，而异步电路的功耗随时间分布得更均匀。
   b) 同步电路中电流峰值的严格周期性会导致在辐射频谱中出现较高的时钟谐波，而这一点在异步电路设计中不存在。
3. 同步电路以外部时钟为参考，异步电路是自同步的。这就意味着，异步电路工作在更宽的电源电压范围（例如从1 V到3.3 V）并自动地调整它们的运行速度。这种特性被称为“自动的性能适应”（automatic performance adoption），意思是说异步电路对电源电压变化的反应是自适应的。也可以通过电压调节使电源满足其性能的需要，从而减小功耗[100]。同步电路中也可以采用自适应电压调节技术，但是必须采取某些相应的方法来调整时钟的频率。

当然，异步电路也是有缺点的。最大的一个缺点就是这种电路是非传统的：设计者、主流设计工具及设计库都是面向同步设计方法的。另外一个缺点是异步电路使用门电路来控制寄存器（锁存器及触发器），而不是同步电路中直接的时钟分布网络。虽然这样减小了功耗，可是会导致电路更大，速度更慢并难于测试。可测性差可能是这些缺点中最基本的一个问题。对于异步电路测试的讨论，参见文献[61, 120]。

飞利浦半导体公司设计了一系列的异步寻呼机芯片的主要原因，就是上面提及的异步电路特性中的2.b[114]。然而，正如将在13.4节中强调的，在非接触式智能卡领域中，其他三个特性会有更好的表现。

13.4 非接触式智能卡

非接触式智能卡相对于接触式智能卡具有以下优点：使用快速而便捷，不怕灰尘及油脂，读卡器不需设置插口，不易折断。

非接触式智能卡与读卡器之间的信号传递是通过读卡器所发射的电磁波来实现的。卡片上的线圈用来从电磁场中吸收能量。智能卡所能吸收的能量取决于它与读卡器间的距离、线圈的匝数及卡在电磁场中的位置。

图 13.10是非接触式智能卡的功能模块，包括一片VLSI电路（虚线内部）及外部线圈。调谐电路由线圈及电容 C_0 组成，其功能是：

- 吸收能量
- 接收时钟频率（与载波频率相同）
- 支持通信

完整的电路还包括电源单元和带有用于储存能量的缓冲电容（C_1）的数字电路。

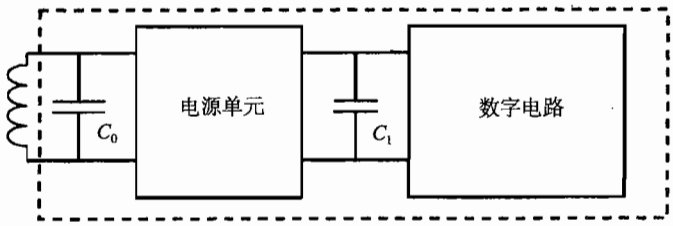


图 13.10　非接触式智能卡

目前已有的几个非接触式智能卡标准是：

- ISO/IEC 10536，具体指明了“贴近型”（close coupling）智能卡的操作距离与读卡器的间距为 1 cm。
- ISO/IEC 14443，定义了“接近型集成电路卡”（Proximity Integrated Circuit Card, PICC），操作范围为距离读卡器10 cm，卡上线圈基本上为5圈。这个标准分为A和B两种类型，主要性能如表 13.1 所示。
- ISO/IEC 15693 定义了“邻近型集成电路卡”（Vicinity Integrated Circuit Card, VICC），操作范围可以距离读卡器 50 cm 以内，卡上线圈需要几百圈。

表 13.1　ISO/IEC 14443 标准的特性

ISO 14443 标准	A (Mifare)	B
载波频率	13.56 MHz	13.56 MHz
吞吐量（由高到低）	106 Kb/sec	106 Kb/sec
向下连接（由读卡器向卡）	ASK 100%	ASK 10%
编码	Miller	NRZ
向上连接（由卡向读卡器）	ASK	BPSK
频率	847.5 kHz	847.5 kHz
调制	Manchester	NRZ

Mifare[63]标准卡（ISO/IEC 14443 A 型）目前应用极为广泛。图 13.11 展示了一张包括芯片及线圈的Mifare卡。Mifare是支持两路信息交换的接近型智能卡（使用时与读卡器的距离最远可达 10 cm）。由于信息交换时间一定小于 200 ms，所以其性能是十分重要的。最早大规模

使用Mifare技术的是“汉城公交联合会”，使用了上百万张这样的公交卡，每个月产生几百万次交易。

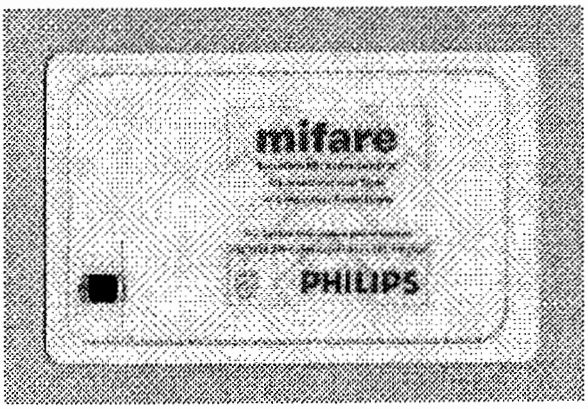


图 13.11 Mifare卡，IC（左下）与线圈

本章所提到的异步Mifare智能卡集成电路在文献[73]中已有介绍。ISO/IEC 14443 B型智能卡标准中所用到的同步[116]及异步[2]电路也在以前的文献中介绍过。由于标准中的A型卡100%采用的是ASK调制方式，Mifare IC在诞生时没有任何内部电源，而B型卡采用ASK调制方式的占10%。

由于通常情况下，非接触智能卡芯片在一次操作的整个时间内平均只能接收到几毫瓦的功率，所以能量的利用率是非常重要的。虽然在电池供电的设备中低功耗也很重要，但这两类设备关键的差别有两点。

1. 为了使电池的供电时间达到最大化，通常应该使平均功耗达到最小。而对于非接触式设备，除此之外还要减少峰值功耗，因此，必须保持峰值低于某一个特定值。这个值取决于输入功率和缓冲电容。
2. 电池供电设备所用的供电电压基本上是不变的，而非接触卡的供电电压在传输数据的过程中会随输入及消耗功率的波动而变化。

对于传统的同步芯片与异步智能卡，我们突出以下几点事实。如稍后看到的那样，使用异步电路将提供改进的机会。

- 尽管输入和有效功耗随时间变化，但同步电路运行速度由时钟确定，是固定不变的，所以同步电路必须在最小的电源电压输入情况下也能保证最大功率需求的作业正常工作。因此，如果接收到较多的电能，那么多余部分的能量将被分流掉；反之，如果接收到的能量太少，供电电压的下降会使电路的执行速度变慢，一旦电路变得太慢而不能满足时

钟所设定的时序要求，传输将被终止。对这一点，非接触式智能卡芯片中含有阈值电压测量电路，用以判断电压的下降到何种程度时取消传输。目前，在非接触智能卡中，微控制器的性能并不受限于电路的速度而是受限于接收到的射频功率。

- 同步电路需要一个容量为几个纳法的缓冲电容，它所占的面积与微控制器的面积处于同一数量级。
- 智能卡与读卡器之间的通信是基于负载调制来实现的，因此负载本身的正常波动可能会对通信产生干扰。

13.5　数字电路

图 13.12 所示的数字电路包括以下几个部分：

- 80C51 微控制器。
- 三种不同类型的低功耗存储器，容量和存取时间如表 13.2 所示（64 个字节可以一次同时写入 EEPROM）。
- 两个加密协处理器
  - 用于公共密钥转换的 RSA 转换器[119]；
  - 用于私有密钥转换的 DES 转换器[96]。
- 用于外部通信的通用异步收发器（UART）。

EEPROM 中存储了相应的程序、密钥和电子货币。ROM 和 RAM 都配有“匹配延迟线”，在 EEPROM 中，我们也设计了一个具有类似功能的计数器。匹配延迟线的作用是为 3 个存储器提供握手接口，即使在温度和供电电压等有所改变的情况下，也可保障存储器在存取时间内可以正常通信。另外一个优点是，当代码在 ROM 中执行时，相对于 EEPROM 而言，控制器的运行速度会自动提高。

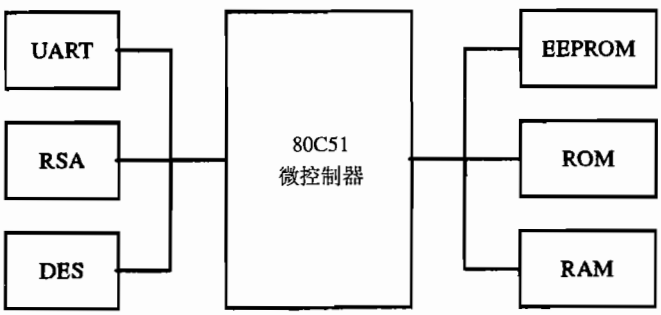


图 13.12　智能卡电路的整体设计

表 13.2 存储器容量和存取时间

存储器	容量	存取时间（ns）	
类型	字节（kB）	读	写
RAM	2	10	10
ROM	38	30	
EEPROM	32	180	4000

智能卡电路被设计成两种接口形式，一张卡同时具有接触式和非接触式的接口。除了应用于接触式操作的 RSA 转换器之外，所有的电路都是异步的。在非接触式操作中，供电电压为 2 V，而在仿真中则需要使用仿真库所要求的 3.3 V 的电压来仿真它的特性。

13.5.1 80C51 微控制器

这里所介绍的 80C51 微控制器是文献[144, 143] 中所提到的控制器的修改版，主要的 4 点修改如下。

为了与慢速存储设备通信，80C51 结构中包括了“预取单元”（prefetch unit）。在 3.3 V 供电及自由运行模式下，平均一条指令的执行时间为 100 ns。相对来说，从内存（ROM /RAM）中存取代码的时间可以忽略不记。但是，如果代码存储在 EEPROM 中，那么微控制器不得不在访问过程中处于等待状态，由于大多数指令长度是一个或两个字节，读取时需要 180 ns 或者 360 ns，因而使系统的性能大大降低。为了避免性能的下降，在微控制器中增加了 2 个字节的 FIFO，也称为指令预取（instruction prefetching）形式，实现对指令的预取，在未满的情况下，它将和微控制器并行操作。预取单元将性能提升了 30%。一个简化的预取单元将在 13.5.2 节中给出。

另外，新的 80C51 中引入了“预写完成”（early write completion）功能，实质是当微控制器发出一个写操作后，不等待操作完成而继续执行其他的指令。这样避免了在对 EEPROM 进行写操作时微控制器不得不等待接近 4 ms 的情形（如更改电子货币的数量时）。同时，这种技术也加快了对 RAM 的写入速度。为了实现这种功能，相应的代码必须存储在 ROM 中。

控制器增加了用于“迅速挂起”（immediate halt）的输入信号，通过这一信号，可以使控制器立即挂起一小段时间。作用是抑制读卡器与智能卡之间的通信，提供 3 μs 的时间用于对已经收到的信息进行编码。由于在这段时间内，智能卡不接收任何能量，控制器必须立即挂起（只运行一些基本的功能）。在同步设计中，由于时钟将在这些周期中停止，所以挂起功能会自然发生。

准同步（quasi synchronous）模式：在指令级，异步微控制器与对应的同步微控制器的时序可以完全兼容。在这种模式下，为了满足时序要求，异步微控制器在执行每一指令后都会等待一段时间，使其与对应的同步微控制器的指令执行周期一致。当软件设计中存在依赖时间功能时，就有必要用到这种模式。由于这种模式是通过软件进行控制的，微控制器可以很容易地根据其所实现的功能完成模式的转换。这一特点对说明“保障性能”（guaranteed performance）

是非常有用的。所谓“保障性能”是指每条指令，在一定的时钟脉冲数内完成的情况下最高的时钟频率。对于大多说的程序来说，“自由运行性能”（free running performance）会比“保障性能”速度提高两倍。

我们已经将Tangram所编译的异步电路与相同功能的同步电路进行了比较，该同步电路基于VHDL进行编译综合并使用相同的CMOS工艺。结果发现，这两种电路的性能相差无几。而且，异步微控制器很好地表现了我们希望在智能卡芯片中所实现的异步电路的 3 个特性：

- 在同样的性能实现及供电电压情况下，异步 80C51 的平均功耗是同步 80C51 的 1/3。
- 在 3.3 V 供电电压情况下，同步及异步 80C51 的电流峰值如图 13.13 所示，异步 8051 运行于准同步模式，比同步电路的性能高 2.5 倍（同步电路工作在 10 MHz，异步电路工作在 25 MHz）。尽管图中并没有给出两种电路的平均功耗，但是也可以看出，异步电路的电流尖峰是同步电路的 1/5。
- 在“自由运行性能”模式下，当在 ROM 中执行指令时，异步微控制器的性能随着供电电压发生变化的函数曲线如图 13.14 所示。由此可见，性能对电压的依赖基本上是线性的。当电压从 1.5 V 升至 3.3 V 时，性能从 3 MIPS 增加到 8.7 MIPS（将近 3 倍）。由于当电压低于 1.5 V 时，ROM 将无法正常工作，所以无法测量到 1.5 V 以下的性能。不过我们发现，对于不需要存储器的 DES 协处理器，在 0.5 V 供电电压的情况下仍然可以继续工作。

图中也给出了供电电流随电压变化的函数曲线。需要注意的是，电流从 0.7 mA 增加到了 6 mA（大约增长了 9 倍）。由于在 CMOS 电路中电流是电子跃迁率与每次跃迁的电荷量的乘积（两者都与电压成线性关系），因此电流的增长与电压的增长成平方关系。而功耗是电流与电压的乘积，因此功耗相对于电压的增长成三次方关系。

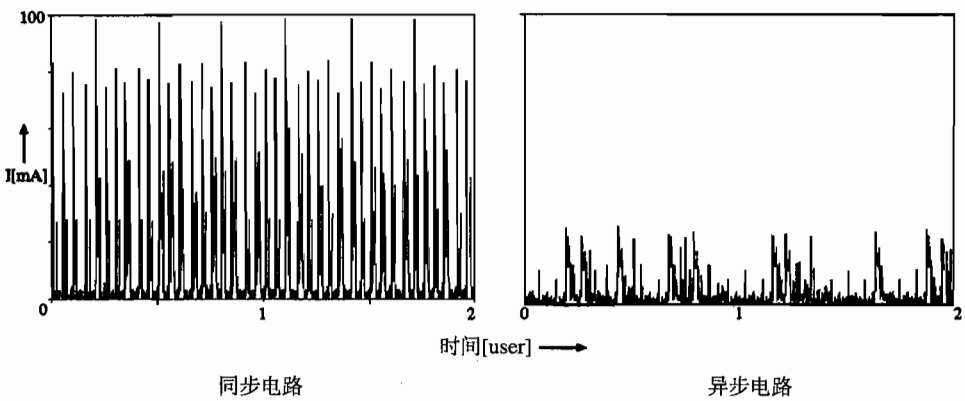


图 13.13　80C51 微控制器电流尖峰

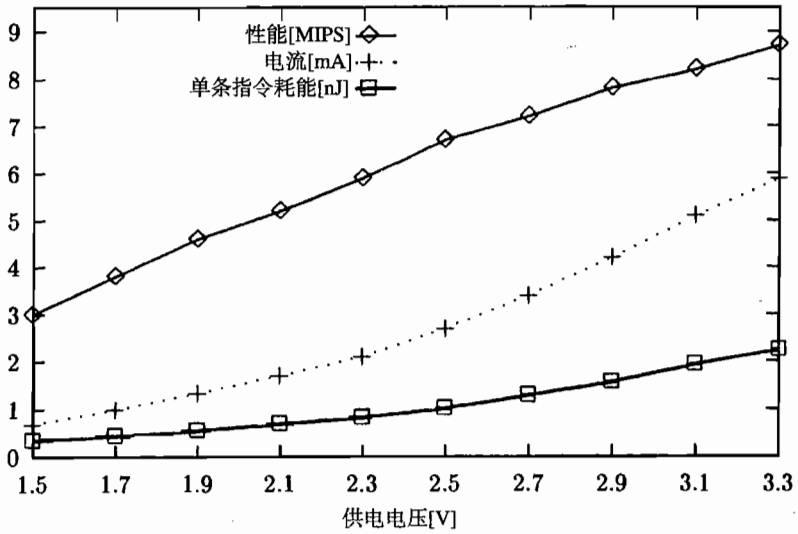


图 13.14 不同供电电压下异步 80C51 性能测量

13.5.2 预取单元

图 13.15 所示为简化了的预取单元的 Tangram 程序。预取单元与 80C51 内核之间通过两个通道连接：通过 StartAddress 通道接收包含预取代码的起始地址；然后将预取代码通过 CodeByte 通道发送出去。由于预取单元在通信过程中一直是被动的，所以它可以通过检测每一个通道内核是否开启通信。预取单元包括：程序计数器（Program Counter）、程序指针（PC）、由 Buffer 阵列构成的两级缓冲器、整数计数器及两个 1 位的指针：getptr 和 putptr。

预取单元将执行无限次循环，在每次循环中都要首先执行一条选择语句（由 sel...les 表示），在这条命令中能够选择由关键词“or”分开的 3 条“哨命令”（guarded command）。哨命令的格式为：

```
哨 then 执行语句,
```

其中，哨为布尔值，当其值为“1”时，then 后面的执行语句开始执行。执行一条选择语句意味着等待，直到 3 条选择语句中至少一条的条件得以满足时，选择该判断语句后面的执行语句执行。

在第一条哨命令中，通过 StartAddress 通道检测内核是否发送了新的开始地址。如果发送了新的开始地址，预取单元中的程序计数器指针将设定为新接收的地址，并刷新缓冲器，取消正在进行的存储器访问（通过将 MemReq 和延迟计数器复位）。哨命令中的 4 个子语句都是并行执行的（“A || B”表示同时执行 A 与 B，而“A ; B”表示 A 与 B 是顺序执行的）。

```
forever do
   sel probe(StartAddress)
       then (  StartAddress?pc
            || putptr := getptr
            || count := 0
            || AbortMemAcc()
            )
    or probe(CodeByte) and (count>0)
       then CodeByte!Buffer[getptr]
          ; (  getptr := next(getptr)
            || count := count-1
            )
    or MemAck
       then Buffer[putptr] := MemData
          ; (  putptr := next(putptr)
            || count := count+1
            || pc := pc+1
            || CompleteMemAcc()
            )
   les
 ; if (count<2) and -MemReq
   then MemReq := true
   fi
od
```

图 13.15　简化了的预取单元的 Tangram 源代码

第二条哨命令的作用是当内核已经准备好接受命令且缓存不为空时，将下一条程序字节通过CodeByte通道转送给内核。第三条哨命令当数据信号的读操作有效，即MemAck变为高电平时执行。这样，从存储器读出的值被放入缓冲器后，存储器的握手过程才结束。

在这些过程完成之后，如果缓冲器还没有满，并且暂时没有存储器进行访问时，将开启下一次的内存访问。由于(count<2) ∨ -MemReq是循环的不变量，因此，循环中最后一条（条件）语句可以简化为无条件赋值语句 MemReq:= (count<2)。

需要注意的是，pc-count的值与内核中程序计数器的值相同，由于它被一条跳转指令设置为目标地址，因此如果向内核传输一个字节，它的值将增加1；如果从内存中读取一个字节，则其值保持不变。因此，内核并不需要保存程序计数器的值，当其他的相关分支程序需要时，它可以从预取单元中恢复计数器的值。显然，这种功能是需要通过扩展图 13.15 中的 Tangram 代码来实现的。

13.5.3　DES 协处理器

一次事务处理可能需要多达 10 次的 DES 变换，如果使用软件来变换的话，每次 DES 计算需要耗费大约 10 ms 的时间，但使用硬件来实现 DES 处理则只需要一半的时间。

DES协处理器的数据路径如图13.16所示。处理器支持1～3次DES变换，对于后者来说，包含两种密钥：公钥（foreground key）及私钥（background key）。单次DES变换只需要公钥，而三次DES变换需要在第一次及第三次变换时使用公钥，在第二次变换时使用私钥。公钥存储于包含56位的触发器CD0中（公钥为56位），私钥存储于包含56位的锁存器CD1中。文本存储于包含64位的锁存器LR中（DES字长为64位）。

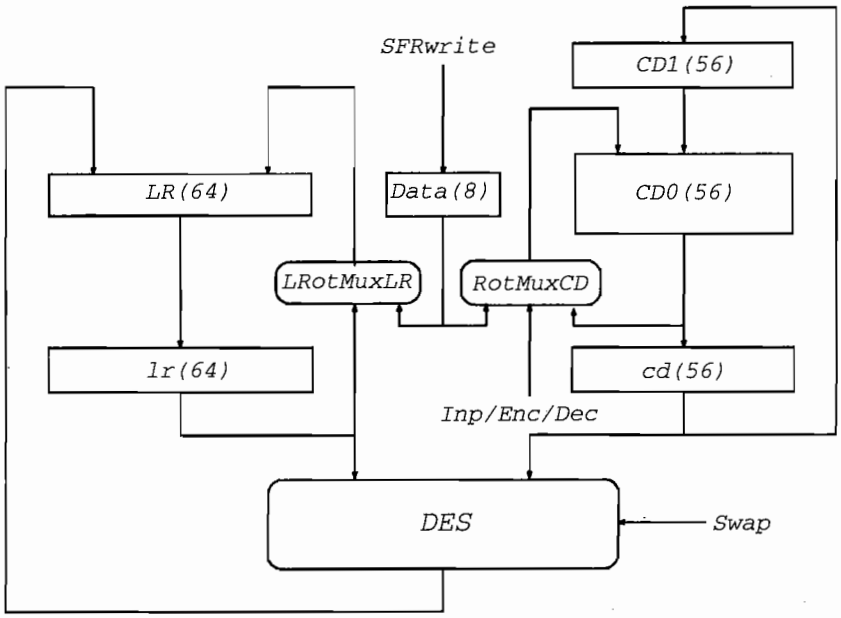


图13.16 DES协处理器的体系结构

单次DES变换包含16个步骤，每个步骤中密钥按次序改变，并由旧的文本值及密钥值计算出新的文本值。为了在转换结束时将密钥值恢复为起始值，在16个步骤中，有12个要进行2次基本置换，而在剩下的4个步骤中，要进行1次基本的置换。也就是说，在一次完整的周期中一共要进行28次基本置换。这些置换操作都是在CD0寄存器组中完成的。

芯片上大部分区域由称为DES的组合电路占用，同时它也是主要的功耗部件。为了减少功耗，我们应该使DES电路输入端的转换数量最小。因此，我们引入了两个锁存器：存放密钥值的cd及存放文本的lr。如果一个步骤要完成两个基本的置换，cd就隐藏了DES组合电路的第一个作用。另外，通过同时加载两个寄存器lr和cd，DES的所有输入在每一步中只变化一次，并把结果存储在LR寄存器中，如下面的Tangram程序所示：

```
(lr:= LR || cd:= CD0 ) ; LR:= DES(lr,cd)
```

因此，锁存器lr与从寄存器有点类似。由于两个密钥通过以下的形式进行互换，锁存器cd在这个过程中实现暂存数据的功能。

```
cd:= CD0; CD0:= CD1; CD1:= cd
```

DES协处理器由大约3250个等效门构成，其中57%是组合逻辑，35%是锁存器及触发器。因此，异步设计（延迟匹配及C元件）占了约8%左右的面积。在3.3 V电压下，单个DES转换需要耗费1.25 μs及12 nJ的能量。

图13.17给出了在3.3 V电压下DES协处理器的电流仿真情况（微控制器在DES计算前后处于活动状态）。实际的电流尖峰会因为供电电压的下降而变得更小（DES协处理器在0.5 V电压下可以正确地实现其功能），可以达到与缓冲电容相类似的电流尖峰（仿真中的最小分辨率为1 ns）。

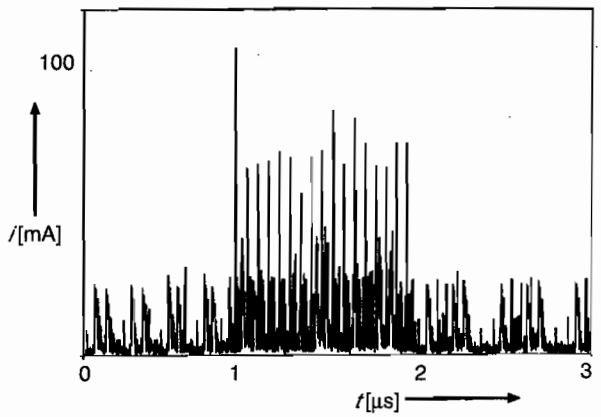


图 13.17　3.3 V 时 DES 协处理器的电流

转换时间只需要几微秒，这样短的时间是由于使用了握手机制实现微控制器与协处理器之间的同步转换。在启动协处理器之后，微控制器可以持续执行指令，只有在握手过程中等待读取结果有效时挂起很短的一段时间。注意，同步设计需要一种“忙碌-等待”（busy-waiting）的形式。

13.6　结果

芯片的版图如图13.18所示，它包括5层金属层，利用了0.35 μm工艺，含焊盘在内的总面积为 $4.52 \times 4.16 \approx 18\ mm^2$。大多数焊盘的作用仅仅是为了测量及评估，实际上一个成品芯片大概只需要10个焊盘。

芯片的版图上，上部两个水平的区域是缓冲电容（在成品芯片中，电容只需要这些面积的1/4）。存储器放置于下部的区域中，从左到右分别是：两个RAM，一个ROM及位于右侧所占面积最大的EEPROM。异步电路放置于接近中心区域的左下侧。

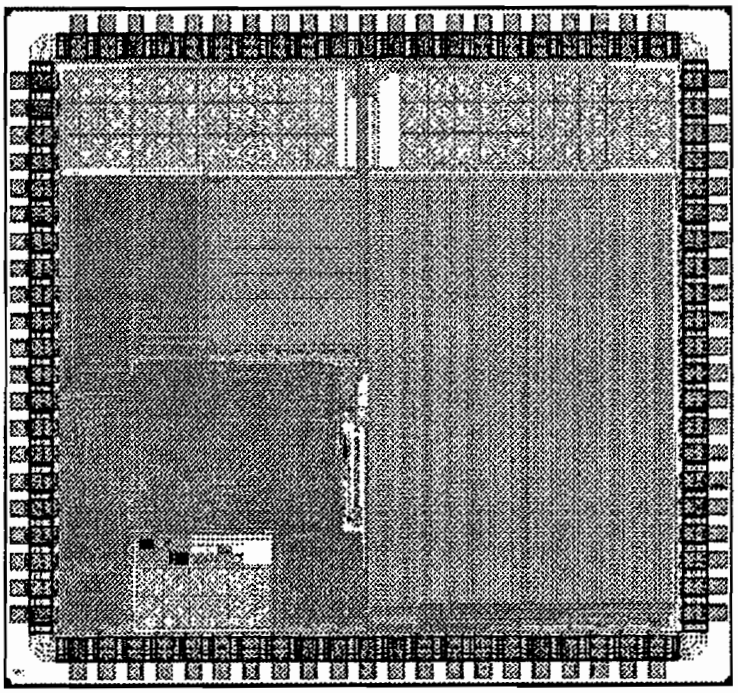


图 13.18 智能卡的版图

表 13.3(a)给出了组成非接触式数字电路的各模块所占的面积，包含异步电路及存储器在内。其他的模块要么是同步电路，要么是模拟电路，其中，同步电路模块并不应用于非接触操作。从下面的表中可以看出，异步逻辑大约占了非接触式数字逻辑电路的 12%。

表 13.3(a) 组成非接触式数字电路模块的面积

模块	面积（mm^2）
RAM	1.2
ROM	1.0
EEPROM	5.6
Async. circ.	1.1
总计	8.9

不同异步电路模块的大小如表13.3(b)所示。在所使用的标准元件库中，对于每平方毫米包含 17 500 门的典型版图密度而言，一个等效门（GE）的面积是 54 μm^2。

表 13.3(c)给出了数字电路模块在控制器执行 ROM 中的代码时的功耗情况（处于正常状态下）。

表 13.3(d)给出了在两个不同层次中，异步设计对于功耗及面积的影响。异步电路降低了 70% 的功耗，但增加了 18% 的面积。但是在非接触式数字电路中，我们仅多耗费了 2% 的面积而降低了 60% 的功耗。需要注意的是，这样的计算结果是在除去了芯片上的同步 RSA 转换器

及缓冲电容、供电单元等模拟电路的基础上的。因此，在对整个芯片进行计算时，所减少的功耗基本上和刚才所计算的功耗相差无几，而面积上的开销将会进一步减小。

表 13.3(b)　异步电路模块的面积

模块	面积[GE]
CPU	7800
Pref. Unit	700
DES	3250
UART	2040
Interfaces	3680
Timer	1080
Total	18 550

表 13.3(c)　非接触式数字电路的功率

模块	功率
Core	56%
ROM	27%
RAM	17%

表 13.3(d)　不同层次异步电路设计功率与面积的影响

层次	功率	面积
Async. circ.	−70%	+18%
Async. + Mem	−60%	+2%

13.7　测试

对异步电路制造缺陷的测试是一个众所周知的难题[61, 120]。主要的问题是，异步电路中包含很多难以用外部信号控制的反馈回路。这就使得扫描测试开销变得非常大，并且设计者要考虑到在功能测试开发方面：要么开发直接的测试模型，要么考虑如何实现“可测试设计”的方法。

本章所描述的芯片也采用了一种功能测试的方法。在测试过程中，微控制器将与包含测试程序的外部ROM相连。测试程序产生一个签名，如果产生的这个签名是不正确的，电路将报告它是有缺陷的。另外，使用电流测试增加了测试的覆盖率。

功能测试开发最初是用于80C51微控制器测试[144]。对处理器的测试及其扩展测试都利用了文献[152]中所介绍的测评工具。工具评估了功能路径的结构化测试覆盖面（可控性及可观性）。

对于数据通道逻辑来说，使用测试工具对输入固定型（stuck-at-input）故障模型在理论上可以达到100%的测试覆盖率。尽管如此，实际要达到这一点，对测试工程师来说仍然是一个挑战，但对于80C51微控制器，由于其内部寄存器及总线的可观性和可控性，做到这一点看来是可行的。

在不用“全扫描”的情况下，对于输入固定型故障模型，可使用一种已经改进的电路在异步控制逻辑中达到100%测试覆盖率，这种电路通过使用功能模式与电流测量相组合的方法来实现在精心选择的握手中的可暂停性[120]。由于这种修改在本文所提及的智能卡电路中并没有完全实现，因此故障测试覆盖率还不能达到100%。

对于此处所用的测试方法，其真正的错误覆盖率还不为人所知，正因为如此，使用验证工具计算功率将是不切实际的要求。不过，据估计，对于异步控制及数据通道子系统，可以达到90%的覆盖率。

13.8 电源

包含整流器及功率调节器的电源如图13.19所示，这两者都是由模拟电路构成的。整流器是传统类型的结构，而对于调节器，我们只讨论与数字电路有紧密联系的部分行为，至于设计细节，我们不做讨论。

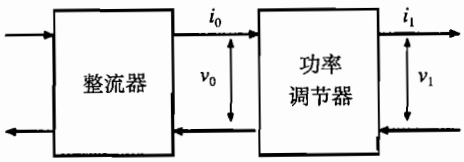


图 13.19 电源

为了避免影响到通信，功率调节器的输入负载被设计成接近于常量。图13.20给出了这种功率调节器在输入电压 V_0 为5 V时的SPICE仿真结果。横坐标是数字电路的活动性（每秒变换次数）。从输入电流 i_0 在整个图中趋于常量可以看出，输入负载基本上也是常量。

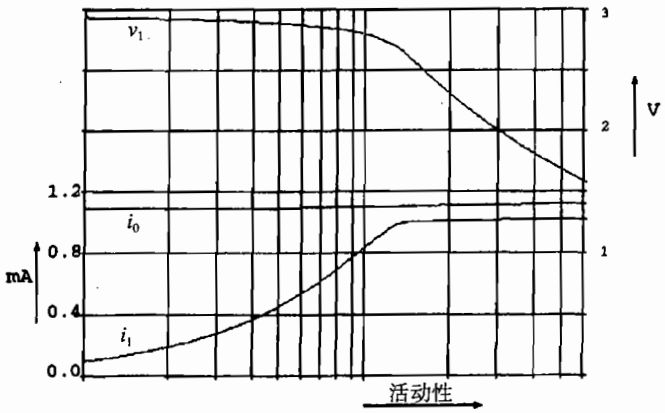


图 13.20 功率调节器特性

当活动性比较低时，输出电压 V_1 是3 V左右的常量。在这一阶段，大量的功率涌入，调节器的功能是实现输出电流 i_1 随活动性的增加而增加的电压源，多余的功率分流到地。但是，

当i_1不断增加，达到饱和点并接近于i_0时，从此时刻起，没有过多的功率分流到地，调节器的功能开始转化为电流源，输出电压V_1随活动性的增加而减少。调节器在这一变化范围的中间点时传输了最大的功率，此时，输出电压和输出电流都很大。需要注意的是，这些仿真结果都

从而会使得V_1产生波动。

电源的这种特性使同步电路设计者必须考虑在性能与鲁棒性之间进行适当的折中。举例来说，假设电源电压为3 V，如果使性能达到最大化，那么在电压低于2.5 V时必须中断处理。另一方面，设计者选择更好的鲁棒性，将工作电压设计为2 V，那么在电压变为3 V时性能将大大降低。而对于异步电路来说，就不需要这种折中，因为异步电路可以根据所获得的电压自动实现最高的性能。

13.9　小结

我们已经设计、构建并评估了非接触式智能卡异步芯片，并挖掘了异步电路的以下特点：

- 低的平均功耗
- 很小的电流尖峰
- 在较大的电源电压范围内正常工作

测试及仿真结果显示了在本设计中异步电路相对于传统的同步电路有以下几个优点：

- 异步电路能够根据所接收到的电源电压达到性能最大化。这主要是由于异步电路仅需较少的功率即可以工作，而功率是性能限制的主要因素。相对于同步设计，异步电路只需要40%的功耗，并只增加了不到2%的面积。另外，异步电路速度的自适应特性使得设计者不需要在性能与鲁棒性之间做折中。由于这种特性，异步电路可以自由地运行而不必考虑其性能保障，这也是异步电路与同步电路的另外一个明显的差别。
- 异步电路对于电压的下降具有很好的适应性，例如，在电压低于1.5 V时，它仍然可以正常工作。
- 异步电路所造成的电流尖峰比同步电路要小很多，这就放宽了对缓冲电容器的要求。
- 功率调节器与异步电路的结合对于通信基本上没有干扰，所以，小的电流尖峰与自适应能力是十分重要的。

致谢

我们衷心地感谢Tangram小组的其他成员：Kees van Berkel, Marc Verra及 Erwin Woutersen。另外，我们也感谢 Klaus Ully 帮助我们正确地实现了 DES 转换器。

第14章　异步维特比（Viterbi）解码器①

Linda E. M. Brackenbury
Department of Computer Science, The University of Manchester
lbrackenbury@cs.man.ac.uk

摘要：维特比（Viterbi）解码器用来对基于卷积码的前向纠错的数据进行解码。这种编码通常用于大规模的数字传输及数字记录系统中。使用这种解码器的理由是，即便在传输信号中混入了严重的噪声，解码器也能够有效地确定最接近的传输数据。

本章描述了一种新型的Viterbi解码器，其目的是通过使用异步设计方法降低功耗。根据计算及存储所要求的格式，新的设计是基于串行一元运算构建的，代替了传统同步系统之中的“相加-比较-选择”（Add-Compare-Select, ACS）的并行算法实现。与所有的Viterbi解码器一样，计算结果的历史记录是建立在很多数据位的基础上的，并以此来确定先前传送的最接近的数据。相比于传统的随机起始点的解码器，新型的解码器可以确定路径运算的起点，由此大大降低了对存储器的要求。另外，系统描述中的异步运算可以实现多重、独立、并发的追踪运算，从历史存储器中通过解耦对新数据进行定位。

关键词：低功率异步电路，Viterbi，卷积解码器

14.1　引言

PREST（Power REduction for Systems Technology，降低系统功耗技术）项目是一个协作项目[1]，每个参与者都设计了一种低功耗Viterbi解码器，作为除工业标准Viterbi解码器之外的另一种选择。曼彻斯特大学项目组的目标是利用异步时序来实现这种低功耗设计。

选择Viterbi解码器进行研究，是由于它是数字电视及移动通信之中的一个重要的功能部件。它可以检测并纠正传输错误，并根据计算结果输出与传送数据最接近的数据流。Viterbi编码广泛应用的原因是它可以处理连续的数据流，而不需要任何“起始块”或“结束块”的帧信息。另外，输出数据流的这种构建方式意味着，即便输出数据是不正确的，系统也可以适时地进行自我纠错。

① Viterbi解码器设计工作得到EPSRC/MoD PowerPack项目GR/L27930和EU PREST项目EP25242的资助。

14.2　Viterbi 解码器

14.2.1　卷积编码

为了理解解码器所实现的功能，首先介绍数据的编码方式。如图 14.1 所示，输入数据流进入一个初始值为零的移位寄存器之中，这个 2 位寄存器中存放着先前的 2 位输入，当前输入数据位的编码输出就是由这 2 个先前的输入位与当前输入位在 2 个模 2 加法器中相加产生的 2 个二进制数字所构成。

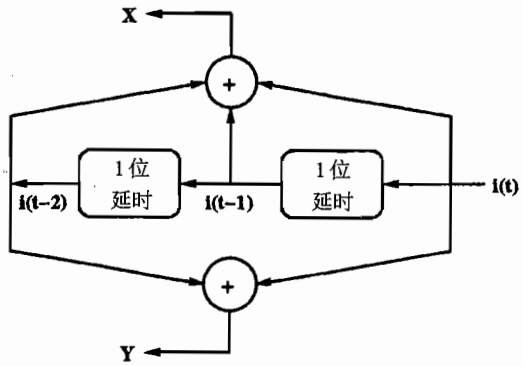


图 14.1　4 状态编码

例如，假设输入流由“011”组成，右边是当前的输入位“1”，左侧寄存器中存放的最早数据位是“0”，那么 X 编码输出为这 3 位数据之和，即“0”；Y 编码输出为当前输入的“1”与左侧寄存器的最早的数据位“0”之和，即为“1”。因此编码器在这个例子中发送的数据为“01”。由于每个输入位将引起 2 个编码位进行传送，故这种编码形式称为“1/2 码率”。

使用 3 位输入数据（称为“约束长度”k）表示编码器具有 $2^{(k-1)}=4$ 种可能的状态，从前面 2 位均为 0 的状态 S0 变到这 2 位数字都为 1 的状态 S3。因此，如果当前的状态为 n，输入为 0，则下一个状态是 $2n$ 对 4 求余；如果输入为 1，则下一个状态是（$2n+1$）对 4 求余。举例来说，如果现在的状态为 2，则下一个状态不是 0 就是 1，也就是说，每个状态将演化为 2 个已知状态。图 14.2 绘出了 4 状态系统的状态变化信息随时间变化的网格图（trellis diagram）。状态表示为节点，根据其预测性、规则性及形状，节点与节点之间的连接表示称为蝶形连接。

如果知道任一时刻网格图的起始状态和接下来的输入数据流，则可以追踪编码路径并计算出系统接下来的状态。例如，在 0 时刻系统处于状态 0，输入流为 1，1，0，1，则产生的曲线路径为：S0→S1→S3→S2→S1。路径如图 14.2 中的黑色粗线所示。

该图也表明了编码输出与网格图中各条路径或分支具有紧密的关系。例如，由 S3 到 S2 的状态变化，相当于当前的输入数据为“0”，而输入数据流一定为“110”（左侧的数据是最先输

入的），因此编码 X 和 Y 输出的是 0 和 1；这条路径则被标识为 01，表示编码器从状态 S3 到 S2 移动时的输出。其他的路径采用同样的计算和相应的标识方法。

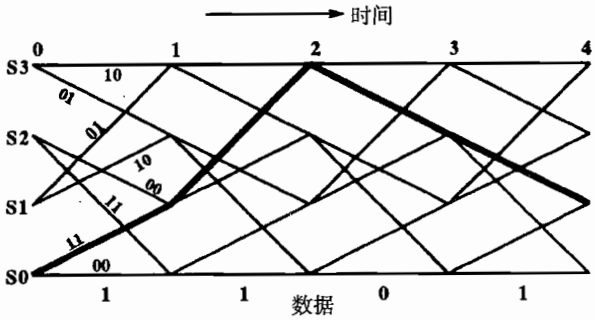


图 14.2 4 节点网格

14.2.2 解码原理

解码器试图重新构建最接近于编码器在网格图中所走过的路径。它通过构建网格图并在每个时间段内赋予节点及路径一定的权重，来表明这些节点和路径与编码器所经路线的相似程度。重新考虑前面的 4 节点的例子，输入数据为 1, 1, 0, 1，编码器输出为 11, 01, 01, 00（从起始编码状态都为 0 开始）。假设解码器没有接到这个序列，而是接到另一个错误的序列：11, 00, 01, 00，并且假设 0 时刻，节点 0 的初始权重为 0，其他节点的初始权重为 2，这符合编码器在 S0 状态开始进行的编码。

对任一分支，分支的权重为接收到的比特与分支所期望的比特之间的距离。这里 X 与 Y 被编码成单比特，根据硬编码规则，分支的权重为该分支接收到的比特与理想编码状态之间的不同比特的个数。图 14.3 标出了接收数据的每一个时间段（timeslot）中的分支权重。在第一个时间段中所接收到的数据为 11，代表编码输出为 11 的分支权重为 0，代表编码输出为 00 的分支权重为 2，代表编码输出为 01 或者 10 的分支权重为 1。这些分支权重代表的是分支与接收到的输入之间的距离。分支权重与节点权重之和构成了总权重。例如，状态 S2 的节点在 0 时刻的权重为 2，它的分支权重分别为 0 和 2，则在这个时间段的末尾接收节点的总权重为 2 和 4。总权重表明了编码器编码路径与所接收数据的接近程度，总权重越轻越接近。

从网格图中可以看出，每一个状态可以由另外两个状态演变而来，也就是说，每个状态都有 2 个输入的总权重。选择这 2 个权重之中较轻的一个，因为它较轻的分支代表着最接近编码器所要到达的下一个特定状态的分支。例如，状态 S1 可以在 1 时刻由状态 S0 或者状态 S2 到达，由 S0 到 S1 的总权重为节点权重加分支权重，即 0 + 0 = 0；而从 S2 到 S1 的总权重为 2 + 2 = 4。由 S0 状态到 S1 状态的分支总权重是较轻的，也就是说，这条分支是最接近于到达 S1 的编码路径的。由 S2 到 S1 的分支总权重较大，因而被放弃。第一个时间段末尾处的新节点权重为由 S0 到 S1 的总权重，即 0。

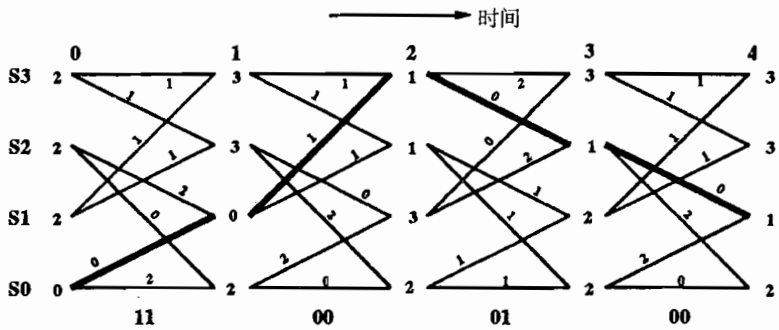


图 14.3　解码网格图

在每个时间段内都要继续进行同样的过程：根据接收到的比特与编码后的分支模式进行比较，比较的结果加上前一节点权重后得到总权重，其较轻者构成了下一个节点的权重，如图 14.3 所示。

为了产生解码输出，需要追踪网格图中带有节点权重累积的历史信息。时刻4的权重表明，由于S1具有最轻的节点权重，编码器在此时刻最有可能在这一状态。另外，S1状态最有可能是从S2到达的。继续在S2之前追踪权重较低的节点，最相似的路径为S3, S1, S0（初始化点）。需要注意的是，即使接收到的是损坏的数据，仍然可以得到编码器所发送的正确数据序列！

注意到要到达状态S0及S2，编码器的当前输入为“0”；而要到达状态S1和S3，编码器的当前输入为“1”；输出数据可以从解码器“解救”出来。也就是说，状态最低位的值代表了当前数据的状态。那么由于解码器在时刻 1, 2, 3, 4 到达的状态分别为 S1, S3, S2, S1，解码器将输出数据流 1101 作为与编码器的输入最接近的数据。

14.3　系统参数

在实际应用中，解码器的设计要比上文所介绍的简单例子更大、更复杂。编码器用当前输入位与先前的6位以不同的组合提供2组编码输出流；这扩展了每位的传输时间，在具有噪声的条件下，增加了无错误接收的可能性。如果当前的输入位为第0位，移位寄存器最早的输入位为第6位，那么X编码输出是将第0位、第1位、第2位、第3位及第6位相加，而Y编码输出则是将第 0, 2, 3, 5, 6 位相加。由于约束长度为7，则系统具有 64 个节点和 128 条路径或分支。那么状态 n 是 0 ~ 63 的整数，它的下一个状态为 $2n$ 对 64 求余或者 $(2n + 1)$ 对 64 求余。

另外，接收到的数据不是硬判决值（直接是 1 和 0），而是软判决值。数据的值由 3 位二进制数决定，“100”代表强零，“011”代表强 1。传输噪声的存在意味着所接收到的字符可由 0 ~ 7 的任一 3 位数表示，并被当成是接收值，把数字记为有符号数是很有用的。那么 011 表示强 1，而 000 表示接收到的数据是弱 1。同样，100 表示强 0，而 111 很可能表示弱 0。为陈

述简单起见，后面采用3位无符号数编码，即3位全0记为“000”（也就是值0），3位全1记为“111”（也就是值7）。

异步解码器的接口是同步的，在块有效（Block-Valid）信号作用下，接口处有效的编码字符的移入与移出与一特定时钟的上升沿同步，有效的编码字符只出现在块有效信号高电平期间。除本章先前介绍的1/2码率以外，通过使用Block-Valid以及Puncture和Puncture-X-*n*Y信号可也以获得其他的码率。如果Block-Valid为高电平，且Puncture信号为高电平，则表示X和Y中只有一个有效，而Puncture-X-*n*Y用来指明其中哪一个有效。如果Block-Valid为高电平而Puncture信号为低电平，那么两个编码的字符都有效。所有的码率都源于1/2码率的编码器，其他码率的数据通过省去其中一些码字符获得。通过这种方法，除了1/2码率之外，系统还可以对码率为2/3（以2个输入位产生3位码字的传输），3/4，5/6，6/7，7/8（由8个码字符定义7个输入位，其中保留第6位不做传输）进行接收与解码。由于码率逐步增长，在传输数据过程中的冗余越来越少，导致解码器的误码率越来越高。1/2码率所产生的误码是最少的。

系统预期在90 MHz的时钟频率下运行，所有的码率都使用这一时钟。码率不仅定义了输入位个数与传输编码字符个数之间的比率，还定义了包含编码信息（1或2个有效字符）的时钟个数与时钟周期个数的比率。例如，3/4码率意味着以3个输入位产生4个传输字符，即每4个时钟中有3个包含编码信息，因此，在3/4码率中所有第4个时钟不包含编码信息（Block-Valid为低电平）。Block-Valid为高电平时的任何一个时钟都被称为包含一个“码符”（symbol），那么对于90 MHz的时钟频率，1/2的编码率，数据在交替的时钟周期下有效，产生的数据传输率是45 M码符/秒，7/8编码率则等于90 × 7/8 M码符/秒 = 78.75 M码符/秒。

14.4 系统概述

为了完成上文所介绍的解码运算，Viterbi解码器由3个单元构成，如图14.4所示。分支度量单元（Branch Metric Unit, BMU）接收传输的数据并计算(0, 0),(0, 7),(7, 0)或(7, 7)的理想分支模式码符与接收到的数据间的距离；计算后的权重传输至路径度量单元（Path Metric Unit, PMU）。PMU保存节点的权重，并完成节点权与分支权的相加计算，选择其中最低的权重作为下一个时间段中这个节点的权重。计算得到这些节点的权重，然后将其送回（在PMU内）并成为下一个时间段的节点权重。

当计算下一个时间段节点权重时，PMU记住获胜者，即从上面的还是下面的分支到达该节点，每一个节点只需要一位进行记录。对于每一个时间段，局部获胜者信息被记录在历史单元HU（History Unit）中，在同步系统中被称为“幸存路径存储单元”（survivor memory unit）。这个信息使HU能够保存网格图中每个节点的历史路径。除了这些局部节点信息，在我们的异步设计中，具有最小节点权重的状态由PMU进行识别并将其存入HU之中。通过查找全局获胜路径得到一个已知的起始点，沿网格历史图向后追踪可以找到最佳的输出数据。

传统的同步设计并不考虑全局获胜节点识别，而是在任意一个随机节点开始搜索。为了确保有足够的历史记录通过网格图最终找到正确的路径，同步设计需要存储相对较大数量的时间段信息。在异步设计中，在PMU中全局获胜节点的识别可以相对简单并很自然地完成。由此可以减少HU之中所存放的时间段历史记录数目，并减少HU处于工作状态的时间，从而达到节能的目的。

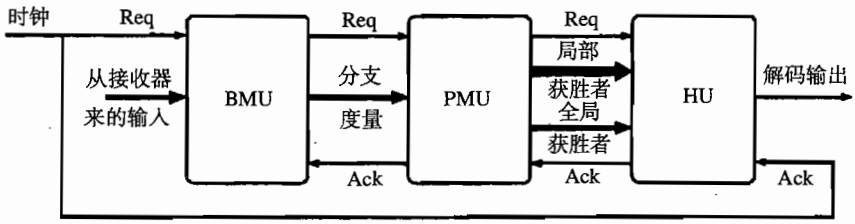


图 14.4　解码器组成单元

HU不仅使用全部获胜节点（全局获胜者）信息，而且使用局部获胜节点来重构网格图，并通过当前的全局获胜节点的路径追踪寻找所存储的最早时间段中的节点，并将这一节点中的比特输出。HU可以被视为一个环形缓存。当数据从最早的时间段中输出时，这个信息将被当前（最新的）获胜节点数据所覆盖，这样最早数据的下一个便成了下一个时间段内的最早数据。

图 14.4 给出了单元间的捆绑数据接口，4 相信号用来组成请求（request）和应答（acknowledge）的握手信号。由于外部系统到解码器是同步的，因此时钟信号还是必要的。在Block-Valid有效的情况下，在时钟的上升沿完成数据的输入与输出。实际上，所有的单元接口处都有缓存，用于隔离各个单元的操作、满足运算过程中局部变量的需要及外部系统对延迟的要求。

14.5　路径度量单元（PMU）

14.5.1　PMU 中的节点对设计

PMU是解码器计算功能的核心，也是设计的起点。一般来讲，计算的过程包括相加（节点的权重加上分支的权重）、比较（节点的上、下分支的权重）和选择（较低的权重作为节点的权重）。由于是蝶形连接，分支的权重与节点j、节点$j+32$有关，而且它们连到节点$2j$、节点$2j+1$，如图 14.5 所示。BMa、BMb代表分支的权重。需要注意的是，由于一个分支代表了理想的卷积字符(0, 0), (0, 7), (7, 0), (7, 7)，所以在任何系统中，只需计算 4 个权重的总和即可表示它们与所接收到的软判决字符之间的距离。

由于这种节点对的计算是独立的，所有的计算都可以被分割为相互独立的节点对，因此PMU中逻辑设计的基本单元就是节点对，并根据系统的要求重复设计若干次（本系统中进行

32次）。另外，由于系统通常采用的是8位并行运算，因此对于64个节点的系统，其蝶形连接中有512个数据信号，PMU中存在1024个互联。

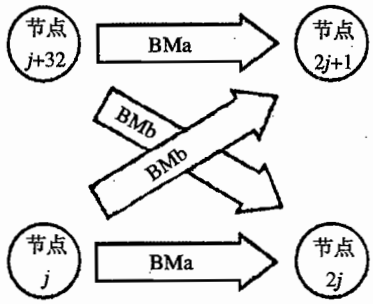


图14.5 节点对计算

为了简化布线问题并以此达到减少功耗的目的，异步设计中采用串行计算，原则上，这可以将蝶形连接减少至64根连线。采用串行计算会对节点的设计及存储权重的方法产生很大的影响。传统的串行运算采用二进制的数值表示方法，但是系统的吞吐量达不到规定的要求，故在本设计中并不适用。这就产生了用进入FIFO缓冲器的数据的个数表示数值的想法。例如，对5计数时要求缓冲器的底部5级显示为满，注意，缓冲级并不需要存储任何数据，只需要一个满/空的指示即可。

通过采用串行一元算法表示缓冲器中的数据（要比“1”代表满，“0”代表空更好），可以使这种不包含数据的满/空FIFO的速度与简明性得到进一步提高。这本质上是数值的2相表示，这样就可以把满/空位数据的跳变次数作为一个计数的完整表示。表14.1是6级FIFO，左侧是数据输入，右侧是数据输出。

表14.1 串行一元运算

数值		跳变表示
0	=	000000
1	=	111111
2	=	000001
3	=	111101
4	=	000101
5	=	110101
6	=	010101

保持路径权重及状态度量的FIFO是用Muller流水线实现的，如图14.6所示（也可参考图2.8）。度量编码M是初始条件为空的Muller流水线在输入经过M次2相握手后的状态。由于所使用的Muller C单元相关工艺会产生1 ns左右的传输延迟，因此FIFO的传输速度为500 MHz左右。

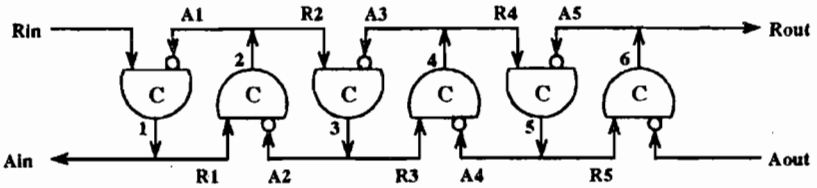


图 14.6　6 位 2 相无数据 FIFO

2 个节点之间的串行一元运算是通过“加减”单元来实现的，这个单元的作用是将节点的权重与分支的权重相加，计算出较小的总权重作为节点新的权重，它是设计的重点。节点对的基本框图情况如图 14.7 所示：

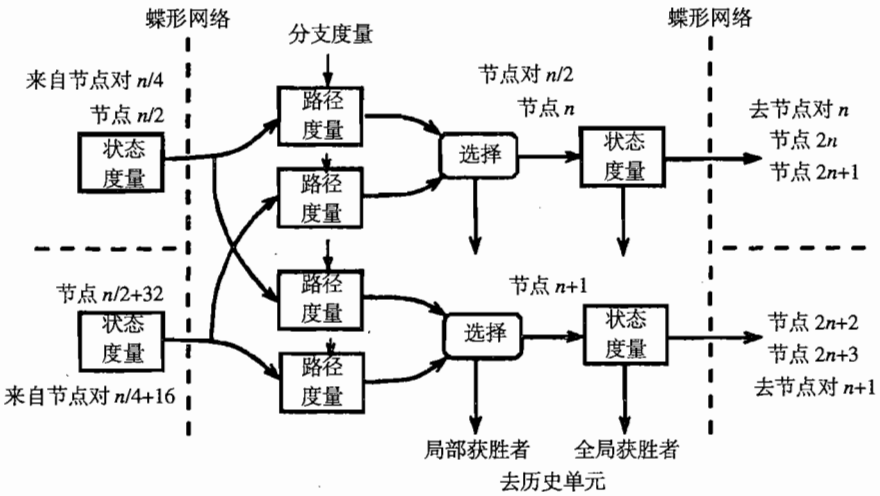


图 14.7　节点对逻辑

每个状态新的权重存储于状态度量 FIFO 中，也就是图 14.7 右侧的部分。当全局和局部获胜者送入 HU 并得到确认后，便进入下一个时间段，此时如图 14.7 左侧所示，把分支权重并行载入到路径度量 FIFO 之中，并覆盖 FIFO 中的所有内容。这里使用并行载入而不是串行输入，原因是为满足速度及清除路径度量 FIFO 中已存数据的需要。

载入的分支权重是由 BMU 进行计算的。BMU 首先基于通过网格图期望的 2 个 3 位字符与接收到的 2 个数值之间的差别计算传统的分支权重，然后转换成跳变的形式。由于路径度量单元外部环境的影响，如有时状态<1>需要对得到的跳变形式进行 2 相反演（2-phase inversion），使得这个过程变得更为复杂。

分支权重一旦载入，节点的权重会与它相加。节点权重会通过蝶形连接从状态度量 FIFO 串行传输至路径度量 FIFO。每传输一次后，2 个路径度量单元 FIFO 进行加 1 操作，状态度量

FIFO则减1。当状态度量FIFO为空时，传输完成。此时，节点对的路径度量FIFO开始进行比较并选择较低的计数值作为节点权重。

实现对较低路径度量FIFO计数值的比较和选择，最简单的方法就是将与接收状态度量FIFO相连接的上下路径度量FIFO的跳变（事件）进行配对比较。成对的事件使每个路径度量FIFO减1并产生一次跳变，用来使状态度量FIFO加1。细心的读者可能会注意到，事件对产生的一次输出变化事实上是通过一个二输入的Muller C单元来实现的，从原理上讲，它们是构成图14.7所示的“选择单元”必要的部件。

当2个路径度量FIFO中较小的计数值减少到0时，配对的动作中止。此时，这个路径度量FIFO是局部路径的获胜者，并且接收状态度量的FIFO中的新节点权重计算完成。重构网格图需要获胜路径度量FIFO（上面的或下面的）的特征信息，这个信息将在PMU中缓存，当得出所有的局部获胜者，且全局获胜节点经过鉴定后，送至HU之中。从而完成当前时间段的行为，然后PMU便可以进行下一个时间段的行为了。

14.5.2 分支度量

只有在FIFO间传输较小的数值，所提的方案才有效。为了确定能满足工业标准设备要求误码率的最小数值，我们设计了一个仿真器。

在BMU之中，需要计算用(0, 0), (0, 7), (7, 0), (7, 7)表示的理想分支到输入数据间的距离，如图14.8所示。输入数据的值假设为(a, c)，它不与任何的理想点对应。距离$d00, d01, d10, d11$的平方分别为$a^2 + c^2$, $a^2 + d^2$, $b^2 + c^2$和$b^2 + d^2$。我们感兴趣的只是这些量之间的相对值。在这个二次表达式中，用$(7 - a)$代替b，$(7 - c)$代替d，于是得到平方距离为：$a^2 + c^2$, $a^2 + (7 - c)^2$, $(7 - a)^2 + c^2$及$(7 - a)^2 + (7 - c)^2$。展开后减去$a^2 + c^2$得到的距离值为：0，$49 - 14c$，$49 - 14a$和$98 - 14a - 14c$。除以7，加上$a + c$，然后以b代替$(7 - a)$，d代替$(7 - c)$，产生线性度量$a + c$, $a + d$, $b + c$, $b + d$。

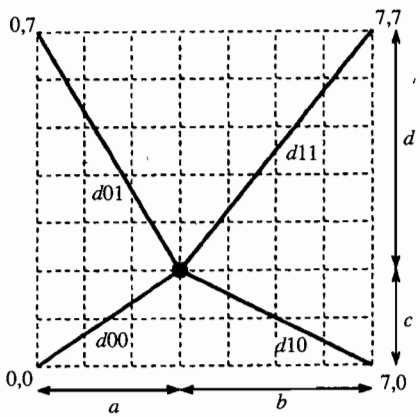


图14.8 分支度量计算

这样，在这一特定的系统之中，欧几里得距离的平方与曼哈坦距离等价，这个结果有点出人意料。这表明，使用平方距离并不比使用线性距离好。实际上，使用平方距离按比例地减少了数据量（某些系统采用这种方法），但会导致不必要的电路复杂性和某些误差。

通过从分支权重中减去x和y与最近的理想点之间的距离，线性权重得到进一步减小，因此最小的线性度量总是0。举例来说，如图 14.8 所示，如果输入的软编码是 7, 7，然后产生线性度量 14, 7, 7, 0。但是，如果输入数据的坐标为 5, 6（由于噪声的影响），那么度量为 11, 6, 8, 3，将所有的值减去 3（x值为 2，y值为 1），变为分支度量 8, 3, 5, 0。到最近点的最小距离变为 0，每个 3 位字符的权重值减少情况如表 14.2 所示。

表 14.2　分支度量权重的生成

	接收到的3位字符							
	0	1	2	3	4	5	6	7
相对于0的权：	0	0	0	0	1	3	5	7
相对于7的权：	7	5	3	1	0	0	0	0

现在最大的度量是 14，且总是出现在输入数据与一个理想的输入组合值一致时。对于运行于 90 MHz 的串行系统来说，这个值还是太大了。因此，度量会进一步非线性地减小，而且为了保持权重的相对值，对 2 个输入 3 位软编码值分开来实现。

参考表 14.2，0 的权重仍然是 0，而 1, 3, 5, 7 的权重成为 1, 2, 3, 4。显然，2 个 3 位软编码数值需要相加而得到全局分支度量权重。例如，输入软编码 7, 7，产生的权重为 4+4, 4+0, 0+4, 0 = 8, 4, 4, 0，而当输入 5, 6 时，产生的权重为 2+3, 2+0, 0+3, 0 = 5, 2, 3, 0。虽然这里的减小是非线性的，并可能会降低系统的准确性，但是，仿真显示按等比例变化和不按等比例变化的结果差别并不显著。

此外，通过仿真显而易见，在重构最相似路径来判断输出的过程中，具有最大权重的路径基本都被抛弃掉了。因此，可以对所产生的权重进行限制和约束，如可以限制 BMU 的最大权重为 6。那么 BMU 对软编码输入 7, 7 所产生并加载的权重为 6, 4, 4 和 0；不过对于输入 5, 6 所产生的权重仍然与上一段的情况一样。

PMU 中所使用的都是 6 位 FIFO，其中 PMU 中的数量再次被限制为 6。为了处理节点度量与分支权重串行相加而造成的超过限定数量情况，在每个路径度量单元之前增加了溢出单元。这个单元接收输入的握手请求但并不将其传递给 FIFO，而是将它作为确认信号返回到发送状态度量的 FIFO 之中。

14.5.3　段的时间选择（Slot timing）

PMU 中特定时间段内的全局（或总体）获胜者是具有最低状态度量计数值的节点。同样，BMU 中的数据经过调整可以使其最低权重值为 0，状态度量值也可以减小并使其为 0。那么，就可以保证 BMU 和 PMU 中的数值在 0 ~ 6 范围内。另外，大多数情况下，软编码数据不包含

噪声，那么一个（仅仅一个）状态度量FIFO将包含一个零值用以表明网格图中的最佳路径。这意味着，在大多数的时间段中，状态度量并不需要进行任何减法操作。

由于FIFO中所有状态为0表示零值，因此检测计算值是否为0是比较容易的。与挑选局部获胜者一样，这个过程也是在局部节点完成，并由应用于此节点的控制信号实现时间选择。每一个节点都有一个能产生节点所需时间选择信号的控制部分，且这个节点内部的时间选择与所有其他节点的时间选择无关。

一个时间段开始于BMU分支权重的载入。节点时间选择传到下一级，并完成“状态到路径”（state-to-path）的度量传输。接着，检测发送数据的状态度量FIFO和接收较小路径度量值的状态度量FIFO是否为空，如果为空则产生状态到路径度量结束信号。然后节点时间选择转移到下一阶段，产生“路径到状态”度量使能信号。如果此时这一信号有效，且这一节点的路径度量FIFO值为0，那么触发器被置位，表明这个节点就会成为全局获胜节点的候选节点。因此，不必传输到状态度量FIFO中就可产生路径到状态度量结束信号，这个信号用于产生将上/下分支的（局部）获胜者存入触发器的时钟，并将一个标志着“已经找到局部获胜者”的触发器置位。

如果任何路径度量FIFO都不为空，则路径到状态使能信号将允许传输继续进行至状态度量FIFO，直到有一个路径度量FIFO为空。从而产生路径到状态度量结束信号，置位“局部获胜者”及“已经找到局部获胜者”触发器。

由于所有需要传送给HU的信息必须在PMU向HU发送请求输出信号之前准备就绪，此时“已找到局部获胜者”和“已找到全局获胜者”信号会将PMU的逻辑层次提高到顶层。另外，当局部和全局获胜者的数据都集中于HU之中时，所有节点必须都被告知，此时间段已经结束，时间选择需要转换到下一个时间段的开始状态。需要注意的是，虽然在节点内的时间选择是局部的，但传递获胜节点信息到HU的通信以及接下来发布的节点都是全局的。

14.5.4 全局获胜者确定

所有“已找到局部获胜者”和“已找到全局获胜者”的确认信息被分布在不同级别的PMU逻辑层次中。在节点对层次中，4个全局获胜者候选信号被送入优先权产生单元，如果其中的一个输入是候选信号的话，在此单元产生2位节点地址和一个“已找到全局获胜者”信号。在4对这样的节点对中，重复这样的逻辑行为，并通过使用每个节点对所产生的2“已找到全局获胜者”信号所得到的两位地址来表明哪对节点是全局获胜节点，然后将其与节点对所产生的2位地址相结合成为4位的节点地址，最后，在具有4组4对节点对的顶层重复这样的逻辑操作。每组中的“已找到全局获胜者”信号再次通过优先权逻辑产生2位的地址信号，将其与获胜节点组中的4位地址相结合，产生6位节点判定信号，并将其送入HU中作为全局获胜者地址。

在节点对、4对节点对和顶层，“已找到局部获胜者”信号只需要由与非门或者或非门组成的逻辑电路产生。在顶层，所有“已找到局部获胜者”信号为真时，才可以向HU发出输出

请求信号。由于全局获胜节点出自于局部获胜节点，这样就确保最后一个局部获胜节点在全局获胜节点前找到。作为响应 PMU 发出的请求，从 HU 发出的确认信号引起一个为所有节点复位的信号，该信号复位全局候选节点和“已找到局部获胜者”触发器并将时间段推进。

如果检测到所有的状态度量寄存器都不为0，则表明接收到了噪声信号。此时，需要通过将一个或者多个包含最小计数值的状态度量 FIFO 中的内容减去一个值，使之变为 0，从而判定状态度量全局获胜节点。这个过程需要耗费一定的时间，并且这种减法运算不在当前的时间段内进行。此时，送入 HU 的并不是局部获胜节点信息，而是一个“非有效”信号，伴随的请求握手用以表明，当局部获胜节点信息是真时，应忽略全局获胜节点的判定。

所有状态度量单元中权重的减少是在下一个时钟周期由溢出单元（位于路径度量 FIFO 之前）完成的。当有信号输入到这一单元时，则表明需要进行减法操作。这导致由状态度量 FIFO 到路径度量 FIFO 进行数据传输的第一个输入请求被忽略。溢出单元返回一个确认信号给它的发送状态度量 FIFO，但并不把请求信号转发给路径度量 FIFO。由此，通过丢弃发送到路径度量 FIFO 数据的首项来有效地使状态度量 FIFO 减 1。

通过这种方式，在任何时间段内只能进行一次减 1 操作。但是，在这种操作之后，仍然可能使所有的状态度量值都不为 0，不过仿真表明，这种状况极少发生。另外，如果溢出单元被用来减去状态度量中的最小值，那就要么需要用额外的逻辑来判断这个最小值的具体数值，要么采用耗时逻辑将所有的状态度量减 1，然后进行比较，如果仍然全不为 0，再进行减 1 操作。这里所使用的是对于所有的状态度量值的非零检测，而不是对于一个状态度量值是否为零值的简单检测方法。

综上所述，最好在当前时间段内将状态度量减小到 0。否则减运算不得不发生在某些时间点并通过移位来延时到下一个时间段。更为重要的是，在所有状态度量 FIFO 都保持一个非零计数值的情况下，产生的对于全局获胜节点的判断失败，意味着 PMU 中的节点信息没有传递到 HU 之中，因此 HU 没有足够的信息用以判断数据的输出。

14.6　历史单元

一般来讲，由 PMU 到历史单元之间传送全局及局部获胜节点信息是通过“请求”握手信号来完成的。确定了与 PMU 的接口之后，就可以将异步 HU 的设计从整个系统的设计之中分离出来。如上文所述，与需要从网格图中任一随机节点向后追踪的系统相比，全局获胜者身份的确定意味着保存于 HU 之中的局部和全局获胜节点历史中的时间段数目可以减少。根据经验，用于决定正确输出而需要存储的时间段最小数目应该是约束长度的5倍左右。如果系统的约束长度为 7，那么需要最少 35 个时间段的历史记录，这点通过仿真可以证明。基于此点，本系统的 HU 之中记录了 65 个时间段的信息。

14.6.1 运行原理

图14.9描述了HU的工作原理，为简单起见，图中的矩形方框内只包括了4个状态的5个时间段。每个时间步用T1, T2等标识，PMU提供局部获胜者信息（由当前时间的每个状态用箭头向后指向前一个时间步的上面/下面的获胜者状态）以及用实心圆表示的全局获胜节点信息。

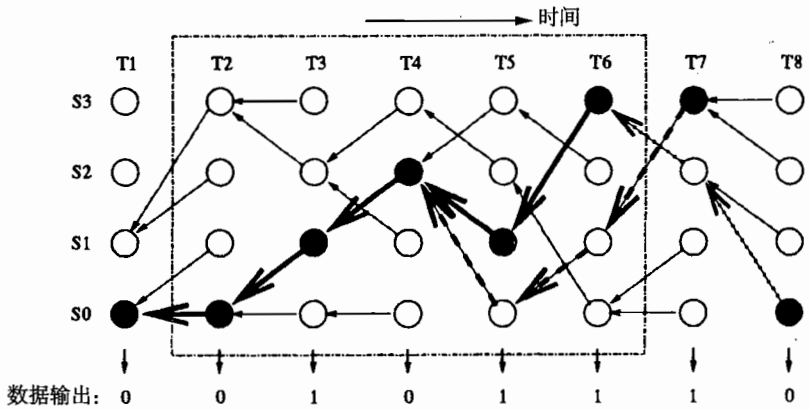


图14.9 历史单元追踪举例

假设PMU所提供的最新数据在T6时刻。那么S3作为全局获胜者，按照箭头追踪到T2的全局获胜者是S0。因此下一个数据输出位是0（S0状态的最低有效位），并且这一输出使得缓冲窗向前滑动至T7。在T7时刻，接收到的数据已经被噪声破坏，全局获胜者（错误的）成为S3。在T6和T5时刻，依照向后追踪的局部获胜者判断箭头，将全局获胜者进行了修正，但是路径与T4旧路径重合。下一个输出数据（来自T3的全局获胜者）仍为正确的输出值“1”。移至T8，全局获胜者为S0，向后追踪到T5, T6, T7，全局获胜者发生了变化，在T5, T6恢复了其前面的正确值。在T7所造成的错误路径并没有造成持续的影响，而且输出的数据流是无差错的。

14.6.2 历史单元回溯

HU中所使用的数据结构如表14.3所示。65个时间段之中的每一个均包含64位的局部获胜者信息及6位的全局获胜者判定信息。除此之外，还有一个“全局获胜有效”标志位用于表示全局获胜者是否已经计算出来。65个时间段构成一个环形缓冲器，时间段的开始位置（结束位置）按顺序在缓冲器中步进。每一步中，由当前起始时间段（head slot）全局获胜者标识的最低有效位产生下一个输出位。然后，新的局部及全局获胜者信息写入至起始时间段之中，起始时间段的指针移动到下一个时间段。新的局部及全局获胜者信息用于初始化回溯，更新了保存于全局获胜者存储器中的当前最优路径。

表 14.3　历史单元数据结构

时间段序号	局部获胜者（64 位）	全局获胜者（6 位）	有效（1 位）	
0	L00[63…0]	G00[5…0]	V00	
1	L01[63…0]	G01[5…0]	V01	
…				
18	L18[6…0]	G18[5…0]	V18	
19	L19[6…0]	G19[5…0]	V19	
20	L20[6…0]	G20[5…0]	V20	←起始
21	L21[6…0]	G21[5…0]	V21	
22	L22[6…0]	G22[5…0]	V22	
…				
64	L64[6…0]	G64[5…0]	V64	

网格图的排列使一个状态与另一个状态之间构成了一种简单的算术关系，以至于在一个时间段内给出全局获胜者身份，那么前一个时间段的全局获胜者可以立即计算出来。“父身份”可以通过“子身份”推导出来，也就是将子状态右移一位，并插入相应的“局部获胜者位”至最高有效位。例如，如果全局获胜者是时间段中的 23 号节点，那么前一个时间段的全局获胜者将是 11 号节点（如果当前 23 号节点的局部获胜者是 0），或者 11+32 = 43 号节点（如果节点 23 的局部获胜者为 1）。

当前全局获胜者的位置就是前一个时间段全局获胜者子关系的位置，当前获胜者不必再通过局部获胜者信息回溯重构网格图，可以继续使用存储于全局获胜者存储器中的优良路径，从而节省了功耗。因此，当接收到来自 PMU 的数据时，如果输入的全局获胜者是最后一个获胜者的子关系，那么唯一需要做的是从最早的全局获胜者入口输出数据，然后用输入的局部和全局获胜者信息覆盖存储器的内容。

但是，如果存在足够大的噪声（或者噪声已经存在，并且现在的数据切换回到一个正确的数据流），那么在输入的全局获胜者和前一个全局获胜者之间就会存在着间断，这可以通过识别当前的全局获胜者是否为前一个获胜者的子关系来判断。在这种情况下，全局获胜者存储器没有保留优良路径，需要通过局部获胜者信息来重构路径。像以往一样，在这里，将数据读出，并将获胜者信息写入。另外，从当前的全局获胜者开始，节点身份用于从当前的局部获胜者中选择它的上/下分支获胜者。那么“父身份”就如前面所说的那样被构建。这种计算得到的“父身份”是相对于前一个时间段的全局获胜身份而言的。如果它们是相同的，那么回溯路线将汇聚到全局获胜节点存储器中的优良路径之中而不需要进一步的行动。但是，如果结果不相同，那么计算的“父身份”需要覆盖前一个时间段的全局获胜者。为了构建到先前的下一个时间段的优良路径，需要重复回溯过程，直到计算得到的“父身份”与存储的全局获胜者符合为止。

逐个时间段的向前回溯直至计算得到的路径收敛于已存储的路径。图 14.10 给出了类似于 Balsa 的伪代码算法（注意，Balsa 没有如“<<”或者“>>”这类的移位运算符，图中的这些符号是为了增加可读性从 C 语言中借用而来的）。实际上，仿真的结果表明，通常在 8 个或者更

少的时间段内，这种路径收敛即可完成。因此，尽管最近的几项可能会被覆盖，而最原始的项并没有改变，从而最早时间段的输出不发生变化。改写整个路径是很少出现的现象，一旦这种情况发生，则系统的输出数据基本上是不正确的。

```
loop
   c := head;                        -- child starts at head
   data_out <- Gc [0];               -- output lab of Gc
   Lc := In.10cal_winners;           -- update head local winners,
   Gc := In.gl0bal_wlnner;           --   global winner, and
   Vc := In.gl0bal_winner_valid;     --   global winner valid bit
   head := head + 1;                 -- step he&a pointer to next slot
   if Vc then                        -- backtrace only from valid head
     p := (c-l) % 65;                -- parent slot number
     while (c /= head                -- detect buffer wrap-around
            and (not Vp              -- over-write invalid parent
            or  Gp /= (Lc[Gc] << 5) + (Gc >> 1)))) -- not converged
     then
       Gp := (LC[GC] << 5) + (GC >> 1));          -- update parent
       Vp := TRUE;                   -- mark ms valid
       c  := p;                      -- next slot
       p  := (c-1) % 65              -- next parent slot number
     end -- while
  end -- if
end -- loop
```

图 14.10 历史单元追踪串行算法

在任何时间段，当全局获胜者无效时，并不进行回溯工作。即使全局获胜者的入口单元被标识成无效的，局部获胜信息照样按照正常情况写入。任何回溯过程遇到无效的全局获胜者，它都会在那个时间段施加一个与输入计算的全局获胜者不相等的值，从而用计算所得的全局获胜者取代无效的存储值并将入口标识为有效。

14.6.3 历史单元的实现

HU 中所用存储器的类型是决定其实现的主要因素。最初，使用单元库中已有的 RAM 单元作为单口或者双口存储器。但是后来发现这种存储器对于追踪一个没有完成却又需要开启新的回溯的过程是十分困难的。另外，全局获胜者与局部获胜者存储器需要分离，但是这使得地址译码的效率不高。还有，在这种情况下提供较多的用于驱动内存的专用时序信号是比较困难的。RAM 的时序与一些简单的门延迟是类似的。这些门将用于构建参考时序信号，对于由供电电压的变化而导致门延时的变化是否与 RAM 延时的变化相同，这一点还不清楚。

鉴于上述原因，存储器是由触发器组成的，系统如图 14.11 所示，它由具有同样存储结构的 64 个段（slot）和 1 个附加段组合形成 65 条线，复位时，附加段拥有头托肯（head token），第 1 个段从 PMU 中接收最新的局部和全局获胜者信息。控制模块的功能是保留全局获胜节点身份信息、处理托肯的流动及进行逻辑回溯。

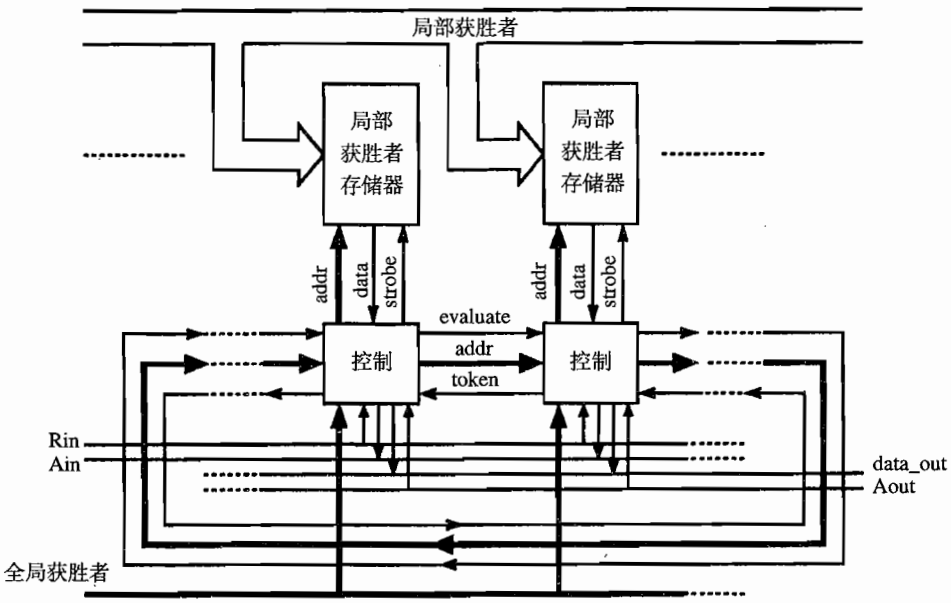


图 4.11　历史单元的实现

图 14.12 提供了并发异步算法的类 Balsa 伪代码，它显示出历史单元中的一级。完整的 HU 由这样的 65 个级组成，其中的一个会被初始化为环形缓冲器的头部。可以将此种算法与图 14.10 所示的顺序算法相比较，通过由顺序算法发展到并发算法是高级异步设计方法，哪怕设计是手工完成（如本文所说的 Viterbi 解码器）而并非用高级语言完成的（如 Balsa）。

第一个段中包含着最早的获胜者信息，并由此决定系统的第一位输出。需要强调一点，奇数状态表示输入为“1”，偶数状态表示输入为“0”，第一段将全局获胜者最低有效位输出到“data-out”线上。这个数据进入缓冲器中，缓冲器就会发出确认信号 Aout，表示数据已经被接收。此时，第一个段即成空闲状态，可写入当前获胜者信息到相应的存储器中。Token（托肯）信号然后（向左）进入相邻的段，并使其成为缓冲器新的起始。

当前全局获胜者的父节点通过上文所说的方法计算出来之后，和一个 Evaluate（评估）信号一起向右移至相邻的段。计算出来的父节点信息与先前存储起来的获胜者信息相比较，如果相同，则不进行进一步的回溯，也被称为退出回溯过程。如果不相同，则重写全局获胜者信息，并且伴随着 Strobe 信号对局部存储器进行寻址（addr）。返回的数据位 Data 用于计算这个获胜者的父节点信息，也就是向右侧移动到先前带有 Evaluate 信号的时间段。这一过程重复至回溯

过程遍历现有的全局获胜者，然后退出。回溯过程必须在进入到第一段之前停止；为了达到这一目的，在这个位置设置了用以检测及控制的仲裁器，这也是系统中唯一的一个需要设置仲裁器的地方，幸运的是，回溯过程与第一段相遇的事情很少发生。

```
loop
  arbitrate In then
    if head then                -- head...
      data_out <- G[0];         --   output next data bit
      L := In.local_winners ||  --   update local values
      G := In.global_winner ||
      V := In.global_winner_valid;
      if V then                 --   if global winner is valid
        addrOut <- (L[G] << 5) + (G >> 1) -- start backtrace
      end; -- if V
      sync tokenOut ||          --   pass on head gtoken
      head := false             --   and clear head Boolean
    end -- if head
  | addrIn then                 --   backtrace input?
    if not head then
      if addrIn /= G or not V then -- path converged? If not...
        G := addrIn ||          --   update global winner
        V := true;
        addrOut <- (L[G] << 5) + (G >> 1) -- propagate backtrace
      end -- if ..
    end -- if not head
  | tokenIn then                --   head token arrived
      head := true              --   set head Boolean
  end -- arbitrate
end -- loop
```

图 14.12 历史单元回溯级

需要进一步注意的是，并不像传统的系统那样，本系统的路径重构只有在必要时才进行，不过无论是传统的设计还是本设计都是低功耗的。另外，HU内部的异步技术使得从PMU中所得到的获胜者信息是独立的并可以在任何路径重构行为中并行写入。使用触发器代替RAM使设计更为简单并且设计灵活性更强。使用多路回溯具有明显的优势，每一路都具有不同的段，并且可以并发运行。

14.7 结果和设计评估

上文所描述的异步Viterbi解码器系统使用了产业界合作伙伴提供的元件库设计实现，并运行于3.3 V电压。非标准元件（如Muller C门）也是由元件库中的标准单元设计构成，唯一完全自行设计的单元是仲裁器。

完成仿真后，解码器用0.35 μm的CMOS工艺进行制造。结果，对没有错误的1/2码率随机比特流进行解码的速度为45 M符号/秒，全局功耗为1333 mW。其中，PMU的功耗为1233 mW，HU只耗费了37 mW，剩余的功耗（60 mW左右）是由BMU及BMU之前的胶连逻辑（glue logic）消耗的。

解码器输入数据的错误只会导致功耗在小范围内波动。如果输入错误率逐渐增加，全局功耗将缓慢下降。从内部单元上看，下降的功耗由PMU小范围的功耗下降及HU功耗较小的增加所合成。其他解码率的功耗是1/2解码率的若干倍。例如，每4个时钟周期接收3个符号的3/4的解码率所耗费的功率是每4个时钟接收2个符号的1/2解码率的1.5倍。

异步PMU不论数据流中存在错误数量的多少，都做着基本同等量的工作。这意味着对于良好的数据流，除了好的（正确）路径上的节点之外，其他节点的权重都为6。由此，PMU基本上是处于“饱和”状态的，与路径相关的所有工作永远都不会被选择。错误使得节点权重的范围有所扩大，特别是在较高的节点之中（4, 5和6）。在错误条件下，PMU中少量节点的功耗会稍有减小。

异步PMU的功耗是非常高的，与采用传统的并行计算来完成相加-比较-选择操作的同步PMU没有什么可比性[15]。为了理解为什么会产生这样的结果，我们有必要仔细检查异步PMU使用的运算和逻辑。对于好的数据（没有错误的），63个节点的权重是6，一个节点的权重是0。对于PMU来说，就是在每个时间段内所有包含数值6的状态度量FIFO都传输至路径度量FIFO，其中包含数值6的依次从路径单元FIFO中移出，配对并传输至接收状态度量FIFO。在FIFO之中输入或者移去数值“6”将导致21个状态变化。另外，由于需要在构成每个FIFO中C门的基本单元之间进行内部传输，因此实际的传输数量要更大些（大概要多出5个）。那么，63个节点之中的每一个在每个时间段的数据路径上大概要经历650次传输。控制及数据通路上的其他开销加在一起大概要多增加30%的逻辑电路。这意味着一个节点在每个时间段要进行850个传输，而在每个时间段内PMU需要进行最多54 400次传输。

遗憾的是，FIFO的设计，尤其是路径度量FIFO的设计会导致C门在每一级的正向或反向输出具有较高的容性负载。每个时间段所进行的54 400次传输耗费1233 mW及每次5.45 × C焦耳的能量。这里的C是平均追踪加门负载电容（单位：法拉），经过测量，平均负载电容为92 fF。

相对来说，HU的低功耗是十分出色的，这是设计中的一个亮点。设计者们意识到它的功耗非常低，比任何其他的系统都低很多。HU的低功耗显示出，在没有错误的条件下，选择一条正确的通路可以通过较少的计算得到需要的输出数据流。另外，当存在很多噪声时，将激发回溯过程，在这一过程中将并行进行许多良好路径的重构过程，此时HU的功耗会有小幅度提高。由此可以看出，访问存储局部获胜者信息的触发器或者覆盖全局获胜者信息并不是消耗功率的操作。

HU的功耗相对于利用本文所描述之原则构建的基于RAM而不是触发器的同步系统来说也是非常具有可比性的。上文所介绍的RAM的局限性会引入更多的复杂性，原因是在任何时

刻只能进行一个回溯过程。因此有必要跟踪不完整的追踪深度，但当遇到新的追踪过程时这种跟踪就会被放弃，并转而重新构建已经完成了一部分的全局获胜者路径。使用触发器的异步HU与使用RAM的其他系统功耗之间的差别反映了访问局部获胜者RAM所需要消耗的功率及在回溯过程中所引入的其他相关重要计算所消耗的功率。对HU信息存储的功率高效率在某种程度上说明了它是最适合此项工作的。

14.8 小结

对于大多数的异步设计，系统设计要从重要的原则入手，并重新考虑如何实现维特比算法。这就使得PMU和HU单元中出现大量新的特点。在PMU中，由于采用串行一元运算，从而实现了用串行算法的无数据FIFO代替了传统的并行相加、比较及选择逻辑。

不过，PMU相对于传统的异步设计来说是一个有趣的特例，它的功耗并没有明显的降低。从这点可以看出，低功耗不仅仅与算法和结构有关，还必须考虑到逻辑、电路和版图等系统各个级别中的有效设计。举例来说，一个建立在相似架构原则基础上的，由低功耗逻辑器件及全定制版图构成的PMU可以运行于45 M符号/秒[15]。很显然，数据路径中采用全定制异步PMU将大规模地减少当前的功耗等级，因此对于具有较高的功率效率的系统来说，对数据路径中PMU改进的要点集中在负载上面。

全局获胜者的认定差不多是PMU设计中最重要的一个部分。这意味着幸存路径和局部获胜者可以被保存在HU之中，从而减少了大量为了输出数据而存在的存储器。触发器的使用对于功率效率也起了重要的作用。它显示出包含逻辑层次在内的各种设计层次优化对降低功耗所产生的作用。

HU也展示了异步设计的优越性，它将当前信息从很多并行运行的回溯操作中分离出来。另外，回溯的速度只与进行这种操作的逻辑相关，而与内部或者外部的系统时序无关。不过，这种解耦的、多重的回溯过程很显然会增加与同步时序环境通信的复杂程度。

14.8.1 致谢

与其他的大型工程一样，有很多人参与到本章所描述的Viterbi解码器的设计与实现工作中。感谢曼彻斯特大学计算机科学系Amulet小组的其他成员，他们是Mike Cumpstey、Steve Furber和Peter Riocreux，他们全程参加了这个项目。我也很感谢他们对于本文手稿的审阅与修订。

14.8.2 进一步阅读

关于维特比解码器的算法可以参考文献[148, 71, 70]。其他的PREST项目可以参考文献[1]。

第15章 处 理 器①

Jim D. Garside
Department of Computer Science, The University of Manchester
jgarside@cs.man.ac.uk

摘要： 计算机系统的设计变得越来越复杂，小规模的异步系统将更加具有吸引力。异步逻辑不仅要能与传统的同步逻辑相匹敌，还要显示出某些重要的优势，否则它不可能在商业领域中得到广泛的应用。

最好的办法，就是做出一个实例来证明异步系统的可行性。为了这个目标，世界上有些研究机构已经将一些实际的、大规模的异步系统组合在一起，有各种各样的组合形式，但大多研究机构选择进行微处理器的异步设计。微处理器有完善的定义说明与完备性，而且对于设计处理器的设计师而言，需要解决的都是已经熟悉的问题，因而它的设计实现将是一个较好的例证。如果异步微处理器能够顺利实现与同步设备一样的功能，那么该异步系统的可用性就得到了很好的证明。

本章描述了一些已经制造好的微处理器，并讨论了这些微处理器实现过程中的一些细节性的问题及采用的解决方案。本章素材的主要来源是实现ARM微处理器功能的Amulet系列异步微处理器，因为这些是作者本人最熟悉的器件。除此之外，本文也适当提及其他一些器件。本章后半部分对微处理器的描述加以扩展，包含存储系统、高速缓存和片上互连，具体阐述了如何构造一个完整的异步片上系统（SoC）。

关键词： 低功耗异步电路，处理器结构

15.1 Amulet微处理器简介

本章大多数例子都是基于曼彻斯特大学研发的Amulet系列微处理器。这些微处理器都是ARM结构的异步实现[65]，这样可以与相应的同步处理器直接进行比较。应该指出的是：正如其他ARM微处理器设计的那样，我们设计异步处理器的初衷是想设计出低功耗的处理器而并非高性能的处理器。

① Amulet系列微处理器受European Union Open Microprocessor systems Initiative (OMI)的资助：OMIMAP(Amulet 1), OMI/DE-ARM (Amulet 2e), OMI/ATOM (Amulet 3)。

下面对三种 Amulet 处理器和其他几种典型的微处理进行简单的描述。

15.1.1 Amulet 1 (1994)

Amulet 1[158]（见图 15.1）是一个异步设计可行性研究的产物，该设计使用的主要是基于 Sutherland 的微流水线[128]技术，尽管通信标准中使用了 2 相信号传输，在其内部使用透明锁存器（transparent latch）要比捕获 - 通过（capture-pass）锁存器（见图 2.11）好，但外用的 2 相接口与外部通用器件的衔接困难。

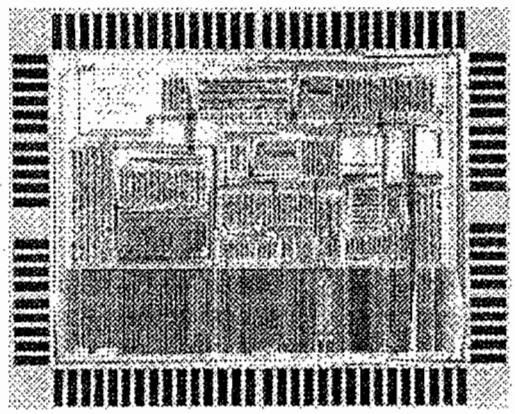


图 15.1 Amulet 1

Amulet 1 使用 1.0 μm、2 层金属工艺，含有 60 000 个晶体管，并可以运行 ARM6 指令系统（除乘 - 累加操作外）。在相同程序和同样的功耗下，Amulet 1 只能达到 ARM6 大约一半的指令处理能力（MIPS/W）。

15.1.2 Amulet 2e (1996)

Amulet 2e[44]（见图 15.2）是一种与 ARM7 兼容的处理器，具有完全相同的指令系统。除了 CPU 以外，Amulet 2e 包含一个异步的 4 KB 高速缓存和一个灵活的外部接口，该接口使得处理器与外部器件的结合更加容易。还增加了其他的一些优化（如结果的推进及分支预测）。

Amulet 2e 内部使用 4 相握手协议而不是 2 相握手协议，在这样一个 0.5 μm、3 层金属层面上包含了 450 000 个晶体管（大多用于高速缓存）。尽管 Amulet 2e 的处理速度是 Amulet 1 的 3 倍，但是其性能仍然只是相应同步处理器的一半。

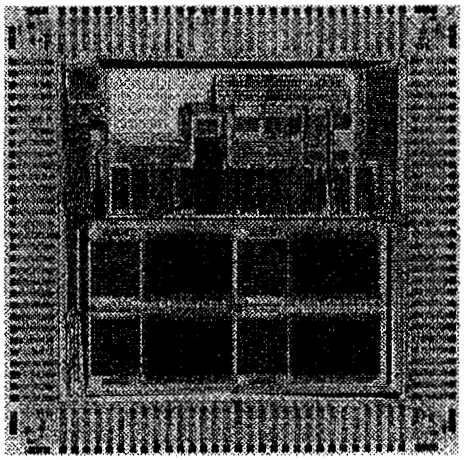


图 15.2　Amulet 2e

15.1.3　Amulet 3i (2000)

Amulet 3i[48]的设计意图不只是创建一个独立的器件，而且还可以作为一个宏单元，从而支持片上系统（System on Chip）的应用。Amulet 3i 与 ARM9 兼容，使用 0.35 μm、3 层的金属工艺，含有大约800 000个晶体管。其组件包含一个Amulet 3 CPU、8 KB的虚拟双端口RAM、16 KB的ROM、一个高效的DMA控制器和一个存储/测试接口，所有的部件都是基于MARBLE[4]片上异步总线。

Amulet 3i处理器获得了与当时相应的ARM处理器大致同等的性能，并具有相等或略高一些的功耗性能。该处理器集成了一些由 Hagenuk GmbH 设计的同步外围设备从而组成DRACO器件（DECT 无线通信控制器）（见图 15.3）。

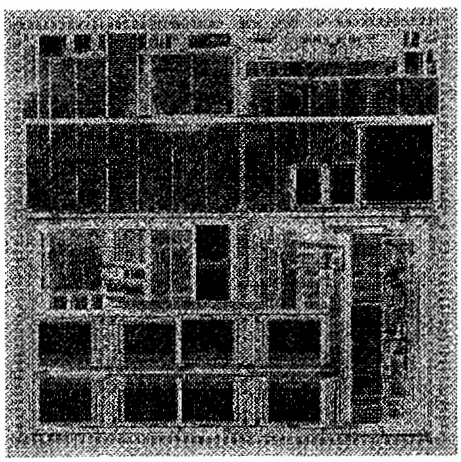


图 15.3　DRACO 器件

15.2 其他几种异步微处理器

在过去的几十年里，世界上还有其他的一些研究机构也在进行异步微处理器的研究和设计。在这一节中，将对其中几种处理器进行简单介绍。

Caltech 已经生产出两款异步微处理器：Caltech Asynchronous Microprocessor (1989)[86]是有局部异步设计的16位RISC ——首款单芯片异步微处理器；MiniMIPS[88]是具有R3000[72]结构的一款异步微处理器。这两款微处理器的设计都使用延时不敏感代码进行自定制设计，而不是Amulet系列设计中使用的捆绑数据。这种设计，迎合了对速度要求大于对低功耗要求的设计理念，从而设计出了这些高性能、高功耗的微处理器。

另一款MIPS微处理器是东京大学研发的使用R2000指令系统的TITAC-2 (1997)[130]（见图15.4）。该微处理器的设计发展了一种不同的设计风格（准延时不敏感），从TITAC-2的图片上可看出相当可观的手工布线。

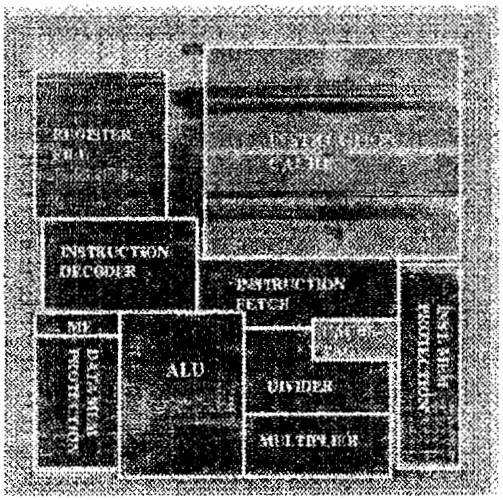


图 15.4 TITAC-2（图片蒙东京大学惠允）

由位于法国格勒诺布尔（Grenoble）的IMAG研究所研发的ASPRO-216 (1998)[117]处理器与一般意义上的微处理器略有差异，它是一个16位的信号处理器，更重要的是，该设计主要由CHP（Communicating Hardware Processes）[84, 118]自动综合生成。尽管该芯片的大部分面积被存储器覆盖，但是，该设计展现了处理器具有“无固定形状”的一面（见图15.5）。

以上介绍的所有处理器已成为设计的原形，虽然在商业上使用异步微处理器技术进行设计的进程发展缓慢，但是目前还是开发了一些具有重大意义的设计实例。

Philips公司位于荷兰埃因霍温（Eindhoven）的实验室开发出“Tangram”[135]电路综合系统，主要目的是用其开发较低性能、超低功耗的电路系统。该电路综合系统已用来实现异步

80C51 (1998)[144]——目前已经用于商业寻呼机，其低功耗和低EMI（电磁干扰）属性尤为突出，在智能卡应用方面也有可能使用该系统（见第13章）。

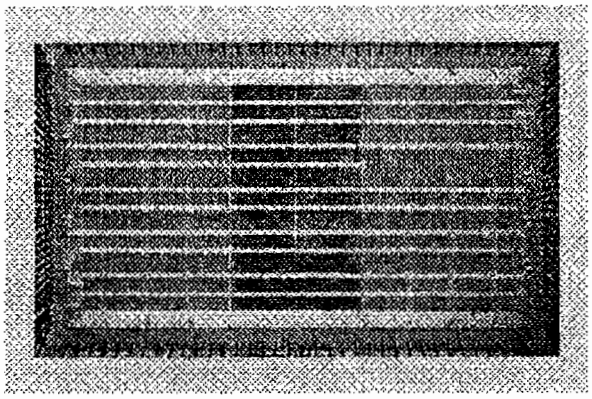


图15.5 ASPRO-216（图片蒙IMAG研究所CIS研究组TIMA实验室惠允）

尽管不是严格意义上的微处理器，Sharp公司的DDMP（Data-Driven Media Processor）(1997)[131]也具有其自身的优点。为了适应多媒体应用，DDMP提供了大量的并行处理单元，这些处理单元根据不同指令的要求，处于使用或者闲置状态。正是由于异步技术便捷的电源管理特点，才使其具有如此巨大的吸引力。

最后，DRACO处理器（见图15.3）是专门为商业交易而设计的（尽管因公司重组等原因未能上市）。在无线电广播基站使用这种处理器的主要原因也在于其异步逻辑的低EMI特性。

15.3 处理器作为设计实例的缘由

为什么要设计一个异步处理器呢？根据上述商业中使用的异步设计例证，其中的部分答案必然是异步微处理器具有低功耗、低EMI等优点。但是为什么偏偏是处理器才能够很好地表现这些特性？

在很多情况下并非这样，异步的优势可以更好地体现在具有规则结构并使用微流水线操作的应用上。目前一些信号处理及网络交换的应用中都有这些特性。然而，进行微处理器的设计还是有诸多的吸引力。首先，微处理器必有完善的定义，并具有完备性。定义一种微处理器应该做什么以及验证它是否满足这样的设计规范是相对容易实现的。其次，设计者必须要面对并且解决在处理器的设计实现过程中出现的未知问题。最后，通常可将设计的异步处理器与同步设备进行比较，对设计结果进行量化和评估。当然，微处理器设计占据着一个快速发展、更新，并具有竞争性的市场，即使是在成熟的技术上，竞争也是相当激烈的。

15.4 处理器实现技术

15.4.1 采用流水线的处理器

流水线[56]操作是改善微处理器性能的一种行之有效的方法。简言之，流水线操作方法可以将某个费时的操作划分为多个彼此之间可以交迭进行的快速操作。如果执行一个理想的5级流水线操作，那么在附加较少的硬件资源（用于流水线锁存器）的基础上可以将执行任务的速度提高约5倍。

典型的同步流水线操作是将完整的逻辑电路划分为多个执行时间相等的电路片段。如果出现不均等划分的话，运行最慢的流水线片段决定了整个系统的时钟周期，因而由于时钟的原因，运行速度较快的逻辑使部分会受其拖累而导致速度减慢（见图15.6）。如果某一段特定的流水线的执行时间是变化的或与数据相关，等时长划分就显得十分困难。数据相关性是微处理器的共性，例如，处理器的ALU单元中，“move”操作较相加操作执行速度快，这是由于前一操作不需要进行进位的传送。另一更具说服力的例子是微处理器的内存读写操作，此时有缓存操作比无缓存操作执行速度快。

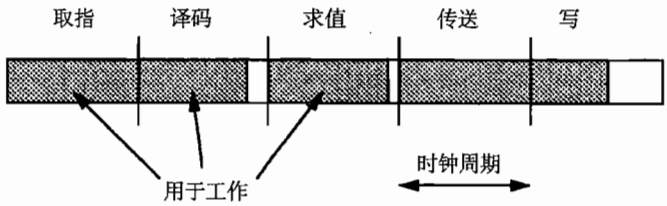


图15.6 同步流水线用法

通常，在这些情况下，时钟必须能够满足最慢的周期要求。这就导致了整个系统时钟频率下降，或者要求设计者投入大量的硬件来加速这些最坏情况操作（worst-case operation），例如，增加快速进位链。如果大部分的操作运行较慢，那么增加硬件进行加速是行之有效的，否则，这将是一种不经济的做法。

同步设计者可采用的另一个解决方案是使运行较慢的操作占据多个时钟周期。在缓存缺失或者处理器必须等待片外存储器读取数据的情况下，这种解决方法有非常明显的优势。

异步流水线操作在概念上很容易理解。它不仅是局部控制，而且能通过可变的延时来适应各段流水线。也就是说，使用流水线操作，通过动态改变流水线段的延时，在不需要增加额外的硬件，且不会对整体处理时间造成重大影响的情况下可改善少数最坏情况操作的运行速度。该特性结合自适应操作使得流水线操作具有相当大的“弹性”。在这种流水线中，无缓存操作仍然只占用一个时钟周期，尽管它是一个特别长的周期。

另一个明显的实例存在于ARM指令系统[65]中，在该指令系统中，数据处理和寻址操作将在ALU操作之前进行操作数的移位。在早期的ARM器件中，该过程包含于一个单独的流水线段中，并且被更慢的存储器存取时间掩盖。为了避免性能的降低，目前大多同步设计都包含以下选项：

- 附加的移位流水线段（增加等待时间）。
- 当要求执行某个移位操作时，延迟一个额外的时钟（增加了复杂度）。

异步流水线在需要时延长执行周期比较容易。由于这种额外时延不会比ALU的延迟长，所以异步流水线较其他同步方法更具灵活性，并且由于移位/运算操作相当少，其对整体处理时间的影响也小。

“平均”性能 这种异步流水线的“弹性”引出一种说法：异步流水线使得每一个流水线段以平均时间而不是以最坏情况的时间来处理信息。这种说法只在相关的单元持续地处于工作的情况下才是正确的，而在一般情况下是不正确的。在有些情况下，处理单元在其操作数到来之前就已经准备就绪了，而在另一些情况下，该单元在后级流水线能接收它的输出之前就已经运行完成了，以上这些情况下都存在强制性的空闲时间。通过在处理单元之间添加快速缓冲可以减少空闲时间，使得某些操作在规定时间内完成，而真正的平均性能只能通过无限容量的缓冲器来实现。任何一个缓冲器的添加都会增加流水线的等待时间，故在使用时要加以慎重考虑。

实际上，异步流水线与同步流水线具有类似的结构，所不同的是异步流水线在不需要流水线中断或者添加额外的硬件的情况下可以执行某些偶然性的、耗时长的操作。它具有同步流水线所没有的一个优点：流水线等待时间可以缩短——尤其在填进一个“空”流水线的情况下，因为此时第一条指令不会因每一级流水线的时钟而发生延时（见图15.6）。接下来需要解决的是如何划分系统，以及由内部操作数的相关性引发的问题。

15.4.2 异步流水线结构

异步流水线设计出来后，将其加入到设计中是相当容易的事情。然而，不加选择地使用流水线操作也会带来问题，至少在通用处理器中是这样的。出现这种情况主要是源于操作数的相关性[56]，某处的一个操作必须将前面指令执行结果作为操作数，如果许多操作同时在流水线中执行，那么极有可能任意一个新指令的执行都要被迫等待其他一些指令完成。在异步环境下解决相关性问题是一项相当复杂的任务，我们将在后续章节中进行讨论。

因此，一个不太明显的现象是：流水线的延时有所增加。这种延迟不只是由于增加了锁存器所导致的，还因为在某些情况下，流水线需要将其中数据全部排空，然后再重新装入。例如，含有一个执行预取功能的快速FIFO缓冲器的处理器流水线，该流水线可以减少在预取（如高速缓存不足）或执行（如乘法操作）等某些偶然发生的慢周期操作时流水线停顿（stall）的次

数，因而最初我们认为这是一个较好的设计。然而，在通用处理器的使用中，流水线的这个特征被掩盖了，因为此时流水线的一次停顿就会让指令充满预取指令缓冲器，然而紧随其后某指令要求将整个流水线排空。这就是Amulet 1的设计中存在的结构上的一个失误。因为Amulet 1具有4级预取缓冲器，所以在性能上存在显而易见的缺陷，当然在其他较少使用流水线的应用中，增加缓存可以带来很多好处。经验表明，对于一个通用的CPU而言，最传统的设计方法就是最好的实现方法，如基于已知周期（如缓存读取周期）设计出相当协调的流水线结构。

异步流水线结构的显著优势在于可以对流水线流程进行局部控制。回想一下RISC结构中的弊病——多周期指令，在同步环境下这样的多周期操作必然会导致流水线中很大一部分操作的延迟执行，并需要大量控制网络对其实行控制；在异步环境下，这样的多周期操作不再需要考虑其他部分的操作，而仅表现为一段较长时间的延时，但如果其他的处理单元要求与该单元交互作用，可能会引起一次流水线停顿。

在ARM处理器中，该局部控制机制在有些时候是十分有用的，那就是多寄存器的装载和存储操作（LDM/STM），它可使16个通用寄存器中的任意一个或多个与存储器进行数据传输。Amulet 3通过一次单输入握手，在执行级产生几个本地指令包来实现这样的数据传输。此时，预取操作将锁存器填满后停顿，这是流水线操作必然的结果。

这种局部控制特性也体现在Amulet 3设计中的其他部分（见图15.7），尤其是其中的Thumb译码器。该译码器可接收32位数据包，可容纳一条ARM指令或者2条经压缩过的16位Thumb指令。在后一种情况下，对每一个输入信号会产生2路输出握手信号。由于译码器减少了指令读取存储器的周期数，并使用低速存储器，利用了所有可用带宽，从而使该异步译码器优于同步Thumb译码器，可以一次性读取16位的指令，存储系统的功耗也相应地减少了。

局部控制在执行诸如“CMP”（比较操作）等不需要遍历全部流水线的指令时也是同样可行的，此时通过不产生输出握手信号的方法删除一个流水线数据包如同产生额外的数据包一样简单。在Amulet 3中，比较操作只影响存在于“执行”级中的处理器标志位，并使其不再在流水线上继续执行下去。

局部控制最终的益处在于流水线操作可由任意处于活跃状态的流水线段进行调整。Amulet 2和Amulet 3都已将“halt”指令（挂起指令，使CPU处于挂起状态直到有中断到来）加入到ARM指令集，该指令本身由一分支实现，因此可在流水线的任意处检测和执行，并对处理器停顿产生相同的影响。实际上，Amulet 2在其执行单元中实现暂停，而Amulet 3是通过暂停指令预取来实现的，但是两者产生的效果是一样的，挂起异步处理器（或其中的一部分）相当于停止同步处理器的时钟，在一个CMOS器件中，这种挂起可将功耗降到接近于0。因此，这一操作使得系统功耗管理相当容易，甚至在某些情况下，近乎是自动化的管理。

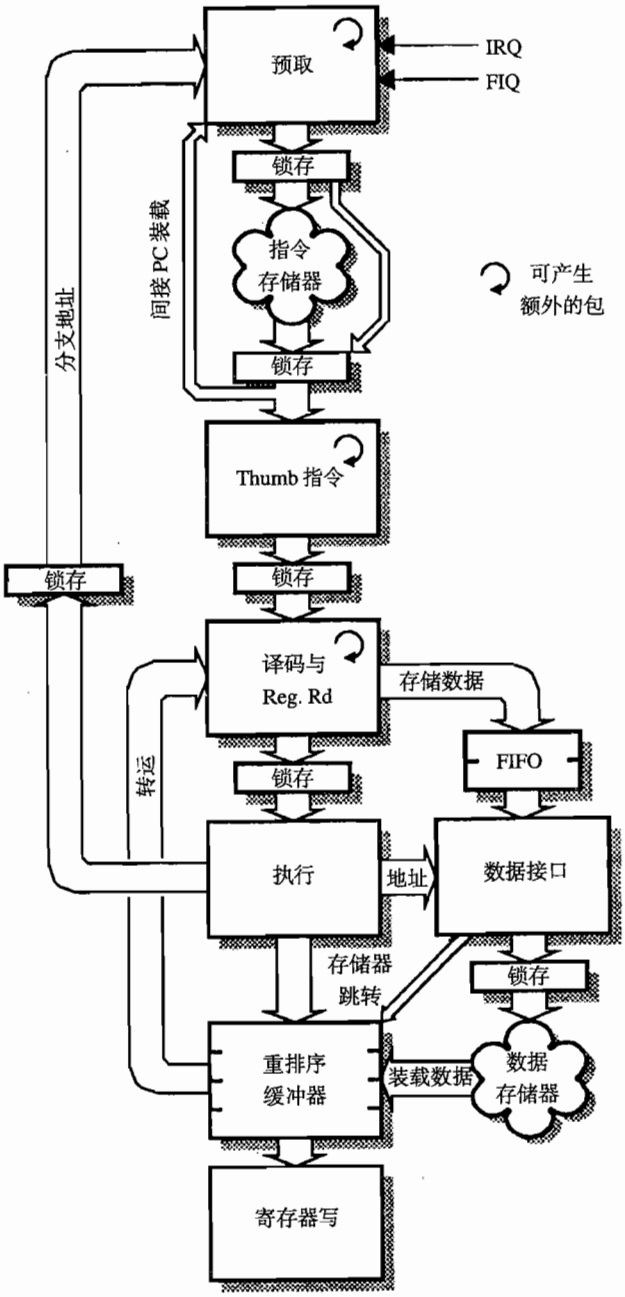


图 15.7　Amulet 3 结构

15.4.3 确定性和非确定性

在检验这些可用于异步处理器的结构化设计方法之前，考虑一下这些设计所采用的设计原则，其中最重要的一点是非确定性（non-determinism），因为异步处理器与同步处理器不同，即使其表现为非确定性，仍可正常工作。

同步系统分析和设计的主要优势在于下一个周期的状态完全由当前状态所决定。在异步系统中也可以是这样的，但由于异步系统时序自主，所以上述的实现方式并非是异步系统的唯一选择。在小型的异步状态机中，不同次序的内部转换可以获得同样的性能（例如，一个Muller C单元的输入次序是可以任意改变的），并且，在宏观层面上也是成立的。这里选取的第一个实例是介绍处理器的预取特性，选择该实例是由于不同的设计方案具有不同的设计原则。

目前所有的Amulet处理器已具有非确定性的预取深度（prefetch depth），该特性是通过允许预取单元自由地运行来实现的，一般仅受限于指令存储器读取指令的速度。为了通过分支实现“halt”指令，预取进程将被中断，并插入新的程序计数器的值。这是发生在非确定点并需要仲裁的一种异步过程。

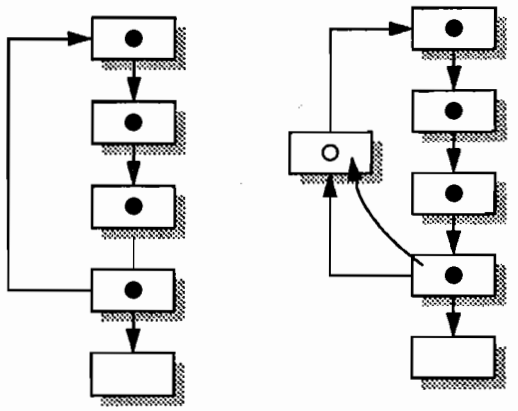
另一种方法是预取固定数量的指令，在ASPRO-216处理器[117]中就使用了这种方法。它的实现过程是每条指令结束时提示预取单元开始新的取操作，且无论是否要求有一路分支，都会有信号通知。实际上此时的处理流水线变成了一个环，在该环中，大量的托肯（token）循环并被重复利用（参考3.8.2节）。

采用确定性的预取简化了某些任务的执行，特别是在推测性预取（speculative）和分支延迟时段的处理上。由于有可能确切地判断出分支后需要预取多少条指令，因此可使用一个简单的计算进程对其进行处理和舍弃。然而，要保证托肯进行有序的环形流动，对流水线的灵活性要添加额外的约束条件，在某些情况下，可能要以牺牲系统性能为代价。

尽管当流水线执行任务时，指令预取存在上限，但是对于具有非确定性预取深度的流水线来说，推测性地预取零条或者多条指令也是有可能的。在没有延时时段（delay slot）（如ARM）的结构中，下限已经不再是问题，但是除指令计数之外的一些方法必须用来说明预取指令流已经发生了变化。Amulet处理器通过将预取指令流“着色”（colouring）来实现。例如，假设处理器预取一串“red”（红色）指令，执行这些指令，到达分支处，该分支要求执行单元请求指令预取从一个新的地址处开始，并标以一种不同的颜色（如绿色）。接下来红色标识的指令必然会被舍弃，第一个绿色指令就成为下一个要完成的操作。随后的绿色分支可能会导致另一种颜色的变化，因为前面所有的红色操作都已流经这一点，此时就有可能使之再转换为红色，所以仅需要两种颜色（一个颜色位）。

对于非确定性预取，还存在另一个不太明显的问题，如果该问题在设计中没有加以考虑，将会导致死锁。在这样的结构中，分支的执行是使用一个仲裁器将托肯插入到处理器流水线

中。如果此时流水线已满，并且没有任何措施防止这种情况的发生，仲裁器就无法确认这种操作，从而流水线发生死锁［如图15.8(a)所示］。当然，如果流水线未满，由分支操作引起的死锁并不都是不可避免的，但是每一次分支操作都有可能引发死锁。



(a)死锁　　(b)外加锁存避免死锁

图15.8 分支死锁和避免

使用一个外加锁存器就可以轻松解决这个问题，该锁存器一般情况下为空，并可将分支操作从主流水线流程上分离出来［如图15.8(b)所示］，使其离开常规操作流程。只有当前分支操作已经被接受，目标操作数已获取并译码，下一个分支才能运行，这里始终使用这样一个锁存器预防死锁。

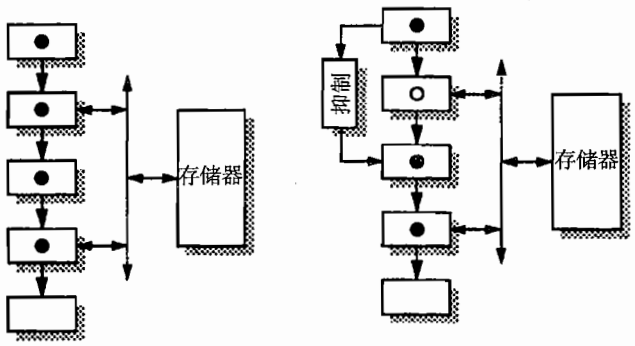
实际上这类问题是比较常见的，在Amulet 1设计中类似的问题变得十分明显。在该设计中，对总线数据的读取和存储过程中，指令获取的完成是非确定性的。当处理流水线满时，指令的获取占用了存储总线，却因没有锁存器来存储结果而不能完成指令取操作。如果数据传输过程中产生延迟，也将会导致流水线阻塞［如图15.9(a)所示］。要解决这个问题，可以通过对指令读取进行抑制，直至可以继续传输数据时，读取指令才可以再次获得总线［如图15.9(b)所示］。由于数据的传输一旦开始就不能停止，因而反过来的问题就不会出现，因此仅指令预取过程需要抑制。

尽管存储系统仍然会出现类似的死锁情况，但具有分离的指令总线和数据总线的Amulet 3处理器并未出现上述问题，这将在15.5节中详述。

如果选择了某一种确定性异步设计方案，那么该方案可以通过在每一条指令中添加一个数据传递相来实现。异步存取的另一个优点是对于“无用数据”的读取非常快，但是在处理器流水线的适应性能上也要付出一定的代价。

逆向流水线处理器（Counterflow Pipeline Processor） 尽管大多数异步微处理器设计都有通用的结构（除了缺少同步的时钟），但具有完全不同的处理器结构的设计也是可以实现的，目

前已经有多款这类处理器投入研究，如文献[126]所提的新型、具有高度非确定性的逆向流水线处理器（CFPP）设计。在该处理器中，指令沿着有处理单元的流水线顺畅地流向寄存器组，而与此同时，操作数也能畅通无阻地流向它们。该设计试图在等待指令间有依赖关系的操作数时减少处理器的停顿次数。同时在对指令进行求值并传送结果完成指令时可以使其结果返回，以备后续操作之用。



(a) 死锁　　(b) 通过抑制预取避免死锁

图 15.9　总线竞争死锁及其避免

尽管CFPP需要耗费大量的总线资源，但该设计在功能单元的数量以及组合上具有相当大的灵活性（见图15.10）。功能单元可以是全部功能（或部分功能）的ALU、存储器访问单元与乘法器等，其中唯一的规则是任何一个操作必须由某一可用功能单元执行。因而仅当某一未完成的操作到达最后一个合适的功能单元处仍无法获得所需的全部操作数时，才需要实行停顿操作。

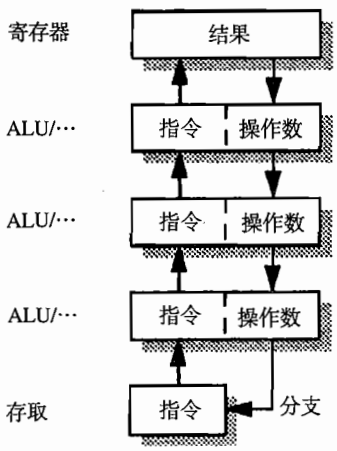


图 15.10　逆向流水线处理器原理

为了确保指令不会因丢失操作数产生其他的执行结果，在流水线的每一级都需要一定程度的同步。在流水线结构中，指令没有预先规定流动方向，因而流水线的每一级都需要有一个仲裁器，通过该仲裁器在不检查具体内容的情况下确保两个数据包没有互相交叉。指令和数据的每一次移动都需要进行优先级判断，因此，对于这样一个追求性能的结构而言，具有快捷、高效的仲裁器至关重要。

当然，由于分支的存在（这些分支用于向后传递至指令提取级），增加流水线深度来适应许多功能单元会加重系统的负荷。因而，良好的分支预测也是很重要的。

仲裁及死锁 设计一个具有确定的操作时序、与相应的同步设计类似的（例如，除了中断以外，其他操作都具有时序确定性）异步处理器也是有可能的。同样地，设计一个具有高度非确定性的处理器也是可行的。

每一种设计方案其自身都有优点和不足之处，强制性的同步会导致系统性能的降低。例如，直到被告知一条指令不需要数据传输的存储器时，存储器与外设的接口才开始预取操作，从而使可用的带宽也减小。另一方面，通过减小系统的可达状态空间使系统行为的可预测性增强，测试过程从而得到简化。在某些特定的场合，设计者似乎必须选择非确定性的方案。然而，需要注意的是，每个非确定性的设计都需要有仲裁器（理论上可以要求一个无限长的解决时间）与之相配合，并可能会引入潜在的“死锁”，我们必须要注意到这点，并尽量避免其发生。

在一般情况下，避免死锁的方法是仔细验证共享资源（如总线）经仲裁后的所有可能情况，并保证无论系统其他部分的状态如何，只有当共享资源确保被释放时才能够访问其他单元。每个仲裁器增加了处理器的可达状态数，使设计更加困难，但是也加大了系统设计上的灵活性。如果能谨慎小心地利用非确定性原则，对设计而言是有益的。

15.4.4 依存性

当处理器的流水线达到一定深度时，有必要插入互锁设备（interlock）以满足指令间的依赖关系，并确保程序正确地执行。即使像MIPS R3000这样具有“没有互锁流水线级”的微处理器也在某种程度上存在着互锁[72]，此时编程器/编译器可使用时钟周期计数来保证操作的正确性，不过这只是权宜之计，在异步环境下是不适用的。在ARM结构中也有类似的限制。

PC流水线 ARM与MIPS使用时钟的方式显然不同，但在它们的结构中有一点是类似的。通用寄存器组中的程序计数器（PC）可为编程器服务，该程序计数器读时，其值是当前指令与两指令相加的地址，这是最初ARM实现得到的历史结果。在ARM结构中，生成指令地址和执行操作之间存在2个时钟周期。即使是在指令的预取和执行相互独立的异步处理器中，也必须保持这种兼容性。

由于在Amulet处理器中，PC的使用可能使数据的生成和后续操作不同步，故必须找到一种方法来传递数据。为此，所有的Amulet处理器对每一个提取的指令的PC进行了备份（见图15.11）。然后这些备份数据伴随着指令一起在流水线中流动，在任何需要PC处都可以随时

进行读取。PC与操作数或者地址一样需要直接隐含在分支指令中，在发生中断或者存储器异常时进行地址的返回。不同的Amulet核具有不同的确定数值（例如PC+8）。使用PC流水线是为了减少后续计算的开销，然而在Amulet 3中，使用的PC没有进行任何固定的预修正，这样就可以通过一个简单的递增来计算PC+2、PC+4、PC+8中任一所需的值。

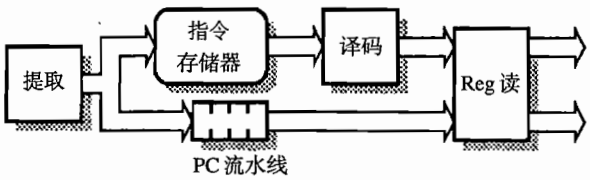


图 15.11 PC流水线

值得注意的是，PC的值并不需要直接与指令捆绑。PC流水线可以与具有不同深度的指令提取的异步流水线分开，并维持着“一个输入，一个输出”的对应关系。这是Amulet 3设计具有的一个特征，以此来抑制指令预取单元，避免因指令提取造成的死锁。关于该机理的详细内容将在15.5.2节中加以描述。

寄存器的依存性 关于寄存器的依存问题，重点在于先写后读（read-after-write）的依赖关系。其中有一种情况存在于某一代码段中，例如：

```
LDR  R1, [R0]     ; Load ...
ADD  R2, R1, R3   ; ... and read
```

在该例子中，必须先把值加载到R1中，R1才能为下一条指令所使用。但是一旦执行路径是流水线形式，这种执行顺序就不能得到保证，并且这种不确定性在异步器件中会增加，因为此时数据加载可能会发生在不确定的时刻。

以下给出3种方法来解决这种寄存器的依存性：

- 不使用流水线结构
- 锁定
- 前向传输（forward）

第一种解决方法是最早的同步ARM设计所采用的方法。该方法是使读取寄存器操作数、执行操作并将结果写入寄存器等操作在一个单周期中完成，从而确保后面的操作都有一个更新后的寄存器数据可使用。该方法虽然简单，但使得赋值周期很长，这在高性能的处理器中是不可行的。

第二种解决方法是用锁定的方法。该方法在操作数寄存器还处于无效时，通过将指令延时，而使指令的执行具有可选择性的流水线操作。实现时，当更改寄存器内容的指令被译码时，对其设置“lock”（锁定）标志；当寄存器中有数据写入时，再将该标志清除。后面的指令在

执行时先检测它的操作数是否具有“lock”标志，有则将该指令延时，直至所有的锁定标志清除。该方法同样适用于异步设计，因为异步设计本身的停止指令就会引发等待延时，并且这个过程不需要仲裁。

实际上，向同一寄存器写多个结果是十分方便的，部分原因在于ARM中大量使用了条件指令，例如：

```
CMP    R1, R2  ; Set flags
MOVNE  R0, #-1 ; If R1 ? R2
MOVEQ  R0, #1  ; If R1 = R2
```

在这种情况下，只有一个锁定标志（该例中指的是R0中的锁定标志）是不够的，还需要其他形式的信号量。假设测试和增加过程是互斥的，这样的信号量操作在异步环境下也是相当容易实现的。因此在一条指令发出时需要做到：

- 尝试读取该指令的操作数，并一直等到所有标志被解锁。
- 通过增加信号量锁住该指令的目的寄存器。

当有结果返回时信号量减少，该操作可将信号量减少，直至为0，从而释放等待的指令，这种情况随时发生。

上述例子给出了异步系统中存在的另一个潜在问题：两个“MOV”操作互斥，只有其中一个可以执行。由于在指令发出时并不清楚这种情况，两个MOV操作都会增加信号量，因而，接下来两者必须都将其减少，否则R0将永远被锁住。在一般情况下，如果一个冒险性的操作开始执行，那么该操作必须完成，这样一个“写”操作也总是发生，尽管有时寄存器内容并未发生变化，仅仅是执行解锁。

在Amulet 1和Amulet 2中，信号量的设计和实现以“锁定FIFO”形式存在，在这些处理器中，信号量也保持着目的寄存器的地址，以避免和指令一起执行。由于指令大多按顺序执行，故结果和目标地址能够在写时刻进行配对。

锁定FIFO以异步流水线的形式存在，如图15.12所示。由于每一个锁存（水平的）的组成单元是透明的锁存器，入口被一个接一个地复制，从而确保了OR门的无误输出。这些输出信号可以用来停止特定寄存器中的任何读操作，直至写操作完成时才可以读。唯一可能存在的问题是：当尝试进行读操作时寄存器仍然被锁。可以通过指令译码器的先读后锁来避免该问题的发生。

解锁可以在任意时刻可靠地发生，并与其他寄存器的读或锁定操作有关，以译码形式存在的目标地址此时已存在于数据FIFO的底端。

重排序缓冲器 尽管锁定FIFO可以顺利地工作，但是因为它要求指令按序完成，并将指令延时直至获得操作数才加以运行，从而使工作效率低下。所以锁定FIFO是能保证功能得以完成的一种有效、廉价的实现方式，但是它对于高性能的结构并不理想。

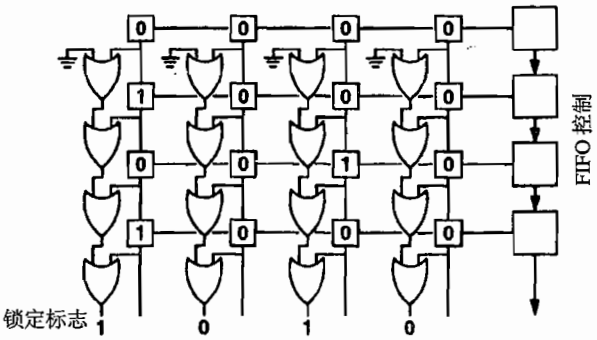


图 15.12 带锁的 FIFO 结构

在 Amulet 3 中，寄存器的依赖性是通过一个异步重排序缓冲器加以解决的[66, 51]。其目的主要是为了简化插入页故障机制，它允许指令以任意次序执行，并且产生的结果可以在任意时刻前向传输。因此，对于一个完全无序的异步处理器而言，该缓冲器的使用是解决依赖问题的一个十分重要的措施。

重排序缓冲器位于各种数据处理单元和寄存器组之间（见图 15.13），并跨越多个时间域。指令首先在译码时受重排序缓冲器控制，此时对指令进行操作数的查找和传输，并为所有的结果分配存储空间。随后可将指令分开执行（原则上在任意时段都可以分开执行），各部分的执行结果可以相互独立、互不相干地到达。在操作数到达和对其进行重写的过程中，可以对操作数进行任意次的前向传输，最后对得到的结果进行有序地回写，存入寄存器组，此时重排序缓冲段得以释放。

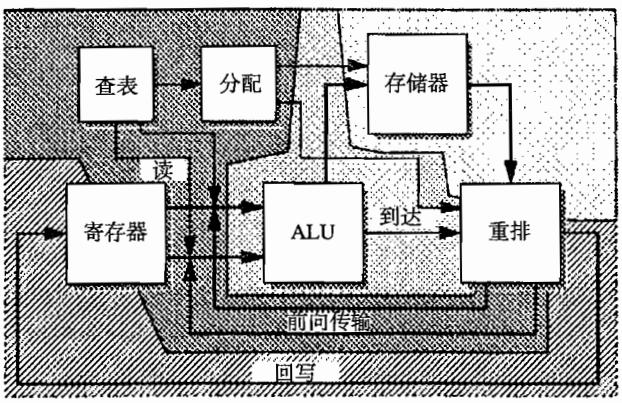


图 15.13 Amulet 3 结构中重排序缓冲的位置

在译码阶段，操作数寄存器的号码与一个较小的内容寻址存储器（CAM）相比较，以确定哪个重排序缓冲段可能包含适合前向传输的值。由于该列表总是结束于寄存器自身，因此，这个值总是可以取得。一旦这种映射确定，重排序缓冲段就可安全地重新分配。

分配进程循环地将特殊的寄存器地址与每个重排序缓冲入口联系起来，并且指令包前向传输的正好是重排序缓冲段中的数据。考虑以下代码段：

```
LDR   R0, [R2]     ;
MOV   R4, #17      ;
LDR   R7, [R0+4]!  ;
ADD   R7, R7, R4   ;
CMP   R3, R4       ;
ADDNE R7, R7, R6   ;
SUB   R1, R7, R0   ;
```

假设重排序缓冲器具有4个入口，并且下一个空闲的入口是0，重排序缓冲器的分配情况已在表15.1中列出。斜体表示的入口是最新已分配的入口。

表15.1 重排序缓冲器分配

指令		段0	段1	段2	段3
LDR	R0, [R2]	*R0*	?	?	?
MOV	R4, #17	R0	*R4*	?	?
LDR	R7, [R0+4]!	R0	R4	R0	*R7*
ADD	R7, R7, R4	*R7*	R4	R0	R7
CMP	R3, R4	*R7*	R4	R0	R7
ADDNE	R7, R7, R6	R7	*R7*	R0	R7
SUB	R1, R7, R0	R7	R7	*R1*	R7

注意：

- 第二个加载操作（LDR）使用了ARM基数回写模式，因而需要2个目的地址。
- 比较指令不需要寄存器目的地址。
- 在重排序缓冲器中，相同的寄存器地址可以出现多次。
- 即使此时指令是条件指令（ADDNE），且可能不产生有效的结果，还是会对该操作分配执行的段。

在指令执行之前，指令译码器仍存有重排序缓冲器的映射。与重分配一起，其执行流程如下，以最后一条指令为例。

- 检查相应的寄存器的位置，此时这些寄存器可能正处于重分配的过程中，但是还没有数据写入，因为此时指令还没有开始执行。
  对R7而言，可选择：段1，段0，段3，寄存器组。
  对R0而言，可选择：段2，寄存器组。
- 试着从以上列表中的每一个位置处读取数据，直到取得有效数据为止。

值得注意的是，需要给出这些可能的情况，因为分配的段中不必包含有效的数据。在ARM中，指令条件代码检测出错是引起这种无效性最明显的原因（例如，段1中R7的取值可能是无效的），但是其他条件（诸如前向分支中）也能导致指令被放弃。

指令译码阶段的控制流程如图15.14所示。需要注意的是，前向传输时间的变化取决于外部因素，而段分配的时间取决于要求的段数（0～2个），并且偶尔还要等待某一个段到可用为止。重排序缓冲段的分配是连续的，甚至由一条指令完成，这大大简化了异步实现。对更复杂的指令会有较小的性能影响，但是这类复杂的指令相当少。

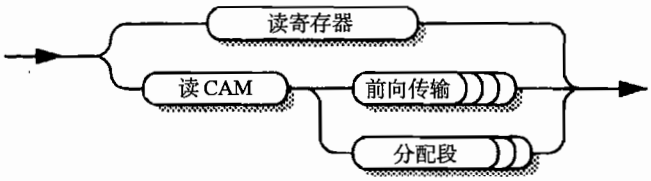


图15.14 Amulet 3译码单元的寄存器进程

分配完成之后，接下来带有重排序缓冲段编号的指令包进入下一级操作。尽管Amulet 3只使用有序的单指令，但ARM指令集通过内部执行单元和外部数据接口可以有效允许两个半独立操作（见图15.15）。每种情形在任意时刻都可能产生执行结果，因而各自拥有接到重排序缓冲器的接口。同时这些输入是异步的，它们被确保放置到不同的段中，因而相互之间是独立的。

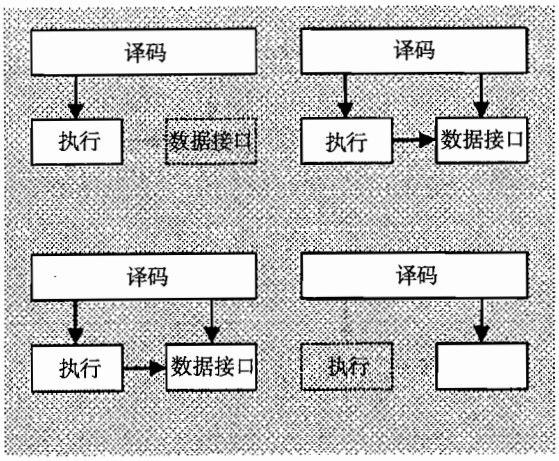


图15.15 SUB指令执行路径

回写进程只是将执行的结果复制并送回寄存器组。当回写进程到达时，每个结果都“填入”到一个重排序缓冲器段，回写进程等待某一特定的段就绪，将结果复制，进入下一个段。在任意顺序下，段都能准备就绪。

传送机制加于回写进程之上，等待结果就绪并复制送回至译码级，该进程可以在回写进程的任意一个阶段执行。实际上这些进程是异步和并发的，关键是两者都采用非破坏性复制数据，从而使得这些数据能为它用。重排序缓冲器中的结果在被改写前一直保存。

如果上述进程需要一个还没有到达的数据，那么就必须等待。为了使得进程之间互不影响，我们使用2个单独的标志来说明段中数据的存在情况。一个标志是“full”，说明此时的段是满的，当回写进程将数据送入寄存器组时清除该标识，该标志处于图15.16所示控制电路的中心位置（将在后面加以简述）。另一个标志是“Fcol”，仅用来表示结果的到达。由于该标志的状态对于每一个通过循环重排序缓冲器的数据都会发生改变，因而可以通过发送请求对结果进行验证，不需要改变标志状态即可检查结果是否已经到达（见表15.2）。一个数据可能传送0次或多次，具体的传输次数在发送时并不知道。

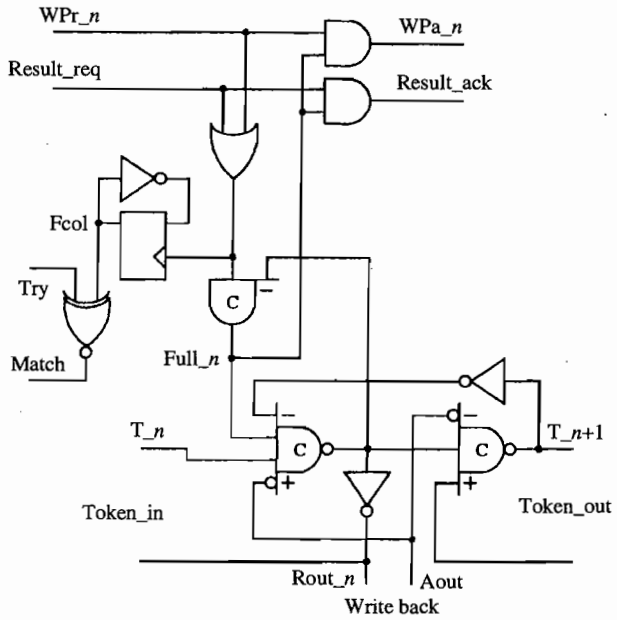


图 15.16 重排序缓冲数据回写电路

表 15.2 重排序缓冲器中反映结果到达的“Fcol”状态

结果次数	0	1	2	3
0	0→1	0	0	0
1	1	0→1	0	0
2	1	1	0→1	0
3	1	1	1	0→1
4	1→0	1	1	1
5	0	1→0	1	1
6	0	0	1→/	ô
7	0	0	0	1→0
8	0→1	0	0	0

最后返回到重排序缓冲器的结果可能是无效的（例如，因条件代码出错导致结果的无效性）。每一个段都有一个相应的标志，该标志的存在可以防止数据被写入寄存器组（full标志仍然被清除）或者向前传输。在后一种情况下，接下来的传输取的可能不是最新执行结果，最终使用的是寄存器默认值。

图15.16所示的回写电路是异步控制电路的一个实例，其执行过程如下。

- 结果（左上角的Result_req）的到达对“Full”置位，并给出相应的应答信号（右上角的Result_ack），该请求来自于许多互斥数据源之一。
  Result_req信号同样锁定了传输标志（Fcol），该标识允许接下来的任意一条指令使用这一结果。
  注意，输入可以来自任意互不相干的通道，虽然这里只给出了其中2个，但是其他情况也是可能的。
- 输入请求可以在任何时刻被移除，使得段标志为“Full”。
- 当轮到该段复制数据时，收到一个托肯（左下角Token_in），该托肯发出输出请求（底部中间Rout_*n*），如果先收到托肯，则电路等待执行结果的到达，托肯被确认。
  该进程允许“Full”位复位，如果还没有进行复位的话，等待输入请求到达。
- 复制完成之后，电路获得应答，以完成托肯的输入握手，并将托肯送至下一段，与右边电路相类似。下一段只有在4个阶段的输出应答完成之后才允许输出。

这些段连接于一个环中，并进行复位，从而确保第一级只存在一个托肯输出，随后该托肯自由运行，并当其获得结果时依次将各段清空。

Amulet 3使用的这种实验方式（一般是非系统性的）意味着将对原始状态机进行定义和改进，其结构与图15.16所示的电路类似，并只能用一种更为规范的方法来进行分析，即由Petrify完成分析（Petrify已在6.7节中介绍）。

首先引起这些问题的因素取决于系统内部的选择。输出通道调用寄存器组，这有利于将输出应答发送至所有复制的电路，并使用唯一活动的请求激活相应的区域。这在单个子电路中建模是很难实现的，Petrify的开发者（Alex Yakovlev）的建议，在整个系统上建模比在单个子电路中更容易实现。相应的STG（见图15.17）很清楚地说明了信号传递循环控制的4个子电路的实现。处理器其他部分被划分为4个小环，当一路输出握手完成时，这些环可以在任意时刻将电路再次复位到“Full”状态。

使用Petrify分析验证电路的操作，并移除分支电路中的多余晶体管。

在这里描述的异步进程存在两种风险。第一种风险是没有一种局部的方法来避免段在清空之前被覆盖。在Amulet 3中，当段被复制进程释放之后才对其进行分配，以此来避免这种风险的发生。实际上空闲的段数量是一定的，当对段加以分配时，其数量减少；当再次释放段时，其数量又会增加。由于段的分配和释放是循环和有序的，因此这里没有必要对每一个段进

行验证，只需存在托肯就可以了。这时“压制”的实现是通过一个简单的、无数据的FIFO，该FIFO同时还作为非同步进程之间的异步缓冲。FIFO用于一个具有4个重排序缓冲的系统中，如图15.18所示，此时的状态是：已经产生了一个执行结果，但是还没有将其送入寄存器组；另两个结果正在生成，并且译码器能够发出另一条指令产生单一结果。

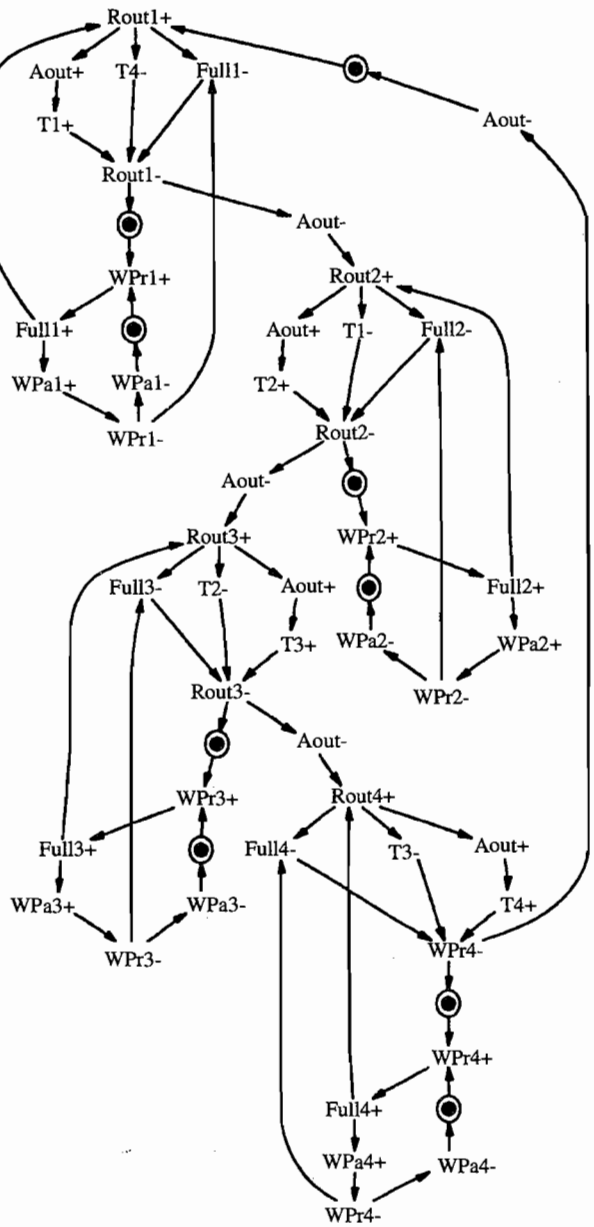


图15.17 4分支电路托肯传递状态转移图

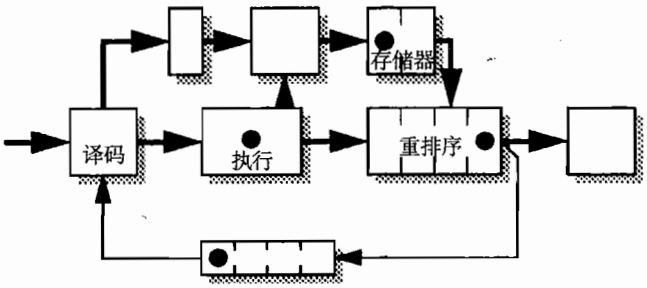


图 15.18 托肯消失抑制重排序缓冲器

另一个风险的产生是由于传输和回写进程不同步，读取的默认寄存器值在读进程中可以被修改，从而导致读取的是随机数据。但是只有当重排序缓冲器中的寄存器存在有效的数据时，这种情况才会发生，因而这个值将被优先传送至寄存器中。假设实现的设备可以确保该随机数据不会在电路内部引发问题，那么这种异步交互是安全的。

ARM 代码的研究表明：对于 Amulet 3 结构，具有 5 个或者更多个入口的重排序缓冲器不可能在任何时候都处于装满的状态[50]，因而 Amulet 3 可实现一个 4 入口缓冲，但其机理可扩展到任何所选的尺寸。

15.4.5 异常

异常（exceptions）是指发生于程序执行过程中的突发事件。Amulet处理器是与ARM指令集兼容的，因而当异常发生时，其类型及特性都是预先定义的。忽略复位，ARM具有以下6种类型的异常：

- 预取异常——指令预取过程中的存储器故障（例如页故障）；
- 数据异常——读写传输过程中的存储器故障；
- 非法指令——仿真程序陷阱；
- 软件中断——系统调用中断（并非真正的异常，但具有类似特性）；
- 中断——正常级别的中断；
- 快速中断——类似于正常的中断，但是中断优先级较高。

对于这些异常，大多都比较容易处理。软件中断和非法指令可在指令译码阶段进行检测，同样预取异常也可以进行检测（此时要求没有要译码的指令）。由于中断是不确定的，因而可以在任意处插入中断。只有数据异常会引发严重的问题，因为数据异常中断在译码已完成并开始执行后才显现。

中断 在任何处理器中，中断都是异步事件。在某种情况下，中断的发生可以视为在以正常次序进行指令读取过程中插入调用指令。初看好像是随意将一个额外的指令插入指令流即可以实现中断，然而这会引起一些问题。

主要的问题在于：中断指令和预取指令流之间互相影响。中断需要程序计数器（PC）的值以得出它的返回地址，但是中断设备不能识别该值。此外，返回的地址必须是有效的。如果中断紧接着分支的预取发生，那么该中断可能会被插入到不应该运行的代码中。

Amulet 3以拦截（hijacking）的方式实现中断，而不是将其插入到指令流中，所有的操作都在预取单元中完成。尽管中断信号（在该情况下是级别敏感型中断信号）的改变是异步的，但是对于预取单元而言，将相互独立的单元视为同步器要优于典型的异步仲裁器，图15.19给出了其中的一种实现方法。在该实现中，中断输入端的任意一个变化都将请求信号延时，直至同步状态被锁存。同步中断信号只有当“完成”为低电平时才发生改变。

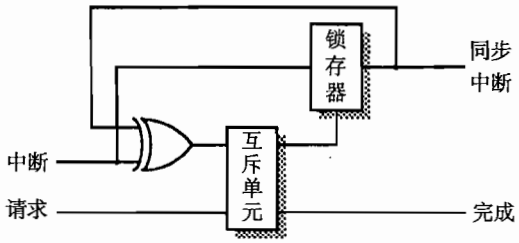


图15.19 中断同步器

当中断信号产生时，当前程序计数器（PC）中的值作为中断指令的地址，并可用来形成返回地址。此时中断信号可作为指令进行发送，并因避开存储器而节省传输时间，然后中断被禁止以避免后续的应答。

由于中断发生在预取单元，故Amulet 3可将中断入口作为一路预定义的分支，并可直接跳转到合适的中断服务程序，在ARM中，该中断服务程序存在于一个固定的地址处。

问题仍然存在，中断入口可能是随机的。如果分支处于挂起状态，那么送回执行级的返回地址可能是无效的——在任何情况下，该分支都可能着色出错，因而被丢弃。分支的运行会更新PC和所有相关的信息，包括中断使能信号。由于中断还没有执行，请求仍然有效，将再另一次尝试进入中断服务程序，此时分支目标地址将被保存，不会有其他的阻碍。

数据异常中断 尽管使用重排序缓冲器解决了寄存器的相关性问题，但该缓存的最初目的是用来简化数据异常的实现。在ARM结构中，一旦异常中断发生，后面指令执行的结果都不会加以保存。早期的Amulet处理器并不考虑存储器操作，而是依赖于任意一个存储管理单元的快速“执行/停止”决策。Amulet 3输出存储器转换请求，并仅在操作结束时检测异常中断，从而允许更加复杂的存储管理（即更慢的存储管理）。该结构允许其他的随机操作并行发生，但是这些随机操作只有当产生执行结果时才会退出。

重排序缓冲器为这些随机结果提供保存空间（在必要时这些随机数据可重新使用），直至这些随机结果被放回寄存器。偶然地发生数据异常中断时这些随机结果被丢弃，并且不对寄存器组产生影响。丢弃的方法有两种：一种是使用着色的方法，使用寄存器回写进程进行检验，

另一种是通过将随机结果记为无效来实现，其使用的标志与由于其他原因导致失败操作的标志一样。尽管在进行的传输操作必须避免因结果无效而产生异步冒险，但为了实现上的方便，Amulet 3使用的是后一种方法。该方法的具体实现是通过执行2个有效位，使其中一个为无效。复制返回进程将这2个位执行与（AND）操作，而传输进程将两者相或（OR），所以传输进程不受干扰，但随后会将其执行结果丢弃。

但是重排序缓冲器只考虑了寄存器的状态，其他需要保留的状态位由ARM控制。两种分离的机制用来实现其他状态位的控制，它主要取决于频率的变化。

第一种是当前程序状态寄存器（CPSR），该寄存器保留处理器的标志、运行模式等状态，所有的状态使用10位表示。在执行过程中标志频繁发生变化，而且这些标志位存在着相关性（例如比较分支序列）。当尝试一个存储器操作时，Amulet 3处理器只是简单地将当前CPSR的状态复制到历史缓存[66] FIFO中，在处理完成之后放弃该入口，但一个异常中断可以恢复CPSR至失败操作之前的状态。

另一种是一组5个程序状态保存寄存器（SPSR），在各种异常入口处，这些寄存器用来对CPRS状态进行临时存储。尽管从理论上讲，在装载和异常中断发生过程中可能改变SPSR，但这种改变一般很少，并且扩大历史缓存以适应这种改变也是不经济的。我们这里所用的解决方法是当任何一个存储器操作变得突出时，使用一个信号量将SPRS锁住，这可以非常经济地实现必须的功能，并且由于SPSR很少发生变化，因而系统性能上的损失也很小。

15.5 存储器——实例研究

看上去，一个异步处理器和一个异步存储器系统交互作用似乎是合理的，这意味着在存储器系统中（包括RAM、ROM及高速缓存）需要有握手接口，这是以下章节所要讨论的主要内容。

一个独立的存储器阵列具有十分规则的结构，并且在稳定的电压和温度等条件下将在固定时间内产生数据。初看似乎在存储器系统中异步设计并没有多大的发展空间，实际上，存储器的每一部分都有它自身特有的时序，在某些情况下，甚至是一个简单存储器也具有变化的周期时间，例如RAM，读RAM的时间明显长于向其写数据的时间。

事实上，存储器系统是一个具有可变时序的计算机的一部分，甚至一个由时钟同步的计算机同样也将使用不同的时长来执行缓存命中和缓存遗失。一个异步系统将能很自然地适应这样的周期时长的变化，并能对许多相当微小的变化加以利用（这些微小的变化在同步系统中可能被用来填充时钟周期）。

15.5.1 顺序存取

静态RAM（SRAM）将数据存储在较小的触发器单元中，这些小触发器仅有非常弱的输出驱动。为了加快读取访问的速度，一般使用读出放大器来检测微小的电压波动，并产生一个早期的数字输出。读出放大器作为模拟部件，对能量的消耗是巨大的。

当电压摆幅大到足以读出位识别时，读出放大器才有用，并且仅在读出位被锁存后才使用。由于这段时长必定小于时钟周期，甚至半时钟周期，因而对于自定时（self-timed）系统而言这是一个理想的应用。当读周期开始时插入一个延时使读出放大器保持在关断状态，当用到时再将其开启。然后RAM中一个特定的位被识别，读取其值用来锁存整个RAM线，并使读出放大器停止工作。相同的信号可用来表明读周期已完成，并可将RAM阵列返回到预充电状态，为下一个周期做准备（见图15.20）。

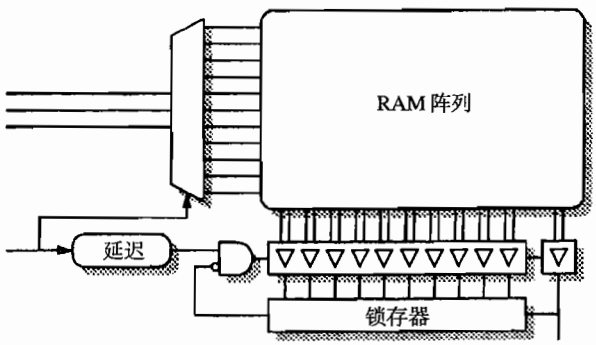


图 15.20 自定时读出放大器

在设计这样一个电路时，很容易检测RAM的操作完成，但是在使能读出放大器之前的延时却很难判断。设计者可以选择短时间的延时，从而确保电路运行于最大速度；或者选择略长时间的延时，从而确保放大器使能之前存储器已经准备就绪，使能量损耗最小。如果设计者在设计中出现失误，速度或者功耗会略微受到影响，但其功能保持不变。

典型的SRAM阵列在结构上大致是正方形的，因而一个1 K的RAM被组织成64 × 128的形式而不是256 × 32，即使在一个给定的周期中处理器只需要32位。这样设计者有两种选择：

- 32路读出放大器复接成所需的字；
- 放大所有的128位，而忽略不需要的位。

当进行一个独立的读操作时，第一种选择看似更好，但是实际上这种单独的读操作很少。典型的存取模式（尤其是代码的提取）表现为相当的有序，并且这种有序性可以用于硬件设计中。

使用第一种选择，可能会使RAM产生短暂的预充电延时，并经更短的延时后执行下一个读操作。Amulet 2e的缓存[49]使用这项技术，因而可在一条RAM线上执行并发的存取操作，且比第一次的存取速度快。这种存取时间上的变化远小于整个时钟周期，因而同步设计者对此没有什么兴趣，但在异步系统中被自动地运用。

上面给出的第二种选择可以在读出放大之后锁存整个RAM线，然后它可以服务于该锁存的并发请求。这样可以释放RAM阵列，使其得以预充电，并可以用做其他目的。Amulet 3i使用了该方法，具体内容在15.5.2节介绍。

15.5.2 Amulet 3i RAM

如图15.7所示，Amulet 3处理器是具有独立的指令和数据总线的哈佛（Harvard）结构。但是在Amulet 3i片上系统（SoC）中，存储器模式是统一的，这暗示着总线必须位于处理器核外部的某处。

实际上，局部总线存在于2个地方，一处是与片上MARBLE总线相连（见图15.21），另一处是访问局部的高速RAM（见图15.21）。在Amulet 3i中，局部RAM是一个存储器映像而不用来构成高速缓存，尽管没有原因表明这里为什么不能实现高速缓存。关于高速缓存的设计将在后面加以讨论。

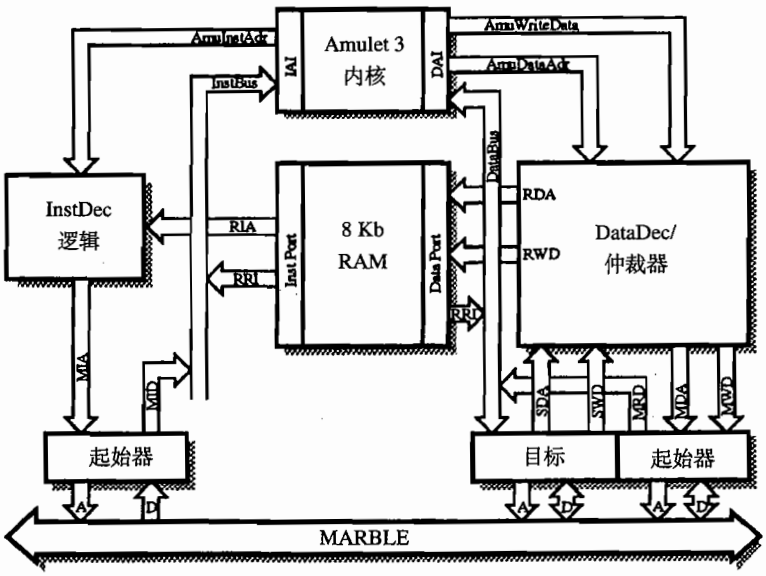


图15.21 Amulet 3i局部存储器

局部RAM（8 KB）划分成大小为1 KB的8个小块。指令和数据总线对每个小块都可访问（见图15.22）。由于各个块之间相互隔开，因而在大多数情况下，指令和数据的提取之间不

存在交互影响。只有在这2条总线需要从同一个RAM读取数据而发生冲突的情况下，才会用到仲裁器进行判别。

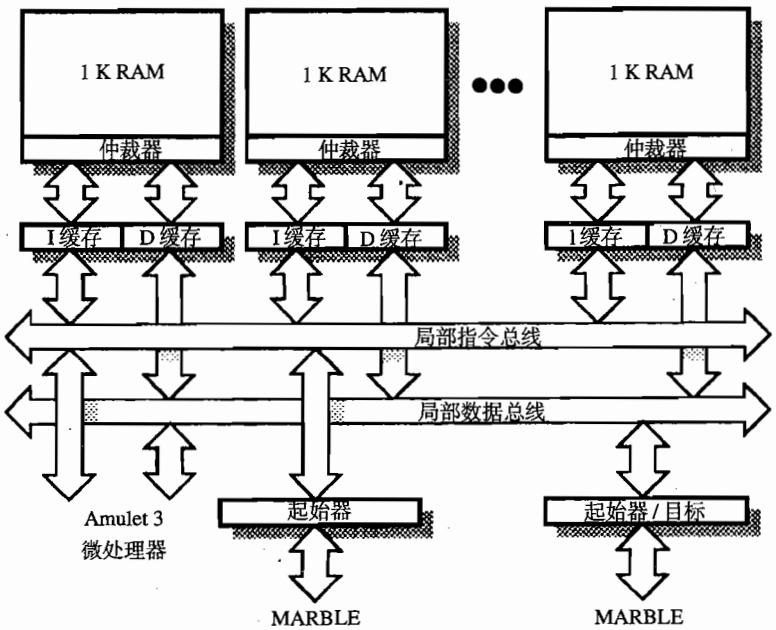


图 15.22 存储块结构

在总线提取发生冲突的情况下，系统并没有为这样一个判别过程分配时钟，该RAM块的访问仅由仲裁器模块（见图15.22）中的一个互斥单元基于先来先服务的原则进行控制。需要注意的是，通常数据和指令的存取不是同步的，因而平均等待时间将大约占典型的RAM存取时间的一半。

通过在读出放大器（见15.5.1节）的输出端使用锁存作为缓存的形式可以进一步减少冲突。此处用于指令和数据读取的锁存是相互独立的，因而顺序的存取很少需要对仲裁资源的竞争。实际上，这使得局部RAM的性能接近于用标准SRAM实现的一个双端口RAM。

这样，局部的RAM结构给存储器周期提供了2种不同的延时（随机延时和顺序延时），并伴随着一个因罕见的冲突而引发的潜在附加延时（该延时可变）。在Amulet 3i中，因2个局部总线循环执行的次数不同，延时情况更加复杂。指令总线可简单看成是一条只读总线，其运行速度明显快于可读可写的局部数据总线，因为数据总线还允许外部总线控制进行DMA（存储器直接访问）操作和检测访问（见图15.20）。系统的异步特性包含了这些不同时长的存在——例如没有必要减慢约25%的指令提取速度以适应数据总线给出的时钟周期组。

存储器块中的仲裁器结构暗示着访问模式是非确定性的，因而必须确保系统不会到达死锁状态。存储器中可能出现的死锁流程如下：

1. 非顺序的数据传输要求访问某一特殊的 RAM 块。
2. 因指令的提取正在使用 RAM 阵列，该访问被禁止。
3. 由于指令译码器仍然处于忙的状态，使得指令提取无法完成。
4. 处理器流水线满，并被数据的提取操作堵塞。
5. 发生死锁！

为了避免死锁的发生，不要对共享资源（RAM 阵列）进行访问，除非确定该操作能将共享资源（RAM 阵列）使用后再次释放，这一点很重要。尽管数据的传输总是能够做到这一点，但是必须制定规定来限制指令提取的发生，直至确保指令的提取操作可以释放 RAM。实际上，读出放大器输出端的锁存构成了一个方便且专用的缓存，该缓存可以保持指令并允许数据访问的进行。在 Amulet 3 结构中，处理器抑制提取请求，因而在任意一个给定的时间内仅存在一个单指令的提取，并且该指令在下一个地址请求发出之前必须从“I缓存”中移除（见图 15.23）。

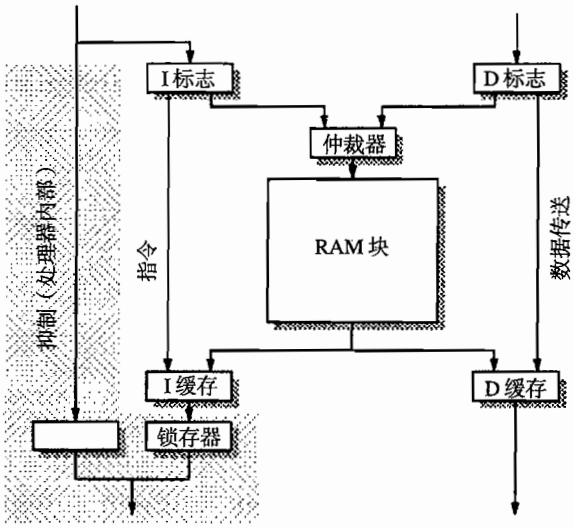


图 15.23 存储块的仲裁和抑制

需要进行仲裁的情形很少，因而通过随机模拟来检验死锁的可能性更小。所以对这样一个非确定性的系统进行彻底分析，以确保去除可能会发生死锁的情况显得尤为重要。

15.5.3 高速缓存

同步高速缓存与异步 RAM 非常类似，大多数高速缓存的设计都是前面所述的异步 RAM 和标准高速缓存设计方法的综合。但是为了达到设计的有效性，必然会存在一些亟待解决的特定问题，这些问题不会在同步缓存中体现出来。

其中最主要的问题在于处理存在于处理器/缓存相互作用中和高速缓存/总线相互作用中的任何冲突，而这些冲突中的首要问题是线提取（line fetch）。

线提取通常发生在当高速缓存尝试访问其位置时丢失的情况下。包含有必要的字和少数相邻字（一条缓存线）的缓存线被从存储器中复制到高速缓存中。解决线提取问题最简单的方法是暂停处理器，提取整条缓存线，然后允许处理器继续运行。因为此时访问处于高速缓存命中的状态，但是该方法要求处理器有延时，并且该延时比实际需要的延时要长得多。

另一个更有效的解决方案是开始提取高速缓存线，一旦所需的字到达就立即将其送入处理器（一般情况下都是第一个被提取的字），然后允许处理器和线提取继续独立地工作。通过允许处理器在进行线提取的同时使用高速缓存的其他部分，以及所取的字一旦到达就立即加以使用来进一步增强系统的性能。但在异步环境下，由于提取的字到达时间与处理器的周期之间并无固定的定时关系，因而这一点很难实现。

最初的想法可能是在高速缓存中使用仲裁，但是不使用仲裁而改用其他方法来解决这一问题也是可能的，即通过使用一个专用的锁存器来保持最后一条提取线，从而维持所有期望的功能。该锁存器称为线提取锁存器。

线提取锁存器（LFL）（见图15.24）实际上是一组存在于真正的RAM阵列之外的锁存器。该锁存器通常存有最后提取的缓存线，它有自身的标签和比较电路，使得它能像其他的缓存线一样运行。需要注意的是，LFL中存有的是该数据唯一的副本。顺便指出，由于LFL是静态的，并且对其进行读取时不需要进行读出放大，因而在异步系统中，它能够提供更快的访问速度。

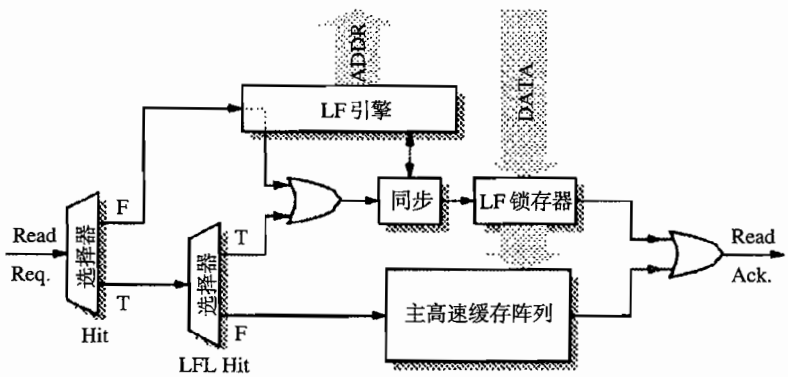


图15.24 请求控制电路

因高速缓存丢失而需要提取缓存线时，来自RAM的一条缓存线被丢弃。在这种情况下，假设高速缓存处于连续写入的状态，从而RAM很容易被改写。此时LFL中的所有内容与其标签一起被复制到选定的缓存线中，然后LFL被标记为“空”（empty）。该操作可在外部访问开始时并行发生。

然后假设处理器获得一个来自于LFL内的高速缓存命中操作，并尝试读取合适的字，在同步情形下这必然会导致延时，因为此时该字为空，并且除非外部存储器速度非常快，否则该字仍不会被重新装满。

一个并发的缓存周期如下。

- 高速缓存命中：该操作独立进行，并且不会与LFL产生交互影响。
- 高速缓存丢失：该操作将引起延时直至线提取进程完成为止，并且提取进程可以重复。
- LFL命中：该操作是在LFL满时尝试对其进行读取。此时可能的两种情况是：一种是所需字已经存在（此时处理器仍正常工作），另一种是该字仍然未到达（此时处理器必须延时直至该字到达为止）。

只有在最后一种情况下，异步进程之间存在交互作用，这种交互作用只是一段等待时间，这个等待可以由一个触发器和一个与门实现（见图15.25）。潜在的等待时间起始于LFL首次为空（由处理器引起，因而与处理器的作用同步），可在任意时间终止，但这种等待只是延迟一个过渡时间，它不会导致异常中断或者改变某一操作，因而其实现不需要进行仲裁，且不存在亚稳态的风险。该机制最初在Amulet 2e高速缓存系统[49]中实现，也在TITAC-2中得到了应用[130]。

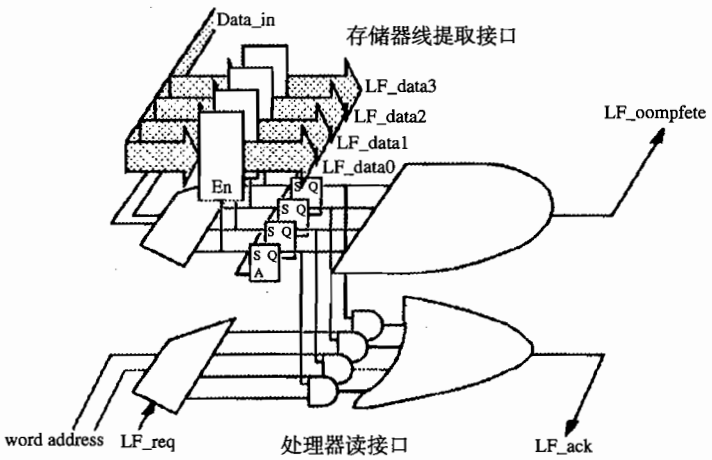


图15.25 线提取锁存的读同步

以上两类处理器都使用了这样一个简单的连续写入缓存来使设计简化，为了达到高性能还需要有回拷（copy-back）机制，该机制也可以通过将LFL机制扩展而得到。在这种情况下，因要求在高速缓存被改写之前将丢弃的线先从该缓存中复制出来，而使得线提取进程变得复杂。可将丢弃的线与其地址（由标志字段提供）一起存放于一个独立的写缓冲器中。需要注意的是，线提取是因高速缓存缺失引起，因而被丢弃的线与被提取线永远都不可能相同（这一点在后面

显得尤为重要）。此时进入RAM阵列的LFL为空，并且开始对其写入。因被丢弃的线对于解决缓存缺失而言不那么重要，因而可推迟其写入操作。

每一条高速缓存线（和写入缓存）都含有一个“dirty”标志，当高速缓存线上的数据有变化时，对其置位。可以通过检验该标志的值，来确定写缓存是否可以进行写出操作（例如，“dirty”为真时，允许写出），或者写入缓存是否已经与存储器保持一致。在后一种情况下，可以旁路回拷进程。

该过程减少了写入的拥挤，但因为只有一个简单的入口写缓存，而且写缓存必须在顺序提取之前为空，所以该过程在减少提取等待时间上并没有发挥多大的作用。然而写缓存是可以扩展的，尽管这种扩展是以引入仲裁器为代价，并因此带来潜在的缺陷。

如果在前一个线提取没有完成之前，需要执行第二个线提取进程，从理论上来讲，可以通过将各种写操作挂起实现，这同样也是我们期望的结果。当写操作已经被挂起时（只是等待下一次总线空闲时释放），确定新的提取请求在写操作开始之前是否到达是有必要的，该过程需要在异步环境下进行仲裁。但是，一旦做出了决定，操作会继续像前面一样执行，但以下两个问题仍然存在：

- 如果写缓存已满，将没有地方来存放被剔除的线，系统发生死锁。
- 如果当前要求的线是最近刚被剔除出去的线，提取操作要挂起一个写操作，因而会失去高速缓存的一致性。

第一个问题较容易解决。一个简单的计数器就可以确保只允许超过一定数量的写进程发生，并且在写缓存中始终有一个空间是空闲的（例如，当最后一个入口被塞满时，下一总线操作必然是写操作，当然，“clean”线除外，此时写操作可以被旁路）。这种情况是可以实现的，例如，作为一个信号量。

第二个问题比较难解决，但是该问题可以通过采用与处理器重排序缓冲器类似的方式来解决。线提取操作检测写缓存中的入口地址，如果检测到是匹配的，则留下来，而不需要请求存储总线。在这种情况下，提取操作的执行没有等待时间，执行速度大大加快，就像在执行一个内部操作一样，并且能在一个单一的操作中复制一条完整的快速缓存线。由于整个系统是一个自定时系统，因而该过程的执行与系统其他部分功能的执行无关。这样的操作并不影响写进程（因为重新提取的“dirty”线在返回时已经被清除），并且不管回拷进程是处于挂起、执行、已完成还是不需要，该传输过程都能够发生。实际上，写缓存是作为一个顺序写入的高速缓存的牺牲品而存在的[56]。

在上述过程中，有一点需要注意，线提取与高速缓存线的剔除是并行发生的，因而在比较过程中写入缓存的入口地址有可能被改变。正如上面所述，该入口地址是为新近被剔除的线所保留的，因而可以在比较过程中安全地排除，这就避免了信号变化导致错误发生的可能性。

15.6 大型异步系统

15.6.1 片上系统（DRACO）

DRACO（DECT射频通信控制器）（见图15.26）是基于类似Amulet 3处理器的片上系统。芯片上一半的区域（7 mm^2）是包含Amulet 3i的异步电路，另一半由VHDL编译出来的同步外设构成。

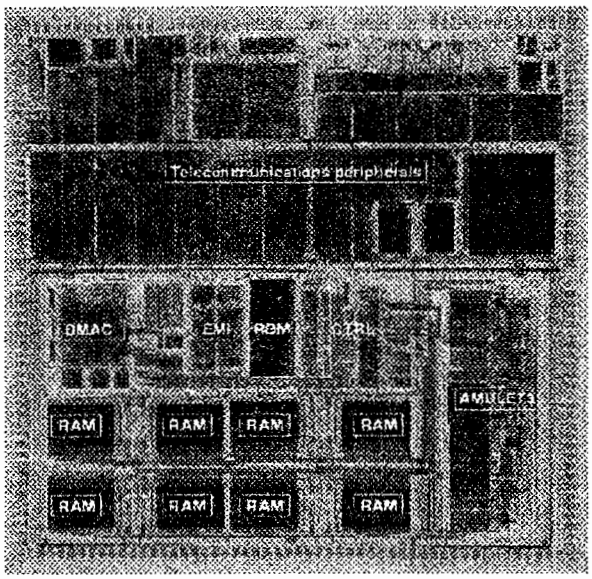


图15.26 DRACO版图

异步子系统（见图15.27）本身是计算机，它的开发既出于商业目的，又是研究新技术的需要。处理器和RAM已经讨论过了，本节就另外的一些新型异步特性加以阐述。

15.6.2 互连

从概念上讲，一个异步系统基本上是基于异步总线的。实际上可以验证，在一个大规模的快速同步系统中，其同步子系统之间也应该采用异步互连方式，以减少高速时钟分配和时钟偏移引发的问题。该模型有时被称为“GALS”（全局异步，局部同步），并为传统的系统囊括异步电路提供了早期的商业契机。

MARBLE 作为开发这样一个互连标准的一步，Amulet 3i使用了MARBLE[5]——一种32位，多控制器的片上总线，使用握手信号而不是时钟进行通信。除此之外其信号的定义（包括32位的地址和数据）与传统的总线十分相似。MARBLE将地址和数据分开通信，并允许流水线操作和独立操作，从而可以在多个设备要求全局访问时增加可用带宽。

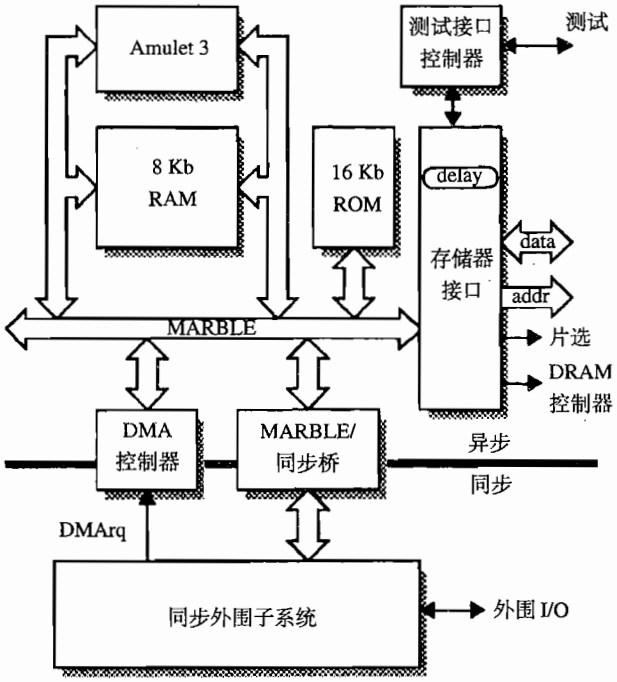


图 15.27　Amulet 3i 异步子系统

MARBLE 可以连接到任何异步元件上的起始和目标接口，其地址和总线提供了各元件之间通信所必需的信息。在 Amulet 3i 中，有 4 个起始端口和 7 个目标接口。例如，处理器的两条局部总线各自终止于MARBLE的起始端，并且局部数据总线也是MARBLE的目标端，它允许其他的起始端口对 RAM 进行 DMA（存储器直接访问）和 RAM 读写数据的测试。

Chain　Chain（芯片区域的互连）作为一种可能取代传统总线用于片上通信的方案目前正在开发中。Chain 是基于这样一种窄带、高速的、点对点的链接，与其说是形成一条总线还不如说是构成一个网络。其想法是在异步系统中利用它进行快速符号传送的潜能，同时减少长距离传输线的数量。

通过使用延时无关的编码方案，Chain 使得芯片设计者不需考虑整个芯片的时序收敛，同时 Chain 还能容许诸如在长距离互连线上存在的串扰等潜在问题。此外，用户只需与传统的并行接口进行通信就可以了。

15.6.3　Balsa 和 DMA 控制器

DMA 控制器是一个复杂的多通道单元，该单元根据外部规范进行了改进。然而即便是在异步领域中，该单元的功能也相当简单，它作为Balsa综合的第一个实际应用是值得注意的[11]。

DMA控制器由一组用于多路DMA通道的控制寄存器和一个控制单元组成，该控制单元选择一个当前的请求并为之服务。寄存器被设计成自定义的VLSI块，从而对其面积加以优化。控制逻辑使用Balsa语言编写，并且当规范改变时，可以对其进行多次修改，修正之后的控制逻辑更能适应当前的环境。

这种综合对于高性能的单元是不适合的，但是对于性能受限于其他的限制因素（在这里是性能受限于总线速度）和程序调试时间占主导的应用是非常有用的。当然，在异步环境下，在性能允许范围内调节设备而不影响整个系统的功能是很容易实现的。

本书的第二部分介绍了Balsa，并给出了一个简单的4通道DMA控制器完整的资源列表。

15.6.4 校准时间延时

为了用在实际系统中，异步处理器必须能实现与现有元件的接口兼容，同时最好带有具有握手接口的存储设备，而这些设备或许市场还没有。相反，商品化的存储器依靠绝对时序假设来保证它们的正确操作，这样，使用这种存储器时，系统必须有精确的时序基准，这在异步设计中是没有的。

单纯地为存储器引入一个时钟，用以提供时序会抹杀异步系统的许多优势，因此如果可以提供充分精确的延迟，保留“一经请求便提供数据”的想法是更可取的。这种延迟不需要时钟，用一个“一经请求”即触发的单稳态触发器会更好。

Amulet 1和Amulet 2e依靠一条外部提供的延迟线路来提供总线时序基准。这是一种灵活的解决方案，因为短的延迟能通过重复使用可以提供更长的定时间隔，从而提供一个灵活的可编程接口。例如，地址空间的一块区域能为DRAM设备设置1个延迟地址建立时间、2个延迟RAS地址保持时间等。那么总线接口可以适当计算延迟。

这是一个合理的解决方案，但是存在一些缺陷：

- 每个延迟周期的片上延迟不可以计算；
- 延迟线不是特别精确；
- 驱动片外延时比较耗能。

使用片外延时是一种更好的解决方法。这种方法的首要问题是，固定延迟不精确，芯片之间的延迟会改变，并且随着温度和所提供的电源电压波动而变化。因此与外部基准相比，板上延迟必定是可以校准的。

Amulet 3i使用的延迟包含一系列的延迟元素（见图15.28），能简化为一个确定的点，能通过计算一个已知的时间间隔里完成的周期数来校准，并且利用图中所示的控制线做相应调整。一旦被校准，延迟将缓慢变化（例如随着温度漂移），除非外部条件改变。因此，校准在软件控制下能在很少的时间间隔内重复。外部定时基准可以是一个长的延迟，比如一段32 kHz监视振荡器，几乎不需要能量！

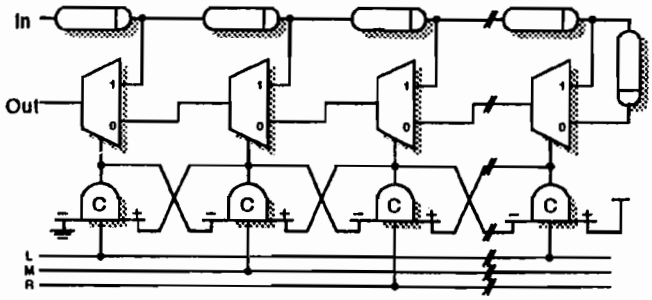


图 15.28 可控的延时电路

15.6.5 产品测试

图 15.27 中有一个测试接口控制器模块，DRACO 芯片是作为一款商业产品而设计的，因此在生产中必须是可测的。在 DRACO 设计中所采用的测试方法基于 MARBLE 总线来访问各种片上模块，并获取它们的测试特征。在常规的使用中，外部存储器接口的目标是MARBLE总线，但是为了测试，可以将其重新配置成一个总线起始器，从而使得外部的测试器可以对总线进行控制，并进行片上存储器的读写。测试接口控制器中的电路使得该接口高效率运行，并支持自动连续地址的生成等。

异步子系统所有的产品测试均使用外部测试器将测试程序加载到片上RAM中，然后通过命令指示 Amulet 3 处理器执行该程序。测试的运行不受测试器的干预，测试器必须在片上存储器区域读取一个 pass/fail（通过/失败）代码之前等待一段固定的时间。测试程序全速运行，并且不存在有外部时序的输入对其进行控制。

在纯粹的功能模式下，某些片上模块很难进行测试，从而需要提供一些额外的访问路径（通过可访问MARBLE的测试寄存器），以使测试更加高效。校准时间延时就是这样一种电路，处理器分支预测也是这种类型的电路。在后一种情况下，分支预测被从线上拿下，这使得处理器（分支预测是其中一部分）可以控制其内部状态在其CAM和RAM单元中运行优化的测试。

虽然DRACO在我们编写阶段还没有完全测试产品，但测试可以在其原型样品中畅通无阻地运行。

15.7 小结

诸如DRACO等器件证实了构建大规模功能性好的异步系统是切实可行的，但和任意一个原型系统一样，该芯片也存在一些问题。在半异步的系统中存在两个众所周知的问题，乘法器中驱动强度问题和预取单元的逻辑失误，前者导致仿真中得不到结果，后者在某一特定条件下会错误地指示一个循环周期是顺序的（这是一个从片外 DRAM 中运行代码所引发的问题）。

以上两个问题都不能归因于该器件的异步本性（实际上该器件的同步部分存在的错误稍多一些），并且二者都很容易修正。异步处理器无论是在物理尺寸、性能还是在功耗上都可以与ARM9相匹敌，初步测量显示其电磁干扰（EMI）更小。

本章针对设计者在设计复杂的异步处理系统和异步存储系统时面临的一系列问题，给出了可能的解决方案（尽管解决方法不止一个），本章开始部分描述的大多数设计是学术团队提出来的，因此可以归为“研究”类，然而，目前一个片上系统是如此复杂，以至于这些设计融入了一个很大的大学团队的力量。事实表明，大规模、功能性好的异步系统设计不仅是可行的，具有竞争性，并且在功耗和EMI等方面具有某些独特的优势。尽管这些设计中含有局部时钟，异步互连也许是解决大规模设计的唯一解决方案。异步芯片设计已经进入发展阶段。

结束语

异步技术在数字电子学发展之初就已经存在——早期的很多计算机都没有使用全局时钟信号。然而随着集成电路的发展，晶体管资源的成倍增长需要简单直接的设计理念，从而同步设计成为主流。今天，大多数有经验的数字电路设计者对异步技术知之甚少，他们所知道的只是如何避险。但是基于时钟的同步设计的不足已经日渐显示出来：对于设计规模的增大，它显得无能为力，并且额外的功耗以及电磁干扰问题也日益增长。

在同步设计占统治地位时期，仍有少数设计者深信异步技术有其自身的优点，新发展的异步技术相较于早期机器中使用的方法而言，会更加适于VLSI设计。在本书中，我们试图以一种能被任意数字电路设计者所接受的方式阐述这些新技术，无论这些设计者先前是否对异步电路有所了解。

对于以上的异步设计技术，我们要有选择地对待，从而不至于因为晦涩的细节部分而分不清首要的目标。很多相当有价值的工作在这里都被省略了，那些被本书调动了兴趣的读者将会发现，在书中并未介绍一些已出版的、含有大量异步设计信息的资料。

尽管已经有基于异步技术的VLSI器件的成功商业实例（有几个在本书中有介绍），但是这些设计只是个例，大多数的异步设计目前仍然处于实验室研究阶段。如果将来这种情形有所改变，那么将会在哪个应用领域首先得到印证呢？

对于时钟设计即将消失的预测已有好多年了，但目前仍然没有发生。如果这种情况真的发生了，那也是迫于一些不得已的原因，因为设计者不会轻易丢掉他们在一种设计风格上积累多年的经验，而转为接受另一种缺少实例，并且没有很好的自动化工具支持的设计风格。

考虑使用异步设计有很多可能的理由，但是没有一个应用可以使得异步设计的使用是必须的。在本书的开头部分，我们已经提到了使用异步设计方法的一些论点，诸如低功耗、低电磁干扰特性及模块化等，但这些观点只是在其自身的小环境下才适用，只有模块化这一特性具有获得广泛应用的潜质。一种有前景的可以支持不同时序环境的设计方法是GALS——即全局异步局部同步的系统设计方法。这种方法中时钟模块的连接通过使用异步的片上互连——诸如

Chain（见15.6.2节）等芯片区域网络互连。模块本身可以足够小以限制时钟偏移，从而使得简单直接的同步设计技术可以很好地工作，并且不同的模块可以使用不同的或者具有不同相位的同一时钟。一旦这种框架可行，基于逐项推进的方法，使各个模块异步无疑便是简单的。

或许已是顺理成章的事情，我们看到了在已有的同步设计技术中融入异步技术的必要，从而大多数的功能设计可以使用成熟的工具和设计方法实现。这种改进的方法与本书第三部分中所描述的革命性的改进形成对比，并为本书中所描述的异步技术的广泛使用给出了最有可能的设想。

但是在短期内，得益于异步技术的应用场合是重要的也是可行的，我们写这本书，是希望更多的设计者了解异步设计的原理及其潜在的优势，从而为新旧问题都提供一种新的解决方案。设计中时钟是十分有用的，但是往往会成为限制因素。不要害怕打破常规来思考问题。

关于异步设计的更多信息可以参阅本书后所列的参考文献，Internet上的异步参考书目[111]，以及Internet异步逻辑主页[47]上可获得的关于异步设计的完整信息。

参考文献

[1] G. Abouyannis et al. Project PREST EP25242. *European Low Power Initiative for Electronic System Design (ESDLPD) Third International Workshop*, pages 5-49, 2000.

[2] A. Abrial, J. Bouvier, M. Renaudin, P. Senn, and P. Vivet. A new contactless smartcard IC using an on-chip antenna and an asynchronous micro-controller. *IEEE Journal of Solid-State Circuits*, 36(7): 1101-1107, July 2001.

[3] T. Agerwala. Putting Petri nets to work. *IEEE Computer*, 12(12):85-94, December 1979.

[4] W.J. Bainbridge and S.B. Furber. Asynchronous macrocell interconnect using MARBLE. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async'98)*, pages 122-132. IEEE Computer Society Press, April 1998.

[5] W.J. Bainbridge and S.B. Furber. MARBLE: An asynchronous onchip macrocell bus. *Microprocessors and Microsystems*, 24(4):213-222, April 2000.

[6] T.S. Balraj and M.J. Foster. Miss Manners: A specialized silicon compiler for synchronizers. In Charles E. Leierson, editor, *Advanced Research in VLSI*, pages 3-20. MIT Press, April 1986.

[7] A. Bardsley. The Balsa web pages. `http://www.cs.man.ac.uk/amulet/balsa/projects/balsa`.

[8] A. Bardsley. *Implementing Balsa Handshake Circuits*. PhD thesis, Department of Computer Science, University of Manchester, 2000.

[9] A. Bardsley and D.A. Edwards. Compiling the language Balsa to delayinsensitive hardware. In C. D. Kloos and E. Cerny, editors, *Hardware Description Languages and their Applications (CHDL)*, pages 89-91, April 1997.

[10] A. Bardsley and D.A. Edwards. The Balsa asynchronous circuit synthesis system. In *Forum on Design Languages*, September 2000.

[11] A. Bardsley and D.A. Edwards. Synthesising an asynchronous DMA controller with Balsa. *Journal of Systems Architecture*, 46:1309-1319, 2000.

[12] PA. Beerel, C.J. Myers, and T.H.-Y. Meng. Automatic synthesis of gate-level speed-independent circuits. Technical Report CSL-TR-94-648, Stanford University, November 1994.

[13] P.A. Beerel, C.J. Myers, and T.H.-Y. Meng. Covering conditions and algorithms for the synthesis of speed-independent circuits. *IEEE Transactions on Computer-Aided Design*, March 1998.

[14] G. Birtwisfie and A. Davis, editors. *Proceedings of the Banff VIII Workshop: Asynchronous Digital Circuit Design, Banff, Alberta, Canada, August 28-September 3, 1993.* Springer Verlag, Workshops in Computing Science, 1995. Contributions from: S.B. Furber, "Computing without Clocks: Micropipelining the ARM Processor," A. Davis, "Practical Asynchronous Circuit Design: Methods and Tools," C.H. van Berkel, "VLSI Programming of Asynchronous Circuits for Low Power," J. Ebergen, "Parallel Program and Asynchronous Circuit Design," A. Davis and S. Nowick, "Introductory Survey".

[15] I. Bogdan, M. Munteau, P.A. Ivey, N.L. Seed, and N. Powell. Power reduction techniques for a Viterbi decoder implementation. *European Low Power Initiative for Electronic System Design (ESDLPD) Third International Workshop*, pages 28-48, 2000.

[16] E. Brinksma and T. Bolognesi. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1), 1987.

[17] E. Brunvand and R.F. Sproull. Translating concurrent programs into delay-insensitive circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 262-265. IEEE Computer Society Press, November 1989.

[18] J.A. Brzozowsky and C.-J.H. Seager. *Asynchronous Circuits*. Springer Verlag, Monographs in Computer Science, 1994. ISBN: 0-387-94420-6.

[19] S.M. Burns. *Performance Analysis and Optimization of Asynchronous Circuits*. PhD thesis, Computer Science Department, California Institute of Technology, 1991. Caltech-CS-TR-91-01.

[20] S.M. Burns. General condition for the decomposition of state holding elements. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.

[21] S.M. Burns and A.J. Martin. Syntax-directed translation of concurrent programs into self-timed circuits. In J. Allen and F. Leighton, editors, *Advanced Research in VLSI*, pages 35-50. MIT Press, 1988.

[22] D.M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Stanford University, October 1984.

[23] K.T. Christensen, P. Jensen, P Korger, and J. Sparsϕ. The design of an asynchronous TinyRISC TR4101 microprocessor core. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 108-119. IEEE Computer Society Press, 1998.

[24] T.-A. Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis, MIT Laboratory for Computer Science, June 1987.

[25] T.-A. Chu and R.K. Roy (editors). Special issue on asynchronous circuits and systems. *IEEE Design & Test*, 11(2), 1994.

[26] T.-A. Chu and L.A. Glasser. Synthesis of self-timed control circuits from graphs: An example. In *Proc. International Conf. Computer Design (ICCD)*, pages 565-571. IEEE Computer Society Press, 1986.

[27] B. Coates, A. Davis, and K. Stevens. The Post Office experience: Designing a large asynchronous chip. *Integration, the VLSI journal*, 15(3):341-366, October 1993.

[28] F. Commoner, A.W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *J. Comput. System Sci.*, 5(1):511-523, October 1971.

[29] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. In *XI Conference on Design of Integrated Circuits and Systems*, Barcelona, November 1996.

[30] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315-325, March 1997.

[31] U. Cummings, A. Lines, and A. Martin. An asynchronous pipelined lattice structure filter. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 126-133, November 1994.

[32] A. Davis. A data-driven machine architecture suitable for VLSI implementation. In *Proceedings of the First Caltech Conference on VLSI*, pages 479-494, Pasadena, CA, January 1979.

[33] A. Davis and S.M. Nowick. Asynchronous circuit design: Motivation, background, and methods. In G. Birtwistle and A. Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 1-49. Springer-Verlag, 1995.

[34] A. Davis and S.M. Nowick. An introduction to asynchronous circuit design. Technical Report UUCS-97-013, Department of Computer Science, University of Utah, September 1997.

[35] A. Davis and S.M. Nowick. An introduction to asynchronous circuit design. In A. Kent and J. G. Williams, editors, *The Encyclopedia of Computer Science and Technology*, volume 38. Marcel Dekker, New York, February 1998.

[36] J.B. Dennis. Data Flow Computation. In *Control Flow and Data Flow — Concepts of Distributed Programming, International Summer School*, pages 343-398, Marktoberdorf, West Germany, July 31 - August 12, 1984. Springer, Berlin.

[37] J.C. Ebergen and R. Berks. Response time properties of linear asynchronous pipelines. *Proceedings of the IEEE*, 87(2):308-318, February 1999.

[38] P.B. Endecott and S.B. Furber. Modelling and simulation of asynchronous systems using the LARD hardware description language. In *Proceedings of the 12th European Simulation Multiconference, Manchester, Society for Computer Simulation International*, pages 39-43, June 1994. ISBN 1-56555-148-6.

[39] K.M. Fant and S.A. Brandt. Null Conventional Logic: A complete and consistent logic for asynchronous digital circuit synthesis. In *International Conference on Application-specific Systems, Architectures, and Processors*, pages 261-273, 1996.

[40] R.M. Fuhrer, S.M. Nowick, M. Theobald, N.K. Jha, B. Lin, and L. Plana. Minimalist: An environment for the synthesis, verification and testability of burst-mode asynchronous machines. Technical Report TR CUCS-020-99, Columbia University, NY, July 1999. http://www.cs.columbia.edu/~nowick/minimalist.pdf.

[41] S.B. Furber and P. Day. Four-phase micropipeline latch control circuits. *IEEE Transactions on VLSI Systems*, 4(2):247-253, June 1996.

[42] S.B. Furber, P. Day, J.D. Garside, N.C. Paver, S. Temple, and J.V. Woods. The design and evaluation of an asynchronous microprocessor. In *Proc. Int'l. Conf. Computer Design*, pages 217-220, October 1994.

[43] S.B. Furber, D.A. Edwards, and J.D. Garside. AMULET3: a 100 MIPS asynchronous embedded processor. In *Proc. International Conf. Computer Design (ICCD)*, September 2000.

[44] S.B. Furber, J.D. Garside, P. Riocreux, S. Temple, P. Day, J. Liu, and N.C. Paver. AMULET2e: An asynchronous embedded controller. *Proceedings of the IEEE*, 87(2): 243-256, February 1999.

[45] S.B. Furber, J.D. Garside, S. Temple, J. Liu, P. Day, and N.C. Paver. AMULET2e: An asynchronous embedded controller. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 290-299. IEEE Computer Society Press, 1997.

[46] EU IST-1999-13515, G3Card - generation 3 smartcard, January 2000.

[47] J.D. Garside. The Asynchronous Logic Homepages. `http://www.cs.man.ac.uk/async/`.

[48] J.D. Garside, W.J. Bainbridge, A. Bardsley, D.A. Edwards, S.B. Furber, J. Liu, D.W. Lloyd, S. Mohammadi, J.S. Pepper, O. Petlin, S. Temple, and J.V. Woods. AMULET3i - an asynchronous system-on-chip. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 162-175. IEEE Computer Society Press, April 2000.

[49] J.D. Garside, S. Temple, and R. Mehra. The AMULET2e cache system. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async'96)*, pages 208-217. IEEE Computer Society Press, March 1996.

[50] D.A. Gilbert. *Dependency and Exception Handling in an Asynchronous Microprocessor*. PhD thesis, Department of Computer Science, University of Manchester, 1997.

[51] D.A. Gilbert and J.D. Garside. A result forwarding mechanism for asynchronous pipelined systems. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async'97)*, pages 2-11. IEEE Computer Society Press, April 1997.

[52] B. Gilchrist, J.H. Pomerene, and S.Y. Wong. Fast carry logic for digital computers. *IRE Transactions on Electronic Computers*, EC-4(4): 133-136, December 1955.

[53] L.A. Glasser and D.W. Dobberpuhl. *The Design and Analysis of VLSI Circuits*. Addison-Wesley, 1985.

[54] S. Hauck. Asynchronous design methodologies: An overview. *Proceedings of the IEEE*, 83(1):69-93, January 1995.

[55] L.G. Heller, W.R. Griffin, J.W. Davis, and N.G. Thoma. Cascade voltage switch logic: A differential CMOS logic family. *Proc. International Solid State Circuits Conference*, pages 16-17, February 1984.

[56] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach (2nd edition)*. Morgan Kaufmann, 1996.

[57] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8): 666-677, August 1978.

[58] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, 1985.

[59] D.A. Huffman. The synthesis of sequential switching circuits. *J. Franklin Inst.*, pages 161-190, 275-303, March/April 1954.

[60] D.A. Huffman. The synthesis of sequential switching circuits. In E. F. Moore, editor, *Sequential Machines: Selected Papers*. Addison-Wesley, 1964.

[61] H. Hulgaard, S.M. Burns, and G. Borriello. Testing asynchronous circuits: A survey. *Integration, the VLSI journal*, 19(3): 111-131, November 1995.

[62] K. Hwang. *Computer Arithmetic: Principles, Architecture, and Design*. John Wiley & Sons, 1979.

[63] ISO/IEC. Mifare identification cards - contactless integrated circuit(s) cards - proximity cards. Standard ISO/IEC Standard 14443 Type A.

[64] H. Jacobson, E. Brunvand, G. Gopalakrishnan, and P. Kudva. High-level asynchronous system design using the ACK framework. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 93-103. IEEE Computer Society Press, April 2000.

[65] D. Jaggar. *Advanced RISC Machines Architecture Reference Manual*. Prentice Hall, 1996.

[66] M. Johnson. *Superscalar Microprocessor Design*. Series in Innovative Technology. Prentice Hall, 1991.

[67] S.C. Johnson and S. Mazor. Silicon compiler lets system makers design their own VLSI chips. *Electronic Design*, 32(20): 167-181, 1984.

[68] G. Jones. *Programming in OCCAM*. Prentice-Hall international, 87.

[69] M.B. Josephs, S.M. Nowick, and C.H. van Berkel. Modeling and design of asynchronous circuits. *Proceedings of the IEEE*, 87(2):234-242, February 1999.

[70] G.C. Clark Jr. and J.B. Cain. *Error correcting coding for digital communication*. Plenum, 1981.

[71] G.D. Forney Jr. The Viterbi algorithm. *Proc. IEEE*, 13(3):268-278, 1973.

[72] G. Kane and J. Heinrich. *MIPS RISC Achitecture*. Prentice Hall, 1992.

[73] J. Kessels, T. Kramer, G. den Besten, A. Peeters, and V. Timm. Applying asynchronous circuits in contactless smart cards. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 36-44. IEEE Computer Society Press, April 2000.

[74] J. Kessels, T. Kramer, A. Peeters, and V. Timm. DESCALE: a design experiment for a smart card application consuming low energy. In R. van Leuken, R. Nouta, and A. de Graaf, editors, *European Low Power Initiative for Electronic System Design*, pages 247-262. Delft Institute of Microelectronics and Submicron Technology, July 2000.

[75] Z. Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill, 1978.

[76] A. Kondratyev, J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Logic decomposition of speed-independent circuits. *Proceedings of the IEEE*, 87(2):347-362, February 1999.

[77] J. Liu. *Arithmetic and control components for an asynchronous microprocessor*. PhD thesis, Department of Computer Science, University of Manchester, 1997.

[78] D.W. Lloyd. VHDL models of asychronous handshaking. (Personal communication, August 1998).

[79] A.J. Martin. The probe: An addition to communication primitives. *Information Processing Letters*, 20(3):125-130, 1985. Erratum: IPL 21(2): 107, 1985.

[80] A.J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226-234, 1986.

[81] A.J. Martin. Formal program transformations for VLSI circuit synthesis. In E.W. Dijkstra, editor, *Formal Development of Programs and Proofs*, UT Year of Programming Series, pages 59-80. Addison-Wesley, 1989.

[82] A.J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In W.J. Dally, editor, *Advanced Research in VLSI: Proceedings of the Sixth MIT Conference*, pages 263-278. MIT Press, 1990.

[83] A.J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C.A.R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1-64. Addison-Wesley, 1990.

[84] A.J. Martin. Synthesis of asynchronous VLSI circuits, 1991.

[85] A.J. Martin. Asynchronous datapaths and the design of an asynchronous adder. *Formal Methods in System Design*, 1(1):119-137, July 1992.

[86] A.J. Martin, S.M. Bums, T. K. Lee, D. Borkovic, and PJ. Hazewindus. The design of an asynchronous microprocessor. In Charles L. Seitz, editor, *Advanced Research in VLSI*, pages 351-373. MIT Press, 1989.

[87] A.J. Martin, S.M. Burns, TK. Lee, D. Borkovic, and P.J. Hazewindus. The first asynchronous microprocessor: The test results. *Computer Architecture News*, 17(4):95-98, 1989.

[88] A.J. Martin, A. Lines, R. Manohar, M. Nyström, P. Penzes, R. Southworth, U.V. Cummings, and T.-K Lee. The design of an asynchronous MIPS R3000. In *Proceedings of the 17th Conference on Advanced Research in VLSI*, pages 164-181. MIT Press, September 1997.

[89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[90] C.E. Molnar, I.W. Jones, W.S. Coates, and J.K. Lexau. A FIFO ring oscillator performance experiment. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 279-289. IEEE Computer Society Press, April 1997.

[91] C.E. Molnar, I.W. Jones, W.S. Coates, J.K. Lexau, S.M. Fairbanks, and I.E. Sutherland. Two FIFO ring performance experiments. *Proceedings of the IEEE*, 87(2):297-307, February 1999.

[92] D.E. Muller. Asynchronous logics and application to information processing. In H. Aiken and W. F. Main, editors, *Proc. Symp. on Application of Switching Theory in Space Technology*, pages 289-297. Stanford University Press, 1963.

[93] D.E. Muller and W.S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching, Cambridge, April 1957, Part I*, pages 204-243. Harvard University Press, 1959. The annals of the computation laboratory of Harvard University, Volume XXIX.

[94] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77 (4):541-580, April 1989.

[95] C.J. Myers. *Asynchronous Circuit Design*. John Wiley & Sons, July 2001. ISBN: 0-471-41543-X.

[96] National Bureau of Standards. Data encryption standard, January 1997. Federal Information Processing Standards Publication 46.

[97] C.D. Nielsen. Evaluation of function blocks for asynchronous design. In *Proc. European Design Automation Conference (EURO-DAC)*, pages 454-459. IEEE Computer Society Press, September 1994.

[98] C.D. Nielsen, J. Staunstrup, and S.R. Jones. Potential performance advantages of delay-insensitivity. In M. Sami and J. Calzadilla-Daguerre, editors, *Proceedings of IFIP workshop on Silicon Architectures for Neural Nets, StPaul-de-Vence, France, November 1990*. North-Holland, Amsterdam, 1991.

[99] L.S. Nielsen. *Low-power Asynchronous VLSI Design*. PhD thesis, Department of Information Technology, Technical University of Denmark, 1997. IT-TR:1997-12.

[100] L.S. Nielsen, C. Niessen, J. Sparsϕ, and C.H. van Berkel. Low-power operation using self-timed circuits and adaptive scaling of the supply voltage. *IEEE Transactions on VLSI Systems*, 2(4):391-397, 1994.

[101] L.S. Nielsen and J. Sparsϕ. A low-power asynchronous data-path for a FIR filter bank. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 197-207. IEEE Computer Society Press, 1996.

[102] L.S. Nielsen and J. Sparsϕ. An 85 μW asynchronous filter-bank for a digital hearing aid. In *Proc. IEEE International Solid State circuits Conference*, pages 108-109, 1998.

[103] L.S. Nielsen and J. Sparsϕ. Designing asynchronous circuits for low power: An IFIR filter bank for a digital hearing aid. *Proceedings of the IEEE*, 87(2):268-281, February 1999. Special issue on "Asynchronous Circuits and Systems" (Invited Paper).

[104] D.C. Noice. *A Two-Phase Clocking Dicipline for Digital Integrated Circuits*. PhD thesis, Department of Electrical Engineering, Stanford University, February 1983.

[105] S.M. Nowick. Design of a low-latency asynchronous adder using speculative completion. *IEEE Proceedings, Computers and Digital Techniques*, 143(5):301-307, September 1996.

[106] S.M. Nowick, M.B. Josephs, and C.H. van Berkel (editors). Special issue on asynchronous circuits and systems. *Proceedings of the IEEE*, 87(2), February 1999.

[107] S.M. Nowick, K.Y. Yun, and P.A. Beerel. Speculative completion for the design of high-performance asynchronous dynamic adders. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 210-223. IEEE Computer Society Press, April 1997.

[108] International Standards Organization. LOTOS — a formal description technique based on the temporal ordering of observational behaviour. ISO IS 8807, 1989.

[109] N.C. Paver, P. Day, C. Farnsworth, D.L. Jackson, W.A. Lien, and J. Liu. A low-power, low-noise cortfigurable self-timed DSP. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 32-42, 1998.

[110] M. Pedersen. Design of asynchronous circuits using standard CAD tools. Technical Report IT-E 774, Technical University of Denmark, Dept. of Information Technology, 1998. (In Danish).

[111] A.M.G. Peeters. The 'Asynchronous' Bibliography.
`http://www.win.tue.nl/~ws inap/a sync.html`.
Corresponding e-mail address: `async-bib@win.tue.nl`.

[112] A.M.G. Peeters. *Single-Rail Handshake Circuits*. PhD thesis, Eindhoven University of Technology, June 1996.
`http://www.win.tue.nl/ ~wsinap/pdf/Peeters96.pdf`.

[113] J.L. Peterson. Petri nets. *Computing Surveys*, 9(3):223-252, September 1977.

[114] Philips Semiconductors. PCA5007 handshake-technology pager IC data sheet, http://www.semiconductors.philips.com/pip/PCA5007H.

[115] J. Rabaey. *Digital Integrated Circuits: A Design Perspective*. Prentice-Hall, 1996.

[116] P. Rakers, L. Connell, T. Collins, and D. Russell. Secure contactless smartcard ASIC with DPA protection. *IEEE Journal of Solid-State Circuits*, 36(3):559-565, March 2001.

[117] M. Renaudin, P. Vivet, and F. Robin. ASPRO-216: A standard-cell QDI 16-bit RISC asynchronous microprocessor. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async'98)*, pages 22-31. IEEE Computer Society Press, April 1998.

[118] M. Renaudin, P. Vivet, and F. Robin. A design framework for asynchronous/synchronous circuits based on CHP to HDL translation. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 135-144, April 1999.

[119] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key crypto systems, June 1978.

[120] M. Roncken. Defect-oriented testability for asynchronous ICs. *Proceedings of the IEEE*, 87(2):363-375, February 1999.

[121] C.L. Seitz. System timing. In C.A. Mead and L.A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.

[122] N.P. Singh. A design methodology for self-timed systems. Master's thesis, Laboratory for Computer Science, MIT, 1981. MIT/LCS/TR-258.

[123] J. Sparsϕ, C.D. Nielsen, L.S. Nielsen, and J. Staunstrup. Design of selftimed multipliers: A comparison. In S. Furber and M. Edwards, editors, *Asynchronous Design Methodologies*, volume A-28 of *IFIP Transactions*, pages 165-180. Elsevier Science Publishers, 1993.

[124] J. Sparsϕ and J. Staunstrup. Delay-insensitive multi-ring structures. *INTEGRATION, the VLSI Journal*, 15(3):313-340, October 1993.

[125] J. Sparsϕ, J. Staunstrup, and M. Dantzer-Sfrensen. Design of delay insensitive circuits using multi-ring structures. In G. Musgrave, editor, *Proc. of EURO-DAC '92, European Design Automation Conference, Hamburg, Germany, September 7-10, 1992*, pages 15-20. IEEE Computer Society Press, 1992.

[126] R.F. Sproull, I.E. Sutherland, and C.E. Molnar. The counterflow pipeline processor architecture. *IEEE Design & Test of Computers*, 11(3):48-59, 1994.

[127] L. Stok. *Architectural Synthesis and Optimization of Digital Systems*. PhD thesis, Eindhoven University of Technology, 1991.

[128] I.E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720-738, June 1989.

[129] Synopsys, Inc. *Synopsys VSS Family Core Programs Manual*, 1997.

[130] A. Takamura, M. Kuwako, M. Imai, T. Pujii, M. Ozawa, I. Fukasaku, Y. Ueno, and T. Nanya. TITAC-2: An asynchronous 32-bit microprocessor based on scalable-delay-insensitive model. In *Proc. International Conf. Computer Design (ICCD'97)*, pages 288-294. MIT Press, October 1997.

[131] H. Terada, S. Miyata, and M. Iwata. DDMPs: Self-timed superpipelined data-driven multimedia processors. *Proceedings of the IEEE*, 87(2):282-296, February 1999.

[132] M. Theobald and S.M. Nowick. Transformations for the synthesis and optimization of asynchronous distributed control. In *Proc. ACM/IEEE Design Automation Conference*, 2001.

[133] S.H. Unger. *Asynchronous Sequential Switching Circuits*. Wiley-Interscience, John Wiley & Sons, Inc., New York, 1969.

[134] C.H. van Berkel. Beware the isochronic fork. *INTEGRATION, the VLSI journal*, 13(3): 103-128, 1992.

[135] C.H. van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.

[136] C.H. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, and F. Schalij. Asynchronous circuits for low power: a DCC error corrector. *IEEE Design & Test*, 11(2):22-32, 1994.

[137] C.H. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roneken, and E Sehalij. A fully asynchronous low-power error eorreetor for the DCC player. In *ISSCC 1994 Digest of Technical Papers*, volume 37, pages 88-89. IEEE, 1994. ISSN 0193-6530.

[138] C.H. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roneken, E Schalij, and R. van de Viel. A single-rail re-implementation of a DCC error detector using a generic standard-cell library. In *2nd Working Conference on Asynchronous Design Methodologies, London, May 30-31, 1995*, pages 72-79, 1995.

[139] C.H. van Berkel, E Huberts, and A. Peeters. Stretching quasi delay insensitivity by means of extended isochronic forks. In *Asynchronous Design Methodologies*, pages 99-106. IEEE Computer Society Press, May 1995.

[140] C.H. van Berkel, M.B. Josephs, and S.M. Nowick. Scanning the technology: Applications of asynchronous circuits. *Proceedings of the IEEE*, 87(2):223-233, February 1999.

[141] C.H. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proc. European Conference on Design Automation (EDAC)*, pages 384-389, 1991.

[142] C.H. van Berkel, C. Niessen, M. Rem, and R. Saeijs. VLSI programming and silicon compilation. In *Proc. International Conf. Computer Design (ICCD)*, pages 150-166, Rye Brook, New York, 1988. IEEE Computer Society Press.

[143] H. van Gageldonk. *An Asynchronous Low-Power 80C51 Microcontroller*. PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, September 1998.

[144] H. van Gageldonk, D. Baumann, C.H. van Berkel, D. Gloor, A. Peeters, and G. Stegmann. An asynchronous low-power 80c51 microcontroller. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 96-107. IEEE Computer Society Press, April 1998.

[145] P. Vanbekbergen. *Synthesis of Asynchronous Control Circuits from Graph-Theoretic Specifications*. PhD thesis, Catholic University of Leuyen, September 1993.

[146] V.I. Varshavsky, M.A. Kishinevsky, V.B. Marakhovsky, V.A. Peschansky, L.Y. Rosenblum, A.R. Taubin, and B.S. Tzirlin. *Self-timed Control of Concurrent Processes*. Kluwer Academic Publisher, 1990. V.I.Varshavsky Ed., (Russian edition: 1986).

[147] T. Verhoeff. Delay-insensitive codes - an overview. *Distributed Computing*, 3(1):1-8, 1988.

[148] A.J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. In *IEEE Transactions on Information Theory*, volume 13, pages 260-269, 1967.

[149] P. Viviet and M. Renaudin. CHP2VHDL, a CHP to VHDL translator - towards asynchronous-design simulation. In L. Lavagno and M.B. Josephs, editors, *Handouts from the ACiD-WG Workshop on Specification models and languages and technology effects of asynchronous design*. Dipartemento di Elettronica, Polytecnico de Torino, Italy, January 1998.

[150] J.F. Wakerly. *Digital Design: Principles and Practices, 3/e*. Prentice-Hall, 2001.

[151] N. Weste and K. Esraghian. *Principles of CMOS VLSI Design – A systems Perspective, 2nd edition*. Addison-Wesley, 1993.

[152] Rik van de Wiel. High-level test evaluation of asynchronous circuits. In *Asynchronous Design Methodologies*, pages 63-71. IEEE Computer Society Press, May 1995.

[153] T.E. Williams. *Self-Timed Rings and their Application to Division*. PhD thesis, Department of Electrical Engineering and Computer Science, Stanford University, 1991. CSL-TR-91-482.

[154] T.E. Williams. Analyzing and improving latency and throughput in selftimed rings and pipelines. In *Tau-92:1992 Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*. ACM/SIGDA, March 1992.

[155] T.E. Williams. Performance of iterative computation in self-timed rings. *Journal of VLSI Signal Processing*, 6(3), October 1993.

[156] T.E. Williams and M.A. Horowitz. A zero-overhead self-timed 160 ns. 54 bit CMOS divider. *IEEE Journal of Solid State Circuits*, 26(11):1651-1661, 1991.

[157] T.E. Williams, N. Patkar, and G. Shen. SPARC64: A 64-b 64-activeinstruction out-of-order-execution MCM processor. *IEEE Journal of Solid-State Circuits*, 30(11): 1215-1226, November 1995.

[158] J.V. Woods, P. Day, S.B. Furber, J.D. Garside, N.C. Paver, and S. Temple. AMULET 1: An asynchronous ARM processor. *IEEE Transactions on Computers*, 46(4):385-398, April 1997.

[159] C. Ykman-Couvreur, B. Lin, and H. de Man. Assassin: A synthesis system for asynchronous control circuits. Technical report, IMEC, September 1994. User and Tutorial manual.

[160] K.Y. Yun and D.L. Dill. Automatic synthesis of extended burst-mode circuits: Part II (automatic synthesis). *IEEE Transactions on Computer-Aided Design*, 18(2):118-132, February 1999.

索　引