作者简介

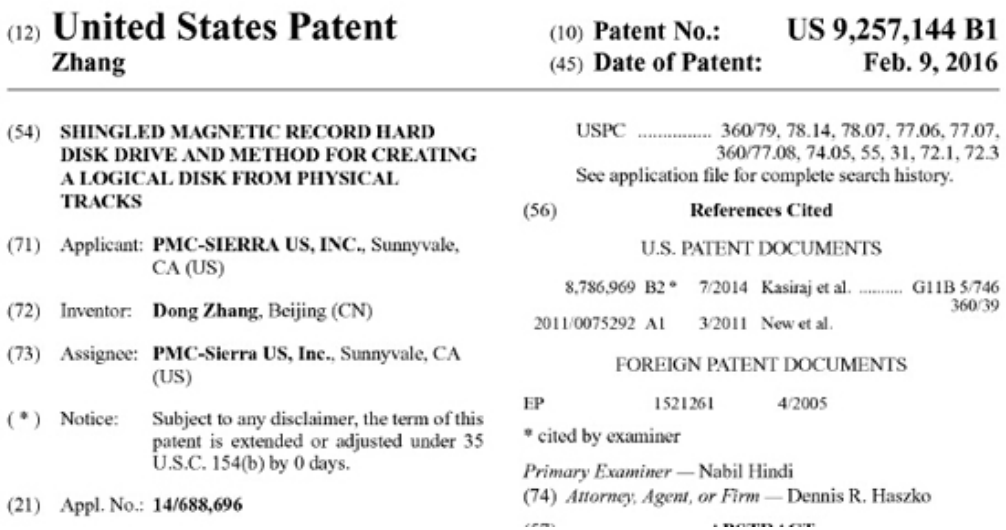
冬瓜哥，现任某半导体公司高级资深架构师

《大话存储》系列图书作者。大话存储 公众号作者

4项专利唯一发明人

大话计算机

大话存储



(12) **United States Patent**
Zhang

(10) **Patent No.: US 9,257,144 B1**
(45) **Date of Patent: Feb. 9, 2016**

(54) **SHINGLED MAGNETIC RECORD HARD DISK DRIVE AND METHOD FOR CREATING A LOGICAL DISK FROM PHYSICAL TRACKS**

(71) Applicant: **PMC-SIERRA US, INC.**, Sunnyvale, CA (US)

(72) Inventor: **Dong Zhang**, Beijing (CN)

(73) Assignee: **PMC-Sierra US, Inc.**, Sunnyvale, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **14/688,696**

USPC 360/79, 78.14, 78.07, 77.06, 77.07, 360/77.08, 74.05, 55, 31, 72.1, 72.3
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

8,786,969 B2 * 7/2014 Kasiraj et al. G11B 5/746 360/39
2011/0075292 A1 3/2011 New et al.

FOREIGN PATENT DOCUMENTS

EP 1521261 4/2005

* cited by examiner

Primary Examiner — Nabil Hindi
(74) *Attorney, Agent, or Firm* — Dennis R. Haszko

¥94.10

大话存储2：存储系统架构与底层原理极

1300+条评价

清华大学出版社

¥141.60

大话存储：存储系统底层架构原理极限剖

7000+条评价

清华大学出版社

¥84.60

大话存储后传：次世代数据存储思维与技

7000+条评价

清华大学出版社

新书简介

新书书名：**大话计算机——计算机系统底层架构原理极限剖析**

页数：12章，1500页，图片相当多，全彩印刷

定位：计算机专业科普书、百科全书。从零到最顶端全技术覆盖深度阐述。IT领域从业者全覆盖。

特点：从零开始，覆盖全面，而且通俗，接地气，代入感强烈，引发读者思考，然后解决问题，将思考过程和解决办法描述下来。符合人类原生态认知、求知思路。

出版日期: 2019年4月

本书特点

01 绝对从初学者角度出发，看了**不迷茫、不撕书、不会骂作者**（我也不想被人骂）。

02 介绍事物绝对庖丁解牛，轻易不留**“坑”**，不得已留了则必填。

03 带着思考来写作，**促发读者思考**；问题导向，带着解决问题过程来写。

04 事物之间带有**前因后果关联**，而不是孤立地介绍，整本书从第一页到最后一页有一条清晰的**因果脉络**。

05 全局框架和局部细节兼顾，**大而全，深而细**，就像一部精美的游戏，宏观场景震撼，局部细节惊艳！

06 覆盖面极广，涵盖多个领域关键知识：**数字电路、模拟电路、计算机体系结构、计算机组成原理、操作系统原理、计算机图形学、高性能计算机集群/超级计算机、信号与系统、存储系统、网络/通信系统、机器学习与人工智能**等。

07 随便点选任何一页的任何一段，都是**精彩和拥有丰富细节的内容**。

08 注重大框架的建立，让读者阅后**成竹在胸**。

09 **便于自学**，看文字就像是在聆听作者当面讲授。

10 图片细节丰富，带有**事物运行的流程**，其过程是动态的而不是静态的。

业界大牛点评

打开来读，他的文字带我走过一个个章节，一直翻到了最后一章，感觉就像自己温习了一遍大学本科的课程，还有从业二十年来看过的许多专业书籍、科研文献、设计文档。纵观市面上所有计算机图书，能在一本书中说清楚计算机工程(COMPUTER ENGINEERING)的方方面面的关键要点的，目前只有冬瓜哥的这本《大话计算机》了。

廖恒

海思半导体 首席科学家、FELLOW

读者反响强烈

【高手】Bottle cap
你这ppt简直登峰造极

【高手】Bottle cap
感觉我对不起CAD这款绘图工具

是啊，这都400多，本屑的书400不过分，帧不过分

初学者
不过分

彩色的牛
不过分

TIM
不过分

IQYA
我之前看书的目录，发现都看不懂，汗颜啊

@Nervermore 内核部分也是迄今为止

Aaron
@冬瓜哥
刚大概看了您发的pdf，想必您一定花了不少心血，书中的每一幅图都简洁明了，每一个问题的解释通俗易懂又恰到好处，几乎做到了极致，堪称完美。

深山野老

【大师】AMP
冬瓜哥这ppt技术我是服气的

Anonymous
我看到了未来的dalao

教官*
等待你的书。快点出来

教官*
@冬瓜哥

Anonymous
我也是，这本书我也等了好久@冬瓜哥

【管理员】等风的旗
ppt真的是牛逼到爆

QYA
不过看了刚才发的具体内容，发现很容易明白。

【大师】蒲公英
ppt真的是牛逼到爆

【高手】旭日东升
ppt真的是牛逼到爆

卧槽 牛啊

【高手】缘来&似梦
冬瓜哥的图确实画的好

【入门】同学甲
冬瓜哥的图确实画的好

【高手】深圳　勇敢的男孩
ppt真的是牛逼到爆

这个搞的比计算机博士生还专业啊

【专家】Captain Sparrow
体量好庞大

【专家】聂永
确实牛逼

夕阳如斯
小白不敢点菜 冬瓜上啥都行！
2018-04-23 17:49:06

彩色的牛
N多年之后，瓜哥的书就被列为义务教育教科书范畴~

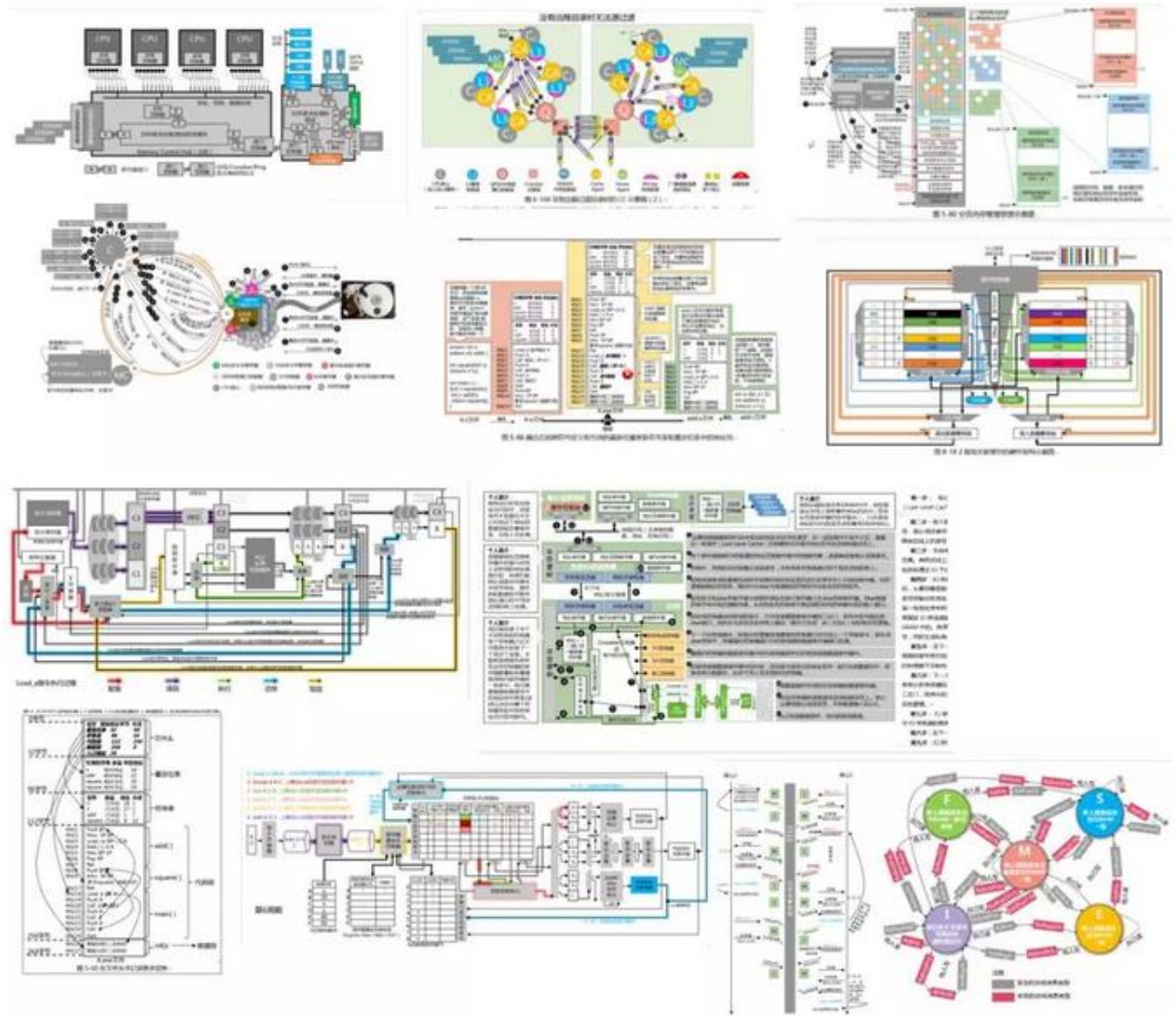
【管理员】不是摄影
确实漂亮 是个好ppt教程

【专家】聂永
这本书

冬瓜哥这本百科全书我是服气的，对于不懂计算机的人能提供强大的工具书作用

彩色的牛
不要，我要看完瓜哥的《大话计算机》就算是慰藉我的计算机生涯了

为何本书会顺手被网友称为PPT教程？（更多图片下面还有）



记得多年前有这么一道研究生入学面试题：“播放幻灯片时，按下一个空格键到屏幕显示下一页，请问这个过程计算机做了什么？”大多数参加面试的学生都答不上来。但是有一天看到冬瓜哥的《大话计算机》时，我第一反应是意识到自己错了。这本将近1400页的恢宏巨作，约500个章节，涵盖了处理器流水线、缓存、内存、并行计算、网络、声卡、GPU、操作系统，甚至包括半导体制造工艺等，每一章节都是深入浅出。不同于一般的教材，这本书采用诙谐幽默的笔法与图文并茂的形式向读者揭示计算机内部各个部件的工作原理，并穿插着技术背后的种种名人轶事，读来生动活泼、引人入胜。这像是一部小说，更像是一本百科全书，按图索骥，总能找到你想了解的知识点。

包云岗

中科院计算所研究员，
先进计算机系统研究中心主任，中国科学院大学岗位教授

当年我看游戏《半条命：反恐精英》的二三百万行源代码时，有同事说，网传看完就只剩下半条命了。当我看到冬瓜哥这本《大话计算机》时，瞬间感觉我看完这本书可能也只剩下半条命了。并不是因为这本书的难度摧残大脑，相反，这完全是另一种感觉，它太通俗了，把事物的关系、流程、架构讲的太清晰了，信息量太大了，看完之后就是身体被掏空感。因为，我发现自己这么多年习得的仅有的一点点计算机方面的绝招、秘密在这本书里竟然一点不落的都有，而且还通俗易懂，让我瞬间感觉之前自己耗费在学习计算机上的时间，简直成了浪费人生。

张勇

深圳市科力锐科技有限公司创始人之一

期待了很久很久的冬瓜哥巨著终于到手，这部拆分为三厚本印刷出版的巨著，图文数量和专业度质量，都直追高纳德圣经，堪称镇宅典藏珍品，终生研读！

终于收到瓜哥的大作，瓜哥付出4年的时间专注于一本书，其对技术的热爱，对专业的专注和坚持，让人敬佩！

凝结智慧与辛劳的巨著，天下贤士共赏之，莘莘学子向往之，必不胫而走，洛阳纸贵

扎实的分量，精致的印刷，这绝对赚了

收到冬瓜哥呕心沥血大作，看到云岗和万青的作序，会心一笑，非常感动，这年头专心致志写一本书，花几年时间，在鎏金镀银随时随地赚大钱的计算机行业，独处书斋，屏蔽万事，放弃了无数赚钱的机会，真是可敬可佩！

平生不看大话计算机，阅尽github所有源码也枉然

嗯 超值

冬瓜哥作图技能点已满

已推荐同事购买

深夜拜读冬瓜哥大作，停不下来了

那些复杂如蜘蛛网的示意图和架构图可能刚开始的时候确实看不懂，但是多看一段时间之后，就会从线条和文字中隐隐约约看出四个字： 瓜哥牛逼

我大致翻了下，很舒服

是舒服

我看这本书的图画的很牛逼

@冬瓜哥 这套书，你确实费了心思

下午 16:39:48

区别其他书，图和内容都是老哥自己搞明白后，自己画的

拿到书了/顶呱呱/顶呱呱

这书写的特别细

@永无止尽 这个盒子回收值钱

下午 12:24

不舍得回收

弱智天君

PLUS会员

★★★★★

一年多前就在书的序言人之一的朋友圈里看到对此书和作者的大力称赞。了解到作者坚持了四年，用心为各个层次的读者，也为中国计算机界带来了一份原创巨作，这份努力实在值得敬佩，国内各行各业都太需要这样精雕细琢的工匠精神。看到预售消息之后，果断买了首发版表示支持。书拿到手快速翻了一下，从印刷质量到排版方式，诚意十足，丝毫不掺水分。

先翻看了前几页内容，暂不做过多吹捧，内容绝对够细致，能感受作者融会贯通之后的娓娓道来。

第3页正文看到了个校对时未发现的typo，相信作者一定是个追求完美的人，略觉可惜。

相信该书一定能成为一个传奇，向坚持奋斗不懈努力的人至敬。

2019-04-27 02

举报 1 0

真的震惊，这三本书的信息量

光看第一章关于东西方人对符号的理解的思维方式就已经让我体验一次恍然大悟了

已经收到的我表示沉迷东哥的书无法自拔

2019-04-28 11:40:24

信息量太大， 我表示没个 几个月，吸收不了。

书很赞,已经推荐给学弟学妹啦

不过书的质量确实可以。 内容丰富

“ 019.4.26.星期五 下午 23:08:41 查看原文

没有一点it知识的人能看懂这个书吗

没问题的

昨晚睡前一口气看了第一本的70%

正在翻看，图片，图表 印刷很好，挺值的

梳理知识体系 挺不错的

晚饭后浏览了这厚厚的大话计算机，此书堪称计算机的百科全书，通俗易懂，从初学者角度出发，看了不迷茫，不撕书，不骂作者。全书只有冬瓜哥一个作者，不禁佩服是什么驱使冬瓜哥坚持完成了此著作？唯有情怀！向东哥致敬，东哥也给自己赞一下吧😄

1小时前

大作已收到，真是学习计算机系统难得一见的好书。如此巨著以一人之力完成，深感钦佩！一定在师生中大力推荐👍

j***y

PLUS会员

★★★★★

计算机书籍的经典之作啊，真的太棒了！

大话计算机　2019-04-28 13:37

k***i

★★★★★

很不错，深得我心

专业必备　正版　脉络清晰　理论基础　性价比高

大话计算机　2019-04-28 13:10

2***2

★★★★★

期待已久的神书，心心念念了好久。精美的插图，详尽的知识。果然冬瓜哥出品必原有的一堆书 仔细阅读它就可以了 希望自己有大的提升！

另：获得了为数不多的彩蛋版 冬瓜哥亲笔签名，激动一下

大话计算机　2019-04-28 12:24

章节导图导出图像时选的分辨率不够，导致文字略微有点糊

我以前搞过高清喷绘、数码快印🤔

书中的图片非常精美！

我要是在高校任教的话一定要选为必修教材

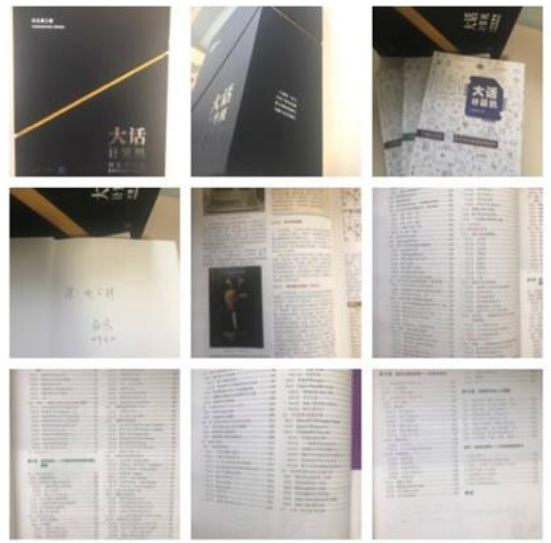
👍👍

中国移动　下午5:21　43%

朋友圈

感谢冬瓜哥皓首穷经完成的鸿篇巨著《大话计算机》，全彩色印刷，图文并茂，堪称科技届的《资治通鉴》。覆盖了硬科技的方方面面，既有前沿的人工智能，又有厚重的科技历史。古有司马光，今有冬瓜哥。大人看完值得再传给孩子读，让他们发现科技的世界竟然这么宏大和有趣👍

收起



16:57 ... 4G 54

详情

15:30

回复[redacted]: 来借书啊

15:31

回复[redacted]: 有这想法，

16:46

回复[redacted] 刚上市的

16:50

回复[redacted]网上评价蛮高的，我也买了这个书！

16:54

回复[redacted]: 冬瓜哥历时四年的憋尿之作，的确值得买来学习

16:56

回复[redacted] 书真还不错，就是有点贵

16:56

回复[redacted]我买了两套，一套给公司，一套自己看

16:57

回复[redacted]

评论 发送

这个书，不好好看，对不起冬瓜哥啊

今天请了半天假，下午溜回去看书了

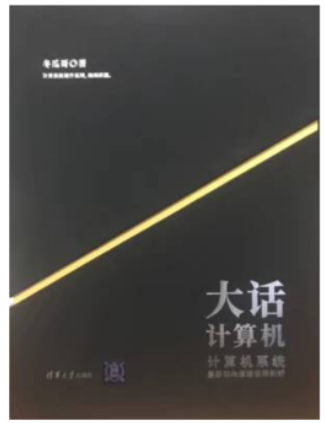
写的太太太……好了

@冬瓜哥 瓜哥，杭州的书到了！可惜我出差，我妈代收的。 我妈问我什么东西这么重，我说是知识的重量

朋友圈

[redacted]

葵花宝典已到手，从来没见过字这么小的大部头书，太实诚了。



4分钟前

[redacted]

冬瓜哥: 要是常规5号字，就得2000页，价格就会更高

[redacted]

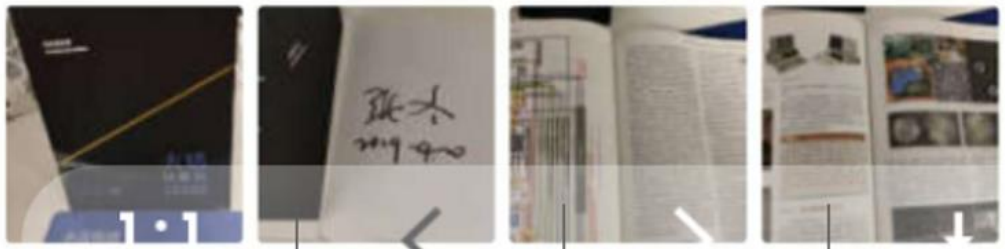
只看当前商品　　好评度100%

全部(70+)　最新　好评(60+)　中评(0)　差评(0)

视频晒单(1)　有图(9)

怎***买 ★★★★★　　2019-04-29

作为一个从事计算机it行业20年的，读书人。强烈推荐！如果你想快速跨入这个行业，本书必读！我已经把这套书作为，公司培训教材了。公司新人必读！这不是教材！人文看浪潮之巅，技术看《大话计算机》！通读过后，你的能力，会跃然提升！可以短时间积累足够的行业经验！菜鸟变专家，这套书是捷径！书贵吗？对比各种培训，简直是超级超级超级的有性价比！



颜色：大话计算机　　回复

@冬瓜哥 大话计算机的印刷真的不错，读起来特别引人入胜，随时可以沉浸进入👍👍👍

「冬瓜哥: @老鹰 哎，误人子弟了。。🤭」

- - - - - - - - - - - - - - - -

我记得你当时和我讲 - 这本书的写作，就是一个边学习边总结的过程，其实这种写法，挺适合学习者的，特别不是科班的学习者 - 咱们自己的科班教育模式，对学习计算机其实并不好，这也是这本书的价值所在，我读的时候，这种感觉。

聊天 公告 相册 文件 活动 设置

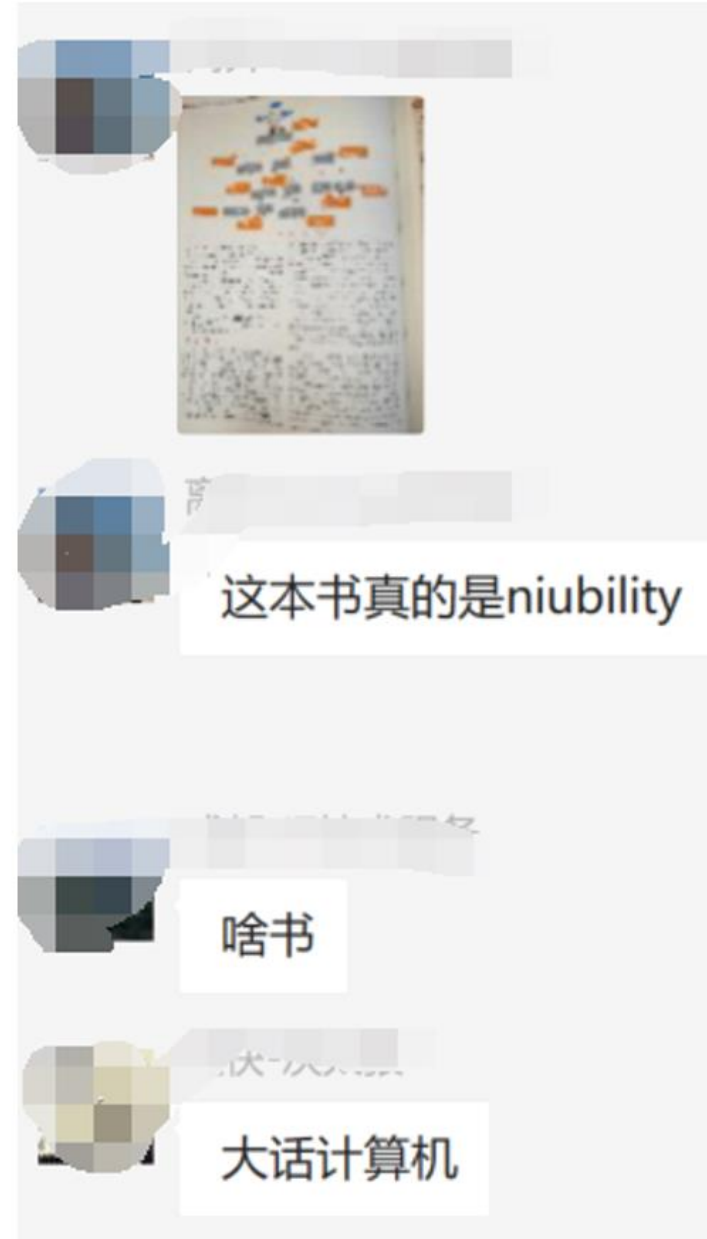
这两天晚上粗略翻了一下这套书，累计一个小时左右吧。

人已四十，咱既不是粉，也不是吹。只是有些想法，想留一下。

首先，这套书写的真是太系统太细致了。让人不自觉的想沉浸进去，享受那种从迷失到恍然大悟的快感，似不似有点类似于吸毒？书到之前，本以为这套书应该是有联合作者的，因为这套书关联了三个领域。这是我第一次看到有这么一套从逻辑电路一点点展开自下而上延伸到计算机结构再延伸到操作系统内核的经典书籍。

那么我会怎么看这套书？首先，由于所从事的领域不在电子方面，所以前四章我估计暂时不看的。我将从第五章开始，正式起步从第五章后半段开始。
我会怎么用这套书？其实迷失在一套书里，那种欲仙欲死的着魔状态非常的痛苦。为了避免，我将会打开这套书，同时也打开电脑中积累的老资料。沿着这套书的思维导图，遍历每个小节。然后让我脑中的认知、老的资料、大话计算机中的论述，让这三部分的内容在大脑中碰撞，或者是相互融合，或者是互相纠正。总之无论是互补互通还是破而后立，都会让自己有质的提高。

每拿到一本书，我总喜欢先琢磨目录结构。最后，说一下从目录结构上看，我的一个粗浅的认识，一点遗憾：这本书专门讲了Linux32位内存/地址结构，但似乎没有讲64位内存/地址空间的内容。相关的64位内容，网上的资料太杂，我一直在找到权威的书籍去澄清脑中认知。看来还要继续找下去。



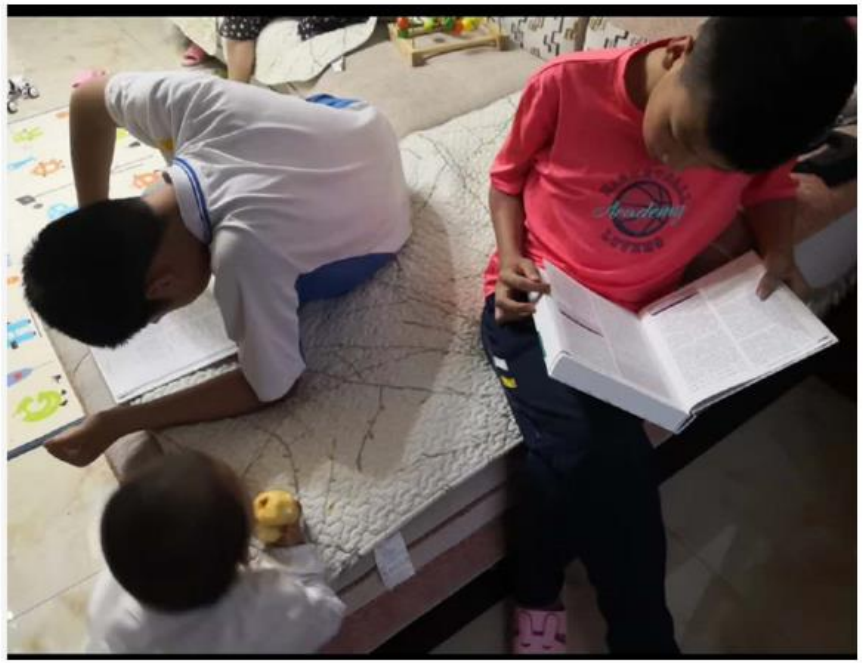
与描述相符
5
★★★★★

1 非常不满　2 不满意　3 一般

◉ 全部 ○ 追评 (0) ○ 图片 (0)

好书，建议店家多大广告一下宣传这个书，作者的书是目前看到最佳的一本没有之二！

不过严格的来说这是一本精美的ppt展示书籍



假装你喜欢我

我敢保证，这确实是我看过国内外最好的同类图书，没有之一。像作者致敬，感谢他分享自己的知识，为后辈写出如此好的著作。

大话计算机 2019-05-02 13:15 举报 0

老***头

PLUS会员

首先对这样一部倾注心血而写成的"奇书"的作者表示感谢，曾几何时自己在看到不同教材，辅导书的知识讲解时，对术解，都需要查阅更多其他资料，而缺乏系统性的整体了解，而断续的知识获取又往往在问题解决前出现新的问题，因此这本书的最终原因，希望与我有同样感受的人都能喜欢这本书。

大话计算机 2019-05-02 08:40 举报

旷世神作，此书已是顶峰，后无人可超越

写的太棒了

共三本 还在看第一本

五一几天 啃完前四章 看的不想放下来

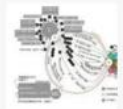
循序渐进 很棒

第三章 挺有意思 不愧是化学毕业 写的疏而不漏

不过还是烧脑的

详情

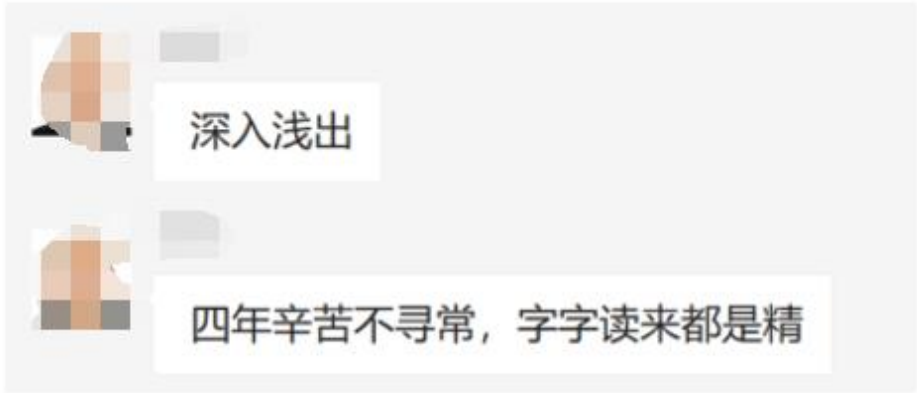
支持冬瓜哥，本科化学，没有读过研究生，纯靠兴趣，对计算机底层的理解和技术的追求已经到了一种洁癖，也是目前计算机著作的一股清流👍👍🥰🥰

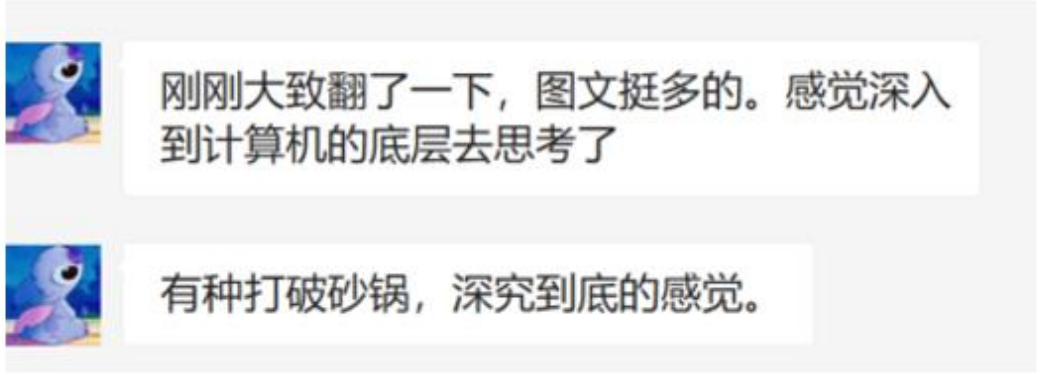

刺猬四年磨一剑 -- 冬瓜哥《大话...

昨天 12:58

13:01

冬瓜哥认识大约十年前，博导帮他第一本书写序，当时博士的时候博导经常拿他来鞭策我们说"你们这些计算机科班的难道还不如学化学的？一定要深入基层，理解原理"，好久没他声音了，这次四年磨一剑，终于出了重磅著作，强烈推荐😄





今天已经收到书了，回家迫不及待打开箱子，一个词形容“震撼”，震撼到难以形容，我对作者的才华和决心佩服至极！如果有机会我想请教作者，是什么支撑着您学习完这么多东西然后再消化、整理再写出这本巨作？我真的很难相信这本书会是一个人写出来的，不在于字数而在于知识面之宽广和深度。佩服至极！

2019-05-03 23:15:19

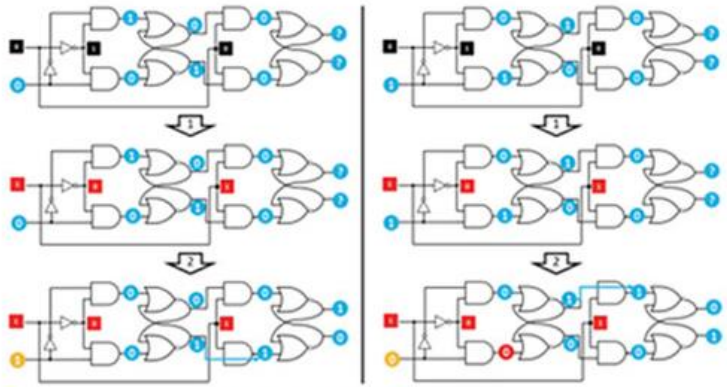


图1-27　D端为0和1时的锁存过程示意图

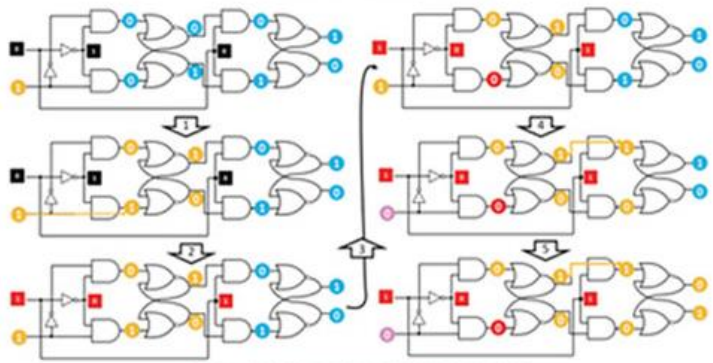


图1-28　连续输入两个新数据时依次锁定过程示意图

橙色信号透传到输出端。与此同时，紫色的新数据到达了D端，等待下一个时钟下沿将其传到中间，然后再来一个时钟上沿将其透传到输出端。

触发器就相当于照相机镜头（输入端）、快门（触发器电路中的与门）和底片（输出端）。时钟信号控制快门开合，时钟边沿的一瞬间，快门打开，镜头投射到快门上的影像会被瞬间透光锁定到底片上感光保存，在下一个时钟下沿之前，输出端信号不受输入端信号影响。

边沿触发器则更像是两道依次开合的闸门，两道闸门之间是一个临时等待区（图1-26中的A、B点）。如图1-29所示，时钟下边沿会将输入端信号传递到触发器内部，但并不输出；时钟上边沿则将上一步暂存在内部的信号输出，并在瞬间封闭输入端通路，直到下一个下边沿到来时，被堵在门口的新输入值才会进门在临时等待区等待。

综上所述，边沿型触发器电路仅当锁定信号从0变成1的瞬间，才能够锁住当前D的值并在Q端输出，后续不管是D变化，还是锁定信号从1变成0，Q值保持不变，所以被称为边沿型触发器。而之前的D触发器，称之为电平型触发器，因为其在0电平时锁闭，1电平时打开，边沿触发，这一点已经比较奇特了，但是这位神人还没有就此罢休，他调皮地将Q'端和D端连接了起来，并分析了电路的时序逻辑，发现了更奇特的结果。有时候真理只差一步，不要断然否认任何尝试，不要受那些事情肯定没意义、发现直觉的人。

个固定的位置先让程序读出来，然后程序就可以一层层地按照指针指向的路径继续读出数据。

图5-26中所示的“**超级块/根入口/根指针**”指的就是这个固定入口位置，超级块可以是一个扇区，或者多个连续的扇区组合起来（称为块），其中存放着一个或者多个根指针（也可以放最终内容，但是这里空间太小，放进去反而不划算，一般只放一些简要的信息比如系统版本号、最后访问时间等）。超级块内部的根指针指向另外的扇区或者块，这些被指向的扇区/块可以存放最终的数据内容，也可以继续存放指针，如果选择后者，则这些位于根指针下一层的存放指针的扇区/块被称为**二级指针**块。指针块内也可以既包含实际文件内容又包含指向下游数据块的指针。二级指针可以继续指向三级指针块或者混存块。同理，也可以有更多级的指针块。最后一级指针块会指向纯用于存放最终数据内容的块，公示表里那些文件名、起始扇区号之类的信息，最终会被存放于此。

按照上述的思想，我们可以将文件系统改造为如图5-27所示的这番景象，这样是不是更壮观了？其实这只是最简易的设计，现实中的文件系统设计远比这复杂。当然，功能、可靠性、性能也都更强。

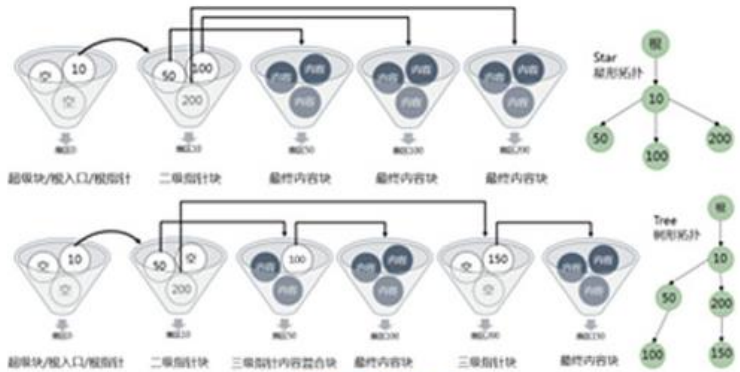


图5-26　星形和树形拓扑

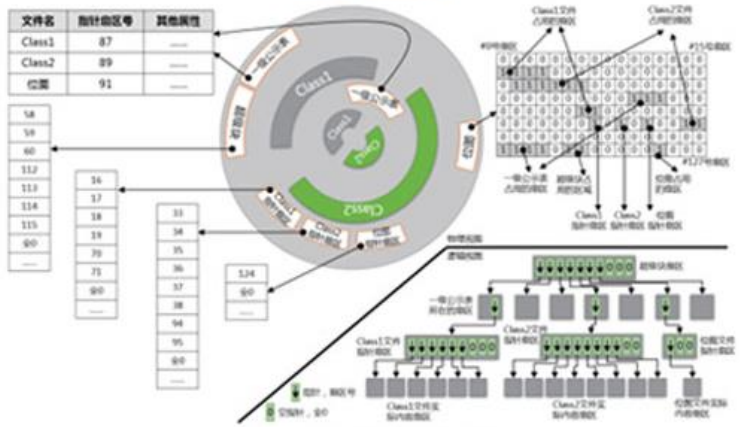


图5-27　一种改良版的简易文件系统设计

图5-66　链接符号定义和引用的最新位置更新符号表和重定位表中的地址列

将对应的目标文件提出（或者直接将多个独立的目标文件融合起来），并按图索骥，将需要被链接到一起的目标文件中的代码融合起来，然后根据地址重定位表里描述的信息，将需要重定位的地方根据融合进来的目标函数/变量当前所处的位置，修正之前被编译器所放置临时地址。这个地址修正过程被称为**地址重定位**过程。链接器最终会生成一个可执行的完整的程序文件（比如后缀名为.out的文件，或者.exe的文件。当然，你可以改成任意后缀名，这不重要）。

也就是说，在源文件级别进行融合，和在编译好的机器码级别进行融合，最终结果都一样。对于源码级别的链接，在源文件中采用#include的编译制导语句即可。相应地，对于二进制机器码级别的链接（多个目标文件链接），很显然，你也需要告诉链接器你要将哪个目标文件和哪个库文件/目标文件链接在一起（准确来说，你必须知道某个目标文件中的代码可能调用了某个库文件/目标文件中的某个/些函数，并告诉连接器，链接器只是将库文件中被调用到的函数名所对应的目标文件抽出，并与被链接的目标文件进行链接，而不是链接整个库文件），而这时就不可能用#include了，而是直接在调用链接器程序的时候，将目标文件名/库文件名作为参数输入给它即可。比如Linux操作系统下的ld程序就是一个链接器程序，ld 1.o 2.o 3.lib 4.o –o main.exe命令的作用就是将这一大堆的目标文件/库文件链接融合，最后生成一个main.exe可执行程序，命令行直接敲./main.exe便可以执行。（如果ld命令不加-o参数指定输出的文件名的话，默认会输出一个名为a.out的可执行文件，./a.out一样可以执行它。a.out的意思是Assembler Output（编译输出）的意思。）

注意

值得注意的是，假设1.o调用了2.o中的函数A，而A里面又调用了位于3.o中的函数B，B则没有再次调用外部函数，那么此时你必须将1/2/3.o这三个文件都给出，链接才能完成。这在实际中是非常普遍的现象。所以，我们一直没有回答的一个问题：为何要将多个.o文件再次打包成库文件呢？此时你应该知道答案了，如果能够将有级连或者相互引用关系的多个库文件打包成一个文件，那么链接的时候只需要给出这个文件名即可，而不需要把所有相关的目标文件名都列出来。

然而从图5-67中可以看到，main()函数的第一句代码Load_a 26 A是该程序的第一句代码，程序必须从这里开始执行。然而，该句代码却被放到了地址18上。我们前文中一直在这样假设：把程序放到存储器的第一行上，第0行放一条NOOP指令，我们设计的这个CPU加电之后，用NOOP指令避免了初始状态的不确定性，然后从第1行开始执行代码。那么说，这个程序就无法被运行了么？想一想都有哪些方法可以让该程序继续运行？

（1）真是闲的没事了，为什么不把main()放到程序的第一行呢？把main()的位置定死，必须从第一行执行。可以这样做。

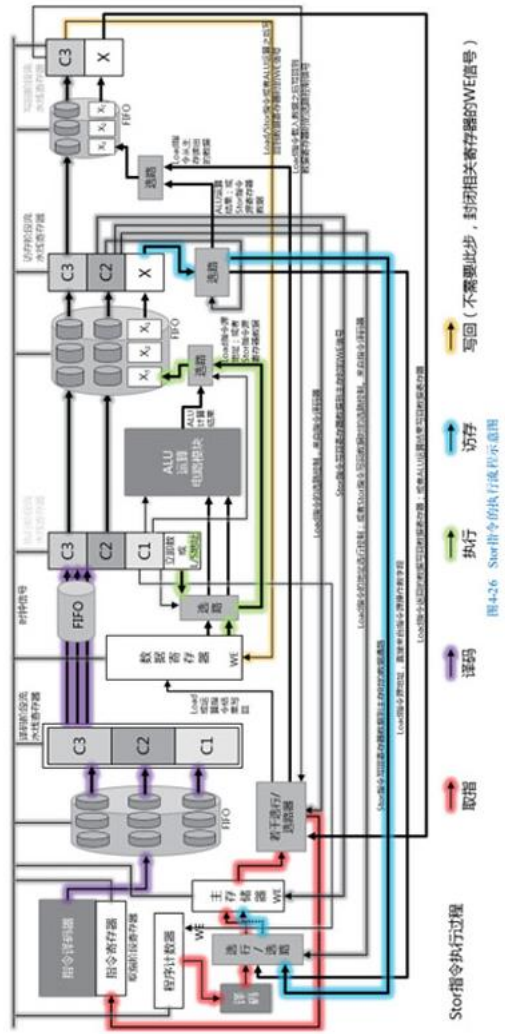


图4-26　Stor指令的执行流程示意图

须将所有Way中的对应行号的数据都选出来输送到Mux从而等待被选出。如果能够先查出来某个地址命中在哪个Way，第二步只从该Way读出数据。另一个Way保持状态不变，这样就可以节省功耗。可以回顾一下第3章中的SRAM电路结构，只要控制读写的开关打开了，bitline就会在逻辑的驱动下进行充放电，从而输出对应的电压。这就是动态功耗。如果不读取它，开关关闭，对应的bitline就不会被充放电，只维持原来的电压，并存在一定的微小漏电流。这就是静态功耗。

于是，有些设计采用了冬瓜哥又花了2个小时瞎琢而成的图6-20所示的方式。将Tag部分拿出来放到

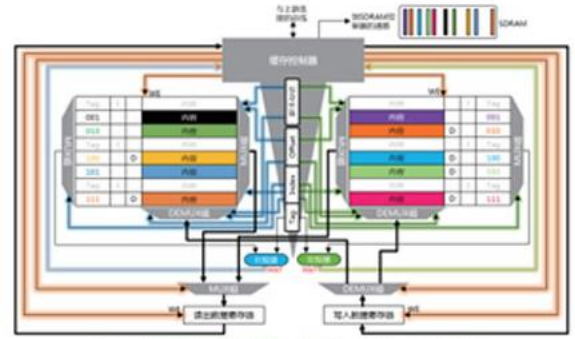


图6-19　2路组关联缓存的硬件架构示意图（Hit：命中；WE：写使能）

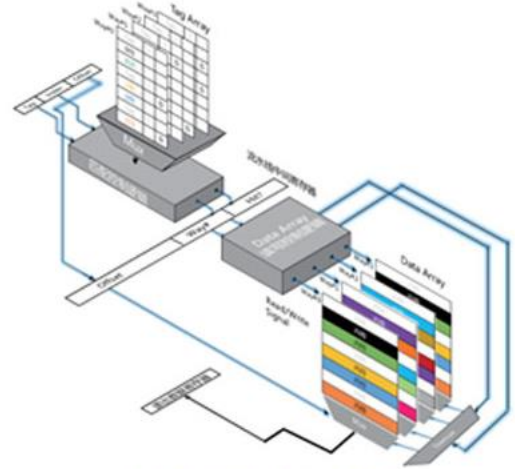


图6-20　分两步访问多路组关联缓存节省功耗

怎么样，是不是比较复杂，但是只要顺藤摸瓜，有了这套状态机，每个核心都可以感知到自己以及他人的状态。虽然无法做到全面量化感知（比如Shared状态，并不知道到底有几个核心、各是谁持有该行），但是每个核心依然可以根据自身缓存行的状态来决定自己的动作，在保证了一致性的同时，还能避免不必要的总线流量。

我们用另外一种图示来展示上述的整个变迁过程。如图6-143所示。观此图时，脑海中可以构建一副场景：这些CC消息在访存网络上错综复杂地高速流动着，每个核心的状态不停闪烁变化着，但是闪烁的速度低于消息流动速度；核心之上，程序的代码一条条地被载入，程序执行的速度又慢于核心状态变化的速度。这就犹如现实世界的场景，最底层的基本构建单元不停地振荡，多个单元通过某种关系耦合起来形成基本粒子，再加上反馈，让这些单元感知到时间的存在。基本粒子再耦合成更高级的机构。底层的基本单元振荡速度为世界最高速度，越往上层速度逐渐降低。上层的一个小变化，对应着底层的大量变化的堆积叠加和反馈。

另外可以看到，从S到E是没有变更路径的，这是因为CPU核心无法感知到自己是否应该从S变更到E。比如，某时刻多个CPU持有处于S态的缓存行，然后其他CPU陆续将该行从缓存中换出，只剩下一个CPU拥有该行，那么该CPU对应的该行状态依然是S。虽然从旁观者视角而言它现在就是E态了，但是由于CPU在换出S态缓存行时为了节省流量并不广播消息通告，所以该CPU的该缓存行因为无法感知到这一点，仍为S态。

提示

如果核心把缓存副除/淘汰的动作也使用消息广播出去，那么就可以实现从S态到E态的状态转换了。但是，这种实现代价太高。因为每个CA均需要一个bitmap来追踪到底哪个核心把这一行换出了（把相应位置为0），以及哪个核心又读入这一行了（把相应位置为1）。如果系统内有4个核心，则需要3位来表示，当3位都置为1后，便可以通知到E态了。E态的好处是自己要改的话直接更改，不需要发送Wrtlvld消息，但是为了节省这一次消息，平时却要做更多工作，得不偿失。

Intel的主流商用CPU使用的就是MESIF缓存一致性协议。

提示

假设缓存行容量为64字节，两个核心持有S态的同一行缓存，核心1更改了该行的前8字节，它会将核心2的该行整行作废，即使核心2上的程序只访

图6-143　经典的MESIF状态机

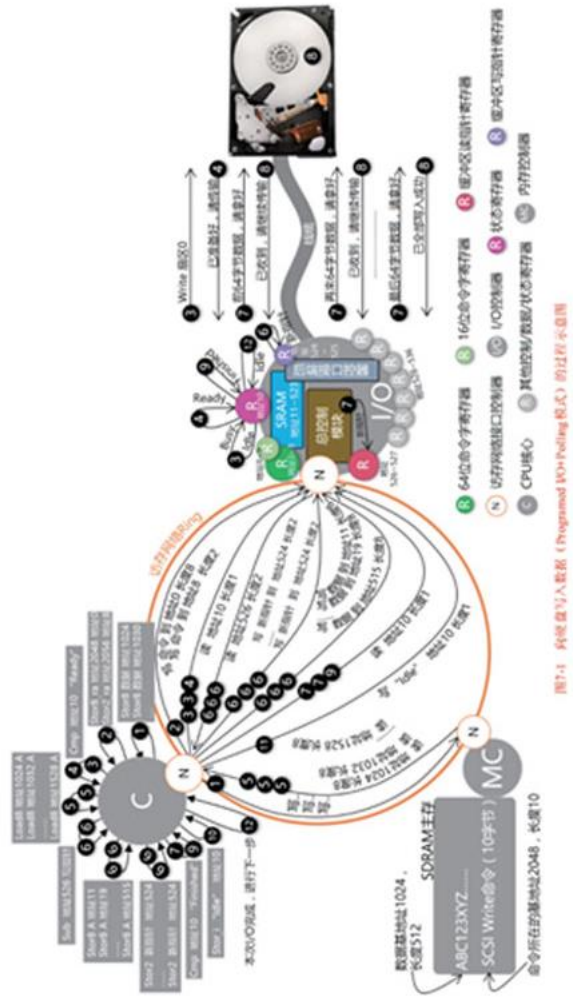


图7-1　向硬盘写入数据（Programmed I/O+Polling模式）的过程示意图

示I/O控制器等。至于那些不太常用的，比如摄像头I/O控制器、10GB以太网控制器、SAS控制器、高端显示控制器等，就不集成到I/O桥内部了，否则成本太高。如果想用上面这些设备，可将它们统一接入到I/O总控制器上。那么，I/O控制器又是怎么与I/O总控制器连接起来的呢？

I/O总控制器通过单独的总线或者某种形式的网络与后面这些I/O控制器连接起来，如图7-14所示。这些连接I/O总控制器与I/O控制器的总线/网络有多种，比如PCI、PCI-X、PCI-E，以及ARM CPU平台常用的AXI等。其中，PCI、PCI-X属于共享总线类型，PCI-E属于交换网络类型。

要做到外部设备的灵活增添，一个合理的办法就是将I/O控制器做成可插拔的单独的硬件单板，采用物理连接器（比如插槽形式）与I/O桥输出的连接器进行对接。I/O总控制器预留一定数量的插槽，就可以让用户灵活添置各种I/O控制器单板。图7-14右上角所示为一片SAS I/O控制卡，其采用PCIE连接器插入I/O总控制输出的PCIE插槽。散热片之下是一片SAS I/O控制器芯片，其前端通过与I/O总控制器相同的总线/网络类型对接，后端则利用SAS接口控制器与其他SAS设备连接，比如SAS硬盘。图7-14中右下角所示为SAS体系的逻辑示意图。通常，服务器类计算机的制造商会在机器箱体内放一块**背板**（图中右上角所示，正反面），背板上焊有一片SAS交换器，所有SAS硬盘先连接到SAS交换器，再通过SAS线缆上连到SAS I/O控制器。从这一点上讲，SAS这个网络也是利用复用的方式将多个SAS设备连接到同一个SAS I/O总控上。

如果抛开实际的物理实体，抽象来看的话，计算机I/O系统在本质上其实是多个不同网络之间的互联，如图7-15所示。图中的每种颜色覆盖的区域都各成一派，是各自独立运行的网络/总线，而各个I/O控制器充当了路由器的角色，其从后端端口上利用后端口所在的网络所使用的交互方式（协议）将数据接收到，然后再从前端端口利用前端口网络所使用的协议将数据发送给上游部件，从前端口接收数据发送到后端口的过程也类似。当然，这些I/O控制器比单纯的

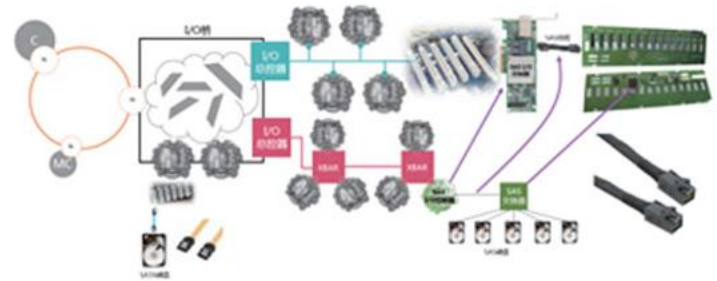


图7-14　利用I/O总控制器接入更多I/O控制器

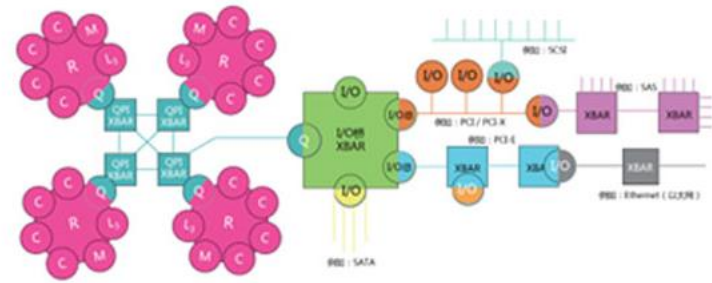


图7-15　计算机I/O本质上是不同网络的互联和协议的转换

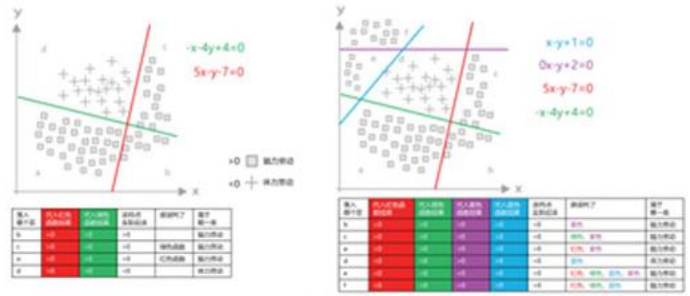


图12-7　两个分类场景下的结果总结

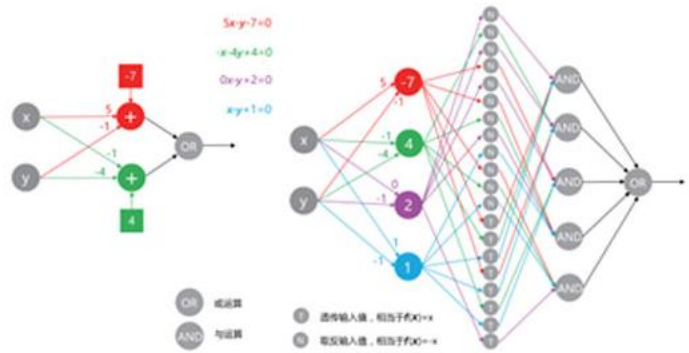


图12-8　对应图12-7中场景的逻辑表达图

除了把事情表达成与或非的关系之外，还可以从另外的角度和方法来入手，从而可以实现更多效果和结论。

如图12-9所示，还是刚才的场景，c区被绿色函数误判，但是对于蓝色函数，c区的判定全部正确，如果以某个样点距离函数直线的绝对距离值作为该样点的分类可能性的话，那么越远离直线越可能正确或不正确。对于c区中的蓝色样点，蓝色函数说该样点是脑力劳动者，可能性假设为1，但是绿色函数说该样点是脑力劳动者的可能性为-5，到底该听谁的？如果**两者都得听**的话，那就**将两者输出的可能性值相加**：1+(-5)=-4，结果显示该点不是脑力劳动者，但是开启上帝视角之后可以看到蓝色样点的确为脑力劳动者，应该听蓝色函数的，但是无奈蓝色函数判定该样点是脑力劳动者的可能性低得可怜，只有1，压不过-5，那怎么办？

好办，我们给蓝色函数一个较高的权重，也就是直接将它乘以10，再与绿色函数相加，如图12-9中间所示，这下好，10+(-5)=5，脑力劳动者！让你嚣张！蓝色函数有了特权，就可以在c区无视一切其他可能性值的贡献度，对于橙色样点，其本来的可能性就已经很高了，已经压过绿色的反对声音了，被放大10倍之后，就更高了，但是回过头来看看a区，蓝色曲线本来就完全误判了a区中所有样点，认为其为脑力劳动者的可能性都<0，现在被放大了10倍，相当于把错误也放大了10倍，而绿色函数的正确声音在a区被完全压制了，a区彻底沦陷！

那好，既然蓝色函数不给力，把绿色函数放大10倍看看，同样的事情发生了，a区收复失地，但是c区

四大理由值得拥有

很专业但能看懂

作者基于多年半导体从业的思考与沉淀，以独特视角切入缤纷计算机世界，用接地气的语言、华丽精准的图表纵情解读。

精细豪华解读

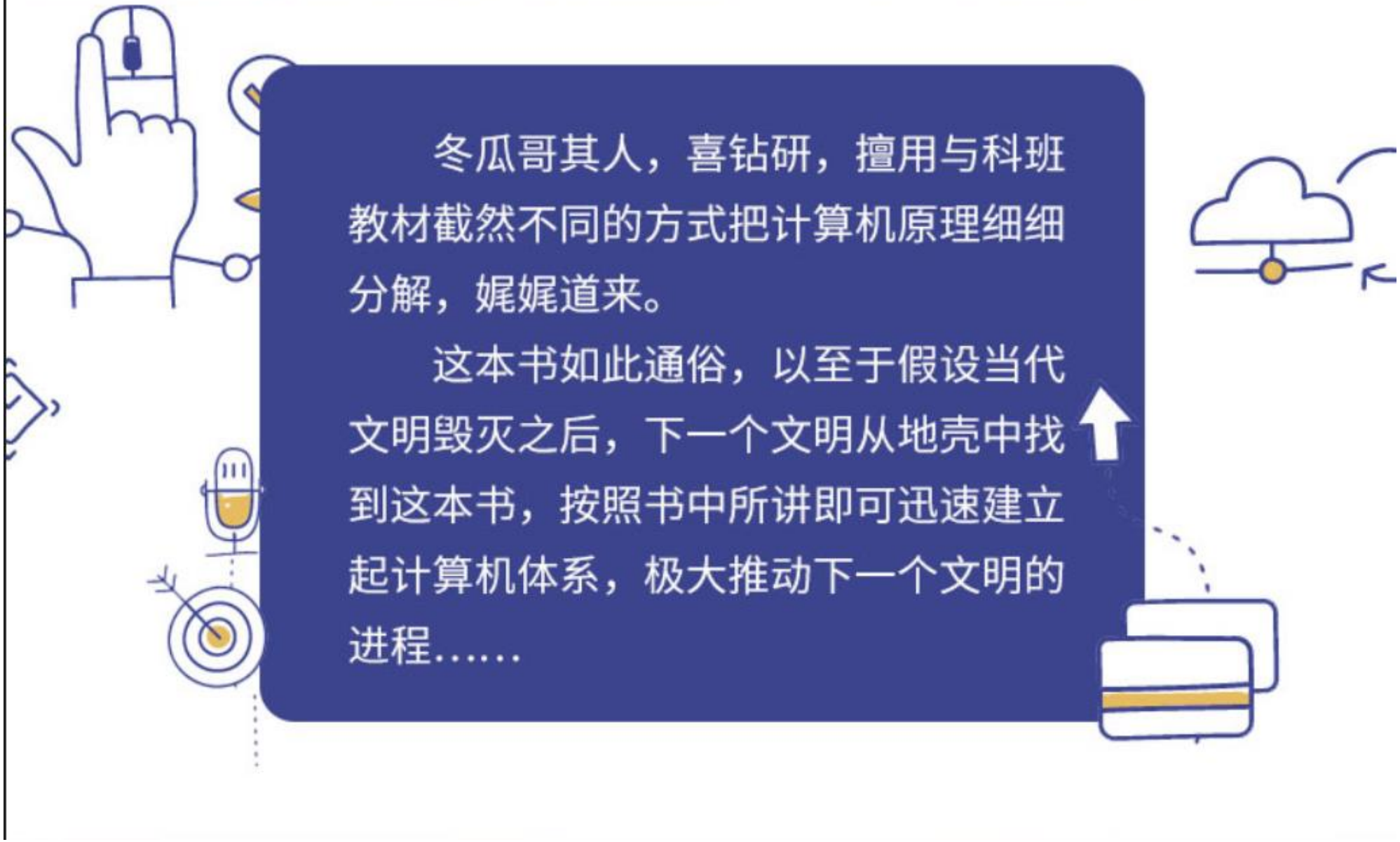
三册精装，超过1500页，超过350万字，超过500张原创图表，11张8开思维导图海报，大量二维码引入图片、音视频……堪称计算机原理饕餮盛宴。

卓越阅读体验

全彩印刷+温婉柔和的纸张+8色顶配印刷机+平铺桌面不回弹的精装书。

再造IT基本功

深入研磨计算机系统，你的软硬件开发思维、AI路线、算法思路，甚至信息安全、芯片制造等领域，一定会出现让你自己大吃一惊的感悟和火花，你的收获将远远超过图书定价。



冬瓜哥其人，喜钻研，擅用与科班教材截然不同的方式把计算机原理细细分解，娓娓道来。

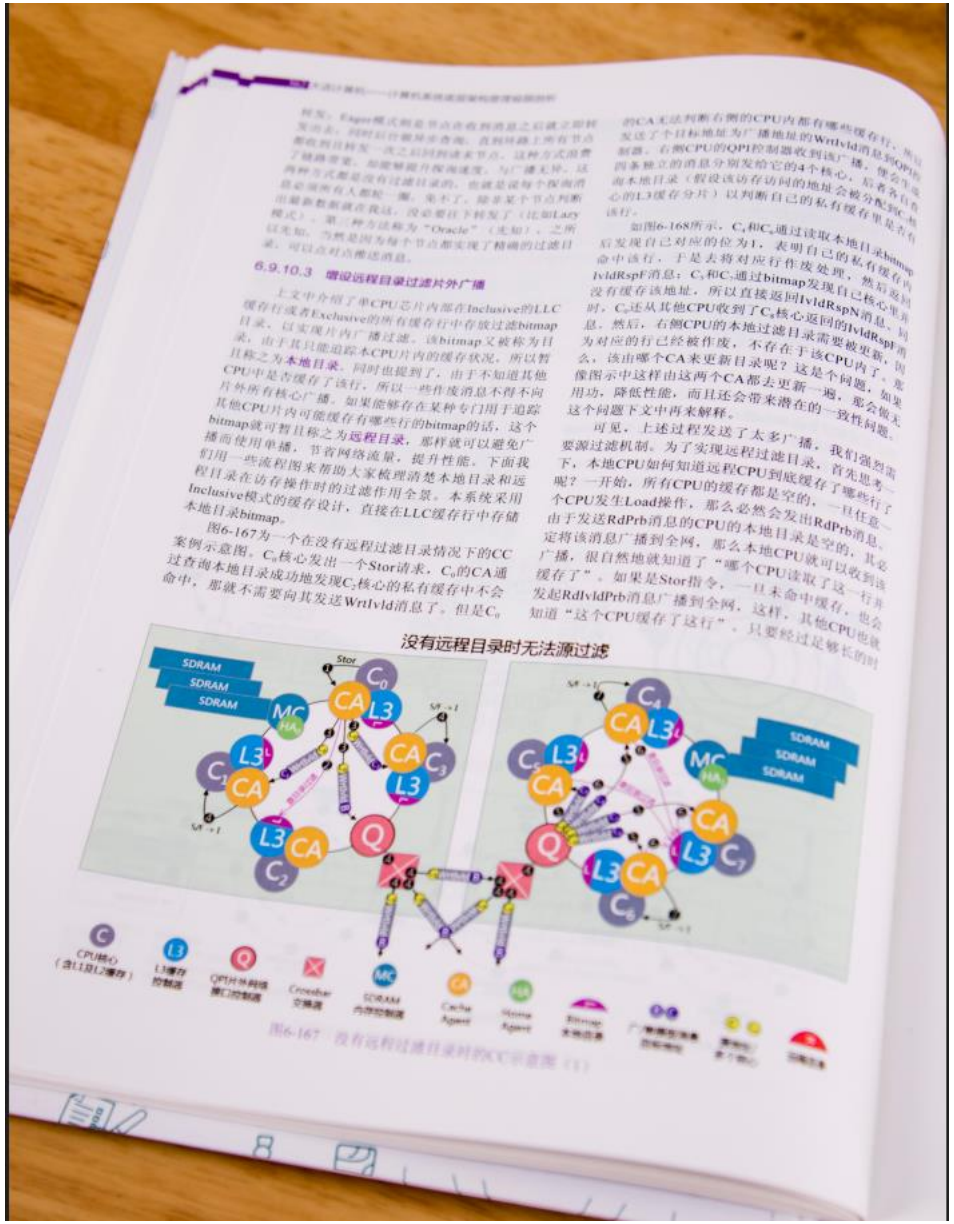
这本书如此通俗，以至于假设当代文明毁灭之后，下一个文明从地壳中找到这本书，按照书中所讲即可迅速建立起计算机体系，极大推动下一个文明的进程……


冬瓜哥◎著
大话计算机
计算机系统底层架构原理极限剖析
清华大学出版社


卷2
大话计算机
冬瓜哥◎著
CONCEPT
计算机系统
底层架构原理极限剖析
清华大学出版社


大话计算机


大话
计算机
冬瓜哥◎著
计算机系统
底层架构原理极限剖析
北京大学出版社


SDRAM
SDRAM
SDRAM
Stor
CA
L3
MC
Q
CPU核心
（含L1及L2缓存）
L3缓存
控制器
QPI片外网络
接口控制器
Crossbar
交换器
SDRAM
内存控制器
Caching
Agent
Home
Agent

转发；Eager模式则是节点在收到消息之后就立即转发出去，同时后台做异步查询，直到环路上所有节点都收到且转发一次之后回到请求节点，这种方式浪费了链路带宽，却能够提升探询速度，与广播无异。这两种方式都是没有过滤目录的，也就是说每个探询消息必须所有人都轮一圈，免不了，除非某个节点判断出最新数据就在我这，没必要往下转发了（比如Lazy模式）。第三种方法称为“Oracle”（先知），之所以先知，当然是因为每个节点都实现了精确的过滤目录，可以点对点推送消息。

6.9.10.3 增设远程目录过滤片外广播

上文中介绍了单CPU芯片内部在Inclusive的LLC缓存行或者Exclusive的所有缓存行中存放过滤bitmap目录，以实现片内广播过滤。该bitmap又被称为目录，由于其只能追踪本CPU片内的缓存状况，所以暂且称之为**本地目录**。同时也提到了，由于不知道其他CPU中是否缓存了该行，所以一些作废消息不得不向片外所有核心广播。如果能够存在某种专门用于追踪其他CPU片内可能缓存有哪些行的bitmap的话，这个bitmap就可暂且称之为**远程目录**，那样就可以避免广播而使用单播，节省网络流量，提升性能。下面我们用一些流程图来帮助大家梳理清楚本地目录和远程目录在访存操作时的过滤作用全景。本系统采用Inclusive模式的缓存设计，直接在LLC缓存行中存储本地目录bitmap。

图6-167为一个在没有远程过滤目录情况下的CC案例示意图。C_0核心发出一个Stor请求，C_0的CA通过查询本地目录成功地发现C_2核心的私有缓存中不会命中，那就不需要向其发送WrtIvld消息了。但是C_0的CA无法判断右侧的CPU内都有哪些缓存行，所以发送了个目标地址为广播地址的WrtIvld消息到QPI控制器。右侧CPU的QPI控制器收到该广播，便会生成四条独立的消息分别发给它的4个核心，后者各自查询本地目录（假设该访存访问的地址会被分配到C_4核心的L3缓存分片）以判断自己的私有缓存里是否有该行。

如图6-168所示，C_4和C_6通过读取本地目录bitmap后发现自己对应的位为1，表明自己的私有缓存内命中该行，于是去将对应行作废处理，然后返回IvldRspF消息；C_1和C_3通过bitmap发现自己核心里并没有缓存该地址，所以直接返回IvldRspN消息。同时，C_0还从其他CPU收到了C_4核心返回的IvldRspF消息。然后，右侧CPU的本地过滤目录需要被更新，因为对应的行已经被作废，不存在于该CPU内了。那么，该由哪个CA来更新目录呢？这是个问题，如果像图示中这样由这两个CA都去更新一遍，那会做无用功，降低性能，而且还会带来潜在的一致性问题。这个问题下文中再来解释。

可见，上述过程发送了太多广播，我们强烈需要源过滤机制。为了实现远程过滤目录，首先思考一下，本地CPU如何知道远程CPU到底缓存了哪些行了呢？一开始，所有CPU的缓存都是空的，一旦任意一个CPU发生Load操作，那么必然会发出RdPrb消息。由于发送RdPrb消息的CPU的本地目录是空的，其必定将该消息广播到全网，那么本地CPU就可以收到该广播，很自然地就知道了“哪个CPU读取了这一行并缓存了”。如果是Stor指令，一旦未命中缓存，也会发起RdIvldPrb消息广播到全网，这样，其他CPU也就知道“这个CPU缓存了这行”。只要经过足够长的时间，就可以积累足够多的追踪记录，从而逐渐实现过滤的目的。这个过程，称之为目录的**预热**过程。

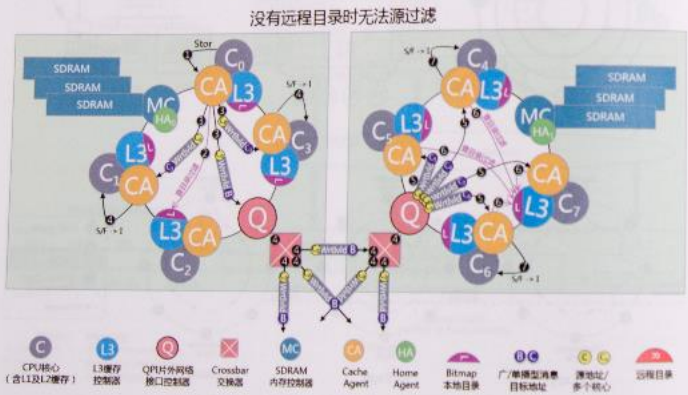


图6-167 没有远程过滤目录时的CC示意图（1）

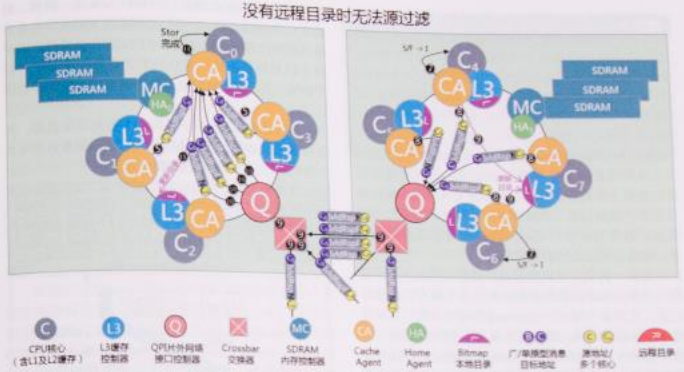


图6-168 没有远程过滤目录时的CC示意图（2）

那么，这个远程追踪目录需要存储什么信息呢？一个bitmap向量是必须的，bitmap中位的数量就是系统中所有远程核心的数量（本地核心就不需要在远程目录里追踪了，有本地LLC目录）。另外，由于追踪的是远程CPU内部可能缓存的行，对应的行可能在本地并没有缓存在LLC中，所以欲将bitmap存在本地LLC对应行中就不可能了。必须单独开辟一块高速存储器空间，存储对应行的地址，对应行的bitmap向量。如果要更加精细的记录，比如不仅仅要记录对应行是否存在于某个核心缓存里，还想知道对应行目前的MESIF状态，那就还必须记录一个状态字段，比如，只要收到对方的WrtIvld消息，那么本地就可以判断对方缓存中的该行一定处于M态了。

缓存行地址需要几十位，比如64位地址，缓存行大小64字节（需要6位来编址），那么缓存行地址长度就为64-6=58位，再加上3位的MESIF状态位，以及bitmap向量的16位（假设系统为4个4核心CPU），则这一条追踪记录就得至少78位大小。如果要追踪所有的缓存行，也就是2^{28}条记录，再乘以78位，其容量将会非常大。所以，该追踪目录势必要做出类似缓存和RAM之间无法全包含时一样的处理，比如直接相连方式，能省掉一些地址位，这样多条追踪记录会相互挤占同一个位置。还需要再次比对缓存行的物理地址Tag才能判断是否该记录有效，如果发现不匹配，则认为未过滤成功，对应的广播必须发出。

图6-169为一条记录项的示意图。思考一下，某个变量并不是任何时候都会被系统中所有核心缓存的。假设系统共有4个核心，那么，某个变量被缓存的场景共可分为：只被1个核心缓存（共4种组合），被2个核心缓存（共6种情况），被3个核心缓存（共4种情况），被全部4个核心缓存（共1种情况），总共15种状态。系统运行的时候，根据不同程序场景而定，在这15种情况中，出现比例最大的可能只有少数几种，我们假设为8种，分别为：被核心1缓存、被核心2缓存、被核心3缓存、被核心4缓存、被核心2/4缓存、被核心1/3缓存、被核心1/2/3缓存、被核心2/3/4缓存。那么，我们只需要存储3位就可以描述这8种状态。每次CA根据接收到的CC消息需要更新该条目时，电路先将当前存储的3位根据上述8种规定译码展开之后输入到一个4位逻辑电路，然后再把本次需要修改的位合入到这4位中，再通过收缩译码器将这4位译码成上述8种情况中的一种，如果本次修改的位恰好没有匹配上述8种场景中的任何一种，比如本次合入之后结果是“被核心3/4缓存”，我们称之为**溢出**。此时，译码器被设计为输出一个溢出信号，然后，电路可以决定，是将该条目直接作废（因为它已经无法反映真实情况了），还是动态在线地变更译码策略，后者这种做法更加智能，但是其需要该译码器是可配置的，可以根据寄存器中的控制字决定如何译码，比如在上述情况中，可以通过变更控制器寄存器内的控制位，让译码器改为可以将“被核心3/4缓存”（对应4位展开bitmap值0011）翻译成比如“010”，从而替代上述规定中的一种。下次如果又变化了，那就再替代一种。

Tag	Status	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b

图6-169 目录记录项的示意图

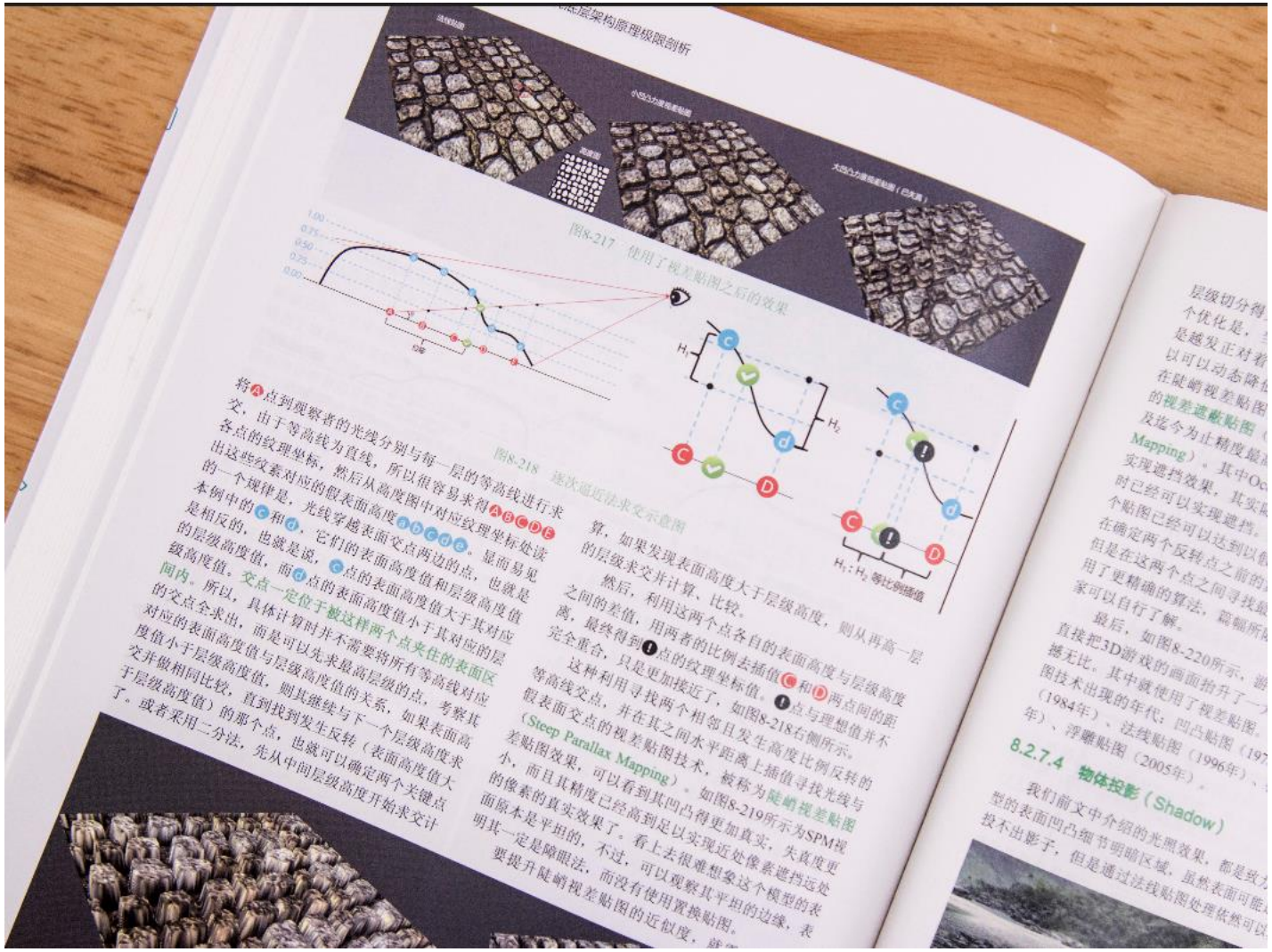


图8-217 使用了视差贴图之后的效果

图8-218 逐次逼近法求交示意图

将A点到观察者的光线分别与每一层的等高线进行求交，由于等高线为直线，所以很容易求得ABCDE各点的纹理坐标，然后从高度图中对应纹理坐标处读出这些纹素对应的假表面高度abcde。显而易见的一个规律是，光线穿越表面交点两边的点，也就是本例中的C和d，它们的表面高度值和层级高度值是相反的，也就是说，C点的表面高度值和层级高度值的层级高度值，而d点的表面高度值大于其对应的层级高度值。所以，交点一定位于被这样两个点夹住的表面区间内。所以，具体计算时并不需要将所有等高线对应的交点全求出，而是可以先求最高层级的点，考察其对应的表面高度值与层级高度值的关系，如果表面高度值小于层级高度值，则其继续与下一个层级高度求交并做相同比较，直到找到发生反转（表面高度值大于层级高度值）的那个点，也就可以确定两个关键点了。或者采用二分法，先从中间层级高度开始求交计算，如果发现表面高度大于层级高度，则从再高一层的层级求交并计算、比较。

然后，利用这两个点各自的表面高度与层级高度之间的差值，用两者的比例去插值C和D两点间的距离，最终得到I点的纹理坐标值。I点与理想值并不完全重合，只是更加接近了，如图8-218右侧所示。

这种利用寻找两个相邻且发生高度比例反转的等高线交点，并在其之间水平距离上插值寻找光线与假表面交点的视差贴图技术，被称为陡峭视差贴图（Steep Parallax Mapping）。如图8-219所示为SPM视差贴图效果，可以看到其凹凸得更加真实，失真度更小，而且其精度已经高到足以实现近处像素遮挡远处的像素的真实效果了。看上去很难想象这个模型的表面原本是平坦的，不过，可以观察其平坦的边缘，表明其一定是障眼法，而没有使用置换贴图。

要提升陡峭视差贴图的近似度，就

层级切分得
个优化是，
是越发正对着
以可以动态降
在陡峭视差贴图
的视差遮蔽贴图
及迄今为止精度最
Mapping）。其中Oc
实现遮挡效果，其实
时已经可以实现遮挡
个贴图已经可以达到以
在确定两个反转点之前的
但是在这两个点之间寻找
用了更精确的算法，篇幅所
家可以自行了解。

最后，如图8-220所示，
直接把3D游戏的画面抬升了一
搪无比。其中就使用了视差贴图
图技术出现的年代：凹凸贴图（
（1984年）、法线贴图（1996年）、
年）、浮雕贴图（2005年）。

8.2.7.4 物体投影（Shadow）

我们前文中介绍的光照效果，都是致
型的表面凹凸细节明暗区域，虽然表面可能
投不出影子，但是通过法线贴图处理依然可以

尾声 狂想计算机——以

地安 上的某个氨基酸残基上的电化学基团产生电动力，从
组成 而驱动蛋白质分子上的具有钾离子选择性的分子马达
A分子 的ATP结合点（油门）暴露，ATP一拥而上将其旋转起
来，从一侧吸引钾离子，转到另一面，释放钾离子。

但是模拟量底层并不是无限连续的，其最小单元
体内变 是一个正弦波形状。那么，一种可能的模型是：细胞
合点被封 一侧的钾离子浓度如果较高，那么该蛋白质分子在这
于某种原 一侧的某些残基上的能够与钾离子相互产生物理电作
导致马达 用或者化学力作用的基团每次受力都会对整个蛋白质
量，成为垃 分子深处的各个原子进行牵拉，当钾离子浓度较高的
结构、这么 时候，这种被牵拉的频率就会越高，当牵拉频率达到
自然生成的。 某个值的时候，引起分子的构象产生一个比较大的跃
的是靠进化自 变，将分子马达的ATP结合点暴露从而转运离子。当
规律在支配， 钾离子浓度不够的时候，阈值频率未达到，分子马达
部，必然会产生 停止工作。随着钾离子被运输到另一侧，本侧浓度持
规律已经是与数 续降低，最后形成一个闭环的负反馈控制系统。
灵曾经尝试用机 蛋白质分子就是利用原子来搭建的精密机械，
现的。分形理论也 是造物者的杰作。分子马达犹如热机一样，也分多
复的单元可以搭建 个冲程，比如结合ATP分子时，构象改变，导致蛋白
命。 质转子旋转一定角度，ATP变为ADP后，转子再次旋
系统的话，那么这 转一定角度，ADP被释放后，转子再次旋转。这就
细胞表面的蛋白质分 像热机的吸气、压缩、爆炸、排气
一次系统调用。而按照 这四个冲程不断循环一样。而蛋白
氨基酸化合成肽链，最 质分子马达纯粹依靠由ATP分子形成
质分子的过程，又好像 的分子间作用力来拉动整个转子旋
ary()从可执行文件中将程 转。扫描二维码观看蛋白质分子马
子生物学就是一场没有源 达的冲程动画。

一切全靠实验。 图5所示其实是一个ATP生成器。对于电动机来
处是，其几乎无所不能，既 讲，通电可以让转子转动，但是如果让转子转动，则
当离子通道，还可以作为在 可以反过来产生电流。对于蛋白质分子马达也是一样
的泵，如图4所示。比如钠钾 的，ATP生成器也是一个分子马达，其在细胞膜内的
时，其可以将细胞外的钾离子 区域是一个转子，转子表面包含多个质子结合点，可
钠离子运到胞外。那么，钠钾 以利用高酸性环境驱动转子转动，转子转动导致胞外
度差来的呢？ 部分构象形变，从而将ADP+Pi合成为ATP，这相当于
用电子计算机来模拟的话， 发电机。

每秒采样到的 对于图4左侧，诗意一些的表达则是：我要送你
然 一棵生命之树，象征着你我共同的目标。树根之下
这是能量的源泉。质子驱动着马达
力把ADP和Pi强行结合成
好吧，编


大话计算机
计算机
清华大学出版社


大话
计算机
冬瓜哥◎著
计算机系统
底层架构原理极限剖析
CONCEPT
清华大学出版社


冬瓜哥◎著
大话
计算机
计算机系统
底层架构原理极限剖析
清华大学出版社


冬瓜哥


大话
计算机
Enjoy!
大话计算机
大话计算机
大话计算机
大话
计算机
计算机系统


Enjoy!
大话
计算机


大话计算机
大话计算机
大话计算机

01-《大话计算机》开篇-苦想计算机-2017-02-14.docx
02-《大话计算机》第1章-数字电路-2016-10-20-102.docx
03-《大话计算机》第2章-简易CPU-2016-11-12-53.docx
04-《大话计算机》第3章-计算机硬件史-2016-11-14-136.docx
05-《大话计算机》第4章-流水线与执行优化-2016-12-21-54.docx
06-《大话计算机》第5章-程序世界-2017-04-06-106.docx
07-《大话计算机》第6章-处理器微体系结构2017-07-15-182.docx
08-《大话计算机》第7章-计算机IO系统-2017-08-5-221（待加入sa
09-《大话计算机》第8章-声音图象-2017-10-10-197 .docx
10-《大话计算机》第9章-并行计算2016-11-19-71.docx
13-《大话计算机》后记-写作经历-2016-11-28.docx
11-《大话计算机》第10章-现代计算机形态与生态-2016-11-19.docx
11-《大话计算机》第10章-现代计算机形态与生态-2016-11-19.pptx
12-《大话计算机》尾声-狂想计算机-2016-11-19.docx

第2章

- 第5章
  - 5.1 基本的数据结构
    - 5.1.1 数组
    - 5.1.2 数据类型与ASCII码
    - 5.1.3 结构体
    - 5.1.4 数据怎么摆很重要
  - 5.2 高级语言
    - 5.2.1 简单的声明和赋值
    - 5.2.2 编译和编译器
    - 5.2.3 向编译器描述数据的编排方式
    - 5.2.4 高级语言编程小试牛刀
    - 5.2.5 人脑编译忆苦思甜
  - 5.3 浮点数及浮点运算
    - 5.3.1 数值范围和精度
    - 5.3.2 浮点数的用处和表示方法
    - 5.3.3 浮点数的二进制表示
      - 5.3.3.1 二进制浮点数转十进制小数
      - 5.3.3.2 十进制小数转二进制浮点数
      - 5.3.3.3 负指数和0的表示
      - 5.3.3.4 无穷与非规格化数的表示
    - 5.3.4 浮点数运算挺费劲
    - 5.3.5 浮点数的C语言声明
    - 5.3.6 十六进制表示法
  - 5.4 程控多媒体计算机
    - 5.4.1 键盘是前提
    - 5.4.2 搜索并显示
    - 5.4.3 实现简易计算器
    - 5.4.4 录入和保存
    - 5.4.5 简易文件系统
    - 5.4.6 计时/定时
    - 5.4.7 发声控制
    - 5.4.8 图像显示
    - 5.4.9 网络聊天
  - 5.5 程序社会
    - 5.5.1 函数和调用
    - 5.5.2 设备驱动程序
    - 5.5.3 函数之间的联络站
    - 5.5.4 库和链接
      - 5.5.4.1 静态库和静态链接
      - 5.5.4.2 头文件
      - 5.5.4.3 API和SDK
      - 5.5.4.4 动态库和动态链接
      - 5.5.4.5 库文件/可执行文件的格式
    - 5.5.5 程序的执行和退出
      - 5.5.5.1 初步解决地址问题
      - 5.5.5.3 更好的人机交互方式
      - 5.5.5.4 程序的退出
      - 5.5.5.5 使用外部设备和内存
    - 5.5.6 多程序并发执行
      - 5.5.6.1 利用时钟中断来切换线程
      - 5.5.6.2 更广泛的使用中断
      - 5.5.6.3 虚拟地址空间与分页
      - 5.5.6.4 虚拟与现实的边界——系统调用
    - 5.5.7 呼唤操作系统
  - 5.6 计算机操作系统
    - 5.6.1 内存管理
      - 5.6.1.1 分区式内存管理
      - 5.6.1.2 分段+分区式内存管理
      - 5.6.1.3 Intel CPU的分段支持机制
      - 5.6.1.4 分页+分段式内存管理
      - 5.6.1.5 DOS和Linux操作系统的内存管理
    - 5.6.2 中断响应及处理
    - 5.6.3 驱动程序与设备管理
    - 5.6.7 文件系统和I/O协议栈
    - 5.6.5 系统调用
    - 5.6.4 线程调度与管理
    - 5.6.6 进程间通信
    - 5.6.8 人机交互界面

第1章

- 1.1 十余年的迷惑
- 1.2 从1+1=2说起
  - 1.2.1 用电路实现1+1=2
  - 1.2.2 或门/OR门
  - 1.2.3 与门/AND门
  - 1.2.4 非门/NOT门和与非门
  - 1.2.5 异或门/XOR门
  - 1.2.6 一位加法器
  - 1.2.7 全手动一位加法机
  - 1.2.8 实现多位加法器
  - 1.2.9 电路的时延
  - 1.2.10 新世界的新规律
  - 1.2.11 先行/并行进位
  - 1.2.12 电路化简和变换
- 1.3 我们需要真正可用的计算器
  - 1.3.1 产生记忆
  - 1.3.2 解决按键问题
  - 1.3.3 数学的懵懂
  - 1.3.4 第一次理解数学
  - 1.3.5 第一次理解语义
  - 1.3.6 七段显示数码管
  - 1.3.7 野路子乘法器
  - 1.3.8 科班乘法器
  - 1.3.9 数据交换器Crossbar
  - 1.3.10 多媒体声光按键转码器
  - 1.3.11 第一次驾驭时间
- 1.4 信息与信号
  - 1.4.1 录制和回放
  - 1.4.2 振动和信号
  - 1.4.3 低通滤波
  - 1.4.4 高通滤波
  - 1.4.5 带通滤波
  - 1.4.6 带阻滤波
  - 1.4.7 傅里叶变换
  - 1.4.8 波动与电磁波
  - 1.4.9 载波、调制与频分复用
- 1.5 完整的计算器
  - 1.5.1 用时序控制增强用户体验
  - 1.5.2 用MUX来实现Crossbar
  - 1.5.3 奇妙的FIFO队列
  - 1.5.4 同步/异步FIFO
  - 1.5.5 全局共享FIFO
  - 1.5.6 多路仲裁
  - 1.5.7 交换矩阵
  - 1.5.8 时序问题的产生与触发器
  - 1.5.9 擒纵机构与触发器
  - 1.5.10 擒纵机构与晶振
  - 1.5.11 Serdes与MUX/DEM...
  - 1.5.12 计算离不开数据传递
  - 1.5.13 几个专业概念的由来
- 1.6 多功能计算器
  - 1.6.1 算术逻辑单元/ALU

写作思路：先提出问题，计算机到底是怎么算1+1=2的。给出思考过程，给出答案，引出基本的组合逻辑电路，然后引出加法器。再提出问题，如何设计一个按键式计算机来算加法，给出思考过程，遇到了各种工程问题，一步步解决，最后引出译码器、触发器等逻辑电路。最后逐渐搭建出一个可以算多位数加减乘除的按键式计算器。最后引出ALU的概念。

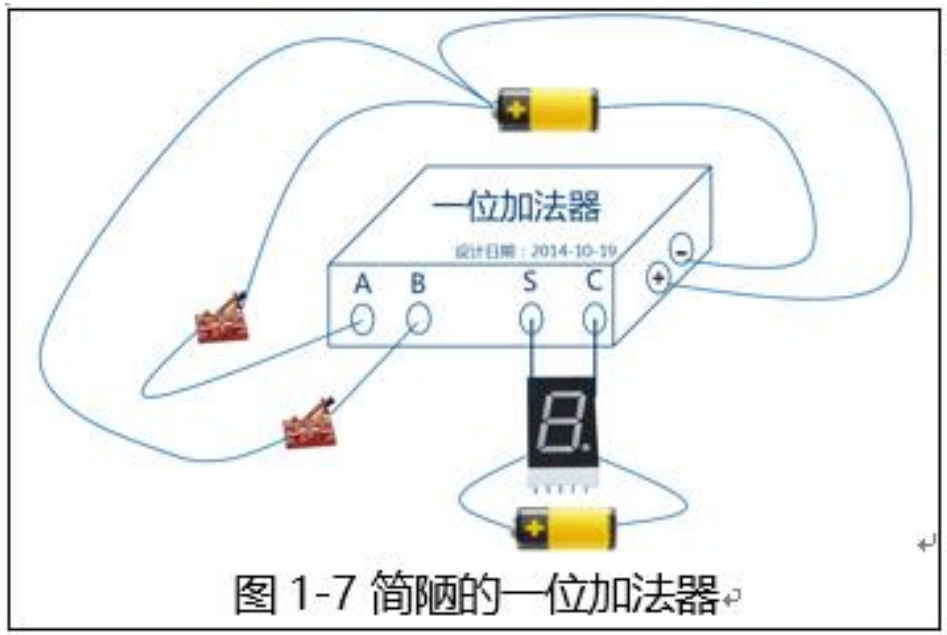


图 1-7 简陋的一位加法器

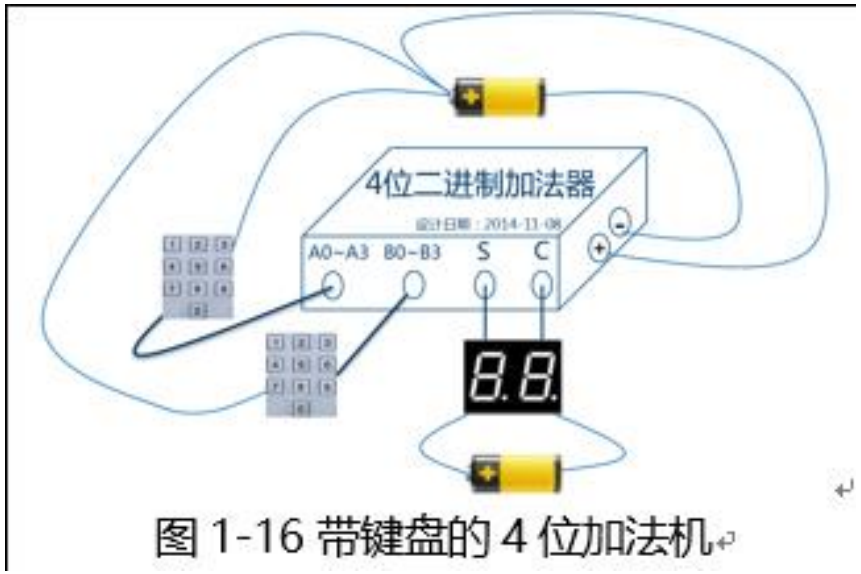
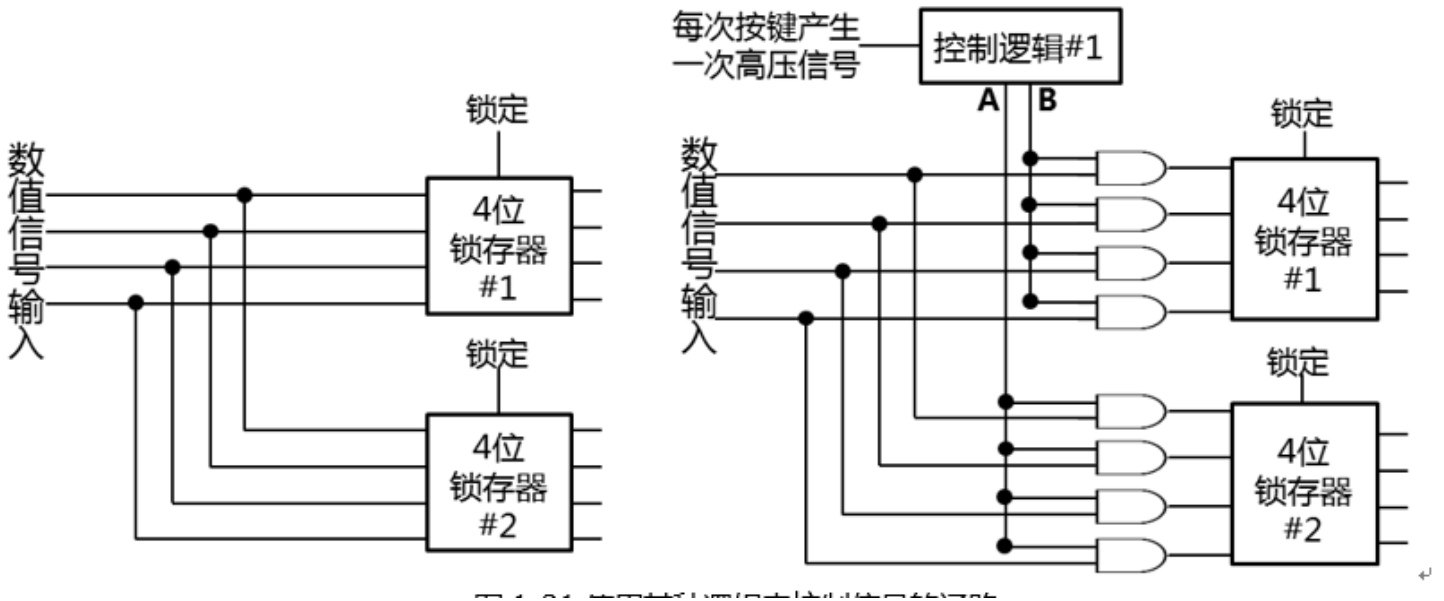


图 1-16 带键盘的 4 位加法机



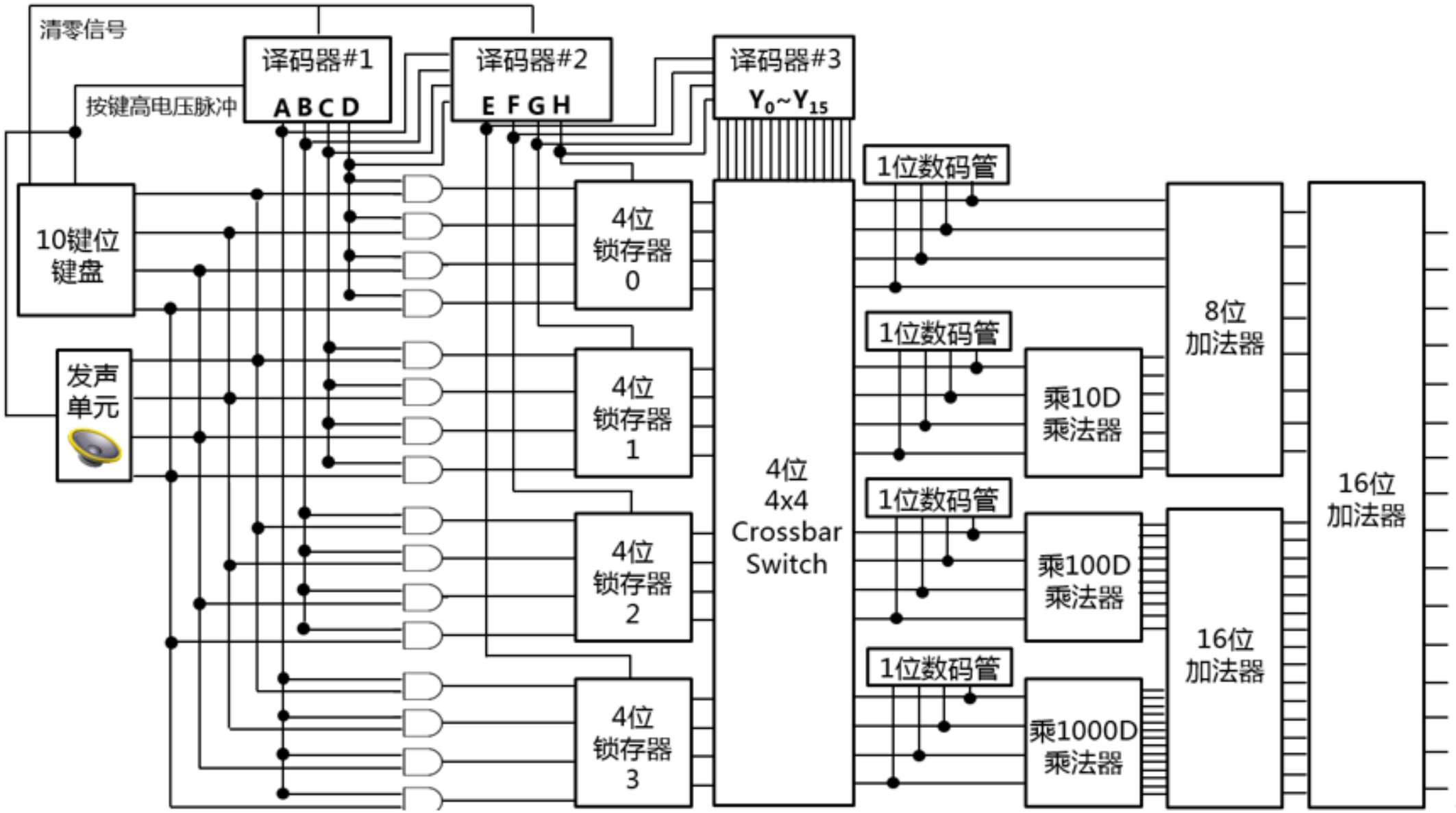
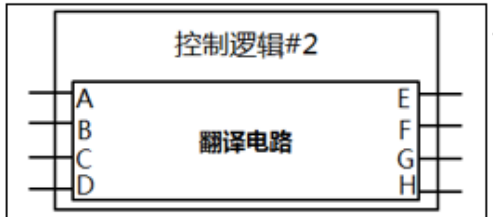


图 1-97 可发声按键转码器架构图



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

图 1-35 控制逻辑#2 电路真值表

同理，我们使用上文中给出的套路，写出 EFGH 这四个输出的表达式，结果遇到了问题。对于 E 这个输出，五行里有 4 行都是 1，如果按照第一行来给出表达式，则为 E= A'•B'•C'•D'，而将第二行的值代入这个表达式，会得出与现实不一样的结果，这就证明，我们之前使用的套路，要么出了问题，要么并不能完整的描述所有场景而只是特例。这可怎么办，空欢喜一场。但是至少 E= A'•B'•C'•D'可以描述第一行，那么第二行既然不能用这个式子描述，那肯定也有其对应的表达式，专门为第二行写出来，就是 E= A'•B'•C'•D，第三行则是 E= A'•B'•C•D'，第四行 E= A'•B•C'•D'，第五行由于 E=0，不用写表达式。这四行都对，又都不对，那到底是个什么状态？任何一种输入组合，要么匹配第一行，要么第二行，总之总得匹配某一行，这思路就来了，既然是“要么”，那么“或”这个逻辑，是否可以用来表达这个规律？我们把这五个式子或一下，**E= A'•B'•C'•D'+ A'•B'•C'•D+ A'•B'•C• D'+ A'•B•C'•D'**，这个式子表达了这样一个意思，任何一组输入组合，同时输入到这 4 个子等式里，一旦某个子等式不匹配，则这个子等式的输出一定是 0，但是这 4 个子等式中总有一个匹配，那么其输出为 1，而因为这 4 个子等式的输出是相或在一起的，所以整个等式的输出就为 1，正好匹配了 E=1 的现实。

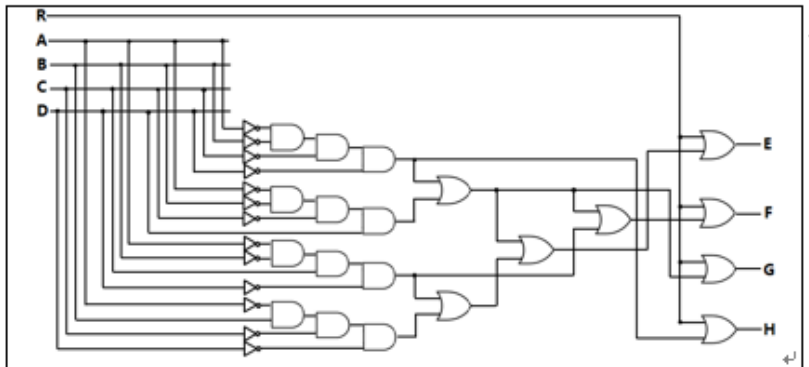


图 1-36 控制逻辑#2 电路原理图

代入验证发现，这个等式的确是正确的，其的确可以完整的描述一个真值表逻辑。同理，我们写出 EFGH 四个输出的各自的表达式：

E=A'•B'•C'•D'+A'•B'•C'•D+ A'•B'•C•D'+ A'•B•C'•D'，F= A'•B'•C'•D'+ A'•B'•C'•D+ A'•B'•C•D'，G=A'•B'•C'•D'+ A'•B'•C'•D，H=A'•B'•C'•D'。

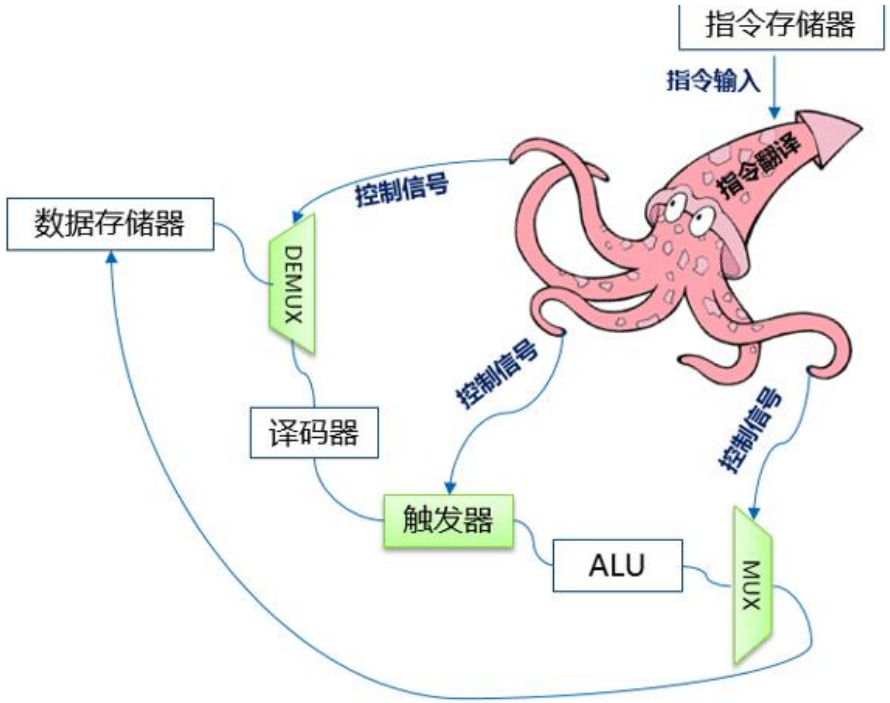
根据表达式画出对应的电路如图 1-36 所示。 这里清零逻辑使用了或门，因为对于控制逻辑#2，清零之后必须将 EFGH 全设置为 1，打开所有锁存器使其处于透传状态。

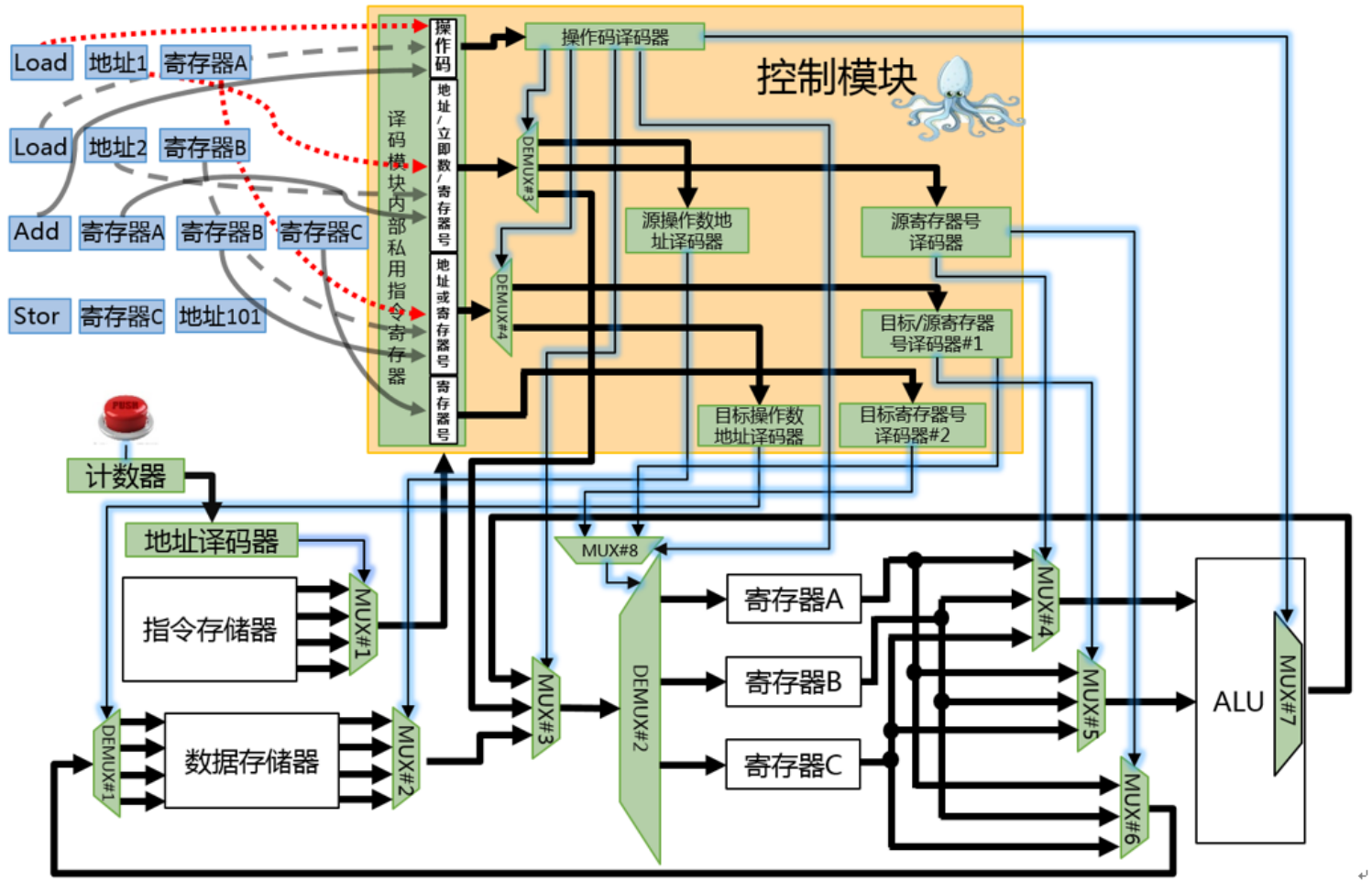
所以，根据真值表生成逻辑电路的基本规律是：忽略输出值为 0 的行，找出输出值为 1 的行，然后观察该行的所有输入信号的值，若为 0，则对信号取反，然后将该行所有输入信号相与，再将所有行相或，即可得出该行输出值。每一行输入信号的正值或者反值相与形成一个**乘积项（Product-Term）**，多个乘积项相或后形成一个输出值。

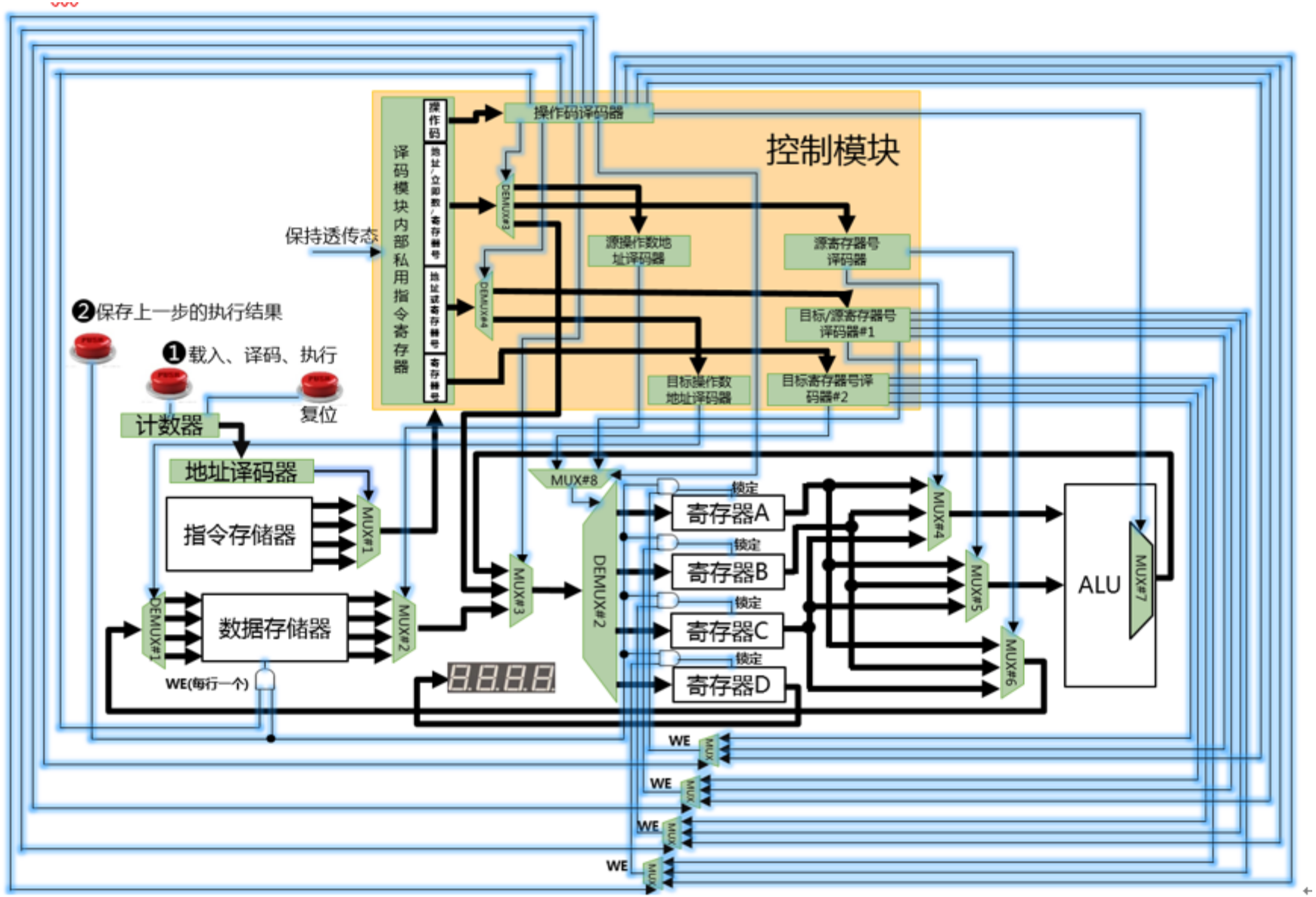
第2章

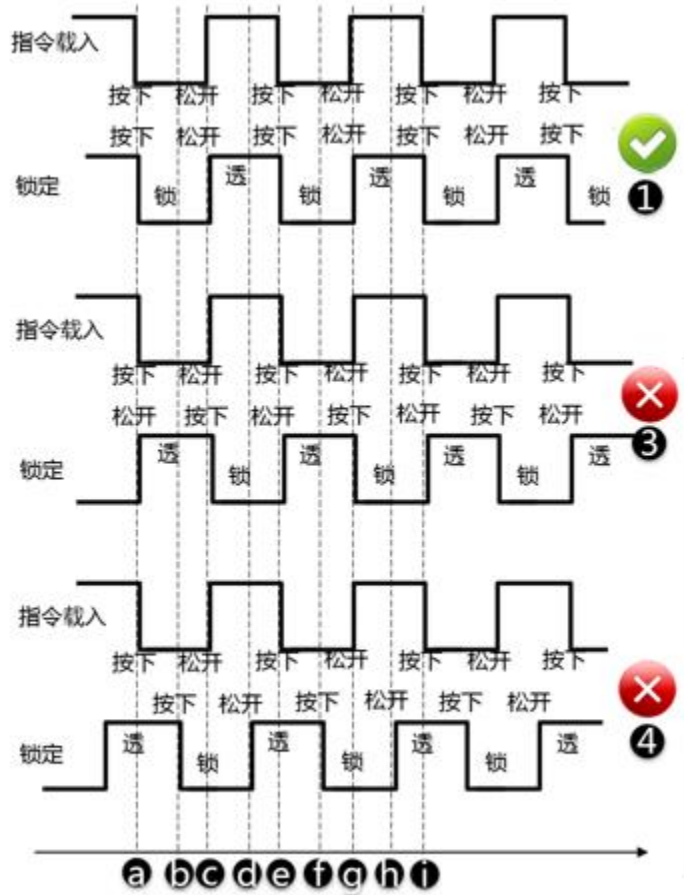
- 2.1 从累积计算说起
  - 2.1.1 实现累积计算
- 2.2 自动执行
  - 2.2.1 将操作方式的描述转化为指令
  - 2.2.2 实现那只章鱼——控制通路及部件
  - 2.2.3 动起来吧！——时序通路及部件
  - 2.2.4 半自动执行！——你得推着它跑
  - 2.2.5 全自动受控执行！——不用扬鞭自奋蹄！
  - 2.2.6 NOOP指令
  - 2.2.7 利用边沿型触发器搭建电路
  - 2.2.8 判断和跳转
  - 2.2.9 再见，Mr.章鱼！
- 2.3 更高效的执行程序
  - 2.3.1 利用循环缩减程序尺寸
  - 2.3.2 实现更多方便的指令
  - 2.3.3 多时钟周期指令
  - 2.3.4 微指令和微码
  - 2.3.5 全局地址空间
  - 2.3.6 多端口存储器
  - 2.3.7 多级缓存与CPU
  - 2.3.8 数据遍布各处
  - 2.3.9 降低数据操作粒度
  - 2.3.10 取指令/数据缓冲加速

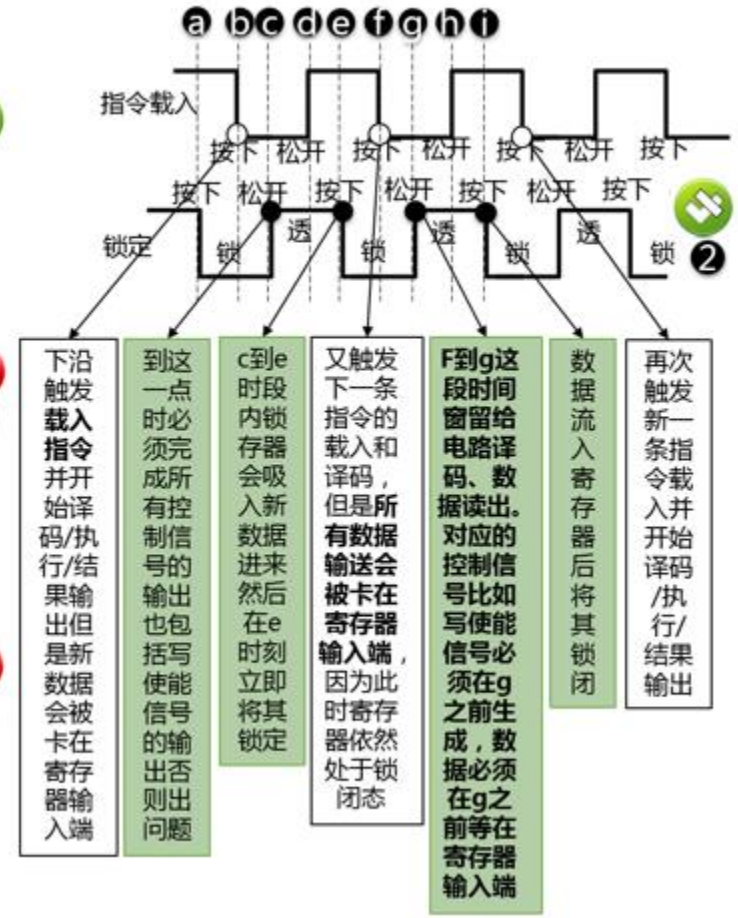
写作思路：在之前的简易计算器基础上，逐步实现自动根据指令来执行计算。比如算全班同学平均成绩，将800多次按键转化成指令，从而引出指令译码、控制逻辑、形成一个简易的CPU。

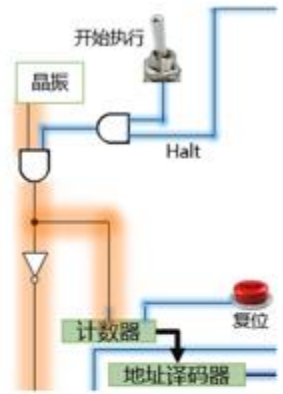

指令存储器
指令输入
指令翻译
控制信号
数据存储器
DEMUX
控制信号
控制信号
译码器
触发器
ALU
MUX

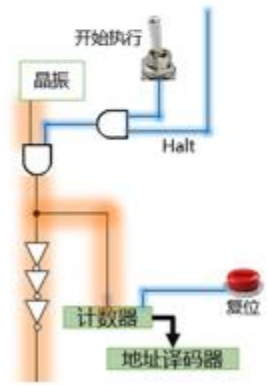

控制模块
Load
地址1
寄存器A
Load
地址2
寄存器B
Add
寄存器A
寄存器B
寄存器C
Stor
寄存器C
地址101
译码模块内部私用指令寄存器
操作码
地址/立即数/寄存器号
地址或寄存器号
寄存器号
操作码译码器
DEMUX#3
DEMUX#4
源操作数地址译码器
源寄存器号译码器
目标/源寄存器号译码器#1
目标操作数地址译码器
目标寄存器号译码器#2
计数器
地址译码器
指令存储器
MUX#1
DEMUX#1
数据存储器
MUX#2
MUX#3
MUX#8
DEMUX#2
寄存器A
寄存器B
寄存器C
MUX#4
MUX#5
MUX#6
ALU
MUX#7


控制模块
译码模块内部私用指令寄存器
操作码
地址/立即数/寄存器号
地址或寄存器号
寄存器号
操作码译码器
DEMUX#3
DEMUX#4
源操作数地址译码器
源寄存器号译码器
目标/源寄存器号译码器#1
目标操作数地址译码器
目标寄存器号译码器#2
保持透传态
❷保存上一步的执行结果
❶载入、译码、执行
复位
计数器
地址译码器
指令存储器
数据存储器
MUX#1
MUX#2
MUX#3
DEMUX#1
DEMUX#2
MUX#8
锁定
寄存器A
寄存器B
寄存器C
寄存器D
MUX#4
MUX#5
MUX#6
MUX#7
ALU
WE(每行一个)
WE


指令载入
按下 松开 按下 松开 按下 松开 按下
锁定
锁 透 锁 透 锁 透 锁
❶
指令载入
按下 松开 按下 松开 按下 松开 按下
松开 按下 松开 按下 松开 按下 松开
锁定
透 锁 透 锁 透 锁 透
❸
指令载入
按下 松开 按下 松开 按下 松开 按下
按下 松开 按下 松开 按下 松开
锁定
透 锁 透 锁 透 锁 透
❹
a b c d e f g h i


a b c d e f g h i
指令载入
按下 松开 按下 松开 按下 松开 按下
按下 松开 按下 松开 按下 松开 按下
锁定
锁 透 锁 透 锁 透 锁
❷
下沿触发**载入指令**并开始译码/执行/结果输出但是新数据会被卡在寄存器输入端
到这一点时必须完成所有控制信号的输出也包括写使能信号的输出否则出问题
c到e时段内锁存器会吸入新数据进来然后在e时刻立即将其锁定
又触发下一条指令的载入和译码，但是**所有数据输送会被卡在寄存器输入端**，因为此时寄存器依然处于锁闭态
F到g这段时间窗留给电路译码、数据读出。对应的控制信号比如写使能信号必须在g之前生成，数据必须在g之前等在寄存器输入端
数据流入寄存器后将其锁闭
再次触发新一条指令载入并开始译码/执行/结果输出


开始执行
晶振
Halt
计数器
复位
地址译码器


开始执行
晶振
Halt
计数器
复位
地址译码器

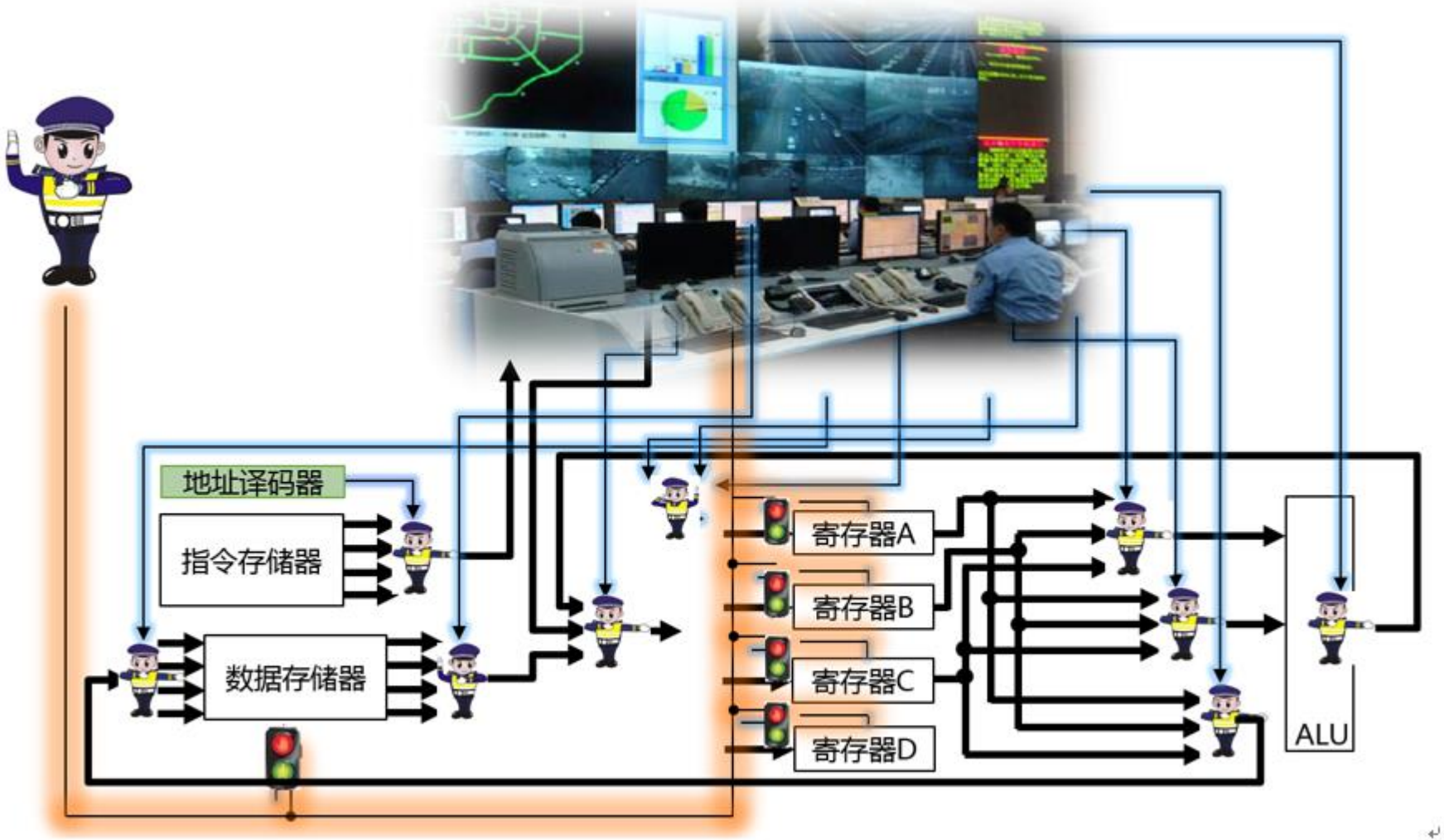


图 2-16 各控制部件只是在根据指令来指挥交通

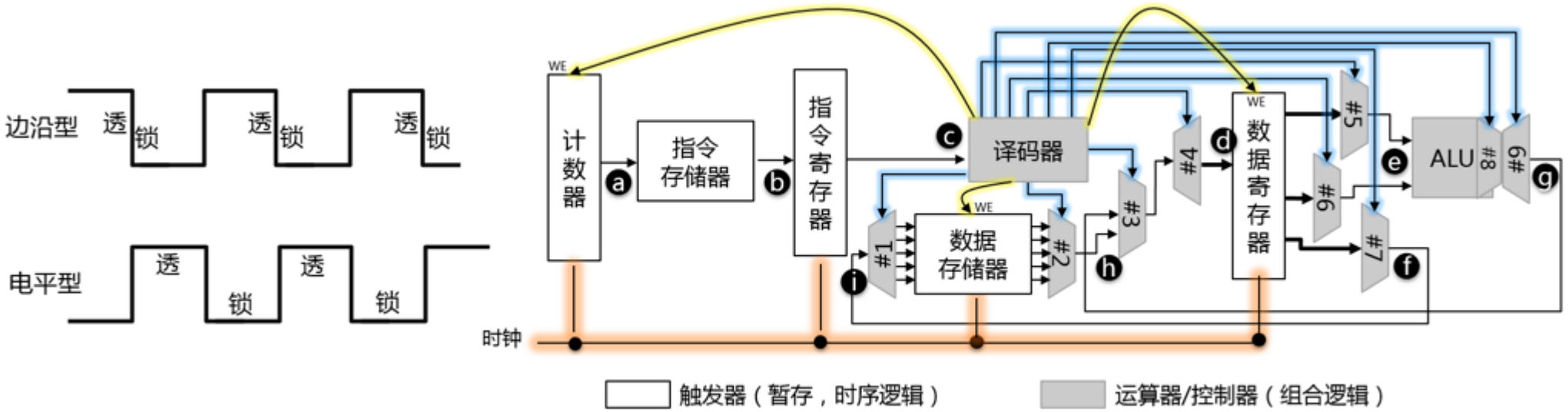


图 2-17 采用边沿型触发器时示意图

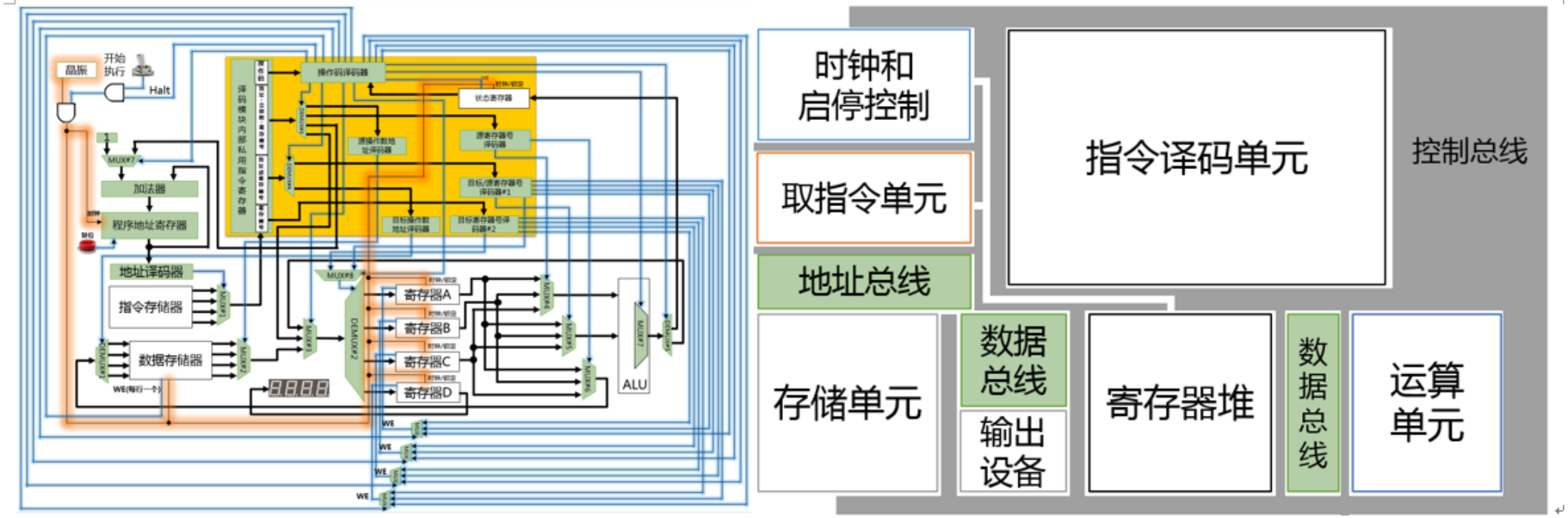


图 2-22 为主要模块起个名字

第3章

- 3.1 从薄铁片到机械计算机
  - 3.1.1 算盘和计算尺
  - 3.1.2 不可编程手动机械十进制计算机
  - 3.1.3 可编程自动机械十进制计算机
  - 3.1.4 可编程自动电动机械二进制计算机
  - 3.1.5 可编程自动全电动二进制计算机
- 3.2 电子管时代
  - 3.2.1 二极电子管
  - 3.2.2 三极电子管
  - 3.2.3 AM广播革命
  - 3.2.4 电子管计算机
  - 3.2.5 石头会唱歌
- 3.3 固态革命——晶体管
  - 3.3.1 P/N结与晶体管
  - 3.3.2 场效应管（FET）
  - 3.3.3 MOSFET
  - 3.3.4 cMOS
  - 3.3.4 晶体管计算机
- 3.3 制造工艺革命——集成电路
  - 3.3.1 从泥活字看量产晶体管
  - 3.3.2 跟冬瓜哥学做P/N结蛋糕
  - 3.3.3 提升集成度
  - 3.3.4 芯片内的深邃世界
  - 3.3.5 cMOS集成电路工艺概述
  - 3.3.6 cMOS工艺步骤概述
  - 3.3.7 cMOS工艺详细步骤
    - 3.3.7.1 热氧化
    - 3.3.7.2 氮化硅积淀
    - 3.3.7.3 浅槽隔离蚀刻
    - 3.3.7.4 pMOS和nMOS生成
    - 3.3.7.5 触点电极的生成
    - 3.3.7.6 通孔和支撑柱（via）的生成
    - 3.3.7.7 第一层导线连接
    - 3.3.7.8 第二层导线连接
    - 3.3.7.9 表面钝化
  - 3.3.8 半导体工艺的瓶颈
    - 3.3.8.1 寄生电容
    - 3.3.8.2 静态/动态功耗
    - 3.3.8.3 栅氧厚度和High-K材料
    - 3.3.8.4 导线连接和Low-K材料
    - 3.3.8.5 驱动能力及时延
    - 3.3.8.6 时钟树
  - 3.3.9 集成电路计算机
  - 3.3.10 微处理器计算机
  - 3.3.11 暴力拆解奔三CPU
- 3.4 存储器-不得不说的故事
  - 3.4.1 机械存储器
    - 3.4.1.1 声波/扭力波延迟线（Delay Line
    - 3.4.1.2 磁鼓存储器（Drum）
    - 3.4.1.3 磁芯存储器（Core）
  - 3.4.2 电子存储器
    - 3.4.2.1 静态随机存储器SRAM
    - 3.4.2.2 动态随机存储器DRAM
    - 3.4.2.3 Flash闪存
    - 3.4.2.4 只读存储器ROM
  - 3.4.3 光存储器
    - 3.4.3.1 光盘是如何存储数据的
    - 3.4.3.2 压盘与刻盘的区别
    - 3.4.3.3 光盘表面微观结构
    - 3.4.3.4 多层记录
    - 3.4.3.5 激光头的秘密
    - 3.4.3.6 蓝光光盘简介
  - 3.4.4 不同器件担任不同角色
    - 3.4.4.1 寄存器和缓存
    - 3.4.4.2 主运行内存/主存
    - 3.4.4.3 Scratchpad RAM
    - 3.4.2.4 内容寻址内存CAM/TCAM
    - 3.4.2.5 外存

写作思路：前两章读者已经充分了解了数字电路是如何进行计算的。本章介绍人们是如何用电子开关搭建出电路的，电子开关的发展，一直到最后的芯片是怎么做出来的。

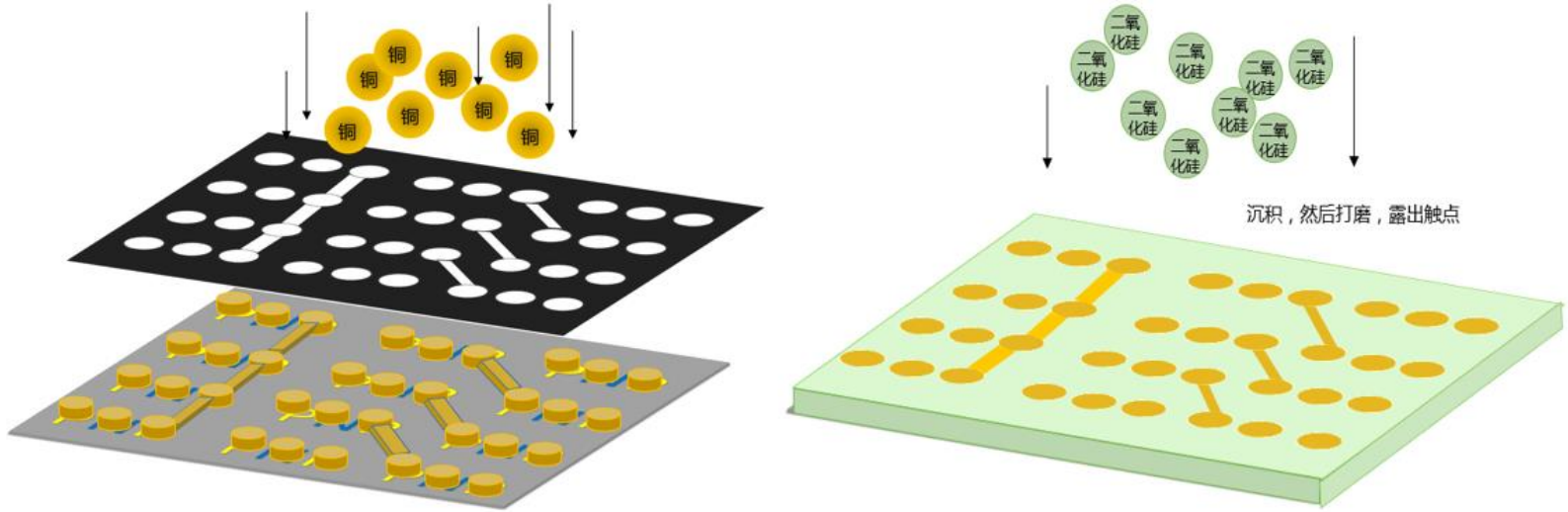


图 3-101 沉积导线以及覆盖一层二氧化硅绝缘层

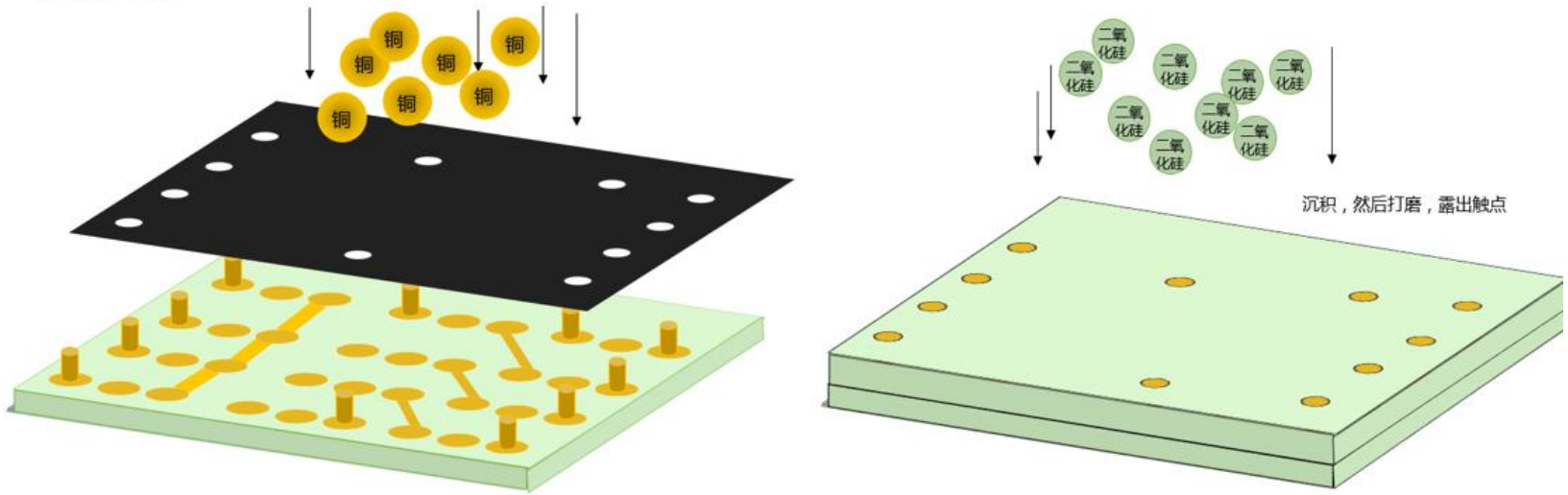
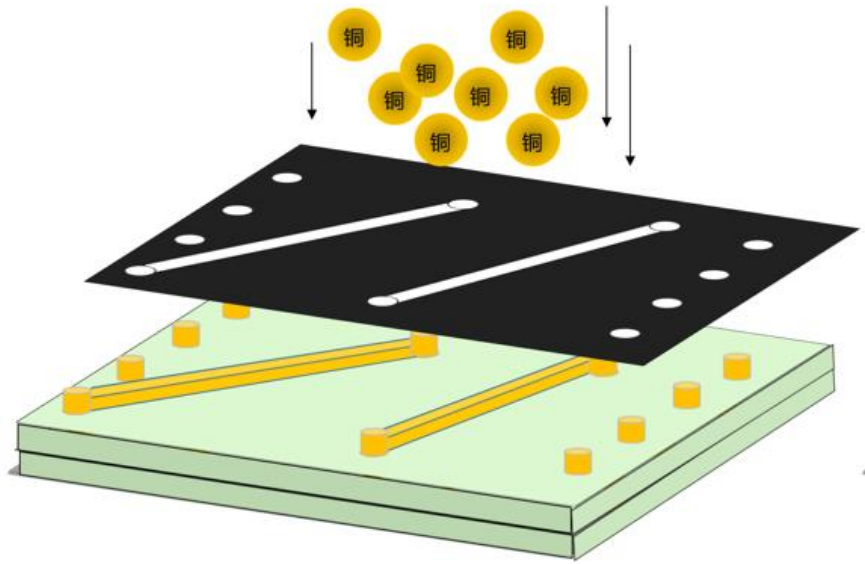
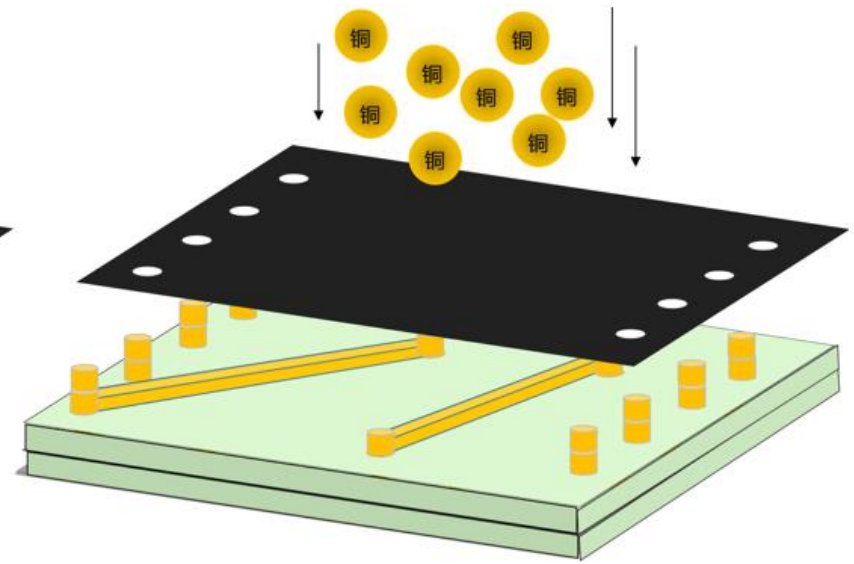
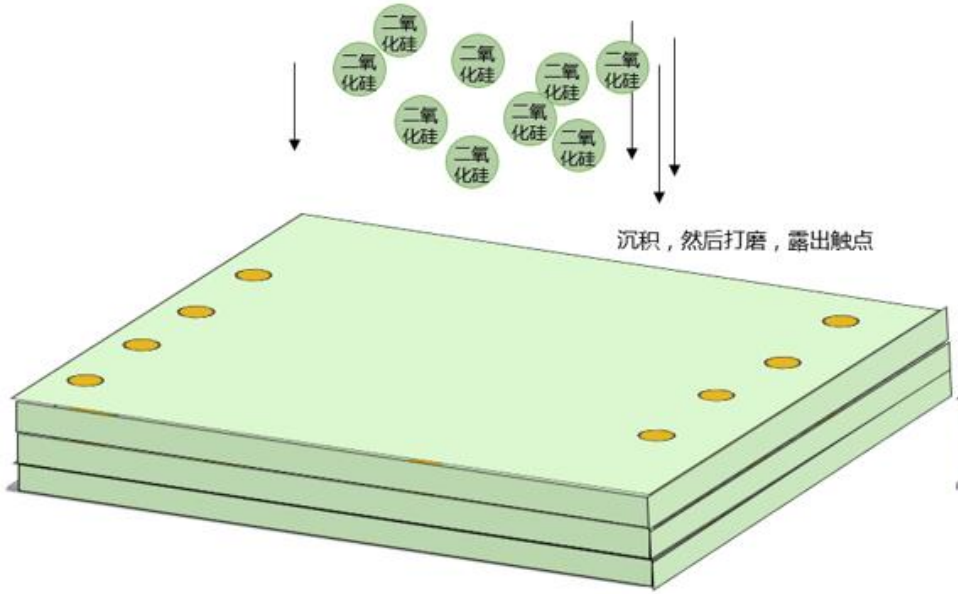


图 3-102 将需要高架的导线两端升起来并再次覆盖绝缘层


铜
铜
铜
铜
铜
铜
铜


铜
铜
铜
铜
铜
铜
铜


二氧化硅
二氧化硅
二氧化硅
二氧化硅
二氧化硅
二氧化硅
二氧化硅
二氧化硅
二氧化硅
沉积，然后打磨，露出触点

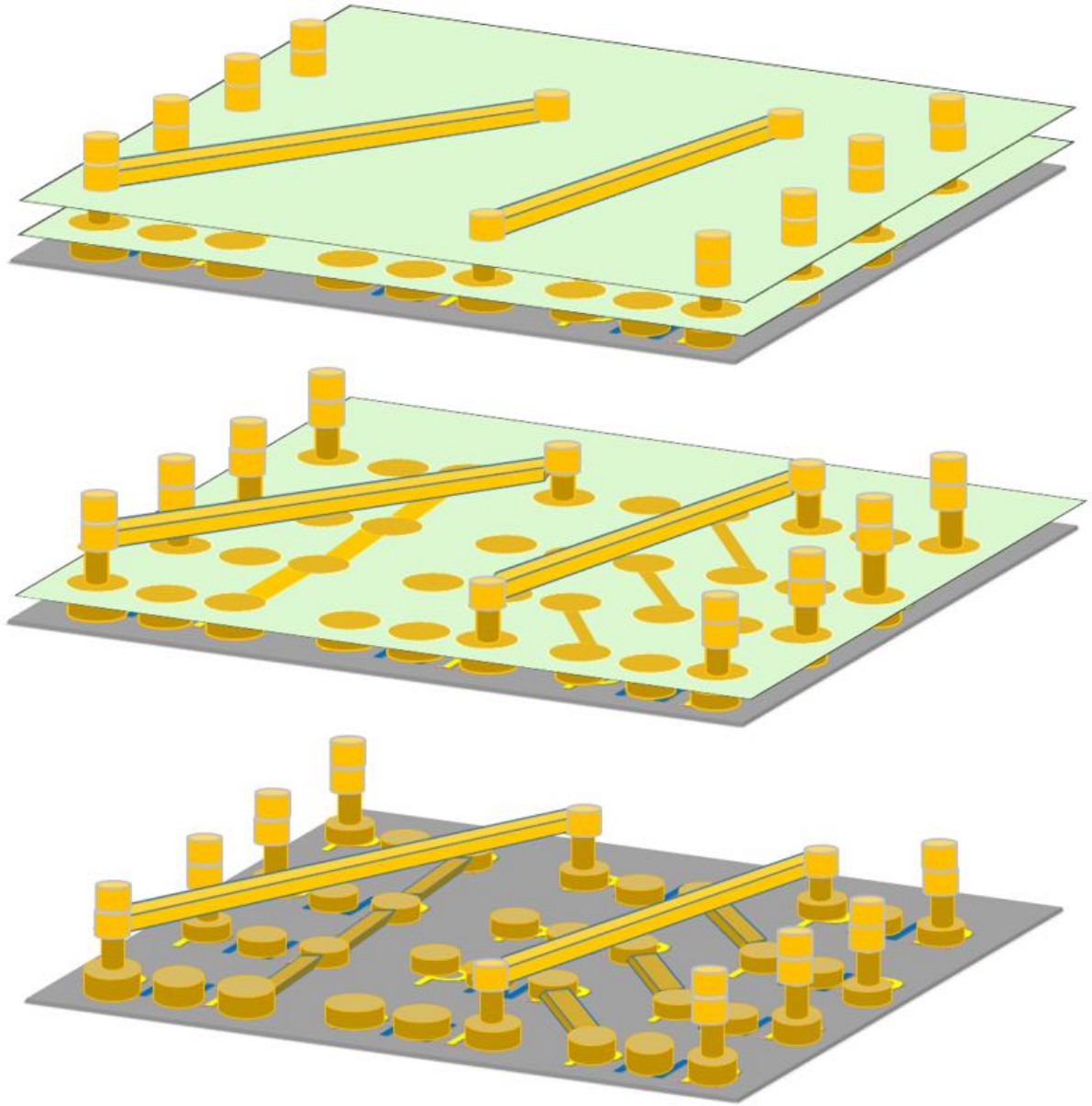
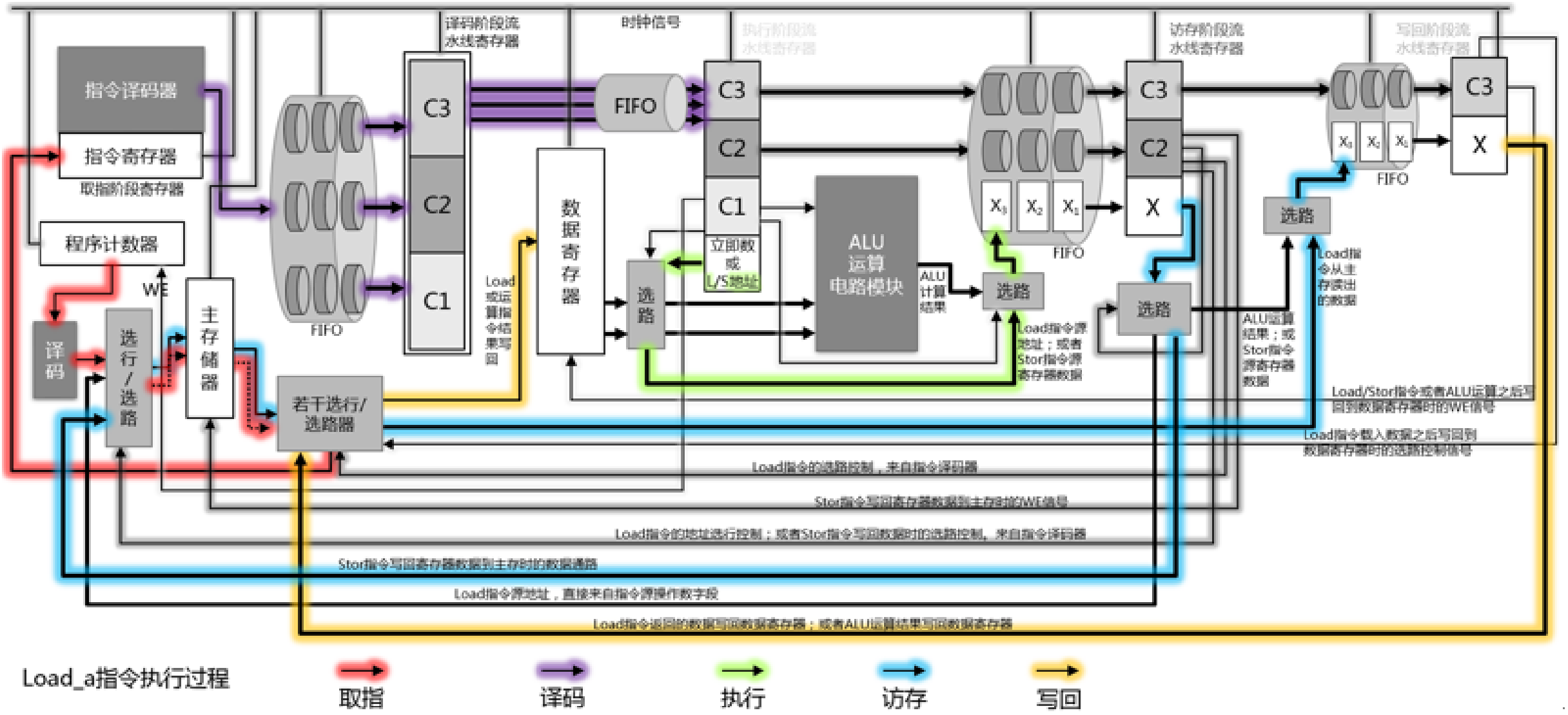


图 3-112 60 年代的人们正在红色胶片上手工雕刻遮罩

- 4.1 大话流水线
  - 4.1.1 不高兴的译码器
  - 4.1.2 思索流水线
    - 4.1.2.1 流水线的本质是并发
    - 4.1.2.2 不同时延的步骤混杂
    - 4.1.2.3 大话队列
    - 4.1.2.4 流水线的应用及优化
- 4.2 优化流水线
  - 4.2.1 拆分慢速步骤
  - 4.2.2 放置多份慢速部分
  - 4.2.3 加入缓冲队列
  - 4.2.4 图解五级流水线指令执行过程
- 4.3 流水线冒险
  - 4.3.1 访问冲突与流水线阻塞
  - 4.3.2 数据依赖与数据前递
  - 4.3.3 跳转冒险与分支预测
- 4.4 指令的动态调度
  - 4.4.1 结构相关与寄存器重命名
  - 4.4.2 保留站与乱序执行
  - 4.4.3 分步图解乱序执行
  - 4.4.4 重排序缓冲与指令顺序提交
- 4.5 物理并行执行
  - 4.5.1 超标量和多发射
  - 4.5.2 VLIW超长指令字
  - 4.5.3 SIMD单指令多数据

写作思路：前三章读者已经充分了解CPU基本原理，以及人们如何制造CPU，通过芯片技术可以集成入大量复杂的电路逻辑。有了这个认知基础，开始介绍CPU内部更高级的控制方法，流水线、多发射、乱序执行、分支预测等等体系结构方面的概念。以“不高兴的译码器”为标题党，强烈将读者思维代入作者设下的套中无法自拔，并最终跟随作者的思考思路完成本章的学习。


指令译码器
指令寄存器
取指阶段寄存器
程序计数器
WE
译码
选行/选路
主存储器
若干选行/选路器
译码阶段流水线寄存器
FIFO
C3
C2
C1
Load或运算指令结果写回
数据寄存器
选路
时钟信号
执行阶段流水线寄存器
FIFO
C3
C2
C1
立即数或L/S地址
ALU运算电路模块
ALU计算结果
选路
Load指令源地址；或者Stor指令源寄存器数据
访存阶段流水线寄存器
FIFO
C3
C2
X
选路
ALU运算结果；或Stor指令源寄存器数据
写回阶段流水线寄存器
FIFO
C3
X
选路
Load指令从主存读出的数据
Load/Stor指令或者ALU运算之后写回到数据寄存器时的WE信号
Load指令载入数据之后写回到数据寄存器时的选路控制信号
Load指令的选路控制，来自指令译码器
Stor指令写回寄存器数据到主存时的WE信号
Load指令的地址选行控制；或者Stor指令写回数据时的选路控制，来自指令译码器
Stor指令写回寄存器数据到主存时的数据通路
Load指令源地址，直接来自指令源操作数字段
Load指令返回的数据写回数据寄存器；或者ALU运算结果写回数据寄存器
Load_a指令执行过程
取指
译码
执行
访存
写回

从这 8bit 记录中，可以观察到该指令以往 8 次跳转的**关联模式/范式**，比如范式 01010101 表示隔一次跳一次；而范式 00110011 表示隔两次跳两次，同理，01100111 也是一种范式，但是看上去没什么明显规律，也没法用简洁的语言表达，但是如果 01100111 这种范式在程序执行期间重复出现的话，也就能从更久远的历史中看出之前根本无法分辨的规律。至于为什么体现出这种规律，那是更底层的计算机程序科学家所研究的问题了。对于分支预测，很显然，如果上一次该代码处于某个范式时，它跳了，那么当它执行了若干次后，如果恰好这 8bit 历史记录又轮回到了同一个范式，那么你说，该跳还是不跳？当然是跳啊！为什么？因为上一次这样的时候它跳了啊！就这么简单。用这种方式可以深入到事物的更深层次的表象中挖掘出更深层的规律。相比之下，如果只是简单的比较 1 和 0 谁多就听谁的，就显得简陋和粗暴了。如图 4-44 中所示的 BHT 表中，其中红、黄、蓝、绿四条指令的跳转范式就出现了轮回，注意同一行中的相同颜色字体表示范式轮回。

好了，看来这个思路已经非常明朗了，现在需要解决如何记录“当某个范式到来时该指令跳了没有”这件事了。请注意，之前的思路是为每一条跳转指令记录一个 2bit 的跳转记录，而现在需要为每一条指令的多个跳转范式分别各自记录一个 2bit 的跳转记录，有多少个可能的范式，就要记录多少个 2bit 的跳转记录，8bit 可以组成 256 种范式，所以需要记录 256 个 2bit。怎么记录？能否直接在 BHT 表的右侧加一列跳转记录列呢？不行，因为我们需要对单个指令的每一可能的范式都记录对应的跳转记录，8bit 可以组成最大 256 个范式，那就意味着，对于单条指令，我们需要建立一张 256 行 2 列的表，第一列 8bit 记录该指令所有可能的 256 种范式，第二列 2bit（每次遇到该范式时跳了就置 1，不跳就置 0，也用 2bit 来记录，增加防抖动缓冲）记录该范式对应的跳转记录，表格共计 256x10bit=320Byte。这还是单条指令，如果记录 256 条指令的话，那么就需要 256x320Byte=80KB 的 CAM 存储器，会占用较大电路面积。

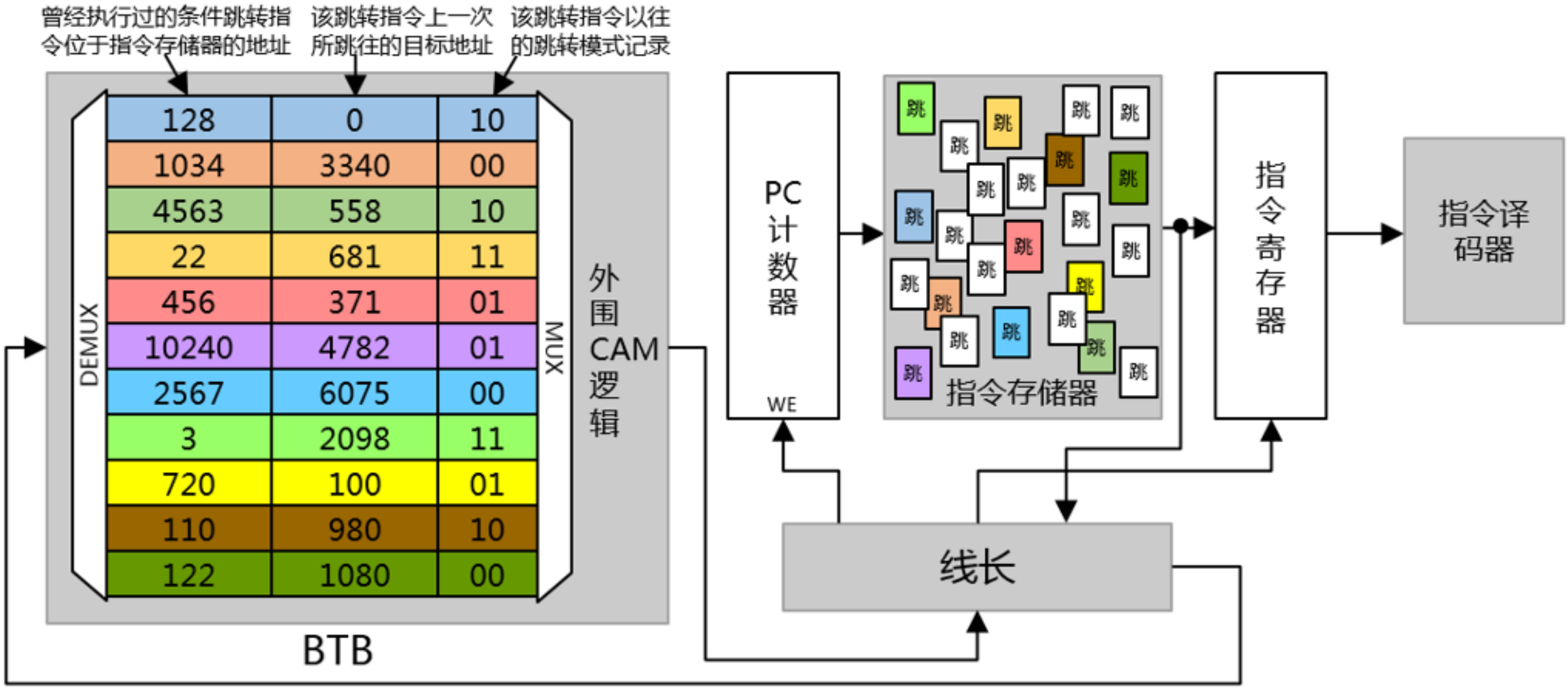


图 4-43 加入跳转目标地址列的 PHT 被称为 BTB

1 Load_a 100 A；//从100号存储器地址载入数据到寄存器A中

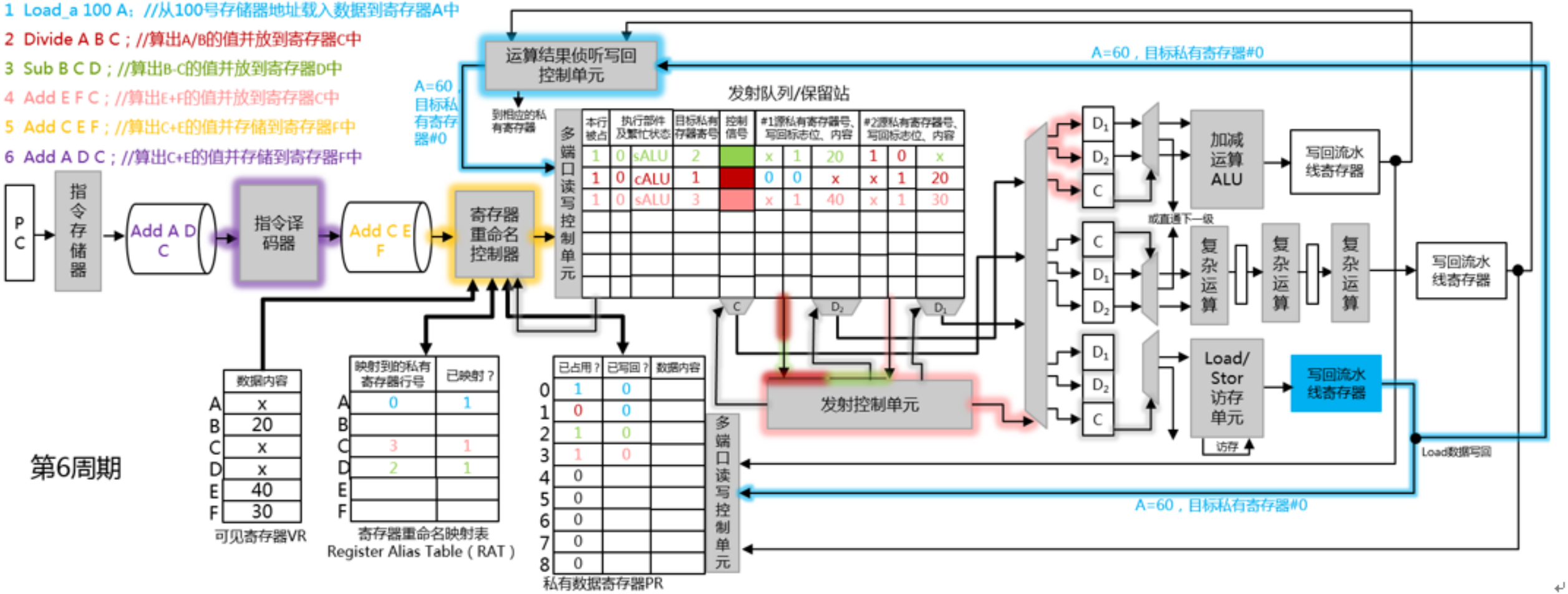
2 Divide A B C；//算出A/B的值并放到寄存器C中

3 Sub B C D；//算出B-C的值并放到寄存器D中

4 Add E F C；//算出E+F的值并放到寄存器C中

5 Add C E F；//算出C+E的值并存储到寄存器F中

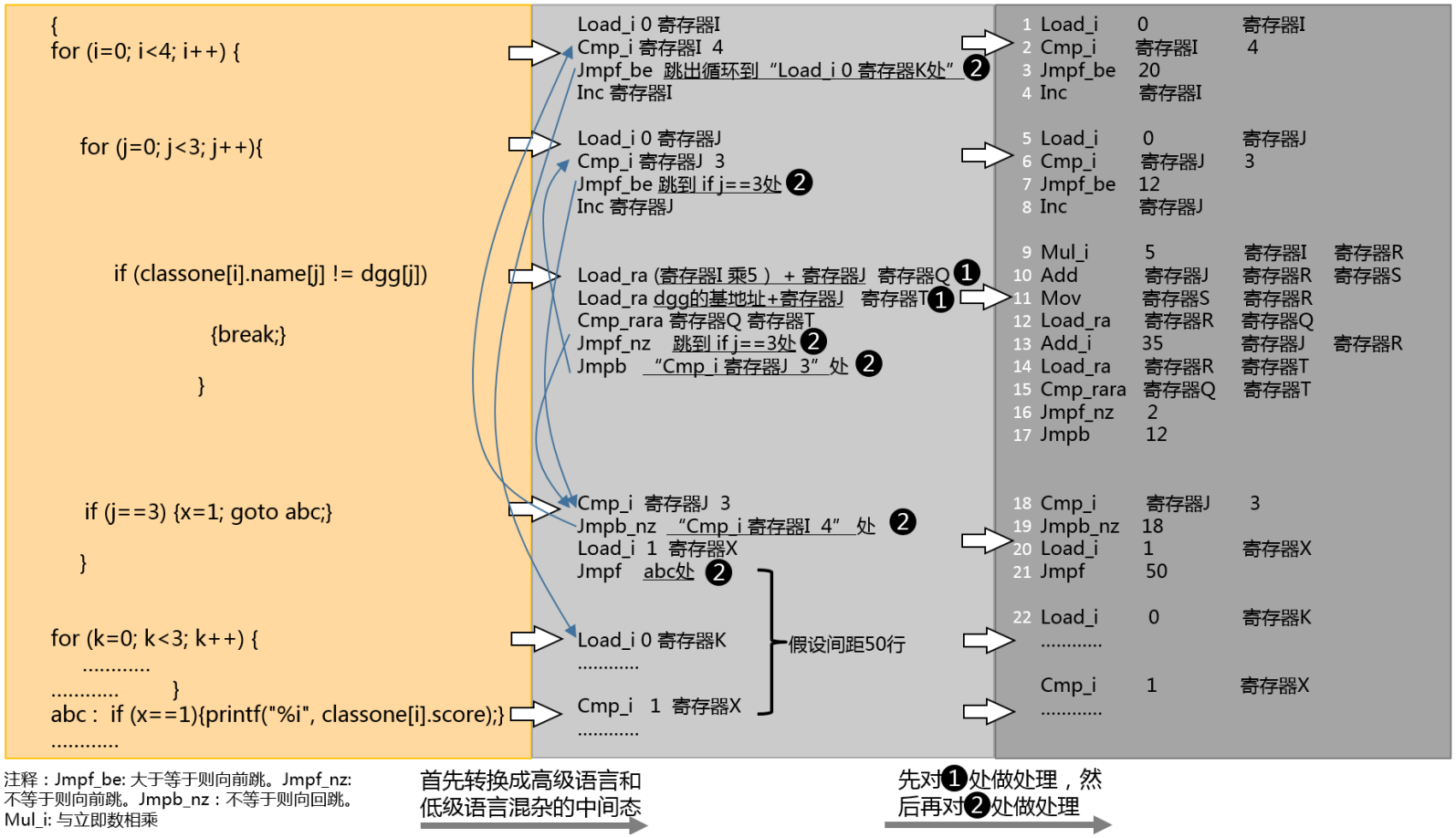
6 Add A D C；//算出C+E的值并存储到寄存器F中



第6周期

- 第5章
  - 5.1 基本的数据结构
    - 5.1.1 数组
    - 5.1.2 数据类型与ASCII码
    - 5.1.3 结构体
    - 5.1.4 数据怎么摆很重要
  - 5.2 高级语言
    - 5.2.1 简单的声明和赋值
    - 5.2.2 编译和编译器
    - 5.2.3 向编译器描述数据的编排方式
    - 5.2.4 高级语言编程小试牛刀
    - 5.2.5 人脑编译忆苦思甜
  - 5.3 浮点数及浮点运算
    - 5.3.1 数值范围和精度
    - 5.3.2 浮点数的用处和表示方法
    - 5.3.3 浮点数的二进制表示
      - 5.3.3.1 二进制浮点数转十进制小数
      - 5.3.3.2 十进制小数转二进制浮点数
      - 5.3.3.3 负指数和0的表示
      - 5.3.3.4 无穷与非规格化数的表示
    - 5.3.4 浮点数运算挺费劲
    - 5.3.5 浮点数的C语言声明
    - 5.3.6 十六进制表示法
  - 5.4 程控多媒体计算机
    - 5.4.1 键盘是前提
    - 5.4.2 搜索并显示
    - 5.4.3 实现简易计算器
    - 5.4.4 录入和保存
    - 5.4.5 简易文件系统
    - 5.4.6 计时/定时
    - 5.4.7 发声控制
    - 5.4.8 图像显示
    - 5.4.9 网络聊天
  - 5.5 程序社会
    - 5.5.1 函数和调用
    - 5.5.2 设备驱动程序
    - 5.5.3 函数之间的联络站
    - 5.5.4 库和链接
      - 5.5.4.1 静态库和静态链接
      - 5.5.4.2 头文件
      - 5.5.4.3 API和SDK
      - 5.5.4.4 动态库和动态链接
      - 5.5.4.5 库文件/可执行文件的格式
    - 5.5.5 程序的执行和退出
      - 5.5.5.1 初步解决地址问题
      - 5.5.5.3 更好的人机交互方式
      - 5.5.5.4 程序的退出
      - 5.5.5.5 使用外部设备和内存
    - 5.5.6 多程序并发执行
      - 5.5.6.1 利用时钟中断来切换线程
      - 5.5.6.2 更广泛的使用中断
      - 5.5.6.3 虚拟地址空间与分页
      - 5.5.6.4 虚拟与现实的边界——系统调用
    - 5.5.7 呼唤操作系统
  - 5.6 计算机操作系统
    - 5.6.1 内存管理
      - 5.6.1.1 分区式内存管理
      - 5.6.1.2 分段+分区式内存管理
      - 5.6.1.3 Intel CPU的分段支持机制
      - 5.6.1.4 分页+分段式内存管理
      - 5.6.1.5 DOS和Linux操作系统的内存管理
    - 5.6.2 中断响应及处理
    - 5.6.3 驱动程序与设备管理
    - 5.6.7 文件系统和I/O协议栈
    - 5.6.5 系统调用
    - 5.6.4 线程调度与管理
    - 5.6.6 进程间通信
    - 5.6.8 人机交互界面

写作思路：是时候引入程序是如何执行的了。从基本数据结构，介绍到高级语言，然后是编译原理，并手动编译一个简易程序示例。都是带着思考，先给出命题，为什么会这样，解决思路，具体设计，实际设计。最终一步步引出操作系统的概念。



注释：Jmpf_be: 大于等于则向前跳。Jmpf_nz: 不等于则向前跳。Jmpb_nz：不等于则向回跳。
Mul_i: 与立即数相乘

首先转换成高级语言和低级语言混杂的中间态

先对❶处做处理，然后再对❷处做处理

5-17 所示为 32bit 浮点数的最终表示法和分布图示。

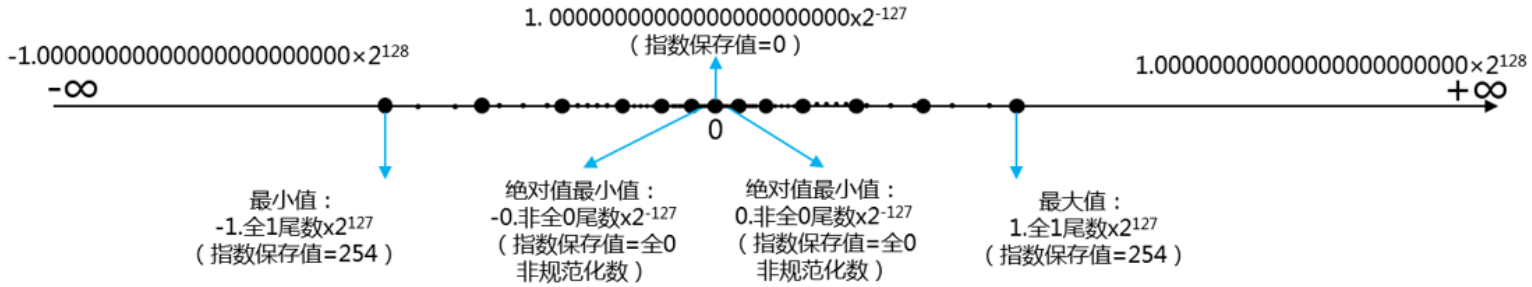


图 5-17 32bit 浮点数的最终表示法和分布图示

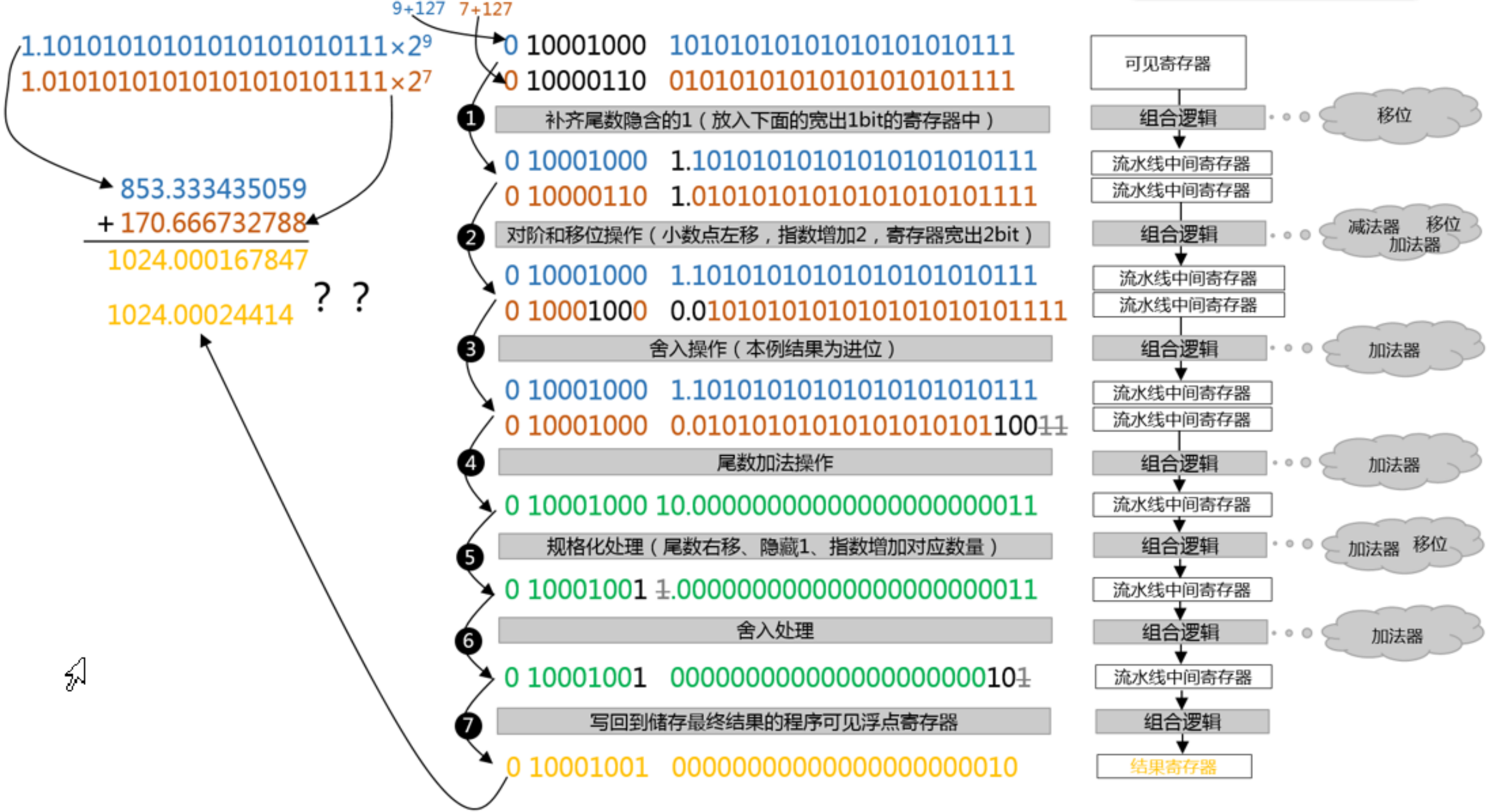
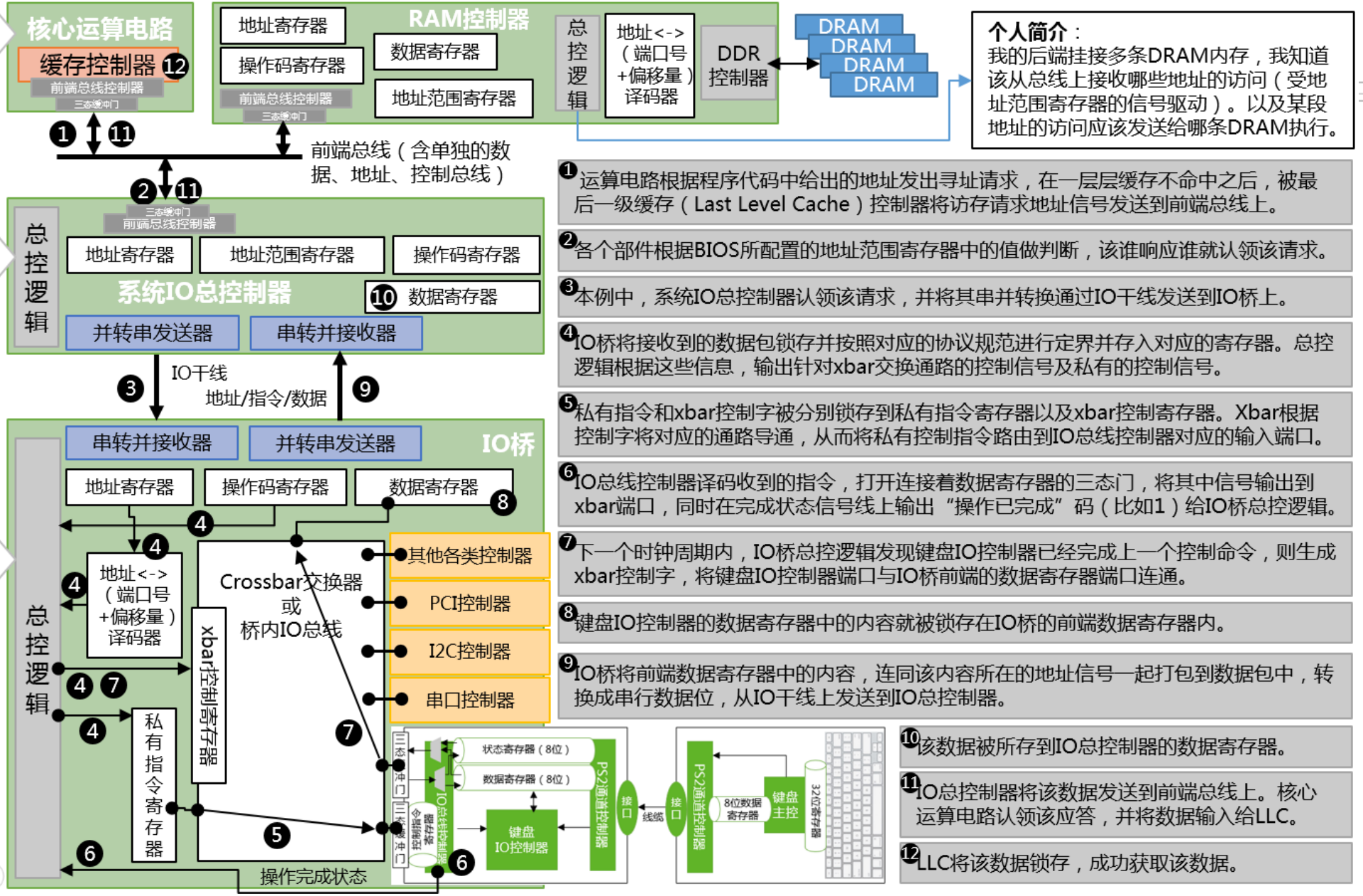


图 5-18 两个相同符号的浮点数相加的执行流程

个人简介：
程序运行时发出地址访问信号，但是我并不知道也不关心对应这个地址的数据到底在哪里存放，自有人会处理。

个人简介：
我根据地址范围寄存器中的值与总线上采样到的地址值做比较，如果匹配，则认领该访问请求，并将其地址、操作码和数据信号暂存，然后通过IO干线发送到IO桥上处理。

个人简介：
我后面挂接了多个不同类型的控制器，每个控制器之后又可能再次挂接了一个或多个设备。主板制造商预先将所有这些控制器的寄存器数量和容量编排到我内部存储的一张表中。我只需要根据前端请求中的地址信号查表/译码以决定向哪个控制器发起对应的地址访问信号即可。



个人简介：
我的后端挂接多条DRAM内存，我知道该从总线上接收哪些地址的访问（受地址范围寄存器的信号驱动）。以及某段地址的访问应该发送给哪条DRAM执行。

1. 运算电路根据程序代码中给出的地址发出寻址请求，在一层层缓存不命中之后，被最后一级缓存（Last Level Cache）控制器将访存请求地址信号发送到前端总线上。
2. 各个部件根据BIOS所配置的地址范围寄存器中的值做判断，该谁响应谁就认领该请求。
3. 本例中，系统IO总控制器认领该请求，并将其串并转换通过IO干线发送到IO桥上。
4. IO桥将接收到的数据包锁存并按照对应的协议规范进行定界并存入对应的寄存器。总控逻辑根据这些信息，输出针对xbar交换通路的控制信号及私有的控制信号。
5. 私有指令和xbar控制字被分别锁存到私有指令寄存器以及xbar控制寄存器。Xbar根据控制字将对应的通路导通，从而将私有控制指令路由到IO总线控制器对应的输入端口。
6. IO总线控制器译码收到的指令，打开连接着数据寄存器的三态门，将其中信号输出到xbar端口，同时在完成状态信号线上输出“操作已完成”码（比如1）给IO桥总控逻辑。
7. 下一个时钟周期内，IO桥总控逻辑发现键盘IO控制器已经完成上一个控制命令，则生成xbar控制字，将键盘IO控制器端口与IO桥前端的数据寄存器端口连通。
8. 键盘IO控制器的数据寄存器中的内容就被锁存在IO桥的前端数据寄存器内。
9. IO桥将前端数据寄存器中的内容，连同该内容所在的地址信号一起打包到数据包中，转换成串行数据位，从IO干线上发送到IO总控制器。
10. 该数据被所存到IO总控制器的数据寄存器。
11. IO总控制器将该数据发送到前端总线上。核心运算电路认领该应答，并将数据输入给LLC。
12. LLC将该数据锁存，成功获取该数据。

第一步： 核心
（Last Level Cach

第二步：各个部
号，谁认领该请求
释放总线上的信号

第三步：本例中
范围。其将总线上
包然后通过 IO 干线

第四步：IO 桥
码，从哪到哪是数
信号则输出给地址
询一张地址表来判
根据该 IO 桥连接的
SRAM 中的。其译
号，同时生成私有
号，同时生成私有

第五步：在下一
根据控制字将对应
时钟周期下沿锁存

第六步：下一个
表明之前有按键码
三态门，将其中的
总控逻辑。

第七步： IO 桥
与 IO 桥前端的数据

第八步：在下一

第九步：IO 桥

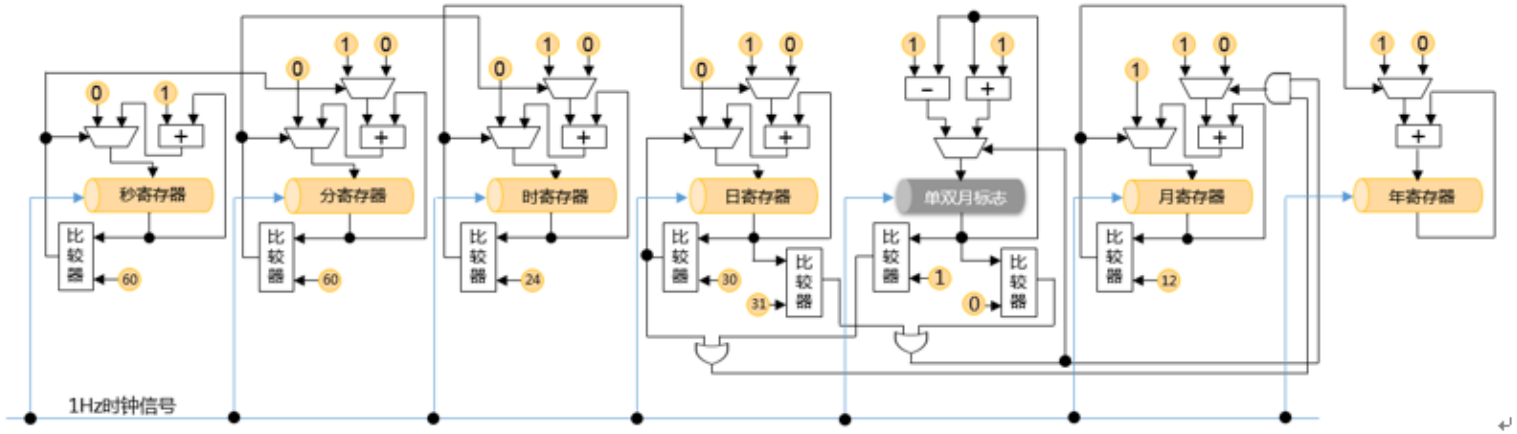


图 5-36 纯逻辑电路搭建的计时器

可以把这个纯数字逻辑计时器，辅之以一个 IO 控制器，然后与主 CPU 一侧的 IO 桥对接起来，也一样可以完成计时和时间读取的功能。

> **软和硬**
>
> 怎么样，你现在应该彻底理解了什么叫“软件定义”、什么叫“**硬件加速**”了。所谓软件定义，就是利用通用 CPU，依靠一条条的机器指令来完成一个任务；而硬件加速，就是直接将这个任务中的每一步做成纯数字逻辑，不需要通用 CPU+代码，所有的逻辑、步骤都被固化到数字逻辑中，这样执行速度最快，效率最高，但是不具备灵活性，也就是“**可编程**”性，比如，一旦某个需求变化了，那么就需要重新设计数字逻辑，而并不能像软件定义那样直接改一改代码重新编译一下即可。冬瓜哥将在第 9 章中更详细的介绍硬件加速。

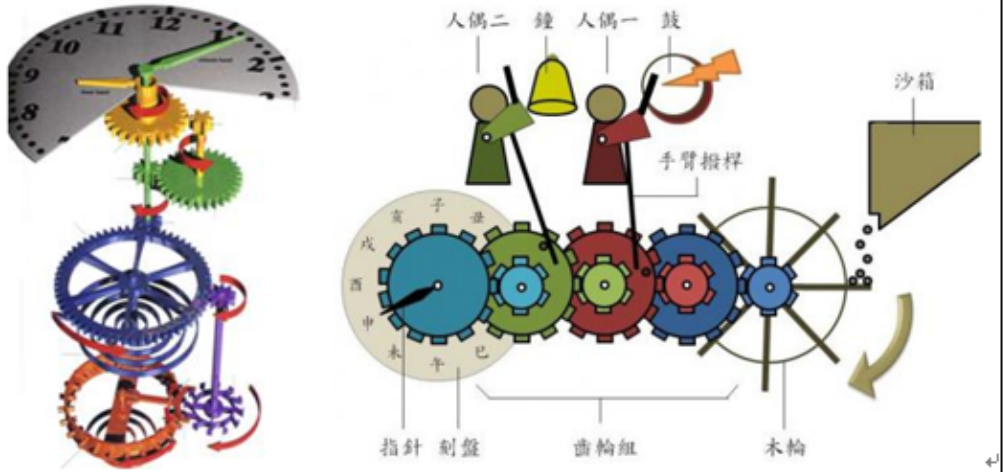
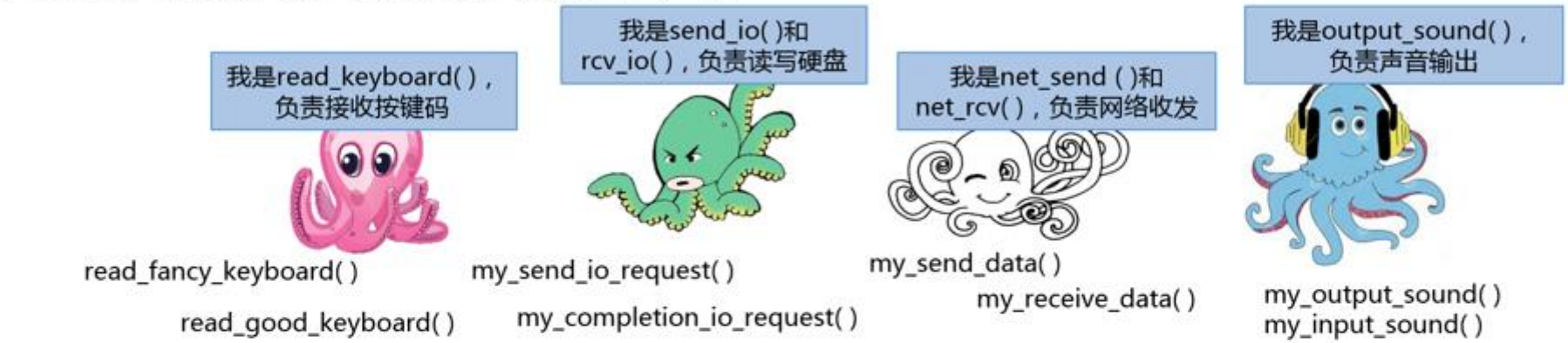


图 5-37 纯硬件执行的机械计时器

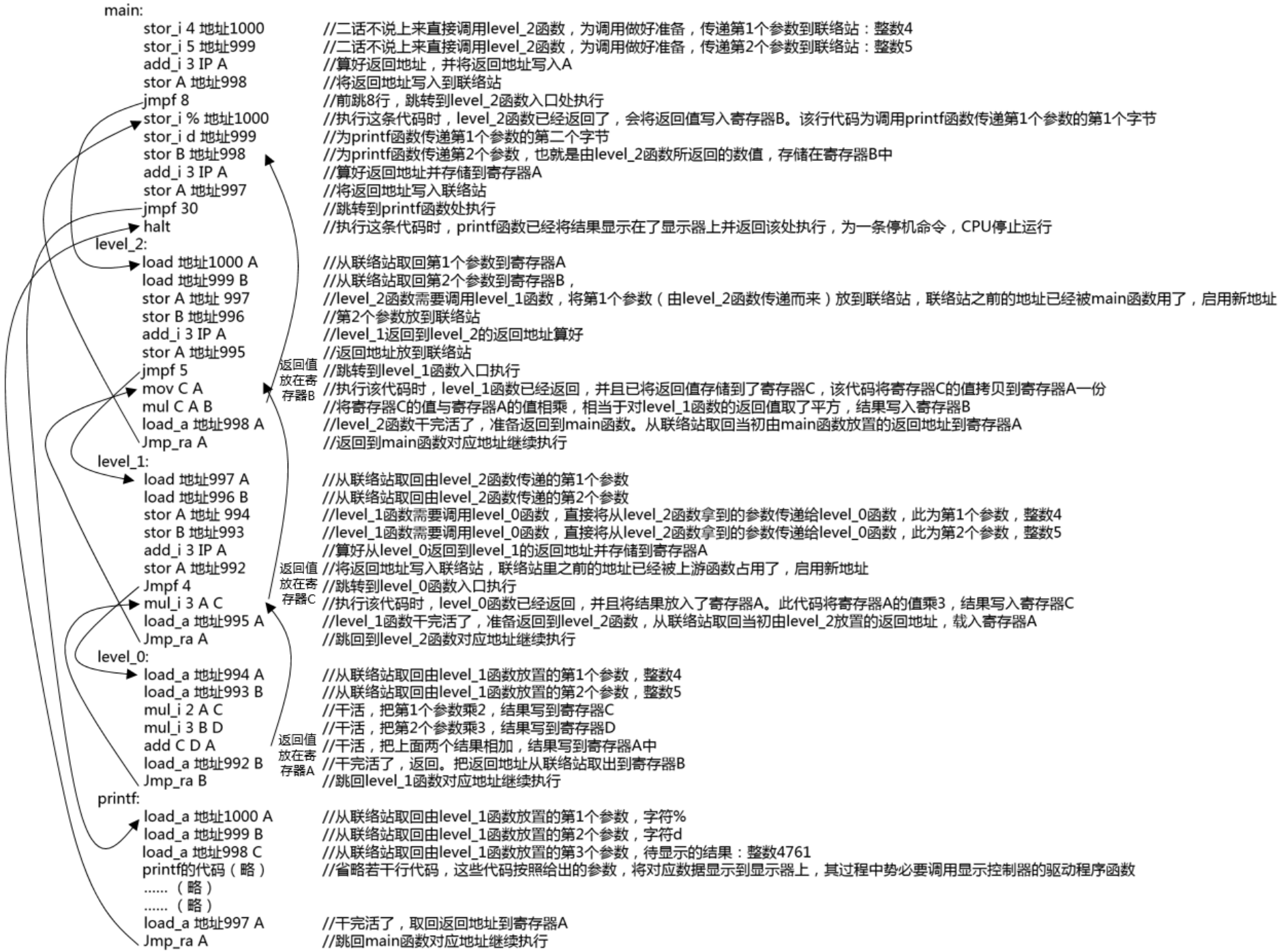
将图 5-36 与 5-37 中的机械计时器相比，是不是感觉到事物底层的相似性？最终，你又一次深刻领会到了 CPU+代码实现某个功能，与用纯数字逻辑实现之间的联系和区别。看到这里，你应当在脑海中自然回想起第 1 章的简易计算器，以及我们为了使之变得更灵活而设计的那个简易 CPU，并结合现在在计时器设计上我们反演了这个过程，如果能有一种奇妙的顿悟，那

read_keyboard( )函数会扫描该表中所有类型为“键盘”的设备从而让用户选择使用哪个键盘来输入，或者read_keyboard( )自行选择一个默认设备作为输入设备；刚才图 5-49 中的菜单，也是由程序扫描设备信息表中所有发声控制设备，从而让用户选择使用哪个设备输出声音。



设备名	厂商	类型	型号	版本号	设备ID	操作函数指针
ABC FancyBoard	ABC	键盘输入设备	FB880	1.1	0	内存地址A
PPT GoodBoard	PPT	键盘输入设备	PT500	5.0	1	内存地址B
XYZ SmartStorage	XYZ	大容量存储设备	XY100	2.5	2	IO发送函数：内存地址C IO完成函数：内存地址D
OK IntellegentNet	OK	网络控制器	OK330	9.0	3	发送函数：内存地址E 接收函数：内存地址F
OPQ UltraAudio	OPQ	声音控制器	UA2100	7.0	4	声音输出函数：内存地址G 声音输入函数：内存地址H

图 5-50 设备信息描述表


main:
stor_i 4 地址1000 //二话不说上来直接调用level_2函数，为调用做好准备，传递第1个参数到联络站：整数4
stor_i 5 地址999 //二话不说上来直接调用level_2函数，为调用做好准备，传递第2个参数到联络站：整数5
add_i 3 IP A //算好返回地址，并将返回地址写入A
stor A 地址998 //将返回地址写入到联络站
jmpf 8 //前跳8行，跳转到level_2函数入口处执行
stor_i % 地址1000 //执行这条代码时，level_2函数已经返回了，会将返回值写入寄存器B。该行代码为调用printf函数传递第1个参数的第1个字节
stor_i d 地址999 //为printf函数传递第1个参数的第二个字节
stor B 地址998 //为printf函数传递第2个参数，也就是由level_2函数所返回的数值，存储在寄存器B中
add_i 3 IP A //算好返回地址并存储到寄存器A
stor A 地址997 //将返回地址写入联络站
jmpf 30 //跳转到printf函数处执行
halt //执行这条代码时，printf函数已经将结果显示在了显示器上并返回该处执行，为一条停机命令，CPU停止运行
level_2:
load 地址1000 A //从联络站取回第1个参数到寄存器A
load 地址999 B //从联络站取回第2个参数到寄存器B，
stor A 地址 997 //level_2函数需要调用level_1函数，将第1个参数（由level_2函数传递而来）放到联络站，联络站之前的地址已经被main函数用了，启用新地址
stor B 地址996 //第2个参数放到联络站
add_i 3 IP A //level_1返回到level_2的返回地址算好
stor A 地址995 //返回地址放到联络站
返回值放在寄存器B
jmpf 5 //跳转到level_1函数入口执行
mov C A //执行该代码时，level_1函数已经返回，并且已将返回值存储到了寄存器C，该代码将寄存器C的值拷贝到寄存器A一份
mul C A B //将寄存器C的值与寄存器A的值相乘，相当于对level_1函数的返回值取了平方，结果写入寄存器B
load_a 地址998 A //level_2函数干完活了，准备返回到main函数。从联络站取回当初由main函数放置的返回地址到寄存器A
Jmp_ra A //返回到main函数对应地址继续执行
level_1:
load 地址997 A //从联络站取回由level_2函数传递的第1个参数
load 地址996 B //从联络站取回由level_2函数传递的第2个参数
stor A 地址 994 //level_1函数需要调用level_0函数，直接将从level_2函数拿到的参数传递给level_0函数，此为第1个参数，整数4
stor B 地址993 //level_1函数需要调用level_0函数，直接将从level_2函数拿到的参数传递给level_0函数，此为第2个参数，整数5
add_i 3 IP A //算好从level_0返回到level_1的返回地址并存储到寄存器A
stor A 地址992 //将返回地址写入联络站，联络站里之前的地址已经被上游函数占用了，启用新地址
返回值放在寄存器C
Jmpf 4 //跳转到level_0函数入口执行
mul_i 3 A C //执行该代码时，level_0函数已经返回，并且将结果放入了寄存器A。此代码将寄存器A的值乘3，结果写入寄存器C
load_a 地址995 A //level_1函数干完活了，准备返回到level_2函数，从联络站取回当初由level_2放置的返回地址，载入寄存器A
Jmp_ra A //跳回到level_2函数对应地址继续执行
level_0:
load_a 地址994 A //从联络站取回由level_1函数放置的第1个参数，整数4
load_a 地址993 B //从联络站取回由level_1函数放置的第2个参数，整数5
mul_i 2 A C //干活，把第1个参数乘2，结果写到寄存器C
mul_i 3 B D //干活，把第2个参数乘3，结果写到寄存器D
返回值放在寄存器A
add C D A //干活，把上面两个结果相加，结果写到寄存器A中
load_a 地址992 B //干完活了，返回。把返回地址从联络站取出到寄存器B
Jmp_ra B //跳回level_1函数对应地址继续执行
printf:
load_a 地址1000 A //从联络站取回由level_1函数放置的第1个参数，字符%
load_a 地址999 B //从联络站取回由level_1函数放置的第2个参数，字符d
load_a 地址998 C //从联络站取回由level_1函数放置的第3个参数，待显示的结果：整数4761
printf的代码（略） //省略若干行代码，这些代码按照给出的参数，将对应数据显示到显示器上，其过程中势必要调用显示控制器的驱动程序函数
.....（略）
.....（略）
load_a 地址997 A //干完活了，取回返回地址到寄存器A
Jmp_ra A //跳回main函数对应地址继续执行

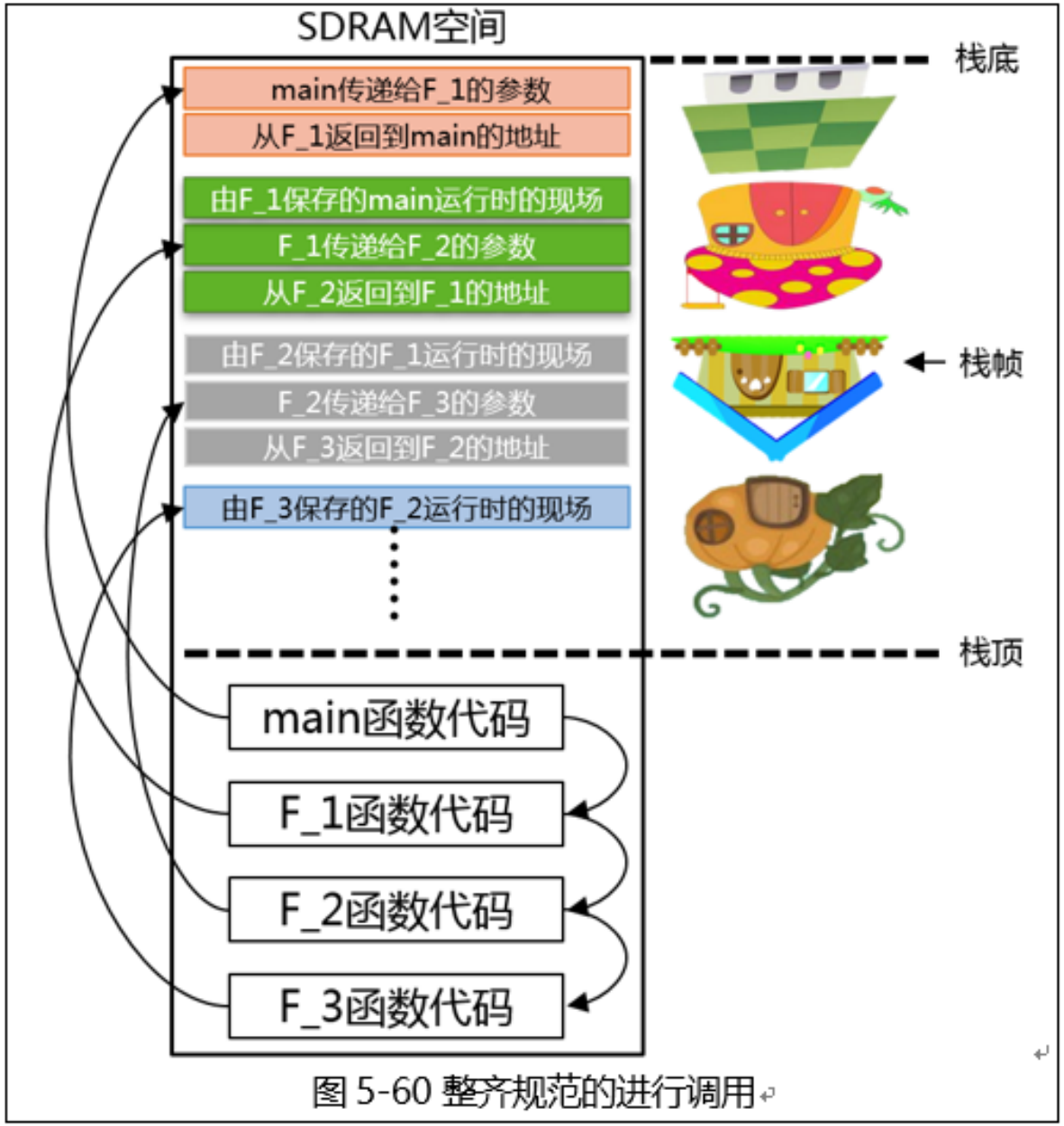


图 5-60 整齐规范的进行调用

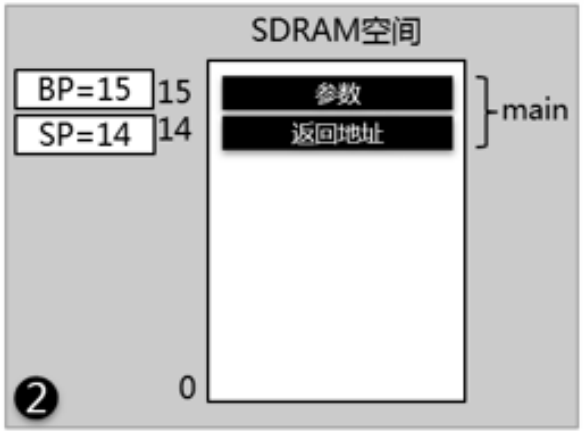
基于这个思路，人们把整个联络站称为“**栈（Stack）**”，就像客栈一样，给函数们提供一个歇脚和相互沟通的地方。而把栈中的每一间客房称为“**栈帧（Stack Frame）**”，函数把自己要传达给下游的参数以及返回地址一行一行的放到栈帧里。

函数们来住店，首领 main 先住了进来，然后就开始派活给其他函数，这老哥们下达完指令就开始睡大觉了，他委派的人可能继续委托其他人，每个人委托其他人之后都开始睡大觉，然后等待下游主动执行跳转指令跳回来，然后醒来继续干活。可以看到，栈的空间是不断扩大的，自顶向下。程序员需要注意，如果级连调用的函数过多，或传递的参数过多，而导致栈空间大到触碰了 SDRAM 低地址区域的代码、数据，那么会导致出错。

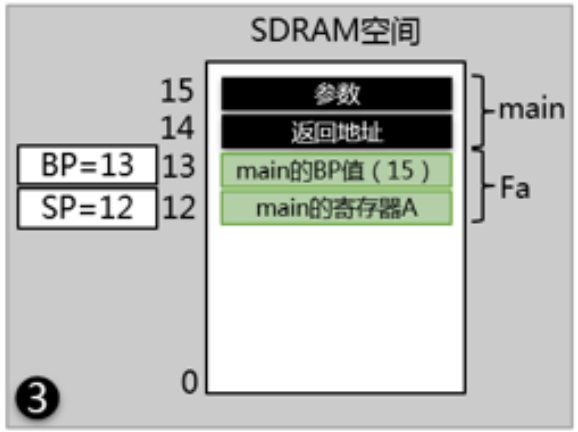
另外，栈空间扩大时一定是一条条追加上去的。比如某函数被调用执行后，第一件事就是保存上一个函数的现场，将数个寄存器值追加到**栈顶**部，当其需要调用其他函数时，再将数个参数追加写入栈顶部、返回地址写入栈顶部；如果其调用的函数继续向

个过程示意图如图 5-61 所示。Main 调用 Fa，Fa 又调用 Fb，Fb 不再调用其他函数。

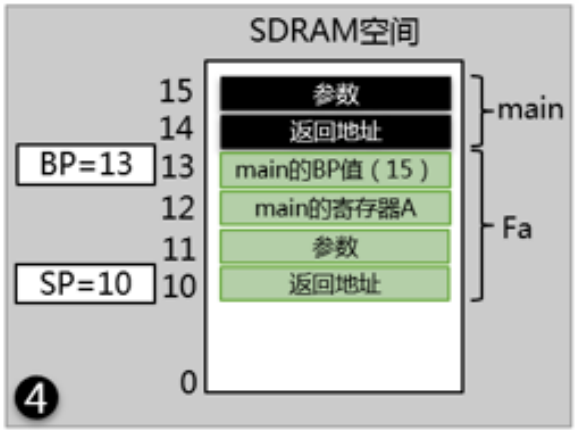
初始状态



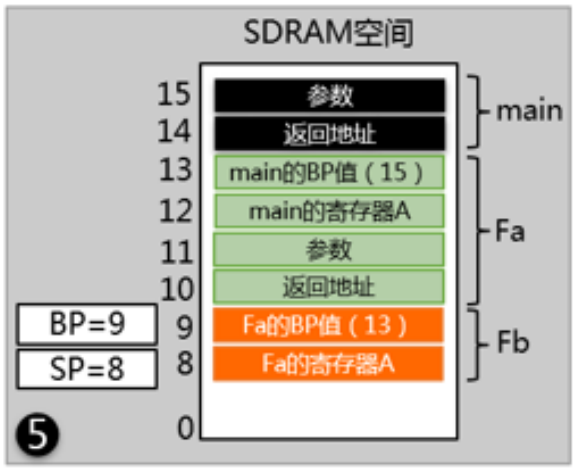
main准备调用下游函数



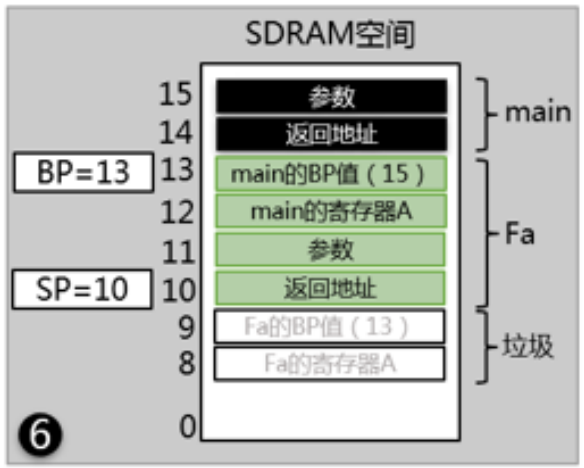
函数Fa开始执行，保护main的现场



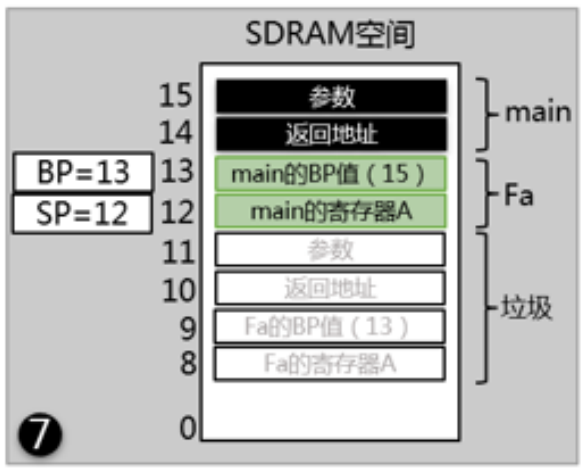
函数Fa准备调用Fb



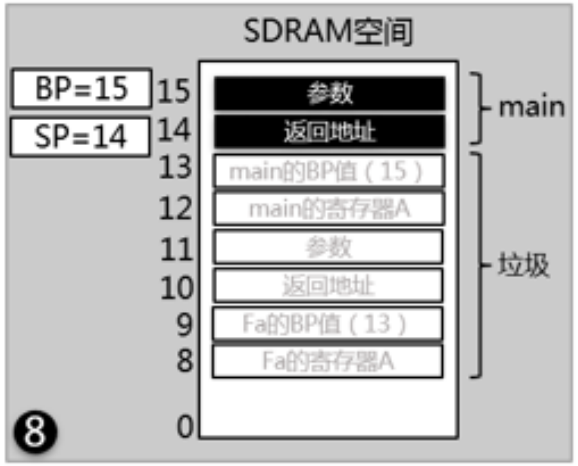
Fa调用Fb，Fb开始执行后



Fb返回到Fa跳转之前的瞬间



Fa回收栈空间中没用的空间



Fa返回到main跳转之前的瞬间

序。如图 5-64 所示，我们可以做个拼图游戏，看看编译器需要从 melonbro_func.c 文件中将哪个零件拿出并贴上。

原始程序代码文件prog.c

将prog.c调用到的位于melonbro_func.c中的函数代码融合到prog.c之后所生成的程序代码文件prog.cfull

图 5-64 设想中的程序代码融合过程

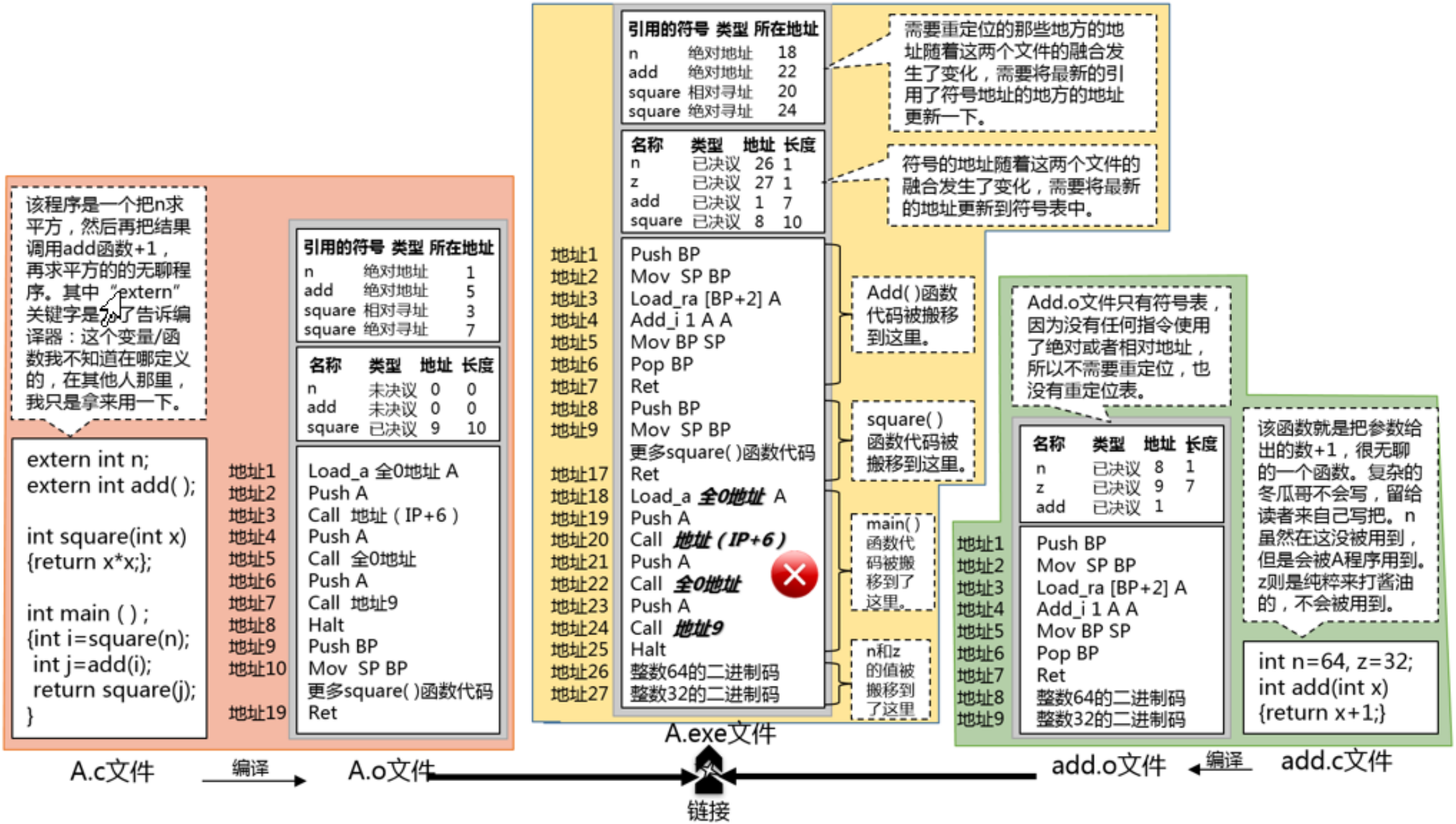


图 5-66 融合后按照符号定义和引用的最新位置更新符号表和重定位表中的地址列

录了文件中代码区域（代码段）以及数据区（数据段）的起始地址和长度。

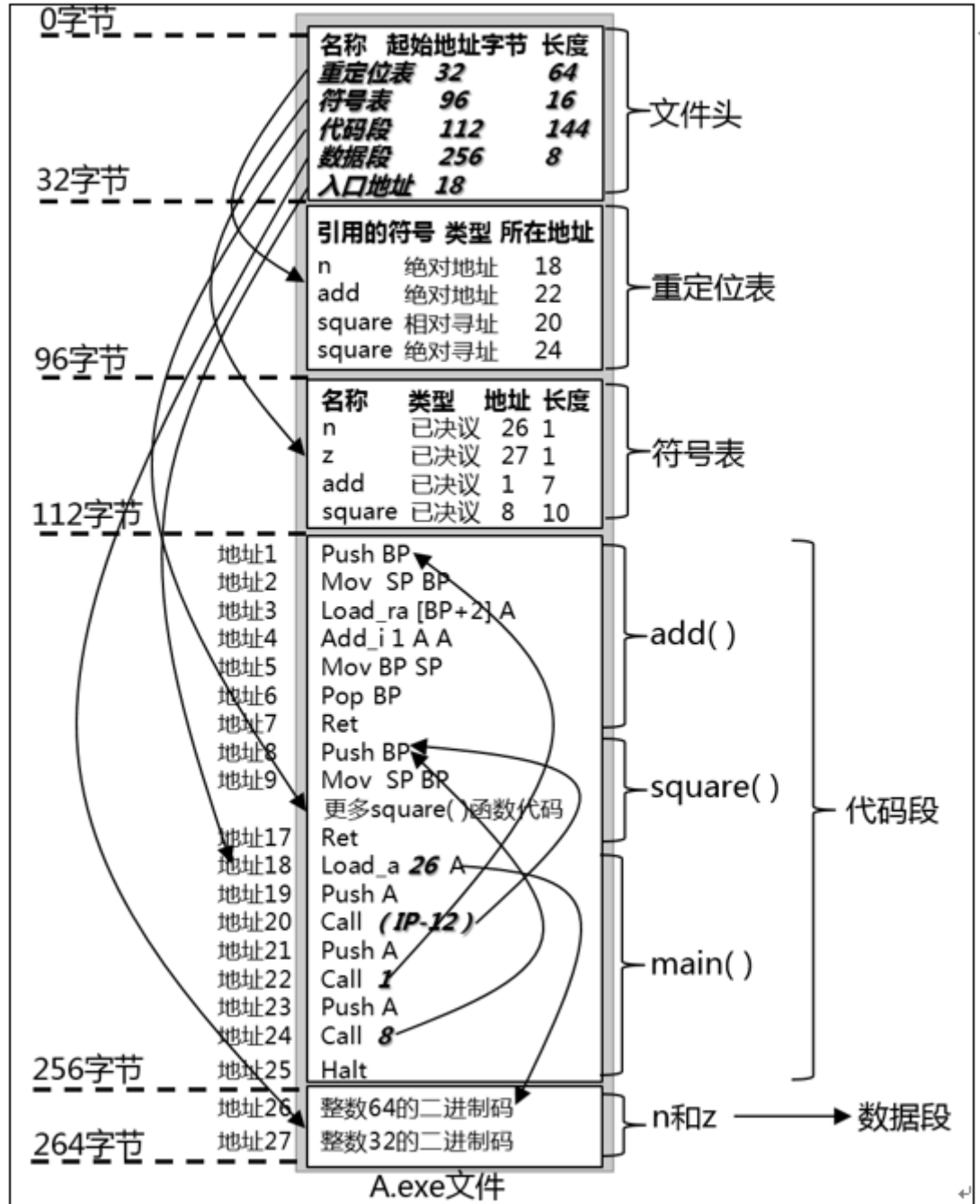


图 5-68 在文件头中记录更多信息

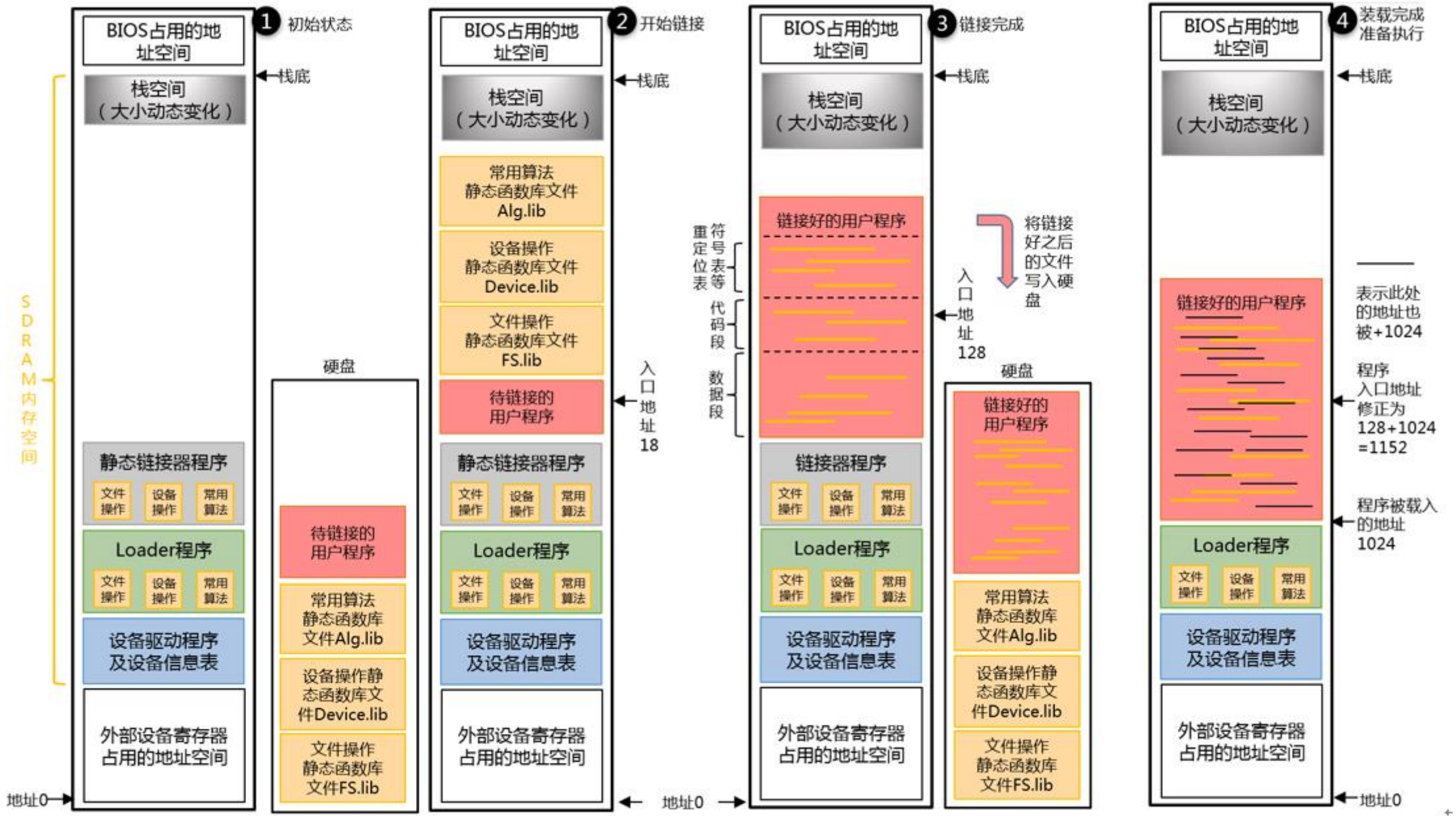
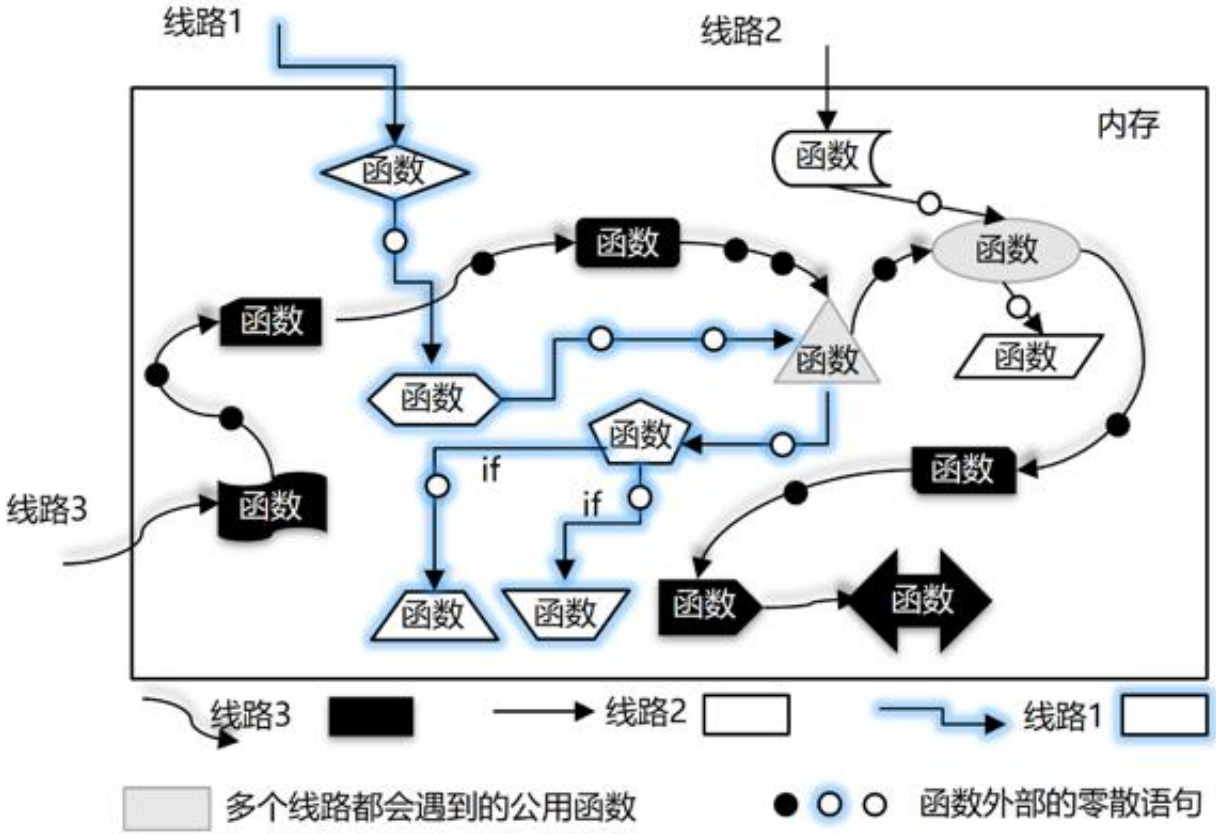


图 5-77 静态链接过程图示



一个线路/线程，就像一条项链（主函数）把一堆代码穿起来

图 5-78 线程示意图

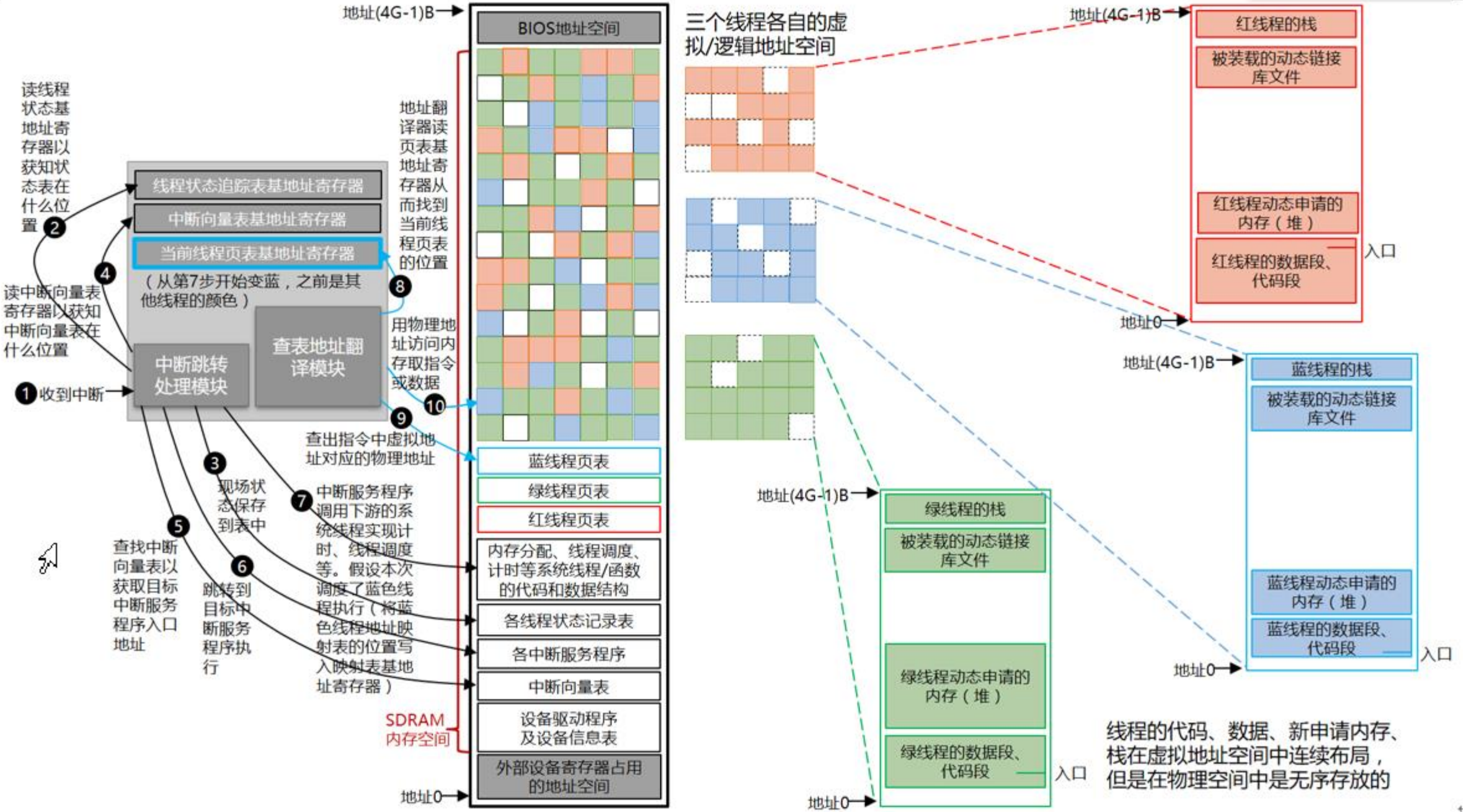


图 5-80 分页内存管理思想示意图

这里禁不住要问了，所谓“DOS 没有限制程序的访存范围”，这意思难道暗指 DOS 是可以去限制的？怎么限制？正如我们前文中所属，当操作系统的 Loader 程序让 CPU 跳转到用户程序运行之后，整个 CPU 就是在运行用户程序了，CPU 此时完全受到用户程序代码的控制，让它走东绝不往西，此时操作系统代码只是静静地呆在内存里起不到任何作用，此时只有靠 CPU 来检查和防止越界。要想实现这个功能，必须将当前执行的代码可访问的地址范围限制在某个区域中，比如从地址 1024 开始的长度 2048 字节的这 2KB 区域中，为了支持这个功能，CPU 必须提供至少两个寄存器，一个用于存放该区域的基地址，也就是上述的地址 1024，另一个用于存放长度，也就是上述的 2048。在操作系统 Loader 程序跳转到用户程序执行之前，必须使用对应的机器指令来更新这两个寄存器，告诉 CPU：“兄弟，后续任何代码只能在这个区域内执行，一旦越界你就报异常，反过来执行我提供的异常服务程序，昂~！”，然后再跳转到用户程序执行，此时用户程序就被框住了。然而，这一招只能防君子，却防不了小人。程序（或者说黑客们）是不会善罢甘休的，程序是不是也可以用对应的机器指令来更新这两个寄存器呢？比如将基地址更新为 0，长度更新为 1GB，从而逃脱限制？这就相当于马路上有个栏杆，守规矩的司机碰到栏杆就会避让，但是不守规矩的可以下车把栏杆往边上移动一下然后说：“你看，我没违规啊”。

设计者早就考虑到这个问题了，所以，当操作系统 Loader 程序让 CPU 跳转到用户程序执行之前，还要告诉 CPU 一件事：“兄弟，接下来任何想更改你的一些特殊控制寄存器的指令你都不执行，一旦发现则直接报异常转回来执行我提供的异常服务程序，昂~！”。这一招下去，相当于接下来执行的程序再也无法逃脱出这个框框了。正如前文中所述，CPU 的指令集中有一些属于特权指令，只有操作系统的代码可以执行特权指令。任何尝试执行特权指令的用户程序，CPU 此时会直接中断程序的运行，跳转到异常服务程序执行，后者直接把该程序终止掉，并弹出窗口或者文字告诉你“刚才这个程序不老实想搞事情，被终止了”，当然，说的好听一点是“非法操作：尝试访问了 xxx 地址”。

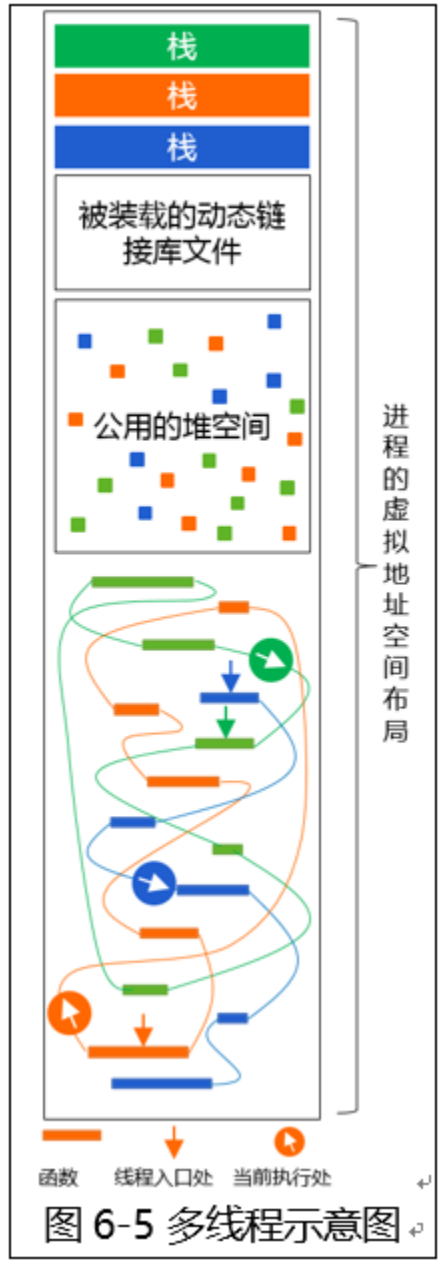


图 6-5 多线程示意图

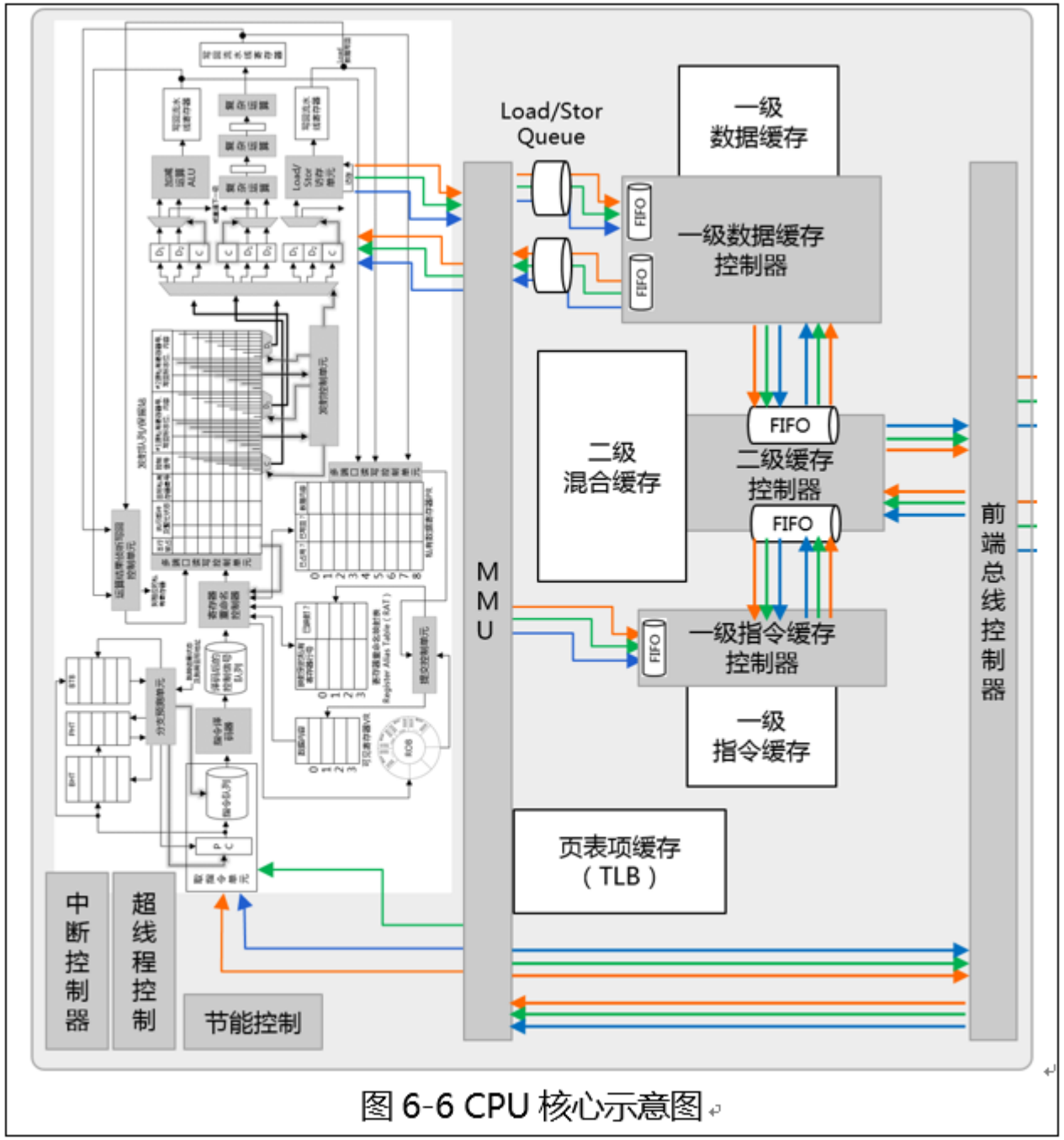


图 6-6 CPU 核心示意图

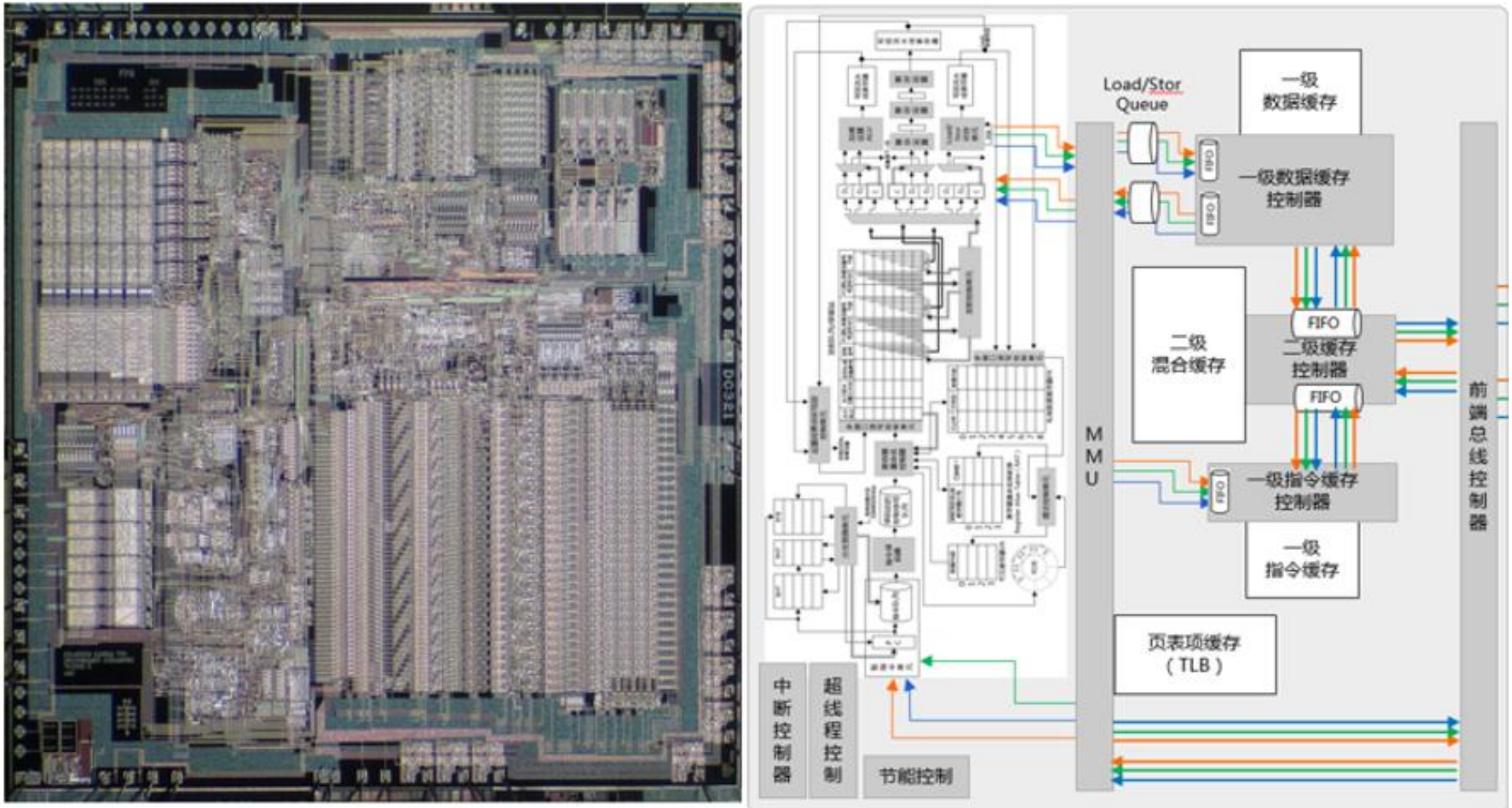


图 6-7 实际的 CPU 核心芯片显微图与架构示意图比较

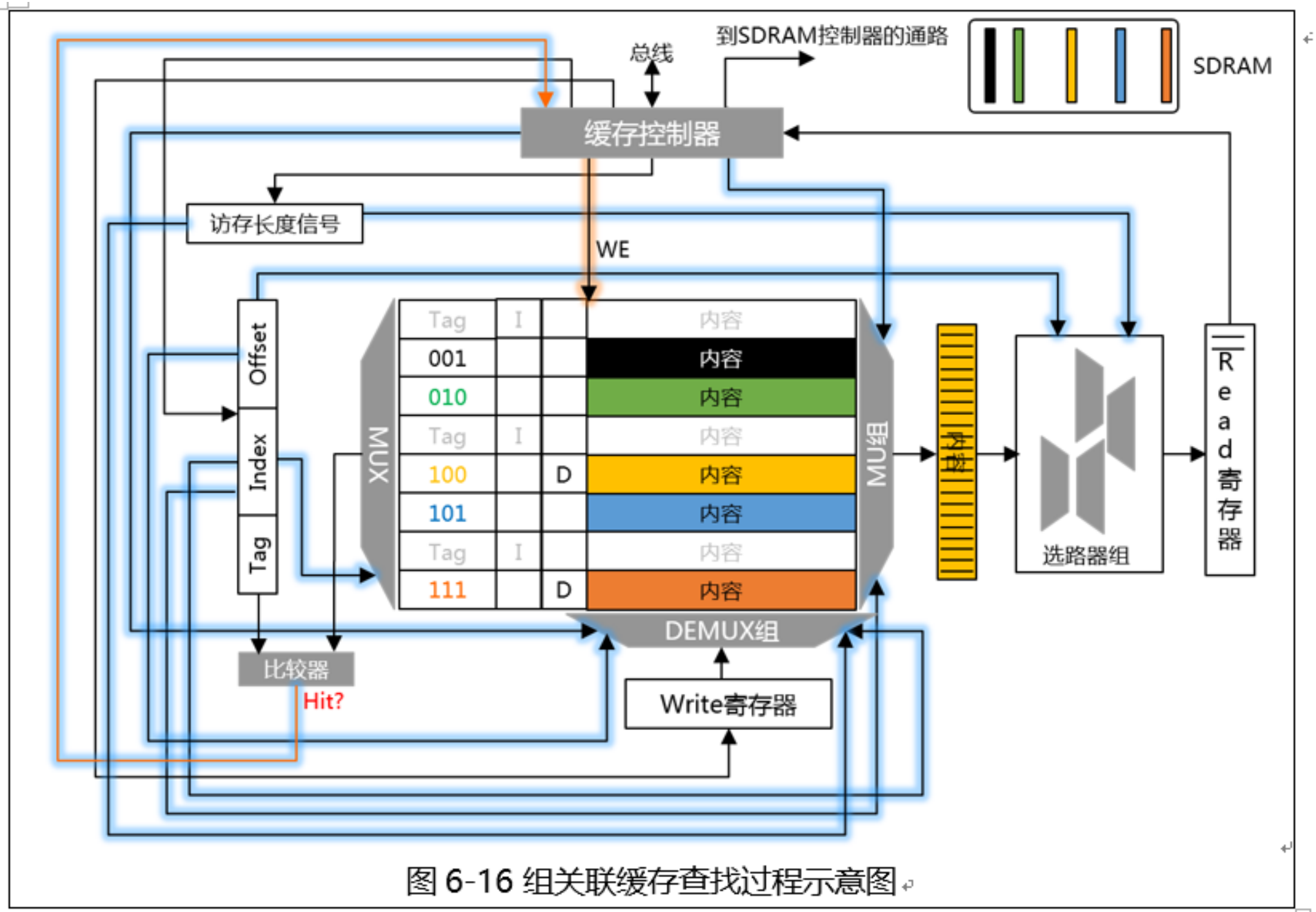


图 6-16 组关联缓存查找过程示意图

不需
地址
直接
出对
然后
段比
出数
SDR

路更
的运
较高
种方
存行
置，
行号

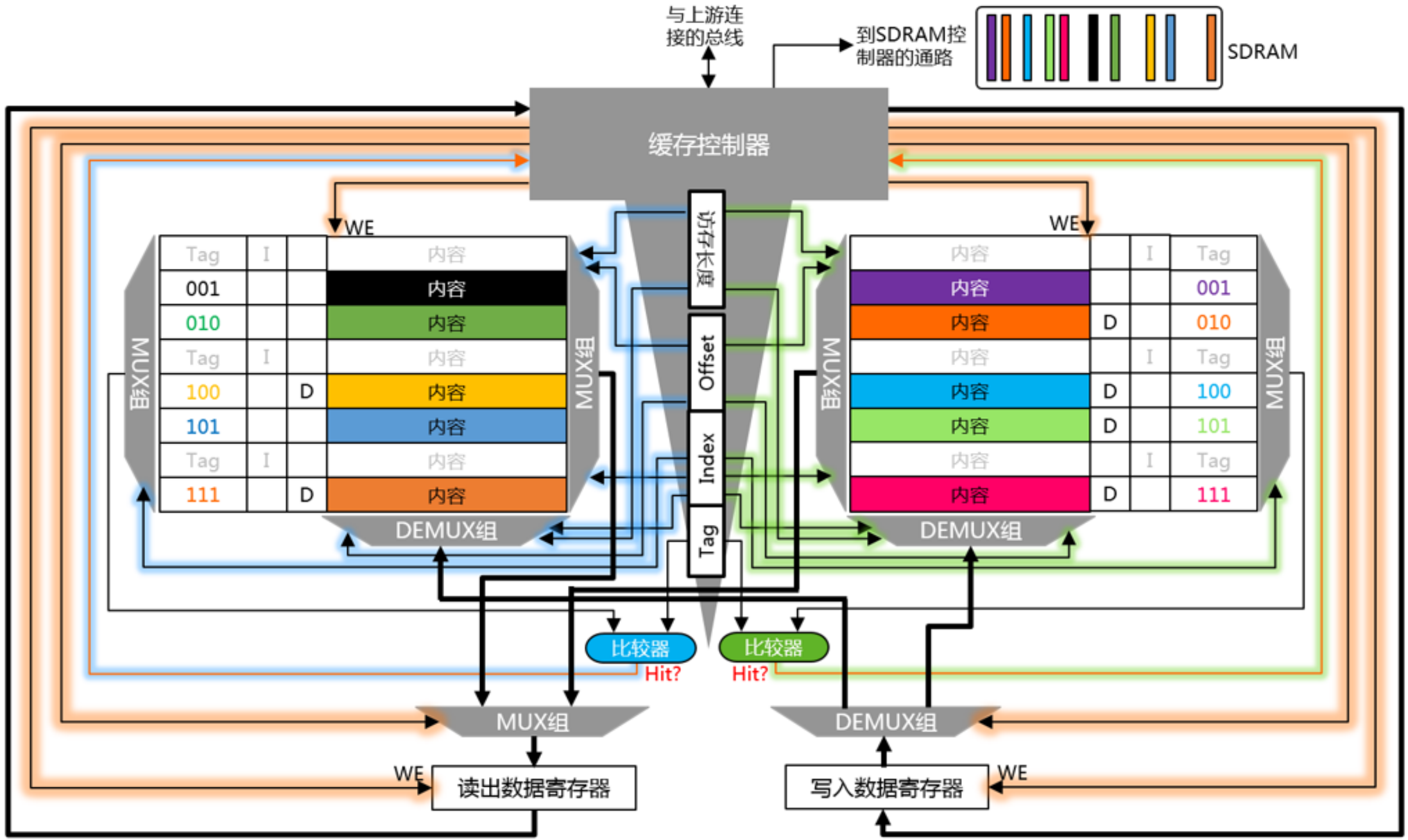
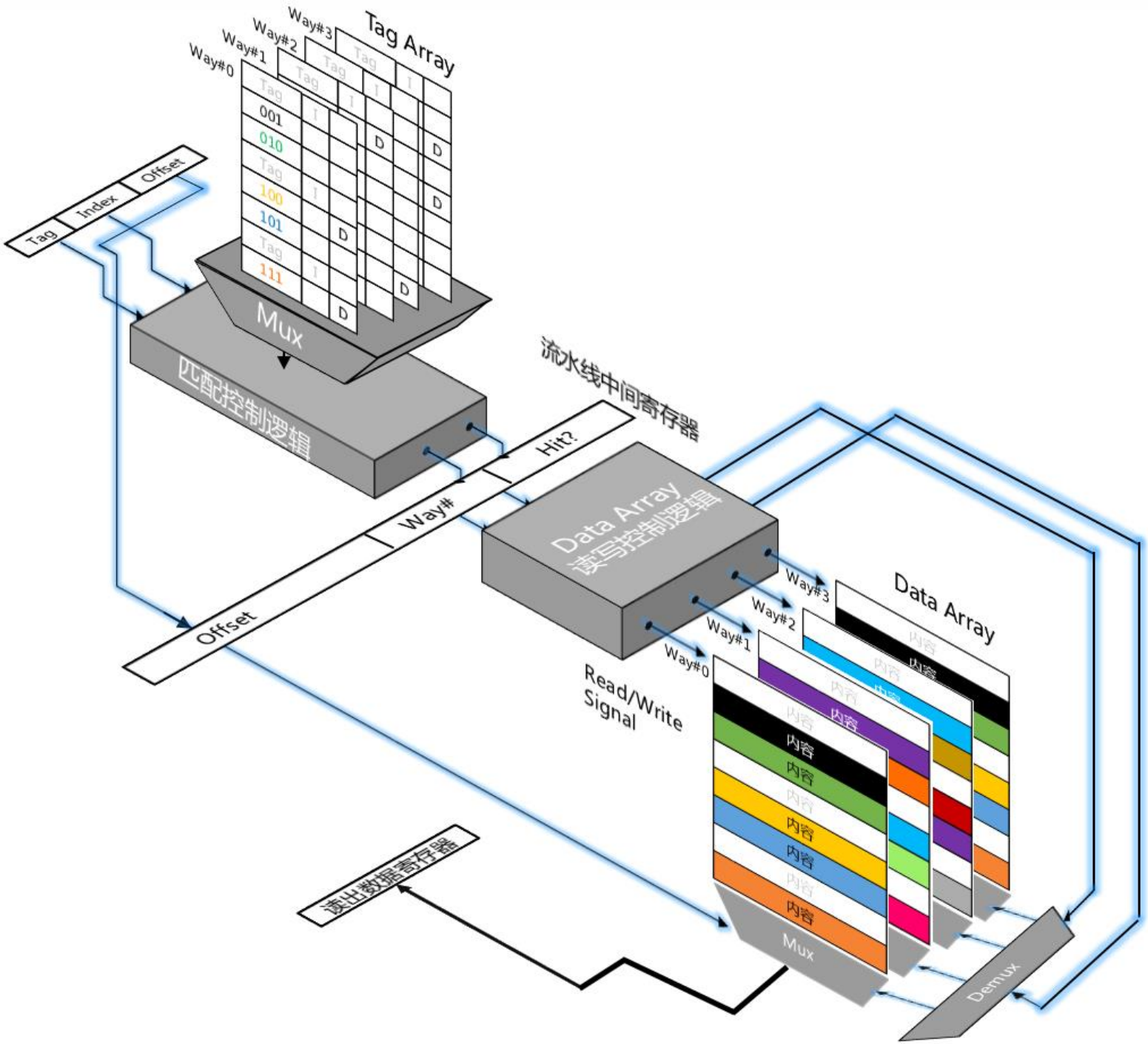


图 6-19 2 路组关联缓存的硬件架构示意图


Tag Array
Way#0
Way#1
Way#2
Way#3
Tag
Index
Offset
001
010
100
101
111
D
Mux
匹配控制逻辑
流水线中间寄存器
Hit?
Way#
Offset
Data Array
读写控制逻辑
Read/Write
Signal
Data Array
内容
Mux
Demux
读出数据寄存器

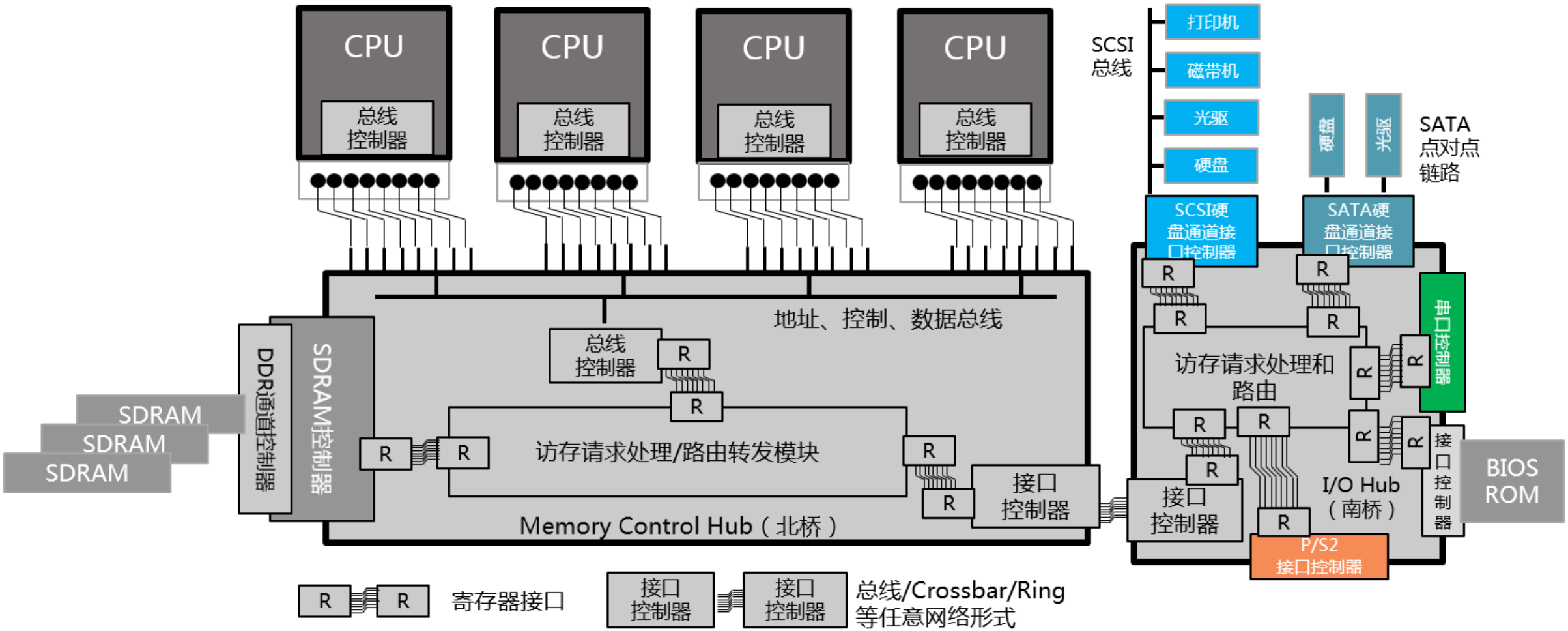
6.2.16 非阻塞 Cache 与 MSHR

对于 CPU 核心，每一条机器指令对它来讲就是一个任务；对于缓存来讲，核心的 L/S 单元发出的访存请求也是一个任务。核心把计算任务交给 ALU 来执行，把访存任务交给 L/S 单元来执行，L/S 单元再把任务交给缓存控制器来执行。在 LSQ 中，可能有多个 Load/Stor 类指令正在等待结果返回，这些请求会按照顺序一条一条的被发送给 L1 缓存控制器去执行。那么假设，排在 LSQ 中队首的某个 Load 指令被发往了 L1 缓存控制器，结果控制器的查找结果是本次访存请求不命中，也就是 Miss 了，那么 L1 缓存控制器将会等待十个左右的时钟周期才能从 L2 缓存中拿到数据，那么在这 10 个左右时钟周期内，难道 L1 缓存个控制器就无事可做了么？显然不划算，我们多么希望 LSQ 中后续的访存请求能够利用这个间隙被缓存控制器执行！

显然，我想所有人都在暗想：如果 L1 缓存控制器能够记录一个状态，比如哪个请求未命中，该请求访问的是哪个地址等等；然后继续执行后续的 Load/Stor 请求，如果又 Miss 了，则继续记录一条追踪记录。当数据返回时，比较这些追踪记录看看是哪个访存请求终于返回了数据，然后缓存控制器就在于 L/S 单元连接的总线上向 L/S 单元返回对应的数据，并对所有的返回数据加以区分，以告诉 L/S 单元这次返回的是哪一条访存请求的数据，比如通过 ID 标记机制，每个请求都有各自的 ID，也可以直接在总线上通告该数据要被载入到的目标寄存器号。L/S 单元拿到返回的数据之后，根据 LSQ 中记录的访存请求的目标数据寄存器号，直接将数据载入对应的核心的数据寄存器从而完成访存操作。当然，这里其实是返回到了内部的私有寄存器中，会被流水线结果侦听单元捕获到，然后更新到保留站以及私有寄存器中的记录上，从而依赖这些数据的、依然等待在保留站中的其他指令就可以在下一个时钟周期内被发射控制单元判断为符合执行条件，从而被调度执行，这个过程建议回顾流水线那一章里的流程图。

这种支持前序访存请求即便 Miss 了也不影响后续访存指令继续执行的缓存被称为**非阻塞缓存，None Blocking Cache**。上面提到过的用于追踪各个 Miss 的访存请求的执行状态的硬件模块被称为 **MSHR（Miss Status Holding/Handling Register）**。

图 6-27 MSHR 寄存器组


CPU
CPU
CPU
CPU
总线
控制器
总线
控制器
总线
控制器
总线
控制器
SCSI
总线
打印机
磁带机
光驱
硬盘
硬盘
光驱
SATA
点对点
链路
SCSI硬
盘通道接
口控制器
SATA硬
盘通道接
口控制器
地址、控制、数据总线
总线
控制器
R
R
R
R
R
R
R
R
R
R
R
R
R
R
R
R
R
R
R
R
访存请求处理/路由转发模块
访存请求处理和
路由
串口控制器
SDRAM
SDRAM
SDRAM
DDR通道控制器
SDRAM控制器
接口
控制器
接口
控制器
接口
控制器
I/O Hub
（南桥）
接口
控制器
BIOS
ROM
P/S2
接口控制器
Memory Control Hub（北桥）
R
R
寄存器接口
接口
控制器
接口
控制器
总线/Crossbar/Ring
等任意网络形式

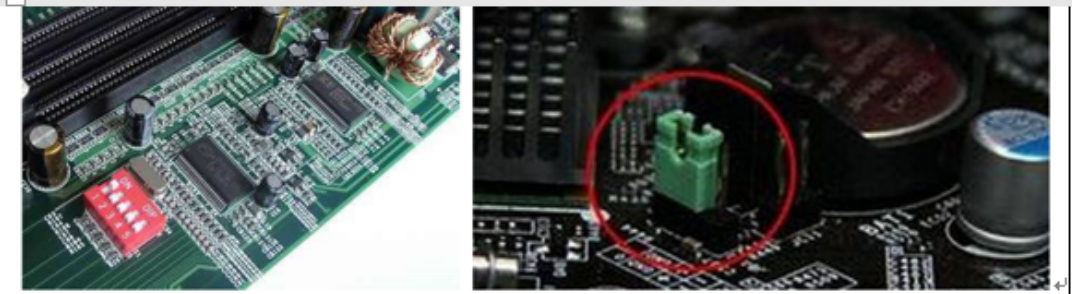


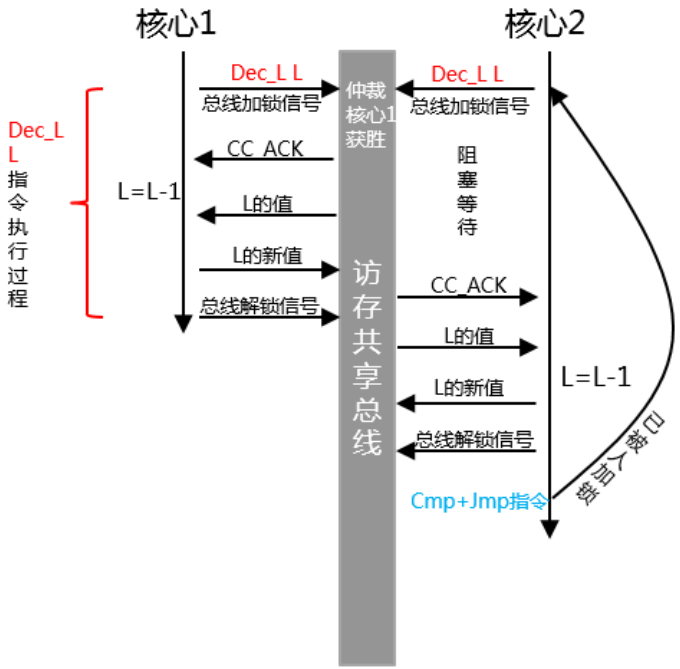
图 6-91 输入开关及跳线

在早期，很多参数都是通过这种办法输入到电路中去的，包括有些 I/O 总线上的节点地址 ID，都要用跳线帽来短接对应的阵脚从而让电路区分不同的设备。如图 6-91 所示为早期主板上的用于配置这些参数的开关。这些开关对应的电路会直接输入到目标模块中，目标硬件模块每次加电之后，直接从这些输入开关获取其所表示的 0/1 信号，将其直接锁存到内部的寄存器中，这样，与将这些寄存器映射到系统全局地址空间，访存网络中配好路由信息，然后用机器指令的方式操纵 CPU 来写入 ID 到这些寄存器的效果是一样的。

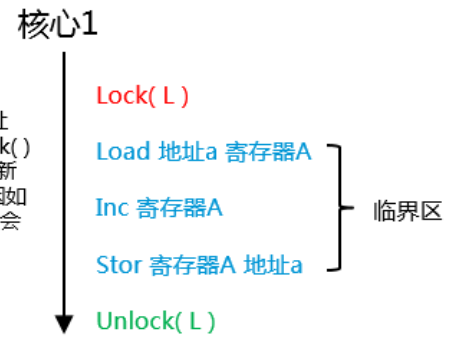
这种操作非常笨拙，人们总是不断追求更方便的方式。于是人们想出了另外一种方式，用另一台计算机运行某个程序，该计算机一方面负责通过 Web 页面等方式与用户交互（当然，用户也需要在用一台计算机打开浏览器来访问这台计算机的 Web 页面），另一方面则与需要被配置的硬件电路对接，对接方式可以通过 i2c 等串行接口方式，从而可以将用户发送过来的配置请求，转换为在 i2c 总线上传递的信息，发送给目标电路，目标电路从 i2c 总线上接收这些配置信息，然后配置到自己的寄存器中，完成分区或者各种其他操作。由于这台计算机需要用 i2c 与待配置的电路模块相连，所以它的实际形态是一片 SoC（System on Chip），集成了 CPU、SDRAM、i2c 接口控制器、以太网控制到一个单一芯片中，该芯片本身就是一台小计算机，与它要配置的那台大计算机放在同一张主板上，其内部运行着程序，包括底层的支撑程序（操作系统、各种控制器驱动程序）、Web 网页服务端程序、后端向 i2c 总线发送配置信息的程序等。Web 服务程序通过以太网接收用户通过网页访问方式发来的配置请求。经过这样的设计，用户再也不用去打开机箱，按照说明手册搬动开关去向电路输入参数了。

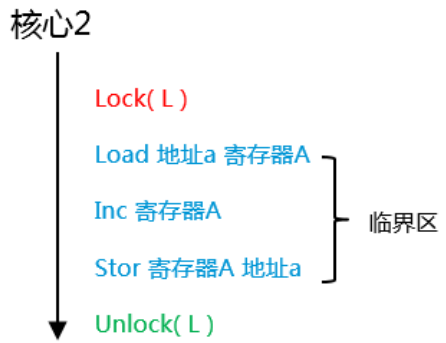
这个专门用于对计算机做更加底层的配置或者监控的片上小计算机，被称为 **BMC，Baseboard Management Controller，基板控制器**。如图 6-92 所示为当前比较流行的 BMC 芯片，可以看到其下方的一片 1GB 容量的 SDRAM，是供 BMC 运行使用的内存。

当前的高端服务器计算机全都使用 BMC 来做基本配置和监控，其不仅仅是配置这些初始参数，其还可以直接对计算机机箱内的其他模块做控制和监控，比如风扇、电源等。只要系统电源有达标的电压




```
Lock( ) {
    Dec_L L的内存地址
    //试图将L减1。如左侧示意图所示。这一步有多个子操作，会向总线申请针对该地址
    //的锁定，不会被打断。要深刻理解，这条指令执行时会对总线加锁，这个锁与Lock( )
    //整体的锁机制是完全不同的两个层面的锁。该指令结束之前会确保将对L地址的最新
    //数据同步到所有核心缓存中之后，再向总线通告解锁该地址，这一步很关键。正因如
    //此，x86体系结构下一切带锁的访存指令结尾必须等待缓存同步过程结束，或者说会
    //自动完成缓存同步

    Jmpnn_f  4
    //如果结果不是负值（nn，none negative），那一定是0，证明之前的值是1，
    //证明其他核心并没有持有锁，抢锁成功，则直接前跳4行到return语句退出函数

    Cmp 1 L的内存地址
    //如果上一步结果是负值，表示之前的值或是0或是负值
    //抢锁不成功，所以执行到该行。该行读出L看看是不是被解锁了

    Jmpz_b 3
    //如果Cmp的结果是0，证明L已经被人解锁，则回跳3行到Dec指令开始抢锁

    Jmp_b 2
    //如果Cmp指令的结果不是0，则证明L的值依然不是1，那就是尚未被解锁
    //则执行到该行，回跳2行到Cmp指令处，继续探查L是否已被解锁

    return
    Unlock ( ) { Stor_i  1  L的内存地址; }
}
```

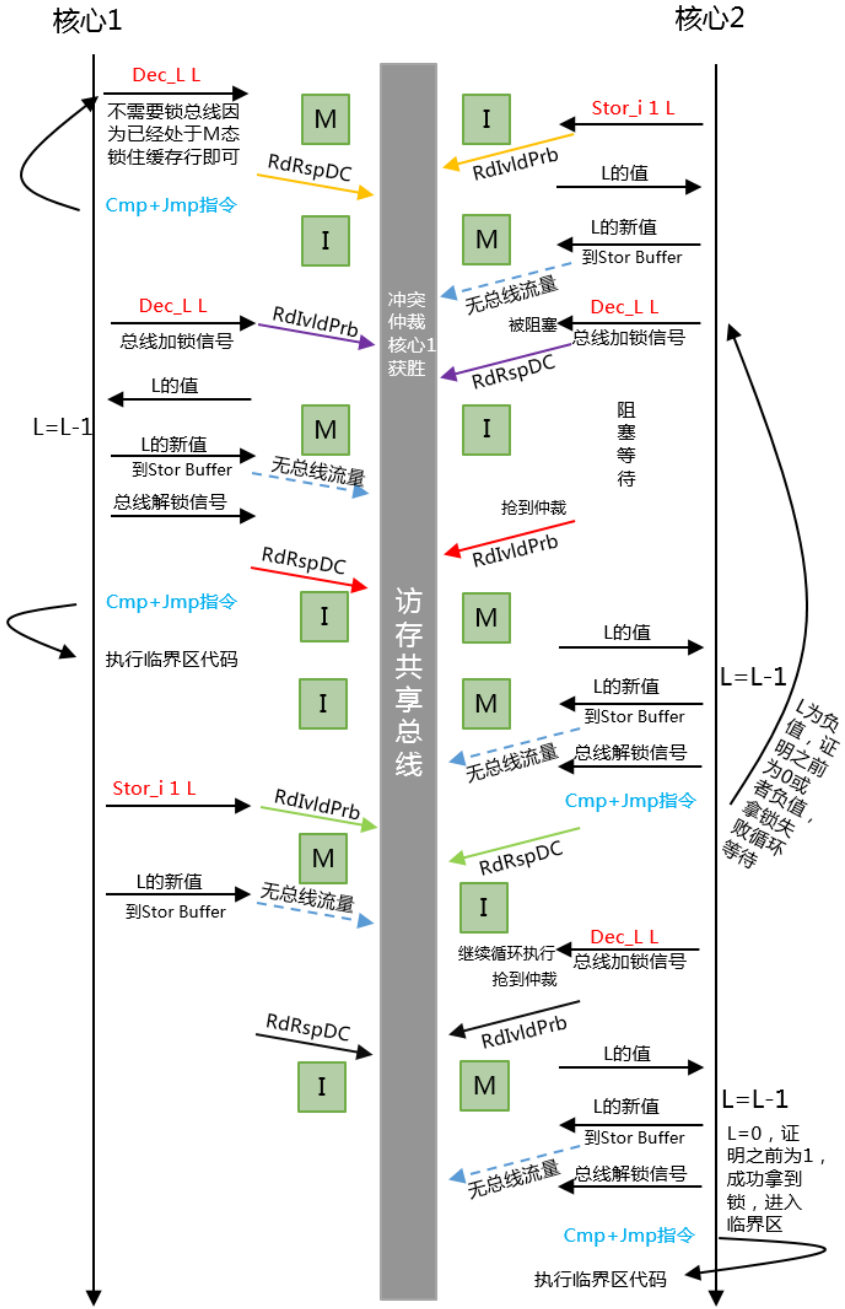


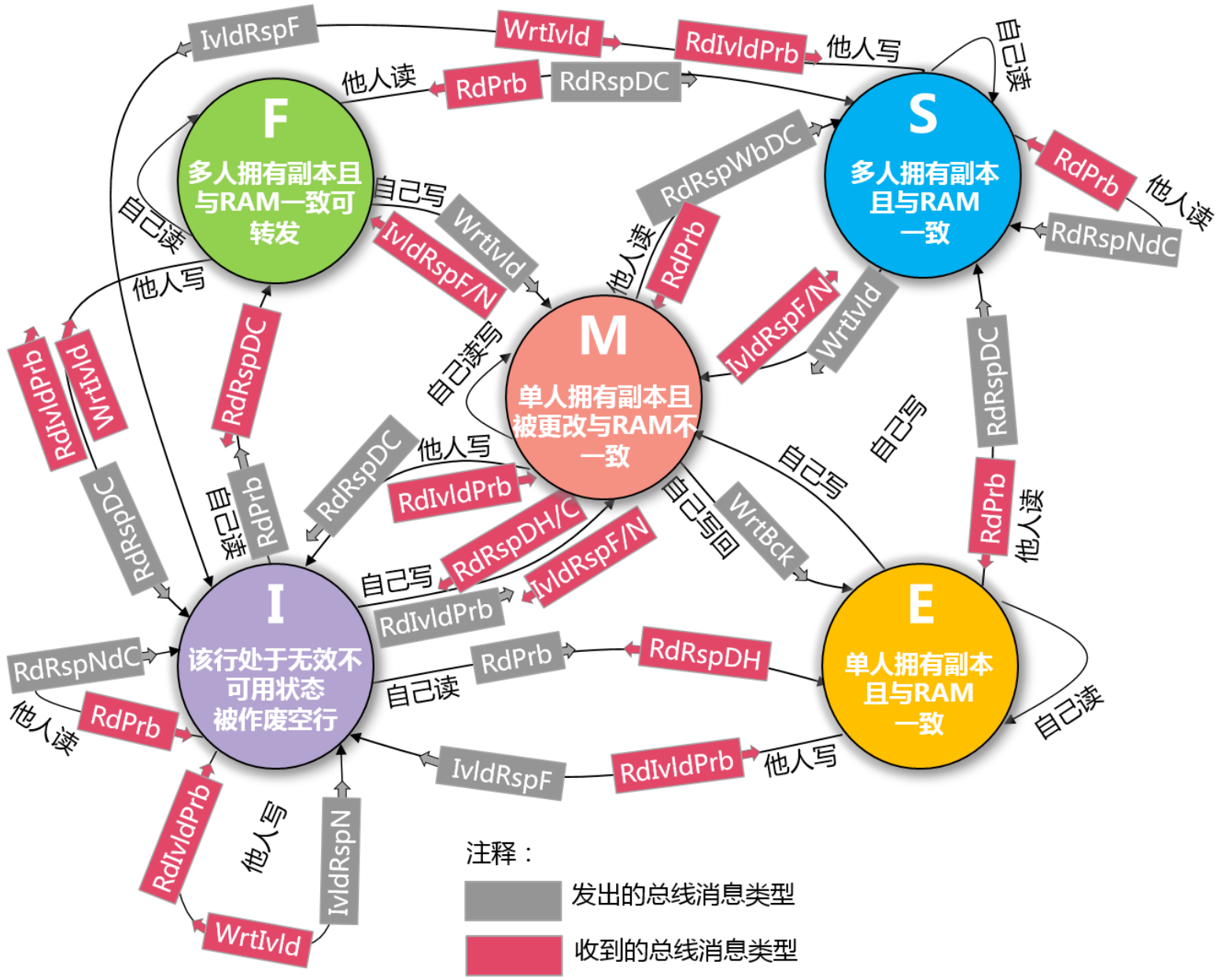


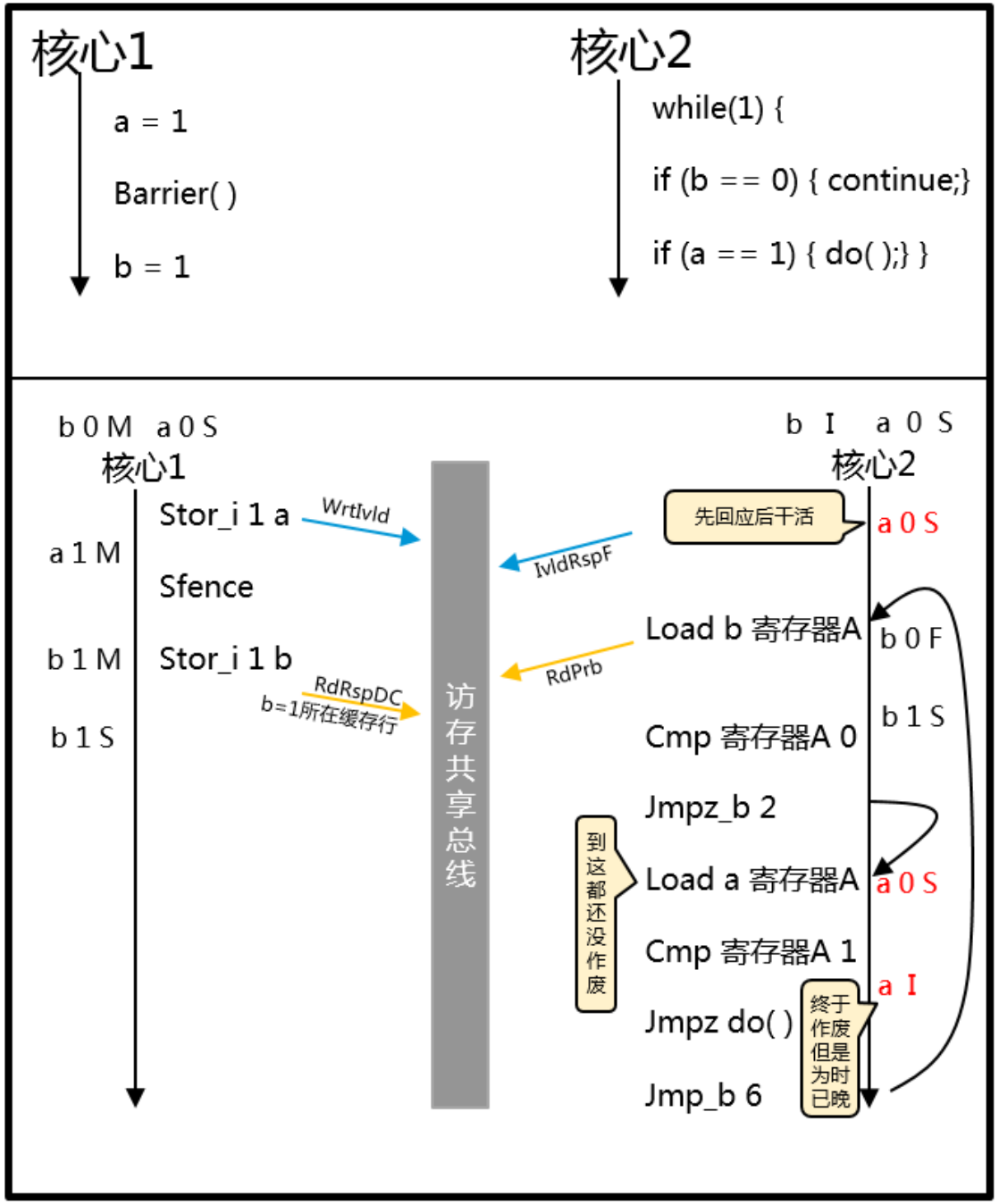
Lock函数退出后才能有机会执行后续的代码。抢不到锁，就会在Lock函数内部进入无限循环等待

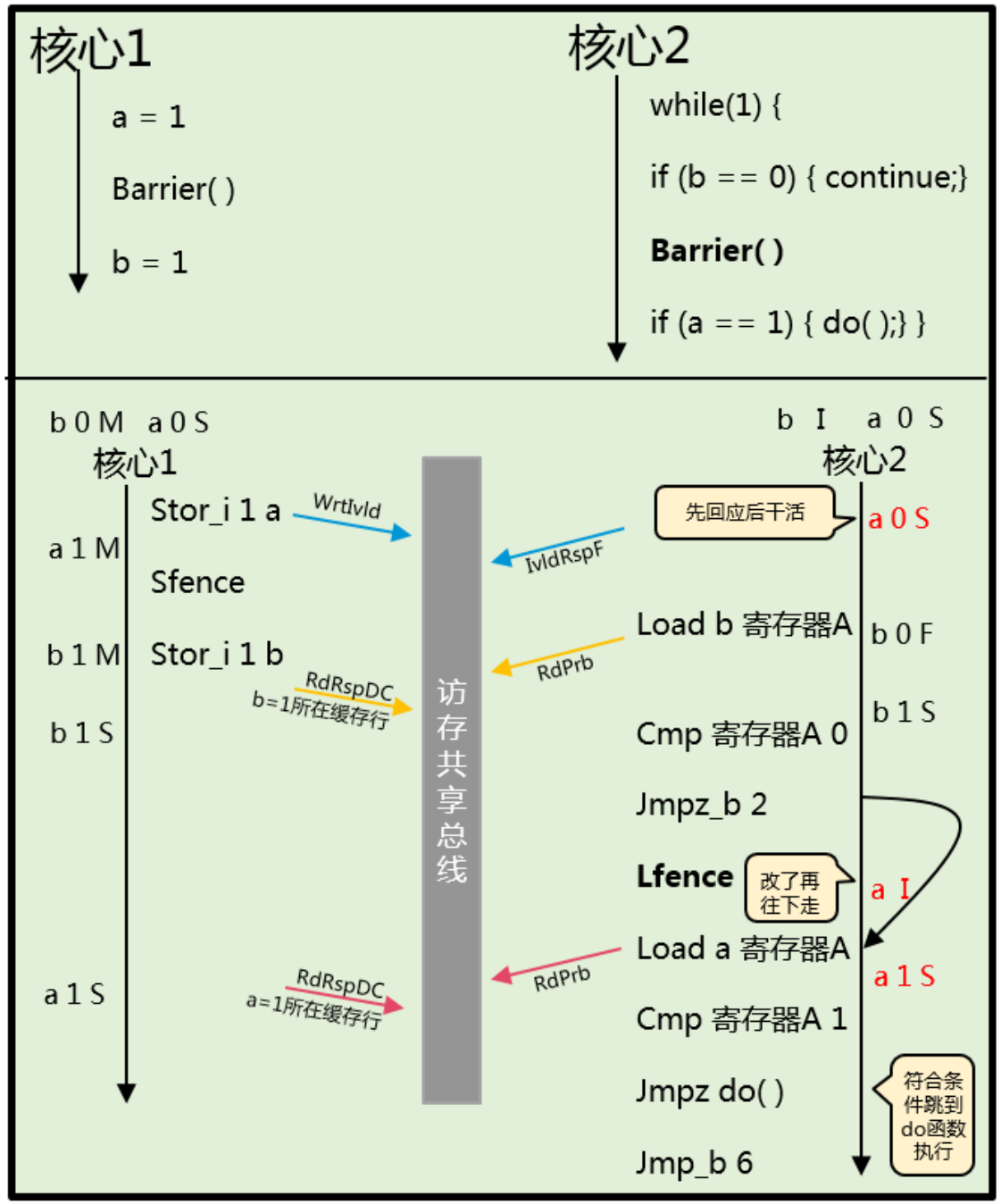
让子弹飞

有了，你看这样行不行，每个线程在获取到锁之后，先不急着去访问共享变量，先让子弹在访存网络上飞一会，估摸着差不多都命中目标了，再去访问。嗯，这的确是个很豪爽的野路子，但是大侠的这套方法略显粗糙，不够儒雅，什么时候子弹飞到了，完全凭您的经验，判断早了，程序出错，判断晚了，等的太久又影响了性能。大侠是体面人，是否可以构建一套既不需要 CPU 每一条访存指令都保障 0 延迟，又可以随时控制 CPU 仅当同步完成之后再往下走的机制呢？兄弟们，放枪！（劈啪噼啪）大哥，怎么还不解锁？别急，让子弹飞一会！一共几节车厢？四节！听了几声响了？3 声了！第 4 声也响了！兄弟们，解锁！让友军一睁眼就能看到那些铿锵的弹痕吧！


核心1
核心2
Dec_L L
不需要锁总线因为已经处于M态锁住缓存行即可
Cmp+Jmp指令
RdRspDC
Stor_i 1 L
RdIvldPrb
L的值
L的新值
到Stor Buffer
无总线流量
被阻塞
总线加锁信号
冲突仲裁核心1获胜
L=L-1
总线解锁信号
阻塞等待
抢到仲裁
执行临界区代码
L为负值，证明之前为0或者负值，拿锁失败循环等待
继续循环执行抢到仲裁
L=0，证明之前为1，成功拿到锁，进入临界区
执行临界区代码
访存共享总线


F
多人拥有副本且与RAM一致可转发
S
多人拥有副本且与RAM一致
M
单人拥有副本且被更改与RAM不一致
I
该行处于无效不可用状态被作废空行
E
单人拥有副本且与RAM一致
IvldRspF
WrtIvld
RdIvldPrb
RdRspDC
RdPrb
RdRspWbDC
RdRspNdC
IvldRspF/N
RdRspDH/C
RdRspDH
WrtBck
RdIvldPrb
RdRspNdC
IvldRspN
他人读
他人写
自己读
自己写
自己读写
自己写回
注释：
发出的总线消息类型
收到的总线消息类型


核心1
a = 1
Barrier( )
b = 1
核心2
while(1) {
if (b == 0) { continue;}
if (a == 1) { do( );} }
b 0 M a 0 S
核心1
Stor_i 1 a
WrtIvld
a 1 M
Sfence
b 1 M
Stor_i 1 b
RdRspDC
b=1所在缓存行
b 1 S
访存共享总线
b I a 0 S
核心2
先回应后干活
a 0 S
IvldRspF
Load b 寄存器A
b 0 F
RdPrb
b 1 S
Cmp 寄存器A 0
Jmpz_b 2
到这都还没作废
Load a 寄存器A
a 0 S
Cmp 寄存器A 1
a I
Jmpz do( )
终于作废但是为时已晚
Jmp_b 6


核心1
a = 1
Barrier( )
b = 1
核心2
while(1) {
if (b == 0) { continue;}
Barrier( )
if (a == 1) { do( );} }
b 0 M a 0 S
核心1
Stor_i 1 a
WrtIvld
a 1 M
Sfence
b 1 M
Stor_i 1 b
RdRspDC
b=1所在缓存行
b 1 S
a 1 S
RdRspDC
a=1所在缓存行
访存共享总线
b I a 0 S
核心2
先回应后干活
a 0 S
IvldRspF
Load b 寄存器A
b 0 F
RdPrb
b 1 S
Cmp 寄存器A 0
Jmpz_b 2
Lfence
改了再往下走
a I
Load a 寄存器A
RdPrb
a 1 S
Cmp 寄存器A 1
Jmpz do( )
符合条件跳到do函数执行
Jmp_b 6

6.9.8.7 蓝色基因/P 中的 Snoop Filter

IBM 在其蓝色基因/P 超级计算机的 CPU 中使用了三种 Snoop Filter 来过滤广播。

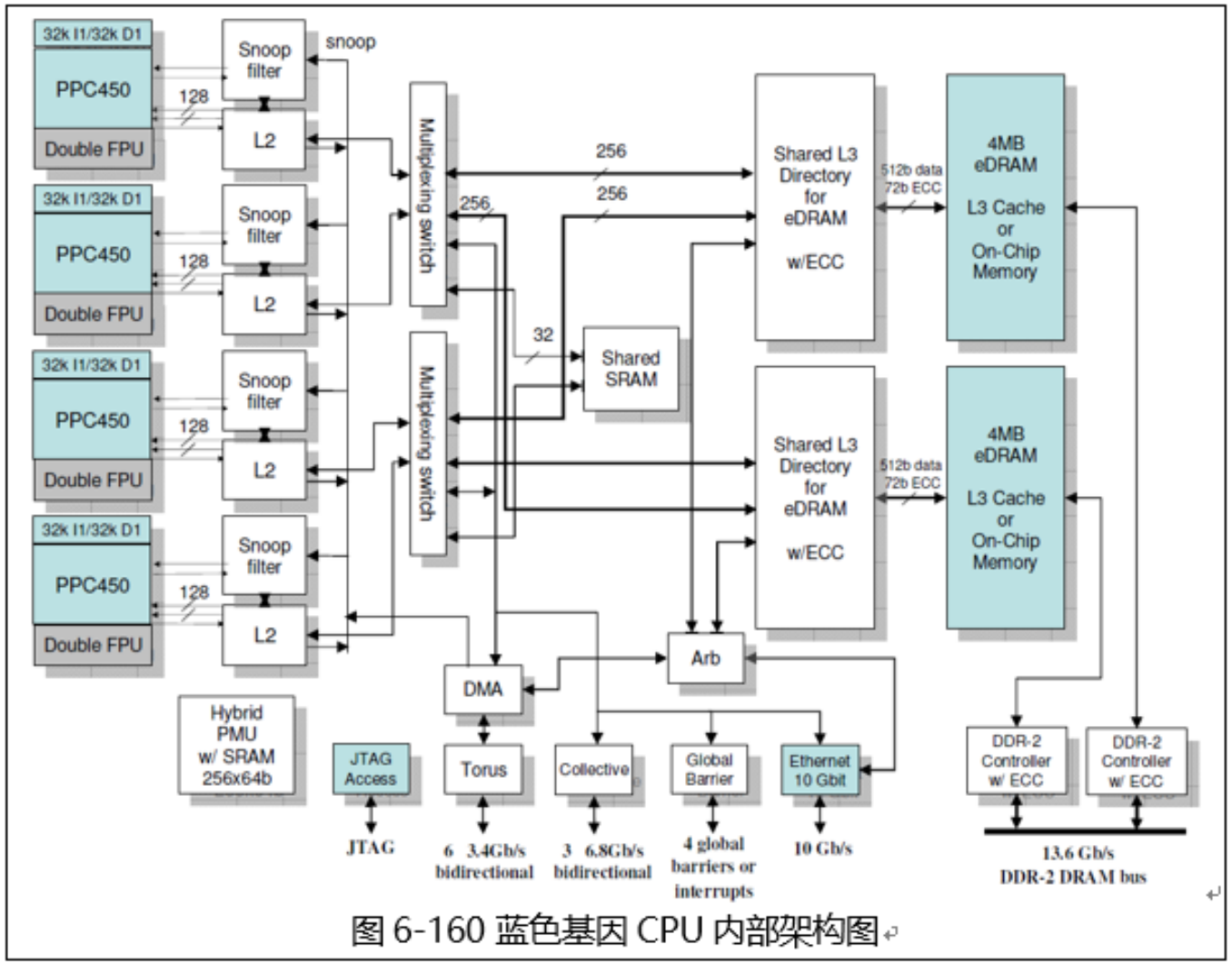


图 6-160 蓝色基因 CPU 内部架构图

如图 6-160 所示为蓝色基因 CPU 内部架构图。其 CPU 的 L1 缓存采用 Write Through 模式透写到 L2 缓存及共享的 L3 缓存，也就是从 L1 直接穿透到 L3，但是 L2 中也会留存一份。所以不需要在 L1 而只需要在最后一层私有缓存处也就是 L2 缓存控制器前端需要过滤装置即可。

如图 6-161 左侧所示，每个 PowerPC 450 CPU 核心前方都放置一个 L2 缓存控制器和一个 Snoop Filter。所有的 L2 缓存控制器与 4 个 Snoop Filter 采用点对点方式广播 CC 请求。

可以看到缓存控制器之间并没有直接通路，所以核心之间无法通过 L2 缓存直接相互转发数据，但是由于 L1 是写透到 L3 的，所以 L3 里始终都是最新的数据，当某个 CPU 收到其他 CPU 的 L2 缓存控制器发送的 Invalidate 消息从而作废了其自己 L2 缓存内某缓存行之后，后续针对该行的读取可直接到 L3 读取，不需要直通转发。

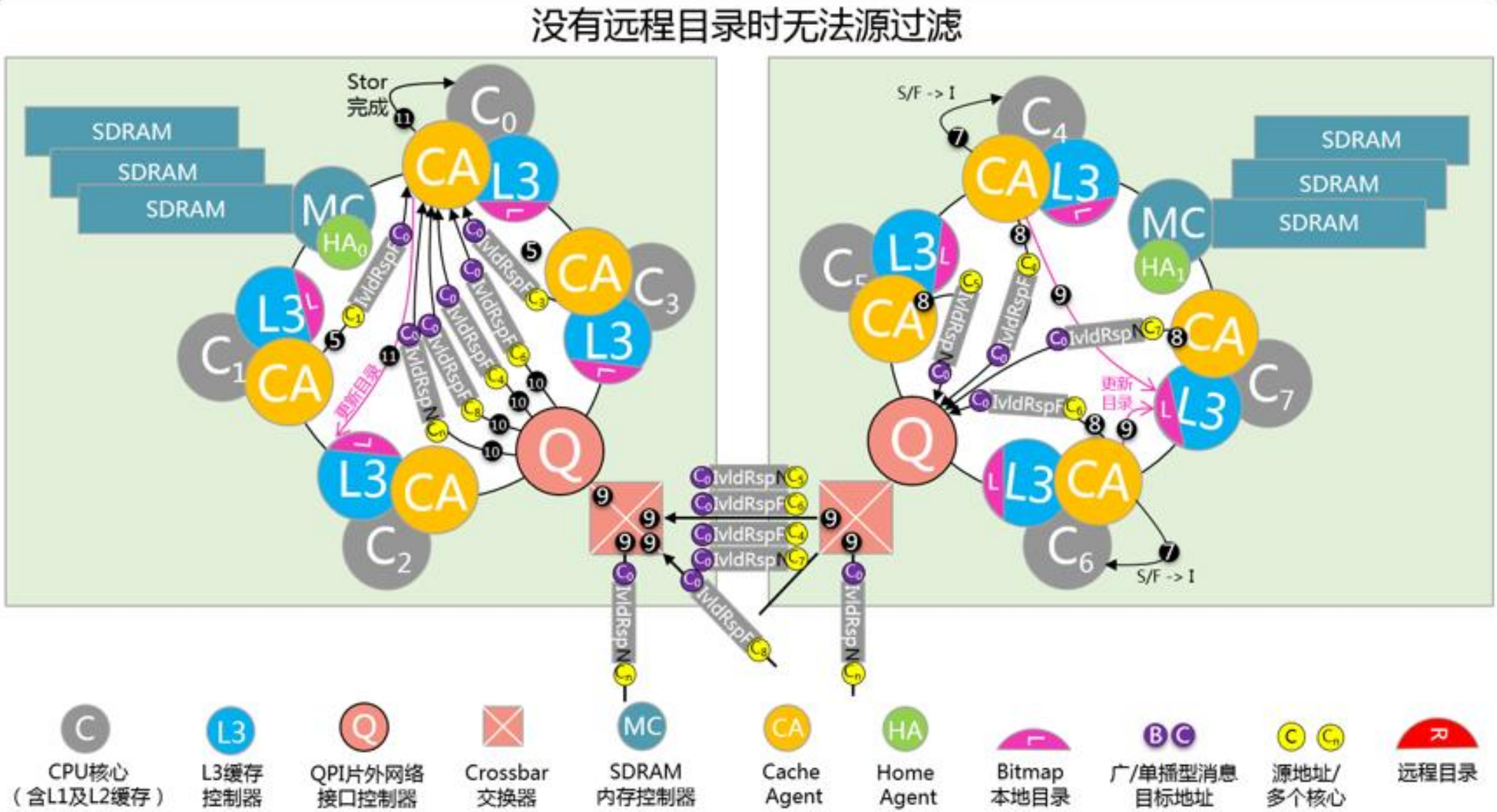
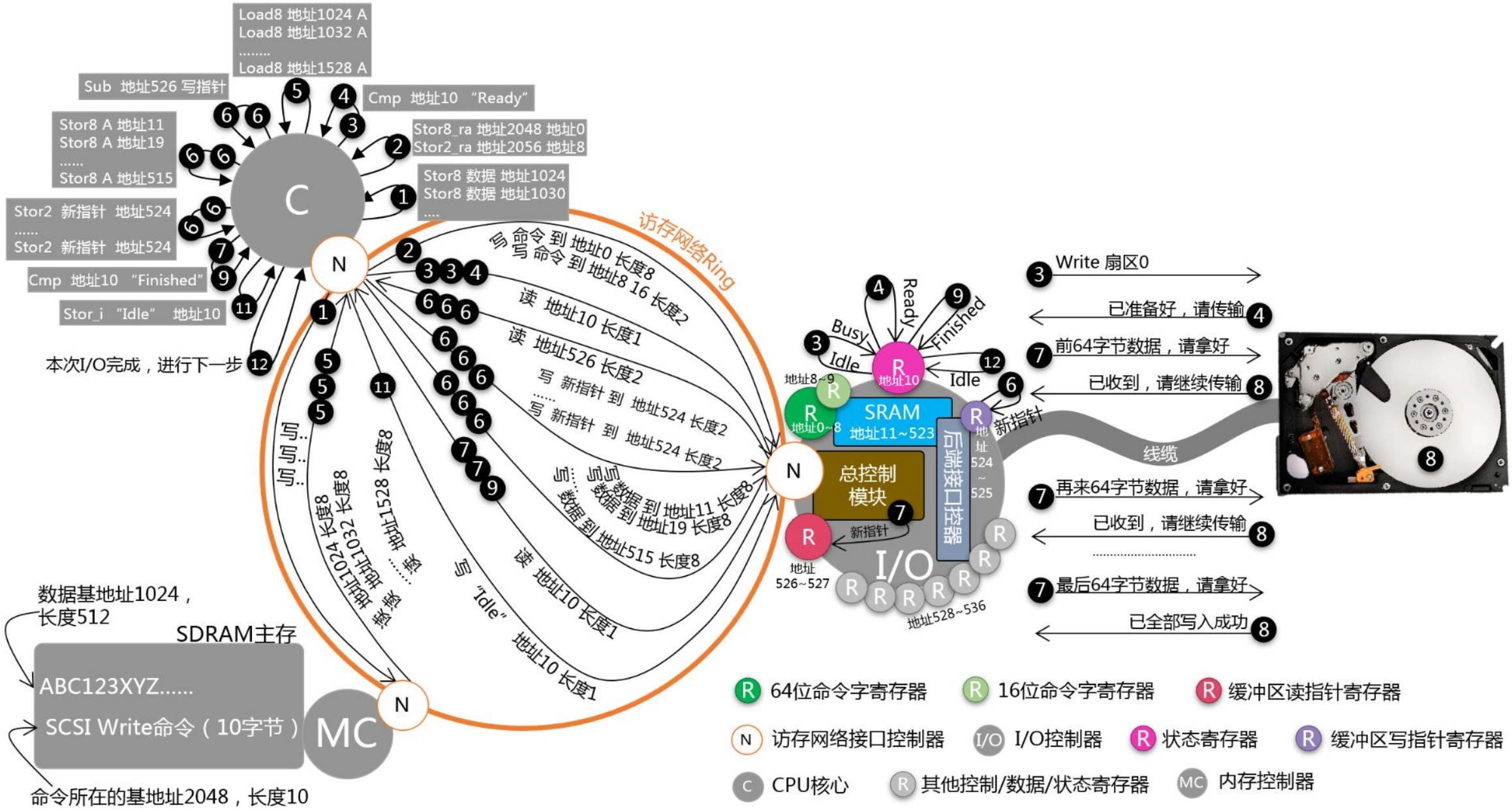


图 6-168 没有远程过滤目录时的 CC 示意图（2）


Load8 地址1024 A
Load8 地址1032 A
.....
Load8 地址1528 A
Sub 地址526 写指针
Cmp 地址10 "Ready"
Stor8 A 地址11
Stor8 A 地址19
.....
Stor8 A 地址515
Stor8_ra 地址2048 地址0
Stor2_ra 地址2056 地址8
Stor8 数据 地址1024
Stor8 数据 地址1030
....
Stor2 新指针 地址524
.....
Stor2 新指针 地址524
Cmp 地址10 "Finished"
Stor_i "Idle" 地址10
本次I/O完成，进行下一步
C
N
访存网络Ring
写 命令 到 地址0 长度8
写 命令 到 地址8 16 长度2
读 地址10 长度1
读 地址526 长度2
写 新指针 到 地址524 长度2
.....
写 新指针 到 地址524 长度2
写 数据 到 地址11 长度8
写 数据 到 地址19 长度8
写 数据 到 地址515 长度8
读 地址10 长度1
写 "Idle" 地址10 长度1
写 地址1528 长度8
写 地址1032 长度8
写 地址1024 长度8
写..
写..
写..
数据基地址1024，
长度512
SDRAM主存
ABC123XYZ.....
SCSI Write命令（10字节）
命令所在的基地址2048，长度10
MC
Busy
Ready
Finished
Idle
Idle
地址8~9
地址0~8
地址10
SRAM
地址11~523
总控制
模块
后端接口控器
新指针
新指针
地址
524
~
525
地址
526~527
I/O
地址528~536
线缆
Write 扇区0
已准备好，请传输
前64字节数据，请拿好
已收到，请继续传输
再来64字节数据，请拿好
已收到，请继续传输
.....
最后64字节数据，请拿好
已全部写入成功
64位命令字寄存器
16位命令字寄存器
缓冲区读指针寄存器
访存网络接口控制器
I/O控制器
状态寄存器
缓冲区写指针寄存器
CPU核心
其他控制/数据/状态寄存器
内存控制器

上角部分中的紫色立方体），然后将该数据包的元数据写回到接收队列中的对应条目的对应位置（图中绿色区块）。然后，网络 I/O 控制器发出中断信号，Host 端程序对该数据包进行后续处理，并更新队尾指针。至于“后续处理”都处理的什么东西，怎么个流程，我们会在后文中的网络协议栈相关章节详细介绍。

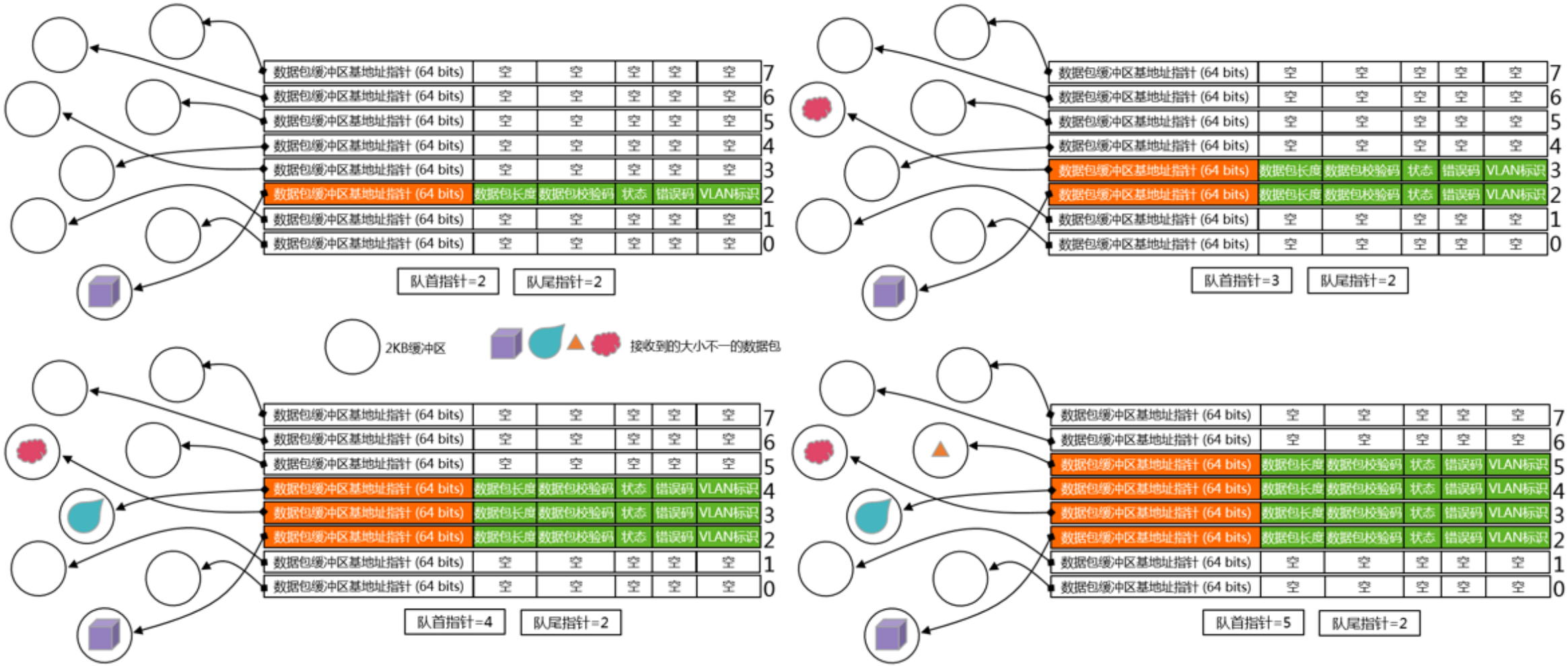
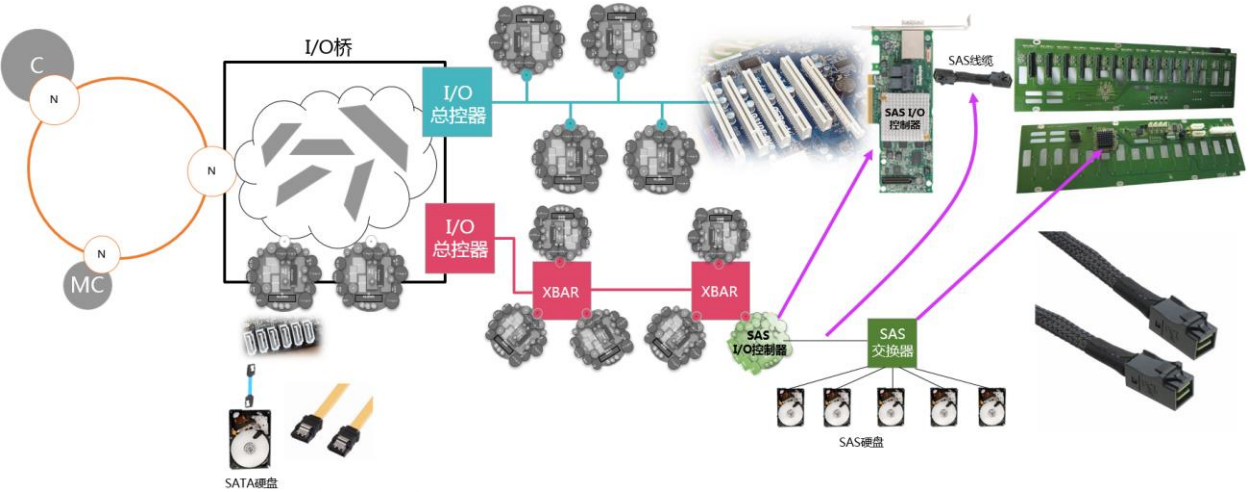
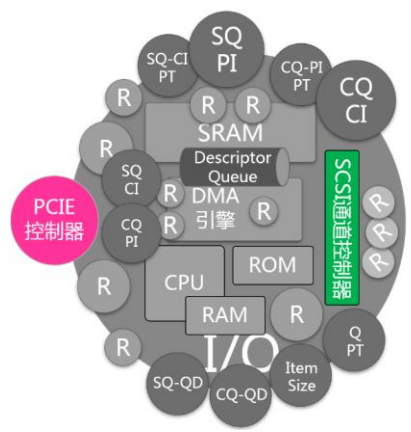
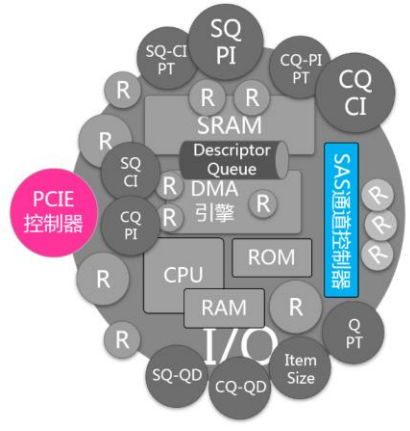


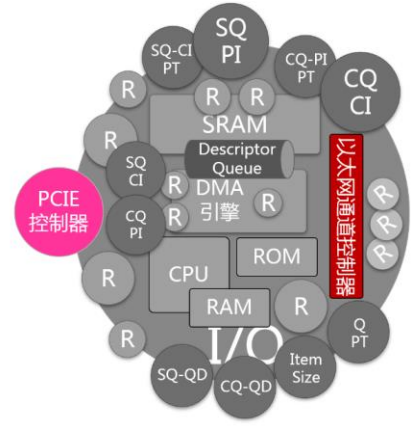
图 7-12 网络 I/O 控制器接受数据包并写入主存中缓冲区的过程

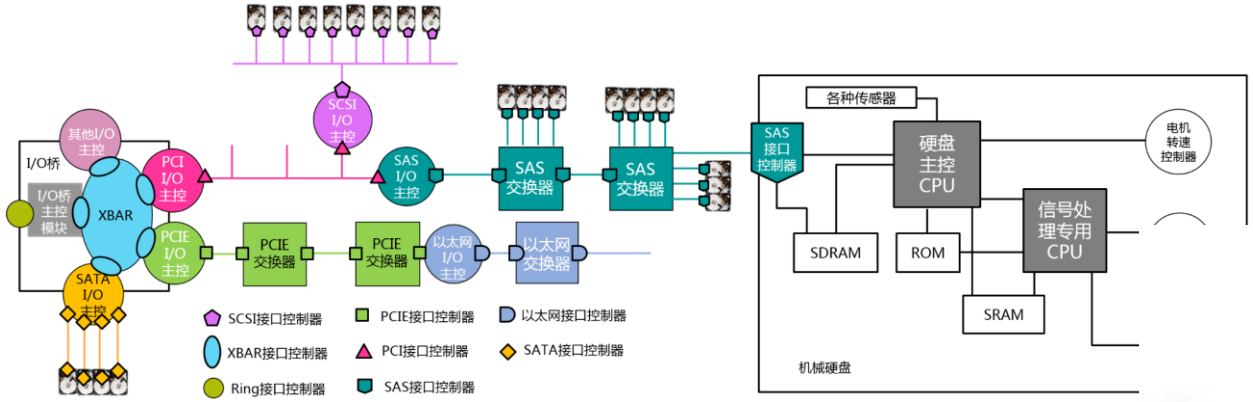
我们假设 Host 端程序还没有来得及处理接收到的数据包，结果网络控制器又接收到了多个数据包。如上图中右上角、左下角、右下角所示。此时接收队列中一共积压了 4 个数据包。


I/O桥
C
N
N
N
MC
I/O
总控器
I/O
总控器
XBAR
XBAR
SAS
I/O控制器
SAS
交换器
SAS I/O
控制器
SAS线缆
SAS硬盘
SATA硬盘

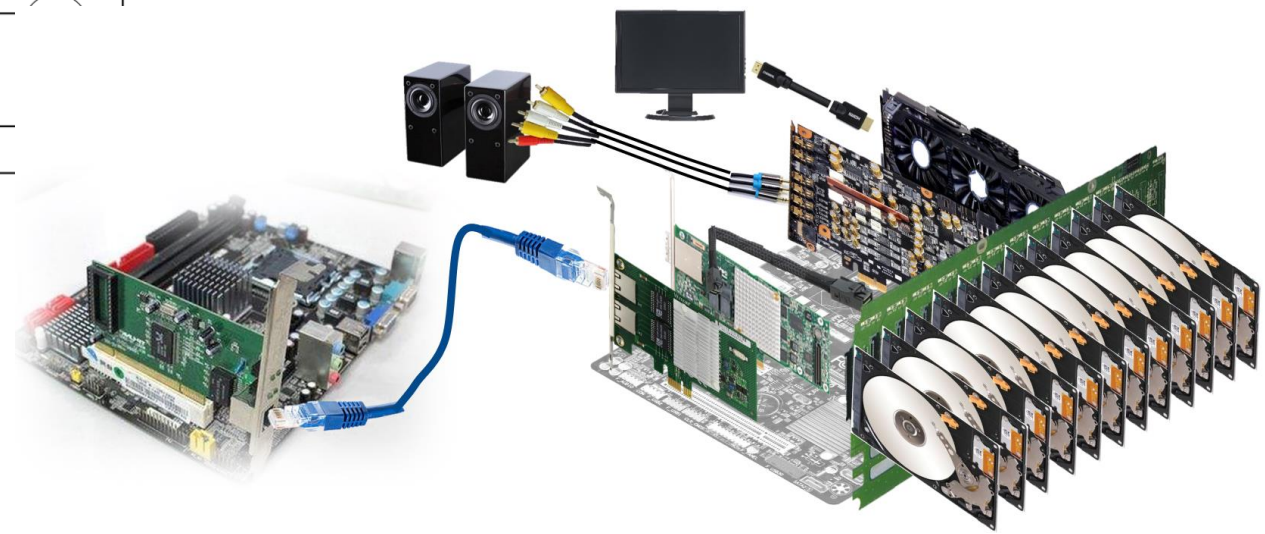

SQ
PI
SQ-CI
PT
CQ-PI
PT
CQ
CI
R
SRAM
Descriptor
Queue
SQ
CI
DMA
引擎
CQ
PI
SCSI通道控制器
PCIE
控制器
CPU
ROM
RAM
I/O
Q
PT
SQ-QD
CQ-QD
Item
Size

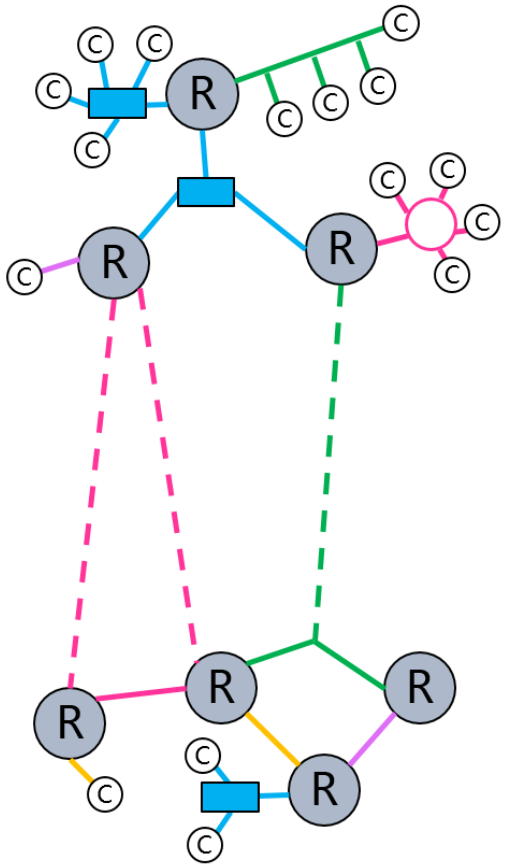

SQ
PI
SQ-CI
PT
CQ-PI
PT
CQ
CI
R
SRAM
Descriptor
Queue
SQ
CI
DMA
引擎
CQ
PI
SAS通道控制器
PCIE
控制器
CPU
ROM
RAM
I/O
Q
PT
SQ-QD
CQ-QD
Item
Size

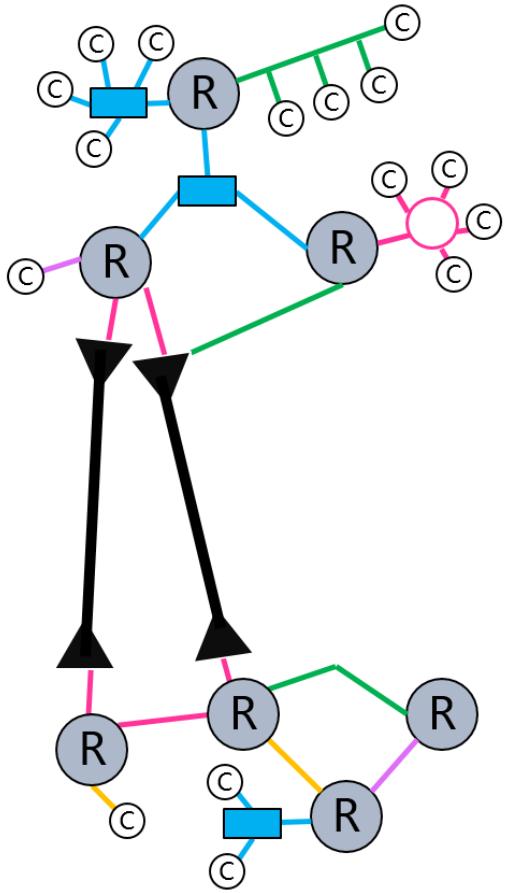

SQ
PI
SQ-CI
PT
CQ-PI
PT
CQ
CI
R
SRAM
Descriptor
Queue
SQ
CI
DMA
引擎
CQ
PI
以太网通道控制器
PCIE
控制器
CPU
ROM
RAM
I/O
Q
PT
SQ-QD
CQ-QD
Item
Size

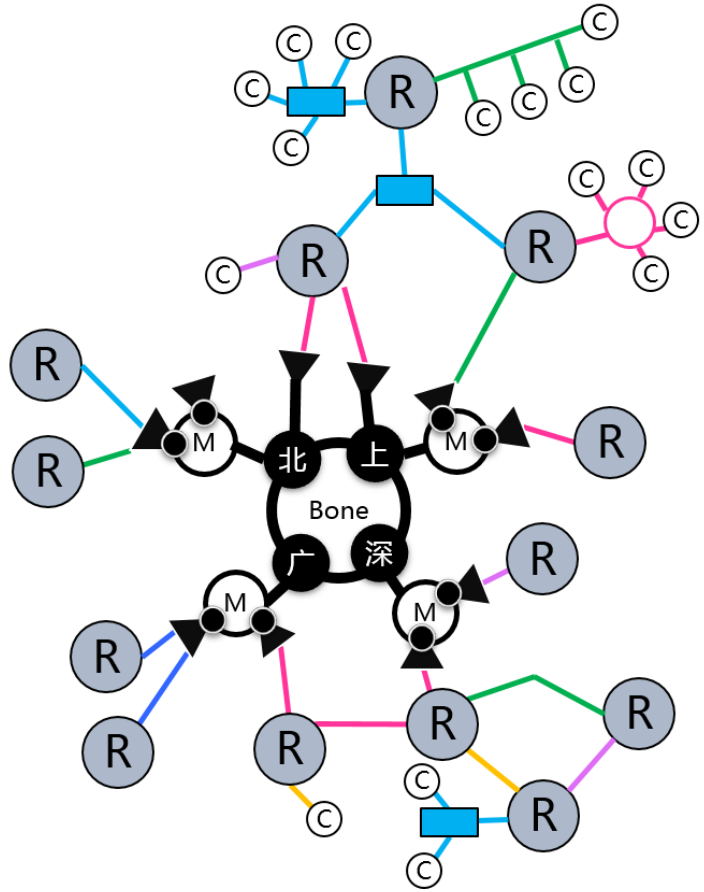

I/O桥
其他I/O
主控
PCI
I/O
主控
I/O桥
主控
模块
XBAR
PCIE
I/O
主控
SATA
I/O
主控
SCSI
I/O
主控
SAS
I/O
主控
SAS
交换器
SAS
交换器
PCIE
交换器
PCIE
交换器
以太网
I/O
主控
以太网
交换器
SCSI接口控制器
XBAR接口控制器
Ring接口控制器
PCIE接口控制器
PCI接口控制器
SAS接口控制器
以太网接口控制器
SATA接口控制器

各种传感器
SAS
接口
控制器
硬盘
主控
CPU
电机
转速
控制器
信号处
理专用
CPU
SDRAM
ROM
SRAM
机械硬盘









接入网传输设备

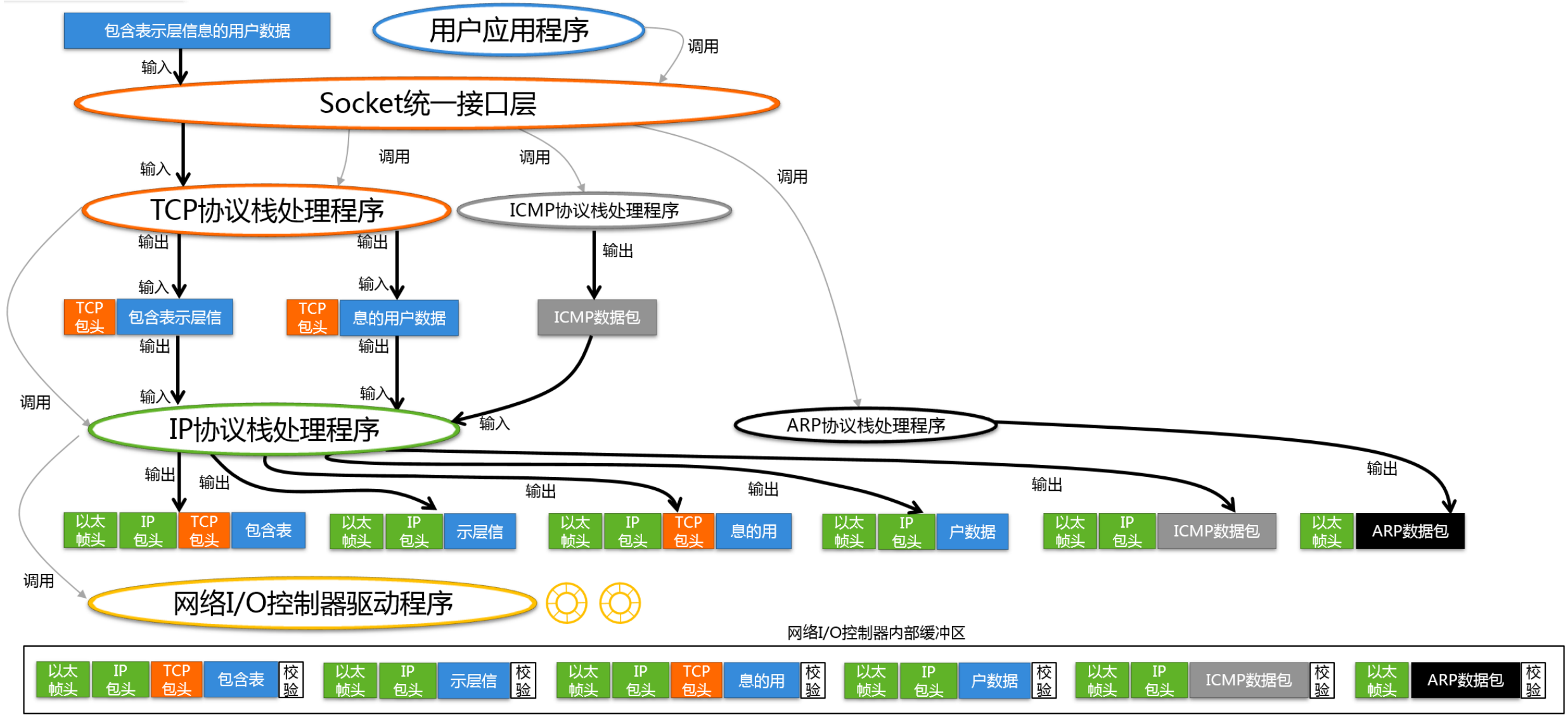
城域网

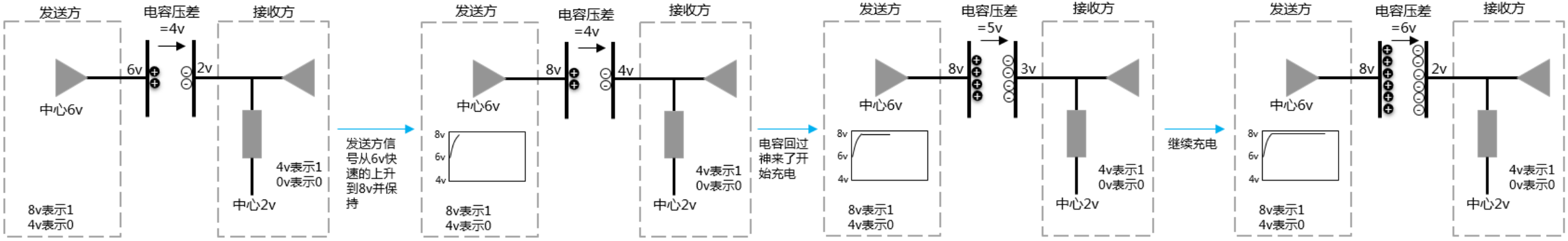
城域网传输设备

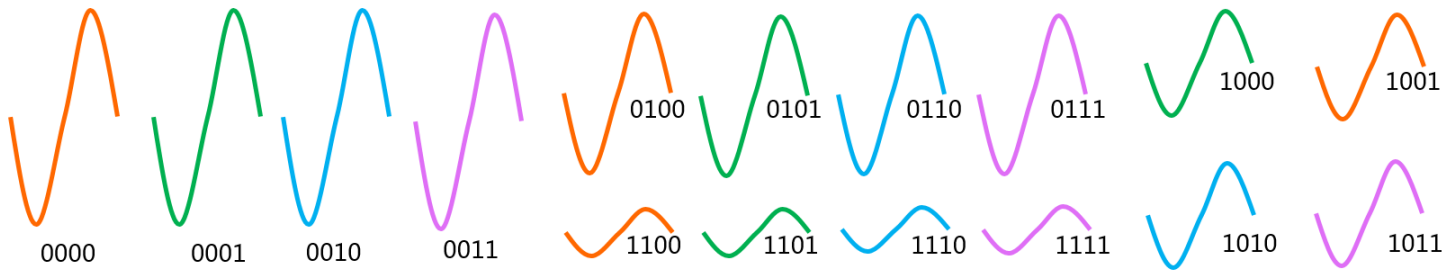
骨干网传输设备

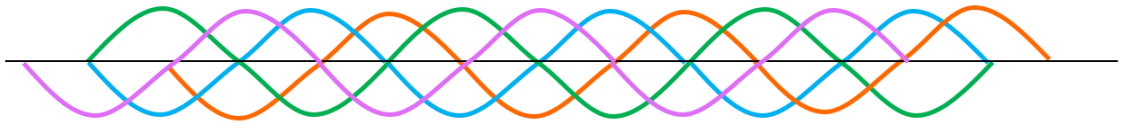
骨干网

圆形部分整体称为传输网

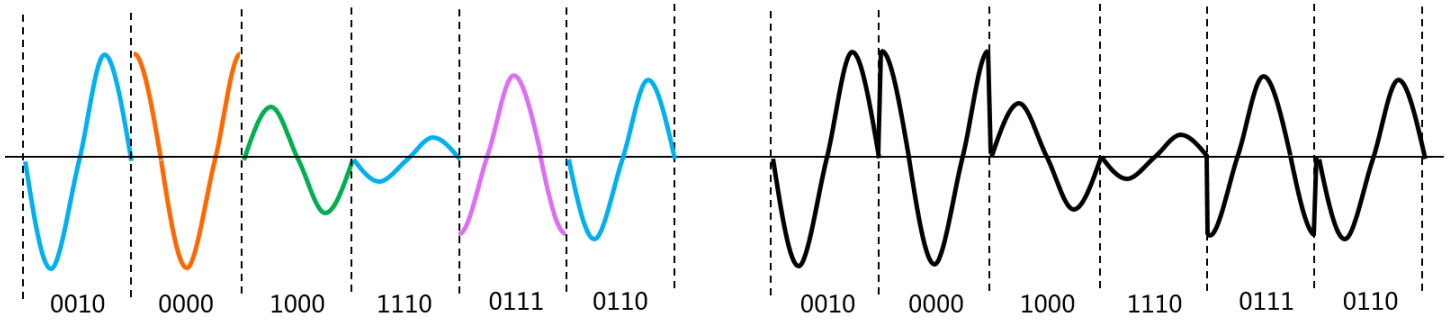

包含表示层信息的用户数据
用户应用程序
调用
输入
Socket统一接口层
输入
调用
调用
调用
TCP协议栈处理程序
ICMP协议栈处理程序
输出
输出
输出
输入
输入
TCP 包头
包含表示层信
TCP 包头
息的用户数据
ICMP数据包
输出
输出
输入
输入
调用
IP协议栈处理程序
输入
ARP协议栈处理程序
输出
输出
输出
输出
输出
输出
以太帧头
IP包头
TCP包头
包含表
以太帧头
IP包头
示层信
以太帧头
IP包头
TCP包头
息的用
以太帧头
IP包头
户数据
以太帧头
IP包头
ICMP数据包
以太帧头
ARP数据包
调用
网络I/O控制器驱动程序
网络I/O控制器内部缓冲区
以太帧头
IP包头
TCP包头
包含表
校验
以太帧头
IP包头
示层信
校验
以太帧头
IP包头
TCP包头
息的用
校验
以太帧头
IP包头
户数据
校验
以太帧头
IP包头
ICMP数据包
校验
以太帧头
ARP数据包
校验

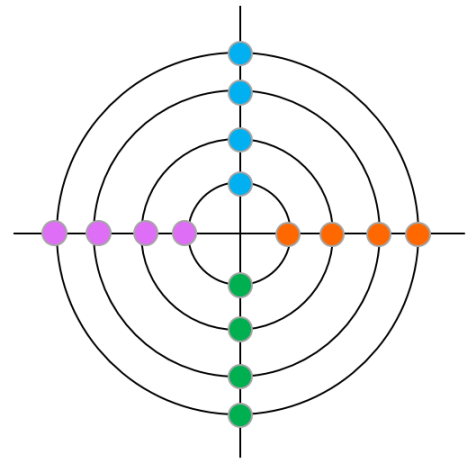


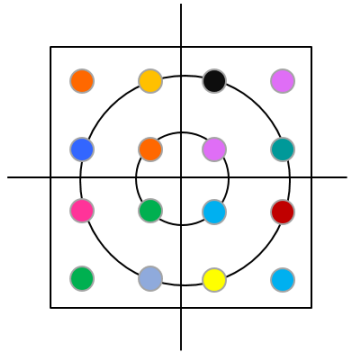


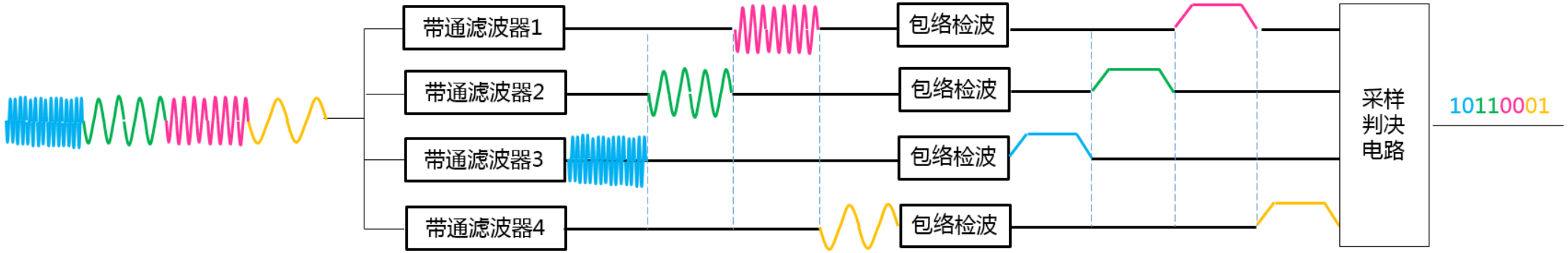


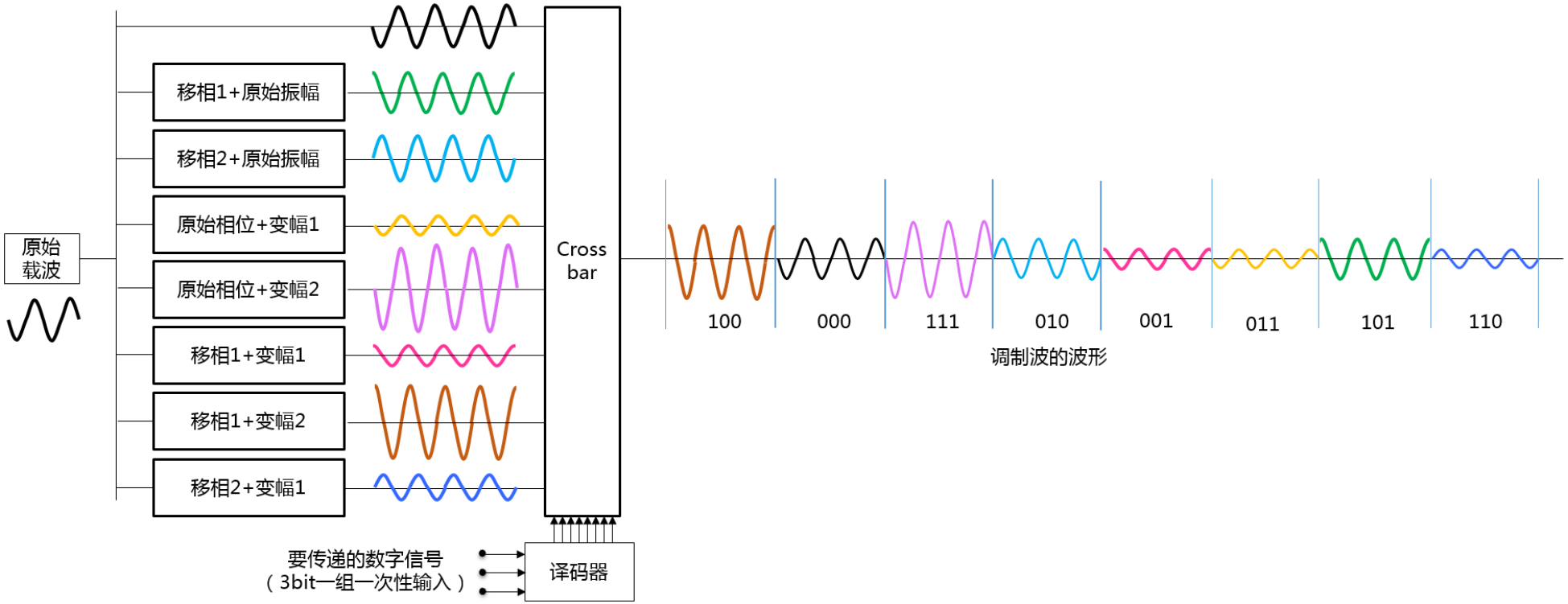
人为定义：
蓝色落后橙色90度相位；
紫色落后橙色180度相位；
绿色落后橙色270度相位；

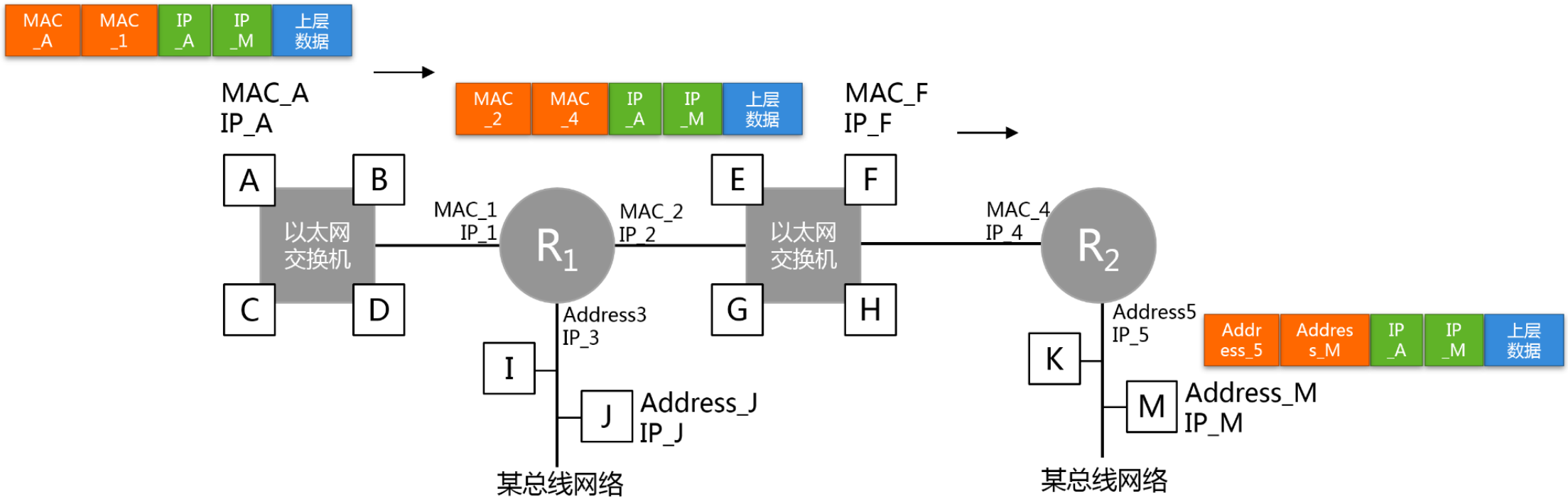


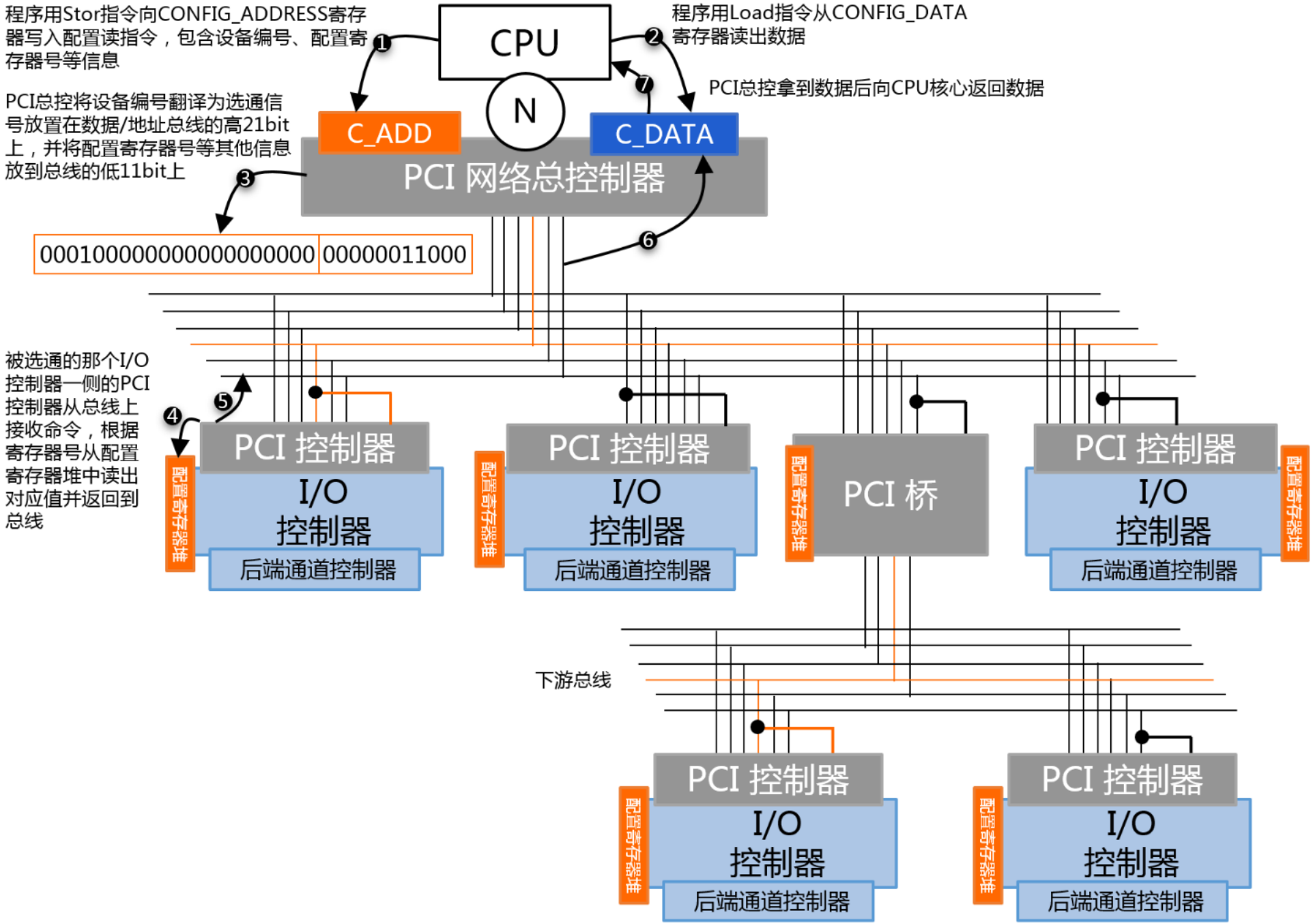


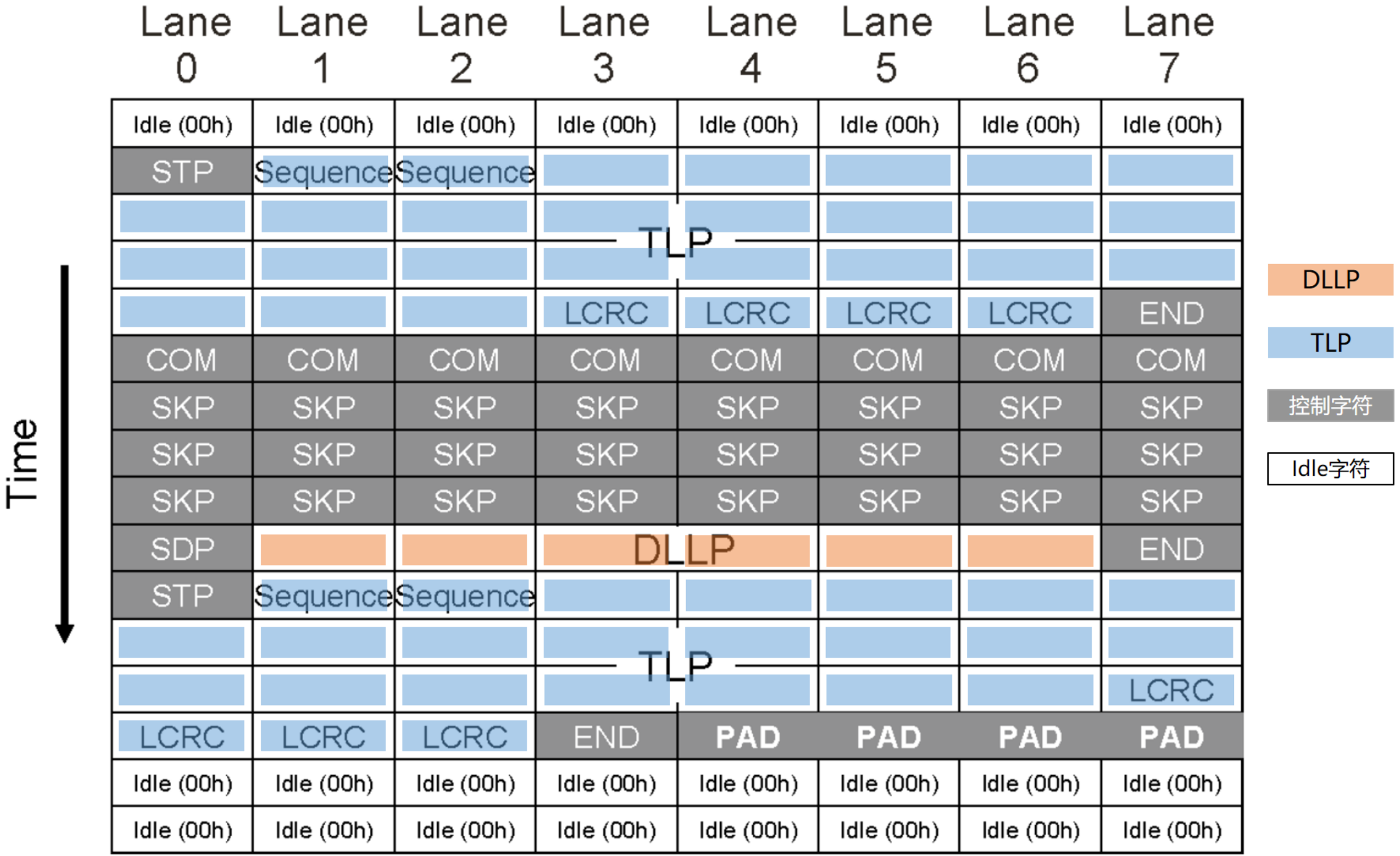


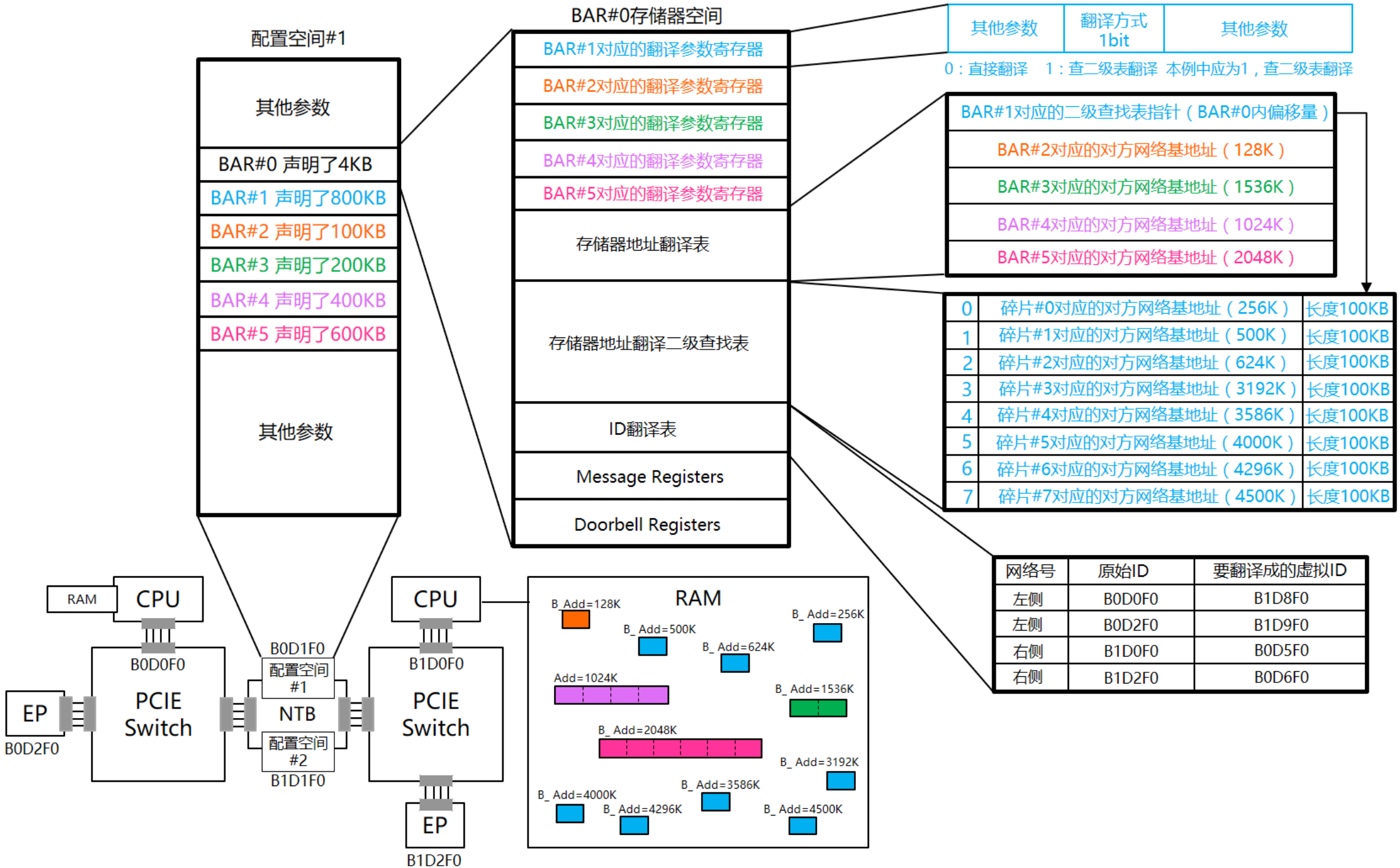

带通滤波器1
带通滤波器2
带通滤波器3
带通滤波器4
包络检波
包络检波
包络检波
包络检波
采样
判决
电路
10110001

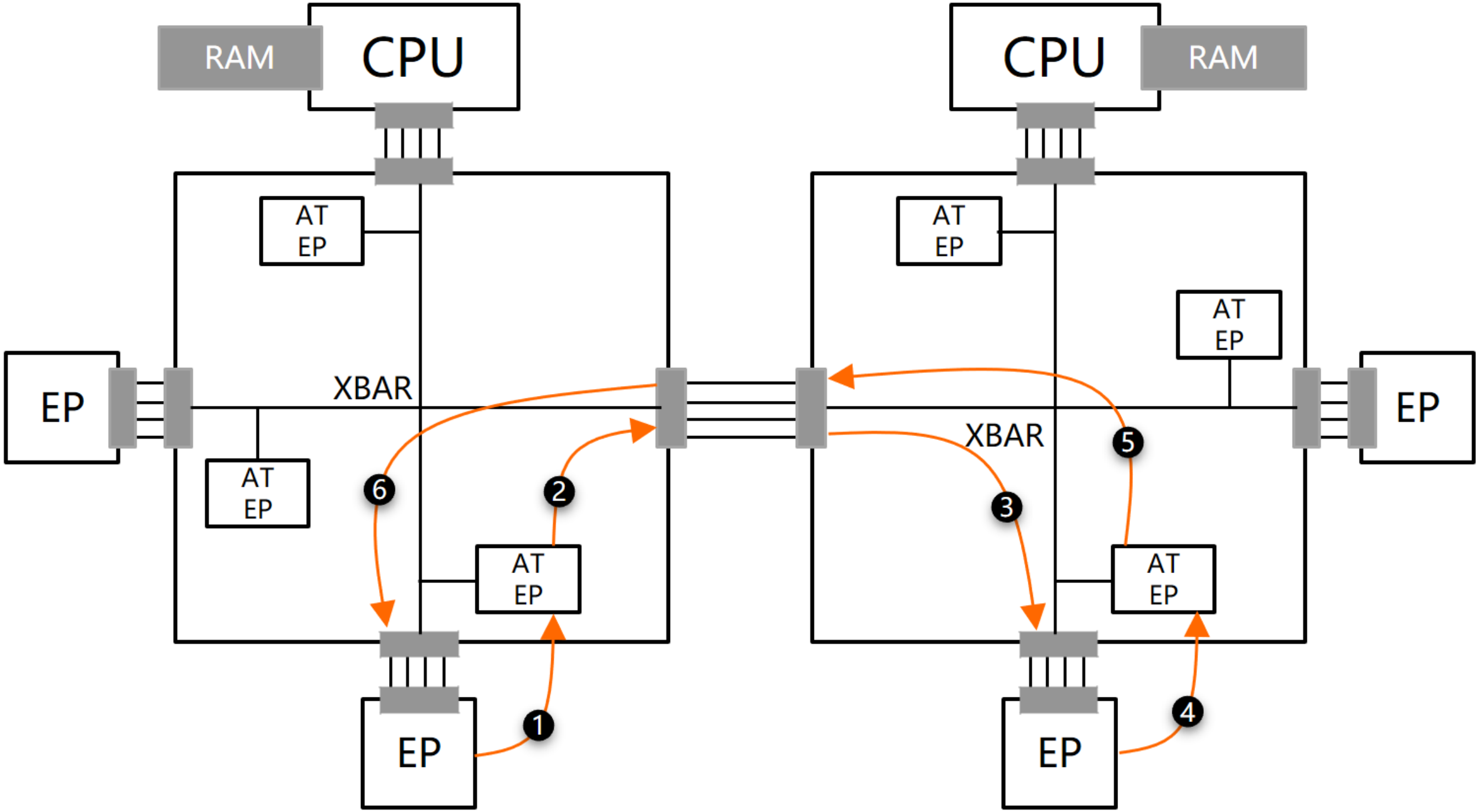

原始
载波
移相1+原始振幅
移相2+原始振幅
原始相位+变幅1
原始相位+变幅2
移相1+变幅1
移相1+变幅2
移相2+变幅1
Cross
bar
要传递的数字信号
（3bit一组一次性输入）
译码器
100
000
111
010
001
011
101
110
调制波的波形

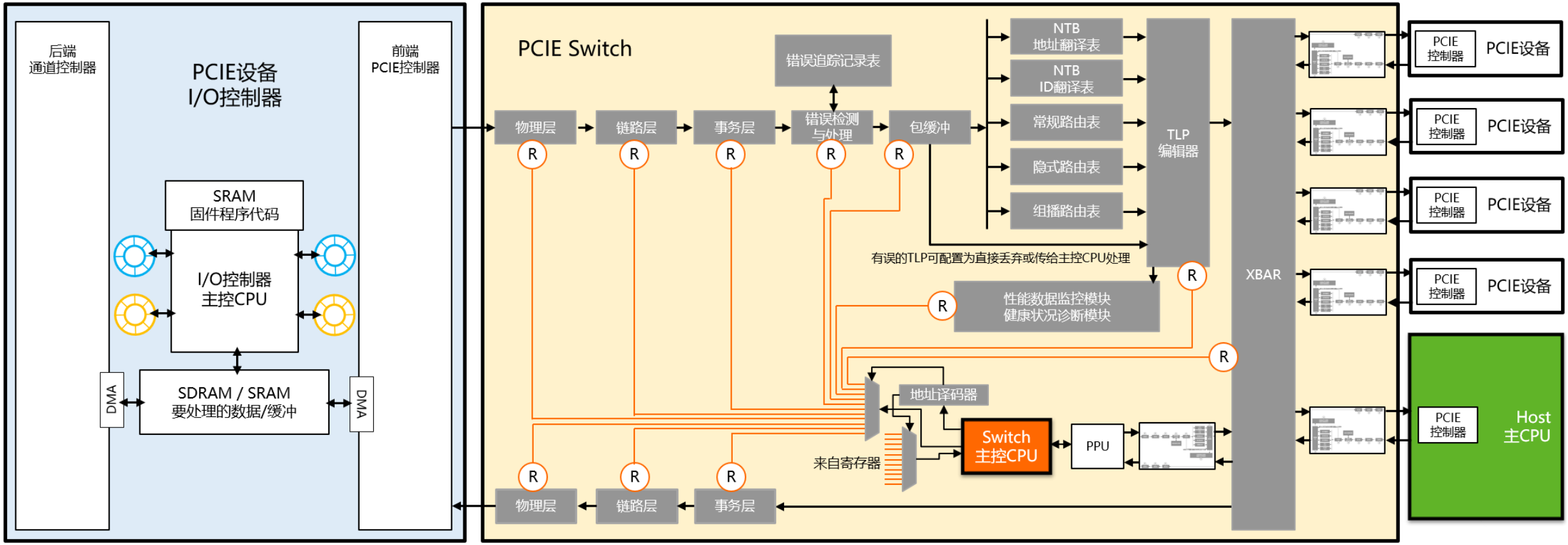

MAC _A
MAC _1
IP _A
IP _M
上层 数据
MAC_A
IP_A
MAC _2
MAC _4
IP _A
IP _M
上层 数据
MAC_F
IP_F
A
B
C
D
以太网 交换机
MAC_1
IP_1
R1
MAC_2
IP_2
E
F
G
H
以太网 交换机
MAC_4
IP_4
R2
Address3
IP_3
I
J
Address_J
IP_J
某总线网络
Address5
IP_5
K
M
Address_M
IP_M
某总线网络
Addr ess_5
Addres s_M
IP _A
IP _M
上层 数据

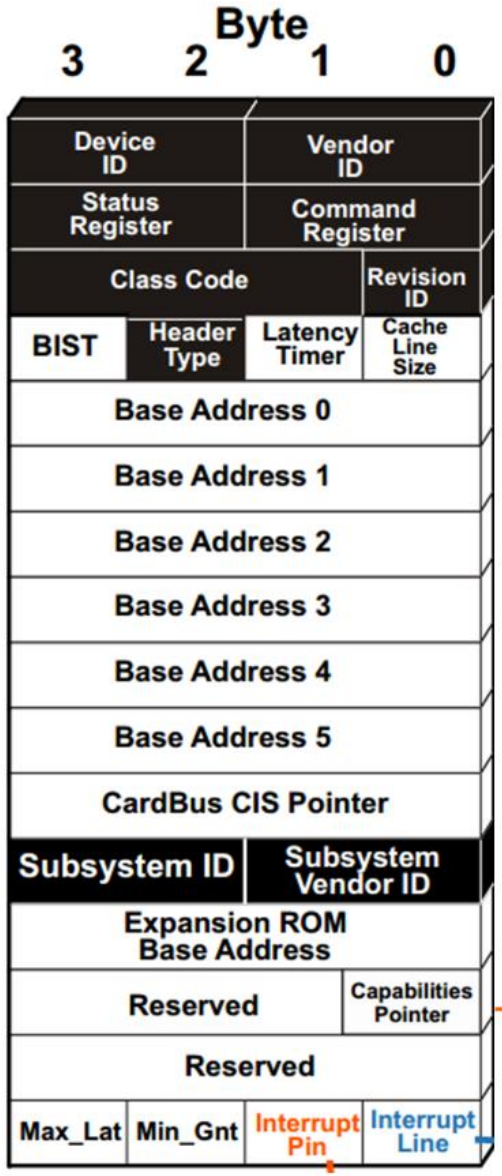

程序用Stor指令向CONFIG_ADDRESS寄存器写入配置读指令，包含设备编号、配置寄存器号等信息
CPU
程序用Load指令从CONFIG_DATA寄存器读出数据
PCI总控拿到数据后向CPU核心返回数据
PCI总控将设备编号翻译为选通信号放置在数据/地址总线的高21bit上，并将配置寄存器号等其他信息放到总线的低11bit上
N
C_ADD
C_DATA
PCI 网络总控制器
00010000000000000000
00000011000
被选通的那个I/O控制器一侧的PCI控制器从总线上接收命令，根据寄存器号从配置寄存器堆中读出对应值并返回到总线
PCI 控制器
I/O 控制器
后端通道控制器
配置寄存器堆
PCI 桥
下游总线


Lane 0
Lane 1
Lane 2
Lane 3
Lane 4
Lane 5
Lane 6
Lane 7
Idle (00h)
STP
Sequence
TLP
LCRC
END
COM
SKP
SDP
DLLP
PAD
Time
DLLP
TLP
控制字符
Idle字符

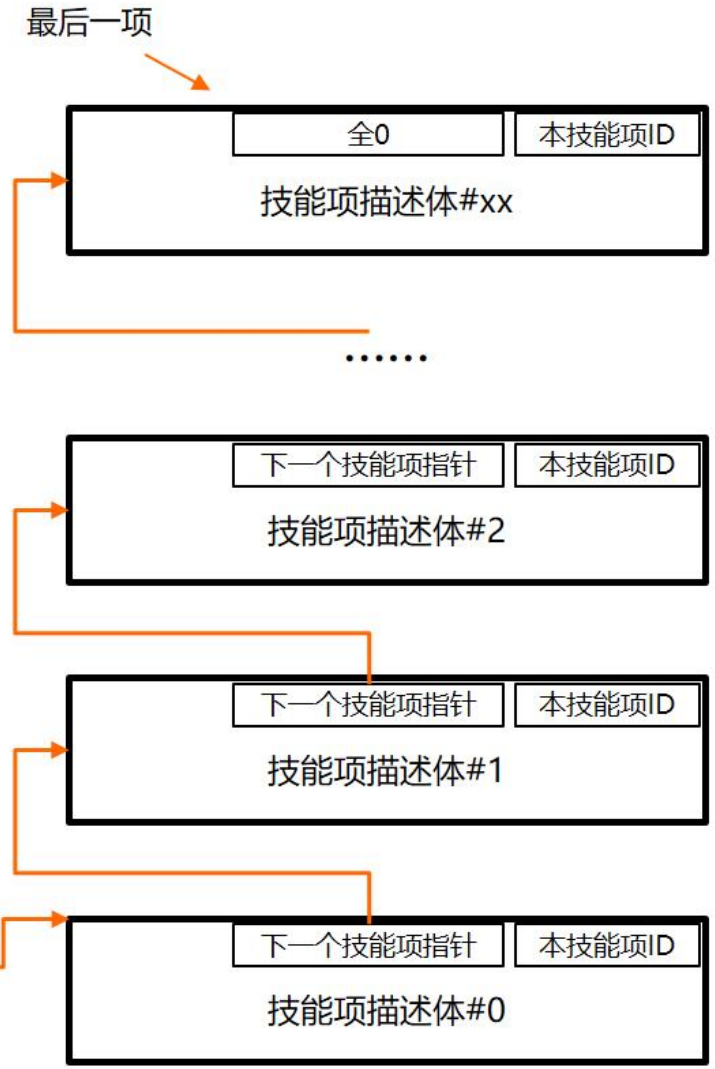

配置空间#1
其他参数
BAR#0 声明了4KB
BAR#1 声明了800KB
BAR#2 声明了100KB
BAR#3 声明了200KB
BAR#4 声明了400KB
BAR#5 声明了600KB
其他参数
BAR#0存储器空间
BAR#1对应的翻译参数寄存器
BAR#2对应的翻译参数寄存器
BAR#3对应的翻译参数寄存器
BAR#4对应的翻译参数寄存器
BAR#5对应的翻译参数寄存器
存储器地址翻译表
存储器地址翻译二级查找表
ID翻译表
Message Registers
Doorbell Registers
其他参数
翻译方式 1bit
其他参数
0：直接翻译 1：查二级表翻译 本例中应为1，查二级表翻译
BAR#1对应的二级查找表指针（BAR#0内偏移量）
BAR#2对应的对方网络基地址（128K）
BAR#3对应的对方网络基地址（1536K）
BAR#4对应的对方网络基地址（1024K）
BAR#5对应的对方网络基地址（2048K）
0 碎片#0对应的对方网络基地址（256K） 长度100KB
1 碎片#1对应的对方网络基地址（500K） 长度100KB
2 碎片#2对应的对方网络基地址（624K） 长度100KB
3 碎片#3对应的对方网络基地址（3192K） 长度100KB
4 碎片#4对应的对方网络基地址（3586K） 长度100KB
5 碎片#5对应的对方网络基地址（4000K） 长度100KB
6 碎片#6对应的对方网络基地址（4296K） 长度100KB
7 碎片#7对应的对方网络基地址（4500K） 长度100KB
网络号 原始ID 要翻译成的虚拟ID
左侧 B0D0F0 B1D8F0
左侧 B0D2F0 B1D9F0
右侧 B1D0F0 B0D5F0
右侧 B1D2F0 B0D6F0
RAM
CPU
B0D0F0
EP
B0D2F0
PCIE Switch
B0D1F0
配置空间 #1
NTB
配置空间 #2
B1D1F0
CPU
B1D0F0
PCIE Switch
EP
B1D2F0
RAM
B_ Add=128K
B_ Add=500K
B_ Add=624K
B_ Add=256K
Add=1024K
B_ Add=1536K
B_ Add=2048K
B_ Add=3192K
B_ Add=3586K
B_ Add=4000K
B_ Add=4296K
B_ Add=4500K


RAM
CPU
CPU
RAM
AT
EP
AT
EP
AT
EP
EP
XBAR
XBAR
EP
AT
EP
AT
EP
AT
EP
1
2
3
4
5
6
EP
EP


后端
通道控制器
PCIE设备
I/O控制器
前端
PCIE控制器
SRAM
固件程序代码
I/O控制器
主控CPU
DMA
SDRAM / SRAM
要处理的数据/缓冲
DMA
PCIE Switch
错误追踪记录表
物理层
链路层
事务层
错误检测
与处理
包缓冲
NTB
地址翻译表
NTB
ID翻译表
常规路由表
隐式路由表
组播路由表
TLP
编辑器
有误的TLP可配置为直接丢弃或传给主控CPU处理
性能数据监控模块
健康状况诊断模块
XBAR
R
地址译码器
来自寄存器
Switch
主控CPU
PPU
PCIE
控制器
PCIE设备
Host
主CPU



64字节的标准配置空间



最大4K字节的扩展配置空间

8.1.1 让蜂鸣器说话

后来，个人电脑上逐渐使用了频响范围较宽的蜂鸣器，这使得其发声时可以带有更多谐振频率，让音质变得相对更柔润了一些。这样就可以驱动蜂鸣器发出任意单一频率或者任意频率叠加之后的杂波了，意味着，其可以发出人声！

图 8-1 蜂鸣器、宽频响喇叭以及第一个利用其发出中文语音的游戏

第一个用 PC 蜂鸣器发出中文语音的游戏程序，是 1991 年智冠科技开发的《三国演义》，张飞：“匹夫，出来与我决一死战！”，吕布：“汝何人也，不配与吾交手！”。在这之前的一两年内，国外的一些游戏已经开始可以用 PC 喇叭发出语音了。要知道在 25 年前，多数 PC 电脑用户是根本没体验过电脑发出人声的，那时候能够看到电脑屏幕上出现图形，而不是一串串的字符，就已经兴奋满足不已了，何况能听到声音？不过，在当时已经出现了独立声卡，只不过非常昂贵，一般只有发烧友级别的消费者才会去购买，所以能用任何 PC 都有的蜂鸣器发出声音便是当时让人难忘的记忆。用 PC 蜂鸣器发出人声，是前人们使用非常聪明的做法完成的。

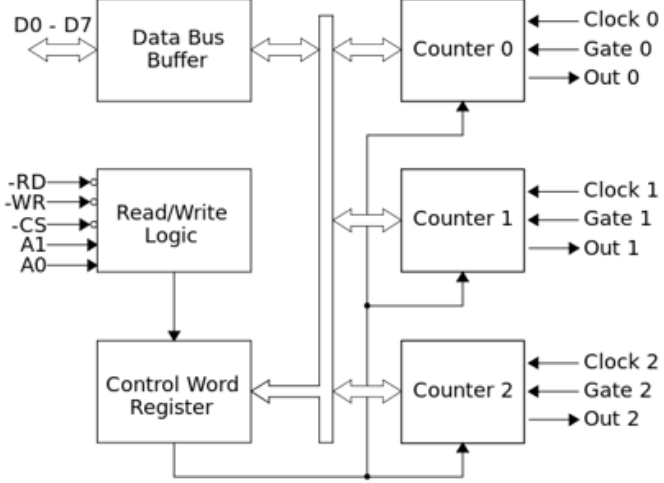
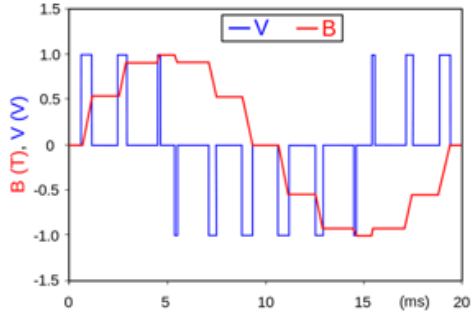


图 8-2 Intel 8235 时钟发生器架构图

当时 IBM 的 PC 兼容机广为流行，其中使用了 Intel 生产的时钟发生器，比如 Intel 8253/8254 型时钟发生器。该发生器的架构如图 8-2 所示。其中内含三个计数器，程序可以将一个数值写入到计数器中，然后计数器就开始自动倒计时，当数值降低到 1 之后，便会将对应的 Out#信号置为高电平。利用这个时钟发生器，可以定时产生中断，从而让系统记录时间以及实现其他功能。其可以被配置为多种模式，程序可以向 Control Word Register 中写入对应的控制字来切换其实现各种模式。这些寄存器中的信号会控制下游电路中的逻辑从而切换到对应模式。比如其中一种模式就是，程序向 Counter2 中写入一个初始值，倒计时结束后电路会向 Out2 发出一个高电平，然后回到低电平，然后继续载入（电路自动保存和载入）使用刚才的初始值继续倒计时，继续向 Out2 发出高电平，这样便会在 Out2 上产生一个恒定周期的方波脉冲，周期的大小可以通过 Counter2 的初始值来给定。

那么，这个时钟发生器与 PC 蜂鸣器又有何关系呢？原来，当时 IBM PC 普遍被设计为使用时钟发生器的 Counter2 的 Out2 输出端来驱动蜂鸣器发声，比如遇到了内存没有插，需要间隔 0.2ms 发出 3 声嘀声，每一声持续发声 0.5ms。这个过程怎么做到？根据上述原理，可以先向 Counter1 中写入一个初始值，只要按照该时钟发生器的运行频率，精确计算出经过 0.2ms 时间该计数器会变化多少次，那就写入对应的数值进去，到了时间自然就会触发一次高电平，该高电平会触发一次中断，中断服务程序运行之后，可以向 Counter1 再写入一个能够持续计数 0.7ms 的值，让 Counter1 开始倒计时，然后立即向 Counter2 中写入一个能够持续极短时间的值，比如持续 0.001ms 的值，这样，Counter2 每隔 0.001ms 便会发出一个高电平，让蜂鸣器振动一下，在 0.5ms 内，蜂鸣器会振动 500 次，那么其振动频率也就是 1KHz，这就能够让人耳感受到声响了，所以人耳听到的就是一声持续 0.5ms 的长鸣。然后 Counter1 会发出中断，程序可以继续这个过程。



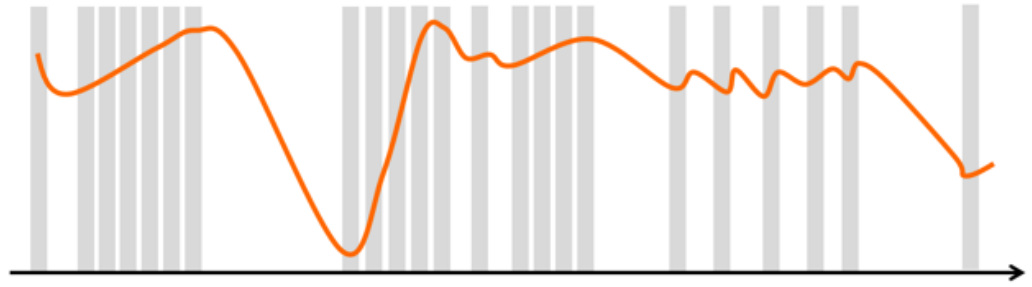
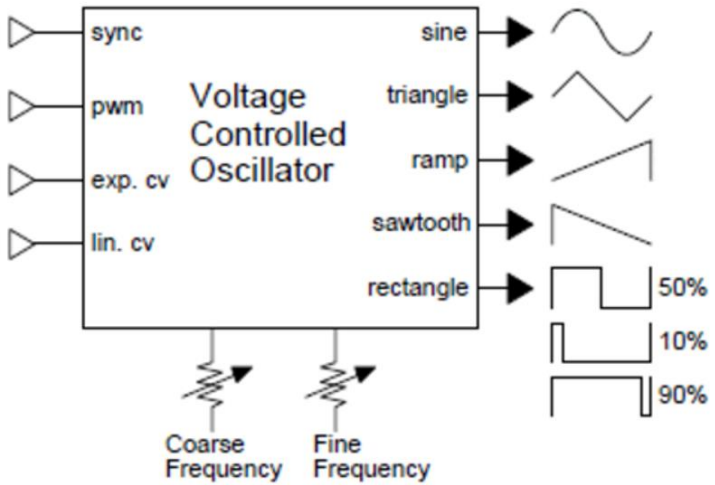


图 8-3 PWM 脉冲宽度调制示意图

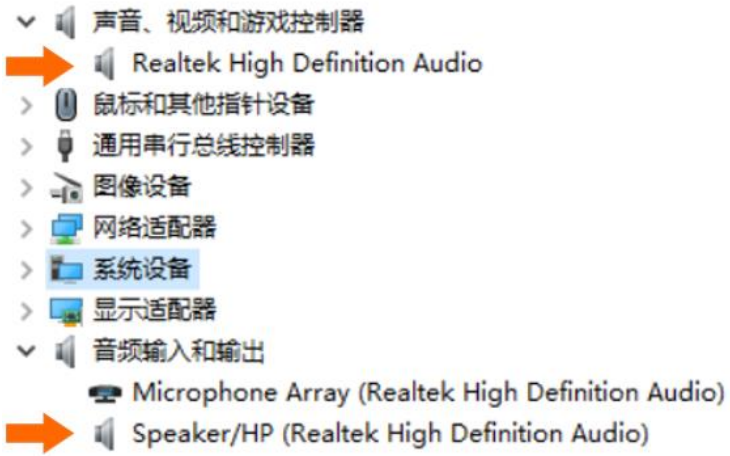


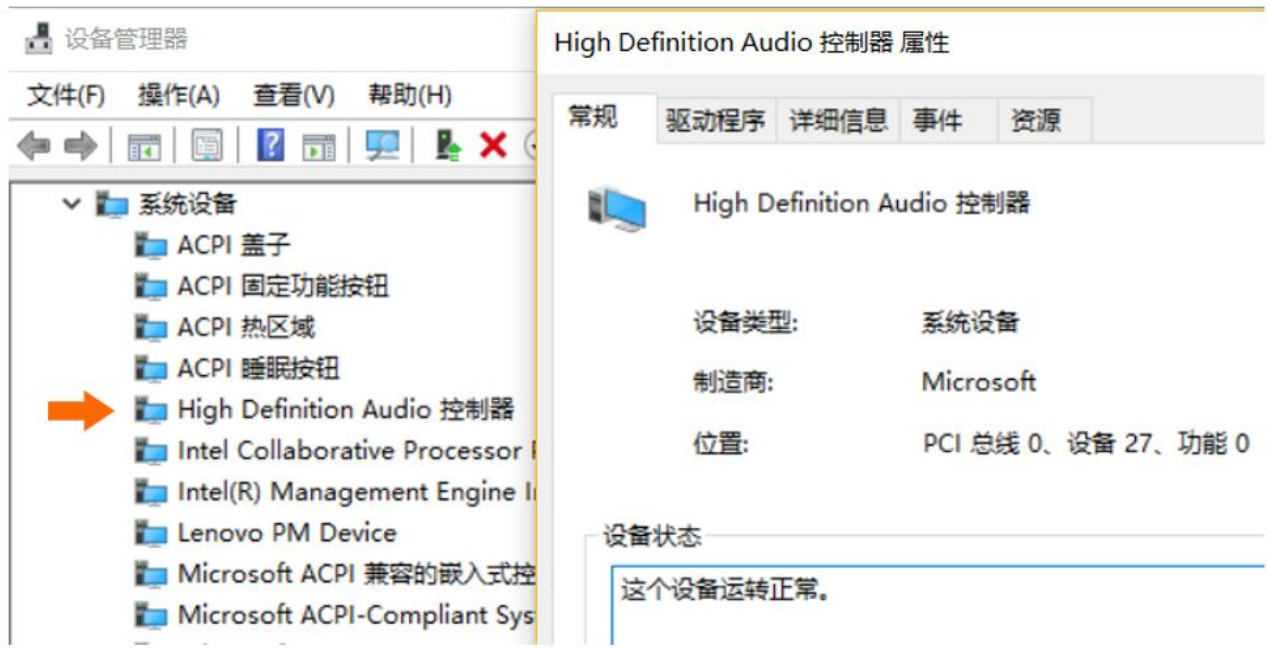
Sine正弦波形

Ramp陡降波形

Triangle三角波形

Square方波形





吉他内部的合成器，合成器加入一些基本的效果比如回响、共鸣等，来模拟不吉他的共鸣腔效果，然后输出最终的模拟信号给功放放大后输出到音响发声。当然，一般吉他内部不会带有较强功能的合成器，所以输出的音色比较单调，如果想要更多的音色和后期处理，需要另外制备独立合成器/效果器。有些高级产品利用两组拾音器产生方向相反、大小相等的两路电流，从而防止各种电磁干扰，这就与第 6 章中介绍的高速通信 PHY 底层的差分信号的本质是类似的。

图 8-17 电吉他上的拾音器

后来，人们采用了野路子，直接对各种乐器的声波采样并保存采样值，只要精度够高，就可以还原出原始波形。采样时只需要针对乐器发出的每个音调的一个大周期采样即可。所有乐器的所有音调的采样值被保存在一张表中，这个表被称为**波表**。波表保存在合成电路旁边一片 ROM 存储器中，当然也可以保存在计算机硬盘中，前者被称为**硬波表**，后者则为**软波表**。早期，由于存储器的成本非常高，人们只能降低采样精度来节省波表占用空间，带来的损失则是音质不高，后来一些合成器逐渐采用大容量存储器，随之而来的音质也就非常高了。有些合成器可以在运行时实时的通过 PCI 接口直接访问被载入 Host 端主存中的波表信息，从而节省卡上的 ROM。

定要组装一台 4 路顶级显卡交火（SLI/CrossFire）的机器，以最高特效，4K 分辨率，流畅运行所有游戏。

图 8-62 游戏《战争机器 4》（左）及《神秘海域 4》（右）截图

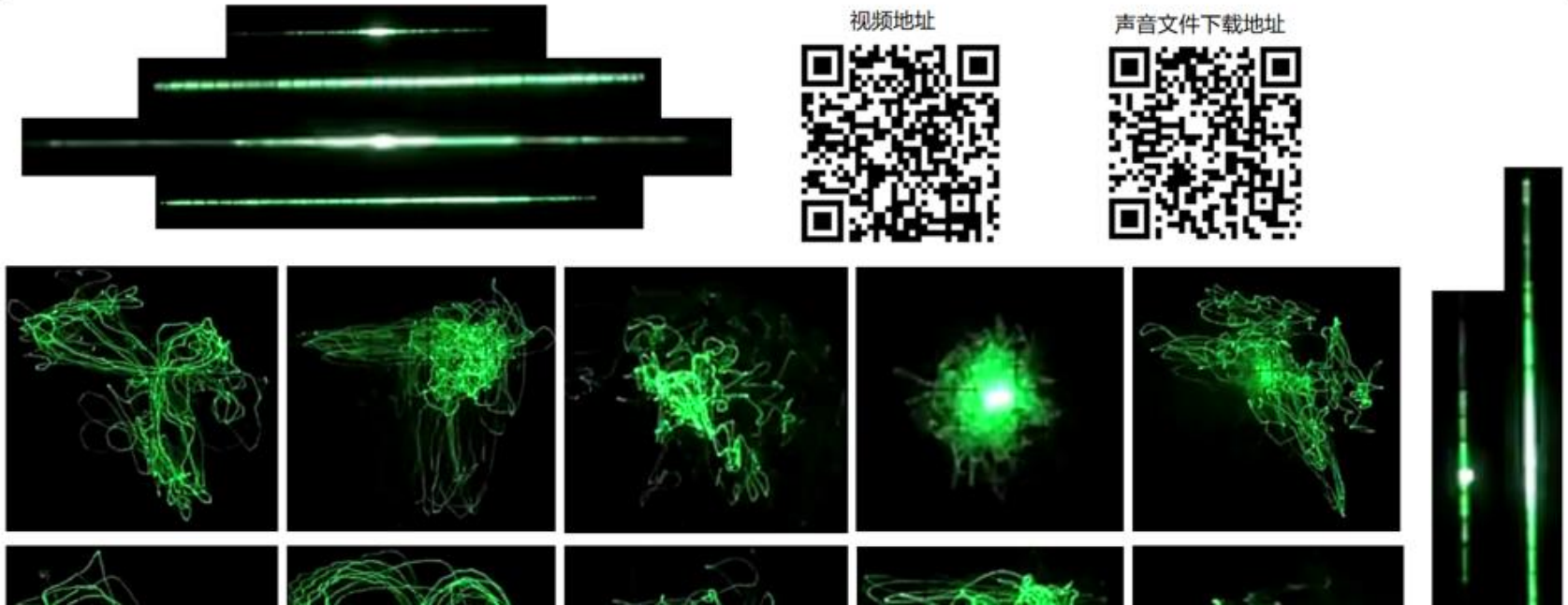
然而，在欣赏这些画面的同时，冬瓜哥心底一直藏着的那个大大的问号，也终于忍不住爆发了：计算机到底是怎么

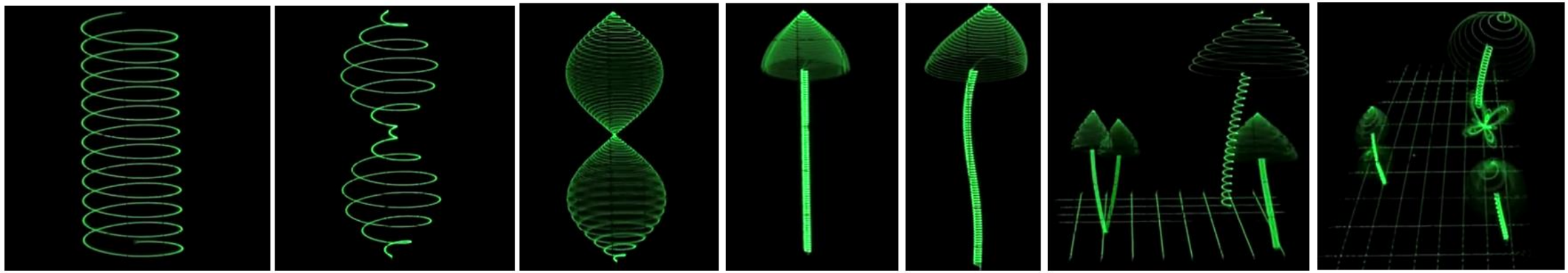
用你的音乐播放器播放任意音乐，就可以了。

吐 8-70 用示波器展示任意波形

现在我们把玩一下，让 X 轴上不再安安分分的加锯齿波，而是加上正弦波或者方波等，让屏幕内的这个波形世界的时间不再匀速流逝，而是扭曲着流逝，甚至可以回退！会发生什么？你可以先冥想一下。如图 8-70 右侧所示的波形。嗯？这镜头好像也不算火爆啊，冬瓜哥这个导演一般啊！

嗯，但是下面的剧情不会让你失望的。如图 8-71 所示为一名高手打算用示波器+声波信号画出蘑菇的图案。建议先不要扫码观看视频，先看完文字。





8.2.2.1 向量文本模式显示

人们是这样做的，如图 8-76 所示，比如对于字符“A”，人们先将其抽象的放入一个栅格中，然后将该字形跨越的栅格节点的坐标位置记录下来，取其中可以组成直线或者近似直线的若干个点，将这些点在栅格中的**相对坐标值**记录下来，并保存到某个地方，比如某个文件中，所有的可能的、常用的字符，都这样处理一番，最后形成的所有字符的形状样点坐标值所保存到的文件，被称为**字库**，或者**字符集**。字库/字符集可以有不同的设计风格，就像每个人写出来的字体都不同一样，有些人写的看上去就那么的端庄，而冬瓜哥写出来就无法让人直视。用户可以替换字库文件以实现装载不同风格字体的字库。字形库，就犹如声音波表一样，它们俩的本质目的是相同的。如图 8-76 所示为需要为向量显示器保存的字形库中的 A 字符的字形信息。

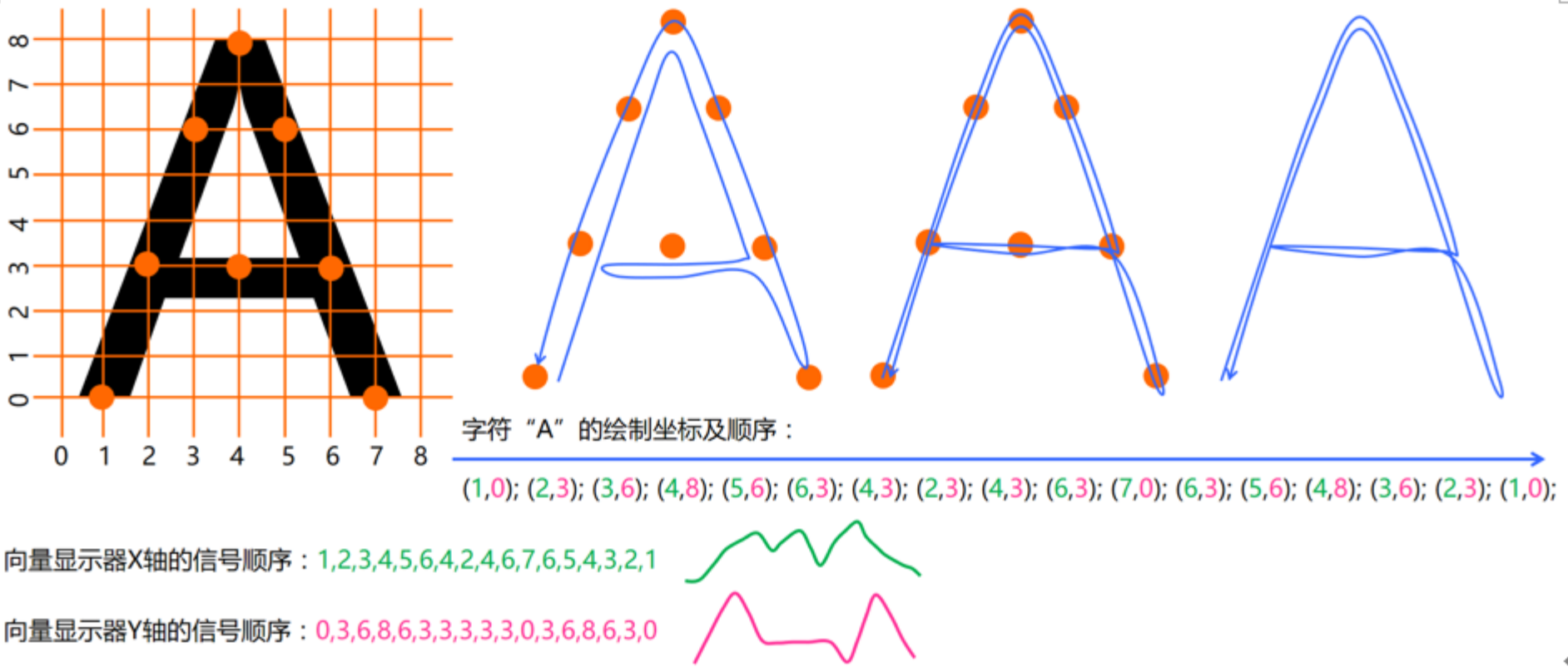
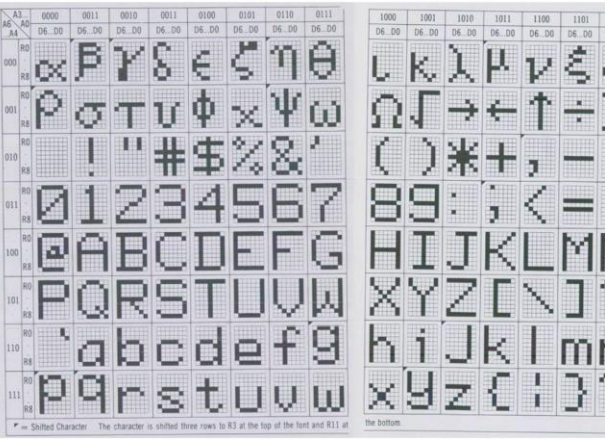


图 8-76 向量模式的字符“A”所需要保存的字形信息

（扫描线）。每个字符所占的方形位置又被称为一个 **Cell**，一横行 Cell 组成一个 **Character Line**。



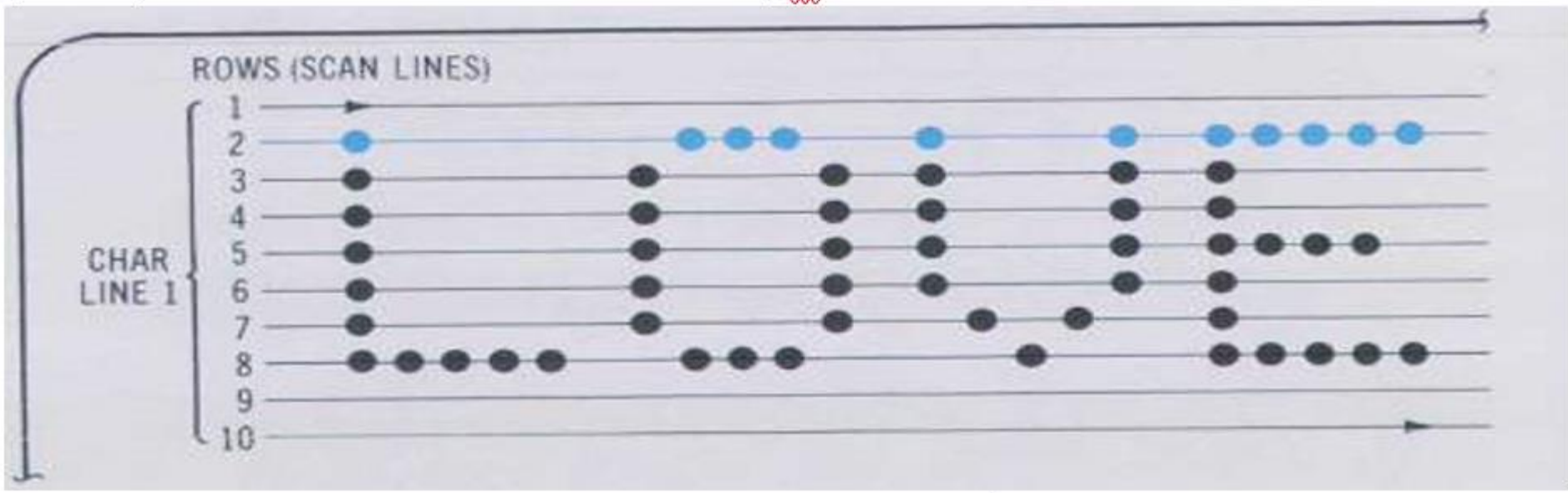
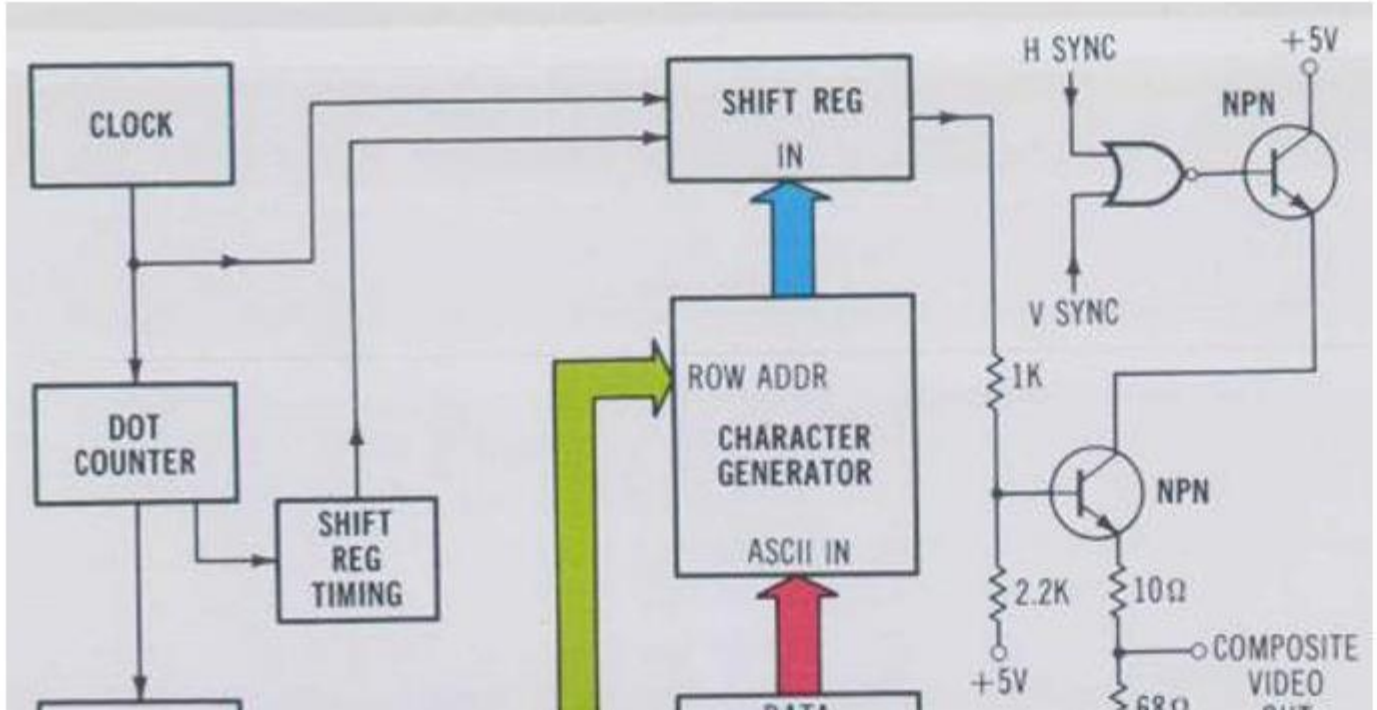


图 8-81 扫描一行会跨过多个字形



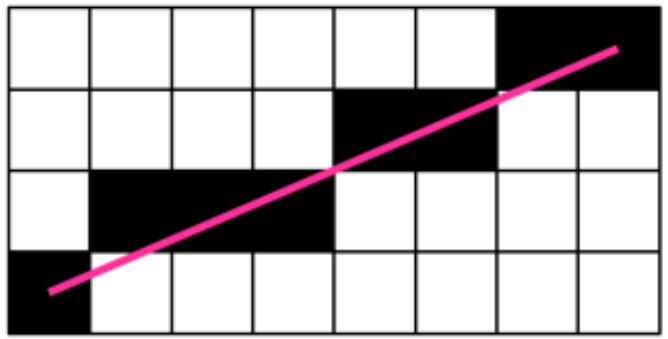
如图 8-82 所示为一个可以向像素显示器播放字形的完整的显卡架构实现。且看它的字形生成器，有两个主要输入，分别为从显存（图中 Display RAM）传送来的用于控制字形选择的 ASCII 码输入，以及从 Cell Line Counter 发来的用于控制让 ROM 输出对

- **8.2.4.3 对模型进行渲染**

现在想象你就是 draw_line(起始坐标，结束坐标)函数的编写者，你怎么根据这两个参数来确定屏幕上哪个像素组成这条线，然后写入对应像素颜色值到 Frame Buffer 对应该坐标的字节？还记得初中学过的，给出两点坐标，求该直线的表达式么？冬瓜哥反正已经忘了。如果早知道今天要研究计算机作图，当年一定不会忘。不过，自己推导一下也能知道，那就是先算出这个直线的斜率，也就是 m=tgθ=两点纵坐标差/两点横坐标差，然后 y=mx+a。**这样，以起始坐标点开始，for（x=起始坐标点，x<结束坐标点+1，x++ ）{ 求 y=mx+b }，即可求出该直线的所有纵坐标像素，将其写入到对应的数组中，也就确定了该直线的所有像素点，这算是真正的图形“渲染”过程中最重要的一步，相当于将图形的骨架勾勒出来，然后就是对每个像素点上色和各种后期特效的加入。**

给出顶点，根据顶点位置计算连接顶点的线段所跨越的像素点的过程，称为**插值（Interpolation）**

提示	实际上，按照上述公式来计算的话，底层函数需要做乘法，而对数字电路来讲，乘法的代价要高于加法。为此，人们使用加法来计算 Y 坐标点。X 每次+1，y 每次增加的其实是 m，那只需要将上一次计算所得的 y 值+m 即可，而不需要用当前的 x 去乘以 m，这就大大节省了计算量。这些函数底层做了大量的这类优化操作。当然，对上层而言，这些底层机制可能可能很少有人去关心了。



且慢！由于我们使用的并非向量显示器而是像素/栅格显示器，所以其每个像素是占有一定面积的，所以直线跨越的像素点并不平滑，如图 8-114 所示，当一条直线跨越了两个像素点时，到底该选哪个像素点呢？这方面就对应了一些不同的算法了，比如根据某种规则来对 Y 值取整数，看与哪个 Y 坐标接近就选哪个。

确认对应图形所占用的所有像素坐标的过程，被称为**栅格化（Rasterization）**，又有人称之为光栅化，因为目前的显示器每个栅格的

8.2.5 3D 图形模型和表示方法

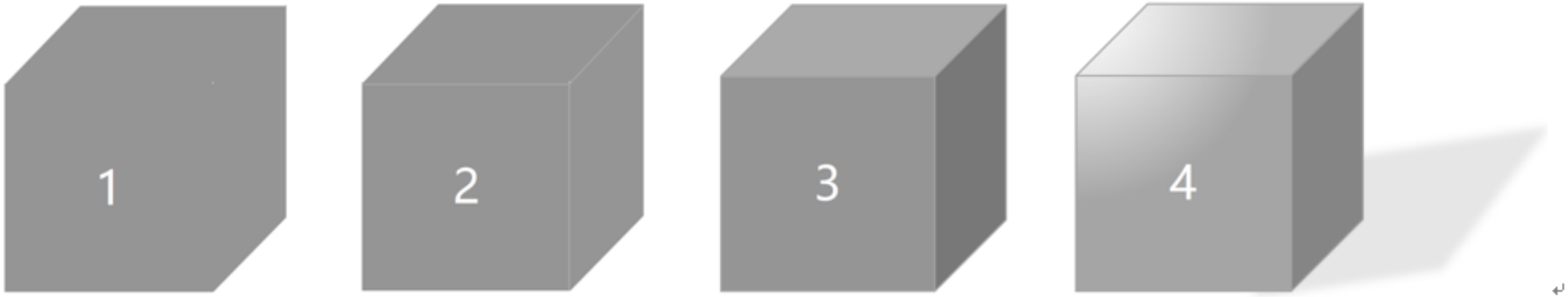
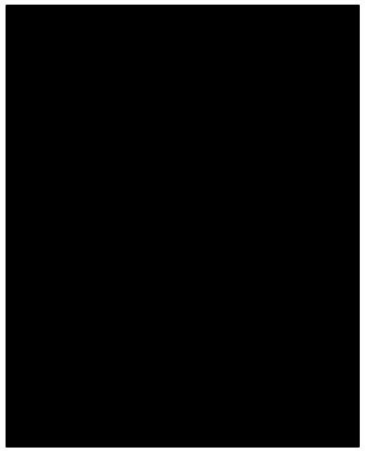


图 8-128 人脑对三维的理解

试想一下，人脑是如何感知三维世界的？如图 8-128 所示。如果挡住 2/3/4 号图形，只观察 1 号，你能判定它是个三维正方体么？不能。对于 2 号图形，多了三条线，产生了三个面，根据经验，已经可以让你判断出大概了。对于 3 号图形，不同的面的亮度不同，已经基本可以确定其意义了。4 号就更不用说了，左上角的光照和右下角的阴影说明了一切。其实，这四个图形在显示器平面上占用的像素面积、形状，是完全一样的。

如果仔细思考一下的话就能体会到，人脑接受的视神经的输入，永远都是 2D 的图像，也就是 3D 世界在人脑中的 2D 投影。人脑之所以理解了 3D，是因为观察一个物体的视角是可以变化的，你可以绕到该物体的侧面重新观察该物体，发现其 2D 投影也会跟着变化，经过对不同视角的投影进行组合识别分析，最终形成一幅 3D 图形的立体感观，所以人自从出生起就不断的接受到周围事物的各种角度投影的训练，形成了经验，才产生了理解 3D 世界的能力。

而对于 3/4 号图形，即便你不用去绕到它侧面去观察，只通过观察其不同位置对光线的反射率/明暗程度，根据经验，也可以判断出明显有一个 z 轴方向的存在，光线就是大自然赐予的参照物和测量尺。但是，设想一下，如果你从来没有尝试过绕到物体侧面观察过，假设你生来就是静止的，动弹不得，连眼珠都转动不了，你有可能只会认为该图形左上角的亮度高一些，右下角有个暗色斑而已，并不会认为该图形存在一个 z 轴方向，你根本不理解阴影为何会存在，产生不了立体感观，所以 4 号图形在你眼中只是一个正方形和两个菱形，和一个一角被挡住的躺着的菱形的组合罢了。



初始阶段

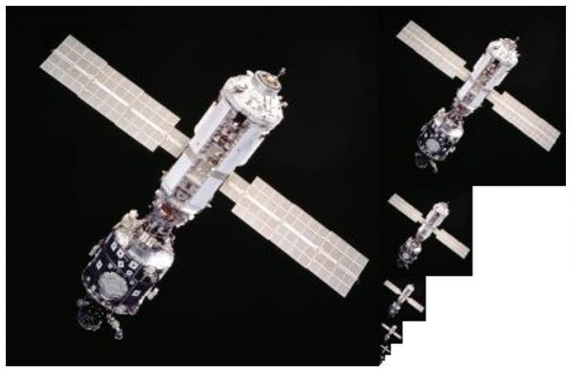
栅格化+环境光

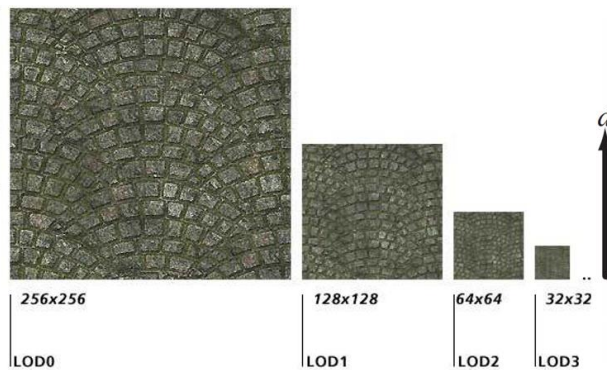
环境光+顶点漫反射光照计算

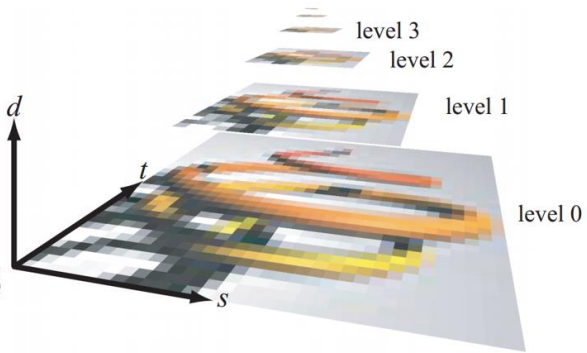
环境光+漫反射插值涂抹

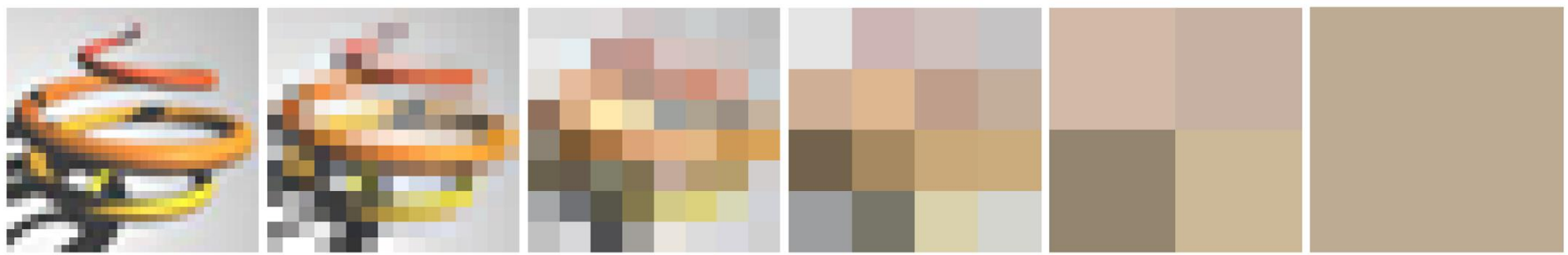
光照+贴图后

灯光减弱









32 × 32　16 × 16　8 × 8　4 × 4　2 × 2　1 × 1

视差贴图的机制是，首先在法线贴图的基础上，像置换贴图一样引入高度图，但是并不用高度图去调制顶点坐标，然后利用高度信息，求出观察者在当前视角下应当观察到的是哪个纹素，然后将相应纹素值写到对应的屏幕像素中，从而形成对应位置上的纹素被抬高的假象。如图 8-215 所示。

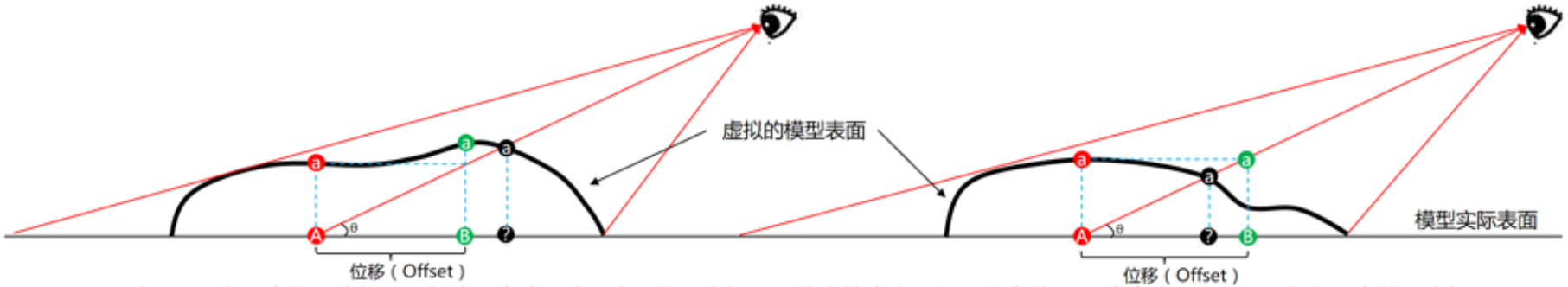


图 8-215 视差贴图基本原理示意图

先假设表面平整，没有高度信息，那么当观察者观察🅐点的时候，就需要对其呈现🅐点纹理坐标对应的纹素，这也是前文中所有贴图方式的做法。但是当要让观察者感觉到高度的时候，当观察者以同样的视角θ观察物体时，其应当观察到的是物体表面的ⓐ点，该点对应的纹素坐标在❓点，**显然，我们需要求出❓点的位置，然后读出对应的纹素写入到屏幕的🅐点，因为观察者最终还是看到屏幕上的🅐点，ⓐ是虚拟出来的，实际上不存在，但是只要将❓的纹素帖到🅐点，观察者就会误认为ⓐ点被抬高了。**这就是视差贴图的奇妙之处。这也是为何称之为视差/位移贴图的原因。

问题是，❓点的坐标无法直接迅速的求出。给定任意一点🅐，其与观察者的连线与物体表面的交点位置，看似是固定的，一定能求出，但是，由于物体表面曲线是无法用方程来描述的，其完全没有规律，其被高度图中的值控制着，高度图是美工人员使用 2D 图像编辑软件，在屏幕上用鼠标刷出来的，就像往脸上抹粉一样，你不可能把你某次抹的粉的分布状况用一个公式描述出来。如图中右侧所示，如果物体表面高度变化了，那么❓点位置也会变化，你根本不知道ⓐ点在哪里。找到光线与表面的交点位置的过程，被称为**求交**，对不规则表面求交的过程非常复杂。

但多数时候，纵使表面被抬高了，但是被抬高的地方有相当比例的位置依然是平坦的，那么这些位置就可以用公式来描述，那就是直线公式，当然，我们抛开晦涩的数学表示法不谈，转看图 8-216 所示。如果被抬高的表面平坦，那么求交操作的结果很容易计算出来，用ⓐ点高度（直接用🅐点坐标从高度图中读出）除以 tagθ，可得出❓点与🅐点之间的 Offset 位移，然后再用🅐点坐标+Offset 就可得出❓点的纹理坐标，然后将对应纹素读出贴入像素即可。

精度。

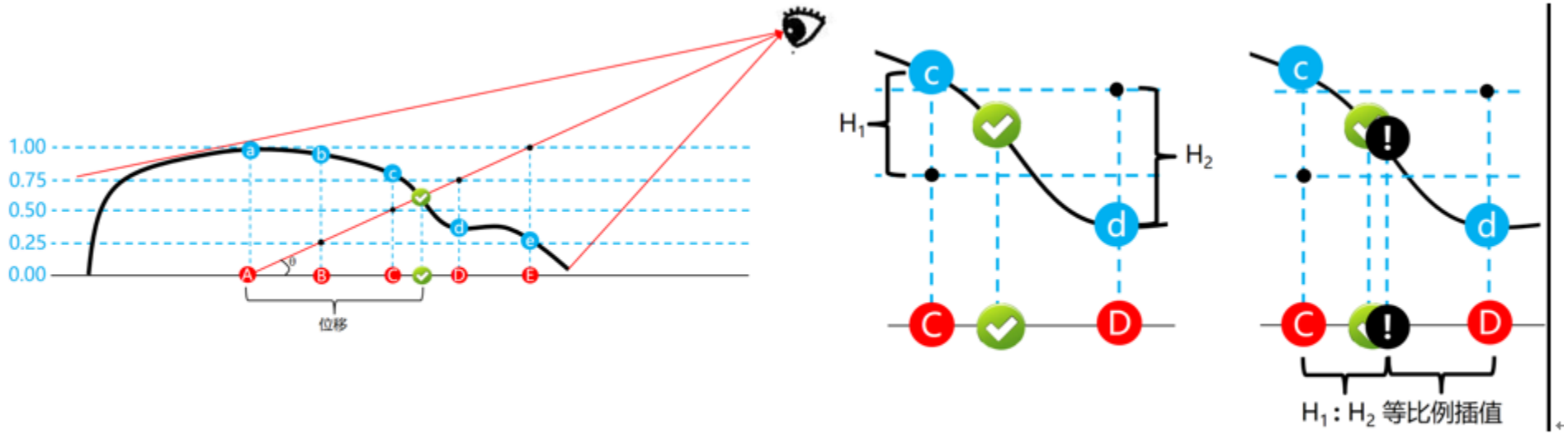


图 8-218 逐次逼近法求交示意图

虽然光线与假表面的交点无法直接求出，但是可以用笨办法来更进一步接近的求出。如图 8-218 所示，在假表面与真表面之间将整个高度切分为多个层级，然后将Ⓐ点到观察者的光线分别与每一层的等高线进行求交，由于等高线为直线，所以很容易求得ⒶⒷⒸⒹⒺ各点的纹理坐标，然后从高度图中对应纹理坐标处读出这些纹素对应的假表面高度ⓐⓑⓒⓓⓔ。显而易见的一个规律是，光线穿越表面的交点的两边的点，也就是本例中的ⓒ和ⓓ，它们的表面高度值和层级高度值是相反的，也就是说，ⓒ点的表面高度值大于其对应的层级高度值，而ⓓ点的表面高度值小于其对应的层级高度值。**交点一定位于被这样两个点夹住的表面区间内。**所以，具体计算时并不需要将所有等高线对应的交点全求出，而是可以先求最高层级的点，考察其对应的表面高度值与层级高度值的关系，如果表面高度值小于层级高度值，则继续与下一个层级高度求交并做相同比较，直到找到发生反转（表面高度值大于层级高度值）的那个点，也就可以确定两个关键点了。或者采用二分法，先从中间层级高度开始求交计算，如果发现表面高度大于层级高度，则从再高一层的层级求交并计算、比较。

然后，利用这两个点各自的表面高度与层级高度之间的差值，用两者的比例去差值ⒸⒹ两点间的距离，最终得

Packet Name	IT_OPCODE	Description
Draw Packets		
DRAW_INDEX_IMMD_BE	0x29	Draw packet used w 16 bit index data in
INDEX_TYPE	0x2A	Sends the current in
DRAW_INDEX	0x2B	Draw packet used w required
DRAW_INDEX_AUTO	0x2D	Draw packet used w
DRAW_INDEX_IMMD	0x2E	Draw packet used w packet
NUM_INSTANCES	0x2F	Sends the number o
MPEG_INDEX	0x3A	MPEG Packed Reg
State Management Packets		
SET_CONFIG_REG	0x68	Write Register Data
SET_CONTEXT_REG	0x69	Write Render State
SET_ALU_CONST	0x6A	Write ALU Constar
SET_BOOL_CONST	0x6B	Write Boolean Cons
SET_LOOP_CONST	0x6C	Write Loop Constar
SET_RESOURCE	0x6D	Write Resource Cor
SET_SAMPLER	0x6E	Write Sampler Cons
SET_CTL_CONST	0x6F	Write Control Cons
SURFACE_BASE_UPDATE	0x73	Inform the CP whic around for CP on R
Wait/Synchronization Packets		
MEM_SEMAPHORE	0x39	Sends Signal & Wa
WAIT_REG_MEM	0x3C	Wait Until a Regist
MEM_WRITE	0x3D	Write DWORD to M
CP_INTERRUPT	0x40	Generate Interrupt f
SURFACE_SYNC	0x43	Synchronize Surfac
COND_WRITE	0x45	Conditional Write t
EVENT_WRITE	0x46	Generate an Event v

第 1 步。上层绘图程序首先创建一个 batchbuffer，在 Host 端，程序的一切行为都要符合规范，申请任何内存空间都要像内存管理程序模块申请，可以将这个过程封装成函数，每次申请时调用即可。这个函数，GPU 厂商为您开发谢谢！实际上 Intel 开发了一整套 Graphic Execution Management（GEM）函数库，就是为您提供方便的，谢谢！申请 batchbuffer 的函数名为 gem_create( )，向该 buffer 中填入命令时，您可以调用 gem_write( )。这里我们创建一个名为 gem_exec 的 batchbuffer 并向其中填入大批 MI_NOOP 命令然后结尾跟着一条 MI_BATCH_BUFFER_END 命令。我们的最终目标是把这个 batchbuffer 的指针通告给 GPU 让它读取执行。

第 2 步。任何命令，不管是单条，还是一批，都必须被放入一个 Descriptor 中，这个套路我们在第 7 章已经介绍的足够充分了。所以，在这一步中，绘图程序需要填个表，把这个 Descriptor 描述出来。GEM 函数库中，该描述表的格式遵循 execbuffer2 结构体（至于为什么会带一个 2 这个冬瓜哥真的不知道）。程序只需要复印这张表，并给其命名为 execbuf，并向其中填入需要执行的命令，如果是批量命令则要将上一步生成的 batchbuffer（名为 gem_exec）填入，然后填入一些其他的控制信息，比如从 batchbuffer 中的哪个位置开始执行指令（batch_start_offset）等，最终这张样式为 execbuffer2 的、名称为 execbuf 的表，需要被传送给 GPU 的驱动程序。

第 3 步。上层绘图程序调用 drmioctl( )将刚才填好的&execbuf（还记得&符号的作用么？在 C 语言中他表示某个对象所在位置的地址指针）表传递给位于内核态的 GPU 驱动程序。Drmioctl( )函数等其实是执行了 ioctl 系统调用（回顾第 5 章的 5.5.6.4 一节，系统调用的更多细节我们将在第 11 章中介绍），从而将信息传入内核态。

第 4 步。位于操作系统内核态的 i915 显卡驱动（i915.ko 程序模块，Linux 操作系统下的内核态的设备驱动程序扩展名为.ko）对应的代码 i915_gem_do_execbuffer( )被系统调用激活，从而将第 2 步生成的 execbuf 表格拿到手。

第 5 步。驱动程序将该命令描述表整理好之后直接 Dispatch 到 Ring Buffer 中，当然，派发过程中其实对应着很多细节步骤，比如修改 Ring Buffer 的控制指针，以及更新 GPU 一侧用于记录 Ring Buffer 首尾指针的寄存器等，这些都由驱动自动完成。整个命令下发过程宣告完成，后续就是 GPU 的渲染流程了。

9.1.3 如何将模拟过程映射为多线程并行计算

那么上述过程具体应该如何映射到多个线程并行运算？比如，可以以原子为单位，每个线程负责计算每个原子在10飞秒后将移动到哪个位置，也就是空间坐标。当然，需要先初始化好对应的数据结构，比如每个原子都用一张表来追踪它们的各种属性，包括：相邻原子的表的指针、本原子的空间坐标值、初速度、当前速度、当前受力值等等。

每个线程的入口函数的输入值是周边原子的作用力，由于原子间力是短程作用力，所以距离较远的原子作用力就可以忽略，一般只考虑其化学键链条上的3个原子对其的作用力，超过3个以后的认为作用力为0。输入值还包括该原子的初始三维坐标位置以及该原子的速度（初始速度为0或者某一既定速度）。输出值则是在牛顿力学公式的作用之下经过10飞秒加速之后该原子的新的空间坐标和速度矢量，并将这些输出值记录在各自的表中。

该线程中又会有大量函数相互作用，比如：有函数会专门根据当前原子的化合价以及与其化合的其他原子都是谁，然后计算静电力，有的函数则负责计算氢键力、范德华力等，这些力的计算公式有些非常复杂，需要较大的运算量。最后，求合力F，算出初速度为0，合力F，质量m，t=10fs之后的该原子的位置和速度，最后这一步相信高中物理及格的朋友都可以算出来了。

那么，第一个10飞秒的模拟运算结束之后（注意，并不是说运算过程持续了10飞秒，而是说算出原子运动10飞秒后的速度和坐标，这个运算过程耗费的时间远大于10飞秒）。应该怎么办呢？当然是要把计算完的值更新到每个原子对应的结构体表的对应项目中。然后呢？当然是每个线程都需要从本原子相邻的其他原子（不超过3级）的记录表中取出它们各自的空间坐标，然后根据各种力计算出本原子在新的位之下所受的新的合力矢量，然后继续开始运算再一个10飞秒之后本原子的坐标和速度值了。就这样，以10飞秒为步进一直向前推进。

这个过程中会产生一个潜在问题，在多核心计算机系统中，多线程在时间上是物理并发执行的，如果某个线程运算的较快，先结束了，而其相邻原子对应的线程运算较慢尚未结束，那么如果先结束的线程直接去读相邻原子的记录表取出坐标，那么取出的将会是旧值，从而导致运算错误。必须实现一种方式，让相互依赖的线程之间形成一种等待关系，只有所有这些线程都执行到某个步骤之后，才能继续下一步的执行，这种方式被称为屏障，Barrier。这个概念我们在第6章介绍访存的时空一致性时中初步涉及过。互斥锁、屏障，都是多线程同步的方式，各自应用场景也不同。

MPI_Bcast(void *buffer,int count,MPI_Datatype datatype,int root,MPI_COMM_WORLD)。该函数用于线程将数据广播给通信域内的所有除了发送方线程之外的节点。参数 root 表示发送广播的源线程标识。发送方调用 MPI_Bcast( )，接收方也必须在同一位置调用 MPI_Bcast。嗯？接收端为什么也要发起广播？不是的。接收端调用该函数的作用是为了接收广播，该函数会判断当前调用者线程的标识是否与参数 root 相同，如果相同，则表明当前线程是发出数据，如果不同，则表明当前线程之所以调用该函数是为了接收 root 线程发来的广播。这就是 MPI 编程方式的特点之一，也就是发送方和接收方永远都要配合起来，一唱一和。如果有人先唱到了这一段，那就阻塞等待，等对方也唱到这一段之后，继续往下唱。下面的函数如无特殊说明，均为发送方和接收方同时调用。MPI_Bcast( )函数底层其实也调用了 MPI_Send( )和 MPI_Recv( )函数，其本质就是将向所有节点的单播过程封装为广播。同理，下面介绍的所有聚合通信类函数，都是用基本的 MPI_Send( )和 MPI_Recv( )函数封装而成的，你也可以自己封装一个聚合通信类函数出来。如图 9-16 所示为 MPI_Bcast( )的执行过程示意图。

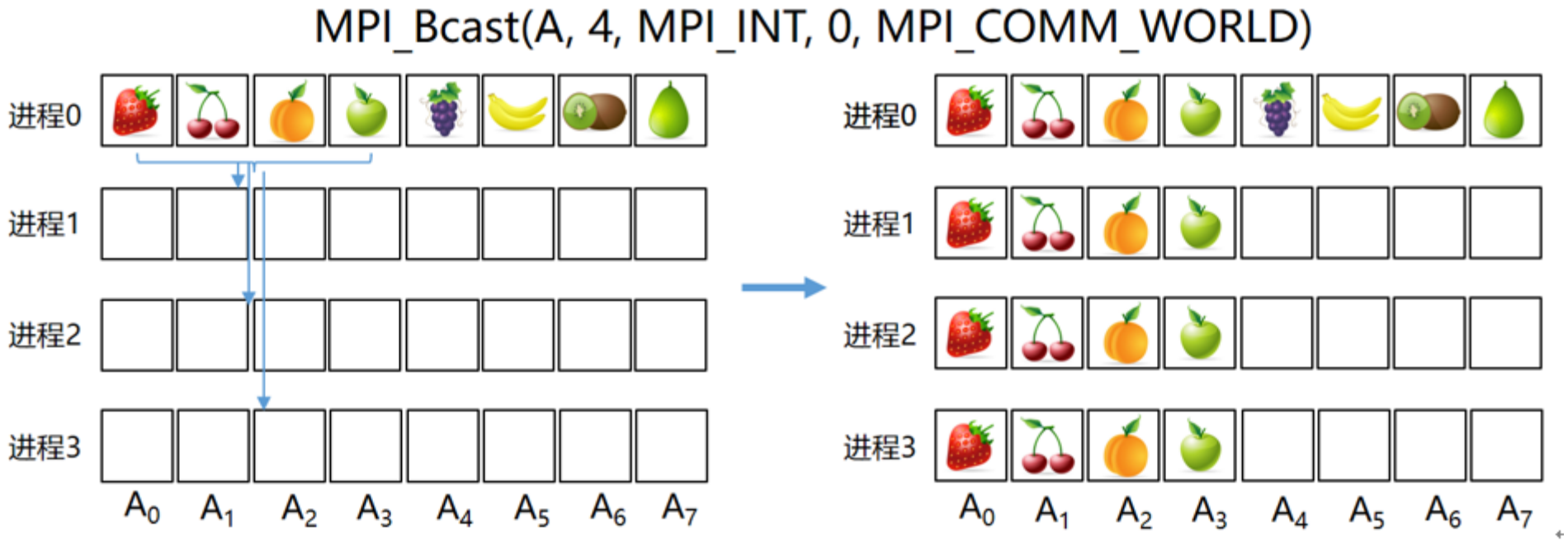


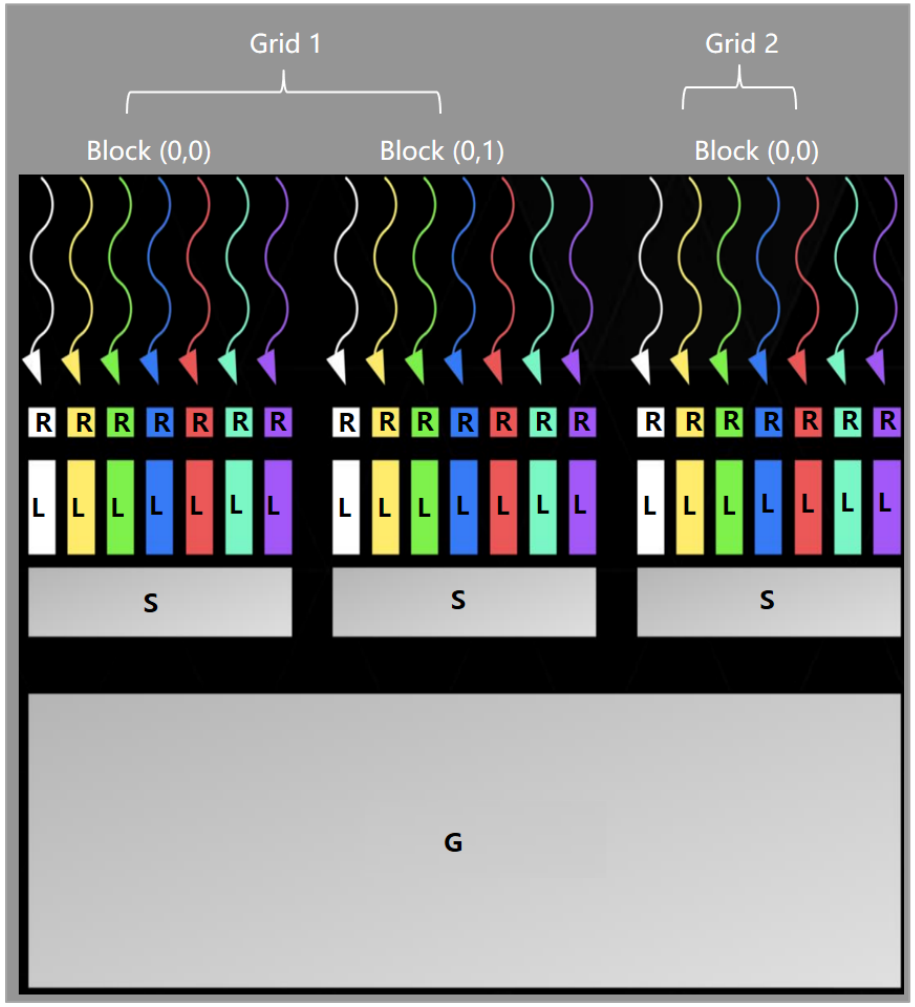
图 9-16 MPI_Bcast( )的执行过程示意图

9.5.3.2 一个极简的 CUDA 程序

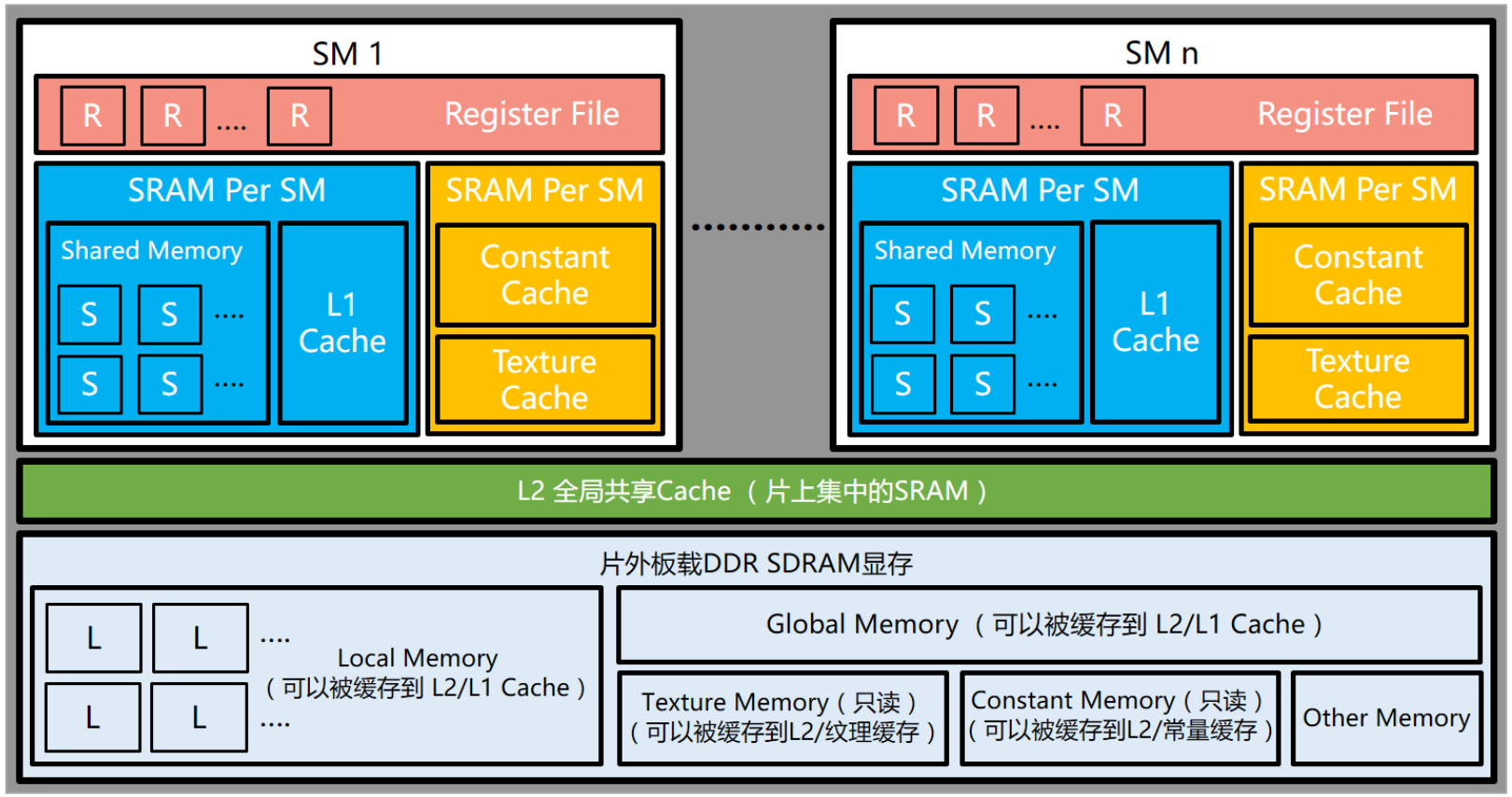
下面我们通过一个极简单的 CUDA 程序来向大家展示一下 CUDA 程序的基本编写方式。

```
#include<stdio.h>
#define N 8  //代码中以 N 代替 8
__global__ void add(int *a,int *b,int *c)  //声明一个函数，该函数逻辑需要在 GPU 上执行
{
    int tid=blockIdx.x;
    if(tid<N)
        c[tid]=a[tid]+b[tid];
}
int main()  {
    int arr1[N],arr2[N]; int sum[N];  //定义三个数组
    for(int i=0;i<N;i++) {arr1[i]=i; arr2[i]=i+1;}  //对数组进行数据填充
    int *a; int *b; int *res;  //定义三个指针用于保存下面分配好的显存的指针
    cudaMalloc((void**)&res,sizeof(int)*N);  //分配 N*4 KB 显存用于盛放两个数组之和
    cudaMalloc((void**)&a,sizeof(int)*N);  //分配 N*4 KB 显存用于盛放数组 a
    cudaMalloc((void**)&b,sizeof(int)*N);  //分配 N*4 KB 显存用于盛放数组 b
    cudaMemcpy(a,arr1,sizeof(int)*N,cudaMemcpyHostToDevice);  //向显存拷贝数据
    cudaMemcpy(b,arr2,sizeof(int)*N,cudaMemcpyHostToDevice);  //向显存拷贝数据
    add<<<N,1>>>(a,b,res); //调用核函数，定义 N 个 Block，每 Block 包含 1 个线程，并行执行该函数
    cudaMemcpy(sum,res,sizeof(int)*N,cudaMemcpyDeviceToHost); //把结果从 res 拷贝回到 sum
    cudaFree(a);  //通知 GPU 释放显存
    cudaFree(b);  //通知 GPU 释放显存
    for(int i=0;i<N;i++) {printf("%d\n",sum[i]);}  //显示计算结果
  }
```

逻辑视图



物理视图



R：Register File，位于GPU内的SM内，SRAM形态，访问延迟1周期，每个线程独享各自的一份。
S：Shared Memory，位于GPU片上，SRAM形态，访问延迟数个周期，多个Block均分容量，每个Block内所有线程共享。
L：Local Memory，位于GPU片外板载显存中，DDR SDRAM形态，访问周期约500周期，每个线程独享各自的一份。
G：Global Memory，位于GPU片外板载显存中，DDR SDRAM形态，访问周期约500周期，所有Grid中线程共享。

- **9.5.3.6 基于 CUDA 的 PhysX 库效果**

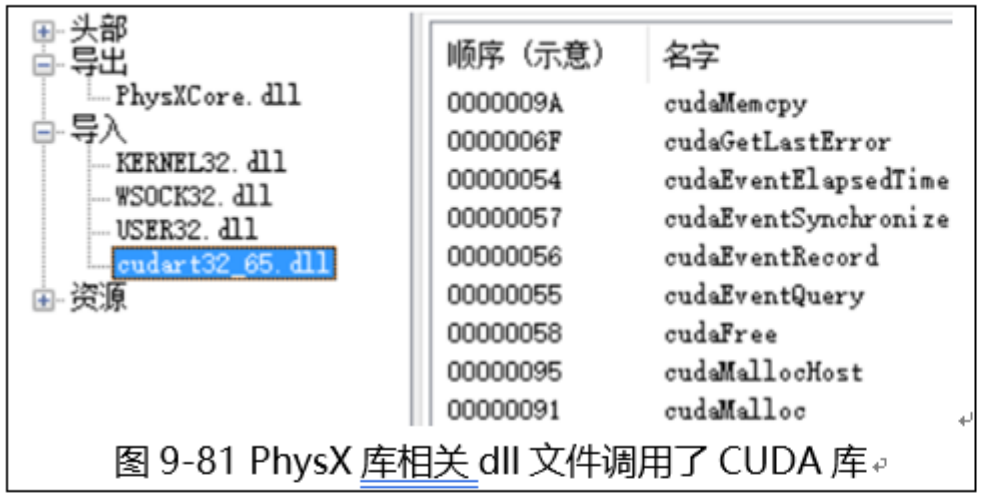


图 9-81 PhysX 库相关 dll 文件调用了 CUDA 库

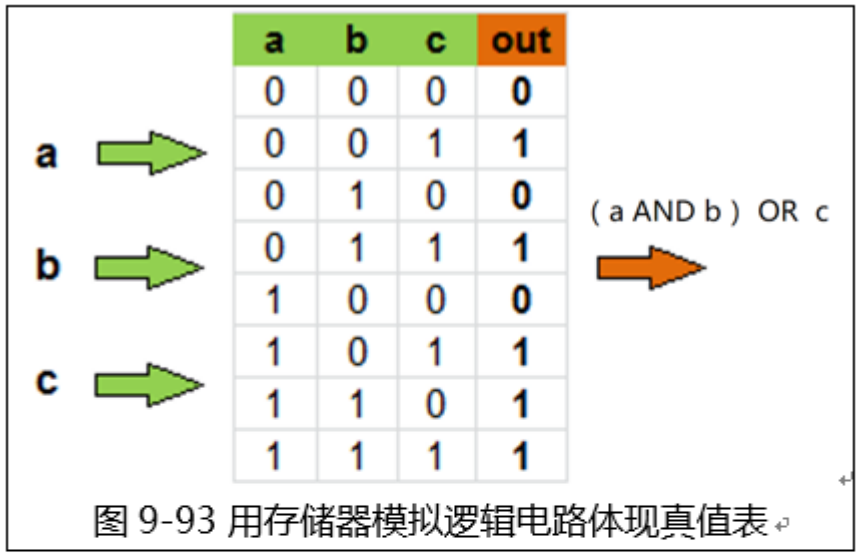
基于 NVidia GPU 的 CUDA 加速计算越来越流行，在各行各业得到了广泛应用。Nvidia 自家搞的物体物理碰撞计算加速库 PhysX 本身就是基于 CUDA 实现的（如图 9-81 所示）。游戏程序调用 PhysX 库中的物理碰撞模拟算法，将模型的顶点坐标位置利用 CUDA 加速计算出来，返回到 Host 主存，然后游戏再走 D3D 渲染流程，将顶点新位置被计算好的模型渲染出来。物理碰撞特效非常适合使用并行计算来加速，比如有一千颗冰晶，游戏中的角色朝某个方向释放后，每颗冰晶都有个初速度和模拟阻力（甚至还可以引入当时的环境风速），然后针对每一颗冰晶计算其下一个时间单位（比如如果决定该物理特效动画流畅度为每秒 30 帧，那么每个时间单位就是 33ms）所在的位置，所使用的算法再简单不过，就是牛顿运动定律，根据合力求时间 t 后的位移量。如果前方有遮挡物，则根据位移量和遮挡物坐标算出发生碰撞的时间点和位置（这个过程被称为**碰撞检测**），然后在该时间点针对该粒子做反弹计算，利用的公式，我想任何一个高中毕业的朋友都做过小球以一定入射角撞墙反弹的题，不再赘述（其实冬瓜哥已经忘了，尴尬一下）。经过大量运算之后，这些冰晶颗粒就按照物理规律运动，产生震撼的物理模特

域 2》对 PhysX 达到了“滥用”的程度，对于当年的显卡当然会力不从心，但是时隔数年之后，可以特效全开 2K 分辨率下达到 70 帧以上，冬瓜哥甚是欣欣。写到这里，冬瓜哥不由自主的又打开了 PC 机载入《圣域 2》进去享受了一下里面的风景，以及令人心旷神怡的物理特效。如图 9-82 所示，在 GTX980 显卡强悍算力的支撑下，人物脚下的火圈在地上留下了按照真实物理相互作用的火星拖尾；火圈发出的热风将地上的石子、火星等向外吹；大量的（据冬瓜哥观察起码有数千个）魔法冰晶颗粒被爆发出来碰到场景中的物体上反弹并相互碰撞然后按照物理规律运动掉落并堆积在地上；风刮起地上数百片落叶并随风飘动下落；人物走路驱赶起地上的石子和落叶，所有石子/树叶都会有投影，所有的石子落叶都可以按照物理规律自旋转。而如果关闭 PhysX 特效，石子、落叶和冰晶都不会出现。《圣域 2》一度让冬瓜哥感觉没有物理特效的游戏都索然无味。

图 9-82 PhysX 物理加速库以及游戏《圣域 2》中的 PhysX 特效截图

值得一提的是，PhysX 可以被配置为使用 CPU+软件方式来计算，但是在冬瓜哥的 Intel 酷睿 i7 CPU 下，使用软计算运行圣域 2，当放出冰晶魔法时，已经不是每秒几帧的帧率了，而是几秒一帧，此时显卡基本处于闲置状态，CPU 使用率 100%，然而，CPU 就算忙的满头汗也赶不上显卡弹指一挥间。游戏画面成了名副其实的幻灯片速度。而且软算法考虑到 CPU 的算力，对物理效果做了大幅简化，无论是粒子数量还是排布的凌乱程度上，真实度差了一大截。

9.6.3 FPGA 是如何工作的



a	b	c	out
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

图 9-93 用存储器模拟逻辑电路体现真值表

与 CPLD 采用与/或门阵列将多个乘积项相或的方式来实现逻辑的方式不同的是，FPGA 采用另一个奇妙的解决办法。我们在第 1 章中介绍过，任何（注意，是任何，没有之一也没有例外）逻辑电路都可以表达成真值表对外进行响应，输入值与输出值严格一一对应。可以这样认为：只要实现了这个真值表，就可以完成该电路的功能，**而实现这个真值表并不一定非要用门电路，可以用存储器。比如，当输入值为 0000 时输出 10，输入为 0001 时输出 11，那么可以准备 16 行、每行 2bit 的存储器，然后用 4bit 输入值作为行号索引，将对应行号的输出值保存在该行的 2bit 上，每次得到某个输入值，就到对应行号读出输出值输出，这也可以真实体现真值表中的逻辑，而不需要任何逻辑门电路。**如图 9-93 所示为使用 4 字节的存储器实现了一个（a AND b） OR c 的逻辑真值表。当然，可以使用容量更大的存储器实现（或者说存储）更复杂的逻辑。用于存放真值表的存储器被称为 **LUT（Lookup Table）。**

利用 LUT 查表来“计算”数据时，其过程并不是“计算”，而是“查询”，也就是说，LUT 中保存的数据，其实都是预先被计算好的，相当于把一块逻辑电路的所有结果展开到了 LUT 中。如果使用逻辑电路来搭建这个逻辑，则只需要一个双输入与门加一个双输入或门，如果换算成开关的话（不考虑实际工程设计），只需要与门两个开关和或门两个开关共 4 个开关即可实现。而 LUT 实现相同逻辑所耗费的晶体管数量显然要大于逻辑门。

为了实现高速查表，LUT 一般使用 SRAM

9.6.4 FPGA 编程及应用形态

基于 PLD 器件进行开发逻辑，首先确定该逻辑的功能，然后将其翻译成 FPGA LUT 内的真值表。这个过程如果全靠人脑，将会非常复杂。为此，人们开发了**硬件描述语言（Hardware Description Language，HDL）**。开发者先把逻辑功能描述成 HDL，然后由 HDL 编译器自动分析并将这些逻辑编排落地到 FPGA 中。

比如 “有个逻辑模块，它有 3 输入 8 输出，当输入为 xxx 时输出为 xxxxxxxx；当输入为 xxx 时输出为 xxxxxxxx；.....” ，我们知道这是在描述一个 3-8 译码器。很显然，这种描述方式中并没有出现任何对逻辑门的描述，它只是在描述这个电路的行为是什么。这种 HDL 描述方式被称为**行为级描述**。

而如果是这样来描述：“有这么个器件，有 xxx 这些输入和输出信号，其内部有个驱动能力 x 延迟 x 的双输入与门连接了 x 和 x 及 x 信号，其内部有个驱动能力 x 延迟 x 的双输入异或门连接了 x 和 x 及 x 信号，.....” 。其被称为**门级描述**。

在行为级和门级之间还存在一种叫做**寄存器传输级描述（Register Transfer Level，RTL）**的中间描述方式。行为级描述层次太高太抽象，代码非常精简，有种你咋不上天的感觉。而门级描述非常底层（其实还有更底层的开关级描述，因为门电路是由开关组成的），代码量非常大，写起来不方便。而 RTL 级描述则成为比较主流的描述方式，其对行为级描述的更加具体，但是却又避免谈及底层的电路结构，其只聚焦在数据在进入组合逻辑和时序逻辑电路时的更具体的行为。

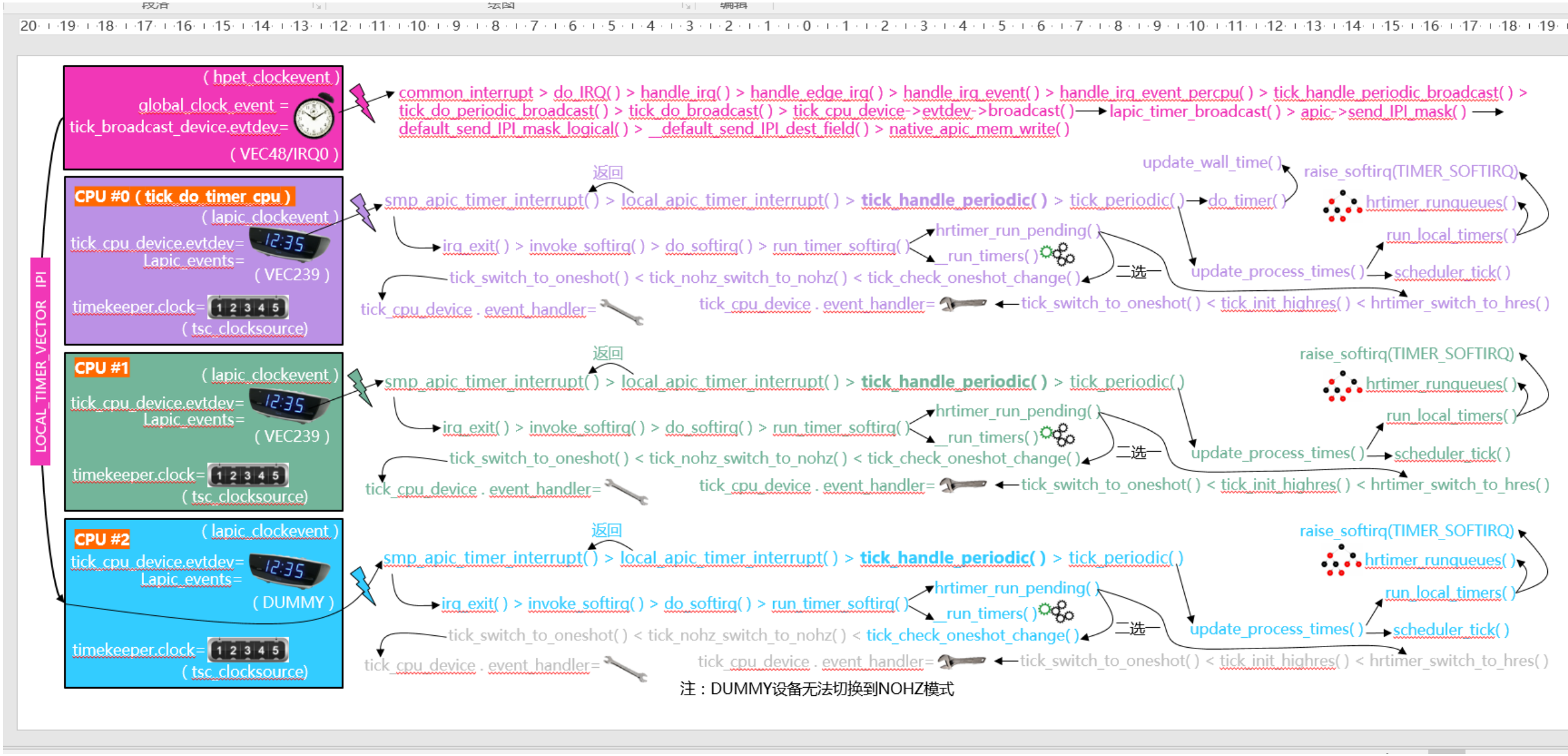
上述描述方式都属于 HDL。目前有两种比较流行的 HDL，分别为 VHDL 和 Verilog HDL。如图 9-104~106 所示为一些简单的行为级和门级描述的例子。

```
1 module q_decode_38(data_in,data_out);
2
3    input[2:0] data_in;          //端口声明
4   output[7:0] data_out;
5   reg[7:0] data_out;
6
7 always@(data_in)
```

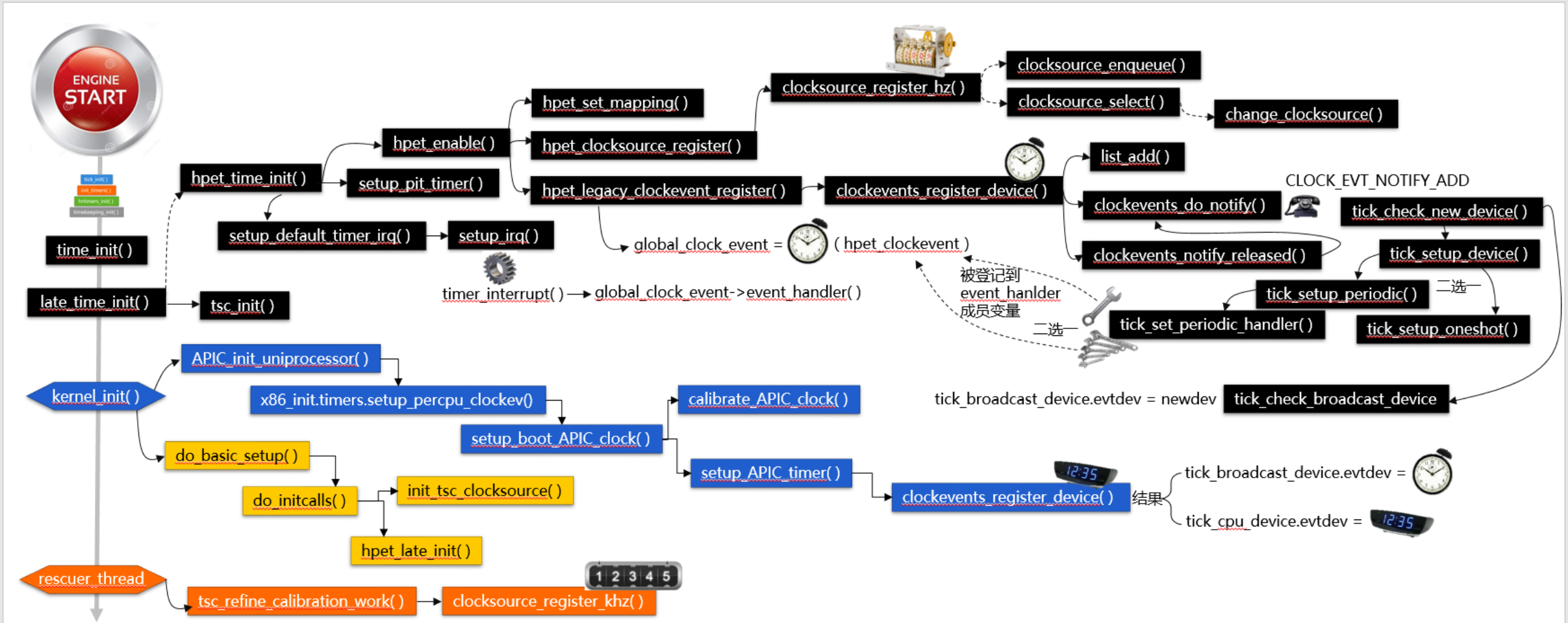
```
1  module ActiveStructure_mux2(datain_A, datain_B, sl, dataout);
2    inputdatain_A,
3    datain_B;
4    inputsl;
5    outputdataout;
6    regdataout;
7
```

```
1  moduleTwoMux(datain_A,datain_B,sl,dataout);
2     inputdatain_A,
3      datain_B,
4      sl;
5      outputdataout;
6
7   not U1(nsl,sl);
```

《大话计算机》第10章部分图示



《大话计算机》第10章部分图示



《大话计算机》第10章部分图示

struct timekeeper { }

HZ | wall_to_monotonic

xtime | total_sleep_time

jiffies | boot_time

hrtimer

global_clock_device | tick_cpu_device

tick_broadcast_device | timer_list

tick_broadcast_mask

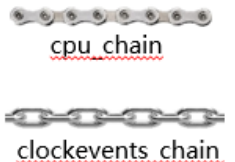
lapic_events

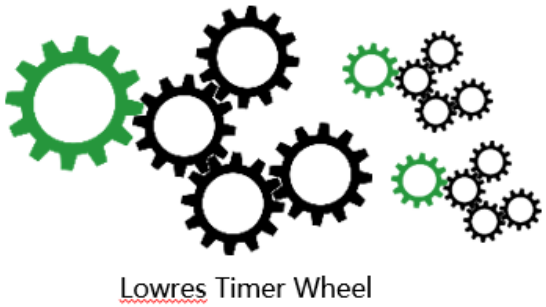
clockevents_released

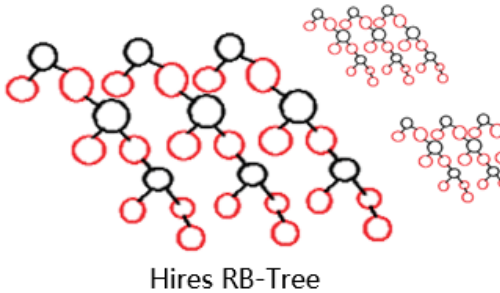
clockevent_devices

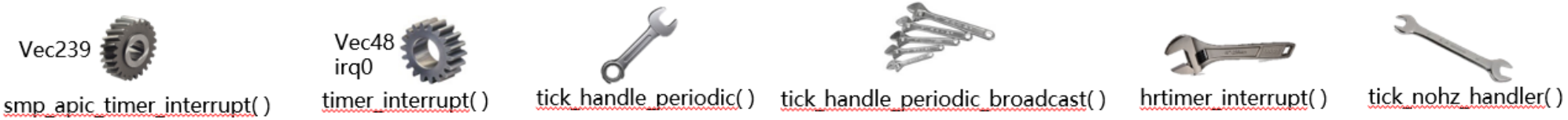
RTC硬件

clocksource_list

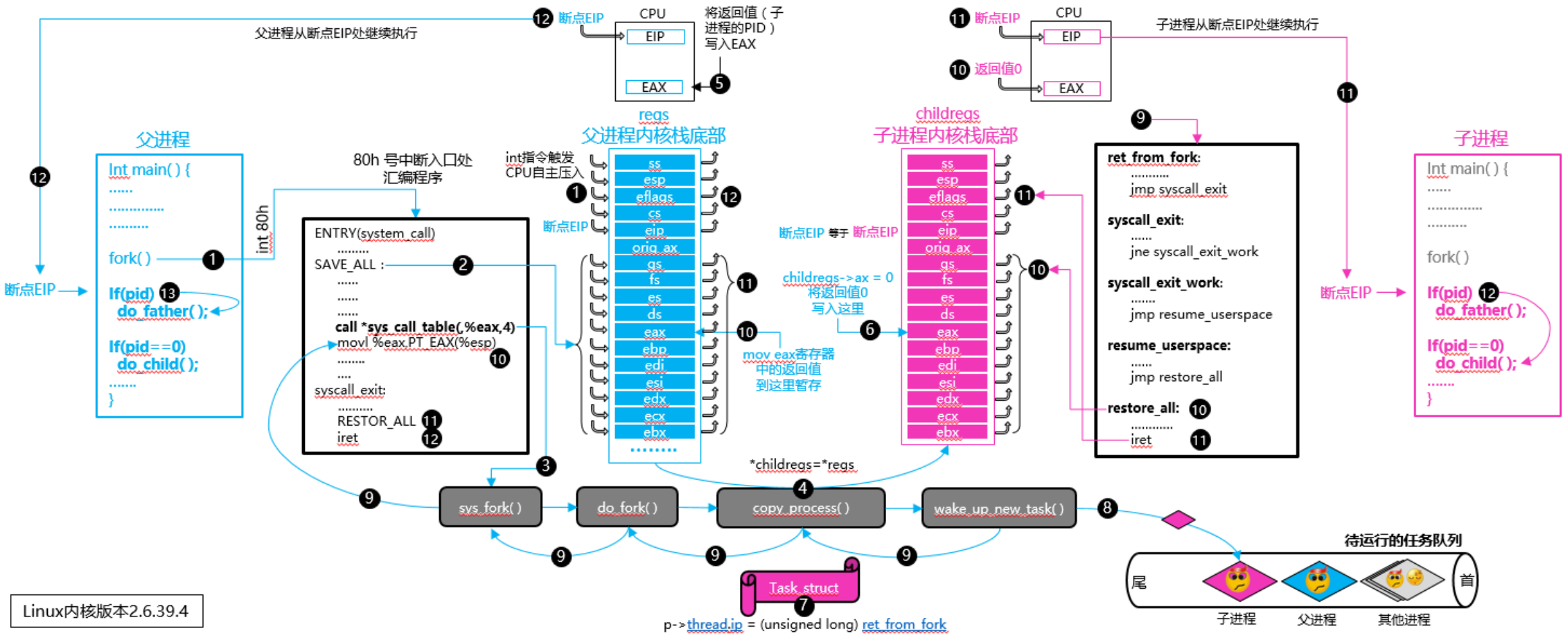




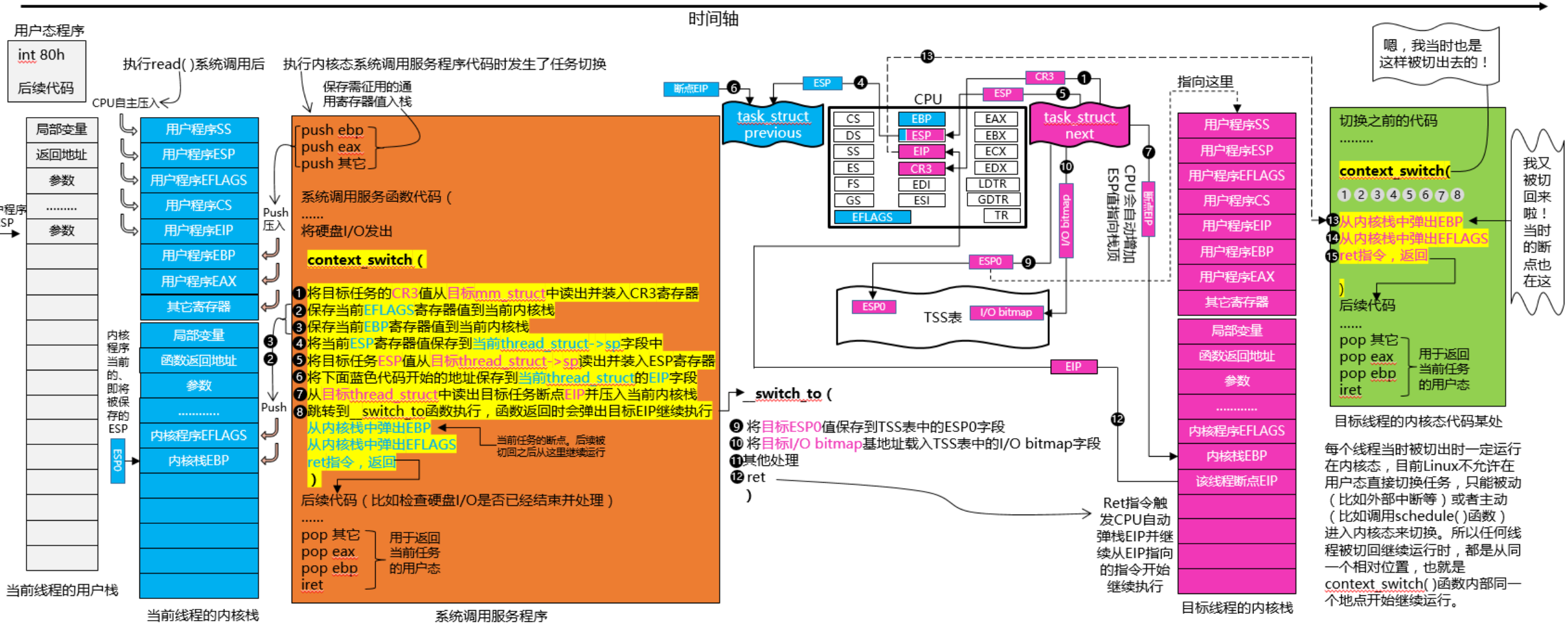




《大话计算机》第10章部分图示



《大话计算机》第10章部分图示

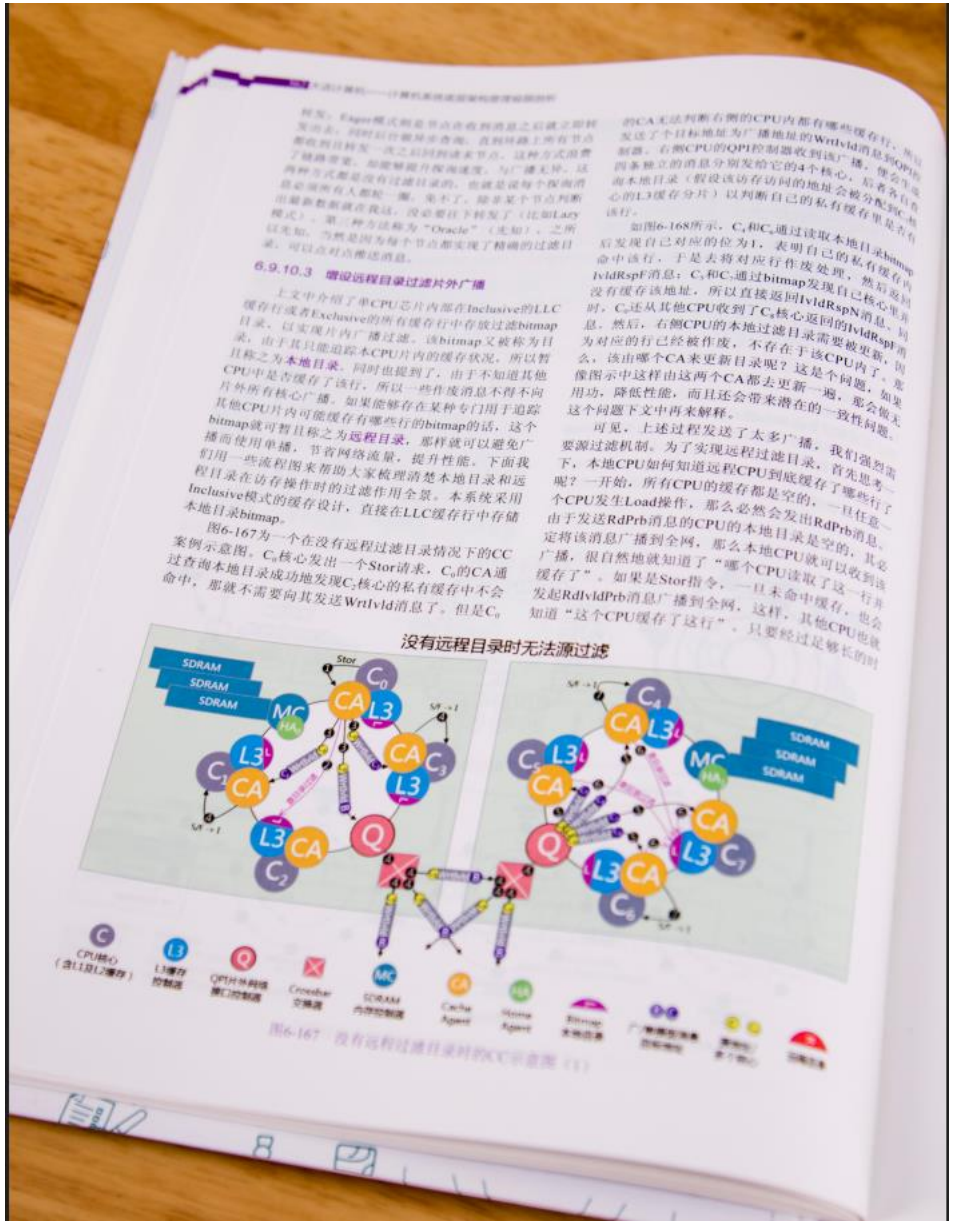



冬瓜哥◎著
大话计算机
计算机系统底层架构原理极限剖析
清华大学出版社


卷2
大话计算机
冬瓜哥◎著
CONCEPT
计算机系统
底层架构原理极限剖析
清华大学出版社


大话计算机


大话
计算机
冬瓜哥◎著
计算机系统
底层架构原理极限剖析
北京大学出版社


SDRAM
SDRAM
SDRAM
Stor
CA
L3
MC
Q
CPU核心
（含L1及L2缓存）
L3缓存
控制器
QPI片外网络
接口控制器
Crossbar
交换器
SDRAM
内存控制器
Caching
Agent
Home
Agent

转发；Eager模式则是节点在收到消息之后就立即转发出去，同时后台做异步查询，直到环路上所有节点都收到且转发一次之后回到请求节点，这种方式浪费了链路带宽，却能够提升探询速度，与广播无异。这两种方式都是没有过滤目录的，也就是说每个探询消息必须所有人都轮一圈，免不了，除非某个节点判断出最新数据就在我这，没必要往下转发了（比如Lazy模式）。第三种方法称为“Oracle”（先知），之所以先知，当然是因为每个节点都实现了精确的过滤目录，可以点对点推送消息。

6.9.10.3 增设远程目录过滤片外广播

上文中介绍了单CPU芯片内部在Inclusive的LLC缓存行或者Exclusive的所有缓存行中存放过滤bitmap目录，以实现片内广播过滤。该bitmap又被称为目录，由于其只能追踪本CPU片内的缓存状况，所以暂且称之为**本地目录**。同时也提到了，由于不知道其他CPU中是否缓存了该行，所以一些作废消息不得不向片外所有核心广播。如果能够存在某种专门用于追踪其他CPU片内可能缓存有哪些行的bitmap的话，这个bitmap就可暂且称之为**远程目录**，那样就可以避免广播而使用单播，节省网络流量，提升性能。下面我们用一些流程图来帮助大家梳理清楚本地目录和远程目录在访存操作时的过滤作用全景。本系统采用Inclusive模式的缓存设计，直接在LLC缓存行中存储本地目录bitmap。

图6-167为一个在没有远程过滤目录情况下的CC案例示意图。C_0核心发出一个Stor请求，C_0的CA通过查询本地目录成功地发现C_2核心的私有缓存中不会命中，那就不需要向其发送WrtIvld消息了。但是C_0的CA无法判断右侧的CPU内都有哪些缓存行，所以发送了个目标地址为广播地址的WrtIvld消息到QPI控制器。右侧CPU的QPI控制器收到该广播，便会生成四条独立的消息分别发给它的4个核心，后者各自查询本地目录（假设该访存访问的地址会被分配到C_4核心的L3缓存分片）以判断自己的私有缓存里是否有该行。

如图6-168所示，C_4和C_6通过读取本地目录bitmap后发现自己对应的位为1，表明自己的私有缓存内命中该行，于是去将对应行作废处理，然后返回IvldRspF消息；C_1和C_3通过bitmap发现自己核心里并没有缓存该地址，所以直接返回IvldRspN消息。同时，C_0还从其他CPU收到了C_4核心返回的IvldRspF消息。然后，右侧CPU的本地过滤目录需要被更新，因为对应的行已经被作废，不存在于该CPU内了。那么，该由哪个CA来更新目录呢？这是个问题，如果像图示中这样由这两个CA都去更新一遍，那会做无用功，降低性能，而且还会带来潜在的一致性问题。这个问题下文中再来解释。

可见，上述过程发送了太多广播，我们强烈需要源过滤机制。为了实现远程过滤目录，首先思考一下，本地CPU如何知道远程CPU到底缓存了哪些行了呢？一开始，所有CPU的缓存都是空的，一旦任意一个CPU发生Load操作，那么必然会发出RdPrb消息。由于发送RdPrb消息的CPU的本地目录是空的，其必定将该消息广播到全网，那么本地CPU就可以收到该广播，很自然地就知道了“哪个CPU读取了这一行并缓存了”。如果是Stor指令，一旦未命中缓存，也会发起RdIvldPrb消息广播到全网，这样，其他CPU也就知道“这个CPU缓存了这行”。只要经过足够长的时间，就可以积累足够多的追踪记录，从而逐渐实现过滤的目的。这个过程，称之为目录的**预热**过程。

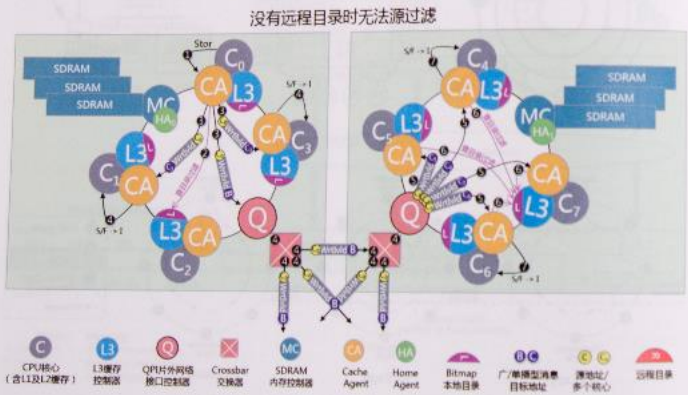


图6-167 没有远程过滤目录时的CC示意图（1）

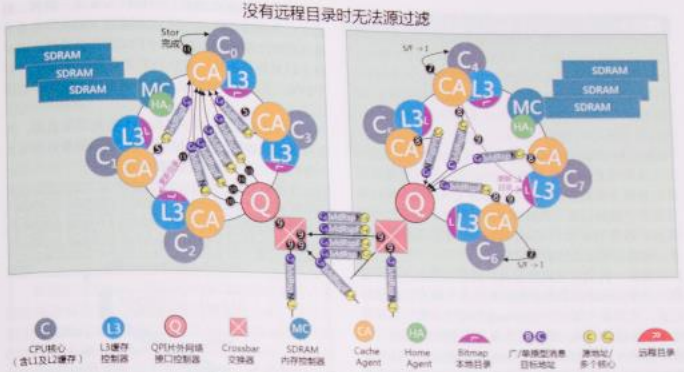


图6-168 没有远程过滤目录时的CC示意图（2）

那么，这个远程追踪目录需要存储什么信息呢？一个bitmap向量是必须的，bitmap中位的数量就是系统中所有远程核心的数量（本地核心就不需要在远程目录里追踪了，有本地LLC目录）。另外，由于追踪的是远程CPU内部可能缓存的行，对应的行可能在本地并没有缓存在LLC中，所以欲将bitmap存在本地LLC对应行中就不可能了。必须单独开辟一块高速存储器空间，存储对应行的地址，对应行的bitmap向量。如果要更加精细的记录，比如不仅仅要记录对应行是否存在于某个核心缓存里，还想知道对应行目前的MESIF状态，那就还必须记录一个状态字段，比如，只要收到对方的WrtIvld消息，那么本地就可以判断对方缓存中的该行一定处于M态了。

缓存行地址需要几十位，比如64位地址，缓存行大小64字节（需要6位来编址），那么缓存行地址长度就为64-6=58位，再加上3位的MESIF状态位，以及bitmap向量的16位（假设系统为4个4核心CPU），则这一条追踪记录就得至少78位大小。如果要追踪所有的缓存行，也就是2^{28}条记录，再乘以78位，其容量将会非常大。所以，该追踪目录势必要做出类似缓存和RAM之间无法全包含时一样的处理，比如直接相连方式，能省掉一些地址位，这样多条追踪记录会相互挤占同一个位置。还需要再次比对缓存行的物理地址Tag才能判断是否该记录有效，如果发现不匹配，则认为未过滤成功，对应的广播必须发出。

图6-169为一条记录项的示意图。思考一下，某个变量并不是任何时候都会被系统中所有核心缓存的。假设系统共有4个核心，那么，某个变量被缓存的场景共可分为：只被1个核心缓存（共4种组合），被2个核心缓存（共6种情况），被3个核心缓存（共4种情况），被全部4个核心缓存（共1种情况），总共15种状态。系统运行的时候，根据不同程序场景而定，在这15种情况中，出现比例最大的可能只有少数几种，我们假设为8种，分别为：被核心1缓存、被核心2缓存、被核心3缓存、被核心4缓存、被核心2/4缓存、被核心1/3缓存、被核心1/2/3缓存、被核心2/3/4缓存。那么，我们只需要存储3位就可以描述这8种状态。每次CA根据接收到的CC消息需要更新该条目时，电路先将当前存储的3位根据上述8种规定译码展开之后输入到一个4位逻辑电路，然后再把本次需要修改的位合入到这4位中，再通过收缩译码器将这4位译码成上述8种情况中的一种，如果本次修改的位恰好没有匹配上述8种场景中的任何一种，比如本次合入之后结果是“被核心3/4缓存”，我们称之为**溢出**。此时，译码器被设计为输出一个溢出信号，然后，电路可以决定，是将该条目直接作废（因为它已经无法反映真实情况了），还是动态在线地变更译码策略，后者这种做法更加智能，但是其需要该译码器是可配置的，可以根据寄存器中的控制字决定如何译码，比如在上述情况中，可以通过变更控制器寄存器内的控制位，让译码器改为可以将“被核心3/4缓存”（对应4位展开bitmap值0011）翻译成比如“010”，从而替代上述规定中的一种。下次如果又变化了，那就再替代一种。

Tag	Status	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b

图6-169 目录记录项的示意图

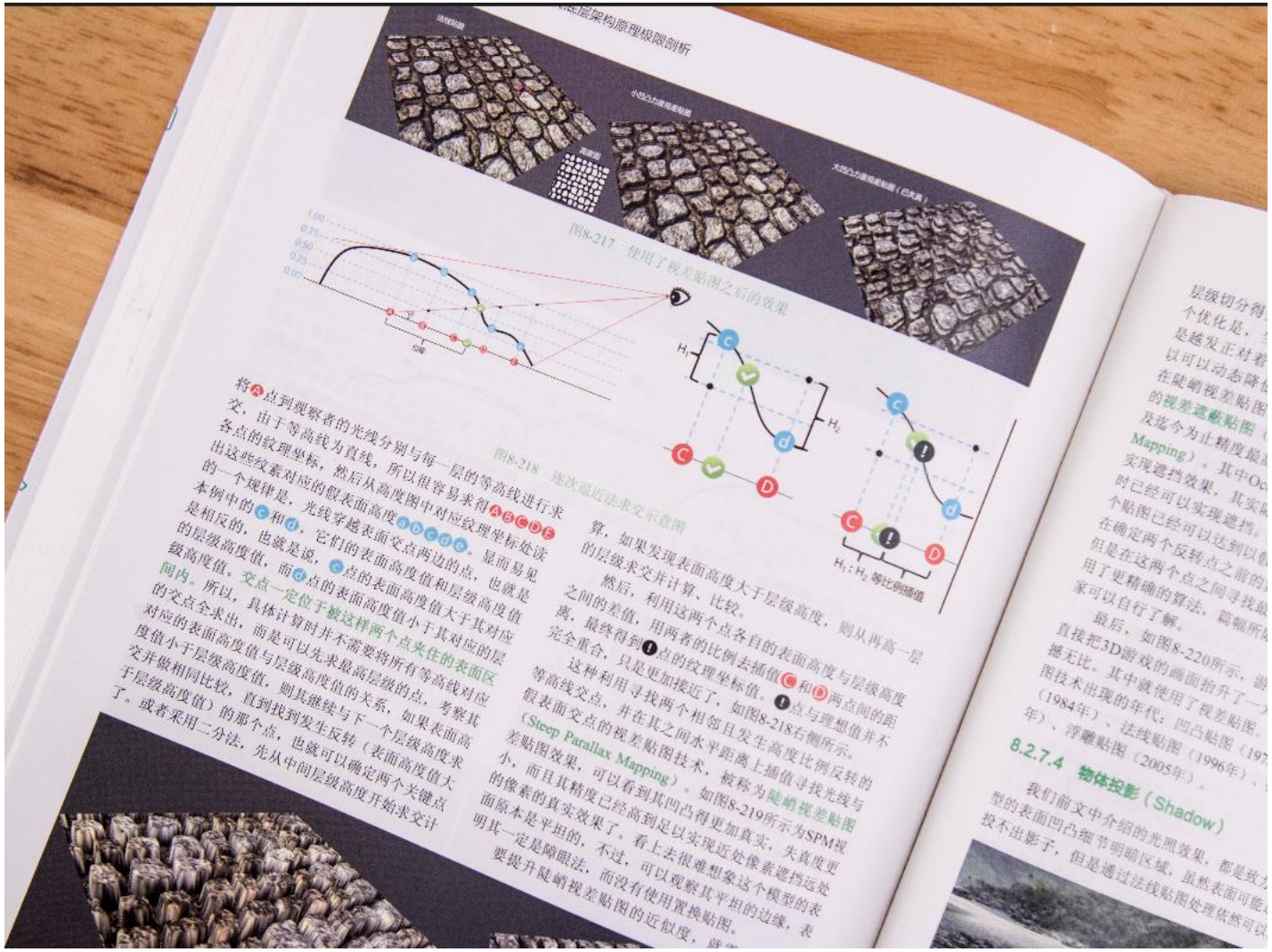


图8-217 使用了视差贴图之后的效果

图8-218 逐次逼近法求交示意图

将A点到观察者的光线分别与每一层的等高线进行求交，由于等高线为直线，所以很容易求得ABCDE各点的纹理坐标，然后从高度图中对应纹理坐标处读出这些纹素对应的假表面高度abcde。显而易见的一个规律是，光线穿越表面交点两边的点，也就是本例中的C和d，它们的表面高度值和层级高度值是相反的，也就是说，C点的表面高度值和层级高度值的层级高度值，而d点的表面高度值大于其对应的层级高度值。所以，交点一定位于被这样两个点夹住的表面区间内。所以，具体计算时并不需要将所有等高线对应的交点全求出，而是可以先求最高层级的点，考察其对应的表面高度值与层级高度值的关系，如果表面高度值小于层级高度值，则其继续与下一个层级高度求交并做相同比较，直到找到发生反转（表面高度值大于层级高度值）的那个点，也就可以确定两个关键点了。或者采用二分法，先从中间层级高度开始求交计算，如果发现表面高度大于层级高度，则从再高一层的层级求交并计算、比较。

然后，利用这两个点各自的表面高度与层级高度之间的差值，用两者的比例去插值C和D两点间的距离，最终得到I点的纹理坐标值。I点与理想值并不完全重合，只是更加接近了，如图8-218右侧所示。

这种利用寻找两个相邻且发生高度比例反转的等高线交点，并在其之间水平距离上插值寻找光线与假表面交点的视差贴图技术，被称为陡峭视差贴图（Steep Parallax Mapping）。如图8-219所示为SPM视差贴图效果，可以看到其凹凸得更加真实，失真度更小，而且其精度已经高到足以实现近处像素遮挡远处的像素的真实效果了。看上去很难想象这个模型的表面原本是平坦的，不过，可以观察其平坦的边缘，表明其一定是障眼法，而没有使用置换贴图。

要提升陡峭视差贴图的近似度，就

层级切分得
个优化是，
是越发正对着
以可以动态降
在陡峭视差贴图
的视差遮蔽贴图
及迄今为止精度最
Mapping）。其中Oc
实现遮挡效果，其实
时已经可以实现遮挡
个贴图已经可以达到以
在确定两个反转点之前的
但是在这两个点之间寻找
用了更精确的算法，篇幅所
家可以自行了解。

最后，如图8-220所示，
直接把3D游戏的画面抬升了一
搪无比。其中就使用了视差贴图
图技术出现的年代：凹凸贴图（
（1984年）、法线贴图（1996年）、
年）、浮雕贴图（2005年）。

8.2.7.4 物体投影（Shadow）

我们前文中介绍的光照效果，都是致
型的表面凹凸细节明暗区域，虽然表面可能
投不出影子，但是通过法线贴图处理依然可以

尾声 狂想计算机——以

地安 上的某个氨基酸残基上的电化学基团产生电动力，从
组成 而驱动蛋白质分子上的具有钾离子选择性的分子马达
A分子 的ATP结合点（油门）暴露，ATP一拥而上将其旋转起
来，从一侧吸引钾离子，转到另一面，释放钾离子。

但是模拟量底层并不是无限连续的，其最小单元
体内变 是一个正弦波形状。那么，一种可能的模型是：细胞
合点被封 一侧的钾离子浓度如果较高，那么该蛋白质分子在这
于某种原 一侧的某些残基上的能够与钾离子相互产生物理电作
导致马达 用或者化学力作用的基团每次受力都会对整个蛋白质
量，成为垃 分子深处的各个原子进行牵拉，当钾离子浓度较高的
结构、这么 时候，这种被牵拉的频率就会越高，当牵拉频率达到
自然生成的。 某个值的时候，引起分子的构象产生一个比较大的跃
的是靠进化自 变，将分子马达的ATP结合点暴露从而转运离子。当
规律在支配， 钾离子浓度不够的时候，阈值频率未达到，分子马达
部，必然会产生 停止工作。随着钾离子被运输到另一侧，本侧浓度持
规律已经是与数 续降低，最后形成一个闭环的负反馈控制系统。
灵曾经尝试用机 蛋白质分子就是利用原子来搭建的精密机械，
现的。分形理论也 是造物者的杰作。分子马达犹如热机一样，也分多
复的单元可以搭建 个冲程，比如结合ATP分子时，构象改变，导致蛋白
命。 质转子旋转一定角度，ATP变为ADP后，转子再次旋
系统的话，那么这 转一定角度，ADP被释放后，转子再次旋转。这就
细胞表面的蛋白质分 像热机的吸气、压缩、爆炸、排气
一次系统调用。而按照 这四个冲程不断循环一样。而蛋白
氨基酸化合成肽链，最 质分子马达纯粹依靠由ATP分子形成
质分子的过程，又好像 的分子间作用力来拉动整个转子旋
ary()从可执行文件中将程 转。扫描二维码观看蛋白质分子马
子生物学就是一场没有源 达的冲程动画。

一切全靠实验。 图5所示其实是一个ATP生成器。对于电动机来
处是，其几乎无所不能，既 讲，通电可以让转子转动，但是如果让转子转动，则
当离子通道，还可以作为在 可以反过来产生电流。对于蛋白质分子马达也是一样
的泵，如图4所示。比如钠钾 的，ATP生成器也是一个分子马达，其在细胞膜内的
时，其可以将细胞外的钾离子 区域是一个转子，转子表面包含多个质子结合点，可
钠离子运到胞外。那么，钠钾 以利用高酸性环境驱动转子转动，转子转动导致胞外
度差来的呢？ 部分构象形变，从而将ADP+Pi合成为ATP，这相当于
用电子计算机来模拟的话， 发电机。

每秒采样到的 对于图4左侧，诗意一些的表达则是：我要送你
然 一棵生命之树，象征着你我共同的目标。树根之下
这是能量的源泉。质子驱动着马达
力把ADP和Pi强行结合成
好吧，编


大话计算机
计算机
清华大学出版社


大话
计算机
冬瓜哥◎著
计算机系统
底层架构原理极限剖析
CONCEPT
清华大学出版社


冬瓜哥◎著
大话
计算机
计算机系统
底层架构原理极限剖析
清华大学出版社


冬瓜哥
著有《大话存储》系列


大话
计算机
Enjoy!
大话计算机
大话计算机
大话计算机
大话
计算机
计算机系统


Enjoy!
大话
计算机


大话计算机
大话计算机
大话计算机

还在看盗版？Out了！这书真的值得买纸张版！

绝对良心，这个作者绝对为了读者负责的角度去写作，就怕读者看着看着蒙了，解释的非常细致。书中内容真的是包罗万象，不仅仅是计算机。全篇透着作者对事物本质的追求和思索。

全彩印刷，作者用ppt做的图堪比autocad，世上绝无仅有的美图，将计算机运行原理动态展现出来，其他书中从没见过这种手法。盗版完全享受不到纸张版的质感。

这本书是一件艺术品，良心性价比。四五百块钱，拿出一两次玩网游吃大餐的钱就够了，可以说他价格高，但是千万别说贵。You can check out any time, but you will never leave.

三卷彩印精装书（正文共1506页）**+11张海报+典藏礼盒**。原价598元，折后价508.30元，券后**到手价458.30元**，这个品质这个价格，应该说很合适了。**预售入口**：长按下面二维码进入购买链接（请确保从下面入口进入购买京东自营，否则无法用代金卷，切记！另外，一些京东第三方店铺会出现低价，不要相信！）。 买买买买买买买买买买买买买！

仅限预售期有效，预售期过后本价格失效

大话计算机

大话存储

作者的两个微信公众号，里面全是宝贝！