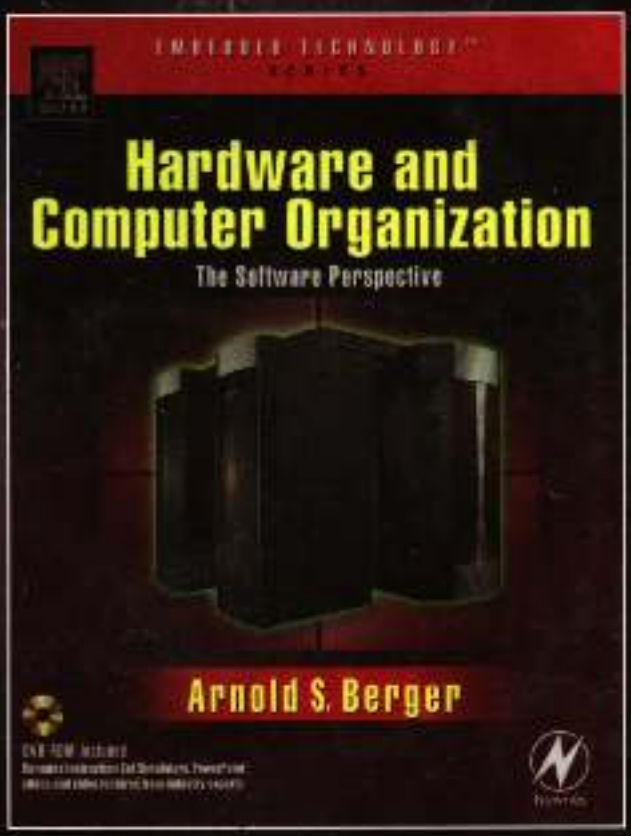
计 算 机 科 学 丛 书

计算机硬件及组成原理

(美) Arnold S. Berger 著 吴为民 喻文健 邓澍军 伍绍贺 译
卡耐基-梅隆大学 清华大学



Hardware and Computer Organization
The Software Perspective

机械工业出版社
China Machine Press

计算机硬件及组成原理

本书从实用角度详细介绍了现代微处理器的体系结构，旨在为读者揭开现代嵌入式计算机系统和PC机的神秘面纱，帮助读者了解这些在日常生活中已经无处不在的复杂机器。书中解释了硬件和软件如何协同作用来完成现实世界中的各项任务。

与其他类似简单演示如何设计计算机硬件的图书不同，本书从软件开发者的角度出发，全面分析了整个计算机，重点讲解了计算机的优势和弱点，解释了如何处理存储器问题，如何写出能直接与底层硬件交互并充分利用底层硬件的高效汇编代码。

此外，本书还介绍了从简单的嵌入式应用的8位微处理器转向PC和工作站工作时应如何进行决策，这在同类图书中独树一帜。同时，书中还阐明了代码行为和机器操作之间的联系，以帮助读者更好地理解计算机在速度和资源方面的局限性。

本书特点：

- 采用目前最常见的三种计算机体系结构作为示例：Freescale 68000 、Intel i86和ARMv3。
- 内容讲解非常直观——书中包含多种简图和图表。
- 汇聚作者在业界多年的实际经验和敏锐的洞察力。

本书附带光盘内容包括：

- 业界多位知名专家关于硬件设计和开发的11个视频讲座。
- 课件使用的幻灯片。
- 三种示例体系结构的指令系统仿真器。

作者简介

Arnold S. Berger 是华盛顿-波泰尔（Washington-Bothell）大学计算和软件系统系的高级讲师，拥有康奈尔大学的学士和博士学位。Berger博士曾担任Applied Microsystems公司研发部门的主管，Advanced Micro Devices公司嵌入式工具的营销经理和惠普公司的研发项目经理。Berger博士已发表了40多篇关于嵌入式系统的论文，持有三项专利，并且是畅销书《Embedded Systems Design: An Introduction to Processes, Tools and Techniques》的作者。

本书译自原版Hardware and Computer Organization: The Software Perspective，并由Elsevier授权出版

上架指导：计算机/计算机体系结构

ISBN 978-7-111-21018-4

投稿热线：(010) 88379604
购书热线：(010) 68995259, 68995264
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com

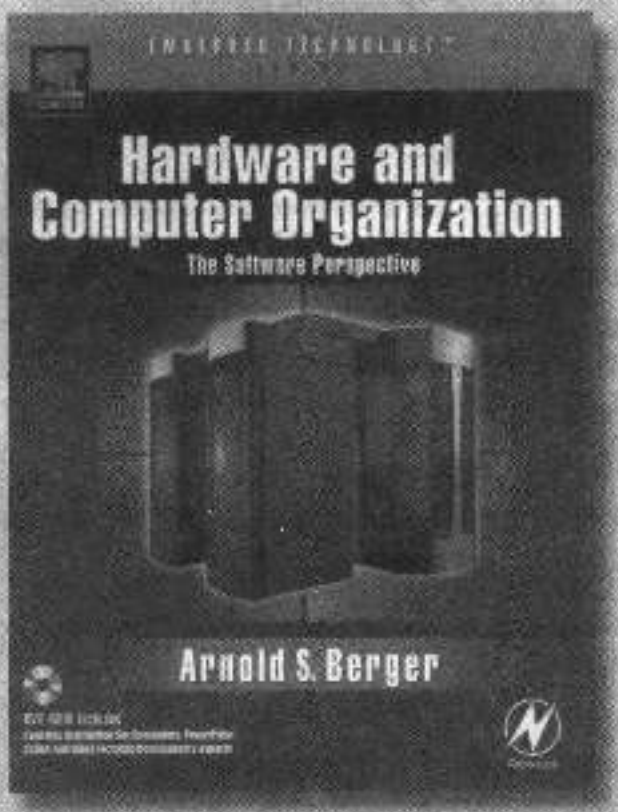
网上购书：www.china-pub.com

ISBN 978-7-111-21018-4
定价：55.00 元（附DVD光盘）

计 算 机 科 学 丛 书

计算机硬件及组成原理

(美) Arnold S. Berger 著 吴为民 喻文健 邓澍军 伍绍贺 译
华盛顿-波莱尔大学 清华大学



Hardware and Computer Organization

The Software Perspective

机械工业出版社
China Machine Press

本书从软件开发者角度出发，详细介绍了现代计算机体系结构，重点讲解如何处理存储器问题以及如何写出能直接与底层硬件交互并充分利用底层硬件的高效汇编代码。

本书主要讲述硬件基础和数字化设计，涵盖现代计算机操作系统下硬件开发的各种元素，从汇编语言讨论软件设计，从宏观角度探讨计算机体系结构，并着重探讨了CISC和RISC两种微处理器体系结构。

本书适合作为高等院校相关专业课程教材，也可供软件开发人员参考。

Arnold S. Berger : Hardware and Computer Organization: The Software Perspective.
ISBN-10 : 0750678860
ISBN-13 : 978-0750678865
Copyright © 2005 by Elsevier Inc. All rights reserved.
Authorized Simplified Chinese translation edition published by the Proprietor.
ISBN-10 : 981-259-438-8
ISBN-13 : 978-981-259-438-9
Copyright © 2007 by Elsevier (Singapore) Pte Ltd. All rights reserved.
Printed in China by China Machine Press under special arrangement with Elsevier (Singapore) Pte Ltd. This edition is authorized for sale in China only, excluding Hong Kong SAR and Taiwan. Unauthorized export of this edition is a violation of the Copyright Act. Violation of this Law is subject to Civil and Criminal Penalties.

本书简体中文版由机械工业出版社与Elsevier (Singapore) Pte Ltd.在中国大陆境内合作出版。本版仅限在中国境内（不包括中国香港特别行政区及中国台湾地区）出版及标价销售。未经许可之出口，视为违反著作权法，将受法律之制裁。

版权所有，侵权必究。
本书法律顾问　北京市展达律师事务所

本书版权登记号：图字：01-2005-4837

图书在版编目（CIP）数据

计算机硬件及组成原理 /（美）格吉尔（Berger, A. S.）著；吴为民等译. -北京：机械工业出版社，2007.5
（计算机科学丛书）
书名原文：Hardware and Computer Organization : The Software Perspective
ISBN 978-7-111-21018-4

Ⅰ.计…　Ⅱ.①格…　②吴…　Ⅲ.硬件　Ⅳ.TP303

中国版本图书馆CIP数据核字（2007）第027346号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）
责任编辑：李东震
北京京北制版厂印刷 · 新华书店北京发行所发行
2007年5月第1版第1次印刷
184mm × 260mm · 25.25印张
定价：55.00元（附DVD光盘）

凡购本书，如有倒页、脱页、缺页，由本社发行部调换
本社购书热线：（010）68326294

机械工业出版社 华章公司
Huazhang Graphics & Information Co., Ltd

教师服务登记表

尊敬的老师:

您好！感谢您购买我们出版的_____教材。

机械工业出版社华章公司本着为服务高等教育的出版原则，为进一步加强与高校教师的联系与沟通，更好地为高校教师服务，特制此表，请您填妥后发回给我们，我们将定期向您寄送华章公司最新的图书出版信息，为您的教材、论著或译著的出版提供可能的帮助。欢迎您对我们的教材和服务提出宝贵的意见，感谢您的大力支持与帮助！

个人资料（请用正楷完整填写）

教师姓名		□先生 □女士	出生年月		职务		职称: □教授 □副教授 □讲师 □助教 □其他
学校			学院			系别	
联系电话	办公: 宅电: 移动:			联系地址及邮编			
				E-mail			
学历		毕业院校		国外进修及讲学经历			
研究领域							

主讲课程	现用教材名	作者及出版社	共同授课教师	教材满意度
课程: □专 □本 □研 □MBA 人数: 学期: □春□秋				□满意 □一般 □不满意 □希望更换
课程: □专 □本 □研 □MBA 人数: 学期: □春□秋				□满意 □一般 □不满意 □希望更换
课程: □专 □本 □研 □MBA 人数: 学期: □春□秋				□满意 □一般 □不满意 □希望更换

备注				
	已出版著作		译著	
	著书 计划	方向一 方向二		
	是否愿意从事翻译工作 □是 □否		翻译方向	
	意见和建议			

填妥后请选择以下任何一种方式将此表返回: （如方便请赐名片）

地 址: 北京市西城区百万庄南街1号 华章公司营销中心 邮编: 100037
电 话: (010) 68353079 88378995 传真: (010) 68995260
E-mail: hzedu@hzbook.com marketing@hzbook.com 图书详情可登录http://www.hzbook.com网站查询

专家指导委员会

（按姓氏笔画顺序）

尤晋元	王　珊	冯博琴	史忠植	史美林
石教英	吕　建	孙玉芳	吴世忠	吴时霖
张立昂	李伟琴	李师贤	李建中	杨冬青
邵维忠	陆丽娜	陆鑫达	陈向群	周伯生
周克定	周傲英	孟小峰	岳丽华	范　明
郑国梁	施伯乐	钟玉琢	唐世渭	袁崇义
高传善	梅　宏	程　旭	程时端	谢希仁
裘宗燕	戴　葵			

译者序

展现在您面前的是这样一本书：它讲的是计算机硬件和体系结构的问题，而且也确实覆盖了这个领域的几乎所有内容，但是，即使是对计算机硬件知识知之甚少的人也能看懂它并从中受益。因为本书是从软件这个视角来介绍计算机体系结构的。

一个软件设计者，如果对其要编程的硬件和体系结构的知识缺乏了解，就失去了很多提高软件性能的手段。然而，要让他们去系统地学习计算机体系结构的知识，再运用这些知识来提高软件性能，则存在一定的困难。因为现今的教科书大都是针对学习体系结构的人而编写的，所以这些教材基本上都是力图完整系统地传授体系结构知识的，并没有考虑一个软件设计者如何利用体系结构的特点设计出高性能的程序。也就是说，对于计算机体系结构，软件开发者在掌握知识和运用知识之间存在着一个鸿沟，本书出现的意义就是填补了这个鸿沟。

本书的讲授风格有别于一般的教科书，它的讲解不是刻板的，而是以一种轻松和诙谐的口吻进行的，就像课堂授课录音的书面记录，读者可以自然地跟进而不会感到疲倦。书中的内容丰富并配有大量的实例，每章后面还有习题并提供了答案（书后和网上）。本书的作者具有丰富的实践经验和教学经验，读者在阅读本书的过程中随处都能体会到。

本书的翻译是4位译者共同努力的结果，其中，吴为民翻译了序言及第1、5、6、7、9、12、15、16章，喻文健翻译了第2、3、4章，邓澍军翻译了第8、10、11章，伍绍贺翻译了第13、14章。最后由吴为民统一校对定稿。对于书中出现的术语，我们基本是按最常用的译法和具体情况进行取舍的。对于原书中出现的错误，我们也完全按照其网站上发布的勘误表进行了改正。尽管如此，疏漏在所难免，我们欢迎和感谢来自读者的任何批评指正。

译者于清华园

2007年1月

前　言

谢谢你购买我的书。我希望你发现本书内容充实且易读，至少，当我着手写这本书时，这是我的目标之一。

本书是我一直在Washington-Bothell大学计算与软件系统系所教授的课程的成果。该课程（CSS 422），即“硬件和计算机组成”（Hardware and Computer Organization），是我系本科学生必修的核心课程之一，也是我系课程表中唯一必修的体系结构课程。当我们的学生在学习算法和数据结构、相关的语言、数值方法以及操作系统时，这是他们唯一一次接触到这些知识外壳下真实的原理。由于华盛顿大学是一学年四学期制，所以我就面临着这样一个艰难的挑战：要在10周左右的时间内讲授尽可能多的计算机体系结构的知识。

本书核心的材料是用5年多的时间制成的大约500张Microsoft PowerPoint幻灯片。后来，我将幻灯片中的材料转换成了HTML，这样我就也能通过远程教育（DL）的形式讲授课程了。自从1999年秋首次讲授该课程，我每个学年都要讲上3~4遍。我还通过DL讲授了3次，取得了很好的效果。事实上，DL学生在总体上与在课堂听讲的学生做得一样好，因此，如果你觉得没有时间选修这门课，那么可以用本书来自学这门课程。

本书适合作为从二年级到四年级中在计算机体系结构方面的第一门课。本书相当地独立，所以应该能成为计算机系学生所需学习的唯一硬件课程，掌握了这门课程就能理解他们所编写的代码的涵义。在Washington-Bothell大学（UWB），这门课程主要是讲授给四年级学生的。作为教员我们发现，通过在其他课程中学习编程概念，达到一定的熟练程度，有助于学生轻松地转向学习低级编程技术。如果本书用于低年级学生，就需要分配额外的时间来熟练掌握汇编语言的编程概念。例如：在介绍某些汇编语言分支和循环结构时，高年级学生很容易掌握这些结构与WHILE、DO-WHILE、FOR及IF-THEN-ELSE结构的相似性，但低年级学生则可能需要更多具体的例子来领会这种相似性。

为什么要写一本关于计算机体系结构的书？在讲授该课程的5年多时间中，我曾4次更换教材。在学期末，当我主持一个非正式的课程听取学生汇报时，他们严厉地批评了我使用过的每本书。几乎每个计算机科学系的学生在体系结构课堂上使用的标准教材都与他们的需要无关。这些学生中的绝大多数在研究生阶段都不会继续学习体系结构，也不会为Intel或AMD设计计算机。因此，他们需要的是理解计算机体系结构及其支撑硬件，以便于编写能在机器上运行的高效的、无缺陷的代码。最近，我确实发现一本教材，至少以我认为应该的方式接近了主题内容，但是我发现该教材仍在几个关键的领域存在不足。从正面看，换成新的教材确实能消除来自学生的抱怨，同时也强化了我的这个观点：并不只我一个人看到了对具有不同视角的教材的需求。遗憾的是，这个教材尽管是一个重大的改进，但仍没有涵盖我认为非常重要的几个领域，因此我下决心写一本教材，本教材确实是以新视角写成的，且没有损失我认为的精髓。

由于UMB校园距华盛顿州Redmond的微软总部不到10英里，所以我们受到了微软文化的强烈影响，这并不奇怪。我的大部分学生只为Windows和Intel体系结构写程序，该体系结构的设计者会让你相信这些计算机是无限快的机器，有无穷的资源。你如何反驳这种观点？

我的学生经常带有挫折感地抱怨，“为什么你要我学这些？”这种情况通常都会发生在期中考试前后。由于我们的校园与波音公司建造737、757飞机的地点华盛顿州Renton和建造宽体767、777飞机的地点华盛顿州Everett大致等距离，用飞机工业做类比通常非常有效，因此我用这个例子简单地回答了他们的问题：“你愿意乘坐一个由某个对飞行原理毫无所知的人设计的飞机吗？”有时这种反问式的回答会很奏效。

本书分为4个主要的主题领域：

1. 硬件和异步逻辑介绍。
2. 同步逻辑、状态机和存储器组织。
3. 现代计算机体系结构和汇编语言编程。
4. 输入/输出、计算机性能、存储器层次，以及计算机组成的未来发展方向。

这些主题领域之间没有明确的界限划分，而且后面章节的主题材料是建立在前面章节的知识基础之上的。然而，我已力图限制这种相互依赖的关系，因此后面的章节可以根据时间和教学提纲的要求进行取舍。

每章末尾都有一些习题，其中奇数号习题的答案位于附录中，偶数号习题的答案则可通过教师资源网址http://textbooks.elsevier.com/0750678860得到。

我们在课本中采取了由下而上的方法描述硬件。正如遗传学家用仅包含腺嘌呤、胞嘧啶、鸟嘌呤、胸腺嘧啶（分别简写为A、C、G、T）4种核苷酸的DNA分子就能描述最复杂的有机生命一样，我们用与门（AND）、或门（OR）、非门（NOT）、三态门（TRI-STATE）这4种逻辑构件块就能描述最复杂的计算机或存储系统。严格地说，三态门不是一个类似于与门的逻辑构件块，它更像“粘结剂”，使我们能以某种方式将计算机元件互连，从而避免过高的复杂性。而且，我确实喜欢DNA这个类比，所以我们需要用4个电子构件块与A、C、G、T相类比。

我曾经给一组中学教师做过一个报告，这些教师那时正努力地想在夏季休息期间修一些在职学分。当我为惠普公司的逻辑系统部门工作时，我还是位于科罗拉多Springs的空军学院校区的志愿者。这些教师中没有一个懂计算机，我要用两个小时教给他们这门技术的一些感性知识。我决定从亚里士多德和作为哲学分支的逻辑操作符开始讲解，然后用DNA做类比继续介绍寄存器概念。我似乎正逐渐让他们听懂，但他们却总在打击我的自信心，使我想在他们的点名册上签字解约。无论如何，我认为表明这样的事实是有价值的：即使是最复杂的计算机功能，也能用我们在本书第一部分所介绍的基本逻辑单元来描述。

我们将采用DNA或构件块方法贯穿于本书前半部分中的大部分。我们将从最简单的门开始，构建复合门，再从这些复合门开始，进一步提出和求解异步逻辑方程。我们将学习用布尔代数和卡诺图（Karnaugh Map, K-map）进行真值表设计和化简的方法。习题和实例强调的是将问题用一组规范化描述来陈述，然后转化为一个真值表，再由真值表转化为卡诺图，最后转化为门设计。此时，要鼓励学生使用随书带的DVD光盘中的Digital Works软件模拟器（见下文）在模拟中实际地“构建”电路。我发现这种把抽象设计和实际模拟相结合是一个极好的教学方法。

采用这个方法的好处之一是能使学生们逐渐习惯于在二进制位一级和变量打交道。虽然大多数学生熟悉C/C++的布尔结构，但是一条承载一个变量状态的线对他们来说还是相当新的概念。

介绍了建立任意复杂的异步代数函数的思想之后，我们就加入时钟和同步逻辑的概念。同步逻辑包括触发器、计数器、移位器、寄存器以及状态机。我们在该领域实际付出了很多

努力，在稍后讲到微代码和指令分解时，还要将这些概念重新介绍几次。

本书的中间部分着重介绍计算机系统的体系结构。尤其是，我们会非常关注存储器到CPU的接口。我们将利用在前面章节学到的知识设计简单的存储器系统和译码电路。我们还简要考察了存储器定时，以便于更好地理解系统设计的一些更全局化的问题。

接下来我们将转而介绍68K、ARM以及x86处理器系列的体系结构，这同时也将是对汇编语言编程的介绍。

每种处理器体系结构都单独讲解，以使读者可跳过某种结构的介绍而不会产生过多的不连贯性。

本书确实在三个体系结构中都强调了汇编语言编程，其原因有两个方面：首先，汇编语言不一定是作为课程表的一部分为计算机科学系学生讲授的，故这可能是他们在机器级接触编程的唯一机会。即使你作为计算机系的学生可能从不需要编写汇编语言程序，你也很有可能要在汇编语言级来调试C++程序的某些部分，因此，这是一个很好的学习机会。而且，通过考察这三个截然不同的指令集，我们将实际强化这种观念：一旦你理解了一种处理器的体系结构，你就能用汇编语言对其进行编程，这就引出了学习汇编语言的第二个原因。汇编语言是从软件开发者的角度学习计算机体系结构的一个很好的出发点。

我是“赛博士”（Dr. Science）的一个狂热爱好者，他经常出现在国家无线广播电台中，并在大学校园中巡回演讲，他的著名时髦语是：“我在某某学科具有硕士学位。”不过，我在赛博士的讲座上听到这样一句话：“我喜欢审视一列列的随机数字，并从中发现模式。”我记住了这句话，并经常用在我的课堂上，以描述如何能够通过看起来很随机的机器语言指令集来逐渐领会计算机体系结构。我只能想像一群摩托罗拉的CPU设计师和工程师围坐在餐馆的桌旁，桌子上的比萨饼盘子散布在各处，他们正试图为最后几条指令计算出正确的位模式，以避免得到一个膨胀的、无效率的微代码ROM表。如果你是一个学生，并且读到这里还没有任何感觉，那么也不要着急。

本书最后一部分又返回来重新考察了计算机体系结构的一般问题。我们将考察CISC与RISC，以及诸如流水线、高速缓存、虚拟存储器和存储器管理等这些现代技术。然而，最重要的主题还是计算机性能，我们将不断地返回到有关软件到硬件接口、编码方法和硬件之间的相互影响等问题上来。

本书的一个独特之处在于随书所附的DVD光盘中的材料，其中包含了下列程序，用来与本书配合使用：

- Digital Works（免费软件）：一个硬件设计和模拟工具。
- Easy68K：一个汇编器/模拟器/调试器免费软件包，用于摩托罗拉（现在免费级别的）68 000体系结构。
- x86emul：一个汇编器/模拟器/调试器共享软件包，用于x86体系结构。
- GNU ARM工具：ARM开发者工具箱，带有来自自由软件基金会（Free Software Foundation）的指令集模拟器。
- 来自11位业界专家的关于硬件设计和开发方面重要主题的视频讲座。

ARM公司有一个极好的工具套件，你可直接从ARM获得。它带有一个免费45天的评估许可证，这对于我们的课程来讲应该足够用了。遗憾的是，我未能通过谈判与ARM公司达成一个许可证协议，使我能将这些ARM工具包含在随书的DVD光盘中。该工具套件极为优秀和易用。如果你想花费一些额外的时间来考察世界上最流行的RISC体系结构，那么就直接和ARM

公司联系，友好地寻求ARM套件的拷贝吧。告诉他们是我让你这样做的。

我还在CSS 422课上广泛地采用了Easy68K汇编器/模拟器。它性能良好，并附有很多调试功能。而且，由于它是免费软件，所以无需考虑许可证和评估期限这些后顾之忧。然而，我们还要对本书中的其他工具做一些引用，因此，最好在你打算使用时再安装它们，而不是在课程开始时就安装它们，这也许是一个好主意。

DVD光盘包括的11个视频讲座，内容与本书各种主题相关，演讲者来自于计算机体系结构领域的专家。这些视频文件是在2004年UWB的沃辛顿技术捐赠基金提供的资助下制作的。每个讲座都是15到30分钟的技术讲解。我希望你花时间看一下，并将它们融汇到相关主题的学习过程中。

虽然编辑、我的学生们以及我本人都已将本书读过几遍了，但据墨菲法则的预言，本书还有存在错误的巨大可能性，因为毕竟它是软件。因此，如果你在本书中看到错误，那么请告诉我，并将你的意见发送到：aberger@u.washington.edu。我将保证这些修正意见能在我的华盛顿大学的网站（http://faculty.uwb.edu/aberger）上张贴出来。

最后一点我想说的是，课本只能做到这样。无论你是正在读本书的学生还是老师，请尽量寻找专家和原始资源。James Patterson教授在2004年7月期的《Physics Today》（今日物理）中写到：

当我们想了解某件事物时，总有一种想在课本中快速寻求答案的倾向。这常常是奏效的，但我们需要养成查看原始论文的习惯。因为课本上的内容通常是把事实经过二次或三次简化而来……

让我们开始吧。

Arnold S. Berger
Sammamish, 华盛顿

致　谢

首先，我要感谢Washington-Bothell大学计算与软件系统系的前主任William Erdly教授的赞助和支持。Erdly教授首先在1999年秋聘用我为助教授，并要求我讲授一门称为“硬件和计算机组成”的课，虽然我那时想讲的是关于嵌入式系统设计的课程。

然后Erdly教授就为我提供经费支持，将我的“硬件和计算机组成”课程中一系列PowerPoint幻灯片讲义转换成一系列HTML形式，供在线课程使用。这些内容成了本书的核心材料。

在Charles Jackels教授和Frank Cioch教授任执行主任期间，都在完善在线材料和将多媒体引入远程教育实践方面给予了我支持。他们的支持使我认识到了课堂中技术的教学价值。

我还感谢沃辛顿基金会的Richard P.和Lois M.，他们提供的2004技术资助奖使我能在全美旅行并做计算机体系结构方面的简短视频录像。我也想感谢那11位演讲者能抽出时间参与该计划。

Newnes图书出版公司的选题编辑Carol Lewis认识到了我的方法的价值，我为此感谢她。遗憾的是，在这本书最终完成之前，她就离开了Newnes。Elsevier出版社的Tiffany Gasbarrini和Borrego出版社的Kelly Johnson使本书最终得以面世，谢谢你们两位。

本书很大程度上是那些曾经学习CSS 422课程的学生所设计的。他们的期末评估和反馈对于帮助我认识到如何构思本书具有无法估量的价值。

最后，也是最重要的，我要感谢我的妻子Vivian。没有她的支持和理解，我不可能写出这样篇幅的一本书。

目　录

出版者的话
专家指导委员会
译者序
前言
致谢

第1章　硬件体系结构简介 ……1
1.1　引言 ……1
1.2　计算技术简史 ……1
1.3　数制 ……9
1.4　将十进制数转换为各种基数的数 ……20
1.5　工程符号 ……21
总结 ……22
参考文献 ……22
习题 ……22
第2章　数字逻辑简介 ……24
2.1　引言 ……24
2.2　电子门描述 ……32
2.3　真值表 ……36
总结 ……38
参考文献 ……38
习题 ……38
第3章　异步逻辑简介 ……40
3.1　引言 ……40
3.2　布尔代数定律 ……41
3.3　卡诺图 ……45
3.4　时钟和脉冲 ……50
总结 ……55
参考文献 ……55
习题 ……55
第4章　同步逻辑简介 ……58
4.1　引言 ……58
4.2　触发器 ……59
4.3　存储寄存器 ……68
总结 ……74
参考文献 ……75
习题 ……75
第5章　状态机简介 ……79
5.1　引言 ……79
5.2　现代硬件设计方法 ……96
总结 ……98
参考文献 ……98
习题 ……99
第6章　总线组织和存储器设计 ……103
6.1　总线组织 ……103
6.2　地址空间 ……115
6.3　直接存储器访问 ……128
总结 ……129
参考文献 ……130
习题 ……130
第7章　存储器组织和汇编语言编程 ……134
7.1　引言 ……134
7.2　标号 ……143
7.3　有效地址 ……147
7.4　伪操作代码 ……154
7.5　数据存储伪指令 ……155
7.6　汇编语言程序的分析 ……156
总结 ……158
参考文献 ……158
习题 ……158
第8章　汇编语言程序设计 ……162
8.1　引言 ……162
8.2　汇编语言和C++ ……175
8.3　堆栈和子程序 ……180
总结 ……186
参考文献 ……186
习题 ……186
第9章　高级汇编语言编程 ……192
9.1　引言 ……192
9.2　高级寻址模式 ……192
9.3　68000指令 ……194
9.4　移动指令 ……195
9.5　逻辑指令 ……195
9.6　其他逻辑指令 ……196

9.7 68000指令总结……199
9.8 用TRAP#15指令模拟I/O ……201
9.9 编译器和汇编器 ……203
总结 ……216
参考文献 ……216
习题 ……216
第10章 Intel x86体系结构 ……220
10.1 引言……220
10.2 8086 CPU的体系结构 ……221
10.3 数据寄存器、变址寄存器和指针寄存器…223
10.4 标志寄存器……226
10.5 段寄存器……226
10.6 指令指针（IP） ……226
10.7 存储器寻址模式……228
10.8 x86指令格式 ……231
10.9 8086指令集总结……233
10.10 数据传送指令 ……234
10.11 算术指令 ……235
10.12 逻辑指令 ……235
10.13 字符串操作 ……236
10.14 控制转移 ……237
10.15 8086体系结构的汇编语言程序设计…239
10.16 系统向量 ……241
10.17 系统启动 ……241
总结 ……241
参考文献 ……242
习题 ……242
第11章 ARM体系结构 ……244
11.1 引言……244
11.2 ARM体系结构简介 ……245
11.3 条件执行……249
11.4 桶式移位器……250
11.5 操作数大小……250
11.6 寻址模式……251
11.7 堆栈操作……253
11.8 ARM指令集 ……255
11.9 ARM系统向量 ……263
总结 ……264
参考文献 ……264
习题 ……265
第12章 与外部接口 ……266
12.1 引言……266
12.2 中断……267
12.3 异常……270
12.4 Motorola 68K的中断 ……270
12.5 模数(A/D)转换和数模(D/A)转换……274
12.6 A/D和D/A转换器的分辨率 ……286
总结 ……288
参考文献 ……288
习题 ……288
第13章 现代计算机体系结构简介 ……292
13.1 处理器体系结构，CISC、RISC及DSP…293
13.2 流水线简介……296
总结 ……305
参考文献 ……305
习题 ……306
第14章 存储器、高速缓存和虚拟存储器…308
14.1 高速缓存简介……308
14.2 虚拟存储器……321
14.3 页……323
14.4 转换旁路缓冲器（TLB） ……324
14.5 保护……325
总结 ……326
参考文献 ……327
习题 ……327
第15章 计算机体系结构的性能问题 …329
15.1 引言……329
15.2 硬件和性能……329
15.3 最佳习惯……342
总结 ……343
参考文献 ……344
习题 ……344
第16章 未来发展趋势与可重构硬件 …346
16.1 引言……346
16.2 可重构硬件……346
16.3 分子计算……354
16.4 局部时钟……355
总结 ……358
参考文献 ……358
习题 ……358
附录 奇数号习题答案……360
索引 ……382

第 1 章　硬件体系结构简介

学习目标

- 描述计算器件的演化以及大多数计算机器件的组织方式；
- 在二进制、八进制、十六进制数之间做简单的转换，并解释这些数制对计算器件的重要性；
- 表明计算机硬件的原子元件和逻辑门的使用方式，并详述支配它们操作的规则。

1.1 引言

今天，当一排计算机围绕在我们周围帮助我们管理日常生活时，我们通常会视这些为理所当然。对于正在学习计算机体系结构和数字硬件的你来说无疑是很好的理解并习以为常，而且你可能在个人计算机和工作站上已编写了数不清的程序，计算机技术已经进展到如此程度，即每个任天堂GameBoy游戏机的计算能力都百倍于用于首次执行水星太空任务的计算机系统。

1.2 计算技术简史

计算机器自出现已经历了漫长的几百年的时光。中国算盘、带有传动装置和齿轮的计算器以及第一台模拟计算机都是计算机器的例子。我们将要介绍的计算机器产生于20世纪40年代，因为第二次世界大战中的炮兵需要一种更精确的方法来计算从战舰上发射炮弹的轨迹。

今天，计算机变得如此普遍流行的主要原因是集成电路制造技术的进步。在加利福尼亚的San Jose北部和Palo Alto南部，曾经主要以橘林闻名的地方如今已被称为硅谷。硅谷是很多公司的大本营，而这些公司正是集成电路技术的推动者。Intel、AMD、Cypress、Cirrus Logic等等这些名字都在全世界家喻户晓。

大约30年前，Intel的创始人之一Gordon Moore观察到，安置于单个硅芯片上的晶体管密度每18个月翻一番。该论点自从Moore首次指出以来一直相当准确，并被公认为摩尔定律。与其他指标相比，存储器容量更适于用作说明摩尔定律准确性的例子。图1-1是存储器容量对于时间的半对数图。很多电路设计者和器件物理学家还在争论摩尔定律持续下去的可能性。晶体管不能无限地缩小，要生产如此小尺寸的硅圆片，制造者也负担不起制造设备的成本。在某个点上，量子物理学定律将开始深刻地改变这些微小晶体管的特性。

今天，我们能够将数以亿记的晶体管（即我们用来建造逻辑门的“活动”开关器件）安置在单个硅片上，边长大概为2cm。从设计计算机芯片的角度看，这个重大突破发生在Mead和Conway[1]描述了一种通过编写软件产生硬件设计的方法之后。这个称为*硅编译*（silicon compilation）的方法导致了*硬件描述语言*（hardware description language, HDL）的产生。硬件描述语言如Verilog和VHDL能够使硬件设计者编写出与C程序设计语言极其相似的程序，然后将该程序编译成一个处方（recipe），半导体制造者就用该处方制造芯片。

回到开始。第一代计算机的引擎由机械装置组成。算盘、加法机、纺织机穿孔卡读卡器属于这一类。第二代跨越了1940~1960年这段时期。这个时期用电子器件（即真空管）作为活动

器件或开关元件。即使一个微型的真空管也比硅圆片上的一个晶体管大几百万倍，其消耗的功率是晶体管的几百万倍，而其使用寿命则比晶体管小几百倍或几千倍。虽然真空管计算机比前一代的机械计算机快得多，但仍比现在的计算机慢几千倍。如果你是20世纪50年代B级科幻小说改编电影的爱好者，这些计算机就是那些充满房间，身上灯光闪烁，仪表针跳动的东西。

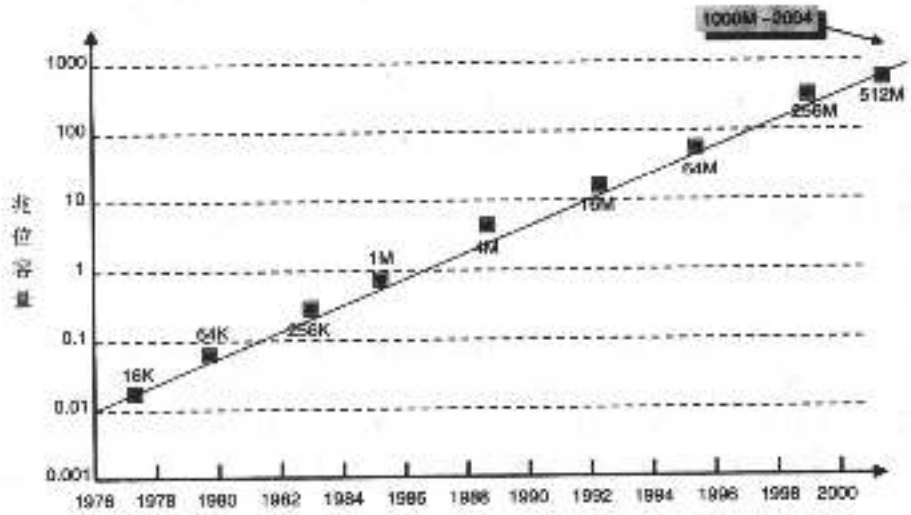


图1-1 动态随机存储器（DRAM）的容量随时间的增长。注意其半对数行为，表明了摩尔定律

第二代大致是从1960年到1968年这段时期，这个时期，晶体管取代了真空管，计算机突然开始有能力做实际工作了。诸如IBM、Burroughs、Univac这样的公司都建造了大型机，IBM 360系列就是当时大型机的典型例子。当时，Xerox的研究者也在他们的Palo Alto研究中心（Xerox PARC）进行一些关于人机接口方面的开创性工作。在那里，他们的研究成果后来演变成为计算机网络、Windows操作系统以及广泛使用的鼠标。程序员们不再用机器语言和汇编语言编程，而是开始使用FORTRAN、COBOL和BASIC。

第四代大致是从1969年到1977年，是小型计算机的时代。小型计算机是大众化的计算机，它虽然还不完全是个人计算机，但它已将计算机搬出了由穿白外套的技术人员维护的计算机房的消毒环境，搬进了我们的实验室。小型计算机还象征着集成电路（即单一封装的逻辑函数集块）的出现，它替代了诸如晶体管、电阻这样布置于印刷电路板上的单个电子零件（称为分立器件）。这段时期出现了小规模和中规模的集成电路。诸如数据设备公司（DEC）、Data General以及HP这样的公司都建造了这一代的小型计算机[2]。在这段时期中，还提出了简单的集成电路微处理器，并由Intel、Texas Instruments、Motorola、MOS Technology以及Zilog等公司将其商品化。早期最能代表这一代的微计算机器件是Intel的4004、8008和8080、Texas Instruments的9900，以及Motorola的6800。第四代的计算机语言是：汇编、C、Pascal、Modula、Smalltalk和Microsoft BASIC。

我们目前处于第五代，虽然有争论认为第五代结束于Intel 80486微处理器，奔腾（Pentium）代表第六代，但我们将忽视这个不同意见，直至它被更广泛地接受。半导体制造技术的进步是对第五代计算机特色的最佳刻画，当今半导体工艺代表了称为超大规模集成电路（Very Large Scale Integration，VLSI）的技术。下一步，甚大规模集成电路（Ultra Large Scale Integration，ULSI）既非就在眼前，也不是即将来临。AMD研究员（Fellow）Daniel Mann博士[3]最近告诉我，现代AMD Athlon XP处理器包含将近6千万个晶体管。

第五代还见证了个人计算机和操作系统的成长，后者还是该种机器的焦点。由标准操作系统控制的标准硬件平台使数以千计的开发者为这些系统编写程序。就软件而论，占支配地位的语言变为ADA、C++、Java、HTML和XML。此外，基于通用建模语言（UML）的图形设计语言开始出现。 3

对当前计算机的两种观点

现代计算机已变得更快、更强大，但是在很多年里计算机的基本体系结构本质上未变。现今我们可以对这种机器持两种等价的观点：硬件观点和软件观点。毫不奇怪，硬件观点关注的是机器，并确实考虑到软件与其存在的这种前提相关。从5万英尺远，我们的计算机看起来如同图1-2所示。

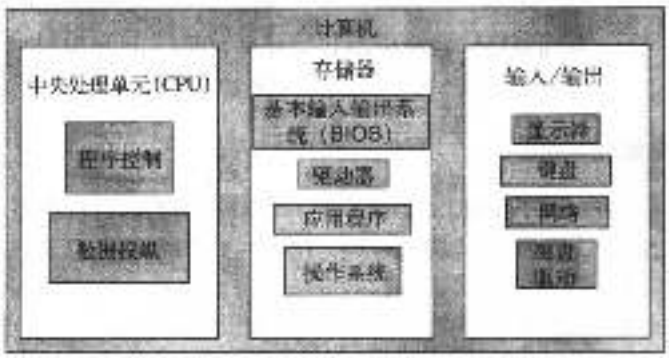


图1-2 计算机的抽象观察。三个主要的要素是：控制和数据处理器、输入和输出（即I/O器件）以及运行于机器上的程序

在本课程中，我们将主要关注CPU和存储器系统，并少许考虑驱动硬件的软件，我们将略微谈及I/O，因为I/O对于好的计算机来说是不可缺少的。

软件开发者的观点大致等价，但其视角却有所变化。图1-3从软件开发者的视角描述了计算机。注意图中显示的对系统的观察有点问题，因为用户接口与应用程序的直接通信并不总是清楚的，在很多情况下，用户接口首先与操作系统通信。然而，我们可以用稍微宽松的眼光来看待该图，将其看成信息流，而不是控制流。



图1-3 根据系统的抽象级别表示计算机

抽象级别

关于计算机设计，一个更现代的概念是抽象级别的思想，每个级别提供的是它下面级别的抽象，位于最低级别的就是硬件。为了控制硬件，就有必要生成一些小的，称为驱动程序（driver）的程序，用来实际操纵硬件的各个控制位。 4

位于驱动程序上面的是操作系统和其他系统程序，操作系统（OS）通过一组标准的应用编程接口（Application Programming Interface，API）与驱动程序通信。API提供了一种结构，通过它，抽象级别中的上一级别可与下一级别通信。这样，为了从计算机的键盘读一个字符，就要有一个低级别的驱动程序，当一个按键被敲击时，激活该驱动程序。操作系统通过其API与该驱动程序通信。

在接下来的级别中，应用软件通过系统API与操作系统通信，这又是一个对较低级别的抽象，

使得硬件和驱动程序在行为上的个体差异可以被忽略。然而，我们需要稍加注意，不要对图1-3的观点理解得过于字面化。我们还可以争辩说，应用程序和操作系统级别应该颠倒过来，因为用户也通过操作系统级别与应用程序进行交互。这样，鼠标和键盘输入实际上是通过操作系统传送给应用程序的，而不是直接从用户传送给应用程序的。无论如何，你现在了解了这些概念。

可将以台式PC机为代表的计算机硬件看作是由4个基本部件组成的：

1. 输入设备：包括鼠标、键盘、麦克风、磁盘、调制解调器及网络等组件。
2. 输出设备：包括显示器、磁盘、调制解调器、声卡和喇叭以及网络等组件。
3. 存储系统：包括内部和外部高速缓存（cache）、主存储器、视频存储器及磁盘。
4. 中央处理器（CPU）：包括算术和逻辑单元（ALU）、控制系统及总线。

总线

总线是计算机的神经系统，它们连接着计算机内外的各种功能块。在计算机内部，一个总线就是一组类似的信号线，因而，你的奔腾处理器就有一个32位地址总线和一个32位数据总线。以总线结构来看，这意味着有两束线，每束都包含着用于完成共同功能的32个单独信号线。我们将在后续课程中更深入地讨论总线。

一台典型的计算机有3个总线：一个用于存储器地址，一个用于数据，一个用于状态（管理和控制）。产业界的标准总线有PCI、ISA、AGP、PC-105、VXI，等等。由于为这些产业标准总线所做的信号定义和时序要求是由维护这些标准的团体谨慎地控制的，所以来自不同制造商的硬件设备一般均可正确地工作并可互换。有些总线相当简单，只有一条线，但传向该线的信号却相当复杂，以致需要特殊的硬件和标准协议来理解信号。这种类型总线的例子有：通用串行总线（USB）、小型计算机系统接口总线（SCSI）、Ethernet以及Firewire。

5

存储器

从软件开发者的观点看，存储系统是计算机最显而易见的部件。如果没有存储器，我们将无法处理漂浮不定的指针问题。但是还有另外一个方面：计算机存储器是存储程序代码（指令）和变量（数据）的地方。我们可以做一个关于指令和数据的简单类比。考虑一个烤蛋糕的处方，该处方本身是一组告诉我们如何做蛋糕的指令，数据代表指令要对其施以操作的配料。此时，如果没有面粉可供筛选，那么筛选面粉的指令就是没有意义的。

我们也可基于速度将存储器描述成层次化的。这里速度的意思是当计算机需要时，能以多快的速度从存储器中检索到数据。最快的存储器也是最昂贵的，因此，当存储器访问时间变慢时，每个位的成本就会降低，我们就能得到更多的存储器。最快的存储器也是最靠近CPU的存储器。因此，CPU就可能会有少量的片上数据寄存器或存储地址，几千的片外cache存储器地址，几百万的主存储器地址，以及几十亿的磁盘存储器地址。最快的片上存储器的访问时间大约是最慢的硬盘存储器的访问时间的一万分之一。两种存储器的成本比率有点难计算，因为最快的半导体存储器是片上cache存储器，而你不能在微处理器本身以外单独购买它。然而，如果我们要估计PC机中每10亿字节主存储器的成本与每10亿字节硬盘存储器的成本（计入邮寄折扣）的比率，则会发现在成本上，具有20~40ns平均访问时间的最快的半导体存储器比具有1ms平均访问时间的硬盘存储器多300倍。

今天，由于PC产业的规模经济，存储器便宜得难以置信。一个具有512MB存储地址容量的标准存储器模块（SIMM）成本大约为60美元。一种称为*动态随机存储器*（dynamic random access memory, DRAM）的存储技术占据着PC存储器的统治地位。DRAM有若干种变型，我

们将在稍后进行更深入的讨论。DRAM的特征是：它必须被经常地访问，否则将丢失存储数据。这迫使我们建造高度专门化的复杂的硬件来支持存储器系统与CPU的接口。这些器件包含在支持芯片集（support chipset）中，对于现代PC来讲，支持芯片集已经变得与CPU同样重要了。为什么要采用这些复杂的存储器呢？DRAM天生就有很大的密度，能保存多达512兆位的信息。为达到这个密度，访问和控制的复杂性就转移到了芯片集中。

静态RAM（SRAM）

我们将要关注的是一种称为*静态随机存储器*（static random access memory, SRAM）的存储器。SRAM器件的每一个单元都比DRAM的更复杂，但对该器件的总体操作更易于理解，故我们在讨论中将关注这类存储器。静态随机存储器即SRAM这个词汇的含义包含如下几个方面：

1. 我们可以从芯片中读取数据或者将数据写入芯片。
2. 一旦相应的单元地址被提交给芯片，芯片中的任何存储单元就皆可在任何时间被访问。 6
3. 只要存储器加上电源，我们就只需要提供SRAM单元的地址，再加上一个读或者写信号，就能够访问或修改单元中的数据。这与DRAM单元的维护数据完整性的要求有很大不同。我们将在后面章节中非常详细地讨论这一点。

有了RAM存储器，就不需要搜索完所有先前的存储器单元后才到达你要读数据的单元了。换句话说，从RAM的最后单元读出数据和从最初单元读出数据一样快。与此相反，磁带备份设备必须查完整个磁带才能检索到最后的数据。这就是采用“随机访问”这个术语的原因。还需要注意的是，当我们讨论一般意义上的SRAM或DRAM时，如第1项和第2项所列，我们就只用术语RAM而不做SRAM或DRAM的区分。

存储器层次

图1-4所示为实际的存储器层次，从中我们可以看到各级存储器元件是如何在大小上呈指数增长的同时，在访问时间上呈指数下降的。与CPU最靠近的存储器最小，典型值位于1KB数据（1 000个8位字符）到1MB数据（1 000 000字符）之间。这些片上cache存储器的访问时间可小到1/2ns到1ns（1秒的10亿分之一）。作为一个基准，光在1ns内能在空气中传播大约1英尺。我们称该cache为初级cache，或L1 cache。

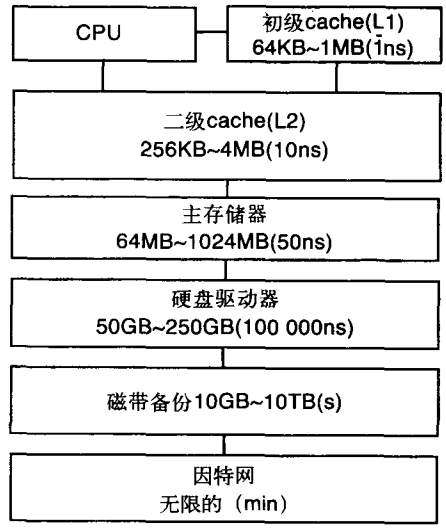


图1-4 存储器层次。注意存储器大小和访问时间之间的反比关系

在L1 cache之下就是二级cache，或叫L2 cache。在当今的奔腾类处理器中，L2 cache通常置于处理器芯片本身上。事实上，如果你揭开一个奔腾或Athlon处理器的盖子，并在显微镜下观看硅片，你就会惊讶地发现，占有芯片面积百分比最大的是cache存储器。

二级cache位于计算机的主存储器之上。主存储器就是你能在计算机商店买到的“存储器条”，用以提高计算机的性能。这种存储器比片上cache存储器慢很多，但你通常可拥有比片上存储器大得多的主存储器。你可能想知道为什么加入更多存储器就会得到更好的性能，为了理解这一点，我们考虑一下主存储器的下一个级别——硬盘驱动器。硬盘驱动器有很大的容量，但这要以牺牲速度为代价。硬盘是一

种机电设备，数据存储于旋转盘上，即使以每分钟7200转的速度旋转，仍需要宝贵的时间使
7 正确的数据处于磁盘读头之下。而且，数据被组织成独立的磁道，这些磁道分布在多个盘片的每一面上。为访问正确的数据，磁盘头必须在磁道间移动。这也要花费时间。

现在，无论你的操作系统是MAC O/S、Linux还是Windows中的一种，都可以利用硬盘作为一个便利的交换场所，将特定时间点不适合存在于主存储器的程序和数据交换出去。这样，如果你在计算机屏幕上开了若干个窗口，且计算机只有64MB的主存储器，那么你就会看到那个沙漏形状的光标经常出现，因为操作系统正不断地将不同的应用交换出/入主存储器。从图1-4看到，硬盘与主存储器访问时间的比率能达到10 000 : 1，因此任何时候我们访问硬盘都要等待。这里的道理就是：你所能给予计算机在性能上的最大提高就是尽可能多地加入存储器。

最后，图1-4中有几个可能使你感到陌生的符号，我们将在适当的课程中详细讨论，但为避免你对其含义感到疑惑，这里先给出一点说明：

符号	名　称	含　义
ns	纳秒（nanosecond）	1秒的10亿分之一
KB	千字节（kilobyte）	2^{10}或1024个8位字符（字节）
MB	兆字节（megabyte）	2^{20}或1 048 576个字节
GB	吉字节（gigabyte）	2^{30}或1 073 741 824个字节
TB	太字节（terabyte）	2^{40}或1 099 511 627 776个字节

希望这些数字很快就能被你所熟知，因为它们是现代计算机技术的衡量尺度。但有一点要注意，kilo（千）、mega（兆）、giga（吉）以及tera（太）这些术语经常承载了多种含义，有时，它们会被用于严格的科学意义，分别代表乘法因子10^3、10^6、10^9以及10^{12}的速记方式。那么，你如何才能知道正在使用的是计算机语言方式（2^{10}、2^{20}、2^{30}、2^{40}），还是传统科学和工程的意义呢？这是个好问题。有时，区分不是那么明显，就可能搞错。例如，每当涉及存储器大小和相关问题时，我们几乎总是采用以2为基数的意思。可是，也不总是这样。硬盘驱动器虽然是存储设备，却采用这些术语的工程意义。因此，一个老式的1GB硬盘驱动器与1GB存储器所拥有的数据并不是等量的，因为术语“giga”采用了两种不同的意义。总之，本书中的术语一般是取以2为基数的意思，除非另外说明。在实际中，*读者应自己小心*。

硬盘驱动器

让我们再稍微深入观察一下硬盘驱动器的动力学问题。磁盘驱动器是工程技术的奇迹。多年以来，电子工业分析家都在预测硬盘驱动器的让位问题。在成本上与硬盘驱动器一样很有效的经济型的半导体存储器总是“一些年之后的事情”。然而，磁盘驱动器制造商无视这些权威评论，而是继续提高磁盘驱动器的容量和性能，增强可靠性，降低成本。目前，一般的磁盘驱动器1GB存储容量的成本大约是60美分。

8 考虑一下现代的高性能磁盘驱动器。特别地，让我们考察一下来自Seagate Technology[4]的ST3146807LC型磁盘驱动器。以下是该驱动器的相关规范：

- 旋转速度：10 000 rpm
- 接口：Ultra320 SCSI
- 盘片数/磁头数：4/8
- 格式化容量（512字节/扇区）146.8 : GB

- 柱面数：49 855
- 每个驱动器扇区数：286 749 488
- 外部传输速率：320MB/s
- 道间寻找时间 读/写（ms）：0.35/0.55
- 平均寻找时间 读/写（ms）：4.7/5.3
- 平均潜伏期（ms）：2.99

这些都是什么意思呢？图1-5给出了上述Seagate Cheetah硬盘驱动器的一个十分简化的示意图。该硬盘由4个铝盘组成，铝盘的每一面都有磁性材料涂层用于记录存储的数据。在每个盘的上方有一个微小的检拾器，或者叫磁头（head），通过气垫浮动于磁盘上。从真实的意义上说，磁头是以远小于人的毛发粗细的距离在盘的表面上飞行，因此，当磁盘崩溃时，其结果就与飞机坠毁时类似。无论是哪种情况，飞机或读/写头都是失去提升力，碰撞到地面或磁盘表面上。当这种情况发生时，磁性材料会被毁坏，磁盘将无法使用。

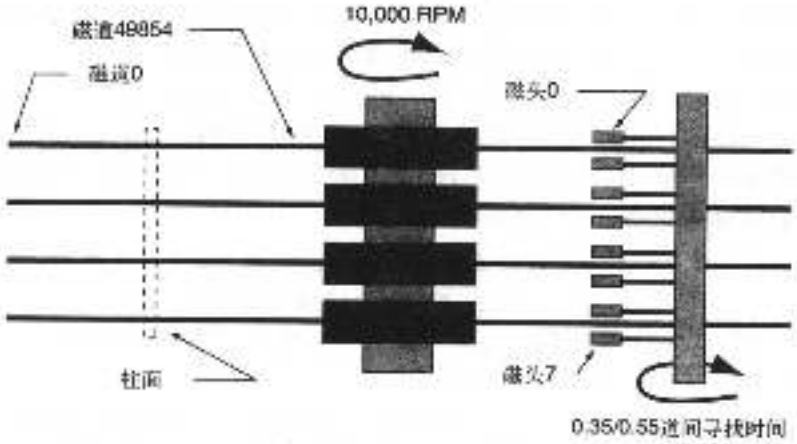


图1-5 Seagate Cheetah硬盘驱动器的示意图

每个表面包含49 855个同心磁道，每个磁道都是分离的，不与相邻磁道连接，就像一个螺旋。这样，为了从一个磁道移动到另一个相邻的磁道，磁头必须稍微移动一点距离。所有磁头都连接到一个共同的轴，该轴能迅速精确地将磁头转动到新位置。由于所有表面的磁道是垂直排列的，所以我们称其为柱面（cylinder），因此，4个盘的柱面0包含8个磁道。

现在，我们如何得到一个146.8GB的驱动器呢？首先，每个盘包含两个表面，而我们有4个盘，所以驱动器的总容量就是一个表面容量的8倍。每个扇区可容纳512字节数据。将总扇区数除以8，我们看到每个表面有35 843 686个扇区。将该数再除以49 855，我们看到每个磁道大约有719个扇区。有趣的是实际数字是每个磁道718.96个扇区，那么，为什么不采用这个值呢？换句话说，我们怎样才能使用一个分数值的每磁道扇区数呢？ 9

有很多可能性，我们无需详述。无论如何，一种可能性是磁盘表面上每个磁道的扇区数不均匀，因为当从磁盘中心向外移动时，扇区间隔会发生变化。在CD-ROM或DVD驱动器中，这种不均匀可以通过在激光移进和移出时改变驱动器的旋转速度来矫正，但是，由于硬盘以固定速率旋转，所以当我们从内磁道向外磁道移动时，改变记录密度最为合理。

让我们回过头来继续计算，如果每个磁道容纳719个扇区，每个扇区512个字节，则每个磁道能容纳368 128字节的数据。由于每个柱面有8个磁道，所以每个柱面就能容纳2 945 024字节。至此，就得到了这个大数字。由于有49 855个柱面，我们得到的总容量

就是146 824 171 520字节，即146.8GB。

在我们离开磁盘驱动器这个主题之前，让我们再考虑一个问题。考察上述的硬盘驱动器的规范，我们会发现访问时间是以毫秒（ms）即千分之一秒作为度量单位的。这样，如果你的数据扇区广布于磁盘各处，那么访问512字节的每个块就很容易用到几秒种的时间。将这个时间与访问存储于主存储器中的数据所需的时间进行比较，就很容易发现为什么硬盘驱动器比主存储器慢10 000倍了。

复杂指令集计算机（CISC）体系结构和精简指令集计算机（RISC）体系结构

目前，有两种占统治地位的计算机体系结构：复杂指令集计算机（CISC）体系结构和精简指令集计算机（RISC）体系结构。CISC的代表就是通常所说的冯·诺依曼体系结构，由普林斯顿大学的约翰·冯·诺依曼所发明。在冯·诺依曼体系结构中，指令存储器和数据存储器共享同一个物理存储空间，这可能会导致产生一种称为*冯·诺依曼瓶颈*（von Neumann bottleneck）的情况，即外部地址和数据总线是同一个，必须提供双重服务：为执行程序而从存储器向处理器传送指令，为存储和检索程序变量而从/向存储器移动数据。

当处理器在移动数据时，将无法获取下一条指令，反之亦然。对这种困境的一种解决方法就是，引入分离的片上指令和数据cache，我们在后面将会看到。Motorola的68000处理器及其后继和Intel的8086处理器及其后继都是典型的CISC处理器。在后面学习到流水线时，我们将更深入地考察CISC处理器和RISC处理器的区别。

注释：你经常会看到处理器被表示为80x86或680x0，其实，这里的“x”是占位符，它表示一个系列的器件。因此，80x86（通常写为x86）代表：8086、80186、
10 *80286、80386、80486和80586（第一个奔腾处理器）。Motorola的处理器就是：*
68000、68010、68020、68030、68040和68060。

哈佛大学的Howard Aiken设计了另一种计算机体系结构，我们现在通常将它归于精简指令集计算机（RISC）体系结构。经典的哈佛体系结构计算机有两个完全分开的存储空间，一个用于指令，一个用于数据。一个具有这两种存储器的处理器能够更高效地工作，因为数据和指令只有在需要时才从计算机的存储器中取出来，而不是在总线可用时就取出来。第一个流行的采用哈佛体系结构的微处理器是AMD的Am29000。该器件在早期的HP激光打印机中相当流行，但后来失宠了，其原因是设计有两个分离存储器的计算机在成本上并非很有效。双存储空间在性能上带来的明显好处是鼓舞CPU设计者以指令和数据cache的形式将存储空间移到芯片上，这在现今的高性能微处理器中是常见的。

如果我们考察过去几年来工作站性能的提高，就会看到CPU能力的增强是相当明显的。有趣的是，仅仅过了4年，一台像Intel 2.4GHz奔腾4这样的高端PC就轻易地超过了一台DEC Alpha 21254/600工作站的性能。

图1-6[5]是工作站性能随时间的变化图。

这些计算机的相对性能是用一组称为*测试基准*（benchmark）的标准程序来度量的。在该例中用的是产业界标准的SPECbase_int92测试基准。用基于这些数字的性能来与更现代的性能度量进行结果比较是困难的，因为测试基准一直在变化。目前，首选的测试基准是SPECint95，它与早期的SPECint92测试基准没有直接关系[6]。然而，根据Mann的估计，用一个大约为38的变
11 换因子就能达到进行粗略比较的目的。因此，根据已发表的结果[7]，一个1.0G Hz的AMD Athlon处理器用SPECint95测试基准得到的结果是42.9，大致相当于用SPECint92得到1630的

结果。DEC公司的AlphaStation 5/300是一款对两组测试基准都公布了结果的工作站，其度量值在图1-6中大概是280，而根据SPECint95测试基准则是7.33，将该数乘以38，我们就得到278.5，与早期的值保持了合理的一致性。在后续章节中，我们还将谈到性能度量问题。

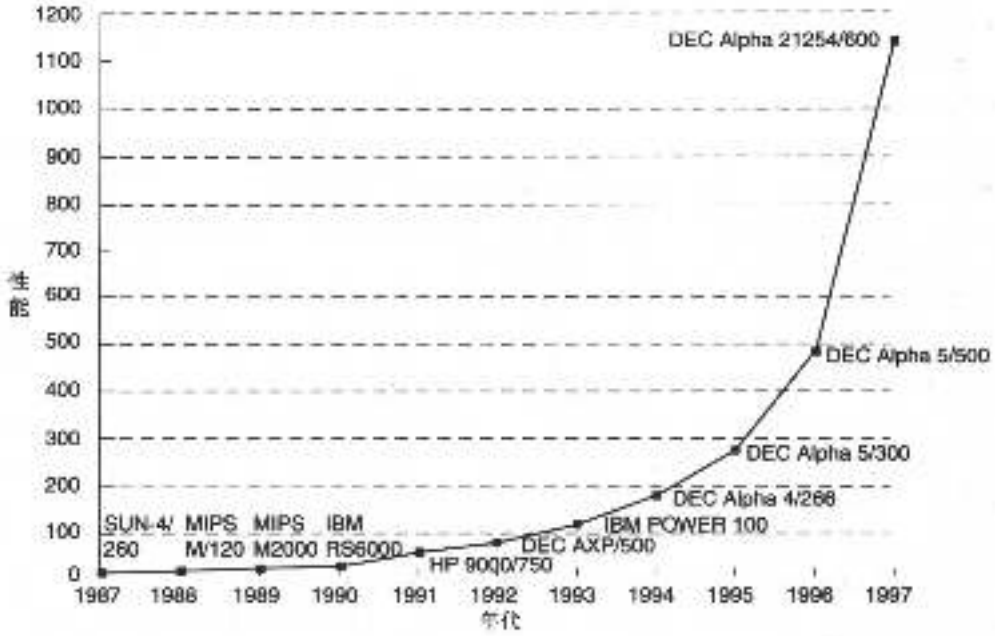


图1-6 工作站性能随时间的改进（由Patterson和Hennessy提供）

1.3 数制

在计算机中如何表示一个数呢？如何在处理器和存储器之间或在微处理器内部传送该数，无论这个数是字符（char）、整数（int）、浮点数（float）还是双精度浮点数（double）？这是一个合情合理的问题，其答案可以使我们明白为什么现代数字计算机是基于二进制的（以2为基数）。为了探讨这个问题，先考虑图1-7。

在图1-7中，我们做一个简单的实验。假设我们能将一个电压置于线上，用来代表我们要在计算机的两个功能元件间传输的数字。这种方法对于传输简单数字是可行的，但若想发送2000.456，我就不想用线来传输了！实际上，这种方法既相当慢又昂贵，并且仅适合于窄范围值的传输。

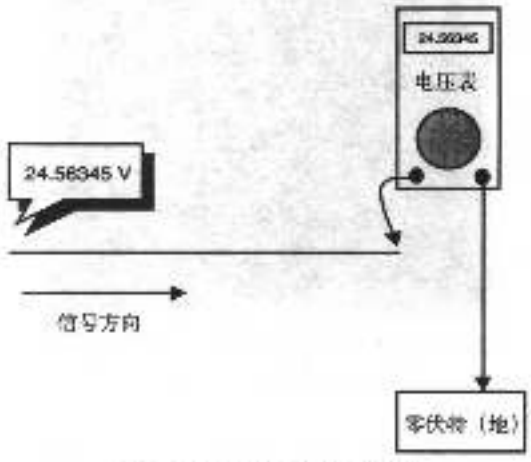


图1-7 用线上电压表示数字值

然而，这不意味着这种方法就根本不能用。事实上，电子计算机的最早家族之一就是模拟计算机（analog computer），模拟计算机是基于线性放大器（linear amplifier）的，就是某种你也许能在家里见到的立体声收音机中的电子电路。关键的一点是：在由电路特性所限定的范围内，变量（这里就指线上电压）可呈现为无穷个值。在很多早期的模拟计算机

中，这个范围可能在-25V~+25V之间。这样，此范围内可表示为稳定或时变电压的任何量均可作为模拟计算机的一个变量。

模拟计算机所利用的事实是，存在能进行如下数学运算的电子电路：

- 加/减
- 对数/反对数
- 乘/除
- 微分/积分

12

通过将这些电路接连地结合，并且在中间采用放大和换算等电路，就能很容易地建立起实时系统模型，当系统运行时，就能求得复杂的线性微分方程的解。

但是，模拟计算机与立体声系统一样具有局限性，即其放大精确性不能达到无限完美，因此，可希望的最好精确度为0.01%，或者说是万分之一。图1-8给出了一个模拟计算机，这是二战期间美国潜艇所使用的型号。鱼雷数据计算机（Torpedo Data Computer, TDC）取罗盘指向、目标船速度、潜艇方向和速度、期望的射击距离作为输入，然后，将算出的正确速度和方向传送给鱼雷，鱼雷就会遵循TDC传输给它们的路线、速度和深度前进。

于是，在20世纪40年代电子电路的局限下，整个计算机体系采用的是基于连续变量的输入和输出，在这种意义上来讲，立体声放大器就是一个模拟计算机，放大器的功能是放大或者提高电信号。一个增益（gain）为10的放大器任意时刻的输出电压都是输入电压的10倍，所以，$V_{out} = 10V_{in}$。这样，我们就有了一个模拟计算模块，它恰好是一个有常数乘数的乘法模块。

让我们还是返回到对数制的讨论。我们也许可以改进这个方法，途径是将数字分割为更易处理的几个部分，并且在同一时间内，在若干条线上（并行）发送范围更有限的信号。这样，每条线就仅需要传输一个窄范围的值。图1-9给出了工作原理。



图1-8 二战期间潜艇上的模拟计算机。照片来自www.fleetsubmarine.com

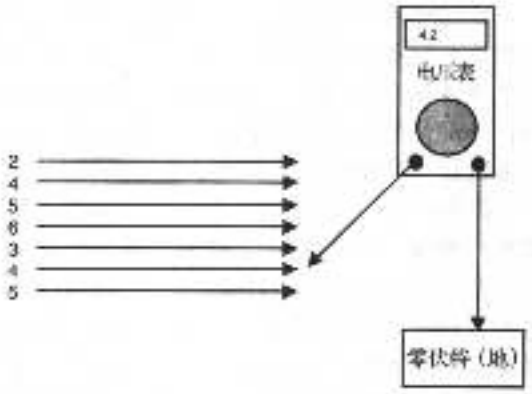


图1-9 在计算机中采用并行线束传输数字值。线在线束中的位置决定了其数字权重，每条线携带的电压值在0V~9V之间

在本例中，该束线中的每一条线代表一个十进制数位，我们发送的每个数字都将用这些线上的相应电压来表示。这样，就不需要传输12 567V这种可能致命的电压，每条线上的电压都不会大于9V电池提供的电压。让我们在这里停留一下，因为这种途径看起来有前途。线上的电压要达到怎样的精确度才能使电路将数字解释为4，而不是3或5呢？在图1-9中，电压表

13

显示，从下向上数的第二条线的测量值为4.2V，不是4V。这样的结果足够好了吗？该值实际上应该达到4.000±0.0005V这样的精度吗？在所有的可能性中，系统在每个电压增加时有大约0.3V偏差的情况下可能会运行良好，因此，为保证电路接收到正确的数字，我们只需要发送4±0.3V（3.7~4.3V）。如果电路出错而发送了4.5V，会怎么样呢？把这个电压看作是4的话太小，看作是5的话又太大，结果就是我们不知道将会发生什么事情，因为由4.5V所代表的值是未定义的。我们希望计算机能正确地工作而没有此类问题。

图1-9提出的方法实际上非常接近现实，但仍不完全是我们所需要的。在现代计算机的速度下，设计既足够快又能足够精确地在线上进行10个不同电压值之间的转换的电路仍然是极其困难的。然而，这个思想将在下一代计算机存储单元中得到考虑。后面还有更多关于这方面的内容，留心哦！

现代的晶体管是极好的开关，它们能在几兆分之一秒（几微微秒）内对电压或电流进行开关变换。能利用这一点吗？让我们想一想。假设扩展线束的概念，进一步对单个线上的值进行限制。那么，由于每条线被一个开关控制，我们将可以在无电压（0V）和某个电压（约3V）之间进行开关。这意味着每条线只可携带两个数字，0或某个值（非0）。这能行吗？让我们看一下图1-10。

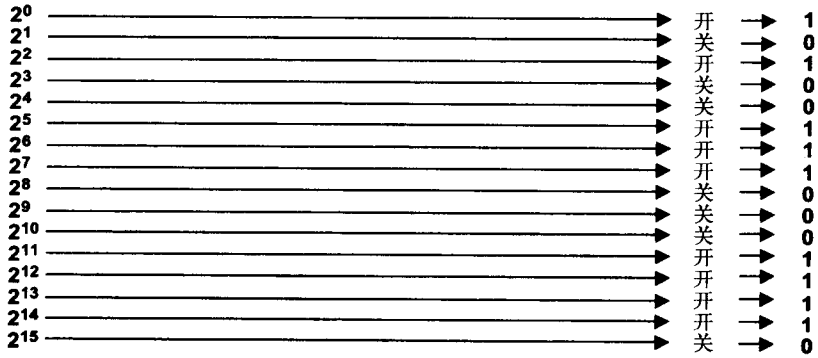


图1-10 以二进制值发送数字。每个箭头线代表一条线，箭头表示信号传输的方向。每条线在线束中的位置代表数字的权重。各行代表2的不断上升的幂

在这个方案中，因为单个线所携带的信息数量被限制为0或者某个值（我们将某个值称为“1”或者“开”），所以我们就需要很多线来传输重要的东西。图1-10显示的是16条线，你很快就会发现：如果处理的是无符号数，那么该数就会被限制在0~65 535之间；如果是有符号数，那么限制范围就在−32 768~32 767之间。这里，十进制数0用二进制数表示为0000000000000000，而十进制数65 535用二进制数表示为1111111111111111。

> 注释：多年以来，大多数标准数字电路都是采用5V来代表1的。然而，随着集成电路变得更小更紧密，逻辑电平也必须降低。目前，现代奔腾或Athlon处理器的电压约为1.7~1.8V，与标准AA电池的差别不是很大。

现在，我们终于将原理弄明白了，我们将利用电子开关元件或晶体管能在线上的两个电压值之间进行快速转换这个事实。最常见的开关形式是在近乎0V和某电压值（大约3V）之间。如果系统工作正常，则我们定义的“零”或0应从不超过1V的大约1/2。这样，我们就可定义数字0为任何小于1/2V（实际上，通常是0.4V）的电压。类似地，我们定义成1的数字应从不小于2.5V。由此，我们就有了定义数制所需的所有信息。这里，数字0从不大于0.4V，数字1从不小于2.5V。任何在这两个范围之间的数都被看成是未定义的和不允许的。 14

应该提到的是我们总是在谈及“线上电压”，那么在计算机中，线究竟在哪里呢？严格地

说，我们应该称这些线为“电导体”。它们可以是实际的线，比如用来将打印机连接到计算机后面并行端口的电缆之中的线；它们也可以是计算机内部印刷电路板中的那些细的导电路径；最后，它们也可以是处理器芯片内部的那些微小的铝导体。图1-11给出了一部分印刷电路板，该印刷电路板取自作者设计的计算机。

请注意，该集成电路（IC）的一些引脚显示出有线将其与另外的器件相连，而另一些引脚则看起来并未与其他器件相连。其原因是印刷电路板实际上是由5个薄层组成的三明治，层的每一面都有印刷线，总共有10层，8个内层之间还有薄的绝缘层以防止电短路。在制造过程中，5个导电层和4个绝缘层是小心地排列和结合在一起的，所形成的10层印刷电路板接近2.5mm厚。

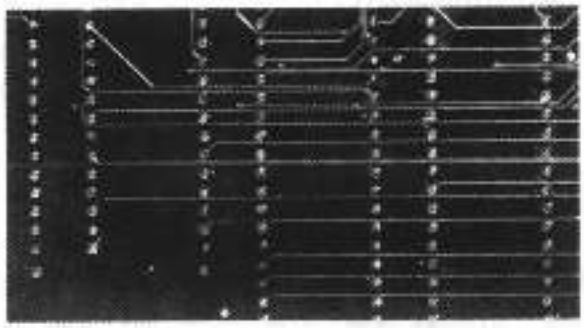


图1-11 计算机电路板上的印刷线。实际上每条线都是铜轨迹线，大约0.08mm宽。轨迹线之间的间距为0.08mm，大的圆点是被焊接的引脚，来自电路板的另一面

没有这种多层制造技术，就不可能建造复杂的计算机系统，其原因是：要在两个组件间连线，而又不穿过具有不同用途的另一条线，这是不可能的。

图1-12［注释：该图的彩色版本包含在随书DVD光盘中。］只给出了内层的情况，这是另一个计算机系统硬件电路的X射线视图，这同你在PC机的母板上所看到的电路有大致相同级别的复杂度。这个视图透过电路板的各层，每层的导电轨迹线都是以不同颜色显示的。

由于这看起来相当复杂，所以大多数版图都是由计算机辅助设计（CAD）软件来制作的。要完成这个电路板的布线，即使是一个熟练的设计者也需要相当多的时间。图1-13是图1-12的一小部分版图的放大，在这里你可以清楚地看到不同层次上的各种轨迹线。每条印刷线大约有0.03mm宽。

15

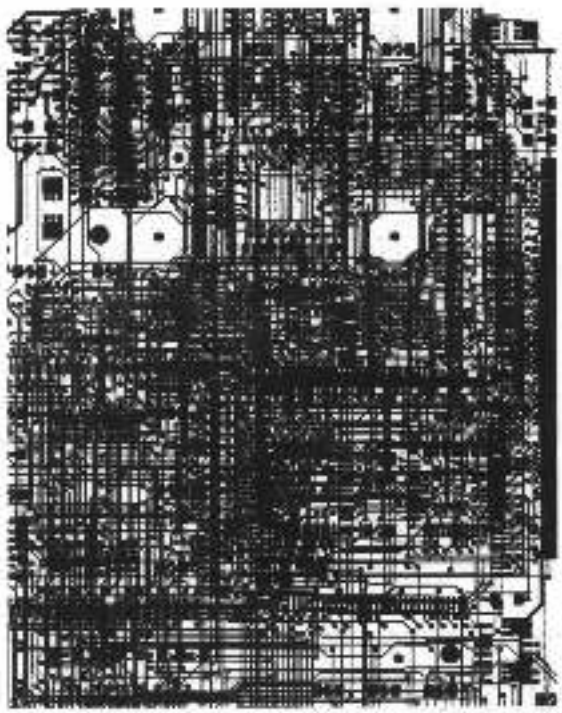


图1-12 某计算机系统的一部分电路板的X射线视图

如果你仔细观察图1-13［注释：该图的彩色版本包含在随书DVD光盘中。］，就会发现某些彩色线接触到了一个黑点，然后似乎是作为一条不同颜色的线转向了另一个方向。这些黑点被称为通孔（via），表示在电路中一条线离开它所在的层面穿到另一层的位置。通孔是垂直导体，它能使信号在层间穿行。如果没有多个层以及层间的通孔，印刷电路板上的线就不能彼此穿过而又不短路。因此，当你看到一条绿线（对图1-13的灰度图像，绿线呈现为点线）穿过一条红线时，要知道这两条线不是在物理上有接触，而是在板的不同层中

穿过。这是一个需要牢记的重要概念，因为我们很快就要看到并绘制我们自己的电子电路图，称为示意图（schematic diagram），而且我们需要牢记如何表示那些看起来互相交叉但在物理上无连接的线，以及那些真正互相连接的线。

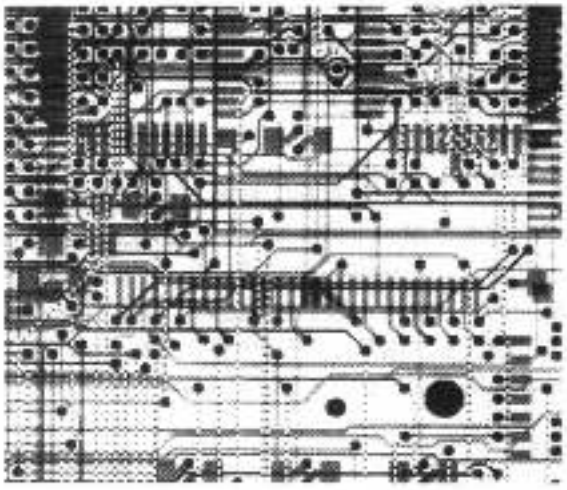


图1-13 图1-12所示电路板的部分放大视图

让我们回顾一下刚才讨论的内容。现代数字计算机采用的是二进制（基为2）数制。采用这种数制的原因是：这种在其数字的自然序列中只有两个数字的数制有助于硬件系统利用开关来指示电路是处于"1"状态（on）还是处于"0"状态（off）。而且，用于建造复杂数字网络的基本电路元件作为逻辑表达式也是基于这些原理。所以，就像我们可能说，在逻辑上一个表达式为TRUE或FALSE，我们可以简单地用"1"（TRUE）或"0"（FALSE）来描述。你很快就会看到，将1与TRUE、0与FALSE关联完全是任意的，我们可以颠倒这种指定而很少或没有负面影响。但是，现在我们还是采用习惯表示，即二进制数1代表TRUE或ON条件，二进制数0代表FALSE或OFF条件。我们将这些总结在下表中： 16

二进制值	电路值	逻辑值
0	OFF	FALSE
1	ON	TRUE

一个简单的二进制实例

由于你可能从来没有接触过电路图，所以我们现在开始学习。图1-14是一个电路的简单示意图，其中包含一个电池、标记为A和B的两个开关和一个灯泡C。电池的正极端标记为加（+）号，负极端标记为减（-）号。想像一下你可能在便携式MP3播放器中使用的典型AA电池，有小凸起的一端是正极端，而有平面部分的另一端是负极端。参见图1-14，正极端用宽线画出，负极端用窄线画出，这看起来可能有些古怪。之所以这么画，是有原因的，但我们在这里就不对此进行讨论了。学电子工程的学生在开始时就被告知了这个原因，但我宜誓保守秘密。

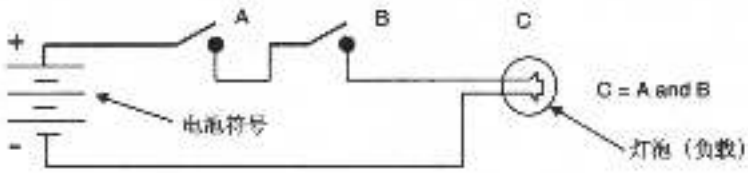


图1-14 采用两个串行开关表示与（AND）函数的简单电路

当有足够电流流过灯泡C使灯丝加热时，灯泡将变亮。我们假设在此电路中，电流从正极流向负极。这样，电流在+端离开电池，经过闭合的开关（A和B），再经过灯，最终到达电池的一端。你可能会觉得奇怪，因为我们从高中的科学课堂上得知，电流实际上是由带负电的电

子组成的，是从电池的负极端流向正极端，而这个的方向却相反。

对这个显然自相矛盾的说法的答案是：历史惯例。只要我们认为电流是带正电的，一切问题就都解决了。

无论如何，为了使电流流经灯丝，必须发生两件事：开关A必须闭合（ON），并且开关B必须闭合（ON）。当这个条件满足时，输出变量C将为ON（变亮）。这样，我们就可以讨论第一个逻辑等式的例子了：

C = A AND B

这是一个非常有趣的结果，我们已经看到采用开关来建造计算机系统的两个显然非常不同的结果。第一个结果是我们必须处理二进制数（基为2），第二个结果是开关也允许我们建
17 立逻辑等式。现在，保留第二项作为有趣的结果，我们将在下一章对其做更深入的讨论。在离开图1-14之前，我们应该指出，开关A和B实际上是机械致动的。某个人轻拨开关即可打开或关闭它。一般情况下，开关是*三端装置*（three-terminal device），有一个控制输入，用于确定其他两端之间的信号传播。

基数

让我们回到对二进制数制的讨论。我们习惯使用十进制（基为10）数制，因为在有iMAC（苹果电脑）之前，我们有10个手指。一个数制的基数（base）或*根*（radix）就是指该数制中不同数字的个数。请看下表：

基数2	0, 1	二进制
基数8	0, 1, 2, 3, 4, 5, 6, 7	八进制
基数10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9	十进制
基数16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F	十六进制

看一下上表的十六进制数，有16个不同的数字（0~9，A~F），表示十进制数的0~15，但这里是在十六进制数制中表示它们。

如果你曾经遭遇过计算机“蓝屏死机”而锁住，你也许能回忆起一些屏幕上出现的样子古怪的字母和数字，其实这些神秘的信息是在试图向你显示，一个十六进制地址值指示的地方刚刚发生了问题。从现在起，蓝屏上的信息不仅仅只是告诉你发生了坏事情并使你丢失了4个小时的工作成果，而且你可以通过领悟这些信息，知道PC存储器的哪些地方发生了非法事件。当然，这对我们的苦恼几乎没有什么安慰。

当用二进制、八进制、十进制或十六进制写一个数字时，我们是在以完全相同的方式表示数字，虽然所采用的基数不同，数字看起来也很不一样。现在让我们考虑一下十进制数65 536，该数恰好是2^{16}。在后面我们会明白这有特殊意义，但现在它只是一个数。图1-15演示了数字65 536的每一位是如何表示成列值乘以列权值的。最左端数字，称为*最高有效位*（most significant digit），是数字6。最右端数字，称为*最低有效位*（least significant digit），
18 恰好也是数字6。因为最高有效位的列权值是10 000（10^4），所以该列的值是$6 \times 10\ 000$，即60 000。如果我们将每列值乘好，然后将它们排列成一个数字列表以便相加，如图1-15右边所示，那么我们将它们加在一起就能得到我们开始时所说的数字。嗯，也许我们在这里夸大了这种明显性，但别急，因为这种情况确实会变得更好。对你来说，这个小例子应该是明显的，因为你已经习惯于使用十进制数字。关键的一点是，列权值恰好是基数值的幂，幂指数的值从最右列的0开始，每向左移一位就加1。由于十进制的基数是10，向左移动时列权值依次就

是：1，10，100，1000，10000，等等。

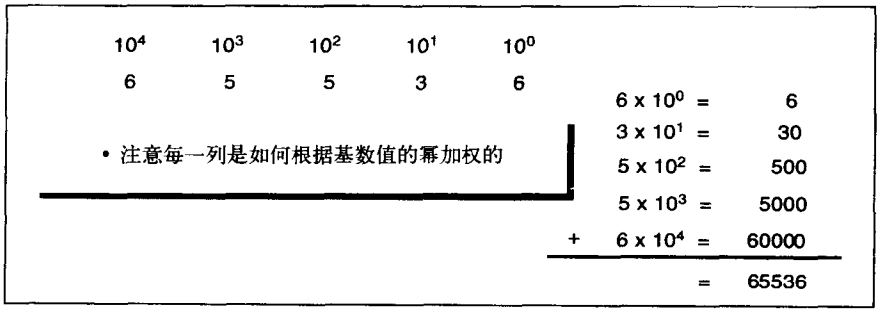


图1-15 以基数10表示数字。从右向左，每个数字乘以基数的幂。该数字正是这些乘积的和
如果能推广这个表示数字的方法，我们就可采用同样的方法，以不同的基数表示数字。

在不同基数之间进行数字转换

让我们重复上面的演示，但这次将采用二进制数字。请看一个8位二进制数10101100，因为该数字有8位二进制数字或位，所以我们称之为8位数。习惯上将8位二进制数称为字节（byte）（在C或C++中代表一个字符）。现在你应该很清楚，为什么二进制数都是由1和0组成的了，除了它们恰好是开关电路（晶体管）的两个状态之外，它们还是二进制数制的仅有的两个数字。

字节也许是最受注意的，因为我们用字节块来度量存储容量。比如，你的PC中的存储器可能至少有256MB（2.56亿字节），而你的硬盘可能也有40GB（400亿字节）或更多的容量。

现在来看图1-16，我们采用的方法与图1-15中十进制例子中的相同。但是，这次列权值是基数2的倍数，而不是基数10的倍数。列权值从最高有效位2^7（即128）到2^0（即1），每一列都比前一列小了二分之一。为了看看这个二进制数的十进制表示是什么样的，我们采用了与前面同样的过程，我们将列中的值与列权值相乘。

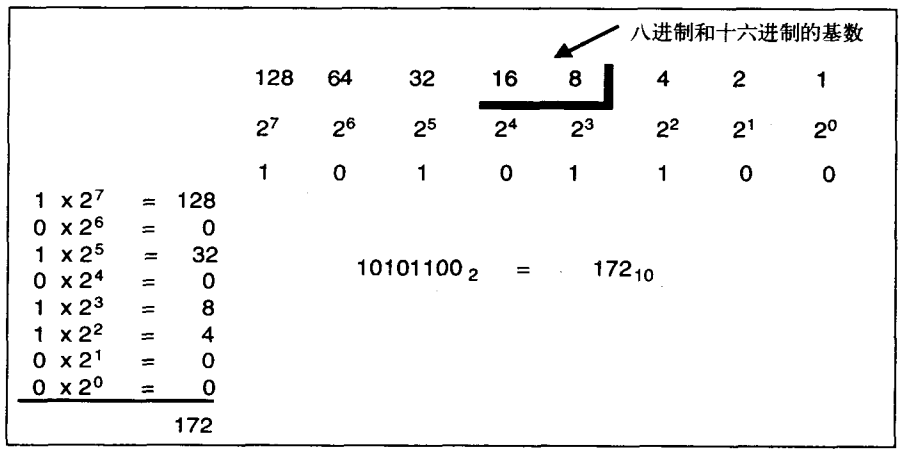


图1-16 将二进制数用2的幂表示。注意八（8）进制和十六（16）进制的基数也是2的幂

于是，我们断定十进制数172等于二进制数10101100。另外，值得注意的是，十六进制数制的基数16和八进制数制的基数8分别等于2^4和2^3。这也许可以给你一个暗示，那就是当我们

和计算机系统打交道时，为什么通常用十六进制表示，偶尔也用八进制表示，但却不用二进制表示。理由很简单，写二进制数字很快就会极其单调乏味，并很容易犯人为错误。

眼见为实，考虑十进制数的二进制形式：

2 098 236 812

这个数用二进制数将写成：

11111010001000010001101100001100

由于二进制数字非0即1，所以通过图1-16所示的过程，二进制数能特别容易地转换成十进制数。这对于我们中那些记不住乘法表的人（PDA的使用已经使我们大脑皮层的重要部分萎缩了）来说，乘法变得容易了。

由于基数2、8和16之间存在着关联，所以，一个合理的假设就是在这三个数制之间进行数字转换要比它们与十进制之间进行数字转换更容易，因为10不是2的自然数幂。图1-17展示了如何将二进制数转换成八进制数。

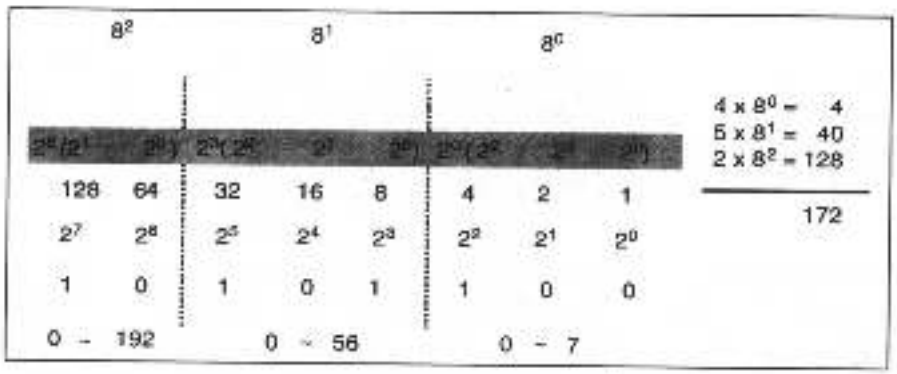


图1-17 将二进制数转换成八进制数。通过提取单数值，我们能将二进制数结合成3个数字的组，并通过观察写出八进制数

图1-17更进一步地利用了图1-16的例子。图1-17从同一个二进制数10101100（即十进制数172）开始。然而通过简单的算术，我们可以提取出2的各种幂，而该幂同时也是8的幂。请看图1-17中的深灰带，我们可做如下简化：

20 由于任何数字的0次幂都等于1，

$$(2^2 2^1 2^0) = 2^0 \times (2^2 2^1 2^0)$$

所以，

$$2^0 = 8^0 = 1$$

我们可以对下一组的3个二进制数字做同样的简化：

$$(2^5 2^4 2^3) = 2^3 \times (2^2 2^1 2^0)$$

因为2^3是该组的公因子。

但是，$2^3 = 8^1$，这是八进制数制中下一列的列权值。

如果对最后两个数字的组再一次重复该过程，那么我们会看到：

$$(2^7 2^6) = 2^6 \times (2^1 2^0)$$

因为2^6是该组的公因子。同样，$2^6 = 8^2$，这正是八进制数制中下一列的列权值。

由于基数8和基数2之间有这种自然的关系，所以在这两个数制之间进行数字转换就变得很容易了。对每一组的3个二进制数字，从最右边（最低有效位）开始，通过简单地将二进制值写成相等的0到7之间的八进制数字，就可以转换成为八进制值。在图1-17中，最右一组3位二进制数字是100，参照列权值，可转换为1×（1×4 + 0×2 + 0×1），即4。中间一组3位二进制数字可转换为8×(1×4 + 0×2 + 1×1)，即8×5(40)。剩下的两个数字转换为：64×（1×2 + 0×1），即128。于是有，4 + 40 + 128 = 172，这也是图1-16中的十进制数。但是八进制数在哪里呢？很简单，每3个一组的二进制数字就给了我们八进制数的列值，即为254。因此，二进制数10101100就等于八进制数254，也等于十进制数172。

很好！这样我们就能按照如下方法进行二进制和八进制之间的转换了：

- 如果数字是八进制的，就将每个八进制数字写成3个二进制数字。例如：
  256773 = 10 101 110 111 111 011
- 如果数字是二进制的，就从最低有效位开始，将这些二进制数字分成3个一组，并写下相等的八进制数字（0-7）。例如：
  $1100010101001101111_2 = 110\ 001\ 010\ 100\ 110\ 111 = 612467_8$。
- 如果最高有效位所在组只留下了一位或两位数字，那么只需用0填补，以使其成为3个数字的组。

目前，虽然八进制不像以前那样经常使用了，但你偶尔还是会遇到。例如，在UNIX（Linux）命令**chmod 777**中，数字777就是各个位的八进制表示，用于定义文件状态。该命令为将要访问文件的用户改变文件的许可状态。

现在我们可以扩展有关二进制数字和八进制数字之间关系的讨论了，从而进一步考虑二进制和十六进制之间的关系。十六进制（hex）数字与二进制之间的转换方法和八进制与二进制之间的转换方法相同，只不过我们现在是用2^4而不是2^3来做公因子。如图1-18所示，可以看到向十六进制转换与向八进制转换所采用的过程相同。 21

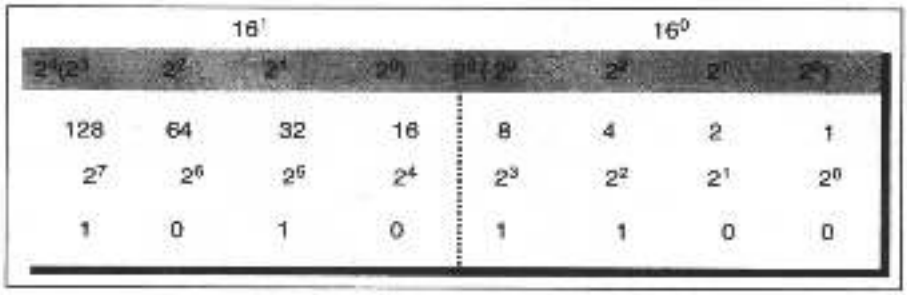


图1-18 将二进制数转换成基为16（十六进制）的数。从最低有效位开始，二进制数分成4个一组，并通过观察写下相等的十六进制数字

在这种情况下，我们提取出基数的公共幂2^4，剩下的就是将二进制数字分组，用列值表示为：

$$2^3 2^2 2^1 2^0$$

很容易看出，二进制数$1111 = 15_{10}$，因此4个二进制数字的组就可用于表示0~15之间的一个十进制数字，或0~F之间的一个十六进制数字。回过头来再看图1-16，既然我们知道了如何做转换，那么与10101100相同的十六进制数是什么呢？最左边的一组4位二进制数字1010正好是8 + 0 + 2 + 0，即A。最右边的一组4位二进制数字1100等于8 + 4 + 0 + 0，即C。因此，该数

的十六进制形式是AC。

让我们做一个16位二进制数转换的例子。

二进制数	0 101 111 111 010 111
八进制数	0 101 111 111 010 111 = 057727（3个数字一组）
十六进制数	0101 1111 1101 0111 = 5FD7（4个数字一组）
十进制数	要转换到十进制，见下表。

八进制转换到十进制	十六进制转换到十进制
$7 \times 8^0 = 7$	$7 \times 16^0 = 7$
$2 \times 8^1 = 16$	$13 \times 16^1 = 208$
$7 \times 8^2 = 448$	$15 \times 16^2 = 3840$
$7 \times 8^3 = 3584$	$5 \times 16^3 = 20480$
$5 \times 8^4 = 20480$	
24,535	24,535

定义

在考虑将十进制转换到十六进制、八进制、二进制这种反向转换过程之前，我们应定义一些术语。这些术语为计算机数字所专用，且为我们在以后表示数字大小给出了一种速记方式。这里所说的“大小”，并不是指数字的大小，而是指数字的二进制位数。你已经熟悉这个概念了，因为大多数编译器都要求你在使用变量前先声明其类型。声明类型实际上意味着要 22 做两件事：

1. 该变量要占用多少存储空间？
2. 必须生成什么类型的汇编语言算法来操作这个数？

下表总结了二进制位的各种分组及其定义。

位（bit）	最简单的二进制数字是1位长
半字节（nibble）	包含4个二进制位的数字。一个半字节也是一个十六进制数字位
字节（byte）	8个二进制位一起构成一个字节。字节是度量计算机存储器和磁盘存储容量的基本单位。字节在C和C++中也等价于一个字符
字（word）	一个字的长度是16个二进制位。这也是4个十六进制数字或2个字节的长度。当我们谈论存储器组成时，这个概念将会变得更重要。在C或C++中，字有时也称为短整数（short）
长字（long word）	也称为长数（LONG），长字是32个二进制位或8个十六进制数字。目前，这在C和C++中就是一个整数（int）
双字（double word）	也称为双数（DOUBLE），双数是64个二进制位，或者16个十六进制数字

从该表中你可能得到这样一个线索，就是为什么八进制表示已大多被十六进制表示所取代。这是因为八进制是由3位二进制数组成的，所以我们通常要处理令人烦恼的余下位数，因此，我们似乎总是会看到在最高的八进制有效位上有额外的1个、2个或3个二进制位。如果计算机的设计者决定用15位和33位而不是用16位和32位作为总线宽，那么也许八进制还会存在并活跃着。另外，十六进制是更为紧凑的数字表示方式，因此它成为目前的标准。图1-19总结了各种数据元素目前的长度和不久的将来有可能会用到的长度。

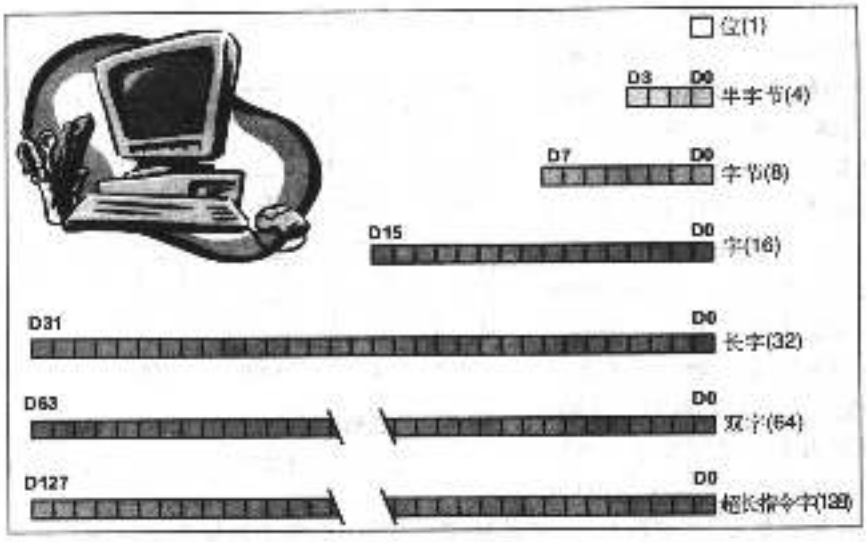


图1-19 计算机系统中各种数据元素的长度

目前，我们已经有了能在一个操作中对64位数进行操纵的计算机，Advanced Micro Devices公司的Athlon6400就是一种这样的处理器。另一个例子是任天堂的N64 Game Cube®中的处理器。还有，如果你认为自己是一个PC游戏者，你喜欢在PC上玩快动作的视频游戏，那么在你的游戏机上就可能具有一个高性能的视频卡，很可能你的视频卡上具有一个视频处理计算机芯片，能够一次处理128位。256位的处理器还会远吗？ 23

小数

我们处理小数的方式与处理整数的方式相同。例如，考虑图1-20。

10^2	10^1	10^0	10^{-1}	10^{-2}	10^{-3}	10^{-4}
5	6	7	4	3	2	1

图1-20 表示基数为10的小数

我们看到对于一个十进制数，小数点右边的列以10的负整数幂增长，所以我们可将刚学到的基数间的转换方法应用于小数的转换。但既然说到这里，就应提及在计算机中小数通常并不是这样表示的。任何小数都将立刻被转换成浮点数，即*float*。浮点数有其自己的表示方法，典型的表示是包含尾数和指数的64位值。我们将在后面的章节中讨论浮点数。

二进制编码的十进制

出于完整性考虑，还有最后一种形式的二进制表示我们应该顺便提及一下。在计算机发展的早期，当需要对来自数字逻辑仪器的数据（而不是现今基于计算机的数据）进行转换时，将数字表示成二进制编码的十进制（binary coded decimal，BCD）更为方便。一个BCD数可以表示成4个二进制数字，就像一个十六进制数，不同之处在于最大数字是9而不是F。计算器和仪表等装置采用BCD的原因是它是一种将数字值与某种显示装置（如7-段显示器）相关联的方便途径。图1-21给出了7-段显示器所显示的数字。

7-段显示器包含7个棒条，通常还有一个小数点，每个元素都是通过发光二极管（LED）发亮。图1-21示意出7-段显示器如何用于显示0到9的数字。事实上，稍加创新，也能显示A到F的十六进制数字。

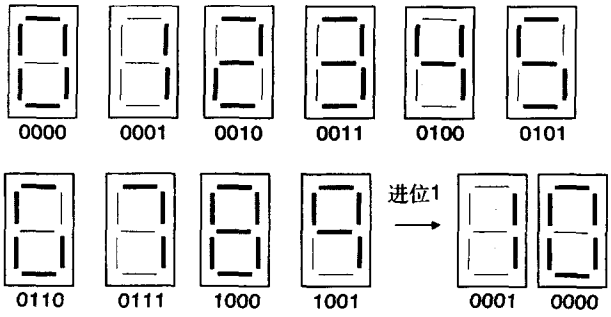


图1-21 二进制编码的十进制（BCD）数表示。如果数字超过计数9，就发生进位操作，二进制数字返回到0

BCD是将数字计数器和电压表数据转换成显示器上易读数字的简便方式。想像一下，如果Radio Shack®电压表的读数是7AV电压，而不是122V电压，那么你会有什么反应。图1-21表明了当我们用BCD计数时会发生什么。显示的数字对于我们是有意义的，因为它们看起来像十进制数字。当计数达到1001（9）时，下一次的加1就会使显示器返回到显示0并进位1，而不是显示A。现今的很多微处理器还带有从BCD到十六进制数转换时期的痕迹，这可以从它所包含的特殊指令看出来，例如，*十进制加法调整*（decimal add adjust）指令，用于为实现BCD算术运算产生算法。

24

1.4 将十进制数转换为各种基数的数

将十进制数转换为二进制、八进制或十六进制稍有些棘手，因为基数10与任何其他基数都没有自然的关系。然而，转换算法是一个十分简单的过程。例如，让我们将十进制数38 070转换成十六进制数。

1. 找到基数中最大的（本例是16），将其对某个尽量大的整数求幂，使得幂值小于你要做转换的数。为了做此转换，我们需要参考如下所示的16的幂值表。从表中我们看到38 070大于4 096但小于65 536。因此我们知道转换的最大列值是16^3。

$16^0 = 1$	$16^1 = 16$	$16^2 = 256$	$16^3 = 4096$
$16^4 = 65\,536$	$16^5 = 1\,048\,576$	$16^6 = 16\,777\,216$	$16^7 = 268\,435\,456$

2. 对数字执行整数除法操作：

a. 38 070 DIV 4096 = 9

b. 38 070 MOD 4096 = 1206

3. 最高有效十六进制位就是9。用步骤2所得到的MOD（余数）重复步骤1。256小于1206而4096大于1206。

a. 1206 DIV 256 = 4

b. 1206 MOD 256 = 182

4. 这样次一位的最高有效位就是4。用步骤3所得到的MOD重复步骤2。182大于16而小于256。

a. 182 DIV 16 = 11（B）

b. 182 MOD 16 = 6

5. 第三个最高有效位就是B。到此我们就可停止了，因为最低有效位可通过观察得出，为6。

6. 因此有：$38\,070_{10} = 94B6_{16}$。

在我们转入下一个话题（即逻辑门）之前，有必要总结一下我们讲解这些内容的原因。当

要用32位二进制数写下一个数字时，如果你能意识到有更好的方式，那么就不用写很长的数字了。更好的方式就是十六进制和八进制。因为十六进制数和八进制数的基数分别为16和8，所以它们与二进制的基数2有着自然的关系。我们可以通过将二进制数字结合成3个一组或4个一组来简化数字操作，这样，我们就能将32位二进制数如10101010111101011110000010110110写成十六进制数AAF5E0B6。

但是，请记住我们仍然是在处理二进制值，只是用来表示这些数字的方式有所不同而已。稍后你将会看到，这种二进制和十六进制之间的自然关系也可延伸到算术操作。为证明这一点，可进行如下的十六进制加法：

0B+1A = 25（记住，25是十六进制的，不是十进制的。十六进制的25就是十进制的37。）

现在，再将0B和1A转换到二进制并做同样的加法。注意在二进制中，1+1=0，并产生进位1。 25

由于用十六进制、二进制和八进制很容易对数字产生误解，所以汇编器和编译器允许我们容易地规定数字的基数。在C和C++中，我们将十六进制数如AA55表示成0xAA55。在汇编语言中，我们采用的是美元符号，这样该数在汇编语言中就是$AA55。然而，由于汇编语言没有一个像ANSI C那样的标准，所以不同的汇编器可能会使用不同的符号。另一个常用的方法是，如果最高有效位是A到F的数字，那么可以在十六进制数之前加一个0，最后附加一个“H”。这样，$AA55在一个不同供应商的编译器中就可能被表示为0AA55H。

1.5 工程符号

虽然大多数学生都学习过用科学符号来表示很大或很小的数字的基本方法，但不是所有的人都学习过如何对科学符号做某些推广，以简化我们在数字系统中要处理的一些常用数量的表达。因此，让我们暂时离开主题来讨论一下这个课题，以便使我们有一个共同的起点。你若已经知道了这些知识，那么你可以提早大约10分钟结束本章的学习了。

工程符号仅仅是对极大数字或极小数字的一种速记表示，是一种有助于工程人员进行简便交流的格式。让我们来看一个例子，这个例子来自于许多年前我从电视上看到的一个关于蝙蝠的自然类节目。蝙蝠在绝对黑暗中利用它们所发出的超声波回声来定位昆虫，昆虫反射声波脉冲，蝙蝠就能定位食物。我尤其记得讲述者说，蝙蝠的神经系统是如此地精于回声定位，它能分辨出小于百万分之几秒间隔内到来的声波脉冲。哇！

但是，“百万分之几”意味着什么呢？若我们说百万分之几秒是百万分之五秒，那就是0.000005秒，科学符号中0.000005秒可写成5×10^{-6}秒。工程上的记法是5μs，读成5微秒，我们采用希文符号μ（mu）来表示微秒的“微”这部分意思。我们常常遇到的符号有哪些呢？下表列出了一些常用的值：

太（tera）= 10^{12}（T）	皮（pico）= 10^{-12}（p）
吉（giga）= 10^{9}（G）	纳（nano）= 10^{-9}（n）
兆（mega）= 10^{6}（M）	微（micro）= 10^{-6}（μ）
千（kilo）= 10^{3}（K）	毫（milli）= 10^{-3}（m）
	飞（femto）= 10^{-15}（f）

既然这样，那么我们如何将科学符号转换成等价的工程符号呢？下面就是转换方法：

1. 调整尾数和指数，使得指数可被3整除，而尾数不是小数。这样，3.05×10^{4}字节就要成了30.5×10^{3}而不是0.03×10^{6}。

2. 将指数项用适当的名称替代。这样，30.5×10^3字节就是30.5K字节，或者30.5千字节。

在99.99%的时候，指数范围在±12之内。然而，随着计算机越来越快，我们将会在皮秒
26 以内来度量时间，所以将飞秒包括进表中也是适当的。作为一个练习，请计算一下在1 飞秒内光能传播多远，假设光在印刷电路板上的传播速度大约是每纳秒15cm。

虽然在较早时我们讨论了这个问题，但在这里还是应该再强调一下，当使用kilo、mega、giga这些工程术语时，我们必须要谨慎。这个问题是计算机界的人不适当地将标准工程符号为己所用而造成的。由于$2^{10} = 1024$，所以计算机界的“滑稽表演者”就认为它与1000非常接近，由此，符号K、M、G就分别承载了1024、1048576、1073741824的意思，而不是1000、1000000、1000000000的意思了。

幸运的是，我们很少混淆，因为计算机的定义通常局限于对有关存储器大小或字节容量的度量。当度量任何其他事物的时候（如时钟速度或时间），我们就采用这些单位的传统意义。

总结

- 集成电路微处理器制造技术的进步促使现代数字计算机迅速发展。
- 计算机存储器的速度与其容量具有反比关系。存储器速度越快，它就越接近于计算机的核心。
- 现代计算机基于两个基本设计：CISC和RISC。
- 由于电子电路可在“开”与“关”之间快速转换，所以计算机可以采用二进制数制（即基数为2），而不是采用其他自然数的数制。
- 二进制、八进制和十六进制是计算机的自然的基数，在它们之间以及它们与十进制数之间有简单的转换方法。

参考文献

[1] Carver Mead, Lynn Conway, *Introduction to VLSI Sysyems*, ISBN 0-2010-4358-0, Addison-Wesley, Reading, MA, 1980.

[2] 关于创建新的超级微型机的优秀著作，可参见 *The Soul of a New Machine*, by Tracy Kidder, ISBN 0-3164-9170-5, Little, Brown and Company, 1981.

[3] Daniel Mann, Private Communication.

[4] http://www.seagate.com/cda/products/discsales/enterprise/family/0,1086,530,00.html.

[5] David A. Patterson and John L. Hennessy, *Computer Organization and Design, Second Edition*, ISBN 1-5586-0428-6, Morgan Kaufmann, San Francisco, CA, p. 30.

[6] Daniel Mann, Private Communication.

27 [7] http://www.specbench.org/cgi-bin/osgresults?conf=cint95.

习题

1. 定义*摩尔定律*。摩尔定律在理解计算机性能的发展趋势方面有哪些寓意？限制你的答案不要超过两段。
2. 假设在2004年1月，AMD发布了一款新的具有1亿个晶体管的微处理器，那么根据摩尔定律，AMD将在什么时候发布具有2亿个晶体管的微处理器。
3. 对于采用抽象层次的计算机组织，描述其一个优点和一个缺点。
4. 什么是典型PC机的工业标准总线？
5. 假设平均存储器访问时间是35ns（纳秒），平均硬盘访问时间是12ms（毫秒），那么，半导

体存储器比硬盘存储器快多少？

6. 十进制数357用基数为9表示是什么？
7. 将下面的十六进制数转换成十进制数：
   (a) 0xFE57
   (b) 0xA3011
   (c) 0xDE01
   (d) 0x3AB2
8. 将下面的十进制数转换成二进制数：
   (a) 510
   (b) 64 200
   (c) 4 001
   (d) 255
9. 假设你正以两星期14弗隆（furlong，长度单位，1弗隆=660英尺）的速度旅行，那么每秒你旅行多少英尺？用工程符号表示你的答案。 28

第 2 章　数字逻辑简介

学习目标

- 了解数字逻辑门的电路基础；
- 理解现代CMOS逻辑电路的工作原理；
- 熟悉各种逻辑门电路。

2.1 引言

还记得图1-14中简单的电池和闪光灯电路吗？其中用线串连起来的两个开关实现了逻辑“与”（AND）的功能。这个例子表达的意思是，“如果开关A闭合并且开关B闭合，那么灯泡C就会亮。”毫无疑问，这个电路远比我们用的电脑简单得多，但其中两个开关实现的逻辑功能却是构成现代计算机的4个关键元件之一。

说出来可能让人惊讶，现代计算机所有的主要数字元件，如中央处理单元（CPU）、存储器和I/O设备都可以由4个基本逻辑功能构造出来：与（AND）、或（OR）、非（NOT）和三态（Tri-State, TS）。其实“三态”并不是一个逻辑功能，它更接近于一种电子电路实现工具。但若没有三态逻辑，就不可能建造出现代计算机。

我们即将看到，三态逻辑引入了一个称为高阻（Hi-Z）的第三种逻辑条件，这里“Z”是阻抗的电路符号，描述的是电流流入电路的难易程度，因此，高阻似乎意味着对电流有很大的阻碍作用。后面将说明，高阻对建造整个系统非常关键。

前面说过，我们可以使用4个基本逻辑门：与、或、非和三态，从下至上地建造出一台计算机。但这并不意味着我们就将直接使用它们。其原因在于，工程师们通常使用一些现成的模块来设计更复杂的电路，而这种有效率的设计方法将使得基本电路元件间的区别变得模糊。但是，这并不能抹煞这4种基本逻辑功能在概念上的重要性。

写到这里，我突然想到了DNA分子和计算机的相似性，并为它所打动。DNA分子有4种核苷酸：腺嘌呤、胞嘧啶、鸟嘌呤和胸腺嘧啶（分别简写为A、C、G和T），而计算机也可以由4种“电子核苷酸”来描述。我们也不要太惊讶这种巧合，因为它们之间的区别远远大于这种相似性。无论如何，想像将来有一种“电子DNA分子”是多么有趣，它可用来作为复制自己的蓝本。会有一本关于这个想法的科幻小说吗？图2-1解释了DNA和计算机硬件基本元件之间的类似关系。

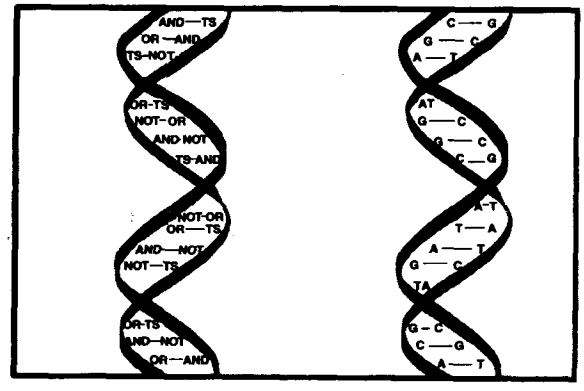


图2-1 假想一个计算机由左边的“逻辑DNA”建造，而右边是一个真实DNA分子结构图的一部分（DNA分子图片来自冷泉港实验室的Dolan学习中心[1]）

关于与DNA的相似性就说这么多！我

们继续新的内容。在谈新内容之前，我要做最后一点说明。造成这种相似性的关键在于，从现在开始我们在数字硬件领域要做的所有事情都将基于这4种基本的逻辑功能。这样做可能不会很容易，但却是我们面临的事情。

图2-2是一个可执行逻辑“与”功能的数字逻辑门的示意图。图中标为F(A, B)的符号是数字电路设计中表示与门的标准记号。输出C是两个二进制输入变量A和B的函数。A和B都是二进制变量，但在这个电路中它们是由电压值来表示的。*正逻辑*（positive logic）是指正（较高）电压表示“1”，而非正（较低）的电压表示“0”。在正逻辑下，与门常被用于这样的电路，其中逻辑“0”用0~0.8V范围的电压表示，“1”则由大约在3.0V和5.0V[⊖]之间的电压表示。0.8~3.0V之间是“无人区”，信号值上下变化时经过这个区间的速度非常快，不会停留下来。如果测量出一个逻辑值的电压在这个区域内，那么这个电路里一定有一个电学错误。

图2-2所示的逻辑函数可以有几种等价的表达方式：

- *如果A为真且B为真，那么C为真*
- 如果A为高电压且B为高电压，那么C也是高电压
- 如果A为1且B为1，那么C也是1
- 如果A开启且B开启，那么C也为开启状态
- 如果A处电压为5V且B处电压为5V，那么C处电压也为5V

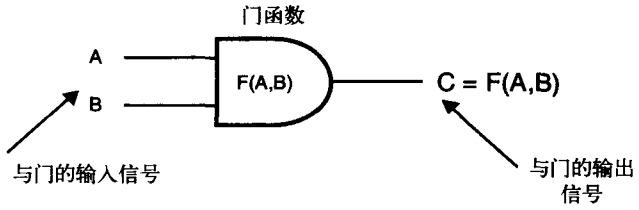


图2-2 逻辑与门。与门是对两个输入值A和B实现逻辑与功能的电子开关电路

最后一项有点问题，因为我们是允许电压值在一定范围内变化，而不是一个绝对的数值。但在这里，说“5V”比说“3V~5V的范围”要简洁得多。

如我们在上一章所讨论的，A和B在实际电路中都是单个线上的信号。这些线可以是用于形成电路的实际导线，也可以是图1-11所示的印刷电路板（PC）上的特别细的铜线（轨），甚至还可以是一个集成电路芯片中的用显微镜才能看到的互连线。无论哪种情况，它都是一根传递数字信号的线，而这个信号的值只有两种可能：“0”或者“1”。

我们要考虑的下一个问题是，为什么我们称这个电路元件为“门”。你可以想像一下现实生活中你家门口的那个门，任何人要进你家都必须打开这个门。与门也可以按同样的方式来理解，它是逻辑信号通路上的一个门。虽然前面的列表把与门描述成了一个个逻辑语句，然而我们也可以这么来描述它：

30

- *如果A为1，那么C等于B*
- *如果A为0，那么C等于0*

当然，我们可以在上面的表述中交换A和B，因为这两个输入是对等的。上述含义也可以用图2-3来表示。注意输入B和输出C处那些奇怪的折线，这是表达随时间变化的数字信号的一种方式，称为*波形*（waveform）表示，或称*时序图*（timing diagram）。可以将它理解成画在某种绘图纸（比如带状记录纸）上的波形，其中水平轴代表时间的改变，纵轴代表电路中一点的逻辑值，这个图中表示的是点B和点C的。

⊖ 假设我们使用的是较老的5V逻辑系列，而不是较新的3.3V逻辑系列。

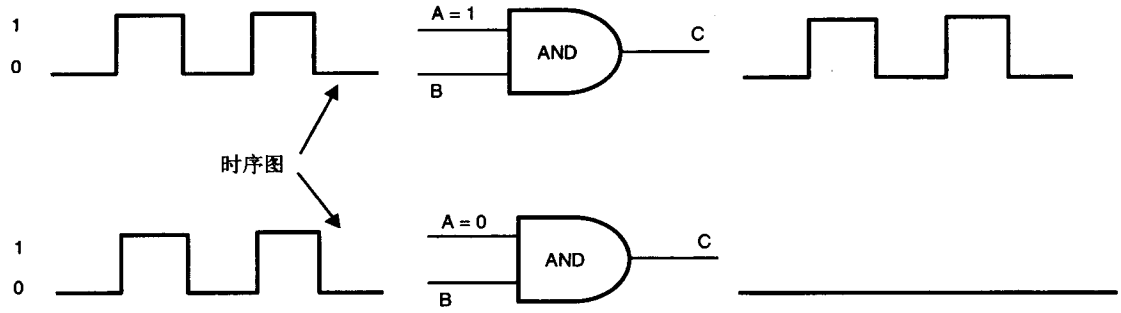


图2-3 用门实现的逻辑与电路。当输入A = 1时，输出C和输入B一样（门为开启状态）；当输入A = 0时，输出C = 0，而与输入B如何变化无关（门为关闭状态）

当A=1时，与B处输入相同的信号出现在输出C上。B处信号的改变并不会立即导致C的变化，因为光速是有限的，电路速度也不是无限快的。然而，以人类日常生活的尺度来看，这已经是相当快了。在一个典型的电路上，如果输入B从0值变到1，则经过大约一秒的十亿分之五（5ns）的延迟，C处即会发生相同的变化。

你实际上以前就见过这种时序图，是在医生检查你的心脏时，以心电图（EKG，electrocardiogram）的形式出现的。图上的每个信号代表着你心肌各个部位的电压随时间的变化。时间由图上较长的坐标轴表示，而每个时间点上的电压则表示为墨水的纵向位移。由于典型的数字信号的变化要比我们从一个带状记录纸上看到的快得多，所以我们就需要特殊的设备（比如示波器和逻辑分析仪）来记录这些波形并以我们能理解的方式加以显示。在接下来的章节中我们还会讨论波形和时序图。

这样，图2-3显示出B处的一个输入波形。如果我们人可以变得非常小，而且有一个非常
31 快的秒表和一个快速的伏特计，可以想像，我们坐在连接到这个门的B点的线上。当B处电压值发生变化时，我们查看秒表并在图上画出这个时间对应的电压值，那么就能得到图2-3所示的波形。

还要注意那条表示从逻辑电平0到逻辑电平1转换的竖线，它称为*上升沿*（rising edge）。同样地，表示逻辑电平1到逻辑电平0转换的竖线称为*下降沿*（falling edge）。我们通常希望上升沿和下降沿所对应的时间间隔都很小，应该是几个一秒的十亿分之一（ns）。这是因为在数字系统中，0和1之间的值是未定义的，因此我们希望信号能尽可能快地通过这个中间地带。但这也不是说这些边就对我们毫无用处，恰恰相反，它们非常重要。事实上，当我们在后面学习系统时钟时将会看到这些边的价值。

在图2-3中，如果A=1（图的上半部分），那么C处的输出就会在一个小的时间延迟后和B一样（译者注：此处原文为“和A一样”，属笔误）。这个延迟时间称为*传输延迟*（propagation delay），代表输入信号上的变化传递到元件的输出所经过的时间。当*控制输入*（control input）A=0时，输出C将总是为0，而不管输入B如何变化。因此，输入A对输入B起到了门控作用。现在，我们将离开硬件设计的精彩世界中的时序图，回到对逻辑门的学习中。

前面，我们提到过与门是三种逻辑门中的一种。我们现在先把三态门排除出去，在后面章节讨论总线组织时再对它进行深入学习。以“原子”元件或独特元件来论，实际上有三种逻辑门：与门、或门和非门，这些是构造所有复杂数字逻辑电路的基石，见图2-4。

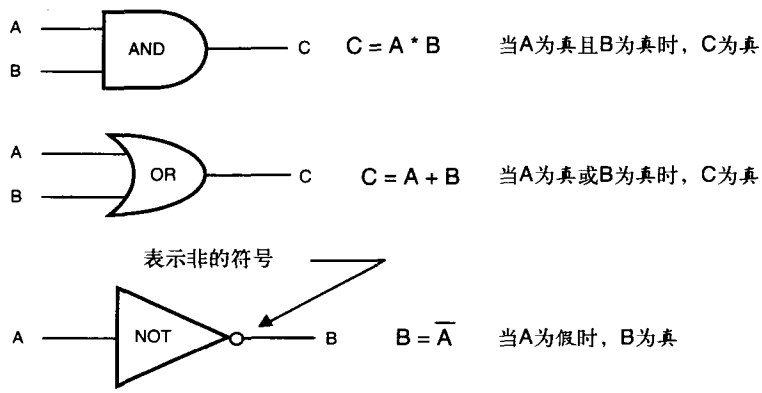


图2-4 三个“原子”逻辑门：与门、或门和非门

表示与函数的符号和我们在代数学中用的乘法符号一样。我们后面将看到，与操作类似于将两个二进制数相乘，因为1×1 = 0并且1×0 = 0。这个星号是用于将两个变量做“与”操作的方便符号。

表示或函数的符号是加号，它在某种程度上类似于加法，因为0 + 0 = 0，1 + 0 = 1。对于1+1则没有这种类似，因为如果是计算加法则1 + 1 = 0（进位1），但若是或操作则1 + 1 = 1。好吧，可以把表示或函数的符号看成是对加号的重载。别对我的解释生气，我仅仅是传递信息的人。 32

表示非的符号有许多种形式，主要是由于仅使用ASCII文字字符无法在一个变量上面画出那条横线。在图2-4中，我们用变量A上的横杠说明输出B是输入A的非、或者相反值。如果A = 1，那么B = 0；如果A = 0，那么B = 1。如果仅使用ASCII文本字符的话，那么你可能会看到非的符号是用波浪号或者前向斜线表示的，写成B = ~A，或B =/A。非也称为补(complement)。

非门的图形表示中，也用输出上的一个小圆圈表示“非”的含义。一个单输入、单输出的门若没有“非”的符号（即小圆圈）则称为缓冲器（buffer）。缓冲器的输出波形总是和输入波形一致，只是减去一个小的传输延迟。从逻辑上讲，没有明显的必要引入缓冲器门，但从电学上看（又是那些麻烦的硬件工程师！）缓冲器是一种重要的电路元件。

在后面学习模拟到数字转换的课程时，我们还会回到缓冲器的概念上来。不同于与门和或门，非门总是只有一个输入和一个输出。而且，由于它太简单，也没法更简单地用闪光灯电路显示非门的效果，因此我们将把重点放在或门上。

就像第一次介绍与门一样，我们也可以用同样的方式来看或门。我们将使用来自图1-14的简单闪光灯电路，但这次将重新布置开关的位置从而产生逻辑或的功能。图2-5是这个闪光灯的电路。

图2-5所示的电路中，两个开关A和B并联，闭合其中任何一个都将允许电流从电池流经开关到灯泡再流回来。将两个开关都闭合也不改变灯泡点亮的现状，灯泡也不会因此而更亮。让灯泡不亮的唯一办法是将两个开关都断开从而切断流向灯泡的电流。最后，图2-5中的小黑点应引起注意，它表明两个相交的导线确实有电连接。正如我们前面讨论的，由于我们的示意图只是二维的，而且印刷电路板经常有10层互相绝缘的导电层，所以当看到两个线交叉但物理上又没有连接时就会造成混乱。通常，当两根电线互相交叉时，我们会看到很多放电火花，房间也变暗。在示意图里，我们用一个小黑点表示两个线实际连接在一起，任何其他的

交叉线都认为是相互绝缘的。

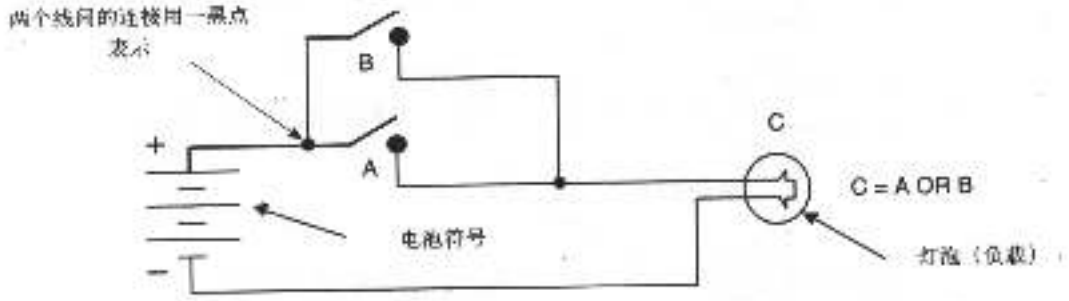
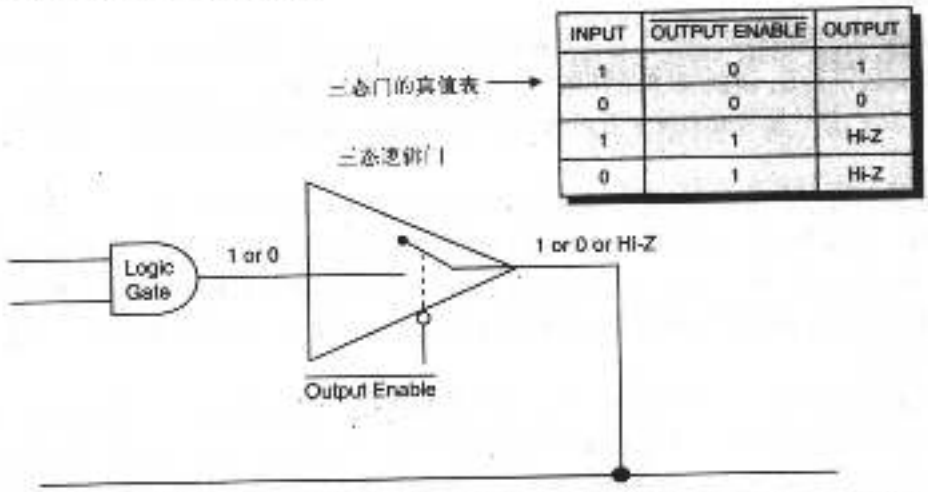


图2-5 用两个并联开关实现的逻辑或功能。闭合开关A，B中任何一个都可以点亮灯泡C

33

到现在为止我们还没有介绍三态逻辑门。虽然对你来说这种元件存在的原因还不明显，但我们可能还是应该做点介绍，因此，冒着泄露机密的危险，让我们看看图2-6中所示的第4种“原子”逻辑元件：三态逻辑门。



INPUT	$\overline{\text{OUTPUT ENABLE}}$	OUTPUT
1	0	1
0	0	0
1	1	HI-Z
0	1	HI-Z

图2-6 三态（TS）逻辑门。当输出使能端（$\overline{\text{OE}}$）为低电压时门的输出跟随输入变化。当$\overline{\text{OE}}$为高电压时门的输出端为高阻状态

关于三态门有几个重要的概念，因此我们要花一些时间来解释这幅图。首先，左边表示门的电路图几乎和非门一样，只是在输出端没有表示“非”的小圆圈，说明输出在某个传输延迟之后跟随输入的逻辑状态，这使得这个门也称为缓冲器门（buffer gate）。然而，这些并不说明我们不能通过用一个非门来得到三态门，但那样三态门就不是“原子”器件了，那将是一个复合门（compound gate）。

三态门还有第二个输入，在图中用$\overline{\text{Output Enable}}$（输出使能）标记。这个输入端上有一个表示“非”的小圆圈，这说明三态门是“低电平有效”（active low）的，这里我们引入了一个新的术语。那么什么叫“低电平有效”呢？前面一章中，我们快速地提到过逻辑电平有任意性。为了方便，我们常令较高的电压为1（或真），而令较低的电压为0（或假）。这就是通常使用的正逻辑。使用正逻辑其实没有什么特别的道理，只是一种广泛使用的习惯而已。

然而也有某些时候，我们希望让逻辑状态“真”代表一个低电压（或0）。其实这又涉及一个更基本的问题，即便我们在处理一个逻辑状态，这个信号的意义也并不十分清晰。可以

举出很多种情况，其中信号被当作一个起控制或使能作用的设备。在这些情况下，真和假的含意就和它们在一个复杂的逻辑方程中的不那么一样了。这就是我们面对三态门缓冲器时遇到的情况。

当 $\overline{\text{Output Enable}}$（简写为 $\overline{\text{OE}}$）上的信号为低电压时，它对三态门缓冲器是有效的。在 34
这个例子里，当 $\overline{\text{OE}}$ 输入为低电压时，缓冲器有效，输出状态跟随输入信号变化。现在，你可能马上会说："别骗人了，那其实是一个与门，不是吗？"确实，你差一点就说对了。在图2-3中，我们通过与门介绍了与逻辑功能。当A输入为0时，门的输出也是0，而与B输入的逻辑状态无关。三态门则与这个有实质的区别。当 $\overline{\text{OE}}$ 端为高电压时，从电路角度来看这个门，门的输出就不存在了，就好像门不在那儿，这就是高阻抗状态。因此，图2-6中所表现的情形是不同于其他的，当 $\overline{\text{OE}}$ 为低电压时三态门就像一个闭合的开关，而当 $\overline{\text{OE}}$ 为高电压时它就像一个断开的开关。换句话说，高阻既不是逻辑状态1、也不是逻辑状态0，而是一种独一无二的状态，它和数字逻辑关系不大，但在建造计算机的实际电路中却缺少不了它。在后面的章节中我们还会继续讨论三态门缓冲器，准备好继续学习！

图2-6中还有最后一个新概念要介绍。注意图中右上角的*真值表*。真值表是描述一个逻辑系统所有可能状态的简洁方式。在这里，三态缓冲器的输入端有两种逻辑状态，而 $\overline{\text{OE}}$ 控制端也有两种状态，所以对这个门来说一共有4种可能的组合。当 $\overline{\text{OE}}$ 为低电压时输出端与输入端相同；当 $\overline{\text{OE}}$ 为高电压时输出端为高阻抗状态，并使得输入不可见。因此，我们把这个器件所有可能的工作状态都用一个整齐的小表格描述出来了。

现在在我们的词汇里有了4种逻辑元件：与门、或门、非门和三态门。就像用DNA构成不同的生命一样，这些也是数字系统的基本元件。事实上，前面三种门经常被组合成一些稍微有点不同的门："与非门"（NAND）、"或非门"（NOR）和"异或门"（XOR）。这三种新的门被称为复合门，因为它们是由与门、或门和非门组合而成的。从电学上讲，由组合元件构成的电路和由基本元件构成的电路一样快，因为组合函数可以方便地用它们实现。我们只是从逻辑的角度对它们加以区别。

图2-7所示为两个复合门：与非门和或非门。与非门由一个与门后接一个非门构成。与非门的逻辑功能可以表述为：

- 当且仅当输入A为高电压且输入B为高电压时，输出C为低电压。

或非门的逻辑功能可以表述为：

- 当输入A为高电压、输入B为高电压、或两者都为高电压时，输出C为低电压。

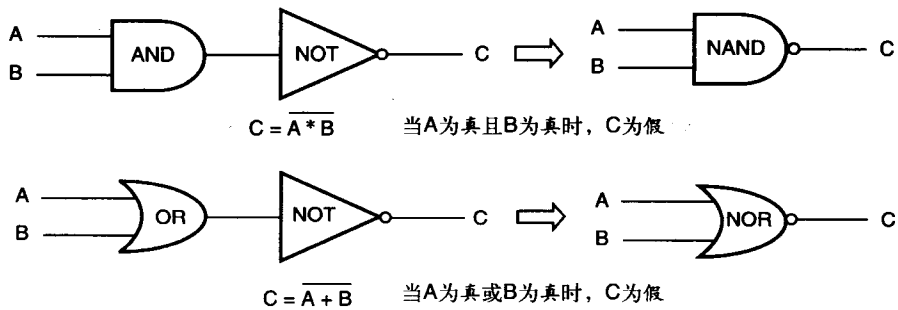
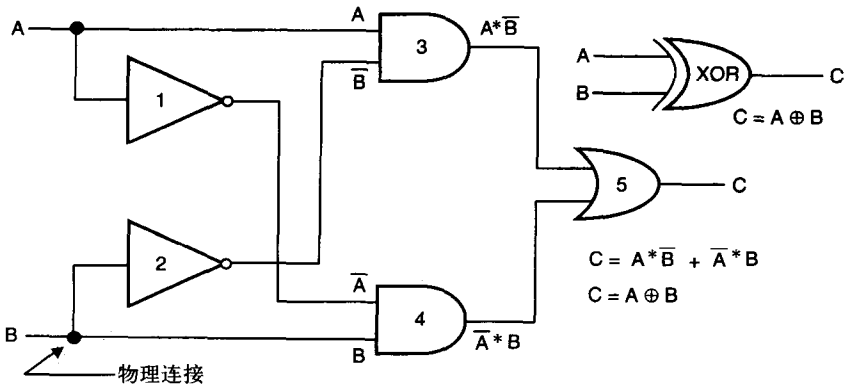


图2-7 与非门和或非门的示意图，它们分别为与门和非门的组合、或门和非门的组合 35

最后，我们再学习一个复合门结构，即异或门。异或门（exclusive OR gate）的英文速记符号为XOR（发音为“ex or”）。异或门和或门差不多，除了当输入A和B都为1时它的输出C为0，而不是1。

图2-8显示了异或复合门的电路图，它比我们在前面看到的任何门电路都要复杂，所以我们要花一些时间加以介绍。异或门有两个输入端（A和B）和一个输出端（C）。输入A直接连接到了与门3的输入，也连接到了非门1的输入并被求非。类似地，输入B连接到了与门4的输入，而它的补（非）则连接到了与门3的输入。因此，每个与门的输入都是变量A和B中的一个，以及另外一个变量的补，即 $\overline{A}$ 或 $\overline{B}$。这里插一句话，现在你应该感谢示意图中黑点的用处，没有它我们就无法区分相互连接的线和那些仅仅是相互交叉的线了。



当A为真或B为真，但它们又不同时为真时，C为真

图2-8 一个异或门（XOR）的电路示意图

因此，与门3的输出可以表示为逻辑表达式A* $\overline{B}$，类似地，与门4的输出为B* $\overline{A}$。最后，用或门5来组合这两个表达式，从而可以把输出变量C表示为关于两个输入变量A和B的函数：C = A* $\overline{B}$ + B* $\overline{A}$。这种复合异或门的图形符号如图2-8所示，它和或门很像，只是在或门的输入上加了一条线。异或的运算符号为一个外面带一个圆圈的加号。

让我们对这个电路进行仔细的考察，以验证它确实如我们所想的那样工作。假设A和B都是0，这意味着这两个与门都有一个值为0的输入，因此它们的输出必然也为0。第5个门即或门的输入均为0，因此它的输出也为0。如果A和B均为1，那么两个非门（第1个门和第2个门）对这个值求非就为0，这时的情形与前面一样，每个与门都有一个值为0的输入。

在第三种情况中，A为0，B为1，或者颠倒过来。无论是哪种情况，都有一个与门其两个
36 输入均为1，因此它的输出也为1。这说明至少有一个或门的输入将为1，所以或门的输出也为1。另一种表述异或门的方式是，当输入A为真或输入B为真，但它们又不同时为真时输出为真。

你可能会问自己，“这些都有什么用？”很快你就会看到，异或门可用于构造计算机的一个重要电路元件——加法电路。为了理解这一点，我们假设要对两个一位的二进制数求和。设这两个数为A和B，它们和的取值有下面的几种可能性：

- A = 0且B = 0: A + B = 0
- A = 1且B = 0: A + B = 1
- A = 0且B = 1: A + B = 1
- A = 1且B = 1: A + B = 0，进位1

只要我们能想办法处理好带进位的情况，这些条件看起来就非常像求A XOR B。

另外，请注意图2-8，其中给出了表示异或门的输出C的一个逻辑方程式，即：

$$C = \overline{A} * B + \overline{B} * A$$

这是我们称异或门为复合门的另一个原因，它可以用我们前面讨论过的多个“原子”门的逻辑表达式加以表示。

现在假设我们对异或门做少许修改，在它的输出端加上一个非门，从效果上看，我们得到了一个“同或门”（XNOR），对于这种门，当两个输入相同时其输出为1，当两个输入不同时其输出为0。因此，我们就构造出了一个可以检查信号相同性的电路元件。利用32个同或门，（经过适当的传输延迟）我们可以马上知道两个32位数是否相等。

就像我们可以用32个同或门来比较两个32位数一样，下面看看如果我们对两个32位数做逻辑与操作又会怎样。从其他的编程课上你已经知道可以对两个变量做逻辑与运算，其得到的结果为布尔真（TRUE）或假（FALSE）。但这样对两个32位数做与运算到底是什么意思呢？在这种情况下，与操作也被称为“按位与”（bitwise AND），因为它对两个数中每个对应的位做与运算。下面将参考图2-9加以说明。为了简单起见，我们给出了两个8位数的按位与运算，而不是32位数的按位与运算，所带来的差别仅仅是所使用的与门数目的不同。

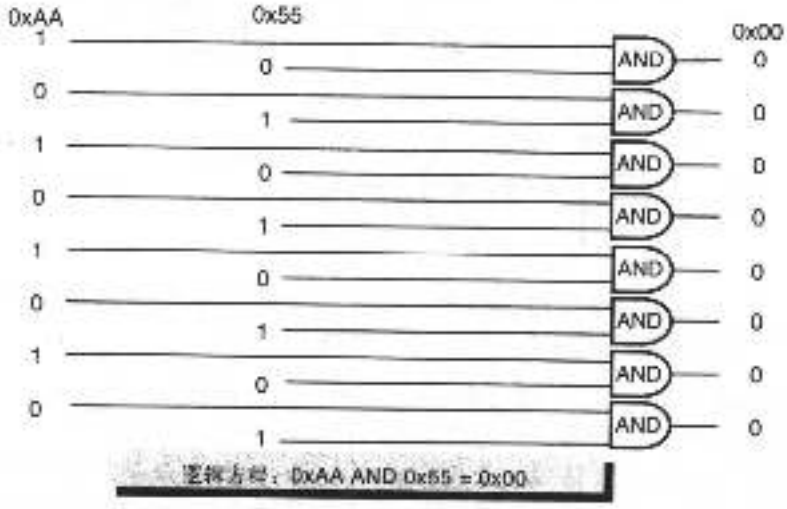


图2-9 两个8位（字节）二进制值的按位与运算。称其为按为与运算的原因是它对两个数的对应位进行操作。在C和C++语言中，十六进制数以前缀0x开始

37

在图2-9中，我们对两个单字节长度的值0xAA和0x55做了按位与运算。因为每个与门都有一个输入为1，而另一个输入为0，所以它们的输出都为0。在C/C++语言中，符号&表示按位与操作，因此我们可以写出下面的程序片段：

程序代码示例

```
char   inA = 0xAA;
char   inB = 0x55;
char   outC = inA & inB;
cout << "The bitwise ANDing of 0x55 and 0xAA = " << outC << endl;
```

如果我们真的写一个程序并运行它，那么得到的结果一定是0。作为对68 000汇编语言的预习，同样的方程用汇编语言可以写成：

程序代码示例

```
MOVE.B   #$AA,D0
ANDI.B   #$55,D0
```

第一条汇编指令将字节0xAA拷贝到了一个内部寄存器D0，第二条指令对0x55和D0中的内容（即0xAA）做了按位与操作。“ANDI.B”解释为，“对寄存器D0的字节值部分和立即（文字）值$55做逻辑与运算”。我们也可以写成“ANDI.W”或“ANDI.L”来指定其分别进行16位操作或32位操作。结果0现在存储于寄存器D0中。如果现在这对你没有任何意义，那么不要着急，它可能以后对你也没有任何意义（但也可能会有意义的）。

到现在为止，我们介绍的与门、或门，以及它们派生出来的门都只有两个输入端和一个输出端。其实与门和或门可以有任意多个输入，为了说明这一点，让我们看图2-10。

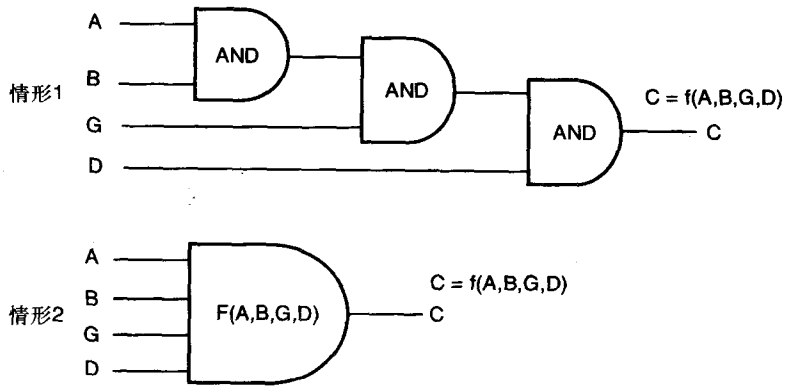


图2-10 多于两个输入端的与门的逻辑等价电路。虽然情形1和情形2逻辑功能完全相同，但是如果考虑逻辑速度（传输延迟），它们就不一样了

38

情形1所示为将三个具有两个输入端的与门连接起来，使得当且仅当A、B、G和D都为真时输出C为真。情形2中的电路看上去更为简单。事实上，我们可以设计出一个与情形1中一个门基本相同的电子电路对应于情形2，它们的唯一差别在于进行与操作的输入端的数目。然而，如果假设图2-10中所有门的传输延迟都一样，那么情形2中4个输入门的传输延迟将仅是情形1中的1/3左右。上述分析同样适用于或门。但是，它不适用于异或门，因为异或运算的数学函数决定了它一次仅能处理两个输入变量。

2.2 电子门描述

现在，大多数的门电路都是由集成电路（IC）做成的电子器件。你可能想知道这种门电路的内部是怎样的。这是个很自然的问题。尽管可能是一种有趣的尝试，但讲授晶体管理论或集成电路设计并不是本书的目的。我们让电子工程师保留一些秘密，毕竟，你已经看到他们写的程序是多么的混乱。下面我们迅速地瞥一眼这样的集成电路。图2-11是一个工业标准部件74LS00的图片。这里数字“74”指明了部件的逻辑系列，以及它们可以正常工作的温度范围。“74”系列部件是为商业用途设计的，而“54”系列则用于军事用途，并且可以在一个比较宽的温度范围内使用。字母“LS”是“低功耗肖特基”（low-power Schottky）的缩写，它

是几种标准集成电路制造工艺之一，最后一个标示“00”表示集成电路的封装类型，这种封装包含4个与非门，其中每个包含两个输入和一个输出。整个部件放在一个带14个引脚（每边7个）的塑料封装中。这被称为DIP封装，为“双列直插塑料”（dual-inline plastic）的缩写。由于显而易见的原因，像这样的集成电路部件也被称为“臭虫”（bug）。

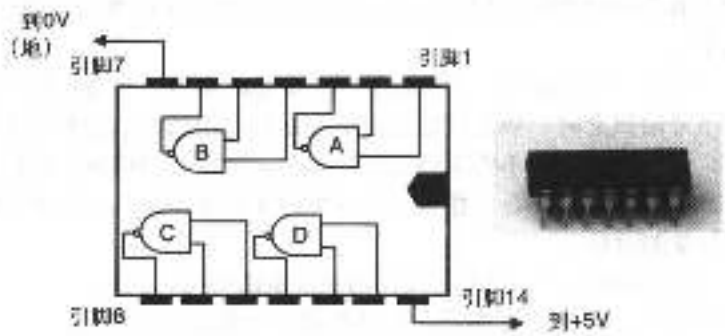


图2-11 4个双输入与非门（工业标准设计74LS00）。塑料封装中包含4个独立的与非门。集成电路被一个塑料封装包裹起来（显示在右边）。门A的输入引脚为1和2，输出引脚为3

图2-11所示的集成电路（IC）包含了4个独立的与非门。每个门有两个输入端和一个输出端，因此，封装结构需要为这些门准备12个输入/输出（I/O）引脚，另外还有一个引脚接电源和一个引脚接地。对应LS逻辑部件系列，电源为+5V。大约花10美分就可以买到一个这样的封装，而且其中任何一个门从输入改变到输出响应的传输延迟大约都是5ns。

我们仍然没有回答那个关于门的内部构造的问题。我曾说过晶体管能起到很好的开关作用，而开关电路又可以构造出很好的逻辑电路，因此我们现在看看它是怎么工作的。与其讨论以较老的工艺制造的LS逻辑电路系列，不如分析一种称为CMOS的更现代的工艺。CMOS的发音为“sea moss”，它是互补型金属氧化物硅（complementary metal-oxide silicon）的缩写。CMOS是当今主流的集成电路工艺，并且到可以预见的未来它一直都会被使用。几乎所有的现代微处理器都是用CMOS工艺制造的。此外，你也可能在有关数码相机的图像传感器的场合听到过这个术语。由于这是一个非常重要的工艺，所以我们应该花一些时间来理解它，哪怕是粗略地。 39

为了理解基本的CMOS构造，让我们回到我们所了解的机械开关上来，请看图2-12。

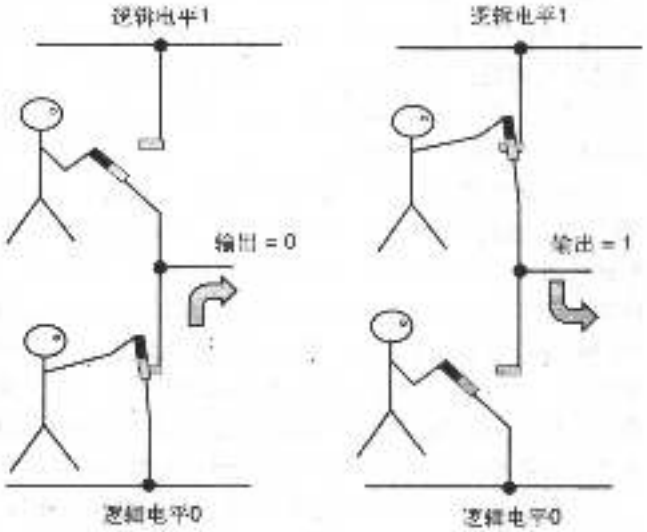


图2-12 用机械开关表示的CMOS开关电路

请不要太在意这幅图的绘图质量，即便只是用线段画出来的，也足以说明电路的工作过程了。想像在逻辑电平1和逻辑电平0之间有两个串联（相继）的机械开关。在实际电路中，逻辑1应该是电源，而逻辑0应该是电路的接地点（0V）。现在，很明显你不能在同一时间同时闭合两个开关，否则会产生不良后果。这就类似于我在5岁时拉直了一个回形针，想看看如果把它插到墙上的电灯插座里会怎么样。虽然我的经历更加让人印象深刻，也许比图2-12提供了更深刻的学习经历，但在概念上有相同的结果。

为了输出一个逻辑电平0，我们可以闭合连接到逻辑0的开关，断开连接到逻辑1的开关。此时电路输出端的观察者将会看到逻辑0，因为开关将它连接到了那个逻辑电平。另一种情况给了我们输出逻辑1的条件，即当闭合上面的开关而断开下面的开关时，我们会“看到”逻辑电平1。

现在，你应该已经弄清楚了一个重要的事实。当我们第一次讨论逻辑信号的上升沿概念和下降沿概念时，我说过这些沿必须是两个逻辑状态之间非常快速的转换。图2-12说明了为什么是这样。我们不希望出现两个开关同时闭合的情况，因此，如果逻辑转换过程控制着这些开关的开启或闭合，那么一个开关的开启过程和另一个开关的闭合过程之间的任何重叠都应尽量的短暂。

40
?
41

图2-13是一个CMOS逻辑门的实际电路结构，它是一个非门。首先要说的是，我是带着巨大的惶恐给大家展示这幅图的。原因之一是，我考虑到我正在透露过多的神圣而机密的信息，电子工程专业的兄弟姐妹们将会要求我予以补偿。另外，我不能肯定在计算机系统结构这个更大的专业范围内这是否是一个必须知道的内容。无论如何，让我们继续下去，你可以自己来判断。灰色圆圈内的符号所表示的两个电路设备被称为MOSFET，它是金属氧化物硅场效应晶体管（Metal Oxide Silicon Field-Effect Transistor）的缩写。MOSFET有两种常用的类型：n型或n沟道型，p型或p沟道型。n沟道型器件符号中有一个指向器件内部的箭头，而p沟道型器件符号中的箭头则指向外面。n型晶体管和p型晶体管的区别在于它们的制造工艺，具体的说，就是为使晶体管赋予所需电学特性而加入硅中的杂质的类型。

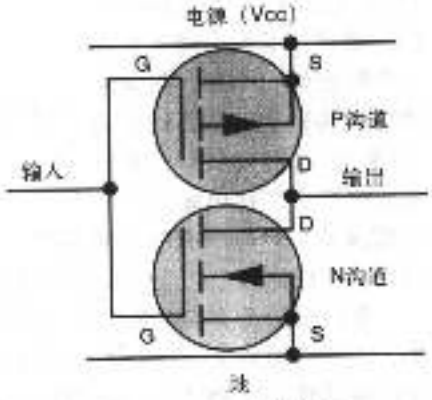


图2-13 CMOS电路结构

CMOS器件中的MOSFET晶体管是成对使用的，一个n型管和一个p型管在一起构成一对。除了前面说的制造工艺上的区别，这两种器件的主要区别是n沟道器件用于正电压情况下，而p沟道器件用于负电压情况下。如果你还不能理解它的意思，那么请继续往下看，我肯定下面会说得更清楚。总之，除了一个是正电压器件，另一个是负电压器件外，两种晶体管的功能几乎完全一样，因此习惯上也将它们称为互补型（complementary）。因此，将特征相似的一个n型MOSFET和一个p型MOSFET放在一起，就形成了一个互补对，即一个CMOS门。

每个器件都有三个不可或缺的端口（或极），标有字母“G”的是栅极（gate），标有字母“D”的是漏极（drain），而字母“S”旁的端口称为源极（source）。有时还要考虑第4个端口，那就是衬底（substrate），也称体极（body），用字母“B”表示。有时候会将这个端口引出来作为一个独立控制端，但在我们的电路结构中它连接到了源极上，因此不需要考虑。为了了解CMOS门是怎么工作的，我们应该快速地看看一个MOSFET器件的简单行为图。图2-14大致给出了一个MOSFET器件的漏极到源极电阻（R_{DS}）随栅极和源极间电压V_{GS}的变化关系。因为我们现在还没有真正给出电阻的定义，所以我也不期望上述介绍有多大效果，但从概念上讲电阻与我们前面讨论三态门时用到的电学阻抗是完全一样的。

考察图2-14中n沟道器件的曲线，我们看到当栅极和源极之间的电压增大时，器件的电阻值会从几千万欧姆（ohm）迅速下降到10欧姆左右。实际上，能够导致这种剧烈变化的栅极电压变化范围就恰好是从逻辑0到逻辑1的电压摆动。考虑图2-12所示的电路功能的类比，升高栅极电压就相当于闭合开关。可见，当栅极电压比源极电压小得更多时，p沟道器件会发生

与上面所说的类似的情况。换句话说，它会显示出互补的行为。现在，我们把这些东西放在一起，就可以理解CMOS门的普遍规律和图2-13中的特殊形式了。

再回到图2-13，记得它是一个非门，我们来看看为什么它是一个非门。假设栅极电压是逻辑0，n型晶体管必然是断开的开关，因为电阻特别大（因为V_{GS}几乎是0）。然而，由于栅极电压比源极电压小得多（$-V_{GS}$非常大），所以互补器件p型晶体管的电阻就特别小，这就形成了图2-12中的闭合开关。

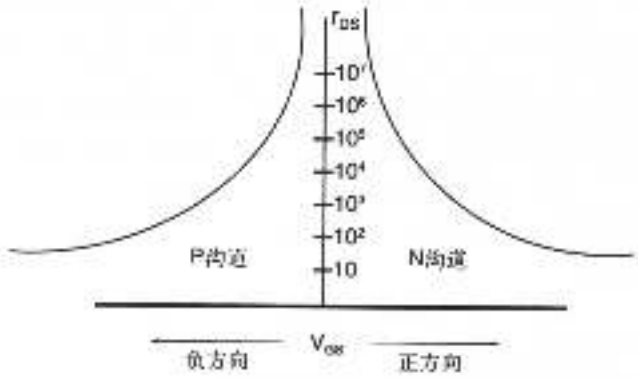


图2-14 n沟道和p沟道MOSFET器件的电学特性。用对数坐标将器件的电阻值绘制成一个关于器件栅极电压的函数。来自Watson[2]

这样，在门的输出端，我们看到一个低电阻（闭合的开关）连着逻辑1，一个高电阻（断开的开关）连着逻辑0，门的输出是逻辑1。因此，当门的输入为逻辑0时，其输出为逻辑1。你应该能够像上面那样分析互补的另一种情况，即输入为逻辑1的情况。因此我们可以把CMOS非门的行为总结成：当输入为逻辑0时，输出为电源电压，即逻辑1；当输入为逻辑1时，输出为接地电压，即逻辑0。

图2-14展示了MOS晶体管的电学特性。考察图右边的N沟道器件，当栅极相对于源极的电压V_{GS}增大（或者说向正方向变化）时，漏极和源极间的电阻呈指数级下降。相反，当V_{GS}趋向于0（或者说向负方向变化）时，漏极和源极间的电阻趋向于无穷大。基本上，当V_{GS}为0或者负数时，我们就有一个断开的开关；而当V_{GS}为几伏特电压时，我们就有一个几乎是闭合的开关。P沟道器件的行为和N沟道的一样，除了电压的正负极是反过来的。

在我们离开CMOS门的电学行为这一话题之前，你可能会问自己："当栅极相对于源极的电压不是逻辑1或0，而在两者之间，会发生什么情况呢？"换句话说，当栅极的逻辑输入的上升沿或下降沿处在两个逻辑状态间进行转换时会怎样？根据图2-14中的曲线，如果电阻非常高，或者不是很低，就像《金发女孩和三只熊》中的粥的温度（译者注：童话故事，指粥既不烫也不凉）。在这种情况下，就形成了一条从电源到地的电流通路，因此会浪费一些电能。 42
幸运的是，状态间的转换过程很快，虽然确实还是有一些能量损耗。

这种情况会带来多坏的影响呢？别忘了现代的微处理器中有千万量级的CMOS门，而这些门的大部分都以每秒十亿次或更快的速度切换逻辑状态。图2-15应该能给你一些关于能量损耗的直观印象。

注意两件事情。首先，这些现代微处理器确实都发热。事实上，它们散发的能量相当于一个普通的60~75瓦的电灯泡。此外，对每个系列的处理器，当提高时钟频率使它运行更快时，我们可以看到功耗也随之变大。

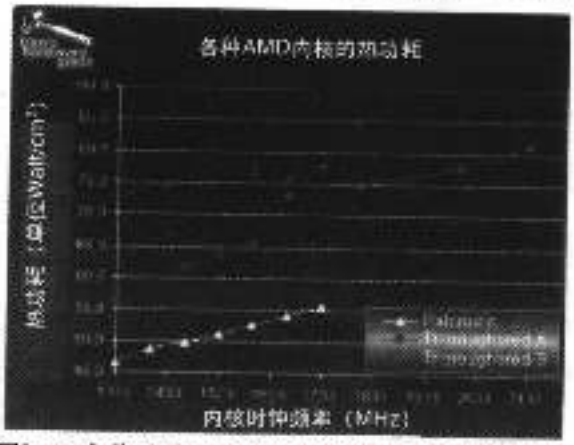


图2-15 各种系列AMD处理器的能量损耗随时钟速率的变化曲线。来自www.tomshardware.com[3]

你也可能会问另一个问题："假设我们让时钟频率变小，直至让它相当小，甚至停下来，会产生相反的效果吗？"很明显，通过减慢时钟速度或者关掉芯片的部分电路，我们可以相应地减少芯片的功率需求。这是用在笔记本电脑上的一个重要策略，它使得电池能支持电脑在从纽约到洛杉矶的飞机上持续工作。这也是许多用于其他嵌入式应用的微处理器的策略，这样的微处理器能系在环绕驼鹿颈脖的项圈子上以跟踪它两年时间多的迁徙过程，而其所用的能量还不超过一节AAA电池。不要惊奇CMOS是这么的普及！

好了，我们已经看了一个非门的结构。那是一种非常简单的门，其他稍微复杂一些的门是怎样的呢？下面我们用CMOS做一个与非门。

回忆一下与非门的功能，当所有输入是逻辑1时它的输出为0。在图2-16中，我们看到，如果所有的输入为逻辑1，那么所有的n沟道器件开启，成为低电阻状态。所有的p沟道器件则关闭，因此从输出往器件内部看，可以看到一个低电阻的通路连接到接地点（逻辑0）。现在，如果4个输入A、B、C和D中有任何一个为逻辑状态0，那么它的n沟道MOSFET处于高电阻状态，而p沟道器件处于低电阻状态。这将阻挡住接地点与输出的连接，而通过p沟道器件形成了一个连着电源的低电阻通路。

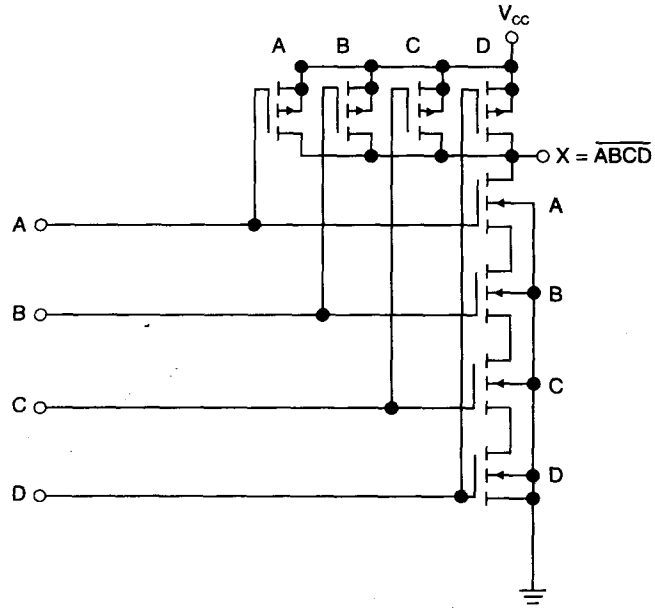


图2-16 一个具有4个输入的CMOS与非门的示意图。来自Fairchild Semiconductor[4]

希望现在你能对自己刚开始掌握的硬件技巧和洞察力更有信心。让我们分析一个电路结构，见图2-17，这个电路常被称为"莎士比亚电路"。你可以任意想像这个设计的深层含义。而我在这里引入这个例子是为了证明不是所有的计算机设计者都是没有幽默感的怪人。

43

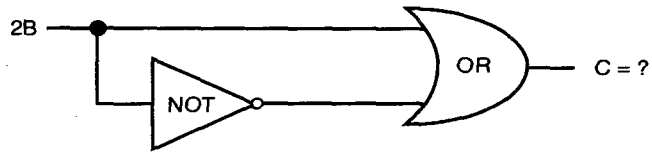


图2-17 莎士比亚电路

2.3 真值表

本章要讨论的最后一个概念是*真值表*（truth table）。当我们前面讨论三态逻辑门的行为时已对真值表做了简单介绍。然而，现在分析了一些电路后正式地讨论它会更合适。真值表就像它的名字所暗示的那样，是表示一个逻辑门或系统真/假（TRUE/FALSE）条件的一个表格。对于我们已经学习了的各种逻辑门（与门、或门、与非门、异或门和同或门），我们已经用口头语言的形式描述了它们的功能。

例如，对于与门，我们说："当且仅当两个输入均为1时与门的输出为1。"这确实表达了它的逻辑含义，但我们需要一种更好的方式来表达，从而设计出更复杂的系统。我们所用的方法就是

为逻辑函数创建一个真值表。图2-18显示了我们迄今所学过的5种逻辑门电路对应的真值表。非门太简单了，所以这里没有包括它。我们已经看到过三态门的真值表了，所以它不在此图中。

AND

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

OR

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

NAND

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

XOR

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

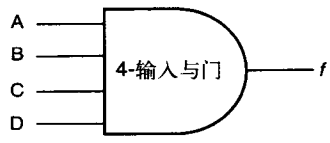
NOR

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

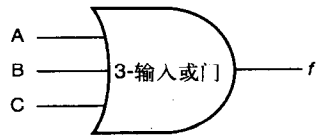
图2-18 逻辑函数AND、OR、NAND、NOR和XOR的真值表表示

真值表给出了所有可能的输入变量A和B所对应的输出变量C的值。由于有两个相互独立的输入变量，所以它们的取值可以是4种组合中的任意一个。参考图2-18，我们看到：当且仅当A和B都是逻辑1时与门的输出C才是1，而所有其他的输入组合都会使C等于0。类似地，如果两个输入变量中有任何一个为1（包括两者都为1），则或门的输出也等于1。真值表为我们提供了一种简明而形象的方法来表达逻辑函数的功能。

假设我们的与门有三四个输入，那么对应的真值表看上去会怎样？如果我们有3个独立的输入变量A、B和C，那么真值表中将有8种可能的输入组合（或者有八行）来代表所有可能的输入变量组合。对于一个3输入与门，这8个输入条件中仅有一个（即A = 1, B = 1, C = 1）会使它的输出端产生1。对于4个输入变量，在真值表中将有16个可能的表项。这样，推而广之，对于一个有N个独立输入变量的系统，真值表将包含有2^N个表项。又是那个麻烦的二进制数系统！图2-19显示了一个4-输入与门和一个3-输入或门的逻辑符号以及对应的真值表。在市场上，还可能可以找到有5个或更多输入端的与门、与非门、或门以及或非门电路。还有一种与非门电路，它被设计成可以扩展到任意多个输入信号，虽然实际上并没有这种必要。



A	B	C	D	f
0	0	0	0	0
1	0	0	0	0
0	1	0	0	0
1	1	0	0	0
0	0	1	0	0
1	0	1	0	0
0	1	1	0	0
1	1	1	0	0
0	0	0	1	0
1	0	0	1	0
0	1	0	1	0
1	1	0	1	0
0	0	1	1	0
1	0	1	1	0
0	1	1	1	0
1	1	1	1	1



A	B	C	f
0	0	0	0
1	0	0	1
0	1	0	1
1	1	0	1
0	0	1	1
1	0	1	1
0	1	1	1
1	1	1	1

图2-19 一个4-输入与门和一个3-输入或门的真值表和门线图

异或门是一个例外，因为它的定义决定了它只允许有两个输入。然而我们也可以设计出一种有（比如说）5个输入（从A到E）和1个输出f的电路，它的特点是当且仅当所有输入都相同时$f = 0$。当然这就不是异或门了，它是某种其他的电路。

考虑图2-20中的那个灰色方块，它表示一个任意数字系统，比如说用于电梯控制、家庭供暖和空调系统或火灾报警控制器的电路。我们最终的目的是用我们刚才讨论过的那些门电路来设计出这个灰盒子里的电路。无论它是什么，

我们都要首先定义每个输出变量和它相应的输入变量之间的逻辑关系。由于这个灰盒子有8个输入变量，所以我们将从这个电路对应的256个真值表表项开始。换句话说，要设计这个电路首先就要指定每种可能的输入（从a到h）状态所对应的X、Y和Z（输出变量）的值。因此，如果我们设计的是一个大楼的供暖系统，而且一个来自温度传感器的信号输入表明房间的温度低于调温器上所设定的值，那么这将触发一个适当的输出反应（比如说点燃某个火炉）。

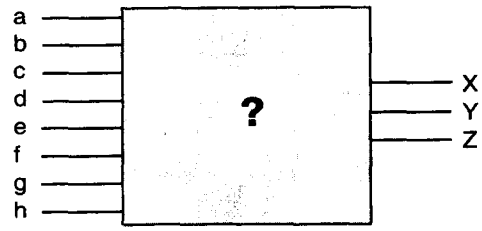


图2-20 一个数字系统设计。对每个输出变量X、Y和Z，都需要根据输入变量的所有组合状态用一个独立的真值表加以描述

参考图2-20，我们的设计过程可以从创建一个256行、11列的表格开始，其中每列对应8个输入变量和3个输出变量中的一个。下一步，我们将艰苦地把256种输入变量组合（从00000000到11111111）填入到真值表中。最后，也是真正工程的开始，为真值表中的每一行决定如何给输出变量赋值。

45 在第3章中，我们将更详细地讨论这些系统的设计过程。现在，作为一个总结，我们发现我们已经以两种既不相同又有联系的方式使用了真值表：

1. 真值表可以作为一种表格的形式来描述一个标准门（与门、或门、与非门、或非门、异或门或三态门）的逻辑行为。

2. 我们可以通过使用真值表来描述任何一个复杂的数字系统，以说明该系统将如何工作。

总结

本章主要内容如下：

- 基本的门电路：与门、或门和非门，并看到了很多可以由它们导出的更复杂的门电路，比如与非门、或非门及异或门。
- 动态（即随时间变化）的逻辑值可以用一个逻辑电平或电压对时间的图来表示，这种图称为*波形*。
- 如何由电子开关元件、MOSFET晶体管产生出CMOS逻辑门。
- 如何将一个门或数字电路的逻辑行为表达为一个*真值表*。

参考文献

[1] http://www.dnaftb.org/dnaftb/20/concept/index.html.

[2] J. Watson, *An Introduction to Field Effect Transistors*, published by Siliconix, Inc., Santa Clara, CA, 1970, p. 91.

46 [3] http://www.tomshardware.com.

[4] Fairchild Semiconductor Corp., Application Note 77, *CMOS, The Ideal Logic Family*, January, 1983.

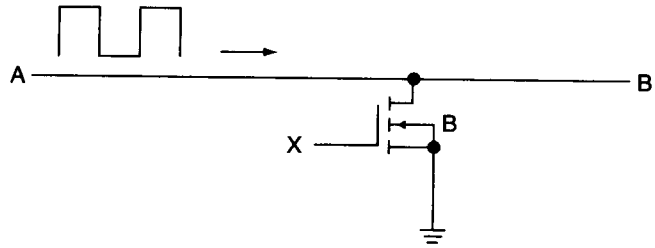
习题

1. 考虑图1-14中简单的与电路和图2-5中的或电路。改变图中的逻辑含义，使得断开的开关（无电流流过）为真，闭合的开关为假。同时，改变灯泡的含义，当灯不亮时认为输出为真，而当它亮时认为是假。在这种表示负逻辑的新条件下，两个电路分别表示什么逻辑功能？
2. 用CMOS晶体管画出2-输入与门的电路结构。提示：用图2-16中的电路作为出发点。
3. 为下面的逻辑等式构造真值表：

a. $F = a*\bar{b}*\bar{c} + b*\bar{a}$

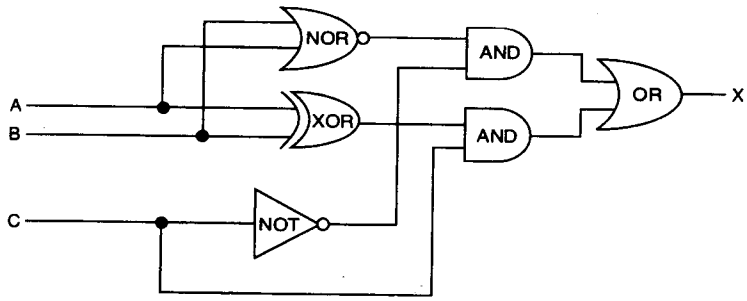
b. $F = a*b + \bar{a}*c + b*c$

4. 当点X处的电压升高到逻辑电平1时，它对点A和B之间的信号会产生什么影响？



5. 为奇偶检测电路（parity detection circuit）构造真值表。这个电路有4个输入变量（a到d）和一个输出变量X。输入d是控制信号，当d=1时电路检测奇信号，当d=0时电路检测偶信号。信号的奇偶性由输入端a、b和c确定，奇信号表示它们三个中有奇数个1，偶信号意味着它们中有偶数个1。例如，如果d=1且(a, b, c)=(1, 0, 0)，那么输出X=1，因为这是一个奇信号。

6. 画出对应于右图显示的逻辑门电路的真值表。



47

7. 图2-8所示为异或函数（即方程a ⊕ b = X）的门电路结构。假设你也可以将异或函数表示为：

$$a \oplus b = \sim[\sim(\bar{a}*b)*\sim(a*\bar{b})]$$

请仅使用与非门重新设计这个电路。

8. 考虑图2-3中所示的电路，假设我们将图中的与门用（a）或门和（b）异或门来代替了。那么当输入A=0和A=1时，输出波形将分别是怎样的？ 48

第3章　异步逻辑简介

学习目标

- 使用布尔代数定律简化和计算逻辑公式，并将它们转化为逻辑门设计；
- 创建真值表来描述数字电路的行为；
- 用卡诺图来简化逻辑设计；
- 描述逻辑信号的物理属性，比如上升时间、下降时间和脉冲宽度；
- 用频率和周期来表示系统时钟信号；
- 通过常用的工程单位记号，如Kilo（千）、Mega（兆）、Giga（吉）、milli（毫）、micro（微）和nano（纳）来表示不同数量级的时间和频率。

3.1　引言

在我们将自己陷入严格的逻辑分析和设计之前，最好先喘口气儿，反思一下所有这些都从何而来。我们可能有一种错误的印象，逻辑是在硅谷随着晶体管的发明而诞生和发展起来的。

我们通常将现代逻辑分析的起源追溯到生于公元前384年的古希腊哲学家亚里士多德，他被公认为现代逻辑思想之父。因此，如果要完全精确（也可能有点慷慨），我们应该称亚里士多德为现代数字计算机之父。然而，当我们仔细看刚才介绍的与门电路有关的数学时，我们会发现很难将它和亚里士多德的逻辑联系到一起。不过下面的简单例子可以给我们一些提示。



图3-1　亚里士多德

亚里士多德逻辑的核心是演绎（deduction）的概念。如果你看过早期福尔摩斯侦探的电影，那么你可能更熟悉演绎这个概念，但演绎推理的思想并不是福尔摩斯发明的，而是亚里士多德发明的。

亚里士多德提出一个演绎断言应该有“假设的条件”或者断言的前提（premise），而“必然得到的结果”就是结论[1]。

一个经常被引用的例子是：

1. 人总会死的；
2. 希腊人是人；
3. 因此，希腊人也会死的。

我们可以将这个例子用一种稍微不同的形式表述：

1. 令A表示“人类总会死”这个命题的状态，即“人类总会死”可能是真也可能是假；
2. 令B表示希腊人的状态，“希腊人是人类的一种”可能为真也可能为假；
3. 因此，结论为真的唯一条件是，如果A为真且B为真那么希腊人也会死，或者说，C（表示希腊人是否会死）= A * B。



图3-2　乔治·布尔

我们可以将处理逻辑表达式的能力追溯到英国数学家和逻辑学家

乔治·布尔（1816–1864）。

“1854年，布尔在《the Laws of Thought》上发表了一篇研究报告，基于它人们创建了逻辑和概率的数学理论。布尔用一种新的方式将逻辑简化为简单的代数符号，并将它与数学运算结合起来。他指出了代数符号和逻辑形式符号的类似性，从而开创了称为布尔代数（Boolean algebra）的逻辑代数学科。现在，布尔代数已广泛应用于计算机建造、开关电路等方面。”[2]

布尔代数给我们提供了设计复杂逻辑系统的一系列工具，并且保证设计出的电路将按我们想要的方式进行工作。此外，就像你将看到的，设计数字电路的过程通常会导致相当冗余的结构，从而需要对其加以简化。布尔代数为我们提供了简化电路设计所需的工具，并确保它能以最少的硬件实现设计时想要的功能。作为电子工程师，这恰恰是我们最需要的分析工具，因为分派给我们的任务经常是要求用最低的成本完成有效的设计。

在设计数字计算系统有关的工作中，有两个明显不同的二进制系统需要我们熟悉。在本书前面的内容中，我们已经介绍了二进制数体系，它很方便，因为我们需要计算机能够操作超过一位长度的数字。另一个部分就是对硬件进行如同对硬件数字系统的操作。为了理解如何处理这些数字，我们需要学习二进制逻辑代数即布尔代数的一些原理。

因此，这个过程的第一步就是给出一些布尔代数的定律。在大多数情况下，这些定律都是显而易见的。而对另一些情况，你可能需要仔细思考，直到意识到我们在逻辑运算中涉及的变量仅能取两种可能的值，而不是对那些可取连续数值的变量进行加法和乘法。此外，请注意表示逻辑“非”的记号并没有统一，因此可能有多种表示方法。这部分是历史原因造成的，因为可打印的基本ASCII字符集中并没有提供一个简单的方式来表示反变量，即在变量上面画一条横线加以表示，如第2章图2-4所示。因此，你可能会看到几种不同的“非”条件的表示。例如，/A、~A、*A和A*都常被用于表示“非A”（NOT A）。我不打算使用*A和A*，因为它太容易和与符号混淆了。然而，为了使等式更容易理解，我偶尔会使用~A或/A来表示非A的含义。在大多数情况下，我会用字母上面加横线的方式来表示非运算。很抱歉没能统一。 50

3.2 布尔代数定律

互补律

• 第一互补律	若A=0，则 $\overline{A}=1$；若A=1，则 $\overline{A}=0$
• 第二互补律	$A * \overline{A}=0$
• 第三互补律	$A + \overline{A}=1$
• 双重互补律	$/(\overline{A}) = //A = A$

交换律

• AND交换律	A * B = B * A
• OR交换律	A + B = B + A

结合律

• AND结合律	A * (B*C) = C*(A * B)
• OR结合律	A + (B + C) = C + (A + B)

结合律使我们可以组合三个以上的变量，它告诉我们可以以任何顺序对变量进行组合而不影响最后的结果。正如我们在第2章所看到的，正是这条定律允许我们通过组合两个2-输入或门得到一个逻辑上等价的3-输入或门。应该注意，当我们说逻辑等价的时候其实忽略了电学上的区别。正如我们已经看到的，在任何实际的系统中都要考虑逻辑门的时序特征。

分配律

• 第一分配律	A * (B + C) = (A * B) + (A * C)
• 第二分配律	A + (B * C) = (A + B) * (A + C)

分配律看上去非常像代数中的提取公因子和乘法展开。

重言律

51

• 第一重言律：A * A = A	若A=1，则A*A=1；若A=0，则A*A=0。因此表达式简化为A
• 第二重言律：A + A = A	若A=1，则1 + 1=1；若A=0，则0+0=0。同样，表达式简化为A

带常数的重言律

- A + 1 = 1
- A * 1 = A
- A * 0 = 0
- A + 0 = A

吸收律

• 第一吸收律	A * (A + B) = A
• 第二吸收律	A + (A * B) = A

得出这个公式需要点技巧。考虑表达式：A * (A + B)。

如果A = 1，它将变成1 * (1 + B)。根据带常数的重言律，1 + B = 1，因此就剩下1 + 1 = 1。如果A = 0，第一个表达式现在就会变成0 * (0 + B)。再次应用带常数的重言律，它会缩减为0 * B，它必然等于0。因此，无论在哪种情况下，表达式的值都等于A的值，而B的值对结果不产生影响。

德·摩根定理

- 情况1：$\overline{(A*B)} = \overline{A} + \overline{B}$
- 情况2：$\overline{(A+B)} = \overline{A} * \overline{B}$

德·摩根定理非常重要，因为它表明了与函数和或函数的关系，以及正逻辑和负逻辑的概念。而且，德·摩根定理也表明任何用与门和反相器（非门）构成的逻辑函数都可被或门和反相器组成的电路所复制。此外，请注意上面两个方程的左边都恰好分别是组合逻辑函数NAND和NOR。

在继续学习之前，我们应该讨论一下德·摩根定理和逻辑极性的关系。到目前为止，我们一直采用的是正向（高电压信号为1）和负向（低电压信号为0）的习惯。这是介绍逻辑知

识的很好的方式，因为真/假、1/0和高/低比较容易看成是一致的概念。然而，当真和假只是逻辑意义的时候，从电路角度来看我们完全可以任意地定义逻辑规则。

这并不是说如果我们将电学上的1和0交换，一个与门电路就仍然代表一个与门。不会这样的，它其实变成了一个或门。类似地，如果我们交换1和0使得低电压表示1，高电压表示0，那么或门将变成与门。你可以根据第1课中学到的与门与或门的真值表来自己验证这一点。如果将与门的真值表中所有1换成0，所有0换成1，那么你将看到这其实变成了或门的真值表（负逻辑）。用同样的方法试试或门，你会发现最后得到了负逻辑下的与门。

应记住的重要一点是，如果采用逻辑1是高电压的正逻辑，相同的电子电路将实现与门的 52
逻辑行为。这样，逻辑1可以是任何高于+3V的电压，而逻辑0可以是任何低于0.5V的电压。我们称这个是*正逻辑*。不过，如果我们反过来定义什么电压代表1和什么电压代表0，那么同样的电路就变成了逻辑或函数（*负逻辑*，negative logic）。

因此，在数字系统中，我们通常会按需要定义真和假，以便能实现出最有效的电路设计。既然不能让真或假一直保持相同的电学意义，我们通常会用到一个术语*置有效*（assert）。当说一个信号被置有效，就表明它处于活动状态，处于活动状态可通过从低电位变为高电位，或者相反从高电位变为低电位来达到。一会儿当我们讨论存储器系统的时候，你将会看到大多数的存储器控制信号都是低有效的，但存储器单元的地址和存储于地址的数据信号则是高有效的。在第2章我们讨论三态门的逻辑行为时你已经看到了这种情况，其中当输出使能端（$\overline{OE}$）被置为低有效时三态门的输出进入低阻状态。这并不意味着$\overline{OE}$信号为假（或负逻辑里的真），它只表示这个信号在低状态时进入活动状态。

在图2-20中，我们考虑了一个最一般的逻辑系统设计的情况。其中，3个输出变量中的每一个都被定义成了多达8个输入变量的函数，即X = *f* (a, b, c, d, e, f, g, h)，等等。请注意输出变量X、Y和Z是关于所有输入变量a到h或其中一部分的各自独立的函数。现在我们需要解决的问题可分为下面4个部分：

1. 我们如何用真值表来描述我们想要这个数字系统来实现的功能？这恰好是定义设计。

2. 一旦有了想要的功能（由真值表所定义），我们怎样用布尔代数的法则将真值表转换为这个系统的布尔代数表达式？

3. 当我们将这个设计表示为一系列方程之后，能否使用布尔代数的某些定律来简化这些方程？

4. 最后，当我们用最简单的代数形式描述了这个系统的方程之后，该怎样将方程转化为一个真实的电路设计？

稍过一会儿，我们将看到如果依次解决这些问题，以及如何用布尔代数的定律将方程加以简化。但是，如果事先知道输出Y仅依赖于输入信号c、d和g，那么我们可以马上将Y的真值表中的输入变量限制为3个，从而简化这个设计任务。很快我们将看到，一般的情况可以变成简化的情况，所以无论用哪种方式最终都能达到相同的结果。经常遇到的情况往往是你事先并不知道某个输出变量和某些输入变量之间没有关系，而在进行了所有的代数化简工作之后你才发现这样的情况。

图3-3是一个随意的真值表设计例子，它描述的不是一个实际的系统，至少我还想不出它能代表哪种实际的系统。我构造出这个真值表只是为了进行一遍上述的逻辑简化过程。

输出E和F是两个独立的变量，它们是依赖于输入变量A到D的两个函数。由于有4个输入变量，所以真值表中总共包括了2^4即16种可能的组合，代表了所有可能的输入变量的取值情况。也请注意每个输入变量的取值是如何写在同一列中的，用这种方法可以保证没有漏掉或者重复一些情况。

因为这是个自己造的例子，所以输出变量和输入变量之间的关系并没有实际的意义，也就是说我
53 只是随意地放了一些1和0在E和F的列中使这个例子看上去有点意思。如果这个例题是一个实际数字设计中的一部分，设计者就应该考虑真值表中的每一行并决定每个输出变量应对特定的输入信号组合如何相应。

例如，假设我们正在设计一个简单的防盗报警系统控制器。不妨认为输出E控制一个房间内的蜂鸣器，而输出F控制一个足以唤醒邻居的大音量喇叭。输入A、B和C是检测入侵者的传感器，输入D是控制防盗报警系统是否开启的按钮。如果D = 0，那么整个系统不工作，如果D = 1，则系统工作，喇叭会响。

A	B	C	D	E	F
0	0	0	0	0	1
1	0	0	0	0	0
0	1	0	0	0	0
1	1	0	0	0	0
0	0	1	0	1	0
1	0	1	0	1	0
0	1	1	0	0	1
1	1	1	0	0	0
0	0	0	1	0	0
1	0	0	1	0	0
0	1	0	1	0	0
1	1	0	1	0	0
0	0	1	1	0	0
1	0	1	1	0	0
0	1	1	1	0	0
1	1	1	1	1	0

$E = \overline{A}*\overline{B}\ C*\overline{D} + A*\overline{B}*C*\overline{D} + A*B*C*D$

$F = \overline{A}*\overline{B}*\overline{C}*\overline{D} + \overline{A}*B*C*\overline{D}$

图3-3 一个数字系统设计例子中的真值表。输出E和F可根据真值表写成乘积和（最小项）形式

在这个例子中，对于D = 0时的所有真值表中的条件，我们都不允许输出有效，不论输入A、B和C处于什么状态。而当D = 1时，我们需要考虑其他输入的影响。这时，真值表中的每一行都给出了一组新的条件，你需要对输出值进行独立的评估。有时候，如果某个变量（D = 0）对系统有全局影响，你可以作出一些显然的决定。

虽然图3-3不代表一个实际系统，让我们还是暂时假设它是真实的。我们考虑输出变量F，它的逻辑方程为：

$$F = \overline{A} * \overline{B} * \overline{C} * \overline{D} + \overline{A} * B * C * \overline{D}$$

第1项　　第2项

该方程告诉我们F在两组不同的输入情况下为真。无论是所有输入都为假（第1项），还是A和D为假而B和C为真（第2项），都会使输出变量F为真。我们怎么知道是这样的呢？是我们设计它这么工作的！作为负责这个数字设计的工程师，我们指定在这两组输入情况下输出F为真。

我们称上面的逻辑方程为*乘积和形式*（sum of products form），或*最小项形式*（minterm form）。还有另一种称为*最大项形式*（maxterm form），它可以表示为*和的乘积*（product of sums）。这两种形式可以根据德·摩根定理相互转换。为了达到效果，我们将仅讨论最小项形式。

记住，这个真值表是某个数字系统设计的例子。它在某种程度上粗糙地表示了这个系统的设计描述。因此，我们并不知道为什么输出变量E和F对应的列上有些行为0，而有些行为1，
54 这些都来自于系统的设计要求。我们是工程师，而那个正处于工程设计阶段。紧接在设计阶段后面的是实现阶段。因此，下面让我们解决这个设计的实现。

参考图3-3，我们看到输出变量E在输入变量A到D的三种可能组合情况下为真：

1. $\overline{A} * \overline{B} * C * \overline{D}$
2. $A * \overline{B} * C * \overline{D}$
3. $A * B * C * D$

我们可以把这个关系用一个逻辑方程加以表示：

$$E = \overline{A} * \overline{B} * C * \overline{D} + A * \overline{B} * C * \overline{D} + A * B * C * D$$

对三个与运算的结果进行或运算意味着，三个与项中任何一个都可以导致E为真。因此，对于组合（combination），我们用与运算。对于聚合（aggregation），我们用或运算。类似地，

我们可以把F的值表示为：

$$F = \overline{A}*\overline{B}*\overline{C}*\overline{D} + \overline{A}*B*C*D$$

到现在为止，将这些方程转化为逻辑门电路并不很困难，这样我们就把这个数字逻辑系统做出来了。是这样吗？事实上，这种看法在一定程度上是对的。我们仍然不知道这些项中是否有冗余，我们是否可以消除这些冗余从而使电路更容易建造出来。按真值表来建造系统自然会带来冗余，因为我们单独考虑真值表中的每一行，某些重复自然有可能混进我们的方程中。

使用布尔代数定律，我们可以对这些方程做进一步处理将其简化。最简单的有冗余项的方程形式是 $A*B + A*\overline{B}$。

很容易证明 $A*B + A*\overline{B} = A$。为什么呢？

1. 首先用分配律： $A*B + A*\overline{B} = A*(B + B)$

2. 再根据第3互补律： $B + \overline{B} = 1$

3. 最后， $A*1 = A$

因此，如果我们能对某些项进行组合从而“提取”一些公共的与项，并且剩下的项正好是一个表达式和其自身补的或，那么我们就可以直接把这个剩下的项扔掉。

3.3 卡诺图

在一篇经典的论文中，Karnaugh[3]（发音为CAR NO，译为“卡诺”）提出了一种图形化的方法用于简化由真值表得到的乘积和方程，而不需要直接使用布尔代数，卡诺图就是对下面布尔代数简化公式的图形解决方法：

$$A*B + A*\overline{B} = A$$

由于真值表的构造方式所自然导致的冗余，这种简化过程是逻辑表达式简化中最常见的。

可以遵循一些简单的规则把卡诺图（也称K图）构造出来，图3-4显示了含3个变量、4个变量和5个变量的K图的构造过程。

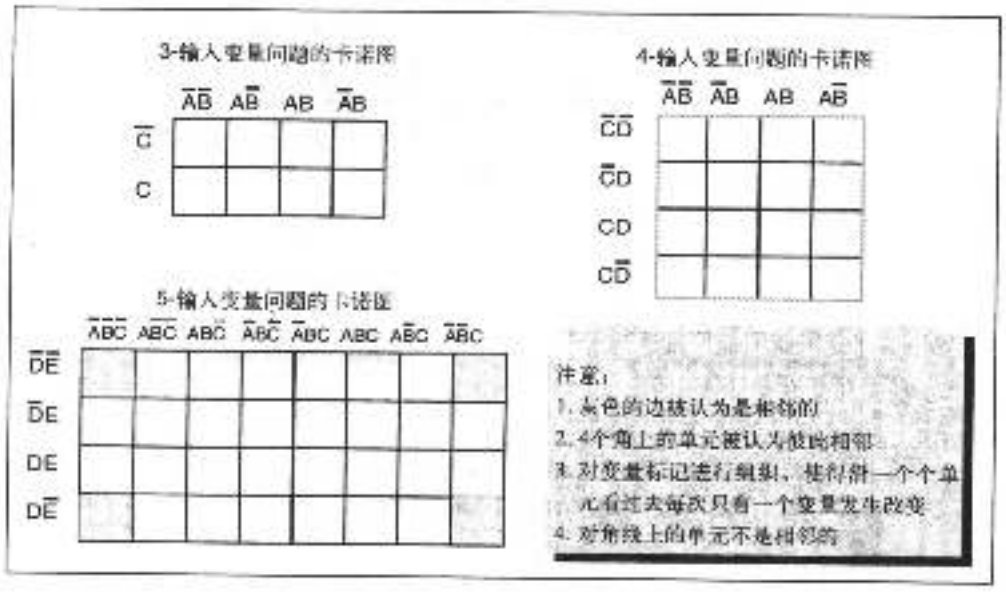


图3-4 创建3个变量、4个变量和5个变量的卡诺图的格式

请看含4个输入变量的K图。注意图中垂直的深灰色边和水平的浅灰色边，它们应分别被
55 看成是相邻的。换句话说，整个图在水平方向和垂直方向都可以被认为是卷成了圆筒形状。

标识图中各列的方法看上去非常奇怪，但当你仔细观察时会发现，沿着图的顶部穿过各项，变量A和B的形式是按下面的规则发生变化的：

1. 一次仅改变一个变量；

2. 所有可能的组合都表示出来了；

3. 第1列和第4列对应的变量是相邻的。

图3-4中所列的变量顺序其实不是唯一可行的，那只是我最习惯使用的，而且根据我的经验这样写能够画出正确的卡诺图。一般来说，很容易列出正确的变量集合，但要使它们的顺序无误倒不是很容易，错误的顺序必然导致错误的结果。俗话说得好："种瓜得瓜，种豆得豆"。

应该注意的另一点是，对于变量数为4或更少的问题借助K图能够生成出最简化的方程形式。而对于含5个或更多变量的图，要得到最好的结果需使用三维K图，其中包含多个4变量K图所代表的平面。为了讨论方便，当后面使用5变量的K图并且它需要进一步简化时，我们会指出来。也就是说，基于平面的K图再做一些布尔代数推导比画出三维的K图要容易实现。

我们可以将用K图简化真值表所对应逻辑函数的过程总结为：

1. K图中单元的数目等于输入变量状态的所有可能的组合数。例如：

   a. 对3个变量A、B、C，有8个单元

   b. 对4个变量A、B、C、D，有16个单元

   c. 对5个变量A、B、C、D、E，有32个单元

56 因此单元的数目 $=2^{\text{输入变量数目}}$。

2. 构造K图，使得从左到右的各列或从上到下的各行所对应的变量标识每次仅有一个变量变化。看看图3-5，注意第一列和最后一列以及第一行和最后一行都被认为是相邻的，就像这个图真是被卷成一个圆筒形状。此外还要注意对角线上的第一个单元和最后一个单元不被看作是相邻的。

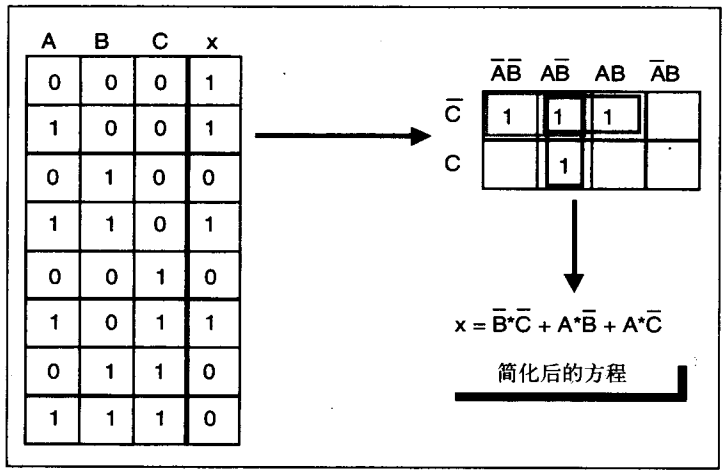


图3-5 将真值表翻译为K图。每个K图的单元代表一个独立变量的组合。对应于真值表中输出为1的行，相应的K图单元也填上"1"

3. 为每一个输出变量构造一个单独的K图。

4. 逐个检查真值表中的行，若该行输出为"1"则对应地在K图上的那个单元上填"1"。

5. 将K图中含1且相邻的单元圈出来，并且让这个圈中的1尽可能多，这样圈出的相邻单元

中单元的数目可能是2、4、8、16、32，等等。

6. 你可能需要画多个圈，而一个单元可以同时在多个圈中，但每个圈必须包含至少一个不在其他圈中的单元。仔细检查图中是否有某个圈中所有单元都属于另一个圈，如果找到就把被包含的圈删除。

7. 最后，通过删除圈中同时包含自身以及补的变量来简化圈对应的表达式，然后将各个圈对应的最小项形式“或”起来就得到了最终简化的方程。

可能用一个例子来解释会更清楚，图3-5显示了这个过程。我们有一个含3个独立输入变量A、B、C和一个输出变量x的真值表，3个输入变量表明真值表中应该有8行。在图3-5中，有4行的“x”列处为1，因此我们可以为x写下没有简化的逻辑方程：

$$x = \overline{A}*\overline{B}*\overline{C} + A*\overline{B}*\overline{C} + A*B*\overline{C} + A*\overline{B}*C$$

- 现在看图3-5，真值表对应的K图显示于右边。与真值表保持一致，对应于变量“x”的K图中有4个单元为“1”。我们可以画出下面的3个圈：
- 浅灰色的圈包括 $\overline{A}*\overline{B}*\overline{C}$ 和 $A*\overline{B}*\overline{C}$ 两项
- 灰色的圈包括 $A*\overline{B}*\overline{C}$ 和 $A*B*\overline{C}$ 两项
- 深灰色的圈包括 $A*\overline{B}*\overline{C}$ 和 $A*\overline{B}*C$ 两项

然后，我们可以从浅灰色圈中删除变量A，从灰色圈中删除B，从深灰色圈中删除C。由此得到的方程为：$x = \overline{B}*\overline{C} + A*\overline{B} + A*\overline{C}$，这就是原始方程的简化版本。 [57]

请注意在这个例子中，代表输入变量状态 $A*\overline{B}*\overline{C}$ 的单元是公共单元，它在三个圈中都出现了。然而，每个圈中还有一个单元不被其他的圈所包含，所以我们画出的这3个圈符合要求。

在我们继续介绍新内容之前，应该看看使用乔治·布尔提出的这些代数法则是否也能推导出相同的简化方程式。下面是用布尔代数简化这个方程的步骤：

步骤1	$x = \overline{A}*\overline{B}*\overline{C} + A*\overline{B}*\overline{C} + A*B*\overline{C} + A*\overline{B}*C$	来自真值表
步骤2	$x = \overline{A}*\overline{B}*\overline{C} + A*B*\overline{C} + A*\overline{B}*(C+\overline{C})$	第一分配律
步骤3	$x = \overline{A}*\overline{B}*\overline{C} + A*B*\overline{C} + A*\overline{B}$	第一互补律
步骤4	$x = \overline{A}*\overline{B}*\overline{C} + A*(B*\overline{C}+\overline{B})$	第一分配律
步骤5	$x = \overline{A}*\overline{B}*\overline{C} + A*[(\overline{B}+\overline{C})*(B+\overline{B})]$	第二分配律
步骤6	$x = \overline{A}*\overline{B}*\overline{C} + A*(\overline{B}+\overline{C})$	互补律
步骤7	$x = \overline{A}*\overline{B}*\overline{C} + A*\overline{B} + A*\overline{C}$	第一分配律
步骤8	$x = \overline{B}*(\overline{A}*\overline{C}+A) + A*\overline{C}$	第一分配律
步骤9	$x = \overline{B}*[(\overline{A}+A)*(\overline{C}+A)] + A*\overline{C}$	第二分配律
步骤10	$x = \overline{B}*(\overline{C}+A) + A*\overline{C}$	第一分配律
步骤11	$x = \overline{B}*\overline{C} + \overline{B}*A + A*\overline{C}$	第一分配律

让我们再看看一个稍微复杂一点的例子。图3-6显示了一个含两个输出变量的4-输入变量问题，同样这也是一个人造的例子，就我所知它不代表任何一个实际系统。如果我们真的要建造一个数字系统，那么系统需求说明会给出每个输出对于给定输入变量集的状态。

如果看了为变量“X”画的K图，你会发现可构造出两个简化的圈。其中灰色的圈被K图的两个竖边折断，因为这两个边被看成是相邻的。类似地，深灰色的圈被K图的上下两个水平边折断。你可能想知道为什么我们不为 $A*B*\overline{C}*\overline{D}$ 和 $\overline{A}*B*\overline{C}*\overline{D}$ 两项也画一个圈。原因是这两项都已经在其他的圈中，所以我们不能再画一个圈。要再画出第三个圈，除非有一个或更多的项不被任何圈所包含。 [58]

借助K图得到的结果并不总是最简化的，但它会非常接近最简化的方程。对于多于4个变量的K图，尤其要注意这一点，事实上，5个变量的K图应该表示为两个4变量K图，其中一个在另一个的上面。因此在使用K图方法对逻辑方程进行简化后，别忘了可能还需要用布尔代数以及德·摩根定理将其进一步简化，以得到最简化的形式。

作为最后一步，让我们把简化后的逻辑方程转换为一个实际的门电路实现。我们将使用图3-5中的那个简化后的逻辑方程，并将其转换为等效的门电路。图3-7显示了用非门、与门和或门实现的这个设计例子。这并不是唯一的设计方案，它只是一个方便的展示硬件设计的例子。3个输入信号并排显示于图中的左上角。对每个输入我们都加了一个非门，这样我们在电路设计中使用这个变量本身以及它的补信号都很方便。

同时注意图中我们用一个实心黑点指明在那个地方两根线物理上连接在了一起。我们这样做是为了区分出两根连在一起的线和两根相互交叉但并未连在一起的线，这个黑点标记起到了这个作用。

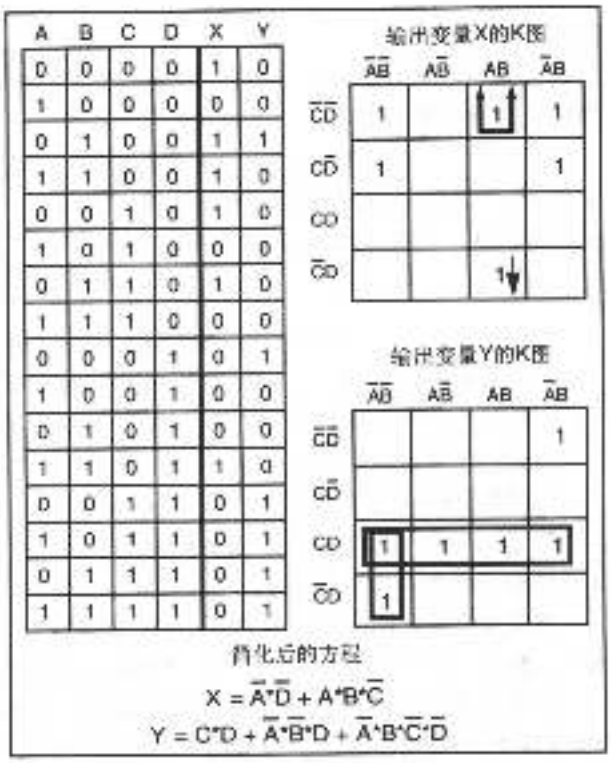


图3-6 对一个含有两个输出变量的4-变量真值表进行简化

图3-7中的电路也显示了输入变量、它们的补、用与门实现的组合项，以及用或门实现的这些组合项的并。注意我们需要三个2-输入与门和一个3-输入或门来实现这个设计。如果遇到一个更复杂的问题，我们可能需要选用带更多输入变量的与门和或门，或者每个门的输入变量较少但用好几级门排列起来。比如说，我们可以用三个3-输入与门实现出一个等效的7-输入与门。

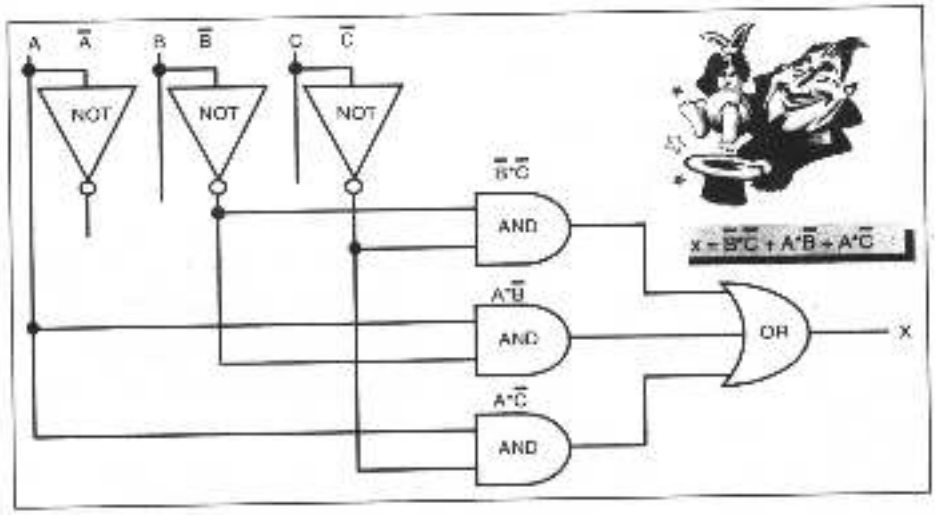


图3-7 逻辑方程 $X = \overline{B}\cdot\overline{C} + A\cdot B + A\cdot\overline{C}$ 的电路实现

图3-7中的电路真的符合我们用真值表表示的原始设计吗？快速地检查一下可能是个好主意，这样在面对更复杂的问题时我们会对已使用的方法有信心。先对真值表的前3项进行检查，剩下的5项检查留给你作为练习。

1. 第1项：A = 0, B = 0, C = 0。非门将输入变量B和C求反，使得到达第一个与门的B信号和C信号均为1。既然两个输入都是“1”，这个与门的输出也为“1”。这个与门的输出是或门的输入，而或门有一个输入为“1”，所以其输出就也是“1”，即x = 1，这正是真值表所需要的结果。

2. 第2项：A = 1, B = 0, C = 0。根据真值表，在这个情况下“x”也应为“1”。由于第一个与门不需要变量A作为输入，而且变量B和C并没有变化，所以与上面的情况1一样，我们也得到x = 1。

3. 第3项：A = 0, B = 1, C = 0，由于B = 1的补是 $\overline{B} = 0$，所以第一个与门现在得出的结果为“0”，即0 AND 1 = 0。第二个与门以A和 $\overline{B}$ 作为它的输入，由于此时A = 0且 $\overline{B} = 0$，所以它的输出也是“0”。第三个与门的输入是A = 0和 $\overline{C} = 1$，它的输出结果还是“0”。因为或门的三个输入都是“0”，所以最后x = 0。

在我们结束有关逻辑门的讨论并开始考虑依赖于同步或时钟信号的系统之前，让我们看看还有什么其他的数字系统需要我们选选。真值表的形式非常有趣，因为它看上去非常像一个存储器的组织形式。图3-8就是图3-3中的真值表，但我们将它显示为一个存储器的样子。这里有4个输入变量和两个输出变量。

A	B	C	D	E	F
0	0	0	0	0	1
1	0	0	0	0	0
0	1	0	0	0	0
1	1	0	0	0	0
0	0	1	0	1	0
1	0	1	0	1	0
0	1	1	0	0	1
1	1	1	0	0	0
0	0	0	1	0	0
1	0	0	1	0	0
0	1	0	1	0	0
1	1	0	1	0	0
0	0	1	1	0	0
1	0	1	1	0	0
0	1	1	1	0	0
1	1	1	1	1	0

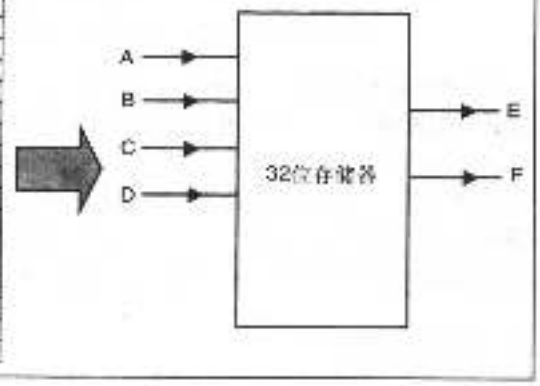


图3-8 将一个真值表转换为一个存储器映像。独立变量A到D提供了存储器的地址，两个输出变量E和F的值则用相应地址（真值表的行）存储单元中的数据表示

让我们看看这样做的含义。我们可以想像正在使用一个真正的存储器，并将数据填充进去。这样，当我们给它合适的地址位值（这个例子中是输入变量A到D的组合）时，从存储器得到的输出数据（变量E和F）就正好符合我们用真值表定义的电路功能。因此我们有两种方式实现逻辑系统。或者用逻辑门进行电子电路设计，或者用存储器设备直接实现真值表的内容。若用存储器实现逻辑功能，我们就不用像用门电路进行设计那样再做逻辑简化。当然，我们可能得不到我们用其他方法时所需的速度。

为更清楚地说明这点，我们重新把图3-8画成图3-9。它们唯一的区别是我们将把它表示成了一个真正的存储器。独立变量（图3-3中的A到D）被表示为存储器的地址位A0—A3。输出变量（图3-3的E和F）现在是存在于各个存储器单元中的数据位。

因此，我们的存储器总共需要有32位的容量，两个输出变量中的每个需要16位。每个二

进制位表示组成存储器单元地址的输入变量组合所对应的输出变量值。

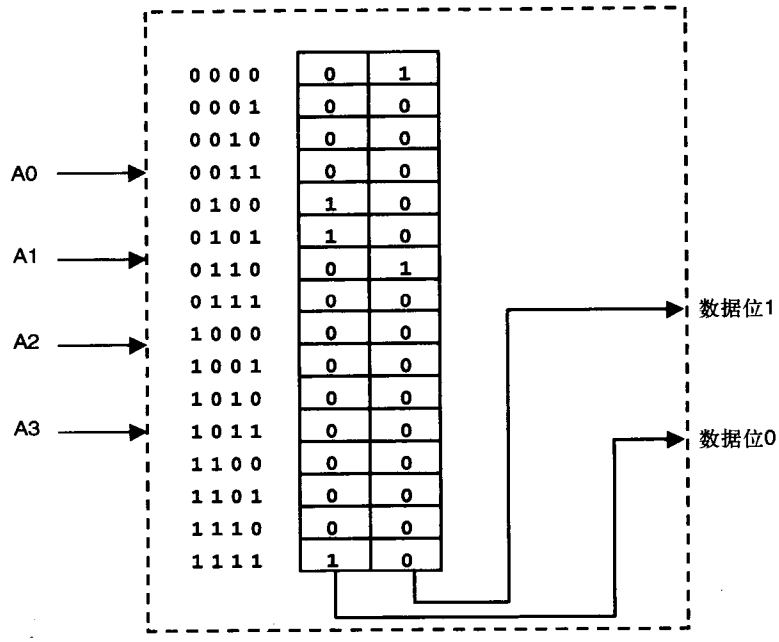


图3-9 将一个真值表转化为存储映像。独立变量A到D提供存储器的地址，两个输出变量E和F用对应存储器地址（真值表的行）上存储单元中的数据表示

图3-8和图3-9表示了用硬件实现逻辑设计的另一种方式。在前一个例子中，我们用布尔代数定律和K图建立了一组简化的逻辑方程，然后我们可以将它实现为逻辑门的组合（也称为*组合逻辑*，combinatorial logic）。图3-8表明我们可以简单地根据真值表将其中所有信息填入一个存储器芯片。事实证明这两种方法都同样有效，可被用于最适合它们的场合。将存储器用作逻辑元件，称为*微编码*（microcode），是现代数字计算机中很多控制电路的基础。当我们在后面的章节中介绍*状态机*（state machine）时还会再讨论这个概念。

61

3.4 时钟和脉冲

在第2章中，我们首次看到数字信号被表示为波形。也就是说，逻辑信号被表示为随时间变化的一系列值，而波形则是这种信号显示在条带记录纸上的图形。我们需要考虑的最简单的数字信号是图3-10所示的简单正脉冲，图中显示的是一个脉冲高度约为3V的单个正脉冲。

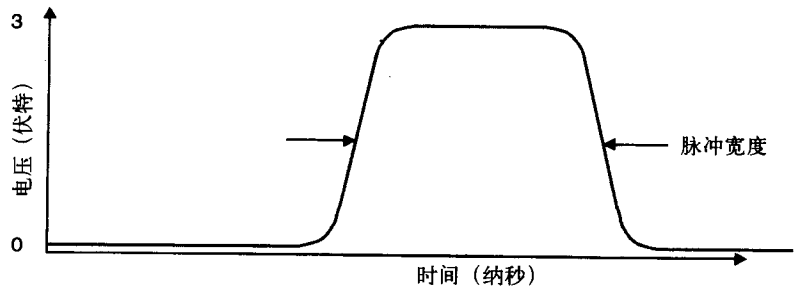


图3-10 一个大约3V高的正脉冲的典型波形

到这里我们要停一下，然后问："什么是脉冲？"你可能知道的一个医学术语是你的"脉搏"（它和脉冲的英文同为"pluse"），它表示每次你心跳时你能感觉到的血管压力的变化。由于心脏对血液的挤压是不连续的，会造成离散的血管跳动，所以你感觉到的是血液流动时产生的压力脉动。当你进行心脏检查时，从得到的心电图上也能看到这种东西，在心脏电信号的特征曲线上你会看到一个个尖峰（图3-11）。

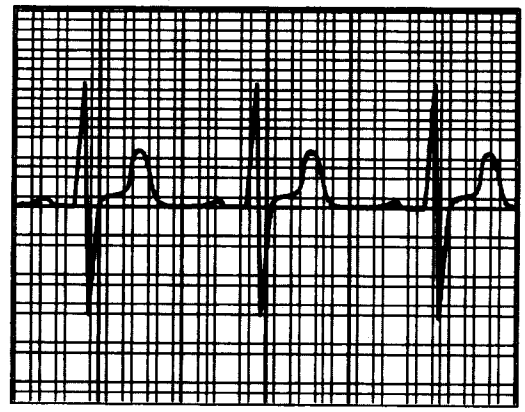


图3-11 心电图的一部分，它显示了心脏处电信号的特征脉冲

任何一个脉冲的特点都是系统从一个松弛的状态（比如低压）变到一个激动的状态（血液澎湃），然后再回到松弛状态。电学上，我们可以将脉冲描述为从低电压到高电压，然后再回到低电压（或者完全相反的过程）的一个电信号。换句话说，一个脉冲可以是这样的一个系统，它从有效状态进入到无效状态，再回到有效状态。我们称从低电压到高电压再回到低电压的脉冲为*正脉冲*（positive pulse），而从高电压到低电压再回到高电压的脉冲为*负脉冲*（negative pulse）。

图3-10中的脉冲是正脉冲，因为它从"低"状态进入"高"状态，结束的时候又回到了"低"状态。在这个例子中，*脉冲宽度*（pulse width）是对脉冲存在时间的度量，既然对这个脉冲没有给出时间刻度（x坐标轴），我们假设脉冲宽度约为50ns（经常被简写为50ns），即50个十亿分之一秒。脉冲宽度是在脉冲的低压和高压之间的中点处的测量值。因此，对于3V高度的脉冲，我们在1.5V处测量波形的脉冲宽度。 62

图3-12或多或少显示了一个"真实的脉冲"。所谓真实的脉冲，是指如果你拿着一个非常快的秒表和一个反应极快的伏特计，并且能够像发疯了那样快速涂写（注意这是一个理想实验），你所能看到的。在真实生活中，人们用一种称为*示波器*（oscilloscope）的分析仪器观看这种波形。我们在本章后面会看到一些真实的示波器波形。注意图3-12中代表时变脉冲电压的灰线在上升和下降时都存在一个坡度，这是由于脉冲状态不能以无穷快的速度进行改变，电压升到1和降为0都需要经历一些时间，我们称这些时间分别为*上升时间*（rise time）和*下降时间*（fall time）。从技术上讲，由于我们不需要考虑的原因，上升和下降时间测量的都是10%电压点和90%电压点之间的时间。

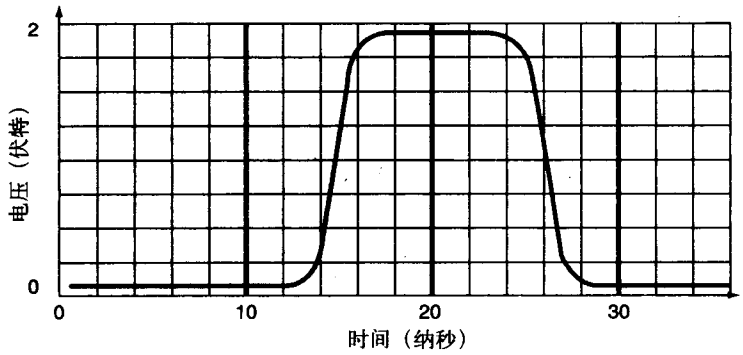


图3-12 示波器上显示的正脉冲的样子

图3-12是脉冲波形在示波器屏幕上看到的大致样子，水平轴代表以某种合适的单位（在

这个例子中是ns）表示的时间，垂直轴显示电压信号如何随时间改变。

图3-13显示了用一个研发实验室中真实示波器进行上升时间测量的图像。示波器能自动测量一些量，比如上升时间、下降时间、脉冲宽度、脉冲高度、频率和周期等。示波器中的电路会自动分析脉冲波形的形状，定位10%电压点和90%电压点，然后计算出这两点之间的时间差。

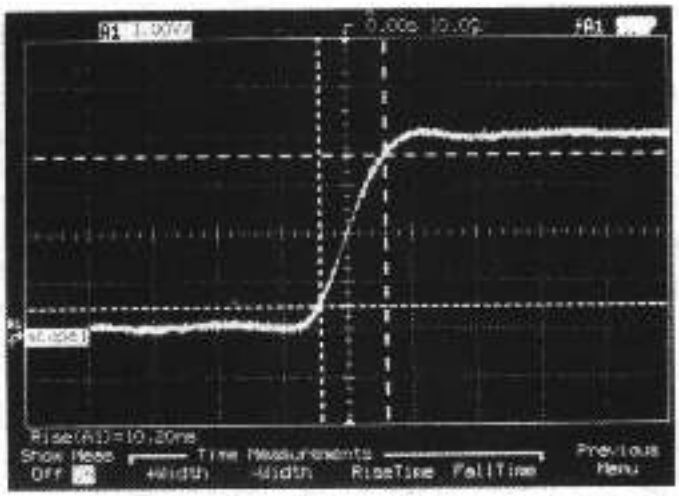


图3-13 用示波器测量上升时间，垂直轴上每格是1V，水平轴上每格是10纳秒

在我们进一步开始考虑时钟之前，我们应该讨论关于门的最后一点。我们以前定义了门的
63 传输延迟为从输入状态改变到相应的输出改变的时间延迟。让我们看看在示波器上一个真实的传输延迟看上去会是什么样子。图3-14显示了对一个非门进行传输延迟测量时看到的景象。如图所示，我们将示波器的探针连到非门的输入端和输出端。为了进行测量，我们在门的输入信号（这里是下降边）到达前一段时间就开始用示波器进行跟踪。示波器能同时显示输入和输出波形的逻辑状态，所以我们能看到输入信号变低，稍过一会儿输出信号变高。事实上，如果凑近示波器显示屏幕，你会看到输入信号下降沿和输出信号上升沿之间的时间间隔是12.60ns。

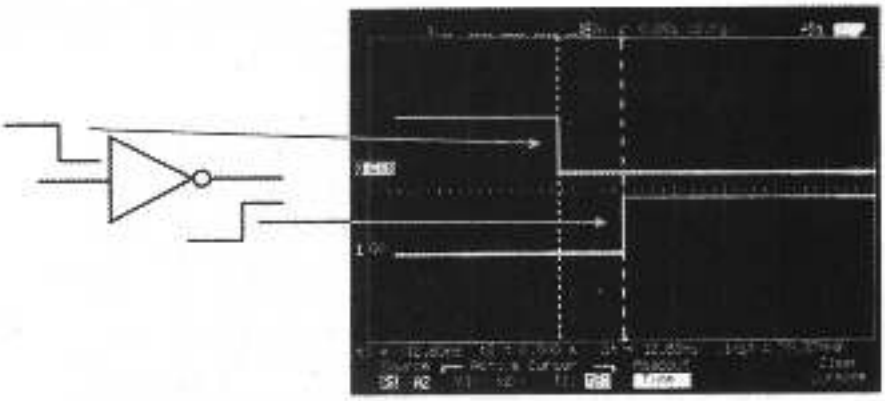


图3-14 用示波器进行传输延迟的测量。上面的轨迹线表示非门输入端的测量结果，下面的轨迹线则代表输出端，从示波器显示屏幕上可以看出，输入信号变低12.60纳秒后输出信号变高

到目前为止，我们都把脉冲看成单个的事件。但是，就像心电图中显示的那样，心脏是

连续跳动的，所以心电图曲线上是一系列的脉冲波形。

在图3-15中，我们又回到理想的波形图，注意这个波形中的上升沿和下降沿都没有任何的倾斜。我们假定这些脉冲以无限快的速度从低电压切换到高电压，同样以无限快的速度由高电压变为低电压。这样的假设不妨碍正确的理解，而且让我们更容易分析清楚这个图表。

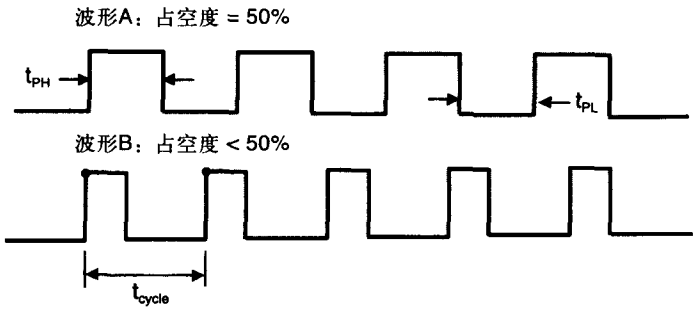


图3-15 时钟波形的例子。当波形中高电压部分的宽度等于低电压部分的宽度时（波形A），我们有50%的占空度。波形B的占空度低于50%。波形B中两个黑点之间的时间间隔为一个周期

我们称图3-15所示的这样一串连续的脉冲为时钟（clock），这个时钟不是指计算机上显示一天中时间的那个时钟，这里我们讨论的时钟是经过严格调整的一串连续的脉冲信号，它通常由晶体振荡器控制。晶体有一种特性，即能够产生出一种调制电路并以一个可预见且稳定的频率共振。例如，哪怕是最便宜的电子手表都能准到一个月误差不超过一分钟，因为它内 64
部使用了大约以32KHz频率震荡的一种晶体。我们用术语赫兹（Hertz）来表示每秒发生的周期数，或一个时钟信号的震荡频率，它对应的符号是Hz。频率的单位赫兹的命名是为了纪念德国科学家Heinrich Rudolf Hertz（1857－1894）。

时钟信号和波形的细节很容易让人糊涂，还是让我们就此打住，想想这样一个问题“时钟到底是什么呢？”

时钟是一串固定的脉冲，它们通常有相当准确不变的间隔。哪怕是一个不贵的电子手表走一个月的误差也就只有几秒或几十秒。考虑到它的脉冲在每秒钟要震荡32 000多次，脉冲间隔的准确性就很让人印象深刻了。为了不得罪电子工程师，我们不会花时间来讨论这么准确的时间信号是怎样得到的，但我们将尽力理解时钟在我们的系统中起何种作用。

假设你正在观察一个老旧的钟的钟摆，而且你可以在钟摆在一端停住时按下秒表，而当它运动到另一端再回来时停止秒表，你就可以量出这个钟摆的周期。这种老式的钟就是根据钟摆的周期变化非常小的事实，再通过齿轮的机械装置来驱动时钟的。这里重要的一点是，实际生活中的钟摆提供了一个准确的同步机制，使得我们能用它来驱动时钟的内部计时装置。钟摆的运动也为整个时钟提供了同步的来源，每次它来回摆动，它都带动齿轮从而报告时间。

假设需要5秒钟让钟摆从一端摆到另一端再摆回来，每5秒是一个循环，那么周期就是5秒，而一秒钟内循环的次数就是周期的倒数，即0.2Hz。

在我们的数字系统中，我们用时钟信号来提供相同的同步机制。在大多数计算机系统里，一个单独的时钟信号分布于整个电路中，它为所有需要同步的内部操作提供了一个准确的时间源。当你去当地的计算机商店去买最新出品的个人电脑时，销售人员会尽力说服你购买带有3.2吉赫兹时钟的电脑，以使你获得最佳的游戏体验。真正卖给你的是什么呢？其实就是一个有更快时钟的高级电脑，其意义在于一秒钟内可以发生更多的事情，所以它运行更快。

如果你买过电脑，就可能已经对术语“赫兹”或其简写“Hz”很熟悉了。销售人员会告诉你这台电脑或那台电脑是3.0“吉赫兹”的机器，而它显然比你那台500“兆赫兹”的老电脑好得多。你现在就理解了，你的旧电脑每秒运行500个百万周期，即其时钟频率是500 MHz。这个花上1000美元就能搬进你车里的电脑每秒运行30亿个周期，即其时钟频率为3 GHz。由于10亿是1000个百万，这也就等于说新电脑的时钟周期为3000 MHz，或者说大约是旧电脑时钟速度的6倍。

我们打算用图3-16展示什么呢？让我们想像一下我们坐在一个典型的计算机或微处理器
65 的时钟信号上。你有一个足够快的Radio Shack牌伏特计，并且正在测量时钟输入的电压（逻辑电平）。时钟电压开始为低（逻辑电平0），过一会儿后变高（逻辑电平1），过一会儿再变低，一直这样循环下去。每个低电压保持的时间和以前的完全一样，每个高电压保持的时间也和其他的一样。从低到高、从高到低的切换时间很快，也不随时间变化。因此，一个完整循环（由低到高再回到低）的时间周期是完全可重复的。

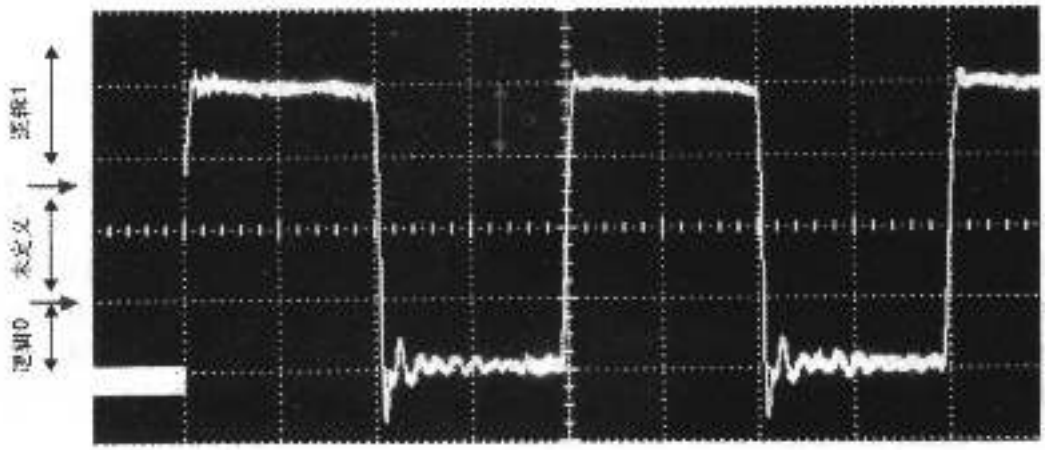


图3-16 一个2.5MHz时钟的震荡图像。垂直方向每格代表1伏，水平方向每格代表100纳秒。注意，只要信号电压超过逻辑1的阈值电压或者低于逻辑0的阈值，就能发挥正确的功能

图3-16是一个2.5 MHz时钟信号的实际震荡波形。注意图中的时钟波形不如我们理想的波形那么“整洁”，当然也差得不远。在这个图中可以清楚地看到上升沿和下降沿，通过看这个图我们可以上一堂有价值的课。虽然这个时钟波形不如理想波形那么漂亮，但它能正确地在电路中工作，原因很简单，因为数字世界中只有1和0，其他的都不重要。只要信号的电压低于信号0的阈值或高于信号1的阈值，它就能被正确地理解。

让我们给出一些术语的定义：

- 频率：单位时间（通常为1秒）内时钟脉冲的数目。频率的度量单位是赫兹，或写作Hz。1 Hz等同于每秒一个时钟周期。
- 周期：频率的倒数，它是相邻脉冲波形上两相同位置点之间的时间间隔。图3-15中波形B上两个黑点之间的时间t_{cycle}代表波形的周期，或者一个波循环的时间。时钟频率等于周期的倒数，一个周期为1秒的时钟波形其频率为1Hz。
- 占空度：时钟信号电压为高的时间占整个周期的比例。50%的占空度意味着时钟信号在
66 正好半个周期的时间内为高，或者时钟信号为高的时间等于时钟信号为低的时间。我们也可以用公式来计算：占空度$=(t_{PH}/(t_{PH}+t_{PL}))\times 100\%$。

图3-17显示了常用（计算机这个领域）的时间单位之间的关系，以及常用的频率单位之间的关系。

从图3-17也可以看出，频率是1 MHz的时钟其周期为1μs（微秒），而频率是1GHz的时钟其周期为1ns（纳秒）。因此，你的装有1 GHz Athlon品牌CPU的计算机有一个震荡极快的时钟，在它的一个周期时间内光也只能走大约1英尺的距离。

让我们复习一些有用的关系：

- 1ns的时钟周期，其频率为1GHz。
- 1MHz的时钟频率，其周期为1μs（微秒）。
- 1ms（毫秒）的时钟周期，其频率为1KHz。
- 1秒的时钟周期，其频率为1Hz。

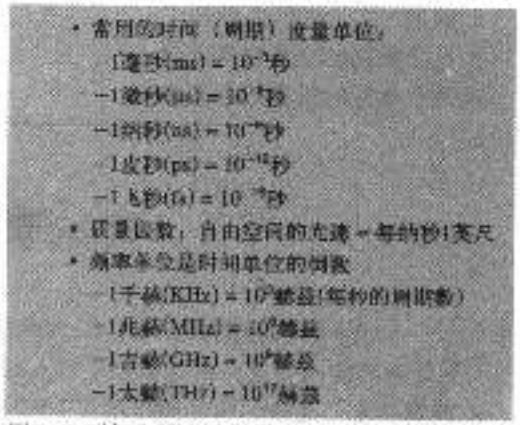


图3-17 时间和频率的常用单位。空气中光的速度非常接近于每纳秒1英尺，而光在集成电路芯片中一条通路中的速度约为空气中的一半，大约是每纳秒6英寸

总结

- 布尔代数提供了处理和简化逻辑方程的规则。
- 根据所有可能的输入变量状态以及相应的输出变量状态，真值表提供了一种描述任意数字系统的有效办法。
- 卡诺图是对真值表得到的最小项形式逻辑表达式进行简化的图形化方法。
- 数字系统由时钟信号驱动，时钟信号是一串连续的脉冲，这些脉冲可以由它们的宽度、高度、上升时间和下降时间来描述。
- 频率和周期之间是互为倒数的关系。
- 我们用工程数字单位系统来描述在数字系统设计中常用到的频率和时间量。

参考文献

[1] Smith, Robin, "Aristotle's Logic," *The Stanford Encyclopedia of Philosophy (Fall 2003 Edition)*, Edward N. Zalta (ed.), http://plato.stanford.edu/archives/fall2003/entries/aristotle-logic/.

[2] J. J. O'Connor and E. F. Robertson, http://www-gap.dcs.st-and.ac.uk/.

[3] M. Karnaugh, *The Map Method for Synthesis of Combinational Logic Circuits*, taken from, *Computer Design Development Principle Papers*, Edited by Earl E. Swartzlander, Jr., ISBN 0-8104-5988-4, Hayden Book Company, Rochelle Park, NJ, 1976, p. 25.

[4] Gerald Williams, *Digital Technology, Second Edition*, Science Research Associates, Inc. ISBN 0-5742-1555-7, 1982.

67

习题

1. 设计一个1位全加法电路，其中全加法器能将两个输入二进制位和一个进位位相加，输出它们的和及进位值。参考下图，创建真值表、卡诺图、简化的布尔方程以及一个门级的电路图。



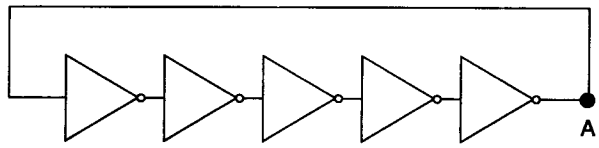
2. 使用真值表证明德·摩根定理中的两种情况。
3. 下图中显示的电路称为环形震荡器（Ring Oscillator），它包括5个非门首尾相连。想像你正在测量A点处的电压电平。经过每个门的传输延迟都正好是10ns。传输延迟的定义是门的输入变化引起门输出变化的间隔时

间。假设时间t=0ns时A点的电压从0变到1。

a. 画出A点处波形的草图。

b. 这个电路的震荡周期是多少?

c. 这个电路的震荡频率是多少?

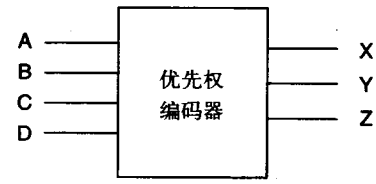


4. 设计一个有4个地址输入A、B、C、D和1个输出x的真值表。假设A和B是控制端输入，C和D是任意的输入变量。在真值表中填入x的值，使得电路根据控制输入A和B的状态实现不同的逻辑功能。这些逻辑功能显示在下表中：

A	B	关于C和D的输出逻辑函数（输出x）
0	0	与非（NAND）
0	1	异或（XOR）
1	0	或非（NOR）
1	1	与（AND）

68

5. *优先权编码器*（Priority Encoder）是一个电路，它的输出是开启的最高有效输入位所对应的二进制码。假设有如下图所示的一个电路图，其中A是输入的最低有效位，而D是输入的最高有效位。X是输出的最低有效位（2^0），而Z是输出的最高有效位（2^2）。如果所有的输入都为0，则所有的输出都为0。输出的优先级别由值为1的输入的最高有效位决定。为这个电路创建真值表，然后使用卡诺图简化真值表并画出简化的电路。



6. 假设一个逻辑电路，其输入为两个4位二进制数（A0到A3，以及B0到B3），而输出为1位二进制数Z。当两个输入的数相等时输出Z为真（高电压）。设计这个电路实现等价性检测功能。

7. 假设你是“天堂之路”（Road to Nirvana）泡澡和温泉公司的首席设计师，给你的任务是设计一个新的温泉控制器来代替那个根据1972年的洗衣机而改造的老控制器。下面是具体的输入要求：

	变量值=0	变量值=1
温度指示器：A	水温低于期望的泡澡温度	水温等于或高于理想的泡澡温度
日常过滤计时开关：B	流通泵关闭	流通泵打开
吹风机开关：D	吹风机关闭	吹风机打开
钥匙开关：E	系统关闭	系统启动
手动泵开关：F	流通泵关闭	流通泵打开

你的逻辑电路要控制下面的输出信号：

*f*代表泵的马达：值为1代表开。

*g*代表吹风机：值为1代表开。

*h*代表加热器：值为1代表开。

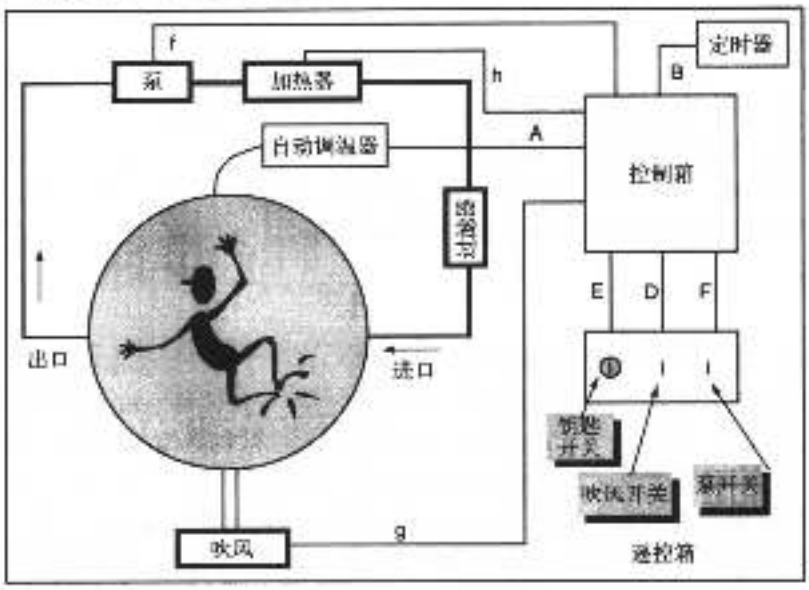
69

参考下图，具体的工作流程是：

a. 为这个泡澡控制器创建真值表，注意可能有多种不同的方式定义澡盆的操作。

b. 用卡诺图、德·摩根定理和布尔代数关系式来简化你的逻辑方程。

c. 当你尽可能地简化了这些方程后，用门电路画出这个逻辑设计。



8. 下面是一个称为3-输入8-输出译码器的真值表。基于输入变量的二进制值（从0到7），它每次只将一个输出置为低有效，根据真值表也可以看出这一点。使用你学过的布尔代数推导方法和卡诺图技术，画出能实现这个功能的最简单的门电路图。提示：注意“输出的激活状态”是低，而不是高，你可以利用这点大大简化这个问题的逻辑表达式。注意称输出为 $\overline{CS0}$ 到 $\overline{CS7}$ 是有原因的，我们将在学习存储器系统组织时介绍它。

A0	A1	A2	$\overline{CS0}$	$\overline{CS1}$	$\overline{CS2}$	$\overline{CS3}$	$\overline{CS4}$	$\overline{CS5}$	$\overline{CS6}$	$\overline{CS7}$
0	0	0	0	1	1	1	1	1	1	1
1	0	0	1	0	1	1	1	1	1	1
0	1	0	1	1	0	1	1	1	1	1
1	1	0	1	1	1	0	1	1	1	1
0	0	1	1	1	1	1	0	1	1	1
1	0	1	1	1	1	1	1	0	1	1
0	1	1	1	1	1	1	1	1	0	1
1	1	1	1	1	1	1	1	1	1	0

9. 假设按如下命令定义了4个变量A、B、C和D：

```
bool    A, B, C, D;
```

控制C的语句为：

```
If (D)
   C = B;
else
   C = A;
```

请画出这个控制电路的等效门电路图。

70

第 4 章　同步逻辑简介

学习目标

- 学习如何连接逻辑门得到触发器电路；
- 学习不同类别的触发器和它们的行为；
- 学习使用D型触发器的不同电路结构，包括分频器、计数器、移位寄存器和存储寄存器；
- 学习如何用D型触发器来同步状态机的状态转换。

4.1 引言

到目前为止，我们对数字逻辑系统的学习还仅限于异步逻辑，异步逻辑是指“不同步的”逻辑。在我们讨论的系统中，这意味着输出变量状态的改变仅依赖于输入变量的状态以及连接输入和输出之间的组合逻辑。改变一个输入变量则输出变量也会改变以保证逻辑的正确性，这个过程中除了通过组合逻辑门的传输延迟外就没有其他的延迟了。然而，在包含数百万逻辑门的一台计算机中，我们必须能够通过某种主控信号（即时钟）来同步逻辑状态的改变，从而使微处理器所做的事情能按定义好的状态序列进行下去。因此，现在让我们把注意力投向同步逻辑。

看图4-1所示的电路结构。先看上面的电路，注意这些反向门的输出是怎么和另一个门的输入相连的。这种结构我们称之为反馈（feedback）。当麦克风被放在正发出大音量声音的喇叭附近时，你听到的尖锐杂音就是反馈的一个例子。让我们分析图4-1中的电路，它有两个输入A、B和两个输出Q、$\overline{Q}$。你马上会看到，电路的输出总是成对互补出现的，但现在，我们只是称它们为Q和$\overline{Q}$。

根据图4-1，输入A、B和输出Q都为逻辑电平“1”，而输出$\overline{Q}$为逻辑0。这表明上面那个与非门的两个输入为1和0，而下面那个与非门的输入都为1。从与非门的真值表可以看出，由于两个门的输入和输出都处于正确的逻辑状态，所以这个电路是稳定的。现在，我们通过在输入端A加一个负脉冲来给这个系统一个扰动，电路的状态将迅速地发生改变。由于输入是1和0，下面那个与非门的输出变为1。现在这个1又加在上面那个与非门的输入端，因为输入都为1，所以它的输出变为0。上面与非门的输出同时也是下面那个与非门的输入，这样使得后者的两个输入均为0，其输出自然为1。最后，我们撤销脉冲让信号恢复到原来的样子，这样做只是让输入端A的信号回到之前的状态。值得注意的是，即使A变为1，两个与非门的输出也仍然会保持它们新的状态，其

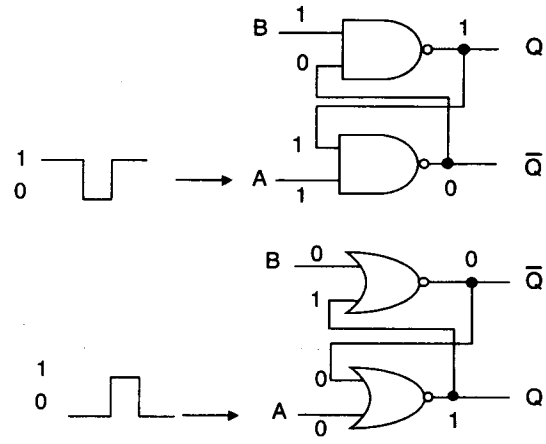


图4-1 RS触发器。上面用与非门设计的电路由负向脉冲触发，下面用或非门设计的电路由正向脉冲触发

原因是来自上面那个门的正反馈信号迫使电路保持在Q = 0和 $\overline{Q}$ = 1状态。

显而易见，如果我们在输入B上加一个负脉冲并重复上面描述的过程，那么输出信号将翻转回它们原来的状态。对于图4-1中或非门组成的电路，你应该可以类似于前面分析与非门电路那样进行这样的分析。这时，导致状态变化的将是正脉冲，而不是负脉冲。

4.2 触发器

对于图4-1中的与非门电路，如果我们在输入端A第二次加上负脉冲，那么系统状态也不会改变，因为上面那个与非门的输出仍然是0。让这个电路回到原来状态的唯一办法是在输入端B加一个负脉冲。因此我们可以看出，一个输入端是置位（SET）输入（置Q = 1），而另一个输入是复位（RESET）输入（置Q = 0）。这类电路元件称为*触发器*（flip-flop），因为两个输出信号像游乐场里的跷跷板那样一个上，另一个必然下。这里介绍的这种触发器为*RS触发器*，因为它的两个输入交替地使输出信号置位（S）和复位（R）。我们也称这种交替置位和复位的现象为两个状态之间的*翻转*（toggling），这很像房间墙上的翻转开关控制着灯的开和关。图4-2是将RS触发器表示为一个独特的电路元件的示意图。我们把图4-1中的与非门电路和或非门电路搬过来，又画了一个不同的电路符号表示它。

虽然图4-2是一个简单的例子，但它说明了一个重要的概念。从第1章到第3章，我们考虑的电路越来越复杂，例如我们学习了如何利用电子开关元件即MOSFET晶体管构造配置一个简单的反相器，然后又怎么将这个基本的配置扩展为更复杂的门电路，比如与非门。我们也学习了如何用一个独特的符号表示复合门（异或门），以简化那些表示它的电路。现在我们正在扩展这个概念，开始用这个简单的电路构件块创造新的、更强大的电路配置。我们将在本章乃至本书剩下的部分继续这样的过程。让我们继续下去！

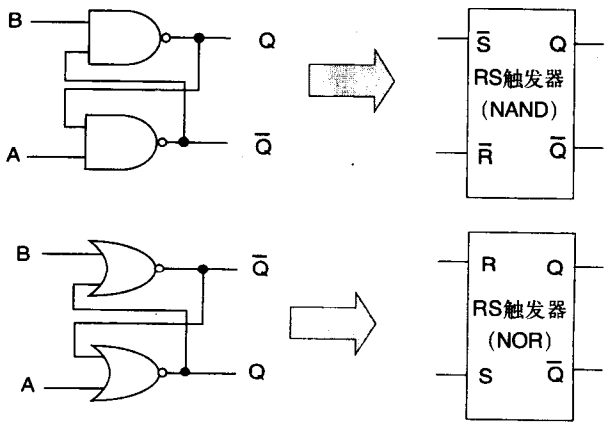


图4-2 表示RS触发器的独特的电路符号

72

RS触发器非常重要，因为它引入了*状态依赖*（state dependency）的概念。RS触发器两个输出端的状态不仅依赖于两个输入变量A和B的值，而且依赖于输出端的先前状态，这与我们所看到的异步组合逻辑的行为完全不同。现在，输出的状态成为了输入状态以及输出先前状态的函数。即使从一个“新概念”的角度来看RS触发器非常重要，它在实际中的使用也仍很有限。因此让我们花一些时间来看看如何将这个概念应用到更实际的电路构造上去。

RS触发器是一个异步器件，给S端或R端输入加上脉冲信号将相应地驱动输出Q和 $\overline{Q}$ 。当输入信号激活后，输出就会跟着响应，这里不涉及时钟信号。这样就有一个问题：“所有这些信号的同步机制是怎样的？”这是个好问题。这里确实没有信号同步，我们用与非门和或非门的门控特性来引入时钟的概念。图4-3显示了这样一个触发器的设计，我们称它为*带时钟的RS触发器*（clocked RS flip-flop）。为了简单，我们仅使用了与非门，采用合适的替换你很容易将这个电路用或非门来实现。

现在通过增加的两个与非门，我们就可以用第三个输入（即时钟信号）来控制R和S的输入。而且，现在R和S输入信号是在电压为高时激活触发器，而不是低时激活触发器，这是由于两个增加的门的反相作用。假设现在Q = 0和 $\overline{Q}$ = 1，我们希望翻转触发器使输出达到相反的状态。假设时钟输入为低电压，那么将SET输入变成高电压将不起作用，因为只有当两个输入同时为高电压时与非门的输出才会变成低电压。这样，当我们将SET输入设成高电压时，只有当时钟信号也变高时才能使触发器的输出状态发生翻转。

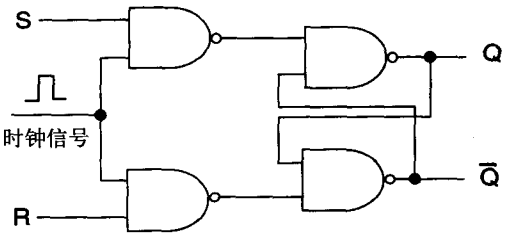


图4-3 带时钟的RS触发器

这已经接近于我们想实现的电路功能，但还不完全。我们已经引入了一个时钟同步机制，但这是一个比较弱的同步。其问题是，这样得到信号同步的程度依赖于实际时钟信号的宽度。如果时钟信号保持高电压的时间比R和S输入信号变化的时间间隔大得多，那么就得不到我们希望的那种同步效果。只要时钟信号为高电压，在R和S输入上加脉冲信号就能使输出信号不断地发生翻转。而理想的情况是，我们希望使用时钟信号的上升沿或者下降沿来同步这些触发器，而不是用时钟电位的高低来进行同步。记住，逻辑信号的上升沿和下降沿是电压从低到高和从高到低的切换过程。为了稳定性，我们要求这种信号切换发生得非常快，而与信号保持在高电压或低电压的时间长度无关。因此，即使一个信号的状态
73 每12小时才改变一次（一个数字时钟的半天变化指示器），我们仍希望输出信号的切换在几纳秒就完成。

图4-4中的电路构造所对应的可能是最著名的一种触发器了，它被称为JK触发器（JK flip-flop）（或简写为“JK FF”）。根据Null和Lobur的著作[1]，JK触发器这个名字是为了纪念Jack Kilby，这位德州仪器公司的工程师是集成电路的发明者之一。这个电路中，我们采用了基本的带时钟的RS触发器，并增加了两个重要特性：

1. 在输出Q、$\overline{Q}$ 与与非门的输入之间加了第二套反馈回路。

2. 第二套输入J和K加在与非门的输入上。

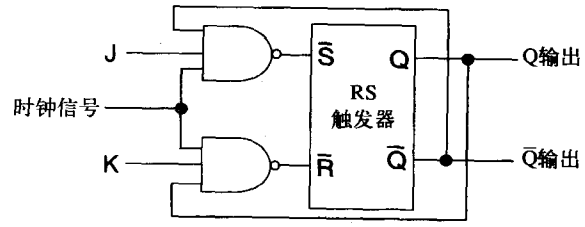


图4-4 JK触发器

从概念上讲，我们所做的就是，通过将门控与非门（gating NAND）从2输入与非门变成3输入与非门，给电路增加了额外的反馈路径。然而，通过这个过程我们离建造一个按我们希望的方式工作的JK触发器已经不远了。不幸的是，它还不完全正确。其中的问题可以追溯到前面那个带时钟的RS触发器，信号是根据时钟的电位高低而不是边沿来同步的。而我们希望JK触发器能根据时钟信号的边沿进行同步。表4-1是一个JK触发器的真值表。最后一个条件是J = 1和K = 1，它们能在时钟信号切换时使触发器发生翻转。不久你就会理解，从很多方面来看这都是一个理想的电路行为。然而，由于我们仍然受时钟持续时间的支配，当三个输入信号都为高电压时，触发器将处于一个不稳定的状态。为了理解这点，可返回到图4-1，分析当输入A和B同时变低时电路所处的状态，这种情况也称为*竞争条件*（race condition），通常会导致不稳定和不可预料的电路操作。

表4-1 JK触发器的真值表

J	K	时钟信号切换后的Q
0	0	不变
1	0	Q = 1
0	1	Q = 0
1	1	输出翻转

下面我们分析电路，看看是什么导致了这个问题。假设触发器处于复位状态（Q=0，$\overline{Q}$=1），J和K均为高电压，而时钟为低电压。由于输出 $\overline{Q}$ 被连回到上面与非门的三个输入之一，当时钟信号变为高电压时，所有三个输入都为高，其输出将变低。这个低的输出将导致触发器中的RS触发器部分发生翻转。好，到现在为止，这正是我们所希望的电路行为。

一旦输出信号Q和 $\overline{Q}$ 翻转，上面那个与非门的输出会再次变为高电压。然而，时钟信号仍是高电压，所以下面那个与非门的输出变低使RS触发器回到了原来的状态。现在这个电路可能会出现两种状态，都是难以预料的：或者输出Q和 $\overline{Q}$ 都变高，并且只要时钟信号是高它们就一直保持高电压的状态；或者由于竞争条件导致的不稳定性使得输出快速翻转。只有仔细分析电路的传输延迟和切换条件，才能准确地确定电路的行为。无论如何，这都不是我们所希望的。

我们所缺乏的是一种将控制转移到时钟边沿的机制，解决的办法是构造一个有两个串联门控RS触发器的电路。图4-5中所示的就是我们所需要的。虽然这个电路图看上去有点复杂，但实际上我们只对图4-3和图4-4中的基本电路构造做了点小的修改。我们在图4-4中构造两个反馈回路，并试图使这个电路在J = K = 1，并且在时钟信号出现正脉冲时发生翻转，别忘了这种电路结构的主要问题。 74

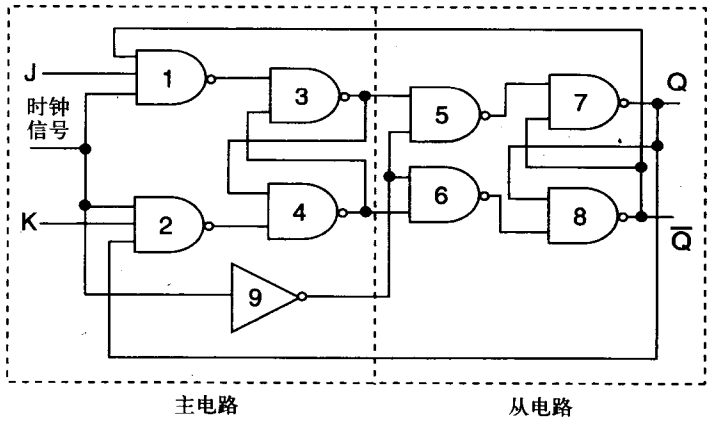


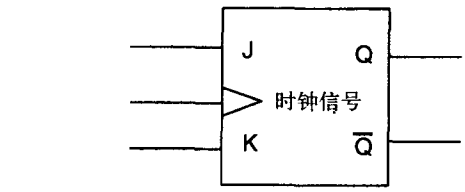
图4-5 主从JK触发器

在图4-5中，通过增加一个非门（第9号门）和第二个门控触发器，我们解决了信号竞争的问题。这个新的触发器设计称为主从（master-slave）JK触发器。在你开始仔细理解其中奥妙之前，让我告诉你它完全是好的。第一组电路元件是“主”元件，用于输入数据并让它们稳定，然后数据被传送给“从”元件，最后数据出现在Q和 $\overline{Q}$ 输出上。非门和“从”元件部分一起创造了前面我们所缺少的电路功能。当主触发器部分（门1到门4）的时钟信号变高时，非门的输出会变低从而锁住从触发器，避免了它的RS触发器输出端的任何变化。

当时钟信号再次变低时，主触发器失效，但它的RS部分不会改变状态。然而，变低的时钟使得从触发器有效，并且它的改变与门控输入保持一致。

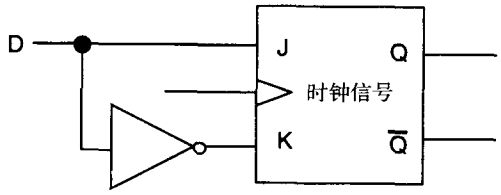
由于阻止了竞争条件的发生，以前造成所有竞争问题的两个反馈回路现在都能正确地工作了。现在触发器中的主、从两部分分别在时钟为高电压和低电压时（在交替相位上）工作。换句话说，在原来的电路中由于从输出到输入的反馈信号只要时钟为高电压就可以自由“竞争”，这导致了竞争问题。现在，我们改变了电路使得反馈回路被有效地进行了隔离，每个时刻只有一半的电路处于活动状态。这个主从式的电路配置最终实现了我们的目标，它仅在时钟信号切换时工作，而不是在时钟的某个电位下工作。

图4-6中将JK主从触发器表示为一个独特的电路模块，也显示了其对应的真值表。注意我们用一个向下箭头指示在时钟反向切换（由高变低）时输出会改变状态，这也是主电路部分向从电路部分传递信息的时候。图4-6中也显示了稍微修改了一点的电路。通过增加一个非门，
75 我们限制了JK触发器仅在J = 1而K = 0的条件下或者相反的条件下工作。这个非门屏蔽了其他两种可能组合的发生，即J = K = 0和J = K = 1。



J	K	时钟信号 ↓	Q输出
0	0		不变
1	0		1
0	1		0
1	1		输出翻转

JK主从触发器



D	时钟信号 ↓	Q输出
0		0
1		1

JK触发器被改造为“D”型触发器

图4-6 对主从JK触发器稍微修改，得到“D”型触发器

这个新的电路结构称为D触发器，是触发器家族中最重要的一员，在本书剩下的部分我们将集中精力讨论这种电路。

观察图4-6右下角的D触发器的真值表，你会注意到一个有趣的现像。当时钟信号变低时，输出Q的改变与输入D完全一致。换一种说法来描述这件事情，“在时钟信号的下降沿，输入D上的数据被存储到这个电路单元中，并且显示于输出Q上。”现在我们知道为什么称它为D触发器了，这个“D”是数据（data）的意思，这其实是一个存储单元。

在我们把全部注意力投向D触发器（缩写为D-FF或D-flop）之前，我们需要做一些介绍性的论述。前面说过，虽然我们能够用一些基本的门电路来表示更复杂的逻辑函数，但这并不意味着复杂函数的实际设计就能直接由这些逻辑门电路得到。时刻记住，在大多数情况下使用一些电路模块可以降低设计的复杂程度。因此，我们会经常讨论电路的逻辑实现，而不是其实际实现。同时，我们也会给电路模块增加一些附加功能，或者稍微改变一点电路功能。如果你翻阅一下数字逻辑集成电路制造商的数据手册，你也许会看到某个标准电路有三四种不同的版本。例如，可能有这样的JK触发器，它在时钟信号的上升沿改变状态，而不是下降沿改变状态。制造商这么做是为了扩大电路设计人员对这种部件的需求量。

例如，假设你公司提供的JK触发器仅仅是下降沿触发的触发器，而大量的用户不想通过在电路中插入一个非门来把时钟信号上升沿转化为时钟信号下降沿，那么这个产品就不能让

这些顾客满意。他们会涌向你公司的市场部，极力说服你们需要在产品系列中增加一款上升沿触发的JK触发器；否则他们会撤销订单，转投能满足他们需要的其他公司。

D触发器是这种情况的一个极好例子。现在几乎所有的D触发器都是上升沿触发的器件，输入端D上的数据是在时钟信号的上升沿传递到输出端Q的。另外，RS输入的最初功能被保留了下来，以提供对设备进行异步置位和复位的功能。我们可以将D触发器的功能总结为：在时钟上升沿即将到来时，输入端D的逻辑电位在时钟信号由低向高切换（上升沿）时传递到输出Q。输出端Q将一直保持这个信号，直到时钟输入的下一个上升沿。

图4-7显示了D触发器的逻辑功能。

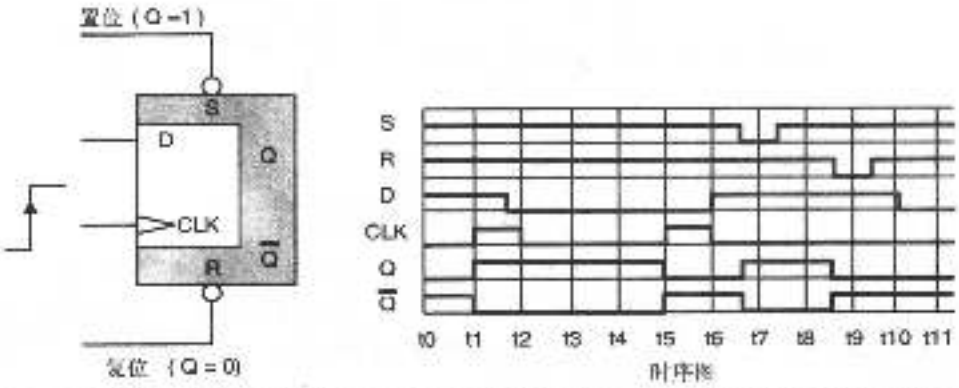


图4-7 D型触发器。灰色区域代表重复RS触发器功能的电路部分，无阴影区域是主从电路部分。通过它加载时钟信号和数据（"D"）输入。D触发器的时序图显示在右边

时序图只是一种观察途径，用来考察随时间变化时多个信号时间之间的关系。如果你见过电视或电影里（或真实生活中）测量人体波形的情况（比如测谎仪），你会看到几个身体状态参数的图线在条带记录纸上同时绘出。换句话说，时序图也可看成是对D触发器进行波形测试的结果。

注意图中时钟输入端的上箭头标志"↑"，它表示一个上升沿。这表明仅在时钟（CLK）输入信号变化之前的一瞬间，输入D上的值才会被捕抓到并传递给输出Q。在D触发器时钟输入端的符号"▷"说明这是一个边沿触发（edge-triggered）的器件，它对逻辑电位的实际值不敏感。

对于R和S输入信号，我们在其输入端画了一个"气泡"标记，说明这两个信号都是在低电位时活跃（active）或有效（asserted）。从图4-7中的时序图可以看出这点。刚过t6时刻，$\overline{S}$信号就变低。再经过一点传输延迟时间，Q变高而$\overline{Q}$变低。注意这时没有任何时钟信号的变化，$\overline{R}$和$\overline{S}$都是异步输入信号，它们可以不管时钟信号的行为。类似地，t8时刻后$\overline{R}$被触发，门的输出又变回原来的状态。

图4-7的D触发器由两个单独的电路功能组成。从前面的讨论我们知道它实际采用的是主从电路结构。输入$\overline{R}$、$\overline{S}$和输出Q、$\overline{Q}$一起形成了一个类似于我们在图4-1中见到的RS触发器。这两个输入比D输入和"D型"部分电路中的时钟输入的优先级更高，在任何时候只要$\overline{R}$或者$\overline{S}$低有效，输出就会被强迫成为相应的值。

图4-8显示了一个D型触发器的逻辑电路实现。虽然从中可以明显地看出主从电路关系，但这个电路实现不再像前面讨论的JK触发器那样容易看清楚。

图4-8中的电路使用了6个与非门实现D触发器的设计。在前面我们使用了两个反相门，故一共需8个门来将JK触发器变成D触发器。此外，注意在图4-8中没有从输出到输入的反馈路径来实现翻转功能。在D触发器中，触发器输出翻转并不是一个工作模式，因此不需要增加反馈

来实现这个功能。

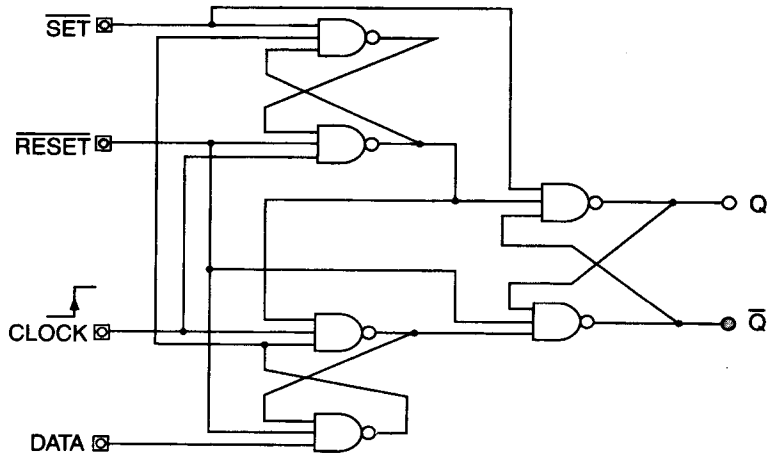


图4-8 D型触发器的门电路描述

触发器输出翻转是指，当有一个时钟输入脉冲时输出信号就改变其状态，这是一个数字系统中非常有用的功能，下面让我们更仔细地研究它。图4-9显示了在一个D触发器上加入信号翻转功能的电路。为了实现这点，$\overline{Q}$输出被接回到D输入端。在JK触发器中，翻转功能是通过分别将输出Q和$\overline{Q}$接回到输入K和J上实现的。

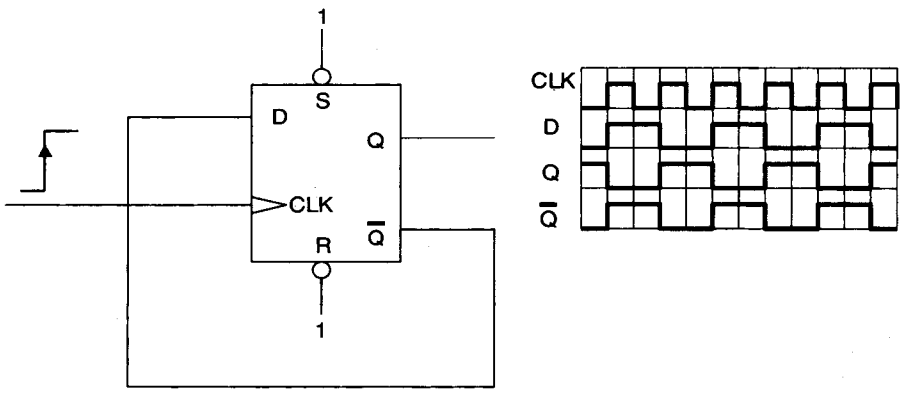


图4-9 在分频电路中使用D触发器

参考图4-9中的时序图，我们看到当在触发器中加入翻转功能后，每两个时钟脉冲在Q或$\overline{Q}$上产生一个完整的时钟脉冲。每次出现一个时钟脉冲，都导致输出信号翻转一次，这样就需要使两个时钟脉冲的输出翻转回原来的状态。

另一种表述方式是，输出Q和$\overline{Q}$上波形的周期是时钟周期的两倍。由于频率是周期的倒
78 数，所以输出Q上波形的频率是时钟输入上波形频率的一半。这样，如果时钟输入的频率为1MHz，则Q输出的频率为0.5MHz，即500KHz。

如果我们把另一个这样的电路放在图4-9中电路的右边，并且也把这个电路的$\overline{Q}$输出接回到它的时钟输入，这样得到的输出信号频率会再减一半，图4-10显示了这样的情况。参考图4-10，输入时钟信号的波形和它导致在输出Q1和$\overline{Q1}$上的改变示于图中，并且它和图4-9中的完全一样。现在，如果我们连接$\overline{Q1}$使它成为右边第二个D触发器的输入，那么电路的输出行为将完全一

样，只是结果波形的周期加倍（频率减半）。

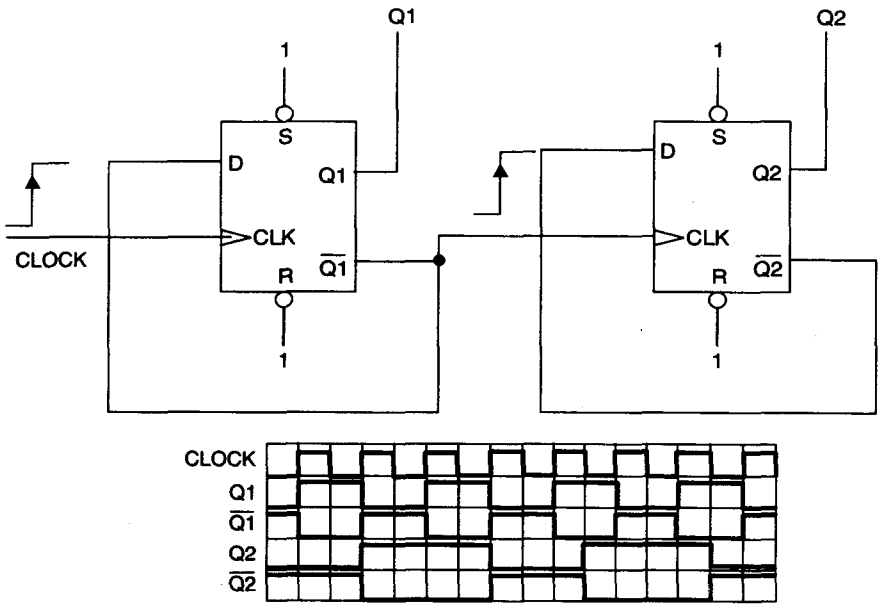


图4-10 两个D触发器串联。一个触发器的 $\overline{Q}$ 输出成为下一个触发器的时钟输入，产生的时钟输入和Q信号的变化示于时序图中

很明显，我们可以增加任意多的D触发器来重复这个练习。每增加一个D触发器，这个器件的输出信号周期就翻倍一次。一般地，对于N个D触发器形成的一串，第N个触发器的周期是这个链上第一个触发器输入时钟周期的2^N倍。只要将足够多的触发器联成一串，我们可以容易地将几兆赫兹的输入波形变成一个每秒一个周期的缓慢波形。

现在，让我们进一步看看这种D触发器串的另一个性质。在图4-11中，我们看到一个含4个D触发器的链式电路。与前面讨论的电路的唯一区别是所有输入 $\overline{R}$ 均被连在一起，接到一个不时会产生一个复位（RESET）脉冲的信号上。这样，只要我们加载一个负脉冲到这个电路的4个 $\overline{R}$ 输入上，那么不管时钟信号是否到达，所有的Q输出都将被强制为0。当我们想让这个电路从一个已知状态开始运行时，这个功能非常方便。事实上，这就是一般台式电脑上那个RESET（重启动）按钮的功能。 79

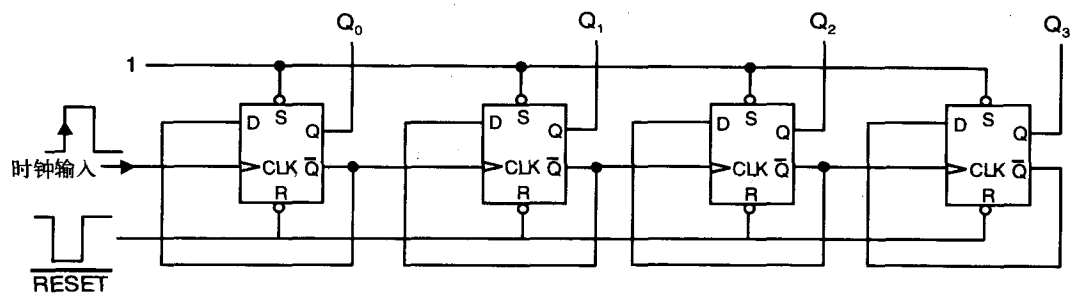


图4-11 用D触发器构成一个4位计数器和16分频器

在图4-11中，有4个和图4-9中电路一样的电路。注意，我们把所有 $\overline{S}$ 输入接在了一起，并

且固定了它们的电位为逻辑电平1，因此，我们再没有机会将Q输出“设置”为1了。在数字设计中，经常把不用的输入端固定接到逻辑电平1或0上，这样可以保证在那个输入端出现信号噪声的情况下电路不会意外地改变状态。通过对RS触发器的讨论，我们知道赋予 $\overline{R}$ 低电平信号将迫使Q输出为0，这样就给了电路一个已知的初始状态。

从前面的讨论我们知道，分频电路中每一个后面的D触发器（或者电路的下一级）将输入的时钟信号一分为二，因此Q_3输出端的信号输出频率是它左边第4级时钟信号的16分之一。如果在CLK输入端是一个16MHz的时钟信号，那么在Q_3输出端得到的将是一个1MHz的信号。参考图4-12中逻辑分析仪的显示图像，我们可以看到这个分频的实际过程。逻辑分析仪不像示波器那样能显示高保真的波形图，它能同时显示多个波形，但会损失一些处理信息，因此，逻辑分析仪得到的视图更像图3-15。

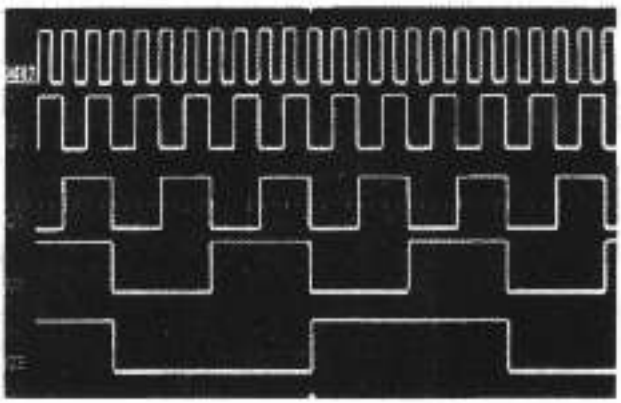


图4-12 用逻辑分析仪显示一个由4个D触发器构成的16分频电路的波形

让我们再仔细地看看图4-12。时钟信号是最上面的那个波形，而每个后续D触发器的Q输出由下面的一个波形表示。让我们比较一下CLK波形和直接在它下面的那个Q0波形。图4-13是图4-12中一部分的放大图。为了易于辨认，时钟的每个上升沿都用数字进行了标记。注意在每个CLK输入的上升沿，Q0输出都改变状态。由于D触发器需要信号上升沿来促使输出改变状态，这导致的直接后果就是需要2个输入信号的上升沿才能使Q0输出走完一个完整的变化周期。

80

我们需要重申，图4-11的电路结构还有一个有趣的特点。我们前面说过，所有的输入 $\overline{R}$ 是连在一起的，这样只要将它们置为有效，那么不管时钟输入的状态如何，所有的Q输出都将被强制为0。这样，为了从一个已知的状态（所有输出= 0）开始运行这个系统，我们就必须激活这个RESET输入。

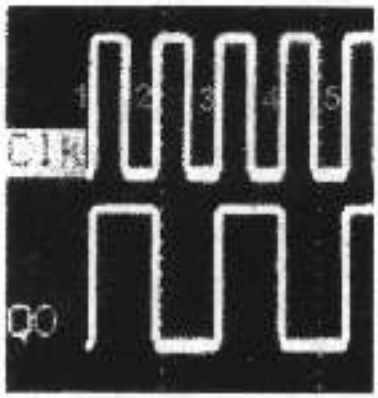


图4-13 图4-12中时钟输入和Q0输出部分的放大视图

假设我们刚刚激活RESET信号，从Q_0到Q_3的4个信号都为0，这时左边的CLK输入端还没有时钟脉冲进来，这个电路处于静止状态。突然，在CLK输入处出现了一个时钟脉冲，由于 $\overline{Q}$ 输出为1，D输入为1，所以时钟脉冲刚过，这个最左边D触发器的Q输出就变成了1，$\overline{Q}$ 输出的值由1变为0。这是一个下降沿，所以对第2个D触发器没有影响。

当第二个时钟脉冲到达CLK输入端时，第一个D触发器变回原来的状态，它的输出为0，但是，它的 $\overline{Q}$ 输出此时从0变为1，导致Q_1变为1。如果从每个时钟脉冲到来之处那列波形向下看，这就是我们在图4-12中看到的。作为练习，请画一张真值表，其中最左边一列顺序列出时钟脉冲的编号，而从Q0到Q3的值对应4列输出。从时钟脉冲0（即电路的RESET状态）开始。

如果做完了这个练习，你马上会发现这个电路也是一个二进制计数器，它的值从0000变

到1111，然后在第16个时钟脉冲时再变回0000。不要惊讶，我们甚至可以一级一级地增加触发器来实现任意位数的计数器。这个独特的计数配置称为*行波计数器*（ripple counter），因为在每一级状态改变时计数值像波浪一样传递。计数值“上下起伏”地通过这个计数器电路意味着，在前一个脉冲导致的计数改变完全传递到整个电路之前我们有可能接收到下一个脉冲。虽然计数器仍会保持准确的计数，但这将对我们准确地读取这个计数值带来影响。

到现在为止，我们所关注的数字电路中每个输入或输出变量都在独自占用一根信号线。如果想在一个微处理器中传输32位数据，我们需要使用包括32根线的一束信号线来移动数据。由于每一个数据位或变量是并行于其他数据位传输的，所以这种情况称为*并行*（parallel）数据分布。此外，还有一些情况下需要使用*串行*（serial）数据传输方式，在这种方式下所有数据位顺序地通过一根线或者最多几根线。你可能比较熟悉的一些并行数据传输协议的例子是：

- 并行端口，LPT
- PCI总线
- AGP总线
- IEEE 488（HP-IB）

而串行数据传输协议的例子是：

- RS-232端口（COM端口，COM1, …, COM4）
- 以太网
- USB端口
- Firewire

我们现在看一个由D触发器搭建的电路，它可将串行数据格式转换为并行数据格式。显然这个功能非常重要，一旦收到了串行数据我们必须有办法让计算机能处理它。我们使用的这个电路结构称为*移位寄存器*（shift register），请看图4-14。 81

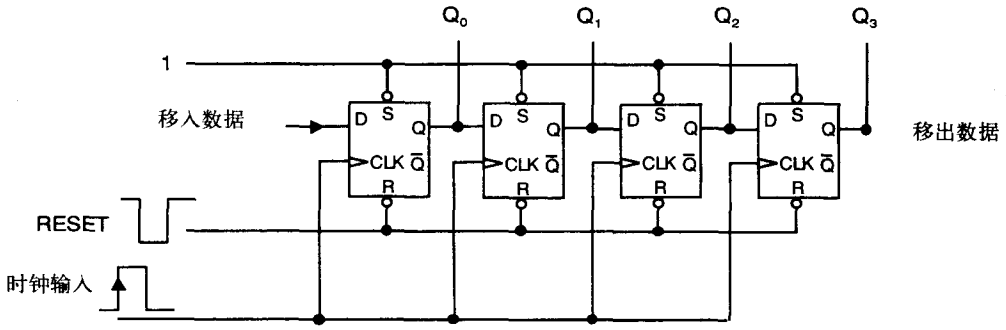


图4-14 由D型触发器构成的4位移位寄存器

像图4-11中的行波计数器一样，这些触发器的RESET输入端（R）连在一起，这使得我们可以让所有触发器输出同时变为0。然而，与图4-11电路最大的区别是，我们将CLK（时钟）输入也连在了一起，而且每个触发器的Q输出都直接接到了下一个触发器的D输入上。这样，每当CLK输入端出现时钟上升沿时，D触发器就获得它左边那个触发器的Q输出值。由于所有的CLK输入同时激活，所以随着每次时钟的上升沿，最左边那个D触发器上的输入数据看上去就像在这个移位触发器上从左到右地一步步移动似的。

让我们看看这会是一个怎样的波形。假设我们想通过一个串行线传递一个数字6，即二进制的0110。将并行数据转换为串行数据的方法类似于将它从串行变为并行的过程，这里我们

不考虑它。我们现在假设数据0110是一个按时钟信号传输的一个波形，如图4-15所示。

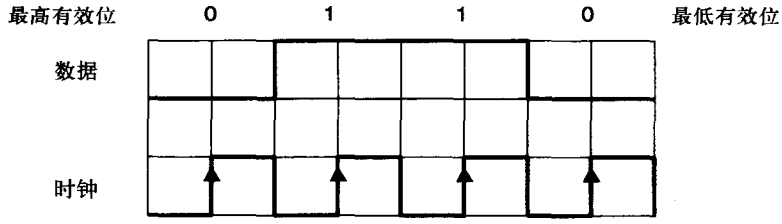


图4-15 二进制数6的串行数据传输

数据值按时钟的上升沿进行同步，而这个同步时钟信号既可能随这个数据进行传输，也可能不随这个数据进行传输。有时时钟信号是由数据接收设备自己生成的，而不是像数据那样从发送设备那里传过来的。计算机上的COM端口就是这样工作的。另外一些技术涉及将时钟信号结合进数据中，并用一根信号线传输它们，而在接收端使用特殊电路恢复数据中的时钟信号。无论哪种情况，都需要时钟信号的同步，以便在触发器的D输入端捕捉数据，所以，需要4个时钟脉冲，每一个对应地接收一个数据位。

图4-16显示了4个数据位通过D触发器的过程。让我们假设在这个例子中使用两根不同的
82 线分别传输数据和时钟，就像图4-15所示的那样。图4-15实际上是一个信号随时间变化的图，因此假设我们有一支很快的笔和纸，每次时钟信号出现上升沿，我们就记录另外一条线上的信号状态，我们将依次看到0、1、1、0。这个图乍一看可能不是那么清楚，我们下面就分析它的意义。在t1时刻的时钟沿到来之前，最左边D触发器的D输入为0，而且所有触发器的输出均已复位为0。在t1时刻，D输入上的0被传输到Q输出，由于输出已经是0，所以就像什么都没发生一样。

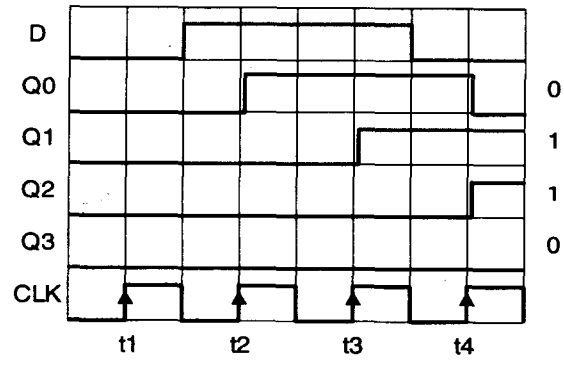


图4-16 串行数据通过4位移位寄存器的过程

现在快要到t2时刻了，最左边的D输入变为1。刚过t2时刻，Q0就变为1，反映出D输入上的值。所有其他的输出还是0，因为它们仍然在传递第一个数据0。在t3时刻，第3个数据位1被时钟脉冲打入最左边的D触发器，而其他数据在向右边移动。这样，刚过时刻t3时，从Q0到Q3的输出分别是1100。最后，在t4时刻之后，数据位又右移了一次，输入的串行数据就完整地存储于移位寄存器中成为并行值了。

4.3 存储寄存器

我们将要看到的D触发器电路的最后一个例子也可能是所有中最重要的，就是*存储寄存器*（storage register）。存储寄存器是一组D触发器，它被设计用来接收和保存数字数据值。这个数据值可以是位、半字节、字节、字、长字、双字，等等。关键点是，一个时钟信号可以使所有的D触发器同时存储它们D输入端的数据，在这个意义上，它是一个并行输入/并行输出设备。

与存储寄存器相比，图4-14所示的移位寄存器是一个串行输入/并行输出的设备。我们之所以需要存储寄存器，是因为计算机中的数据流动瞬息万变，我们必须有一个临时存储这些数据的地方，使得我们能把它们保存到我们需要的时候，或者存储计算指令的结果直到我们将它们移动到某个其他地方。

图4-17显示了一个存储寄存器的配置图以及其中的8个D触发器。我们可以看到，在D输入端（图中的深灰色线）上的任何数据经过一个时钟上升沿之后将存储于寄存器的触发器中。然后，这些数据就可作为输出（浅灰色线）传递出去。然而，关键点其实是输入数据值（深灰色线上的信号）现在可以改变，但寄存器仍然保留以前的数据。

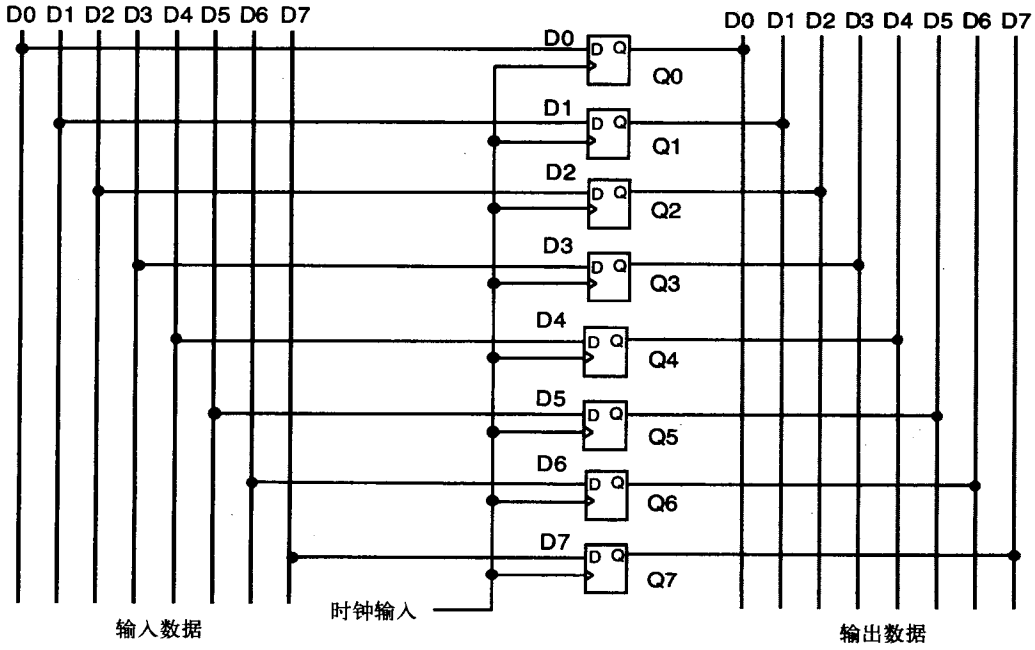


图4-17 一个8位存储寄存器。在时钟上升沿时D触发器捕捉到输入数据（深灰色线），这些数据随后出现在输出端（浅灰色线）。这种寄存器通常没有SET和RESET输入，因此图中也没有画出它们

在大多数计算机中，形成存储寄存器的D触发器的数目和计算机数据通路的数据位数是一致的。当前我们用的计算机中，数据通路的宽度从4位到128位不等。这些寄存器是现代计算机和微处理器的关键“数据容器”。在后面的章节中，你将看到不同的存储寄存器的数量和类型定义了不同的计算机体系结构。 83

为了强调存储寄存器是一个独特的电路元件，而不是一组D触发器，我们将图4-17重画为图4-18。

在本章前面的部分，我们反复提到一个触发器的“状态”这个概念。实际上我们想说的是，为了知道Q和 $\overline{Q}$ 输出的新值，我们必须了解适当的时钟信号边沿将要到来之前输入和输出的当前值（或状态）。所有这些都是为了引入状态机的概念而做的准备。

状态机的状态代表了一个数字系统可能存在的所有输入和输出的组合，也包括了这个系统在状态机的各种状态之间变化的路径。此外，也可能是最重要的一点是，状态机可以根据其输入条件的变化改变其状态迁移的路径。不像我们前面学习的异步组合逻辑，状态机是一种同步设备。这意味着状态机仅能在时钟的边沿改变状态。从中你可能会得到暗示，D触发器

和状态机应该有某种共同的血缘关系。这完全正确！通过一个表或一个图（称为状态迁移图（state transition diagram）），我们可以最好地描述一个状态机的行为。请看图4-19。

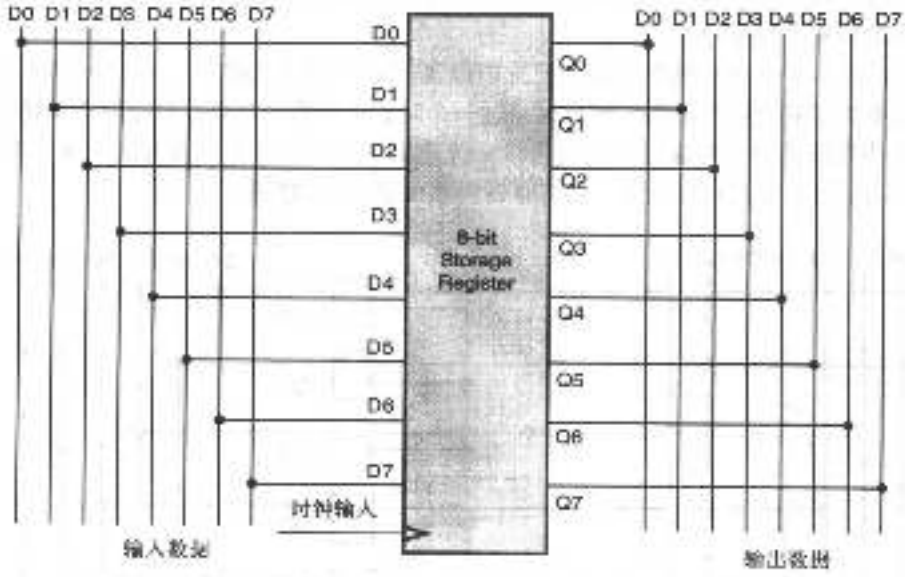
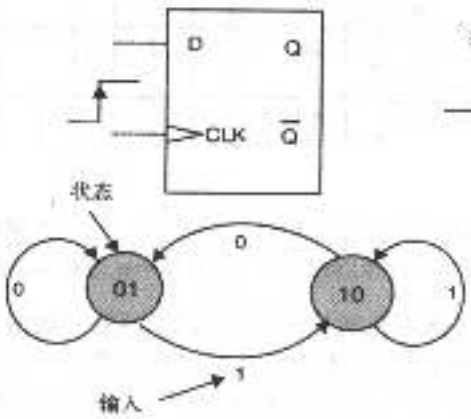


图4-18 重画图4-17中的电路，以显示8个独立的D触发器聚集成一个总个器件。因为存储寄存器是大多数计算机系统的有机组成部分，所以它应该被看成是完全不同于D触发器的



当前输入	当前状态	下一状态
D	Q	Q^n
0	0	0
1	1	1
1	0	1
0	1	0

图4-19 一个D型触发器的状态迁移图和真值表。为了清晰，省略了异步输入SET和RESET

图4-19中深色的圈是系统的状态。这个系统有两个状态：01和10，分别对应于Q=0、$\overline{Q}$=1和Q=1、$\overline{Q}$=0。带箭头的线表示状态间的迁移，而其旁边的数字表示在迁移时的输入条件。因此，如果当前设备处于状态01，则当D=0时它将保持这个状态，而当D=1时它的状态变为10。类似地，如果设备处于状态10，那么当时钟信号到来时，如果D=1则它保持这个状态，如果D=0则转换到状态01。

84

状态之间的转换发生在时钟的上升沿到来的时候，图中的箭头暗示了这一点，即便是时钟信号到来之后D触发器还保持原来状态的情况，我们仍然画一根带箭头的线表示经过状态转换它还回到原来的状态。

85

与状态迁移图一样，真值表（示于图4-19的右半部分）也提供了状态机的有关信息，只是没状态迁移图那么直观。为了简单，我们在真值表中忽略了D触发器的 $\overline{Q}$ 输出。表的4行分别代表4种可能发生的状态迁移情况。值得注意的是，输出变量有两个含义，分别表示在时钟脉冲上升沿到来前后的输出值。

图4-20是我们讨论过的图4-11中4位计数器的状态迁移图。由于这个电路中除了时钟外没有任何同步输入信号，所以这个状态图非常简单、直观。无论这个计数器当前是什么状态，它的下一个状态都是预先确定的。如果RESET输入可以是同步信号（这很容易办到，但我们不想增加麻烦），那么图4-20中的每个圈都可以通过一根箭头线回到0000状态。这些RESET箭头线上都应该加一个RESET = 0的标签，用来说明RESET输入为0将使计数器在下一个时钟脉冲到来时回到0000状态。而所以其他的箭头线上则应标上RESET = 1，说明正常情况下计数器如何迁移到下一个状态。

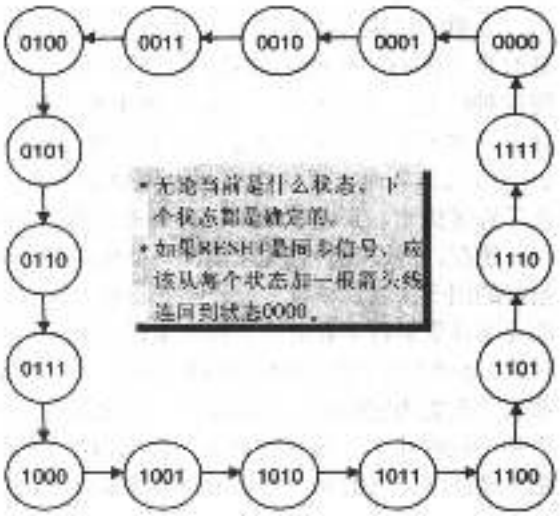


图4-20 一个4位计数器的状态迁移图

在前面的第3章中，我们考虑过一种实现异步逻辑的可能方式是直接将真值表存在存储设备中，就像图3-9所示的那样，这样可以立刻消除所有门电路设计中的问题。当时我们没有讨论这种系统与其他电路实现的优劣对比，只是指出我们在后面还会详细说明。现在让我们详细地讨论一下这个问题！但是，在我们讨论这个与状态机有关的问题之前，先回顾一下计算机存储器的结构可能会有帮助。我们在后面某章中会学到更多的关于存储器的知识，但现在让我们复习一下编程课上学习的有关存储器的内容。

一个存储器件，比如说随机访问存储器RAM或只读存储器ROM芯片，包含了大量的存储单元（memory cell）。每个单元可以存单个1值或0值。可以想像，这些单元可以按各种组合组织成不同存储字宽（memory width）的存储器。只要我们知道了一个存储设备总共有多少个单元以及这个存储器的字宽，就可以算出它需要多少存储地址（memory address）。举一个例子可能说得更清楚。假设我们有一个存储器，它包含16 384个存储单元，每个单元存一个二进制位的数据。如果我们将它设计成每次读存储器访问可得到一个8位的数字，换句话说，我们想用这个存储器存字符（char）类型的数据，那么，总的地址数就是16 384/8 = 2 048，这样，该存储器将包含：

- 16 384个存储单元，
- 用来读写存储器数据的8根数据线，
- 2 048个可单独寻址的存储器位置，
- 提供2 048种不同地址组合的11根地址线。

你可能想知道这个11根地址线是怎么算出来的。考虑到我们要访问2 048个不同的地址，因此第一个地址应该是0000，而最后一个地址应该是2 047。各位都是0的数也是个有效地址，这看上去有点奇怪，但计算机科学家是一群不怕辛苦的家伙，他们能够处理它！

在二进制数的系统中，2 048个独特的地址表示为从000 0000 0000到111 1111 1111之间的数。注意我们需要11位二进制数字表示从0000到2 047的所有可能地址。由于0000也是一个有效地址，所以一般来说最高地址（所有二进制位都为1）比不同地址的数目小1。

这样，你可以想像这个独特的存储器设备被组织成2 048 × 8的结构。我们可以改变要求，比如说希望用这个存储器存32位字宽的数。由于每个地址对应的位置上都有更多的存储单元，所以只能用较少的处理地址来安排。因为存储器的字宽是前面那种情况的4倍，所以地址数量也就只有原来的四分之一。新的存储器将包括：

- 16 384个存储单元，
- 用来读写存储器数据的32根数据线，
- 512个可单独寻址的存储器位置，
- 提供512种不同地址组合的9根地址线。

这些和状态机有什么关系呢？让我们回忆一下图3-9，那个图中显示了一个真值表，并且看上去非常像一个存储器。图3-9中的真值表有4个独立的输入变量（从A到D）和两个输出变量（E和F）。从前面的讨论中，我们知道这是一个有4根地址线、2位输出字宽的存储器，这个存储器包含32个存储单元。

上面的讨论并不意味着存储器是存储状态机的状态迁移信息的唯一方法。既然存储器只是对真值表的一种精确映射，我们也可以用本章前面的方法来实现这种映射。也就是说，为每个输出变量建立一个卡诺图，并推导出简化的最小项表达式。然后我们就可以得到真值表的门电路表达形式，从而实现状态机。如何做各种选择来实现这个设计取决于许多因素，还是让硬件工程师来考虑这个问题吧。



图4-21 普通的硬件设计者[2]

现在，假设我们想实际设计一个状态机，那么微处理器是一个很好的例子。我们该怎么设计呢？首先，我们可能需要上一些更多的关于计算机科学和电子工程的课程，但这并不能阻止我们在这里讨论，我们可以比较粗略地进行分析。如果要建造一个微处理器，那么首先要为它的核心功能设计一个状态机。这个状态机可能有上千种可能的状态、几百种输入、输出信号（变量），使用逻辑门电路来设计它会很困难，那么用存储器来实现它的真值表将会比较容易一些。下面考虑图4-22。



图4-22 一个16×4存储阵列组织成状态机的真值表形式

这个电路有4个输入变量（从A_0到D_0）和4个输出变量（从A_{out}到D_{out}）。输入变量形成某 87
个存储单元的地址，其值从0000到1111，输出变量则为输入地址指定的存储单元中的数据。图4-23进一步解释了这点。

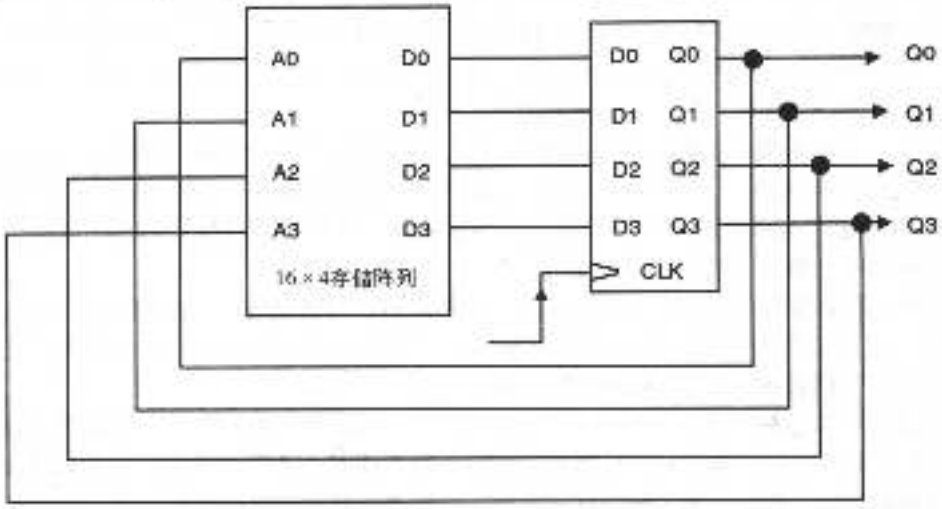


图4-23 用16×4存储阵列和一个4位存储寄存器实现状态机。输出变量反馈回输入提供下一个状态的地址

在图4-23中，从A0到A3是存储器单元的地址，从D0到D3为相应单元中存储的4位数据。 88
状态机的输出，从Q0到Q3驱动剩下的电路，同时也连回存储器的输入来提供下一个状态的地址。这个4位存储寄存器提供了关键的同步功能。由于数据仅在正时钟跳变到来时从D输入传递给Q输出，所以这个寄存器隔离了输入信号和输出信号之间的相互干扰，防止了电路以一种不可控制的方式工作。

在状态机中，这个4位D存储寄存器提供了和边沿触发的触发器中主-从电路完全一样的功能。如果没有这个D触发器提供合适的同步机制，我们的这个电路将变得不可控制。为了解释这一点，请看图4-24中的电路。

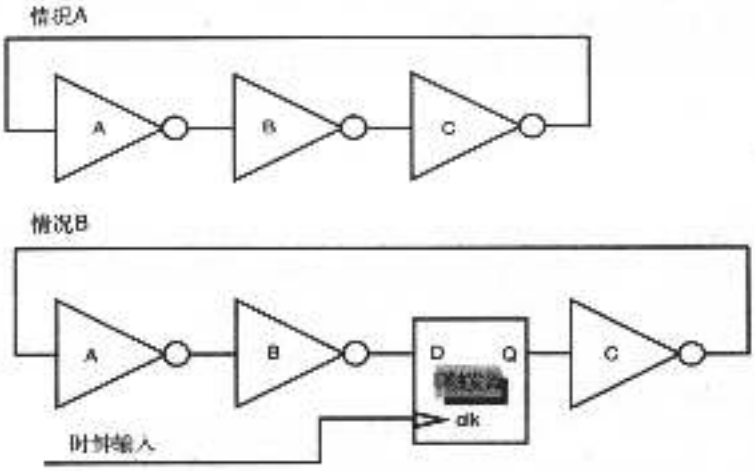


图4-24 情况A：存在竞争条件的电路。由输出反馈回输入的信号总与当前输入信号有相反的极性，电路无法建立一个稳定的状态。情况B：D触发器使用电路稳定，并将状态迁移限制为仅在时钟上升沿时发生

在情况A的电路中，三个非门（其实任何多于1的奇数个非门都可以）连成了一个回路。我们可以这么分析这个电路：假设某一时刻门A的输入变为高电压，经过一合适的时间间隔门A的输出信号将变为低电压，这个时间间隔由门A的延迟时间决定（我们假设它为10ns）。由于A的输出是B的输入，所以上面这个过程在门B和门C上重复，最后信号传递回门A的输入。因此在门A的输入信号变高后再过30ns，这个输入信号将变为低电压，并且这个过程会一直进行下去。经过60ns后，门A的输入再次变高，这就形成了一个完整的周期。这是一个经典的*竞争条件*（race condition），并且一般不应该让它发生。

事实上，如果需要这种情况也不是那么坏。这是我们产生时钟脉冲信号的一种方式，我们称这个电路为*振荡器*（oscillator）电路。通过一些简单的操作，我们可以在这个电路中加入一个晶体，这个振荡器的频率和周期就会非常精确。在图4-24所显示的例子电路中，振荡频率将是60ns的倒数，大约16.7 MHz。

情况B中的电路明显地消除了这种振荡。虽然在每次时钟信号上升沿到来时D输入的值都发生翻转，但这个电路是稳定的。这里没有不受控制的信号状态改变，系统中所有状态迁移都由时钟的上升沿到来并更新D触发器所引起。

上面举的令人愉快的这个小例子和状态机有什么关系呢？关系大了！如果在电路中没有
89 放D触发器，那么将存储器的输出反馈回它的输入将产生竞争条件，状态机对我们就没有用处了。通过D触发器，我们控制了输出反馈回输入、改变存储器地址的时机，这就是我们所希望的状态迁移方式。

这种采用触发器的设计隔离了当前状态（输出Q0到Qn）和下一个状态，仅仅在时钟上升沿到来的那段短暂时间内，触发器的输出可以按照D输入的信号改变状态。在那个时刻，状态机进入新的状态并且将输出反馈回存储器的地址输入端，为下一个状态提供地址。存在那个地址上的数据就为下一次状态转换提供值，并且这个转换在时钟的下一次上升沿到来时发生。

现在，我们可以总结一下这种电路结构：

- 就像真值表包含系统所有可能的逻辑项一样，存储单元中存放了系统的所有可能状态。
- 存储阵列（真值表）的输出是D存储寄存器的输入，并且会在下一个时钟脉冲上升沿到来时成为状态机的新状态。
- D存储寄存器的输出是系统的当前状态。这个输出用于控制外部电路的输入，也为下一个状态提供了存储阵列的地址信号。
- D存储寄存器隔绝了状态机的输出和状态机的输入，仅允许在时钟信号上升沿发生状态转换。

总结

- 信号反馈创造了一种有状态依赖性的电路元件。也就是说，这种元件的输出状态不仅依赖于输入状态，还依赖于时钟脉冲以及时钟脉冲到来之前的输出状态。
- 这种电路的最简单形式是RS触发器，其用处非常有限。
- 一旦加入主-从触发器的功能，我们就能克服由RS触发器设计带来的许多不稳定性。
- JK触发器是这种设备的最通用形式。
- 一个派生出来的设计，D触发器具备数据存储元件的附加特性。
- D触发器可以配置成计数器、移位寄存器和存储寄存器。
- D触发器是实现状态机的关键元件，因为它限制了可能的系统状态变化，让它仅在时钟信号上升沿时发生。

参考文献

[1] Linda Null and Julia Lobur, *the essentials of Computer Organization and Architecture*, ISBN 0-7637-0444-X, Jones and Bartlett Publishers, Sudbury, MA, 2003, p. 116.

[2] 在我曾工作过多年的惠普公司的那个部门里，对硬件设计者和软件设计者有不同的绰号。硬件设计者被称为"蟾蜍"。软件设计者被称为"火鸡"。我不知道这种习惯叫法是怎么来的，他们就这么叫。既能设计硬件也能设计软件的人有时被叫作"公鸡"，我们真实很少这么叫。

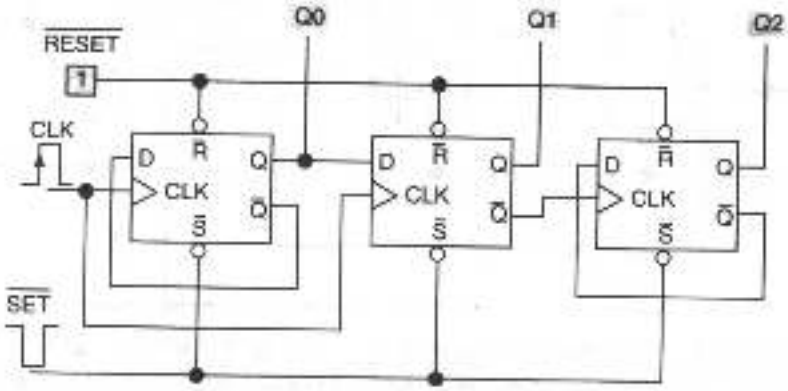
过了这些年，我又发现了一些其他的关于硬件设计者和软件设计者的不同现象。硬件设计者经常穿那种不同系鞋带的耐克慢跑鞋，而软件设计者则经常穿凉鞋（Birkenstock）。我过去习惯于穿慢跑鞋，但最近我发现我越来越多地穿Birkenstock了。是在计算机系教书改变了我的本性吗？只有时间才能给出答案。 90

习题

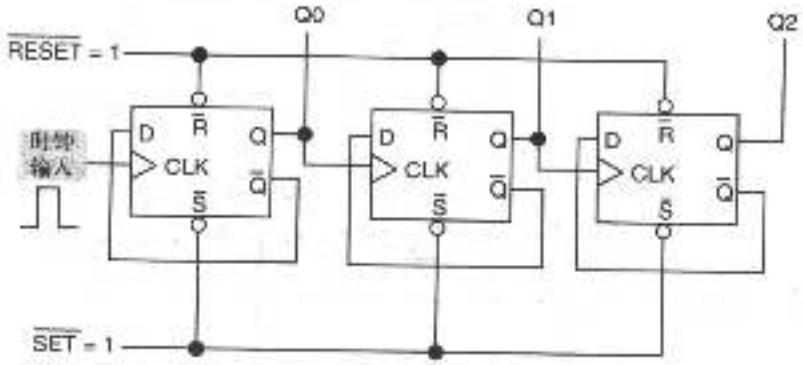
（注意：用星号标记的题目尤其有挑战性）

1. 下面图中显示的电路包括三个D型触发器。图中黑圆点表示相关的线互相之间有物理连接。$\overline{\text{RESET}}$输入端永远接在逻辑1上，因此永远不会有效，在时钟脉冲到来之前，所有的$\overline{\text{SET}}$输入都会收到一个负脉冲以建立电路的初始状态。

画出这个电路的状态迁移图。注意按Q0、Q1和Q2的顺序表示状态。一定要显示题目中给出的起始状态，以及所有其他的后续状态。用带箭头的线表示从一个状态到另一个状态的迁移。



2. 下图显示的是一个由正边沿触发的D触发器组成的3位计数器：

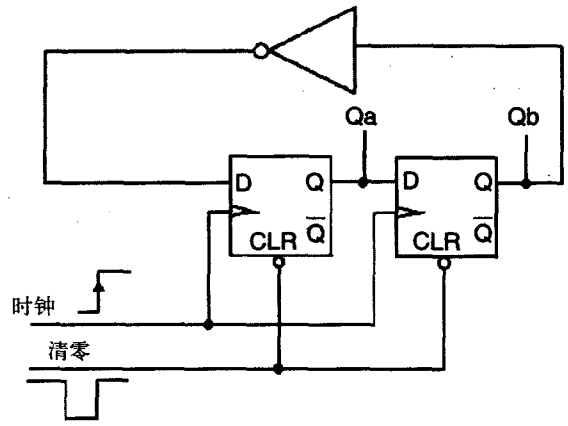


假设在时刻T0，刚刚出现一个$\overline{\text{RESET}}$信号，使得输出Q0、Q1和Q2为低电压状态。请画出输出Q0、Q1和Q2的时序图。

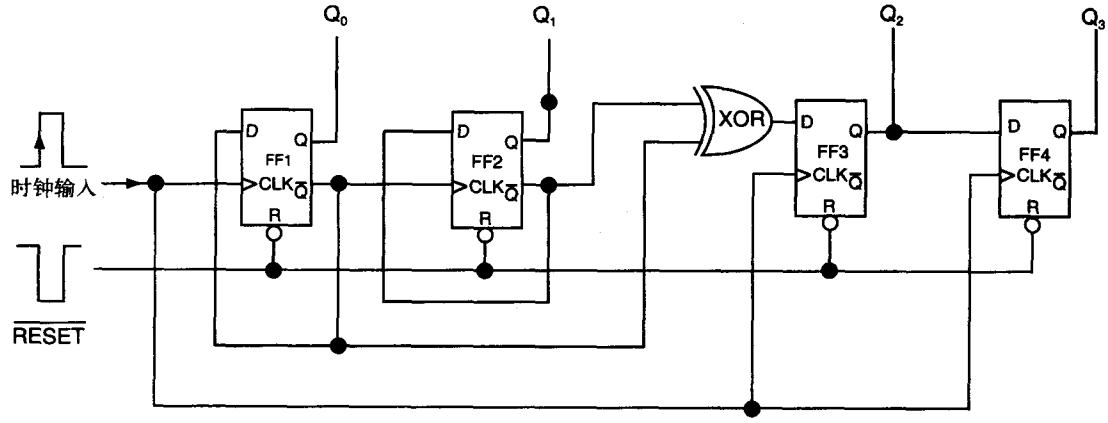
为了简单起见，你可以假设信号通过触发器时没有延迟，这样一旦输入时钟信号出现上升沿，输出就改变其状态。

91

3. 考虑下面显示的电路。假设已经出现了CLEAR脉冲信号让状态清零。为这个电路创建一个真值表，显示经过4个时钟周期后的输出状态。

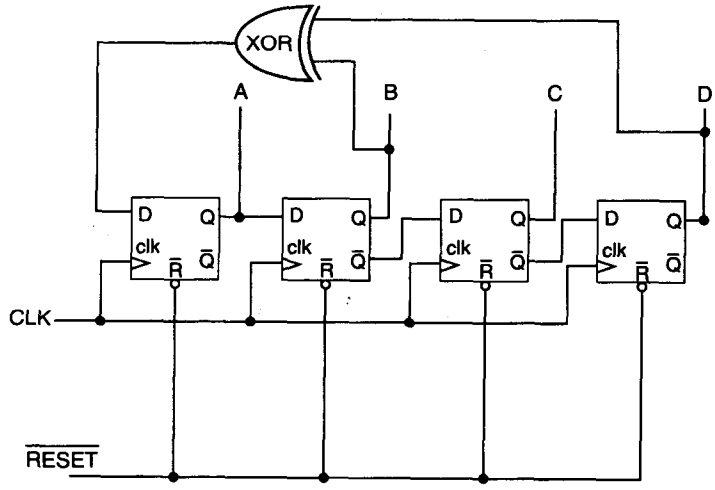


4. 下面显示的图中有4个D型触发器和一个异或门（XOR）。首先RESET信号将所有Q输出清零。图中线交叉处的黑圆点表示两根线有电学上的连接，否则没有。请创建真值表，显示经过4个时钟周期后Q0到Q3的输出。



92

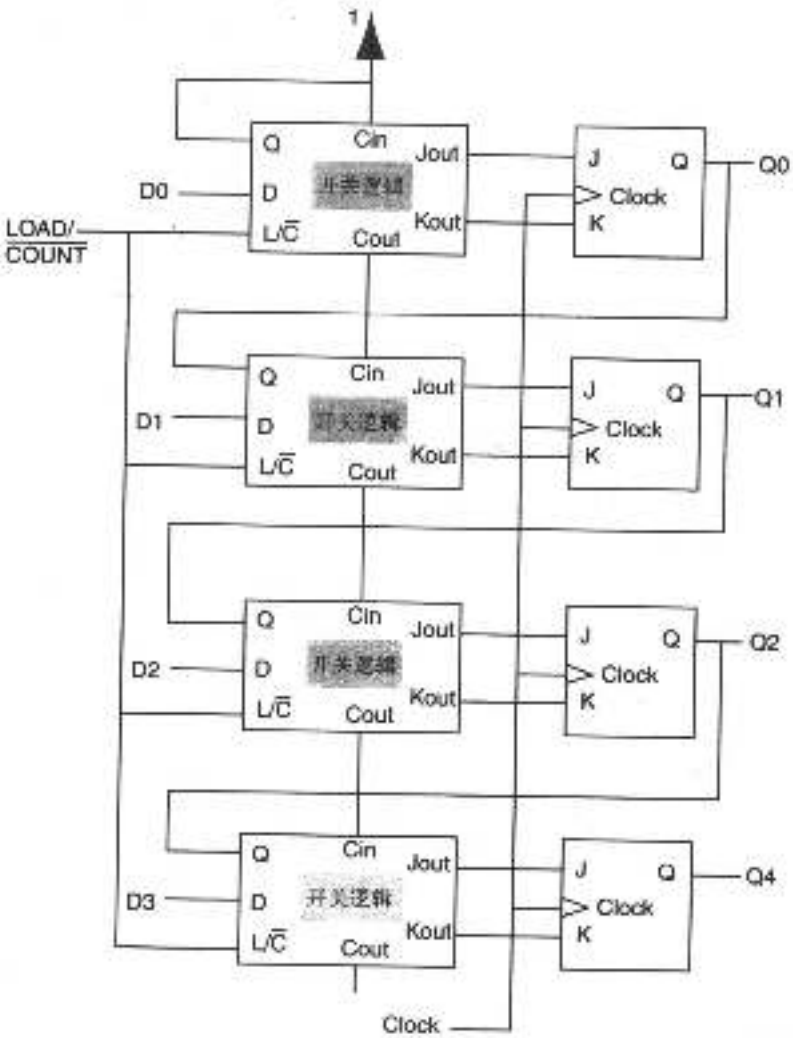
5. 考虑下面的电路，它包括4个D型触发器和一个异或门（XOR）。假设电路上加载了一个RESET信号，让输出A、B、C和D都变为零。请创建真值表显示经过8个时钟周期后的输出状态。这个输出模式会重复吗？如果会，多久重复一次？



6. 考虑由8级D触发器串联构成的计数器/分频器电路。假设通过一个触发器的传输延迟是25ns。这里传输延迟是指从时钟上升沿出现到输出状态改变并稳定的时间。在不至于出现下一个时钟信号到达时最后一级计数器还在改变状态的前提下，最大可能的输入时钟频率是多少？你可以假设时钟输入为一个方波信号。

7. JK触发器最常见的用处是建造同步计数电路。同步计数器不同于行波计数器，主要区别在于时钟信号并行地输入到所有的触发器，每级触发器根据J、K输入的状态或者翻转或者保持其现有状态。因此，这种计数器比行波计数器速度快得多。请使用JK触发器设计一个4阶段同步计数器，类似于图4-6中显示的那种。提示：你将额外需要三个2-输入与门。 93

8*. 下面的电路被称为带并行载入功能的同步计数器，它在计算机应用领域非常有用。LOAD/ $\overline{\text{COUNT}}$输入端决定了这个电路的功能是同步向上计数器还是并行载入寄存器。例如，保持LOAD/ $\overline{\text{COUNT}}$端为高电压，那么时钟输入的脉冲将D0到D3的值载入JK触发器。如果LOAD/ $\overline{\text{COUNT}}$输入为低电压，时钟输入上的脉冲则使计数器从以前载入的值开始向上计数。这样，如果数据值 (D0, D1, D2, D3) = (0, 1, 1, 0) 通过为高的LOAD/ $\overline{\text{COUNT}}$输入信号载入电路，那么LOAD/ $\overline{\text{COUNT}}$变低后的下一个脉冲将导致开始向上计数，输出Q0到Q3变为 (1, 1, 1, 0)。



图中标为“开关逻辑”（Switching Logic）的框为实现并行载入同步计数器基本功能的电路。请设计必要的门电路实现这个设备的开关逻辑。

提示：完成前面的习题将有助于理解这个电路设计的要求。一旦完成了习题7，为这个开关逻辑构造真值表和卡诺图就比较直接了。 94

第5章 状态机简介

学习目标

- 描述状态机的操作；
- 设计一个简单的状态机；
- 解释如何用状态机控制一个简单微处理器的指令顺序。

5.1 引言

有关计算机系统中状态机的话题值得我们做出更多的讨论。既然这样说了，就让我们奋力前进，以获得对这一主题的了解。你可能从来都没有意识到，硬件设计实现和软件设计实现之间存在对称性。作为软件开发者，你已经熟悉了算法是什么，你可能在其他的计算机课程中已经写了很多不同的算法，你甚至也许在某门单独的课程中学习了算法的性质。在本讲中，我们将了解如何只用硬件而不是用一组C或C++指令来产生算法。

算法可用硬件解决，与用软件同样容易，这是一个事实。例如，你可能是一个“游戏玩家”，正在你的PC机上享受视频游戏的乐趣。你知道要真正进入游戏，就需要一个强有力的视频卡，能够高速地生成图像。如果没有这样一个卡，你的游戏也能玩，但与有快速视频卡时相比，它会很慢而且缺少真实感。

另一个例子是PC机上的56Kbps的调制解调器。比较昂贵的调制解调器是全自治的，价格在80美元到150美元之间。较便宜的“Win调制解调器”必须与Windows操作系统一起使用，而且需要足够快的PC以保证正确运行。这种调制解调器的价格可能在10美元到50美元之间。两种调制解调器的区别是：较昂贵的调制解调器是在硬件中做数据转换，而Win调制解调器则是在软件中做数据转换。

另一个好的例子可见于最近一批投入市场的低价格激光打印机。前一代的激光打印机包含了高性能硬件，用以将输入数据流转换为感光鼓上的激光印记图案，这个数据转换的过程称为*光栅化*（rasterization），类似于在CRT屏幕上生成图像所采用的方法。光栅化是由激光打印机的专用硬件处理的，但随着PC的计算能力越来越强大，将光栅化过程移回到PC作为打印机驱动程序的一部分就有了商业意义。打印机简单地将到来的光栅数据转换为感光鼓上的激光位。当然，你只有采用适合的软件驱动器才能使用打印机，打印机不能作为一个独立的设备。

一般来说，一个算法用硬件解决比用软件解决要快得多，有时要快很多个数量级。目前，已经发展出一种产生*专用集成电路*（application-specific integrated circuit, ASIC）的方法，作为产生算法的硬件实现的一种途径。在PC中的视频处理器、语音芯片、调制解调器芯片都是ASIC的例子，但它们更流行于工业界。在当今很多的产业和消费应用中，ASIC做数据处理和数据操纵，而微处理器做错误处理和初始化。

在前面章节中，我们接触到了系统状态的概念。触发器的引入就自然产生了基于状态的系统。也就是说，一个装置的输出不仅取决于其输入，还取决于其输出的目前状态。同样，

我们还介绍了这样的系统概念：只有出现同步时钟信号时，状态才能改变。

我们将要接触的是一类有顺序的数字系统行为。我们很快就会看到，在进行从存储器取出指令、对指令译码、执行指令、将结果写回存储器，并重新开始该序列行为的过程中，计算机是如何通过有限步骤的序列来完成这些工作的。这是一个完美的时序过程的例子。

计算机被引领走过这一序列步骤就构成了状态机引擎的操作。该引擎可如我们在本章例子中所见的那样由存储器中的数据控制，也可以如我们早先学设计时所看到的由组合逻辑控制。我们也将做一个这种类型设计的例子。我们能够在概念上描述和实现成为同步数字系统的任何时序过程均可称为有限状态机（finite state machine）[1]。我们可以将时序（有限状态）机定义为服从如下一组要求的模型：

- 它可处于某有限状态集合S中的一个状态s；
- 存在一个初始状态s_0，它是集合S中的一个成员；
- 输入i的有限集合I；
- 次态函数$d = d(s, i)$将现态值和输入映射到次态（在S中）；
- 输出函数$f = f(s, i)$。

上述表达是对我们直觉上已知东西的描述吗？图4-20显示了4位二进制计数器的状态迁移图，它服从我们对于有限状态机（FSM）的定义吗？

- 它仅可能在一个具有16个状态的集合中存在，集合中的状态标记为从0000到1111。这16个状态就定义了S。

96

- 有一个初始状态0000，该状态在S中，可通过对设备的初始化达到，也可通过一序列步骤达到。
- 在该例子中无输入，但这并不会减弱分析的正确性。
- 次态函数取决于现态。
- 计数器的输出取决于计数器的现态。

如果f仅是其现态的函数$f = f(s)$，就像4-位二进制数这种情况，我们称之为Moore机（Moore machine）。如果f是现态和输入的函数$f = f(s, i)$，则称之为Mealy机。如果状态机在状态s接收到输入i，则它要进入的次态就由$d = d(s, i)$来决定。

我们可将FSM原理应用于对诸如视频卡或调制解调器等过程性问题的求解。当我们采用这些方法时，我们所描述的就是*算法状态机*（algorithmic state machine）。也就是说，我们正在将FSM设计的方法和硬件实现技术应用于算法的求解。

虽然所有这些看起来似乎是高度结构化和数学化的（确实是这样），但通过对一个简单的时序过程实现其FSM，我们或许能对该过程获得一些感性认识。

图5-1是一个任意硬件算法的状态图。一个复位脉冲过后，系统初始化到状态000。暗灰箭头显示的是输入变量X=1时将要发生的状态迁移，而亮灰箭头显示的是X=0时将要发生的状态迁移。这样，如果系统处在状态011且

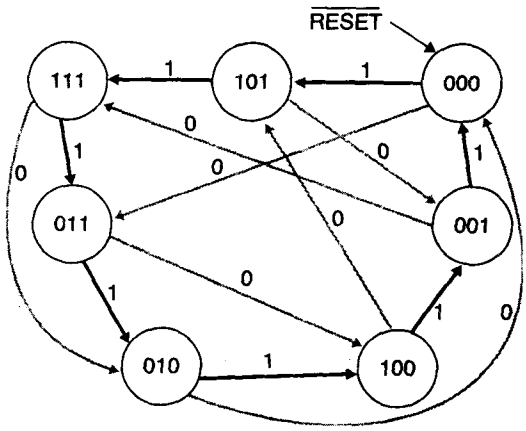


图5-1 一个任意时序过程的状态图

X=0，则次态将是100。但若X=1，则时钟到来后次态将成为010。

图5-2是这个算法的实现方法。如图所示，电路的输出是状态变量Aout、Bout和Cout。注意，状态000就意味着Aout=0，Bout=0，Cout=0。这些变量也反馈到真值表中，成为输入变量，用于确定下一个时钟到来后的状态迁移。在这个例子中，我们将把状态表的要求转移到真值表中，然后以组合逻辑来产生真值表的硬件实现。当组合逻辑和D触发器结合在一起时，我们就有了一个FSM，它复制了状态图。所以，我们将首先完成真值表，然后用K图技术对其进行化简。最后，我们将用硬件来实现这些逻辑方程。

97

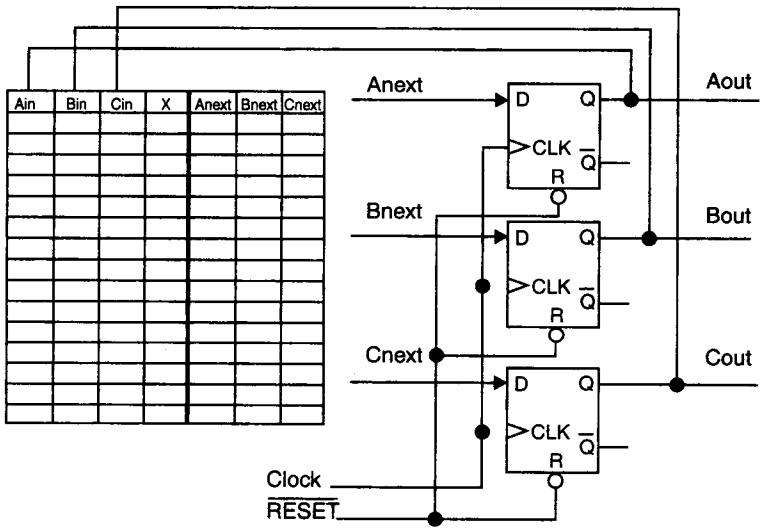


图5-2 图5-1时序过程的实现方法。真值表将被转换为组合逻辑

图5-3就是完成的真值表。首先要问的是："它有用吗？"让我们来看一下。根据图5-1，$\overline{\text{RESET}}$信号一过，系统就处于初始状态，从该状态出发，既可以在X=1时进入状态101，也可以在X=0时进入状态011。参见图5-3的真值表，我们看到当Aout，Bout，Cout=000时，若X=0，则Anext，Bnext，Cnext=011，若X=1，则Anext，Bnext，Cnext=101，这确实与状态图相符。

需要注意的重要一点是状态110不是时序过程的组成部分。在真值表中，我们看到这个状态表示为XXX，这是一种速记方式，表示该状态永远不会达到（如果电路工作正常）。我们能做的就是保留将X替换成1或0的权利，这种替换可能会简化K图。我们能这样做的原因是没有其他状态会将我们带进状态110，就像你在K图中看到的。

在开始化简过程之前，我们应该停留一下并回忆这样的事实，即真值表实际上是存储器内容的映像，它能够提供关于时序过程的电路设计。图5-4显示出对于逻辑设计的另一种视角。这里，真值表的输入Ain、Bin、Cin以及X成为存储器的4个地址输入，

Ain	Bin	Cin	X	Anext	Bnext	Cnext
0	0	0	0	0	1	1
1	0	0	0	1	0	1
0	1	0	0	0	0	0
1	1	0	0	X	X	X
0	0	1	0	1	1	1
1	0	1	0	0	0	1
0	1	1	0	1	0	0
1	1	1	0	0	1	0
0	0	0	1	1	0	1
1	0	0	1	0	0	1
0	1	0	1	1	0	0
1	1	0	1	X	X	X
0	0	1	1	0	0	0
1	0	1	1	1	1	1
0	1	1	1	0	1	0
1	1	1	1	0	1	1

图5-3 图5-1时序过程例子的真值表

而数据输出位DATA0到DATA2对应于真值表的输出变量或者过程的次态变量。

变量和简化方程的K图显示于图5-5。在图中，Anext的简化方程包括一个XOR项，该项的加入是为了进一步化简 $A*\overline{X}+\overline{A}*X$ 这样的项。是否可做进一步的化简是很容易被问到的问题。显然，采用布尔代数法则，对某些项可重新组合。然而，从门的复杂性角度看，将项组合起来可能会在实际上增加复杂性，因为在硬件实现中会引进额外的AND门，因此，对于最好的代数解会导致最小的硬件解这一点，并不总是很清楚的，会有一些另外的因素，如：可得到多少具有所需输入端数量的门，封装中电路其他地方所需要的额外门数，等等。

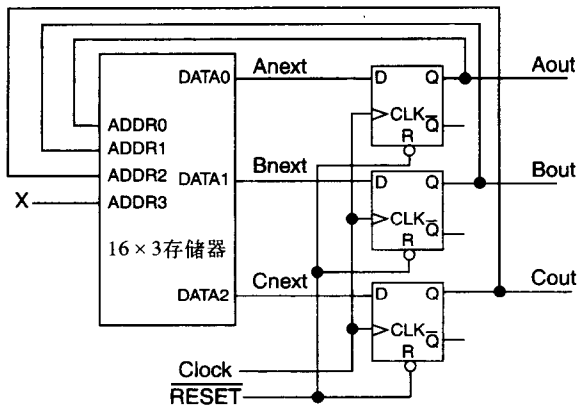


图5-4 时序过程的基于存储器的解决方案

98

硬件实现如图5-6所示。注意D触发器的 $\overline{Q}$ 输出是如何用作真值表的反相门的，不是简单地用NOT门来产生Anext、Bnext、Cnext及X的补，$\overline{Q}$ 输出就是便利的补信号来源。

图5-6的电路满足有限状态机的要求，即满足：

- 有一个包含7个状态的集合S，我们可连续地按序进行访问，两个状态之间的迁移发生于D触发器输入时钟的上升沿。
- 我们可用RESET脉冲确立一个初始状态000，初始状态是状态集合S的一部分。
- 系统有一个输入X，三个输出Aout、Bout及Cout。
- 次态函数 $d = d(s, i)$ 和输出函数 $f = f(s, i)$ 相同。在后面的例子中，我们将看到这个函数全然不同的情况。

99

显然，该例子纯粹是人为造出来的，而不是真正代表了实际的数字设计（虽然我确信你能在剩余电子产品商店买到一些零件，并借助简单的用法说明，就能建造起电路并详尽地观察状态序列）。

这很有趣！让我们看另外一个例子，这一次我们将做一个可能有实际用途的例子。我们考虑有一个串行的位流正在进入系统，设想我们需要检测每次出现3个或更多连续1的出现。我一时不知道为什么要做这件事，但若给一些时间，我确信你能想出有这种电路的需求。图5-7就是该电路

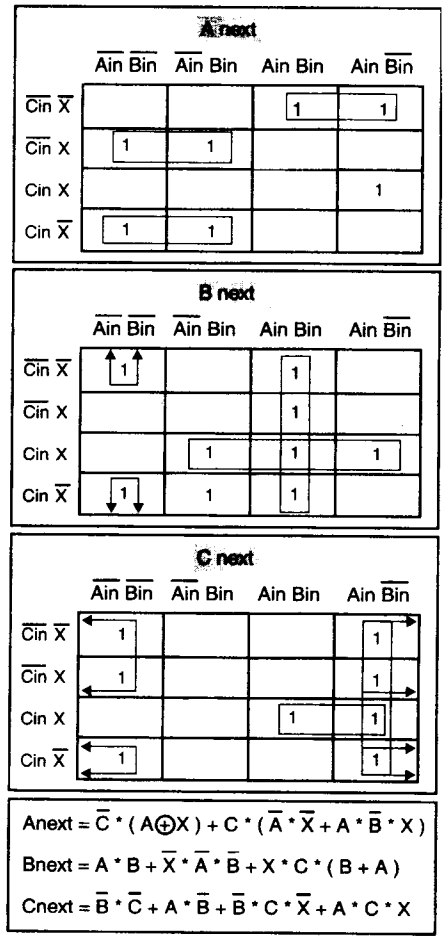


图5-5 变量Anext、Bnext及Cnext的K图和简化方程

的状态图。

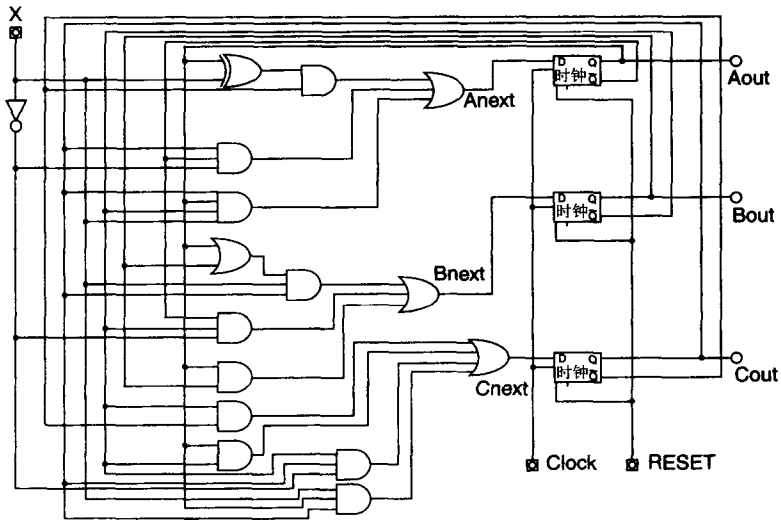


图5-6 图5-1中时序过程的硬件实现

这里我们有4个状态，分别是00、10、01和11。箭头表示在时钟沿发生的状态迁移。附于每个箭头上的是用前向斜线分隔的两个数字。第一个数字是在时钟上升沿时输入位的状态，第二个数字是输出信号T，当在位流中检测到三个或更多个连续1时T的值为TURE。

在本例中，我们所采用的时钟信号就像是系统用来同步数据位流的时钟。这样，在每一个时钟上升沿就出现一个新的数据位。每次出现一个0位，电路就返回到00状态，并等待1的到来。第一个1的到来使电路迁移到状态10，第二个1的到来使电路迁移到状态01。如果第二个到来的是0，则电路返回到状态00。但是，如果第三个1到来，电路就迁移到状态11，T输出位被置有效，显示已发现三个连续的1位。现在，只要到来的位是1，电路就保持在状态11，T也为1。一旦到来的是0，电路就返回到状态00。在该例子中，新加入了输出变量位T。

图5-8显示了电路的真值表、K图以及简化的逻辑方程。

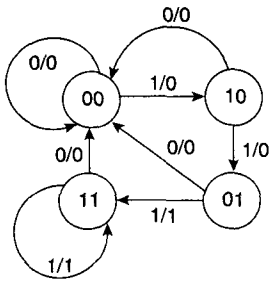


图5-7 一个用于在串行位流中检测3个或更多个1连续出现的电路的状态迁移图

A	B	X	Anext	Bnext	T
0	0	0	0	0	0
0	0	1	1	0	0
1	0	0	0	0	0
1	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	1	1
1	1	0	0	0	0
1	1	1	1	1	1

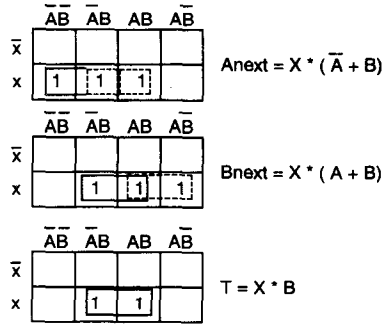


图5-8 图5-7状态图的真值表、K图以及简化的逻辑方程

电路图显示于图5-9中。该问题也是一个FSM的例子，其中输出位T是输入和组合逻辑的

100 函数。事实上，状态位A和B对输出T未起任何作用。显然，次态函数$d = d(s, i)$和输出函数$f = f(s, i)$不同。

另一个有趣的观察是FSM实际上实现了算法。用C、C++、或Java写一个算法，用软件模仿这个硬件电路的实现，这可能不会花费你很长时间。事实上，我们已经实现了一个*算法状态机*（algorithmic state machine, ASM）。

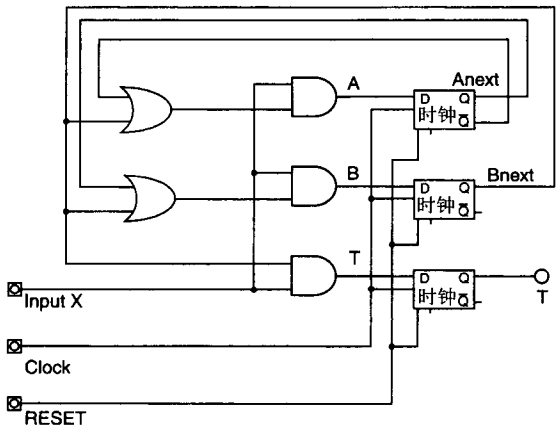


图5-9 图5-7状态迁移图的硬件实现

算法状态机的例子：十字路口交通控制器

让我们遍历一下（至少一次）一个实际问题的细节，这个问题是可用算法状态机[2,3]解决的一个典型问题。在这个例子中，我们要考虑的是为一个典型的十字路口的交通灯设计控制电路。图5-10显示出十字路口的示意图，其中：

- W = 西面车辆的信号灯
- E = 东面车辆的信号灯
- N = 北面车辆的信号灯
- S = 南面车辆的信号灯

由于南北交通是一个繁忙的具有4条车道的路，而东西方向的路只有2条车道，我们就可假设南北方向的路比东西方向的路有更高的交通流，所以，我们不想在十字路口停止南北交通，除非在东西方向有等待的车辆。例如，有多少次在十字路口面对红灯，你坐在车里，也许还有二三十辆其他的车，和你一起等待信号灯改变颜色，而同时十字路口的交叉方向却没有车辆通过呢？但我在这里脱离主题了……

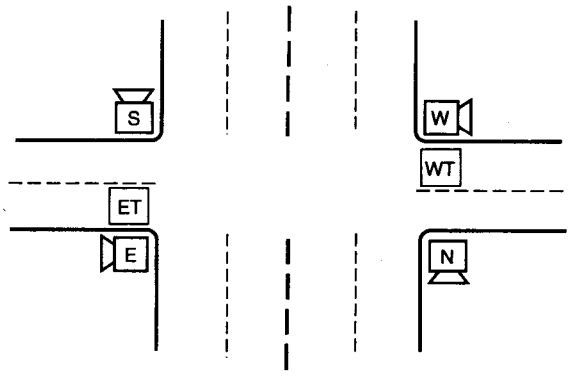


图5-10 具有4条南北方向车道和2条东西方向车道的交通十字路口。东西方向的路具有传感器来检测等待的车辆

这里给出算法的伪码描述：

```
While (1)
{
 timer = 20 seconds;
 NS = green;
 EW = red;
 while ( timer != 0)
   wait;
 if (EW sensor != 0 )
 {
```

101

```
        timer = 5 seconds;
    NS = yellow;
        while ( timer != 0)
         wait;
        timer = 20 seconds;
        NS = red;
        EW = green;
        while ( timer != 0)
         wait;
           timer = 5 seconds;
        EW = yellow;
        while ( timer != 0)
         wait;
    }
}
```

注意其中有一个新的不同，就是时间的不同。我们要在南北状态存在20秒，然后，如果有东西方向的车在等待，就要通过5秒黄灯后，将东西方向的绿灯打开20秒。

语句while (1) 经常在嵌入式系统中用于标识程序代码中永远运行的部分。一旦一个系统（比如打印机）被初始化，它就将永远运行（除非你关掉电源）。

让我们来看图5-11所示的状态迁移图。东西交通传感器的状态是算法的一个输入，它能更改系统的行为。我们可利用这样的事实来表示这个条件，即一旦状态机处于NS GRN/EW RED状态，就有两条可行路径。只要两个东西方向的传感器都没有检测到等待车辆，交通控制算法就将保持在这个状态。然而，如果在NS GRN时间间隔结束时检测到一个等待车辆，则状态机就迁移到状态NS YEL/EW RED。一旦处于此状态，算法在重新进入状态NS GRN/EW RED之前，就必须顺序进入NS RED/EW GRN和NS RED/EW YEL，不存在可以改变这个顺序的其他可能的输入。

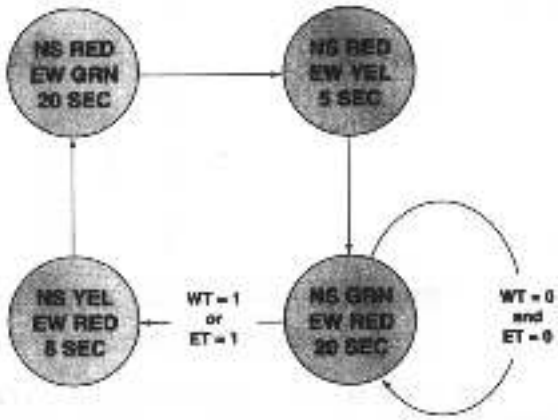


图5-11 交通十字路口的状态迁移图

再假设我们能照顾到时间要求，你能轻易地将此算法的伪代码改为C或C++语言的实际程序。当然，我们需要提供一个用于驱动信号灯和定时装置的I/O端口，但这些都是简单的实现细节。让我们暂时从算法状态机设计中转移一下，来介绍一个新的概念。图5-12为我们展示了表示成时序图的输出状态。这个时序图向我们展示了用于控制信号灯的输入信号的状态随时间的变化情况。注意，在图5-12中，横轴采用的是5秒的刻度，这仅仅是表明输出只在5秒刻度的时刻才变化。

图5-12的时序图代表了对系统的另外一个视角。事实上，这就是在逻辑分析仪的屏幕上所应看到的对交通灯驱动电路的控制器输出。这种视角是重要的，因为它为我们引入了另一个重要概念，就是将系统状态表示成一组状态向量（state vector）的思想。状态向量就是系统所处的输入和输出的所有可能状态组合。这样，如果你生成了所有这些向量的表，你就有了表示系统所有可能状态的另一种方式。图5-13就是图5-12时序图的状态向量集的表示。

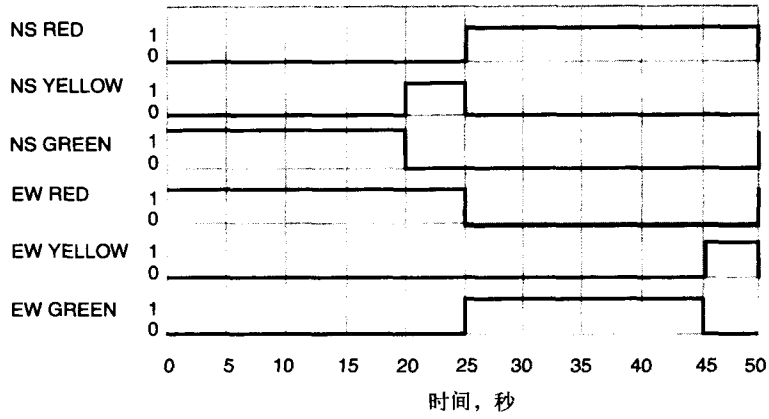
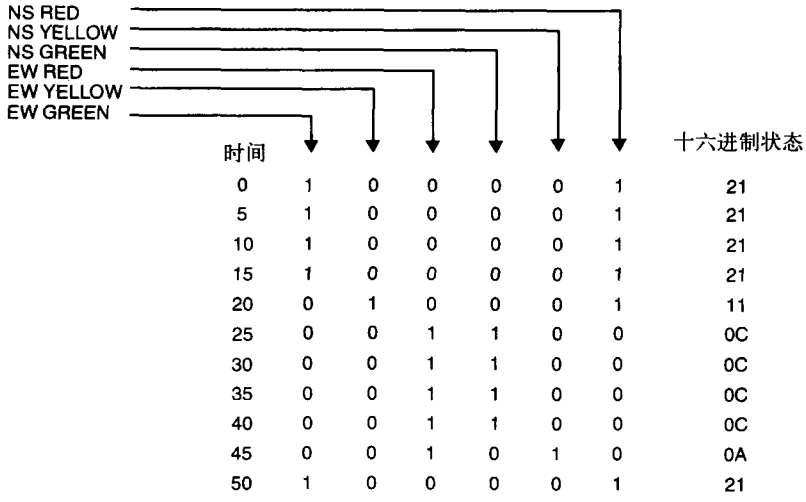


图5-12 交通信号灯运行序列表示为时序图



时间							十六进制状态
0	1	0	0	0	0	1	21
5	1	0	0	0	0	1	21
10	1	0	0	0	0	1	21
15	1	0	0	0	0	1	21
20	0	1	0	0	0	1	11
25	0	0	1	1	0	0	0C
30	0	0	1	1	0	0	0C
35	0	0	1	1	0	0	0C
40	0	0	1	1	0	0	0C
45	0	0	1	0	1	0	0A
50	1	0	0	0	0	1	21

图5-13 将时序图表示成一组状态向量。右侧的十六进制状态表示是6个灯信号的各个二进制状态总体的速记方式，因此，十六进制数字作为数字本身没有实际意义

在每个时钟迁移时，系统的状态可用二进制向量集合或十六进制向量集合来描述。这里我们必须小心一点，图5-13的每个十六进制状态表示不应与数字相混淆。它不是数字，而是一些单独位的汇集，表示成位的总体，换句话说，就是状态向量。

103

如你所见，十六进制表示在为交通控制器生成实际存储器映像方面是便利的，但是，更为准确和有意义的表示应该是用二进制向量表示，使用十六进制的好处仅仅是很便利。

现在，让我们开始对控制器的实际设计过程。作为第一次设计，我们将忽略东西交通传感器而仅考虑一个简单的模型，该模型在每个方向上给20秒的时间，黄灯给5秒时间。关注这些问题会更好一些。图5-14显示出转换成状态变量的简化算法，我们按每个状态的持续时间是5秒来重新绘制了该图。通过增加状态数量，即使若干状态“似乎”表示相同条件，我们也能将问题转化成相同时间间隔的问题。实际上，那4个表示20秒时间间隔的状态是不一样的，它们仅仅是有相同的输出函数。

现在，我们已经给了每个状态一个唯一的数字标识，从0000到1001，即0H到9H。这看起

来也许像是存储器地址，但我对此表示缄默，我要到最后再加以说明。让我们规定输出和每 104
一位的二进制权值，这仅是为方便起见，我们将强行这样做。

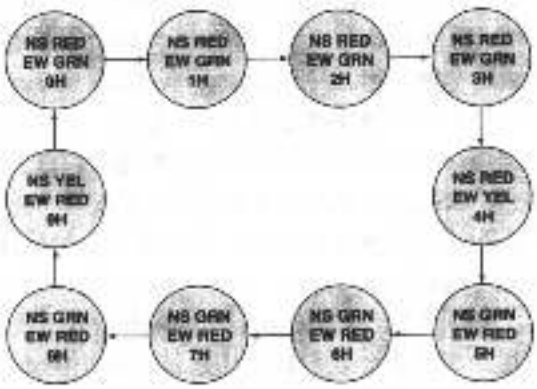


图5-14 将十字路口交通算法表示成一组状态变量，每个状态按序用数字标记，状态持续时间是5秒

考虑状态0000（0Hex）的6个输出：

- 南北红灯亮：NSR，即数据位0（D0）=1
- 南北黄灯灭：NSY，即数据位1（D1）=0
- 南北绿灯灭：NSG，即数据位2（D2）=0
- 东西红灯灭：EWG，即数据位3（D3）=0
- 东西黄灯灭：EWY，即数据位4（D4）=0
- 东西绿灯亮：EWG，即数据位5（D5）=1

这样，状态0000的状态变量就是100001，即状态0hex = 21hex。我们还用集总的数据位（data bit）表示状态变量的总和，而且，给控制信号灯的每个位一个总体的标识，使得我们可以将它们作为一个组而不是作为一个单独的标识来处理，这将是便利的。然而，我们应该记住每个输出变量实际上是与其他输出变量相互独立的，例如，在这个例子中，在状态值上加一个数字（21hex）并没有逻辑上的意义。下一步就是将我们的规范定义转换成真值表。

还记得我们早先见过的32位存储器阵列吗？我们再考虑一下，但这一次我们将输出数扩展到6个，以对应于交通信号的6个输出。这样就需要一个具有96个存储单元的存储器，排列成16×16。图5-15显示出这个项目的存储器（真值表）布局图。

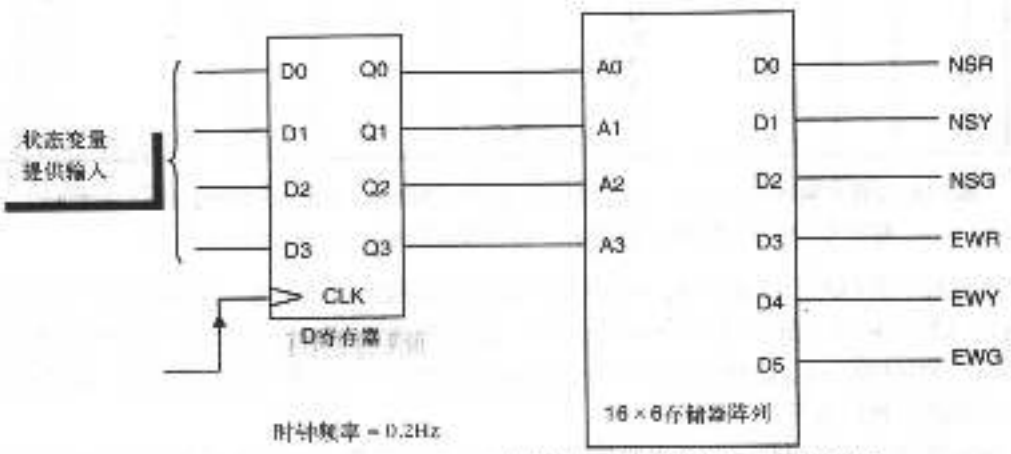


图5-15 具有存储寄存器来提供定序机制的状态机。该电路仍缺乏对系统进行状态转换的定序机制以及东西车辆有等待时改变算法的机制

时钟生成器在系统的其他地方。我们迄今一直在讨论的时钟频率是0.2Hz（每5秒一个周期），这个频率绝对是太慢了，但这是该例子所需要的速度。作为另一个选择，我们考虑将时
105 钟周期提高到1Hz的可能性，这会明显增加系统状态的数量，但我们可用其他方式来重新设计系统以解决这个问题。增加系统状态数量的一个优势是减少了系统的反应时间。

假设我们想将一个紧急车辆检测器（emergency vehicle detector）加入到十字路口中，这是一个光敏装置，用于检测紧急车辆顶部的闪光灯。如果一个警车正以每小时100公里（约每秒28米）的速度冲向十字路口，则在十字路口的所有灯变红以前，该警车能行驶约139米（在5秒的时钟周期内）。显然，我们的十字路口交通算法还有改进的余地。

D寄存器的输出值就是6个存储单元的地址，它决定了时钟脉冲上升沿到来之后，系统次态的数据。我们可以通过创建状态表（state table）来概括这些数据，这仅仅是将所有部分信息汇总起来的简便方式。记住我们仍需要加入一种机制来为系统状态之间的迁移进行定序。

图5-16就是初始设计的状态表。我们称之为初始设计是因为我们还需要加入定序机制，我们不久就会涉及到这个问题。参见图5-16，你就会明白我们已经泄露了秘密，状态的数字标识0000到1001确实就是即将成为设计真值表的ROM（read only memory）的地址。你可能已经从你PC的BIOS ROM那里熟悉了“ROM”这个术语，ROM就是一个存储器件，可预编程，即使电源关掉后数据也能保持，这正是我们的交通控制器所需要的。

状态					输出						
Q3	Q2	Q1	Q0	ROM地址	D5	D4	D3	D2	D1	D0	ROM内容
0	0	0	0	0H	1	0	0	0	0	1	21H
0	0	0	1	1H	1	0	0	0	0	1	21H
0	0	1	0	2H	1	0	0	0	0	1	21H
0	0	1	1	3H	1	0	0	0	0	1	21H
0	1	0	0	4H	0	1	0	0	0	1	11H
0	1	0	1	5H	0	0	1	1	0	0	0CH
0	1	1	0	6H	0	0	1	1	0	0	0CH
0	1	1	1	7H	0	0	1	1	0	0	0CH
1	0	0	0	8H	0	0	1	1	0	0	0CH
1	0	0	1	9H	0	0	1	0	1	0	0AH
1	0	1	0	AH	X	X	X	X	X	X	无关项
1	0	1	1	BH	X	X	X	X	X	X	无关项
1	1	0	0	CH	X	X	X	X	X	X	无关项
1	1	0	1	DH	X	X	X	X	X	X	无关项
1	1	1	0	EH	X	X	X	X	X	X	无关项
1	1	1	1	FH	X	X	X	X	X	X	无关项

图5-16 交通控制器的状态表，显示出状态变量（ROM地址）和每个ROM地址包含的数据。标识为“X”的值是指“无关项”，因为状态机永远不会进入到这些状态

注意我们的ROM中实际上有一些没有存储任何数据的存储单元，这些就是从地址0Ahex到地址0Fhex中的“无关项”（don't care），这是因为我们的设计总共只有10个有效状态
106 （00hex到地址09hex）。虽然存储器中有这些额外的状态，但只要我们的定序器不将我们引入到这些状态，我们就不会为此操心。

加入状态定序机制是一个相当简单的过程，我们要做的就是增加状态表中数据输出的数量，这些输出将要反馈到D存储寄存器。这样，我们就推广了状态机的概念，表明将所有输出

反馈到输入是不必要的，我们只需要将足够的输出反馈到输入，用来为状态机提供定序机制。我们称这些额外的输出为*激励输出*（excitation output），因为它们的功能是为状态机提供定序（激励）机制。图5-17显示出新的设计。

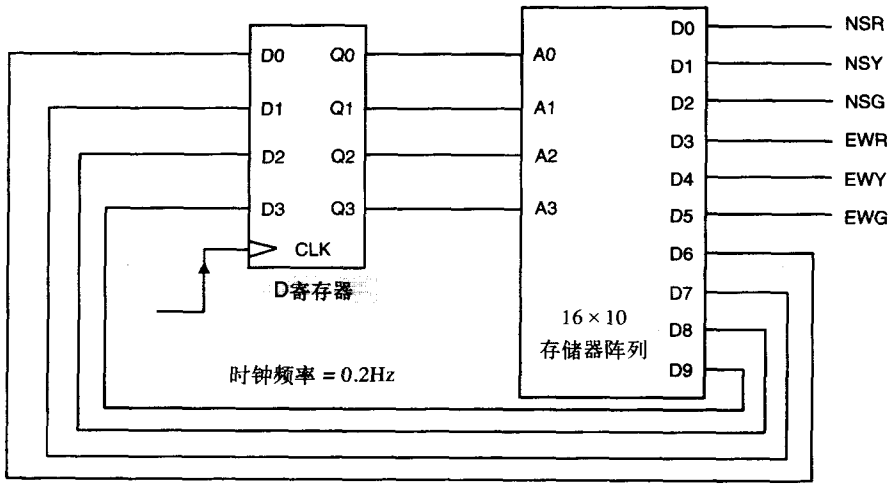


图5-17 状态机在ROM中加入了额外的数据位，用来提供对状态的定序机制

现在，就有必要扩展ROM中数据输出的位数，以便为定序系统提供对D寄存器的反馈。ROM仍有同样数量的存储地址，但我们已经加入4个输出位，这样，ROM的每个地址就有10个二进制位。图5-18显示出扩展的状态表。 107

	次态				输出						
现态ROM地址	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	ROM内容
0H	0	0	0	1	1	0	0	0	0	1	061H
1H	0	0	1	0	1	0	0	0	0	1	0A1H
2H	0	0	1	1	1	0	0	0	0	1	0E1H
3H	0	1	0	0	1	0	0	0	0	1	121H
4H	0	1	0	1	0	1	0	0	0	1	151H
5H	0	1	1	0	0	0	1	1	0	0	18CH
6H	0	1	1	1	0	0	1	1	0	0	1CCH
7H	1	0	0	0	0	0	1	1	0	0	20CH
8H	1	0	0	1	0	0	1	1	0	0	24CH
9H	0	0	0	0	0	0	1	0	1	0	00AH
AH	X	X	X	X	X	X	X	X	X	X	无关项
BH	X	X	X	X	X	X	X	X	X	X	无关项
CH	X	X	X	X	X	X	X	X	X	X	无关项
DH	X	X	X	X	X	X	X	X	X	X	无关项
EH	X	X	X	X	X	X	X	X	X	X	无关项
FH	X	X	X	X	X	X	X	X	X	X	无关项

图5-18 修改后的状态表，现在包含了额外的用于定序的数据位

最后，我们现在就有条件回过头来考虑为东西交通加入额外输入的问题了，我们迄今一直忽略这个问题。让我们更新一下记忆（已经过去很长时间了），回顾一下我们以前设计的算法。图5-19给出了我们曾设计和修改过的状态图，对每个状态我们可采用5秒时钟周期。

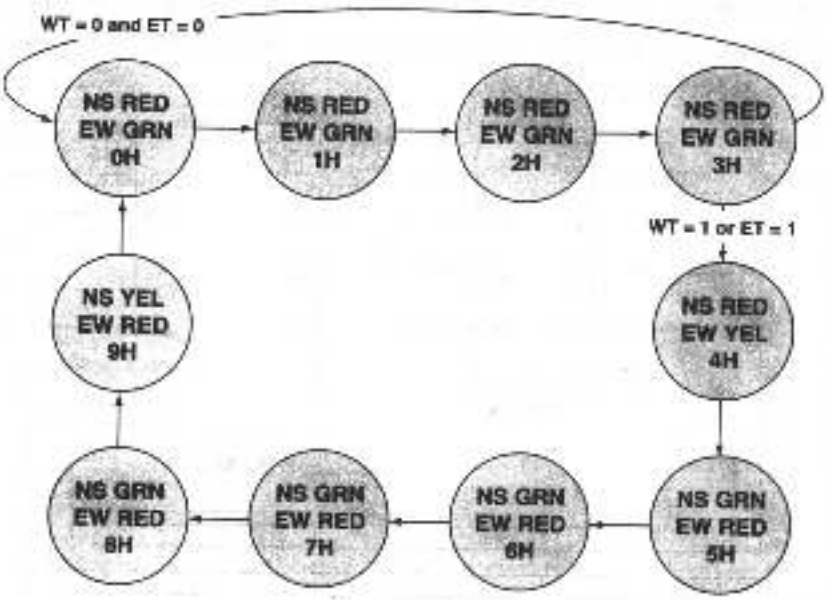


图5-19 状态机的状态迁移图，包含了为等待东西车辆而设的输入

这个新的加入使我们得到了如图5-20所示的状态机设计。

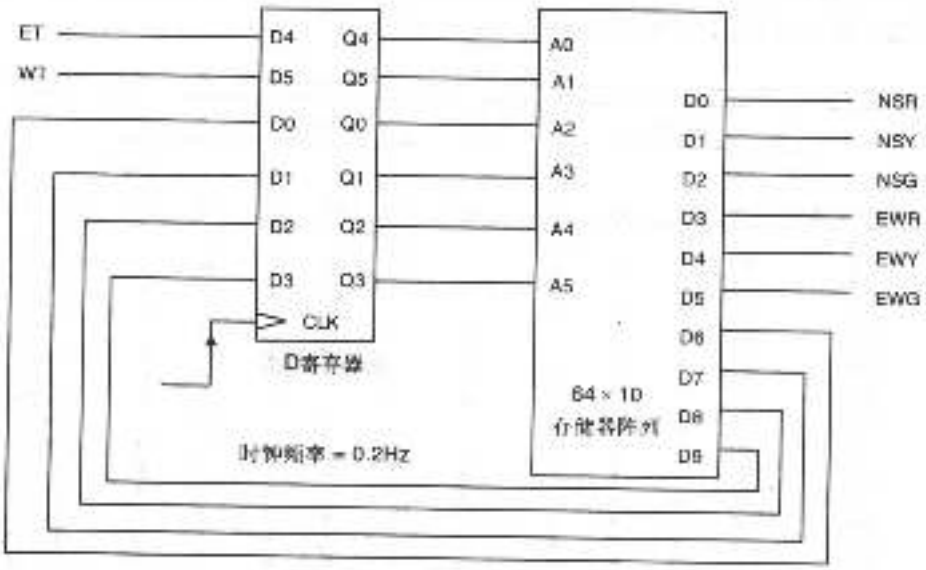


图5-20 具有独立输入的状态机

两个独立的输入ET和WT的加入是极其重要的，它们分别表示了在路内等待东面和西面车辆的传感器，这两个独立输入的条件能影响状态机的走序，这样，我们就向算法状态机中加入了全新的内容：响应输入数据的变化并改变状态流向的能力。你还认得这个新特征吗？你应该记得，它称为计算机。

独立输入的加入能够为状态机定义新的状态，状态机开始变得像一个决策支持设备。当然，它不是真正做决策，因为我们已经对它精确地进行了预编程，规定它如何响应新的数据。

计算机的状态机要比这个简单的状态机复杂几千倍，但你正逐渐了解它。我们称微处理器的状态转换表（ROM）为*微代码*（microcode），是微代码给了计算机以个性。计算机的*指令集体系结构*（instruction set architecture, ISA）是由微代码所定义的。

108

当计算机从存储器中取出一条指令时，代表指令的位模式就是输入到微代码ROM的外部输入组合，这个模式确立了解释和执行该指令所要求的步骤序列。你曾想知道为什么会遇到臭名昭著的*蓝屏死机*（Blue Screen of Death）吗？有时（不总是）这可能是由一个漂浮不定的指针使程序从存储器中的非法区域取指令而引起的。这里的位模式就可能只是数据值或垃圾，而不是合法指令的位模式。由于微代码不能解释这个模式，处理器就通知操作系统，程序自己停止。无论如何，让我们还是回到手头的问题上。

让我们重新审视状态ROM。我们已经又加入了两个输入，因此可能的地址组合的数量就上升到了64个。现在，我们的状态ROM就有64个位置，每个位置有10位宽的数据，因此我们的简单交通系统控制器已有640个存储单元。这就不难理解为什么现代计算机的微代码引擎有数百万的存储器单元了。图5-21是ROM的状态表，包含了所有细节。

状态				WT	ET	次态				输出					
A5	A4	A3	A2	A1	A0	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1
0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	1
0	0	0	0	1	0	0	0	0	1	1	0	0	0	0	1
0	0	0	0	1	1	0	0	0	1	1	0	0	0	0	1
0	0	0	1	0	0	0	0	1	0	1	0	0	0	0	1
0	0	0	1	0	1	0	0	1	0	1	0	0	0	0	1
0	0	0	1	1	0	0	0	1	0	1	0	0	0	0	1
0	0	0	1	1	1	0	0	1	0	1	0	0	0	0	1
0	0	1	0	0	0	0	0	1	1	1	0	0	0	0	1
0	0	1	0	0	1	0	0	1	1	1	0	0	0	0	1
0	0	1	0	1	0	0	0	1	1	1	0	0	0	0	1
0	0	1	0	1	1	0	0	1	1	1	0	0	0	0	1
0	0	1	1	0	0	0	0	0	0	1	0	0	0	0	1
0	0	1	1	0	1	0	1	0	0	1	0	0	0	0	1
0	0	1	1	1	0	0	1	0	0	1	0	0	0	0	1
0	0	1	1	1	1	0	1	0	0	1	0	0	0	0	1
0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	1
0	1	0	0	0	1	0	1	0	1	0	1	0	0	0	1
0	1	0	0	1	0	0	1	0	1	0	1	0	0	0	1
0	1	0	0	1	1	0	1	0	1	0	1	0	0	0	1
0	1	0	1	0	0	0	1	1	0	0	0	1	1	0	0
0	1	0	1	0	1	0	1	1	0	0	0	1	1	0	0
0	1	0	1	1	0	0	1	1	0	0	0	1	1	0	0
0	1	0	1	1	1	0	1	1	0	0	0	1	1	0	0
0	1	1	0	0	0	0	1	1	1	0	0	1	1	0	0
0	1	1	0	0	1	0	1	1	1	0	0	1	1	0	0
0	1	1	0	1	0	0	1	1	1	0	0	1	1	0	0
0	1	1	0	1	1	0	1	1	1	0	0	1	1	0	0

图5-21 交通控制器ROM的状态表。注意从状态1010开始的无用状态有一个09H的数据输出，对应于南北红灯和东西红灯打开。这对于系统进入非法状态这样的不可靠事件提供了一个安全特性

0	1	1	1	0	0	1	0	0	0	0	0	1	1	0	0
0	1	1	1	0	1	1	0	0	0	0	0	1	1	0	0
0	1	1	1	1	0	1	0	0	0	0	0	1	1	0	0
0	1	1	1	1	1	1	0	0	0	0	0	1	1	0	0
1	0	0	0	0	0	1	0	0	1	0	0	1	1	0	0
1	0	0	0	0	1	1	0	0	1	0	0	1	1	0	0
0	1	1	1	1	0	1	0	0	0	0	0	1	1	0	0
0	1	1	1	1	1	1	0	0	0	0	0	1	1	0	0
1	0	0	0	0	0	1	0	0	1	0	0	1	1	0	0
1	0	0	0	0	1	1	0	0	1	0	0	1	1	0	0
1	0	0	0	1	0	1	0	0	1	0	0	1	1	0	0
1	0	0	0	1	1	1	0	0	1	0	0	1	1	0	0
1	0	0	1	0	0	0	0	0	0	0	0	1	0	1	0
1	0	0	1	0	1	0	0	0	0	0	0	1	0	1	0
1	0	0	1	1	0	0	0	0	0	0	0	1	0	1	0
1	0	0	1	1	1	0	0	0	0	0	0	1	0	1	0
1	0	1	0	0	0	0	0	0	0	0	0	1	0	0	1
1	0	1	0	0	1	0	0	0	0	0	0	1	0	0	1
1	0	1	0	1	0	0	0	0	0	0	0	1	0	0	1
1	0	1	0	1	1	0	0	0	0	0	0	1	0	0	1
1	0	1	1	0	0	0	0	0	0	0	0	1	0	0	1
1	0	1	1	0	1	0	0	0	0	0	0	1	0	0	1
1	0	1	1	1	0	0	0	0	0	0	0	1	0	0	1
1	0	1	1	1	1	0	0	0	0	0	0	1	0	0	1
1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	1
1	1	0	0	0	1	0	0	0	0	0	0	1	0	0	1
1	1	0	0	1	0	0	0	0	0	0	0	1	0	0	1
1	1	0	0	1	1	0	0	0	0	0	0	1	0	0	1
1	1	0	1	0	0	0	0	0	0	0	0	1	0	0	1
1	1	0	1	0	1	0	0	0	0	0	0	1	0	0	1
1	1	0	1	1	0	0	0	0	0	0	0	1	0	0	1
1	1	0	1	1	1	0	0	0	0	0	0	1	0	0	1
1	1	1	0	0	0	0	0	0	0	0	0	1	0	0	1
1	1	1	0	0	1	0	0	0	0	0	0	1	0	0	1
1	1	1	0	1	0	0	0	0	0	0	0	1	0	0	1
1	1	1	0	1	1	0	0	0	0	0	0	1	0	0	1
1	1	1	1	0	0	0	0	0	0	0	0	1	0	0	1
1	1	1	1	0	1	0	0	0	0	0	0	1	0	0	1
1	1	1	1	1	0	0	0	0	0	0	0	1	0	0	1
1	1	1	1	1	1	0	0	0	0	0	0	1	0	0	1

图5-21（续）

当然，过程的下一个逻辑步骤就是将防备紧急车辆的过程放回到我们的设计中。这将在设计中再加入一个输入，使得ROM有7个地址位。为什么到此就停止了呢？我们还可以为南

北车辆加入一个左转的信号，这将至少再加入一个输出（NS转向灯）和一个输入（NS转向车辆等待，NST）。如果我们继续为这种更复杂的情况设计算法，我们会看到系统的状态数增加到超过16，所以我们需要增加一个输出，并反馈到输入用来定序，这就使ROM有总共11个输出和8个输入。如果你在下雨的周末没有任何事可做，为什么不为这样一个真实的交通十字路口制作一个状态表呢？我把这作为一个可选择的、但又是有教益的习题留给有积极性的学生。

109 ~ 111

图5-22显示出了电路的重要组件和数据通路，并且没有陷入使电路实际运行所要考虑的很多细节。振荡器模块产生0.2Hz的时钟流，我们需要它为状态机定序。八进制的D触发器是一个集成电路构件块，它在具有一个共同时钟输入的单一封装中包含8个D触发器。因此，所有8个触发器总是在同一个时钟上升沿触发。

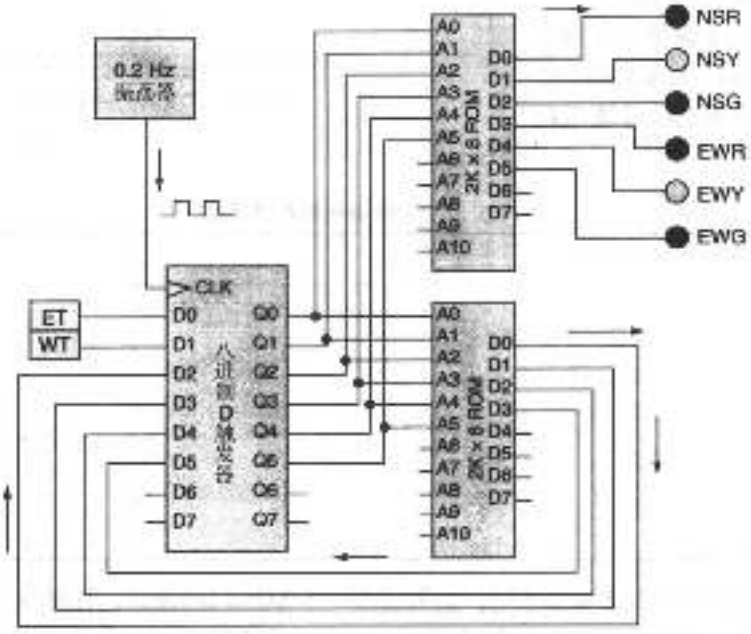


图5-22 交通十字路口控制器的简化示意图

另外两个器件是交通控制器的心脏，它们包含了我们设计的用来实现算法状态机的ROM代码。采用两个独立ROM的原因是我们需要10个输出，而大多数商业上可得到的ROM只有8个输出，所以我们需要将设计在两个ROM之间进行划分。为了方便，上面ROM的6个输出用于控制信号灯，下面ROM的4个输出用于定序。你可以看到下面ROM的4个输出又返回成为D触发器的输入。这就清楚地示意出定序函数$d(x,i)$和输出函数$f(s,i)$的不同。

112

该例子中所使用的ROM每个的容量是16K位，以2K × 8组织。由于我们在每个ROM中总共只使用了64个地址，你可能会认为在计算机体系结构的历史上这不是最伟大的工程解决方案。然而，存储器相对来说不昂贵，而且在这个特殊的例子中，一个具有16K存储单元的ROM并不比一个小得多的ROM更昂贵。

在继续讨论之前，让我们总结一下我们学过的有关状态机的知识，并用这些原理解释计算机是如何工作的。图5-23显示出数字计算机的基本定序机制，你可以清楚地看到Mealy机的元件。为方便，我们省略了计算机内部很多的其他东西，它们只是连接到微定序器ROM的输出端。微定序器的外部输入包括机器语言指令以及能影响程序执行流的所有其他输入。

假如我们想在计算机的内部将两个数字相加。

运用新学到的状态机设计的知识，我们该如何做这件事呢？假设我们想将两个4位的数相加。在计算机中将两个数相加的电路为*算术和逻辑单元*（arithmetic and logic unit, ALU）中。实际上，你对生成ALU的基本构建块的数字设计知识已经知道得足够多了。基本的加法电路有3个输入（A、B及Carry In）和2个输出（SUM和Carry Out）。我们可将基本的2-位加法器电路的行为总结并显示在表5-1的真值表中。

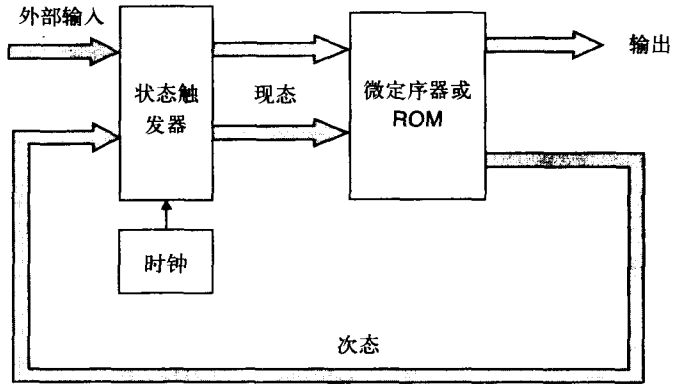


图5-23 广义的时序数字机

表5-1 2-位加法器电路的真值表

A	B	Cin	Sum	Cout
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

一旦设计了这个电路并布置了门，如果回顾一下XOR门的设计，你将很快发现你能极大地简化这个电路。正如你在真值表中所看到的，加法器的每一级只是简单地将两个输入相加再加上前面一级的进位，并生成一个和以及一个到下一级的进位。如果我们想将两个32位数（整数）加在一起，我们就要有32个这样的加法器，见图5-24。

113

既然知道了如何将数字相加，让我们再看一下状态机如何对操作进行定序，使数字实际地加在一起，得到结果并存起来。图5-25就是这个操作序列的示意图。

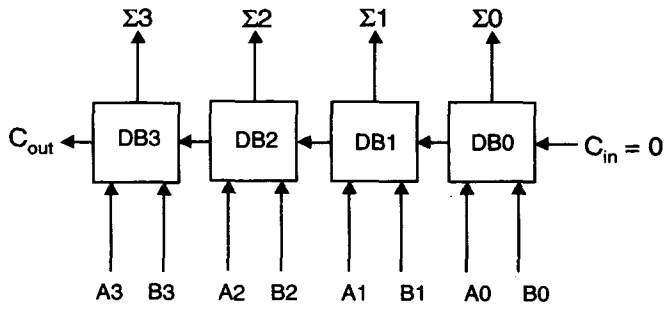


图5-24 将两个数字相加

在这个特殊的例子中，完成加法需要5个时钟周期。注意我们已经略去了一些预备的步骤，例如先要对加法指令进行译码才能转到这里相加，还有就是如何先将要加的数字放到寄存器中。我们显然在隐藏一些材料。我们将在后面的一章里全面和详细地讨论这个问题，现在我们就只关注概念。我们按如下步骤进行这个过程：

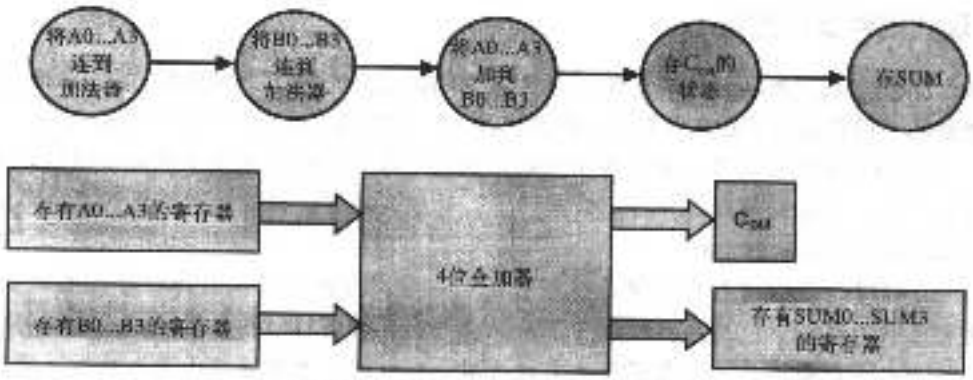


图5-25 两个4位数相加的状态机序列

1. 将存有第一个操作数A0...A3的存储寄存器连接到加法器电路，使得加法器的A输入端能看到第一个操作数。

2. 然后将存有第二个操作数B0...B3的存储寄存器连接到加法器。

3. 一段适当的传播延迟之后（一旦B输入已经稳定），结果就出现在加法器的输出端。

4. 进位输出位的状态存储在适当的寄存器中。你很快就会看到，我们还要存一些其他的结果。例如，如果相加的结果为零，我们也要存储这个信息。

5. 将相加的和存储在一个输出寄存器中以备将来使用。

我们总结一下我们学过的有关状态机的知识：

- 次态取决于：现态，任何存储寄存器的输入，以及任何返回到存储寄存器输入的输出。
- D型寄存器由D型触发器组成，其作用是在时钟边沿上对状态机进行同步。时钟的转换时间必须远快于状态机发生变化所需时间，这样才能同步状态机，防止电路失控。
- ASM的流图或状态图就是正在实现的算法。
- 基于ROM的状态表是实现用于设计的真值表的一种方式。我们也可以遵循设计步骤，利用真值表为每个输出变量建立积（最小项）和（sum of product）逻辑方程。一旦有了真值表，我们就能生成卡诺图，通过卡诺图就能生成简化的电路门设计。 114

算法状态机是几乎所有目前的计算引擎的基础。现代计算机的指令集体系结构由其内部的微代码ROM所决定，它实现了状态机。处理器如何按序遍历一系列状态由如下因素决定：

- 执行的指令
- 内部寄存器的内容
- 算术和逻辑操作的结果
- 所采用的存储器访问模式
- 异步的内部或外部事件（中断，RESET，错误条件）

图5-26显示出示意图。

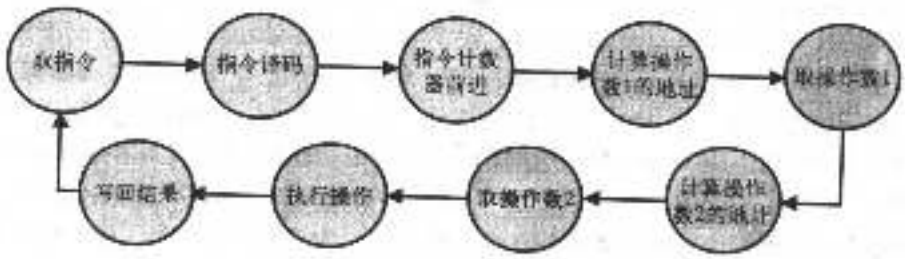


图5-26 一个简单微处理器中的状态序列

5.2 现代硬件设计方法

Mead和Conway[4]为超大规模集成（VLSI）电路的设计提出了一种新的方法。基于对复杂集成电路的一种自顶向下的开发方法，他们描述了一个结构化的设计系统。他们写到：

> VLSI系统的结构化设计方法可开始于在一个层次结构中将本章所提出的概念合并在一起。然后，设计以一种“自顶向下”的方式进行，但设计师要对相继的低层的层次结构有完全的了解。
>
> 开始时，我们将数字处理系统规划成寄存器到寄存器的数据传输路径的组合，由有限状态机控制。然后，对所有子系统模块的几何形状、相对尺寸以及互连拓扑关系一起进行规划，使得所有模块安适地融合在一起，并使随机互连布线所花费的空间和时间最小化……
>
> 如果我们将每个层次的每个模块都看作是一个有限状态机或由有限状态机控制
115 的数据通路，就显现出对这种嵌套的模块所组成的系统的特别一致的观点。在最低层，诸如堆栈和寄存器单元等元件可看作是状态机，具有一个反馈项（输出）、两个外部输入（控制信号）以及一个1-位状态寄存器。这些基本的状态机以结构化的方式聚集在一起，在层次结构的上一层形成一个状态机的组成部分或者由状态机控制的数据通路。

随后[5]，他们继续描述了设计实际集成电路的方法，该方法是通过建立计算机辅助设计（computer-aided design tool, CAD）工具来实现的，该工具的作用就像一个宏汇编器对于软件的作用。如果基本的电路单元可表示成一些标准的构件块或单元，则与宏编译器的类比，某种符号版图语言就能从这些标准单元生成IC版图。他们说：

> 用户定义符号（宏）来描述基本系统单元的版图。这些符号实例的位置和方位是用语言描述的，就像描述带有合适参数的函数一样。这些符号描述接着可能被以某种方式进行机械化处理，就类似于宏汇编语言程序的扩展，所产生的系统版图的中间格式描述就类似于用来产生输出文件的机器代码。

Mead和Conway所描述的是你应该很熟悉的两个概念。一个概念是硬件设计结构化的思想。你可能将这称为“软件工程”，该方法开始于对首先是需求文档然后是形式化规范描述的建立。从那开始，通过自上而下的分解，定义各种功能组件（模块），在开发过程中软件设计不断进展。

如果你已经学习过现代编译器如何将一种高级语言转换为中间语言（汇编语言）再转换为机器代码，则第二个概念对你来说就很熟悉。这里，低级机器代码用标准的低级单元表示，这些单元表示晶体管级电路以及这些单元间的互连。

作者正在描述的就是我们现今所说的*硅编译器*（Silicon Compiler），设计现代集成电路的过程就称为*硅编译*（silicon compilation）。硬件电路设计者采用的是高级开发语言，或者是Verilog或者是VHDL。VHDL是在IEEE标准1076-2001——IEEE标准VHDL参考手册[6]中正式定义的。Verilog被看作是“另一种”硬件描述语言。Verilog起初是专有的模拟语言，但后来转给IEEE并发布为IEEE标准1364-1995——IEEE基于Verilog硬件描述语言的标准描述语言[7]。

1981年，Silicon Compliers公司成立，其目的是将设计描述从实现中脱离出来[8]。他们的主张是关心设计实现，使得设计者的关注点在算法上。为了做到这一点，他们建造能转化成定制集成电路模块的库和代码模块，与设计专家手工设计电路的方法相同。

1985年，CMOS工艺在商业上已经可以应用，这个突破驱动了ASIC产业的发展。如我们所见，CMOS门是接近于理想的开关，采用它，就实现了用硅编译技术设计集成电路的商业可行性。在CMOS工艺出现之前，为了使集成电路工作正常，还需要非常多的手工设计工作。 116
根据Cheng的描述[9]："功耗下降，噪声裕度增加，1就是1，0就是0（即容易区分）"。

伴随ASIC设计在商业上的应用，出现了硬件描述语言（hardware description language，HDL）工具的标准化，也极大地促进了采用硅编译设计硬件的逻辑综合技术的发展。Silicon Compliers公司与Silicon Design Labs在1987年合并成立了Silicon Compiler Systems（SCS）公司。1990年，SCS被在电子设计自动化工具方面领先的供应商Mentor Graphics公司所收购。

你可以在图5-27中看到硅编译的影响。

如果将图5-27的数据画在半对数图纸上，你将会得到一个令人惊奇的图形，它非常接近直线，从而显著地证实了摩尔定律。如果考察设计一个集成电路所需的设计人员数，比如具有29 000个晶体管的8086对比于具有42 000 000个晶体管的奔腾4处理器，我们一定会得出这样的结论，就是Intel不可能有一个比8086设计队伍大约大2000倍的奔腾4设计队伍。我们必然得出的结论是每个设计者将晶体管（或CMOS门）置入硅片上的效率必然是导致这种增长的原因，因此，就像C++将程序员从汇编语言编程问题中解放出来一样，硅编译也将硬件设计者从集成电路设计工艺的低级问题中解放出来。

Intel微处理器

	出现时间	晶体管数
4004	1971	2,250
8008	1972	2,500
8080	1974	5,000
8086	1978	29,000
286	1982	120,000
386™ processor	1985	275,000
486™ DX processor	1989	1,180,000
Pentium® processor	1993	3,100,000
Pentium II processor	1997	7,500,000
Pentium III processor	1999	24,000,000
Pentium 4 processor	2000	42,000,000

来源：Intel公司

图5-27 Intel系列微处理器中晶体管数量的增长。由Cheng[9]提供

作为熟悉高级语言的人，你应该能快速适应这种语言的结构。事实上，相对来说，你更像是在写软件，而不是在设计硬件。几家商业软件公司已达成一致，而且目前已有将分开的硬件和软件的设计过程紧密集成为高层次系统设计的商业工具。

然而，软件和硬件开发过程还存在一个差异，就是修改缺陷的成本。作为软件开发者，你知道重新编译和重新建立软件映像需要几天的时间。这个过程涉及很好地确立将代码的新版本分发到消费者的方式。为了发出代码更新信息，你可能要向一个FTP地址邮寄一个新版本。

硬件开发者就没有这种奢侈。即使设计过程正在接近完成，硬件设计者仍然面临着这样的现实，就是硬件设计是一个复杂的、昂贵的和耗时的过程，很少或没有发生错误的余地。硬件返工的成本可轻易地耗掉500 000美元，而且还要拖延3个月或更长的时间。这样，即使有硅编译技术，硬件设计工具也是面向测试和设计模拟而高度结构化的。通常开发一个新的 117
ASIC所需时间要在实际设计时间和用模拟对硬件进行彻底测试所需时间之间进行平均分配。当然，硬件设计者总是有一个B计划备用。对硬件缺陷的普遍解决方案在现在和可预见的未来都是："在软件中改正"（fix it in software）。

在我们离开这个话题之前，实际地看一些VHDL代码并和我们已经熟悉的语言相比较是有益处的。下面是一个加法电路的例子（来自Ashenden[10]）。我们首先需要声明一个实体（entity），这和声明一个变量类似。

```
entity adder is
    port ( a : in word ;
           b : in word ;
           sum : out word ) ;
end entity adder ;
```

其次，我们需要描述模块的内部操作。换句话说，我们需要编写用来描述变量行为的语句。这将在代码的结构体（architectural body）中实现。

```
architecture abstract of adder is
begin
    add_a_b : process ( a,b ) is
    begin
        sum <= a + b ;
    end process add_a_b ;
end architecture abstract;
```

结构体命名为abstract，它包含一个进程add_a_b，用来描述实体的操作。就像C++中的模
118 板函数一样，该进程假设加法操作符“+”在以前已经针对数据类型“word”定义过。

我们可以容易地描绘出这个代码片段如何能编译成一个由1-位加法器基本元件组成的电路，如图5-24所示的那样，但不同的是，这里它配置成了加16位或更大的变量，称为“word”。

总结

- 第5章开始于对算法的一般概念的讨论，即一个算法既可以是基于一序列软件步骤的解，也可以是基于硬件的解。
- 我们考察了有限状态机的定义，并了解了如何将状态机定义成Mealy机或Moore机。
- 我们了解了如何将当前状态的反馈与D型触发器相结合，使我们能够同步和稳定状态迁移。
- 我们考察了如何利用我们建立真值表的知识来构造状态机的硬件实现。
- 通过构造一个交通十字路口控制器的基于存储器的实现，我们走过了将算法建造成状态机的过程。
- 最后，我们了解了如何用硬件描述语言来设计硬件系统。

参考文献

[1] Thomas Richard McCalla, *Digital Logic and Computer Design*, ISBN 0-6752-1170-0, Merrill, New York, 1992, p. 265.

[2] Carver Mead and Lynn Conway, *Introduction to VLSI Systems*, ISBN 0-2010-4358-0, Addison-Wesley Publishing Company, Reading, MA, 1980, pp. 85–87.

[3] Claude A. Wiatrowski and Charles H. House, *Logic Circuits and Microcomputer Systems*, ISBN 0-0707-0090-7, McGraw-Hill Book Company, New York, 1980, pp. 1–11.

[4] Carver Mead and Lynn Conway, *Introduction to VLSI Systems*, ISBN 0-2010-4358-0, Addison-Wesley Publishing Company, Reading, MA, 1980, p. 89.

[5] Carver Mead and Lynn Conway, *Introduction to VLSI Systems*, ISBN 0-2010-4358-0, Addison-Wesley Publishing Company, Reading, MA, 1980, p. 98.

[6] Peter J. Ashenden, *The Designer's Guide to VHDL, Second Edition*, ISBN 1-55860-674-2, Morgan-Kaufmann Publishers, San Francisco, 2002, p. 671.

[7] Ashenden, *ibid.*, p. 677.

[8] Ed Cheng, http://bwrc.eecs.berkeley.edu/Seminars/Cheng%20-%209.27.02/Silicon%20Compilation,%2021%20years%20young.pdf.

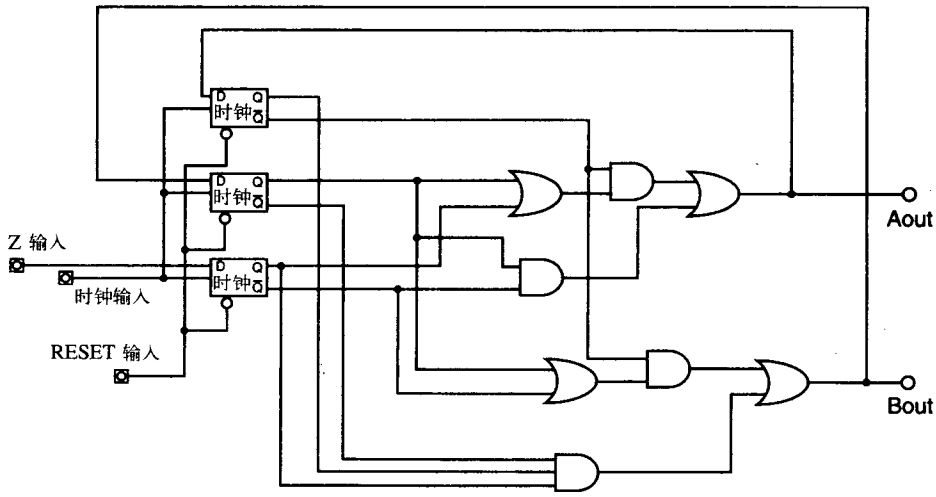
119

[9] Cheng, *ibid.*

[10] Peter J. Ashenden, *The Designer's Guide to VHDL, Second Edition*, ISBN 1-55860-674-2, Morgan-Kaufmann Publishers, San Francisco, 2002, pp. 108–111.

习题

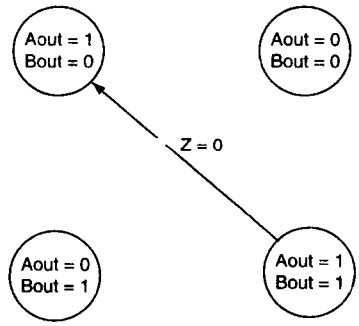
1. 下图是一个由三个D型触发器和一些逻辑门组成的状态机。



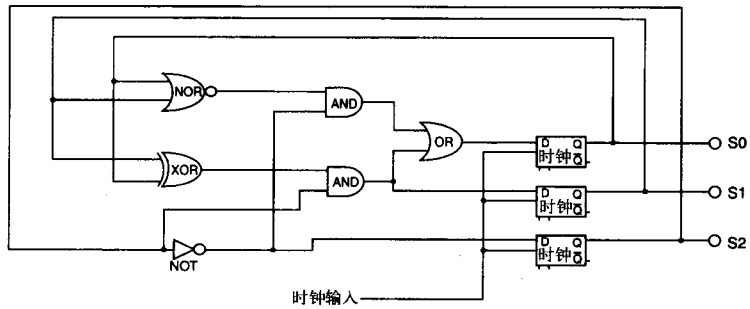
假设RESET到三个触发器的输入已经发生过作用了。完成下面所示的状态机的真值表。

Ain	Bin	Z	Aout	Bout
0	0	0		
1	0	0		
0	1	0		
1	1	0		
0	0	1		
1	0	1		
0	1	1		
1	1	1		

画出该状态机的完整的状态迁移图。为简化，不必考虑RESET信号的影响。提示：从给出的部分状态迁移图开始。



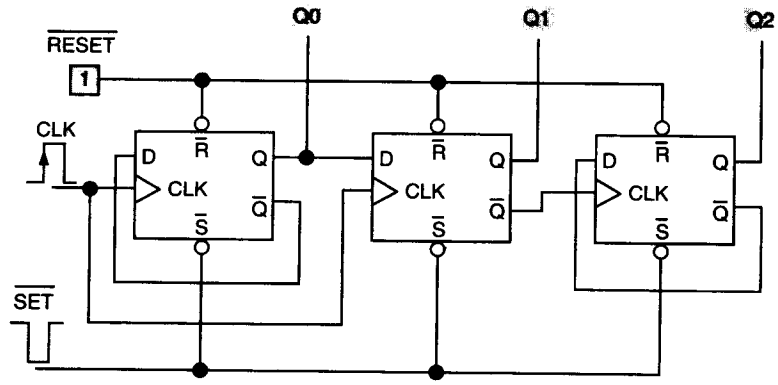
2. 下图所示的是某个任意算法的状态机。



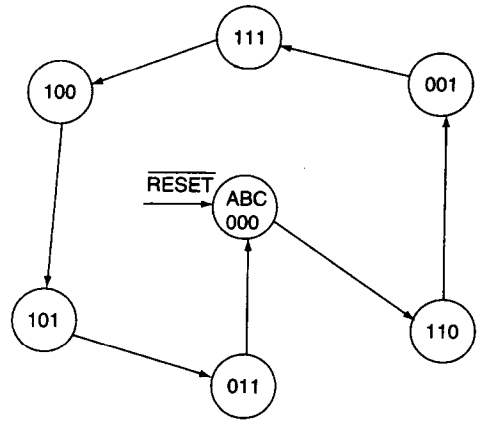
假设状态机已接收到RESET脉冲，并处于状态000（S0 = 0, S1 = 0, S2 = 0）。完成状态迁移图的绘制。

120

3. 下图显示的电路包含三个D型触发器。黑点指示的是相互有物理连接的导线。$\overline{\text{RESET}}$ 输入端始终连接到逻辑1，故永远不起作用。在接收到时钟脉冲之前，$\overline{\text{SET}}$ 输入端接收到一个负脉冲，确立了电路的初始条件。画出该电路的状态迁移图。



4. 下图所示是一个硬件算法的状态图。

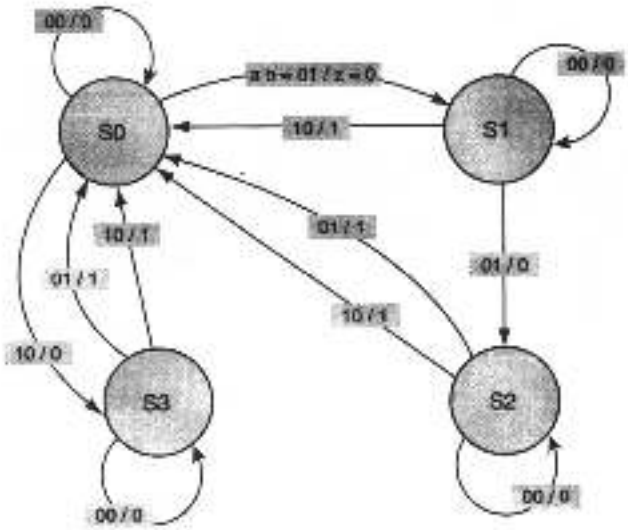


一个reset脉冲过后，系统初始化到状态000。图示的是算法的实现方法。电路的输出就

是所示的状态变量ABC。这些变量还反馈到真值表成为输入变量，在下一时钟到来时进行下一个状态迁移。画出对应于这个硬件算法的真值表。化简真值表，然后画出用等价的门电路代替真值表的全部电路。提示：记住，有一个可能的状态在算法中不出现，你可以将其加入到真值表中，这可能有助于你得到更简化的结果。

5. 假设你是欢乐时光防风暴外门和售卖机公司的首席设计师。你接到一个任务，要重新设计公司最畅销的售卖机，这是一个简单的墙模型，安装在自助商店的休息室中。由于有电源可供使用，你决定用漂亮的新式电子设备取代老式的机械模型。

你决定采用状态机设计形式，状态机如下图所示。



状态机可在4个状态S0~S3中循环。另外，它有两个外部输入a和b，一个输出z。输入a表示四分之一美元放入硬币槽中，输入b表示一角（10美分）硬币放入硬币槽中。商品的价格是30美分，如果投入的钱超过30美分，也不找零钱，输出z=1就引起商品的交付。 121

4个状态定义如下：

- S0 = 静止状态，没有钱投入
- S1 = 投入10美分
- S2 = 投入20美分
- S3 = 投入25美分

假设顾客投入10美分（ab=01），这将导致进入状态S1。由于钱数不够，所以这个状态迁移没有导致商品交付（z=0）。这时就有两种可能：如果又投入10美分，就进入状态S2，还是没有商品交付；但如果又投入25美分，则交付商品并返回到状态S0。

如果先投入25美分硬币，则进入状态S3，但没有商品交付。再投入硬币就导致商品交付并返回到状态S0。

不可能同时投两枚硬币，所以输入ab=11是不允许的。此外，状态迁移标记00/0只表示在时钟脉冲到来时没有发生任何事情。完成真值表，用K图化简逻辑方程，并为该电路画出门设计。

6. 设计一个状态机电路，用于检测位串1001的出现。你的答案应包括状态迁移图、真值表、

K-图，最后还要给出电路实现。

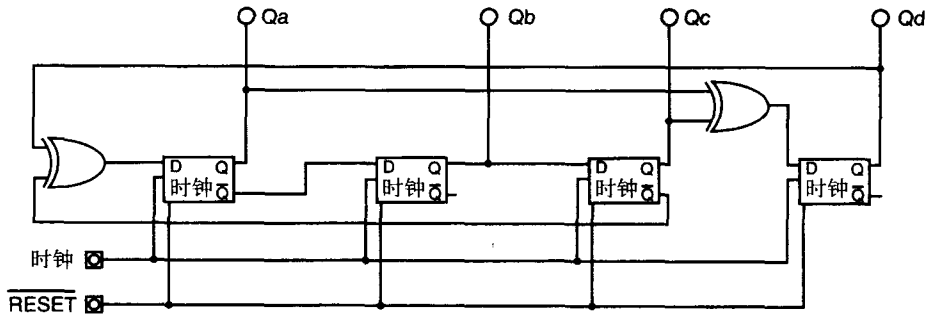
7. 下图所示电路由4个D型触发器和两个异或门组成。注意没有采用SET输入，只有RESET可使用。做一个真值表，显示：

   a. $\overline{\text{RESET}}$ 脉冲后，输出Q_0、Q_1、Q_2和Q_3的状态。

   b. 16个时钟脉冲后输出的状态。

   c. 输出再回到相同模式，这个循环需要多少时钟脉冲？

   d. 将电路重新设计成有限状态机。为简化，使每个模式在8个时钟脉冲后再循环。为每个状态和次态建立真值表，化简真值表，画出门电路。



122

第 6 章　总线组织和存储器设计

学习目标

- 理解总线组织的必要性；
- 采用三态逻辑原理设计面向总线的系统；
- 为现代微处理器设计存储器译码电路；
- 采用现代存储器电路的地址、数据及控制的I/O引脚设计任何宽度和深度的存储系统。

6.1　总线组织

在第2章中，我们看到逻辑门的输出连接到其他逻辑门的输入，这是一个一般的准则。输出必须连到输入，你可以同时将一个输出连接到多个输入，使得当该输出改变自身状态时，所有连接到它的输入同时看到这个变化（当然，这仍然在光速的约束范围内），这称为1到N（one to N）电路配置。

然而，你不能将多个输入连到一起，除非它们是被连接到一个输出的，这是因为需要有某种信号来将这些输入驱动到1或0。若没有一个输出来驱动，这些输入将趋于漂浮不定，频繁地在1、0之间变化，并在计算机中产生噪声信号。一般来说，所有不使用的输入都"捆绑"到了地或电源电压（V_{cc}）。稍后你将会看到，若我们试图将两个或更多的输出连到一起，也存在类似的问题。如果一个处于逻辑1状态的输出连接到一个处于逻辑0状态的输出，你认为会发生什么情况呢？我们会最终得到平均数0.5吗？为了看看会发生什么，你可以拿一个1.5V的电池（如AA或AAA电池），用一段导线迅速将电池的正端和负端短路。如果接触良好，你将会看到火花，甚至一股烟。一般来说，计算机设计者不喜欢看到哪怕一点来自计算机内部的烟，因此，希望这个实验使你确信，将输出连在一起不是一个好主意。

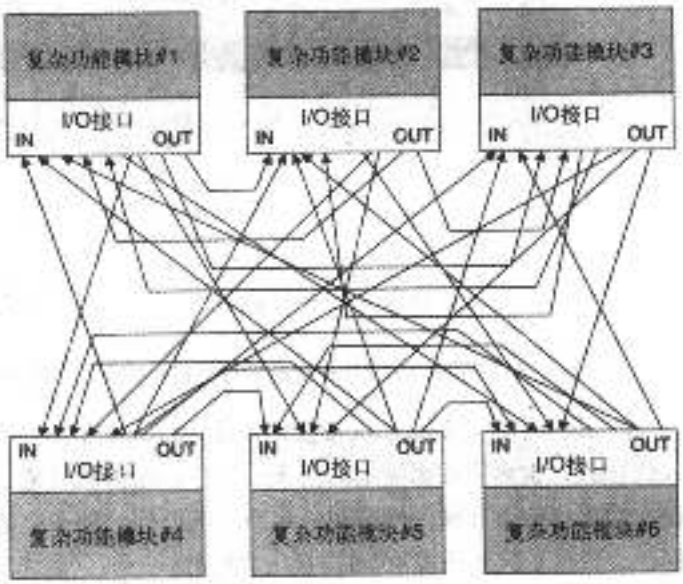


图6-1　基于点对点连线的计算机组织。该图显示出在一个典型的计算机系统中，在6个不同的内部组件之间连接一个数据位所需的信号线数（注：在DVD光盘中有该图的一个彩色版本）

从以上讨论你可能猜测到了，将两个输出连在一起类似于电路的短路，事实上，我们可能损坏了电路元件，因为每个元件都在试图强迫改变另一个，就像我第一次结婚，但那是另外一个故事了。

图6-1向我们展示了计算机设

123 计的基本困境，有很多的线遍布各处。

图6-1是要显示出：需要这些如“鼠巢”般的导线才能将6个计算机电路模块互连。在这个例子中，我们不关心这些功能模块到底是什么，我们只对它们如何连接到一起感兴趣。而且，互连只显示出一个数据位。对于一个真正的计算机系统，我们需要将这些迷宫般的线乘以32。而且，现代计算机有远多于6个的功能模块需要互连。

每个功能模块将其输出信号连接到其他5个模块的输入，线上的箭头指示的是哪个模块在发送信号。在每个功能模块内部，必然有某种输入/输出(I/O)接口和控制电路，使得计算机能够对发送模块和接收模块进行同步。因为通常在任意时间点只有一个模块发送，也只有一个模块接收。因此，所有模块都在输出1和0，但是在一个时间点，某模块应该只接收一个模块的输出，这意味着I/O接口电路必须用某种方式来决定接收哪个功能模块的输出，忽略哪些模块的输出。

124 图6-2显示出更详细一些的I/O组织。每个模块的输出以“1到5”结构同时驱动其他模块的输入。在输入这一侧，每个模块必须有用来决定在每个时间点接收哪个输出的逻辑，这样就需要决策逻辑，称为多路选择器（multiplexer，MUX），用来做“5到1”的选择，使得正确的数据被读到输入端，并传递到复杂功能模块中。这很复杂，如果拿不出比较好的解决方案，我们就会被淹没在这些线中。如果我们能有像图6-3那样连接的电路就比较理想了。

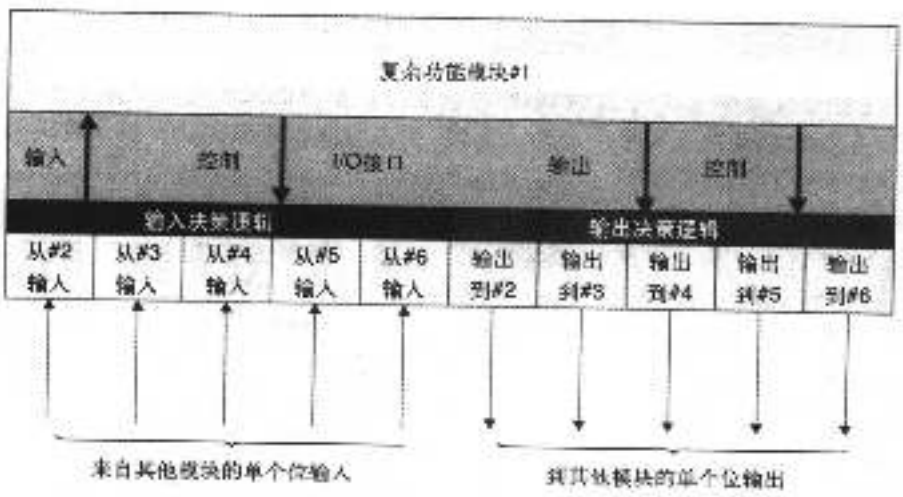


图6-2 单个位通信协议所需电路的详细模块视图，用来实现6个功能模块之间的点对点连线

在图6-3中，我们试图大幅度地简化设计。每个功能模块只引出一条线，它既承担数据输入的任务，也承担数据输出的任务。记住，这仍然只是一位数据。这也许简单化了，但你若说它因为将输出和输入捆绑到一起而不能工作，那么你是正确的，这正是我们早先考虑过的问题。我们正试图从模块#1向模块#6发送一个1（灰虚线箭头）。所有其他输出是0（黑虚线箭头），所以数据不能发送。我们需要以某种方式管理通过电路的交通流（注意到巧妙的夹钳技术）。我们解决这个问题的方式就是将数据通路组织成总线（bus），然后利用我们在第2章学习到的第4个基本门结构，即三态缓冲器。

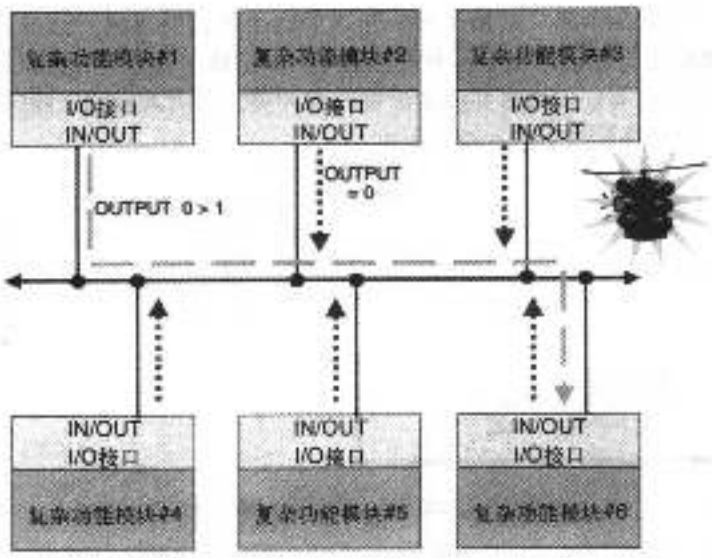


图6-3 采用总线协议来互连功能模块。由于输入和输出连在一起，就产生了问题

总线是作为简化计算机系统内部的组织和数据流的一种方式而发明出来的。我们使用总线就能使很多设备同时连到同一个数据通路上。一个总线就是一组类似的信号。我们马上就会看到总线的更多细节，但现在让我们首先解决如何将这些恼人的信号连在一起，而又不至于烧坏这个问题。

在大多数计算机中有三条主要的总线，称为地址总线（address bus）、数据总线（data bus）及状态总线（status bus）。我们将在下一章更详细地讨论这些总线，但现在让我们再一次考虑输出问题。图6-4显示出了这种困境，左边与门的两个输入都是1，所以其输出是1，即5V。

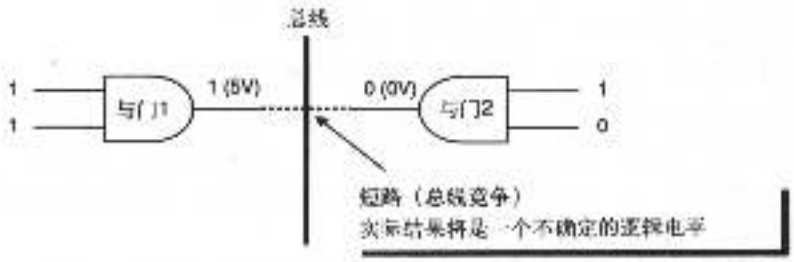


图6-4 试图将两个逻辑门的输出捆绑到同一个数据总线会产生的问题。我们将在总线上看到什么样的逻辑电平

右边逻辑门的两个输入中一个是1一个是0，所以其输出是0（即0V），总线上的结果信号就是不确定的。为解决这个问题，我们需要采取某种方法将逻辑函数从接口中分离出来，并加入到总线中去。要做到这一点，可通过将连接到总线的逻辑设备电路划分成两部分：逻辑函数和总线接口单元。在图6-1和图6-2中我们看到了接口电路的必要，然而，在本例中总线接口单元不是一个多路选择器（用来对来到设备的各种输入信号进行选择）。接口逻辑要比这简单得多，它是一个简单的开关，用来将输出和总线连接或断开。 125

图6-5给出了图示。到总线的接口是一个电子开关。正如你已知道的，该电子开关是一个三态缓冲器。总线控制信号可快速激活开关，使得输出连接到总线或从总线上断开。如果除

了一个输出以外，其他各个模块的输出都断开，则这个连接到总线的输出就会“驱动总线”达到高逻辑电平或低逻辑电平。由于这个输出是唯一的“讲话者”，所有其他的器件都是“听者”（从1到N），所以没有任何来自其他输出器件的冲突。所有其他输出通过其电子开关与总线保持断开状态，对总线信号的状态没有影响。

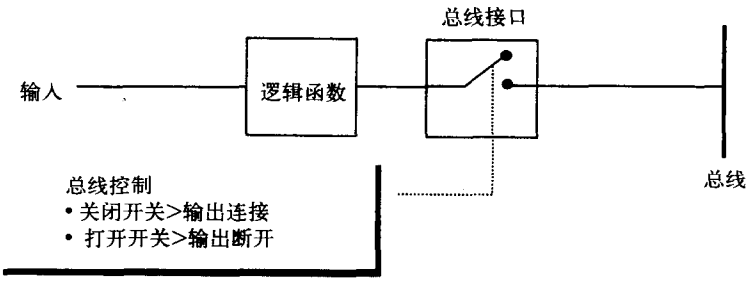


图6-5 总线接口逻辑的机械图表示

如你所知，用三态缓冲器并不是真要打破门到总线的电连续性，我们只是简单地对电路的一小部分的导电性做了一个快速的改变，而该部分电路居于门的逻辑输出和总线信号之间。在图6-5中，我们将电路的连接/断开部分画成像是房间墙上的灯开关，以此来简化这个电路魔术。但请记住，我们可不能像从高阻抗（无信号流）到低阻抗（连接到总线）那样快速而干净利落地打开和关闭一个机械开关。

通常总线控制信号是低有效的。不管是什么逻辑状态（1或0）要放到总线上，我们都必须首先将那个门、功能模块、存储单元等的总线控制置为低，以此将门连接到总线上，这样逻辑器件的输出就可连接到总线。这个信号有若干个名字，有时它叫输出使能（output enable，$\overline{OE}$），也叫芯片使能（chip enable，$\overline{CE}$）或者芯片选择（chip select，$\overline{CS}$）。这些信号并不都是一样的，我们很快就会看到，它们将在器件内部执行不同的任务，但它们都有一个共同的特性，就是将器件从总线上断开使其不能输出。

回过头来看图2-6的三态逻辑门，我们看到当 $\overline{OE}$ 即总线控制信号为低时，电路总线接口
126 （BUS I/F）部分的输出就遵循输入的值。当 $\overline{CS}$ 信号为高时，BUS I/F的输出就变为高阻抗，有效地将该门从电路中断开。还有最后一点需要强调，电路的三态门不改变门、存储器器件或其他任何与它连接的器件的逻辑状态。它的唯一功能就是将门的输出与总线隔离开来，使得另一个器件可以控制总线和发送信号。

现在让我们看看如何使单个位的总线成为真正数据总线的一部分。参见图6-6，在这里我们看到总线实际上是由8条类似信号所构成的组。在该例中，它代表8个数据位。如果我们拆开录像机并观察里面的微处理器，我们可能会在里面发现一个8位的微处理器。数据总线的位数指的是我们在一个操作中能取出的数的大小。回忆可知，8位能使我们表示从0到255（即2^8）之间的数。

总线仍由8个单独信号线组成，这8个信号线是互相电隔离的。记住这一点很重要，原因是为了使我们的制图简化，我们通常都将总线画成一条线，而不是画成8条、16条或32条线。总线的每条线用相应的数据位标记，从DB0到DB7，其中DB0表示数字2^0或1，DB7表示数字2^7或128。

虚线内就是连接到总线的器件，器件内的每个数据位都通过三态缓冲器与总线上的相应数据线相连。注意 $\overline{CE}$ 信号是如何同时连接到所有的三态缓冲器的：通过将 $\overline{CE}$ 置为低，就将所有8个单独的数据位与数据总线连接起来了。还要注意这8个三态缓冲器（画成三角形，旁

边有圈）决不会改变数据值。它们只是将功能模块内部的8个数据位（DB0...DB7）上的信号连接到功能模块外相应的数据线上。不要混淆三态缓冲器和非门，非门是将输入信号取反，而三态缓冲器则是控制是否允许信号通过缓冲器传播。

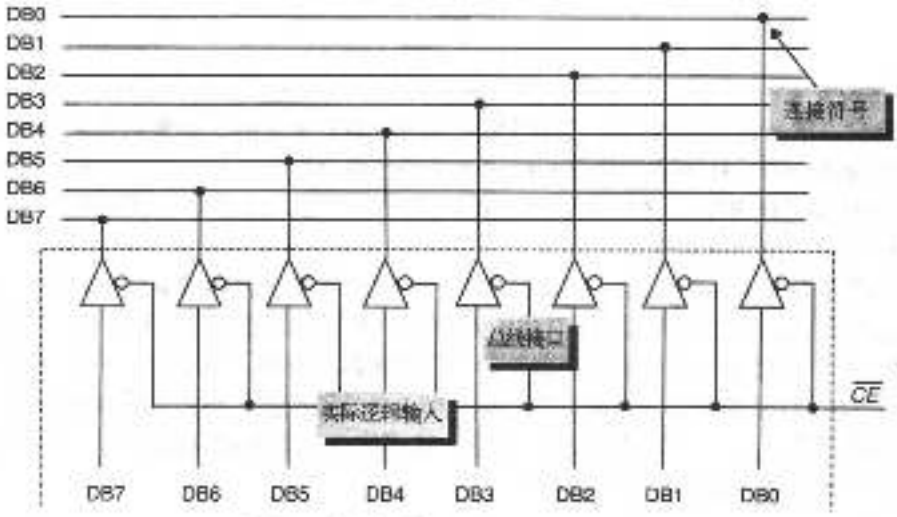


图6-6 8位宽数据总线的三态总线组织

图6-7更进一步地引申了这个概念，这里我们有4个32位存储寄存器连接到32位数据总线。 127
注意这里是画出数据位D0...D31来显示这些单独的位是怎样变成32位总线的。图6-7还显示了一个新的逻辑元件，就是标记为“2:4 地址译码器”的模块。回忆可知，我们需要用某种方式来决定哪一个器件可将其数据放到总线上，因为每次只能有一个输出。译码器电路做的正是这件事情。设想一下，有两个来自电路中某个其他部件的信号A0和A1，用来确定在给定的时间图中显示的4个32位寄存器中的哪个连接到总线。这两个标记为A0和A1的“地址”输入位给了我们4种可能的组合，这样，我们就能生成某种相对简单的逻辑门电路，用来将输入A0和A1的组合转化为4个可能的输出，这就是芯片选择位 $\overline{CS0}$、$\overline{CS1}$、$\overline{CS2}$及 $\overline{CS3}$。这个电路设计与你已经习惯的电路设计有点不同，因为我们想让“真”的输出变低，而不是变高。这个结果是由三态门逻辑通常是低输入信号有效这个事实所造成的。

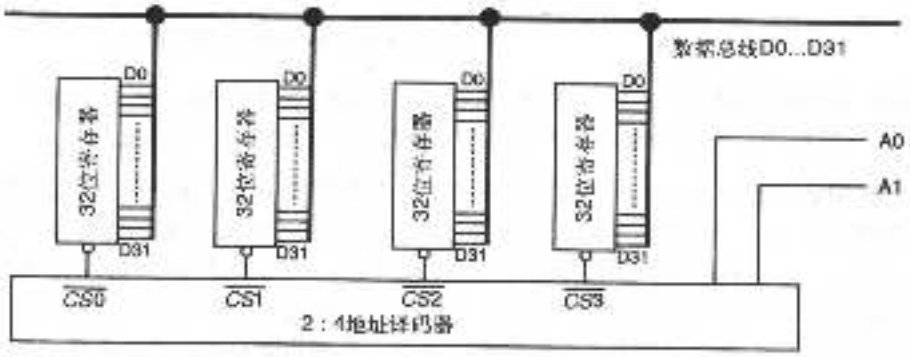


图6-7 4个32位存储寄存器连接到总线。2：4译码器接收2个输入变量A0和A1，并从4个输出 $\overline{CS0}$ ... $\overline{CS3}$ 中只选择一个使之有效

表6-1就是图6-7中2：4译码器电路的真值表。

表6-1 2：4译码器电路的真值表

A0	A1	CS0	CS1	CS2	CS3
0	0	0	1	1	1
1	0	1	0	1	1
0	1	1	1	0	1
1	1	1	1	1	0

如同我们以前所讨论过的，在三态门的输入端画圈是常见的，这意味着该信号是低电平有效的。正像NAND门和NOR门中反相门输出处的圈指出信号反相一样，门输入处的圈指出它是低电压有效（TURE）。这就又回到了我们以前的讨论，就是关于1和0哪个代表TRUE、哪个代表FALSE是相当随意的。

在继续考察存储器组织之前，让我们重温一下前面章节中讲到的算法自动机。通过本节对总线概念的介绍，我们对理解基于总线的系统操作如何由状态机来控制就有了更好的基础。图6-8为我们展示了计算机控制系统一部分的一个简化的示意图，所示的每个寄存器（即寄存器A、寄存器B、临时寄存器以及输出寄存器）都有单独的控制，用来在时钟输入的上升沿将数据读入到寄存器，还有一个输出使能（$\overline{OE}$）信号使寄存器能将数据放到公共数据总线上。该总线连接了计算机内部的所有功能元件。

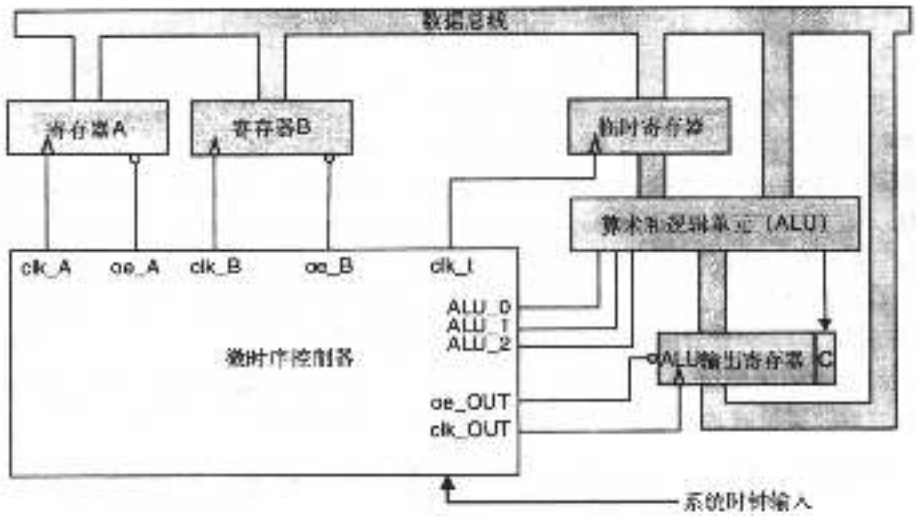


图6-8 计算机控制系统的一部分示意图。寄存器采用三态门和总线互连。微时序控制器必须为每个寄存器的输出使能（$\overline{OE}$）正确地定序，使得不存在总线竞争

128 为了将数据放入寄存器A，我们首先必须使另一个数据源有效（也许就是存储器），并将数据放到数据总线上。当数据稳定后，时钟信号clk_A经过一个从低到高的转换并将数据存到寄存器。记住寄存器A只是一组具有共同时钟的D触发器。现在假设我们想将寄存器A和寄存器B的内容加起来。算术和逻辑单元（ALU）需要两个输入源来进行数字相加。因为ALU本身是一个异步门设计，所以它要有一个临时寄存器和一个输出寄存器。也就是说，因为没有D寄存器在那里同步其行为，所以如果输入改变，输出也将立即改变。

图6-9显示出用于控制微时序控制器行为的一部分真值表。为了将两个数相加，比如将A和B的内容相加，我们可能要发出汇编语言指令：

ADD B, A

clock	clk_A	oe_A	clk_B	oe_B	clk_T	clk_OUT	oe_OUT	ALU_0	ALU_1	ALU_2		clk_A	oe_A	clk_B	oe_B	clk_T	clk_OUT	oe_OUT	ALU_0	ALU_1	ALU_2
1	0	1	0	1	0	0	1	0	0	0		1	1	0	1	0	0	1	0	0	0
2	1	1	0	1	0	0	1	0	0	0		0	1	1	1	0	0	1	0	0	0
3	0	1	1	1	0	0	1	0	0	0		0	0	0	1	0	0	1	0	0	0
4	0	0	0	1	0	0	1	0	0	0		0	1	0	0	1	0	1	0	0	0
5	0	1	0	0	1	0	1	0	0	0		0	1	0	0	0	1	1	0	0	0
6	0	1	0	0	0	1	1	0	0	0		0	1	0	1	0	0	0	0	0	0
7	0	1	0	1	0	0	0	0	0	0		1	1	0	1	0	0	1	0	0	0
8	1	1	0	1	0	0	1	0	0	0		0	1	0	1	0	0	1	0	0	0

图6-9 图6-8的微时序控制器的真值表，显示出每个时钟脉冲前后的逻辑电平变化。表的左侧显示的是时钟脉冲之前输出的状态，而右侧显示的是时钟脉冲之后输出的状态。灰色区域是为了突出显示每个时钟脉冲发生时的变化

这个指令告诉计算机将寄存器A和寄存器B的内容相加，并将结果放回到寄存器A中。如果该加法操作产生了进位，则将其放到进位位的位置，就显示在配属于ALU输出寄存器的"C"位上。这样，我们就明白了需要若干步骤将这些数相加。我们用文字来描述这个过程：

1. 将数据从存储器拷贝到寄存器A中。这个数据将是操作数#2。
2. 将操作数#1从存储器拷贝到寄存器B中。
3. 将数据从寄存器A移动到临时寄存器中。
4. 将ALU中的加法结果移动到ALU的输出寄存器和进位位。
5. 将数据从输出寄存器移回到寄存器A。

129

参见图6-9的真值表，我们在控制器中可遵循下面的指令流：

时钟脉冲1： 寄存器A的输入时钟有一个从低到高的转换，将目前数据总线持有的数据存到寄存器A中

时钟脉冲2： 寄存器B的输入时钟有一个从低到高的转换，将目前数据总线持有的数据存到寄存器B中

时钟脉冲3： 输入到B时钟返回到0，寄存器A的输出使能信号变成有效。这就将原来存储于寄存器A的数据放回到数据总线。时钟信号变低对寄存器B没有影响，因为这是一个下降沿

时钟脉冲4： 临时寄存器的输入时钟有一个上升沿。该动作存储了原先在寄存器A中而现在在数据总线上的数据。寄存器A的输出使能信号关闭，而寄存器B的输出使能信号打开。这时，存储于寄存器B中的数据就在总线上，而存储于寄存器A中的数据就存于临时寄存器中。ALU的输入信号ALU_0，ALU_1及ALU_2被置位用于相加*。所以ALU的输出就是A与B的和以及产生的进位

时钟脉冲5： 在clk_out信号有上升沿，该和被存入输出寄存器

时钟脉冲6： 寄存器B的输出使能被关闭，B的时钟输入返回到0，临时寄存器中的数据被放到数据总线上

时钟脉冲7： 数据通过时钟被存入寄存器A，临时寄存器的输出使能被关闭

时钟脉冲8： 系统返回到初态

注：*ALU是一个电路，能根据三个输入变量ALU_0...ALU_2的状态执行最多8个不同的算术和逻辑操作。在这个例子中，我们假设ALU的代码为000就表示ALU准备好对两个输入变量执行加法操作

130

现在，在结束这个话题之前，假如你是一个合格的CPU设计师，我要提醒一下，你还要涉及比这个例子更多的东西，但是，原理是一样的。对于初学者，我们不讨论指令ADD B，A本身如何实际地进入计算机，以及计算机如何断定该指令的功能。我们也不讨论在这之前如何确立A和B寄存器的内容。然而，至少对于状态机中实际做加法过程的那一部分，它可能是非常接近于电路真实工作情况的。

关于如何用状态机对一系列逻辑操作定序，以及在计算机中这些操作如何构成指令执行的基础，你现在已经看到了两个例子。没有基于总线体系结构和三态逻辑门设计的概念，要完成这些事情是很困难的，也是不可能的。

存储系统设计

我们已经知道了触发器的概念，尤其是，我们看到了如何用D触发器存储单个位的信息。我们记得，在时钟信号的上升沿，D触发器输入端的数据就将被存到器件中，存储的值就将被传输到Q输出。即便时钟信号已经过去，Q输出端也仍会呈现这个数据。换句话说，我们刚存了1位数据，D触发器电路就是一个存储单元。

历史上曾经存在很多不同的器件作为计算机的*随机存储器*（random access memory，RAM）来存储信息，如集成电路（IC）得到广泛应用之前，*磁芯存储器*（core memory）这样的磁器件曾是非常重要的。今天，一个IC存储器件能存512MB的数据位，随着这种器件的小型化，磁器件已经无法在速度和容量上赶上半导体存储器了。然而，基于磁存储的存储器与IC存储器相比一直有一个优势，那就是它们在电源关掉后不会丢掉数据，所以，我们仍然还用硬盘驱动器和磁带存储器作为第二级存储系统，因为即使去掉电源它们也能保持数据。

很多产业分析家都在预测硬盘驱动器的终结。在你的PDA、数字相机、或MP3播放器中会用到的现代FLASH存储器已经达到1GB存储容量。FLASH存储器在去掉电源后也能保持信息，而且比硬盘驱动器更快、更强健。但是，硬盘驱动器和磁带驱动器在容量和单个位的价格方面更胜一筹。在写这本书的时候（2004年夏），如果你愿意递送所有的回扣信息而公司也实际地向你返回了回扣单，那么一个容量为160GB的现代硬盘驱动器用大约90美元就可买到，但那是在另外一个时间的另外一回事情了。

硬盘和磁带系统是*机电系统*（electromechanical system），带有电机和运动部件。机械组件导致它们远没有IC存储器可靠，也慢得多，一般要比IC存储器慢10 000倍。正是这个原因，我们不采用硬盘作为我们计算机系统的主存储器，它们太慢了。然而，你将在后面的课程中看到，硬盘可提供几乎无限的存储能力，使得像Linux和Windows这样的操作系统能给用户这
131 样的印象，即每个台式机上打开的应用总是具有其所需要的那么多存储量。

在本部分中，我们将从D触发器作为一个单独器件开始，看一看如何将很多这样的器件互连起来构成存储器阵列。为了看看数据如何沿着同一信号路径写入和读出存储器（虽然不是在同一时刻），请看图6-10。

图中的黑箱只是基本D触发器的一个稍微简化的形式，我们去掉了 $\overline{S}$、$\overline{R}$输入和 $\overline{Q}$输出。深灰箱是三态缓冲器，它由单独的 $\overline{OE}$（输出使能）输入所控制。当 $\overline{OE}$ 为高时，三态缓冲器无作用，存储器单元的Q输出与数据线（数据输入/输出线）隔离（高阻抗状态）。然而，数据线与单元的D输入还是连着的，所以可以将数据写入单元，但是，在三态缓冲器起作用之前，写到单元的新数据对于要从单元读数据的人来说不是立即可见的。当我们将基本触发器单元和三态缓冲器结合起来时，我们就有了制造1位存储单元所需要的一切东西。这一点由图中包围的刚才所讨论的两个元件的浅灰箱所标明。

写信号有点容易被误解，所以我们要讨论一下。我们知道数据是在一个脉冲的上升沿被写入到D触发器的，这在图6-10中用写脉冲（$\overline{W}$）上的向上箭头指出了。为什么写信号$\overline{W}$被写成好像是低有效的信号呢？原因是我们通常将写信号保持在1状态。为了完成一个写操作，$\overline{W}$必须变低，然后再回到高。正是这个低到高的转换完成了实际的数据写操作，但由于我们必须将写线变低才能完成对数据的实际写操作，我们就认为写信号是低有效的。而且，你应该从这个讨论中推断出，你永远不能让$\overline{W}$线和$\overline{OE}$线同时有效，你或者让$\overline{W}$变低而$\overline{OE}$保持为高，或者反之，它们永远不能同时为低。现在，让我们回到对存储器阵列的分析。

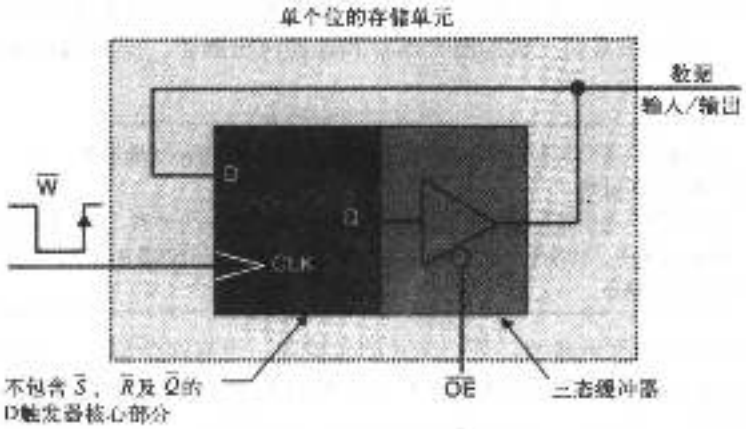


图6-10 一个单个位存储器的示意图表示，单元输出上的三态缓冲器控制什么时候Q输出可连接到总线

在解决复杂性方面，我们将采取另一个途径，就是由三态门器件和D触发器来建造存储器。图6-11显示出一个简单（也许并不那么简单）的16位存储器，组织成4个4位半字节。每个存储位是一个微型的D触发器，它在内部也有一个三态缓冲电路，所以我们可用它建造一个总线系统。

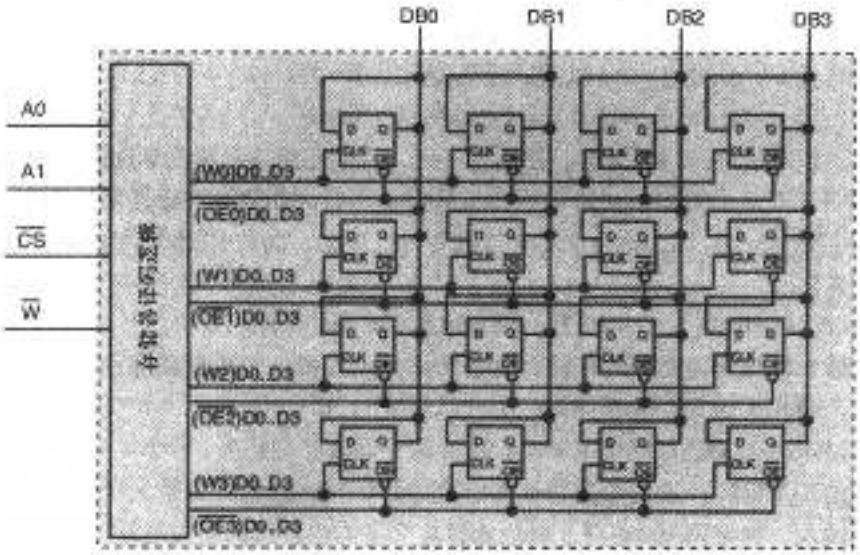


图6-11 用分立的D触发器建造的16位存储器。在这4个行中，如果将地址位A0和A1置为0，我们将访问最上面一行。类似地，(A0, A1) = (1, 0)、(0, 1) 或 (1, 1) 将分别访问第1、2、3行

每一行的4个D触发器有两个共同的控制线，用来提供时钟功能（写），并为将数据放到I/O总线提供使能功能。注意每一行相应位的位置是如何连接到同一条线的，这就是为什么我
132 们需要在每个位单元（D触发器）上都需要三态控制信号 $\overline{OE}$。例如，如果我们想将数据写入第2行的D触发器，则必须将数据从外部器件放入到DB0到DB3，而且W2信号必须变高以存储数据。此外，要将数据写进单元，$\overline{OE}$信号必须保持高状态，以防止已经存储在单元中的数据被放置到数据线上并毁掉正在写入到单元的新数据。

到16位存储器的控制输入显示在图6-11的左边，数据输入和输出（即I/O）显示在器件的顶部。注意每个数据位只有一条I/O线，这是因为数据可在同一条线上流入和流出。换句话说，我们已经使用了总线组织来简化数据流入器件和数据流出器件。让我们对每个控制输入做一下定义：

A0和A1	地址输入，用于选择存储器的哪一行正在被寻址以进行输入或输出操作。由于该器件有4行，故需要两条地址线
$\overline{CS}$	芯片选择。这个低有效的信号是该器件的主开关。如果 $\overline{CS}$ 为高，你就不能对器件进行读写
$\overline{W}$	如果 $\overline{W}$ 为高，则芯片中的数据可被像计算机芯片这样的外部器件来读。如果 $\overline{W}$ 为低，数据将被写进存储器

信号 $\overline{CS}$（片选）如你所猜测，是全芯片的总控制。没有这个信号，则这16个D触发器的任何Q输出都是不能被允许的，所以整个芯片就保持高阻抗状态，直到有外部电路介入。这样，为了读第一行的数据，不仅必须 (A0, A1) = (0, 0)，还需要 $\overline{CS}$ = 0。但等一下，还有更多要考虑的！

我们尚未完全解决问题的原因是，我们还必须决定是要从存储器中读还是要向存储器中写。如果我们想从存储器中读，我们就要对一行存储器单元的4个D触发器的每个Q输出给予
133 使能信号。这意味着为了从存储器的任何一行读出，需要下述条件为真：

- 从第0行读 > (A0=0) 且 (A1=0) 且 ($\overline{CS}$=0) 且 ($\overline{W}$=1)
- 从第1行读 > (A0=1) 且 (A1=0) 且 ($\overline{CS}$=0) 且 ($\overline{W}$=1)
- 从第2行读 > (A0=0) 且 (A1=1) 且 ($\overline{CS}$=0) 且 ($\overline{W}$=1)
- 从第3行读 > (A0=1) 且 (A1=1) 且 ($\overline{CS}$=0) 且 ($\overline{W}$=1)

假设我们想向第1行写4位数据。在该例子中，我们不想让单独的到D触发器的 $\overline{OE}$ 输入有效，因为那将打开三态输出缓冲器并导致与我们正要写入存储器的数据发生冲突。然而，我们仍然需要主 $\overline{CS}$ 信号，因为这可使芯片能被写入，因此，为将4位数据写入第1行，我们需要下面的方程成立：

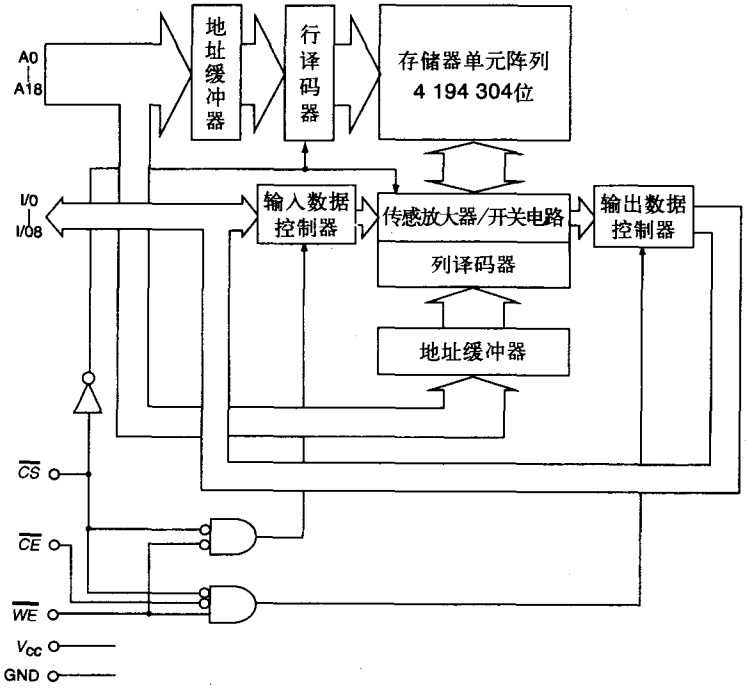
写入第1行 > (A0=1) 且(A1=0) 且 ($\overline{CS}$=0) 且 ($\overline{W}$=0)

图6-12是一个商业上使用的存储器电路的简化示意图，它来自总部设在日本的全球电子和半导体生产商NEC公司。该器件是一个被组织成512K × 8位字宽（字节）的μPD444008[1]型号的4M位CMOS快速静态RAM(SRAM)，实际的存储器阵列是由一个有4 194 304个独立存储器单元组成的X-Y矩阵，这正像我们早先讨论过的16位存储器，只是大了许多。这个电路有19条地址线输入，标记为A0...A18。我们需要这么多的地址线的原因是2^{19} = 524 288，所以19条地址线将给出恰当的组合数，我们需要这么多组合数来访问阵列中的每个存储字。

名字为 $\overline{WE}$ 的信号与以前例子中的 $\overline{W}$ 一样，只是标记不同，但仍需要一个从低到高的转换来写数据。$\overline{CS}$ 信号与以前例子中的 $\overline{CS}$ 信号一样，有一个不同是商业部件还提供了一个明

确的输出使能信号（在图6-12中称为 $\overline{CE}$），用于在读操作期间控制三态输出缓冲器。在我们的例子中，$\overline{W}$ 输入的状态就意味着 $\overline{OE}$ 操作。在实际使用中，对 $\overline{OE}$ 进行独立控制的能力有助于形成更灵活的部件，因此 $\overline{OE}$ 通常如该例子这样被加入到存储器芯片。这样，你就会看到16位存储器在操作上与商业部件一样。 134

在继续讲解之前，让我们再回到图6-11看一下。注意每一行的D触发器所具有的两个控制信号是如何进入到每个芯片的。一个信号进入到 $\overline{OE}$ 三态控制，而另一个进入到时钟输入。左边模块内的电路实际上是什么样的呢？如今，你具有了设计它所需要的所有知识和信息。



真值表

$\overline{CS}$	$\overline{CE}$	$\overline{WE}$	模式	I/O	供电流
H	x	x	未选	高阻抗	Icc
L	L	H	读	D_{OUT}	Icc
L	x	L	写	D_{IN}	
L	H	H	使不能输出	高阻抗	

图6-12 一个NECμPD444008型号的4M位CMOS快速静态RAM的逻辑图。该图来自NEC公司

注释：x为无关项。

让我们看看该电路的真值表应该是什么样的。图6-13就是该真值表。

你可以看到，当数位从16增加到4百万时，一个像图6-12中真实存储器器件μPD444008的控制逻辑会变得极为复杂，但原理还是一样的。而且，你如果参考图6-13就应该会发现，译码逻辑是高度规整的、可伸缩的，这将使硬件设计更为简明。

A0	A1	R/$\overline{W}$	$\overline{CS}$	W0	$\overline{OE0}$	W1	$\overline{OE1}$	W2	$\overline{OE2}$	W3	$\overline{OE3}$
0	0	0	0	0	1	1	1	1	1	1	1
1	0	0	0	1	1	0	1	1	1	1	1
0	1	0	0	1	1	1	1	0	1	1	1
1	1	0	0	1	1	1	1	1	1	0	1
0	0	1	0	1	0	1	1	1	1	1	1
1	0	1	0	1	1	1	0	1	1	1	1
0	1	1	0	1	1	1	1	1	0	1	1
1	1	1	0	1	1	1	1	1	1	1	0
0	0	0	1	1	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	1	1	1	1
0	1	0	1	1	1	1	1	1	1	1	1
1	1	0	1	1	1	1	1	1	1	1	1
0	0	1	1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1

图6-13 16位存储器译码器的真值表

数据总线宽和可寻址存储器

在继续考察具有更高复杂度的存储器系统设计之前，我们需要停下来喘口气，考虑一些额外的信息，以有助于使接下来的部分更容易理解。我们需要考察两部分的信息：

1. 数据总线宽
2. 可寻址存储器

一个计算机的数据总线宽决定了它在一个操作或指令中所能处理的数字大小。如果我们既考虑嵌入式系统也考虑台式PC、服务器、工作站以及大型机，我们就会看到一个数据总线宽的谱系，从4位宽直到128位宽，256位数据总线宽刚刚问世。你很容易就会问："为什么有这么多种宽度呢？"答案就是速度和成本的取舍。一个有8位数据通路连接到存储器的计算机可编程做任何一个有16位数据通路的处理器所能做的任何事情，只不过需要更长的时间。考虑这样一个例子，假设我们想将两个16位数加在一起产生一个16位的结果，要加的数放在存储器中，结果也放在存储器中。对于8位宽存储器的情况，我们就需要将每个16位的字以两个连续的8位字节存放，下面是数字相加的算法。

135

情况1：8位宽数据总线

1. 从存储器中取出第一个数的低字节并放入到一个内部存储寄存器中。
2. 从存储器中取出第二个数的低字节并放入到另一个内部存储寄存器中。
3. 将两个低字节数相加。
4. 将低字节相加结果写到存储器。
5. 从存储器中取出第一个数的高字节并放入到一个内部存储寄存器中。
6. 从存储器中取出第二个数的高字节并放入到另一个内部存储寄存器中。
7. 将两个高字节数以及前面低字节加操作的进位（如果有的话）相加。
8. 将高字节相加结果写到存有低字节相加结果的存储器位置的下一个位置。
9. 将进位（如果有的话）写到下一个存储器位置。

情况2：16位宽数据总线

1. 从存储器中取出第一个数并放入到一个内部存储寄存器中。
2. 从存储器中取出第二个数并放入到另一个内部存储寄存器中。
3. 将两个数相加。
4. 将结果写到存储器。
5. 将进位（如果有的话）写到存储器。

如你所见，情况1几乎需要2倍于情况2的步骤。用更宽数据总线所获得的效率提高依赖于所执行的算法，其变化范围可从小到百分之几的提高到大到几乎4倍速度的提高，具体取决于实现的算法。

下面是对各种总线宽的常用场合的总结：

- 4位、8位：装置、调制解调器、简单应用
- 16位：工业控制器、汽车应用
- 32位：远程通信、激光打印机、台式PC
- 64位：高端PC、UNIX工作站、游戏（任天堂64）
- 128位：用于游戏的高性能视频卡
- 128位、256位：下一代超长指令字（VLIW）机器

有时我们试图通过采用宽的内部数据总线和窄的存储器来达到经济化。例如，我们在本课中将要学习的Motorola 68000处理器有16位外部数据总线和32位内部数据总线。它需要用两次存储器取操作将32位数从存储器中取出来，但一旦在计算机内部进行操作，它就能处理单个的32位值。

6.2 地址空间

136

计算机设计的下一个考虑就是为计算机配备多少可寻址的空间。计算机的地址空间（address space）的定义为外部可访问的存储数量。这个地址空间小到一个简单器件的1024字节，大到一个高性能计算机的60GB，而且，一个处理器所能寻址的存储量与系统中实际的存储量无关。PC中的奔腾处理器能寻址超过40亿字节的存储器，但大多数用户的计算机中很少有超过1GB的存储器。下面是可寻址存储器的一些简单例子：

- 一个简单的微控制器（如在Mr. Coffee机器中）可有10条地址线A0...A9，能寻址1024字节的存储器（$2^{10}=1024$）。
- 一个通用的8位微处理器（如防盗自动警铃中的微处理器）有16条地址线A0...A15，能寻址65 536字节的存储器（$2^{16}=65\ 536$）。
- 一个原始的并引起PC革命的Intel 8086微处理器有20条地址线A0...A19，能寻址1 048 576字节的存储器（$2^{20}=1\ 048\ 576$）。
- Motorola 68000微处理器有24条地址线A0...A23，能寻址16 777 216字节的存储器（$2^{24}=16\ 777\ 216$）。
- 奔腾微处理器有32条地址线A0...A31，能寻址4 294 967 296字节的存储器（$2^{32}=4\ 294\ 967\ 296$）。

你马上就会看到，我们通常根据字节（8位值）来提及可寻址存储器，即使存储器宽大于8也是这样，这就导致了各种存储器的寻址定义不清，我们马上就会遇到这些问题。

分页

假设你正在读一本书，特别是，这还是一本非常奇怪的书，它在每一页恰好有100个字，

且这些字每一个都是按0到99编号。这本书恰好有100页，也从0到99编号。速算一下你就会知道这本书有10 000个字（100字/页×100页），而且，在每一页的每一个字的紧接着的地方就是该字在全书的绝对编号，第0页的第一个数字给出的是地址0000，最后一页的最后数字给出的是9 999。这确实是一本非常奇怪的书！

无论如何，我们注意到了相当有趣的事情。一页上的每个字可以用两种方式在书中唯一地识别出来：

1. 给出字在0000到9 999之间的绝对编号。

2. 给出该字在从00到99之间的页编号，然后再给出该字在页上从00到99之间的位置。

这样，在第36页上的第45个字就可在绝对寻址中被编号为3644，或者为页号＝36，偏移＝44。如你所见，无论我们选择哪种方法形成地址，都能找到正确的字。如你所预料到的，这种类型的寻址称为分页（paging）。分页要求提供两个数字来形成我们感兴趣的存储器位置的正确地址：

1. 存储器中包含该数据的页的页号（page number）。

2. 在该页中存储器位置的页偏移（page offset）。

图6-14给出了一个这样的微处理器方案（有时我们将希腊字母“mu”和“P”用在一起，即μP，作为微处理器的速记表示）：该微处理器有20条地址线A0...A19，所以能寻址1 048 576个字节的存储器。不幸的是，我们没有一个正好匹配该处理器存储地址空间大小的存储器芯
137 片，这是通常的情况，所以我们需要加入额外的电路（并成倍增加存储器设备）来提供足够的存储器，使得处理器发出的每个可能的地址都能链接到一个相应的存储器位置。

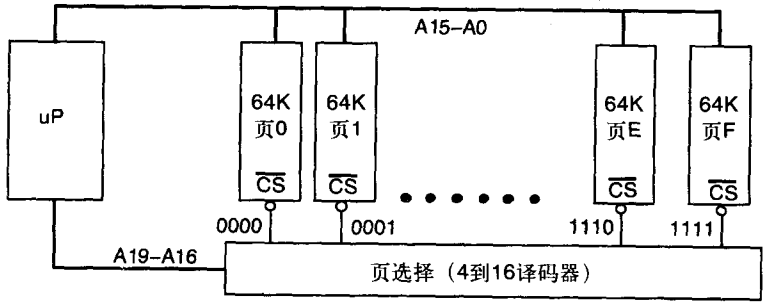


图6-14 20位微处理器的存储器组织。存储空间被组织成16个64KB的存储页

由于这个存储器系统是用64KB存储器件构建成的，所以16个存储器芯片的每一个都有A0到A15的16条地址线，因此，从A0到A15的每一条地址线都进入了每个存储器芯片的每个地址引脚。

来自处理器的剩余4条地址线A16到A19用于选择将要寻址16个存储器芯片中的哪一个。提醒一下，4条最高有效的地址线A16到A19可有从0000到1111（或十六进制0到F）的16个组合值。

让我们考虑图6-14中的微处理器，假设它发出十六进制地址9A30D。来自处理器的最低有效地址线A0到A15都连接到了16个存储器件的相应地址输入，因此，每个存储器件都能看到十六进制地址值A30D。地址位A16到A19到达页选择电路。这样，我们也许会怀疑系统究竟能不能工作良好，每个存储器件存储在地址A30D的数据不会互相干涉并给出垃圾数据吗？

答案是否定的，这多亏每个存储器芯片上的 $\overline{CS}$ 输入。假设存储器真想要得到存储器位置9A30D中的字节，则其余4条来自处理器的地址线A16到A19就用于选择将要寻址16个存储器

芯片中的哪一个。记住，4条最高有效地址线A16到A19可有从0000到1111（或十六进制0到F）的16个值组合。

这看起来有些可疑，就像我们先前讨论过的译码器设计问题一样。这个存储器设计有一个4：16译码器电路用来以4条最高有效地址位进行页选择，并以其余的16条地址位形成数据在存储器芯片中的页偏移。注意，同样的地址线A0到A15进入到了每个存储器芯片。所以，如果处理器发出十六进制地址E3AB0，所有的16个存储器芯片都会看到地址3AB0。为什么不会出现问题呢？现在我确信你们会异口同声地说：是因为有三态缓冲器使我们能将这16个页连接到共同的数据总线上。地址位A16到A19确定了这16个 $\overline{CS}$ 信号中的哪一个打开，其余15个信号保持高状态，所以相应的那些芯片不会被选中，对数据传输没有影响。

分页是计算机系统中的一个基本概念，随着我们对计算机系统操作的进一步的探究，这个概念会反复出现。在图6-14中，我们将处理器的20位地址空间组织成16页，每页64KB的形式。我们这样做的一个可能原因是我们正在使用64K存储器芯片，这有一定的随意性，因为根据能获得的存储器件的类型，我们还可能以完全不同的方式制定分页方案。图6-15显示出另 138
外一种可能的存储器组织方式。此外，我们还可以从多个芯片构造存储器的每一页，所以这些页本身也需要额外的硬件进行译码。

页地址	页地址位	页偏移	偏移地址位	
NONE	NONE	0 to 1,048,575	A0 to A19	线性地址
0 to 1	A19	0 to 524,287	A0 to A18	
0 to 3	A19–A18	0 to 262,143	A0 to A17	
0 to 7	A19–A17	0 to 131,071	A0 to A16	
0 to 15	A19–A16	0 to 65,535	A0 to A15	我们的例子
0 to 31	A19–A15	0 to 32,767	A0 to A14	
0 to 63	A19–A14	0 to 16,383	A0 to A13	

图6-15 一个20位地址空间的可能分页方案

应该强调的是，在计算机设计中，一般来说，存储器组织的型式对软件开发者应该是透明的。硬件设计的规范当然要为软件开发者提供存储器映像，为每种存储器（如RAM、ROM、FLASH等）提供地址范围。然而，软件开发者无需操心存储器译码是如何组织的。

从软件设计者的角度来看，处理器发出存储器地址，而正确地解释该地址并分发到适当的存储器件则是硬件设计的任务。

分页是很重要的，因为它在将微处理器的线性地址空间（linear address space）映射到存储器件的物理容量时是必要的。有些微处理器（如Intel 8086及其后继产品）实际上采用分页作为主要的寻址模式。外部地址是由来自一个寄存器的页号和来自另一个寄存器的偏移值形成的。下次你的计算机崩溃，你看到臭名昭著的“蓝屏死机”时，仔细观察那个古怪的十六进制地址。你可能会看到：

BD48: 0056

这是用页-偏移表示的32-位地址。

磁盘驱动器采用分页作为其唯一的寻址模式，每个磁盘被分为512B的扇区（页）。一个4GB的磁盘有8 388 608页。

设计一个存储系统

你也许不同意，但我们就要将学到的知识整理起来设计一个真实的存储系统了。嗯，也许我们还没有完全准备好，但也非常接近了，接近到足以做一次尝试。图6-16就是一个具有16位宽数据总线的计算机系统的示意图。

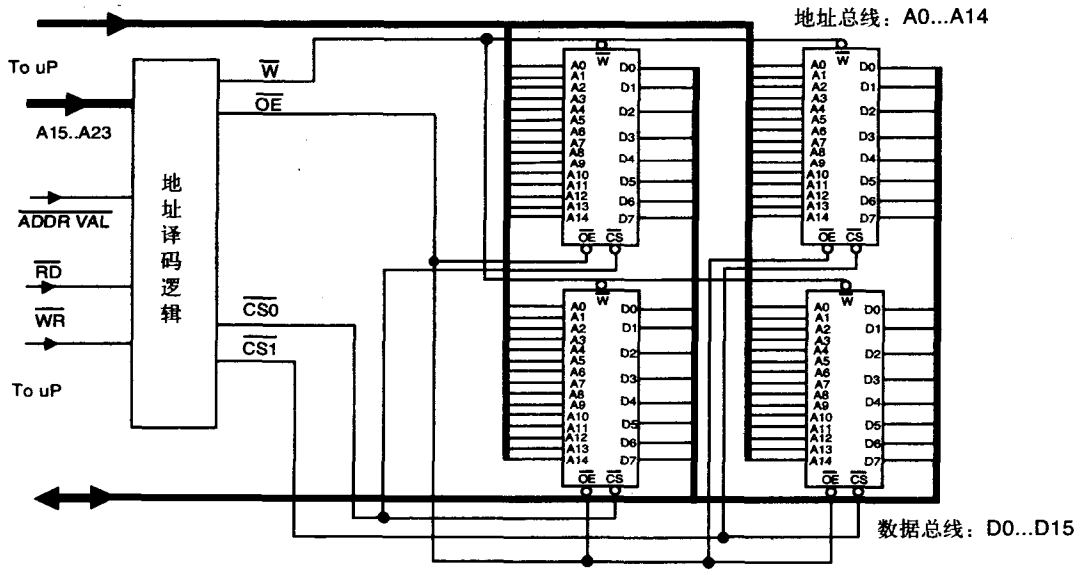


图6-16 一个从4个32K×8存储器芯片构建的64K×16存储器系统的示意图

首先，快速回忆一下二进制算术，就像大多数工程应用一样，我们使用速记符号“K”表示1024，而不是1000。这样，256K实际上是指262 144而不是256 000。通常，上下文信息可消除歧义，但不总是这样，所以要谨慎。

图6-16的电路看起来比我们迄今考虑过的任何电路都要复杂得多，但它确实与我们已经
139 学过的电路没有多大差别。首先，我们看看这些存储器芯片。每个芯片有15条地址线输入，意味着它有32K的存储地址，因为2^{15}=32 768。此外，每个芯片有8条数据输入/输出（I/O）线进入其中。然而，你应该紧记，图6-16的数据总线实际上是16位宽（D0...D16）的，所以我们实际上需要两个8位宽的存储器芯片来提供与数据总线宽度匹配的正确存储器宽度。这一点我们将在图6-17中进行详细讨论。

图6-16中4个存储器芯片的内部组织与我们已经学过的相同，除了这些器件包含256K存储单元，而我们在图6-11中学过的存储器有16个存储单元。它虽然更复杂一点，但思想是一样的。而且，画256K存储器单元要比画16K存储器单元需要更多的时间，所以我采取了更容易的方式。

在如何加入更多的器件来同时提高宽度（数据总线的大小）和深度（可得到的存储位置的数量）这个意义上，这个32K存储位置且每个位置8位宽的存储器芯片的排列在概念上与图6-11中16位的例子具有相同的思想。在图6-11中，我们讨论了一个16位存储器，它组织成了4个存储位置，每个位置4位宽。在图6-16中，因为有32 768行和8列，故在每个芯片中总共有262 144个存储单元。

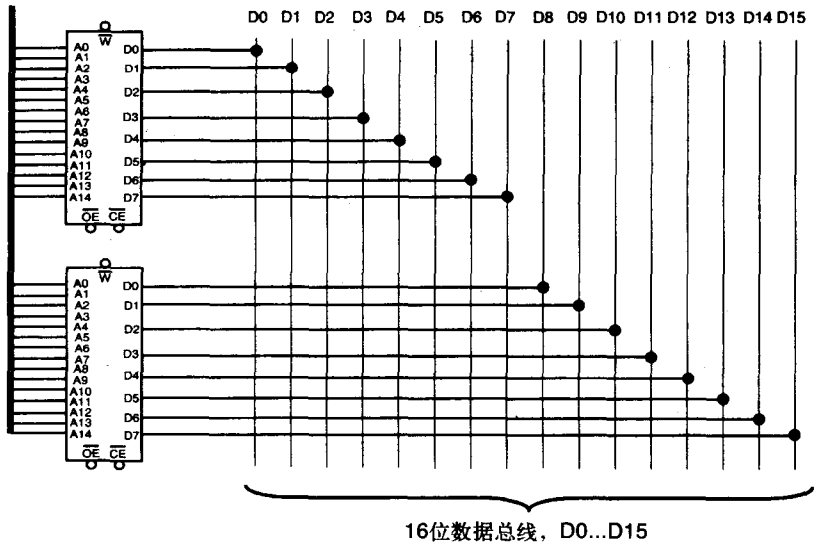


图6-17 扩展一个存储系统的宽度

每个芯片有 $\overline{OE}$、$\overline{CS}$ 和 $\overline{W}$ 三个控制输入。为了从存储器件中读，我们必须做如下步骤：

1. 将我们要从中读的存储位置的正确地址放到A0到A14上。
2. 将 $\overline{CS}$ 置低，打开芯片。
3. 保持 $\overline{W}$ 为高，使得不能向芯片写。
4. 将 $\overline{OE}$ 置为低，打开三态输出缓冲器。

存储器芯片然后就将数据从一个芯片相应的存储器位置放到数据线D0到D7，从另一个芯片相应的存储器位置放到数据线D8到D15。为了对存储器件写入，我们必须做如下步骤： 140

1. 将我们要向其中写的存储位置地址放到A0到A14。
2. 将 $\overline{CS}$ 置低，打开芯片。
3. 将 $\overline{W}$ 置低，使得能向芯片写。
4. 保持 $\overline{OE}$ 为高，使得三态输出缓冲器无效。
5. 将数据放到数据线D0到D15，其中D0到D7到一个芯片，D8到D15到另一个芯片。
6. 将 $\overline{W}$ 从低变高，将数据写入相应的存储器位置。

既然理解了单独的存储器芯片如何工作，让我们继续进行对电路整体的讨论。在这个例子中，微处理器有从A0到A23的24条地址线。A0到A14直接导入存储器芯片，因为每个芯片有32KB的地址空间。从A15到A23这9条最高有效地址位用来为译码逻辑块提供分页信息。这9位告诉我们，该存储器空间可被分成512页，每页32K个地址。然而，精明的读者立刻会注意到，系统中总共只有4个存储器芯片，肯定有什么地方出了差错！我们没有足够的存储器芯片来充满512页。唉，真见鬼！我真不愿意看到这种情况！

实际上，这根本不是一个问题。它意味着在512页可寻址空间中，我们的计算机有两页真实存储器，而另外的510页只是空页，这是一个问题吗？很难说。如果我们能够将代码装入这两页而且确实这么做了，为什么还要加入我们用不到的存储器的成本呢？我可以从个人经验告诉你，为节省成本，在将代码塞入较小的存储器芯片方面已经付出了很多努力。

你要问的另一个问题是这样的："好，既然μP的可寻址空间不是完全填满的，那么存储器被置于处理器地址空间的哪个地方了呢？"这是一个非常好的问题，因为我们目前还没有足够

141 的信息来回答这个问题。但是，在尝试规划这个计算机和存储系统之前，我们的硬件设计必须要使存储器芯片被正确地译码为其被设计到的页位置，稍后我们就会看到这是如何做到的。

让我们返回到图6-16。由于数据总线有16位宽，但每个存储器芯片只有8数据位宽，所以对于每页的存储，我们实际上需要两个存储器芯片，理解这一点是很重要的。这样，为了建造16位宽存储器，我们需要两个芯片，这一点可见图6-17。注意每个存储器件是如阿连接到数据总线中一组8条线的。当然，地址总线引脚A1到A14必须连接到地址总线的相同线上，因为我们正对两个存储器芯片的相同地址位置进行寻址。

既然你已看到两个存储器芯片是如何“堆叠”起来形成一个32K × 16存储器中的一页，采用4个芯片设计一个32K × 32的存储器对于你就不应是一个问题了。

你可能注意到了，这个存储系统设计的例子中没有微处理器的时钟。存储器到处理器作为一个计算机系统中最重要的连接之一，需要一个时钟信号来对处理器到存储器进行同步。事实上，很多存储器系统不需要时钟信号来确保可靠的性能，唯一需要考虑的事情就是存储器电路和处理器总线操作之间的时序关系。在下一章中，我们将会更详细地考虑存储器总线周期，这里是一个预览。NEC μPD444008有三个型号，实际型号的号码是：

- μPD444008-8
- μPD444008-10
- μPD444008-12

数字后缀8、10、12是指每个芯片的最大访问时间（access time）。访问时间是一个基本的规范，它决定了一旦控制输入已经正确地确立之后，芯片能以多快的速度可靠地返回数据。因此，假设到芯片的地址已稳定，$\overline{CS}$和$\overline{OE}$置为有效，则在延时8、10、12ns（取决于使用的芯片型号）后，数据就可被读入处理器。芯片生产商NEC保证，在芯片被设计时所要求的工作温度范围内，访问时间是满足要求的。对于大多数电子器件，商业温度范围是0摄氏度到70摄氏度。

让我们做一个简单的例子来理解其意义。后面我们还要对此做更详细的考察，但现在做一下准备也没有害处。假设我们有一个具有500MHz时钟的处理器，这意味着每个时钟周期是2ns。我们的处理器需要5个时钟周期来做一次存储器读，在第5个时钟周期的下降沿将数据读入处理器。地址和控制信息在第1个时钟周期的上升沿由处理器发出。这意味着处理器需要4.5 × 2即9ns的时间做一次存储器读操作。然而，我们还没有完全完成计算，我们的译码逻辑电路也会产生一些时延。假设从处理器给出控制信号和地址信号到译码逻辑向存储系统提供正确的信号之间的时间为1ns，这意味着我们实际上需要8ns而不是9ns准备好数据。这样，只有最快型号的产品（通常这意味着最昂贵的型号）才能在该设计中可靠地工作。

我们能做什么吗？我们可以减慢时钟。假设我们将时钟频率从500MHz改变为400MHz，
142 这就将每时钟周期加长到2.5ns。现在4.5个时钟周期就需要11.25ns而不是9ns，减去通过译码逻辑的时延1ns，我们就需要一个10.25ns或更快的存储器，以使得工作可靠。这看起来非常鼓舞人心。我们甚至可以更多地减慢时钟，这样就能使用更便宜的存储器件。项目经理能不高兴嘛！但不幸的是，我们刚才只是做了折中，这个折中就是我们刚才也将处理器减慢了20%。处理器做的任何事现在都要延长20%的时间。我们能够忍受吗？此时，我们也许不知道。在能完全回答这个问题之前，我们需要对代码的执行时间和性能需求做一些谨慎的衡量，甚至我们接着会做出一些相当粗略的假设。

无论如何，以上讨论的关键是在这个存储系统的设计中没有明确的时钟。时钟依赖性是隐含于存储器到处理器接口的时序要求中的，但时钟本身并不需要。在这个特定的设计中，

我们的存储系统异步地连接到处理器。

目前，大多数的PC存储器是同步设计，时钟信号是处理器到存储器接口的控制电路的组成部分。如果你曾经在你的PC中加入过存储“条”，那么你就是采用同步动态随机存储器（synchronous dynamic random access memory, SDRAM）芯片对PC的能力进行了提升。印刷电路板（即存储条）就是将存储芯片机械地连接到PC母板的方便方式。

图6-18是一个64MB（Mbyte）SDRAM存储器模块的照片，这个模块存有64MB数据，被组织成1M×64。模块上总共有16个存储芯片（正面和反面），每个芯片有32M字的容量，被组织成8M×4。我们将在本章的后面看到异步（即静态）存储系统与同步（即动态）存储系统之间的差异。

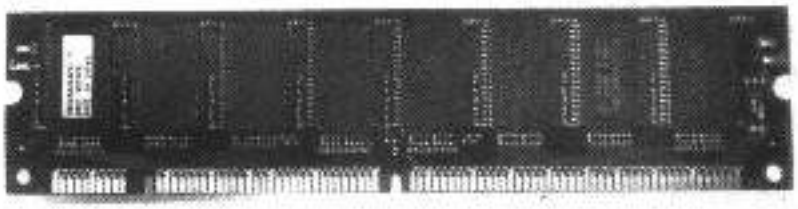


图6-18 64MBSDRAM存储模块

真实存储系统中的分页

图6-16中的4个存储芯片给出了两个32K×16的存储页，这就留出了510个空的存储页。我们如何知道在哪里找到这两个存储页，在哪里存储页是空的呢？答案是这要由你（或硬件设计者）来规定存储页在哪里。你很快就会看到，在68000系统中，我们想让ROM或FLASH这样的非易失性存储器（nonvolatile memory）处于存储器的开始处，并从那里开始升高地址。为做这个练习，我们将真实存储器的两个页放置于第0页和第511页。

假设处理器有24个地址位，这对应于大约16M的可寻址存储器（2^{24}个地址位置）。习惯上 143
将RAM存储器（读/写）置于存储器的顶端，但也不要求一定这样。在多数情况下，这取决于处理器的体系结构。在该例子中，我们需要计算出这两个真实存储页之一如何对从0x000000到0x007FFF的地址进行反应。存储器的前32K对应于第0页，另一个32K字的存储区域应置于0xFF8000到0xFFFFFF，即第511页。我们是如何知道的？很简单，就是分页。我们的总共16 777 216字的存储系统可被划分为512页，每页32K。由于有9位用于分页，所以我们能将绝对地址如表6-2那样划分。

表6-2 一个24位寻址系统的页号和存储地址范围

页号（二进制）A23.....A15	页号（十六进制）	绝对地址范围（十六进制）
000000000	000	000000 to 007FFF
000000001	001	008000 to 00FFFF
000000010	002	010000 to 017FFF
000000011	003	018000 to 01FFFF
...	...	
111111111	1FF	FF8000 to FFFFFF

当存储地址落在正确的范围内时，我们想让两个突出显示的存储范围用 $\overline{CS0}$ 或 $\overline{CS1}$ 有效来作出反应，而其他存储范围保持这两个信号无效。第1FF页的译码电路如图6-19所示。第

000页的电路留给你作为练习。

注意有一个称为 $\overline{ADDRVAL}$（地址有效）的新信号，这个地址有效信号（或某个其他类似的信号）由处理器发出，用于通知外部存储器在总线上的当前地址是稳定的。为什么这是必要的？记住地址总线上的地址总是在变化的，仅仅一条执行指令就涉及用不同的地址值对存储器进行5次或更多次的访问。地址停留时间越长，处理器的性能就越低，因此，处理器必须向存储器发出当前地址值正确的信号，然后存储器对此作出反应。此外，有些处理器可能有 $\overline{RD}$ 和 $\overline{WR}$ 两个独立的信号，分别表示读操作和写操作，另外一些处理器只有单一的信号线 $R/\overline{W}$。每个方法都有其优势和缺点，我们在此不必考虑。现在，我们假设处理器有两个独立的信号，一个用于读操作，一个用于写操作。

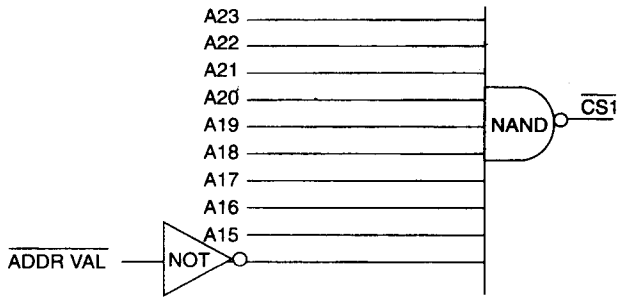


图6-19 用于对图6-16的存储器的顶页进行译码的电路示意图

正如你从图6-16中和从存储器芯片如何在系统中工作的讨论中所了解的，显而易见，我们可以将对存储器进行读和写的必要逻辑条件表达如下：

$$\text{存储器读} = \overline{OE} * \overline{CS} * WR$$

144

$$\text{存储器写} = OE * \overline{CS} * \overline{WR}$$

在这两种情况下，我们都需要使 $\overline{CS}$ 信号有效以对存储器进行读或写。正是对芯片使能（或芯片选择）信号的控制，才使我们能控制在处理器存储空间的哪个区域的特定存储芯片能变成活动状态。

除了对SDRAM存储器做了简要介绍，我们只考虑了用静态RAM（SRAM）作为存储器件。如你所见，静态RAM源自D触发器。它与处理器的接口相对简单，因为我们所要做的事就是给出地址和适当的控制信号，等待适当的时间，然后就能读或写存储器。如果我们长时间不访问存储器也没有问题，因为只要对电路施加了电源，触发器门设计的反馈机制就能正确地保持数据。然而，我们不得不为这种简单性付出代价，一个现代的SRAM存储器单元需要5或6个晶体管来实现实际的门设计。当你在谈论关于存储256M位数据的存储芯片时，6个晶体管的存储单元就占用了大量宝贵的硅片（硅模）空间。

目前，计算机（像你的PC）中的大多数高密度存储器采用的都是另外一种不同的存储技术，称为动态RAM，即DRAM。DRAM单元要比SRAM单元小得多，典型情况是每个单元只用一个晶体管。一个晶体管不足以生成在单元中存储数据所需的反馈电路，所以DRAM采用了一种完全不同的机制，这个机制称为存储电荷（stored charge）。

如果你曾经在冬季干燥的一天走过一个地毯，然后又接触到类似冰箱的某金属时受到了电击，那么你就熟悉存储电荷了。当你走过地毯时，你的身体拾取了过多的电荷（现在你代表逻辑1状态），然后当电荷离开你的身体时你受到电击，你就回到了逻辑0状态。DRAM单元恰以同样的方式工作。每个DRAM单元都能存储少量电荷，可被DRAM电路检测为1。存储一些电荷，单元是逻辑1，消除电荷，单元就是逻辑0。（然而，就像电荷存储于你的身体一样，如果你不做任何事情来补充电荷，那么电荷最终会泄漏完。）这有点更复杂了，存储的电荷最终会代表0而不是1，但这已足够使我们理解这个概念了。

对于一个DRAM单元的情况，我们补充电荷的方式就是周期性地读单元，所以，DRAM的命名就是取自于我们经常地对其进行读操作的事实，即使我们实际上并不需要存储在其中的数据，这就是DRAM名字中动态部分的意思。从单元中读的过程称为刷新周期（refresh cycle），必须在时间空隙中完成。事实上，DRAM的每个单元必须每几毫秒刷新一次，否则单元就有失去其数据的危险。图6-20显示出64M位DRAM存储器组织的示意图。 145

存储器被组织成8192行×8192列的矩阵（2^{23}）。为了对任一DRAM存储单元进行唯一寻址，需要26位地址。由于我们已经将其建造成矩阵，而且封装上的26个引脚会引入额外的复杂性，所以就通过对XY矩阵提供一个独立的行地址和一个独立的列地址来对存储器进行寻址。对我们来说幸运的是，产生这些地址的过程是由置于PC母板上的特殊芯片组来处理的。让我们回到刷新问题。假设我们必须在每10ms对64M单元至少刷新一次，这是否意味着我们必须做64M次刷新周期呢？实际上不需要。只要向存储器发出行地址就足够保证该行的所有8192个单元被立即刷新。现在我们的问题就更容易处理了，举个例子，如果设计规范给我们16.384ms来刷新存储器中的8192行，则我们必须平均在每$16.384\times10^{-3}/8.192\times10^{3}$秒刷新一行，即每两微妙刷新一行。

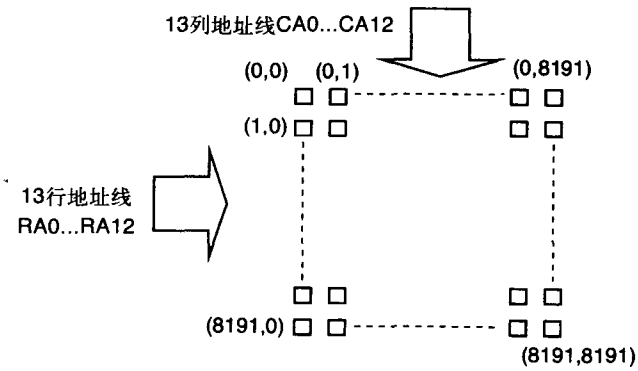


图6-20 64M位DRAM存储器的组织

所有这些看起来很复杂，确实是这样的。设计一个DRAM存储系统不适合于一个初学的硬件设计者，DRAM引入了若干个新级别的复杂性：

- 我们必须将总体地址分裂为一个行地址和一个列地址。
- 我们必须在每大约1μs左右停止访问存储器，并做一个刷新周期。
- 如果处理器需要使用存储器，而刷新也需要访问存储器，我们就需要某种方式来同步两个竞争的过程。

这使得将DRAM与现代处理器进行接口成了一个相当复杂的操作。但幸运的是，现代的支持芯片组（support chip set）能很好地处理这个问题。而且，如果每两微秒必须做一次刷新有些过多，那么别忘了你的2GHz的Athlon或奔腾处理器在每微秒发出的4 000个时钟周期，所以在需要做一次刷新周期前我们能做很多处理工作。

竞争的存储器访问操作（读、写及刷新）所引起的冲突问题在很大程度上被缓解了，因为现代PC处理器包含称为cache的片上存储器。我们将在后面章节中更详细地讨论cache，但现在我们在片外DRAM存储器上就能看到cache的影响，这个影响就是大大地降低了处理器对外部存储系统的要求。

我们将要看到，处理器所需要的指令和数据已在cache中的概率通常大于90%，虽然精确的概率受运行算法的影响。这样，只有10%的时间处理器需要转到外部存储器来访问没有在cache中的数据或指令。在现代处理器中，数据在外部存储系统和处理器之间的传输是成组的(in bursts)，而不是一次传输一个字节或一个字。成组访问是非常高效的数据传输方式，事实上，你可能已经很熟悉这个概念了，因为你PC中的很多其他系统也依赖于成组数据传输。例如，你的硬盘驱动器每次以一个扇区为一组的方式将数据传送到存储器。如果你的计算机连

146 接到一个10Base-T网络或100Base-T网络，则它一次能处理256个字节的数据包，系统资源若一次只传输一个字节就太低效太浪费了。

SDRAM存储器还被用来高效地与带有片上cache的处理器进行接口，特别是用于存储器与处理器片上cache之间的成组访问。图6-21是从一个SDARM存储器件的数据单所摘录的一个选段。该器件来自Micron Technology®公司，这是一个位于爱达荷州Boise的半导体存储器制造商。这个时序图是MT48LC128MXA2[2]系列SDRAM存储器的，这种器件是512M位的，可被组织成4位宽、8位宽或16位宽的数据通路，"X"是为组织（4位宽、8位宽或16位宽）而加入的占位符。这样，MT48LC128M4A2就被组织成32M×4，而MT48LC128M16A2则被组织成8M×16。

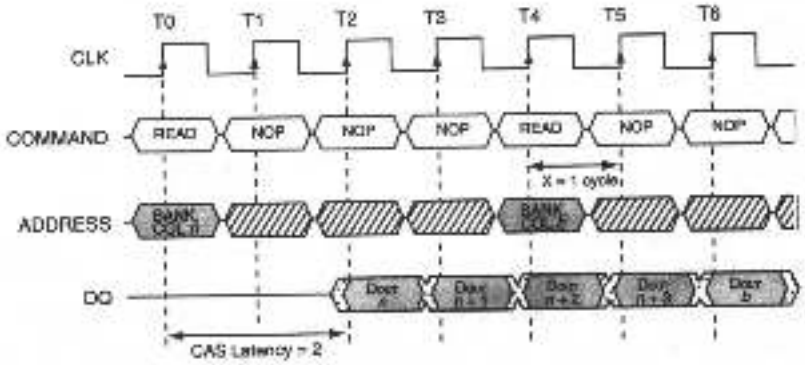


图6-21 成组存储器访问的时序图。是针对Micron Technology公司部件编号为MT48LC128MXA2的SDRAM存储器芯片的，该图来自Micron Technology公司

这些器件在操作上要比我们迄今所看到的简单SRAM存储器要复杂得多。然而，我们在图6-21中就能看到基本的成组传输行为。

标记为COMMAND、ADDRESS和DQ的域表示的是数据带而不是单个位，这是一个简化，能使我们显示1组信号，如14个地址位，而不必显示每个单独信号的状态。这个带用于显示信号在哪里必须是稳定的，在哪里才允许变化。注意所有信号是如何在时钟的上升沿被同步的。一旦发出READ命令，而且提供了成组传输起始处的地址，就有两个时钟周期的等待时间，然后在每个接续的时钟周期，就可持续地得到芯片中存储的数据，显然，这远比一次读一个字节要有效率。

当我们更详细地考察cache存储器时，我们会看到片上cache也被设计成用外部存储器以成组数据的方式填充。这样，我们在不得不为从外部存储器到片上cache的数据传输建立初始条件而招致怨罚，但一旦装入了传输参数，存储器到存储器的数据传输就可相当快速地进行。对于该系列的器件，数据传输能以133Hz的最大时钟速率进行。

更新的SDRAM器件称为双数据速率（double data rate，DDR）芯片，在时钟的上升沿和下降沿都能传输数据。这样，一个具有133MHz时钟输入的DDR芯片就能以266MHz的速度传输数据。这些部件以未知的原因被指定为PC2700器件，任何SDRAM芯片，只要能遵照266MHz的时钟速率就是PC2700。

现代DRAM设计呈现出很多不同的形式。我们一直在讨论SDRAM，因为这是在现代PC中最常见形式的DRAM。你的图形卡包含视频DRAM，较旧的PC包含扩展数据输出
147 (extended data out，EDO) DRAM，目前，最常见的SDRAM类型就是DDR SDRAM。其中最令人惊异的事情是这种类型存储器的令人难以置信的低成本。在写本书的时候（2004年夏），

你可以花大约每兆字节10美分的价格买到512MB的SDRAM。一个具有相同容量的静态RAM存储器的价格则超过了2 000美元。

存储器到处理器的接口

本章我们要解决的最后一个主题涉及存储系统与处理器如何进行相互通信的细节。诚然，我们可以简略地介绍一下，因为在当今世界上已有超过300种的商业微处理器系列，所以关于一个主题有太多的变化。让我们尝试采用概括的看法而不过深地陷入个别差异中。

一般来说，大多数基于微处理器的系统包含三个主要的总线组：

- 地址总线：一个从处理器到存储器的单向总线。
- 数据总线：一个双向总线，它在读操作期间将数据从存储器运送到处理器，在写操作期间将数据从处理器运送到存储器。
- 状态总线：一个异质总线，由各种控制和内务处理信号组成，这些信号用于协调处理器、存储器以及其他外设的操作。典型的状态总线信号包括：
  a. 复位
  b. 中断管理
  c. 总线管理
  d. 时钟信号
  e. 读信号和写信号

Motorola* MC68000处理器的这些信号都在图6-22中。该68000处理器有24位地址总线和16位外部数据总线。然而在内部，地址和数据总线可达32位宽。我们将在本节的后面讨论中断系统和总线管理系统。 148

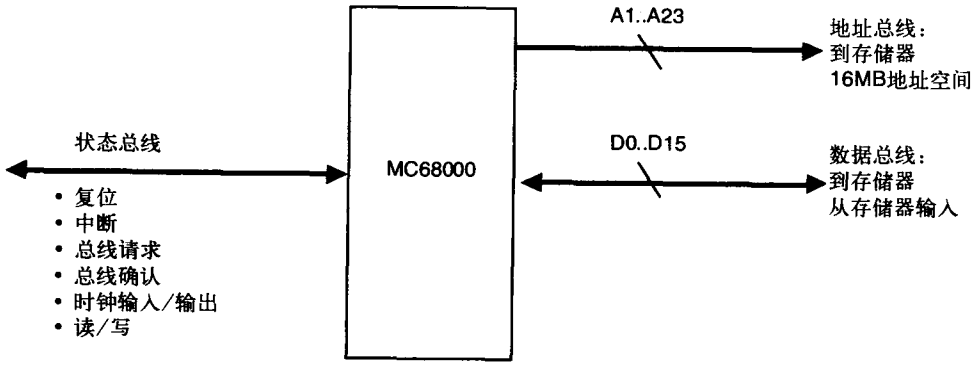


图6-22 Motorola 68000处理器的三个主要总线

* Motorola公司最近将其半导体产品部门（SPS）分割出来形成了一个新的公司——Freescale公司。但是，老习惯很难改变，所以我们继续将源于68000体系结构的处理器称为Motorola MC68000

地址总线是所有单个地址线的集总，我们之所以说它是同质总线（homogeneous bus），是因为构成总线的所有单独信号都是地址线，地址总线也是单向的，地址由处理器生成并送到存储器。存储器不生成任何地址并通过总线将它们送到处理器。

数据总线也是同质的，但它是双向的。数据通过读操作从存储器送到处理器，通过写操作从处理器送到存储器。这样，数据可在两个方向中的任一方向流动，具体流动方向取决于被执行的指令。

状态总线是*异质的*（heterogeneous），它由不同种类的信号构成，所以我们不能像对地址总线和数据总线一样对其进行聚合。而且，一些信号是单向的，另一些信号是双向的。状态总线是“内务管理”总线。所有在控制系统操作方面也需要的信号就被归入到状态总线。

现在，让我们考察这些总线上的信号如何同存储器一起工作，使得我们能够读和写。图6-23显示了存储器接口的处理器这一侧。

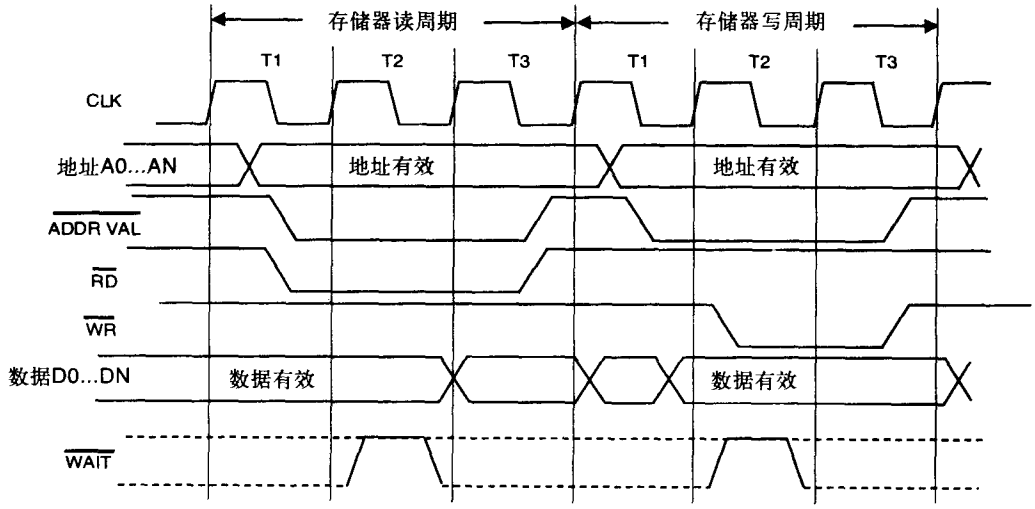


图6-23 一个典型微处理器的时序图

现在我们就能看到处理器和时钟如何一同工作来为存储器数据访问进行定序。虽然开始时这可能看起来很令人困惑，但它实际上非常简单易懂。图6-23是处理器的一个“简化的”时序图，我们已经忽略了很多在各种处理器中可能出现或不出现的附加信号，并力图将我们的讨论限制在本质问题上。

Y轴显示了来自处理器的各种信号。为简化，我们将所有的地址总线和数据总线组合成信号“带”。遵照这种方式，在任意给定时间，我们可假设一些是1一些是0，但关键问题是我们
149 必须规定它们什么时候是有效的。地址总线和数据总线中的交叉点X是时间点的符号表示，表示总线上的地址和数据可能正在变化，比如地址正变为一个新的值，或者数据正来自处理器。

由于微处理器是一个状态机，所以任何事情都用时钟的边沿同步。一些事件在上升沿发生，而一些则可能用下降沿同步。此外，为了方便，我们将总线周期划分成可识别的时间标签，称为“T状态”。不是所有的处理器都以这种方式工作，但这是很多处理器实际如何工作的一个合理近似。要记住的是处理器总是运行这些总线周期，这些操作形成了处理器和存储器之间交换数据的基本方法，所以，我们就能回答本章开始时提出的一个问题。回忆一下，操作ADD B,A的状态机真值表省去了首先数据如何进入寄存器，以及指令本身如何进入计算机的任何解释。

这样，在考察处理器/存储器接口的时序图之前，我们需要提醒我们自己，接口的控制是由状态机的另一个部分操纵的。用算法的术语来说，就是我们对状态机操纵存储器接口的部分做了一次“函数调用”，数据由该算法来读或写。

让我们从读周期开始。在T1中时钟的下降沿期间，地址变得稳定，$\overline{ADDRVAL}$信号变为低，成为有效状态。而且，$\overline{RD}$信号变低，指明这是一个读操作。在T3的下降沿，读和地址有效信号取消，向存储器指明该周期正在结束，来自存储器的数据正在被处理器读。这样，

存储器必须能在两个全时钟周期内（T2的全部加上半个T1和半个T2）提供数据给处理器。

假设存储器没有快到足以保证数据能及时准备好，我们针对NEC静态RAM芯片的例子讨论这种情况，并判定一个可行的解决方案就是减慢处理器的时钟直到存储器的访问时间要求确保在规定范围内。现在，我们将考虑另一种方案。在这个方案中，存储系统可返回给处理器 $\overline{WAIT}$ 有效信号，处理器在T2周期期间的时钟下降沿检查 $\overline{WAIT}$ 信号的状态。如果 $\overline{WAIT}$ 信号有效，处理器就生成另一个T2时钟并再次进行检查。只要 $\overline{WAIT}$ 信号为低，处理器就保持在T2的标记时间，只有当 $\overline{WAIT}$ 变高处理器才完成该总线周期，这称为**等待状态**（wait state），用于将较慢的存储器与较快的处理器同步。

写周期与读周期类似。在T1中时钟的下降沿期间，地址变为有效。在T2中时钟上升沿期间，要写的数据被放到数据总线上，并且写信号变低，指明这是一个存储器写操作。在写周期，$\overline{WAIT}$ 信号在T2有相同的功能。在T3中时钟下降沿期间，$\overline{WR}$ 信号被取消，给了存储器一个上升沿来存储数据。$\overline{ADDRVAL}$ 也被取消，写周期结束。

在我们继续讨论之前，还有几个有趣的概念需要解释一下，它们在以前的讨论中被忽略了。第一个就是在时钟的两个边沿都操作的状态机的思想，我们先考虑这一点。当我们为了同步其内部操作而给处理器输入单个时钟信号时，我们并不是真正地看到了对于内部时钟发生了什么。很多处理器会在内部将时钟转换成2-**相时钟**（2-phase clock）。图6-24显示了一个2-相时钟的时序图。 150

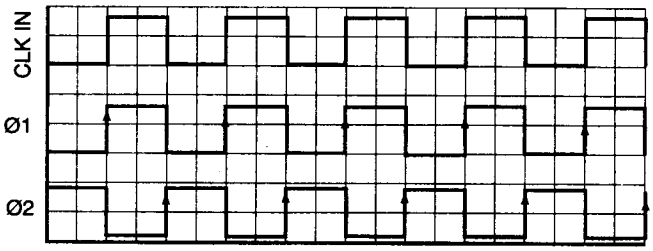


图6-24 一个2-相时钟的时序图

在图6-24中，一个由外部震荡器生成的输入时钟被转换成2-相时钟，标记为φ1和φ2。这两个时钟的相彼此在相位上相差180°，所以CLK IN信号的每个上升沿或下降沿都产生了一个内部的时钟上升沿。我们如何能生成一个2-相时钟呢？你实际上已经知道如何做了，但还有一点知识我们需要知道。图6-25就是能用来产生2-相时钟的电路。

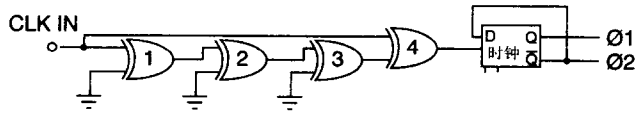


图6-25 一个2-相时钟生成电路

因为通常使用的集成电路部件有在一个封装中包含4个异或门的，所以4个异或门很方便使用。该电路利用了逻辑门固有的传播延迟，假设每个异或门的传播延迟是10ns。假设时钟输入为低，异或门1到3的一个输入永久性地接地（逻辑低）。由于门1的两个输入均为低，其输出也为低。这种情况对门2、3、4也是一样。现在，CLK IN逻辑状态变为高。门#4的输出在10ns后变为高并使D触发器状态翻转。由于Q和 $\overline{Q}$ 输出彼此是相反的，所以我们通过利用D触发器的二分连线的特性就能很便利地拥有两个交替的时钟相位源。

在30ns的传播延迟之后，门#3的输出也变为高，这导致异或门#4的输出再次变低，这是因为若异或门的两个输入相同，其输出就为低，不同则异或门的输出就为高。在后面的某

个时间，时钟输入再次变低，我们就在门#4的输出端生成另一个30ns宽的正脉冲，因为在30ns后，两个输出均不同。这就导致D触发器在时钟的两个边沿都翻转，而Q和 $\overline{Q}$ 输出就给出我们所需要的交替相位。图6-26显示出相关的波形。

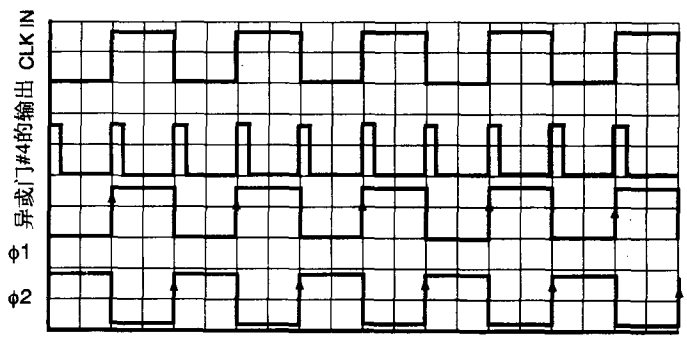


图6-26 一个2-相时钟生成电路的波形

对于任何其周期大于4号异或门延迟的时钟频率，该电路都能工作。而且，通过利用D触发器的两个输出，我们就确保了一个2-相时钟输出，其中两个相位彼此正好相差180度。

151 现在我们就能重新考察图6-23并领会隐藏在图中的另一个微妙的地方了。由于我们明显是在时钟的上升沿和下降沿改变状态的，所以我们就知道处理器的内部状态机实际上使用的是一个2-相时钟，这些“T”状态中的每一个实际上都是两个状态。这样，我们就能将读周期的时序图重画为一个状态图，它能清楚地表明等待状态开始活动的方式。图6-27显示出总线周期中读这一相的状态图表示。

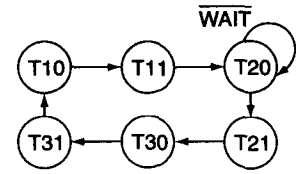


图6-27 一个处理器读周期的状态图

参见图6-27，我们能清楚地看到，在状态T21，处理器测试 $\overline{WAIT}$ 输入的状态。如果该输入为低，处理器就保持在T21状态，有效地延长了总线周期的时间。与降低时钟频率相比，等待状态的优势是我们能将系统设计成只有在处理器访问某存储器区域时才招致等待的惩罚，而不是对所有操作都减慢速度。我们现在将整个总线读周期总结如下：

- T10：读周期开始。处理器为读操作输出新的存储器地址。
- T11：地址此时稳定，$\overline{ADVAL}$ 变为低。$\overline{RD}$ 变为低，指明读周期正在开始。
- T20：读周期继续。
- T21：处理器对 $\overline{WAIT}$ 输入采样。如果有效则T21周期继续。
- T30：读周期继续。
- T31：读周期结束。$\overline{ADVAL}$ 和 $\overline{RD}$ 取消，处理器从存储器输入数据。

6.3 直接存储器访问

我们将以对另一种形式存储器访问的简要讨论结束第6章，该种访问形式称为DMA，即*直接存储器访问*（direct memory access, DMA）。对DMA系统的需要是得自于存储系统和处理器通过总线互连这样一个事实。由于总线是进出系统的唯一路径，当外部设备如硬盘驱动器或网卡携有处理器需要的数据而处理器正忙于执行程序代码时，就会引起冲突。

在很多系统中，外部设备和存储器与处理器共享相同的总线。当一个设备（比如一个硬盘驱动器）需要传输数据到处理器时，我们可以想像出两种情景。

情景#1

1	磁盘驱动器：	“抱歉打扰，老板，我有512个字节给你。”
2	处理器：	“这是一个大10-4的小磁盘伙伴，给我第一个字节。”
3	磁盘驱动器：	“好，老板，给你。”
4	处理器：	“我收到了，给我下一个字节。”
5	磁盘驱动器：	“给你。”

152

对步骤4和步骤5重复进行510次。

情景#2

1	磁盘驱动器：	“嗨，老板，我有512个字节，它们要在我的磁盘上烧一个洞，我要支持不住了，我要支持不住了。”（总线请求）
2	处理器：	“好，好，请住嘴，让我完成这个指令我就放开总线。好，我做完了，总线是你的了，别浪费时间，我很忙。”（总线授权）
3	磁盘驱动器：	“谢谢老板，你是好伙伴，我感谢你，我已经得到总线了。”（总线确认）
4	磁盘驱动器：	“我将把数据放到通常的地点。”（自言自语）
5	磁盘驱动器：	“嘿，老板！醒醒，我要离开总线了。”
6	处理器：	“谢谢磁盘，我会从通常的地点找回数据。”
7	磁盘驱动器：	“通常的地点是10-4，我走了。”

从这两个情景你可以推断出，第二个情景的效率更高，因为外部设备即硬盘能从处理器接管对存储器的控制，并以成组方式写所有的数据。存储器已将其存储器接口置于三态条件中，并等待硬盘驱动器的信号，表明其能归还总线。这样，DMA就允许在处理器空闲时，或者从独立的cache存储器进行处理时，其他设备接管总线并实现到存储器的数据传输或从存储器过来的数据传输。而且，在很多的现代处理器有很大的片上cache的情况下，将外部总线移交给外部设备对于处理器来说几乎没有损失任何东西。让我们记住关于这两个情景的幽默讨论，现在我们严肃一会儿。图6-28显示出简化的DMA过程。你也可以得出结论说我不应该放弃我白天的工作而成为一个系列幽默剧的作者，但那是其他时间要讨论的了。

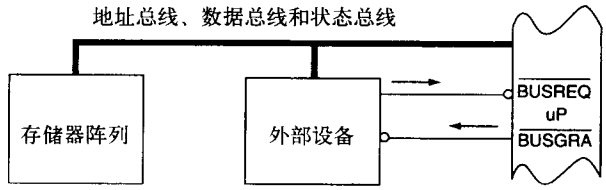


图6-28 DMA传输的示意图表示

简单地说，在处理器和外部设备之间发生的是*握手*（handshake）过程。一次握手就是一个简单的动作，它期望一个回应，表明该动作被接受。该过程可描述如下：

- 外部设备通过向处理器输入有效的总线请求（$\overline{BUSREQ}$）信号，要求从处理器得到对总线的控制。
- 当处理器完成当前指令周期，且没有更高级的中断待解决时，它就发出一个总线授权（$\overline{BUSGRA}$）信号给发出请求的设备，允许其开始自己的存储器周期。
- 处理器然后就处于空闲或者继续处理cache内部的数据，直到BUSREQ信号消失。

总结

- 我们介绍了在一个计算机系统内部进行总线组织的需要，以及总线如何组织成地址总线、数据总线和状态总线。
- 继续展开上一章的讨论，我们看到微代码状态机如何与总线组织一起工作来控制内部总线上的数据流。
- 我们看到三态缓冲电路如何使独立的存储单元能被组织成较大的存储器阵列。 153
- 我们引入分页的概念，作为形成存储地址的途径和建立存储系统的方法。
- 我们介绍了不同类型的现代存储技术，以理解静态RAM技术和动态RAM技术的使用。
- 最后，直接存储器访问是在存储器和外设之间移动大块数据的高效方式。

参考文献

[1] http://www.necel.com/memory/pdfs/M14428EJ5V0DS00.pdf.

[2] http://download.micron.com/pdf/datasheets/dram/sdram/512MbSDRAM.pdf.

154 [3] Ralph Tenny, *Simple Gating Circuit Marks Both Pulse Edges*, Designer's Casebook, Prepared by the editors of *Electronics*, McGraw Hill, p. 27.

习题

1. 给定如下所示的真值表，设计一个2输入、4输出的存储译码器：

输 入			输 出		
A	B	01	02	03	04
0	0	0	1	1	1
1	0	1	0	1	1
0	1	1	1	0	1
1	1	1	1	1	0

2. 参见图6-11。外部输入和输出（I/O）信号定义如下：

A0、A1：地址输入，用于选择对哪一行存储器单元（D触发器）进行读出和写入。

$\overline{CS}$：芯片使能信号。当其变低时，存储器起作用，你可以对其进行读出和写入。

$R/\overline{W}$：读/写线。在高电平时，阵列内适当的一行就被某外部设备读取。在低电平时，某外部设备就可以将数据写到适当的行，这个适当的行由地址位A0和A1的状态定义。

DB0、DB1、DB2、DB3：双向数据位。这4个数据位定义了对适当行进行写入或读出的数据，该适当的行由A0、A1确定。

这个阵列工作如下：

A. 为了从阵列中的特定地址（行）读数据：

a. 将地址放到A0和A1

b. 将 $\overline{CS}$ 置为低

c. 将 $R/\overline{W}$ 置为高

d. 数据在D0..D3可读

B. 为向阵列中的特定地址写数据：

a. 将地址放到A0和A1

b. 将 $\overline{CS}$ 置为低

c. 将 $R/\overline{W}$ 置为低

d. 将数据放置到D0..D3

155 e. 将 $R/\overline{W}$ 置为高

C. 每个单独的存储器单元是一个标准的D触发器，每个单独的存储器单元上还有一个三态输出缓冲器。输出缓冲器由每个触发器上的 $\overline{OE}$ 信号控制。当这个信号为低时，输出就被连接到数据线，存储在触发器中的数据就可读。当这个信号为高时，触发器的输出就与数据线隔离，使得数据可被写入器件。

在存储器阵列图中请看标记为“存储器译码逻辑”的方框。为该电路设计真值表，用K-图对其进行化简，画出门逻辑来实现这个设计。

3. 假设你有一个处理器，具有26位宽地址总线和32位宽数据总线。

a. 假如你正在使用组织成512K深×8位宽（4M位）的存储器芯片。需要多少存储器芯片来为该处理器构建存储系统，使得地址空间完全充满，不留空区域？

b. 假如我们采用512K为一页的大小，对于前三页完成下表：

页号	起始地址（Hex）	结束地址（Hex）

4. 考虑图6-23中的存储器时序图。假设时钟频率是50MHz，而且你不想加入任何等待状态来减慢处理器。在这种情况下对于该处理器来说能正常工作的最慢的存储器访问时间是多少？

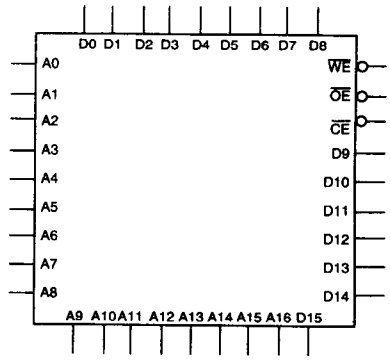
5. 用几句话定义下面的术语：

a）直接存储器访问

b）三态逻辑

c）地址总线、数据总线、状态总线

6. 下图显示的是一个存储器件的示意图，用于一个计算机的存储器系统，其设计规范定义如下：



- 20位地址总线
- 32位数据总线
- 存储器在页0、页1和页7

a. 在每个存储器件中有多少可寻址的存储位置？

b. 在每个存储器件中有多少个存储器位？

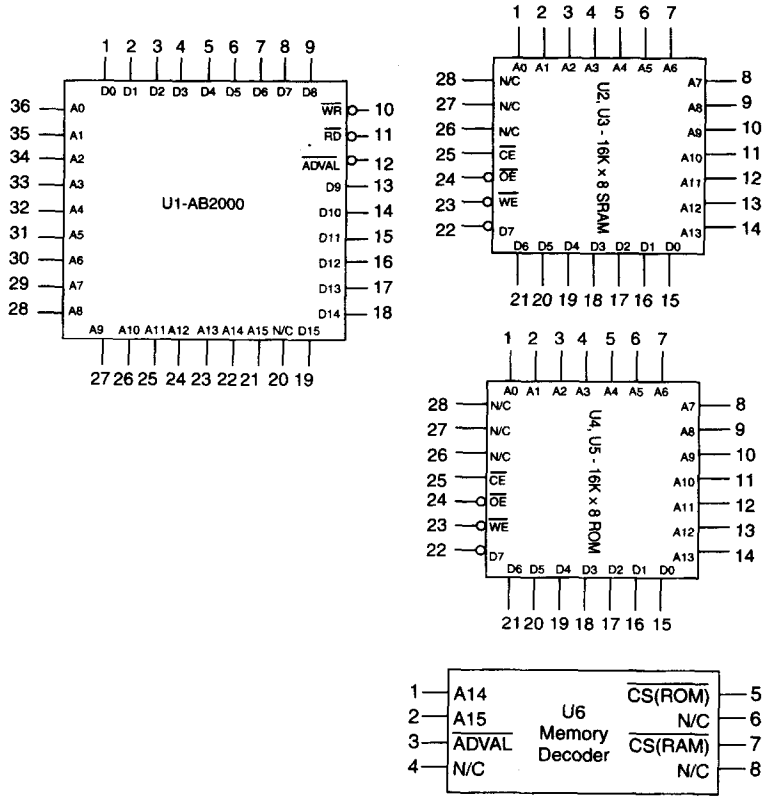
c. 在计算机的地址空间中，每个存储器件所覆盖的十六进制地址范围是什么？你可以假设存储器的每一页大小与一个存储器件的地址范围相等。

d. 在这个存储器设计中，需要的存储器件总数是多少？

e. 为什么基于这种类型存储器件的存储系统设计不能在字节级别对存储位置进行寻址？用一两句话说明原因。 156

7. 假设你是百吉城灵魂和飞行控制系统公司（Soul of the City Bagel and Flight Control Systems Company）的首席硬件设计师。你的工作就是为一个新的、自动化的厨房和百吉饼制造器设计一个计算机到存储器的子系统，这是为正在设计中的下一代商业客机所设计的。微处理器是AB2000，这是一个热门的新芯片，你正等着有机会在设计中使用它。

该处理器有16位地址总线和16位数据总线，如下图所示（为简化，对于这个习题该图只显示了相关的处理器信号）。图中还显示出你在设计中要使用的ROM和SRAM芯片的示意图和引脚指定。



存储器子系统有三个状态信号，你必须使用它们来设计存储器阵列：

a. $\overline{WR}$：低有效，它指出一个向存储器写的周期。

b. $\overline{RD}$：低有效，它指出一个从存储器读的周期。

c. $\overline{ADVAL}$：低有效，它指示地址在总线上稳定，可看作是一个有效地址。

存储器芯片占据下面的存储区域：

a. ROM从0x0000直到0x3FFF

b. SRAM从0xC000直到0xFFFF

c. 其他（不需要你管的）从0x4000直到0xBFFF

1）设计一个你将要用到的存储器译码电路，它必须能为存储器芯片各自的地址范围进行正确的译码，也能对处理器的 $\overline{WR}$、$\overline{RD}$以及 $\overline{ADVAL}$进行译码来产生 $\overline{OE}$、$\overline{CE}$以及 $\overline{WR}$。提示一下，参考课本中关于存储系统的部分就可得到 $\overline{OE}$、$\overline{CE}$ 以及 $\overline{WR}$的方程。

假设你设计的电路将要被制作进一个存储器接口芯片U6中。由于印刷电路设计者需要知道所有部件的引脚指定，你就需要为图中U6提供引脚函数的规范。然后，你就对这些引
157 脚指定设计存储器译码器。最后，为简化，你决定不实现字节可寻址性，所有的存储器访问都将是字宽度的访问。

2）为你的设计生成一个网表。一个示例网表如下图所示。网表就是电路互连或“线网”的表。一个线网就是在电路封装上的所有要连接在一起的I/O引脚的共同连接。这样，在这个设计中，你可能有一个标记为“ADDR13”的线网，它是芯片U1上的引脚#23（简写为U1-23）、U2-14、U3-14、U4-14、U5-14的共同连接。网表由印刷电路板制造商用来实际地制造PC板。网表开始部分如下所示：

线网名称					
addr0	U1-36	U2-1	U3-1	U4-1	U5-1
addr1	U1-35	U2-2	U3-2	U4-2	U5-2
addr2	U1-34	U2-3	U3-3	U4-3	U5-3
addr3	U1-33	U2-4	U3-4	U4-4	U5-4
•					
•					
data0	U1-14	U2-15	U4-15		

8. 对下列的每一种情况完成分析：

a. 一个微处理器，有20位的地址范围，采用8个存储器芯片，每个芯片的容量是128K：建立一个表，以十六进制显示出每个存储器芯片所覆盖的地址范围。

b. 一个微处理器，有24位的地址范围，采用4个存储器芯片，每个芯片的容量是64K：两个存储器芯片占据地址范围的前128K，两个芯片占据地址空间顶端的128K。建立一个表，以十六进制显示出每个存储器芯片所覆盖的地址范围。注意在中间有一个大的地址范围，其中有没有起作用的存储器，这是相当正常的情况。

c. 一个微处理器，有32位的地址范围，采用8个存储器芯片，每个芯片的容量是1M（1M＝1 048 576）：两个存储器芯片占据地址范围的前2M，6个芯片占据地址空间顶端的6M。建立一个表，以十六进制显示出每个存储器芯片所覆盖的地址范围。

d. 一个微处理器，有20位的寻址范围，采用8个不同大小的存储器芯片。4个存储器芯片每个有128K的容量，并占据着从00000起始的512K连续地址。其他4个存储器芯片每个有32K的容量，并占据着寻址范围最顶端的128K。建立一个表，以十六进制显示出每个存储器芯片所覆盖的地址范围。 158

第 7 章　存储器组织和汇编语言编程

学习目标

- 描述一个典型的存储系统是如何根据存储地址模式进行组织的；
- 描述计算机的指令集体系结构与其汇编语言指令集的关系；
- 采用简单的寻址模式写一个简单的汇编语言程序。

7.1　引言

从这一章起，我们将从硬件设计者转回到软件工程师。我们将利用迄今所学到的关于硬件特性的知识，考察它们如何与一个典型微处理器的指令集体系结构（ISA）相联系。我们对体系结构的学习将从一个软件开发者的观点出发，其中软件开发者需要理解体系结构以便将其优势发挥到最佳。在这种意义上，我们学习汇编语言其实是从另一种途径来学习计算机体系结构。这个观点与某些想有能力设计计算机硬件的人的观点截然不同。本书关注的是对硬件和体系结构问题的理解，使得我们的软件开发能最大限度地发挥硬件设计的优势。

我们首先考察Motorola 68000处理器的体系结构。68K的ISA是一个很成熟的体系结构，产生于20世纪80年代早期。你很显然会问："我为什么要学这样古老的计算机体系结构？它难道不是过时了吗？"回答是响亮的："不！"。68K体系结构一直是最流行的计算机体系结构之一，68K的衍生技术在新产品设计中也仍有应用，掌上PDA还在使用由68K衍生的体系结构。而且，Motorola的一款全新的处理器ColdFire系列就采用了68K体系结构，目前，它是最常使用的处理器之一。例如，ColdFire用在很多喷墨打印机中。

在接下来的章节中，我们还将看到另外两种流行的体系结构，即Intel x86处理器系列和ARM系列。熟悉了这三种微处理器体系结构，我们就熟识了当今世界上最流行的三种体系结构。当开始我们的探讨时，我们所讨论的很多东西对这三种体系结构来说都是共同的，所以当以理解基本原理为目的时，学习某一种体系结构与学习其他种体系结构相比，效果是一样的。

从68000开始学习的另一个原因是该体系结构对学习过程有帮助。它的存储器寻址模式简单易懂，从软件开发者的角度来看，与高级语言相联系更易理解。

我们将再次考察存储器模型，并将此作为起点来学习计算机如何实际地对存储器进行读和写。从硬件角度，我们已经知道这些原理，因为我们刚刚完成对它的学习。但是，我们现在要从ISA的角度来看待这个问题。在这个学习过程中，我们将会看到为什么这个看起来有些奇怪的看待存储器的方式，实际上从高级语言如C和C++的角度来看却是非常重要的。女士们和先生们，让我们开始吧……

存储器规约

当你开始从体系结构级别来审视一个计算机时，你很快就会意识到，处理器和存储器的关系是定义机器的行为和操作特性的关键因素之一。实际上，关于计算机如何围绕它到存储器的接口循环反复以完成其编程任务已经有很多事例了。当我们开始用汇编语言编程时，你很快就会看到，计算机所做的大多数工作都涉及将数据移进和移出存储器。事实上，无论原始程序是用C++编写的，还是用汇编语言编写的，如果制作一个在程序中使用的汇编语言指令数的条形图，你将会发现使用的大多数指令是移动指令。因此，我们最先要做的事情之一

就是理解存储系统是如何组织的，并且从处理器的角度来看存储系统。

图7-1就是68K处理器的一个相当典型的存储器布局图。处理器可对16 777 216（16M）字节的存储器进行寻址，它有23条外部的字地址线和2-字节的选择控制信号（用于在存储于字位置的两个字节中选择其一）。外部数据总线是16位宽的，但所有内部数据通路都是32位宽的。从68020开始的68K系列的后续成员都有32位的外部数据总线。在图7-1中，我们看到存储器的前128K的字（16位）占据了从0x000000到0x01FFFE的地址范围，这些字一般是程序代码和向量表，向量表占据了存储器的前256个长字，这就是为什么非易失性只读存储器占据较低地址范围的原因。较上面的地址范围通常包含RAM存储器，这是易失性的读/写存储器，存储的是变量。其余的存储空间包含了I/O设备的地址。图7-1中所有剩下的存储空间都是空的。而且，我们很快将会看到，为什么ROM和RAM的最后一个字的地址分别是0x01FFFE和0xFFFFFE。 160

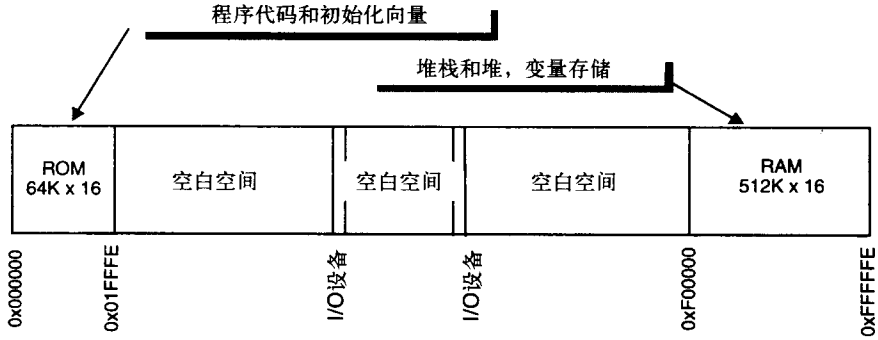


图7-1 一个基于68K的计算机系统的存储器布局图

由于8位（即一个字节）是我们通常要处理的最小数据量，所以当我们要打交道的存储器也是8位宽时，在存储器中存储字节大小的字符（char）就会很方便。然而，我们的PC都有32位宽的存储器，现今正在使用的很多计算机也有16位宽的数据通路。作为一个极端的例子，我们不想仅仅因为我们想将一个8位的量存到一个32位存储器中就浪费24位的存储空间，所以计算机就设计成允许按字节寻址。可以将字节寻址看作是分页的一个例子，页地址是32位字，偏移是页中的4种可能的字节位置。然而，字节寻址和真实的分页之间有一个很大的不同，对于字节寻址，我们没有一个对应于页地址的单独的字地址。图7-2显示出了这个重要的不同点。

长字地址

000000	字节0－地址000000	字节1－地址000001	字节2－地址000002	字节3－地址000003
000004	字节0－地址000004	字节1－地址000005	字节2－地址000006	字节3－地址000007
000008	字节0－地址000008	字节1－地址000009	字节2－地址00000A	字节3－地址00000B
00000C	字节0－地址00000C	字节1－地址00000D	字节2－地址00000E	字节3－地址00000F
000010	字节0－地址000010	字节1－地址000011	字节2－地址000012	字节3－地址000013
⋮				
FFFFF0	字节0－地址FFFFF0	字节1－地址FFFFF1	字节2－地址FFFFF2	字节3－地址FFFFF3
FFFFF4	字节0－地址FFFFF4	字节1－地址FFFFF5	字节2－地址FFFFF6	字节3－地址FFFFF7
FFFFF8	字节0－地址FFFFF8	字节1－地址FFFFF9	字节2－地址FFFFFA	字节3－地址FFFFFB
FFFFFC	字节0－地址FFFFFC	字节1－地址FFFFFD	字节2－地址FFFFFE	字节3－地址FFFFFF

图7-2 在一个32位宽的字中，字寻址和字节寻址的关系

我们称这种类型的存储器存储为字节包装（byte packing），因为我们实际上是将充满字节的32位存储器包装起来。这种类型的寻址引起了若干的含糊性，有一个相当严重，其他的则对我们来说纯粹是新的。我们立刻讨论这个严重的问题，参见图7-2，我们看到在存储器地址FFFFF0（字符）的字节和在FFFFF0的32位长字（整数）有相同的地址。这个灾难会发生吗？答案是绝对“可能”。例如在C和C++中，在使用一个变量之前，你必须声明该变量及其类型，现在你明白其中的原因了吧。除非编译器知道存储于地址FFFFF0的变量的类型，否则它不知道必须生成什么类型的代码来对其操作。而且，如果你想在FFFFF0存储一个字符，则编译器必须知道为其分配多少存储空间。

此外，还要注意的是，在不能被4整除的地址上，我们不能访问32位字。一些处理器允许我们在奇数边界存储32位值，比如字节地址000003－000006，但其他很多的处理器都不允许。其中的原因是处理器将不得不做若干次附加的存储器操作来读取全部的值，然后再做一些额外的工作来以正确的次序重建这些字节，我们称此为非对齐访问（nonaligned access），若允
161 许其发生，则通常在处理器性能方面的成本相当高。事实上，如果你考察存储器映像，了解编译器在存储器中是如何存储对象和结构的，那么你就会常常看到数据中的空闲区，它们所对应的是有意留出的间隙，是为了不产生包含非对齐访问的数据区域。

还要注意，当我们进行32位字的访问时，地址位A0和A1没有被使用。这不禁使你要问：“我没有使用它们，它们有什么用呢？”然而，当需要访问一个32位字内的特定字节时，我们确实需要它们。A0和A1经常被叫做字节选择线（byte selector），因为这是它们的主要功能。另一点是当我们在向存储器写的时候，我们确实需要字节选择线。从存储器读还算无害，但写会改变所有东西。因此，你就想确保你只改变了感兴趣的字节而不是其他字节。从一个硬件设计者的角度，有字节选择线就使你能限定写操作只针对你感兴趣的字节。

很多处理器根本没有明确的字节选择线，而是在状态总线上提供信号用来限定对存储器的写操作。将16位数（一种短数据类型）存储到32位存储器位置会怎样呢？相同规则也同样适用于这种情况。有效的地址只是那些能被2整除的地址，如000000、000002、000004，等等。在16位字寻址的情况下，不需要最低地址位A0。对于68K处理器，有到存储器的16位宽数据总线，我们能在存储器的每个字中存储两个字节，所以A0没被用于字寻址，而是成了处理器的字节选择线。

图7-3显示出一个典型的32位处理器及其存储系统接口。为清楚起见，来自处理器的读信号和片选信号被省略了。处理器有32位数据总线和32位地址总线。存储器芯片代表了处理器地址空间某处的一页RAM，确切的存储页号由地址译码逻辑块的设计所决定。每个RAM芯片有1M位的容量，组织成128K×8的形式。

由于我们有32位宽数据总线，而每个RAM芯片有8条数据I/O线，所以对于每个128K宽的
162 页就需要4个存储器芯片。芯片#1连接到数据线D0至D7，芯片2连接到数据线D8至D15，芯片3连接到数据线D16至D23，芯片4连接到数据线D24至D31。

来自处理器的地址总线包含30条地址线，这意味着可寻址2^{30}长的字（32位宽）。寻址全部2^{32}字节空间所需要的另外的地址位隐含地由处理器控制，显式的则由4个写使能信号来控制，标记为 $\overline{WE0}$ 至 $\overline{WE3}$。

来自处理器的地址线A2到A18连接到RAM芯片的地址输入A0到A16，来自处理器的A2连接到4个芯片中的每个芯片的A0，依此类推。这开始看起来有点古怪，但你仔细考虑后就会明白了。事实上，没有特殊的理由要求来自处理器的每个地址必须连接到每个存储器件的相同地址输入引脚。例如，来自处理器的A2可连接到芯片＃1的A14、芯片＃2的A3和芯片＃3的

A16。来自处理器的相同地址显然在寻址4个存储器芯片的不同字节位置，但只要来自处理器的17条地址线连接到存储器件的17条地址线，存储器就应该能正确地工作。

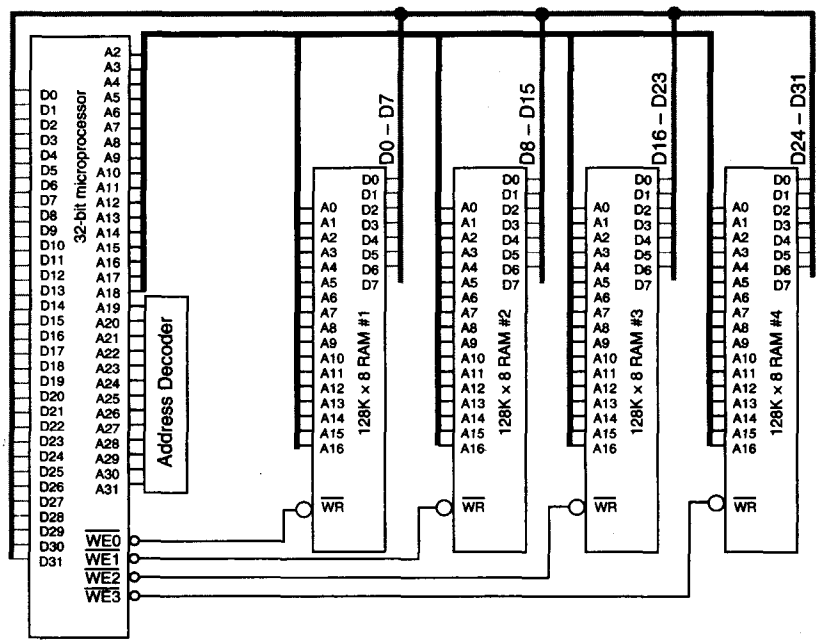


图7-3 一个32位微处理器的存储器组织。为清楚起见，将片选信号和读信号省略了

来自处理器的高地址位A19到A31用于页选择过程，这些信号被发送到地址译码逻辑来生成适当的 *CHIP SELECT* 信号。在图7-3中略去了这些信号。在本例中有13个高地址位，这些地址位给出了2^{13}即8 192页的存储空间，每一页占据128K的32位宽的字，合计2^{30}个长字。根据地址计算，每一页实际包含512KB，所以每一页的字节地址范围就是从字节地址（用十六进制表示）00000直到7FFFF。记住在这个存储方案中有8 192页，每页占据512KB，这样，页地址就是从0000到1FFF。对你来说，页地址和偏移地址的相互关系可能看起来不那么明显，但是如果你将十六进制地址扩展成二进制，并将其依次排列，你就会看到全部的32位地址。

现在，让我们看看处理器如何处理小于长字的数据。假设处理器要从地址ABCDEF64中读一个长字，而这个地址被译码到图7-3的页上。这个地址在32位边界上，A0和A1都等于0且未用作进入存储器的外部地址的一部分。然而，如果处理器想对位于地址ABCDEF64或地址ABCDEF66的两个字之一做一次字访问，那么它仍将生成同样的外部地址。当数据被读进处理器时，长字中不需要的那1/2就会被抛弃掉。由于这是一个读操作，存储器的内容不会受影响。

如果处理器想读位于地址ABCDEF64、ABCDEF65、ABCDEF66或ABCDEF67中的4个字节中的任何一个，那么它仍然会像以前一样执行相同的读操作，这时它还是只保留感兴趣的字节而抛弃其他字节。

现在，让我们考虑一个写操作。在这种情况下，我们关心的是可能破坏存储器的内容，所以当我们向存储器写一个小于一个长字的量时，我们想确保不会意外地写入更多的量。我们假设只想在存储器位置ABCDEF65写一个新值，在这种情况下，信号 $\overline{WE1}$ 就必须有效，所以只修改那个字节地址1位置的数据。这样，要向存储器写一个字节，我们只要使4个写使能信号之一有效即可。要向存储器写一个字，我们就要使 $\overline{WE0}$ 和 $\overline{WE1}$ 一起有效，或者 $\overline{WE2}$

163

和$\overline{WE3}$一起有效。最后，要写一个长字，所有4个写使能线均应有效。

将32位字存到16位存储器的情况又怎么样呢？在这种情况下，32位字可被存到任意偶数字边界上，因为处理器必须总是进行两次连续的存储器访问才能得到全部32位的量。然而，多数的编译器还是试图将32位字存到自然边界上（可被4整除的地址）。这就是为什么汇编语言程序员常常能节省一点空间或能加速算法的原因，汇编语言程序员无视编译器生成代码的方法，而是对其进行调整以获得更高的效率。

让我们再回到原来的话题。对于一个32位处理器，地址位A2...A31用于寻址1 073 741 824个可能的长字，而A0...A1用于在长字内寻址4个可能的字节，这就在32位处理器中给予我们总共4 294 967 296个可寻址的字节位置。换句话说，我们有4GB的字节寻址空间。对于一个像68K这样有16位宽数据总线的处理器，地址线A1-A23用于字寻址，而A0用于字节选择。

结合以上所有这些，你就会看到问题所在。你可能在相同的地址上有8位的字节、16位的字或32位的长字，这样不会造成含义模糊吗？确实会。当用高级语言编程时，我们依靠编译器来保存这些杂乱的细节，这就是为什么将一种变量类型强制转换（casting）成另一种类型会如此地危险。当我们用低级语言编程时，我们就依赖于程序员的技巧来保存这些细节。

看起来很容易，但事实并非如此。这个计算机生活中的小小事实就是软件故障的主要原
164 因。一个简单的概念怎么会如此复杂呢？这不是复杂，而只是含糊。图7-4阐明了这个问题。图7-4中最左一列显示了一个字符串（适当地命名为“string”），存储于8位存储器空间中。每个ASCII字符占据连续的存储位置。

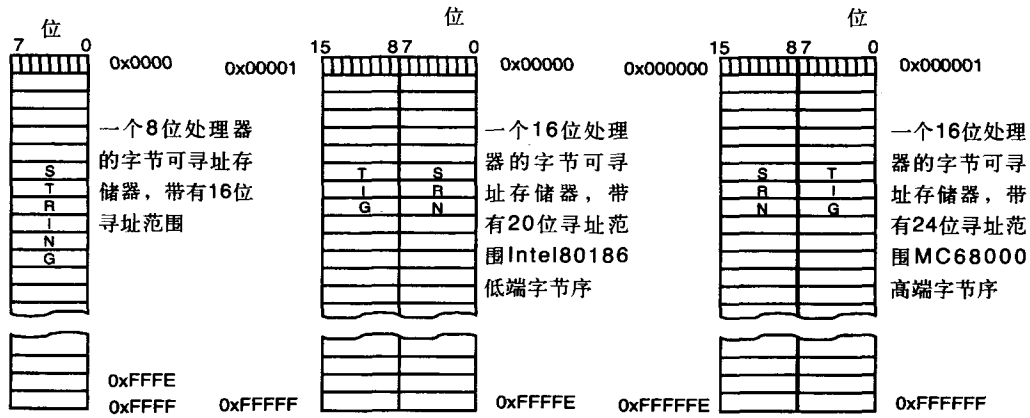


图7-4 将字节组装成16位存储器字的两种方法。将低序字节放到字的低序端称为低端字节序。将低序字节放到字的高序端称为高端字节序

中间一列显示了一个16位存储器，其组织方式使得连续的字节从右向左存储，对应于A0=0的字节与16位字的低序部分DB0...DB7对齐，对应于A0=1的字节与16位字的高序部分DB8...DB15对齐，这称为低端字节序（Little Endian）组织。最右一列以从左到右的方式将这些字符存储成了连续的字节，对应于A0=0的字节位置与16位字的高序部分对齐，这称为高端字节序（Big Endian）组织。作为一个习题，请看图7-2中字节是如何存储的，它们是高端字节序还是低端字节序？

Motorola和Intel选择使用了不同的字节序规约，为编程世界打开了潘多拉的盒子。这样，为一种规约所写的C或C++代码当被转向另一种规约时，就会有微妙的错误。还有更坏的情况。当设计意图所使用的规约被假设成另外一种规约时，相互进行项目合作的工程师们就会曲解

设计规范。ARM体系结构允许程序员在上电启动时就确立处理器采用哪种字节序，因此，虽然ARM处理器既能处理高端字节序也能处理低端字节序，但一旦字节序确立，它就不能动态地切换字节序模式。图7-5针对一个用4个字节打包的32位字显示了两种规约的不同。

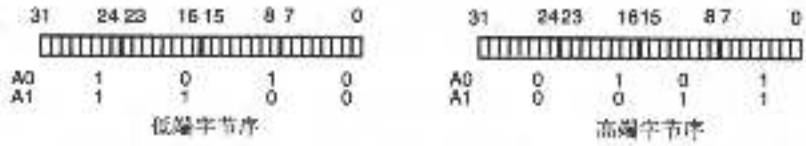


图7-5 以低端字节序和高端字节序对32位字进行字节打包

如果你想从本课本学到一些东西，就可以考虑这个问题，因为在你作为软件开发者的生涯中，你至少会遇到一次这个问题。

在你指责我死揪住这个题目不放之前，让我们再从硬件的角度考虑一下这个问题。整个存储器寻址区域对于新手程序员和经验丰富的老手都是容易搞糊涂的，而且，还存在体系结构和制造商的技术所引入的含糊性。所以，现在让我们看一下对68K Motorola是如何处理这个问题的，也许这会帮助我们更好地理解真实情况，至少是对于Motorola处理器的情况，虽然我们以前在图7-3中已经接触过这个问题。图7-6总结了68K处理器的存储器寻址方案。

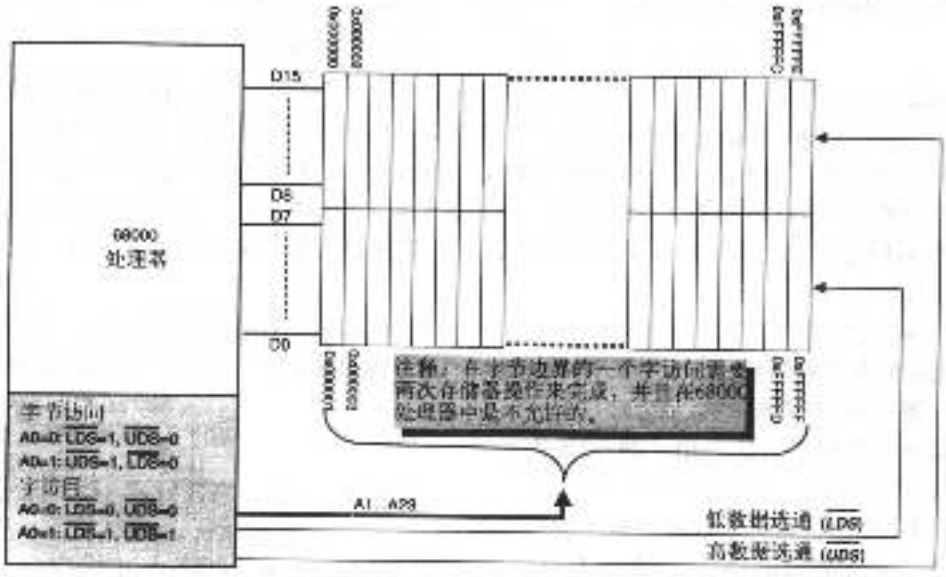


图7-6 Motorola 68K处理器的存储器寻址模式

68K处理器能直接寻址16MB的存储器，需要24条“有效的”地址线，为什么？是因为2^{24} = 16 777 216。在图7-6中我们看到23条地址线，那条失踪了的线A0是由另外两个控制信号 $\overline{LDS}$ 和 $\overline{UDS}$ 综合出来的。

对于一个16位宽的外部数据总线，我们通常将A0作为字节选择线。当A0为0时我们就选择了偶数字节，而A0为1时我们就选择了奇数字节。68K的字节序（endianness）是高端字节序，所以偶数字节与数据总线的D8到D15是对齐的。参见图7-6，我们看到从处理器实际发出了两个总线信号，分别为 $\overline{UDS}$（Upper Data Strobe）和 $\overline{LDS}$（Lower Data Strobe）。

165 当处理器在对存储器做一个字节的访问时，或者 $\overline{LDS}$ 有效或者 $\overline{UDS}$ 有效，向存储器指明正在访问字的哪个部分。如果在偶数地址的字节正在被访问（A0＝0），则 $\overline{UDS}$ 有效而 $\overline{LDS}$ 保持为高，即关闭状态。如果奇数字节正在被访问（A0＝1），则 $\overline{LDS}$ 有效而 $\overline{UDS}$ 保持为高。对于一个字访问，$\overline{UDS}$ 和 $\overline{LDS}$ 都有效。这种行为在图7-6中的表中进行了总结。

你也许通常会将 $\overline{LDS}$ 和 $\overline{UDS}$ 用作到存储器控制系统的门控信号。例如，你可能会采用图7-7显示的电路来控制向哪个字节写。

你可能会为这个电路而抓耳挠腮。为什么要使用或门？我们可以通过两种途径来回答这个问题。首先，由于所有信号都是低有效的，所以我们实际上所处理的是与函数的负逻辑等价的函数。与门的负逻辑等价的函数就是或函数，因为输出为0当且仅当两个输入均为0。

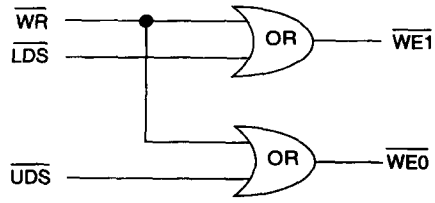


图7-7 控制68K处理器字节写的简单电路

第二种途径是通过德摩根定理的等价方程。回忆一下：

$$\overline{(A*B)} = \overline{A} + \overline{B} \quad (1)$$

$$\overline{(A+B)} = \overline{A} * \overline{B} \quad (2)$$

这样，方程1显示出 $\overline{A}$ 和 $\overline{B}$ 的或就等价于使用正逻辑信号*A*和*B*的与非。

166 现在，假设你试图在奇数地址做一次字访问。例如，假设你写出了下面的汇编语言指令：

```
move.w  D0, $1001      * 这是一个非对齐访问!
```

这条指令告诉处理器对存于内部寄存器D0中的字做一个拷贝，并将该字的拷贝存入到外部存储器中地址$1001开始的地方。因为字节序要求访问两个存储器位置才能正确地安置这些字节，所以处理器需要执行两个存储器周期来完成访问。一些处理器有能力做这种类型的访问，但是68K是没有这种能力的处理器之一。如果发生了非对齐访问，处理器将产生异常并自动转移到某种用户定义的代码（但愿编程者已考虑到这种情况）来试图处理错误，或者至少优雅地中止。

由于68K处理器内部是32位宽的，只是有一个到外部存储器的16位宽数据总线，我们就需要知道它是如何也在外部存储器中存储32位量的。图7-8为我们展示了在存储器中存储长字的规约。

虽然图7-8可能看起来使人迷惑，但它只是以某种更简略的方式重申了图7-2的内容。该图告诉我们，被称为长字（long或long word）的32位数据量在存储器中的存储方式是：最高有效的16位（D16－D31）存于第一个字地址位置，而最低有效的16位（D0－D15）存于下一个最高的字位置。而且，偶数字节地址与16位字地址的高序是部分对齐的（高端字节序）。

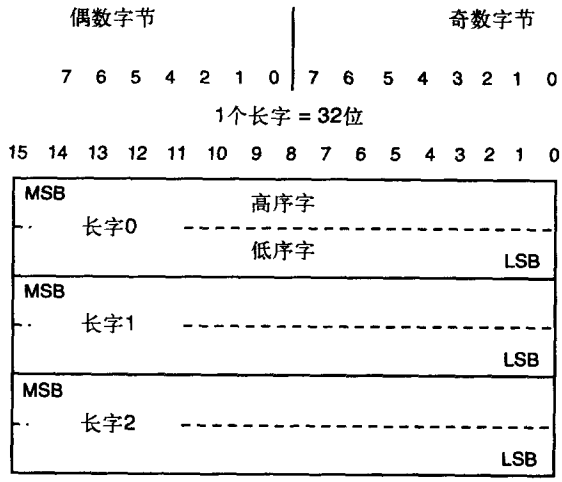


图7-8 Motorola 68000处理器的存储器存储规约

汇编语言介绍

PC世界是由一个指令集体系结构（ISA）所支配的，这个ISA是Intel在25年前首先定义的，称为x86，这是因为这个家族

的成员有如下的血统关系：

8080 → 8086 → 80186 → 80286 → 80386 → 80486 → 奔腾

嵌入式微处理器的世界（就是在如手机这样的设备内部的单用途的微处理器）由Motorola 680x0 ISA所支配：

68000 → 68010 → 68020 → 68030 → 68040 → 68060 → ColdFire

CodeFire将一种称为RISC的现代处理器体系结构以向后兼容的方式与原始68K ISA联合起来。（我们将在后面课程中学习这些体系结构。）向后兼容非常重要，因为还有如此多的68K代码广泛存在并被使用着。Motorola 68K指令集是最多被学习的ISA之一，如果你用“68K”或“68000”做一个Web搜索，你会发现符合的数量之多令人难以置信。

每个计算机都有一个其所能执行的基本操作的集合，这些操作由处理器的指令集（instruction set）来定义。一个特定的指令集的存在是由计算机的内部组织和操作设计的方式所决定的，这就是我们所称的计算机体系结构。体系结构从其所能执行的汇编语言指令反映出来，因为这些指令就是我们访问计算机资源的机制。指令集是处理器的原子要素，所有复杂操作都是通过构建这些基本操作的序列来实现的。 167

计算机不理解汇编语言，而是接受机器代码的指令。如你所知，机器代码定义了到状态机微代码表的进入点，从此点开始指令执行过程。汇编语言是这些机器语言指令的人可读（human-readable）的形式。机器语言指令没有什么神秘的，很快你就会理解它们并看到那些引导现代微处理器内部状态机的模式。现在，我们将关注学习汇编语言这个任务，见图7-9。

机器语言程序如下：

```
00000412 307B7048
00000416 327B704A
0000041A 1080
0000041C B010
000004E 67000008
00000422 1600
00000424 61000066
00000428 5248
0000042A B0C9
```

我们用汇编语言将程序写成：

```
MOVEA.W  (TEST_S,PC,D7),A0     * 我们将采用间接寻址
MOVEA.W  (TEST_ E , PC, D7), A1  * 得到结束地址
MOVE.B   D0, (A0)              * 写这个字节
CMP.B    (A0), D0              * 测试
BEQ      NEXT_LOCATION         * 正常，继续进行
MOVE.B   D0, D3                * 拷贝坏数据
BSR      ERROR                 * 坏字节
ADDQ.W   #01, A0               * 地址递增
CMPA.W   A1, A0                * 我们做完了吗?
```

图7-9 右边方框中是用汇编语言写的68K代码片段，左边方框中是等价的机器语言代码

注意，我们在C或C++中用前缀“0x”表示十六进制数，这是该语言的标准化。在汇编语言中则没有相应的标准，不同的汇编器开发者用不同的方式表示十六进制数。在本课本中，我们将采用Motorola的惯例，即用“$”作为十六进制数的前缀。

机器语言代码实际上是汇编程序的输出。汇编程序将你用文本编辑器写成的汇编语言源代码转换成相应的能在680x0处理器上运行的机器语言代码。左边方框实际上有两列，虽然辨别出来有点困难。左列从十六进制存储器位置开始，该处存储了机器语言指令。在本例中，存储器位置$00000412存有机器语言指令代码0x307B7048。下一条指令开始于存储器位置0x00000416，包含的指令代码是0x327B704A。这两条机器语言指令是由下面两条汇编语言指令给出的：

```
MOVEA.W (TEST_S,PC,D7),A0
MOVEA.W (TEST_E,PC,D7),A1
```

很快你就会看到这些指令的实际意思。眼下，我们可这样总结上述讨论：

168

- 机器指令代码$307B7048开始于存储器位置$00000412，运行通过位置$00000415。对应该机器语言指令的汇编语言指令是**MOVEA.W (TEST_S, PC, D7), A0**。
- 机器指令代码$327B704A开始于存储器位置$00000416，运行通过位置$00000419。对应该机器语言指令的汇编语言指令是**MOVEA.W (TEST_E, PC, D7), A1**。

此外，对于68K指令集，最小的机器语言指令是16位长（4位十六进制数字）。虽然一些指令长到5个16位长，但没有比16位更短的指令。

在汇编语言指令和机器语言指令之间有一个1对1的对应关系。汇编语言指令称为助记法（mnemonics），它们为该指令实际是做什么的提供速记形式的线索。例如：

MOVE.B	移动一个字节的数据
MOVEA.W	将一个字的数据移动到地址寄存器
CMP.B	比较两个字节数据的大小
BEQ	如果结果等于零，就转移到一个不同的指令
ADDQ.W	将两个值（快速）相加
BRA	总是转移到一个新位置

你会注意到，我为汇编语言指令选择了一个不同的字体。这是因为具有固定间隙的字体（比如“Courier”）能使字符保持列对齐，使得汇编语言指令更易读。没有规定你必须用这种字体，汇编器对此可能不在意，但如果你这样做，就会便于你阅读和理解程序。

指令中告诉计算机做什么的部分称为操作代码（opcode，“operation code”的缩写），该部分只占指令的一半，而另一半则是告诉计算机如何执行该操作和在哪里执行该操作。实际的操作代码（例如MOVE.B）实际上包括一个操作代码和一个修饰符。操作代码是MOVE，它是说一些数据应从一个地方移动到另一个地方。修饰符是“.B”后缀，它说的是要移动一个字节的数据，而不是一个字或长字。为了完成该指令，我们必须告诉计算机：

1. 从哪里找到数据（称为操作数1）？
2. 将结果放到哪里（操作数2）？

一个完整的汇编语言指令必须有一个操作代码，并可能有0、1或2个操作数。请看这条称为NOP（读作No op）的汇编语言指令，它的意思是不做任何事情。你可能会怀疑这条指令不正常，但它实际上是相当有用处的，编译器很好地利用了这条指令，NOP指令就是指令中需要0个操作数的例子。

指令CLR.L D4是指令中需要1个操作数的例子，它的意思是将内部寄存器的所有32位（.L修饰符）清零，即置为零。

指令MOVE.W D0，D3是需要两个操作数指令的例子。注意用逗号分隔两个操作数D0和D3。指令告诉处理器将16位的数据（“.W”修饰符）从数据寄存器D0移动到数据寄存器D3。该操作不改变D0的内容。所有的汇编语言程序都遵从下面的结构：

169

第1列	第2列	第3列	第4列……
标号	操作代码	操作数1，操作数2	* 注释

每条指令占用一行的文本，开始于第1列，最多可达132列。

1. 标号域是可选的，但必须总是开始于一行的第1列。我们很快就会讲到如何使用标号。
2. 接着就是操作代码。它必须用TAB字符或若干个空格的空白区与标号分隔，而且必须

从第2列或更后面的列开始。

3. 下面就是操作数，用空白区（通常用tab字符）与操作代码分隔。两个操作数应该用逗号分隔开。操作数和逗号之间没有空白区。

4. 注释是一行的最后一个域。它通常开始于一个星号或分号，这取决于使用哪种汇编器。你也可以有注释行，但这时星号必须在第1列。

7.2 标号

虽然标号是可选择的，但它是汇编语言编程中的一个非常重要的部分。当你给一个变量或常量一个符号名时，你就已经知道如何使用标号。你也可以用标号为函数命名。在汇编语言中，我们通常使用标号来指定程序中一个指令或数据的存储器地址。标号必须在程序的第一列中定义。标号使程序更易读懂，写程序不要标号也行，但几乎没有人这样做。标号使汇编程序能自动（并正确地*）计算操作数和目地地址。例如，考虑下面图7-10的代码片段。

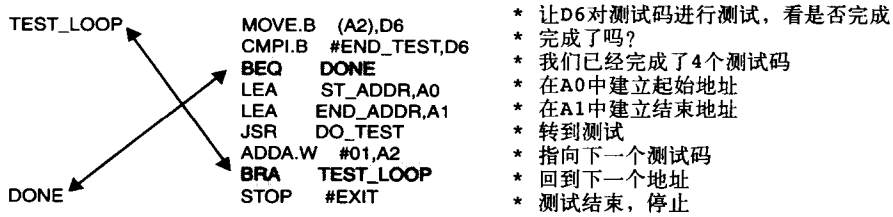


图7-10 说明标号用途的汇编语言代码片段

图7-10中的代码例子有TEST_LOOP和DONE两个标号，这两个标号分别对应于指令“MOVE.B (A2), D6”和“STOP #EXIT”的存储器位置。当汇编程序将汇编语言指令转换为机器语言指令时，它会记住每条指令在存储器中的位置。当它遇到一个标号作为操作数时，就将标号文本替换为数字值。这样，指令“BEQ DONE”就告诉汇编器计算必要的数字值，使得当测试条件（即相等性）满足时，促使程序跳转到对应于标号“DONE”的存储器位置的指令。我们很快就会看到如何进行相等测试。如果测试失败，就忽略转移指令而执行下一条指令。

注释

在更深入地学习之前，我们需要提一些建议，使你采取适当的形式为汇编语言程序做注 170
释。如你在图7-10中所见到的，每条汇编语言指令都有一个注释与其关联。不同的汇编器以不同的方式处理注释，一些汇编器要求注释单独在一行上，并用一个星号“*”或分号“;”打头。与指令或汇编伪指令关联的注释可能要用分号或星号作为注释的开始，或者也许不需要任何特殊的字符，因为起始的空格就定义了注释块的位置。重要的一点是汇编语言难以从代码本身推测程序的意义，在一个星期左右过去后，你很容易忘记你的那个算法究竟要用来做什么。

汇编代码应注释丰富，不仅为了使你自己清楚，而且也为在你离开后其他要维护代码的人提供方便。汇编代码不能做到与有很好文档的C++程序一样易读是没有理由的。使用等于伪指令（equate）和标号可用来消除难理解的数字和帮助解释代码在做什么，使用注释块可解

释算法的什么部分在进行和做了什么假设。最后，对每条指令或者小指令段进行注释是为了对正在发生的事情做到绝对清楚。

Steven Levy[1]在他的《黑客：计算机革命的英雄》一书中，这样描述一位MIT的学生和早期程序员Peter Samson的编码风格：

...Samson虽然极其顽固地拒绝在其源代码中加入注释，但也解释在给定时间他在做什么。一个Samson编写的发行很好的程序有几百行汇编语言指令，只在一条包含有数字1750的指令旁边有一个注释。这个注释是“RIPJSB”，人们为弄清其含义绞尽脑汁，直到一个人想到1750年是巴赫逝世年，Samson的注释是：约翰·塞巴斯蒂安·巴赫安息吧（Rest In Peace Johann Sebastian Bach）。

程序员模型体系结构

为了用汇编语言编程，我们必须熟悉处理器的基本体系结构。我们对体系结构的观点称为处理器的程序员模型（Programmer's Model）。我们必须理解体系结构的两个方面：

1. 指令集
2. 寻址模式

计算机的寻址模式描述了其访问操作数或取回要操作数据的不同方式。寻址模式还描述了操作完成后对数据做什么。对于如转移或跳转到新地址这样的非顺序的取指令，寻址模式也告诉处理器如何计算目地地址。寻址方式对于理解计算机是如此重要，所以在我们能编写汇编语言程序之前需要学习一点这方面的知识。与C、C++或JAVA不同，在能够实际编写一个汇编程序之前，我们需要对于要编程的机器有一定程度的理解。

与C或C++不同，汇编语言在不同计算机间是不可移植的，一个为Intel 80486写的汇编语言程序不会运行于Motorola 68000上。只要初始源代码对于68000指令集进行了重新编译，那么一个为运行于Intel 80486上而写C程序就可能会运行在Motorola 68000上，但是像高端字节序和低端字节序这样的体系结构不同可能会引起错误。

171

我们也许值得停下片刻反省一下，作为程序员为什么学习汇编语言是重要的。计算机科学和编程依赖于对处理器及其局限性和效能的实际知识。理解汇编语言就是理解你的代码要运行于其上的计算机。虽然如C++和JAVA这样的高级语言已经很好地抽去了低级细节，但紧记这一点仍是重要的：计算机并非无限强大，其资源并非无限多。

汇编语言与处理器的设计紧密关联，代表了从二进制指令集体系结构到人可读形式的第一级的简化。通常在汇编语言指令和其所产生的二进制或十六进制指令之间有1对1的匹配关系，这与C或C++很不同，一个C语句可能会产生几百行的汇编代码。

对于中断处理程序以及为专用处理器（如数字信号处理器（digital signal processor, DSP））所写的算法，尤其需要高效的代码。很多有经验的程序员提出，任何具有绝对确定性的代码都不能用C来编写，因为你不能事先预测由编译器所产生的代码的执行时间。用汇编语言你就能在单个时钟周期的级别上控制你的程序。此外，C运行环境的启动代码的某些部分必须用汇编语言编写，因为除非C代码在能够开始执行之前就确立了运行环境，用C或C++编写的程序是不能正确运行的。这样，引导代码就趋向于用汇编语言编写。最后，理解汇编语言对于调试实时系统也是极其重要的。如果你曾在如微软的Visual C++这样的环境中编写过程序，并且无意中进入过库函数，那么你就会发觉你已经深陷x86汇编语言中了。

Motorola 68000微处理器体系结构

图7-11是一个68K体系结构的简化示意图。由于Motorola有很多的系列成员，所以这个特定的体系结构也称为CPU16。CPU16是CPU32体系结构的子集，而CPU32就是68020及其后续处理器的指令集体系结构。

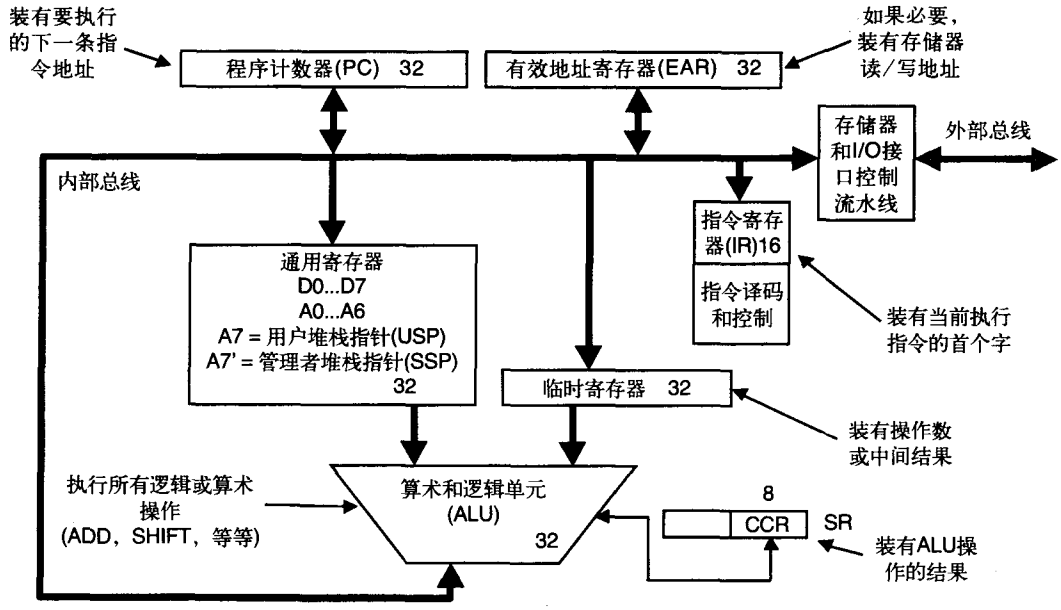


图7-11 Motorola 68K处理器的体系结构

让我们简单地认识一下我们后面要用到的一些重要的功能模块：

- *程序计数器*（program counter）：用于保存要从存储器取出的下一条指令的地址。只要处理器对当前指令进行了译码，程序计数器（PC）就会被更新成存储器中顺序的下一条指令的地址。
- *通用寄存器*（general register）：68K处理器有15个通用寄存器和两个专用寄存器。通用寄存器进一步分为8个数据寄存器D0...D7和7个地址寄存器A0...A6。数据寄存器用于保存和操作数据变量，地址寄存器用于保存和操作存储器的地址。两个专用寄存器A7和A7'用于实现两个独立的堆栈指针。我们将在课本稍后讨论堆栈。
- *状态寄存器*（status register）：状态寄存器是一个16位寄存器。这些位用于描述每次指令执行后计算机的当前状态。条件码寄存器CCR是状态寄存器的一部分，它保存了直接与最近一条指令的执行结果相关的位。
- *算术和逻辑单元*（arithmetic and logic unit, ALU）：ALU是一个功能模块，所有的数学和逻辑数据操作都在其中执行。 172

图7-12就是68K体系结构的*程序员模型*。它只关注体系结构中与编写程序相关的细节，故与图7-11不同。

在本课本中，我们不打算讲解状态寄存器（SR）或者管理堆栈指针（A7'）。我们从现在起就关注D0...D7，A0...A6，CCR以及PC。从这些寄存器中，我们将学到我们需要知道的关于这种计算机体系结构和汇编语言编程的所有知识。 173

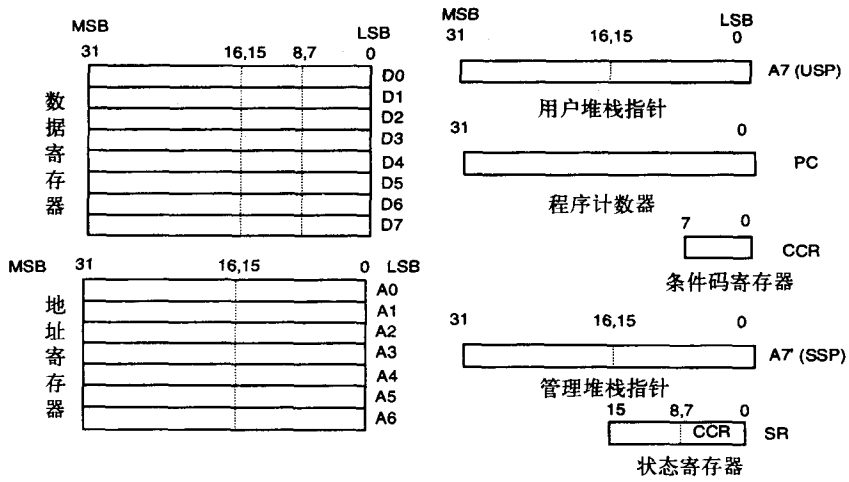


图7-12 68K处理器的程序员模型

条件码寄存器

条件码寄存器（CCR）应给予额外解释。图7-13中比较详细地描述了寄存器。

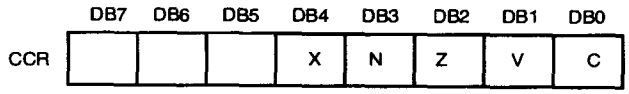


图7-13 条件码寄存器。阴影区的位没有使用

CCR包含5个条件位，其值可随每条指令的执行结果而改变。每个位的精确定义如下：

- X位（扩展位）：用于多精度算术运算
- N位（负数位）：指出结果是一个负数
- Z位（零位）：指出结果等于0
- V位（溢出位）：指出结果可能超出了操作数的范围
- C位（进位位）：指出在一次数学操作中产生了进位

这些位的重要性与BEQ、BNE、BPL、BMI等等这一族测试和转移指令相关联。这些指令测试单独的标志（flag），即CCR位，如果条件为真则转移，如果条件为假则忽略转移而进入下一条指令。例如，BEQ的意思是相等转移（branch equal）。那么，与什么相等呢？BEQ指令实际上是测试零标志位Z的状态。如果Z=1，就意味着寄存器中的结果是零，所以我们执行转移。因此，相等转移就是这样一条指令，如果有一个最近的操作产生了零结果，则它就告诉处理器执行转移。

但是BEQ还有其他的意思。假设我们想知道两个变量是否彼此相等，我们如何测试这个相等性呢？很简单，只要将它们相减。如果我们得到了结果零（Z=1），它们就相等。如果它们彼此不相等，我们就得到非零的结果且（Z=0）。这样，如果Z=1则BEQ为真，如果Z=0则BNE（非零转移）为真。如果N=0则BPL（正数转移）为真，如果（N=1）则BMI（负数转移）为真。

在68K指令集中总共有14个条件转移指令。一些指令只检测单个标志的条件，而其他指令则检测多个标志的逻辑组合。例如，BLT指令（小于转移）定义为：$BLT= N*\overline{V}+\overline{N}*(N\oplus V)$（不是定义为bacon（熏肉），lettuce（莴苣）和tomato（番茄））。

7.3 有效地址

注释：虽然68K处理器只有24条外部地址线，但是在内部它仍然是32位的处理器，因此，我们就能在例子中将地址表示成32位值。

现在，让我们回过头来考虑指令的格式。也许最常用的指令就是移动指令，你会发现你在汇编语言中做的最多的事情就是将数据向四处移动。移动指令需要两个操作数，如下所示：

```
MOVE.W    source(EA),destination(EA)
```

例如：

MOVE.W $4000AA00, $10003000

该指令告诉处理器将存储于位置0x4000AA00的16位值拷贝到位置0x10003000。 174

而且，MOVE助记符有点误导。指令做的是用源操作数覆盖目的操作数的内容，源操作数没有被操作所改变。这样，指令完成后，两个存储器位置包含的都是指令执行前$4000AA00所包含的数据。

在前一个例子中，源操作数和目的操作数是精确规定的存储器地址，是绝对地址(absolute address)。它们是绝对的是因为指令精确地规定了从哪里取来数据，并将数据放到哪里。绝对地址只是68K的可能寻址模式之一，我们称这些寻址模式为有效地址（effective address)。这样，当我们将指令的一般形式写成如下形式时：

MOVE.W source(EA), destination(EA)

我们是在说，数据要移动的源地址和目的地址将由各自的有效寻址模式独立地决定。

例如：

MOVE.W D0, $10003000

将处理器的8个内部数据寄存器之一（在这个例子中是数据寄存器D0）的内容移动到存储器位置$10003000。在这个例子中，用于源有效地址的模式称为数据寄存器直接（data register direct）寻址，而用于目的有效地址的模式是绝对寻址。

我们也可以将移动指令写为：

MOVE.W A0, D5

该指令可将目前存储于地址寄存器A0中的16位字移动到数据寄存器D5。源有效地址是直接地址寄存器（address register direct)，目的有效地址是直接数据寄存器。

假设地址寄存器A0的内容是$4000AA00（我们可用速记符号写成<A0> = $4000AA00)。见下面指令：

MOVE.W D5, (A0)

该指令将数据寄存器D5的内容移动到存储器位置$4000AA00。这是一个地址寄存器间接(address register indirect）寻址的例子。你可能已经熟悉了这种寻址模式，因为这在C++中就是一个指针，地址寄存器的内容就成为存储器操作的地址。我们称之为间接寻址（indirect addressing）模式是因为地址寄存器的内容并不是我们想要的数据，而是我们想要数据的存储器地址。这样，我们不是将数据直接存储到寄存器A0，而是间接地通过将A0的内容作为最终目的的指针，即存储器位置$4000AA00。我们是通过将地址寄存器用括号括起来来指明地址寄存器是作为指向存储器的指针的。假设<A1>=$10003000且<A6>=$4000AA00，看下面的指令：

MOVE.W (A1), (A6)

该指令将位于存储器位置$10003000处的数据拷贝到存储器位置$4000AA00。源有效地址模式和目的有效地址模式都是地址寄存器间接模式。

175 让我们再看一个例子：

MOVE.W #$234A, D2

该指令将十六进制数$234A直接放入到寄存器D2中。这是一个“立即”寻址模式（“immediate” addressing mode）的例子。立即寻址存储是将变量进行初始化的途径。英镑符号（#）告诉汇编器这是一个数，不是一个存储器地址。当然，只有源有效地址才能是立即地址。目的地址不能是数，它必须是一个存储器位置或寄存器。

有效地址（EA）规定了如何访问指令的操作数。不是所有的有效地址都能使用，这取决于被执行的指令。随着课本学习的进行，我们还要更多地讨论这个问题。指令用来访问一个或更多个操作数的有效地址的类型实际上被规定为指令的一部分。回忆一下，一个操作代码字的最小尺寸是16位长。操作代码字提供了处理器需要来执行指令的所有信息，然而，这并不意味着16位操作代码字本身就包含了指令的所有信息。它可能包含了完成整个指令所需的足够信息（如对于NOP），但通常它只包含了足够的信息来知道为完成指令还需要从存储器中取出哪些东西，所以，很多指令比操作代码字要长。

如果我们考虑用基于微代码的状态机来驱动处理器，那么所有这些就开始有意义了。我们需要操作代码字给我们进入微代码的入口，这就是微处理器对指令译码时所要做的。在执行整个指令的过程中，可能需要再出去到存储器去取另外的操作数来完成指令。它知道它不得不做这些另外的存储器取数操作，因为由操作代码字所定义的状态机中路径预先确定了这些操作。

考虑图7-14所示的移动指令的操作代码的形式。

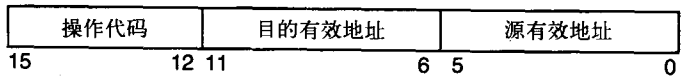


图7-14 移动指令的机器语言指令格式

在操作代码域可包含三种类型的关于移动指令的信息，这些是由DB15−DB12这4个数据位来定义的。这些可能性定义如下：

- 0001=MOVE.B
- 0011=MOVE.W
- 0010=MOVE.L

源有效地址和目的有效地址都是6位的域，每个域又进一步分为两个3位的域。一个3位的域称为寄存器域（register field），该域可取值000到111，对应于数据寄存器D0到D7和地址寄存器A0到A7之一。另一个3位的域就是模式域（mode field），这个域描述了该操作在用哪种
176 有效寻址模式。我们以后还会再回到这个问题。先看一个真实的例子。考虑指令：

MOVE.W #$0A55, D0

这条指令汇编后会是什么样子呢？参考一下汇编语言编程手册。比如程序员参考手册[2]，这样会有助于你更好地理解将发生什么事情。下面就是我们所知道的要发生的事情。这是一条移动指令，传输数据的大小为16位，即一个字。这样，操作代码字的4个最高有效位（D15~D12）就是0011。源有效地址（D5~D0）是立即数，译码为111 100。目地有效地址（D11~D6）是数据寄存器D0，译码为000 000。将这些整理在一起，就得到机器语言操作代码字为：

0011 000 000 111 100，即$303C

从0011 000 000 111 100（二进制）到$303C（十六进制）的转换可能看起来有些费解，因为指令的位域通常不在4位边界处对齐。然而，如果你从右向左进行，按4位一组，你每次都会做对。

这就是指令的所有要素吗？不是，因为此时我们所知道的一切就是源操作数是立即操作

数，我们还不知道这个立即数是什么，所以，操作代码字告诉处理器，它还必须再到存储器取回另一部分的源操作数。由于目地寄存器D0的有效寻址模式是数据寄存器直接寻址，这样就不需要从存储器中取回另外的信息，因为所有用于确定目的地址的信息都已包含在操作码字本身了。这样，在存储器中的完整指令就是$303C 0A55。图7-15图解了这个例子。

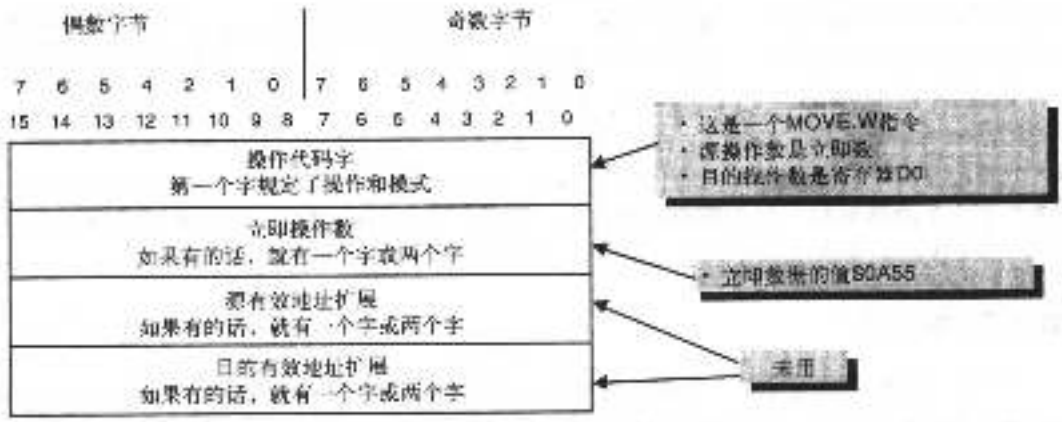


图7-15 指令MOVE.W #$0A55,D0的存储器存储方式

现在，假设指令使用两个绝对地址表示源有效地址和目的有效地址

MOVE.W $0A550000,$1000BB00

这个机器语言操作代码字译码为：0011 001 111 111 001，即$33F9。我们做完了吗？还是没有，这是因为还有两个绝对地址需要从存储器中取出。完整的指令如下： 177

$33F9 0A55 0000 1000 BB00

完整的指令总共占据存储器5个16位字，需要处理器分别做5次存储器读操作才能完整地得到这条指令。这个过程的图解见图7-16。

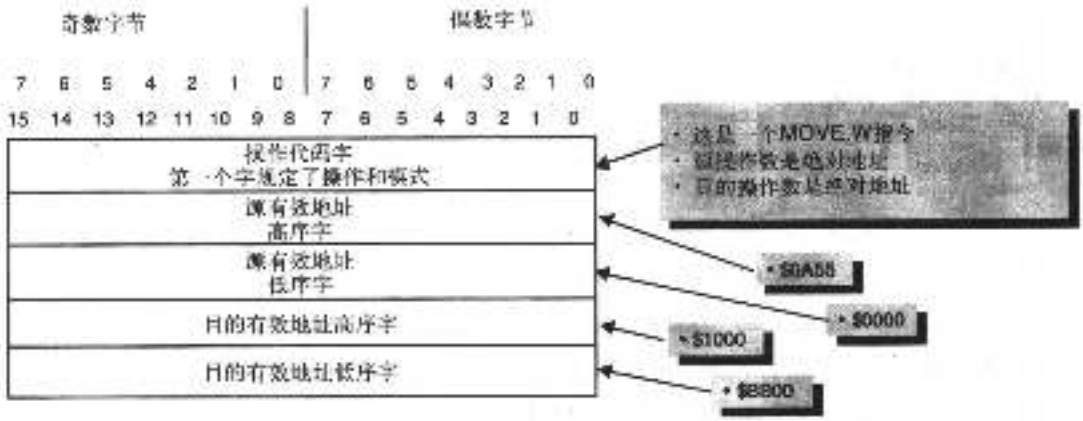


图7-16 指令MOVE.W $0A550000，$1000BB00的存储器存储方式

字对齐

为了准备好进行摆在我们面前的编程任务，我们需要讨论的最后一个主题就是存储器中

的字对齐问题。我们在该课程的早期接触到了这个主题，但回顾一下这个概念是值得的。回忆可知，我们通过使用最低有效位A0来指出我们对哪个字节感兴趣就能对单个字节进行寻址。然而，如果我们试图访问一个在奇数字节边界上的字或长字时，将会发生什么情况呢？图7-17对这个问题做了图解。

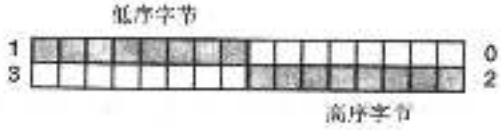


图7-17 非对齐访问的困难

为了取出位于奇数字节边界的字，处理器就必须进行两次16位取数操作，然后丢弃两个字的一部分以得到正确的数据。一些处理器可做此操作。这就是我们先前讨论过的非对齐访问(nonaligned access)模式的例子。一般来说，这种类型的访问在处理器效率方面的代价非常高。68K不能执行这个操作，如果遇到这种情况，它将产生一个内部异常。汇编器检测不到它，这是一个运行时错误。采取的措施就是绝对不要将一个字或长字操作编程到字节（奇数）存储器边界上。

阅读程序员参考手册

当开始写汇编语言程序时，你面临的最令人畏惧的任务就是努力理解生产商的数据手册。Motorola 68K程序员参考手册是用我们称之为“相当简明扼要的风格”写成的。它是为已经知道如何写汇编语言程序的人所提供的参考，而不是为正在努力学习编程方法的学生所提供的学习工具。从现在开始，你将需要程序员参考手册，或者关于68K汇编语言编程的一本好书，
178 比如Clements[3]的教科书。但是，当要写程序并需要程序员参考手册这样的手边参考书的时候，教科书就不是特别有效率。幸运的是，从Freescale或其他公司网址（见本章参考文献中的URL）很容易得到参考手册的免费拷贝。

因此，假设你面前有这本书，让我们上一堂速学课程来理解Motorola技术资料编写人员试图想告诉你的知识。下面的文字类似于一条指令在参考手册页中的布局：

1. 标题：*ADDI* 加立即数 *ADDI*

2. 操作：立即数＋目的→目的

3. 语法：*ADDI* #<data>, <ea>

4. 属性：大小＝（字节，字，长字）

5. 描述：将立即数加到目的操作数，并将结果存到目的位置。操作的大小可规定为字节，字或长字，立即数的大小与操作的大小匹配。

6. 条件码

X	*N*	*Z*	*V*	*C*
•	•	•	•	•

N：若结果为负则置位，否则清零。

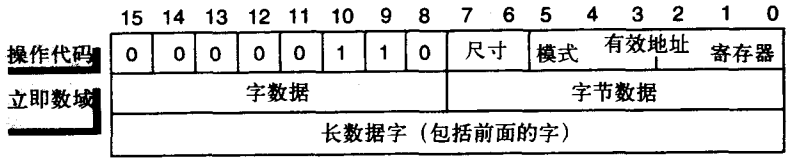
Z：若结果为零则置位，否则清零。

V：若产生溢出则置位，否则清零。

C：若产生进位则置位，复杂清零。

X：与进位的情况相同。

7. 指令格式：



8. 尺寸域：

规定了操作的尺寸

DB7	DB6	操作尺寸
0	0	字节操作
0	1	字操作
1	0	长字操作

179

9. 有效地址域：

规定了目地操作数。只允许数据可改变的寻址模式，如下所示：

寻址模式	模式	寄存器	寻址模式	模式	寄存器
Dn	000	reg.num:Dn	(XXX).W	111	000
An	---	不允许	(XXX).L	111	001
(An)	010	reg.num:An	#<data>	---	不允许
(An)+	011	reg.num:An			
−(An)	100	reg.num:An			
(d_{16}, An)	101	reg.num:An	(d_{16}, PC)	---	不允许
(d_8, An, Xn)	110	reg.num:An	(d_8, PC, Xn)	---	不允许

10. 立即数域：（数据紧接着操作代码字）

若 尺寸＝00，则数据是字立即数的低序字节。

若 尺寸＝01，则数据是整个字立即数。

若 尺寸＝10，则数据是下两个字立即数。

让我们一步一步地完成这个过程：

1.	指令和助记符：ADDI—立即数加
2.	操作：立即数数据＋目的→目的 将立即数加入到目的有效地址的内容中，并将结果放回到目的有效地址
3.	汇编器语法：ADDI #(data), <ea> 这解释了如何用汇编语言写指令。注意，除了ADD，还有一个用于立即数加指令的特殊操作代码，就是ADDI，虽然你仍然必须在源操作数域插入#符号
4.	属性：字节、字或长字（long） 你可以加一个字节、一个字或一个长字。如果你略去.B、.W或.L修饰符，默认的修饰符就是.W
5.	描述：告知指令做什么。 这是该页中唯一真正的英文语法，所以是你试图理解指令一切情况的最好机会
6.	条件码： 列出受指令影响的条件码（标志）
7.	指令格式： 如何生成机器语言指令。注意该指令只需要一个有效地址

（续）

8.	有效地址域： 该指令所允许的有效地址模式。它还给出了有效地址的模式和寄存器值。注意地址寄存器不允许作为目的有效地址，立即数也不行。还要注意，两个涉及到程序计数器PC的寻址模式也不允许用在这条指令中

180

流程图

汇编语言程序因其高度结构化的特性，非常适于用流程图帮助进行程序结构的规划。一般来说，流程图不用于规划如C或C++这样的高级语言的编写。然而，流程图对于汇编语言程序的规划仍十分有用。

在生成流程图时，我们用矩形表示“操作”。操作可以是一条指令、若干条指令或者是一个子程序（函数调用），所以矩形与做某件事相关联。我们用菱形代表一个决策点（程序流控制），用箭头代表程序流。图7-18就是计算机起动一个程序的简单的流程图例子。

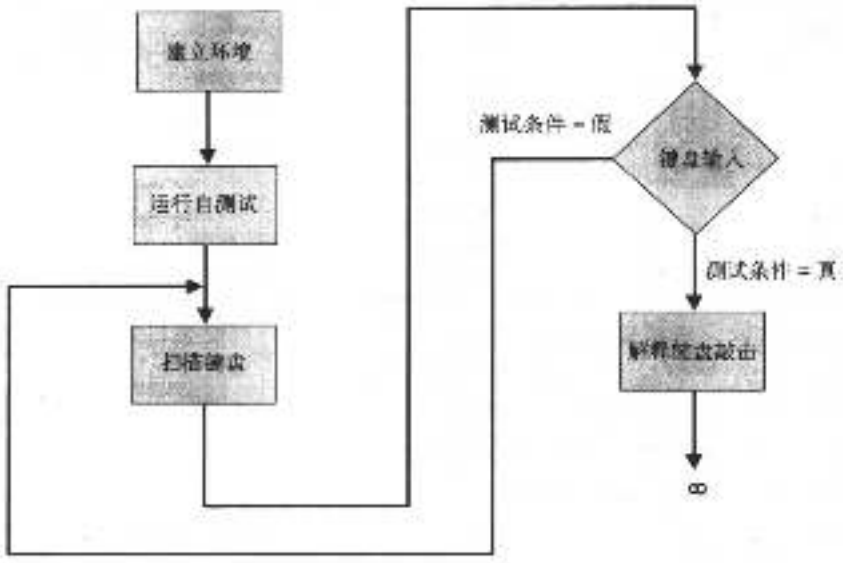


图7-18 一个简单程序的流程图，用于起动系统

流程图起始于“建立环境”这个操作，这可能是几条指令或几十条指令。一旦确立了环境，下一个操作就是“运行自测试”，这可能又需要几条指令或很多条指令，具体多少我们不知道，要依赖于应用。决策点要等待一个键盘输入。程序扫描键盘，然后测试看是否有键盘被敲击。如果没有，则返回（循环）并继续扫描。如果有键盘被敲击，则程序解释该敲击并继续前进。

流程图对于帮助你规划程序而言是一个非常强大的工具。不幸的是，大多数学生（以及很多职业程序员）采取“代码堆砌”的方式，就是立即开始写代码，很像写一本小说。这可能就解释了为什么大多数程序员穿着便鞋而不是篮球运动鞋，鞋带也不系，但那完全是另外一个问题了。

编写一个汇编语言程序

记得莱昂纳多·迪卡普里奥在电影《泰坦尼克号》（Titanic）中有这样一句著名的台词：

181

“我是这个世界的国王！”在汇编语言编程中，你就是这个计算机世界的绝对君主，没有什么

能阻止你犯不可思议的编码错误。没有类型检查或编译警告，只有你和机器面对面。为了用汇编语言编写程序，你必须持续地了解系统的状态。你必须紧记你有多少存储器及其位置在哪里，你的外部设备的位置以及如何访问它们。简而言之，你控制一切并为一切负责。

为了能用68K汇编语言编写和运行程序，我们将不是实际地使用一个68K处理器。要用68K处理器，你就需要基于68K系列处理器的计算机，比如最早的Apple Macintosh®。我们将采取一种不同的方式来解决问题，即用一个称为指令集模拟器（instruction set simulator, ISS）的程序来代替68K处理器。

商业上可得到的指令集模拟器的价钱是10 000美元，但我们的是免费的。我们很幸运，已经有两个很好的68000处理器的模拟器了。第一个模拟器是由Clements[4]和他在英国Teesside大学的同事一起开发的，可从他的网站下载。

Kelley[5]开发了第二个模拟器，称为Easy68K。这是一个更新的模拟器，并有广泛的调试支持，而这正是Teesside版本所缺乏的。两个模拟器都是运行在Windows操作系统下的，而且，Easy68K是为32位windows所编译的，因此有更好的机会运行于Windows 2000、XP等这些操作系统下，虽然Teesside模拟器看起来在Windows更现代的操作系统版本下运行得也不错。

模拟器与集成设计环境（IDE）更为紧密。IDE在一个软件包中包括了文本编辑器、汇编器、模拟器以及调试器。模拟器就像一个结合在一起的计算机和调试器。你可以做很多你习惯的调试操作，比如：

- 对存储器进行存取
- 检查和修改寄存器
- 设置断点
- 从断点开始运行或者运行到断点
- 单步运行和观察寄存器

一般来说，编制和运行汇编语言程序是简单和直截了当的：

1. 使用你最喜爱的ASCII文本编辑器或者指令集模拟器包中包含的编辑器，编制你的源程序并以老版本的DOS 8.3文件格式存储。一定要用.x68的扩展名存储，例如，将你的程序存储为myprog.x68。编辑器将自动地在你的文件上附加x68后缀。

2. 用包含的汇编器对程序进行汇编。汇编器将产生完全的机器语言文件，所以你能在模拟器上运行该文件。如果你的程序汇编没有错误，你就能接着在模拟器中运行它。汇编器实际上产生了两个文件：一个完全的二进制文件和一个列表文件（listfile）。列表文件为你显示原始源文件和其产生的十六进制机器语言文件。如果你的源文件存在任何错误，则错误将显示在列表文件输出上。

在Teesside汇编器中，汇编语言程序运行于一个模拟的计算机上，该计算机有1MB的存储器，占据的虚拟地址范围是$00000...$FFFFF。Easy86K模拟器运行于全部的16M地址空间上。Teesside模拟器包也有一些小毛病，但总体来看，在Windows下运行良好。

为了用汇编语言编程，你应该已经成为一个胜任的程序员。你应该已经修过几门编程方面的课程并理解编程结构和数据结构。汇编语言编程可能开始时看起来很奇怪，但它也仍是编程。虽然C和C++是自由形式的语言，但汇编语言是非常结构化的。而且，要由你来掌握资源的使用情况，没有编译器可用来为你做资源分配。 182

不要为68K或任何处理器的指令集体系结构的操作代码和操作数的数量所吓倒，真正的要点是你只采用这些可能指令和寻址模式的子集就能写出相当合理的程序。不要为你可能会使用的指令数量所淹没，使用一些你能理解的指令和寻址模式轻松自在地写程序，然后当你需

要更高效或仅仅是想扩展你的技能时再开始结合进其他的指令和寻址模式。

随着更深入地了解面临的编程问题，你自然地就会寻求更高效的指令和有效寻址模式，使你的工作更轻松并编写出更好的程序。你将很快发现，当我们考察不同的计算机体系结构时，很少的程序员或编译器利用了指令集中的所有指令。在大多数时间，指令的一个小的子集就能使工作完成得相当好。然而，也不时会出现问题，这就需要某些晦涩的指令来解决。祝编程快乐！

7.4 伪操作代码

实际上，你在程序中可以使用两种类型的操作代码。第一种就是68000处理器使用的实际指令的操作代码的集合，它们构成了68K的指令集体系结构。第二种就是称为伪操作代码（pseudo-op）的集合。这些代码就像真正的操作代码一样置于你的程序中，但它们实际上是针对汇编器的指令，告诉汇编器如何对程序进行汇编。只要将伪操作代码看作编译器指示命令和定义语句就行了。

伪操作代码还称为汇编命令（assembler directive），它们被用于使程序更易读，或者用于为汇编器提供关于对程序如何进行处理的附加信息。重要的是要意识到，商业上可得到的、具有工业应用能力的汇编器在任何一点上都和任何现代编译器同样复杂。让我们考察一些这样的伪操作代码。

- **ORG**（置起始值）：ORG伪操作代码告诉汇编器在存储器的哪个地方开始汇编程序。知道程序要放在存储器的哪个地方不是必要的，这只是告诉汇编器你打算要它在哪里运行程序。如果你省略了**ORG**语句，程序将被汇编成在存储器位置$00000处开始运行。由于从$00000到$003FF的范围是为系统向量保留的，所以我们通常将程序“**ORG**”到从存储器地址$00400处开始。因此，程序的第一行应该是：

```
<标号>    ORG    $400    <*注释>
```

下一个伪操作指令是置于你源程序末尾的。它有两个功能：首先，它告诉汇编器在这一
183 点停止汇编；其次，它告诉模拟器从哪里将程序装入到存储器。

- **END**（源程序结束）：END后的任何东西都被忽略。格式：

```
<无标号>    END    <地址>
```

注意**ORG**伪指令和**END**伪指令是互补的。**ORG**伪指令告诉汇编器如何解决地址引用，实际上就是你打算在哪里运行程序。**END**伪指令指示装载程序（模拟器的一部分）将程序代码放在存储器中的哪个地方。在大多数时间，**ORG**和**END**的地址是相同的，但它们也不必这样。很有可能一个程序装入到一个地方，然后在运行时又重定位到另一个地方。我们的意思是，你通常应该用下面的伪指令作为程序的开始：

```
ORG    $400
```

并用下面的伪指令作为程序的结束：

```
END    $400
```

- **EQU**（等于伪指令）：等于伪操作与C中的`#define`是一样的，它允许你为常数值提供一个符号名字。格式为：

```
<标号>    EQU    <表达式>    <*注释>
```

表达式可以是数学表达式，但最有可能只是一个数字。你也可使用等于伪指令来从其他符号值生成一个新的符号值。然而，在新符号被求值时，这些其他符号的值必须已知，这意

味着你不能有向前引用。

等于伪指令像C和C++中的“#define”命令，是针对C编译器的指令，用于将你源文件中的符号名替换为数字值。例如：

```
Bit0_test    EQU    $01    * 隔离出数据位0
ANDI.B    #Bit0_test,D0    * D0的第0位等于1吗?
```

以上指令对你来说要比下面指令更富有涵义：

```
ANDI.B    #$01,D0    * 不可理解的数字
```

尤其在你几天没有看到代码之后更是如此。

- **SET**（置符号）：SET与EQU一样，除了它在随后可用于将符号重新定义成另一个值。格式为：

```
<标号>    SET    <表达式>
```

7.5 数据存储伪指令

下一组伪操作代码称为*数据存储伪指令*（data storage directive）。这些指令的用途是指示汇编器分配存储块，也许还要用值来初始化存储器。

- **DC**（定义常量）：生成一个存储块，包含列于源文件中的数据值。格式为：

```
<标号>    DC.<尺寸>    <项>, <项>, ...
```

例子

```
error_msg    DC.B    'Error 99', $0D, $0A, $00    * 错误消息
```

一个置于单引号内部的文本串被汇编器解释为一个ASCII字符串。这样，这条伪指令就等 184
价于如下写法：

```
error_msg    DC.B    $45, $72, $72, $6F, $72, $20, $39, $39, $0D, $0A, $00
```

如你所见，采用单引号就使你的意图容易理解得多。在这个例子中，与标号error_msg相关联的存储器位置将包含字符串的第1个字节，即$45。

- **DCB**（定义常量块）：将一个存储器块初始化到相同的值。长度是字节、字或长字的数量。格式是：

```
<标号>    DCB.<尺寸>    <长度>, <值>
```

- **DS**（定义存储）：生成一个未初始化的存储块。如果你需要定义一个以后要用来存储数据的存储区域，就使用这个伪指令。格式：

```
<标号>    DS.<尺寸>    <长度>
```

- **OPT**（置可选项）：告诉汇编器，你想让汇编器在你还没有明确指出的地方生成程序代码，想让列表文件如何被格式化。

在这里值得注意的唯一可选项是**CRE**选项。这个可选项告诉汇编器在列表文件上生成一个交叉引用的列表。当到了调试程序的时候，这个选项具有无法估量的价值。例如，考察情况1的代码块。

情况1：没有CRE可选项的列表文件

```
源文件：EXAMPLE.X68
默认项：ORG $0/FORMAT/OPT A,BRL,CEX,CL,FRL,MC,MD,NOMEX,NOPCO
1
2 ************************
3 *
4         * 这是一个不用交叉引用的例子
5         * 没有使用交叉引用
6 *
7 ************************
8 00000400 ORG $400
9
10 00000400 103C0000      START:    MOVE.B #00,D0
11 00000404 66FA          TEST:     BNE    START
12 00000406 B640          COMPARE:  CMP.W  D0,D3
13 00000408 6BFC          WAIT:     BMI    COMPARE
14 00000400                         END    $400
Lines: 14, Errors: 0, Warnings: 0.
```

情况2：有CRE可选项的列表文件

```
源文件：EXAMPLE.X68
默认项：ORG $0/FORMAT/OPT A,BRL,CEX,CL,FRL,MC,MD,NOMEX,NOPCO
1
2 ************************
3 *
4 * 这是一个使用交叉引用的例子
5 * 引用
6 ************************
7
8                           OPT CRE
9 00000400                  ORG $400
11 00000400 103C0000 START:  MOVE.B  #00,D0
12 00000404 66FA     TEST:   BNE     START
13 00000406 B640     COMPARE: CMP.W  D0,D3
14 00000408 6BFC     WAIT:   BMI     COMPARE
15 00000400                  END     $400
Lines: 15, Errors: 0, Warnings: 0.

SYMBOL TABLE INFORMATION
COMPARE (ABSL 00000406 13 14.
START (ABSL 00000400 11 12.
TEST (ABSL 00000404 12 * * NOT USED * *
WAIT (ABSL 00000408 14 * * NOT USED * *
```

185

注意**OPT CRE**伪指令是如何生成一个符号表的。这个表是根据你在程序中定义的标号、它们的值、它们首次被定义的行号，以及所有对它们进行引用的行号生成的。你很快就会看到，这对你的调试过程有非常宝贵的帮助。

7.6 汇编语言程序的分析

假设我们想将简单的C赋值语句Z=Y+24实现成为汇编语言程序，该程序会是什么样的呢？让我们考察下面的简单程序。假设Y=27，程序如下：

```
            ORG         $400    * 代码开始处
            MOVE.B      Y,D0    * 取第一个操作数
            ADDI.B      #24,D0  * 做加法
            MOVE.B      D0,Z    * 做赋值
            STOP        #$2700  * 告诉模拟器停止
            ORG         $600    * 告诉模拟器停止
Y           DC.B        27      * 数据区开始处
Z           DS.B        1       * 在存储器中存储常量27
            END         $400    * 为Z保留一个字节
```

注意，当我们使用没有“$”打头的数字27时，汇编器会将其解释为一个十进制数。还要注意那几个**ORG**伪指令，它在地址$400处定义了一个代码空间，在$600处定义了一个数据空间。下面是我们已经生成的汇编器列表文件，我们略去了注释。

```
1 00000400                      ORG         $400
2 00000400      103900000600    MOVE.B      Y,D0
3 00000406      06000018        ADDI.B      #24,D0
4 0000040A      13C000000601    MOVE.B      D0,Z
5 00000410      4E722700        STOP        #$2700
6 * Comment line
7 00000600                      ORG         $600
8 00000600      1B          Y:  DC.B        27
9 00000601      00000001    Z:  DS.B        1
10 00000400                     END         $400
```

186

让我们逐行分析该程序：

行1：**ORG**语句定义了程序的起始点。

行2：将位于地址$600(Y)的字节数据移动到数据寄存器，这就将值27($1B)移动到了寄存器D0中。

行3：将字节数24（$18）加到D0的内容中，并将结果存回到D0。

行4：将字节数据从D0移动到存储器位置$601。(与行2保持一致)

行5：这条指令由模拟器用来停止程序执行，它是指令集模拟器的产物，在大多数真实程序中没有。

行6：注释。

行7：**ORG**指令复位汇编器指令计数器，以开始在地址$600处再次计数，这就有效地为程序定义了数据空间。如果没有用**ORG**指令，数据区就紧接着**STOP**指令之后开始。

行8：用标号“Y”定义存储器位置$600，并将其初始化为$1B。注意，我们虽然使用了伪指令**DC**，但没有什么能阻止我们向这个存储器位置写数据来改变其值。

行9：用标号“Z”定义存储器位置$601，并保留1个字节的存储。没有必要对其初始化，因为它将呈现加法的结果。

行10：**END**伪指令。程序将装入地址$400开始处。

187

注意我们的实际程序只有3条指令长，但其中两条指令是移动指令，这是相当典型的。你将会发现你的程序代码的大部分被用来将变量到处移动，而不是对其进行任何操作。

总结

- 一个典型的计算机系统的存储器是如何组织和寻址的。

- 字节寻址是如何实现的，以及高端字节序和低端字节序两种字节寻址方法之间的含糊意义。
- 汇编语言指令的基本组织，以及操作代码字如何被解释成为机器语言指令的一部分。
- 产生汇编语言程序的基本原理。
- 采用伪操作代码来控制汇编程序的操作。
- 分析一个简单的汇编语言程序。

参考文献

[1] Steven Levy, *Hackers: Heroes of the Computer Revolution,* ISBN 0-385-19195—2, Anchor Press/Doubleday, Garden City, 1984, p. 30.

[2] Motorola Corporation, *Programmer's Reference Manual,* M68000PM/AD REV 1. This is also available on-line at: http://e-www.motorola.com/files/archives/doc/ref_manual/M68000PRM.pdf.

[3] Alan Clements, *68000 Family Assembly Language,* ISBN 0-534-93275-4, PWS Publishing Company, Boston, 1994.

188

[4] Alan Clements, http://www-scm.tees.ac.uk/users/a.clements

[5] Charles Kelley, http://www.monroeccc.edu/ckelly/tools68000.htm.

习题

1. 一个处理器的外部数据总线宽度和内部数据总线宽度必须是相同的吗？论述一下它们为什么可以不同。
2. 解释一下为什么一个微处理器的地址总线和数据总线被称为是同质的，而状态总线被认为是异质的总线。
3. 下面的所有指令或者是非法的，或者将引起程序崩溃。对于每个指令，简要叙述为什么该指令是错误的。
   a. MOVE.W $1000, A3
   b. ADD.B D0, #$A369
   c. ORI.W #$55AA007C, D4
   d. MOVEA.L D6, A8
   e. MOVE.L $1200F7, D3
4. 用几句话简要解释下面的每一个术语或概念。
   a. 高端字节序/低端字节序
   b. 非对齐访问
   c. 地址总线、数据总线、状态总线
5. 考虑下面的68000汇编语言程序。程序结束后，存储于位置$0000A002的字节值是什么？

```
* 系统的等于伪指令定义
foo     EQU     $AAAA
bar     EQU     $5555
mask    EQU     $FFFF
start   EQU     $400
memory  EQU     $0000A000
plus    EQU     $00000001
magic   EQU     $2700
```

```
* 程序开始于此
ORG     start
        MOVE.W      #foo,D0
        MOVE.W      #bar,D7
        MOVEA.L     #memory,A0
        MOVEA.L     A0,A1
        MOVE.B      D0,(A0)
        ADDA.L      #plus,A0
        MOVE.B      D7,(A0)
        ADDA.L      #plus,A0
        MOVE.W      #mask,D3
        MOVE.W      D3,(A0)
        MOVE.L      (A1),D4
        SWAP        D4
        MOVE.L      D4,D6
        MOVE.L      D6,(A1)
        STOP        #magic
        END         start
```

189

6. 考察下列的代码片段。在存储器位置$4000的长字值是什么？

```
Start   LEA         $4000,A0            * 初始化A0
        MOVE.L      #$AAAAFFFF,D7       * 初始化D7和D6
        MOVE.L      #$55550000,D6
        MOVE.W      D7,D0               * 装入D0
        SWAP        D6                  * 交换
        SWAP        D0
        MOVE.W      D6,D0               * 装入D0
        MOVE.L      D0,(A0)             * 存储它
```

7. 考察下列每个Motorola 68 000汇编语言指令，指出哪些指令是正确的，哪些指令是错误的。对于错误的指令，简要解释为什么是错误的。

```
a.  MOVE.L      D0,D7
b.  MOVE.B      D2,#$4A
c.  MOVEA.B     D3,A4
d.  MOVE.W      A6,D8
e.  AND.L       $4000,$55AA
```

8. 4个字节的数据位于从$4000开始的连续存储位置。在不使用任何附加存储位置作为临时存储的情况下，颠倒字节在存储器中的次序。

9. 考察下面的68000汇编语言代码片段。在ADDQ.B指令执行后，D0中字的内容是什么？提示：所有逻辑操作都是按位依次进行的。

190

```
MOVE.W    #$FFFF,D1
MOVE.W    #$AAAA,D0
EOR.W     D1,D0
ADDQ.B    #01,D0              * D0中字的内容是什么?
```

10. 考虑下面的汇编代码片段。第4条指令执行后，寄存器D1中长字值是什么？

```
START   MOVE.L      #$FA865580,D0
        MOVE.L      D0,$4000
        LSL.W       $4002
        MOVE.L      $4000,D1            * <D1> = ?
```

11. 这个问题是让你感受汇编器和模拟器的。在这个介绍的末尾是一个样本程序，你应该生成一个汇编源文件，然后无错误地对其进行编译。一旦编译正确，你接着就应该在你选择的模拟器上运行它，最好从程序开始处使模拟器单步运行。该习题的最终目标是回答问题："当程

序正好要循环回到起始处并重新开始运行时，位于存储器位置$4000的数据字的值是什么？”

```
*************************************************************************
*
* 我的第一个68000汇编语言程序。
*
*************************************************************************
* 注释行开始于一个星号。
* 如果有标号，如“addr1”和“start”，则标号必须起始于一行的第一列。
* 操作代码（如MOVE.W）或伪操作代码（如EQU）必须开始于第二列或以后的列。
* 当心注释的使用，如果注释文字超出一行而到了下一行，而你又忘了使用星号，则你将会得到一
* 个汇编错误。
*************************************************************************
*
* EQUates部分的开始，就像C中的#define一样。
*
*************************************************************************
addr1     EQU     $4000
addr2     EQU     $4001
data2     EQU     $A7FF
data3     EQU     $5555
data4     EQU     $0000
data5     EQU     4678
data6     EQU     %01001111
data7     EQU     %00010111

*************************************************************************
*
191  * 代码段的开始。这些是实际的汇编语言指令。
*
*************************************************************************

          ORG       $400          * 程序开始于$400
start     MOVE.W    #data2,D0     * 装入D0
          MOVE.B    #data6,D1     * 装入D1
          MOVE.B    #data7,D2     * 装入D2
          MOVE.W    #data3,D3     * 装入D3
          MOVEA.W   #addr1,A0     * 装入地址寄存器
          MOVE.B    D1,(A0)+      * 将字节传送到存储器
          MOVE.B    D2,(A0)+      * 传送第2个字节
          MOVEA.W   #addr1,A1     * 装入地址
          AND.W     D3,(A1)       * 逻辑与

*在这里结束。下个指令显示如何使用标号。

          JMP     start           * 程序永远循环
          END     $400            * 在这里结束汇编
```

程序的注释：

1. “EQU”、“END $400”以及“ORG $400”指令是伪操作指令。它们是为汇编器设置的指令，不是68000机器指令，不产生任何68000代码。
2. 还要注意默认的数制是十进制。要写一个二进制数，如00110101，你就应该采用一个百分符号写成%00110101。十六进制数，如72CF，则采用美元符号写成$72CF。
3. 注意每条指令都被注释了。
4. 观察代码。注意多数指令只是到处移动数据，这在汇编语言编程中很平常。
5. 指令**AND.W D3, (A1)**是另一种寻址模式的一个例子，这种模式称为间接寻址（indirect addressing）。在C++中，我们称此为指针。包围A1的圆括号的意思是内部寄存器A1包含的

内容应该是一个存储器地址而不是一个数据值。这条指令告诉计算机取出存于寄存器D3中的16位量，并与存于A1寄存器所指向地址中的16位值进行与操作，然后将结果回送到A1所指向的存储器位置。

6. 指令**MOVE.B D1, (A0)+**是带后递增的间接寻址（indirect addressing with post increment）的一个例子。指令将一个字节的数据（8位）从内部数据寄存器D1移动到地址寄存器A0的内容所指向的存储器位置。这种方式类似于前面的指令。然而，一旦指令执行， 192
A0的内容就会递增一个字节，所以A0将指向存储器的下一个字节。

第8章 汇编语言程序设计

学习目标

- 像计算机中表示的那样处理负数和实数；
- 编写带有循环结构的程序；
- 使用68000 ISA的最常用寻址模式编写汇编语言程序；
- 将C++语言的标准程序控制结构表示为汇编语言中等价的结构；
- 使用程序栈指针寄存器编写包含子程序调用的程序。

8.1 引言

在第7章中，我们介绍了汇编语言程序的基本结构。这一章我们将继续学习68K的寻址模式，然后学习一些高级的汇编语言程序结构，例如循环和子程序，目的是要使用尽量多的新方法以提高我们用汇编语言设计程序的能力。

由于已经懂得怎样使用诸如C、C++和Java等高级语言编写程序，我们将根据这些已经熟悉的程序结构（例如DO、WHILE和FOR循环）来学习与它们类似的汇编语言结构。记住，你用C或C++编写的任何代码最终都会编译成机器语言，所以每一个高级语言结构必然有一个汇编语言类似结构。

学习汇编语言程序设计的最好途径之一是仔细分析一个汇编语言程序。学习大量的指令本身没有太大的用处，因为一旦你熟悉了68K的程序员模型以及寻址模式，你就可以在参考手册中找到你需要的指令并从那里开始编写程序。因此，我们将精力集中学习程序是怎样设计和编写的，然后学习几个示例程序来理解它们的工作原理。

数值表示

在我们深入探究汇编语言的奥秘之前我们需要先学习一些零碎的知识，特别是，我们需
193 要学习各种不同类型的数字是怎样存储在一台计算机中的。到目前为止，我们只是在给定数字的二进制位数的情况下，从数字范围的角度考察了正数。

可以表示的最大正数为：

$$2^N-1$$

其中N是二进制位数。

因此，我们可以表示的正整数范围如下：

二进制位数	最 大 值	名 字	C语言中等价名字
4	15	半字节（nibble）	----
8	255	字节(byte)	char
16	65 535	字(word)	short
32	4 294 967 295	长字(long)	int
64	18 446 744 073 709 551 616	长长字(long long)	---

很明显，这里遗漏了两类数：负数和实数。我们先看负数然后再看实数。

对你来说可能很奇怪，68K内没有用于从一个数减去另一个数的电路。然而，减法是非常重要的，不仅因为它是4种常用算术运算（加、减、乘、除）之一，而且因为它是用来判断一个值等于、大于或小于另一个值的基本方法。这里包含所有的比较操作，诸如CMP或CMPA之类的比较指令的目的是设置条件控制寄存器（CCR）中的相应标志，但并不改变比较中用到的任何数。因此，若我们执行指令：

```
test    CMP.W  D0,D1          * <D0> = <D1> ?
```

那么如果<D0> = <D1>，则零标志位将被设置为1，因为这相当于将这两个值相减，如果减法的结果等于0，那么这两个数必然相等。即使你现在还不清楚，也没有关系，我们将在这一章的后面更详细地讨论CCR。

为了在68K体系结构中将两个数相减，有必要将减数转换为负数然后再将两个数字加在一起。这样：

$$A-B = A + (-B)$$

68K处理器中加在一起的数字既可以为正数又可以为负数。你编写的算法必须利用进位位C，负数位N，溢出位V和扩展位C来决定操作的上下文情况。处理器也必须使用负数来向后跳转，它是通过将一个负数加到程序计数器PC的内容来实现的，这有效地在程序流中产生了向后跳转。为什么是这样的呢？这值得稍微说明一下。回忆可知，一旦一个操作代码字被解码，计算机就知道当前正在执行的指令中包含有多少个字。它首先做的一件事就是将PC增加 194
一定的值以使PC中包含的值为下一条即将要执行的指令地址。

转移指令通过简单地将其操作数加到PC中来促使处理器执行一条非顺序指令，该操作数称为*位移*（displacement）。若位移为正数，那么该转移就向前进行；如果位移是一个负数，那么转移就向后进行。

为了将一个数从正数转换为负数，我们将它转换为它的*补码*（two's complement）表示，这是一个两步的过程。

*步骤*1：字节、字或长字中的各位取反码。求各位的反码就是每个1变为0，每个0变为1。这样，取反（01010101）= 10101010，故$55的反码就是$AA。00的反码是$FF，如此等。

*步骤*2：将补码加1。因此−$55 = $AB。

补码是称为*基补码*（radix complement）的减法中的一种。当你使用2、8、10、16或任意其他的基数系统时这种减法仍然可以正确地工作。这种方法的优点在于只保留一种算术运算（加法），并通过将减数转换为负数来处理减法。

对于以特定基数（radix）表示的数字N，若包含d位数字，则其基补码定义如下：

非零值N的基补码 = r^d-N，0的基补码为0。换句话说，−0 = 0。

让我们看看对于十进制数字4 934，怎样求其基补码。因为$r^d-N = 10^4-4\ 934 = 5\ 066$，所以5 066等于−4 934。我们从8 013减去4 934看看这是否真的有效。首先，我们将按照传统的形式来完成减法操作：

$$8\ 013-4\ 934 = 3\ 079$$

现在，我们将8 013和4 934的基补码相加：

$$8\ 013 + 5\ 066 = 13\ 079 \Rightarrow 1\ 3079$$

很明显，4位最低有效位是一样的，但是在最高有效位留下一个1。这是减法的基补码

方法的传统用法，我们将弃用这种方法。我们之所以留下这个数字就是要看看结果是否大大增加。

如果x = a−b，那么x = a−(r^d−b) = r^d + (a−b)。因此，留给我们的数值结果分为两部分：基数的数字幂和减法的实际结果。通过丢弃基的数字幂，我们就能得到想要的结果。因此，补码运算只是一种基补码运算，用于基数为2的数字系统。

为了将一个负数转换回它的正数形式，可以用完全相同的方法实现补码转换。数字的最高有效位通常被称为符号位（sign bit），因为负数的最高有效位为1，但这并不是一个严格的符号位，因为+5的反并不是−5。

无论如何，这是一个次要的问题，因为如果操作结果将字节、字或长字的最高有效位设
195 为1，那么标志N将总被置位。请看下面的一段代码：

例子

```
        ORG         $400
start   MOVE.B      #00,D0
        MOVE.B      #$FF,D0
        MOVE.W      #$00FF,D0
        MOVE.W      #$FFFF,D0
        MOVE.L      #$0000FFFF,D0
        MOVE.L      #$FFFFFFFF,D0
        END         $400
```

在模拟器中跟踪这段代码时，模拟器显示每次当前执行操作结果的最高位为1时，位N就被置位。因此，对于字节操作，DB7 = 1；对于字操作，DB15 = 1；对于长字操作，DB31 = 1。

下面是程序的模拟器跟踪结果。位N和寄存器D0的内容用高亮灰度显示。还要注意模拟器既可以表示正数也可以表示负数，因此，$FF在指令2中显示为−1，但在寄存器中却显示为$FF。

```
PC=000400 SR=2000 SS=00A00000 US=00000000 X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00A00000 Z=0   程序开始
D0=00000000 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->MOVE.B #0,D0

PC=000404 SR=2004 SS=00A00000 US=00000000 X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00A00000 Z=1   指令MOVE.B #0, D0
D0=00000000 D1=00000000 D2=00000000 D3=00000000 V=0   执行完后
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->MOVE.B #-1,D0

PC=000408 SR=2008 SS=00A00000 US=00000000 X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=1
A4=00000000 A5=00000000 A6=00000000 A7=00A00000 Z=0   指令MOVE.B #-1 D0
D0=000000FF D1=00000000 D2=00000000 D3=00000000 V=0   执行完后
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->MOVE.W #255,D0
PC=00040C SR=2000 SS=00A00000 US=00000000 X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
196 A4=00000000 A5=00000000 A6=00000000 A7=00A00000 Z=0   指令MOVE.W #255,D0
```

```
D0=000000FF D1=00000000 D2=00000000 D3=00000000 V=0   执行完后
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->MOVE.W #-1,D0

PC=000410 SR=2008 SS=00A00000 US=00000000 X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=1
A4=00000000 A5=00000000 A6=00000000 A7=00A00000 Z=0   指令MOVE.W #-1,D0
D0=0000FFFF D1=00000000 D2=00000000 D3=00000000 V=0   执行完后
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->MOVE.L #65535,D0

PC=000416 SR=2000 SS=00A00000 US=00000000 X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00A00000 Z=0   指令MOVE.L #65535,D0
D0=0000FFFF D1=00000000 D2=00000000 D3=00000000 V=0   执行完后
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->MOVE.L #-1,D0

PC=00041C SR=2008 SS=00A00000 US=00000000 X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=1
A4=00000000 A5=00000000 A6=00000000 A7=00A00000 Z=0   指令MOVE.L #-1,D0
D0=FFFFFFFF D1=00000000 D2=00000000 D3=00000000 V=0   执行完后
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
```

通过创建补码负数，所有的算术操作都被转换成了加法。但是，当我们使用负数时，数值的范围要减半。因此，

1. 8位数字的范围是−128到+127（0是正数）

2. 16位数字的范围是−32 768到+32 767

3. 32位数字的范围是−2 147 483 648到+2 147 483 647

当算术操作超出了你正在处理数字的范围时会发生什么呢？作为软件开发者我可以肯定你以前碰到过这种错误，这里有一个简单的例子能说明这种错误。在下面的C++程序中bigNumber是32位的整数，初始值是刚好小于整数范围内的最大正数值+2 147 483 647。当循环并增加数字10次后，我们最终将超出数字的最大允许值。我们可以看到输出端的溢出结果。

很不幸，除非我们编写一个错误处理程序来检测这种溢出，否则错误不会被发现。补码运算中也会碰到同样的问题。

补码表示使硬件实现更容易，但是对于粗心的程序员来说却是一个很容易犯错的地方！算术操作的算法必须小心地设置，并且要使用条件标志来保证操作正确。

源代码	程序输出

#include <iostream.h>
int main(void)
{
int bigNumber = 2147483640 ;
for (int i = 1; i <= 10 ; i++)
cout << "The big number equals"
<< bigNumber + i << endl ;
return 0 ;
}
	The big number equals 214748361
	The big number equals 214748362
	The big number equals 214748363
	The big number equals 214748364
	The big number equals 214748365
	The big number equals 214748366
	The big number equals 214748367
	The big number equals -2147483648
	The big number equals -2147483647
	The big number equals -2147483646

例如，当两个负数（字节）相加的和小于−128时会发生什么呢？你需要考虑溢出的状态。在这种情况下，如果补码加法溢出则V = 1，如果没有溢出则V = 0。对于一个n位的带符号数字，V = 1意味着实际结果大于$2^{n-1}-1$或小于-2^{n-1}，如果没有溢出标志，正数结果将被解释为 197

负数，反之亦然。

表达式**N XOR V**总是给出补码结果的正确符号，这可能并不很明显。假设你将字节$6A和$7C相加，这是两个正数，如果我们将这两个数加到一起就会得到$E6。然而，$E6是一个负数，计算机将它解释为−$1A，这是明显错误的。溢出标志必须被设置为1且负数标志也必须设置为1，所以我们至少应当知道结果的符号是正号而且已经发生了溢出。你应当怎样来消除这种情况的发生呢？与C和C++中的一样：使用一种更大的数据类型。现在，如果我们将$006A和$007C相加就会得到$00E6，但是这个数字可以正确地解释为正数。如果你正在将两个正数相加，那么你还需注意进位标志，它可以告诉你加法结果是否溢出了。

下面我们将学习怎样使用补码来表示带符号数和将无符号数的范围映射到带符号数的范围。如果我们将相应的带符号数和无符号数相对应，我们就会明白负数是怎样表示的。

无符号的32位数　　0.....2 147 483 647 | 2 147 483 648.....4 294 967 295

带符号的32位数　　0.....2 147 483 647 | −2 147 483 648.....−1

因此，无符号数可以达到的最大值等于补码中的−1，比带符号数最大值大1的数就等于带符号数范围内的最大的负数（即最小数）。

实数，即包含整数部分和小数部分的数字，在大多数计算中都只能取近似值。你们都熟悉浮点（float）和双浮点（double）数据类型，这些都只能说是一个近似的实数。在计算机中问题就成为："你想以怎样的精度近似一个实数？"我确信你作为一个软件专业人员永远都不会写这种if语句，因为你知道，由于累积舍入误差或转换错误，这种测试可能会被错误地通过或不通过。

```
float a;
float b;
if ( b < a)
    [执行这段代码];
else
    [执行这段代码];
```

让我们来看看在现代计算机中实数是如何表示的。就像我们表示十进制数字中的小数部分那样，从小数点往右依次为10的连续的负数次幂，我们也用相同的方法表示任意基数的小数部分。因此，二进制数10100101($A5)就等价于：

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	•	2^{-1}	2^{-2}	2^{-3}	2^{-4}		
情形1	1	0	1	0	0	1	0	1	•					X	2^0
情形2		1	0	1	0	0	1	0	•	1				X	2^1
情形3			1	0	1	0	0	1	•	0	1			X	2^2
情形4				1	0	1	0	0	•	1	0	1		X	2^3
情形5					1	0	1	0	•	0	1	0	1	X	2^4

198

因此，二进制整数10100101可以表示为实数：

$$1010.0101 \times 2^4$$

虽然这种表示形式初看起来有点古怪，但其实这只是你已经熟悉的十进制数的尾数和指数表示法。从上面的例子中你可以看到小数被限制为4位。如果在不增加其他数字的情况下再进一步，数字就成为：

$$101.0010 \times 2^5$$

这样，当我们转换回整数时，我们的原始值$A5就变成了$A4。如果你是一个会计师这是

可以的，但是对于计算机科学家和工程人员来说却是不行的！

目前，我们将浮点数表示为IEEE-754工业标准形式。1985标准由电子电气工程协会（IEEE）维护和出版，该标准描述了计算机系统中的单精度浮点数和双精度浮点数的表示格式，它几乎被广泛地接受了，其原因是很多高性能微处理器都包含片上浮点单元（FPU）。如果没有关于浮点数表示的标准，那么浮点数的表示方法将留给微处理器公司，这样很多软件库就不再是在平台间可移植的了。

IEEE-754标准分别定义了32位单精度浮点数，64位双精度浮点数和128位四精度浮点数的格式。图8-1示意出对于每种格式的字段划分。

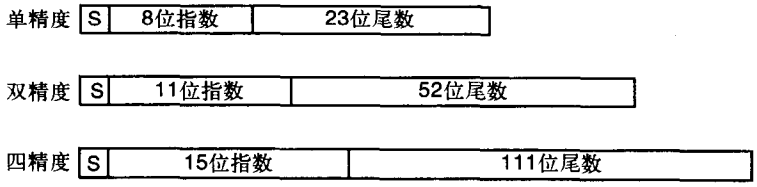


图8-1 IEEE-754-1985浮点数表示

数字的实际表示并不是如此简单明了的，格式中还有很多字段需要作进一步的解释。首先，浮点数表示法是用符号和量级来表示一个数的，而不是补码的整数表示法。符号位S为0表示正数，为1则表示负数。

尾数常常被调整使之成为一个1到2之间的数。这样，尾数的表示形式就与我们在科学计数法中的十进制数表示方式类似。通常，科学计数法中的一个数字都有一个大于1小于10的尾数，因此，小数点左边有一个数字，其余则为小数部分。

在IEEE标准中，一个基为2的数将被调整，使尾数在分数部分的左边总有一个1，但由于1总在那里，所以它会被从数字字段中忽略掉，使得尾数字段多得到一位的精度。这样，单精 199
度数实际上需要33位来表示，但只有32位被存储，因为在小数点的左边总是一个1。

指数也需要探讨一下。指数中没有符号位，所以首先可能想到的是我们会使用补码来表示指数。这是一个很好的猜想，我们使用的方法与补码在原理上非常相似。比补码更优越的地方是位字段的范围通过一个偏置值来弥补。对于8位的指数，偏置值是127（01111111）。因此，如果指数的8位值是10000001，那么指数的值计算如下：

$$129-127=2$$

双精度浮点数的偏置值是1023，四精度浮点数的偏置值则为32 767。于是我们可以将这些部分组合起来从而定义一个浮点数N如下：

$$N=-1^{s}\times 1.F\times 2^{E-B}$$

这里S是符号位，F是尾数的小数部分，E是指数部分而B是偏置值。这样，对于一个单精度数，我们可以表示的指数范围是2^{-127}到2^{128}。

转移和程序控制

大多数程序在进行转移（非顺序取指令）前只执行5到7条指令。大多数转移指令在程序中经常与测试指令（CMP指令）配对使用，这样在条件被测试完后就会根据测试结果马上执行转移。测试指令和转移指令的配对是一种通用规则，因为你不想丢失标志的状态，这些状态由测试指令通过执行另一条指令来设置。但是，尽管测试指令和转移指令的配对是最常用的配对，但也并不总是必要的。请看下面的两段代码：

代码段#1

```
Snippet #1
        MOVE.W   #05,D0
loop    SUBI.W   #01,D0
        CMPI.W   #00,D0
```

代码段#2

```
Snippet #2
        MOVE.W   #05,D0
loop    SUBI.W   #01,D0
        BNE      loop
```

两段代码运行得同样好，但是代码段#2节省了一条指令，因为零标志可以被SUBI.W指令自动置位。使用指令CMPI.W来测试标志的状态是冗余的，因为标志已经被之前的减法指令置位了。你可能认为这有一点走极端，但是在汇编语言中我们要考虑这些事情。使用汇编语言编程的主要原因之一就是能对系统的每个周期进行绝对的控制。节省一条冗余指令可以加速一个实时事务几个微秒。这重要吗？这很难立即回答，但是我可以有一定把握地断言作为一个软件开发者，这在你的将来应该是相当重要的。

条件代码寄存器（CCR）中标志（Z、N、V、X、C）的状态决定了是否会发生一个特别的转移操作。有时，异常处理会导致转移，例如中断、总线错误、陷阱指令或其他错误。陷阱（TRAP）指令是非常重要的，因为它们通常用于操作系统（O/S）和用户应用程序的接口。后面当我们将汇编语言程序与键盘和显示器进行接口以实现用户I/O时，我们就要用到陷阱（TRAP）指令。陷阱（TRAP）指令提供了模拟器中访问操作系统服务的一种很好控制的方法。

每条指令每执行一次，CCR中的状态位（标志）都有可能改变状态。因此，就像前面说过的，通常的做法是在转移指令的紧前面加入转移测试指令，使得任何其他指令都不能修改
200 标志值。下面的表格总结了CCR寄存器标志的意义。

位	定义	意　义
Z	零	如果结果为零则置为1，如果结果非零则置为0
N	负数	等于结果中的最高有效位(MSB)
C	进位	对于一次加法，如果操作数的MSB产生进位则置为1。如果发生借位也要置为1。否则置为0
V	溢出	如果有算术溢出则置位，这意味着结果不能用操作数表示。否则置为0
X	扩展	对数据移动指令透明。当被算术指令影响时，它与进位一样被置位

转移指令测试相应CCR标志的状态，不管是单个标志还是标志的逻辑组合。如果逻辑条件取值为TRUE，就要进行转移。回顾可知，程序计数器（PC）总是指向即将执行的下一条指令的地址。在从存储器中取得下一条指令之前，对PC的内容加上或减去一个*偏移值*（offset value），就能使处理器从一个不同的位置取下一条指令。转移指令的操作数是一个8位或16位的偏移，称为*位移*（displacement）。因此，如果转移测试条件取值为真，则位移被加至PC中的当前值，这就形成了下一条指令的地址。

换句话说：<PC> + 位移 → PC

指令的形式是：**Bcc <位移>**。这里，“cc”是实际转移的条件代码的速记符。你必须用适当的测试条件代替cc。图8-2是所有条件转移指令以及求值方法的总结。

图8-3是包含三个条件转移例子的一段代码。第一个转移测试是以下指令的结果：

```
CMP.L (A0), D0      * 读回
```

这里，寄存器A0是指向存储器中一个位置的指针。那个存储器位置的值与D0的内容作比较，如果它们相等，就转移到标识符**addr_ok**所指向的指令（BEQ）。

第二个转移测试是下面这个比较指令的结果：

```
CMPI.B    #max_cnt, (A3)    * 我们是否已经访问了4个坏地址?
```

Bcc	意　义	逻辑测试
BCC	如果进位清零则转移	C = 0
BCS	如果进位置位则转移	C = 1
BEQ	如果结果等于0则转移	Z = 1
BNE	如果结果不等于0则转移	Z = 0
BGE	如果结果是大于或等于则转移	$N*V+\overline{N}*\overline{V}=1$
BGT	如果结果是大于则转移	$N*V*\overline{Z}+\overline{N}*\overline{V}*\overline{Z}=1$
BHI	如果结果是HI则转移	$\overline{C}*\overline{Z}=1$
BLE	如果结果是小于或等于则转移	$Z+N*\overline{V}+\overline{N}*V=1$
BLS	如果结果等于或或一样则转移	C + Z = 1
BLT	如果结果是小于则转移	$N*V+\overline{N}*V=1$
BMI	如果结果是负数则转移	N = 1
BPL	如果结果是正数则转移	N = 0
BVS	如果结果溢出了溢出则转移	V = 1
BVC	如果结果没有溢出则转移	V = 0

图8-2　条件转移指令及其意义

这里，立即数**max_cnt**与A3所指存储器地址处的内容进行了比较。如果它们相等，就转移到标识符所定义的指令。

第三个转移测试是下面这个比较指令的结果：

```
            CMPA    A0, A1      * 我们是否已经访问了最后的地址?
test_loop   MOVE.L  D0,(A0)     * 将值载入存储器
            CMP.L   (A0),D0     * 读回
            BEQ     addr_ok     * 测试通过，它们是一样的
not_ok      MOVE.W  A0,(A4)     * 保存坏地址位置
            ADDQ.B  #1,(A3)     * 计数器递增
            CMPI.B  #max_cnt,(A3)  * 我们是否已访问了4个坏地址?
            BEQ     done        * 是的，退出测试
addr_ok     ADDQ.L  #inc_addr,A0   * 地址指针递增
            CMPA    A0,A1       * 我们是否已经访问了最后的地址?
            BNE     test_loop   * 没有，继续测试
            BRA     next_test   * 是的，跳到下一个测试
done        STOP    #exit_pgm   * 返回到模拟器
```

如果A3当前所指存储地址处的内容等于数字max_cnt，那么Z标志将被置位且转移到标记为"done"处的指令

如果A0当前所指存储地址处的内容等于寄存器D0的内容，那么Z标志将被置位且转移到标记为"addr_ok"处的指令

图8-3　带有条件和无条件转移的代码片断

[201]

与CMP指令不同，CMPA指令是在我们需要比较一个地址寄存器的值时被使用的。这里，我们比较两个地址寄存器A0和A1的内容，若A1大于等于A0就转移。还要注意一点，我偶而会在尖括号中放入寄存器，例如：<A0> = 0。这是"内容A0 = 0"的缩写形式。你将发现有很多指令只用于地址寄存器或数据寄存器。最后，总是转移指令（branch always，BRA）是用来无条件跳转到另一条指令的（这一章不再说明）。

寻址模式

到目前为止，我们主要是集中精力熟悉了汇编语言程序设计的一些基本原理。为了达到这个效果，我们忽略了对68K指令集体系结构的寻址模式的系统学习。在我们进一步学习体系结构的其他方面（例如堆栈和子程序）之前，我们至少需要简单浏览一下主要的寻址模式。

202 通过一些例子的学习，你可能马上就能熟悉大多数寻址模式。下面所列出的寻址模式是基于操作代码字的有效地址域。这样，在适当的地方就有“模式/寄存器”组合，或者当需要进一步细分模式时的“模式/子类”组合。当我们讨论寻址模式时就会更清楚了。

模式0，数据寄存器直接寻址

源或目的是一个数据寄存器（D0...D7）。

模式1，地址寄存器直接寻址

源或目的是一个地址寄存器（A0...A6）。

寄存器直接寻址（register direct addressing）是最简单的寻址模式。一个操作数的源或目的是一个数据寄存器或地址寄存器，且这个特定源寄存器的内容提供了源操作数。类似地，如果一个寄存器是一个目的操作数，那么它将被装载为指令所指定的值。下面这些例子中的源操作数和目的操作数都使用了寄存器直接寻址模式。

- **MOVE.B** **D0, D3**：将寄存器D0中的源操作数拷贝到寄存器D3
- **SUB.L** **A0, D3**：从寄存器D3中减去寄存器A0中的源操作数
- **CMP.W** **D2, D0**：将寄存器D2中的源操作数与寄存器D0进行比较
- **ADD.W** **D3, D4**：将寄存器D3中的源操作数加至寄存器D4

寄存器是计算机中最宝贵的资源，它们是计算机体系结构中不可缺少的部分且它们在绝大多数汇编语言操作中具有更快的访问速度。大多数算术和逻辑操作必须使用一个数据寄存器或地址寄存器作为计算中的操作数之一。例如，**ADD**指令必须使用一个数据寄存器作为源操作数或目的操作数，另一个操作数则可以是几种有效地址模式之一。

寄存器直接寻址也是非常高效的，因为它使用短指令，它只使用三位来指定8个数据寄存器之一。寄存器直接寻址是快速的，因为外部存储器并不是必须访问的。通常，程序员使用寄存器直接寻址来保存经常访问的变量（也就是中间结果暂存器）。

好的编译器利用寄存器直接寻址来提高性能。实际上，C或C++中的声明使用关键字“register”：

```
register  int foo = 0;
```

该声明告诉编译器，如果可能，你宁愿将“foo”保存为一个可用的寄存器变量而不是一个存储器地址。编译器可能并不能准予你的要求，但是你确实已经提出了这样的请求。

跟踪寄存器及其它们的内容以及最有效地使用它们对汇编语言程序员来说确实相当难。当我们考察RISC处理器的体系结构时，这将变得更加明显。RISC处理器拥有很大的寄存器集（AMD 29000拥有256个寄存器）来给编译器提供很多的快速局部存储。

模式2，地址寄存器间接寻址

203 地址寄存器（A0到A6）包含源操作数或目的操作数的有效存储地址，这在C中是一个指针。我们也可以称这为间接寻址，因为地址寄存器的内容不是我们想要的数据，它是到数据的一个指针，所以它允许我们间接访问存储器。

因此，在间接寻址时，地址寄存器的内容是我们想要访问的数据的地址。

到现在为止，我们总是直接指定源存储地址或目的存储地址，通过使用地址寄存器的内容来指定一个操作数的有效地址，间接寻址就允许我们对地址进行操作。如果你检查一个C编译器产生的汇编语言代码，你将发现指针被直接翻译成了地址寄存器间接寻址模式。一些处理器体系结构（例如Intel 80X86系列中的那些）几乎只使用间接寻址模式。

在地址寄存器间接寻址（address register indirect addressing）中，指令指定68000地址寄存器A0到A6之一来保存我们所需数据的存储地址。编写汇编语言指令时，圆括号表示某某的内容。例如，指令**MOVE.W (A3), D7**告诉处理器将地址寄存器A3所指的存储地址处的内容载入数据寄存器D7。源地址寄存器包含操作数的地址，处理器然后访问地址寄存器所指的操作数，最后，A3所指地址寄存器的内容被拷贝到数据寄存器。我们可以在图8-4中看到这个例子。

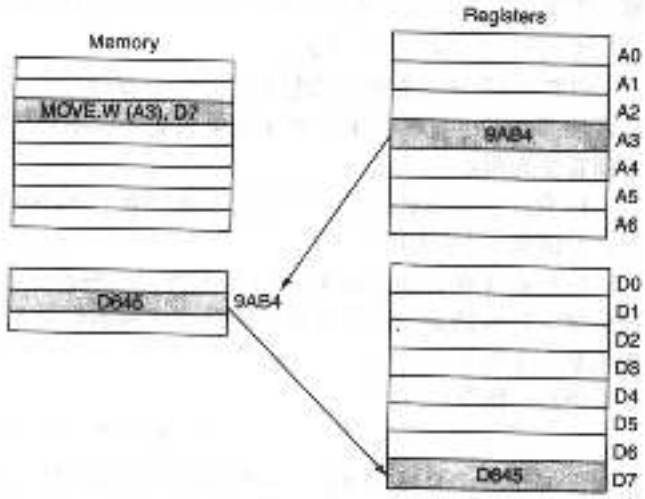


图8-4 地址寄存器间接寻址模式的例子

我们可以用状态机的行为来解释地址寄存器间接寻址。请看下面的两段代码：

代码段 #1

```
MOVE.W    $0600,D0
MOVE.W    D0,$4000
```

代码段 #2

```
MOVEA     #$600,A0
MOVEA     #$4000,A1
MOVE.W    (A0),(A1)
```

在代码段#1中，操作代码字被译码，说明源有效地址是一个绝对字。这意味着处理器必须处理一些事情：

1. 跳出到存储器并读取指令的下一个字。它读入了$0600。
2. 将$0600放置在地址总线上，读取存储地址$0600处的内容到D0中。

在下一条指令中，目的有效地址是一个绝对地址：

1. 跳出到存储器并读取指令的下一个字。它读入了$4000。
2. 将$4000放置在地址总线上，并将存储在D0中的数据写出到地址$4000。

关键在于状态机必须使用有效地址模式以便决定源地址位置或目的地址位置的值，它如何按顺序通过状态机在很大程度上是根据地址模式计算所确定的实际存储地址来决定的。一旦确定好了实际地址值，读写存储器的顺序就总是一样的。 204

在代码段#2中，任务看起来稍微复杂一些，但是它实际上高效得多。两条MOVEA指令用来为地址寄存器A0和A1装载初始值，于是下面的处理就更加高效了：

1. 将地址寄存器A0的内容放置在地址总线上，并将该存储位置的内容读入处理器内的一个暂存寄存器。

2. 将地址寄存器A1的内容放置在地址总线上，并将暂存寄存器的内容写出到那个存储位置。

注意在上面的例子中，一旦将开始地址值装载到了地址寄存器，我们就不再关心这

些值是什么了。我们能完成地址值的各种增减操作，也能正确处理存储器的装载和存储操作。

我希望读者们明白的是地址寄存器间接寻址是一种功能非常强大的寻址模式。实际上，相对于其他模式，你可能更常使用这种模式。它如此重要的原因是因为间接寻址使我们可以在程序执行过程中计算存储地址，而不是程序被汇编过程中将地址固定。这样，如果能够计算一个地址，那么我们就能够很简单地通过表格和结构进行移动。

模式7，子类4：立即寻址

以符号#打头的源值是数据而不是一个存储地址。立即操作数也被称为文字操作数。我们最常使用立即寻址模式来初始化变量或提供程序中的常用数字。当要使用立即寻址时请牢记下面重要的两点：

1. 立即寻址模式只能用于指定源操作数（source operand），因为你不能将一个数字（数据）存储到一个数字中。

2. 你必须在数值前添加一个#号来告诉汇编器这是立即数，否则，它可能被解释为一个存储位置。这可能是一个很难发现的错误，因为你得不到汇编器错误，以指明是一个存储地址而不是一个数字。

例如，指令

```
MOVE.B #4, D0
```

使用了一个文字源操作数和一个寄存器目的操作数。文字源操作数4是指令的一部分，目的寄存器D0使用寄存器直接寻址模式寻址。指令的每个操作数都能使用一种不同的寻址模式。该指令的作用是将文字值4拷贝到数据寄存器D0中。

模式7，子类000：绝对寻址（字）

存储位置被显式地指定为一个16位字。但是，由于68K体系结构的全地址是32位长，所以
205 这种寻址模式使用了16位符号扩展地址（sign extended address）。地址操作数中的最高有效位被用来填充A15到A31的所有高地址位。如果MSB = 0，那么地址就是较低的32K（A0...A14）；如果MSB = 1，那么地址就是较高的32K。

例如，如果16位地址是$B53C，即二进制的1011 0101 0011 1100，那么“符号扩展”的32位地址就是$FFFFB53C。如果16位地址是$7ABC，即二进制的0111 1010 1011 1100，那么符号扩展地址就是$00007ABC。

你可能感到疑惑，“为什么要这么麻烦呢？”首先，绝对短地址（或字）这种地址形式节省了存储空间并且速度更快，因为它从存储器中少取了一个扩展字。其次，大多数ROM代码存于低存储位置而RAM代码存于高存储位置。例如，68K的异常向量表就位于存储器中最开始的256个长字位置。

模式7，子类001：绝对寻址（长字）

存储位置被显式指定为一个32位字。操作数的实际地址被包含在紧跟指令字的两个字中。当从存储器中读出两个地址字后，被操作的数据就从外部的32位存储地址处读取，这里的32位存储地址是通过连接高位字和低位字以形成全存储地址而得到的。

由于68K只有24个外部地址位，这就存在一个叫做混淆（aliasing）的隐含问题。混淆是不留心复制一个地址的一种结果。请记住一个32位全地址可能包含256个24位页，所以最好的办法是尽量保持所有的高地址位（A24到A31）等于0。

在直接寻址或绝对寻址中，指令提供了操作数在存储器中的地址。直接寻址需要两次存储器访问才能取到完整的指令：第一次是访问指令，第二次是访问实际操作数。例如，指令

CLR.B 1234清空（置为0）存储位置1234处的内容。

尽管看起来非常直接，但绝对寻址并不经常使用。除了在最简单的计算机系统中，我们都需要一种灵活性，就是在不考虑代码实际位置的情况下在存储器中任意移动代码。绝对寻址不允许代码重定位，而且它减慢了处理器速度，因为为了确定操作数的存储地址必须进行多次的存储器访问。

模式3，带后递增的地址寄存器间接寻址

地址寄存器（A0…A6）包含源有效地址或目的有效地址。当指令被执行完后，地址寄存器的内容就递增1，这是68K实现退栈（POP）操作的办法。

模式4，带前递减的地址寄存器间接寻址

地址寄存器（A0…A6）包含源有效地址或目的有效地址。在指令被执行之前，地址寄存器的内容就递减1，这是68K实现压栈（PUSH）操作的办法。我们将在后面一部分讨论堆栈的压栈（PUSH）操作和退栈（POP）操作。

模式3和模式4是称为*自动增量*（auto-incrementing）的一种通用寻址方法的例子。如果寻址模式被指定为(A0)+，在被使用后*地址寄存器的内容就递增*。例如：假设地址寄存器A3包含值$9AB4，当被用作一个间接指针时它指向存储地址$00009AB4。假定我们将要执行指令 206

```
ADD.L   (A3)+, D4
```

简单的说，这条指令将把源有效地址的内容加至目的有效地址的内容，并用新值替换目的有效地址中的内容。因为寄存器A3的内容<A3> = $9AB4，所以，存储位置$9AB4处的长字内容被加至寄存器D4的内容中并将结果存回D4。还需要增加地址寄存器A3的内容才能完成指令。应该增加多少呢？如果你回答1，那还是留给你自己吧。因为操作发生在长字量级上，所以应该是<A3>→<A3 + 4>，即$9AB8。因此，递增操作的大小必须与正被操作的操作数大小相匹配。

如果指令**ADD.L (A3)+, D4**后的指令是跳回程序的转移语句，那么结果就是不断将连续存储位置处的内容加至D4。自动递增指令是非常有用的，因为很多存储位置都是有序排列的数据，就如你马上会看到的一样，自动递增指令还被用来实现基于堆栈的寻址操作。

让我们通过一段示例代码来看看带后递增的地址寄存器间接寻址模式的使用。下面的例子使用带后递增的地址寄存器间接寻址模式将存储在连续存储位置的5个数字加在了一起。请注意指令LEA（装载有效地址）的使用，这条指令是专门为将一个地址放入地址寄存器而设计的。程序将存储在存储器中的5个字节的数值相加。

例子

```
        ORG     $400
        MOVE.B  #5,D0       *5个数字相加
        LEA     Table,A0    *A0指向数字
        CLR.B   D1          *清空和
Loop    ADD.B   (A0)+,D1    *重复循环并累加数字到总和
        SUB.B   #1,D0       *减少循环计数
        BNE     Loop        *直到所有的数字都被加
        STOP    #$2700

Table   DC.B    1,4,2,6,5   *一些虚构的数据

        END     $400
```

模式3和模式4在操作数从存储器中取得后自动增加地址寄存器的值。如果相应的操作分别是针对字节、字或长字的，则寄存器的值就分别递增1个、2个或4个字节。

让我们总结一下主要的寻址模式：

207

- 寄存器直接寻址用于可以保存在寄存器中的变量。
- 文字（立即）寻址用于不可改变的常数。它主要的用途是初始化变量。
- 直接（绝对）寻址用于直接指定存储器中变量的地址。尽管概念上简单，但它不允许程序重定位。
- 地址寄存器间接寻址用于当地址需要计算或顺序寻址时。
- 带前递增或后递减的地址寄存器间接寻址用于顺序数据操作或堆栈的PUSH和POP操作。

寄存器直接寻址和直接寻址的唯一区别在于前者使用寄存器存储操作数而后者使用存储器。

让我们来看一个简单的汇编语言程序，该程序总结了我们刚刚所讨论的大多数概念。假定我们有一序列字节存储在从标记为“data”位置开始的连续存储位置。图8-5是算法的流程图。

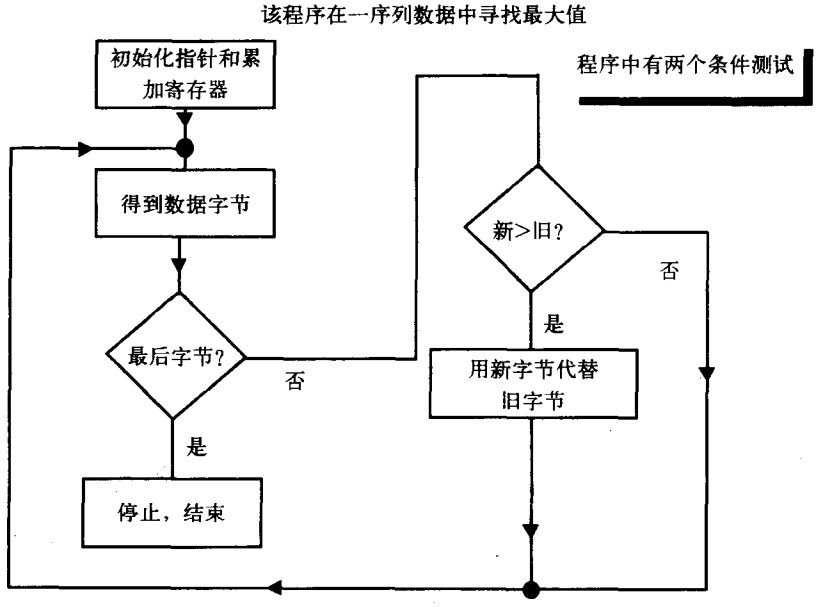


图8-5 寻找一序列数据中的最大值且NULL字节必须出现的一个算法的流程图

在该程序中，我们假设字节串以一个包含00的字节结束，我们将称这个字节为NULL字节，这与C语言中字符串的结束方式是一样的。程序允许存储器中有长度为0的数字串，但是NULL字节必须出现。

程序如下：

```
*****************************************************************************************
* 在字节数组中寻找最大值的程序
*
* 数组从存储位置$1000开始并以一个NULL字节（$00）结束
*
*****************************************************************************************
1  start     ORG      $400              * 程序开始处
2            LEA      data,A0           * 使用A0作为一个指针
3            CLR.B    D0                * D0将保存最大的字节，将它清零
4  next      MOVE.B   (A0)+,D1          * 循环，读取一个字节
5            BEQ      exit              * 是不是NULL字节?
6            CMP.B    D0,D1             * D1-D0，看是否新>旧
7            BLE      end_test          *
8            MOVE.B   D1,D0             * 然后旧=新
9  end_test  BRA      next              * 返回并取得下一个字节
10 exit      STOP     #$2700            * 返回到模拟器
```

208

```
11
12              ORG     $1000                   * 数据区域
13 data         DC.B    12,13,5,6,4,8,4,10,0    * 样本数据
14              END     $400                    * 程序终止和入口
```

让我们来分析这个程序。注意最左列中的数字在实际程序中是不应该有的，加入它们只是用来帮助识别每一行指令。还要注意实际程序代码前星号括起来的一块，这是用来描述算法的，与C++函数前的注释块差不多。

1. ORG伪指令指定程序的开始。

2. LEA（装载有效地址）用数据的存储地址初始化地址寄存器A0。实质上，我们是将一个指针赋值给数组变量data。我们将把指针保存在A0寄存器。在前面一个代码片段中我们使用MOVEA代替LEA将一个值载入了地址寄存器。如果你是为以后使用而装载一个地址寄存器，那么LEA是首选的指令。

3. CLR.B被用来清零寄存器D0的内容，使得我们可以正确地测试一个更大的数。而且，若遇到数组长度为0，我们不会报告一个错误的值。

4. MOVE.B将A0所指的字节拷贝到寄存器D1。由于我们使用的是带后递增的寄存器间接寻址模式，所以当数据被取回之后，A0的内容自动递增，指向字符串中的下一个字节。

5. BEQ进行测试，看我们取回的字节是不是NULL字节。如果是，就进行转移，转向程序的出口。如果测试失败，则向后执行下一条指令。

6. CMP.B检查新字节是否大于当前的最大字节。

7. BLE执行前面CMP.B指令的结果。如果新字节不大于D0中的当前字节，它就跳过下一条指令返回到循环的开始处。如果大于，则执行下一条指令。注意行6和行7是怎样配对成一条指令的，该指令用一条指令比较两个值并根据比较结果进行转移。

8. MOVE.B用新的最大值来代替D0中的值。

9. BRA总是返回到循环的开始处。

10. STOP将我们带回到模拟器。

11. ORG将重定位指令计数器，在地址$1000及其以上部分载入汇编代码。

12. DC.B是将样本数据载入到标号“data”定义的存储位置的伪操作代码。

13. END告诉汇编器停止汇编并装载程序从$400处开始运行。

8.2 汇编语言和C++

直到现在，我们都是孤立地看待汇编语言，就像我们在学习一种新的神秘的编程语言一样，这么说是有几分道理的。然而，别忘了实际上很多高级语言的编译器都输出汇编语言，然后这些编译器的汇编语言输出被汇编成目标代码。这意味着我们在其他编程语言课程中学习过的编码结构也必须翻译成相对应的汇编语言结构。 209

C和C++已有用于改变程序流程的内置结构。**IF/ELSE WHILE DO/WHILE SWITCH FOR**以及函数调用都可以改变程序的流程。在汇编语言中，我们必须使用有效的汇编语言指令来建立我们自己的结构，这些有效的汇编指令为转移（**Branch**）、跳转（**Jump**）、跳转到子程序（**Subroutine**，函数调用）。

当我们考察编译器是怎样处理一些非常熟悉的C++结构时，我们就会看到这些结构仍然存在于汇编语言中。首先回顾可知，汇编语言的比较指令（**CMP CMPI**和**CMPA**）为了设置正确的条件代码标志而将两个操作数相减，但并不保存结果。因此，如果<D0> = 10且<D1> = 10，那么指令：

```
CMP.B   D0, D1
```

将从数据寄存器D0的内容中减去数据寄存器D1的内容。由于这两个寄存器都包含数字10，所以这个减法的结果等于0，伴随的结果是Z = 1。然而，与实际的减法指令不同，

```
SUB.B   D0, D1
```

通过这个比较操作，D0和D1的内容并不改变。让我们比较两个代码片段。第一个是如下所示的C++的一个简单的IF结构：

```
int a = 3, b = 5;
if( a == b)
    { 执行该代码 };
else
    { 执行该代码 };
```

编译器可能会将该程序转换为如下的汇编语言代码片段：

```
          MOVE.L   #3,D0
          MOVE.L   #5,D1
          CMP.L    D0,D1
          BEQ       equal
not_equal { 执行该代码 }
equal     { 执行该代码 }
```

汇编语言中的循环结构与C++循环结构相似。详细地学习这些循环结构是有益的，因为循环结构是较难的几种汇编语言代码结构之一。让我们来看一个简单的FOR循环结构：

```
for ( int counter = 1; counter < 10 ; counter++ )
       { 执行这些语句 }

          MOVE.L  #1,D0                    * D0是计数器
          MOVE.L  #10,D1                   * D1保存终止值
for_loop  CMP.B   D0,D1                    * 执行测试
          BEQ     next_code                * 我们已经完成了吗?
          { 执行另外一些循环指令 }          * 递增计数器
          ADDQ.B          #1,D0
          BRA             for_loop         * 返回
next_code { 执行循环后的指令 }
```

210

*DO/WHILE*结构在C++中的使用也很普遍。指令的形式是：

```
DO
    { 执行这些语句 }
WHILE ( 测试条件为真 )
```

汇编语言的类似结构如下所示：

```
          MOVEA.W  #start_addr,A2     * 初始化循环条件
test_loop JSR      subroutine         * 这是C++中的一次函数调用
          CMPI.B   #test_value,(A2)+  * 比较新值
          BNE      test_loop          * 测试条件仍然为真
          { 后续的指令集合 }          * 循环后的代码
```

注意函数调用*JSR* subroutine至少发生一次，因为这是一个*DO/WHILE*循环结构。指令

```
CMPI.B   #test_value, (A2)+
```

是这个相等测试的核心。这里的值test_value与地址寄存器A2所指存储位置处的值进行比较。指令执行后，寄存器A2的值自动递增以指向下一个存储地址。因此，自动递增存储指针A2就为我们提供了再次进入循环后用来测试的下一个值。

让我们在汇编语言算法的分析上再多做一次练习。在这个例子中，我们将使用一个非常简单的汇编语言代码例子并不断改进它使之尽可能高效。在这个例子中，高效性将通过代码紧凑性和执行速度来衡量。下面是问题的陈述：

将数据值$FF写入从$1000到$1005的存储地址处，包括$1000和$1005。

例1 蛮力法

```
*****************************************************************
* 该程序将FF存入$1000到$1005的存储地址
*****************************************************************
* 系统的等于伪指令

load_val    EQU     $FF          * 要存储的字节
pgm_start   EQU     $400         * 程序从这里开始运行
stack       EQU     $2000        * 把堆栈放在这里

            ORG     pgm_start
            LEA     stack,SP          * 初始化栈指针
            MOVE.B  #load_val,$1000   * 存储第一个值
            MOVE.B  #load_val,$1001   * 存储第二个值
            MOVE.B  #load_val,$1002   * 存储第三个值
            MOVE.B  #load_val,$1003   * 存储第四个值
            MOVE.B  #load_val,$1004   * 存储第五个值
            MOVE.B  #load_val,$1005   * 存储第六个值
            STOP    #$2700            * 返回到模拟器
            END     start             * 在此处停止汇编
```

在我们分析程序之前，我们先看一条比较奇怪的指令，即指令：

```
LEA         stack, SP          * 初始化栈指针
```

将定义为stack（存储位置$2000）的变量的地址放入叫做SP的地方。SP是汇编器指定寄存器A7或A7'（即堆栈指针）的一种方式。我们将在后面更详细地讨论堆栈指针。现在我们接受这个信念，就是有必要将栈指针定位在高存储位置，并且我们应该把它作为程序首先要做的事情之一，还要注意我们是使用*装载有效地址*（LEA）指令来完成这个任务的。 211

现在，让我们来分析这个程序。例1中的程序是可以运行的，但它还远远称不上高效。MOVE.B指令所有的目的操作数都是绝对地址。我们在浪费指令的存储空间，因为我们为每次数据传送严格地指定了存储地址。假设我们要移动600 000字节而不是6个字节，那么很明显这个方案就不可行了。

我们可以通过使用一个数据寄存器来保存移至存储器的数据值$FF，还可以使用一个地址寄存器来指向我们想要存储数据的存储位置来改进例1。这可以为我们节省时间和空间，因为数据$FF现在存储在寄存器中供我们使用，而不是每次从存储器中重新取回它。缺点是我们必须在初始化时给一些寄存器装载值，这就需要一些额外的指令。不过，由于使用短指令，执行时间较短，装载寄存器的开销将被抵消一部分。

例2 使用寄存器

```
*****************************************************************
* 该程序将FF存入$1000到$1005的存储地址
*****************************************************************
* 系统的等于伪指令
load_val        EQU     $FF          * 将要存储的字节
pgm_start       EQU     $400         * 程序从这里开始运行
stack           EQU     $2000        * 把堆栈放在这里
start_addr      EQU     $1000        * 存储的首地址

* 程序从这里开始

                ORG     pgm_start
                LEA     stack,SP          * 初始化栈指针
                LEA     start_addr,A0     * 设置A0为栈指针
                MOVE.B  #load_val,D0      * 将它放入到数据寄存器中
                MOVE.B  D0,(A0)           * 存储第一个值
                ADDA.W  #01,A0            * 指向下一个地址
                MOVE.B     D0,(A0)        * 存储第二个值
```

```
ADDA.W   #01, A0        *指向下一个地址
MOVE.B   D0, (A0)       *存储第二个值
ADDA.W   #01, A0        *指向下一个地址
MOVE.B   D0, (A0)       *存储第三个值
ADDA.W   #01, A0        *指向下一个地址
MOVE.B   D0, (A0)       *存储第四个值
ADDA.W   #01, A0        *指向下一个地址
MOVE.B   D0, (A0)       *存储最后一个值
STOP     #$2700         *返回到模拟器
END      pgm_start      *停止汇编
```

这个方案更好，因为移动数据的每条指令只取决于寄存器的内容。我们能否做得更好呢？是的，我们可以通过使用自动递增寻址模式（官方称为*带后递增的地址寄存器间接寻址模式*）来删掉指令：

```
ADDA.W          #01, A0
```

例3 *自动递增*

212

```
*****************************************************************
* 该程序将FF存入$1000到$1005的存储地址
*****************************************************************
* 系统的等于伪指令
load_val        EQU         $FF         * 将要存储的字节
pgm_start       EQU         $400        * 程序从这里开始运行
stack           EQU         $2000       * 把堆栈放在这里
start_addr      EQU         $1000       * 存储的首地址

* 程序从这里开始

                ORG         pgm_start
                LEA         stack,SP            * 初始化栈指针
                LEA         start_addr,A0       * 设置A0为指针
                MOVE.B      #load_val,D0        * 将它放入数据寄存器
                MOVE.B      D0,(A0)+            * 存储第一个值并递增
                MOVE.B      D0,(A0)+            * 存储第二个值并递增
                MOVE.B      D0,(A0)+            * 存储第三个值并递增
                MOVE.B      D0,(A0)+            * 存储第四个值并递增
                MOVE.B      D0,(A0)+            * 存储第五个值并递增
                MOVE.B      D0,(A0)             * 存储最后一个值
                STOP        #$2700              * 返回到模拟器
                END         pgm_start           * 在此处停止汇编
```

例3是否已经做到最好了呢？这是一个值得深入分析的有趣问题。为什么呢？因为对于一个具有6条数据写语句的简单程序，试图用循环结构来使代码更加紧凑可能实际上会降低而不是提高性能。但是，如果我们正在移动的是很多数据，那么很明显这个内联算法将不能运行。

从计算机性能的角度来看，关于内联算法与循环结构的对比问题是非常有趣的。当计算机能够执行大块的内联代码且不需要进行转移或循环时，计算机的效率是最高的。编译器常常将for循环转换为内联代码（如果迭代的数字在可处理的范围内）以提高性能。在后面有一课中讨论流水线时，我们就会明白其中的原因。

总之，让我们插入一个循环结构来看看是否提高了效率。

例4 *DO/WHILE循环结构*

```
*****************************************************************
* 该程序将FF存入$1000到$1005的存储地址
*****************************************************************
* 系统的等于伪指令

load_val        EQU         $FF         * 将要存储的字节
```

```
pgm_start       EQU         $400        * 程序从这里开始运行
stack           EQU         $2000       * 把堆栈放在这里
start_addr      EQU         $1000       * 存储的首地址
end_addr        EQU         $1005       * 存储的最后地址
                ORG         pgm_start

                LEA         stack,SP        * 初始化栈指针
                LEA         start_addr,A0   * 设置A0为指针
                LEA         end_addr,A1     * A1将保存终点
                MOVE.B      #load_val,D0    * 将它放入数据寄存器
loop            MOVE.B      D0,(A0)+        * 存储值并递增
                CMPA.W      A1,A0           * 是否已完成?
                BLE         loop            * 没有完成，跳回
                STOP        #$2700          * 返回模拟器
                END         pgm_start       * 在此处停止编译
```

213

我们已经设法将算法的实际指令从10条减少到了8条（不计伪代码）。这虽然更好，但是对于这个简单算法来说，循环的价值被创建它的费用所掩盖了。然而，如果循环要执行大约1 000 000次左右，那么将算法写成内联代码是不可能的。让我们来看看这个带有*for*循环结构的算法。

例5 *for*循环结构

```
*****************************************************************
* 该程序将FF存入$1000到$1005的存储地址
*****************************************************************
* 系统的等于伪指令
load_val        EQU         $FF         * 将要存储的字节
pgm_start       EQU         $400        * 程序从这里开始运行
stack           EQU         $2000       * 把堆栈放在这里
start_addr      EQU         $1000       * 存储的首地址
loop_ctr        EQU         6           * 循环的次数

                ORG         pgm_start

                LEA         stack,SP        * 初始化栈指针
                LEA         start_addr,A0   * 设置A0为指针
                MOVE.B      #loop_ctr,D1    * D1将保持跟踪
                MOVE.B      #load_val,D0    * 将它放入数据寄存器
loop            MOVE.B      D0,(A0)+        * 存储值并递增
                SUBQ.B      #01,D1          * 递减计数器
                BNE         loop            * 是否已完成?
                STOP        #$2700          * 返回模拟器
                END         pgm_start       * 在此处停止编译
```

计数指令告诉我们，for循环结构和do/while循环结构给出了相同的结果。但是，为了真正知道是否一种方法比另一种更好，我们需要仔细看看每条指令并计算实际需要的时钟周期数。也只有这时候我们才能够知道到底for循环比do/while更高效还是更低效。

214

此时我们可以说的就是我们已经做得足够好了。我们还能做得更好吗？可以，但如何做得更好并不是显而易见的。答案在于需要使用一条更复杂的指令——DBcc指令。请参考Motorola程序员参考手册中关于该指令的完整讨论。这是一条很难掌握的指令，但是它将下面两条指令组合成了一条指令：

```
SUBQ.B          #01,D1      * 计数器递减
BNE             loop        * 是否已完成?
```

通过使用BDcc指令我们可以使程序更加紧凑。

例6 使用DBcc指令

```
*****************************************************************
* 该程序将FF存入$1000到$1005的存储地址
*****************************************************************
* 系统的等于伪指令

load_val        EQU         $FF         * 将要存储的字节
pgm_start       EQU         $400        * 程序从这里开始运行
stack           EQU         $2000       * 把堆栈放在这里
start_addr      EQU         $1000       * 存储的首地址
loop_ctr        EQU         5           * 循环的次数

                ORG         pgm_start

                LEA         stack,SP        * 初始化栈指针
                LEA         start_addr,A0   * 设置A0为指针
                MOVE.B      #loop_ctr,D1    * D1将保持跟踪
                MOVE.B      #load_val,D0    * 将它放入数据寄存器
loop            MOVE.B      D0,(A0)+        * 存储值并递增
                DBF         D1,loop         * 循环计数器递减，如果D1! = -1则转移
                STOP        #$2700          * 返回模拟器
                END         pgm_start       * 在此处停止编译
```

这段程序也许如其所达到的那么好。DBF指令是非常复杂且非常难掌握的指令，但是一旦你掌握了它，你就可以把它加到你的工具箱，当需要节省一两个时钟周期的时候你就可以使用它。该指令是这样运行的：首先，它要么测试条件码标志（DBcc），要么被迫总为假（DBF）。每次通过循环，如果条件为假，数据寄存器（这里指D1）就递减。循环在两种情况下退出：

1. 条件为真；

2. 数据寄存器的值 = −1。

因此，指令**DBF D1, loop**保证转移返回“loop”并递减D1直到<D1> = −1，然后退出循环。由于我们使用的总是测试为假的指令形式，所以我们总是转移返回。

让我们来复习一下我们刚刚所学的东西。在这一系列中，我们从一个非常简单明了的方
215 案（取到数据并把它放在这里）一直走到了一个非常紧凑一流的方案。在这个过程中，我们从简单指令和寻址模式走到了更加复杂的指令、寻址模式和算法结构。为了使程序尽可能紧凑，我们使用了一条相当复杂的指令DBF来组合寄存器递减操作和测试转移操作。下面的表格显示了前面使用的各种指令的时钟周期数和执行时间（假定为16MHz的时钟频率）。

指　令	时钟周期	指令执行时间（微秒）
MOVE.B #$FF, $1000	28	1.75
MOVE.B D0, $1000	20	1.25
ADDA.W #01, A0	12	0.75
MOVE.B D0, (A0)	8	0.5
MOVE.B D0, (A0)+	8	0.5

将字节值$FF移动到每个存储位置所需的时钟周期数有一个3.5的因数，该因数取决于数据每次是从存储器装载并写回特定存储位置还是使用数据和地址寄存器来存储和操作该值。

8.3 堆栈和子程序

堆栈的概念是计算机管理其数据和地址的基础。堆栈是后进先出（LIFO）数据结构的一个实例。在68K体系结构中，有两个专用寄存器A7和A7’，它们是专用于堆栈操作的。A7是用户堆栈指针（user stack pointer），在汇编语言指令中它一般被称为SP而不是A7，A7’是管

理堆栈指针（supervisor stack pointer）。

实际上，管理堆栈指针与管理模式（supervisor mode）下的一系列专用指令一起都是保留给操作系统使用的。通常，我们的程序运行在比操作系统低的优先级。我们在用户模式（user mode）下运行程序，并自动访问A7寄存器作为堆栈指针而不是管理堆栈指针。但是，由于我们运行在模拟器上的程序十分简单，并不要求使用操作系统（除了你的PC操作系统），所以我们就总是运行在管理模式下并使用管理堆栈指针进行操作。管理模式和用户模式都称栈指针为SP，你会得到哪个堆栈指针取决于你处于哪个模式下。图8-6总结了堆栈的操作。

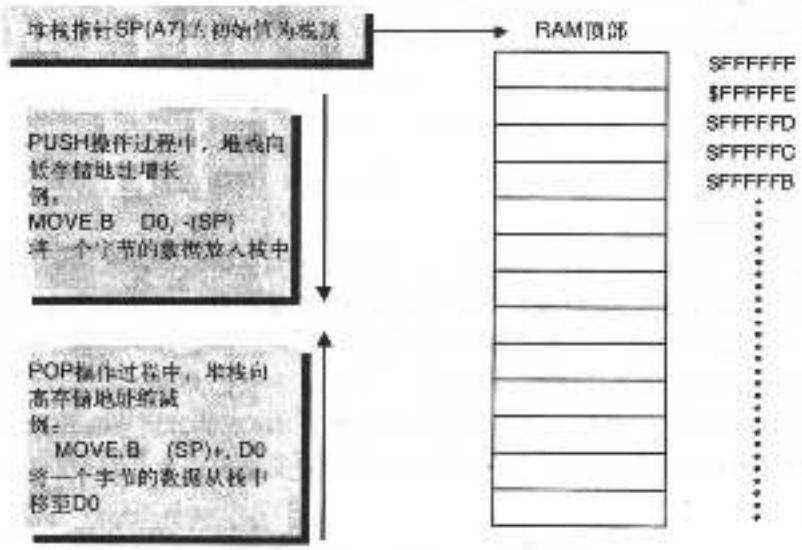


图8-6 68000堆栈指针寄存器A7的操作

前递减寻址模式和后递增寻址模式的原因现在已经显而易见了。堆栈指针总是指向栈顶的存储地址。当其他数据需要放入存储器时，指针必须先递减以指向下一个有效地址，从而保证当发生数据存储时，数据移至空闲存储位置。后递增模式访问堆栈中的当前存储地址然后再递增，所以SP指向栈中存储位置的前一项。

现在我们已经说明了堆栈是怎样实现LIFO数据结构的，接下来我们将转向子程序的学习，汇编语言中简单明了的子程序与C语言中的函数调用等价。当程序碰到跳转到子程序（JSR）指令时，处理器会自动将下一条指令的长字地址压入栈中，然后再跳转到操作数所指定的位置。由于程序计数器总是指向待取的下一条指令，所以JSR指令的作用就是首先将程序计数器 216
的内容移至栈指针所指的当前位置，然后用新的操作数地址装载程序计数器。这可以有效地使得程序“跳转”到存储器中的新位置。

从子程序返回（RTS）的指令被放在子程序的最后。它使得堆栈将返回地址返回给PC，下一条指令从程序跳转到子程序的地方开始执行。

当你用C或C++编写程序时，编译器将管理过程调用或函数调用时的所有常规事务。在汇编语言中，程序员应该：

- 保证所有后来的压栈和退栈操作成对出现。
- 保证子程序用到的所有资源（寄存器和存储器）在子程序使用它们之前要正确保存下来，在子程序返回时又要将它们恢复。
- 决定在子程序和主程序之间进行参数传递的一种机制。

通常，一个子程序很可能需要用到主程序或另一个子程序正在使用的寄存器资源。68K处理器有一套用来管理这个的机制。**MOVEM**指令就是用来快速指定那些在进入子程序时需要保存在栈中且从子程序退出时又要恢复的一系列寄存器。通过在入口处将子程序中将要用到的寄存器保存下来然后在退出之前恢复它们，子程序就可以自由地使用这些寄存器而不会破坏正被其他过程使用的数据。因此，在进入子程序时通常没有必要保存所有的寄存器，当然若全保存也没有什么害处。一旦你决定了哪些寄存器不会在你的子程序中用到，你就可以简化代码了。还有，就像在C语言中一样你可能需要将参数传进传出子程序，但是，决定怎样来完成参数的传进传出是你的任务，因为没有编译器会为你提供一套规则。

因此，在子程序的入口，使用

```
MOVEM          <寄存器列表>, -(SP)
```

就可以将寄存器压入栈中，而在子程序的出口，使用

```
MOVEM          (SP)+, <寄存器列表>
```

217 就可以将寄存器从栈中退出。

值得告诫的是：使用寄存器将参数传入一个子程序和使用寄存器从子程序返回结果是非常方便的。C/C++的关键字return通常会使编译器将函数调用的结果放入指定寄存器然后再从子程序返回。如果你想要将一个寄存器中的结果返回，那么你最好别用那个特定的寄存器来完成子程序入口处的保存和出口处的恢复操作。为什么呢？因为当你在子程序出口恢复寄存器时你会把要返回的结果覆盖掉。很多学生曾经花费了大半夜的时间来查找这个特别的错误。

通过使用*REG*这条汇编伪指令，创建寄存器列表会变得更容易。它允许定义一系列寄存器，格式是：

```
<标识符>      REG      <寄存器列表>
```

寄存器可以被指定为单个的寄存器（A0或D0）、用斜线分割的列表（即A1/A5/A7/D1/D3），也可以被指定为寄存器范围（如A0—A3）。举个例子，下面的语句定义了一个叫做“save_reg”的寄存器列表

```
save_reg       REG       A0-A3 / A5 / D0-D7
```

寄存器按照一种固定的次序自动保存和恢复，REG指令中指定的范围没有明确规定其次序。请参考你的程序员手册以得到**MOVEM**指令的一个全面解释（尽管难以理解）。图8-7总结了进入和退出子程序时的资源保存和恢复操作。

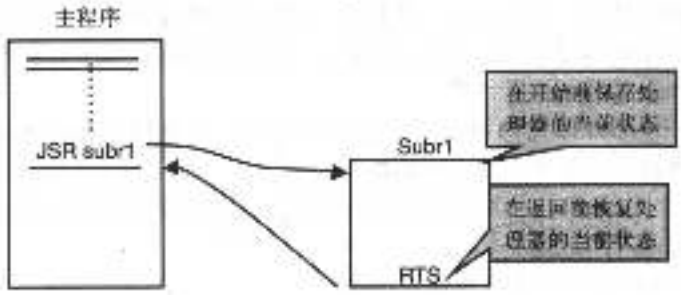


图8-7 子程序的资源管理

就像C语言中的函数调用一样，子程序可能是嵌套的。重要的是牢记子程序需要谨慎的堆

栈管理，因为返回路径取决于存储在堆栈中的正确的返回地址序列。子程序应该总是返回到它们被调用的地方，也就是JSR指令的下一条指令。但是，跳转表（jump table）是这个规则的少数例外之一，我们将在后面的一课中学习跳转表。图8-8说明了子程序的嵌套。

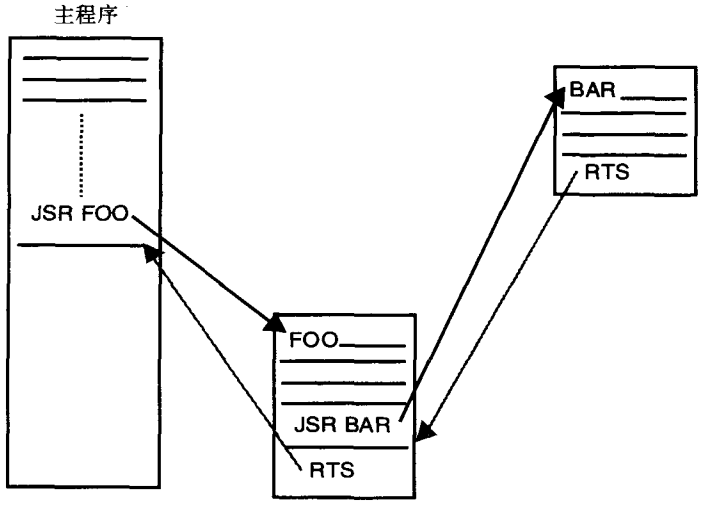


图8-8 嵌套子程序。RTS指令应该总是用来返回到最近的JSR指令后的那条指令

子程序可以使编码更高效，因为相同的代码块可以被使用很多次。另一个选择是所有的代码都用内联方式编写。这里是编写和使用子程序时的几点指导方针：

1. 在试图执行JSR指令之前必须确立用户栈。
2. 将子程序放在程序主体部分之后，但在数据存储区域之前。
3. 每个子程序应以一个注释块开头，列出：

a）子程序名字
b）完成什么工作
c）使用和保存的寄存器
d）输入和返回的参数

4. 子程序的第一行必须有带有子程序名字的标号。汇编器正是这样插入目的操作数的有效地址的。
5. 在子程序的入口保存将要用到的寄存器。
6. 退出子程序时恢复寄存器。
7. 总是返回子程序被调用的地方。换句话说，不要压入一个新的返回地址到栈中，也不要从子程序跳转或转移到其他地方而将返回地址留在栈中。
8. 子程序允许嵌套，就像C语言中的函数调用一样。

一个示例程序：具体细节

我们已经花费了相当多的时间来分析主要的寻址模式和学习一些编程概念，现在就可以更详细地看两个例子了。该程序将我们到目前为止所学的所有概念都放在了里面，是一个完整的程序。

对于下一个示例程序，让我们来编写一个能真正做点有意义事情的程序。它将检查系统中存储器的完整性。这是一个非常普通的程序，在很多计算机系统中被广泛使用，特别是在系统进行自检时的引导时间内。每次当你开机或按下重启键（RESET）时PC都会执行该程序。我们的计划是，通过一个存储器测试程序示例来考察到目前为止所讨论的在实际程序中使用

218

的所有知识点。我们的目标是：

- 研究汇编语言程序的结构；
- 了解怎样使用注释；
- 了解怎样使用等于伪指令（EQU）来定义常数；
- 了解怎样使用标号来定义存储位置；
- 研究实际中怎样使用伪操作指令（ORG、DS、DC、END）；
- 了解怎样使用装载有效地址（LEA）命令；
- 观察程序中用到的各种寻址模式；

219

- 了解比较指令是怎样与转移指令组合来测试和更改程序流程的；
- 观察子程序的头模块；
- 了解怎样使用子程序，以及参数是怎样传递的。

作为介绍的一种方式，让我们讨论一下程序在做什么。程序已经被固定地编写成总是测试从$2000到$6000的存储区域，尽管开始地址和结束地址在一定程度上是任意的，而且我们可以很容易地通过“等于伪指令”改变被测试的范围。

程序将一个字节值写出到存储器然后立即将它读回，读回的值应该与写出的值相同。如果不相同，就可能存在一个坏的存储位置，或者也可能是断掉的或短路了的存储线。作为确定问题起因的一种方法，我们使用4个不同的字节值：$00、$FF、$55和$AA，它们分别代表所有的数据线为低、所有的数据线为高、偶数据线低且奇数据线高以及偶数据线高且奇数据线低。我们也可以通过保存存储器测试中失败的10个存储位置来跟踪检测到的失败次数。

程序的第一部分包含了系统的等于伪指令，可以把这当作头文件，这里是你存放所有#define 语句的地方。应该注意的是几乎每个常数或初始值都在这里被定义并被给予一个符号名。还要注意坏位置数的最大值*maxcnt*被定义成了一个十进制数，因为这样它的含义会更清晰。汇编器会将它转换为十六进制的。

```
*************************************************************
*
* 存储器测试程序
*
* 这是一个用来测试从字节地址$2000到$6000处存储器的程序。它使用了
* 4个测试码：00、$FF、$AA、$55
* 并且它能存储多至10个的坏地址位置
*
*************************************************************
*
* 系统的等于伪指令
*
end_test    EQU    $11      * 测试向量终止处
test1       EQU    00       * 第一个测试码
test2       EQU    $FF      * 第二个测试码
test3       EQU    $55      * 第三个测试码
test4       EQU    $AA      * 第四个测试码
st_addr     EQU    $2000    * 测试开始地址
end_addr    EQU    $6000    * 测试结束地址
stack       EQU    $7000    * 堆栈位置
maxcnt      EQU    10       * 坏地址的最大数量
```

220

下列代码块中的黑体部分说明了怎样使用*装载有效地址*（**LEA**）指令来初始化一个栈指针或地址寄存器中的一个地址。子程序调用指令**JSR**被初始化。两个符号变量**tests**和**bad_cnt**是程序末尾处的数据存储位置。

```
            ORG       $400          * 程序开始
start       LEA       stack,SP      * 初始化栈指针
            CLR.B     D0            * 初始化D0
            CLR.B     bad_cnt       * 初始化坏地址计数
            LEA       tests,A2      * A2指向测试码
            LEA       bad_addr,A3   * 指向坏地址计数存储处
test_loop   MOVE.B    (A2)+,D6      * 用D6来测试完成码
            CMPI.B    #end_test,D6  * 是否完成?
            BEQ       done          * 是的，退出
            LEA       st_addr,A0    * 在A0中建立开始地址
            LEA       end_addr,A1   * 在A1中建立结束地址
            JSR       do_test       * 跳到测试
            MOVE.B    bad_cnt,D7    * 得到当前的计数
            CMPI.B    #maxcnt,D7    * 是否已超过最大值
            BLT       test-loap     * 继续
done        STOP      #$2700        * 返回到模拟器
```

注意我们保存了在子程序中使用的寄存器。我们不需要保存A0、A1和A2是因为它们是用来传递我们使用的地址参数的。

```
*************************************************************************
*
* 子程序do_test
*
* 该子程序做实际的测试。
* A0保存开始地址。
* 保存结束地址。A2指向本次测试中使用的测试码。
*  该程序将测试从A0到A1的存储位置，将任意失败存储位置的地址存入bad_addr并递增
* bad_cnt的计数。如果计数超过10测试就会终止。
*
*************************************************************************
do_test     MOVEM.W   A3/D1/D7,-(SP)  * 保存寄存器
check_loop  MOVE.B    (A2),(A0)       * 写字节
            MOVE.B    (A0),D1         * 用D1保存所写的值
            CMP.B     (A2),D1         * 进行比较
            BNE       error_byte      * 更新计数器
            BRA       next_test       * 没有问题，继续测试
error_byte  MOVE.W    A0,(A3)+        * 存储地址并递增指针
            ADDI.B    #01,bad_cnt     * 递增坏地址计数
            MOVE.B    bad_cnt,D7      * 是否已超出?
            CMPI.B    #maxcnt,D7      * 检查
            BGE       exit            * 返回，已经完成。
next_test   ADDA.W    #01,A0          * 递增A0
            CMPA.W    A0,A1           * 测试我们是否已完成
            BGE       check_loop      * 跳回并测试下一个地址
            MOVEM.W   (SP)+,A3/D1/D7  * 恢复寄存器
exit        RTS                       * 返回到测试程序
```

221

数据存储区域包括测试码和用来存储计数和坏地址的保留存储区域。注意**END**指令出现在所有源代码的最后，而不只是程序代码的最后。end_test定义的值与C语言用来终结字符串的*NULL*字符相似，每次通过测试循环我们都要检查这个字符以知道程序是否已完成。

```
* 数据存储区域

tests       DC.B  test1,test2,test3,test4,end_test  * 测试码
padding DC.B  00                                    * 填充函数
bad_cnt     DS.W  1                                 * 坏地址计数
bad_addr    DS.W  10                                * 保留10个地址的空间
            END   $400                              * 程序结束点并装载地址
```

建议的练习

仔细阅读代码然后建立流程图来描述它是如何工作的。然后，创建一个源文件并在模拟

器中运行程序。为了测试程序，将测试的结束地址改为适当接近开始位置的地方，或许离开始位置10或20个字节的位置。下一步，汇编该程序，并使用列表文件，在子程序中将数据写到存储器的指令上设置断点。使用跟踪指令将数据写到存储器，但在再次开始跟踪程序之前改变存储在存储器中的值。换句话说，强迫使测试失败。查看程序流程并确认程序正按照你所预料的方式运行。如果你不太确定为什么要使用一些特定指令或寻址模式，那么请复习一下你的笔记或参考你的*程序员参考手册*。最后，使用这个程序作为一个框架，看看你能不能使用另外一些寻址模式或指令来改进它。请为这个练习安排足够多的时间，它相对于我们目前所学的所有编程概念来说是非常基础的。

总结

本章讲述了以下主题：

- 计算机中负数和实数是怎样表示和操作的。
- 转移和基于CCR中标志状态的条件代码执行的一般过程。
- 68K体系结构的主要寻址模式。
- 高级语言循环结构以及它们在汇编语言中的类似结构。
- 在汇编语言程序设计中使用子程序。
- 一个用来测试存储器的汇编语言程序的详细讲解。

参考文献

222

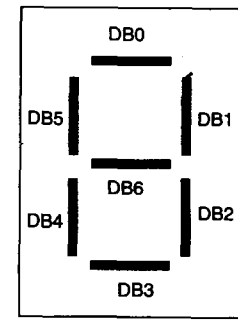
[1] Alan Clements, *68000 Family Assembly Language*, ISBN 0-5349-3275-4, PWS Publishing Company, Boston, 1994, p. 29

习题

1. 下面所示的是个7段码显示器的示意图。所示表格表示的是在显示器上显示相应数字的二进制代码。因此，为了在显示器上显示数字“4”，你应该将DB1、DB2、DB5和DB6置为逻辑电平1，而其他所有数据位置为逻辑电平0。
   - 显示器存储器映射到字节地址$1000。
   - 地址$1002处有一个硬件定时器。
   - 通过将1写到DB4将定时器启动。DB4是一个只写位，读取它将总会得到DB4 = 0的结果。
   - 当定时器被启动时，DB0（$\overline{BUSY}$）下降并保持为低500ms。500ms后，定时器超时，DB0再次上升。
   - DB0是只读的且对它写不会影响定时器。所有的其他位都可以忽略。定时器控制寄存器如图所示。

   编写一个较短的68K汇编语言子程序，该程序从寄存器D0.B传送给它的数据开始进行反计数到零。反计数的当前状态显示在7段码显示器上。反计数的

计数	二进制位码 DB7 DB0
0	00111111
1	00000110
2	01011011
3	01001111
4	01010110
5	01101101
6	01111101
7	00000111
8	01111111
9	01100111



DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
X	X	X	START	X	X	X	$\overline{BUSY}$

X = 无关项

速率是每两秒一个数字。

注：

- 这是一个子程序。没有必要使用ORG来开始程序，也不需要建立堆栈指针或使用END伪指令。
- 你可以假设传递给D0.B的数据在1到9的范围内。你不必做任何错误检测。当计数器到达0时子程序退出。

2. 假设某个外部设备给你的计算机传送了一系列字节值，伴随的还有一个检验和值(checksum value)，可用来检验接收到的字节序列与发送的字节序列是否相同。为了做到这一点，你必须对你接收到的字节流计算其检验和，并将该值与传送给你的检验和进行比较。如果它们相等，那么传送过程就很可能没有错误。 223

A部分：编写一个汇编语言子程序，而不是一个程序，用它来计算连续存储地址中的字节流的检验和。检验和就是简单地取所有字节值的和，这就很像对一列数据求和。检验和值是一个16位值。任何超出16位的溢出和进位都会被忽略。

1）子程序所需信息通过如下所示的过程传给子程序：

a. 寄存器A0 = 存储器中字节串的指针，用一个长字表示。

b. 寄存器D0 = 传给子程序用来比较的检验和，用一个字值表示。

c. 寄存器D1 = 字节流的长度，用一个字值表示。

2）检验和计算过程中，任何超过16位的溢出和进位都会被忽略。只有由求和得到的字值与检验和相关。

3）如果计算得到的检验和与传送来的检验和相同，那么地址寄存器A0返回一个指针，指向字符串开始的地方。

4）如果检验和比较失败了，A0中的返回值就被置为0。

5）除了地址寄存器A0之外，所有的寄存器从子程序返回时都应当保持它们的初始值。

B部分：如果在字节流中存在错误但是却不能用这种方法检测到的可能性是多大？

3. 高亮指令完成后，寄存器D0中的值是什么？

```
00000400 4FF84000      START       LEA         $4000,SP
00000404 3F3C1CAA                  MOVE.W      #$1CAA,-(SP)
00000408 3F3C8000                  MOVE.W      #$8000,-(SP)
0000040C 223C00000010              MOVE.L      #16,D1
00000412 203C216E0000              MOVE.L      #$216E0000,D0
00000418 E2A0                      ASR.L       D1,D0
0000041A 383C1000                  MOVE.W      #$1000,D4
0000041E 2C1F                      MOVE.L      (SP)+,D6
00000420 C086                      AND.L       D6,D0
00000422 60FE          STOP_HERE   BRA         STOP_HERE
```

4. 编写一个满足下列描述的子程序，子程序将下列变量作为它的输入参数列表：

- 寄存器A0中的一个长字存储地址，其中A0指向已经存在于存储器的32位整数流的第一个元素。
- 寄存器D1中的一个32位长字值，其中D1中的值是一个搜索关键字。
- 寄存器D0中一个在1到65 535之间的正数，其中D0中的值决定了到底A0所指字节流中有多少个整数元素将被搜索。
- 子程序返回值存在于D2中，如果搜索关键字和被搜索字节流中的数字没有匹配，返回值就为0；否则为第一个匹配数的存储位置值。

注意一旦发生匹配搜索就不需再继续，假设你并不知道调用该子程序的主程序的其余 224

状态。

5. 某特定计算机系统的存储器布局情况为：ROM地址从0000到0x7FFF，RAM地址从0x8000到0xFFFF。下面的一段代码有一处错误，该错误是什么？

代码段：

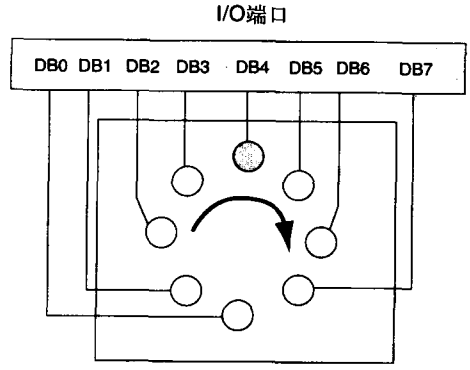
```
MOVE.W      $1000,D0
MOVE.W      D0,$9000
LEA.W       $2000,A1
MOVEA.W     A1,A2
MOVE.W      D0,(A2)
```

6. 编写一个简短的68K汇编语言程序，该程序将两个64位值相加并保存结果。说明如下：
   a. 操作数1：高32位存储在存储位置$1000
   b. 操作数1：低32位存储在存储位置$1004
   c. 操作数2：高32位存储在存储位置$1008
   d. 操作数2：低32位存储在存储位置$100C

   结果存储如下：
   a. 高32位在存储位置$1020
   b. 低32位在存储位置$1024

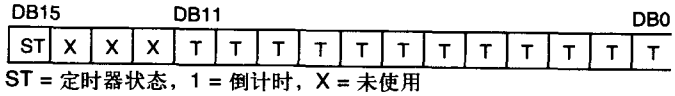
   任何高位加法产生的进位都在存储位置$101F。

7. 右图显示的是一个由8盏灯组成的环形阵列，这个阵列被连接到了一个8位带存储器映射和I/O端口的68K计算机系统。每盏灯由I/O端口中的一个相应位控制。将相应位设为1就会打开灯，设为0就会将灯关闭。



编写一个简短的68K汇编语言程序，该程序顺序打开每盏灯，保持两秒后又将它关闭，接着再打开下一盏灯。说明如下：

- 8位宽的并行I/O端口映射到存储地址$4000。
- 存储地址$8000处有一个16位宽的时延端口。DB0到DB11代表一个可以产生时延的倒计时定时器。向DB0-DB11写一个值将会使DB15由低变高，然后定时器就会从存储在DB0-DB11中的数开始倒计时到0。当定时器到达0时，DB15会再次变低且定时器停止计时。定时器的每次滴答代表1ms。因此，向定时器写$00A将会使定时器倒计时10ms。



225

- DB15是一个只读位，向该位写一个值不会改变它也不会产生任何问题。
- 定时器没有中断，你必须一直检查相应存储位置以知道定时器什么时候停止计数。

8. 编写一个简短的68K汇编语言子程序，该程序根据以下要求将一个ASCII码字符串发送到一个串口：
   - 串口被存储器映射到地址$4000和$4001的两个连续字节位置。发送和接收字符的实际端口在地址$4000，状态端口在地址$4001。

- 状态端口的数据位0（DB0）被指定用来指示传送缓冲器的状态。当串行设备准备好传送下一个字符时，传送缓冲器空（Transmit ter Buffer Empty）信号为真（TBE = 1）。当传送缓冲器为空时，下一个字符将被发送。向地址$4000写一个字符可能会启动串行数据传送并置TBE = 0。当TBE = 1时下一个字符就可以被发送。
- 在入口处，子程序应该保存可能使用到的任何寄存器的值，在出口时再将它们恢复。
- 待打印的字符串位置从寄存器A0传送到子程序中。
- 字符串以空字符$FF结束。
- 堆栈指针已经被初始化，没有必要建立一个堆栈指针。
- 假设这是一个没有中断发生的轮流检测的循环。你必须不断测试TBE的状态以便知道什么时候发送下一个字符。

9. 编写一个程序将两个指定位置之间的存储器都以字码$5555填充。这与指令集模拟器（ISS）中的块填充（BF）命令类似，但是你必须用68K汇编语言代码来完成。要填充的存储区域是$2000到$20FF之间并包括两端的范围。

10. 编写一个存储器测试程序，该程序能够以字（一次16位）来测试从$00001000到$0003FFFF之间的存储区域。它应该向上测试，包括$0003FFFF，但不超过它。存储器测试程序的工作如下所示：
    - 用字数据值填充被测试区域中的每一个存储位置。
    - 读回每一个存储位置并将它与你所写的数据值进行比较。如果它们不相同，那么就有坏存储位置。
    - 使用两个不同的测试码$FFFF和$AAAA开始测试存储器。
    - 当你用一种测试码测试完后，对它的位取反（1改成0且0改成1）并用新的测试码重复进行测试。因此，你将使用这些测试码重复测试4次。
    - 使用开始的测试向量$0001至少再重复测试一次。使用ROL.L指令在每次通过存储器测试时左移1位，直到你已经运行了16次存储器测试，每次你重复测试时都要把1左移一位。
    - 将你刚才使用的测试码取反，重复上面的移位测试。
    - 这里是任务中的一些细节：
      - 程序应该从存储地址$00000400处开始运行（用ORG）。
      - 程序测试从$00001000到$0003FFFF的存储区域，包括两端。
      - 堆栈指针应该定位在$000A0000。
      - 初始的测试码是：$FFFF、$AAAA和$0001。
      - 该测试将用测试码之一来填充相关的存储区域。接着，它把测试码读回并将读回的值和所写的值进行比较。如果你编写的程序在向存储器中写一个字后立即又将其读回，那么你就没有按照要求来编程。
      - 测试不断重复，所用的测试码分别是：两个开始测试码、他们的反码、移位码以及补码。
      - 如果检测到错误，则错误发生处的存储地址、所写数据以及读回数据都将被存储在存储变量中。
      - 如果发生了两次（含两次）以上的错误，那么程序应该存储总的错误数（发现的坏存储位置数），但只保存最近一次错误的地址信息和数据信息。
      - 应该允许多至65 535个坏存储位置的计数。

226

讨论

该程序包含了汇编语言程序设计的很多基础知识。如果你学习了该程序，你将看到该程序使用了一个子程序来实际检测存储器。想像一下，如果你用C语言编写，那会是怎样的呢？你如何将测试码传给函数？

- 别忘了初始化栈指针！
- 程序中有几个循环结构，它们是哪些？你将怎样定义被测试的存储区域？你怎么能知道你完成了所有的测试？你如何知道你是否写完了所有要写的存储位置？
- 确认你掌握了怎样使用伪操作代码EQU、ORG、CRE、DC.L、DC.B、DC.W以及DS.L、DS.W、DS.B、END。
- 掌握指令JRS、RTS、LEA和寻址模式(An)和(An)+。
- 该程序可以用不到50条的指令来完成，但是你要知道你现在想要做的是什么。Easy68K模拟器计算周期。程序若能在更少的时钟周期内完成，就算它需要更多的指令，也通常是更高效的程序。
- 分阶段进行编程。一旦你已经完成了程序流程图，就编写每一块的汇编代码并测试该代码。你怎样测试呢？你可以将它汇编。如果你能正确地汇编，那你就已经取得进步了。下一步，将其运行在模拟器上并验证它是不是正在做你想要它所做的事。
- 当你测试你的程序时，一个好的编程技巧是使用等于伪指令（EQUate）来将你测试的存储范围改为几个字，而不是整个空间，那样你就能很快地修改代码。因此，可以不测试从$00001000到$0003FFFF的范围而测试从$00001000到$0000100A的区间。

227

- 看看当一个存储位置上有坏数据时会发生什么。在程序用测试码填充完某个存储地址之后，使用模拟器改变该存储位置处的数据值。你的程序是否会发现该错误呢？它能处理这个错误吗？Easy68K模拟器有一个很好的能够从视图窗口访问的存储器接口。
- 该程序几乎只是由MOVE、Bcc和CMP（CMPA）这三种指令组成的。寻址模式可能是寄存器间接寻址，因为你们总是不断地向连续存储位置写并不断从连续存储位置读回。地址寄存器是保存开始地址、结束地址以及你当前在存储器中的地址的理想地方。为了指向下一个存储位置，就递增地址寄存器的内容。你可以显式地用加法完成，或者隐式地使用（An）+寻址模式。
- 思考当你使用ROL指令时你怎样结束测试。你可以建立一个计数器，然后计数32次，这种方法是可行的。但是，仔细看看ROL指令，当“1”超出最高有效位（MSB）时它跑到哪里去了呢？什么指令可以用来测试这个条件呢？

程序的大致结构应当如下：

a. 注释头块：描述程序做什么。
b. 系统的等于伪指令：定义变量。
c. ORG声明：程序从这里开始。
d. 主程序代码：除子程序之外的所有东西都在这里。
e. 指令STOP $#2700：正常结束程序并返回模拟器。
f. 子程序头块：所有的子程序都应有自己的头块。
g. 子程序标号：每个子程序在第一条指令处都应该有一个标号。否则你不能从当前位置跳到那里。
h. 子程序代码：这就是真正工作的代码。别忘了弄清楚哪些寄存器在工作以及参数是怎样

传递的。

i. RTS：每个子程序最终都必须返回。

j. 数据区域：用DC和DS伪代码定义的所有变量都存在这里。

k. END：程序的最后一行应该是END $400。这告诉汇编器程序在这里结束并装载到$400。

如果你仍然失败了，请复习一下本章中存储器测试程序的例子。最后，注意如果你将程序运行在多种Windows形式下你可能会发现奇怪的行为。如果运行在一小块存储区域程序可能运行得很快，但是如果运行在一块更大的存储区域程序就会死掉。这不是程序的错误。这是Windows在运行控制台应用程序时出现的问题。

Windows在控制台窗口中监视I/O活动。当它在窗口中没有看到任何输入或输出时，它将严格限制给予窗口中应用程序的CPU周期数。因此，一旦你的程序开始占用时钟运行时，Windows就会进一步压制它。

关于这个问题有几种方法，具体要取决于你的Windows版本。当程序在运行时你可以试图敲击ENTER键。它在窗口中不会做任何事，但是它会愚弄操作系统，使你的程序保持活动。

另一种方法是打开窗口中的属性菜单进行灵敏度设置，使得Windows不会将你的程序 228
关掉。这种方法在学校里的Win 98SE和Win 2000上可行。

第9章 高级汇编语言编程

学习目标

- 使用68K处理器体系结构的所有寻址模式和指令进行汇编语言编程；
- 描述汇编语言指令和寻址模式是如何支持高级语言的；
- 将存储器映像反汇编到指令集体系结构；
- 描述单板计算机系统的元件和功能。

9.1 引言

既然我们有了充足的68K编程原理的背景知识，让我们更密切地考察一下另外一些指令和寻址模式，以便更深入地探究这个问题。从一个汇编语言编程者的角度看，我们现在将要探究的寻址模式更加冷僻，但如果你要用C或C++作为开发语言来为68K系列处理器编程，那么这些寻址方式就极为重要了。

由于占压倒性的绝大多数的程序员都用C或C++写程序，所以对计算机设计者来说，具有支持高级语言结构的寻址模式就是一个重要的考虑事项。我们将要学习的寻址模式就能使我们实现数据结构和编写与位置无关的代码，或称*可重定位*（relocatable）代码。

因为没有对包含指令或数据的存储空间的*绝对引用*（absolute reference），所以可重定位代码可在处理器地址空间的任何地方运行。能装入地址空间任何地方的这个特性是重要的，因为操作系统必须能以这样一种方式来管理任务和存储器：一个任务（程序）可能一次在地址$A30000处开始运行而另一次在地址$100000处开始运行。另外，从占用的存储大小和速度方面考虑，与位置无关的程序效率更高，因为跳转和取数的目的地是由寄存器的内容决定的，而不是作为指令的一部分从存储器中得到的。

在开始学习高级寻址模式前，你可能想知道可重定位代码和绝对地址引用的思想。请看下面的代码片段：

229

```
start    LEA      $4000,A0
         MOVE.W   D0,(A0)
```

我们做了绝对地址引用了吗？绝对是（对不起）！然而，一旦我们有一个地址在寄存器中，即使它开始来源于绝对引用，我们也有能力来根据程序装入存储器的地点来修改这个地址。

9.2 高级寻址模式

模式5，带位移的寄存器间接寻址

将一个带符号的（正或负）16位位移与地址寄存器内容相加来形成有效地址。这样，有效地址（EA）=（An）+/− 16位位移。例如，如果 <A6> = $1000，则指令：

```
MOVE.L  $400(A6),D0
```

将取出位于存储地址$1400中的长字，并将其拷贝到数据寄存器D0中。这个寻址模式是很重要的，因为它被用来在C函数中定位局部变量。

这就是上述指令的寻址方法。在程序员参考手册中，该指令的另外一种表现形式为（d_{16},

An)。这两种形式都有可能使你相信上例中的$400是一个位移值。然而，大多数汇编程序不指望你自己计算位移值。汇编器料想你会插入一个标号或绝对存储器引用，它就为你计算出位移值。这样，数字$400不应该被解释成$400的位移，而是应该作为你希望从那里计算位移的存储位置。因此，如果你的汇编程序向你报错，如*位移过大*（displacement too large）或*越界错误*（out-of-range error），那么这指的可能是绝对地址或标号而不是偏移。

当你在C或C++语言中进行函数调用时，编译器会建立一个*堆栈帧*（stack frame），用地址寄存器之一作为堆栈指针。不同变量的地址用它们相对于指针的位移来辨别。这样，如果A6指向函数**foo( )**堆栈帧的开始处，则要取出的局部变量位于从指针开始的$400字节处。

模式6，带变址的寄存器间接寻址

地址寄存器的内容和变址寄存器（A0...A6或D0...D6）的内容相加，再加上一个8位的位移：

$$EA = (An) + (Xn) + d_8$$

如果 <A5> = $00001000且<D3> = $AAAA00C4，则指令

```
MOVE.W $40(A5,D3.W),D4
```

将取出存储器位置$00001104的字内容，并将数据拷贝到寄存器D4。为了解这个过程，在模拟器中试着运行一下这个代码片段：

例子

```
org     $400
lea     $00001000,A5
move.l  #$AAAA00C4,D3
move.w  #$AAAA,D1
move.w  D1,$40(A5,D3.W)
stop    #$2700
end     $400
```

起先，你可能想有效地址应该是$AAAA1104。然而，这个指令的变址寄存器D3用的只是字的值（而不是长字的值）来计算有效地址。 230

这个寻址模式可能看起来很陌生，但假设你已经用C建立过一个复合数据类型（结构）的数组，每个数据类型距结构的开始处有固定的偏移量。为了访问特定结构的特定数据元素，你必须用D3索引到数组中，然后再用固定偏移量$40寻找特定的元素。

模式7，子类2：带位移的程序计数器

将一个带符号的16位位移加入到程序计数器当前的内容中。

$$EA = (PC) + d_{16}$$

这是一个称为PC*相对*（PC relative）的一般寻址模式类的一个例子。请不要将它的意思误解为是与PC相关的一些东西，如Palm Pilot。PC相对寻址是产生可重定位代码所需要的最重要的寻址模式。在PC相对寻址中，操作数的有效地址是通过将字的*带符号扩展*（sign extended）值加入到PC的当前值中来计算的。然后所产生的地址被放入到地址线上，并从外部存储器中取出数据。

例如，假设<PC> = $D7584420。

$7AFE = 0111 1010 1111 1110。对最高有效位（加下划线的）进行带符号扩展至32位，就得到0000 0000 0000 0000 0111 1010 1111 1110。

$$EA = \$D7584420 + \$00007AFE = \$D758BF1E$$

当指令**MOVE.W $100(PC), D4**运行时，如果在程序中该点处<PC> = $00000400，则存储器位置$500处的字内容就从存储器中取出并拷贝到D4。然而，若该代码片段被重新放置到

存储器中的另一个地方并重新运行，则只要被取的数据位于指令$100字节的地方就仍能正确取出该数据。

其原因就是程序计数器的当前值将被用于计算有效地址。PC总是指向要从存储器中取出的下一条指令，不管该程序代码处于存储器的哪个地方。只要计算出来的有效地址位于一个距PC当前值固定的距离，就万事大吉了。

跳转指令（JUMP）和跳转到子程序（JUMP TO SUBROUTINE）指令也是这种情况。到目前为止，我们总是认为JMP指令和JSR指令的目的地址是绝对的。然而，考虑下面两个代码
231 例子：

例子：情况1

```
JSR    foo    * 汇编器为foo生成绝对地址
```

例子：情况2

```
JSR    foo(PC)      * 汇编器生成相对地址

foo  {这里有更多指令}
     RTS          * 从子程序返回
```

在情况1中，汇编器为标号**foo**计算了绝对地址。当指令执行时，下一条指令的地址被放入到堆栈，而**foo**的绝对地址被放入到PC。从存储器中取出的下一条指令就是位于地址**foo**处的指令。

在情况2中，汇编器计算从PC当前值到**foo**的位移（距离）。当指令执行时，下一条指令的地址（PC的当前值）被放入堆栈，位移被加到PC的当前值，并将相加的和返回到PC。下一条指令就从地址**foo**处取出。

然而，只有情况2才允许将程序（主程序代码加上子程序）移到存储器中的其他位置而无须重调整所有地址。

模式7，子类3：带变址的程序计数器

程序计数器的内容加上变址寄存器（A0...A6或D0...D6）的内容，再加上8位位移。这样，EA = (PC) + (Xn) + d_8。

例子

```
MOVE.W $40(PC,D3.L),D6
```

这个寻址模式与带位移的寄存器间接寻址完全相同，除了采用PC而不是地址寄存器作为基寄存器来进行地址计算。

总之，对于基于表、字符串、数组、链表等数据类型的算法，变址寻址（用另一个寄存器提供可变的位移）是一种强有力的实现方法。位移值提供了从程序计数器当前值到表（基地址）开始处的恒定的偏移。变址寄存器提供了到表中的可变指针。通过采用PC相对寻址模式，整个表自动地随程序移动。这在编译器操作中非常有用，在那里需要用一个在结构中恒定的偏移（位移）来对变量寻址。

9.3 68000指令

在这里的每个种类的指令都有若干个变种。例如，ADD指令系列包括：

- ADD：将一个有效地址加到一个数据寄存器或将一个数据寄存器加到一个有效地址。
- ADDA：将一个有效地址加到一个地址寄存器。
- ADDI：将一个立即数值加到一个有效地址。
- ADDQ：将一个从1到8之间的立即数加到一个有效地址。

• ADDX：将一个数据寄存器值加到一个数据寄存器，包括X位的值。

如你所知，有效地址也许是我们迄今所学过的某些或全部寻址模式。显然，要完成你要做的工作，可有很多种方式来构造指令。结论就是学习用汇编语言编程就像用字典来学习说话，先学习一些简单的词，然后扩大你的词汇量，以提高你用词的丰富性和效率。大多数算法都可用多种方式编写，但没有丰富的编程经验就很难编写出最有效的算法。首先要使你的算法能够运行，然后再努力调整提高它！ 232

9.4 移动指令

MOVE（移动数据）：将数据从源拷贝到目的地。你如今应该知道这条指令了。

MOVEA（移动地址）：将数据拷贝到地址寄存器。目的地总是地址寄存器，数据大小必须是字或长字。请看下面给出的**MOVE**和**MOVEA**指令的格式：

MOVE指令：

15	14	13 12	11 9	8 6	5 3	2 0
0	0	大小	目的寄存器	目的模式	源模式	源寄存器

MOVEA指令：

15	14	13 12	11 9	8	6	5 3	2 0
0	0	大小	目的寄存器	0	0 1	源模式	源寄存器

它们看起来不相同，直至你认识到位8、7、6只是定义了一种模式1寻址模式，就是寄存器直接寻址模式，所以我们仍没有真正理由要单独设一条指令。其真正理由是由于操作大小所施加的限制使得我们需要有不同的表示。由于MOVEA指令只能将字或长字值移动到地址寄存器，所以我们要尽可能地减少粗心大意地写出非法操作代码的情况的可能性。

MOVEM（移动多个寄存器）：MOVEM指令只能用于将寄存器移动到存储器或相反。它最常与带后递增和前递减的寻址模式一起使用。前递减用于将寄存器传送到存储器中的一个堆栈结构，而后递增用于将数据从存储器堆栈传回到寄存器。对于系统堆栈，堆栈指针可能是SP寄存器；对于用户堆栈帧，堆栈指针则是其他地址寄存器中的一个。寄存器列表的顺序并不重要，因为68000总是以A7到A0，然后D7到D0的顺序写存储器。它总是以D0到D7，然后A0到A7的顺序从存储器读。

9.5 逻辑指令

AND

AND指令执行按位逻辑与操作。例如，如果在指令**AND.W D0, D1**执行之前，<D0> = $3795AC5F且<D1> = $B6D34B9D，则指令执行后，<D0> = $3795AC5F，<D1> = $B6D3081D。注意在操作中只使用了各寄存器的低16位。为了弄明白为什么是这个结果，让我们用二进制来做按位与操作： 233

例子

```
$3795AC5F = 0011 0111 1001 0101 1010 1100 0101 1111
                          与
$B6D34B9D = 1011 0110 1101 0011 0100 1011 1001 1101
                          等于
$3691081D = 0011 0110 1001 0001 0000 1000 0001 1101
```

要记住的一个简单窍门是：

- 任意一个十六进制数字和F做与操作后得到的还是这个数字
- 任意一个十六进制数字和0做与操作后得到的还是0

ANDI

ANDI：和立即数做与操作。例如，**ANDI.B #$5A, D7**

9.6 其他逻辑指令

- **OR**：执行按位逻辑或
- **ORI**：和立即数做或操作
- **EOR**：执行按位逻辑异或
- **EORI**：和立即数做异或操作
- **NOT**：对操作数做按位逻辑反操作

NOT指令只需要单个操作数。如果<D3> = $FF4567FF，则指令**NOT.B D3**将使D3中的数据变为<D3> = $FF456700。注意你不能用NOT立即数这种指令，因为**NOTI.B#$67**没有意义。

移位指令和循环移位指令

这组指令根据移动多少个位置和有效地址是寄存器还是存储器而有不同的规则。移位指令和循环移位指令能对位和数据字的各部分进行操作，以便对这些数据更好地进行算术和逻辑操作，因此是非常重要的。

例如，假设你从一个调制解调器中读了4个ASCII字符，这些字符是$31、$41、$30和$30。而且，假设字符是4位十六进制数字。如果你参考ASCII值的表，就会发现它们表示数字$1A00，但是如何从这4个ASCII字节得到数字$1A00呢？我们需要一个转换算法。

我们很快就会回到这个问题并学习一个算法来解决这个问题。现在，让我们看一下移位和循环移位是什么意思。见图9-1。

ASL指令将字节、字、长字或更多位置的每一位向左移动。每次移位，0就被插入到最低有效位的位置DB0。占据最高有效位位置的位DB7、DB15或DB31就移动到条件码寄存器（CCR）的进位位和扩展位，在CCR中的位就被抛弃了。这样，执行指令**ASL.B #3, D0**且执行前<D0> = $AB005501，则

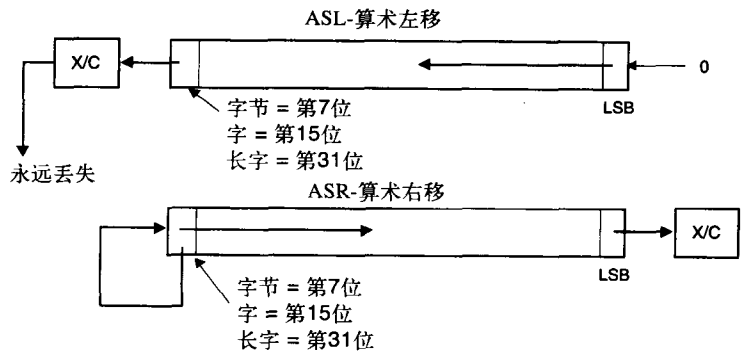


图9-1 算术左移和算术右移指令的操作

234 指令完成后，D0被移位三次的结果就是<D0> = $AB005508。通过每次将数字向左移位，**ASL**还有将数据乘以2的效果。

其他的移位和循环移位指令与**ASL**和**ASR**的操作方式类似。图9-2总结了这些指令的行为。比较图9-1和图9-2，我们看到一些指令似乎展示出相同的行为，比如**ASL**和**LSL**指令，但**ASR**和**LSR**稍有不同。

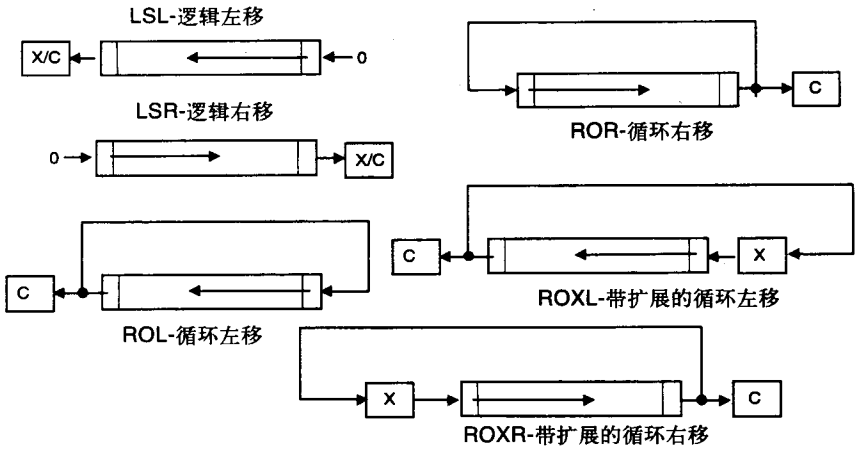


图9-2 逻辑左/右移和循环左/右移指令

让我们总结一下使用移位和循环移位指令的精要点：

- 这个组的指令可以是字节、字或长字的操作。
- 操作数只能是数据寄存器D0-D7的内容，或者是存储器位置的内容。
- 当涉及到数据寄存器时，指令必须提供移位的次数。
- 如果移位的次数少于8，就采用立即数的形式。
- 如果移位的次数多于8，则该次数必须放在另一个数据寄存器中。
- 当涉及到存储器位置时，一次只能移动一位，且只允许字操作数。

最后，我们返回前面讨论过的例子程序，以此来总结我们的讨论。对该例子，我们假设有4个ASCII数字，位于称为“ascii-val”的存储缓冲器中。这个程序将ASCII字符转换成4位十六进制数，并被存回到一个称为“hex_val”的存储器位置。请特别注意一下采用ASL指令 235
移动数位的方式。

这个程序称为Get Value，它能将存储器中的4个ASCII数字转换成4位长十六进制数。

```
*************************************************************************
* Get_value
* 将4个ASCII值转换成4位十六进制数字
* 输入参数：无
* 假设：缓冲器、ascii_val包含4个有效的ASCII字符，其范围为0...9、A...F或a...f
*************************************************************************

* 系统的等于伪指令
mask        EQU         $00FF           * 分离出字节值
stack       EQU         $B000           * 堆栈指针的位置

* 程序从这里开始

            ORG         $0400           * 程序从这里运行
start       LEA         stack,SP        * 建立堆栈指针

            CLR.W       D7              * 我们将用到这个寄存器
            LEA         ascii_val,A1    * A1指向存储缓冲器

            MOVE.B      (A1)+,D0        * 取第一个字节
            ANDI.W      #mask,D0        * 分离出这个字节
            jsr         strip_ascii     * 去掉ASCII码
```

```
                ASL.W       #8,D0           * 左移8位
                ASL.W       #4,D0           * 左移4位
                OR.W        D0,D7           * 将这些位装到D7
                MOVE.B      (A1)+,D0        * 取下一个字节
                ANDI.W      #mask,D0        * 分离出这个字节
                jsr         strip_ascii     * 去掉ASCII码
                ASL.W       #8,D0           * 左移8位
                OR.W        D0,D7           * 加下一个十六进制数字
                MOVE.B      (A1)+,D0        * 取下一个字节
                ANDI.W      #mask,D0        * 分离出这个字节
                jsr         strip_ascii     * 去掉ASCII码
                ASL.W       #4,D0           * 左移4位
                OR.W        D0,D7           * 加下一个十六进制数字
                MOVE.B      (A1)+,D0        * 取下一个字节，指向con_val
                ANDI.W      #mask,D0        * 分离出这个字节
236             jsr         strip_ascii     * 去掉ASCII码
                OR.W        D0,D7           * 加最后一个十六进制数字
                MOVE.W      D7,(A1)+        * 存转换结果
                STOP        #$2700          * 返回到模拟器

*************************************************************************
* 子程序：strip_ascii
* 从数字0~9  a~f或A~F中除去ASCII码
* 输入参数：<D0> = ASCII码
*
* 返回参数：D0.B = 数字0...F，返回00...0F
* 使用的内部寄存器：D0
* 假设：D0包含$30~$39、$41~46、$61~66
*
*************************************************************************

strip_ascii  CMP.B    #$39,D0           * 它在0~9的范围内吗?
             BLE      sub30             * 它是一个数字
             CMP.B    #$46,D0           * 它是A...F吗?
             BLE      sub37             * 它是A...F
             SUB.B    #$57,D0           * 它是a...f
             BRA      ret_sa            * 返回
sub37        SUB.B    #$37,D0           * 去掉37
             BRA      ret_sa            * 返回
sub30        SUB.B    #$30,D0           * 去掉30
ret_sa       RTS                        * 返回

* 数据

ascii_val    DC.B     $31,$41,$30,$30   * 测试值$1A00
con_val      DS.W     1                 * 将它存到这里
             END      $400
```

算术指令

ADD（加二进制数）

将源操作数和目的操作数相加，并将结果放入目的操作数。ADD指令的源操作数或目的操作数中有一个必须是数据寄存器。换句话说，你不能直接将两个存于存储器中的数据加在一起，操作数之一必须在数据寄存器中。事实上，只有**MOVE**指令被设计成其两个有效地址都可以是存储器地址。

让我们做一个例子。假设在执行**ADD**指令之前，<D2> = $12345678, <D3> = $5F02C332

```
ADD.B   D2,D3
```

加法指令执行后，<D2> = $12345678, <D3> = $5F02C3AA

对条件码有什么影响?

- N=1：负标志：第7位是1，所以结果假定为负。
- C=0：进位标志：从第7位没有产生进位。
- X=0：扩展标志：用于某些操作，所受影响通常与进位标志类似。
- Z=0：零标志：非零结果。
- V=1：溢出标志：这是需要慎重考虑的！目的寄存器D3的第7位变化了，所以可能遇到了溢出条件。

237

ADDA（加地址）

将数据加到一个地址寄存器。只允许字操作和长字操作，条件码不受影响。

ADDI（加立即数）

将数字加到数据寄存器或存储器。允许字节操作、字操作或长字操作。考虑一下这个问题，如果<D2> = $25C30F7，执行下面这两条指令的结果是什么？注意它们的不同仅在于操作的大小。为核对你的答案，写一个测试程序并在模拟器中执行。

代码实例

```
ADDI.B    #$10,D2    * <D2> = ??, C = ??
ADDI.W    #$10,D2    * <D2> = ??, C = ??
```

CLR（对一个操作数清零）

CLR指令可用于字节、字和长字。除了X的所有条件码都受影响。

CMP（将数据与一个数据寄存器进行比较）

CMP指令根据结果设置条件码。这条指令是从目的操作数中减去源操作数，但不将结果放回到目的操作数。因此，哪个操作数也没有改变，只有条件码标志受到了影响。

9.7 68000指令总结

我们就不再继续艰难地逐条学习68K指令集中的各条指令了，让我们按功能分组对指令进行简要地总结。

数据传输指令总结

EXG	寄存器交换
LEA	装入有效地址
LINK	链接和分配
MOVE	移动数据
MOVEA	移动地址
MOVEM	移动多个寄存器
MOVEP	移动外设数据
MOVEQ	快速移动
PEA	有效地址压栈
SWAP	半寄存器交换
UNLK	解除链接

某些数据传输指令如**LINK**和**UNLK**看起来可能会很陌生，这些是特殊用途的指令，仅用于支持高级语言。我们将在本章的后面更详细地讨论这些指令。

238

算术指令总结

ADD	加二进制数
ADDA	加地址
ADDI	加立即数
ADDQ	快速加
CLR	操作数清零
CMP	比较
CMPI	立即数比较
CPM	存储器比较
DIVS	带符号数除
DIVU	无符号数除
MULS	带符号数乘
NEG	取反
NEGX	带X取反
SUB	减二进制数
SUBA	减地址
SUBI	减立即数
SUBQ	快速减
SUBX	带X减
TAS	测试和置位
TST	测试
EXT	扩展符号

程序控制指令总结

Bcc	根据条件码（cc）标志状态转移
DBcc	递减并根据cc转移
Scc	设置cc
BRA	总是转移
BSR	转移到子程序
JMP	无条件跳转
JSR	跳转到子程序
RTR	返回和恢复
RTS	从子程序返回

逻辑和移位指令总结

AND	逻辑与
ANDI	对立即数与
OR	逻辑或
ORI	对立即数或
EOR	异或
EORI	对立即数异或
NOT	逻辑补
ASL	算术左移
ASR	算术右移
LSL	逻辑左移
LSR	逻辑右移
ROL	循环左移
ROR	循环右移
ROXL	带扩展的循环左移
ROXR	带扩展的循环右移

特权指令总结

ANDI SR	将立即数加到状态寄存器
EORI SR	将立即数异或到状态寄存器
MOVE SR	移入/移出状态寄存器
MOVE USP	移入/移出USP
RESET	复位处理器
RTE	从异常返回
STOP	停止处理器
CHK	检验寄存器
ILLEGAL	强加一个非法指令异常
TRAP	陷阱调用
TRAPV	溢出时陷阱调用
ANDI CCR	将立即数与到条件码寄存器
ORI CCR	将立即数或到条件码寄存器
EORI CCR	将立即数异或到CCR
MOVE CCR	移入/移出CCR
NOP	无操作——不做任何事

位操作指令总结

BCHG	位改变
BCLR	位清零
BSET	置位
BTST	位测试

二进制编码的十进制数（BCD）指令总结

ABCD	加BCD数
NBCD	取反BCD数
SBCD	减BCD数

BCD指令需要解释一下。它们是计算机早期的遗产，当时很多仪器仍用数字逻辑电路设计，这些电路计算出结果并将其显示为十进制数，即使计算时用的是二进制数。一个BCD数与在0到9范围内的4位十六进制数相同。当计数从9到A时，BCD数将产生一个进位，并且数字返回到0。换句话说，就是用基数10而不是基数16来计数。这些指令现今已经很少使用了。

如你所见，我们有相当多种类的68K指令可用来解决广泛的真实世界的编程问题。68K指令集体系结构已经历了时间的考验，而且正在被设计到新的产品中。很多设计是用于管理操作系统之下的应用的，另一些设计是为了支持高级语言的，还有一些是用于处理程序中断的。在对汇编语言编程的介绍中，我们对此已经讨论了若干次。重要的是，首先用你知晓的指令和寻址模式轻松自如地解决算法，然后再向更有效率（也许更困难）的指令和寻址模式过渡。事实上，一贯地使用相对少量的指令绝不表示专业程度低。在下一课你将会看到：只用全部指令的一部分来解决大多数问题的事实导致了一种现代计算机体系结构的产生，就是*精简指令集计算机*，即RISC。

9.8 用TRAP #15指令模拟I/O

包含这部分内容的原因是我们需要能用68K模拟器写出有意义的程序。迄今我们看到和写出的程序都是独立的，没有任何与用户交互的功能。这样的计算机程序对学习体系结构来说

是很好的，但在真实世界中它们的用处极其有限。为了使一个计算机有用，我们就必须能与它交互。在这一小节中我们将向你介绍模拟器的一个非常有用的特征，就是I/O程序的**TRAP #15**系列。

在前面章节中你看到过一个称为**TRAP**的指令。**TRAP**指令就像一个软件产生的中断，当处理器执行一个**TRAP**指令时，它将从向量表中的固定位置自动获得TRAP地址，将返回地址放到堆栈上，并在新的位置开始进行处理。**TRAP #15**只不过是异常向量表中**TRAP**位置列表中的一个位置，异常向量表存储在从$080到$0BC的区域中，由处理器维护。

如果你进入68K模拟器的Teesside版本，并键入“HE”来响应提示符，你将进入到帮助程序中。**TRAP #15**指令就是模拟器设计者所开发的便于用户和程序之间进行I/O操作的一种方式。你可以通过学习与E68K程序一起提供的帮助程序来获得**TRAP #15**指令的一些知识。或者，你也可以在Clements[1]的书中读到关于**TRAP #15**的功能。

记住，**TRAP #15**指令是模拟器的产物，它是被设计用来使模拟器和用户之间能发生I/O
240 操作的。如果你真要写I/O程序，你可能要做一些不同的事情，但很多事情是一样的。

与**TRAP #15**指令相关联的是各种任务，每个任务都有编号。与每个任务关联的是一个*应用程序员接口*（application programmer's interface），即API，用来解释该任务如何完成其工作。例如，task #0将一个字符串显示到显示器上并加入一个换行符，使得光标前进到下一行的开始处。为了使用task #0，你必须建立下面的寄存器（这就是API）：

- D0以字节形式装载任务编号。
- A1装载字符串开始处的存储地址。
- D1装载要打印字符串的长度。

一旦建立了这三个寄存器，你在程序中将**TRAP #15**作为指令调用，你的消息就会出现在模拟器的显示屏上。这样，TRAP指令就成了68K模拟器和PC操作系统之间的接口。下面的这个样例程序示意出了它是如何工作的。为输出字符串“Hello world!”，你可以采用如下的代码片段：

“Hello World!”例子

```
*****************************************************
*
* 将字符串显示到显示器的测试程序
*
*****************************************************
            OPT         CRE
task0       EQU         00

            ORG         $400

start       MOVE.B      #task0,D0       * 将任务编号装入到D0
            LEA         string,A1       * 取字符串地址
            MOVE.W      str_len,D1      * 字符串长度在D1中
            TRAP        #15             * 开始做
            STOP        #$2700          * 回到模拟器

* 数据区

string      DC.B        'Hello world'   * 消息存储在这里
str_len     DC.W        str_len-string  * 取字符串长度
            END         $400
```

看这一行：

```
str_len   DC.W   str_len-string   * 取字符串长度
```

将会发生什么情况？我们正在让汇编器为我们做一些工作，汇编器通过将字符串后第一个地址减去字符串起始地址（标记为“string”）来计算字符串的长度（标记为“str_len”）。在汇编语言中，用这种方法计算程序中两个位置（存储器位置）之间的距离也是一种相当常用的技术。 241
当你在C++中减指针时，你做的也是同一件事情。

除了没有“换行”符，task #1和task #0几乎是等同的。为用户提供提示符来输入信息是很方便的，所以，你会使用task #1而不是task #0来给出提示。

还有，你也许想从用户那里得到信息。比如他们想在哪里运行存储器测试，要用什么测试码。你可能还要问他们是否要用不同的测试码再进行测试。

task #2是另一个非常有用的函数。在执行这条指令时，模拟器会等待你从终端输入一个以ENTER键结束的字符串。模拟器会将你输入的字符串放入A1所指的一个存储缓冲器中。

TRAP #15有很多功能程序。理解如何使用它们的最好途径就是首先进行阅读，然后写几个小的测试程序来检验你学到的知识。

9.9 编译器和汇编器

到现在为止，我们一直与C和C++这样的高级语言保持距离。我们确实曾短暂地离开主题看了一下如何用汇编语言来模仿C语言的一些典型的循环结构，但在大部分篇幅中，我们还是分开介绍的。

然而，我们不应该觉得奇怪的是，即使我们用C++写程序，我们也可以用汇编语言调试程序。而且，还可以从计算机体系结构的角度来理解为什么会存在某些指令和寻址方式。完整地学习编译器如何工作和如何利用计算机的指令集体系结构超出了本书的范围，但简要地看一看编译器如何工作并没有害处，所以，我们将要了解一下C语言的交叉编译器是如何将C的源文件转换成汇编语言的。

考虑下面这个简单的C程序：

```
int funct(int,int *) ;
void main()
{
   int i = 0, aVar = 5555, j ;
   i++ ;
   j = funct(i, &aVar );
   j++ ;
}

int funct( int var, int * aPtr )
{
   return var + *aPtr ;
}
```

这里我们声明了i、j、aVar这3个变量，并做了一些简单的操作。要注意的是，我们向函数*funct*传送了两个变量。一个变量i是通过值传送的，另一个变量aVar是通过引用传送的。该程序将用68000交叉编译器进行编译，该交叉编译器是由HP公司在几年前开发的，而且是与其嵌入式软件开发工具一起使用的。编译器生成汇编语言源文件，然后由68000汇编器进行汇编，该汇编器也是由HP公司开发的。 242

下面就是编译器为该程序生成的68000汇编语言代码。实际的汇编语言指令或伪指令用灰色字体显示。我还加入了灰色框来解释编译器为什么做和做了些什么。这个汇编语言源文件接着就将被68000汇编器进行汇编并产生目标文件。现代C或C++编译器通常会跳过汇编语言源文件的生成过程而直接形成目标代码。但是，我们的意图是用这种“老式的”编译器来确

切地看看所发生的事情。

```
CHIP    68000 NAME    test3
```

CHIP 68000告诉汇编器只使用68000规约
NAME 告诉汇编器所要生成的可重定位模块的名字

```
* 汇编器的可选项:
OPT     BRW,FRL,NOI,NOW
```

OPT BRW 所有转移都应该用16位的位移
OPT FRL 所有到绝对地址的向前引用都应该采用32位模式
OPT NOI 不要列出那些由于条件变量而没有被汇编的指令
OPT NOW 在汇编过程中不要显示警告信息

```
*
* 为调用运行时的程序库而做的宏定义
* 每次调用的字节=6
*
```

CALL：一个宏在每次代替一条指令被使用时，以"routine"作为伪输入参数，产生其所定义的一序列指令。宏定义是将几条汇编语言指令组合成一条新指令的简便方式。在这个例子中，它被用于访问一个位于库模块中的运行时子程序，该库模块是在程序链接到一起时被引入的。

XREF是伪操作代码，它告诉汇编器，程序是在另一个文件中定义的

```
CALL  MACRO   routine
      XREF  routine
      JSR   (routine,PC)
      ENDM
```

SECT：是一个伪操作码，指出这是一个名为"prog"的可重定位代码块。C指出它是一个代码段（指令）而且是第2部分。"P"是EP内部类型指示。

243

编译器都是在函数调用前插入NOP指令以帮助调试。

```
SECT    prog,2,C,P
        NOP
        NOP
        NOP
```

显视数布的C代码和汇编代码。编译器将在汇编代码中插入C源指令作为注释。我加入了另外两条指令使之能在模拟环境中运行。

```
        ORG $400
        LXA $1000,SP
*     1          int funct(int,int *) ;
*     2          void main()
*     3          {
```

XDEF：是一个伪操作，告诉汇编器这个名称为"_main"的函数在链接的时候，其他模块应该可以得到它。这样，完整的程序就知道从哪里开始。

```
        XDEF    main
```

LINK：这个指令确立了堆栈域。寄存器A6是这个函数（_main）堆栈的指针。这个指令做了如下几件事情：
1. 将A6的当前值压入到堆栈。
2. 堆栈指针的当前值SP(A7)被传输到A6。
3. 为了在堆栈上保存4个字节，指带符号扩展的16位位移-4加入到堆栈指针。这样，A6现在就是一个4字节堆栈的易知堆栈指针。系统堆栈指针位于它的下方。这个操作保存了函数main()用于存储其局部变量所需要的空间。

```
_main
        LINK    A6,#-4
                MOVE.L  D3,-(A7)        * 保存D3的当前值
                MOVE.L  D2,-(A7)        * 保存D2的当前值
                MOVEQ   #$FF,D1         * 不完全确信为什么
```

```
        MOVE.L   D1,(-4,A6)
 * 这就将FF放入到了_main堆栈帧的底部
        MOVEQ    #$FF,D2          * 谁知道?
        MOVE.L   D2,D3            * 同上
```

244

寄存器"D2"就是寄存器变量"S_i"。编译器有足够能力认识到，它可将变量保存到寄存器而不是存储器，以更高效地运行程序。在下面它将D2初始化为了0。

```
        MOVEQ    #0,D2
```

SET是伪操作代码，它给一个数赋值，并可重赋值。一个变量若由等于伪操作（EQU）命名，就不能重赋值新名字或值。

```
S_aVar  SET      -4
```

值5555被放入到堆栈栈中，寄存器"D3"将被赋值成寄存器变量"S_i"。ADDQ指令实现了i++。

```
        MOVE.L   #5555,(S_aVar+0,A6)

*    4              int i = 0, aVar = 5555, j ;
*    5              i++ ;

        ADDQ.L   #1,D2

*    6              j = funct(i, &aVar );
```

aVar的地址将被放入到A0，然后A0所指向的地址内容将被压入到堆栈。其中后一块代码称为前导。哪里发生了函数调用（即我们所知道的JSR指令），就生成像这样的代码。这里：

* A0保存aVar的地址
* aVar的地址被压入堆栈
* i被压入堆栈
* 用PC相对寻址实现JSR指令

```
        LEA      (S_aVar+0,A6),A0
        PEA      (A0)
        MOVEA.L  D2,A0
        PEA      (A0)
        JSR      (_funct+0,PC)

        ADDQ.L   #8,SP            * 函数越界
        MOVE.L   D0,D3            * D0是JSR的返回寄存器

*    7              j++ ;
  ADDQ.L   #1,D3            * 这是j++
*    8            }
```

结束括号"}"导致生成了下面的函数退出代码。

245

```
functionExit1  NOP
               MOVE.L   (A7)+,D2      * 恢复D2
               MOVE.L   (A7)+,D3      * 恢复D3
               UNLK     A6        * 释放堆栈帧

returnLabel1      RTS
               NOP
               NOP
               NOP
*    9           int funct( int var, int * aPtr )
*   10           {
                 XDEF     _funct
```

这是"_funct"子程序的入口点。注意编译器是通过在函数名前加下划线字符"_"的办法来产生子程序的汇编语言标号的。还要注意的是当堆栈帧被分配后，函数不需要任何额外的存储空间。

```
funct
        LINK    A6,#-0
S_var   SET     8
S_aPtr  SET     12
*   11              return var + *aPtr ;
```

接下来的这个代码块相当有趣。在指针被取消引用并读到aVar值之前，编译器检查aVar的地址是否是一个空指针。如果是，就调用一个运行时错误处理程序。这就是生成这代码的意图。该指令所做事情如下：

- 从堆栈中得到aVar的值，并将其放入到A0中。
- 将A0移动到D1，使得如果<A0> = 0，就会设置Z标志。
- 如果A0 = 0，则有一个空指针。
- 如果指针是坏的，就做BSR调转，跳转到库程序"ptrFault"。

```
        MOVE.L  (S_aPtr+0,A6),A0
        MOVE.L  A0,D1
        BEQ.S   L0_APtrChk
        MOVEQ   #$FF,D0
        CMP.L   D0,D1
        BNE.S   L0_BPtrChk
L0_APtrChk
        CALL    ptrFault
```

246

下一条指令的加入是为了使程序可运行于68K模拟环境。

```
 ptrFault   RTS
L0_PtrInfo
        DC.L    11
        DC.B    'test3.c'
        DC.B    0
```

ALIGN，是一个伪操作，用于对齐指令在字边界上。后三个指令实际上实现了C指令：

```
                                return var + *aPtr
```

- 变量aVar移动到寄存器D0中。
- 将S加入到D0，并将结果放回到D0中。

```
        ALIGN   2
L0_BPtrChk
        MOVE.L  (S_var+0,A6),D0
        ADD.L   (A0),D0
        BRA     functionExit2
12          }
```

"}"生成函数退出代码

```
functionExit2
        NOP                 * 这是函数调用的返回
        UNLK    A6          * 释放堆栈帧returnLabel2
        RTS                 * 这是程序的返回

        END $400
```

上面代码的运行过程就是编译器如何工作的粗略一瞥。从汇编语言的角度来看，一些指令序列没有任何意义。然而，我们必须记住这样的事实，就是编译器只是一个程序，它必须能处理C语言的所有可能情况。这意味着一旦为进行数据处理而确立了内务管理规则，这些规则就永远要遵守，即使它们在特定情况下是不必要的和冗余的。所以你就会明白为什么手工编写的汇编代码常常能使性能提高。

现在我们就可以问问关于$65 536的问题了。它能工作吗？上面的汇编代码正确地运行了原始的C代码了吗？让我们看一下。我们将在模拟器中实际地运行这个代码。我们还将监视堆

栈的行为，因为那里是实际操作的地方。模拟器代码列在左边，堆栈状态列在右边。我们用 247
深灰色显示堆栈帧指针的当前值A6，用浅灰色显示堆栈指针SP。图示如下：

```
A6  SP
PC=000400 SR=2000 SS=00A00000 US=00000000 X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0 1000 00 00 00 00
A4=00000000 A5=00000000 A6=00000000 A7=00A00000 Z=0 0FFC 00 00 00 00
D0=00000000 D1=00000000 D2=00000000 D3=00000000 V=0 0FF8 00 00 00 00
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0 0FF4 00 00 00 00
---------->LEA.L $1000,SP

PC=000404 SR=2000 SS=00001000 US=00000000 X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00001000 Z=0
D0=00000000 D1=00000000 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->LINK A6,#-4

PC=000408 SR=2000 SS=00000FF8 US=00000000 X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0 1000 00 00 00 00
A4=00000000 A5=00000000 A6=00000FFC A7=00000FF8 Z=0 0FFC 00 00 00 00
D0=00000000 D1=00000000 D2=00000000 D3=00000000 V=0 0FF8 00 00 00 00
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0 0FF4 00 00 00 00
---------->MOVE.L D3,-(SP)

PC=00040A SR=2004 SS=00000FF4 US=00000000 X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0 1000 00 00 00 00
A4=00000000 A5=00000000 A6=00000FFC A7=00000FF4 Z=1 0FFC 00 00 00 00
D0=00000000 D1=00000000 D2=00000000 D3=00000000 V=0 0FF8 00 00 00 00
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0 0FF4 00 00 00 00
---------->MOVE.L D2,-(SP)
```

以上的代码部分显示出堆栈指针被置为地址$1000，并用LINK指令生成了局部堆栈帧。下面就是发生的情况：

- 存储器从$0FFD到$1000的位置存放寄存器A6的当前值。
- 4个字节被保存在堆栈上，A6变成这些字节的堆栈指针，A6被置为$0FFC。 248
- 系统堆栈指针A7重置为$0FF8，在局部堆栈之下。

接下来，将D2和D3压入堆栈，因为它们将要被用到。

```
PC=00040C SR=2004 SS=00000FF0 US=00000000        X=0 1000 00 00 00 00
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0 0FFC 00 00 00 00
A4=00000000 A5=00000000 A6=00000FFC A7=00000FF0 Z=1 0FF8 00 00 00 00
D0=00000000 D1=00000000 D2=00000000 D3=00000000 V=0 0FF4 00 00 00 00
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0 0FF0 00 00 00 00
---------->MOVEQ #-1,D1

PC=00040E SR=2008 SS=00000FF0 US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=1
A4=00000000 A5=00000000 A6=00000FFC A7=00000FF0 Z=0
D0=00000000 D1=FFFFFFFF D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->MOVE.L D1,-4(A6)

C=000412 SR=2008 SS=00000FF0 US=00000000         X=0 1000 00 00 00 00
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=1 0FFC 00 00 00 00
A4=00000000 A5=00000000 A6=00000FFC A7=00000FF0 Z=0 0FF8 FF FF FF FF
D0=00000000 D1=FFFFFFFF D2=00000000 D3=00000000 V=0 0FF4 00 00 00 00
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0 0FF0 00 00 00 00
---------->MOVEQ #-1,D2

PC=000414 SR=2008 SS=00000FF0 US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=1
A4=00000000 A5=00000000 A6=00000FFC A7=00000FF0 Z=0
D0=00000000 D1=FFFFFFFF D2=FFFFFFFF D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->MOVE.L D2,D3
```

注意突出显示的指令MOVE.L D1, -4(A6)，这是一个带偏移地址的寄存器间接寻址的例子。编译器使用A6作为堆栈指针将数据放置到了其局部堆栈帧。

堆栈帧只是变量存储区域，是编译器在堆栈上为其函数变量建立的。余下的模拟结果没有提供任何细节，所以现在给大家留一个习题：请追历该指令和堆栈条件以确信其确实像预期那样工作了。

```
PC=000416 SR=2008 SS=00000FF0 US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=1
A4=00000000 A5=00000000 A6=00000FFC A7=00000FF0 Z=0
D0=00000000 D1=FFFFFFFF D2=FFFFFFFF D3=FFFFFFFF V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
----------->MOVEQ #0,D2
PC=000418 SR=2004 SS=00000FF0 US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000FFC A7=00000FF0 Z=1
D0=00000000 D1=FFFFFFFF D2=00000000 D3=FFFFFFFF V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
----------->MOVE.L #5555,-4(A6)
PC=000420 SR=2000 SS=00000FF0 US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000FFC A7=00000FF0 Z=0
D0=00000000 D1=FFFFFFFF D2=00000000 D3=FFFFFFFF V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
----------->ADDQ.L #1,D2
PC=000422 SR=2000 SS=00000FF0 US=00000000        X=0
A0=00000000 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000FFC A7=00000FF0 Z=0
D0=00000000 D1=FFFFFFFF D2=00000001 D3=FFFFFFFF V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
----------->LEA.L -4(A6),A0
PC=000426 SR=2000 SS=00000FF0 US=00000000        X=0
A0=00000FF8 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000FFC A7=00000FF0 Z=0
D0=00000000 D1=FFFFFFFF D2=00000001 D3=FFFFFFFF V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
----------->PEA (A0)
PC=000428 SR=2000 SS=00000FEC US=00000000        X=0
A0=00000FF8 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000FFC A7=00000FEC Z=0
D0=00000000 D1=FFFFFFFF D2=00000001 D3=FFFFFFFF V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
----------->MOVEA.L D2,A0
PC=00042A SR=2000 SS=00000FEC US=00000000        X=0
A0=00000001 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000FFC A7=00000FEC Z=0
D0=00000000 D1=FFFFFFFF D2=00000001 D3=FFFFFFFF V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
----------->PEA (A0)
PC=00042C SR=2000 SS=00000FE8 US=00000000        X=0
A0=00000001 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000FFC A7=00000FE8 Z=0
D0=00000000 D1=FFFFFFFF D2=00000001 D3=FFFFFFFF V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
----------->JSR 28(PC)
```

1000	00	00	00	00
0FFC	00	00	00	02
0FF8	FF	FF	FF	FF
0FF4	00	00	00	00
0FF0	00	00	00	00

1000	00	00	00	00
0FFC	00	00	00	00
0FF8	00	00	15	B3
0FF4	00	00	00	00
0FF0	00	00	00	00

1000	00	00	00	00
0FFC	00	00	00	00
0FF8	00	00	15	B3
0FF4	00	00	00	00
0FF0	00	00	00	00
0FEC	00	00	00	F8

1000	00	00	00	00
0FFC	00	00	00	00
0FF8	00	00	15	B3
0FF4	00	00	00	00
0FF0	00	00	00	00
0FEC	00	00	0F	F8
0FE8	00	00	00	01

249

250

```
C=00044A SR=2000 SS=00000FE4 US=00000000          X=0
A0=00000001 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000FFC A7=00000FE4 Z=0
D0=00000000 D1=FFFFFFFF D2=00000001 D3=FFFFFFFF V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
----------->LINK A6,#0
PC=00044E SR=2000 SS=00000FE0 US=00000000          X=0
A0=00000001 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000FE0 A7=00000FE0 Z=0
D0=00000000 D1=FFFFFFFF D2=00000001 D3=FFFFFFFF V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
----------->MOVEA.L 12(A6),A0
PC=000452 SR=2000 SS=00000FE0 US=00000000          X=0
A0=00000FF8 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000FE0 A7=00000FE0 Z=0
D0=00000000 D1=FFFFFFFF D2=00000001 D3=FFFFFFFF V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
----------->MOVE.L A0,D1
PC=000454 SR=2000 SS=00000FE0 US=00000000          X=0
A0=00000FF8 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000FE0 A7=00000FE0 Z=0
D0=00000000 D1=00000FF8 D2=00000001 D3=FFFFFFFF V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
----------->BEQ.S $00000456
PC=000456 SR=2000 SS=00000FE0 US=00000000          X=0
A0=00000FF8 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000FE0 A7=00000FE0 Z=0
D0=00000000 D1=00000FF8 D2=00000001 D3=FFFFFFFF V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
----------->MOVEQ #-1,D0
PC=000458 SR=2008 SS=00000FE0 US=00000000          X=0
A0=00000FF8 A1=00000000 A2=00000000 A3=00000000 N=1
A4=00000000 A5=00000000 A6=00000FE0 A7=00000FE0 Z=0
D0=FFFFFFFF D1=00000FF8 D2=00000001 D3=FFFFFFFF V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
----------->CMP.L D0,D1
PC=00045A SR=2001 SS=00000FE0 US=00000000          X=0
A0=00000FF8 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000FE0 A7=00000FE0 Z=0
D0=FFFFFFFF D1=00000FF8 D2=00000001 D3=FFFFFFFF V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=1
----------->BNE.S $00000470
PC=000470 SR=2001 SS=00000FE0 US=00000000          X=0
A0=00000FF8 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000FE0 A7=00000FE0 Z=0
D0=FFFFFFFF D1=00000FF8 D2=00000001 D3=FFFFFFFF V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=1
----------->MOVE.L 8(A6),D0
PC=000474 SR=2000 SS=00000FE0 US=00000000          X=0
A0=00000FF8 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000FE0 A7=00000FE0 Z=0
D0=00000001 D1=00000FF8 D2=00000001 D3=FFFFFFFF V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
----------->ADD.L (A0),D0
PC=000476 SR=2000 SS=00000FE0 US=00000000          X=0
A0=00000FF8 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000FE0 A7=00000FE0 Z=0
D0=000015B4 D1=00000FF8 D2=00000001 D3=FFFFFFFF V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
----------->BRA.L $0000047A
PC=00047A SR=2000 SS=00000FE0 US=00000000          X=0
A0=00000FF8 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000FE0 A7=00000FE0 Z=0
D0=000015B4 D1=00000FF8 D2=00000001 D3=FFFFFFFF V=0
```

1000	00	00	00	00
OFFC	00	00	00	00
OFF8	00	00	15	B3
OFF4	00	00	00	00
OFF0	00	00	00	00
OFEC	00	00	0F	F8
OFE8	00	00	00	01
OFE4	00	00	04	30
OFE0	00	00	00	00

1000	00	00	00	00
OFFC	00	00	00	00
OFF8	00	00	15	B3
OFF4	00	00	00	00
OFF0	00	00	00	00
OFEC	00	00	0F	F8
OFE8	00	00	00	01
OFE4	00	00	04	30
OFE0	00	00	00	00

1000	00	00	00	00
OFFC	00	00	00	00
OFF8	00	00	15	B3
OFF4	00	00	00	00
OFF0	00	00	00	00
OFEC	00	00	0F	F8
OFE8	00	00	00	01
OFE4	00	00	04	30
OFE0	00	00	00	00

1000	00	00	00	00
OFFC	00	00	00	00
OFF8	00	00	15	B3
OFF4	00	00	00	00
OFF0	00	00	00	00
OFEC	00	00	0F	F8
OFE8	00	00	00	01
OFE4	00	00	04	30
OFE0	00	00	00	00

```
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->NOP
PC=00047C SR=2000 SS=00000FE0 US=00000000        X=0
A0=00000FF8 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000FE0 A7=00000FE0 Z=0
D0=000015B4 D1=00000FF8 D2=00000001 D3=FFFFFFFF V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->UNLK A6
PC=00047E SR=2000 SS=00000FE4 US=00000000        X=0
A0=00000FF8 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000FFC A7=00000FE4 Z=0
D0=000015B4 D1=00000FF8 D2=00000001 D3=FFFFFFFF V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->RTS
PC=000430 SR=2000 SS=00000FE8 US=00000000        X=0
A0=00000FF8 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000FFC A7=00000FE8 Z=0
D0=000015B4 D1=00000FF8 D2=00000001 D3=FFFFFFFF V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->ADDQ.L #8,SP
PC=000432 SR=2000 SS=00000FF0 US=00000000        X=0
A0=00000FF8 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000FFC A7=00000FF0 Z=0
D0=000015B4 D1=00000FF8 D2=00000001 D3=FFFFFFFF V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->MOVE.L D0,D3
PC=000434 SR=2000 SS=00000FF0 US=00000000        X=0
A0=00000FF8 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000FFC A7=00000FF0 Z=0
D0=000015B4 D1=00000FF8 D2=00000001 D3=000015B4 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->ADDQ.L #1,D3
PC=000436 SR=2000 SS=00000FF0 US=00000000        X=0
A0=00000FF8 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000FFC A7=00000FF0 Z=0
D0=000015B4 D1=00000FF8 D2=00000001 D3=000015B5 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->NOP
PC=000438 SR=2000 SS=00000FF0 US=00000000        X=0
A0=00000FF8 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000FFC A7=00000FF0 Z=0
D0=000015B4 D1=00000FF8 D2=00000001 D3=000015B5 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->MOVE.L (SP)+,D2
PC=00043A SR=2004 SS=00000FF4 US=00000000        X=0
A0=00000FF8 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000FFC A7=00000FF4 Z=1
D0=000015B4 D1=00000FF8 D2=00000000 D3=000015B5 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->MOVE.L (SP)+,D3
PC=00043C SR=2004 SS=00000FF8 US=00000000        X=0
A0=00000FF8 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000FFC A7=00000FF8 Z=1
D0=000015B4 D1=00000FF8 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
---------->UNLK A6
PC=00043E SR=2004 SS=00001000 US=00000000        X=0
A0=00000FF8 A1=00000000 A2=00000000 A3=00000000 N=0
A4=00000000 A5=00000000 A6=00000000 A7=00001000 Z=1
D0=000015B4 D1=00000FF8 D2=00000000 D3=00000000 V=0
D4=00000000 D5=00000000 D6=00000000 D7=00000000 C=0
```

252

1000	00	00	00	00
0FFC	00	00	00	00
0FF8	00	00	15	B3
0FF4	00	00	00	00
0FF0	00	00	00	00
0FEC	00	00	0F	F8
0FE8	00	00	00	01
0FE4	00	00	04	30
0FE0	00	00	0F	FC

1000	00	00	00	00
0FFC	00	00	00	00
0FF8	00	00	15	B3
0FF4	00	00	00	00
0FF0	00	00	00	00
0FEC	00	00	0F	F8
0FE8	00	00	00	01
0FE4	00	00	04	30
0FE0	00	00	0F	FC

1000	00	00	00	00
0FFC	00	00	00	00
0FF8	00	00	15	B3
0FF4	00	00	00	00
0FF0	00	00	00	00
0FEC	00	00	0F	F8
0FE8	00	00	00	01
0FE4	00	00	04	30
0FE0	00	00	0F	FC

1000	00	00	00	00
0FFC	00	00	00	00
0FF8	00	00	15	B3
0FF4	00	00	00	00
0FF0	00	00	00	00
0FEC	00	00	0F	F8
0FE8	00	00	00	01
0FE4	00	00	04	30
0FE0	00	00	0F	FC

1000	00	00	00	00
0FFC	00	00	00	00
0FF8	00	00	15	B3
0FF4	00	00	00	00
0FF0	00	00	00	00
0FEC	00	00	0F	F8
0FE8	00	00	00	01
0FE4	00	00	04	30
0FE0	00	00	0F	FC

```
---------->RTS
```

1000	00	00	00	00
OFFC	00	00	00	00
OFF8	00	00	15	B3
OFF4	00	00	00	00
OFF0	00	00	00	00
OFEC	00	00	0F	FB
OFE8	00	00	00	01
OFE4	00	00	04	30
OFE0	00	00	0F	FC

你可能奇怪为什么最后一条指令是RTS。如果这个程序要在一个操作系统下运行（C编译器的预计就是这样的），那么，该指令会将控制归还给操作系统。

在我们离开68000汇编语言并继续进入到其他体系结构之前，还有最后一件事情要讨论，就是指令集分解问题。我们已经从指令集、寻址模式以及内部寄存器资源等角度考察了体系结构问题，现在我们试图将这些与我们对状态机的讨论相关联，来看看指令的编码实际上是如何进行的。 253

将机器语言转换回到汇编语言的过程称为反汇编（disassembly）。反汇编器是一个程序，用于检查存储器中的机器代码并试图将其转换回到汇编语言。这是一个极其有用的调试工具，事实上，多数调试器都有内置的反汇编特性。

你可能还记得，我们前面介绍汇编语言时讲到过，一个指令的第一个字称为操作代码字（opcode word）。操作代码字包含一个操作代码，用于告诉计算机要做什么，它还包含零个、一个或两个有效地址域（effective address field, EA）。有效地址域包含编码过的信息，告诉处理器如何从存储器中取回操作数。换句话说，操作数的有效地址是什么。

你已知道，一个操作数可以是一个地址寄存器也可以是一个数据寄存器。操作数可处于存储器中，但由地址寄存器指向它。考虑如下所示的指令形式：

操作代码		目的EA		源EA	
DB15	DB12	DB11	DB6	DB5	DB0

操作代码域包含从DB12到DB15的4个位，它告诉计算机这是一条移动指令，也就是将由源EA所规定的存储器位置的内容移动到由目的EA所规定的存储器位置中。此外，有效地址域进一步被分解为一个3位的模式域和一个3位的子类（寄存器）域。

这个信息已足够告诉计算机它要完成指令所需要知道的任何事情，但操作代码本身不一定有完成指令所必需的所有信息。为完成指令，计算机也许要再访问存储器一到两次来取出关于源EA或目的EA的额外信息。如果我们回想起操作代码字是指令的一部分，而指令必须由微代码驱动的状态机进行译码，这就开始变得有意义了。一旦通过对操作代码字进行译码而确立了适合的状态机序列，如果必要的话，就要从存储器中取出另外的操作数。

例如，假设源EA和目的EA都是数据寄存器，换句话说，指令是：

```
MOVE.W  D0,D1
```

由于源和目的都是内部寄存器，处理器就不需要额外的信息来执行指令。在这个例子中，指令的长度和操作代码的长度一样，是16位长，即一个字的长度。

然而，假设指令是：

```
MOVE.W  #$00D0,D1
```

254 现在，#号告诉我们源EA是一个立即操作数（16进制数$00D0），目的EA仍是寄存器D1。在该例中，操作代码告诉我们源是一个立即操作数，但它不能告诉我们该立即操作数的实际值。计算机必须再到存储器中取回操作代码字之后的下一个字，这就是数据值$00D0。如果这条指令处于存储器$1000的位置，则操作代码字将占据字地址$1000，而操作数将处于位置$1002。

有效地址域是16位宽的域，它又进一步分为两个3位域，分别称为模式（mode）和寄存器（register）。3位宽的模式域可指定8种可能的寻址方式之一，寄存器域可指定8个可能的寄存器之一，或称为模式域的一个子类（subclass）。

MOVE指令的独特性在于其有两个可能的有效地址域，所有其他的指令可能只包含一个有效地址域。你开始时也许对此感到奇怪，但是你将会看到，几乎所有其他采用两个操作数的指令都必须涉及一个寄存器和一个有效地址。

让我们将注意力放回到MOVE指令。如果我们完全将其分解成其组成部分，则将如下所示：

0	0	大小		目的寄存器			目的模式			源模式			源寄存器		
DB15	DB14	DB13	DB12	DB11	DB10	DB9	DB8	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0

子类只用于模式7寻址模式。这些模式是：

模式7寻址	子类
绝对字	000
绝对长字	001
立即数	100
带位移的PC相对	010
带变址的PC相对	011

你可能已经注意到了移动到地址寄存器（MOVEA）指令，并疑惑我们为什么还需要再创造一个需要特殊记忆的一般的MOVE指令。首先，地址寄存器是极其有用的重要内部资源，有地址寄存器就意味着我们能做寄存器算术操作和指针操作。我们也能比较寄存器的值并根据数据的地址做决策。

MOVEA指令是一个更标准的指令类型，因为只有一个有效地址用于源操作数，而目的操作数是7个地址寄存器之一。这样，目的模式就硬性地编码为地址寄存器。然而，我们需要创造一个特殊的指令用于MOVEA操作，因为在奇数字节边界取字是一个非法的访问，而MOVEA指令防止了这种情况的发生。

大部分指令呈现如下的形式：

操作码		寄存器		操作模式		源EA或目的EA	
DB15	DB12	DB11	DB9	DB8	DB6	DB5	DB0

操作码域规定指令是否是ADD、AND、CMP，等等，而寄存器域规定哪一个内部寄存器是操作的源或目的。操作模式域（op mode field）是一个新术语，它为指令规定了8种细分情况中的一种。例如这个域规定，是否内部寄存器就是操作的源或目的，以及是否大小是字节、字或长字。我们将在稍后看到这些情况。

单操作数指令如CLR（将目的EA的内容置为0）有稍微不同的形式：

操作码	大小	目的EA
DB15 DB8	DB7 DB6	DB5 DB0

255

JMP（跳转）和JSR（跳转到子程序）也是单操作数指令。两个指令都将目的EA放入到了程序计数器，使得下一条指令从新位置取出，而不是序列的下一条指令。在PC的内容被目的EA取代之前，JSR指令也将自动地将程序计数器的当前值放入到堆栈。这样，返回位置就被存到堆栈中。

转移指令Bcc（根据条件码转移）在修改PC内容使下一条指令不按序取出这方面类似于跳转指令。然而，转移指令在两个基本方式上有所不同：

1. 没有有效地址。位移值会被加入到PC的当前内容中，使得新值由一个当前值的正移动或负移动来确定。因此，位移是一个相对跳转，而不是绝对跳转。

2. 只有被测试的条件码求值为真时才进行转移操作。

0 1 1 0	条件	位移
DB15 DB12	DB11 DB8	DB7 DB0

位移域是8位的值，这意味着转移可以移动到一个离目前位置有+127字节或−128字节远的另一个位置。如果想得到更远的转移，那么可以将位移域全部置零且将下一个存储器字用作位移的立即数值，就可达到从+16 383字节到−16 384字节的范围。

这样，如果计算机正在执行一个短循环，则用8位的位移操作起来效率会更高。16位的位移可以使指令转移到更远的地方，但其代价是需要额外的存储器取数操作。

早些时候我曾说过如果条件码求值为真则执行转移，这意味着与仅能测试CCR中一个标记的状态相比，我们还可测试更多的条件。下面的表为我们列出了转移条件求值的可能方式。

CC	进位清零	0100	$\overline{C}$	LS	低或相等	0011	C+Z
CS	进位置位	0101	C	LT	小于	1101	$N*\overline{V}+\overline{N}*V$
EQ	相等	0111	Z	MI	负	1011	N
GE	大于或等于	1100	$N*V+\overline{N}*\overline{V}$	NE	不相等	0110	$\overline{Z}$
GT	大于	1110	$N*V*\overline{Z}+\overline{N}*\overline{V}*\overline{Z}$	PL	正	1010	$\overline{N}$
HI	高	0010	$\overline{C}*\overline{Z}$	VC	溢出清零	1000	$\overline{V}$
LE	小于或等于	1111	$Z+N*\overline{V}+\overline{N}*V$	VS	溢出置位	1001	V

有如此多转移测试条件的原因是，转移指令BGT、BGE、BLT以及BLE是与有符号算术操作一起使用的，而转移指令BHI、BCC、BLS和BCS则是与无符号算术操作一起使用的。

有些指令，如**NOP**和**RTS**（从子程序中返回）不需要操作数，完全由其操作代码字规定。

早些时候我们讨论了操作模式域在如何对一般指令操作做进一步限制方面所扮演的角色。为示意操作模式域如何工作，一个好的途径是更详细地考察ADD指令及其所有变形，见图9-3。256
注意包含在第6、7、8位的操作模式域是如何定义指令格式的。**ADD**指令是大多数“常规”指令的代表，而其他类的指令则需要特殊考虑。例如，**MOVE**指令包含两个有效地址。此外，所有立即寻址模式指令的源操作数寻址模式都是硬编码到指令中的，所以它不是一个真正的有效地址，这包括诸如加立即数（**ADDI**）和减立即数（**SUBI**）等指令。

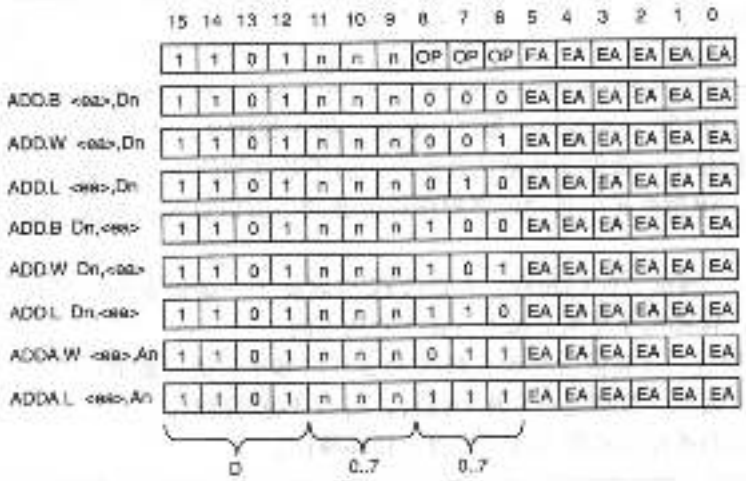


图9-3 ADD指令的操作模式分解

一个真实机器的例子

在我们继续考虑其他体系结构之前，让我们做一些略有不同的，但仍与我们对汇编代码和计算机体系结构的讨论有关的事情。迄今为止，我们实在是忽视了我们编写的代码最终要运行于一台真实机器的事实。存储器测试程序确实不那么令人兴奋，但至少我们可以想像它们是如何与实际硬件一起使用的。为了对我们关于汇编语言编码的讨论做出结论，让我们来看一个实际的基于68000系统的设计。

图9-4和图9-5是一个68K计算机系统的处理器和存储器部分的简化示意图，并没显示出所有的信号，但这不会影响讨论。而且，我们不想让细节来烦我们。

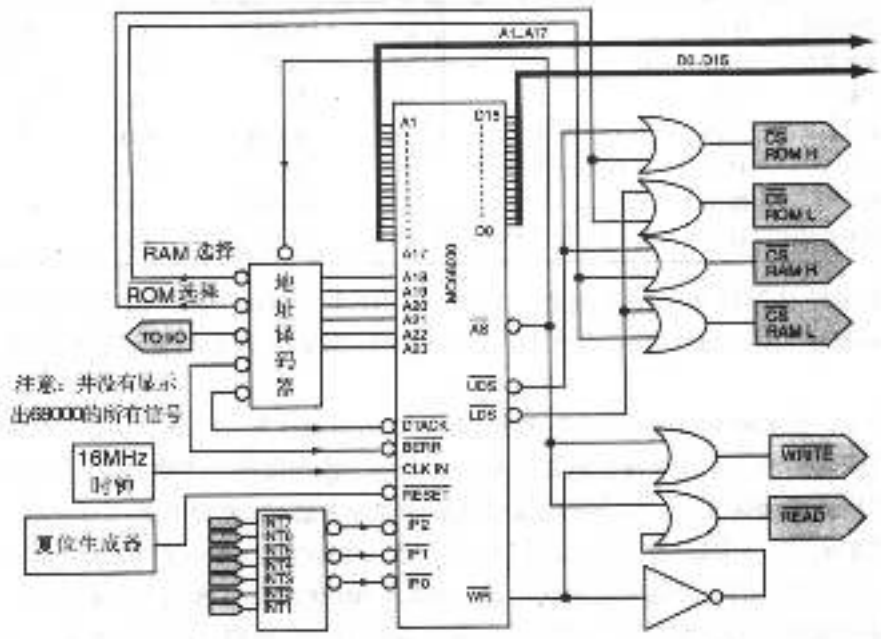


图9-4 基于68000计算机系统的简化示意图，此图中只显示了68000的CPU

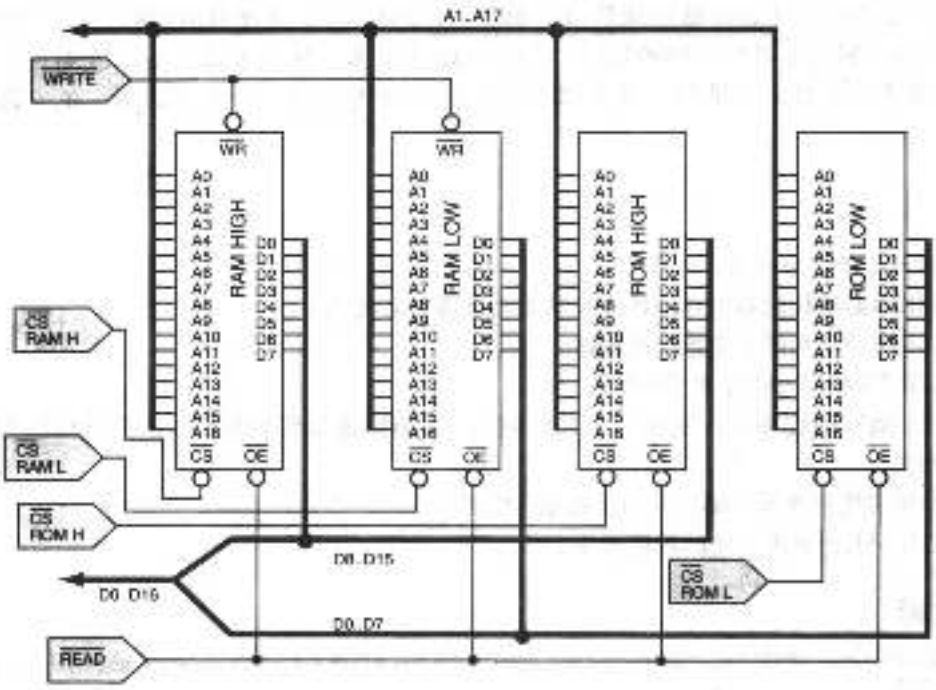


图9-5 图9-4的68K计算机系统的存储器系统的简化示意图

参见地址总线，如图9-4和图9-5所示，我们看到有没有标记为A0的地址位。正如我们在图7-6中所见到的，A0是由状态机通过 $\overline{UDS}$ 或 $\overline{LDS}$ 合成出来的。由于所有相关的控制信号都是低有效的，所以图右侧的或门实际上是被用作反相的逻辑与门。 257

68K环境中的有效地址（valid address）信号称为地址选通（address strobe），在图中用 $\overline{AS}$ 标记。这个信号被传送到两个地方。它被用于读操作和写操作的准许信号，在图中通过下面的两个或门触发（要一直想着是"反相的逻辑与门"），并到达地址译码模块，辨别出这个功能模块并弄明白它是如何工作的，对你来说应该没有困难。紧要时你可能要自己设计！因此，在 $\overline{AS}$ 信号变低保证地址有效之前，我们将不对地址进行译码。I/O译码器还可以对I/O设备进行译码，虽然在这个图中没有显示。

由于68K的读信号和写信号不是独立的，所以我们用非门和或门来合成 $\overline{READ}$ 信号。我们还可以用或门和 $\overline{AS}$ 信号来限制READ信号和WRITE信号，或门起到了一个反相逻辑与门的作用。严格地说这是不必要的，因为我们已经在用同样的方式限制地址译码信号了。 258

最后一个要注意的模块是位于图中下面左侧的中断控制器模块。68K采用了低有效的中断输入，标记为 $IP0$、$IP1$ 和 $\overline{IP2}$。所有三条线都变低将产生一个第7级中断。我们将在第12章中研究中断。中断控制器的输入是7个标记为INT1－INT7的低有效输入。如果INT7变低，则所有的输出线也将变低。指出INT7是一个第7级中断。现在，如果INT6为低（$\overline{IP0}=1$、$\overline{IP1}=0$、$\overline{IP2}=0$），INT5也变低，则输出将不改变，因为中断控制器不但要对中断输入译码，而且还要对它们进行优先级排队。

图9-5显示的是计算机的存储器一侧。它由两个128K×8的RAM芯片和两个128K×8的ROM芯片组成。注意成对的器件和来自处理器的两个片选信号是怎样一起得出字节写控制的。电路的其余部分对于你应该是非常熟悉了。

这个计算机设计的存储系统是由256KB的ROM组成，处于从$000000到$3FFFF的地址位置。RAM处于从$100000到S13FFFF的地址位置。从这个简化的示意图来看，这个地址映射并不是显而易见的，因为你不知道图9-4中标记为“地址译码器”的电路模块的细节。

总结

本章涵盖的内容如下：

- 68K体系结构的高级寻址模式及其与高级语言的关系。
- 68K体系结构的各类指令的概述。
- 采用TRAP #15指令来模拟I/O。
- 用C编写的程序在汇编语言中如何运行，C编译器如何利用68K体系结构的高级寻址模式。
- 如何实现程序反汇编及其与体系结构的关系。
- 基于68K计算机系统的功能模块。

参考文献

259 [1] Alan Clements, *68000 Family Assembly Language*. ISBN 0-534-93275-4, PWS Publishing Company, Boston, 1994, p. 704

习题

1. 考察下面显示的汇编语言代码块。
   a. 在指示出的指令中，存储字节FF的存储器地址是什么？
   b. 该代码段是可重定位的吗？为什么？

```
        org         $400
start   movea.w     #$2000,A0
        move.w      #$0400,D0
        move.b      #$ff,($84,A0,D0)  * 什么意思？
        jmp         start
        end         $400
```

提示：记住位移涉及补码数。另外，该指令也可写成：

```
move.b #$FF,$84(A0,D0)
```

2. 考察下面的代码段，然后回答关于该代码段所执行的操作的问题。你可以假设堆栈已适当地初始化了。简要描述那条突出显示的指令的结果。什么值被移动到目的地址？

```
00001000  41F9 00001018        10  START:   LEA     DATA+2,A0
00001006  2C50                 11           MOVEA.L (A0),A6
00001008  203C 00001B00        12           MOVE.L  #$00001B00,D0
0000100E  2C00                 13           MOVE.L  D0,D6
00001010  2D80 6846            14           MOVE.L  D0,$70(A6,D6.L)
00001014  60FE                 15  STOP_IT: BRA     STOP_IT
00001016  00AA0040 C8300000    16           DATA:   DC.L
$00AA0040,$C8300000
```

3. 考察下面的代码段，然后回答关于该代码段所执行的操作的问题。你可以假设堆栈已适当地初始化了。简要描述那条突出显示的指令的结果。突出显示的指令完成后，寄存器D0的值是什么？

```
00000400 4FF840C0      START:   LEA      $40C0,SP
00000404 3F3C1CAA               MOVE.W   #$1CAA,-(SP)
00000408 3F3C8000               MOVE.W   #$8000,-(SP)
0000040C 223C00000010           MOVE.L   #16,D1
00000412 203C215E0000           MOVE.L   #$215E0000,D0
00000418 E2A0                   ASR.L    D1,D0
0000041A 383C1000               MOVE.W   #$1000,D4
0000041E 2C1F                   MOVE.L   (SP)+,D6
00000420 C086                   AND.L    D6,D0
00000422 60FE          STOP_HERE: BRA    STOP_HERE
```

4. 用一、两句话回答下述问题：

a. 当C++函数退出时，为什么局部变量就不起作用了？

b. 给出两个理由，说明为什么高级语言（如C）中的变量在使用前必须先声明？

c. 68000汇编语言指令LINK和ULINK是这样一些指令的代表，它们被创造的目的是为了支持一种高级语言。为什么？

5. 下面是32字节存储器内容的显示，你可以在调试器中用"显示存储器"看到这些内容。这个内容所显示的3条指令是什么？

```
00000400 06 79 55 55 00 00 AA AA 06 B9 AA AA 55 55 00 00
00000410 FF FE 06 40 AA AA 00 00 FF 00 00 00 00 00 00 00
```

6. 假设你正在试图为存储器中的68K指令写一个反汇编程序，寄存器A6指向你正要反汇编的下一条指令的操作代码字。考察下面的算法，用语言描述它是如何工作的。对于这个例子，你可假设<A6> = $00001A00且<$00001A00> = %1101111001100001。

```
shift       EQU     12              *移12位

start       LEA     jmp_table,A0    *变址进入表中
            CLR.L   D0              *对其清零
            MOVE.W  (A6),D0         *在这里处理
            MOVE.B  #shift,D1       *右移12位
            LSR.W   D1,D0           *移动这些位
            MULU    #6,D0           *形成偏移
            JSR     00(A0,D0)       *指令五的间接跳转
            〈 其他指令 〉

jmp table   JMP     code0000
            JMP     code0001
            JMP     code0010
            JMP     code0011
            JMP     code0100
            JMP     code0101
            JMP     code0110
            JMP     code0111
            JMP     code1000
            JMP     code1001
            JMP     code1010
            JMP     code1011
            JMP     code1100
            JMP     code1101
            JMP     code1110
            JMP     code1111
```

7. 将第8章习题10的存储器测试程序转换成可重定位的。为了看看你是否已经成功了，请按如下要求写程序：

a. 程序起始(ORG)于$400。

b. 在起始处加一些代码，使得当它开始运行时，重定位到存储器中$000A0000的位置。

c. 测试从$00000400到$0009FFF0的存储器区域。

d. 确保对堆栈的定位使之不被写覆盖。

8. 这个习题将通过对第8章的存储器测试习题引入几个新的概念来拓展你对68K的掌握，它也是一个用适当的子程序构建汇编代码的好例子。这个习题通过TRAP #15指令的运用来引入用户I/O的概念，你应该通过研究与E68K程序一起提供的帮助软件来熟知TRAP #15指令。另外，你也可以熟读Clements[1]教材的704页中关于TRAP #15指令的细节。

TRAP #15指令是模拟器的产物，设计它的目的是允许模拟器和用户之间发生I/O操作。如果你真的在写I/O程序，那么你可能会做一些不同的事情，但很多事情是相同的。

与TRAP #15指令相关联的是各种任务，每个任务都有编号，与每个任务相关联的是一个API，用来解释它如何工作。例如，任务#0要将一串字符打印在显示器上，并加入一个换行符使得光标前进到下一行的开始处。为了使用任务#0，你必须建立下面的寄存器(这就是API)：

- D0以字节装载任务编号
- A1装载字符串开始处的地址
- D1以字存储着要打印字符串的长度

一旦建立了这三个寄存器，你就能将TRAP #15作为一条指令调用，你的消息就能打印在显示器上了。这里是一个示意其如何工作的样例程序。为了输出字符串“Hello world!”，你可以采用如下的代码片段：

```
*************************************************
*   将一个字符串打印到显示器的测试程序
*
*************************************************
          OPT         CRE
task0     EQU         00

          ORG         $400

start     MOVE.B      #task0,D0          *将任务编号装入D0
          LEA         string,A1          *获得字符串的地址
          MOVE.W      str_len,D1         *获得字符串的长度
          TRAP        #15                *开始打印
          STOP        #$2700             *回到模拟器

* 数据区域

string    DC.B    'Hello world'          *这里存储消息
str_len   DC.W     str_len-string        *获得字符串的长度
          END     $400
```

262

除了不打印换行符，任务#1与任务#0几乎相同。当你想提示用户输入信息时，这就很便利。你就应该采用任务#1而不是任务2来发出提示符。

此外，你可能想从用户那里获得信息。比如，“他们想从哪里运行存储器测试以及他们想用什么测试码？”还有，你可能会问他们是否想用一个不同的测试码再运行一次测试。

问题陈述

1. 拓展存储器测试程序（第8章，习题10），使用户能够输入存储器测试的起始地址、存储器测试的结束地址以及测试所采用的字测试码。只使用输入测试码，不需要对位取反或做任何其他测试。
2. 要测试的存储区域在$00001000和$000A0000之间。

3. 程序首先打印出一个测试概要标题，包含如下信息:

地址（ADDRESS） 写数据（DATA WRITTEN） 读数据（DATA READ）

4. 每次发生错误，你要打印错误地址、你写的数据码以及你读回的数据码。
5. 将堆栈定位在$A0000以上。
6. 当程序开始运行时，它提示用户输入测试的起始地址（在$00001000之上），用户就输入测试的起始地址。然后，程序就提示用户输入测试的结束地址（在$FFFF以下），用户就输入这个地址。最后，程序提示用户输入测试所采用的字测试码。

你必须检查一下，看看结束地址是否比起始地址至少大一个字的长度。你不必为数字输入的正确性进行测试，在这个问题中你可以假设只输入了正确的地址和数据值。正确的输入是：数字0到9、小写字母a到f、大写字母A到F。

一旦这些信息被输入，测试概要标题行就打印到显示器，测试开始。遇到的任何错误 263
也如上所述打印到显示器上。

讨论

一旦你从用户那里获得字符串，你就必须认识到它是ASCII格式的。ASCII是用于打印字符和读取字符的代码。数字0到9的ASCII代码是$30...$39，字母a到f的ASCII代码是$61...$66，字母A到F的ASCII代码是$41...$46。

因此，如果提示用户输入地址，而用户输入了9B56，则当TRAP #15指令结束时，存储器中将有这样4个字节：$39、$42、$35、$36。那么，你如何从这些字节得到地址$9B56呢？我们必须写一个算法。换句话说，你必须将ASCII值转换成等价的16位字，这是练习移位和屏蔽技术的好机会。为了得到一个表示成ASCII 0-9的数字，你必须从这个ASCII值中减去$30，剩下的就是这个数字值。如果数字是“B”，则你也要减去一个不同的数字来得到十六进制值$B。类似地，如果数字是“b”，则你也必须减去另一个不同的值。

下面就是地址$9B56的二进制形式：

```
  DB15          DB12          DB7          DB3          DB0
1 0 0 1       1 0 1 1       0 1 0 1       0 1 1 0
```

现在，我有从ASCII $39译码得到的值9，且它处于位置0到3，我们如何使它处于位置 264
13到15呢？用一个流程图设计出这个算法是很有意义的。

第10章 Intel x86体系结构

学习目标

- 描述8086和8088微处理器的处理器体系结构；
- 描述8086和8088处理器的基本指令集体系结构；
- 使用段的地址存储器：偏移的寻址技术；
- 描述8086体系结构和68000体系结构之间的异同；
- 结合8086体系结构的所有寻址模式和指令，编写8086汇编语言的简单程序。

10.1 引言

Intel是当今世界上最大的半导体生产厂商。它上升到目前的地位是由于与IBM的供应伙伴关系。当时，IBM需要为它正在佛罗里达州波卡瑞顿开发的新型PC-XT个人计算机配置一种处理器。IBM的最佳选择是Zilog公司的Z-800，而Zilog公司是Intel的竞争对手之一，该公司由那些研究最初的Intel 8080处理器的Intel员工组成。Zilog公司生产了一种与8080编码兼容的增强型处理器——Z80, Z80能够执行所有的8080指令和一些其他指令，而且也比8080似乎更易于进行接口。

电脑爱好者们乐于使用Z80，Z80成了几乎所有早期基于CP/M操作系统（由Digital Research公司的Gray Kidall开发）的PC的首选处理器。当时传说Zilog公司正在研究Z80的16位版本——Z800，这种处理器将比8位Z80有很大的性能提高。IBM最初提议Zilog作为PC-XT的可能的CPU供应商。今天，应该承认的是：Intel发展到目前这种辉煌的地步是因为Zilog当时没能按期交货给IBM。

1978年，Intel有了8080的16位继承者——8086/8088。8088在内部与8086相同，只不过8088是8位外部数据总线，而8086是16位。这当然会限制设备的性能，但是对于IBM来说，它吸引人的特性是能够大幅度降低系统成本。Intel能够满足IBM的订货时间要求，一个频率为4.077MHz的8088 CPU被配置到最初的IBM PC-XT个人计算机上。对于操作系统，IBM选择了西雅图一个小软件公司的类似CP/M的16位操作系统，该公司名叫微软。尽管Zilog还是发展到了今天，但他们仍然没有从未能发行Z800的失败中恢复过来。这是一个错过交付日期的经典例子，所有软件开发商们都应该牢记其教训。顺便提一下，最初的Z80也沿用到了今天并将继续作为嵌入式控制器使用。很难估计它的影响到底有多大，但是必定还有数十亿行Z80代码仍然在使用。

PC产业的发展基本上是沿着Intel x86体系结构的不断发展和改进的方向前进的。Intel开发出80286作为后继的CPU，IBM则制造出使用这种CPU的PC-AT计算机。80286引入了保护模式（protected mode）的概念，这使得操作系统能够实现任务管理，以至于单任务操作系统MS-DOS都被扩展到能够允许简单形式的多任务并发。80286仍然是16位机器。在这个时期，Intel还开发了80186/80188集成微控制器系列，它们是使用8086/8088核和额外的片上外围设备的CPU，这使得它们对于很多嵌入式计算机产品来说具有作为片上解决方案的巨大魅力。这些片上外围设备包括：

- 2个直接存储器访问控制器（DMA）
- 3个16位可编程定时器
- 时钟发生器
- 片选单元
- 可编程控制寄存器

随着其他半导体厂商生产出更高集成度的变种，例如AMD公司的E86™系列和NEC的V-系列，80186/188系列在今天仍然很受欢迎。特别是外围设备的组合使得186系列对控制器的磁盘驱动器制造商具有很大的吸引力。

伴随着80386的引入，x86体系结构最终被扩充到了32位。PC界对硬件和软件（Windows 3.0）的反应主要集中在这种体系结构。80386的后继产品80486和奔腾（Pentium）系列继续改进了这种在i386中定义的基本体系结构。

对这个系列的兴趣使得我们从i386体系结构开始回溯并关注于最初的8086系列体系结构。这种体系结构将给我们提供一个很好的参考点，使我们能获得68000和8086的对比性了解。68000通常被规定为16/32位的体系结构，而8086也能够做很多32位的操作。而且，68K的默认操作数大小是16位，所以这两种体系结构之间的相关对比是相当有效的。

因此，我们将从8086的视角来看待x86系列。还有很多关于8086后继体系结构的文献可供参考，感兴趣的读者可以找找。

当开始学习8086体系结构和指令集体系结构时，显然我们很多的注意力是集中在DOS操作系统和PC运行时环境上的。对于这一观点，我们有个关于68K体系结构的有趣的反例。当研究68K时，我们是在一个“仿真”环境中运行，因为PC机和68K的指令集是不兼容的。所以，应该能够由程序产生虚拟的68K，而且该程序创建的68K虚拟机可供你的代码在上面 266
运行。

对于8086 ISA来说，则执行在“本地”环境下，因为汇编程序的指令集与机器是兼容的。

因此，你可以很轻松地执行（运行良好的）在PC机DOS窗口内编写的8086程序。事实上，通过PC机的基本输入输出系统（BIOS）调用DOS操作，就能很容易地实现来自键盘的I/O和到显示器的I/O。这是有利有弊的，因为当编写和执行8086程序时我们不得不处理DOS环境本身的接口问题，这意味着需要学习一些新的只适用于DOS环境的汇编指令。

最后，与我们开始学习的68K体系结构相比，掌握i86的基本指令集体系结构需要一个更加努力的学习过程。我肯定这其中会有几个比较艰难的时期，但请允许我对揭露这个过程本身有个人偏见。

为了处理这些问题，我们选择了“KISS”方法（Keep It Simple, Stupid），并将我们的目光集中在最终目标上。由于我们想要完成任务的是对现代计算机体系结构有一个相当程度的了解，所以我们将把掌握i86体系结构的汇编程序设计技巧的问题留到其他时间考虑。因而，我们将有区别地把重点放在以下几方面：把学习8086寻址模式和指令作为我们的主要目标，而掌握DOS汇编语言程序设计的基本规则作为第二目标（需要时才学习）。然而，你应该能够从i86和DOS汇编语言程序设计的大量资源中获得我为了清楚起见所省略的任何信息。请继续往下学习！

10.2 8086 CPU的体系结构

图10-1是8086 CPU的一个简化框图。你可能会认为这个图看起来比图7-11所示的68000处理器的框图要复杂得多。尽管这两个图看起来很不相同，但是它们之间有一些相似点。在功

能上，通用寄存器基本对应于68K的数据寄存器。图中右侧标记为*总线接口单元*（bus interface unit, BIU）的寄存器叫做*段寄存器*（segment register），大体上对应于68K的地址寄存器。

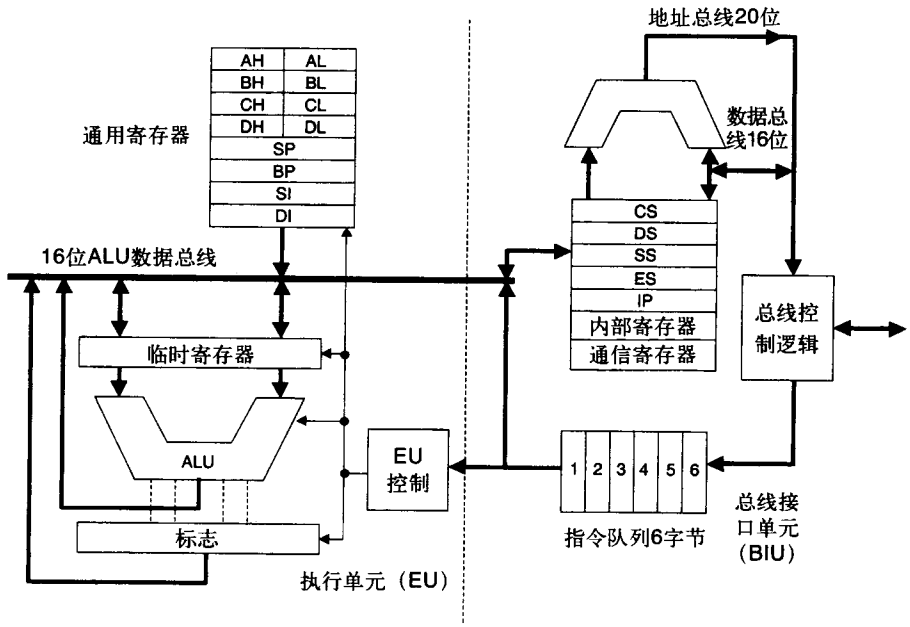


图10-1 Intel 8086 CPU的简化框图。由Tabak[1]提供

8086严格地由两个独立自治的功能模块组成。*执行单元*（execution unit, EU）处理数据的算术和逻辑操作，它拥有一个6字节的*先进先出*（FIFO）指令队列（8088是4字节）。BIU的段寄存器负责访问从存储器中来的指令和操作数。两个功能模块之间的主要连接就是这个指令队列，BIU预先从当前执行指令向前看以保证队列不空，从而让EU对队列中的指令进行译码和操作。

在BIU一侧看起来像木工锯脚架的符号叫做*多路选择器*（multiplexer MUX），它的功能是将地址和数据信息组合成单一的20位外部总线。多路选择（共享）总线使得8086 CPU封装后
267 只有40个引脚，而68000则有64个引脚，这主要是由于23位地址和16位数据带来了额外的引脚。然而，没有免费的午餐。这种多路总线要求使用8086的系统必须在板上有外部逻辑，该外部逻辑在总线周期的前半部分将地址锁存在保持寄存器中，这样做是为了在总线周期的后半部分有一个稳定的地址送给存储器。如果你回忆一下图6-23（一个通用微处理器的时序图），那么我们就可以将8086的总线周期描述为4个“T”状态，标记为T1到T4。如果要在总线周期中包含等待状态，那么可以把它看成是T3状态的延伸。

处理器在T1的下降沿将20位地址送给外部逻辑电路并发出锁存信号，即*地址锁存使能*（address latch enable, $\overline{\text{ALE}}$）。$\overline{\text{ALE}}$用来锁存总线周期的地址部分。数据在T3上升沿时输出并在T3下降沿时读入。20位宽的地址总线使得8086拥有1MB的地址范围。最后，8086对地址字的对齐方式没有任何限制，一个字可以存在于奇数边界也可以存在于偶数边界。BIU管理用于提取字中两个字节和除了性能惩罚之外的额外总线周期，这种行为对软件开发者是透明的。

268 图10-2显示的是8086的程序员模型。

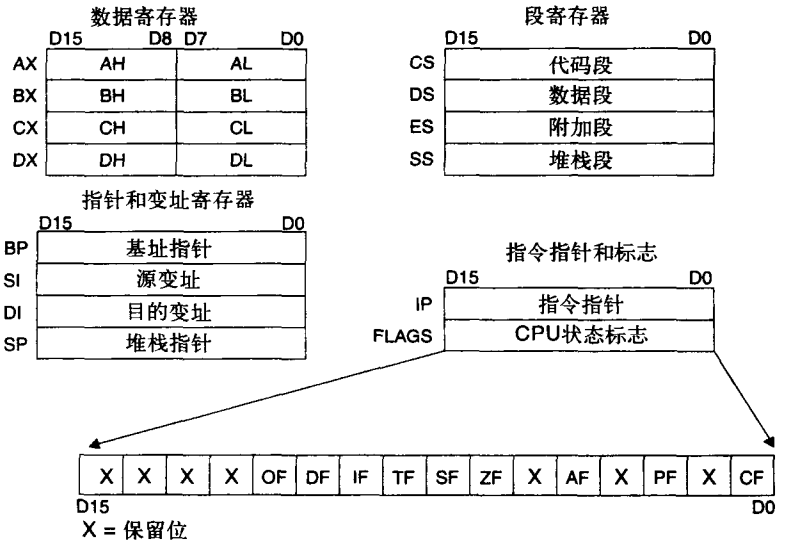


图10-2 8086寄存器组的程序员模型

10.3 数据寄存器、变址寄存器和指针寄存器

在8086中，有8个16位的通用寄存器用来进行算术操作和逻辑操作。此外，对于8位操作，4个数据寄存器AX、BX、CX和DX可以进一步分成高字节和低字节两个寄存器，这取决于字节存储在寄存器中的什么位置。因此，对于字节操作，寄存器可单独进行寻址。例如，**MOV AL, 6D**指令将十六进制的立即数6D放入寄存器AL。是的，我知道，这是后退。

而且，当存储在16位寄存器中的数据被转存到存储器中时，它们将按照与寄存器中相反的次序进行存储。因此，如果<AX> = 109C，那么该数据就会被写到存储器的1000和1001的地址中，存储器中的字节顺序将为：

<1000> = 9C, <1001> = 10 {低端字节序！还记得吗？}

另外，这些数据寄存器不像68K中D0到D7那样完全通用。AX寄存器及其两个8位寄存器AH和AL都是*累加器*（accumulator）。累加器是一个用于算术运算和逻辑运算的寄存器。当你使用乘法（**MUL**）指令和除法（**DIV**）指令时，必须使用AX寄存器。例如，下面的代码片段：

```
MOV AL,10
MOV DH,25
MUL DH
```

可以将AL寄存器中的内容和DH寄存器中的内容做一个8位乘8位的乘法，并把16位的结果值存储在AX寄存器中。注意在乘法指令中没有必要指明AL寄存器，因为字节乘法中必须使用它。对于16位的操作数，结果则存在DX:AX寄存器对中，其中高位字存在DX寄存器中而低位字存在AX寄存器中。例如：

```
MOV AX,0300
MOV DX,0400
MUL DX
```

将产生十进制的结果：<DX:AX> = 0001:D4C0 = 120 000

这里有几个值得注意的地方：

- 被执行指令的类型（字节或者字）由所使用的寄存器及操作数的大小决定。

- 数字被认为是字面值，不用任何特别的符号，比如68000汇编器中使用的“#”符号。
- 在某些情况下，16位寄存器可以组合在一起形成32位宽的寄存器。
- 尽管在这个例子中没有显示，但十六进制数是由紧跟其后的“h”所指示的。而且，以A到F开头的十六进制数需要在前面加一个0，以免被认为是标号。

269

– 0AC55h = 十六进制数AC55

– AC55h = 标号“AC55h”

BX寄存器可以用作16位偏移地址指针，下面的代码片段将值0AAh装载到了绝对存储地址1000Ah处。

```
MOV   AX, 1000h
MOV   DS, AX
MOV   BX, 000Ah
MOV   [BX], 0AAh
```

从这个例子我们可以看到：

- 段寄存器DS必须从一个寄存器进行装载，而不能以一个立即数值进行装载。
- 将BX寄存器放在括号中就将有效寻址模式改为了寄存器间接寻址。完全的存储器装载地址是[DS:BX]。注意DS寄存器没有指明，是隐含的。
- DS寄存器是数据移动操作默认使用的寄存器，就像CS寄存器用于引用指令一样。但是，可以通过明确指明使用的段寄存器来强行修正默认寄存器。例如，下面的代码片段忽略了DS段寄存器，并强制指令使用[ES:BX]寄存器来操作。

```
MOV   AX, 1000h
MOV   CX, 2000h
MOV   DS, AX
MOV   ES, CX
MOV   BX, 000Ah
ES:MOV w.[BX], 055h
```

这里也要注意“w.”的用法。这是显式地告诉指令把文字解释为一个字值——0055h。否则，汇编器可能会把它解释为一个字节值，因为它可表示为“055h”。

CX寄存器被用作计数寄存器，用于循环、移位、重复和计数操作。下面的小段代码说明了CX寄存器的作用。

```
          MOV CX, 5
myLoop:   NOP
          LOOP myLoop
```

LOOP指令的功能与68K语言中的**DBcc**指令相似。但是，**LOOP**指令只能使用CX寄存器作为循环计数器，而**DBcc**指令则可以使用任意数据寄存器作为循环计数器。在上面那段代码中，NOP指令将被执行5次。每次通过循环，CX寄存器都会自动递减，当<CX> = 0时循环指令将会停止执行。注意标号“myLoop”是以冒号“:”结尾的。8086汇编器需要一个冒号来指示这是一个标号。标号也可以放在指令或数据的上一行：

270

```
myLoop: NOP                    myLoop:
                                  NOP
```

这两个是等价的。

DX寄存器是唯一可以用来指定I/O地址的寄存器。由于8086只能寻址64K的I/O位置，所以不要求I/O具有空白区段。下面的代码片段读取了地址为A43Eh的I/O端口，并将数据放入了AX寄存器。

```
MOV  DX,0A43Eh
IN   AX,DX
```

与68K不同，i86家族将I/O作为一个拥有自己总线信号集和定时器的独立存储空间来处理。

DX寄存器也可以用作16位和32位的乘法和除法操作。像我们在前面那个例子中看到的一样，当两个16位数相乘产生一个32位的结果时，DX寄存器与AX寄存器将连在一块使用。类似地，当除数是16位时这两个寄存器会被连在一起用来形成一个32位的被除数。

```
MOV DX,0200
MOV AX,0000
MOV CX,4000
DIV CX
```

这里将32位数字的被除数00C80000h放入了DX:AX，除数为CX寄存器中的0FA0h。两数相除后所得的商0CCCh存在AX中，余数0C80h存在DX中。

目的变址寄存器DI和源变址寄存器SI用于数据移动和字符串操作。在寻址源操作数和目的操作数的时候，每个寄存器都有专门的用途。DI指针和SI指针也是不同的，因为在进行字符串操作的时候，SI寄存器与DS段寄存器配对使用，DI与ES配对使用。当进行非字符串操作的时候，它们都与DS连用。

这看起来可能很奇怪，但这是合情合理的事。例如，当你想要在两个独立的大于64K的存储区域间拷贝一个字符串时，使用两个不同的段寄存器自然就给了你最大可能的存储器区域。下面的代码片段从DS:SI最初所指的存储地址拷贝了5个字节到ES:DI所指的存储地址。

```
            MOV CX,0005
            MOV AX,1000h
            MOV BX,2000h
            MOV DS,AX
            MOV ES,BX
            MOV DI,200h
            MOV SI,100h
myLoop:     MOVSB
            LOOP myLoop
```

这个程序将DS段寄存器初始化为1000h，将SI段寄存器初始化为100h。因此将要拷贝的字符串定位在地址<1000:0100>，或者物理存储地址10100h。ES段寄存器被初始化为2000h，DI寄存器被初始化为0200h，因此目的数在存储器中的物理地址是20200h。CX寄存器被初始化为循环计数5，指令LOOP使得**MOVSB**（移动字符串字节）指令执行5次。**MOVSB**指令每运行一次，DI和SI寄存器的内容都会自动增加1个字节。 271

不管你喜不喜欢，这些都是功能强大且简洁的指令。

基址指针和堆栈指针（BP和SP）通用寄存器与BIU中的堆栈段（SS）寄存器一起使用，分别指向堆栈底和堆栈顶。由于系统栈由一个不同的段寄存器管理，所以需要额外的偏移地址寄存器来指示SS所指向的区域中的地址。将SP寄存器看作就是这个堆栈指针，而BP寄存器是一个通用的存储器指针，指向SS段寄存器所指向的存储区域。BP被高级语言用来为基于堆栈的操作提供帮助，例如参数传递和局部变量的操作。从某种意义上说，当高级语言为68K而编译时，SS和BP的结合取代了局部帧指针（frame pointer）（通常是寄存器A6）。

所有基于堆栈的指令（**POP**、**POPA**、**POPF**、**PUSH**、**PUSHA**和**PUSHF**）都使用堆栈指针（SP）寄存器。SP寄存器总是被用作从堆栈段（SS）寄存器所指向的地址到当前栈地址的一个偏移值。

这些指针和变址寄存器与68K中对应的寄存器有一个重要的不同。就如上面的寄存器定义所说的，这些寄存器可以与BIU中的段寄存器连用以形成操作数的物理存储地址。我们很快就会看到这一点。

10.4 标志寄存器

标志寄存器中的一些位与68K状态寄存器中的位有相似的定义，其他的位则不同。而且，这些标志的置位和复位比68K体系结构中的更严格。一条指令执行完后，这些标志可能会被置位为1，清空或者复位为0，也可能未变或者未定义。未定义意味着一条指令执行前的标志值可能不被保留，指令执行后的标志值不能预知[2]。

- 位0：进位标志（CF），加法的高位进位操作或者减法的高位借位操作都会将其置位；否则清零。
- 位1：保留的。
- 位2：奇偶标志（PF），如果一个结果的低8位中包含偶数个1（奇偶性为偶数）就置位；否则清零（奇偶性为奇数）。
- 位3：保留的。
- 位4：辅助进位标志（AF），当通用寄存器AL中的低4位有进位或借位时置位；否则清零。
- 位5：保留的。
- 位6：零标志（ZF），如果结果为0则置位；否则清零。
- 位7：符号标志（SF），置为与结果最高位相等的值。当MSB＝0时置为0（结果为正数），当MSB = 1时则置为1（结果为负数）。
- 位8：陷阱标志（TF），当TF标志被置为1的时候，每条指令执行后都会有一个陷阱中断发生。TF标志是在处理器状态标志被压栈后自动被陷阱中断清零的。当从中断指令（IRET）返回时，陷阱服务程序能通过将标志退出而继续执行陷阱。因此，这个标志实现了单步机制的调试。

272

- 位9：中断使能标志（IF），当置为1的时候，就允许可屏蔽的或者低优先级的中断，而且可能中断处理器。当中断发生时，CPU将控制转移到中断向量（指针）指向的存储地址。
- 位10：方向标志（DF），DF标志的置位使得字符串指令自动递增相关的变址寄存器。标志的清零则使得指令自动递减寄存器。
- 位11：溢出标志（OF），如果带符号的结果不能在目的操作数的位数范围内表示则置位；否则清零。
- 位12~15：保留的。

10.5 段寄存器

这4个16位段寄存器是BIU的一部分，它们存放着操作数地址的段（页）值。这些寄存器（CS、DS、ES和SS）定义了存储器的段，而这些段对于代码或取指令（CS）、数据读写（DS和ES）以及基于栈的操作（SS）来说是可立即寻址的。

10.6 指令指针（IP）

指令指针寄存器包含下一条即将执行的顺序指令的偏移地址。因此，其功能与68K中的程序计数寄存器相同。不能直接修改IP寄存器。像PC一样，引起转移、跳转和子程序调用的非顺序指令将会改变IP寄存器的值。

这些寄存器的描述逐渐为我们引入了一种新的存储器寻址模式，称为段-偏移寻址。段-偏移寻址在很多方面与页寻址相似，但与我们在这本书前面所讨论的页寻址方法并不完全一

样。段寄存器用来指向任意一个在20位地址空间中存在的64K的16字节边界（称为段(paragraph)）。图10-3说明了地址是怎么形成的。 273

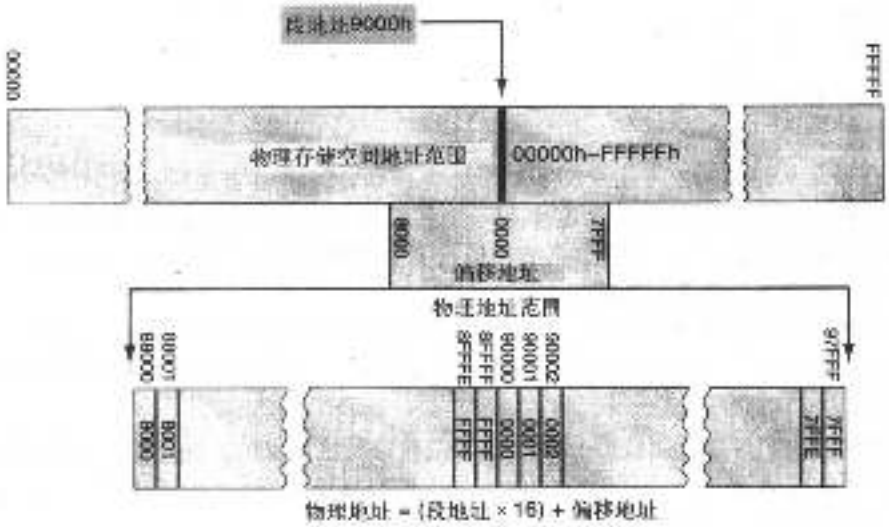


图10-3 基于段和偏移模型的存储器寻址

从这个模型很容易看出，相同的存储地址可以用很多种不同的方式指定。这在某些方面与68K寻址模式引起的混淆问题并没有太大的区别，尽管对你来说在这一点上68K寻址模式可能看起来更直截了当一些。无论如何，20位的地址值都是通过取得相应段寄存器的16位地址值并将其算术左移4位而得到的。由于每一次左移都相当于乘以2，4次左移就是将地址乘以16，这就得到了如图10-3所示的段边界的基地址。偏移值是指以段寄存器指定的段边界为中心的正位移值或负位移值。图10-3说明从一个值为9000h的段寄存器能够访问从88000h到97FFFh的物理地址范围。从0000h到7FFFh的偏移地址表示正位移（指向高地址），而从0FFFFh到8000h的地址则表示负位移。

一旦段边界由段寄存器确定后，通过符号扩展得到的偏移值（或者是字面值，或者是寄存器内容）就能与段寄存器的移位值相加而形成20位全地址，如图10-4所示。

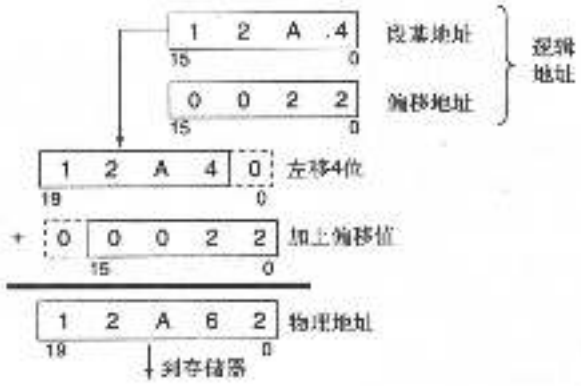


图10-4 8086体系结构中从逻辑地址到物理地址的转换

虽然存储操作数的物理地址是20位（即1MB）的，但是I/O空间的地址范围却是16位（即64K）的。当寻址I/O设备时高4位没有被使用。

你应该开始明白的一点是对于大多数用途来说，存储器中的物理地址如何实际工作并没有关系。大多数你将使用的调试工具都将地址表示成了段：偏移形式，所以你没有必要去试图计算存储器中的物理地址。因此，如果给你的地址是1000:0055，那么你可以只关注从1000h开始的段中偏移值为55h的地址。如果必须使用物理存储器，那么你可以将这个地址转换为10055h。但是这只是例外，而不是惯例。

段寄存器

正如在前面那些例子中看到的，在BIU中有4个段寄存器。它们是：

- 代码段（CS）寄存器
- 数据段（DS）寄存器
- 堆栈段（SS）寄存器
- 附加段（ES）寄存器

274 CS寄存器是从存储器中取指令的基址指针寄存器。DS寄存器是所有数据操作的默认段寄存器。然而你已看到，默认寄存器能够用一个汇编寄存器作为指令操作码前缀来强制修改。很明显这与68K是不相同的，这种体系结构想有一个清晰的划分，区分哪些是取来作为指令的数据，哪些是取来作为操作的数据。

堆栈段寄存器所起的作用类似于在68K体系结构中与SP寄存器一起使用时的堆栈指针寄存器。BIU中的最后一个寄存器是附加段寄存器即ES，这个寄存器为基于字符串的操作提供了第二个段指针。

从这些讨论我们可以看出段寄存器的行为是非常严格定义的。由于它们是BIU的一部分，所以不能用来做算术操作，而且它们的内容只能由寄存器到寄存器的传输来修改。按照CPU清楚地分为EU和BIU，这是很有意义的。

通过这些代码例子，在某些方面我们已经超前学习了，目的是在专心和系统地学习8086指令和有效寻址模式之前让你对这种体系结构的工作方式有一个感性认识。

10.7 存储器寻址模式

8086体系结构提供了8种寻址模式。前两种模式对存储在内部寄存器中的值和一部分指令值（立即值）进行操作。这两种模式是：

1. *寄存器操作数模式*：操作数存放在一个8位或者16位寄存器中。寄存器操作数模式的指令例子如下：

```
MOV   AX,DX
MOV   AL,BH
INC   AX
```

2. *立即操作数模式*：操作数是指令的一部分。立即操作数模式的例子为：

```
MOV   AX,0AC10h
ADD   AL,0AAh
```

这里没有用“#”之类的符号作为前缀来表明操作数是立即数。

下面的6种寻址模式与存储操作数（即存储在存储器中的数据值）一起使用。这些有效的寻址模式用来计算和构造偏移地址，这些偏移地址与段寄存器一起用来创建用于查找或写存储数据的实际物理地址。物理存储地址是由包含在段寄存器中的逻辑地址和偏移值合成得到的，其中的偏移值不一定包含在寄存器中。段寄存器一般是由要执行操作的类型隐含地选择的。因此，指令的获取与CS寄存器值相关，数据的读写与DS寄存器相关，堆栈操作与SP寄存器相关，而字符串操作与ES寄存器相关。寄存器也可以通过在指令头规定一个期望的寄存器来强制修正。例如：

```
ES:MOV AX,[0005h]
```

将取得ES:0005而不是DS:0005处的数据。有效地址（偏移地址）可以通过将以下三个地址元素相加而得到：

a. 一个作为指令一部分的8位或者16位的立即偏移值。

b. 一个包含在BX或者BP寄存器中的基址寄存器值。

c. 一个存储在DI或者SI寄存器中的变址值。 275

3. *直接模式*：包含的存储偏移值是一个8位或者16位的正位移值。

与68K不同，地址的偏移部分总是一个正数，因为位移是从段的起始位置开始算的。直接模式与68K的绝对寻址模式最相近，但是不同的地方在于总是需要隐含的段寄存器来构成完整的物理地址。例如：

```
MOV   AX,[00AAh]
```

将数据从DS:00AA拷贝到AX寄存器，而

```
CS:MOV    DX,[0FCh]
```

则将数据从CS:00FC拷贝到DX寄存器。

注意方括号是用来符号化存储指令的，没有括号则表示是立即数。而且，MOV指令不允许两个存储操作数。指令

```
MOV   [00AAh],[1000h]
```

是非法的。与68K的MOVE指令不同，这里只允许一个存储操作数。

4. *寄存器间接模式*：操作数偏移被包含在以下寄存器之一中：

- BP
- BX
- DI
- SI

方括号用来指示间接寻址。下面的代码片段将值55h写入到存储地址DS:0100。

```
MOV   BX,100h
MOV   AL,55h
MOV   [BX],AL
```

5. *基址模式*：存储操作数是基址寄存器BX或BP的内容与8位或16位位移值的和。下面的代码片段将值0AAh写入到存储地址DS:0104。

```
MOV   BX,100h
MOV   AL,0AAh
MOV   [BX]4,AL
```

基址模式指令也可以写成：**MOV [BX - 4], AL**。

因为位移既可以是正数也可以是负数，所以上面的指令与**MOV[BX 0FFFCh],AL**等价。 276

6. *变址模式*：存储操作数是变址寄存器DI或SI的内容与8位或16位位移值的和。下面的代码片段将值0AAh写入到存储地址ES:0104。

```
MOV   DI,100h
MOV   AL,0AAh
MOV   [DI]4,AL
```

上面的代码说明了使用变址寄存器与基址寄存器的另一个区别。DI寄存器默认以ES寄存器作为它的源地址段，尽管ES寄存器也可以通过在指令前面指定一个段而被强制修改。因此，**DS:MOV [DI+4], AL**将强制指令使用DS寄存器而不是ES寄存器。

7. *基址变址模式*：存储操作数的偏移值是基址寄存器BP或BX的内容与变址寄存器DI或SI的内容之和。在变址模式中，DI寄存器通常与ES寄存器配对使用，但是当DI寄存器在这种模式下使用时，DS寄存器是默认的段寄存器。下面的代码片段将值0AAh写入到存储地址DS:0200h。

```
MOV   DX,1000h
MOV   DS,DX
MOV   DI,100h
MOV   BX,DI
MOV   AL,0AAh
MOV   [DI+BX],AL
```

指令**MOV [DI+BX],AL**也可以写成：**MOV [DI][BX],AL**。

8. 带位移的基址变址模式：存储操作数的偏移值是基址寄存器BP或BX的内容、变址寄存器DI或SI的内容以及一个8位或16位的位移值三者的和。在变址方式中，DI寄存器通常与ES寄存器配对使用，但是当DI寄存器在这种方式下使用时，DS寄存器是默认的段寄存器。下面的代码片段将值0AAh写入到存储地址DS:0204h。

```
MOV   DX,1000h
MOV   DS,DX
MOV   DI,100h
MOV   BX,DI
MOV   AL,0AAh
MOV   [DI+BX]4,AL
```

指令**MOV [DI+BX]4, AL**也可以写成：**MOV [DI][BX]4, AL**或者**MOV [DI BX 4], AL**。

偏移值的计算可能会导致一个非常有趣而微妙的代码错误。请查看下面的这段代码：

```
MOV   DX,1000h
MOV   DS,DX
MOV   BX,07000h
277   MOV   DI,0FF0h
MOV   AL,0AAh
MOV   [DI+BX+0Fh],AL
```

这段代码汇编时没有错误，而且将值0AAh写入了DS:7FFF。物理地址是17FFFh。现在，查看另一段代码：

```
MOV   DX,1000h
MOV   DS,DX
MOV   BX,07000h
MOV   DI,0FF0h
MOV   AL,0AAh
MOV   [DI+BX+10h],AL
```

这段代码汇编时没有错误，而且将值0AAh写入了DS:8000。然而，物理地址却不是存储器中的下一个物理位置18000h，它实际上是08000h。我们已经绕过了偏移地址并从存储地址17FFFh转移到了8000h。如果我们已经在一个循环中，并正要写一个连续值到存储器中，那我们可能就得调试一个有趣的问题了。

可以将存储器寻址模式总结如下。假定项#1、#2或者#3是可选的，而且要求你至少有一种表示形式来指定偏移值。这看起来很有道理。但是，我们要记住，除非默认段寄存器被强制修改，否则DI寄存器将与ES寄存器配对使用来决定物理地址，而其他三个寄存器则使用DS段寄存器。不过，如果DI寄存器与BX或BP寄存器连用，那么DS就是默认的段寄存器。

#1		#2		#3
BX 或 BP	+	DI 或 SI	+	位移

10.8 x86指令格式

一条8086指令可能短至一个字节也可能长至几个字节。所有的汇编语言指令都遵循如图10-5所示的格式。每个域解释如下：

1. 指令前缀：某些字符串操作指令在以REP、REPE、REPZ、REPNE以及REPNZ作为前缀时可以重复执行。当然也有前缀项用于指定总线锁定（BUS LOCK）信号到外部接口。

2. 段超越前缀：为了强制修正计算存储操作数地址的默认段寄存器，段超越字节前缀被添加到指令的开始处。只可以使用一个段超越前缀。段超越的格式如图10-6所示。



图10-5 8086指令格式

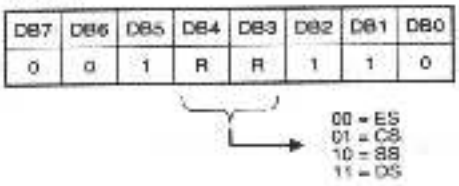


图10-6 段超越前缀格式

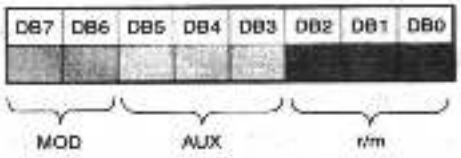


图10-7 操作数地址字节的格式

3. 操作码：操作码域指定指令的机器语言操作代码，大多数操作码的长度是一个字节，尽管有些指令需要第二个操作码字节。

4. 操作数地址：操作数地址字节有点棘手。这个字节的格式如图10-7所示，一共有三个域，分别标为mod、aux和r/m。操作数地址字节控制指令的寻址形式。

mod域的定义如下：

DB7	DB6	描　述
1	1	r/m被当作寄存器域
0	0	DISP域=0；disp-low和disp-high为空
0	1	DISP域=扩展到16位的符号disp-low，disp-high为空
1	0	DISP域=disp-high:disp-low

mod（修饰符）域与r/m域还用可以决定r/m域是怎样被解释的。如果mod=11，那么r/m域被解释为一个寄存器操作数。对于一个存储操作数，mod域则用来决定该存储操作数是直接寻址还是间接寻址。对于间接寻址的操作数，mod域用来指定指令中的字节数和位移。

寄存器/存储器（r/m）域指定操作数是一个寄存器还是一个存储地址。操作数的类型是由mod域指定的，如果mod位是11，那么r/m域则被当作是寄存器标识。如果mod域是00、10或者01，那么r/m域则指定一种我们以前讨论过的有效寻址模式。r/m域值可以总结如下：

DB2	DB1	DB0	有效地址
0	0	0	[BX] + [SI] + DISP
0	0	1	[BX] + [DI] + DISP
0	1	0	[BP] + [SI] + DISP
0	1	1	[BP] + [DI] + DISP
1	0	0	[SI] + DISP
1	0	1	[DI] + DISP
1	1	0	[BP] + DISP（如果mod=00，那么EA=disp-high:disp-low）
1	1	1	[BX] + DISP

当r/m被用来指定寄存器时，位码如下：

DB2	DB1	DB0	字	字节
0	0	0	AX	AL
0	0	1	CX	CL
0	1	0	DX	DL
0	1	1	BX	BL
1	0	0	SP	AH
1	0	1	BP	CH
1	1	0	SI	DH
1	1	1	DI	BH

279

从上表可以看出寄存器的位码是怎样随着指令中操作数的大小而改变的。操作数的大小是由指令的操作码这部分决定的。我们马上就会看到这是怎么工作的。

辅助域（aux）有两种用途。某些指令在需要的时候可能要求有一个操作码扩展域，这种扩展域就包含在aux域中。另外，当指令需要指定第二个寄存器操作数时，辅助域就用于根据上表来指定第二个寄存器。

5. *位移*：位移值是一个在有效地址计算中要加到偏移部分的8位或16位数。

6. *立即数*：立即数域包含多至2字节的立即数。

和你看到的一样，8086指令格式比68K指令格式确实要复杂一些。很多复杂性可以归根于段:偏移的地址格式，但这并不是全部的原因。我们应该将8086体系结构作为一个整体来考虑，以便了解位模式是怎样用来创建微代码中的路径的，这决定了指令的状态机路径。最后一个要考虑的难题是指令的操作码部分。

回忆一下，在68K体系结构中，指令的操作码部分可以很容易地被剖析为位的固定模式：一个寄存器域、一个模式域以及一个有效地址域。在8086体系结构中，区别没有那么明显，控制也更加分散。让我们考虑一下**MOV**指令的可能操作码。下面的表中列举了可能的指令代码。操作码列中以斜杠“/”开头的值是操作数地址字节中的辅助域的值。操作码列中紧跟在加号“+”后的值是要加到操作码基本值上的寄存器码。

格　式	操作码	描　述
MOV r/m8, r8	88 /r	将存储在寄存器/r中的字节值拷贝到寄存器或存储器字节r/m8
MOV r/m16, r16	89 /r	将存储在寄存器/r中的字值拷贝到寄存器或存储器字r/m16
MOV r8, r/m8	8A /r	将存储在r/m8中的字节值拷贝到字节寄存器/r中
MOV r16, r/m16	8B /r	将存储在寄存器或存储器字r/m16中的值拷贝到字寄存器/r中
MOV r/m16, sreg	8C /sr	将段寄存器/sr拷贝到寄存器或存储器字r/m16
MOV sreg, r/m16	8E /sr	将寄存器或存储器字r/m16拷贝到段寄存器
MOV AL, moffs8	A0	将存储在存储器中段:moffs8处的字节值拷贝到AL寄存器
MOV AX, moffs16	A1	将存储在存储器中段:moffs16处的字值拷贝到AX寄存器
MOV moffs8, AL	A2	将AL寄存器中的内容拷贝到存储器中地址为段:moffs8的字节处
MOV moffs16, AX	A3	将AX寄存器中的内容拷贝到存储器中地址为段:moffs16的字节处
MOV r8, imm8	B0 + rb	将立即数字节imm8拷贝到寄存器rb
MOV r16, imm16	B8 + rw	将立即数字imm16拷贝到寄存器rw
MOV r/m8, imm8	C6 /0	将立即数字节imm8拷贝到寄存器或存储器字节r/m8
MOV r/m16, imm16	C7 /0	将立即数字imm16拷贝到寄存器或存储器字r/m16

280

通过上面的操作码，我们可以看出无论何时只要字节操作被指定，那么操作码中的最低有

效位就将为0。相反，无论何时只要字操作被指定，最低有效位就为1。如果源或者目的是一个段寄存器，那么在操作数地址字节aux域中的相应段寄存器DS、SS或者ES的代码将分别是011、010或者000。这些与用于段超越位的代码是相同的。CS寄存器不能直接通过MOV指令来修改。

这里是为了轻松吸收几页内容而做的大量努力，所以让我们把它放在一起，通过观察一些实际指令的位模式来判断在实际上它们是否与我们所预言的一致。

助记指令	机器码	mod	aux	r/m
MOV AX, BX	8B C3	11	000	011

操作码8Bh告诉我们这是一条形式为MOV r/16, r/m16的MOV指令。目的寄存器是字寄存器AX。而且，寄存器在aux域中被指定且它的值为000，对应于AX寄存器。mod域是11，这意味着r/m16域是一个寄存器值。r/m域的值为011，这与BX寄存器相匹配。因此，我们对指令的分析从操作码的适当形式开始，这将会把我们引至操作数字节的适当形式。

上面那个例子是非常直接的，下面我们将往前进一小步。让我们增加段超越和存储偏移。

助记指令	机器码	mod	aux	r/m
ES:MOV [100h], CX	26 89 0E 00 01	00	001	110

段超越字节26h指明了ES寄存器，操作码89h告诉我们这是形式为**MOV r/m16, r16**的指令。aux域告诉我们源寄存器是CX（001），而mod域告诉我们寄存器/存储器地址是一个存储器位置，只是在计算偏移地址时并不需要位移值。r/m域值110意味着存储偏移值为disp-high:disp-low，这在存储器中被存储为00 01（低端字节序）。

最后一个例子，我们将把它们合起来。

助记指令	机器码	mod	aux	r/m
CS:MOV [BX+DI+0AAh], 55h	2E C6 81 AA 00 55	10	000	001

能明白该指令吗？段超越字节意味着强制指定的段是CS（01）。操作码C6h告诉我们指令是如下所示的形式：

```
MOV r/m8,imm8
```

操作数地址字节给出mod值10，这告诉我们在计算偏移地址时有一个必须使用的位移域，而且它的形式是disp-high:disp-low，我们可以从后面的两个字节AA 00看到。r/m域值001告诉 281
我们计算地址的最终形式为：[BX + DI + DISP]。和操作码的格式所显示的一样，指令的最后一个字节55h是8位立即数imm8。

如果我们将指令修改如下：

```
CS:MOV      [BX+DI+0AAh],5555h
```

指令代码就会变为2E C7 81 AA 00 55 55。操作码由C6变为C7，意味着是一个字操作，而且该立即数域现在是两个字节长。现在整个指令的长度是7个字节，所以我们就能明白为什么8086体系结构必须允许存储器的不对齐访问，这是因为如果当前指令从一个字边界开始，那么下一条指令将会从奇数边界开始。

10.9 8086指令集总结

虽然详细地解释所有8086系列指令已经超出了本书的范围，但是我们可以通过学习并使

用每一类中的代表性指令来得到对所有指令的一些认识。而且，其中一些指令将使我们的视野扩展到整体的8086系列体系结构。

我们可以把最原始的x86指令集划分为以下的指令类：

- 数据传送指令
- 算术指令
- 逻辑指令
- 字符串操作指令
- 控制转移指令

这些类指令中的大多数对你来说都比较熟悉了，另外一些则比较生疏。68K系列指令集中没有用于循环控制或者字符串操作的专用指令，很难估计这是否代表体系结构的一个缺点。很明显，在68K体系结构中你也可以操作字符串和构成软件循环。现在让我们看看每一类中的一些代表性指令。

10.10 数据传送指令

就像68K系列中的MOVE指令一样，MOV指令可能是指令集中使用最频繁的指令。在这一章中我们已经较多地接触了MOV指令，可以把它作为了解x86指令集体系结构如何运作的一条原型指令。我们可以将数据转移类中的指令总结为下面的表格[6]。注意并不是所有的指令都列举出来了，一些指令还有其他一些具有独特助记符的变型。这个表是用来总结最常用类型的。

助记符(Intel)	指　令	描　述
MOV	移动	从一个源操作数拷贝数据到一个目的操作数
PUSH	压栈	为一个栈中操作数创建存储空间并将操作数拷贝到栈里
POP	退栈	拷贝栈顶元素到存储器或寄存器
XCHG	交换	交换两个操作数
IN	输入	从I/O空间地址读数据
OUT	输出	写数据到I/O空间地址
XLAT	解释	将存储器中字节的偏移地址转换为字节值
LEA	载入有效地址	计算一个数据元素的有效地址偏移并将该值载入一个寄存器
LDS	将段载入DS，偏移载入寄存器	读取在存储器中以一个32位双字存储的全地址指针，将段部分存在DS中，偏移部分存在一个寄存器中
LES	将段载入ES，偏移载入寄存器	除了存储器指针的段部分是载入ES而不是DS之外，与LDS相同
LAHF	标志载入AH	将处理器状态寄存器的标志部分拷贝到AH寄存器
SAHF	AH存入标志	将AH寄存器中的内容拷贝到处理器状态寄存器的标志部分
PUSHF	标志压栈	在栈上创建空间并将处理器状态寄存器中的标志部分压栈
POPF	标志退栈	将栈顶的数据拷贝到处理器状态寄存器的标志部分并从栈上移除存储空间

检查一下数据传送指令集，我们发现除了XLAT、IN和OUT指令之外其他指令都在68K指令集中有类似的指令。

282

10.11 算术指令

下面这张表总结了8086的算术指令。

助记符(Intel)	指　令	描　述
ADD	加	两个整数或无符号数相加
ADC	带进位加	两个整数或无符号数与进位标志的内容一起相加
INC	递增	寄存器内容或存储器值增加1
AAA	加后ASCII调整AL	将两个非压缩BCD数的无符号二进制和转换为等价的非压缩十进制数
DAA	十进制加调整	将加法结果的8位无符号值转换为等价的BCD数
SUB	减	两个整数或无符号数相减
SBB	带借位减	减去一个整数或一个无符号数，以及一个相同类型数字的进位标志的内容
DEC	递减	一个寄存器或存储器位置的内容减1
NEG	取补	将一个寄存器或存储器的内容替换为它的补码
CMP	比较	两个整数或者无符号数相减并设置相应的标志，但不保存减法的结果。源操作数和目的操作数都不会改变
AAS	减后ASCII调整	将两个非压缩BCD数的无符号二进制差转换为等价的非压缩十进制数
DAS	减后十进制调整	将减法结果的8位无符号值转换为等价的BCD数
MUL	乘	两个无符号数相乘
IMUL	整数乘	两个带符号数相乘
AAM	乘后ASCII调整	将两个非压缩BCD数的无符号二进制积转换为等价的非压缩十进制数
DIV	除	两个无符号数相除
IDIV	整数除	两个带符号数相除
AAD	除后ASCII调整	将两个非压缩BCD数的无符号二进制商转换为等价的非压缩十进制数
CWD	字转换为双精度	将一个16位的整数转换为符号扩展的32位整数
CBW	字节转换为字	将一个8位的整数转换为符号扩展的16位整数

283

除了4个用于将BCD数字转换为易转化为等价ASCII码的ASCII调整指令之外，上面所有的指令都在68K指令集中有类似的指令。

10.12 逻辑指令

下面这张表总结了8086的逻辑指令。

助记符(Intel)	指　令	描　述
NOT	取反	寄存器或存储器操作数取反
SHL	逻辑左移	SHL和SAL将操作数各位左移，空位用0填充，高位移入到CF中
SAL	算术左移	
SHR	逻辑右移	将操作数的各位右移，空位用0填充，低位移入到CF中
SAR	算术右移	将操作数的各位右移，空位用原来的最高位填充，低位移入到CF中
ROL	循环左移	将操作数的各位循环左移，高位移入到CF和低位中
ROR	循环右移	将操作数的各位循环右移，低位移入到CF和高位中

284

（续）

助记符(Intel)	指　令	描　述
RCL	通过进位位循环左移	将操作数的各位循环左移，高位移入到CF中，而CF的内容移入到低位中
RCR	通过进位位循环右移	将操作数的各位循环右移，低位移入到CF中，而CF的内容移入到高位中
AND	按位与	计算两个操作数的按位逻辑与
TEST	逻辑比较	判断一个操作数的特定位是否为1。结果不保存，只有标志受到影响
OR	按位或	计算两个操作数的按位逻辑或
XOR	异或	计算两个操作数的按位逻辑异或

可以看出，逻辑指令类与68K系列中的对应指令非常接近。

10.13 字符串操作

这一组指令在68K体系结构中没有对应的指令，它们提供了一组功能强大而简洁的字符串操作。下面的表格总结了8086的字符串操作指令。

助记符（Intel）	指　令	描　述
REP	重复	重复执行一条简单的字符串指令
MOVS	移动一个字符串成员	从一个地址拷贝字符串的一个字节（MOVSB）或字（MOVE SW）到另一个地址
CMPS	比较一个字符串成员	将一个字符串的一个字节（CMPSB）或字（CMPSW）与第二个字符串的字节或字进行比较
SCAS	扫描一个字符串寻找成员	将一个字符串的一个字节（SCASB）或字（SCASW）与一个寄存器的值进行比较
LODS	载入字符串成员	拷贝一个字符串中的字节（LODSB）或字（LODSW）到AL（字节）或AX（字）寄存器
STOS	存储字符串成员	拷贝AL（字节）或AX（字）寄存器中的字节（STOSB）或字（STOSW）到一个字符串

重复（**REPEAT**）指令还有几种上面表格中没有列举的变化形式。像68K中的**DBcc**指令一样，指令的重复取决于标志寄存器中其中一位的值。这组指令如何很容易地实现C和C++库中的字符串操作指令是显而易见的。

重复指令是很独特的，因为它并没有作为一个单独的操作码使用，而是放置在需要重复的其他字符串指令前面。因此，

```
REPMOVSB
```

285 将使得MOVSB指令重复CX寄存器中存储的次数。而且，MOVS指令自动增加或减少DI和SI寄存器的内容，所以，为了进行一次字符串拷贝操作，你应该做以下几步：

1. 初始化方向标志DF。
2. 初始化计数寄存器CX。
3. 初始化源变址寄存器SI。

4. 初始化目的变址寄存器DI。

5. 执行**REPMOVSB**指令。

将这个与68K指令集中的相应操作相比较，在68K中没有方向标志要设置，但是步骤2~4都需要执行相类似的操作。自动递增和自动增减的寻址模式将与MOVE指令一起使用，所以你应当用下面的指令来模仿MOVSB指令：

MOVE.B (A0)+, (A1)+

不同的是你需要有一条额外的指令例如DBcc作为你的循环控制器。这是否意味着在这种字符串拷贝操作中8086会比68K更好呢？在没有深入地分析前还很难说。因为REPMOVSB是一条相当复杂的指令，有理由认为它比一条更简单的指令占用了更多的时钟周期。来自AMD的186EM[4]处理器需要占用8+8×n个时钟周期来执行这条指令，这里n是指令重复的次数。因此，这条指令在两个存储器位置之间拷贝100个字节时最少需要占用808个时钟周期。然而，为了执行68K的MOVE和DBcc指令对，我们还需要反复地从存储器中取每一条指令，所以存储器取指令的开销在比较中占很大部分。

10.14 控制转移

助记符（Intel）	指　令	描　述
CALL	调用过程	挂起当前指令序列的执行，保存下一条指令的段（如果必要）和偏移，并将执行转移到操作数所指向的指令
JMP	无条件跳转	停止当前指令序列的执行，将控制转移到操作数所指向的指令
RET	从过程返回	与CALL指令连用，RET指令恢复IP寄存器的内容，也可能恢复CS寄存器的内容
JE	若相等则跳转	如果零标志（ZF）置位，控制将被转移到操作数所指向的指令地址。如果ZF被清零，该指令被忽略
JZ	若为0则跳转	
JL	若小于则跳转	如果符号标志（SF）和溢出标志（OF）不一样，那么控制将被转移到操作数所指向的指令地址。如果它们相同，该指令被忽略
JNGE	若非大于等于则跳转	
JB	若低于则跳转	如果进位标志（CF）被置位，控制将被转移到操作数所指向的指令地址。如果CF被清零，该指令被忽略
JNAE	若非高于等于则跳转	
JC	若有进位则跳转	
JBE	若低于等于则跳转	如果进位标志（CF）或者零标志（ZF）被置位，控制将被转移到操作数所指向的指令地址。如果CF和ZF都被清零，该指令被忽略
JNA	若不高于则跳转	
JP	若为奇数则跳转	如果奇偶标志（PF）被置位，控制将被转移到操作数所指向的指令地址。如果PF被清零，该指令被忽略
JPE	若为偶数则跳转	
JO	若溢出则跳转	如果溢出标志（OF）被置位，控制将被转移到操作数所指向的指令地址。如果OF被清零，该指令被忽略
JS	若有符号则跳转	如果符号标志（SF）被置位，控制将被转移到操作数所指向的指令地址。如果SF被清零，该指令被忽略
JNE	若不相等则跳转	如果零标志（ZF）被清零，控制将被转移到操作数所指向的指令地址。如果ZF被置位，该指令被忽略
JNZ	若非零则跳转	

286

（续）

助记符（Intel）	指　令	描　述
JNL	若不小于则跳转	如果符号标志（SF）和溢出标志（OF）一样，那么控制将被转移到操作数所指向的指令地址。如果它们不相同，该指令被忽略
JGE	若大于等于则跳转	
JNLE	若非小于等于则跳转	如果逻辑表达式ZF * (SF XOR OF)取值为真，那么控制将被转移到操作数所指向的指令地址。如果该表达式取值为假，该指令被忽略
JG	若大于则跳转	
JNB	若不低于则跳转	如果进位标志（CF）被清零，控制将被转移到操作数所指向的指令地址。如果CF被置位，该指令被忽略
JAE	若高于等于则跳转	
JNC		
JNBE	若非低于等于则跳转	如果进位标志（CF）或者零标志（ZF）都被清零，控制将被转移到操作数所指向的指令地址。如果其中一个标志被置位，该指令被忽略
JA	若高于则跳转	
JNP	若非奇数则跳转	如果奇偶标志（PF）被清零，控制将被转移到操作数所指向的指令地址。如果PF被置位，该指令被忽略
JO	若为奇数则跳转	
JNO	如果没有溢出则跳转	如果溢出标志（OF）被清零，控制将被转移到操作数所指向的指令地址。如果OF被置位，该指令被忽略
JNS	若无符号则跳转	如果符号标志（SF）被清零，控制将被转移到操作数所指向的指令地址。如果SF被置位，该指令被忽略
LOOP	当CX寄存器非零时循环	重复执行一个指令序列。循环执行的次数存储在CX寄存器中
LOOPZ	为0时循环	重复执行一个指令序列。循环重复的最大次数存储在CX寄存器中。如果零标志（ZF）被置位，循环在CX计数减到0之前停止
LOOPE	相等时循环	
LOOPNZ	非零时循环	重复执行一个指令序列。循环重复的最大次数存储在CX寄存器中。如果零标志（ZF）被清零，循环在CX计数减到0之前停止
LOOPNE	不相等时循环	
JCXZ	CX为0时跳转	如果前面的指令在CX寄存器中留下的是0，那么控制被转移到操作数所指向的指令地址
INT	产生中断	当前指令序列被挂起，处理器状态标志、指令指针（IP）寄存器和CS寄存器压栈。指令从存储在相应中断向量位置的存储地址处接着执行
IRET	中断返回	恢复标志寄存器、IP寄存器和CS寄存器的内容

尽管可能的条件跳转指令的列表很长而且印象深刻，但是你应当注意到大多数助记指令都是同义词并且测试的是相同的状态标志条件。而且，由于存储器寻址使用的是段方法，所以跳转类型（JUMP-type）的指令集还需要进一步的解释。

跳转可以分为两类，依据是跳转的目的地址离跳转指令当前地址的远近。如果打算跳转到CS寄存器当前值所指的存储区域的另一个地址，那么你要执行的是一次段内跳转（intrasegment jump）。相反，如果跳转的目的地址超出了CS指针所指的区域，那么你要执行的是一次段间跳转（intersegment jump）。段间跳转要求CS寄存器也被修改以使跳转指令可以

287

覆盖物理存储器的整个范围。

跳转操作数可以有几种形式。下面是跳转指令中的操作数类型：

- 短跳转标识：一个8位的位移值。该标识所指定的指令地址在跳转指令本身的带符号8位位移范围内。
- 近跳转标识：一个16位的位移值。该标识所指定的指令地址在当前代码段范围内。
- 16位存储器指针（Memptr16）或16位寄存器指针（Regptr16）：一个存储在存储器或寄存器中的16位偏移值。存储在存储器或寄存器中的该值被拷贝到IP寄存器并形成从存储器中待取的下一条指令的偏移部分。CS寄存器中的值不被改变，所以这种类型的操作数只能引起当前代码段范围内的跳转。
- 远跳转标识或32位存储器指针（Memptr32）：跳转操作数的地址是一个32位的立即数。第一个16位被载入IP寄存器的偏移部分，第二个16位被载入CS寄存器。操作数Memptr32也可能被用来指定一个双字长度的间接跳转。也就是说，由Memptr32指定的两个连续的16位存储地址包含跳转指令的IP值和CS值。而且，特定的寄存器对（例如DS和DX）可能配对来指定跳转的CS值和IP值。 288

你需要使用的跳转指令类型通常由汇编器来处理，除非你用汇编指令强制改变了默认值。在这一章的后面我们会详细讨论这个问题。

10.15 8086体系结构的汇编语言程序设计

前面几章所阐述的汇编语言程序设计原理与68K系列中的没有什么区别。然而，尽管原理可能一样，但实现方法却有些区别，因为：

1. 8086与PC的体系结构和操作系统有密切的关系，

2. 分段的存储器体系结构要求我们声明所写程序的类型，并指定汇编器将要生成的操作码的存储模型（memory model）。

为了针对8086处理器编写可以在PC机上自由运行的可执行汇编语言程序，你至少要遵循MS-DOS的规则。这就要求你不能预设代码段寄存器的值，因为在操作系统将程序装入存储器的时候它会初始化这个寄存器。因此，当我们想要在68K环境下编写可重定位的代码时，我们应当使用PC或地址寄存器相对寻址模式。这里，我们允许操作系统指定CS寄存器的初始值。

像Borland的Turbo汇编器（TASM）和微软的MASM之类的汇编器能为你处理许多常规事情。因此，只要遵循规则，你仍可以写出运行良好的汇编语言程序。当然，这些程序可以在任何仍然运行16位兼容PC操作系统的机器上运行。较新的32位版本则有更多的问题，因为在仿真模式下运行老版本的DOS程序可能识别不出老版本的BIOS调用。但是，大多数进行简单I/O控制的汇编语言程序都可以在DOS窗口中顺畅运行。作为一个不信任科技发展的人，我仍然拥有自己信任的运行着老版本MS-DOS的486机器，尽管我现在是在运行着Windows XP的系统中写这本书的。

让我们首先来看看段指令和存储模型问题。通常，有必要明显地指定程序中处理代码、数据和栈的部分。这与你已经看到的相类似。我们使用指令：

- `.code`
- `.stack`
- `.data`

来指示代码中这些段的位置（注意这些命令都以一个句点开头）。例如，如果使用命令：

```
.stack    100h
```

289 你就为这个程序保留了256个字节的栈空间。没有必要指定堆栈本身应该在哪里，因为操作系统会替你管理，而且当载入这个程序时操作系统已经启动并运行了。

.data命令指明程序的数据空间。例如，你的程序中可能有下面这些变量：

```
    .data
var16     dw    0AAAAh
var8      db    55h
initMsg   db    'Hello World',0Ah,0Dh
```

该数据空间声明了三个变量var16、var8和initMsg并初始化了它们。为了能够在程序中使用该数据空间，需要将DS段寄存器初始化为数据段的地址。但是，由于你不知道地址在哪，你需要间接指定：

```
MOV   AX,@data          ;数据段的地址
MOV   DS,AX
```

这里，@data是使得汇编器计算正确的DS段值的保留字。

.code命令指明了代码段的开始位置。当程序被装入存储器的时候，CS寄存器将被初始化为指向该段的起始位置。

除了指明各种程序段驻留在存储器中的什么地方之外，你还得给汇编器和操作系统提供所要求的寻址类型的选择以及程序所需要的存储资源量，这些可以用**.model**命令完成。就像我们所看到的执行段内跳转和段间跳转时需要不同指针类型一样，模型的指定说明了程序的大小和数据空间要求。有效的存储模型是[3]：

- 微模型（tiny）：程序代码和数据都在相同的64K的段中，而且，代码和数据都被定义为near，这意味着它们是通过重载IP寄存器来进行转移的。
- 小模型（small）：程序中的代码都在一个大小为64K的段中，而数据则都在一个单独的64K段中。代码和数据都是near。
- 中模型（medium）：程序中的代码可能大于64K，但是程序数据的大小必须足够小以适应单个64K的段。代码定义为far，意味着段和偏移都必须指定，而数据访问则是near。
- 紧凑模型（compact）：程序中的代码都在一个64K段中，但数据的大小可能超过64K，没有像数组之类的大于64K的单个数据元素。代码访问是near，数据访问是far。
- 大模型（large）：代码和数据空间都可能大于64K。但是没有单个数据数组大于64K。所有的数据和代码访问都是far。
- 巨模型（huge）：代码和数据空间都可能大于64K，而且数据数组也可能大于64K。所有的数据、代码和数组指针都是far。

存储模型的使用是非常重要的，因为它们与PC中编译器使用的存储模型一致。它保证了将要编译链接的汇编语言模块与用高级语言编写的模块相互兼容。

让我们来查看一个可以在你的PC机DOS仿真窗口中运行的程序。

290

```
    .MODEL      small
    .STACK      100h
    .DATA
PrnStrg         db    'Hello World$'          ;将要输出的字符串
    .CODE
Start:
    mov   ax,@data                            ;设置数据段
    mov   ds,ax                               ;初始化数据段寄存器
    mov   dx,OFFSET   PrnStrg                 ;将数据的偏移值装入dx
    mov   ah,09                               ;输出字符串的DOS调用
    int   21h                                 ;DOS调用输出字符串
    mov   ah,4Ch                              ;准备退出
    int   21h                                 ;退出并返回到DOS
    END   Start
```

也许你已经猜到了，这个程序代表的只是8086汇编语言程序设计的最初级步骤。回忆一下第一个你真正编译和运行的C++程序，你可能感动得眼睛都湿润了。

我们正在使用的是“小”（small）存储模型，尽管“微”（tiny）模型也可以运行良好。我们已经为栈空间保留了256个字节，但很难说我们已经使用了所有的栈空间，因为我们没有调用任何子过程。

.data命令定义了数据空间，而且我们定义了一个字节字符串“Hello World$”。符号$用来告诉DOS停止字符串输出。Borland[7]建议指令标号本身应该独占一行，因为这样更容易识别一个标号，而且当一个指令需要添加到标号后面时，这样更容易实现。但是，标号可能与它涉及的指令在同一行，涉及指令的标号必须以一个冒号结束，而涉及数据对象的标号则不需要冒号。当标号是一个程序中的目标时，例如循环指令或跳转指令中的标号，也不需要使用冒号。

保留字offset用来指示汇编器计算从指令到标号“PrnStrg”的偏移，并将该值存入DX寄存器。这完善了用来完整指示将要输出的字符串数据的段和偏移值所必需的代码。一旦我们确定了字符串的指针，我们就可以将输出一个字符串的DOS功能调用09载入AH寄存器。该调用通过一个软件中断INT 21h来实现，INT 21h与68K模拟器中的指令TRAP #15的功能相同。

程序通过一个DOS终止调用（AH = 4Ch时INT 21h）来停止，END保留字告诉汇编器停止汇编。紧跟着END命令的标号告诉汇编器程序是从哪里开始执行的。这可能与代码段的开头有区别，如果你想要在某个位置而不是代码段的开头进入程序，那么这是非常有用的。

10.16 系统向量

与68K一样，最开始的1K存储地址是为系统中断向量和异常保留的。在8086体系结构中，中断号是一个0~255的8位无符号数。中断操作数左移两次（乘以4）可以得到中断处理程序的指针地址。因此，INT 21h指令使得处理器通过存储地址00084h能得到操作系统入口处4字节 291
的段和偏移。在这个例子中，IP偏移定位在字地置00084h，而CS指针则定位在字地址00086h。AH寄存器中的功能代码09能让DOS打印DS:DX所指向的字符串。

10.17 系统启动

一个基于8086的系统重启（RESET）时，除了CS寄存器被设置为0FFFFh之外，所有的寄存器都被设置为0，所以第一个取指令的物理地址是0FFFFh:0000或者0FFFF0h，这是离物理存储器顶部16个字节处的地址。因此，8086系统通常在存储器高处有非易失性存储器，以便在系统重启时装载引导代码。一旦重启后，这16个字节也足以用来执行少量代码，包括跳转到实际进行初始化的代码的开头。一旦进入了ROM代码的开始处，系统就会通过将值写入RAM中来初始化存储器低处的中断向量，而这些值也就占据了地址空间的存储器低端部分。

总结

这一章马上就要结束了，你可能非常高兴也可能会失望。毕竟我们仔细研究了68K的指令集，也学习了很多汇编语言程序的例子。在这一章中，我们接触了很多的代码片段，但只有一个相对很小的程序，为什么呢？

在这本书的开始部分我们学习了汇编语言程序设计的基础，同时也学习了计算机的体系结构。68K系列本身的体系结构是相当简单的，这就可以让我们集中精力于寻址和算法的基本原理。8086体系结构是比较难掌握的一种体系结构，所以我们把它的介绍一直推迟到了这本

书的后面。既然你已经学习了汇编语言程序设计的一般方法，我们就应该把精力集中在掌握8086体系结构本身的错综复杂方面。无论如何，毕竟理论上是这样的。

下一章我们将学习第三种体系结构，你可能又一次发现它很新鲜或很令人沮丧。令人沮丧的是你不再拥有所有在8086体系结构中的功能强大的指令模式和寻址模式；新鲜的是你不需要掌握所有功能强大而复杂的指令模式和寻址模式。

本章讲述了以下内容：

- 8086和8088微处理器的基本体系结构
- 8086的存储模型和寻址模型
- 8086系列的指令集体系结构
- 8086汇编语言程序设计的基础

参考文献

[1] Daniel Tabak, *Advanced Microprocessors, Second Edition*, ISBN 0-07-062843-2, McGraw-Hill, NY, 1995, p. 186.

[2] Advanced Micro Devices, Inc, *Am186™ES and Am188™ES User's Manual*, 1997, p. 2-2.

[3] Borland,*Turbo Assembler 2.0 User's Guide*, Borland International, Inc. Scotts Valley, 1988.

[4] Advanced Micro Devices, Inc, *Am186™ES and Am188™ES Instruction Set Manual*, 1997.

[5] Walter A. Triebel and Avatar Singh, *The 8088 and 8086 Microprocessors*, Third Edition, ISBN 0-13-010560-0, Prentice-Hall, Upper Saddle River, NJ, 2000. Chapters 5 and 6.

[6] Intel Corporation, *8086 16-Bit HMOS Microprocessor*, Data Sheet Number 231455-005, September 1990, pp. 25–29.

[7] Borland, *op cit*, p. 83.

习题

1. 存储地址0C0020h处的内容为0C7h，且存储地址0C0021h处的内容为15h。0C0020h处存储的字是什么？它是对齐的还是非对齐的？
2. 假设你有一个存储在存储器中字节地址0A3004h到0A3007h的指针（段:偏移），如下所示：
   <0A3004h> = 00
   <0A3005h> = 10h
   <0A3006h> = 0C3h
   <0A3007h> = 50h
   将这个指针用段:偏移的形式表示。
3. 如果段寄存器的值是0A300h，那么物理存储地址0A257Ch处的偏移值是什么？
4. 将下面的汇编语言指令转换为与它们等价的目标代码：
   MOV AX, DX
   MOV BX[SI], BX
   MOV DX, 0A34h
5. 写一段实现下面功能的简单代码：
   a. 将值10载入BX寄存器，将值4载入CX寄存器。
   b. 执行一个循环，使BX每次增1并递减CX直到<CX> = 00。
6. 将值0AA55h载入AX寄存器并交换寄存器中的字节。
7. 当执行完下面的两个指令后AX寄存器的内容是什么？

```
MOV    AX, 0AFF6h
ADD    AL, 47h
```

8. 假设你想执行数学操作X = Y * Z，其中：
   X是偏移地址200h处的一个32位无符号变量，
   Y是偏移地址204h处的一个16位无符号变量，
   Z是偏移地址206h处的一个16位无符号变量，
   写一段实现该操作的8086汇编语言代码。
9. 编写一段程序，将从地址82000h开始的1000个字节数据移到地址82200h处。 294
10. 修改第9题中的程序，使得程序将1000个字节的数据从82000h移到C4000h。

第11章 ARM体系结构

学习目标

- 描述ARM系列的处理器体系结构；
- 描述ARM7处理器的基本指令集体系结构；
- 描述ARM体系结构和68000体系结构之间的异同；
- 结合体系结构的所有寻址模式和指令，编写ARM汇编语言的简单程序。

11.1 引言

我们将把注意力从68K和8086体系结构转移到一个新的方向。你可能会认为这次方向的改变非常新鲜，因为我们将要学习的体系结构的特征是：削减指令，只剩下最基本的指令模式和寻址模式。我们称围绕该体系结构建立起来的计算机为RISC计算机，这里RISC是*精简指令集计算机*（Reduced Instruction Set Computer）的缩写。68K和8086处理器是以一种称为*复杂指令集计算机*（Complex Instruction Set Computer, CISC）的体系结构为特征的。我们将在后面的一章比较这两种体系结构。目前，我们只需往前学习ARM体系结构，权当我们没有听说过CISC和RISC。

1999年，ARM 32位体系结构最终在受欢迎程度上超过了Motorola 68K体系结构[1]。68K体系结构从它被发明开始就一直统治着嵌入式系统世界，但是ARM则成了今天最受欢迎的32位嵌入式处理器。而且，今天的ARM处理器比Intel奔腾系列卖得还多，基本上是3 : 1的比例[2]。因此，你应该已经明白了我们讲授这3种微处理器体系结构的道理所在。

如果你碰巧在得克萨斯州奥斯汀，那么你应该到城南去访问一下AMD公司的有影响力的硅片制造厂FAB 25。在这个现代的价值几十亿美元的工厂中，硅圆片被转变为Athlon微处理器。不远处，Freescale（Motorola）的FAB（制造设备）则生产PowerPC处理器。Intel则在美国亚利桑那州钱德勒和加州圣何塞以及世界其他地方的FAB生产它的处理器。ARM的FAB在哪里呢？小心，这是一个圈套。事实上，ARM没有任何FAB，它是一个没有制造厂的微处理器生产商。

ARM控股公司（Holdings PLC）是在1990年以Advanced RISC Machines Ltd.[3]的名字建立的，它的基地在英国，是Acorn计算机集团、苹果公司和VLSI Technology联合投资建立的。ARM自己并不制造芯片，它将芯片设计授权给合作者们，如VLSI Technology、Texas Instruments、Sharp、GEC Plessey和Cirrus Logic，他们将ARM处理器融于它们生产和销售的定制器件中。正是ARM创建了*知识产权*（Intellectual Property, IP）的销售模式，而不是将硅芯片封装在包中。在这种意义上说，这无异于购买软件，实际上也确实是。想要生产诸如PDA/手机/照相机/MP3播放器等*硅上系统*（system-on-silicon）的客户就要与VLSI Technology订合同来生产物理部件。获ARM授权的VLSI公司提供给客户一个用硬件描述语言（如Verilog）编写的加密库。与客户许可的其他IP以及他们自己设计的IP一起，就创建了芯片的Verilog描述，VLSI公司就可以利用这个描述来创建物理部件。因此，你就可以看到像ARM之类公司的出现，Mead和Conway预言的集成电路设计模式成为现实。

今天，ARM为广阔的应用领域提供了各种处理器设计。就像学习68K和8086时我们所做的一样，我们将把精力集中在该系列的大多数产品中最普通的基本32位ARM体系结构。

当我们讨论ARM处理器时，我们经常用一个术语“核”（core）。核是一个微处理器中最基本的部分。当设计硅上系统（或片上系统）时，一个或多个微处理器核与外围设备组合在一起可以在单个硅片上形成完整的系统设计。简单形式的SOC已经出现一段时间了。我们今天称这些商业上可得到的部件为微控制器（microcontroller）。Intel和Motorola是微控制器制造的先驱。历史上，像Motorola之类的半导体公司都会开发一个新的微处理器（如68K系列），并将这个新产品高价卖给专门的客户，就是那些一直在做前沿设计并愿意以高价买到最高性能的最新处理器的客户。

随着处理器得到更广泛的认可，半导体公司改进了其制造工艺，这些公司常常移植CPU核，将其置入拥有其他外围器件（例如定时器、存储控制器、串口和并口）的另一个设备中。由于这些设备有更高的集成度，所以这些部件在注重节省成本的应用中变得非常流行。然而，一些客户只需要购买提供方产品的一些特别部件，如果客户需要一个不同的部件，他们或者购买他们可以找到的最相近的部件并在印刷电路板上增加额外的电路，或者与卖方合作为他们的产品设计一个定制的微控制器。

因此，我们所讨论的可能是采用原始的68K核如68302（称为CPU16）的Motorola微控制器，也可能采用全32位68K核如68360（CPU32）的更先进的微处理器。Intel的80186处理器采用8086核在单个芯片上组建了一个完整的PC，AMD的SC520（Elan）则采用486核在单个芯片上组建了一个完整的PC。最初的PalmPilot使用一个Motorola 68328微控制器（代号Dragonball）作为它的引擎。因此，由于所有的ARM处理器本身就是（要被设计植入片上系统的）核，所以我们将继续使用这个术语。

11.2 ARM体系结构简介

图11-1是ARM核体系结构的一个简化的示意图。数据和指令进入后例行地被送入指令译码器或标记为r0-r15的某个通用目标寄存器。与68K和8086处理器中到内存的单一双向数据总线不同，很多ARM都被设计成单独的数据总线和地址总线进入指令存储器（代码空间）和数据存储器。这种带有单独数据存储器和指令存储器的设计叫做哈佛体系结构（Harvard Architecture）。 296

所有的数据操作都发生在寄存器文件（register file）中，这是一组16位或32位宽的通用目标寄存器。这与我们前面讨论的专用地址寄存器、数据寄存器和变址寄存器的概念截然不同。尽管一些寄存器确实拥有专门的功能，但是另外一些寄存器却只可以用作源操作数、目的操作数或存储器指针。从存储器到寄存器的8位和16位宽的数据在存到寄存器之前被自动转换成了带符号扩展的32位数字。这些寄存器被认为是正交的（orthogonal），因为它们在地址或数据存储方面可完全互换。

所有的算术操作和逻辑操作都发生在寄存器之间。如下所示的68K的存储操作数和寄存器混合加法之类的指令是不允许的：

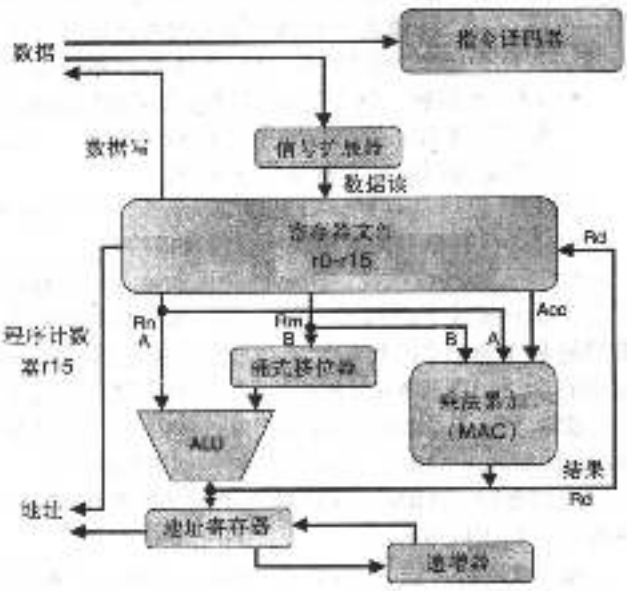


图11-1 ARM体系结构

```
ADD  D5,$10AA     *将D5加入到$10AA并将结果存储在$10AA中
```

而且，大多数算术和逻辑操作都涉及3个操作数。因此，两个操作数Rn和Rm被处理，结果Rd被返回至目的寄存器。例如，指令：

```
ADD  r7,r1,r0
```

将寄存器r0和r1的32位带符号数相加并将结果放入r7。通常，3操作数指令的格式为：

```
opcode        Rd,Rn,Rm
```

297 其中Rm操作数在进入ALU之前还要通过一个桶式移位器单元，这意味着在一条汇编语言指令中移位操作将在Rm操作数上执行。除了标准的ALU，ARM体系结构还包括一个专用的*乘法-累加*（multiply-accumulate, MAC）单元，它既可以完成两个寄存器的一次标准乘法操作，又可以将结果与另一个寄存器累加。基于MAC的指令在信号处理应用中在是非常重要的，因为MAC操作是数值积分的基础。请看图11-2所示的数值积分的例子。

为了计算曲线下面的面积或者求解一个积分方程，我们可以使用一种数值近似方法。曲线下面积的每一次连续计算都涉及计算一个小直角梯形的面积并将这些面积求和。每个面积的计算都是一次乘法操作，而这些梯形面积的和就是总面积。MAC单元做乘法并保持求和操作在一次操作内完成。

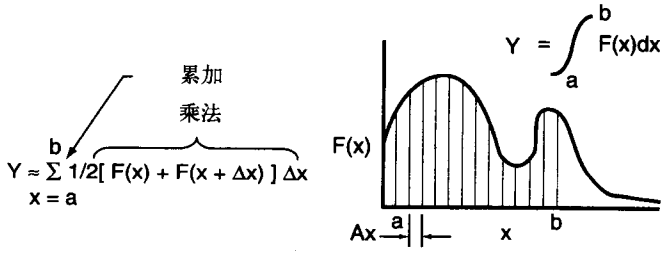


图11-2 一个乘法-累加（MAC）单元能够加速数值积分计算

ARM处理器使用*装载/存储体系结构*。*装载*（load）是指数据从存储器传输到寄存器文件中的一个寄存器，*存储*（store）是指数据从寄存器文件传输到存储器。装载和存储指令使用ALU计算存储在地址寄存器中的地址值以便传送到地址总线上。递增器用来为后续的装载或存储操作增加地址寄存器的值。

在任何时候，一个ARM处理器可能处于7种操作模式之一。最基本的模式叫做*用户模式*（user mode），用户模式在7种模式中是优先级最低的。当处理器在用户模式下时它正在执行用户代码。在这种模式下有18个活动的寄存器：16个32位的数据寄存器和2个32位宽的状态寄存器。在16个数据寄存器中，r13、r14、r15被分配了特别的任务。

- r13：栈指针。这个寄存器指向当前操作模式下的栈顶。在特定的情况下，这个寄存器也可能被用作另一个通用寄存器，然而，当运行在一个操作系统下时这个寄存器经常被指定为指向一个有效的堆栈帧。
- r14：连接寄存器。当处理器执行一个子程序分支时保存返回地址。在特定情况下，这个寄存器也可能被用作一个通用寄存器。
- r15：程序计数器：保存将要从存储器中获取的下一条指令的地址。

在用户模式下进行操作时，*当前程序状态寄存器*（current program status register, cpsr）的功能是作为标准的程序标志和处理器标志的仓库。cpsr也是寄存器文件的一部分并且也是32位宽的（尽管在基本的ARM体系结构中有很多位没有使用）。

实际上有两个程序状态寄存器。第二个程序状态寄存器叫做*程序状态保存寄存器*（saved program status register, spsr），当发生模式改变时，它被用来存储cpsr的状态。因此，当发生
298 上下文切换时，ARM体系结构将状态寄存器保存在一个特定的地方而不是压入栈中。程序状态寄存器如图11-3所示。

程序状态寄存器分为4个域：*标志*、*状态*、*扩展*和*控制*。在基本的ARM体系结构中，没有用到状态和扩展域，它们是为以后的扩展所保留的。标志域包含4个状态标志：

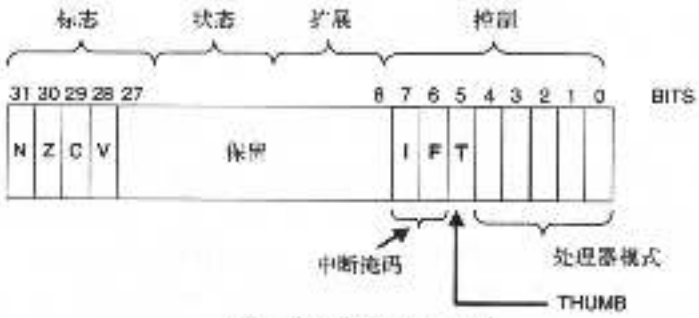


图11-3 状态寄存器配置

- N位：负标志。当结果的第31位为负数时置位。
- Z位：零标志。当结果为0或相等时置位。
- C位：进位标志。当结果产生无符号进位时置位。
- V位：溢出标志。当结果产生符号溢出时置位。

中断掩码位用来使能或屏蔽到处理器的两种类型的中断请求。当中断掩码位为使能时，处理器可以接收正常的中断请求（IRQ）或快速中断请求（FIQ）。当其中一位被设置为1时，相应类型的中断就会被屏蔽，或者说是阻止中断源停止处理器当前执行线程并为中断服务。

Thumb模式位则与你手头的工作没有联系。它是一种特殊的模式，是为了通过将初始的32位ARM指令集压缩成16位形式从而提高ARM的代码密度而设计的，因此在程序存储空间需求上可以减少到一半。特别的片上硬件即时地将Thumb指令解压缩回标准的32位宽指令。然而，世上没有免费的午餐，使用Thumb模式也有一些固有的局限。例如，当处理器处于Thumb模式时，只有通用目标寄存器r0到r7可以使用。

位0到位4定义了当前处理器的操作模式。ARM处理器可能处于下表所定义的7种模式之一：

模　式	缩写	特权	模式位[4:0]
异常中止	abt	是	10111
快速中断请求	fiq	是	10001
中断请求	irq	是	10010
管理	svc	是	10011
系统	sys	是	11111
未定义	und	是	11011
用户	usr	否	10000

用户模式的优先级最低，这意味着它不能修改处理器状态寄存器的内容，换句话说，它不能使能或屏蔽中断、进入Thumb模式或修改处理器的操作模式。在逐个讨论这些模式前，我们需要先弄明白每种模式是怎样使用寄存器文件的。基本的ARM体系结构中一共有37个寄存器，我们讨论了在用户模式下能够访问的18个寄存器。当其他操作模式活动时，剩下的19个寄存器就会产生相应的作用。图11-4显示了每个处理器操作模式的寄存器配置。

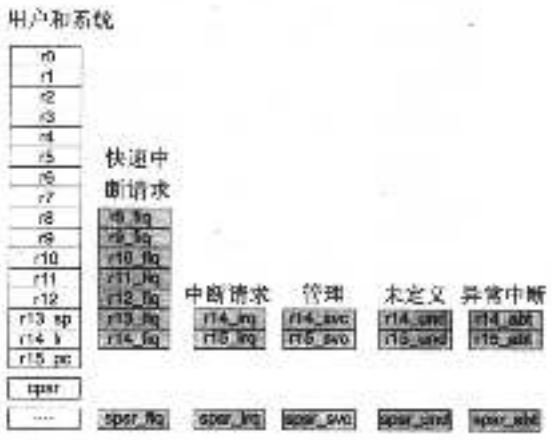


图11-4 寄存器文件结构

当处理器运行在用户模式或系统模式下时，13个通用寄存器、sp、lr、pc和cpsr是

299 活动的。如果处理器进入快速中断请求模式，标记为r8_fiq到r14_fiq的寄存器会自动与相应的寄存器r8到r14进行交换。当前程序状态寄存器（cspr）的内容也自动转移到程序状态保存寄存器（spsr）。

因此，当一个快速中断请求到达处理器，而且CPSR中的FIQ掩码被置有效时，处理器能够快速自动地建立一组新的可以使用的寄存器来服务这个快速中断请求。而且，由于当中断请求被服务时cpsr的内容需要用来存储用户程序的上下文，所以cpsr被自动保存到了spsr_fiq。除了用户和系统模式，只要其他模式处于活动状态，spsr寄存器都自动保存cpsr的内容。

无论处理器在何时改变模式，它都必须能够最终返回至以前的模式并且从它离开的确切位置重新开始。因此，当处理器进入一种新模式时，新寄存器既指向r13_xxx中新模式的存储栈，也指向r14_xxx中前一种模式的返回地址。总之，当处理器转换模式时，新的上下文自动建立而老的上下文则自动保存。当然，我们也可以在软件中完成这个任务，但是对于实时应用，使用这些额外的硬件资源可以获得更好的处理器性能。

由于使用的寄存器都是成组进行交换的（称为*存储体切换*（bank switch)），它们的内容可能被预先装载了服务快速中断请求所需要的适当值。而且，由于FIQ模式是一个更高优先级的模式，所以当中断完毕时，用于服务快速中断请求的程序代码也能将操作模式修改回用户模式或系统模式。我们将在后面一章更详细地讨论中断和中断服务的一般概念。

其他的处理器模式：中断请求、管理、未定义和异常中止都与快速中断模式的运行大体相同。当进入一个新模式时，该模式相应的寄存器会与用户模式下的寄存器进行存储体切换。基寄存器的当前内容不被修改，以便当用户模式被恢复时，旧的工作寄存器用存储体切换发生之前的值再次激活。

我们剩下的最后任务是学习这7种处理器模式。

- *用户模式*：这是通常的程序执行模式。处理器按常规使用寄存器r0到r12，而且cpsr的标志位是根据能够修改标志的汇编语言指令的执行结果来进行相应修改的。
- *系统模式*：系统模式是具有更高优先级的用户模式，这意味着在系统模式下处理器可
300 以修改cpsr的值。你可能回想起68K也有一种用户和管理模式，该模式下可以访问修改状态寄存器位的附加指令。当你的程序还没有复杂到需要操作系统的加入时，系统模式是可以用来使用的恰当模式。对于更简单的应用，对处理器模式的访问将会是一个优势，所以系统模式将是一个显而易见的选择。
- *快速中断请求模式*：FIQ和IRQ模式是为处理处理器中断而设计的。快速中断请求模式提供的成组寄存器比标准中断请求模式的多，这样在中断服务进行过程中需要保存额外的寄存器时，处理器的开销就比较小。而且，对于创建一个*直接存储器访问*（DMA）控制器的软件模拟来说，FIQ模式中使用的7个成组寄存器已经足够了。
- *中断请求模式*：像FIQ模式一样，IRQ模式是为服务处理器中断而设计的。当处理器确认中断并将上下文转向中断服务程序时，有两组成组寄存器可用。
- *管理模式*：管理模式是为了与操作系统内核一起使用而设计的。在管理模式下时，所有的CPU资源都可用。当处理器第一次重启后，管理模式是活跃模式。
- *未定义模式*：这个模式是为非法指令或目前使用的ARM体系结构版本不支持的指令而保留的。
- *异常中止模式*：在特定情况下，处理器可能会试图进行非法的存储器访问。异常中止模式是为处理访问受限存储操作而保留的。例如，特定的硬件被设计用来探测对只读存储器进行写操作的非法意图。

11.3 条件执行

ARM体系结构提供了一个我们以前没有考虑的相当独特的特征，即基于条件标志的状态或状态的逻辑组合来条件执行大多数指令的能力。条件代码以及它们的逻辑定义如下表所示：

代码	描　述	标　志	操作码[31:28]
EQ	等于0	Z = 1	0000
NE	不等于0	Z = 0	0001
CS HS	进位置位/无符号高于或相等	C = 1	0010
CC LO	进位清零/无符号低于	C = 0	0011
MI	负数或减	N = 1	0100
PL	正数或加	N = 0	0101
VS	溢出置位	V = 1	0110
VC	溢出清零	V = 0	0111
HI	无符号高于	$\overline{Z}$*C	1000
LS	无符号低于或相等	Z+$\overline{C}$	1001
GE	带符号大于或等于	(N*V)+($\overline{N}$*$\overline{V}$)	1010
LT	带符号小于	N xor V	1011
GT	带符号大于	(N*$\overline{Z}$*V)+($\overline{N}$*$\overline{Z}$*$\overline{V}$)	1100
LE	带符号小于或等于	Z + (N xor V)	1101
AL	总是（无条件）	未使用	1110
NV	从不（无条件）	未使用	1111

301

条件代码可以加到指令助记符的后面以使指令根据标志的当前状态执行。例如：

```
SUB    r3,r10,r5
```

从r10寄存器减去r5寄存器的内容并将结果放入寄存器r3中。将指令写为：

```
SUBNE  r3,r10,r5
```

则只有在零标志 = 0时才执行减法操作。

因此，不需插入专门的转移或跳转指令，指令的条件执行就可以实现。由于很多汇编语言的结构都涉及跳转到下一条指令，条件执行机制的添加极大地提高了指令集的代码密度。

ARM体系结构的另一个独特的特征是，它被认为是数据处理指令这一类的指令，包括移动指令、算术和逻辑指令、比较指令和乘法指令，并不自动改变条件代码标志的状态。就像前面那个例子可以选择基于条件代码标志的状态来条件执行一条代码一样，我们也可以控制一条数据处理指令的结果是否改变相应标志的状态。例如：

```
SUB    r3,r10,r5
```

执行减法操作但不改变标志的状态。

```
SUBS   r3,r10,r5
```

执行相同的减法操作，但是改变标志的状态。

```
SUBNES    r3,r10,r5
```

条件执行相同的操作并改变标志的状态。条件执行标志“NE”先出现，紧跟着的是条件标志更新符号“S”。因此：

```
SUBNES    r3,r10,r5
```

是一条合法指令，但是

```
SUBSNE    r3,r10,r5
```

是非法的并会产生一个未知指令错误（unknown opcode error）。

11.4 桶式移位器

从图11-1我们看到Rm操作数在进入ALU之前通过了一个桶式移位器硬件块，因此，我们可以在一条指令中既执行算术操作又执行移位操作。桶式移位器能将32位Rm操作数在其进入ALU之前移动任意位数（最多32位，左移或右移）。移位完全是一次异步操作，所以不
302 需要额外的时钟周期。我们将把设计一个能够移动任意位数的移位器的门电路实现留作一个练习。

例如，**MOV**指令用于寄存器和寄存器之间的数据传输和给寄存器装载立即数。考虑下面的指令：

```
MOV  r6,r1          ; 拷贝r1到r6
```

现在，考虑增加了逻辑左移操作的相同指令：

```
MOV  r6,r1, LSL #3   ; 拷贝r1*8到r6
```

在第二个例子中，寄存器r1的内容左移3位（乘以8）后才拷贝到寄存器r6。移动的位数被指定为一个立即数或者一个寄存器的值。下面的代码段执行了与前面例子相同的移位和装载操作：

```
MOV  r9,#3          ; 初始化r9
MOV  r6,r1, LSL r9  ; 拷贝r1*8到r6
```

下面的表格总结了桶式移位器的操作：

助记符	操　作	移 位 量
LSL	逻辑左移	#0-31或寄存器
LSR	逻辑右移	#1-32或寄存器
ASR	算术右移	#1-32或寄存器
ROR	循环右移	#1-32或寄存器
RRX	扩展的循环右移	33

循环右扩展有效地将所有位右移1位并将位31拷贝到进位标志位置。注意标志位是根据S位的状态来按条件更新的。

11.5 操作数大小

ARM体系结构允许对字节（8位）、半字（16位）和字（32位）进行操作。这一次你又可以看到一组稍微不同的定义。然而，作为三种体系结构中最新的一种，ARM的定义可能是最自然的，且更符合C语言标准。

字节可以在任意边界寻址，而半字必须在偶地址边界对齐（A0 = 0），字必须在4字节边界对齐（A0和A1= 0）。这比68K体系结构甚至更严格，但这是可以理解的，因为大多数ARM实现都会有一条到内存的完全的32位宽的数据通路。因此，从A0 = 0和A1 = 1的任意地址获取一个32位宽的存储值需要两次访问操作，或者一次等价的非对齐访问。

通过对内核的正确配置，ARM体系结构支持高端字节序和低端字节序两种数据存放方式。然而，该体系结构以低端字节序作为其默认模式。载入和存储数据的指令可以通过附加操作数类型和大小来可选择地进行修改。下面的表格列举了ARM体系结构中的载入/存储操作的
303 类型：

助记符	描　述	操　作
LDR	从存储器中载入一个字到寄存器中	寄存器 < 32位存储器
STR	将一个寄存器中的字内容存入存储器	32位存储器 < 寄存器
LDRB	从存储器中载入一个字节到寄存器中	寄存器 < 8位存储器
STRB	将一个寄存器中的字节内容存入存储器	8位存储器 < 寄存器
LDRH	从存储器中载入半字到寄存器中	寄存器 < 16位存储器
STRH	将一个寄存器中的半字内容存入存储器	16位存储器 < 寄存器
LDRSB	载入一个带符号字节到寄存器中	寄存器 < 符号扩展的8位存储器
LDRSH	载入一个带符号半字到一个寄存器中	寄存器 < 符号扩展的16位存储器

11.6 寻址模式

对ARM体系结构中可用寻址模式的讨论比我们以前学习的8086或68K体系结构更加严格一些。原因在于寻址外部存储器的寻址模式局限于寄存器文件和存储器间的数据传输操作。在寄存器之间使用数据处理指令有两种可用的寻址模式：*寄存器寻址和立即数寻址*。

```
SUB  r3,r2,r1    ; r3 < r2 - r1
SUB  r7,r7,#1    ; r7递减
```

上面的例子中r7也是减法操作的目的寄存器。通过将r7减1并将结果存回r7，我们有效地执行了一条递减指令。

这看起来可能非常奇怪，但是并不是0到$2^{32}-1$之间的每一个立即数都可以指定。原因在于32位指令的立即数域只有12位长，而且，这12位域进一步又分为一个8位常数操作数和一个4位移位域。如图11-5所示。

32位立即值是通过将8位立即值循环右移一些偶数位来创建的，所以在位域11:8为0001的值将使得8位立即值循环右移2位。因此，所有有效的立即操作数将由最多有8个相邻二进制1的位于偶数边界的组所组成。

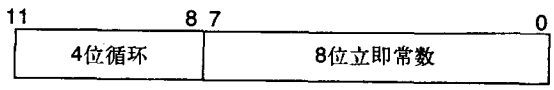


图11-5 定义一个立即操作数的指令的0~11位

这可能看起来像不可思议的约束，但是它似乎没有它最初出现时那么严格了[6]。汇编器将试图通过转换操作数的意义（如果它能）来转换超界的常数值。例如，移动指令（MOV）将转换为*移动取反*（move not, MVN）指令，它会将字中的所有位取反。而且，加指令将被转换为减指令以便得到范围内的操作数。总之，如果汇编器不能将操作数限定在值域范围内，它就将报告一个汇编错误。然后程序员可能会再增加额外的指令（例如一条或多条逻辑或指令）来向立即数中放置增加位。在任何情况下，在实际编程问题中产生的大多数数字（例如字节值或指针地址）都趋向于服从这些规则。

就存储器到寄存器和寄存器到存储器而言，我们有几种可以使用的寻址模式。而且我们 304
也必须区分从寄存器到存储器的寻址模式和那些导致存储器和寄存器之间块移动的指令。还要注意在ARM体系结构中没有绝对寻址模式，所有的存储器寻址模式都要用一个寄存器作为存储器指针，不管是寄存器本身还是作为一次变址操作的基址寄存器。你可能很惊讶为什么要这样，因为所有的ARM指令（除了Thumb子集）都是单个32位长的字。我们不能使用一个绝对地址作为一个操作数，因为要精确地指定一个32位地址需要第二个指令字。

请看下面的例子：

```
LDR    r12,[r7]
```

该指令将寄存器r7所指向的存储字的内容载入了寄存器r12。载入指令的格式为：目的←源。存储指令的格式相同：

```
STR    r12,[r7]
```

该指令将r12的字内容存入了r7所指的存储位置。格式为源→目的。糊涂了？是的。

在这两个例子中，r7中存储的值是存储器访问的基地址。如果我们正在使用一种变址寻址模式，该寄存器将提供地址计算的开始值。我们可以总结字或无符号字节的变址模式，如下表所示：

变址模式	描　述	寄存器偏移	立即数偏移	比例寄存器
不写回的前变址	在使用前计算地址	[Rn,±Rm]	[Rn,#±Imm12]	[Rn,±Rm,shift #imm]
带写回的前变址	在使用前计算地址；基址←基址+偏移	[Rn,±Rm]!	[Rn,#±Imm12]!	[Rn,±Rm,shift #imm]!
带写回的后变址	在使用后计算地址；基址←基址+偏移	[Rn],±Rm	[Rn],#±Imm12	[Rn],#±Rm,shift #imm

如果操作数的大小为半字、带符号的半字、带符号字节或双字，那么可用的变址模式需要稍微修改：

变址模式	描　述	寄存器偏移	立即数偏移	比例寄存器
不写回的前变址	在使用前计算地址	[Rn,±Rm]	[Rn,#±Imm8]	N/A
带写回的前变址	在使用前计算地址；基址←基址+偏移	[Rn,±Rm]!	[Rn,#±Imm8]!	N/A
带写回的后变址	在使用后计算地址；基址←基址+偏移	[Rn],±Rm	[Rn],#±Imm8	N/A

这里我们使用了以下一些术语：

305 Rn：基址寄存器　　Rm：变址寄存器　　Imm8：8位立即数偏移

Imm12：12位偏移　　!:带写回　　shift：移位操作

下面的指令集给出了上面表格中的几种变址寻址模式的例子：

例1　　`LDR　　r12,[r0,+r7]`

基址寄存器r0和变址寄存器r7的内容加在一起形成一个对齐于4字节边界的存储地址指针addr。addr处的字存储内容被拷贝到寄存器r12。r0的内容未被改变。

例2　　`LDRB　　r12,[r0,-#0x6A0]`

从基址寄存器r0的内容中减去立即值0x6A0形成一个存储地址指针addr。addr的值没有限制。addr处的字节内容被拷贝到寄存器r12。r0的内容未被改变。

例3　　`STR　　r12,[r0],+#0x6A0`

寄存器r12的字内容被写至r0所指向的存储位置。当数据传送发生后立即数值0x6A0被加至r0的内容，其和存回至r0。

例4　　`STRH　　r12,[r0,+r7]!`

基址寄存器r0和变址寄存器r7的内容加在一起形成一个存储地址指针addr。结果地址必须在偶数边界。寄存器r12的半字内容被写到存储地址addr。基址寄存器的内容被更新为包含值addr。

例5　　`LDR　　r12,[r0,-r7,LSL #3]!`

变址寄存器r7的内容逻辑左移3位。

因此，如果在计算前<r7> = 0x00000AC4，那么从r0减去的值将为 (0x00000AC4) × 8 = 0x00005620。

从基址寄存器r0中减去结果值就形成了存储指针addr。结果地址必须对齐于4字节边界。

存储在存储器中addr处的字值被拷贝到寄存器r12中，且寄存器r0被更新为值addr。

例6 `LDREQ r12,[r0,-r7,LSL #3]!`

当零标志Z被置位时执行例5的操作。

你可能在这些细节的琐碎介绍中遗漏了一些东西，所以我将在这里提及一下。如果在地址计算中使用的基址寄存器是程序计数器r15，那么所有的变址寻址操作都是与pc相对的和位置独立的。回忆可知在68K中我们必须显式地选择一种pc相对的寻址模式。在ARM体系结构中，pc相对寻址只是选择r15作为基址寄存器时的附带结果。

11.7 堆栈操作

ARM体系结构直接强调需要一套有效机制以支持基于堆栈的高级语言、外加载入和存储指令这两种变型，以及支持压栈（push）和退栈（pop）操作的额外寻址模式。我们也要指出这些指令不仅仅局限于堆栈操作，当多个寄存器要保存时也是非常有用的。 306

这两条指令是：

LDM：载入多个寄存器

STM：保存多个寄存器

这两条指令使用的语法与68K的MOVEM指令的语法非常相似。需要保存或恢复的寄存器放在括号中，可以用一个连字符来指示一个连续范围内的寄存器，也可以用以逗号分隔的寄存器列表来指示。数据传送从指向存储器的基址寄存器Rn中存储的地址开始进行。当数据传送发生后，指针寄存器Rn可以通过添加一个“!”符号到指令中来选择性地更新，就像我们刚刚学过的变址寻址模式一样。这种寻址模式区别于变址寻址模式的地方在于使用的寻址模式是附加在指令操作代码后而不是放在操作数域中。例如，为了从存储器中装载寄存器r0到r3且使得存储指针r10在每一次数据传送后都递增，我们使用以下指令形式：

```
1  LDMIA   r10,    {r0-r3}      或
2  LDMIA   r10!,   {r0-r3}
```

假定在指令执行前<r10> = 0xAABBCC00。第一个寄存器r0从存储地址0xAABBCC00处恢复，寄存器r1从存储地址0xAABBCC04处恢复，寄存器r2从存储地址0xAABBCC08处恢复，而r3则从地址0xAABBCC0C处恢复。指令执行完后：

情况（1）：<r10> = 0xAABBCC00

情况（2）：<r10> = 0xAABBCC0C

多个寄存器传送指令有4个寻址条件：

助记符	描　述	注　释
IA	后递增	第一次数据传送发生在基址寄存器Rn所指向的存储位置。接下来的传送则发生在连续的更高存储位置
IB	前递增	第一次数据传送发生在比基址寄存器Rn的初始值高4个字节的存储位置。接下来的传送则发生在连续的更高存储位置
DA	后递减	第一次数据传送发生在基址寄存器Rn所指向的存储位置。接下来的传送则发生在连续的更低存储位置
DB	前递减	第一次数据传送发生在比基址寄存器Rn的初始值低4个字节的存储位置。接下来的传送则发生在连续的更低存储位置

多个寄存器数据传送操作的语法可以总结如下：

LDM（可选择的条件执行）（寻址模式）Rn（可选择的更新），{寄存器列表}

STM（可选择的条件执行）（寻址模式）Rn（可选择的更新），{寄存器列表}

因此，对于指令：

```
LDMNEIB        r11!, {r3, r5, r8}
```

307 如果零标志被清零，该指令将会把寄存器r11+4位置指向的存储地址的内容载入寄存器r3、r5和r8。假定指令被执行，r11中的值将会增加12个字节。

为了专门实现标准的压栈和退栈操作，ARM为我们刚才讲述的4种寻址模式定义了额外的别名助记符。在讨论它们之前，我们应当花一点时间回顾一下我们所知道的栈操作。

栈是后进先出（LIFO）数据结构的硬件实现。和我们在68K和8086体系结构中所看到的一样，栈从高存储地址向低存储地址方向增长，栈指针指向当前栈顶的数据。当数据在一次退栈操作中从栈中取出时需要用到栈指针的当前值，然后栈指针增加该存储操作数的大小。当数据在一次压栈操作中存入堆栈时，栈指针减少相应的数量，然后这个新数据被添加到栈中。ARM的定义中，这种类型的堆栈对于压栈操作将采用前递减寻址模式，而对于退栈操作则采用后递增寻址模式。

然而，我们并不是必须创建一个使用这种模型的堆栈，我们也可以很容易地建立一个向高存储区增长、向低存储区缩减的堆栈，这对于使用ARM处理器的硬件和软件设计者来说肯定更加灵活。如果我们决定使用一个向高存储地址增长的栈，那将更加符合我们头脑中的“增长”栈的图景。让我们考虑堆栈的这种模型。我们必须决定的是我们想要栈指针指向什么地方和我们想要它怎样运行。假设我们想要压入一个数据项到栈中，我们有以下两种选择：

1. 栈指针指向堆栈中的下一个可用的空白空间。当我们添加一个元素时，栈指针的当前值提供了该数据指针，然后栈指针被递增从而为容纳下一个数据项做准备。

2. 栈指针指向最后压入栈中的项的地址。在我们压入另一个项到栈中之前，我们必须在存储数据前递增指针以使指针指向下一个可用的存储地址。

类似地，当我们进行退栈操作时也有两种选择。退栈操作的选择并不是独立于压栈操作的，当我们选择压栈的一种策略时，退栈策略也就确定了，反之亦然。对于一个退栈操作：

1. 如果栈指针指向栈中的下一个可用的空白存储空间，我们必须首先减少栈指针以使栈指针指向最后一个入栈的项，然后我们就可以将数据载入寄存器了。

2. 如果栈指针指向最后入栈的项，我们就可以直接装载存储器，然后递减栈指针值。

从上面的讨论你可以看出对于压栈操作，第一种情况是后递增（increment after）的例子，第二种情况是先递增（increment before）的例子。对于退栈操作，第一种情况是先递减（decrement before）的例子，第二种情况是一个后递减（decrement after）的例子。

现在回到堆栈的话题。这4种堆栈实现的选择取名如下：

308 1. 满上升（Full Ascending）：这是一种满堆栈模型。栈指针指向栈中最后填充的存储位置。如果栈在增长，我们需要使用前递增方法。在堆栈助记符中，IB模型的别名是FA。

2. 满下降（Full Descending）：仍然是基于满模型的，在将数据存到存储器之前我们必须递减栈指针。如果堆栈在下降，我们就必须使用先递减方法。在堆栈助记符中，DB模型的别名是FD。

3. 空上升（Empty Ascending）：空栈模型中的栈指针指向栈中下一个可用（空）位置。如果栈在增长，我们就需要使用后递增方法。在堆栈助记符中，IA模型的别名是EA。

4. 空下降（Empty Descending）：如果栈在下降，我们就需要使用后递减方法。在堆栈助

记符中，DA模型的别名是ED。

因此，为了采用满下降模型将r7到r10的4个寄存器压入栈中，我们应该使用以下指令：

```
STMFD SP!, {r7-r10}
```

注意“SP”是寄存器r13的合法别名。

在一些应用中我们实际上不应该选择多数据传送指令，尽管它们提高了代码密度。假设从外设来的时间要求非常高的中断进入了处理器，那么它必须尽可能快地被服务，而且它什么时候可得到服务必须是高度可预知的，或者说，在它被服务前必须等待的最长时间（潜伏期）必须是高度可预知的。在一个中断可以被接收前，包括ARM在内的大多数处理器必须完成它们正在处理的指令。使用多数据传送指令，我们就会增加因每次寄存器到存储器的数据传输操作而产生的潜伏期。尽管使用多数据传送可能更加高效，但我们还是应当使用一组更小的单个数据传送操作，以使中断等待的最长时间为一条简单指令的执行时间，该时间可能是一两个时钟周期，而不是20个或更多的时钟周期。

11.8 ARM指令集

自从ARM1核和ARMv1指令集体系结构面世以来，ARM指令集体系结构就一直在不断发展。ARM系列一直在不断地向前发展，今天的ARM11核支持ARMv6 ISA。为达到我们的目标，我们一直在关注并将继续关注ARMv4体系结构。这种体系结构也包括ARMv4T体系结构，它包含16位压缩的Thumb指令。我们在这个概述中将不再讨论Thumb指令集，但是我们鼓励感兴趣的同学参考本章后面的引文仔细学习Thumb指令集。

所有的ARM指令都是32位字长的，而且大部分（不是全部）指令都是在一个时钟周期内执行的。例外的情况是装载和存储多个寄存器（LDM和STM）的指令以及必须添加等待状态的到慢速存储器的装载和存储操作。

我们可以将ARM指令集分成4类指令：

1. 数据处理指令
2. 载入/存储指令
3. 转移指令
4. 控制指令

下面我们将分别讨论每种类型的一个典型指令示例。

309

数据处理指令

数据处理指令包括下面两条寄存器数据传送指令：

助记符	定　义	操作模式位[25:21]
MOV	将一个32位值移动到寄存器	1101
MVN	将一个32位值的补码移动到寄存器	1111

寄存器移动指令被归入数据处理指令可能会让你感到惊讶。严格地说，它并不是。然而，ARM体系结构假设所有的数据处理操作都是在寄存器之间进行的，而载入和存储操作发生在存储器和寄存器之间。因此，载入一个值到寄存器的操作被归入数据处理指令，不管这个值是一个立即数还是另一个寄存器的内容。

下一组数据处理指令包含算术指令：

助记符	定　义	操作模式位[25:21]
ADD	两个32位数字相加	0100
ADC	带进位的两个32位数字相加	0101
SUB	两个32位数字相减	0010
SBC	带借位的两个32位数字相减	0110
RSB	两个32位数字的反向减	0011
RSC	带借位的两个32位数字的反向减	0111

反向减操作允许减法中使用的两个寄存器操作数颠倒过来，所以如果对于SUB指令是Rd = Rn−Rm，那么RSB指令将会执行Rd = Rm−Rn操作。回忆可知Rm所代表的操作数也可能是一个文字或者一个寄存器的移位值，所以反向减法指令为减法操作提供了进一步的灵活性。

逻辑操作如下所示：

助记符	定　义	操作模式位[25:21]
AND	两个32位操作数按位与	0000
ORR	两个32位操作数按位或	1100
EOR	两个32位操作数按位异或	0001
BIC	按位逻辑清零（与非）	1110

我们对BIC指令并不熟悉。逻辑上它是Rd = Rn * ($\overline{Rm}$)。因为Rm的各位被取反，Rm中任意为1的位都会使得Rn中相应的位被清零。因此，如果Rn = 0xAB且Rm = 0x01，那么BIC操作将得到Rd = 0xAA的结果。

助记符	定　义	操作模式位[25:21]
CMP	两个32位值进行比较	1010
CMN	取反的比较	1011
TEQ	测试两个32位数字是否相等	1001
TST	测试一个32位数字的各个位（逻辑与）	1000

一共有16条数据处理指令，指令的格式就是如图11-6所示的3种可能形式之一。指令一共有3种形式，取决于是否涉及移位，或者第3个操作数是否是一个立即数。位25叫作立即数位，
310 如果第3个操作数是立即数则该位为1，如果是寄存器操作数则为0。让我们来看一些例子。如果指令的格式为：

```
ADD     r0, r1, #0x00AC0000     ; r0 = r1 + 立即数
```

则该指令采用的是立即操作数的形式。指令代码是0xE28108AB。我们可以看到如下所示的详细解析：

- 位31:28 = 1 1 1 0。总是执行。
- 位27:25 = 0 0 1。固定的。
- 位24:21 = 0 1 0 0。加法操作代码。
- 位20 = 0。指令设置标志的结果。1 = 设置标志，0 = 不设标志。
- 位19:16 = 0 0 0 1。第二个操作数寄存器 = r1。
- 位15:12 = 0 0 0 0。结果寄存器 = r0。
- 位11:8 = 1 0 0 0。立即操作数0xAC向右滚动16次（2*<滚动>）。
- 位7:0 = 1 0 1 0 1 1 0 0，立即操作数。

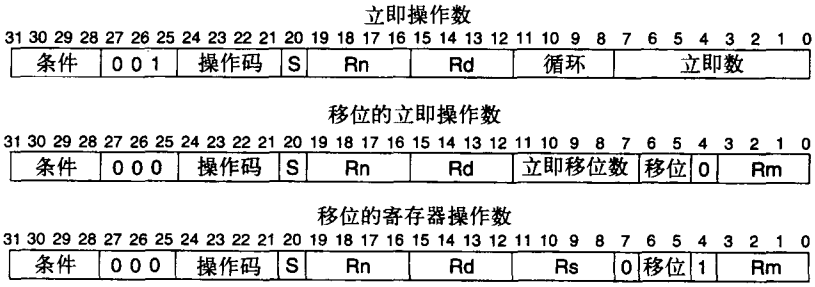


图11-6 ARM数据处理指令的格式

当一个寄存器被指定为第二个操作数，并且一个移位值也被指定为一个立即操作数时，那么就使用立即移位操作数形式。指令：

```
ADD       r3, r7, r4        ; r3 = r7 + r4
```

的指令代码为0xE0873004。16进制数字中的3位最低有效位意味着移位寄存器是r7，移位的类型是逻辑左移，移动的位数是0。

如果我们将指令改为：

```
ADD       r3, r7, r4,  LSL #4        ; r3 = r7 + (r4*16)
```

则指令代码就变为0xe0873204，译码后就是4位的LSL。

最后一个例子是使用第4个寄存器提供移位计数：

```
ADD       r5, r6, r7,  LSR r8        ; r5 = r6 +移位r8
```

指令代码为0xe0865837，让我们对这个代码进行译码：

- 位3:0 = 0 1 1 1。寄存器r7提供移位计数。
- 位6:4 = 0 1 1。逻辑右移。
- 位7 = 0。固定值。
- 其他所有位都可以与前面那个例子一样进行译码。

用于比较的指令组除了设置标志外不会产生任何的结果，因此，结果域没有被使用。在ARM 的行话中，那个域被定义为SBZ (should be zero)。该域任何其他值都可能产生不可预料的结果。[⊖]比较指令的另一个特征是它们总会设置标志，所以S位的状态可以被忽略。 311

数据处理指令的下一组是关于乘法的功能非常强大的指令组。下表所示的是大小不同的乘法指令：

助记符	描　述	语　法
MUL	两个32位数字相乘产生一个32位的结果：Rd = Rm*Rs	`MUL{条件} {S}  Rd, Rm, Rn`
MLA	两个32位数字相乘，并加上第三个数，得到32位的结果：Rd = Rn + (Rm * Rs)	`MLA{条件} {S}  Rd, Rm, Rn, Rs`
UMULL	两个32位无符号数相乘产生一个无符号的64位结果，存储在两个寄存器中：[RdHi][RdLo] = Rm * Rs	`UMULL{条件} {S}  RdLo, RdHi, Rm, Rs`
UMLAL	两个32位无符号数相乘，并加上两个寄存器中的一个无符号64位数，产生一个存储在两个寄存器中的无符号64位结果：[RdHi][RdLo] = [RdHi][RdLo] + Rm * Rs	`UMLAL{条件} {S}  RdLo, RdHi, Rm, Rs`

⊖ 这里有一个关于PC界无畏的爱好者们/先锋们的精彩故事。当时为了解决未实现的操作码到底会完成什么工作的小工业已经初具规模。换句话说，“如果SBZ域被设置为001将会发生什么事？”有时一些非常有趣的未记载的指令被发现后，会被设计成商业产品。不幸的是，当CPU制造商修改芯片时，他们在考虑“谁会使用它们”的问题后经常会改变那些不受支持的指令代码的编码。你可以想像当一批新的处理器大量投入市场之际产品开始失效的糟糕情景。

（续）

助记符	描　述	语　法
SMULL	两个32位带符号数相乘产生一个存储在两个寄存器中的64位结果	**SMULL{条件} {S}　RdLo, RdHi, Rm, Rs**
SMLAL	两个32位带符号数相乘，并与在两个寄存器中的一个带符号64位数相加，产生一个存储在两个寄存器中的带符号64位结果：[RdHi][RdLo] = [RdHi][RdLo]+ Rm * Rs	**SMLAL{条件} {S}　RdLo, RdHi, Rm, Rs**

作为指令的一类，乘法指令也需要执行多于一个时钟周期的时间。

最后，你可能惊讶于ARM指令集不包含任何除法指令。Sloss等[7]描述了用于将除法转化为乘法的近似方法。

载入/存储指令

寄存器和存储器之间的所有数据传输都使用载入和存储类指令。

所有的存储地址都是使用一个基址寄存器指针与一个额外的立即数偏移值、寄存器值或比例寄存器值相加而得到的。而且，计算出来的存储地址指针可在并不将基址寄存器指针更
312 新为新地址值的情况下使用。最后，地址计算可能会在指令使用地址之前或之后发生。

载入/存储指令还必须处理操作数的大小和类型，因为字节和半字也是允许的。

因为我们在寻址模式讨论中已经涉及了很多关于装载/存储指令的操作，所以下面我们只是简单地看一下装载/存储指令字的格式。

装载寄存器指令可以采用以下形式中的任意一种：

助记符	描　述	句　法
LDR	将一个32位存储器字载入寄存器	**LDR{条件} Rn, <地址模式>**
LDRB	将一个8位存储器字节载入寄存器	**LDRB{条件} Rn, <地址模式>**
LDRH	将一个16位存储器半字载入寄存器	**LDRH{条件} Rn, <地址模式>**
LDRSB	将一个8位带符号存储器字节载入寄存器	**LDRSB{条件} Rn, <地址模式>**
LDRSH	将一个16位带符号存储器半字载入寄存器	**LDRSH{条件} Rn, <地址模式>**

带符号字节和半字数据类型必须使用特别的助记符，因为当从存储器装载到寄存器时，这些值必须符号扩展到32位。对于32位存储器值不需要特别的指令，因为它是ARM体系结构的默认数据大小。

图11-7显示了字或无符号字节的装载/存储指令的格式。注意装载和存储基本上是相同的，不同点在于位置20的L位的状态。指令格式对半字和带符号字节有一些微弱的区别。

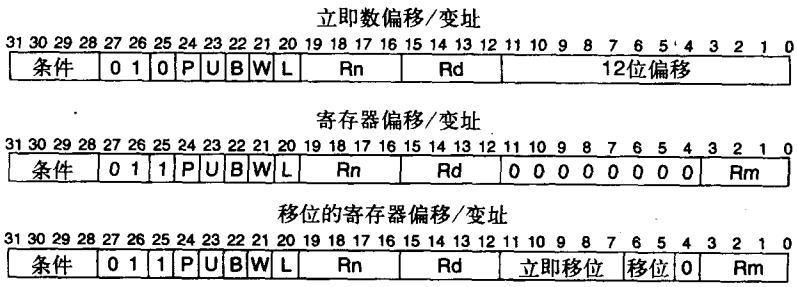


图11-7　ARM字或无符号字节装载/存储指令的格式

通用域的意思如下：

- 位31:28：条件执行域。
- 位27:25：固定的。
- 位24：对于前变址模式，P = 1。偏移被加到基址寄存器，基址寄存器和偏移的和被用作装载/存储的存储器地址。对于后变址模式，P = 0。基址寄存器被用作存储指针，然后偏移值和基址寄存器值的和被写回到基址寄存器。
- 位23：当U = 1时偏移地址与基址寄存器值相加形成存储地址。如果U = 0，则从基址寄存器值减去偏移地址。
- 位22：当B = 1时，访问的存储器处是一个无符号的字节。如果B = 0，那么访问的字节就是一个32位的字。
- 位21：如果位P = 1，那么位W将决定计算出来的存储地址值是否要写回以更新基址寄存器。如果W = 0，基址寄存器不被更新。如果位P = 0且W = 1，那么当前访问就被认为是用户模式访问。如果位P = 0且W = 0，那么就被认为是一次正常的存储器访问。 313
- 位20：如果L = 1那么这是一次存储器装载操作。如果L = 0那么这是一次存储器存储操作。
- 位19:16：基址寄存器指针。
- 位15:12：装载操作的目的寄存器或者存储操作的源寄存器。
- 位11:0：取决于寻址模式。

让我们看看存储器装载和存储操作的一些例子。

助记符	描述	指令代码
`LDR r5, [r8]`	将r8所指的字载入r5	0xE5985000
`LDRSH r6, [r0, -r2]!`	将r0−r2所指的带符号的半字载入到寄存器r5。在存储器装载操作完成后将r0更新为计算的地址值	0xE13060F2
`LDRNE r0, [r9, #-12]`	如果零标志 = 0则条件执行。将寄存器r9所指的字值减去12个字节载入寄存器r0。r9中的值不改变	0x1519000C
`LDRVCB r11, [r4], r2, LSL #4`	如果溢出标志 = 0则条件执行。将寄存器r4所指的无符号字节载入寄存器r11，然后更新r4使得r4 = r4 + r2 * 16	0x76D4B202
`LDR R8, [PC, r5]`	将程序计数器（r15）和r5的当前值之和所指的字值载入寄存器r8	0xE79F8005
`STRB r7, [r3, #0xAA]!`	将寄存器r7的无符号字节存储到r3+0xAA所指的存储地址，将和写回到寄存器r3	0xE5E370AA
`STRCCH r11, [r2, #-&A]`	如果进位标志 = 0则条件执行。将寄存器r11中的半字存入r2−10所指的存储地址，R2不改变	0x3142B0BA
`STREQ r0, [r4, r5, lsr #7]`	如果零标志 = 1则条件执行。将寄存器r0中的字存入r4+（r5右移7位）的和所指的存储地址。R5不改变	0x078403A5
`STRB r6, [r4], r3`	将寄存器r6中的字节存储到r4所指的存储地址。然后将r3和r4的内容相加并用该和更新r4	0xE6C46003
`STRPLH r11, [r9, #-2]!`	如果负数标志 = 0则条件执行。将寄存器r11中的半字内容存入r9-2所指的存储地址。更新r9为新地址	0x5169B0B2

上面的表格应该让你对单一数据传送指令运行的各种形式的语法以及它们是怎样被编码为一个32位指令的有了一个认识。 314

现在让我们来看看多数据项的装载和存储操作的几种形式。

装载多个寄存器和存储多个寄存器的一般形式如图11-8所示。位域15:0中的各位对应于将被装载或存储的寄存器。位中的1代表相应寄存器将被装载或存储。最低编号的寄存器被存在最低存储地址，而最高编号的寄存器被存在最高存储地址。

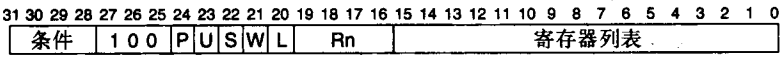


图11-8 ARM多寄存器载入/存储操作的格式

位域的定义如下：

- 位31:28：条件执行域。
- 位27:25：固定的。
- 位24：当P = 1时，地址在存储访问前递增或递减（前变址）。当P = 0时，先使用当前存储指针地址，然后存储指针才被改变（后变址）。
- 位23：当U = 1时每一次传送存储地址都递增，当U = 0时存储地址递减。
- 位22：当S=1且LDM指令正被载入程序计数器（r15）时，程序状态保存寄存器（SPSR）将被装载到当前程序状态寄存器（CPSR）。如果对所有STM指令的装载指令都不涉及r15，那么位S指明当处理器在优先模式时，被传送的是标准用户模式寄存器而不是当前模式的寄存器。位S的状态通过在指令末尾附加一个符号“^”来置位。
- 位21：如果W = 1，指针寄存器将在多寄存器传送发生后永久更新。由于每个数据传送是4个字节长，存储器指针将被更新4倍次的寄存器传送数量。如果W = 0，寄存器将不被更新。
- 位20：如果L = 1那么将会发生一次存储器到寄存器（载入）操作。如果L = 0那么将会发生一次寄存器到存储器（存储）的操作。
- 位19:16：表示指针寄存器。
- 位15:0：寄存器列表。

多数据装载或存储指令的通用语法格式如下所示，括号中的是可选项：

```
LDM 或 STM{条件}XY     Rn{!}, <寄存器列表>{^}
```

这里，XY代表:

- IA：后递增
- IB：前递增
- DA：后递减
- DB：前递减

315

下面是多数据装载和存储指令的两种表示形式。

指　令	描　述	指令代码
`LDMDB SP!, {r0-r3, r5, r7-r9}`	将栈指针（SP）寄存器r13所指的存储块载入到寄存器r0、r1、r2、r3、r5、r7、r8和r9。首先将地址SP-4处的内容载入寄存器r8，并不断递减SP直到r0被装载。更新SP为最后载入寄存器r0的存储字	0xE93D03AF
`STMNEIA r0, {r2-r9}`	如果零标志 = 0，则条件执行这条指令。将寄存器r2到r9的内容存入r0所指的存储块。存储寄存器r2，然后为下一次存储操作递增r0。多个数据传送操作完成后，r0的值被恢复为原来的值	0x188003FC

交换指令（SWP）是载入存储操作的一种特殊类型，它用于交换存储位置的内容与寄存器的内容。现在，你可能会认为这是一条很好的指令，但是它并不太适合我们这种计算机指令集体系结构的精简模型。难道你不能用一个传统的算法去交换存储器和寄存器的内容吗？例如，假设我们想要交换r0的内容和r10所指存储位置的内容：

```
MOV  r8,r0          ; 将r0移到一个临时寄存器
LDR  r0,[r10]       ; 得到存储值，交换的一半已完成
STR  r8,[r10]       ; 保存r8，交换完成
```

交换指令的相应形式为：

```
SWP  r0, r0, [r10]  ; 交换<r10>和r0
```

交换指令的通用形式为：

```
SWP{B}{条件}        Rd,Rm,[Rn]
```

这里，Rn所指的存储位置被载入寄存器Rd，且存储位置的内容被改写为Rm中的值。因此，在通常情况下，交换可以发生在两个寄存器之间和单个的存储位置上。

问题依然存在，“为什么还有交换指令呢?”答案是因为交换指令是*原子*（atomic）操作。原子操作是不能被中断的。大多数指令都是原子指令，也就是说，一旦一条指令已经开始，且处理器接收到一个外部中断，该指令就必须在中断被服务前完成。在上面那个存储器到寄存器交换操作的例子中，我们需要用3条指令完成数据传送。这3条指令并不是原子指令，因为中断可能会使数据交换的过程出现间隙。如果中断也要在这些寄存器或存储器中改变数据，那么数据交换可能会被破坏。交换指令是通过锁存总线使得另一个事件得到控制权之前必须 316
完成当前指令的一种方式。

转移指令

有两种形式的转移指令：*转移*（B）和*带连接的转移*（BL）。指令都很相似，不同的是带连接指令的转移在BL指令后自动地将下一条指令的地址保存到了寄存器r14中。这只是一次子程序调用。为了从子程序返回，你只要将连接寄存器拷贝到程序计数器就可以了。

```
MOV PC, LR
```

转移指令的范围是+/−32MB。像68K一样，ARM体系结构中的转移指令是一个PC相对位移。PC的当前值加上或者减去该位移并将这个新值重新载入PC，就实现了一个程序转移。转移指令的格式如图11-9所示。

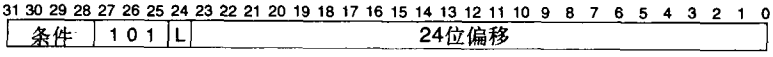


图11-9 ARM转移指令和带连接转移指令的格式

转移地址的计算如下：

1. 24位的偏移值被符号扩展到32位。
2. 结果左移2位（乘以4）以提供一个字对齐的位移值，或者说一个有效的26位字地址。
3. 位移被加到程序计数器，结果存回到程序计数器。

软件中断指令

软件中断指令允许应用代码通过存储在存储器中的向量来改变程序执行的上下文。这条指令类似于68K的TRAP指令和8086的INT指令。通常，软件中断（SWI）被应用程序用来产生一个到操作系统服务的调用。

由于SWI指令被用于改变上下文，所以它也必须保存当前处理器的上下文以便它能够在中断后返回。SWI的行为如下：

1. 将SWI指令后的指令地址保存到寄存器r14_svc。
2. 将CPSR存入SPSR_svc。

   进入管理模式并屏蔽正常的中断，但不包括快速中断请求。
3. 将PC装载为地址0x00000008并执行那里的指令。

我们并不使用异常向量表作为一个软件中断服务程序开始处的间接地址，而是利用向量表位置所包含的一条指令空间来转移到代码的开始处。如果你考虑的是68K的向量表组织，这看起来就很奇怪，但对于ARM体系结构而言这确实没什么关系，因为所有的指令都是一个字长，你不需要使用间接指针去得到ISR代码的开始位置。Motorola必须使用一个向量是因为无条件跳转指令会占用太多的空间。然而，因为一条ARM指令与一个地址占用了相等的空间，所以每种方法都可以。

软件中断指令也包括一个24位的立即操作数域，该域被用于将参数传送到中断服务子程
317 序。因此，使用一个单一向量代替了多个软件中断向量，但是所要求的中断服务类型信息可以被传到指令的操作数域。

程序状态寄存器指令

最后将要学习的ARM指令类型包含两条实现了CPSR或SPSR寄存器与通用寄存器之间的载入或存储操作的指令。这两条指令的句法如下所示：

```
MRS{条件}  Rd, <cpsr或spsr>
MSR{条件}  <cpsr或spsr>_<域>, Rm
MSR{条件}  <cpsr或spsr>_<域>, #立即数
```

MRS指令用于将CPSR或SPSR的当前值移至一个通用寄存器。MSR指令用于将一个通用寄存器的内容或一个立即数值移入CPSR或SPSR。

下面是关于状态寄存器指令的一些解释。如果处理器在用户模式下且指令试图修改除了标志域外的其他域，那么这条指令就会被忽略。为了使这条指令被执行，处理器必须处于那些特权模式中的一种，因为程序状态寄存器只有在处理器运行在特权模式下时才能被修改。

域变量的值如下所示：

- *_C*：控制域表示程序状态寄存器的位0到位7，进一步细分为：
  - 位0:4：处理器模式
  - 位5：使能Thumb模式
  - 位6：使能快速中断请求模式
  - 位7：使能中断请求模式
- *_X*：扩展域表示位8:15。目前这个域没有被使用，只是为ARM以后的扩展保留的。这些位不应该被修改。
- *_S*：状态域表示位16:23。目前这个域没有被使用，只是为ARM以后的扩展保留的。这些位不应该被修改。
- *_F*：标志域表示位24:31。这个域被进一步细分为：
  - 位28：位V－溢出标志
  - 位29：位C－进位标志
  - 位30：位Z－零标志
  - 位31：位N－取反标志

立即操作数只能修改标志域的位。而且，为了防止不小心修改程序状态寄存器中不应修改的位，程序状态寄存器应该采取以下三个步骤来修改：

1. 使用MRS指令将PSR的内容拷贝到一个通用寄存器，
2. 修改通用寄存器的适当位，
3. 使用MSR指令将通用寄存器拷贝回PSR。

下面的指令序列使能FIR模式：

```
MRS   r6, c_spsr   ; 将spsr拷贝到r6
MOV   r7, #&40     ; 将位6置为1
ORR   r6,r7,r6     ; 置位
MSR   c_spsr, r6   ; 重新载入寄存器
```

域的位在逻辑上是“或”在一起的，因此，举个例子，你可以使用cxsf_cpsr去修改cpsr的所有域。

MSR和MRS指令的格式如图11-10所示。如果位R = 1，则使用的程序状态寄存器是SPSR，如果R = 0，则程序状态寄存器是CPSR寄存器。立即数域以循环域所给出的次数值进行循环，使得相应位移动到PSR的标志位。

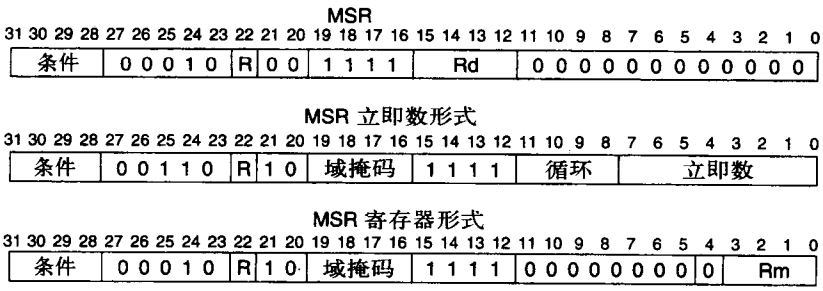


图11-10 ARM修改状态寄存器的指令格式

11.9 ARM系统向量

与68K和8086体系结构相比，ARM体系结构中的系统向量相对较少。如下表所示总共只有8个系统向量：

异常向量	地 址
重启	0x00000000
未定义指令	0x00000004
软件中断	0x00000008
预取异常中断	0x0000000C
数据异常中断	0x00000010
保留	0x00000014
中断请求	0x00000018
快速中断请求	0x0000001C

快速中断请求（FIR）向量是表中最后一个向量是因为该向量并不是显而易见的。回忆可知，每个向量都是32位的字，刚好能够装下一条指令。该指令通常是到用户服务子程序入口的转移指令。FIR向量位于表格的顶端以便FIR服务子程序能够从地址0x0000001C开始并从那里继续，而不需要添加一条转移指令来获得实际代码。如果想要速度快，每一个时钟周期都

很重要!

当处理器试图在没有权力访问某条指令的情况下从一个地址取得该指令时，就要使用预取异常中断向量。它被称为预取异常中断是因为实际的指令译码发生在指令被取之后，但是
319 异常实际上是在预取指令的期间发生。当我们在后面一章学习流水线处理器时，我们将更深入地学习这些内容。

数据异常中断向量除了是针对数据的之外，与预取异常中断向量相同。因此，数据异常中断发生在处理器试图在没有正确访问权的情况下从一个地址区域取数据的时候。

重启向量也是独特的，因为当重启中断被确认时处理器将立即停止执行并开始重启序列。对于其他异常，处理器将在接受异常序列之前完成当前指令。当然，这是非常有意义的，因为重启操作不需要恢复系统上下文，所以你就可以尽可能快地开始该指令。

总结

ARM指令集是一个全新的32位RISC指令集。与8086和68K处理器不同，除了一小部分指令在一个时钟周期内执行的例外情况，所有指令的长度都是相同的。寄存器集几乎完全通用。16个用户模式寄存器中只有3个是专用的。所有的数据处理指令都发生在寄存器之间，而且所有的存储器操作都被限制为存储器到寄存器的载入操作和寄存器到存储器的存储操作。所有的存储器访问都使用一个通用寄存器作为基址存储指针。附加的有效寻址模式通过增加递增寄存器、递减寄存器、变址寄存器、立即偏移值和比例寄存器进一步增强了这种模式。

虽然你可能并不同意，但ARM指令集体系结构确实比我们前面接触过的体系结构要简单得多，严格得多。这种简单性更加依赖于可以生成最优化代码流因而也是最高效代码的编译器。

通过这一章对ARM体系结构的概述，我们将搁下通用体系结构的学习而转移到其他话题。当在后面一章详细学习流水线时我们将再一次回到对体系结构的学习。那时，我们将再一次回到ARM体系结构，但是我们下一次将在一个更高的层次讨论问题。虽然你们中的有些人在阅读这些章节的时候可能已经发现，这就如同看着绘画变干一样令人兴奋，但也有使人发疯的方法。为了从软件的角度来理解计算机的体系结构，我们必须学习一些关于很多位模式是怎样被用来形成指令字的例子。

美国公共电台的演员Science博士曾经说过："我喜欢读取一列列的随机数字并从中找出规律。"

本章讲述了以下内容：

- ARM体系结构的发展简史
- ARM7TDMI处理器体系结构的概述
- ARM指令集和寻址模式的介绍

参考文献

[1] Jim Turley, *RISCy Business*, Embedded Systems Programming, March, 2003, p. 37.

[2] *Ibid.*

[3] ARM Corporate Backgrounder, http://www.arm.com/miscPDFs/3822.pdf, p. 1.

[4] Andrew N. Sloss, Dominic Symes and Chris Wright, *ARM System Developer's Guide*, ISBN 1-55860-874-5, Morgan-Kaufmann, San Francisco, CA.

[5] Dave Jagger, Editor, *Advanced RISC Machines Architectural Reference Manual*, ISBN 0-13-736299-4, Prentice-Hall, London.

320

[6] Steve Furber, *ARM System-on-chip Architecture*, ISBN 0-201-67519-6, Addison-Wesley, Harlow, England.

[7] Andrew N. Sloss, Dominic Symes and Chris Wright, *ibid*, pp. 143–149.

习题

1. ARM系统的操作模式是什么？与68K相比较它们是怎样的？
2. 为什么要有快速中断请求模式？它是如何实现的？
3. 将ARM体系结构中的16个基址寄存器和68K体系结构中的16个寄存器进行比较。
4. 指令

   MOV r4, #&103

   是合法指令还是非法指令？为什么？注：&103是16进制数的ARM表示。
5. 写一段代码将立即数值&103载入寄存器r4。
6. 用值&06AA4C01初始化寄存器r7。
7. 假设寄存器r8的内容 = &0010AA00且寄存器r6的内容 = &0000CFD3。当下面的指令执行完后寄存器r11中存储的值是什么？

   ADD r11, r8, r6 LSL #2
8. 将下面的68K指令重写为等价的ARM操作。提示：不要忘记标志（flag）。

   ADD.L D3, $00001000
9. 假定<r1> = &DEF02340。尽可能详细地描述下面指令执行的操作： 321

```
LDRNEH  r4, [r1, #4]!
```

第12章　与外部接口

学习目标

- 描述为什么中断是计算机和外部世界相互作用所固有的；
- 解释为什么中断要设定优先级；
- 理解I/O端口的概念；
- 解释模拟信号和数字信号之间如何进行相互转换；
- 理解模拟到数字转换过程中，速度和精确性的折中问题。

12.1　引言

在前面课程中，我们看到只能在自身环境内运行而不能与外部世界交互的计算机是相当无用的，虽然这对于学习体系结构是一个好的环境，但其所有益处也仅限于此。当你能够用TRAP #15指令将输入和输出功能（I/O）加入到你的程序当中时，会有某种耳目一新的感觉（我希望）。现在，让我们通过图12-1开始对计算机和外部世界的讨论。图中虚线内的部分表示计算机运行所需要的最少量的组件，虚线外部表示计算机能做有用工作所需的所有其他部分。

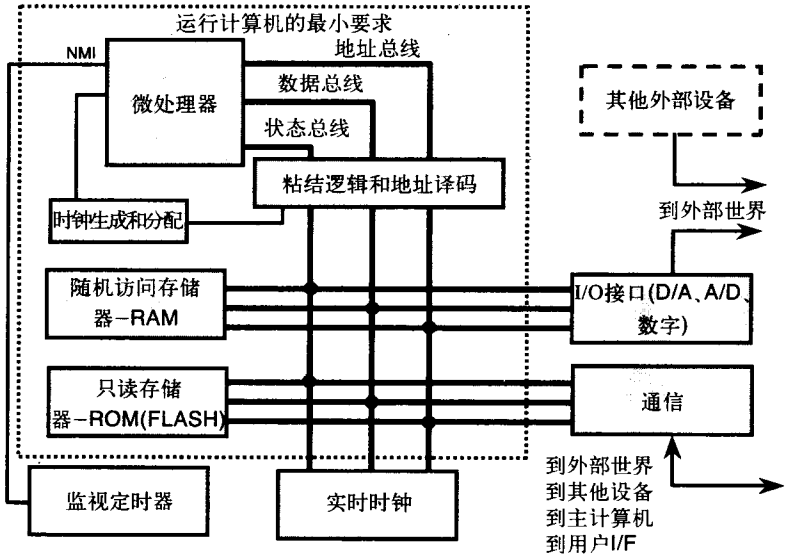


图12-1　一个典型的计算机系统。显示在虚线内的功能模块是使计算机能实际运行的最小需求

如你所见，处理器、存储器阵列、粘结逻辑（存储器译码等）以及时钟构成了基本的计算机，但这个计算机相对人而言是无价值的。我们需要能以某种方式与外部刺激进行交互（接口）。

外部世界（真实世界）有其自身的一组约束，我们必须能够处理它。与你的PC主板相比，外部世界是非常杂乱的。其中的一些约束是：

- 外部事件在本质上通常不是数字的；
- 与计算机中的基本时钟周期相比，外部事件的发生频率有很大差异；
- 外部事件是短暂的，如果未在适当时间内得到服务，事件可能会消失；
- 外部事件通常发生于脏、湿、极热、极冷、或者有大量背景噪声（电干扰）的环境； 322
- 服务于外部事件的计算机的失效有外部的后果。关键系统可能会失效（2000年问题），或者可能有人丧生。

如果我们要接受一种事实，即有必要从外部的混沌中制造出某种秩序，那么我们首先就需要了解外部和计算机之间是如何相互通信的。由于事件发生的频率有很大差异，所以我们需要能使我们在计算机环境的内部和外部之间进行同步的方法。我们要讨论的第一种方法就是中断的概念。

12.2 中断

到此我们已经在处理器和存储器方面考察了计算机系统。加入一个时钟后，这就是一个功能上的计算机了，但却是一个无用的计算机。在前面的课程中，你可能会注意到中断（interrupt）这个词偶尔在这里或那里闪现，现在，我们就来了解一下中断到底是什么。回忆可知，当在前面章节学习计算机体系结构时，我们实际上提到了中断。特别地，我们看到了ARM体系结构和它的中断以及它的快速中断请求模式，我们还看到了软件中断和它们如何改变处理器的上下文环境以及访问操作系统。现在，让我们返回来考察中断过程本身。

为了使计算机值得你为其付出电力成本，它就必须能与你和环境相互作用。你操纵键盘和鼠标，计算机以屏幕动作、声音、磁盘访问等作为响应。如果有时发生了被称为异常（exception）的意外事件，计算机必须有能力来对付它们。一个典型的异常可能是漂浮不定的指针所产生的结果，它导致程序试图从实际上并没有存储器的区域中去取数据。或者，你试图用零去除。

当处理器内部和外部的异步事件需要吸引处理器的注意时，它们通过生成中断来达到此目的。中断强制处理器中止其正常的程序执行，并开始执行另一个称为中断服务程序（interrupt service routine，ISR）的代码块。ISR程序代码结束后，中断已得到处理，处理器返回到离开的地方并恢复应用代码的执行。 323

假设我们没有中断，处理器为了照顾到这些事件，它就不得不周期性地检测每个可能请求服务的事件，看看事件是否准备好接受服务。一个恰当的类比就是电话铃。你正在吃晚餐（应用），电话铃响了（中断），你放下叉子接电话（ISR），你告诉电话推销员你不想要这个去纽芬兰里维埃拉度假胜地的免费度假的机会，然后你回来继续就餐（从中断返回）。

现在，假设你的电话铃坏了。你要得知是否有人给你打电话的唯一办法就是每过几分钟就拿起电话说："喂，喂，有人吗？"在计算机中这个类似的过程称为轮询（polling），它很快就变得过时了。在一个轮询系统中，计算机周期性地检查每个可能请求服务的事件，这将作为其常规程序代码的一部分。当应用服从于轮询结构时，轮询就仍然是一种非常可接受的计算机编程方式。防盗自动警铃控制器就是轮询系统的一个极好例子。程序依次检查每个传感器，看看是否被夜贼或家猫绊住。如果一个传感器被绊住，程序就打开报警铃。程序在称为轮询循环（polling loop）的连续循环中运行，并同时检查传感器。

你可能会想像到，当一台计算机在轮询其外部设备看是否需要服务时，它就不能多做别的事情，这就是我们需要中断的原因。因为中断是异步的，所以其发生的多与少有些随机性，因此处理器就"按需"对其进行处理。然而，我们不应无视这样的事实，就是中断既可能随

机发生，也可能在精确的时间间隔发生。例如，你的Windows操作系统有外部定时器每隔几毫秒就提供时钟滴答，而其他系统可能有规律地在每几微秒就有中断出现。关键的一点是中断不与我们程序的执行同步。现在，让我们考察一下你在一个计算机系统中可能会遇到的一些通常的中断类型。

最常见的中断是复位（$\overline{RST}$）。复位是一个非常引人注目的中断，它从头启动处理器。它不像其他中断那样返回到它在应用中离开的那一个点。复位中断假设任何事情都是可疑的，你真正想做的是从起始点开始。当你通过在计算机上按复位健而使复位有效时，就引起了以下事件序列的发生：

1. 清除内部寄存器的内容；
2. 将处理器确立在一个已知的状态；
3. 从一个已知的存储器位置开始执行程序。

请注意，上述过程是复位中断的一般动作序列。如前所见，不同处理器用不同的方式启动。现代奔腾和Athlon CPU在复位有效时，有非常复杂的启动序列。

如果你在硬件级别考察复位中断，你就会惊讶地发现，为了使复位工作完全，你必须在
324 相当多的时钟脉冲内保持复位输入有效，这就是算法状态机在幕后忙碌的暗示。典型情况是，为了将状态机引入到正确状态，复位输入可能要在50个到几百个时钟周期内保持有效。

有时我们关心，为进行中断服务我们可以用多长的时间。如果我们正试图捕获和处理一个快速的数据流（比如数字视频便携式摄像机），而且我们不想丢失任何帧，那么我们就可能会给这个中断以一个较高的优先级。

另一个因素可能对于中断至关重要。大多数膝上计算机都有一个高优先级中断，它由监视电池电量水平的电路所驱动。当电池几乎已经丧失其对计算机的供电能力时，一个高优先级的中断服务程序将自动接管并保存计算机的状态，使得你能关闭计算机并在电池充电后恢复。谈到中断的至关重要性，最高优先级的中断机制通常保留给对人员生命的保护。

由于较高优先级中断能屏蔽（mask）来自较低优先级的中断信号，所以，如果较高优先级的中断正在被服务，那么较低优先级的中断就不得不等待。这就是为什么当你的PC正忙于磁盘驱动时，你在Windows中会看到沙漏形状的符号。Windows在以这种形式告诉你，正在读磁盘数据，鼠标正在等待轮到自己。

在很多情况下，尤其是对于像Windows和Linux这样的操作系统，计算机和操作系统用来响应中断的时间是不可预知的，也就有可能不是足够地快，不能保证在分配的时间内可靠地服务所有的中断。为了使基于计算机的系统在处理外部事件时工作可靠，一种不同类型的操作系统被设计出来。能够处理以外部的时间帧发生的外部事件的操作系统被称为实时操作系统（real-time operating system, RTOS）。与Windows不同，RTOS不是一个平等主义的操作系统。个人PC上的操作系统是以循环的方式对任务进行调度的，每个被执行的任务都能从操作系统（通常简写为O/S）那里得到一个时间片，这与该任务有多么“重要”无关。有时，O/S会在幕后给在进行输入和输出的任务更多的时间。然而，关键是我们无法准确地预测，在所有条件下，是否所有的任务都能以其重要程度的次序来执行，而且所有中断都能在要求的时间间隔内得到服务。当然，用更快速的计算机总是有好处的，但有时经济上的现实问题更突出，使得这成为不可行的选择。

RTOS所采用的调度机制与PC的O/S截然不同。一个准备好运行的较高优先级的RTOS任务将总是抢占当前正在运行的较低优先级的任务。而且，O/S的核心软件都是谨慎设计的，能使任务之间进行切换所花的时间最小化。这就是为什么ARM体系结构具有一个快速中断请求

模式和寄存器堆的原因。两个特性都是对体系结构的增强，用以加快处理器对苛刻请求的响应时间，而这些请求以适时的方式对中断进行服务。

图12-2是在RTOS控制之下运行的计算机系统的行为图。x轴以秒来度量，显示出的每个时间刻度是100μs。y轴显示出系统的各种任务和中断。中断和任务以优先级下降的顺序排列。最高优先级的中断Int_1比Int_2有更高的优先级，而Int_2又比其他任何任务都有更高的优先级。最低优先级的任务就是系统处于空闲状态。

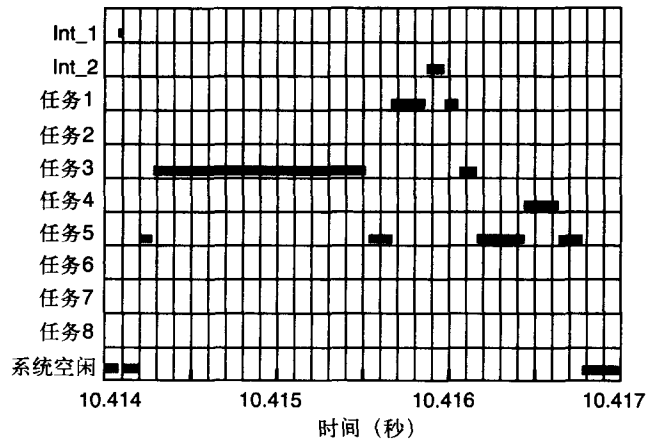


图12-2 一个典型的实时操作系统的中断和任务切换响应图。X轴跨越了3ms的执行时间

请注意每个较高优先级的任务是如何抢占较低优先级的任务的，直至较高优先级的任务完成或者在继续进行前必须等待某种信息。请看标记为任务3的任务，当任务3开始运行时，它抢占了任务5，并持续运行直至在大约时间=10.4155秒处才停止。在这个时刻，任务5再次开始，直至再次被抢占，这次是被最高优先级任务1所抢占。

任务1运行，但被Int_2抢占。当Int_2的中断服务程序完成时，任务1再次开始，然后结束，使得任务3再次接管。当任务3最终结束执行时，任务5就能开始执行了。然而，任务4突然活跃起来并抢占了任务5。最后，任务4结束后，任务5就能完成运行了。当任务5完成时，系统返回到空闲状态。

如果这一切看起来非常复杂就对了。甚至在这个“相对简单”的例子中，你也能发现潜在的问题。如果任务5持续不断地被较高优先级的任务和中断所抢占而总也不能运行完怎么办呢？这的确是一种可能性。我们也可能遇到某种事件序列，导致系统锁住。其中一种这样的情况就是*优先级颠倒*（priority inversion）。优先级颠倒发生的原因是：较低优先级任务本身被抢占，而较低优先级任务又保留了对一种系统资源的控制权（比如存储缓冲器），但较高优先级的任务却要使用这种资源，这样就使较高优先级任务被悬挂而不能运行。一个非常有趣的优先级颠倒问题发生于*火星探测器*（Mars Rover）计划中[1]，就是在探测器刚刚停在火星上的时候。在帕萨迪纳的任务控制人员用一个在喷气推进实验室的实验场中运行的相同探测器进行了模拟，他们发现探测器的RTOS由于优先权颠倒而锁住了。最后他们得以纠正了这个问题并将新代码上载到探测器的基于8086的计算机系统中。

从以上例子你已经看到，由于中断是异步的，所以能彼此中断。一个处理器可在处于中断服务程序中时，允许另一个中断进入。处理器怎么做呢？为处理这种情况，通常对中断进行优先权化。一个更重要（较高优先权）的中断可以总是抢占一个较低优先权的中断，就像一个较高优先级的任务可以抢占一个较低优先级的任务一样。高优先级中断和低优先级中断都有哪些例子呢？如果一个较高优先级的中断正在被服务，则较低优先级的中断就得等待，因为较高优先级的中断能*屏蔽*来自较低优先级中断的信号。回忆可知，ARM处理器没有优先级化的中断，两个中断FIR和IRQ有相同优先级，它们若在PSR中被允许，就可以总有效。

然而，在大多数计算机中总是有一个最高优先级的中断。这是一种不能被忽视的，必须总是立即得到服务的中断，这就是*不可屏蔽中断*（nonmaskable interrupt, NMI）。一般来说，我们将NMI保留给灾难性事件，如系统供电停止、检测到存储器失效等。在飞机上，NMI可能保留给即将发生的有生命威胁的情况。从一旦被服务完就返回到你代码中首次发生中断的那一点的意义上说，NMI是一个真正的中断。

余下的中断是优先权化的，可通过各种外部和内部方法对其赋予优先级别。我们在本课本中不关心这是如何实现的，只是要了解如果较高优先级的中断正在被服务，则较低优先级中断就必须要等待。在Motorola 68K体系结构中，最低优先级中断就是优先级-1中断，NMI中断是优先级-7中断。当一个中断正在被服务时，只有具有较高优先级别的中断才能接管控制。还要注意，我们不对不可屏蔽中断赋予优先级别，因为它是硬连线到处理器的状态机中的。

12.3 异常

异常与中断相似，但异常是由与程序相关的事件生成的，如存储器访问错误（那里没有存储器）、非法指令（指针错误）、用零除错误或者其他需要特殊处理的程序故障。从结构的角度看，除了不能被屏蔽以外，可以像处理中断一样处理它们。如果有异常生成的情况发生，异常处理过程立即开始。

12.4 Motorola 68K的中断

Motorola 68K处理中断的方式十分标准，所以我们将采用这种体系结构作为原型并花一些时间来讨论它。在68K处理器的地址空间中，前1024个字节的存储器是保留的，用于处理异常和中断。因此，从0x000000到0x0003FF之间的字节地址就是为处理中断和异常保留的，用户程序不应该从地址0x000400以下开始。我们将这些地址中的每一个称为*向量*（vector），因为向量指向了程序代码，而程序代码就是被设计用来服务于异常的。换句话说，与这些中断向量关联的存储器位置的内容本身就是地址，就是中断服务程序在存储器中的地址。

在每个长字存储器位置，程序员放置的是相应的中断服务程序或异常的第一条指令。例如，一个不可屏蔽中断（NMI）的中断服务程序存于地址0x00007C。处理器执行如下序列来响应该NMI：

- 完成当前指令
- 将状态寄存器的一个拷贝存入堆栈
- 将下一条要执行指令的地址存入堆栈
- 切换到管理模式
- 取出存放在存储器位置0x00007C的32位数据
- 从存于0x00007C处的存储地址开始执行中断服务程序

327

这种类型的寻址也称为*间接寻址*（indirect addressing）。存储于某存储器位置的数据就是实际上的真实数据的地址。在C和C++中，我们称之为指针。这样，存储器的前256个长字就被保留为*系统向量*（system vector），这些系统向量是指向某些存储区域的指针，而这些区域存储的是实际的异常处理代码或中断服务程序。

在存储器中的前两个长字地址0x000000和0x000004有特殊的重要性。复位信号有效后，68K将取出在0x000000的向量并将其放入堆栈指针寄存器中。接着它将取出位于0x000004的向量并将其放入*程序计数器寄存器*（program counter register）中。然后，它将在程序计数器

寄存器中所存储的地址处开始执行程序。

中断告诉我们什么时候一个外部世界任务需要服务，但我们如何实际地与外部世界交换真实数据呢？在与外部世界接口中，最基本的任务之一就是将计算机中的事件与外部世界中的事件进行同步。计算机中的事件可能每秒几亿次，而外部世界中的事件在几个小时时间都可能不变化。同步这两个时间尺度的最简单方式之一就是用D型触发器作为存储寄存器，D寄存器或锁存器的典型应用就是用来将外部世界事件同步到处理器总线。当它们用于与外部世界接口时，我们称之为I/O端口（I/O port）而不是寄存器。 328

你已经看到Intel x86系列将I/O端口看作是独立的寻址空间，而ARM和68K体系结构将I/O设备看作是处理器存储空间的一部分。因此，8086就有一个独立的汇编语言指令，用于对I/O空间进行读和写，而不是对存储器进行读和写。有时，与存储器空间相比，I/O空间的时序没那么严格，地址译码也比较容易。将I/O设备映射到处理器的存储空间后，就产生了较简单的指令集，因为I/O传输就与存储器传输相同了。在这两种系统中，中断和状态寄存器用于通知什么时候数据可用。

让我们考察一下一个简单的8位I/O端口的操作，来看看我们如何将计算机和外部世界接口。图12-3就是一个I/O端口的简单示意图。

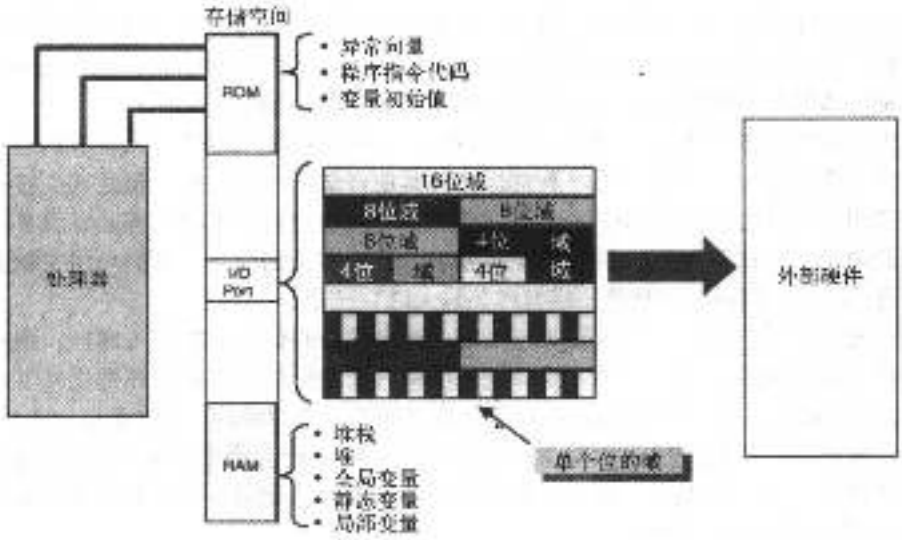


图12-3 微处理器存储空间内的I/O端口的总体观察

I/O端口可以是单个的端口，也可以是全部I/O的连续块。例如，在你PC内部的图形芯片可能有一百个或更多的I/O端口（称为寄存器阵（register map））与图形环境的数据传送和控制相关联。在图12-3中，我们看到这个I/O设备被分为独立端口，端口是单个位、4位或更多位宽的域。每个端口按其要执行的I/O功能进行了配置。

让我们考虑如图12-4所示的更特殊的电路。图12-4中的I/O端口在计算机中呈现出两个连续的存储器位置。实际形成I/O端口的器件有两个部件与其关联。位于偶数地址（A0=0）的部分是I/O端口本身，位于奇数地址（A0=1）的部分决定了该I/O如何使用。对于大多数I/O端口来说都有一个约束，即它们不能同时既输入又输出。我们必须将器件的各个位编程为输入或者输出。

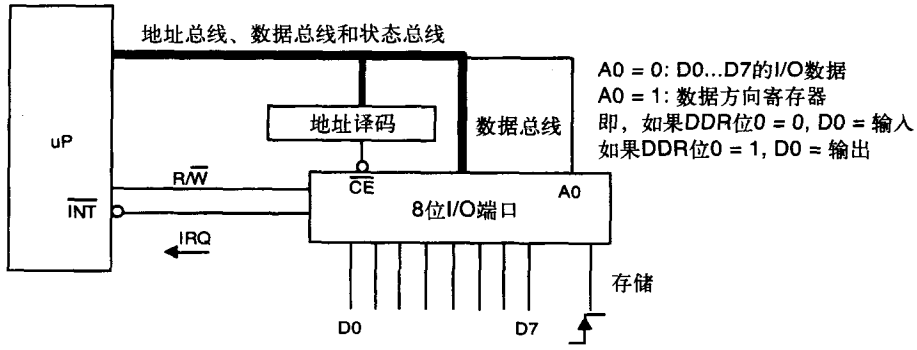


图12-4 8位I/O端口

地址译码模块决定了I/O端口在处理器地址空间的哪个地方起作用。例如，假设端口分别占据着字节地址$00A600和$00A601。实际的I/O端口出现在偶数地址，控制寄存器出现在奇数地址，我们称这个控制寄存器为*数据方向寄存器*（data direction register, DDR）。

当我们向DDR写入时，我们就是在逐个位地对I/O端口的配置进行编程。任何位的位置，若被写入“0”，就使得I/O端口的相应位成为输入，反之若被写入“1”就是输出。当然，使“1”对应输入和使“0”对应输出更有道理，但是硬件设计者就是这样一伙开玩笑的家伙。因此，若将$FF写入DDR就会使I/O端口的所有位成为输出位，而若写入$00，就会使所有位成为输入位。将$AA写入DDR就会使奇数位成为输入而偶数位成为输出。

假设我们将DDR编程为$FF，我们现在就有了一个8位的输出端口。从计算机这一边，我们可以向这个寄存器写一个值，就好像它是任何其他存储器位置一样，但是我们接着就会在I/O端口的输出一侧看到这个数据。而且，这个数据是持久性的，如果不停止计算机的话它就能在端口的输出一侧保留数天。我们现在就有了一个数字控制信号，我们可用它做我们想做
329 的任何事情。现在，假设我们想将该端口作为输入端口使用。

当外部数据出现在端口时，该数据必须在时钟上升沿的作用下被写入端口。现在，数据就被存入到了端口的输入部分，但一般来说，计算机无法得知数据是否在那里或者存在那里的数据是否已经改变。这是I/O端口和存储器位置之间的一个微妙的、非常重要的不同。同样，如果我向存储器写一个数据值，然后立即将其读回，我预计会看到我刚写入的数据。然而，对于一个I/O端口，我写入端口（输出）的数据通常不是我从端口（输入）读来的数据，因为端口的输出和输入部分是不同的。

对于存储器，我们假设存在那里的数据不会变化，除非我们以某种方式改变了它。我假定你们都熟知创建全局变量的危险，即全局变量非常容易被出乎预料的方式改变，程序员在同它们打交道时要尤其警惕。I/O端口变量怎么样呢？我们有同样的问题，甚至更坏。对于I/O端口，我们必须假设存储在端口（输入侧）的数据可自发地变化，而且我们所习惯遵循的所有存储器完整性规则可能再也不会有效了。

由于对I/O端口的操作常常就好像它们是存储器一样，但实际上I/O端口不是存储器，所以编译器就必须有关于如何处理I/O端口的专用指令。告诉编译器不要假设一个存储器位置（变量）是简单存储器的一个最一般的方式就是使用volatile关键字。例如：

```
volatile  unsigned short int * foo;
```

告诉编译器，foo是一个指向正的16位存储器变量的指针，该变量可能在无警告的情况下自发地改变值。这意味着编译器不应对包含该指针的代码做任何优化的设想。被指针废弃引用

(dereferenced)的值不应被赋值到寄存器或进行其他任何形式的优化。这个指针总是用于改变存储器的值或读该值，仅此而已。

作为一个例子，让我们考虑一个现实中的I/O端口。我们将要考察的例子是一个通用异步接收器/发送器，即UART，你可能不太知道这个名字，因为很有可能你是通过将其作为你PC上的一个串行通信端口（com port），如COM1或COM2而认识这个词的。一个串行通信端口就是串行的发送和接收装置。假设你的计算机中有一个UART，硬件设计者已经将其设计成在存储地址$006000处映像而成的16位宽I/O端口。下面是操作规范：

- 8位串行数据（一个字节）被发送，并在位DB8~DB15处被接收。
- DB0~DB7提供关于设备的状态信息。
- DB0－数据就绪（DR）状态位。
- DR=1意味着数据可用，可从DB8~DB15读取。
- DR=0意味着目前没有数据等待被读取。
- DB1＝发送缓冲器（TB）状态位。
- TB=0意味着目前发送器缓冲区空闲，数据可传送到其中以便发送。
- TB=1意味着目前发送器缓冲区正在发送数据，新数据不应向其写入。

考察以上规范，我们知道：

- 当DR=0时，如果我们读DB8~DB15，我们不能确保数据是有效的。
- 当DR=1时，如果我们读DB8~DB15，我们将读到有效数据，UART将自动将DR复位到0。
- 当TB=1时，如果我们向DB8~DB15写，那么我们就可能会覆盖正在被发送的数据，从而破坏这些数据。 330

图12-5显示出处于计算机内部的UART。

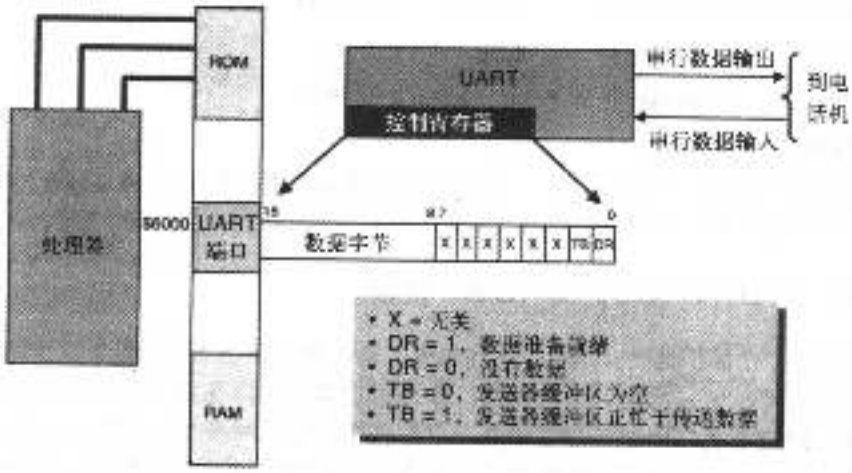


图12-5 UART设备的示意图表示，在这个例子中，UART从计算机向拨号调制解调器发送串行数据

让我们看两个68K汇编语言代码的片段，这两个片段说明了我们如何对UART进行读和写。注意，在这个例子中，我们将采用轮询循环而不是中断服务程序。这使得对位编码更易于演示，虽然你立即就会看到这是多么的低效。一个典型的用于拨号连接的50K波特（bps）的调制解调器每秒发送和接收的字符大约为5000个。我是怎么知道这一点的呢？波特（BAUT）是电话业中每秒位数的行话，在这个传输限制下，需要10位来传输8位字符，所以50K波特的

真实情况是每秒5000字节的实际数据流。

如果我的计算机正以1GHz的时钟速率运行，则在UART发送一个字符的时间里，就发生了200 000个计算机时钟脉冲。如果计算机能平均大约每两个时钟脉冲执行一条指令，则在它等待UART发送一个字符期间就应该能执行大约100 000条指令。至少通过中断，它可在等待时做一些其他事情。但是，由于采用轮询循环，它就要围绕这个循环运行几万次来等待UART结束。哎……

* 一个用于测试串行数据准备好被读的短程序

```
START    LEA      $00006000,A5      * 装入地址
LOOP1    MOVE.W   (A5),D0           * 得到UART的状态
         ANDI.W   #$0001,D0         * 测试DR
         BEQ      LOOP1             * 保持等待
```

* 一个用于看数据是否可被发送的短程序

```
START    LEA      $00006000,A5      * 装入地址
LOOP2    MOVE.W   (A5),D0           * 得到UART状态
         ANDI.W   #$0002,D0         * 测试TB
         BNE      LOOP2             * 保持等待
```

我们一直在拘泥于用汇编语言来阐明计算机体系结构的各个方面。然而，大多数程序员是用高级语言编程的，所以，人们自然会问到C++程序员如何处理由I/O端口所施加的硬件限制。记住，I/O端口处在存储器中的固定地址。C++编译器通常希望能够管理存储变量的空间分配（或者调用操作系统作为协助），因此，为了将变量分配到特定的存储地址，我们就不得不降服编译器。

331

让我们看一个具有相同汇编代码的例子，但这次用的是C++语言。

```
char *p_status; // 指向状态端口的指针
char *p_data;   // 指向数据端口的指针

p_status = (char*) 0x6001; // 将指针分配给状态端口

p_data =   (char*) 0x6000; // 将指针分配给数据端口

do { } while (( *p_status & 0x01) == 0 ); // 等待

char inData = *p_data;
```

为了将指针分配到I/O设备地址，我们必须明确地将指针强制转换（cast）到0x6000和0x6001。在C++用于嵌入式系统编程时，这种对硬件进行寻址的方法相当典型。

12.5 模数（A/D）转换和数模（D/A）转换

外部世界是一个模拟世界。这是什么意思呢？这个意思就是外部世界中发生的物理事件可能呈现无限个可能的值。风速、户外温度、阳光强度都是可以在某个取值范围内光滑地、无限地进行变化的物理量。例如，在地球上，户外温度可能在冬季南极点的−100华氏度到夏季中午死亡峡谷的+140华氏度之间的范围变化。虽然我们通常以整数度数来报告温度，但没有什么能阻止采用72.56435791度这样的精度。问题是我们如何精确、快速、低成本地测量温度。

为了测量这些物理量，我们首先需要一个电路或装置来提供该物理量的一个电气上的“类似物”。模拟电子学正是可用来代表平滑变化的电信号的电路，该电信号就是物理量的类比物。例如，我可能想控制炉子内部的温度，为了精确地保持一个固定的温度或者温度随时间变化的轮廓，我首先需要能测量炉内温度。一般是用热电偶或铂电阻温度计（PRT）来测量炉内温度。然后，其他电路将测量的温度作为输入信号，将对火炉功率进行控制的驱动作为输出信号。

PRT有一个电阻值的范围，这个电阻值可从房间温度中的100欧姆到800华氏度中的2000欧姆之间平滑地变化。我们可以说PRT有一个每度2.6欧姆的*传递函数*（transfer function）。重要的一点是，对于温度这个平滑变化的物理量，模拟电子装置能给出平滑变化的电信号，这是模拟电子学的精髓。虽然不是所有时间，但大多数时间我们想将物理量转换为模拟电压。这也许要采取几个步骤，因为我们使用的传感器可能首先将物理信号转换为电流或电阻。而且，这些信号的量值可能过小，不使用某种放大器就测量不了。

计算机是一个数字装置，它能处理被称为位的以离散量表示的数字。一个8位的数字能给出从0~255或者从−128~+127的范围，一个16位的数字能给出从0~65 535或者从−32 768~32 767的范围，依此类推。这样，问题就变为："我们如何从一个模拟值转换到与其等价的数字值，又 332
如何从一个数字值转换到与其等价的模拟值？"前一个过程称为*模拟到数字转换*（analog-to-digital conversion），即A/D转换；后一个过程称为*数字到模拟转换*（digital-to-analog conversion），即D/A转换。

在你的计算机中就有A/D转换器和D/A转换器。它们在哪里？你的声卡能将声音数字化（A/D）并将其存为*.wav文件，你可以通过扬声器播放这个*.wav文件（D/A）。输出到监视器的视频由快速变化的模拟电压组成，这些电压驱动监视器内部的红、蓝、绿发射"枪"。下面是一些我们转换成电压的物理量的例子：

- 热电偶：温度的度量
- 电阻温度计：温度的定义
- 应变仪：对位移或压力的度量
- 麦克风：声音级别的度量
- 磁性拾音器：对磁盘数据的度量
- 光电池：对光强度的度量

如你所想，我们越想要精确地度量模拟电压，就越要消耗更多的时间和费用。这样，当谈论A/D转换时，我们倾向于将注意力集中在做模拟到数字转换的分辨率和速度上。一个有12位分辨率的A/D转换器可能需要10μs将一个模拟信号转换成数字表示。这意味着什么呢？假设我们有兴趣要测量的物理量可转换成一个从0V到10V之间的模拟电压，0V是物理量能度量到的最低可能值，10V是我们可用来度量这个物理量的最高电压。注意这覆盖了一个范围的值。因此，假设要测量的物理量是温度，我们的模拟电路就可做如下设计：

1. 如果低于100华氏度，则模拟电压是0V。

2. 如果温度在100到500华氏度之间，则模拟电压随温度在0V到10V之间线性变化。这时的传递函数是每伏40度。

3. 对于超过500华氏度的所有温度，模拟电压保持在10V。

现在，我们的12位转换器有从0~4095的2^{12}个可能值，所以10V的模拟值范围可划分成4095个片段。数字信号的每个递增就代表10/4095的变化，约2.44×10^{-3}伏/位。这样，在我们注意到温度变化之前，温度可能就变化了（40度/伏）×（2.44×10^{-3}伏/位），即大约0.01度。

如果我们采用的是一个8位A/D，那么在我们能察觉任何变化之前，温度可能已变化了1.5度。这样，A/D转换器的分辨率就是数字输出的位数和代表输入跨度的模拟电压范围的函数。

作为讨论外部接口的切入点，让我们做一个外部情况的实例研究[2]。这个实例恰好是我在 333
2000年变更期间经历的。虽然只有一些孤立的2000年问题实际地发生了，但这并不意味着2000年问题未曾真实存在过。事实上，这个问题及其潜在的具有生命威胁的故障，都确实是真实的。

我们将要研究的情况是一个核电站的监视和控制系统的一部分。这个核电站的2000年程序管理小组仔细查看他们的计算机系统，在系统中寻找潜在的与2000年相关的弱点。假设我们作为高薪雇用的顾问，被要求帮助他们确定其系统是否存在2000年问题。我们将会做一个分析，并向2000年程序经理及其小组做一个报告。

图12-6是这个报告中最重要的一张幻灯片，让我们看一下。从图12-6中，我们看到数据集中器是一个嵌入式计算机系统，用来监视大量的遥感设备（大约256个），然后定期地用这些遥感传感器的状态来更新中央计算机系统。它是通过一个RS-232数据回路发送一个ASCII字符流来做到这些的。

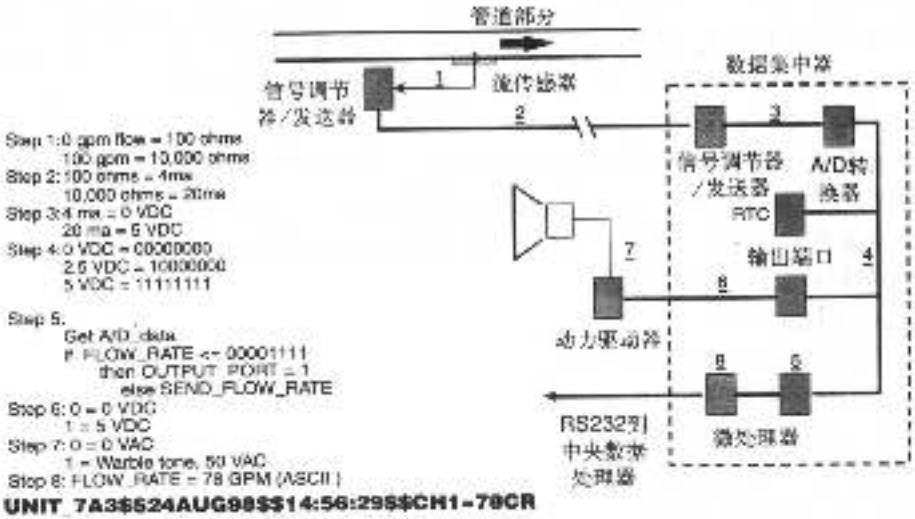


图12-6 核电站的一个水冷却管道中的数据集中系统的示意图。粗体字符是局部数据集中器向核电站中心计算机发送的串行数据流

我们感兴趣的那个特殊的传感器位于图的顶部。我们想监视管道中水流的速率。水的作用是保持反应堆冷却。如果水停止在管道中流动，我们就想关闭反应堆，并在对人员生命产生任何损害和威胁前对员工发出警告。

让我们就好像在核电站中一样实际地进行一下这个过程：

1. 位于水管中的一个传感器测量管道中的水流速率。传感器的传递函数将管道中的水压转换成电阻。对于这个传感器，我们已经知道传感器中100欧姆的电阻意味着管道中没有水在流动，10 000欧姆的电阻意味着水的流速是每分钟100加仑（G/M）。传感器的电阻在100欧姆
334 和10 000欧姆之间呈线性变化。我们可以将流速表示成传感器电阻的方程：

流速(G/M) = 0.01 (电阻) −1

2. 由于传感器位于距数据集中器几百米以外的地方，所以有必要将电阻值转换成电流值。确切的原因是电气工程师保密誓言的一部分，所以我不能说，只要相信我说的就行了。在工业环境中一个标准的传输方法就是用4~20mA的电流环，这意味着我们可采用从4mA的低值到20mA的高值之间的任何电流值。这个转换是由位于传感器附近的信号调节器/发送器来完成的。4~20mA发送器用另一个传递函数转换电阻：

100欧姆电阻→4mA

10 000欧姆电阻→20mA

3. 在数据集中器中，来自水管的信号被接收为4~20mA的电流环。并再一次经过转换过程

成为另一个信号调节器/接收器中的电压。现在，转换函数就是：

4mA→0V

20mA→5V

所以，经过3个转换过程，我们就能说，数据集中器的0V意味着没有水流，5V意味着每分钟100加仑的水流。

4. 在数据集中器内部是一个具有8位分辨率的模拟到数字的转换器。这意味着我们有256个单独的数字码，从$00到$FF，用来表示0V到5V范围内无限多个可能取值的模拟电压。这意味着什么呢？假设模拟电压是0V，很显然数字码应该是$00。现在，让我们缓慢地提高模拟电压，什么时候数字码会从$00跳转到$01呢？这几乎不可能严格地给出答案，但是如果将5V的模拟电压范围用可能的数字范围来进行划分，我们就会得到一些要领。这样，每个数字范围就是5V/255=0.0196V，所以，我们就应该能很粗略地预计，当模拟电压上升到大约0.020V左右时，A/D的输出将变为$01。这样，我们就在A/D转换过程中有了大约0.020V这样的不确定性，这也导致在管道实际流速上的不确定性。想要更精确些吗？就要使用有更多位分辨率的A/D转换器。我们还知道2.5V应该给出$80的A/D码。

5. 现在我们就准备好使用计算机了。计算机读A/D转换器的输出。如果数字码是$0F或更低，我们就有了严重的问题，因此转到步骤6，否则转到步骤8，因为如果该值大于$0F，就不需要发警报。

6. 数据集中器有一个输出端口，该端口能发送信号给警报器。向输出端口发送一个数字0或1将分别置端口的输出于0V或5V，这就成为警报器的一个输入。

7. 在警报器上一个5V的输入电压会打开警报器，警报器发出声音报警，核电站将立即进入紧急关闭模式。

8. 数据集中器将A/D转换器的输出转换成流速率，并与一些辨识信息捆绑起来，打上时间印戳，然后继续发送到控制室的计算机。来自集中器的数据流像这样： 335

UNIT_7A3$$24AUG98$$14:56:29$$CH1=78CR

这个消息解释如下：

1. 这是数据集中器7A3在报告。

2. $$是域定界符。

3. 日期是1998年8月24日。

4. 时间是14点56分29秒。

5. 在通道1读到的值是78加仑/分。

6. CR（回车）是消息定界符的结尾。

2000年问题在哪儿？请看第3个域值，这个日期被表示成了两位数字，它应该被解释成1998？还是2098呢？我们说不清楚，但部件7A3在其数据流中引入了含糊性，所以对这种含糊性必须做进一步的研究。陈述结束，我们将这个清单送到哪儿？

让我们再回到A/D转换。现今使用的典型的A/D转换方法有：

- 快闪转换
- 逐步逼近
- 单斜率
- 双斜率
- 电压到频率

快闪转换是最快、最昂贵和最不精确的，只有6到12位的分辨率。逐步逼近是成本、精确

度和速度的最佳平衡。在商业设备中可达到从12位到20位中的任何一个分辨率。单斜率是类型最简单的A/D，具有12位到14位的适度分辨率。双斜率（和三斜率）最精确，具有22位到24位的分辨率。电压到频率转换（即V/F）是最慢的，但比其他方法好的地方是，具有平均掉信号中噪声的能力。

由于A/D和D/A的电路设计通常是电气工程师的职责，所以我们在这里就不得不谨慎，不要厚着脸皮去刺激这些警惕性很强的人。我们要采取妥协的立场，使得你能够与电气工程师谈论有关你计算机系统中的A/D转换问题，但你不要知道得过多以威胁他们工作的安全。

我们需要通过学习被称为比较器（comparator）的一种重要电路元件来开始我们对A/D和D/A转换的学习。比较器存在于模拟和数字之间的下层世界中，其输入是模拟信号，输出是数字信号。见图12-7所示的电路。

该电路有两个输入（一个+输入和一个-输入）和一个输出。输入接受某个范围值的模拟电压。假设输入电压的范围是从-10V到+10V，即±10V。任何从-10V到+10V的模拟电压都可施加于+输入或者-输入端而不会对电路造成任何损害。

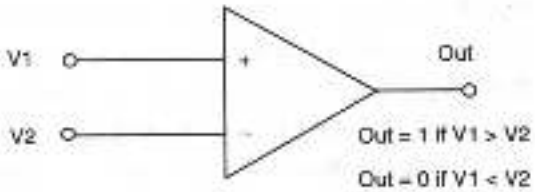


图12-7 模拟比较器

在-10V到+10V这个范围，如果在+输入

336 (V1) 的电压比在-输入（V2）的电压更正（more positive），则比较器的数字输出就是真（1）。如果-输入的电压比+输入的电压更正，则输出为假（0）。图12-8对这个行为做了图示。假设在图12-8中，标为"比较器阈值电压"的灰线被连接到-输入（V2），该输入保持0V的稳定值。进一步，假设一个具有正负峰值±5V的正弦波电压被连接到比较器的+输入（V1）。

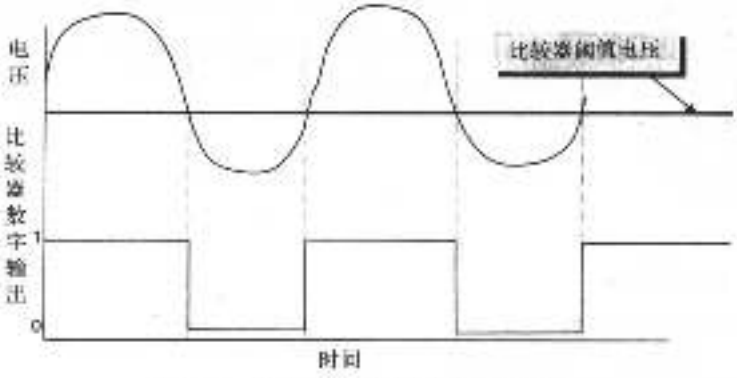


图12-8 图12-7中模拟比较器的传递函数

每次正弦波上升到参考电压（0V）以上时，比较器输出就上升到1。每次其下降到参考电压以下，输出就变为0。从某种观点看，比较器就是一个1位A/D转换器。我们知道未知电压是大于还是小于参考电压，但这就是我们知道的一切。实质上，比较器回答的是下面的逻辑问题：

输入V1处的电压比输入V2处的电压更正（more positive），是真还是假？

让我们稍微增加一点复杂性。现在假设正弦波电压在最小值0V和最大值4V（Vmax）之间变化。现在，让我们取来3个比较器，将它们标记为A、B和C。我们将所有比较器的+输入连接到正弦波电压，对每个比较器的-输入做如下连接：

337 1. 比较器A连接到Vmax的75%，即3V。

2. 比较器B连接到Vmax的50%，即2V。

3. 比较器C连接到Vmax的25%，即1V。

现在考虑图12-9的输出。如你所见，相同的未知电压同时施加到了3个相同比较器的所有+输入，然而，每个一输入连接的是等间隔的不同参考电压。这个电路告诉了我们更多东西，但仍很粗糙。例如，如果比较器A的输出是1，则我们知道模拟电压大于3V，但小于正弦波的最大值4V。如果比较器B的输出是1，但比较器A是0，则电压大于2V，小于3V。

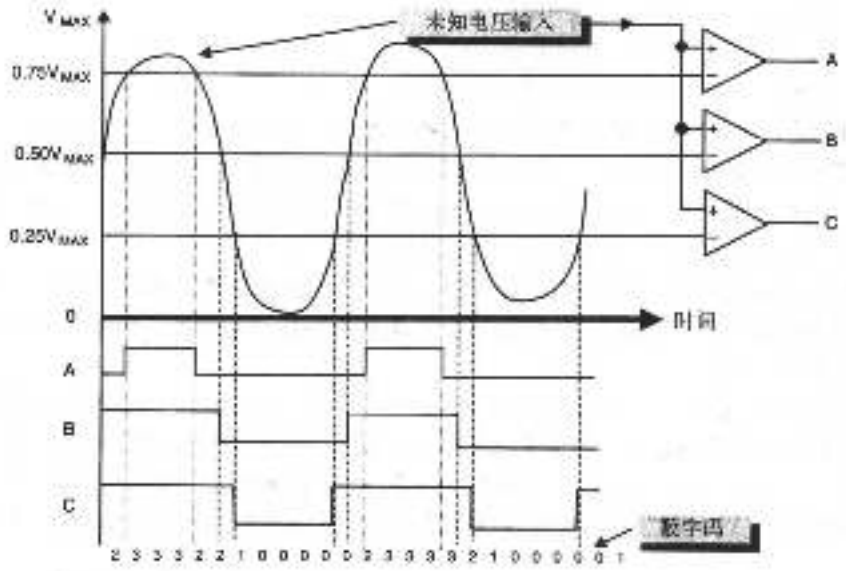


图12-9 2位模拟到数字转换器。模拟输入电压是通过具有输出1的最正（most positive）的比较器输出的，而不是提供二进制码

在图12-9的底部，我们可看到表示模拟电压瞬时值的数字码。你会看到，随着我们加入越来越多的比较器，将会发生什么情况。事实上，这正是我们将要做的。用3个比较器，我们能通过将其划分为4部分从而"数字化"未知的电压：

- 小于1V
- 在1V和2V之间
- 在2V和3V之间
- 大于3V

如果有更多的比较器，我们就能分成更多的部分，有更好的精确度。无论如何，你开始明白这个问题了，这是一个二进制方面的进展。用15个比较器，我们就能得到16个范围，但这只是4位的分辨率。为了得到8位的分辨率，我们将需要255个比较器和附加电路来将这一系列的比较器阶梯转换成数字码。但等一下，还有更多要考虑的，我们还必须以某种方式为每个比较器提供适当的阈值电压值。这意味着我们需要能产生255个不同的模拟阈值电压，每个比较器一个。由于这部分问题的解决方法非常有趣，所以我们需要暂时岔开思路再来处理它。

冒着招致电气工程师联盟愤怒的危险，我们将学习电子工程的基本方程。严格地说，这是额外的学习材料，我们可按需要学习关于计算机体系结构的知识而不用学习欧姆定律（Ohm's Law），但这个定律是应该学习的一个很好的方程，即使你永远不会从职业角度来使用它。而且，它还有助于我们理解将要遇到的一些电路，所以，凭此理由就应该学习这个定律。欧姆定律通过一个简单的方程将电流、电压及电阻联系在一起。

欧姆定律是说，穿过一个电路元件的电压V等于通过该电路元件的电流I乘以电路元件的电阻R。简单地说：

$$V = I \times R$$

用计量单位的话说，电压（用伏特度量）等于电流（用安培度量）乘以电阻（用欧姆度量）。这样，1安培电流流过1欧姆电阻（R=1Ω）将在该电阻上产生1V的电压。另一个常见的例子是：你房间中的灯泡连接到120V交流线。如果有1安培电流流过灯泡，其电阻就是120欧姆。

值的典型范围是：

- 电压：1mV到1kV，即10^{-3}V到10^{3}V
- 电流：1μA到10A，即10^{-6}安培到10安培
- 电阻：1Ω到1MΩ，即1欧姆到10^{6}欧姆

338 在我们继续学习并使用欧姆定律之前，让我们考虑一下其含意。一个恰当的类比是：一个公园的软管连接到野外的龙头上。假设我们处理的是水压而不是电压，流经软管的是水量而不是电流。此外，假设我们有一捆不同直径和长度的软管备用，从内径1/8"的微小软管到有4"口径的软管（灭火水龙管），长度从几英尺到几百英尺。

假设我们在龙头处有很大的水压（高电压）。对于多数软管来说，水量（电流）都很充足，所以软管的阻力不是很大。然而，如果我们使用具有最小直径的最长的软管，则在软管末端流出的就仅仅是一个滴流。

现在，假设水压（低电压）低。即使用一个短粗的软管（低电阻），得到的水流也仍然很小。这就是实际在起作用的欧姆定律。现在考虑图12-10。情况1显示出单个电阻的欧姆定律，这是一个称为电阻器（resistor）的电路元件，电阻器可以是一个灯泡的灯丝、电路板上的一个电阻器件或者集成电路内部的一个电阻。请注意电流I是如何通过电阻元件R在电阻元件两侧产生电压V的。

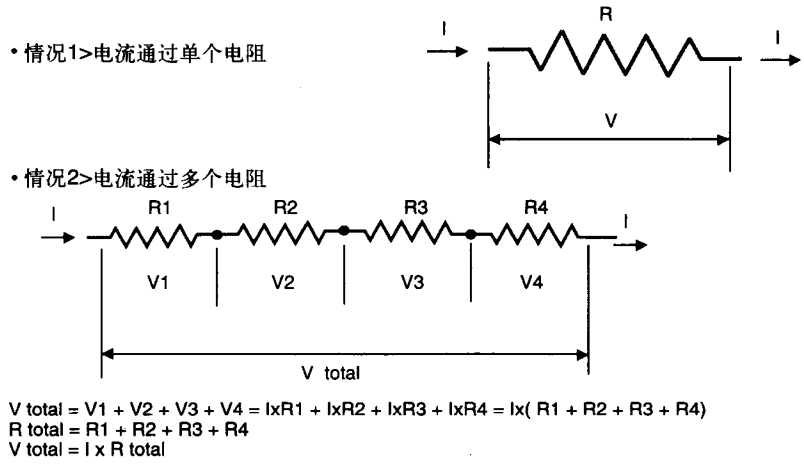


图12-10 针对单个电阻器和一系列电阻器的欧姆定律

情况2与情况1类似，不同之处是现在有了一系列的电阻器。电流从一个电阻器流向下一个电阻器。由于流经每个电阻器的电流都是相同的，我们就可以如图所示简化方程。如你所见，穿越整个电路的电压就是穿越每个单独电阻器的电压之和。然而，对于我们的A/D讨论来说重要的一点是，如果电阻值都相同，则穿越每个电阻器的电压就相同。

假设在图12-10中我们有10个电阻器，每个电阻器恰好都是1000欧姆。现在，我们将这一连串电阻器的一端连接到10V，另一端连接到0V。这10个串联的电阻器加起来共10 000欧姆，而穿越所有10个电阻器的总电压是10V。根据欧姆定律，流经电路的电流I=10V/10 000Ω，即1毫安（1mA）。1mA流经每个电阻器产生恰好1V电压，所以，每个电阻器的连接点处的电压

恰好比前一个高1V，这种类型的电路称为分压器（voltage divider），因为它将总电压分成了比较小的部分。

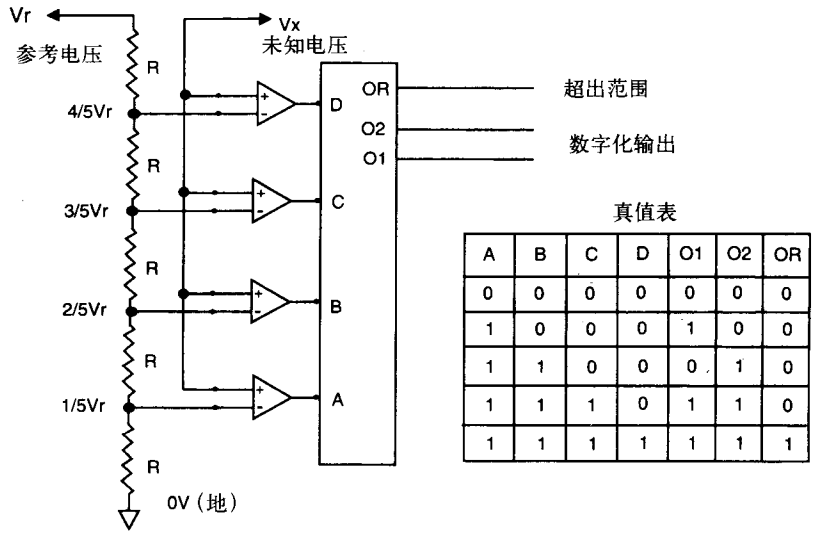
现在，我们有了足够的信息来理解如何真正地构建A/D转换器（但不要告诉国际电子电路设计者同业会）。图12-11是一个快闪A/D转换器（flash A/D converter）的简化图。它获得这样一个名称的原因是其极为快速。快闪转换器包含一堆比较器。 339

商业上可得到的电路有的多达4 096个比较器。每个比较器的负端连接到分压器的一个连接点（分接头）。分压器中的电阻器数量比电路中的比较器的数量多一个。每个电阻器具有相同电阻值，所以每个分接头处的电压都以相同的幅度递增。该电阻串的末端连接到一个稳定的参考电压，比如10.000V。

关键的一点是这一串电阻器中的每一个都有相同的值，这使得我们能将参考电压划分成任意多的参考点，这些参考点是为产生一个具有足够精确度的快闪A/D转换器所需要的。然而，我们不能只是盲目地进行。每次我们要求多一位的分辨率，我们所需要的比较器和电阻器的数量就会成倍地增长。

现在你可能在想，既然能建造具有5千万晶体管的集成电路，为什么不能建造一个只具有微不足道的8千个电阻器和比较器的A/D转换器呢？问题就在于模拟电路不能像数字电路那样严格地进行压缩，而电阻器是非常大的电路元件。而且，数字电路能够吸引我们的一个特色就是其对电平在一个相当宽范围内的变化不敏感，这恰与我们在使用模拟电子器件时相反。这样，制造8 192个相同的电阻器和8 192个具有完全相同开关特性的比较器就是电路设计的一个巨大（昂贵）的壮举。

图12-11的电路与我们在图12-9中看到的简单模型进行的工作是完全一样的，只有一点不同，就是我们加入了一个逻辑电路，它将来自这一堆比较器的数字码转换成了真正的二进制码。这样，如果我们有一个含255个比较器的快闪转换器，而且未知的电压恰好是参考电压的1/2，那么我们就能预料到最下面的128个比较器将全会输出1，而其他127个比较器将会输出0。数字电路接着就会将输入码转换成真正的二进制输出代码。算法会是什么样的呢？虽然看起来有点冗长乏味，但凭你现在的知识来做这个设计已经足够用了。你将会有255个输入和8个输出。仅需要多加一些逻辑门就能实现这个电路。



真值表

A	B	C	D	O1	O2	OR
0	0	0	0	0	0	0
1	0	0	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	1	1	0
1	1	1	1	1	1	1

图12-11 快闪A/D转换器

为了将具有逻辑“1”输出的最高比较器的编号转换成二进制码，我们需要一个称为优先级编码器（priority encoder）的逻辑电路。这样，255个比较器就会产生一个真正的8位二进制码作为输出。

快闪A/D转换器是快速的，因为其用于信号数字化所需时间仅仅是信号在比较器和优先级
340 编码器中的传播速度的简单函数。通常快闪A/D转换器都能在每10ns数字化一个信号，在一些极端情况下，需要的时间更少。快闪转换器最为从事冲击波研究的人所喜爱，因为在冲击波毁坏所有传感器前，你可以获得很多炸弹冲击波数据。

让我们总结一下快闪转换器的知识：

- 可允许的输入电压范围是0V到4/5Vr（参考电压）。
- A/D转换器的分辨率是2位。
- 假设参考电压V_{ref} = 5V，我们只能将未知电压V_x数字化成1V的精确度。
- 为了达到8位精确度，我们需要256个相配的电阻器和255个相配的比较器。

从概念上看，快闪A/D转换器是最容易理解的，但它却不是最常用的A/D转换器类型，主要是因为当我们试图得到10位以上的精确度时，就会导致指数复杂度。记住，每增加额外一位的分辨率，就需要将相配的比较器和电阻器数量加倍。

为了理解通常的A/D转换器如何工作，我们需要理解D/A转换器的电路。数字模拟转换器与A/D转换器有相反的功能，因为它是基于数字输入码来产生模拟输出电压的。下面图12-12所示的就是一个简单的4位D/A转换器。

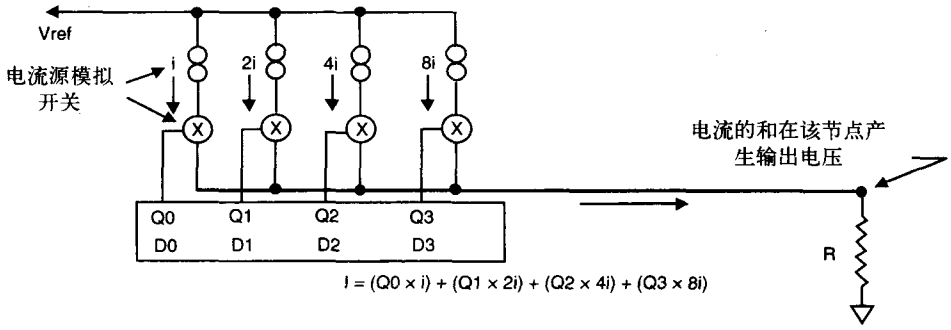


图12-12 一个4位的数字到模拟转换器

看起来像数字8的符号就是称为电流源（current source）的电路元件。只要有完备的电路，每个电路元件都能产生持续稳定的电流。来自每个电流源的电流幅度以2为级数增长，即1、2、4、8、16，等等。假设图12-12中“i”的值是0.1mA，则第一个电流源输出到地的电流就是0.1mA，第二个电流源输出0.2mA，其他电流源的输出依此类推。每个电流源还有一个开关（开关由中心画“x”的圆圈符号表示），该开关由表示成4位输出端口的数字输入信号控制。如果该数字信号是0，就没有电流流过电流源。如何该数字信号是1，电流就通过公用导线和电阻器流到地。

341 来自每个电流源的电流汇聚在一起，流过一个电阻器R。在这个电路中，如果R＝1000欧姆，并且所有电流源都打开，则我们就有15 × 0.1mA，即1.5mA流经一个1000欧姆的电阻器，能得到1.5V的电压。这样，从0到$F的数字码将以0.1V为台阶给出从0V到1.5V的模拟电压。现在我们就准备好来了解A/D转换器是如何实际工作的。

图12-13是一个16位模拟到数字转换器的简图。在它的中心是一个16位D/A转换器和一个比较器。电路的操作非常简单。我们首先将数字码$0000施加于该D/A转换器，D/A转换器的

输出是0V。该输出被施加在了比较器的负输入，而我们想要数字化的电压施加于比较器的正输入。然后，我们将数字码加1并将该16位码施加于D/A转换器。因为可有65 534个码，所以输出电压只略微增加。然而，我们也检查比较器的输出，看看其是否从1变为0。当比较器的输出改变状态时，我们就知道D/A转换器的输出电压只是略微大于该未知电压。

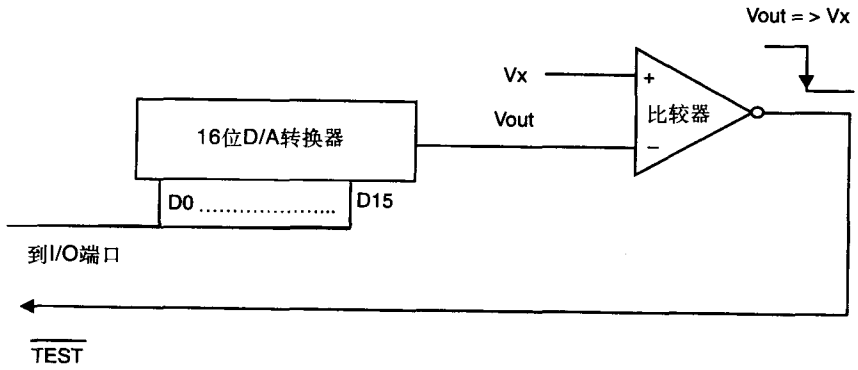


图12-13 一个16位单坡模拟到数字转换器

当其改变状态时，我们停止加数，比较器改变状态时的数字码就是未知模拟电压的数字值，我们称此为单坡（single-ramp）A/D转换器，因为我们是按一个线性斜坡增加测试电压直至测试电压和未知电压相等的。

想像你正在建立一个单坡A/D转换器，作为基于计算机的数据记录系统的一部分。你应该有一个16位的I/O端口作为数据输出端口，还有一个单个位的（$\overline{TEST}$）输入来对比较器的输出状态进行采样。你从一个初始状态开始，持续地对数字码加1，并对 $\overline{TEST}$ 输入进行采样，直至看到 $\overline{TEST}$ 输入变低。单坡A/D转换算法的流程图如图12-14所示。

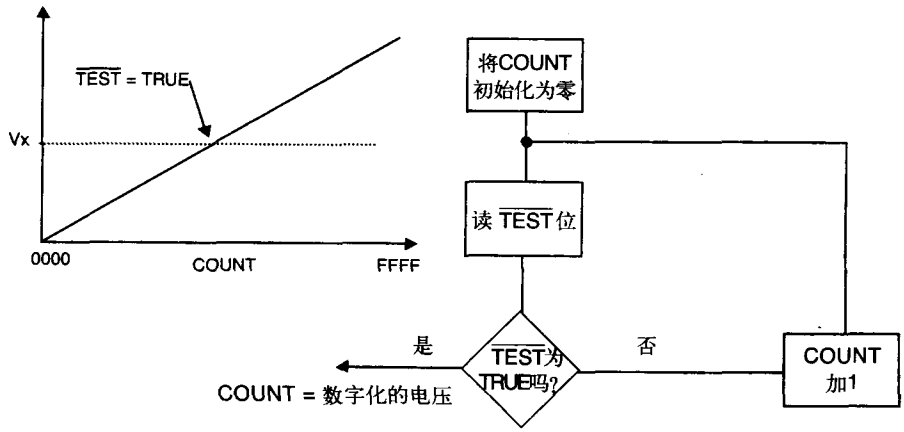


图12-14 单坡A/D转换器的算法

单坡方法有一个问题，就是数字化时间是变化的。低电压数字化得很快，高电压就需要较长的时间。而且，单坡算法类似于线性搜索算法。我们已经知道，二叉搜索的效率比线性搜索高，所以，你可能也想像得到，我们也可以用这种类型的电路将二叉搜索实现在硬件中，并在未知电压处比较器得到连续零的收入。这就是称为逐步逼近（successive approximation）的A/D转换器，它是目前最常使用的设计方式。 342

逐步逼近A/D转换器的算法就如同二叉搜索。我们是从0x8000，而不是从0x0000开始。

我们检查比较器的输出是1还是0，并依此将下一个最高有效位设置为1或0。这样，通过16次测试（而不是多达65 535次），16位A/D转换器就能确定未知电压。

最后一种类型的A/D转换器就是电压到频率转换器，即V/F转换器。这个转换器将输入电压转换成数字脉冲流，这个脉冲流的频率与模拟电压成正比。例如，一个V/F可有每伏特10kHz的转换函数。这样，输入1V，就有10 000Hz的频率输出。对于10V输入，就有100 000Hz的输出，等等。由于我们知道如何精确地测量与时间相关的量，所以就有可能非常精确地测量频率和计算脉冲，我们就能有效地进行电压到时间的转换。

V/F转换器有一个非常有吸引力的特性，就是在过滤掉输入信号中的噪声方面极其有效。假设V/F转换器的输出大约是50 000Hz。每一秒V/F发出大约50 000个脉冲。如果持续计数并累加，那么在10秒内将计数到500 000个脉冲，在100秒内将计数到5 000 000个脉冲，等等。

在更精确的标度上，也许每秒的计数有时比50 000还要稍大一些，有时要稍小一些。我们持续计数的时间越长，我们就越能将未知电压中的噪声平均掉。这样，如果我们原意等待足够长的时间，而且在这段时间输入电压是稳定的，我们就能将其平均到一个很高的精确度。

既然我们明白了模拟到数字转换器如何实际地工作，就让我们看一个可用于测量若干个模拟输入的完整的数据记录系统。

图12-15就是这种数据记录系统的简化示意图，图中有几个电路元件我们以前没有讨论过。在这个例子中没有必要详细分析其如何工作，我们只考察它们的总体操作，以理解数据记录过程如何进行。

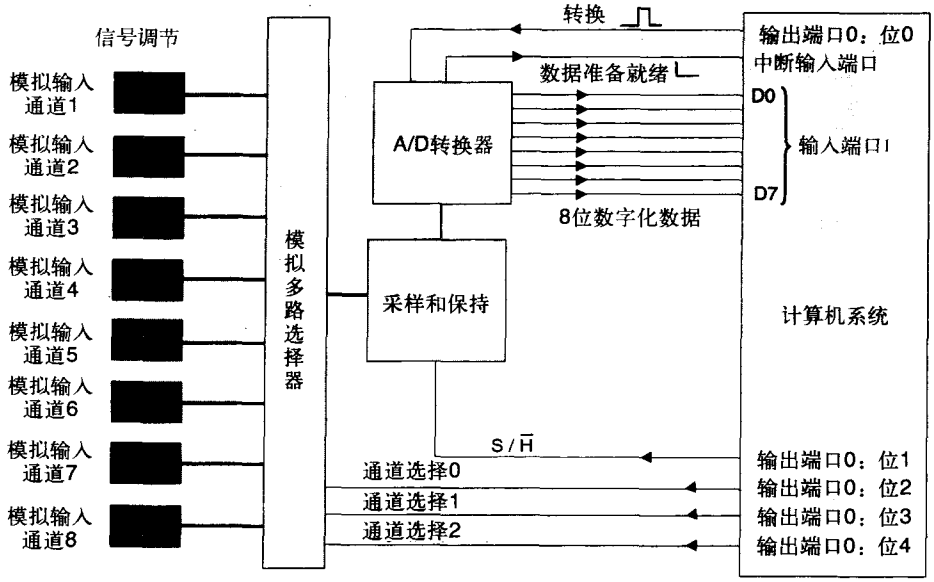


图12-15 基于计算机的数据记录系统的简化示意图

标记为“信号调整”的功能块通常是一组放大器或其他形式的信号转换器，其目的是将来自传感器的模拟信号转换为A/D转换器范围内的某个电压。例如，假设我们正在试图测量来自传感器的输出信号，该输出电压的范围是0到1mV。如果我们将该信号直接馈入一个具有0－10V输入范围的A/D转换器，则我们将永远也看不到传感器的输出。这样，我们就要用一个模拟放大器将传感器的0到0.001V范围的信号放大到0到10V范围。

假设每个模拟通道有不同的放大要求，这样每个通道就用自己的放大器或其他类型的信

号调节器独自处理。其要点就是我们想让每个通道的传感器范围与A/D转换器的输入范围达到最优匹配。请注意这个数据记录系统被设计成用来监视8个输入通道的。我们可将A/D转换器连接到每个通道，但通常这不是最经济的解决方案。另一个模拟电路元件称为*模拟多路选择器*（analog multiplexer），用于顺序地将每个模拟通道连接到A/D转换器。从真实的意义上看，模拟多路选择器就像一组三态输出器件连到公用总线，一次只允许一个输出连接到总线。这里的不同之处是模拟多路选择器能够保持其输入信号的模拟电压。

下一个器件称为*采样和保持模块*（sample and hold module, S/H）。要弄清楚其功能需要稍加解释。S/H模块使我们能对随时间变化的模拟信号进行数字化。从前面我们知道对模拟电压进行数字化可能需要相当多的时间。单坡A/D可能要计数到几千次才能与未知电压匹配。在以上这些例子中我们总是假设未知模拟电压是精密且恒定的。假设我们要精确数字化的是小提琴的声音，在某时刻我们想知道小提琴波形上的一个电压点，怎么办呢？如果在A/D转换器进行数字化期间小提琴的未知电压变化很大，就会产生很大误差。S/H就能解决这个问题。

S/H模块就像一个视频冻结帧。当数字输入处于采样位置（S/$\overline{H}$=1）时，模拟输出就追随数字输入。当S/$\overline{H}$变低时，模拟电压就及时冻结，A/D转换器就有精确对其进行数字化的机会。为弄明白为什么是这样，让我们看一个简单的例子。设想我们要试图对一个以10kHz频 344
率震荡的正弦波进行数字化，假设该正弦波的振幅是±5。这样，

$$V(t) = 5\sin(\omega t)$$

其中ω是该正弦波的角频率，以每秒的弧度来度量。如果这对你比较陌生，那么只要相信我继续下去就行了。角频率就是$2\pi f$，其中f是正弦波的实际频率，用赫兹（每秒周期数）度量。电压变化的速率就是V(t)的一阶导数：

$$dV/dt = -5\omega\cos(\omega t) = -10\pi f\cos(\omega t)$$

电压随时间变化的最大速率发生在当$\cos(\omega t) = 1$时，故

$$dV/dt\ (\text{最大值}) = -10\pi f,\ \text{即}-31.4 \times 10 \times 10^3$$

这样，电压随时间变化的最大速率是每μs0.314V。现在，如果我们的A/D转换器做一次转换需要5μs，则在转换进行期间未知电压可能改变约1.5V。由于这通常是一个不可接受的大的错误源，我们就需要S/H模块在转换进行期间向A/D转换器提供一个稳定的信号。

我们现在已知道了足够的关于系统如何发挥作用的知识，让我们做一个逐步的分析：

1. 输出端口0的位2：4用于选择期望的模拟通道连接到S/H模块。

2. 调节后的模拟电压出现在S/H模块的输入端。

3. 输出端口0的位1变低，将S/H模块置于保持模式。S/H变低的时刻，要数字化的模拟输入电压就锁定其值。

4. 输出端口0的位0向A/D转换器发出一个正脉冲，引发一个转换周期。

5. 在需要的转换间隔后，*转换结束*信号（$\overline{EOC}$）变低，引起一个计算机的中断。

6. 计算机进入其A/D转换器的ISR并读进数字数据。

7. 依靠这个算法，它还可以选择另一个通道并读另一个输入值，或者继续如前面一样继续对这个通道进行数字化。

图12-16总结了建立任意速度和精确性的A/D转换器的困难程度。标记为“SK”的区域虽然在理论上做起来是相当简单的，但常常需要应用领域的专门知识。例如，一个心脏监视器可能相对比较慢，并且具有中等精确度，但由于需要保护患者免除任何休克危险，这就对设 345
计者提出了额外的要求。

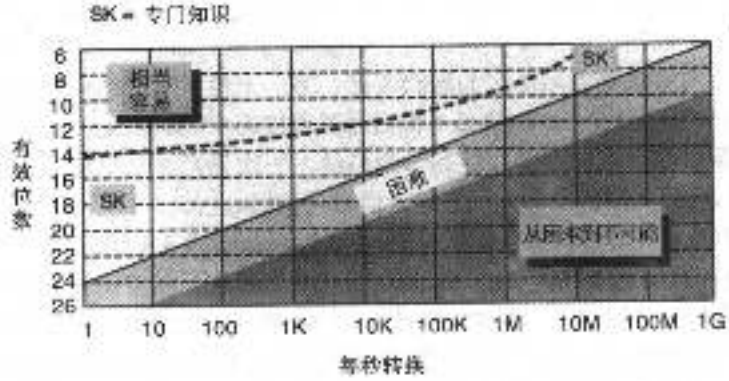


图12-16 该图总结了在给出精确度和转换速率情况下制造A/D转换器的困难程度。来自Horn[8]

12.6 A/D和D/A转换器的分辨率

在离开模拟数字转换和数字模拟转换这个话题之前，我们应该对转换器分辨率的意思作一讨论总结。这个讨论也同样适用于D/A转换器，但从A/D转换器的角度来看更容易理解一些，所以我们就这样做了。当我们试图将模拟电压、电流或电阻（记得欧姆定律吗？）转换成相应的数字值时，我们就面临一个基本的问题。模拟电压是连续变化的量，而数字只能表示成离散的值。

从C++编程类中你已经熟悉了这个问题。你知道（或者应该知道）某些操作是有潜在危险的，因为它们会导致错误的结果。在编程中，我们称此为"舍入错误"。考虑下面的例子：

```
float A=3.1415906732670;
float B=3.1415906732566;
if (A==B)
      { 做某些事}
else
      { 做另外一些事}
```

该程序会做什么呢？除非你知道在计算机上用一个浮点数能表示多少位的精度，否则你不一定能得到你期望的结果。对于A/D转换器我们有同样的问题。假设我有一个精密的电压源，这是一个能在长时间提供非常稳定电压的电子设备。典型情况下，可采用称为标准电池（standard cell）的特殊电池。假设我们刚花了500美元，并将我们的标准电池送回了位于马里兰州Gaithersburg的国家标准测试研究所（NIST）。

过了几周，我们从NIST取回标准电池和校准证，校准证说明标准电池上的电压在23摄氏度时是+1.542324567V（相对于温度变化有一个轻微的电压变化，但我们能计算出来）。现在，我们将这个电池装配到我们的A/D转换器上并读数。我们将会度量到什么值呢？

现在你还没有足够的信息来回答，所以让我们更具体一点：

A/D范围：0V~+2.00V

A/D分辨率：10位

A/D精确度：±1/2最低有效位（LSB）

这意味着对从0.00V到+2.00V的模拟输入范围，我们有1024个数字码可用来表示模拟电压。我们知道0.00V应该给出数字值00 0000 0000，+2.00V应该给出数字值11 1111 1111，但在这两个值之间是什么情况呢？在哪一点数字码从0x000变化到0x001？换句话说，我们的A/D转换器对模拟输入电压的变化或波动有多么敏感？

让我们尝试计算出来。由于在0x000和0x3FF之间有1023个区间间隔，所以我们能计算出模拟电压的什么区间间隔对应于数字化值的1位变化。

因此，2.00/1023=1.9550 × 10^{-3}V。这样，每次模拟电压变化2mV（毫伏）左右时，我们就应该能看到数字码也变化了一个单位。这个2mV的值也就是我们所称的最低有效位，因为这个电压量的变化将导致LSB变化1个单位。 346

参见图12-17。阶梯形曲线代表A/D转换器的转换函数，它表明了数字码作为模拟输入电压的函数是如何变化的。注意数字码0x000是如何在模拟电压上升到差不多1mV时增加的。由于精确度是LSB的1/2，我们就有一个围绕模拟区间间隔中心（垂直虚线）的模拟电压范围。这是一个由线的水平部分定义的区域。例如，对于一个刚好处于1mV以下、3mV以上这个范围内的模拟电压，数字码将是$001。

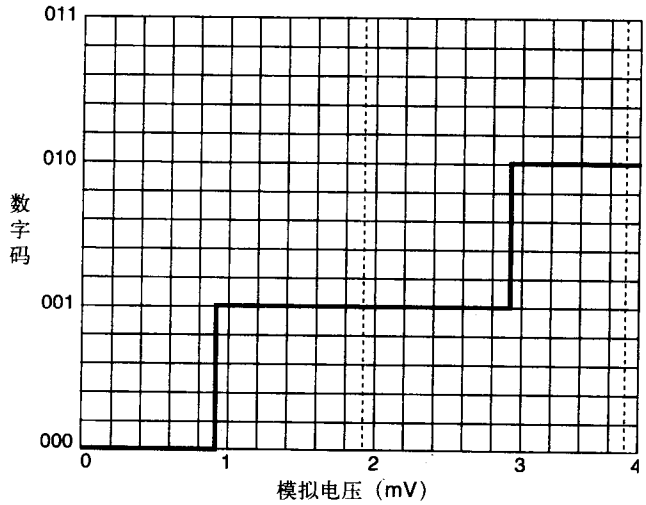


图12-17 范围为0到2.00V的10位A/D转换器的转换函数。精确度是1/2 LSB

如果模拟电压刚好处于转换点会发生什么情况呢？假设其刚好是0.9775mV会怎么样？数字码会是$000或$001吗？答案是："谁知道呢？"有时它可能会被数字化成$000，而另外一些时候它会被数字化成$001。

现在回到我们的标准电池。回忆可知，标准电池上的电压是+1.542324567V。数字码应该是多少呢？+1.542324567/(1.9550 × 10^{-3}) =788.913，近似等于789。用十六进制，789_{10}等于0x315，所以这就是我们可能会看到的数字码。

这个分辨率足够好了吗？这是一个难以回答的问题，除非我们知道问题的背景。假设我们得到了一个任务，要写一个软件包，用于控制制造车间中的熔炉。熔炉中发生的过程对温度的波动相当敏感，所以我们必须进行严密的控制。具体地说，就是温度必须保持在恰好400摄氏度±0.1摄氏度。现在，熔炉中的温度由热电偶监视，其电压输出度量如下：

电压输出@400摄氏度 = 85.000mV

转换函数 = 0.02mV/摄氏度

至此，看起来不是很有希望。但是，我们能做一些事情来改善这种状况。我们能做的第一件事情就是用热电偶对很低的电压输出进行放大，将其提高到更易处理的水平。如果我们使用一个能将输入电压放大20倍（增益=20）的放大器，则模拟信号就会变为：

电压输出@400摄氏度（×20）= 1.7000V 347

转换函数（×20）= 0.4mV/摄氏度

现在模拟电压的范围就可以了。我们的信号在400摄氏度时是1.7V，这低于A/D转换器的最大电压2.00V，所以我们就没有任何超出度量范围的危险。分辨率怎么样呢？我们知道，在A/D转换器检

测到一个变化之前，模拟信号可在差不多2mV的范围变化。参考一下我们对放大热电偶的指定，这就意味着在A/D转换器检测到一个变化之前，温度可变化大约5摄氏度。由于我们需要将系统控制到好于0.1度，所以我们需要使用具有更好分辨率的A/D转换器。要多好呢？我们预知0.1摄氏度的温度变化将会引起0.04mV的电压变化，因此，我们必须将分辨率改进2mV/0.04mV，即50倍！

这可能吗？假如我们决定在eBay上卖掉10位A/D转换器并用所得收益买一个新的A/D转换器。12位转换器怎么样？这会给我们4096个数字码。从1024个数字码到4096个数字码在分辨率上只有4倍的改进，而我们需要50倍的改进。一个16位的A/D转换器能给我们65 536个数字码，这是64倍的改进。这刚好能行！现在，我们就有：

A/D范围：0V ~ +2.00V

A/D分辨率：16位

A/D精确度：±1/2最低有效位（LSB）

我们的模拟分辨率现在就是每数字码2.00V/65 536，即0.03mA。由于我们需要能检测0.04mV的变化，所以这个新的转换器能为我们工作。

总结

第12章涵盖了以下内容：

- 作为一种处理异步事件方法的中断概念。
- 计算机系统如何通过I/O端口与外部世界打交道。
- 通过模拟到数字转换和数字到模拟转换，如何将外部事件中的物理量转换成与计算机兼容的格式或反之。
- 对被称为比较器的模拟到数字接口器件的需要。
- 如何用欧姆定律为A/D转换和D/A转换确立固定电压点。
- 不同类型的A/D转换器及其优缺点。
- 精确度和分辨率是如何影响A/D转换过程的。

参考文献

[1] Glenn E. Reeves, "Priority Inversion: How We Found It, How We Fixed It," Dr. Dobb's Journal, November, 1999, p. 21.

[2] Arnold S. Berger, *A Brief Introduction to Embedded Systems with a Focus on Y2K Issues*, Presented at the Electric Power Research Institute Workshop on the Year 2000 Problem in Embedded Systems, August 24–27, 1998, San Diego, CA.

348

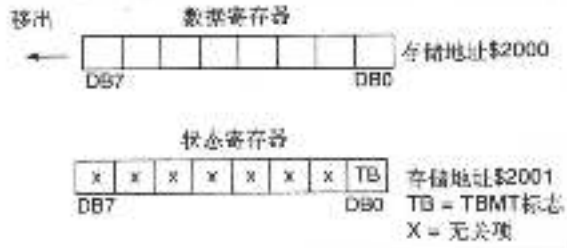
[3] Jerry Horn, *High-Performance Mixed-Signal Design,* http://www.chipcenter.com/eexpert/jhorn/jhorn015.html.

习题

1. 用Motorola 68000汇编语言写一个子程序，使得一个串行UART器件能根据下述定义传送一串ASCII字符：
   a. UART是映射在存储器字节地址位置$2000到$2001的。
   b. 向地址$2000写一个字节数据将会自动启动一个数据传送过程，该过程会将状态寄存器的传送缓冲器空标志（TBMT）置为0。
   c. 当数据字节被发送后，TBMT标志将自动返回到1，指明TBMT为真。
   d. 状态寄存器是映射在存储器字节位置$2001的。这是一个只读寄存器，有用的位只有DB0，就是TBMT标志。
   e. 要传送的字符串的存储地址通过寄存器A6传送给子程序。
   f. 子程序不返回任何值。
   g. 子程序中使用的所有寄存器必须在进入子程序时被存储起来，并在返回时恢复。
   h. 所有的字符串都由可打印的ASCII字符集中的00到$7F的字符组成，这些字符具有连续的

存储位置，字符串以$FF结束。

UART的示意图如下所示：



注释：

- 记住，你只是在写一个子程序，没有必要加入一个程序所需要的伪操作代码。
- 你可以假设已经定义了堆栈。
- 你可以在程序源码中使用EQU（等于）伪指令，以便于利用符号名。

2. 考察如下所示的68K汇编语言代码块。代码中有一个严重的错误。下面还给出了存储器前32字节的内容。 349

a. 代码中的错误是什么？

b. 当错误发生时处理器将会做什么？根据你已有的信息，尽量给出完整的解释。

```
                org     $400
start:          lea     $2000,A0
                move.l  #$00001300,D0
                move.l  #$00000010,D1
loop            divu    D1,D0
                move.l  D0,(A0)+
                subq.b  #08,D1
                bpl     loop
                end     $400
```

存储器内容（部分）

```
00000000    00 00 A0 00 00 00 04 00 00 AA 00 00 00 AA 00 00
00000010  00 AA 00 00 00 00 CC AA 00 00 AA 00 00 00 AA 00 00
```

注释：异常向量表中的前几个向量列表如下：

向量 #	存储器地址	描　述
0	$00000000	复位：管理堆栈指针
1	$00000004	复位：程序计数器
2	$00000008	总线错误
3	$0000000C	地址错误
4	$00000010	非法指令
5	$00000014	用零除
6	$00000018	CHK指令
7	$0000001C	TRAPV指令
8	$00000020	违反特权

3. 假设你有两个模拟数字转换器，如下表所示：

转换器类型	分辨率（位）	时钟速率（MHz）	范围（伏特）
单坡	16	1.00	0到+6.5535
逐步逼近	16	1.00	0到+6.5535

每种类型的转换器要数字化+1.5001V的模拟电压需要多长时间（μs）？

4. 假设你为下列每个处理器中断事件赋予了从0（最低）到7（最高，NMI）的优先级。对于下列每个事件，赋予其优先级，并简要描述你赋予这种优先级的原因。
   a. 键盘敲击输入。
   b. 即将到来的电源失效。
   c. 监视定时器。
   d. MODEM有要读的数据。
   e. A/D转换器有新的数据。
   f. 10ms的实时时钟滴答。
350 g. 鼠标点击。
   h. 机器人的手已经接触到固体表面。
   i. 存储器奇偶错误。
5. 假设你有一个11位的A/D转换器，能对从−10.24V到+10.23V范围的模拟电压进行数字化。A/D转换器的输出格式是正数的2补码还是负数的2补码取决于模拟输入信号的正负。
   a. 模拟输入电压的变化值最小是多少才能保证该变化被检测到（即引起数字输出值变化）？
   b. 表示模拟电压−5.11V的二进制数字是什么？
   c. 假设A/D转换器被连接到一个具有16位宽数据总线的微处理器，则对于+8.96V的模拟电压，该十六进制数字是什么？提示：没有必要将11位的数字调整为16位。
   d. 假设A/D转换器是一个逐步逼近型的A/D转换器，则在最终对模拟电压进行数字化之前需要多少个样本？
   e. 假设A/D转换器被1MHz时钟信号所控制，采样发生于每个时钟的上升沿，则数字化一个模拟电压需要多长时间？
6. 假设你是一家医学电子公司的首席软件设计师。你的新项目就是为一种便携式心脏监视器设计一些关键算法。为了检验你的算法，你将用一些基本的硬件做一个简单的实验。监视器将采用一个10位模拟数字转换器（A/D），输入范围是0V到10V。0V的输入电压会导致0000000000的二进制输出，而10V的输入电压会导致1111111111的二进制输出。每200μs就对模拟信号进行一次数字化。你要获得一些数据，下列显示的就是数字化后的数据值（用十六进制显示）。

   2C8, 33B, 398, 3DA, 3FC, 3FB, 3D7, 393, 334, 2BF, 23E, 1B8, 137, 0C4, 067, 025, 003, 004, 028, 06C, 0CB, 140, 1C1, 247

   一旦你收集到这些数据，你就想将数据写到带状记录仪并显示，使得医生能读到。带状记录仪的输入范围是−2V到+2V。幸运的是，你的硬件工程师已经设计了一个10位数字到模拟（D/A）电路，使得0000000000的二进制输入值产生−2V的模拟输出，而1111111111则产生+2V的输出。你写一个将数字化数据发送到记录仪的简单算法，使得你能了解是否一切工作正常。
   a. 通过将上述数据描绘在图纸上来表明记录仪会输出什么。
   b. 波形有周期吗？如果有，该波形的周期和频率是多少？
7. 假设你有一个14位的逐步逼近式的A/D转换器，转换时间是25μs。
   a. 假设你想在未知波形的每个周期内最少收集4个样本，则你能度量的交流波形的最大频率是多少？
351 b. 假设转换器能转换从−5V到+5V的输入电压，则转换器能度量的最小电压变化是多少？
   c. 假设你想将这个A/D转换器与一个特定的采样保持电路（S/H）一起使用，而该S/H具有每

ms1V的下降速率。那么，这个特定的S/H电路与A/D转换器相容吗？如果不是，为什么？

8. 将各种应用与最适合它的A/D转换器相配。转换器列举如下：

A. 28位逐步逼近A/D转换器，每秒2个样本。

B. 12位逐步逼近A/D转换器，20μs的转换时间。

C. 0~10kHz的电压到频率转换器，0.005%的精确度。

D. 8位快闪转换器，20ns转换时间。

a. 在军事研究实验室中进行炮弹冲击波测量。_____

b. 用于天气遥测的通用数据记录仪。_____

c. 7数字的实验室用电压表。_____

d. 铸造厂的熔钢温度控制器。_____

9. 下面是一列C语言函数原型。将它们以正确的次序排列，使你的嵌入式处理器接口到一个8通道12位A/D转换系统。

a. boolean Wait(int) /* 真＝做完，int定义了超时前等待的毫秒数 */

b. int GetData(void) /* 返回数字化的数据值 */

c. int ConfidenceCheck(void) /* 执行对硬件的信任检测 */

d. void Digitize(void) /* 打开A/D转换器进行数字化 */

e. void SelectChannel(int) /* 选择模拟输入通道进行读 */

f. void InitializeHardware(void) /* 将硬件状态初始化到一个已知的状况 */

g. void SampleHold(boolean) /* 真＝采样，假＝保持 */

10. 假设你有16位D/A转换器，在设计上类似于图12-12所示。最低有效数据位D0的电流源产生0.1微安培的电流。要使得该D/A转换器的满刻度是10.00V，所需要的电阻值是多少？ 352

第13章　现代计算机体系结构简介

学习目标

- 描述CISC和RISC体系结构的基本特征；
- 解释在现代计算机中为什么要使用流水线；
- 解释流水线的优点及其产生的性能问题；
- 描述处理器如何在一个时钟周期里执行多条指令；
- 解释编译器为了提高系统性能而充分利用计算机体系结构的一些方法。

目前，微处理器的速度、功耗、功能和成本都跨越了很大的范围。你可以只花不到25美分就买一个4位的微控制器，也可以花超过1万美元买一个用于空间计算的专用处理器。现在正在使用的微处理器就有超过300种，我们如何区分如此种类繁多的处理器呢？而且，出于本书的目的，我们这里将不考虑大型计算机（IBM、VAX、Cray、Thinking Machines，等等），而是将我们的讨论局限在微处理器这个范畴。

一般来说，现今使用的微处理器系统结构主要有三种：CISC、RISC和DSP。稍后我们将讨论这些缩写的意思，现在我们先讨论一下如何区分这么多种器件，哪些因素能确定和区分出不同的类别。首先让我们试着来确定一下产生各种配置的途径：

1. 时钟速度：目前处理器的时钟速度可从几乎为0到几个GHz。对于现代的CMOS电路设计，一个器件所消耗的功率通常是与它的时钟频率成正比的。如果你希望让一个安装在鲸鱼脊背上的微处理器使用一个AAA电池维持运行两年，那么它的时钟频率就不能太快，比较好的做法是让它根本不运行，而仅仅是在需要它做一些有用事情的时候唤醒它，然后再让它继续休眠。

2. 总线宽度：我们还可以通过数据通路的宽度来区分处理器：4位、8位、16位、32位、64位、VLIW（超长指令字）。一般来说，如果总线的宽度扩大一倍，处理一个算法的速度会提高2到4倍。

3. 处理器的寻址空间也各有不同。一个简单的微控制器可能只有1KB的寻址空间，而
353 Pentium、SPARC、Athlon和Itanium系列的机器有几个GB的寻址能力。Freescale公司的PowerPC处理器有64位的存储器寻址能力。

4. 微控制器/微处理器/ASIC：这类设备能称为严格意义上的CPU吗，譬如Pentium或者Athlon？或者说是一个带外设的集成化CPU，譬如68360？又或者说是一个加密的Verilog或者VHDL代码库，譬如ARM7TDMI，最终将被用于定制的集成电路设计？

如你所知，我们还可以通过*指令集体系结构*（instruction set architecture, ISA）来区分处理器。从一个软件开发者的角度来看，这就是处理器的体系结构，ISA的不同就决定了特定处理器所适用的应用领域。在本书中我们已经学习了Motorola 68K 的ISA，以及Intel x86和ARM v4的ISA,但这些只是目前正在使用的众多不同的ISA的三种。其他的例子还有29K、PPC、SH、MIPS以及各种DSP的ISA。即使同一种ISA内也可以有超过100种不同的微处理器或集成器件。例如，Motorola公司的微处理器系列代号为680X0，其中X用不同的数字替代时就代表这个系列中一个型号。如果我们在一个型号为68000的处理器核上加上一些外设器件，它就变成6830X系列。其他的公司也有类似的分类策略。

现代的处理器在时钟速度上的跨度也很大，从不到1MHz到超过3GHz（3000MHz）。不久以前，CRAY超级计算机花费了超过一百万美元才使其时钟速度达到前所未有的1GHz。为了达到这个速度，CRAY的工程师们必须要构建额外的、带有液冷设备的电路板，并且还要通过变更线路的长度来控制在上面传输的信号时延。现在，我们大部分人桌上的个人电脑都已经达到了这一性能。事实上，我写这本书时用的PC就带有AMD Athlon 2.0GHz的CPU，这被认为是AMD公司生产的第三代产品。如果这本书出版得成功，我就可以用得到的版税收入来把PC升级到Athlon64了。

13.1 处理器体系结构，CISC、RISC及DSP

68K处理器及其指令集，8086处理器及其指令集，这两者都属于CISC，即复杂指令集计算机体系结构。CISC的特点是指令系统大、寻址模式多。你很可能已经见过许多汇编语言的指令，还有一些已有指令经过变化后的指令。而且，这些指令执行时需要的时钟周期数有很大的差异。如下表所示，执行一条MOVE指令时，由于执行方式的不同，需要的时钟周期数最少的仅为8，最多的达28。

指　令	时钟周期数	指令执行时间（μs）*
MOVE.B #$FF, $1000	28	1.75
MOVE.B D0, $1000	20	1.25
MOVE.B D0, (A0)	12	0.75
MOVE.B D0, (A0)+	8	0.50

* 假设时钟周期为16MHz。

指令执行的时间差异较大也是CISC体系结构的特征。CISC指令集可以非常紧凑，因为一条复杂指令能做多个操作。我们回忆一下**DBcc**，即先进行条件判断，然后递减并根据条件进 354
行转移，这是一个典型的CISC指令。CISC体系结构又称为冯·诺依曼体系结构（von Neumann architecture），这是为了纪念最先提出这一结构的约翰·冯·诺依曼。下面我们就来讨论一下冯·诺依曼体系结构一个方面的问题。

CISC处理器往往需要大量的电路，在集成电路的硅片上需要占用大块的面积，这使得公司在试图发展CISC技术时面临两个问题：较高的成本和较慢的时钟速度。导致较高成本的原因是集成电路的价格在很大程度上是由制造的成品率所决定的，制造成品率度量的是，从通过IC制造过程的每个硅圆片上可获得多少个好的芯片（成品率）。包含复杂电路的大芯片与小芯片相比有更低的成品率。而且，复杂芯片很难提高速度，因为在整个芯片区域上分配和同步时钟是一个有难度的工程任务。

一个具有冯·诺依曼体系结构的计算机只有一个存储空间，用于存放指令和数据，如图13-1所示。CISC计算机只有一组总线连接CPU和存储器，指令和数据必须共享同一条数据通路从存储器进入CPU，因此如果CPU正在写一个数据值到存储器，它就不能读取下一条要执行的指令，它必须要等到数据被写入存储器之后才能进行操作。这一问题被称为冯·诺依曼瓶颈（von Neumann bottleneck），因为它限制了处理器的运行速度。

哈佛大学的Howard Aiken发明了哈佛体系结构（他肯定是太谦虚了所以没有用他自己的名字来命名）。哈佛体系结构的特点是指令和数据分开存储，这样就能够独立地对指令和数据进行操作。哈佛体系结构和冯·诺依曼体系结构还有一个微妙的差别，后者允许程序自行修改代码，但前者不可以。因为在冯·诺依曼体系结构中指令代码与数据位于同一个存储空间

中，所以一条指令可以修改代码空间中其他部分所存放的指令。而在哈佛结构中，存取操作只能在数据存储器中进行，自行修改代码是非常困难的。

哈佛体系结构通常与精简指令集计算机（RISC）体系结构的思想是相关联的，但是你肯定可以用哈佛体系结构设计CISC计算机。实际上，目前的CISC体系结构具有分开的片上cache存储器分别用于存储指令和数据的情况是相当常见的。

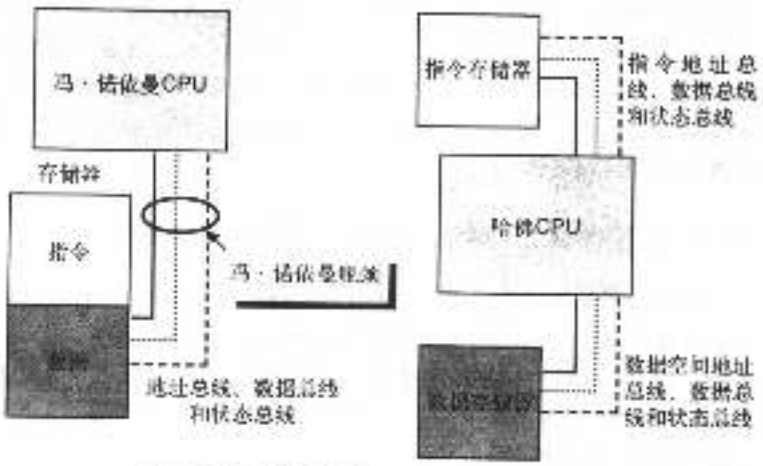


图13-1 冯·诺依曼（CISC）体系结构和哈佛（RISC）体系结构的存储器体系结构

哈佛体系结构在AMD（Advanced Micro Device）公司生产的Am29000 RISC微处理器上得到了商业化使用。虽然Am29K处理器在惠普公司的首批LaserJet系列打印机上得到了商业化的使用，但是设计者们很快就向AMD抱怨基于29K的设计成本太高，因为需要设计两个完全独立的存储空间。为此，AMD公司的后续处理器都采用了单存储空间来存储指令和数据，这样也就放弃了哈佛体系结构的优势。然而，就像我们很快就要看到的，哈佛体系结构在很多具有片上指令和数据cache的现代微处理器中还是生存下来了。今天，要实现ARM处理器既可以使用冯·诺依曼体系结构，也可以使用哈佛体系结构。

355

在20世纪80年代早期，很多研究者都在探究通过流水线技术来提高微处理器的性能，而不是把指令集做得越来越复杂[1, 2]。根据Resnick[3]的讲述，早在60年代末期，Thornton[4]在设计CDC 6600计算机时就已经研究了RISC体系结构的某些方面。那些从事RISC计算机研究的计算机科学家进行了早期的研究，这些研究涉及指令集中的那些部分是编译器设计者和高级语言实际要用到的。在一项研究中[5]研究者发现，在所有执行的指令中，10条指令占了总数的80%，而仅仅30条指令就占了总数的99%。这样，研究者所发现的就是，在绝大部分时间，只有少数的指令和寻址模式是真正要用到的。直到那时，ISA的广度和复杂性都是CPU设计者们所自豪的，会有“我的指令集比你的指令集大”等等之类的竞争。

Patterson和Ditzel在他们的论文引言中提到：

> 也许这种复杂性的增加对于提高新模型的成本效益有积极的作用，但本文认为，这一倾向并不总是划算的，实际上甚至很可能带来的害处比好处还要多一些。我们将给出RISC结构与CISC结构同样经济有效的例子。

为了寻求越来越复杂和优雅的指令和寻址模式，CPU设计者们设计了越来越复杂的CPU，这种复杂性反而遏制了CPU的发展。科学家们开始思考：“假设我们只留下最最必需的指令和寻址模式，而把其他的统统去掉。简化后不可避免地会使得程序的代码量增加，这样做是否值得呢？”

答案绝对是肯定的。如今RISC已经成为主流的体系结构，原因是RISC的好处远远超过了CISC，即使代码量可能要增加1.5到2倍，但速度的提高以及全流水线的设计远超过这些代价。现代的RISC处理器能够在一个时钟周期内运行多条指令，称为超标量体系结构（superscalar architecture）。当我们考察流水线时，我们就会了解这一奇迹是如何实现的。最初的RISC设计

往往采用哈佛结构，但随着cache的增大，设计者们也就只设置单个的外部存储空间了。

然而，事情并没有这么简单，一些现代RISC设计的ISA（如PowerPC）已经变得和CISC处 356
理器一样复杂了。而且，CISC体系结构和RISC体系结构的某些方面一直在相互融合，要在它们之间划一条明确的界限已不容易。例如，现代的Pentium和Athlon CPU所执行的ISA是从Intel经典的x86 CISC体系结构所演化而来的，但是，从它们内部所表现出来的却是RISC处理器的特性。而且，Intel和AMD引领着速度的提高，目前3GHz以上的处理器都是Athlon和Pentium的。

RISC和CISC都能完成工作，虽然RISC处理器非常快速和高效，但由于提供给编译器的指令较少，所以RISC处理器的执行代码趋向于更大。尽管这种差别在迅速减小，CISC计算机还是更多地倾向于被使用在控制应用中，如工业控制器、仪器控制器等等。

另一方面，RISC计算机则被广泛应用于数据处理的应用领域，一般主要做这样的事情：

数据输入 >>> 数据处理 >>> 数据输出

RISC处理器因其指令简化、高速，非常适合于注重数据移动的算法，比如用于远程通信和游戏的算法。

数字信号处理器（digital signal processor, DSP）是对数据进行数学数据处理的一类专用处理器。DSP进行数学运算，它不进行控制（CISC）或者数据处理（RISC）。传统上，DSP是典型的CISC处理器，它带有若干个增强部件来加速特殊数学运算的执行。这些部件都是硬件电路，譬如在前面讨论过的ARM乘法器模块中的桶型移位器和乘法/累加（MAC）指令（见图11-2）。回想一下内循环的过程：

- 取常量X和变量Y
- 两数相乘，并累加（SUM）结果
- 判断循环是否结束

DSP用一条指令就能实现CISC处理器需要8条或更多条指令才能完成的工作。回忆一下积分知识，我们知道求一个函数的积分等价于求它对应的曲线与X轴之间的面积。要求出这个面积，可以在曲线上每隔一小段向X轴做一个很小的矩形，最后把所有矩形的面积累加起来。这就是DSP的一个MAC指令。

求解积分在求解许多数学方程和实时数据转换中都是很重要的一个部分。一般DSP的主要工作是从A/D转换器接收数据流，对数据流进行操作，再将数据传给D/A转换器。图13-2显示了一个连续变化的信号通过A/D转换器后进入DSP，经DSP处理后又传给了D/A转换器。这个过程中DSP要对数据流进行实时处理，模拟信号被转换为数字信号，经过处理后又被转化为模拟信号。

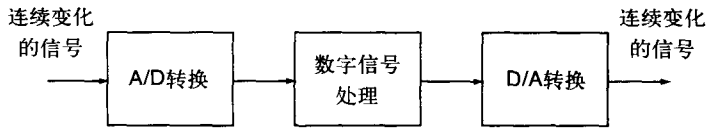


图13-2 DSP对连续数据的处理

DSP已有一些很好的应用，首当其冲的就是PC机上的调制解调器和声卡。在PC机出现之前，DSP属于专用设备，大都局限于军事或者CIA之类的应用。现在如果你参加一个电话会议，并在会议上使用话筒发过言，那你就一定已经使用了DSP进行处理。如果没有DSP，那将会有 357
令人烦恼的回音和反馈造成的刺耳声。DSP在这里进行了消除回音的操作，并在你说话的时候实时地消除反馈。DSP在我们日常生活中的最新应用是数码相机，这些设备里面有着很多复杂的DSP，能够处理一个3到8百万像素的图像，可以在短短的数秒钟内将原始图片转化为一个压缩的jpeg格式的图片。

13.2 流水线简介

在讨论流水线之前，我们先回顾一下CISC和RISC，它们与流水线技术有着重要的联系。当然，我们需要明确一下我们这里说的流水线是指什么，在某种意义上而言，流水线是计算机里一个必要的恶魔，Turley[8]提到：

> 处理器有许多事情要去做，但这些事情大都不会在一个时钟周期内完成。执行一个命令需要从存储器中取指令、译码、存取操作数和结果、进行位移、加法、乘法运算以及其他任何程序要求的操作。一种解决方案是降低CPU的时钟频率，直到它能够在一个时钟周期内能完成所有的工作，但是显然没有人会喜欢这种做法。流水线技术就是要在CPU必须完成的工作量和完成这些工作需要的时间这两者之间找到一个平衡。

让我们讨论得更加深入一些。回顾可知逻辑门（与、或、非）及由这些逻辑门构成的更大的电路模块都属于电子电路，一个信号从某个电路的输入传输到输出需要花费一定的时间。信号经过的门越多，传输的延迟也就越大。

如图13-3，这是一个有着8个输入、3个输出的复杂功能模块。我们可以假设在模块里进行了某种类型的字节处理。我们假定在电路中每一个功能模块的传输延迟是Xns。这个模块可能是一个简单的门，也可能是一个更复杂的功能模块，但这里为了讨论方便，我们假定每个模块的延时是相同的。

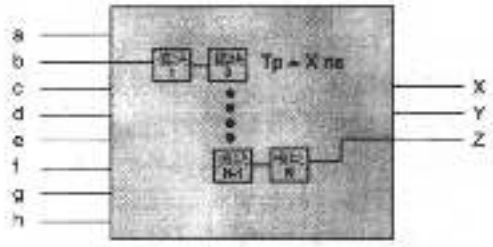


图13-3 通过一系列功能模块的传输延迟

同样，我们还假设，通过对这个电路的分析得出从输入b到输出Z的路径是该电路的最长路径。换句话说，从输入端b必须通过N个门才能到达输出端Z。因此，不管在什么情况下，从a到h的一组新输入进入电路时，必须要等到输入端b的信号最终到达输出端Z时，我们才认为电路是稳定，这时在输出端X、Y、Z上的数据才是正确的。

如果每个功能模块的传输延时都是Xns，那么通过该电路的最大传输延时为N×Xns。这里我们给出一个实际的数据。令X=300ps（30×10^{-12}秒），N=6，则传输延迟为1800ps（1800ps）。如果该电路是某同步数字系统中的一部分，这个系统由一个时钟驱动，而且我们希望系统能够在一个时钟周期内完成所有任务，那么这个系统可以设定的最大时钟速率为（1/1800ps）=556MHz。

358 请记住整个计算机的最高运算速度将取决于这一条电路路径。那么我们如何来提高计算机的速度呢？有如下一些方案可以选择：

1. 通过改进到较快的IC制作工艺来降低传输延时；
2. 减少信号传输路径上的门数量；
3. 解雇硬件设计师，雇佣技术更好的设计团队；
4. 使用流水线技术。

上面这些选项都是我们通常考虑的一些做法，但通常开发团队会选择第4种方案，高层管理者则会选择第3种方案。现在我们来研究一下第4种方案，如图13-4，系统的每一级有3个模块的传输延迟，即900ps。当然，为了稳当可靠我们还要加上D型寄存器的传输延迟，这是必要的一部分代价。假设在时刻t=0，有一个时钟上升沿，数据到达了输入端a到h。经过900ps之后，稳定的中间信号出现在第一个D触发寄存器的输入端D0到D10。之后一个新的时钟沿到达，数据就会通过D触发寄存器，经过一定的传输延迟后信号到达D触发寄存器的输出端Q0到

Q10。这样经过900ps + t_p（寄存器的传输延迟）后，数据将会通过流水线的第二级，稳定地出现在第二个D触发寄存器的输入端。在下一个时钟沿到达后数据就会通过第二个D触发器，电路的最终结果就出现在输出端X、Y和Z。

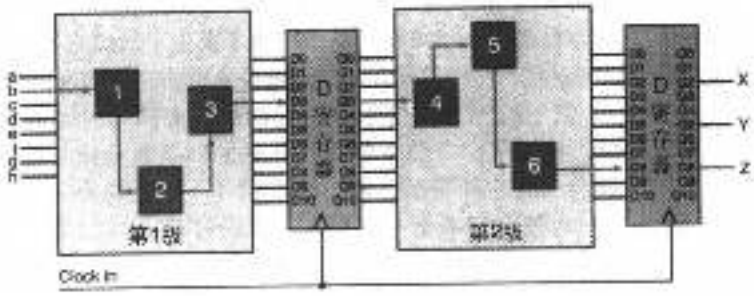


图13-4 数字电路的两级实现。通过每级的传输延迟从6个门延迟降低到3个门延迟

我们把这个例子再简化一下，假设通过D寄存器的传输延迟为0，这样就只需要考虑在两级中的功能模块的传输延迟。在这种情况下，新的数据通过两级流水线到达输出端仍然需要1800ps。但是，这与非流水的情况相比存在一个很大的差异。那就是在每次时钟上升沿的时候我们都可以输送新的数据到第一段的输入端，因为我们使用了D寄存器来进行中间信号的存储和同步，这样在第一个流水段处理新的信息时第二个流水段仍然能够处理之前的输入数据。

因此，尽管在处理头一个数据时使用流水线需要花费的时间没有改善（在这个例子中是两个时钟周期），但是每一个接下来的时钟周期在输出端（X、Y和Z）都会有结果输出，不再需要两个周期。

我们在第11章中讨论过ARM的指令集体系结构，这个体系结构与ARM7TDMI内核有紧密关联，这个内核的CPU设计有一个3级流水线。ARM9有一个5级流水线。见图13-5。

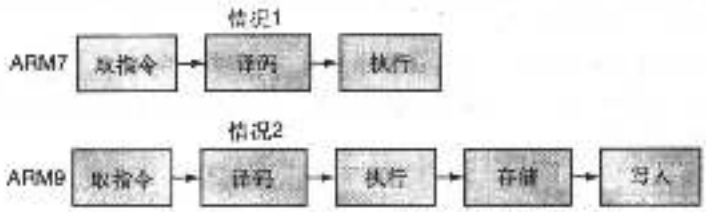


图13-5 ARM7和ARM9 CPU内核的流水线体系结构

在取指令阶段，指令被从存储器中取出来，在译码阶段，32位的指令字被译码，确定了指令序列。在执行阶段，指令被运行，相应的结果被写回到寄存器。ARM9TDMI的内核采用了5级流水。增加的两级分别为存储和写回，它们使得ARM9的指令吞吐量比ARM7大概提高了13%[7]。其中的原因就体现了多级流水线设计的优势。在ARM7中，执行阶段需要做三件事情： 359

1. 读取源寄存器
2. 执行指令
3. 把结果写回到寄存器

在ARM9的设计中，寄存器在译码段读取，执行段只进行指令的执行操作，再由回写段把结果写入到目的寄存器里。存储段是ARM9中一个独特的流水段，在ARM7中没有与之对应的部件，ARM7只支持单一的存储空间来存放指令和数据。当取一条新的指令时，不能同时进行数据的存取操作，因此，在出现冲突时流水线必须等待存取操作或者取指令操作结束。这就是冯·诺依曼体系结构的瓶颈，ARM9采用了数据和指令分开存储的方式，在存储段，存取操作能够与第1段中的取值操作同时进行。

到目前为止，我们讨论的流水线技术都是与提高速度相关。为了提高处理器的速度，我们需要做的事情就是让流水线的每一段具有更细的粒度，这样我们就可以提高时钟的频率。然而，基于这种思想设计的处理器也存在一个问题。实际上，有很多潜在的冒险情况会使得流水线不能达到最佳的效率。例如在ARM7结构中，如果遇到一条分支语句，发生了转移，那么我们怎么办呢？本来在转移指令后面的两条指令已经进入了流水线，但是由于发生了转移它们全都无效了。换句话说，我们只能清空（flush）流水线，从转移指令的目标地址处重新开始读取指令放入流水线。

回忆一下图13-4，第一个新的数据通过流水线得到结果需要两个时钟周期。设想一下一个3级流水的设计，如ARM7，从转移指令结束到其转移到的那条目标指令执行结束需要花费3个时钟周期。每次跳转时就会出现这种情况，需要等待额外一些时钟周期，这将降低流水线
360 的吞吐量。由于大多数程序平均每5到7条指令中就有一条转移指令，如果这样的转移发生的比较多，系统的速度将会变慢。现在考虑一下7或9级流水的情况，每一次非顺序的读取指令对处理器代码流的高效运行都是一种潜在的阻碍。

我们将会在这一章的后面讨论一些解决这个问题的方法，但是现在我们只需要了解到流水线体系结构并不是像我们想像中的那么完美。最后，在我们继续讨论之前，先做一点点小节收尾工作。首先，重要的一点是流水线不一定要与整个CPU的时钟频率相同。换句话说，我们可以用一个分频电路产生一个新的时钟，其频率只有系统时钟周期的1/4。然后我们用这个慢的时钟去控制流水线，而较快的系统时钟则用来在每一段流水内控制状态机的运作。其次，在程序的正常执行过程中，某一阶段的操作也很有可能会导致流水线停止运行。存储器里存取数据或者取指令等等往往会比内部操作花费更长的时间，因此每次外存操作都很容易导致流水线暂停一到两个时钟周期。

让我们回到之前讨论的体系结构，看看简单的、无流水线的情况。首先，我们必须意识到处理器是一个比较昂贵的资源，正如一个贵重的机器一样，应该尽量让它保持忙碌的状态。空闲的处理器是对空间、能量、时间等的浪费，我们需要找到一个方法来提高性能。为了弄清楚这个问题，我们先来看看一个处理器（如68K）是如何执行一条简单的驻留内存的指令的，如 `MOVE.W $XXXX, $YYYY`

根据68K处理器的程序员手册，这条MOVE.W指令需要40个时钟周期才能执行完成，而该处理器中指令执行需要的最少时间仅为7个时钟周期，那么多的时钟周期到底花在了什么上呢？如图13-6所示为其执行过程。

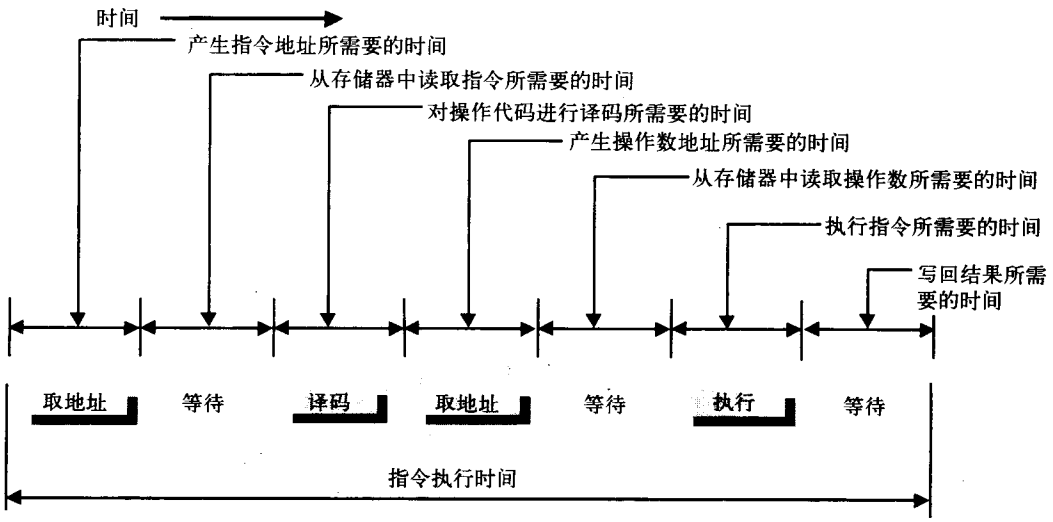


图13-6 68K处理器中MOVE指令的执行过程

一个新的取指令周期始于程序计数器的内容被读取到地址总线上。经过几个时钟周期之后，存储器中指定位置的指令操作代码字被读取出来。

1. 处理器对指令的操作代码进行译码。
2. 产生第一个操作数（源地址）的存储器地址。
3. 从存储器中读取源地址中的数据。
4. 从存储器中读取第二个操作数（目的地址）。
5. 将数据写入到目的地址中。

在这些活动进行时，处理器的其他大部分功能模块都处于闲置状态。由此可以看出，流水线技术的另一个潜在的优势在于把这些工作划分成几个小任务（级）来完成，使得CPU的资源得到最大限度的利用。这与在装配线上生产汽车的思想完全相同。执行每一条指令需要 361
的总时间是不变的，但是我们不用每次都等40个时钟周期才得到下一条指令的结果。正如我们在ARM处理器中所看到的，假设我们没有重置流水线，那么下一条指令只需要完成最后一级流水就可以输出结果。我们所提高的是整个系统的吞吐量（throughput）。

如图13-7，假设洗完一桶衣服总共需要2个小时的时间，其中包括4个步骤（洗涤、甩干、折叠和打包），那么我们串行地洗4桶衣服的话总共需要花8个小时。然而，如果我们把任务重叠起来的话，例如一桶衣服洗涤完毕，下一桶衣服立刻开始洗涤，这样总共只需要花费3个半小时（而不是8小时）的时间就能洗完4桶衣服。

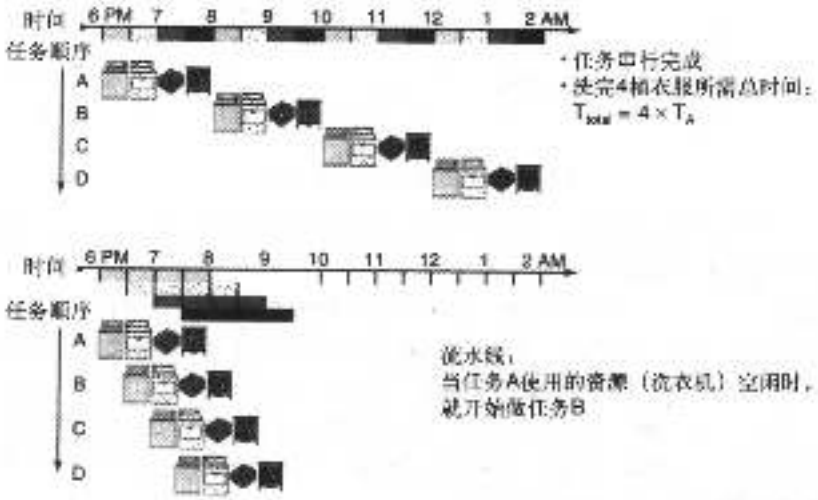


图13-7 洗衣流程：串行任务执行与重叠任务执行的比较。由Patterson和Hennessy[3]提供

一条指令的执行可以看成一个有着一定逻辑顺序的物理过程，它能被分解为一系列更细的步骤。以三级过程为例，没有流水线的情况下处理步骤为：

1. 做第一件事情；
2. 做第二件事情；
3. 做第三件事情；
4. 给出结果；
5. 回到第一步。

采用流水线技术后，处理过程变为：

1. 做第一件事情，并为第二步的执行保存结果；

2. 获得第一步的结果，并做第二件事情，为第三步的执行而保存结果；
3. 获得第二步的结果，开始做第三件事情；
4. 给出结果；
5. 回到第一步。

现在我们看一下3级流水线的执行过程，如图13-8所示。由于第1级的硬件在完成任务后
362 空闲下来，所以我们可以设计一个流水线。这样，在一个工作没有结束前我们就可以在第1级上开始执行下一个工作。我们把第一个工作从开始执行到离开流水线需要花费的时间称为流经时间（flowthrough time），把通过流水线得到两条连续的结果之间花费的时间称为时钟周期时间（clock cycle time）。

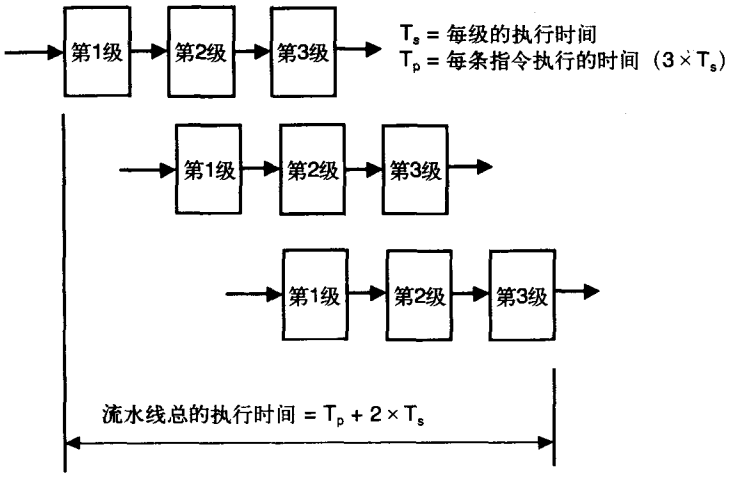


图13-8 将处理过程分解为流水级

可以看出，对于有流水线的情况，执行3条指令需要花费的总时间为$5 \times T_s$。在没有流水线的情况下，3条指令顺序地执行将会需要$9 \times T_s$的时间。

现在我们暂停一下问一个合理的问题。如我们前面提到的那样，程序执行时平均每5~7条指令会出现一次跳转。也就是说JSR、BRANCH或者JMP指令会以较为频繁的出现导致流水线的处理过程比我们之前所假设的情况要复杂得多。例如，在图13-8中如果第一条离开流水线的指令是BNE，而且发生了跳转，那会出现什么情况呢？这是一个不易回答的问题，在本书中我们不试图去解决这一问题。然而，我们可以研究一下影响流水线效率的一些因素，对处理现实系统中的这些问题给出一些通常的建议。

现在我们考虑这个问题：一个什么样的体系结构有助于我们设计一个基于流水线的计算机呢？

- 所有指令的长度都相等；
- 只有少数几种指令格式；
- 存储器操作数只出现在load和store指令中；
- ALU操作只发生在寄存器之间。

什么样的体系结构会让CPU设计者们想辞职变成科罗拉多州Boulder的滑稽演员？

- 变长的指令
- 有着可变执行时间的指令
- 广泛的指令和寻址方式

- 既有寄存器操作数又有存储器操作数的指令

换句话说，RISC处理器易于采用流水线结构，但是CISC处理器要困难很多。但并不是说不能设计一个基于CISC的流水线，实际上也是可以的，只不过要在传统的CISC结构上设计一条流水线非常的麻烦。例如，AMD 64位系列的处理器采用了12级流水线，其中有5级专门用 363
来对x86 CISC指令进行分解，以使得它能在流水线上继续进行处理。

现在我们需要考虑一下其他的影响因素，它们与体系结构的类型关系不大，主要是算法类型方面的影响。这些影响取决于体系结构和代码流互相协作的优劣。影响流水线的另外三个因素是：

- 结构的影响：假设我们只有一个存储器，在流水线的某一级需要读取一个操作数时，另一级可能需要读取一条指令。
- 控制的影响：我们需要考虑转移指令。
- 数据的影响：某条指令的执行依赖于前面某条指令的执行结果，但是那条指令还在流水线上执行。

我们看看ARM体系结构是如何处理结构影响的。通过设置两个独立的存储空间，维护独立的数据存储器和指令存储器，数据的读取和指令的读取能够独立地同时进行。转移语句是难免的，我们怎么来减轻转移语句带来的损失呢？而且，流水线的规模越大，转移语句产生的影响也越严重。现在有一个复杂的流水线，我们得到一个指令序列，正在流水线的各个级中进行着译码和执行等任务。那么，往往会碰到这种情况，一条转移指令后面的指令已经在执行或几乎执行完毕的时候，处理器发现需要进行转移。

解决该问题的一个方法是分支预测（branch prediction）。也就是说，让CPU通过学习具备一定的智能来预测将要发生的跳转。Turley[9]就提出：研究表明在碰到转移指令时向后跳转发生的可能性比较大，而向前跳转发生的情况比较少。从你在汇编语言习题中写出的循环语句，你可能会猜到这个原因。基于这一点，当CPU遇到一个转移指令时，如果跳转的目的地址比当前程序计数器的地址要小，那么流水线就自动从新的地址开始装载指令。如果目的地址比当前的地址大，那么就还是按照正常顺序转载。当然这不会总是一帆风顺的，当预测失败时，流水线必须清空，然后从正确的地址处重新开始。这样当然会使执行的速度变慢，但不会产生严重的后果。

解决问题的另一个方法叫做动态分支预测（dynamic branch prediction）。假设处理器维护了很多个小型的二进制计数器，即2位的计数器。这个计数器最小值为00，最大值为11。最小值和最大值为终态，即饱和态。当计数器在状态00时，如果继续减小，状态仍为00。同样，在状态11时，如果继续增大，状态仍为11。

4个状态值的意义分别为：

- 00 = 强不接受
- 01 = 弱不接受
- 10 = 弱接受
- 11 = 强接受

对每个分支都尽可能设置一个计数器与之相关联。每当一个分支发生跳转，相应的计数器就递增；如果没有发生跳转计数器就递减。这样分支预测逻辑就能根据之前统计的结果对跳转做出预测。显然，预测结果的好坏取决于计数器位数和预测电路中计数器的个数。

有些处理器，例如AMD的29K系列采用了一个专用的片上高速缓存，称为分支目标cache（branch target cache）[9]，用于存放以前发生跳转的前4条指令。在下一章里，我们将更深入地讨 364
论高速缓存（cache）问题，但是现在我们先假定已经有一个硬件实现的算法，它在不断地更新

分支目标cache存储器，加入最近访问的分支，覆盖最久未访问的分支。把分支预测机制和分支目标cache结合起来，处理器就能够降低因为指令的乱序执行带来的影响。

现代的RISC处理器几乎和相应的CISC结构一样复杂，但正是RISC的指令集体系结构使得它们能采用流水线和更高的时钟频率来获得速度的提升。图13-9是Freescale公司（Motorola的前身）603e PowerPC RISC处理器的结构示意图。

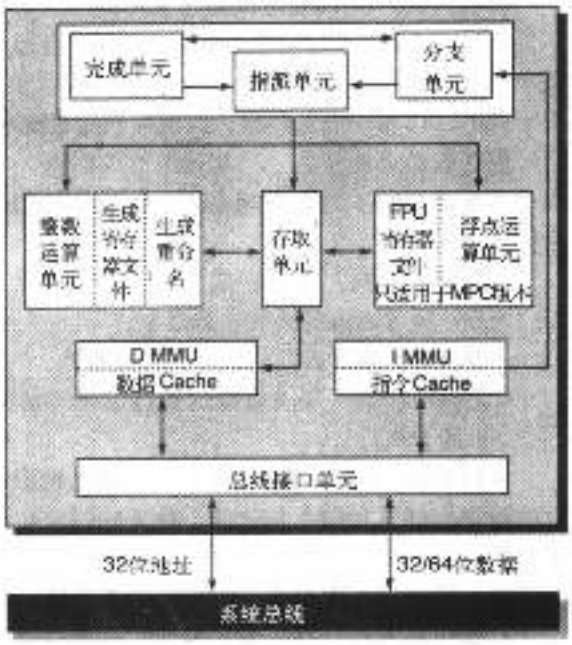


图13-9 PowerPC 603e RISC处理器，来自Freescale公司

你可能熟悉这个处理器，因为603e被用在了苹果计算机的Macintosh G系列上。603e是超标量处理器，有3个独立的指令处理系统，能够通过独立的流水线并发地执行指令。

流水线体系结构的概念也引导我们把计算机看成是由内部资源、流水线和控制系统等组合起来的一个松散的系统。前面提到的PowerPC 603e就可以看成这样一个系统，它与68K处理器紧密集成的体系结构相当不同。虽然从图13-9中看起来不明显，但我们可以把处理器画成图13-10的样子。

每一条指令通过流水线时，控制系统都会不断地监视可用来执行指令的可得到的资源。
365 这就好比去自助洗衣店洗衣服一样，洗衣店里有多个洗衣机、烘干机和叠衣服的桌子。如果一个资源在使用中，另一个空闲，那么在这种情况下，根据墨菲法则所有的设备要么在使用中，要么就是坏的。除非你正在忙，没功夫去用。

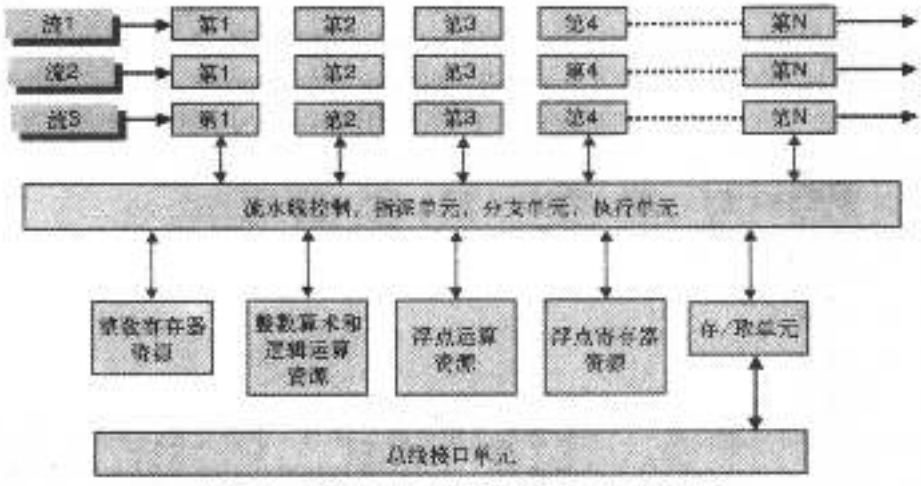


图13-10 将流水线处理器看作资源和流水线的集成

如果我们需要考虑的是："有多少衣服被送到了洗衣店，单位时间有多少衣服被处理完毕"，那么显然洗衣店处理流程的效率要比某一个人在家里洗的效率要高得多。当然，把一套洗衣设备安装在洗衣店里要比安装在家里花费的代价还是要大一些。

把指令执行的部件分割成各种资源，然后在这些资源可得到时再使用这些资源，我们将这个过程称为动态调度（dynamic scheduling）。处理器中有各种复杂的硬件来执行调度的任务，这些硬件都试图寻求它们所能执行的指令。在某些情况下，它将尝试不按照正常顺序来执行指令，它可能会假定某个跳转不会发生，然后提前执行转移语句后面的指令。我们把这种情况称为投机执行（speculative execution）。

关于流水线的效率还有一个常常被忽视的方面，就是支持处理器的编译器的效率。设计CISC处理器的初衷之一就是复杂指令能够使编译器的工作变得容易一些。然而，事实上并非如此，因为编译器的设计者往往并没有利用那些特殊的指令。

另一方面，RISC体系结构可以有不同的策略。编译器对处理器吞吐量的提高担负着更多的责任。如果使用了一个低效的编译器，RISC体系结构的许多优势都可能不复存在。不用具体说，相信大家都知道有一个商业的RISC微处理器支持3个独立的C++编译器，其中最好的编译器性能是最差的那个的两倍。也就是说，前者在处理器频率只有后者一半的情况下和后者的效率都是持平的，这是多么大的一个差距。

图13-11显示了与编译器相关的一些数据。这些数据是在Intel x86体系结构的PC机上得到的，我们可以看到即使在这样一个结构上编译器对生成代码的效率影响有多么大。譬如说Dhrystone测试基准，编译器I和编译器C的性能就相差6倍。对此感兴趣的读者可以去查一下原始的资料看看这些编译器和厂商的名字到底是什么。 366

测试基准	编译器A	编译器B	编译器C	编译器D	编译器E	编译器F	编译器G	编译器H	编译器I
Dhrystones/s									
Dhrystone	1,847,675	1,605,696	**2,993,718**	2,496,510	2,606,319	2,490,718	2,263,404	2,450,608	484,132
time (ms)									
Int2string a (sprintf())	7642	5704	5808	7714	7933	9419	7802	7813	**5539**
Int2string b (STLSoft)	3140	1289	3207	1679	**1156**	1624	1808	1843	DNC
StringTok a (Boost)	4746	3272	DNC	6809	**1450**	2705	2641	2341	DNC
StringTok b (STLSoft)	636	809	**280**	385	382	579	383	406	DNC
RectArr (1 iteration)	1082	910	997	1590	859	915	**824**	887	DNC
RectArr (10 iterations)	6922	3168	5589	3853	1649	1995	**1533**	1828	DNC
zlib (small)	92	110	88	92	87	87	91	**78**	90
zlib (large)	**8412**	12,550	8847	11,310	9390	10,875	10,266	9117	15,984

图13-11 9个测试基准针对x86的一系列编译器编译后的执行时间（ms），黑体字为最好的结果。来自Wilson[10]

现在我们来研究一下RISC体系结构的一些特性，以及这些特性与流水线技术的关系。大部分的RISC处理器都具备如下所示特性中的很多特性：

- 指令比较简单。
- 存储器和寄存器之间交换数据只能通过LOAD和STORE指令来实现。
  - LOAD = 存储器到寄存器
  - STORE = 寄存器到存储器
- 所有的算术操作都是在寄存器之间进行的。
- 指令长度单一。
- 指令格式唯一或者很少。
- 指令集是正交的（没有特殊的寄存器来存放地址和数据）。
- 指令的功能几乎没有重叠。
- 寻址模式单一或者很少。
- 几乎所有的指令都是在一个时钟周期内完成的。
- 主要针对速度进行优化。
- RISC处理器可能会有很多个寄存器。

– AM29K有256个通用寄存器
– 这些寄存器对编译优化非常有用，使得一些中间结果不需要写到外存
• 采用多流水线技术每个时钟周期能够执行多条指令。

现在，绝大多数高性能处理器都是RISC体系结构。下面是一些现代RISC处理器的例子：

• Motorola、IBM: PowerPC 8XX、7XX、6XX、4XX
• Sun: SPARC
• MIPS: RXXXX
• ARM: ARM7、ARM9
• HP: PA-RISC
• Hitachi: SHX

直到10年前，计算机专家们都还在争论RISC和CISC的优缺点。然而，从那时起RISC体系结构已经在性能比较上取得了胜利。RISC的基本硬件体系结构更简单、更便宜，速度更快。这样在速度上的提升远比通过增大指令集获得的提升大得多。而且对编译器而言，实现RISC结构的编译也要比CISC简单得多。

现在针对Intel x86体系结构的编译技术做得非常完美，因为其编译器的设计者们
367 (Microsoft) 已经投入了巨大的努力使得它与RISC一样高效。我们都知道，在RISC体系结构中引入并行技术更加容易。由于指令格式单一，流水线技术也更容易预测。此外，现在的应用程序大都是数据密集型的。台式机用于多媒体和游戏，工作站用于数据密集型的应用，如设计和模拟。嵌入式系统则用于游戏（如任天堂）和远程通讯应用（如路由器、网桥、交换机）。

Motorola在68060时停止了CISC的开发，并取代680X0系列设计了两个新的RISC体系结构：ColdFire和PowerPC。AMD停止了586的研究，Intel也停止了最初的Pentium设计方案。AMD和Intel的现代处理器都融合了RISC和CISC的技术，而且被设计成能够兼容以前的Intel x86指令集，并通过在处理器内部将那些指令翻译成一个类似RISC的指令序列，体系结构能非常迅速地处理这些指令。

现在我们都知道，RISC处理器使用流水线技术来加速指令的译码和程序的执行。RISC结构通常不容许程序自行修改代码，也就是通过对指令空间进行写覆盖来修改一条指令。纯粹的RISC处理器采用哈佛体系结构来消除指令与数据共享一条地址总线和数据总线而带来的冯·诺依曼瓶颈问题。然而，大多数先进的处理器仍然只有一条总线与外存储器相连，不过它们采用了内部指令和数据cache来替代哈佛体系结构。

RISC处理器采用众多的寄存器堆来降低从存储器到寄存器的延迟。这些寄存器用于广泛的目的，任何寄存器都可以作为一个间接寻址的存储器指针。此外，一些RISC处理器还有独立的浮点数寄存器堆。所有的RISC处理器都有独立的指令译码和指令执行单元，很多甚至还有独立的浮点运算单元。

RISC处理器常采用*延迟转移*（delayed branch）技术来降低转移语句带来的影响。在*转移延时槽*（branch delay slot）里，CPU总是执行指令。如果转移发生了，指令就被取消。此外，RISC处理器还会利用编译器来降低*装入–使用惩罚*（load-use penalty）。这种惩罚之所以会产生，是由于操作数处于存储器中的操作要比内部寄存器间的操作需要更长的时间。我们必须首先取来存储器中的操作数，然后才能使用它。这就出现了问题，假设我们在做一个算术运算，两个操作数相加得到一个结果。第一条指令从存储器中读取一个操作数放入寄存器，第二条指令把这个寄存器和另一个寄存器中的操作数相加得到一个结果。由于大多数存储器操作比寄存器之间的操作要花费更长的时间，所以流水线需要等待至少一个时钟周期才能把数据读入寄存器。这里就需要编译器发挥作用了。一个好的编译器应该知道什么时候需要进行

读取数据的操作，把这个操作提前进行，使得在运算操作执行时两个操作数都已经被读取到了对应的寄存器中。这是一个很好的乱序执行的例子。当然，如果将一条指令提前执行会破坏算法的逻辑流程，我们就不能这么做了。

在讨论RISC体系结构对性能的提高时，我们不能不提针对RISC体系结构的编译技术的改进。RISC计算机应该使用优化的编译器以尽可能地利用处理器频率高的特性。去读一个优化的RISC编译器的汇编语言输出是没有任何意义的，因为编译器通常都会把指令重新组合以充 368
分利用处理器的并行特性。这些编译器采用的技术有：

- 只要有可能，就把独立的指令放入读取延时槽或者转移延时槽。
- 尽量利用寄存器来降低从存储器存取数据带来的延迟。
- 将LOAD指令尽可能地放到指令序列的前面以降低读取数据延迟的影响。
- 把计算转移地址的指令尽量提前以降低转移延迟的影响。
- 尽可能提前执行条件判定指令，使得转移指令序列能够得到优化。
- 修改转移的条件语句使得在大多数情况不发生跳转。
- 把循环语句展开（内嵌循环代码）以避免指令乱序执行带来的影响。
- 尽量扩大程序中的一个基本模块（basic block，一个基本模块只有一个入口和一个出口，且没有内部循环）的大小。

总结

- 流水线技术是通过提高指令吞吐量而不是通过减少指令执行时间来提高性能。
- 冒险导致了复杂化。
- 为了使处理器速度更快，我们使用独立的多条指令流水线来实现在同一时刻执行多条指令，我们称之为超标量体系结构。
- 编译器能够对它们生成的指令代码重新进行排序从而提高流水线的性能。
- 硬件调度技术之所以会被现代RISC微处理器所采用是因为处理器在内部要通过动态调度使得它的各个资源尽量处于忙碌状态。

参考文献

[1] David A. Patterson and David R. Ditzel, *The Case for the Reduced Instruction Set Computer,* ACM SIGARCH *Computer Architecture News,8* (6), October 1980. pp. 25–33.

[2] Manolis G. H. Kavantis, *Reduced Instruction Set Computer Architectures for VLSI,* The MIT Press, Cambridge, MA, 1986.

[3] David Resnick, Cray Corporation, Private Communication.

[4] J.E. Thornton, *Design of a Computer: The Control Data 6600,* Scott, Foresman and Company, Glenview, Ill, 1970.

[5] W.C. Alexander and D.B. Wortman, *Static and Dynamic characteristics XPL Programs*, pp. 41–46, November 1975, Vol. 8, No. 11.

[6] Jim Turley, *Starting Down the Pipeline Part I,* Circuit Cellar, Issue 143, June, 2002, p. 44.

[7] See Sloss et al, Chapter 11.

[8] David A. Patterson and John L. Hennessy, *Computer Organization and Design, Second Edition*, ISBN 1-5586-0428-6, Morgan-Kaufmann Publishers, San Francisco, 1998, p. 437.

[9] Jim Turley, *op cit,* p. 46.

[10] Matthew Wilson, "Comparing C/C++ Compilers," Dr. Dobbs Journal, October 2003, p. 16.

[11] Daniel Mann, *Programming the 29K RISC Family,* ISBN 0-1309-1893-8, Prentice-Hall, Englewood Cliffs, NJ, 1994, p. 10.

[12] Mike Johnson, *Superscalar Microprocessor Design,* ISBN 0-1387-5634-1, Prentice-Hall, Englewood Cliffs, NJ, 1991. 369

习题

1. 看如下两个汇编语言的代码片段。为了简单起见，我们采用的是68K的汇编语言指令。在流水线上执行时其中的一个代码片段比另一个效率高。指出该代码片段，并说明原因。

代码片段A	
MOVE.W	D1,D0
MOVE.W	#$3400,D2
ADDA.W	D7,A2

代码片段B	
MOVE.W	D1,D0
ADD.W	D0,D3
MULU	D3,D1

2. 简述一下增加一条流水线的级数会带来什么样的问题。现代的微处理器，如Intel的奔腾和AMD的Athlon，它们运行测试码的效率大致相当，但是AMD的处理器时钟频率只有奔腾处理器频率的65%左右。简要说明为什么会出现这种情况。
3. 对于下面每条汇编语言指令，分别指出它们是否符合RISC指令的典型特征。为了简单起见，我们采用68K的助记符来表示指令。
   a. MOVE.L $0A0055E0, $C0000000
   b. ADD.L D6, D7
   c. MOVE.L D5, (A5)
   d. ANDI.W #$AAAA, $10000000
   e. MOVE.L #$55555555, D3
4. 指令：

   MOVEM.L D0-D3/D5/A0-A2, -(SP)

   是一个典型的CISC指令（尽管ARM指令集体系结构中也有一个类似的指令）。用一串RISC指令重写这条指令。假设有效的寻址模式只有：
   a. 立即数寻址
   b. 数据寄存器直接寻址
   c. 地址寄存器直接寻址
   d. 地址寄存器间接寻址

370 记住：你所用的指令结构必须基于典型的RISC指令集体系结构。

5. 假设LD（LOAD）指令用于把存储器中的数据读入到寄存器，ST（STORE）指令用于把寄存器中的数据写入存储器。下面的一段指令序列是某RISC处理器的汇编语言程序的一部分，该处理器有64个通用寄存器（R0-R63），每个寄存器都可以作为地址寄存器（存放存储指针）或者数据寄存器，存放算术运算的操作数。用一两句话简述为什么该指令序列属于RISC类型的处理器指令。

   假设指令格式为：操作代码 源操作数，目的操作数

```
*************************************************************
* 该代码序列将存储器中的两个数相加后，将结果写入了第三个存储地址。
*************************************************************
addr1   EQU     $00001000       * 第1个操作数
addr2   EQU     $00001004       * 第2个操作数
result  EQU     $00001006       * 结果

start   LD      #addr1,R0       * 建立第1个地址
        LD      #addr2,R1       * 建立第2个地址
        LD      #result,R3      * 指向结果的指针
        LD      (R0),R60        * 获得第1个操作数
        LD      (R1),R61        * 获得第2个操作数
        ADD     R60,R61         * 相加
        ST      R61,(R3)        * 存储结果

* 代码序列结束。
```

6. 编译器优化程序的一个方法是把一些特定的循环结构（例如for循环）转化成一段较长的内联代码。那么这样做的优势在哪里呢？
7. 假定某个处理器有一个7级的流水线，运行的时钟频率为100MHz。流水线被设计为每级流水需要2个时钟周期。假定有一个包含10条指令的基本程序块进入流水线。
   a. 假定流水线上没有阻塞发生，那么从第一条指令进入流水线到所有指令执行完毕需要花费多长时间？
   b. 如果在执行时，每两条指令中有一条指令会导致流水线阻塞4个时钟周期，那么10条指令全部执行完毕需要多少时间？
8. 下面是一段68K处理器的汇编语言指令代码。把指令执行顺序重新调整一下，以便这些指令在流水线处理器体系结构上执行的效率更高。

```
MOVE.W          D1,D0
ADD.W           D0,D3
MULU            D3,D1
LEA             (A4),A6
MOVE.W          #$3400,D2
ADDA.W          D7,A2
MOVE.W          #$F6AA,D4
```

第14章 存储器、高速缓存和虚拟存储器

学习目标

- 解释使用cache的原因以及如何组织cache；
- 描述各种cache是如何组织的；
- 设计一个典型的cache组织；
- 讨论cache性能的相关问题；
- 解释虚拟存储器是如何组织的；
- 描述计算机体系结构是如何支持虚拟存储管理的。

14.1 高速缓存简介

在介绍高速缓存（cache）以及基于cache系统的相关主题之前，我们先回顾一下以前讨论过的存储器类型。存储器的主要类型有静态随机存储器（SRAM）、动态随机存储器（DRAM）和非易失性只读存储器（ROM）。SRAM存储器基于交叉耦合连接的反向逻辑门原理。输出的值被反馈到输入，使得逻辑门被锁定在一个或另一个状态。SRAM存储器的速度很快，但每个存储单元需要5或6个晶体管才能实现，因此它往往比DRAM存储器成本要高。

DRAM以电荷的形式在一个微小的电容上存储逻辑值。如果不定期刷新电荷，电容上的电荷就会发生泄漏，因此这种类型的存储器必须定期进行读取操作。这就是为什么它被称为动态RAM而不是静态RAM的原因。DRAM的访问周期也比静态RAM要复杂得多，因为刷新周期也必须考虑在内。

然而，DRAM存储器的最大优势在于它的高密度和低成本。现在，你只花60美元就可以为你的PC买一个单列直插内存模块（SIMM），这是具有512M容量的DRAM。在这样的价格下，我们就能付得起将DRAM接口管理放到一个专用芯片上的费用，而该专用芯片位于CPU与存储器之间。如果你是一个电脑爱好者，喜欢对自己的PC进行升级，那么你肯定会去买一个新的主板，并采用芯片组AMD、nVidia或者VIA。在决定计算机的性能方面，芯片组已经变得和CPU同样重要。

随着应用程序和操作系统的日益复杂，计算机系统也相应地需要越来越大的存储量。在写这一章时所用电脑就有1024MB（1GB）的存储量，现在看来这个容量似乎高于平均水平，但是不出三年，它很可能就是推荐配置的最低容量。不久以前，10MB的硬盘容量还让人觉得很大，现在你花100美元左右就能买到一个200GB的硬盘，存储容量增长了10 000倍。鉴于我们对存储要求的无止境增长，不管是对易失性存储器（如RAM）还是对档案存储器（如硬盘），我们都宜于从体系结构的角度来寻找对这种复杂性的管理方式。

存储器的层次

存储器是分层次的。这里的层次并不是说某些存储器比其他存储器更重要，而是从存储器“靠近”CPU的程度来区分的。越“靠近”CPU，等级就越高，反之就越低。注意，我们说到“靠近”时是指一般意义上的靠近，而不仅仅是指“物理位置上的靠近”（尽管这种靠近

也是一个重要的因素）。为了尽量提高处理器的存吐量，速度最快的存储器往往被放在最靠近处理器的位置。这个速度最快的存储器同时也是最昂贵的。图14-1是对存储器层次的一个定性表示。从这个金字塔的顶端开始，越往下的等级，对应类型的存储器访问时间就越长。

我们来看一些实际的例子。现在，SRAM的访问时间是2~25ns，费用大概是每兆字节50美元。DRAM的访问时间是30~120ns，费用为每兆字节0.06美元。硬盘的访问时间是1千万~1亿ns，费用约为每兆字节0.001~0.01美元。可以看到，随着等级的降低，容量按指数级增长，相应的访问时间也是按指数级增长。

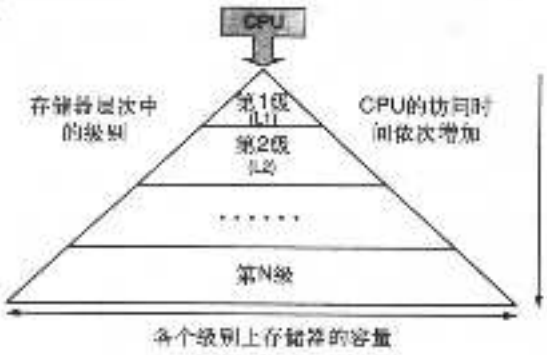


图14-1 存储器层次。离CPU越远，容量越大，访问时间越长

图14-2显示出了一个典型计算机系统中的存储器层次。你家里用的个人电脑可能就是这样的。注意，这个系统中有两个独立的cache，一个在芯片内，通常被称为L1 cache，另一个在芯片外，称为L2 cache。显然，级别越低，存储容量越大，访问速度越慢。我们可以想像出来，在这个金字塔的最底层就是Internet了。在这一级，存储容量几乎是无限大的，访问时间也常常是无限长的。 373

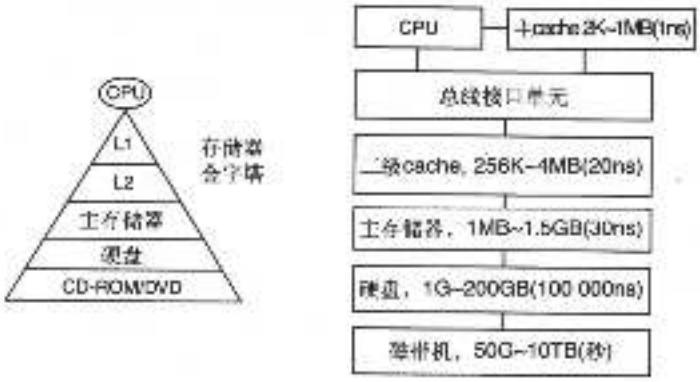


图14-2 典型计算机系统中的存储器层次

局部性

在继续讨论cache之前，我们得先弄清楚cache到底是什么。cache是一个邻近的本地存储系统。我们可以把CPU中的寄存器称为0级cache。此外，在CPU芯片上还可以看到另外一些更大的cache存储系统，这个存储器一般与CPU的频率同步，有时候偶尔也会比正常的访问速度慢一些。大多数处理器都有两个独立的初级cache，一个用于存放指令，一个用于存放数据。我们知道，这是哈佛体系结构的内部实现方式。

cache的用处源于程序所具有的一般特性——局部性（locality），有两种类型的局部性，它们分别以不同的方式表现了同样的规律。引用局部性（Locality of Reference）是指程序常常会访问最近访问过的数据和指令，或者与它们存储的位置邻近的指令和数据。程序往往是依次读取存储器中的指令来执行，循环操作往往是把相近的几条指令重复执行。从数据结构的角度来说，编译器把数组存放在内存中相邻的块内，而程序一般是顺序访问数组元素。而且，

编译器把不相关的变量存放在一起，例如存放在栈内的局部变量。*时间局部性*（temporal locality）是说一旦一个元素被访问后，它很可能在不久的将来再次被访问。*空间局部性*（spatial locality）是说邻近的元素还将很快被访问。

我们来研究一下在有2个级别的存储器等级结构中局部性原则是如何被利用的。该例子有一个较高级别的存储器（cache存储器）和一个较低级别的存储器（主存储器）。二级的结构意味着当需要的数据不在cache中时，我们要去低一级的主存储器中检索至少一个数据块。我们定义*cache命中*（cache hit）为CPU需要的数据或指令正好在cache中，相反如果不在cache中，则称为*cache不中*（cache miss）。

我们还需要定义*块*（block）作为数据传输的最小单元。一个块可以只有一个字节大小，也可以有几百字节大小。在实际应用中，一个块的大小一般是16到64个字节。说到这里，可能有人要问："为什么要从主存储器里读取整个块？为什么不只取我们需要的指令或者数据？"这是因为局部性规律告诉我们，如果需要的第一部分信息不在cache中，那么很可能接下来需要的信息也不在cache中，所以我们就把整个数据块都读入cache。

这样做还有另一个实际的原因。DRAM存储器要花一些时间来建立对第一个数据的访问。但是只要访问建立起来，CPU就可以只再花很小的额外开销就能从存储器中传输更多的连续字节，尤其是在一次有大量的数据要从存储器读取到CPU的时候。这种情况被称为*突发模式访问*（burst mode access）。现代的SDRAM存储器能很好地支持突发模式访问，这非常适合现代的处理器要求。突发模式访问需要花费一些时钟周期来建立环境，使得存储器的支持芯片集能确立突发访问的初始地址。然而，在地址被确立后，SDRAM就能在外部总线的每个时钟
374 周期内完成两个存储器读取周期。目前，在总线频率200MHz，存储器宽64位的情况下，在一次实际的突发传输中从存储器到CPU的传输速率能达到每秒3.2GB。

再打一个比方，用你日常生活中的事情来类比存储器的分级。想像一下你在书桌前不停地工作，处理教授按时分配下来的无穷无尽的任务。你在身边放了一些你最经常用到的书籍，譬如上课用的教材，把它们放在书桌上或者书架上。这些书放在手边，非常方便查阅，但是你只能放有限的一些书。

假设现在某个作业需要你去工程图书馆借另外的一本书来参考。工程图书馆里面的书显然比你放在身边的书多很多，但是找到需要的书花费的时间也更多一些。而且，如果在工程图书馆里找不到需要的书，那么接着你可能还得到华盛顿的国会图书馆去找。显然，在每一级图书馆，为了能获得更多的馆藏资料，我们就得花更多的时间。这里，我们传输的单元是一本书，所以在这个类比中，一本书就等同于一个块。

现在我们回顾一下，并按这个例子中的术语重新进行定义：

- *块*（block）：数据传输的单元（一本书）
- *命中率*（hit rate）：访问的数据在cache中的概率（在书桌上的概率）
- *不中率*（miss rate）：访问数据不在cache中的概率（1－命中率）
- *命中时间*（hit time）：从cache中读取数据所需要的时间（从书桌上拿一本书）
- *不中惩罚*（miss penalty）：把cache中的块用你需要的块替换所需要的时间（去图书馆取另一本书）

我们可以得到一个计算*有效执行时间*（effective execution time）的简单公式。这就是，给定了所需指令是否在cache中的概率，一条指令的平均执行时间。这里有一个细微的地方需要说明一下，不中惩罚就是由于处理器必须执行不在cache中的指令所产生的时间延迟。尽管大多数带cache的处理器容许你设置是否使用cache，但我们假设你使用cache。

$$\text{有效执行时间} = \text{命中率} \times \text{命中时间} + \text{不中率} \times \text{不中时间}$$

如果指令或数据不在cache中，那么处理器在读下一条指令前必须重新装载cache，而不只是直接到存储器中取指令。因此，还需要额外的时间来等待块从存储器载入到cache。

让我们做一个实际的例子。假设我们有一个带cache的处理器，时钟频率为100MHz。在cache中的指令要执行两个时钟周期。不在cache中的指令必须从主存储器中按块读入，块的大小为64个字节。从主存储器中读取数据需要先花10个时钟周期建立数据传输，之后处理器每个时钟周期能读取一个32位宽的字。假设cache的命中率为90%。

1. 这个练习的困难部分是计算不中惩罚，我们先把它求出来。

a. 时钟频率为100MHz ->时钟周期为10ns

b. 建立突发访问需要10个周期 = 10 × 10ns = 100ns

c. 32位宽的字 = 4字节 -> 64字节的块需要16次数据传输

d. 16 × 10ns = 160ns

e. 不中时间 = 100ns + 160ns = 260ns

2. 每条指令执行需要2个时钟周期，或者说20ns。

3. 有效执行时间 = 0.9 × 20 + 0.1 × 260 = 18 + 26 = 44ns

即使从这个简单的例子也可以看出，有效执行时间与cache的一些参数是息息相关的。有效执行时间是cache内指令执行时间的2倍多。因此，理论上效率可以提高一倍上下，这样一来设计者们就有得忙了。

现在有几个基本的问题：

1. 我们怎样才能提高cache命中率呢？

2. 我们怎样才能降低cache不中惩罚呢？

对于第1个问题，我们可以设置更大的cache。一个较大的cache能存放更多的主存储器中的数据，这样就能提高cache命中的可能性。我们还可以改变cache的设计。也许通过一些方法来组织cache可以让我们更好地利用cache中已经存在的数据。记住，与随机逻辑相比，存储器在芯片上要占用很大的空间，如果用几千个逻辑门来实现一个好的调度算法，可能比增加100K的cache更有效。

我们还可以从编译器设计者那里寻求帮助。也许他们能够对代码结构进行改进，从而使cache命中率有更好的贡献。当然，这个并不那么容易实现，因为cache的行为有时会与直觉相反。算法中一个小的改变有时就可使有效执行时间产生大的波动。例如在我的《嵌入式系统实验室》课里，学生们做实验试图微调一个算法，使得在没有cache和有cache情况下的运行时间差别最大化。我们将这变成了一个小竞赛，最好的学生得到的结果是15 : 1。

375

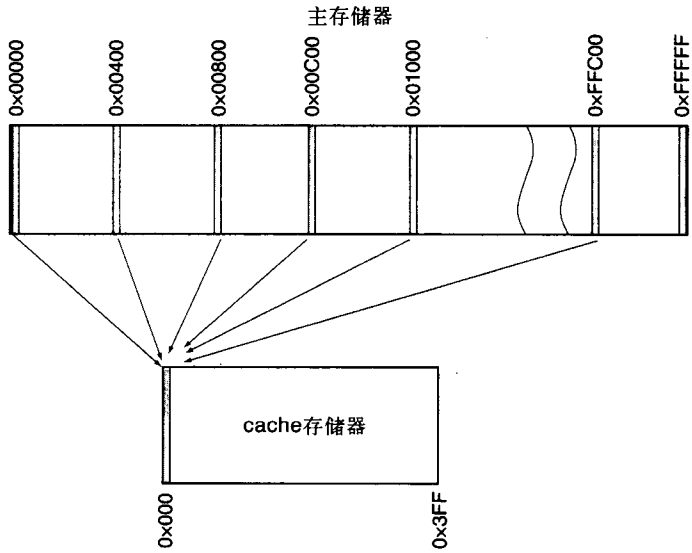
cache组织

我们将要涉及的第一个问题非常简单："我们怎样才能知道一个元素（指令或数据）在cache里面？"如果是在cache里，"我们怎么找到它？"考虑这个问题非常重要。要记住，你的程序是在主存储器中而不是在cache中被编写、编译并连接来运行的。一般来说，编译器并不了解cache，尽管它可以做一些编译优化来利用带cache的处理器。程序中引用的地址都是指主存储器地址，而不是cache地址。因此，我们需要设计一个方法，以某种方式把主存储器中的地址映射到cache中的地址。

我们还有另外一个问题。如果我们改变了一个值，而这个新值必须立刻写回到主存储器中去，那会怎样呢？从效率上来考虑，我们只把它写入cache就行了，但是这会导致一个潜在的灾难性问题，cache中的数据和主存储器中的数据不再一致了。最后，我们如何设计一个

cache使命中率达到最大呢？下面我们将尝试回答这些问题。

在第一个例子中，块的大小正好是一个存储器的字，这种cache设计我们称为直接映像cache（direct-mapped cache）。在这种方式下，低一级存储器中的每一个字刚好在cache中有一个可找到的位置。这样，对于cache中的每一个存储位置，低一级存储器中就有大量的位置与其对应，如图14-3所示。



376 图14-3 将1K的cache直接映像到1M的主存储器。cache中的每个位置映像到主存储器中的1024个位置

由图14-3可以看出，假设cache大小是1024个字（1K），主存储器大小为1 048 576个字（1M），则cache中的一个位置要映射到主存储器中的1024个位置。这没有什么，但是我们需要知道在某一时刻，这1024个单元中的哪一个位置在cache中，因此，cache中每个位置还需要包含更多的信息，以指出在主存储器中相应的位置。

每个cache存储位置包括多个cache入口，每个入口由几部分组成。每个cache位置包含与之映像的1024个主存储器位置之一中的指令和数据。每个cache单元还包含一个地址标签（address tag），这个标签指出1024个主存储器位置中的哪一个恰好在cache中的相应位置。关于这个问题值得我们进一步讨论。

地址标签

几节课前我们开始讨论存储器组织时，我们知道了分页的概念。在这里，你可以把主存储器看作是由1024个页组成的，每页包含1024个字。主存储器中的一页映像到cache中的一页。因此，主存储器中第一个字的二进制地址是0000 0000 0000 0000 0000，最后一个字的地址是1111 1111 1111 1111 1111。我们把它分成页和偏移。第一字的页地址就是00 0000 0000，偏移地址是00 0000 0000。最后一个页地址为11 1111 1111，偏移地址是11 1111 1111。

用十六进制地址来表示的话，我们可以称最后一个存储器字的地址为$3FF/$3FF（页/偏移）。这样做并没有改变什么，只是通过将位分组后把存储地址表示成与直接映像cache的地址表示相对应的方式。这样，每个cache中的页也要存放与之映像的主存储器的页地址。

377 现在，cache中的数据要么是主存储器中数据（指令、数据）的副本，要么是新存储的数据，还没有放入到主存储器中。数据的cache入口称为标签（tag），包含了块在主存储器中的

位置信息以及有效性（一致性）信息。因此，每个cache入口必须包含的信息有：主存储器中的指令或数据，块来自主存储器中的哪个页，以及cache中的数据是否与主存储器中的数据一致。如图14-4所示。

我们可以十分简单地总结一下cache的操作。我们必须使这种情况的概率最大：即每当CPU进行取指令操作或者读数据操作时，指令或数据可在cache中得到。对于许多CPU设计来说，用于cache管理的算法状态机设计都是公司的最高机密之一。这个复杂硬件模块的设计将极大地影响cache命中率，进而影响处理器的总体性能。

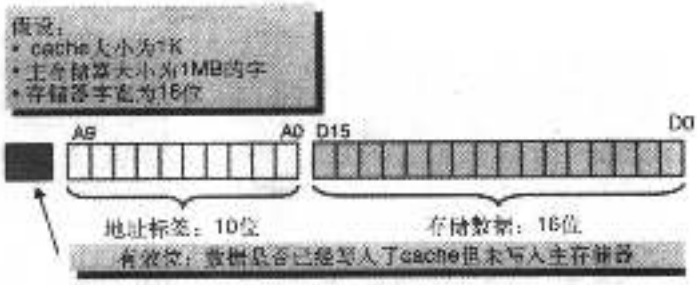


图14-4 将1K的cache直接映像到1M的主存储器上。cache中的每个位置块像到主存储器的1024个存储位置

绝大部分cache实际上都可以分为3个基本部分。我们实际上已经把各个部分都讨论过了，下面只是花一点时间总结一下：

- cache存储器：保存存储器中的映像。
- 标签存储器：存放地址信息和是否有效的位。确定数据是否在cache中，以及在cache中的数据是否和主存储器中的数据一致。
- 算法状态机：cache的控制机制。其主要功能是保证CPU所需数据在cache中。

说到这，我们一直在采用这样一个模型，就是cache和存储器间按块传输数据，而且每个块的大小为1个字。事实上，cache和主存储器都被划分成了相同大小的量，称为重充线（refill line）。一个重充线通常为4到64个字节（2的幂数次方），它是cache与主存储器之间交换数据的最小量。主存储器丢失一个字节就会导致对包含这个字节的重充线的全填充。这就是为什么大多数带cache的处理器都采用突发模式来访问存储器，而通常不会只读取一个字节。重充线就是我们前面讨论过的数据块的另一种叫法。

目前，常用的cache有4种类型：

1. 直接映像
2. 相联
3. 组相联
4. 区映像

其中最常用的一种是4路组相联cache，因为它在可接受的代价和复杂度下有最好的性能。下面我们将考察每种类型的cache。

直接映像cache

我们在介绍cache设计时讨论过直接映像cache。现在我们再从重充线（而不是单个字数据）的角度来重新考察它。直接映像cache把主存储器划分为一个二维矩阵，包含K列，每列N个重充线。cache的宽度为一列，长度为N个重充线。第N行cache可以存放主存储器K列中任何一列的第N个重充线。标签地址保存了存储器列的地址。例如，假设有一个处理器，有32位的 378

可按字节寻址的空间，有一个256K的直接映像cache，cache用64字节长的重充线来重新载入。这样一个系统会是什么样子呢：

1. 根据重充线对cache和主存储器重新进行划分。

a. 主存储器包含2^{32}字节/2^6字节每重充线 = 2^{26}个重充线

b. cache存储器包含2^{18}字节/2^6字节每重充线 = 2^{12}个重充线

2. 将cache存储器表示成包含2^{12}行的单个列，主存储器表示成2^{12}行、$2^{26}/2^{12}=2^{14}$列的XY矩阵。如图14-5所示。

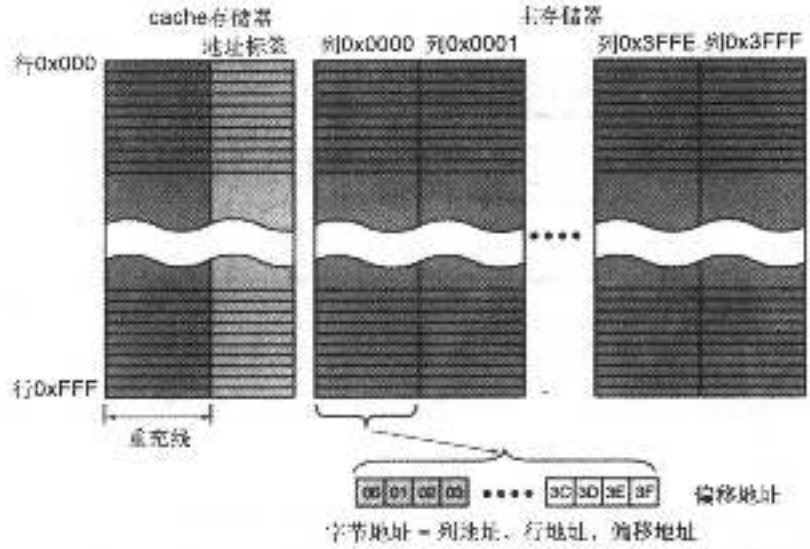


图14-5 一个256KB的直接映像cache示例。其中主存储器大小4G，重充线宽度64B

在图14-5中，我们把主存储器分为3个不同的区域：

- 在重充线中的偏移地址；
- 在一列中的行地址；
- 列地址。

我们把主存储器中一个重充线的字节位置映像到了cache中对应重充线的对应字节位置。换句话说，一个字节在主存储器中的偏移地址和在cache中的是相同的。而且，cache的每一行也对应于主存储器中的每一行。主存储器中每列的一行（重充线）都映像到了cache中同样行号的行，也就是重充线，而列地址则存放在cache中的标签RAM中。

地址标签域必须能够存放14位宽的列地址，对应于从0x0000到0x3FFF的列地址。主存储器和cache都有4096行，对应于从0x000到0xFFF的行地址。

下面这个例子中，我们把任意一个字节地址映像为列号/行号/偏移地址的模式。

字节地址 = 0xA7D304BE

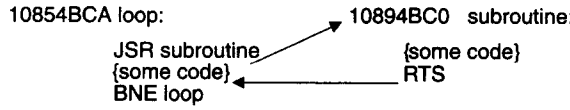
由于列号、行号和偏移地址不一定都在十六进制数字边界上（可被4除），所以使用二进制表示比用十六进制更方便。我们先把字节地址0xA7D304BE表示成一个32位的二进制数，然后再按直接映像cache的列号、行号和偏移的组织方式进行分组。

379

```
1010 0111 1101 0011 0000 0100 1011 1110  *8个十六进制数字
偏移：11 1110 = 0x3E
行号：1100 0001 0010 = 0xC12
列号：10 1001 1111 0100 = 0x29F4
```

这样，当我们将存储器重映像为具有64字节宽重充线的XY矩阵时，主存储器中0xA7D304BE的字节地址将被重新表示为0x29F4、0xC12、0x3E。如果这个字节被映像到cache中，那么cache中包含这个字节的重充线行号为0xC12，标签地址的值为0x29F4，字节所在的位置为距离重充线第一个字节0x3E处。

直接映像cache设计时相对容易实现，但是性能却相当受限制，因为在任意时刻，主存储器中每一行的所有重充线中最多只能有一个在cache中。下面这个例子显示了这个限制是如何影响到处理器性能的。



有两个比较类似的地址，一个是循环的入口，另一个是循环中调用的子程序入口。如果把它们按照上面的方法分解，可以得到循环的地址是：

- 偏移 = 0x0A
- 行号 = 0x52F
- 列号 = 0x0421

子程序的地址是：

- 偏移 = 0x00
- 行号 = 0x52F
- 列号 = 0x0422

因此，这种比较特殊的情况是可能发生的，或者是由于汇编语言算法造成的，或者是编译器和连接器共同组织目标代码映像的结果。总之，循环的入口及其调用的子程序入口刚好分别处于相邻两列的同一行。

每次子程序被调用时，cache控制器必须在执行子程序之前重新载入第0x422列的第0x52F行。同样，在遇到RTS指令时，cache中的那一行从相邻的那一列重新载入。我们在前面讨论过如何计算有效执行时间，可以看出这种情况下代码的执行速度很可能会比两个代码片段不在同一行的情况下慢上10倍。

出现这个问题主要就是因为直接映像cache的限制。由于对任意一个给定行的重充线只能映像到cache中的一个位置，所以当我们需要访问另一列中同一行的重充线时，我们没有选择只能替换了。

在灵活性方面与这种方法相对照的是相联cache（associative cache）。接下来我们来讨论这个方式。 380

相联cache

正如我们前面已经讨论过的，直接映像cache是相当受限的，因为来自主存储器的重充线映像到cache的位置受到严格限制。如果主存储器中映像到cache中某个地址的两条重充线都经常被访问，那么计算机将要花费大量的时间将两条重充线换入换出cache。试想如果能把主存储器中的某个重充线映像到cache中任意一个可用的重充线位置，那将会极大地提高性能。我们把这种组织方式称为相联cache。图14-6是相联cache的一个示意图。

在这个例子中，存储空间大小为1MB，相关的cache为4KB，重充线大小为64B。cache包

含64个重充线，主存储器被组织成单列的2^{14}个重充线（16KB）。

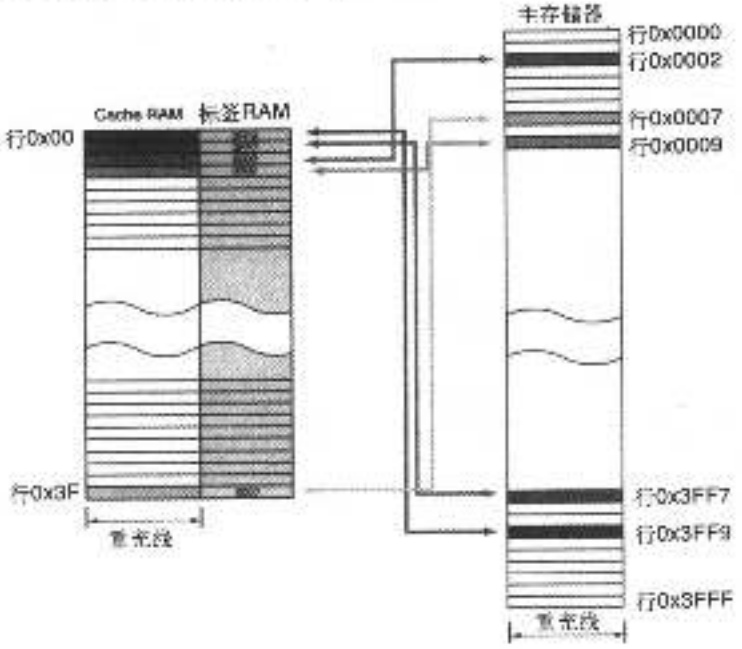


图14-6 一个4KB的相联cache示例，其中主存储器大小为1M，重充线大小为64字节[注：DVD-ROM光盘中有彩色图]

这个例子代表的是全相联cache（fully associative cache）。主存储器中的重充线可以映像到cache中任意一个可用的重充线位置，这样做非常好，不再存在直接映像cache的限制了，图14-6试图用多种颜色来显示cache中的行到主存储器的行的近乎随机的映像。然而，相联cache的复杂性随着cache大小和主存储器大小的增长成指数级增长。考虑两个问题：

1. 当cache中所有可用的行中都存放了主存储器中的有效行时，cache控制硬件如何决定从主存储器中读入的下一个重充线存放在cache的什么位置呢？

2. 由于主存储器中的任何重充线可以映像到cache的任意重充线位置，cache的控制硬件如何确定主存储器中的某个重充线目前是否在cache中呢？

对于第1个问题，我们可以在cache每一行的后面设置一个二进制计数器，每经过一个时钟周期，所有的计数器都加一次。每当我们访问cache中某行的数据时，都将该行的计数器复位为0。当计数器达到最大值时，就不再累加，值保持不变，并不返回到0。

381 所有的计数器值将被输入到一个优先级电路，该电路的输出是计数最高的计数器所在行的地址，这个行地址就是下一个读入cache的数据存放的位置。换句话说，我们用硬件实现了最近最少使用（least recently used, LRU）算法。

至于第2个问题，这里引入一种新的存储器设计，称为可按内容寻址存储器（content addressable memory, CAM）。CAM可被看作是反向意义的标准存储器。在CAM中，输入的是数据，输出的是CAM中存放该数据的地址。CAM的每一个存储单元还包含一个数据比较电路。当cache的控制单元把一个标签地址送入CAM时，所有的比较器都并行地进行内容搜索。如果输入的标签地址与某个cache标签地址位置所存储的地址相符，那么电路将给出地址匹配（命中）的信息，并输出该主存储器标签地址的cache行地址。

随着cache和主存储器空间的增长，cache控制硬件要处理的信息规模和复杂性也相应迅速

增加。因此，在实际运用的cache体系中，全相联cache并不是一个在经济上可行的解决方案。

组相联cache

实际地讲，在灵活性和性能之间最好的折中是组相联cache（set-associative cache）设计。组相联cache综合了直接映像和相联cache的特点。事实上，在现代处理器中，四路组相联cache是最常用的设计。它等于是在多列上的直接映像。例如，两路组相联cache有两个直接映像的列，每列能够存放主存储器中相应行的任何重充线。

图14-7显示了一个两路组相联cache的设计。

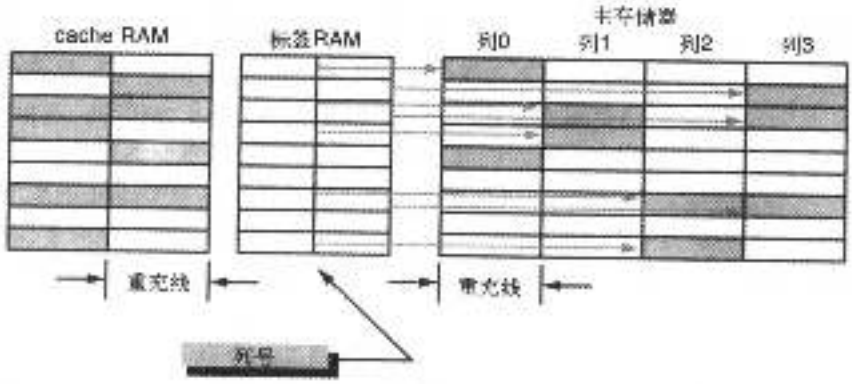


图14-7 两路组相联cache设计。由Baron和Higbie[1]提供[注：彩色图在DVD光盘中]

在一行内，主存储器中的任意两个重充线可根据标签RAM映像到cache中的两个重充线位置之一。一个一路的组相联cache就退化成了一个直接映像cache。

四路的组相联cache已经成为现代微处理器事实上的标准cache设计。大多数微处理器都采用了这个设计（或者这个设计的变型）作为片上的指令cache和数据cache。 382

图14-8描述了一个4路的组相联cache设计，下面是一些参数：

- 主存储器大小为4G字节
- cache存储器大小为1M字节
- 重充线大小为64字节

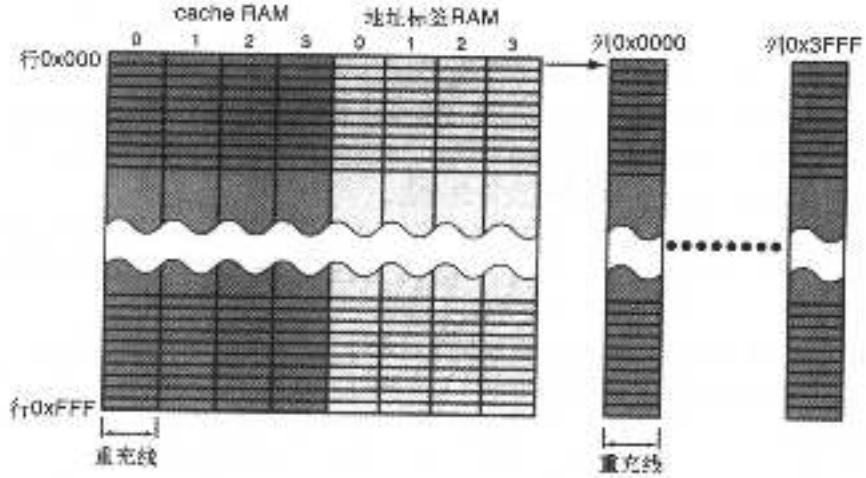


图14-8 具有32位存储空间和1MBcache的4路组相联cache。重充线大小为64B

行地址为0x000到0xFFF（4096个行地址），列地址为0x0000到0x3FFF（16 384个列地址）。如果把直接映像cache与一个同样大小的4路组相联cache做比较，我们会看到前者的列数只有后者的1/4，前者行数为后者的4倍。这就是把相同数量的重充线从单列重新分配成为4×N矩阵的结果。

这意味着在这个4路组相联cache设计中，cache每一行能映像的主存储器重充线的数量是直接映像cache的4倍。乍一看，这似乎对性能没有太大的提高。然而，关键还在于4路设计关联性。即使有4倍的列数，而且行在cache中可映像到4个可能位置，但在新的重充线可以放到这4个位置中的哪一个方面，我们还有很多灵活的策略。譬如，我们可以在这4个可能cache位置处应用一个简单的LRU算法，这就避免了直接映像cache中产生的抖动现象。

你可以容易地看出，4路组相联cache比直接映像cache具有额外的复杂性。我们需要与全相联cache相类似的附加电路，用来决定cache替换策略和检测地址标签命中情况。但是，与全相联相比4路组相联的设计实现起来简单很多。记住直接映像cache就是一个1路组相联cache。

最后我们要讨论的cache设计叫作区映像cache（sector-mapped cache），这种方式是对相联映像的一种改进。主存储器和重充线被分组成多个区（行）。主存储器中的一个区能够映像到cache中的一个区，而cache采用一个相联存储器来进行映像。标签RAM中的地址是区地址。
383 区映像引入的一个额外的复杂性是标签RAM需要有效位（validity bit），该有效位跟踪主存储器中目前在cache中的重充线。图14-9a图示了区映像cache设计。

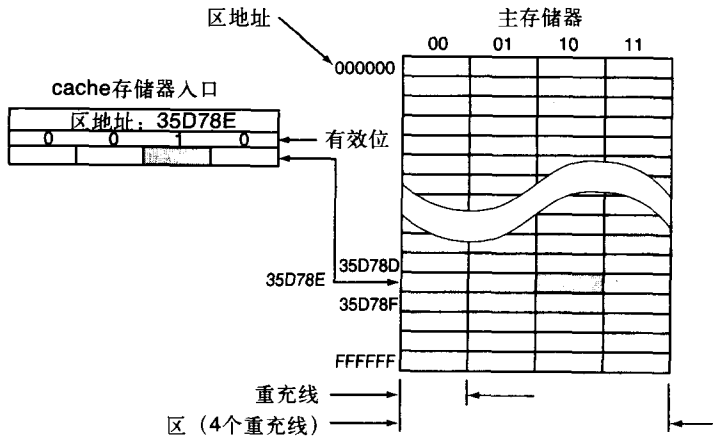


图14-9a 一个32位地址存储系统的区映像cache的示意图。其中，每个区有4个重充线，每个重充线大小为64字节。当前只有区地址为0x35D78E的10位置处的重充线是有效的

在这个例子中，我们将一个有32位地址范围的存储系统映像到了一个任意大小的cache中，每个区包含4个重充线，每个重充线包含64个字节。在这个特殊的例子中，第0x35D78E区中的第10号重充线是有效的，所以相应的有效位设置如图。

为什么需要有效位可能还不明确，这个简单的例子应该有助于说明这一点。记住，我们是通过区地址来将主存储器映射到cache的，一个区中的重充线在主存储器和cache中保持相同的相对区地址，而我们每次只是访问一个区中的一条重充线。

由于相对于主存储器的区，cache是全相联的，所以我们可采用某种LRU算法来决定cache中的哪一条重充线可以被替换。一个新区中的重充线第一次映像到cache中时，区地址会被更新，只有导致cache入口被更新的重充线是有效的，剩下的3个重充线00、01和11对应于以前的区，不对应于主存储器在这个新区地址的重充线。所以，我们需要有效位。

通过把重充线按行分组，我们降低了纯粹的相联cache设计的复杂性。在这个例子中，我们

将问题简化了4倍。在一个区内，每个主存储器中的重充线都必须映像到cache中的对应位置。然而，cache的相联特性还引入了另一层的复杂性。当我们从区中装载一个重充线到cache中时，标签RAM的地址必须对应于刚加入重充线的区地址。其他重充线可能还存放有别的区的数据。因此，我们需要一个有效位来指明在一个cache区内哪些重充线对应于主存储器中的正确的重充线。

图14-10显示了不中率（miss rate）在不同cache大小时随cache相联性的变化情况。

显然相联cache的加入极大地提高了cache的命中率。而且，随着cache容量增加，命中率受相联度的影响降低，因此当cache从4路改到8路时不中率并没有明显的改善。

图14-11显示了当cache和重充线的大小增加时，不中率降低的情况。注意，这些曲线是渐近线，而且当重充线大小超过64字节时，不中率几乎就没什么改善了。而且，当cache大小达到64KB时，系统的性能也开始急剧下降。当然，我们现在知道这是局部性的一种表现，只要给我们一些好的数据，我们就能知道什么样的cache最适合我们的需求。 384

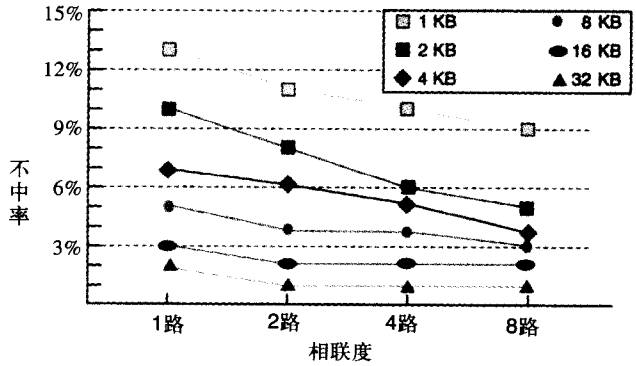


图14-10 cache相联度随不中率的变化情况，由Patterson和Hennessy[5]提供

下面我们更定量地考察一下性能。下面两个公式给出了一个简化的性能模型：

1. 执行时间 = （执行周期+延迟周期）×（周期时间）
2. 延迟周期 = （指令数）×（不中率）×（不中惩罚）

执行时间，或者说运行一个算法所需要的时间取决于两个因素。首先，算法中实际有多少条指令（执行周期），以及有多少个周期花费到了填充cache上（延迟周期）。要记得处理器总是从cache中得到数据执行，所以如果数据没有在cache中，处理器就必须等到cache被重新填充后才能继续执行。

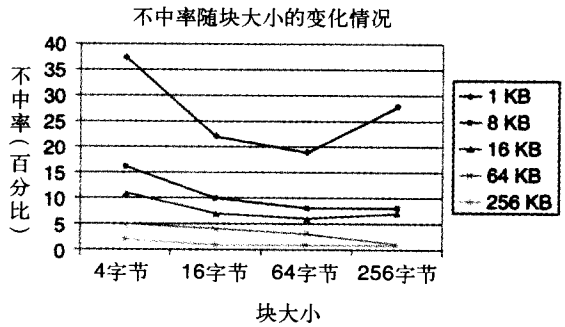


图14-11 不中率随块大小的变化情况，由Agarwal[2]提供

延迟周期是cache命中率的函数，所以它取决于要执行的指令总数。有些指令没有被cache

命中，所以对于每次未命中，都会招致一个不中惩罚。惩罚就是在程序能继续执行前，重填充cache所需的时间。

由此我们可以看到两种改善性能的策略：

1. 降低不中率

2. 降低不中惩罚

如果我们增加块的大小会怎样呢？如图14-11所示，随着趋进64字节，性能会有些改善，但是重充线越大，不中惩罚就越大，因为每次不中时就要重填充更多的数据，所以性能就可能变坏而不是更好。

从图14-10中我们也可以看出为什么四路组相联cache会如此流行。要注意的是，只要cache本身变得足够大，对于不同类型的组相联cache，不中率就没有显著变化。

下面继续我们关于如何提高整个cache性能的讨论，我们将考虑一些目前还没有考虑到的
385 因素。在给定不中率的情况下，我们可以通过降低不中惩罚来提高总体性能。因此，如果不中发生在主cache，我们可以加入一个2级cache来降低不中惩罚。

通常，主cache（L1）与处理器在同一个芯片上。我们可以在主存储器（DRAM）以上用速度非常快的SRAM加入另一级cache（L2）。采用这种方式，当数据在二级cache中时，不中惩罚就会降低。例如，假设有一个处理器，每个时钟周期执行1条指令（周期每指令CPI＝1.0），时钟频率为500MHz，不中率为5%，DRAM的访问时间为200ns。

通过加入一个访问时间为20ns的L2 cache，我们能将总的不中率降低到2%。因此，通过采用多级cache策略，我们可以尽量提高1级cache的命中率，降低2级cache的不中率。

cache写策略

只要你把指令和数据从存储器中读出并将它们映像到cache以获得更好的性能，这个cache操作就会相对简单。但新产生的数据必须要写入存储器时，复杂度就会急剧增加。如果数据存回cache，那么cache中的数据和存储器中的对应数据就不相同（一致）了。这是一个严重的问题，可能会在某些情况下导致系统崩溃，因此值得关注。一般而言，从数据写入的角度来看，cache的活动可以分为两类：

通写cache（write-through cache）：数据写入cache后立刻写入到主存储器。通写cache在向外存储器写数据时的性能不错，但该策略要求cache与主存储器中的数据必须始终一致。

回写cache（write-back cache）：数据被暂时保存，直到总线空闲不会影响其他的操作时，才允许写数据。实际上回写过程也可以等到整个块的数据都要写时才开始进行。我们把数据的回写也叫做*后写*（post-write）。此外，我们需要记录哪些cache单元包含不一致的数据。如果某个存储单元还其更新的数据还在cache中，就称之为*脏单元*（dirty cell）。对于采用回写策略的cache，其标签RAM必须包含有效位来跟踪脏单元。

如果数据的映像不在cache中，就不会有什么问题，因为数据可以直接写入到外存储器，就好像没有cache一样。这称为*绕写cache*（write-around cache），因为不在cache中的数据被直接写到了存储器。另外一种方式是，如果有可用的cache块，而且没有对应的脏存储单元，则cache可先把数据存入cache，然后再根据cache的设计策略进行通写或者后写。

我们来总结一下对cache的讨论：

- 有两种类型的两种局部性：空间和时间。
- cache中的内容包括数据、标签和有效位。
- 空间局部性要求更大的块容量。
- 因为处理器速度变得比存储器越来越快，不中惩罚也不断增加，所以现代处理器都采用

组相联cache。

- 我们采用了指令与数据分开存储的cache。

为了避免出现冯·诺依曼瓶颈：

- 多级cache被用于降低不中断的（假设L1 cache在片上）；
- 存储系统设计成支持突发模式访问的cache。

386

14.2 虚拟存储器

还记得存储器的层次吗？在存储器层次中，一旦到了主存储器以下，就需要用虚拟存储器提供的方法来将较低层次存储器映像到较高层次存储器。我们使用大量的慢速存储器（硬盘、磁带机、Internet）为实际的主存储器提供指令和数据。虚拟存储管理器（通常就是操作系统）就能使程序似乎有无限的存储资源来运行和存储数据。

为什么Win 9X、Win 2K和Win XP推荐使用至少64~128MB的主存储器呢？因为如果计算机的物理存储资源太少，PC就会花时间不停地在内存和硬盘之间来回交换数据。之前在关于cache的讨论中，我们考虑了未命中所招致的惩罚，就是还要去访问主存储器。硬盘的平均访问时间大约为1ms。无cache的存储器的平均访问时间约为10ns。因此当数据不在RAM中，需要从硬盘中读取时，访问时间大约要慢$10^{-3}/10^{-8}$=10 000倍。

有了虚拟存储器，主存储器的角色就是作为二级存储器（一般就是硬盘）的cache。这一技术的好处是让每个程序都感到好像有无限大的物理存储量。在这种情况下，CPU发出的都是虚拟地址（virtual address）。它不知道这个虚拟地址在物理存储器中是否真的存在，或者是否实际地被映像到了硬盘，也不知道操作系统中是否还有另外一个程序目前已经占用了这个虚拟地址所指向的物理存储空间。

有了基于虚拟存储器模型的系统，程序的重定位就成了建立可执行目标代码的简单自然的方法，系统就能够给每个程序分配属性（通常是以保护方案的形式）使得两个都在运行的程序保持相互隔离。过去有一家叫做数据设备公司（DEC）的企业提出了虚拟存储器的思想，并以此为基础建立了公司。DEC VAX这个术语就是这个公司的同义词，VAX代表虚拟地址扩展（virtual address extension），即虚拟存储器。

图14-12是虚拟存储器模型的一个简化示意图。随着处理器变得越来越复杂，我们需要将CPU与其他硬件如存储管理分开来设计。这样，CPU只需给数据或指令发出地址就行了。

387

由于CPU并不知道在某一时刻程序代码或数据具体在什么地方，它给出的地址就是虚拟地址（virtual address）。实际存放的位置可能是指令cache、数据cache、主存储器或者二级存储器。CPU的功能就是向系统的其他部分提供所需要的虚拟地址。

如果数据在L1 cache中，CPU可以直接检索到它。如果未能命中，则CPU被阻塞，系统从主存储器中将重

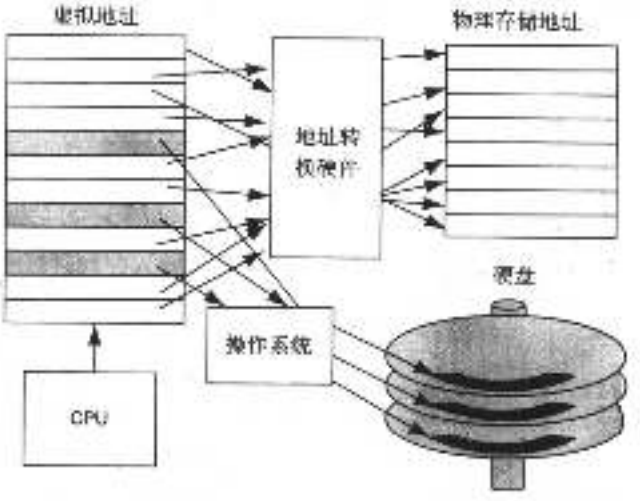


图14-12 一个虚拟存储器模型。由CPU给出一个虚拟地址，相应的硬件和操作系统将这个虚拟地址映像到物理存储器或者二级存储器（硬盘）上

充线读取到cache。如果数据也不在主存储器中，CPU就需要继续等待，直到数据从硬盘中被读入。然而，你将会看到，如果数据不在物理存储器中，指令将被中止，操作系统必须接管控制从磁盘取得虚拟地址。

由于虚拟存储器与操作系统密切相关，所以我们用更一般的术语来描述CPU的可用存储空间，叫作*逻辑存储器*（logical memory）。我们以非常紧密关联的方式使用逻辑存储器、物理存储器和虚拟存储器这三个术语，但它们之间存在真实的差别，其中操作系统管理下的硬盘也称为虚拟存储器。我们来仔细地研究一下虚拟存储系统的组成部分，请看图14-13。

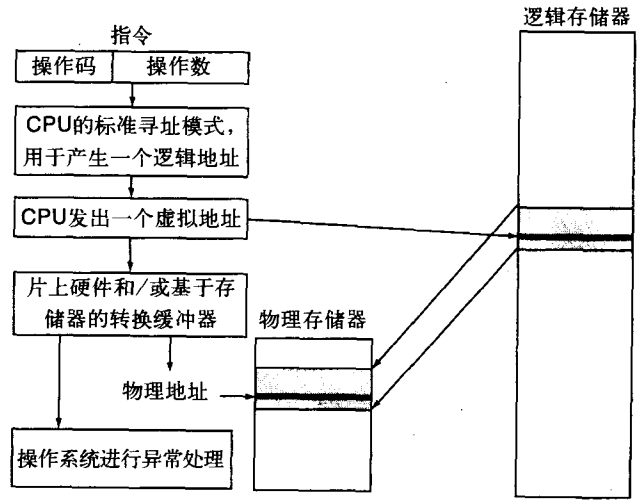


图14-13 一个虚拟存储系统的组成部分

CPU执行一条指令时会取出存储器中的操作数，程序计数器也会给出下一条指令的执行地址。不管什么情况下，CPU都要通过标准的寻址方式（寻址模式）产生一个地址。这个地址指向处理器的*逻辑存储空间*中的一个位置。处理器中还有称为*存储管理单元*（memory management unit，MMU）的专用硬件，位于CPU之外，它将逻辑地址映射到计算机存储空间的物理地址。如果指令发出时，给出的地址在物理存储器中，物理存储器就满足这个地址要求。但是，不管在什么情况下，采用的都是突发访问模式来重填充cache，而不是仅仅取单个字。

然而，有时候存储访问要求不能被满足，因为所需要的数据在硬盘的虚拟存储器中。如果处理器中的存储管理硬件检测到该地址不在物理存储器中，就产生一个异常。这个异常类似于一个内部产生的中断，不同之处在于一个真正的中断在中断被接受之前容许当前指令继续执行完毕。而当一个异常发生时，指令必须马上中止，因为操作系统必须接管并处理这个异常请求。

当讨论片上cache时我们知道，在主存储器和cache之间传输数据时，是以块为单位，块的大小为一个重充线时效率最高，一般是64个字节。现在我们讨论虚拟存储器，数据可能保存在硬盘上，这时我们不得不增加块的大小，因为现在的不中惩罚要比主存储器的不中惩罚要多上几千倍。

388

我们知道，硬盘是一个带有移动和旋转单元的机械设备。最快的硬盘转速为10 000PRM，大约为每秒钟167转。如果存在硬盘上磁道上的数据刚好错过了硬盘的读写头，我们就不得不再等上60ms数据再转回来。此外，读写头在磁道间移动还需要大约1ms的时间。因此，如果实在必须要读取虚拟存储器中的数据，我们就应该读入足够的数据，这样才值得我们花费那么长的等待时间。

如果你对硬盘进行过*碎片整理*，你就会知道它对性能的提高有多大。当你不停地在硬盘上添加和删除数据之后，硬盘（扇区）上可用于存放程序和数据文件的整块连续空间会变得越来越少。这就意味着如果你必须要从硬盘上检索存放在4个扇区中的数据（大约是2000字节），这4个扇区可能分布在硬盘的各个地方。让我们做一个简单的计算看看有什么发现。

假设硬盘的读写头从一个磁道移动到下一个磁道需要的时间是5ms。统计表明，在一个碎片相当多的磁盘上，两个扇区之间的平均距离为10条磁道。这是一个平均值，有时候要访问的数据刚好就在相邻的柱面上，有时候却不是这样的。记住柱面就是我们正在访问的磁道，但考虑到硬盘可能会有多个盘片，多个读写头，因此一个柱面可能包含多个磁道。

最后，假设每次都必须要移动到一个新的柱面，在数据到达读写头下之前磁盘平均旋转1/2圈（旋转延迟）。

在这样的情况下，我们将需要等待：

1. 5ms/磁道 × 10个磁道（每次访问磁道）+ 1/2 ×（1/167）= 53ms

2. 4（访问次数）× 53ms（每次访问时间）= 212ms

现在，假设数据位于硬盘上连续的4个扇区内。换句话说，硬盘已经进行过碎片整理。那么访问第一扇区时，读写头移动的时间和旋转的时间都没有变化，但是接着访问剩下的扇区速度就快多了。若硬盘的每个磁道有64个扇区，转速为10 000rpm，那么读取4个扇区花费的时间就是（4/64）×（1/167）=374μs。

因此，读取数据的时间就是50ms + 3ms + 374μs = 53.00374ms，省下了超过150ms的时间。局部性原则告诉我们这样做是非常明智的，因为我们很可能马上就需要所有的这2000字节的数据。

14.3 页

在一个虚拟存储系统中，我们将物理存储器和二级存储器（硬盘、磁带机、快闪卡）之间作为一个单位进行映像的存储块称作页（page）。映像时虚拟存储系统只把程序中的如下一些部分读入物理存储器：

- 能够被存入物理存储器的
- 已经被操作系统分配的
- 目前正在被使用的

这里我们又看到了分页的运用。然而，分页虽然是物理存储器中二进制地址的属性，但 389
它对于虚拟存储系统非常重要，因为通常硬盘上的一个扇区刚好自然地映像到虚拟存储器上的一个页。我们可以把逻辑地址分为两个部分：

1. 虚拟页编号（页号）

2. 页内的字偏移（或者字节偏移）

回忆一下物理存储器中分页的例子。假设有一个16位的地址$F79D，那么我们可以把它表示成：

- 页号 = F7
- 字节偏移 = 9D

这个例子中我们有256个页，每页有256个字节。不同的操作系统采用页的大小不尽相同。当需要的数据不在存储器中时，产生一个页故障（page fault）的异常。操作系统（O/S）必须接管并从硬盘上检索到需要的页。正如我们前面讨论的，页故障意味着巨大的不中惩罚。因此，我们希望通过将页设计得相当大（譬如1到8KB）以降低这个损失。

操作系统（O/S）通过MMU对页进行管理。采用的策略是将页故障作为最高优先级。一次页故障及其后续的从硬盘到主存储器的数据重载入可能要花费几十万个时钟周期，所以，花代价设计一个LRU算法是值得的。在操作系统接管后，页故障可以用软件而不是硬件来处理。软件处理的速度显然要比硬件慢很多，但是灵活性要好得多。而且，虚拟存储系统采用回写策略，因为通写策略的代价太高。你想过为什么要通过关闭操作系统来关闭计算机吗？其原因就在于操作系统在采用回写策略来管理虚拟存储器。如果你在操作系统关闭打开的文件之前就直接关闭计算机，你就有可能丢失物理页所存储的数据，因为这些页有可能还没有回写到虚拟页上，就是硬盘上。

图14-14是一个分页系统的简化示意图。由于我们可以简单地将32位的地址分为页段和偏移段，所以接着我们可以把页段映像到物理存储器或者虚拟存储器（硬盘）。由于硬盘是按照读写头、磁道和扇区模型来存储数据的，所以我们需要操作系统把硬盘映像到虚拟存储器中。

我们将这种系统称为按需调页虚拟存储器（demand-paged virtual memory），因为操作系统根据需要把相应的页载入到了物理存储器中。

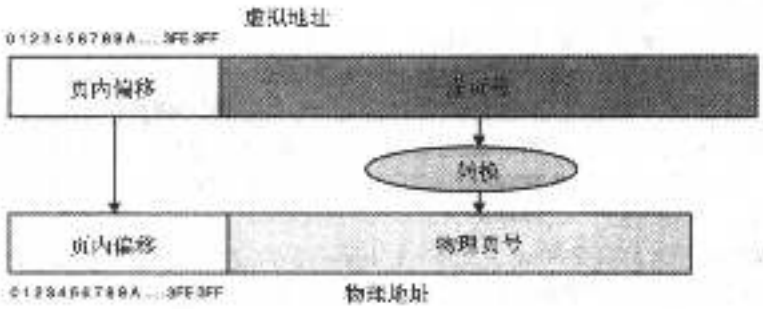


图14-14 分页系统示意图，整个虚拟地址被分为页号和页内偏移两部分。虚拟页号被映像到物理存储器和虚拟存储器上

由于页故障的代价太大，所以我们可以每次传输和存储多个页。我们将这些物理存储块称为页帧（page frame），它们承载了从硬盘传输来的所有页，其大小可在1 024到8 192个字节之间。最后，正如cache需要一个标签存储器来保存cache与主存储器之间的映像信息一样，操作系统也维护着一个页映像（page map），用来保存虚拟存储器与物理存储器之间的映像信息。

390

此外，页映像中还可能包含其他一些与当前正在运行的操作系统和应用程序相关的信息。页表（page table）就是页映像的一部分，它为操作系统所拥有，用来记录当前正在使用的页，以及已经被映像到物理存储器和虚拟存储器上的页。

一些页表的表项总结如下：

- 虚拟页号：在页表中的偏移位置
- 有效位：页当前是否在存储器中
- 脏位：程序是否修改（写入）了页
- 保护位：哪些用户（进程、程序）可以访问该页
- 页帧号：页在物理存储器时的页帧地址

14.4 转换旁路缓冲器（TLB）

大多数计算机系统都把页表保存在主存储器中。但处理器还可以包含一个专用的寄存器，就是页表基址寄存器（page-table base register），它指向页表的起始位置。只有操作系统使用管理模式的指令才能修改页表基址寄存器。从理论上讲，对主存储器（无cache）的访问需要两次，因为每次访问主存储器时，必须首先访问页表。

因此，现代处理器还维护了一个转换旁路缓冲器（translation look-aside buffer,TLB）作为页映像的一部分。TLB是一个片上cache，通常是全相联的，用于进行虚拟存储器管理。TLB中保存有与页表的一部分相同的信息，把虚拟页号映像到了页帧号。TLB中只保存最近访问的页的信息，最近访问最少（LRU）的表项将被从TLB中删除。而且TLB中只保存有效页的映像（不保存脏页）。

图14-15显示了分页系统的各个组成部分，包括片上硬件、TLB以及页表基址寄存器所指向的主存储器中的页表。CPU产生一个虚拟地址，系统首先在片上cache进行检验和匹配。如

391

果cache未命中，那么该指令的流水线将被阻断（我们假定这是一个超标量的处理器，其他的指令流水线仍然在处理指令），cache的控制逻辑就通知TLB看一看该页是否在物理存储器中。

如果在TLB中，就由总线逻辑从物理存储器中检索该重充线，并将其放入到cache，然后就可继续进行处理过程。

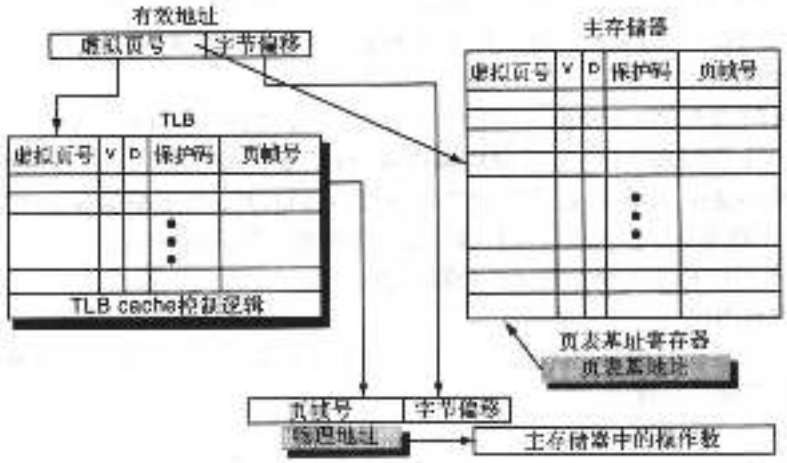


图14-15 分页系统的组成部分（由Baron和Higbie[3]提供）

如果页不在TLB的列表中，那么处理器就利用页表基址寄存器建立一个指针，并开始在页表中搜索所需要的页，看该页是否在物理存储器中。如果在，那么TLB就被更新，重充线被检索到后读入cache。如果该页在虚拟存储器中，就必须产生页故障，操作系统就介入并从硬盘中检索需要的页，并把它放入页帧。图14-16显示了虚拟分页处理的流程图。

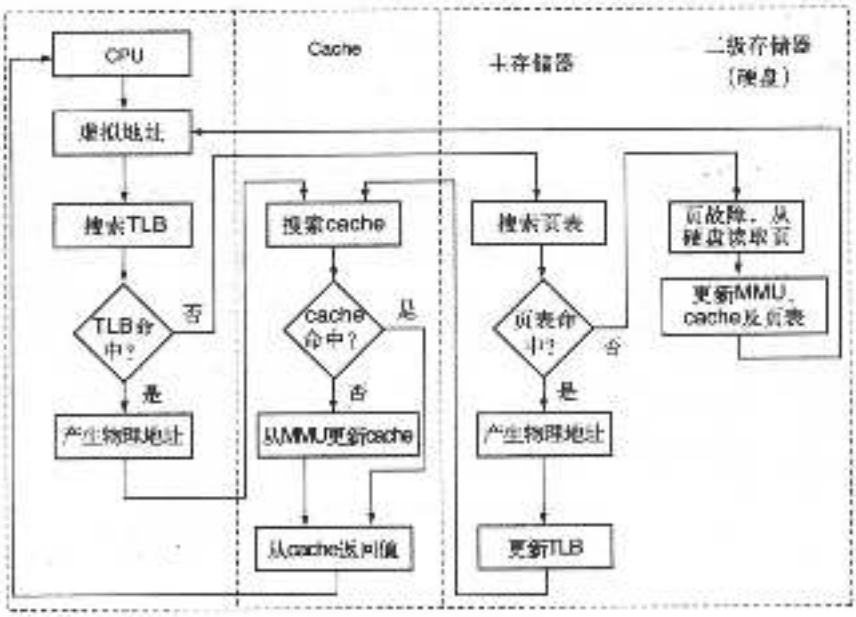


图14-16 虚拟页访问过程的流程图（由Heuring和Jordan[4]提供）

14.5 保护

由于虚拟存储器的有效管理依赖于专门的硬件来支持分页处理，所以需要讨论一下这个

硬件必须提供的另一个重要功能——保护不同的内存区域不被有意或者无意地互相访问。

在68K中，我们已经见过一个简单的保护方案，就是采用管理模式和用户模式。当处理器处于用户模式时，指令集只有一部分可以被执行。如果某个程序试图执行一个管理模式的指令，就会产生异常。一般来说，这个异常向量就是指向操作系统的一个入口，操作系统将来处理这个非法的指令。

此外，我们还看到68K支持两个独立的栈指针，一个用户栈指针和一个管理栈指针。然而，
392 68K中并没有一个机制来防止一个存储区域被另一个存储区域非法访问。为做到这一点，我们需要MMU的保护硬件。MMU采用了与CPU独立的不同芯片，但是随着集成度的提高，MMU电路就被合并在处理器中，大多数现代高性能处理器都有片上的MMU。

因此，我们可以把基于硬件的保护功能小结如下：

- 用户模式和内核（管理）模式
- 限制用户只能读取用户/内核模式位，提供从用户模式到内核模式的切换机制，一般是通过操作系统的系统调用
- TLB
- 页表基地址指针
- MMU

最后，我们看一下两个现代的处理器，比较一下它们对MMU、TLB和cache的支持，如下表[5]所示。这里我们不打算全面研究这个表格，而只是讨论其中最关键的一个信息。这个表格是现今最先进、最强大的两种商业微处理器的数据总结。现在这个表格中比较两个处理器的数据你应该都能看明白了。

特性	Intel Pentium Pro	Freescale PowerPC 604
虚拟地址范围	32位	52位
物理地址范围	32位	32位
页大小	4KB，4MB	4KB，选择表，256MB
TLB组织	指令和数据的TLB分开。两者都是4路组相联的，采用了伪LRU替换策略。指令TLB：32个表项；数据TLB：64个表项。TLB未命中由硬件处理	指令和数据的TLB分开。两者都是2路组相联的，采用了LRU替换策略。指令TLB：128个表项；数据TLB：128个表项。TLB未命中由硬件处理
cache组织	指令cache和数据cache分开	指令cache和数据cache分开
cache大小	每个8KB	每个16KB
cache相联方式	4路组相联	4路组相联
替换策略	近似LRU替换策略	LRU替换策略
块大小	32字节	32字节
写策略	回写	回写或者通写

总结

这一章介绍的内容有：

- 存储器层次的概念。
- 局部性的概念以及它如何证明cache的使用是正确的。
- cache的主要类型、直接映像、全相联、组相联和区映像。

- cache性能的度量。
- 虚拟存储器的体系结构。
- 虚拟存储器的不同组成部分以及操作系统是如何管理虚拟存储器资源的。 393

参考文献

[1] Robert J. Baron and Lee Higbie, *Computer Architecture*, ISBN 0-2015-0923-7, Addison-Wesley, Reading, MA, 1992, p. 201.

[2] A. Agarwal, *Analysis of Cache Performance for Operating Systems and Multiprogramming*, PhD Thesis, Stanford University, Tech. Report No. CSL-TR-87-332 (May 1987).

[3] Robert J. Baron and Lee Higbie, *op cit*, p. 207.

[4] Vincent P. Heuring and Harry F. Jordan, *Computer Systems Design and Architecture*, ISBN 0-8053-4330-X, Addison-Wesley Longman, Menlo Park, CA, 1997, p. 367.

[5] Patterson and Hennessy, *op cit*, p. 613. 394

[6] Daniel Mann, *op cit*.

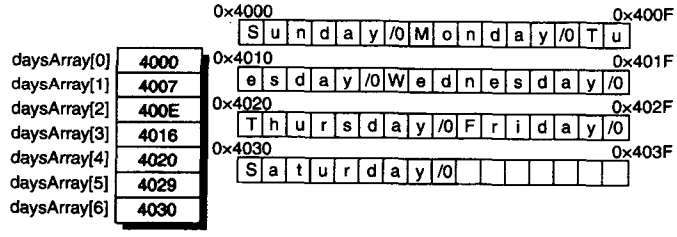
习题

1. a. “存储器的层次结构”是什么意思？描述一下它，并用一些典型的值来说明。
   b. 什么是空间局部性和时间局部性？局部性对于片上cache的意义在于什么？
   c. 为什么片上cache一次从存储器读取整个“重充线”，而不是仅仅读取需要的指令或数据？
   d. 比较一下“回写cache”和“通写cache”的结构。
2. 与主存储器相比，片上cache的容量非常小，然而对大多数程序而言cache的命中率一般都大于90%。解释一下为什么会是这样。
3. 假设下面的C++程序被编译后在68000体系结构的计算机上运行。指令代码在存储器中的位置是$00001000到$00001020。变量DataStream[10]被放在堆存储器中，地址范围是$00008000到$00008028。其他的所有变量都存储在从$FFFFFF00开始的栈空间中。请尽可能完全地解释下面的程序是如何表现出时间局部性和空间局部性的。

```
int main(void)
{
      int count ;
      const int maxcount = 10 ;
      int DataStream[10] ;
      for ( count = 0 ; count <= maxcount ; count++ )
        *(DataStream + count) = count*count ;
      return 0;
}
```

4. 考虑下面的C++变量声明和图示。其中的数字表示各种数组元素在存储器中的字节地址。简单地讨论一下，这个声明是否体现了局部性原则。

```
const  char *daysArray[7] = { "Sunday", "Monday", "Tuesday",
  "Wednesday", "Thursday", "Friday", "Saturday", "Sunday" };
```



395

5. 假设你有一个微处理器，具有20位的字节地址范围和16位宽的数据总线。处理器有一个片上cache，其特性如下：
   - cache为直接映像模式；
   - cache大小为4096字节，不包括标签存储器；
   - 一个重充线的大小为64字节。

   a. 主存储器中有多少个重充线？
   b. cache中有多少个重充线？
   c. 对于直接映像cache，主存储器应该划分为多少行和多少列？
   d. 为了对主存储器一行的所有重充线进行寻址，与cache相联的标签存储器中的地址长度需要多少位？不考虑脏数据位等其他标志位。
   e. 画出这个cache组织的草图。尽可能全面地显示存储器组织的地址部件。
6. 假设你有一个微处理器，具有1MB的寻址范围和4KB的直接映像cache。在cache中重充线的块大小为64字节。那么，
   1. cache中有多少个重充线？
   2. 为了与这个cache设计协调，主存储器应该如何组织（多少行，多少列）？
   3. 标签存储器所需位数是多少？
   4. 假设你试图要从地址$3FB0A处取一条指令，那么包含这个地址的重充线的行地址和列地址分别是多少？
7. 假设某个RISC处理器的片上cache的命中率为98%。指令在cache中时，需要一个时钟周期执行完毕。如果指令不在片上cache中，则导致处理器暂停程序的执行，并对cache的某部分进行重填充。这个过程需要100个时钟周期。如果该RISC处理器的时钟频率为100MHz，问有效执行时间是多少纳秒？
8. 假设某个32位RISC处理器的片上cache的命中率为98%。指令在cache中时，需要一个时钟周期执行完毕。如果指令不在片上cache中，则导致处理器暂停程序的执行，采用突发访问模式从主存储器中读取一条64字节的重充线，并将其写到cache。取外存储器中的数据每次需要2个时钟周期。如果处理器的时钟频率为100MHz，问有效执行时间是多少纳秒？
9. 考虑一下虚拟存储系统的各种组件，特别注意TLB的结构。主存储器中的页表被读入cache，
396 并且在发生虚拟页操作时都会对其进行例行更新，那么为什么每个TLB的入口都需要一个有效位呢？

第15章　计算机体系结构的性能问题

学习目标

- 硬件的各个方面如何能影响一个计算机系统的性能；
- 软/硬件划分决策是如何影响性能的；
- 增量式提高计算机系统性能的方法；
- 在测量计算机性能时对测试基准的利用和误用；
- 测量性能的方法；
- 一个项目计划如何能达到规定的性能目标。

15.1　引言

在一个课本的一个章节中要覆盖计算机性能的主题，篇幅是很不够的。然而，这在过去从来没能阻止我，所以在深入进去之前，让我们考虑一下这个问题及其范围。当我们谈到性能时，我确信你们中的很多人在假设我所指的是：最新和最好的PC在一系列接踵而来的称为*测试基准*的测试实例上工作得怎么样。对基准的测试可采取的形式包括：对重新计算一个大电子数据表所需的时间进行测量，或者对一个特定的图形密集的视频游戏在每秒种所播放的帧数进行测量。如果你选择PC的准则不是电子数据表操作或者视频游戏，则这两个性能测试基准的例子对你可能有效，也可能无效。

如果你的任务是设计软件（读这本书的大多数人将来会从事这种工作），那么你面临的将是性能问题的不同方面，你的问题将会变成："我怎样才能保证我编写的软件满足设计规范所提出的性能要求？"这样，你的问题就不是选择当天最热门的CPU，而是力图确保软件在设计时就能满足性能目标，即不是通过高代价的锤炼代码或重新设计来满足性能目标。

在本章中，我们将从三个角度来考察性能：

1. 硬件是如何直接影响性能的？
2. 你如何测量性能？
3. 你如何进行性能设计？

由于本课程是关于硬件的，所以我们先处理第1项。然后，我们再考虑测量计算机系统性能的方法。最后，我们从一个软件开发者的角度来看性能。

15.2　硬件和性能

同样的微处理器，既能驱动你家里的DVD播放机，也能运行当今市场上一些最热门的视频游戏，这看起来有点不太可能。为什么选择视频游戏？我们已经逐渐将视频游戏与高性能计算机联系在一起，视频游戏杂志总在比较这种计算机和那种计算机，或者这种视频卡和那种视频卡。当然，正因为我说过的原因，我们能运行某机器，但这不意味着你愿意用它玩视频游戏。为什么？是因为动作速度慢得令人难以接受。与你的2GHz Athlon或奔腾相比，帧速率可能远小于每秒1帧，所以屏幕更新很容易延迟。

是什么因素使得视频游戏在一种情形下慢得令人难以接受，而在另一种情形下却能使你

实时地玩游戏呢？为了回答这个问题，让我们实际地演习一下如何将一个典型的基于pc的视频游戏与一个8位计算机接口。

为进行实验，假设我们有一个具有10MHz时钟的8位处理器，能对1MB的存储器直接进行寻址，该处理器是一个单板计算机的组成部分，该单板计算机带有32MB的RAM，还有一个视频芯片用于驱动标准PC监视器（VGA），该监视器具有1024×768像素的分辨率和8位的颜色深度。视频存储器能够一次显示一个帧，并映射到一个3MB的存储块。特殊的硬件可以使存储器成为双工端口的（dual-ported）这样，视频芯片就能通过读存储器来显示当前帧，同时，处理器可通过向存储器块写数据来更新视频帧，并且从某存储器地址读数据。外部硬件被映射为I/O后，就成了基存储器指针，所以，通过向该硬件寄存器写数据就可以使处理器能在任何1MB边界上直接对存储器进行读写。最后，视频芯片仅支持从视频存储块到屏幕的映射过程。在视频芯片内部没有做图形处理。

我们首先面临的问题是8位微处理器不太可能与PC有相同的指令集体系结构（ISA）（除非我们选择一个4.77MHz的Intel 8088处理器），所以我们需要去访问视频游戏生产商，说服他们给我们游戏的C++源代码。当然，他们将立即同意我们的请求，使问题得以解决。下一步，我们需要用我们的ISA重写游戏的汇编语言程序。

所有的操作系统调用和图形例程也需要重写和重编译。由于我们只有1MB存储器模型，所以我们需要获得编译器和链接器映像的汇编语言输出，并将汇编语言指令插入到代码中，在1MB边界上形成一个桥梁。每次代码必须要穿越一个边界时，我们就插入I/O调用以重新映射基地址指针。

最后，我们需要写图形引擎仿真器驱动程序，使得视频正确地显示在屏幕上。假设我们完成了所有这些任务，现在你觉得困惑的是整个这套系统是多么愚笨，而你认为我们只要将游戏加载到板上让它跑就行了。实际上，它不会跑，它会非常地慢，慢到我们无法玩游戏。但是问题是，它还在运行。代码做了为它设计的工作，程序并没有崩溃，但是性能糟透了。

398 让我们将PC和8位单板计算机（SBC）的性能规范做一个比较。下面的表列出了两个系统的主要不同之处。我们可以看到，结果确实对8位SBC很不利。在初学者看来，PC的时钟速度是SBC时钟速度的200倍。不过，这里我们需要加一点小心，因为PC时钟速率是CPU的内部时钟速率。对于PC3200 DDR存储器，外部存储器的时钟速度的典型值为200MHz。

参　数	PC	SBC	注　释
时钟速度	2 000MHz	10MHz	
数据总线宽度	32位	8位	
可寻址存储器	4 096MB	32MB	
每条指令时钟数	<1	8~20	PC上的CPU有多级流水线
浮点计算	在硬件中	无	片上FPU
cache	片上cache和D-cache	无	
外部存储器	256MB	32MB	不是一个因素
图形加速	在视频卡上	无	

同样，如果PC外部存储器是SDRAM，那么，对每次大量的存储器载入就需要若干个时钟周期。然而，片上cache使外部存储器的载入保持到了最小。我们不能准确地说出cache的命中率，但90%大概是一个公平的猜测。

我们能假设较快的时钟就等同于更好的性能吗？在本例中，它确实是一个主要的因素，但我们能一般化地说较快的时钟就等同于更好的性能吗？在做比较的测试基准上，一个具有

3.2GHz时钟速度的Intel 奔腾4处理器与一个AMD 3000+ Athlon处理器有类似的性能结果，但Athlon的实际时钟频率大约是2.2GHz。

这就产生了对AMD的感知问题。对大多数PC购买者来说，时钟频率就等同于性能[1]。为保持竞争力，AMD被迫采用了一种虚拟时钟频率3000＋。它是虚拟的，因为它的意思是：AMD的产品性能比工作于3000MHz真实时钟频率的奔腾处理器性能要好一点（＋的意思）。

知道了关于流水线处理器的原理，我们就会猜测，与Intel奔腾体系结构相比较，AMD Athlon的体系结构一定是让处理器在每个流水线阶段做了“更多”的工作。然而，正因为它做得更多，它不得不运行得更慢。这样，两个部件的时钟频率就有了一个差别。

除了桌上计算环境，大多数计算机系统都力图运行得尽量慢，而不是尽量快。对于基于CMOS技术的处理器来说，速度等同于功率耗散。图2-15表明功率耗散随CMOS处理器时钟频率的增加而近乎线性增加。时钟频率的增加也带来了系统成本的相应增加。一个更快的处理器要求更快的支持芯片和更快的存储器。随着速度提高，功耗需求也会提高。更快的总线速度也意味着更有可能出现*无线电频率干涉*（radio frequency interference, RFI），就需要做更多的工作，将*电磁屏蔽*（electromagnetic shielding）结合进产品中来抑制RFI的发射。如果你曾经惊讶于为什么商业客机乘客舱的所有电子设备在10 000英尺高度以下必须关闭，这就是原因，航班和联邦航空局（FAA）担心在起飞和着陆过程中这些电子设备可能会与导航设备发生干涉，所以他们要求关闭所有电子设备（意味着所有带有时钟振荡器的东西）。

由于大多数嵌入式系统都是高度注重成本的，所以性能与CPU时钟速度几乎是呈逆向关 399
系的。一个嵌入式设计小组可能会问的问题是：“为了把工作做好，我们可以使处理器运行得多慢？”对于他们来说，重要的是性能的度量，而不是速度的度量。

甚至连Intel这个“时钟频率等于性能”团体的领导者也在寻找其他方式来解决其处理器在未来的性能问题。Rattner[2]描述了在Intel的一个“右倾”做法。公司的Centrino处理器采用了Banias CPU技术，该技术关注于在研制过程的每一步限制功率消耗，而不是提高时钟的速度。Intel的未来处理器将以多CPU核并行运转为特色，而不是继续提高时钟速度。

另一个浪费性能的地方归因于数据总线宽度。8位宽的数据总线只能处理范围在－128到127之间的整数。任何更小的负数或更大的正数将需要多次存储器访问。这样，数据总线越宽，系统访问存储器的效率就越高，CPU在内部移动和操作数据就越有效率。对于除了ASCII字符串操作的大多数处理算法，数据总线宽度加倍能将性能提高到2至4倍之间。

有更宽的总线就意味着在每次访问中可在CPU和存储器之间移动更多数据。虽然奔腾和Athlon处理器是32位机器，但它们的外部存储器总线宽是64位，这使之能达到每秒3.2GB的数据传输速率。我们如何达到这一点呢？采用PC3200 SDRAM存储器，我们能够在200MHz存储器时钟的每个相位都传输数据，这就达到了400MHz的数据传输速率。这样，每个存储器周期我们都能并行发送或接收两次8个字节，这等于$8 \times 2 \times 200$MHz，即3200MB/sec。而且，有直接访问4GB存储器的能力就意味着在每次穿过1MB存储器边界时，存储器访问将不需要分页指令的开销。

由于8位处理器没有片上cache，外部存储器是唯一的存储器，所以，所有的存储器访问都根据处理器的读取存储器模型来进行。我们可以假设对于每次存储器访问，CPU需要最少4个时钟周期。为简化，我们还假设，我们的32MB存储器是0时间等待状态的存储器，所以，所有的存储器访问就需要刚好4个时钟来完成一次读或写。这样，从存储器读数据将会以每秒2.5MB的速率进行，比PC慢1280倍。但等一下，事情越来越糟了。由于PC上的CPU具有片上指令cache和数据cache，所以对操作数和指令的访问将以处理器的全内部时钟速度（即2GHz）进行。

下面我们有一个有趣的差别需要考虑。PC的CPU包含一个片上浮点数学运算加速器和专

用浮点寄存器。8位处理器必须用算法仿真所有的浮点指令，这些指令对表示成浮点的数执行整数算术操作。我们在下一章很快就会看到，相比于通用编程方法，专用硬件能极大地加速算法的运行。

最后，在PC的视频卡上以图形加速器形式存在的片外支持硬件，使得图形密集的操作负担从CPU中解放出来。8位处理器必须自己计算所有的图形变换。

400

从这个讨论中我们知道，硬件体系结构对性能的影响不能达到最小化。粗略估算，PC的视频速率是每秒75帧，而8位处理器的视频速率是小于每秒1帧。虽然这是一个荒唐可笑的例子，但它在有关硬件和性能的重要问题中确实是主要因素。让我们试着来总结一下：

- 更快的时钟速率通常意味着更高的性能，但并不全部是这种情况。
- 更大的总线宽度意味着在每次操作中能移动更多数据，并且每条指令能操纵更多数字。
- 更大的可寻址存储器意味着更高效的存储器访问。
- 多流水线意味着CPU在每个时钟周期常常能完成多于1条指令的执行。
- 片上cache以时钟的速度提供指令和数据。
- 专用的片上硬件，如浮点单元，大大地加快了那些将数表示成浮点格式进行操作的指令的执行速度。
- 专用的片外硬件加速器，如图形处理器、声音芯片、以太网芯片等等，能够使CPU免除I/O数据处理的负担。

这些要点中有对你产生重大启发的吗？可能没有，但你不能确信。至少我们对它们进行了逐一的列举。然而，有一个有趣的概念隐藏在以上的讨论中，我们需要对其进行更深入一点的考察。在两个例子中，我们考虑了如何用专门硬件加速浮点运算和图像操作计算。用硬件对算法加速是实现更高性能的通常做法。图15-1示意出了这个原理。

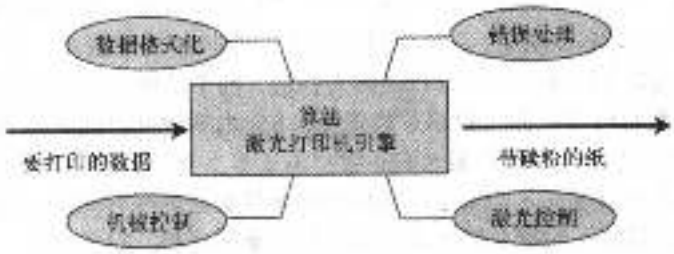


图15-1 将图像打印到一个激光打印机的算法的一般模型。由Berger[2]提供

在这个算法中，以图像信息形式表示的数据由激光引擎接收，该信息用于控制光敏鼓对激光束的曝光。哪里激光束被写到鼓上，精细颗粒的黑碳粉就附着于哪里的感光区域。然后这个图像就转移和溶化到普通的纸张上，从而得到打印机的输出。

在描述这个算法时，我们确实提及到一些硬件，但成像过程本身有意地未予明确说明。数据的位是如何转换成调制激光光束的呢？我们可以在软件中做这件事，或者我们可以想像由某种专门硬件为我们做这种转换。事实上，目前两种方法都有广泛的使用，不同厂商可采用不同的性能策略来达到有竞争力的结果。A公司可能关注于精细地调整他们的软件算法，而

401

B公司可能通过采用不那么强大的处理器来缓和软件处理要求，但要用专用硬件加速器来加速转换过程（称为banding）。

目前，存在第三种选择。由于在PC上可得到如此强的处理能力，很多打印机厂商只需给打印机配备必要的最小化智能器件，这使得激光打印机的价格大幅度下降。所有的处理需求

都已放回到了PC的打印机驱动程序中。

我们将这个现象称为软件和硬件的对偶性（duality），因为使用它们中任何一个（或者两个都用）均可实现算法。算法在软件（低速、低成本、灵活的）和硬件（快速、高成本、严格定义的）之间的划分要由规划师和设计师来决定。这种对偶性不是非黑即白，它代表的是各种折中和设计决策结果的一个谱系。图15-2图示了从专用硬件加速到只用软件加速的一个连续的范围。

因此，我们可以换一个角度来看待性能问题。我们可以问："必须在体系结构上做什么样的折中才能达到期望的性能目标吗？"

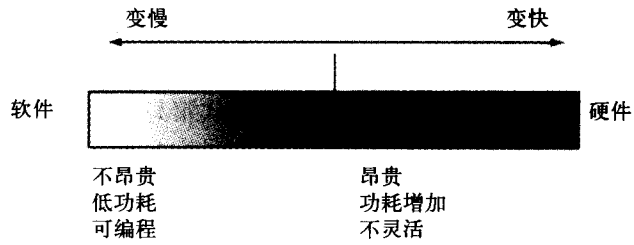


图15-2 硬件/软件折中

随着硬件描述语言的出现，我们现在就能像设计软件一样，通过只关注算法来设计硬件了。我们可采用面向对象的设计技术和基于UML的工具来生成C++或HDL源文件作为设计的输出。有了这些用于在设计过程中对硬件进行精细调节的方法，随着算法平滑地在软件组件和硬件组件之间的划分，性能改进就成了逐步可达到的目标。

超频

一种非常有趣的亚文化已经发展起来了，它通过对处理器、存储器或处理器和存储器进行超频（overclocking）来提高性能。超频的意思是有意识地用比设计时假定的时钟速度更高的速度运行。现代PC的母板具有令人惊异的灵活性，使有知识的或不那么有知识的用户都能滥用如时钟频率、总线频率、CPU核电压以及I/O电压这些东西。

在Web上搜索一下，你就会发现很多关于这个有趣技术的专门web站点。每年我教的很多学生都会问我这个问题，所以我认为本章应该讲讲这个问题。由于超频从定义上违反了制造商的设计规范，所以CPU制造商想方设法阻止这些狂热者，虽然结果通常是混乱的。

现代CPU通常将内部时钟频率锁相（phase lock）到外部总线频率。一个称为锁相环（phase-locked loop, PLL）的电路产生内部时钟，其频率是外部时钟频率的倍数。如果外部时钟频率是200MHz（PC3200存储器），乘因子是11，那么内部时钟频率就是2.2GHz。PLL然后将其内部时钟频率除以11，并将除后得到的频率与外部频率比较，这个局部频率差异就用于加速或减慢内部时钟频率。 402

你可以通过下面两种方法之一来实现处理器超频：

1. 改变CPU的内部乘因子；

2. 提高外部参考时钟频率。

CPU制造商是通过使用硬连线将该乘因子连接到一个固定值来解决这个问题的，虽然有进取心的爱好者已经想出了打破这个固定值的方法。如果母板支持相关特性，那么改变外部时钟频率就相对容易做，而且很多配件市场的母板生产商已经加入了这些特性以迎合想超频的团体大众。一般来说，当你改变了外部时钟频率时，你也就改变了存储器的时钟频率。

好，那么，不好的方面是什么呢？简单的回答是CPU的设计规范中没有比规定速度运行更快的情况，所以如果你要那样做，就违反了设计规范。让我们更深入地看一下这个问题。一个集成电路要设计成在规定的温度范围内满足所有的性能参数。例如，AMD的Athlon处理器规定要在摄氏90度以下满足其参数的设计规范。通常，在芯片的操作温度范围内，每个时序参数用三个参数

来规定：最小值、典型值及最大值（最坏情况）。这样，如果你取来大量芯片并将它们放置在昂贵的参数检测机器上，你将发现大部分芯片的时序参数都呈现钟形的曲线。曲线的峰值围绕着典型值，最大值和最小值之间的大概范围就定义了典型值的两侧。最后，你将芯片温度变得越低，它运行得越快。器件物理学告诉我们，当芯片变热时，集成电路中的电迁移就会变慢。

如果你近距离地观察一个充满刚刚流片的Athlon或奔腾的IC圆片，那么你还会发现有一些外表不同的芯片均匀地分布在圆片的表面上。这些芯片就是实际用来表征制造的每一批量圆片的参数的芯片。这样，如果制造过程恰好进行顺利，那么你就会得到一批比典型CPU更快的CPU。如果该过程勉强可接受，那么你得到的可能就是一批比典型芯片慢的CPU。

假设你作为一个制造者已经确实很好地调节了制造过程，使得制造出的所有芯片都比平均情况下制造的好得多。你要怎么做呢？如果你曾买了一台个人计算机，或者用部件组装了一台个人计算机，你就会知道更快的电脑价格就更贵，因为CPU制造商为更快的部件设定了更高的价格。这样，一个速率为3200+的Athlon XP处理器就比一个速率为2800+的Athlon XP处理器更快，价格也应该更贵。但是，假设制造的都是真正的快速器件，由于仍然需要提供具有不同价位的器件，你就要将较快的芯片标记为较慢的芯片。

因此，超频可能采用如下的策略：

1. 加速处理器，这既可能由于制造商保守地标定速率，也可能由于市场或销售的原因而有意地将速率标定得小于其实际性能。

2. 加速处理器，提高系统的冷却能力，以尽可能保持芯片冷却，使其能应付更高时钟频率所产生的额外热量。

403 3. 提高CPU核的电压或I/O的电压或者两者都提高，以降低逻辑信号上升和下降的时间。这样做的后果是增加了芯片产生的热量。

4. 提高时钟频率直到计算机变得不稳定，然后再退回某些值。

5. 提高时钟频率、核的电压以及I/O电压直到芯片自己毁坏。

超频的危险现在就应该很清楚了：

1. 运行时，芯片越热就越有可能失效。

2. 依赖典型的样本不能保证在所有温度和参数条件下的性能。

3. 突破制造商规定的阈值将使你失去保障。

4. 你的计算机可能勉强运行稳定，但对故障和干扰高度敏感。

那么，你应该为了提高性能而对你的计算机进行超频吗？下面的方针有助于回答这个问题：

如果你用PC是一种业余爱好行为，如玩游戏，则可以尽一切办法进行超频实验。但是，如果你要PC做真正的工作，就不要冒险进行超频。如果你真想提高PC的性能，就多加一些存储器。

度量性能

在个人计算机和工作站的世界中，性能度量通常是留给其他组织来做的。例如，大多数人都熟悉SPEC系列这一套软件测试基准。SPECint和SPECfp这两个测试基准分别用来度量整数和浮动数运算的性能。SPEC是标准性能评估公司（Standard Performance Evaluation Corporation）的缩写，这是一个由计算机制造商、系统集成商、大学以及其他研究机构所组成的非盈利性团体。他们的目标是为计算机系统建立、维护和发布一组相关的测试基准以及测试基准的结果[4]。

为回应“为什么使用测试基准？”这个问题，SPEC的常见问题（Frequently Asked Question）那一页写到：

在理想上，最好的系统对比测试环境应该是你自己的应用并采用你自己的工作负荷。但不幸的是，要为你自己的应用和采用你自己的工作负荷进行不同系统的比较，想得到有广泛基础的可靠的、可重复的、可比较的度量常常是很困难的。这可能是由时间、金钱、保密或其他限制等原因所造成的。

这里的关键是：最好的基准测试就是你实际的计算环境。然而，很少有要买PC的人有时间和倾向去将他们所有的软件装入若干台计算机中，然后对每个机器运行几天他们自己的应用软件，以此得到对每个系统相对能力的感性认识。因此，我们倾向于让其他人，通常是计算机制造商或者是第三方评论者来为我们做基准测试。即使在这时，在绝对公平的赛场上，404
要比较几台机器也几乎是不可能的。潜在的差异可能包括：

- 每台机器的存储量不同。
- 每台机器的存储器类型不同（PC2700和PC3200不同）。
- CPU的时钟速率不同。
- 硬件驱动程序的修正版本不同。
- 视频卡不同。
- 硬盘驱动器不同（串行ATA或者并行ATA，SCSI或者RAID）。

一般来说，我们会给予我们正在采用或打算采用的应用类似的测试基准以更多的信任。因此，如果你有兴趣为一个动画工作室购买高性能工作站，你就可能要从SPEC提供的图形测试套组中选择。

在嵌入式的世界中，性能度量和测试基准要更加难以获得和弄清楚，基本原因是嵌入式系统不是像工作站和PC那样的标准平台。几乎每个嵌入式系统在CPU、时钟速度、存储器、支持芯片、使用的编程语言、使用的编译器，以及使用的操作系统方面都是独具特点的。

由于大多数嵌入式系统对成本是极其敏感的，所以在设计系统时，除了实际恰好所需以外，理论上通常就很少或没有提高性能的余地。而且，嵌入式系统通常是用于实时控制应用中，而不是计算应用中。因为系统的性能会受外界特性和实时事件频率的严重影响，所以这些事件必须在明确定义的时间范围内得到服务，否则整个系统可能会出现灾难性的故障。

想像你正在为一个新式的由电子控制的喷气式战斗机设计一个飞行控制系统。飞行员不是以传统的方式控制飞机，因为从座舱到飞行控制表面没有直接的缆线连接。飞行员通过控制杆和方向舵踏板向飞行控制计算机（或多个计算机）发出请求，计算机响应请求，对翼面和尾面作出调整。使飞机具有如此高机动性的因素也正是使它难以飞翔的因素。使飞机具有如此高机动性的因素也正是使它难以飞翔的因素。没有计算机对飞机状态的持续监视以及对飞行控制表面的快速调整，飞机就会失控。

除非计算机能读到其所有的输入传感器并在适当的时间范围内作出所有要求的校正，否则飞机在飞行中将不会稳定。我们称这种情形为“时间关键的”（time critical）。换句话说，除非能在限定的时间内响应，否则系统将失效。

现在，让我们换一下雇主。这次你正在为彩色照片打印机设计一些软件。市场部门已经写出了需求文档，规定了每分钟4页的输出速率。头一个样机实际上每分钟输出了3.5页，这个打印机能工作，没有人受到损害，但它仍未满足设计规范。这是一个时间敏感的（time sensitive）例子。系统能工作，但没有达到要求。大多数有实时性能要求的嵌入式应用都包括在这两类中。

问题还有待回答，“什么样的测试基准与嵌入式系统是相关的？”我们可以采用SPEC测试基准集，但它们与我们关心的应用领域相关吗？换句话说，“一个做素数计算的测试基准对405
于比较三个嵌入式处理器在火炉控制系统中的潜在效用方面有多大意义呢?”。

在很长一段时期都没有适于嵌入式系统界使用的测试基准。可得到的测试基准多是可买卖的，而不是有用的技术评估工具。其中最声名狼藉的是MIPS测试基准。MIPS测试基准的意思是每秒百万条指令数（millions of instructions per second）。然而，它变成了下面的意思：

对销售员来说它是无意义的性能指示器。

MIPS测试基准实际上是将你的CPU性能与一台VAX 11/780计算机的性能相比较的一个相对度量。11/780是一个1MIPS的机器，可在1秒内执行1757次Dhrystone[5]测试基准的循环。因此，如果你的计算机能执行2400次这个测试基准的循环，那么它就是2400/1757=1.36MIPS的机器。Dhrystone测试基准是一个小的C、Pascal或Java程序，能编译成大约2000行的汇编代码。它是被设计用来测试处理器整数性能的，它不使用任何操作系统服务。

除了人们用Dhrystone测试基准做了会产生经济影响的技术决策以外，Dhrystone测试基准本身并没有什么错。例如，如果因为处理器A在Dhrystone测试基准上得到了更好的结果，我们就选择处理器A而不是B的话，就会导致客户在他们的设计中使用很多个A型的处理器。你如何能使你的处理器在Dhrystone测试基准上看起来确实很好呢？由于该测试基准是用一种高级语言写的，编译器开发者可针对Dhrystone测试基准做特殊的优化。当然，编译器卖家决不会做这样的事，但是每个人都指责其他人走类似的捷径。根据Mann和Cobb所说[6]：

> 不幸的是，所有常用于处理器评估的测试基准程序都相对较小且指令有高的cache命中率。像Dhrystone这样的程序就有这个特点。而且它们没有展示出很多真实应用中具有代表性的大量数据移动行为。

Mann和Cobb引用了下面的例子：

> 假设你在一个处理器上运行Dhrystone并发现μP（微处理器）在P周期内执行了某些次迭代，且cache命中率接近100%。现在，假设你从你的应用固件中选取了一个类似长度的代码序列并在相同μP上运行这段代码。你可能期望着该段代码能运行类似长的时间。
>
> 让你沮丧的是，你发现这次cache命中率变成了只有80%。在目标系统中，每一次的未命中都要付出11个处理器周期的代价，这期间系统要等待cache从慢速的存储器中再填满；11个周期对于50MHz的CPU来说只是220ns。但执行时间却从Dhrystone的P周期增加到了(0.8×P) + (0.2×P×11)=3P。换句话说，如果你的预测完全是基于Dhrystone的结果，则cache命中率下跌到80%就将使总体性能削减到只有期望水平的33%。

为了集中力量研究嵌入式产业的测试基准问题，芯片供应商和工具提供商在1997年组成了一个团体，由EDN杂志的技术编辑Marcus Levy领导。该团体的目标是寻求创建对嵌入式系
406 统中硬件和软件[7]有意义的性能测试基准。EDN嵌入式微处理器测试基准协会（EEMBC，读作“Embassy”）采用了来自各种产业领域的真实的测试基准。

所代表的领域有：

- 汽车/工业
- 消费者
- Java
- 网络
- 办公自动化
- 远程通信
- 8位和16位微控制器

例如，在远程通信组有5个类别的测试，在每个类别内有若干不同的测试。这些类别是：

- 自相关
- 卷积编码器
- 定点位分配
- 定点复数FFT
- Viterbi GSM解码器

如果这些对你来说有点晦涩难懂，它们多数确实是这样的。这些算法深深地扎根于远程通信产业的技术中。让我们看一个例子，这是在一台750MHz德州仪器TMS320C4X数字信号处理器（DSP）芯片上运行EEMBC自相关测试基准的结果。该结果显示于图15-3中。

这个柱形图显示出的测试基准采用了没有开启优化的C编译器；进行了有进取性的编译；手工对汇编语言进行了精细调整。这个结果给人以深刻印象。如果把已经优化了的C代码用汇编语言进一步地手工优化，那么测试基准的结果几乎会有100%的提高。而且，这两个优化后的测试基准的结果要比没有优化的结果分别好19.5%和32.2%。



图15-3 远程通信组自相关测试基准中EEMBC测试基准的结果[9]

让我们再细致地考察一下这个问题。在所有其他情况都相同的情况下，我们将需要把无优化结果的时钟速度从750MHz提高到24GHz，才能达到与手工调整的汇编语言程序测试基准相同的性能。

虽然EEMBC测试基准是重大的进步，但仍有一些因素致使比较结果变得相当无意义。例如，我们刚看到编译器优化对测试基准结果的影响。除非用可比较的编译器并对测试基准施加优化，否则结果可能就是被歪曲的、错误解释的。

另一个相当独特的嵌入式系统问题是热板（hot board）问题。制造商建造了，承载着处理器的评估板（evaluation board），使得还没有得到硬件的嵌入式系统设计者可在该处理器上运行测试基准代码或其他程序。评估板的定价经常是高于嗜好者所愿意花费的价钱，但低于总经理可直接同意购买的价钱。从一个芯片制造商的角度，我知道如果我的芯片被选用于客户的新产品，则评估板就代表一个潜在的滚滚流来的收入。因此，我将使评估板的性能特性最优化，使得测试基准的结果尽量好。这样的板称为热板，它们通常不代表真实硬件的性能特性。图15-4就是AMD AM186EM微控制器的一个评估板。毫不奇怪，它的价格是186美元。

评估板包含可得到的最快型号的处理器（40MHz），而且RAM存储器足够快，无需任

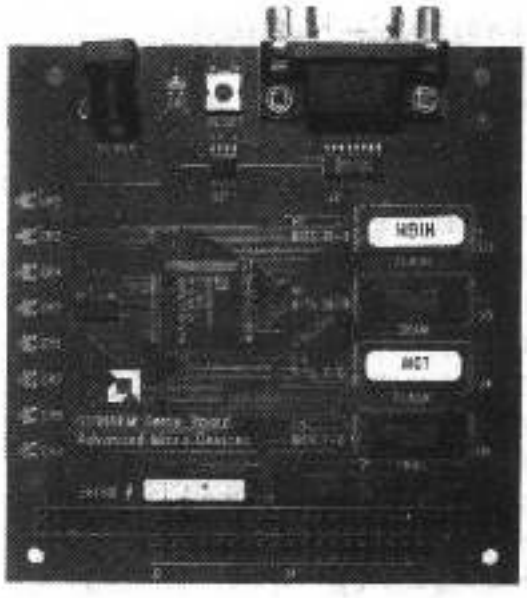


图15-4 来自AMD的AM186EM-40微控制器的评估板

何额外的等待状态就跟得上处理器的速度。使用该板所需要的就是加一个5V的直流供电电源和一根接到PC串行通讯端口的RS232电缆。与评估板一起提供的还有一个存储在ROM中的板上监视程序，用于在加电时启动一个通信会话。所有的一切都是很方便的，但你必须确信这反映了你的目标硬件的实际操作状况。

另一个要考虑的重要因素是你的应用是否要运行在一个操作系统上。操作系统本身引入了额外的开销，有可能会降低性能，而且，如果你的应用是一个低优先级的任务，它就有可能由于高优先级任务不断地进行而中断，从而因得不到CPU周期而饿死。

一般来说，所有的测试基准都是相对于某个时间线来度量的。我们或者度量一个测试基准运行所需的时间量，或者度量一个测试基准在单位时间（如1小时、1分钟、1秒钟或1秒钟的若干分之几）所能重复运行的次数。有时，我们可简单地对运行足够长的事件计时，我们可以用一个秒表来度量两次向控制台写入的时间间隔。你可以通过在代码中插入printf( )或cout语句轻松地做到这一点。但是，如果你要计时的事件只运行几毫秒或几微秒该怎么度量呢？如果你能得到操作系统的服务，你就能用更高的时间分辨率来记录进入点和退出点。然而，每一次对O/S服务或库程序的调用都会对你试图要度量的系统有潜在的巨大干扰，这有点像海森堡的测不准原理。

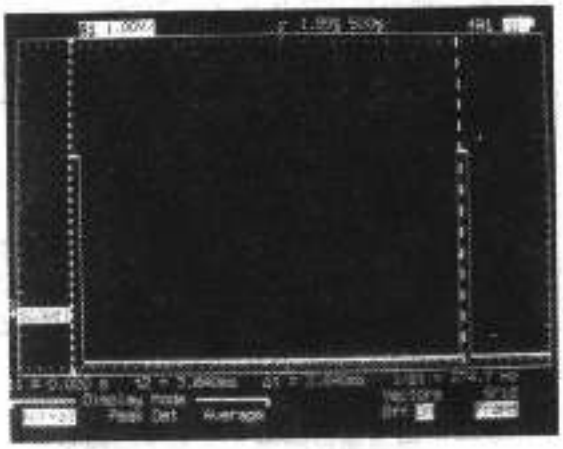


图15-5 采用示波器测量函数进入点和退出点之间的时间差，从而对软件性能进行度量

在有些例子中，评估板可能包含I/O端口，你可以对其开关。用一台示波器或其他某个高速数据记录仪，你可以直接对某事件或多个事件进行计时，并且对系统产生最小干扰。图15-5显示出用示波器对一个函数的进入点和退出点进行记录而作出的软件时序度量。参考此图你会看到，当有函数进入时，I/O引脚就会被打开，然后关闭，产生一个短脉冲。在退出时，再生成一个脉冲。这两个脉冲的时间差就是函数执行时间的度量。两条垂直的虚线是指示标，可放置于波形上以确定时序参考标记。在本例中，两个指示标之间的时间差是3.640ms。

408

另一种方法是采用数字硬件设计者选择的工具，即逻辑分析仪。图15-6就是由Tektronix公司生产的TLA7151逻辑分析仪的照片。在照片中，逻辑分析仪有多股线通过专用电缆连接到计算机板的总线。在板上提供专用端口使得逻辑分析仪可以很容易地连接到板上，这是电路板设计者的共同经验和良好思想。逻辑分析仪使设计者能在同一时间记录很多数字位的状态。想像一下，你能对80个数字位宽的数字系统的数据和时间同时记录1百万个采样。你可能将32位用于数据，32位用于地址总线，其余的16位用于各种状态信号。而且，逻辑分析仪



图15-6 Tektronix TLA7151逻辑分析仪的照片。用逻辑分析仪的电缆探测计算机板上的总线信号。照片由Tektronix公司提供

中的电路可被编程为只记录特定的位模式。例如，假设我们将逻辑分析仪编程为只记录向存储器地址0xAABB0000中写入的数据。逻辑分析仪将监视所有的位，但只有在地址与0xAABB00匹配时且状态位指示数据写在进行时，才记录数据线上的32位数据。此外，每次逻辑分析仪记录数据写事件时，都会给事件加上时间印记，并将时间和数据一起记录下来。

我们在这个例子中要考虑的最后一个要素是在代码中插入适当的参考单元，使得当它们发生时逻辑分析仪能检测并记录它们。例如，假如我们要用位模式0xAAAAXXXX作为一个函数的进入点，0x5555XXXX作为退出点。“X”的意思是“无关”，它可以是任意值。然而，我们可利用它们为程序中的每个函数赋予唯一的标志符。 409

让我们看一个程序中的典型函数。下面就是这个函数：

```
int typFunct(int aVar, int bVar, int cVar)
{
------------ /*Lines of code*/
------------ /*Lines of code*/
------------
------------
}
```

现在，让我们加上我们的度量“标签”。我们称这个过程为对代码加检测（instrumenting）。下面就是加检测后的函数：

```
int typFunct(int aVar, int bVar, int cVar)
{
*(volatile unsigned int*)0xAABB0000=0xAAAA03E7
------------  /*Lines of code*/
------------
------------
------------
*(volatile unsigned int*)0xAABB0000=0X555503E7
}
```

这个相当难解的C语句*(volatile unsigned int*) 0xAABB0000 = 0xAAAA03E7产生了一个指向地址0xAABB0000的指针，并立即将值0xAAAA03E7写入了这个存储器位置。我们可以假设0x03E7就是我们分配给函数typFunct( )的代码，这个语句就是我们的标签生成器，它生成数据写动作，然后逻辑分析仪就能对其进行捕获和记录。关键字volatile告诉编译器这个写操作不用进行缓存。图15-7就是这个过程的示意图。我们将图15-7的数据总结在一个表中。 410

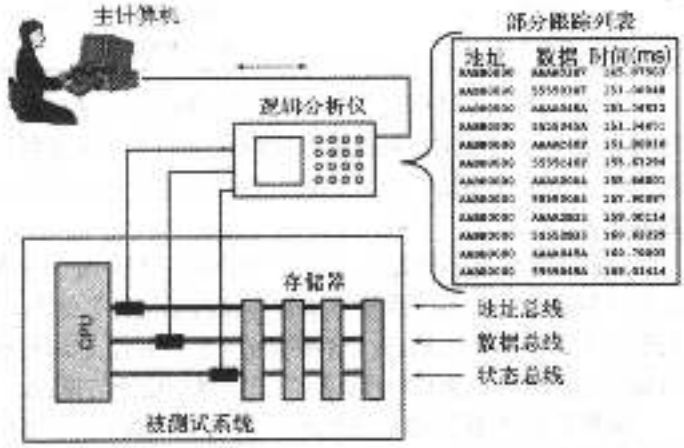


图15-7 通过采用逻辑分析仪记录进入点和退出点对软件性能作出度量

查看该表，注意标记为045A的函数为什么分别有两个不同的执行时间0.27819和8.33411。这看起来可能有点奇怪，但实际上很平常。例如，一个递归函数可能有不同的调用数学库程序的函数，也就有不同的执行时间。然

而，这可能也预示着这个函数会被中断，所以其运行时间就可能要取决于系统的当前状态和I/O活动而大幅度地变化。

这里的关键问题是你几乎可以将度量做得任意地谨慎。对无cache存储器的单个写操作所产生的开销应该不会对度量产生过大的扭曲。而且，还要注意逻辑分析仪是连接到另一个主计算机上的。假如这个主计算机是用于做初始源代码测试的，它就应该对符号表和链接映像有所访问。因此，它就能通过实际地提供函数名而不是标识符码来提交结果。

这样，只要待测系统运行足够长的时间间隔，我们就能像图15-7所显示的那样持续地收集数据，然后做一些简单的统计分析来为这些函数确定最小平均执行时间、最大平均执行时间及平均执行时间。

这种类型的度量还能使我们获得什么其他类型的性能数据呢？一些度量总结如下：

1. 实时跟踪：当程序实时运行时，记录函数的进入点和退出点就能提供程序执行路径的历史信息。这种调试技术不停止程序的执行流程，它不是单步运行的，也不是运行到断点就停止的。

2. 覆盖测试：该测试保留程序运行部分和未运行部分的踪迹。这对于定位死代码和执行附加有效性测试都是有价值的。

3. 存储器泄漏：在存储器动态分配和释放的每个位置上放置标识，就可以确定系统是否存在存储器泄漏或碎片问题。

4. 分支分析：通过对程序分支加入测试，就能确定代码中是否存在不可跟踪的或没有彻底测试的路径。对于任何被认为是有关键使命的（mission critical）并且在应用到实际产品前必须由政府管理机构鉴定的代码，这个测试是必需的。

函数	进入点/退出点（msec）	时间差
03E7	145.87503/151.00048	5.12545
045A	151.06632/151.34451	0.27819
C40F	151.90018/155.63294	3.73276
00A4	155.66001/157.90087	2.24086
2B33	158.00114/160.62229	2.62115
045A	160.70003/169.03414	8.33411

虽然逻辑分析仪提供了一个侵扰性很低的测试环境，但并不是所有的计算机系统都能以这种方式来进行度量。就像前面所讨论过的，如果有操作系统，则标识的生成过程和记录可作为另一个操作系统任务来完成。当然，这显然更有侵扰性，但对某些情况也许是一个合理的解决方法。

到此你也许忍不住建议："为什么为标识烦恼呢？如果逻辑分析仪能记录发生在系统总线上的任何事情，只要记录所有事情不就行了吗？"这是一个好的观点，但它只能在无cache的处理器上工作良好。然而，只要你有一个具有片上cache的处理器，总线活动就不再是处理器活动的指示器。这就是为什么标识工作得如此好的原因。

虽然逻辑分析仪对于这些种度量而言工作得相当好，但也有其局限性，因为必须要停
411 止收集数据并成批地上传跟踪存储器的内容。这意味着像中断服务例程这样的低占空因数的事件可能未被捕获。Metrowerks[9]有像CodeTest这样的商业产品，它能在不停止的情况下，

通过不断地对标识进行收集，压缩，并发送到主计算机来解决这个问题。图15-8就是CodeTest系统的图片，图15-9显示的是性能度量的数据。



图15-8 针对实时系统的CodeTest软件系统性能分析仪。图片来自Metrowerks公司

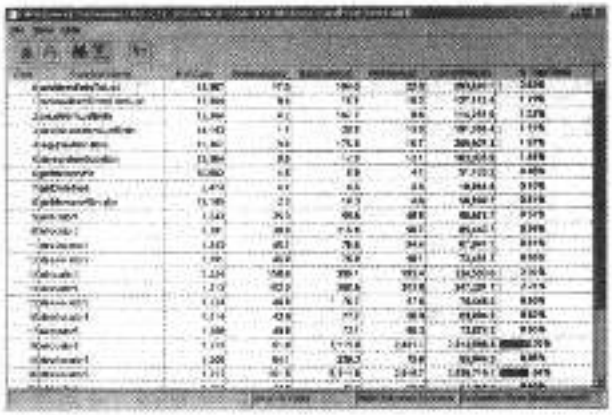


图15-9 CodeTest的屏幕照片显示出了软件性能度量。当目标系统实时运行时，数据是连续更新的。照片来自Metrowerks公司

性能设计

学软件的学生应该学习计算机体系结构的最重要原因是为了理解他们的软件所要运行于其中的机器和环境的弱项和局限性。没有对机器在操作上的特性的一个理性的洞察，就很容易写出低效率的代码。更坏的是，很容易将低效率的代码误认为是由硬件平台本身的局限性造成的。这可能导致对硬件进行重新设计的决策，以使系统性能提高到期望的水平。虽然简单地重写一些关键函数就可能解决问题。

下面的故事是一个实际事件，可用来阐明这一点：

很久以前，在很远的地方，我是CodeTest产品研发的主管。一个主要的远程通信制造商正考虑买一大批CodeTest的设备，所以我们从工厂派出一个队伍来演示产品。客户打算要对其出售的一种主要的远程交换系统进行重新设计，因为他们认为该系统已经达到了硬件性能的极限。

我们的队伍到达他们那里后，在他们的硬件中安装了一个CodeTest部件。在对他们的交换器测试了若干个小时后，我们一起检查数据，在我们考察的几百个函数中，没有一个工程师能识别出那个耗费了15%CPU时间的函数。在对源代码进行钻研后，工程师们发现了一个由实习学生加入的调试程序。这个实习生将调试该系统的一部分作为他夏季的项目，为了跟踪程序流程，他编制了一个高优先级的函数，使交换器的一个电路板的灯闪光。作为一个实习生，他绝不会担心能否正确地识别出这个函数是一个临时调试函数，而且不知何故这个函数就被包装进了发行的产品代码中。

在消除函数和重新建立文件后，客户获得了15%的额外性能空间。他们对此结果非常兴奋，慷慨地酬谢了我们，并招待了我们一顿丰盛的宴会，不幸的是，他们不再需要CodeTest仪器了，我们失去了这笔买卖。这个故事的寓意是，没有人会去真正检查系统的性能特性。每个人都假设他们的代码工作良好，系统总体运行在最佳状态。 412

Stewart[10]注意到实时软件开发者所犯的首要错误是不知道他们代码的实际运行时间。这不仅仅是一个学术问题，即使你正在开发的软件是为了运行于PC或工作站，你也要考虑系统能否获得最佳性能的问题；这与任何其他形式的工程问题是一样的。你应该努力以最高效率利用你可以得到的资源。

性能问题在具有有限资源或实时性能约束的系统中最为关键。一般来说，这是大多数嵌入式系统的研究领域，所以我们将注意力集中在该领域。

Ganssle[11]不主张在C或C++中写中断服务程序，因为其执行时间是不可预测的。使代码执行达到可预测的唯一途径是用汇编语言写程序。是这样的吗？如果你采用一个带有片上cache的处理器，你如何知道代码的cache命中率是多少？中断服务程序实际上的运行时间要比汇编语言周期数所预测的时间长得多。

Hillary和Berger[12]描述了一个能满足软件设计性能目标的4步过程：

1. 确立一个性能预算；
2. 对系统建模并分配预算；
3. 测试系统模块；
4. 验证最终设计的性能。

性能预算可定义为：

$$\begin{aligned}\text{性能预算} &= \text{求和（最坏情况条件下所需的操作）}\\ &= [1/(\text{数据速率})] - \text{操作系统开销} - \text{裕量}\end{aligned}$$

数据速率就是数据被生成并将被处理的速率。由此可知，你必须减去操作系统的开销，而且最终要为因加入额外特性而总是要加入的代码留有余地。

对系统建模意味着将预算分解成所需的功能模块，并为每一个模块分配时间。大多数工程师没有关于不同功能所需时间量的线索，所以他们进行“推测”。实际上，这没有那么坏，因为至少他们在生成预算。有很多种方式来细化这些猜测而不用实际地写出最终的代码并对实际情况进行测试。关键是提高对可得到时间和所需时间相比较的意识。

一旦软件开发开始，在模块级测试执行时间而不是等到集成阶段再看软件性能是否满足规范要求就是有意义的。这将给你一个关于代码对于性能预算的即时反馈。记住，猜测可能
413 有两种情况，太长或太短，所以你也可能有比你想像的更多的时间富余（虽然Murphy定律总是保证这是很小概率的事件）。

最后一步是验证最终设计。这意味着要采用我们已经讨论过的一些方法对系统性能进行精确的测量。拥有这些数据将使你能够在软件需求文件上签字通过，同时也能为以后的项目提供有价值的数据。

15.3 最佳习惯

让我们用一些最佳习惯的讨论来结束本章。有几百个最佳习惯，要全部讨论就太多了。然而，我们可以对某些性能问题以及该做哪些和不该做哪些进行一番了解。

1. 在你开始写代码之前制作一个需求文档和设计规范。遵循一个公认的软件开发过程。与大多数学生所想的相反，编制代码（code hacking）对于一个职业程序员并不是令人羡慕的特征。如果可能，在系统的体系结构设计决策完成前参与其中。如果没有其他的学习计算机体系结构的理由，这就是一个。在项目开始时，如果划分决策不佳，那么通常会导致软件团队在项目后期修补混乱方面产生压力。

2. 采用好的编程习惯。不管你是为一个PC编码，还是为一个嵌入式控制器编码，这条软件设计规则都同样适用。要很好地理解算法设计的一般原理。例如，如果数据量很大，就不要用$O(n^2)$的算法。无论硬件多好，低效的算法都能使最快的处理器停止。

3. 学习你要使用的编译器，并了解如何最大程度地利用它。大多数工业质量的编译器都是极其复杂的程序，通常其文档让人很难理解。因此，大多数工程师继续以他们过去的方式使用编译器，而不关心通过探究可利用的优化选项能获得什么样的性能改进。如果编译器本

身是针对特定的CPU体系结构构建的，就更是应该学习它。例如，有一个用于Intel i960处理器系列的GNU[13]编译器版本，能通过执行程序来生成性能的轮廓数据，并将该数据用于后续的编译-执行循环中，以提高代码的性能。

4. 了解代码的执行限制。例如，Ganssle[14]建议，为了确定为堆栈分配多少存储器，你应该用 414
如0xAAAA或0x5555这样的可辨别的存储模式将堆栈区充满。然后，将你的程序运行足够长的时间，确信它已被彻底地使用了。现在就可以通过观察位模式在哪里被覆盖来得到堆栈区的高水位标记（high water mark），然后加入一个安全因数，这就是你的堆栈空间。当然，这暗示着你的代码对于堆栈是确定性的。在高可靠性软件设计中，最不该做的事情之一就是使用递归函数。递归函数每次调用自身就生成一个堆栈帧，这就要继续建立堆栈。除非你绝对地知道最坏情况下的递归调用序列，否则不要使用递归。虽然递归函数简单易用，但在严格限制了资源的系统中也是危险的。而且，它们在函数调用和返回代码中有非常大的开销，所以性能会受到影响。

5. 当需要绝对控制时使用汇编语言。如果你知道如何用汇编语言编程，就不要害怕钻研进去做一些手工工作。所有的编译器都有将汇编语言包含进你的C或C++程序的机制。采用能满足所要求的性能目标的语言。

6. 当你为任何嵌入式系统或其他有高可靠性要求的系统进行设计时，要十分小心动态存储分配问题。即使在设计中没有存储泄漏问题，比如忘记释放分配的存储空间或者坏指针错误等，如果内存处理程序代码与你的应用不能很好地匹配，存储器也可能会碎片化。

7. 不要忽视处理器提供的所有异常向量。错误处理程序是重要的代码片段，能帮助你保持系统存活。如果你不利用它们，或者仅仅用它们导向一般的系统重新启动，那么你将永远不能追踪到为什么在每四年的2月29日系统就崩溃一次。

8. 确认你和硬件设计者在低地址存放最高有效位还是最低有效位两种模型的选择上达成一致。

9. 明智地使用全局变量。即使冒着招致计算机科学家愤怒的危险，我也不会说："不要使用全局变量"，因为全局变量为传递参数提供了一种非常高效的机制。然而，应该明白使用全局变量有危险的副作用。例如，Simon[15]在他对共享数据问题的讨论中阐明了像全局变量这样的与存储缓冲相关联的问题。如果一个全局变量被用于保存共享数据，则若一个任务试图读该数据同时另一个任务正在写这个数据，就可能会引起错误。系统体系结构会影响这种情况，因为全局变量的大小和外部存储器的大小可能在一个系统中产生问题但在另一个系统中就不是问题。例如，假设一个32位值正被用作全局变量，如果存储器是32位宽，则只需要一次存储器写来改变变量的值，所以两个任务可同时访问这个变量而不会产生问题。但是，如果存储器是16位宽，则需要两次连续的数据写来更新变量。如果在第一次存储器访问之后第二次访问之前，第二个任务中断了第一个任务，那么就会读到毁坏的数据。

10. 采用合适的工具进行工作。大多数软件开发者都不愿意在没有一个好的调试器的情况下调试程序。不要仅仅因为示波器或逻辑分析仪是"硬件设计者的工具"就害怕使用它们。 415

总结

本章包含如下内容：

- 各种硬件因素和软件因素会如何影响一个计算机系统的实际性能。
- 性能如何度量。
- 为什么性能并不总是意味着"尽量快"。
- 用于满足性能要求的方法。

参考文献

[1] Linley Gwennap, *A numbers game at AMD, Electronic Engineering Times,* October 15, 2001.

[2] http://www.microarch.org/micro35/keynote/JRattner.pdf (Justin Rattner is an Intel Fellow at Intel Labs.).

[3] Arnold S. Berger, *Embedded System Design,* ISBN 1-57820-073-3, CMP Books, Lawrence, KS, 2002, p. 9.

[4] http://www.spec.org/cpu2000/docs/readme1st.html#Q1.

[5] R.P. Weicker, *Dhrystone: A Synthetic Systems Programming Benchmark, Communications of the ACM,* Vol. 27, No. 10, October, 1984, pp. 1013–1030.

[6] Daniel Mann and Paul Cobb, *Why Dhrystone Leaves You High and Dry, EDN,* May 1998.

[7] http://www.eembc.hotdesk.com/about%20eembc.html

[8] Jackie Brenner and Markus Levy, *Code Efficiency and Compiler Directed Feedback, Dr. Dobb's Journal,* #355, December 2003, p. 59.

[9] www.metrowerks.com.

[10] Dave Stewart, *The Twenty-five Most Common Mistakes with Real-Time Software Development,"* a paper presented at the Embedded Systems Conference, San Jose, CA, September 2000.

[11] Jack Ganssle, *The Art of Designing Embedded Systems,* ISBN 0-7506-9869-1, *Newnes,* Newnes, Boston, MA, p. 91.

[12] Nat Hillary and Arnold Berger, *Guaranteeing the Performance of Real-Time Systems, Real Time Computing,* October, 2001, p. 79.

416 [13] www.gnu.org.

[14] Jack Ganssle, *op cit,* p. 61.

[15] David E. Simon, *An Embedded Software Primer,* ISBN 0-201-61569-X, Addison-Wesley, Reading, MA, 1999, p. 97.

习题

1. 热衷于在他们的PC上玩视频游戏的人经常加入液体冷却系统来排除来自CPU的热量。为什么？
2. 为什么在你的PC上加入更多存储器常常比用更快的CPU代替当前的CPU对性能有更大的影响？
3. 假设你正在试图比较两个计算机的相对性能。计算机1具有100MHz的时钟频率，计算机2具有250MHz的时钟频率。计算机1执行其指令集中的所有指令都用1个时钟周期。平均而言，计算机2执行其指令集中的40%的指令要用一个时钟周期，指令集其余的指令要用两个时钟周期。那么，运行一个包含先是1000条连续指令，接着100条指令的200次循环的测试程序，每个计算机各要运行多长时间。

   注释：你可以假设，对于计算机2，测试程序中的指令是以如上所述的匹配计算机总体性能的方式而随机分布的。
4. 对于给定的指令集体系结构，讨论可提高处理器性能的三种方式。
5. 假设计算机1平均每条指令需要2.0个时钟，时钟频率为1GHz。计算机2平均每条指令需要1.2个时钟，时钟频率为500MHz。哪一个计算机有相对更好的性能？将你的答案表达成百分比。
6. 假设你正在努力评估两个不同的编译器。为此你采用了某个标准测试基准，并用每种编译器分别将其编译成汇编语言。这个特定处理器的指令集体系结构是这样的，根据每条指令执行所需要CPU时钟周期数，汇编语言指令可分为4类，如下表所示：

指令类别	执行时所需CPU周期数
A类	2
B类	3
C类	4
D类	6

417 观察每个编译器的汇编语言输出，确定编译器产生的每类指令的相对分布。

编译器A将程序编译成1000条汇编语言指令，并产生如下所示的指令分布：

指令类别	每类指令所占百分比（%）
A类	40
B类	10
C类	30
D类	20

编译器B将程序编译成1200条汇编语言指令，并产生如下所示的指令分布：

指令类别	每类指令所占百分比（%）
A类	60
B类	20
C类	10
D类	10

对于这个测试程序，你预期哪个编译器会给出更好的性能？为什么？请尽量详述。

7. 假设你正在考虑对某个产品进行设计权衡，该产品对成本极端敏感。为了降低硬件成本，你考虑使用一款具有8位宽外部数据总线的处理器而不采用另一款具有16位宽数据总线的处理器。两款处理器以相同的时钟频率运行，其内部都是完全的32位宽。如果你打算将两个32位宽的存储器中的变量相加，并将和也存到存储器中，你预期会看到什么样的性能差异。
8. 为什么编译器会通过最大化代码中基本模块的大小和数量来优化程序？回顾可知，基本模块就是一个代码片段，具有一个入口、一个出口，且没有内部循环。 418

第16章　未来发展趋势与可重构硬件

学习目标

- 可编程逻辑如何实现；
- ABEL编程语言的基本要素；
- 什么是可重构硬件，如何实现；
- 现场可编程门阵列的基本体系结构；
- 可重构计算机器的体系结构；
- 分子计算的一些未来发展趋势；
- 无时钟计算的未来发展趋势。

16.1　引言

从第1章开始到现在我们已经走了很长的路，本章将结束本书并展望未来发展方向。这并不是说我们将跳到明星企业正在使用的计算机（虽然这样做会是一件令人高兴的事情），而是看看目前发展趋势会将我们引向哪里。沿此方向，我们将着眼于一个主题，该主题正变得越来越重要，但我们至今还没有在一个适宜的地方讨论它。

本课本的关注点一直是以软件开发者的角度来看待硬件。一个在若干年来已经发生并在增长的趋势是：硬件和软件之间的边界变得模糊了。无疑，软件是硬件状态机的驱动代码。但是，如果硬件本身就像软件算法一样可编程的话将会怎样？你能想像一台在加载软件之前没有任何个性的计算机吗？换句话说，68K、x86或ARM之间不存在任何区别，直至你加载一个新的软件。程序的一部分实际上是用于将硬件配置成期望的体系结构。你说这是科幻小说吗？不全是。继续读下去吧。

16.2　可重构硬件

从历史的视角来看，可配置硬件开始于20世纪70年代的数字集成电路PAL，即可编程阵列逻辑(programmable array logic)。PAL包含一组通用门和触发器，它们将以某种方式组织在一起，使得设计者能轻松地创建从简单到中等复杂的积和逻辑电路或状态机。

图16-1所示的门是一个非反相缓冲门。一个不提供逻辑功能的门可能看起来有点奇怪，但有时纯粹的逻辑应当遵从于数字电路电子特性的现实，这样我们就需要电路在

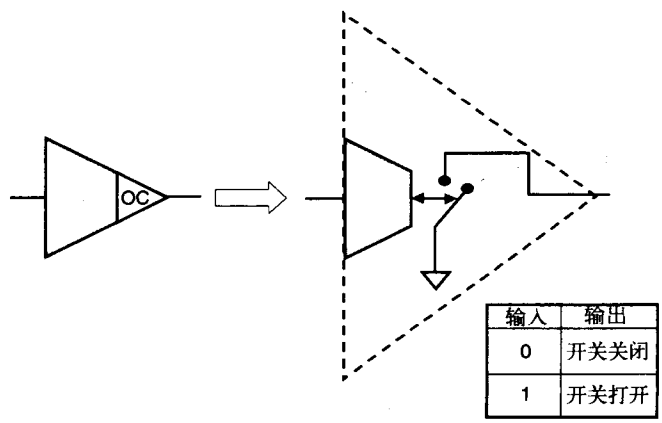


图16-1　非反相缓冲门的简化示意图，具有开集电极输出构造

输入到输出间存在非反相逻辑。在这个特定的例子中，使我们感兴趣的是输出电路的构造。这种电路构造称为开集电极（open collector）或开漏极（open drain），取决于所采用的集成电路工艺类型。就目前情况来说，开集电极这个词更通用，所以我们将采用这种叫法。

开集电极输出非常类似于三态输出，但有一些差别。回顾可知，三态输出能够将器件的逻辑功能（1或0）与器件的输出引脚隔离开来。开集电极器件以类似的方式工作，但其意图不是隔离输入逻辑和输出引脚，而是使多个输出能够连接在一起来实现像“与”和“或”这样的逻辑功能。当我们通过将开集电极门输出连接在一起来实现与函数时，我们称电路是线与（wired-AND）输出。

为理解电路如何工作，想像你正在观察图16-1中门的输出引脚。如果门输入是逻辑0，则你会看到开集电极输出的“开关”是关闭的，输出连到了地（逻辑0）。如果输入是1，则输出开关是打开的，没有连到任何地方，开关处于高阻抗状态，就像一个三态门。因此，开集电极门有两个输出状态，0或者高阻抗。

图16-2图解了一个3-输入的线与函数。请暂时忽略标记为F1、F2和F3的电路元件，它们是可永久“烧断”的熔丝，我们将随后讨论它们的用途。由于三个门都是开集电极器件，它们的输出或者连接到地（逻辑0），或者处于高阻抗状态。如果所有输入A、B和C都是逻辑1，则三个输出都是高阻抗。这意味着没有一个输出连接到公共线上。可是，电阻将公共线连接到了系统电源上，这样你就会看到该线是逻辑1。不同的是逻辑1是由电源通过电阻来提供的，而不是门输出提供的。我们将该电阻称为上拉电阻（pull-up resistor），因为它将线上的电压拉到了电源电压。

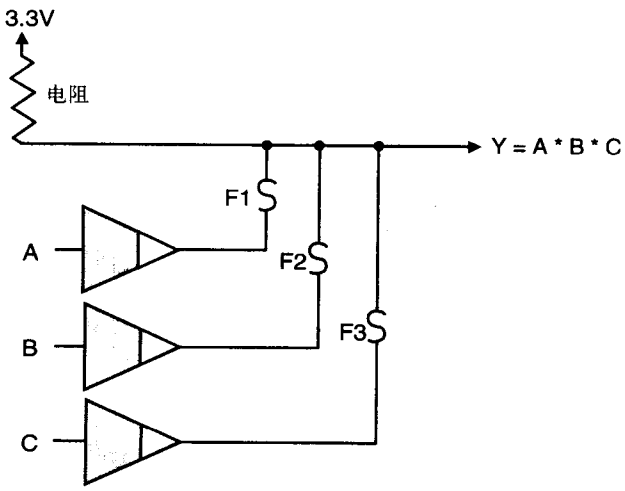


图16-2 3-输入的线与逻辑函数。标记为F1、F2和F3的电路符号是熔丝，可按特定意图烧断，从而使门输出永久性地从逻辑等式中消除

如果输入A、B和C是逻辑0，则该线被连接到地。电阻是必要的，它防止了电源直接连接到地，引起短路，产生有趣的烟火和气味。关键是电阻将从电源到地的电流限制到了一个安全值，同时也为我们提供了衡量逻辑电平的参考点。而且，有多少门输入处于逻辑0是没有关系的，其效果都是将线连接到地，从而强制输出为逻辑0。

熔丝F1、F2和F3对电路加入了另外的一维。如果有一种蒸发掉形成熔丝的导线的方法，比方说用大的电流脉冲，那么那个特定的开集电极门将被完全地从电路中移除。如果我们烧掉熔丝F3，则电路就是2-输入与门，包含输入A和B以及输出Y。当然，一旦我们决定烧掉某个熔丝，我们就不能按原样恢复它。我们称这样的器件为一次性可编程（one-time programmable）器件,即OTP器件。

图16-3显示出我们如何扩展线与函数的概念生成一个能实现任意积和逻辑函数的通用器件，该器件具有4个输入和2个输出。图中的圆圈代表可编程的交叉点开关（cross-point switch）。每个开关可能是一个如图16-2所示的OTP熔丝，也可能是一个如同三态门输出开关那样的电子开关。对于电子开关，我们还需要配置某种额外的存储器用来存储每个交叉点开关的状态，

420

或者需要一些其他的对开关进行重编程的方法。注意每个输入是如何通过输入/反相盒转换成输入和反相输入的，然后，每个输入和反相输入连接到阵列的每个水平线。水平线对该组垂直线实现了线与功能。通过有选择地对与平面（AND plane）的交叉点开关进行编程，每个或（OR）门输出就可成为任意的单级积和项。

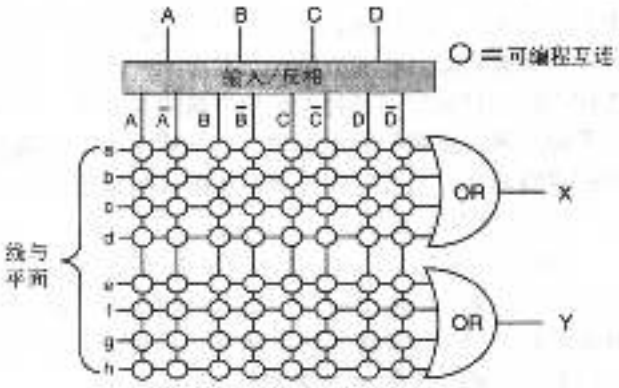


图16-3 一部分可编程阵列逻辑（PAL）器件的简化示意图

图16-4显示出两个工业标准PAL器件16R4和16L8的引脚外观图。参见16L8，我们看到它有10个引脚（引脚1到9和引脚11）只能用于输入，6个引脚（引脚12到18）可设定成输入或输出，两个引脚（引脚12和19）只能用于输出。另一个器件16R4类似于16L8，但它包括4个D型触发器以便于设计简单的状态机。

421

即使采用这些相对简单的器件，需要进行快速编程的互连点数量也有成百上千。为控制复杂度，可编程器件的编程工具制造商Data I/O公司[1]发明了最早的硬件描述语言之一，称为ABEL，这是高级布尔方程语言（Advanced Boolean Equation Language）的简写。ABEL现为一家加州San Jose的可编程硬件器件和可编程工具制造商Xilinx所拥有。

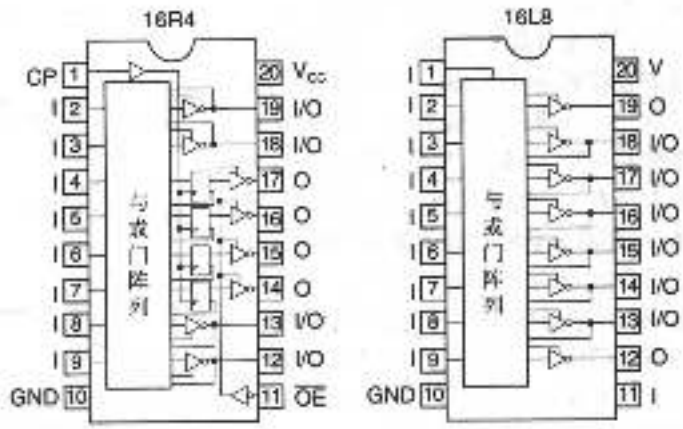


图16-4 工业界标准PAL器件16L8和16R4的引脚外观图

ABEL是一个比Verilog或VHDL简单的语言，但仍能定义相当复杂的电路配置。这里有一个ABEL源文件的部分实例。过程如下所示：

1. 生成ASCII源文件source.abl。

2. 编译该源文件，生成一个编程图source.jed。

3. 将source.jed文件载入到一个器件编程器（比如Data I/O开发的编程器）。对于OTP器件，可通过“烧断”（消除）适当的熔丝来编程。如果器件可重编程，则打开适当的交叉点开关。

一些适当的声明如下所示。关键字用粗体显示。

```
module      AND_OR;

title

Designer    Arnold Berger *
Revision    2
Company     University of Washington-Bothell
Part Number      U52
```

422

```
declarations
```

```
" Inputs

AND-OR     Device              PAL20V10 ;

a1   pin   1      ;
b1   pin   2      ;
b2   pin   3      ;
b3   pin   4      ;
b4   pin   5      ;
c1   pin   6      ;
c2   pin   7      ;
c3   pin   8      ;
c4   pin   9      ;
a4   pin   19     ;
a3   pin   18     ;
a2   pin   17     ;

"Outputs

zc   pin   11     istype        'com,buffer'       ;
yc   pin   12     istype        'com,buffer'       ;
yb   pin   13     istype        'com,buffer'       ;
zb   pin   14     istype        'com,buffer'       ;
za   pin   15     istype        'com,buffer'       ;
ya   pin   16     istype        'com,buffer'       ;

Equations

ya   =     a1 # a2 # a3 ;
!za  =     a1 # a2 # a3 ;

yb   =     b1 # b2 # b3 # b4 ;
!zb  =    b1 # b2 # b3 # b4 ;

yc   =     c1 # c2 # c3 # c4 ;
!zc  =     c1 # c2 # c3 # c4 ;

Test_vectors

( [ a1,a2,a3,a4,b1,b2,b3,b4,c1,c2,c3,c4]) -> [ya,za,yb,zb,yc,zc] ;

[1,1,1,x,1,1,1,1,1,1,1,1]   ->  [1,0,1,0,1,0] ;
[0,1,1,x,0,1,1,1,0,1,1,1]   ->       [1,0,1,0,1,0] ;
[1,0,1,x,1,0,1,1,1,0,1,1]   ->  [1,0,1,0,1,0] ;
[1,1,0,x,1,1,0,1,1,1,0,1]   ->  [1,0,1,0,1,0] ;
[1,1,1,x,1,1,1,0,1,1,1,0]   ->  [1,0,1,0,1,0] ;
[0,0,0,x,0,0,0,0,0,0,0,0]   ->  [0,1,0,1,0,1] ;

end ;
```

你可能对ABEL文件结构的很多地方都很清楚，但是源文件中还是存在某些部分需要详细阐述： 423

- 关键字device用于将一个物理部件和源模块联系起来。在这里，AND-OR电路将被映射到一个20V10 PAL器件。
- 关键字pin定义了哪个局部项被映射到哪个实际输入引脚或输出引脚。
- 关键字istype用于将一个明确的电路函数分配到一个引脚。这里输出引脚被声明为既是组合的（combinatorial）又是非反向的（buffer）。
- 在equations部分，符号‘#’用于表示逻辑或（OR）函数。虽然器件叫作AND-OR模块，但与（AND）部分实际上是反相逻辑意义上的与。德摩根定律提供了用或操作实现反相逻辑与函数的桥梁。
- 关键字test-vectors允许我们为输入和相应输出的有代表性的状态提供一个参考测试。编译器和编程器可以用它来验证逻辑方程和编程结果。

PLD很快就被复杂可编程逻辑器件（complex programmable logic device, CPLD）超越了。CPLD提供了更多数量的门和更强的功能。这两个系列的器件在可编程硬件应用上仍然非常流行。这里的关键点是产业标准硬件描述语言（ABEL）已经为硬件开发者提供了与软件开发者相同（或近乎相同）的设计环境。

演化过程的下一步就是现场可编程门阵列（field programmable gate array, FPGA）的出现。FPGA是作为原型工具引入的，其对象是进行定制集成电路（如ASIC）设计的工程师。ASIC工程师面临的是令人畏惧的任务，因为赌注巨大，与软件不同，硬件故障是不可原谅的。一旦硬件制造过程开始，几十万美元和数月时间就投入于制造部件。因此，硬件设计者要花费大量的时间对以软件描述的设计进行模拟。简言之，他们构造大量的测试向量（如同在前面ABEL源文件中显示的），并在设计付诸制造前尽可能多地用这些测试向量对设计进行验证。事实上，在设计付诸制造前，硬件设计者在测试上所花费的时间与实际用于设计所用的时间一样多。

更糟糕的是，全体开发队伍通常必须要等待直到设计周期的很晚阶段才能得到和使用硬件。FPGA的产生就为这种包含定制ASIC器件的硬件原型化问题提供了一种解决方案，但在这过程中，创造了一种全新的计算机体系结构的领域，称为可重构计算（reconfigurable computing）。在讨论可重构计算之前，我们需要更详细地考察FPGA。在图16-3中我们介绍了交叉点开关的概念。开关的技术有多种，包括：

424

- 可熔链接：开关初始都是关闭的，通过烧断熔丝使不需要的连接断开达到编程的目的。
- 反一可熔链接：链接初始是断开的。熔烧熔丝会引起开关单元永久性地开启，合上开关，使交叉导体连接上。
- 电可编程的：交叉点开关是可重编程器件，当用电流脉冲编程时，它可以保持自己的状态直至被重编程。这项技术与在数字相机、MP3播放机、计算机中的BIOS ROM中使用的FLASH存储器件类似。
- 基于RAM的：每个交叉点开关都可连接到RAM存储阵列中的位。阵列中的这个位的状态就决定了相应的开关是打开还是关闭。显然，基于RAM的器件可通过重写配置存储器中的这些位来快速地进行重编程。

FPGA引入了另一个新的概念：查找表（look-up table）。与PLD和CPLD器件采用线与体系结构且分别用或门将这些与项结合在一起来实现逻辑的做法不同，查找表是一个小的RAM存储元件，可通过编程提供任何组合逻辑方程。实际上，查找表就是一个直接实现在硅上的真值表。因此，与将真值表作为起点来产生门级电路或者用HDL实现逻辑方程不同，查找表只简单地将真值表看作是RAM存储器并直接实现逻辑函数。

图16-5图解了查找表的概念。FPGA可容易地采用一组查找表来实现，通常表示为5个输入和2个输出的表，并带有以D-触发器形式存在的寄存器以及用于对逻辑资源和存储资源之间的互连进行布线的交叉点开关网络。同样，时钟信号也需要布线以对电路提供同步。

在图16-6中，我们看到了一个完整的FPGA体系结构。该体系结构的不同寻常之处是它是完全基于RAM的。所有布线和组合逻辑都在FPGA内通过对存储器表进行编程实现。器件的全部个性化可以很容

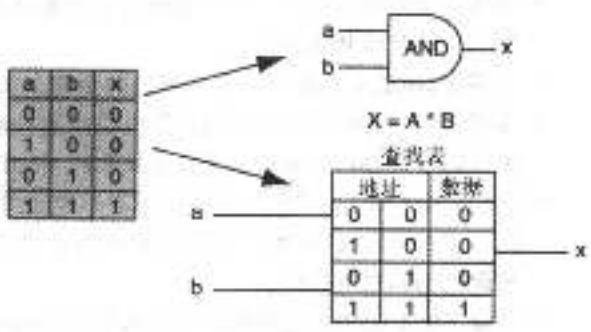


图16-5 将逻辑方程实现为门或查找表

易地改变，就像对器件重编程一样。

FPGA的引入对ASIC的设计方式有深远的影响。虽然FPGA比定制的ASIC昂贵得多，又不能用时钟同步得那样快，而且门容量很低，可是，由于它具有建造运行得和成品几乎一样快的工作原型硬件的能力，所以极大地降低了ASIC设计的固有风险，大大地缩小了基于ASIC产品所需的开发时间。

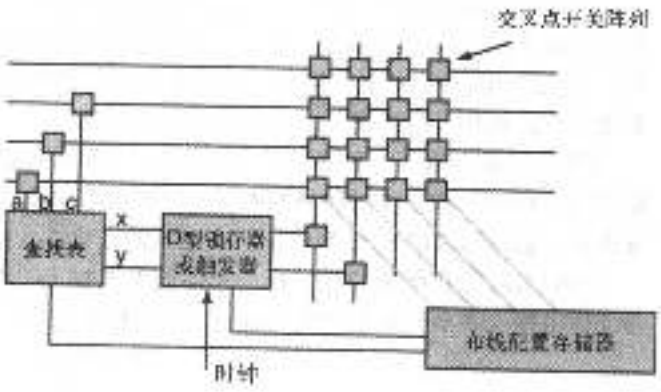


图16-6 一部分现场可编程门阵列的示意图

目前，像Xilinx[2]和Actel[3]这样的公司都提供具有相当于几百万个等价门的FPGA。例如，Xilinx XC3S5000包含5百万个等价门，并且在封装中有784个用户I/O，总共有1156个引脚。其他版本的FPGA包含片上RAM和乘法单元。

随着FPGA的流行，相应的软件支撑工具也成长了起来。对于商业上可得到的FPGA，ABEL、Verilog及VHDL可编译到配置图中。像ARM、MIPS及PowerPC这样的系列微处理器都已经端口化，所以可以直接插入到商业FPGA中[4]。

随着FPGA作为独立器件和原型化工具的流行，研究者和商业创业者正在建造由数百或数千个互连的FPGA排列而成的可重构数字平台。瞄准这些大型系统的公司是Intel和AMD这样的公司，它们有经济实力和建造复杂微处理器的业务。由于将一种新的微处理器设计推向市场所具有的高投入和高风险，那些使设计者能将其设计的模拟加载到硬件加速器上并实际运行适当时间的系统就成了非常重要的工具。

通过以解释模式运行Verilog或VHDL设计文件的方式，完全用软件模拟一个微处理器就成为可能，这时在每秒中，模拟可能只在被模拟时钟周期的10或100的整数倍处运行。想像你试图要在一个以5KHz运行的计算机上启动操作系统。在美国，有两个互相竞争的企业Quickturn和PiE生产和销售大型的可重构硬件加速器（hardware accelerator），硬件加速器这个术语是指这些机器在相关方面的使用，不是试图用软件来模拟像微处理器这样的复杂数字设计，而是将设计装载到硬件加速器中并以接近1MHz的速度运行。

Quickturn和PiE后来合并了，然后又卖给了加州San Jose的Cadence Design Systems。Cadence是世界上最大的电子设计自动化（EDA）工具的供应商。

当这些硬件加速器最初出现时，其成本大约是每个等价门1美元。Quickturn的最早的主要商业应用之一就是Intel通过将若干个硬件加速器连接成一个大型系统来完成对整个奔腾处理器的模拟。他们能够在几十分钟内将系统引导到DOS提示符下[5]。

当商业部门正将其注意力集中在硬件加速器的时候，其他研究者正在为可重构计算机的可能应用进行实验。那时我作为其中一成员的HP公司实验室的一个研究组正在建造一个定制的可重构计算机，用于进行研究和硬件加速两种用途[6]。HP机器的不同之处在于其采用的新颖的用于设计布线的方法使其能在几百个互连的FPGA中进行适当的划分。

所有FPGA必须要处理的最大问题之一就是最大化片上资源的利用率，该利用率受限于可使用的布线资源。只有50%的可使用逻辑资源能够在FPGA内布线是很平常的事情，需要非常复杂的布线算法来处理所有的细节问题。一个单FPGA的布线耗费几个小时是很平常的事，而像可重构计算机这样的复杂设计，即使是布线算法分布在一个UNIX工作站集群上，也需要几

周的布线时间。

产生这种状况有几个原因，但主要原因是商业上可得到的FPGA没有设计成互连阵列，而是设计成了独立的ASIC原型工具，它们的逻辑资源富足但布线资源有限。其原因很容易理解，在一个大的FPGA阵列中，信号可能需要穿过某些FPGA，其目的只是到达其他FPGA。这样，即使没有使用片上逻辑资源，也消耗了布线资源。

HP的团队采取了一种不同的途径。从一开始，他们就确信要始终服从Rent规则。Rent规则在20世纪60年代首先被IBM的Richard Rent所阐明。他将结果写在一个内部备忘录中，从未发表过，虽然后来的研究者[7]验证了该结果的正确性。

Rent当时正在研究的是：在一个印刷电路板上，组件的数量以及它们之间互连线的数量与支撑电路所要求的输入和输出数量之间有什么关系。Rent能用数学方程表达这种关系：

$$N = k^* G^e$$

其中：

N=进入任意部分或电路器件群组的I/O信号数量。

k=每个器件的平均I/O数量。

G=在该部分中器件的数量。

e=在0.5≤e≤0.7范围内的指数。

图16-7用只有几个门的简单情况对Rent规则作了图解。

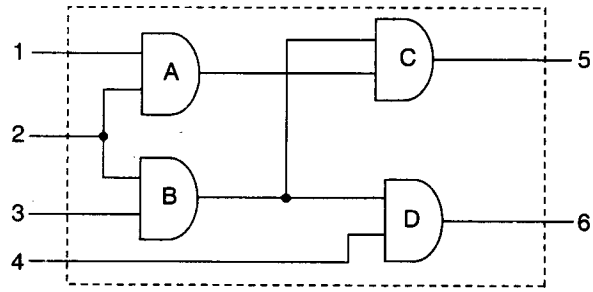


图16-7 Rent规则

这里我们看到有6个I/O引脚进入该部分。每个门有3个I/O引脚，而该部分内有4个门。取e＝0.5时，用Rent规则为该部分建模是正确的。指数值可在一个小范围内变化，这是因为不同的电路种类（比如非常规整的存储器阵列）所导致的结果与随机逻辑阵列不同。

Rent规则可用于预测支撑任意器件群组所必需的布线（I/O）数量。为设计一个遵循Rent规则的可重构计算机，HP设计了一个称为Plasma芯片的定制FPGA。Plasma是“Programmable Logic And Switch Matrix”[8]的缩写词。Plasma芯片的版图示意图由图16-8给出。

每个PLASMA芯片包含16组，每组16个可编程原子逻辑元件（Programmable Atomic Logic Element, PALE）。16个PALE组成的组带有自己的称为Hextant的交叉互连接。每个PALE由一个6-输入、2-输出的查找表组成，后接一个D型存储寄存器。每个hextant都馈入一个大的中心交叉矩阵。

427

中心交叉矩阵只被占用了四分之一，这意味着只有1/4的主信号线与给定的

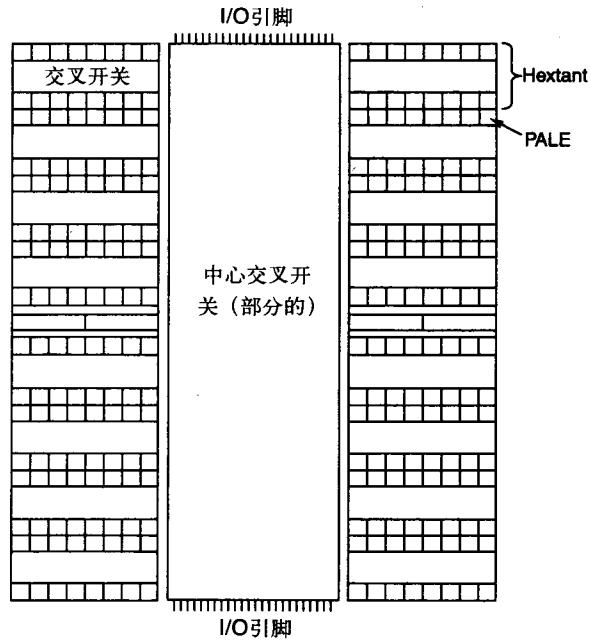


图16-8 PLASMA芯片的示意图

hextant信号线相连，这样做是出于空间的考虑。就这样，中心交叉矩阵包含400条垂直线和1600条来自hextant的线，Rent规则总是会被遵守，因为这样可以确保电路上不产生无法布通线的情况。图16-9显示了一个PLASMA芯片的照片，照片的显著之处在于布线资源和逻辑资源的相对大小，这是为了强调Rent规则在为广义的可重构计算机提供环境模型方面的重要性。在前面的章节中，我们讨论了如何用总线承载信息在机器中传输，这里，我们看一下为了指引数据在机器内任意布线，总线结构的大小为何必须要增长。

HP团队制造的可重构计算机称为Teramac，Teramac每秒执行10^{12}次门操作（在1MHz频率下有1百万门在进行开关），因此用"Tera"做前缀。由于是可重构的，故称为多体系结构计算机（mac）。

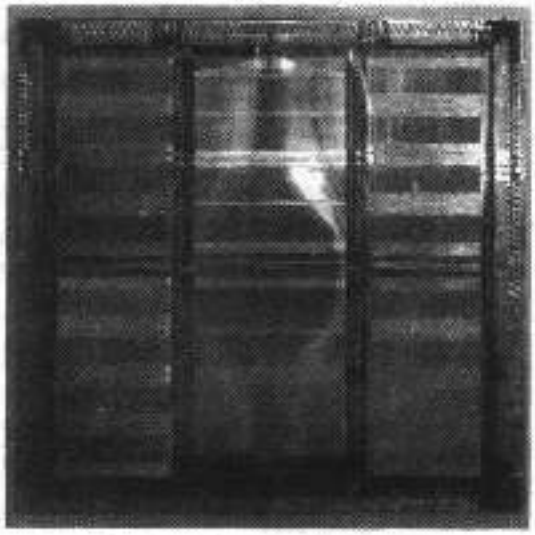


图16-9 PLASMA芯片的照片

Teramac包含1728个Plasma芯片，排列成复杂的网络。通过遵循Rent规则，任何Plasma芯片上的任何PALE都可被连接到其他芯片的PALE上。

Teramac的另一个方面是独特的，可能与本章关于计算的未来问题更相关。Teramac是容缺陷的。Plasma芯片或它们之间的互连可能存在缺陷，但机器仍然能工作。在某种程度上，容缺陷设计的体系结构与你硬盘驱动器的类似。很少有磁盘是完美的，每个表面或盘都可能包含小缺陷，从而致使一个或多个磁位不可读。在对磁盘进行低级格式化过程中缺陷区域会被标记出来，所以不用将整个磁盘扔掉，磁盘的一个特殊区域存储了标记信息，使得坏区域不会被使用。

目前，微处理器成品率的典型测量值低于25%，这意味着圆片上的芯片有75%是有缺陷的。原因是现代微处理器为保证能工作就必须是完美的，Teramac体系结构就没有这种限制，在测试过程中，在机器上运行具有某种特性的软件。当结果返回时，测试电路不能正确运行的区域将被映射出设计数据库。电路采用伪随机数生成电路来测试，电路包含一个由D型触发器组成的32级移位寄存器。移位寄存器的每一级馈入一个3输入XOR门，XOR门的输出作为输出馈入下一级。每个XOR门余下的两个输入随机地连接到其他31级的Q输出。

采用这个电路可产生长串的伪随机数。如果在电路中检测到故障，则随机数串将与期望结果有重大的偏离。这种测试称为标记分析（signature analysis），也用于在印刷电路板上检测开路和短路。为了检测一个缺陷，资源被分组成一序列伪随机数生成器并进行测试。如果检测到缺陷，故障电路的每个元件将被再一次当作垂直测试电路的一部分进行独立的测试。缺陷源存在于两个故障测试的交集，见图16-10。

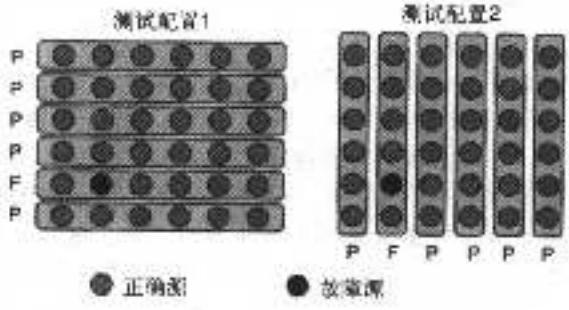


图16-10 Teramac系统的缺陷检测方案

在Plasma芯片本身内部，只有7%的芯片区域被看作是关键的，这意味着在这个

428

区域内的一个或多个缺陷将导致电路无法使用。这意味着在Plasma芯片上或芯片之间还可能存在大量其他的缺陷，但系统却仍能工作。然而，在该机器上测试系统缺陷，仅测试16个电路板中的一个就要用去一周多的时间。

现在，我们最终得到了所有以前讨论的原因。在很多情况下，我们将算法的硬件解决方案和软件解决方案之间看作是平行的关系。硬件可以很容易地将特定的算法加速一千倍或更高。想想你视频卡上的图形处理器芯片，想像一下如果CPU自己计算所有的图像转换，枪战游戏怎么玩。

硬件由于能并行使用专用资源而加速了算法解。典型的情况是算法需要多少硬件资源，就使用多少。如果你需要处理一个1024位的图像向量，那么就可以使用一个1024位的寄存器组。

事实上，假如设计了适合的算法并加载到Teramac中，有人估计[10]，像Teramac这样的机
429 器在1MHz频率运行时，其执行DNA串匹配算法的速度要比最快的通用超级计算机快1000倍。像Teramac这样的机器的另一个应用领域是数据加密和安全。破解密码和产生不可破解代码都需要未来计算机具有执行大规模并行计算的能力。例如，目前最快的计算机需要几十亿年才能对一个300位的密钥解码[12]。当然，目前研究者也在研究将大量传统计算机结合成并行网络这种途径。SETI计划（Search for Extraterrestrial Intelligence）就是这样一个例子。当你的PC空闲时你可以自愿将它出借给SETI，该计划将利用它分析来自射电望远镜的数据。

现在，想像一下在未来某个时间，我们的计算机由大量像FPGA阵列的自由资源组成。未来的编译器在编译目标代码的同时，也编译用来实现解决方案的最优体系结构。如果这样来设计硬件，即配置存储器可快速地用一个块存储器转移指令来加载，就有可能在每个函数调用时将硬件重配置成最优配置。

虽然这看起来牵强，但公司已经在建造这种体系结构的早期型式。加州Mountain View的Triscend公司[11]建造了可配置的片上系统平台。这些产品包含自由的逻辑和布线资源，这些资源环绕一个具有工业标准的微处理器核。将配置存储器映射到微处理器的存储空间，这样它就能按实现I/O设备（如以太网控制器、定时器、端口，等等）的需要对外设硬件进行重配置了。同样，自由的门可配置为算法加速器，如浮点单元、图形处理器之类。Triscend A7系列包含一个由门海环绕着的ARM7TDMI处理器核。

16.3 分子计算

计算机产业最令人惊异的就是摩尔定律的确实性。自从Intel创始人Gordon Moore在1965年首次提出以来[14]，摩尔定律在预测集成电路密度呈指数增长方面一直令人惊奇地准确。集成电路工艺技术的进步起到了推动作用。最初在1971年出现的Intel 4004处理器有2 250个晶体管，而2000年出现的奔腾4处理器有42 000 000个晶体管[15]。

器件物理学家已经在预言摩尔定律在不久的将来可能就会开始瓦解。这并不是说计算机就会停止在复杂性和处理能力上提高，而是对半导体器件持续无限制缩小的能力的表述。这些限制中有些是经济上的，有些与物理定律有关。

经济因素能轻易地促使在方向上的改变。每次集成电路制造工艺前进一步，设备的资金成本就相应提高。你可以说摩尔定律对于制造成本来说仍是成立的。目前，我们几乎处在使用光波让电路掩膜感光的能力的极限。随着集成电路的尺寸持续缩小，就有必要使用如同步
430 加速器这样的高能辐射源来产生硅圆片曝光所需要的那种“光”。

物理定律甚至更为苛刻。随着尺寸缩小，量子力学效应将变得更显著。我们已经利用了量子效应来为FLASH存储设备编程。通过对FLASH存储单元施加一个电压脉冲，量子力学效

应就使我们能穿过二氧化硅绝缘层“注入”电荷载体。当脉冲移走后，电荷就陷于绝缘层的另一侧。随着我们不断缩小尺寸，就需要越来越小的电压来使绝缘层降低，电荷逃逸的可能性就将增加，并导致电路不再可靠。

尺寸缩小所产生的另一个效应称为*电迁移*（electromigration）。在这里导体路径变得如此之小，以至于实际上电子流能在其扫过的金属导体中形成杂质。这股“电子风”将会把杂质移动到一个它们可以聚集的区域，并最终导致电路失效。

当然IC制造工程师和器件物理学家将继续研究这些制造问题，在未来几年我们将继续建造密度更高的电路。但是，显然我们不能持续地缩小电路元件的尺寸直至它们达到电子和亚原子粒子的尺寸。我们能吗？

Teramac以一种非常令人信服的方式显示出，采用“非完美”策略建造高度并行、可重构的计算机仍是可行的。让我们简要认识一下这种技术。

建造单个分子组成的阵列，以模仿目前的逻辑和其他电子电路组件行为是可能的，分子电子学就是基于这个前提。各种团队正在研究在分子中存储电荷和影响分子导电性的方法，使之很像目前MOS晶体管的行为。其他课题组已经用有机分子制造出基本的逻辑函数（AND门）。

该研究的商业可行性目前还难以估计（至少对于我是这样）。清楚的是这些“器件”的尺寸要比目前最小的集成电路至少小1000倍。当然，还需要满屋子非常昂贵的仪器才能使分子开关进行“开关”，但那只是因为我们看到的技术还处于婴儿期。回忆一下过去是明智的，最初的Eniac计算机有17 000个真空管，充满了宾夕法尼亚大学的一间屋子，在20KHz的时钟速率下运行时，一开机整个费城的灯就都变暗了。

要实现分子计算机还有很多巨大的障碍必须克服。例如，什么是线？你如何将分子互连使得电流能容易地在它们之间流动？另一个问题，你如何检测这些电路中微小的信号？目前使用的满屋子的强大分析工具不适合在宿舍中用，在宿舍中适用的是简便的膝上型电脑。

这样，基本问题就归结为寻找与这些基本电路元件等价的分子元件：[16]

- 开关器件：与晶体管等价的分子器件。
- 存储单元。
- 互连技术。
- 信号放大。

431

16.4 局部时钟

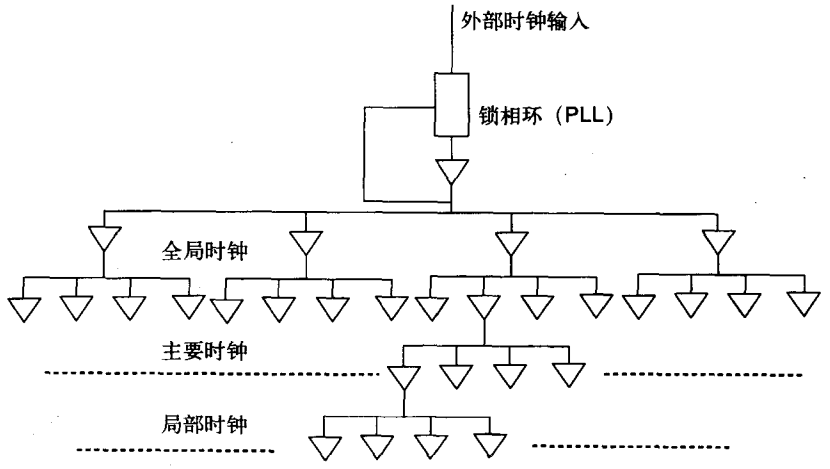
我想讨论的最后一个未来趋势就是局部时钟的概念。在观察这个现象之前，我们应该花一些时间探究一下我们要解决的问题。首先让我们界定一下这个问题。在写这本书的时候（2004年8月），最快的微处理器时钟频率将近3.5GHz。预测我们将会在第二年左右轻易地达到5GHz，达到10GHz也是不远的事情。

5GHz的时钟速率对应于200皮秒（ps）的时钟周期。由于在自由空间中光速大约是每纳秒12英寸，而通过导线的速度大约是每纳秒6英寸，所以这意味着在200皮秒时间光能传播大约1.2英寸。现代微处理器每边大约是3/4英寸，这意味着62%的时钟周期将浪费在只是使时钟信号从一边传播到另一边。由于微处理器是全同步的机器，这就是一个非常严重的问题。我们将该问题称为*时钟偏差*（clock skew）。时钟偏差就是时钟各相应部分之间在时间上的差异（相差），这是由时钟向芯片所有部分同时散布所引起的问题。在Teramac中，时钟偏差是一个

主要的设计问题，必须在机器设计的所有元件中作为考虑因素。还有，最初的Cray超级计算机控制时钟偏差的办法是通过调节同轴电缆的长度，这些电缆会将时钟运载到机器上的不同电路板。

另一个潜在的问题是所有的晶体管并不是精确地以相同的方式进行开关。芯片不同部分的时钟电路的开关特性会存在细微差别。测量显示这些在开关特性上的差别会大到约180ps[17]。这样，随着芯片变得越来越大、越来越快，我们使时钟在芯片上均匀分布的能力就更成问题了。

目前，大多数时钟的分布网络是层次化的。图16-11显示出一个典型的时钟分布网络。标记为*锁相环*（phase-locked loop）的电路模块表示的是现代计算机将内部时钟频率乘倍到大于外部时钟输入所采用的方法。例如，如果外部时钟频率是200MHz，你在BIOS中设定或锁在芯片中的乘因子是11。那么，内部时钟频率就是2200MHz，即2.2GHz。你可能会看到，IC制造过程参数的一个简单变化就会导致时钟偏差问题，因为时钟要分布到芯片上所有的同步电路中。



432

图16-11 同步时钟分布网络

回顾可知，现代处理器是流水线驱动的器件，在流水线的不同级内不同的组合逻辑电路在发挥作用。所有的级都由同一个同步时钟驱动，如图16-12所示。这里我们就可领会为什么限制时钟偏差是如此的关键。流水线的每一级必须在时钟到来之前完成它的工作，以将结果锁存到流水线的下一级。流水线每级的组合逻辑就取决于它在下一个时钟沿到来之前完成工作所需要的时间预算。时钟沿的偏差意味着流水线的某些级被时钟触发的时间比其他级早，破坏了流水线的同步性。

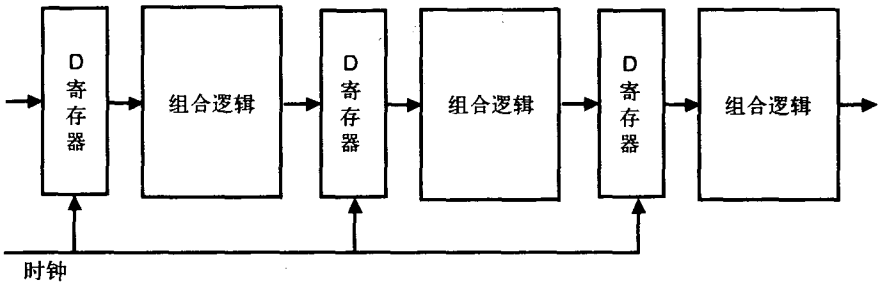


图16-12 带有同步时钟的流水线

现在，让我们稍微修改一下体系结构，允许每个组合模块按其自己的步调运行。图16-13给出了异步时钟触发流水线的示意图。

在流水线的每一级，系统时钟用于驱动局部时钟控制器。然而，由于流水线的每一级都是自主的，所以它的局部时钟既不用前一级的时钟来同步，也不用后一级的时钟来同步。

当某个级的组合逻辑完成了其工作时，该级的逻辑就会向局部时钟控制器输出一个请求，将结果锁存到馈入下一级的D寄存器。当数据锁存到下一级的输入寄存器时，局部时钟控制器就会向下一级发出一个确认信号，指示数据有效并可以使用。纯粹的效果就是我们建立了一个在级间进行握手控制（handshake control）的流水线。每一级必须请求数据传输，而锁存机制用传输确认向下一级作出响应。

这个方案的缺点是：因为局部时钟不是同步的，所以握手控制就可能错过时钟沿，数据就可能不得不等待另一个时钟才能传送到下一级。由于每一级要等待前一级完成，流水线中的这种延迟就很容易反向传播并使流水操作拖延。然而，当我们要求处理器以超过10GHz的时钟速度运行时，这个方案的优势可能就远大于其缺点。假定我们还能建造能够以如此高的时钟速率运行的数字逻辑电路，那么局部时钟也许是唯一的解决方案。

433

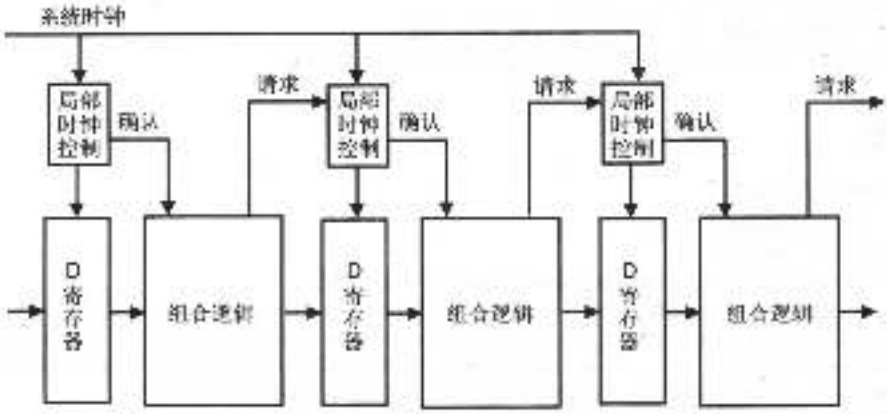


图16-13 带有异步时钟体系结构的流水线

这就提出一个有趣的问题，“为什么还要采用时钟？”我们能建立一个完全异步（无时钟）的计算机吗？根据Marculescu等人的意见[17]，全异步设计可能还有很长的路要走。用于现代处理器设计和验证的计算机辅助设计（CAD）工具还没有能力来处理全异步的设计。此外，还有一个惰性的问题，我们就是不用这种方法来设计计算机。然而，局部时钟仍是解决时钟偏差问题的一个可行的折中方案。

已经有几个新创立的公司要利用全异步微处理器设计的思想了。Fulcrum Microsystems[18]就源于加州理工学院的工作。图16-14说明了异步处理器的一个潜在优势。

对于异步系统，流水线中的数据以其自身的速率流动，需要额外的电路来防止失控情况，而在传统的时钟驱动的微处理器系统中是采用时钟和寄存器来防止失控的。

这个概念类似于局部时钟的使用，但在这种情况中，需要额外的逻辑来检测什么时候某一级

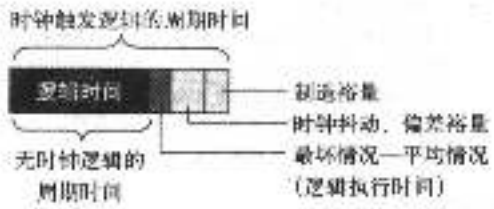


图16-14 无时钟逻辑相对于传统时钟触发逻辑的优势。来源于Fulcrum Microsystems

完成了其工作，以使流水线的下一级能够工作。图16-15显示了这种情况。

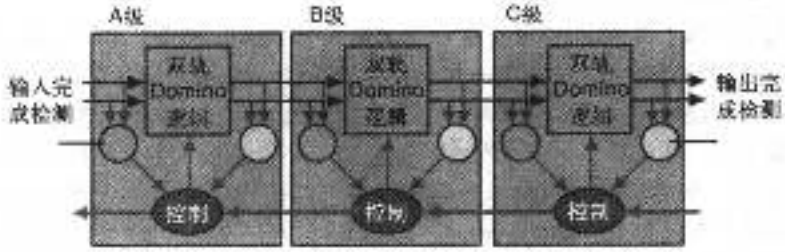


图16-15 无时钟流水线。来源于Fulcrum Microsystems

总结

在第16章，我们讲述了：

- 可编程逻辑器件的体系结构。
- 现场可编程门阵列的体系结构。
- 基于现场可编程门阵列来开发可重构计算机。
- 分子计算、局部时钟及无时钟计算机的未来发展趋势。

参考文献

[1] http://www.datio.com.

[2] http://www.xilinx.com.

[3] http://www.actel.com.

[4] http://www.xilinx.com/company/press/kits/v2pro/backgrounder.pdf.

[5] "Inside Intel: It's Moving at Double-Time to Head Off Competitors," Business Week, June 1, 1992.

[6] Greg Snider, Philip Kuekes, W. Bruce Culbertson, Richard J. Carter, Arnold S. Berger, Rick Amerson, *The Teramac Configurable Computer Engine*, Proceedings of the 5th International Workshop on Field-Programmable Logic and Applications, edited by Will Moore and Wayne Luk, Oxford, UK, September 1995, p. 44.

[7] B.S. Landman and R.L. Russo, *IEEE Trans. Comp., C20*, 1469, 1971.

[8] Rick Anderson, Richard J. Carter, W. Bruce Culbertson, Philip Kuekes, Greg Snider, Lyle Albertson: *Plasma: An FPGA for Million Gate Systems*. FPGA '96. Proceedings of the 1996 Fourth International Symposium on Field Programmable Gate Arrays, February 11-13, 1996, Monterey, CA, USA. ACM, 1996, pp. 10–16.

[9] B. Culbertson, R. Amerson, R. Carter, P. Kuekes, G. Snider, *The Teramac Custom Computer: Extending the limits with defect tolerance*, IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, November 1996.

[10] Barry Shakleford, HP Labs, Private Communication.

[11] http://www.riscend.com.

[12] Daniel Tynan, "Silicon is Slow," Popular Science, June, 2002, p. 25.

[13] http://setiathome.ssl.berkeley.edu/.

[14] Gordon E. Moore, *Cramming More Components onto Integrated Circuits*, *Electronics*, Volume 28, Number 8, April 19, 1965.

[15] http://www.intel.com/research/silicon/mooreslaw.htm.

[16] Mark A. Reed and James M. Tour, *Computing with Molecules*, *Scientific American*, June, 2000, p. 89.

[17] Diana Marculescu, Dave Albonesi, Alper Buyuktosunoglu, *Tutorial: Partially Asynchronous Microprocessors*, *Micro-35*, Istanbul, Turkey, Nov. 18, 2002.

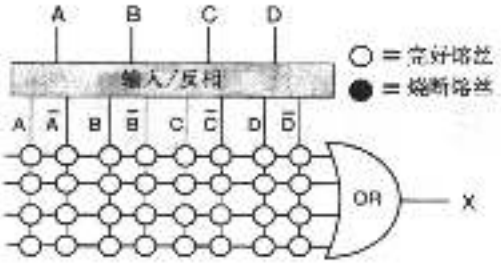
[18] http://www.fulcrummicro.com.

习题

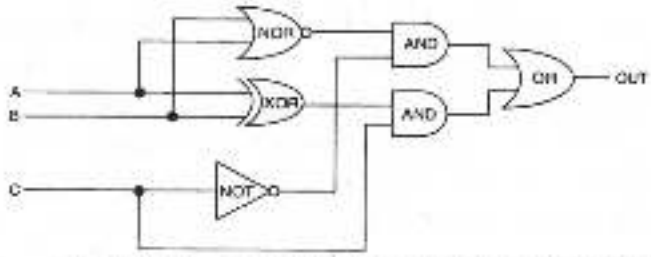
1. 考虑下图所示的一部分PLD电路。熔断的熔丝用黑实体表示，连接的熔丝用空白圆表示。复

制一份该图，通过填充你要烧断的表示互连的圆来对器件进行“编程”。编程实现逻辑方程：

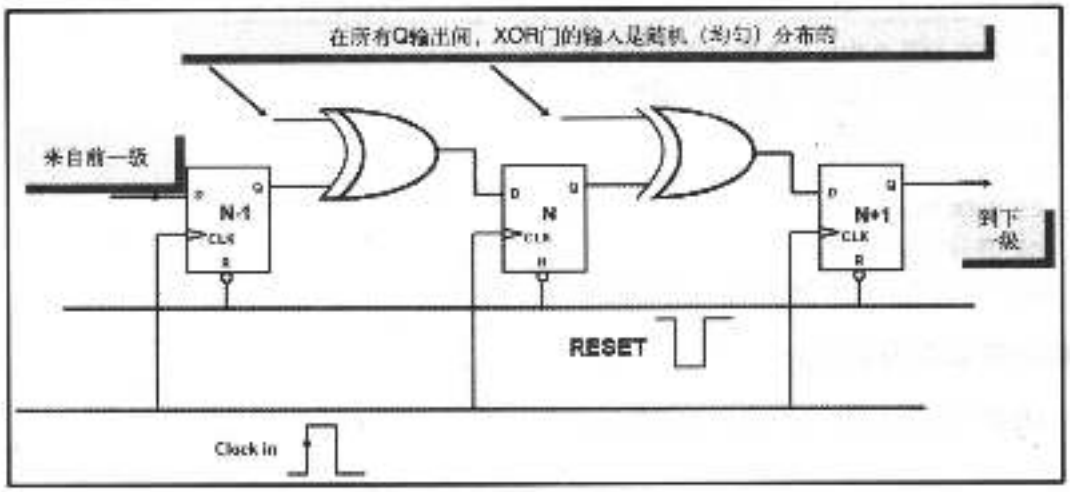
$$X = (A \oplus B) + C^* \overline{D}$$



2. 下图所示的电路服从Rent规则吗？



3. 类似下图显示的电路，包含16~32级，在容缺陷的计算机器中用于检测有缺陷的互连元件或有缺陷的逻辑元件。为什么这个电路是这种任务的特别好的选择？



4. 假设你想设计一个具有10GHz时钟速率的同步CPU。最坏情况下通过逻辑门的传播延迟是28皮秒。流水线的每一级都有不超过3级的逻辑电路。考虑到制造的不确定性、器件启动时间以及电路中器件开关特性的差异，你还需要保持10皮秒的安全裕度。这个设计所能容忍的时钟路径长度的最大差异大约是多少？ 436

附录　奇数号习题答案

第1章奇数号习题答案

1. 摩尔定律说，在一个集成电路硅模上的晶体管数量大约每18个月加倍。由于电路设计者能在单块硅模上放置的晶体管数量持续地增加，这意味着所使用的计算机和存储器的复杂性也在增加。而且，由于晶体管数量在增加，晶体管尺寸在减小，所以晶体管被更紧密地安置，电信号的传输距离也在下降，这意味着电路能更快速地运行。

   于是，存在两种效应。计算机能在诸如总线宽度和复杂性等方面取得更高的性能，因为我们可以利用放置于单个硅模上的电路数量这个优势，而且，这些复杂设计可以更快地运行。最后，复杂电路设计甚至允许运行更复杂的软件应用程序，因为我们有更快速、更大容量的存储器来实现算法。

3. 抽象层次概念的一个优点是：你能够隐藏较低层次的细节和差异，使得在较高层次的程序只需编写一次就能在很大范围的不同机器上运行。一个缺点是：在你调用较低层次的函数时，若必须穿过不同的层次并在每一步做转换，那么就可能会损失效率。

5. 半导体存储器平均比硬盘驱动器快342 857倍。

7. 将下列的十六进制数转换成十进制数：
   (a) 0xFE57 = 65 111
   (b) 0xA3011 = 667 665
   (c) 0xDE01 = 56 833
   (d) 0x3AB2 = 15 026

9. 每秒7.64微英尺，即每秒7.64×10^{-3}英尺。

第2章奇数号习题答案

1. 与电路变成或电路，而或电路变成与电路。

3.

Part a

a	b	c	F
0	0	0	0
1	0	0	1
0	1	0	1
1	1	0	0
0	0	1	0
1	0	1	0
0	1	1	1
1	1	1	0

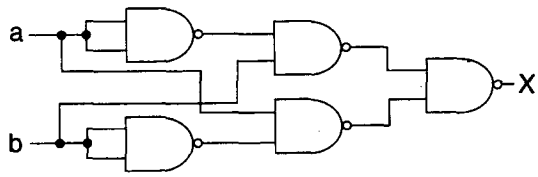
Part b

a	b	c	F
0	0	0	0
1	0	0	0
0	1	0	0
1	1	0	1
0	0	1	1
1	0	1	0
0	1	1	1
1	1	1	1

5. 真值表如图所示：

a	b	c	d	X
0	0	0	0	1
1	0	0	0	0
0	1	0	0	0
1	1	0	0	1
0	0	1	0	0
1	0	1	0	1
0	1	1	0	1
1	1	1	0	0
0	0	0	1	0
1	0	0	1	1
0	1	0	1	1
1	1	0	1	0
0	0	1	1	1
1	0	1	1	0
0	1	1	1	0
1	1	1	1	1

7. 电路如图所示：



第3章奇数号习题答案

1. 真值表和卡诺图如下图所示：

A	B	Cin	SUM	Cout
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

SUM的卡诺图

	*A*B	*AB	AB	A*B
Cin	1		1	
*Cin		1		1

Cout的卡诺图

	*A*B	*AB	AB	A*B
Cin		1	1	1
*Cin			1	

$$SUM = \overline{A} * \overline{B} * Cin + A * B * Cin + \overline{A} * B * \overline{C}in + A * \overline{B} * \overline{C}in$$

$$SUM = Cin * (\overline{A} * \overline{B} + A * B) + \overline{C}in * (\overline{A} * B + A * \overline{B})$$

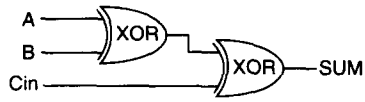
注意到公式 $\overline{A} * B + A * \overline{B}$ 正好表示的是异或门（XOR），我们可以对第二项进行简化。此外，第一项 $\overline{A} * \overline{B} + A * B$ 正好是异或函数的补，所以实际上是两个嵌套的XOR项：

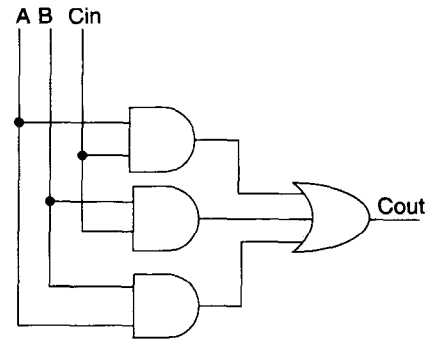
$$SUM = Cin \oplus [A \oplus B]$$

我们可以用卡诺图来简化Cout的逻辑，一共有三个圈，对应的表达式为：

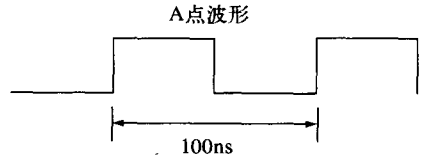
$$Cout = B*Cin + A*Cin + A*B$$

下面是SUM和Cout的逻辑电路实现。





3. 假设T = 0时逻辑电平由0变为1，如下图所示。我们可以看到，传递这个信号的变化经过一个门就增加10ns的延迟。当经过50ns后信号到达A点，它将相反极性的信号输入到第一个门，后续的信号序列的极性将完全相反。在T = 100ns的时刻，情况与T = 0时完全相同，只是过去了100ns的时间。所以，电路以100ns的周期震荡，点A处信号的频率是10MHz。



从A点处看到的波形为：

5. 真值表和卡诺图如下所示：

真值表

A	B	C	D	X	Y	Z
0	0	0	0	0	0	0
1	0	0	0	1	0	0
0	1	0	0	0	1	0
1	1	0	0	0	1	0
0	0	1	0	1	1	0
1	0	1	0	1	1	0
0	1	1	0	1	1	0
1	1	1	0	1	1	0
0	0	0	1	0	0	1
1	0	0	1	0	0	1
0	1	0	1	0	0	1
1	1	0	1	0	0	1
0	0	1	1	0	0	1
1	0	1	1	0	0	1
0	1	1	1	0	0	1
1	1	1	1	0	0	1

X的卡诺图

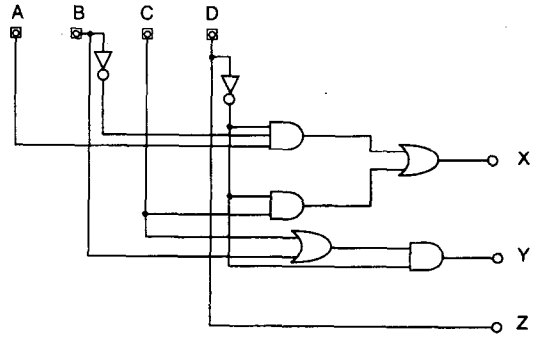
	$\bar{A}\bar{B}$	$A\bar{B}$	AB	$\bar{A}B$
$\bar{C}\bar{D}$		1		
$\bar{C}D$	1	1	1	1
$C\bar{D}$				
CD				

Y的卡诺图

	$\bar{A}\bar{B}$	$A\bar{B}$	AB	$\bar{A}B$
$\bar{C}\bar{D}$			1	1
$\bar{C}D$	1	1	1	1
$C\bar{D}$				
CD				

Z的卡诺图

	$\bar{A}\bar{B}$	$A\bar{B}$	AB	$\bar{A}B$
$\bar{C}\bar{D}$				
$\bar{C}D$				
$C\bar{D}$	1	1	1	1
CD	1	1	1	1

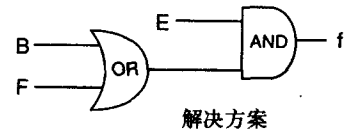


被简化的方程为：

$$X = A * \overline{B} * \overline{D} + C * \overline{D}$$
$$Y = C * \overline{D} + B * \overline{D} = \overline{D} * (C + B)$$
$$Z = D$$

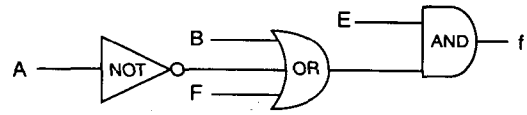
7. 让我们看一看解决方案的逻辑。有关泵的马达的设计逻辑可以是，当温度太低时，泵不会自动开启马达和加热器。另一种可能的理解是，当温度很低时马达和加热器会自动开启。泵的控制电路应该体现这两种选择。

a. 泵的马达：当计时器（B）开启，或者手动开关（F）开启并且钥匙开关（E）打开时，泵的马达打开（f = 1）。注意在另一种解决方案里温度变低也能使泵开启，因此我们在电路中加了一项考虑这个情况。



解决方案

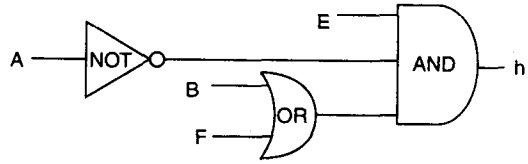
b. 加热器：当温度传感器（A）指示水温低于控制面板上设定的温度时，加热器应该打开（h = 1）。我们还有更实际的考虑，就是除非泵也打开了，否则加热器就不用开。如果加热水时水不流动是很危险的。这个设计体现在加热器h的控制电路图中。



另一种解决方案

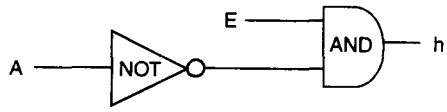
这样，在上面的电路中必须具备三个条件才能使加热器打开：

1. 钥匙开关（E）打开。
2. 泵打开（B + F）。
3. 温度低（$\overline{A}$）。



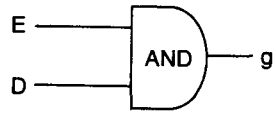
解决方案

按另一种解决方案能得到更简单的安排。要打开加热器仅需要钥匙开关和低温这两个条件。我们不必担心泵，因为 $\overline{A}$ 为真时它就打开了。

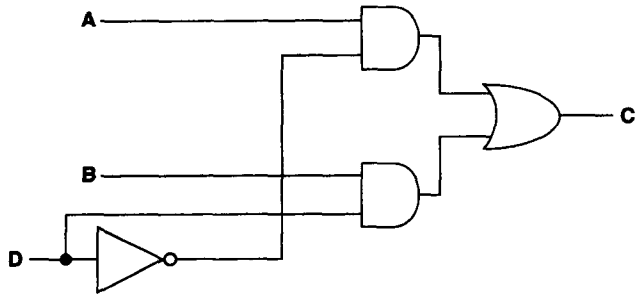


另一种解决方案

c. 吹风机：空气吹风机（g）的控制很简单，钥匙开关必须开（E = 1），同时吹风机开关也必须开（D = 1），这样就能在完成家庭作业辛苦了一天后享受温柔的气泡和暖风。这个控制电路如下所示：

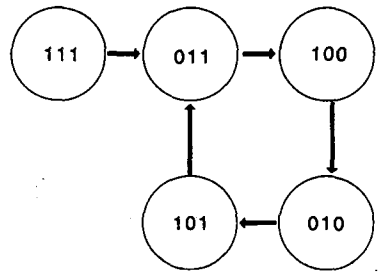


9. 电路如下图所示：



第4章寄数号习题答案

1. 下面是状态迁移图：



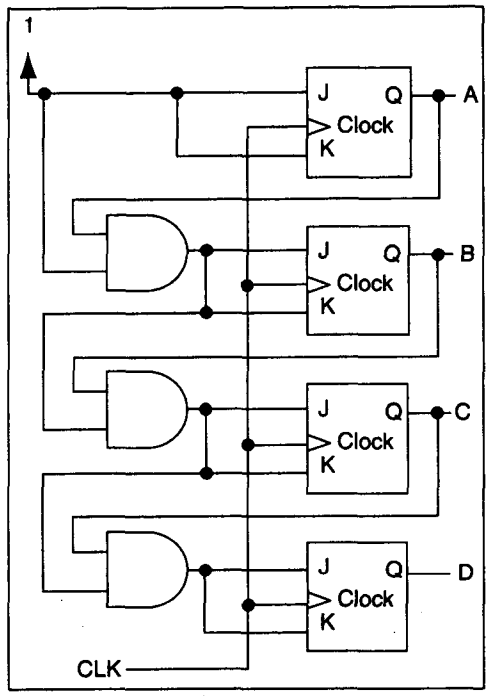
3. 真值表为：

时 钟 脉 冲	Qa	Qb
时钟脉冲到来前	0	0
第1个时钟脉冲后	1	0
第2个时钟脉冲后	1	1
第3个时钟脉冲后	0	1
第4个时钟脉冲后	0	0

5. 表格如下所示，经过6个时钟脉冲后工作模式发生重复。

	脉冲到来前				脉冲到来后			
	A	B	C	D	A	B	C	D
1	0	0	0	0	0	0	1	1
2	0	0	1	1	1	0	1	0
3	1	0	1	0	0	1	1	0
4	0	1	1	0	1	0	0	0
5	1	0	0	0	0	1	1	1
6	0	1	1	1	0	0	0	0
7	0	0	0	0	0	0	1	1
8	0	0	1	1	1	0	1	0

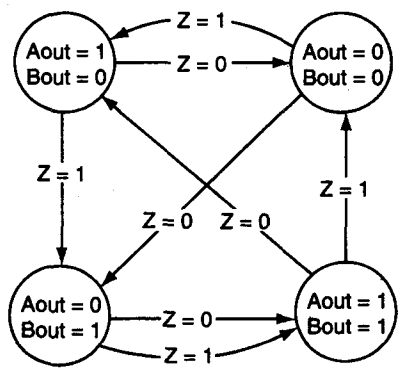
7. 同步计数电路如下所示：



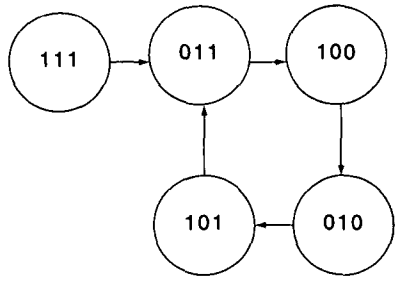
第5章奇数号习题答案

1. 真值表和状态图如下所示：

Ain	Bin	Z	Aout	Bout
0	0	0	0	1
1	0	0	0	0
0	1	0	1	1
1	1	0	1	0
0	0	1	1	0
1	0	1	0	1
0	1	1	1	1
1	1	1	0	0



3. 答案见下图：



5. 有S0到S3四个状态，所以需要X和Y两个变量，提供到寄存器的输出和两个真值表的输入。这样，我们就可做如下断言：

S0 → X = 0, Y = 0

S1 → X = 1, Y = 0

S2 → X = 0, Y = 1

S4 → X = 1, Y = 1

让我们首先按字分析该系统，可从填写真值表开始。假设系统处于S0且没有投入钱，系统就处在此状态，我们就可用下面的表项来描述：

a	b	x	y	X	Y	Z
0	0	0	0	0	0	0

现在，假设我们处于状态S0（S0 → X=0, Y=0），则可能性有：

1. 无钱投入，仍停留在S0。
2. 投入1角钱（a=0, b=1），迁移到状态S1。
3. 投入四分之一美元（a=1, b=0），迁移到状态S3。

我们可将这个条件表示如下：

a	b	x	y	X	Y	Z
0	0	0	0	0	0	0
0	1	0	0	1	0	0
1	0	0	0	1	1	0

现在，假设我们处于状态S1（S1 → X=1, Y=0），则可能性有：

1. 无钱投入，仍停留在S1。
2. 投入1角钱，迁移到状态S2。
3. 投入四分之一美元，返回到状态S0，并交付商品。

我们可将这表示为如下的条件：

a	b	x	y	X	Y	Z
0	0	1	0	1	0	0
0	1	1	0	0	1	0
1	0	1	0	0	0	1

现在，假设我们处于状态S2（S2 → X=0, Y=1），则可能性有：

1. 无钱投入，仍停留在S2。
2. 投入1角钱，迁移到状态S0，并交付商品。
3. 投入四分之一美元，返回到状态S0，并交付商品。

我们可将这表示为如下的条件：

a	b	x	y	X	Y	Z
0	0	0	1	0	1	0
0	1	0	1	0	0	1
1	0	0	1	0	0	1

现在，假设我们处于状态S3（S3 → X=1, Y=1），则可能性有：

1. 无钱投入，仍停留在S3。
2. 投入1角钱，迁移到状态S0，并交付商品。
3. 投入四分之一美元，返回到状态S0，并交付商品。

我们可将这表示为如下的条件：

a	b	x	y	X	Y	Z
0	0	1	1	1	1	0
0	1	1	1	0	0	1
1	0	1	1	0	0	1

这样我们就考虑了所有的可能性。现在就用这些已知的情况填写真值表：

	a	b	x	y	X	Y	Z
S0	0	0	0	0	0	0	0
S0	0	1	0	0	1	0	0
S0	1	0	0	0	1	1	0
S0	1	1	0	0	×	×	×
S1	0	0	1	0	1	0	0
S1	0	1	1	0	0	1	0
S1	1	0	1	0	0	0	1
S1	1	1	1	0	×	×	×
S2	0	0	0	1	0	1	0
S2	0	1	0	1	0	0	1
S2	1	0	0	1	0	0	1
S2	1	1	0	1	×	×	×
S3	0	0	1	1	1	1	0
S3	0	1	1	1	0	0	1
S3	1	0	1	1	0	0	1
S3	1	1	1	1	×	×	×

X表示“无关条件”。它们在实际操作中永远不会发生，所以我们将它们保留下来看看是否能帮助我们化简电路的K图。

状态变量X的K图如下所示：

	$\bar{a}*\bar{b}$	$\bar{a}*b$	$a*b$	$a*\bar{b}$
$\bar{x}*\bar{y}$		1	1	1
$\bar{x}*y$				
$x*y$	1			
$x*\bar{y}$	1			

我加入了灰色的项（$a*b*\bar{x}*\bar{y}$），因为它将方程化简了一位：

$$X = \bar{a}*\bar{b}*x + b*\bar{x}*\bar{y} + a*\bar{x}*\bar{y}$$

状态变量Y的K图显示如下：

	$\bar{a}*\bar{b}$	$\bar{a}*b$	$a*b$	$a*\bar{b}$
$\bar{x}*\bar{y}$			1	1
$\bar{x}*y$	1			
$x*y$	1			
$x*\bar{y}$		1	1	

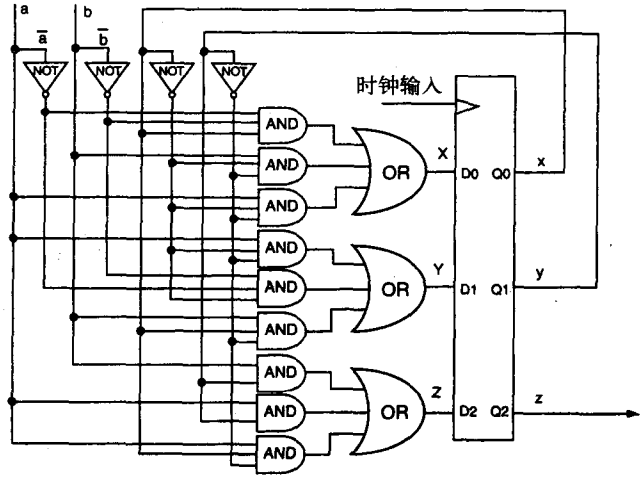
$$Y = a*\bar{x}*\bar{y} + \bar{a}*\bar{b}*y + b*x*\bar{y}$$

最后，输出变量Z的K图显示如下：

	$\bar{a}*\bar{b}$	$\bar{a}*b$	$a*b$	$a*\bar{b}$
$\bar{x}*\bar{y}$				
$\bar{x}*y$		1	1	1
$x*y$		1	1	1
$x*\bar{y}$			1	1

$$Z = b*y + a*y + a*x*\bar{y}$$

门级的图显示如下：

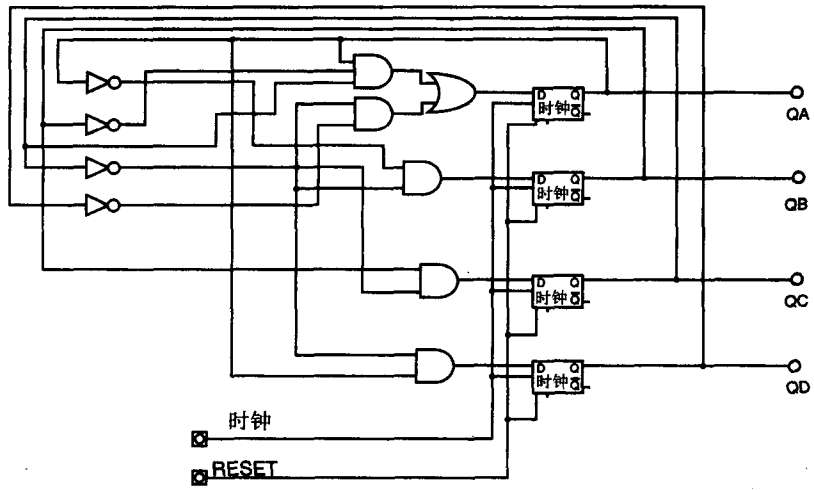


7. RESET过后，所有输出都是0，这就保证了机器从一个已知状态开始运行。在每个时钟脉冲

后，系统的状态显示在如下的表中：

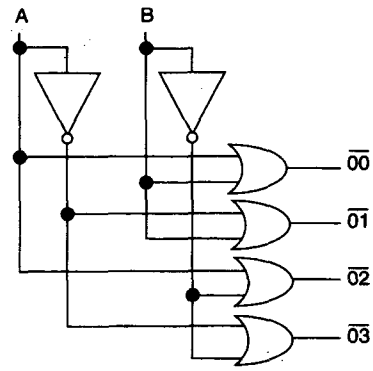
时钟	RESET	1	2	3	4	5	6	7	8	9	10	11	12	13	14
输出	0000	1100	1011	1000	1001	0001	0100	1110	0010	0101	0110	0111	1111	1010	0000

这样，14个时钟脉冲后状态开始重复。



第6章奇数号习题答案

1. 门设计如下所示。注意这很像一个从负逻辑转换成正逻辑的问题。



3a. 由于存储器宽是32位，我们需要4个存储器芯片来形成一个32位的页。我们有总共2^{26}个地址位。每个页是512K，需要2^{19}条地址线。这样，$2^{26}-2^{19}=2^{7}$，我们有128页存储器。由于每页需要4个器件，我们就需要总共512个存储器芯片。

3b.

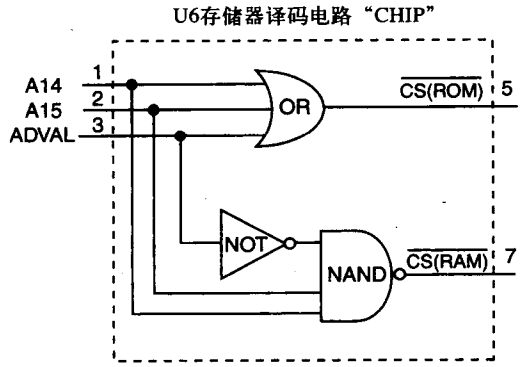
页号	起始地址（hex）	结束地址（hex）
0	0000000	007FFFF
1	0080000	00FFFFF
2	0100000	017FFFF

5a. 直接存储器访问：一种提高在外部设备和计算机存储器之间数据传输效率的方法。DMA过程允许外部设备在处理器空闲时掌握对存储器总线的控制，外设绕开处理器，直接处理对存储器的数据传输。

5b. 三态逻辑：一种电路设计，在存储器或其他器件上加入额外的输出控制，使得其输出在总线上捆绑在一起。当三态逻辑关闭器件的输出时，输出对总线呈现出高阻抗。换句话说，就好像它没有连接到总线，这就使另一个输出能驱动总线。

5c. 地址总线、数据总线、状态总线：这些是处理器的三个主要的总线。地址总线为存储系统提供下一个存储器操作的地址，它是单向的，所有信号都是从处理器向存储器输出。数据总线是双向的，数据流在同一总线上流入到处理器和流出到存储器。状态总线是异质的，一些信号只是输入，一些信号只是输出，其他是双向的。状态总线携带了处理器的所有事务处理信号。

7a. 存储器译码电路显示如下：



7b. 网表显示如下：

线网名					
addr0	U1-36	U2-1	U3-1	U4-1	U5-1
addr1	U1-35	U2-2	U3-2	U4-2	U5-2
addr2	U1-34	U2-3	U3-3	U4-3	U5-3
addr3	U1-33	U2-4	U3-4	U4-4	U5-4
addr4	U1-32	U2-5	U3-5	U4-5	U5-5
addr5	U1-31	U2-6	U3-6	U4-6	U5-6
addr6	U1-30	U2-7	U3-7	U4-7	U5-7
addr7	U1-29	U2-8	U3-8	U4-8	U5-8
addr8	U1-28	U2-9	U3-9	U4-9	U5-9
addr9	U1-27	U2-10	U3-10	U4-10	U5-10
addr10	U1-26	U2-11	U3-11	U4-11	U5-11
addr11	U1-25	U2-12	U3-12	U4-12	U5-12
addr12	U1-24	U2-13	U3-13	U4-13	U5-13
addr13	U1-23	U2-14	U3-14	U4-14	U5-14
addr14	U1-22	U6-1			
addr15	U1-21	U6-3			
data0	U1-1	U2-15	U4-15		
data1	U1-2	U2-16	U4-16		

（续）

线网名					
data2	U1-3	U2-17	U4-17		
data3	U1-4	U2-18	U4-18		
data4	U1-5	U2-19	U4-19		
data5	U1-6	U2-20	U4-20		
data6	U1-7	U2-21	U4-21		
data7	U1-8	U2-22	U4-22		
data8	U1-9	U3-15	U5-15		
data9	U1-13	U3-16	U5-16		
data10	U1-14	U3-17	U5-17		
data11	U1-15	U3-18	U5-18		
data12	U1-16	U3-19	U5-19		
data13	U1-17	U3-20	U5-20		
data14	U1-18	U3-21	U5-21		
data15	U1-19	U3-22	U5-22		
$\overline{ADVAL}$	U1-12	U6-3			
$\overline{WR}$	U1-10	U2-23	U3-23		
$\overline{RD}$	U1-11	U2-24	U3-24	U4-24	U5-24
$\overline{CSROM}$	U6-5	U4-25	U5-25		
$\overline{CSRAM}$	U6-7	U2-25	U3-25		

第7章奇数号习题答案

1. 大小不必一定匹配。有很多事例显示出内部存储器总线有的比外部总线小，有的大。例如，现代PC有64位宽的外部存储器总线，但目前的Athlon和奔腾处理器有32位宽内部总线。

在对成本敏感的系统中，为了使设计者能够建立一个更经济的系统，外部总线可能要比内部数据总线窄。然而，窄的存储器宽度意味着必须进行更多次的存储器取数操作，总体性能将会下降。

3a. MOVE.W $1000, A3：你必须使用MOVEA.W来装入一个地址寄存器。

3b. ADD.B D0, #$A369：目的操作数不能是一个文字。

3c. ORI.W #$55AA007C, D4：操作的尺寸在本例中是字，它必须与文字源操作数的大小匹配。

3d. MOVEA.L D6, A8：没有地址寄存器A8。

3e. MOVE.L $1200F7, D3：这是一个非对齐访问错误。对于一个字或长字，访问源地址或目的地址必须在一个偶数字地址边界上。

5. $AA。

7a. MOVE.L D0, D7 这是合法的。

7b. MOVE.B D2, #$4A 非法：不能将一个值存储到一个文字中。

7c. MOVEA.B D3, A4 非法：不能将一个地址作为一个字节存储。

7d. MOVE.W A6, D8 非法：D8不是一个有效的寄存器。

7e. AND.L $4000, $55AA 非法：AND操作的源操作数和目的操作数中至少有一个必须是数据寄存器。

9. XOR指令的逻辑操作就是按位做“异或”，这样，任何一对的位，若全为1则结果为0；若一个为1一个为0，则结果为1。字FFFF的效果是使字AAAA成为5555的补，将其加1就得5556。

FFFF XOR AAAA = 5555 + 1 = 5556

11. <004000> = $4515

第8章奇数号习题答案

1.

```
*****************************************************************
*
* 子程序定时器：
*
* 该子程序从1到9之间的一个数开始倒计时，这个数由寄存器D0.B传入。
* 倒计时的速率是每两秒钟一个数字，使用一个定位在地址$00001002的500ms硬件
* 定时器。
* 7段码显示器定位在地址$00001000
* 程序中不进行任何错误检查
*
* 所有寄存器都在退出时恢复
* All registers used are save upon exit.
*****************************************************************
disp0       EQU   $3F          * 显示的位码
disp1       EQU   $06
disp2       EQU   $5B
disp3       EQU   $4F
disp4       EQU   $56
disp5       EQU   $6D
disp6       EQU   $7D
disp7       EQU   $07
disp8       EQU   $7F
disp9       EQU   $67
trigger     EQU   $10          * 这用来启动定时器
time_out    EQU   $01          * 这用来测试是否完成
display     EQU   $00001000    * 显示器的存储位置
delay       EQU   $00001002    * 定时器硬件的存储位置
                               * 代码从这里开始
timer       MOVEM.L    A0/A1/D1/D2/D3,-(SP)  * 在入口处保存寄存器
entry
  LEA              patterns,A0              * A0指向显示码
display
            MOVEA.L    #display,A1          * A1指向显示器
            MOVEA.L    #delay,A2            * A2指向时延电路
circuit
            CLR.L      D1              * D1是变址寄存器
            MOVE.B     D0,D1           * 得到变址值
loop1       MOVE.B     00(A0,D1),(A1)  * 发送码到显示器
            CMPI.B     #00,D1          * 是否已经D1 = 0?
            BEQ        return          * 是的，返回
            MOVE.B     #4,D2           * 建立反计时定时器

loop2       MOVE.B     #trigger,(A2)   * 定时器开始
loop3       MOVE.B     (A2),D3         * 得到状态
            ANDI.B     #time_out,D3    * 孤立出DB0
            BEQ        loop3           * 保持等待
            SUBQ.B #1,D2               * 递减D2
            BNE        loop2           * 返回
            SUBQ       #1,D1           * 指向下一个码
            BRA        loop1
return      MOVEM.L    (SP)+,D3/D2/D1/A1/A0  * 恢复寄存器
            RTS
patterns    DC.B
      disp0,disp1,disp2,disp3,disp4,disp5,disp6,disp7,disp8,disp9
```

3. **<D0> = $0000002A**

5. ROM是只读设备。最后一条指令MOVE.W D0, (A2)将对ROM进行一次写操作。这是不正确的。

7.

```
delay     equ         2000          * 2秒延迟
mask      equ         $8000         * 定时器状态
bits      equ         01            * 位码
io_port   equ         $4000         * I/O端口的位置
timer     equ         $8000         * 定时器端口

          org   $400

start     move.b      #bits,io_port     * 载入IO端口
loop      move.w      #delay,timer      * 载入定时器
          move.b      io_port,d0        * 得到端口
          rol.b       #bits,d0          * 左移
          move.b      d0,io_port        * 送回
          move.w      #delay,timer      * 设置时延
wait      andi.w      #mask,timer       * 检查状态
          beq         wait              * 非零，保持等待
          bra         loop              * 重做
          end $400
```

9.

```
*************************************************************
*
* 这是一个将存储器用字码$5555填充的程序
*
*************************************************************
          OPT     CRE
fill_st   EQU     $00002000         * 填充块的开始
fill_end  EQU     $000020FF         * 填充的最后地址
pattern   EQU     $5555             * 填充码

start     EQU     $400              * 程序从这里开始

          ORG     start
          LEA     fill_st,A0        * 装载开始地址
          LEA     fill_end,A1       * 装载结束地址
          MOVE.W  #pattern,D0       * 装载将要写的字码
loop      MOVE.W      D0,(A0)+      * 移动它并递增指针值
          CMPA.L      A1,A0         * 我们是否已完成
          BLE         loop          * 没有？返回并重复
          STOP        #$2700        * 跳回到模拟器
          END         start
```

第9章奇数号习题答案

1. 这是一个“带变址和位移的寄存器间接寻址”的例子。有效地址是A0中的地址值、变址值D0以及位移的2补码三者之和。由于$84是一个负数（−7C），所以，有效地址EA = $2000 + $0400 + (-$7C)。

$$EA = \$2384$$

该程序不可重定位，原因有二：

1. 有一个到绝对地址start的跳转。
2. 绝对地址装入了A0。通过控制对A0和D0的装入值，程序仍可以是可重定位的，但

跳转指令强制使之成为了绝对地址。

3. 突出显示的指令执行后，D0中的值是$0000002A。

5.

```
00000400 067955550000AAAA          ADDI.W    #$5555,$0000AAAA
00000408 06B9AAAA55550000FFFE      ADDI.L    #$AAAA5555,$0000FFFE
00000412 0640AAAA                  ADDI.W    #$AAAA,D0
```

7.

```
*************************************************************************
*
* 可重定位的存储器检测程序
*
*************************************************************************

* 系统的等于伪指令

pattern1    EQU     $AAAA           * 第一个测试码
pattern2    EQU     $FFFF           * 第二个测试码
pattern3    EQU     $0001           * 第三个测试码
st_addr     EQU     $00000400       * 测试的起始地址
end_addr    EQU     $0009FFF0       * 测试的结束地址
stack       EQU     $000C0000       * 堆栈指针的位置
word        EQU     2               * 字的长度，即字节数
byte        EQU     1               * 一个字节长，不是什么神秘的数!
bit         EQU     1               * 移位
exit_pgm    EQU     $2700           * 模拟器退出码
data        EQU     $500            * 数据存储区域
start       EQU     $400            * 程序从这里开始
new_ad      EQU     $000A0000       * 重定位程序从这里运行
pr_cmd      EQU     00              * 打印消息命令

* 主程序

            OPT     CRE             * 打开交叉引用
            ORG     start           * 程序从这里开始
            LEA     stack,SP        * 初始化堆栈指针
            LEA     relo,A0         * 起始地址指针
            LEA     last_addr,A1    * 结束指针
            LEA     new_ad,A3       * 目的
relo_lp     MOVE.W  (A0)+,(A3)+     * 移动一个字
            CMPA.L  A0,A1           * 移动足够了吗?
            BPL     relo_lp
            JMP     new_ad
relo        LEA     test_patt(PC),A3 * A3指向使用的测试码
            LEA     bad_cnt(PC),A4   * A4指向坏的存储计数器
            LEA     bad_addr(PC),A5  * A5指向坏的地址位置
            LEA     data_read(PC),A6 * A6指向数据存储区
            CLR.B   (A4)            * 坏地址计数器清零
            MOVE.W  (A3)+,D0        * 获得当前测试码，并指向下一个
one
            BSR     do_test         * 运行第一个测试
            NOT.W   D0              * 对位求反用于第二个测试
            BSR     do_test         * 运行第二个测试
            MOVE.W  (A3)+,D0        * 获得下一个测试码
            BSR     do_test         * 运行第三个测试
            NOT.W   D0              * 对位求反用于第四个测试
            MOVE.W  (A3),D0         * 获得最后一个测试码
shift1      BSR     do_test         * 运行移位测试

            ROL.W   #bit,D0         * 移位的位数
            BCC     shift1          * 做完了吗? 没有的话返回来
            MOVE.W  -(A3),D0        * 再一次获得测试码3
            NOT.W   D0              * 对测试码3求反
shift2      BSR     do_test         * 运行这个测试
            ROL.W   #bit,D0         * 移动这些位
```

```
          BCS       shift2              * 做完了吗？没有的话返回来
message   MOVE.B    #pr_cmd,D0          * 装入指令来打印标语
          LEA       string(PC),A1       * 指向消息
          MOVE.W    str_len(PC),D1
          TRAP      #15                 * 开始做


done      STOP      #exit_pgm           * 退出回到模拟器
*************************************************************************

* 子程序：do_test
*
* 执行实际的存储器测试。用感兴趣的测试码填充存储器。
* 使用的寄存器：D1、A0、A1、A2
* 返回值：无
* 保存的寄存器：无
* 输入参数：
* D0.W = 测试码
* A4.L = 指向保存了坏地址数的存储器位置
* A5.L = 指向保存了最后一个被发现的坏地址的存储器位置
* A6.L = 指向保存了数据读回和数据写入的存储器的位置
*
* 假设：保存了所有内部使用的寄存器

*************************************************************************


do_test       MOVEM.L   A0-A2/D1,-(SP) * 保存寄存器
              LEA       st_addr,A0     * A0指向起始地址
              LEA       end_addr,A1    * A1指向最后地址
              MOVE.L    A0,A2          * 填充A2，将指向存储器
fill_loop     MOVE.W    D0,(A2)+       * 填充和指针递增
              CMPA.L    A1,A2          * 做完了吗？
              BLE       fill_loop
              MOVE.L    A0,A2          * 指针复位
test_loop     MOVE.W    (A2),D1        * 从存储器中读回值
          CMP.W     D0,D1              * 它们相同吗？
              BEQ       addr_ok        * 好，检验下一个位置
not_ok        MOVE.L    A0,(A5)        * 保存坏位置的地址
tion
              ADDQ.W    #byte,(A4)     * 计数器递增
              MOVE.W    D1,(A6)+       * 保存读回的数
              MOVE.W    D0,(A6)        * 保存写入的数
              SUBQ.L    #word,A6       * 将A6恢复为指针
addr_ok   ADDQ.L    #word,A2           * A2指向下一个存储器位置
              CMPA.L    A1,A2          * 我们到达最后一个地址了吗？
              BLE       test_loop      * 没有，继续测试
              MOVEM.L   (SP)+,D1/A0-A2 * 恢复寄存器
              RTS                      * 返回
* 数据空间
test_patt DC.W          pattern1,pattern2,pattern3 * 存储器测试码
bad_cnt   DS.W 1                       * 跟踪坏地址
bad_addr  DS.L 1                       * 在这里存储发现的最后一个坏地址
data_read DS.W 1                       * 我读回的是什么？
data_wrt  DS.W 1                       * 我写了什么？
string    DC.B 'End of test'           * 退出消息
str_len   DC.W str_len-string
last_addr DS.W 1

              END       start
```

第10章奇数号习题答案

1. <0C0020h> = 15C7h，该字是对齐的。

3. 0F57Ch

5.

```
        MOV CX,4
        MOV BX,10

loop1:
        inc BX
        dec CX
        jnz loop1
```

7.

```
<AX> = 0AF3DH
```

9.

```
      MOV  AX,8200H    ; 得到段值
      MOV  DX,AX       ; 装载段寄存器
      MOV  SI,0000     ; 装载源变址寄存器
      MOV  DI,0200H    ; 装载目的变址寄存器
      MOV  CX,1000     ; 装载计数器
loader:
      MOV  AL,[SI]     ; 得到字节
      MOV  [DI],AL     ; 存储字节
      INC  SI          ; 指针前进
      INC  DI
      DEC  CX
      JNZ  loader
```

第11章奇数号习题答案

1. 68K有两种操作模式，用户（user）和管理（supervisor）。ARM体系结构允许7种操作模式。用户模式是优先级最低的。其他模式为：系统、管理、异常中止、快速中断请求、中断请求和未定义。

3. 最大的不同在于，除了寄存器r13-r15，所有的寄存器都是完全通用的。任何寄存器都可以用作算术操作的一部分或用作地址指针，这与68K体系结构中地址寄存器A0-A6与数据寄存器D0-D7的区别形成了鲜明的对比。

5.
```
MOV r4, #&100
ORR r4, r4, #3
```

7. **<r11> = &0013E94C**

9. 如果零标志 = 0，那么r1的值&DEF02340增加4变为&DEF02344，然后该值被用作地址指针来检索存储在那个存储地址的16位数据对象，16位值然后被载入通用寄存器r4。如果零标志 = 1，那么指令不被执行。

第12章奇数号习题答案

1.

```
*********************************************************************
*  子程序：xmitStr
*  目的：将一串字符传送到UART串行口。
*  输入寄存器列表：
*      A6－指向要发送的数据串的指针。
*  返回寄存器表：
*      A6－指向字符串结束字符之后的字符的指针。
*  寄存器用法：所有xmitStr使用的寄存器将保存起来，并在退出后恢复。
```

```
*
*  假设：
*     -至少有一个字符要传送。
*     -字符串以$FF结束。
*
*************************************************************************

* 数据定义

eom         EQU         $FF          *消息结束字符
status      EQU         $2001        *状态寄存器
xmit        EQU         $2000        *传送数据寄存器
tbmt_mask   EQU         $01          *隔离传送缓冲器

* 子程序从这里开始

xmitStr     MOVEM.L     D0/D1/A0/A1,-(SP) *保存寄存器
            LEA.L       xmit,A0           *A0指向传送者
            LEA.L       status,A1         *A1指向状态寄存器
xmit_loop   MOVE.B      (A1),D1           *获得状态
            ANDI.B      #tbmt_mask,D1     *隔离位
            BEQ         xmit_loop         *仍然忙，保持等待
            MOVE.B      (A6)+,D0          *得到字节
            CMPI.B      #eom,D0           *是最后一个字节?
            BEQ         quit              *是，我们做完了
            MOVE.B      D0,(A0)           *将它装上
            BRA         xmit_loop         *返回去

quit        MOVEM.L     (SP)+,D0/D1/A0/A1 *恢复寄存器
            RTS
```

3. 逐步逼近总是用相同数量的时钟周期来数字化未知信号。由于它是16位分辨率，就需要16个时钟周期。由于是1MHz时钟速率，进行数字化就需要16μs。

   单坡A/D必须计数到未知电压，因此，我们需要确定到达1.5001V需要计数多少次。然而，我们可容易地看到16位转换器的范围是0到65 535（十六进制$0000到$FFFF），所以，A/D转换器的最小电压增量是0.0001V，因此，需要15 001个时钟周期或15 001μs来数字化未知电压。

5a. 一个11位的2补码数可表示一个范围在－1024到＋1023的数，所以数字值变化1就对应于0.010V。任何更小的电压是检测不出来的。

5b. 由于我们知道每个数字码的递增代表0.01V，所以我们就知道＋5.11V就代表511（5.11V/0.01V＝511）。用二进制，＋511就是00011111111，所以2补码值（－5.11）就是11100000001。

5c. 8.96V对应于数字值896，即01110000000。为了将其正确地表示成16位数字，我们需要在前面加入适当数量的零。这样，结果就是0000 0011 0000，即0x0380。

5d. 为了采用逐步逼近方法数字化一个11位的值，即二叉搜索算法的硬件类比，我们需要$LOG_2 2^{11}$即11个样本。

5e. 由于我们在时钟的每个上升沿取一个样本，而且我们需要11个样本，所以我们就需要11个上升沿。时钟频率是1MHz，所以周期就是1μs。这样，就需要11μs来数字化该模拟信号。

7a. 25μs＝40MHz频率。为了在每个周期采集4个样本，未知波形的最大频率必须不能大于10KHz。

7b. 14位转换＝16 384之1。10V/16 384＝ $\overline{0.0006V}$

7c. 在1ms内，它下降1V。在25μs，它下降 .(25/1000)*1=0.025V。由于这比转换器的分辨率0.0006大得多，所以S/H将引起重大错误。因此，不能使用它。

9. 解答：

   A. f－初始化硬件

   B. c－信任检测

C. e－选择通道
D. g－S/H
E. d－数字化
F. a－等待
G. b－取数据

另一可选择的解答：
A. c－信任检测
B. f－初始化硬件
C. e－选择通道
D. g－S/H
E. d－数字化
F. a－等待
G. b－取数据

第13章奇数号习题答案

1. 在这个例子中，代码段A通过流水线执行的效率要高一些。其原因在于A中的各个指令相对独立，没有依赖关系。而在代码段B中，必须一条指令执行完毕后，下一条指令才能运行。譬如，只有指令MOVE.W D1,D0把结果放入D0之后ADD.W指令才能开始运行。同样，ADD指令完成之前，MULU指令也不能执行。因而，在流水线的操作中，各条指令必须都必须等前一条指令完成后才能执行。

3. a. 否，因为它包含从存储器到存储器的操作。
   b. 是，加法操作发生在两个寄存器之间。
   c. 是，这里的MOVE是把一个寄存器中的内容读取到存储器中的存储操作。
   d. 否，AND操作发生在一个立即数和一个存储器中的数之间。
   e. 是，该操作是从存储器中把数据读入到寄存器的立即装入操作。

5. 该指令序列有如下几条RISC的特征。最重要的一条是ADD操作只发生在通用寄存器之间。此外，序列中没有通过直接指定存储器地址访问的寻址模式，存储器地址必须先装入寄存器，然后该寄存器作为存储器的指针来访问。因而我们只看到了两种寻址模式。

7a. 因为该流水线有7段，每段需要2个时钟周期，所以第一条指令下流水线时需要14个时钟周期。因为每个时钟周期为10ns，所以第一条指令执行完毕的总时间为140ns。

7b. 假设没有阻塞，那么第一条指令执行完毕后，剩下的9条指令将依次每隔2个时钟周期执行一次，即需要9个20ns，总共180ns来完成基本模块。然而由于每4个时钟周期流水线会阻塞两次，所以会额外增加80ns的开销（$2 \times 4 \times 10$），因此总时间为：

$$ET = 140ns + 180ns + 80ns = 400ns$$

第14章奇数号习题答案

1. a. 存储器的层次常常被表示成一个金字塔，顶端是CPU。它表示最靠近CPU的存储器，其访问速度最快，容量最小。反之，离CPU越远，容量越大，速度越慢。因此，存储器的访问速度与容量成反比关系。而且，离顶端越远，单个位的成本也越低。
   b. 空间局部性指的是指令和数据通常是成堆的放在一起的。指令是顺序存储，数据倾向于成组存放。对于cache，这就意味着如果指令和数据已经在cache中了，那么很可能接下

来访问的指令和数据也在cache中，因此，cache可比主存小很多，但仍然很高效。

时间局部性是指如果指令和数据最近被访问了，那么它很可能在不久的将来会被再次访问。因此，如果在cache中的内容最近被访问过了，那么它很可能还会被访问，这样就提高了cache的效率。

c. 对于cache，我们希望尽可能地提高命中率，降低不中惩罚。降低不中惩罚的一种方法是每次发生cache不中时，采用突发方式对cache的一部分而不只是一个字进行重填充。现代的SDRAM存储器都被设计成采用突发数据读取的形式来重填充片上cache，从而极大地降低了重新读入的时间。

d. 通写cache在将数据写入cache的同时也写入了主存储器。这样就避免了cache与主存储器中的数据不一致的问题，但也牺牲了一部分性能。回写cache则是暂时将数据放在cache中，等到总线可用时再将数据写入到主存储器。虽然性能得到了提高，但却冒着存储器崩溃的危险。

3. 空间局部性可以从如下三个方面看出来:

a. 编译后的指令在存储器中只占用了很小的一部分空间，长度只有32字节。因此我们可以假定它们存放的位置非常靠近。

b. 由于数组DataStream里的变量是通过对指针变量解引用，也就是DataStream加上一个偏移值count来进行访问的，所以数组里的各个元素在主存储器中必须是顺序存放在一起的。

c. 变量count和maxcount是函数main()的局部变量。因为编译器在系统栈上创建了一个栈空间，大小足够存放2个整型变量，所以它们必须被放置在一起。

时间局部性可以从如下一些方面表现出来：

a. 由于程序的主体部分是一个for循环，所以循环里的一行指令被执行了11次。

b. 变量count和maxcount被重复地访问，因为每次循环count都会递增并且与maxcount相比较。

c. 指针变量DataStream不断地被解引用，从而把count的平方放入连续的存储位置。

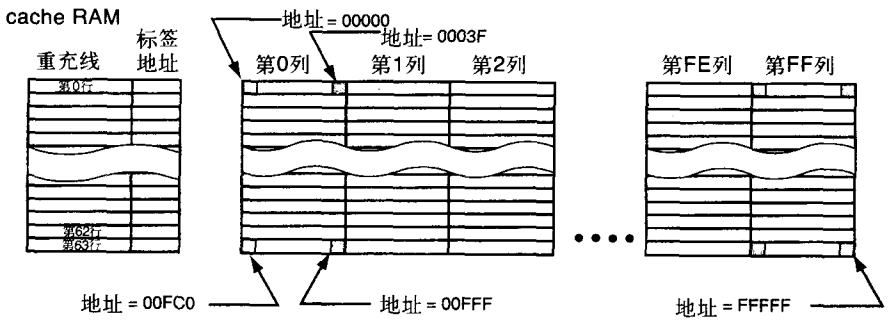
5. a. 主存的地址范围是00000...FFFFF，共2^{20}个地址，大约1MB。如果每个重充线大小为64B，即2^6，则重充线的数量为$2^{20}/2^6=2^{14}$，即主存储器中有16 384个重充线。

b. cache存储器的容量为4096个字节。采用与上面a中同样的方法，可得cache中有$2^{12}/2^6=2^6$，共64个重充线。

c. 由于这是一个直接映像cache，所以cache存储器和主存储器中重充线的行数相同。因此，重充线的行数乘以列数＝16 384。因此，重充线的列数＝16 384/64＝$2^{14}/2^6=2^8=256$，即主存储器中有256列重充线。

d. 由于列数为256，所以标签存储器中必须包含8位才能寻址256列中的任意一列。因此，标签存储器的地址需要8位。

e. 如下表：



存储器中字节地址 = 列地址 + 行地址 × 64 + 字节偏移

7. 有效执行时间＝命中率×命中访问时间＋不中率×不中惩罚
有效执行时间＝0.98×10＋0.02×100×10＝9.8+20=29.8ns

9. 当处理器开机初始化的时候，所有的TLB表项都是无效的。我们需要通过有效位来知道一个有效的表项是否已被放入TLB，或者只是垃圾。

第15章奇数号习题答案

1. 视频游戏者在对CPU进行超频以从机器榨取最后一点性能这方面是声名狼籍的。因为超频会产生更多的热量，这将减慢内部过程，也将导致处理器在接近设计极限处运行。液体冷却在排除热量方面比较有效，所以CPU能在有较高热负载时以较低温度运行。

3. 对于计算机1：
（1）每个指令在一个时钟周期内执行，即1/100MHz＝10ns。
（2）它必须执行总共1000＋200×100 = 21000条指令。
（3）总执行时间 = 21000×10ns＝2.1×10^4乘以10×10^{-9}
$= 21\times10^{-5} = 0.210\times10^{-6} = 210\mu s$

对于处理器2：
（1）它必须执行同样的21000条指令，但一些指令比其他指令需要两倍长的时间。因此，21000条指令中的40%执行一个时钟周期，60%执行两个时钟周期。
（2）在250MHz下，一个时钟周期需要4ns，2个时钟周期需要8ns。
（3）因此，总执行时间是：$0.4\times21000\times4\times10^{-9}+0.6\times21000\times8\times10^{-9}$
$= (8.4\times10^3)\times(4\times10^{-9}) + (12.6\times10^3)\times(8\times10^{-9})$
$= (33.6\times10^{-6}) + (100.8\times10^{-6}) = 134.4\times10^{-6} = 134.4\mu s$

5. 每条指令的周期数×每个时钟周期的时间 = 每条指令的时间
这就是我们想要的度量：
计算机1每条指令需要2个周期且每个时钟周期占用时间是1ns（1/1GHz），所以，计算机1执行一条指令需要2ns。
计算机2每条指令需要1.2个周期且每个时钟周期占用时间是2ns（1/500MHz），所以，计算机2执行一条指令需要2.4ns。
这样，性能＝2.4/2.0＝1.2，也就是说计算机1具有20%更好的性能。

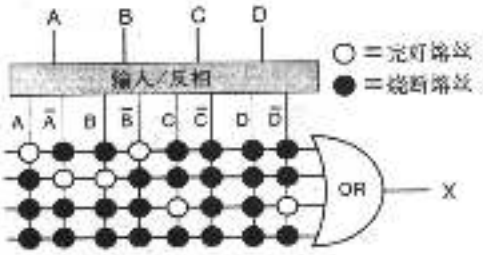
7. 分析这个问题需要我们考虑指令和实际加法操作两者所需要的访问次数。让我们在该例子中采用68000汇编语言。下面是一个有代表性的代码片段：

```
MOVE.L  var1, D0      *6字节长
ADD.L   var2, D0      *6字节长
MOVE.L  D0, var3      *6字节长
```

这样，加法操作需要18个字节从存储器读出或向存储器写入。8位宽总线就需要18次存储器访问，而16位宽总线就需要9次访问，所以在本例中，还必须计入额外的9次访问。

第16章奇数号习题答案

1. 熔丝图显示如下：



3. 像这样的具有大数量级数的电路能产生长序列的伪随机数。如果存在有缺陷的元件，那么该序列数将会快速地偏离电路完好时所期望的序列。在某种意义上，这是一个好的杂凑函数的硬件类比物，因此，任何缺陷都会迅速地产生与标准结果截然不同的结果。

索　引

页码为原书页码。

数字
2-phase clock（2相时钟），151
4 to 20 milliampere current loop（4到20毫安的电流环），335
68000 instructions（68000指令），232
  ADD（加），232
  ADDA（加到地址寄存器），232
  ADDI（加立即数），232
  ADDQ（加1到8之间的立即数），232
  ADDX（加X值），232
68K，
  instructions（指令），238
  MOVEM（移动多个寄存器），307
80286，266
8086，
  CPU（中央处理器），267
  instruction set summary（指令集总结），282

A

A/D conversion（A/D转换），333，336
  dual slop（双斜率），336
  flash conversion（快闪转换），336
  single slope（单斜率），336
  successive approximation（逐步逼近），336
  voltage to frequency（电压到频率），336
ABEL programming language（ABEL编程语言），419，422
abort mode（异常中止模式），300，301
absolute addresses（绝对地址），175
absolute references（绝对引用），229
abstraction levels（抽象级别），4
  application programming interfaces（应用编程接口），5
  arithmetic and logic unit (ALU)（算术和逻辑单元），5
accumulator（累加器），269
Actel，426
active（活跃的），77
addressable memory（可寻址存储器），135
addressing modes（寻址模式），171，202，304
address bus（地址总线），125
address decoder（地址译码器），128
address latch enable（地址锁存使能），268
address modes（地址模式），171
address registers（地址寄存器），
  a0...a6（68K处理器的地址寄存器），172
  direct（直接），175
  indirect（间接）175
  direct addressing（直接寻址），208
  indirect addressing（间接寻址），204
  indirect with postincrement or predecrement（带后递增或前递增的间接寻址），208
address space（地址空间），136
address strobe（地址选通），258
address tag（地址标签），377
ADD instruction family（加法指令系列），232
advanced addressing modes（高级寻址模式），230
advanced Boolean equation language（高级布尔方程语言），422
AGP，5
  bus（总线），81
algorithmic state machine（算法状态机），97，378
aliasing（混淆现象），206
ALU（算术逻辑单元），113
analog-to-digital conversion（模拟到数字转换），333
analog computer（模拟计算机），12
analog multiplexer（模拟多路选择器），344
AND（与），29
  instruction（指令），233
API（应用编程接口），5
application-specific integrated circuits（专用集成电路），96
application programmer's interface or API（应用程序员接口），241
application programming interfaces（应用编程接口），5
Aristotle（亚里士多德），49
arithmetic and logic unit (ALU)（算术和逻辑单元），5，113，173
arithmetic instructions（算术指令），237，239，283
ARM，159
  architecture（体系结构），295，296
  instruction set（指令集），309
  processors（处理器），298
  system vectors（系统向量），319
ASR（算术右移），303
assembler directives（汇编指令），183
assembly code（汇编代码），171
assembly language（汇编语言），167，168，171，209，229
  instruction（指令），129
  program（程序），170
  programming（编程，程序设计），159，193
  programming the 8086 architecture（8086体系结构编程），289
  source file（源文件），168
assert（置为有效），53
asserted（被置为有效），77
associative cache（相联cache），378，381
Associative Laws（结合律），51
  Associative law for AND（AND的结合律），51
  Associative law for OR（OR的结合律），51
  asynchronous（异步），71
  auto-incrementing（自动递增），206，212

auxiliary field (Aux)（辅助域），280

B

banding（将数据位转换为激光调制光束的过程），401
bank switch（存储体切换），300
barrel shifter（桶式移位器），302
  ASR（算术右移），303
  LSL（逻辑左移），303
  LSR（逻辑右移），303
  operations（操作），303
  ROR（循环右移），303
  RRX（扩展的循环右移），303
based indexed mode（基址变址模式），277
  with displacement（带位移的），277
based mode（基址模式），276
bases（基数），18
  base 2（基数2），18
  base 8（基数8），18
  base 10（基数10），18
  base 16（基数16），18
  binary（二进制），18
  decimal（十进制），18
  hexadecimal（十六进制），18
  octal（八进制），18
  radix（基数），18
basic block（基本模块），369
BCD（二进制编码的十进制数），24
benchmarks（测试基准），11,397
  SPECbase_int92，11
  SPECint95，11
big Endian（高端字节序），165
binary（二进制），18
binary-coded decimal (BCD)（二进制编码的十进制），24
  instructions（指令），239
binary bits（二进制位），23
  bit（位），23
  byte（字节），23
  double word（双字），23
  long word（长字），23
  nibble（半字节），23
  word（字），23
bit（位），23
bitwise AND（按位与），37
bit manipulation instructions（位操作指令），239
BIU（总线接口单元），267
block（模块），374
Boolean algebra（布尔代数），50
branch (B)（转移、分支），210,317
  analysis（分析），411
  delay slot（延时槽），368
  equal（相等），174
  instructions（指令），317
  prediction（预测），364
  target caches（目标cache），364
  with link (BL)（带链接的转移），317
branching and program control（转移和程序控制），200
buffer（缓冲器），33
burst mode access（突发模式访问），374
busses（总线），5
  AGP，5
  ISA，5
  PC-105，5
  PCI，5
  VXI，5
BUS GRANT（总线授权），153
bus interface unit（总线接口单元），267
bus organization（总线组织），123
BUS REQUEST（总线请求），153
bus width（总线宽），353
byte（字节），23
  addressing（寻址），161
  packing（包装），161
  selector（选择线），162

C

C++，209
cache（高速缓存），
  hit（命中），374
  memory（存储器），378
  organization（组织），376
  types（类型），378
    associative（相联），378
    direct-mapped（直接映像），378
    sector mapped（区映像），378
    set-associative（组相联），378
  write strategies（写策略），386
caches（高速缓存），146,372
CARRY（进位标志位），201
CCR（条件码寄存器），174
char（字符），12
chip enable (CE)（芯片使能），126
chip select (CS)（芯片选择），126
CISC（复杂指令集计算机），295,353,354
clock（时钟），64
  cycle time（周期时间），362
  skew（偏差），432
  speed（速度），353
clocked RS flip-flop（时钟触发的RS触发器），73
clockless computer（无时钟计算机），434
clocks and pulses（时钟和脉冲），62
CMOS（互补型金属氧化物硅），39
code segment (CS) register（代码段寄存器），274
coherent（一致的），376
combinatorial logic（组合逻辑），61
comment（注释），170
comparator（比较器），336
compilers and assemblers（编译器和汇编器），242
complement（补，反），33
complementary metal-oxide silicon（互补型金属氧化物硅），39
complex instruction set（复杂指令集），
  architecture（体系结构），10
  computer（计算机），295
  computer (CISC)（复杂指令集计算机），354
complex programmable logic devices（复杂可编程逻辑器件），424
compound gates（复合门），34,35

computer-aided design (CAD) software（计算机辅助设计软件），15, 116
COM ports（COM端口），81, 82, 330
conditional execution（条件执行），301
condition code register (CCR)（条件码寄存器），174
  C BIT (carry bit)（进位位），174
  N BIT (negative bit)（负数位），174
  V BIT (overflow)（溢出位），174
  X BIT (extend bit)（扩展位），174
  Z BIT (zero bit)（零位），174
condition control register (CCR)（条件控制寄存器），194
contents addressable memory（可按内容寻址存储器），382
converting（转换），
  decimals to bases（十进制转换为其他基数），25
  numbers（数），20
core（核），296
  memories（磁心存储器），131
coverage testing（覆盖测试），411
CPLD（复杂可编程逻辑器件），424
CPU16（原始的68K核），296
CPU32（全32位68K核），296
CRE option（CRE选项），185
cross-point switches（交叉点开关），421
current program status register (CPSR)（当前程序状态寄存器），298
current sources（电流源），341
cylinder（柱面），9
C and C++, 210, 229, 328
C BIT (carry bit)（进位位），174, 299

D

D/A conversion（D/A转换），333
data abort（数据异常中断），319
data bus（数据总线），125
data bus width（数据总线宽），135
data direction register（数据方向寄存器），329
Data I/O Corporation（数据I/O公司），422
data processing instructions（数据处理指令），310
data register direct（数据寄存器直接），175
data segment (DS) register（数据段寄存器），274
data storage directives（数据存储伪指令），184
  DC (define constant)（定义常量），184
  DCB (define constant block)（定义常量块），185
  DS (define storage)（定义存储），185
  CRE option（CRE选项），185
  OPT (set options)（置可选项），185
data transfer instructions（数据传输指令），238, 282
DBcc Instruction（DBcc指令），215
DCB (define constant block)（定义常量块），185
DC (define constant)（定义常量），184
DDR（双数据速率），147
decimal（十进制），18
decimal add adjust（十进制加调整），24
decode stage（译码阶段），359
DEC VAX（DEC公司的VAX计算机），387
defect tolerant（容缺陷的），428
defragmented（碎片整理），389
delayed branches（延迟的转移），368
demand-paged virtual memory（按需调页虚拟存储器），390
DeMorgan's Theorems（德·摩根定理），52, 166
destination index (DI) register（目的变址寄存器），271
Dhrystone，367
digital-to-analog (D/A) conversion（数字到模拟转换），332
digital-to-analog conversion（数字到模拟转换），333
digital logic（数字逻辑），29
Digital Research Corporation（数字研究公司），265
digital signal processor (DSP)（数字信号处理器），172, 357
direct-mapped cache（直接映像cache），376, 378
directives（命令），183
direct (absolute) addressing（直接（绝对）寻址），208
direct memory access (DMA)（直接存储器访问），152, 301
direct mode（直接模式），276
dirty bit（脏位），391
disassembly（反汇编），254
displacement（位移），195, 201, 280
  too large（过大），230
Distributive Laws（分配律），51
  First distributive law（第一分配律），51
  Second distributive law（第二分配律），51
DO/WHILE and FOR loops（DO/WHILE和FOR循环），193
DO/WHILE, 210 211
DO/WHILE Loop Construct（DO/WHILE循环结构），213
double（双），12
double and quad formats for floating point numbers（双精度和四精度浮点数），199
double data rate（双数据速率），147
double data type（双数据类型），198
double word（双字），23
DRAM（动态随机访问存储器），6, 145, 372
DRAM memory（DRAM存储器），146
drivers（驱动程序），4
DSP（数字信号处理器），353, 354
DS (define storage)（定义存储），185
duality（对偶性），402
dual slope（双斜率），336
duty cycle（占空度），66
dynamic（动态），
  memory（存储器），143
  branch prediction（转移预测），364
  RAM（随机存储器），145
  random access memory (DRAM)（随机存储器），6, 372
dynamic scheduling（动态调度），366
D flip-flop（D触发器），76

E

Easy68K，182
edge-triggered（边沿触发的），77
EDO DRAM（扩展数据输出DRAM），147
EEMBC（EDN嵌入式微处理器测试基准协会），406
effective address (EA)（有效地址），175, 176, 254
effective execution time（有效执行时间），375
electromagnetic shielding（电磁屏蔽），399
electromigration（电迁移），431
empty ascending（空上升），309
empty descending（空下降），309
empty stack（空栈），309

end-of-conversion（转换结束），345
END (end of source file)（源文件结束），183
engineering notation（工程符号），26
EQU (equate directive)（等于伪指令），184
Ethernet（以太网），5, 81
evaluation boards（评估板），407
exceptions（异常），323 327
excitation outputs（激励输出），107
EXTENDED（扩展位），201
extended data out（扩展数据输出），147
extra segment (ES) register（扩展段寄存器），274

F

falling edge（下降沿），32
fall time（下降时间），63
fast interrupt request mode（快速中断请求模式），301，319
feedback（反馈），71
fetch（取），176
fetch stage（取指令阶段），359
field programmable gate array（现场可编程门阵列），419, 424
finite state machine（有限状态机），96
Firewire（一种总线），5, 81
first-in, first-out (FIFO)（先进先出），267
flags（标志），174, 201
  CARRY（进位），201，
  EXTENDED（扩展），201
  NEGATIVE（负数），201
  OVERFLOW（溢出），201
  ZERO（零），201
flags field（标志域），299
  C bit（进位位），299
  N bit（负数位），299
  V bit（溢出位），299
  Z bit（零位），299
flag registers（标志寄存器），272
  Bit 0: Carry Flag (CF)（位0：进位标志），272
  Bit 2: Parity Flag (PF)（位2：奇偶标志），272
  Bit 4: Auxiliary Carry (AF)（位4：辅助进位），272
  Bit 6: Zero Flag (ZF)（位6：零标志），272
  Bit 7: Sign Flag (SF)（位7：符号标志），272
  Bit 8: Trace Flag (TF)（位8：陷阱标志），272
  Bit 9: Interrupt-Enable Flag (IF)（位9：中断使能标志），273
  Bit 10: Direction Flag (DF)（位10：方向标志），273
  Bit 11: Overflow Falg (OF)（位11：溢出标志），273
flash A/D converter（快闪A/D转换器），339
flash conversion（快闪转换），336
flip-flop（触发器），72
  D flip-flop（D触发器），76
  JK flip-flop（JK触发器），73
  RS flip-flop（RS触发器），72
  toggling（翻转），72
  float（浮点数），12, 24, 198
floating point（浮点），
  number（数），24
  units（单元），199
flowthrough time（流经时间），362
flow charting（流程绘制），181
flush（清空），360
FOR（FOR语句），210
FOR loop（FOR循环），210
FOR loop construct（FOR循环结构），214
FPGA（现场可编程门阵列），424
FPUs（浮点单元），199
frame pointer（帧指针），272
frequency（频率），66
fully associative cache（全关联cache），381
full ascending（满上升），308
full descending（满下降），309
full stack（满堆栈），308

G

Gary Kidall，265
general registers（通用寄存器），172
  eight data registers, D0...D7（8个数据寄存器D0...D7），172
  seven address registers, A0...A6（7个地址寄存器A0...A6），172
  stack pointers（堆栈指针），172
George Boole（乔治·布尔），50
giga（吉），8, 27
Gordon Moore（戈登·摩尔），2

H

handshake（握手），
  control（控制），433
  process（进程），153
hardware accelerators（硬件加速器），426
hardware architecture（硬件体系结构），1
hardware description language（硬件描述语言），2
Harvard architecture（哈佛体系结构），11, 355, 297
hazards（冒险情况），360
HDL（硬件描述语言），2
head（磁头），9
Heinrich Rudolf Hertz，65
Hertz（赫兹），65
heterogeneous（异构型的），149
hexadecimal（十六进制），18
hextant（一种交叉互连接结构），427
Hi-Z（高阻抗），29
homogeneous bus（同构型总线），149
hot boards（热板），407
Howard Aiken，11, 355

I

I/O ports（I/O端口），328
IBM PC-XT，265
IEEE-754，199
IEEE488，81
IEEE Standard 1364-1995，116
IF，210
IF/ELSE，210
IF statement（IF语句），198
immediate（立即），280

address mode（地址模式），176
operand mode（操作数模式），275
impedance（阻抗），29
incrementer（递增器），298
index addressing mode（变址寻址模式），305
preindex without write back（不写回的前变址），305
preindex with write back（带写回的前变址），305
post index with write back（带写回的后变址），305
index mode（变址模式），276，305
DS: MOV [DI+4], AL，277
indirect addressing（间接寻址），175，328
instruction（指令），
pointer (ip)（指针），273
prefixes（前缀），278
set（集），167，171
set architecture (isa)（集体系结构），108，159，354
set simulator (iss)（集模拟器），182
instrumenting（加检测），410
int（整数），12
integrated design environments (ide)（集成设计环境），182
intellectual property（知识产权），296
Intel x86 processor（Intel X86处理器），159
interface（接口），322
interrupts（中断），323
controller（控制器），259
request mode（请求模式），301，300
service routine (isr)（中断服务程序），323
intersegment jump（段间跳转），288
intrasegment jump（段内跳转），288
IP（知识产权），296
ISA（指令集体系结构），5，108

J

JK flip-flop（JK触发器），74
John von Neumann（约翰·冯·诺依曼），355
JSR（跳转到子程序），216
jump（跳转），210
to subroutine (JSR)（跳转到子程序），210，216

K

K-map（K-图），55
Karnaugh map（卡诺图），55
kilo（千），8，27

L

L1 cache（初级cache），7，373
L2 cache（二级cache），7，373
label（标号），170
last in, first out (LIFO)（后进先出），308
latency（潜伏期），309
Laws of Absorption（吸收律），52
First law of absorption（第一吸收律），52
Second law of absorption（第二吸收律），52
Laws of Commutation（交换律），51
Commutation law for AND（AND交换律），51
Commutation law for OR（OR交换律），51
Laws of Complementation（互补律），51
First law of complementation（第一互补律），51
Second law of complementation（第二互补律），51
Third law of complementation（第三互补律），51
Law of double complementation（双重互补律），51
Laws of Tautology（重言律），51
First law of tautology（第一重言律），51
Second law of tautology（第二重言律），51
Law of Tautology with Constants（带常数的重言律），52
LDM（装入多个寄存器），307
least recently used (LRU)（最近最少使用的），382
least significant digit（最低有效位），18
level 1（1级），7
level 2（2级），7
linear address space（线性地址空间），139
linear amplifiers（线性放大器），12
listfile（列表文件），182
literal (immediate) addressing（文字（立即）寻址），207
little Endian（低端字节序），165
load-use penalty（装入-使用惩罚），368
load/store（装入/存储），298
instructions（指令），312
loads（装入），298
load effective address (lea)（装入有效地址），212，219
locality（局部性），374
of reference（引用的），374
local clocks（局部时钟），432
logical and shift instructions（逻辑和移位指令），239
logical equation（逻辑方程），17
logical instructions（逻辑指令），233
AND（与），233
ANDI（与立即数），234
EOR（异或），234
EORI（异或立即数），234
OR（或），234
ORI（或立即数），234
NOT（反），234
logical memory（逻辑存储器），388
logical memory space（逻辑存储空间），388
logic analyzer（逻辑分析仪），409
logic instructions（逻辑指令），284
longs or long words（长字），23，167
LOW（低），77
low-power Schottky（低功耗肖特基），39
lower data strobe（低数据选通），165
LSL（逻辑左移），303
LSR（逻辑右移），303

M

machine language（机器语言）
code（代码），168
instructions（指令），168
Mars Rover（火星探测器），326
mask（屏蔽），325，326
master-slave（主从），75
Mead and Conway，2
Mealy machine（Mealy机），97
measuring performance（度量性能），404

mega（百万），8，27
memory（存储器），60
  addressing modes（寻址模式），275
  design（设计），123
  hierarchy（层次），7，373
    Level 1, or L1 cache（初级cache），7
    Level 2, or L2 cache（二级cache），7
  leaks（泄漏），411
  management unit（管理单元），388
  models（模型），289，290
    tiny（微），290
    small（小），290
    medium（中），290
    compact（紧凑），290
    large（大），290
    huge（巨），290
  organization（组织），159，162
  storage conventions（存储规约），160
  system design（系统设计），131
microcode（微代码），61，108
microcontrollers（微控制器），296
microseconds（微秒），26
millions of instructions per second（每秒运行的百万指令数），406
MIPS（每秒运行的百万指令数），406
  benchmark（测试基准），406
MMU（存储管理单元），388
mnemonics（助记法），169
mod（修饰符），279
Mode 0: Data Register Direct（模式0：数据寄存器直接寻址），203
Mode 1: Address Register Direct（模式1：地址寄存器直接寻址），203
Mode 2: Address Register Indirect（模式2：地址寄存器间接寻址），203
Mode 3: Address Register Indirect with Postincrement（模式3：带后递增的地址寄存器间接寻址），206
Mode 4: Address Register Indirect with Predecrement（模式4：带前递减的地址寄存器间接寻址），206
Mode 5: Address Register Indirect with Displacement（模式5：带位移的地址寄存器间接寻址），230
Mode 6: Address Register Indirect with Index（模式6：带变址的地址寄存器间接寻址），230
Mode 7, Subclass 000: Absolute Addressing (Word)（模式7，子类000：字绝对寻址），205
Mode 7, Subclass 001: Absolute Addressing (Long)（模式7，子类001：长字绝对寻址），205
Mode 7, Subclass 2: Program Counter with Displacement（模式7,子类2: 带位移的程序计数器），231
Mode 7, Subclass 3: Program Counter with Index（模式7，子类3：带变址的程序计数器），232
Mode 7, Subclass 4: Immediate Addressing（模式7，子类4：立即寻址），205
mode field（模式域），176
mod(modifier) field（修饰符域），279
Molecular Computing（分子计算），430
Moore's Law（摩尔定律），2
Moore machine（Moore机），97
MOSFET（金属氧化物半导体场效应晶体管），41，72
most significant digit（最高有效位），18
Motorola，148
Motorola 68000 processor（Motorola 68000处理器），159
MOVEA（移动地址），233
MOVEM（移动多个寄存器），217，233
MOVE instructions（移动指令），233
  MOVEA（移动地址），233
  MOVEM（移动多个寄存器），233
move not (MVN) instruction（移动取反指令），304
multiplexer（多路选择器），125，267
multiply-accumulate (MAC) unit（乘法-累加单元），297
MUX（多路选择器），125，267

N

negation（非），33
negation bubble（表示“非”的小圆圈），34
NEGATIVE（负数），201
negative logic（负逻辑），53
negative numbers（负数），194
negative pulse（负脉冲），62
nibble（半字节），23
Nintendo Game Boy（任天堂GameBoy游戏机），1
nonaligned access（非对齐访问），161
nonmaskable interrupt (NMI)（不可屏蔽中断），327
nonvolatile read-only memory (ROM)（非易失性只读存储器），372
NOP（不做任何事的指令），169，176
NOT（非），29
NOT instruction（取反指令），234
number systems（数制），12
  char（字符），12
  int（整数），12
  float（浮点数），12
  double（双精度浮点数），12
numeric representations（数值表示），193
N BIT (negative bit)（负数位），174，299

O

octal（八进制），18
offset（偏移），291
offset value（偏移值），201
Ohm's Law（欧姆定律），338
one-time programmable（一次性可编程），421
one to N（1到N），123
opcode（操作代码），169，170，183，279
  word（字），254
open collector（开集电极），420
open drain（开漏极），420
operands（操作数），170
  address（地址），279
  size（大小），303
OPT (set options)（置可选项），185
OR（或），29
ORG (set origin)（置初始值），183
orthogonal（正交的），297
oscillator（振动器），89
oscilloscope（示波器），63

OTP（一次性可编程），421
out-of-range error（越界错误），230
output enable (OE)（输出使能），126
overclocking（超频），402
OVERFLOW（溢出），201

P

page-table base register（页表基址寄存器），391
pages（页），389
page（页），
  fault（故障），390
  frames（帧），390
  frame number（帧号），391
  map（映像），391
  number（号），137
  offset（偏移），137
  table（表），391
  table entries（表入口），391
paging（分页），137
PAL（可编程阵列逻辑），419
PALEs（可编程原子逻辑元件），427
paragraphs（段），273
parallel port, LPT（并行端口，LPT），81
PC-105，5
PCI，5
PCI bus（PCI总线），81
PC relative（PC相对），231
performance issues（性能问题），397
period（周期），64, 66
phase lock（锁相），402
phase-locked loop (PLL)（锁相环），402, 432
pipelining（流水线），358
plasma（可编程逻辑和开关矩阵），427
polling（轮询），324
  loop（循环），324
positive logic（正逻辑），34, 52
positive pulse（正脉冲），62
postindex with write back（带写回的后变址），305
prefetch abort（预取异常中断），319
preindex（前变址），
  with write back（带写回的），305
  without write back（不写回的），305
priority encoder（优先级编码），340
priority inversion（优先级颠倒），326
privileged instructions（特权指令），239
processor architectures（处理器体系结构），354
processor modes（处理器模式），300
  abort mode（异常中止模式），301
  fast interrupt request mode（快速中断请求模式），301
  interrupt request mode（中断请求模式），301
  supervisor mode（管理模式），301
  system mode（系统模式），300
  undefined mode（未定义模式），301
  user mode（用户模式），300
processor status register（处理器状态寄存器），299
programmable array logic（可编程阵列逻辑），419
programmer's model（程序员模型），171, 173
Programmer's Reference Manual（程序员参考手册），178
program control instructions（程序控制指令），239
program counter（程序计数器），172
program counter register（程序计数器寄存器），328
program status register instructions（程序状态寄存器指令），318
propagation delay（传播延时），32, 63
protected mode（保护模式），266
protection bits（保护位），391
pseudo-ops（伪操作代码），183
pseudo opcodes（伪操作代码），183
  END (end of source file)（源文件结束），183
  EQU (equate directive)（相等伪指令），184
  ORG (set origin)（置起始位置），183
  SET (set symbol)（置符号），183
pull-up resistor（上拉电阻），421
pulse width（脉冲宽度），62
push and pop（压入和弹出、压栈和退栈），308

R

r13: stack pointer（堆栈指针），298
r14: link register（链接寄存器），298
r15: program counter（程序计数器），298
race condition（竞争条件），74, 75, 89
radio frequency interference (RFI)（射频干扰），399
radix（基数），18
radix complement（基补码），195
RAM（随机访问存储器），131
random access memory（随机访问存储器），131
rasterization（光栅化），95
real-time（实时），
  operating systems（操作系统），325
  trace（跟踪），411
real numbers（实数），194, 198
real world（真实世界），322
reconfigurable（可重构的），
  computing（计算），424
  hardware（硬件），419
reduced instruction set computer, RISC（精简指令集计算机），240, 295, 355
refill line（重充线），378
refresh cycle（刷新周期），145
register/memory (R/M)（寄存器/存储器），279
register（寄存器），212
  and immediate（和立即数），304
  D0...D7，172
  direct addressing（直接寻址），203, 207
  field（域），176
  file（文件），297
  indirect mode（间接模式），276
  maps（图），329
  operand mode（操作数模式），275
relocatable（可重定位的），229
Rent's Rule（Rent规则），427
RESET（复位），81, 82, 324
resistor（电阻），339
return（返回），218
return from subroutine (RTS)（从子程序返回），217
ripple counter（行波计数器），81

RISC（精简指令集计算机），295, 353, 354, 355
RISC processors（RISC处理器），368
rise time（上升时间），63
rising edge（上升沿），32
ROM（只读存储器），372
ROR（循环右移），303
RRX（扩展的循环右移），303
RS-232，81
RS flip-flop（RS触发器），72
  clocked RS flip-flop（时钟触发的RS触发器），73
  state dependency（状态依赖），73
RTOS（实时操作系统），325

S

S/H（采样和保持），344
sample and hold module（采样和保持模块），344
saved program status register (spsr)（程序状态保存寄存器），298
SBZ（应为零），311
schematic diagrams（示意图），16
SDRAM memory（SDRAM存储器），143, 146
sector-mapped cache（区映像cache），383
sectors (rows)（区（行）），383
segment: offset（段：偏移），280
segment override prefix（段超越前缀），278
segment registers（段寄存器），267, 273, 274
  code segment (CS)（代码段），274
  data segment (DS)（数据段），274
  stack segment (SS)（堆栈段），274
  extra segment (ES)（扩展段），274
self-modifying code（自修改代码），355
set-associative cache（组相联cache），382
SET (set symbol)（置符号），184
Shakespeare circuit（莎士比亚电路），44
shift and rotate instruction（移位和循环移位指令），234
shift register（移位寄存器），81
signature analysis（标记分析），429
sign bit（符号位），195
sign extended（符号扩展的），231
sign extended address（符号扩展的地址），205
silicon compilation（硅编译），2, 116
silicon compilers（硅编译器），116
Silicon Valley（硅谷），1
SIMM，6
single-ramp（单坡），342
single slope（单斜率），336
software interrupt instructions（软件中断指令），317
SP（堆栈指针），211
spatial locality（空间局部性），374
SPECbase_int92，11
SPECint92，11
SPECint95，11
speculative execution（投机执行），366
SRAM（静态随机存储器），6, 372
stack（堆栈），211
  and subroutines（和子程序），216
  implementation options（实现的选项），308
    full ascending（满上升），308
    full descending（满下降），309
    empty ascending（空上升），309
    empty descending（空下降），309
  operations（操作），306
    LDM（装入多个寄存器），307
    STM（存储多个寄存器），307
  pointer（指针），172
  segment (SS) register（段寄存器），274
standard cells（标准单元），346
standard memory module (SIMM)（标准存储模块），6
Standard Performance Evaluation Corporation（标准性能评估公司），404
state dependency（状态依赖），73
state machine（状态机），61, 84, 95-118
state transition diagram（状态迁移图），84
static（静态的），
  memory（存储器），143
  RAM (SRAM)（随机存储器），6, 145
  random access memory (SRAM)（随机存储器），6, 372
status bus（状态总线），125
status register（状态寄存器），173
STM（存储多个寄存器），307
storage register（存储寄存器），83
stored charge（存储电荷），145
stores（存储），298
string manipulation（字符串操作），285
successive approximation（逐步逼近），336, 343
superscalar architecture（超标量体系结构），356
supervisor mode（管理模式），216, 300, 301
supervisor stack pointer（管理堆栈指针），216
SWITCH（开关转移），210
synchronous dynamic random access memory（同步动态随机存储器），143
system-on-silicon（硅上系统），296
system interface bus (SCSI)（系统接口总线），5
system mode（系统模式），300
system vectors（系统向量），291, 328

T

tag（标签），377
tag memory（标签存储器），378
Teesside assembler（Teesside编译器），182
Temporal locality（时间局部性），374
tera（太），8
Teramac，428
text editor（文本编辑器），168
three-terminal device（三端装置），18
throughput（吞吐量），362
thumb mode bit（thumb模式位），299
time critical（时间关键的），405
time sensitive（时间敏感的），405
timing diagram（时序图），77
toggling（翻转），72
translation lookaside buffer (TLB)（转换旁路缓冲器），391
TRAP #15 instruction（TRAP #15指令），240
TRAP instructions（TRAP指令），200
tri-state (TS)（三态），29
TRUE or FALSE（真或假），16
truth tables（真值表），35, 4
two's complement（补码），195

U

UART（通用异步接收器/发送器），330
undefined mode（未定义模式），300, 301
universal asynchronous receiver/transmitter（通用异步接收器/发送器），330
universal serial bus (USB)（通用串行总线），5
upper data strobe（高数据选通），165
USB（通用串行总线），81
user mode（用户模式），216, 298, 300, 392
  r13: stack pointer（堆栈指针），298
  r14: link register（链接寄存器），298
  r15: program counter（程序计数器），298
user stack pointer（用户堆栈指针），216

V

validity bit（有效位），383, 391
vectors（向量），327
Verilog，2
very large scale integration（超大规模集成），115
VHDL，2
vias（通孔），16
virtual（虚拟），
  address（地址），387, 388
  extension（扩展），387
  memory（存储器），372, 387
  page number（页号），390, 391
VLSI（超大规模集成），115
volatile（易失性），330
voltage divider（分压器），339
voltage to frequency（电压到频率），336
von Neumann architecture（冯·诺依曼体系结构），355
von Neumann bottleneck（冯·诺依曼瓶颈），355
VXI，5
V bit（溢出标志位），299
V BIT (overflow)（溢出位），174

W

wait state（等待状态），150
WHILE，210
wired-AND（线与），420
word（字），23
  alignment（对齐），178
  offset（偏移），390
write-around cache（绕写cache），386
write-back cache（回写cache），386
write-through cache（通写cache），386

X

X86 instruction format（X86指令格式），278
Xilinx，426
X BIT (extend bit)（扩展位），174

Z

Z80，265
ZERO（零），201
zero flag（零标志），174
Zilog, Inc.（Zilog公司），265
Z BIT (zero bit)（零位），174, 299