Embedded Systems:
Using Open Source and StrongARM/XScale

毛德操　胡希明　著

嵌入式系统

采用公开源代码和 StrongARM / XScale 处理器

```
OS_ENTER_CRITICAL();
if (pevent->OSEventType != OS_EVENT_TYPE_Q) {
    OS_EXIT_CRITICAL();
    *err = OS_ERR_EVENT_TYPE;
    return ((void *)0);
}
pq = pevent->OSEventPtr;
if (pq->OSQEntries != 0) {
    msg = *pq->OSQOut++;
    pq->OSQEntries--;
    if (pq->OSQOut == pq->OSQEnd) {
        pq->OSQOut = pq->OSQStart;
    }
    OS_EXIT_CRITICAL();
    *err = OS_NO_ERR;
}else if (OSIntNesting > 0) {
    OS_EXIT_CRITICAL();
    *err = OS_ERR_PEND_ISR;
}else{
    OSTCBCur->OSTCBStat  |= OS_STAT_Q;
    OSTCBCur->OSTCBDly    = timeout;
    OSEventTaskWait(pevent);
    OS_EXIT_CRITICAL();
    OSSched();
    OS_ENTER_CRITICAL();
    if ((msg = OSTCBCur->OSTCBMsg) != (void *)0) {
```

浙江大學出版社

嵌入式系统

采用公开源代码和 StrongARM/XScale 处理器

毛德操　胡希明　　著

浙江大學出版社

图书在版编目(CIP)数据

嵌入式系统——采用公开源代码和 StrongARM/XScale 处理器/毛德操,胡希明著.—杭州:浙江大学出版社,2003.10
ISBN 7-308-03362-7

Ⅰ.嵌… Ⅱ.①毛…②胡… Ⅲ.①Linux 操作系统②微处理器,ARM—系统设计 Ⅳ.①TP316.89②TP332

中国版本图书馆 CIP 数据核字(2003)第 059157 号

本书限中国大陆地区发行

责任编辑:张 明
封面设计:俞亚彤
出版发行:浙江大学出版社
(杭州浙大路 38 号 邮政编码 310027)
(E-mail:zupress@mail.hz.zj.cn)
(网址:http://www.zjupress.com)
排 版:浙江大学出版社电脑排版中心
印 刷:杭州市长命印刷厂
开 本:889mm × 1194mm 1/16
印 张:45.75
字 数:1288 千
版 印 次:2003 年 10 月第 1 版 2003 年 10 月第 1 次印刷
印 数:0001—3000
书 号:ISBN 7-308-03362-7/TP·243
定 价:80.00 元

序

近来有一本书在中国的IT业界广为流传，这就是毛德操和胡希明先生所著的《Linux内核源代码情景分析》(上、下册)，它在短短的时间里印刷了四次，可见其受欢迎的程度。这一方面表明Linux在中国已日益受到重视，另一方面也表明该书写得很好，尤其是它采取了独特的情景分析方法，使读者对整个过程有比较完整、比较全面的了解，从而可以把各方面的知识有机地融会贯通、加深理解。当时，我在为它写的序言中曾经奢望“有更多像本书那样的优秀著作问世”，今天，令人高兴的是，毛先生等不辞辛劳，又为读者奉献了一本新书——《嵌入式系统——采用公开源代码和StrongARM/XScale处理器》。

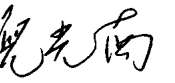
嵌入式系统是计算机系统的一个分支，既“古老”又“年轻”。说它古老，那是因为计算机发明伊始就在许多领域得到应用，那时就已经出现了后来所谓的嵌入式系统了。计算机用于某些实时控制或过程控制就是例子。说它年轻，则是说它在近年来蓬勃发展，如日中天。随着移动通讯、手持电脑、数字家电等等的发展以及IT技术在工业、交通、军事等领域里的渗透，嵌入式系统已经成为IT行业的热门。

一个嵌入式系统综合了多方面的技术，这对于从事这一领域工作的人提出了很高的要求。现在做这方面工作的人，有的来自软件领域，虽熟悉通用操作系统，但对嵌入式操作系统，特别是实时操作系统缺少深入的了解，对嵌入式系统开发所必需的硬件知识也很匮乏；有的人来自硬件领域，虽有较深的硬件背景，但对操作系统缺乏了解；也有的人来自控制或其他应用领域，对嵌入式系统的软件和硬件都缺乏系统和深入的了解。所有这些人都希望能有一本关于嵌入式系统的好书，来帮助他们解决工作中遇到的各种问题。而现存的资料却往往只针对某一具体的、商品化的嵌入式系统，多为介绍性的或使用手册之类，很少有对嵌入式系统从原理、技术到实际设计作全面、系统、深入的论述。正因为如此，本书的出版显得十分必要和及时。

在本书中，作者除了继续采用情景分析方法以外，还进一步采用了比较研究的方法。全书自成体系，深浅适度，论述有据，令人信服，给人以全新的感觉。

总之，我对本书的评价和对他们的前一本书一样，认为本书的内容在深度和广度两方面都是罕见的，相信读者看了以后也会得出同样的结论。

中国工程院　院士
中国科学院计算所 研究员



前　言

许多年以前，我们也曾人云亦云地说着“实时系统”，后来又是“嵌入式系统”（这个词出现得晚一些），可是那时我们并不真正知道这些词的含义。我们找资料，找书，可是却无奈地发现：似乎没有能使我们“一站式”地搞通个大概、而又理论联系实际的著作。现在，经过多年的努力与实践，我们终于“觉今是而昨非”，自以为对嵌入式系统有了较多的了解和较深的理解，可是却发现在这方面的好书仍旧不多。于是，我们就想把自己的理解和体会写出来，为正在从事嵌入式系统开发的同行以及有意于此的后学者提供一点参考。这就是作者撰写此书的初衷。

要说清楚嵌入式系统，特别是嵌入式操作系统，光用一个特定的系统作为实例是不够的。“有比较才能有鉴别”，所以我们在本书中采用了比较研究的方法。我们剖析、比较μC/OS 和 Linux 两个操作系统内核。前者可以说是最小的微内核，后者则是几乎无所不包的一体化（宏）内核。这是两个结构很不相同、各自起着标杆性作用的操作系统内核，然而又都成功地用于嵌入式系统。通过对比研究，一方面归纳、总结出嵌入式系统设计和实现过程中所必须面对的问题、难点及其解决办法，同时也归纳、总结出嵌入式操作系统与通用操作系统的共同点和（特别是）不同点。

虽然名曰“嵌入式系统”，本书的重点则放在软件，特别是放在操作系统上（读者看一下目录中所列的章节就清楚了）。之所以如此，是因为就工作量而言，嵌入式系统开发的重点在于软件，而其中操作系统内核又往往是难度最大的部分。但是，与通用操作系统相比，嵌入式操作系统与硬件的关系更为密切，因而必须结合相关的硬件平台。考虑到嵌入式系统主流的最新发展趋势，书中我们以 StrongARM 及其系统结构为硬件平台，与此相关的硬件知识则以最低必要为度。至于更为详细的知识，则只能请读者自行参阅相关的技术资料或产品使用说明。

总的来说，我们先以两章的篇幅对嵌入式（计算机）系统的硬件和软件两个方面作一综述性的介绍，重点在于说明嵌入式系统与通用计算机系统的不同之处，某种程度上也是对二者的比较研究。然后，第 3 章简要地介绍了 ARM 系统结构以及采用 ARM 核的 StrongARM/XScale 芯片系列，这是后面阅读有关代码、特别是汇编代码所必须的。在这三章的基础上，第 4 章着重介绍和分析 μC/OS 的代码。而第 5 章则着重介绍将 Linux 内核用于（基于 StrongARM 的）嵌入式系统时所特有的问题。如上所述，这两章是本书的核心。第 6 章进一步介绍了一些将 Linux 内核用于嵌入式系统时常用的设备驱动，读者可以把这一章的内容用作开发嵌入式系统的参考资料。最后，第 7 章和第 8 章着眼于嵌入式系统的开发与调试。我们的目的不在于告诉读者怎样开发和调试，而是想要让读者理解他们可能已经在用或者从其他资料上学到的一些手段是怎样实现的，以知其然并且知其所以然。本书可以供大学有关专业的高年级学生或研究生用作教材或参考读物，也可以供从事嵌入式系统开发的工程师用作参考资料。

在写作过程中，我们考虑到了非计算机专业出身的读者、甚至初学者的需要。事实上，有不少从非计算机专业乃至文科出身的人转入软件行业，成为了优秀的软件工程师并作出了突出的贡献，此种事例实非少见。应该说，转入软件行业比转入其他行业要容易一些，因为软件技术的基

础是逻辑，而人们在日常生活中时时在运用逻辑，因而最容易“无师自通”。但是，许多背景知识和概念、术语是他们必须了解的。为了照顾到这一部分读者的需要，书中对一些（我们自认为必要的）基本概念和术语作了一些铺垫和介绍。可是，这么一来，很大一部分基础较好、程度较高的读者（应该是本书读者群的主体）就可能感到有些多余或者啰嗦。对此，我们只好请这一部分读者多付出一点耐心，自己加以去粗存精。

另外，书中有些材料的取舍，也曾让作者感到为难。说得少了怕读者说我们语焉不详，甚至故意卖关子，说得多了又怕说我们拼凑篇幅，着意谋杀读者银子。事实上，书中有好几节原来已经写好却又撤了下来，反复考虑又加了回去。好在估计此书出版发行时定价可能还抵不上一条领带或一件衬衣的价钱吧。不过，就像《Linux 内核源代码情景分析》中一样，我们在本书中也基本上没有涉及计算机网络方面的内容。原因是网络这话题实在太大了，一旦“落网”就轻易脱不出来，这是个一定要有大部头的专著才能对付的话题。再说，网络方面的内容跟系统之是否是嵌入式也关系不大。

最使我们惶恐的还是书中一定还会有的错误。在《Linux 内核源代码情景分析》一书的前言中，我们就曾说过：“就像软件免不了有错一样，对软件的理解和诠释也一定会有错误，人们能做的只是尽量减少错误。我们可以负责地说，本书付印前在文字中已经没有我们知道而没有改正的错误，更没有故意误导读者的内容。但是，我们深知错误一定是有的，我们欢迎讨论，欢迎批评。”果然，该书出版后不久，就有位热心的读者向我们指出了一处错误，后来我们自己又发现了几处。对于本书，我们又一次面临同样的困境，知道一定有错，又不知道错在哪里。如果再拖上一年半载，再读上三遍五遍，想必还能再找出一些错误并加以改正。但是，什么时候才能肯定地说不再有错呢？只好先付印再说了。以后，我们也许会借浙江大学出版社网站为本书，也为《Linux 内核源代码情景分析》一书设一个勘误专栏。希望读者能说我们是严肃、认真的作者。如此足矣，夫复何求！

在美国生活了十几年后，本书的第一作者终于难抑乡愁回到了故国，目前任职于浙江大学网新科技股份有限公司。本书的第二作者则还在杭州恒生电子股份有限公司供职。在本书的写作过程中我们分别得到了两个公司的领导和同事的支持和鼓励，特此表示感谢。

还有，在本书的写作过程中，电子工业出版社曾经给予很多的关注和鼓励。作为作者，我们感谢电子工业出版社的关心和好意。

特别地，倪光南院士为《Linux 内核源代码情景分析》一书的第四次印刷写了代序，又为本书写序。这使我们很受鼓舞，特在此表示谢意。

目　录

第 1 章　嵌入式系统的硬件成分 1

§1.1　嵌入式系统 1
§1.2　CISC 系统结构与 RISC 系统结构 3
§1.3　微处理器与微控制器 11
§1.4　嵌入式系统的组成 14
§1.5　边界扫描测试技术 JTAG 19

第 2 章　嵌入式系统的软件成分 23

§2.1　嵌入式操作系统 23
§2.2　实时操作系统 27
§2.3　微内核与一体化内核 39
§2.4　常用的商品嵌入式操作系统 50
§2.5　一些公开源码的嵌入式操作系统 54
§2.6　嵌入式操作系统的量身定制 58

第 3 章　ARM、StrongARM、XScale 系统结构 61

§3.1　ARM 系统结构简史 61
§3.2　ARM 核的系统结构 63
§3.3　第一个 StrongARM 芯片 SA-110 78
§3.4　片上系统 SA-1110 80
§3.5　配套芯片 SA -1111 84
§3.6　几种典型的 StrongARM 系统 85
§3.7　Intel 的 XScale 系统结构 85

第 4 章　微内核 μC/OS-II 与硬件抽象层μHAL 89

§4.1　概述 89
§4.2　μC/OS 测试台的构筑 91
§4.3　ARM 处理器上的 μC/OS 96
§4.4　μC/OS 代码中的临界区 120
§4.5　μC/OS 的进程管理与调度 124
§4.6　μC/OS 的进程间通信 151
§4.7　扩充的μC/OS-II 进程间通信机制 172
§4.8　μC/OS 的系统调用 195
§4.9　μC/OS 的中断处理 198
§4.10　μC/OS-II 的设备驱动 233
§4.11　留给读者的思考 274

第 5 章 Linux 用于嵌入式系统.....275

§5.1 Linux 的各种“修补版”.....275
§5.2 Arm-Linux 的内存管理.....281
§5.3 高速缓存的锁定.....307
§5.4 Arm-linux 的进程管理与调度.....314
§5.5 Arm-Linux 的中断响应和处理.....334
§5.6 Arm-linux 的系统调用.....388
§5.7 Linux 内核的可剥夺进程调度.....398
§5.8 Linux 内核的优先级倒转问题及其解决.....408

第 6 章 Linux 对若干常用设备的驱动.....411

§6.1 概述.....411
§6.2 Ramdisk.....414
§6.3 闪存.....436
§6.4 “看门狗”与重启动.....487
§6.5 LCD 显示屏.....492
§6.6 触摸输入屏.....513
§6.7 DMA.....534
§6.8 FPGA 映像的装入.....551
§6.9 StrongARM 的电源管理.....559

第 7 章 嵌入式系统的引导与装入.....563

§7.1 关于引导/装入程序.....563
§7.2 一个基本的引导/装入程序——Blob.....565
§7.3 μC/OS 的引导与初始化.....603
§7.4 通过 JTAG 接口写入引导/装入程序.....626
§7.5 更复杂的引导/装入程序.....655

第 8 章 嵌入式软件的开发与调试.....657

§8.1 概述.....657
§8.2 嵌入式软件的调试.....659
§8.3 调试监控程序 Angel.....666

第1章

嵌入式系统的硬件成分

§1.1 嵌入式系统

所谓“嵌入式系统（embedded system）”，实际上是“嵌入式计算机系统”的简称。那么什么是嵌入式计算机系统呢？

通常，计算机连同一些常规的外设是作为独立的系统而存在，并非为某一方面的专门应用而存在的。例如一台PC机就是一个计算机系统，整个系统存在的目的就是为人们提供一台可编程、会计算、能处理数据的机器。你可以用它作为科学计算的工具，也可以用它作为企业管理的工具。所以，人们把这样的计算机系统称为“通用” 计算机系统。但是有些系统却不是这样。例如，医用的CT扫描仪也是一个系统，这里面也有计算机，但是这种计算机（或处理器）是作为某个专用系统中的一个部件而存在的，其本身的存在并非目的而只是手段。像这样“嵌入”到更大的、专用的系统中的计算机系统，我们就称之为“嵌入式计算机”、“嵌入式计算机系统”或“嵌入式系统”。从字面上讲，后者似乎比前者更为广义，因为系统中常常还包括一些机电、光电、热电或者电化的执行部件，但是实际上却往往不作严格的区分。在不致引起混淆的情况下，一般都把这三者用作同义词，并且一般总是指系统中的核心部分，即嵌入在系统中的计算机。

不过，虽然计算机在整个大系统中只是一个部件，却通常起着相当于“大脑”的作用。所以事实上所嵌入的计算机就是整个系统的核心，而系统中的其他部件则是其外部设备。只不过这些外部设备不同于常规的计算机外部设备，而所嵌入的计算机的作用和目的又只限于对这些外部设备的控制和管理而已。所以，也可以说，常规的计算机系统是面向计算（包括数值和非数值）和处理的，而嵌入式计算机则一般是面向控制的。

所谓将计算机“嵌入”到系统中，一般并不是指直接把一台通用计算机原封不动地安装到目标系统中，也不只是简单地把原有的机壳拆掉并安装到目标系统的机壳中，而是指为目标系统构筑起合适的计算机系统，再把它有机地植入，甚至融入目标系统。

实际上，人们还常常从更广泛的角度谈及“嵌入式”。最明显的例子就是“手持式计算机”和“掌上计算机”，明明是面向计算的，却常常被看成是“嵌入式”的。究其原因，一方面，可能是因为这些计算机往往是作为特殊的、专用的设备，即PDA（个人数据管理器）或PCS（个人通信系统）而开发、

而存在的；并且其输入/输出手段与常规的计算机系统有所不同。另一方面，也确实有直接把常规的计算机嵌入到一些大型系统中的，例如当初 AT&T 开发的计算机 3B2 是一种常规的计算机，但是其开发的意图却是用在电话系统中，因而也可以看作是嵌入式计算机。至于把 PC 机用在工业控制环境中，自然也不在少数。由此可见，计算机是否属嵌入式，其实也并没有一条明确的分界线。

嵌入式系统既然是计算机系统，就不可避免地必须由三大部分构成，那就是“中央处理器 CPU”、内存以及输入/输出手段。此外，当然还得有将这三大部分连接起来的“总线”。这是所有计算机系统的共性，是万变不离其宗的。但是，尽管如此，使用目的的特殊性常常会导致性质和结构上的特殊性，嵌入式系统通常有下述一些特点，不过这些特点中的任何一点都不是必须的，所以并不成为嵌入式系统的判定条件。另一方面，我们当然也无法穷举嵌入式系统所有的特点。

可靠性与稳定性对于嵌入式系统有着特别重要的意义。所以即使逻辑上的系统结构相同，在物理组成上也会有所不同。同时，对使用的元器件包括接插件、电源等等的质量和可靠性要求都比较高，所以元器件的“平均无故障时间 MTBF（Mean Time Between Failure）”成为关键性的参数。此外允许的环境温度也是个需要重点考虑的问题。所以，即使同样是采用 PC 机的系统结构，在嵌入式系统中一般都采用 PC“工控机”，而不是一般的 PC 机母板，因为二者在选用元器件、制造工艺、质量控制等方面都（应该）有不同的要求。军用设备的要求就更高了。

嵌入式系统常常还有减小功耗的要求。这一方面是为了省电，因为嵌入式系统往往以电池供电；另一方面是要减少发热量，因为嵌入式系统中常常没有风扇等排热手段。

还有，嵌入式系统对物理尺寸也常常有特殊的要求。

嵌入式系统常常是在无人照看的条件下运行的，有的甚至是在人迹罕至的地方运行。这就产生了一个问题，那就是万一系统中的程序运行陷入了混乱，例如死循环或者死锁，使整个系统“死”了怎么办？在采用 Windows 操作系统的 PC 机上，此时能做的恐怕就是按机器上的“Reset”键重新引导，或关了电源再开（我们的经验是平均数天一次）。可是在这无人的环境中怎么办？显然，需要在无人照看条件下运行的嵌入式系统需要有能够检测到这种状况，并采取对策的手段。为此，嵌入式系统中通常都采用一种称为“看门狗（Watch-dog）”的机制，这是一般通用计算机中所没有的。

还有些嵌入式系统需要长期连续运行（如电话交换机）。在这些系统长期的运行中，如果某个部件损坏，是不允许将系统关闭断电然后加以替换的，而必须在系统仍旧加电、继续在运行的条件下加以替换。这就是所谓“热替换（Hot Swapping）”或“热插拔”，即在系统还在运行的条件下将电路板插入系统的总线，或者从总线上拔下电路板。不过，这一般是指整个模块或电路板的替换，而不是指个别元器件的替换。还有一种类似但稍为简单一些的要求是在系统还在运行的条件下增加一个模块，称为“热插入”。无论是热替换还是热插入，都要求硬件和软件两方面的配合，后面我们还将提到这个问题。

进一步，有些要求高可靠的嵌入式系统还需要采用“容错（Fault Tolerance）”技术，就是系统在部件损坏时能自动切换到它的备份，或者对系统进行重构。最常见、也是最简单的容错技术是电源的自动切换。一些嵌入式系统往往带有备份电源，一旦检测到电源故障就自动切换到备份电源并发出警报。还有一种很常见的容错技术是一些数据通信设备中对链路的“自动保护切换（APS）”。在电信系统中，最可能发生故障的“部件”就是物理的链路（例如两个城市之间的电缆），所以实践中常常以一定的比例为这些电缆配上备份，例如每三条平行的链路就配上一条作为备份。而在两端的设备中，则各有相应的设施，使得监测到某条链路发生故障时，就很快切换到备份链路继续运行。至于系统中其他部件的损坏，就比较复杂了。一般而言，容错技术是一个需要有专著加以论述的话题，我们在本书中将不作深入探讨。

如前所述，嵌入式系统的主要目的是控制，而控制对象的精度要求一般不会超出整型数所能提供的分辨率（例如 16 位整数的分辨率为 1/64 K），所以浮点运算一般没有必要，许多嵌入式系统都不带浮点运算部件。

一方面，需要由嵌入式系统提供的功能以及面对的应用和过程都是预知的、相对固定的，而不像通用计算机那样有很大的随意性。既然是专用的系统，在可编程方面就不需要那么灵活了。另一方面，一般也不会用嵌入式系统作为开发应用软件的环境，在嵌入式系统上通常也不会运行一些大型的软件，例如编译器或复杂的数据库等。所以，一般而言，嵌入式系统对 CPU 计算能力的要求并不像通用计算机那么高。

同时，考虑到功耗、体积、价格等等因素，嵌入式系统的内存通常也比较小。

许多嵌入式系统都有实时要求，需要有对外部事件迅速作出反应的能力。以前，“嵌入式系统”几乎是“实时系统”的代名词，近年来出现了许多不带实时要求的嵌入式系统，这两个词的区别才又变得显著起来。但是，即使是现在，多数嵌入式系统还是有着不同程度的实时要求。

在系统的组成上，因为嵌入式系统常常是用于控制目的，其外设接口自然就比较多样，并且数量也往往比较多。

不过，嵌入式系统一般都不带用于大容量存储目的的外部设备，也就是不带磁盘。而操作系统的映像和可执行程序一般都存放在只读存储器（ROM）或“闪存”（Flash Memory）中。

许多嵌入式系统的人机界面也有其特殊性。许多嵌入式系统都不提供图形人机界面，而只是提供一个面向字符的控制台接口。不过往往还带有如小型的 LCD 显示屏、发光二极管（LED）等辅助的显示设备，甚至报警装置。

嵌入式系统中的 CPU 应当尽量适应上述的众多特点，所以通常采用一些不同于“微处理器”的“微控制器”，而且往往是 RISC 结构的。这些微控制器通常比较小，比较省电，价格也比较低。另外，为了减小体积与价格，增加集成密度（从而提高可靠性），这些微控制器往往与一些外设接口集成在同一块芯片上，甚至成为单片机或“片上系统（System on Chip）”，即 SoC。

正如“经济基础决定上层建筑”一样，嵌入式系统可能具有的这种种不同，势必要在其软件、特别是在操作系统中有所反映，从而使嵌入式软件的开发与常规软件的开发有显著的区别。特别地，典型的嵌入式操作系统与常规的操作系统有着显著的区别，并因而成为操作系统的一个重要分支和一个独特的研究方向。不过，这里也要说明，并不是所有的嵌入式系统都需要有真正意义上的操作系统，有的嵌入式系统所采用的其实只是个简单的监控程序。这方面的一个例子是 IC 卡(也叫“智能卡”)上的“片上操作系统(Chip Operating System)”COS，那实际上就只是个监控程序，而 IC 卡则应该划入嵌入式系统一类。对于什么样的软件可以称为操作系统并没有明确的大家都能接受的判定标准，我们的倾向是：如果系统中没有引入进程的概念，又很简单，我们就不认为是个操作系统。当初甚至有人质疑说 DOS 不能算是操作系统，也是因为 DOS 中没有进程的概念（不过它并不简单）。

§1.2　CISC 系统结构与 RISC 系统结构

前面提到，嵌入式系统中的 CPU 往往是 RISC 结构的。所谓 RISC 是指“简约指令系统”，也称“简约指令集”，以及相应的 CPU 系统结构。这是与 CISC，即传统的“复杂指令集”相对立的概念与系统结构。

早期微处理器的系统结构基本上是与当时的“小型机”一脉相承的。所谓“系统结构”是指程序员在为特定处理器编制程序时所“看到”从而可以在程序中使用的资源及其相互间的关系，其中最重要的就是处理器所提供的指令系统与寄存器组。早期计算机的 CPU，包括早期微处理器的指令系统全都是 CISC 的。

一个处理器的系统结构就是它的逻辑抽象。至于这个抽象的处理器具体是怎样实现的，则称为其（硬件）组成，或者就称为“实现”。然而，同一个系统结构可以有不同的实现。例如相同的指令系统既可以通过“硬连接”的方法实现，也可以通过“微程序”的方法实现。所以一个处理器的系统结构与其具体实现是不同层次上的概念，原则上是相互独立的。但是，实际上这二者是相互影响的，甚至有着相当重要的关系。

特别地，微程序的采用对指令系统的设计有着比较密切的关系和影响。

一方面，微程序的采用使一些比较复杂的指令成为可能或可行，或者至少使其实现大大简化。例如，Motorola 的 M68030 处理器中有一条指令 CAS2，用来将一个数据结构链入一个双向链表。由于整个链入的过程仅由一条指令完成，在此过程中就不会发生中断。读者不妨试着用 C 语言写出实现这个过程所需的代码，注意这个过程是不容许中断的。我们知道，C 语言是比较低级的语言，但毕竟还是高级语言。可是在这里，机器语言和汇编语言反倒比 C 语言还高级了，因为这里的一条汇编指令就相当于好几条 C 语句（当然，并非每条指令都是如此）。这么复杂的功能，要放在一条指令内完成，如果采用硬连接而不是微程序的方法来实现，则其电路的复杂程度以及实现整个处理器的难度，真是有点难以想像了。可是，采用了微程序技术情况就不同了。

另一方面，在采用微程序技术的处理器中，每条指令的执行都通过一个“微指令”序列，即一段微程序来完成，就好像调用了一个子程序一样。每条指令都对应着一段特定的微程序（不妨看作“微函数”），指令的操作码决定了具体微程序的入口，所以起着相当于函数名的作用。所有的微程序都存放在一个只读存储器（ROM）中，就好像一个程序库，而这个 ROM 就是处理器的一个组成部分。用于微程序的 ROM 是高速的半导体存储器，比一般的内存快很多，所以从微程序 ROM 取微指令也要比从内存取指令快很多。

与指令相比，微指令有几个特点：

1. 每条微指令所代表的都是很简单的基本操作。指令则不一定，有的指令也很简单，有的却比较复杂，甚至非常复杂。
2. 所有微指令的格式都很规则、很简单，因而易于译码。相比之下，指令的格式往往比较多样，从而不够规则，也不够简单，增加了译码的难度。
3. 取微指令的速度很快，而取指令的速度则较慢（因为来自速度较慢的内存）。
4. 微指令的执行速度很快，而指令的执行速度较慢（因为一条指令要通过若干条微指令的序列来完成）。

也就是说，在采用微程序技术的处理器中，实际上有着两种不同层次的指令。一种是面向程序员（软件）的、高层的“指令”；另一种是面向硬件实现的、低层的“微指令”。这样，只要由电子元件和线路实现一套基本的功能，就可以通过微程序对这些功能加以各种不同的组合和编排，以实现高层指令所要求的复杂功能。而高层的“指令系统”，则相当于由函数库向用户提供的“API”（应用程序界面）。所以，微程序的采用实际上是对软件设计中“子程序调用”这个概念的推广，将其适用的范围向

下推进了一个层次，从而对所要求的操作进一步化整为零，分而治之（即所谓的 Divide and Conquer），使指令系统的实现得到简化。

另一方面，正因为如此，微程序的采用又反过来鼓励、助长了在指令系统中采用复杂指令的倾向。人们曾经普遍认为：既然微程序的采用使复杂指令的实现成为可能和可行，处理器就应该为一些典型的复杂操作（如双链表操作）和典型的高级语言成分提供相应的指令。这样执行效率应该比较高，因为同一功能如果只用一条指令实现就只需取一次指令，对指令进行一次解码，而如果用两条指令实现就要取两次指令，等等。这样一来，程序所需要的存储空间也可以随之减小。更重要的是，高级语言的编译过程也可随之简化。表面上看来，这样的观点似乎是毋庸置疑的，因而在当时得到了广泛的认同。实际上，在小型机时代采用微程序的实现方法已经成为主流。当时的 DEC 公司甚至在其 PDP11-60 小型机中，提供由用户自行定义和增添微指令的手段。如果用户经常要用到某种计算或功能，例如计算“卷积”，就可以自己编写一段相应的微程序，并在调试以后将其写入 CPU 中的微程序 ROM，这样就为 CPU 增添了一条新指令，以后只要在有关的汇编语言程序中直接使用这条指令就可以了。

指令的功能与复杂程度还跟所允许的（对操作数的）寻址方式有关。如“间接寻址”实际上就是实现了“指针”的概念，而“索引寻址”、“变址索引寻址”等则对数组元素和结构内部元素的访问提供了方便，这都是与高级语言中的相应成分很接近的。我们看下面的 C 代码：

```
k = array[m];
n = k + 20;
```

这里的第一条语句实际上就是对内存的变址索引寻址，第二条语句则是算术运算。可是，如果这里的变量 k 不是别有用处，那么恐怕谁都会写成这样：

```
n = array[m] + 20;
```

即使不写成这样，C 编译的优化模块也会自动来做这件事，就是把对内存的访问结合到算术运算中，或者说为算术运算的操作数提供灵活的寻址。事实上，看一下 x86 的指令系统就可以明白，它的算术运算指令都支持对内存的各种寻址方式，其操作数可以通过各种寻址方式直接来自内存，而不必先把它装入到某个寄存器中。

凡此种种，自然就使人们得到一个印象，即指令系统的设计应该向高级语言看齐和靠拢，其“最高境界”就是与某种高级语言相同，即使因条件限制而暂时办不到，也是“虽不能至，心向往之”。所以，当时很有些人醉心于研究和设计“直接执行”高级语言的处理器，例如直接执行 FORTRAN 语言程序的“FORTRAN 机”等等。这种处理器中的指令就相当于某种高级语言中的语句，因此原则上就不需要编译而可以直接执行了。在这样的倾向下，各种处理器的指令系统变得愈来愈庞大，而一些指令（除必要的基本指令外）则又愈来愈复杂。于是，CPU 指令的功能愈来愈强，包括的寻址方式也愈来愈多、愈来愈灵活。

微处理器技术的发展当然也是“站在前人的肩膀上”，因而很自然地把采用微程序的实现方法连同上述的倾向也继承了下来，这自然要影响到各种微处理器指令系统的设计，使一些微处理器中也提供了一些功能很强的复杂指令。事实上，这些复杂指令往往成为厂家相互争奇斗艳、吸引用户的一种手段。前面我们提到了 M68030 中的 CAS2 指令，其实别的处理器又何尝不是如此。就拿 Intel 的 80386

来说，我们只举一个简单的例子：在 MOVE 指令的基础上，Intel 提供了“成串”MOVE 指令。这样，对于内存中按字节的成块复制就可以使用 MOVESB 指令（加上前缀 REP）完成，而这一条指令就相当于下面这一行 C 语句：

```
while (n--) *dest++ = *src++;
```

这对于数据结构的复制（赋值）应该是很有好处的。

然而，随着技术的发展，人们对此发生了怀疑。这些怀疑并非凭空而生，而是来自实践，有着坚实的基础，因而不容忽视，主要有下述几个方面。

第一，为了提高运算速度，在微处理器中采用了“预取指令”等“流水线操作”技术。例如，要是所有指令都分四步（取指令、取操作数、运算、存放结果），在四个时钟周期中完成，那就可以在处理器中建立起一条“四工位”的流水线，使得当某条指令的执行已经接近完成，正在把结果写回内存的时候，其下一条指令已经进入运算阶段，再下一条指令已经进入取操作数阶段，更下一条指令则已进入取指令阶段。这样，假定每个阶段都只需一个时钟周期就能完成，则虽然每条指令都需要四个时钟周期才能完成，但是在理想条件下总体的“吞吐量”却可以达到每个时钟周期一条指令。之所以说“理想条件下”，是因为流水线，特别是较长的流水线，常常不能被有效地充满，每当遇到转移指令时就会“断流”。但是，尽管如此，对于一般的“单指令流/单数据流”处理器而言，除半导体技术本身的发展以外，流水线就是提高总体运算速度的主要手段了。可是，恰恰在这一点上，复杂指令的存在引起了问题。首先，复杂指令的存在使各种指令的执行时间（以时钟脉冲的个数计）长短不一，有的可以在四五个时钟周期中完成，有的却需要几十个。而且，即使是操作简单的指令，由于不同寻址方式的存在，其执行时间也可以长短不一。例如，间接寻址就要比直接寻址多访问一次内存，从而至少要多一个节拍，更不用说其他更为复杂的寻址方式了。更何况指令本身的长度也不一致，有的是 4 字节指令，有的是 8 字节指令，还有的是 12 字节指令，甚至同一指令的长度也可以因采用的寻址方式的不同而变化。在 32 位的系统结构中，指令长度为 8 字节就说明在取指令阶段要访问内存（或高速缓存）两次，因而至少需要两个时钟周期，12 字节则至少需要三个。在这样的情况下，怎么设计流水线的长度呢？当然，我们可以按照短的指令设计流水线，让它每当遇到复杂指令或复杂寻址方式的时候就暂停流动。我们也可以按最长的指令设计流水线，但是使得较短的指令在执行中跳过若干工位。可是，这样会使流水线，从而整个处理器的设计与实现复杂化。而且，由于这些复杂化，在实现中往往不得不牺牲简单指令的“利益”，例如，因为要顾及某些比较复杂的过程而只好略为降低时钟频率。

第二，即使不考虑流水线，在实现中也往往会因为要顾及一些特殊的复杂指令的实现，而只好让简单指令作出一些牺牲。问题在于，这样做从总体和全局看是否划算？这就取决于这些指令的（相对）使用频度。然而，调查和研究表明，多数复杂指令都很少用到，甚至从未用到，尤其是在应用软件中。原因在于，应用软件都是用高级语言编写的，如果相应编译程序中的代码生成模块不采用这些特殊的复杂指令，这些指令就不会被用到。而编译程序为了易于移植和保持版本间的兼容，又常常倾向于不采用这些特殊的指令。即使在采用汇编语言编写的软件中，例如操作系统内核和一些库程序中，使用这些特殊指令的频率也不高，这些指令对全局的贡献一般不超过几个百分点。反之，如果不提供这些特殊的复杂指令，则处理器的设计和实现可以简化，要是因此而能将时钟频率提高几个百分点（事实上这是完全可能的），那就可以得到弥补。

第三，微处理器的集成规模是受半导体技术以及生产成本限制的，而微程序 ROM 又是微处理器内部“占地”最大的部分，复杂指令又是其中的“大户”。如果能削减微程序 ROM 的大小，则或者可以在同样大小的芯片中集成其他的功能部件，或者可以减小芯片的尺寸和耗电，或者二者兼得。这二者对于嵌入式系统有着特殊的意义，因为用于嵌入式系统的处理器/控制器，常常要求将一些外围模块（如外设接口等等）集成在同一芯片中。而要削减微程序 ROM 的大小，从指令系统中砍掉复杂指令当然是首选。进一步还可以设想，最好是根本不用微程序。但是，如果要以硬连接来实现比较复杂的操作，那又南辕北辙了，硬连接只能用来实现一些最基本的操作。那么，能不能既不用微程序，又不必以硬连接来实现比较复杂的操作呢？

第四，现代的微处理器都带有较大的高速缓存，运行中访问内存时命中高速缓存的概率可以达到较高的程度。命中时，高速缓存的访问速度与微程序 ROM 相似。在这样的条件下，使用微程序 ROM 还有必要吗？为什么不像 inline 函数或宏操作那样，预先把指令展开成微程序，直接存储在内存中呢？诚然，这样会占用更多的内存空间（类似于 C 语言中函数调用与宏操作展开之间的区别），但是现在内存已经很便宜了。那么，能否将作为汇编语言程序设计 API“指令”的含意向下推一层，推到原先的微指令，或者很接近于微指令这一层呢？换言之，能不能把指令的“粒度”做成像微指令那样呢？

于是，“简约指令集计算机”，即 RISC（Reduced Instruction Set Computer）的概念与技术便应运而生。这方面的研究最初由 IBM 公司开始，后来加州大学伯克利分校以及斯坦福大学的研究人员也基本上各自独立地进行了研究（其中 IBM 的 John Cocke 因为在这方面的贡献而荣获 1987 年的图灵奖），并各自取得丰硕的成果。由于这些研究是各自独立进行的，他们所研发的 RISC 系统结构也各具特色，这些成果后来分别发展成为 PowerPC、SPARC，以及 MIPS 等几种最主要的 RISC 系统结构。与 RISC 相对，传统的系统结构便成为“复杂指令集计算机”，即 CISC（Complex Instruction Set Computer）。

本书并非关于 RISC 的专著，更不是关于某一种特定 RISC 系统结构的专著，所以只能在这里介绍一些 RISC 的基本知识。不过，由于本书所专注的 StrongARM 处理器也是一种 RISC 芯片，所以也确有必要加以介绍。

一般而言，RISC 系统结构都有如下一些共性，但是其中有些是特定 RISC 系统结构所专有的特色，我们会在叙述中予以说明。

1．RISC 系统结构中的指令系统都比较小，即不同指令的数量较少，并且只提供简单指令。这些指令的执行都能在四五个时钟周期中完成。所谓“简约指令集”，一方面是说指令集的大小，另一方面是说每条指令的复杂程度，两个条件缺一不可。

2．每条运算指令的操作数都必须预先存于寄存器中，也就是说，一般运算指令在执行过程中是不访问内存的，所有的操作数不是立即数就是来自某个寄存器，而执行的结果也只能存入某个寄存器。为此，就需要配备专门的访内指令，主要就是两条，即 LOAD 和 STORE（或者其他类似的名称），前者用来从内存（或高速缓存）装入数据到某个寄存器，后者与此相反。此外，如果有的话，堆栈操作指令 PUSH/POP 也要访问内存。同时，即使对于专门用来访问内存的 LOAD 和 STORE 指令，也只允许使用简单的寻址方式，目标地址必须预先装入某个寄存器中。这样，所有指令的执行长度就变得均匀划一，而寻址方式的实现也变得既简单又整齐。

3．不光是指令的执行长度和寻址方式整齐划一，连指令的格式也很整齐划一，就像微指令一样。一般在典型的 32 位 RISC 系统结构中，每条指令的长度都是 32 位，包括一个操作码位段和三个操作数

位段（两个源操作数和一个目标操作数），每个操作数位段都指明一个寄存器。显然，这种整齐划一的指令格式有利于简化指令系统的实现。由于每条指令的长度都是 32 位，其执行长度（时钟脉冲个数）又是固定的，采用了流水线以后就可以基本上达到每个时钟脉冲执行一条指令的目标。因此，“每个时钟脉冲一条指令”成了 RISC 结构拥护者的旗号和目标。

4. 由于一般指令的操作数都必须事先存在于寄存器中，计算过程中的中间结果自然就不应该存放在内存中，而应该也保存在寄存器中。以 C 程序为例，则中间结果不应该保存在普通的局部变量或全局变量中，而应该保存在说明为“register”模式的局部变量中。这样，就需要使用较多的通用寄存器。可是传统的（CISC）系统结构中寄存器的数量一般不超过 16 个（包括堆栈指针 sp 和取指令指针 pc 在内，但不包括用于内存管理等的寄存器），其中可以分配用于此种目的者只有很少几个，显然不适应 RISC 系统结构的要求。所以，RISC 系统结构中一般都有比较多的寄存器组，通常是 32 个寄存器。研究与实验表明，当寄存器的数量从 16 个上升为 32 个时，所取得的效果十分明显，可以说是质的变化，但是进一步再上升时效果就不太明显了。所以，一般认为 32 个是最佳选择。由于这个变化，高级语言的编译程序需要把几乎所有的局部变量都当成是“register”模式的，努力为这些变量分配/调度使用寄存器。因此（以及下述的其他原因），RISC 技术需要编译技术的配合。另外，为了便于寄存器的分配/调度，RISC 系统结构中的寄存器基本上都是均匀一致的通用寄存器，而不像在 CISC 系统结构中那样大多各有特殊的用途（如“变址寄存器”等等）。有的时候，寄存器可能会分配不过来，这时候就需要调度，将一部分可以暂时不用的寄存器内容转移到内存中，称为“溅出（spill）”，待以后需要时再恢复。

5. RISC 的子程序调用过程，或曰“子程序链接”，与 CISC 的也不同。在 CISC 系统结构中，CALL 指令（或其他同类指令）将返回地址存入内存中的堆栈，因而需要访问内存。而 RISC 系统结构的原则之一是尽量少访问内存，所以返回地址是放在寄存器中而不是堆栈中。为此，一般专门拿一个寄存器作为“（子程序） 链接寄存器”。这样，如果所调用的是底层的“叶”子程序，便不需要为子程序链接而访问内存。而如果是中层的子程序，则仍可按一定的调用约定将此寄存器的内容溅出到堆栈中。此外，调用参数的传递也不是通过堆栈，而是通过寄存器。对于有大量子程序调用，特别是有大量很小的“叶”子程序调用的软件，这样的链接方法可以节约许多访问内存的开销，有利于提高执行效率。

为了减少寄存器内容的溅出，在 SPARC 的系统结构中还采用了一种称为“寄存器窗口”的技术。在这种系统结构中有数百个寄存器，但是 CPU 在同一时间内仍只能使用其中的 32 个（限于 5 位的操作数位段），称为一个寄存器窗口。每层子程序都使用自己的窗口，调用下一层子程序时，或者从子程序返回时，便切换一个窗口，就好像“旋转舞台”一样。不过，相邻窗口之间都有几个寄存器重叠，用来传递参数与返回结果。在每个寄存器窗口中，都将一部分寄存器用于传入调用参数，称为“输入部（in）”；另一部分用于传出调用参数或返回结果，称为“输出部（out）”；其余的（大部分）才用于计算的中间结果及临时变量。当“舞台”正向旋转（调用子程序），从而启用一个新的窗口时，后一个窗口的输入部与前一个窗口的输出部相重合，所以这一部分的内容不变。反向旋转（返回），从而释放一个窗口时则类推。这样，只要调用的路径不太长，就不会为了传递参数而不得不溅出寄存器的内容。当然，如果调用路径过长（层次过深），以致所有的窗口都已被占用，再旋转下去就要发生冲突时，就只好溅出一个或几个窗口了。此时 CPU 会产生一次异常（trap），由异常服务程序溅出若干窗口。当然，所有这些都可以由编译程序和操作系统内核安排解决，对于使用高级语言的程序员是透明的。读者也许已经意识到，这其实是页式存储管理中页面交换的推广。不过，“RISC 圈内”对此项技术的作用还是有争议的，SPARC 的拥护者当然喝彩，而 MIPS 的拥护者却争论说只要编译程序聪明一些，对寄存

器的分配/调度得当，就可以达到相当接近的效果，所以由此而来的（硬件和系统结构上的）复杂化其实是得不偿失。

6．中断的过程可以看作是特殊的子程序链接。传统的方法是在进入中断响应之初便把（几乎）所有寄存器的内容都存入堆栈，到从中断返回时再予以恢复。Linux 源码中的 SAVE 和 RESTORE 就是典型的保存现场和恢复现场操作。可是，对于有 32 个寄存器的 RISC 系统结构，这显然是不合适的。所以要对中断时的寄存器使用作个约定，哪些寄存器的内容是必须保存，因而在中断处理程序中是可以自由使用的。如果在中断处理程序中要使用更多的寄存器，便必须先行一步溅出这些寄存器的内容。所以，有些 RISC 处理器中把中断分成两种，一种是“轻量级”的，通过类似于旋转舞台的机制保存和恢复现场；另一种是“重量级”的，那就是常规的中断。

在实现上，如前所述，RISC 处理器都采用流水线，都带有高速缓存，一般都不采用微程序。由于指令系统的简约，RISC 处理器的实现也远较 CISC 处理器简单。

另外，RISC 系统结构中流水线的采用为程序设计也带来一些特点。虽然这些特点只有在汇编程序这一级上才能看到，而对于使用高级语言的程序员是透明的；但是对系统软件的开发与调试是常常需要深入到汇编程序这一级的。

首先是关于 LOAD 指令。按传统的程序设计思路，如果通过一条指令从内存将一数据读入一个寄存器，那么紧接着的指令就能使用这个寄存器的新内容。可是，在流水线中相继两条指令的执行只相差一个节拍（时钟脉冲），所以当后面的指令要使用这个数据时寄存器中的内容还是老的，新的内容尚未到达。一般新的内容要再过一个脉冲才能到达，所以在这当中有个一条指令的“空当”，称为“装入空当（Load Delay Slot）”，在这个空当中不能使用正在装入的数据，而需要放上一条与此数据无关的指令，如果没有这样的指令就放上一条空操作（NOP）指令。所以，编译程序在生成目标代码时要对此进行“指令调度”。例如：

```
LOAD    data1,   r2      ; 将变量data1的内容从内存装入r2
LOAD    data2,   r3
ADD     r2,  r3,  r4     ;(r2 + r3) => r4
LOAD    data3,   r5
LOAD    data4,   r6
ADD     r5,  r6,  r7     ; (r5 + r6) => r7
```

这是一段假想的汇编代码，其语意不言自明。可是，这里的两条 ADD 指令都有问题，因为 data2 和 data4 都还不能到位，这就是 LOAD 指令后面的“装入空当”。经过编译程序的指令调度，调整了指令的次序以后，就会变成这样：

```
LOAD    data1,   r2
LOAD    data2,   r3
LOAD    data3,   r5
LOAD    data4,   r6
ADD     r2,  r3,  r4
```